**Cognizant Digital Nurture 4.0**

***WEEK-1 Module 2 - Design Patterns and Principles Task Solutions***

**Exercise 1: Implementing the Singleton Pattern** **(Mandatory)**

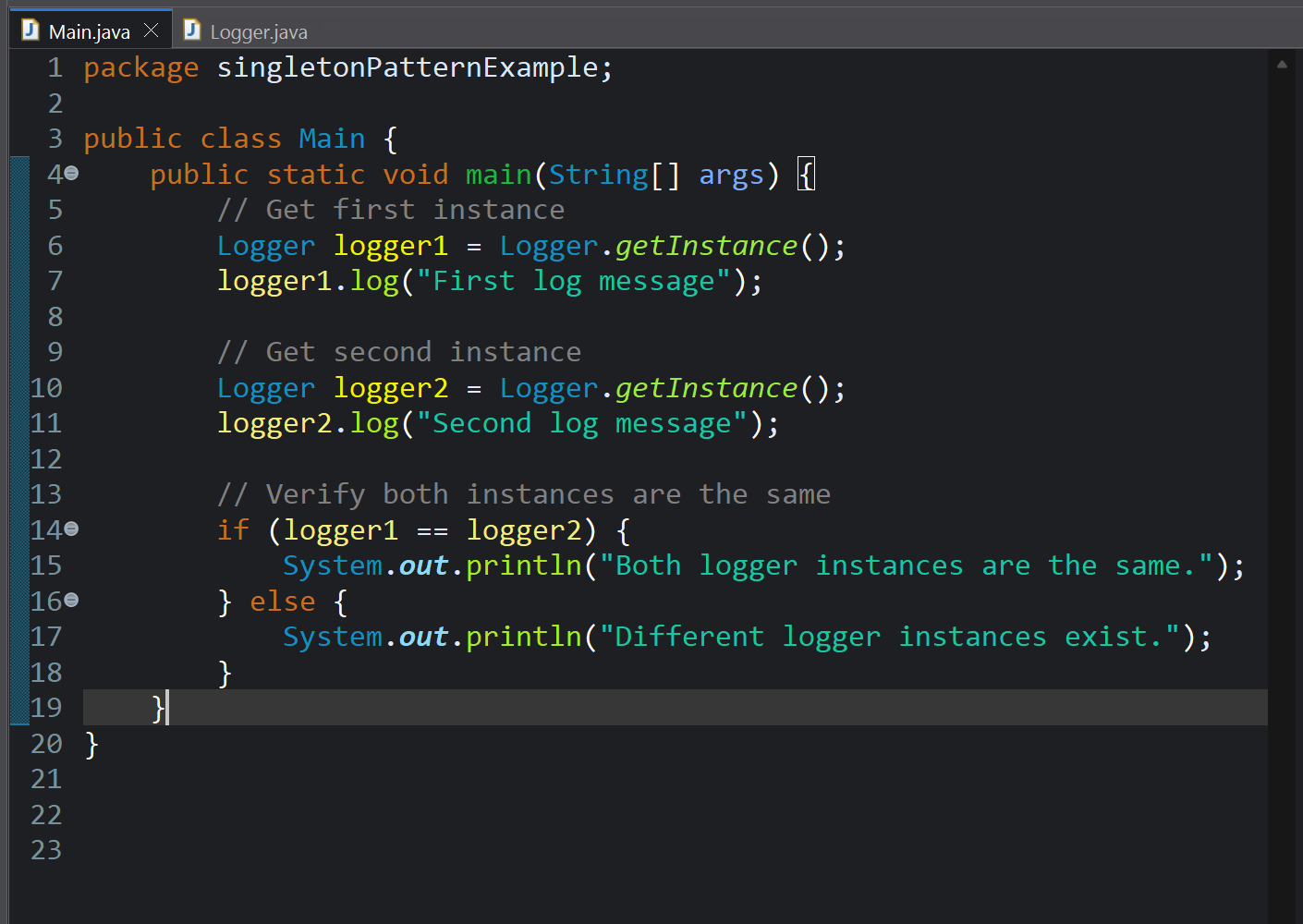
**Scenario:**

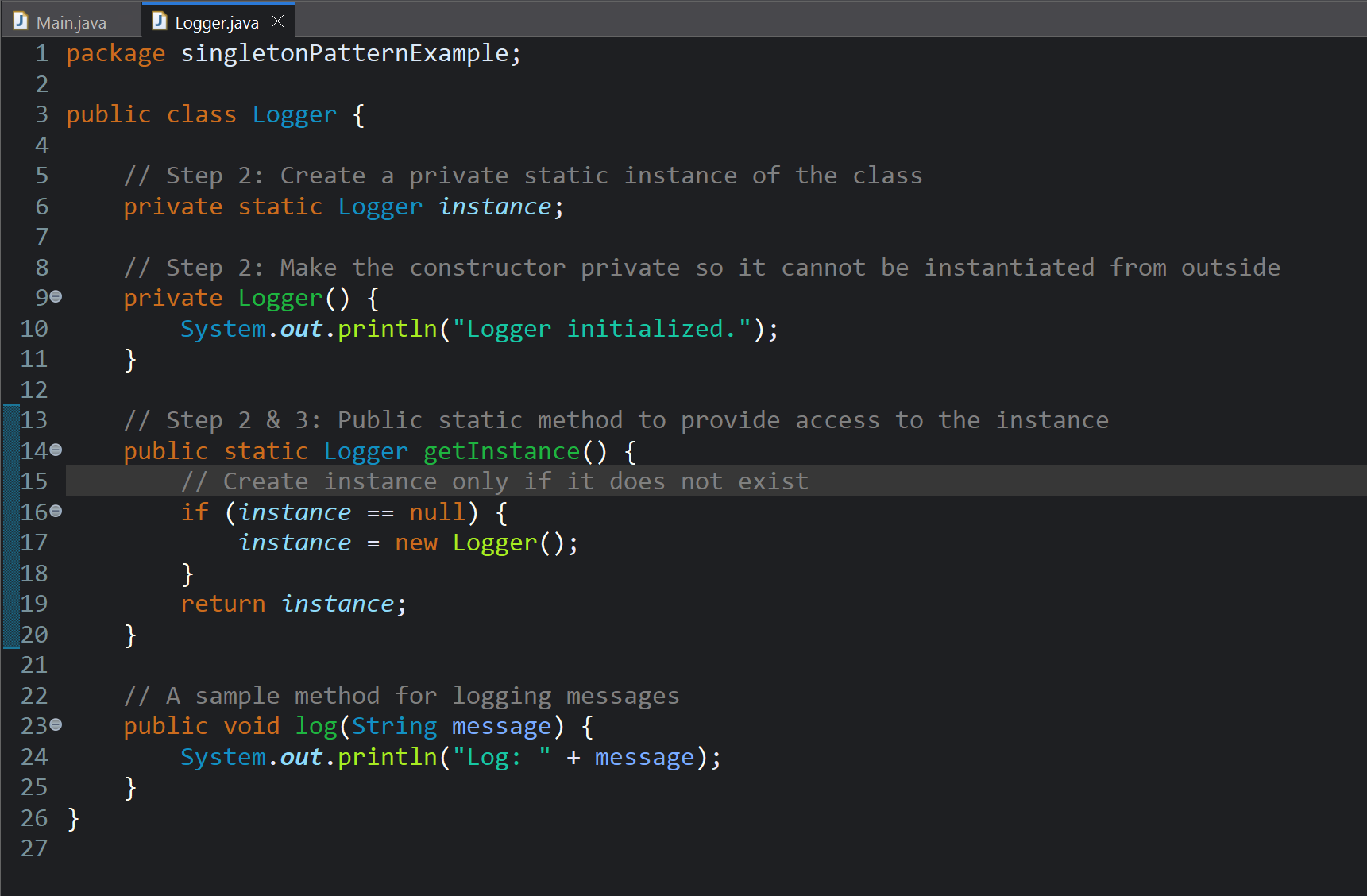
You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

