Object-Oriented Programming

Lecturer:
Teaching Assistant:

Lab 03: Basic Object-Oriented Techniques

In this lab, you will practice with:
- Working with Release workflow
- Method overloading
- Parameter passing
- Classifier member vs. Instance member
- Practicing memory management with String and StringBuffer and other cases
- Debugging with Eclipse
- Re-organizing your project by creating packages to manage classes in Eclipse

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/lab03**" 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 write down answers for all questions into a text file named "**answers.txt**" and submit it within your repository.

1. Branch your repository

Day after day, your repository becomes more and more sophisticated, which makes your codes harder to manage. Luckily, a Git workflow can help you tackle this. A Git workflow is a **recipe for how to use Git** to control source code in a consistent and productive manner. Release Flow[1] is a lightweight but effective Git workflow that helps teams cooperate with a large size and regardless of technical expertise. Refer to the **Release-Flow-Guidelines.pdf** file for a more detailed guide.

Applying Release Flow is required from this lab forward.

However, we would use a modified version of Release Flow for simplicity.
- We can create as many branches as we need.
- We name branches with meaningful names. See Table 1-Branching policy.

---

[1] https://docs.microsoft.com/en-us/azure/devops/repos/git/git-branching-guidance?view=azure-devops

- We had better **keep branches as close to `master` as possible**; otherwise, we could face merge hell.
- Generally, when we merge a branch with its origin, that branch has been history. We usually do not touch it a second time.
- **We must strictly follow the policy for release branch. Others are flexible.**

Branch	Naming convention	Origin	Merge to	Purpose
`feature` *or* `topic`	+ `feature/feature-name` + `feature/feature-area/feature-name` + `topic/description`	`master`	`master`	Add **a** new feature or **a** topic
`bugfix`	`bugfix/description`	`master`	`master`	Fix **a** bug
		`feature`	`feature`	
`hotfix`	`hotfix/description`	`release`	`release & master[1]`	Fix **a** bug in a **submitted assignment after deadline**
`refactor`	`refactor/description`	`master`	`master`	Refactor
		`feature`	`feature`	
`release`	`release/labXX`	`master`	none	Submit assignment [2]

Table 1: Branching policy

[1] If we want to update your solutions within a week after the deadline, we could make a new hotfix branch (e.g., `hotfix/stop-the-world`). Then we merge the `hotfix` branch with `master` and with `release` branch for last submitted assignment (e.g., `release/lab05`). In case we already create a `release` branch for the current week assignment (e.g., `release/lab06`), we could merge the `hotfix` branch with the current `release` branch **if need be,** or we can delete and then recreate current release branch.

[2] **Latest versions of projects in release branch serve as the submitted assignment**

Let's use Release Flow as our Git workflow and apply it to refactor our repositories.

Step 1: Create new branch in our local repository. We create a new branch `refactor/apply-release-flow` from our `master` branch.

Step 2: Make our changes, test them, and push them. We move the latest versions of all our latest file from previous labs such that they are under the master branch directly.

See https://www.atlassian.com/git/tutorials/undoing-changes to undo changes in case of problems. To improve commit message, see https://thoughtbot.com/blog/5-useful-tips-for-a-better-commit-message.

Step 3: Make a pull request for reviews from our teammates[2]. We **skip** this step since we are **solo** in this repository. We, however, had better never omit this step when we work as a team.

Step 4: Merge branches. Merge the new branch `refactor/apply-release-flow` into `master` branch.
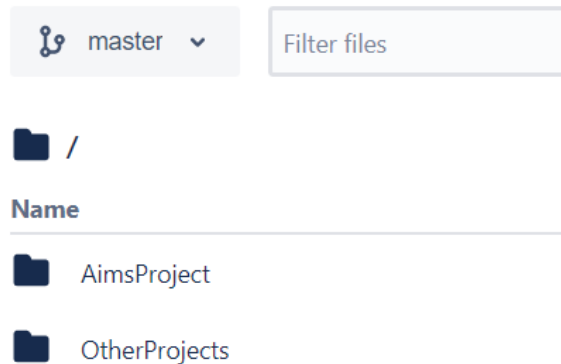
The result is shown in the following figure.

---

[2] https://www.atlassian.com/git/tutorials/making-a-pull-request

Figure 1. Merging result

Hints:

Typical steps for a new branch:

☐ Create and switch to a new branch (e.g. abc) in the local repo: **git checkout -b abc**
☐ Make modification in the local repo
☐ Commit the change in the local repo: **git commit -m "What you had change"**
☐ Create a new branch (e.g. abc) in the remote repo (GitHub through GUI)
☐ Push the local branch to the remote branch: **git push origin abc**
☐ Merge the remote branch (e.g. abc) to the master branch (GitHub through GUI)

After completing all the tasks of that week, and merge all branches into master branch, you should create a release/labxx branch from the master in the remote repo (GitHub).

For example, in the lab03, there may be 9 main tasks. So, one possible way to apply release flow is to create 9 branches:

- Create a branch **refactor/apply-release-flow** for refactoring the repository following the Release Flow
- Create a branch **topic/method-overloading** for the exercise on method overloading
- Create a branch **topic/passing-parameter** for the exercise where you investigate on Java's parameter passing
- Create a branch **topic/class-members** for the exercise where you practice with classifier member and instance member
- Create a branch **feature/print-cart** for the implementation of the print items in cart feature
- Create a branch **feature/search-cart** for the implementation of the search items in cart feature
- Create a branch **topic/store** for the implementation of the class `Store`
- Create a branch **refactor/packages** for refactoring the projects in your repository using packages
- Create a branch **topic/memory-management-string** for the `String`, `StringBufer` & `StringBuilder` exercise

Refer to the demonstration of Release Flow in the last section of this lab for more detailed guide.

2. Working with method overloading

Method overloading allows different methods to have the **same name** but different signatures where signature can differ by **number** of input parameters or **type** of input parameter(s) or **both.**

2.1 Overloading by differing types of parameter

- Open Eclipse

- Open the JavaProject named "`AimsProject`" that you have created in the previous lab.

- Open the class `Cart.java`: you will overload the method **`addDigitalVideoDisc`** you created last time.

+ The current method has one input parameter of class **DigitalVideoDisc**

+ You will create a new method that has the same name but with different type of parameter.

addDigitalVideoDisc(DigitalVideoDisc [] dvdList)

This method will add a list of DVDs to the current cart.

+ Try to add a method **addDigitalVideoDisc** which allows to pass an arbitrary number of arguments for dvd. Compare to an array parameter. What do you prefer in this case?

2.2. Overloading by differing the number of parameters

- Continuing focus on the `Cart` class

- Create new method named `addDigitalVideoDisc`

+ The signature of this method has two parameters as following:

addDigitalVideoDisc(DigitalVideoDisc dvd1,DigitalVideoDisc dvd2)

3. Passing parameter

- Question: ***Is JAVA a Pass by Value or a Pass by Reference programming language?***

First of all, we recall what is meant by **pass by value** or **pass by reference**.

- Pass by value: The method parameter values are **copied** to another variable and then the copied object is passed to the method. That's why it's called pass by value
- Pass by reference: An alias or reference to the actual parameter is passed to the method. That's why it's called pass by reference.

Now, you will practice with the **DigitalVideoDisc** class to test how JAVA passes parameters. For this exercise, you will need to temporarily add a setter for the attribute title of the DigitalVideoDisc class.

