# COMP2511

Tutorial 5

## Last Week's Tutorial

- Design Principles
  - SOLID Principles
  - Law of Demeter
- Introduction to Design Patterns
  - Composite Pattern
  - Factory Pattern

## This Week's Tutorial

- Streams
- More Design Patterns
  - Strategy Pattern
  - Observer Pattern

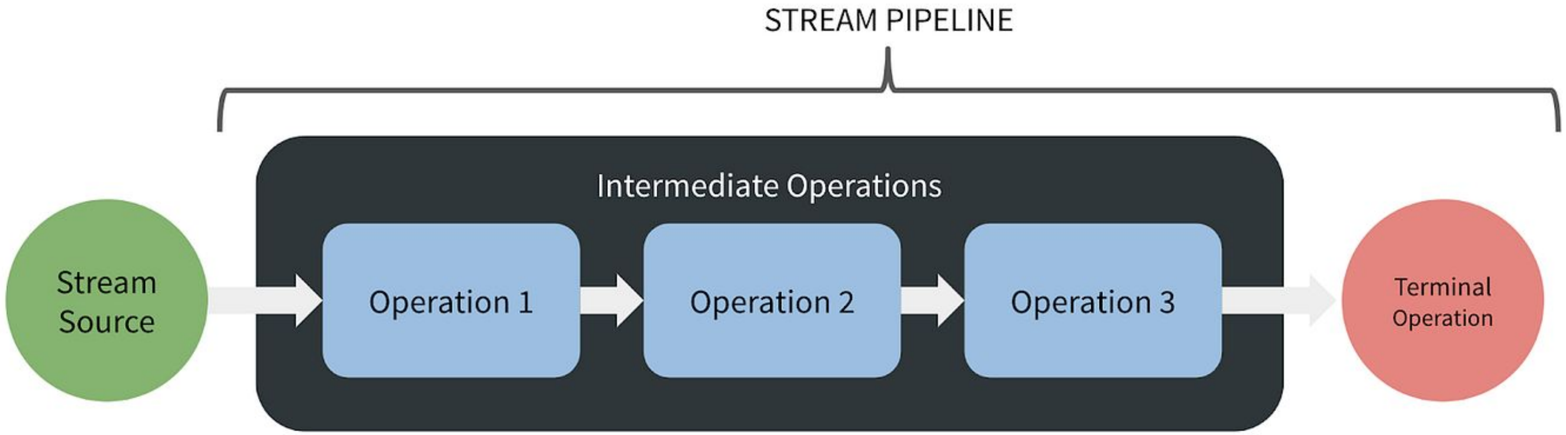
## Notices

- Assignment I is due **this Friday 3pm.**
    - Make sure that your code compiles on CSE (`2511 dryrun ass1`).
- Assignment II will be released this week.
    - Groups and channels on Teams have been created, please inform of any issues.
    - You are free to communicate on any platform that you prefer.
- Flex Week next week!
    - Try to find time to unwind so that you're ready for the next half of the term.
    - Lab 5 will be due on **Week 7 Monday**.
    - Spend time communicating with your partner to figure out a plan of attack for Assignment II according to each of your availabilities.

## Streams

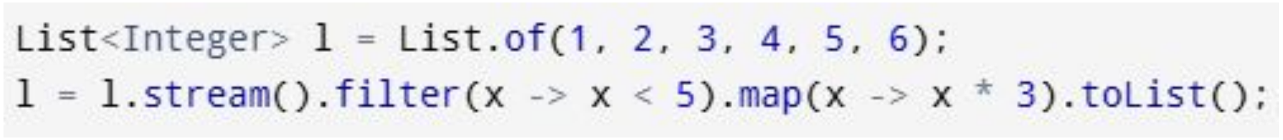
- Collections in Java (eg. `ArrayList`, `TreeMap`, `TreeSet`, `HashMap`) can be transformed into an *abstracted* sequence of the same elements, referred to as a **stream**.
- Elements in the stream can only be interacted with through methods that provide high-level abstractions to perform operations like `map`, `filter`, and `reduce`. These operations can be chained together.
- After all necessary operations have been done, a stream can be converted back into the original type of collection, as required.

