Object-Oriented Programming

Lab 03: Object-Oriented Techniques

In this lab, you will practice with:

- Re-organizing your project by creating packages to manage classes in Eclipse
- Debugging with Eclipse
- Practicing memory management with String and StringBuffer and other cases
- Java Inheritance mechanism
- Abstract class and interface
- Use the Collections framework (specifically, **`ArrayList`**)
- Polymorphism

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.
- **A week before the next class**: for this deadline, you should include the **source code** of all sections of this lab. Additionally, for the reading assignment, submit a **pdf** file of the reading assignment in the folder namely "**ReadingAssignment**". After you finish all your work in the "**master**" branch, please create a branch namely "**release/lab03**" of the valid repository to submit all your work.

Notice: 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 own 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. Import the existing project into the workspace of Eclipse

- Step 1: Open Eclipse
- Step 2: 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.

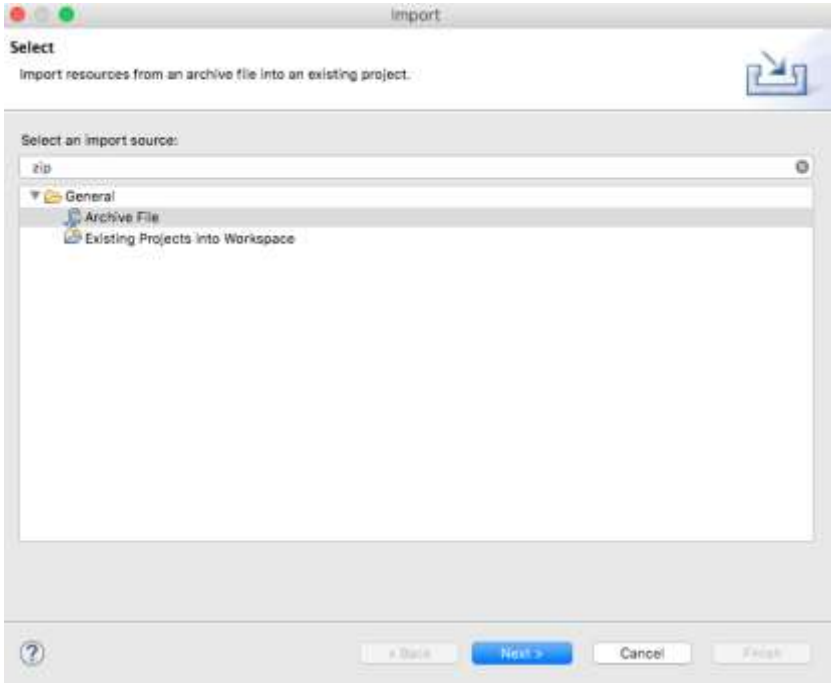

Figure 1. Import existing project

- Step 3: 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`**.

`Notice:` 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. Use debug run:

2.1. Debugging Java in Eclipse

- **Video: https://www.youtube.com/watch?v=9gAjIQc4bPU&t=8s**

Debugging is the routine process of locating and removing bugs, errors or abnormalities from programs. It's a must have skill for any Java developer because it helps to find subtle bugs that are not visible during code reviews or that only happen when a specific condition occurs. The Eclipse Java IDE provides many debugging tools and views grouped in the Debug Perspective to help you as a developer debug effectively and efficiently.

Debug run allows you to run a program interactively while watching the source code and the variables during the execution. A *breakpoint* in the source code specifies where the execution of the program

should stop during debugging. **Once the program is stopped you can investigate variables, change their content, etc.**

2.2. Example of debug run for the `swap` method of `TestPassingParameter`

2.2.1. Setting, deleting & deactivate breakpoints:

To set a breakpoint, place the cursor on the line that needs debugging, hold down Ctrl+Shift, and press B to enable a breakpoint. A blue dot in front of the line will appear (Figure 2). Alternatively, you can right-click in the left margin of the line in the Java editor and select Toggle Breakpoint. This is equivalent to double-clicking in the left margin of the line.

```
1
2  public class TestPassingParameter {
3
4      public static void main(String[] args) {
5          // TODO Auto-generated method stub
6          DigitalVideoDisc jungleDVD = new DigitalVideoDisc("Jungle");
7          DigitalVideoDisc cinderellaDVD = new DigitalVideoDisc("Cinderella");
8
9          swap(jungleDVD, cinderellaDVD);
10         System.out.println("jungle dvd title: "+jungleDVD.getTitle());
11         System.out.println("cinderella dvd title: "+cinderellaDVD.getTitle());
12
13         changeTitle(jungleDVD, cinderellaDVD.getTitle());
14         System.out.println("jungle dvd title: "+jungleDVD.getTitle());
15     }
16
17     public static void swap(Object o1, Object o2) {
18         Object tmp = o1;
19         o1 = o2;
20         o2 = tmp;
21     }
22
23     public static void changeTitle(DigitalVideoDisc dvd, String title) {
24         String oldTitle = dvd.getTitle();
25         dvd.setTitle(title);
26         dvd = new DigitalVideoDisc(oldTitle);
27     }
28
```

Figure 2. A breakpoint is set

To delete a breakpoint, toggle the breakpoint one more time. The blue dot in front of the line will disappear (Figure 3).

```
1
2  public class TestPassingParameter {
3
4      public static void main(String[] args) {
5          // TODO Auto-generated method stub
6          DigitalVideoDisc jungleDVD = new DigitalVideoDisc("Jungle");
7          DigitalVideoDisc cinderellaDVD = new DigitalVideoDisc("Cinderella");
8
9          swap(jungleDVD, cinderellaDVD);
10         System.out.println("jungle dvd title: "+jungleDVD.getTitle());
11         System.out.println("cinderella dvd title: "+cinderellaDVD.getTitle());
12
13         changeTitle(jungleDVD, cinderellaDVD.getTitle());
14         System.out.println("jungle dvd title: "+jungleDVD.getTitle());
15     }
16
17     public static void swap(Object o1, Object o2) {
18         Object tmp = o1;
19         o1 = o2;
20         o2 = tmp;
21     }
22
23     public static void changeTitle(DigitalVideoDisc dvd, String title) {
24         String oldTitle = dvd.getTitle();
25         dvd.setTitle(title);
26         dvd = new DigitalVideoDisc(oldTitle);
27     }
28
```

Figure 3 The breakpoint is deleted

To deactivate the breakpoint, navigate to the Breakpoints View and uncheck the tick mark next to the breakpoint you want to deactivate (Figure 4). **The program will only stop at activated breakpoints**.

