**BÁO CÁO THỰC HÀNH LAP 03  
LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG**

Nội dung

[**1.** **Branch your repository** 2](#_Toc119851561)

[**2.** **Working with method overloading** 5](#_Toc119851562)

[**2.1** **Overloading by differing types of parameter** 5](#_Toc119851563)

[**2.2**. **Overloading by differing the number of parameters** 6](#_Toc119851564)

[**3.** **Passing parameter** 7](#_Toc119851565)

[**4.** **Use debug run:** 9](#_Toc119851566)

[**4.1.** **Debugging Java in Eclipse** 9](#_Toc119851567)

[***4.2.*** **Example of debug run for the swap method of TestPassingParameter** 9](#_Toc119851568)

[***4.2.1.*** **Setting, deleting & deactivate breakpoints:** 9](#_Toc119851569)

[***4.2.2.*** **Run in Debug mode:** 10](#_Toc119851570)

[***4.2.3.*** **Step Into, Step Over, Step Return, Resume:** 10](#_Toc119851571)

[**4.2.4. Investigate value of variables:** 11](#_Toc119851572)

[***4.2.4.*** **Change value of variables:** 13](#_Toc119851573)

[**5.** Classifier Member and Instance Member 15](#_Toc119851574)

[**6.** **Open the Cart class** 15](#_Toc119851575)

[**7.** Implement the **Store** class 19](#_Toc119851576)

[**8.** **Re-organize your projects** 20](#_Toc119851577)

[**9.** **String**, **StringBuilder** and **StringBuffer** 21](#_Toc119851578)

# **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]](#footnote-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.
* 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 labssuch 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]](#footnote-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.

