Question 9 solid principles

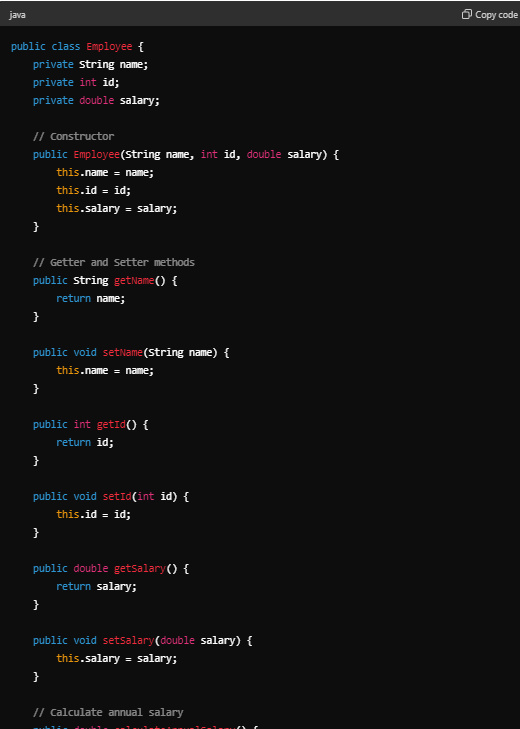
**Single Responsibility Principle (SRP)**

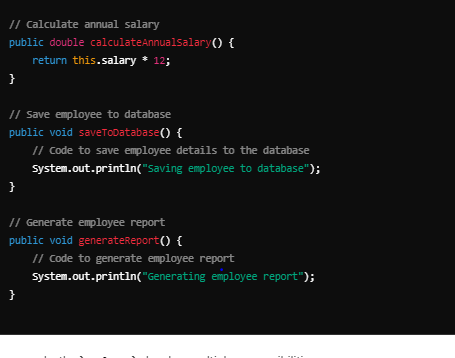
The **Single Responsibility Principle (SRP**) is one of the five SOLID principles of object-oriented design. It states that a class should have only one reason to change, meaning that it should have only one job or responsibility. When a class adheres to SRP, it is easier to understand, maintain, and modify.

**Example of SRP Violation in Java**

Violates

Here's an example of a class that violates the Single Responsibility Principle:





In this example, the Employee class has multiple responsibilities:

1. It handles employee data (fields and their getters/setters).
2. It calculates the annual salary.
3. It saves the employee to a database.
4. It generates a report for the employee.

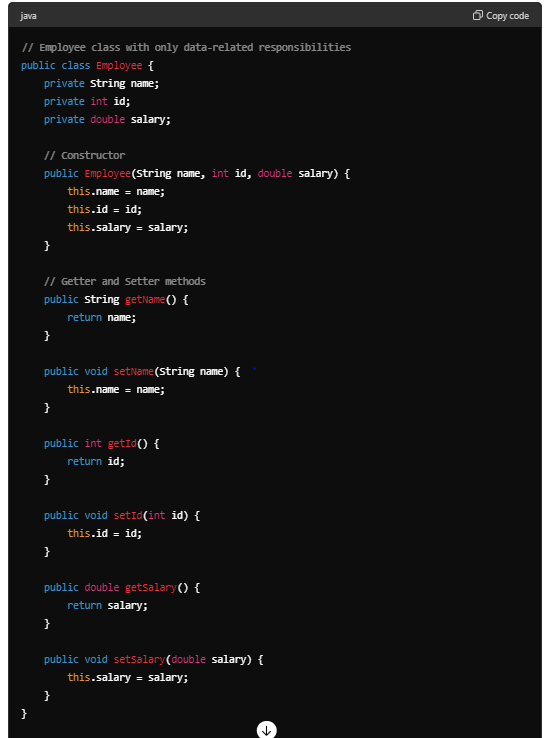
**Refactored Example to Adhere to SRP**

To adhere to the Single Responsibility Principle, we can refactor the code by creating separate classes for each responsibility:

1. A class for employee data.
2. A class for salary calculation.
3. A class for database operations.
4. A class for report generation.