Create a new class named **TestPassingParameter** in the current project

- Check the option for generating the main method in this class like in Figure 2
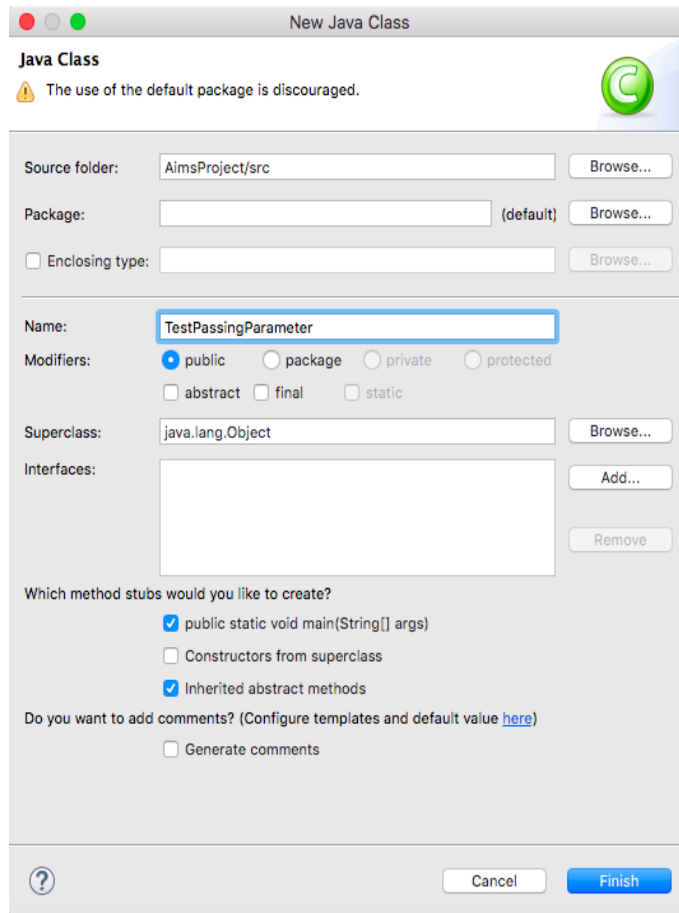
Figure 2. Create TestPassingParameter by Eclipse

In the **main()** method of the class, typing the code below in Figure 3:

```java
public class TestPassingParameter {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        DigitalVideoDisc jungleDVD = new DigitalVideoDisc("Jungle");
        DigitalVideoDisc cinderellaDVD = new DigitalVideoDisc("Cinderella");

        swap(jungleDVD, cinderellaDVD);
        System.out.println("jungle dvd title: " + jungleDVD.getTitle());
        System.out.println("cinderella dvd title: " + cinderellaDVD.getTitle());

        changeTitle(jungleDVD, cinderellaDVD.getTitle());
        System.out.println("jungle dvd title: " + jungleDVD.getTitle());
    }

    public static void swap(Object o1, Object o2) {
        Object tmp = o1;
        o1 = o2;
        o2 = tmp;
    }

    public static void changeTitle(DigitalVideoDisc dvd, String title) {
        String oldTitle = dvd.getTitle();
        dvd.setTitle(title);
        dvd = new DigitalVideoDisc(oldTitle);
    }
}
```
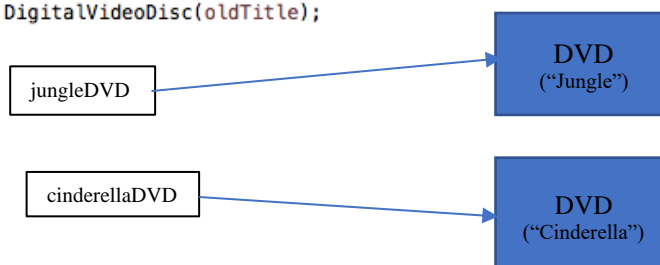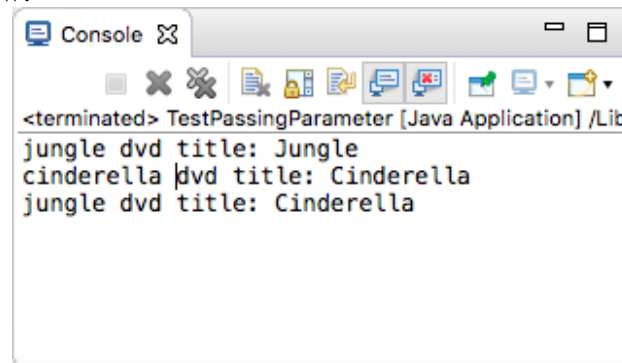
Figure 3. Source code of TestPassingParameter.

The result in console is below:

Figure 4. Results(1)

To test whether a programming language is passing by value or passing by reference, we usually use the **swap** method. This method aims to swap an object to another object.

- After the call of **swap(jungleDVD, cinderellaDVD)** why does the title of these two objects still remain?
- After the call of **changeTitle(jungleDVD, cinderellaDVD.getTitle())** why is the title of the JungleDVD changed?

After finding the answers to these above questions, you will understand that JAVA is always a pass by value programming language.
Please write a swap() method that can correctly swap the two objects.

4. Use debug run:

4.1. *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.**

4.2. *Example of debug run for the* **swap** *method of* **TestPassingParameter**

4.2.1. 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 5). 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.

```java
public class TestPassingParameter {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        DigitalVideoDisc jungleDVD = new DigitalVideoDisc("Jungle");
        DigitalVideoDisc cinderellaDVD = new DigitalVideoDisc("Cinderella");

        swap(jungleDVD, cinderellaDVD);
        System.out.println("jungle dvd title: "+jungleDVD.getTitle());
        System.out.println("cinderella dvd title: "+cinderellaDVD.getTitle());

        changeTitle(jungleDVD, cinderellaDVD.getTitle());
        System.out.println("jungle dvd title: "+jungleDVD.getTitle());
    }

    public static void swap(Object o1, Object o2) {
        Object tmp = o1;
        o1 = o2;
        o2 = tmp;
    }

    public static void changeTitle(DigitalVideoDisc dvd, String title) {
        String oldTitle = dvd.getTitle();
        dvd.setTitle(title);
        dvd = new DigitalVideoDisc(oldTitle);
    }
```

