File - C:\Users\kaden\Desktop\Assign 3\src\Main.java

```java
import java.sql.SQLException;
import java.text.ParseException;
import java.util.Scanner;

public class Main {

    /**
     * The main class is used to dictate the direction of the program, first by asking if the database should be reset
     * and cleared and then by entering the main menu where the user can select between different tasks.
     * @param args Required by java as part of defining the main class
     * @throws SQLException At lower levels of the program there are SQL Exceptions thrown
     * @throws ParseException At lower levels of this program there are ParseExceptions thrown (when dealing with dates)
     */
    public static void main(String[] args) throws SQLException, ParseException {
        boolean blnEndProgram = false; //determines based on user input if the program should stop running
        int intProgramDictator; //user input for dictating menus

        //reference to the class that is responsible for resetting the database
        DataReset resetInventory = new DataReset();
        ImportFile importing = new ImportFile();

        Scanner reader = new Scanner(System.in);

        //two prompts and responses regarding wheather data should be reset or persisted
        System.out.println("Would you like to reset the Inventory to original values? (y/n)");
        if (reader.nextLine().toLowerCase().equals("y")){
            resetInventory.defaultInventory();
        }
        System.out.println("Would you like to empty the Customer and Order tables? (y/n)");
        if(reader.nextLine().toLowerCase().equals("y")){
            resetInventory.emptyCustomerAndOrders();
        }

        System.out.println("Would you like to import an initial file? (y/n)");
        if(reader.nextLine().toLowerCase().equals("y")){
            importing.fileParsing();
        }

        //main menu loop that loops until user specifies it to stop.
        while (!blnEndProgram){
            System.out.println("\n" +
                    "\n\n----------------Basics----------------\n0 - For Exiting The Program \n1 - For File Importing " +
                    "\n\n---------Database Management----------\n2 - For Creating A New Customer \n3 - For Creating A New Order\n4 - For Modifying An Order" +
                    "\n\n--------------Reporting---------------\n5 - For Customer Reporting\n6 - For Delivery Reporting\n7 - For Inventory Reporting");
            System.out.print("Response: ");

            intProgramDictator = reader.nextInt();

            switch (intProgramDictator){ //case switch for accessing different menus
                case 0:
                    blnEndProgram = true;
                    break;
                case 1:
                    importing.fileParsing();
                    break;
                case 2:
                    customerToAddDetails(); //used to gather information from user
                    break;
                case 3:
                    orderToAddDetails(); //used to gather information from user
                    break;
                case 4:
                    orderToModifyDetails(); //used to gather information from user
                    break;
                case 5: //provides reporting for Customer orders
```

```
66                      CustomerOrderReport customerReport = new CustomerOrderReport();
67                      customerReport.customerReportingMenu();
68                      break;
69                  case 6: //provides reporting for when deliveries are scheduled
70                      DeliveryReporting deliveryReport = new DeliveryReporting();
71                      deliveryReport.deliveryReport();
72                      break;
73                  case 7: //provides reporting for remaining inventory
74                      InventoryReporting inventoryReport = new InventoryReporting();
75                      inventoryReport.inventoryReport();
76                      break;
77              }
78          }
79      }
80
81      /**
82       * This method is used to collect information regarding adding a customer to
    database, it only collects information
83       * the processing of the information is done in the AddCustomer class
84       * @throws SQLException Thrown when talking to the database to get the newest TUID
85       */
86      public static void customerToAddDetails() throws SQLException {
87          Scanner reader = new Scanner(System.in);
88
89          //reference to class that manages database access
90          DatabaseManager databaseManager = new DatabaseManager();
91
92          //reference to add a new customer in AddCustomer class
93          AddCustomer newCustomer;
94
95          //information needed to create a new customer
96          int intTUID = databaseManager.getNewCustTUID(); //auto filled from the database
    based on newest
97          String strFirstName;
98          String strLastName;
99          String strPhoneNum;
100
101         //prompts to fill out data
102         System.out.println("Please enter customer information of the customer you would
    like to add");
103         System.out.println("Order ID: " + databaseManager.getNewCustTUID() + " (Auto
    Filled)");
104
105         System.out.print("Customer First Name: ");
106         strFirstName = reader.nextLine();
107
108         System.out.print("Customer Last Name: ");
109         strLastName = reader.nextLine();
110
111         System.out.print("Customer Phone Number: ");
112         strPhoneNum = reader.nextLine();
113
114         //add the new customer
115         newCustomer = new AddCustomer(intTUID, strFirstName, strLastName, strPhoneNum);
116         newCustomer.customerAdder();
117
118     }
119
120     /**
121      * This method is used to collect information regarding adding a Order to database,
    it only collects information
122      * the processing of the information is done in the AddOrder class
123      * @throws SQLException Thrown when talking to the database to get the newest TUID
124      * @throws ParseException Thrown at a lower level when parsing dates
125      */
126     public static void orderToAddDetails() throws SQLException, ParseException {
127         Scanner reader = new Scanner(System.in);
128
129         DatabaseManager databaseManager = new DatabaseManager();
130
131         //instance of the class that deals with orders
132         AddOrModifyOrder orderToAdd;
133
134         //information needed to create a new order
135         int intTUID = databaseManager.getNewOrderTUID();
```

```
136         int intCustomerTUID;
137         int intItemTUID;
138         int intQuantity;
139 
140         //prompts to fill out data
141         System.out.println("Please enter customer information of the customer you would
    like to add");
142         System.out.println("Order TUID: " + intTUID + " (Auto Filled)");
143 
144         System.out.print("Enter the TUID of the customer that is ordering: ");
145         intCustomerTUID = reader.nextInt();
146 
147         System.out.print("Enter the the item TUID that the customer is ordering: ");
148         intItemTUID = reader.nextInt();
149 
150         System.out.print("Enter the quantity of item (" + intItemTUID + ") The customer
    is ordering: ");
151         intQuantity = reader.nextInt();
152 
153         //create the new order
154         orderToAdd = new AddOrModifyOrder(intTUID, intCustomerTUID, intItemTUID,
    intQuantity);
155         orderToAdd.orderAdder();
156 
157     }
158 
159     /**
160      * This method is used to collect information regarding modifying a Order in the
    database, it only collects information
161      * the processing of the information is done in the AddOrder class
162      * @throws SQLException Thrown when talking to the database to get the newest TUID
163      * @throws ParseException Thrown at a lower level when parsing dates
164      */
165     public static void orderToModifyDetails() throws SQLException, ParseException {
166         Scanner reader = new Scanner(System.in);
167 
168         DatabaseManager databaseManager = new DatabaseManager();
169 
170         //information needed to modify an order
171         int intTUID;
172         int intItemTUID;
173         int intQuantityChange;
174 
175         //temp order is used to access the customer ID that is associated with the order
     specified by the user
176         Order tempOrder;
177 
178         //instance of the class that deals with orders
179         AddOrModifyOrder orderToModify;
180 
181         System.out.println("Please enter Order Information of order you would like to
    modify");
182         System.out.print("Order TUID: ");
183         intTUID = reader.nextInt();
184 
185         //gets the order specified from the database
186         tempOrder = databaseManager.getOrder(intTUID);
187 
188         System.out.print("What Item from this order would you like to add/remove/or
    modify? (ItemTUID): ");
189         intItemTUID = reader.nextInt();
190 
191         System.out.print("Enter either +/- value of (" + intItemTUID + ") The customer
    is changing: ");
192         intQuantityChange = reader.nextInt();
193 
194         //modify the order
195         orderToModify = new AddOrModifyOrder(intTUID, tempOrder.getCustomer_TUID(),
    intItemTUID, intQuantityChange);
196         orderToModify.orderUpdater();
197     }
198 }
```

File - C:\Users\kaden\Desktop\Assign 3\src\DataReset.java

```java
import java.sql.SQLException;

/**
 * Class that is used to reset data in the database by either clearing tables or resetting the inventory
 */
public class DataReset {

    DatabaseManager databaseManager = new DatabaseManager();

    /**
     * This method resets the inventory items to their original starting values
     * @throws SQLException Thrown if there is an issue adding inventory items
     */
    public void defaultInventory() throws SQLException {

        //first clear out the inventory
        databaseManager.clearInventory();

        //create an array holding all inventory items
        Inventory[] initialInventory = new Inventory[12];

        //fill array
        initialInventory[0] = new Inventory(101, "Peanut Butter - Chunky", 100, 10.00);
        initialInventory[1] = new Inventory(102, "Peanut Butter - Smooth", 100, 11.00);
        initialInventory[2] = new Inventory(201, "Jelly - Strawberry", 100, 9.50);
        initialInventory[3] = new Inventory(202, "Jelly - Raspberry", 100, 9.50);
        initialInventory[4] = new Inventory(203, "Jelly - Peach", 100, 11.25);
        initialInventory[5] = new Inventory(301, "Bread - White", 40, 17.00);
        initialInventory[6] = new Inventory(302, "Bread - Wheat", 30, 19.00);
        initialInventory[7] = new Inventory(303, "Bread - Sourdough", 40, 21.50);
        initialInventory[8] = new Inventory(401, "Milk - White", 10, 17.00);
        initialInventory[9] = new Inventory(402, "Milk - Chocolate", 15, 17.00);
        initialInventory[10] = new Inventory(403, "Milk - White 2%", 20, 18.00);
        initialInventory[11] = new Inventory(404, "Milk - Chocolate 2%", 20, 18.00);

        //persist the new inventory to the database
        for (int i = 0; i < 12; i++){
            databaseManager.addInventory(initialInventory[i]);
        }
    }

    /**
     * This method clears the order tables and the customer table
     * @throws SQLException
     */
    public void emptyCustomerAndOrders() throws SQLException {

        //all done through the database manager class
        databaseManager.clearCustomer();
        databaseManager.clearOrder();
        databaseManager.clearITO();
    }
}

