Westeros Nobility Catalog

1 Introduction

This assignment will cover inheritance, simple polymorphism, and simple lists.

2 Problem Description

In the present day Westeros, the only Nobility are the Noble Houses. The DragonLords of old have either been wiped out or banished, but they share a lot of similarities with Houses. In this homework you will implement multiple classes that inherit from their parent classes, so that the methods properly work with the driver that has been provided.

3 Solution Description

Westeros.java (DO NOT CHANGE THIS FILE) will be provided in this homework assignment. This is a driver (where the main method is) and will handle all the command line input, as well as displaying the methods that you will actually implement. You will personally be creating the following items:

- NobilityList.java
- Nobility.java
- DragonLord.java
- House.java
- NorthHouse.java
- WesterlandHouse.java
- ColdResistance.java

Below are the specific requirements that you will have to meet for each class. Be sure to follow the below instructions carefully! Additionally, for those of us that aren't Game of Thrones savvy, some basic descriptions of what each class is supposed to represent are included. Not that these descriptions are only for your benefit of understanding how all of the classes are connected to each other and are not supposed to be additional requirements for you to follow. The things you are supposed to account for are itemized below each description.

1. `NobilityList`
   This is a "list" of Nobility in Westeros. The purpose of this homework is to create a program that stores all of the data of each of the Nobility into a "catalogue" of sorts for viewing.

   - Instance variables:
     - An array that holds the nobilities in the list
     - A counter variable that represents how many nobility are in the list
   - A constructor that sets the backing array to an initial size passed in by the parameter.
   - A method called `listNobilities` that lists the Nobilities inside of the `nobility` backing array. It lists the nobilities by printing out to the console. This method should NOT ever print `null`.
   - A method called `add` that takes in an instance of a `Nobility` to add to the backing array. If the backing array has empty space the instance should be added to the next available spot in the array. If there is not an available space in the array you should take care of this properly by doubling the size of the array and appropriately adding the new instance.

2. `Nobility`

   - Instance variables:
     - An appropriate representation of a name for the Nobility
   - A setter method to change the Nobility's name.
   - A properly overridden `toString` method that returns a string representation of the Nobiliy. This can just return the `name` + "."

3. `DragonLord`
   "In the *A Song of Ice and Fire* novels, "dragonlords" were the titles for members of the forty noble families that vied to actually rule the Valyrian Freehold. They were called dragonlords because they could ride and control dragons. They controlled dragons with whips, horns and sorcery. They were strikingly (some say inhumanly) beautiful. It had long been the custom amongst the dragonlords of Valyria to wed brother to sister, to keep the bloodlines pure.

   The Targaryens were a minor dragonlord family, far from the most powerful, but are the only family that survived the Doom of Valyria with dragons. Thus members of House Targaryen are known as dragonlords in Westeros, even after the death of all the dragons." Read more here. You are responsible for implementing the following class description:

- This class has an "is-a" relationship with `Nobility`
- Instance variables:
  - An appropriately-typed variable `isFireResistant` that represents whether or not this instance of a `DragonLord` is fire resistant
  - An appropriately-typed counter that represents how many dragons an instance of a `DragonLord` has called `dragonCount`
- A constructor that takes in a name representation of the DragonLord, whether or not the DragonLord is fire resistant, and how many dragons the DragonLord has. This constructor should not repeat code that has already been written.
- A properly overridden `toString` method that returns a string representation of the `DragonLord` instance (see examples for formatting and what to include).

4. `House`
   "The Seven Kingdoms of Westeros are divided between many hundreds of noble houses of various sizes. In Westeros ultimate power derives from the King on the Iron Throne and descends through the Great Houses that rule the constituent regions of the continent to their vassals.

   There are nine Great Houses, and each of them have a number of vassal houses (sometimes also referred to as lesser houses) in their liege. The most powerful vassal houses may themselves field armies of a few thousand and control large regions, while the smallest houses may be little more than impoverished landholders with only a few men to their name. Members of the nobility are called "highborn", in contrast to lowborn commoners." Read more here. You are responsible for implementing the following class description:

   - This class has an "is-a" relationship with `Nobility`
   - Instance variables:
     - An appropriately typed variable, `words`, that represents the house's "catch phrase" so to speak.
     - A variable `knightCount` that represents how many knights the class has.
   - A constructor that takes in the the name, words, and knightCount associated with the `House` instance and sets associated instance data. This constructor should not repeat code that has already been written.
   - A properly overridden `toString` method that returns a string representation of a house in the following format: "[House Name]: [words]. Has [number of knights] knights."

