10 Creational: Builder Pattern for Constructing a House

**Objective** Start with a “telescoping-constructor” House class, identify its pain-points, then refactor through three Builder “maturity” levels—basic, fluent, and director-driven—so you can create different house variants without constructor-hell or mutable setters.

#### **Starter code (src/main/java/legacy/)**

package legacy;

/\*\* A mutable, telescoping-constructor House (anti-pattern). \*/

public class House {

private String foundation;

private String walls;

private String roof;

private String interior;

public House(String foundation){ this.foundation = foundation; }

public House(String foundation,String walls){

this.foundation = foundation; this.walls = walls;

}

public House(String foundation,String walls,String roof,String interior){

this.foundation = foundation; this.walls = walls;

this.roof = roof; this.interior = interior;

}

/\* setters mutate state → thread-unsafe \*/

public void setRoof(String roof){ this.roof = roof; }

/\* getters omitted for brevity \*/

}

#### **Tasks**

1 **Problem analysis** Write analysis/builder\_problems.md listing at least three drawbacks of telescoping constructors (readability, scalability, immutability).  
 2 **Level 1 – Basic Builder** Create clean.house.House with an **inner static Builder** offering setFoundation() … build(); no director, no chaining.  
 3 **Level 2 – Fluent Builder** Enhance the builder with chaining (return this) and make the resulting House immutable (all fields final, no setters).  
 4 **Level 3 – Director + Presets** Add HouseDirector that orchestrates common variants: constructBasicHouse(), constructLuxuryHouse().  
 5 **Client** clean.Main builds

* a basic house directly with Level 2 fluent calls
* a luxury house via HouseDirector
* print their parts to verify.  
   6 **Reflection (reflection.md)**
* Compare the three builder maturity levels: when is each sufficient?
* How does the pattern improve readability and immutability?
* What trade-offs (extra classes, boilerplate)?

#### **Deliverables**

analysis/builder\_problems.md

src/main/java/clean/house/House.java ← Level 2 immutable product + Builder

src/main/java/clean/director/HouseDirector.java

src/main/java/clean/Main.java

reflection.md

README.md

## **Solution (three levels)**

**Level 1 – Basic Builder** *Goal:* separate construction from representation, but no chaining or immutability yet.

package clean.level1;

public class House {

private String foundation, walls, roof, interior;

private House(Builder b){

this.foundation=b.foundation; this.walls=b.walls;

this.roof=b.roof; this.interior=b.interior;

}

public static class Builder {

private String foundation, walls, roof, interior;

public Builder setFoundation(String f){ this.foundation=f; return this; }

public Builder setWalls(String w){ this.walls=w; return this; }

public Builder setRoof(String r){ this.roof=r; return this; }

public Builder setInterior(String i){ this.interior=i; return this; }

public House build(){ return new House(this); }

}

}

**Level 2 – Fluent + Immutable (final solution class for assignment)**

package clean.house;

/\*\* Immutable House built via fluent Builder. \*/

public final class House {

private final String foundation;

private final String walls;

private final String roof;

private final String interior;

private House(Builder b){

this.foundation=b.foundation;

this.walls =b.walls;

this.roof =b.roof;

this.interior =b.interior;

}

/\* getters \*/

public String getFoundation(){ return foundation; }

public String getWalls(){ return walls; }

public String getRoof(){ return roof; }

public String getInterior(){ return interior; }

/\*\* Fluent Builder (Level 2). \*/

public static class Builder{

private String foundation;

private String walls;

private String roof;

private String interior;

public Builder foundation(String f){ this.foundation=f; return this; }

public Builder walls(String w){ this.walls=w; return this; }

public Builder roof(String r){ this.roof=r; return this; }

public Builder interior(String i){ this.interior=i; return this; }

/\*\* Basic validation example \*/

public House build(){

if(foundation==null) throw new IllegalStateException("foundation required");

return new House(this);

}

}

}

**Level 3 – Director with pre-sets**

package clean.director;

import clean.house.House;

/\*\* Orchestrates common construction recipes (Level 3). \*/

public final class HouseDirector {

public House constructBasic(House.Builder b){

return b.foundation("Concrete")

.walls("Brick")

.roof("Tile")

.build();

}

public House constructLuxury(House.Builder b){

return b.foundation("Stone")

.walls("Marble")

.roof("Slate")

.interior("Modern")

.build();

}

}

**Client**

package clean;

import clean.house.House;

import clean.director.HouseDirector;

public class Main {

public static void main(String[] args) {

// Level 2 direct fluent build

House basic = new House.Builder()

.foundation("Concrete")

.walls("Wood")

.roof("Shingle")

.build();

// Level 3 via director

HouseDirector dir = new HouseDirector();

House luxury = dir.constructLuxury(new House.Builder());

print("Basic", basic);

print("Luxury", luxury);

}

private static void print(String label, House h){

System.out.println(label+" House -> Foundation:"+h.getFoundation()+

", Walls:"+h.getWalls()+", Roof:"+h.getRoof()+

(h.getInterior()!=null? ", Interior:"+h.getInterior():""));

}

}

**Console output**

Basic House -> Foundation:Concrete, Walls:Wood, Roof:Shingle

Luxury House -> Foundation:Stone, Walls:Marble, Roof:Slate, Interior:Modern

### **Level Summary**

| **Level** | **Features** | **Pros** | **Cons** | **Typical use** |
| --- | --- | --- | --- | --- |
| **1** Basic builder (no chaining, mutable product) | Separates construction logic | Minimal refactor | Product still mutable | Quick escape from telescoping constructors |
| **2** Fluent, immutable product | Readable, thread-safe, validation in build() | Extra boilerplate | Sufficient for most apps |  |
| **3** Director orchestrating presets | Re-usable recipes, enforces build order | Another class to maintain | Complex configs, different SKUs |  |

The Builder pattern now lets you materialise varied House representations without exploding constructors, while keeping each House instance immutable and easy to extend with new parts.