

# Builder pattern level 1



Builder pattern

now we go level by level growth like a game on this java topic, provide coding example with its explanation, real-world use case example,necessary hints, points to remember,usual mistakes,etc.

start lvl1

Perfect! Let's start your **Builder Pattern journey in Java**, level by level like a game. 🎮

## Level 1: Builder Pattern – Basic Introduction

**Goal:** Understand the simplest form of the Builder pattern, why it exists, and its basic implementation.

### 1 What is Builder Pattern?

The Builder Pattern is a **creational design pattern** used to construct complex objects step by step. It's particularly useful when:

- An object has **many optional parameters**.
- You want the object to be **immutable** after creation.
- You want **clear and readable code** instead of multiple constructors.

**Key idea:** Separate the **construction** of an object from its **representation**.

### 2 Why use it?

Without Builder:

```
java

class Pizza {
    private String size;
    private boolean cheese;
    private boolean pepperoni;
    private boolean bacon;

    public Pizza(String size, boolean cheese, boolean pepperoni, boolean bacon) {
        this.size = size;
        this.cheese = cheese;
        this.pepperoni = pepperoni;
        this.bacon = bacon;
    }
}
```

```
// Usage
Pizza pizza = new Pizza("Large", true, true, false);
```

### ✓ Problems:

- Too many constructor parameters → confusing.
- Hard to remember the order of boolean flags.
- Difficult to create objects with only some optional fields.

## 3 Builder Pattern Implementation

```
java

// Product class
class Pizza {
    private String size;
    private boolean cheese;
    private boolean pepperoni;
    private boolean bacon;

    // Private constructor to force usage of builder
    private Pizza(PizzaBuilder builder) {
        this.size = builder.size;
        this.cheese = builder.cheese;
        this.pepperoni = builder.pepperoni;
        this.bacon = builder.bacon;
    }

    // Static nested Builder class
    public static class PizzaBuilder {
        private String size;
        private boolean cheese;
        private boolean pepperoni;
        private boolean bacon;

        public PizzaBuilder(String size) {
            this.size = size;
        }

        public PizzaBuilder addCheese() {
            this.cheese = true;
            return this;
        }

        public PizzaBuilder addPepperoni() {
            this.pepperoni = true;
            return this;
        }

        public PizzaBuilder addBacon() {
            this.bacon = true;
            return this;
        }

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

    @Override
    public String toString() {
        return "Pizza [size=" + size + ", cheese=" + cheese +
               ", pepperoni=" + pepperoni + ", bacon=" + bacon + "]";
    }
}

// Usage
```

```

public class Main {
    public static void main(String[] args) {
        Pizza pizza = new Pizza.PizzaBuilder("Large")
            .addCheese()
            .addPepperoni()
            .build();

        System.out.println(pizza);
    }
}

```

## 4 Explanation

- **PizzaBuilder class** → Builds Pizza object step by step.
- **Fluent interface** → Methods return `this` for chaining.
- **Private constructor** → Forces usage of builder.
- **build() method** → Returns the final Pizza object.

### ✓ Advantages:

- Code is **clean and readable**.
- Supports **immutable objects**.
- Easy to **add/remove optional fields**.

### ⚠ Common mistakes:

1. Forgetting `return this` in builder methods → breaks chaining.
2. Making the constructor public → defeats the purpose.
3. Overcomplicating simple objects → builder is useful mainly for **complex objects**.

## 5 Real-world use case

- **Building HTTP requests** in Java (`HttpRequest.Builder`).
- **Creating GUI components** (Swing, JavaFX) with many optional parameters.
- **Constructing complex domain objects** in enterprise apps (e.g., `UserProfile`, `Invoice`).

## 6 Level 1 Hints / Points to Remember

- Use **Builder** when you have **many optional parameters**.
- **Immutable objects** are safer in multithreaded code.
- Builder improves **readability** and reduces **constructor overloading hell**.
- Always `return this` in builder methods for fluent chaining.

If you want, I can take you to **Level 2**, where we'll **compare Builder with telescoping constructors & setters**, and then show **advanced Builder with Director pattern** for real-world complex objects.

