CSC216: Project 2

Project 2, Part 2: PackDoption

Project 2, Part 2: PackDoption

Printable version (project2-part2-print)

Deliverables and Deadlines

Part 1 of this assignment (project2-part1) laid out the requirements for the PackDoption application that you must create. Part 2 details the design that you must implement. For this part, you must create a complete Java program consisting of multiple source code files and JUnit test cases. You must electronically submit your files to NCSU GitHub by the due date.

Process Points 1 Deadline: Thursday, July 11 at 11:45PM

Process Points 2 Deadline: No deadline; Process Point 2 is meant to guide you from Process Point 1 to your completed implementation.

Part 2 Due Date: Thursday, July 18 at 11:45PM

Late Deadline: Saturday, July 20 at 11:45PM

Set Up Your Work Environment

Before you think about any code, prepare your work environment to be compatible with the teaching staff grading criteria. The teaching staff relies on NCSU GitHub and Jenkins for grading your work. Follow these steps:

1. Create an Eclipse project named `Project2`: Right click in the package explorer, select *New > Java Project*. Name the project `Project2`. (The project naming is case sensitive. Be sure that the project name begins with a capital P. Also, no space between `Project` and `2`!)
2. If you have not already done so, clone your remote NCSU GitHub repository corresponding to this project. Note that each major project will have its own repository.

Working with a Partner

> ## Note: Assigned Partner
>
> If you were assigned a partner for Project 2 Part 2, you may work only with your assigned partner. Otherwise, you are expected to complete the project individually. All work must be strictly (a) your own work or (b) the work of you and your partner.

If you are working with a partner do the following to make sure that you both are working on the same project and to avoid collisions.

1. Identify who will be partner A and who will be partner B.
2. Partner A should create the Eclipse project as described above. Then Partner A will clone the team repo and share Project 2 to the local copy of the cloned repo. Then Partner A should commit/push the shared project to NCSU's GitHub. Verify that the project is there by navigating to the repo on the GitHub website. The project should be listed as `Project2` and should contain the `src/` folder, a `.classpath`, and `.project` files (at a minimum).
3. Partner B should clone the repo AFTER Partner A has pushed the project. Then Partner B will import the existing project into their Eclipse workspace through the GitHub repositories view.
4. Communicate frequently to avoid merge conflicts. If you have a merge conflict see the CSC Git Guide (https://pages.github.ncsu.edu/engr-csc216-staff/CSC216-SE-Materials/git-tutorial/) for help.

Requirements

The requirements for this project are described in Project 2 Part 1 (project2-part1) in nine different use cases:

- Use Case 1: Startup and Shutdown (project2-part1#use-case-1-startup-and-shutdown)
- Use Case 2: Open a PackDoption data file (project2-part1#use-case-2-open-a-packdoption-data-file)
- Use Case 3: Save a PackDoption data file (project2-part1#use-case-3-save-a-packdoption-data-file)
- Use Case 4: Add rescue to system (project2-part1#use-case-4-add-rescue-to-system)
- Use Case 5: View animals (project2-part1#use-case-5-view-animals)
- Use Case 6: Filter animals (project2-part1#use-case-6-filter-animals)
- Use Case 7: Manipulate animal information (project2-part1#use-case-7-manipulate-animal-information)
- Use Case 8: Add animal to vet queue (project2-part1#use-case-8-add-animal-to-vet-queue)
- Use Case 9: Remove animal from vet queue (project2-part1#use-case-9-remove-animal-from-vet-queue)

Last minute clarification on requirements

- Use Case 7: Manipulate animal information (project2-part1#use-case-7-manipulate-animal-information): Notes are only displayed in this view. Notes are added from a file or from the appointments view.
- Use Case 9: Remove animal from vet queue (project2-part1#use-case-9-remove-animal-from-vet-queue): A third Alternative Flows exists if a note with same message and date as is listed in the GUI already exists for the animal; an error dialog is shown.

Design

Your program must implement the teaching staff design, which is described below. The design consists of 16 different classes (two as inner classes and one abstract class), three enumerations, and three interfaces. We are providing the `PackDoptionGUI` GUI class (assets/part2/code/PackDoptionGUI.java) and the interfaces (`Queue` interface (assets/part2/code/Queue.java), `SimpleListIterator` interface (assets/part2/code/SimpleListIterator.java), and `SortedList` interface (assets/part2/code/SortedList.java)).

> **Important**
>
> The project you push to NCSU GitHub must contain *unmodified* versions of the files that we provide for you, unless stated otherwise by the instructors.

