```
/**
 * @author Harley Phung - Project 3
 * Task 8: Create a market list of transactions and start to trade between orders
 */

import java.util.NoSuchElementException;

public class Market {
  /** Create a first Node of the buy list */
  private LLNode<Order> firstBuyNode = null;

  /** Create a first node of the sell list */
  private LLNode<Order> firstSellNode = null;

  /** A field that stored market stock symbol */
  private char stockSymbol = ' ';

  /** A field that stored the transaction number */
  private int transactionNumber = 0;

  /**
   * Construct the new list for the market
   * @param stockSymbol  the stock symbol of the market
   * @param firstBuyNode  the first node in the buy list
   * @param firstSellNode  the first node in the sell list
   */
  public Market(char stockSymbol, LLNode<Order> firstBuyNode, LLNode<Order>
firstSellNode) {
    this.stockSymbol = stockSymbol;
    this.firstBuyNode = firstBuyNode;
    this.firstSellNode = firstSellNode;

  }

  /**
   * Returns the first node of the buy list
   * @return firstBuyNode  the first node of the buy list
   */
  protected LLNode<Order> getFirstBuyNode() {
    return this.firstBuyNode;
  }

  /**
   * Returns the first node of the sell list
   * @return firstSellNode  the first node of the sell list
   */
  protected LLNode<Order> getFirstSellNode() {
    return this.firstSellNode;
  }

  /**
   * Changes the first node of the buy list
   * @param firstBuyNode  the node that will be the first node of the buy list
   */
  protected void setFirstBuyNode(LLNode<Order> firstBuyNode) {
    this.firstBuyNode = firstBuyNode;
  }

  /**
```

```
 * Changes the first node of the sell list
 * @param firstSellNode  the node that will be the first node of the sell list
 */
protected void setFirstSellNode(LLNode<Order> firstSellNode) {
  this.firstSellNode = firstSellNode;
}

/**
 * Returns the stock symbol of the market
 * @return stock symbol  the stock symbol of the market
 */
public char getStockSymbol() {
  return this.stockSymbol;
}

/**
 * Returns list of LLNode containing all the sell orders
 * @return firstSellNode  the first sell node in sell list
 */
public LLNode<Order> getSellOrders() {
  return this.firstSellNode;
}

/**
 * Returns list of LLNode containing all the Buy orders
 * @return firstBuyNode  the first buy node in buy list
 */
public LLNode<Order> getBuyOrders() {
  return this.firstBuyNode;
}

/**
 * Returns a list of LLNodes contianing all the buy and sell orders
 * @param order  containing LLNodes list for buy and sell orders
 * @return returnHead  the first open orders in the market
 */
public LLNode<Order> getOpenOrders(Trader trader) {
  LLNode<Order> b = this.firstBuyNode;
  LLNode<Order> s = this.firstSellNode;
  LLNode<Order> returnHead = null; //head of the returned list
  LLNode<Order> returnTail = null; //tail of the returned list
  while (b != null) {
    Order tempOrder = b.getElement();
    if (tempOrder.getTrader().equals(trader)) {
      if(returnHead == null) {
        returnHead = new LLNode<Order>(tempOrder, null);
        returnTail = returnHead;
      }
      else {
        returnTail.setNext(new LLNode<Order>(tempOrder,null));
        returnTail = returnTail.getNext();
      }
    }
    b = b.getNext();
  }
  while(s != null) {
    Order tempOrder = s.getElement();
    if (tempOrder.getTrader().equals(trader)) {
      if(returnHead == null) {
```

```
          returnHead = new LLNode<Order>(tempOrder, null);
          returnTail = returnHead;
        }
        else {
          returnTail.setNext(new LLNode<Order>(tempOrder, null));
          returnTail = returnTail.getNext();
        }
      }
      s = s.getNext();
    }
    return returnHead;
  }


  /**
   * Returns the highest buy price of the not-all-or-nothing buy orders
   * @return highestBuyPrice  the highest buy price of not-all-or-nothing buy
orders
   */
  public double getCurrentBuyPrice() {
    double highestBuyPrice = this.firstBuyNode.getElement().getPrice();
    return highestBuyPrice;
  }

  /**
   * Returns the lowest sell price of the not-all-or-nothing sell orders
