Drake Fafard

CIS 18A

Final Project: Pet Supply Manager

This program will bring up two windows: a Register and a Supply Summary window. The Register window is similar to a cash register where it can purchase a product given the name and the amount to buy. It will access a list of products and check if the given name is in the list. If it is found, it will update the list's info on the amount of that product which is in stock, has been purchased, the amount of money the product has made, and it will update the register's daily earned money. The Supply Summary window allows the user to overview the list of products, along with a shorter list for shelf restock purposes which only displays the name, amount on shelf and total money made on each product. The Supply Summary window can increment the day, which resets the daily variables and adds to the total money made. It also has an update button which updates the table with the newest data stored in the item list. This program can be used to manage supplies on shelfs, overview how effective certain products are, and register items for a pet supply retailer, although with some changes it can assist other retailers.

To describe the process of the code, we begin with the imported packages and the project package:

**package petsupplymanager;**

**import java.awt.\*; // For flow layout in swing**

**import java.awt.event.\*; // Necessary for ActionListener**

**import javax.swing.\*; // Swing GUI**

**import javax.swing.event.\*; // Necessary for ChangeListener**

The first line declares what package this entire project will be stored in. The next lines state that this project will be importing Swing packages, which is necessary for creating an interface to work with.

**public static void main(String args[])**

This part begins the code, and uses the lines in order to create the frame of the code on the event dispatching thread for Swing, allowing for the Graphic User Interface:

**SwingUtilities.invokeLater(new Runnable() {**

**public void run() {**

**new PetSupplyManager();**

**}**

**});**

**}**

Next, we return to the top of the code to observe the interface to hold search methods, which will handle searching for a name in a list of names:

**interface SearchInt {**

**// This will always return a boolean and search for a string**

**public int LinearSearch(String n);**

**}**

This interface called SearchInt only describes what a method named LinearSearch will receive and return. It will receive a String, the name, and return an integer, the index of the array.

The first class in the code is called ProdList and it will store multiple arrays as a way of creating a list of products.

**protected static String name[] = { "Blue Buffalo Wilderness Natural Adult "**

**+ "High Protein Grain Free Chicken Dry Dog Food",**

**"CANIDAE Beef & Oatmeal Dry Dog Food",**

**"Merrick Full Source Raw-Coated Kibble Real Salmon"**

**+ " & Whitefish with Healthy Grains Dry Dog Food",**

**"Instinct Raw Boost Whole Grain Real Chicken & "**

**+ "Brown Rice Recipe Dry Dog Food with Freeze-"**

**+ "Dried Raw Pieces",**

**"Hill's Science Diet Adult Light Large Breed with "**

**+ "Chicken Meal & Barley Dry Dog Food"};**

This segment creates a list of product names, which can be accessed by using an integer.

**protected static float price[] = { 50.78f, 37.49f, 74.99f, 51.99f, 37.99f };**

This segment creates a list of prices, which can be accessed by using the same integer as the name. These lists are parallel, and by taking one number on the list you are referring to a single product.

**// Amount on shelfs**

**protected static int amtOnShelf[] = { 15, 9, 18, 5, 12 };**

**// Number sold in the current day**

**protected static int numSoldDay[] = { 0, 0, 0, 0, 0 };**

**// Total sold over all the days**

**protected static int numSoldTot[] = { 0, 0, 0, 0, 0 };**

These lines create a list of amounts on a shelf, numbers of each product sold in the current day, and numbers of each product sold in total. In order to access these protected arrays outside of the class and the extending classes, we must use accessor methods:

**// Method to return name value at a given index**

**public static String getName(int i) { return name[i]; }**

**// Method to return price value at a given index**

**public static float getPrice(int i) { return price[i]; }**

**// Method to return amount on shelf at a given index**

**public static int getAmtOnShelf(int i) { return amtOnShelf[i]; }**

**// Method to return number of product sold on current day at a given index**

**public static int getNumSoldDay(int i) { return numSoldDay[i]; }**

**// Method to return total number of product sold at a given index**

**public static int getNumSoldTot(int i) { return numSoldTot[i]; }**

These methods take an integer which refers to an index on the lists and returns what is stored at that index. This next method returns the amount of names in the name list by using .length:

**// Method to return list size**

**public static int getListSize() { return name.length; }**

**The next class is called SupplySummary, and it extends ProdList. This gives it access to all of what was inside ProdList, allowing it to modify the list of products. ProdList has three local variables:**

**private int lowAmt; // The low amount to check for with amount on shelf**

**// The total money earned since the list began recording each day**

**private float ttlMoney;**

**private int day; // The current day**

The lowAmt variable stores the low amount to check for on the shelfs, or the amount at which the products need to be restocked. The ttlMoney variable stores the total money earned since the program started. The day variable stores the current day for the program. Next is a constructor for this class which sets the beginning values for lowAmt, ttlMoney, and day.

