CENG 443
Introduction to Object-Oriented Programming Languages and Systems

Streams – 2

Operations that Limit the Stream Size

- limit(n) returns a Stream of the first n elements.
- skip(n) returns a Stream starting with element n (i.e., it throws away the first n elements)
- limit is a short-circuit operation. E.g.,
    - strm.map(funct1).filter(pred).map(funct2).limit(10)
    - if you have a 1000-element stream, it applies funct1 exactly 10 times, evaluates pred at least 10 times (until 10 elements pass), and applies funct2 at most 10 times

Example

```
List<Employee> googlers = EmployeeSamples.getGooglers();
List<String> emps = googlers.stream()
                             .map(Person::getFirstName)
                             .limit(8)
                             .skip(2)
                             .collect(Collectors.toList());
System.out.printf("Names of 6 Googlers: %s.%n", emps);
```

- getFirstName called 6 times, even if Stream is very large
- **Output**
  Names of 6 Googlers: [Eric, Nikesh, David, Patrick, Susan, Peter]

Comparisons

- **sorted**
  - sorted with a Comparator works just like Arrays.sort, discussed earlier
  - sorted with no arguments works only if the Stream elements implement Comparable
  - Sorting Streams is more flexible than sorting arrays because you can do filter and mapping operations before and/or after
    - Note the inconsistency that method is called sorted, not sort
- **min and max**
  - It is faster to use min and max than to sort forward or backward, then take first element
  - min and max take a Comparator as an argument
- **distinct**
  - distinct uses equals as its comparison

Examples

Sorting by salary

```
empStream.sorted((e1, e2) -> e1.getSalary() - e2.getSalary())
```

Richest Employee

```
empStream.max((e1, e2) -> e1.getSalary() – e2.getSalary()).get()
```

Words with duplicates removed

```
stringStream.distinct()
```

Sorting

- The advantage of someStream.sorted(…) over Arrays.sort(…) is that with Streams you can first do operations like map, filter, limit, skip, and distinct
- Doing limit or skip after sorting does *not* short-circuit in the same manner as in the previous section
  - Because the system does not know which are the first or last elements until after sorting
- If the Stream elements implement Comparable, you may omit the lambda and just use someStream.sorted(). Rare.

```
List<Integer> ids = Arrays.asList(9, 11, 10, 8);
List<Employee> emps1 =
  ids.stream().map(EmployeeSamples::findGoogler)
              .sorted((e1, e2) -> e1.getLastName().compareTo(e2.getLastName()))
              .collect(Collectors.toList());
System.out.printf("Googlers with ids %s sorted by last name: %s.%n", ids, emps1);
```

Example

```
public int firstNameComparer(Person other) {
   System.out.println("Comparing first names");
   return(firstName.compareTo(other.getFirstName()));
}

List<Employee> emps3 =
  sampleEmployees().sorted(Person::firstNameComparer)
                   .limit(2)
                   .collect(Collectors.toList());
System.out.printf("Employees sorted by first name: %s.%n", emps3);
```

- The use of limit(2) does *not* reduce the number of times firstNameComparer is called (vs. no limit at all)

min and max

- min and max use the same type of lambdas as sorted, letting you flexibly find the first or last elements based on various different criteria
  - min and max could be easily reproduced by using reduce, but this is such a common case that the short-hand reduction methods (min and max) are built in
- min and max both return an Optional
- Unlike with sorted, you must provide a lambda, regardless of whether or not the Stream elements implement Comparable
- **Performance implications**
- Using min and max is faster than sorting in forward or reverse order, then using findFirst
  - min and max are O(n), sorted is O(n log n)

Examples

```
Employee alphabeticallyFirst =
  ids.stream().map(EmployeeSamples::findGoogler)
              .min((e1, e2) -> e1.getLastName().compareTo(e2.getLastName()))
              .get();
System.out.printf("Googler from %s with earliest last name: %s.%n",
                                                  ids, alphabeticallyFirst);


Employee richest =  ids.stream().map(EmployeeSamples::findGoogler)
              .max((e1, e2) -> e1.getSalary() - e2.getSalary())
              .get();
System.out.printf("Richest Googler from %s: %s.%n",  ids, richest);
```

distinct: Example

- distinct uses equals as itscomparison

```
List<Integer> ids2 = Arrays.asList(9, 10, 9, 10, 9, 10);
List<Employee> emps4 =
  ids2.stream().map(EmployeeSamples::findGoogler)
                .distinct()
                .collect(Collectors.toList());
System.out.printf("Unique Googlers from %s: %s.%n", ids2, emps4);
```

Output

```
Unique Googlers from [9, 10, 9, 10, 9, 10]:
  [Jeffrey Dean [Employee#9 $800,000],
   Sanjay Ghemawat [Employee#10 $700,000]].
```