Do you want me to proceed to **Level 2?**



next

Perfect! Let's move to **Level 2.** 🎉

## Level 2: Builder vs Telescoping Constructors & Setters

**Goal:** Understand why Builder is preferred over multiple constructors or setters, and explore more structured usage.

### 1 The Problem: Telescoping Constructors

Imagine a `Computer` class with many optional parameters:

```
java

class Computer {
    private String CPU;
    private int RAM;
    private int storage;
    private boolean graphicsCard;
    private boolean wifi;

    // Multiple constructors for different combinations
    public Computer(String CPU, int RAM) {
        this(CPU, RAM, 0);
    }

    public Computer(String CPU, int RAM, int storage) {
        this(CPU, RAM, storage, false);
    }

    public Computer(String CPU, int RAM, int storage, boolean graphicsCard) {
        this(CPU, RAM, storage, graphicsCard, false);
    }

    public Computer(String CPU, int RAM, int storage, boolean graphicsCard, boolean wifi) {
        this.CPU = CPU;
        this.RAM = RAM;
        this.storage = storage;
        this.graphicsCard = graphicsCard;
        this.wifi = wifi;
    }
}
```

#### ✓ Problems:

- Hard to remember the order of parameters.
- Hard to extend when more optional fields are added.
- Code is **ugly and error-prone**.

### 2 The Problem: Setters

```
java

Computer comp = new Computer();
comp.setCPU("Intel i9");
comp.setRAM(32);
comp.setStorage(1024);
```

### ✓ Problems:

- Object is **mutable** → unsafe in multithreading.
- No guarantee that required fields are set → can create invalid objects.
- Chain of setters can be confusing if many fields exist.

## 3 Builder Approach

Builder solves both problems:

```
java

Computer comp = new Computer.ComputerBuilder("Intel i9", 32)
    .setStorage(1024)
    .setGraphicsCard(true)
    .setWifi(true)
    .build();
```

### ✓ Advantages:

- Required fields in **constructor of builder** (CPU, RAM in this case).
- Optional fields using **methods** (setStorage, setGraphicsCard).
- **Immutable object** once built.
- Code is **readable and maintainable**.

## 4 Advanced Builder Structure

Sometimes, you use **Director + Builder** in complex scenarios:

```
java

// Builder interface
interface HouseBuilder {
    void buildWalls();
    void buildRoof();
    void buildGarage();
    House getHouse();
}

// Concrete Builder
class ConcreteHouseBuilder implements HouseBuilder {
    private House house = new House();
    public void buildWalls() { house.setWalls("Concrete walls"); }
    public void buildRoof() { house.setRoof("Tile roof"); }
    public void buildGarage() { house.setGarage("2 car garage"); }
    public House getHouse() { return house; }
}

// Director
```

```

class HouseDirector {
    private HouseBuilder builder;
    public HouseDirector(HouseBuilder builder) { this.builder = builder; }
    public House construct() {
        builder.buildWalls();
        builder.buildRoof();
        builder.buildGarage();
        return builder.getHouse();
    }
}

// Usage
HouseBuilder builder = new ConcreteHouseBuilder();
HouseDirector director = new HouseDirector(builder);
House house = director.construct();

```

### Benefits:

- Separate **construction logic** (Director) from **representation** (Builder).
- Very useful in **complex real-world object creation** (e.g., furniture, vehicles, multi-step setup).

## 5 Points to Remember

- **Telescoping constructors** → messy, hard to read, hard to maintain.
- **Setters** → mutable, error-prone, unsafe for required fields.
- **Builder** → clear, readable, maintains immutability, scalable.
- **Director** → optional, only for very complex multi-step builds.

### Mistakes to avoid:

- Overusing Director → simple objects don't need it.
- Forgetting to make object immutable → defeats the pattern purpose.
- Ignoring required fields → always enforce required fields in builder constructor.

## 6 Real-world Use Cases

- **StringBuilder** in Java (name literally contains Builder).
- **HttpRequest.Builder** in Java 11.
- Complex **order or invoice objects** in enterprise applications.
- **GUI component creation** in JavaFX or Swing.

If you want, we can move to **Level 3**, where we'll explore **Builder Pattern with Inheritance & Polymorphism**, which is very useful for enterprise-grade objects.

Do you want me to go to **Level 3**?



next

Awesome! Let's level up to **Level 3.** 🎉

---

## Level 3: Builder Pattern with Inheritance & Polymorphism

**Goal:** Learn how to implement the Builder pattern when classes are in an inheritance hierarchy, a common real-world scenario.