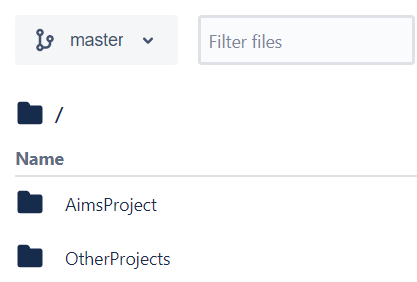


Figure 1. Merging result

**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

Ảnh có chứa bàn

Mô tả được tạo tự động

# **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.

Ảnh có chứa văn bản

Mô tả được tạo tự động

+ 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)**

**Ảnh có chứa văn bản

Mô tả được tạo tự động**

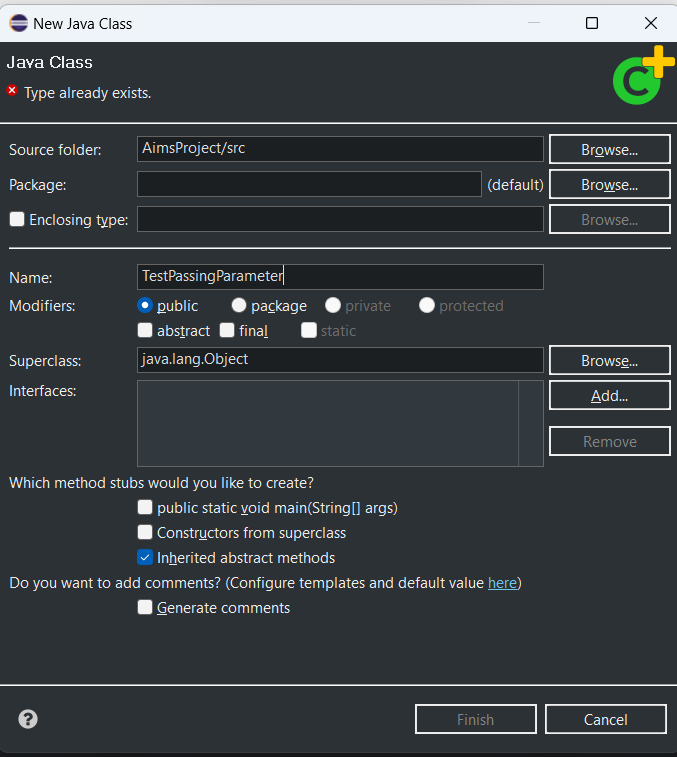
# **Passing parameter**

- 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

In the **main()** method of the class, typing the code

Ảnh có chứa văn bản

Mô tả được tạo tự động

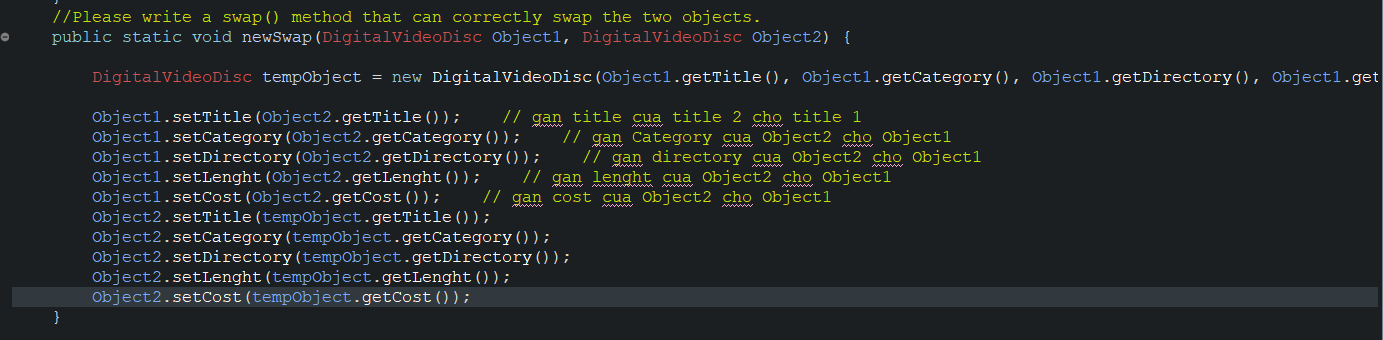
Ảnh có chứa văn bản

Mô tả được tạo tự động

**Java là một ngôn ngữ chỉ cho phép truyền tham số kiểu Tham trị**. Tức là giá trị bên trong phương thức truyền vào chỉ là một bản sao của giá trị bên ngoài, việc thay đổi giá trị này bên trong phương thức Java không gây ảnh hưởng hay thay đổi giá trị của biến bên ngoài phương thức.

* ở phương thức swap(Object o1, Object o2) thì hai đối tượng o1, o2 là hai đối tượng hoàn toàn mới được java tạo ra. Khi gọi phương thức swap(jungleDVD, cinderellaDVD) thì hai đối tượng jungleDVD và cinderellaDVD là hai đối tượng hoàn toàn mới, và chỉ là bản sao của hai đối tượng jungleDVD và cinderellaDVD bên trong hàm main. Khi đó khi ta gọi swap thì giá trị của hai đối tượng jungleDVD và cinderellaDVD bên trong hàm main ko thay đối, Vì vậy giá trị title của hai đối tượng vẫn giữ nguyên.
* ở phương thức changeTitle(DigitalVideoDisc dvd, String title) , ở đây tạo ra một đối tượng mới (dvd) và một thuộc tính mới (dvd), thuộc tính mới này sẽ lấy giá trị trực tiếp của title thông qua phương thức getTitle() và gán cho đối tượng mới (dvd) thông qua phương thức setTitle(). Nhờ đó mà tiêu đề khi gọi phương thức changeTitle() sẽ đc thay đổi.

