CS367 Programming Assignment 1

Lecture 1, Spring 2018

Due by 11:59 pm on Thursday, February 8, 2018

In this page: Overview | Specifications | Handing in Related pages: Assignments

Overview

Why are we doing this program?

Description

Epoch Systems of Anorev, Wi is a leading software firm. It often requires that employees visit customers to discuss implementation and integration issues. Such meetings are often called "Go Jive" sessions. To improve employee morale, Epoch has decided to allow employees to declare a "wish list" of destinations. Travel assignments will be matched against employee preferences (to the extent possible). Epoch is not quite ready to go live — it needs a way to manage employees and their wish lists. It has hired you to start working on a simple prototype.

For this assignment you will write a program that constructs a simple database of employees and wish lists containing destinations encoded as airport identification codes (e.g., MSN or ORD). (Click here for a utility that maps airport names and their three letter codes). Your program will process a text file, specified as a command-line argument, using it to construct an employee database.

For Programming Assignment 1 you will not be building an abstract data type (ADT) from scratch. Instead, you will use Java's `ArrayList` class to implement the `EmployeeDatabase` class specified below.

Goals

The goals of this assignment are to:

- Practice compiling and running Java programs in Eclipse.
- Become familiar with Java's `List` interface by using Java's `ArrayList` class.
- Gain experience using iterators (from Java's `Iterator` interface) to traverse lists.
- Get practice throwing exceptions.
- Use command-line arguments.
- Review basic file and console I/O.
- Learn how Graphical User Interfaces (GUIs) are implemented in Java.

Specifications

What are the program requirements?

Input text files

Text files storing employees and destinations will be read by your program. Each line in the text file lists an employee (as a username) followed by the destinations (as airport id codes) in that employee's wish list in the following format:

```
employee,destination1,destination2,...,destinationN
```

where the employee and destinations 1 through N are sequences of characters **excluding commas and spaces** (i.e., "," and " "). You may assume that the text files are in the specified format and contain at least one employee. You may also assume that each employee has at least one destination in their wish list and that there are no duplicate destinations within a single employee's wish list (see this sample input file for an example). You may also assume that employee usernames and airport id codes are unique* and are to be represented by your program as lower-case `String`s. *Note: for both employees and airport id codes,

differences in capitalization should be ignored, e.g., `UGM2017`, `ugm2017`, and `Ugm2017` should all be considered the same and represented in the database as `ugm2017`.

Note: employees are to be added to the employee database in the order in which they appear in the text file.

Information about Java I/O is available in both a long version and a short version. Examples of file input are linked to in those documents.

The `Employee` class

An `Employee` class is provided for you (see `Employee.java`). The `Employee` class represents a single employee. It keeps track of a username (as a `String`) and the airport id codes in the employee's wish list (as a `List` of `Strings`). The `Employee` class has the following constructor and methods:

Constructor	Description
`Employee(String name)`	Constructs an employee whose username is `name`.
Method	**Description**
`String getUsername()`	Return the username of this employee.
`List<String> getWishlist()`	Return the wish list for this employee.

You may **not** modify the `Employee` class. If you need to make a change to the wish list a particular employee has, use `getWishList()` to get the list of airport id codes in the employee's wish list and then use list operations to add, remove, etc.

The `EmployeeDatabase` class

The `EmployeeDatabase` class stores the employees. The `EmployeeDatabase` class has the following constructor and methods (do **not** add any additional public methods other than those listed below):

Constructor	Description
`EmployeeDatabase()`	Constructs an empty employee database.
Method	**Description**
`void addEmployee(String e)`	Add an employee with the given username e to the **end** of the database. If an employee with username e is already in the database, just return.
`void addDestination(String e, String d)`	Add the given destination d to the wish list for employee e in the database. If employee e is not in the database throw a `java.lang.IllegalArgumentException`. If d is already in the wish list for employee e, just return.
`boolean containsEmployee(String e)`	Return true if and only if employee e is in the database.
`boolean containsDestination(String d)`	Return true if and only if destination d appears in at least one employee's wish list in the database.
`boolean hasDestination(String e, String d)`	Returns true if and only if destination d is in the wish list for employee e. If employee e is not in the database, return false.
`List<String> getEmployees(String d)`	Return the list of employees who have destination d in their wish list. If destination d is not in the database, return a null list.
`List<String> getDestinations(String e)`	Return the wish list for the employee e. If an employee e is not in the database, return null.
`Iterator<Employee> iterator()`	Return an `Iterator` over the `Employee` objects in the database. The employees should be returned in the order they were added to the database (resulting from the order in which they are in the text file).
`boolean removeEmployee(String e)`	Remove employee e from the database. If employee e is not in the database, return false; otherwise (i.e., the removal is successful) return true.
`boolean removeDestination(String d)`	Remove destination d from the database, i.e., remove destination

