Project #1

Computer Science 2334
Spring 2011

User Request:

"Create a simple place data system."

Milestones:

1.	Use program arguments to specify a file name.	*10 points*
2.	Use simple File I/O to read a file.	*10 points*
3.	Create an abstract data type (ADT) to store information on a single place.	*15 points*
4.	Create an ADT that *abstracts* the use of an array of places (i.e., a list of places).	*15 points*
5.	Implement a program that allows the user to search the place list as described below.	*20 points*
►	Develop and use a proper design.	*15 points*
►	Use proper documentation and formatting.	*15 points*

Description:

For this project, you will put together several techniques and concepts learned in CS 1323 and some new techniques to make an application that searches a large collection of data on places (such as cities and towns) which we will call an "place data system" or "PDS[1]." This project creates a simple program that allows users to enter the names of places and see the population of each place in particular years.

One of the best things about this project is that it will use lots of data. Your program must be capable of handling data on over 10,000 places. To the surprise of no one, the best approach to this somewhat large problem is to decompose the problem into separate classes that can be gradually built up. Note that much of the code you write for this program could be reused in more complex applications, such as a full-fledged census data system that also includes information on income, housing, and so forth.

Operational Issues:

Your program will read the place data file (a text file) as specified by a file name. The file name will be given as a program argument. (See below for information on how to read program arguments). Each line of the file contains a place name, followed by a comma, followed by a space, followed by a state abbreviation (or similar), followed by a comma, followed by a space, followed by a population, followed by a comma, followed by a year. For example:

```
Oklahoma City, OK, 1,227,278, 2009
```

You will need to store each place name and its associated data as a single object and the collection of all places will be stored as a list of these objects.[2]

Once the place list has been read into your program and stored, your program will use a **JOptionPane** to display to the user a dialog box requesting an input name.

1. Not to be confused with a particularly dangerous situation (PDS) watch issued by the Storm Prediction Center.
2. If you are wondering about the population given, it is for the "metropolitan statistical area," not the city proper.

When the user enters a name into the dialog, your program will search for it in the place list. If the name is in the list, your program will use another dialog to inform the user of that fact along with that place's data. If the name appears in the list multiple times, all of the data associated with that name will be displayed. If the name is *not* in the list, your program will use a dialog to inform the user of *that* fact.

After checking whether the name is in the place list and informing the user one way or the other, your program will again use a dialog to request a place name. It will continue in this loop until the user clicks on cancel, at which time the program should gracefully exit.

Implementation Issues:

There are two Java elements in this project that may be new to some students: reading from a file and program arguments. These Java features are summarized below.

Reading from a file:

We will discuss File I/O in more depth later in the class, this project is just designed to give you a brief introduction to the technique. Reading files is accomplished in Java using a collection of classes in the **java.io** package. To use the classes you must import the following package:

```
import java.io.IOException;
```

The first action is to open the file. This associates a variable in the program with the name of the file sitting on the disk.

```
String fileName = "PlaceList.txt";
FileReader fr = new FileReader(fileName);
```

Next the **FileReader** is wrapped with a **BufferedReader**. A **BufferedReader** is more efficient than a **FileReader** since a **BufferedReader** saves groups of characters during a single operation instead of working with characters individually. Another advantage of using a **BufferedReader** is that there is a command to read an entire line of the file, instead of a single character at a time. This feature comes in particularly handy for this project.

```
BufferedReader br = new BufferedReader(fr);
```

The **BufferedReader** can now read in Strings.

```
String nextline;
nextline = br.readLine();
```

Look at the Java API listing for **BufferedReader** and find out what `readLine()` returns when it encounters the end of the file (stream). When you are finished with the **BufferedReader**, the file should be closed. This informs the operating system that you're finished using the file.

```
br.close();
```

Closing the **BufferedReader** also closes the **FileReader**.

Any method which performs I/O will have to throw or catch an **IOException**. If it is not caught, then it must be passed to the the calling method. The syntax is given below:

```
public void myMethod(int argument) throws IOException {
        //method body here
}
```

Program Arguments:

Sometimes it is handy to be able to give a program some input when it first starts executing. Program arguments can fulfill this need. Program arguments in Eclipse are equivalent to MS-DOS or Unix command line arguments. Program arguments are handled in Java using a String array that is traditionally called `args` (the name is actually irrelevant). See the slides from Lab #2 for how to supply program arguments in Eclipse.

