Homework 04 – The Dessert Conundrum

Authors: Nathan, Chloe, Jack, Melanie, Chelsea
Topics: ArrayList using generics, Asymptotics, Searching, and Sorting

Problem Description

Please make sure to read all parts of this document carefully.
Bob's favorite food are desserts (as it is for most of you, I assume). But unfortunately, Bob is also a very picky eater and needs help figuring out which dessert to eat today. Help Bob pick the best dessert to complete his meal!

Solution Description

You will need to complete and turn in 5 classes: Dessert.java, Cake.java, IceCream.java, Store.java, Bob.java. All variables should be inaccessible from other classes and must require an instance to be accessed through, unless specified otherwise. All methods should be accessible from everywhere and must require an instance to be accessed through, unless specified otherwise. *Make sure to reuse code when applicable.*

Dessert.java

This class will be the superclass for the dessert classes below. Implement this class to define the basic behaviors of dessert objects. This class should implement the Comparable interface using generics. This class should never be instantiated.

Variables:

- `String flavor` – the flavor of the cake
- `double sweetness` – the sweetness of the cake

Constructor(s):

- A constructor that takes in `flavor` and `sweetness` of the dessert (in this order).
- A default constructor that sets `flavor` to vanilla and `sweetness` to 25.0.

Methods:

- `toString`
  - This method should properly override Object's `toString` method.
  - It should return

    `"This is a {flavor} dessert with a sweetness of {sweetness}."` (without the curly braces)

  - Round all floating-point values to 2 decimal places
- `equals`
  - This method should properly override Object's `equals` method.

    - If two desserts have the same `flavor` and `sweetness` then they are equal.
- `compareTo`
    - This method will implement the `compareTo` method from the Comparable interface.
    - A Dessert is greater than the other if it has greater `sweetness` than the other.
    - If the `sweetness` of both desserts are equal, then the Dessert with the lexicographically greater `flavor` is greater.
- If necessary, add getters and setters for the variables in the class.

Cake.java

This class will describe a certain kind of dessert that Bob can pick: cakes. This class extends the Dessert class.

Variables:

- `String frosting` – what the frosting of the cake is

Constructor(s):

- A constructor that takes in `flavor`, `sweetness`, and `frosting` of the cake (in this order).
- A constructor that takes in the `flavor` and sets the `sweetness` to 45.0 and `frosting` to vanilla.

Methods:

- `toString`
    - This method should properly override Dessert's `toString` method.
    - It should return

      `"This is a {flavor} cake with a {frosting} frosting and has a sweetness of {sweetness}."`

    - Round all floating-point values to 2 decimal places
- `equals`
    - This method should properly override Dessert's `equals` method.
    - If two cakes have the same `flavor`, `sweetness`, and `frosting` then they are equal.
- If necessary, add getters and setters for the variables in the class.

IceCream.java

This class will describe a certain kind of dessert that Bob can pick: ice cream. Write this class so Bob can have ice cream. This class will have to extend the Dessert class.

Variables:

- `int scoops` – the number of scoops of ice cream you get
- `boolean cone` – represents if the ice cream has a cone

Constructor(s):

- A constructor that takes in `flavor, sweetness, scoops,` and `cone` of the ice cream (in this order).
- A constructor that takes in `scoops` and `cone` of the ice cream and sets `flavor` to vanilla and `sweetness` to 45.0.
- A default constructor that assigns `flavor` to vanilla, `sweetness` to 45.0, `scoops` to 1, and `cone` to false.

Methods:

- `toString`
  - This method should properly override Dessert's `toString` method.
  - It should return

    `"This is a {flavor} ice cream with {scoops} scoops and has/does not have a cone."`

  - The "has/does not have" depends on the `cone` variable.
- `equals`
  - This method should properly override Dessert's `equals` method.
  - If two ice creams have the same `flavor, sweetness, scoops,` and `cone` then they are equal.
- If necessary, add getters and setters for the variables in the class.

Store.java

This will be the class that stores the data for the dessert store. We are assuming that a store can sell all kinds of dessert.

Variables:

- `String name` – name of the store.
- `ArrayList desserts` – an ArrayList of type Dessert

Constructor(s):

- A constructor that only takes in `name` and creates an empty `ArrayList` of type Dessert.

Methods:

- `addDessert`
  - This method should take in a valid Dessert object and add it to the back of the ArrayList.
  - This method should not return anything.
  - This method should run in O(1) time.
- `removeDessert`
  - This method should take in a valid Dessert object and remove the first instance of it from the ArrayList.
  - Return the object that is being removed or null if the dessert is not found.

    - Use `equals` when identifying if the object in the ArrayList is the same as the passed in Dessert object.
    - This method should run in O(n) time.
- `findDessert`
    - This method should take in a valid Dessert object and find/return the dessert that has the same sweetness and flavor.
    - If a dessert with the same sweetness and flavor is not found in the Store, return null.
    - This method should run in O(log n) time.
    - Assume the ArrayList has **unique** dessert items and is sorted in ascending order based on sweetness and flavor.
- `sortStore`
    - This method should sort the ArrayList in **ascending** order based on sweetness and flavor.
    - This method should run in $O(n^2)$ time worst and best case.
- `checkStore`
    - This method should take in a valid Dessert object and return the number of desserts in the store that is greater or equal to the Dessert passed in.
    - This method should run in O(n) time.
- If necessary, add getters and setters for the variables in the class.

