EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 22: November 10

Lecturer: Vijay Garg *Scribe: Zihan Yang*

Agenda

This lecture is a general introduction of Stream, a new abstract layer in Java 8:

- Stream basics
- Stream and Collection
- Simple examples of Stream
- 2 puzzles

22.1 Introduction

A stream is a sequence of elements supporting sequential and parallel aggregate operations.

To perform a computation, stream operations are composed into a stream pipeline. A stream pipeline consists of a source (which might be an array, a collection, a generator function, an I/O channel, etc), zero or more intermediate operations (which transform a stream into another stream, such as filter(Predicate)), and a terminal operation (which produces a result or side-effect, such as count() or forEach(Consumer)).

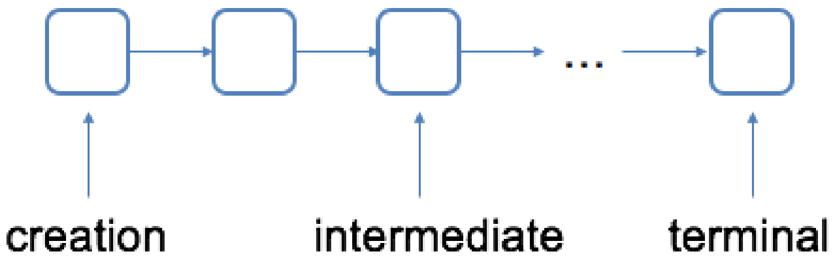


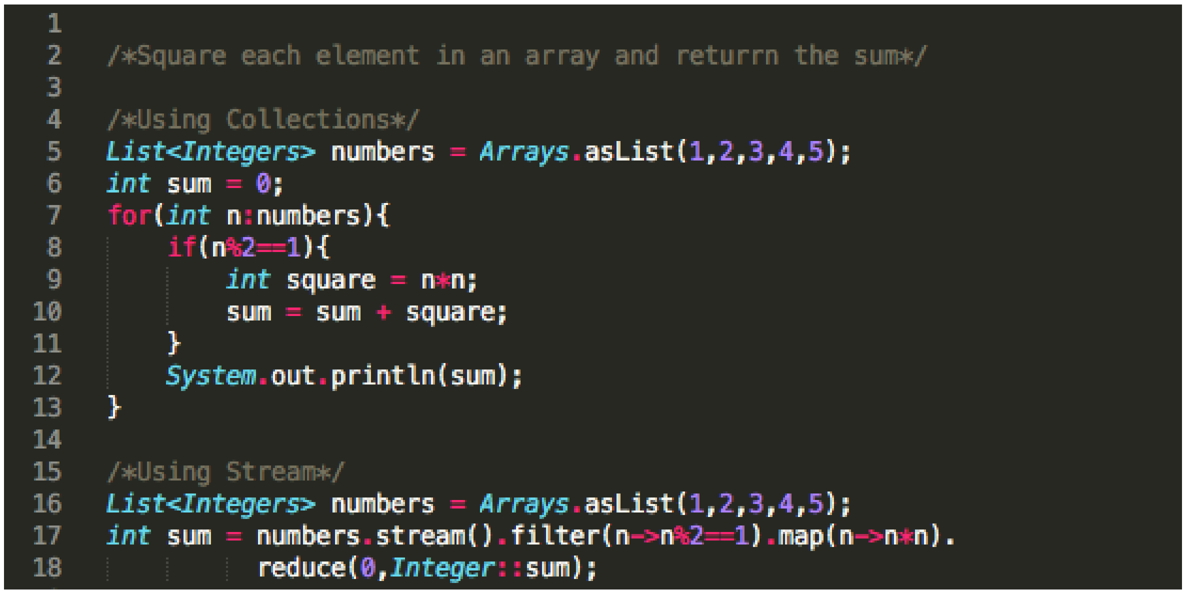
Figure 22.1: Stream in pipeline fashion

Stream pipelines may execute either sequentially or in parallel. This execution mode is a property of the stream. Streams are created with an initial choice of sequential or parallel execution. For example, Collection.stream() creates a sequential stream, and Collection.parallelStream() creates a parallel one. Operations on a sequential stream are processed in serial by one thread. Operations on parallel stream stream are processed in parallel by multiple threads. Most of the methods in the Streams API produce sequential streams by default.