```

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.sql.SQLException;
import java.text.ParseException;
import java.util.Scanner;

public class ImportFile {
    public void fileParsing() throws SQLException, ParseException {
        Scanner reader = new Scanner(System.in);

        //used for reading from the file, set in a try catch so initialized to null
        Scanner fileReader = null;


        String strFilePath; //path dictated by the user for the file
        String strParsedLine; //Holds a protocol line from the file to be parsed
        File file;
        char chrImportType; //checks the begging of each line in the file to determine what the line is trying to do

        //prompt user
        System.out.println("Please enter the name of the file you would like to import (include the full path)");
        System.out.print("File Path and Name: ");

        strFilePath = reader.nextLine();
        if(validFile(strFilePath)){ //check if its a valid import file or not
            try {
                file = new File(strFilePath);
                fileReader = new Scanner(file);
            } catch (Exception ignored){

            }
        } else //if not exit the class and return to main menu
            return;


        //read the entire file.
        while (fileReader.hasNext()){
            strParsedLine = fileReader.nextLine();
            chrImportType = strParsedLine.charAt(0);

            //determine what kind of protocol is being requested
            switch (chrImportType) {
                case 'C':
                    addCustomer(strParsedLine);
                    break;
                case 'O':
                case 'M':
                    addOrModifyOrder(strParsedLine);
                    break;
            }
        }
    }

    /**
     * Sends new customer information to the AddCustomer class
     * @param strImportDetails the information that was obtained from the file
     * @throws SQLException Thrown when attempting to add new customer to database
     */
    public void addCustomer(String strImportDetails) throws SQLException {
        //information needed for creating a new customer
        int intTUID;
        String strFirstName;
        String strLastName;
        String strPhoneNum;

        //split the line by \t (the way it was created) and add that information the the variables
        String[] strAddCustomerDetails = strImportDetails.split("\t");

        //fill in information
        intTUID = Integer.parseInt(strAddCustomerDetails[1]); //start at index 1 because index 0 is the letter dictating protocol
        strFirstName = strAddCustomerDetails[2];
```

```
72          strLastName = strAddCustomerDetails[3];
73          strPhoneNum = strAddCustomerDetails[4];
74
75          //create new customer
76          AddCustomer addCustomer = new AddCustomer(intTUID, strFirstName, strLastName,
    strPhoneNum);
77          addCustomer.customerAdder();
78
79      }
80
81      /**
82       * Sends new order or modify order information to the AddOrder class
83       * @param strImportDetails The information that was obtained from the file
84       * @throws SQLException Thrown when adding order to database
85       * @throws ParseException Thrown when parsing dates for scheduling
86       */
87      public void addOrModifyOrder(String strImportDetails) throws SQLException,
    ParseException {
88          //information needed for crating a new customer
89          int intTUID;
90          int intCustomerTUID;
91          int intItemTUID;
92          int intQuantity;
93
94          String strAddOrModify;
95
96          //split the line by \t (the way it was created) and add that information the the
     variables
97          String[] strOrderDetails = strImportDetails.split("\t");
98
99          strAddOrModify = strOrderDetails[0];
100
101         //fill in information
102         intTUID = Integer.parseInt(strOrderDetails[2]); //ordering is backwards from how
     it is defined in the database that is why start with 2
103         intCustomerTUID = Integer.parseInt(strOrderDetails[1]);
104         intItemTUID = Integer.parseInt(strOrderDetails[3]);
105         intQuantity = Integer.parseInt(strOrderDetails[4]); //this will be the amount
    the customer wants if its a new order
106         //or it will be the amount the customer wants to change by if its an modify
    order
107
108         //create new order
109         AddOrModifyOrder orderToAdd = new AddOrModifyOrder(intTUID, intCustomerTUID,
    intItemTUID, intQuantity);
110         if (strAddOrModify.equals("O")){
111             orderToAdd.orderAdder();
112         } else
113             orderToAdd.orderUpdater();
114
115
116     }
117
118     /**
119      * Checks to see if the file the user is attempting to reach is valid or not based
    on different conditions
120      * @param strFilePath Path of the file the user entered
121      * @return true or false regarding the validity of the file
122      */
123     public boolean validFile(String strFilePath) {
124         //file reader and file objects for reading the file.
125         Scanner fileReader;
126         File file;
127
128         try { //this will try to create the file object, if it fails then the file doesn
    't exist
129             file = new File(strFilePath);
130             fileReader = new Scanner(file);
131         } catch (FileNotFoundException e){
132             System.out.println("hmmm, that file doesn't seem to exist, check your path
    and try again");
133             return false;
134         }
135
136         //this will check if the file is a text file.
```

```
137         if(!strFilePath.substring(strFilePath.length() - 4).contains(".txt")){
138             System.out.println("Uh, that's not a text file? Please input a file ending
    in (.txt)");
139             return false;
140         }
141 
142 
143         try { //finally if the file is a text file then check to see if its empty by
    trying to read it. if its not empty then make sure the file starts with one of 3
    protocol letters
144             char firstChar = fileReader.nextLine().charAt(0);
145             if (firstChar != 'O' && firstChar != 'M' && firstChar != 'C') {
146                 System.out.println("It would seem this file doesn't meet the format
    requirements, try a different file");
147                 return false;
148             }
149         } catch (Exception e){
150             System.out.println("looks like that file might be empty... try a different
    file");
151             return false;
152         }
153 
154         //if it makes it through everything it is a valid file
155         return true;
156     }
157 
158 }
159 
```

File - C:\Users\kaden\Desktop\Assign 3\src\AddCustomer.java

```java
import java.sql.SQLException;

/**
 * This class is used for adding a customer to the database based on information received from the user/file
 * Note this does not have any error checking so if a customer is manually added it could break
 */
public class AddCustomer {
    private final int intTUID;
    private final String strFirstName;
    private final String strLastName;
    private final String strPhoneNumber;

    DatabaseManager databaseManager = new DatabaseManager();

    /**
     * Constructor for Customer adder class
     * @param intTUID ID for customer
     * @param strFirstName First name
     * @param strLastName Last name
     * @param strPhoneNumber Phone number
     */
    public AddCustomer(int intTUID, String strFirstName, String strLastName, String strPhoneNumber) {
        this.intTUID = intTUID;
        this.strFirstName = strFirstName;
        this.strLastName = strLastName;
        this.strPhoneNumber = strPhoneNumber;
    }

    /**
     * Creates a new Customer object and adds it to the database
     * @throws SQLException Thrown when adding new customer to database
     */
    public void customerAdder() throws SQLException {

        //create new customer with data retrieved
        Customer customerToAdd = new Customer(intTUID, strFirstName, strLastName, strPhoneNumber);

        //add to database
        databaseManager.addCustomer(customerToAdd);
    }
}

```

```java
import java.sql.SQLException;
import java.text.ParseException;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.util.ArrayList;

/**
 * Class that is used for adding orders or modifying orders in the database
 * before contacting the scheduler it will see if there is a date opening, if so it will take it
 */
public class AddOrModifyOrder {

    private final int intTUID;
    private final int intCustomerTUID;
    private final int intItemTUID;
    private int intQuantity;
    DatabaseManager databaseManager = new DatabaseManager();

    /**
     * Constructor for Order adder/updater class
     * @param intTUID Order TUID
     * @param intCustomerTUID Customer TUID
     * @param intItemTUID Item TUID
     * @param intQuantity Quantity to modify or add
     */
    public AddOrModifyOrder(int intTUID, int intCustomerTUID, int intItemTUID, int intQuantity) {
        this.intTUID = intTUID;
        this.intCustomerTUID = intCustomerTUID;
        this.intItemTUID = intItemTUID;
        this.intQuantity = intQuantity;
    }

    /**
     * method that is used for creating a new order
     * @throws SQLException Thrown when adding order to database
     * @throws ParseException Thrown when parsing dates
     */
    public void orderAdder() throws SQLException, ParseException { //we will need to check if an order already exsists to see if we are just adding to it or making a new order