Here is how we can do it:

Solution

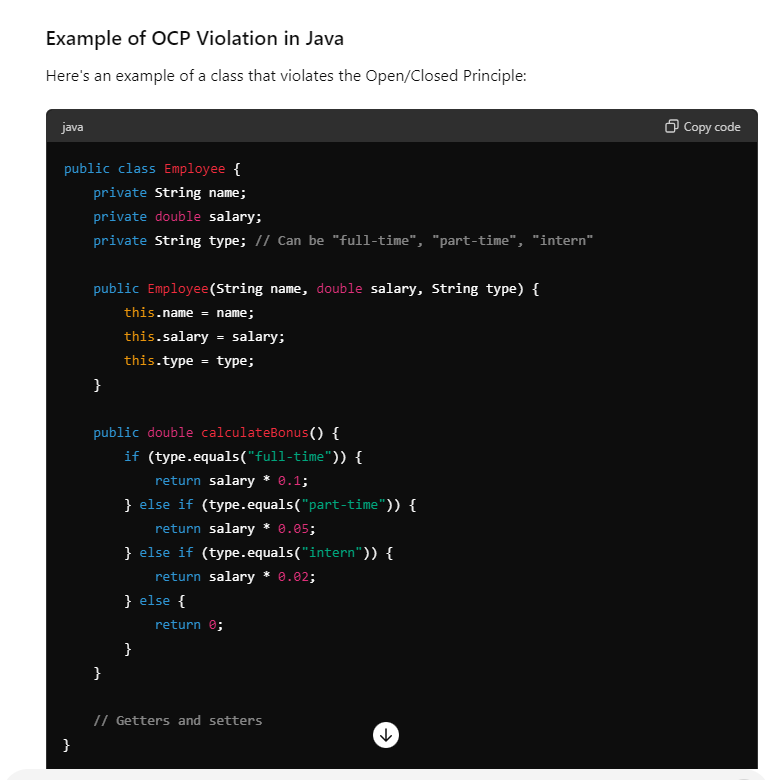




Open/Closed Principle (OCP)

The Open/Closed Principle (OCP) is another one of the SOLID principles of object-oriented design. It states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. This means you should be able to add new functionality to an entity without changing its existing code.

Violates

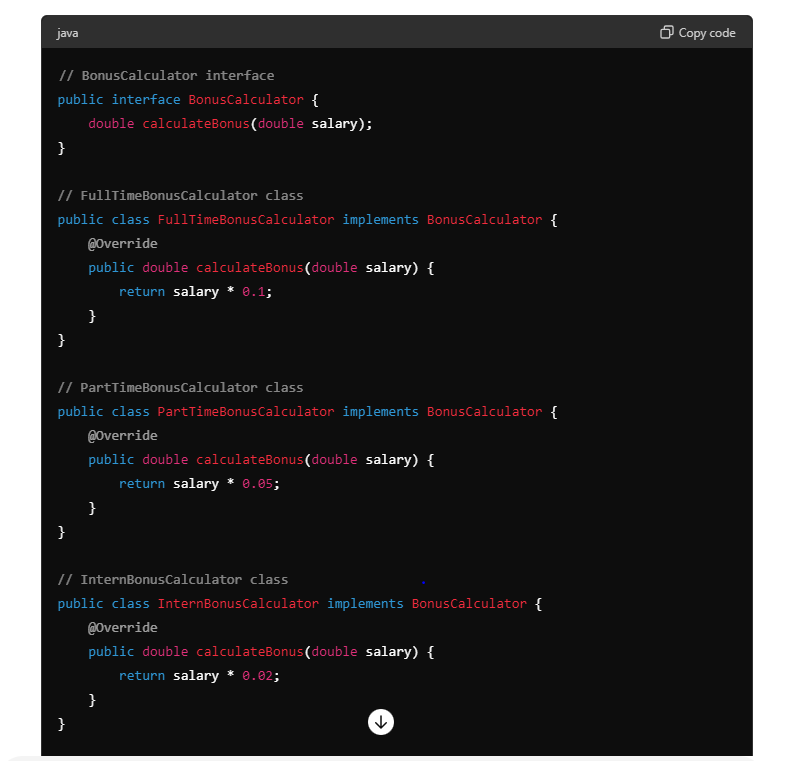


In this example, every time we need to add a new employee type, we have to modify the calculateBonus method, which violates the Open/Closed Principle.

