Project 1
Lost and Found

Due Date/Time: June 6th at 11:45 pm

This assignment requires you to create a complete Java program consisting of 5 source code files and two JUnit test case files. You must electronically submit your source files by the due date. Directions for using submit are at the end of this page.

Requirements

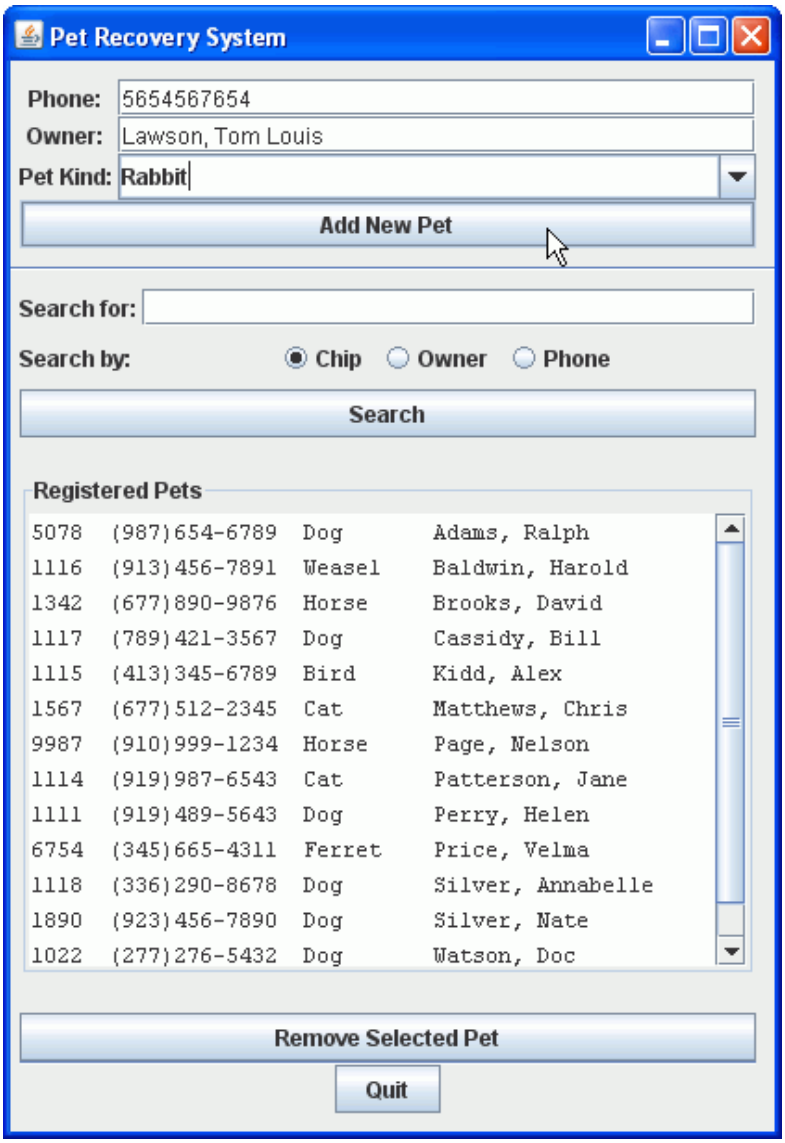
Companion Animal Recovery is a non-profit affiliate of the American Kennel Club with a mission of recovering lost pets. CAR is located in Raleigh, but it services pet owners from all over the world.

CAR has a database of information on more than 2.5 million owners and their pets. Each pet enrolled in the database has a collar tag, tattoo, or microchip for identification. Since microchips are permanent (they do not tear off and they do not fade), CAR encourages pet owners to microchip their pets and enroll them in their database. If an animal shelter or vet scans a lost pet and calls CAR with the microchip string, CAR will contact the owner. Microchip strings are unique (no two are the same).

You must write a program that is a simple prototype of CAR's database management software to illustrate the pet registration and recovery process.

Program functionality

We have created a graphical user interface (the view) for your program, which is shown here.