   * @return lowestSellPrice  the lowest sell price of not-all-or-nothing sell
orders
   */
  public double getCurrentSellPrice() {
    double lowestSellPrice = this.firstSellNode.getElement().getPrice();
    return lowestSellPrice;
  }

  /**
   * Check if the market contains both MarketBuyOrder and MarketSellOrder
   * @return true  the market contains both MarketBuyOrder and MarketSellOrder
   * @return false  the market does not contain both MarketBuyOrder and
MarketSellOrder
   */
  public boolean isOpen() {
    LLNode<Order> b = this.firstBuyNode;
    LLNode<Order> s = this.firstSellNode;
    int haveMarketBuy = 0;
    int haveMarketSell = 0;
    while(b != null) {
      if(b.getElement() instanceof MarketBuyOrder) {
        haveMarketBuy = 1;
        break;
      }
      b = b.getNext();
    }
    while(s != null) {
      if(s.getElement() instanceof MarketSellOrder) {
        haveMarketSell = 1;
        break;
      }
      s = s.getNext();
    }
```

```
  if (haveMarketBuy == 1 && haveMarketSell == 1) {
    return true;
  }
  return false;
}

/**
 * Check if the order is a valid order
 * @return true  the order is valid
 * @return false  the order is not valid
 */
public boolean isValidOrder(Order o) {
  if(o.getStockSymbol() == this.getStockSymbol()) {
    if(o instanceof BuyLimitOrder) {
      LLNode<Order> s = this.firstSellNode;
      while (s != null) {
        if(s.getNext() == null) {
          if(o.getPrice() <= s.getElement().getPrice()) {
            return true;
          }
          return false;
        }
        s = s.getNext();
      }
      return false;
    }
    if(o instanceof SellLimitOrder) {
      LLNode<Order> b = this.firstBuyNode;
      while(b != null) {
        if (b.getNext() == null) {
          if(o.getPrice() >= b.getElement().getPrice()) {
            return true;
          }
          return false;
        }
        b = b.getNext();
      }
      return false;
    }
    return true;
  }
  return false;
}

/**
 * Add a new order into the market with appropriate position associate to price
 * @param newOrder  the new order that will be in market
 * @throws NoSuchElementException  when the order is not in the market
 */
public void addOrder(Order newOrder) {
  if(newOrder.getStockSymbol() != this.getStockSymbol()) {
    throw new NoSuchElementException("Not appropriate order");
  }
  else {
    LLNode<Order> b = this.firstBuyNode;
    LLNode<Order> s = this.firstSellNode;
    if(newOrder instanceof BuyOrder) {
      if(b == null) {
        b = new LLNode<Order>(newOrder, null);
```

```
        }
        else if (newOrder.getPrice() >= this.firstBuyNode.getElement().getPrice())
{
          this.firstBuyNode = new LLNode<Order>(newOrder, this.firstBuyNode);
        }
        else {
          while(b.getNext() != null && newOrder.getPrice() <=
b.getNext().getElement().getPrice()) {
            b = b.getNext();
          }
          if(b.getNext() == null) {
            b.setNext(new LLNode<Order>(newOrder, null));
          }
          else {
            b.setNext(new LLNode<Order>(newOrder, b.getNext()));
          }
        }
      }
      if(newOrder instanceof SellOrder) {
        if(s == null) {
          s = new LLNode<Order>(newOrder, null);
        }
        else if (newOrder.getPrice() <= this.firstSellNode.getElement().getPrice())
{
          this.firstSellNode = new LLNode<Order>(newOrder, this.firstSellNode);
        }
        else {
          while(s.getNext() != null && newOrder.getPrice() >=
s.getNext().getElement().getPrice()) {
            s = s.getNext();
          }
          if(s.getNext() == null) {
            s.setNext(new LLNode<Order>(newOrder, null));
          }
          else {
            s.setNext(new LLNode<Order>(newOrder, s.getNext()));
          }
        }
      }
    }

  /**
   * Remove an order from the market with the same stock symbol, number of shares,
price and trader
   * @param order  the order that is checked to remove or not
   */
  public void removeOrder(Order order) {
    LLNode<Order> b = this.firstBuyNode;
    LLNode<Order> s = this.firstSellNode;

    if(b == null) {
