CSE 11: Introduction to Programming and Computational Problem Solving: Accelerated Pace
Assignment 6

Inheritance and Polymorphism

Due: Wednesday, May 22, 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.

NOTE: This programming assignment must be done individually.

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 and the autograder fails to execute properly, you will receive an automatic zero on the autograder portion of the assignment. Make sure that you do not change the method headers or import other packages unless otherwise specified.

Coding Style (10 points)

For this programming assignment, we will be enforcing the CSE 11 Coding Style Guidelines. These guidelines can also be found on Canvas and the class website. Please ensure to have COMPLETE file headers, class headers, and method headers, use descriptive variable names and proper indentation, and 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 cse11 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, Item.java, Drink.java, Snack.java, Refreshing.java, Energizing.java, Sweet.java, Savory.java, and VendingMachine.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, Drink.java, Snack.java, Refreshing.java, Energizing.java, Sweet.java, Savory.java, and VendingMachine.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:37: error: missing return statement
 } ^
./Item.java:45: error: missing return statement
 } ^
6 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 vending machine has opened up outside the UCSD CSE Building! It sells four specific categories of Items: Refreshing and Energizing (Drinks), and Sweet and Savory (Snacks).
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 VendingMachine 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 fields or method headers 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 11 Coding 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 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 VendingMachine class having the majority of the functionality of this program.

Part 2: Implement Item Class (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 (including those overridden by subclasses) of all the subclasses 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.

```
Item
- name: String
- price: double
- calories: int
- HIGH_LEVEL_TYPE: String
- TYPE: String

+ Item()
+ Item(String name, double price, int calories)
+ getName(): String
+ getPrice(): String
+ getCalories(): int
+ getHighLevelType(): String
+ getType(): String
+ setPrice(double Price): void
+ equals(Object object): boolean
+ toString()
```

The Item class has the following data fields:

- private String name
  - The name of the item
- private double price
  - The price of the item
- private int calories
The number of calories the item has

- private static final String HIGH_LEVEL_TYPE
  - Must be string literal "Untyped High Level Item"

- private static final String TYPE
  - Must be string literal "Untyped Item"

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.

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:

- public String getName()
  - Simple getter method that returns the name instance variable.

- public double getPrice()
  - Simple getter method that returns the price instance variable.

- public int getCalories()
  - Simple getter method that returns the calories instance variable.

- public String getHighLevelType()
  - Simple getter method that returns the HIGH_LEVEL_TYPE static variable.

- public String getType()
  - Simple getter method that returns the TYPE static variable.

- public void setPrice(double price)
  - Simple setter method that sets the price instance variable.

You also need to implement the following constructor:

- public Item(String name, double price, int calories)
  - 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:

- public boolean equals(Object object)
  - 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, and calories. 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: Implement Drink and Snack Classes (10 points)**

Drink and Snack are two subclasses of Item. Notice the Drink and Snack 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 method headers or the fields.**

There are two highLevelType classes of Items: Drink and Snack. The type member describes the more specific type of an Item. There are four “type” classes: Refreshing, Energizing, Sweet, and Savory. As seen in the UML diagram above, Refreshing and Energizing “is-a” Drink, and Sweet and Savory “is-a” Snack.
The Drink class has four fields:

- **private double volume**
  - The volume of the drink in milliliters.

- **private String flavor**
  - The flavor or essence of the drink.

- **private static final String HIGH_LEVEL_TYPE**
  - Must be string literal "Drink"

- **private static final String TYPE**
  - Must be string literal "Untyped Drink"

Implement the following constructor and methods:

- **public Drink(String name, double price, int calories, double volume, String flavor)**
  - This constructor must set the name, price, and calories in its superclass (HINT: use `super` to call the superclass constructor!) from the constructor parameters. Then, set the volume and flavor members using the remaining constructor parameters. Remember, you must use the `this` keyword to access member variables hidden by local variables.

- **public double getVolume()**
  - Simple getter method that returns the volume instance variable.

- **public String getFlavor()**
  - Simple getter method that returns the flavor instance variable.

- **public String getHighLevelType()**
- Simple getter method that returns the `HIGH_LEVEL_TYPE` static variable.
- This method overrides the `getHighLevelType()` method in Item.

  ```java
  public String getType()
  ```

- Simple getter method that returns the `TYPE` static variable.
- This method overrides the `getType()` method in Item.

  ```java
  public boolean equals(Object object)
  ```

  This method overrides the `equals()` method in Item. This method checks whether the current `Drink` object is considered equal to the input `Object`. This method must return `true` only when the current object has the same `name`, `price`, `calories`, `volume`, and `flavor`. 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. You will be using the `override` annotation for all overridden `getHighLevelType()`, `getType()`, and `equals()` methods throughout this assignment.

