Object-Oriented Programming

LECTURER: NGUYEN THI THU TRANG, TRANGNTT@SOICT.HUST.EDU.VN
TEACHING ASSISTANTS: NGUYEN T.T. GIANG, GIANG.NTTI94750@SIS.HUST.EDU.VN
VUONG DINH AN, AN.VDI800003@SIS.HUST.EDU.VN

Lab 05: Aggregation and Inheritance

In this lab, you will practice with:

- Java Inheritance mechanism
- Use the Collections framework (specifically, **`ArrayList`**)

0 Assignment Submission

For this lab class, you will have to turn in your work twice, specifically:

- **Right after the class**: for this deadline, you should push any work you have done within the lab class to Github.
- **10 PM three days after the class**: for this deadline, you should include your work in this lab, and push it to a branch called "***release/lab05***" of the valid repository.

After completing all the exercises in the lab, you have to update the use case diagram and the class diagram of the AIMS project.
Each student is expected to turn in his or her work and not give or receive unpermitted aid. Otherwise, we would apply extreme methods for measurement to prevent cheating. Please write down answers for all questions into a text file named "**answers.txt"** and submit it within your repository.

1 Additional requirements of AIMS

Starting from this lab, you extend the AIMS system that you created in the previous exercises to allow customer to order 2 new types of media: books and CD.
A book's information includes: id, title, category, cost and list of authors.
A CD's information includes: id, title, category, artist, director, track list and cost. Additionally, each track is unique in a CD with its own title and length. The length of a CD is sum of the lengths of its tracks. When a user sees the details of a media in the store, the information displayed depends on the type of media.

- For books, the system shows their title, category, author list, the contentlength (i.e., the number oftokens).
- For CDs, the system displays the CD's information (i.e. CD title, category, artist, director, CD length, and the cost for the CD) and then displays the information of all the tracks in that CD.
- For DVDs, the system displays the DVD's information (i.e. DVDtitle, category, director, DVD length, and the cost for the DVD).

Additionally, the user can choose to play some media when browsing the list of media in the store or seeing the current cart. For simplicity, we establish the way the system plays a media is as follows: When a CD is played, the system displays the CD information (i.e., CD title and CD length) and plays all the tracks of the CD. To play a track, the system displays the track's name and its length. Similarly, a DVD can also be played, i.e., the system displays the title and length of the DVD. If a DVD or track has the length 0 or less, the system must notify the user that the track, the DVD or the CD of that track cannot be played.

2 Creating the **Book** class

- In the Package Explorer view, right-click the project and select New -> Class. Adhere to the following specifications:

- Package: **hust.soict.globalict.aims.media**
- Name: **Book**
- Access modifier: **public**
- Superclass: **java.lang.Object**
- **public static void main(String[] args)**: **do not check**
- Constructors from Superclass: **Check**
- All other boxes: **Do not check**

Add fields to the Book class

- To store the information about a **Book**, the class requires five fields: an **int** field **id, String** fields **title** and **category**, a **float** field **cost,** and an **ArrayList** of **authors**. You will want to make these fields private, with public accessor methods for all but the **authors** field.

```
public class Book {

    private int id;
    private String title;
    private String category;
    private float cost;
    private List<String> authors = new ArrayList<String>();

    public Book() {
        // TODO Auto-generated constructor stub
    }
```

Figure 3. Adding fields to Book class

- Instead of typing the accessor methods for these fields, you may use the **Generate Getter and Setter** option in the **Outline** view pop-up menu (i.e., Right Click -> Source -> Generate Getters and Setters...). Note that in reality, not all attributes need to have getter and setter. We only create this when necessary. Getter and setter generator of Eclipse also let you decide which attribute will get getter or setter or both.
- Next, create **addAuthor(String authorName)** and **removeAuthor(String authorName)** for the **Book** class
    - The **addAuthor**(...) method should ensure that the author is not already in the **ArrayList** before adding
    - The **removeAuthor**(...) method should ensure that the author is present in the **ArrayList** before removing
    - Reference to some useful methods of the **ArrayList** class

3 Creating the abstract **Media** class

At this point, the **DigitalVideoDisc** and the **Book** classes have some fields in common namely id, title, category, and cost. Here is a good opportunity to create a common superclass between the two, to eliminate the duplication of code. This process is known as refactoring. You will create a class called **Media** which contains these fields and their associated get and set methods.

