**WEEK-1**

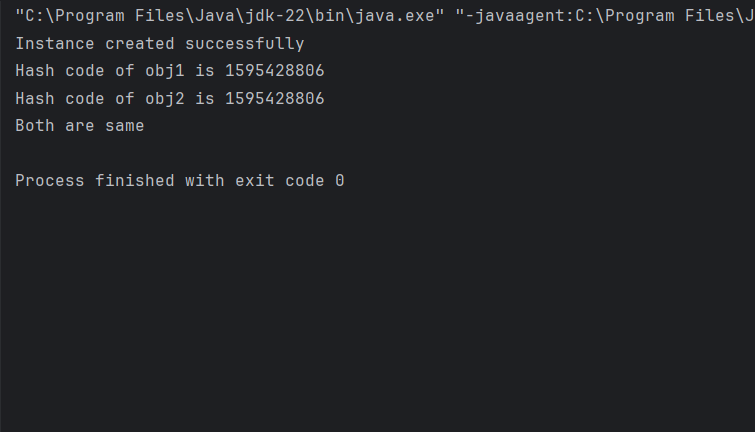
**Design Patterns and Principles HandsOn**

**Exercise 1: Implementing the Singleton Pattern**

**Code:**

class Logger{  
 private static Logger *object*;  
 private Logger(){  
 System.*out*.println("Instance created successfully");  
 }  
 public static Logger getInstance(){  
 if(*object*==null){  
 *object*=new Logger();  
 }  
 return *object*;  
 }  
}  
class Test{  
 public void testSingleton(){  
 Logger obj1=Logger.*getInstance*();  
 Logger obj2=Logger.*getInstance*();  
 if(obj2==obj1){  
 System.*out*.println("Hash code of obj1 is "+obj1.hashCode());  
 System.*out*.println("Hash code of obj2 is "+obj2.hashCode());  
 System.*out*.println("Both are same");  
 }  
 }  
}  
public class Main {  
 public static void main(String[] args) {  
 Test tester=new Test();  
 tester.testSingleton();  
 }  
}

**Output:**

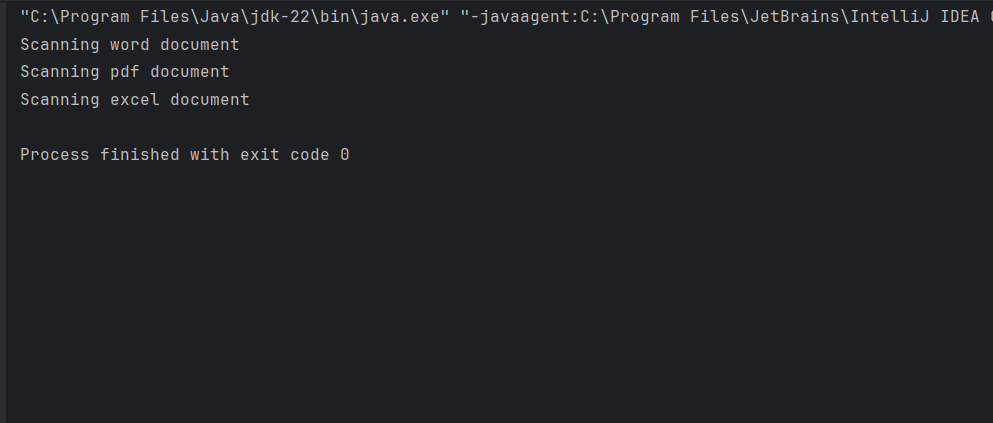
****

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

**Code:**

interface Feature{  
 void scan();  
}  
class WordDocument implements Feature{  
 public void scan(){  
 System.*out*.println("Scanning word document");  
 }  
}  
class PdfDocument implements Feature{  
 public void scan(){  
 System.*out*.println("Scanning pdf document");  
 }  
}  
class ExcelDocument implements Feature{  
 public void scan(){  
 System.*out*.println("Scanning excel document");  
 }  
}  
  
