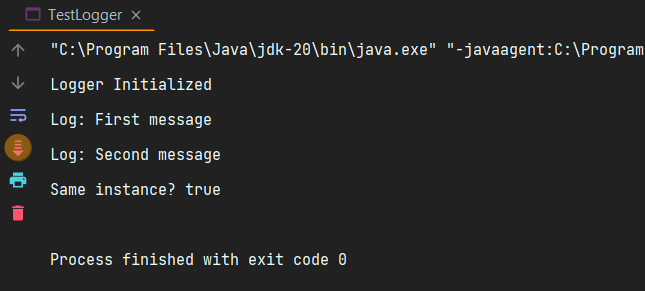
**Exercise 1: Implementing the Singleton Pattern**

**Java** **Code**:

*class* Logger {  
 *private static* Logger *instance*;  
 *private* Logger() {  
 System.***out***.println("Logger Initialized");  
 }  
 *public static* Logger getInstance() {  
 *if* (*instance* == *null*) {  
 *instance* = *new* Logger();  
 }  
 *return instance*;  
 }  
 *public void* log(String message) {  
 System.***out***.println("Log: " + message);  
 }  
}  
*public class TestLogger*{  
 *public static void* main(String[] args) {  
 Logger logger1 = Logger.*getInstance*();  
 Logger logger2 = Logger.*getInstance*();  
 logger1.log("First message");  
 logger2.log("Second message");  
 System.***out***.println("Same instance? " + (logger1 == logger2));  
 }  
}

**Output**:

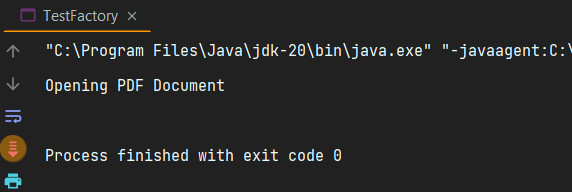


**Exercise 2: Implementing the Factory Method Pattern**

**Java Code:**

*interface Document* {  
 *void* open();  
}  
*class* WordDocument *implements Document* {  
 *public void* open() { System.***out***.println("Opening Word Document"); }  
}  
*class* PdfDocument *implements Document* {  
 *public void* open() { System.***out***.println("Opening PDF Document"); }  
}  
*class* ExcelDocument *implements Document* {  
 *public void* open() { System.***out***.println("Opening Excel Document"); }  
}  
*abstract class* DocumentFactory {  
 *abstract Document* createDocument();  
}  
*class* WordFactory *extends* DocumentFactory {  
 *Document* createDocument() { *return new* WordDocument(); }  
}  
*class* PdfFactory *extends* DocumentFactory {  
 *Document* createDocument() { *return new* PdfDocument(); }  
}  
*class* ExcelFactory *extends* DocumentFactory {  
 *Document* createDocument() { *return new* ExcelDocument(); }  
}  
*public class* TestFactory {  
 *public static void* main(String[] args) {  
 DocumentFactory factory = *new* PdfFactory();  
 *Document* doc = factory.createDocument();  
 doc.open();  
 }  
}

**Output:**

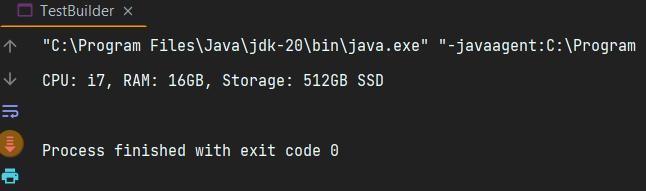
****

**Exercise 3: Implementing the Builder Pattern**

**Java Code:**

*class* Computer {  
 *private* String CPU, RAM, Storage;  
 *private* Computer(Builder builder) {  
 *this*.CPU = builder.CPU;  
 *this*.RAM = builder.RAM;  
 *this*.Storage = builder.Storage;  
 }  
 *public static class* Builder {  
 *private* String CPU, RAM, Storage;  
 *public* Builder setCPU(String cpu) { *this*.CPU = cpu; *return this*; }  
 *public* Builder setRAM(String ram) { *this*.RAM = ram; *return this*; }  
 *public* Builder setStorage(String storage) { *this*.Storage = storage; *return this*; }  
 *public* Computer build() {  
 *return new* Computer(*this*);  
 }  
 }  
 *public* String toString() {  
 *return* "CPU: " + CPU + ", RAM: " + RAM + ", Storage: " + Storage;  
 }  
}  
*public class* TestBuilder {  
 *public static void* main(String[] args) {  
 Computer c1 = *new* Computer.Builder().setCPU("i7").setRAM("16GB").setStorage("512GB SSD").build();  
 System.***out***.println(c1);  
 }  
}