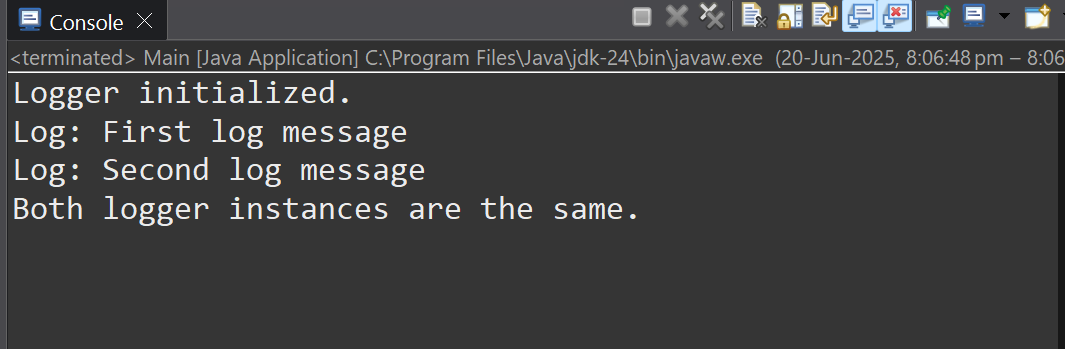
**Singleton Pattern:**

In software development, we use design patterns. One of them is the Singleton Pattern. In the Singleton Pattern, the class can have only one instance. It will provide a global point of access that instance. We generally use the Singleton Pattern to make all parts of a program use the same instance and ensure consistency over the project. It is useful for sharing resources, such as logging services and database connections, with others.

I created a package called “**SingletonPatternExample**” (instead of project I used a package). It follows Singleton design pattern. The class “Logger.java” has private static instance of itself. The constructor of this class is private. The main method gets an instance of Logger class.





Output:  


**Exercise 2: Implementing the** **Factory Method Pattern (Mandatory)**

**Scenario:**

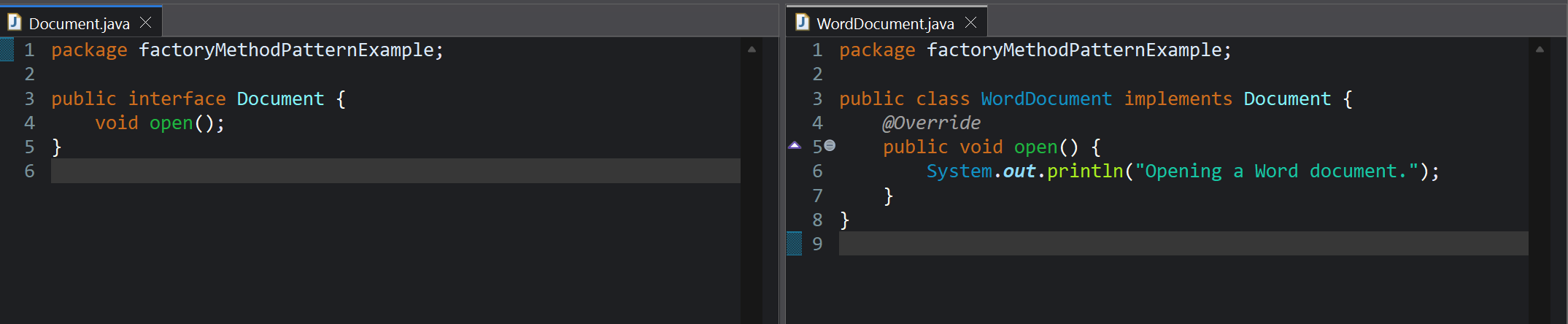
You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

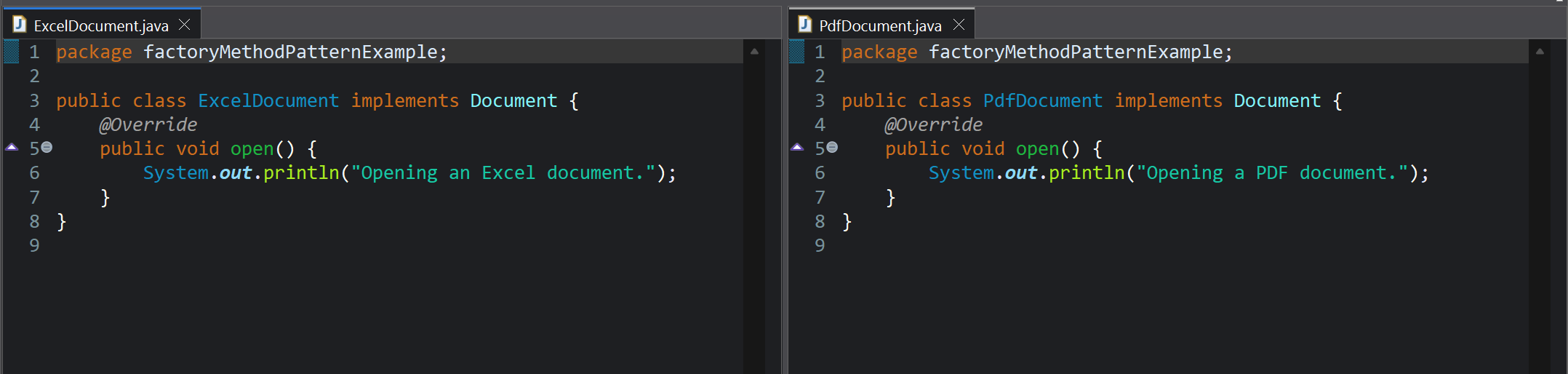
**Factory Method Pattern:**

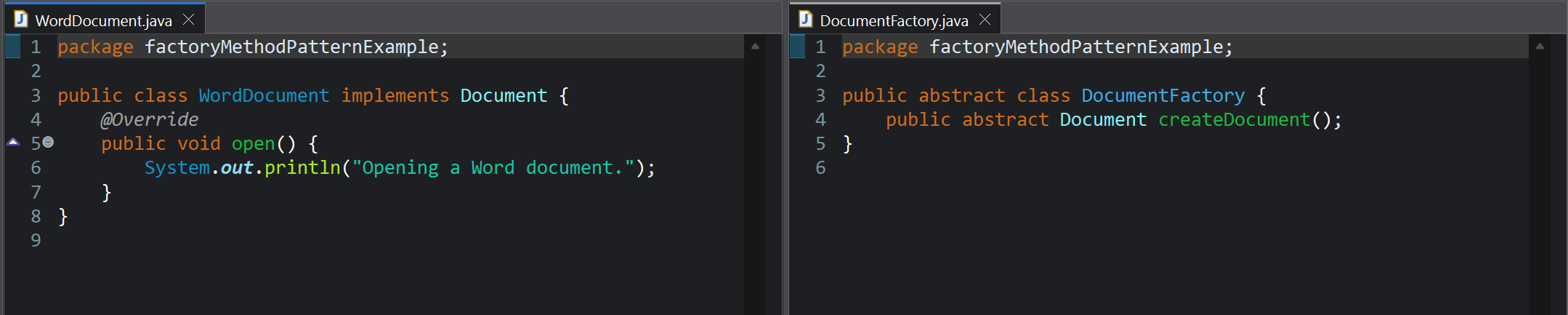
Factory Method Pattern is a type of creational design pattern. It provides different interface to create objects in a superclass. In addition, it also allows the change the type of objects that can be created in a sub class.

