**Exercise 3: Implementing the Builder Pattern**

**Scenario:**

You are developing a system to create complex objects such as a Computer with multiple optional parts. Use the Builder Pattern to manage the construction process.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **BuilderPatternExample**.
2. **Define a Product Class:**
   * Create a class **Computer** with attributes like **CPU**, **RAM**, **Storage**, etc.
3. **Implement the Builder Class:**
   * Create a static nested Builder class inside Computer with methods to set each attribute.
   * Provide a **build()** method in the Builder class that returns an instance of Computer.
4. **Implement the Builder Pattern:**
   * Ensure that the **Computer** class has a private constructor that takes the **Builder** as a parameter.
5. **Test the Builder Implementation:**
   * Create a test class to demonstrate the creation of different configurations of Computer using the Builder pattern.

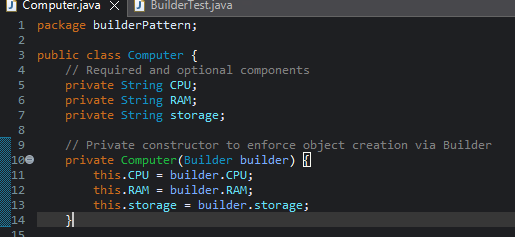
**Solution:**

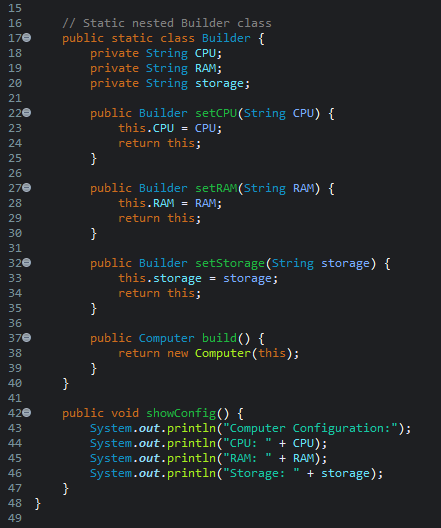
**1. Class Computer (The Product)**

* Represents the complex object you want to build.
* Has attributes: CPU, RAM, and Storage.
* The constructor is private to force object creation through the builder only.

**. Static Nested Class Builder (The Builder)**

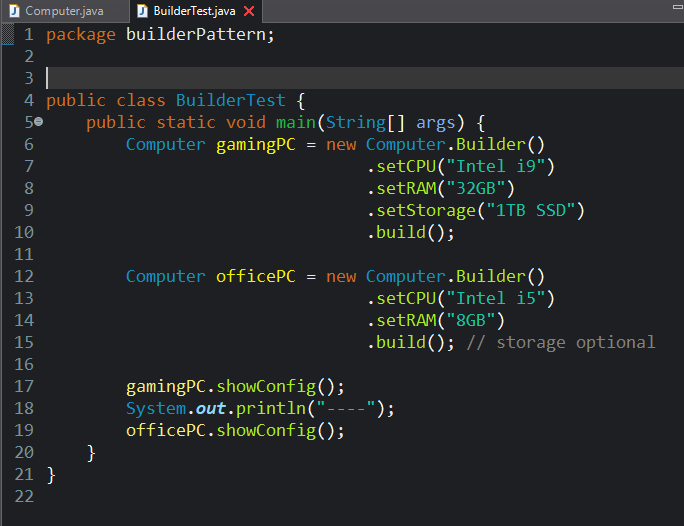
* It's a helper class inside Computer responsible for setting values and constructing the final object.
* Contains the same fields as Computer (e.g., CPU, RAM, Storage).
* Uses chained setter methods (each returns this) for fluent interface style.
* The build() method returns the final Computer object.

****

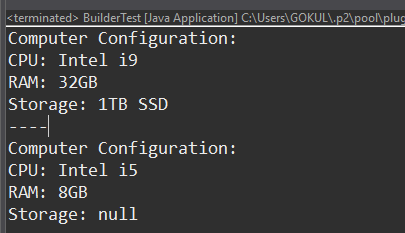


**BuilderTest.java**

* A new Builder object is created.
* The builder methods setCPU, setRAM, setStorage are called in a fluent chain
* .build() creates and returns the Computer object using the values provided.

****

**Output:**

****