## Streams

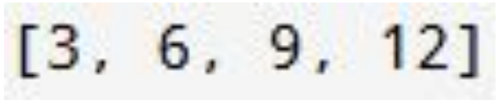

STREAM PIPELINE
Intermediate Operations
Stream Source
Operation 1
Operation 2
Operation 3
Terminal Operation

## Streams

- Suppose we have a list of integers and we want to take all of the numbers less than 5, then multiply all remaining numbers by 3. What sequence of stream operations could be used to achieve this?
- `filter`: take only elements that satisfy a specific condition (*predicate*)
  - `.filter(x -> x < 5)`
- `map`: apply a function to all elements in the collection
  - `.map(x -> x * 3)`
- `toList`: convert stream back into a list (returned list may be **immutable**)



```
List<Integer> l = List.of(1, 2, 3, 4, 5, 6);
l = l.stream().filter(x -> x < 5).map(x -> x * 3).toList();
```



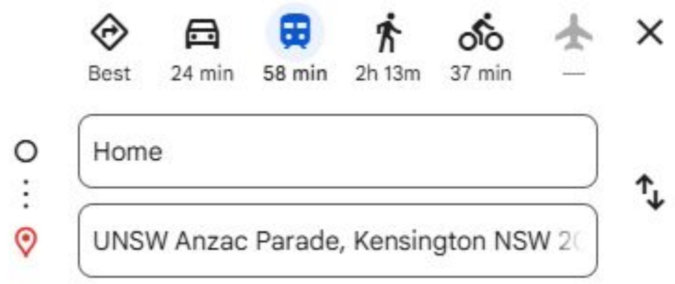
```
[3, 6, 9, 12]
```

# 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.

## Strategy Pattern

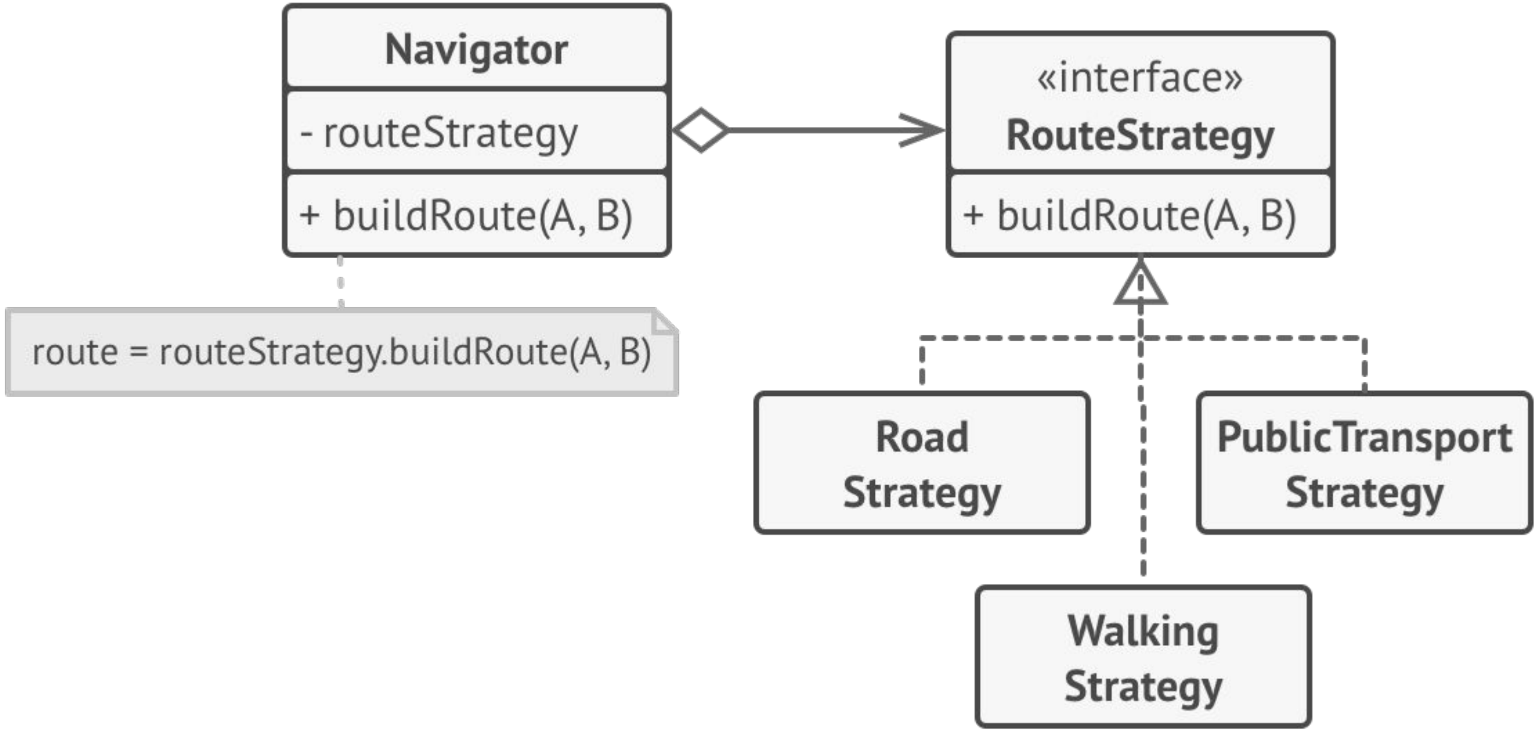
- Consider how you have multiple options on Google Maps for your mode of transport to get to a specific location - for example, you could walk, drive or take public transport.
    - Regardless of which mode of transport you use, your overall goal is to arrive at your destination. The specific route you have to take, the amount of distance you have to cover, the costs incurred etc. differ depending on which mode of transport you use.



