Refactoring for the Long Term - ELSA (Sprint 5)

Due Tuesday, April 25 at 8 a.m.

CSE 1325 - Spring 2023 - Homework #11 / Sprint 5 / Rev 0

Assignment Background

The Exceptional Laptops and Supercomputers Always (ELSA) store offers the coolest (ahem) deals in computing technology for the savvy computer geek and their lucky friends. Each computer can be hand-crafted to match the technologist's exact needs, with a growing selection of convenient predefined configurations already purchased by your discerning peers (and competitors). They now belatedly seek to automate their physical and online storefronts, replacing paper forms and ink pens with the miracle of modern computing technology. Your goal is to prove that you can implement their store management system and thus win the contract to build it, with all of the associated fame and cash.

This is Sprint 5 of 6.

IMPORTANT: Do NOT use full_credit, bonus, or extreme_bonus subdirectories for this project. Instead, organize your code into two packages, `store` and `gui`. ALWAYS build and run from the P11 directory.

The Scrum Spreadsheet

The Feature backlog HAS changed slightly for this sprint. Please incorporate the change into your own planning.

The intent of using a *very* simplified version of Scrum on this project is to introduce you to the concept of planning your work rather than just hacking code at the last minute and hoping for the best. **Professionals make a plan and then execute it.** Amateurs sling code and suffer the consequences.

Submitting an *accurate* spreadsheet is part of your grade for these assignments. This is not what you *wish* had happened - it's what actually happened. **Be certain you update the spreadsheet and add, commit, and push it at cse1325/P11/Scrum.xlsx before the end of the sprint!** If you prefer to use an online spreadsheet, ensure that it is publicly visible and include a cse1325/P11/Scrum.url text file instead containing its *public* link.

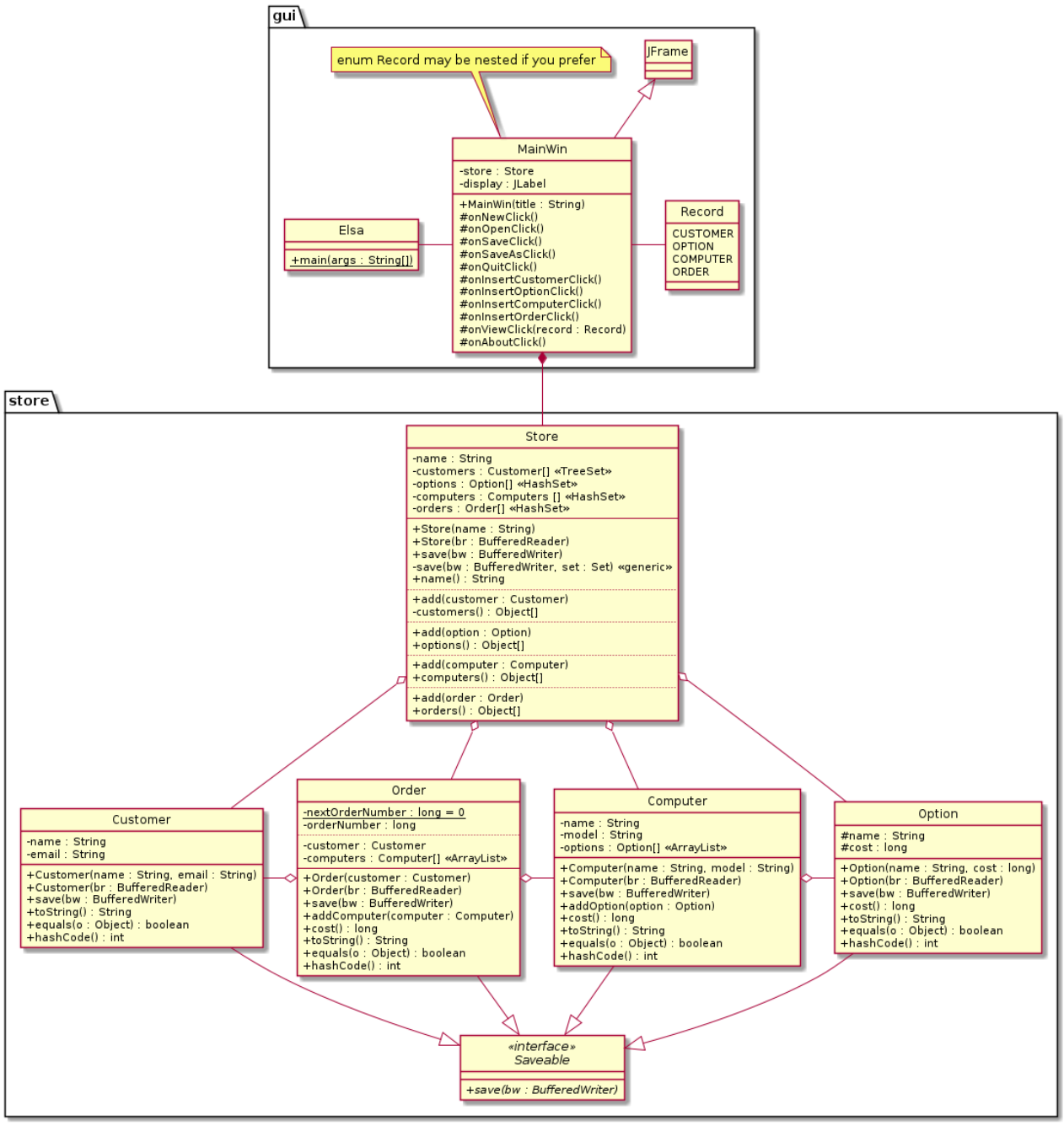
For this sprint, at a minimum, you will update the Product Backlog AND Sprint 04 Backlog tabs. Then add, commit, and push your Scrum spreadsheet.

