CSE 114 Spring 2024: Assignment 5

Due: April 19, 2024 at 11:59 PM [KST]

Read This Right Away

For the due date of this assignment, don't go by the due date that you see on Brightspace because it is in EST. Go by the one given in this handout.

Directions

- At the top of every file you submit, include the following information in a comment
  - Your first and last name
  - Your Stony Brook email address
- Please read carefully and follow the directions exactly for each problem.
- Your files, Java classes, and Java methods must be named as stated in the directions (including capitalization). Work that does not meet the specifications may not be graded.
- Your source code is expected to compile and run without errors. Source code that does not compile will have a heavy deduction (i.e. at least 50% off).
- You should create your program using a text editor (e.g. emacs, vim, atom, notepad++, etc).
- You may use the command-line interface to compile and run your programs or you can now use IntelliJ IDEA.
- Your programs should be formatted in a way that is readable. In other words, indent appropriately, use informative names for variables, etc. If you are uncertain about what a readable style is, see the examples from class and textbook as a starting point for a reasonable coding style.

What Java Features to Use

For this assignment, you are *not* allowed to use more advanced features in Java than what we have studied ***at the time the assignment is posted***. If you have a question about features you may use, please ask!

What to Submit

Combine all your .java files from the problems below into a single zip file named as indicated in the **Submission Instructions** section at the end of this assignment. Upload this file to **Brightspace** as described in that section.

Multiple submissions are allowed before the due date. Only the last submission will be graded.

Please do **not** submit `.class` files or any that I did not ask for.

Naming Conventions In Java And Programming Style In General

Please use these conventions as you establish your *programming style.* Programming professionals use these, too.

- **Names:** Choose informative names,
  - e.g. `hourlyRate` rather than `hr` for a variable name.
  - `CheckingAccount` rather than `CA` for a Java class name.
- **Class names:** We start a class name with an uppercase letter as I have in my examples: Hello not hello or `HELLO`. For multi-word names you should capitalize the first letter of each word,
  - e.g. `UndergraduateStudent, GraduateStudent, CheckingAccount`
- **Variable names:** We start a variable name with a lowercase letter and capitalize each 'word' after that. This is called 'Camel case':
  - e.g. `count, hourlyRate, averageMonthlyCost`
- **Method names:** Naming convention for method names is the same as that for variable names.
- **Use of white space:** Adopt a good indentation style using white spaces and be consistent throughout to make your program more readable. Most text editors (like emacs and vim) should do the indentation automatically for you. If this is not the case, see me and I can help you configure your setup.

I will not repeat these directions in each assignment, but you should follow this style throughout the semester.

Problem 1 (25 Points)

The goal of this problem is to have you familiarize yourself with the mechanics of creating a class and some objects using the class. Then, do some more interesting things with the objects that you create. This is more of a tutorial than a problem. Follow along as I guide you through the process.

Let us design a few classes. First, design a class named Message to model email messages in a file named `Message.java`. This class should satisfy the following:

- It should have the following attributes:
  - A *from* field represented as a string (e.g., "alee@sunykorea.ac.kr")
  - A *to* field represented as a string (e.g., "jdoe@gmail.com")
  - A *date* which is represented as a string (e.g., "Fri, Apr 19, 2019 at 11:28 PM")
  - A *returnReceiptRequested* field which is a boolean and is true if the sender wants to be notified when the recipient reads the message. [Note that this field will not be useful unless we were to build a whole message delivery system which we are not for this assignment (but we may in the near future!)]
  - A *subject* field represented as a string (e.g., "Meeting time")
  - A *body* field represented as a string (e.g., "Hey, John, The meeting was moved up to this coming Tuesday. We will be discussing the new proposed feature.").

    I can imagine adding a few more attributes, but for the purpose of this exercise, these will do. Note that each attribute will be represented as a *private* field in your class. Do not use public fields in the classes that you design. Use private fields and provide getters and setters so that accesses to the fields from outside the class can only be made through getters and setters. More on this next.

- For each field in your class, add a method that can be used to read the value of the field. That is, add a getter method for each field if it makes sense to have a getter. In addition, add another getter named *getLength* which returns the length of the message body in number of characters. Note that there is no field name length, but we can still add a getter *getLength* since that would be a useful method. It is also called a reader.

- For fields in your class where it makes sense to, add a method that can be used to change the current value of the field. That is, add a setter method if it makes sense to include one. Given a field of a class, a value can be assigned through one of several different ways: (i) It may be assigned by passing a value to a constructor as an argument at the time an object is created. This is how you initialize a field. (ii) It may be assigned by calling a setter after an object has already been created. Or, (iii) it can be assigned with a value generated inside a constructor at the time the object is being created rather than receiving one through a parameter. It all depends on what sort of field we are dealing with. So, think about each field and decide which way would make sense for each. For now you will want to use one of the first two ways I described above. We will learn the third way a little later in the semester. It is also called a writer. For some fields it may not make sense to add a getter or setter. For example, a PIN number in a bank account should be hidden from the outside world, so it would not make sense to add a getter. The methods inside the class can still access the field even if it is private, right? Even for a PIN number, you may allow a setter method though because sometimes you may want to allow the PIN to be changed. For a field representing a social security number, you would not allow a setter although you might allow a getter. Although we added a getter named *getLength*, we would not want to add a setter for it, right?