22.2 Stream basics

Stream	Collection
Concept of time	Concept of space
Use it only once	Use it any time
Focus on aggregate operations	Focus on how to store and access elements
Number of elements could be infinite	Always finite
Lazy evaluation	All the elements are always there

In order to compare stream with collection, Fig 22.2 shows a snippet of code in which we square every odd element in an array and return its sum using collection and stream. With the help of stream, we can implement it just in one line. By contrast, the collection implementation is much longer.



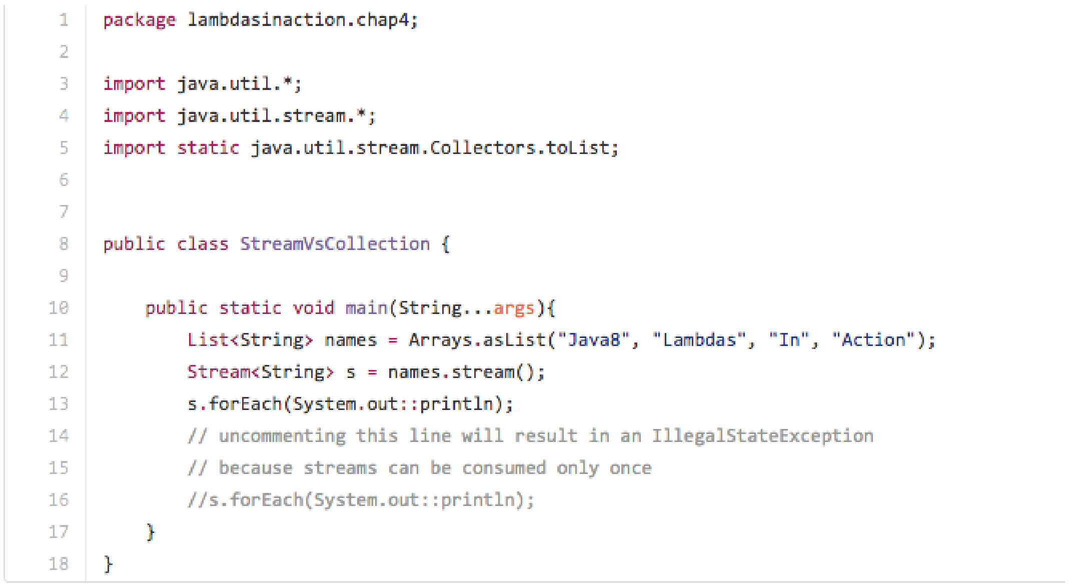
```java

/*Square each element in an array and returrn the sum*/

/*Using Collections*/
List<Integers> numbers = Arrays.asList(1,2,3,4,5);
int sum = 0;
for(int n:numbers){
    if(n%2==1){
        int square = n*n;
        sum = sum + square;
    }
    System.out.println(sum);
}

/*Using Stream*/
List<Integers> numbers = Arrays.asList(1,2,3,4,5);
int sum = numbers.stream().filter(n->n%2==1).map(n->n*n).
            reduce(0,Integer::sum);
```

Figure 22.2: Comparison between Collection and Stream

Even though Streams and Collections seem to share some superficial similarities. They actually serve for different goals. Collections are primarily concerned with the efficient management of, and access to, their elements. By contrast, streams do not provide a means to directly access or manipulate their elements and are instead concerned with declaratively describing their source and the computational operations which will be performed in aggregate on that source.

Besides, streams are not reusable. A stream can't be reused after calling a terminal operation. Look at the code in Fig 22.3, everything works well in line 13, but the line 16 should be commented because the streams can only be consumed once. However, all the elements in collections are always there and could be operated at any time. Streams are lazy because computation on the source data is only performed when the terminal operation is initiated, and source elements are consumed only as needed. To some extend, stream is more like a concept of time while collection is of concept of space.