The GUI window has several components:

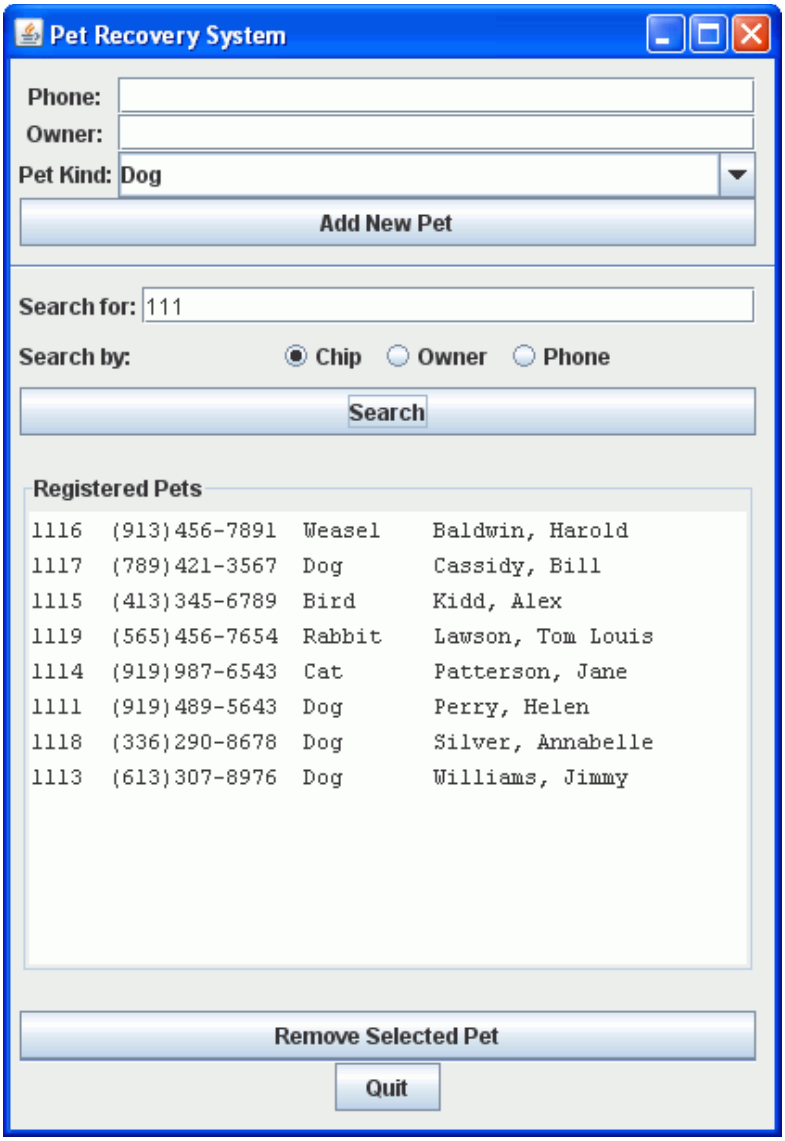
- Text fields, combo box, and a button adding a pet record to the database. The user must enter the phone number and name of the owner and the kind of pet. Both the number and name are simply text, though the convention is to add the name in the form <last name>, <first name>. If the desired kind is not on the dropdown list (combo box), the user can type the kind in the list before adding.
- Text field, radio buttons, and button for searching for all pets matching a particular criteria. The user must enter a search string and select which field to search in (chip, owner, or phone). When the user clicks Search, the Registered Pets list displays all pet records in which the string entered is the prefix for the value of the selected field. If the string entered is blank, then all records are shown.
- A button for removing a pet record. The user must select the pet from the Registered Pet list then click the button to remove.
- A quit button. When the user clicks this button, the pet record data is written to a text file and the program quits execution.

Displaying the registered pets data

The Registered Pet list is shown when the program begins execution and whenever any button except Quit is clicked. The list displays all of the fields for each pet record in the database. The fields are aligned neatly in columns. The pet records are in alphabetic order according to owner name. The ordering is case insensitive. A single pet record contains the following information:

- Chip. A string of maximum length 10. Chips are in the first column.
- Phone. A string in the standard phone number form (xxx)yyy-zzzz.
- Owner name. Any string.
- Pet kind. A string of maximum length 8.