### Snack.java

<table>
<thead>
<tr>
<th>Snack</th>
</tr>
</thead>
<tbody>
<tr>
<td>-servingSize: int</td>
</tr>
<tr>
<td>-texture: String</td>
</tr>
<tr>
<td>-HIGH_LEVEL_TYPE: String</td>
</tr>
<tr>
<td>-TYPE: String</td>
</tr>
</tbody>
</table>

- `+Snack(String name, double price, int calories, int servingSize, String texture)`
- `+getServingSize(): int`
- `+getTexture(): String`
- `+getHighLevelType(): String`
- `+getType(): String`
- `+equals(Object object): boolean`
- `+toString()`

The `Snack` class has four fields:

- `private int servingSize`
  - The amount of times a `Snack` item can be consumed in a single sitting. (e.g. if the `servingSize` is 2, then the `Snack` item can be consumed in two sittings)

- `private String texture`
○ The texture of the snack upon consumption.

• private static final String HIGH_LEVEL_TYPE
  ○ Must be string literal "Snack"

• private static final String TYPE
  ○ Must be string literal "Untyped Snack"

Implement the following constructor and methods:

• public Snack(String name, double price, int calories, int servingSize, String texture)
  ○ This constructor must set the name, price, and calories in its superclass (HINT: use super to call the superclass constructor!) from the constructor parameters. Then, set the servingSize and texture members using the remaining constructor parameters. Remember, you must use the this keyword to access member variables hidden by local variables.

• public int getServingSize()
  ○ Simple getter method that returns the servingSize instance variable.

• public String getTexture()
  ○ Simple getter method that returns the texture instance variable.

• public String getHighLevelType()
  ○ Simple getter method that returns the HIGH_LEVEL_TYPE static variable.
  ○ This method overrides the getHighLevelType() method in Item.

• public String getType()
  ○ Simple getter method that returns the TYPE static variable.
  ○ This method overrides the getType() method in Item.

• public boolean equals(Object object)
  ○ This method overrides the equals() method in Item. This method checks whether the current Snack object is considered equal to the input Object. This method must return true only when the current object has the same name, price, calories, servingSize, and texture. 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: Implement Refreshing and Energizing classes (10 points)

Refreshing.java

The Refreshing class has two fields:

- `private int coolnessLevel`
  - The coolness factor of the refreshing drink.
- `private static final String TYPE`
  - Must be string literal "Refreshing"

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.

Implement the following constructor and methods:

- `public Refreshing(String name, double price, int calories, double volume, String flavor, int coolnessLevel)`
  - This constructor must set the `name`, `price`, `calories`, and `volume`, and `flavor` in its superclass (HINT: use `super` to call the superclass constructor!) from the constructor parameters. Then, set the `coolnessLevel` members using the remaining constructor parameter. Remember, you must use the `this` keyword to access member variables hidden by local variables.

- `public int getCoolnessLevel()`
  - Simple getter method that returns the `coolnessLevel` instance variable.
- public String getType()
  - Simple getter method that returns the TYPE static variable.
  - This method overrides the getType() method in Drink.
- public boolean equals(Object object)
  - This method overrides the equals() method in Drink. This method checks whether the current Refreshing object is considered equal to the input Object. This method must return true only when the current Refreshing object has the same name, price, calories, volume, flavor, and coolnessLevel. 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.

Energizing.java

The Energizing class has two fields:
- private int boostLevel
  - The energy boost provided by the energizing drink.
- private static final String TYPE
  - Must be string literal "Energizing"

Implement the following constructor and methods:
- public Energizing(String name, double price, int calories, double volume, String flavor, int boostLevel)
This constructor must set the name, price, calories, volume, and flavor in its superclass (HINT: use super to call the superclass constructor!) from the constructor parameters. Then, set the boostLevel members using the remaining constructor parameter. Remember, you must use the this keyword to access member variables hidden by local variables.

- public int getBoostLevel()
  - Simple getter method that returns the boostLevel instance variable.

- public String getType()
  - Simple getter method that returns the TYPE static variable.
  - This method overrides the getType() method in Drink.

