



# Streams in Java 8: Part 2

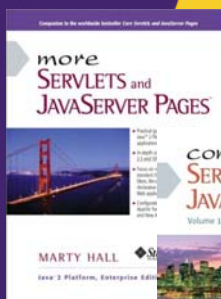
Originals of slides and source code for examples: <http://www.coreservlets.com/java-8-tutorial/>

Also see the general Java programming tutorial – <http://courses.coreservlets.com/Course-Materials/java.html>  
and customized Java training courses (onsite or at public venues) – <http://courses.coreservlets.com/java-training.html>



**Customized Java EE Training:** <http://courses.coreservlets.com/>

Java 7, Java 8, JSF 2, PrimeFaces, Android, JSP, Ajax, jQuery, Spring MVC, RESTful Web Services, GWT, Hadoop  
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.



**For live Java-related training,  
email [hall@coreservlets.com](mailto:hall@coreservlets.com)**

**Marty is also available for consulting and development support**



**Taught by lead author of *Core Servlets & JSP*, co-author of *Core JSF* (4<sup>th</sup> Ed), & this tutorial. Available at public venues, or customized versions can be held on-site at your organization.**

- Courses developed and taught by Marty Hall
  - JSF 2.2, PrimeFaces, servlets/JSP, Ajax, jQuery, Android, Java 7 or 8 programming, GWT, custom mix of topics
  - Courses available in any state or country. Maryland/DC area companies can also choose afternoon/evening courses.
- Courses developed and taught by coreservlets.com experts (edited by Marty)
  - Spring MVC, Core Spring, Hibernate/JPA, Hadoop, HTML5, RESTful Web Services

Contact [hall@coreservlets.com](mailto:hall@coreservlets.com) for details



# Topics in This Section

- **More stream methods**
  - reduce, sum
  - limit, skip
  - sorted, min, max , distinct
  - noneMatch, allMatch, anyMatch, count
- **Parallel streams**
  - Why stream approach is often better than traditional loops
  - Knowing when parallel reduce is safe
- **Infinite streams**
  - Really unbounded streams with values that are calculated on the fly
- **Grouping stream elements**
  - Fancy uses of collect

6

© 2015 Marty Hall



## reduce: Combining Stream Elements



Customized Java EE Training: <http://courses.coreservlets.com/>

Java 7, Java 8, JSF 2, PrimeFaces, Android, JSP, Ajax, jQuery, Spring MVC, RESTful Web Services, GWT, Hadoop  
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.

# reduce

- **Big idea**

- 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
  - Version 1 takes starter value and BinaryOperator. Returns result directly. Version 2 takes BinaryOperator, with no starter. It starts by combining first 2 values with each other. Returns an Optional.
    - reduce is particularly useful when combined with map or filter
    - Works in parallel if operator is associative and has no side effects

- **Quick examples**

- Maximum of numbers
  - `nums.stream().reduce(Double.MIN_VALUE, Double::max)`
    - There is also builtin “max” method of DoubleStream:  
`DoubleStream.of(...).max().orElse(Double.MIN_VALUE);`
- Product of numbers
  - `nums.stream().reduce(1, (n1, n2) -> n1 * n2)`

8

# Concatenating Strings

- **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.

- **Results**

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

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.

9

# Concatenating Strings: Variations

- **Data**

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

- **Various reductions**

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

- "abcd"

- Remember that `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.toUpperCase());`

- "DCBA"

10

# 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.

11

# Finding Biggest Number

- **Using reduce directly**

```
List<Double> nums1a = Arrays.asList(1.2, -2.3, 4.5, -5.6);
double maxNum1 =
    nums1a.stream().reduce(Double.MIN_VALUE, Double::max);
System.out.printf("Max of %s is %s.%n", nums1a, maxNum1);
```

- **Using DoubleStream and max**

```
double[] nums1b = {1.2, -2.3, 4.5, -5.6};
double maxNum2 =
    DoubleStream.of(nums1b).max()
                  .orElse(Double.MIN_VALUE);
System.out.printf("Max of [same numbers] is %s.%n",
    maxNum2);
```