The following picture is a snapshot of the program after the user searched for all records where the chip begins with 111.



Getting initial data and saving final data

When your program starts execution, it must attempt to read the data for the database from a text file. Each line of the text file corresponds to a registration record, where the individual pieces of data for the record are separated by tab characters. The format of the line is as follows:

<pet kind><tab><chip><tab><phone><tab><owner>

Each piece of data is a string (there is no numeric data). The data on any line may be defective in any of the following ways:

- Duplicate microchips. If a chip duplicates an existing one, then it must be replaced by a new, unique string.
- Incorrect number of digits or inappropriate characters in a telephone number. There should be 10 digits in every number. If there are more than 10, the 11th digit and beyond are ignored. If there are too few, the remaining digits are filled in with 1's. Telephone numbers should be formatted as: (xxx)yyy-zzzz, but the parentheses and dash may be missing. Any extraneous characters (non-digits) should be ignored.
- Too many characters in the pet kind. There should be no more than 8 characters, and the first one cannot be a whitespace character.

If the input text file cannot be found, the database is considered initially empty.

When your program quits execution, it must write the database contents to a separate text file, formatted the same as the input file (or the same as the input described above if there is no input file). The name of the new text file must be the same as the original, with "-1" appended to the name immediately before the .suffix. For example:

Original file name	New file name
database.txt	database-1.txt

petinfo	petinfo-1
my.info.txt	my.info-1.txt
<no input file found>	pets-1.txt

Design

Your program must consist of at least five different classes:

- `Pet` represents a single pet registration.
- `PetData` represents the collection all pets. This is the "database," and it implements `IDataCollection`.
- `RecoveryGUI` is the GUI that we provide. It contains the `main()` method that starts program execution.
- `IDataCollection`, is an interface that we provide. It describes some of the public methods that are required in `PetData`.
- `ChipFactory` is the class that registers old chips (from the input file) and creates new ones. We provide this as well.

The `Pet`, `PetData` , and `IDataCollection` , and `ChipFactory` classes serve as the "model" for the program while the `RecoveryGUI` class provides the "view" and "controller." You can define other additional classes if you wish.

> **Important:** Do not change any of the code in `RecoveryGUI`, `IDataCollection`, or `ChipFactory`. Your project must use these classes as intended by the assignment.

Pet design

The `Pet` class should contain data members for the data described above. It should also have getters for that data but no setters.

PetData design

The `PetData` class must contain an array-based list `Pets` maintained in increasing order of names (ignoring case). Limit the size of the array to 500 `Pets`, and follow the array-based strategy shown in the course notes. Do *not* use the `ArrayList` class or any other list classes provided by the Java API.

Your `PetData` class must have a null constructor (that starts with an empty list) and another constructor `String` parameter, which is the name of a text file with lines representing `Pet` records. `PetData` must also implement the methods declared in `IDataCollection`.

- `String search(String prefix, String field)`. Searches for `Pets` in the database (array) according to the search criteria (a field and a string which is a prefix for the value of the field). The method should return a `String` consisting of lines, one per `Pet`. Each `Pet` in the `String` satisfies the search criteria.
- `void add(String name, String phone, String kind)`. Enrolls a `Pet` in the database with the given owner, phone, and kind (cat, dog, etc).
- `void remove(String value)`. Removes a `Pet` from the database. The chip of the `Pet` removed should match the first token in the `String` parameter.
- `void saveToFile()`. Saves the contents of the database to a text file.

To describe the search string fields for `RecoveryGUI`, `PetData` must declare a public final static array of Strings, like this:

```
public final static String[] SEARCH_FIELD = { "Chip", "Owner", "Phone" };
```

In addition to the methods declared in `IDataCollection`, `PetData` must have one more method, which is not called by `RecoveryGUI` but which you will use for unit testing:

`public Pet petAt(int k)`. Returns the `Pet` at position `k` in the list.

ChipFactory, IDataCollection, and RecoveryGUI

You do should read the code that we provide. All three Java files have interesting features.

`ChipFactory` has only static methods and it cannot be instantiated because there is no reason to do so. `ChipFactory` has one purpose, to generate new chips and to register existing chips. Both of its public methods are static.