abstract class DocumentFactory{  
 public abstract Feature createDocument();  
}  
class WordDocumentry extends DocumentFactory{  
 public Feature createDocument(){  
 return new WordDocument();  
 }  
}  
class PdfDocumentry extends DocumentFactory{  
 public Feature createDocument(){  
 return new PdfDocument();  
 }  
}  
class ExcelDocumentry extends DocumentFactory{  
 public Feature createDocument(){  
 return new ExcelDocument();  
 }  
}  
  
class Factory {  
 public static void main(String[] args) {  
 DocumentFactory word= new WordDocumentry();  
 Feature wordd= word.createDocument();  
 wordd.scan();  
  
 DocumentFactory pdf= new PdfDocumentry();  
 Feature pdfd= pdf.createDocument();  
 pdfd.scan();  
  
 DocumentFactory excel= new ExcelDocumentry();  
 Feature exceld= excel.createDocument();  
 exceld.scan();  
 }  
}

**Ouput:**

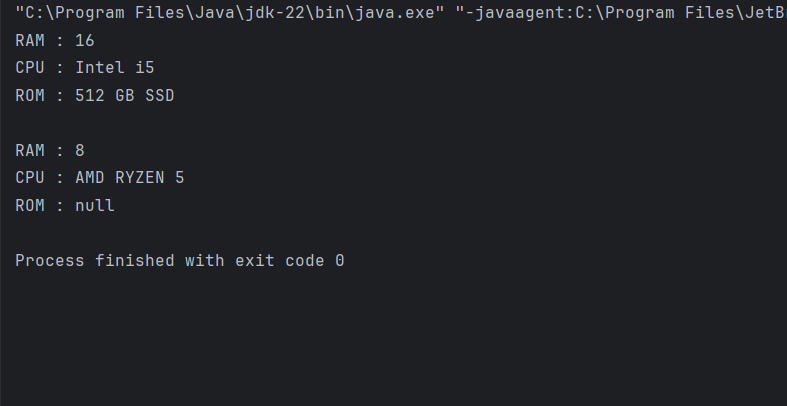
****

**Exercise 3: Implementing the Builder Pattern**

**Code:**

class specs{  
 private int RAM;  
 private String CPU;  
 private String ROM;  
 private specs(Builder object){  
 this.RAM=object.RAM;  
 this.CPU=object.CPU;  
 this.ROM=object.ROM;  
 }  
 public void details(){  
 System.*out*.println("RAM : "+RAM);  
 System.*out*.println("CPU : "+CPU);  
 System.*out*.println("ROM : "+ROM);  
 }  
 public static class Builder{  
 int RAM;  
 String CPU;  
 String ROM;  
 public Builder ram(int ra){  
 this.RAM=ra;  
 return this;  
 }  
 public Builder cpu(String c){  
 this.CPU=c;  
 return this;  
 }  
 public Builder rom(String rO){  
 this.ROM=rO;  
 return this;  
 }  
 public specs build(){  
 return new specs(this);  
 }  
  
 }  
}  
class builderPattern {  
 public static void main(String[] args) {  
 specs pc1=new specs.Builder()  
 .ram(16)  
 .cpu("Intel i5")  
 .rom("512 GB SSD")  
 .build();  
 pc1.details();  
 System.*out*.println();  
 specs pc2=new specs.Builder()  
 .ram(8)  
 .cpu("AMD RYZEN 5")  
 .build();  
 pc2.details();  
 }  
}