- **Results**

```
Max of [1.2, -2.3, 4.5, -5.6] is 4.5.
Max of [same numbers] is 4.5.
```

12

# Summing Numbers: Three Alternatives

- **Version 1: same reduce approach as before**

```
List<Integer> nums2 = Arrays.asList(1, 2, 3, 4);
int sum1 = nums2.stream().reduce(0, Integer::sum);
```

- **Version 2: using reduce with no starter**

- Starts by combining first two Stream entries. Since there might not be 1 or more entries, returns an Optional.

```
int sum2 =
    nums2.stream().reduce(Integer::sum).orElse(0);
```

- **Version 3: using sum method of IntStream**

- Supplying int[] instead of Integer[] to Arrays.stream results in IntStream. IntStream has builtin sum method.

```
int[] nums3 = { 1, 2, 3, 4 };
int sum3 = Arrays.stream(nums3).sum();
// Or IntStream.of(nums3).sum()
```

13



# Combining map and reduce: Version 1 (direct reduce)

- **Code**

```
int sum4 = nums2.stream().map(EmployeeSamples::findGoogler)
                      .map(Employee::getSalary)
                      .reduce(0, Integer::sum);

System.out.printf
    ("Combined salaries of Googlers with IDs %s is $%,d.%n",
     nums2, sum4);
```

- **Results**

Combined salaries of Googlers with IDs [1, 2, 3, 4]  
is \$33,333,330.

14

# Combining map and reduce: Version 1 (direct reduce)

- **Code**

```
int sum4 = nums2.stream().map(EmployeeSamples::findGoogler)
                      .map(Employee::getSalary)
                      .reduce(0, Integer::sum);

System.out.printf
    ("Combined salaries of Googlers with IDs %s is $%,d.%n",
     nums2, sum4);
```

- **Results**

Combined salaries of Googlers with IDs [1, 2, 3, 4]  
is \$33,333,330.

15

# Combining map and reduce: Version 2 (mapToInt and sum)

- **Code**

```
int sum5 = nums2.stream().map(EmployeeSamples::findGoogler)
                    .mapToInt(Employee::getSalary)
                    .sum();

System.out.printf
    ("Combined salaries of Googlers with IDs %s is $%,d.%n",
     nums2, sum5);
```

- **Results**

```
Combined salaries of Googlers with IDs [1, 2, 3, 4]
is $33,333,330.
```

16

© 2015 Marty Hall



## Operations that Limit the Stream Size: limit, skip



Customized Java EE Training: <http://courses.coreservlets.com/>

Java 7, Java 8, JSF 2, PrimeFaces, Android, JSP, Ajax, jQuery, Spring MVC, RESTful Web Services, GWT, Hadoop  
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.

# 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)
  - Note: this was called `substream` in most Java-8 beta releases, and was renamed to `skip` quite late. But, `skip` is better name.
- Both are short-circuit operations. E.g., if you have a 1000-element stream and then do the following, it applies `fn1` exactly 10 times, evaluates `pred` exactly 10 times, and applies `fn2` at most 10 times  
`strm.map(fn1).filter(pred).map(fn2).limit(10)`

- **Quick examples**

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

18

## 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` is called only 6 times, even if Stream has 10,000 elements

- **Results**

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

19





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



Customized Java EE Training: <http://courses.coreservlets.com/>

Java 7, Java 8, JSF 2, PrimeFaces, Android, JSP, Ajax, jQuery, Spring MVC, RESTful Web Services, GWT, Hadoop  
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.

