**Dawson Hotel Reservation System**

**Project Specifications**

**Phase II**

**Due: Friday, October 21**

The typical workflow in the final hotel reservation system will use a single room datastore, a single customer datastore and a single reservation datastore (each either a file or a database table).

In this phase, you must load and sort the unsorted sequential rooms file into a sorted sequential file. You will next load, sort and merge 10 unsorted sequential customer files into a sorted sequential file. You will then load, sort and merge 10 unsorted sequential reservation files into a sorted sequential file. Recall that each file contains a series of asterisks delimited records as specified in the preparatory phase of the project. The individual files will be posted on the S; drive in the **317\datafiles\unsorted** folder. With Windows Explorer, copy the **unsorted** folder into your **dcs314\eclipse\ReservationSys datafiles** folder. Note that you must not change the original contents of the unsorted sequential files.

**Part I - The groupX.hotel.data.HotelFileLoader class.**

Create a utility class called HotelFileLoader (with a private no-parameter constructor to prevent instantiation) as a member of the groupX.hotel.data package. This class will contain a series of static utility methods that can be used to load files of various formats (i.e. text sequential and binary files). During this phase, we will only be concerned with text sequential files.

* Code a **static getRoomListFromSequentialFile method**. The method must have the following method interface (make sure to properly document it using JavaDoc style comments).

public static Room[] getRoomListFromSequentialFile(String filename)

throws IOException

* The Room array returned by the above method must be an array whose size is equal to its capacity (i.e. the array must be full to capacity).
* Create a test application as a member of the groupX.hotel.data package in your test source folder. Execute your test application to ensure that the method functions as expected. Create and use files with test data to ensure that you test all conditions including incorrect records.
* Code a **static getCustomerListFromSequentialFile method**. The method must have the following method interface (make sure to properly document it using JavaDoc style comments).

public static Customer[] getCustomerListFromSequentialFile(String filename)

throws IOException

* The Customer array returned by the above method must be an array whose size is equal to its capacity (i.e. the array must be full to capacity).
* Add a method in the test application to ensure that the method functions as expected. Create and use files with test data to ensure that you test all conditions.

**Part II – Expand the ListUtilities class by adding a static generic save, sort and merge methods.**

* Create a ListUtilities class as a member of a package called groupX.util in your CommonX project folder. The ListUtilities class will contain a set of static methods for manipulating list objects. (nb: you may have already done this as part of **Practice Exercise 3 in Lab 4)**
* In order to prevent instantiation of the class, code a private no-parameter constructor.
* Add a general static **saveListToTextFile** method (as described in the lectures and lab 7) to the ListUtilities class.
* Code a test application as a member of a package called groupX.util in your CommonX project’s test source folder. Execute your test application to ensure that the save method functions as expected.
* Add a generic static **sort** method (as described by the javaDoc below) with the following method interface to the ListUtilities class. Note that for this phase we will ignore the rawType and unchecked warnings.

/\*

\* Sorts a list of objects in ascending natural order using

\* selection sort.

\*

\* Precondition: Assumes that the list is not null and that the

\* list's capacity is equal to the list's size.

\*

\*

\* @param list A list of objects. Assumes that the

\* list's capacity is equal to the list's size.

\*

\* @throws IllegalArgumentException if the parameter is \* not full to capacity.

\*

\* @throws NullPointerException if the list is null.

\*/

@SuppressWarnings({ "rawtypes", "unchecked" })

public static void sort(Comparable[] list)

throws IllegalArgumentException, NullPointerException

* Code your selection sort method test application. Execute the test application to ensure that the sort method functions as expected.
* Add a generic **merge method** to the ListUtilities class (as described by the JavaDocs below). The merge method must have the following interface and must return a reference to a list object whose **capacity is equal to its size**.

/\*

\* Efficiently merges two sorted lists of objects in ascending

\* natural order. If the duplicate objects are in both lists,

\* the object from list1 is merged into the resulting list, and

\* both objects are written to the duplicate file.

\*

\* Precondition: Assumes that the lists are not null and that

\* both lists contain objects that can be compared to

\* each other and are filled to capacity.

\*

\*

\* @param list1 A naturally sorted list of objects. Assumes

\* that the list contains no duplicates and that

\* its capacity is equal to its size.

\* @param list2 A naturally sorted list of objects. Assumes

\* that the list contains no duplicates and that

\* its capacity is equal to its size.

\* @param duplicateFileName The name of the file in

\* datafiles\duplicates to which duplicate pairs

\* will be appended.

\*

\* @throws IllegalArgumentException if either parameter is

\* not full to capacity.

\*

\* @throws NullPointerException if the either list is

\* null.

\*/

@SuppressWarnings({ "rawtypes", "unchecked" })

public static Comparable[] merge(Comparable[] list1,

Comparable[] list2, String duplicateFileName)

throws IOException

The method must merge two sorted list objects referenced by **list1** and **list2** into a third list object (created in the merge method). When the lists have been merged, the merge method returns a reference to the third list object. NOTE: this method must be **efficient**! Take advantage of the fact that the two lists are already sorted!