**SupplySummary(int lAmt, float m, int d) {**

**lowAmt = lAmt;**

**ttlMoney = m;**

**day = d;**

**}**

When an object of this class is created and sent an integer value, a decimal value, and another integer value it will set the beginning values for the variables. Next is the method to switch to the next day.

**// Switch to next day**

**public void SwitchDay() {**

**day++; // Increment day**

**ttlMoney += Register.getDlyMoney(); // Add daily money to total**

**Register.resetDlyMoney(); // Reset the daily money amount**

**// Set all daily values stored in ProdList to 0**

**for (int i = 0; i < ProdList.getListSize(); i++) {**

**numSoldDay[i] = 0;**

**}**

**}**

This will increase the day amount, add onto the total money amount, reset the register’s daily money amount, and then set the number sold on the current day list to 0. For the last part of this class we have the functions which can return the current day and total money to whatever code needs it.

**// Retrieve day**

**public int getDay() { return day; }**

**// Retrieve total money**

**public float getTtlMoney() { return ttlMoney; }**

The next class is Register and it extends ProdList and implements SearchInt. By implementing SearchInt it will describe how the LinearSearch method will work. At the beginning we have the class’ local variable and the constructor which sets the beginning value for the daily money.

**private static float dlyMoney; // Money earned on current day**

**// Constructor for register, will initialize daily money**

**Register(float m) { dlyMoney = m; }**

This is followed by a method to retrieve the money earned that day and a method to set it back to 0:

**// This method will return daily money earned**

**public static float getDlyMoney() { return dlyMoney; }**

**// This method will reset daily money earned**

**public static void resetDlyMoney() { dlyMoney = 0; }**

Next is the method LinearSearch which is implemented from the SearchInt interface.

**@Override**

**public int LinearSearch(String n) {**

**int s = name.length; // Size of list**

**for(int i = 0; i < s; i++) // Check through entire list for the name**

**{**

**if(name[i].equals(n)) // Compare stored item names with given item name**

**return i;**

**}**

**return -1;**

**}**

This will receive a name, retrieve the size of the list of names, and then check through the entire list of names for that given name. If it finds it, it will return the index at which the name was located, otherwise it will return -1. Lastly we have the Purchase function, which will handle purchasing a product and updating the list with it.

**public String Purchase(String n, int c) {**

**int indx = LinearSearch(n);**

**if (indx < 0) // Purchase will fail if name was not found**

**return "Purchase Failed: Invalid Name";**

**// If amount attempting to purchase is above the available amount**

**else if (c > amtOnShelf[indx])**

**return "Purchase Failed: Amount Too High";**

**else { // If nothing fails**

**amtOnShelf[indx] -= c; // Subtract amount purchased from shelf count**

**numSoldDay[indx] += c; // Add amount purchased to total sold per day**

**numSoldTot[indx] += c; // Add amount purchased to total sold**

**dlyMoney += c \* price[indx]; // Add amount earned from purchase**

**return "Purchase Successful"; // return successful purchase message**

**}**

**}**

It will receive a name and an amount, then it will call the LinearSearch method and store the index it returns. If the index was -1, it will say that the name was not found. Otherwise, it will move on and see if the given amount is too big. If it is, it will say that the amount was too high. If nothing is wrong, it will take the given amount and subtract it from the amount of that product on the shelves. It will add the amount of that product purchased to the product’s day sales and total sales, and it will increase the daily money earned.

