

PG4200: Algorithms And Data Structures

Lesson 07: Iterators, Lambdas and Streams

Prof. Andrea Arcuri

Iterators

Iterating Over All Elements

- Many situations in which you need to look at and process all elements in a collection
- For *lists* and *arrays*, can look at indices from 0 to N-1
- But what about *maps* and *sets* that are unordered?
 - or other kinds of data structures like *graphs* that we haven't seen yet
- Would like to write loops like “*for*(X *x* : *collection*){...}”

java.util.Iterator

```
public interface Iterator<E> {  
    /**  
     * Returns {@code true} if the iteration has more elements.  
     * (In other words, returns {@code true} if {@code #next} would  
     * return an element rather than throwing an exception.)  
     *  
     * @return {@code true} if the iteration has more elements  
     */  
    boolean hasNext();  
  
    /**  
     * Returns the next element in the iteration.  
     *  
     * @return the next element in the iteration  
     * @throws NoSuchElementException if the iteration has no more elements  
     */  
    E next();
```

java.lang.Iterable

```
public interface Iterable<T> {  
    /**  
     * Returns an iterator over elements of type {@code T}.  
     *  
     * @return an Iterator.  
     */  
    Iterator<T> iterator();  
}
```

Iterators

- The idea is to make our collections to implement *Iterable* interface
- Need to implement an iterator for each collection, which keeps track of *one element at a time*
- Java compiler is aware of *Iterable*, and so can automatically handle “*for(X x : collection){...}*”
 - ie, do not need to call “*hasNext()*” and “*next()*” by yourself
 - but only as long as that collection does implement *Iterable*
- Collection should not be changed (eg, add/remove) while iterating over them

Lambdas

Functions as Parameters

- At times, you need to pass “*code*” as input parameter to another method
- To do that, you need to create a class with a method implementing the code you need
- Writing a whole class definition for just a single line of code is too much boilerplate

Interfaces in *java.util.function*. *

- *Runnable.run()*
 - Nothing as input/output, just execute some code with side-effects
- *Consumer<T>.accept(T t)*
 - take an instance of T as input, and do something with it, and return nothing
- *Predicate<T>.test(T t)*
 - input T, and then return a boolean
- *Function<T,R>.apply(T t)*
 - take T as input, and return something of type R
- There are more, but those 4 are enough for what we need

Anonymous Classes

- Because interfaces, need to create concrete instances with our implementations
- If used only once, can create class on the fly
- Note that *Consumer* is an interface with method *accept()*

```
Consumer<String> anonymousClass = new Consumer<String>() {  
    @Override  
    public void accept(String s) {  
        //your code here  
    }  
};
```

Lambdas

- Anonymous classes work fine, but are tedious to write
- *Lambdas*: syntax sugar to reduce boilerplate
- Java compiler is aware of the interfaces in *java.util.function.**

```
Consumer<String> lambda = s -> { /* your code */ }  
//this is equivalent to previous example
```

$() \rightarrow \{ \}$

- Left-side: the input parameter name(s), with **()** if none
 - can choose the names you want, but usually a single letter
- Right-side: the instruction to execute. If more than one, should be in a **{ }** block
- Based on input/output types, the compiler will automatically create the right class, eg *Runnable*, *Consumer*, *Predicate* or *Function*