Bob.java

This will be the class for Bob where we will define Bob's behaviors. All methods should be static. Notice that you have already implemented most of the code necessary for Bob's behavior.

Methods:

- `compareStores`
    - This method should take in two valid Store objects.
    - The sweetness and flavors of each individual dessert object in Store 1 must all be present in Store 2. Return true if this is the case. False if otherwise.
    - Assume Store 2's dessert inventory is sorted in **ascending** sweetness and flavor order.
    - This method should run in O(n log n) time.
- `shop`
    - This method should take in a valid Store object and a valid Dessert object
    - The Store's desserts should become sorted in ascending order based on sweetness and flavor to help Bob find his dessert.
    - This method should return true or false depending on whether Bob was able to find his dessert.
    - This method should run in $O(n^2)$ time.

Checkstyle

You must run checkstyle on your submission (To learn more about Checkstyle, check out cs1331-style-guide.pdf under CheckStyle Resources in the Modules section of Canvas.) **The Checkstyle cap for this assignment is 30 points.** This means there is a maximum point deduction of 30. If you don't have Checkstyle yet, download it from Canvas -> Modules -> CheckStyle Resources -> checkstyle-8.28.jar. Place it in the same folder as the files you want to run Checkstyle on. Run checkstyle on your code like so:

```
$ java -jar checkstyle-8.28.jar yourFileName.java
Starting audit...
Audit done.
```

The message above means there were no Checkstyle errors. If you had any errors, they would show up above this message, and the number at the end would be the points we would take off (limited by the checkstyle cap mentioned in the Rubric section). In future homeworks we will be increasing this cap, so get into the habit of fixing these style errors early!

Additionally, you must Javadoc your code.

Run the following to only check your Javadocs:

```
$ java -jar checkstyle-8.28.jar -j yourFileName.java
```

Run the following to check both Javadocs and Checkstyle:

```
$ java -jar checkstyle-8.28.jar -a yourFileName.java
```

For additional help with Checkstyle see the CS 1331 Style Guide.

Turn-In Procedure

Submission

To submit, upload the files listed below to the corresponding assignment on Gradescope:

- `Dessert.java`
- `Cake.java`
- `IceCream.java`
- `Store.java`
- `Bob.java`

Make sure you see the message stating the assignment was submitted successfully. From this point, Gradescope will run a basic autograder on your submission as discussed in the next section. **Any autograder test are provided as a courtesy to help "sanity check" your work and you may not see all the test cases used to grade your work.** You are responsible for thoroughly testing your submission on your own to ensure you have fulfilled the requirements of this assignment. If you have questions about the requirements given, reach out to a TA or Professor via Piazza for clarification.

You can submit as many times as you want before the deadline, so feel free to resubmit as you make substantial progress on the homework. We will only grade your latest submission. **Be sure to submit**

every file each time you resubmit.

Gradescope Autograder

If an autograder is enabled for this assignment, you may be able to see the results of a few basic test cases on your code. Typically, tests will correspond to a rubric item, and the score returned represents the performance of your code on those rubric items only. If you fail a test, you can look at the output to determine what went wrong and resubmit once you have fixed the issue.

The Gradescope tests serve two main purposes:

- Prevent upload mistakes (e.g. non-compiling code)
- Provide basic formatting and usage validation

In other words, the test cases on Gradescope are by no means comprehensive. Be sure to thoroughly test your code by considering edge cases and writing your own test files. You also should avoid using Gradescope to compile, run, or Checkstyle your code; you can do that locally on your machine.

Other portions of your assignment can also be graded by a TA once the submission deadline has passed, so the output on Gradescope may not necessarily reflect your grade for the assignment.

Allowed Imports

- `java.util.ArrayList`

Feature Restrictions

There are a few features and methods in Java that overly simplify the concepts we are trying to teach or break our auto grader. For that reason, do not use any of the following in your final submission:

- `var` (the reserved keyword)
- `System.exit`
- `System.arraycopy`

Collaboration

Only discussion of the Homework (HW) at a conceptual high level is allowed. You can discuss course concepts and HW assignments broadly, that is, at a conceptual level to increase your understanding. If you find yourself dropping to a level where specific Java code is being discussed, that is going too far. Those discussions should be reserved for the instructor and TAs. To be clear, you should never exchange code related to an assignment with anyone other than the instructor and TAs.

Important Notes (Don't Skip)

- Non-compiling files will receive a 0 for all associated rubric items
- Do not submit `.class` files
- Test your code in addition to the basic checks on Gradescope
- Submit every file each time you resubmit

- Read the "Allowed Imports" and "Restricted Features" to avoid losing points
- **Check on Ed Discussion for a note containing all official clarifications and sample outputs**

It is expected that everyone will follow the Student-Faculty Expectations document, and the Student Code of Conduct. The professor expects a **positive, respectful, and engaged academic environment** inside the classroom, outside the classroom, in all electronic communications, on all file submissions, and on any document submitted throughout the duration of the course. No inappropriate language is to be used, and any assignment, deemed by the professor, to contain inappropriate, offensive language or threats will get a zero. You are to use professionalism in your work. Violations of this conduct policy will be turned over to the Office of Student Integrity for misconduct.