      ;
    }
    else {
      if(b.getElement().equals(order)) {
        this.firstBuyNode = b.getNext();
      }
      while (b.getNext() != null) {
```

```
        if(b.getNext().getElement().equals(order)) {
          b.setNext(b.getNext().getNext());
          break;
        }
        b = b.getNext();
      }
    }

    if(s == null) {
      ;
    }
    else {
      if(s.getElement().equals(order)) {
        this.firstSellNode = s.getNext();
      }
      while (s.getNext() != null) {
        if(s.getNext().getElement().equals(order)) {
          s.setNext(s.getNext().getNext());
          break;
        }
        s = s.getNext();
      }
    }
  }

  /**
   * Match 2 input orders if their price fit
   * @param bOrder  the buy Limit Order
   * @param sOrder  the sell Limit Order
   * @return true  if the 2 input orders can be matched
   * @return false  if the 2 iput orders cannot be matched
   */
  public boolean matchingOrders(Order bOrder, Order sOrder) {
    if(!(bOrder instanceof BuyOrder) && (sOrder instanceof SellOrder)) {
      return false;
    }
    if((bOrder instanceof BuyLimitOrder) && (sOrder instanceof MarketSellOrder)) {
      if (bOrder.getPrice() >= sOrder.getPrice()) {
        if(bOrder.isAllOrNone() == true){
          if (sOrder.getNumberShares() >= bOrder.getNumberShares()) {
            return true;
          }
          return false;
        }
        return true;
      }
      return false;
    }
    else if((bOrder instanceof MarketBuyOrder) && (sOrder instanceof
SellLimitOrder)) {
      if (bOrder.getPrice() >= sOrder.getPrice()) {
        if(sOrder.isAllOrNone() == true) {
          if(bOrder.getNumberShares() >= sOrder.getNumberShares()) {
            return true;
          }
          return false;
        }
        return true;
      }
```

```
      return false;
    }
    else if((bOrder instanceof BuyLimitOrder) && (sOrder instanceof
SellLimitOrder)) {
      if (bOrder.getPrice() >= sOrder.getPrice()) {
        if(bOrder.isAllOrNone() == true) {
          if(bOrder.getNumberShares() <= sOrder.getNumberShares()) {
            return true;
          }
          return false;
        }
        else if(sOrder.isAllOrNone() == true) {
          if(bOrder.getNumberShares() >= sOrder.getNumberShares()) {
            return true;
          }
          return false;
        }
        return true;
      }
      return false;
    }
    else if((bOrder instanceof MarketBuyOrder) && (sOrder instanceof
MarketSellOrder)) {
      if(bOrder.getPrice() >= sOrder.getPrice()) {
        return true;
      }
      return false;
    }
    return false;
  }


  /**
   * Place order for transactions
   * @param order  the order that will be placed
   * @return listHead  the list of Transactions
   * @throws NoSuchElementException  if the order is not valid or open
   */
  public LLNode<Transaction> placeOrder(Order order) {
    //1st situation
    if(this.isOpen() == false || this.isValidOrder(order) == false) {
      throw new NoSuchElementException("Cannot place order");
    }

    //2nd situation
    boolean matched = false; //if true then there's matched order, false then
there's no matched order
    LLNode<Order> b = this.firstBuyNode;
    LLNode<Order> s = this.firstSellNode;
    Order matchingOrder = null;
    if(order instanceof BuyOrder) {
      while (s != null){
        if (this.matchingOrders(order, s.getElement()) == true) {
          matched = true;
          matchingOrder = s.getElement();
          break;
        }
        s = s.getNext();
      }
```

```
    }
    if (order instanceof SellOrder) {
      while(b != null) {
        if (this.matchingOrders(order, b.getElement()) == false) {
          matched = true;
          matchingOrder = b.getElement();
          break;
        }
        b = b.getNext();
      }
    }

    // 3rd situation
    LLNode<Transaction> listHead = null;
    LLNode<Transaction> listTail = null;
    Transaction newTransaction = null;
    int numShares = 0;
    if(matched == false) {
      return null;
    }
    /**if there's matched order, check if there's already a list for transaction.