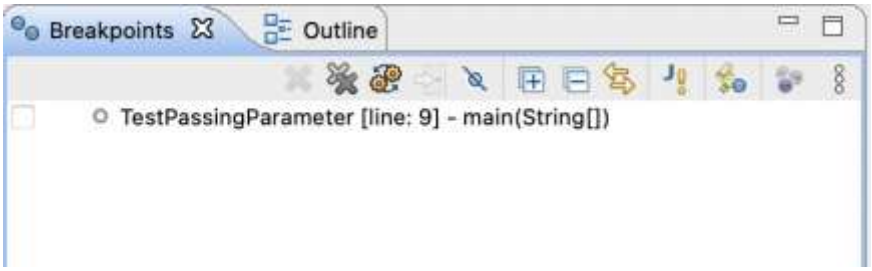

Figure 4. Deactivated breakpoint in Breakpoints View

For this example, we will need to keep this breakpoint, so make sure to set the breakpoint again after practicing with deleting/deactivating it before moving to the next section.

2.2.2. Run in Debug mode:

Select a Java file with a main method that contains the code that you need to debug from the Project Explorer. In this example, we choose the **TestPassingParameter.java** file. Right click and choose Debug As > Java Application (Figure 5).

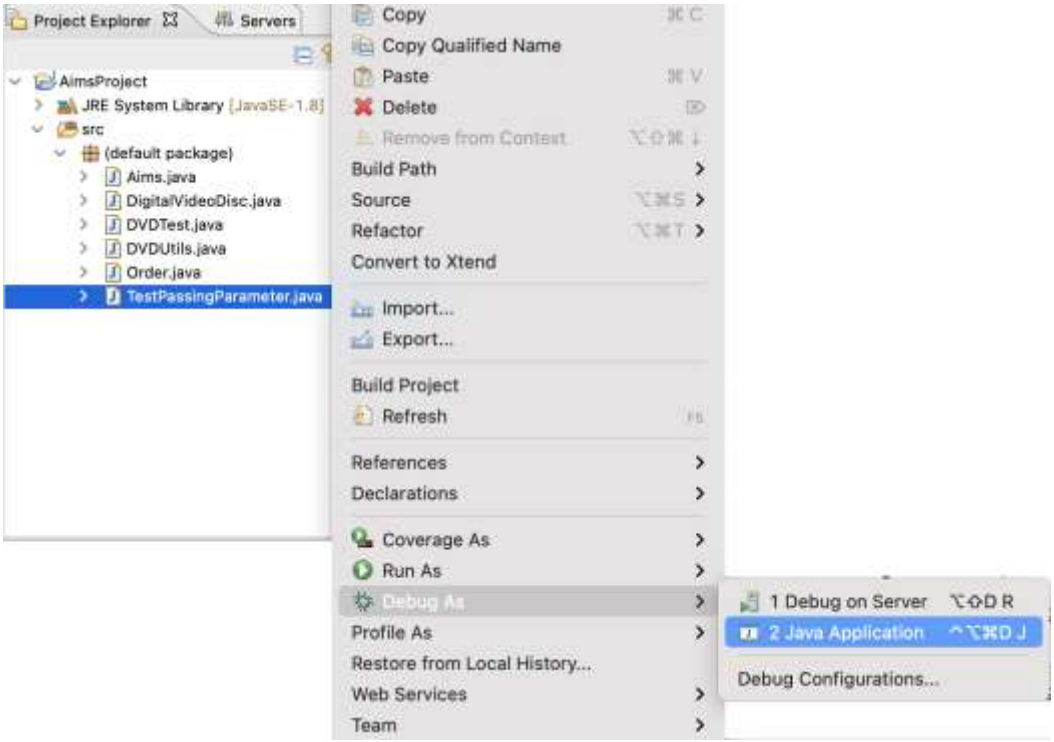

Figure 5. Run Debug from a class

Alternatively, you can select the project root node in the Project Explorer and click the debug icon in the Eclipse toolbar (Figure 6)

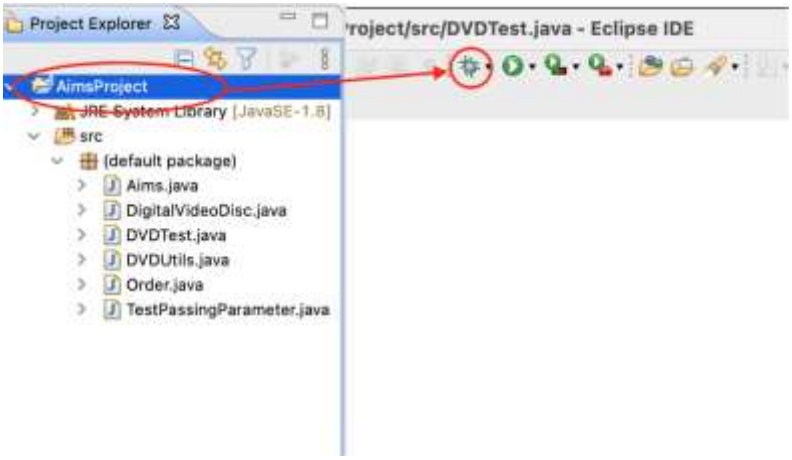

Figure 6. Run debug from a project

The application will now be started with Eclipse attached as a debugger. Confirm to open the Debug Perspective.