---

### 1 The Problem

Suppose you have a **Vehicle** superclass and subclasses like **Car** and **Bike**. Both have **common and unique fields**:

```
java

class Vehicle {
    protected String engine;
    protected int wheels;
}

class Car extends Vehicle {
    private boolean airConditioner;
    private boolean sunroof;
}

class Bike extends Vehicle {
    private boolean hasCarrier;
}
```

If we want a Builder pattern here, **simple builder for Vehicle won't handle subclass-specific fields**.

---

### 2 Solution: Recursive Generics

We use **generic builders** to maintain **fluent interface across inheritance**.

```
java

// Base class
class Vehicle {
    protected String engine;
    protected int wheels;

    protected Vehicle(VehicleBuilder<?, ?> builder) {
        this.engine = builder.engine;
        this.wheels = builder.wheels;
    }

    @Override
    public String toString() {
        return "Vehicle [engine=" + engine + ", wheels=" + wheels + "]";
    }
}

// Base builder with recursive generics
static abstract class VehicleBuilder<T extends Vehicle, B extends VehicleBuilder<T, B>> {
    private String engine;
    private int wheels;

    public B setEngine(String engine) {
        this.engine = engine;
        return self();
    }
}
```

```

        public B setWheels(int wheels) {
            this.wheels = wheels;
            return self();
        }

        protected abstract B self();
        public abstract T build();
    }

// Subclass
class Car extends Vehicle {
    private boolean airConditioner;
    private boolean sunroof;

    private Car(CarBuilder builder) {
        super(builder);
        this.airConditioner = builder.airConditioner;
        this.sunroof = builder.sunroof;
    }

    @Override
    public String toString() {
        return super.toString() + ", Car [AC=" + airConditioner + ", Sunroof=" + sunroof + "]";
    }

    public static class CarBuilder extends VehicleBuilder<Car, CarBuilder> {
        private boolean airConditioner;
        private boolean sunroof;

        public CarBuilder setAirConditioner(boolean ac) {
            this.airConditioner = ac;
            return this;
        }

        public CarBuilder setSunroof(boolean sr) {
            this.sunroof = sr;
            return this;
        }

        @Override
        protected CarBuilder self() {
            return this;
        }

        @Override
        public Car build() {
            return new Car(this);
        }
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        Car car = new Car.CarBuilder()
            .setEngine("V8")
            .setWheels(4)
            .setAirConditioner(true)
            .setSunroof(true)
            .build();
        System.out.println(car);
    }
}

```

## 3 Explanation

- **VehicleBuilder<T, B>** → Generic builder for base class, B ensures **method chaining in subclasses**.
- **self()** method → Returns the correct builder type to maintain **fluent interface**.

- **CarBuilder** → Extends base builder and adds **subclass-specific fields**.
  - **build()** → Returns the **final immutable object**.
- 

## 4 Real-world Use Case

- Vehicle manufacturing systems (Cars, Bikes, Trucks).
  - GUI frameworks with **base component and extended components**.
  - Complex configuration objects with **common and specialized parameters**.
- 

## 5 Points to Remember

- Use **recursive generics** for inheritance with Builder.
- Always call **super(builder)** in subclass constructor.
- Keep **base fields and subclass fields separate** in builders.
- Maintain **immutable objects**.

 Common mistakes:

- Forgetting **self()** in base builder → breaks fluent interface.
  - Overloading **build()** incorrectly → causes wrong object types.
  - Directly instantiating subclass without builder → defeats pattern purpose.
- 

## ✓ Level 3 Hint:

Builder with inheritance is **advanced but powerful**. Think of **base configuration + optional advanced configuration** scenario — like creating multiple flavors of a product in an enterprise system.

---

If you want, we can go to **Level 4**, where we'll explore **Builder Pattern with Director & Stepwise Construction**, often used in **multi-step real-world processes**.

Do you want me to proceed to **Level 4**?



i want actual builder pattern example

Perfect! Let's do a **real-world, actual Builder Pattern example** that mimics something you'd see in production code.

We'll build a "**Computer Configuration Builder**" — something companies like Dell or HP might use to let customers configure PCs online. 