- Since your getters and setters are added to be used by other classes outside the Message class, they should be declared as public. Since the fields will be accessed (for read and write accesses) via getters and setters respectively, there is no reason to keep the fields themselves to be public. In fact, the fields must be declared to be private so that the methods within Message can access them directly, but no other methods in any other class in your system can access them directly without going through the getters and setters. This is how you control the visibility of the state information in an object. The values of all the fields collectively represent the state of the object at any point in time. Since the values can be changed using the setters, the state information can change. So, the state information in an object is time-dependent. The fields, getters, and setters must be declared to be non-static, i.e., do not add the static keyword in their declarations. Since we are planning on creating many message objects based on the definition of this Message class (the blueprint), they should be declared to be non-static. In fact, so far in the class Message, nothing should be declared as static.

- Introduce a constructor that does not take any argument. In this case, the fields should be initialized with some reasonable default values within the constructor: for the from, to, date, subject, and body fields use an empty string (“”) as the default value.

- Let us also add another constructor that takes five arguments: one for each of the five fields. You know what to do in the body of this constructor, right?

Now that we have a class representing messages, let us build a class that we can use to test the implementation of the Message class. We will call the new class `UseMessage` in a file named UseMessage.java. (We could have chosen the name of this class to be anything we wanted, but `UseMessage` would be a logical choice since we will be using that class to test the implementation of the Message class.) `UseMessage` class includes only one method (a static) method in it named main that does the following in the given order:

1. Create one object (instance) of the Message class and name it *msg1*. Be sure to use the names exactly as I specify them. Think about what the type of this variable msg1 should be. This message object should be created with the constructor that does not take any argument.
2. Now print the value of each field in *msg1* to the standard output device (i.e., screen) using the values returned by the getters. Include *getLength* as well. As you print out the values for the fields, properly annotate the values being printed out so they will be meaningful (in other words, print the object name, field name, and then the value).
3. This time change the value of each field in *msg1* using the setters if available. You may use any reasonable values for the fields as you modify them.
4. Now print the value of each field in *msg1* again using the getters including *getLength*. Be sure that your getters are now seeing the new values now that we have changed them using the setters. *getLength* would most likely return a different value than the previous call since the body of the message would most likely have a different length after it was modified. Again as you print out the values for the fields, properly annotate the values being printed out.
5. Create another object of type Message and name it *msg2* this time. This message object should be created with the constructor that takes five arguments this time. You can use any reasonable values as arguments in the call to the constructor.
6. Now, print the value of each field in msg2 using the getters including *getLength*. Be sure that your getters are seeing the correct values. Again as you print out the values for the fields, properly annotate the values being printed out.

Now, run the main in `UseMessage`.

Assuming that everything works as expected so far, let us add some more as follows in the given order:

1. Add a piece of code in the main of `UseMessage` to change the message body of msg2 to be ”It is too hot to hike and too cold to plunge.”.
2. Add to the class Message a non-static method named *isImportant* that returns true if the message is important, false otherwise. A message is considered important if the subject contains the words ”notice”, “meeting”, ”deadline”, or “agent” in it (which should be done as a ***case independent*** comparison!) AND the message was communicated in the current year, where the current year is 2024. Note that the return type of *isImportant* must be boolean. Should *isImportant* take any parameter? If you answered ”Yes” to this question, you would be wrong! Note that the body is stored in a field and any method in a class has direct access to the field in the object.
3. Add a piece of code in the main of `UseMessage` to call the *isImportant* method that you just added with the object *msg2*. Print the returned value of *isImportant* to the standard output device. Do it once more with msg1. Be sure to select the subjects in both message objects carefully so that the returned values when you call *isImportant* are different, e.g., one true and the other false.
4. Add to the class Message a non-static method named *print* that prints the state information (field values) of the current message to the standard output device. This method would not take any parameter, right? Let us have it print each field value on a separate line.
5. Add a piece of code in the main of `UseMessage` that prints the state information of msg1 using the *print* method that you just added to Message.
6. Add a piece of code in the main of `UseMessage` that creates an array of messages of length 5. Name that array *messages*. Add *msg1* as the first element of the array and *msg2* as the second element of the array. Create three more Message objects with contents (subject, to, from, date, and body) of your choice and add them to the array also.
7. In the main, add a for loop that loops through the message objects in the messages array and prints each message in the array using the *print* method that you added to Message.
8. Add another for loop that loops through the message objects in the *messages* array once more and prints only important messages in the array using *print*.

Hand in both `Message.java` and `UseMessage.java`.

Submission Instructions

Please follow this procedure for submission:

1. Place the deliverable source files into a folder by themselves. The folder's name should be CSE114_HW5_<yourname>_<yourid>. So if your name is Alice Kim and your id is 12345678, the folder should be named 'CSE114_HW5_AliceKim_12345678'

2. Compress the folder and submit the zip file. ````
    a. On Windows, do this by clicking the right-mouse button while hovering over the folder. Select 'Send to -> Compressed (zipped) folder'. The compressed folder will have the same name with a .zip extension. You will upload that file to the Brightspace.
    b. On Mac, move the mouse over the folder then right-click (or for single button mouse, use Control-click) and select **Compress**. There should now be a file with the same name and a .zip extension. You will upload that file to the Brightspace.
3. Navigate to the course Brightspace site. Click **Assignments** in the top navbar menu. Look under the category 'Assignments'. Click **Assignment5**.
    a. Scroll down and under **Submit Assignment**, click the **Add a File** button**.**
    b. Click **My Computer** (first item in list).
    c. Find the zip file and drag it to the 'Upload' area of the presented dialog box.
    d. Click the **Add** button in the dialog.
    e. You may write comments in the comment box at the bottom.
    f. Click **Submit**. ← Be sure to do this so I or the grading TA can retrieve the submission!