**Please write a swap() method that can correctly swap the two objects.**



# **Use debug run:**

## **Debugging Java in Eclipse**

**Video:** [**https://www.youtube.com/watch?v=9gAjIQc4bPU&t=8s**](https://www.youtube.com/watch?v=9gAjIQc4bPU&t=8s)

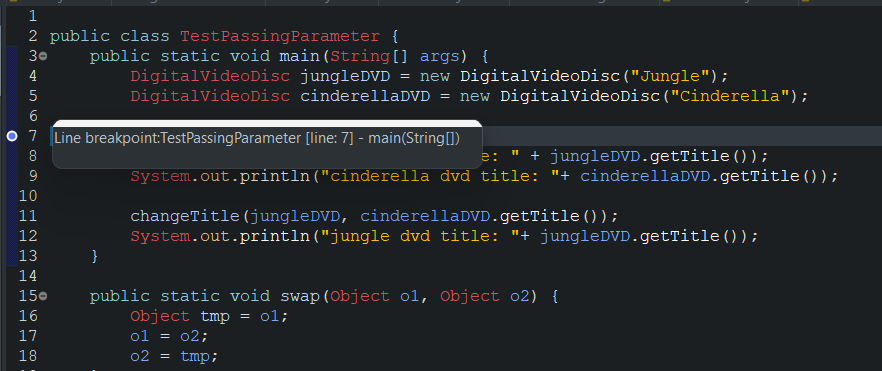
Debugging is the routine process of locating and removing bugs, errors or abnormalities from programs. It’s a must have skill for any Java developer because it helps to find subtle bugs that are not visible during code reviews or that only happen when a specific condition occurs. The Eclipse Java IDE 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.**

