**Section 1 – Travel**

package week\_7.q1\_travel\_wish\_list;

import java.util.Date;

/\*\*

\* Represents one place in a wish list of places to travel to.

\*/

public class Place implements Comparable<Place> {

// Private fields

private String name; // A String for the place name (for example, Hawaii)

private String reason; // A String reason (a reason for visiting, for example, to go surfing)

private Date created; // A Date created (when the Place object was created)

/\*\*

\* Constructor that takes two arguments, the name and the reason.

\* The constructor creates and sets the Date created.

\*

\* @param name A String for the place name

\* @param reason A String reason for visiting the place

\*/

public Place(String name, String reason) {

this.name = name;

this.reason = reason;

this.created = new Date(); // Set the current date/time as the creation date/time

}

// Getters and setters

/\*\*

\* Get the name of the place

\*

\* @return A String representing the place name

\*/

public String getName() {

return name;

}

/\*\*

\* Set the name of the place

\*

\* @param name A String representing the new place name

\*/

public void setName(String name) {

this.name = name;

}

/\*\*

\* Get the reason for visiting the place

\*

\* @return A String representing the reason for visiting the place

\*/

public String getReason() {

return reason;

}

/\*\*

\* Set the reason for visiting the place

\*

\* @param reason A String representing the new reason for visiting the place

\*/

public void setReason(String reason) {

this.reason = reason;

}

/\*\*

\* Get the date and time the Place object was created

\*

\* @return A Date object representing the creation date/time

\*/

public Date getCreated() {

return created;

}

/\*\*

\* Set the date and time the Place object was created

\*

\* @param created A Date object representing the new creation date/time

\*/

public void setCreated(Date created) {

this.created = created;

}

// Comparable interface method

/\*\*

\* Compare two Place objects based on their name.

\*

\* @param other The other Place object to compare to

\* @return An integer representing the comparison result

\*/

@Override

public int compareTo(Place other) {

return this.name.compareTo(other.name);

}

// toString method

/\*\*

\* Return a human-readable String representation of the Place object

\*

\* @return A String with all the information about the Place object

\*/

@Override

public String toString() {

return "Place to visit: " + name + ". Reason: " + reason + ". Date created: " + created.toString();

}

}

**Section 1 – Wishlist**

package week\_7.q1\_travel\_wish\_list;

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

import static input.InputUtils.stringInput;

import static input.InputUtils.yesNoInput;

/\*\*

Program to create and display a list of places a user wishes to travel to.

\*/

public class WishList {

public static void main(String[] args) {

// Create an ArrayList to store Place objects

List<Place> wishList = new ArrayList<>();

// Prompt the user to add a new place to their wish list

do {

String name = stringInput("Enter the name of the place");

String reason = stringInput("Why do you want to visit " + name + "?");

// Call the newPlace method to create a new Place object with the given name and reason

Place newPlace = newPlace(name, reason);

// Add the new Place object to the wishList

wishList.add(newPlace);

} while (yesNoInput("More places to add to your wish list?"));

// Call the displayPlacesInNameOrder method to print the list of places sorted by name

displayPlacesInNameOrder(wishList);

}

public static Place newPlace(String placeName, String reason) {

// Create a new Place object with the given name and reason

Place place = new Place(placeName, reason);

// Return the new Place object

return place;

}

public static void displayPlacesInNameOrder(List<Place> places) {

// Sort the list of Place objects in name order, ignoring case

Collections.sort(places, new Comparator<Place>() {

@Override

public int compare(Place p1, Place p2) {

return p1.getName().compareToIgnoreCase(p2.getName());

}

});

// Print each Place object, one per line

for (Place place : places) {

System.out.println(place);

}

}

}

**Q2 Problem-1**

The TicketManager class is responsible for managing the different operations related to the ticket system. It delegates responsibilities to other classes such as TicketUI for displaying information and getting input from the user, TicketStore for storing and managing tickets, and ResolvedTicketStore for storing resolved tickets. TicketCounter is responsible for generating new ticket IDs.

TicketManager delegates tasks related to displaying information and getting user input to TicketUI. It delegates tasks related to storing and managing tickets to TicketStore. This separation of responsibilities makes the code easier to understand and maintain. If TicketManager did everything, the code would become more complex and harder to maintain.