5. `NorthHouse`
   "The North is one of the constituent regions of the Seven Kingdoms, and was formerly a sovereign nation known as the Kingdom of the North before the Targaryen conquest.

   The North is ruled from the castle of Winterfell by House Bolton following the fall of House Stark during the War of the Five Kings. It is the largest of the nine major regions of the continent, almost equal in size to the other eight combined." Read more here. You are responsible for implementing the following class description:

- This class has in "is-a" relationship with `House`
- Instance variables:
  - A `coldResistance` variable that represents the resistance to cold for the house. Use the `ColdResistance` enum.
  - An appropriately typed instance variable that describes the house's ability to warg called `wargAbility`
- A constructor that takes in and sets all appropriate instance data. This constructor needs to make use of code that is already written.
- A properly overridden `toString` method that returns a string representation of a `NorthHouse` instance. Please see the example output at the end of this assignment description for proper formatting.

6. `WesterlandHouse`
   "The Westerlands is one of the constituent regions of the Seven Kingdoms. It was formerly a sovereign realm known as the Kingdom of the Rock before the Targaryen conquest.

   The Westerlands are ruled from the castle of Casterly Rock by House Lannister. It is one of the smaller regions of the Seven Kingdoms, but is immensely rich in natural resources, particularly metals. Predominantly mountainous, the hills of the Westerlands are riddled with veins of gold and silver, the mining of which has made the Lannisters and their bannermen immensely rich. While the Lannister armies are not as huge as those of the Reach, they are the best-equipped in the realm, with heavily-armored soldiers and cavalry." Read more here. You are responsible for implementing the following class description:

   - This class has an "is-a" relationship with `House`
   - Instance variables:
     - A `goldCount` variable that represents how much gold an instance of the house has. For our purposes you either have a gold coin, or you don't. No such thing as 1.5 gold coins!
   - A constructor that takes in and sets all appropriate instance data. This constructor needs to make use of code that is already written.
   - A properly overridden `toString` method that returns a string representation of a `WesterlandHouse` instance. Please see the example output at the end of this assignment description for proper formatting.

7. `ColdResistance`

   - This is an enum
   - It should have LOW, MEDIUM, and HIGH values.

4 Example Output

This is example code of adding a House, a DragonLord, a NorthHouse, and a WesterlandHouse.

```
$ java Westeros
Welcome to the Westeros!

Which option would you like?
0. List the nobilities
1. Add a nobility
2. Exit

0

1. DragonLord Blackfyre. Has 0 dragons. Isn't fire resistant.
2. House Baratheon: Ours is the fury. Has 10000 knights.
3. Westerland House Lannister: Hear me roar. Has 20000 knights. Has 1000000 gold coins.
4. North House Mormont: Here we stand. Has 2000 knights. Has MEDIUM coldResistance. Does not
    have warg abilities.

Welcome to the Westeros!

Which option would you like?
0. List the nobilities
1. Add a nobility
2. Exit

1

Which type of nobility would you like to add?
0. DragonLord
1. House
2. NorthHouse
3. WesterlandHouse

2

Please enter their name.
Stark

Please enter their words.
Winter is coming

Please enter the number of knights they have.
18000

Please enter their cold resistence (l/m/h)
h

Please enter whether they can warg (y/n).
y

Welcome to the Westeros!

Which option would you like?
0. List the nobilities
1. Add a nobility
2. Exit

0

1. DragonLord Blackfyre. Has 0 dragons. Isn't fire resistant.
2. House Baratheon: Ours is the fury. Has 10000 knights.
3. Westerland House Lannister: Hear me roar. Has 20000 knights. Has 1000000 gold coins.
4. North House Mormont: Here we stand. Has 2000 knights. Has MEDIUM coldResistance. Does not
    have warg abilities.
5. North House Stark: Winter is coming. Has 18000 knights. Has HIGH coldResistance. Has warg
    abilities.

Welcome to the Westeros!
```

```
Which option would you like?
0. List the nobilities
1. Add a nobility
2. Exit

2

$
```

5 Tips

- Look at the `Westeros.java` driver provided (but don't edit it!) and see what methods are being called when the user selects certain options.
- Write out a diagram of the inheritance structure before you start the homework! Knowing how all the classes interact with each other will make the implementation much simpler!
- Make private instance variables available through getters and setters!