        //creates a to the scheduler class that is used for getting the next delivery date
        OrderScheduler scheduler = new OrderScheduler();
        Order newOrder;

        int intQuantityLeft; // hold the remaining quantity of an item that is being ordered
        String strItemName = databaseManager.getItemName(intItemTUID);
        String strDateTimeDeliveryPerson; //string that is returned from scheduler class that needs to be split
        String[] strScheduleData; //split string containing date of delivery and who will deliver it
        String strDateOfNewOrder;
        int intDeliveryPersonOfNewOrderTUID;

        //find out if there is enough remaining inventory left to complete this order
        intQuantityLeft = databaseManager.getItemQuantityRemaining(intItemTUID);


        //update the inventory to reflect how much was taken
        //will allocate inventory based on what is left, if nothing is left it will return false an crete no order
        if (!updateInventory(intQuantityLeft, strItemName)){
            return;
        }

        //Checks to see if there was a time slot opening, if there was use it and don't continue to scheduler
        if(checkTimeSlotOpening()){
            return;
        }
```

```
66
67          //checks to see if this belongs to a new order or if it already existed
68          if(databaseManager.orderExists(intTUID)){ //if the order exists then we are just
     changing the total
69              Order tempOrder = databaseManager.getOrder(intTUID);
70              tempOrder.setOrder_Total(getTotal());
71              databaseManager.updateOrder(tempOrder);
72          } else{ //if the order does not exist then we need to schedule it, find the
    total, and determine who will be delivering it
73
74              //get the delivery schedule and person from the scheduler class and parse it
75              //send the most recent order before this order so a time can be scheduled
    after it, do that here to avoid another
76              //reference to the database manager
77              strDateTimeDeliveryPerson = scheduler.newOrder(databaseManager.
    getMostRecentOrder());
78              strScheduleData = strDateTimeDeliveryPerson.split("\t");
79
80              //assign variables to parsed data
81              strDateOfNewOrder = strScheduleData[0];
82              intDeliveryPersonOfNewOrderTUID = Integer.parseInt(strScheduleData[1]);
83
84              //create the new order
85              newOrder = new Order(intTUID, intCustomerTUID, strDateOfNewOrder,
    intDeliveryPersonOfNewOrderTUID, getTotal());
86              databaseManager.addOrder(newOrder);
87          }
88
89
90      }
91
92      /**
93       * method used for updating existing orders from the user or from a file
94       * @throws SQLException Thrown when updating the database order
95       */
96      public void orderUpdater() throws SQLException {
97
98          //gets the name of the item that is being updated
99          String strItemName = databaseManager.getItemName(intItemTUID);
100         int intQuantityLeft;
101
102         //find out if there is enough remaining inventory left to complete this order
103         intQuantityLeft = databaseManager.getItemQuantityRemaining(intItemTUID);
104
105         //update the inventory to reflect how much was taken or added back
106         //will allocate inventory based on what is left, if nothing is left it will
    return false an crete no order
107         if (!updateInventory(intQuantityLeft, strItemName)){
108             return;
109         }
110
111         //update order
112         Order tempOrder = databaseManager.getOrder(intTUID);
113         tempOrder.setOrder_Total(getTotal());
114
115         if (tempOrder.getOrder_Total() == 0){ //everything has been removed, create a
    place holder order
116             tempOrder.setCustomer_TUID(0); //this will indicate it is just a place
    holder order
117         }
118
119         //update the order
120         databaseManager.updateOrder(tempOrder);
121     }
122
123     /**
124      * gets the total for a specific order from a customer
125      * @return returns the total dollar amount of the order
126      * @throws SQLException Thrown when retrieving a table from the database
127      */
128     public double getTotal() throws SQLException {
129         //array list that will hold all the inventory to orders to search through
130         ArrayList<InventoryToOrder> inventoryToOrders;
131
132         //vars for total, prices, and quantity ordered of each item
```

```
133         double dblRunningTotal = 0;
134         double dblItemPrice;
135         double dblQuantity;
136 
137         //fill the array list
138         inventoryToOrders = databaseManager.getInventoryToOrders();
139 
140         //iterate over all the inventory to orders and see if their order TUID matches
    the current orders
141         for (InventoryToOrder ITO: inventoryToOrders) {
142             if (ITO.getOrder_TUID() == intTUID){ // if it does then increase the total
143                 dblItemPrice = databaseManager.getItemPrice(ITO.getInventory_TUID());
144                 dblQuantity = ITO.getQuantity();
145                 dblRunningTotal += dblItemPrice * dblQuantity;
146             }
147         }
148 
149         //return the total
150         return dblRunningTotal;
151     }
152 
153     /**
154      * Checks to see if there is a open time slot before going to the scheduler
155      * @return boolean if there was an open time slot or not
156      * @throws SQLException Thrown when getting orders from database
157      * @throws ParseException Thrown when parsing dates
158      */
159     public boolean checkTimeSlotOpening() throws SQLException, ParseException {
160         //array list to hold all order
161         ArrayList<Order> orders;
162         String[] strDateAndTime; //string array for holding the Date and the time
163 
164         //parsed values of strDateAndTime array
165         LocalDate dateOfOpenSlot;
166         LocalTime timeOfOpenSlot;
167 
168         //create dateAndTimeOfSlot from the localdate + the local time
169         LocalDateTime dateAndTimeOfSlot;
170         LocalDateTime currentDateAndTime = LocalDateTime.now();
171 
172         //get all the orders and iterate over them to find opening
173         orders = databaseManager.getOrders();
174         for (Order o: orders) {
175             if (o.getCustomer_TUID() == 0){ //earlier we set the customerTUID to 0 in
    the Order table if it was remove, find an order like that and takes it delivery time
176                 strDateAndTime = o.getDelivery_Date_Time().split(" ");
177                 dateOfOpenSlot = LocalDate.parse(strDateAndTime[0]);
178                 timeOfOpenSlot = LocalTime.parse(strDateAndTime[1]);
179                 dateAndTimeOfSlot = dateOfOpenSlot.atTime(timeOfOpenSlot);
180                 if(currentDateAndTime.compareTo(dateAndTimeOfSlot) <= 0){ //makes sure
    the current date isn't already passed this open slot time
181                     databaseManager.deleteOrder(o); //if everything is good, git rid of
    the place holder order and assign its date to the new order
182                     o.setTUID(intTUID);
183                     o.setCustomer_TUID(intCustomerTUID);
184                     o.setOrder_Total(getTotal());
185                     databaseManager.addOrder(o);
186                     return true;
187                 }
188             }
189         }
190         return false;
191     }
192 
193     /**
194      * updates the inventory table and the InventoryToOrder table based on remaining
    inventory
195      * @param intQuantityLeft How much of an item is left
196      * @param strItemName The name of the item for readability
197      * @return boolean indicating if the inventory was updated or not (no inventory left
    )
198      * @throws SQLException Thrown when getting inventory from the database
199      */
200     public boolean updateInventory(int intQuantityLeft, String strItemName) throws
    SQLException {
```

```
201         //references to inventory and inventory to order
202         Inventory inventory;
203         InventoryToOrder inventoryToOrder;
204 
205 
206         if (intQuantityLeft == 0 && intQuantity > 0){ //if the quantity left is 0 then
    just don't place an order and end there
207             System.out.println("\nThere is no (" + strItemName + ") left, No Order has
    been placed\n");
208             return false;
209         } else if (intQuantityLeft < intQuantity){ // if the quantity left is less than
    what was requested, give them the max amount
210             intQuantity = intQuantityLeft;
211             System.out.println("\nThere was not enough (" + strItemName + ") to fulfill
    your order, the max remaining has been allocated: " + intQuantity + "\n");
212         }
213 
214         //get the inventory for a specific item
215         inventory = databaseManager.getInventoryItem(intItemTUID);
216 
217         //update the inventory
218         if (intQuantityLeft - intQuantity < 0){ //if the amount requested brought it
    below 0 set it to 0
219             inventory.setQuantity(0);
220         }else{ //else set it as the amount remaining after taking it out
221             inventory.setQuantity(intQuantityLeft - intQuantity);
222         }
223         databaseManager.updateInventory(inventory);
224 
225         //checks to see if the Inventory to order already exists, if it does, just
    change the amount
226         if (databaseManager.inventoryToOrderExists(intTUID, intItemTUID)){ //if this
    item is already in the order
227             inventoryToOrder = databaseManager.getInventoryToOrder(intTUID, intItemTUID
    );
228             inventoryToOrder.setQuantity(inventoryToOrder.getQuantity() + intQuantity);
229             databaseManager.updateITO(inventoryToOrder);
230         } else{ //if this item is new to the order
231             inventoryToOrder = new InventoryToOrder(intTUID, intItemTUID, intQuantity);
232             databaseManager.addInventoryToOrder(inventoryToOrder);
233         }
234 
235         //if the order was successful
236         return true;
237     }
238 }
239 
```

```java
import java.sql.SQLException;
import java.text.ParseException;
import java.time.LocalDate;
import java.util.ArrayList;