**Output:**

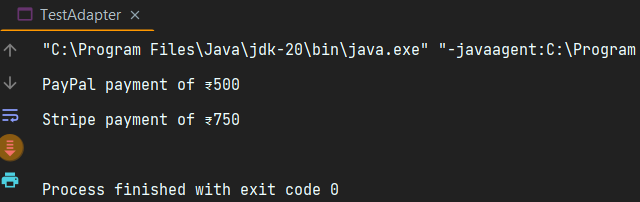
****

**Exercise 4: Implementing the Adapter Pattern**

**Java Code:**

*interface PaymentProcessor* {  
 *void* processPayment(String amount);  
}  
*class* PayPal {  
 *void* sendPayment(String amt) {  
 System.***out***.println("PayPal payment of " + amt);  
 }  
}  
*class* Stripe {  
 *void* makePayment(String amt) {  
 System.***out***.println("Stripe payment of " + amt);  
 }  
}  
*class* PayPalAdapter *implements PaymentProcessor* {  
 *private* PayPal paypal = *new* PayPal();  
 *public void* processPayment(String amount) {  
 paypal.sendPayment(amount);  
 }  
}  
*class* StripeAdapter *implements PaymentProcessor* {  
 *private* Stripe stripe = *new* Stripe();  
 *public void* processPayment(String amount) {  
 stripe.makePayment(amount);  
 }  
}  
*public class* TestAdapter {  
 *public static void* main(String[] args) {  
 *PaymentProcessor* pp = *new* PayPalAdapter();  
 pp.processPayment("₹500");  
 pp = *new* StripeAdapter();  
 pp.processPayment("₹750");  
 }  
}

**Output:**

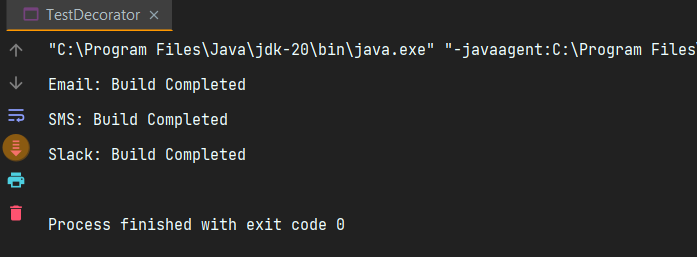
****

**Exercise 5: Implementing the Decorator Pattern**

**Java Code:**

*interface Notifier* {  
 *void* send(String message);  
}  
*class* EmailNotifier *implements Notifier* {  
 *public void* send(String message) {  
 System.***out***.println("Email: " + message);  
 }  
}  
*abstract class* NotifierDecorator *implements Notifier* {  
 *protected Notifier* notifier;  
 *public* NotifierDecorator(*Notifier* notifier) {  
 *this*.notifier = notifier;  
 }  
}  
*class* SMSNotifier *extends* NotifierDecorator {  
 *public* SMSNotifier(*Notifier* notifier) {  
 *super*(notifier);  
 }  
 *public void* send(String message) {  
 notifier.send(message);  
 System.***out***.println("SMS: " + message);  
 }  
}  
*class* SlackNotifier *extends* NotifierDecorator {  
 *public* SlackNotifier(*Notifier* notifier) {  
 *super*(notifier);  
 }  
 *public void* send(String message) {  
 notifier.send(message);  
 System.***out***.println("Slack: " + message);  
 }  
}  
*public class* TestDecorator {  
 *public static void* main(String[] args) {  
 *Notifier* notifier = *new* SlackNotifier(*new* SMSNotifier(*new* EmailNotifier()));  
 notifier.send("Build Completed");  
 }  
}

**Output:**

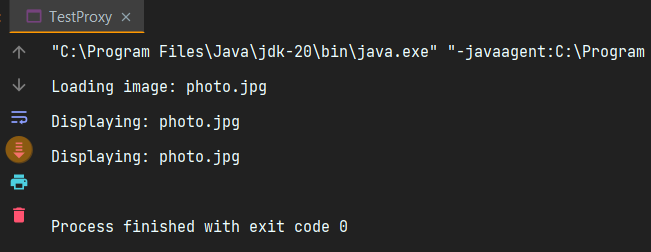
****

**Exercise 6: Implementing the Proxy Pattern**

**Java Code:**

*interface Image* {  
 *void* display();  
}  
*class* RealImage *implements Image* {  
 *private* String filename;  
 *public* RealImage(String filename) {  
 *this*.filename = filename;  
 loadImage();  
 }  
 *private void* loadImage() {  
 System.***out***.println("Loading image: " + filename);  
 }  
 *public void* display() {  
 System.***out***.println("Displaying: " + filename);  
 }  
}  
*class* ProxyImage *implements Image* {  
 *private* RealImage realImage;  
 *private* String filename;  
 *public* ProxyImage(String filename) {  
 *this*.filename = filename;  
 }  
 *public void* display() {  
 *if* (realImage == *null*) {  
 realImage = *new* RealImage(filename);  
 }  
 realImage.display();  
 }  
}  
*public class* TestProxy {  
 *public static void* main(String[] args) {  
 *Image* img = *new* ProxyImage("photo.jpg");  
 img.display();   
 img.display();   
 }  
}