The program below will print out the program arguments.

```java
public static void main(String[] args) {
        System.out.println(args.length + " program arguments:");
        for (int i=0; i< args.length; i++)
                System.out.println("args[" + i + "] = " + args[i]);
}
```

Milestones:

A milestone is a "significant point in development." In other words, milestones serve to guide you in the development of your project. Listed below are a set of milestones for this project along with a brief description of each.

Milestone 1. Use program arguments to specify a file name.

The name of the file that stores the list of place data will be passed to the program using program arguments as discussed above. Type in the sample program given in the section on program arguments and make sure that you understand how the program arguments you provide affect the `String[] args` parameter that is passed into the main method of the program. Then, write a main method for your program that reads in the name of the data file from the program arguments.

Milestone 2. Use simple File I/O to read a file.

Before you can allow the user to search the place list, you must first be able to read a text file. Examine the section above on reading from a file. A good start to the program is to be able to read in the name of a file from the program arguments, read each line from the file, one at a time, and print each line to the console using `System.out.println()`. Later, you will want to remove the code that prints out each line read in from the file, since the project requirements do not specify that the file is to be written out to the console as it is read.

Milestone 3. Create an abstract data type (ADT) to store data on a single place.

You must create a class that holds the place data for a single place in the data file before you can store that data. Think about what data is associated with each place and how to most efficiently store the data. Also, think about any methods that may help you to manage and compare the data by abstracting operations to be performed on individual entries in the list. Such methods may be used by other classes.

Milestone 4. Create an ADT that abstracts the use of a list of place data.

You are to store the object representing each place into a list of objects. However, it is not necessary for the portions of the program that will carry out user actions to directly operate on this list as they would if you simply used an array of place objects. Instead, you should create a class that abstracts and encapsulates this list and allows for the addition of new places and also supports the required search operations on it.

This class will represent the collection of information associated with the program. Think about the operations that this class needs to support and how it will use the ADT created for Milestone 3. At this point, you should be able to read in the input file and create an object for each place in the file, and store that object into the list. Note that the data file used for grading may be larger (or smaller) than the data file provided for testing.

Milestone 5. Implement a program that allows the user to search the place list as described below.

This is where the entire program starts to take on its final form and come together. Here you will create the input and output dialogs and the menu system. Start by creating the input dialogs and the output dialogs. Tie together the input dialogs, the ADT from Milestone 4, and the output dialogs to make this search functional and test its functionality.

Finally, you are ready to create the main loop of the program that will take input and invoke the correct methods to create appropriate output.

Remember that when the user clicks on "cancel," the program must gracefully exit. This can be accomplished by using `System.exit(0)`.

How to Complete this Project:

Preliminary Design:

1 During the lab session and in the week following, you should work with your partner(s) to determine the classes, variables, and methods needed for this project and their relationship to one another. This will be your preliminary design for your software.

1.1 Be sure to look for nouns in the project description. More important nouns describing the items of interest to the "customer" should probably be incorporated into your project as classes and objects of those classes. Less important nouns should probably be incorporated as variables of the classes/objects just described.

1.2 Be sure to look for verbs in the project description. Verbs describing behaviors of the desired objects and the systems as a whole should probably be incorporated into your project as methods.

1.3 Be sure to use UML class diagrams as tools to help you with the design process.

2 Once you have completed your UML design, create Java "stub code" for the classes and methods specified in your design. Stub code is the translation of UML class diagrams into code. It will contain code elements for class, variable, and method names; relationships between classes such as inheritance, composition, and aggregation as appropriate; variable types; and method signatures. Stub code does not, however, contain method bodies. Because we are using object-oriented design, in which the inner workings of one class are largely irrelevant to the classes with which it interfaces (that is, we are using encapsulation), we do not need to complete the implementation of the classes until the design is completed.

3 Add comments to your stubbed code as specified in the documentation requirements posted on the class website. Run your commented stubbed code through Javadoc as described in the Lab #2 slides. This will create a set of HTML files in a directory named "docs" under your project directory.

