Introduction to Systems Programming

Anonymous classes, Functional interfaces, Streams, Lambda functions

2025 - Simón Gutiérrez Brida *(Based on material by Dr. Nazareno Aguirre)*

But before… a quick demo from last week

Comparable, Comparable<T>, and Comparable<? super T>

From a simple maximum of Integer, to a maximum of Anything(an incomplete solution), a maximum of Circle, and a maximum of Anything (a complete solution).

And finally, how would the profile of an generic OrderedList would look like, and why.

Implementing a sort method

Let's use selection sort to keep it simple

What if we want a different ordering of our elements (like, sort in reverse order), what can we do?

Anonymous classes

“You cannot have instances of Interfaces or Abstract classes”, this is true.

But you can have an Anonymous class implementing an Interface or Abstract class.

- What is an Anonymous class?

- Why would you want one?

Anonymous classes

“You cannot have instances of Interfaces or Abstract classes”, this is true.

But you can have an Anonymous class implementing an Interface or Abstract class.

- What is an Anonymous class?
  - It’s a class without a given name.

- Why would you want one?
  - When an abstract class or interface requires few methods and you don’t want to introduce new classes that will only be used once.

Implementing a generic sort method

Let’s use selection sort to keep it simple

Now we want to sort elements according to a given ordering.

Limitations of anonymous classes

- An anonymous class is actually syntactic sugar, a class will be generated, only that it will not have a “good” name, e.g.: “Main$1”.
- Since a new class is being created, then you are not really creating an instance of an interface or abstract class.
- If used wrongly, can make your code unreadable.

Functional Interfaces

- It’s an annotation (@FunctionalInterface) for an interface.

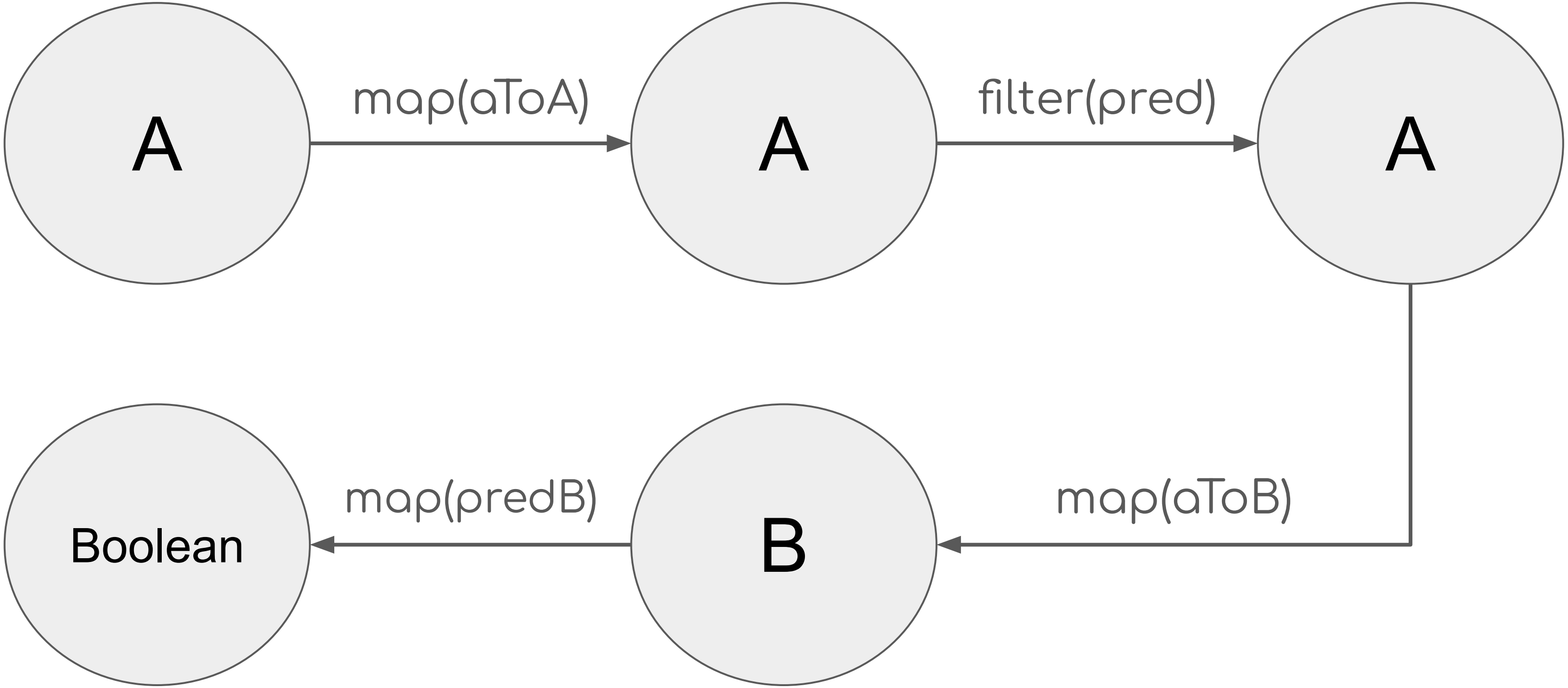
- It can only be used for interfaces with only one abstract method (*).

- It allows instances of an interfaces to be created with:
  - Lambda expressions.
    - (String s) -> {return s;} could be used for any FunctionalInterface with an abstract method that takes a String and returns a String.

  - Method references.
    - Integer::valueOf (a method reference) could be used for any FunctionalInterface that takes a String and returns an Integer.

  - Constructor references.
    - String::new (a constructor reference) could be used for any Functional Interface that takes a String and returns a String.

Collections from Collections



Why Streams if we can use Collections?

- To go from one Collection to another, we need to go over all elements in the source collection.

- Even though the functions/predicates/actions applied to elements on a collection might only depend on the current element, they are applied sequentially.

- Streams work in a Producer-Consumer fashion, the resulting stream can be used while not all elements in the source streams are consumed.

- Streams allow for parallel application of functions/predicates/actions.
  - Although it will not always be faster (there is an overhead for parallelism)

Why Streams if we can use Collections?

- Must be careful with streams when the order of evaluation/generation is important (specifically if you will use a parallel stream).

- Overuse can make your code become unreadable.

- In Java one needs to remember several Classes/Methods names, languages like Haskell (which is fully functional) are much better suited to work with streams of data.

Lambdas

- Lambdas allows for anonymous functions.

- Very useful when those functions are short.

- The format is: (list-arguments) -> {body}
  - Where list-arguments can be empty, can only contain symbol names, or can be of the form Type name.

  - If the function is meant to return then the body must end with a return statement.

- Overuse can make your code be unreadable.