Homework 06 – Library

Problem Description

Hello! Please make sure to read all parts of this document carefully.

You have been asked by GT's very own Crosland library to simulate how they store physical media such as books and magazines. It wouldn't be a library without order either, so you will oversee keeping the books sorted as you go adding books to the catalogue. You are asked to create a Summarizable.java, LibraryItem.java, Library.java, Book.java, and Magazine.java as the composition of the simulation.

Remember to test for Checkstyle Errors and include Javadoc Comments!

Solution Description

You will create one interface, one abstract class, and two concrete subclasses. You will be creating a number of fields and methods for each file. Based on the description given for each variable and method, you will have to decide whether or not the variables/method should be static, and whether it should be private or public. To make these decisions, you should carefully follow the guidelines on these keywords as taught in the modules.

Hint: A lot of the code you will write for this assignment can be reused. Try to think of what keywords you can use that will help you! You should be able to put @Override on the line before the method header for any methods you override.

For all classes, include getters and setters when applicable.

`Summarizable.java`

This file defines an **interface** for objects that can be summarized.

Methods

- `summarize()`
  - Abstract method that does not take in anything and returns a summary of the object as a String
  - (Note: any class that implements `Summarizable` must provide a method definition for this method)

`LibraryItem.java`

This class represents a library item. You must not be able to create an instance of this class (**Hint:** there is a keyword that prevents us from creating instances of a class). **LibraryItem should implement the `Comparable` interface with the proper type parameter. LibraryItems should also be `Summarizable`.**

Variables

- `String title` – The title of the item
- `int libraryCode` – This represents a code given to the item and will be used to compare and sort items

Constructors

- A constructor that takes in the `title` and `libraryCode` in that order and sets the instance variables accordingly

Methods

Do not create any other methods than those specified. Any extra methods will result in point deductions. All methods must have the proper visibility to be used where it is specified they are used.

1. `summarize()`
    a. Adhering to the Summarizable contract, this method returns the String summary "This item is called [title]."
2. `compareTo(LibraryItem other)`
    a. Override the `compareTo` method (You should be able to put @Override on the line before the method header)
    b. Takes in a `LibraryItem` object and returns an int, adhering to the API contract (Comparable Interface) (This will only work if you have the proper generic type in the class header)
    c. A `LibraryItem` is greater than another `LibraryItem` if its `libraryCode` is less than the other one's. For example, if `LibraryItem` itemOne has `libraryCode` = 8 and `LibraryItem itemTwo` has `libraryCode` = 5, `itemOne` is less than `itemTwo`. (Logically, this will allow us to sort from highest to lowest `libraryCode`).

`Library.java`

This file defines a class called Library.

Variables

- `LibraryItem[] bookshelf` - An array that simulates a bookshelf where we will hold objects of type LibraryItem. This array should not contain any null values for input. You don't have to check since we'll only input non-null values and full arrays but keep that in mind for your own testing. This array should always be sorted from highest code to lowest code.

Constructors

- A zero argument constructor that initializes an empty bookshelf (size zero).
- A one argument constructor that takes in an array of `LibraryItems` that will be the books on the bookshelf. Remember that we want the `bookshelf` to be always sorted, and that you should not assume that the passed in array is sorted (there's an easy method for this operation)

Methods

- browseLibraryItems()
    - This method should print out the summaries of all LibraryItems in the bookshelf for the Library.
    - Format for the print statements: "[libraryCode]: [summary]" on a new line
    - e.g.
    - 80: This item is called Whales. The book is 700 pages long. It's all about whales!
    - 44: This item is called Time. The cover looks like a weird piece of art.
    - Remember to disregard the brackets
    - The method should return nothing.
- addLibraryItem(LibraryItem newItem)

