Placing the Order - ELSA (Sprint 1)

Due Tuesday, February 28 at 8 a.m.

CSE 1325 - Spring 2023 - Homework #5 / Sprint 1

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 1 of 6.

IMPORTANT: Do NOT use full_credit, bonus, or extreme_bonus subdirectories for this project. Keep all files to be graded in the cse1325/P05 directory on GitHub. We will add packages next sprint.

The Scrum Spreadsheet

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/P05/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/P05/Scrum.url text file instead containing its *public* link.

How to Read the Product Backlog

For the Product Backlog tab's list of features, read the header for columns H, I, and J right before the feature's cell. So for example, read

As a...	I want to...	So that I can...	Notes
Sales Staff	Remember each customer's name and contact info	Ship them products repeatedly	Validate the email address (lightly)

like this: "As a Sales Staff, I want to Remember each customer's name and contact info So that I can Ship them products repeatedly." That's a concise description of who wants the feature, what the feature is, and why they want it.

Also read the Notes, in this case, "Validate the email address (lightly)", which may give you some gentle hints as to what is expected for that feature.

How to Update the Spreadsheet

You need to add, commit, and push the spreadsheet as cse1325/P05/Scrum.xlsx (or Scrum.url) in addition to your usual code and build.xml *every sprint*.

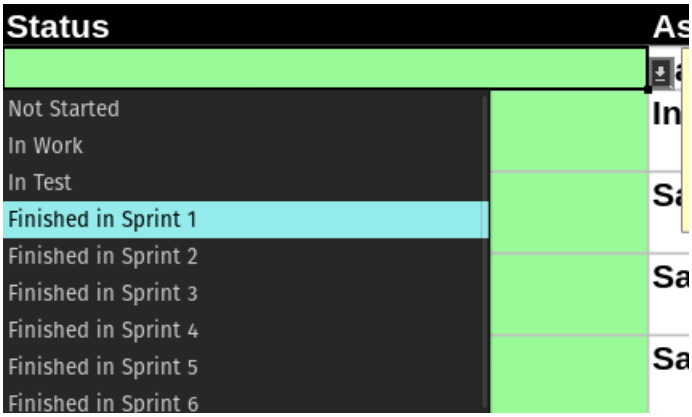
Product Backlog

- On the Product Backlog tab, fill in the green cells at the top and (at least) on rows 24-28 - the 4 features that will be graded this sprint plus Feature ID STORE that has been provided for you. (

You know these will be graded at the end of Sprint 1 because of the "1" in the Required column for each of those features.

Provide status for (at least) each feature to be graded this sprint.

- Planned (column F) is the sprint in which you plan to implement that feature - in this case, "1".
- Status (column G) is the sprint in which you actually implement that feature - in this case, "Finished in Sprint 1" if you complete it, "In Work" or "In Test" if you started but it's not complete yet, or "Not Started" if you didn't start it.



Remember, this should reflect *reality*, not dreams. We won't count off if you accurately report that you didn't finish the assignment. Rather, if you report that you finished the assignment *but did not*, then you may lose points.

Sprint 01 Backlog

- **On the Sprint 01 Backlog tab, fill in the Feature ID (column B), Description (column D), and Status (column E), for *at least* 2 to 5 tasks associated with *each* feature. For example, for Feature ID "CUST" (from the Product Backlog tab, cell A24), your tasks might be "Write Customer class", "Copy build.xml into repo and test compile Customer", "Write TestCustomer regression test" (if you choose to do so - not *required*), and "Add Customer to GitHub".**
  - Select the Feature ID (column B) from the drop-down list. This matches up with column A on the Product Backlog tab. We call this "traceability", because it answers the question, "Why are you doing this task?" The answer is, "It's need to implement feature CUST".
  - Write a brief to do item under Description (column D) similar to the example above.
  - Set the Status (column E) to the day of the sprint on which you finished that task. For example, if you finished the first task on Wednesday, set Status to "Completed Day 2". If you don't complete the task, set Status to "In Work" if started or leave blank if not started. If you don't complete all of the tasks for a specific Feature ID, then you can't set the associated Feature on the Product Backlog tab to "Finished in Sprint 1", right?

Feature ID	Assigned To	Task	Status	N
CUST		Write Customer class		
CUST		Copy build.xml into repo and test compile Customer		
CUST		Write TestCustomer regression test		
CUST		Add Customer.java and build.xml to GitHub		

