Object-Oriented Language and Theory

Nguyen Thi Thu Trang, trangntt@soict.hust.edu.vn

Mini-Project Guidelines

1. Mini-Project Guidelines

1.1. Deadline

- **Week 15 (10PM December 11th, 2023):** Each team must choose a topic that their team wants to work on developing.
- **Week 15 (10PM December 16th, 2023):** Submission of Use case diagram & General class diagram
- **Week 17 (10PM December 30th, 2023):** Submission of Detailed class diagram
- **Week 18 (10PM January 6th, 2024):** Submission of all required documents and source code as described below
- **From Week 19:** Presentation of Mini-project

1.2. Overview

- All the mini-projects must be designed, applied object-oriented theory learnt in this course (e.g., **Encapsulation, Abstraction, Polymorphism, Inheritance**), and implemented in **Java** with a complete GUI and by students themselves. **If the teacher finds out that students didn't write the source code (even a part of it), the score will be 0**.
- You can use a library or even an open source to develop your mini-project, but **not encourage**. **You will be highly evaluated if you did all your mini-project by yourselves**. However, if you use any library or modify an open source, please remember to claim them in the report and presentation.
- Mini-Project Submission: Commit and push all your results as soon as possible to github (through git) before the announced deadline:
    - Report (pdf: pdf document): You don't need to submit the **printed version,** only need to push to the github into a directory, named "**report**", inside the root directory of the mini-project.
        - Assignment of members
            - Detail for classes/methods
            - Claim clearly if you copy/copy with modify/use the idea of any source. Otherwise, you will receive 0 for the midterm score.
        - Mini-project description
            - Describe in detail about your mini-project requirement
            - Use case diagram and explanation: How the users interact to the software with use cases
        - Design
            - A general class diagram: Class diagram may be with packages, including all classes without attributes/operations
            - Several class diagrams for each package or several packages, with detail attributes/operations for each class

        - Explanation of the design: Describe the relationships between classes, the implementations of some important methods

    - Slide to be pushed to github into a directory, named "**presentation**", inside the root directory of the mini-project: **Maximum of 10 slides:**
        - Slide 1: Members & assignment
        - Slide 2: Problem statement
        - Slide 3: Use case diagram
        - Slide 4: General class diagram
        - Slide 5,6: Class diagrams for packages/modules
        - Slide 7,8,9: Explanation of OOP techniques in your design: e.g. Inheritance, Polymorphism, Association/Aggregation/Composition
        - Slide 10: Demo scenario with **video link.**

    - Use case diagram and Class diagram: Put the .asta files and their exported images (in .png format) into a directory, named "**design**", inaside the root directory of the mini-project. These diagrams **must be designed by Astah UML software**.

    - Source codes of the mini-project in a directory, named "**sourcecode**"

    - Readme.md:
        - **Member information and assignment**
        - **Link of demo video for the application**

- Using a database is **not encouraged**.
- Using an additional framework, library, and/or API is **not encouraged**.
- Mini-Project Defense:

    - Application Demonstration (**live**): 5 minutes

    - **Design Explanation**: 10 minutes

    - **Q&A**: unlimited

1.3. Development Process

Step 1: Feasibility Study. All the team members analyze, understand the problem, and propose solutions/strategies to solve the problem.

Step 2: Requirement Analysis. From understanding of the problem, all the team members ought to be involved in this step so as to create a *sketch version* for the Use case diagram of your application.

Step 3: Design. We can create a work breakdown structure here to assign tasks to the team members.

UI/UX Design: Scene Builder (JavaFX) or Window Builder (Swing).

Component Design: Use case diagram and Class diagram.

Step 4: Implementation and Testing. Every piece of code had better be immediately tested after being written. When we integrate our codes, we must test, too.

Note: We always can go back to previous step(s) to make changes since nothing is perfect.

1.4. Version Control

1.4.1. Requirement

- Create a private repository with the naming convention **OOLT.HEDSPI.20231-TeamID** , e.g. OOLT.HEDSPI.20231-04. If you do not follow this naming convention, your repository will be ignored.
- Add trangntt.for.student@gmail.com as a member of your repository.

Note: Commit history reflects the contribution of each team member to the team.

1.4.2. Git Workflow in A Team

Applying Release Flow is required.

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 branches. Others are flexible.**

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

Table 1-Branching policy

[1] If we want to update your newly created branch, we could add codes to a new hotfix branch. Then we merge the `hotfix` branch with `master` and with the `release` branch. There is another way: we can delete the latest release branch, update master, and then create a new release branch.

[2] **Latest version of the project in the latest release branch serves as the submitted project.**

We can create a new release branch when we add a new feature, fix a critical bug or a few bugs, or refactor our code. Usually, we would create a new release branch when we add a new feature.

