Saving abUTA

Due Tuesday, March 4 at 8 a.m.

CSE 1325 - Sprint #4 of 4 - Homework #6 - Rev 0

Assignment Overview

Everyone seems to be unhappy with their social media apps these days - Facebook, YouTube, WhatsApp, Instagram, the almost-late (in the US) TikTok, and X (née Twitter). What should any self-respecting geek do? Write our own, of course!

This is sprint 4 of a 4-sprint, 4-week project that you can add to your growing resume. We'll have a final, stand-alone Java assignment to practice threading before Exam #2. Implementation guidance will be provided each week to help you with your implementation, and you have one last checkpoint where you may switch to the professor's suggested solution if you have lost your footing.

For the fourth and final sprint (P06), we'll add the ability to save our tree of messages *without* breaking encapsulation. We'll take the object-oriented approach as discussed in Lecture 11.

If your first three sprints left something to be desired, contact the professor and ask about baselining the suggested solution. While we *prefer* that you make this project entirely your own, we want to ensure that you're able to complete it. Let's talk!

Sprint #4

Sprint Overview

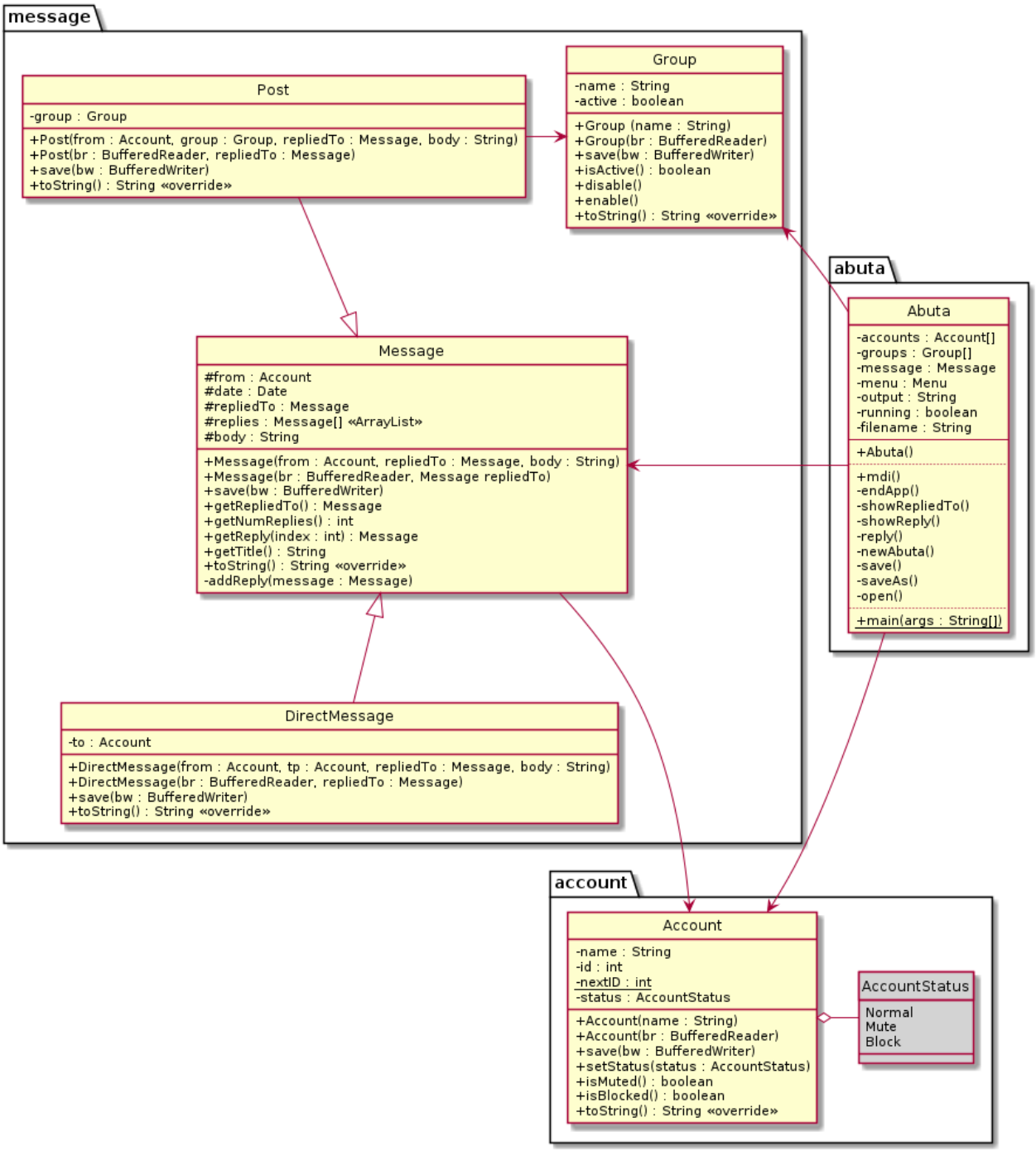
For this sprint we will add two new members to each of our data-containing classes - a `save(BufferedWriter bw)` method that streams out the fields, and a `<classname>(BufferedReader br)` constructor that streams them back in to construct a replacement.

The menu should offer 4 *additional* behaviors: New abUTA (method `newAbuta`), Save (`save`), Save As (`saveAs`), and Open (`open`). Each method is responsible for updating the fields to implement those behaviors for our social media app.

Overview of the Data Class Changes

Consider this class diagram on the next page. All classes except for `AccountStatus` (the enum) get a new constructor and a save method.

