CS2110 Fall 2021 Assignment A1
Monitoring Elephants

Introduction

The website www.worldwildlife.org/elephants tells you that elephants, the largest living land animals, are threatened by shrinking living space and poaching (for their tusks). The site says that elephants are key players in the forest. The water wells they dig are used by other animals, they create habitat for grazing animals, and the roadways they make act as fire breaks and drainage conduits.

The pygmy elephant in Borneo is also endangered. Much smaller than African elephants, they don't get over 6.5 feet tall. The website www.worldwildlife.org/species/borneo-pygmy-elephant talks about tagging pygmy elephants in order to study their habits.

Elephants are not the only endangered species. Web page www.redlist.org/ says that 37,00 species are threatened with extinction, which is more that 1/4 of all assessed species. See www.worldwildlife.org/endangered for more info on endangered species.

Some animals are tagged to study their living habits. Some tags emit a signal, so that the animal can be tracked. Here in Ithaca, one can see deer with tags on their ears wandering in the fields. We see them around our houses in Cayuga Heights.

Your task in this assignment is to develop a Java class `Elephant` that will maintain information about elephants and a JUnit class `ElephantTest` to maintain a suite of test-cases for `Elephants`.

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 the code presentation conventions, which help make your programs readable and understandable.
- Learn and practice incremental coding, a sound programming methodology that interleaves coding and testing.
- Learn about and practice thorough testing of a program using JUnit testing.
- Learn to write *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 one *carefully*. A low grade is almost always due to lack of attention to detail 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 reading carefully as you do this assignment.

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 CMS for the course and do what is required to form a group. Both people must do something before the group is formed: one proposes, the other accepts. If you need help with the CMS, visit https://www.cs.cornell.edu/Projects/cms/userdoc/.

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.

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.

Managing your time

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

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 —a course instructor, a TA, a consultant. Do not wait. A little in-person help can do wonders.

Using the Java assert statement to test preconditions

As you know, 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, the method can do whatever it wants.

However, especially during testing and debugging, it is useful to use Java assert statements at the beginning of a method to check that the precondition is true. For example, for the precondition, "this elephant's nickname is at least one char long", one can use the following assert statement (using `name` for field). The additional test `n != null` is there to protect against a null-pointer exception, which will happen if the argument corresponding to `n` in the call is **`null`**. [This is important!]

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

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 `Elephant` and test it using class `ElephantTest` 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. (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, `Elephant`. It should be placed