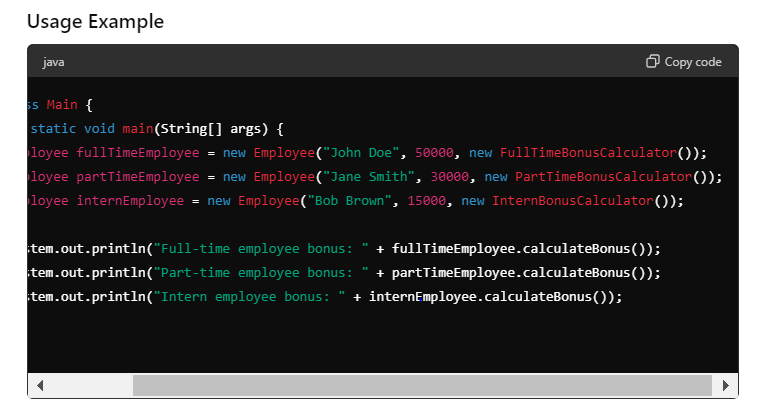
**solution**

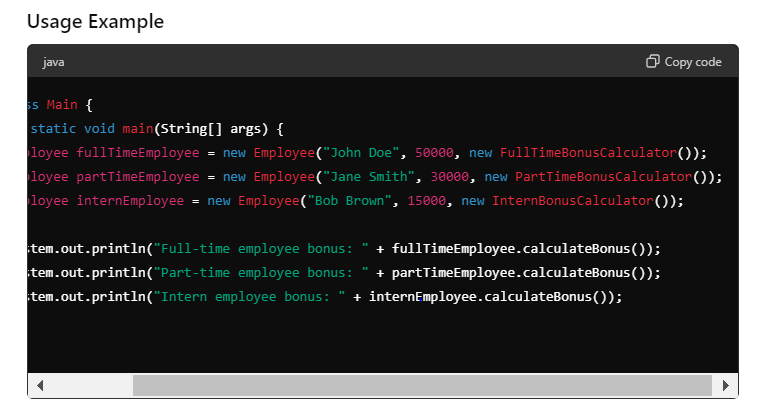
**Refactored Example to Adhere to OCP**

To adhere to the Open/Closed Principle, we can use polymorphism. We'll create an interface for calculating bonuses and implement this interface for each employee type.









Liskov Substitution Principle (LSP)

**subclass does not change the expected behavior of the base class**

The **Liskov Substitution Principle (LSP)** is another SOLID principle that states that objects of a superclass should be replaceable with objects of a subclass without affecting the functionality of the program. In other words, subclasses should be substitutable for their base classes.

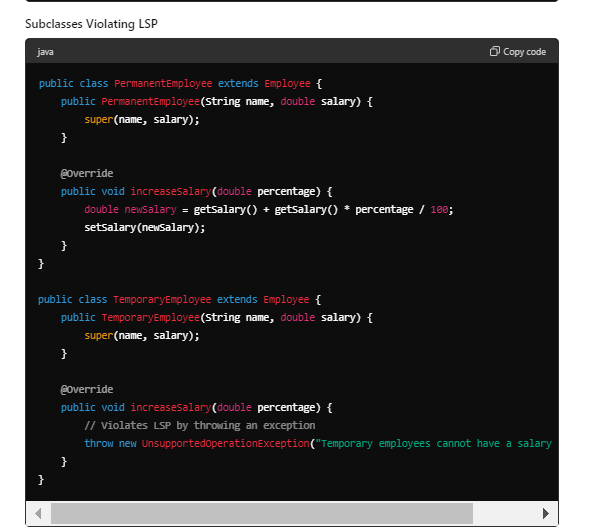
Violation

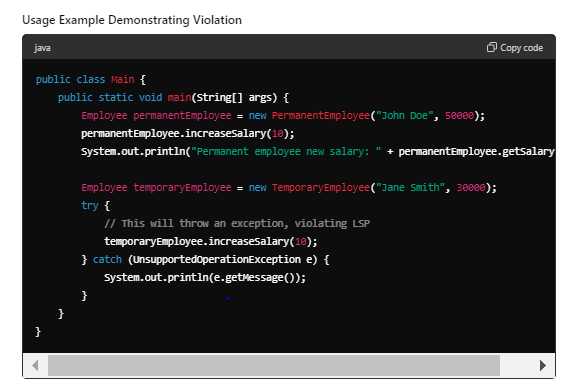
Certainly! Let's start by showing how a violation of the Liskov Substitution Principle (LSP) can occur even when using an abstract class, followed by a corrected version that adheres to LSP.

