

Overview of Programming with Java 8 Lambdas & Streams

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

**Institute for Software
Integrated Systems**

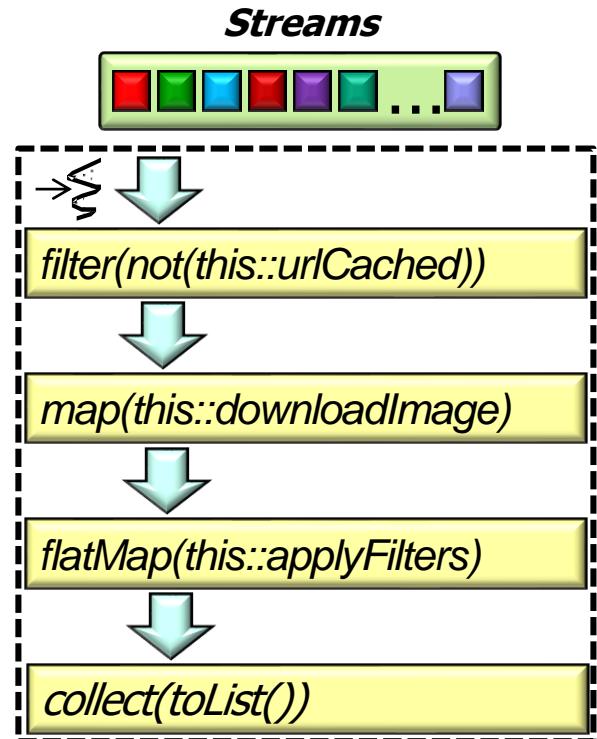
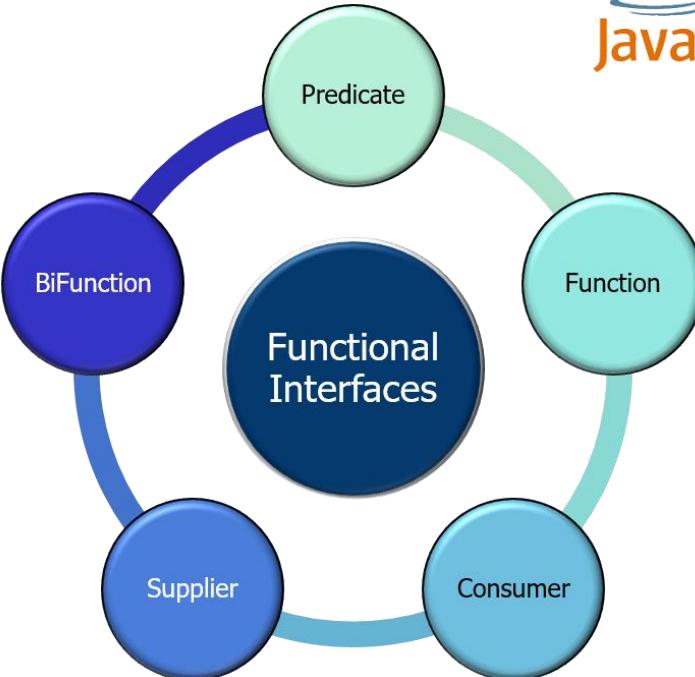
**Vanderbilt University
Nashville, Tennessee, USA**



Learning Objectives in this Lesson

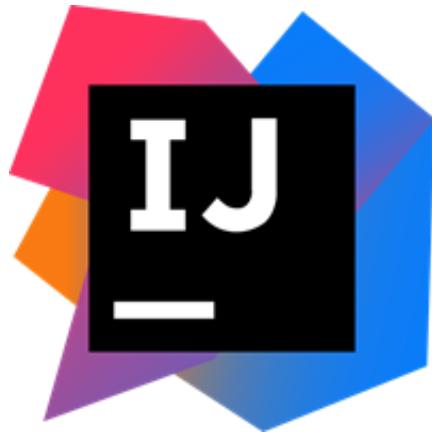
- Know what topics we'll cover

λ



Learning Objectives in this Lesson

- Know what topics we'll cover
- Learn where to find Java 8 & relevant IDEs



Learning Objectives in this Lesson

- Know what topics we'll cover
- Learn where to find Java 8 & relevant IDEs
- Be aware of other digital learning resources



Learning Objectives in this Lesson

- Know what topics we'll cover
- Learn where to find Java 8 & relevant IDEs
- Be aware of other digital learning resources
- Be able to locate examples of Java 8 programs

USE THE
SOURCE LIKE
YODA



douglasraigschmidt updates		
		Latest commit a67fd89 35 minutes ago
BarrierTaskGang	Updates	10 months ago
BuggyQueue	updates	a day ago
BusySynchronizedQueue	updates	4 months ago
DeadlockQueue	Refactored	3 years ago
ExpressionTree	Updates	10 months ago
Factorials	update	5 days ago
ImageStreamGang	updates	22 days ago
ImageTaskGangApplication	Updates	10 months ago
Java8	update	3 days ago
PalantirManagerApplication		7 months ago
PingPongApplication		10 months ago
PingPongWrong		3 years ago
SearchStreamForkJoin		35 minutes ago
SearchStreamGang		3 hours ago
SearchStreamSpliterator		37 minutes ago
SearchTaskGang		4 months ago
SimpleAtomicLong		2 years ago
SimpleBlockingQueue	Updates	4 months ago
SimpleSearchStream	update	6 days ago
ThreadJoinTest	updates	22 days ago
ThreadedDownloads	Updates	10 months ago
UserOrDaemonExecutor	Refactored	3 years ago
UserOrDaemonRunnable	Updates.	2 years ago
UserOrDaemonThread	Updates	10 months ago
UserThreadInterrupted	Update	2 years ago
.gitattributes	Committed.	3 years ago
.gitignore	Updates	10 months ago
README.md	updates	21 days ago

*We'll review many of
these examples, so
feel free to clone or
download this repo!*