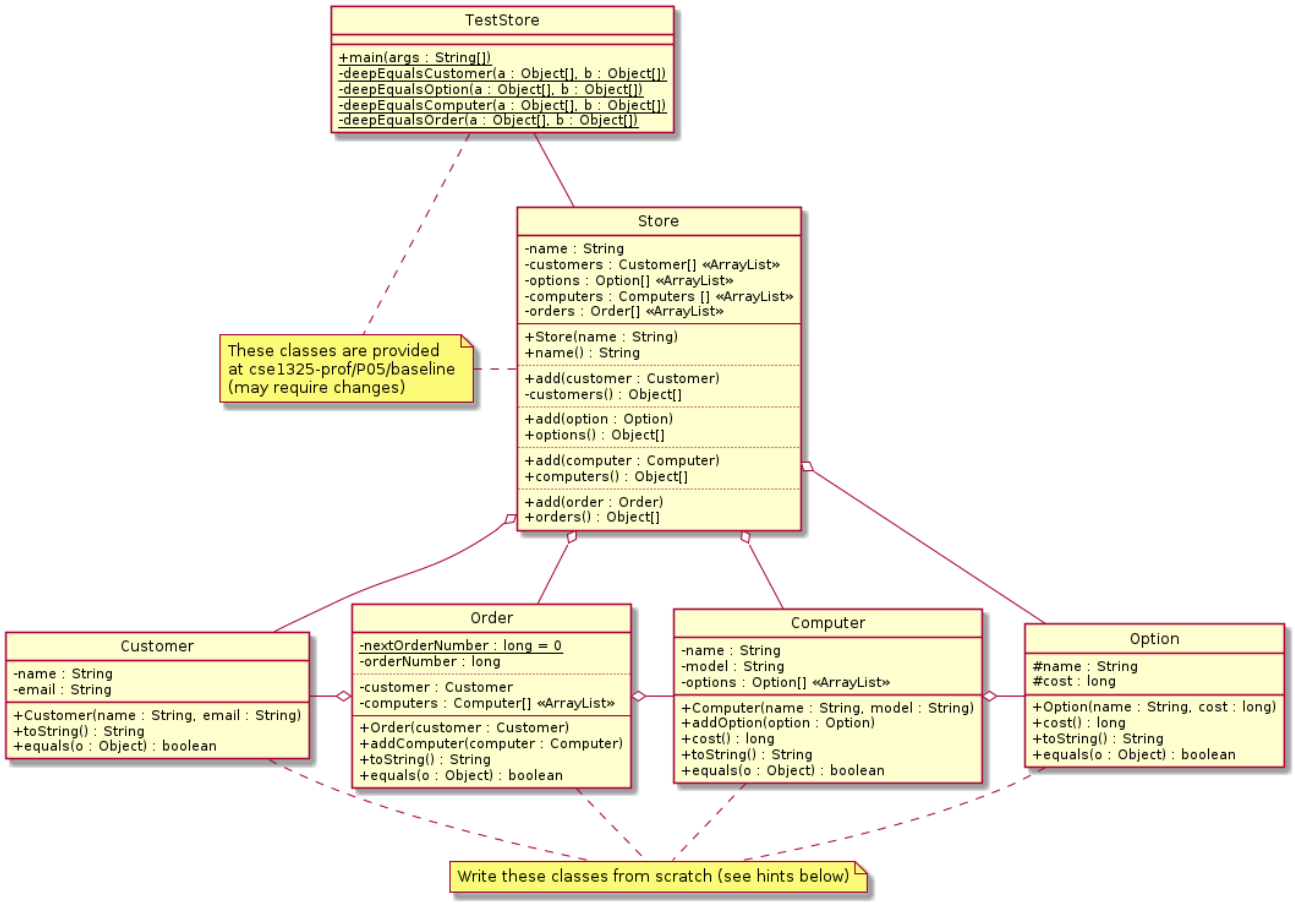
In Work
Completed Day 1
Completed Day 2
Completed Day 3

Now add, commit, and push. That's all you need to do.

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

Class Diagram

The diagram on the next page covers just Sprint 1. A more complete diagram will be provided once you have a basic understanding of Graphical User Interfaces (GUI) and Java Swing (our GUI toolkit). And updates to fix bugs and design issues are all but inevitable. (Dotted lines are purely visual and have no semantic meaning.)



Customer

I would start here, because it is simple and has no dependencies at all. A Customer is a person (or other legal entity) that has (or we hope will) buy a few (or a LOT of) ELSA computers.

Perform data validation on email in the constructor. The minimum validation rule for full credit is that a valid email must contain an '@' character, and must have at least one '.' character after the '@'. String has three `indexOf` methods from which you may choose. If the email is invalid, throw an `IllegalArgumentException` with a clear and appropriate message.

More enthusiastic students may (of course) use a regular expression. Even with a regex, proving that an email address conforms to RFC 5321 is surprising challenging! But a Java raw String regex like `^([a-zA-Z0-9_-.]+)@([a-zA-Z0-9_-.]+).([a-zA-Z]{2,5})$` would come close.

The suggested Customer solution implements `toString` as (for example) "Prof Rice (george.rice@uta.edu)", and checks for `equals` by comparing both the name and email fields. If you prefer to just compare email, that's fine as well - it's often used as a "unique key" in industry, I believe.

Option

This class also has no dependencies. An Option is a component that would go into a computer, such as a motherboard (the circuit board that is the heart of the computer), a Central Processing Unit (CPU, the "brains" of the machine"), Random Access Memory (RAM, temporary program and data storage), or a disk or flash drive (persistent program and data storage).

Perform data validation on cost in the constructor. If the cost is negative, throw an `IllegalArgumentException` with a clear and appropriate message. An Option instance with $0.00 cost MUST be supported, however, since some items need to be on the options list (such as a quick-start guide or invoice, perhaps) with no added cost to the Customer.

