Flyweight Pattern

Michal Moravik, SD20w2

Name: Flyweight Pattern

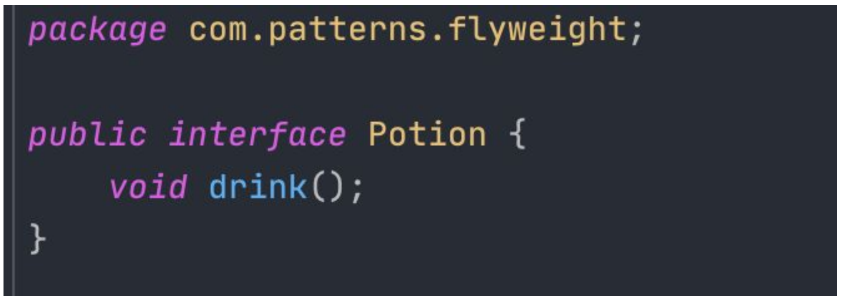
Category: Structural

Intent: To reduce memory usage

Motivation: Used when we need to create a large number of similar objects. To reduce memory usage we can share objects that are similar in some way rather than creating new ones.

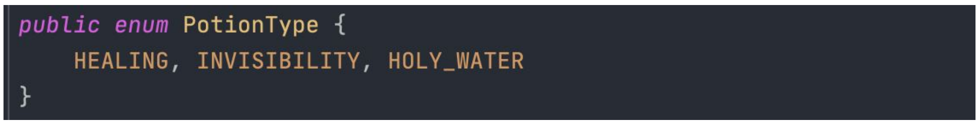
UML:

Implementation:



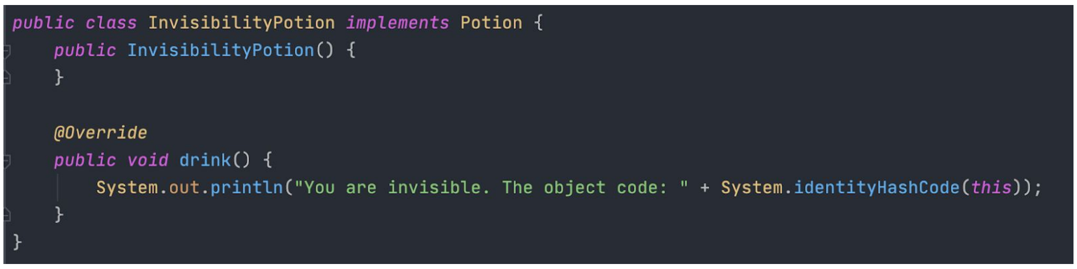
```java
package com.patterns.flyweight;

public interface Potion {
    void drink();
}
```

Common interface

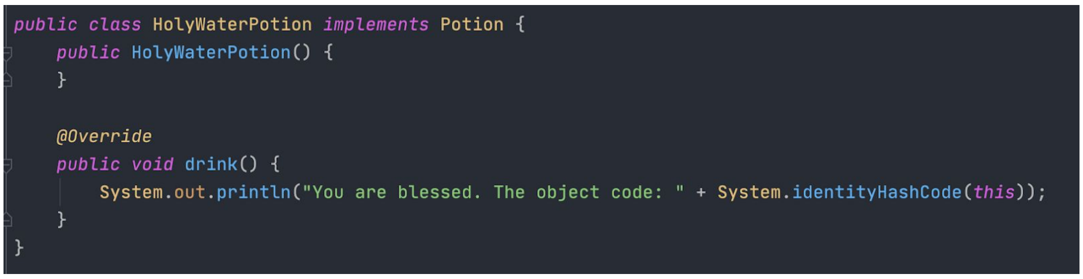


```java
public enum PotionType {
    HEALING, INVISIBILITY, HOLY_WATER
}
```