---

## Actual Builder Pattern Example: Computer Configurator

## 1 Product Class (Computer)

```

java

public class Computer {
    // Required parameters
    private String CPU;
    private int RAM;

    // Optional parameters
    private int storage;
    private boolean graphicsCard;
    private boolean wifi;
    private boolean bluetooth;

    // Private constructor
    private Computer(ComputerBuilder builder) {
        this.CPU = builder.CPU;
        this.RAM = builder.RAM;
        this.storage = builder.storage;
        this.graphicsCard = builder.graphicsCard;
        this.wifi = builder.wifi;
        this.bluetooth = builder.bluetooth;
    }

    @Override
    public String toString() {
        return "Computer [CPU=" + CPU + ", RAM=" + RAM +
            ", storage=" + storage + "GB, graphicsCard=" + graphicsCard +
            ", wifi=" + wifi + ", bluetooth=" + bluetooth + "]";
    }
}

// Builder Class
public static class ComputerBuilder {
    // Required
    private String CPU;
    private int RAM;

    // Optional
    private int storage = 256; // default value
    private boolean graphicsCard = false;
    private boolean wifi = false;
    private boolean bluetooth = false;

    public ComputerBuilder(String CPU, int RAM) {
        this.CPU = CPU;
        this.RAM = RAM;
    }

    public ComputerBuilder setStorage(int storage) {
        this.storage = storage;
        return this;
    }

    public ComputerBuilder setGraphicsCard(boolean graphicsCard) {
        this.graphicsCard = graphicsCard;
        return this;
    }

    public ComputerBuilder setWifi(boolean wifi) {
        this.wifi = wifi;
        return this;
    }

    public ComputerBuilder setBluetooth(boolean bluetooth) {
        this.bluetooth = bluetooth;
        return this;
    }

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

```

```
}
```

## 2 Usage

```
java

public class Main {
    public static void main(String[] args) {
        Computer myComputer = new Computer.ComputerBuilder("Intel i9", 32)
            .setStorage(1024)
            .setGraphicsCard(true)
            .setWifi(true)
            .setBluetooth(true)
            .build();

        System.out.println(myComputer);

        // Another configuration
        Computer officeComputer = new Computer.ComputerBuilder("Intel i5", 16)
            .setStorage(512)
            .build();

        System.out.println(officeComputer);
    }
}
```

### Output:

```
pgsql
```

```
Computer [CPU=Intel i9, RAM=32, storage=1024GB, graphicsCard=true, wifi=true, bluetooth=true]
Computer [CPU=Intel i5, RAM=16, storage=512GB, graphicsCard=false, wifi=false, bluetooth=false]
```

## 3 Why This is “Actual Builder”

1. **Real-world fields:** CPU, RAM, storage, GPU, WiFi, Bluetooth.
2. **Optional parameters with defaults** → Users can skip setting them.
3. **Immutable object** → Once `Computer` is built, you cannot change it.
4. **Fluent API** → Easy chaining `.setStorage()` `.setWifi()` etc.
5. **Scalable** → Easy to add more options without changing existing constructors.

## 4 Real-world Use Cases

- PC configurators (Dell, HP, Lenovo).
- Pizza builders in online ordering apps.
- Building HTTP requests (`HttpRequest.Builder`) or API payloads.
- Constructing complex domain objects in banking, insurance, e-commerce.

## 5 Points to Remember

- Required fields go in **builder constructor**.
- Optional fields have **setter-style methods** returning **this**.
- Final object is built using **.build()**.
- Don't make setters in the actual object class → maintain **immutability**.

**⚠️ Usual mistakes:**

- Forgetting **return this** in builder → breaks chaining.
- Making the object mutable → defeats Builder purpose.
- Adding too many optional fields → consider splitting into **sub-builders**.

If you want, I can make a **Level 4 “Ultimate Real-World Example”** where **Director + multiple builders + steps** simulate a full **online PC ordering system** with multi-step configuration — this is what enterprise apps actually do.

Do you want me to do that?



explain this code properly how its working..

Absolutely! Let's break down the **Computer Builder example** step by step so you fully understand **how it works internally and why it's structured this way**.

## 1 The Product Class: Computer

```
java

public class Computer {
    private String CPU;
    private int RAM;
    private int storage;
    private boolean graphicsCard;
    private boolean wifi;
    private boolean bluetooth;

    private Computer(ComputerBuilder builder) {
        this.CPU = builder.CPU;
        this.RAM = builder.RAM;
        this.storage = builder.storage;
        this.graphicsCard = builder.graphicsCard;
        this.wifi = builder.wifi;
        this.bluetooth = builder.bluetooth;
    }
}
```

## Explanation

- **Computer** is the **final product** we want to create.
- Fields are **private** → ensures **encapsulation**.

