Object-Oriented Programming

Lecturer: NGUYEN Thi Thu Trang, trangntt@soict.hust.edu.vn
Teaching Assistant: TRINH Thu Hai, hai.tt184255@sis.hust.edu.vn

Lab 04: Inheritance and Polymorphism

In this lab, you will practice with:

- Java Inheritance mechanism
- Abstract class and interface
- Use the Collections framework (specifically, **`ArrayList`**)
- Polymorphism
- Re-organize the application with menu-based UI for users.

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 include any work you have done within the lab class.
- 10PM **three** days after the class: for this deadline, you should include the **source code** of all sections of this lab, into a branch namely "**release/lab04**" 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 AIMS project.

Each student is expected to turn in his or her own work and not give or receive unpermitted aid. Otherwise, we would apply extreme methods for measurement to prevent cheating. Please note that you need to write down answers for all questions into a text file and submit within it within your repository.

1. Import the existing project into the workspace of Eclipse

- Open Eclipse

- Open File -> Import. Type zip to find Archive File if you have exported as a zip file before. You may choose Existing Projects into Workspace if you want to open an existing project in your computer. Ignore this step if the AimsProject is already opened in the workspace.

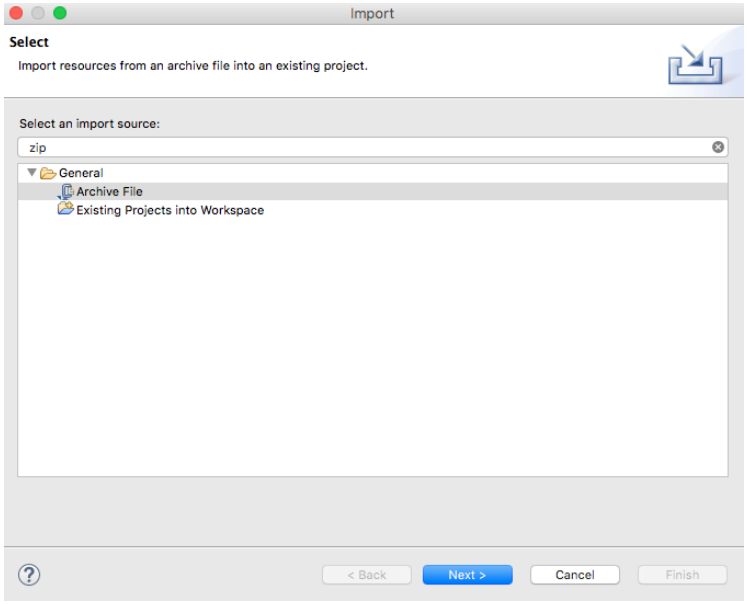

Figure 1: Import existing project

- Click Next and Browse to a zip file or a project to open

Once the project is imported, you should see the classes you created in the previous lab, namely, **`Aims`**, **`Cart`**, **`DigitalVideoDisc`**.

- We can apply Release Flow here by creating a branch, e.g., **`topic/aims-project/add-media-class`**, writing our codes, testing them, pushing them, and then merging it with master.

2. 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 its id, title, category, cost and list of authors.

A CD's information includes: id, title, category, artist, director, track list and price. 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 content length (i.e., the number of tokens).
- 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. DVD title, 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.

3. 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.dsai.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 2: 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 with 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

4. Creating the abstract **Media** class

At this point, the **DigitalVideoDisc** and the **Book** classes have some fields in common namely 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 an abstract class called **Media** which contains these three fields and their associated get and set methods.

Create the Media class in the project

- In the **Package Explorer** view, right click to 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**, **abstract**
  - 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: **String title**, **String category** and **float cost**
  - 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 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.

5. Creating the **CompactDisc** class

As with **DigitalVideoDisc** and **Book**, the **CompactDisc** class will extend **Media**, inheriting the **title**, **category** and **cost** fields and the associated methods.

5.1. 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

5.2. Create the **`Track`** *class which models a track on a compact disc and will store information incuding 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

5.3. 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.`** Create 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 lengths of all its tracks.
- Save your changes

6. 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 **`Playable`** interface.