Figure 5. 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 6).

```java
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 6. 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 7). **The program will only stop at activated breakpoints**.
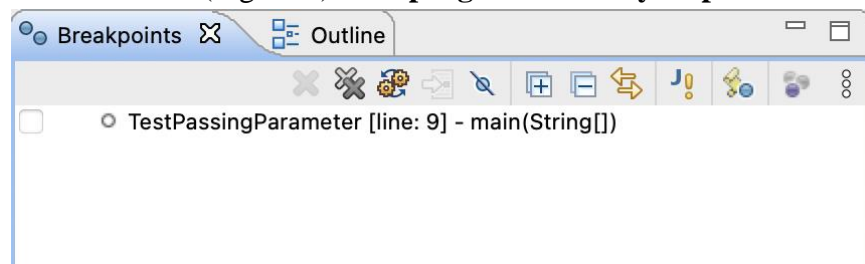

Figure 7. Deactivated breakpoint in Breakpoints View

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.

4.2.2.   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 8).
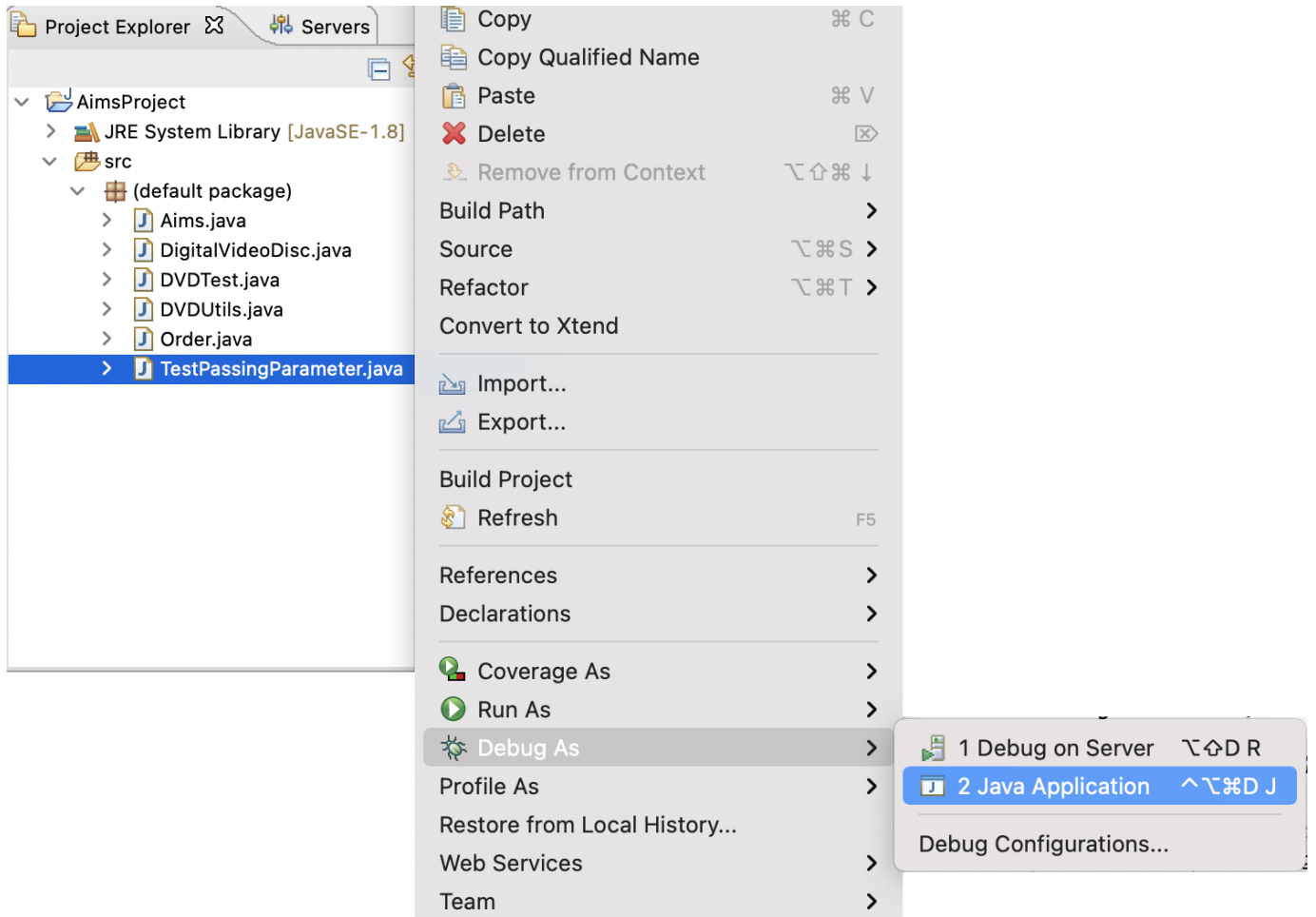
Figure 8. Run Debug from a class

Alternatively, you can select the project root node in the Project Explorer and click the debug icon in the Eclipse toolbar (Figure 9)
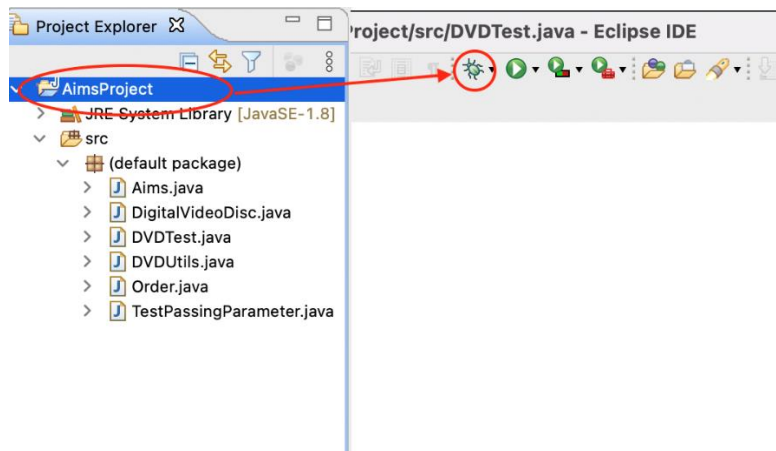

Figure 9. Run debug from a project

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

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

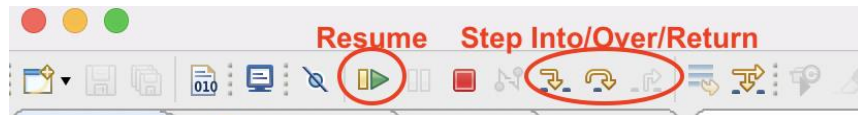- In the Debug Perspective, you can observe the Step Into/Over/Return & Resume/Terminate buttons on the toolbar as in Figure 10.

Figure 10. 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:

o "Step into" (F5) means that the debugger steps into the function

o "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 11).
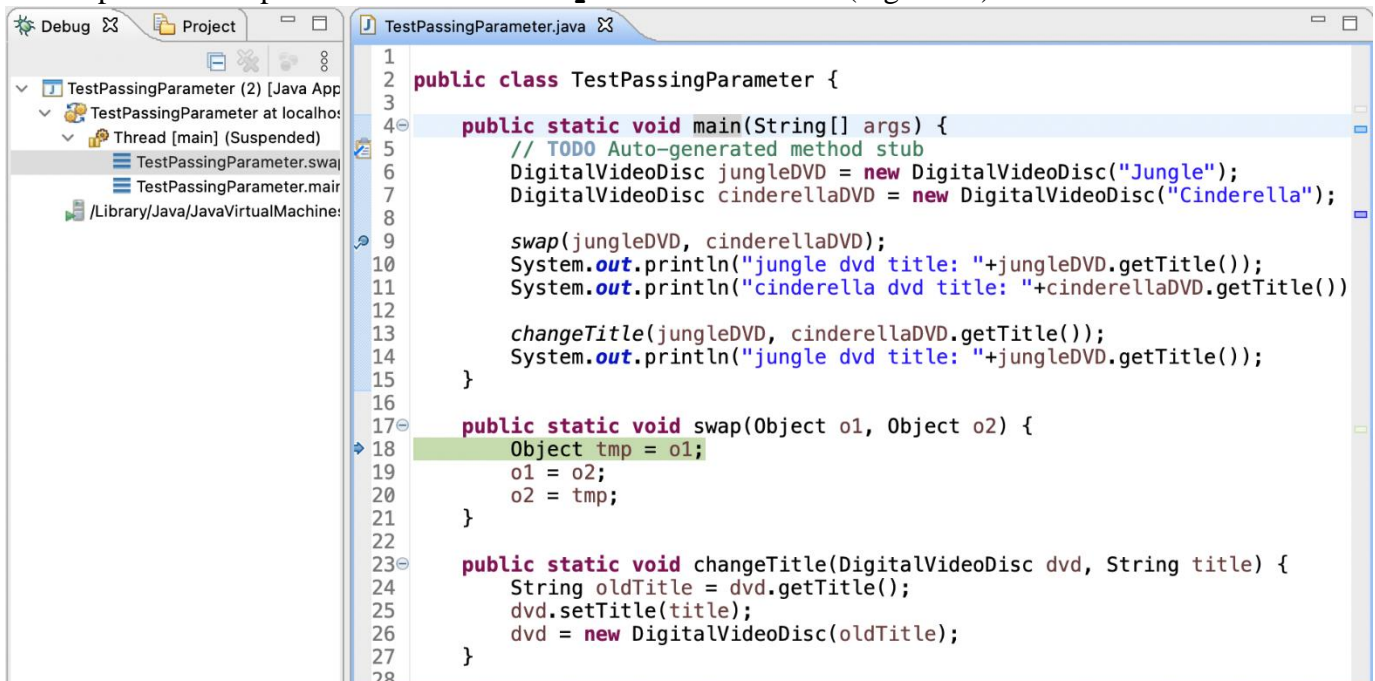

Figure 11. Step into swap function

4.2.4. 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 12). You can click the drop-down arrow to investigate attributes of variables.
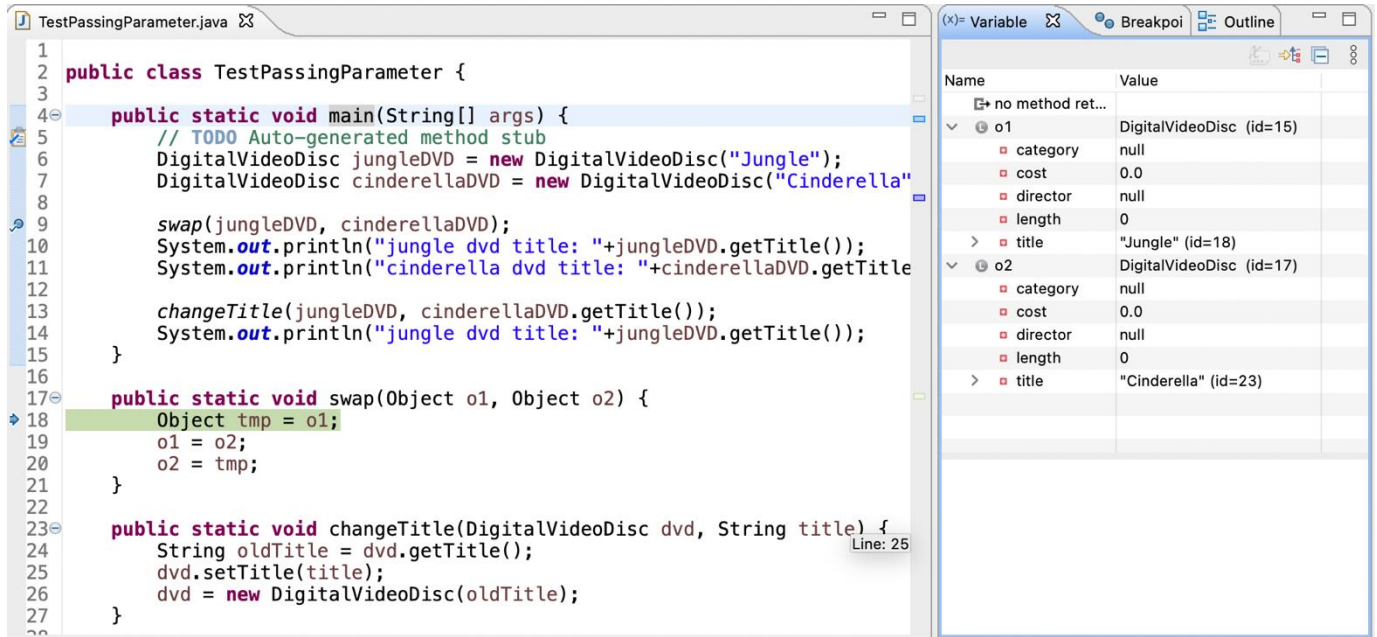
10

Figure 12. 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 13, Figure 14, Figure 15).
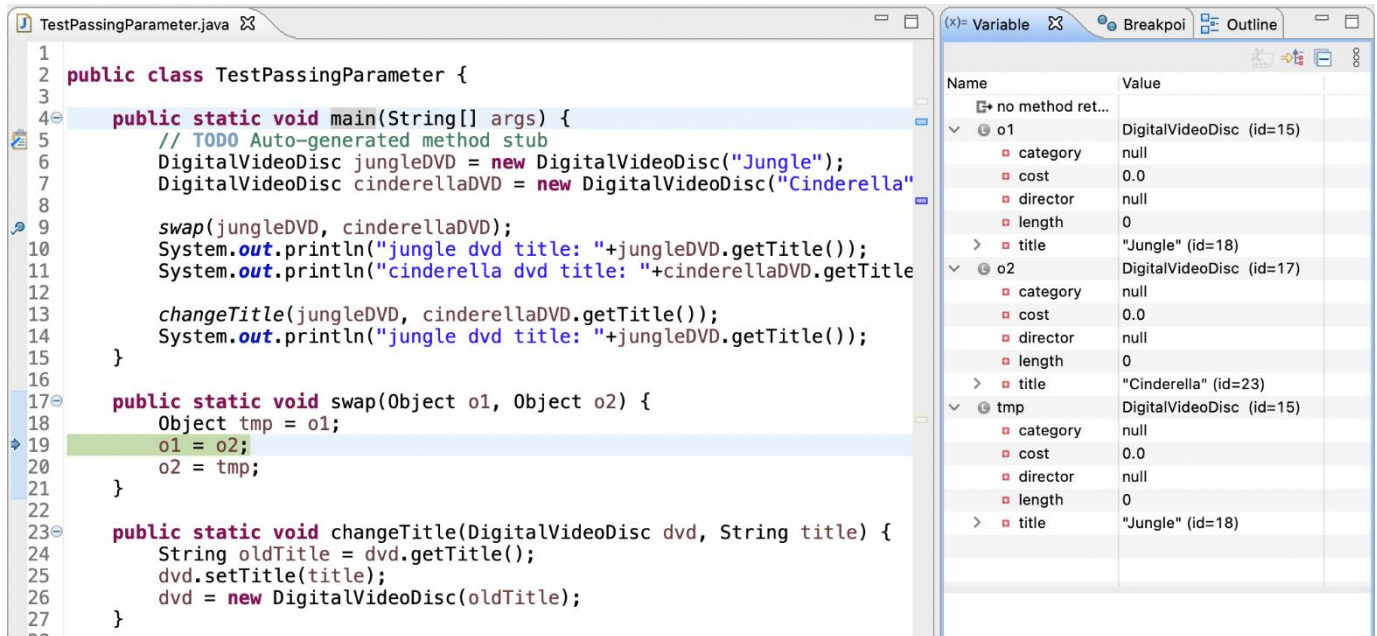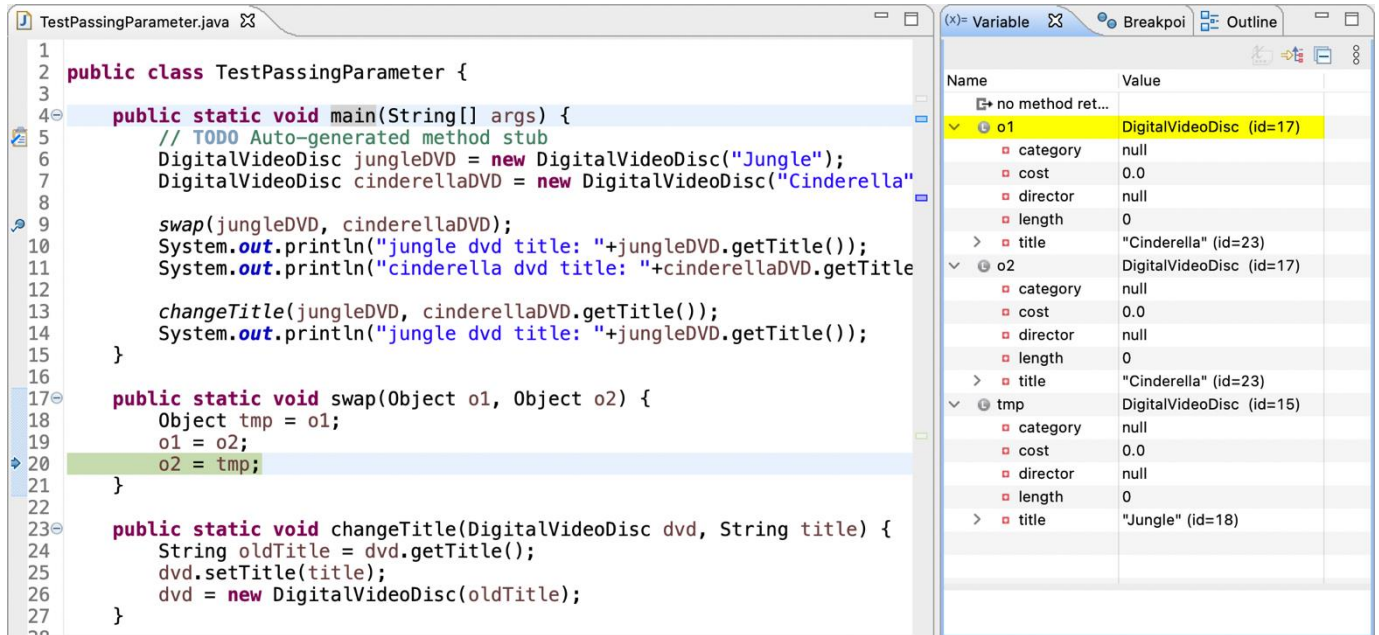

Figure 13. Step over line 18 of swap function
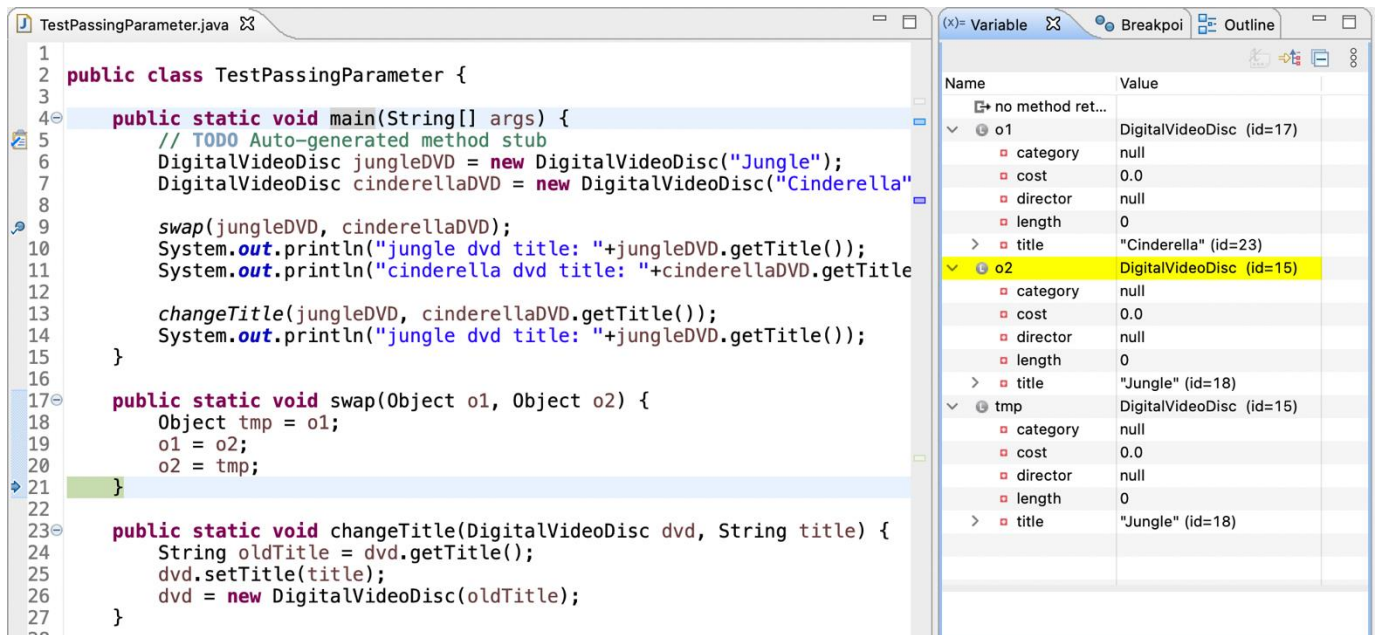
Figure 14. Step over line 19 of swap function

Figure 15. Step over line 20 of swap function

4.2.4. Change value of variables:

In the Variable Perspective, you can also change the value of variable while debugging.

Click Step Return so the debugger returns from the **swap** function back to the line after the call to it. (Figure 16)
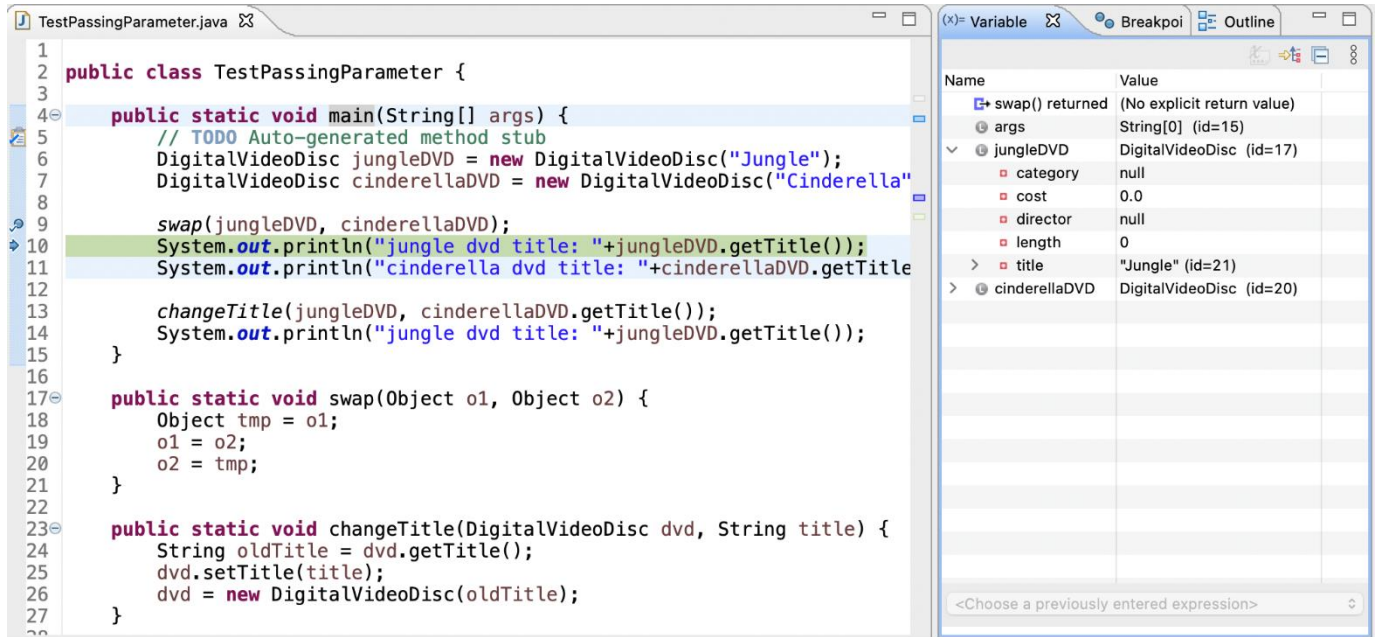
Figure 16. Step return to main function

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 17).
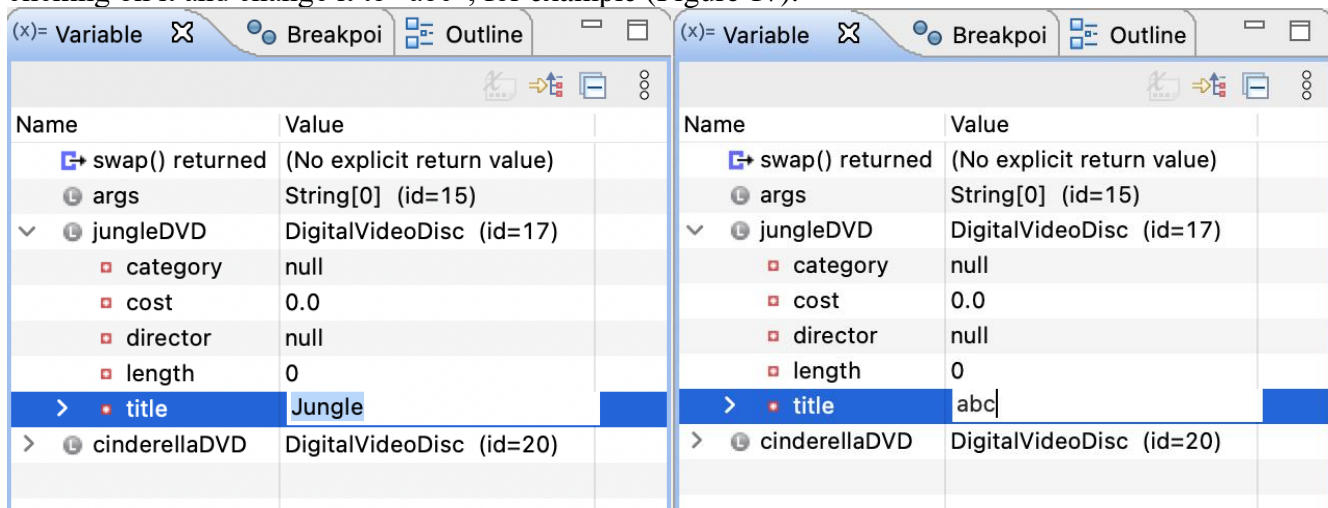

Figure 17. Change title of jungleDVD

Click Step Over and see the result in the output in the Console (Figure 18)
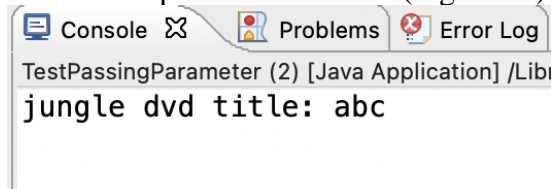

Figure 18. Results(2)

5. Classifier Member and Instance Member

- Classifier/Class member:
  - o Defined in a class of which a single copy exists regardless of how many instances of the class exist.
  - o Objective: to have variables that are **common** to all objects

- o Any object of class can change the value of a class variable; that's why you should always be careful with the side effect of class member
- o Class variables can be manipulated without creating an instance of the class
- Instance/Object member:
  - o Associated with only objects
  - o Defined inside the class but outside of any method
  - o Only initialized when the instance is created
  - o Their values are unique to each instance of a class
  - o Lives as long as the object does

Open the DigitalVideoDisc class:
- You should note that this class only has instance variables: **title**, **category**, **director**, **length**, **cost**.

- Now, we know that each DVD has a unique id assigned by the system. One simple way to manage all the ids is to give them out to new DVDs as consecutively incremented values. In order to do this, we must keep track of the number of DVDs created.

- Create a class attribute named "**nbDigitalVideoDiscs**" in the class **DigitalVideoDisc**

- Create an instance attribute named "**id**" in the class **DigitalVideoDisc**

