Streams in Java 8: Part 2

Originals of slides and source code for examples: http://courses.coreservlets.com/Course-Materials/java.html
Also see Java 8 tutorial: http://www.coreservlets.com/java-8-tutorial/ and many other Java EE tutorials: http://www.coreservlets.com/
Customized Java training courses (onsite or at public venues): http://courses.coreservlets.com/java-training.html

For additional materials, please see http://www.coreservlets.com/. The Java tutorial section contains complete source code for all examples in this tutorial series, plus exercises and exercise solutions for each topic.

coreservlets.com – custom onsite training

For customized training related to Java or JavaScript, please email hall@coreservlets.com

Marty is also available for consulting and development support

The instructor is author of several popular Java EE books, two of the most popular Safari videos on Java and JavaScript, and this tutorial.

Courses available at public venues, or custom versions can be held on-site at your organization.

- **Courses developed and taught by Marty Hall**
  - JSF 2.3, PrimeFaces, Java programming (using Java 8, for those new to Java), Java 8 (for Java 7 programmers), JavaScript, jQuery, Angular 2, Ext JS, Spring Framework, Spring MVC, Android, GWT, custom mix of topics.
  - Java 9 training coming soon.
  - Courses available in any state or country.
  - Maryland/DC companies can also choose afternoon/evening courses.
- **Courses developed and taught by coreservlets.com experts (edited by Marty)**
  - Hadoop, Spark, Hibernate/JPA, HTML5, RESTful Web Services

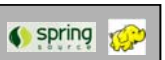
Contact hall@coreservlets.com for details

Topics in This Section

- **More stream methods**
  - limit, skip
  - sorted, min, max , distinct
  - noneMatch, allMatch, anyMatch, count
- **Number-specialized streams**
  - IntStream, DoubleStream, LongStream
- **Reduction operations**
  - reduce(starterValue, binaryOperator)
  - reduce(binaryOperator).orElse(...)
  - min, max, sum, average





Operations that Limit the Stream Size: limit, skip

Limiting Stream Size

- **Big ideas**
  - 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., if you have a 1000-element stream and then do the following, it applies funct1 exactly 10 times, evaluates pred at least 10 times (until 10 elements pass), and applies funct2 at most 10 times

```
strm.map(funct1).filter(pred).map(funct2).limit(10)
```

- **Quick examples**
  - First 10 elements
    - `someLongStream.limit(10)`
  - Last 15 elements
    - `twentyElementStream.skip(5)`

limit and skip: Example

- **Code**

```
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);
```

- **Point**
  - getFirstName called 6 times, even if Stream is very large
- **Results**

```
Names of 6 Googlers: [Eric, Nikesh, David, Patrick, Susan, Peter].
```





Operations that use Comparisons: sorted, min, max, distinct

Comparisons: Big Ideas

- **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

Comparisons: Quick 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

- **Big ideas**
  - 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.
- **Supporting code from Person class**

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

Sorting by Last Name: Example

- **Code**

```
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);
```

- **Results**

```
Googlers with ids [9, 11, 10, 8] sorted by last name:
  [Gilad Bracha [Employee#11 $600,000],
   Jeffrey Dean [Employee#9 $800,000],
   Sanjay Ghemawat [Employee#10 $700,000],
   Peter Norvig [Employee#8 $900,000]].
```


Sorting by First Name then Limiting: Example

- **Code**

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

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

```
Employees sorted by first name:
  [Amy Accountant [Employee#25 $85,000],
   Archie Architect [Employee#16 $144,444]].
```

min and max

- **Big ideas**
  - 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)

min: Example

- **Code**

```
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);
```

- **Results**

```
Googler from [9, 11, 10, 8] with earliest last name:
  Gilad Bracha [Employee#11 $600,000].
```

max: Example

- **Code**

```
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);
```

- **Results**

```
Richest Googler from [9, 11, 10, 8]:
  Peter Norvig [Employee#8 $900,000].
```


distinct: Example

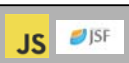
- **Code**

```
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);
```

- **Results**

```
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, noneMatch, count

Checking Matches

- **Big ideas**
  - 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
- **Quick examples**
  - Is there at least one rich dude?
    - employeeStream.anyMatch(e -> e.getSalary() > 500_000)
  - How many employees match the criteria?
    - employeeStream.filter(somePredicate).count()

Matches: Examples

- **Code**

