**Exercise 4: Implementing the Adapter Pattern**

**PaymentProcessor.java**

package mypackage;

public interface PaymentProcessor {

void processPayment(double amount);

}

**PayPal.java**

package mypackage;

public class PayPal {

public void sendPayment(double amount) {

System.out.println("Processing PayPal payment of ₹" + amount);

}

}

**Stripe.java**

package mypackage;

public class Stripe {

public void makePayment(double amount) {

System.out.println("Processing Stripe payment of ₹" + amount);

}

}

**Razorpay.java**

package mypackage;

public class Razorpay {

public void executePayment(double amount) {

System.out.println("Processing Razorpay payment of ₹" + amount);

}

}

### Implement Adapter Classes

**PayPalAdapter.java**

package mypackage;

public class PayPalAdapter implements PaymentProcessor {

private PayPal paypal;

public PayPalAdapter(PayPal paypal) {

this.paypal = paypal;

}

public void processPayment(double amount) {

paypal.sendPayment(amount);

}

}

**StripeAdapter.java**

package mypackage;

public class StripeAdapter implements PaymentProcessor {

private Stripe stripe;

public StripeAdapter(Stripe stripe) {

this.stripe = stripe;

}

public void processPayment(double amount) {

stripe.makePayment(amount);

}

}

**RazorpayAdapter.java**

package mypackage;

public class RazorpayAdapter implements PaymentProcessor {

private Razorpay razorpay;

public RazorpayAdapter(Razorpay razorpay) {

this.razorpay = razorpay;

}

public void processPayment(double amount) {

razorpay.executePayment(amount);

}

}

### 5. Test the Adapter Implementation

**TestAdapterPattern.java**

package mypackage;

import java.util.Scanner;

public class TestAdapterPattern {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.println("Select Payment Gateway:");

System.out.println("1. PayPal");

System.out.println("2. Stripe");

System.out.println("3. Razorpay");

System.out.print("Enter your choice: ");

int choice = sc.nextInt();

System.out.print("Enter amount to pay (in ₹): ");

double amount = sc.nextDouble();

PaymentProcessor processor = null;

switch (choice) {

case 1:

processor = new PayPalAdapter(new PayPal());

break;

case 2:

processor = new StripeAdapter(new Stripe());

break;

case 3:

processor = new RazorpayAdapter(new Razorpay());

break;

default:

System.out.println("Invalid choice!");

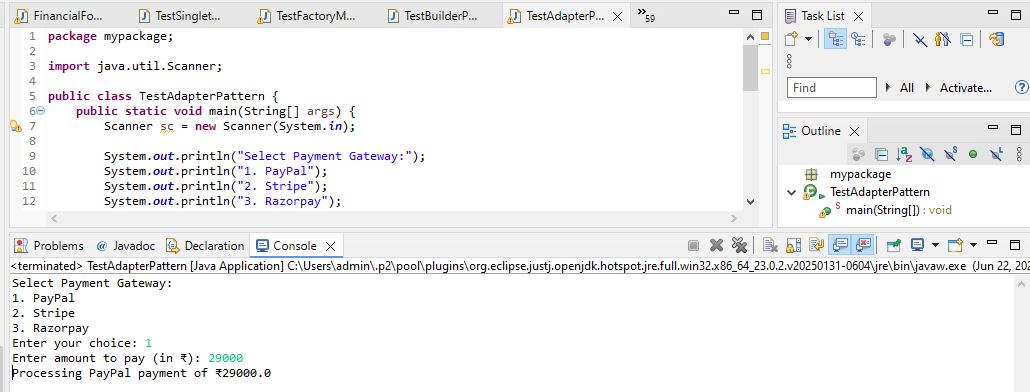
System.exit(0);

