Tarea 1

Nombre: Franco Lopez Angel Sebastian Grupo: M5

Definicion del paradigma orientado a objetos

La programación orientada a objetos (OOP) es un paradigma de programación que modela los datos y el comportamiento de los objetos del mundo real. Un objeto es una entidad que tiene estado y comportamiento. El estado de un objeto es representado por sus atributos, mientras que el comportamiento de un objeto es representado por sus métodos.

OOP es un paradigma poderoso que permite a los programadores crear programas que sean más fáciles de entender y mantener. OOP también permite a los programadores reutilizar el código, lo que puede ahorrar tiempo y esfuerzo.

Algunos de los beneficios de utilizar OOP incluyen:

- Código más fácil de entender y mantener.
- Código más reutilizable.
- Código más flexible y extensible.
- Código más portable.
- Código más robusto y confiable.

OOP es un paradigma popular que se utiliza en una variedad de lenguajes de programación, incluyendo Java, C++, Python, y Ruby.

información adicional sobre el paradigma orientado a objetos:

- **Objetos:** Un objeto es una entidad que tiene estado y comportamiento. El estado de un objeto es representado por sus atributos, mientras que el comportamiento de un objeto es representado por sus métodos.
- **Clases:** Una clase es un modelo para un objeto. Una clase define los atributos y métodos de un objeto.
- **Herencia:** La herencia es la capacidad de una clase para heredar los atributos y métodos de otra clase. La herencia permite a los programadores crear clases más complejas a partir de clases más simples.
- **Polimorfismo:** El polimorfismo es la capacidad de una función para tener diferentes implementaciones dependiendo del tipo de objeto al que se llama. El polimorfismo permite a los programadores escribir código más general que se puede utilizar con diferentes tipos de objetos.
- **Encapsulamiento:** El encapsulamiento es la práctica de ocultar los detalles de implementación de un objeto de los usuarios del objeto. El encapsulamiento permite a los programadores proteger los datos de un objeto de ser modificados por usuarios no autorizados.

OOP es un paradigma poderoso que permite a los programadores crear programas que sean más fáciles de entender, mantener y reutilizar. OOP también permite a los programadores crear programas más flexibles, extensibles, portátiles, robustos y confiables.

Concepto de clase, objeto y constructor

Una clase es un modelo para crear objetos. Una clase define los atributos y métodos de un objeto. Los atributos son las características de un objeto, mientras que los métodos son las acciones que un objeto puede realizar.

Un objeto es una instancia de una clase. Un objeto tiene los mismos atributos y métodos que su clase, pero los valores de los atributos de un objeto pueden ser diferentes de los valores de los atributos de otros objetos de la misma clase.

Un constructor es un método especial que se utiliza para crear un objeto. Un constructor se llama automáticamente cuando se crea un objeto.

En Java, las clases se declaran utilizando la palabra clave `class`. Los objetos se crean utilizando la palabra clave `new`. Los constructores se declaran utilizando la palabra clave `public`.

Por ejemplo, la siguiente clase define una clase llamada `Person` con dos atributos, `name` y `age`. La clase también define un método llamado `greet()` que devuelve un saludo.

```
public class Person {

    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String greet() {
        return "Hello, my name is " + name;
    }
}
```

Para crear un objeto de la clase `Person`, podemos utilizar el siguiente código:

```
Person person = new Person("John Doe", 30);
```

El objeto `person` tendrá los siguientes atributos:

- `name` = "John Doe"
- `age` = 30

Podemos llamar al método `greet()` del objeto `person` utilizando el siguiente código:

```
System.out.println(person.greet());
```

Esto imprimirá la siguiente salida:

```
Hello, my name is John Doe
```

Modelo MVC

El modelo de vista de controlador (MVC) es un patrón de diseño de software que separa los datos (modelo), la interfaz de usuario (vista) y la lógica de procesamiento (controlador). Este patrón es muy popular para el desarrollo de aplicaciones web, ya que permite que los desarrolladores se centren en cada capa de forma independiente.