/**
 * Class is used for talking to the databaseDAO classes, acts as a middle man.\
 * NOTE: BASIC OPERATION COMMENTING IS DONE AT THE DAO CLASS LAYER. NOT HERE
 */
public class DatabaseManager {

    private final CustomerDAO customers;
    private final DeliveryPersonDAO deliveryPersons;
    private final InventoryDAO inventorys;
    private final InventoryToOrderDAO inventoryToOrders;
    private final OrderDAO orders;

    /**
     * Constructor that creates references to all the DAO classes and sends the path of the database to them all
     */
    DatabaseManager(){
        String DB_PATH = "jdbc:sqlite:..\\Assignment 3 DB.db";
        customers = new CustomerDAO(DB_PATH);
        deliveryPersons = new DeliveryPersonDAO(DB_PATH);
        inventorys = new InventoryDAO(DB_PATH);
        inventoryToOrders = new InventoryToOrderDAO(DB_PATH);
        orders = new OrderDAO(DB_PATH);
    }

    //talking to customers table -------------------------

    public ArrayList<Customer> getCustomers() throws SQLException {
        return customers.getAll();
    }

    public Customer getCustomer(int intCustTUID) throws SQLException {
        return customers.getCustomer(intCustTUID);
    }

    public void addCustomer(Customer custToAdd) throws SQLException {
        customers.insert(custToAdd);
    }

    public void clearCustomer() throws SQLException {
        customers.deleteAll();
    }

    /**
     * method that gets a new customer TUID in ascending order
     * @return integer containing new customer TUID
     * @throws SQLException if there was an issue getting all customers
     */
    public int getNewCustTUID() throws SQLException {
        ArrayList<Customer> customers = getCustomers();
        int intLastCustNumber;

        intLastCustNumber = customers.get(customers.size() - 1).getTUID() + 1;
        return intLastCustNumber;
    }

    //talking to delivery person table -------------------------

    /**
     * gets the name of a specific delivery person
     * @param intDeliveryPersonTUID TUID of delivery person
     * @return String containing the name of the delivery person
     * @throws SQLException if there was an issue contacting delivery person table
     */
    public String getDeliveryPerson(int intDeliveryPersonTUID) throws SQLException {
        return deliveryPersons.getDeliveryPerson(intDeliveryPersonTUID).getName();
    }

    /**
     * gets the pay rate for a delivery person
```

```
75       * @param intDeliveryPersonTUID TUID of delviery person
76       * @return double containing delivery persons pay rate
77       * @throws SQLException if there was an issue contacting delivery person talbe
78       */
79      public double getDeliveryPersonPay(int intDeliveryPersonTUID) throws  SQLException {
80          return deliveryPersons.getDeliveryPerson(intDeliveryPersonTUID).getPayRate();
81      }
82
83      //talking to inventory table -------------------------
84
85      public Inventory getInventoryItem(int intInventoryTUID) throws SQLException {
86          return inventorys.getInventory(intInventoryTUID);
87      }
88
89      public String getItemName(int intInventoryTUID) throws SQLException {
90          return inventorys.getInventory(intInventoryTUID).getItem_Name();
91      }
92
93      public double getItemPrice(int intInventoryTUID) throws SQLException {
94          return inventorys.getInventory(intInventoryTUID).getUnit_Price();
95      }
96
97      public void updateInventory(Inventory inventoryToUpdate) throws SQLException {
98          inventorys.update(inventoryToUpdate);
99      }
100
101     public void addInventory(Inventory inventoryToAdd) throws SQLException {
102         inventorys.insert(inventoryToAdd);
103     }
104
105     public void clearInventory() throws SQLException {
106         inventorys.deleteAll();
107     }
108
109     public ArrayList<Inventory> getEntireInventory() throws SQLException {
110         return inventorys.getAll();
111     }
112
113
114
115     //talking to inventory to order table -------------------------
116
117     public InventoryToOrder getInventoryToOrder(int intOrderTUID, int intItemTUID)
    throws SQLException{
118         return inventoryToOrders.getInventoryToOrder(intOrderTUID, intItemTUID);
119     }
120
121     public ArrayList<InventoryToOrder> getInventoryToOrders() throws SQLException {
122         return inventoryToOrders.getAll();
123     }
124
125     public void addInventoryToOrder(InventoryToOrder ITOadd) throws SQLException {
126         inventoryToOrders.insert(ITOadd);
127     }
128
129     public void updateITO(InventoryToOrder IOTToUpdate) throws SQLException {
130         inventoryToOrders.update(IOTToUpdate);
131     }
132
133     public void clearITO() throws SQLException {
134         inventoryToOrders.deleteAll();
135     }
136
137     /**
138      * Checks to see if a given inventory to order exists or not based on the two
    foreign keys
139      * @param intOrderTUID primary key 1
140      * @param intItemTUID primary key 2
141      * @return boolean stating if the Inventory to order exists
142      */
143     public boolean inventoryToOrderExists(int intOrderTUID, int intItemTUID){
144         try{ //if the try fails then it doesn't exist
145             return inventoryToOrders.getInventoryToOrder(intOrderTUID, intItemTUID) !=
    null;
146         } catch (Exception e){
```

```
147             return false;
148         }
149     }
150 
151     //talking to orders table -------------------------
152 
153     public Order getOrder(int intOrderTUID) throws SQLException{
154         return orders.getOrder(intOrderTUID);
155     }
156 
157     public int getItemQuantityRemaining(int intInventoryTUID) throws SQLException {
158         return inventorys.getInventory(intInventoryTUID).getQuantity();
159     }
160 
161     public ArrayList<Order> getOrders() throws SQLException {
162         return orders.getAll();
163     }
164 
165     public void addOrder(Order orderToAdd) throws SQLException, ParseException {
166         orders.insert(orderToAdd);
167     }
168 
169     public void updateOrder(Order orderToUpdate) throws SQLException{
170         orders.update(orderToUpdate);
171     }
172 
173     public void deleteOrder(Order orderToDelete) throws SQLException{
174         orders.delete(orderToDelete);
175     }
176 
177     public void clearOrder() throws SQLException {
178         orders.deleteAll();
179     }
180 
181     /**
182      * Checks to see if a order exists
183      * @param intOrderTUID primary key
184      * @return boolean indicating if it exists or not
185      */
186     public boolean orderExists(int intOrderTUID){
187         try{ // if the try fails it doesn't exist
188             return orders.getOrder(intOrderTUID) != null;
189         } catch (Exception e){
190             return false;
191         }
192 
193     }
194 
195     /**
196      * gets the most recent order entered in the system
197      * @return the most recent order
198      */
199     public Order getMostRecentOrder(){
200         try { //try to return the order, (may fail if it doesn't exist return null)
201             ArrayList<Order> orders = getOrders();
202             return orders.get(orders.size() - 1); // most recent order will be last item
    in list
203         } catch (Exception e) {
204             return null;
205         }
206     }
207 
208     /**
209      * Check to see if a given customer has an order placed or not
210      * @param customerTUID search order table for this ID
211      * @return return true if customer has an order, false if not
212      * @throws SQLException If there is an issue retrieving the orders
213      */
214     public boolean customerHasOrder(int customerTUID) throws SQLException {
215         ArrayList<Order> orders = getOrders();
216         for (Order o: orders) { //iterates over all orders to see if there is a customer
    to order association
217             if (o.getCustomer_TUID() == customerTUID){
218                 return true;
219             }
```

```
220         }
221         return false; // if it didnt find any orders
222     }
223 
224     /**
225      * returns a localDate with the first order date in the database
226      * @return LocalDate of the first order placed
227      * @throws SQLException if there is an issue getting orders
228      */
229     public LocalDate getFirstValidDate() throws SQLException {
230         ArrayList<Order> orders = this.orders.getAll();
231         String[] dateAndTimeOfOrder;
232 
233         //get the date and time of the first entry in the database
234         dateAndTimeOfOrder = orders.get(0).getDelivery_Date_Time().split(" ");
235         //the index 0 contains the date (1 contains time)
236         return LocalDate.parse(dateAndTimeOfOrder[0]);
237 
238     }
239 
240     /**
241      * gets a new order TUID sequentially based on the biggest order ID
242      * @return new order TUID
243      * @throws SQLException if there is an issue getting orders
244      */
245     public int getNewOrderTUID() throws SQLException {
246         ArrayList<Order> orders = getOrders();
247         int intLastOrderNumber; // holds the largest order number
248 
249         //increment the largest order number by 1
250         intLastOrderNumber = orders.get(orders.size() - 1).getTUID() + 1;
251         return intLastOrderNumber;
252     }
253 
254 }
255 
```

File - C:\Users\kaden\Desktop\Assign 3\src\OrderScheduler.java

```java
import java.time.LocalDate;
import java.time.LocalTime;

/**
 * Class that schedules orders when a new order comes in
 */
public class OrderScheduler {

    //holds Localtime var of all possible delivery times
    LocalTime[] strDeliveryTimes = {LocalTime.parse("09:00"), LocalTime.parse("11:00"), LocalTime.parse("14:00"), LocalTime.parse("16:00")};