## Strategy Pattern

- The Strategy Pattern is a (behavioural) design pattern where multiple implementations are defined for a specific action, and the class is able to switch between these implementations at **run-time**.
- Rather than putting all of the different implementations in a single class, each implementation is separated into different classes called **strategies**.
  - Each strategy should implement a specific **interface** (or inherit an abstract class) that defines the functionality that each strategy should be performing differently.
  - The original class then stores **instances** of these strategies, and executes instructions by calling the methods defined in the interface of the strategy.
  - Switching between strategies can be done through a setter.

# Strategy Pattern: Implementation



## Live Example: Strategy Pattern

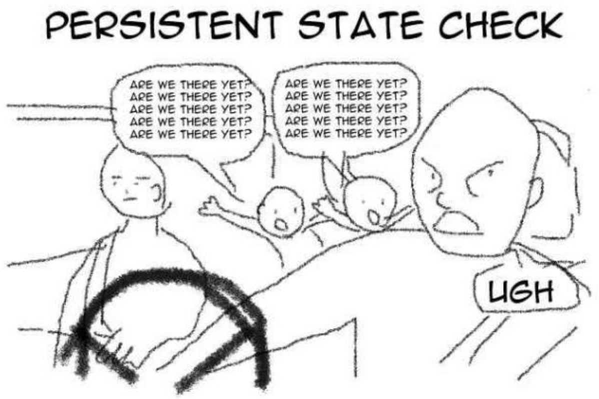
- Inside **src/restaurant** is a solution for a restaurant payment system with the following requirements:
  - The restaurant has a menu (a list of meals). Each meal has a name and price.
  - The user can enter their order as a series of meals, and the system returns their cost.
- The prices on meals often vary in different circumstances. The restaurant has three different price settings (so far):
  - **Standard** - normal rates
  - **Holiday** - 15% surcharge on all items for all customers
  - **Happy Hour** - registered members get a 40% discount, standard customers get 30%
- **Refactor the code to using the Strategy Pattern to handle the three settings**, then implement the **Prize Draw** Strategy - every 100th customer (since the start of the promotion) gets their meal for free!

