CS1110 Fall 2009 Assignment A1
Mercury in the food chain

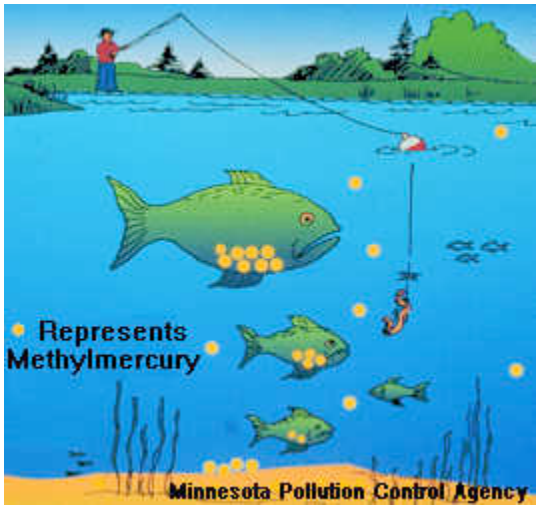
Submit on the CMS by midnight on Friday, 18 September

Introduction

According to the US Environmental Protection Agency, methylmercury poses a serious health risk:

> Outbreaks of methylmercury poisoning have made it clear that adults, children, and developing fetuses are at risk from dietary exposure to methylmercury. During these poisoning outbreaks some mothers with no symptoms of nervous system damage gave birth to infants with severe disabilities and it became clear that the developing nervous system of the fetus may be more vulnerable to methylmercury than is the adult nervous system. Mothers who are exposed to methylmercury and breast-feed their babies may also expose their infant children through their milk.

Where does methylmercury come from? According to "The Life Cycle of Methylmercury", posted on Fri Mar 21 2008 by Jessica Taylor-Cassan at www.aboutmyplanet.com/theories/methyl-mercury-draft,



> Methylmercury is an organic form of Mercury [sic] produced directly and indirectly from many industrial practices and the burning of fossil fuels, especially of coal. It is also formed from the massive amount of inorganic mercury that is released into the atmosphere each year ... by the burning of fossil fuels and on a much larger scale by natural sources such as forest fires and volcanoes. Methylmercury is formed from inorganic mercury by the absorption of mercury by anaerobic organisms that live in aquatic systems.

Human exposure to methylmercury, notes the EPA, happens through bioaccumulation:

> Nearly all methylmercury exposures in the U.S. occur through eating fish and shellfish. Microscopic organisms convert inorganic mercury into methylmercury, which accumulates up the food chain in fish, fish-eating animals, and people.
>
> In 2004 EPA and FDA issued the first-ever joint consumer advice about methylmercury in fish and shellfish. This advice was for women who might become pregnant; women who are pregnant; nursing mothers; and young children. The advisory provides three recommendations for selecting and eating fish or shellfish to ensure that women and young children will receive the benefits of eating fish and shellfish and be confident that they have reduced their exposure to the harmful effects of methylmercury.

Imagine being able to apply your programming and computer science skills to educating young students about the bioaccumulation of methylmercury, by developing some sort of game. You haven't worked out the details yet, but assume the game will involve various aquatic organisms eating other organisms.

Since we are talking about young children, you also envision that the game will incorporate a lot of simplifications. For instance, you will only deal with food chains instead of food webs; and in your simulated world, the way methylmercury levels change is simply: "if X eats Y, then X's new mercury level is its old level plus the level that Y had", so there is complete absorption not mediated by any other factors. Further, the only way something dies is by being eaten.

Finally, because you are just beginning to program, you will use a simple implementation in which organisms always exist, in a sense, but have a boolean field that indicates whether they are alive or not.

Your task in this assignment is to develop a Java class `Organism`, which will maintain information about an organism in this game, and a JUnit class `OrganismTester` to maintain a suite of testcases for `Organisms`. This assignment will help illustrate how Java classes and objects can be used to maintain data about a collection of things –like entities in a game.

Learning objectives

- Gain familiarity with DrJava and the structure of a basic class within a record-keeping scenario (a common type of application)
- Work with examples of good Javadoc specifications to serve as models for your later code
- Learn to write class invariants
- Learn the code format conventions for this course (use of "Constructor:" and "=" in specifications, indentation, short lines, etc.), which help make your programs readable and understandable and allow us to process your assignments and give you feedback more quickly
- Learn and practice incremental coding, a sound programming methodology that interleaves method writing and testing
- Learn about and practice thorough testing of a program
- Learn about *preconditions* of a method, which need not be tested

Iterative grading feedback (revise-and-resubmit cycle)

To help ensure that the learning objectives above are met, for this assignment, we will engage in the following iterative feedback process. If an objective has not been met, we will give you feedback so you can update your code and resubmit it. This process will ensure that every single student completely masters this material.