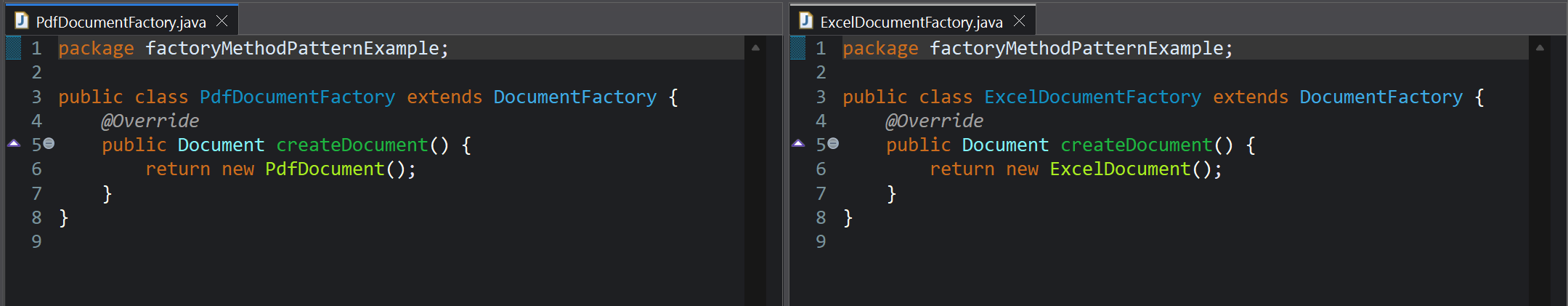
In a case where programmer have no idea about the exact type of object required until the runtime, we use these Factory Method Patterns. It is created when the subclass wants to specify the type of object it creates. It follows Open/Close principle which refers to “open for extension, closed for modification”.

Here for this exercise, I am creating a package (named FactoryMethodPatternExample) only for simplicity. Here, an interface called “Document” is defined to represent all types of documents. I included a method named “open()” which is commonly implemented by all docement types. Three unique classes: “WordDocument, PdfDocument, and ExcelDocument” are created to implement the interface.

The code is as follow:  