Modelo

El modelo es la capa de datos de la aplicación. Contiene los datos que se utilizan por la aplicación, como los datos de los usuarios, los productos, los pedidos, etc. El modelo es responsable de almacenar, recuperar y actualizar los datos.

Vista

La vista es la capa de interfaz de usuario de la aplicación. Contiene la interfaz que los usuarios ven y utilizan para interactuar con la aplicación. La vista es responsable de mostrar los datos del modelo a los usuarios y de recopilar los datos de los usuarios para el modelo.

Controlador

El controlador es la capa de lógica de procesamiento de la aplicación. Es responsable de recibir las entradas de los usuarios, interactuar con el modelo y actualizar la vista. El controlador es responsable de controlar el flujo de la aplicación.

El patrón MVC es un patrón muy poderoso que puede ayudar a los desarrolladores a crear aplicaciones web sólidas y escalables. Es un patrón muy popular y hay muchos frameworks web que lo utilizan, como Laravel, Django y Rails.

Beneficios del patrón MVC

El patrón MVC tiene una serie de beneficios, que incluyen:

- **Facilidad de mantenimiento:** El patrón MVC facilita el mantenimiento de las aplicaciones web, ya que las diferentes capas de la aplicación pueden mantenerse de forma independiente.
- **Facilidad de escalado:** El patrón MVC facilita el escalado de las aplicaciones web, ya que las diferentes capas de la aplicación pueden escalarse de forma independiente.
- **Flexibilidad:** El patrón MVC es muy flexible y puede utilizarse para crear una amplia variedad de aplicaciones web.
- **Reusabilidad:** El patrón MVC facilita la reutilización del código, ya que las diferentes capas de la aplicación pueden reutilizarse en diferentes aplicaciones web.

Desventajas del patrón MVC

El patrón MVC también tiene algunas desventajas, que incluyen:

- **Puede ser más complejo que otros patrones de diseño:** El patrón MVC puede ser más complejo que otros patrones de diseño, ya que requiere que los desarrolladores piensen en las diferentes capas de la aplicación.
- **Puede requerir más código:** El patrón MVC puede requerir más código que otros patrones de diseño, ya que requiere que los desarrolladores creen diferentes clases para cada capa de la aplicación.
- **Puede ser más difícil de depurar:** El patrón MVC puede ser más difícil de depurar que otros patrones de diseño, ya que los problemas pueden estar relacionados con diferentes capas de la aplicación.

A pesar de sus desventajas, el patrón MVC es un patrón muy poderoso que puede ayudar a los desarrolladores a crear aplicaciones web sólidas y escalables.

Codigo de la clase Imagen

```
package domain;

/**
 *
 * @author angel
 */

public class Imagen {
    private Integer altura;
    private Integer ancho;
    private String nombre;

    //  segun el color
    private Integer [] escalaGris;
    private Integer [][] escalaColor;

    public String getNombre() {
        return nombre;
    }

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

    public Imagen() {
    }

    public Imagen(Integer altura, Integer ancho, String nombre, Integer []
escalaGris) {
        this.altura = altura;
        this.ancho = ancho;
        this.nombre = nombre;
        this.escalaGris = escalaGris;
    }

    public Imagen(Integer altura, Integer ancho, String nombre, Integer [][]
escalaColor) {
        this.altura = altura;
```

```
        this.ancho = ancho;
        this.nombre = nombre;
        this.escalaColor = escalaColor;
    }

    public Integer getAltura() {
        return altura;
    }

    public void setAltura(Integer altura) {
        this.altura = altura;
    }

    public Integer getAncho() {
        return ancho;
    }

    public Integer[] getEscalaGris() {
        return escalaGris;
    }

    public void setEscalaGris(Integer[] escalaGris) {
        this.escalaGris = escalaGris;
    }

    public Integer[][] getEscalaColor() {
        return escalaColor;
    }

    public void setEscalaColor(Integer[][] escalaColor) {
        this.escalaColor = escalaColor;
    }

    public void setAncho(Integer ancho) {
        this.ancho = ancho;
    }

    @Override
    public String toString() {
        return "Imagen{" + "altura=" + altura + ", ancho=" + ancho + ", nombre=" +
nombre + '}';
    }
}
```