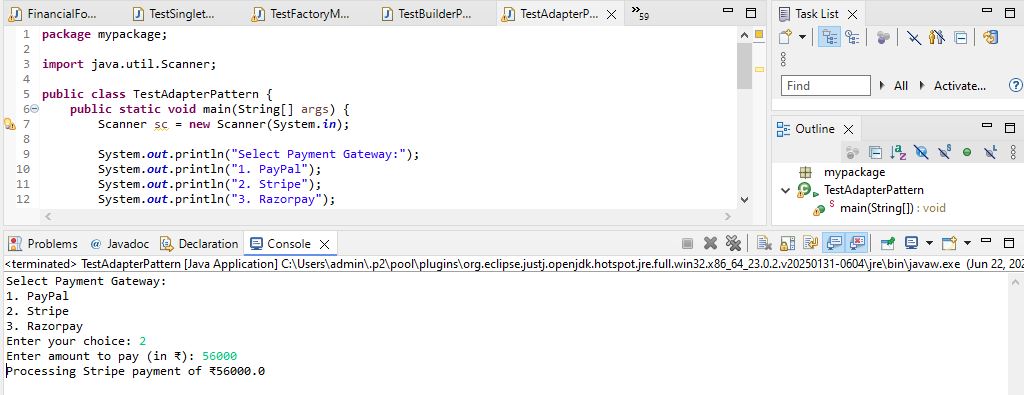
}

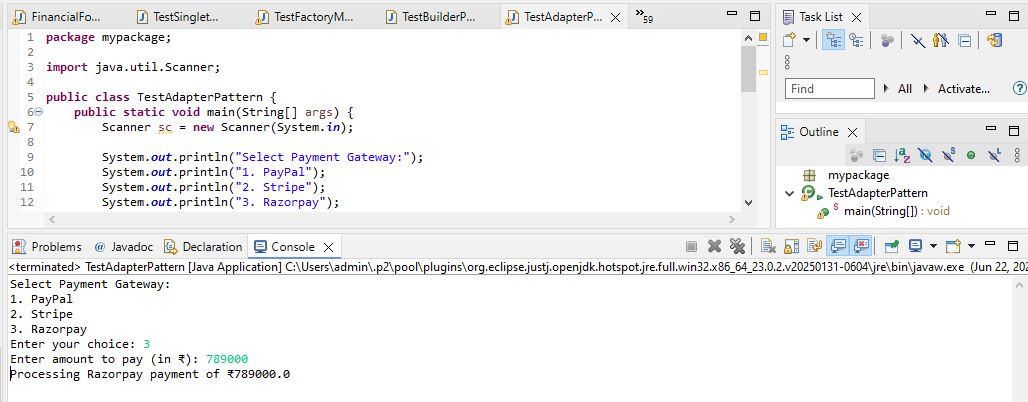
processor.processPayment(amount);

}

}







We are developing a payment processing system that needs to integrate with multiple third-party payment gateways (such as PayPal, Stripe, Razorpay), each having its own interface. To standardize interaction with these gateways, we apply the Adapter Pattern.

**1. Target Interface**

We define a common interface PaymentProcessor with the method processPayment(). This interface is used by the client code to process payments without knowing the internal implementation of each gateway.

**2. Adaptee Classes**

The existing payment gateways have their own interfaces and methods:

* PayPal class has the method sendPayment().
* Stripe class has the method makePayment().
* Razorpay class has the method executePayment().

These classes are called Adaptees because their interfaces do not match the standard PaymentProcessor interface.

**3. Adapter Classes**

For each Adaptee class, we create an Adapter class that implements the PaymentProcessor interface. These adapters internally translate the processPayment() call into the corresponding Adaptee method:

* PayPalAdapter calls sendPayment().
* StripeAdapter calls makePayment().
* RazorpayAdapter calls executePayment().

This allows the client code to use a unified interface while hiding the differences in the underlying payment gateway implementations.

**4. Advantages of Adapter Pattern**

* Allows integration of multiple incompatible systems through a common interface.
* Promotes loose coupling between the client and external libraries or APIs.
* Simplifies future extensions if new payment gateways need to be added.
* Ensures that client code remains consistent and does not require changes when integrating additional payment systems.

**5. Time Complexity**

The Adapter methods simply delegate the call to the Adaptee methods. Therefore, each call operates in constant time: O(1)

**6. Real-life Applications**

* Payment gateway integrations in e-commerce platforms.
* Legacy system integration where old and new interfaces must work together.
* Wrapping third-party libraries to conform to an internal standard interface.