- Create **`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 a **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

7. Update the **Cart** class to work with **Media**

You must now update the **Cart** class to accept not only **DigitalVideoDisc** but also **Book** and **CompactDisc**. Currently, the **Cart** class has methods:

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

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

- Remove the **itemsOrdered** array, as well as its add and remove methods.
  a. From the **Package Explorer** view, expand the project
  b. Double-click **Cart.java** to open it in the editor
  c. In the **Outline** view, select the **itemsOrdered** array and the methods **addDigitalVideoDisc**() and **removeDigitalVideoDisc**() and hit the **Delete** key
  d. Click **Yes** when prompted to confirm the deletion
  e. Recreate the **itemsOrdered** field, this time as an object **ArrayList** instead of an array.
  - 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

f. 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

g. A quicker way to achieve the same affect is to use the Organize Imports feature within Eclipse

h. 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.

i. Save your class

- Create **addMedia**() and **removeMedia**() to replace **addDigitalVideoDisc**() and **removeDigitalVideoDisc**()
- Update the **totalCost**() method

8. 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**()

9. 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/lab04,** push the diagram and its image, and then merge with 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

10. Unique item in a list

To make sure the list of media in cart or list of tracks in a CD should not contain identical objects, we can override the **equals()** method of the **Object** class

Previously, when you add a media to a cart or a track to a CD, you may use the `contains()` methods of the list of medias in the cart or the list of tracks in the CD to ensure that a similar object is not added to that list.

When calling this by default, `contains()` will check a variety of condition in order to confirm that 2 tracks are identical. However, you can define your own requirements to perform the verification.

The `contains()` method returns true if the list contains the specific element. More formally, it returns true if and only if the list contains at least one element `e` such that `e` such that

`(o==null?e==null:o.equals(e))`. So the `contains()` method actually use `equals()` method to check equality.

Please override the `boolean equals(Object o)` of the `Media` and the `Track` class so that two objects of these classes can be considered as equal if:

+ For the `Media` class: the `title` is equal
+ For the `Track` class: the `title` and the `length` are equal

When overriding the `equals()` method of the `Object` class, you will have to cast the `Object` parameter `obj` to the type of `Object` that you are dealing with. For example, in the `Media` class, you must cast the `Object` obj to a `Media`, and then check the equality of the two objects' attributes as the above requirements (i.e. `id` for `Media`; `title` and `length` for `Track`). If the passing object is not an instance of `Media`, what happens?

Note: We can apply Release Flow here by creating a topic branch for the override of `equals()` method.

11. Polymorphism with toString() method

This exercise gives an illustration for polymorphism at behavior level.

Recall that for each type of media, we have implemented a `toString()` method that prints out the information of the object. When calling this method, depending on the type of object, corresponding `toString()` will be performed.

- Create an ArrayList of Media, then add some media (CD, DVD or Book) into the list.
- Iterate through the list and print out the information of the media by using toString() method. Observe what happens and explain in detail. Compare with the polymorphism examples in the previous exercises.

Sample code:

```
List<Media> mediae = new ArrayList<Media>();
// create some media here
// for example: cd, dvd, book
mediae.add(cd)
mediae.add(dvd)
mediae.add(book)
for (Media m : mediae)
     m.toString()
```

12. Sort media in the cart

As mentioned before, when seeing the current cart, the user can sort the items in the cart by title or by cost:

- Sort by title: the system displays all the medias in the alphabet sequence by title. In case they have the same title, the medias having the higher cost will be displayed first.
- Sort by cost: the system the system displays all the medias in decreasing cost order. In case they have the same cost, the medias will be ordered by title (alphabetical).

Here, we can use **`Comparator`** to allow multiple sorting ways of **`Media`:**

Note: The `Comparator` interface is a comparison function, which impose a total ordering on some

collection of objects. Comparators can be passed to a sort method (such as `Collections.sort()`) to allow precise control over the sort order.

Please open the Java docs to see the information of this interface.

