CS2110 Spring 2019 Assignment A1
PhD Genealogy

Website http://genealogy.math.ndsu.nodak.edu contains the PhD genealogy of over 237,000 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. Clarkson's known genealogy doesn't go back that far back; his parents are Myers and Schneider of Cornell, his grandparent is Turing winner Barbara Liskov, and his great grandparent is John McCarthy, Turing award winner and the father of AI. See the Piazza note for Assignment 1 for their PhD 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 Java 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!

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 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.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 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 development, 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

Page 6 contains 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

Read JavaHyperText entry *Study/work habits* to learn about time management for programming assignments.

Statistics on time spent on A1

Last semester, the average and mean time spent on a similar A1 assignment was about 5 hours.

Suggested Time Table for Completing A1
Wed. 30 Jan. Scan the handout. Don't expect to understand everything.

Fri. 01 Feb. Create the Eclipse project in the default package, create class PhD, and put in the field declarations and the class invariant.

Sat. 02 Feb. Create the JUnit class A1Test. Write the methods in group A, put in the necessary test cases, and test/debug until all methods are correct. Complete and test the Group B methods.

Mon. 03 Feb. Complete and test the Group C methods. Complete and test the Group D methods.

Tues. 05 Feb. Put in testing of assert statements, make sure they work properly.

Wed. 06 Feb. Look at point 9 on page 7 of the handout and check everything that is mentioned there, carefully. Submit the project a day early.

Collaboration policy —Academic integrity

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.

Working with a partner requires certain skills that are not always apparent. You may want to watch a YouTube video on *pair programming:* https://www.youtube.com/watch?v=YhV4TaZaB84&feature=youtu.be.

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 one character long", use an assert statement like the following one (using variable `name` for the field):

```
assert name != null  &&  1 <= name.length();
```

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, check all preconditions of methods using assert statements in the method body. 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

Follow the directions here, *one step at a time*. Don't go on to another step until the one you are doing is finished. You will be developing class `PhD` and testing it using class `PhDTest` in an 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. Make sure you set your Eclipse preferences appropriately, as instructed on the following JavaHyperText webpage. The staff will ask you to set them if they see that they are not yet there.

   http://www.cs.cornell.edu/courses/JavaAndDS/eclipse/Ecl01autoformat.html

   Please be sure you set the additional preferences as instructed in pinned Piazza note @80.

2. Create a new Java project in Eclipse, called `a1`. In `a1`, create a new class, `PhD`. Make it public. 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`:

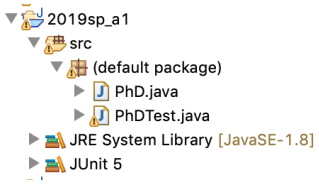
   ```
   /** NetId: nnnnn, nnnnn. Time spent: hh hours, mm minutes. */
   /** An instance maintains info about the PhD of a person. */
   ```

   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 the "class invariant" section below).

   Checking that it is in the default package

   Is there a package statement at the top of PhD.java? If yes, the class is not in the default package.

   When you create the class, a window pops up for information about the class. That window contains a field labeled "package:". Make sure there is NO text in that field. It may give a warning, saying that putting it in the default package is not a good idea. Disregard the warning. Just make sure the package field is empty. Also, your Package Explorer should look as shown to the right for the project (the project name may be different).

3. 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. Before each declaration, put a javadoc comment that gives its meaning and constraints on it (see the "class invariant" section below).

   Note: Eclipse will break long lines automatically for you (except for /* … */ 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 but Eclipse will allow a few more, up to 120.

   - month PhD was awarded (an **`int`**). In range 1..12 with 1 being January, etc.
   - year PhD was awarded (an **`int`**). Can be any integer (to save work, we do not put constraints on it.)
   - name (a `String`). Name of the person with a PhD, a String of length > 0.
   - 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. A javadoc comment must precede each field to describe what the field means, what legal values it may contain, and what constraints hold on the values. For example, for the name-of-the-person field, state that the field contains the person's name and it must be a string of at least 1 character. The collection of the comments on these fields is called the *class invariant*. Here is an example:

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

Note that the comment does *not* give the type, since it is obvious from the declaration, and it does not use noise phrases like "this field contains …". Don't put such unnecessary stuff in comments. The comment *does* contain constraints on the field.

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.

4. In Eclipse, use menu item **File —> New —> JUnit Test Case** to start a new JUnit test class, called `PhDTest`. Add the "New Unit Jupiter test", also called *JUnit5*, if asked. If asked, add the JUnit library to the build path.

5. 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) For each method in the group, c*opy-paste* its specification (from this pdf file) into a Javadoc comment (more information on this below) and then copy-paste the method header underneath the Javadoc comment. Use access modifier public for each method. Note: we consider constructors to be methods.

