Homework 2 :: Flights

15-121 Fall 2015
Due: September 15, 2015 (11:50pm)
Basecode, Hand-In

Overview

In this homework, you will be constructing a scaled version of a flight tracker. You will also be implementing a variety of methods aimed at improving your abilities with ArrayLists. As such, it may be helpful to have the ArrayList JavaDoc readily available. You will also be working with a class we've provided for you — the `Flight` class.

As with all assignments, you will be graded in part on your coding style. Your code should be easy to read, organized, and well-documented. Be consistent in your use of indentation and braces. See the style guide for more details. For this homework (but not the next homework!), style penalties will only be half as severe as usual to give you time to learn the syntax of the language and to become familiar with the style we expect.

A word of caution: by the nature of this homework, your data and results may differ from our examples in the spec. The way to check for correctness is to manually examine the `flights.txt` file for expected results. You should not hand in `flights.txt`, so you're welcome to modify it (to, for example, have fewer flights, to make manual checking easier).

Background :: Code

Take a moment to review the code in the `Flight` class. Each method has a comment above it describing what it does. Be familiar with this class before you start coding.

A `Flight` object represents an airline flight from one point to another point. It contains the departure airport (where the flight leaves from) and time (when it leaves) as well as the arrival airport (where the flight goes) and time (when it gets where it's going). It also tells you the distance of the flight from the departure airport to the arrival airport.

Finally, it has a flight identifier. This identifies the flight path that this flight is in. A flight path is a set of flights from the same airline on the same plane. For example, say that United Airlines has one plane that goes from JFK (New York City) to Pittsburgh (PIT) to ORD (Chicago) to LAS (Las Vegas). This flight path would be represented by three `Flight` objects -- one that goes from JFK to PIT, one that goes from PIT to ORD, one that goes from ORD to LAS -- which all have the same flight identifier (e.g. UA2658).

Exercise :: Identify

Download the Basecode and fill in the required fields like so:

```
/**
 * @author [First Name] [Last Name] <[Andrew ID]>
 * @section [Section Letter]
 */
```

```
/**
 * @author Jess Virdo <jvirdo>
 * @section A
 */
```

Exercise :: loadFlights

Complete the `loadFlights` method which reads your given flight data from the `flights.txt` file. A large portion of this method has been provided for you, but you still need to build each `Flight` object and add it to the global `flights ArrayList`. You should examine the `Flight` constructor in the `Flight.java` file to determine the way you create a `Flight` object.

You should also look up the `String.split` method in the Java API and understand how it works.

Exercise :: getFlightsDepartingFrom

The `getFlightsDepartingFrom` method searches the global `flights ArrayList` and returns an `ArrayList<Flight>` that depart from the given airport (e.g. PIT). This method must run in `O(n)` time — in other words, you should only look at each `Flight` in the global `flights ArrayList<Flight>` once in this method.

```
FlightManager f = new FlightManager();
ArrayList<Flight> theFlights = f.getFlightsDepartingFrom("PIT");
System.out.println(theFlights); // could print [DL8273, WN2834, WN5243]
```

Exercise :: getNonStopFlights

The `getNonStopFlights` method takes two parameters, a `departureAirport` and an `arrivalAirport`. It returns an `ArrayList<Flight>` representing all non-stop flight paths between those two airports. A flight path is considered non-stop if one `Flight`'s departure airport and arrival airport match the given parameters simultaneously.

For example, a flight path that goes from Pittsburgh, PA to Las Vegas, NV is non-stop from Pittsburgh, PA to Las Vegas, NV. However, a flight path that goes from Pittsburgh, PA to Las Vegas, NV to Chicago, IL is not non-stop from Pittsburgh, PA to Chicago, IL.

```
PIT -> LAS       // non-stop PIT to LAS
PIT -> LAS -> ORD // non-stop PIT to LAS, non-stop LAS -> ORD, not non-stop PIT -> ORD
```

This method must run in `O(n)` time. Sample output might be:

```
FlightManager f = new FlightManager();
```

```
ArrayList<Flight> theFlights = f.getNonStopFlights("PIT", "LAS");
System.out.println(theFlights); // could print [UL2334, JBU5233]
```

Exercise :: cancelFlights

Complete the `cancelFlights` method, which returns an `ArrayList<Flight>` representing all the flights that have not be canceled. A Flight should be canceled if its `departureAirport` or `arrivalAirport` match the given parameter. To cancel a flight, you must remove it from the `ArrayList`, but we don't want to modify the global `flights`. How do we do this?

The answer is simple: create a deep copy of it. Recall that in Java, objects are assigned by reference. This means that passing an `ArrayList` as a parameter to another method directly modifies the `ArrayList` (this is called a shallow copy). For example:

```
  ArrayList<String> list1 = new ArrayList<String>();
  list1.add("foo");
  list1.add("bar");
  list1.add("zip");

// list1 = [foo, bar, zip]

  list2 = list1;
  list2.remove("foo");

// list2 = [bar, zip]
// list1 = [bar, zip]
```

Because we want to remove flights, we need to make a deep copy first to avoid modifying the original global `ArrayList`. A `deepCopy` method is very simple. Use the following algorithm:

1. Create a new `ArrayList`
2. For each item in the original `ArrayList`, add it to the new `ArrayList`
3. Return the new `ArrayList`

This is a very common design pattern. (You should familiarize yourself with the concept of a deep copy, as you will certainly need to write a deep copy method in future homework assignments, and maybe on quizzes or exams!) A method stub for `deepCopy` has been provided for you. Once you've written a `deepCopy` method, complete the `cancelFlights` method as described above.