By the way, `cost` is the number of *cents*, not dollars, so divide by 100.0 when printing. A `long` integer is used for money to avoid round-off error (as we discussed in lecture) and because we're planning for ELSA to manage more than $21,474,836.47 (the maximum value for a Java int)!

The protected fields may give you a hint that inheritance is in Option's future.

The suggested solution would convert an Option `toString` something like "PNY CS900 500GB Internal SSD SATA ($29.99)", for example, and checks for `equals` by comparing both the name and cost.

Computer

Finally we get to an aggregate class! A Computer is a collection of Options (parts) with a name and model number that we can actually sell at ELSA.

No data validation is needed in the constructor.

The `addOption` method simply adds its parameter to the `options` field.

The `cost` method returns the sum of the `cost` of each Option in the `options` ArrayList. We'll add pricing (to cover overhead and profit) in a later sprint.

The suggested solution converts a Computer `toString` something like

```
SuperCalc (1Z200XL)
   Mainboard ($195.99)
   RAM ($93.28)
   SSD ($55.00)
   Case ($39.00)
```

The suggested solution checks for `equals` by (ahem) cheating a bit. Using an Old Programmer's Trick, it simply converts both Computers `toString` and compares the strings.

A more comprehensive `equals` would compare the scalar fields, then sort each `options` ArrayList before comparing each corresponding element (because the same Option objects may have been added in a different order). You do NOT need to do this, but feel free if you'd like the experience. (Hint for such an adventurous student: Your Option class will also need to *implement* the `Comparable` interface so that `Collections.sort` will work with Option objects as ArrayList elements.)

Order

The `Order` class aggregates one or more Computer objects into a `computers` ArrayList along with the `Customer` who is paying for the Order as well as an auto-generated `orderNumber`.

No data validation is needed in the constructor. The `orderNumber` should be initialized to static field value `nextOrderNumber++` as we have done before.

Method `addComputer` simply adds its parameter to the `computers` field.

The suggested solution converts an Order `toString` something like

```
Order 0 for Prof Rice (george.rice@uta.edu)

SuperCalc (1Z200XL)
   Mainboard ($200.00)
   RAM ($195.00)
   SSD ($55)
   Case ($39)
```

The Old Programmer's Trick mentioned for `Computer.equals` won't work for `Order` because of the `orderNumber` field - it will be different for every single object. So you'll need to compare the `customer` field and each element of the `computers` ArrayList explicitly, ignoring the `orderNumber` field.

Store

IMPORTANT: Because this assignment was provided to you a bit late, a complete implementation of this class is provided to you at `cse1325-prof/P05/baseline`. You may need to make minor changes for it to work with your classes, which is your responsibility to identify and implement.

The `Store` class simply manages the `Customer`, `Option`, `Computer`, and `Order` objects instanced by the system and acts as the store interface for the upcoming GUI.

The constructor sets the `name` of the store (say, "Arlington 303" once ELSA is a Fortune 500 corporation - color me "optimistic") with no data validation.

The overloaded `add` methods store the object provided as a parameter into the associated ArrayList, respectively. It's a good idea (but not required) to avoid adding duplicate objects into your ArrayLists. `ArrayList.contains` will help here for now, but we'll discover a better Collection to accomplish this automatically in Lecture 23.

The `customers()`, `options()`, `computers()`, and `orders()` methods return the elements of each associated ArrayList, respectively, as an `Object[]` array. Do NOT return an array of the actual type - many of the Swing widgets we will use in our Graphical User Interface require an `Object[]` array, so you will be making a LOT of work for yourself if you don't follow this advice. Not by coincidence, ArrayList's `toArray()` method returns an `Object[]` array of the ArrayList elements by default - just what you want!

The class diagram doesn't specify `toString` and `equals` methods for this class, because no need has been identified in design for them yet.

TestStore

IMPORTANT: A complete implementation of this class is provided to you at `cse1325-prof/P05/baseline`. You may need to make minor changes for it to work with your classes, which is your responsibility to identify and implement.

The `TestStore` class is a regression test (see Lecture 05) the provides a simple check that each of your classes can be instanced without losing the provided data. Remember:

- If no errors are detected, the program should produce NO output!
- If one or more errors are detected, each error should be reported to System.err with as much diagnostic information as possible. and a non-zero int should be returned from the program to notify the operating system that at least one failure was detected.

The `main` method runs at least one test vector against each class.

The private `deepEqualsCustomer`, `deepEqualsOption`, `deepEqualsComputer`, and `deepEqualsOrder` static methods accept two `Object[]` arrays and compares their contents for equality. "deep" means the algorithm iterates through each field of each of the corresponding elements, recursively, to ensure that all of the data in both arrays are "equal" per whatever `equals` method each of your classes has implemented.

Please include a working TestStore.java in your repository for the grader's use - the provided class with any modifications needed to work with your code, and / or a hard-coded set of objects for each class (except Store) that verifies that the code works, and / or an interactive command-line interface for instancing and displaying these classes (but remember that we will begin building a GUI interface next sprint).

You may use the provided class to verify that the other classes are working correctly before we begin building a user interface with them next sprint.