P01 Vending Machine

Overview

In this assignment, we are going to implement a simple version of a Vending Machine. Each item in the vending machine has a name (description) and an expiration date. Our vending machine has a specific policy for dispensing items. Whichever item matching the requested description will expire soonest will be dispensed first. We are going to use two-dimensional (2D) Java Arrays to implement this vending machine.The Java array is one of several data storage structures that can be used to store and manage a collection of data. Throughout CS300, we are going to spend a fair amount of time using arrays and managing collections of data. This first programming assignment provides a review of using arrays in java.

Grading Rubric

5 points	**Pre-Assignment Quiz:** The P1 pre-assignment quiz is accessible through Canvas before having access to this specification by **11:59PM on Sunday 09/11/2022**. You CANNOT take the pre-assignment quiz for credit passing its deadline. But you can still access it. The pre-assignment quiz contains hints which can help you in the development of this assignment.
20 points	**Immediate Automated Tests:** Upon submission of your assignment to Gradescope, you will receive feedback from automated grading tests about whether specific parts of your submission conform to this write-up specification. If these tests detect problems in your code, they will attempt to give you some feedback about the kind of defect that they noticed. Note that passing all of these tests does NOT mean your program is otherwise correct. To become more confident of this, you should run additional tests of your own.
15 points	**Additional Automated Tests:** When your manual grading feedback appears on Gradescope, you will also see the feedback from these additional automated grading tests. These tests are similar to the Immediate Automated Tests, but may test different parts of your submission in different ways.
10 points	**Manual Grading Feedback:** After the deadline for an assignment has passed, the course staff will begin manually grading your submission. We will focus on looking at your algorithms, use of programming constructs, and the style and readability of your code. This grading usually takes about a week from the hard deadline, after which you will find feedback on Gradescope.

Learning Objectives

The goals of this assignment include:

- reviewing the use of procedure oriented code (prerequisites for this course),
- practicing the use of control structures, custom static methods, and arrays in methods.
- practicing how to manage an ordered collection of data (the vending machine represented as an oversized 2D array) which may contain multiple occurrences of the same element,
- learning how to approach an algorithm, and how to develop tests to demonstrate the functionality of code, and familiarizing yourself with the CS300 grading tests.

Additional Assignment Requirements and Notes

(Please read carefully!)

- Make sure to read carefully through the WHOLE specification provided in this write-up before starting its implementation. Samples of inputs/outputs are provided at the last section of this specification. Read TWICE the instructions and do not hesitate to ask for clarification on piazza if you find any ambiguity.
- Pair programming is **ALLOWED** but not required for this assignment. If you decide to work with a partner on this assignment, REGISTER your partnership NO LATER than **11:59PM on Sunday 09/11/2022** and MAKE SURE that you have read and understood the CS300 Pair Programming Policy
- **NO import statements** are allowed in your `VendingMachine` class.
- **You CAN import** java.util.Arrays in your `VendingMachineTester` class.
- Only your submitted `VendingMachineTester` class contains a **main** method.
- You are NOT allowed to add any constant or variable in this write-up **outside of any method**.
- You CAN define local variables (inside a method) that you may need to implement the methods defined in this program.
- You CAN define **private static** helper methods to help implement the different public static methods defined in this write-up if needed.
- You MUST NOT add any **public methods** to your `VendingMachine` class other than those defined in this write-up.