In order to create the third list from within the merge method with the same base type as those of the arrays that you are merging you will have to use the approach below. **This is necessary to ensure that you will not end up with a ClassCastException when you cast the array returned from the merge array to an Customer[ ] or Reservation[] in the application class.**

**Note that the Array class is a member of the java.lang.reflect package.**

Comparable[] list3 = (Comparable[]) Array.*newInstance*(

list1.getClass().getComponentType(), list1.length + list2.length);

If the equal objects appears in both of the lists that are being merged (whether the lists contain Customers or Reservations), merge the customer or reservation from list1 and record the identical customer or reservation in the duplicates file (which must be stored a directory called **datafiles\duplicates**), make sure to indicate which of the duplicate customer/reservation was actually added to the merged list (by appending the string “(merged)” at the end of the string representation of the customer/reservation that was merged in the list as shown in the sample below). Note that the duplicates file must be opened for append in the merge method using the filename referenced by its third parameter.

**Sample duplicates message:**

Duplicate element:

j@doe.com\*John\*Doe\*visa\*4556737586899855 (merged)

j@doe.com\*Jane\*Doe\*\*

* Create a test application as a member of a package called groupX.util in your test source folder. Execute your test application to ensure that the merge method functions as expected.

**Part III – Add getReservationList method to the groupX.hotel.data.HotelFileLoader class.**

* Code a **static getReservationListFromSequentialFile method**. The method must have the following method interface (make sure to properly document it using JavaDoc style comments).

public static Reservation[] getReservationListFromSequentialFile

(String filename,

Customer[] customerList,

Room[] roomList)

throws IOException, IllegalArgumentException

This method must use the email field to find the associated customer, and the room number to find the associated room. NOTE: you will implement a simple sequential search though the customerList and roomList in this phase. If the customer or room cannot be found, throw an IllegalArgumentException .

* Create a test application as a member of the groupX.hotel.data package in your test source folder. Execute your test application to ensure that the method functions as expected.

**Part IV – Defining more ways to sort Reservations**

Add a second generic static **sort** method (as described by the javaDoc below) with the following method interface to the **ListUtilities** class. Note that for this phase we will ignore the rawType and unchecked warnings.

/\*

\* Sorts a list of objects in the given order.

\*

\* Precondition: Assumes that the list is not null and that the

\* list's capacity is equal to the list's size.

\*

\*

\* @param list A list of objects. Assumes that the

\* list's capacity is equal to the list's size.

\* @param sortOrder A Comparator object that defines the

\* sort order

\*

\* @throws IllegalArgumentException if the parameter is \* not full to capacity.

\*

\* @throws NullPointerException if the list or sortOrder \* are null.

\*/

@SuppressWarnings({ "rawtypes", "unchecked" })

public static void sort(Comparable[] list, Comparator sortOrder)

throws IllegalArgumentException, NullPointerException

This method will simply use the java.util.Arrays.sort 2-parameter overloaded method.

In package groupX.hotel.business, code the following **ReservationByCustSorter** class.

package groupX.hotel.business;

import java.util.Comparator;

import dw317.hotel.business.interfaces.Reservation;

public class ReservationByCustSorter implements Comparator<Reservation> {

@Override

public int compare(Reservation r1, Reservation r2) {

if (r1.equals(r2))

return 0;

if (!r1.getCustomer().equals(r2.getCustomer()))

return r1.getCustomer().compareTo(r2.getCustomer());

//if same customer, order by check in (i.e., default)

return r1.compareTo(r2);

}

}

ReservationByCustSorter implements the Comparator<Reservation> interface to enable sorting of Reservations on customer and checkin date. The Comparator interface defines the compare method.

* Create a test application as a member of the groupX.hotel.data package in your test source folder. Execute your test application to ensure that the method functions as expected.
* Now code your own **ReservationByCheckOutSorter** that implements the Comparator<Reservation> interface. This will compare two Reservations based on their checkout date. If two reservations have the same checkout date, order them by the default order (i.e., compareTo) Test your method.

**Part V – Code the application class.**

The SortMergeApp class (member of the groupX.hotel.data package) will be responsible for loading and sorting the room file. The sorted rooms file must be stored in a file called **rooms.txt** in a subdirectory called **datafiles\database.** Each record in the file will consist of a string representation of a room object (formatted as 108\*normal)

Note that all new directories must be created by the application class.

Next, the SortMergeApp class will load, sort, and merge the 10 original customer files. The individual sorted customer files must be stored in a subdirectory called **datafiles\sorted** and appropriately called sortedCustomers1.txt, …, sortedCustomers10.txt. Each record in the file will consist of a string representation of a customer object (formatted as raj@king.ru\*Raj\*Wong\*visa\*4556737586899855).

The complete merged customers must be stored in a file called **customers.txt** in the **datafiles\database** subdirectory**, and duplicates in** duplicateCustomers.txt.

Next, the SortMergeApp class will load, sort, and merge the 10 original reservation files. The sorted files must be stored in a subdirectory called **datafiles\sorted.** The files must be appropriately called sortedResevations1.txt, …, sortedResevations10.txt. The reservations are merged, and stored in a file called **reservations.txt** in a subdirectory called **datafiles\database.** Each record in the file will consist of a string representation of a reservation object (formatted as raj@wong.ru\*2016\*9\*10\*2016\*9\*15\*507). Store duplicates in duplicateReservations.txt.

**Requirements:**

Include the following in your submission folder:

1. Meeting logs and member contribution sheet.
2. A hardcopy of the properly documented source files (only those changed in this phase, including ListUtilities class).
3. A hardcopy of the contents of the following data files.

rooms.txt (sorted version)

customers.txt

reservations.txt

duplicateCustomers.txt (or whatever file name you used to store the

duplicates when merging Customers)

duplicateReservations.txt (or whatever file name you used to store the

duplicates when merging Reservations)

1. An electronic copy (a zip file) of your ReservationSys and CommonX project files. Make sure that it contains all of the required resources.