	d from every wish list in which it appears. If destination d is not in the database, return false; otherwise (i.e., the removal is successful) return true.
`int size()`	Return the number of employees in this database.

For the internal structure of the `EmployeeDatabase` class, you may write your own `ArrayList` class or use Java's `ArrayList` class that implements Java's `List` interface. **You must also make use of Java `Iterator`s where appropriate** (i.e., whenever you need to traverse a list). You may also write additional helper classes to be used by the `EmployeeDatabase` class.

Note: in addition to the information given in the table above, your `EmployeeDatabase` class will need to be able to recognize null parameter values and, when a null parameter value is passed, throw a `java.lang.IllegalArgumentException`.

The `InteractiveDBTester` class

The `InteractiveDBTester` class creates and tests an `EmployeeDatabase` that represents information about employees and wish lists. Employee and wish list information is read from a text file (explained above) using the method `populateDB`. The class also contains a variety of methods used to test the database.

This assignment must make use of Java `Iterator`s where appropriate (i.e., whenever a list is traversed).

The `InteractiveDBTester.java` file contains the outline of the `InteractiveDBTester` class. Download this file and use it as the starting point for your `InteractiveDBTester` implementation.

The `populateDB` method of the `InteractiveDBTester` class does the following:

1. Checks whether exactly one command-line argument is given; if not, display "`Please provide input file as command-line argument`" and quit.
2. Checks whether the input file exists and is readable; if not, display "`Error: Cannot access input file`" and quit.
3. Loads the data from the input file and uses it to construct an employee database.

The `InteractiveDBTester` class also contains a number of methods of the form `pushXXX`. Each method implements a particular testing command. For example `pushList()` implements the *List* command, which lists the current contents of the employee database. Note that these methods (in final form) **do not** do any printing. Rather, they retuen a `String` representing the result of the command. The reason for this is that these testing commands will be used in a GUI-based database tester. GUIs do not use print statements. Rather, they use methods to place text within a "text box" that may be placed anywhere in a GUI interface. The text box may specify not only the text itself, but also its font, size and colour.

The testing commands we will use are:

Testing Method	Description
`String pushDiscontinue(String destination)`	If `destination` is not in the database, return "`destination not found`". Otherwise, discontinue `destination` (i.e., remove the destination from all the wish lists in which it appears) and return "`destination discontinued`".
`String pushFind(String employee)`	If `employee` is not in the database, return "`employee not found`". Otherwise, find `employee` and return the employee (on one line) in the format: `employee:destination1,destination2,destination3`
`String pushHelp()`	Provide help by displaying the list of command options. This command has already been implemented for you.
`String pushInformation()`	Return information about this database by doing the following: 1. Return a line: "Employees: *integer*, Destinations: *integer*" This is the number of employees followed by the total number of unique destinations.

	2. Return a line: "# of destinations/employee: most *integer*, least *integer*, average *decimal fraction*" where most is the largest number of destinations that any employee has in their wish list, least is the fewest, and average is the arithmetic mean number of destinations per employee rounded to the nearest tenth (e.g., 1.2 or 0.7). 3. Return a line: "# of employees/destination: most *integer*, least *integer*, average *decimal fraction*" where most is the largest number of employee wish lists in which any destination appears, least is the fewest, and average is the arithmetic mean number of employees per destination rounded to the nearest tenth (e.g., 1.2 or 0.7). 4. Return a line: "Most popular destination: *destination(s)* [*integer*]" This is the destination that shows up in the greatest number of wish lists followed by the number of wish lists containing that destination in square brackets. If there is a tie for most popular destination, display all those tying in the order they appear in the database separated by commas.
`String pushSearch(String destination)`	If `destination` is not in the database, return "`destination not found.`" Otherwise, search for `destination` and return the destination along with the employees who have that destination in their wish list (on one line) in the format: `destination:employee1,employee2,employee3`
`String pushRemove(String employee)`	If `employee` is not in the database, return "`employee not found`". Otherwise, remove `employee` and return "`employee removed`".
`String pushList()`	Return a list of the contents of the entire employee database, one employee per line in the format: `employee:destination1,destination2,destination3`
`String pushQuit()`	Quit execution of the test. This command has already been implemented for you.