    /**
     * main method for scheduling new orders
     * @param mostRecentOrder holds the most recent order in the database
     * @return returns a string formatted ("DeliveryDate DeliveryTime\tDeliveryPersonTUID")
     */
    public String newOrder(Order mostRecentOrder) { //needs to tell what time and who will take care of the order
        int deliveryPerson; //holds either a 1 or a 2 depending on delivery person
        boolean blnNewDay = false; //boolean to determine if orders need to be placed on a new day or not



        //Local dates for determining when to schedule
        LocalDate currentDay = java.time.LocalDate.now();
        LocalDate mostRecentDay; //most recent day in the database for a delivery
        LocalTime mostRecentOrderTime;

        try { //try to get the most recent order time and day, this will fail if the DB is empty
            mostRecentOrderTime = getRecentOrderTime(mostRecentOrder);
            mostRecentDay = getRecentOrderDate(mostRecentOrder);
        } catch (Exception e) { //if the db is empty, set the time to 00:00 and the day to today
            mostRecentOrderTime = LocalTime.parse("00:00");
            mostRecentDay = currentDay;
        }

        //if the current day is equal to the most recent day in the database, or ahead, the set the most recent day to today plus 1 (orders always begin on the next day)
        if(currentDay.compareTo(mostRecentDay) >= 0){
            mostRecentDay = currentDay.plusDays(1);
        }

        //if statement checks to most recent order to see if it is end of day
        if (mostRecentOrderTime.compareTo(strDeliveryTimes[3]) == 0){
            //if it is end of day, see if it is yet filled by barb, if not, give it to alan
            if (mostRecentOrder.getDeliveryPerson_TUID() == 2){ // if it is already given to alan, then increase the most recent day and set new day to true
                mostRecentDay = mostRecentDay.plusDays(1);
                blnNewDay = true;
            }
        }



        //get most recent schedule

        //checks to see if it is (not a new day, and not the first entry in the database)
        if (!blnNewDay && mostRecentOrder != null) {

            //everything in here is about determining how to schedule the current day

            //alternate between delivery people
            if(mostRecentOrder.getDeliveryPerson_TUID() == 1){
                deliveryPerson = 2;
            } else {
                deliveryPerson = 1;
            }


```

```
67              // formatted string if the order belongs to alan
68              if (deliveryPerson == 2){
69                  return mostRecentDay + " " + strDeliveryTimes[getIndexOfTime(
    getRecentOrderTime(mostRecentOrder))] + "\t" + deliveryPerson;
70              } else // formatted string if the order belongs to barb, if it does, then
    the delivery time needs to be incremented by 1
71                  return mostRecentDay + " " + strDeliveryTimes[getIndexOfTime(
    getRecentOrderTime(mostRecentOrder)) + 1] + "\t" + deliveryPerson;
72
73
74          } else { // if this is the first entry in the order list, or its the first order
     of a new day
75
76              //formatted string, will assign the time to 9:00 and person to barb, she
    always gets first one
77              return mostRecentDay + " " + strDeliveryTimes[0] + "\t" + "1";
78          }
79      }
80
81      /**
82       * gets the last used delivery time
83       * @param mostRecentOrder Holds the most recently placed order
84       * @return Time of day of most recent order
85       */
86      public LocalTime getRecentOrderTime(Order mostRecentOrder) {
87          String[] deliveryDateAndTime;
88          LocalTime timeOfMostRecentOrder;
89
90          //split the date/time var from the most recent order
91          deliveryDateAndTime = mostRecentOrder.getDelivery_Date_Time().split(" ");
92
93          //parse the second value (the time)
94          timeOfMostRecentOrder =  LocalTime.parse(deliveryDateAndTime[1]);
95
96          //return it
97          return timeOfMostRecentOrder;
98      }
99
100     /**
101      * get the date of the most recent order
102      * @param mostRecentOrder Holds the most recently placed order
103      * @return Date of most recent order
104      */
105     public LocalDate getRecentOrderDate(Order mostRecentOrder) {
106         String[] deliveryDateAndTime;
107         LocalDate dateOfMostRecentOrder;
108
109         //split the date/time var from the most recent order
110         deliveryDateAndTime = mostRecentOrder.getDelivery_Date_Time().split(" ");
111
112         //parse the second value (the date)
113         dateOfMostRecentOrder =  LocalDate.parse(deliveryDateAndTime[0]);
114
115         //return it
116         return dateOfMostRecentOrder;
117     }
118
119     /**
120      * gets the index in the strDeliveryTimes array for printing purposes
121      * @param timeOfMostRecentOrder holds the time of the most recent order
122      * @return the index that matches the time of the most recent order
123      */
124     public int getIndexOfTime(LocalTime timeOfMostRecentOrder){
125         for (int i = 0; i < 4; i++){ //loop through all the times
126             if (timeOfMostRecentOrder.equals(strDeliveryTimes[i])){
127                 return i;
128             }
129         }
130         //return -1 if for some reason this fails
131         return -1;
132     }
133 }
134
```

File - C:\Users\kaden\Desktop\Assign 3\src\CustomerOrderReport.java

```java
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.Scanner;

/**
 * class that is used for customer reporting, (how much each customer bought, totals, order numbers...)
 */
public class CustomerOrderReport {
    DatabaseManager databaseManager = new DatabaseManager();

    /**
     * Main menu for customer reporting allows user to pick between seeing all customer reports or a specific customer report
     * @throws SQLException thrown if there is an issue contacting several tables
     */
    public void customerReportingMenu() throws SQLException {
        Scanner reader = new Scanner(System.in);

        //user choice for menu selection
        int intChoice;

        //prompt user
        System.out.println("Enter a 1 if you would like to see a report of all customers and orders\nEnter a 2 if you would just like to see a specific customer");
        System.out.print("response: ");

        intChoice = reader.nextInt();

        //dictate by user choice
        if(intChoice == 1){
            allCustomerOrders();
        } else if(intChoice == 2){
            specificCustomerOrder();
        }
    }

    /**
     * Generates a report of all customer orders currently in the database
     * @throws SQLException if there is an issue talking to several tables
     */
    public void allCustomerOrders() throws SQLException {

        //holds all customers in the database
        ArrayList<Customer> customers = databaseManager.getCustomers();

        //first iterate over all customers
        for (Customer c: customers) {

            if(databaseManager.customerHasOrder(c.getTUID())){ //make sure the customer has a valid orders
                customerReportFormatter(c);
            }
        }
    }

    /**
     * generates a report of a specific customer in the database
     * @throws SQLException if there is an issue talking to several tables
     */
    public void specificCustomerOrder() throws SQLException {
        Scanner reader = new Scanner(System.in);

        Customer customer;

        //find out what customer the user wants reported on
        System.out.println("Enter the customer ID of the customer you would like a report on");
        System.out.print("Response: ");

        customer = databaseManager.getCustomer(reader.nextInt());


        if(databaseManager.customerHasOrder(customer.getTUID())){ //check to see if specified customer has orders
```

```
70              customerReportFormatter(customer);
71          } else { // if not...
72              System.out.println("Customer does not have any orders");
73          }
74      }
75
76      /**
77       * Formats the report for a customer and iterates over all of their orders
78       * @param c customer that is being reported on
79       * @throws SQLException if there is an issue talking to the database
80       */
81      public void customerReportFormatter(Customer c) throws SQLException {
82
83          //creates arraylists of several different tables needed for generating report
84          ArrayList<Order> orders = databaseManager.getOrders();
85          ArrayList<InventoryToOrder> inventoryToOrders = databaseManager.
    getInventoryToOrders();
86
87          //the grand total each customer spent
88          double dblGrandTotal = 0;
89
90          //create the header
91          System.out.println("\n\n\n");
92          System.out.printf("Customer: %-15s ID: %-5d Phone Number: %-15s %nhas the
    following orders:%n%n", c.getFirst_Name() + " " + c.getLast_Name(), c.getTUID(), c.
    getPhone_Number());
93          for (Order o: orders) { //check over all orders for the orders a specific
    customer has
94              if (o.getCustomer_TUID() == c.getTUID()){ // if the order and the customer
    match up print that order number
95                  dblGrandTotal += o.getOrder_Total(); //increment the grand total
96                  System.out.printf("Order number: %-10d ", o.getTUID());
97                  System.out.println("\n
    ------------------------------------------------------------------------------");
98                  for(InventoryToOrder ito: inventoryToOrders){ //iterate over all
    inventory to orders to print each item the customer has in that order
99                      if(ito.getOrder_TUID() == o.getTUID()){ //print out the item number
    , name, quantity purchased, and cost
100                         System.out.printf("Item Number: %-4d Item Name: %-23s Quantity
    : %-2d Cost: $%7.2f %n", ito.getInventory_TUID(), databaseManager.getItemName(ito.
    getInventory_TUID()), ito.getQuantity(), ito.getQuantity() * databaseManager.
    getItemPrice(ito.getInventory_TUID()));
101                     }
102                 }
103                 System.out.println(
    "------------------------------------------------------------------------------");
104                 System.out.printf("%62s %10s %6.2f%n", "Subtotal:", "$",o.getOrder_Total
    ()); //print out the total for the order being processed
105
106             }
107
108         }
109         //finally print the grand total for that order
110         System.out.println("\n");
111         System.out.printf("%62s %10s %6.2f%n", "Grand Total:", "$", dblGrandTotal);
112     }
113 }
114
```