   (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. Call the four test procedures `testGroupA`, `testGroupB`, `testGroupC`, and `testGroupD`.

   (4) Test the methods in the group thoroughly. Note: *Don't deal with testing that assert statements are correct until step 6*.

Discussion of the groups of methods. The descriptions below represent the level of completeness and precision required in Javadoc specifications. *Copy-paste them into Javadoc comments that precede each method*. The html tag
 means to start a new line. When you copy-paste into a Javadoc comment and save the file, lines will be broken at these tags. If this doesn't happen, you have not properly installed the formatting preferences required for this class into Eclipse. Stop and do so now.

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 match: any mismatch will cause our testing programs to fail and will result in loss of points for correctness. Therefore, copy-paste! Parameter names will not be tested —change them if you want.

Order of components in class PhD. Declarations can be in *any* order in a class. Order doesn't matter. An oft-used convention is to put fields first, then constructors, then getters, then setters, and then other methods. However, put them in any order you want. We won't expect them in any specific order.

In this assignment, you may *not* use if-statements, conditional expressions, or loops. Reason: We are concentrating on writing a class, not a complicated algorithm, and we want you to get practice with boolean expressions.

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

Constructor	**javadoc specification**	
PhD(int m, int y, String n)	Constructor: a PhD with PhD month m, PhD year y, and name n. The advisors are unknown, and there are 0 advisees. Precondition: n has at least 1 char and m is in 1..12.	
Getter Method	**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!)
advisees()	Return the number of PhD advisees of this person.	**int**

Testing these methods. Based on the spec of the constructor, 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 `testGroupA` 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 correct values. Since there are 6 fields, there should be 6 `assertEquals` statements. As a by-product, all getter methods are also tested.

Note: An import statement may have been removed from file `PhDTest.java` so that the first `assertEquals` statement you write is a syntax error. A red x will appear to the left of the line. Hover your mouse over the red x, and a pop-up window with the syntax error will appear: *assertEquals is undefined*. Press the mouse on the red x. A window will open with 2-3 options. Choose the option: *add static import for 'Assertions.assertEquals'*; an import statement will be added near the top of the file and the syntax error will disappear.

If a field's default value is the correct value to make the class invariant true, there is no need to assign it a value in the constructor. Look up entry *default value* in JavaHyperText for the default values.

Group B: The setter/mutator methods. Note: methods `setAdvisor1` and `setAdvisor2` must change fields of both this `PhD` and its new advisor in order to maintain the class invariant.

In class `PhDTest`, create a testing procedure named `testGroupB` and put all testing of these methods in that procedure. 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! These setter methods may change more than one field; your testing procedure should check that *all* fields that may be changed are changed correctly. (It does not have to test fields that don't change.)

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 `setAdvisor1(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.

Setter Method	javadoc specification
setAdvisor1(PhD p)	Make p the first advisor of this person. Precondition: the first advisor is unknown and p is not null.
setAdvisor2(PhD p)	Make p 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. Test procedure `testGroupC` 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`.

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

Group D: Write two functions. Write these functions 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.

Note: Two PhDs are "intellectual siblings" if (1) they are not the same object and (2) they have a non-null advisor in common.

Functions	javadoc specification	Return type
gotAfter(PhD p)	Return value of: this PhD got the PhD after p. Precondition: p is not null.	**boolean**
areSiblings(PhD p)	Return value of: p is not null and this PhD and p are intellectual siblings.	**boolean**

You have to check whether one PHD comes after another, using both years and months, without an if-statement. To do that, write down in English (or Chinese, Korean, Telugu, German, whatever) what it means for date p1 to come after date p2, considering years and months. Do it in terms of > and = (e.g. p1's year = p2's year) and in terms of ANDs and Ors. Once you have that, translate it into a boolean expression.

Checking whether they are the same object. Consider a call `q.areSiblings(p)`. In method `areSiblings`, you have to check whether this object and `p` are the same object. You will learn in lecture 4 that keyword **this**, when it appears in a method in an object, is a pointer to the object in which it appears. So write this check as **this** == p (or **this** != p, depending on what you want). See JavaHyperText entry **this**.

Testing gotAfter

Since the boolean expression will involve relations ==, <, > on years and months, there are actually 9 different test cases, given below. We have found over the years with similar assignments that if we did not test all 9 cases in grading, we did not catch all errors. So our grading program tests them all. You should too.

