The flyweight pattern is a design pattern which will help you save memory by sharing common parts of objects instead of keeping them in each individual object.

```java
interface Pizza {
    void bake();
}
```

This is the Pizza interface which declares a method bake().
All the concrete classes need to implement the bake() method.

```java
//concrete class1
class MeatPizza implements Pizza {
    private String meat;

    public MeatPizza(String meat) {
        this.meat = meat;
    }

    @Override
    public void bake() {
        System.out.println("Baking Meat pizza with " + meat);
    }
}
//concrete class2
class VeggiePizza implements Pizza {
    private String veggies;

    public VeggiePizza(String veggies) {
        this.veggies = veggies;
    }

    @Override
    public void bake() {
        System.out.println("Baking veggie pizza with " + veggies);
    }
}
```

Here we have two concrete classes: MeatPizza and VeggiePizza. Those two classes implement the Pizza interface. They both represent specific types of pizzas with different properties(Veg and Non-veg Pizza). They also have their own bake() method. For both classes I have declared a private instance (private String meat, and private String veggies) . **I have done that because they will store specific meat ingredients for a particular instance of MeatPizza or VeggiePizza.** This is good because **external classes** can not directly access or **modify** the **meat and veggie variable**, which ensures better control over the class's state.
This also saves me from Unintended changes, which means that when I make both variables private nobody can change the value of these variables which will guarantee me to maintain the expected behavior of the 'MeatPizza' class.

```
class PizzaFactory {
    private static final Map<String, Pizza> pizzaMap = new HashMap<>();

    public static Pizza getPizza(String type, String meat, String veggies) {
        String key = type + meat;
        Pizza pizza = pizzaMap.get(key);

        if (pizza == null) {
            switch (type) {
                case "Meat Pizza":
                    pizza = new MeatPizza(meat);
                    break;
                case "Veggie Pizza":
                    pizza = new VeggiePizza(veggies);
                    break;
                default:
                    throw new IllegalArgumentException("Invalid Pizza type: " + type);
            }

            pizzaMap.put(key, pizza);
            System.out.println("Creating a new " + type + " with " + (type.equals("Meat Pizza
        }
        return pizza;
    }
```

This is the PizzaFactory class. Which has a private static final Map<String, Pizza> pizzaMap = new HashMap<>(); This line is responsible for establishing a shared immutable (due to final) map with the PizzaFactory class. This map is used to store and manage instances of Pizza based on their types and specific attributes. Enabling reuse of the object.

private: Only accessible from within the class and not from outside classes.
static: makes the pizzaMap variable shared amongst all instances of the 'PizzaFactory' class.
final: this keyword makes sure that the 'pizzaMap' variable cannot be reassigned once initialized. This will make sure that the reference to the HashMap object cannot be changed once it's set.
Map<String, Pizza>: this declares the type of the 'pizzaMap' variable. This is a map that has a string key with the value type 'Pizza'. The key is used to uniquely identify and retrieve pizza instances in the map.
= new HashMap<>(); : This initializes the 'pizzaMap' variable with a new instance of 'HashMap<String,Pizza. This HashMap will be used to store instances of 'Pizza' with unique keys.

This whole class will check if there is an instance with the same attributes, if there is no instance for this specific type of Pizza then it will create one and store it in the map and then it will return the new object.

There are a lot of benefits in using the flyweight pattern. It is Memory Efficient which means that by reusing instances of pizzas with similar attributes, it reduces the overall memory and resource usage. Instead of creating a new object each time it checks if there is an instance already available with the same attributes and then uses it from the pizzaMap. This in general will make sure that the performance will not be affected in situations where object creation is an expensive operation. If I were not to use flyweight patterns then I would face a lot of issues like Increased Memory usage, which means that there is nothing which will save the created objects so the system would need to sacrifice system Memory and resources to keep creating the instances which could be very bad for a long time.

In summary, the Flyweight pattern, as implemented in the provided code, offers advantages in terms of memory efficiency, performance optimization, and centralized state management. Not using it might lead to increased memory usage, potential performance issues, and reduced efficiency in managing a large number of similar objects.