```java
import java.sql.SQLException;
import java.time.LocalDate;
import java.time.LocalTime;
import java.util.ArrayList;
import java.util.Scanner;

/**
 * class that allows the user to enter a date range and print the orders that were delivered between those dates
 */
public class DeliveryReporting {

    //variables used to hold statistics
    double dblBarbDailyProfit = 0;
    double dblAlanDailyProfit = 0;
    double dblDayExpenditure = 0;
    double dblDayTotal = 0;
    double dblTotalExpenditure = 0;

    /**
     * method that generates the delivery report between dates
     * @throws SQLException if there is an issue talking to the database
     */
    public void deliveryReport() throws SQLException {
        Scanner reader = new Scanner(System.in);
        DatabaseManager databaseManager = new DatabaseManager();

        //start date and end date dictated by the user
        LocalDate startDate;
        LocalDate endDate;

        //trackers to iterate over dates
        LocalDate dateTracker;
        LocalDate dateTracker2;

        //individuals day gains, and total profit for a day
        double dblBarbPayRate = databaseManager.getDeliveryPersonPay(1);
        double dblAlanPayRate = databaseManager.getDeliveryPersonPay(2);

        //total statistic variables
        double dblTotalAlanProfit = 0;
        double dblTotalBarbProfit = 0;
        double dblTotalNetProfit = 0;


        //determines the time of day for a delivery
        LocalTime timeOfDay;

        //holds all orders
        ArrayList<Order> orders = databaseManager.getOrders();

        //string array that holds a date and a time
        String[] dateAndTime;

        //prompt user for dates
        System.out.println("please specify a date range for deliveries");
        System.out.println("please use the format: YYYY-MM-DD");
        System.out.print("date to start searching from (inclusive): ");
        startDate = LocalDate.parse(reader.nextLine());

        System.out.print("date to end the search on (inclusive): ");
        endDate = LocalDate.parse(reader.nextLine());

        //set the date tracket 2 to equal the first valid date
        dateTracker2 = databaseManager.getFirstValidDate();
        System.out.println("\n");

        //if the start date specified bgy the user is lower than the most recent date, print the most recent date header
        if (startDate.compareTo(dateTracker2) <= 0){
            System.out.println(dateTracker2.toString());
        }

        //iterate over all orders in database
        for (Order o: orders) {
```

```
74
75              //set the date tracker to be the first date picked up in the datebase
76              dateAndTime = o.getDelivery_Date_Time().split(" ");
77              dateTracker = LocalDate.parse(dateAndTime[0]); //update it every time
    through with the date that is currently being printed
78
79              //if that date is between the the start date specified, print it out
80              if(dateTracker.compareTo(startDate) >= 0 && dateTracker.compareTo(endDate
    ) <= 0){
81                  if (dateTracker.compareTo(dateTracker2) != 0){ // this if statement will
     be triggered when there is a date change..
82                      dateTracker2 = dateTracker2.plusDays(1); //prints a new date line
    and adds a day to the tracker 2
83
84                      printDailyTotals();
85
86
87
88                      System.out.println("\n" + dateTracker2.toString());
89
90
91                      dblAlanDailyProfit = 0;
92                      dblBarbDailyProfit = 0;
93
94                      dblDayTotal = 0;
95
96
97                      dblDayExpenditure = 0;
98
99                  }
100
101                 //print out the time of day, order number, and who will deliver that
    order
102                 timeOfDay = LocalTime.parse(dateAndTime[1]);
103                 if(o.getOrder_Total() > 0) //make sure it isnt a place holder order
104                     System.out.printf("Time: %5s Order Number: %03d   Customer Name: %-
    15s   Delivered by: %6s   Order Total $%6.2f%n",
105                             timeOfDay.toString(),  //time of day
106                             o.getTUID(),  //order number
107                             databaseManager.getCustomer(o.getCustomer_TUID()).
    getFirst_Name() + " " + databaseManager.getCustomer(o.getCustomer_TUID()).getLast_Name
    (), //first and last name
108                             databaseManager.getDeliveryPerson(o.getDeliveryPerson_TUID
    ()),
109                             o.getOrder_Total()); //Delivery person name
110
111                     if(o.getDeliveryPerson_TUID() == 1){
112                         dblTotalBarbProfit += dblBarbPayRate;
113                         dblBarbDailyProfit += dblBarbPayRate;
114                     } else {
115                         dblTotalAlanProfit += dblAlanPayRate;
116                         dblAlanDailyProfit += dblAlanPayRate;
117                     }
118                     dblDayTotal += o.getOrder_Total();
119                     dblTotalNetProfit += o.getOrder_Total();
120
121
122
123             }
124         }
125         printDailyTotals();
126
127         //print statistics
128         System.out.println("\n\n-----------------------------------------------");
129         System.out.printf("Barb's total profit was           : $%6.2f%n",
    dblTotalBarbProfit);
130         System.out.printf("Alan's total profit was           : $%6.2f%n",
    dblTotalAlanProfit);
131
132         System.out.printf("Resulting in a total expense of   : $%6.2f%n",
    dblTotalExpenditure);
133         System.out.printf("%n%nThe total gain during this time   : $%6.2f%n",
    dblTotalNetProfit);
134
135         System.out.println("\n");
```

```
136
137
138     }
139
140     /**
141      * used for printing statistics after each day
142      */
143     public void printDailyTotals(){
144         dblDayExpenditure = dblAlanDailyProfit + dblBarbDailyProfit;
145         System.out.println("-----------------------------------------------");
146         System.out.printf("Barb's profit on this day      : $%6.2f%n",
    dblBarbDailyProfit);
147         System.out.printf("Alan's profit on this day      : $%6.2f%n",
    dblAlanDailyProfit);
148
149         System.out.printf("Resulting in a daily expense of: $%6.2f%n", dblDayExpenditure
    );
150         System.out.printf("%nToday's total from orders was   : $%6.2f%n", dblDayTotal);
151         dblTotalExpenditure += dblDayExpenditure;
152     }
153 }
154
```

File - C:\Users\kaden\Desktop\Assign 3\src\InventoryReporting.java

```java
import java.sql.SQLException;
import java.util.ArrayList;

/**
 * Class that creates a report of the inventory for the database
 */
public class InventoryReporting {

    /**
     * creates a report of all the inventory in the database
     * @throws SQLException if there is an issue talking to the database
     */
    public void inventoryReport() throws SQLException {
        DatabaseManager databaseManager = new DatabaseManager();

        //holds the inventory data
        ArrayList<Inventory> inventory = databaseManager.getEntireInventory();

        //create header
        System.out.printf("|%-5s|%-25s|%-10s|%-7s|","TUID", "Item Name", "Quantity", "Price");
        System.out.println("\n-----------------------------------------------------");

        //iterate and print everything on each inventory line
        for (Inventory i: inventory) {
            System.out.printf("|%-5d|%-25s|    %-6d|$%7.2f|",i.getTUID(), i.getItem_Name(), i.getQuantity(), i.getUnit_Price());
            System.out.println("\n-----------------------------------------------------");
        }
    }
}

```

File - C:\Users\kaden\Desktop\Assign 3\src\Order.java

```java
/**
 * Holds information for a Order
 */
public class Order {
    int TUID;
    int Customer_TUID;
    String Delivery_Date_Time;
    int DeliveryPerson_TUID;
    double Order_Total;

    public Order(int TUID, int customer_TUID, String delivery_Date_Time, int deliveryPerson_TUID, double order_Total) {
        this.TUID = TUID;
        Customer_TUID = customer_TUID;
        Delivery_Date_Time = delivery_Date_Time;
        DeliveryPerson_TUID = deliveryPerson_TUID;
        Order_Total = order_Total;
    }

    public int getTUID() {
        return TUID;
    }

    public void setTUID(int TUID) {
        this.TUID = TUID;
    }

    public int getCustomer_TUID() {
        return Customer_TUID;
    }

    public void setCustomer_TUID(int customer_TUID) {
        Customer_TUID = customer_TUID;
    }

    public String getDelivery_Date_Time() {
        return Delivery_Date_Time;
    }

    public void setDelivery_Date_Time(String delivery_Date_Time) {
        Delivery_Date_Time = delivery_Date_Time;
    }

    public int getDeliveryPerson_TUID() {
        return DeliveryPerson_TUID;
    }

    public void setDeliveryPerson_TUID(int deliveryPerson_TUID) {
        DeliveryPerson_TUID = deliveryPerson_TUID;
    }

    public double getOrder_Total() {
        return Order_Total;
    }

    public void setOrder_Total(double order_Total) {
        Order_Total = order_Total;
    }
}

```

File - C:\Users\kaden\Desktop\Assign 3\src\OrderDAO.java