Another difference is that a collection cannot represent a group of infinite elements whereas a stream can. A stream can pull its elements from a data source. The source can be a collection, an I/0 channel or a function which can generate infinite number of elements.



```java
package lambdasinaction.chap4;

import java.util.*;
import java.util.stream.*;
import static java.util.stream.Collectors.toList;


public class StreamVsCollection {

    public static void main(String...args){
        List<String> names = Arrays.asList("Java8", "Lambdas", "In", "Action");
        Stream<String> s = names.stream();
        s.forEach(System.out::println);
        // uncommenting this line will result in an IllegalStateException
        // because streams can be consumed only once
        //s.forEach(System.out::println);
    }
}
```

Figure 22.3: A stream is not resuable

22.3 Simple examples of Stream

22.3.1 Java Stream Create

We can create stream in the following ways:

- Create Streams from values
- Create Streams from Empty streams
- Create Streams from functions
- Create Streams from arrays
- Create Streams from collections
- Create Streams from files
- Create Streams from other sources

For detailed descriptions and examples, please refer to: `http://www.java2s.com/Tutorials/Java/Java_Stream/0040__Java_Stream_Create.htm`

22.3.2 Java Stream opearations

There are two types of operations:terminal and intermediate. The commonly used stream operations are listed as follows:

- Distinct(intermediate): Returns a stream consisting of the distinct elements by checking equals() method.

- Filter(intermediate): Returns a stream that match the specified predicate.
- FlatMap(intermediate): Produces a stream flattened.
- Limit(intermediate): truncates a stream by number.
- Map(intermediate): Performs one-to-one mapping on the stream
- Peek(intermediate): Applies the action for debugging.
- Skip(intermediate): Discards the first n elements and returns the remaining stream. If this stream contains fewer than requested, an empty stream is returned.
- Sorted(intermediate): Sort a stream according to natural order or the specified Comparator. For an ordered stream, the sort is stable.
- allMatch(terminal): Returns true if all elements in the stream match the specified predicate, false otherwise. Returns true if the stream is empty.
- anyMatch(terminal): Returns true if any element in the stream matches the specified predicate, false otherwise. Returns false if the stream is empty.
- findAny(terminal): Returns any element from the stream. Returns an empty Optional object for an empty stream.
- findFirst(terminal): Returns the first element of the stream. For an ordered stream, it returns the first element; for an unordered stream, it returns any element.
- noneMatch(terminal): Returns true if no elements in the stream match the specified predicate, false otherwise. Returns true if the stream is empty.
- forEach(terminal): Applies an action for each element in the stream.
- reduce(terminal): Applies a reduction operation to computes a single value from the stream.

For detailed descriptions and examples, please refer to: `http://www.java2s.com/Tutorials/Java/Java_Stream/0200__Java_Stream_Operations.htm`

In the class, Prof.Garg walked us through some of the operations and concepts of streams using the examples on the github: `https://github.com/java8/Java8InAction`, including:

- syntax of **lambda function**(`https://github.com/java8/Java8InAction/blob/master/src/main/java/lambdasinaction/chap3/Lambdas.java`)
- **filter** operation(`https://github.com/java8/Java8InAction/blob/master/src/main/java/lambdasinaction/chap1/FilteringApples.java`)
- **sort** operation(`https://github.com/java8/Java8InAction/blob/master/src/main/java/lambdasinaction/chap3/Sorting.java`)
- **reduce** operation(`https://github.com/java8/Java8InAction/blob/master/src/main/java/lambdasinaction/chap5/Reducing.java`)
- mechanism of **lazy evaluation** of stream(`https://github.com/java8/Java8InAction/blob/master/src/main/java/lambdasinaction/chap5/Laziness.java`)
- **parallel streams**(`https://github.com/java8/Java8InAction/blob/master/src/main/java/lambdasinaction/chap7/ParallelStreams.java`)

References

[1] Tutorials on Java Streaming, `http://www.java2s.com/Tutorials/Java/Java_Stream/index.htm`

[2] Java8InAction on github, `https://github.com/java8/Java8InAction/tree/master/src`

[3] Documentation of Java stream on Oracle Help Center, `https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html`