See www.github.com/douglasraigschmidt/LiveLessons

Overview of this Course

Overview of this Course

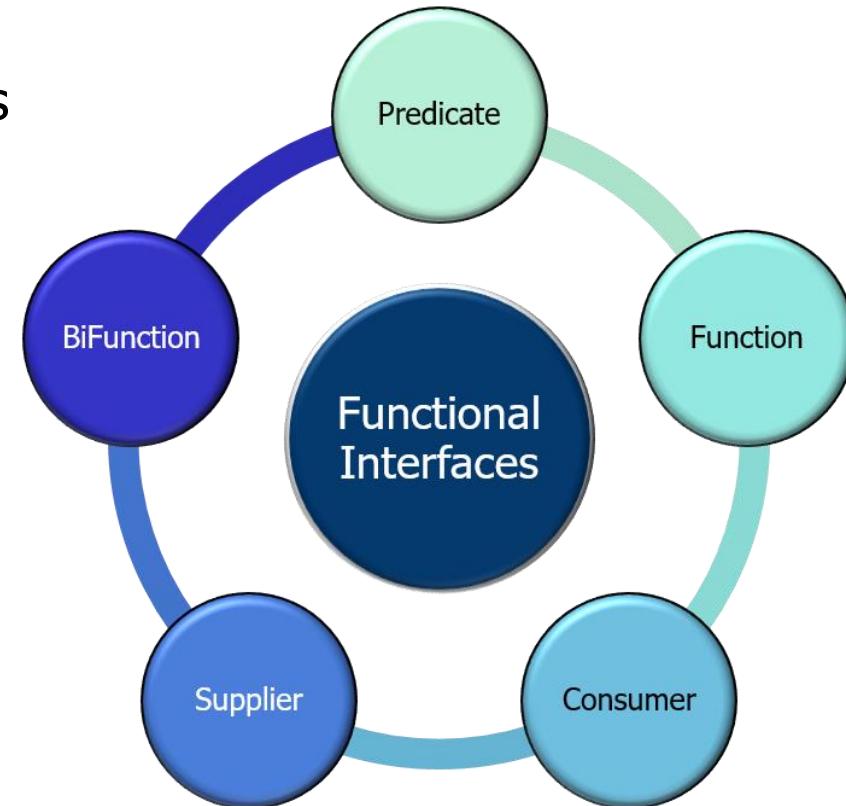
- We focus on using core Java 8 functional programming features, e.g.



Overview of this Course

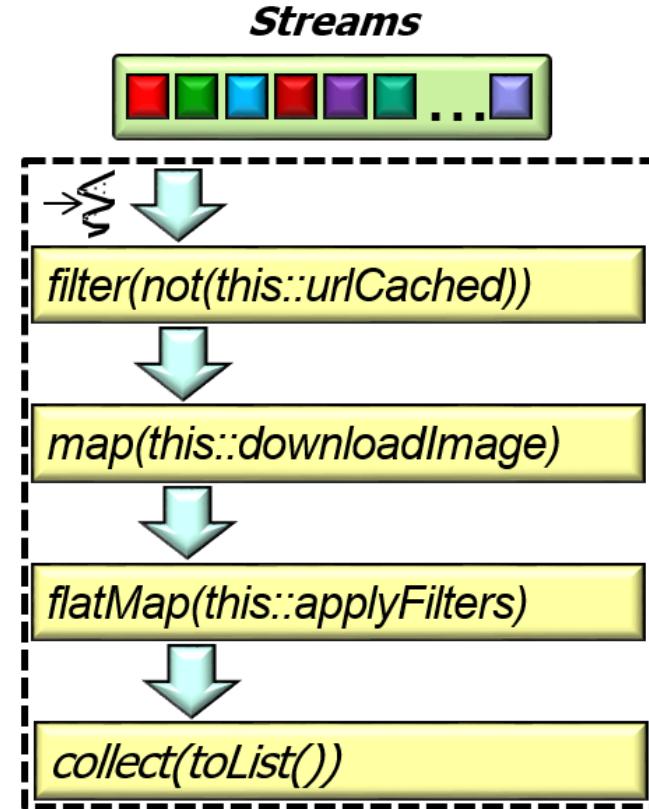
- We focus on using core Java 8 functional programming features, e.g.
 - Lambda expressions, method references, & functional interfaces

λ



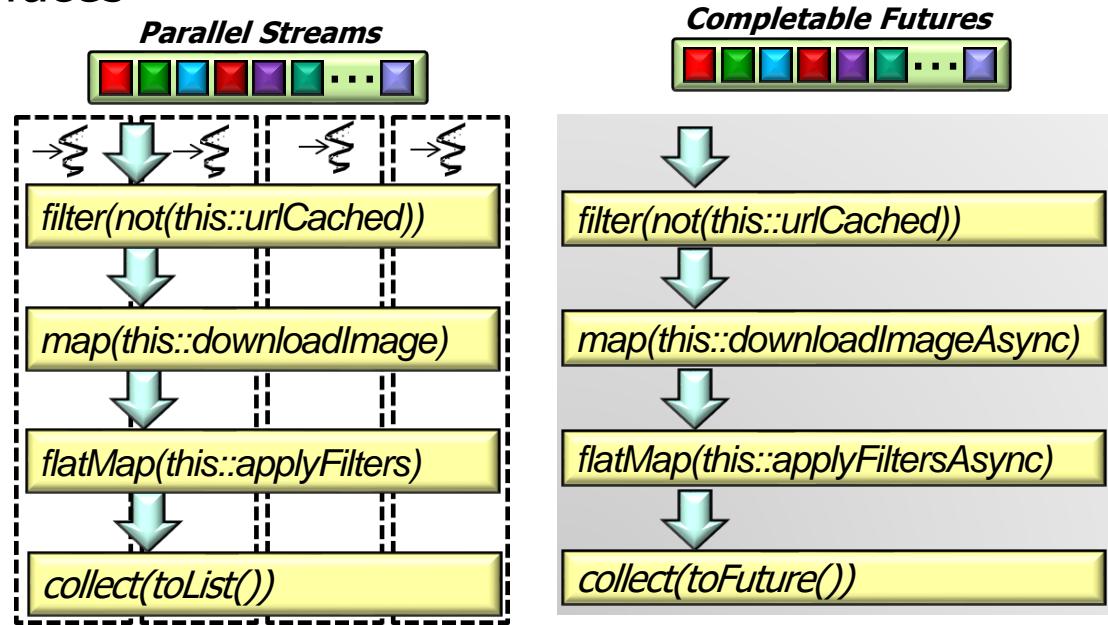
Overview of this Course

- We focus on using core Java 8 functional programming features, e.g.
 - Lambda expressions, method references, & functional interfaces
 - The streams framework
 - Manipulate flows of elements declaratively via function composition



Overview of this Course

- We focus on using core Java 8 functional programming features, e.g.
 - Lambda expressions, method references, & functional interfaces
 - The streams framework



These features are the foundation for Java 8's concurrency/parallelism frameworks

Overview of this Course

- All Java 8 programming examples we cover are available on github

douglascraigschmidt	updates	Latest commit a67fd89 38 minutes ago
BarrierTaskGang	Updates	10 months ago
BuggyQueue	updates	a day ago
BusySynchronizedQueue	updates	4 months ago
DeadlockQueue	Refactored	3 years ago
ExpressionTree	Updates	10 months ago
Factorials	update	5 days ago
ImageStreamGang	updates	22 days ago
ImageTaskGangApplication	Updates	10 months ago
Java8	update	3 days ago
PalantiriManagerApplication	Updates	7 months ago
PingPongApplication	Updates	10 months ago
PingPongWrong	Refactored	3 years ago
SearchStreamForkJoin	updates	38 minutes ago

See www.github.com/douglascraigschmidt/LiveLessons

Accessing Java 8 Features & Functionality

Accessing Java 8 Features & Functionality

- The Java 8 (& beyond) runtime environment (JRE) supports Java 8 features

Overview Downloads Documentation Community Technologies Training

Java SE Downloads

 DOWNLOAD  DOWNLOAD

Java Platform (JDK) 8u101 / 8u102 NetBeans with JDK 8

Java Platform, Standard Edition

Java SE 8u101 / 8u102
Java SE 8u101 includes important security fixes. Oracle strongly recommends that all Java SE 8 users upgrade to this release. Java SE 8u102 is a patch-set update, including all of 8u101 plus additional features (described in the release notes).
[Learn more](#)