```
List<Employee> googlers = EmployeeSamples.getGooglers();
boolean isNobodyPoor = googlers.stream().noneMatch(e -> e.getSalary() < 200_000);
Predicate<Employee> megaRich = e -> e.getSalary() > 7_000_000;
boolean isSomeoneMegaRich = googlers.stream().anyMatch(megaRich);
boolean isEveryoneMegaRich = googlers.stream().allMatch(megaRich);
long numberMegaRich = googlers.stream().filter(megaRich).count();
System.out.printf("Nobody poor? %s.%n", isNobodyPoor);
System.out.printf("Someone mega rich? %s.%n", isSomeoneMegaRich);
System.out.printf("Everyone mega rich? %s.%n", isEveryoneMegaRich);
System.out.printf("Number mega rich: %s.%n", numberMegaRich);
```

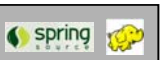
- **Results**

```
Nobody poor? true.
Someone mega rich? true.
Everyone mega rich? false.
Number mega rich: 3.
```









Number-Specialized Streams

IntStream

- **Big idea**
  - 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.
- **Motivation**
  - Simpler methods
    - min(), max(), sum(), average()
      - min and max take no arguments, unlike the Stream versions that need a Comparator
  - Output as int[]
    - toArray()
  - Can make IntStream from int[], whereas Integer[] needed to make Stream<Integer>
- **Similar interfaces**
  - DoubleStream
  - LongStream

Quick Examples

- **Cost of fleet of cars**

```
double totalCost =
  carList.stream().mapToDouble(Car::getPrice).sum();
```

- **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
```

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)
    - Can also use Arrays.stream for this
- **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

IntStream Methods

- **Specific to number streams**
  - min(), max(): No arguments, output is OptionalInt
  - sum(): No arguments, output is int. Returns 0 for an empty IntStream.
  - average: No arguments, output is OptionalDouble
  - toArray(): No arguments, output is int[]
    - Although building an int[] from an IntStream is more convenient than building an Integer[] from a Stream<Integer>, turning an IntStream into a List<Integer> is hard, and you cannot simply do yourIntStream.collect(Collectors.toList()).
- **Similar to regular streams**
  - map, mapToDouble, mapToObject
    - Function for map must produce int
  - filter, reduce, forEach, limit, skip, parallel, anyMatch, etc.
    - Most methods from Stream, but IntStream does *not* extend Stream (only BaseStream)

Similar Stream Specializations

- **DoubleStream**
  - Creating
    - `regularStream.mapToDouble`
    - `DoubleStream.of`
    - `someRandom.doubles`
  - Methods
    - min, max, sum, average (no args, output is double)
    - toArray (no args, output is double[])
- **LongStream**
  - Creating
    - regularStream.mapToLong, LongStream.of, someRandom.longs
  - Methods
    - min, max, sum, average (no args, output is long)
    - toArray (no args, output is long[])

Common Incorrect Attempts at Making IntStream

- **Stream.of(int1, int2, int3)**

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

  - Builds Stream<Integer>, not IntStream
- **Stream.of(integerArray)**

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

  - Builds Stream<Integer>, not IntStream
- **Stream.of(intArray)**

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

  - Builds Stream containing one element, where that one element is an int[]
  - See analogous code on next slide

Building Stream Containing Array: Analogous Example with Varargs for Object

```
public class UseArgs {
  public static int firstNumber(int... nums) {
    return(nums[0]);
  }

  public static Object firstObject(Object... objects) {
    return(objects[0]);
  }
}
```


Analogous Example Continued

```
public class SupplyArgs {
  public static void main(String[] args) {
    int[] nums = { 1, 2, 3 };
    int result1 = UseArgs.firstNumber(1, 2, 3);
    System.out.printf("result1: %s%n", result1);
    int result2 = UseArgs.firstNumber(nums);
    System.out.printf("result2: %s%n", result2);
    Object result3 = UseArgs.firstObject(1, 2, 3);
    System.out.printf("result3: %s%n", result3);
    Object result4 = UseArgs.firstObject(nums);
    System.out.printf("result4: %s%n", result4);
  }
}
```

```
result1: 1
result2: 1
result3: 1
result4: [I@659e0bfd
```






The reduce method and Related Reduction Operations

Reduction Operations

- **Big idea**
  - Reduction operations take a Stream<T>, and combine or compare the entries to produce a single value of type T
- **Trivial examples**
  - findFirst().orElse(…)
  - findAny().orElse(…)
- **Examples in Stream**
  - min(comparator), max(comparator)
  - reduce(starterValue, binaryOperator)
  - reduce(binaryOperator).orElse(…)
- **Examples in IntStream**
  - min(), max(), sum(), average()

reduce: Big Idea

- **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.

reduce: Quick Examples

- **Maximum of numbers**

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

- **Product of numbers**

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

- **Concatenation of strings**

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

Concatenating Strings: More Details

- **Code**

```
List<String> letters = Arrays.asList("a", "b", "c", "d");
String concat = letters.stream().reduce("", String::concat);
System.out.printf("Concatenation of %s is %s.%n", letters, concat);
```

This is the starter (identity) value. It is combined with the first entry in the Stream.