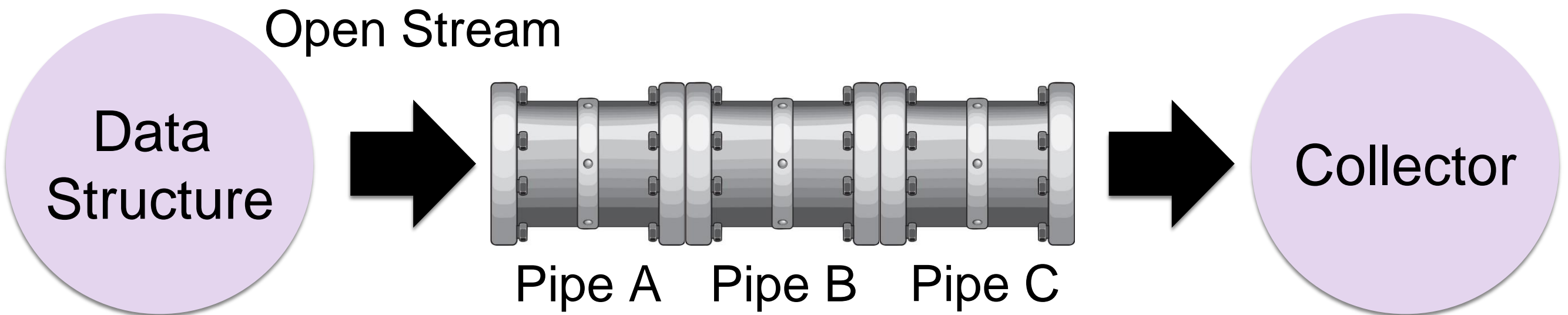
Streams

Iterators + Lambdas

- When a container has an iterator, and once we can write custom code in lambda expressions, we can introduce the concept of **Stream**
- The idea is that we can iterate over all elements in collection, and *easily* execute code in sequence on each of the elements
- *This can drastically reduce the amount of code you write, and easier to understand (once you get familiar with it)*

Stream / Pipeline

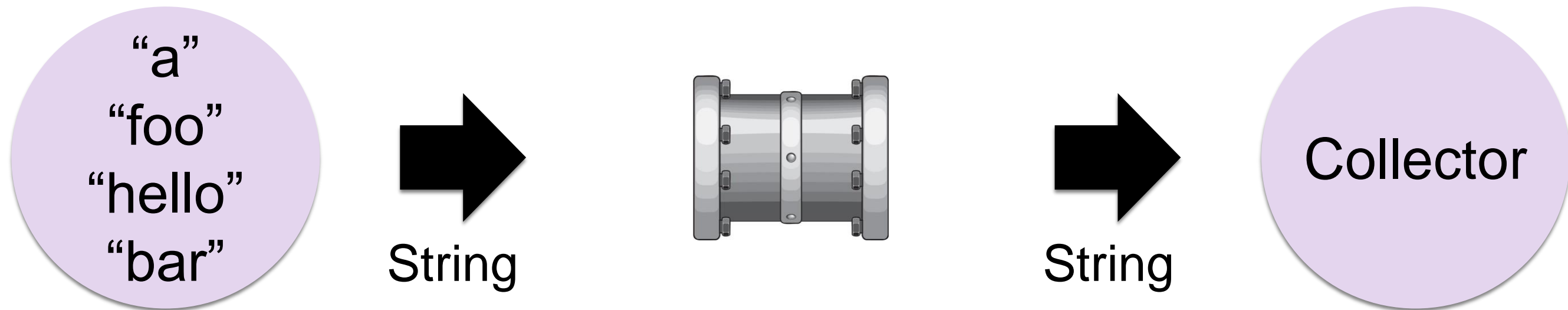
- At each pipe, the elements can be transformed and/or blocked (ie not going to next pipe)
- At the end of the stream, we need a *collector*, which defines what to do with elements that arrive at the end of the pipeline



Type of Pipes

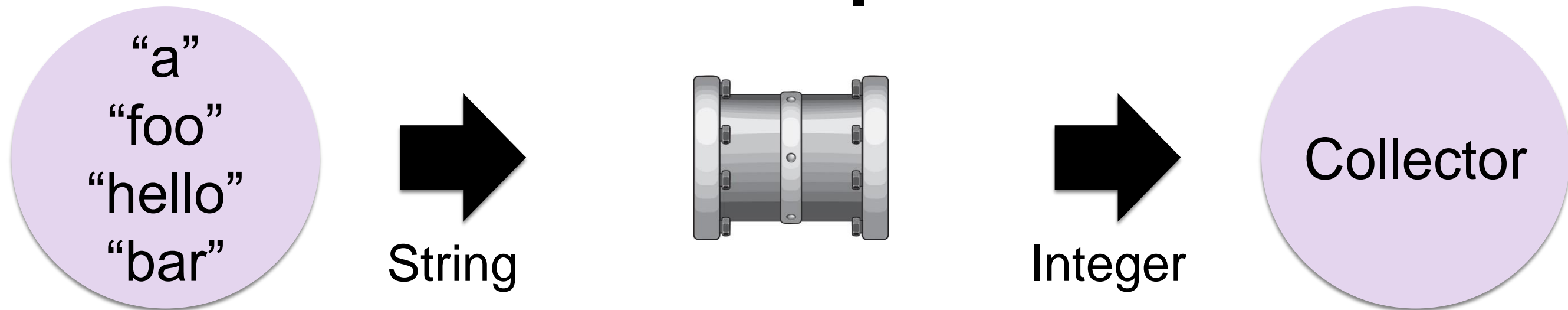
- *Filter*: take as input a *Predicate* $\langle T \rangle$, and based on that decide if elements propagate to next pipe
- *Map*: transform input, and also change type, based on a *Function* $\langle T, R \rangle$
- *FlatMap*: get a stream from input element, and flatten it into the current stream (examples later)
- There are more, but these are the main ones we will see in details