Returning to the main class PetSupplyManager, which implements ActionListener and ChangeListener in order to work with user inputs, we have the creation of class objects for Register and SupplySummary:

**Register registerObj = new Register(0f); // Object for Register**

**// Object for Supply Summary**

**SupplySummary supplySumObj = new SupplySummary(10, 0f, 1);**

This will allow the code within this class to access an instance of the Register and SupplySummary. It also sends the initial values to these objects so their local variables can be set up. Next are the JFrames created for the supply summary and register:

**JFrame sumFrm; // Declare summary window frame**

**JFrame regFrm; // Declare register window frame**

This creates swing containers for the components that actually display the data. Next are the JTables which will display data off of the product list class.

**// Table to display the refill priority list**

**JTable refPriTbl;**

**// Table to display all the products stored, including how much each made**

**JTable prodListTbl;**

Next are the JPanels which serve as a way of organizing the components in the window frames.

**JPanel regPnlName; // Declare a panel to group together product name**

**JPanel regPnlAmt; // Declare a panel to group together amount purchased**

**JPanel sumPnl; // Declare a panel to group together summary components**

Next are the text fields and sliders, which allow the user to input their own values into the GUI.

**JTextField prodFld; // holds the product to be purchased**

**JSlider amtSldr; // determines the amount to purchase**

**JSlider lowAmtSldr; // determines the amount to purchase**

Next are all the labels, which display single lines of text explaining what the different buttons do or displaying data like the total money earned.

**JLabel purchLab, dlyMoneyLab, dayLab, prodLab, amtLab, lowAmtLab, ttlMoneyLab;**

Next is a constructor for PetSupplyManager which will set up the Swing graphic user interface. It begins with creating the JFrames using the defined variables for them.

**// Create a new JFrame container for the supply summary window**

**sumFrm = new JFrame("Supply Summary Window");**

**// Create a new JFrame container for the register window**

**regFrm = new JFrame("Register Window");**

This creates two windows on the computer named “Supply Summary Window” and “Register Window”. Next we specify the flow layout for the windows so that the components are organized:

**sumFrm.setLayout(new FlowLayout());**

**regFrm.setLayout(new FlowLayout());**

Next the frames are given initial sizes:

**sumFrm.setSize(1100, 600);**

**regFrm.setSize(600, 250);**

Next, the program is told to terminate whenever these windows are closed:

**sumFrm.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);**

**regFrm.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);**

Next, the panels are created using the defined variables:

**regPnlName = new JPanel(); // Create the register panel for product name**

**regPnlAmt = new JPanel(); // Create the register panel for amount**

**sumPnl = new JPanel(); // Create the summary panel**

The text field and sliders are also created with values that describe how large they are:

**// Create the text field for the product name to purchase**

**prodFld = new JTextField(40);**

**// Create a slider for the amount of a product to purchase**

**amtSldr = new JSlider(1, 50);**

**// Create a slider for the low amount to check for.**

**lowAmtSldr = new JSlider(0, 20);**  
Next, tick marks are added to the sliders, and the spacing for these is specified:

**// Add tick marks to the amount slider**

**amtSldr.setPaintTrack(true);**

**amtSldr.setPaintTicks(true);**

**amtSldr.setPaintLabels(true);**

**// Set tick spacing for amount slider**

**amtSldr.setMajorTickSpacing(10);**

**amtSldr.setMinorTickSpacing(1);**

This code is repeated with slightly different values for the slider lowAmtSldr. Next a ChangeListener is added for the sliders so the program can respond to the user changing the amount on them:

**// setChangeListener**

**amtSldr.addChangeListener(this);**

**lowAmtSldr.addChangeListener(this);**

Next, the tables are set up by declaring data for the values and headers

**String refPriData[][] = new String[ProdList.getListSize()][3];**

**String refPriCNames[] = { "Name", "Amount on Shelf" , "Total Earned"};**

**String prodListData[][] = new String[ProdList.getListSize()][6];**

**String prodListCNames[] = { "Name", "Price" , "Amount", "# Sold Today",**

**"# Sold Total", "Total Earned"};**

The data in the ProdList array is imported from the ProdList class by using a loop that will take each individual value on the list.

