CS2110 Fall 2018 Assignment A1
PhD Genealogy

Website http://genealogy.math.ndsu.nodak.edu contains the PhD genealogy of over 232,000 mathematicians and computer scientists, showing their PhD advisors and advisees. Gries traces his intellectual ancestry back to Gottfried Wilhelm Leibniz (1646–1716), who dreamed of a "general method in which all truths of the reason would be reduced to a kind of calculation". Leibniz foresaw symbol manipulation and proofs as we know them today. On this website, the tree of Anne Bracy goes back only 8 people to Wallace Cassell in 1928 because Cassell's PhD thesis was not in CS or Math and his advisor was not added to the genealogy website. See the Assignments page of the course website for these trees.

It's a laborious, error-prone task to search the genealogy website by hand and construct a tree of someone's PhD ancestors, so we wrote a Java program to do it. It uses a class PhD much like the one you will build, but it has many more fields because of the complexity of the information on that website. The program also uses a class that allows one to read a web page. At the appropriate time, we may show you this program and discuss its construction, so you can learn how to write programs that crawl webpages. Another benefit of 2110!

Your task in this assignment is to develop a class `PhD` that will maintain information about the PhD of a person and a JUnit class `PhDTest` to maintain a suite of test cases for class `PhD`. An object of class `PhD` will contain a PhD's name, date of the PhD, the PhD's known advisors, and the number of known advisees of the PhD.

The term PhD is not used in all countries! Gries's degree is a *Dr. Rerum Natura* from MIT (Munich Institute of Technology). It is abbreviated *Dr. rer nat*, which Gries speaks as *rare nut*. In A1, we use only the term PhD.

Your last task before submitting the assignment will be to tell us how much time you spent on this assignment, so please keep track. This will allow us to publish the mean, median, and maximum times, so you have an idea how you are doing relative to others. It also helps us ensure that we don't require too much of your time in this course.

Learning objectives

- Gain familiarity with the structure of a class within a record-keeping scenario (a common type of application)
- Learn about and practice reading carefully.
- Work with examples of good Javadoc specifications to serve as models for your later code.
- Learn about our code presentation conventions, which help make your programs readable and understandable.
- Practice incremental coding, a sound programming methodology that interleaves coding and testing.
- Learn about and practice thorough testing of a program using JUnit5 testing.
- Learn about *class invariants*.
- Learn about *preconditions* of a method and the use of the Java assert statement for checking preconditions.

The methods to be written are short and simple; the emphasis is on "good practices", not complicated computations.

Reading carefully

At the end of this document is a checklist of items for you to consider before submitting A1, showing how many points each item is worth. Check each item *carefully*. A low grade is almost always due to lack of attention to detail, to sloppiness, and to not following instructions —not to difficulty in understanding OO. At this point, we ask you to visit the webpage on the website of Fernando Pereira, research director for Google:

http://earningmyturns.blogspot.com/2010/12/reading-for-programmers.html

Did you read that webpage carefully? If not, read it now! The best thing you can do for yourself —and us— at this point is to read carefully. This handout contains many details. Save yourself and us a lot of anguish by carefully following all instructions as you do this assignment.

Managing your time

Please read JavaHyperText entry *Study/work habits* to learn about time management for programming assignments. This is important!

Collaboration policy

You may do this assignment with one other person. If you are going to work together, then, as soon as possible —and certainly before you submit the assignment— get on the course CMS and form a group. Both people must do something before the group is formed: one invites, the other accepts.

If you do this assignment with another person, you must *work together*. It is against the rules for one person to do some programming on this assignment without the other person sitting nearby and helping. You should take turns "driving" —using the keyboard and mouse. If you are the weaker of two students on a team and you let your teammate do more than their share, you are hurting only yourself. You can't learn without doing.

With the exception of your CMS-registered partner, you may not look at anyone else's code, in any form, or show your code to anyone else, in any form. You may not look at solutions to similar previous assignments in 2110. You may not show or give your code to another person in the class. While you can talk to others, your discussions should not include writing code and copying it down.

Getting help

If you don't know where to start, if you don't understand testing, if you are lost, etc., please SEE SOMEONE IMMEDIATELY —an instructor, a TA, a consultant. Do not wait. A little in-person help can do wonders.

Using the Java assert statement to test preconditions

A *precondition* is a constraint on the parameters of a method, and it is up to the user to ensure that method calls satisfy the precondition. If a call does not satisfy the precondition, the method can do *anything*.

