Code for the Online Stock Brokerage System

Write the object-oriented code to implement the design of the 'online stock brokerage system' problem.

We'll cover the following

- Online stock brokerage system classes
  - Enumerations and custom data type
  - Account
  - Watchlist and stock
  - Search and stock inventory
  - Stock position and stock lot
  - Order
  - Transfer money
  - Notification
  - Stock exchange
- Wrapping up

We've reviewed different aspects of the online stock brokerage system and observed the attributes attached to the problem using various UML diagrams. Let's explore the more practical side of things, where we will work on implementing the online stock brokerage system using multiple languages. This is usually the last step in an object-oriented design interview process.

We have chosen the following languages to write the skeleton code of the different classes present in the online stock brokerage system:

- Java
- C#
- Python
- C++
- JavaScript

Online stock brokerage system classes

In this section, we'll provide the skeleton code of the classes designed in the class diagram lesson.

> **Note:** For simplicity, we are not defining getter and setter functions. The reader can assume that all class attributes are private and accessed through their respective public getter methods and modified only through their public method functions.

Enumerations and custom data type

First of all, we will define all the enumerations required in the stock brokerage system. According to the class diagram, there are three enumerations used in the system i.e. `OrderStatus`, `ElasticEnforcementType` and `AccountStatus`. The code to implement these enumerations and custom data types is as follows:

> **Note:** JavaScript does not support enumerations, so we will be using the `Object.freeze()` method as an alternative that freezes an object and prevents further modifications.

```
// Enumeration
enum OrderStatus {
  OPEN,
  FILLED,
  PARTIALLY_FILLED,
  CANCELED
}

enum TimeInforcementType {
  GOOD_TILL_CANCELED,
  FILL_OR_KILL,
  IMMEDIATE_OR_CANCEL,
  ON_THE_OPEN,
  ON_THE_CLOSE
}

enum AccountStatus {
  ACTIVE,
  CLOSED,
  CANCELED,
  BLACKLISTED,
  NONE
}

// Custom Address data type class
public class Address {
  private int zipCode;
  private String address;
  private String city;
  private String state;
  ...
```

Definition of enums and custom datatypes

Account

The `Account` class will be an abstract class, which will have the actors, `Admin` and `Member`, as child classes. The definition of these classes is given below:

```
// Account is an abstract class
public abstract class Account {
  private String id;
  private String password;
  private String name;
  private AccountStatus status;
  private Address address;
  private String email;
  private String phone;

  public abstract boolean resetPassword();
}

public class Member extends Account {
  private double availableFundsForTrading;
  private Date dateOfMembership;
  private MarginString, StockPositions: stockPositions;
  private HashMap<Integer, Order> activeOrders;

  public ErrorCode placeSell(int OrderString stockId, float quantity, int limitPrice, TimeInforcementType enforce
  public ErrorCode placeBuy(int OrderString stockId, float quantity, int limitPrice, TimeInforcementType enforce
  public void callbackStockExchange(int orderId, List<OrderPart> orderParts, OrderStatus status);
  public boolean resetPassword(){
    // definition
  }
}

public class Admin extends Account {
  public boolean blockMember();
  public boolean unblockMember();...
```

Account and its child classes

Watchlist and stock

A `Watchlist` class is a list of stocks that an investor keeps an eye on, to profit from price drops. The `Stock` class is an equity or a security that represents a portion of the issuing company's ownership.

```
public class Watchlist {
  private String name;
  private List<Stock> stocks;

  public List<Stock> getStocks();
}

public class Stock {
  private String symbol;
  private double price;
}
```

The Watchlist and the Stock classes

Search and stock inventory

The `StockInventory` class implements the `Search` interface.

```
public interface Search {
  // Interface method (does not have a body)
  public Stock searchSymbol(String symbol);
}

public class StockInventory implements Search {
  private String inventoryName;
  private Date lastUpdate;
  public Stock searchSymbol(String symbol) {
    // definition
  }
}
```

The Search interface and the StockInventory class

Stock position and stock lot

All the stocks that the user owns will be included in the `StockPosition` class. A member may purchase various lots of the same stock at various dates. The `StockLot` class will represent these particular lots.

```
public class StockPosition {
  private String symbol;
  private double quantity;
}

public class StockLot {
  private String lotName;
  private Order buyingOrder;

  public double getBuyingPrice();
}
```

The StockPostion and StockLot classes

Order

Members can place stock trading orders when they want to sell or acquire `StockPosition`.

```
public class OrderPart {
  private double price;
  private double quantity;
  private Date executedAt;
}

// Order is an abstract class
public abstract class Order {
  private String orderNumber;
  public boolean isBuyOrder;
  private OrderStatus status;
  private TimeInforcementType timeInforcement;
  private Date creationTime;
  private HashMap<int, OrderPart> parts;

  public void setStatus(OrderStatus status);
  public boolean sendToDatabase();
  public void addOrderParts(OrderParts parts);
}

public class LimitOrder extends Order {
}

public class StopLimitOrder extends Order {
}

public class StoplossOrder extends Order {
}

public class MarketOrder extends Order {
}
```

OrderPart, Order and its derived classes

Transfer money

Members should be able to deposit and withdraw money either via `Check`, `Wire`, or `ElectronicBank` transfer.

```
// TransferMoney is an abstract class
public abstract class TransferMoney {
  private int id;
  private Date creationDate;
  public int fromAccount;
  private int toAccount;

  public abstract boolean initiateTransaction();
}

public class ElectronicBank extends TransferMoney {
  private String bankName;

  public boolean initiateTransaction(){
    // definition
  }
}

public class Wire extends TransferMoney {
  private int wire;

  public boolean initiateTransaction(){
    // definition
  }
}

public class Check extends TransferMoney {
  private String checkNumber;

  public boolean initiateTransaction(){
    // definition
  }
}
```

TransferMoney and its derived classes

Notification

The `Notification` class is another abstract class responsible for sending notifications to the users, with the `SmsNotification` and `MailNotification` classes as its child. The implementation of this class is shown below:

```
public abstract class Notification {
  private String notificationId;
  private Date creationDate;
  private String content;

  public abstract boolean sendNotification();
}

public class SmsNotification extends Notification {
  private int phoneNumber;

  public boolean sendNotification(){
    // definition
  }
}

public class EmailNotification extends Notification {
  private String email;

  public boolean sendNotification(){
    // definition
  }
}
```

Notification and its derived classes

Stock exchange

The stock brokerage system will get all the stocks and their current pricing from the `StockExchange` class.

```
public class StockExchange {
  // The StockExchange is a singleton class that ensures it will have only one active instance at a time
  private static StockExchange instance = null;

  // Created a private constructor to add a restriction (due to Singleton)
  private StockExchange() {}

  // Created a static method to access the singleton instance of StockExchange
  public static StockExchange getInstance()
  {
    if(instance == null) {
      instance = new StockExchange();
    }
    return instance;
  }

  public boolean placeOrder(Order order);
}
```

The StockExchange class

Wrapping up

We've explored the complete design of an online stock brokerage system in this chapter. We've looked at how a basic online stock brokerage system can be visualized using various UML diagrams and designed using object-oriented principles and design patterns.