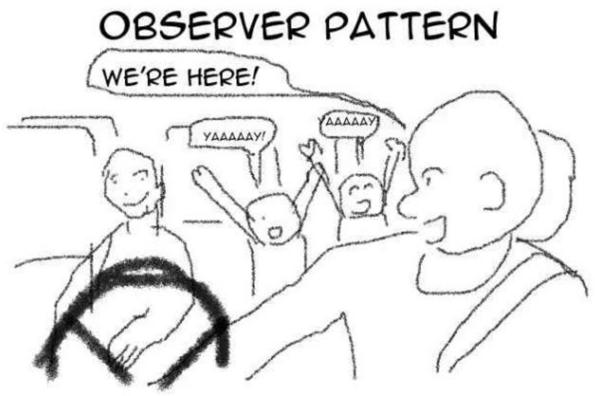
## Observer Pattern

- If you were subscribed to a YouTube channel and wanted to watch that channel's videos as soon as they came out, would you rather:
    - Keep refreshing the YouTube channel every 5 seconds to see if they have uploaded a new video?
    - **Receive a notification whenever a new video has been uploaded?**
- When a YouTuber publishes a new video, their subscribers get notified.
- We can frame this in a specific way: The subscribers wait on some important **event** (in this instance, a new video being uploaded), and when that event happens, some actions are performed to **respond** to it (a notification is sent).

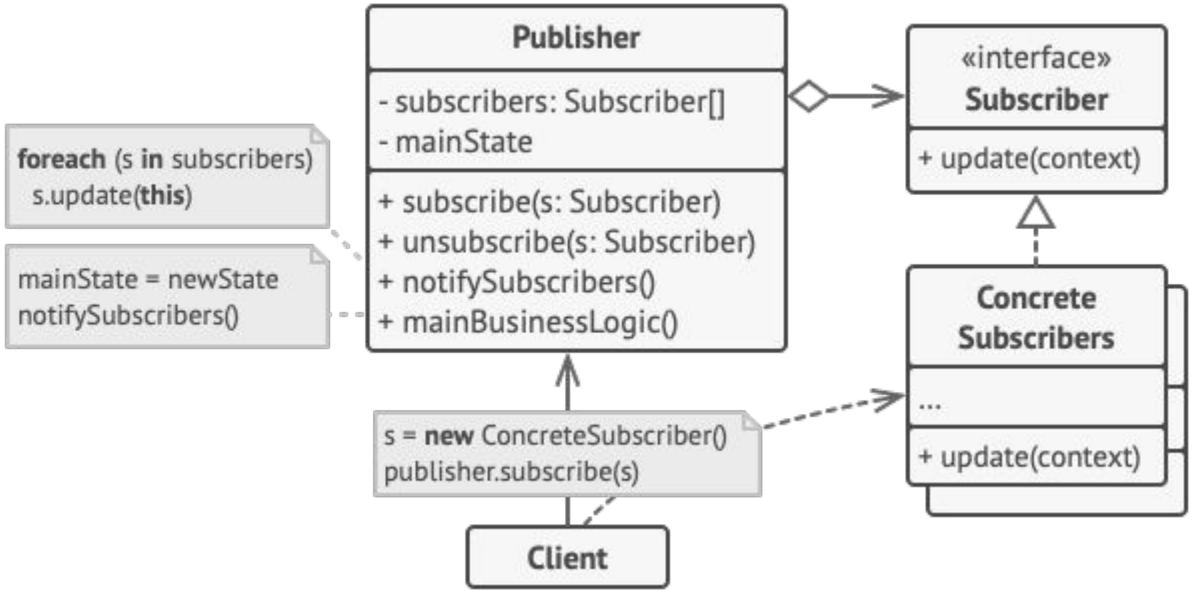
## Observer Pattern

- The observer pattern is a design pattern where a **subject** maintains a list of **observers**, who are **notified** when some event occurs.
    - Upon being notified, the observers will perform some action which is usually related to some sort of data that the subject sends out.


PERSISTENT STATE CHECK
ARE WE THERE YET? ARE WE THERE YET? ARE WE THERE YET? ARE WE THERE YET? ARE WE THERE YET?
ARE WE THERE YET? ARE WE THERE YET? ARE WE THERE YET? ARE WE THERE YET? ARE WE THERE YET?
UGH



## Observer Pattern: Implementation



## Live Example: Observer Pattern

- In **src/youtube**, create a model for the following requirements of a YouTube-like video creating and watching service using the Observer Pattern.
    - A YouTuber has a name, subscribers and videos.
    - A User has a name and can subscribe to any YouTuber.
    - When a YouTuber posts a new video, the video gets added to the 'Watch Later' lists of all Users subscribed to that YouTuber.
    - Write a simple test with print statements inside YoutubeTest.java.