If TicketStore used permanent storage like a database, TicketUI or TicketManager would need to interact with the database to retrieve and store data instead of using the in-memory LinkedList.

TicketCounter is used to generate unique ticket IDs for new tickets. It keeps track of the last used ticket ID and increments it by one to generate a new ID for the next ticket.

Ticket\_Manager accesses a TicketStore object using the getInstance() method of the TicketStore class. This method returns a singleton instance of the TicketStore object, which ensures that there is only one instance of TicketStore throughout the application. It is not possible to create more than one TicketStore object because of the Singleton design pattern, which restricts the instantiation of a class to one object. This pattern is used to ensure that only one instance of the class is created and used throughout the application.

**Q2 TicketUI**

package week\_7.q2\_ticket;

import java.util.\*;

import static input.InputUtils.\*;

/\*\* User interaction - user input and printing - for the ticket manager \*/

public class TicketUI {

// Constants used to display menu options, and figure out what user entered.

// TODO Problem 3: add constants for the two new options given in the instructions.

// Make sure you use int values that haven't been used for other menu options.

static final int ADD\_TICKET\_MENU\_OPTION\_NUMBER = 1;

static final int SEARCH\_BY\_ID\_MENU\_OPTION\_NUMBER = 2;

static final int DELETE\_BY\_TICKET\_ID\_MENU\_OPTION\_NUMBER = 3;

static final int SHOW\_NEXT\_TICKET\_MENU\_OPTION\_NUMBER = 4;

static final int SHOW\_ALL\_TICKETS\_MENU\_OPTION\_NUMBER = 5;

static final int DELETE\_BY\_DESCRIPTION = 6;

static final int SEARCH\_BY\_DESCRIPTION = 7;

static final int QUIT\_MENU\_OPTION\_NUMBER = 99;

private Map<Integer, String> options = new TreeMap<>();

/\*\* This constructor sets up the map of options for the menu.

\* The keys are numbers and correspond to number the use will enter to make a choice

\* We are using constants that store the numbers 1, 2, 3 ... instead of numbers 1, 2, 3...

\* for two reasons.

\*

\* Reason 1: it's easier to see the meaning of a constant called ADD\_TICKET\_MENU\_OPTION\_NUMBER

\* than the number 1, especially in other parts of the code. Note that TicketManager

\* has a block of code that starts like this,

\*

\* if (userChoice == TicketUI.ADD\_TICKET\_MENU\_OPTION\_NUMBER) {

\* menuOptionAddTicket();

\* } else if (userChoice == TicketUI.SEARCH\_BY\_ID\_MENU\_OPTION\_NUMBER) {

\* menuOptionSearchById();

\* } else if (userChoice == TicketUI.DELETE\_BY\_TICKET\_ID\_MENU\_OPTION\_NUMBER) {

\* .... etc...

\*

\* Which is much clearer than

\*

\* if (userChoice == 1) { // now we have to remember what 1 means

\* menuOptionAddTicket();

\* } else if (userChoice == 2) {

\* menuOptionSearchById();

\* } else if (userChoice == 3) {

\* .... etc...

\*

\* Because we have to remember what 1 means, and what 2 means, and what 3 means...

\* So, using a named constant or variable that contains a value, instead of

\* using the value directly, makes code more readable.

\*

\* Reason 2: If we want to change the numeric values of the options, then we only need to

\* change the value of the constants in code in this file, and we don't need to modify

\* TicketManager, since it uses the value of those constants. Our code is easier to modify.

\*

\* The values are a text description of that menu item. \*/

TicketUI() {

options.put(ADD\_TICKET\_MENU\_OPTION\_NUMBER, "Add new ticket");

options.put(SEARCH\_BY\_ID\_MENU\_OPTION\_NUMBER, "Search by ticket ID");

options.put(DELETE\_BY\_TICKET\_ID\_MENU\_OPTION\_NUMBER, "Delete by ticket ID");

options.put(SHOW\_NEXT\_TICKET\_MENU\_OPTION\_NUMBER, "Show next ticket in ticket queue");

options.put(SHOW\_ALL\_TICKETS\_MENU\_OPTION\_NUMBER, "Show all open tickets");

// TODO Problem 3: add the new options to the options HashMap.

options.put(QUIT\_MENU\_OPTION\_NUMBER, "Quit the program");

}

/\*\* Prints a menu using the options Map and asks the user for their

\* choice. Validates that the choice made is one of the available choices from the Map.

