Homework 6

1 Introduction

In the world of the Divergent series people have normal attributes such as a name and age. In addition to this, people are associated with factions that require that they possess different skills. It will be your job to create classes to represent these different types of people.

2 Problem Description

You will need to create a class hierarchy with the classes listed below in accordance with the descriptions we provide for each class. Like the past assignment, you will not need to explicitly make a main method, however it is highly encouraged that you do so for testing purposes.

There is A LOT here, and it is easy to get lost. Make sure you read everything *carefully*, write down how you think each class should look at a high level on paper before diving into the code, and above all else **start early**. We will be grading based off of the functionality described in this document as well as good coding and OOP principles. If there's something not in here, use your best judgment to handle it ensuring that your code does not crash in doing so.

3 Solution Description

Here is the full listing of classes/Enums that you need to implement:

- Person
- Faction
- Factioned
- Factionless
- Divergent
- Dauntless
- Erudite

The first thing you should do is to read through the next sections carefully and draw out a hierarchy tree (as shown in class and recitation) about how these classes should inherit from one another. Figure out what some classes have in common with others; you will be graded based on your ability to limit code reuse by applying inheritance correctly!

3.1 Faction

This is actually going to be an Enum, not a class. You may put it in a separate file if you wish (that's probably the right way to go as far as organization is concerned anyway) or in the Person class.

This future world heavily relies on the categorization of people into categories known as factions. These factions are most prominently based on ones family line and/or a mental test taken when people become of a certain age. This enum should hold the factions that people have the ability to choose from which include AMITY, ABNIGATION, DAUNTLESS, EURIDITE, and CANDOR.

3.2 Person

This class should be abstract (why?). As usual, there are a few things that are common to all people, including: a first, last name, and an age. Important pieces of functionality to implement in this class are: an overriding equals and toString method (**if you get any errors in checkstyle or otherwise involving hashCode() you may ignore them**). Two people are considered equal if their first, last name and age are equal. This class should also be able to compare itself to other members of the Person class (what interface have we learned about that will allow this?). Instances from the Person class should first be compared by age (I.E. someone older is "greater than" someone younger), then by last name, then by first name.

3.3 Factioned

This class should be asbtract and a subclass of Person. The factioned and the factionless live very different lives and therefore should be modeled differently. Most people are born into a faction which will be represented by the Faction passed into the constructor (note: they are also born with a first, last name and age too). You should be able to retrieve a person's Faction at any time but not change it.

3.4 Factionless

This class should be a subclass of person but *not* abstract (why?). The Factionless are the outcasts of this society. Factionless have a danger level between 0 and 50 (inclusive) that should be able to be retrieved and set as needed as well as passed into a constructor.

3.5 Divergent

This should be a subclass of Factionless. In extremely rare cases members of society are found to be whats known as divergent. Divergents have the ability to represent multiple factions, which should passed in as an array of Factions when created (assume it will not be null). Not much known about these type of people because they often go into hiding within other factions or are killed. For this reason each Divergent has a unique percent chance of being found. This should be calculated by obtaining a random number between 0 and 0.5 and multiplying it by the amount of factions a Divergent can represent. If this number ever goes above 1, create a new identity for them by changing their first and last names (we don't care how you do this) and reset it to 0. You should create a method that changes this value again when called based on these rules (create a new random number and follow the multiplication as before).

Since there are so few in existence the Divergent class should keep a record of how many Divergents have been created and be able to return this at any time (this something that belongs to the Divergent class and not any individual instance).

3.6 Dauntless

This should be a subclass of Factioned. Members of Dauntless are extreme members of society that like to run fast, jump high and live on the edge. As such, they need to be able to represent both their agility and endurance in some fashion (an integer value from 0-10 is sufficient). You should be able to access any of these attributes, but not directly change them outside the class (with the exception of what is described below).

In order to pass their initiation tests, members of Dauntless must go through grueling trials of endurance and agility. These trials often cause competition between members of the faction. You should have some method that is called on a Dauntless object and takes in an instance of Dauntless (what other method do we know has a structure like this?) and puts them through a competition. This method must return the Dauntless member that won the competition (think of some creative way to declare a winner that utilizes Dauntless members attributes). How will you handle the case where the object that the method is called on needs to be returned (what keyword do we know refers to the current object being acted upon)?

In addition, each member of Dauntless must keep track of a rival. The rival is set when the faction member loses in a competition; the person they lose to becomes their rival. You should be able to access a members rival at any time, and if they currently have none simply returning null will suffice.

Finally, members of Dauntless train hard. You should have some way for a member of Dauntless to increase their agility and endurance separately by some fixed amount. You should also print something out to the console when a member trains. Perhaps a former loser of a competition will be able to defeat their rival after a little training!

