COMP 322: Parallel and Concurrent Programming

Lecture 5: Streams

Zoran Budimlić and Mack Joyner
{zoran, mjoyner}@rice.edu

http://comp322.rice.edu

Lazy lists

```java
class LazyCons<T> implements LazyList<T> {
    final T head;
    final Lazy<LazyList<T>> tail;

    LazyCons(T head, Supplier<LazyList<T>> tail) {
        this.head = head;
        this.tail = Lazy.of(tail);
    }

    public T head() {
        return head;
    }

    public LazyList<T> tail() {
        return tail.get();
    }
```

a lambda that will return the tail

Build a memo around the tail supplier so that we only call the lambda once

tail() hides the implementation details

From laziness to parallelism: Java Streams

Generalizing the laziness concept to arbitrary collections of objects

Idea:

- Take a bunch of objects
- Turn them into a Stream (a lazy representation)
- Perform a series of lazy operations on them (all running in constant time)
- Eventually, compute the final result of your computation, which triggers evaluation of ***only*** of those lazy operations necessary to compute your result

Operations on Java Streams can be executed in parallel!!!

Creating Streams

Empty Stream:

```
Stream<String> streamEmpty = Stream.empty();
```

Stream from a collection:

```
Collection<String> collection = Arrays.asList("a", "b", "c");
Stream<String> streamOfCollection = collection.stream();
```

Stream from an array:

```
Stream<String> streamOfArray = Stream.of("a", "b", "c");
String[] arr = new String[]{"a", "b", "c"};
Stream<String> streamOfArrayFull = Arrays.stream(arr);
Stream<String> streamOfArrayPart = Arrays.stream(arr, 1, 3);
```

Using *Stream.builder()*:

```
Stream<String> streamBuilder =
    Stream.<String>builder().add("a").add("b").add("c").build();
```

Creating infinite Streams

Using *Stream.generate()*. Infinite Stream of strings “element”:

```
Stream<String> streamGenerated = Stream.generate(() -> "element");
```

Using *Stream.iterate()*. Infinite Stream of even Integers, starting with 40:

```
var streamIterated = Stream.iterate(40, n -> n + 2); // Stream<Integer>
```

Take a finite number of elements from an infinite stream. Just like our *LazyList take():*

```
var tenStrings = streamGenerated.limit(10); // Stream<String>. Runs in constant time
var fiveInts = streamIterated.limit(5); // Stream<Integer>. Runs in constant time
```

Still lazy!

Streams of primitive types

Stream<T> cannot be used for primitive types

Instead, use *IntStream*, *LongStream* and *DoubleStream* for streams of *ints*, *longs* and *doubles*

```
IntStream intStream = IntStream.range(1, 3); // IntStream of (1, 2)
LongStream longStream = LongStream.rangeClosed(1, 3); // LongStream of (1, 2, 3)
```

Using *Random*:

```
Random random = new Random();
DoubleStream doubleStream = random.doubles();   // Infinite DoubleStream of random double numbers
var fiveIntsStream = random.ints(5);            // IntStream of five random int numbers
var alsoFiveIntsStream = random.ints().limit(5);// IntStream of five random int numbers
```

Other ways to create Streams

Stream from a *String:*

```
IntStream streamOfChars = "abc".chars();
Stream<String> streamOfString =
    Pattern.compile(", ").splitAsStream("a, b, c");
```

Stream of lines of *Strings* from a *File* (`java.nio.file.*`):

```
Path path = Paths.get("C:\\file.txt");
Stream<String> streamOfStrings = Files.lines(path);
Stream<String> streamWithCharset =
    Files.lines(path, java.nio.charset.Charset.forName("UTF-8"));
```

Stream pipeline

```java
List<String> list = Arrays.asList("Rice", "Owls", "are", "the", "best");
long size =
    list.stream()
        .skip(1)
        .map(element -> element.substring(0, 3))
        .filter(element -> element.charAt(2) == 'e')
        .sorted()
        .count();
```

Stream ***source***

Intermediate operations
All lazy!

Terminal operation

Intermediate operations. Lazy!

