Q1

a) Make Abstract Class Order and Extend Class to BulkOrder and SmallOrder.

**Order Class**

package retail;  
  
import java.math.BigDecimal;  
import java.math.RoundingMode;  
import java.util.Collections;  
import java.util.List;  
  
public abstract class Order {  
 protected final List<Product> items;  
 protected final CreditCardDetails creditCardDetails;  
 protected final Address billingAddress;  
 protected final Address shippingAddress;  
 protected final Courier courier;  
  
 public Order(List<Product> items, CreditCardDetails creditCardDetails, Address billingAddress, Address shippingAddress, Courier courier) {  
 this.items = Collections.*unmodifiableList*(items);  
 this.creditCardDetails = creditCardDetails;  
 this.billingAddress = billingAddress;  
 this.shippingAddress = shippingAddress;  
 this.courier = courier;  
 }  
  
 public abstract void process();  
  
 protected BigDecimal round(BigDecimal amount) {  
 return amount.setScale(2, RoundingMode.*CEILING*);  
 }  
}

**BulkOrder Class**

package retail;  
  
import java.math.BigDecimal;  
import java.math.RoundingMode;  
import java.util.Collections;  
import java.util.List;  
  
public class BulkOrder extends Order {  
  
 private final BigDecimal discount;  
  
 public BulkOrder(  
 List<Product> items,  
 CreditCardDetails creditCardDetails,  
 Address billingAddress,  
 Address shippingAddress,  
 Courier courier,  
 BigDecimal discount) {  
 super(items, creditCardDetails, billingAddress, shippingAddress, courier);  
 this.discount = discount;  
 }  
  
 @Override  
 public void process() {  
  
 BigDecimal total = new BigDecimal(0);  
  
 for (Product item : items) {  
 total = total.add(item.unitPrice());  
 }  
  
 if (items.size() > 10) {  
 total = total.multiply(BigDecimal.*valueOf*(0.8));  
 } else if (items.size() > 5) {  
 total = total.multiply(BigDecimal.*valueOf*(0.9));  
 }  
  
 total = total.subtract(discount);  
  
 CreditCardProcessor.*getInstance*().charge(round(total), creditCardDetails, billingAddress);  
  
 courier.send(new Parcel(items), shippingAddress);  
 }  
  
}

**SmallOrder Class**

package retail;  
  
import java.math.BigDecimal;  
import java.math.RoundingMode;  
import java.util.Collections;  
import java.util.List;  
  
public class SmallOrder extends Order{  
  
 private static final BigDecimal *GIFT\_WRAP\_CHARGE* = new BigDecimal(3);  
 private final boolean giftWrap;  
  
 public SmallOrder(  
 List<Product> items,  
 CreditCardDetails creditCardDetails,  
 Address billingAddress,  
 Address shippingAddress,  
 Courier courier,  
 boolean giftWrap) {  
 super(items, creditCardDetails, billingAddress, shippingAddress, courier);  
 this.giftWrap = giftWrap;  
 }  
  
 public void process() {  
  
 BigDecimal total = new BigDecimal(0);  
  
 for (Product item : items) {  
 total = total.add(item.unitPrice());  
 }  
  
 total = total.add(courier.deliveryCharge());  
  
 if (giftWrap) {  
 total = total.add(*GIFT\_WRAP\_CHARGE*);  
 }  
  
 CreditCardProcessor.*getInstance*().charge(round(total), creditCardDetails, billingAddress);  
  
 if (giftWrap) {  
 courier.send(new GiftBox(items), shippingAddress);  
 } else {  
 courier.send(new Parcel(items), shippingAddress);  
 }  
 }  
}

b) Make a OrderBuilder Class and then modify RetailExample.java class according Builder pattern.  
  