Now that you *have a plan* - on to coding!

Class Diagram

The diagram on the next page covers through Sprint 5. Dotted lines are purely visual and have no semantic meaning.

For this sprint, we will refactor (improve) parts of our our Store-related code into package `store` using some of the intermediate Java features we've been studying - generic methods, the hashCode() method, sorted TreeSet, and the de-duplicating HashSet!



Overview of Changes

To avoid storing duplicate data in Store, we *could* search each of the 4 ArrayLists to detect and avoid the duplicates (relying on the `equals` method, of course). But better than writing our own algorithm would be to use code already in the Java Class Library - the `Set` interface implementations! Attempting to add a duplicate value to a `Set` object will simply discard the duplicate. BUT this will require that the `hashCode` method be properly overridden for each of the 4 data classes - Customer, Option, Computer, and Order. Then, a `HashSet` for Option, Computer, and Order objects in Store will ensure we have no duplicates.

In addition, we want to see the Customer list alphabetized, first by name and then (if equal) by email address. A `TreeSet` is just the ticket - it will not only avoid duplicates but also automatically return our Customer objects in sort order! But TreeSet requires that our Customer class implement the Comparable generic interface to provide the compareTo method for another Customer object on which it relies for sorting.

Finally, our `Store.save` method from P10 has 4 blocks of very similar code, one each for writing the objects in the customers, options, computers, and orders Sets (formerly ArrayLists) to a BufferedWriter stream. We'll practice the DRY (Don't Repeat Yourself) principle by writing a single, generic Store.save method that accepts both a BufferedWriter *and* a Set (implemented by both HashSet and TreeSet) and does the writing for us. But how to tell Java that our 4 independent Set fields each store objects that have the required save method? We will need to write a Saveable interface and implement it for Customer, Option, Computer, and Order! Then, we can tell our generic save method to accept a Set of any type that implements Saveable.

If that's not clear, don't worry. Step by step guidance is provided below. But you might want to review Lectures 22 and 23 after working through that code to ensure you understand generics, collections, and algorithms for the exam.

Write the Code

Avoiding Duplicate Records

(UNIQ) The Set collection is much like an ArrayList, except that it ignores attempts to add a duplicate member (that is, a *different* object with the *same* values as an object already in the Set). Sets also don't have integer indices (and thus no `get` method), although they work just fine with for-each loops and iterators.

Sets require their generic type to override the `hashCode()` method, so **override hashCode() for Customer, Option, Computer, and Order.**

Then, **change Store's four ArrayList fields customers, options, computers, and orders to HashSet instead.**

At this point, you should be able to **rebuild and test** your code with little or no further modification. Verify that it compiles and runs, and that any records with duplicate field data in each of the four types are ignored. Also verify that you can save and open .elsa files (P10 files are likely still compatible, too). If anything is broken, fix it before continuing.

Sorting the Customer List

(SORT) Your display code in MainWin for the 4 data types most likely originally displayed them in the order entered, because of the ArrayLists. To force customers to be listed is sort order, we must first **update Customer to implement the generic Comparable interface.**

When writing the required compareTo method for Customer, use the `name` field's compareTo first to determine a possible result - negative if less than, 0 if equal, and positive if greater than. Then, if 0, use the `email` fields' compareTo instead. This will result in customers being sorted by `name` first, and if two names are identical they will be further sorted by `email`.

Finally, **for Store.customers only, change the type from HashSet to TreeSet.** (A HashSet is unsorted, but a TreeSet relies on the Comparable interface to ensure its data is always sorted.)

Now **rebuild and test**, ensuring that your customer list not only rejects any duplicate name / email combinations, but also lists the customers in sort order - by name first, then by email address. If anything is broken, fix it before continuing.

Using Generic Save Code

(GSAVE) Finally, in Store, we will want to specify a second, private, and generic save method that can be called 4 times by the existing save method to save the customers, options, computers, and orders fields, respectively. To do this, we first need a way to show that those 4 fields each have a generic type that includes a save method.

Do this by writing a Saveable interface with an abstract method, `save(BufferedWriter)`.

Once that compiles, **change Customer, Option, Computer, and Order to implement Saveable**. They already have the correct save method, so just add the `@Override` and declare victory! (Note that Customer *also* implements Comparable from "Sorting the Customer List" above - the syntax is simply `class C implements I1, I2 {` for C implementing the two interfaces I1 and I2.

Then write the second, private, and generic save method for Store that accepts not only a BufferedWriter (like the existing save method) but also the Set to be saved. Use a constraint that ensures the generic type for the second parameter implements Saveable. The body for this new generic set method is the same as for saving the four collections used by Store - write the number of elements to the stream first, then iterate with a for-each over each collection to call its save method. You'll need to use the `var` keyword in your for-next loop.

Finally, **replace the repetitive code in the legacy Store.save method with 4 simple calls to the new, private, generic save method.** Your (maybe) 13 lines of code will become (maybe) 8 lines of code counting the code in the generic method - about a 40% reduction. Nice!

What Does and Doesn't Change

Your user interface - menu, toolbar, and dialogs - are unlikely to change at all, which is why no screenshots for this sprint. The core use cases are the same.

Your data files should also be backward compatible with P10 data files, since we're saving the same data. Future saves will write the data in a different order, but we shouldn't care.

Your list of customers in the main window *should* change - it should now be sorted by name first, then email address.

Your add commands *should* also change their behavior - duplicate records should now be ignored rather than being added to the Store.