## Comparisons

- **Big ideas**

- Sorting Streams is more flexible than sorting arrays because you can do filter and mapping operations first
- min and max are more efficient than sorting in forward or reverse order, then taking first
- distinct uses equals as its comparison

- **Quick examples**

- Sorting by salary

```
empStream.map(...).filter(...).limit(...)  
          .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()));  
}
```

22

# 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]].
```

23

# 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]].
```

24

# 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  $O(n \log n)$

25

## 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].

26

## 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].

27

# 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]].
```

28

© 2015 Marty Hall



## Operations that Check Matches: allMatch, anyMatch, noneMatch, count



Customized Java EE Training: <http://courses.coreservlets.com/>

Java 7, Java 8, JSF 2, PrimeFaces, Android, JSP, Ajax, jQuery, Spring MVC, RESTful Web Services, GWT, Hadoop  
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.



# 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 based on the count

- **Quick examples**

- Is there at least one rich dude?
  - employeeStream.anyMatch(e -> e.getSalary() > 500000)
- How many employees match the criteria?
  - employeeStream.filter(somePredicate).count()

30

# Matches: Examples

- **Code**

```
List<Employee> googlers = EmployeeSamples.getGooglers();
boolean isNobodyPoor =
    googlers.stream().noneMatch(e -> e.getSalary() < 200000);
Predicate<Employee> megaRich = e -> e.getSalary() > 7000000;
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.
```

31



# Parallel Streams



Customized Java EE Training: <http://courses.coreservlets.com/>

Java 7, Java 8, JSF 2, PrimeFaces, Android, JSP, Ajax, jQuery, Spring MVC, RESTful Web Services, GWT, Hadoop  
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.

## Parallel Streams

- **Big idea**
  - By designating that a Stream be parallel, the operations are automatically done in parallel.
  - No explicit multi-threading code is required.
    - This is easiest to see with methods like `forEach` (assuming the operation is thread-safe!): the performance should scale linearly with the number of cores. But, even operations like `map`, `filter`, `reduce`, and `findAny` can get big performance gains. Operations in those cases must be stateless and non-interfering (and `reduce` op must be associative).
- **Quick examples**
  - Do side effects serially
    - `someStream.forEach(someThreadSafeOp)`
  - Do side effects concurrently
    - `someStream.parallel().forEach(someThreadSafeOp)`

## Helper Methods for Timing

```
private static void timingTest(Stream<Employee> testStream) {
    long startTime = System.nanoTime();
    testStream.forEach(e -> doSlowOp());
    long endTime = System.nanoTime();
    System.out.printf(" %.3f seconds.%n",
        deltaSeconds(startTime, endTime));
}

private static double deltaSeconds(long startTime,
    long endTime) {
    return((endTime - startTime) / 1000000000);
}
```

34

## Helper Method: Operation that Takes One Second

```
// Simulate a time-consuming operation

private static void doSlowOp() {
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException ie) {
        // Nothing to do here.
    }
}
```

35

## Parallel Example: Code

```
List<Employee> googlers = EmployeeSamples.getGooglers();
System.out.print("Serial version [11 entries]:");
timingTest(googlers.stream());
int numProcessorsOrCores =
    Runtime.getRuntime().availableProcessors();
System.out.printf("Parallel version on %s-core machine:",
    numProcessorsOrCores);
timingTest(googlers.stream().parallel());
System.out.print("Serial version [4 entries]:");
timingTest(googlers.stream().limit(4));
System.out.printf("Parallel version on %s-core machine:",
    numProcessorsOrCores);
timingTest(googlers.stream().parallel().limit(4));
```

In these examples, you could replace `googlers.stream().parallel()` with `googlers.parallelStream()`. But, it is important to know about `.parallel()`, since `parallel()` can be done in the middle of a series of operations, e.g.: `someList.stream().filter(...).map(...).parallel().moreOps(...)`, and `parallel()` can be applied to streams created other ways, e.g.: `Stream.of(someArray).parallel().ops()`.

36

## Parallel Example: Results

```
Serial version [11 entries]: 11.000 seconds.
Parallel version on 4-core machine: 3.000 seconds.
Serial version [4 entries]: 4.000 seconds.
Parallel version on 4-core machine: 1.000 seconds.
```

37

# Advantage of Using Streams vs. Traditional Loops: Parallel Code

- **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();
```

