Builder pattern builds a complex object using simple objects and using a step by step approach. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

A Builder class builds the final object step by step. This builder is independent of other objects.

**Implementation**

We have considered a business case of fast-food restaurant where a typical meal could be a burger and a cold drink. Burger could be either a Veg Burger or Chicken Burger and will be packed by a wrapper. Cold drink could be either a coke or pepsi and will be packed in a bottle.

We are going to create an *Item* interface representing food items such as burgers and cold drinks and concrete classes implementing the *Item* interface and a *Packing* interface representing packaging of food items and concrete classes implementing the *Packing* interface as burger would be packed in wrapper and cold drink would be packed as bottle.

We then create a *Meal* class having *ArrayList* of *Item* and a *MealBuilder* to build different types of *Meal* objects by combining *Item*. *BuilderPatternDemo*, our demo class will use *MealBuilder* to build a *Meal*.

![Diagram](https://via.placeholder.com/150)

**Step 1**

Create an interface *Item* representing food item and packing.

*Item.java*

```java
public interface Item {
    public String name();
    public Packing packing();
    public float price();
}
```

*Packing.java*
Step 2

Create concrete classes implementing the Packing interface.

**Wrapper.java**

```java
public class Wrapper implements Packing {
    @Override
    public String pack() {
        return "Wrapper";
    }
}
```

**Bottle.java**

```java
public class Bottle implements Packing {
    @Override
    public String pack() {
        return "Bottle";
    }
}
```

Step 3

Create abstract classes implementing the Item interface providing default functionalities.

**Burger.java**

```java
public abstract class Burger implements Item {
    @Override
    public Packing packing() {
        return new Wrapper();
    }
    @Override
    public abstract float price();
}
```

**ColdDrink.java**

```java
public abstract class ColdDrink implements Item {
    @Override
    public Packing packing() {
        return new Bottle();
    }
    @Override
    public abstract float price();
}
```

Step 4

Create concrete classes extending Burger and ColdDrink classes

**VegBurger.java**
public class VegBurger extends Burger {

    @Override
    public float price() {
        return 25.0f;
    }

    @Override
    public String name() {
        return "Veg Burger";
    }
}

ChickenBurger.java

public class ChickenBurger extends Burger {

    @Override
    public float price() {
        return 50.5f;
    }

    @Override
    public String name() {
        return "Chicken Burger";
    }
}

Coke.java

public class Coke extends ColdDrink {

    @Override
    public float price() {
        return 30.0f;
    }

    @Override
    public String name() {
        return "Coke";
    }
}

Pepsi.java

public class Pepsi extends ColdDrink {

    @Override
    public float price() {
        return 35.0f;
    }

    @Override
    public String name() {
        return "Pepsi";
    }
}

Step 5

Create a Meal class having Item objects defined above.

Meal.java

import java.util.ArrayList;
import java.util.List;
public class Meal {
    private List<Item> items = new ArrayList<Item>();

    public void addItem(Item item) {
        items.add(item);
    }

    public float getCost() {
        float cost = 0.0f;
        for (Item item : items) {
            cost += item.price();
        }
        return cost;
    }

    public void showItems() {
        for (Item item : items) {
            System.out.print("Item : ");
            System.out.print(item.name());
            System.out.print(", Packing : ");
            System.out.print(item.packing().pack());
            System.out.print(", Price : ");
            System.out.println(item.price());
        }
    }
}

Step 6
Create a MealBuilder class, the actual builder class responsible to create Meal objects.

MealBuilder.java

public class MealBuilder {
    public Meal prepareVegMeal() {
        Meal meal = new Meal();
        meal.addItem(new VegBurger());
        meal.addItem(new Coke());
        return meal;
    }

    public Meal prepareNonVegMeal() {
        Meal meal = new Meal();
        meal.addItem(new ChickenBurger());
        meal.addItem(new Pepsi());
        return meal;
    }
}

Step 7
BuiderPatternDemo uses MealBuider to demonstrate builder pattern.

BuilderPatternDemo.java

public class BuilderPatternDemo {
    public static void main(String[] args) {
        MealBuilder mealBuilder = new MealBuilder();
        Meal vegMeal = mealBuilder.prepareVegMeal();
        System.out.println("Veg Meal");
        vegMeal.showItems();
        System.out.println("Total Cost: "+ vegMeal.getCost());
        Meal nonVegMeal = mealBuilder.prepareNonVegMeal();
        System.out.println("\nNon-Veg Meal");
    }
}
nonVegMeal.showItems();
    System.out.println("Total Cost: " + nonVegMeal.getCost());
}

**Step 8**

Verify the output.

**Veg Meal**
Item : Veg Burger, Packing : Wrapper, Price : 25.0  
Item : Coke, Packing : Bottle, Price : 30.0  
Total Cost: 55.0

**Non-Veg Meal**
Item : Chicken Burger, Packing : Wrapper, Price : 50.5  
Item : Pepsi, Packing : Bottle, Price : 35.0  
Total Cost: 85.5