**Exercise 3: Implementing the Builder Pattern (Non - Mandatory)**

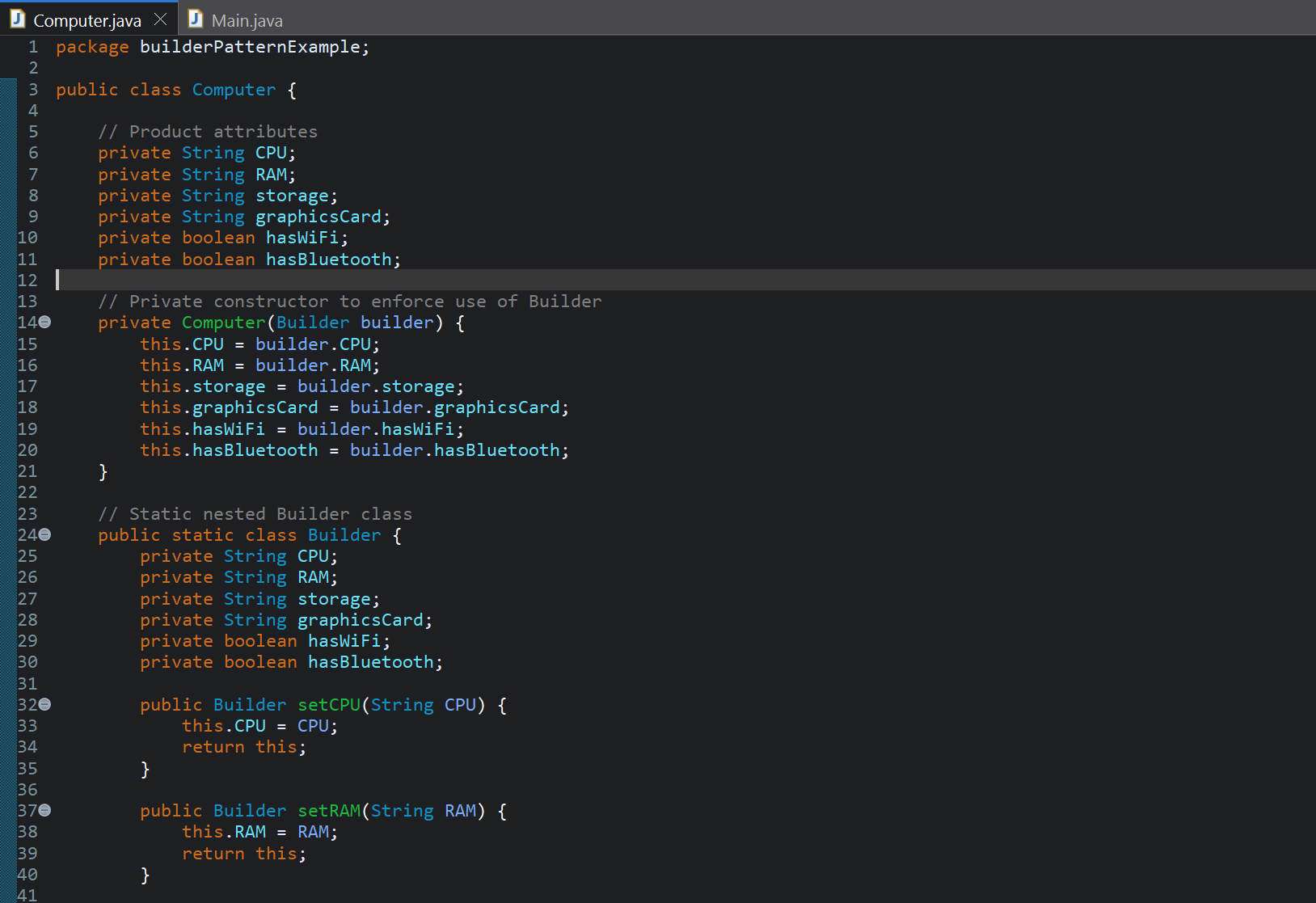
**Scenario:**

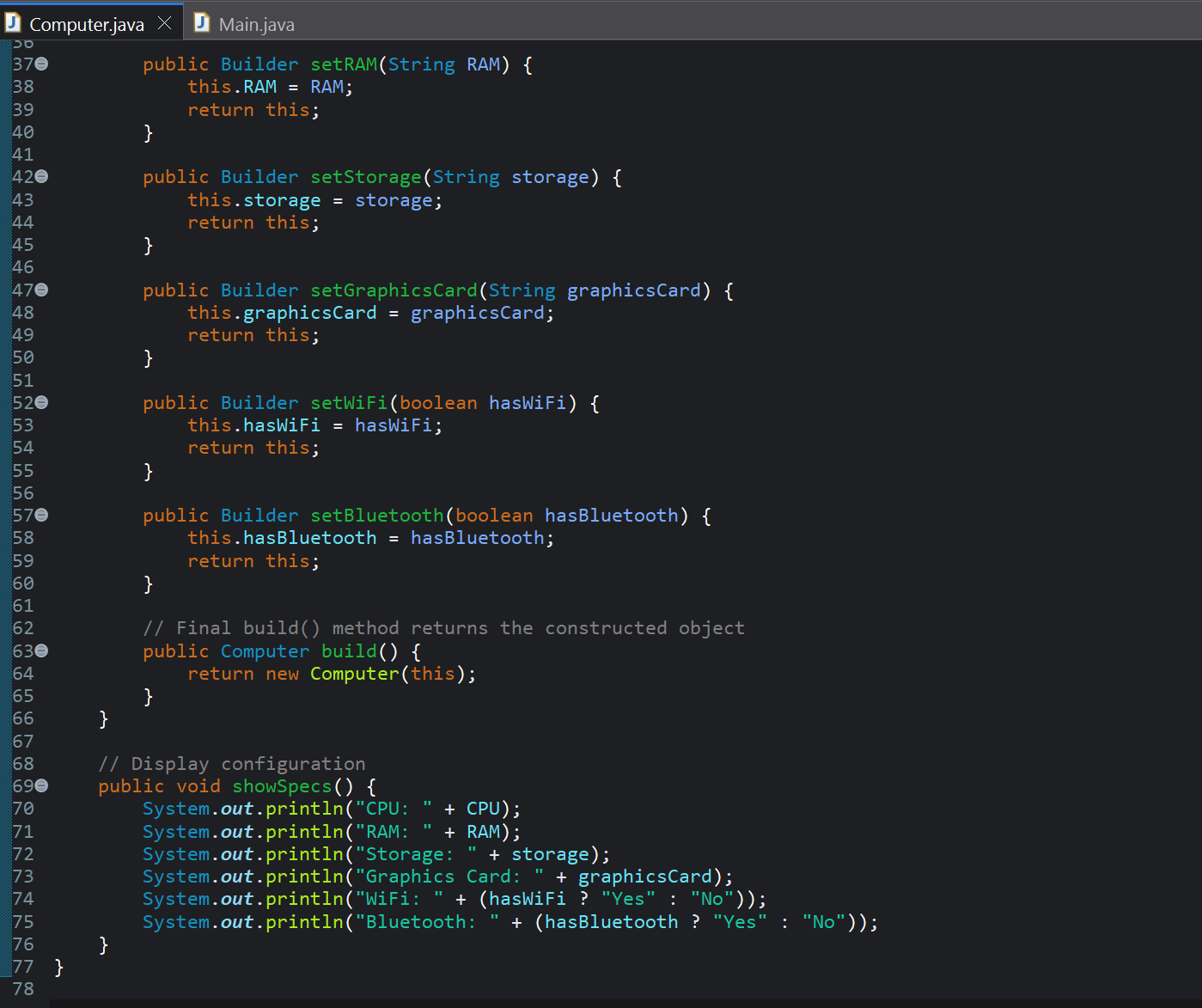
You are developing a system to create complex objects such as a Computer with multiple optional parts. Use the Builder Pattern to manage the construction process.

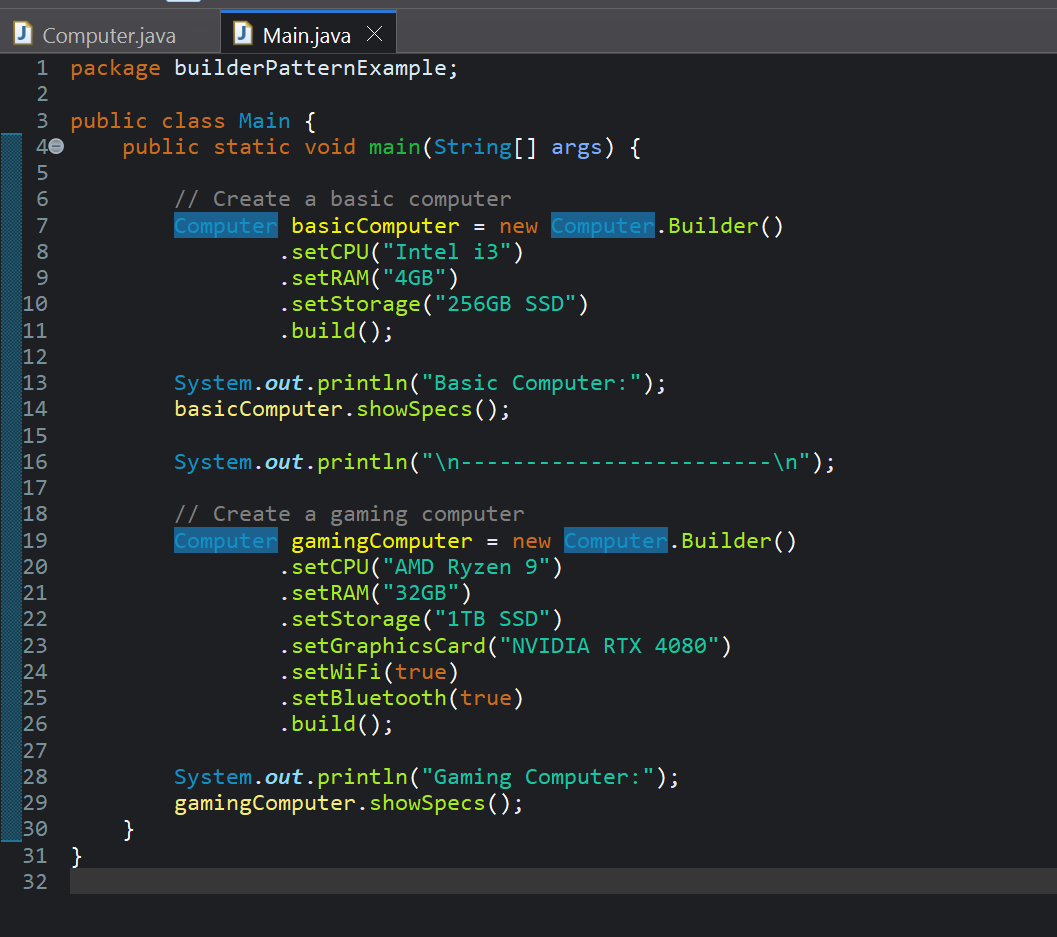
In java, we have **Builder patterns** which is a type of creational design pattern. it is used to construct the object which are complex in step-to-step manner. it is preferred as it allows the same construction process to create other representations by separating the object construction from its actual representation.

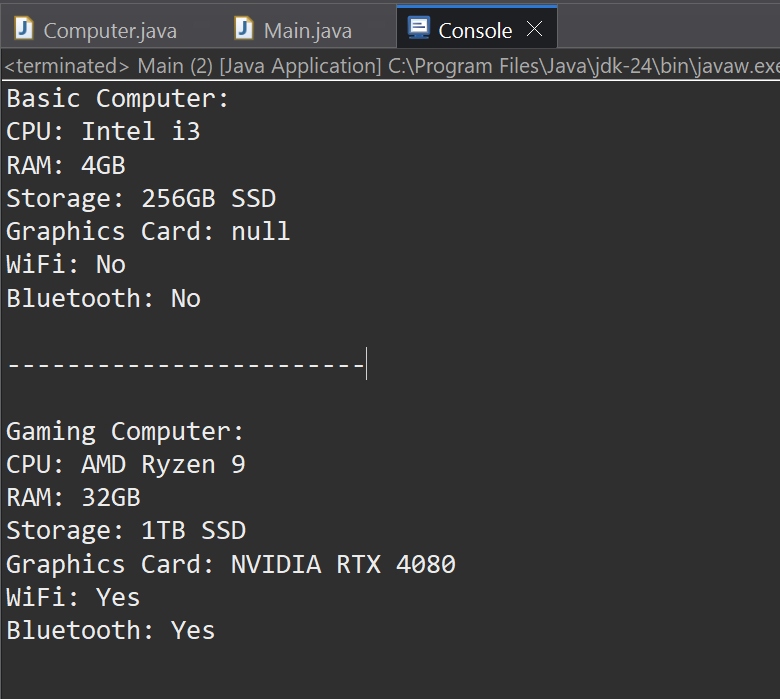
**Builder pattern is preferred**

* When we have many optional patterns for one object.
* When the process of object creation involves multiple steps and is very complex.
* When the priority is more readable and maintainable code.









The Product class "Computer" has both optional and required parts. i have created a static nested Builder class which is used to set values and construct Computer.

