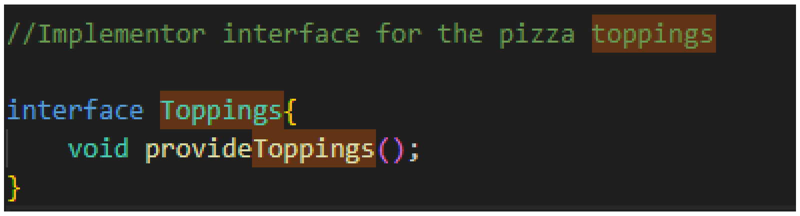
The bridge pattern is a separation of abstraction and implementation, with separation I mean that changes in one will not affect the other.

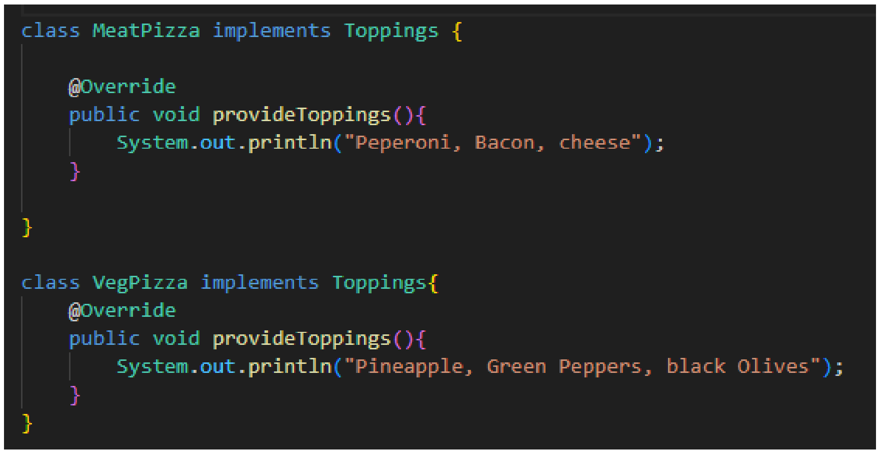
**THE BRIDGE PATTERN EXPLAINED WITH MY CODE EXAMPLE.**



```java
//Implementor interface for the pizza toppings

interface Toppings{
    void provideToppings();
}
```

This interface defines the implementation for the pizza toppings. Both MeatPizza and VegPizza are the concrete implementations of this interface.



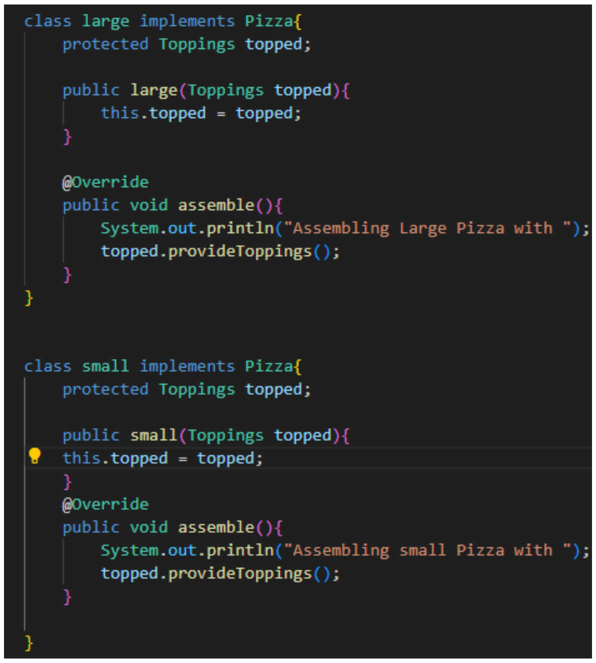
```java
class MeatPizza implements Toppings {

    @Override
    public void provideToppings(){
        System.out.println("Peperoni, Bacon, cheese");
    }

}

class VegPizza implements Toppings{
    @Override
    public void provideToppings(){
        System.out.println("Pineapple, Green Peppers, black Olives");
    }
}
```

In this code snippet I have declared two concrete classes **MeatPizza** and **VegPizza**.
The MeatPizza class implements the **‘Toppings’** interface, which is indicating that it provides an implementation for the methods declared in the **Toppings interface**.
The **@Override** indicates that the following method is intended to Override a method declared in the interface, in this case the **provideToppings()** method of the **Topping interface**.
The line **public void provideToppings()** declares a method of the **Toppings interface** with the public access modifier. It is part of the ‘Toppings’ interface, and this class must provide a concrete implementation for it.
The line **System.out.println(“Pepperoni,Bacon,cheese);** contains the implementation of the **‘provideToppings()method’**. In this case indicating that the toppings are for a meat pizza.
The **VegPizza class** follows a similar structure. The **provideToppings()** method in this case provides a **different indication** which is for a vegetable pizza.



```java
class large implements Pizza{
    protected Toppings topped;

    public large(Toppings topped){
        this.topped = topped;
    }

    @Override
    public void assemble(){
        System.out.println("Assembling Large Pizza with ");
        topped.provideToppings();
    }
}


class small implements Pizza{
    protected Toppings topped;

    public small(Toppings topped){
    this.topped = topped;
    }
    @Override
    public void assemble(){
        System.out.println("Assembling small Pizza with ");
        topped.provideToppings();
    }

}
```

This code defines two classes ‘large’ and ‘small’. Both classes are implementing the ‘Pizza’ interface. Both of these classes have a constructor that takes a ‘Toppings’ object and an ‘assemble’ method. The ‘assemble’  method prints a message and delegates the task of providing toppings to the ‘provideToppings’ method of the associated ‘Toppings’ object. These two classes follow a common interface ‘Pizza’ allowing for consistent structure while also having the variability in pizza sizes. The use of a ‘Toppings’ object ensures a modular and flexible approach to specifying toppings of pizzas for different sizes.

The benefits of bridge pattern in this software and In general is that it separates the **abstraction** from its implementation, which means that it allows them to work independently making it easier for them to operate. There is also an improved Extensibility which means that the pizza types (The refined abstractions) or the new toppings in this case the implementation can be added without modifying the existing code. This makes the system more extensible and adaptable to change. Both Abstractions and Implementation can be reused independently which means that, going back to my code. I could have different pizza sizes with the same toppings or different toppings with different pizza sizes. The bridge pattern has a lot of benefits, but if I would not use it to create this code, then I would have a hard time with tight coupling. This means that the pizza interface would need to include methods like ‘addPepperoni’ or addCheese’ in the Piazza interface making it more complex. The other issue I would face is that I would introduce code duplication, which means that, if I were to introduce new pizza types I would need to duplicate the topping related methods in each pizza class, which could lead to maintenance challenges or code duplication. I would also face problems in extending, which means, any modification to the toppings would also require changes to the ‘Pizza’ interface and all its concrete implementations, which means I would need to do modification in different places which would introduce new errors.

In conclusion I think that the bridge pattern proves that it is crucial for enhancing the flexibility in software design.Without it adapting or adding new Toppings would be challenging because I would need to make changes across the entire Pizza interface and its implementation.