2.2.3. Step Into, Step Over, Step Return, Resume:

In the Debug Perspective, you can observe the Step Into/Over/Return & Resume/Terminate buttons on the toolbar as in Figure 7.

Figure 7. Stepping Commands on the Toolbar in Debug Perspective

With debugger options, the difference between "Step into" and "Step over" is only noticeable if you run into a function call:

- "Step into" (F5) means that the debugger steps into the function
- "Step over" (F6) just moves the debugger to the next line in the same Java action

With "Step Return" (pressing F7), you can instruct the debugger to leave the function; this is basically the opposite of "Step into."

Clicking "Resume" (F8) instructs the debugger to continue until it reaches another breakpoint.

For this example, we need to see the execution of the **`swap`** function, so we choose Step Into. The debugger will step into the implementation of the **`swap`** function in line 18 (Figure 8).

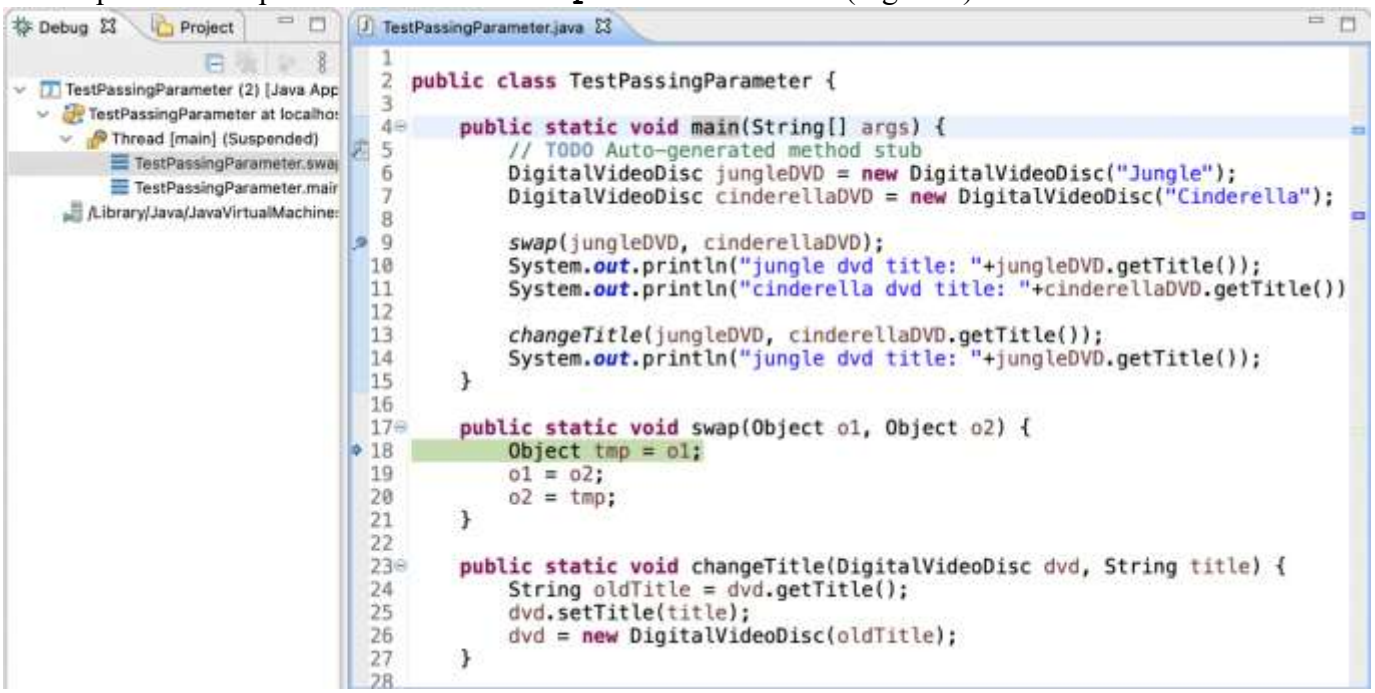

Figure 8. Step into swap function

2.2.4. Investigate value of variables:

We can observe the value of variables & expression in the Variables/Expression View. You can also add a permanent watch on an expression/variable that will then be shown in the Expressions view when debugging is on.

Alternatively, place your cursor on any of the variables in the Java action to see its value in a pop-up window.

Open the Variable Perspective and observe the values of variables **`o1`** & **`o2`** (Figure 9). You can click the drop-down arrow to investigate attributes of variables.