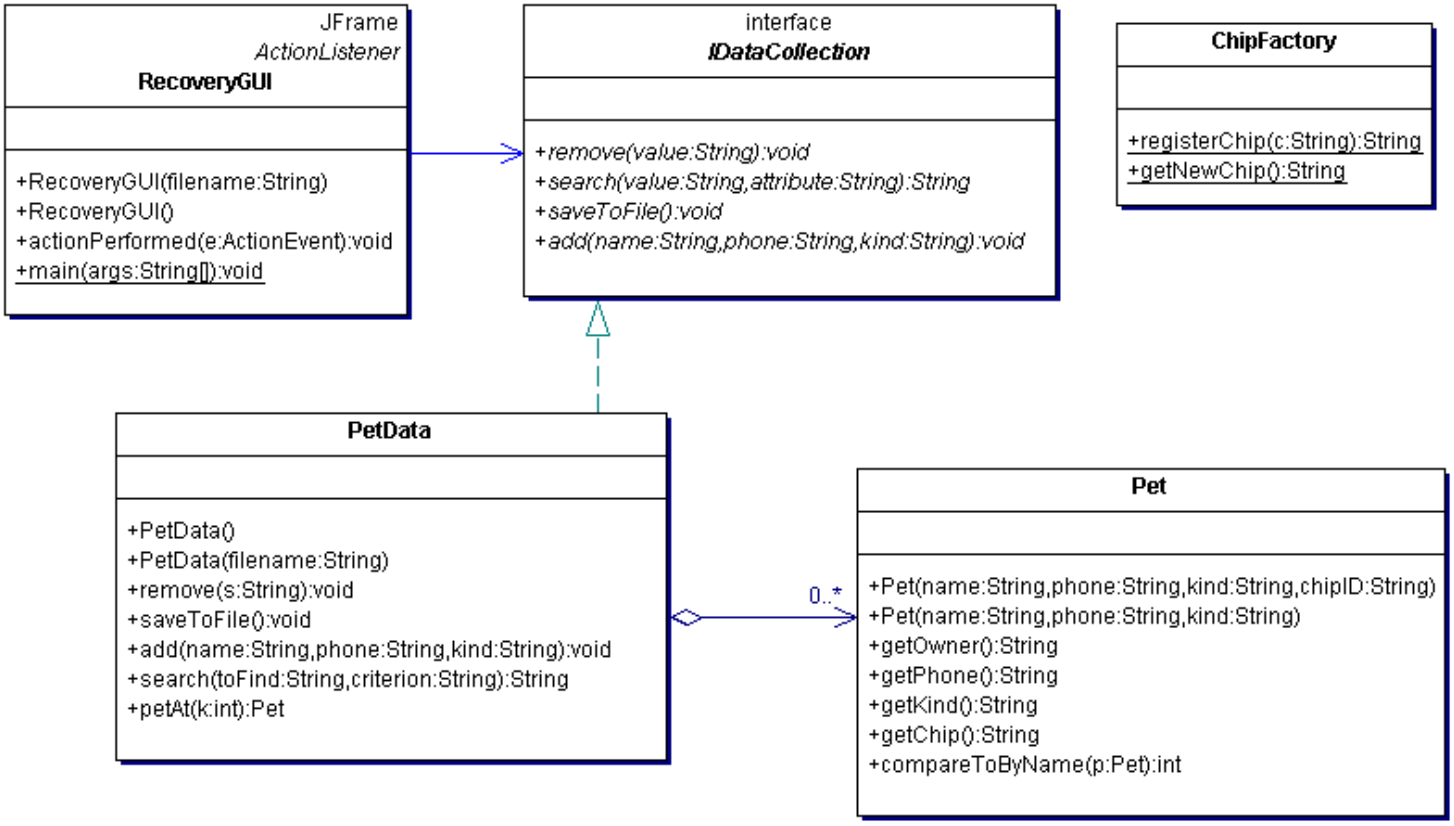
`IDataCollection` is an interface. An interface dictates what behaviors any implementing class *must* provide. It's essentially a guarantee -- a class that implements a particular interface is guaranteed to have at least public methods

declared in the interface. Implementing classes can have additional behaviors as well.