- **Required fields:** CPU, RAM.
  - **Optional fields:** storage, graphicsCard, wifi, bluetooth.
  - **Private constructor:**
    - Forces the user to **use the builder**.
    - Takes a **ComputerBuilder** object and copies its data into the **Computer** object.
  - After construction, the object is **immutable** — no setters in **Computer**.
- 

## 2 The Builder Class: ComputerBuilder

```
java

public static class ComputerBuilder {
    private String CPU;
    private int RAM;
    private int storage = 256; // default value
    private boolean graphicsCard = false;
    private boolean wifi = false;
    private boolean bluetooth = false;

    public ComputerBuilder(String CPU, int RAM) {
        this.CPU = CPU;
        this.RAM = RAM;
    }

    public ComputerBuilder setStorage(int storage) { this.storage = storage; return this; }
    public ComputerBuilder setGraphicsCard(boolean graphicsCard) { this.graphicsCard = graphicsCard; return this; }
    public ComputerBuilder setWifi(boolean wifi) { this.wifi = wifi; return this; }
    public ComputerBuilder setBluetooth(boolean bluetooth) { this.bluetooth = bluetooth; return this; }

    public Computer build() { return new Computer(this); }
}
```

## Explanation

### 1. Fields in Builder:

- Same as the **Computer** class.
- Optional fields can have **default values** (**storage=256**, **graphicsCard=false**, etc.).

### 2. Constructor for required fields:

```
java

public ComputerBuilder(String CPU, int RAM)
```

- Forces the builder to always have these required fields.
- Prevents invalid **Computer** objects.

### 3. Setter methods for optional fields:

```
java
```

```
public ComputerBuilder setStorage(int storage) { ... return this; }
```

- Each method **returns this**, allowing **method chaining** (fluent interface).
- Example: `.setStorage(1024).setWifi(true).setBluetooth(true)`

#### 4. `build()` method:

- Final step.
- Creates a **new Computer object** using the private constructor and passes **this** builder instance.
- Returns the fully constructed **immutable Computer object**.

## 3 Usage in Main Class

```
java
```

```
Computer myComputer = new Computer.ComputerBuilder("Intel i9", 32)
    .setStorage(1024)
    .setGraphicsCard(true)
    .setWifi(true)
    .setBluetooth(true)
    .build();
```

## Step-by-Step Flow

### 1. Create Builder:

```
java
```

```
new Computer.ComputerBuilder("Intel i9", 32)
```

- **CPU** and **RAM** are required → set immediately.
- Builder object now has default values for optional fields.

### 2. Set optional parameters (fluent chaining):

```
java
```

```
.setStorage(1024).setGraphicsCard(true).setWifi(true).setBluetooth(true)
```

- Each call updates the builder's internal fields.
- Returns the same builder object to chain next method.

### 3. Build the final object:

```
java
```

```
.build()
```

- Calls **Computer** constructor with this builder object.

- Copies all fields from builder → **Computer** object created.
- Builder can now be discarded; object is immutable.

#### 4. Result:

```
nginx
Computer [CPU=Intel i9, RAM=32, storage=1024GB, graphicsCard=true, wifi=true,
bluetooth=true]
```

## 4 Key Concepts Highlighted

Concept	How it's implemented here
<b>Immutability</b>	No setters in <b>Computer</b> , all fields <b>private final</b> .
<b>Fluent interface</b>	Each builder setter returns <b>this</b> for chaining.
<b>Required vs Optional</b>	Required fields in builder constructor, optional fields in setters.
<b>Encapsulation</b>	Fields private, builder handles construction.
<b>Scalability</b>	Easy to add more optional fields without touching existing constructors.

## 5 Common Mistakes to Avoid

1. Forgetting `return this` in builder setter → breaks chaining.
2. Making **Computer** constructor public → allows bypassing builder.
3. Adding setters in **Computer** → object becomes mutable.
4. Not enforcing required fields in builder → can create invalid objects.

### 💡 Analogy:

Think of **ComputerBuilder** as a **form you fill step by step**.

- Required fields → name and email.
- Optional fields → extra toppings.
- `build()` → submit form → you get the final product.