**Output:**

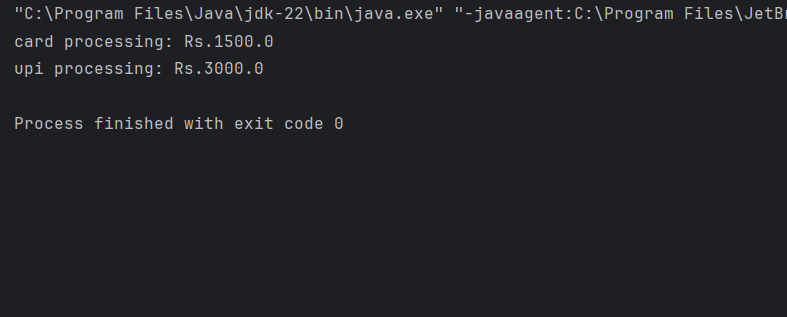
****

**Exercise 4: Implementing the Adapter Pattern**

**Code:**

interface PaymentProcessor {  
 void processPayment(double amount);  
}  
class card {  
 public void send(double amount){  
 System.*out*.println("card processing: Rs." + amount);  
 }  
}  
class upi {  
 public void send(double amount) {  
 System.*out*.println("upi processing: Rs." + amount);  
 }  
}  
class cardAdapter implements PaymentProcessor {  
 private card credit;  
  
 public cardAdapter(card credit) {  
 this.credit = credit;  
 }  
  
 @Override  
 public void processPayment(double amount) {  
 credit.send(amount);  
 }  
}  
class upiAdapter implements PaymentProcessor {  
 private upi phonepe;  
  
 public upiAdapter(upi phonepe) {  
 this.phonepe = phonepe;  
 }  
  
 @Override  
 public void processPayment(double amount) {  
 phonepe.send(amount);  
 }  
}  
  
class AdapterPattern {  
 public static void main(String[] args) {  
 PaymentProcessor card = new cardAdapter(new card());  
 card.processPayment(1500);  
  
 PaymentProcessor upi = new upiAdapter(new upi());  
 upi.processPayment(3000);  
 }  
}

**Output:**

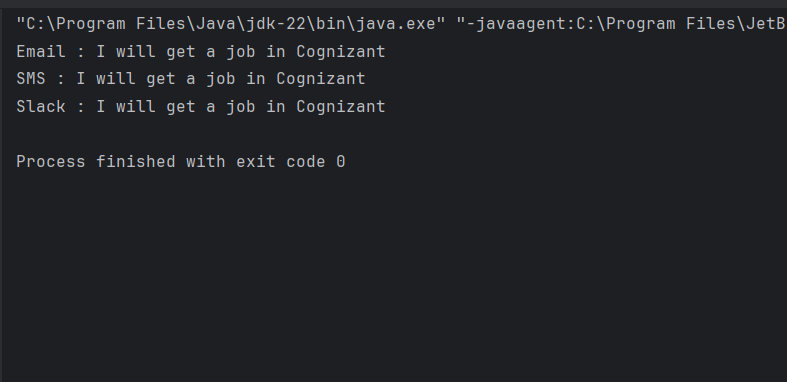
****

**Exercise 5: Implementing the Decorator Pattern**

**Code:**

interface Notifier {  
 void send(String text);  
}  
class EmailNotifier implements Notifier {  
 @Override  
 public void send(String text) {  
 System.*out*.println("Email : " + text);  
 }  
}  
abstract class NotifierDecorator implements Notifier {  
 protected Notifier obj;  
  
 public NotifierDecorator(Notifier notifier) {  
 this.obj = notifier;  
 }  
 @Override  
 public void send(String text) {  
 obj.send(text);  
 }  
}  
class SMSNotifierDecorator extends NotifierDecorator {  
 public SMSNotifierDecorator(Notifier notification) {  
 super(notification);  
 }  
  
 @Override  
 public void send(String text) {  
 super.send(text);  
 sms(text);  
 }  
  
 private void sms(String text) {  
 System.*out*.println("SMS : " + text);  
 }  
}  
class SlackNotifierDecorator extends NotifierDecorator {  
 public SlackNotifierDecorator(Notifier notification) {  
 super(notification);  
 }  
  
 @Override  
 public void send(String text) {  
 super.send(text);  
 slack(text);  
 }  
  
 private void slack(String text) {  
 System.*out*.println("Slack : " + text);  
 }  
}  
class DecoratorPattern {  
 public static void main(String[] args) {  
 Notifier notification = new EmailNotifier();  
  