- Installation Instructions
- Release Notes
- Oracle License
- Java SE Products
- Third Party Licenses
- Certified System Configurations
- Readme Files
 - JDK ReadMe
 - JRE ReadMe

JDK DOWNLOAD
Server JRE DOWNLOAD
JRE DOWNLOAD



See docs.oracle.com/javase/8/docs/technotes/guides/install/install_overview.html

Accessing Java 8 Features & Functionality

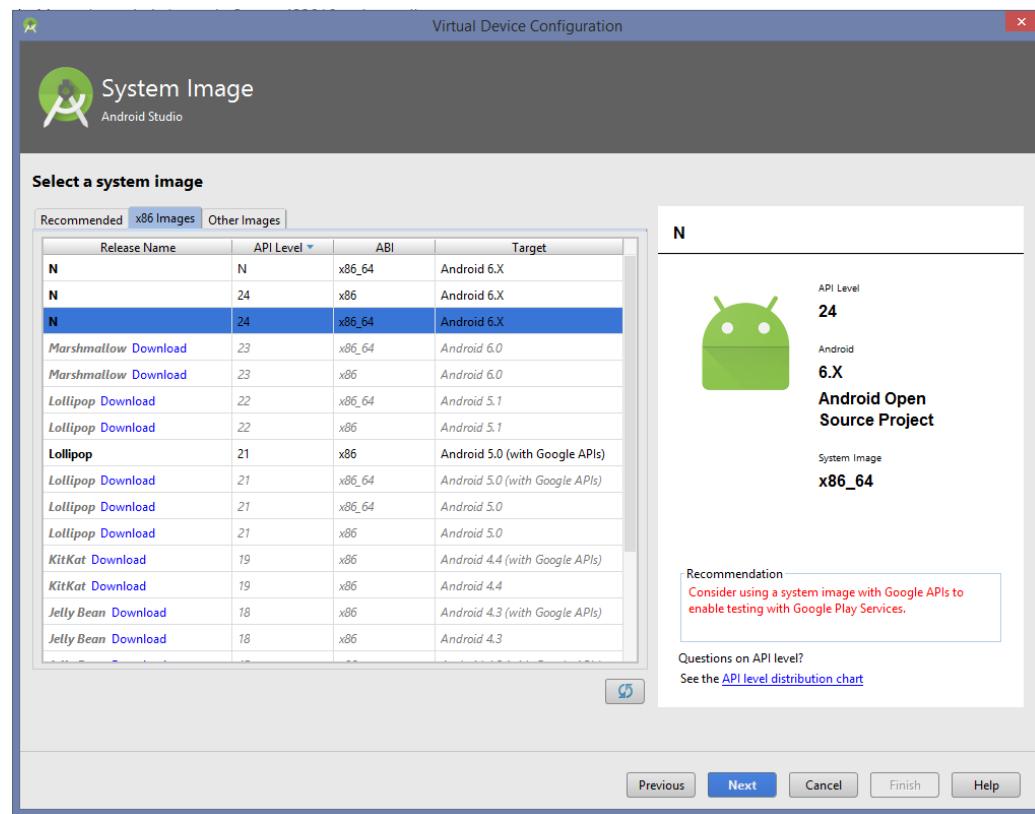
- The Java 8 (& beyond) runtime environment (JRE) supports Java 8 features
 - IntelliJ & Eclipse are popular Java 8 IDEs



See www.eclipse.org/downloads & www.jetbrains.com/idea/download

Accessing Java 8 Features & Functionality

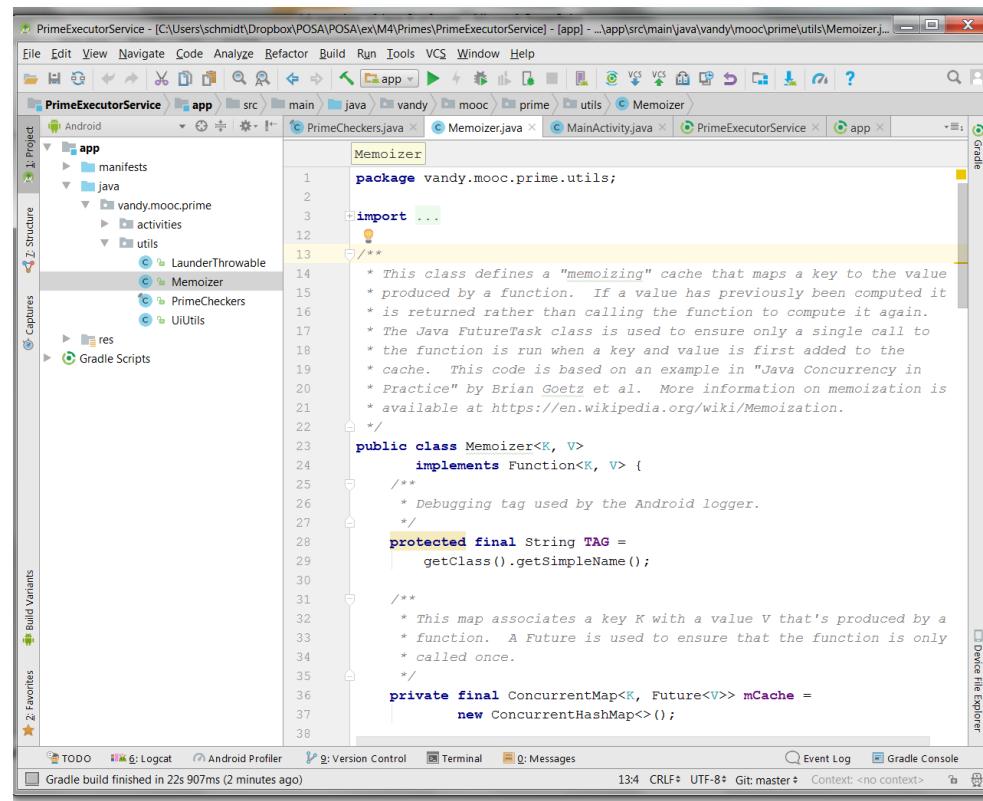
- The Java 8 (& beyond) runtime environment (JRE) supports Java 8 features
 - IntelliJ & Eclipse are popular Java 8 IDEs
 - Most Java 8 features are supported by Android API level 24 (& beyond)



See android-developers.googleblog.com/2017/03/future-of-java-8-language-feature.html