However, especially during testing and debugging, it is useful to use Java assert statements at the beginning of a method to check that preconditions are true. For example, if the precondition is "this person's name is at least two characters long", use an assert statement like the following (using variable `name` for the field):

```
assert name != null  &&  name.length() >= 2;
```

The additional test `name != null` is important! It protects against a null-pointer exception, which will happen if the argument corresponding to `name` in the call is **`null`**. [This is important! Read it again!]

In A1, all preconditions of methods must be checked using assert statements in the method. Where possible write the assert statements as the first step in writing the method body, so that they are always there during testing and debugging. Also, when you generate a new JUnit class, make sure you use JUnit5 (Jupiter) and make sure the VM argument -ea is present in the Run Configuration. Assert statements are helpful in testing and debugging.

How to do this assignment

Scan the whole assignment before starting. Then, develop class `PhD` and test it using class `PhDTest` in the following incremental, sound way. This methodology will help you complete this (and other) programming tasks quickly and efficiently. If we detect that you did not develop it this way, points will be deducted.

1. Create a new Eclipse project, called `a1`. In `a1`, create a new class, `PhD`. It *must*, repeat *must* be in the default package, and it does not need a method `main`. Insert the following as the first lines of file `PhD.java`:

```
/** NetId: nnnnn, nnnnn. Time spent: hh hours, mm minutes.
    An instance maintains info about the PhD of a person. */
```

Remove the constructor with no parameters (if it is there), since it will not be used and its use can leave an object in an inconsistent state (see below, the class invariant).

2. In class `PhD`, declare the following fields (you choose the names of the fields), which will hold information describing a person with a PhD. Make these fields private and properly comment them (see the "class invariant" section below).

   Note: You *must* break long lines (including comments) into two or more lines so that the reader does not have to scroll right to read them. A good guideline: No line is more than 80 characters long.

   - name (a `String`). Name of the person with a PhD, a String of length > 1.
   - year PhD was awarded (an **`int`**). Can be any integer.
   - month PhD was awarded (an **`int`**). Month PhD awarded, in range 1..12 with 1 being January, etc.
   - first advisor of this person (of type `PhD`). The first PhD advisor of this person —null if unknown.
   - second advisor of this person (of type `PhD`). The second advisor of this person —null if unknown or if the person has less than two advisors.
   - number of PhD *advisees* of this person.

   About the field that contains the number of advisees: The user *never* gives a value for this field; it is completely under control of the program. For example, whenever a `PhD p` is given an advisor `m`, `m`'s number of advisees must be increased by 1. *It is up to the program, not the user, to increase the field*. This is similar to your GPA being updated when a faculty member inputs your grade for a course on Cornell's system.

   THIS IS IMPORTANT. Do NOT misinterpret the number of advisees as the number of advisors. *You will lose a lot of points*. This has happened in the past, due to lack of careful reading. My advisees are those I am advising; my advisor is the person who advised me. This is important, repeat, important.

   The class invariant. Comments must accompany the declarations of all fields to describe what the fields mean, what legal values they may contain, and what constraints hold for them. For example, for the name-of-the-person field, state in a comment that the field contains the person's name and must be a string of at least 2 characters. The collection of the comments on these fields is called the *class invariant*. Here is an example of a declaration with a suitable comment. Note that the comment does *not* give the type (since it is obvious from the declaration), it does not use noise phrases like "this field contains …", and it *does* contain constraints on the field.

   ```
   int month; // month PhD was awarded. In 1..12, with 1 meaning January, etc.
   ```

   Note again that we did not put "(an int)" in the comment. That information is already known from the declaration. Don't put such unnecessary information in comments.

   Whenever you write a method (see below), look through the class invariant and convince yourself that the class invariant still holds when the method terminates. This habit will help you prevent or catch bugs later on.

3. In Eclipse, start a new JUnit test class and call it `PhDTest`. You can do this using menu item **File —> New —> JUnit Test Case** (always add the "New Unit Jupiter test", also called *JUnit5*, if asked).

4. Below, we describe four ***groups*** A, B, C, and D of methods. Work with *one* group at a time, performing steps (1)..(4). **Don't go on to the next group until the group you are working on is thoroughly tested and correct.**

   (1) Write the Javadoc specifications for each method in that group. They must be complete and correct —you can copy-and-paste from the specs we give below to make it easy, in fact we urge you to copy-and-paste. To learn more about javadoc, read the JavaHyperText entry for "javadoc".