Filter



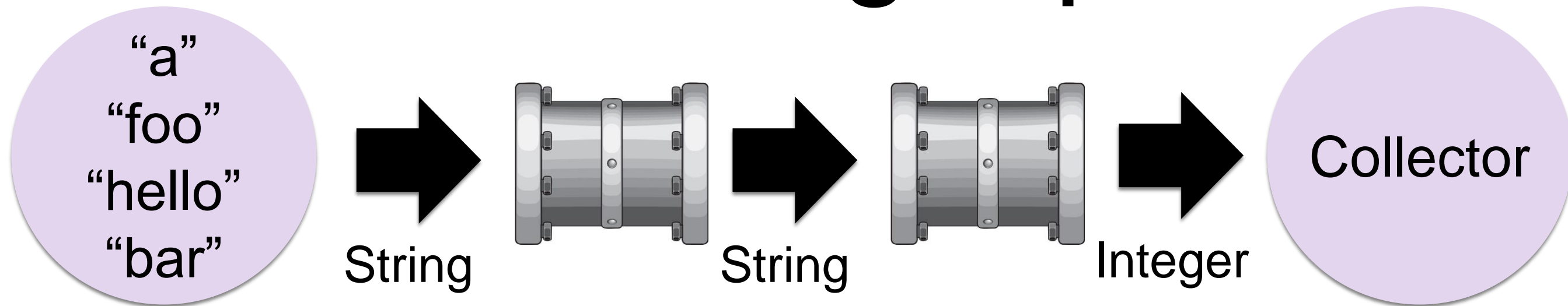
- Example: `collection.stream().filter(s -> s.length() > 3)`
- Input/Output: *String*, does not change
- Blocked: "a", "foo", "bar"
- Allowed: "hello"
- Collector will only get "hello"

Map



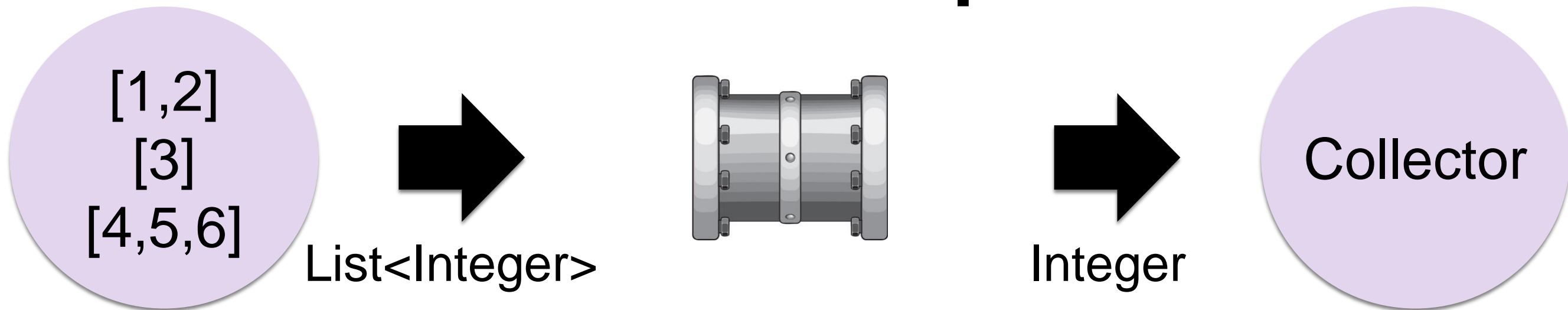
- Example: `collection.stream().map(s -> s.length())`
- Input *String*, Output *Integer*
- Note: compiler automatically infer type "*Integer*" based on the type returned by the function "*String.length()*"
- Collector will receive: 1, 3, 4, 3

