Homework 04 – The Dessert Conundrum

Topics: ArrayList using generics, Comparable, asymptotics, searching, sorting

Problem Description

Please read this entire document carefully.
Bob's favorite food are desserts (as it is for most of you, I assume). 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 write and submit 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.
- You may assume that inputs to constructors will be valid.
- Floating-point values in Strings should be rounded and displayed to two decimal places.
- **Reuse code when possible!**

Dessert.java

This class is the superclass for other types of dessert. Implement this class to define the basic behaviors of Dessert objects. This class should implement the Comparable interface using generics so that it is comparable to other Desserts. 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.
- A no-argument 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 a String in the following format:

    `"This is a {flavor} dessert with a sweetness of {sweetness}."`

- `equals`

  - This method should properly override Object's `equals` method.
  - Two desserts are equal if they have the same `flavor` and `sweetness`.
- `compareTo`
  - This method will implement the `compareTo` method from the Comparable interface.
  - You may assume that the object passed in will not be null.
  - This dessert is greater than the other if this dessert has greater `sweetness`.
  - If both desserts have equal `sweetness`, then the dessert with the lexicographically greater `flavor` is considered greater.
  - **Note**: Since this implementation only compares the fields of the Dessert class and not those of subclasses, it should only be used to determine a natural ordering between desserts. It should not be used to determine object equality.
- Getters and setters as necessary.

Cake.java

This class is a subclass of Dessert and represents a certain kind of dessert that Bob can pick: cake.

Variables:

- `String frosting` – the frosting of the cake

Constructor(s):

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

Methods:

- `toString`
  - This method should properly override Dessert's `toString` method.
  - It should return a String in the following format:

    `"This is a {flavor} cake with a {frosting} frosting and has a sweetness of {sweetness}."`
- `equals`
  - This method should properly override Dessert's `equals` method.
  - Two cakes are equal if they have the same `flavor, sweetness,` and `frosting`.
- Getters and setters as necessary.

IceCream.java

This class is a subclass of Dessert and represents a certain kind of dessert that Bob can pick: ice cream.

Variables:

- `int scoops` – the number of scoops of the ice cream
- `boolean cone` – whether the ice cream has a cone

Constructor(s):

- A constructor that takes in `flavor`, `sweetness`, `scoops`, and `cone`.
- A constructor that takes in `scoops` and `cone` and sets `flavor` to "vanilla" and `sweetness` to 45.0.
- A no-argument constructor that sets `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 a String in the following format:

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

    - The "has/does not have" depends on the value of the `cone` field.
- `equals`
  - This method should properly override Dessert's `equals` method.
  - Two ice creams are equal if they have the same `flavor`, `sweetness`, `scoops`, and `cone`.
- Getters and setters as necessary.

Store.java

This class represents a store that sells desserts. We are assuming that a store can sell all kinds of dessert.

Variables:

- `String name` – the name of the store
- `ArrayList<Dessert> desserts` – the desserts the store sells

Constructor(s):

- A constructor that takes in `name` and sets `desserts` to an empty `ArrayList`.
  - You may assume the input will be valid.

Methods:

- `addDessert`
  - This method takes in a valid Dessert object and add it to the back of `desserts`.
  - This method should not return anything.
  - This method should run in $O(1)$ time.
- `removeDessert`
  - This method takes in a valid Dessert object and remove and return the first occurrence of an equal dessert from `desserts`.
  - If the dessert is not found, return null.
  - This method should run in $O(n)$ time.

- `findDessert`

  - This method takes in a valid Dessert object and finds and returns the dessert that has equal sweetness and flavor.
  - If a dessert with the same sweetness and flavor is not found, return null.
  - **Note**: You should only consider the sweetness and flavor, and not any other fields.
  - This method should run in $O(log\ n)$ time.
  - Assume that `desserts` has **unique** dessert items and is **sorted** in ascending order according to their natural ordering.
- `sortStore`
  - This method takes no parameters and does not return anything.
  - Sort `desserts` in **ascending** order according to their natural ordering.
  - This method should run in $O(n^2)$ time.
- `checkStore`
  - This method takes in a valid Dessert object and return the number of desserts in the store that is greater than or equal to the dessert passed in.
    - Equivalently, return the number of desserts in the store that are *not* lesser than the dessert passed in.
  - This method should run in $O(n)$ time.
- Getters and setters as necessary.

Bob.java

This class defines Bob's behaviors. All methods should be static. Notice that you have already implemented most of the code necessary for Bob's behaviors, so reuse code as much as possible!

Methods:

- `compareStores`
  - This method takes in two valid Store objects, `store1` and `store2`.
  - Return true if all desserts in `store1` are found in `store2`. Return false otherwise.
  - Assume that the desserts in `store2`'s inventory is **sorted** in ascending order according to their natural ordering.
  - 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 according to their natural ordering to help Bob find his dessert.
  - Return true if Bob is able to a dessert with equal sweetness and flavor. Return false otherwise.
  - 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-uide.pdf under the Checkstyle Resources module on Canvas.) **The Checkstyle cap for this assignment is 30 points.** This means there is a maximum point deduction of up 30 points. 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). In future assignments 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 the class forum 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 assignment (submit early and often). 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 autograder. 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 assignment 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.

You **MAY NOT** use code generation tools to complete this assignment. This includes generative AI tools like ChatGPT.

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.

THIS GRADED ASSESSMENT IS NOT FOR DISTRIBUTION.
ANY DUPLICATION OUTSIDE OF GEORGIA TECH'S LMS IS UNAUTHORIZED.