- All your test methods should be defined and implemented in your `VendingMachineTester.java`.
- All your test methods must be **public static**. Also, they must take **zero arguments**, **return a boolean**, and must be defined and implemented in your `VendingMachineTester` class.
- All String comparisons in the assignment are CASE SENSITIVE.
- We DO NOT consider erroneous input or exceptional situations in this assignment. We suppose that all the input parameters provided to method calls are valid.
- You are also responsible for maintaining secure back-ups of your progress as you work. The OneDrive and GoogleDrive accounts associated with your UW NetID are often convenient and secure places to store such backups.
- Make sure to submit your code (work in progress) of this assignment on Gradescope both early and often. This will 1) give you time before the deadline to fix any defects that are detected by the tests, 2) provide you with an additional backup of your work, and 3) help track your progress through the implementation of the assignment. These tests are designed to detect and provide feedback about only very specific kinds of defects. **It is your responsibility to implement additional testing to verify that the rest of your code is functioning in accordance with this write-up.**
- You can submit your work in progress multiple times on gradescope. Your submission may include methods not implemented or with partial implementation or with a default return statement. But avoid submitting a code which does not compile. A submission which contains compile errors won't pass any of the automated tests on gradescope.
- Feel free to **reuse** any of the provided source code in this write-up verbatim in your own submission.
- All implemented methods including the main method MUST have their own javadoc-style method headers, with accordance to the CS300 Course Style Guide.
- All your classes MUST have a javadoc-style class header.
- If you are not familiar with JavaDoc style comments, zyBook section 1.4 includes some additional details.
- You MUST adhere to the Academic Conduct Expectations and Advice

1 Getting Started

Start by creating a new Java Project in eclipse called P01 Vending Machine, for instance. You MUST ensure that your new project uses Java 17, by setting the "Use an execution environment

JRE:" drop down setting to "JavaSE-17" within the new Java Project dialog box. DO NOT create any module within to your java project. Then, download the two following files and save them directly within your project's src folder (both inside the **default** source package): `VendingMachine.java` and `VendingMachineTester.java`.

CS300 is a course with students coming with different programming language experiences not necessarily Java. Since this is the first assignment, we provided you with the starter code for the defined classes. This won't be the case for the next assignments. You will be required to create the different classes on your own in the future.

2 VendingMachine and VendingMachineTester

Read carefully through the provided starter code of both files. We are going to develop a vending machine that stores a collection of items and check the correctness of its functionalities in the `VendingMachineTester` class. You are required to implement the methods defined in both classes. Below, we provide some more details. Read carefully through them before starting your implementations.

- In this assignment, we are going to use procedural programming. You can notice that all the methods are *static*.
- This assignment involves the use of arrays within methods in Java. Before you start working on the next steps, make sure that you have already completed chapter1 zybooks reading activities.
- The vending machine will be represented by an **oversize** two-dimensional array defined by an array String[][] and a variable of type int keeping track of the number of items available in the vending machine.
  - **String[][] items**: a 2D array of **case-sensitive** strings where items[i][0] represents the description (name) of a given item, and items[i][1] represents its expiration date. We assume per default that the name is not empty nor blank, and that the expiration date is parsable to a positive integer. The date "0" represents January 1st 2023.
  - **int itemsCount**: an int variable which is used to keep track of the number of the size of the vending machine (number of items stored in the array).
- The vending machine can contain multiple occurrences of the same item. Items with the same description can have the same or different expiration dates.
- The `VendingMachine` defines **eight** static methods for you to complete. The specification of each method is provided in each javadoc style method header.
- A javadoc style method header represents the **abstraction** of the method (what the method is supposed to do, what it does take as input, and what is expected to return

as output) **independently of its implementation details**. Read carefully through the provided specification of each method, and do not hesitate to ask for clarification on piazza if it is not clear to you what a method is supposed to do, take as input, or provide as output.

