



**1. Overview**

The **Builder Pattern** is a **creational design pattern** that allows you to construct complex objects step by step, separating the construction process from the final representation.  
It is particularly useful when:

* The object has **many optional parameters**.
* The object creation process is **complex** and might vary.

**2. Use Case**

* Creating **immutable objects** with many optional parameters (e.g., StringBuilder, java.lang.StringBuilder).
* Building **complex configuration objects** like HTTP requests, XML documents, or reports.
* Avoiding **telescoping constructors** (constructors with many parameters).

**Example scenario:**  
Suppose you want to build a User object that can have multiple optional fields like email, phone, and address. Instead of making multiple constructors or using setters, you use a **Builder**.

**3. Advantages**

✅ **Readable object creation** — improves clarity by showing what’s being set.  
✅ **Immutable objects** — build object fully before making it immutable.  
✅ **Avoids constructor explosion** — no need for multiple overloaded constructors.  
✅ **Step-by-step construction** — can construct objects in multiple steps.

**4. Disadvantages**

❌ **More boilerplate code** — requires an extra Builder class.  
❌ **Overkill for simple objects** — best for objects with many optional parameters.  
❌ **Not suitable for mandatory step ordering unless enforced** — may need extra validation logic.

## ****5. Example in Java****

java

CopyEdit

// Product class

class User {

private final String firstName; // required

private final String lastName; // required

private final String email; // optional

private final String phone; // optional

// Private constructor - only Builder can create User

private User(Builder builder) {

this.firstName = builder.firstName;

this.lastName = builder.lastName;

this.email = builder.email;

this.phone = builder.phone;

}

public static class Builder {

private final String firstName;

private final String lastName;

private String email;

private String phone;

public Builder(String firstName, String lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

public Builder email(String email) {

this.email = email;

return this;

}

public Builder phone(String phone) {

this.phone = phone;

return this;

}

public User build() {

return new User(this);

}

}

@Override

public String toString() {

return firstName + " " + lastName + ", Email: " + email + ", Phone: " + phone;

}

}

// Client code

public class BuilderPatternExample {

public static void main(String[] args) {

User user = new User.Builder("John", "Doe")

.email("john@example.com")

.phone("1234567890")

.build();

System.out.println(user);

}

}

**Output:**

yaml

CopyEdit

John Doe, Email: john@example.com, Phone: 1234567890

## ****6. Where It's Used in Java****

* StringBuilder and StringBuffer
* java.lang.StringBuilder#append()
* java.nio.ByteBuffer and related buffer classes
* java.sql.PreparedStatement style chainingThe **Builder Pattern** is useful when:
* An object requires **many optional parameters** or **complex construction steps**.
* The object is **immutable** and needs to be built step-by-step.
* You want to **separate object construction** from its representation.
* Example scenarios:
  1. **Creating a complex UI form** with optional fields.
  2. **Building an HTTP request** where only some headers or body parts are set.
  3. **Creating domain objects** like User, Car, House with optional parameters.
  4. **Parsing XML/JSON** into objects with partial data.