    - This method should create a new array of length + 1 that copies over all the previous elements as well as adds the new item. (Note: For Arrays.sort() to work, the array must not have any null values, which is why we keep it full)
    - It should then sort the array as to maintain the bookshelf's order.
    - The method should return nothing.
- getLibraryItem(int code)
    - Returns the library item with given code. If the code doesn't exist, return null.
    - If there are libraryItems with the same code, return the first one found with the code.
    - The method should return a LibraryItem
- getNumberOfItems()
    - Returns the number of items on the shelf
    - The method should return int

Book.java

This file defines a Book. **Have Book extend the LibraryItem class.**

Variables

All variables must be not allowed to be directly modified outside the class in which they are declared, unless otherwise stated in the description of the variable. **Hint**: there is a specific visibility modifier that can do this!

The Book class must have these variables. Do **NOT** re-declare any of the instance variables declared in LibraryItem class:

- `backcoverBlurb` - a String that represents a short description of the Book.
- `pages`- an int that represents the number of pages the Book has. Cannot be negative.

Constructors

- A constructor that takes in the title, libraryCode, backcoverBlurb, and pages and sets all fields accordingly. It must accept the variables in the specified order.

Methods

Do not create any other methods than those specified. Any extra methods will result in point deductions. All methods must have the proper visibility to be used where it is specified they are used.

- `summarize()`
    - Return LibraryItem's summarize() + "The book is [pages] pages long. [backcoverBlurb]."
    - e.g. This item is called [title]. The book is [pages] pages long. [backCoverBlurb].
        - Note, the brackets should not be included in the returned String
        - Note the spacing before "The book …"

Magazine.java

This file defines a Magazine. **Have Magazine extend the LibraryItem class.**

Variables

All variables must be not allowed to be directly modified outside the class in which they are declared, unless otherwise stated in the description of the variable. **Hint**: there is a specific visibility modifier that can do this!

- coverDescription- A string that represents the short description of the magazine

Constructors

- A constructor that takes in the title, LibraryCode, coverDescripton and sets all fields accordingly. It must accept the variables in the specified order.

Methods

Do not create any other methods than those specified. Any extra methods will result in point deductions. All methods must have the proper visibility to be used where it is specified they are used.

- `summarize()`
    - Return LibraryItem's summarize() + "The cover looks like [coverDescription]." without the brackets
    - e.g. This item is called Times. The cover looks like a man with a hat.

Reuse your code when possible. Certain methods can be reused using certain keywords. If you use the `@Override` tag for a method, you will not have to write the Javadocs for that method. These tests and the ones on Gradescope are by no means comprehensive so be sure to create your own!

Rubric

[5] Summarizable.java

- [1] Correct interface header
- [4] Correctly implements specified methods
    - [4] Correct summarize() method

[14] LibraryItem.java

- [3] Correct class header
- [2] Correct instance variables
- [3] Correctly implements constructors
- [6] Correctly implements specified methods
    - [3] Correct summarize()
    - [3] Correct compareTo(LibraryItem other)

[50] Library.java

- [2] Correct instance variables
- [5] Correctly implements constructors
- [43] Correctly implements specified methods
    - [13] browseLibraryItems()
    - [12] addLibraryItem(LibraryItem newItem)
    - [10] getLibraryItem(int code)
    - [8] getNumberOfItems()

[16] Book.java

- [1] Correct class header
- [2] Correct instance variable(s)
- [5] Correctly creates constructors
- [8] Correctly implements specified methods

    - [8] Correctly implements and overrides summarize()

[15] Magazine.java

- [1] Correct class header
- [1] Correct instance variables
- [5] Correctly creates constructors
- [8] Correctly implements specified methods
    - [8] Correctly implements and overrides summarize()

We reserve the right to adjust the rubric, but this is typically only done for correcting mistakes.

Allowed Imports

To prevent trivialization of the assignment, you are only allowed to import `java.util.Arrays`. You are *not* allowed to import any other classes or packages.