38

# Parallel Reduce: When is it Safe?

- **Combining operation is associative**

- I.e.,  $(a \text{ op } b) \text{ op } c == a \text{ op } (b \text{ op } c)$ 
  - Determining this is trickier than it looks;  
see the exercises where we ask if multiplication  
satisfies this requirement

- **No side effects on global data are performed**

- Binary operator itself is 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

39





# Infinite (Unbounded On-the-Fly) Streams



Customized Java EE Training: <http://courses.coreservlets.com/>

Java 7, Java 8, JSF 2, PrimeFaces, Android, JSP, Ajax, jQuery, Spring MVC, RESTful Web Services, GWT, Hadoop  
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.

## Infinite Streams: Big Ideas

- **Stream.generate(valueGenerator)**
  - Stream.generate lets you specify a Supplier. This Supplier is invoked each time the system needs a Stream element.
    - Powerful when Supplier maintains state, but won't work in parallel
- **Stream.iterate(initialValue, valueTransformer)**
  - Stream.iterate lets you specify a seed and a UnaryOperator f. The seed becomes the first element of the Stream, f(seed) becomes the second element, f(second) becomes third element, etc.
- **Usage**
  - The values are not calculated until they are needed
  - To avoid unterminated processing, you must eventually use a size-limiting operation like limit or findFirst (but not skip alone)
    - The point is not really that this is an “infinite” Stream, but that it is an unbounded “on the fly” Stream – one with no fixed size, where the values are calculated as you need them.

# generate

- **Big ideas**

- You supply a function (Supplier) to Stream.generate. Whenever the system needs stream elements, it invokes the function to get them.
- You must limit the Stream size.
  - Usually with limit. skip alone is not enough, since the size is still unbounded
- The function can maintain state so that new values are based on any or all of the previous values

- **Quick example**

```
List<Employee> emps =  
    Stream.generate(() -> randomEmployee())  
        .limit(...)  
        .collect(Collectors.toList());
```

42

## Stateless generate Example: Random Numbers

- **Code**

```
Supplier<Double> random = Math::random;  
System.out.println("2 Random numbers:");  
Stream.generate(random).limit(2).forEach(System.out::println);  
System.out.println("4 Random numbers:");  
Stream.generate(random).limit(4).forEach(System.out::println);
```

- **Results**

```
2 Random numbers:  
0.00608980775038892  
0.2696067924164013  
4 Random numbers:  
0.7761651889987567  
0.818313574113532  
0.07824375091607816  
0.7154788145391667
```

43

## Stateful generate Example: Supplier Code

```
public class FibonacciMaker implements Supplier<Long> {  
    private long previous = 0;  
    private long current = 1;  
  
    @Override  
    public Long get() {  
        long next = current + previous;  
        previous = current;  
        current = next;  
        return(previous);  
    }  
}
```

Lambdas cannot define instance variables,  
so we use a regular class instead of a  
lambda to define the Supplier.

44

## Helper Code: Simple Methods to Get Any Amount of Fibonacci

```
public static Stream<Long> makeFibStream() {  
    return(Stream.generate(new FibonacciMaker()));  
}  
  
public static Stream<Long> makeFibStream(int numFibs) {  
    return(makeFibStream().limit(numFibs));  
}  
  
public static List<Long> makeFibList(int numFibs) {  
    return(makeFibStream(numFibs)  
        .collect(Collectors.toList()));  
}  
  
public static Long[] makeFibArray(int numFibs) {  
    return(makeFibStream(numFibs).toArray(Long[]::new));  
}
```

45

# Stateful generate Example

- **Main code**

```
System.out.println("5 Fibonacci numbers:");
FibStream.makeFibStream(5)
    .forEach(System.out::println);
System.out.println("25 Fibonacci numbers:");
FibStream.makeFibStream(25)
    .forEach(System.out::println);
```

- **Results**

```
5 Fibonacci numbers:
1
1
2
3
5
25 Fibonacci numbers:
1
1
...
75025
```