- public boolean equals(Object object)
  - This method overrides the equals() method in Drink. This method checks whether the current Refreshing object is considered equal to the input Object. This method must return true only when the current object has the same name, price, calories, volume, flavor, and boostLevel. 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 4b: Implement Sweet and Savory Classes (10 points)

#### Sweet.java

<table>
<thead>
<tr>
<th>Sweet</th>
</tr>
</thead>
<tbody>
<tr>
<td>-sweetnessLevel: int</td>
</tr>
<tr>
<td>-TYPE: String</td>
</tr>
</tbody>
</table>

- Sweet()
- Sweet(String name, double price, int calories, int servingSize, String texture, int sweetnessLevel)
- getSweetnessLevel(): int
- getType(): String
- equals(Object object): boolean
- toString()

The **Sweet** class has two fields:
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.

Implement the following constructor and methods:

- **public Sweet(String name, double price, int calories, int servingSize, String texture, int sweetnessLevel)**
  - This constructor must set the `name`, `price`, `calories`, `servingSize`, and `texture` in its superclass (HINT: use `super` to call the superclass constructor!) from the constructor parameters. Then, set the `sweetnessLevel` members using the remaining constructor parameter. Remember, you must use the `this` keyword to access member variables hidden by local variables.

- **public int getSweetnessLevel()**
  - Simple getter method that returns the `sweetnessLevel` instance variable.

- **public String getType()**
  - Simple getter method that returns the `TYPE` static variable.
  - This method overrides the `getType()` method in `Snack`.

- **public boolean equals(Object object)**
  - This method overrides the `equals()` method in `Snack`. This method checks whether the current `Refreshing` object is considered equal to the input `Object`. This method must return `true` only when the current object has the same name, `price`, `calories`, `servingSize`, `texture`, and `sweetnessLevel`. 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.
The `Savory` class has two fields:

- **private int savorinessLevel**
  - The salt level in the snack.
- **private static final String TYPE**
  - Must be string literal "Savory"

Implement the following constructor and methods:

- **public Savory(String name, double price, int calories, int servingSize, String texture, int savorinessLevel)**
  - This constructor must set the `name`, `price`, `calories`, `servingSize`, and `texture` in its superclass (HINT: use `super` to call the superclass constructor!) from the constructor parameters. Then, set the `savorinessLevel` members using the remaining constructor parameter. Remember, you must use the `this` keyword to access member variables hidden by local variables.

- **public int getSavorinessLevel()**
  - Simple getter method that returns the `savorinessLevel` instance variable.

- **public String getType()**
  - Simple getter method that returns the `TYPE` static variable.
  - This method overrides the `getType()` method in `Snack`.

- **public boolean equals(Object object)**
  - This method overrides the `equals()` method in `Snack`. This method checks whether the current `Refreshing` object is considered equal to the input `Object`. This method must return `true` only when the current object has the same `name`,
price, calories, servingSize, texture, and savorinessLevel. 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 5: Implement VendingMachine Class (50 points)

Finally, the cool part! You are now responsible for keeping track of inventory (not selling items!). You will be implementing a VendingMachine class with various unique methods to help you keep track of all the Items in the vending machine.

VendingMachine Basic Methods (5/50 points)

The VendingMachine class has one field:

- private ArrayList<Item> itemList

  - An ArrayList containing Item objects. Please refer to the ArrayList documentation 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 VendingMachine constructor.

- public VendingMachine()

  - The no-arg constructor must initialize itemList to an empty ArrayList of Item elements.
Next, implement the methods below along with a method called addToItemList overloaded with **two** different implementations.

- **public void addToItemList(Item item)**
  - Adds item to itemList (a single Item)
- **public void addToItemList(Item[] items)**
  - Adds each Item in items to itemList (can be multiple Item objects)
- **public boolean hasItem(String itemName)**
  - Checks if Item is available in the vending machine based on its name
- **public Item getItem(String itemName)**
  - Retrieves Item from the vending machine based on its name. (return the Item, and **remove** Item from ItemList)
  - If there are two Items with the same name, return and **remove** the first Item found inItemList.
  - If the item does not exist, return null

Now, we will start implementing the core part of the VendingMachine class - **three** methods with unique functionality. This is where you will be applying your knowledge of **polymorphism**.

**Method 1 - compareSize (15/50 points)**

The first method is compareSize. The method header for it is:

```
public static int compareSize(Item item1, Item item2)
```

The first thing we need to notice is that Refreshing objects and Energizing objects each have a volume field because they have an “is-a” relationship with Drink, and that Sweet objects and Savory objects each have a servingSize field because they have an “is-a” relationship with Snack.