**Violation of LSP with Abstract Class**

In this example, the TemporaryEmployee class overrides the increaseSalary method in a way that breaks the contract expected by clients of the Employee class.







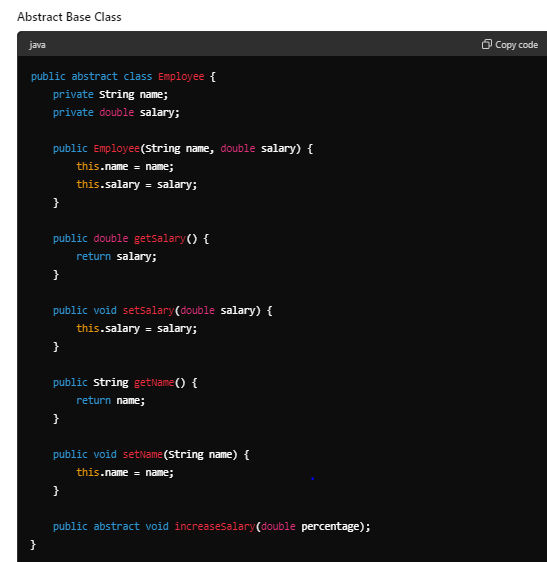
In this violation example, the TemporaryEmployee class throws an UnsupportedOperationException in the increaseSalary method, which breaks the LSP. Clients expecting to use an Employee object should not encounter such exceptions when calling increaseSalary.

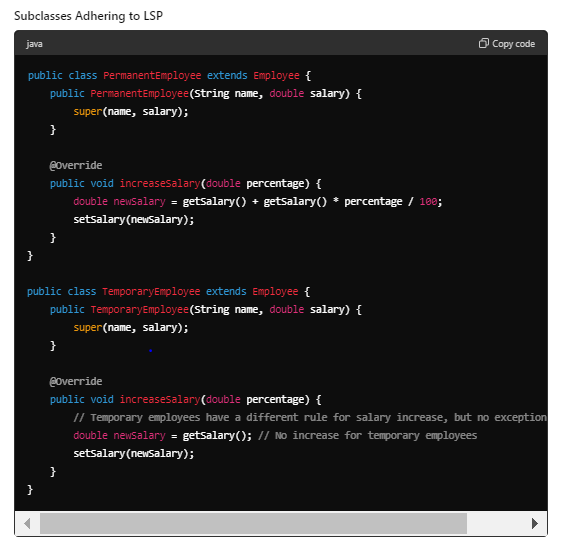
solution

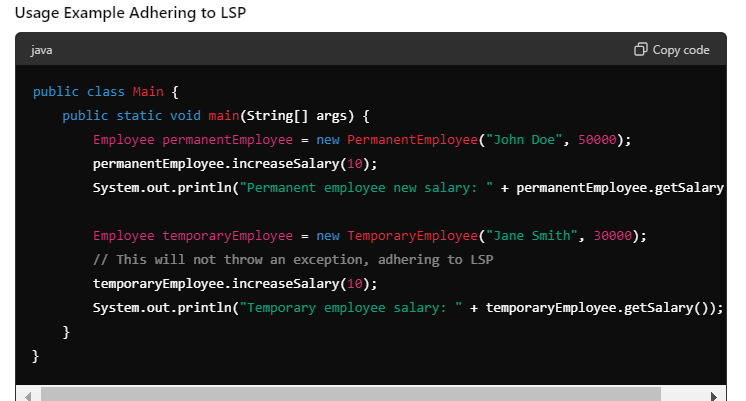
### Adhering to LSP -------

To adhere to LSP, we can redesign the classes to ensure that the behavior expected by the Employee class is preserved in its subclasses.

#### Abstract Base Class







In this corrected example, the TemporaryEmployee class overrides the increaseSalary method in a way that doesn't break the expected behavior of the Employee class. Instead of throwing an exception, it simply doesn't change the salary. This adheres to the Liskov Substitution Principle, ensuring that clients can rely on the consistent behavior of Employee objects regardless of their specific type.