- The vending machine allows the following operations.
  - Users can check whether a vending machine contains an item given its description. This functionality is implemented by the `containsItem` method.
  - Users can ask how many occurrences of an item given its description are in the vending machine (may be 0 or more). This functionality is implemented by the `getItemOccurrences` method.
  - Users can ask how many occurrences of items with a given description and whose expiration date is greater then a given one. This functionality is implemented by the `getItemsOccurrencesByExpirationDate` method.
  - Users can get the index of the next item to be dispensed given its description. This functionality is implemented by the `getIndexNextItem` method.
  - Users can get a String representation of an item stored at a given index. This functionality is implemented by the `getItemAtIndex` method.
  - The vending machine can contain duplicate items (items with the same name or description regardless of their expiration dates). Thus, users can display a summary of the content of the vending machine (formatted as *item_description (item_count)* separated by newlines). This functionality is implemented by the `getItemsSummary` method.
  - Users can add new items. This functionality is implemented by the `addItem` method.
  - Users can remove (dispense) an item. This functionality is implemented by the `removeNextItem` method.
- Items can be added to and removed from the vending machine as follows.
  - A new item defined by its description and expiration date must be appended, meaning added to the end of the vending machine if it is not full.
  - Given its description, the item with the soonest (smallest) expiration date must be removed (dispensed) first. If no match found with the provided item description, no action is required. If there are multiple items with **the same description and the same expiration date**, the item stored at the lowest index (meaning coming first in the order of the array) will be dispensed. You will have to traverse the whole array looking for a match of the item with the smallest expiration date.
  - The vending machine is an ordered array. This means that the order of elements in the array matters. The remove operation should include a shift operation so that the array *items* is contiguous with no gaps (no null references) in the range of indexes `0..itemsCount-1` after decrementing *itemsCount* by one.

- All the above methods are not simple and can be decomposed in more than one step. Make sure to add implementation level comments to highlight each step.

Test methods come FIRST!

- Before writing any code for the above methods, it is HIGHLY recommended to implement the tester methods first. This will allow you to better understand the functionality of each method and define tester scenarios to cover normal and edge cases. For a given input (test scenario), the tester method checks the correctness of the output and the state of the vending machine after the method returns (expected behavior).
- A tester method should return true if and only if NO bug is detected. If it detects any bug it must return false. A tester method is better than another tester method if it can detect more bugs within a broken implementation.
- A tester method must be **static**, returns a **boolean**, and takes **NO** input.
- We provided you with an example of the implementation of a tester method named `testGetIndexNextItem` in the `VendingMachineTester` class. You can use it as a reference to implement the remaining tester methods.
- A tester method includes only ONE `return true` statement at the end (last statement). Any additional test scenario must be added before that `return true` statement. If any bug is detected, return false. It is recommended to print out more specific feedback when tests fail (before returning false). This helps you detect which test scenario was failed by the broken implementation, locate the bug, and fix it.
- A common trap when writing tests is to make the test code as complex or even more complex than the code that it is meant to test. This can lead to there being more bugs and more development time required for testing code, than for the code being tested. To avoid this trap, we aim to make our test code as simple as possible while varying our test scenarios to account edge cases.
- We did not provide you with javadoc style method header for the tester methods. Make sure to add a complete one of each of these methods.
- Remove the comments including the `TODO` tags from your last submission to gradescope.

3 VendingMachine methods: Samples of input/output

In the following, we provide you with a set of inputs and their expected outputs for each of the methods defined in the `VendingMachine` class. These scenarios do not cover edge cases. To avoid any integrity issues, avoid using these exact test scenarios in your tester methods and try to come up with your own test scenarios.

3.1 addItem

```
input items: {{"Water", "50"}, {"Soda", "60"}, {"Cookies", "10"}, {"Water", "20"},
              null, null}
input itemsCount = 4;

addItem("Soda", "100", items, itemsCount); // must return 5 and
the contents of the items array is expected to be:
items: {{"Water", "50"}, {"Soda", "60"}, {"Cookies", "10"}, {"Water", "20"},
        {"Soda", "100"}, null}
```

3.2 containsItem

```
input items: {{"Water", "50"}, {"Soda", "60"}, {"Cookies", "10"}, {"Water", "20"},
              {"Soda", "100"}, null}
input itemsCount = 5;

containsItem("Soda", items, itemsCount); // must return true and
the contents of the items array is expected to be:
items: {{"Water", "50"}, {"Soda", "60"}, {"Cookies", "10"}, {"Water", "20"},
        {"Soda", "100"}, null}

containsItem("Juice", items, itemsCount); // must return false and
the contents of the items array is expected to be:
items: {{"Water", "50"}, {"Soda", "60"}, {"Cookies", "10"}, {"Water", "20"},
        {"Soda", "100"}, null}
```