We will look at your code in several steps.

1. If the field specifications, javadoc specifications, or formatting are not appropriate, we will ask you to fix them and resubmit.

2. If the field and javadoc specs are ok, we will look at your test cases. If they are inadequate, we will ask you to fix them, test your program again yourself with the new cases, and resubmit.

3. If the test cases are adequate, we will test your program with our own testing program. If there are errors, we will ask you to correct them and resubmit.

Until mastery is achieved, your "grade" on the CMS will be the number of revisions so far, so that we can keep track of your progress. So don't be alarmed if you see a "1" for the assignment at first! The assignment will be considered completed when it passes all three steps outlined above.

Several rounds of resubmission may be necessary; we thus encourage you to submit early if you can. After you (re-)submit your assignment, please check the CMS often to see whether the grader has given you more feedback. This process should be finished within a week.

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. See the Staff page on the course homepage, www.cs.cornell.edu/courses/cs1110, for contact information.

How to do this assignment

Read the whole assignment before starting.

Develop and test class `Organism` and `OrganismTester` in the following incremental, methodologically sound way. This will help you complete this (and other) programming tasks quickly and efficiently. If we detect that you did not develop it this way, we may ask you to start from scratch and do a different assignment.

1. Make sure your preferences in DrJava are set to indent 4 spaces (if necessary, use menu item Preferences->Miscellaneous and change the first entry, "Indent Level", to 4). This makes understanding the structure of a program much easier. Also, as you work, please break any long lines in your code (including comments) so that we do not have to scroll right to read them; this makes your code much easier for us (and you) to read.
2. Start a new directory on your hard drive to contain the files for this project. (You should create a new directory for

every assignment in this course). In DrJava, write a skeleton class definition for class `Organism`, with no methods or fields in it, and save it in the new directory.

3. Next, in class `Organism`, declare the following fields (you can choose the names), which are meant to hold information describing a single organism. Make all these fields private and properly comment them (see also the "The class invariant" section below). Compile often as you proceed.
    - level (an int)
        - Position in foodchain (0 is lowest)
    - mercury (an int)
        - amount of methylmercury in this organism, in some predefined unit
    - alive (a boolean)
        - = "this Organism is alive" (has not been eaten)
    - eatenBy (an Organism)
        - the name of the object of class `Organism` that (directly) ate this `Organism`, if this `Organism` is no longer alive —null if this `Organism` *is* alive. For example, if a shark eats a sea bass that has eaten a shrimp, the eatenBy field for the shrimp should still be the sea bass, not the shark.
    - numVictims (an int)
        - number of Organisms this one has directly eaten —for example, if a shark eats a sea bass that has eaten a shrimp, the number of Organisms the shark has eaten should be 1, not 2
    - nickname (a String)
        - optional name (String of length >=1) for this `Organism` (e.g. smellyfish) —null if none

    Again, please break any long lines in your code (including comments) so that we can see everything without scrolling. This makes your code much easier for us (and you) to read. Also, indent comments properly, so that they follow the structure of the code.

    The class invariant. Recall that comments should accompany the declarations of the fields to describe what each field means, what constraints hold on the fields, and what the legal values are for the fields. For example, for the level field, state in a comment that the field represents the organism's position in the foodchain and that it must be non-negative. The collection of the comments on these fields is called the class invariant.

    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 catch or prevent bugs later on!

4. Start a new JUnit class, calling it `OrganismTester`.

5. For each of the following groups of methods, do the following. (1) Write the Javadoc specifications for each method in that part. (2) Write the methods. (3) Unless otherwise specified, write *one* test procedure in class `OrganismTester` and add test cases to it for all the methods in the group. (4) Test the methods in the group thoroughly. Do not go on to the next group of methods until the group of methods you are working on is thoroughly tested and correct.

    The descriptions below, while informal, represent the level of completeness and precision we are looking for in your Javadoc comments. In fact, you may copy and paste these descriptions to create the first draft of your Javadoc comments. If you do not cut and paste, please adhere to the conventions we use —such as using the prefixes "Constructor: ...", or "= ..." (the equals sign), or double-quotes to enclose an English boolean assertion—: we have a large number of assignments to grade, and your following these commenting conventions will make the grading process much smoother.

