Assignment 1

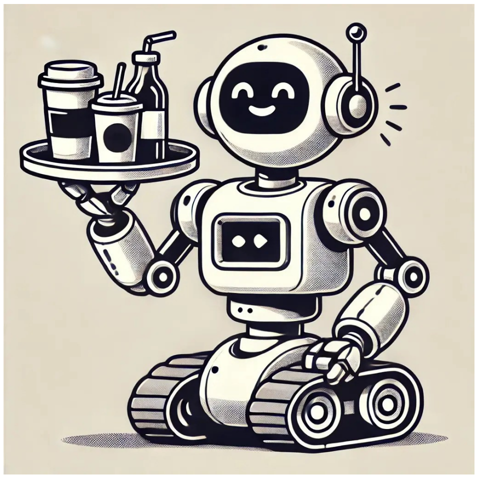
ECSE 250 Fall 2024

Posted on:	Sept. 24, 2024
Deadline:	Oct. 9, 2024 at 17:00 Montreal Time

Learning Objectives

This assignment is meant for you to practice what we have learned in class in the past few weeks. Several of the design decisions have been completed for you, but it is important for you to ask yourself why each choice has been made and whether there could be a better way of doing it. Some of our choices were influenced by the intention of having you practice most of what you have learned in class. Others, were dictated by our need to be able to fully test your assignments. Finally, we will not be grading your work on efficiency for the first assignment, but your implementation decisions can affect how efficient the code for this assignment is. Try to keep this in mind while coding. Make sure to ask yourselves why certain modifiers (`private`, `public`, `static`,...) were used and if you would have made the same decisions. We hope that the assignment will help you appreciate the importance of class design. We suggest you take the time to draw out the UML diagram. This should help you develop a clear picture of the relationship between all these classes.

A New Store



McGill has decided to unveil a new convenience store in Trottier that sells various food and drink items! You have been asked to program an AI assistant that monitors and handles all the store's purchases. You have learned object-oriented programming and data structure. The store's future is in your hands!

- We need to represent various products available in the store. Can we use OOD to reduce the amount of copy-and-paste and re-use our implementations as much as possible?
- We need to maintain a list of inventory, so we can add, remove, and retrieve items. Can we use a data structure for this?

General Instructions

- Submissions will be accepted up to 3 days late. Note that submitting one minute late is the same as submitting 3 days late. We will deduct points for any student who has to resubmit after the due date (i.e. late) irrespective of the reason, be it the wrong file submitted, the wrong file format submitted or any other reason. We will not accept any submission after the 3-day grace period. This policy will hold regardless of whether or not the student can provide proof that the assignment was indeed "done" on time.
- Your first late penalty is waved **automatically** at the end of the semester. There is no need to make an extra request.
- Don't worry if you realize that you made a mistake after you submitted: you can submit multiple times but **only the latest submission will be evaluated**. We encourage you to submit a first version a few days before the deadline (computer crashes do happen and Ed Lessons may be overloaded during rush hours).
- **Do not submit any other files, especially .class files and the tester files. Any failure to comply with these rules will give you an automatic 0.**
- Whenever you submit your files to Ed, **you will see the results**. If you do not see the results, your assignment is not submitted correctly. **If your assignment is not submitted correctly, you will get an automatic 0. If your submission does not compile on ED, you will get an automatic 0.**
- The assignment shall be graded automatically on ED. **Requests to evaluate the assignment manually shall not be entertained, and it might result in your final marks being lower than the results from the auto-tests.** Please make sure that you follow the instructions closely or your code may fail to pass the automatic tests.
- Next week, a mini-tester will also be posted. We encourage you to modify and expand it. ***You are welcome to share your tester code with other students on Ed.*** Try to identify tricky cases. Do **not** hand in your tester code.
- **Failure to comply with any of these rules will be penalized**. If anything is unclear, it is up to you to clarify it by asking either directly a TA during office hours or on Ed.
- In addition to the required methods, you are free to add as many other **private** methods and classes as you want (granted that all of the classes and methods listed below have been made). Unless otherwise specified, no other non-private fields or methods are allowed.
- You can also add `public toString()` methods in each class to help with the debugging process, if you find it helpful.

Submission Instructions

- These are the files you should be submitting on Ed:
  - `StoreItem.java`
  - `Drink.java`
  - `FizzWiz.java`
  - `SnoozeJuice.java`
  - `Snack.java`
  - `MyDate.java`
  - `ItemList.java`
  - `Store.java`
- You will have to create all the above classes from scratch. You are NOT allowed to import any other class (including for example `ArrayList` or `LinkedList`). **Any failure to comply with these rules will give you an automatic 0.**
- Never make copies (neither shallow, nor deep) of any of the objects the methods return or receive as input, unless specified in the instructions.