**OrderBuilder Class** (Not sure if CreditCardDetails, BillingAddress and Courier should be mandatory in Builder. Did them optional)

package retail;  
  
import java.math.BigDecimal;  
import java.util.List;  
  
public class OrderBuilder {  
 private List<Product> items;  
 private CreditCardDetails creditCardDetails;  
 private Address billingAddress;  
 private Address shippingAddress = null;  
 private Courier courier;  
 private BigDecimal discount = new BigDecimal("0");  
 private boolean giftWrap = false;  
  
 public OrderBuilder addItem(Product item) {  
 this.items.add(item);  
 return this;  
 }  
  
 public OrderBuilder withCreditCardDetails(CreditCardDetails creditCardDetails) {  
 this.creditCardDetails = creditCardDetails;  
 return this;  
 }  
  
 public OrderBuilder withBillingAddress(Address billingAddress) {  
 this.billingAddress = billingAddress;  
 if(shippingAddress == null)  
 this.shippingAddress = billingAddress;  
 return this;  
 }  
  
 public OrderBuilder withShippingAddress(Address shippingAddress) {  
 this.shippingAddress = shippingAddress;  
 return this;  
 }  
  
 public OrderBuilder withCourier(Courier courier) {  
 this.courier = courier;  
 return this;  
 }  
  
 public OrderBuilder withGiftWrap(boolean giftWrap) {  
 this.giftWrap = giftWrap;  
 return this;  
 }  
  
 public OrderBuilder withDiscount(BigDecimal discount) {  
 this.discount = discount;  
 return this;  
 }  
  
 public Order build() {  
 if(items.size() <= 3){  
 return new SmallOrder(items, creditCardDetails, billingAddress, shippingAddress, courier, giftWrap);  
 }  
 else{  
 return new BulkOrder(items, creditCardDetails, billingAddress, shippingAddress, courier, discount);  
 }  
 }  
}

**RetailExample.java**

import retail.\*;  
  
import java.math.BigDecimal;  
import java.util.List;  
  
public class RetailExample {  
  
 public static void main(String[] args) {  
  
 Courier royalMail = new RoyalMail();  
 Courier fedex = new Fedex();  
  
 Order bigOrder = new OrderBuilder()  
 .addItem(new Product("One Book", new BigDecimal("10.00")))  
 .addItem(new Product("One Book", new BigDecimal("10.00")))  
 .addItem(new Product("One Book", new BigDecimal("10.00")))  
 .addItem(new Product("One Book", new BigDecimal("10.00")))  
 .withCreditCardDetails(new CreditCardDetails("1234123412341234", 111))  
 .withBillingAddress(new Address("180 Queens Gate, London, SW7 2AZ"))  
 .withCourier(fedex)  
 .withDiscount(BigDecimal.*ZERO*)  
 .build();  
  
// new BulkOrder(  
// List.of(  
// new Product("One Book", new BigDecimal("10.00")),  
// new Product("One Book", new BigDecimal("10.00")),  
// new Product("One Book", new BigDecimal("10.00")),  
// new Product("One Book", new BigDecimal("10.00")),  
// new Product("One Book", new BigDecimal("10.00")),  
// new Product("One Book", new BigDecimal("10.00"))),  
// new CreditCardDetails("1234123412341234", 111),  
// new Address("180 Queens Gate, London, SW7 2AZ"),  
// new Address("180 Queens Gate, London, SW7 2AZ"),  
// fedex,  
// BigDecimal.ZERO);  
  
 bigOrder.process();  
  
 Order smallOrder = new OrderBuilder()  
 .addItem(new Product("One Book", new BigDecimal("10.00")))  
 .withCreditCardDetails(new CreditCardDetails("1234123412341234", 111))  
 .withBillingAddress(new Address("180 Queens Gate, London, SW7 2AZ"))  
 .withCourier(royalMail)  
 .withGiftWrap(false)  
 .build();  
  
  
// new SmallOrder  
// List.of(  
// new Product("One Book", new BigDecimal("10.00"))  
// ),  
// new CreditCardDetails("1234123412341234", 111),  
// new Address("180 Queens Gate, London, SW7 2AZ"),  
// new Address("180 Queens Gate, London, SW7 2AZ"),  
// royalMail,  
// false);  
  
 smallOrder.process();  
 }  
}

c)

i) The credit card processor uses a singleton pattern with eager initialization. A singleton is a pattern that restricts the possibility of having more than 1 instance of a class. This pattern should be used carefully and only when a constraint like this is necessary (e.g. when instantiating a large object that takes a large amount of memory)

ii) The use of a singleton produces many problems:

* Introduce a global variable -> In general this kind of variables are not welcome because they are hard to reason about
* Introduce dependencies into our system that span widely along the object graph
* Introduce thigh coupling (dependency)
  + Make difficult to reuse code
  + Make testing hard -> testing in isolation is not possible because singleton is necesary

iii) Apply Dependency Inversion Pattern in following classes:

* CreditCardProcessor
* CardProcessor (Interface created)
* Order, BulkOrder and SmallOrder
* RetailExample.java file

**CreditCardProcessor**

package retail;  
  
import java.math.BigDecimal;  
  
// DEPENDENCY INVERSION:  
// Introduce Interface CardProcessor in CreditCardProcessor Class  
public class CreditCardProcessor implements CardProcessor {  
  
 private static final CreditCardProcessor *INSTANCE* = new CreditCardProcessor();  
  
 private CreditCardProcessor() {}  
  
 // DEPENDENCY INVERSION:  
 // Applying Dependency Inversion  
 // Changing return type of method getInstance() from CreditCardProcessor to CardProcessor  
 public static CardProcessor getInstance() {  
 return *INSTANCE*;  
 }  
  
  
 @Override  
 public void charge(BigDecimal amount, CreditCardDetails account, Address billingAddress) {  
  
 System.*out*.println("Credit card charged: " + account + " amount: " + amount);  
  
 // further implementation omitted for exam question  
 }  
}

**CardProcessor**

package retail;  
  
import java.math.BigDecimal;  
  
// DEPENDENCY INVERSION:  
// Creating Interface  
public interface CardProcessor {  
 void charge(BigDecimal amount, CreditCardDetails account, Address billingAddress);  
}

**Order**

**…**

public abstract void process(CardProcessor cardProcessor);

**...**

**BulkOrder**

**…**

// DEPENDENCY INVERSION:  
// Applying Dependency Inversion. Passing CardProcessor as Parameter  
@Override  
public void process(CardProcessor cardProcessor) {  
  
 BigDecimal total = new BigDecimal(0);  
  
 for (Product item : items) {  
 total = total.add(item.unitPrice());  
 }  
  
 if (items.size() > 10) {  
 total = total.multiply(BigDecimal.*valueOf*(0.8));  
 } else if (items.size() > 5) {  
 total = total.multiply(BigDecimal.*valueOf*(0.9));  
 }  
  
 total = total.subtract(discount);  
  
 // DEPENDENCY INVERSION:  
 // Applying Dependency Inversion. Changing CreditCardProcessor for cardProcessor  
 cardProcessor.charge(round(total), creditCardDetails, billingAddress);  
  
 courier.send(new Parcel(items), shippingAddress);  
}

**...**

**SmallOrder**

**…**

// DEPENDENCY INVERSION:  
// Applying Dependency Inversion. Passing CardProcessor as Parameter   
public void process(CardProcessor cardProcessor) {  
  
 BigDecimal total = new BigDecimal(0);  
  
 for (Product item : items) {  
 total = total.add(item.unitPrice());  
 }  
  
 total = total.add(courier.deliveryCharge());  
  
 if (giftWrap) {  
 total = total.add(*GIFT\_WRAP\_CHARGE*);  
 }  
  
 // DEPENDENCY INVERSION:  
 // Applying Dependency Inversion. Changing CreditCardProcessor for cardProcessor  
 cardProcessor.charge(round(total), creditCardDetails, billingAddress);  
  
 if (giftWrap) {  
 courier.send(new GiftBox(items), shippingAddress);  
 } else {  
 courier.send(new Parcel(items), shippingAddress);  
 }  
}

**...**

**RetailExample.java**

…

// DEPENDENCY INVERSION: Pass a CreditCardProcessor singleton as parameter  
bigOrder.process(CreditCardProcessor.*getInstance*());

…

…

// DEPENDENCY INVERSION: Pass a CreditCardProcessor singleton as parameter  
smallOrder.process(CreditCardProcessor.*getInstance*());