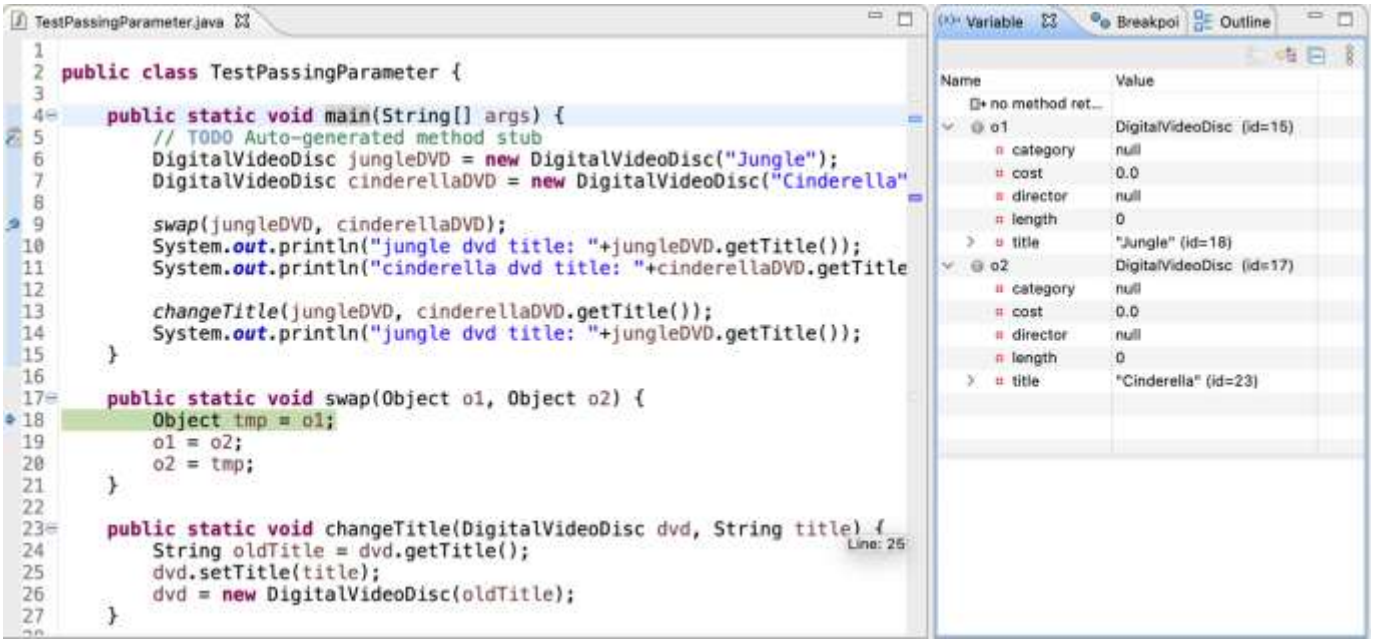

Figure 9. Variables shown in Variable View

Click Step Over and watch the change in the value of variables **`o1, o2`** & **`tmp`**. Repeat this until the end of the **`swap`** function (Figure 10, Figure 11, Figure 12).

