HW7 Polly's Pet Shop

Problem Description

Polly owns her own pet shop and wants a digital application of her store. She has called you, a software developer, to help her do this. She has already created the frontend of the application, the user interface, and wants you to finish the backend, logical side, which will power the frontend.

Grabbing a cup of coffee from the coffee machine with Polly, you learn that she wants the digital application to give her customers a way to search and sort through her pets.

Solution Description

You will need to create an animal hierarchy to organize the pets and will need to determine a way to search and sort the pets. (HINT: what allows you to compare two objects?) We will provide you with some guidance on how to design the backend of the program, but there will be a bit of creative freedom with the implementation. Working as a software developer it will be important for you to think of possible edge cases that could break your code. When providing a program like this to a user you must consider that you cannot always predict what the user will do.

Reminders:

- Use the naming conventions we have taught in class. For example a getter method for a variable called count should be named getCount()
- Be sure to use appropriate visibility modifiers for variables and methods.

1. Animal Class:

You know that each animal in the pet store has a `storeId` (whole numbers), `name`, and a `price`. Don't allow there to be an instance of the Animal class as each pet will be directly instantiated as its specific species. Be sure any constructors for this class have parameters in the same order as the variables listed above, because the PetShop class will not compile otherwise.

You will need to write two constructors for this class. One where the user can decide the store id, name, and price and a constructor only accepting the store id. In the latter constructor, the default values are name: "Buzz" and price: 222.00

Write setters and getters for all of the instance variables. These methods should be named getVariableName and setVariableName, but replacing VariableName with the actual name of the instance variable.

Animals have a NATURAL ORDERING following the ordering of the store id. If two animals have the same store id they should then be compared alphabetically using the animal's name. Implement an interface that expresses this natural ordering.

Write a toString method that contains the instance variables for the Animal. Look at the examples section and follow the same formatting of "Store ID: < insert store id >, Name: < insert name > , Price: < insert price >". Be sure to print exactly 2 decimal places for the price.

2. Dog Class:

The Dog class should inherit the methods and instance variables defined in the Animal class, but do not duplicate code. The Dog class has a natural ordering but it CAN delegate to Animal's natural ordering (Consider how that changes your implementation).

All dogs also have a variable specifying whether or not they have a `curlyTail` (true or false) and a whole number representing the `droolRate`. The dog class should have two constructors. One constructor should allow the client to set the name, price, curly tail, and drool rate. The second constructor should allow the client to set only the curly tail and drool rate. For the second constructor, set the default values for price

and name to be 50.0 and "none" respectively. All dogs should have a store id of 100. Again, be sure any constructors for this class have parameters in the same order as the variables listed above.

Write getters and setters for all of the instance variables. These methods should be named getVariableName and setVariableName, but replacing VariableName with the actual name of the instance variable.

Dogs have a natural ordering similar to that of the Animal class. Override a method in the Dog class that compares two dogs. If the Animal class determines that two dogs are equal, then sort the dogs by the descending order of drool rate. In other words, be sure you are reusing code from the parent class when possible. If the animal passed into this method is not a Dog refer to the parent's compare method.

You should define a toString method that returns the toString defined in the Animal class and additionally the dog's drool rate. You should be reusing code here! Look at the examples section, your toString must follow the same formatting. The format should be "Store ID: < insert store id > , Name: < insert name >, Price: < insert price >, Curly Tail: < insert curly tail >, Drool Rate: < insert drool rate >".

3. Cat Class:

The Cat class should inherit the methods and instance variables defined in the Animal class. Avoid duplicating code. The Cat class has a natural ordering but it CAN delegate to Animal's natural ordering (Consider how that changes your implementation).

Every cat has a variable representing the number of whole `miceCaught` and whether or not the cat `likesLasagna` (true or false). This class should have two constructors one that allows the client to choose the name, price, mice caught, and if it likes lasagna. The other constructor should just take in the mice caught and if it likes lasagna. For the second constructor, set the default values of price and name to be 30.0 and "none" respectively. All cats have a store id of 200. Be sure any constructors for this class have parameters in the same order as the variables listed above.

