COMP2511

Tutorial 4

Last Week's Tutorial

- Testing
  - JUnit, debugging, exceptions
- Domain Modelling
  - UML diagrams, looking at requirements

This Week's Tutorial

- Code Smells
- SOLID Principles
- Streams
- Design by Contract

Notices

- Assignment 1 is due next week Friday!
  - **Try to start if you haven't already!** The design is very hard to get right in the first go, so the more time you have to reiterate on your design the better.
  - Standard late penalties apply, applied per hour (but don't use this as an excuse to start the night before!).
- Assignment 2 groups will be formed soon.
  - Make sure to fill the form, regardless of whether you have a partner in mind.

Code Smells

- A **code smell** is a feature of code that is indicative of an underlying design problem.
- Importantly, code smells are **indicators** of design problems - they may not necessarily *guarantee* that something is wrong and some potential design issues could be unavoidable or are the best available solution depending on the scenario, but you should still acknowledge these when they are present.

Examples of Code Smells

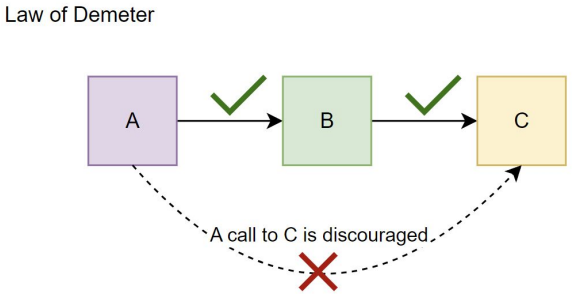
- **Data Class** - a class contains only fields and methods to modify or access them, without providing particularly useful functionality to act on that data.
- **Feature Envy** - a class uses the fields and methods of another class more than it uses its own fields and methods.
- **Shotgun Surgery** - changing one part of a system has a collateral effect on many other classes, requiring many small changes to be made to those classes.
- **Refused Bequest** - a class inherits fields or methods that do not make sense or are widely unused.
- **Message Chains** - a class performs a method, then another method is performed on top of that method call, and so on.

SOLID Principles

- A set of five design principles that are intended to make object-oriented systems more understandable, flexible and maintainable.
  - **S**ingle Responsibility Principle: Each class should have one specific role, and perform only that role.
  - **O**pen-Closed Principle: Entities should be open for extension, but closed for modification.
  - **L**iskov Substitution Principle: Subclasses should be valid instances of their superclass; substituting them for their parent class should not alter the program's correctness.
  - **I**nterface Segregation Principle: An interface should only prescribe methods that are relevant to the classes that will implement it (i.e. make bite-sized interfaces).
  - **D**ependency Inversion Principle: Classes should interact with their dependencies only through abstractions, and not be required to know any concrete details of those dependencies.

The Law of Demeter/Principle of Least Knowledge

- Objects should only interact with their immediate dependencies, and assume as little as possible about anything else in the system.
- Essentially, this means that classes should only talk to its neighbours, rather than any strangers or neighbours of neighbours.
- In particular, abiding by this principle helps to avoid **tight coupling** by preventing changes in one class from cascading to many other classes.



When is it OK for a class to use another class' methods?

- Within the method of a class, that class is allowed to use:
  - its own methods (of course!)
  - methods of objects that are fields of the class
  - methods of objects that are passed in as method parameters
  - methods of objects that are **instantiated** within the method
- Classes should NOT use:
  - the methods of objects that are returned by a method of another class
- Common examples of Law of Demeter violations:
  - `x.getA().getB();`
  - `A a = x.getA(); B b = a.getB();`
  - The second may seem like a sneaky workaround, but it is the exact same as the first!
- A potential Law of Demeter violation, up to interpretation:
  - `x.getA().equals(...);`