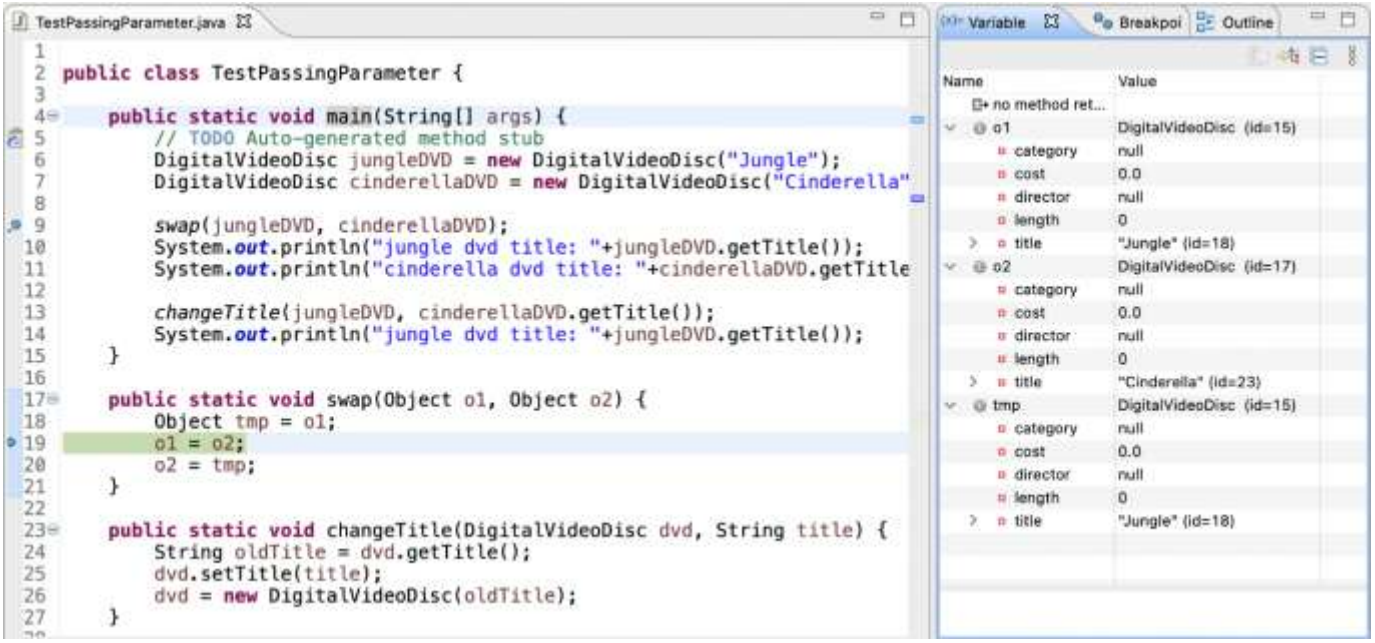

Figure 10. Step over line 18 of swap function

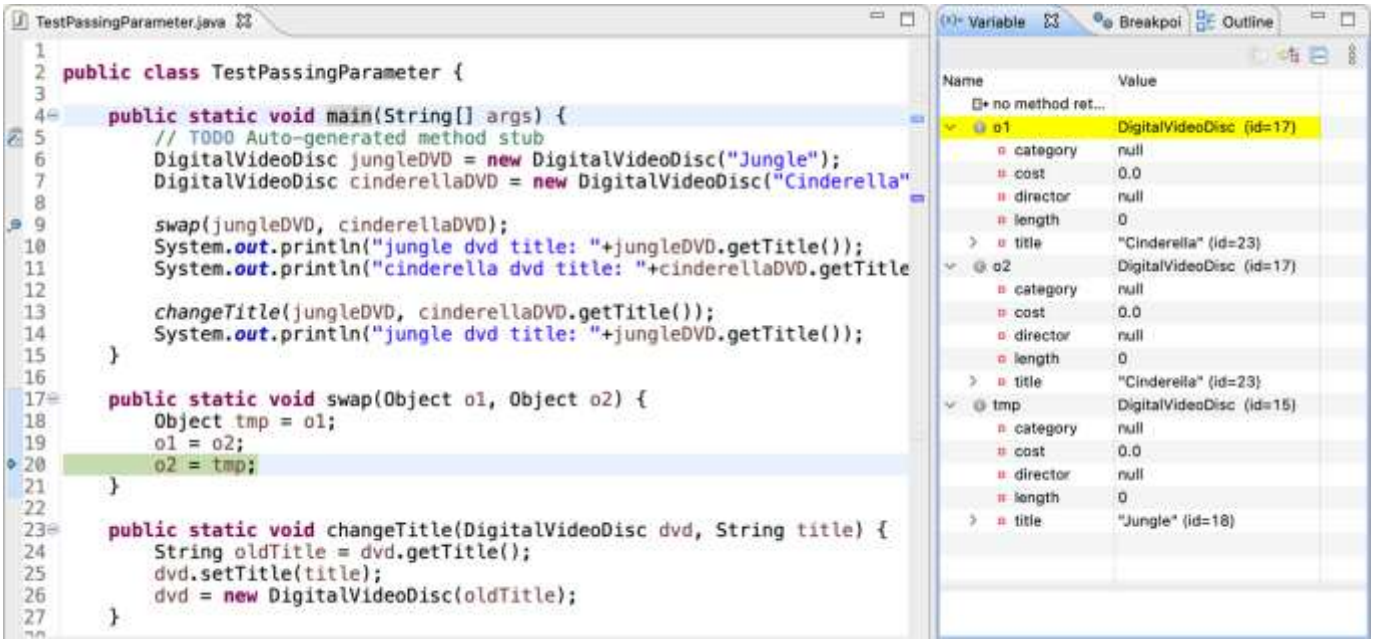

Figure 11. Step over line 19 of swap function

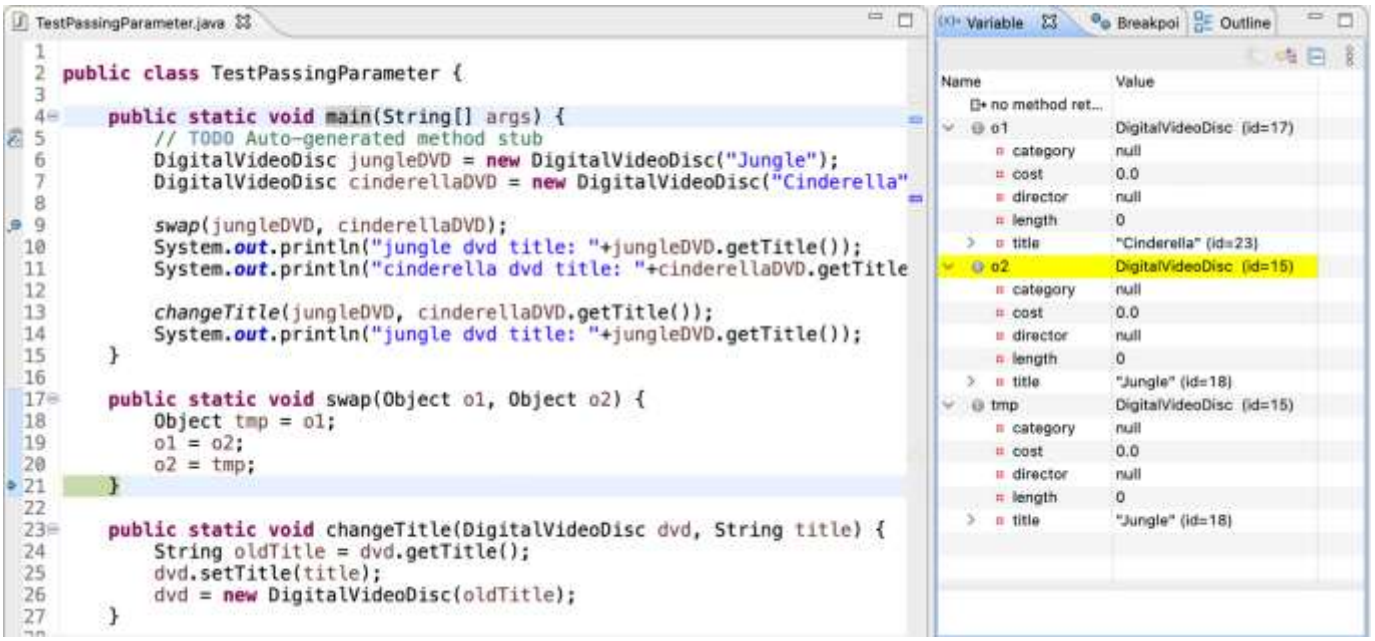

Figure 12. Step over line 20 of swap function

2.2.5. Change value of variables:

In the Variable Perspective, you can also change the value of the variable while debugging. Click Step Return so the debugger returns from the **`swap`** function back to the line after the call to it. (Figure 13)