…

Then create class OrderTest so you can tests orders whitout the need of depending on a singleton (for running *process()* method). For this purpose we create a CardProcessor Mock Object

package retail;  
  
import org.jmock.Expectations;  
import org.jmock.integration.junit4.JUnitRuleMockery;  
import org.junit.Rule;  
import org.junit.Test;  
  
import java.math.BigDecimal;  
  
  
public class OrderTest {  
  
  
  
 private final Order bigOrder = new OrderBuilder()  
 .addItem(new Product("One Book", new BigDecimal("10.00")))  
 .addItem(new Product("One Book", new BigDecimal("10.00")))  
 .addItem(new Product("One Book", new BigDecimal("10.00")))  
 .addItem(new Product("One Book", new BigDecimal("10.00")))  
 .withCreditCardDetails(account)  
 .withBillingAddress(address)  
 .withCourier(fedex)  
 .build();  
  
 private final Order smallOrder = new OrderBuilder()  
 .addItem(new Product("One Book", new BigDecimal("10.00")))  
 .withCreditCardDetails(account)  
 .withBillingAddress(address)  
 .withCourier(fedex)  
 .build();  
  
  
 @Rule  
 public JUnitRuleMockery context = new JUnitRuleMockery();  
  
 CardProcessor cardProcessor = context.mock(CardProcessor.class);  
  
 @Test  
 public void processNewSmallOrderInCardProcessor() {  
  
 context.checking(new Expectations(){{  
 exactly(1).of(cardProcessor).charge(new BigDecimal("18.00"),account,address);  
 }});  
 smallOrder.process(cardProcessor);  
 }  
  
 @Test  
 public void processNewBigOrderInCardProcessor() {  
  
 context.checking(new Expectations(){{  
 exactly(1).of(cardProcessor).charge(new BigDecimal("40.00"),account,address);  
 }});  
 bigOrder.process(cardProcessor);  
 }  
  
 class Fedex implements Courier {  
 @Override  
 public void send(Parcel shipment, Address shippingAddress) {  
 System.*out*.println("Fedex will deliver your parcel to: " + shippingAddress);  
 }  
  
 @Override  
 public BigDecimal deliveryCharge() {  
 return new BigDecimal(8);  
 }  
 }  
  
}

**Q2**

a), b), c), d) All together

**ScorerEngine Class**

package tennis;  
  
import java.util.ArrayList;  
import java.util.List;  
  
public class ScorerEngine {  
 private int playerOneScore;  
 private int playerTwoScore;  
 private final String[] scoreNames;  
 private List<Updatable> observers = new ArrayList<>();  
  
 public ScorerEngine(String[] scoreNames){  
 this.playerOneScore = 0;  
 this.playerTwoScore = 0;  
 this.scoreNames = scoreNames;  
 }  
  
 public void addObserver(Updatable observer){  
 observers.add(observer);  
 }  
  
 private void notifyObservers() {  
 for (Updatable observer : observers) {  
 observer.updateWith(this.score());  
 }  
 }  
  
 public String score() {  
  
 if (playerOneScore > 2 && playerTwoScore > 2) {  
 int difference = playerOneScore - playerTwoScore;  
 switch (difference) {  
 case 0:  
 return "Deuce";  
 case 1:  
 return "Advantage Player 1";  
 case -1:  
 return "Advantage Player 2";  
 case 2:  
 return "Game Player 1";  
 case -2:  
 return "Game Player 2";  
 }  
 }  
  
 if (playerOneScore > 3) {  
 return "Game Player 1";  
 }  
 if (playerTwoScore > 3) {  
 return "Game Player 2";  
 }  
 if (playerOneScore == playerTwoScore) {  
 return scoreNames[playerOneScore] + " all";  
 }  
 return scoreNames[playerOneScore] + " - " + scoreNames[playerTwoScore];  
 }  
  
 public void playerOneWinsPoint() {  
 playerOneScore++;  
 notifyObservers();  
 }  
  
 public void playerTwoWinsPoint() {  
 playerTwoScore++;  
 notifyObservers();  
 }  
  
 public boolean gameHasEnded() {  
 return score().contains("Game");  
 }  
  
  
}

**TennisScorer Class**

package tennis;  
  
import javax.swing.\*;  
  
public class TennisScorer implements Updatable{  
  
 private final JTextField scoreDisplay = new JTextField(20);  
  
 public static void main(String[] args) {  
 new TennisScorer().display();  
 }  
  
 private void display() {  
  
 // Create instance of an Scorer Engine (Model)  
 String[] scoreNames = {"Love", "15", "30", "40"};  
 ScorerEngine scorerEngine = new ScorerEngine(scoreNames);  
  
 scorerEngine.addObserver(this);  
  
