**CS2110 Spring 2021 Assignment A1**
**Phd Genealogy**

The website mathgenealogy.org contains the Phd genealogy of over 263,900 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. See the Assignment A1 page of the Canvas website for Gries's tree. Muhlberger is not in the math-CS Phd genealogy website because his Phd was in Physics.

Someone started a new academic family tree —academictree.org— which now has entries for over 75 fields. But it's far behind the Math-CS one, with only 22,000 entries for CS and 34,000 entries for Math. Here's what's known of Muhlberger's tree: academictree.org/physics/tree.php?pid=122011.

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 number of known advisees.

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 and what your thought of this assignment. Please keep track of the time spent on A1. We will publish the mean, median, and maximum times and show you all submitted comments, so your experience compares to others. This 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 tests.
- 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, requiring only assert statements, assignments, and returns. 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 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 proposes, the other accepts.

If you do this assignment with another person, you must *work together*. Usually, we say that 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.

During this pandemic, where you and your partner may not be physically together, we can't require the above. However, you should zoom often and program together, looking at each other's suggestions for a method, say, and agreeing on the best solution. This includes looking at and discussing test cases in the JUnit testing class.

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 (using variable `name` for the field):

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

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 may be deducted.

1. (1) Create a new Eclipse project, called `A1` —actually, you can call it anything you like. If a prompt opens to create `module-info.java`, do **NOT** create it.
   (2) With project `A1` selected in the Package Explorer pane, create a new package. It *must* be called `a1`. When creating the package, do not create package-info.java.
   (3) With package `a1` selected in the Package Explorer pane, create a new class, `Phd`. It should be placed in package `a1`. It does not need a method `main`. The created class should have `package a1;` on the first line.
   (4) Insert the following lines underneath the package statement (copy and paste):

   ```
   /** NetId: nnnnn, nnnnn. Time spent: hh hours, mm minutes.   <br>
       What I thought about this assignment:                   <br><br>
       An instance maintains info about the Phd of a person. */
   ```

   If Eclipse added a constructor, remove it 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, which will hold information describing a person with a Phd. You choose the names of the fields, but read the Style Guide in JavaHyperText on naming variables: https://www.cs.cornell.edu/courses/JavaAndDS/JavaStyle.html#NameVariable ). Make these fields private and properly comment them (see the "class invariant" section below).

   - name (a `String`). Name of the person with a Phd, a String of length > 0.
   - year Phd was awarded (an **`int`**). Must be > 1000.
   - month Phd was awarded (an **`int`**). In range 1..12 with 1 being January, etc.
   - First advisor of this person (of type `Phd`) —null if unknown.
   - Second advisor of this person (of type `Phd`) —null if unknown or only one advisor.
   - number of Phd *advisees* of this person (an **`int`**).

   **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` 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.

   **IMPORTANT.** Do NOT misinterpret the number of advisees as the number of advisors. 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 1 character. The collection of the comments on these fields is called the *class invariant*.

   Use Javadoc comments, placed *before* each field declaration. Here is an example of a declaration with a suitable comment. Note: 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.

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

Note again that we did not put "(an int)" in the comment. That information is already known from the declaration. Don't put such redundant 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 specs for each method in that group. Make sure they are complete and correct —look at the specs we give you below. Copy-and-paste from this handout to make it easy.

   (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. Note: Do NOT use fields in class `PhdTest`; use only local variables.

   (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, it's best to copy and paste these descriptions to create the first draft of your Javadoc comments. Copy-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.

The names of your methods must match those listed below exactly, including capitalization. Make all these methods **`public`**. 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, switches, or loops**.

**Group A**: **The first constructor and the 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, Phd month m. Its advisors are unknown, and it has no advisees.<br>Precondition: n has at least 1 char, y > 1000, and m is in 1..12 | |
| **Observer Method** | **Description (and suggested javadoc specification)** | **Return Type** |
| name() | = the name of this person. | String |
| date() | = the date on which this person got the Phd. In the form "month/year", with no blanks, e.g. "6/2007" | String |
| advisor1() | = the first advisor of this Phd (null if unknown). | Phd (not String!) |
| advisor2() | = the second advisor of this Phd (null if unknown or non-existent). | Phd (not String!) |
| nAdvisees() | = 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 observer methods, that *all* fields have the correct values. As a by-product, all observer methods are also tested.