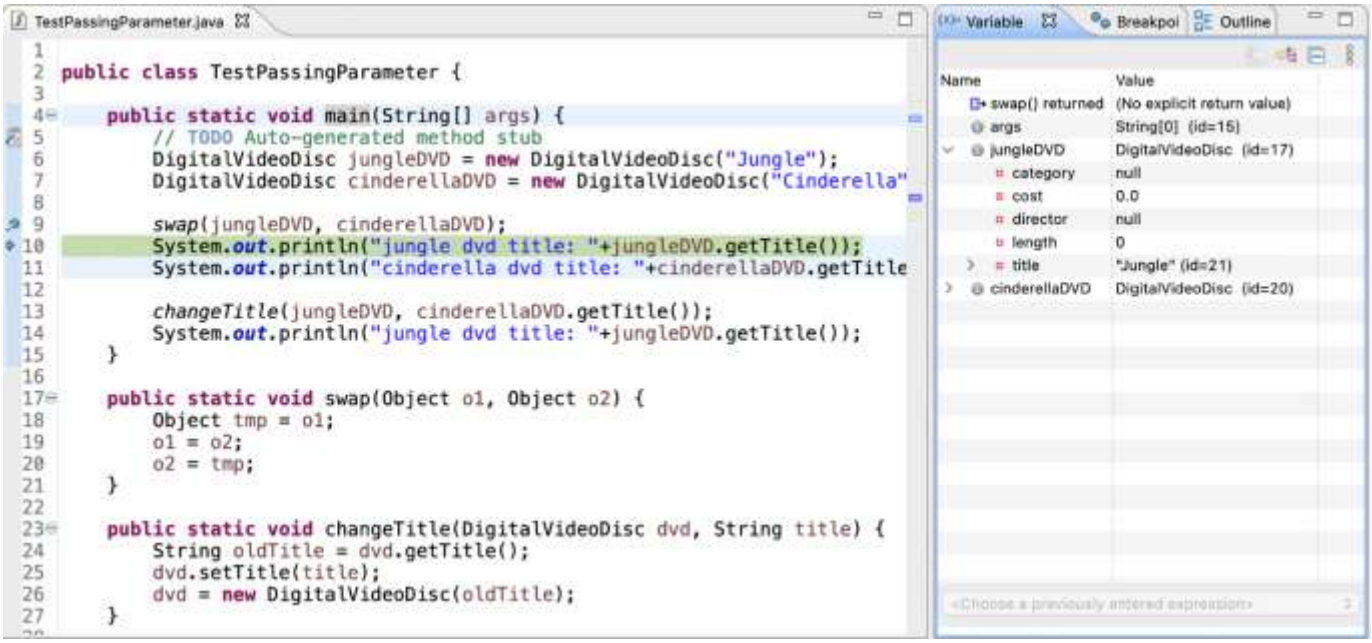

Figure 13. Step return to main function

The variable **`jungleDVD`** still has a title attribute with value “Jungle”. You can change this value by clicking on it and change it to “abc”, for example (Figure 14).

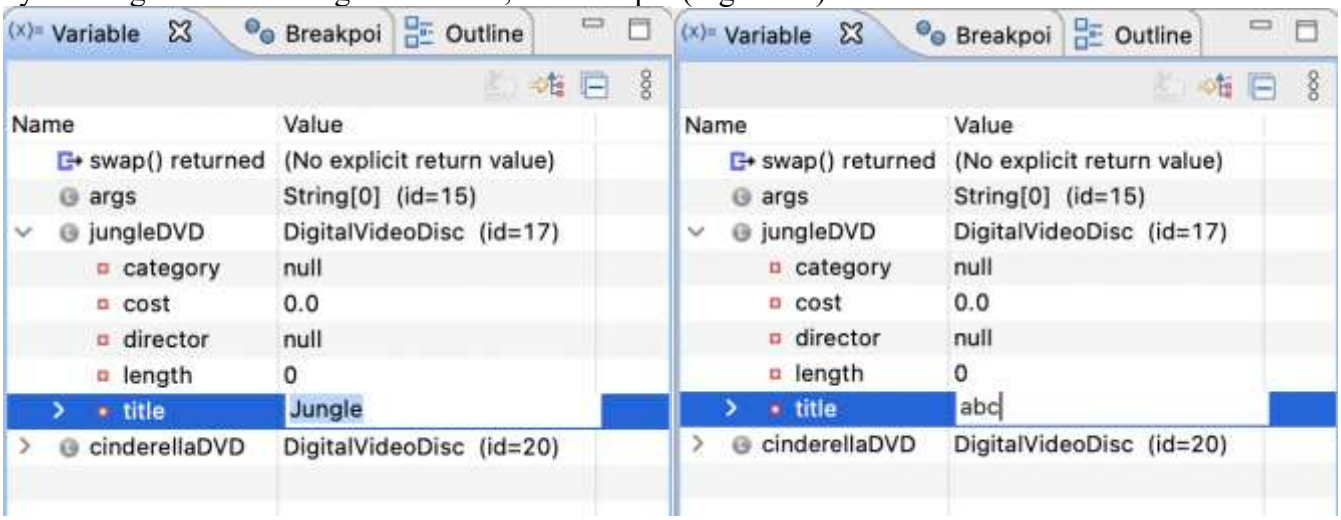

Figure 14. Change title of jungleDVD

Click Step Over and see the result in the output in the Console (Figure 15)

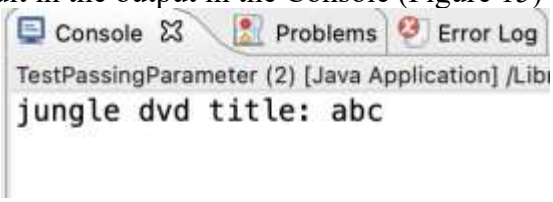

Figure 15. Results(2)

3. Re-organize your project

- Rename project, use packages and reorganize all hands-on labs and exercises from the Lab01 up to now.
    - For renaming or moving an item (i.e. a project, a class, a variable…), right click to the item, choose Refactor -> Rename/Move and follow the steps.