**Goal:** Given two Item objects, item1 and item2, compare the sizes of the two items.

- Return `-1` (negative one) if item1 is smaller than item2
- Return `0` if item1 is the same exact size as item2
- Return 1 if item1 is larger than item2

To make this comparison, we need to be able to compute the number of ounces each item has. This changes depending on the HIGH_LEVEL_TYPE of the Item.

- If an Item “is-a” Drink (which you can check by looking at its HIGH_LEVEL_TYPE), the number of ounces the Item has is its volume field.
- If an Item “is-a” Snack, the number of ounces the Item has follows the formula:
  \[ \text{servingSize} \times 12.0 \]

This is the servingSize “multiplied by” 12.0.

**Concrete Example:**

Suppose item1 is Refreshing with volume = 24.0.
Suppose item2 is Sweet with servingSize = 1.

item1 has 24 ounces.
item2 has 12 ounces (1 * 12.0).

The method returns 1 (positive one) since item1 is larger than item2.

**Important Note:** item1 can be either Drink or Snack. The same applies for item2. Make sure to handle all of these cases!

**Method 2 - applyPriceSurge (15/50 points)**

The second method is applyPriceSurge. The method header for it is:

```java
public int applyPriceSurge(double increaseRate)
```

**Goal:** Select a random Item from itemList and apply a price increase 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. Cast 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 new increased price**, computed using the `increaseRate` parameter. You set the increased price by multiplying the `increaseRate` and the original price together to produce a new price. If the `increaseRate` value is less than 1, then do NOT modify the original 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 `increaseRate` is 1.5, then the new price for the Item is 9.6. The int 2 is returned.

If the selected Item is at index 3 in `ItemList` and has the price 10 and `increaseRate` is 0.3, then the price for the Item remains 10.0. The int 3 is returned.

**Method 3 - `getItemsByType` (15/50 points)**

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

```java
public Item[] getItemsByType(String type)
```

**Goal:** Return an array of Item objects from `ItemList` where all the Items’ type matches the parameter type.

**Concrete Examples:**
Suppose `ItemList` has: a Drink object of type "Untyped Drink", a Refreshing object of type "Refreshing" with name "Tea", a Snack object of type "Untyped Snack", a Refreshing object of type "Refreshing" with name "Matcha", a Sweet object of type "Sweet" with name "Chocolate", and an Item object of type "Untyped Item" with name "A", **in that order**.

The method `getItemsByType("Refreshing")` will return an array with the Item objects in the following order: [Refreshing "Tea", Refreshing "Matcha", null, null, null, null]

Consider the two following examples.
The method `getItemsByType("Untyped Drink")` will return an array with the `Item` objects in the following order: `[Drink object, null, null, null, null, null, null]`

The method `getItemsByType("Drink")` will return an array with the `Item` objects in the following order: `[null, null, null, null, null, null, null]`

Observe that while `getItemsByType("Untyped Drink")` returns `[Drink object]` and `getItemsByType("Drink")` returns `[]`. This difference is due to us checking the TYPE rather than the HIGH_LEVEL_TYPE of the `Item`.

**Note:** The size of the returned array must be equal to the size of `itemList`.

**Hint:** Start by using a loop to iterate through `itemList`. In the loop, think about how you could get the TYPE of the current `Item` to compare against the parameter type. Depending on the `Item`'s type, you may want to store or ignore the item. **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 - `compareSize()`, `applyPriceSurge()`, 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 VendingMachine 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)*

```bash
> 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:

- Drink and Snack Tests - 10 points
- Refreshing and Energizing Tests - 10 points
- Sweet and Savory Tests - 10 points
- VendingMachine Basic Methods (constructor, `addToList`, `hasItem`, `getItem`) Tests - 5 points
- `compareSize` Tests - 15 points
- `applyPriceSurge` Tests - 15 points
- `getItemByType` 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 → PA6.
2. Click the DRAG & DROP section and directly select the required files:
   Assignment6.java, Item.java, Drink.java, Snack.java, Refreshing.java, Energizing.java, Sweet.java, Savory.java, and VendingMachine.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

Add files via Drag & Drop or Browse Files.

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Progress</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment6.java</td>
<td>3.8 KB</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Drink.java</td>
<td>1.7 KB</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Energizing.java</td>
<td>1.3 KB</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Item.java</td>
<td>2.4 KB</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Refreshing.java</td>
<td>1.3 KB</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Savory.java</td>
<td>1.3 KB</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Snack.java</td>
<td>1.7 KB</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Sweet.java</td>
<td>1.3 KB</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>VendingMachine.java</td>
<td>2.9 KB</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Submitting For
Amanda Quach