`edu.ncsu.csc216.packdoption.util` package

- `NoSuchListElementException` : Custom unchecked exception to be thrown when there's no such list element.
- `Date` : A class that represents a date as a month, a day, and a year (M/D/YYYY).
- `Note` : A class that represents a note as a date and a message.
- `Queue<E>` (assets/part2/code/Queue.java): A modification (assets/part2/code/Queue.java) of the `java.util.Queue` interface that is customized for the PackDoption application. **You cannot change this code in any way.**
- `SortedList<E extends Comparable<E>>` (assets/part2/code/SortedList.java): A modification (assets/part2/code/SortedList.java) of the `java.util.List` interface that is customized for the PackDoption application. **You cannot change this code in any way.**
- `SimpleListIterator<E extends Comparable<E>>` (assets/part2/code/SimpleListIterator.java): A highly restricted modification (assets/part2/code/SimpleListIterator.java) of the `java.util.ListIterator` interface that is customized for the PackDoption application. **You cannot change this code in any way.**
- `ArrayListQueue<E>` : Implements the `Queue` interface with an array data structure.
- `SortedLinkedList<E extends Comparable<E>>` : An implementation of the `SortedList` interface with a data structure of linked `Node` s.

- `Node` : `Node` is a private, static inner class of `SortedLinkedList` that contains an `E` and a reference to the next `Node` in the `SortedLinkedList` .
- `Cursor` : A private inner class of `SortedLinkedList` that provides a cursor for iterating forward through the list without changing the list.

`edu.ncsu.csc216.packdoption.model` package

`edu.ncsu.csc216.packdoption.model.animals`

Sub-package of `edu.ncsu.csc216.packdoption.model` containing representation of animals.

- `Animal` : An abstract class representing an animal. Contains two inner enumerations, which will be provided for you in the Implementation section, below.
  - `AgeCategory` . An enumeration contained in the `Animal` class. Represents one of the three possible age categories for an animal.
  - `Size` . An enumeration contained in the `Animal` class. Represents one of the three possible Sizes for an animal.
- `Cat` : A concrete class extending `Animal` representing a cat.
- `Dog` : A concrete class extending `Animal` representing a dog. Contains one inner enumeration, which will be provided for you in the Implementation section, below.
  - `Breed` . An enumeration contained in the `Dog` class. Represents one of the 12 possible breeds for a dog.

`edu.ncsu.csc216.packdoption.model.rescue`

Sub-package of `edu.ncsu.csc216.packdoption.model` containing representation of rescues.

- `Rescue` : A representation of a rescue.
- `RescueList` : A `RescueList` has a `SortedLinkedList` of `Rescue` s.

`edu.ncsu.csc216.packdoption.model.io`

Sub-package of `edu.ncsu.csc216.packdoption.model` containing file processing classes.

- `PackDoptionReader` . A class for reading PackDoption files. If there are any errors while processing, the entire file is considered invalid and an `IllegalArgumentException` is thrown with the message "Unable to load file."
- `PackDoptionWriter` . A class for writing PackDoption files. If there are any errors while processing, an `IllegalArgumentException` is thrown with the message "Unable to save file."

`edu.ncsu.csc216.packdoption.model.manager`

Sub-package of `edu.ncsu.csc216.packdoption.model` containing the overarching `PackDoptionManager` functionality.

- `PackDoptionManager` . Maintains the data structures for the entire application. `PackDoptionManager` implements the Singleton Design Pattern (https://en.wikipedia.org/wiki/Singleton_pattern).

`edu.ncsu.csc216.packdoption.view` package

`edu.ncsu.csc216.packdoption.view.ui`

Sub-package of `edu.ncsu.csc216.packdoption.view` containing the view-controller elements of the PackDoption Model-View-Controller pattern.

- `PackDoptionGUI` GUI class (assets/part2/code/PackDoptionGUI.java): The graphical user interface for the project. This is the class that starts execution of the program. **You cannot change this code in any way.**

UML Diagrams

The UML class diagram (assets/part2/PackDoptionClassDiagram.png) for the design is shown in the figure below. The fields and methods represented by green circles (public) or yellow diamonds (protected) represent the minimum set of visible state and behavior required to implement the project.

Access Modifiers

Note: You can modify the names of private variables, parameters, and private methods. However, you MUST have the non-private data and methods (names, return types, parameter types and order) exactly as shown for the teaching staff tests to run. YOU MAY NOT ADD ANY ADDITIONAL PUBLIC OR PROTECTED CLASSES, METHODS, OR STATES.