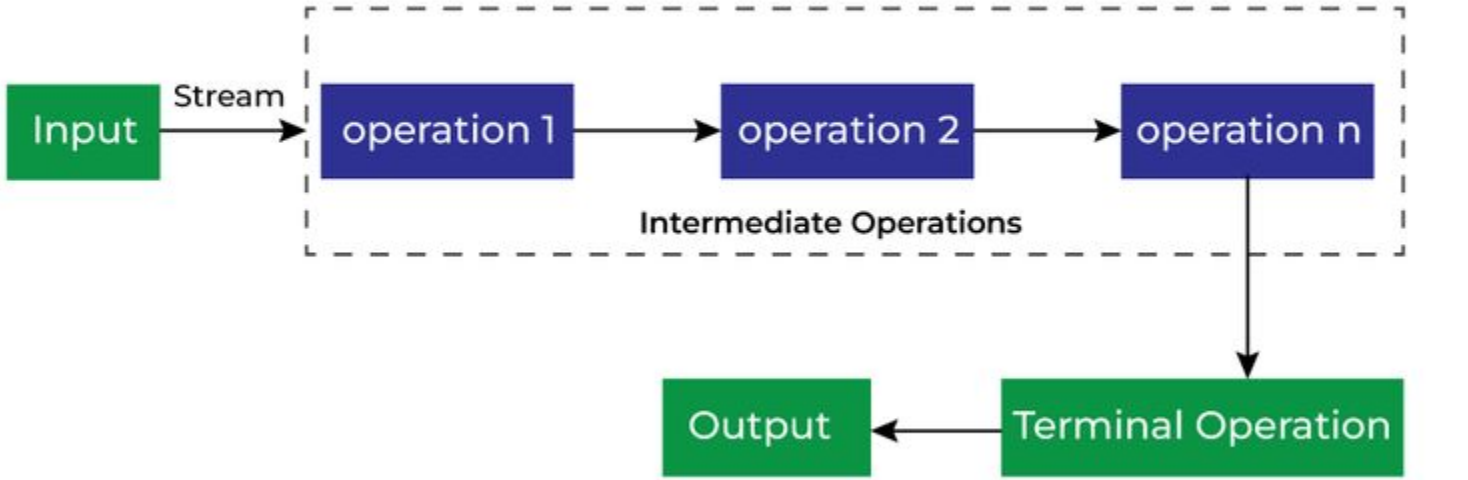
Live Example: Design Principles

- In **src/training**, there is some skeleton code for a training system.
  - Trainers are hosting seminars, with the restriction that each seminar cannot have more than 10 attendees.
  - An attendee can attend a seminar if they are available on the date of the seminar.
- In the TrainingSystem class, there is a method to book a seminar for an employee given the dates on which they are available. This method violates the principle of least knowledge (Law of Demeter).
  - How and why does it violate this principle?
  - In violating this principle, what other properties of this design are not desirable?
  - Refactor the code so that the principle is no longer violated.
  - [KEY TAKEAWAYS] Refactoring the message chains code smell.

Streams

- Collections in Java (eg. ArrayList, TreeMap, TreeSet, HashMap) can be transformed into an abstracted sequence of the same elements, referred to as a **stream**.
- Elements in the stream can only be interacted with through methods that provide high-level abstractions to perform operations like **mapping**, **filtering**, or **reducing**.
    - Multiple stream operations can be chained together as required. For example, if you had a stream of integers and you wanted to take all of the numbers that were less than 5 and then multiply all remaining numbers by 3, you would **filter** the stream, then **map** the stream accordingly.
- After all necessary operations have been done, a stream can be converted back into the original type of collection, as required.

Streams


Input
Stream
operation 1
operation 2
operation n
Intermediate Operations
Output
Terminal Operation

Live Example: Streams

- In **src/stream** there is a program that uses standard for-loops to print out a list of strings, and parse integers from a list of strings and print out the new list of parsed integers.
  - Rewrite the first for-loop using the forEach() method and a lambda expression.
  - Rewrite the second for-loop using streams and the map() method.
  - [KEY TAKEAWAYS] Writing a lambda expression, using a stream to map elements and converting streams back into a list.

Defensive Programming vs. Design by Contract

- **Defensive Programming** is a programming approach that accounts for unforeseen circumstances, and tries to ensure that programs will behave predictably despite unexpected inputs or user actions.
  - For example, checking for null inputs in every method before any logic is performed.
  - This approach makes programs more secure and robust, but it could potentially obscure the main logic of the program and be redundant.
- **Design by Contract** refers to writing your code around a set of clear and well-defined specifications, which **when adhered to** guarantees correct and expected behaviour.
  - For example, a programmer applying design by contract does not need to perform any null checks if it is explicitly stated in the requirements (contract) that null will not be given as an input.
  - This can a program's logic clearer, but vulnerable when this contract is not abided to.

Design by Contract

- In the context of a method,
  - a **precondition** is a condition on the **input** that the method will perform as expected.
  - a **postcondition** is a guarantee on what the method will output/perform, *given that the precondition holds*.
  - an **invariant** is a property of the system that is guaranteed to be maintained once the method has been performed.
    - this property can change *during* the method's execution, the importance is that it should still hold once the method is completed.
- Essentially, if someone defines a precondition for a specific method and it isn't satisfied when the method is executed, then too bad!

Design By Contract and LSP

- Contracts are inherited from superclasses to their subclasses.
- According to the Liskov Substitution Principle, every instance of a subclass must be a valid instance of the superclass.
- This means that:
  - **preconditions** in the subclass must either stay the same or be **loosened**,
  - **postconditions** in the superclass must either stay the same or be **tightened**.

Design By Contract and LSP

- Suppose B is a subclass of A. Is the inheritance in the following examples valid?
  - A has a method with a precondition that it accepts integers between 0-50. B overrides the method to accept any integer between 0-100.
    - This is valid. Any input that satisfies the preconditions of A's method also satisfies the preconditions of B's method.
    - The other way around would be invalid!
  - A has a method with a postcondition that it returns a sorted list of integers. B overrides the method to return an unsorted list.
    - This is invalid, as an unsorted list is not a sorted list so the postcondition is not satisfied by B's method.
    - The other way around would be valid!

Live Example: Design by Contract

- In the **src/people**, there are a few classes which represent the people at a university.
  - Briefly discuss the preconditions and postconditions of the constructors, getters and setters in Person.java.
  - Fill in the preconditions and postconditions for setSalary in Person.java.
  - Discuss the validity of the subclasses of Person, and why they are/aren't valid subclasses.
  - Fix any issues identified.
  - [KEY TAKEAWAYS] Reasoning with preconditions and postconditions, exploring its relationship with inheritance