**Output:**

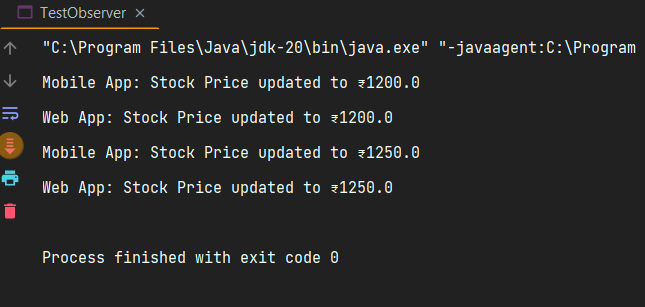
****

**Exercise 7: Implementing the Observer Pattern**

**Java Code:**

*import* java.util.\*;  
*interface Observer* {  
 *void* update(*float* price);  
}  
*interface Stock* {  
 *void* register(*Observer* o);  
 *void* deregister(*Observer* o);  
 *void* notifyObservers();  
}  
*class* StockMarket *implements Stock* {  
 *private List*<*Observer*> observers = *new* ArrayList<>();  
 *private float* price;  
 *public void* setPrice(*float* price) {  
 *this*.price = price;  
 notifyObservers();  
 }  
 *public void* register(*Observer* o) { observers.add(o); }  
 *public void* deregister(*Observer* o) { observers.remove(o); }  
 *public void* notifyObservers() {  
 *for* (*Observer* o : observers) o.update(price);  
 }  
}  
*class* MobileApp *implements Observer* {  
 *public void* update(*float* price) {  
 System.***out***.println("Mobile App: Stock Price updated to ₹" + price);  
 }  
}  
*class* WebApp *implements Observer* {  
 *public void* update(*float* price) {  
 System.***out***.println("Web App: Stock Price updated to ₹" + price);  
 }  
}  
*public class* TestObserver {  
 *public static void* main(String[] args) {  
 StockMarket market = *new* StockMarket();  
 *Observer* mob = *new* MobileApp();  
 *Observer* web = *new* WebApp();  
 market.register(mob);  
 market.register(web);  
 market.setPrice(1200);  
 market.setPrice(1250);  
 }  
}

**Output:**

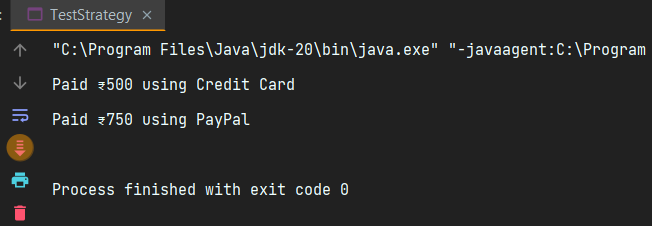
****

**Exercise 8: Implementing the Strategy Pattern**

**Java Code:**

*interface PaymentStrategy* {  
 *void* pay(*int* amount);  
}  
*class* CreditCardPayment *implements PaymentStrategy* {  
 *public void* pay(*int* amount) {  
 System.***out***.println("Paid ₹" + amount + " using Credit Card");  
 }  
}  
*class* PayPalPayment *implements PaymentStrategy* {  
 *public void* pay(*int* amount) {  
 System.***out***.println("Paid ₹" + amount + " using PayPal");  
 }  
}  
*class* PaymentContext {  
 *private PaymentStrategy* strategy;  
 *public void* setPaymentStrategy(*PaymentStrategy* strategy) {  
 *this*.strategy = strategy;  
 }  
 *public void* pay(*int* amount) {  
 strategy.pay(amount);  
 }  
}  
*public class* TestStrategy {  
 *public static void* main(String[] args) {  
 PaymentContext context = *new* PaymentContext();  
 context.setPaymentStrategy(*new* CreditCardPayment());  
 context.pay(500);  
 context.setPaymentStrategy(*new* PayPalPayment());  
 context.pay(750);  
 }  
}