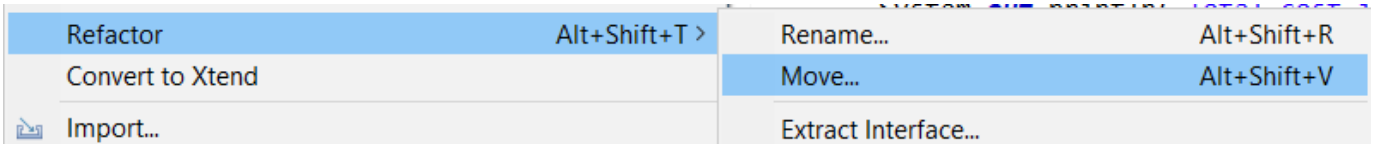

Figure 16. Refactoring

    - For creating a package, right click to the project (or go to menu File) and choose New -> Package. Type the full path of the package including parent packages, separated by a dot.
- Keep the text file for answering questions in the lab, the "**Requirement**", "**Design**" and "**Reading Assignment**" folders should be moved inside the root folder of **AimsProject**, next to its **src/** and **bin/** folder.
- Your **structure of your labs** should be at least as below. You can create sub-packages for more efficiently organizing your classes in both projects and all listing packages. All the exercises of lab01 should be put in the corresponding package of one project - the OtherProjects project.
- This is an example of how to refactor your project's folder structure if you are a DS-AI student.

```
+ AimsProject
    hust.soict.dsai.aims.disc.DigitalVideoDisc
    hust.soict.dsai.aims.cart.Cart
    hust.soict.dsai.aims.Aims
    hust.soict.dsai.test.cart.CartTest
    hust.soict.dsai.test.disc.TestPassingParameter
+ OtherProjects
    hust.soict.dsai.lab01
    hust.soict.dsai.lab02
    hust.soict.dsai.lab03
```

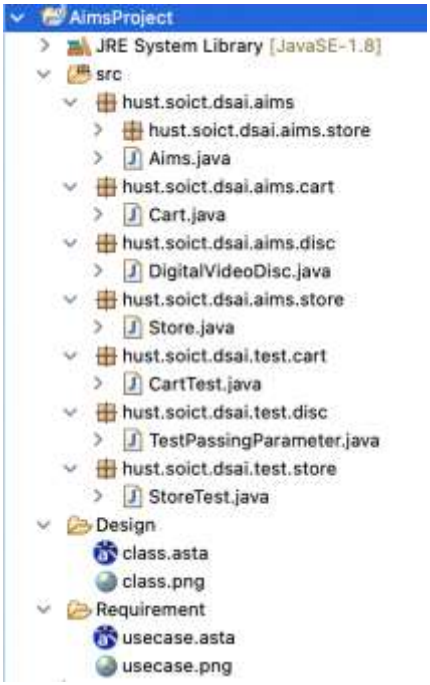

Figure 17. Recommended Structure for DS-AI

Notice: From this section onwards, it is assumed that you are a DS-AI student, so your folder structure will contain the **"dsai"** package. If you are an HEDSPI or ICT student, you should replace the **"dsai"** string with **"hedspi"** or **"globalict"**.

4. Update the `Cart` class and `CartTest` class

Write new methods to implement the following functions:

- Create a new method to print the list of ordered items of a cart, the price of each item, and the total price. Format the outline as below:

  **********CART**********
  Ordered Items:
  1. DVD - [Title] - [category] - [Director] - [Length]: [Price] $
  2. DVD - [Title] - ...
  Total cost: [total cost]
  ****

  Suggestion: Write a **toString()** method for the **DigitalVideoDisc** class. What should be the return type of this method?
- Search for DVDs in the cart by ID and display the search results. Make sure to notify the user if no match is found.
- Search for DVDs in the cart by title and print the results. Make sure to notify the user if no match is found. ***Refer to the problem statement in Lab02 for the matching rule(Section 4).***

  Suggestion: write a `boolean isMatch(String title)` method in the `DigitalVideoDisc` which finds out if the corresponding disk is a match given the title.
- In the `CartTest` class, write codes to test all methods you have written in this exercise. You should create sample DVDs and carts, like in this code snippet:

```
public class CartTest {
    public static void main(String[] args) {
            //Create a new cart
            Cart cart = new Cart();

            //Create new dvd objects and add them to the cart
            DigitalVideoDisc dvd1 = new DigitalVideoDisc("The Lion King",
                    "Animation", "Roger Allers", 87, 19.95f);
            cart.addDigitalVideoDisc(dvd1);

            DigitalVideoDisc dvd2 = new DigitalVideoDisc("Star Wars",
                    "Science Fiction", "George Lucas", 87, 24.95f);
            cart.addDigitalVideoDisc(dvd2);

            DigitalVideoDisc dvd3 = new DigitalVideoDisc("Aladin",
                    "Animation", 18.99f);
            cart.addDigitalVideoDisc(dvd3);

            //Test the print method
            cart.print();
            //To-do: Test the search methods here

    }
}
```

Figure 18. Code snippet for CartTest

5. Implement the **Store** class

- Create a **Store** class inside the package: **hust.soict.dsai.aims.store**, which contains one attribute **itemsInStore[]** – an array of DVDs available in the store.
- To add and remove DVDs from the store, implement two methods called **addDVD** and **removeDVD**
- Test these two methods in the **StoreTest** class.

As a result, you should have two classes: the **Store** class and the **StoreTest** class inside the **hust.soict.dsai.aims.store** package.

6. **String**, **StringBuilder** and **StringBuffer**

- In the **OtherProjects** project, create a new package **hust.soict.dsai.garbage**. We work with this package in this exercise.
- Create a new class **ConcatenationInLoops** to test the processing time to construct **String** using **+** operator, **StringBuffer** and **StringBuilder**.

```java
public class ConcatenationInLoops {
    public static void main(String[] args) {
        Random r = new Random(123);
        long start = System.currentTimeMillis();
        String s = "";
        for (int i = 0; i < 65536; i++)
            s += r.nextInt(2);
        System.out.println(System.currentTimeMillis() - start);  // This prints roughly 4500.

        r = new Random(123);
        start = System.currentTimeMillis();
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < 65536; i++)
            sb.append(r.nextInt(2));
        s = sb.toString();
        System.out.println(System.currentTimeMillis() - start);  // This prints 5.
    }
}
```

Figure 19. ConcatenationInLoops

Notice: For more information on String concatenation, please access this link: https://redfin.engineering/java-string-concatenation-which-way-is-best-8f590a7d22a8.