```java
import java.sql.*;
import java.util.ArrayList;

public class OrderDAO {
    Connection connection = null;
    String url;

    /**
     * class used to talk to the database with SQL and prepared statements
     */
    public OrderDAO(String url){
        this.url = url;
    }

    public void insert(Order o) throws SQLException {
        connection = DriverManager.getConnection(url);

        PreparedStatement preparedStatement = connection.prepareStatement("Insert into Order_Table(TUID, Customer_TUID, Delivery_Date_Time, DeliveryPerson_TUID, Order_Total) values (?,?,?,?,?)");
        preparedStatement.setInt(1, o.getTUID());
        preparedStatement.setInt(2, o.getCustomer_TUID());
        preparedStatement.setString(3, o.getDelivery_Date_Time());
        preparedStatement.setInt(4, o.getDeliveryPerson_TUID());
        preparedStatement.setDouble(5, o.getOrder_Total());

        preparedStatement.executeUpdate();

        connection.close();
    }

    public void delete(Order o) throws SQLException {
        connection = DriverManager.getConnection(url);

        PreparedStatement preparedStatement = connection.prepareStatement("delete from Order_Table where TUID=?");
        preparedStatement.setInt(1, o.getTUID());

        preparedStatement.executeUpdate();

        connection.close();
    }

    public void deleteAll() throws SQLException {
        connection = DriverManager.getConnection(url);

        PreparedStatement preparedStatement = connection.prepareStatement("delete from Order_Table");

        preparedStatement.executeUpdate();

        connection.close();
    }

    public void update(Order o) throws SQLException {
        connection = DriverManager.getConnection(url);

        PreparedStatement preparedStatement = connection.prepareStatement("update Order_Table set Customer_TUID = ?, Delivery_Date_Time = ?, DeliveryPerson_TUID = ?, Order_Total = ? where TUID = ?");
        preparedStatement.setInt(1, o.getCustomer_TUID());
        preparedStatement.setString(2, o.getDelivery_Date_Time());
        preparedStatement.setInt(3, o.getDeliveryPerson_TUID());
        preparedStatement.setDouble(4, o.getOrder_Total());
        preparedStatement.setInt(5, o.getTUID());

        preparedStatement.executeUpdate();
        connection.close();
    }

    public ArrayList<Order> getAll() throws SQLException{
        connection = DriverManager.getConnection(url);

        PreparedStatement preparedStatement = connection.prepareStatement("select * from Order_Table ORDER BY Delivery_Date_Time");
```

```
69          ResultSet rs = preparedStatement.executeQuery();
70
71          ArrayList<Order> temp = new ArrayList<>();
72          while (rs.next()){
73              temp.add(new Order(rs.getInt(1), rs.getInt(2), rs.getString(3), rs.getInt(4
    ), rs.getDouble(5)));
74          }
75          connection.close();
76          return temp;
77      }
78
79      public Order getOrder(int TUID) throws SQLException {
80          connection = DriverManager.getConnection(url);
81
82          PreparedStatement preparedStatement = connection.prepareStatement("select * from
     Order_Table where TUID = ?");
83          preparedStatement.setInt(1, TUID);
84          ResultSet rs = preparedStatement.executeQuery();
85          Order temp = new Order(rs.getInt(1), rs.getInt(2), rs.getString(3), rs.getInt(4
    ), rs.getDouble(5));
86          connection.close();
87
88          return temp;
89      }
90
91  }
92
```

File - C:\Users\kaden\Desktop\Assign 3\src\InventoryToOrder.java

```java
/**
 * Holds information for the inventory to order table (association table)
 */
public class InventoryToOrder {
    int Order_TUID;
    int Inventory_TUID;
    int Quantity;

    public InventoryToOrder(int order_TUID, int inventory_TUID, int quantity) {
        Order_TUID = order_TUID;
        Inventory_TUID = inventory_TUID;
        Quantity = quantity;
    }

    public int getOrder_TUID() {
        return Order_TUID;
    }

    public void setOrder_TUID(int order_TUID) {
        Order_TUID = order_TUID;
    }

    public int getInventory_TUID() {
        return Inventory_TUID;
    }

    public void setInventory_TUID(int inventory_TUID) {
        Inventory_TUID = inventory_TUID;
    }

    public int getQuantity() {
        return Quantity;
    }

    public void setQuantity(int quantity) {
        Quantity = quantity;
    }
}

```

File - C:\Users\kaden\Desktop\Assign 3\src\InventoryToOrderDAO.java

```java
import java.sql.*;
import java.util.ArrayList;

/**
 * class used to talk to the database with SQL and prepared statements
 */
public class InventoryToOrderDAO {
    Connection connection = null;
    String url;

    public InventoryToOrderDAO(String url){
        this.url = url;
    }

    public void insert(InventoryToOrder i) throws SQLException {
        connection = DriverManager.getConnection(url);

        PreparedStatement preparedStatement = connection.prepareStatement("Insert into Inventory_To_Order_Table(Order_TUID, Inventory_TUID, Quantity) values (?,?,?)");
        preparedStatement.setInt(1, i.getOrder_TUID());
        preparedStatement.setInt(2, i.getInventory_TUID());
        preparedStatement.setInt(3, i.getQuantity());

        preparedStatement.executeUpdate();

        connection.close();
    }

    public void delete(InventoryToOrder i) throws SQLException {
        connection = DriverManager.getConnection(url);

        PreparedStatement preparedStatement = connection.prepareStatement("delete from Inventory_To_Order_Table where Order_TUID=? AND Inventory_TUID=?");
        preparedStatement.setInt(1, i.getOrder_TUID());
        preparedStatement.setInt(2, i.getInventory_TUID());

        preparedStatement.executeUpdate();

        connection.close();
    }

    public void update(InventoryToOrder i) throws SQLException {
        connection = DriverManager.getConnection(url);

        PreparedStatement preparedStatement = connection.prepareStatement("update Inventory_To_Order_Table set Quantity = ? where Order_TUID=? AND Inventory_TUID=?");
        preparedStatement.setInt(1, i.getQuantity());
        preparedStatement.setInt(2, i.getOrder_TUID());
        preparedStatement.setInt(3, i.getInventory_TUID());

        preparedStatement.executeUpdate();
        connection.close();
    }

    public ArrayList<InventoryToOrder> getAll() throws SQLException{
        connection = DriverManager.getConnection(url);

        PreparedStatement preparedStatement = connection.prepareStatement("select * from Inventory_To_Order_Table");
        ResultSet rs = preparedStatement.executeQuery();

        ArrayList<InventoryToOrder> temp = new ArrayList<>();
        while (rs.next()){
            temp.add(new InventoryToOrder(rs.getInt(1), rs.getInt(2), rs.getInt(3)));
        }
        connection.close();
        return temp;
    }

    public void deleteAll() throws SQLException {
        connection = DriverManager.getConnection(url);

        PreparedStatement preparedStatement = connection.prepareStatement("delete from Inventory_To_Order_Table");

```

```
71          preparedStatement.executeUpdate();
72
73          connection.close();
74      }
75
76      public InventoryToOrder getInventoryToOrder(int Order_TUID, int Inventory_TUID)
    throws SQLException {
77          connection = DriverManager.getConnection(url);
78
79          PreparedStatement preparedStatement = connection.prepareStatement("select * from
     Inventory_To_Order_Table where Order_TUID=? AND Inventory_TUID=?");
80          preparedStatement.setInt(1, Order_TUID);
81          preparedStatement.setInt(2, Inventory_TUID);
82
83          ResultSet rs = preparedStatement.executeQuery();
84          InventoryToOrder temp = new InventoryToOrder(rs.getInt(1), rs.getInt(2), rs.
    getInt(3));
85          connection.close();
86
87          return temp;
88      }
89  }
90
```

File - C:\Users\kaden\Desktop\Assign 3\src\Inventory.java

```java
/**
 * Holds information for Inventory table
 */
public class Inventory {
    private int TUID;
    private String Item_Name;
    private int Quantity;
    private double Unit_Price;

    public Inventory(int TUID, String Item_Name, int Quantity, double Unit_Price){
        this.TUID = TUID;
        this.Item_Name = Item_Name;
        this.Quantity = Quantity;
        this.Unit_Price = Unit_Price;
    }

    public int getTUID() {
        return TUID;
    }

    public void setTUID(int TUID) {
        this.TUID = TUID;
    }

    public String getItem_Name() {
        return Item_Name;
    }

    public void setItem_Name(String item_Name) {
        Item_Name = item_Name;
    }

    public int getQuantity() {
        return Quantity;
    }

    public void setQuantity(int quantity) {
        Quantity = quantity;
    }

    public double getUnit_Price() {
        return Unit_Price;
    }

    public void setUnit_Price(double unit_Price) {
        Unit_Price = unit_Price;
    }
}

```