46

## iterate

- **Big ideas**

- You specify a seed value and a UnaryOperator f. The seed becomes the first element of the Stream, f(seed) becomes the second element, f(second) [i.e., f(f(seed))] becomes third element, etc.
- You must limit the Stream size.
  - Usually with limit. skip alone is not enough, since the size is still unbounded

- **Quick example**

```
List<Integer> powersOfTwo =
    Stream.iterate(1, n -> n * 2)
        .limit(...)
        .collect(Collectors.toList());
```

47

# Simple Example: Twitter Messages

- **Idea**

- Generate a series of Twitter messages

- **Approach**

- Start with a very short String as the first message
- Append exclamation points on the end
- Continue to 140-character limit

- **Core code**

```
Stream.iterate("Base Msg",  
              msg -> msg + "Suffix")  
      .limit(someCutoff)
```

48

# Twitter Messages

- **Code**

```
System.out.println("14 Twitter messages:");  
Stream.iterate("Big News!!",  
              msg -> msg + "!!!!!!!!!!!!")  
      .limit(14)  
      .forEach(System.out::println);
```

- **Results**

```
Big News!!  
Big News!!!!!!!!!!!!!!  
Big News!!!!!!!!!!!!!!!!!!!!!!!!!!!!  
Big News!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  
[Etc]
```

49



## More Complex Example: Large Prime Numbers

- **Idea**
  - Generate a series of very large consecutive prime numbers (e.g., 100 or 150 digits or more)
  - Large primes are used extensively in cryptography
- **Approach**
  - Start with a prime BigInteger as the seed
  - Supply a UnaryOperator that finds the first prime number higher than the given one
- **Core code**

```
Stream.iterate(Primes.findPrime(numDigits),  
              Primes::nextPrime)  
      .limit(someCutoff)
```

50

## Helper Methods: Idea

- **Idea**
  - Generate a random odd BigInteger of the requested size, check if prime, keep adding 2 until you find a match.
- **Why this is feasible**
  - The BigInteger class has a builtin probabilistic algorithm (Miller-Rabin test) for determining if a number is prime without attempting to factor it. It is ultra-fast even for 100-digit or 200-digit numbers.
  - Technically, there is a  $2^{100}$  chance that this falsely identifies a prime, but since  $2^{100}$  is about the number of particles in the universe, that's not a very big risk
    - Algorithm is not fooled by Carmichael numbers

51

## Helper Methods: Code

```
public static BigInteger nextPrime(BigInteger start) {
    if (isEven(start)) {
        start = start.add(ONE);
    } else {
        start = start.add(TWO);
    }
    if (start.isProbablePrime(ERR_VAL)) {
        return(start);
    } else {
        return(nextPrime(start));
    }
}

public static BigInteger findPrime(int numDigits) {
    if (numDigits < 1) {
        numDigits = 1;
    }
    return(nextPrime(randomNum(numDigits)));
}
```

52

Complete source code can be downloaded from <http://www.coreservlets.com/java-8-tutorial/>

## Making Stream of Primes

```
public static Stream<BigInteger> makePrimeStream(int numDigits) {
    return(Stream.iterate(Primes.findPrime(numDigits),
                          Primes::nextPrime));
}

public static Stream<BigInteger> makePrimeStream(int numDigits,
                                                  int numPrimes) {
    return(makePrimeStream(numDigits).limit(numPrimes));
}

public static List<BigInteger> makePrimeList(int numDigits,
                                              int numPrimes) {
    return(makePrimeStream(numDigits, numPrimes).
           collect(Collectors.toList()));
}

public static BigInteger[] makePrimeArray(int numDigits, int numPrimes) {
    return(makePrimeStream(numDigits, numPrimes).toArray(BigInteger[]::new));
}
```

53

# Primes

- **Code**

```
System.out.println("10 100-digit primes:");  
PrimeStream.makePrimeStream(100, 10)  
    .forEach(System.out::println);
```

