This problem set is due **Friday, November 11 at 11:59pm, KST**. Note that the due date that you see on Blackboard is not accurate since it shows the time in EST. You should go by the due date in this handout.

- To solve each problem below, you will be implementing classes.
- Please carefully read 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 your preferred text-editor or the Eclipse text editor.
- Your programs should be formatted in a way that's readable. In other words, indent appropriately, use informative names for variables, etc. If you are uncertain about what is a readable style, see the examples from lectures and textbook as a starting point for a reasonable coding style.
- Your programs should compile and run without errors. Please do not submit programs that do not compile! Its better to submit partial implementation that compiles as opposed to complete implementation that does not compile. You may use either Eclipse or the command-line to compile and run your programs.
- Be sure to include your name and your email address as comments at the top of each file that you submit.

### What Java Features to Use

For this assignment, you are not allowed to use more advanced features than what we have covered in Lecture 1 through Lecture 13 (Chapters 1 through 9 of our textbook plus my notes).

### What to Submit

Submit the following files on **Blackboard**. Please do **not** submit `.class` files or Eclipse-specific project files or any I did not ask for.

```
Message.java
UseMessage.java
Player.java
Team.java
UseTeam.java
(And others if you added more)
```
Problem 1 (75 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. You may want to create a project in Eclipse to include these classes that you will define if you are using Eclipse for your programming exercises. 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., "alae@sunykorea.ac.kr"), to represented as a string (e.g., "jdoe@gmail.com"), date represented as a string (e.g., "Fri, Nov 4, 2016 at 10:28 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 the 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. getBalance that we added to Account is an example of a getter. 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 for each field 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. Depending on the situation the third way may be beyond what we can do at the moment. setBalance that we added to Account is an example of a setter. 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 would 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 2016. 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 of your choice and add them to the array too.

9. 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.

Problem 2 (75 points)

In this problem we will model a couple of baseball teams. As you will see below, the entire program will consist of at least three classes: Player, Team, and UseTeam in Player.java, Team.java, and UseTeam.java respectively.

I am providing an almost complete implementation of UseTeam.java and your job is to complete the remainder of UseTeam.java and create Player.java and Team.java in such a way that UseTeam.java will work without modifying UseTeam.java. By that I mean that you may add more to UseTeam.java but you are NOT allowed to modify the methods that I have already included. In other words, you are not allowed to change the signatures of the methods that I have already used.

In the [given] link next to this handout, you will see two files:

- UseTeam.java: the use code of the classes that you will provide. Read my comments in this file carefully. Additional problem requirements are specified in this file.
- output.txt: a sample output generated by running the main of UseTeam in my model solution. Your completed program will have to generate the output that is identical to my sample output, except possibly the last several digits of a floating point number output. If you want to add other features in addition to my sample output, you are welcome to do so, but try to be brief for readability.

Restrictions: Do not use public fields in the classes that you design. Use private fields and provide getters and setters where appropriate so that accesses can be made to the fields through the getters and setters.

Hand in your Player.java, Team.java, and UseTeam.java. If your solution uses additional files, hand in the source files for them as well.