   (2) Write the method bodies, starting with assert statements (unless they can't be the first statement) for the preconditions.

   (3) Write *one* test procedure for this group in `PhDTest` and add test cases to it for all the methods in the group.

(4) Test the methods in the group thoroughly. Note: *Do not deal with testing that assert statements are correct until step 5*.

Discussion of the groups of methods. The descriptions below represent the level of completeness and precision required in Javadoc specification-comments. In fact, you should copy and paste these descriptions to create the first draft of your Javadoc comments. Copy-and-paste is the easier way to adhere to the conventions we use, such as using the prefix "Constructor: …" and double-quotes to enclose an English boolean assertion.

Method specs do not mention fields because the user may not know what the fields are, or even if there are fields. The fields are private. Consider class `JFrame`: you know what methods it has but not what fields, and the method specs don't mention fields. In the same way, a user of class `PhD` will know the methods but not the fields.

The names of your methods must match those listed below exactly, including capitalization. The number of parameters and their order must also match: any mismatch will cause our testing programs to fail and will result in loss of points for correctness. Parameter names will not be tested —change the parameter names if you want.

In this assignment, you may *not* use if-statements, conditional expressions, or loops.

Group A: **The first constructor and the getter/observer methods of class `PhD`**.

Constructor	**Description (and suggested javadoc specification)**	
PhD(String n, int y, int m)	Constructor: an instance for a person with name n, PhD year y, and PhD month m. The advisors are unknown, and there are 0 advisees. Precondition: n has at least 2 chars, and m is in 1..12.	
Getter Method	**Description (and suggested javadoc specification)**	**Return Type**
name()	Return the name of this person.	String
year()	Return the year this person got their PhD.	**int**
month()	Return the month this person got their PhD.	**int**
advisor1()	Return the first advisor of this PhD (null if unknown).	PhD (not String!)
advisor2()	Return the second advisor of this PhD (null if unknown or non-existent).	PhD (not String!)
numAdvisees()	Return the number of PhD advisees of this person.	**int**

Consider the constructor. Based on its specification, figure out what value it should place in each of the 6 fields to make the class invariant true. In `PhDTest`, write a procedure named `testConstructor1` to make sure that the constructor fills in ALL 6 fields correctly. The procedure should: Create one `PhD` object using the constructor and then check, using the getter methods, that all fields have the correct values. Since there are 6 fields, there should be 6 `assertEquals` statements. As a by-product, all getter methods are also tested.

Group B: the setter/mutator methods. Note: methods `addAdvisor1` and `addAdvisor2` must change fields of both this `PhD` and its advisor in order to maintain the class invariant.

When testing the setter methods, you will have to create one or more `PhD` objects, call the setter methods, and then use the getter methods to test whether the setter methods set the fields correctly. Good thing you already tested the getters! Note that these setter methods may change more than one field; your testing procedure should check that *all* fields that may be changed are changed correctly.

We are *not* writing methods that change an existing advisor. This would require if-statements, which are not allowed. Read the preconditions of methods carefully.

IMPORTANT. In `addAdvisor1(p)` ,, p gets a new advisee, so `p`'s number of advisees increases. Do not mistake the number-of-advisees field for the number of advisors. There is a difference. *Get this wrong*? *Lose lots of points*.

Setter Method	Description (and suggested javadoc specification)
addAdvisor1(PhD p)	Add p as the first advisor of this person. Precondition: the first advisor is unknown and p is not null.
addAdvisor2(PhD p)	Add p as the second advisor of this PhD. Precondition: The first advisor is known, the second advisor is unknown, p is not null, and p is different from the first advisor.

Group C: Two more constructors. The test procedure for group C has to create a `PhD` using the constructor given below. This will require first creating two `PhD` objects using the first constructor and then checking that the new constructor sets *all* 6 fields properly —and also the number of advisees of adv1 and adv2.

Constructors	Description (and suggested javadoc specification)
PhD(String n, int y, int m, PhD adv1)	Constructor: a PhD with name n, PhD year y, PhD month m, first advisor adv1, and no second advisor. Precondition: n has at least 2 chars, m is in 1..12, and adv1 is not null.
PhD(String n, int y, int m, PhD adv1, PhD adv2)	Constructor: a PhD with name n, PhD year y, PhD month m, first advisor adv1, and second advisor adv2. Precondition: n has at least 2 chars, m is in 1..12, adv1 and adv2 are not null, and adv1 and adv2 are different.