Typical steps[1]:

- 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 have changed"
- 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**
- **File a pull request via GitHub GUI.** The rest of the team reviews the code, discusses it, and alters it.
- **Team leader merges** the remote branch (e.g. **abc**) into the official repository and closes the pull request (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).

2. Shades of Java

Here are some Java Features & Terminologies that we might see while working on the mini-project.
Note: This part is just for **reading purposes only** and will not be included in the final examinations.
For the new features of Java 8, please see https://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html and https://o7planning.org/en/10323/syntax-and-new-features-in-java-8.

2.1. Threading in Java

Keywords: Threads, Runnable, run(), start(), yield(), stop(), scheduling
See the following links

- https://developer.ibm.com/technologies/java/tutorials/j-threads/
- https://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html
- https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

2.2. Stream

Keyword: stream()

Why? For **aggregate computations** of collections of objects

For example,

```
int total = transactions.stream() .filter(t -> t.getBuyer().getCity().equals("London"))
                                  .mapToInt(Transaction::getPrice)
                                  .sum();
```

[1] https://www.atlassian.com/git/tutorials/making-a-pull-request#how-it-works

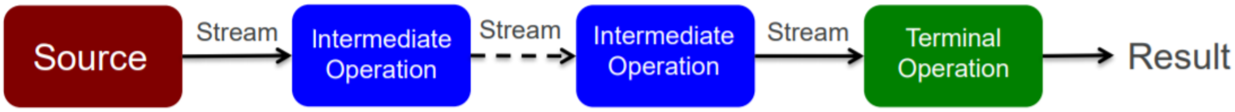

Figure 1-Stream Pipeline[2]

If you need more details, you can see here (you will need an Oracle account).

2.3. Lambda Expressions

Lambda expressions basically express instances of Functional Interfaces in Java (see 2.6).

Why?

- Lambda expressions simplify how to **pass behavior as a parameter**
- A **function** that can be **created without belonging to any class**.

See some other reasons here.

Syntax: **(** [parameter(s)] **) -> {** <**codes** of a function> **}**

See more about the syntax at this link.

If you need more details, you can see here (you will need an Oracle account).

Note: https://www.geeksforgeeks.org/difference-between-anonymous-inner-class-and-lambda-expression/?ref=rp

2.4. JavaBeans

A JavaBean is a specially constructed Java **class written in the following standards**:

- It provides a default, no-argument constructor.
- It should be serializable and that which can implement the **Serializable** interface.
- It may have a number of properties which can be read or written.
- It may have a number of "**getter**" and "**setter**" methods for the properties.

2.5. Method and Constructor References

Method references let us reuse a method as a lambda expression, and same concept for constructor references. Also, they use **double colon (::) operator**. For better understanding, see this link.
Format:

target_reference::method_name

2.6. Functional Interfaces in Java

A functional interface is an **interface** that contains **only one abstract method**. That's all.

[2] https://www.oracle.com/webfolder/technetwork/tutorials/moocjdk8/documents/week2/lesson-2-2.pdf

Beside the way to create a new class implementing functional interface, we could use Lambda Expressions as shown in section 2.3 above. See more at the following links.
Functional Interfaces And Their Definition
Functional Interfaces in the java.util.function Package

2.7. Anonymous Classes

It is an **inner class without a name**. Use them if you need to use a **local class only once**.
See more at https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html
Syntax:

```
new <ClassName> ( [parameter(s)] ) {
        // attributes & methods
}
```

Note:
https://www.geeksforgeeks.org/difference-between-anonymous-inner-class-and-lambda-expression/?ref=rp

2.8. Annotation in Java

Annotations start with "**@**" and **provides supplement information** of a program. Annotations have **no direct effect** on the operation of the code they annotate.

For better understanding, see https://www.geeksforgeeks.org/annotations-in-java/.

3. Reference for GUI programming

The following might help you with GUI programming through tutorials, sample codes and explanations.

- Swing:
  https://www3.ntu.edu.sg/home/ehchua/programming/java/j4a_gui.html
  https://zetcode.com/javaswing/
- JavaFX:
  https://o7planning.org/11009/javafx
  https://code.makery.ch/library/javafx-tutorial/
- Oracle Java tutorial (includes both JavaFX and Swing)
  https://docs.oracle.com/javase/8/javase-clienttechnologies.htm
- Advanced 2D graphics programming: http://docs.oracle.com/javase/tutorial/2d/index.html
- General tips for UI/UX design:
  https://www.cs.umd.edu/~ben/goldenrules.html
  http://athena.ecs.csus.edu/~buckley/CSc238/Psychology%20of%20UX.pdf