in package `a1`. It does not need 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 tan Elephant. */
```

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 `Elephant`, declare the following fields, which will hold information describing an elephant. 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):

   - Nickname (a `String`). Name given to this `Elephant`, a String of length > 0.
   - Year of birth (an **`int`**). Must be >= 2000.
   - Month of birth (an **`int`**). In range 1..12 with 1 being January, etc.
   - Gender of this `Elephant` (a **`char`**). 'F' means female and 'M' means male.
   - Mother (an `Elephant`). Mother of this elephant—null if unknown.
   - Father (an `Elephant`). Father of this —null if unknown.
   - Number of known children of this `Elephant`.

   About the field that contains the number of children: The user *never* gives a value for this field; it is completely under control of the program. For example, whenever an elephant is given a mom `m`, `m`'s number of children 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.

   NOTE. When elephant `e` is given mom `m`, `m`'s number of children increases, not `e`'s number of children!

   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 year-of-birth field, state in a comment that the field contains the year of birth and that it must be >= 2000. The collection of comments on these fields is called the *class invariant*.

   Use Javadoc comments, placed *before* each field declaration. Below 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 this elephant was born. 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 unnecessary information in the 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 `ElephantTest`. You can do this using menu item **File —> New —> JUnit Test Case**. (If asked, add the "New Unit Jupiter test", also called *JUnit5*.)

4. Below, four *groups* A, B, C, and D of methods are described. 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 part. Make sure they are complete and correct — look at the specs we give you below. *Copy-and-paste from this handout makes this 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 class `ElephantTest` and add test cases to it for all the methods in the group. Note: Do NOT use fields in class `ElephantTest`; use only local variables

(4) Test the methods in the group thoroughly. Note: *Don't 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, or loops.

Group A: **The first constructor and the getter methods of class `Elephant`**.

Constructor	**Description (and suggested javadoc specification)**	
Elephant(String n, char g, int y, int m)	Constructor: an instance with nickname n, gender g, birth year y, and birth month m. Its parents are unknown, and it has no children. Precondition: n has at least 1 character, y >= 2000, m is in 1..12, and g is 'F' for female or 'M' for male.	
Observer Method	**Description (and suggested javadoc specification)**	**Return Type**
name()	= this elephant's nickname	String
isFemale()	= the value of "this elephant is a female"	**boolean**
date()	= the date of birth of this elephant. In the form "month/year", with no blanks, e.g. "6/2007"	String
mom()	= this elephant's mother (null if unknown)	Elephant (not String!)
dad()	= this elephant's father (null if unknown)	Elephant (not String!)
children()	= the number of children of this elephant	**int**

Consider testing the constructor. Based on its specification, figure out what value it should place in each of the 7 fields to make the class invariant true. Then, write a procedure named `testConstructor1` in `ElephantTest` to make sure that the constructor fills in ALL fields correctly. The procedure should: Create one `Elephant` 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.

We advise creating a second `Elephant` (in `testConstructor1`) of the other sex than the one first created and testing —using function `isFemale()` — that its sex was properly stored.

Group B: the setter/mutator methods. Note that methods `addMom` and `addDad` may have to change fields of both this `Elephant` and its parent in order to maintain the class invariant —the parent's number of children changes!

When testing the mutator methods, you will have to create one or more `Elephant` 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.

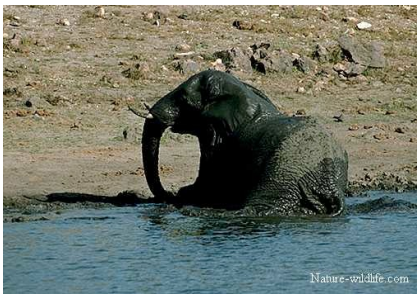
We are *not* asking you to write methods that change an existing mom or dad to a different `Elephant`. This would require if-statements, which are not allowed. Read preconditions of methods carefully.

Setter Method	Description (and suggested javadoc specification)
addMom(Elephant e)	Add e as this elephant's mother. Precondition: this elephant's mother is unknown and e is female.
addDad(Elephant e)	Add e as this elephant's father. Precondition: This elephant's father is unknown and e is male.

Group C: Another constructor. The test procedure for group C has to create an `Elephant` using the constructor given below. This will require first creating two `Elephants` using the first constructor and then checking that the new constructor set *all* 7 fields properly —and also the number of children of parameters `ma` and `pa`.

Constructors	Description (and suggested javadoc specification)
Elephant(String n, char g, int y, int m, Elephant ma, Elephant pa)	Constructor: an elephant with nickname n, gender g, birth year y, birth month m, mother ma, and father pa. Precondition: n has at least 1 character, y >= 2000, g is `'F'` for female or `'M'` for male, m is in 1..12, ma is a female, and pa is a male.

Group D: Write two comparison methods —to see which of two elephants is older and to see whether two elephant are siblings. Write these using only boolean expressions (with !, &&, and || and relations <, <=, ==, etc.). *Do not use if-statements, switches, addition, multiplication, etc*. Each is best written as a single return statement.



Note: two elephants are siblings if (1) they are not the same object and (2) they have a non-null mom or non-null dad in common.

Comparison Method	Description (and suggested javadoc specification)	Return type
isOlder(Elephant e)	Return value of `"this elephant is older than e."` Precondition: e is not null.	**boolean**
areSibs(Elephant e)	Return value of `"this elephant and e are siblings."` (note: if e is null they can't be siblings, so false is returned).	**boolean**

Writing isOlder. You have to check whether one elephant was born 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. Ideally, this requires at least 9 test cases.

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 `Elephant`s are the same object, use == or !=; do NOT use function `equals`.

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*.

   There are two places to put tests for assert statements in the JUnit testing class. Use either one; just make it clear what is being tested where. (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.

   You don't have to test all assert statements! It's a lot of work. Our testing procedure to test all assert statements is almost 80 lines long, consisting only of comments, blank lines, assignments, and assertThrows statements. The correctness of the assert statements is worth a total of 5 points, and there are about 15-20 individual tests one can make. If you make a mistake on 4-5 of them you lose only about 1 point.

   We *do* suggest that you test assert statements for these preconditions: (1) In the first constructor, the nickname of the element is at least one character long. For this, be sure to read carefully the section on "Using the Java assert statement" on page 2. (2) The precondition of procedure `addMom()`.

   Finally, suppose you use local variables `b` and `c` in testing that one assert statement is correct. Then don't use `b` and `c` in testing that another assert statement is correct. At this point we do not explain why.

6. Check that your method specs are appropriately written in Javadoc comments before each method header. To do this, in class `ElephantTest`, hover your mouse over calls on methods in class `Elephant`. Does the small window that pops up contain the method specification? If not, something is wrong.

7. Check carefully that a method that adds a mom or dad to an `Elephant` updates the mother's or father's number of children. Three methods do this —one is a constructor.

8. **Checklist:** 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*.

   a. 3 Points. Eclipse formatting preferences are not properly installed.

   b. ≥ 5 points: (1)The classes are not named `Elephant` and `ElephantTest`, (2) they do not have `package a1;` on the first line, or (3) the method names and signatures are not what they should be. (More points may be deducted if we have difficulty fixing your submission so that it compiles with our grading program.)

   c. ≤ 11 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?

   d. ≤ 11 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.

   e. ≤ 5 points. Does each method have the necessary assert statements for preconditions?

   f. ≤ 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 7 fields have the correct value? Do you have enough test cases?

g. 2 points. Did you put the time you spent on this project in the comment at the top of class Elephant, along with a comment saying what you thought about this assignment? (See step 9 below.)

9. Change the first line of file `Elephant.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 `Elephant`, 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 `Elephant.java` and `ElephantTest.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.