Write getters and setters for all of the instance variables. These methods should be named getVariableName and setVariableName, but replacing VariableName with the actual name of the instance variable.

Similar to the Dog class, the Cat class should override a method which compares two cats. If the Animal class determines that two cats are equal, then you should sort them in descending order by mice caught. Remember to reuse code from the parent class. If the animal passed in to compare is not an instance of Cat then refer to just the Animal's compare method.

You should define a toString method that returns the same string as the Animal class and additionally prints out the cat's mice caught. Be sure to reuse code! Your toString should follow the same formatting as in the Examples section. The format should be "Store ID: < insert store id >, Name: < insert name >, Price: < insert price >, Likes Lasagna: < insert likes lasagna >, Mice Caught: < mice caught >".

4. Store Class:

Please be sure this class properly implements the StoreOrganizer interface because the PetShop GUI will not compile otherwise.

This will be the class where the sorting and searching of the animals are completed. You should have a `pets` variable that will be responsible for holding all of the animals. `pets` should be an array that can ONLY hold animals and can hold any type of animal.

You should have one constructor that takes in an `int numPets` and instantiates the pets variable in such a way that it can hold `numPets` pets.

This class should implement the provided interface `StoreOrganizer`. Below we explain in more detail what each implemented method should do.

The `getPets` method will return all of the pets in the store.

There should be a method called `add(Animal a)` that allows the client to pass in an animal which will be added to the pets variable. Be sure that this method does not allow any pet to be added if the array is full.

The `sort()` method should sort the animals in the pets variable by using the bubble sort method. Click this link for a good visual representation as to what bubble sort does: here

This class should have a method `binarySearch(Animal a)` To search for an animal, a client should be able to pass in an instance of an animal and the method will return the index if the animal is found or -1 if it cannot be found. Use the binary search method to search through the pets. Click this link for a good visual representation of the binary search method: here

There will also be a method called `linearSearch(Animal a)` which similarly to `binarySearch(Animal a)`, will take in an animal and return either the index that the pet is found or -1 if it cannot be found. This method however should use linear searching instead of binary searching. Click this link for a visual representation of linear searching: here

You should understand the efficiency of the search and sort methods that you have implemented. In the javadocs of the search and sort methods write the Big(O) efficiency and give a brief explanation as to why this is the efficiency.

5. StoreOrganizer Interface:

This is the interface that the Store class should implement. Do not change this interface and be sure your Store class implements this correctly or your code will not properly compile.

6. PetShop Class:

Do not change this class at all! This is the class creating the GUI. You do not need to understand how this code is working yet. If you are having trouble running this class then go to the Background section. If your code is not working or not compiling with this class then you should be changing your classes, not this one.

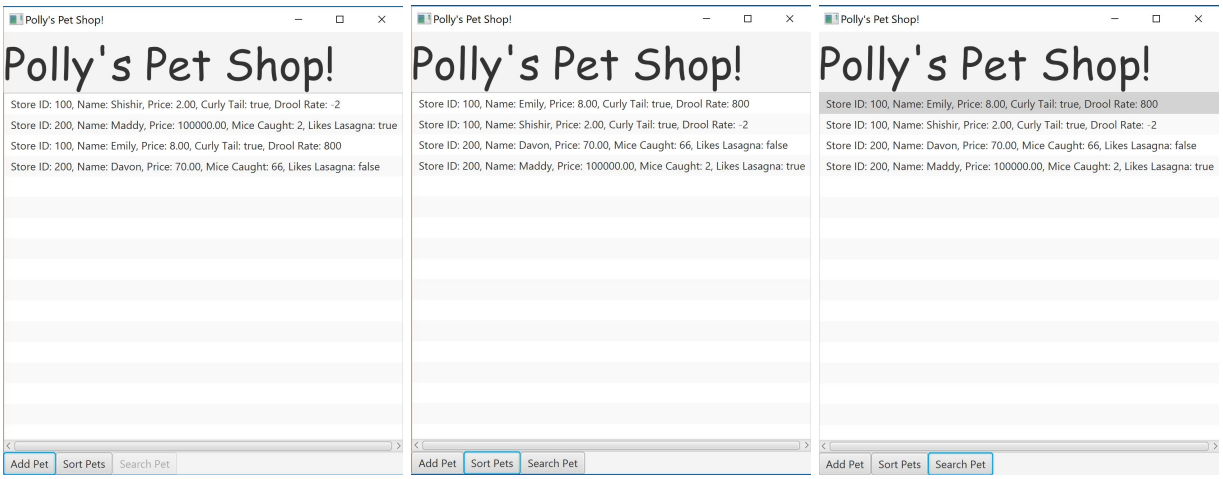
7. PetValidator Class:

Do not change this class at all! This is a utility class which ensures validity of user input and that the GUI will not crash. You will not be turning this class in and as such if you change this class to make your code run better then you will lose points. Read the javadocs to learn more about the methods in this class if you are interested, but that is not required.

Examples

Remember that you have written the backend for Polly's Pet Shop application. The frontend, or GUI, displays the results of the backend. This is helpful when testing your code. You can use the GUI to add, search, and sort pets and be sure that your code is working properly. We have included some pictures below of what the GUI may look like if working properly.

After using the add button, this is an example of what the GUI should look like. Note that this is before sorting.



The image on the left shows what the GUI should look like after using the add button. The image in the center is an example of sorting the same pets from the first image. The image on the right is an example of searching for the pet named Emily. Note that the found pet is highlighted.

Background

You will need to use JavaFX in order to compile and run the application. Java11 does not come build with JavaFX, so you will have to download this. Below are instructions to download JavaFX and how to compile and run JavaFX files for Windows and Mac. If you have troubles with this please come to office hours as it is difficult to resolve computer download problems remotely through Piazza.

NOTE: If you are having trouble with the GUI not reflecting changes you are making in your code, then you may have to manually delete the .class files in the directory and recompile.

Windows and Mac:

1. Use this link to download the Java SDK. Be sure it is the SDK and not jmod: here
2. After doing this you will have a new zipped folder on your computer called `javafx-sdk-11.0.2`. Extract the contents of this zipped folder to the directory holding your hw7 code. Be sure it is in the same folder as your code. There is a way to run JavaFX with the SDK in a different folder but it involves setting environment variables and is much more complex.
3. Compile the file with the JavaFX code (in this case it will be the provided file) by typing into the command prompt (being sure to change fileName.java to the file holding the JavaFX code):

```
javac --module-path javafx-sdk-11.0.2/lib --add-modules=javafx.controls fileName.java
```

4. To run the code type in the command prompt(being sure to change fileName to the file holding the JavaFX code):

```
java --module-path javafx-sdk-11.0.2/lib --add-modules=javafx.controls fileName
```

Tips and Considerations

- The difficulty of this assignment stems from you having more freedom in how you code this. Start early on this and think about good programming techniques taught in lecture.

Allowed Imports

To prevent trivialization of the assignment, you are not allowed to import anything on the classes you write. The imports written in the PetShop class are okay because we are providing this class to you and you should not be modifying this class.

If you would like to import anything else, ask on Piazza.