 Notifier sms= new SMSNotifierDecorator(notification);  
 Notifier multi= new SlackNotifierDecorator(sms);  
  
 multi.send("I will get a job in Cognizant");  
 }  
}

**Output:**

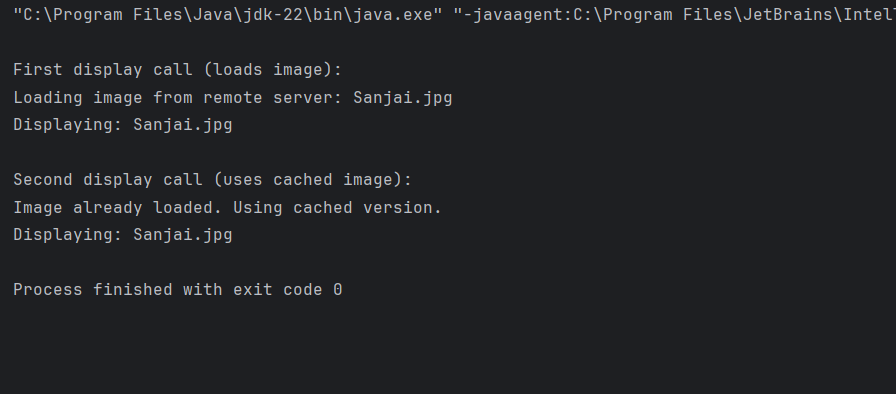
****

**Exercise 6: Implementing the Proxy Pattern**

**Code:**

interface Image {  
 void display();  
}  
class RealImage implements Image {  
 private String name;  
  
 public RealImage(String name) {  
 this.name = name;  
 loadFromRemoteServer();  
 }  
  
 private void loadFromRemoteServer() {  
 System.*out*.println("Loading image from remote server: " + name);  
 try {  
 Thread.*sleep*(2000); // simulate time delay for loading  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
  
 @Override  
 public void display() {  
 System.*out*.println("Displaying: " + name);  
 }  
}  
class ProxyImage implements Image {  
 private String name;  
 private RealImage realImage;  
  
 public ProxyImage(String name) {  
 this.name = name;  
 }  
  
 @Override  
 public void display() {  
 if (realImage == null) {  
 realImage = new RealImage(name);  
 } else {  
 System.*out*.println("Image already loaded. Using cached version.");  
 }  
 realImage.display();  
 }  
}  
  
class ProxyPattern {  
 public static void main(String[] args) {  
 Image img1 = new ProxyImage("Sanjai.jpg");  
  
 System.*out*.println("\nFirst display call (loads image):");  
 img1.display();  
  
 System.*out*.println("\nSecond display call (uses cached image):");  
 img1.display();  
  
 }  
}

**Output:**

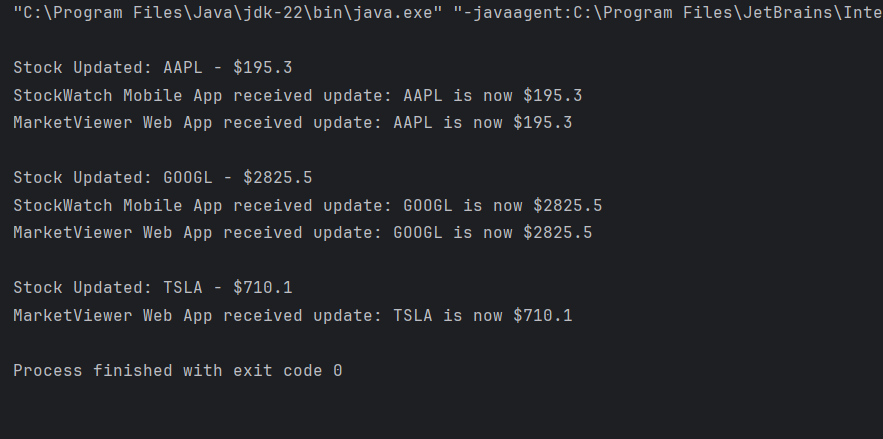
****

**Exercise 7: Implementing the Observer Pattern**

**Code:**

import java.util.\*;  
interface Stock {  
 void register(Observer obs);  
 void deregister(Observer obs);  
 void notifyer();  
}  
interface Observer {  
 void update(String symbol, double price);  
}  
  
  
class StockMarket implements Stock {  
 private List<Observer> observer = new ArrayList<>();  
 private String symbol;  
 private double price;  
  