STEP 1: Building in Pieces

Let's start by writing a few abstract classes. These classes should be part of a package named `items` which is a subpackage of `assignment1`.

StoreItem

Write an `abstract` class `StoreItem`.

- The class `StoreItem` has the following `private` fields:
  - A `double` indicating the price of the item.
  - An `int` representing the happiness index associated with this item. This is what measure of how much joy the item brings when consumed.
- The `StoreItem` class will also have the following `public` methods:
  - A constructor that takes as input a `double` for the price and an `int` for the happiness index. The constructor uses its inputs to initialize the corresponding fields. If either of the values provided is negative, then an `IllegalArgumentException` should be thrown. To make the program raise the exception, write the following statement:

    ```
    throw new IllegalArgumentException(errMsg);
    ```

    where `errMsg` is a `String` containing an error message of your choice.
  - A `final getPrice()` method that retrieves the price of this item.
  - A `getHappinessIndex()` method that retrieves the happiness index of this item.
  - An `equals()` method that takes an `Object` as input and returns `true` if and only if it matches `this` in type, price, and happiness index. Be careful when comparing double: you do not want to compare values of type double directly! Instead you should consider the two values to be the same if their difference is very small, i.e. less than `0.001`.

Drink

Now, we will write an abstract class called `Drink` in the same package which extends `StoreItem`.

- The `Drink` class includes the following `public static` fields:
  - An `int MAX_PACK_SIZE` representing the maximum number of bottles a drink pack can contain. The default value is 6.
  - An `int BUZZY_HAPPINESS_BOOST` indicating the amount by which the happiness index of a drink will be increased if the drink is *buzzy*. Set the default value to 1.
- The class should also have the following non-static fields:
  - A `protected int` representing the number of bottles available in this pack.

  - A `private boolean` indicating whether or not this drink is *buzzy*. Buzzy drinks generate extra excitement when consumed.

- The class must also have the following `public` methods:
  - A constructor that takes as input a `double` for the price, an `int` for the happiness index, an `int` for the number of bottles, and `boolean` for the buzziness. The method updates the corresponding fields. Note that the happiness index and price provided for a drink are representative of a single bottle, and not of the entire pack.
  - A `getNumOfBottles()` method that retrieves the number of bottles of this drink.
  - An `equals()` method that takes an `Object` as input and returns `true` if and only if it matches `this` in type, price, happiness index, and buzziness. You should not be comparing the number of bottles! Note that you do not want to rewrite code that you have already written in the superclass. How can you access methods from the superclass that have been overridden?
  - A `getHappinessIndex()` method that overrides the one from `StoreItem`. For buzzy drinks, their happiness index should be boosted by the amount stored in the appropriate class variable.
  - A `combine()` method that takes another `Drink` as input and, if the input drink is considered equal to the current drink, combines as many bottles from the input drink as possible with this one. The number of bottles for both drinks should be updated to reflect the combination. The method returns `true` if the drinks were successfully combined (i.e., their number of bottles changed), and `false` otherwise. Note that no drink can exceed the `MAX_PACK_SIZE` number of bottles.

    For example, assume `MAX_PACK_SIZE` is 4, `Drink` A has 3 bottles in its pack, `Drink` B has 3 bottles in its pack, and the two drinks are equivalent. Then after executing `A.combine(B)`, A has 4 bottles, B has 2 bottles, and the method returns `true`. On the other hand, if A had 4 bottles in its pack, `A.combine(B)` would return `false`.
  - An `abstract getPortion()` method that takes as input an `int` and returns a `Drink`.

STEP 2: Expanding the Collection

Now that we have built some abstract classes, we can construct classes that rely on them! Inside the same package (i.e. `items`) create the following classes:

MyDate

We need to be able to record the expiration date of our products. We'll define a new data type that we can use as a replacement! Write a class called `MyDate`.

- The class has the following `private` fields:
  - An `int` representing the day
  - An `int` representing the month
  - An `int` representing the year
- The class should also include two `public static` fields:
  - An `int[] SUMMER_MONTHS` representing the hot months of the year. By default, the summer months are June, July, and August
  - An `int[] WINTER_MONTHS` representing the cold months of the year. By default, the winter months are December, January, and February.
- Finally, the class includes the following public methods
  - A constructor that takes as input 3 `int`s representing the day, month and year in this order, and updates the relevant fields. The constructor should verify that the day and month are valid entries for a calendar date (assuming no leap years), and throw a `IllegalArgumentException` otherwise.
  - `getDay(), getMonth(), getYear()` methods that take no argument and return the corresponding field.
  - An `equals()` method which takes as input an `Object` and returns `true` if it matches `this` in type, day, month and year. Otherwise, the method returns `false`.
  - A `static` method called `today()` that takes no input and returns today's date as a `MyDate` object. You can use Java functions `java.time.LocalDate.now().getYear()` to grab the year, `java.time.LocalDate.now().getMonthValue()` to grab the month, and `java.time.LocalDate.now().getDayOfMonth()` to grab the day.
  - You can add a `toString()` method to get it in that lovely YYYY-MM-DD format.