\*/

protected int showMenuGetChoice() {

while (true) {

for (int option : options.keySet()) {

System.out.printf("%d: %s\n", option, options.get(option));

}

int userSelection = intInput("Enter your selection");

// If the user's option is in the map's key set, it's a valid choice. Return it.

if (options.containsKey(userSelection)) {

return userSelection;

}

// Otherwise, loop until user enters valid choice from the menu

}

}

/\*\*

\* Ask user for the description, reporter, and priority of a new ticket.

\* The date created is assumed to be the time this code runs

\* @return a new Ticket object with the data entered by the user and the current date. \*/

protected Ticket getNewTicketInfo() {

Date dateReported = new Date(); // Default Date constructor creates Date with current date & time

String description = stringInput("Enter description of the problem: ");

String reporter = stringInput("Who reported this problem? ");

int priority = 0;

while (priority < 1 || priority > 5) {

priority = intInput("Enter priority of " + description + " (1-5): ");

}

return new Ticket(description, priority, reporter, dateReported);

}

/\* Methods for getting information from the user \*/

/\*\* Ask user for a ticket ID, ensure it is an integer

\*

\* @return the ticket ID entered.

\*/

protected int getTicketID() {

return intInput("Enter Ticket ID: ");

}

/\*\* Print a question or a message, and ask the user to enter a text response

\* @param question the text to print

\* @return the String the user enters in response

\*/

protected String askUserQuestion(String question) {

return stringInput(question);

}

/\*\* Print a question or a message, and ask the user to enter 'yes' or 'no'.

\* @param question the text to print

\* @return a boolean true if the user enters 'yes' or false if the user enters 'no'

\*/

protected boolean areYouSure(String question) {

return yesNoInput(question);

}

/\* Methods for displaying information to the user \*/

/\*\* Displays a message to the user. \*/

public void userMessage(String s) {

System.out.println(s);

}

/\*\* If there are tickets in the ticket list, print each Ticket, one per line.

\* If the ticket list is empty, print a "no tickets" message.

\*

\* @param tickets a list of Ticket objects. The list may be empty, but may not be null.

\*/

protected void displayTickets(List<Ticket> tickets) {

if (tickets.isEmpty()) {

System.out.println(" \*\*\*\*\*\*\*\* No tickets \*\*\*\*\*\*\*\* ");

} else {

System.out.println(" --------- All tickets --------- ");

for (Ticket t : tickets) {

displayTicket(t);

}

System.out.println(" --------- End of ticket list --------- ");

}

}

/\*\* Prints information about one Ticket.

\* @param ticket the ticket to print. \*/

public void displayTicket(Ticket ticket) {

System.out.println(ticket);

}

}

**Q2 TicketStore**

package week\_7.q2\_ticket;

import java.util.\*;

/\*\*

\* A data structure to store Tickets in memory as the program runs

\* Stores tickets in priority order, so tickets with priority 5 are at the start

\* If more than one ticket with same priority, oldest tickets are before newer tickets

\* Supports add, delete, search operations on the list of Tickets

\*

\*

\* Implemented as a Singleton - there can only ever be one TicketStore.

\* If you want to work with the TicketStore, call TicketStore.getInstance()

\*

\*/

public class TicketStore {

private LinkedList<Ticket> ticketQueue;

private static TicketStore instance;

private TicketStore() {

ticketQueue = new LinkedList<>();

}

public static TicketStore getInstance() {

if (instance == null) {

instance = new TicketStore();

}

return instance;

}

/\*\* Add all of the tickets in a list to the Ticket Queue, in priority order.

\* @param tickets a list of tickets. \*/

public void addAll(LinkedList<Ticket> tickets) {

ticketQueue.addAll(tickets);

Collections.sort(ticketQueue);

}

/\*\* Add ticket, and then sort list to keep the highest-priority at the top of the queue.

\* @param newTicket the new Ticket to add. \*/

public void add(Ticket newTicket) {

ticketQueue.add(newTicket);

Collections.sort(ticketQueue);

}

/\*\* Returns all tickets in the queue.