6 You are not Allowed to Use or do the Following

Please note that what you may not be allowed to use or do is not limited to this list. But since there have been a lot of instances of people not being aware of what was allowed and not here are some definite no-no's. Remember it is your responsibility to keep up with piazza questions!

- `Arrays.java`, `ArrayList.java`, any other `Collection`, `System.arraycopy`, etc. Nothing that trivializes array manipulation
- Exception handling. This means no try/catch.
- Don't use System.exit(). Not sure why you would use this, but nonetheless!

Basically, if you haven't learned about it in class, you should ask before using it.

7 Checkstyle

You must run checkstyle on your submission. The checkstyle cap for this assignment is **50** points. Review the Style Guide and download the Checkstyle jar. Run Checkstyle on your code like so:

```
$ java -jar checkstyle-6.2.2.jar -c cs1331-checkstyle.xml *.java
Audit done. Errors (potential points off):
0
```

The message above means there were no Checkstyle 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.

The Java source files we provide contain no Checkstyle errors. For this assignment, there will be a maximum of **50** points lost due to Checkstyle errors (1 point per error). In future homeworks we will be increasing this cap, so get into the habit of fixing these style errors early!

8 Javadocs

For this assignment you will be commenting your code with Javadocs. Javadocs are a clean and useful way to document your code's functionality. For more information on what Javadocs are and why they are awesome, the online documentation for them is very detailed and helpful.

You can generate the javadocs for your code using the command below, which will put all the files into a folder called javadoc:

```
$ javadoc *.java -d javadoc
```

The relevant tags that you need to have are `@author`, `@version`, `@param`, and `@return`. Here is an example of a properly Javadoc'd class:

```
import java.util.Scanner;

/**
 * This class represents a Dog object.
 * @author George P. Burdell
 * @version 13.31
 */
public class Dog {

   /**
    * Creates an awesome dog (NOT a dawg!)
    */
   public Dog(){
      ...
   }

   /**
    * This method takes in two ints and returns their sum
    * @param a first number
    * @param b second number
    * @return sum of a and b
    */
   public int add(int a, int b){
      ...
   }
}
```

Take note of a few things:

1. Javadocs are begun with `/**` and ended with `*/`.
2. Every class you write must be Javadoc'd and the `@author` and `@version` tag included.

The comments for a class start with a brief description of the role of the class in your program.

3. Every non-private method you write must be Javadoc'd and the `@param` tag included for every method parameter. The format for an `@param` tag is `@param <name of parameter as written in method header> <description of parameter>`. If the method has a non-void return type, include the `@return` tag which should have a simple description of what the method returns, semantically.

8.1 Javadoc and Checkstyle

You can use the Checkstyle jar mentioned in the following section to test your javadocs for completeness. Simply add -j to the checkstyle command, like this:

```
$ java -jar checkstyle-6.2.1.jar -j *.java
Audit done. Errors (potential points off):
0
```

There will be a javadoc cap of 10 points for this homework. Please javadoc and checkstyle as you go. Note that you can lose a total of 60 points just for style errors.

9 Turn-in Procedure

Non-compiling or missing submissions will receive a zero. NO exceptions

Submit all of the Java source files you modified and resources your program requires to run to T-Square. Do not submit any compiled bytecode (`.class` files) or the Checkstyle jar file. When you're ready, double-check that you have submitted and not just saved a draft.

Please remember to run your code through Checkstyle!

9.1 Verify the Success of Your Submission to T-Square

Practice safe submission! Verify that your HW files were truly submitted correctly, the upload was successful, and that the files compile and run. It is solely your responsibility to turn in your homework and practice this safe submission safeguard.

1. After uploading the files to T-Square you should receive an email from T-Square listing the names of the files that were uploaded and received. If you do not get the confirmation email almost immediately, something is wrong with your HW submission and/or your email. Even receiving the email does not guarantee that you turned in exactly what you intended.

2. After submitting the files to T-Square, return to the Assignment menu option and this homework. It should show the submitted files.

3. Download copies of your submitted files from the T-Square Assignment page placing them in a new folder.

4. Recompile and test those exact files.

5. This helps guard against a few things.

   (a) It helps insure that you turn in the correct files.

   (b) It helps you realize if you omit a file or files. [1] (If you do discover that you omitted a file, submit all of your files again, not just the missing one.)

   (c) Helps find last minute causes of files not compiling and/or running.

[1] Missing files will not be given any credit, and non-compiling homework solutions will receive few to zero points. Also recall that late homework will not be accepted regardless of excuse. Treat the due date with respect. The real due date is midnight. Do not wait until the last minute!