Combining Pipes



- Example: `.filter(s -> s.length() > 2).map(s -> s.length())`
- "a" is the only blocked element by the filter
- Collector will receive: 3, 4, 3

FlatMap



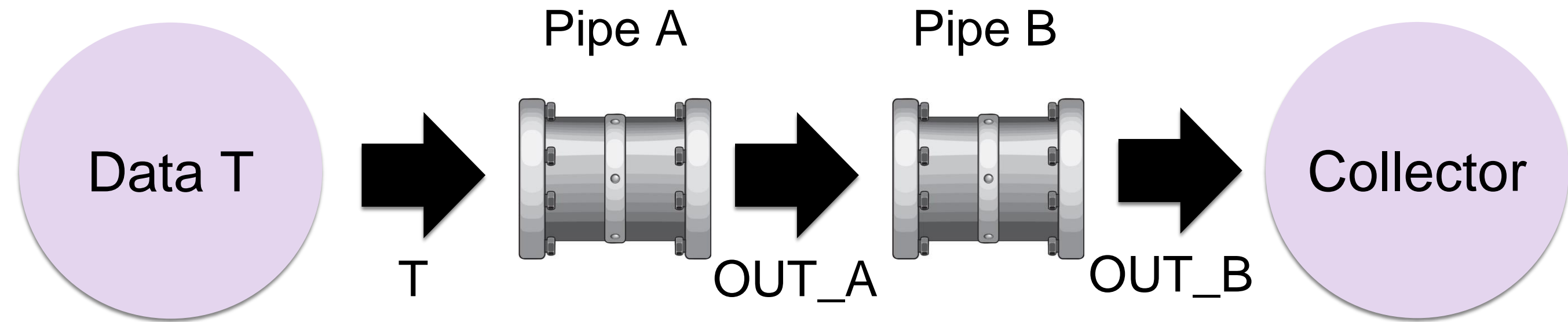
- Example: `collection.stream().flatMap(l -> l.stream())`
- Input `List<Integer>`, Output `Integer`
- On each of the 3 input lists we open a stream, and propagate its output, one element at a time
- Collector will receive: 1, 2, 3, 4, 5, 6 and NOT `[1,2]`, `[3]`, `[4,5,6]`
- So the values of the 3 lists are flattened into a single stream of integers, including all values in those lists

Collectors

- *collectToList()*: each element that arrives at the end of the pipeline will be added to a new List container
- *forEach(Consumer<X> action)*: for each element that arrives at the end of the pipeline, execute the action specified by the user, which takes as input type X being the output of the last pipe in the pipeline
- There are more, but these are the main ones we will see in details

Implementation

- The collector is the one that starts pulling data from the collection using its iterator
- At each step, the output of a pipe is going to be the input to the next pipe
- Going to represent it with a *chain* of Consumers



- IN_A is T , and IN_B is OUT_A
- Collector has a *Consumer* $\langle OUT_B \rangle$, as it consumes data from Pipe B
- Pipe B has a *Consumer* $\langle OUT_A \rangle$, which will call *Consumer* $\langle OUT_B \rangle$ in Collector (ie, chained)
- Pipe A has a *Consumer* $\langle T \rangle$, which will call *Consumer* $\langle OUT_A \rangle$ of Pipe B (ie, chained)

Collector Call

- Get iterator from collection of type T
- On each element of type T, call consumer of the first pipe



Pipe A: Consumer<T>

- Compute output of type OUT_A
- Call consumer of Pipe B with such output



Pipe B: Consumer<OUT_A>

- Compute output of type OUT_B
- Call consumer of Collector with such output



Collector: Consumer<OUT_B>

- Do its computation
- End of the chain

- The collector is what starts the stream by pulling data
- The consumer of the first pipe is called, and that will trigger a chain until the last consumer
- However, the collector itself has to define a Consumer for the last pipe

Homework

- No Book Chapter
- Study code in the *org.pg4200.les07* package
- Do exercises in *exercises/ex07*