\* @return All the tickets. \*/

public LinkedList<Ticket> getAllTickets() {

return ticketQueue;

}

/\*\* Returns, but does not remove, the tickets from the top of the TicketQueue

\* @return the ticket at the top of the queue \*/

public Ticket peekNextTicket() {

return ticketQueue.peek();

}

/\*\* How many currently open tickets in the list?

\* @return the number of open tickets \*/

public int ticketsInQueue() {

return ticketQueue.size();

}

/\*\* Searches store for ticket with given ID.

\* @param id The ticket ID

\* @return The ticket with this ID, if found; null otherwise \*/

public Ticket getTicketById(int id) {

for (Ticket t : ticketQueue) {

if (t.getTicketID() == id) {

return t;

}

}

return null; // If ticket with this ID is not found

}

/\*\* Delete a single ticket by ticket ID.

\* @return true if a ticket was found and deleted, false if a ticket with this ID is not in the queue \*/

public boolean deleteTicketById(int deleteID) {

//Loop over all tickets. Delete the one with this ticket ID

for (Ticket ticket : ticketQueue) {

if (ticket.getTicketID() == deleteID) {

ticketQueue.remove(ticket);

return true;

}

}

// Not found? Return false

return false;

}

/\*\* Create a list of tickets, with a description containing

\* the given String. The search is not case sensitive.

\* @param description Text to search for in Ticket descriptions

\* @return a list of matching Tickets. If no matches, return an empty list.

\*/

public List<Ticket> searchByDescription(String description) {

List<Ticket> matchingTickets = new ArrayList<>();

if (description == null || description.isEmpty()) {

return matchingTickets; // Return an empty list if the search string is null or empty

}

for (Ticket ticket : ticketList) {

if (ticket.getDescription().toLowerCase().contains(description.toLowerCase())) {

matchingTickets.add(ticket); // Add the ticket to the matching list if the description contains the search string

}

}

return matchingTickets;

}

/\*\* Remove all tickets from the Ticket queue \*/

public void removeAll() {

ticketQueue.clear();

}

}

**Q2 Ticket Manager**

package week\_7.q2\_ticket;

import java.util.LinkedList;

/\*\*

\* This code controls the program. It shows the menu, asks user for their choice,

\* and carries out that action.

\* It delegates tasks to methods in TicketUI, TicketStore and ResolvedTicketStore.

\*

\* The instruction are in the Lab 7 Questions.md file \*/

public class TicketManager {

public static void main(String[] args) {

new TicketManager().manage();

}

// Global objects - the data stores, and the user interface object

// you can access these from any method in this class

private TicketUI ticketUI = new TicketUI(); // delegate all user interaction to the ticketUI object

private TicketStore ticketStore = TicketStore.getInstance(); // stores all the open tickets

// TODO Problem 6: create a ResolvedTicketStore object here.

private ResolvedTicketStore resolvedTickets = ResolvedTicketStore.getInstance();

void manage() {

boolean wantToQuit = false;

while (!wantToQuit) {

int userChoice = ticketUI.showMenuGetChoice();

// TODO Problem 3: extend this if-else if-else statement to add the new

// choices for the two new menu options here. Call the appropriate method for each choice.

if (userChoice == TicketUI.ADD\_TICKET\_MENU\_OPTION\_NUMBER) {

menuOptionAddTicket();

} else if (userChoice == TicketUI.SEARCH\_BY\_ID\_MENU\_OPTION\_NUMBER) {

menuOptionSearchById();

} else if (userChoice == TicketUI.DELETE\_BY\_TICKET\_ID\_MENU\_OPTION\_NUMBER) {

menuOptionDeleteById();

} else if (userChoice == TicketUI.SHOW\_NEXT\_TICKET\_MENU\_OPTION\_NUMBER) {

menuOptionShowNextTicket();

} else if (userChoice == TicketUI.SHOW\_ALL\_TICKETS\_MENU\_OPTION\_NUMBER) {

menuOptionDisplayAllTickets();

} else if (userChoice == TicketUI.QUIT\_MENU\_OPTION\_NUMBER) {

wantToQuit = true;

} else if (userChoice == week\_7.q2\_ticket.TicketUI.DELETE\_BY\_DESCRIPTION) {

menuOptionDeleteTicketByDescription();

} else if (userChoice == week\_7.q2\_ticket.TicketUI.SEARCH\_BY\_DESCRIPTION) {

menuOptionSearchByDescription();

}

}

}

private void menuOptionAddTicket() {

// Get ticket data. Use ticketUI method to ask for data.