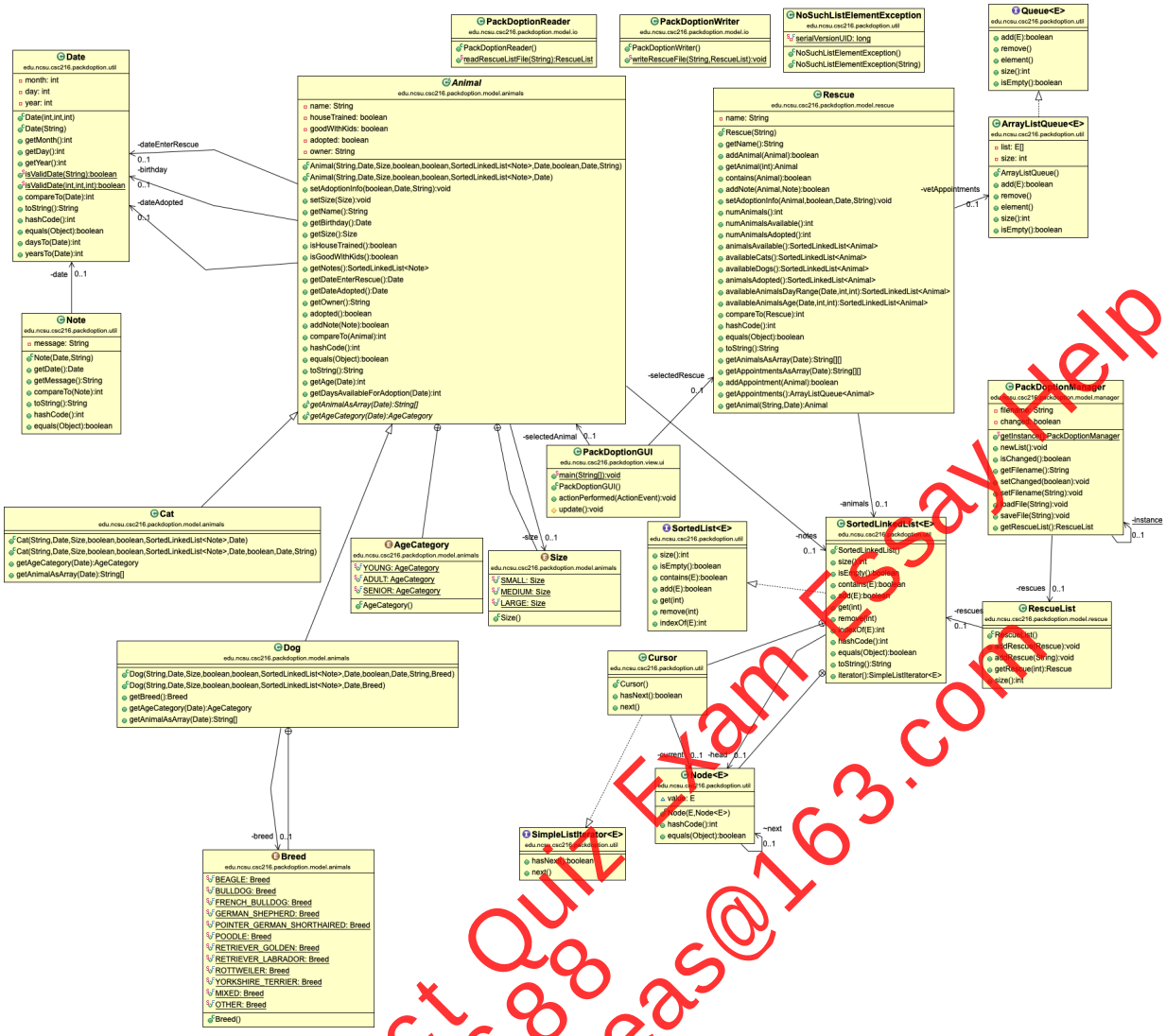

Figure 1: PackDoption UML class diagram

UML Diagram Notations

UML uses standard conventions for display of data, methods, and relationships. Many are illustrated in the UML diagram above. Here are some things you should note:

- `-` or a red square (empty or solid) in front of a class member means private.
- `+` or a green circle (empty or solid) in front of a class member means public.
- `#` or a yellow triangle (empty or solid) in front of a class member means protected.
- Static members (data or methods) are underlined or have `S` in icon.
- Methods that are declared but not defined (abstract methods or those declared in interfaces) are in italics or have `A` in icon, unless they are in an interface.
- The names of abstract classes are in italics or have `A` in icon.
- Dotted arrows with triangular heads point to interfaces from the classes that implement them.
- Solid arrows with triangular heads go from concrete classes to their parents.
- Solid arrows with simple heads indicate *has-a* relationships (composition). The containing class is at the tail of the arrow and the class that is contained is at the head. The arrow is decorated with the name and access of the member in the containing class. The arrow is also decorated

with the "multiplicity" of the relationship, where 0..1 means there is 1 instance of the member in the containing class. (See `notes` on the arrow from `Animal` to `SortedLinkedList`.) A multiplicity of 0..* means there are many, usually indicating a collection such as an array or collection class.

- A circle containing an "X" or cross sits on a the border of a class that contains an inner (nested) class, with a line connecting it to the inner class. (See the class `Node` and its corresponding outer class `SortedLinkedList`. Also see the enumerations.)
- SF in front of a name means static, final.
- Methods embellished with C are constructors. (See `SortedLinkedList. SortedLinkedList()`.) Note that while all classes require a constructor, not all constructors require implementation. In some cases the default constructor is sufficient.

Implementation Process

Professional software development is more than building working software. The process of building software is important in supporting teams of developers to build complex systems efficiently. We expect you to follow good software engineering practices and processes as you create your software to help you become a better software developer and to help with understanding your progress on the project.