```
private static int nbDigitalVideoDiscs = 0;
```

- Each time an instance of the **DigitalVideoDisc** class is created, the **nbDigitalVideoDiscs** should be updated. Therefore, you should update the value for this class variable inside the constructor method and assign the appropriate value for the **id**.

6. Open the **Cart** 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 problem statement in Lab02 for the matching rule.* **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 19. Code snippet for CartTest

7. Implement the **Store** class

- Create a **Store** class, 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 **StoreTest** class.

8. Re-organize your projects

- Rename project, use packages and re-organize 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.

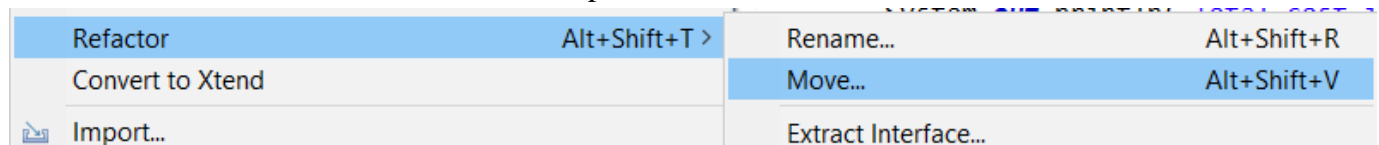|			
Refactor	Alt+Shift+T >	Rename...	Alt+Shift+R
Convert to Xtend		Move...	Alt+Shift+V
Import...		Extract Interface...	

Figure 20. Refactoring

+ For creating a package, right click to the project (or go to menu File) and choose New -> Package. Type the full path of package including parent packages, separated by a dot.
- Keep the text file for answering questions in the lab, the "**Requirement**" & "**Design**" 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 efficently organizing your classes in both projects and all listing packages. All the excercises of lab01 should be put in the corresponding package of one project - the OtherProjects project.

+ AimsProject
```
hust.soict.dsai.aims.disc.DigitalVideoDisc
hust.soict.dsai.aims.cart.Cart
hust.soict.dsai.aims.store.Store
hust.soict.dsai.aims.Aims
hust.soict.dsai.test.cart.CartTest
hust.soict.dsai.test.store.StoreTest
hust.soict.dsai.test.disc.TestPassingParameter
```
+ OtherProjects
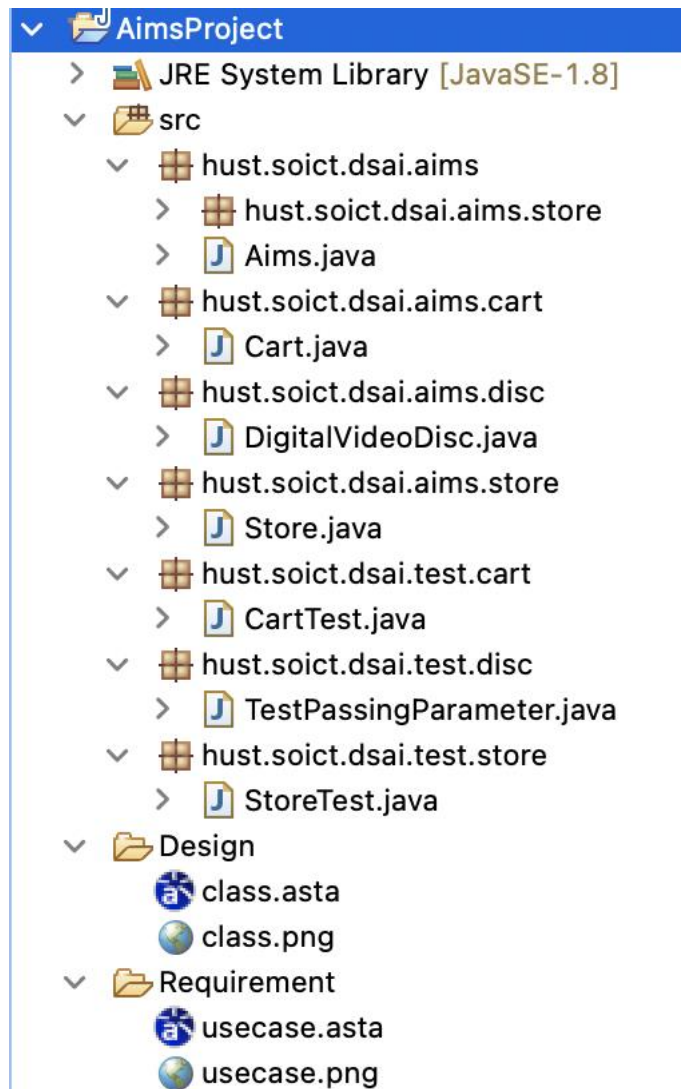```
hust.soict.dsai.lab01
```