## **Example of debug run for the swap method of TestPassingParameter**

## **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.



## **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).

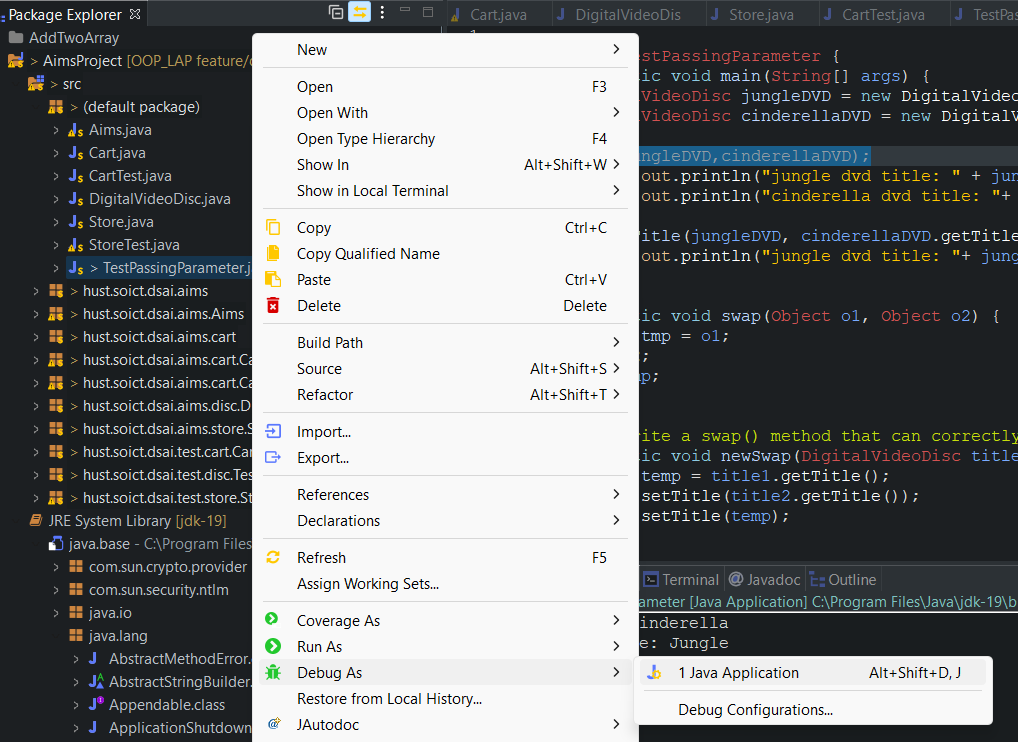


Figure 8. Run Debug from a class

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

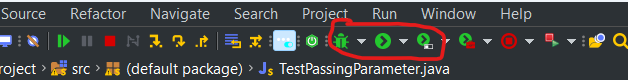


Figure 9. Run debug from a project

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

## **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.

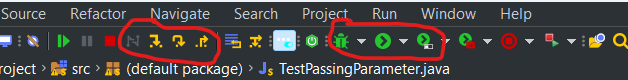


Figure 10. Stepping Commands on the Toolbar in Debug Perspective

- 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).

Ảnh có chứa văn bản

Mô tả được tạo tự động

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.

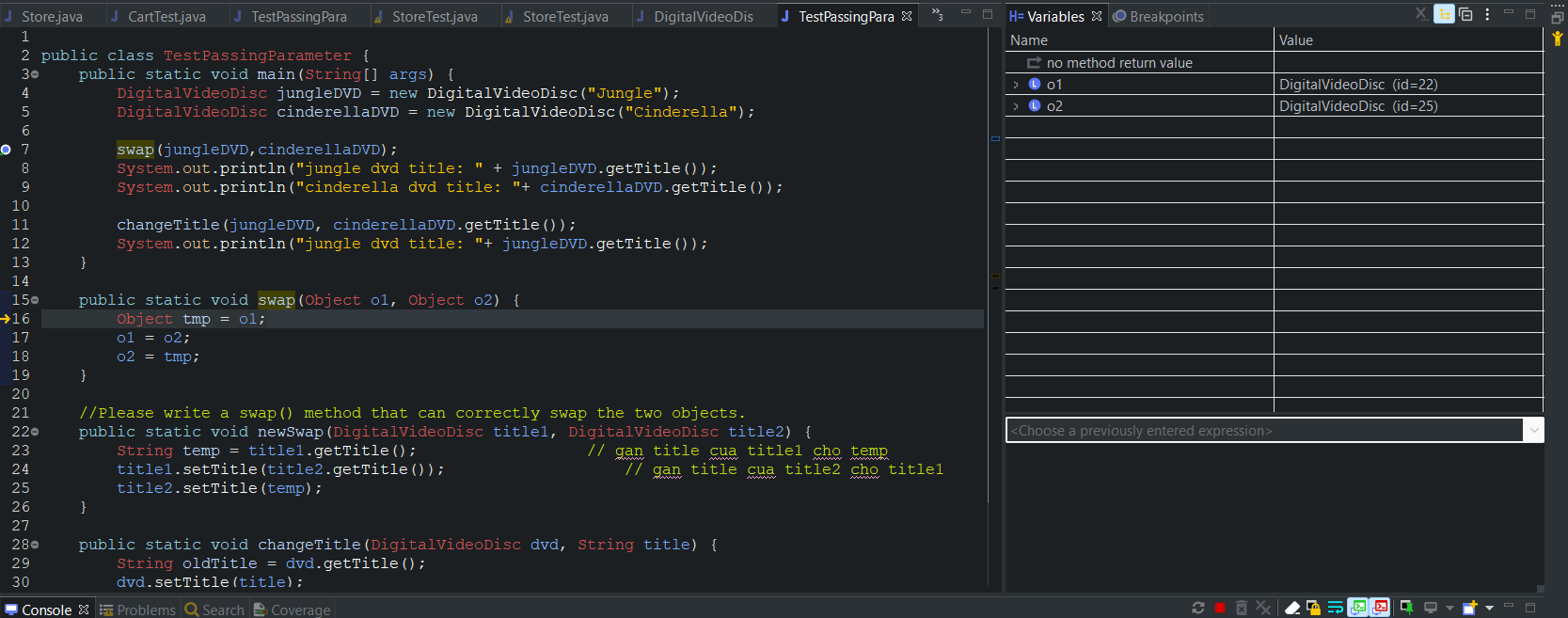


Figure 12. Variables shown in Variable View

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).