Operations that Check Matches

- allMatch, anyMatch, and noneMatch take a Predicate and return a boolean
- They stop processing once an answer can be determined
  - E.g., if the first element fails the Predicate, allMatch would immediately return false and skip checking other elements
- count simply returns the number of elements
  - count is a terminal operation, so you cannot first count the elements, then do a further operation on the same Stream
- Example:
  - Is there at least one rich dude?
    - employeeStream.anyMatch(e -> e.getSalary() > 500_000)
  - How many employees match the criteria?
    - employeeStream.filter(somePredicate).count()

Number-Specialized Streams

- IntStream, DoubleStream, LongStream
- A specialization of Stream that makes it easier to deal with ints. Does not extend Stream, but instead extends BaseStream, on which Stream is also built.
- Can make IntStream from int[], whereas Integer[] needed to make Stream<Integer>

Total population in region

```
int population = countryList.stream()
                            .filter(Utils::inRegion)
                            .mapToInt(Country::getPopulation)
                            .sum();
```

Average salary

```
double averageSalary =
  employeeList.stream()
              .mapToDouble(Employee::salary)
              .average()   // average returns OptionalDouble,
              .orElse(-1); // not double
```

How to make IntStream

Stream.of(int1, int2, int3)

```
Stream.of(1, 2, 3, 4)
```

Stream.of(integerArray)

```
Integer[] nums = { 1, 2, 3, 4 };
Stream.of(nums)
```

Stream.of(intArray)

```
int[] nums = { 1, 2, 3, 4 };
Stream.of(nums)
```

Making an IntStream

- regularStream.mapToInt
  - Assume that getAge returns an int. Then, the following produces an IntStream
    - personList.stream().mapToInt(Person::getAge)
- IntStream.of
  - IntStream.of(int1, int2, int2)
  - IntStream.of(intArray)
- IntStream.range, IntStream.rangeClosed
  - IntStream.range(5, 10)
- Random.ints
  - new Random().ints(), anyInstanceOfRandom.ints()
    - An “infinite” IntStream of random numbers. But you can apply limit to make a finite stream, or use findFirst
    - There are also versions where you give range of ints or size of stream

The reduce method

- **Repeated combining**
  - You start with a seed (identity) value, combine this value with the first entry of the Stream, combine the result with the second entry of the Stream, and so forth
    - reduce is particularly useful when combined with map or filter
    - Works properly with parallel streams if operator is associative and has no side effects
- **reduce(starter, binaryOperator)**
  - Takes starter value and BinaryOperator. Returns result directly.
- **reduce(binaryOperator)**
  - Takes BinaryOperator, with no starter. It starts by combining first 2 values with each other. Returns an Optional.

Quick Examples

Maximum of numbers

```
nums.stream().reduce(Double.MIN_VALUE, Double::max)
```

Product of numbers

```
nums.stream().reduce(1, (n1, n2) -> n1 * n2)
```

String Concatenation

```
List<String> letters = Arrays.asList("a", "b", "c", "d");  String
concat = letters.stream().reduce(                         );
```

- “” → starter (identity) value. Combined with the first entry in the Stream
- String::concat → This is the BinaryOperator.
  - It is the same as (s1, s2) -> s1 + s2.
  - It concatenates the seed value with the first Stream entry, concatenates that resultant String with the second Stream entry, and so forth.

```
letters.stream().reduce("", String::concat);
→  "abcd"
```

- String::concat here is the same as if you had written the lambda (s1,s2) -> s1+s2

```
letters.stream().reduce("", (s1,s2) -> s2+s1);
→  "dcba"
```

- This just reverses the order of the s1 and s2 in the concatenation

```
letters.stream().reduce("", (s1,s2) -> s2.toUpperCase() + s1);
→  "DCBA"
```

- Turns into uppercase as you go along

```
letters.stream(). reduce("", (s1,s2) -> s2+s1).toUpperCase();
→  "DCBA"
```

- Alternative to the above that turns into uppercase at the end after reduce is finished

Parallel Streams

- By designating that a Stream be parallel, the operations are automatically done in parallel, without any need for explicit fork/join or threading code
- Designating streams as parallel
  - anyStream.parallel()
  - anyList.parallelStream()
    - Shortcut for anyList.stream().parallel()
- Quick examples
  - Do ops serially
    - someStream.forEach(someThreadSafeOp);
  - Do ops in parallel
    - someStream.parallel().forEach(someThreadSafeOp);

Parallel Streams vs. Concurrent Programming with Threads

- **Parallel streams**
  - Use fork/join framework internally
  - Have one thread per core
  - Are beneficial even when you never wait for I/O
  - Have no benefit on single-core computers
  - Can be used with minimal changes to serial code
- **Concurrent programming with explicit threads**
  - Are usually beneficial only when you wait for I/O (disk, screen, network, user, etc.)
  - Are often beneficial even on single-core computers
  - May have a lot more threads than there are cores
  - Require very large changes to serial code
- **Bottomline**
  - Parallel streams are often *far* easier to use than explicit threads
  - Explicit threads apply in more situations

Advantage of Using Streams

Computing sum with normal loop

```
int[] nums = …;
int sum = 0;
for(int num: nums) {
   sum += num;
}
```