Group D: Write two functions. The second tests whether two people are "intellectual siblings", that is: they are not the same object and they have a non-null advisor in common. Write these using only boolean expressions (with !, &&, and || and relations <, <=, ==, etc.). *Do not use if-statements, conditional expressions, switches, addition, multiplication, etc*. Each is best written as a single return statement.

Functions	Description (and suggested javadoc specification)	Return type
gotBefore(PhD p)	Return value of: p is not null and this PhD got the PhD before p.	**boolean**
isSiblingOf(PhD p)	Return value of: this PhD is an intellectual sibling of p. Precondition: p is not null.	**boolean**

Note: The Piazza note for Assignment A1 describes test cases for `gotBefore` and `isSiblingOf`.

5. **Testing assert statements**. It is a good idea to test that at least some of the assert statements are correct. To see how to do that, look at item 2 under entry "JUnit testing" in the *JavaHyperText*. It's up to you how many you test. The assert statements are worth a total of 5 points, and there are over 25 different tests one can make, so if you miss 4-5 of them you lose about 1 point.

   There are two places to put tests for assert statements in the JUnit testing class. (1) Put them in the appropriate existing testing method —for example, put tests for assert statements in the first constructor at the end of the testing procedure for the first constructor. (2) Insert a fifth testing procedure to test all (or most of) the assert statements.

   Implement the assert-statement testing using either method (1) or (2). Just make it clear what is being tested where.

6. In Eclipse, click menu item **Project -> Generate Javadoc**. In the window that opens, make sure you are generating Javadoc for project a1, using visibility **`public`** and storing it in a1/doc. Then open doc/index.html. You should see your method and class specifications. Read through them from the perspective of someone who has not read your code. Fix the comments in class `PhD`, if necessary, so that they are appropriate to that perspective. You *must* be able to understand everything there is to know about writing a call on each method from the specification that you see by clicking the Javadoc button —that is, without knowing anything about the private fields. Thus, the fields should not be mentioned. Then, and only then, add a comment at the top of file PhD.java saying that you checked the Javadoc output and it was OK.

7. Check carefully that each method that adds an advisor for a `PhD` updates the advisor's number of advisees. Four methods do this. *Make sure the field contains the number of advisees and not the number of advisors*.

8. Review the learning objectives and reread this document to make sure your code conforms to our instructions. Check each of the following, one by one, carefully. Note that 50 points are given for the items below and 50 points are given for actual correctness of methods. *More points may be deducted if we have difficulty fixing your submission so that it compiles with our grading program*.

    1. 5 Points. Are all lines short enough that horizontal scrolling is not necessary (about 80 chars is long enough). Do you have a blank line before the specification of each method and no blank line after it?
    2. 10 Points. Is your class invariant correct —are all fields defined and all field constraints given?
    3. 5 Points. Is the name of each method and the types of its parameters exactly as stated in step 4 above? (The simplest way to ensure this was to copy and paste!)
    4. 10 Points. Are all specs complete, with any necessary preconditions? Remember, we specified every method carefully and said to copy our specs and paste them into your code. Are they in Javadoc comments?
    5. 5 points. Do you have assert statements in each method that has a precondition to check that precondition?
    6. 5 Points. Did you check the Javadoc output and then put a comment at the top of class `PhD`?
    7. 10 Points. Did you write *one* (and only one) test procedure for each of the groups A, B, C, and D of step 4 and another for assert statements? Thus, do you have 4 or 5 test procedures? Does each procedure have a name that gives the reader an idea what the procedure is testing, so that a specification is not necessary? Did you properly test? For example, in testing each constructor, did you make sure to test that all 6 fields have the correct value? Do you have enough test cases? For example, testing whether one date comes before another date, when each is given by a year and a month, probably requires at least 5 test cases.

9. Change the first line of file PhD.java: replace "nnnnn" by your netids, "hh" by the number of hours spent, and "mm" by the number of minutes spent. If you are doing the assignment alone, remove the second "nnnn". For example, if gries spent 4 hours and 0 minutes, the first line would be as shown below.

```
/** NetIds: djg17. Time spent: 4 hours, 0 minutes.
```

Being careful in changing this line will make it easier for us to automate the process of calculating the median, mean, and max times. Be careful: Help us.

10. Upload files `PhD.java` and `PhDTest.java` on the CMS by the due date. Do not submit any files with the extension/suffix .java~ (with the tilde) or .class. It will help to set the preferences in your operating system so that extensions always appear.