`filter(p)`	Keep only elements satisfying the given Predicate *p*
`map(f)`	Apply the given function *f* to all elements
`flatMap(f)`	Like map, but when result of *f* is a stream. Final result is flattened
`distinct()`	Unique elements of the stream (w.r.t. *Object.equals(Object)*)
`sorted(c)`	Elements of the stream, sorted according to Comparator *c*
`peek(a)`	Perform the Consumer action *a* on all elements, return original Stream
`limit(n)`	Take first *n* elements
`skip(n)`	Discard the first *n* elements

Intermediate operations are lazy

What will this print?

```
List<String> list = Arrays.asList("Rice", "Owls", "are", "the", "best");
Stream<String> stream =
    list.stream()
        .filter(e -> {
          System.out.println("Predicate was called on " + e);
          return e.contains("e");});
```

Nothing!

Terminal operations. Drive the computation!

`reduce(zero, f)`	Just like our fold. Start with accumulator *zero*, apply *f* to all the elements of the stream
`toArray()`	Produce an array from elements of the result Stream
`collect()`	Collect the elements of the result Stream into an object (usually a Java Collection)
`count()`	Counts the elements in the result Stream
`forEach(a)`	Perform Consumer action *a* on all elements
`forEachOrdered(a)`	Same as forEach, but in order of the stream, if ordered (i.e. with *sorted*())
`min(c), max(c)`	Minimum/maximum element, according to the Comparator *c*
`(any)(all)(none)Match(p)`	True if any/all/none elements of the stream match Predicate *p*
`findFirst()`	Pick the first element of the result stream
`findAny()`	Pick any element of the result stream

All of these are just special cases of reduce()!

Computation is driven by terminal operations

```java
List<String> list = Arrays.asList("Rice", "Owls", "are", "the", "best");
Optional<String> value =
    list.stream()
        .filter(e -> {
          System.out.println("Filter was called on " + e);
          return e.contains("s");})
        .map(e -> {
          System.out.println("Map was called on " + e);
          return e.toUpperCase();})
        .findFirst();
System.out.println(value.get());
```

```
Filter was called on Rice
Filter was called on Owls
Map was called on Owls
OWLS
```

Ordering matters

```java
List<String> list =
    Arrays.asList("Rice", "Owls", "are", "the", "best");
Optional<String> value =
    list.stream()
        .filter(e -> {
          System.out.println("Filter was called on " + e);
          return e.contains("s");})
        .map(e -> {
          System.out.println("Map was called on " + e);
          return e.toUpperCase();})
        .findFirst();
System.out.println(value.get());
```

```
Filter was called on Rice
Filter was called on Owls
Map was called on Owls
OWLS
```

```java
List<String> list =
    Arrays.asList("Rice", "Owls", "are", "the", "best");
Optional<String> value =
    list.stream()
        .map(e -> {
          System.out.println("Map was called on " + e);
          return e.toUpperCase();})
        .filter(e -> {
          System.out.println("Filter was called on " + e);
          return e.contains(“S");})
        .findFirst();
System.out.println(value.get());
```

```
Map was called on Rice
Filter was called on RICE
Map was called on Owls
Filter was called on OWLS
OWLS
```

Parallel Streams!

```
List<String> list = Arrays.asList("Rice", "Owls", "are", "the", "best");
Optional<String> value =
    list.stream().parallel()
        .filter(e -> {
          System.out.println("Filter was called on " + e);
          return e.contains("s");})
        .map(e -> {
          System.out.println("Map was called on " + e);
          return e.toUpperCase();})
        .findFirst();
System.out.println(value.get());
```

```
Filter was called on are
Filter was called on Owls
Map was called on Owls
Filter was called on Rice
Filter was called on the
OWLS
```

```
Filter was called on are
Filter was called on Owls
Map was called on Owls
Filter was called on Rice
OWLS
```

```
Filter was called on are
Filter was called on Owls
Filter was called on best
Filter was called on Rice
Map was called on Owls
Filter was called on the
Map was called on best
OWLS
```

Parallel Streams

Stream.parallel(): convert a sequential Stream into a parallel one

- Changes the mode of execution of lazy operations (literally just sets a flag in Stream)
- Java may perform the intermediate and terminal operations on it in parallel
- No guarantee of parallel execution, nor the amount of parallelism
- No ordering on operations on elements can be assumed
- If your ***source*** is a Collection, you can use *Collection.parallelStream()* instead

Stream.sequential(): convert a parallel Stream into a sequential one

- Changes the mode of execution to sequential

Reductions (*Stream<T>*)

```
Optional<T> reduce(BinaryOperator<T> accumulator)
```

