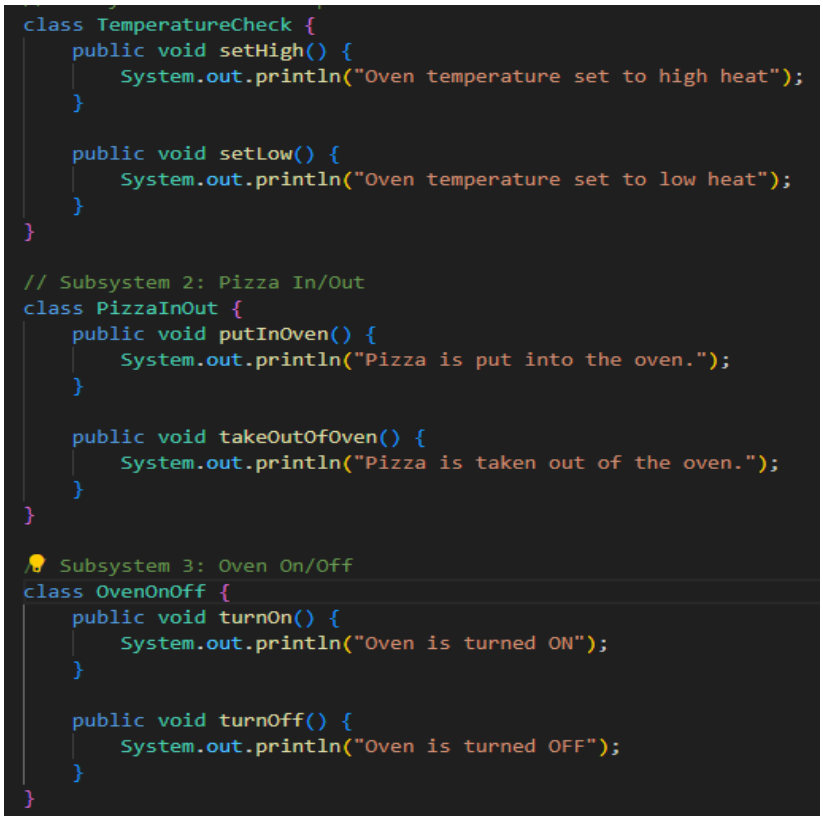
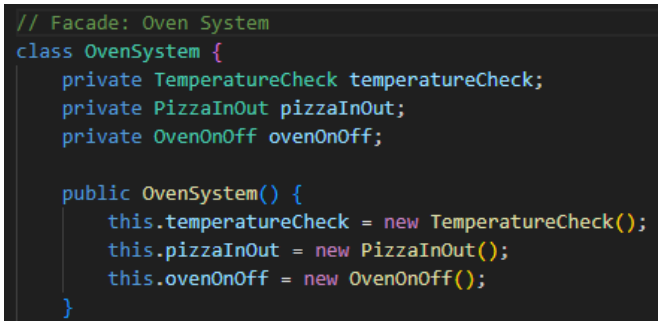
The facade pattern is like the front door to a complex building. Each room has its own key, and accessing different rooms can be complex. This pattern provides an easy-to-use entrance where you just need to interact with the main entrance.



```
class TemperatureCheck {
    public void setHigh() {
        System.out.println("Oven temperature set to high heat");
    }

    public void setLow() {
        System.out.println("Oven temperature set to low heat");
    }
}

// Subsystem 2: Pizza In/Out
class PizzaInOut {
    public void putInOven() {
        System.out.println("Pizza is put into the oven.");
    }

    public void takeOutOfOven() {
        System.out.println("Pizza is taken out of the oven.");
    }
}

// Subsystem 3: Oven On/Off
class OvenOnOff {
    public void turnOn() {
        System.out.println("Oven is turned ON");
    }

    public void turnOff() {
        System.out.println("Oven is turned OFF");
    }
}
```

The TemperatureCheck, PizzaInOut and OvenOnOff are subsystem classes. They represent individual components of the oven system.
They encapsulate specific functionalities related to oven temperature control, which means that they manage the pizzas(objects) state inside and outside the oven. They also can turn the oven on and off.



```
// Facade: Oven System
class OvenSystem {
    private TemperatureCheck temperatureCheck;
    private PizzaInOut pizzaInOut;
    private OvenOnOff ovenOnOff;

    public OvenSystem() {
        this.temperatureCheck = new TemperatureCheck();
        this.pizzaInOut = new PizzaInOut();
        this.ovenOnOff = new OvenOnOff();
    }
```

This is the Facade class. In this case I called it OvenSystem. This acts as a simplified interface to the complex subsystem, which provides a way to act with the oven functionalities. This makes it easier for clients, because it makes it easier for them to use the oven without worrying about the details of each subsystem. It also initializes instances of the subsystems in the constructor

of this class, which means that the pizza baker does not need to worry about creating or managing individual subsystems.

```
public void bakePizza() {
    System.out.println("Baking the pizza...");
    temperatureCheck.setHigh();
    pizzaInOut.putInOven();
    ovenOnOff.turnOn();
    System.out.println("Pizza is successfully baked");
}

public void turnOffOven() {
    System.out.println("Turning off the oven");
    temperatureCheck.setLow();
    pizzaInOut.takeOutOfOven();
    ovenOnOff.turnOff();
}
```

This part of the code has two methods: bakePizza() and turnOffOven(). They orchestrate the interactions with the subsystems to perform higher-level actions.
The method bakePizza() simulates the process of baking the pizza. The method turnOffOven() simulates turning the oven off.

The facade pattern is beneficial in many ways like the interface is simplified which means that it provides a simple and unified interface or a set of interfaces in a subsystem, this means that it reduces the complexity for clients by offering a straight forward API for interacting with the subsystems. It also Encapsulates the Complexity, which means that the details of the subsystems are hidden from clients. This is good because it protects the clients of individual subsystems, allowing for changes in the subsystem implementation without affecting the client code. If I were not to use this pattern then the clients would face an increased Complexity, which means that the clients would need to directly interact with the subsystems understanding their individual interfaces.
It would also introduce code duplication which means that the clients may duplicate code related to subsystem interactions across various parts of their codebase. This would lead to Code redundancy and increased maintenance efforts due to duplicate logic.