– *Cannot* be easily parallelized

Computing sum with reduction operation

```
int[] nums = …;
int sum = IntStream.of(nums).sum();
```

– *Can* be easily parallelized

```
int sum2 = IntStream.of(nums).parallel().sum();
```

Best Practices: Both Fork/Join and Parallel Streams

- **Check that you get the same answer**
  - Verify that sequential and parallel versions yield the same (or close enough to the same) results. This can be harder than you think when dealing with doubles.
- **Check that the parallel version is faster**
  - Or, at the least, no slower. Test in real-life environment.
  - If running on app server, this analysis might be harder than it looks. Servers automatically handle requests concurrently, and if you have heavy server load and many of those requests use parallel streams, all the cores are likely to *already* be in use, and parallel execution might have little or no speedup.
    - Another way of saying this is that if the CPU of your server is already consistently maxed out, then parallel streams will not benefit you (and could even slow you down).

Will You Always Get Same Answer in Parallel?

- **sorted, min, max**
  - No. (Why not? And do you care?)
- **findFirst**
  - Yes. Use findAny if you do not care.
- **map, filter**
  - No, but you do not care about what the stream looks like in intermediate stages. You only care about the terminal operation
- **allMatch, anyMatch, noneMatch, count**
  - Yes
- **reduce (and sum and average)**
  - It depends!

Equivalence of Parallel Reduction Operations

- **sum, average are the same if**
  - You use IntStream or LongStream
- **sum, average could be different if**
  - You use DoubleStream. Reording the additions could yield slightly different answers due to roundoff error.
    - "Almost the same" may or may not be acceptable
    - If the answers differ, it is not clear which one is "right"
- **reduce is the same if**
  - No side effects on global data are performed
  - The combining operation is associative (i.e., where reordering the operations does not matter).
    - Reordering addition or multiplication of doubles does not necessarily yield exactly the same answer. You may or may not care

Parallel reduce: No Global Data

- **Binary operator itself should be stateless**
    - Guaranteed if an explicit lambda is used, but not guaranteed if you directly build an instance of a class that implements BinaryOperator, or if you use a method reference that refers to a statefull class
- **The operator does not modify global data**
    - The body of a lambda is allowed to mutate instance variables of the surrounding class or call setter methods that modify instance variables. If you do so, there is no guarantee that parallel reduce will be safe.

Parallel reduce: Associative Operation

- **Reordering ops should have no effect**
  - I.e., (a op b) op c == a op (b op c)
- **Examples: are these associative?**
  - Division? No.
  - Subtraction? No.
  - Addition or multiplication of ints? Yes.
  - Addition or multiplication of doubles? No.
    - Not guaranteed to get *exactly* the same result. This may or may not be tolerable in your application, so you should define an acceptable delta and test. If your answers differ by too much, you also have to decide which is the “right” answer.

Ex: Parallel Sum of Square Roots of Doubles

```
public class MathUtils {
  public static double fancySum1(double[] nums) {
    return DoubleStream.of(nums)
                       .map(d -> Math.sqrt(2*d))
                       .sum();
  }

  public static double fancySum2(double[] nums) {
    return DoubleStream.of(nums)
                       .parallel()
                       .map(d -> Math.sqrt(2*d))
                       .sum();
  }

 public static void compareOutput() {
   double[] nums = MathUtils.randomNums(10_000_000);
   double result1 = MathUtils.fancySum1(nums);
   System.out.printf("Serial result   = %,.12f%n", result1);
   double result2 = MathUtils.fancySum2(nums);
   System.out.printf("Parallel result = %,.12f%n", result2);
 }
```

Representative output

```
Serial result   = 16,328,996.081106223000
Parallel result = 16,328,996.081106225000
```

Comparing Performance

```
public static void compareTiming() {
  for(int i=5; i<9; i++) {
    int size = (int)Math.pow(10, i);
    double[] nums = MathUtils.randomNums(size);
    Op serialSum = () -> MathUtils.fancySum1(nums);
    Op parallelSum = () -> MathUtils.fancySum2(nums);
    System.out.printf("Serial sum for length   %,d.%n", size);
    Op.timeOp(serialSum);
    System.out.printf("Parallel sum for length %,d.%n", size);
    Op.timeOp(parallelSum);
  }
}
```

```
Serial sum for length   100,000.
  Elapsed time: 0.012 seconds.
Parallel sum for length 100,000.
  Elapsed time: 0.008 seconds.
Serial sum for length   1,000,000.
  Elapsed time: 0.005 seconds.
Parallel sum for length 1,000,000.
  Elapsed time: 0.003 seconds.
Serial sum for length   10,000,000.
  Elapsed time: 0.047 seconds.
Parallel sum for length 10,000,000.
  Elapsed time: 0.024 seconds.
Serial sum for length   100,000,000.
  Elapsed time: 0.461 seconds.
Parallel sum for length 100,000,000.
  Elapsed time: 0.176 seconds.
```