3.3 getIndexNextItem

```
input items: {{"Water", "50"}, {"Soda", "60"}, {"Cookies", "10"}, {"Water", "20"},
              {"Soda", "100"}, {"Water", "30"}, null, null, null}
input itemsCount = 6;

getIndexNextItem("Water", items, itemsCount); // must return 3 and
the contents of the items array is expected to be:
items: {{"Water", "50"}, {"Soda", "60"}, {"Cookies", "10"}, {"Water", "20"},
       {"Soda", "100"}, {"Water", "30"}, null, null, null}
```

3.4 getItemAtIndex

```
input items: {{"Water", "50"}, {"Soda", "60"}, {"Cookies", "10"}, {"Water", "20"},
              {"Soda", "100"}, {"Water", "30"}, null, null, null}
```

```
input itemsCount = 6;

getItemAtIndex(0, items, itemsCount); // must return "Water 50" and
the contents of the items array is expected to be:
items: {{"Water", "50"}, {"Soda", "60"}, {"Cookies", "10"}, {"Water", "20"},
       {"Soda", "100"}, {"Water", "30"}, null, null, null}

getItemAtIndex(4, items, itemsCount); // must return "Soda 100" and
the contents of the items array is expected to be:
items: {{"Water", "50"}, {"Soda", "60"}, {"Cookies", "10"}, {"Water", "20"},
       {"Soda", "100"}, {"Water", "30"}, null, null, null}

getItemAtIndex(6, items, itemsCount); // must return  "ERROR INVALID INDEX"
and the contents of the items array is expected to be:
items: {{"Water", "50"}, {"Soda", "60"}, {"Cookies", "10"}, {"Water", "20"},
       {"Soda", "100"}, {"Water", "30"}, null, null, null}
```

3.5 getItemOccurrences and getItemOccurrencesByExpirationDate

```
input items: {{"Water", "50"}, {"Soda", "60"}, {"Cookies", "10"}, {"Water", "20"},
              {"Soda", "100"}, {"Water", "30"}, {"Soda", "20"}, null, null, null}
input itemsCount = 7;

getItemOccurrences("Soda", items, itemsCount); // must return 3 without changing
the contents of the array items

getItemOccurrencesByExpirationDate("Soda", 50, items, itemsCount); // must return 2
without changing the contents of the array items
```

3.6 getItemsSummary

```
input items: {{"Water", "50"}, {"Soda", "60"}, {"Cookies", "10"}, {"Water", "20"},
              {"Soda", "100"}, {"Water", "30"}, null, null, null}
input itemsCount = 6;

getItemsSummary(items, itemsCount); // must return the following formatted as a string:
Water (3)
Soda (2)
Cookies (1)
```

3.7 removeNextItem

```
input items: {{"Water", "50"}, {"Soda", "60"}, {"Cookies", "10"}, {"Water", "20"},
             {"Soda", "100"}, {"Water", "30"}, null, null, null}
input itemsCount = 6;

removeNextItem("Water", items, itemsCount); // must return 5 and
the contents of the items array is expected to be:
items: {{"Water", "50"}, {"Soda", "60"}, {"Cookies", "10"}, {"Soda", "100"},
       {"Water", "30"}, null, null, null, null}
```

4 Assignment Submission

Congratulations on finishing this CS300 assignment! After verifying that your work is correct, and written clearly in a style that is consistent with the CS300 Course Style Guide, you should submit your final work through Gradescope. The only TWO files that you must submit include: `VendingMachine.java` and `VendingMachineTester.java` . Your score for this assignment will be based on your **"active"** submission made prior to the hard deadline. The second portion of your grade for this assignment will be determined by running that same submission against additional offline automated grading tests after the submission deadline. Finally, the third portion of your grade for your submission will be determined by humans looking for organization, clarity, commenting, and adherence to the CS300 Course Style Guide.