 @Override  
 public void register(Observer obs) {  
 observer.add(obs);  
 }  
  
 @Override  
 public void deregister(Observer obs) {  
 observer.remove(obs);  
 }  
  
 @Override  
 public void notifyer() {  
 for (Observer o : observer) {  
 o.update(symbol, price);  
 }  
 }  
  
 public void setPrice(String symbol, double price) {  
 this.symbol = symbol;  
 this.price = price;  
 System.*out*.println("\nStock Updated: " + symbol + " - $" + price);  
 notifyer();  
 }  
}  
class MobileApp implements Observer {  
 private String appName;  
  
 public MobileApp(String name) {  
 this.appName = name;  
 }  
  
 @Override  
 public void update(String symbol, double price) {  
 System.*out*.println(appName + " Mobile App received update: " + symbol + " is now $" + price);  
 }  
}  
class WebApp implements Observer {  
 private String appName;  
  
 public WebApp(String name) {  
 this.appName = name;  
 }  
  
 @Override  
 public void update(String symbol, double price) {  
 System.*out*.println(appName + " Web App received update: " + symbol + " is now $" + price);  
 }  
}  
  
class ObserverPattern {  
 public static void main(String[] args) {  
 StockMarket stock = new StockMarket();  
  
 Observer mobile = new MobileApp("StockWatch");  
 Observer web = new WebApp("MarketViewer");  
  
 stock.register(mobile);  
 stock.register(web);  
  
 stock.setPrice("AAPL", 195.30);  
 stock.setPrice("GOOGL", 2825.50);  
  
 stock.deregister(mobile);  
 stock.setPrice("TSLA", 710.10);  
 }  
}

**Output:**

****

**Exercise 8: Implementing the Strategy Pattern**

**Code:**

interface PaymentStrategy {  
 void pay(double amount);  
}  
class CreditCardPayment implements PaymentStrategy {  
 private String number;  
 private String name;  
  
 public CreditCardPayment(String number, String name) {  
 this.number = number;  
 this.name = name;  
 }  
  
 @Override  
 public void pay(double amount) {  
 System.*out*.println("Paid ₹" + amount + " using Credit Card (" + name + ").");  
 }  
}  
class PayPalPayment implements PaymentStrategy {  
 private String email;  
  
 public PayPalPayment(String email) {  
 this.email = email;  
 }  
  
 @Override  
 public void pay(double amount) {  
 System.*out*.println("Paid ₹" + amount + " using PayPal account: " + email);  
 }  
}  
class PaymentContext {  
 private PaymentStrategy strategy;  
  
 // Set strategy at runtime  
 public void setPaymentStrategy(PaymentStrategy strategy) {  
 this.strategy = strategy;  
 }  
  
 public void processPayment(double amount) {  
 if (strategy == null) {  
 System.*out*.println("No payment method selected.");  
 } else {  
 strategy.pay(amount);  
 }  
 }  
}  
  
public class StrategyPattern {  
 public static void main(String[] args) {  
 PaymentContext context = new PaymentContext();  
  
 context.setPaymentStrategy(new CreditCardPayment("4685-0976-9472-0476", "Sanjai"));  
 context.processPayment(1500);  
  
 context.setPaymentStrategy(new PayPalPayment("sanjai.sonatech@gmail.com"));  
 context.processPayment(3000);  
 }  
}