Project-wide Process Points

There are certain practices that you should follow throughout project development.

- **Commit with meaningful messages.**

> #### Meaningful Commit Messages
>
> The quality of your commit messages for the entire project history will be evaluated for meaning and professionalism ***as part of the process points***.

Process Points 1 Milestone

Process Points 1 Milestone gets you started with the project and ensures that you meet the design and that you start commenting your code.

- **Compile a skeleton.** The class diagram provides a full skeleton of what the implemented PackDoption program should look like. Start by creating an Eclipse project named **Project2**. Copy in provided code and create the skeletons of the classes you will implement, **including the test classes**. Ensure that the packages, class names, and method signatures match the class diagram *exactly!* If a method has a return type, put in a place holder (for example, `return 0;` or `return null;`) so that your code will compile. Push to GitHub and make sure that your Jenkins job shows a yellow ball. A yellow ball on Jenkins means that:
  - your code compiles,

- the teaching staff tests compile against your code, which means that you have the correct code skeleton for the design, and
- you have a skeleton of your test code in the `test\` folder.

- **Comment your code.** Javadoc your classes and methods. When writing Javadoc for methods, think about the inputs, outputs, preconditions, and post conditions. For this milestone, you must have NO CheckStyle notifications on Jenkins.
- **Run your CheckStyle and PMD tools locally** and make sure that all your Javadoc is correct. Make sure the tools are configured correctly (see configuration instructions (https://pages.github.ncsu.edu/engr-csc216-staff/CSC216-SE-Materials/install/)) and set up the tools to run with each build so you are alerted to style notifications shortly after creating them.
- **Practice test-driven development.** Start by writing your test cases. This will help you think about how a client would use your class (and will help you see how that class might be used by other parts of your program). Then write the code to pass your tests.

> ### Compiling Skeleton & Fully Commented Classes
>
> Fully commented classes and methods on at least a skeleton program are due ***at least two weeks before*** the final project deadline to earn the associated process points. See the deadline for Process Points 1 at the top of this document.

Process Points 2 Milestone

The Project 2 build process is set up differently than the builds Guided Project 3 and Labs. Instead of requiring that you have unit tested all of your classes with high quality tests that executed 80% of statements (i.e., 80% statement coverage) before running and reporting the results of the teaching staff tests, the Project 2 build process will provide teaching staff test feedback when you have united tested all of your classes with high quality tests that executed 60% of statements (i.e. 60% statement coverage). **You are still responsible for achieving 80% statement coverage on all of your classes by the project deadline for full credit.**

If a class like `DogTest` has a JUnit related PMD notification or insufficient coverage, teaching staff test feedback for `Dog` will not be provided. To earn full credit for Process Points 2, you should have 80% statement coverage from high quality unit tests, no PMD JUnit notifications, and be passing all of your unit tests. You should be passing your teaching staff tests and reach 80% statement coverage for all non-ui source files by the project deadline.

If any of the teaching staff test feedback is not revealed to you, your build's console output will contain the following string `NOT ALL TS TESTS HAVE RUN`. The number of teaching staff tests that have run are displayed a few lines higher up in the console output. While the build *should* be configured to only go to green ball when all teaching staff tests are revealed and you have sufficient coverage and no static analysis notifications, the system is in beta.

It is your responsibility to ensure that all 12 teaching staff test files are reporting results and that you're passing the tests to ensure completion of the project. You can check see if the teaching staff tests have run by exploring the `Test Results` links in on the build and checking that all twelve

teaching staff test files are listed.

If you run into any problems with the build that you think are scaffolding related, please email Mr. Matthew Zahn directly (mszahn@ncsu.edu) or `@` him in a Piazza post so he will receive the notification about the issue. Mr. Zahn will work to resolve issues within 24 hours of notification, but an issue may not be resolved after 4pm on the Process Points 2 deadline until the following day. Please plan accordingly and start early!

Implementation

We suggest you attack Project 2 Part 2 one piece at a time. The details below provide the suggested implementation order and details about the implementation. Getter methods are listed in the UML diagram but are not listed below in the details sections. All getters return the appropriate fields.

`edu.ncsu.csc216.packdoption.util` package

`NoSuchListElementException` Details

Custom unchecked exception to be thrown when there's no such list element. The default message should be "No such element in list."

`Date` Details

A class that represents a date as a month, a day, and a year (M/D/YYYY). M/D means one or two digits for each the month and the date. A valid date must have year between 2000 and 2050 (inclusive). `Date` implements the `Comparable` interface.

`Date` has two constructors:

- `Date(int month, int day, int year)` : The constructor should throw an `IllegalArgumentException` with "Invalid date" message if month, day, and year do not form a valid date.
- `Date(String date)` : The constructor should throw an `IllegalArgumentException` with "Invalid date" message if (a) `date` is not in the correct format (`M/D/YYYY`) or (b) month, day, and year do not form a valid date. Note that for this method, any month and day can be listed with two digits. For example, all of the following are valid: 10/8/2019, 10/18/2019, 9/7/2020, 9/21/2020, and 09/07/2020.