Ticket newTicket = ticketUI.getNewTicketInfo();

// Add to the ticket store

ticketStore.add(newTicket);

ticketUI.userMessage("Ticket added to the ticket queue");

}

/\* Ask for a ticket ID, request store searches for a ticket with that ID, displays the

\* ticket if one is found, or a message if the ticket with that ID is not found. \*/

protected void menuOptionSearchById() {

int ticketID = ticketUI.getTicketID();

Ticket ticket = ticketStore.getTicketById(ticketID);

if (ticket == null) {

ticketUI.userMessage("No ticket found with that ID");

} else {

ticketUI.displayTicket(ticket);

}

}

/\* Ask for ticket ID, call deleteTicketByID to find ticket, confirm delete operation, and delete \*/

protected void menuOptionDeleteById() {

int ticketID = ticketUI.getTicketID();

deleteTicketById(ticketID);

}

/\* Display next ticket in the queue - the highest priority \*/

protected void menuOptionShowNextTicket() {

Ticket next = ticketStore.peekNextTicket();

ticketUI.displayTicket(next);

}

/\* Show all tickets, in priority order \*/

protected void menuOptionDisplayAllTickets() {

LinkedList<Ticket> allTickets = ticketStore.getAllTickets();

ticketUI.displayTickets(allTickets);

}

/\* Ask user for a search term, display all tickets matching that term \*/

protected void menuOptionSearchByDescription() {

String searchTerm = ticketUI.askUserQuestion("Enter search term: ");

List<Ticket> matchingTickets = ticketStore.searchByDescription(searchTerm);

if (matchingTickets.isEmpty()) {

ticketUI.displayMessage("No matching tickets.");

} else {

ticketUI.displayTickets(matchingTickets);

}

}

/\* Ask user for a search term, display all matching tickets, ask for ID of ticket to delete \*/

protected void menuOptionDeleteTicketByDescription() {

// Ask user for search term

String searchTerm = ticketUI.askUserQuestion("Enter a search term:");

// Find matching tickets

List<Ticket> matchingTickets = ticketStore.searchByDescription(searchTerm);

// If there are no matching tickets, display a message and return

if (matchingTickets.isEmpty()) {

ticketUI.userMessage("No matching tickets.");

return;

}

// Display list of matching tickets

ticketUI.displayTickets(matchingTickets);

// Ask user for ticket ID to delete or to cancel

int ticketID = ticketUI.getTicketID();

if (ticketID == -1) {

// User cancelled, do nothing

return;

}

// Delete the selected ticket

deleteTicketById(ticketID);

}

/\* Fetch a ticket with the given ID, verify user wishes to delete, gather resolution information.

Remove ticket from store, and add to resolved tickets. \*/

protected void deleteTicketById(int ticketID) {

// Fetch the ticket

Ticket ticketToDelete = ticketStore.getTicketById(ticketID);

// Check if a ticket with this ID is in the store

if (ticketToDelete == null) {

ticketUI.userMessage("Ticket not found in the ticket store.");

} else {

// Display ticket info, and verify delete operation

ticketUI.displayTicket(ticketToDelete);

if (ticketUI.areYouSure("Delete the above ticket, are you sure?")) {

// Ask for resolution and save it

String resolution = ticketUI.getResolution();

ticketToDelete.setResolution(resolution);

// Save current date and time as the date resolved

ticketToDelete.setDateResolved(LocalDateTime.now());

// Add ticket to resolved ticket store

ResolvedTicketStore.getInstance().addResolvedTicket(ticketToDelete);

// Delete ticket from ticket store

ticketStore.deleteTicketById(ticketID);

// Display confirmation message

ticketUI.userMessage("Ticket deleted.");

} else {

ticketUI.userMessage("Cancelled - ticket not deleted.");