**Output:**

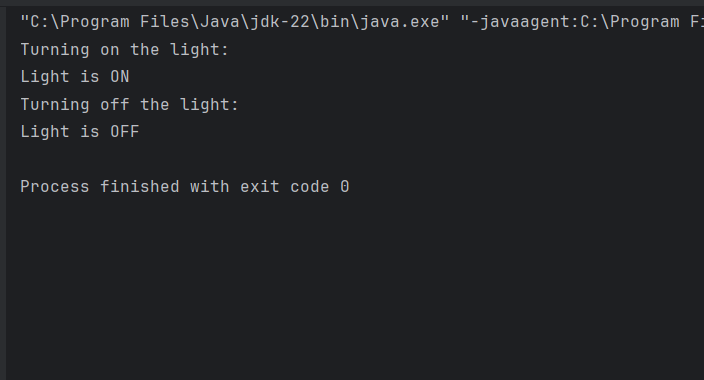
****

**Exercise 9: Implementing the Command Pattern**

**Code:**

interface Command {  
 void execute();  
}  
class LightOnCommand implements Command {  
 private Light light;  
  
 public LightOnCommand(Light light) {  
 this.light = light;  
 }  
  
 @Override  
 public void execute() {  
 light.turnOn();  
 }  
}  
class LightOffCommand implements Command {  
 private Light light;  
  
 public LightOffCommand(Light light) {  
 this.light = light;  
 }  
  
 @Override  
 public void execute() {  
 light.turnOff();  
 }  
}  
class Light {  
 public void turnOn() {  
 System.*out*.println("Light is ON");  
 }  
  
 public void turnOff() {  
 System.*out*.println("Light is OFF");  
 }  
}  
class RemoteControl {  
 private Command command;  
  
 public void setCommand(Command command) {  
 this.command = command;  
 }  
  
 public void pressButton() {  
 if (command != null) {  
 command.execute();  
 } else {  
 System.*out*.println("No command set.");  
 }  
 }  
}  
  
public class CommandPattern {  
 public static void main(String[] args) {  
 Light room = new Light();  
  
 Command On = new LightOnCommand(room);  
 Command Off = new LightOffCommand(room);  
  
 RemoteControl remote = new RemoteControl();  
  
 System.*out*.println("Turning on the light:");  
 remote.setCommand(On);  
 remote.pressButton();  
  
 System.*out*.println("Turning off the light:");  
 remote.setCommand(Off);  
 remote.pressButton();  
  
 }  
}

**Output:**

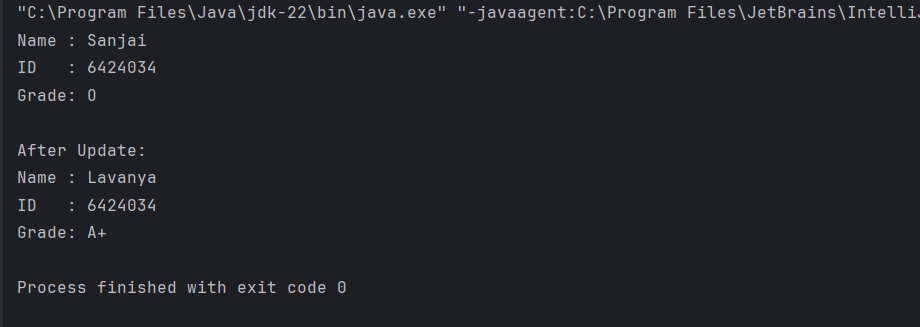
****

**Exercise 10: Implementing the MVC Pattern**

**Code:**

class Student {  
 private String name;  
 private String id;  
 private String grade;  
  
 public Student(String name, String id, String grade) {  
 this.name = name;  
 this.id = id;  
 this.grade = grade;  
 }  
  
 public String getName() { return name; }  
 public String getId() { return id; }  
 public String getGrade() { return grade; }  
  
 public void setName(String name) { this.name = name; }  
 public void setId(String id) { this.id = id; }  
 public void setGrade(String grade) { this.grade = grade; }  
}  
  
class StudentView {  
 public void displayStudentDetails(String name, String id, String grade) {  
 System.*out*.println("Name : " + name);  
 System.*out*.println("ID : " + id);  
 System.*out*.println("Grade: " + grade);  
 }  
}  
class StudentController {  
 private Student model;  
 private StudentView view;  
  