There are two `static` methods (`isValidDate`) to test whether given parameter(s) form valid date. See above for valid years and string format.

Other methods:

- `compareTo(Date o)` : `Date` s are compared based on year, month, day (in that order). For example, if two dates have different years, we can order based on year only. Whereas if two dates have the same year, we would need to look at month as well.
- `toString()` : Returns `Date` as a string with the format `M/D/YYYY` , where months and days less than ten would be single digit and not have a leading zero.
- `hashCode()` and `equals(Object obj)` : These methods are based on the `month` , `day` , and `year` fields.

- `daysTo(Date other)` : Returns the number of days between `this` date and `other` . Number will be negative if `other` is before `this` .
- `yearsTo(Date other)` : Returns the number of years between `this` date and `other` . Only reports full years. For example, years between 11/8/2018 and 11/7/2019 would return 0, and years between 11/8/2018 and 11/8/2019 would return 1. Number will be negative if `other` is at least a year before `this` .

`Note` Details

A class that represents a note as a date and a message. `Note` implements the `Comparable` interface.

- `Note(Date date, String message)` :The constructor should throw an `IllegalArgumentException` with message "Invalid note" if (a) `date` is null, (b) `message` is null, (c) `message` is whitespace only, or (d) `message` contains `\n` or `,` . `message` should be trimmed of leading and/or trailing whitespace.
- `compareTo(Note o)` : `Note` s are compared based on date then message.
- `toString()` : Returns `Note` as a string with the format `date message` . For example, "10/3/2019 a note".
- `hashCode()` and `equals(Object obj)` :These methods are based on the `date` and `message` fields.

`ArrayListQueue<E>` Details

The `Queue` interface contains five methods that must be implemented by `ArrayListQueue` .The Javadoc of the `Queue` interface describes what each method should do, including when exceptions must be thrown.

> #### Custom List
>
> ***Note:*** The provided `Queue` interface is a CUSTOM version of the `java.util.Queue` interface specifically for this project. Details of this custom `Queue` interface vary from the Java API's `List` interface.

`SortedLinkedList<E extends Comparable<E>>` Details

`SortedList<E extends Comparable<E>>` is an interface containing seven methods that must be implemented by `SortedLinkedList<E extends Comparable<E>>` . `SortedLinkedList` should be implemented as a singly linked list, with an ordinary nested class `Node` .

The Javadoc of the `SortedList` interface describes what each method should do, including when exceptions must be thrown. Note: The provided `SortedList` interface is a *custom* version of the `java.util.List` interface specifically for this project. Details of this custom sorted list interface vary from the Java API's `List` interface.

Iterators are objects that can travel through lists. `SortedLinkedList` has an additional nested class, `Cursor`, which is for traveling through the list, one item at a time. It implements `SimpleListIterator<E extends Comparable<E>>`, which requires only two simple behaviors: 1) tell what the next item is then travel down the list to the following item, and 2) tell you whether there is a next item. Such a simple iterator cannot travel backwards, it cannot change, add, or remove list items.

Four `SortedLinkedList` methods are not declared in `SortedList`:

- `iterator()`: Returns a `SimpleListIterator`. Since the inner class `Cursor` implements `SimpleListIterator`, this method should simply return an instance of `Cursor`.
- `toString()`: Returns a string representation of the list in the format "-A\n-B\n...-X" where "A" is the first list item, "B" is the second, ..., and "X" is the last. For example, if the list contains "Apple", "Betty", "Claude", and "Matthew," then the string representation would be "-Apple\n-Betty\n-Claude\n-Matthew". An empty list would return "".
- `hashCode()` and `equals(Object obj)`: These methods are based on the `head` field.

> ## Custom Iterators
>
> ***Note:*** A *simple iterator,* which represents a cursor into a list, provides only two behaviors: (1) a method to determine if there are additional items in the list for the iterator to visit; and (2) a method that returns the next element to be visited and pushes the cursor beyond that element. These are far fewer behaviors than java.util.Iterator.

`edu.ncsu.csc216.packoption.model` package

`Animal` Details

An abstract class representing an animal.

`Animal` should include enumerations for the animal size (`Size`) and age category (`AgeCategory`). Our textbook defines enumeration as "a type that has only a small number of predefined constant values." Since there are a discrete number of sizes and age categories, an enumeration is quite appropriate to list those values.

Copy the code below into the `Animal` class exactly as given. You can include the enumeration code with the fields of `Animal`. Enumerations are pseudo-objects (like arrays). **Constructors are listed in the class diagram, but are not needed in the implementation.** You should Javadoc each enumeration like a field.

JAVA

```
1 public enum AgeCategory { YOUNG, ADULT, SENIOR }
2
3 public enum Size { SMALL, MEDIUM, LARGE }
```

To access a value in the enumeration, use the enumeration name followed by the value (for example, `Size.SMALL`). Enumerated types are essentially a name given to an integer value. Therefore, you can use primitive comparison operators (`==` and `!=`) to compare enumerated type

variables (of the same enumerated type- don't try to compare a `AgeCategory` with a `Size` !). Enumerated types can also be the type of a variable. For more details on enumerated types, see pp. 1133- 1134 in the Reges and Stepp 4th edition textbook or pp. 1187- 1188 in the 5th edition.

There are two `Animal` constructors:

```java
// Constructor 1
Animal(String name, Date birthday, Size size, boolean houseTrained, boolean goodWithKids
       boolean adopted, Date dateAdopted, String owner)

// Constructor 2
Animal(String name, Date birthday, Size size, boolean houseTrained, boolean goodWithKids
```

`name` and `owner` should be trimmed of leading and/or trailing whitespace. For constructor 2, `adopted` is set to `false`, `dateAdopted` is set to `null`, and `owner` is set to `null`.

The constructors should throw an `IllegalArgumentException` if any of the following are true:

1. `name` is `null`
2. `name` is whitespace only
3. `name` contains `\n` or `,`
4. `birthday` is `null`
5. `size` is `null`
6. `notes` is `null`
7. `dateEnterRescue` is `null`
8. `dateEnterRescue` is before `birthday`
9. `adopted` is `false` but `dateAdopted` or `owner` is non-null
10. `adopted` is `true` but `dateAdopted` or `owner` is null
11. `dateAdopted` is before `dateEnterRescue`
12. `owner` is whitespace only
13. `owner` contains `\n` or `,`

Methods:

- `setAdoptionInfo(boolean adopted, Date dateAdopted, String owner)`: Should set the appropriate fields. `owner` should be trimmed of leading and/or trailing whitespace. Throws an `IllegalArgumentException` if one or more items 9-13 from constructor list are true.
- `setSize(Size size)`: Set `size` field. Throws an `IllegalArgumentException` if `size` is `null`.
- `addNote(Note note)`: Adds `note` to `notes`. Returns `true` if added, `false` otherwise. Throws an `IllegalArgumentException` if `note` is `null` or if `notes` already contains `note`.
- `hashCode()` and `equals(Object obj)`: These methods are based on the `name` and `birthday` fields.
- `compareTo(Animal other)`: `Animal`s are compared based on birthday then name.
- `toString()`: Returns `Animal` as a string with the format `name (birthday)\nnotes`. For example, "Peyton (9/8/2012)\n-9/8/2012 Birthday!!!".
- `getAge(Date today)`: Returns age in years. Throws an `IllegalArgumentException` if (1) `today` is `null` or (2) `today` is before `birthday`.

- `getDaysAvailableForAdoption(Date today)` : Returns days in rescue (from `dateEnterRescue` to `today`). Returns `-1` if animal has been adopted. Throws an `IllegalArgumentException` if (1) `today` is `null` or (2) `today` is before `dateEnterRescue` .
- `getAgeCategory(Date today)` : Abstract method that returns age category (project2-part1.html#requirements) based on `today` . Throws an `IllegalArgumentException` if (1) `today` is `null` or (2) `today` is before `birthday` .
- `getAnimalAsArray(Date today)` : Abstract method that returns a String array with seven elements based on `today` : Name, Type (Dog or Cat), Birthday, Age, Age Category, Adopted (Yes or No), Days in Rescue (if adopted then empty string). Throws an `IllegalArgumentException` if (1) `today` is `null` or (2) `today` is before `birthday` .

`Cat` Details

Concrete class of `Animal` representing a cat.

`Dog` Details

Concrete class of `Animal` representing a dog.

Copy the code below into the `Dog` class exactly as given. **Constructors are listed in the class diagram, but are not needed in the implementation.** You should Javadoc each enumeration like a field.

```
➤JAVA
1 public enum Breed { BEAGLE, BULLDOG, FRENCH_BULLDOG, GERMAN_SHEPHERD, POINTER_GERMAN_SHO
```

`Rescue` Details

A representation of a rescue. Each rescue has a name, a `SortedLinkedList` of `Animal` s (`animals`) who are part of the rescue, and an `ArrayListQueue` of `Animal` s (`vetAppointments`) that are waiting to see the veterinarian.