Ảnh có chứa văn bản, màn hình, ảnh chụp màn hình, TV

Mô tả được tạo tự động

Figure 13. Step over line 18 of swap function

Ảnh có chứa văn bản, màn hình, ảnh chụp màn hình, TV

Mô tả được tạo tự động

Ảnh có chứa văn bản, màn hình, TV, ảnh chụp màn hình

Mô tả được tạo tự độngFigure 14. Step over line 19 of swap function

Figure 15. Step over line 20 of swap function

## **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)

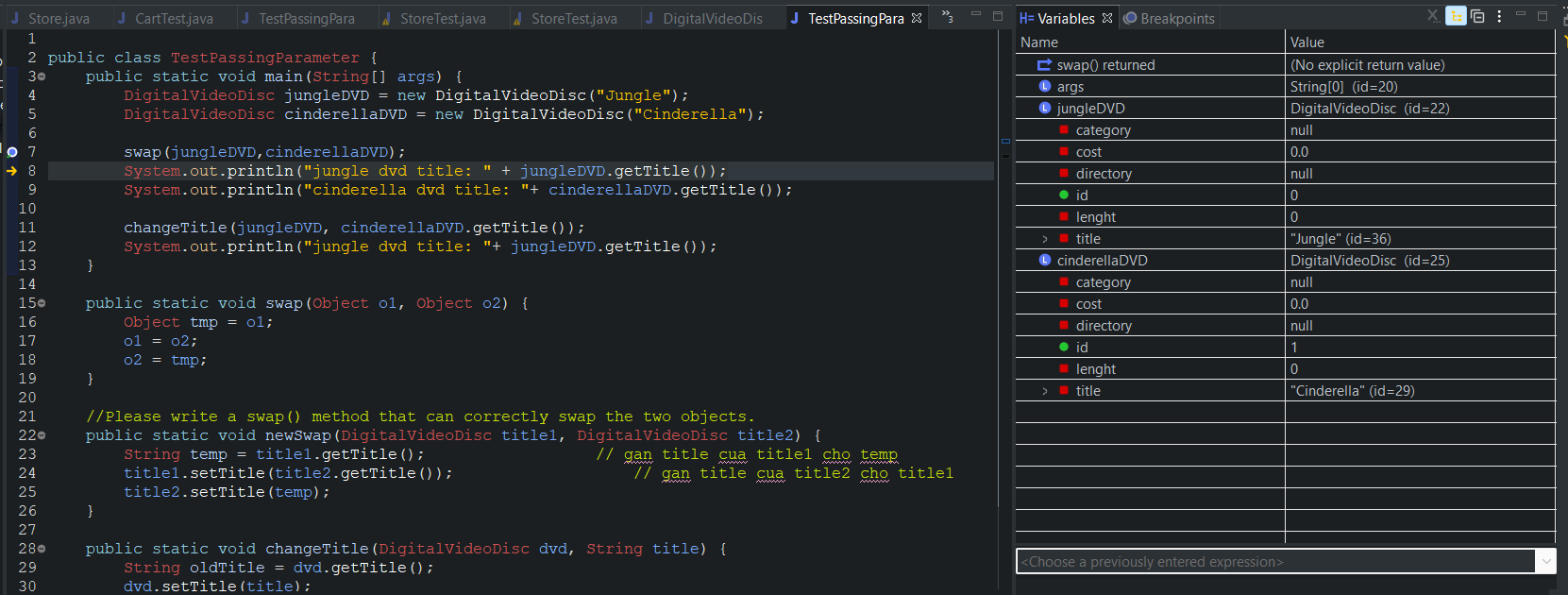
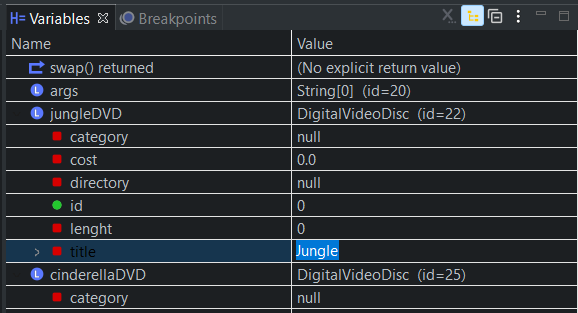


Figure 16. Step return to main function

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).