**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.

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

No method will change an existing advisor. This would require if-statements, which are not allowed. Read 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** | **Description (and suggested javadoc specification)** |
|---|---|
| setAdvisor1(Phd p) | Make p the first advisor of this person.<br>Precondition: the first advisor is unknown and p is not null. |
| setAdvisor2(Phd p) | Make p the second advisor of this person.<br>Precondition: The first advisor (of this person) is known, the second advisor is unknown, p is not null, and p is different from the first advisor. |

**Group C**: Another constructor. 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 a1 and a2.

| **Constructor** | **Description (and suggested javadoc specification)** |
|---|---|
| Phd(String n, int y, int m, Phd a1, Phd a2) | Constructor: a Phd with name n, Phd year y, Phd month m, first advisor a1, and second advisor a2.<br>Precondition: n has at least 1 char, y > 1000, m is in 1..12,<br>a1 and a2 are not null, and a1 and a2 are different. |

**Group D**: Write three functions. The third 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 |
|---|---|---|
| hasNoAdvisees() | = "this Phd has no advisees",<br>i.e. true if this Phd has no advisees and false otherwise. | boolean |
| gotBefore(Phd p) | = "p is not null and this person got the Phd before p." | boolean |
| areSibs(Phd p) | = "this person and p are intellectual siblings."<br>Precondition: p is not null. | boolean |

**Writing gotBefore**. You have to check whether one person got their Phd before the other using both years and months, without an if-statement, addition, multiplication, etc. To do that, write down in English (or Chinese, Korean, Telugu, German, whatever) what it means for date `d1` to come before date `d2`, considering years and months. Do it in terms of < and = (e.g. `d1`'s year = `d2`'s year) and in terms of ANDs and ORs. Once you have that, translate it into a boolean expression.

**Writing areSibs**. Consider a call `q.areSibs(p)`. In method `areSibs`, you have to check whether this object and `p` are the same object. You will learn in a lecture 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**. When testing whether two Phds are the same object (are equal), use == or !=; do NOT use function `equals`.

**Testing gotBefore.** Since the boolean expression will involve relations ==, <, > on years and months, there are 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. Therefore, our grading program tests them all. You should too.

**9 test cases for a call `q.gotBefore(p)`**

| | |
|---|---|
| same year: | `q`'s month before `p`'s, same month, and `q`'s month after `p`'s |
| `q`'s year before `p`'s: | `q`'s month before `p`'s, same month, and `q`'s month after `p`'s |
| `q`'s year after `p`'s: | `q`'s month before `p`'s, same month, and `q`'s month after `p`'s |

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);
```

**Testing areSibs**. 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:

Neither A nor B has an advisor.
A and B are the same object and they have the same non-null first advisor.
A and B are different objects and they have the same first advisor.
A and B are different objects and they have the same second advisor.
A and B are different objects and the first advisor of one is the second advisor of the other.
A and B are different objects and they have different first advisors.
A and B are different objects and they have different second advisors.

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 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 about 15-20 individual tests one can make, so if you miss 4-5 of them you lose about 1 point.

   There are two places to put tests 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. Check that your method specifications are appropriately written in Javadoc comments before the method header. To do this, in class PhdTest, hover your mouse over calls on methods in class Phd. Does the small window that pops up contain the method specification? If not, something is wrong.

7. Check carefully that each method that adds an advisor for a `Phd` updates the advisor's number of advisees. Three methods do this (one is a constructor). *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. *More points may be deducted if we have difficulty fixing your submission so that it compiles with our grading program.*

    1. 5 points: Eclipse formatting preferences are not properly installed.
    2. ≥ 5 points: (1) The classes are not named `Phd` and `PhdTest`, (2) they do not have `package a1;` on the first line, or (3) the method names and signatures are not what they should be.
    3. <= 10 points: Method specs are not complete, with any necessary preconditions, and are not in Javadoc comments that precede the method header. Did you copy-paste the specifications?.
    4. <= 10 points. The class invariant is not correct —not all fields are properly defined, along with necessary constraints, and/or it is not given in Javadoc comments.
    5. 10 points. Your program contains an if-statement, conditional expression, switch statement, etc. Also, method `gotBefore` should not use addition or multiplication.
    6. 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?

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.

In that same comment at the top of class Phd, please add a comment in the appropriate place telling us what you thought of this assignment. We will extract your comments, read them carefully, and show them all to you so you can see what everyone thought of this assignment.

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.