CSE 8B: Introduction to Programming and Computational Problem Solving - 2

Assignment 6

Inheritance and Polymorphism

Due: Wednesday, May 24, 11:59 PM

Be sure to start this assignment as EARLY as possible! This document may be long, but a majority of the methods required are relatively simple to implement. You got this!

Learning goals:

- Develop further mastery of classes with getters/setters.
- Practice inheritance by defining multiple superclasses and subclasses.
- Apply knowledge of polymorphism in several methods.

Your grade will be determined by your most recent submission. If you submit to Gradescope after the deadline, it will be marked late and the late penalty will apply regardless of whether you had past submissions before the deadline.

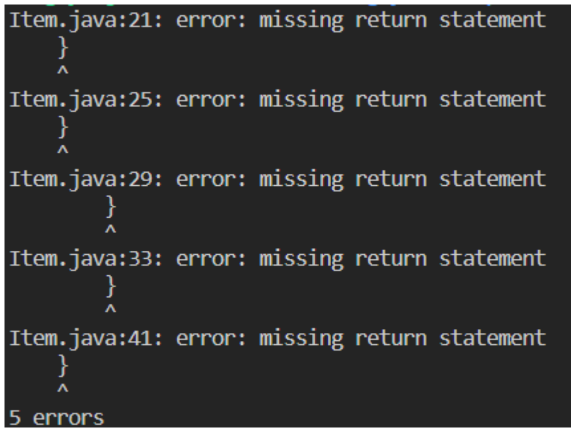
If your code does not compile on Gradescope, you will receive an automatic zero on the assignment.

Coding Style (10 points)

For this programming assignment, we will be enforcing the CSE 8B Coding Style Guidelines. These guidelines can also be found on Canvas. Please ensure to have COMPLETE file headers, class headers, and method headers, to use descriptive variable names and proper indentation, and to avoid using magic numbers.

Part 0: Getting started with the starter code (0 points)

1. If using a personal computer, then ensure your Java software development environment does not have any issues. If there are any issues, then review Assignment 1, or come to the office/lab hours before you start Assignment 6.
2. First, navigate to the `cse8b` folder you created in Assignment 1 and create a new folder named `assignment6`.
3. Download the starter code. You can download the starter code from Piazza → Resources → Homework → `assignment6.zip`. The starter code contains nine files: `Assignment6.java`, `Consumable.java`, `Drink.java`, `Item.java`, `Nonconsumable.java`, `Notebook.java`, `Snack.java`, `Store.java`, and `Toiletry.java`. Place the starter code within the `assignment6` folder you just created.
4. Compile the starter code within the `assignment6` folder. You can compile all files using the single command `javac *.java` and you should get a compiler error saying that the methods in either `Item.java`, `Consumable.java`, `Drink.java`, `Snack.java`, `Nonconsumable.java`, `Notebook.java`, or `Toiletry.java` are missing return statements and have not been implemented yet. For example:



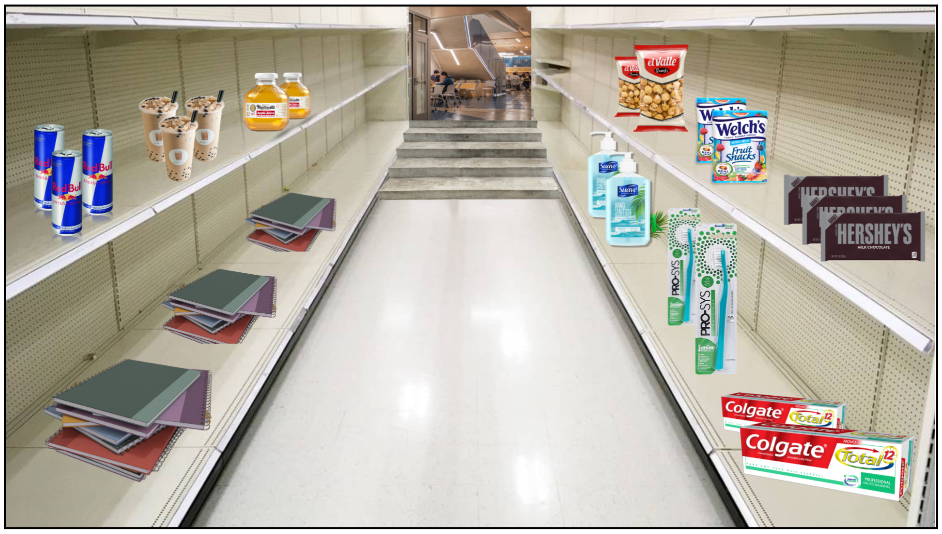
```
Item.java:21: error: missing return statement
    }
    ^
Item.java:25: error: missing return statement
    }
    ^
Item.java:29: error: missing return statement
        }
        ^
Item.java:33: error: missing return statement
        }
        ^
Item.java:41: error: missing return statement
    }
    ^
5 errors
```

5. The objective of this assignment is to get the classes working by implementing the class methods and testing them.

Part 1: Overview

Scenario:

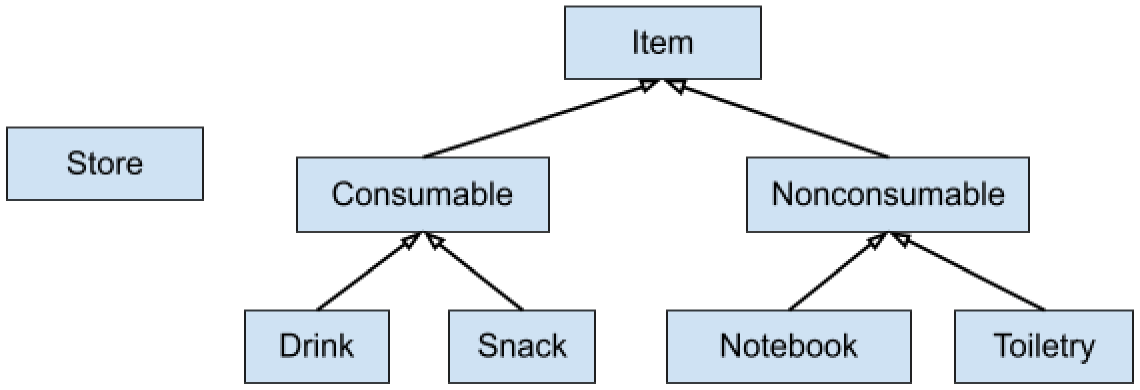
A new type of store has opened up in the basement below the basement of the UCSD CSE Building! Walking in, the store seems like a lifeline for CSE students! It sells four specific categories of Items: Drinks and Snacks (Consumables), and Notebooks and Toiletries (Nonconsumables).



We are not sponsored by any of the products above. 🙁

Logistics:

For this assignment, you will be implementing the classes shown in the following UML diagram.



Each rectangle in the UML diagram represents a class, and each directional hollow triangle represents an inheritance relationship from a subclass to a superclass. Notice how `Store` is unrelated to `Item` and `Item`'s subclasses.

Before you start programming, please take some time to review the starter code and to read the instructions below **CAREFULLY**. Some methods are already implemented for you, but you will still need to supply those methods with a method header for coding style points. You should fully understand the purpose of each variable and the usage for each method before you implement anything.

NOTE 1: You must NOT change any data field or method signature in the starter code. As such, do NOT add any additional parameters to methods, and do NOT import any Java packages. Feel free to add any helper methods if desired.

NOTE 2: You must implement and comment on everything with a `TODO`**.** Do NOT forget to adhere to the CSE 8B style guidelines.

NOTE 3: You can assume that all inputs will be valid. For example, all `int` and `double` arguments will be non-negative.

Be sure to compile your code often, so that you can catch compile errors early on! Recall, to compile multiple Java files, use:

```
> javac *.java
```

You will be implementing methods in every provided Java class, with the `Store` class having the majority of the functionality of this program.

Notice how each member field is declared `private`.

Recall: This means that the member is only visible within the class, not from any other class. In other words, you will need to use accessors (i.e., getter methods) and mutators (i.e., setter methods) to access and modify, respectively, these `private` members. You must also use the `this` keyword to modify and access member variables hidden by local variables.

Part 2: `Item.java` (0 points)

First, you need to implement the class named `Item`. This is the superclass of most of the other classes in this assignment (as seen in the UML diagram). The `Item` class defines the default behavior of methods (e.g., the method `equals`, which is later overridden by subclasses) of all the subtype classes in this assignment. Although this class is worth 0 points, you will not be able to complete the rest of the classes without this class working.

The `Item` class has four data fields:

1. **`private String name`**
2. **`private double price`**
3. **`private String highLevelType`**

   → must be string literal `"Consumable"`, `"Nonconsumable"`, or `"Item"` (described below)
4. **`private String type`**

   → must be string literal `"Drink"`, `"Snack"`, `"Notebook"`, `"Toiletry"`, or `"Untyped Item"` (described below)

Notice how each member field is declared `private`. This means that the member is only visible *within* the class, not from any other class. In other words, you will need to use accessors (i.e., getter methods) and mutators (i.e., setter methods) to access and modify, respectively, these `private` members. **You must also use the `this` keyword to access member variables hidden by local variables.**

There are two `highLevelType` classes of `Items`: `Consumable` and `Nonconsumable`. The `type` member describes the more specific type of an `Item`. There are four "type" classes: `Drink`, `Snack`, `Notebook`, and `Toiletry`. As seen in the UML diagram above, `Drink` and `Snack` "is-a" `Consumable`, and `Notebook` and `Toiletry` "is-a" `Nonconsumable`.

First, in `Item.java`, complete the getters and setters to access and mutate, respectively, the data fields. You may notice that there are a lot of getters, but the only getters/setters that you have to implement are:

1. **`public String getName()`**
2. **`public String getType()`**
3. **`public double getPrice()`**
4. **`public String getHighLevelType()`**
5. **`public void setPrice(double price)`**

You also need to implement the following constructor:

1. **`public Item(String name, double price, String type, String highLevelType)`**

- This constructor sets the corresponding instance variables of the object to what the caller of the constructor passed in as arguments. Remember, you must use the `this` keyword to access member variables hidden by local variables.

Then, implement the following method:

1. **`public boolean equals(Item item)`**
   - This method must return `true` **only** when the current object (referring to this object - this entire writeup will use the same terminology for this) and the input item have the same `name`, `price`, `highLevelType`, AND `type`. Otherwise, it must return `false`. By implementing the `equals` method, it allows the user of the class to compare `Item` objects on deep equality (similar to deep copy for arrays). Rather than just checking for equality of reference, it will compare equality by checking the contents of the object instead.

Part 3: `Consumable.java` and `Nonconsumable.java` (10 points)

`Consumable` and `Nonconsumable` are two subclasses of `Item`. Notice the `Consumable` and `Nonconsumable` classes both extend `Item`, telling Java to create the superclass/subclass relationship. Complete all remaining constructors and methods in those classes. The no-arg constructors and `toString()` methods are already provided to you. You can use the given implementation of the no-arg constructor as guidance for the other constructor. **Remember, you must NOT change the existing signature or the fields.**

Consumable

The `Consumable` class has two fields:

1. **`private int calories`**
   → an integer denoting the number of calories of this consumable item
2. **`private int daysLeft`**
   → an integer denoting the number of days until this consumable item expires

Implement the following constructor and methods:

1. **`public Consumable(String name, double price, String type,`**
   `int calories, int daysLeft)`

    - This constructor must set the `name`, `price`, `type`, and `highLevelType` in its superclass (HINT: use [`super`]() to call the superclass constructor!) from the constructor parameters and setting the `highLevelType` to the string literal `"Consumable"`. Then, set the `calories` and `daysLeft` members using the remaining constructor parameters. Remember, you must use the [`this`]() keyword to access member variables hidden by local variables.
2. **`public int getCalories()`**
    - Simple getter method that returns the `calories` instance variable.
3. **`public int getDaysLeft()`**
    - Simple getter method that returns the `daysLeft` instance variable.
4. **`public boolean equals(Item item)`**
    - This method overrides the `equals()` method in `Item`. This method checks whether the current `Consumable` object is considered equal to the input `Item`. This method must return `true` **only** when the current object has the same `name`, `price`, `highLevelType`, `type`, `calories`, AND `daysLeft`. Otherwise, it must return `false`. (HINT: use the `equals()` method from the superclass!) Remember, you must use the [`super`]() keyword to access a method in the superclass hidden by a method in the current (sub)class.

(**Note:** When overriding a superclass method, remember to use the `@Override` annotation (see lecture 12, slide 33). You will be using the override annotation for all overridden `equals()` methods throughout this assignment.)

Nonconsumable

The `Nonconsumable` class has one field:

1. **`private int maxNumUsages`**

→ an integer denoting the number of times a `Nonconsumable` item can be used before it expires. For example, an arbitrary `Toiletry` may be used a maximum of 120 times.

Implement the following constructor and methods:

1. **`public Nonconsumable(String name, double price, String type, int maxNumUsages)`**
    - This constructor must set the `name`, `price`, `type`, and `highLevelType` in its superclass (HINT: use [`super`]() to call the superclass constructor!) from the

constructor parameters and setting the `highLevelType` to the string literal `"Nonconsumable"`. Then, set the `maxNumberUsage` member using the remaining constructor parameter. Remember, you must use the [`this`] keyword to access member variables hidden by local variables.

2. **`public int getMaxNumUsages()`**
   - Simple getter method that returns the `maxNumUsages` instance variable.
3. **`public boolean equals(Item item)`**
   - This method overrides the `equals()` method in `Item`. This method checks whether the current `Nonconsumable` object is considered equal to the input `Item`. This method must return `true` **only** when the current object has the same `name`, `price`, `highLevelType`, `type`, and `maxNumUsages`. Otherwise, it must return `false`. (HINT: use the `equals()` method from the superclass!) Remember, you must use the [`super`] keyword to access a method in the superclass hidden by a method in the current (sub)class.

Part 4a: Drink.java and Snack.java (10 points)

`Drink` and `Snack` are two subclasses of `Consumable` (since they are things you can consume). **Complete all remaining constructors and methods in these classes.**

Drink

The Drink class has one field:

1. **`private double liters`**

   → a double denoting the number of liters contained in a drink item

Implement the following methods:

1. **`public Drink(String name, double price, int calories,`**
   `int daysLeft, double liters)`
   - This constructor must set the `name`, `price`, `type`, `calories`, and `daysLeft` in its superclass (HINT: use [`super`] to call the superclass constructor!) Then, set the `liters` member using the remaining constructor parameter. Remember, you

must use the `this` keyword to access member variables hidden by local variables.

2. **`public double getLiters()`**
   - Simple getter method that returns the `liters` instance variable.
3. **`public boolean equals(Item item)`**
   - This method overrides the `equals()` method in `Consumable`. This method checks whether the current `Drink` object is considered equal to the input `Item`. This method must return `true` **only** when the current object has the same `name`, `price`, `highLevelType`, `type`, `calories`, `daysLeft`, and `liters`. Otherwise, it must return `false`.(HINT: use the `equals()` method from the superclass!) Remember, you must use the `super` keyword to access a method in the superclass hidden by a method in the current (sub)class.

Snack

The Snack class has one field:

1. **`private String texture`**

   → a String denoting the texture of the snack upon consumption.

Implement the following methods:

1. **`public Snack(String name, double price, int calories,`**
   `int daysLeft, String texture)`
   - This constructor must set the `name`, `price`, `type`, `calories`, and `daysLeft` in its superclasses (HINT: use `super` to call the superclass constructor!) from the constructor parameters. Then, set the `texture` member using the remaining constructor parameter. Remember, you must use the `this` keyword to access member variables hidden by local variables.
2. **`public String getTexture()`**
   - Simple getter method that returns the `texture` instance variable.
3. **`public boolean equals(Item item)`**

- This method overrides the `equals()` method in `Consumable`. This method checks whether the current `Snack` object is considered equal to the input `Item`. This method must return `true` **only** when the current object has the same `name`, `price`, `highLevelType`, `type`, `calories`, `daysLeft`, AND `texture`. Otherwise, it must return `false`. Remember, you must use the [`super`] keyword to access a method in the superclass hidden by a method in the current (sub)class.

Part 4b: Notebook.java and Toiletry.java (10 points)

`Notebook` and `Toiletry` are two subclasses of `Nonconsumable` (since they are things you cannot consume). **Complete all remaining constructors and methods in these classes.**

Notebook

The `Notebook` class has three fields:

1. **`private String color`**
   → a String denoting the color of the notebook.
2. **`private boolean isCollegeRuled`**
   → a boolean denoting whether the notebook is college ruled (`true`) or wide ruled (`false`).
3. **`private int subject`**
   → an int denoting the subject of the notebook (**only use** 1, 3, or 5). The subject of the notebook is how many sections the notebook has. For instance, a 5-subject notebook will be divided into five sections (outlined in blue below).

Implement the following constructor and methods:

1. **`public Notebook(String name, double price, int maxNumUsages, String color, boolean isCollegeRuled, int subject)`**
   - This constructor must set the `name`, `price`, `type`, and `maxNumUsages` in its superclass (HINT: use `super` to call the superclass constructor!) from the constructor parameters. Then, set the `color`, `isCollegeRuled`, and `subject` members using the remaining constructor parameters. Remember, you must use the `this` keyword to access member variables hidden by local variables.
2. **`public String getColor()`**
   - Simple getter method that returns the `color` instance variable.
3. **`public boolean getIsCollegeRuled()`**
   - Simple getter method that returns the `isCollegeRuled` instance variable.
4. **`public int getSubject()`**
   - Simple getter method that returns the `subject` instance variable.
5. **`public boolean equals(Item item)`**
   - This method overrides the `equals()` method in `Nonconsumable`. This method checks whether the current `Notebook` object is considered equal to the input `Item`. This method must return `true` **only** when the current object has the same `name`, `price`, `highLevelType`, `type`, `maxNumUsages`, `color`, `isCollegeRuled`, and `subject`. Otherwise, it must return `false`. Remember, you must use the `super` keyword to access a method in the superclass hidden by a method in the current (sub)class.

Toiletry

The `Toiletry` class has two fields:

1. **`private int packSize`**
   → an int denoting the number of items in the pack (**only use** 1, 2, or 3).
2. **`private boolean IsBrandName`**
   → a boolean denoting whether the item is a brand name (`true`) or a no-name brand (`false`).

Implement the following constructor and methods:

1. **`public Toiletry(String name, double price, int maxNumUsages, int packSize, boolean isBrandName)`**

- This constructor must set the `name`, `price`, `type`, and `maxNumUsages` in its superclass (HINT: use `super` to call the superclass constructor!) from the constructor parameters. Then, set the `packSize` and `isBrandName` members using the remaining constructor parameter. Remember, you must use the `this` keyword to access member variables hidden by local variables.

6. **`public int getPackSize()`**
   - Simple getter method that returns the `packSize` instance variable.
7. **`public boolean getIsBrandName()`**
   - Simple getter method that returns the `isBrandName` instance variable.
8. **`public boolean equals(Item item)`**
   - This method overrides the `equals()` method in `Nonconsumable`. This method checks whether the current `Toiletry` object is considered equal to the input `Item`. This method must return `true` **only** when the current object has the same `name`, `price`, `highLevelType`, `type`, `maxNumUsages`, `packSize` and `isBrandName`. Otherwise, it must return `false`. **Follow the same hint for the previous equals methods above!**

Part 5: Store.java (50 points)

Finally, the cool part! You are now an employee of the UCSD CSE Basement-Basement store, with your main job being to keep track of inventory (not selling items!). You will be implementing a `Store` class with various unique methods to help you keep track of all the `Items` in the store.

Store Basic Methods (5/50 points)

The `Store` class has one field:

1. **`private ArrayList<Item> itemList`**

   → an `ArrayList` containing `Item` objects. Please refer to the ArrayList documentation and the Lecture 11 slides for more information about `ArrayList` methods. Recall: an

`ArrayList` is like an array except that it could change size as you and/or remove elements from it.

First, implement the Store constructor and simple getter method.

1. **`public Store()`**
   - The no-arg constructor must initialize `itemList` to an empty `ArrayList` of `Item` elements.

2. **`public ArrayList<Item> getItemList()`**
   - Getter method for `itemList`.

Next, implement a method called `addToItemList` overloaded with **two** different implementations.

1. **`public void addToItemList(Item item)`**
   - Adds `item` to `itemList` (a single `Item`)
2. **`public void addToItemList(Item[] items)`**
   - Adds each `Item` in `items` to `itemList` (can be multiple `Item` objects)

Now, we will start implementing the "meat" of the Store class **- 3** methods with unique functionality. This is where you will be applying your knowledge of **Polymorphism**.

Method 1 - compareFinalUsageDate (15/50 points)

The first method is `compareFinalUsageDate`. The method signature for it is:
`public static int compareFinalUsageDate(Item item1, Item item2, double rateOfUsage)`

The first thing we need to notice is that `Drink` objects and `Snack` objects each have a `daysLeft` field because they have an "is-a" relationship with `Consumable`, and that `Notebook` objects and `Toiletry` objects each have a `maxNumUsages` field because they have an "is-a" relationship with `Nonconsumable`.
<u>Goal:</u> Given two `Item` objects, `item1` and `item2` , compare the final usage dates of the two items.

- Return `-1` (negative one) if `item1` will expire before `item2`
- Return `0` if `item1` will expire at the same exact time as `item2`
- Return `1` if `item1` will expire after `item2`

To make this comparison, we need to be able to **compute how many days left** an `Item` will last until it expires. This changes depending on the `highLevelType` of the `Item` .

- If an `Item` "is-a" Consumable (which you can check by looking at its `highLevelType`), the number of days left the Item has (i.e., the number of days until it expires) is its `daysLeft` field.
- If an Item "is-a" Nonconsumable, the number of days left the Item has (i.e., the number of days until it expires) follows the formula:

maxNumUsages / rateOfUsage

This is the `maxNumUsages` "divided by" the `rateOfUsage`. `rateOfUsage` is defined as the number of times a `Nonconsumable Item` is used in one day.

Concrete Example:
Suppose we have a Drink (item1) with daysLeft == 10.
Suppose we have a Notebook (item2) with maxNumUsages == 100.
Suppose the rateOfUsage is 5.0.
item1 will expire in 10 days.
item2 will expire in 20.0 days, since it will be used at a rate of 5.0 times a day.
The method returns -1 (negative one) since item1 expires before item2.

Important Note: item1 can be either `Consumable` or `Nonconsumable`. The same applies for item2. Make sure to handle all of these cases!

Method 2 - generateAndApplyDiscount (15/50 points)

The second method is `generateAndApplyDiscount`. The method signature for it is:
`public int generateAndApplyDiscount(double discountRate)`

Goal: Select a random Item from this Store object's `itemList` and apply a discount on it.

To select a random Item from the `itemList`, you can generate a random index. Use the `Math.random()` method to generate a random number in the range **[0, itemList.size())**, i.e., 0 *inclusive* and **itemList.size()** *exclusive*. Remember that `Math.random()` generates a random number (double) greater than or equal to `0.0` and less than `1.0`. You should use this to generate a number in the required range. Typecast the `double` value to an integer and return the generated random number as an `int` from the method.

Once you have selected an `Item`, **set its price to the discount price**, computed using the `discountRate` parameter. You set the discounted price by multiplying the discount rate and the original price together to produce a new price. Return the index of the Item that was selected from ItemList. See the example below.

Concrete Example:

If the selected `Item` is at index `2` in `ItemList` and has the price `6.4`, and `discountRate` is `0.5` the new, discounted price for the Item is `3.2`. The int `2` is returned.

If the selected `Item` is at index `2` in `ItemList` and has the price `10`, and `discountRate` is `0.3` the new, discounted price for the Item is `3.0`. The int `2` is returned.

Method 3 - getItemsByType (15/50 points)

The third method is `getItemsByType`. The method signature for it is:

```
public Item[] getItemsByType()
```

Goal: Return an Array containing all items in this Store's `itemList` with `Item` objects categorized by (i.e., ordered-by) type in the following order: `Drink`, `Snack`, `Notebook`, `Toiletry`, and other types.

Concrete Examples:

Suppose itemList has: a Notebook, a Drink named "Milk", a Snack, a Drink named "Juice", an object "A" with some other type, and a Toiletry object, **in that order**. The method getItemsByType() will return an Array with the Item objects in the following order: [Drink "Milk", Drink "Juice", Snack object, Notebook object, Toiletry Object, object "A" with some other type]

Important Note: Within the groups of types, the order does not matter. The Array would still be correct if Drink "Milk", and Drink "Juice" were switched.

Hint: Start by using a loop to iterate through itemList. In the loop, get the type of the current Item. Depending on what the type is, you may want to treat that Item differently compared to Item's with different types so that you can group Item objects together by type. **Note that there are several ways to implement this method.**

Part 6: Compile, Run and UnitTest Your Code (10 points)

Just like in previous assignments, **in this part of the assignment, you need to implement your own test cases in the method called `unitTests` in `Assignment6.java`.**

In the starter code, a test case is already implemented for you. You can regard it as an example to implement other cases. Recall, the general approach is to come up with different inputs and manually give the expected output, then call the method with that input and compare the result with expected output.

You are encouraged to create as many test cases as you think to be necessary to cover all the edge cases. The `unitTests` method returns `true` only when all the test cases are passed. Otherwise, it returns `false`. **To get full credit for this section, you must create at least four test cases that cover different situations (including the one we have provided) for the three Store methods - `compareFinalUsageDate()`, `generateAndApplyDiscount()`, and `getItemsByType()`.** In other words, you will need to create at least **three** more tests that test these methods, with at least **one test for each**.

To compare ArrayLists by the equality of contents, you must use the List **equals** method. See the given unit tests for examples.

If a test is not passing, try temporarily printing the result of your method(s) and comparing them to the expected output. Notice that we have already implemented the `toString()` method with the `@Override` annotation for all classes. These `toString()` methods will help you debug when you call `System.out.print` on any Item objects. Notice also that we have defined a method you can use called `printItemArray(Item[] itemArr)` for printing out an Array of Items. Please take a look at the comments under the `TODO` section in the unitTests method for suggestions on how to test the Store methods!

You can compile all the files present in the starter code and run your unit tests from `main()` using the following commands: (Make sure you are in the correct directory, else navigate to the starter code using `cd`)

```
> javac *.java
> java Assignment6
```

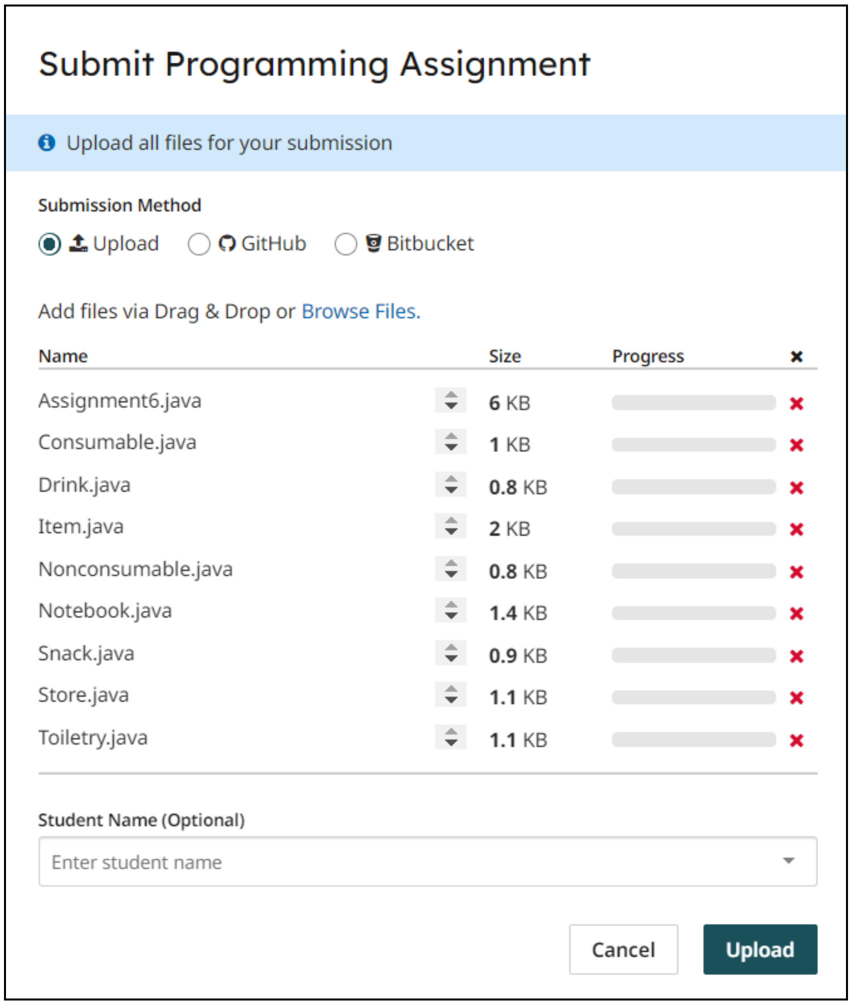
Remember that writing unit tests will help you find bugs in your code and ensure that it is correct for different inputs. So you can have idea of what you may want to test more extensively, here is a reminder of what we test in the Autograder (worth 80 points) and how much each part is worth:

- Consumable and Nonconsumable Tests - 10 points
- Drink and Snack Tests - 10 points
- Notebook and Toiletry Tests - 10 points
- Store Basic Methods (constructor, getItemList, addToItemList) Unit Tests - 5 points
- compareFinalUsageDate Unit Tests - 15 points
- generateAndApplyDiscount Unit Tests - 15 points
- getItemsByType Unit Tests - 15 points

Submission

You're almost there! Please follow the instructions below carefully and use the **exact submission format**. Because we will use scripts to grade, **<u>you may receive a zero</u>** if you do not follow the same submission format.

1. Open Gradescope and login. Then, select this course → Assignment 6.
2. Click the DRAG & DROP section and directly select the required files: `Assignment6.java`, `Consumable.java`, `Drink.java`, `Item.java`, `Nonconsumable.java`, `Notebook.java`, `Snack.java`, `Store.java`, and `Toiletry.java`. Drag & drop is fine. Do not submit a zip, just the nine files in one Gradescope submission. Make sure the names of the files are correct.
3. You can resubmit unlimited times before the due date. Your score will depend on your final (most recent) submission, even if your former submissions have higher scores.
4. Your submission should look like the below screenshot. If you have any questions, feel free to post on Piazza!



Submit Programming Assignment

Upload all files for your submission

Submission Method

[✓ Upload] GitHub Bitbucket

Add files via Drag & Drop or Browse Files.

Name	Size	Progress
Assignment6.java	6 KB	
Consumable.java	1 KB	
Drink.java	0.8 KB	
Item.java	2 KB	
Nonconsumable.java	0.8 KB	
Notebook.java	1.4 KB	
Snack.java	0.9 KB	
Store.java	1.1 KB	
Toiletry.java	1.1 KB	

Student Name (Optional)

Enter student name

Cancel Upload