Ảnh có chứa bàn

Mô tả được tạo tự động

Figure 17. Change title of jungleDVD

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

Ảnh có chứa văn bản

Mô tả được tạo tự động

# Classifier Member and Instance Member

**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**

private static int nbDigitalVideoDiscs = 0;

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

- 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**.

Ảnh có chứa văn bản

Mô tả được tạo tự động

# **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] $

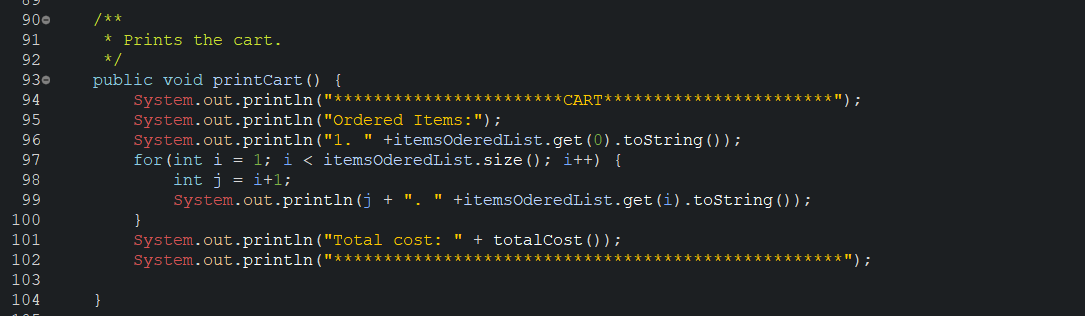
2. DVD - [Title] - ...

Total cost: [total cost]

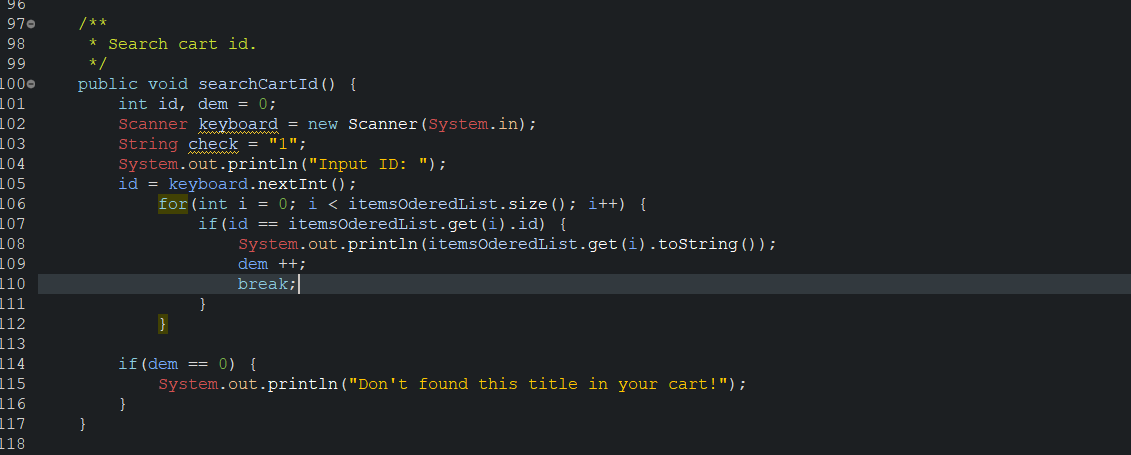
**Suggestion**: Write a **toString()** method for the **DigitalVideoDisc** class. What should be the return type of this method?

