Array and Classes - Zipcode lookup

CS 151 - Introduction to Data Structures

Assignment 1 - due Friday 1/27

In this project, we will practice file input, class design and arrays. Note that all programming assignments will go through auto-testing for correctness and thus it is important that you pay attention to naming conventions. In other words, name your classes, methods and directories EXACTLY as given here, including cases.

All programming assignments in this class will follow the convention of naming the class that contains the `main` method `Main`. Thus the corresponding file should be named `Main.java`. For now, you can keep all of your other classes in the same `Main.java` file. Because Java allows only one `public` class per file, it is okay to leave out the access modifier for all your classes - i.e. instead of `public class Place` just write `class Place`. This will change in subsequent assignments and more instructions will be given then.

In addition, please read the `program design principles` and `code formatting standards` carefully. You are expected to adhere to all stated standards. Violations will result in deductions.

1 Input File Format

All data files are found in `~dxu/handouts/cs151/data`. Look under the appropriate subdirectory. For example, for this assignment (#1), look under `a1`.

The file `uszipcodes.csv` contains all zipcodes used in the United States. Here are the details of the data file's format:

The first line is a special line, giving some basic info about the file, in the following comma-separated values:

```
<num>,zip,city,state,population,males,females,
```

where the first field in the line is an integer giving you the number of zip codes stored in the file. The rest of the line contains column headers for the file. You will ignore the rest of this line in this assignment.

The rest of the lines come in the following format:

`<zip>,<town name>,<state code>,<population>,<males>,<females>,`

where the comma-separated fields have the following meanings:

`zip`	the 5-digit zipcode
`town name`	name of the town with the zipcode
`state code`	2-character encoding of the state name
`population`	population in this zipcode, an integer
`males`	number of males in this zipcode, an integer
`females`	number of females in this zipcode, an integer

Sample snippet:

```
42613,zip,city,state,population,males,females,
00501,Holtsville,NY,,,,
00544,Holtsville,NY,,,,
00601,Adjuntas,PR,18570,9078,9492,
00602,Aguada,PR,41520,20396,21124,
00603,Aguadilla,PR,54689,26597,28092,
...
```

As you can see from the above, there are 42,613 entries following the first line. Zipcode 00501 belongs to the town of Holtsville, NY, for which we have no population recorded.
In this assignment, you will ignore the population numbers, and store only the zip code, the town and the state. Note however that there are towns whose names have more than one word, such as "Palm Springs".

2 Specific Tasks

1. Create a class called `Place` to model each zipcode to contain the following data fields: zipcode, town, state. In order to support data encapsulation, these fields should be labeled private. You will thus need a constructor and several accessor methods in the `Place` class to set up the fields and to access them.

2. Make sure to include a constructor in your `Place` class according to this signature:

```
/** Creates a Place with the given zip, town name, and
 *  state
 *  @param zip The 5-digit zip code
 *  @param town The town name
 *  @param state The state abbreviation
 */
public Place(String zip, String town, String state)
```

3. Override `toString` of the `Place` class so that `System.out.print(ln)` will generate the expected output (see 5) when called on an object of `Place` type.

4. Write a separate class `LookupZip` that will contain several public static methods to implement the main part of the assignment with the following signatures:

```
/** Parses one line of input by creating a Place that
 *  denotes the information in the given line
 *  @param lineNumber The line number of this line
 *  @param line One line from the zipcodes file
 *  @return A Place that contains the relevant information
 *  (zip code, town, state) from that line
 */
public static Place parseLine(int lineNumber, String line)

/** Reads a zipcodes file, parsing every line
 *  @param filename The name of the zipcodes file
 *  @return The array of Places representing all the
 *  data in the file.
 */
public static Place[] readZipCodes(String filename)
  throws FileNotFoundException

/** Find a Place with a given zip code
 *  @param zip The zip code (as a String) to look up
 *  @return A place that matches the given zip code,
 *  or null if no such place exists.
 */
public static Place lookupZip(Place[] places, String zip)
```

Implement the class `LookupZip` and the above methods.

5. Write a `main` method that ties it all together in a class called `Main`. Your program should continuously prompt the user for a zipcode untill the user enters `00000` to quit.

   Here's a sample session:

```
zipcode: 19010
Bryn Mawr, PA

zipcode: 99400
No such zipcode

zipcode: 91729
Rancho Cucamonga, CA

zipcode: 00000
Good Bye!
```

3 Testing

A reminder that your program will undergo auto-testing, thus it is important that you stick to the output format EXACTLY. That means exactly the same amount of white spaces, exactly the same punctuations and exactly the same error message. Below are a few tips on how to check for correctness yourself:

1. A smaller input file (of the same format as `uszipcodes.csv`), named `testZip.csv`, is provided for you as well, in case you want to debug on a smaller dataset.

2. Check your output (manually) against the sample output given in 2.5 above.

3. Test some more zipcodes of your choice (your hometown, some other place you know, or just select a few random lines from the data file).

4. In the subfolder `tests/a1`, there are two files `in.txt` and `out.txt`, provided for your testing purposes. `in.txt` contains 17 test inputs and `out.txt` contains the expected output if you ran your program with `in.txt`. Of course, you are not expected to type each input into your program by hand. You should use the command-line redirection feature of the Linux shell, which re-directs terminal I/O to files, like this:
   `java Main < in.txt > my-out.txt`

`<` specifies replacing terminal input with the named file
`>` specifies placing terminal output into the named file

The Linux command `diff` can be used to compare two files for differences. If your program works correctly, `diff my-out.txt out.txt` should return nothing.

4 Electronic Submissions

Your submission will be handed in using the `submit` script provided on our Linux system. Note that your program will be graded based on how it runs on the department's Linux server, not how it runs on your PC, thus it is highly recommended that you ssh into the server and test even if you choose to program on your own laptop.
At this point, you probably only have one source file `Main.java`. The submission should then include the following items:

1. **README:** There must be a plain text file called `README`, which contains any helpful information on how to compile and run your program. This includes:

   Your name:

   How to compile: for now should just be `javac Main.java`

   How to run it: for now should just be `java Main`

   Known Bugs and Limitations: List any known bugs, deficiencies, or limitations with respect to the project specifications. Documented bugs will receive less deduction versus uncaught ones.

2. **Source files:** `Main.java`

3. **Data files used:** `uszipcodes.csv`

DO NOT INCLUDE: Please delete all executable bytecode (`.class`) files prior to submission.

To submit, store everything (README, source files and data files) in a directory called `A1`. Then follow the directions here:
`https://cs.brynmawr.edu/systems/submit_assignments.html`