Create the Media class in the project

- In the **Package Explorer** view, right-click on the project and select New -> Class. Adhere to the following specifications for the new class:

- Package: **hust.soict.dsai.aims.media**
- Name: **Media**
- Access Modifier: **public**
- Superclass: **java.lang.Object**
- Constructors from Superclass: Check
- **public static void main (String[] args)**: do not check
- All other boxes: Do not check

- Add fields to the **Media** class

- To store the information common to the **DigitalVideoDisc** and the **Book** classes, the **Media** class requires four private fields: **int id, String title**, **String category** and **float cost**
- The **Media** class now handles the tasks of managing the **id**s of all **Media** objects. Previously, the **DigitalVideoDisc** class managed the **id**s of all **DigitalVideoDisc** objects via the classifier member **nbDigitalVideoDiscs** to keep track of the number of DVDs created. Please add an appropriate field to the **Media** class to allow a similar way to manage the **id**s of all **Media** objects.
- Just like a DVD, each **Media** also has a date when it is added to the store. Add an appropriate field to the **Media** class to keep this information.
- You will want to make public accessor methods for these fields (by using **Generate Getter and Setter** option in the **Outline** view pop-up menu)

- Remove fields and methods from **Book** and **DigitalVideoDisc** classes

- Open the Book.java in the editor
- Locate the Outline view on the right-hand side
- Select the fields id, title, category, cost and their accessors & mutators (if exist)
- Right-click the selection and select Delete from the pop-up menu
- Save your changes

- Do similarly for the **DigitalVideoDisc** class and move it to the package **hust.soict.dsai.aims.media**. Remove the package **hust.soict.dsai.aims.disc.**

- After doing that you will see a lot of errors because of the missing fields

- Extend the **Media** class for both **Book** and **DigitalVideoDisc**
  - **public class Book extends Media**
  - **public class DigitalVideoDisc extends Media**
- Save your changes.

4 Update the Cart class to work with Media

You must now update the **Cart** class to accept both **DigitalVideoDisc** and **Book**. Currently, the **Order** class has methods:

- **addDigitalVideoDisc**()
- **removeDigitalVideoDisc**().

You could add two more methods to add and remove **Book**, but since **DigitalVideoDisc** and **Book** are both subclasses of type **Media**, you can simply change **Cart** to maintain a collection of **Media** objects. Thus, you can add either a **DigitalVideoDisc** or a **Book** using the same methods.

- Remove the **itemsOrdered** array, as well as its add and remove methods.
  - From the **Package Explorer** view, expand the project
  - Double-click **Cart.java** to open it in the editor
  - In the **Outline** view, select the **itemsOrdered** array and the methods **addDigitalVideoDisc**() and **removeDigitalVideoDisc**() and hit the **Delete** key
  - Click **Yes** when prompted to confirm the deletion
- The **qtyOrdered** field is no longer needed since it was used to track the number of **DigitalVideoDiscs** in the **itemsOrdered** array, so remove it and its accessor and mutator (if exist).
- Add the **itemsOrdered** to the **Cart** class
  - Recreate the **itemsOrdered** field, this time as an object **ArrayList** instead of an array.
  - To create this field, type the following code in the **Cart** class, in place of the **itemsOrdered** array declaration that you deleted:

    private ArrayList<Media> itemsOrdered = new ArrayList<Media>();
- Note that you should import the **java.util.ArrayList** in the **Cart** class
  - A quicker way to achieve the same effect is to use the Organize Imports feature within Eclipse
  - Right-click anywhere in the editor for the Cart class and select Source -> Organize Imports (Or Ctrl+Shift+O). This will insert the appropriate import statements in your code.
  - Save your class
- Create **addMedia**() and **removeMedia**() to replace **addDigitalVideoDisc**() and **removeDigitalVideoDisc**()
- Update the **totalCost**() method

5 Update the **Store** class to work with **Media**

- Similar to the **Cart** class, change the **itemsInStore[]** attribute of the **Store** class to **ArrayList<Media>** type.
- Replace the **addDigitalVideoDisc**() and **removeDigitalVideoDisc**() methods with **addMedia**() and **removeMedia**()

6 Extending the AIMS project to allow the ordering of CDs

You will further extend the AimsProject to allow the ordering of another type of media – CompactDisc (CD). As with **DigitalVideoDisc** and **Book**, the **CompactDisc** class will extend **Media**, inheriting the **id**, **title**, **category** and **cost** fields and the associated methods.

You can apply Release Flow here.

*6.1 Create the **Disc** class extending the **Media** class*

- The **Disc** class has two fields: **length** and **director**
- Create **getter** methods for these fields
- Create constructor(s) for this class. Use super() if possible.
- Make the **DigitalVideoDisc** extending the **Disc** class. Make changes if need be.
- Create the **CompactDisc** extending the **Disc** class. Save your changes.