If you want, I can draw a **diagram showing the flow of Builder** → **Builder fields** → **Computer object** so you can **visualize exactly how it works internally**.

Do you want me to do that?



explain about this object creation

Absolutely! Let's dive deep into **how the Computer object is actually created step by step** in the Builder pattern. We'll look at the **object creation process internally** to understand how data flows from the

builder to the final object.

---

## 1 Step 1: Builder Object Creation

java

```
Computer.ComputerBuilder builder = new Computer.ComputerBuilder("Intel i9", 32);
```

- A new **Builder object** is created in memory.
- Required fields (**CPU** and **RAM**) are initialized immediately in the builder's constructor.
- Optional fields get **default values**:
  - **storage = 256**
  - **graphicsCard = false**
  - **wifi = false**
  - **bluetooth = false**

**Memory view:**

makefile

```
builder:
CPU = "Intel i9"
RAM = 32
storage = 256
graphicsCard = false
wifi = false
bluetooth = false
```

✓ At this point, **no Computer object exists yet**. We are only preparing the builder.

---

## 2 Step 2: Setting Optional Fields

java

```
builder.setStorage(1024)
    .setGraphicsCard(true)
    .setWifi(true)
    .setBluetooth(true);
```

- Each setter **modifies the builder's internal fields**.
- Each setter **returns this**, so calls can be **chained**.
- The builder **accumulates all the data** necessary to create the final object.

**Memory view after setters:**

makefile

```
builder:
CPU = "Intel i9"
```

```
RAM = 32
storage = 1024
graphicsCard = true
wifi = true
bluetooth = true
```

✓ Still, **no Computer object exists**. Builder is like a “template” or “draft” for the object.

## 3 Step 3: Building the Final Object

```
java

Computer myComputer = builder.build();
```

### What happens internally:

1. **build()** calls the private **Computer** constructor:

```
java

return new Computer(this);
```

2. Inside the **Computer** constructor:

```
java

private Computer(ComputerBuilder builder) {
    this.CPU = builder.CPU;
    this.RAM = builder.RAM;
    this.storage = builder.storage;
    this.graphicsCard = builder.graphicsCard;
    this.wifi = builder.wifi;
    this.bluetooth = builder.bluetooth;
}
```

- All fields are copied from builder → **Computer object**.
- **Computer** now has its **own independent copy** of the data.
- Builder can now be discarded; object is immutable.

### Memory view:

```
makefile

myComputer:
CPU = "Intel i9"
RAM = 32
storage = 1024
graphicsCard = true
wifi = true
bluetooth = true
```

✓ The **final Computer object** is ready.

## 4 Step 4: Using the Object

```
java
System.out.println(myComputer);
```

- Calls `toString()` of `Computer`.
- Reads **its own fields** and prints a summary of the configuration.

## 5 Why This Way of Object Creation Is Useful

- **Step-by-step construction** → easy to set optional fields.
- **Immutable object** → thread-safe, reliable.
- **Readable code** → you can see exactly what options are set.
- **Scalable** → can add more options in the builder without touching `Computer` class.
- **Avoids constructor overloading hell** → no need for multiple confusing constructors.

## 6 Analogy

Think of object creation like **ordering a custom pizza online**:

1. **Builder object** = the order form.
2. **Set optional fields** = choose toppings, size, crust.
3. **Build()** = place the order → pizza is cooked.
4. **Computer object** = the final pizza delivered to your table.

If you want, I can make a **step-by-step memory diagram** showing **Builder** → **Builder fields** → **Final Object** with arrows, which will make the **object creation flow visually crystal clear**.

Do you want me to do that?



```
Computer.ComputerBuilder builder= new Computer.ComputerBuilder("intel", 32);
Computer c1=builder.build();
```

explain this way of object creation, how do i say the computer as by which name and computerBuilder

Great question — you want to understand the **naming and roles** when you do this style of object creation:

```
java
```

```
Computer.ComputerBuilder builder = new Computer.ComputerBuilder("intel", 32);
Computer c1 = builder.build();
```

Let's carefully explain it.