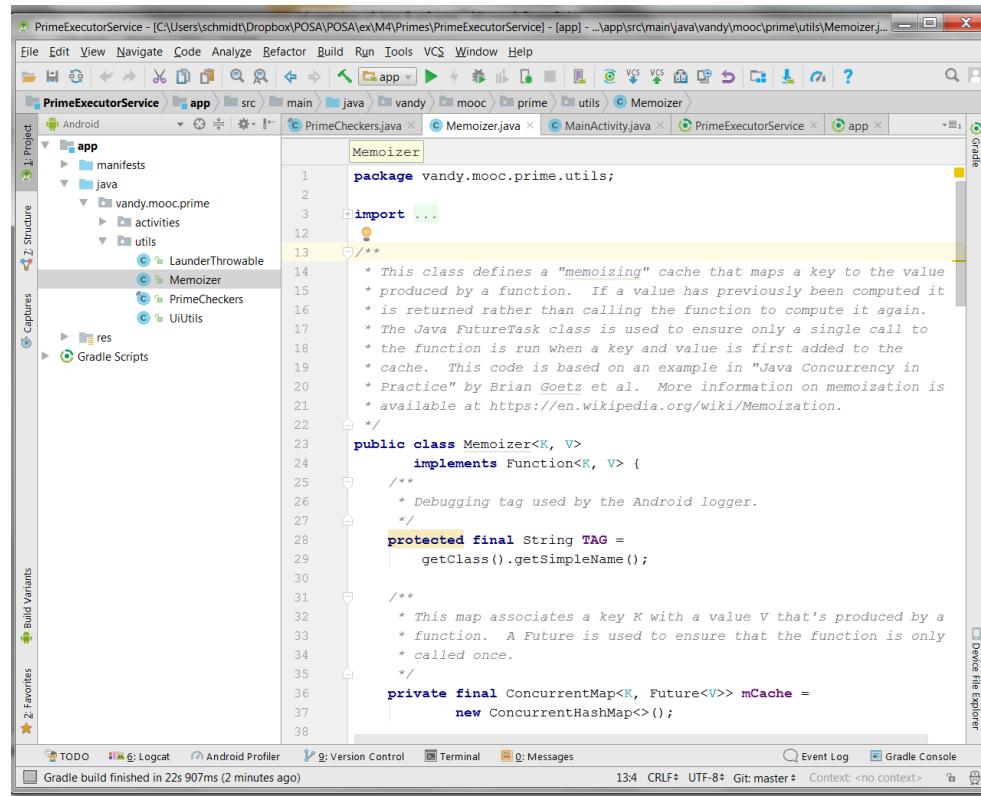
Accessing Java 8 Features & Functionality

- The Java 8 (& beyond) runtime environment (JRE) supports Java 8 features
 - IntelliJ & Eclipse are popular Java 8 IDEs
 - Most Java 8 features are supported by Android API level 24 (& beyond)
 - A subset of Java 8 features are available in earlier Android releases, as well



Accessing Java 8 Features & Functionality

- The Java 8 (& beyond) runtime environment (JRE) supports Java 8 features
 - IntelliJ & Eclipse are popular Java 8 IDEs
 - Most Java 8 features are supported by Android API level 24 (& beyond)
 - A subset of Java 8 features are available in earlier Android releases, as well
 - Make sure to get Android Studio 3.x or later!



See developer.android.com/studio/preview/features/java8-support.html

Accessing Java 8 Features & Functionality

- Java 8 source code is available online
 - For browsing
grepcode.com/file/repository.
 - grepcode.com/java/root/jdk/openjdk/8-b132/java
- For downloading
jdk8.java.net/download.html



The screenshot shows the Java.net website with the URL java.net/projects/jdk8. The page features a header with the Java.net logo and the tagline "The Source for Java Technology Collaboration". On the right, there are links for "Login | Register | Help". A sidebar on the left has a "JDK 8" section with links to "Downloads", "Feedback Forum", "OpenJDK", and "Planet JDK". The main content area is titled "JDK 8 Project" with the subtitle "Building the next generation of the JDK platform". It includes a "JDK 8 snapshot builds" section with a list of items like "Download 8u40 early access snapshot builds", "Source code (instructions)", "Official Java SE 8 Reference Implementations", and "Early Access Build Test Results (instructions)". There's also a "We Want Contributions!" section and a "Feedback" section with instructions for reporting bugs.

JDK 8 Project
Building the next generation of the JDK platform

JDK 8 snapshot builds

- Download 8u40 early access snapshot builds
- Source code (instructions)
- Official Java SE 8 Reference Implementations
- Early Access Build Test Results (instructions)

Feedback

Please use the [Project Feedback](#) forum if you have suggestions for or encounter issues using JDK 8.

If you find bugs in a release, please submit them using the usual [Java SE bug reporting channels](#), not with the Issue tracker accompanying this project. Be sure to include complete version information from the output of the `java -version` command.

Other Digital Learning Resources

Other Digital Learning Resources

- There are several related Live Training courses coming soon

Scalable Programming with Java 8 Parallel Streams



DOUGLAS SCHMIDT



March 29th, 2019
10:00am – 2:00pm CST

[SEE PRICING OPTIONS](#)

Reactive Programming with Java 8 CompletableFuture



DOUGLAS SCHMIDT



February 19th, 2019
9:00am – 1:00pm CST

[SEE PRICING OPTIONS](#)

Design Patterns in Java



DOUGLAS SCHMIDT



January 7th & 8th 2019
11:00am – 3:00pm CST

[SEE PRICING OPTIONS](#)

Scalable Concurrency with the Java Executor Framework



DOUGLAS SCHMIDT



December 11th, 2018
9:00am – 12:00pm CST

[SEE PRICING OPTIONS](#)