```java
import java.sql.*;
import java.util.ArrayList;

/**
 * class used to talk to the database with SQL and prepared statements
 */
public class InventoryDAO {
    Connection connection = null;
    String url;

    public InventoryDAO(String url){
        this.url = url;
    }

    public void insert(Inventory i) throws SQLException{
        connection = DriverManager.getConnection(url);

        PreparedStatement preparedStatement = connection.prepareStatement("Insert into Inventory_Table(TUID, Item_Name, Quantity, Unit_Price) values (?,?,?,?)");
        preparedStatement.setInt(1, i.getTUID());
        preparedStatement.setString(2, i.getItem_Name());
        preparedStatement.setInt(3, i.getQuantity());
        preparedStatement.setDouble(4, i.getUnit_Price());

        preparedStatement.executeUpdate();

        connection.close();
    }

    public void delete(Inventory i) throws SQLException {
        connection = DriverManager.getConnection(url);

        PreparedStatement preparedStatement = connection.prepareStatement("delete from Inventory_Table where TUID=?");
        preparedStatement.setInt(1, i.getTUID());

        preparedStatement.executeUpdate();

        connection.close();
    }

    public void deleteAll() throws SQLException {
        connection = DriverManager.getConnection(url);

        PreparedStatement preparedStatement = connection.prepareStatement("delete from Inventory_Table");

        preparedStatement.executeUpdate();

        connection.close();
    }

    public void update(Inventory i) throws SQLException {
        connection = DriverManager.getConnection(url);

        PreparedStatement preparedStatement = connection.prepareStatement("update Inventory_Table set Item_Name = ?, Quantity = ?, Unit_Price = ? where TUID = ?");
        preparedStatement.setString(1, i.getItem_Name());
        preparedStatement.setInt(2, i.getQuantity());
        preparedStatement.setDouble(3, i.getUnit_Price());
        preparedStatement.setInt(4, i.getTUID());

        preparedStatement.executeUpdate();
        connection.close();
    }

    public ArrayList<Inventory> getAll() throws SQLException{
        connection = DriverManager.getConnection(url);

        PreparedStatement preparedStatement = connection.prepareStatement("select * from Inventory_Table");
        ResultSet rs = preparedStatement.executeQuery();

        ArrayList<Inventory> temp = new ArrayList<>();
        while (rs.next()){
```

```
71              temp.add(new Inventory(rs.getInt(1), rs.getString(2), rs.getInt(3), rs.
   getDouble(4)));
72          }
73          connection.close();
74          return temp;
75      }
76
77      public Inventory getInventory(int TUID) throws SQLException {
78          connection = DriverManager.getConnection(url);
79
80          PreparedStatement preparedStatement = connection.prepareStatement("select * from
    Inventory_Table where TUID = ?");
81          preparedStatement.setInt(1, TUID);
82          ResultSet rs = preparedStatement.executeQuery();
83          Inventory temp = new Inventory(rs.getInt(1), rs.getString(2), rs.getInt(3), rs.
   getDouble(4));
84          connection.close();
85
86          return temp;
87      }
88  }
89
```

File - C:\Users\kaden\Desktop\Assign 3\src\DeliveryPerson.java

```java
/**
 * Holds information for a Delivery Person
 */
public class DeliveryPerson {
    int TUID;
    String Name;
    double PayRate;

    public DeliveryPerson(int TUID, String name, double payRate) {
        this.TUID = TUID;
        Name = name;
        PayRate = payRate;
    }

    public int getTUID() {
        return TUID;
    }

    public void setTUID(int TUID) {
        this.TUID = TUID;
    }

    public String getName() {
        return Name;
    }

    public void setName(String name) {
        Name = name;
    }

    public double getPayRate() {
        return PayRate;
    }

    public void setPayRate(double payRate) {
        PayRate = payRate;
    }
}

```

File - C:\Users\kaden\Desktop\Assign 3\src\DeliveryPersonDAO.java

```java
import java.sql.*;

/**
 * class used to talk to the database with SQL and prepared statements
 */
public class DeliveryPersonDAO {
    Connection connection = null;
    String url;

    public DeliveryPersonDAO(String url){
        this.url = url;
    }

    public void insert(DeliveryPerson d) throws SQLException {
        connection = DriverManager.getConnection(url);

        PreparedStatement preparedStatement = connection.prepareStatement("Insert into DeliveryPerson_Table(TUID, Name, PayRate) values (?,?,?)");
        preparedStatement.setInt(1, d.getTUID());
        preparedStatement.setString(2, d.getName());
        preparedStatement.setDouble(3, d.getPayRate());

        preparedStatement.executeUpdate();

        connection.close();
    }

    public void delete(DeliveryPerson d) throws SQLException {
        connection = DriverManager.getConnection(url);

        PreparedStatement preparedStatement = connection.prepareStatement("delete from DeliveryPerson_Table where TUID=?");
        preparedStatement.setInt(1, d.getTUID());

        preparedStatement.executeUpdate();

        connection.close();
    }


    public DeliveryPerson getDeliveryPerson(int TUID) throws SQLException {
        connection = DriverManager.getConnection(url);

        PreparedStatement preparedStatement = connection.prepareStatement("select * from DeliveryPerson_Table where TUID = ?");
        preparedStatement.setInt(1, TUID);
        ResultSet rs = preparedStatement.executeQuery();
        DeliveryPerson temp = new DeliveryPerson(rs.getInt(1), rs.getString(2), rs.getDouble(3));
        connection.close();

        return temp;
    }
}

```

File - C:\Users\kaden\Desktop\Assign 3\src\Customer.java

```java
public class Customer {
    private int TUID;
    private String First_Name;
    private String Last_Name;
    private String Phone_Number;

    /**
     * Holds information for a customer
     */
    public Customer(int TUID, String First_Name, String Last_Name, String Phone_Number){
        this.TUID = TUID;
        this.First_Name = First_Name;
        this.Last_Name = Last_Name;
        this.Phone_Number = Phone_Number;
    }

    public int getTUID() {
        return TUID;
    }

    public void setTUID(int TUID) {
        this.TUID = TUID;
    }

    public String getFirst_Name() {
        return First_Name;
    }

    public void setFirst_Name(String first_Name) {
        First_Name = first_Name;
    }

    public String getLast_Name() {
        return Last_Name;
    }

    public void setLast_Name(String last_Name) {
        Last_Name = last_Name;
    }

    public String getPhone_Number() {
        return Phone_Number;
    }

    public void setPhone_Number(String phone_Number) {
        Phone_Number = phone_Number;
    }
}

```

File - C:\Users\kaden\Desktop\Assign 3\src\CustomerDAO.java

```java
import java.sql.*;
import java.util.ArrayList;

/**
 * class used to talk to the database with SQL and prepared statements
 */
public class CustomerDAO {
    Connection connection = null;
    String url;

    public CustomerDAO(String url){
        this.url = url;
    }

    public void insert(Customer c) throws SQLException {
        connection = DriverManager.getConnection(url);

        PreparedStatement preparedStatement = connection.prepareStatement("Insert into Customer_Table(TUID, First_Name, Last_Name, Phone) values (?,?,?,?)");
        preparedStatement.setInt(1, c.getTUID());
        preparedStatement.setString(2, c.getFirst_Name());
        preparedStatement.setString(3, c.getLast_Name());
        preparedStatement.setString(4, c.getPhone_Number());

        preparedStatement.executeUpdate();

        connection.close();
    }

    public void delete(Customer c) throws SQLException {
        connection = DriverManager.getConnection(url);

        PreparedStatement preparedStatement = connection.prepareStatement("delete from Customer_Table where TUID=?");
        preparedStatement.setInt(1, c.getTUID());

        preparedStatement.executeUpdate();

        connection.close();
    }

    public void deleteAll() throws SQLException {
        connection = DriverManager.getConnection(url);

        PreparedStatement preparedStatement = connection.prepareStatement("delete from Customer_Table");

        preparedStatement.executeUpdate();

        connection.close();
    }

    public ArrayList<Customer> getAll() throws SQLException{
        connection = DriverManager.getConnection(url);

        PreparedStatement preparedStatement = connection.prepareStatement("select * from Customer_Table");
        ResultSet rs = preparedStatement.executeQuery();

        ArrayList<Customer> temp = new ArrayList<>();
        while (rs.next()){
            temp.add(new Customer(rs.getInt(1), rs.getString(2), rs.getString(3), rs.getString(4)));
        }
        connection.close();
        return temp;
    }

    public Customer getCustomer(int TUID) throws SQLException {
        connection = DriverManager.getConnection(url);

        PreparedStatement preparedStatement = connection.prepareStatement("select * from Customer_Table where TUID = ?");
        preparedStatement.setInt(1, TUID);
        ResultSet rs = preparedStatement.executeQuery();
```

```
70          Customer temp = new Customer(rs.getInt(1), rs.getString(2), rs.getString(3), rs.
    getString(4));
71          connection.close();
72
73          return temp;
74      }
75  }
76
```