---

## 1 Step 1 – Understanding the Classes

- **Computer** → The **final product class** (immutable object you actually want).
- **ComputerBuilder** → A **static nested class** inside **Computer**. It is not an independent class file; it “lives inside” **Computer**.
  - That’s why you must write **Computer.ComputerBuilder** (fully qualified name).
  - This tells Java: “*Go inside the Computer class, and use the nested class ComputerBuilder.*”

So:

- **Computer** → **product**
  - **Computer.ComputerBuilder** → **builder** (the tool used to construct the product)
- 

## 2 Step 2 – Creating a Builder Object

```
java
```

```
Computer.ComputerBuilder builder = new Computer.ComputerBuilder("intel", 32);
```

- You are creating an **instance of the builder**.
- Required fields (**CPU = "intel"**, **RAM = 32**) are immediately set in this step.
- At this point, no **Computer** object exists yet. You only have the “order form” (builder).

👉 Name explanation:

- This variable is called **builder** (because it is an instance of **ComputerBuilder**).
  - Its **type** is **Computer.ComputerBuilder**.
  - Its **job** is to hold temporary values until you call **.build()**.
- 

## 3 Step 3 – Building the Actual Computer

```
java
```

```
Computer c1 = builder.build();
```

- The **build()** method:

- Calls the **private constructor of Computer**.
- Copies all values from the builder into a new **Computer object**.
- Now you have an actual **Computer object** stored in variable **c1**.

👉 Name explanation:

- This variable is called **c1** (because it is an instance of **Computer**).
  - Its **type** is **Computer**.
  - Its **job** is the final, immutable object you can use.
- 

## 4 How to "Say" It in Words

When you explain this code, you can phrase it like this:

- `Computer.ComputerBuilder builder = new Computer.ComputerBuilder("intel", 32);`  
👉 "I am creating a builder object (of type **ComputerBuilder**) to configure a Computer with CPU = Intel and RAM = 32."
  - `Computer c1 = builder.build();`  
👉 "I am using the builder to build the final Computer object, which is now stored in variable **c1**."
- 

## 5 Analogy

Think of it like buying a car:

- **Builder (builder)** = the **order sheet** where you select engine, wheels, color.
  - **Computer (c1)** = the **finished car** delivered to you.
  - Once the car is delivered, you don't need the order sheet anymore.
- 

✓ So, in your code:

- **builder** → is a **builder object** (temporary configuration holder).
  - **c1** → is the actual **Computer object** (final product).
- 

Do you want me to also show you how to do the same in a **one-line way** (without keeping the builder variable separately), like this? 👇

```
java

Computer c1 = new Computer.ComputerBuilder("intel", 32).build();
```

This is usually how it's written in real projects. Want me to explain that version too?



does if I create separate `c2`, it has its own static builder

Good catch  You're asking:

 “If I create another `Computer` (say `c2`), does it have its **own static builder**? Or does the same builder get reused?”

Let's break this down carefully.

## 1 Important: ComputerBuilder is a Static Nested Class

```
java

class Computer {
    public static class ComputerBuilder { ... }
}
```

- `ComputerBuilder` being **static** means it does **not depend on any specific Computer instance**.
- It is like a separate class that just happens to live inside `Computer`.
- Every time you write `new Computer.ComputerBuilder(...)`, you are creating a **new, separate builder object**.

## 2 Example with Two Computers

```
java

Computer.ComputerBuilder builder1 = new Computer.ComputerBuilder("Intel", 32);
Computer c1 = builder1.build();

Computer.ComputerBuilder builder2 = new Computer.ComputerBuilder("AMD", 16);
Computer c2 = builder2.build();
```

- `builder1` is **independent** from `builder2`.
- They don't interfere with each other.
- Each `.build()` call creates its **own Computer object**.

## 3 Is the Builder Shared?

No .

- The **class ComputerBuilder** is static (one definition).
- But the **objects** you create (`builder1`, `builder2`, ...) are separate instances.
- So `c1` and `c2` come from **different builders**.

## 4 Analogy

Think of `ComputerBuilder` as a **blueprint** (the static class).

- Every time you say `new Computer.ComputerBuilder(...)`, you are printing a **fresh order form** based on that blueprint.
  - Each form is **independent**.
  - Submitting each form (`.build()`) gives you a **new Computer**.
- 

 So answer:

If you create `c2`, it will have its **own builder object**. The builder class is static, but each `new Computer.ComputerBuilder(...)` creates a separate instance, so there's no conflict between `c1` and `c2`.

---

Do you want me to **draw a memory diagram** (showing how `builder1`, `builder2`, `c1`, `c2` exist separately in the heap) to make this crystal clear?