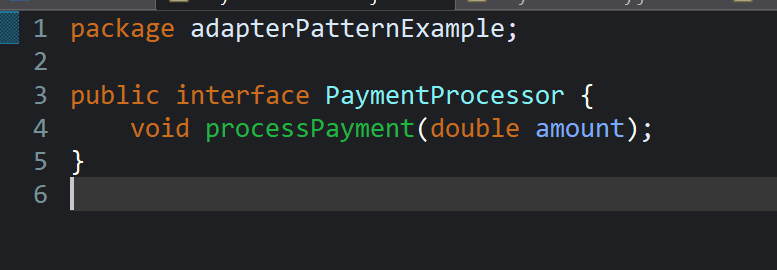
**Exercise 4: Implementing the Adapter Pattern**

**Scenario:**

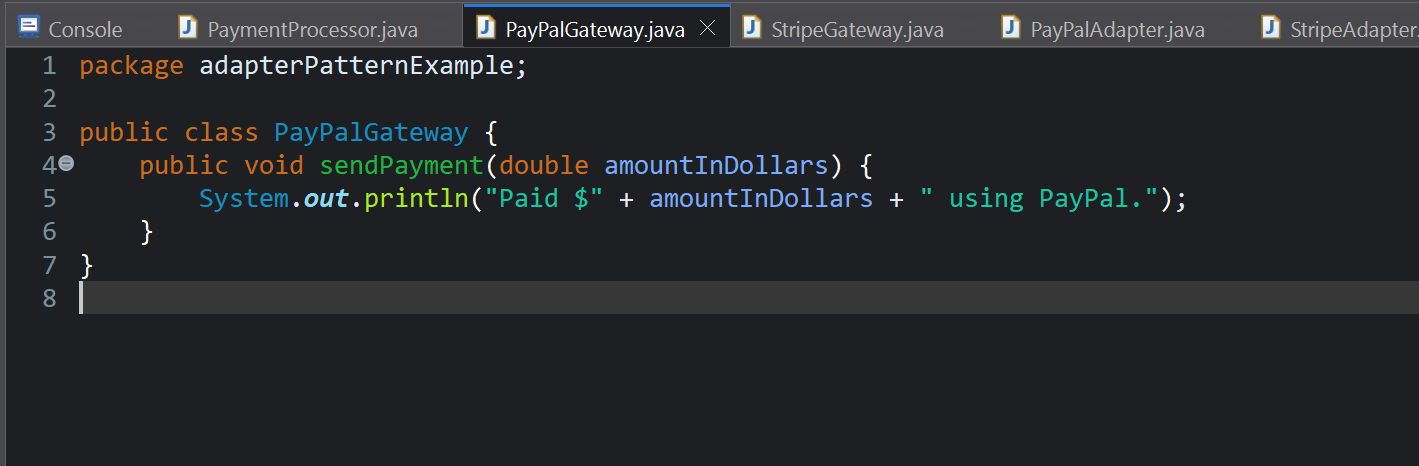
You are developing a payment processing system that needs to integrate with multiple third-party payment gateways with different interfaces. Use the Adapter Pattern to achieve this.

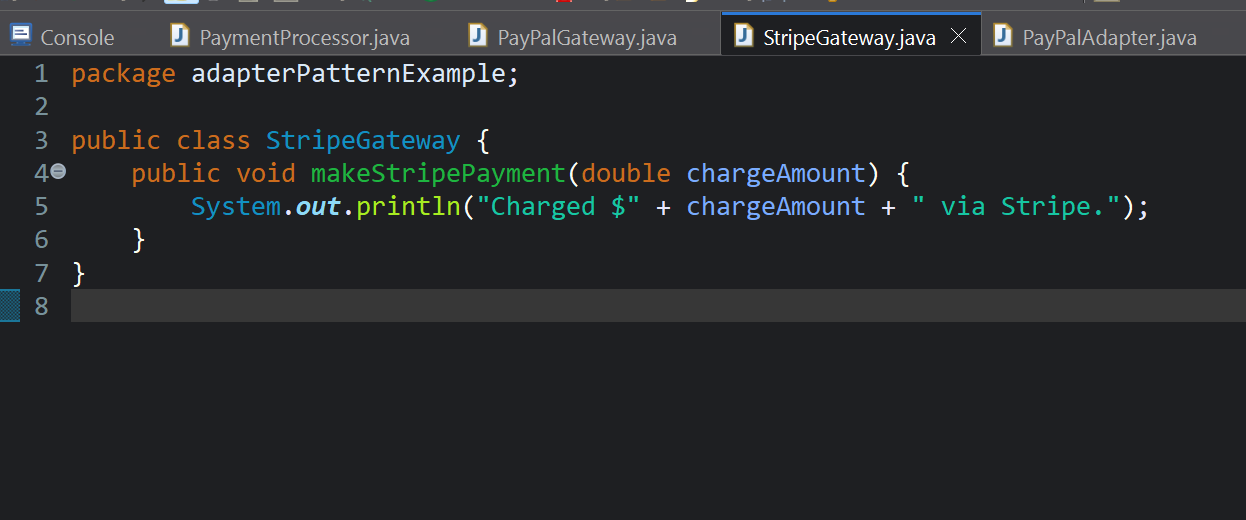
The Adapter Pattern is a type of Structural design pattern used in Java. Through such patterns, we can make objects with incompatible interfaces work together. This pattern acts like a bridge between two interfaces. It converts one interface into another when the client expects it to be.

Created a standard interface PaymentProcessor:



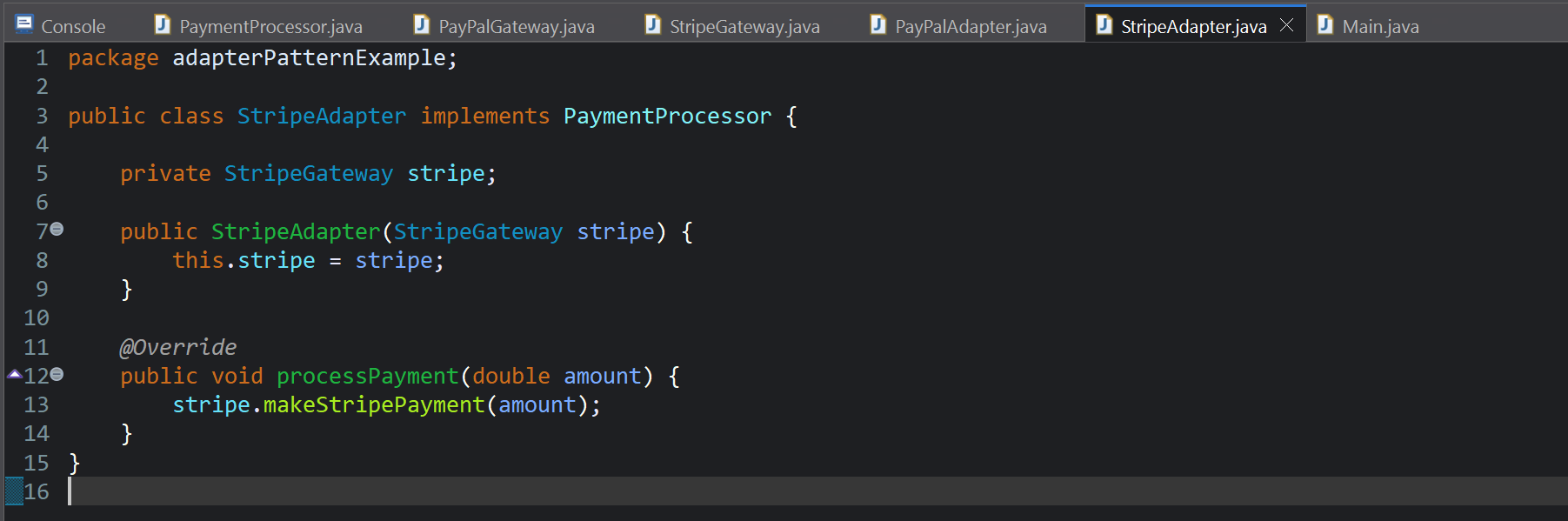
Created 3rd-party gateways: PayPalGateway and StripeGateway