  In the class `MyDate`, you may add additional methods and fields, whether public or private.

Snack

We begin by writing a class called `Snack` which extends `StoreItem`

- The `Snack` class has the following `private` fields:

  - A `string` representing the type of snack
  - A `MyDate` representing the date of expiration

- The `Snack` class also has the following `public` methods:
  - A constructor that takes as input a `double` for the price, an `int` for the happiness index, a `string` for the snack type, and a `MyDate` representing the expiration date, and updates the corresponding fields.
  - An `equals()` method which takes as input an `Object` and returns `true` if and only if it matches `this` in type, price, happiness index, snack type, **and expiration date.** ~~Note that we do not compare their expiration dates.~~
  - An `isExpired()` method that takes no inputs and returns a boolean indicating whether or not this snack's expiration date has passed.
  - A `getHappinessIndex()` method that overrides the one from `StoreItem`. For obvious reasons, the happiness gained from consuming a snack plummets when the snack in question tastes a little stale. Hence, if the snack is expired, its happiness index should be halved.

FizzWiz

Next, we'll focus on drinks and write a class called `FizzWiz`, which extends `Drink`. This class represents a carbonated drink, famous for its bubbly and fizzy quality.

- The `FizzWiz` class should include the following `public` methods:
  - A constructor that takes as input a `double` for the price, an `int` for the happiness index, and anther `int` for the number of bottles. Note that `FizzWiz` is a very buzzy drink!
  - An `equals()` method that takes an `Object` as input and returns `true` if and only if it matches `this` in type, price, happiness index, and buzziness.
  - A `getPortion()` method that takes as input an `int` indicating the number of bottles desired. If enough bottles are available in this drink, it returns a new `FizzWiz` with the specified number of bottles, while also updating the number of bottles in this drink accordingly. The drink returned should be equivalent to this `FizzWiz`. If there are not enough bottles available, the method should return `null`.

SnoozeJuice

Let's now add another subclass of `Drink` named `SnoozeJuice`. This represents a relaxing drink like herbal tea, designed to help someone unwind or sleep - just the right beverage our students need before an intense study session! Like herbal teas, this beverage can be served hot or cold, so we should include a `private double` field to track its temperature (in Celsius).

- The class should have the following `private` field.
  - A `double` representing the temperature in Celsius.

- The class should also have the following `public` methods:
  - A constructor that takes as input a `double` for the price, an `int` for the happiness index, anther `int` for the number of bottles, and one more `double` for the temperature. As you can guess, snooze juices are not buzzy at all!
  - An `equals()` method that takes an `Object` as input and returns `true` if and only if it matches `this` in type, price, happiness index, buzziness, and temperature. Remember that when comparing doubles you should look at their difference and ensure it is smaller than 0.001.
  - A `getHappinessIndex()` method that overrides the one from `Drink`. The serving temperature of a drink can truly make or break the experience. Nobody enjoys a brain freeze or hot blisters: if the temperature is below 4°C or above 65°C, the happiness index drops to 0. On the other hand, a cold drink (with a temperature between 4°C and 10°C, inclusive) during a hot summer month is what everyone craves, so it should get a well-deserved boost. Add a `public static int` field named `HOT_COLD_BOOST` to track this boost, and use it to increase the happiness index when appropriate. Set a default value of 2 to `HOT_COLD_BOOST`. Since we're at it, the same boost applies when a hot drink (with a temperature between 55°C and 65°C, inclusive) is served during a cold winter month. The current date should be used to decide whether or not the boost should be applied.
  - A `getPortion()` method that takes as input an `int` indicating the number of bottles desired. If enough bottles are available in this drink, it returns a new `SnoozeJuice` with the specified number of bottles, while also updating the number of bottles in this drink accordingly. The `SnoozeJuice` returned should be equivalent to this one. If there are not enough bottles available, the method should return `null`.

STEP 3: Adding a Data Structure

Before we create our store, it would be nice to dynamically adjust the type and size of our offerings. To do this, we will create our own `ItemList` class.

- The class will be part of the `assignment1` package. We want to stress once again that you are not allowed to import any external classes for this assignment. You will have to import at least one class from `assignment1.items`.
- The `ItemList` class should have the following `private` field:
  - An array of `StoreItem`s which will be used to store the items that are part of the `ItemList`.
- The class should also have the following `public` methods:
  - A constructor that takes no inputs and creates an empty `ItemList`. You may update the relevant fields however you like, but the `ItemList` fields should reflect the fact that no items are currently stored in this list.
  - A `getSize()` method that takes no inputs and returns the number of items that are in the `ItemList`
  - A `getAllItems()` method that takes no input and returns all the items in the `ItemList` (in any order) in a `StoreItem` array. This array should not contain any unnecessary duplicates or `null` elements and should be exactly the size returned by `this.getSize()`.
  - An `addItem()` method which takes as input a `StoreItem` and does not return any value. The method adds the item to the `ItemList`. This may require updates to various fields, depending on your implementation.
  - A `removeItem()` method that takes a `StoreItem` as input and returns a `StoreItem`. The method removes the first occurrence of an item in the list that is equivalent to the specified item. The removed item is then returned by the method. If no such item exists, the method returns `null`.
  - A `findEqualItems()` method that takes as input a `StoreItem` and returns an array of `StoreItem`s containing all of the items from this list that are equivalent to the specified one. If no equivalent items are found, the method returns an array of size 0.
- In this class, you are permitted to add additional `private` fields, along with any `private` methods you find necessary.

STEP 4: Finalizing the Store

Now that the basis is complete, it is time to create our store!

- The `Store` class belongs to the package `assignment1`. Inside it, you can import all classes from `assignment1.items`.
- The class has the following `private` fields:
  - An `ItemList`, representing the items available at the store
  - A `double` representing the total revenue of the store
- The `Store` also has the following `public` methods:
  - A constructor that takes as input an `ItemList`, and updates the corresponding fields. Note that the store will initially have zero revenue.
  - Two get methods `getRevenue()` and `getItems()` to retreive the values stored in the respective fields.
  - A `cleanUp()` method that takes no inputs and returns no value. The method goes through the store's inventory and tosses out any snacks that have gone stale - nobody likes expired goodies!
  - A `completeSale()` method that takes an `ItemList` as input and returns an `int`. The method should purchase every item from the given `ItemList` from this store and compute the total happiness index gained from consuming all the items. It should also update the store's revenue accordingly. Be careful: not all the items in the list may belong to the store, and for drinks, a specific number of bottles will be required. If the number of bottles required is available, only those bottles should be sold, no more. If fewer bottles are available, only the available ones can be purchased. When computing the happiness index and updating the store's revenue, you must account for all these factors, and the remaining inventory in the store should reflect these changes. **That is, items that have been purchased should no longer appear in the store, including drink objects that have 0 bottles left.**
  - A `refillDrinkInventory()` method that takes an array of `Drink`s as input and helps restock the store. But watch out - space at McGill is limited, so every bottle counts! If we already have some bottles of the same drink in stock, be sure to squeeze them together as efficiently as possible. Only add a new drink to the shelves if there's really no more room. Note that, you cannot make any assumptions about the number of bottles in each drink item.

Rubric

Here is a description that outlines what you need to demonstrate to achieve a particular competency level (Proficiency, Approaching Mastery, Mastery).

- **Proficiency:** At this level, you are expected to demonstrate a good understanding of basic object-oriented programming (OOP) concepts. The tasks at this level involve a straightforward application of what has been covered in class. This includes implementing constructors, getters, and simple methods to handle core functionality, such as managing the price and happiness index of store items, adding items to an `ItemList`, and overriding `equals()` where needed to ensure proper comparison of store items. Achieving proficiency means you are able to apply these fundamental concepts correctly and confidently, even if the problems are simple and familiar.
- **Approaching Mastery:** At this level, you are expected to demonstrate a solid understanding of OOP by handling more complex behaviors specific to each object. This includes manipulating object states dynamically (e.g., calculating happiness based on buzziness or expiration), work with methods like `getPortion()` to handle partial sales, and understanding how objects interact with each other. You'll also need to carefully override methods and work with item comparison in more advanced scenarios. This level involves applying class concepts to situations that may be more complex than what was directly shown in class, but still rely on a firm grasp of core OOP principles.
- **Mastery:** At this level, you are expected to demonstrate a deep understanding of object-oriented programming by combining various pieces of what you've learned to solve new problems. You will need to design simple algorithms that weren't directly presented in class, applying your knowledge in real-world scenarios. This includes creating methods like `completeSale()` and `refillDrinkInventory()`, where you manage dynamic changes to inventory and handle partial sales. These tasks require not only method overriding but also thoughtful interaction between objects, demonstrating your ability to handle polymorphism, dynamic object manipulation, and more advanced OOP techniques effectively. Mastery also involves recognizing and handling edge cases, ensuring that your solutions are robust. Successfully achieving mastery shows that you can think creatively, combine existing tools and knowledge in new ways, and design algorithms that address the complexities of real-world situations.