Ảnh có chứa văn bản, ảnh chụp màn hình, đóng

Mô tả được tạo tự động



* Search for DVDs in the cart by ID and display the search results. Make sure to notify the user if no match is found.



Ảnh có chứa văn bản

Mô tả được tạo tự động

- 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.

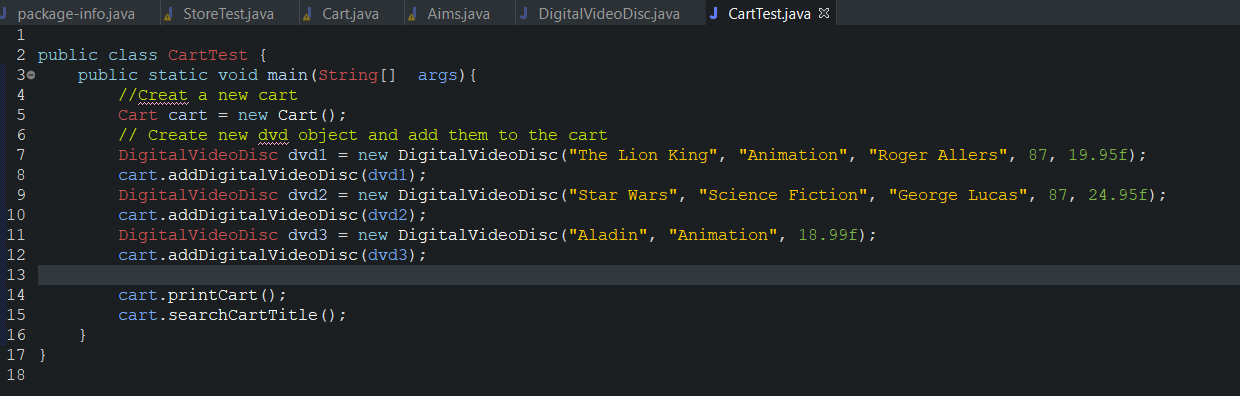
***Ảnh có chứa văn bản

Mô tả được tạo tự động***

***Ảnh có chứa văn bản

Mô tả được tạo tự động***

- 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:



***Ảnh có chứa văn bản

Mô tả được tạo tự động***

# Implement the **Store** class

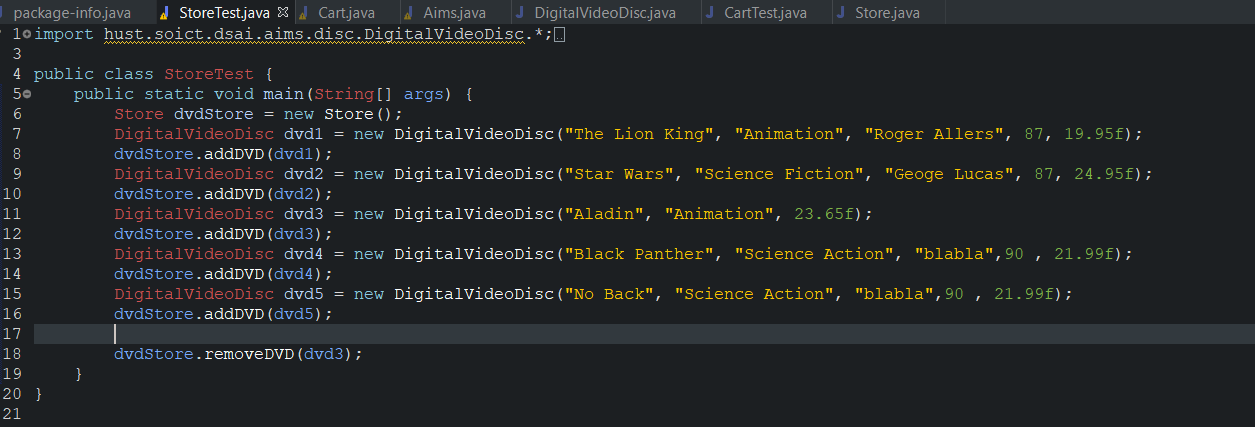
- Create a **Store** class, which contains one attribute **itemsInStore[]** – an array of DVDs available in the store.