- **Results**

```
10 100-digit primes:  
3484894489805924599033259501599559961057903572743870105972345438458556253531271262848463552867976353  
3484894489805924599033259501599559961057903572743870105972345438458556253531271262848463552867976647  
3484894489805924599033259501599559961057903572743870105972345438458556253531271262848463552867976663  
3484894489805924599033259501599559961057903572743870105972345438458556253531271262848463552867976689  
3484894489805924599033259501599559961057903572743870105972345438458556253531271262848463552867977233  
3484894489805924599033259501599559961057903572743870105972345438458556253531271262848463552867977859  
3484894489805924599033259501599559961057903572743870105972345438458556253531271262848463552867977889  
3484894489805924599033259501599559961057903572743870105972345438458556253531271262848463552867977989  
3484894489805924599033259501599559961057903572743870105972345438458556253531271262848463552867978031  
3484894489805924599033259501599559961057903572743870105972345438458556253531271262848463552867978103
```

54

© 2015 Marty Hall



**collect:**  
**Fancier Stream Output**



Customized Java EE Training: <http://courses.coreservlets.com/>

Java 7, Java 8, JSF 2, PrimeFaces, Android, JSP, Ajax, jQuery, Spring MVC, RESTful Web Services, GWT, Hadoop

Developed and taught by well-known author and developer. At public venues or onsite at *your* location.

# collect

- **Big idea**

- Combined with methods in the Collectors class, you can build many data types out of Streams

- **Quick examples**

- List (shown in previous section)

- anyStream.collect(toList())

- String

- stringStream.collect(joining(*delimiter*)).toString()

- Set

- anyStream.collect(toSet())

- Other collection

- anyStream.collect(toCollection(*CollectionType::new*))

- Map

- strm.collect(partioningBy(...)), strm.collect(groupingBy(...))

For brevity, the examples here assume you have done  
"import static java.util.stream.Collectors.\*",  
so that toList() really means Collectors.toList()

56

# Building Lists

- **Code**

```
List<Integer> ids = Arrays.asList(2, 4, 6, 8);
List<Employee> emps =
    ids.stream().map(EmployeeSamples::findGoogler)
        .collect(Collectors.toList());
System.out.printf("Googlers with ids %s: %s.%n",
    ids, emps);
```

- **Results**

```
Googlers with ids [2, 4, 6, 8]:
[Sergey Brin [Employee#2 $8,888,888],
Nikesh Arora [Employee#4 $6,666,666],
Patrick Pichette [Employee#6 $4,444,444],
Peter Norvig [Employee#8 $900,000]].
```

Remember that you can do a static import on  
java.util.stream.Collectors.\* so that you can use toList()  
instead of Collectors.toList().

Also recall that List has a builtin toString method that  
prints the entries comma-separated inside square brackets.  
Here and elsewhere, line breaks and whitespace added to  
printouts for readability.

57

## Aside: The StringJoiner Class

- **Big idea**

- Java 8 added new StringJoiner class that builds delimiter-separated Strings, with optional prefix and suffix
- Java 8 also added static “join” method to the String class; it uses StringJoiner

- **Quick examples (result: "Java, Lisp, Ruby")**

- Explicit StringJoiner with no prefix or suffix

```
StringJoiner joiner1 = new StringJoiner(", ");
String result1 =
    joiner1.add("Java").add("Lisp").add("Ruby").toString();
```
- Usually easier: String.join convenience method

```
String result2 = String.join(", ", "Java", "Lisp", "Ruby");
```

58

## Building Strings

- **Code**

```
List<Integer> ids = Arrays.asList(2, 4, 6, 8);
String lastNames =
    ids.stream().map(EmployeeSamples::findGoogler)
          .map(Employee::getLastName)
          .collect(Collectors.joining(", "))
          .toString();
System.out.printf("Last names of Googlers with ids %s: %s.%n",
    ids, lastNames);
```

