Builder Design Pattern

- The **Builder Design Pattern** is used when you want to construct a complex object step by step.
- Instead of creating the object in one go (e.g., using a big constructor), you use a builder to add parts gradually.

You're creating a class called **House**. A house can have:

- A **garden** (optional).
- A **swimming pool** (optional).
- A **number of floors** (required).
- A **garage** (optional).

If you don't use the Builder pattern, you'll end up with:

- **Too many constructors**
- **Long, confusing constructors**

House(int **floors**)

House(int **floors**, boolean **hasGarden**)

House(int **floors**, boolean **hasGarden**, boolean **hasGarage**)

House(int **floors**, boolean **hasGarden**, boolean **hasGarage**, boolean **hasSwimmingPool**)

<u>In the Main Method-</u>

House h1 = new House(2, true, false, true);

Builder Pattern Solution:

The Builder pattern solves this by:

1. Breaking the object creation process into **steps**.
2. Making the code **readable**.
3. Allowing **optional parts** to be added only when needed.

```java
// Product: House
class House {
    private int floors;
    private boolean hasGarden;
    private boolean hasSwimmingPool;
    private boolean hasGarage;

    // Private constructor to force the use of Builder
    private House(Builder builder) {
        this.floors = builder.floors;
        this.hasGarden = builder.hasGarden;
        this.hasSwimmingPool = builder.hasSwimmingPool;
        this.hasGarage = builder.hasGarage;
    }

    @Override
    public String toString() {
        return "House [floors=" + floors + ", hasGarden=" + hasGarden +
               ", hasSwimmingPool=" + hasSwimmingPool + ", hasGarage=" + hasGarage + "]";
    }
```

```java
// Builder Class (Nested)
public static class Builder {
    private int floors; // Required
    private boolean hasGarden = false; // Optional
    private boolean hasSwimmingPool = false; // Optional
    private boolean hasGarage = false; // Optional

    // Constructor for required fields
    public Builder(int floors) {
        this.floors = floors;
    }
```

```java
        // Methods for optional fields
        public Builder addGarden() {
            this.hasGarden = true;
            return this;
        }

        public Builder addSwimmingPool() {
            this.hasSwimmingPool = true;
            return this;
        }

        public Builder addGarage() {
            this.hasGarage = true;
            return this;
        }

        // Final build method
        public House build() {
            return new House(this);
        }
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        // Build a simple house with only floors
        House house1 = new House.Builder(2).build();

        // Build a luxury house with all features
        House house2 = new House.Builder(3)
                            .addGarden()
                            .addSwimmingPool()
                            .addGarage()
                            .build();

        System.out.println(house1);
        System.out.println(house2);
    }
}
```

When Does Normal Code Work Fine?

For **simple objects**, normal code works fine, such as:

- Objects with only a few fields.
- Objects where all fields are mandatory or there are no complex dependencies.

For example:

```java
Person person = new Person("John", 30); // All fields are mandatory
```

When is Builder Better?

The Builder pattern shines when:

1. **Complex Objects**:
    - The object has many optional fields or parts (e.g., a `House` with optional features like a garden, pool, garage).
2. **Readability and Maintainability**:
    - You want the object creation process to be clear and easy to understand.
3. **Immutable and Valid Objects**:
    - The Builder ensures objects are immutable and valid when they are created.
4. **Extensibility**:
    - It is easier to add new optional features in a Builder without breaking existing code.

Example Question

You are tasked with developing a system for a smartphone customization app that allows users to build their own smartphone based on their preferences. Each smartphone has mandatory attributes like `model name` (e.g., "ProX", "LiteZ") and `processor type` (e.g., "Snapdragon", "Apple A-Series"). In addition, users can choose optional features such as `wireless charging`, `water resistance`, `5G support`, and `stylus compatibility`. The system must ensure that every smartphone has the mandatory attributes set, while allowing customers to add only the features they want.

Design the `Smartphone` class using the Builder Pattern to fulfill these requirements. Then, write code to demonstrate how the Builder Pattern can create a basic smartphone (e.g., `Model: LiteZ, Processor: Snapdragon`) and a fully-featured smartphone (e.g., `Model: ProX, Processor: Apple A-Series` with all optional features). Discuss why the Builder Pattern is better suited for this task compared to constructors or setters.

```
// Product: Smartphone
class Smartphone {
    // Mandatory attributes
    private final String modelName;
    private final String processorType;

    // Optional attributes
    private final boolean wirelessCharging;
    private final boolean waterResistance;
    private final boolean has5G;
    private final boolean hasStylus;

    // Private constructor to force the use of Builder
    private Smartphone(Builder builder) {
        this.modelName = builder.modelName;
        this.processorType = builder.processorType;
        this.wirelessCharging = builder.wirelessCharging;
        this.waterResistance = builder.waterResistance;
        this.has5G = builder.has5G;
        this.hasStylus = builder.hasStylus;
    }

    @Override
    public String toString() {
        return "Smartphone [Model Name=" + modelName + ", Processor=" + processorType +
                ", Wireless Charging=" + wirelessCharging + ", Water Resistance=" + waterResistance
+
                ", 5G Support=" + has5G + ", Stylus=" + hasStylus + "]";
    }

    // Nested Builder class
    public static class Builder {
        // Mandatory fields
        private final String modelName;
        private final String processorType;

        // Optional fields with default values
        private boolean wirelessCharging = false;
        private boolean waterResistance = false;
        private boolean has5G = false;
        private boolean hasStylus = false;

        // Constructor for mandatory fields
        public Builder(String modelName, String processorType) {
```

```
        this.modelName = modelName;
        this.processorType = processorType;
    }

    // Methods to set optional attributes
    public Builder addWirelessCharging() {
        this.wirelessCharging = true;
        return this;
    }

    public Builder addWaterResistance() {
        this.waterResistance = true;
        return this;
    }

    public Builder add5GSupport() {
        this.has5G = true;
        return this;
    }

    public Builder addStylus() {
        this.hasStylus = true;
        return this;
    }

    // Final build method
    public Smartphone build() {
        return new Smartphone(this);
    }
  }
}

public class Main {
  public static void main(String[] args) {
    // Build a basic smartphone
    Smartphone basicPhone = new Smartphone.Builder("LiteZ", "Snapdragon").build();

    // Build a fully-featured luxury smartphone
    Smartphone luxuryPhone = new Smartphone.Builder("ProX", "Apple A-Series")
                            .addWirelessCharging()
                            .addWaterResistance()
                            .add5GSupport()
                            .addStylus()
                            .build();
```

```
        // Print the details
        System.out.println(basicPhone);
        System.out.println(luxuryPhone);
    }
}
```

OUTPUT

```
Smartphone [Model Name=LiteZ, Processor=Snapdragon, Wireless Charging=false, Water Resistance=false, 5G Support=false, Stylus=false]
Smartphone [Model Name=ProX, Processor=Apple A-Series, Wireless Charging=true, Water Resistance=true, 5G Support=true, Stylus=true]
```