- `Rescue(String name)` : Constructs rescue with given name, empty `SortedLinkedList` , and empty `ArrayListQueue` . Throws an `IllegalArgumentException` if (1) `name` is `null` , (2) `name` is whitespace only, or (3) `name` contains `\n` . `name` should be trimmed of leading and/or trailing whitespace.
- `addAnimal(Animal animal)` : Adds `animal` to `animals` and returns `true` . Returns `false` if try to add a duplicate animal. Throws an `IllegalArgumentException` if `animal` is `null` .
- `getAnimal(int i)` : Returns animal at `i` index of `animals` . Throws an `IndexOutOfBoundsException` if `i` is negative or greater than size - 1.
- `getAnimal(String name, Date birthday)` : Returns animal in rescue with given name and birthday; otherwise, returns `null` . Throws an `IllegalArgumentException` if (1) `name` is `null` or (2) `birthday` is `null` .
- `contains(Animal a)` : Returns whether `animals` contains `a` .
- `addNote(Animal animal, Note note)` : Adds note to `animal` and returns `true` if `animal` is in the rescue. Returns `false` and does not add note if animal is not in the rescue. Throws an `IllegalArgumentException` if `animal` is `null` , `note` is `null` , the animal's notes already contains `note` .

- `setAdoptionInfo(Animal animal, boolean adopted, Date dateAdopted, String owner)` : Sets adoption information for animal if animal is in the rescue. Does nothing if animal is not in the rescue. Throws an `IllegalArgumentException` if `animal` is `null` . See `Animal.setAdoptionInfo()` for other cases when `IllegalArgumentException` is thrown.
- `numAnimals()` : Returns number of animals in rescue.
- `numAnimalsAvailable()` : Returns number of animals in rescue that are available for adoption (have not been adopted).
- `numAnimalsAdopted()` : Returns number animals in rescue that have been adopted.
- `animalsAvailable()` : Returns `SortedLinkedList` of all animals in rescue that are available for adoption (have not been adopted).
- `availableCats()` : Returns `SortedLinkedList` of all cats in rescue that are available for adoption (have not been adopted).
- `availableDogs()` : Returns `SortedLinkedList` of all dogs in rescue that are available for adoption (have not been adopted).
- `animalsAdopted()` : Returns `SortedLinkedList` of animals in rescue that have been adopted.
- `availableAnimalsDayRange(Date today, int min, int max)` : Returns `SortedLinkedList` of all animals in rescue that are available for adoption (have not been adopted) and have been available between `min` and `max` days (inclusive). Throws an `IllegalArgumentException` if (1) `today` is `null` , (2) `today` is before one of the animal's `dateEnterRescue` , (3) `max` is less than `min` , or (4) `min` is less than zero.
- `availableAnimalsAge(Date today, int min, int max)` : Returns `SortedLinkedList` of all animals in rescue that are available for adoption (have not been adopted) and are between `min` and `max` years old (inclusive). Throws an `IllegalArgumentException` if (1) `today` is `null` , (2) `today` is before one of the animal's `birthday` , (3) `max` is less than `min` , or (4) `min` is less than zero.
- `compareTo(Rescue o)` : `Rescue` s are compared based on name.
- `hashCode()` and `equals(Object obj)` : These methods are based on the `name` field.
- `toString()` : Returns `Rescue` name.
- `getAnimalsAsArray(Date today)` : Returns a 2D array of string representing all animals in rescue. First dimension is the animal (in sorted order). Second dimension are the different elements: Name, Type (Dog or Cat), Birthday, Age, Age Category, Adopted (Yes or No), Days in Rescue (if adopted then empty string). If the list is empty, an empty 2D array is returned.
- `addAppointment(Animal animal)` : Adds `animal` to `vetAppointments` and returns `true` if `animal` is in rescue. Otherwise, returns `false` . Throws a `NullPointerException` if `animal` is `null` .
- `getAppointmentsAsArray(Date today)` : Returns a 2D array of string representing animals in veterinary queue. First dimension is the animal (in same order as queue). Second dimension are the different elements: Name, Type (Dog or Cat), Birthday, Age, Age Category, Adopted (Yes or No), Days in Rescue (if adopted then empty string). If the queue is empty, an empty 2D array is returned.

`RescueList` Details

An `RescueList` has a `SortedLinkedList` of `Rescue` s.

- `RescueList()` : Constructs new `RescueList` with an empty `SortedLinkedList` of `Rescue` s.

- `addRescue(Rescue r)` : Adds `r` to list of rescues. Throws an `IllegalArgumentException` if (1) `r` is `null` or (2) `r` is already in the list.
- `addRescue(String name)` : Adds a rescue with `name` to list of rescues. Throws an `IllegalArgumentException` if (1) `name` is `null` , (2) `name` is whitespace only, or (3) `name` contains `\n` , or (4) a rescue with `name` is already in the list.
- `getRescue(int idx)` : Returns rescue at index `idx` . Throws an `IndexOutOfBoundsException` if `idx` is negative or greater than size - 1.
- `size()` : Returns number of rescues in list.

`PackDoptionReader` Details

`readRescueListFile(String filename)` : Returns a `RescueList` from the given file. The file is processed as described in Use Case 2 (project2-part1#use-case-2-open-a-packdoption-data-file). An `IllegalArgumentException` with message "Unable to load file." is thrown on any error or incorrect formatting.

`PackDoptionWriter` Details

`writeRescueFile(String filename, RescueList list)` writes the given `RescueList` to the given file. The file is created as described in Use Case 3 (project2-part1#use-case-3-save-a-packdoption-data-file). An `IllegalArgumentException` with message "Unable to save file." is thrown on any error.