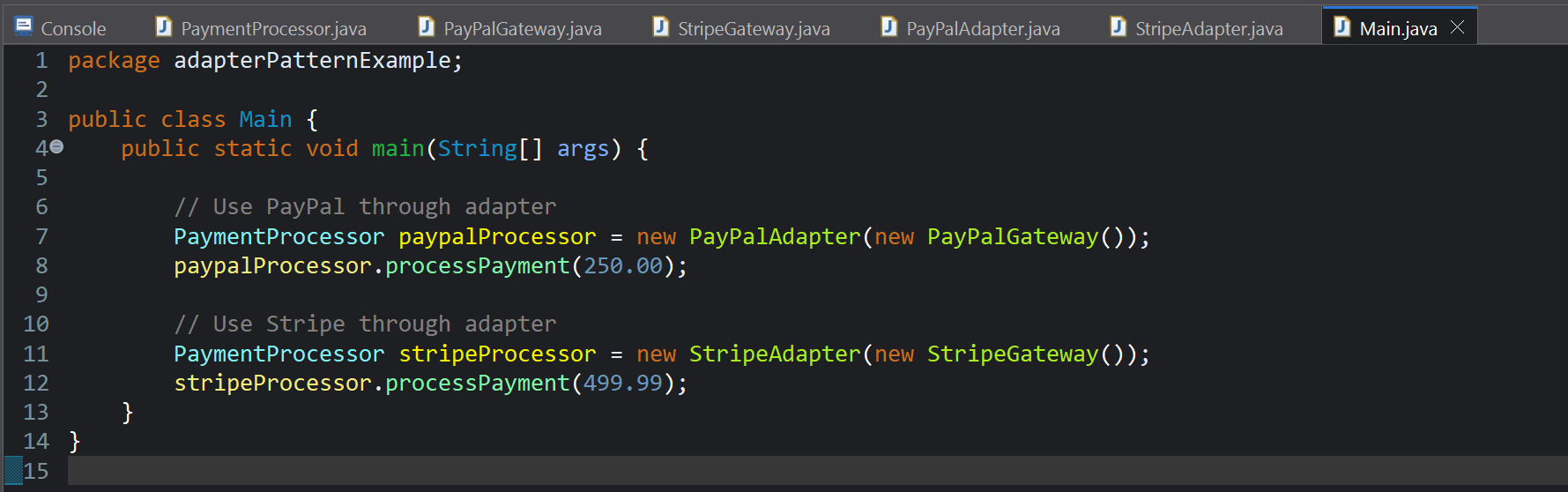


Built adapter classes: PayPalAdapter and StripeAdapter





Tested the logic using Main method:

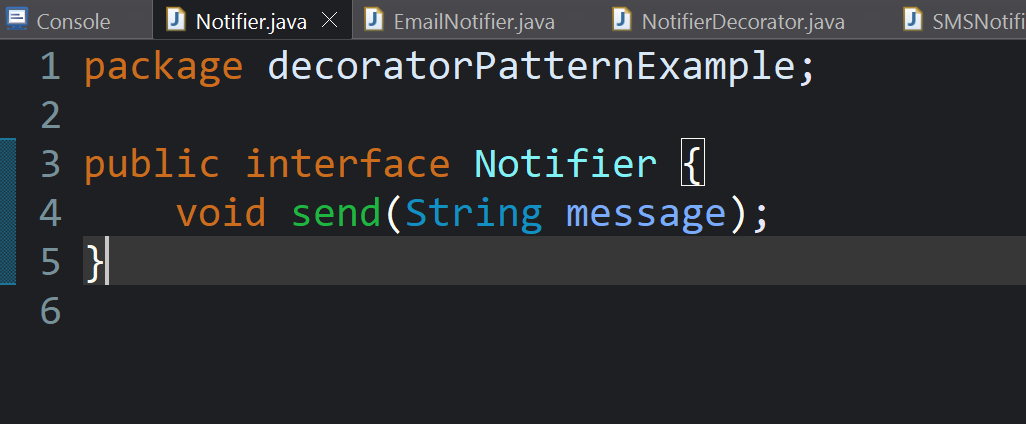


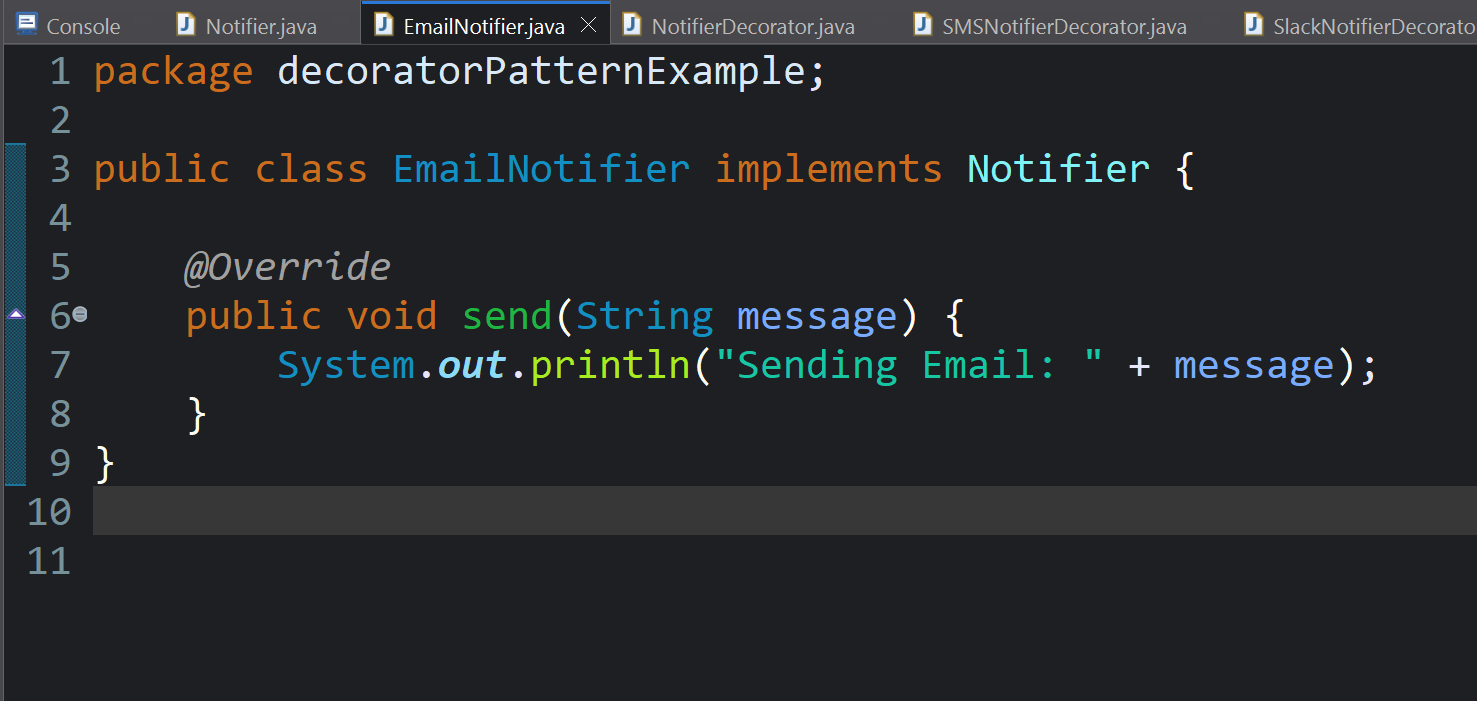
**Exercise 5: Implementing the Decorator Pattern**