If not then create a new list using
      * matching order and input order. If already have one, add new transaction to
the end of the list */
    else {
      //The list is not exist
      while(true) {
        if (order instanceof BuyOrder) {
          if(order.getNumberShares() <= matchingOrder.getNumberShares()) {
            numShares = order.getNumberShares();
          }
          else {
            numShares = matchingOrder.getNumberShares();
          }
          newTransaction = new Transaction (order.getStockSymbol(), numShares,
order.getPrice(), order.getTrader(),
                                            matchingOrder.getTrader(),
this.transactionNumber);
          this.transactionNumber = this.transactionNumber + 1;
        }
        else if(order instanceof SellOrder) {
          if(order.getNumberShares() <= matchingOrder.getNumberShares()) {
            numShares = order.getNumberShares();
          }
          else {
            numShares = matchingOrder.getNumberShares();
          }
          newTransaction = new Transaction (order.getStockSymbol(), numShares,
order.getPrice(),

matchingOrder.getTrader(),order.getTrader(), this.transactionNumber);
          this.transactionNumber = this.transactionNumber + 1;
        }

        if(listHead == null) {
          listHead = new LLNode<Transaction>(newTransaction, null);
          listTail = listHead;
        }
        else {
```

```
          LLNode<Transaction> tempTransactionNode = new
LLNode<Transaction>(newTransaction, null);
          listTail.setNext(tempTransactionNode);
          listTail = listTail.getNext();
        }

        if(matchingOrder.getNumberShares() <= newTransaction.getNumberShares()) {
          if(matchingOrder instanceof MarketOrder) {
            MarketMaker newMaker = (MarketMaker) matchingOrder.getTrader();
            this.removeOrder(matchingOrder);
            if(matchingOrder instanceof MarketBuyOrder) {
              this.addOrder(new MarketBuyOrder(matchingOrder.getStockSymbol(),
newMaker.getDefaultOrderSize(),
                                               (matchingOrder.getPrice() -
newMaker.getPriceOffset()), newMaker));
            }
            if(matchingOrder instanceof MarketSellOrder) {
              this.addOrder(new MarketSellOrder(matchingOrder.getStockSymbol(),
newMaker.getDefaultOrderSize(),
                                                (matchingOrder.getPrice() +
newMaker.getPriceOffset()), newMaker));
            }
          }
          else {
            this.removeOrder(matchingOrder);
          }
        }
        else if(matchingOrder.getNumberShares() > newTransaction.getNumberShares())
{
          matchingOrder.setNumberShares(matchingOrder.getNumberShares() -
newTransaction.getNumberShares());
        }


        if(order.getNumberShares() > matchingOrder.getNumberShares()) {
          order.setNumberShares(order.getNumberShares() -
matchingOrder.getNumberShares());
          if(order instanceof BuyOrder) {
            LLNode<Order> tempSell = this.firstSellNode;
            int tempMatch = 0;
            while(tempSell != null) {
              if(this.matchingOrders(order, tempSell.getElement()) == true) {
                matchingOrder = tempSell.getElement();
                tempMatch = 1;
              }
              tempSell = tempSell.getNext();
            }
            if(tempMatch == 1) {
              continue;
            }
            else {
              return listHead;
            }
          }
          else if(order instanceof SellOrder) {
            LLNode<Order> tempBuy = this.firstBuyNode;
            int tempMatch = 0;
            while(tempBuy != null) {
              if(this.matchingOrders(tempBuy.getElement(), order) == true) {
```

```
              matchingOrder = tempBuy.getElement();
              tempMatch = 1;
            }
            tempBuy = tempBuy.getNext();
          }
          if(tempMatch == 1) {
            continue;
          }
          else {
            return listHead;
          }
        }
      }
      else if(order.getNumberShares() <= matchingOrder.getNumberShares()) {
        return listHead;
      }
    }
  }
}

/**
 * Close the market, all market maker orders are expired
 */
public void closeMarket() {
  /**loop buy list, sell list, remove het */
  LLNode<Order> b = this.firstBuyNode;
  while(b != null) {
    if(b.getElement() instanceof MarketBuyOrder) {
      this.removeOrder(b.getElement());
    }
    else if (b.getElement().isDayOrder() == true) {
      this.removeOrder(b.getElement());
    }
    b = b.getNext();
  }
  LLNode<Order> s = this.firstSellNode;
  while(s != null) {
    if(s.getElement() instanceof MarketSellOrder) {
      this.removeOrder(s.getElement());
    }
    else if (s.getElement().isDayOrder() == true) {
      this.removeOrder(s.getElement());
    }
    s = s.getNext();
  }
}

/**
 * toString method that format the output string
 * @return output string
 */
public String toString() {
  return "The market symbol: " + this.getStockSymbol();
}

/**
 * equals method to check if there's identical markets
 * @return true  when there's identical markets
 * @return false  when there's no identical markets
```

```
   */
  public boolean equals(Object m) {
    if (m instanceof Market) {
      Market newMarket = (Market)m;
      if (this.getStockSymbol() == newMarket.getStockSymbol()
            && this.getBuyOrders() == newMarket.getBuyOrders()
            && this.getSellOrders() == newMarket.getSellOrders()){
        return true;
      }
    }
    return false;
  }
}
```