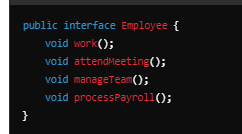
**Interface Segregation Principle (ISP)**

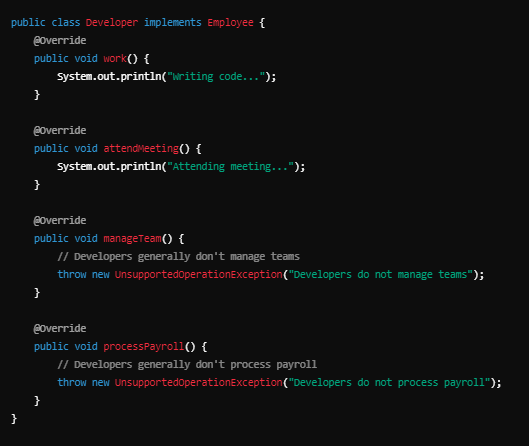
The **Interface Segregation Principle (ISP)** states that no client should be forced to depend on methods it does not use. Essentially, it's better to have multiple, smaller, and more specific interfaces than one large, general-purpose interface.

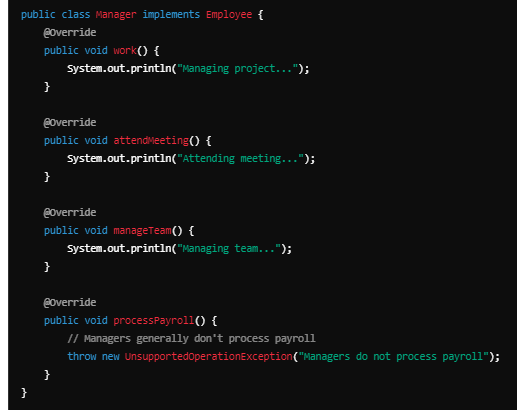
### Example of ISP Violation in Java

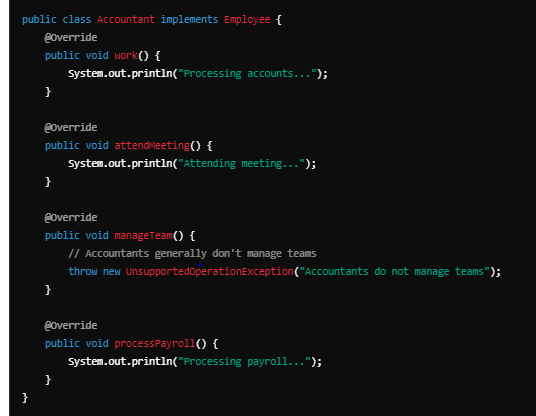
Let's consider an example where we have a general Employee interface that is too broad and forces all implementing classes to provide implementations for methods they don't need.

#### Interface that Violates ISP ----- violation







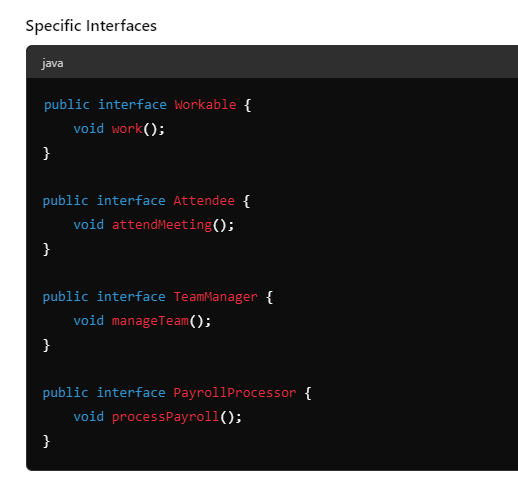


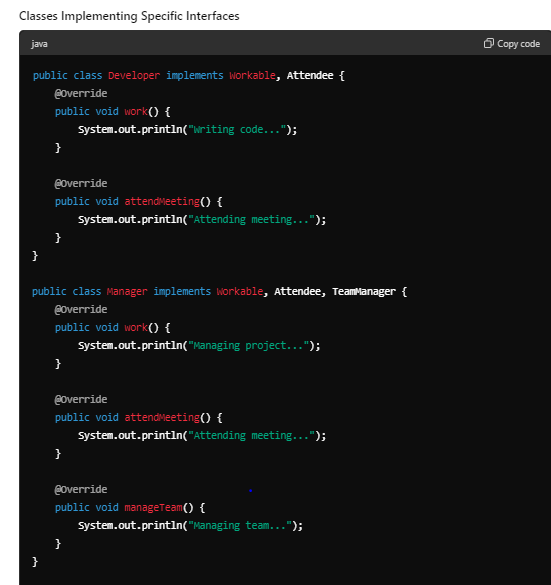
In this example, the Employee interface forces Developer, Manager, and Accountant classes to implement methods they do not need, leading to UnsupportedOperationException in many cases.

solution

**Adhering to ISP by Creating Specific Interfaces**

To adhere to the Interface Segregation Principle, we can create more specific interfaces that capture the responsibilities of different types of employees.