- Create two classes of comparators, one for each type of ordering

```
public class MediaComparatorByCostTitle implements Comparator<Media>

public class MediaComparatorByTitleCost implements Comparator<Media>
```

- Implement the `compare()` method of each comparator class to reflect the ordering that we want, either by title then cost, or by cost then title. You may utilize the method `Comparator.thenComparing()` to sort using multiple fields.
- Add the comparators as attributes of the `Media` class:

```
public class Media {

    public static final Comparator<Media> COMPARE_BY_TITLE_COST =
            new MediaComparatorByTitleCost();
    public static final Comparator<Media> COMPARE_BY_COST_TITLE =
            new MediaComparatorByCostTitle();
```

- Pass the comparator into `Collections.sort`:

`java.util.Collection.sort(collection,Media.COMPARE_BY_TITLE_COST)`
or
`java.util.Collection.sort(collection, Media.COMPARE_BY_COST_TITLE)`

Question: Alternatively, to compare items in the cart, instead of using `Comparator`, we can use the `Comparable` interface and override the `compareTo()` method. You can refer to the Java docs to see the information of this interface.

Suppose we are taking this `Comparable` interface approach.

- What class should implement the `Comparable` interface?
- In those classes, how should you implement the `compareTo()` method be to reflect the ordering that we want?
- Can we have two ordering rules of the item (by title then cost and by cost then title) if we use this `Comparable` interface approach?
- Suppose the DVDs has a different ordering rule from the other media types, that is by title, then decreasing length, then cost. How would you modify your code to allow this?

13. Create a complete console application in the Aims class

In the **`main`** method of **`Aims`**, you will now implement a complete console application, by first create an instance of the **`Store`** class and then, provide a list of functionalities through a menu that the user can interact with. For the home interface, you will create the main menu as following:

```
public static void showMenu() {
            System.out.println("AIMS: ");
```

```
        System.out.println("--------------------------------");
        System.out.println("1. View store");
        System.out.println("2. Update store");
        System.out.println("3. See current cart");
        System.out.println("0. Exit");
        System.out.println("--------------------------------");
        System.out.println("Please choose a number: 0-1-2-3");
}
```

- From the main menu, if the user choses option "**View store**", the application will display all the items in the store, and a menu as following:

```
public static void storeMenu() {
        System.out.println("Options: ");
        System.out.println("--------------------------------");
        System.out.println("1. See a media's details");
        System.out.println("2. Add a media to cart");
        System.out.println("3. Play a media");
        System.out.println("4. See current cart");
        System.out.println("0. Exit");
        System.out.println("--------------------------------");
        System.out.println("Please choose a number: 0-1-2-3-4");
}
```

The option "**1. See a media's details**" should display the information of the media and ask the user if they wish to add that DVD to the current cart. If the media type is CD or DVD, the user can choose to play the media.

The option "**2. Add a media to cart**" will ask the user to enter the title of the media that he/she sees on the screen (the list of medias in store), then add the media to cart. Please remember to check the validity of the title. After adding a DVD to cart, the system will display the number of DVDs in the current cart.

The option "**3. Play a media**" will ask the same input from the user of option 2. You should again check the validity of the title.

- From the main menu, if the user choses option "**Update store**", the application will allow the user to add a media to or remove a media from the store
- From the main menu, if the user choses option "**See current cart**", the application will display the information of the current cart, and a menu as following:

```
public static void cartMenu() {
        System.out.println("Options: ");
        System.out.println("--------------------------------");
        System.out.println("1. Filter medias in cart");
        System.out.println("2. Sort medias in cart");
        System.out.println("3. Remove media from cart");
        System.out.println("4. Play a media");
        System.out.println("5. Place order");
        System.out.println("0. Exit");
        System.out.println("--------------------------------");
        System.out.println("Please choose a number: 0-1-2-3-4-5");
}
```

The “**1. Filter medias in cart”** option should allow the user to choose one of two filtering options: by id and by title.
The “**2. Sort medias in cart**” option should allow the user to choose one of two sorting option: by title or by cost.
Note: When the user chooses option “Place order”, the system is supposed to move on to the Delivery Information gathering & Payment step. However, for simplicity, within the scope of this lab course, when the user chooses this option, we only need to notify the user that an order is created and empty the current cart.