    The names of your methods must match those listed below exactly, including capitalization. The number of parameters and their order must also match. Our testing will expect those method names and parameter types, so any mismatch will cause our testing to fail, meaning that you will have to resubmit. Parameter names will not be tested —you can change the parameter names if you want.

    Thorough testing involves the following principles. The more interesting or complex a method is, the more test cases you should have for it. What makes a method 'interesting' or complex can be the number of interesting combinations of inputs that method can have, the number of different results that the method can have when run several times, the different results that can arise when other methods are called before and after this method, and so on.

    For the purposes of this assignment, you may *not* use if-statements in any of your methods; this constraint will help you see that there are often more elegant alternatives to if-statements and will give you some practice with boolean expressions. Submissions containing if-statements will be returned for you to revise.

    Also, for the purposes of this assignment, do *not* write code that checks whether preconditions hold, because it is the responsibility of the caller to ensure that the precondition is met. This means, for example, that you should *not* worry

about (or write code that checks for) invalid levels.

However, in general, you *do* need to check for "bad" conditions that aren't ruled out by preconditions. For example, if an argument may be null according to the specification, your code should check that the argument is not null before trying to perform an operation on it (otherwise, your code can crash due to a "null pointer exception"). There are no such examples in this assignment, but bear this point in mind for the future.

Again, remember to break long comments or code into several lines so that we do not need to scroll right to read.

Group A: The first constructor and all the getter methods of class Organism except for toString (not included in this assignment).

Constructor	Description (and suggested javadoc specification)
Organism(int lev)	Constructor: a new Organism at food-chain level lev. The Organism contains no methylmercury, is alive and thus has not been eaten by anything, has not eaten any Organism, and has no nickname. Precondition: lev is non-negative.

When you write a constructor body, be sure *all* the fields are set to appropriate values after execution, not necessarily just those corresponding to the constructor's parameters. (Example: what should the nickname be after the first constructor is executed?)

Getter Method	Description (and suggested javadoc specification)	Return Type
getLevel()	= the level of this organism in the food chain	int
getMercury()	= the amount of methylmercury in this Organism	int
isAlive()	= "this Organism is alive" (has not been eaten)	boolean
wasEatenBy()	= Organism that ate this Organism, or null if none	Organism —null if none.
getNumVictims()	= number of Organisms eaten by this Organism so far	int
getNickname()	= this Organism's nickname (null if none)	String —null if none.

Remember that every getter method in Organism needs at least one test case in OrganismTester. The test procedure for this group will create one Organism using the constructor and then check, using the getter methods, that all fields have the correct values.

Group B: the other constructor. The test procedure for group B should create one Organism using this constructor and then test, using the getter methods, that each field contains the correct value.

Constructor	Description (and suggested javadoc specification)
Organism(int lev, int m, String nn)	Constructor: a new Organism at food-chain level lev containing m units of methylmercury; its nickname is nn. It is alive and thus has not been eaten by anything, and it has not eaten any Organism. Precondition: lev is non-negative, m is non-negative, and nn has at least 1 character.

Group C: the setter methods. When testing the setter methods, you will have to use the getter methods. Good thing you already tested the getters thoroughly!

Setter Method	Description (and suggested javadoc specification)
setNickname(String nn)	Set this Organism's nickname to nn. Precondition: nn is a String of length at least 1.
eat(Organism victim)	Have this Organism eat victim. Precondition: victim is not null and is alive, and this Organism is alive and higher in the food chain than victim is.

Remember that your eat method may have to change several fields for both the eater and the victim. Also, recall that you do not need to write code to check whether preconditions hold; for example, method setNickname does not need to check nn's length.

6. Click the Javadoc button in DrJava and examine the output. You should see your method and class specifications. Read through them from the perspective of someone who has not read your code, and fix them 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 —without knowing anything about the private fields. Then and only then, add a comment to the top of your code saying that you checked the Javadoc output and it was OK.

7. We suggest you review the learning objectives and re-read this document to make sure your code conforms to our instructions; this may help reduce the number of rounds of revision that will be needed.

8. Upload files Organism.java and OrganismTester.java on the CMS. 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.

9. Check the CMS daily until you see that a grader has given you feedback. Within 24 hours, read the feedback, fix what it says to fix, resubmit, and **request a regrade** using the CMS.