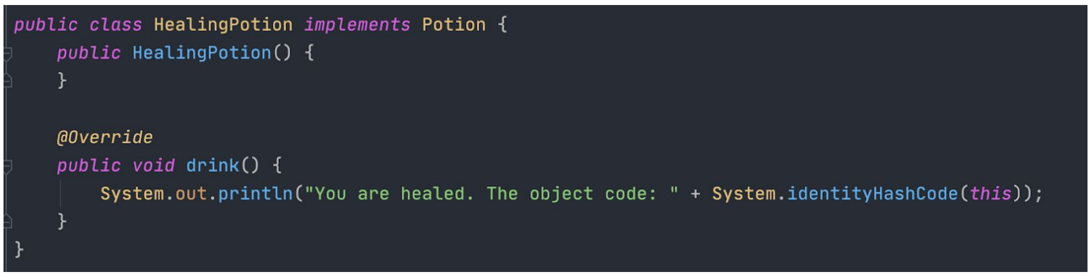
Enum with all possible potions



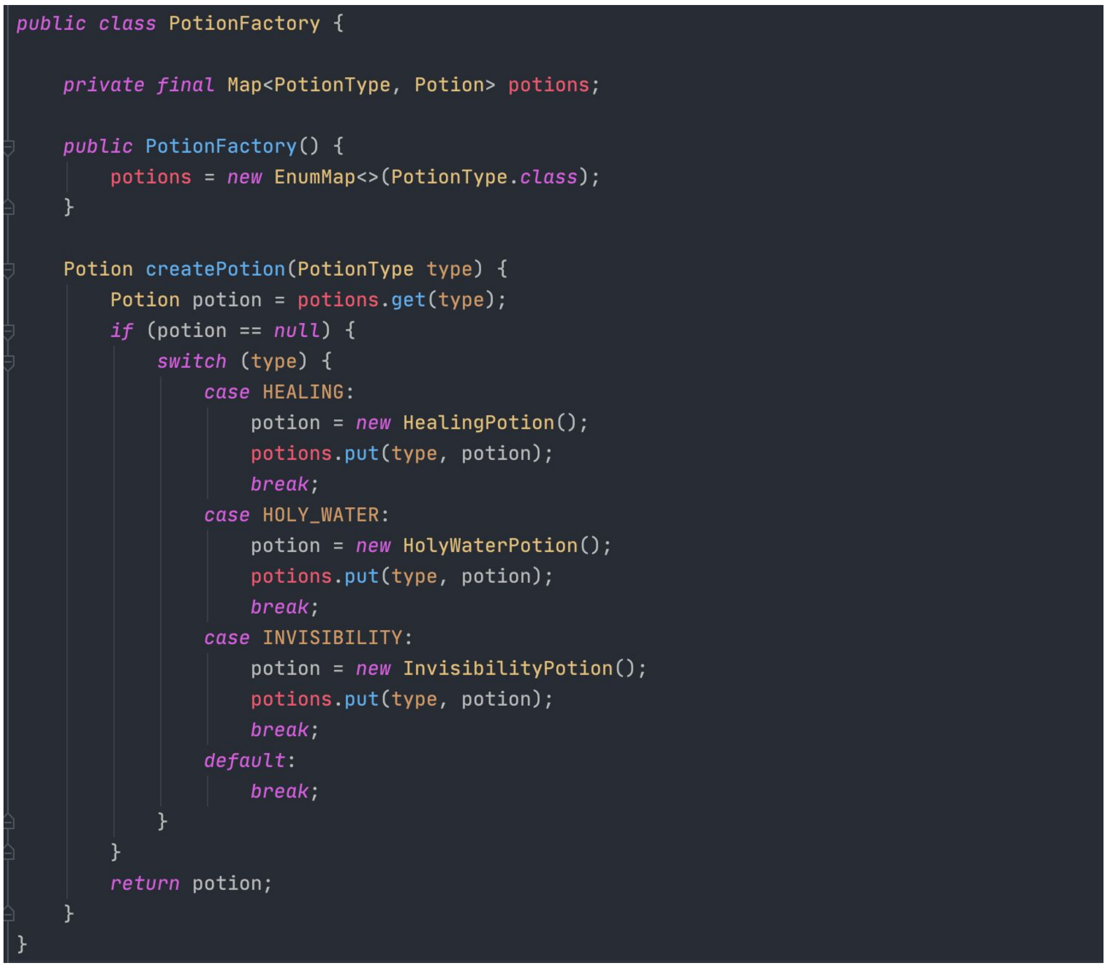
```java
public class InvisibilityPotion implements Potion {
    public InvisibilityPotion() {
    }

    @Override
    public void drink() {
        System.out.println("You are invisible. The object code: " + System.identityHashCode(this));
    }
}
```