- Works when the elements of the stream and the result of the reduction are of the same type
- *accumulator* needs to be associative, stateless, non-interfering (does not modify the source of the stream)
- Result is empty if the stream has no elements

```
T reduce(T identity, BinaryOperator<T> accumulator);
```

- Elements of stream and the result of same type, accumulator is associative, stateless and non-interfering
- *identity* should be the real identity element for the *accumulator* (strange results when running in parallel otherwise)

```
<U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner);
```

- General form, works when the result of the reduction is of a different type than the elements of the stream
- *combiner* should be able to combine results of two partial reductions into one
- *combiner* and *accumulator* should be associative, stateless, non-interfering
- *identity* should be the real identity element for the *combiner* (strange results when running in parallel otherwise)

Always use the real identity

```java
String seqString =
    Stream.of("Rice ", "Owls ", "are ", "the ", "best")
        .reduce("HI ", String::concat, (a, b) -> {
          System.out.println("Sequential combiner was called");
          return a.concat(b);
        });
String parString =
    Arrays.asList("Rice ", "Owls ", "are ", "the ", "best").parallelStream()
        .reduce("HI ", String::concat, (a, b) -> {
          System.out.println("Parallel combiner was called");
          return a.concat(b);
        });
System.out.println("Sequential result: " + seqString);
System.out.println("Parallel result: " + parString);
```

```
Parallel combiner was called
Parallel combiner was called
Parallel combiner was called
Parallel combiner was called
Sequential result: HI Rice Owls are the best
Parallel result: HI Rice HI Owls HI are HI the HI best
```

Always use the real identity

```java
String seqString = "HI " +
    Stream.of("Rice ", "Owls ", "are ", "the ", "best")
        .reduce("", String::concat, (a, b) -> {
          System.out.println("Sequential combiner was called");
          return a.concat(b);
        });
String parString = "HI " +
    Arrays.asList("Rice ", "Owls ", "are ", "the ", "best").parallelStream()
        .reduce("", String::concat, (a, b) -> {
          System.out.println("Parallel combiner was called");
          return a.concat(b);
        });
System.out.println("Sequential result: " + seqString);
System.out.println("Parallel result: " + parString);
```

```
Parallel combiner was called
Parallel combiner was called
Parallel combiner was called
Parallel combiner was called
Sequential result: HI Rice Owls are the best
Parallel result: HI Rice Owls are the best
```

Collecting (*Stream<T>*)

Sometimes, you don't want to produce a single value, but a new Collection instead

Technically, this can be done using reduce, but Java Streams give you a more convenient way:

```
<R> R collect(Supplier<R> supplier,
              BiConsumer<R, ? super T> accumulator,
              BiConsumer<R, R> combiner);
```

- *supplier* creates brand new empty objects of type R (usually a Collection)
- *accumulator* takes an element, and adds it to the object of type R (usually a Collection). *Mutable reduction*
- *combiner* combines two objects of type R (usually Collections) into one. *Mutable reduction*

```
List<String> asList = stringStream.collect(ArrayList::new, ArrayList::add, ArrayList::addAll);
```

```
<R, A> R collect(Collector<? super T, A, R> collector);
```

- Same as above, but the Collector object encapsulates all 3 (supplier, accumulator and combiner)
- Java *Collector* class has quite a few handy methods for creating Collectors

```
List<String> asList = stringStream.collect(Collectors.toList());
```

General guidelines

Try to put the operations that reduce the size of the stream early

- *skip(), filter(), distinct(), limit()*
- May reduce the amount of work for later operations

Lambdas passed to both the intermediate and terminal operations should be *pure*

- No side-effects, no IO
- No modifying of the underlying source

Construct your Stream pipelines so that the partitioning and ordering of the reductions and collections doesn't matter

- Always use the real identity in reductions and collections
- Simple *parallel()* mode switch will trigger parallel execution, with the exact same answer

Summary

Java Streams are a mechanism to create lazy sequences of operations on collections of objects

Typically used by constructing a Stream pipeline:

- Create a stream from a ***source*** (such as a Collection)
- Perform a bunch of ***intermediate*** operations (all lazy!)
- Perform a ***terminal*** operation to drive the computation of the result

Streams are easily parallelized!

- Just be careful with lambdas and reductions