In the real world, the algorithm provided for `cancelFlights` is flawed. Do you know why? (Hint: consider the flight path, not just the individual flights.) For 1 point of extra credit, you can implement another method called `cancelFlightsCorrectly` which takes the same parameters as `cancelFlights` but fixes the real world flaw. You must write a comment above `cancelFlightsCorrectly` explaining what the problem is to get the extra credit.

Exercise :: getTotalDistance

Complete the `getTotalDistance` method which returns the total distance traveled by a given list of airline identifiers (i.e. flight paths).

```
  FlightManager f = new FlightManager();
  ArrayList<String> identifiers = new ArrayList<String>();
  identifiers.add("UW3291");
  identifiers.add("JUB3924");
  identifiers.add("SW9329");

Integer sum = f.getTotalDistance(identifiers);
System.out.println(sum); // could print 7291
```

Exercise :: arrangeFlights

This is probably the most difficult part of this assignment. You should budget extensive time for completing this method. You may find it helpful to use this method in later parts of the assignment.

The `arrangeFlights` method does not take any parameters. It arranges all flights in the global `flights ArrayList<Flight>` by the unique flight identifier -- that is, it groups out each flight path. For example, if our `flights.txt` contained:

```
DL8273|PIT|0920|JFK|1065|421
DL8273|JFK|1140|ORD|1382|814
DL1111|JFK|1110|PIT|1255|421
DL8273|ORD|1450|MDW|1510|55
WN5243|PIT|1750|LAS|1450|2190
```

Then `DL8273`, `DL1111`, and `WN5243` would each be a unique flight path. This structure is an `ArrayList<ArrayList<Flight>>`. At each index, there is an `ArrayList<Flight>`, each of which contains all of the `Flight` objects for a given flight identifier. Here is a visual example using the previous example, where the horizontal row is the outermost `ArrayList<ArrayList<Flight>>`, and the columns are the inner `ArrayList<Flight>`.

```
------[0]------------------[1]--------------------[2]
[0] (DL8273)          [0] (DL1111)           [0] (WN5243)
   PIT => JFK            JFK => PIT             PIT => LAS
       |
[1] (DL8273)
   JFK => ORD
       |
[2] (DL8273)
   ORD => MDW
```

To make this method a little easier, the flights should be added in the order in which they appear in the `flights.txt` file. This is because the flights in the `flights.txt` are organized in sequential order, i.e., the DL8273 Flights will be in the order PIT => JFK, JFK => ORD, and ORD => MDW.

Here is a usage example:

```
FlightManager f = new FlightManager();
ArrayList<ArrayList<Flight>> arrangedFlights = f.arrangedFlights();

for(ArrayList<Flight> alFlights : arrangedFlights) {
  System.out.println(alFlights);
}

// could produce the following output:
[JBU3288, JBU3288, JBU3288, JBU3288]
[UA284, UA284]
[SW9320, SW9320]
```

Exercise :: getMultiHopFlights

Complete the `getMultiHopFlights` method. It takes two parameters, a `departureAirport` and an `arrivalAirport`, and returns an `ArrayList<ArrayList<Flight>>` of all flights that begin in `departureAirport`, end at `arrivalAirport`, and have at least 1 stop between.

You should return an `ArrayList<ArrayList<Flight>>`.

```
FlightManager f = new FlightManager();
ArrayList<ArrayList<Flight>> theFlights = f.getMultiHopFlights("ORD", "BWI");

// would produce the following output:
[[AA2839(0600), AA2839(0920)], [AA2840(1600), AA2840(1920)]]

ArrayList<ArrayList<Flight>> theFlights = f.getMultiHopFlights("PIT", "LAS");

// would produce the following output:
[]
```

Exercise :: morningDepartingFlights

The `morningDepartingFlights` method returns an `ArrayList<Flight>` of all the flights that depart from the given airport in the AM (i.e. between 0000 and 1159, midnight and noon). This method must run in `O(n)` time.

Exercise :: shortestFlight

The `shortestFlight` method returns an `ArrayList<Flight>` of the Flight(s) that go from the specified departure airport to the specified arrival airport in the fewest number of miles. The shortest flight path can be nonstop or multi-hop. If there is no flight path that goes between the two specified airports, return an empty List.

Submitting your Work

When you have completed the assignment and tested your code thoroughly, create a `.zip` file on your work. Only include the following file(s):

1. `FlightManager.java`

Do not include any `.jar` or `.class` files when you submit, and zip only the files listed above. Do not zip not an entire folder containing the file(s).

Once you've zipped your files, visit the Autolba site -- there is a link on the top of this page -- and upload your zip file.

Keep a local copy for your records.