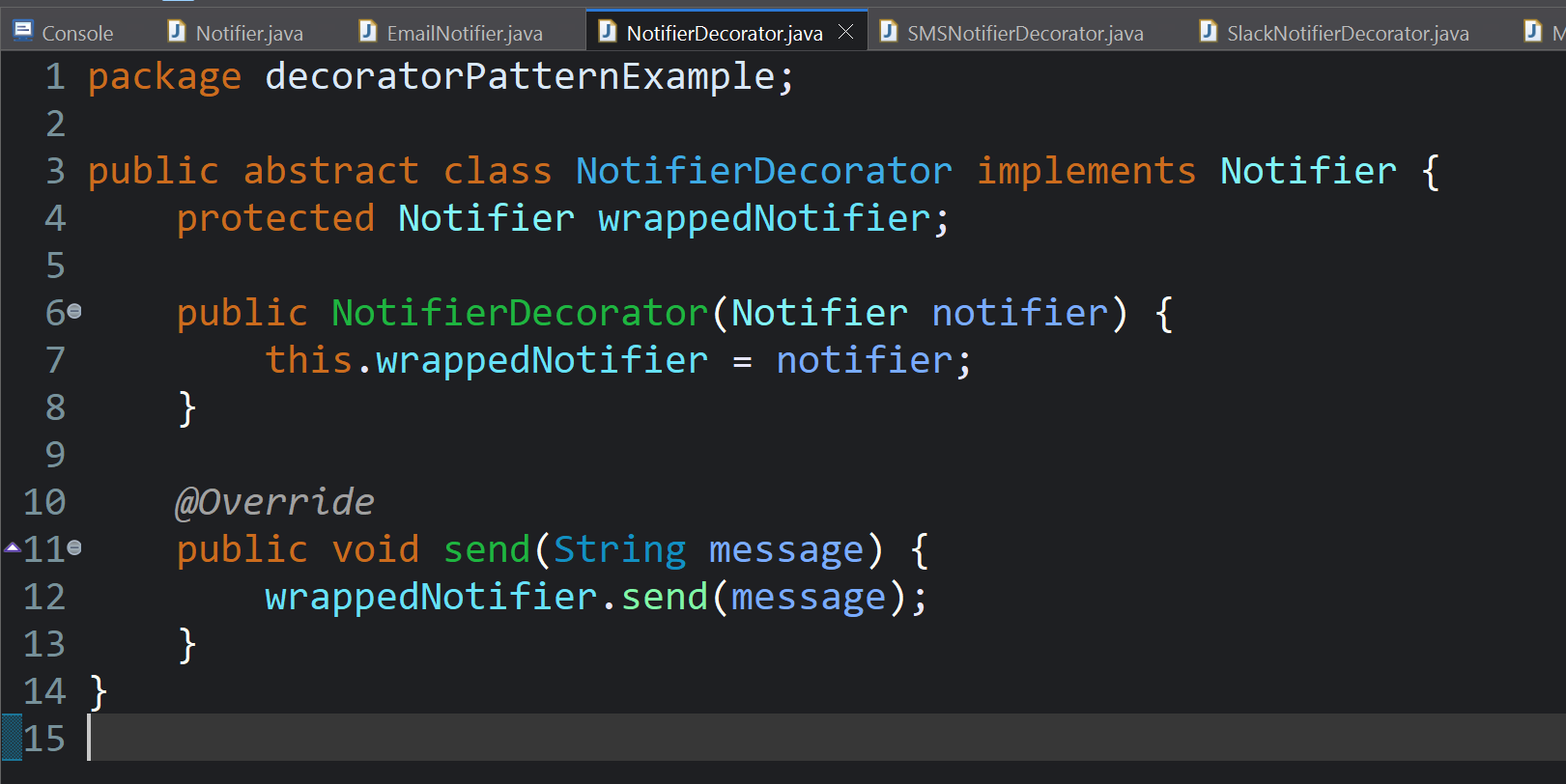
**Scenario:**

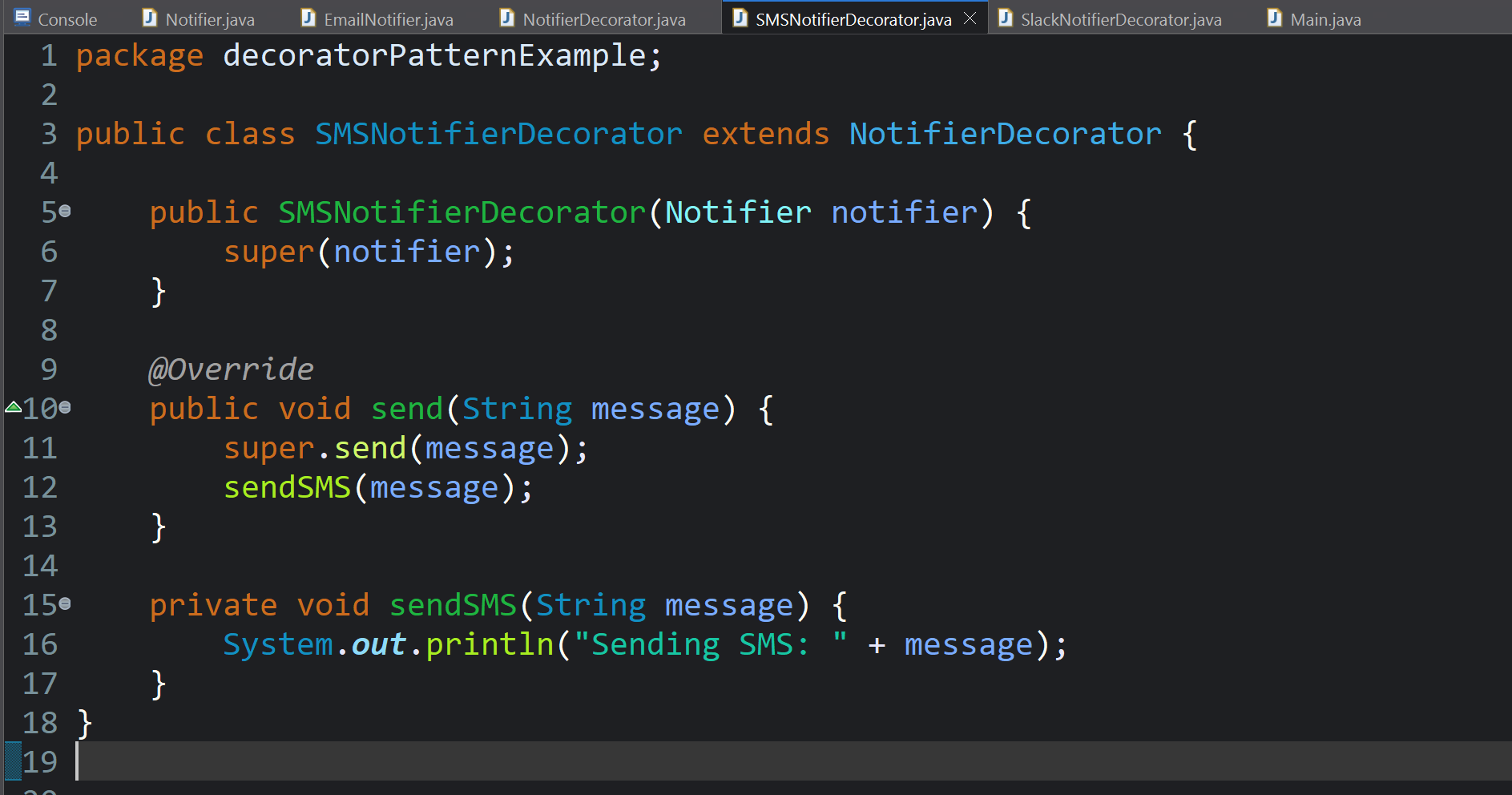
You are developing a notification system where notifications can be sent via multiple channels (e.g., Email, SMS). Use the Decorator Pattern to add functionalities dynamically.

