COMP2511

Tutorial 4

Last Week's Tutorial

- Testing
  - Writing JUnit tests
  - Using VSCode debugging tools
  - Exceptions
- Domain Modelling
  - System design before writing any concrete code
  - Creating UML diagrams

This Week’s Tutorial

- Identifying code smells and refactoring
  - SOLID principles, and the Law of Demeter/Principle of Least Knowledge
- Streams
  - Making our logic look nicer!
  - A taste of functional programming
- Design by contract
  - Contrasts ‘defensive programming’
  - Documenting preconditions and postconditions

Week 4 Reminders

- Assignment 1 is due next week Friday!
  - At this stage, you should ideally have finalised or be finalising the design of your solution to the problem.
  - **Please 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 please don't use this as an excuse to start the night before!
- Assignment 2 groups will be formed soon
  - Sorry about mentioning this while Assignment 1 is still ongoing...
  - Basically everyone has filled in the form, so thank you!

Code Smells

- What are **code smells**?
    - Features of code that are indicative of an inherent design problem.
    - Importantly, remember that they are **indicators** - they may not necessarily guarantee that something is wrong, and some potential design issues could be unavoidable depending on the scenario (but still acknowledge them!)
- What are some examples of code smells?
    - Duplicated code (maybe suggestive of a lack of inheritance usage)
    - Refused bequest (inherited fields/methods not making sense for specific subclasses/being widely unused, suggestive of a bad inheritance hierarchy)
    - Divergent change (needing to significantly change a class as changes are made)
    - Shotgun surgery (needing to make small changes to many different classes as changes are made)

SOLID Principles

- A set of good practices that you should always try to consider in this course.
- The intent is that following these practices will eliminate design smells, but **don’t overcook things when you don’t have to!** Your design could become needlessly complicated if you cook too hard.

SOLID Principles

- **S**ingle Responsibility Principle: Classes should have specific designated roles, and do only the things that they were created to do.
- **O**pen-Closed Principle: Software entities should be open for extension, but closed for modification.
- **L**iskov Substitution Principle: Subclasses should be valid instances of their parent classes.
- **I**nterface Segregation Principle: Classes should never implement interfaces that they will never use. (Separate functionality into interfaces!)
- **D**ependency Inversion Principle: Rely on high-level abstractions, rather than concrete implementations!

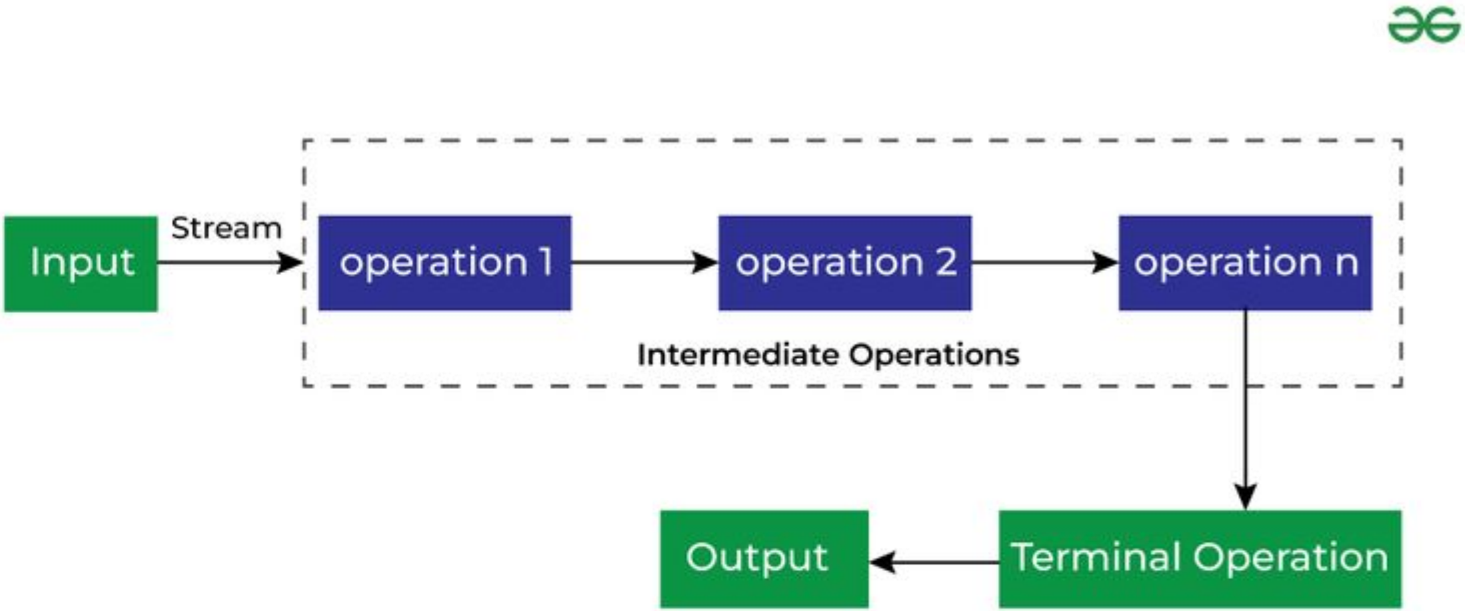
Code Example

src/training

Streams

- A **stream** (in Java) is an abstracted sequence of elements.
  - You won't be able to interact with the stream elements directly, but high-level abstractions have already been implemented to deal with them instead, and you can use those abstractions instead.
- A **pipeline** consists of a collection of elements (eg. a stream) and zero or more operations that operate on this collection.
  - You can think of this like function composition from maths, like (g . f) (x), where x is the collection, and f, g are functions acting on this collection.
- This is an idea borrowed from **functional programming**, where the main logic revolves around receiving an input and outputting the result of the composition of many functions together.

Streams


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

Code Example

src/stream

Defensive Programming

- You’ve probably been used to having to write code that accounts for a lot of different edge cases, like checking if something is null, if something is positive etc…
- This style of programming can be referred to as **defensive programming**.
- What are some pros of this approach?
- What are some cons of this approach?

Design By Contract

- **Design By Contract** refers to writing your code around a set of clear and well-defined specifications.
- Important terminology to consider (with respect to methods):
  - **Preconditions**: What are the assumptions this method can make?
  - **Postconditions**: What are the guarantees of what this method should be outputting?
  - **Invariants**: What things should not be changing as a result of executing this method?
- If someone defines a precondition for a specific method and it isn't satisfied when the method is executed, then too bad!
- What are some pros of this approach?
- What are some cons of this approach?

Code Example

src/calculator