`RecoveryGUI` is the user interface for this project. The final attribute declared in `RecoveryGUI` is the access to the model. The type of that attribute is declared to be `IDataCollection` but instantiated as a `PetData` object.

UML diagram

The following UML diagram shows the relationships among the classes in the instructor solution. The data and the private methods are hidden in this diagram.



Implementation

All of your classes must be in a package named `csc216project1`.

In order to get a working version of your program right away, you can write skeleton versions of the `Pet` and `PetData` classes, stubbing out the methods until you have written their actual code.

> ***Writing stubs:*** It is a great idea to test your code as you write it. One good technique is to write stubs for each method that you know must be in the class. Even though the results won't be correct, writing stubs gives you a working program right away.
>
> A stub is a dummy method with these characteristics:
>
> - The stub must have the same signature as the actual method. This includes the same access (public/private), return type, method name, and formal parameter list.
> - If the method is void, the stub body can be empty (no statements between the braces).
> - If the method returns a value, the stub should have a return statement. The value can be any appropriate default for the type (0, null, and so on).

Pet

The `Pet` class should have two constructors. Both need parameters for the owner, phone, and kind. One needs an additional parameter for the chip.

The `Pet` class works in conjunction with `ChipFactory`, which has two static methods for generating chip numbers. The method `getNewChip()` generates a new chip that is guaranteed not to match any existing chip. The method

`registerChip()` either records the chip sent as the parameter or, if that chip matches one that already exists, the method generates a new unique chip.

The `Pet` class in the instructor solution has a method that compares two `Pet` objects by name. This enables the `PetData` class to sort the array of `Pets` appropriately. You should make use of the built-in `String` methods (look at the Java API for details on these methods). Some methods that we found useful include:

- `trim()`. Removes leading and trailing whitespace.
- `compareToIgnoreCase(String str)`. Compares two strings while ignoring any difference between upper- and lower-case letters. For example, "cAt", "CAT", and "cat" would be considered the same.
- `startsWith(String prefix)`. Indicates whether the string begins with the given prefix. For example, `"January".startsWith("Ja")` is `true` and `"(919)234-5678".startsWith("9")` is `false`.

Our `Pet` class also has private method that formats a phone number properly. It takes only the digits and creates a `String` for the phone attribute using parentheses, a dash, and the first 10 digits.

PetData

Work on the methods of the `PetData` class *after* you have successfully completed *and tested* the `Pet` class. The class declaration for `PetData` must be as follows:

```
public class PetData implements IDataCollection
```

`PetData` must maintain an array-based list of `Pets`. That is, the `Pets` must be elements of an array, where the type of the array is `Pet[]`. Since this class must know how to name the output file correctly, it should have a data member to represent the output file name.

`PetData` has two constructors:

1. The null constructor makes a new `PetData` instance with an empty list.
2. The other constructor should have a `String` parameter representing the name of the input file. It reads data from a file to initialize the list items.

When `search()` is called with an empty string for the first actual parameter, it should return a string of all `Pets` in the database. Otherwise, the return string contains only the matching `Pets`. In either case, the string returned must have newlines between successive `Pets`, and the first token for each line should be the chip for the corresponding `Pet`. Attributes in the return string should be formatted neatly so they will print out in columns when using a fixed-width font. [Note: when `RecoveryGUI` starts executing, it calls `search()` with an empty string as the first parameter to populate its registration list.]

The parameter for `remove()` is a string that was captured from the most recent search. Since the first token is a chip, `remove()` can look at chip values in its list to figure out which `Pet` to remove.

Text file input and output

The notes for the online course have helpful information on dealing with text file I/O (see Input with Java 5 and Text Files). Below are some additional ideas for your work.