A Graphical Test Interface

Graphical User interfaces (GUIs) are widely used to interact with programs. They easy to use and need little documentation. A GUI testing interface for our employee database is shown below. This tester is implemented in class `GUItester` (see `GUItester.java`). You need not change anything in this class. `GUItester` is a subclass of `InteractiveDBTester`. It inherits your definition of the `pushXXX` commands and the `populateDB` method. You start execution with the `GUItester` class. It calls `populateDB` to read in the initial employee database. It then creates the GUI shown to the below. When you click on a button, the corresponding `pushXXX` method is called. Input parameters are typed in the text box to the right of the command button. Output is displayed in the large text box at the bottom.

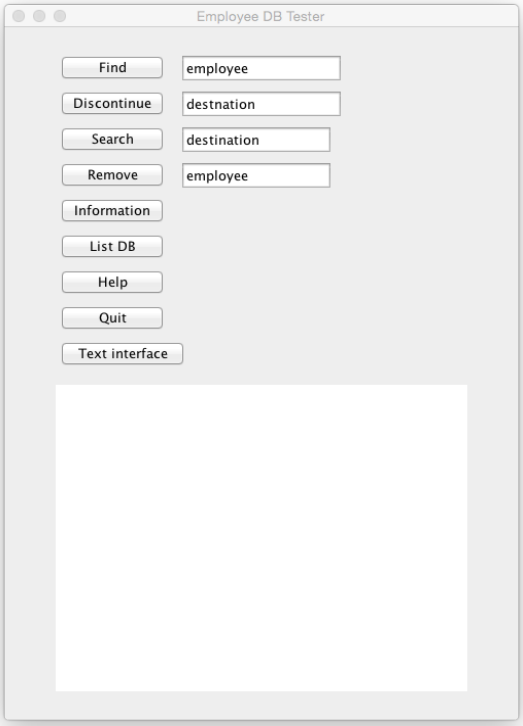
Thus if you click the "List DB" button, the `pushList()` method is called. It determines the employee database's contents, and places the information in the large text box at the bottom. To terminate testing, you simply click on "Quit."

Command-line arguments

Recall that in Java, when you run a program, it is a `main` method that runs. The `main` method always has the following header:

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

The array-of-`Strings` parameter, `args`, contains the *command-line arguments* that are specified when the program is run. If the program is run from a command prompt (i.e., not from a programming environment like Eclipse), the command-line arguments are simply typed after the name of the class whose `main` method is to be executed. For example:




```
java GUItester  employeeData.txt
```

runs the Java interpreter on the `main` method of the `GUItester` class, passing the string "`employeeData.txt`" as the command-line argument.

`main` passes the file name to `populateDB` which reads the file and creates the employee database to be tested.

To use command-line arguments when you run a Java program using Eclipse:

1. Right click on the source file that contains the main class you want to run in the "Package Explorer" window.
2. Select "Run As" from the pop-up menu.
3. Select "Run Configurations..." from the pop-up menu, which brings up the "Run Configurations" window.
4. Click on the "(x)= Arguments" tab.
5. Enter the arguments in the "Program arguments:" text box.
6. Click either the "Run" button or the "Apply" and "Run" buttons.

After the command line file is set, you can run your program by simply right-clicking "Run" on the source file you want to run. If you want to use a different file to create your database, you simply redo the above steps.

A Text-based Test Interface

GUI interfaces, while popular, can be problematic. Quality assurance teams must thoroughly test a program before it is released. This may involve hundreds or thousands of tests. Running so many tests by hand is tedious and error-prone. But how do we automate a mouse-click?

