Builder Pattern Implementation:

public class PizzaBuilder{

public static void main(String[] args){

Pizza margherita = new Pizza.Builder("Margherita", "Large")

.addCheese("Mozzarella")

.addSauce("Tomato")

.build();

Pizza pepperoni = new Pizza.Builder("Pepperoni", "Medium")

.addCheese("Mozzarella")

.addSauce("Tomato")

.addTopping("Pepperoni")

.addTopping("Mushrooms")

.setExtraCheese(true)

.build();

Pizza veggie = new Pizza.Builder("Veggie Delight", "Small")

.addCheese("Vegan Cheese")

.addSauce("Pesto")

.addTopping("Bell Peppers")

.addTopping("Onions")

.addTopping("Olives")

.addTopping("Spinach")

.setSpicy(true)

.build();

System.out.println("Margherita Pizza:");

System.out.println(margherita);

System.out.println("\nPepperoni Pizza:");

System.out.println(pepperoni);

System.out.println("\nVeggie Delight Pizza:");

System.out.println(veggie);

System.out.println("\nBuilding a custom pizza step by step:");

Pizza.Builder customBuilder = new Pizza.Builder("Custom", "Large");

customBuilder.addCheese("Cheddar");

customBuilder.addSauce("BBQ");

customBuilder.addTopping("Chicken");

customBuilder.addTopping("Bacon");

customBuilder.addTopping("Jalapenos");

customBuilder.setExtraCheese(true);

customBuilder.setSpicy(true);

Pizza customPizza = customBuilder.build();

System.out.println(customPizza);

}

static class Pizza {

// Required parameters

private final String name;

private final String size;

private final String cheese;

private final String sauce;

private final List<String> toppings;

private final boolean extraCheese;

private final boolean spicy;

private Pizza(Builder builder) {

this.name = builder.name;

this.size = builder.size;

this.cheese = builder.cheese;

this.sauce = builder.sauce;

this.toppings = builder.toppings;

this.extraCheese = builder.extraCheese;

this.spicy = builder.spicy;

}

@Override

public String toString() {

StringBuilder sb = new StringBuilder();

sb.append("Pizza Name: ").append(name).append("\n");

sb.append("Size: ").append(size).append("\n");

sb.append("Cheese: ").append(cheese).append("\n");

sb.append("Sauce: ").append(sauce).append("\n");

sb.append("Toppings: ").append(String.join(", ", toppings)).append("\n");

sb.append("Extra Cheese: ").append(extraCheese ? "Yes" : "No").append("\n");

sb.append("Spicy: ").append(spicy ? "Yes" : "No");

return sb.toString();

}

public static class Builder {

private final String name;

private final String size;

private String cheese = "Regular";

private String sauce = "Tomato";

private List<String> toppings = new ArrayList<>();

private boolean extraCheese = false;

private boolean spicy = false;

public Builder(String name, String size) {

this.name = name;

this.size = size;

}

public Builder addCheese(String cheese) {

this.cheese = cheese;

return this;

}

public Builder addSauce(String sauce) {

this.sauce = sauce;

return this;

}

public Builder addTopping(String topping) {

this.toppings.add(topping);

return this;

}

public Builder setExtraCheese(boolean extraCheese) {

this.extraCheese = extraCheese;

return this;

}

public Builder setSpicy(boolean spicy) {

this.spicy = spicy;

return this;

}

public Pizza build(){

if (cheese == null)

throw new IllegalStateException("Cheese must be specified");

return new Pizza(this);

}

}

}

}

Key Differences from Previous Example:

1. **Different Domain**: Pizza ordering instead of computer configuration
2. **List of Items**: Handles multiple toppings (a List) instead of fixed components
3. **Boolean Flags**: Uses boolean flags for options like extra cheese and spicy
4. **Validation**: Includes simple validation in the build() method
5. **String Manipulation**: Uses String.join() for displaying toppings list

Design Pattern Benefits Demonstrated:

1. **Complex Object Construction**: Handles pizza with many customizable options
2. **Readable Client Code**: Fluent interface makes ordering clear
3. **Optional Parameters**: Some options (like toppings) are truly optional
4. **Immutable Object**: Pizza object is immutable after construction
5. **Step-by-Step Construction**: Shows how to build an object incrementally

This implementation shows how the Builder Pattern can be applied to a completely different domain while maintaining all the benefits of the pattern.

Output:



