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

Assignment 6

Inheritance and Polymorphism

Due: Wednesday, November 9, 11:59 PM

Hi again! 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 POJOS (Plain Old Java Object) classes with getters/setters.
- Practice inheritance by defining multiple superclasses and subclasses.
- Apply knowledge of polymorphism in several methods.

NOTE: This programming assignment must be done individually. Paired programming is NOT allowed for this assignment.

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. Make sure there is no problem with your Java software development environment. If there is any, then review Assignment 1, or come to the office/lab hours before you start Assignment 6.
2. First, navigate to the `cse8b` folder that you have created in Assignment 1 and create a new folder titled `assignment6`
3. Download the starter code. You can download the starter code from Piazza → Resources → Homework → `assignment6.zip`. The starter code should contain nine files: `Assignment6.java`, `Cereal.java`, `Item.java`, `Nonwearable.java`, `Pants.java`, `Phone.java`, `Shirt.java`, `Store.java`, and `Wearable.java`. Place the starter code within the assignment 6 folder that you have just created
4. Compile the starter code within the assignment 6 folder. You can compile all files using the single command `javac *.java` and you should get a series of compiler errors since you have not implemented the classes yet.The objective of this assignment is to get the classes working by implementing the class methods and testing them.
5. You will be turning in all of the original files included in **`Assignment6.zip`**.

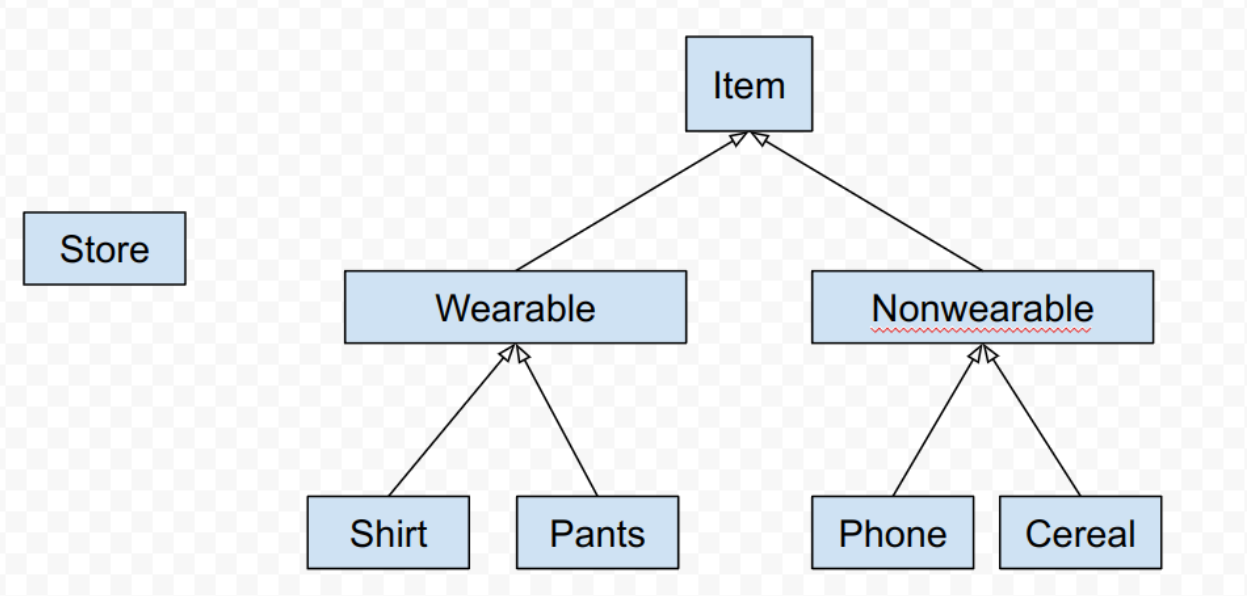
Part 1: Overview

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 pretty weird, and you can't really tell what type of store it is… It strangely only sells four specific categories of Items: Pants and Shirts (Wearables), and Phones and Cereals (Nonwearables).

Logistics:

For this programming 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 arrow 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`s and `double`s 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 called `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 `"Wearable"`, `"Nonwearable"`, or `"Untyped Item"` (described below)
4. **`private String type`**
   → must be string literal `"Pants"`, `"Shirt"`, `"Phone"`, `"Cereal"`, or `"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 `Item`s: `Wearable` and `Nonwearable`. The `type` member describes the more specific type of an `Item`. There are four “type” classes: `Pants`,

`Shirt`, `Phone`, and `Cereal`. As seen in the UML diagram above, `Pants` and `Shirt` "is-a" `Wearable`, and `Phone` and `Cereal` "is-a" `Nonwearable`.

First, in `Item.java`, complete the getters and setter 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. q

Part 3: `Wearable.java` and `Nonwearable.java` (10 points)

`Wearable` and `Nonwearable` are two subclasses of `Item`. Notice the `Wearable` and `Nonwearable` 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.**

Wearable

The `Wearable` class has two fields:

1. **`private double warmth`**
   → an integer denoting the warmth of this wearable item. The greater this field is, the warmer the `Wearable` object is.
2. **`private String fabric`**
   → a String denoting the type of fabric this wearable item is made of

Implement the following constructor and methods:

1. **`public Wearable(String name, double price, String type, int warmth, int fabric)`**
   - 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 `"Wearable"`. Then, set the `warmth` and `fabric` members using the remaining constructor parameters. Remember, you must use the `this` keyword to access member variables hidden by local variables.
2. **`public int getWarmth()`**
   - Simple getter method that returns the `warmth` instance variable.
3. **`public String getFabric()`**
   - Simple getter method that returns the `fabric` instance variable.
4. **`public boolean equals(Item item)`**
   - This method overrides the `equals()` method in `Item`. This method checks whether the current `Wearable` 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`, `warmth`, AND `fabric`. 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 10, slide 33). You will be using the override annotation for all overridden `equals()` methods throughout this assignment.)

Nonwearable

The `Nonwearable` class has one field:

1. **`private int maxNumUsages`**

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

Implement the following constructor and methods:

1. **`public Nonwearable(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 `"Nonwearable"`. 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 `Nonwearable` 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: Pants.java and Shirt.java (10 points)

`Pants` and `Shirt` are two subclasses of `Wearable` (since they are things you can wear).

Complete all remaining constructors and methods in these classes.

Pants

The Pants class has one field:

1. **`private int waistSize`**

   → an integer denoting the waist size of a pair of pants

Implement the following methods:

1. **`public Pants(String name, double price, int warmth, int fabric, double waistSize)`**
   - This constructor must set the `name`, `price`, `type`, `warmth`, and `fabric` in its superclass (HINT: use `super` to call the superclass constructor!) Then, set the `waistSize` member using the remaining constructor parameter. Remember, you must use the `this` keyword to access member variables hidden by local variables.
2. **`public double getWaistSize()`**
   - Simple getter method that returns the `waistSize` instance variable.
3. **`public boolean equals(Item item)`**
   - This method overrides the `equals()` method in `Wearable`. This method checks whether the current `Pants` 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`, `warmth`, `fabric`, and `waistSize`. 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.

Shirt

The Shirt class has one field:

1. **`private String size`**

→ a String denoting the size of the shirt. It can **only** take on values of `"Small"`, `"Medium"`, or `"Large"`. We have provided constants in the starter code so that you can remain consistent with the autograder.

Implement the following methods:

1. **`public Shirt(String name, double price, int warmth, String fabric, String size)`**
    - This constructor must set the `name`, `price`, `type`, `warmth`, and `fabric` in its superclasses (HINT: use `super` to call the superclass constructor!) from the constructor parameters. Then, set the `size` member using the remaining constructor parameter. Remember, you must use the `this` keyword to access member variables hidden by local variables.
2. **`public String getSize()`**
    - Simple getter method that returns the `size` instance variable.
3. **`public boolean equals(Item item)`**
    - This method overrides the `equals()` method in `Wearable`. This method checks whether the current `Shirt` 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`, `warmth`, `fabric`, AND `size`. 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: Phone.java and Cereal.java (10 points)

`Phone` and `Cereal` are two subclasses of `Nonwearable` (since they are things you cannot wear). **Complete all remaining constructors and methods in these classes.**

Phone

The `Phone` class has two fields:

1. **`private double cpuSpeed`**

    → a double denoting the speed of the phone in gigahertz.
2. **`private boolean includesCharger`**

    → a boolean denoting whether the phone comes with a charger or not (e.g., iPhones).

Implement the following constructor and methods:

1. **`public Phone(String name, double price, int maxNumUsages, double cpuSpeed, boolean includesCharger)`**
    - 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 `cpuSpeed` and `includesCharger` member using the remaining constructor parameters. Remember, you must use the `this` keyword to access member variables hidden by local variables.
2. **`public double getCpuSpeed()`**
    - Simple getter method that returns the `cpuSpeed` instance variable.
3. **`public double getIncludesCharger()`**
    - Simple getter method that returns the `includesCharger` instance variable.
4. **`public boolean equals(Item item)`**
    - This method overrides the `equals()` method in `Nonwearable`. This method checks whether the current `Phone` 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`. `cpuSpeed`, and `includesCharger`. 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.

Cereal

The `Cereal` class has one field:

1. **`private int calories`**
   → an int denoting the amount of calories that is contained in the cereal

Implement the following constructor and methods:

1. **`public Cereal(String name, double price, int maxNumUsages, int calories)`**

- 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 `calories` member using the remaining constructor parameter.

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

5. **`public int getCalories()`**
    - Simple getter method that returns the `calories` instance variable.
6. **`public boolean equals(Item item)`**
    - This method overrides the `equals()` method in `Nonwearable`. This method checks whether the current `Cereal` 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`, AND `calories`. 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 - compareBangForBuck (15/50 points)

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

Goal: Given two `Item` objects, item1 and item2, determine which item is more the one that is more “worth it.”

- Return `1` if `item1` is more “worth it” than `item2`
- Return `0` if `item1` and `item2` have the same “worth”
- Return `-1` if `item2` has more “worth” than `item1`

Notice how every item in the store has a price. We calculate an item’s worth as follows:

1. If the `Item` is of type `Phone`, then its worth is calculated as its `cpuSpeed` divided by its `price`
2. If an `Item` is of type `Cereal`, then its worth is calculated as its `calories` divided by its `price`
3. If an `Item` is of type `Wearable` (`Shirt` or `Pants`), then its worth is calculated as its `warmth` divided by its `price`.

An item1 is more worth it than item2 if item1 has a higher “worth” than item2.

Concrete Example:

Suppose we have a `Phone` (item1) with `cpuSpeed == 5.0` and `price == 215.63`
Suppose we have a `Phone` (item2) with `cpuSpeed == 3.57` and `price == 180.65`
The method returns `1` (positive one) since the worth of item1 ≈0.023 and the worth of item2 is ≈0.020

You should not be rounding the results you get when computing the worth of your items but if you do, make sure it is at least accurate to the fifth decimal pass in order to pass the autograder.

Important Note: item1 can be either `Wearable` or `Nonwearable`. 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 - containsMatchingPantsAndShirt (15/50 points)

The third method is `containsMatchingPantsAndShirt`. The method signature for it is:
`public boolean containsMatchingPantsAndShirt()`

Goal: Return a boolean denoting if the store contains **at least one** pair of "matching" `Pants` and `Shirt`. A pair of `Pants` and `Shirt` is considered to be "matching" according to the conditions below:

1. The `waistSize` of the `Pants` is greater than or equal to `21` but less than or equal to `30` and `size` of the `Shirt` is `"Small"`
2. The `waistSize` of the `Pants` is greater than or equal to `31` but less than or equal to `40` and `size` of the `Shirt` is `"Medium"`
3. The `waistSize` of the `Pants` is greater than or equal to `41` but less than or equal to `50` and `size` of the `Shirt` is `"Large"`

Concrete Examples:
If `itemList` contains 1 pair of `Pants`, 1 `Phone`, and 1 `Cereal` for a total of 3 items, then return `false` because there is no potential `Shirt` to be paired with the `Pants`.

If `itemList` contains 2 pairs of `Pants` (one of `waistSize` `25` and one of `waistSize` `36`), 1 `Phone`, and 1 `Cereal` and 1 `Shirt` of size `"Large"` for a total of 5 items, then return `false` because there is no matching `Pants` of `waistSize` [41, 50] to be paired with the `"Large"` shirt.

If `itemList` contains 2 pairs of `Pants` (one of `waistSize` `25` and one of `waistSize` `50`), 1 `Phone`, and 1 `Cereal` and 1 `shirt` of size `"Large"` for a total of 5 items, then return `true` because the `Pants` of `waistSize` 50 can be paired with the `"Large"` shirt.

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 -** `compareBangForBuckDate()`**,** `generateAndApplyDiscount()`**, and** `containsMatchingPantsAndShirt()`**.** 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:

- Wearable and Nonwearable Tests - 10 points
- Pants and Shirt Tests - 10 points
- Phone and Cereal Tests - 10 points
- Store Basic Methods (constructor, getItemList, addToItemList) Unit Tests - 5 points
- compareBangForBuckDate Unit Tests - 15 points
- generateAndApplyDiscount Unit Tests - 15 points
- containsMatchingShirtAndPants 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, **you may receive a zero** 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`, `Cereal.java`, `Item.java`, `Nonwearable.java`, `Pants.java`, `Phone.java`, `Shirt.java`, `Store.java`, and `Wearable.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	
Cereal.java	**1** KB	
Item.java	**2.3** KB	
Nonwearable.java	**1** KB	
Pants.java	**1** KB	
Phone.java	**1.4** KB	
Shirt.java	**1** KB	
Store.java	**5** KB	
Wearable.java	**1.3** KB	

SUBMITTING FOR

Darren Yeung

Upload Cancel