}

}

}

}

**Q2 Ticket Counter**

package week\_7.q2\_ticket;

/\*\*

\* This class keeps track of the Ticket ID numbers

\* Each Ticket gets its own unique ID, starting from 1 and counting up

\*

\* You don't need to modify this class

\*/

public class TicketCounter {

private static int counter = 1;

public static int getNextCounterValue() {

return counter++;

}

public static void setCounter(int value) {

counter = value;

}

}

**Q2 Ticket**

package week\_7.q2\_ticket;

import java.util.Date;

public class Ticket implements Comparable<Ticket> {

private int priority;

private String reporter; // Stores person or department who reported problem

private String description;

private String resolution;

private Date dateReported;

private Date dateResolved;

private final int ticketID; // The ID for each ticket - an instance variable. Each Ticket will have it's own ticketID variable

// TODO Problem 5: Tickets need to store the resolution Date in a variable called dateResolved

// TODO Problem 5: and a String describing the resolution, in a variable called resolution.

// Add two new fields (variables) here.

// You do not need to modify the constructor. When a new tickets is created,

// the user will not know the data resolved or the resolution.

public Ticket(String desc, int p, String rep, Date date) {

this.description = desc;

this.priority = p;

this.reporter = rep;

this.dateReported = date;

this.ticketID = TicketCounter.getNextCounterValue();

}

public int getTicketID() {

return ticketID;

}

protected int getPriority() {

return priority;

}

public void setPriority(int priority) {

this.priority = priority;

}

public String getDescription() { return description; }

public String getResolution() { return resolution; }

public void setDescription(String description) {

this.description = description;

}

public void setResolution(String resolution) {

this.resolution = resolution;

}

public String getReporter() {

return reporter;

}

public void setReporter(String reporter) {

this.reporter = reporter;

}

public Date getDateReported() {

return dateReported;

}

public Date getDateResolved() {

return dateResolved;

}

public void setDateReported(Date dateReported) {

this.dateReported = dateReported;

}

public void setDateResolved(Date dateResolved) {

this.dateResolved = dateResolved;

}

public String toString(){

if (this.resolved) {

return "ID: " + this.ticketID

+ " Description: " + this.description

+ " Priority: " + this.priority

+ " Reported by: " + this.reporter

+ " Reported on: " + this.dateReported

+ " Resolution: " + this.resolution

+ " Date resolved: " + this.dateResolved;

} else {

return "ID: " + this.ticketID

+ " Description: " + this.description

+ " Priority: " + this.priority

+ " Reported by: " + this.reporter

+ " Reported on: " + this.dateReported;

}

}

@Override

public int compareTo(Ticket otherTicket) {

// Joint equal priorities, sort with oldest first

if (this.getPriority() == otherTicket.getPriority()){

return this.getDateReported().compareTo(otherTicket.getDateReported());

}

else {

// Sort with smallest priority number at the start

return this.getPriority() - otherTicket.getPriority();

}

}

}

**Q2 ResolvedTicketStore**

package week\_7.q2\_ticket;

import java.util.LinkedList;

import java.util.List;

/\*\*

\*

\* Storage for Resolved Ticket objects.

\* Implemented as a "Singleton" - there can only ever be one ResolvedTicketStore.

\* This avoids the possibility of creating multiple stores, and then

\* If there is only one ResolvedTicketStore, then all the the tickets must be in this

\* single store. There's no way to have multiple stores and multiple places where tickets might be.

\*

\* Singleton is a design pattern, a way of doing things in code. More info:

\* https://www.geeksforgeeks.org/singleton-class-java/

\*

\* If you want to work with the ResolvedTicketStore, call ResolvedTicketStore.getInstance()

\*

\* You don't need to modify this class.

\*

\* You will need to call the addTicket method.

\* \*/

public class ResolvedTicketStore {

private static LinkedList<Ticket> resolvedTickets;

private static ResolvedTicketStore instance;

private ResolvedTicketStore() {

resolvedTickets = new LinkedList<>();

}

public static ResolvedTicketStore getInstance(){

if (instance == null) {

instance = new ResolvedTicketStore();

}

return instance;

}

public void addTicket(Ticket t) {

resolvedTickets.add(t);

}

public List<Ticket> getAll() {

return resolvedTickets;

}

}