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.

- **Results**

```
Concatenation of [a, b, c, d] is abcd.
```


Concatenating Strings: Variations

- **Data**
  - `List<String> letters = Arrays.asList("a", "b", "c", "d");`
- **Various reductions**
  - `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

Finding "Biggest" Employee

- **Code**

```
Employee poorest = new Employee("None", "None", -1, -1);
BinaryOperator<Employee> richer = (e1, e2) -> {
  return(e1.getSalary() >= e2.getSalary() ? e1 : e2);
};
Employee richestGoogler = googlers.stream().reduce(poorest, richer);
System.out.printf("Richest Googler is %s.%n", richestGoogler);
```

- **Results**

```
Richest Googler is Larry Page [Employee#1 $9,999,999].
```

reduce uses the BinaryOperator to combine the starter value with the first Stream entry, then combines that result with the second Stream entry, and so forth.

Finding Sum of Salaries: Two Alternatives

- **Alternative 1**
  - Use mapToInt, then use sum()
- **Alternative 2**
  - Use map, then use reduce

Finding Sum of Salaries

```
public class SalarySum {
  private static List<Employee> googlers = EmployeeSamples.getGooglers();

  public static int sum1() {
    return googlers.stream()
                   .mapToInt(Employee::getSalary)
                   .sum();
  }

  public static int sum2() {
    return googlers.stream()
                   .map(Employee::getSalary)
                   .reduce(0, Integer::sum);
  }
}
```


Finding Smallest Salary: Three Alternatives

- **Alternative 1**
  - Use mapToInt, then use min()
- **Alternative 2**
  - Use map, then use min(comparator)
- **Alternative 3**
  - Use map, then use reduce

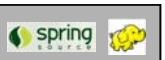
Finding Smallest Salary

```
public static int min1() {
  return googlers.stream().mapToInt(Employee::getSalary)
                          .min()
                          .orElse(Integer.MAX_VALUE);
}

public static int min2() {
  return googlers.stream().map(Employee::getSalary)
                          .min((n1, n2) -> n1 - n2)
                          .orElse(Integer.MAX_VALUE);
}

public static int min3() {
  return googlers.stream().map(Employee::getSalary)
                          .reduce(Integer.MAX_VALUE, Integer::min);
}
```









Wrap-Up

Summary: More Stream Methods

- **Limiting Stream size**
  - limit, skip
    - Can trigger short-circuiting
- **Using comparisons**
  - sorted, min, max, distinct
    - Must traverse entire stream
- **Finding matches**
  - allMatch, anyMatch, noneMatch
    - Can be short-circuited
  - count
    - Must traverse entire stream

Summary: Specializations and Reductions

- **Reduction operations on Stream<T>**
  - min(comparator)
  - max(comparator)
  - reduce(starterValue, binaryOperator)
  - reduce(binaryOperator).orElse(…)
- **IntStream and DoubleStream**
  - regularStream.mapToInt, regularStream.mapToDouble
  - IntStream.of, DoubleStream.of
- **Reduction operations on IntStream and DoubleStream**
  - min(), max(), sum(), average()

coreservlets.com – custom onsite training







Questions?

More info:
http://courses.coreservlets.com/Course-Materials/java.html – General Java programming tutorial
http://www.coreservlets.com/java-8-tutorial/ – Java 8 tutorial
http://courses.coreservlets.com/java-training.html – Customized Java training courses, at public venues or onsite at *your* organization
http://coreservlets.com/ – JSF 2, PrimeFaces, Java 7 or 8, Ajax, jQuery, Hadoop, RESTful Web Services, Android, HTML5, Spring, Hibernate, Servlets, JSP, GWT, and other Java EE training
Many additional free tutorials at coreservlets.com (JSF, Android, Ajax, Hadoop, and lots more)

For additional materials, please see http://www.coreservlets.com/. The Java tutorial section contains complete source code for all examples in this tutorial series, plus exercises and exercise solutions for each topic.