- **Results**

```
Last names of Googlers with ids [2, 4, 6, 8]:
Brin, Arora, Pichette, Norvig.
```

59



# Building Sets

- **Code**

```
List<Employee> googlers = EmployeeSamples.getGooglers();
Set<String> firstNames =
    googlers.stream().map(Employee::getFirstName)
                .collect(Collectors.toSet());
Stream.of("Larry", "Harry", "Peter", "Deiter", "Eric", "Barack")
    .forEach(s -> System.out.printf
                ("%s is a Googler? %s.%n",
                 s,
                 firstNames.contains(s) ? "Yes" : "No"));
```

- **Results**

```
Larry is a Googler? Yes.
Harry is a Googler? No.
Peter is a Googler? Yes.
Deiter is a Googler? No.
Eric is a Googler? Yes.
Barack is a Googler? No.
```

60

# Building Other Collections

- **Big idea**

- You provide a `Supplier<Collection>` to collect. Java takes the resultant `Collection` and then calls “add” on each element of the `Stream`.

- **Quick examples**

- `ArrayList`
  - `someStream.collect(toCollection(ArrayList::new))`
- `TreeSet`
  - `someStream.collect(toCollection(TreeSet::new))`
- `Stack`
  - `someStream.collect(toCollection(Stack::new))`
- `Vector`
  - `someStream.collect(toCollection(Vector::new))`

The examples here assume you have done  
“import static java.util.stream.Collectors.\*”,  
so that `toCollection(...)` really means  
`Collectors.toCollection(...)`

61

# Building Other Collections: TreeSet

- **Code**

```
TreeSet<String> firstNames2 =  
    googlers.stream()  
        .map(Employee::getFirstName)  
        .collect(Collectors.toCollection(TreeSet::new));  
Stream.of("Larry", "Harry", "Peter", "Deiter", "Eric", "Barack")  
    .forEach(s -> System.out.printf  
        ("%s is a Googler? %s.%n",  
            s,  
            firstNames2.contains(s) ? "Yes" : "No"));
```

- **Results**

```
Larry is a Googler? Yes.  
Harry is a Googler? No.  
Peter is a Googler? Yes.  
Deiter is a Googler? No.  
Eric is a Googler? Yes.  
Barack is a Googler? No.
```

62

# partitioningBy: Building Maps

- **Big idea**

- You provide a Predicate. It builds a Map where true maps to a List of entries that passed the Predicate, and false maps to a List that failed the Predicate.

- **Quick example**

```
Map<Boolean, List<Employee>> oldTimersMap =  
    employeeStream().collect  
        (partitioningBy(e -> e.getEmployeeId() < 10));
```

- Now, oldTimersMap.get(true) returns a List<Employee> of employees whose ID's are less than 10, and oldTimersMap.get(false) returns a List<Employee> of everyone else.

63

# partitioningBy: Example

- **Code**

```
Map<Boolean,List<Employee>> richTable =  
    googlers.stream().collect  
        (partitioningBy(e -> e.getSalary() > 1000000));  
System.out.printf("Googlers with salaries over $1M: %s.%n",  
    richTable.get(true));  
System.out.printf("Destitute Googlers: %s.%n",  
    richTable.get(false));
```

- **Results**

Googlers with salaries over \$1M: [Larry Page [Employee#1 \$9,999,999],  
Sergey Brin [Employee#2 \$8,888,888], Eric Schmidt [Employee#3 \$7,777,777],  
Nikesh Arora [Employee#4 \$6,666,666], David Drummond [Employee#5 \$5,555,555],  
Patrick Pichette [Employee#6 \$4,444,444],  
Susan Wojcicki [Employee#7 \$3,333,333]].

Destitute Googlers: [Peter Norvig [Employee#8 \$900,000],  
Jeffrey Dean [Employee#9 \$800,000], Sanjay Ghemawat [Employee#10 \$700,000],  
Gilad Bracha [Employee#11 \$600,000]].