*Figure 21. Recommended Structure for DS-AI*

16

# 9. **String**, **StringBuilder** and **StringBuffer**

- In the **OtherProjects** project, create a new package **hust.soict.globalict.garbage** for ICT or **hust.soict.dsai.garbage** for DS-AI. 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**.

```
1   public class ConcatenationInLoops {
2       public static void main(String[] args) {
3           Random r = new Random(123);
4           long start = System.currentTimeMillis();
5           String s = "";
6           for (int i = 0; i < 65536; i++)
7               s += r.nextInt(2);
8           System.out.println(System.currentTimeMillis() - start);  // This prints roughly 4500.
9
10          r = new Random(123);
11          start = System.currentTimeMillis();
12          StringBuilder sb = new StringBuilder();
13          for (int i = 0; i < 65536; i++)
14              sb.append(r.nextInt(2));
15          s = sb.toString();
16          System.out.println(System.currentTimeMillis() - start);  // This prints 5.
17      }
18  }
```

Figure 22. ConcatenationInLoops

For more information on **String** concatenation, please refer 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 hangs or even stop working when 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

```
 8          String filename = "test.exe"; // test.exe is the name or path to an executable file
 9          byte[] inputBytes = { 0 };
10          long startTime, endTime;
11
12          inputBytes = Files.readAllBytes(Paths.get(filename));
13          startTime = System.currentTimeMillis();
14          String outputString = "";
15          for (byte b : inputBytes) {
16              outputString += (char)b;
17          }
18          endTime = System.currentTimeMillis();
19          System.out.println(endTime - startTime);
```

Figure 23. 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 24. New code using StringBuffer

10. Release flow demonstration

10.1. Hypothesis

We hypothesis that the Figure 25 shows the branches of our current remote repository.
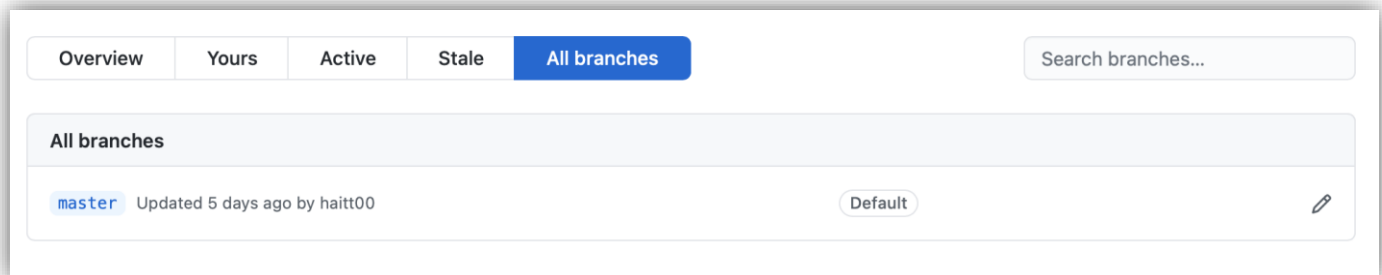

Figure 25. Branches of Remote Repository

Now we add a new topic or a new feature to our application. The next section shows us how to apply Release Flow in this hypothesis.

10.2. Demonstration

Step 1. Update local repository.
Issue the following command and resolve conflicts if any.

(master) $ git pull

Step 2. Create and switch to a new branch in the local repository.

(master) $ git checkout -b feature/demonstrate-release-flow

Step 3. Make modification in the local repository.

Step 4. Commit the change in the local repository.

(feature/demonstrate-release-flow) $ git commit –m "Add a feature for demonstration"

Step 5. Create a new branch in the remote repository (GitHub through GUI).

- Firstly, under the "Code" tab of the top navigation bar, choose the drop-down button with the branch name (in this case "master") on the top left.
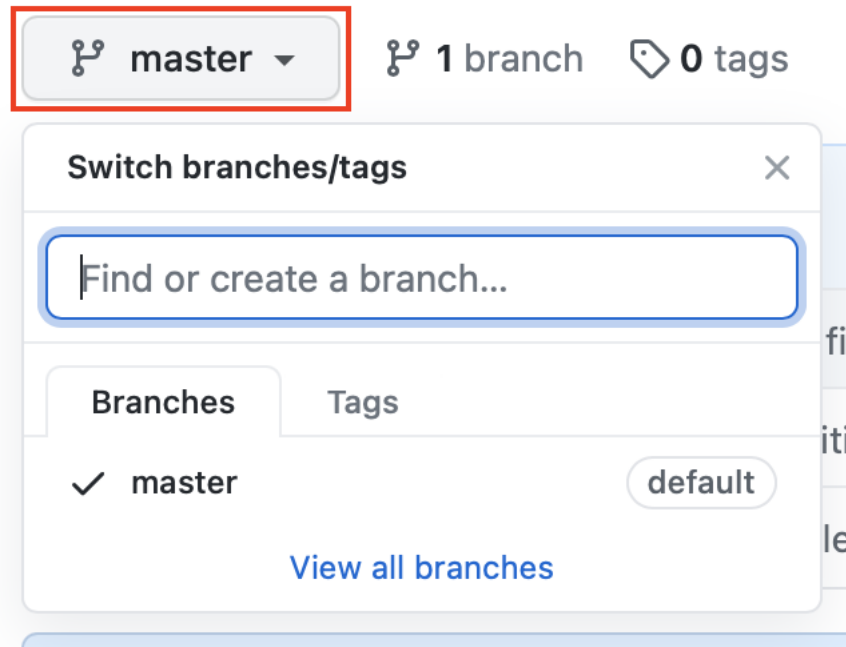