 JFrame window = new JFrame("Tennis");  
 window.setSize(400, 150);  
  
 JButton playerOneScores = new JButton("Player One Scores");  
 JButton playerTwoScores = new JButton("Player Two Scores");  
  
  
 scoreDisplay.setHorizontalAlignment(JTextField.*CENTER*);  
 scoreDisplay.setEditable(false);  
  
 playerOneScores.addActionListener(  
 e -> {  
 scorerEngine.playerOneWinsPoint();  
  
 if (scorerEngine.gameHasEnded()) {  
 playerOneScores.setEnabled(false);  
 playerTwoScores.setEnabled(false);  
 }  
 });  
  
 playerTwoScores.addActionListener(  
 e -> {  
 scorerEngine.playerTwoWinsPoint();  
 if (scorerEngine.gameHasEnded()) {  
 playerOneScores.setEnabled(false);  
 playerTwoScores.setEnabled(false);  
 }  
 });  
  
 JPanel panel = new JPanel();  
 panel.add(playerOneScores);  
 panel.add(playerTwoScores);  
 panel.add(scoreDisplay);  
  
 window.add(panel);  
  
 window.setDefaultCloseOperation(WindowConstants.*EXIT\_ON\_CLOSE*);  
 window.setVisible(true);  
  
 }  
  
 // Query Methods to Model  
 @Override  
 public void updateWith(String score){  
 scoreDisplay.setText(score);  
 }  
  
}

**Updatable Interface**

package tennis;  
  
public interface Updatable {  
 void updateWith(String score);  
}

**ScorerEngineTest**

package tennis;  
  
import org.jmock.Expectations;  
import org.jmock.integration.junit4.JUnitRuleMockery;  
import org.junit.Assert;  
import org.junit.Rule;  
import org.junit.Test;  
  
import java.util.ArrayList;  
import java.util.List;  
  
import static org.hamcrest.MatcherAssert.*assertThat*;  
import static org.hamcrest.Matchers.*is*;  
  
public class ScorerEngineTest {  
  
 private final String[] scoreNames = {"0", "10", "20", "30"};  
 private final ScorerEngine engine = new ScorerEngine(scoreNames);  
  
 @Rule  
 public JUnitRuleMockery context = new JUnitRuleMockery();  
 Updatable display = context.mock(Updatable.class);  
  
 @Test  
 public void playerOneScoreAndGetsFirstPointAfterFirstPlay(){  
 engine.playerOneWinsPoint();  
 String output = engine.score();  
 *assertThat*(output,*is*("10 - 0"));  
 }  
  
 @Test  
 public void playerTwoScoreAndMatchPlayersOneAfterSecondPlay(){  
 engine.playerTwoWinsPoint();  
 engine.playerOneWinsPoint();  
 String output = engine.score();  
 *assertThat*(output,*is*("10 all"));  
 }  
  
 @Test  
 public void playerOneWinsGame(){  
 engine.playerOneWinsPoint();  
 engine.playerTwoWinsPoint();  
 engine.playerOneWinsPoint();  
 engine.playerTwoWinsPoint();  
 engine.playerOneWinsPoint();  
 engine.playerOneWinsPoint();  
 String output = engine.score();  
 *assertThat*(output,*is*("Game Player 1"));  
 }  
  
 // JMock Tests  
  
 @Test  
 public void updatesDisplayWhenAnyPlayerScores(){  
 engine.addObserver(display);  
  
 context.checking(new Expectations(){{  
 exactly(1).of(display).updateWith("10 - 0");  
 }});  
  
 engine.playerOneWinsPoint();  
 }  
  
}