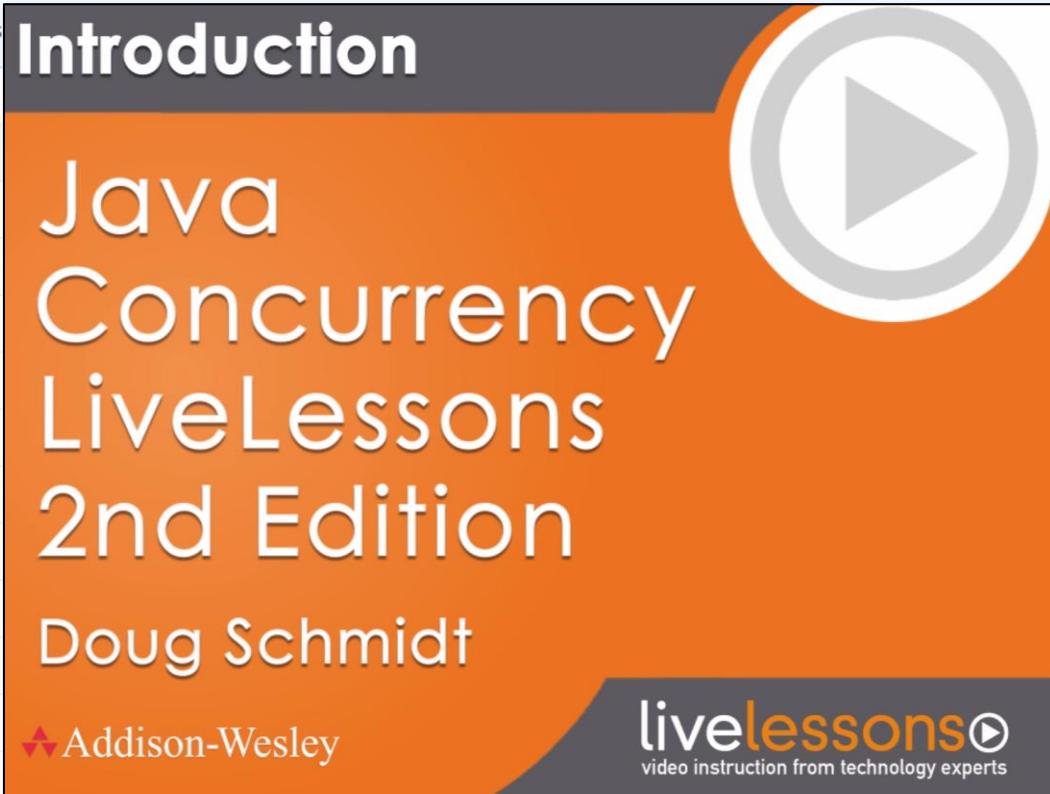
194 spots available

Registration closes October 28, 2018 5:00 PM

See www.dre.vanderbilt.edu/~schmidt/DigitalLearning

Other Digital Learning Resources

- Examples not covered in these courses are covered in my LiveLessons course



The image shows the cover of the book "Java Concurrency LiveLessons 2nd Edition" by Doug Schmidt. The cover is orange with white text. At the top, it says "Introduction". In the center, there is a large play button icon. Below the play button, the title "Java Concurrency LiveLessons 2nd Edition" is written in large, bold, white font. Below the title, the author's name "Doug Schmidt" is also in white. At the bottom left, there is a Addison-Wesley logo. On the right side of the cover, there are three smaller images: a video player showing a man in a red shirt, a screenshot of a computer screen displaying code, and another screenshot of a computer screen displaying code.

douglascaigrschmidt updates

- BarrierTaskGang
- BuggyQueue
- BusySynchronizedQueue
- DeadlockQueue
- ExpressionTree
- Factorials
- ImageStreamGang
- ImageTaskGangApplication
- Java8
- PalantiriManagerApplication
- PingPongApplication
- PingPongWrong
- SearchStreamForkJoin

Introduction

Java Concurrency LiveLessons 2nd Edition

Doug Schmidt

livelessons

video instruction from technology experts

©2017 Pearson, Inc.

See www.dre.vanderbilt.edu/~schmidt/LiveLessons/CPiJava

Other Digital Learning Resources

- There's also a Facebook group dedicated to discussing Java 8-related topics

The screenshot shows a Facebook group page. At the top, there's a blue header with the Facebook logo, a search bar, and a profile picture for 'Douglas'. Below the header, the group name 'Java Concurrency LiveLessons' is displayed in large white text on an orange background. To the right of the group name, there's a small video thumbnail showing a man with a laptop. On the left side, there's a sidebar with various group management options like 'Concurrent Programming in Java', 'Discussion', 'Members', 'Events', 'Photos', 'Group Insights', and 'Manage Group'. Below the sidebar, there's a search bar and a 'Write Post' button. The main content area has a text input field with placeholder text 'Write something...' and several interaction buttons: 'Photo/Video', 'Poll', 'Feeling/Activ...', and '...'. On the right side, there's a section titled 'ADD MEMBERS' with a search bar and a list of member profiles.

See www.facebook.com/groups/1532487797024074

Other Digital Learning Resources

- See my website for many more videos & screencasts related to programming with Java 8 & associated topics



Digital Learning Offerings

Douglas C. Schmidt (d.schmidt@vanderbilt.edu)
Associate Chair of [Computer Science and Engineering](#),
[Professor](#) of Computer Science, and Senior Researcher
in the [Institute for Software Integrated Systems \(ISIS\)](#)
at [Vanderbilt University](#)



O'Reilly LiveTraining Courses

- Programming with Java 8 Lambdas and Streams
 - [January 9th, 2018, 9:00am-12:00pm central time](#)
 - [February 1st, 2018, 9:00am-12:00pm central time](#)
 - March 1st, 2018, 9:00am-12:00pm central time
- Scalable Programming with Java 8 Parallel Streams
 - [January 10th, 2018, 11:00am-3:00pm central time](#)
 - February 6th, 2018, 11:00am-3:00pm central time
 - March 6th, 2018, 11:00am-3:00pm central time
- Reactive Programming with Java 8 Completable Futures
 - [January 12th, 2018, 10:00am-1:00pm central time](#)
 - [February 13th, 2018, 10:00am-2:00pm central time](#)
 - March 13th, 2018, 10:00am-2:00pm central time

Pearson LiveLessons Courses

- [Java Concurrency](#)
- [Design Patterns in Java](#)

Coursera MOOCs

- [Android App Development](#) Coursera Specialization
- [Pattern-Oriented Software Architecture \(POSA\)](#)

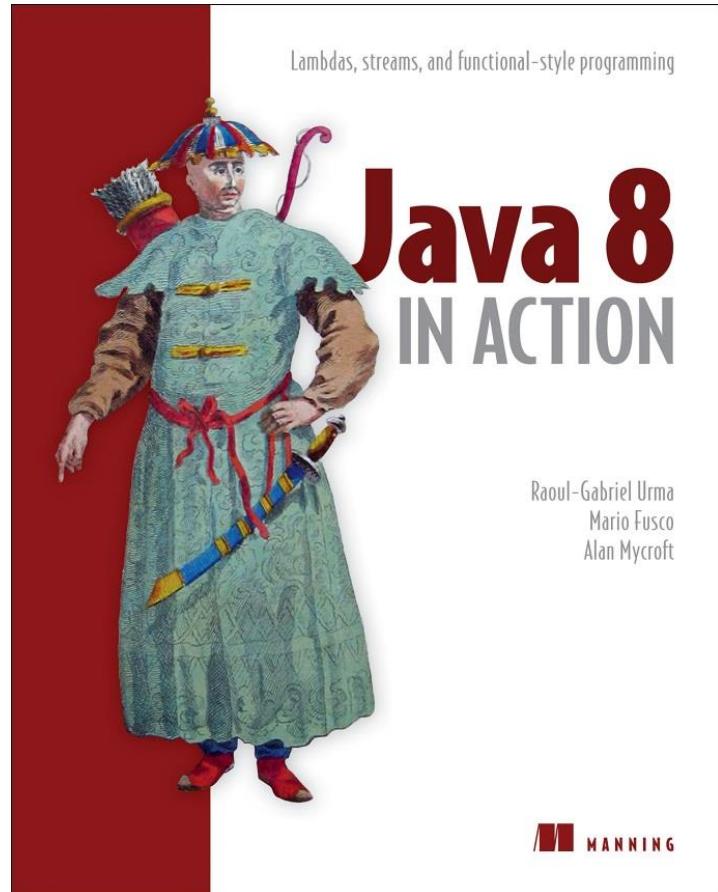
Vanderbilt University Courses

- [Playlist](#) from my [YouTube Channel](#) videos from [CS 891: Introduction to Concurrent and Parallel Java Programming with Android](#)
- [Playlist](#) from my [YouTube Channel](#) videos from [CS 892: Concurrent Java Programming with Android](#)
- [Playlist](#) from my [YouTube Channel](#) videos from [CS 251: Intermediate Software Design with Java](#)
- [Playlist](#) from my [YouTube Channel](#) videos from [CS 282: Concurrent Java Network Programming in Android](#)
- [Playlist](#) from my [YouTube Channel](#) videos from [CS 251: Intermediate Software Design with C++](#)
- [Playlist](#) from my [YouTube Channel](#) videos from [CS 282: Systems Programming for Android](#)

See www.dre.vanderbilt.edu/~schmidt/DigitalLearning

Other Digital Learning Resources

- Another excellent source of material to consult is the book *Java 8 in Action*



See www.manning.com/books/java-8-in-action

Other Digital Learning Resources

- Another excellent source of material to consult is the book *Java 8 in Action*
 - There are also good online articles based on this book

Processing Data with Java SE 8 Streams, Part 1

by Raoul-Gabriel Urma

Use stream operations to express sophisticated data processing queries.