Figure 26. Branch Creation in GitHub GUI (1/3)

- Secondly, enter the new branch name "feature/demonstrate-release-flow" into the text field and click "Create branch: feature/demonstrate-release-flow from 'master'".
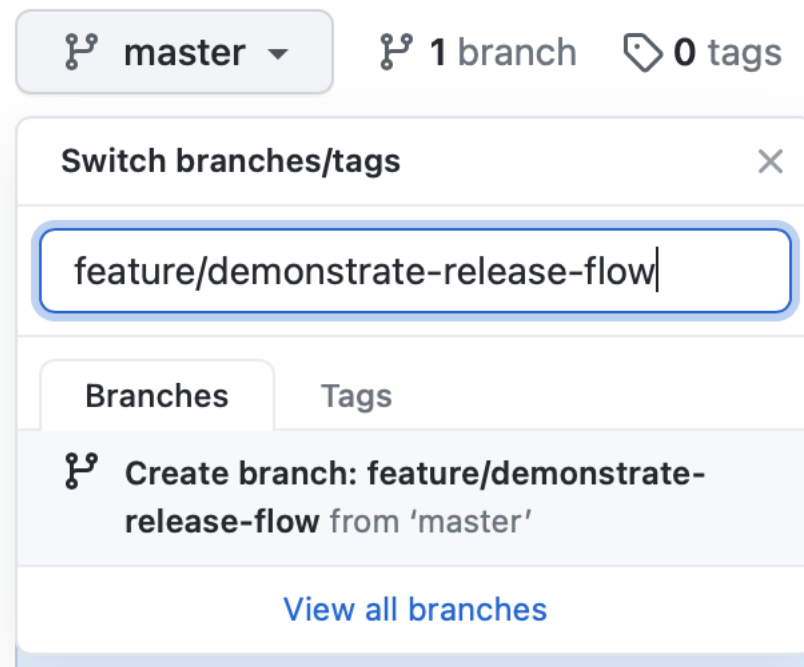

Figure 27. Branch Creation in GitHub GUI (2/3)

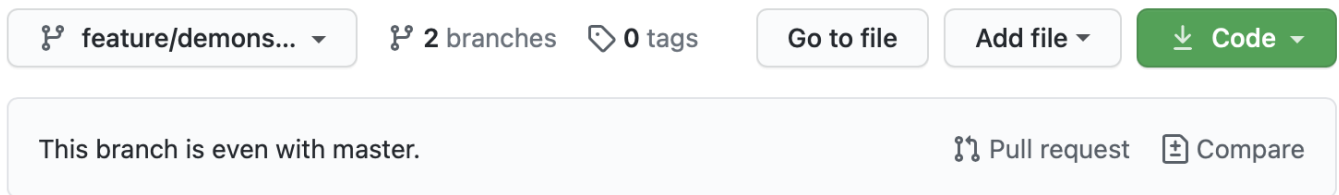- The following figure shows the result of our efforts in this step.

Figure 28. Branch Creation in GitHub GUI (3/3)

Step 6. Push the local branch to the remote branch

```
(feature/demonstrate-release-flow) $ git push origin feature/demonstrate-release-flow
```

Step 7. Create a pull request in GitHub GUI (for working in a team only)

- Firstly, choose "Pull requests" tab from the top navigation bar.

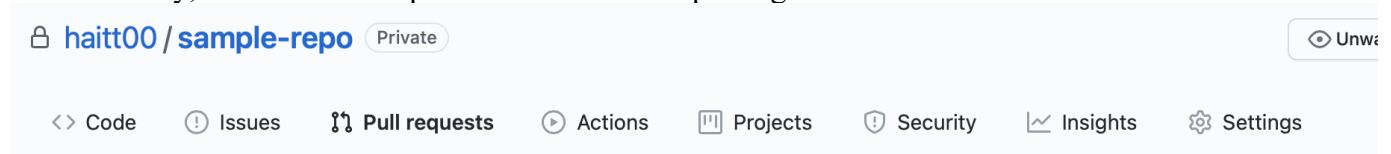*Figure 29. Creation of a Pull Request in GitHub GUI (1/4)*

- Secondly, click the button "New pull request" in the top right corner of the interface.
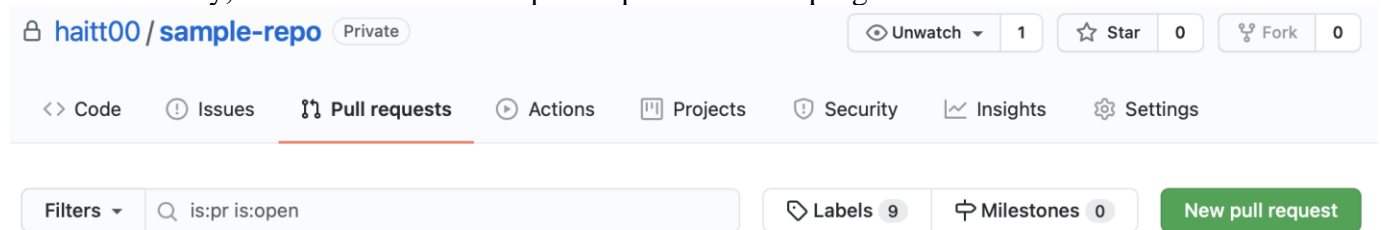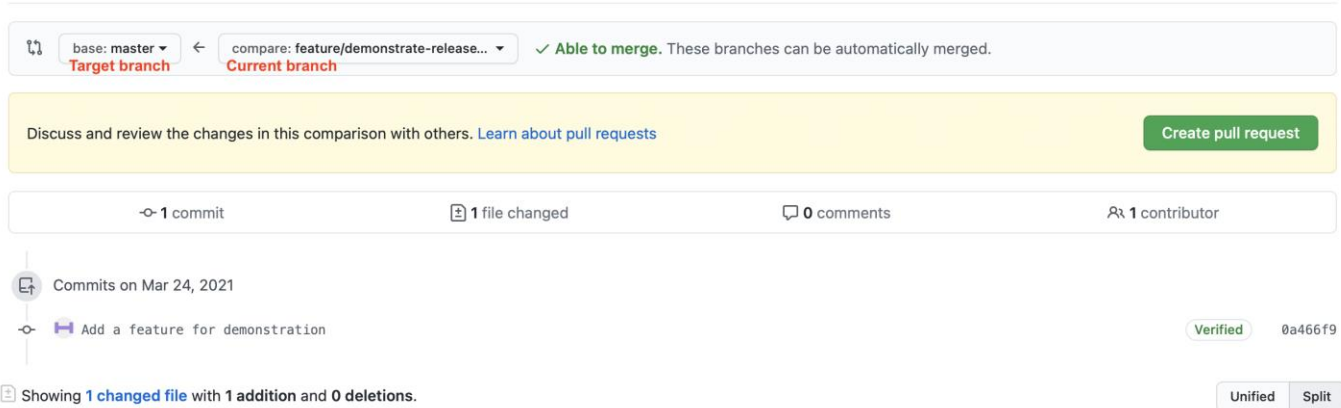

Figure 30. Creation of a Pull Request in GitHub GUI (2/4)

- Then, pick the target branch and current branch. Besides, at the bottom of the interface, we can see the changes between current branch and the target branch. Choose "Create pull request" to the top right.
  Note: the target branch will affect the destination branch which we want our branch merge to in the next step.

Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also compare across forks.

base: master ← compare: feature/demonstrate-release... ✓ **Able to merge.** These branches can be automatically merged.
Target branch **Current branch**

Discuss and review the changes in this comparison with others. Learn about pull requests   Create pull request

-○- 1 commit   ⬆ 1 file changed   💬 0 comments   👥 1 contributor

Commits on Mar 24, 2021

-○- ┝ Add a feature for demonstration   Verified   0a466f9

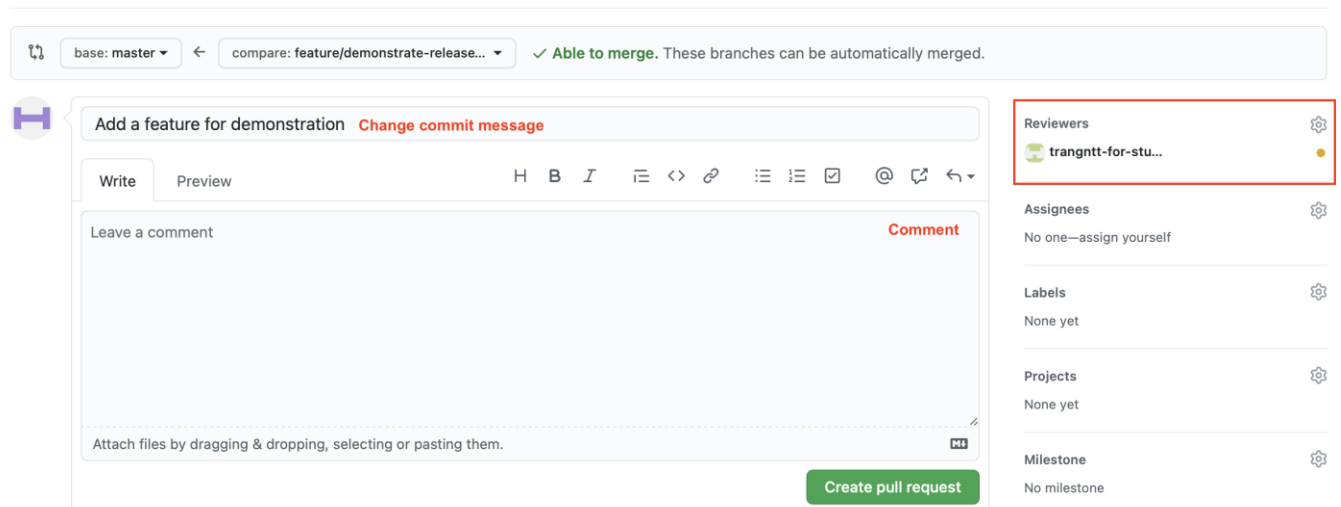Showing 1 changed file with 1 addition and 0 deletions.   Unified | Split

- *Figure 31. Creation of a Pull Request in GitHub GUI (3/4)*

- Lastly, choose reviewers for the pull request. We can also change the commit message, and add comment as we desire. Choose "Create pull request"

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks.

base: master ← compare: feature/demonstrate-release... ✓ **Able to merge.** These branches can be automatically merged.

Add a feature for demonstration   **Change commit message**

Write   Preview   H B I ≣ <> 🔗 ≣ ≣ ☑ @ ↗ ↩

Leave a comment   **Comment**

Attach files by dragging & dropping, selecting or pasting them.

Create pull request

Reviewers ⚙
🖼 trangntt-for-stu...   ●

Assignees ⚙
No one—assign yourself

Labels ⚙
None yet

Projects ⚙
None yet

Milestone ⚙
No milestone

- The following figure shows the result of our efforts in the dashboard of GitHub. The added reviewers also can see the pull requests in their dashboard. When the changes are viewed, we can merge the branches.
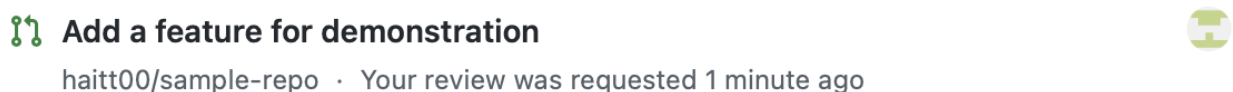

Recent activity

↕ **Add a feature for demonstration**
haitt00/sample-repo · Your review was requested 1 minute ago

Figure 32. Creation of a Pull Request in GitHub GUI (4/4)

Step 8. Merge the new remote branch to the master branch.
- Open the pull request.
- Choose "Merge pull request". You can choose one of several merge options from the drop-down menu
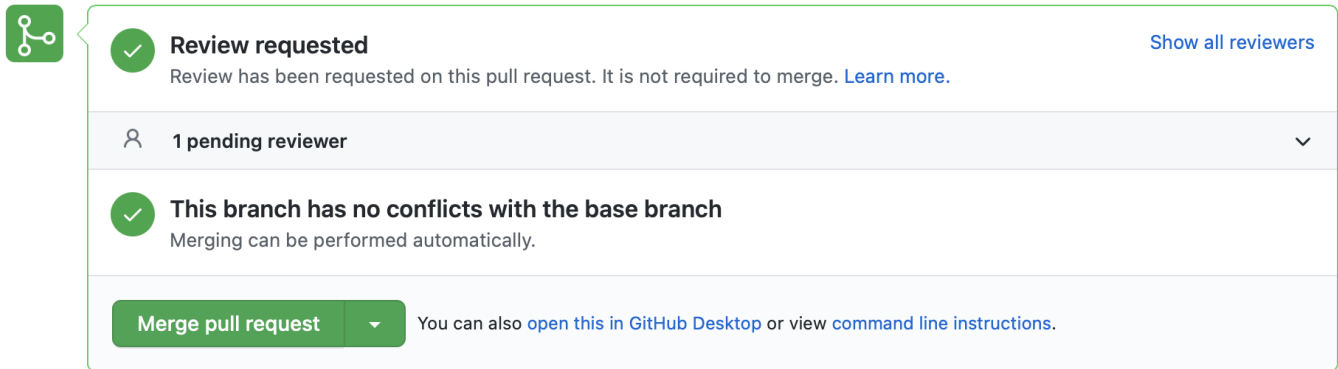

Figure 33. Branch merging (1/3)

- Lastly, change the commit message if need be. We cannot change the destination branch. Choose "Confirm merge" (as shown in Figure 33)
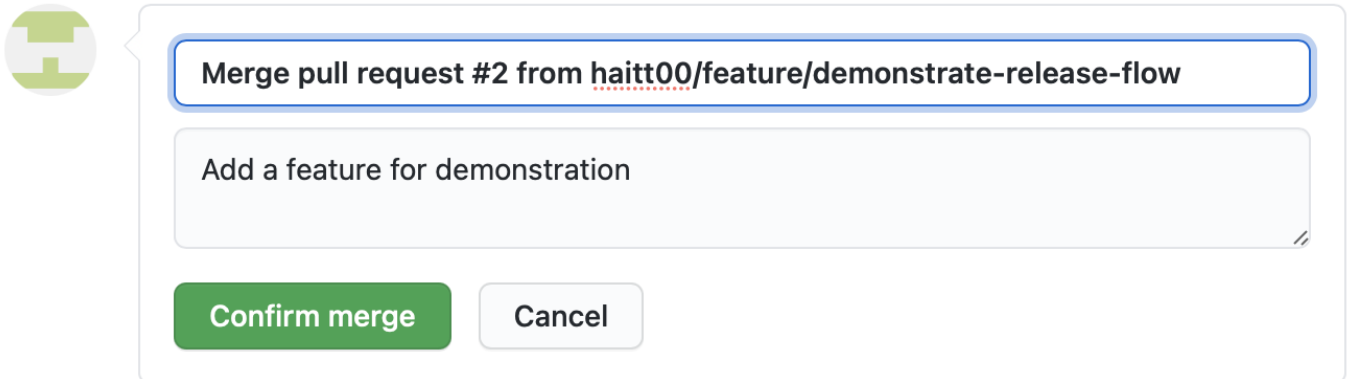

Figure 34. Branch merging (2/3)

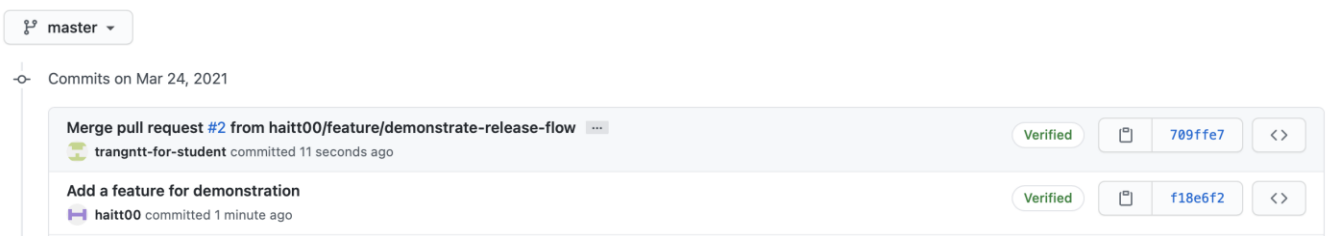- The following figure shows the result of our efforts. The changes from the target branch have been merged to the target branch "master".

Figure 35. Branch merging (3/3)