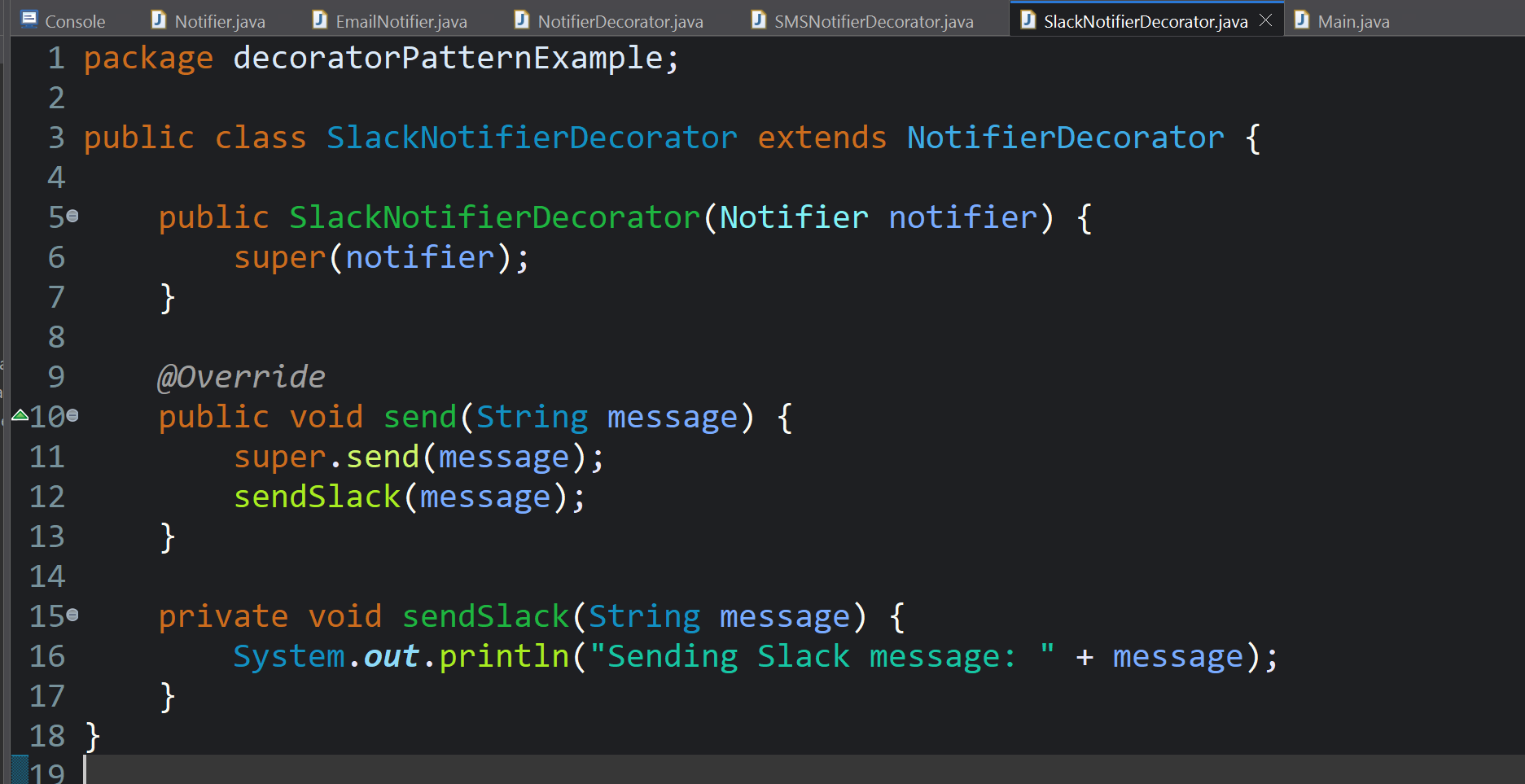
The "Decorator pattern" is also a structural design pattern that can add external behaviour to an individual object dynamically, without actually modifying any class. Here, we add responsibilities additionally to an object at runtime.

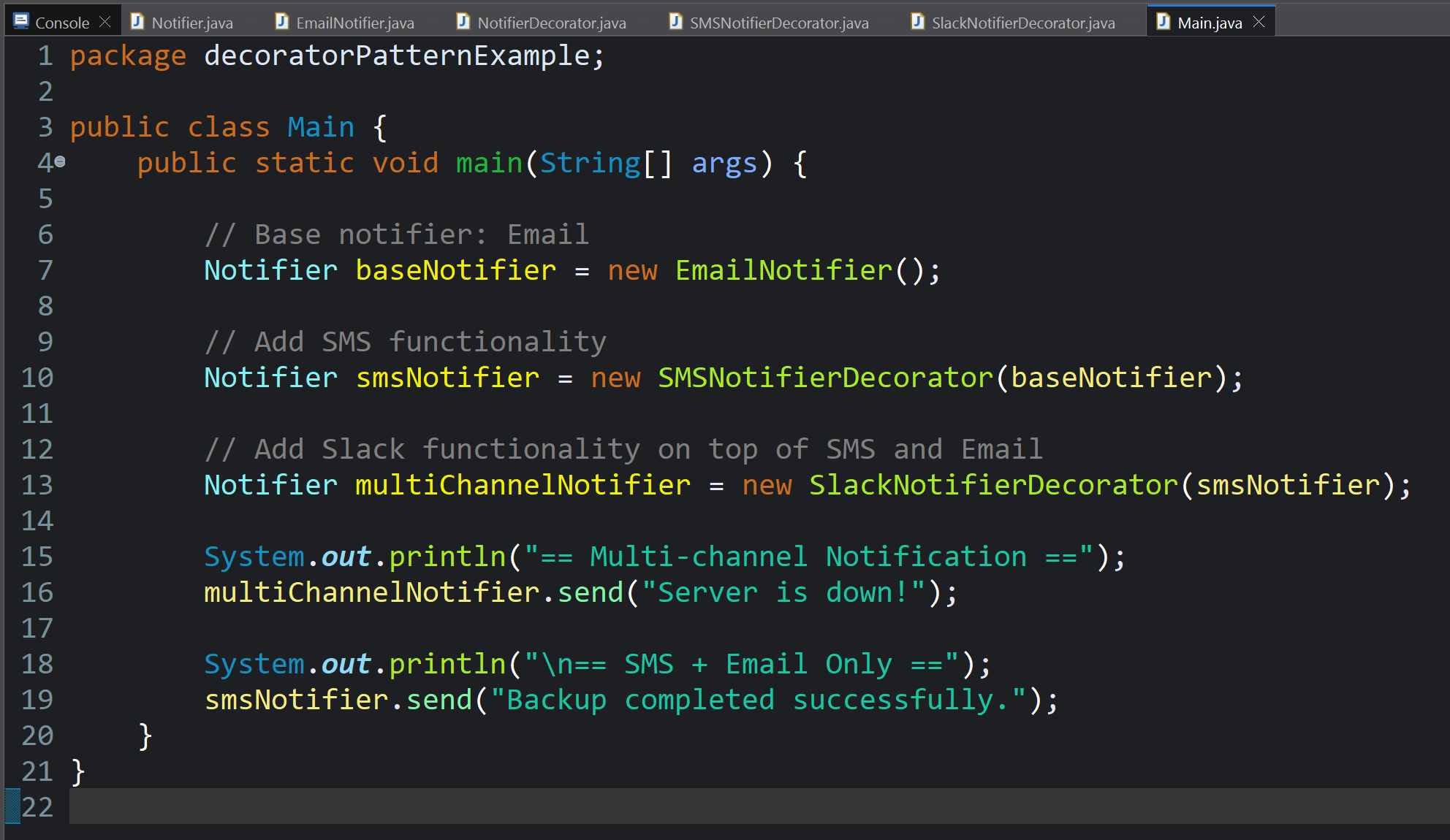












Output:

