COMS 228: Introduction to Data Structures, Summer 2013
Homework Assignment 1

Assignment Overview

If you are familiar with early 2000's computer games, you might be familiar with *Zoo Tycoon*[1]. In this game, animals are kept in cages and attract guests to your zoo. Zookeepers in your zoo will require the assistance of an ace programmer to help them maintain their animal cages. In this assignment you will be programming small subsections of game logic and practicing using JUnit, generics, inheritance, polymorphism and exception handling.

Your job is to create a generic Cage class that holds animals and simulates interactions amongst animals in the cage. In addition, you will be implementing three types of animals that will fit into those cages: tigers, bengals, and sheep. The curators for your zoo have very interesting priorities. One Exception class, one class for JUnit tests and one client Class are also due. There are eight classes to write, but some are very trivial.

Important Note:

In this assignment it is possible to use classes which implement the List interface (these include classes such as LinkedList, ArrayList, Vector, etc.). Please do not use these classes. When you need to store a collection of items, use an array.

Each of the four classes should be written in a separate file. They should all be included in the package "edu.iastate.cs228.hw1". All eight .java files should be located in the directory "edu/iastate/cs228/hw1" as specified by your package name. When submitting your homework through Blackboard, please place the "edu" folder in a zip file. Make sure you include your last name in the name of your zip file.

Specification for the Animal Class

Animal should be an *abstract class that implements Comparable<Animal>.* Animals should have two *protected* instance variables: a name represented by a String and health represented by an int. Animals have concrete methods for getting and setting instance variables and a concrete toString() method that returns the Animal's name and health, separated by a comma.

Animal also has one abstract method called interact.

[1] *http://en.wikipedia.org/wiki/Zoo_ Tycoon*

public abstract void interact(Animal a);

The concrete implementations of Animal will say exactly what the specific rules for animal interactions are. When an Animal is constructed, it should take in a String and an int as such:

public Animal(String name, int health)

Since the Animal is implementing Comparable, you must provide a method called *public int compareTo(Animal a)*. When comparing two Animals, the Animal with more health is 'greater than' an Animal with less health. That is if *Animal a* has more health than *Animal b*. *a.comareTo(b)* will return 1. If *Animal a* has the same health as *Animal b*. *a.comareTo(b)* will return 0. if *Animal a* has less health than *Animal b*. *a.comareTo(b)* will return -1.

Specification for the Tiger Class

A Tiger *is* (extends) an Animal so it must implement the interact method (rules for interacting are below). In addition, a Tiger can also growl. When a Tiger growls it will simply print "Prrrr."

Interact for a Tiger follows these rules:

- If the animal that a Tiger is interacting with is a sheep
  - The Tiger will set the sheep's health to 0.
  - The Tiger will add the sheep's old health to its current health.
- If the animal that a Tiger is interacting with is another Tiger (remember, a Bengal is a type of Tiger)
  - The Tiger will growl at the other Tiger.
- If the animal that a Tiger is interacting with is a Bengal
  - The animal with less health will lose 10% of its current health. (Round the numbers down since we are using integers).

Specification for the Bengal Class

A Bengal *is* (extends) a Tiger so it must implement the interact method (rules for interacting are below). In addition, a Bengal growls differently than a Tiger. When a Bengal growls it will print "RAWR" and gain an additional health point.

Interact for a Bengal follows these rules:

- If the animal that a Bengal is interacting with is a sheep
  - The Bengal will set the sheep's health to 0.
  - The Bengal will add the sheep's old health to its current health.

- If the animal that a Bengal is interacting with is another Bengal
  - The Bengal will growl at the other Bengal.
- If the animal that a Bengal is interacting with is a Tiger (remember, a Bengal is a type of Tiger)
  - The animal with less health will lose 10% of its current health. (Round the numbers down since we are using integers).

Specification for the Sheep Class

A Sheep *is* (extends) an Animal so it must implement the interact method (rules for interacting are below).

Interact for a Sheep follows these rules:

- Sheep are pretty docile so they only print out "Baaaaah", no matter what they interact with.

Specification for the CageException Class

A cage exception is (extends) Exception. The constructor for a CageException will take in an error message as a String and will pass that String to the super constructor of Exception. You may get a warning in regards to serialization, ignore it.