You may ONLY add the public members shown on the diagram to your classes. No additional public members are permitted. Don't duplicate any class responsibilities in your other methods, including main, and do NOT change the visibility of your data fields!


message
Post
-group : Group
+Post(from : Account, group : Group, repliedTo : Message, body : String)
+Post(br : BufferedReader, repliedTo : Message)
+save(bw : BufferedWriter)
+toString() : String «override»
Group
-name : String
-active : boolean
+Group (name : String)
+Group(br : BufferedReader)
+save(bw : BufferedWriter)
+isActive() : boolean
+disable()
+enable()
+toString() : String «override»
Message
#from : Account
#date : Date
#repliedTo : Message
#replies : Message[] «ArrayList»
#body : String
+Message(from : Account, repliedTo : Message, body : String)
+Message(br : BufferedReader, Message repliedTo)
+save(bw : BufferedWriter)
+getRepliedTo() : Message
+getNumReplies() : int
+getReply(index : int) : Message
+getTitle() : String
+toString() : String «override»
-addReply(message : Message)
DirectMessage
-to : Account
+DirectMessage(from : Account, tp : Account, repliedTo : Message, body : String)
+DirectMessage(br : BufferedReader, repliedTo : Message)
+save(bw : BufferedWriter)
+toString() : String «override»
abuta
Abuta
-accounts : Account[]
-groups : Group[]
-message : Message
-menu : Menu
-output : String
-running : boolean
-filename : String
+Abuta()
+mdi()
-endApp()
-showRepliedTo()
-showReply()
-reply()
-newAbuta()
-save()
-saveAs()
-open()
+main(args : String[])
account
Account
-name : String
-id : int
-nextID : int
-status : AccountStatus
+Account(name : String)
+Account(br : BufferedReader)
+save(bw : BufferedWriter)
+setStatus(status : AccountStatus)
+isMuted() : boolean
+isBlocked() : boolean
+toString() : String «override»
AccountStatus
Normal
Mute
Block

Group

I recommend you start with the easy class: `Group`. The `save` method should write the two fields to the `BufferedWriter` stream, and the constructor should read them back in.

Account

Next up would be `Account`. I recommend saving all 4 fields to the `BufferedWriter` stream, including the static field. While this does duplicate the static field many times in the save file, ints are small, and the effort to only save the static field once is not worth the effort. The constructor should read them all back in.

Message (superclass)

Once these work, move along to class `Message`. Don't panic! It's a bit bigger, but just take it one step at a time. It will all come together if you take it in baby steps, I promise! Refer back to Lecture 11 as needed for each step of the way.

First you'll need to decide how to save and restore the `Date` field. Check the documentation - what `Date` method returns a value that is compatible with a non-deprecated constructor? It's a `long` shot - look carefully.

With that settled, in the `save` method, save the `from` field, write out the `date` field, and then the `body` field. Finally, write out the `replies` ArrayList - first the `size()` (number of Messages), and then iterate over the messages. For each message, first write the classname of the object (using the `getClass().getName()` methods per Lecture 11), then tell the message to `save(bw)` itself.

Do NOT write out the `repliedTo` field. Remember, this is a tree data structure. We'll be starting with the root of that tree, so we only want to save "downward", that is, toward new *replies*. The nodes that are "upward", that is, the `inRelyTo` fields, have already been saved.

Finally, write the `Message` constructor. Restore the `from`, `date`, and `body` fields exactly as written out in the `save(bw)` method. Then construct the ArrayList and read the size of the ArrayList of messages you need to restore. Loop that many times: First, read the *type* of the object, which is the constructor to be called - either "message.Post" or "message.DirectMessage" - then call the respective constructor and add the resulting object to the `messages` ArrayList.

One more *very* important step in the `Message` constructor: Assign the `repliedTo` parameter to the field of the same name, and if it is not null, call the `addReply(this)` method on it. This reconstructs the tree from the `BufferedReader` stream as each `Message` subclass is reconstructed, exactly as the original constructor does.

Post and DirectMessage (subclasses)

These are simple compared to the superclass. First, call the `super` save method or constructor with the correct parameters. Then, also save or restore the field that is unique to that subclass.

Abuta

Now that the classes in the account and message packages can save and reconstruct themselves, we just need to add menu items to command it.

Method `newAbuta()` should simply overwrite the `message` field with the original root message. In fact, it would be DRY-compliant to move this code from the `Abuta` constructor to here, and then call `newAbuta()` from the constructor.

Method `save()` should use a try-with-resources to construct a `BufferedWriter` using the new `filename` field. Then find the root message (either add a field that stores it permanently, or use the `getRepliedTo()`

method until you reach the root, that is, when `getRepliedTo()` returns null). Call the `save(bw)` method on the root message.

Be sure to catch `IOException` and set `output` to an appropriate error message. I suggest you also use the exception's `printStackTrace()` method - this won't show up on the menu screen, but you'll be able to scroll upward and see why you got an exception, which is *very* helpful while debugging.

Method `saveAs()` should obtain a new filename (think `Menu.getString`) and if not null, assign it to the new `filename` field. Then call `save()`.

Method `open()` should obtain a filename to open (think `Menu.getString`) and if not null, set the new `filename` field to the entered filename. Then use a try-with-resources to construct a `BufferedReader` using `filename`, and construct a new `Post` object to be referenced by the `message` field (the second parameter is null). As above, catch `IOException` and set `output` to an appropriate error message while also calling the exception's `printStackTrace()` method.

NOTE: You do NOT need to save any of the other `Abuta` fields. The `accounts` and `groups` ArrayLists are essentially `final` - we didn't provide a way (yet) to add or remove accounts or groups, so we don't need to save them. And the rest of the fields just manage the main method's execution and shouldn't persist from one program execution to the next.

Bonus

TBD