Feature Restriction

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`

Javadoc

For this assignment, you will be commenting your code with Javadoc. Javadoc comments are a clean and useful way to document your code's functionality. For more information on what Javadoc comments are and why they are awesome, the online overview for them is extremely detailed and helpful.

You can generate the Javadoc overview for your code using the command below, which will put all the files into a folder named "javadoc". Note you should execute this after adding Javadoc comments to your code:

```
$ javadoc *.java -d javadoc
```

The relevant tags that you need to include are `@author`, `@version`, `@param`, and `@return`. Here is an example of a properly Javadoc'd class:

```
import java.util.Scanner;


/**
 *This class represents a Dog object.
 *@author George P.Burdell
 *@version 1.0
 */
 public class Dog {
```

```
    /**
     *Creates an awesome dog(NOT a dawg!)
     */
     public Dog() {
     ...
     }

     /**
      *This method takes in two ints and returns their sum
      *@param a first number
      *@param b second number
      *@return sum of a and b
      */
      public int add(int a,int b) {
      ...
      }
}
```

A more thorough tutorial for Javadoc Comments can be found here.

Take note of a few things:

1. Javadoc comments begin with `/**` and end with `*/`.
2. Every class you write must be Javadoc'd and the `@author` and `@verion` tag included. The comments for a class should start with a brief description of the role of the class in your program.
3. Every non-private method you write must be Javadoc'd and the `@param` tag included for every method parameter. The format for an `@param` tag is `@param <name of parameter as written in method header> <description of parameter>`. If the method has a non-void return type, include the `@return` tag which should have a simple description of what the method returns, semantically.

Checkstyle can check for Javadoc comments using the `-a` flag, as described in the next section.

Checkstyle

For this assignment, we will be enforcing style guidelines with Checkstyle, a program we use to check Java style. Checkstyle can be downloaded here.

To run Checkstyle, put the jar file in the same folder as your homework files and run

```
java -jar checkstyle-6.2.2.jar -a *.java
```

The Checkstyle cap for this assignment is **20 points**. This means that up to 20 points can be lost from Checkstyle errors.

Collaboration

Collaboration Statement

To ensure that you acknowledge a collaboration and give credit where credit is due, **we require that you place a collaboration statement as a comment at the top of at least one .java file that you submit**. That collaboration statement should say either:

I worked on the homework assignment alone, using only course materials.

or

In order to help learn course concepts, I worked on the homework with [give the names of the people you worked with], discussed homework topics and issues with [provide names of people], and/or consulted related material that can be found at [cite any other materials not provided as course materials for CS 1331 that assisted your learning].

Allowed Collaboration

When completing homeworks for CS1331 you may talk with other students about:

- What general strategies or algorithms you used to solve problems in the homeworks
- Parts of the homework you are unsure of and need more explanation
- Online resources that helped you find a solution
- Key course concepts and Java language features used in your solution

You may **not** discuss, show, or share by other means the specifics of your code, including screenshots, file sharing, or showing someone else the code on your computer, or use code shared by others.

Examples of approved/disapproved collaboration:

- **approved**: "Hey, I'm really confused on how we are supposed to implement this part of the homework. What strategies/resources did you use to solve it?"
- **disapproved**: "Hey, it's 10:40 on Thursday… Can I see your code? I won't copy it directly I promise"

In addition to the above rules, note that it is not allowed to upload your code to any sort of public repository. This could be considered an Honor Code violation, even if it is after the homework is due.

Turn-In Procedure

Submission

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

- Summarizable.java
- Library.java
- LibraryItem.java
- Book.java
- Magazine.java

Make sure you see the message stating "HW06 submitted successfully". From this point, Gradescope will run a basic autograder on your submission as discussed in the next section.

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 last submission: be sure to **submit every file each time you resubmit**.

Gradescope Autograder

For each submission, you will be able to see the results of a few basic test cases on your code. Each test typically corresponds 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.

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 Piazza for a note containing all official clarifications