Here's a tip to help write tests correctly: Give the PhD objects names that contain the month and year, e.g.

```
PhD feb77= new PhD("feb77", 1977, 2);
PhD jun77= new PhD("jun77", 1977, 6);
```

9 test cases for a call q.gotAfter(p)
same year, same month
same year, q's month before p's
same year, q's month after p's
q's year before p's, same month
q's year before p's, q's month before p's
q's year before p's, q's month after p's
q's year after p's , same month
q's year after p's, q's month before p's
q's year after p's, q's month after p's

Testing AreSiblings

Here is the definition of intellectual sibling (from above):

(1) They are not the same object and
(2) They have a non-null advisor in common

Suppose the two PhDs are named A and B. Then at least these test cases are needed:

A and B are the same object
Neither A nor B has an advisor
A.advisor1 is not null and equals B.advisor1
A.advisor1 is not null and equals B.advisor2
A.advisor2 is not null and equals B.advisor1
A.advisor2 is not null and equals B.advisor2
The call A.areSiblings(null) should return FALSE (originally we had a mistake, it said true. Answer is false).

6. **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 between 16 and 24 different tests one can make, so if you write 4-5 of them incorrectly, you lose about 1 point.

   There are two possible places for tests for assert statements in the JUnit testing class: (1) 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) In a fifth testing procedure to test all (or most of) the assert statements.

   Implement the assert-statement testing using either method (1) or (2).

7. Did you copy-and-paste the method specifications? If they were not copy-and-pasted, you may have made a typo on typing or even left something out. Compare your specs to those in this handout.

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

9. 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 may be deducted for the items below. *More points may be deducted if we have difficulty fixing your submission so that it compiles with our grading program*.

   1. 10 points will be deducted if we determine that you have not installed the required Eclipse formatting preferences.

2. 10 points will be deducted if your method specifications are not those that we told you to copy-and-paste. All lines should be at most 80 characters long —more than 120 will cause deductions.
3. 10 Points. Is your class invariant correct —are all fields defined and all field constraints given?
4. 5 Points. Are the class names `PhD` and `PhDTest`? `Phd` and `PHD` are wrong.
5. Is the name of each method and the types of its parameters exactly as stated in step 4 above? Did you copy and paste? Note: the getter methods do *not* have "get" in front of them. E.g. it's `name`(), not `getName`().
6. 5 points. Do you have assert statements in each method that has a precondition to check that precondition?
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 perhaps 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 write all the test cases as required in the discussion of each group above?
8. Deduction by grading program: If a method spec does not have a Precondition, *do not* put an assert statement in its body. If a method spec *does* have a Precondition, *do* put appropriate assert statements in its body.

10. 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.

11. 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.

    Note that CMS allows you to download your submission so that you can verify that you submitted the correct files. Please do that.

Frequently asked questions from the Piazza note for Assignment A1.

Below are the FAQs and answers from the Assignment A1 Piazza note as of 28 January 2019. Any changes to that note will be place at the top, where you will be obvious. You will be notified of changes.

Frequently asked questions and concerns

1. CHECK THIS NOTE FOR TYPICAL ERRORS IN SUBMISSIONS: See Piazza note @11.

2. How many times can I submit on the CMS? Submit as often as you like. Only the last one is graded.
Don't take this as an opportunity to submit every 3 minutes, submitting 15-20 times.
Submit when you know it is done, but if later you see how to do something better, submit again.

3. I get a warning about my method specification because I am not using @param or @return.
What do I do?
Ignore the warnings and delete those @param things for now. We do not need them. Read about method specifications in our Style Guidelines:

http://www.cs.cornell.edu/courses/JavaAndDS/JavaStyle.html

Or, look on the JavaHyperText page and click "Style Guide" in the left part of the top navigation bar.

4. Should we check for situations like a PhD being their own advisor and an advisor being younger than their advisee or a year that is < 0? No. Please implement the spec of each method exactly; do nothing more, nothing less. We are not attempting to create a completely realistic class to be used in a real live situation. We are attempting to teach you about OO with a minimum amount of your time.

5. Can we add other methods to class PHD? Do not add other methods. None are needed.

6. If I want to call constructor 1 from a constructor 2, I can't put assert statements in the beginning of constructor 2. What do I do?
If you are not contemplating calling one constructor from another, which hasn't been taught as of this writing, no need to read further. But the answer to this question appears in Piazza note @12.

7. ABOUT THE NUMBER OF ADVISEES
Suppose my field for the number of advisees is numAdvisees. Since numAdvisees is private, from PhD object p (say), how can I reference/change q.numAdvisees, the number of advisees in PhD object q?
Just do it; you will see that it works. Any method in class PhD can reference private fields and methods of any object of class PhD.

8. May I use fields in PhDTest to reduce the number of declarations needed in methods in this class?
No, don't do that, for at least three reasons.
(1) You don't know enough about JUnit testing to do this right. For example, you don't know the order in which methods are called.
(2) Such fields often create interactions between testing procedures, but those testing procedures should be completely independent of each other.
(3) A declaration may be far from its use, making it more difficult to understand the methods.

Remember, once you write a declaration-assignment, you can easily copy-and-paste it into another method.