4 At the end of the first week, you will turn in your preliminary design documents (see ***Due Dates and Notes***, below), which the TA will grade and return to you with helpful feedback on your preliminary design. **Please note:** You are encouraged to work with the instructor and the TAs during their office hours during the design week to get feedback throughout the design process as needed.

Final Design and Completed Project

5 Using feedback from the instructor and TAs as well as your own (continually improving) understanding of OO design, revise your preliminary UML design.

6 Make corresponding changes to your stub code, including its comments.

7 Implement the design you have developed by coding each method you have defined. A good approach to the implementation of your project is to follow the project's milestones in the order they have been supplied. If you find that your design does not allow for the implementation of all methods, repeat steps 5 and 6.

8 Test your program and fix any bugs.

9 Once you have completed the project and are ready to submit it for grading, create a new set of Javadoc files using Eclipse and inspect them to make sure your final design is properly documented in your source code.

Extra Credit Features:

You may extend this project with more search features for an extra 5 points of credit. Think of ways to enable a wider range of searches to be used, such as searching based on state or year or population range or regular expression or wild cards. Alternatively, think of ways to decompose the class for place data into logical subclasses. You could also revise user interface elements. If you revise the user interface, you **must** still read the file name from the program arguments and the place list from the text file.

To receive the full five points of extra credit, your extended feature must be novel (unique) and it must involve effort in the design and integration of the feature into the project and the actual coding of the feature. Also, you must indicate, on your final UML design, the portions of the design that support the extra feature(s); and you must include a write-up of the feature(s) in your milestones file. The write-up must indicate what the feature is, how it works, how it is unique, and the write-up must cite any outside resources used.

Due Dates and Notes:

The electronic copy of your preliminary design (UML, stub code, and detailed Javadoc documentation) is due on **Wednesday, February 9th**. Submit the project archive following the steps given in the submission instructions **by 9:00pm**. Submit your revised UML design on *engineering paper* or a hardcopy using UML layout software and a hardcopy of the stubbed source code at the **beginning of lab on Thursday, February 10th**.

The electronic copy of the final version of the project is due on **Wednesday, February 16th**. Submit the project archive following the steps given in the submission instructions **by 9:00pm**. Submit your final UML design on *engineering paper* or a hardcopy using UML layout software and a hardcopy of the source code at the **beginning of lab on Thursday, February 17th**.

You are not allowed to use the **StringTokenizer** class. Instead you must use `String.split()` and a regular expression that specifies the delimiters you wish to use to "tokenize" or split each line of the file.

You may write your program from scratch or may start from programs for which the source code is freely available on the web or through other sources (such as friends or student organizations). If you do not start from scratch, you must give a complete and accurate accounting of where all of your code came from and indicate which parts are original or changed, and which you got from which other source. Failure to give credit where credit is due is academic fraud and will be dealt with accordingly.

ADTs

Do not be confused by the term “abstract data type” (ADT). An ADT is **not** the same as an abstract class, even though they both contain the word “abstract” in them.

A data type is simply a description of how bits in a computer are grouped and interpreted Maybe one set of 32 bits is interpreted as a character, whereas another set of 32 bits is interpreted as an integer, and a set of 64 other bits is interpreted as an integer that can hold larger magnitudes, etc. With *concrete* data types, implementation details matter, such as the number of bits, whether the bits are ordered from least to most significant, etc. If you try to mix implementations, you’ll screw things up.

With an abstract data type, you hide the implementation details of the data type from the user, so that what matters is how one *interacts* with instances of the type, not how they are *implemented* internally. So, if you add two integers whose internal representations differ, you should still get a sensible result.

This means that if you create a class using object-oriented techniques (such as making variables private and only accessible through methods, etc.), then even a *concrete* class is an *abstract* data type.

The reason this description doesn't just tell you to create a class to store a place’s data is because you don't have to use just one class. You could use two classes, or three, or more. You could arrange them in an inheritance hierarchy (where one is a subclass of another). You could use composition or aggregation (the types of has-a links we have discussed). All of these classes could be concrete or some of them could be concrete and some could be abstract. You could also include interfaces, if you saw a good reason to do so. All of these alternatives would count as creating an ADT.

If the term 'ADT' is still confusing you, think of the assignment as saying “Create something appropriate that the computer can use to store place data.” This is what it means.