**for(int i = 0; i < ProdList.getListSize(); i++) {**

**// Store product name**

**prodListData[i][0] = ProdList.getName(i);**

**// Store product price**

**prodListData[i][1] = String.format("%.2f", ProdList.getPrice(i));**

**// Store product amount on shelf**

**prodListData[i][2] = Integer.toString(ProdList.getAmtOnShelf(i));**

**// Store number sold today**

**prodListData[i][3] = Integer.toString(ProdList.getNumSoldDay(i));**

**// Store number sold in total**

**prodListData[i][4] = Integer.toString(ProdList.getNumSoldTot(i));**

**// Store the total money the product has made, convert to string**

**prodListData[i][5] = ("$" + String.format("%.2f",**

**((ProdList.getPrice(i) \* ProdList.getNumSoldTot(i)))));**

**}**

The final line calculates the total money using the price of the item multiplied by the amount sold. Next the data for the refill priority table is initialized

**int r = 0; // To increment rows**

**for(int i = 0; i < ProdList.getListSize(); i++) {**

**if (lowAmtSldr.getValue() >= ProdList.getAmtOnShelf(i)) {**

**// Store product name**

**refPriData[r][0] = ProdList.getName(i);**

**// Store product amount on shelf**

**refPriData[r][1] = Integer.toString(ProdList.getAmtOnShelf(i));**

**// Store the total money the product has made, convert to string**

**refPriData[r][2] = ("$" + String.format("%.2f",**

**((ProdList.getPrice(i) \* ProdList.getNumSoldTot(i)))));**

**r++; // Increment row**

**}**

**}**

In this case two variables are used to keep track of the rows, since it will skip inputting any values that are above the low amount to check for. This will also only import the name, the amount on shelf, and the total money made by the product to the list. This next line takes these arrays of data and uses them to initialize the JTables:

**prodListTbl = new JTable(prodListData, prodListCNames);**

**refPriTbl = new JTable(refPriData, refPriCNames);**

Next the bounds for the tables are set:

**prodListTbl.setBounds(60, 80, 200, 300);**

**refPriTbl.setBounds(60, 80, 200, 300);**

Additionally, the user is allowed to sort the rows by clicking on them by using this:

**prodListTbl.setAutoCreateRowSorter(true);**

**refPriTbl.setAutoCreateRowSorter(true);**

Lastly, the tables are added to scroll panes so that a scrollbar can appear when they get too large

**JScrollPane sp1 = new JScrollPane(refPriTbl);**

**JScrollPane sp2 = new JScrollPane(prodListTbl);**

Next, the buttons are created:

**// Create the button to switch to Register view**

**JButton jbtnReg = new JButton("Register");**

**// Create the button to switch to Supply Summary view**

**JButton jbtnSupSum = new JButton("Supply Summary");**

**// Create the button to purchase a product**

**JButton jbtnPch = new JButton("Purchase");**

**// Create the button to update list**

**JButton jbtnUpd = new JButton("Update");**

**// Create the button to switch to next day**

**JButton jbtnNxtDy = new JButton("Next Day");**

Action listeners for these buttons must be added so that the code can respond to them being clicked:

**// Add action listener for the buttons**

**jbtnReg.addActionListener(this);**

**jbtnSupSum.addActionListener(this);**

**jbtnPch.addActionListener(this);**

**jbtnUpd.addActionListener(this);**

**jbtnNxtDy.addActionListener(this);**

Next, the labels are created:

**purchLab = new JLabel("This will state purchase success/failure");**

**dlyMoneyLab = new JLabel(" Amount earned today: $" +**

**String.format("%.2f", registerObj.getDlyMoney()));**

**dayLab = new JLabel("Day : " + Integer.toString(supplySumObj.getDay()));**

**prodLab = new JLabel("Product Name: ");**

**amtLab = new JLabel("Amount to buy: " + amtSldr.getValue());**

**lowAmtLab = new JLabel("Alert for refill at: " + lowAmtSldr.getValue());**

**ttlMoneyLab = new JLabel("Total $ Earned: " +**

**String.format("%.2f", supplySumObj.getTtlMoney()));**

All these values are converted to string so the label can use them, which is why String.format and Integer.toString is used. Next, all the components are added using the name of the panel and frame followed by .add and the component name within (). Then, the register frame is displayed and the summary frame is hidden initially:

regFrm.setVisible(true);

sumFrm.setVisible(false);

Next we have a stateChanged method overridden from ChangeListener:

@Override

public void stateChanged(ChangeEvent e)