What would you do without collections? Nearly every Java application *makes and processes* collections. They are fundamental to many programming tasks: they let you group and process data. For example, you might want to create a collection of banking transactions to represent a customer's statement. Then, you might want to process the whole collection to find out how much money the customer spent. Despite their importance, processing collections is far from perfect in Java.

First, typical processing patterns on collections are similar to SQL-like operations such as "finding" (for example, find the transaction with highest value) or "grouping" (for example, group all transactions related to grocery shopping). Most databases let you specify such operations declaratively. For example, the following SQL query lets you find the transaction ID with the highest value: "SELECT id, MAX(value) from transactions".

As you can see, we don't need to implement *how* to calculate the maximum value (for example, using loops and a variable to track the highest value). We only express *what* we expect. This basic idea means that you need to worry less about how to explicitly implement such queries—it is handled for you. Why can't we do something similar with collections? How many times do you find yourself reimplementing these operations using loops over and over again?

Second, how can we process really large collections efficiently? Ideally, to speed up the processing, you want to leverage multicore architectures. However, writing parallel code is hard and error-prone.

Java SE 8 to the rescue! The Java API designers are updating the API with a new abstraction called *Stream* that lets you process data in a declarative way. Furthermore, streams can leverage multi-core architectures without you having to write a single line of multithread code. Sounds good, doesn't it? That's what this series of articles will explore.

Before we explore in detail what you can do with streams, let's take a look at an example so you have a sense of the new programming style with Java SE 8 streams. Let's say we need to find all transactions of type *grocery* and return a list of transaction IDs sorted in decreasing order of transaction value. In Java SE 7, we'd do that as shown in Listing 1. In Java SE 8, we'd do it as shown in Listing 2.

```
List<Transaction> groceryTransactions = new ArrayList<>();
for(Transaction t: transactions){
    if(t.getType() == Transaction.GROCERY){
        groceryTransactions.add(t);
    }
}
Collections.sort(groceryTransactions, new Comparator<>(){
    public int compare(Transaction t1, Transaction t2){
        return t2.getValue().compareTo(t1.getValue());
    }
});
List<Integer> transactionIds = new ArrayList<>();
for(Transaction t: groceryTransactions){
    transactionIds.add(t.getId());
}
```



Originally published in the March/April 2014 issue of *Java Magazine*. Subscribe today.

Here's a mind-blowing idea:
these two operations can produce elements "forever."

Other Digital Learning Resources

- Another excellent source of material to consult is the book *Java 8 in Action*
 - There are also good online articles based on this book

Part 2: Processing Data with Java SE 8 Streams

by Raoul-Gabriel Urma

Published May 2014

Combine advanced operations of the Stream API to express rich data processing queries.

In the first part of this series, you saw that streams let you process collections with database-like operations. As a refresher, the example in Listing 1 shows how to sum the values of only expensive transactions using the Stream API. We set up a pipeline of operations consisting of intermediate operations (`filter`, `map`) and a terminal operation (`reduce`), as illustrated in Figure 1.

```
int sumExpensive =  
    transactions.stream()  
        .filter(t -> t.getValue() > 1000)  
        .map(Transaction::getValue)  
        .reduce(0, Integer::sum);
```

Listing 1

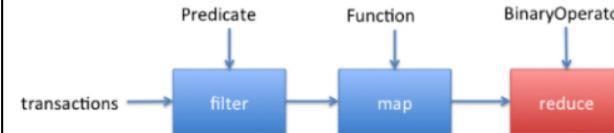


Figure 1

However, the first part of this series didn't investigate two operations:

- `flatMap`: An intermediate operation that lets you combine a "map" and a "flatten" operation
- `collect`: A terminal operation that takes as an argument various recipes (called collectors) for accumulating the elements of a stream into a summary result

These two operations are useful for expressing more-complex queries. For instance, you can combine `flatMap` and `collect` to produce a Map representing the number of occurrences of each character that appears in a stream of words, as shown in Listing 2. Don't worry if this code seems overwhelming at first. The purpose of this article is to explain and explore these two operations in more detail.

```
import static java.util.function.Function.identity;  
import static java.util.stream.Collectors.*;  
  
Stream<String> words = Stream.of("Java", "Magazine", "is",  
    "the", "best");  
  
Map<String, Long> letterToCount =  
    words.map(w -> w.split(""))  
        .flatMap(Arrays::stream)  
        .collect(groupingBy(identity(), counting()));
```



Originally published in the
May/June 2014 issue of *Java Magazine*. Subscribe today.

End of Course Overview

Overview of Java 8 Foundations

Douglas C. Schmidt

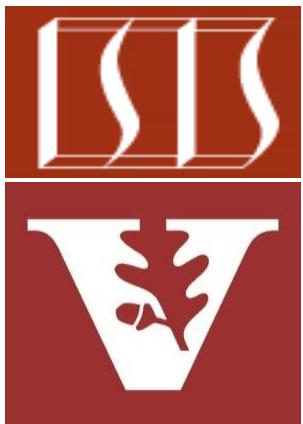
d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

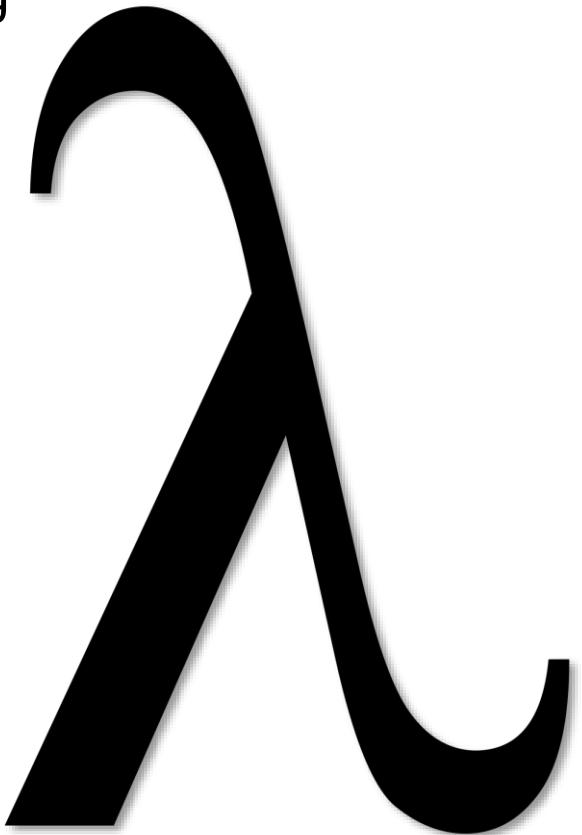
Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA



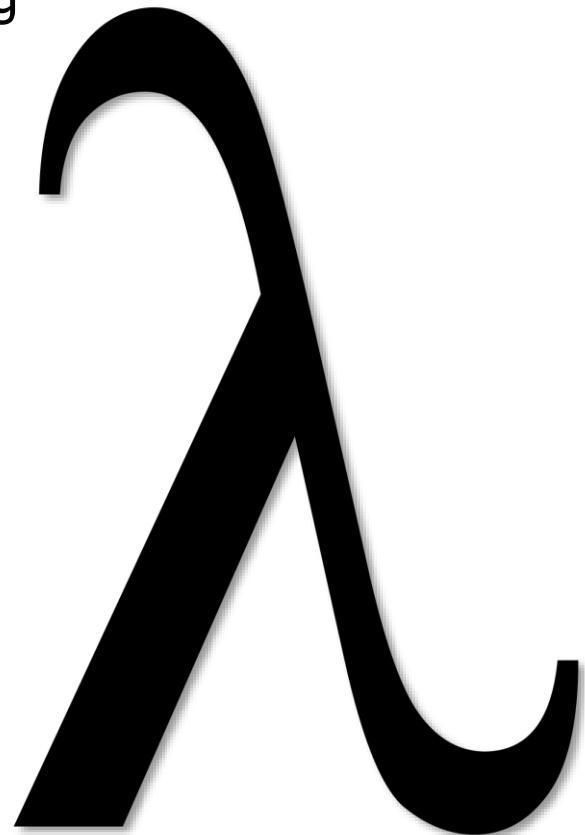
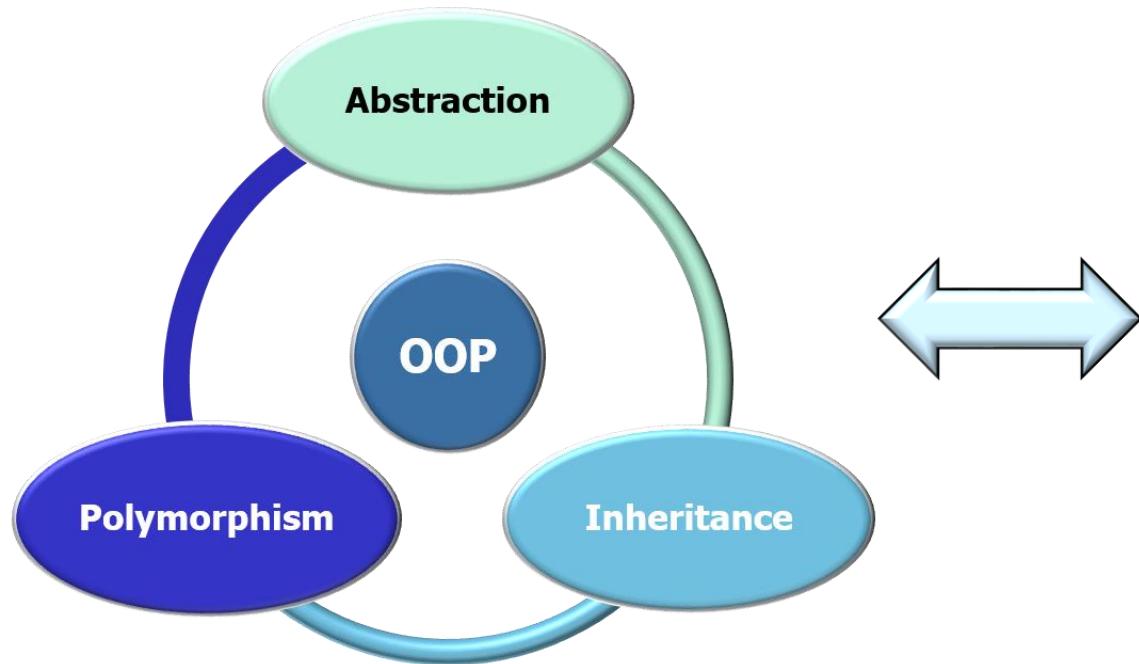
Learning Objectives in this Lesson

- Understand key aspects of functional programming



Learning Objectives in this Lesson

- Understand key aspects of functional programming
 - Contrasted with object-oriented programming



We'll show some Java 8 code fragments that will be covered in more detail later

Learning Objectives in this Lesson

- Understand key aspects of functional programming
- Recognize the benefits of applying functional programming in Java 8



Learning Objectives in this Lesson

- Understand key aspects of functional programming
- Recognize the benefits of applying functional programming in Java 8
 - Especially when used in conjunction with object-oriented programming