3.7 Erudite

This should be a subclass of Factioned. Erudites are known as the most intellectual of all the factions. They are highly logical thinkers and greatly invested in the faction system upon which this society has be founded. All Erudites have some sort of degree so they should be able to be created with separate fields of their university and major. These values should able to be changed and returned since one has the ability to transfer between schools and change majors. You should also revise the equals and toString methods in this class to be unique to Erudites. The equals method should test for the school and major in addition to the parameters tested in the person class.

Finally, Erudites love to read books. On instantiation, an array of book titles should be passed into a new Erudite and saved as an instance variable (don't worry about null being passed in). Erudites are compared based on how many books they read (I.E. an Erudite who reads more books is "greater than" one who reads less).

4 Testing Your Code

Since it is so strongly encouraged to test your code for this assignment, we are going to give you a few pointers for how to write your own "TestDriver" class for this assignment. First off, there's really no need to take user input if you are just testing to see if you've modelled your classes correctly (we assume by now you are more than comforatble with getting user input). Second, you'll want to make sure that you only test *one thing* at a time. If you try to run a bunch of tests at once it may be difficult to tell what actually caused the problem. Third, once you write a few lines of code that test a certain piece of functionality, you'll want to save them for later to make sure they still work correctly after you finish more of the assignment. Save any tests you write as helper methods in your "TestDriver" and only call the ones you need when you actually move to the testing phase of the assignment. These are three principles you should utilize for testing any programming assignment you write.

As far as this specific assignment goes, there are a few things you'll definitely want to make sure work. Create an instance of a child class and make sure you can call methods and return data from parent classes; this should tell you if you have your inheritance heirarchy syntactically correct. Create a few Dauntless objects and make sure they compete correctly; make sure you check their rivals before and after! Create a Divergent object and force their "found percentage" to be above 1 and see what happens. If their identity doesn't change as we described take a second look at your Divergent class before putting the randomness back in. Create a Person object and make sure the *only* things you do with them are things that all people can do as described in the document.

You'll notice there's a pattern with most of these examples. Make an object of a class, pass in some data to it that you can control, and call a method. Most tests you write will be of this form if you've organized your classes nicely. Please don't overlook this very important step in programming, testing is critically important!

5 Tips

- This is your longest assignment yet by far. Start early; use it to practice for the test and coding exam! We won't be able to help you much if you come into Office Hours a day or two before it is due.
- **It is critically important that you write a main method to test your code**. It will be very difficult to get everything right if you don't test it all.
- Don't forget about your debugging tools! jGrasp, Eclipse, IntelliJ and other IDEs have fantastic debugging tools that were reviewed in class. These tools will be *incredibly* helpful for this assignment, and if you come into Office Hours for a question that can be solved by the debugger, we will point you in that direction.
- Remember the difference between static and non-static!
- What visibility modifiers allow for you to easily access instance fields from parent classes?
- You will be graded on limited code reuse and good OOP practices. Use the class hierarchy to you advantage when implementing methods deeper in the hierarchy.

6 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.

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:

```java
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 begin 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.

6.1 Javadoc and Checkstyle

Be sure you download the updated Checkstyle file (6.2.1) from T-Square along with this assignment. It will be used to verify both your Checkstyle and Javadocs. Javadocs will count towards your grade on this assignment.

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
```

7 Checkstyle

Checkstyle counts for this homework. You may be deducted up to 40 points for having Checkstyle errors in your assignment. Each error found by Checkstyle is worth one point. This cap will be raised next assignment. Again, the full style guide for this course that you must adhere to can be found by clicking **here**.

Come to us in office hours or post on Piazza if you have specific questions about what Checkstyle is looking for and how to fix Checkstyle errors.

First, make sure you download the `checkstyle-6.2.1-all.jar` from the T-Square assignment page. Then, make sure you put this file in the same directory (folder) as the `.java` files you want to run Checkstyle on. Finally, to run Checkstyle, type the first line into your terminal while in the directory of your Java files and press enter.

```
$ java -jar checkstyle-6.2.1.jar *.java
Starting audit...
Audit done.
```

The message above means there were no Checkstyle errors. You can easily count Checkstyle errors by piping the output of Checkstyle through `wc -l` and subtracting 2 for the two non-error lines printed above (which is how we will deduct points). For example:

```
$ java -jar checkstyle-6.2.1.jar *.java | wc -l
     2
```

Alternatively, if you are on Windows, you can use the following instead:

```
C:\> java -jar checkstyle-6.2.1.jar *.java | findstr /v "Starting audit..." | findstr /v "Audit
    done" | find /c /v "hashcode()"
0
```

8 Turn-in Procedure

Submit all of your `.java` files on T-Square as an attachment. **Note: if you wrote a main method for testing, you are not required to submit it**. 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.

9 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 8PM Thursday. Do not wait until the last minute!