**Output:**

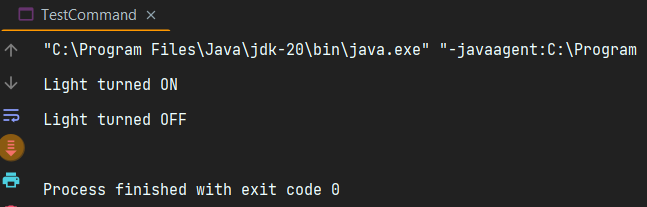
****

**Exercise 9: Implementing the Command Pattern**

**Java Code:**

*interface Command* {  
 *void* execute();  
}  
*class* Light {  
 *public void* on() { System.***out***.println("Light turned ON"); }  
 *public void* off() { System.***out***.println("Light turned OFF"); }  
}  
*class* LightOnCommand *implements Command* {  
 Light light;  
 *public* LightOnCommand(Light light) { *this*.light = light; }  
 *public void* execute() { light.on(); }  
}  
*class* LightOffCommand *implements Command* {  
 Light light;  
 *public* LightOffCommand(Light light) { *this*.light = light; }  
 *public void* execute() { light.off(); }  
}  
*class* RemoteControl {  
 *Command* command;  
 *public void* setCommand(*Command* command) {  
 *this*.command = command;  
 }  
 *public void* pressButton() {  
 command.execute();  
 }  
}  
*public class* TestCommand {  
 *public static void* main(String[] args) {  
 Light light = *new* Light();  
 *Command* on = *new* LightOnCommand(light);  
 *Command* off = *new* LightOffCommand(light);  
 RemoteControl remote = *new* RemoteControl();  
 remote.setCommand(on);  
 remote.pressButton();  
 remote.setCommand(off);  
 remote.pressButton();  
 }  
}

**Output:**

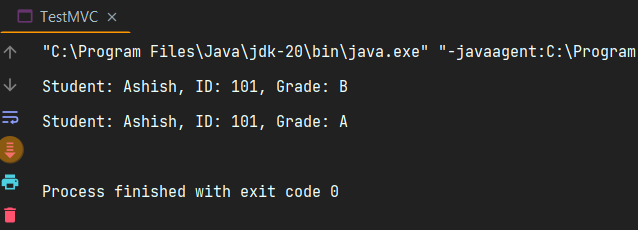
****

**Exercise 10: Implementing the MVC Pattern**

**Java Code:**

*class* Student {  
 String name;  
 String id;  
 String grade;  
 *public* Student(String name, String id, String grade) {  
 *this*.name = name;  
 *this*.id = id;  
 *this*.grade = grade;  
 }  
}  
*class* StudentView {  
 *public void* displayStudentDetails(Student s) {  
 System.***out***.println("Student: " + s.name + ", ID: " + s.id + ", Grade: " + s.grade);  
 }  
}  
*class* StudentController {  
 *private* Student student;  
 *private* StudentView view;  
 *public* StudentController(Student student, StudentView view) {  
 *this*.student = student;  
 *this*.view = view;  
 }  
 *public void* setGrade(String grade) {  
 student.grade = grade;  
 }  
 *public void* updateView() {  
 view.displayStudentDetails(student);  
 }  
}  
*public class* TestMVC {  
 *public static void* main(String[] args) {  
 Student s = *new* Student("Ashish", "101", "B");  
 StudentView view = *new* StudentView();  
 StudentController controller = *new* StudentController(s, view);  
 controller.updateView();  
 controller.setGrade("A");  
 controller.updateView();  
 }  
}

**Output:**

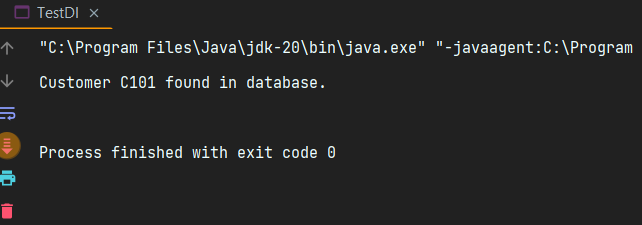
****

**Exercise 11: Implementing Dependency Injection**

**Java Code:**

*interface CustomerRepository* {  
 String findCustomerById(String id);  
}  
*class* CustomerRepositoryImpl *implements CustomerRepository* {  
 *public* String findCustomerById(String id) {  
 *return* "Customer " + id + " found in database.";  
 }  
}  
*class* CustomerService {  
 *private CustomerRepository* repo;  
 *public* CustomerService(*CustomerRepository* repo) {  
 *this*.repo = repo;  
 }  
 *public void* findCustomer(String id) {  
 System.***out***.println(repo.findCustomerById(id));  
 }  
}  
*public class* TestDI {  
 *public static void* main(String[] args) {  
 *CustomerRepository* repo = *new* CustomerRepositoryImpl();  
 CustomerService service = *new* CustomerService(repo);  
 service.findCustomer("C101");  
 }  
}

**Output:**

****