*6.2 Create the **Track** class which models a track on a compact disc and will store information including the **title** and **length** of the track*

- Add two fields: **String title** and **int length**
- Make these fields **private** and create their **getter** methods as **public**
- Create constructor(s) for this class.
- Save your changes

6.3 Open the **CompactDisc** *class*

- Add 2 fields to this class:
  - a **String** as **artist**
  - an **ArrayList** of **Track** as **tracks**
- Make all these fields as **private** attributes. Create a public **getter** method for only **artist**.
- Create constructor(s) for this class. Use super() if possible.
- Create methods **addTrack**() and **removeTrack**()
  - The **addTrack**() method should check if the input track is already in the list of tracks and inform users

- The **removeTrack**() method should check if the input track existed in the list of tracks and inform users

- Create the **getLength**() method
  - Because each track in the CD has a length, the length of the CD should be the sum of the lengths of all its tracks.
- Save your changes

7 Create the Playable interface

The **Playable** interface is created to allow classes to indicate that they implement a **play**() method. You can apply Release Flow here by creating a **topic** branch for implementing the **Playable** interface.

- Create a **Playable** interface, and add to it the method prototype: **public void play();**
- Save your changes
- Implement the **Playable** with **CompactDisc**, **DigitalVideoDisc** and **Track**
  - For each of these classes **CompactDisc** and **DigitalVideoDisc**, edit the class description to include the keywords **implements Playable**, after the keyword **extends Disc**
  - For the **Track** class, insert the keywords **implements Playable** after the keywords **public class Track**
- Implement **play**() for **DigitalVideoDisc** and **Track**
  - Add the method **play**() to these two classes
  - In the **DigitalVideoDisc**, simply print to screen:

```
public void play() {
      System.out.println("Playing DVD: " + this.getTitle());
      System.out.println("DVD length: " + this.getLength());
}
```

  - Similar additions with the **Track** class
- Implement **play**() for **CompactDisc**
  - Since the **CompactDisc** class contains an **ArrayList** of **Tracks**, each of which can be played on its own. The **play**() method should output some information about the **CompactDisc** to console
  - Loop through each track of the ArrayList and call **Track's** play() method

8 Get a lucky item in the cart

- In the **Cart** class, write a method **Media getALuckyItem()** which randomly picks out (remember to use **Math.random()**) an item for free. The minimum quantity of media in the order to get a lucky item is 5.
- When the customer place order, the system will display which media is a free and lucky item. Remember to update the total bill of this order after getting a lucky item.

9 Update **AIMS** class

- You will update the **AIMS** class along with all your changes above.

- Update the application to allow the system to work with new types of media.
- ~~Update the menu of choices:~~
  - ~~When the user is browsing the list of media in the store, the software should allow them to play any media they choose if it is a **CompactDisc** or **DigitalVideoDisc**~~
  - ~~When seeing the current cart, the user should also be able to play any media of the type **CompactDisc** or **DigitalVideoDisc** in the cart~~
- Update the place order feature to get a lucky item as required above

10 Constructors of whole classes and parent classes

- Update the UML class diagram for the **AimsProject**. Update the new .astah & .png file in the **Design** directory. We can apply Release Flow here by creating a branch, e.g., **topic/update-class-diagram/aims-project/lab05,** pushing the diagram and its image, and then merging it with the master.
- Which classes are aggregates of other classes? Checking all constructors of whole classes if they initialize for their parts?
- Write constructors for parent and child classes. Remove redundant setter methods if any

11 Practice with **Threads** changes

Threads allow you to perform multiple jobs concurrently. This helps utilize the power of your system, as well as makes the program run more productive. This is especially useful when you run functionalities that have infinite loops.

*11.1 Create the **MemoryDaemon** class*

A class called **MemoryDaemon** runs as a daemon thread, tracking the memory usage in the system.

- The **MemoryDaemon** class implements the **java.lang.Runnable** interface
- Implement the **run**() method
- Add the field **long memoryUsed** to the **MemoryDaemon** class
- Initialize this field to 0. It keeps track of the memory usage in the system.
- Add code to the **run**() method to check memory usage as the program runs

Create a loop that will log the amount of memory used as the **Aims.main**() method executes. You will make use of the **java.lang.Runtime** class, which has a static method **getRuntime**().

```
public void run(){
   Runtime rt = Runtime.getRuntime();
   long used;

   while (true) {
      used = rt.totalMemory() - rt.freeMemory();
      if (used != memoryUsed) {
        System.out.println("\tMemory used = " + used);
        memoryUsed = used;
      }
   }
}
```

Figure 2. Example of getting runtime of the program

- Save your changes

*11.2 Update the **Aims** class*

The **Aims** class will create a new **MemoryDaemon** object and run it as a normal object and as a daemon thread

- Open the **main**() method of **Aims** class
- Add code to create a new **MemoryDaemon** object
- Add code to get the **MemoryDaemon** performing **run**() method.
- Run the program, observe the result
- Modify the code by using this object in the constructor of the **Thread** class to create a new **Thread** object
- Using the **setDaemon**() method to indicate that this thread is a daemon thread, add code to start the thread.
- Save and run the program to see the changes.