**Explanation**

* **Specific Interfaces**: Instead of one large Employee interface, we create smaller interfaces like Workable, Attendee, TeamManager, and PayrollProcessor to capture specific responsibilities.
* **Implementing Classes**: Each class implements only the interfaces relevant to its responsibilities. For example, Developer implements Workable and Attendee, while Manager implements Workable, Attendee, and TeamManager.
* **Usage**: The classes use only the methods they need without any unsupported operations, adhering to the Interface Segregation Principle.

By following ISP, we ensure that classes are not forced to implement methods they do not use, resulting in cleaner and more maintainable code.

**Dependency Inversion Principle (DIP)**

The **Dependency Inversion Principle (DIP)** states that:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g., interfaces).
2. Abstractions should not depend on details. Details should depend on abstractions.

Violation

### DIP Violation Example

Imagine a scenario where an OrderService depends directly on a concrete PayPalPaymentProcessor class. This setup violates DIP because the high-level OrderService is dependent on the low-level PayPalPaymentProcessor.

#### Concrete Classes and DIP Violation



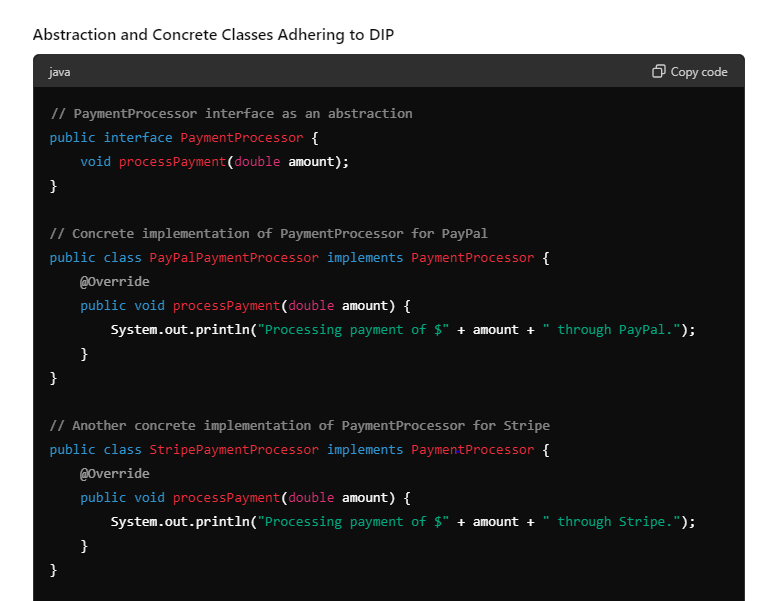
In this example, OrderService is tightly coupled to PayPalPaymentProcessor, making it difficult to switch to a different payment processor or to test the OrderService in isolation.

### Adhering to DIP Using Interfaces

To adhere to the Dependency Inversion Principle, we can introduce an interface PaymentProcessor that both the OrderService and PayPalPaymentProcessor depend on. This way, OrderService depends on an abstraction rather than a concrete implementation.

#### Abstraction and Concrete Classes Adhering to DIP

Solution





**Explanation**

* **Interface PaymentProcessor**: This interface provides the abstraction that both OrderService and payment processors (e.g., PayPalPaymentProcessor and StripePaymentProcessor) depend on.
* **Concrete Classes PayPalPaymentProcessor and StripePaymentProcessor**: Implement the PaymentProcessor interface.
* **High-level Class OrderService**: Depends on PaymentProcessor, making it independent of the specific payment processor implementations.

**Usage**

In the Main class, we can easily switch between different payment processor implementations (e.g., PayPalPaymentProcessor and StripePaymentProcessor) by injecting the appropriate implementation into OrderService. This makes the OrderService more flexible and easier to test.

By following DIP, the high-level OrderService does not depend on the low-level PayPalPaymentProcessor or StripePaymentProcessor directly. Instead, both the high-level and low-level modules depend on the PaymentProcessor abstraction. This results in a more decoupled, maintainable, and testable codebase.