 public StudentController(Student model, StudentView view) {  
 this.model = model;  
 this.view = view;  
 }  
  
 public void setStudentName(String name) { model.setName(name); }  
 public void setStudentId(String id) { model.setId(id); }  
 public void setStudentGrade(String grade) { model.setGrade(grade); }  
  
 public String getStudentName() { return model.getName(); }  
 public String getStudentId() { return model.getId(); }  
 public String getStudentGrade() { return model.getGrade(); }  
  
 public void updateView() {  
 view.displayStudentDetails(model.getName(), model.getId(), model.getGrade());  
 }  
}  
  
public class MVCPattern {  
 public static void main(String[] args) {  
 Student student = new Student("Sanjai", "6424034", "O");  
  
 StudentView view = new StudentView();  
  
 StudentController controller = new StudentController(student, view);  
  
 controller.updateView();  
  
 controller.setStudentName("Lavanya");  
 controller.setStudentGrade("A+");  
  
 System.*out*.println("\nAfter Update:");  
 controller.updateView();  
  
 }  
}

**Output:**

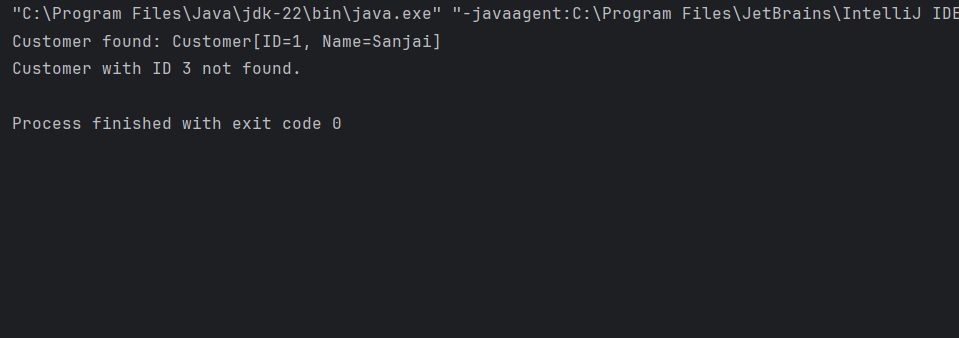
****

**Exercise 11: Implementing Dependency Injection**

**Code:**

import java.util.\*;  
interface CustomerRepository {  
 Customer findCustomerById(int id);  
}  
class Customer {  
 private int id;  
 private String name;  
  
 public Customer(int id, String name) {  
 this.id = id;  
 this.name = name;  
 }  
  
 public int getId() { return id; }  
 public String getName() { return name; }  
  
 @Override  
 public String toString() {  
 return "Customer[ID=" + id + ", Name=" + name + "]";  
 }  
}  
  
  
class CustomerRepositoryImpl implements CustomerRepository {  
 private Map<Integer, Customer> customers = new HashMap<>();  
  
 public CustomerRepositoryImpl() {  
 customers.put(1, new Customer(1, "Sanjai"));  
 customers.put(2, new Customer(2, "Cognizant"));  
 }  
  
 @Override  
 public Customer findCustomerById(int id) {  
 return customers.get(id);  
 }  
}  
class CustomerService {  
 private CustomerRepository repo;  
  
 public CustomerService(CustomerRepository repo) {  
 this.repo = repo;  
 }  
  
 public void displayCustomerById(int id) {  
 Customer customer = repo.findCustomerById(id);  
 if (customer != null) {  
 System.*out*.println("Customer found: " + customer);  
 } else {  
 System.*out*.println("Customer with ID " + id + " not found.");  
 }  
 }  
}  
  
public class DependencyInjection {  
 public static void main(String[] args) {  
 CustomerRepository repo = new CustomerRepositoryImpl();  
  
 CustomerService service = new CustomerService(repo);  
  
 service.displayCustomerById(1);  
 service.displayCustomerById(3);  
  
 }  
}

**Output:**

****