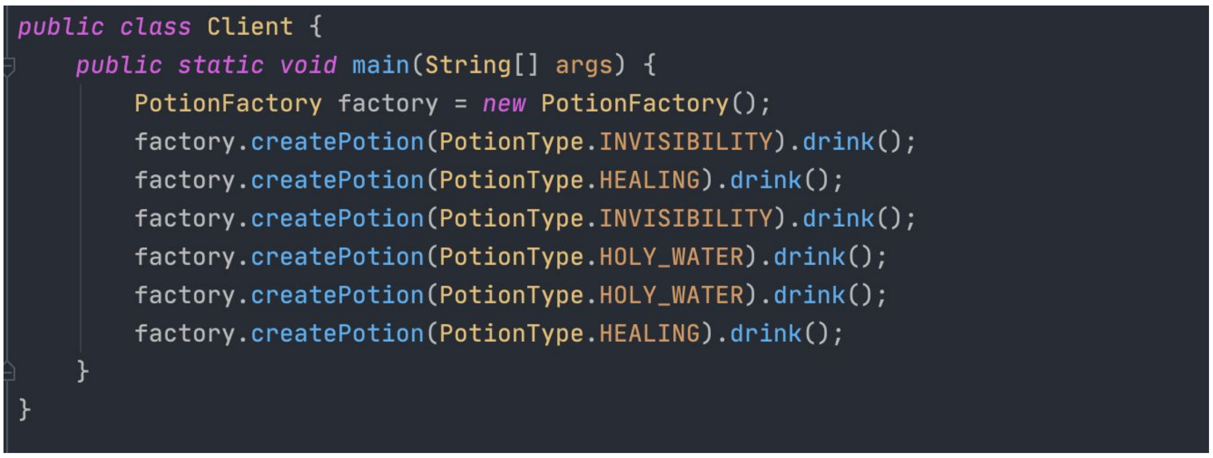
```java
public class HolyWaterPotion implements Potion {
    public HolyWaterPotion() {
    }

    @Override
    public void drink() {
        System.out.println("You are blessed. The object code: " + System.identityHashCode(this));
    }
}
```



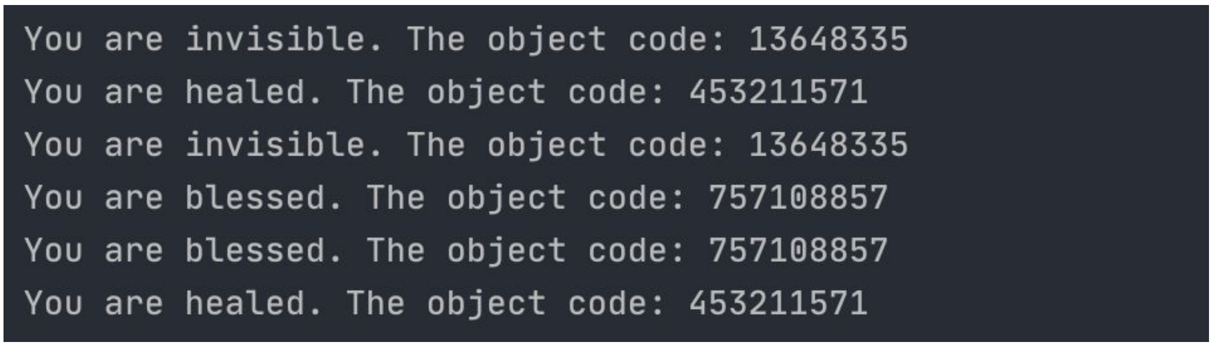
```java
public class HealingPotion implements Potion {
    public HealingPotion() {
    }

    @Override
    public void drink() {
        System.out.println("You are healed. The object code: " + System.identityHashCode(this));
    }
}
```



```java
public class PotionFactory {

    private final Map<PotionType, Potion> potions;

    public PotionFactory() {
        potions = new EnumMap<>(PotionType.class);
    }

    Potion createPotion(PotionType type) {
        Potion potion = potions.get(type);
        if (potion == null) {
            switch (type) {
                case HEALING:
                    potion = new HealingPotion();
                    potions.put(type, potion);
                    break;
                case HOLY_WATER:
                    potion = new HolyWaterPotion();
                    potions.put(type, potion);
                    break;
                case INVISIBILITY:
                    potion = new InvisibilityPotion();
                    potions.put(type, potion);
                    break;
                default:
                    break;
            }
        }
        return potion;
    }
}
```



```java
public class Client {
    public static void main(String[] args) {
        PotionFactory factory = new PotionFactory();
        factory.createPotion(PotionType.INVISIBILITY).drink();
        factory.createPotion(PotionType.HEALING).drink();
        factory.createPotion(PotionType.INVISIBILITY).drink();
        factory.createPotion(PotionType.HOLY_WATER).drink();
        factory.createPotion(PotionType.HOLY_WATER).drink();
        factory.createPotion(PotionType.HEALING).drink();
    }
}
```



```
You are invisible. The object code: 13648335
You are healed. The object code: 453211571
You are invisible. The object code: 13648335
You are blessed. The object code: 757108857
You are blessed. The object code: 757108857
You are healed. The object code: 453211571
```

As we can see, if the same object was already instantiated before, instead of creating a new one, it just returns the one from the HashMap. This is convenient and we save an expensive RAM power while creating an object.

Consequences:
Pros:
- Reduces the number of objects to be instantiated
- Spares RAM power

Cons:
- Introduces a lot of complexity to the system

Known uses:
- Use when you need to deal with heavy RAM processes due to a high number of objects

Related patterns:
- Factory pattern