{

amtLab.setText("Amount to buy: " + amtSldr.getValue());

lowAmtLab.setText("Alert for refill at: " + lowAmtSldr.getValue());

}

This allows the sliders to update the values on the labels next to them. Next is the overridden actionPerformed method, which handles button presses. Within it is a switch statement that decides what will happen with each button click:

@Override

public void actionPerformed(ActionEvent ae) {

int r = 0; // Create variable to increment rows

switch(ae.getActionCommand()) {

case "Register" :

// Switch to the Register view

sumFrm.setVisible(false);

regFrm.setVisible(true);

break;

The first button “Register” simply brings the register into view and hides the summary frame. The next case does the opposite:

case "Supply Summary" :

// Switch to the Supply Summary view

sumFrm.setVisible(true);

regFrm.setVisible(false);

break;

The statement following this states what will happen when “Purchase” is pressed:

case "Purchase" :

/\* If "Purchase" is clicked, call the Purchase method using the

product name inputted and the value on the slider, and store

what it returns in the purchase status label \*/

purchLab.setText(registerObj.Purchase(prodFld.getText(),

amtSldr.getValue()));

dlyMoneyLab.setText(" Amount earned today: $" +

String.format("%.2f", registerObj.getDlyMoney()));

break;

This will call the register object and tell it to purchase a product with the value stored in the product field as the name and amount slider as the amount. It will then return a statement on the success of the purchase which is sent to the purchase label at the top of the window. Next, the daily money label is altered based on the new daily money value.

The next case decides what happens when update is pressed on the Summary Window:

**case "Update" :**

**// If "Update" is clicked, first reset all values to 0 in**

**// refill priority table**

**for(r = 0; r < ProdList.getListSize(); r++) {**

**for(int c = 0; c < 3; c++) {**

**refPriTbl.setValueAt("", r, c); // Set blank value**

**}**

**}**

**// Fill in the table with products which are understocked**

**r = 0; // Reset row count**

**for(int i = 0; i < ProdList.getListSize(); i++) {**

**if (lowAmtSldr.getValue() >= ProdList.getAmtOnShelf(i)) {**

**// Store product name**

**refPriTbl.setValueAt(ProdList.getName(i), r, 0);**

**// Store product amount on shelf**

**refPriTbl.setValueAt(Integer.toString(ProdList.getAmtOnShelf(i)), r, 1);**

**// Store the total money the product has made, convert to string**

**refPriTbl.setValueAt("$" + String.format("%.2f",**

**((ProdList.getPrice(i) \* ProdList.getNumSoldTot(i)))), r, 2);**

**r++; // Increment row**

**}**

**}**

**// Also update the second table by checking if the total sold changed**

**for(int i = 0; i < ProdList.getListSize(); i++) {**

**if(prodListTbl.getValueAt(i, 4) != Integer.toString(ProdList.getNumSoldTot(i))) {**

**prodListTbl.setValueAt(Integer.toString(ProdList.getAmtOnShelf(i)), i, 2);**

**prodListTbl.setValueAt(Integer.toString(ProdList.getNumSoldDay(i)), i, 3);**

**prodListTbl.setValueAt(Integer.toString(ProdList.getNumSoldTot(i)), i, 4);**

**prodListTbl.setValueAt("$" + String.format("%.2f",**

**((ProdList.getPrice(i) \* ProdList.getNumSoldTot(i)))), i, 5);**

**}**

**}**

**break;**

This will update the lists on the JTables by accessing the most recent data in the ProdList class. The last case decides what happens when “Next Day” is pressed:

**case "Next Day" :**

**// Switch daily items to 0, and update them on the list**

**supplySumObj.SwitchDay();**

**dayLab.setText("Day : " + Integer.toString(supplySumObj.getDay()));**

**for (int i = 0; i < ProdList.getListSize(); i++) {**

**prodListTbl.setValueAt("0", i, 3);**

**}**

**// Update labels**

**dlyMoneyLab.setText(" Amount earned today: $" +**

**String.format("%.2f", registerObj.getDlyMoney()));**

**ttlMoneyLab.setText("Total $ Earned: " +**

**String.format("%.2f", supplySumObj.getTtlMoney()));**

**break;**

This will switch it over to the next day and reset any daily variables.