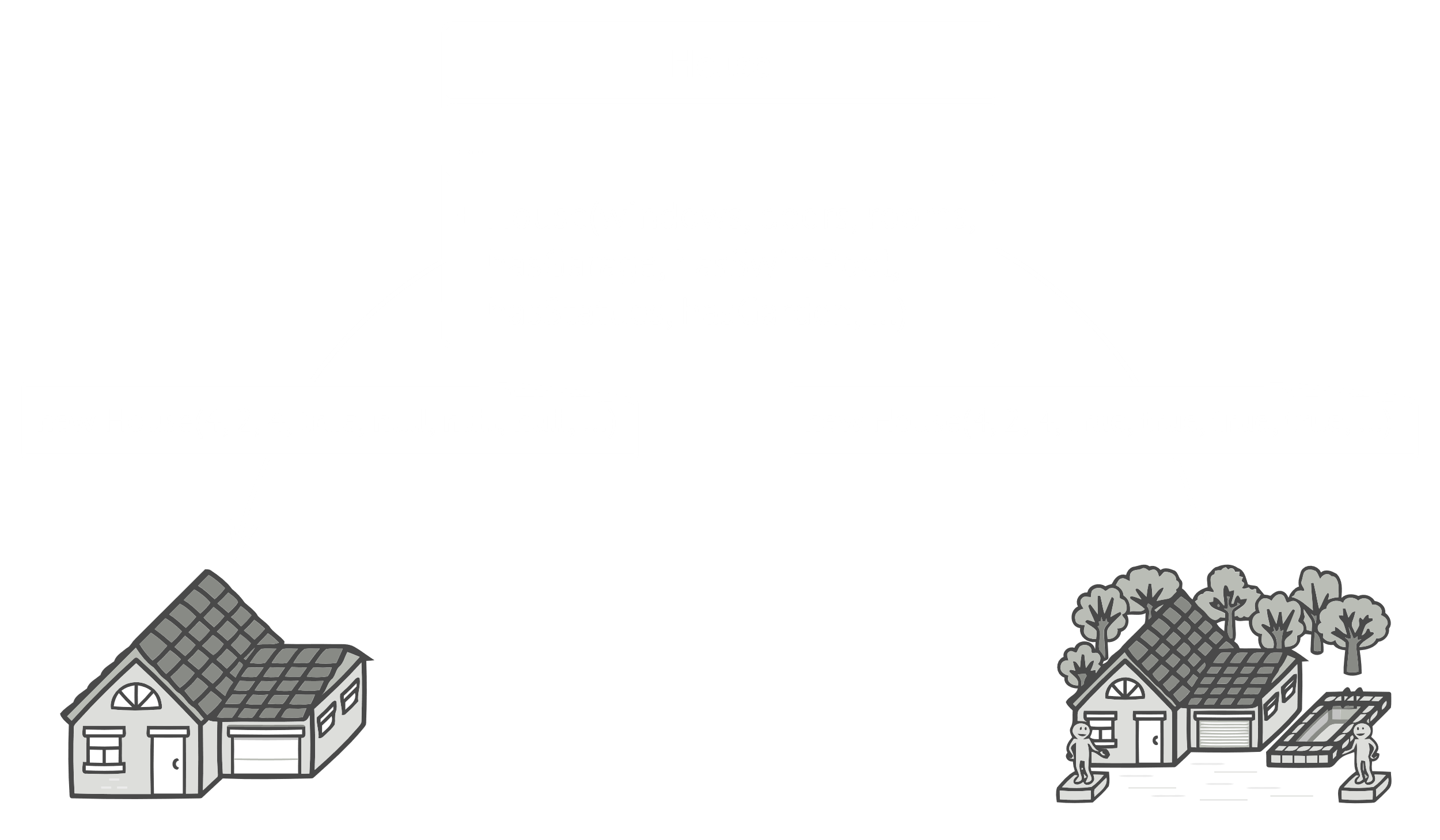
Builder Pattern

The **Builder Pattern** is used to build complex objects step by step. This is a **creational pattern** that allows us to separate the construction process from the representation of the object, so that the same construction process can be used for different representations.

Consider the building process of a house that can have a variety of features, such as a swimming pool, a garden, a garage, etc. In order to know which features the client wants, we would have to force the client to specify a **large number of parameters**. Additionally, a large number of the parameters would be left empty as well.



Building the house based on so many parameters would make the object’s class messy. The Builder Pattern allows us to **extract the building logic** into its own class.

Consider that we want to ‘build’ a meal, which can consist of a variety of items. The actual object would just have a list of items, while the **builder adds items** one by one based on the requirements.

public class Meal {  
 private *List*<*Item*> items = new ArrayList<>();  
  
 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 : " + item.name());  
 System.*out*.print(", Packing : " + item.packing().pack());  
 System.*out*.println(", Price : " + item.price());  
 }  
 }  
}  
  
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;  
 }  
}

JAVA

The items themselves are arbitrary as long as they implement the same interface.