`PackDoptionManager` Details

Maintains the data structures for the entire application. `PackDoptionManager` implements the Singleton design pattern (https://en.wikipedia.org/wiki/Singleton_pattern). This means that only one instance of the `PackDoptionManager` can ever be created. The Singleton pattern ensures that all parts of the `PackDoptionGUI` are interacting with the same `PackDoptionManager` at all times. Note that the `PackDoptionGUI` does not have a global reference (or a field) to `PackDoptionManager` (that's why there's no composition relationship modeled using a connector in the class diagram). Since `PackDoptionManager` is a Singleton, the GUI can access that single instance at any time with the static `getInstance()` method.

- `getInstance()` : Returns instance of `PackDoptionManager` using Singleton design pattern (https://en.wikipedia.org/wiki/Singleton_pattern).
- `PackDoptionManager()` : Constructs new `RescueList` and sets `changed` to false.
- `newList()` : Sets `rescues` to a new `RescueList` .
- `isChanged()` : Returns `changed` (whether data has changed from last save).
- `setFilename(String filename)` : Sets `filename` with leading and/or trailing whitespace removed. Throws `IllegalArgumentException` if (a) `filename` is `null` or (b) `filename` is only whitespace.
- `loadFile(String filename)` : Loads RescueList from file and sets `changed` to `false` . Throws `IllegalArgumentException` if any errors occur reading the file.
- `saveFile(String filename)` : Writes RescueList to file and sets `changed` to `false` . Throws `IllegalArgumentException` if any errors occur writing to the file.

Testing

For Part 2 of this project, you must do white box testing via JUnit AND report the results of running your black box tests from Part 1.

Test Files

We have provided an updated (with correct ordering) sample file (assets/part2/sample-updated.md) that will be helpful when testing the PackDoption system.

White box testing

Your JUnit tests should follow the same package structure as the classes that they test. You need to create JUnit tests for *all* of the concrete classes that you create (even inner classes). At a minimum, you must exercise every method in your solution at least once by your JUnit tests. Start by testing all methods that are not simple getters, simple setters, or *simple* constructors for all of the classes that you must write and check that you are covering all methods. If you are not, write tests to exercise unexecuted methods. You can test the common functionality of an abstract class through a concrete instance of its child.

When testing void methods, you will need to call other methods that do return something to verify that the call made to the void method had the intended effects.

For each method and constructor that throws exceptions, test to make sure that the method does what it is supposed to do and throws an exception when it should.

At a minimum, you must exercise at least 80% of the statements/lines in all ***non-GUI*** classes. You will likely cover the simple getters, setters, and constructors as part of testing more complex functionality in your system. You must have 80% method coverage to achieve a green ball on Jenkins (assuming everything else is correct).

We recommend that you try to achieve 100% condition coverage (where every conditional predicate is executed on both the true and false paths for all valid paths in a method).

Black box testing and submission

Use the provided black box test plan template (../assets/STP_Template.docx) to describe your tests. Each test must be repeatable and specific; all input and expected results values must be concrete. All inputs required to complete the test must be specified either in the document or the associated test file must be submitted. Remember to provide instructions for how a tester would set up, start, and run the application for testing (What class from your design contains the main method that starts your program? What are the command line arguments, if any? What do the input files contain?). The instructions should be described at a level where anyone using Eclipse could run your tests.

If you are planning for your tests to read from or write to files, you need to provide details about what the files so that the teaching staff could recreate them or find them quickly in your project. Similar to Part 1 submissions, the test files may be one of the provided test file (assets/part2/sample-updated.md).

Follow these steps to complete submission of your black box tests:

1. Run your black box tests on your code and report the results in the Actual Results column of your BBTP document.
2. Save the document as a pdf named **BBTP_P1P2.pdf**.
3. Create a folder named **project_docs** at the top level in your project and copy the pdf to that folder.
4. Push the folder and contents to your GitHub repository.

Deployment

For this class, deployment means submitting your work for grading. Submitting means pushing your project to NCSU GitHub.

Before considering your work complete, make sure:

1. Your program satisfies the style guidelines (https://pages.github.ncsu.edu/engr-csc116-staff/CSC116-Materials/course-resources/style-guidelines/).
2. Your program 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. Your program satisfies the gradesheet (../assets/gradesheet.html). You can estimate your grade from your Jenkins feedback (https://pages.github.ncsu.edu/engr-csc216-staff/CSC216-SE-Materials/jenkins/#grade-estimation-example).
4. You generate javadoc documentation on the most recent versions of your project files.
5. In addition to pushing `src/` and `test/`, you should also push any updated `project_docs`, `doc`, and `test-files` folders to GitHub.

Deadline

The electronic submission deadline is precise. Do not be late. You should count on last minute failures (your failures, ISP failures, or NCSU failures). Push early and push often!