64

# groupingBy: Another Way of Building Maps

- **Big idea**

- You provide a Function. It builds a Map where each output value of the Function maps to a List of entries that gave that value.
  - E.g., if you supply Employee::getFirstName, it builds a Map where supplying a first name yields a List of employees that have that first name.

- **Quick example**

```
Map<Department,List<Employee>> deptTable =  
    employeeStream()  
        .collect(Collectors.groupingBy(Employee::getDepartment));
```

- Now, deptTable.get(someDepartment) returns a List<Employee> of everyone in that department.

65

## groupBy: Supporting Code

- **Idea**

- Make a class called Emp that is a simplified Employee that has a first name, last name, and office/location name, all as Strings.

- **Sample Emps**

```
private static Emp[] sampleEmps = {
    new Emp("Larry", "Page", "Mountain View"),
    new Emp("Sergey", "Brin", "Mountain View"),
    new Emp("Lindsay", "Hall", "New York"),
    new Emp("Hesky", "Fisher", "New York"),
    new Emp("Reto", "Strobl", "Zurich"),
    new Emp("Fork", "Guy", "Zurich"),
};
public static List<Emp> getSampleEmps() {
    return(Arrays.asList(sampleEmps));
}
```

66

## groupBy: Example

- **Code**

```
Map<String,List<Emp>> officeTable =
    EmpSamples.getSampleEmps().stream()
        .collect(Collectors.groupingBy(Emp::getOffice));
System.out.printf("Emps in Mountain View: %s.%n",
    officeTable.get("Mountain View"));
System.out.printf("Emps in NY: %s.%n",
    officeTable.get("New York"));
System.out.printf("Emps in Zurich: %s.%n",
    officeTable.get("Zurich"));
```

- **Results**

Emps in Mountain View:

[Larry Page [Mountain View], Sergey Brin [Mountain View]].

Emps in NY: [Lindsay Hall [New York], Hesky Fisher [New York]].

Emps in Zurich: [Reto Strobl [Zurich], Fork Guy [Zurich]].

67



# Wrap-Up



Customized Java EE Training: <http://courses.coreservlets.com/>

Java 7, Java 8, JSF 2, PrimeFaces, Android, JSP, Ajax, jQuery, Spring MVC, RESTful Web Services, GWT, Hadoop  
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.

## Summary: More Stream Methods

- **reduce**
  - `someStream.reduce(seedValue, binaryOperator)`
    - Also `someStream.reduce(binaryOperator)`, returning an Optional
- **Limiting Stream size**
  - `limit`, `skip`
    - Can trigger short-circuiting
- **Using comparisons**
  - `sorted`, `min`, `max`, `distinct`
- **Finding matches**
  - `allMatch`, `anyMatch`, `noneMatch`, `count`
- **Fancy output for “collect”**
  - `toList`, `joining`, `toSet`, `toCollection`, `partitioningBy`, `groupingBy`

# Summary: Fancy Stream Types

- **Parallel Streams**

- `anyStream.parallel().normalStreamOps(...)`
  - Useful also for `findAny`, `map`, `filter`, `reduce`, etc.
  - You are responsible for being sure the operations are threadsafe (especially with `forEach` operations that modify shared data) and that the operation for `reduce` is associative

- **Infinite (unbounded on-the-fly) Streams**

- `Stream.generate(someStatelessSupplier).limit(...)`
- `Stream.generate(someStatefullSupplier).limit(...)`
- `Stream.iterate(seedValue, operatorOnSeed).limit(...)`
  - The benefit is that there is no predetermined number of entries: values are calculated as you need them

70

© 2015 Marty Hall



## 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](http://coreservlets.com) (JSF, Android, Ajax, Hadoop, and lots more)



71

**Customized Java EE Training:** <http://courses.coreservlets.com/>

Java 7, Java 8, JSF 2, PrimeFaces, Android, JSP, Ajax, jQuery, Spring MVC, RESTful Web Services, GWT, Hadoop

Developed and taught by well-known author and developer. At public venues or onsite at *your* location.