Specification for the Cage Class

A cage is a generic class that can **only contain something of type Animal or things that inherit from Animal**; you must specify an upper bound on the type-variable to do this. A Cage can hold one or more animals but, only a limited amount of animals. When constructing a Cage, it should be given an integer greater than zero or else it will throw a new *CageException* with an appropriate error message. *Hint: The Java compiler does not allow for the creation of generic arrays using the normal syntax.*

T genericArray=new T[10];//will not compile

Instead we will have to do something a little bit goofy.

T arrayOfAnimals=(T[]) new Animal[10];

A Cage also has six public methods:

public Animal[] getOccupants()

This method should simply return your array of Animals

public int getCapacity()

This method should simply return the capacity of the Cage.

public void add(T animal)

This method adds an Animal to your array of Animals *in the first available cell.* There are two errors that can happen here. If an Animal already has the same name, throw a new CageException with an appropriate error message. If adding this Animal will exceed the Cage's capacity, throw a new *CageException* with an appropriate error message.

public void remove(String animalNAme)

This method removes an Animal from your array of Animals. Remove the Animal with the same name as the given string. When an animal is removed, the cells WILL NOT be shifted over. If the given animal does not exist in the Cage, throw a new *CageException* with an appropriate error message.

public void simulate()

This method simulates one round of interaction amongst Animals based off of the following rules.

- Every Animal interacts with the other Animals except itself
  - That is: given Animal A, Animal B and Animal C
    1. Animal A will interact with Animal B and Animal C
    2. Animal B will interact with Animal A and Animal C
    3. Animal C will interact with Animal A and Animal B
- *Immediately* after an interaction, if an Animal's health is below or equal to 0, that Animal will be removed from the Cage (don't worry, the zookeepers are just removing it for precautionary reasons).

public String[] listAnimalsSorted()

This method will return the array of Animal's toString representations in sorted order. This sorting order is based off of an Animal's health (an Animal with lower health will come before an Animal with higher health). There is no need to write your own sorting algorithm (for this assignment at least), Java allows us to sort with a utility class called *Arrays.* The Arrays utility sorts in ascending order, based off of the compareTo method you wrote earlier. To sort, it is as easy as typing

Arrays.sort(arrayToSort)

The sorting utility *only sorts full arrays, however.* Since our array is allowed to have null elements, you will have to translate your Animal array into a temporary array with no null elements before calling the sort utility.

Specification for the CageTest Class

This class will consist of JUnit tests for your Cage class. It is up to you to come up with the expected behavior. You will be required to test the following cases:

1. Write a test that checks if a CageException is thrown when you create a cage with a capacity of 0.
2. Write a test that checks if getCapacity behaves correctly when you create a cage with capacity of 5.
3. Write a test that adds a new Bengal to a valid Tiger Cage, check if the first cell contains that Bengal.
4. Write a test that adds a Bengal and a Sheep to a valid Cage of Animals, check if they are in the first two cells.
5. Write a test that checks if a CageException is thrown when you remove an Animal that does not exist in a valid Cage.
6. Write a test that checks if a CageException is thrown when you add an Animal with the same name to a valid Cage twice.
7. Create a Tiger with 10 health, a Bengal with 5 health and a Sheep with 20 health and put them in a valid Animal Cage. Write a test that checks if the health of the Animals in the Cage after one round of simulation is correct according to the simulation rules.

You will find the JUnit methods "assertEquals" to be helpful when performing some of these tests. Also, make sure to use the "@Test" annotation above each test method. Finally, when writing a test that is testing for some kind of exception, be sure to use the proper annotation. For example, if you are testing for a RuntimeException, the annotation should be written like this: @Test(expected=CageException.class)

Specification for the CageClient Class

In this class, you will write code that sorts and simulates the state of a Cage. This class will only use the "main" method. It will collect user input using a Scanner and print messages via the Eclipse console.

First, print a message that asks the user to input the size of an Animal Cage. If the user inputs a number less than one, the program will ask the user to try again. This command should create a new Cage for holding Animals.

Next, print a message that asks the user to input an Animal. A valid input for an Animal is **typeOfAnimal, name, initialHealth**. Then you should parse the input and ensure that the input is formatted correctly. If the input is not formatted correctly, an invalid Animal type is given, the cage has reached its capacity, or the user tries to add a duplicate name, the program should inform the user of this and ask for input once again. The programe should continue to prompt the user for input until the user decides to begin the simulation by entering "s".

After the user has enter "s", the Cage will simulate one round of interaction and output a list of the Animals in sorted order.