The State Universtiy of New York, Korea
Computer Science

CSE 114 Handout 9: PS 6 April 19, 2019

This problem set is due at **11:55pm on Thursday, April 25, 2019, KST**. Don't go by the due date that you see on Blackboard because it is in EST. Go by the one given in this handout.

- Please read carefully and follow the directions exactly for each problem. Files and classes should be named exactly as directed in the problem (including capitalization!) as this will help with grading.
- You should create your programs using emacs.
- Your programs should be formatted in a way thats 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.
- This problem set assumes you have installed Java and emacs on your computer. Please see the course web for installation instructions.
- You must use the command-line interface to run your programs. That is, you must use the `javac` and `java` commands to do this problem set. Do not use Eclipse yet.
- **Be sure to include** a comment at the top of each file submitted that gives **your name** and **email address**.

Your submission should include the following file. Do not hand in `.class` files or any other that I did not ask for:

```
Ttt.java
Ttt2.java
Ttt3.java (optional)
Message.java
UseMessage.java
```

What Java Features to Use

For this assignment, you are not allowed to use more advanced features than what we have covered in Lectures 1 through 12 (April 15 lecture).

Partial vs. Complete Solutions

Please read what I said in PS 1 on this issue.

Naming Conventions In Java And Programming Style In General

Refer to the ones given in PS 1.

Problem 1 (30 points)

In this problem, you will implement a Tic Tac Toe game. You are given an implementation of Tic Tac Toe (see the `[given]` link next to this handout), and will modify it to meet the requirements given below.

Inside the class `Ttt`, you will see nine static variables declared as follows:

```
private static char s1;
private static char s2;
private static char s3;
private static char s4;
private static char s5;
private static char s6;
private static char s7;
private static char s8;
private static char s9;
```

You are to replace these nine lines with the following single line:

```
private static char[][] board;
```

In other words, we will be using a two dimensional array of an appropriate size to represent the board. (Hint: This one line declares a 2D array, but it does not actually create a 2D array object, right? If so, where in your program would you create it?)

Obviously, the rest of the program will have to be updated in such a way that the program now works with this new data structure, namely the two dimensional array.

Your task is to update the remainder of `Ttt.java` and make it work without any error. Do not change the signature of the methods given in `Ttt.java` unless you absolutely have to, in which case you must justify the change(s).

Hand in the updated version of `Ttt.java`.

Before you try to solve the problem, play with it (the one I gave you) and try to understand the given program.

Problem 2 (30 points)

The given version of Tic Tac Toe continues until the entire board is filled up even when it is already a tie game. Improve your game (the one you finished above with a 2D array) so that it can detect a tie game if it is a tie game even if the board has not been completely filled up yet. Call the file `Ttt2.java` for this version, and hand it in.

Problem 3 (Extra credit problem worth 30 points)

The versions in Problems 1 and 2 above are played by two humans. Revise your result from Problem 2 so that it is played by a human and a computer. Call this version `Ttt3.java`. This one is optional. You should toss a coin to decide who starts first in this version. Think about how to simulate ”a coin toss”. If you do this problem, hand in `Ttt3.java`.

Problem 4 (40 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: *from* represented as a string (e.g., ”alee@sunykorea.ac.kr”), *to*

represented as a string (e.g., ”jdoe@gmail.com”), *date* represented as a string (e.g., ”Fri, Apr 19, 2019 at 11:28 PM”), *subject* represented as a string (e.g., ”Kite flying”), and *body* represented as a string (e.g., ”Hey, John, This Sunday looks like a good day to fly a kite or jump in a lake. Do you want to hike up to Mt. Sulak and fly a kite or go to TaeAn Beach and plunge?”). 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 named `length`, but we can still add a getter `getLength` since that would be a useful method. It is also called a *reader*.

- For each field in your class, 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.

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 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.”. With a message, this sort of operation may not be useful, but doing so will help you understand better how to use an object. So, bear with me.

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 *body* contains the word ”kite” or ”plunge” in it and the message was communicated in the current year, where the current year is 2019. 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 bodies 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 to the class `Message` a non-static method named `toString` that takes no parameter and returns a string representation of the object. For the exact signature of the method, refer to the documentation for the `toString` method in `java.lang.Object` class. The `toString` that you added should include only the from, to, and subject fields. That will make the output a little easier to read.

7. Add a piece of code in the `main` of `UseMessage` that prints the state information of `msg1` using the `toString` method that you just added to `Message`.

8. 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 of your choice and add them to the array too.

9. 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`.

10. Add another `for` loop that loops through the message objects in the `messages` array again and prints each message in the array using the `toString` method that you added to `Message` this time.

11. 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 `toString` this time.

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