One approach is to represnt testing commands as text, with one command per line. Thus rather than entering the name Waldo in a text box and clicking the Find button, we simply enter the line:

```
find Waldo
```

Each testing method is assignment a command name:

Testing Method	Text Command
String pushDiscontinue(String destination)	discontinue destination
String pushFind(String employee)	find employee
String pushHelp()	help
String pushInformation()	information
String pushSearch(String destination)	search destination
String pushRemove(String employee)	remove employee
String pushList()	list
`String pushQuit()`	quit

Each text command can be abbreviated with its first character. Also, case is insignificant. Thus `quit, QUIT, Quit, Q` and `q` all terminate testing. The command `gui` is also implemented. It changes testing from the text based interface to the GUI interface. (The GUI tester has a button "Text interface" that changes testing from the GUI interface to the text interface.)

One of the advantages of the text interface is that testing commands can be read from a file rather than entered by hand, one by one. Similarly, the output of a testing session can be stored in a file rather than displayed on the console screen. Thus a quality assurance tester can store a standard set of test commands in

a file. The output is saved in a file and then compared against an "expected results" file. This makes thorough testing much more convenient. (This is how we will test your solution to this project.)

At the command line level, adding file names for input and output data is quite simple:

```
java Texttester  employeeData.txt  < myInput > myOutput
```

`Texttester` is run with `employeeData.txt` as the command line argument. Input is read from `myInput` and output is written to `myOutput`.

In Eclipse (version 4.5 or later):

1. Right click on the source file that contains the main class you want to run in the "Package Explorer" window.
2. Select "Run As" from the pop-up menu.
3. Select "Run Configurations..." from the pop-up menu, which brings up the "Run Configurations" window.
4. Click on the "Common" tab.
5. Enter your input or output file in the text box at the bottom right. (Click "Workspace..." to access files within your project's workspace.)
6. Click either the "Run" button or the "Apply" and "Run" buttons.

Accessibility Issues

Unfortunately, not everyone can click a mouse or read a display. It is important to design interfaces that allow the fullest range of individuals to use an application. **Screen readers** convert characters on a screen into speech (or Braille). **Voice recognition systems** (like Apple's Siri or Amazon's Alexa) convert speech into text. These tools could be used with our text interface to enhance accessibility to our employee database. For example, a user could issues voice commands that are translated into text and fed to the employee database as characters in `System.in`. Characters written to `System.out` would be translated into voice form. A lot of development work would be needed, but such an enhanced interface is clearly feasible.

How to proceed

After you have read this program specification and given thought to the program design we suggest the following steps:

1. Review these style and commenting standards that are used to evaluate your program's style.
2. You may use the Java programming environment of your choice in CS 367, at level 5 or above. However, all programs must compile and run using the Java 8 SE for grading. We recommend that you use Eclipse. You may want to review the Eclipse tutorial to learn the basics.

3. Download the following files to your programming assignment 1 directory:
    - `Employee.java`
    - `InteractiveDBTester.java`
    - `GUItester.java`
    - `Texttester.java`

4. Implement and thoroughly test your `EmployeeDatabase` class and any additional supporting classes.
5. Incrementally implement the `InteractiveDBTester` class as specified above. This involves implementing the `populateDB` method and the various `pushXXX` methods. Test each component to ensure your program is working correctly before implementing the next component. Create small text files for testing (it will be easier to debug than larger ones) and make sure you test all the boundary and negative cases as well as the positive cases. Try this sample employee data base file with this command file. Your program output should match this file exactly. Once you've got your program working on small text files, create larger text files and make sure your program works on those as well. You may use either the GUI or text interface for your testing. (You should try both to verify that each works properly.)
6. Submit your work for grading.

Handing in

What should be handed in?

Make sure your code follows the style and commenting standards used in CS 367.

Electronically submit the following files to the Program 1 tab on Canvas:

- "`InteractiveDBTester.java`" containing your employee database testing tools,
- "`EmployeeDatabase.java`" containing your implementation of the `EmployeeDatabase` class, and
- "`*.java`" additional classes (if any) that you've implemented for your program.

Please turn in only the files named above. Extra files clutter up the "handin" directories.