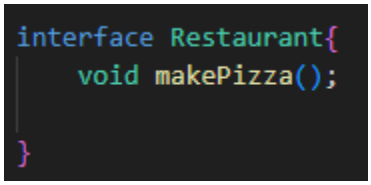
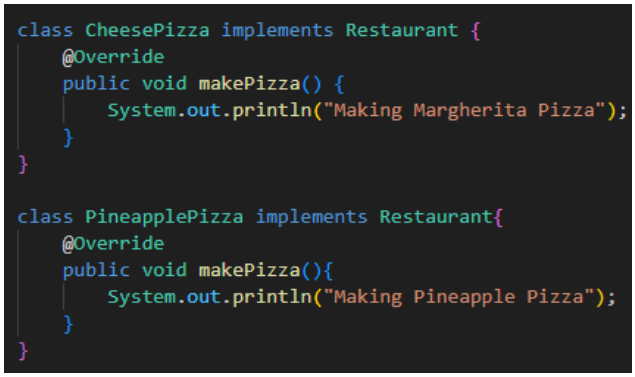
The strategy pattern is a pattern, like imagine you have a toy that can change its appearance. So one day you want your toy to dance and the other day you want your to sing, but you don't want you to change yourself everytime you want to do something new. So the strategy pattern will act like a costume for the toy. Each of the costumes will make sure that the toy can do different things. You can easily take of the costume or add the costume whenever you want to.



```
interface Restaurant{
    void makePizza();

}
```

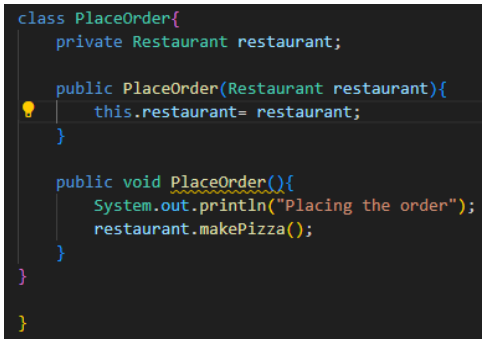
This is the strategy interface Restaurant which declares the method ‘makePizza’. This is the method which serves as the strategy for making the pizza.
The strategy Interface is an interface which declares a set of methods which will represent a family of algorithms or behaviors. Having this makes sure that all classes have a common contract which means that it makes it easier to introduce new strategies and allows the client code to work with different strategies dynamically. This means that I can select different algorithms and behaviors at run-time without changing the client code.



```
class CheesePizza implements Restaurant {
    @Override
    public void makePizza() {
        System.out.println("Making Margherita Pizza");
    }
}

class PineapplePizza implements Restaurant{
    @Override
    public void makePizza(){
        System.out.println("Making Pineapple Pizza");
    }
}
```

Then after the interface I have created two concrete classes CheesePizza and PineapplePizza. They both implement the strategy interface Restaurant. Each of the concrete classes have a different implementation on how to use the ‘makePizza’ method.



```
class PlaceOrder{
    private Restaurant restaurant;

    public PlaceOrder(Restaurant restaurant){
        this.restaurant= restaurant;
    }

    public void PlaceOrder(){
        System.out.println("Placing the order");
        restaurant.makePizza();
    }
}

}
```

At the end of the code I have created Context Class which name is (PlaceOrder):
This is the context class which is using a pizza making strategy. I have also added a constructor to it as a parameter and this constructor will set the strategy for making the pizza. This class

also has a method ‘PlaceOrder’ . This method will basically print that the order has been placed and then guide the task of making a pizza to the selected strategy. They are getting selected with the restaurant.makePizza();

There are many benefits of using the Strategy pattern and my code example profited a lot from this pattern, like Flexibility and Extensibility. This means that new pizza types(strategies) can be easily added by creating new classes and implementing the Restaurant interface. This all can happen without modifying the existing code. The client code can dynamically switch between different pizza types at runtime.
Without the pattern in this case adding a new pizza Type(strategies) would involve changing the client code which would violate the open/close principle.(……)
The pizza making strategy is encapsulated within individual strategy classes, promoting code which will make it easier to maintain.
Changes to the pizza making process can be done within the class without affecting others.
Without the pattern at this point the pizza making logic could be scattered throughout the ‘PlaceOrder’ class, making the code less modular and harder to maintain.
The system can also dynamically choose different pizza making strategies at runtime based on user preference or other conditions.
If the pattern would not be used here then I would need to add more code in PlaceOrder, something like switch cases, which would then handle the different pizza types which would lead to that the code become less dynamic. This could lead to unnecessary error corrections.