Conception et programmation objet

Laura Fontanella

Licence 3 Informatique et DLMI

September 2, 2025

Static

static

- utilisé pour méthodes et attributs
- Alloue une place dans la mémoire seulement une fois (pas à chaque instanciation de la classe).
- Utilisé pour les propriétés communes à tous les objets de la classe.
- Les méthodes et attributs statiques sont accessibles sans qu'il soit nécessaire créer un objet de la classe.

static-exemple

```java
public class Student{
private int studentID;
private String name;
private static int classSize;

public Student(int studentID, String name){
this.name = name;
this.studentID = studentID;
classSize++;
}

public void display(){
System.out.println("name : "+name +" id: "+studentID+" class size : "+classSize );
}

}
```

Singleton Pattern

Le problème

On veut s'assurer que la classe a toujours une unique et même instance.

La solution

- On crée une **variable instance:** elle est **statique** afin de pouvoir être partagée à travers toutes les instances potentielles de la classe, mais elle est aussi privée pour éviter un accès direct depuis l'extérieur.
- Un **constructeur privé:** Cela empêche la création d'une nouvelle instance en utilisant le mot-clé new à l'extérieur de la classe.
- On défint une méthode **getInstance()**: cette méthode est statique et renvoie la seule instance disponible. Si l'instance n'a pas encore été créée, elle le sera à ce moment-là.

Singleton Pattern

```java
public class Singleton
{
    private static Singleton instance;

    private Singleton()
    {
        System.out.println("Construction du Singleton");
    }
    public static Singleton getInstance()
    {
        if(instance==null)
        {
            instance = new Singleton();
        }
        return instance;
    }
    @Override
    public String toString()
    {
       return "Je suis le Singleton";
    }
}
```

Singleton Pattern

```
public class Main
{
    public static void main(String[] args)
    {
        //creation du singleton
        Singleton singleton = Singleton.getInstance();
        //affichage
        System.out.println(singleton.toString());
    }
}
```

Composition

Le problème

- On veut ajouter des responsabilités à la classe A
- On ne veut/peut pas modifier la classe A
- On ne veut pas hériter de A

La solution avec la composition

La nouvelle classe B comporte un attribut de type A

Composition- Exemple

La classe de départ

```
public class Triangle implements IPolygon{
    private Point a;
    private Point b;
    private Point c;
public Triangle(Point a, Point b, Point c){
    this.a = a;
    this.b = b;
    this.c = c; }
public String description() {
    return " triangle de points "+ a.getCoordinates() +
    b.getCoordinates() +c.getCoordinates();
}}
```

L'interface IPolygon comporte une méthode description().

Composition-Exemple

```
public class ColoredTriangle implements IPolygon{
    private Triangle triangle;
    private Color color;

    public ColoredTriangle(Triangle triangle, Color color){
        this.triangle = triangle;
        this.color = color;
    }
    @Override
    public String description(){
        return triangle.description()+" de couleur "+color;
    }
    public Color getColor(){
        return Color;
    }
}
```

Decorator Pattern

Le problème

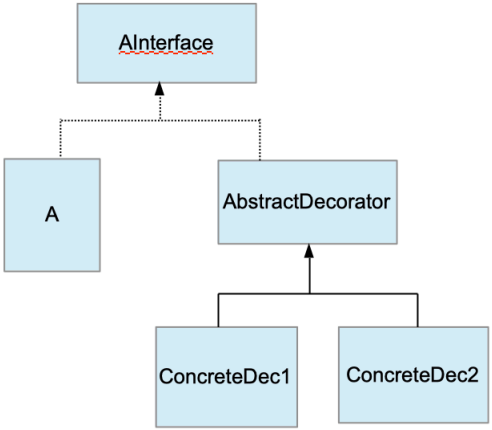
- On veut ajouter des responsabilités à la classe A
- On ne veut/peut pas modifier la classe A
- On ne veut pas hériter de A

La solution avec le decorateur

On crée un **decorateur** c'est à dire une classe abstraite qui enveloppe l'objet d'origine et lui ajoute un nouveau comportement.

Le decorateur implémente la même interface que A. On peut rajouter d'autres décorateurs sans alterer le code.

Decorator pattern



Decorator- Exemple

Le décorateur abstrait:

```
abstract class DecoratedPolygon implements IPolygon{
    protected IPolygon polygon;

    public DecoratedPolygon(IPolygon polygon){
        this.polygon = polygon;
    }

    public String description(){
        return polygon.description();
    }

}
```

Decorator- Exemple

Un decorateur concret:

```
public class ColoredPolygon extends DecoratedPolygon{
    private Color color;

    public ColoredPolygon(IPolygon polygon,  Color color){
        super(polygon);
        this.color = color;

    }
    @Override
    public String description() {
        return super.description()+" de couleur "+color;
    }
}
```

Decorator- Exemple

Dans le main on peut enchainer:

```
public class App {
public static void main(String[] args) throws Exception {
    System.out.println("Program started");

    Point a = new Point(2, 3);
    Point b = new Point(5, 8);
    Point c = new Point(3, 7);

    IPolygon triangle = new LabeledPolygon(new ColoredPolygon(
    new Triangle(a, b, c), Color.RED), "mon label");
    System.out.println(triangle.description());
    }
}
```