The instructor solution for reading and processing a text file uses two `Scanners`. One `Scanner` reads entire lines from a file and another peels the individual `Pet` data off each line. An outline of the code is below. You should use this as your starting point, filling in more code as needed for the lines that are commented.

```
public PetData(String filename) throws FileNotFoundException {
   File f = new File(filename);
   Scanner scanner = new Scanner(f);
   // Save the file name to use it later.
   try {
      readFromFile(scanner);
   }
   finally  {
      scanner.close();
    }
}

// .....

private void readFromFile(Scanner fileScanner){
   while (fileScanner.hasNext()) {
      String line = fileScanner.nextLine();
```

```
      try {
         Scanner lineScanner = new Scanner(line);
         lineScanner.useDelimiter("\t"); // Attribute data are separated by tabs.
         // Now pick up each attribute for the Pet of this line,
         //   create a new Pet with it, and add it to the list.
      }
      catch (Exception e) {
      }
   }
      // Do any kind of cleanup (sorting, etc) you need here.
}
```

Writing `Pet` data to a file is similar to writing to standard output. The main difference is that you have to deal with creating the file before writing anything and saving the file after you finish all writing. Here is an outline of the instructor code.

```
public void saveToFile() {
   try {
      FileWriter fw = new FileWriter(textFileName);
      BufferedWriter buffer = new BufferedWriter(fw);
      PrintWriter outfile = new PrintWriter(buffer);
     // Now write data for each Pet to the file, using tab delimiters, like this:
     //    For each Pet:  outfile.println(line of data here)
     outfile.close();
   }
   catch (Exception e)   {
   }
}
```

Testing

You should create JUnit tests for `Pet` and `PetData`. Do not unit test setters or getters, but do unit test every other public method, including any complicated constructor.

Since the `Pet` class has so few public methods, you need to test most of the functionality (such as getting unique chip values and formatting phone numbers) using the constructors. You should test these before beginning substantial work on `PetData`.

Test each method of `PetData` immediately after you write it. You will need to test removing and adding `Pets` to the `PetData` list several times (test for removing from the front, the middle, the rear, and so on). Use a separate test method for each one. This is where the method `petAt()` becomes very convenient!

> ***Best Practices:*** Following best practices saves good programmers huge amounts of time and effort even though doing so may require some preliminary work.
>
> Here are four best practices that you should follow with this assignment:
>
> - Write stubs and test as you go along. We cannot emphasize this enough.
> - Make all instance variables private.
> - Document your code according to the departmental style guidelines.
> - Make all of your methods do **critical checking** on their parameters.

When you have completed the constructor for the `PetData` class, you can use our test input data file, petdata.txt, to see that the constructor reads the data correctly.

Eclipse instructions

If you are using Eclipse, you can run your program inside Eclipse. Follow these instructions:

1. Put `petdata.txt` in your project directory.
2. Select **Run** from the Eclipse main menu. A run configuration window opens.
3. From the menu on the left, select **Java Application.**
4. Click the **New** button (on the left near the top).
5. Under the Main tab, choose the Main class to start the application. Either **Browse** for **RecoveryGUI**, or type it into the Main class field.
6. Open the **Arguments** tab.

7. In the Program arguments field, type **petdata.txt**.
8. Click **Apply.**
9. Click **Run.**

The next time you want to run the program with the same input file, select **Run** from the main menu.

You can execute your program from the command line using the `petdata.txt` file as input. First move `petdata.txt` to your project directory and change to that directory. Then use this command:

```
% java csc216project1.RecoveryGUI petdata.txt
```

Of course, you can create and enter other files on the command line for testing purposes.

Deployment

For this class, deployment means submitting your work for grading. Before you submit your work, make sure your program:

1. Satisfies the style guidelines.
2. Behaves as specified in this document. You should test your code thoroughly. Be sure to know what messages should be displayed for each major scenario.
3. Satisfies the gradesheet for this assignment.

You must submit the Java source code files that you created for the backend and their JUnit test files:

- Pet.java
- PetTest.java
- PetData.java
- PetDataTest.java

We will use our own copies of `IDataCollection` , `ChipFactory`, and `RecoveryGUI` to execute your code.

You should submit your program through wolfware:

http://courses.ncsu.edu/csc216/

The name of the assignment is Project_1. Submit your source code files only (the .java files). Do not attempt to submit .class files.

The electronic submission deadline is precise. Do not be late. You should count on last minute system failures (your failures, ISP failures, or NCSU failures). You are able to make multiple submissions of the same file (the later submissions overwrite the earlier ones with the same file name). To be on the safe side, start submitting your work as soon as you have completed a substantial portion.