- Create a new class **`GarbageCreator`**. Create "garbage" as much as possible and observe when you run a program (it should let the program hang or even stop working when there is too much "garbage"). Write another class **`NoGarbage`** to solve the problem.

Some suggestions:

- Read a text/binary file to a **`String`** without using **`StringBuffer`** to concatenate String (only use **`+`** operator). Observe and capture your screen when you choose a very long file
- Improve the code using **`StringBuffer`.**

The following piece of code is a suggestion for your implementation

```java
        String filename = "test.exe"; // test.exe is the name or path to an executable file
        byte[] inputBytes = { 0 };
        long startTime, endTime;

        inputBytes = Files.readAllBytes(Paths.get(filename));
        startTime = System.currentTimeMillis();
        String outputString = "";
        for (byte b : inputBytes) {
            outputString += (char)b;
        }
        endTime = System.currentTimeMillis();
        System.out.println(endTime - startTime);
```

Figure 20. Sample code for GarbageCreator

Change the code in line 14-17 above to use StringBuffer instead of "+" operator to build string and observe result

```
14          StringBuilder outputStringBuilder = new StringBuilder();
15          for (byte b : inputBytes) {
16              outputStringBuilder.append((char)b);
17          }
```

Figure 21. New code using StringBuffet

7. Additional requirements of AIMS

Starting from this lab, you extend the AIMS system that you created in the previous exercises to allow customers to order 2 new types of media: books and CDs.

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 price. Additionally, each track is unique in a CD with its own title and length. The length of a CD is the 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 price 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.

8. Implementation of 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 22. 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

9. Implementation of 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 an abstract class called **`Media`** which contains these fields and their associated get and set methods.

Please follow these steps to 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: **int id**, **String title**, **String category** and **float cost**
- 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.

10. Implementation of the **CompactDisc** class

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

*10.1. 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.

*10.2. 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

*10.3. Update the **CompactDisc** class*

- Add 2 fields to this class:
  - a **String** as **artist**

        - an **ArrayList** of **Track** as **tracks**
    - Make all these fields **private.** Create a public **getter** method for only **artists**.
    - 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

11. Implementation of 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**() method 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

12. 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:
    - Step 1: From the **Package Explorer** view, expand the project
    - Step 2: Double-click **`Cart.java`** to open it in the editor
    - Step 3: In the **Outline** view, select the **`itemsOrdered`** array and the methods **`addDigitalVideoDisc`**() and **`removeDigitalVideoDisc`**() and hit the **Delete** key
    - Step 4: Click **Yes** when prompted to confirm the deletion
    - Step 5: 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
    - Step 6: 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

Notice: 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

13. 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`**()

14. Constructors of whole classes and parent classes

- **Write constructors for parent and child classes. Remove redundant setter methods if any**
- **Which classes are aggregates of other classes**? Checking all constructors of whole classes if they initialize for their parts?

15. 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 conditions 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 uses `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. `title` for `Media`; `title` and `length` for `Track`). If the passing object is not an instance of `Media`, what happens?

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

16. 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.

Sample code:

```
16 List<Media> mediae = new ArrayList<Media>();
17
18 // create some media here
19 // for example: cd, dvd, book
20
21 mediae.add(cd);
22 mediae.add(dvd);
23 mediae.add(book);
24
25 for(Media m: mediae) {
26     System.out.println(m.toString());
27 }
```

Figure 23. Polymorphism sample code

Reading Assignment: Please answer three questions below:

- What are the advantages of Polymorphism?
- How is Inheritance useful to achieve Polymorphism in Java?
- What are the differences between Polymorphism and Inheritance in Java?

17. 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 media having the higher cost will be displayed first.
- Sort by cost: the system displays all the media in decreasing cost order. In case they have the same cost, the media 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 imposes 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 the `Comparator` class I have mentioned, you can use the `Comparable` interface[1] 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 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 have 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?

18. 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, providing 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 chooses option "**View store**", the application will display all the items in the store, and a menu as following:

[1] Comparable (Java Platform SE 8) (oracle.com)

```
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. Back");
        System.out.println("--------------------------------");
        System.out.println("Please choose a number: 0-1-2-3-4");
}
```

- The option "**See a media's details**" will ask the user to enter the title of the media and display the information of that media. Please remember to check the validity of the title. Under the information display, the system also shows the following menu (note that the "**Play**" option is only available to CD and DVD type.

```
public static void mediaDetailsMenu() {
        System.out.println("Options: ");
        System.out.println("--------------------------------");
        System.out.println("1. Add to cart");
        System.out.println("2. Play");
        System.out.println("0. Back");
        System.out.println("--------------------------------");
        System.out.println("Please choose a number: 0-1-2");
}
```

- The option "**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 media in store), then add the media to cart. Please remember to check the validity of the title. After adding a DVD to the cart, the system will display the number of DVDs in the current cart.
- The option "**Play a media**" will ask the same input from the user as option 2. You should again check the validity of the title.

- From the main menu, if the user chooses 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 chooses 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 media in cart");
        System.out.println("2. Sort media 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. Back");
        System.out.println("--------------------------------");
        System.out.println("Please choose a number: 0-1-2-3-4-5");
```

```
}
```

- The "**Filter media in cart"** option should allow the user to choose one of two filtering options: by id and by title.
- The "**Sort media in cart**" option should allow the user to choose one of two sorting options: 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.

- 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/lab03,`** push the diagram and its image, and then merge with master