- To add and remove DVDs from the store, implement two methods called **addDVD** and **removeDVD**

Ảnh có chứa văn bản

Mô tả được tạo tự động

- Test these two methods in **StoreTest** class.



Ảnh có chứa văn bản

Mô tả được tạo tự động

# **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.

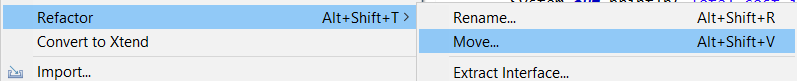


Figure 20. Refactoring

+ 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**

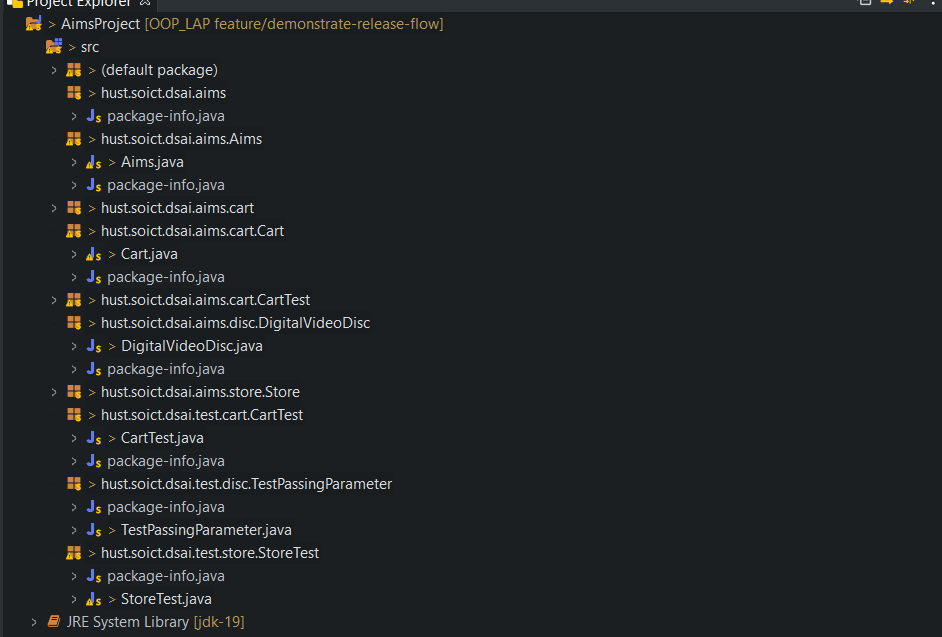
**hust.soict.dsai.test.cart.CartTest**

**hust.soict.dsai.test.store.StoreTest**

**hust.soict.dsai.test.disc.TestPassingParameter**

**+ OtherProjects**

**hust.soict.dsai.lab01**

 Ảnh có chứa văn bản

Mô tả được tạo tự động

# **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**.

Ảnh có chứa văn bản

Mô tả được tạo tự động

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.

Ảnh có chứa văn bản

Mô tả được tạo tự động

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.**

Ảnh có chứa văn bản

Mô tả được tạo tự động

Figure 23. Sample code for GarbageCreator

1. <https://docs.microsoft.com/en-us/azure/devops/repos/git/git-branching-guidance?view=azure-devops> [↑](#footnote-ref-1)
2. <https://www.atlassian.com/git/tutorials/making-a-pull-request> [↑](#footnote-ref-2)