Feature Restrictions

There are a few features and methods in Java that overly simplify the concepts we are trying to teach. For that reason, do not use any of the following in your final submission:

- var (the reserved keyword)
- System.arraycopy

Rubric

- [15] **Animal Class**
  - [2] Class is abstract
  - [2] Class implements Comparable
  - [3] Two correctly written constructors
  - [3] `toString` properly formatted
  - [5] `compareTo` proper implementation
- [17] **Dog Class**
  - [2] Class implements Comparable
  - [2] Class extends Animal
  - [2] Setters and getters work properly
  - [3] Two correctly written constructors
  - [3] `toString` properly formatted
  - [5] `compareTo` proper implementation
- [17] **Cat Class**
  - [2] Class implements Comparable
  - [2] Class extends Animal
  - [2] Setters and getters work properly
  - [3] Two correctly written constructors
  - [3] `toString` properly formatted
  - [5] `compareTo` proper implementation
- [51] **Store Class**
  - [2] Class implements StoreOrganizer
  - [2] Properly implements the StoreOrganizer interface
  - [2] Big O is correct for the sort and search methods
  - [3] Explanation for Big O is given and correct
  - [4] Sorting method properly sorts the pets
  - [4] Search method returns correct index
  - [4] Add method properly adds a pet
  - [10] Sort properly uses bubble sort
  - [10] `linearSearch` properly uses linear search
  - [10] `binarySearch` properly uses binary search

Checkstyle and Javadocs

You must run checkstyle on your submission. The checkstyle cap for this assignment is **20** points. Review the style guide and download the checkstyle jar. Run checkstyle on your code like so:

```
$ java -jar checkstyle-6.2.2.jar -a *.java
Audit done. Errors (potential points off):
0
```

Make sure to include the -a flag included for testing both checkstyle and javadocs

The message above means there were no Checkstyle or javadoc errors. If you had any errors, they would show

up above this message, and the number at the end would be the points we would take off (limited by the checkstyle cap mentioned above). The Java source files we provide contain no Checkstyle errors. In future homeworks we will be increasing this cap, so get into the habit of fixing these style errors early!

Depending on your editor, you might be able to change some settings to make it easier to write style-compliant code. See the customization tips page for more information.

Collaboration

Collaboration Statement

To ensure that you acknowledge a collaboration and give credit where credit is due, **we require that you place a collaboration statement as a comment at the top of at least one java file that you submit**. That collaboration statement should say either:

I worked on the homework assignment alone, using only course materials.

or

In order to help learn course concepts, I worked on the homework with [give the names of the people you worked with], discussed homework topics and issues with [provide names of people], and/or consulted related material that can be found at [cite any other materials not provided as course materials for CS 1331 that assisted your learning].

Turn-In Procedure

Submission

To submit, upload the files listed below to the corresponding assignment on Gradescope:

- Animal Class
- Dog Class
- Cat Class
- Store Class

Make sure you see the message stating "HW## submitted successfully". From this point, Gradescope will run a basic autograder on your submission as discussed in the next section.

You can submit as many times as you want before the deadline, so feel free to resubmit as you make substantial progress on the homework. We will only grade your last submission: be sure to **submit every file each time you resubmit**.

Gradescope Autograder

For each submission, you will be able to see the results of a few basic test cases on your code. Each test typically corresponds to a rubric item, and the score returned represents the performance of your code on those rubric items only. If you fail a test, you can look at the output to determine what went wrong and resubmit once you have fixed the issue.

The Gradescope tests serve two main purposes:

1) Prevent upload mistakes (e.g. forgetting checkstyle, non-compiling code)
2) Provide basic formatting and usage validation

In other words, the test cases on Gradescope are by no means comprehensive. Be sure to thoroughly test your code by considering edge cases and writing your own test files. You also should avoid using Gradescope to compile, run, or checkstyle your code; you can do that locally on your machine.

Other portions of your assignment are also autograded once the submission deadline has passed. The autograders used are often dependent on specific output formats, so **make sure that your submission passes all test cases marked "FORMAT:"**.

Important Notes (Don't Skip)

- Non-compiling files will receive a 0 for all associated rubric items
- Test your code in addition to the basic checks on Gradescope
- Submit every file each time you resubmit
- Ensure you pass all "FORMAT:" tests
- Read the "Allowed Imports" and "Restricted Features" to avoid losing points
- Check on Piazza for a note containing all official clarifications