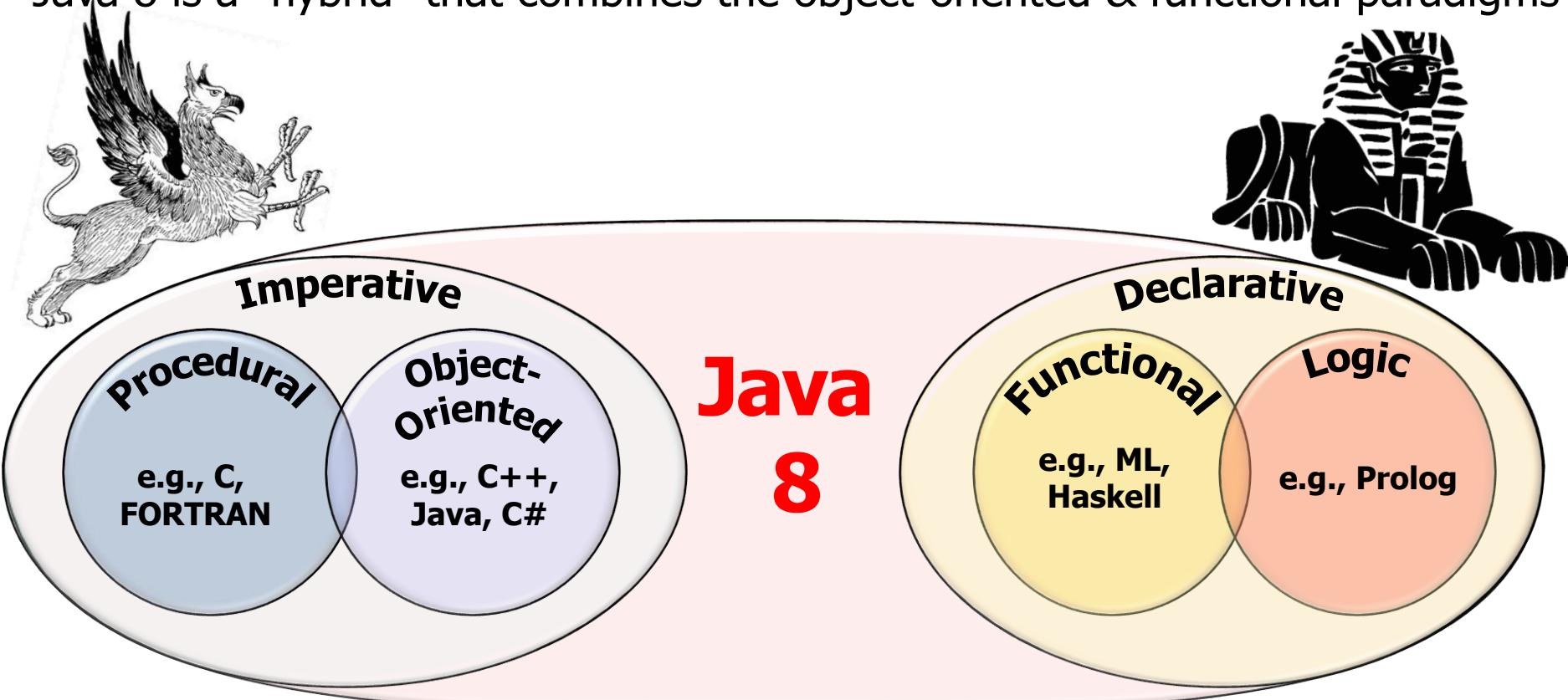


Again, we'll show Java 8 code fragments that'll be covered in more detail later

Overview of Programming Paradigms in Java 8

Overview of Programming Paradigms in Java 8

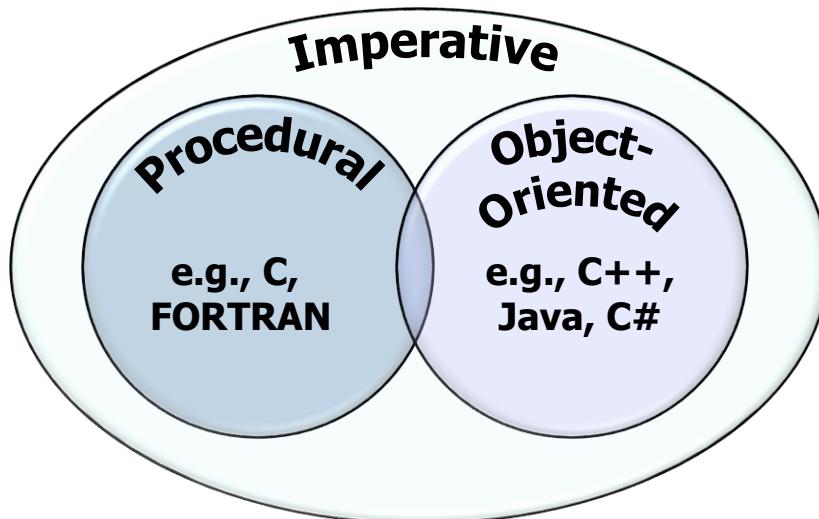
- Java 8 is a “hybrid” that combines the object-oriented & functional paradigms



See www.deadcoderising.com/why-you-should-embrace-lambdas-in-java-8

Overview of Programming Paradigms in Java 8

- Object-oriented programming is an “imperative” paradigm

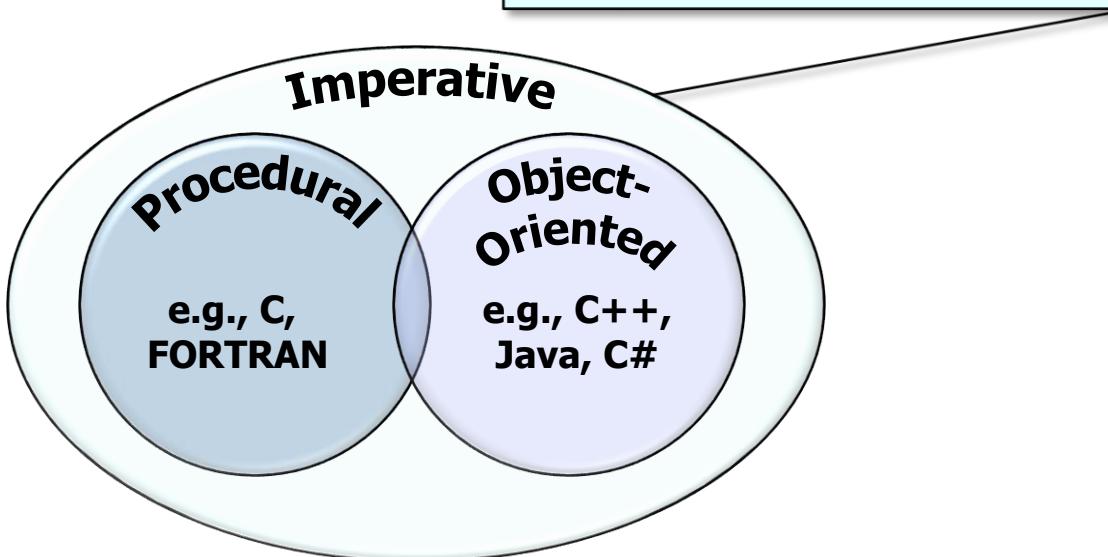


See en.wikipedia.org/wiki/Imperative_programming

Overview of Programming Paradigms in Java 8

- Object-oriented programming is an “imperative” paradigm
 - e.g., a program consists of commands for the computer to perform

Imperative programming focuses on describing how a program operates via statements that change its state

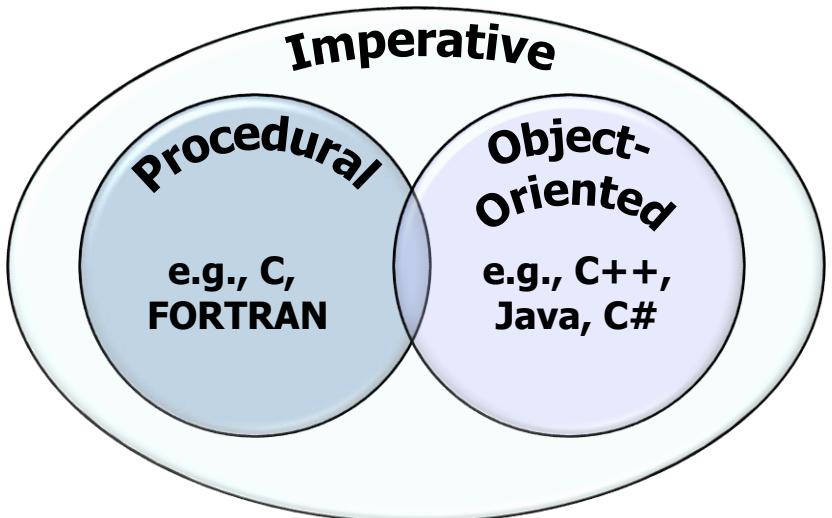


Overview of Programming Paradigms in Java 8

- Object-oriented programming is an “imperative” paradigm
 - e.g., a program consists of commands for the computer to perform

```
List<String> zap(List<String> lines,  
                  String omit) {  
  
    List<String> res =  
        new ArrayList<>();  
    for (String line : lines)  
        if (!omit.equals(line))  
            res.add(line);  
    return res;  
}
```

Imperatively remove a designated string from a list of strings

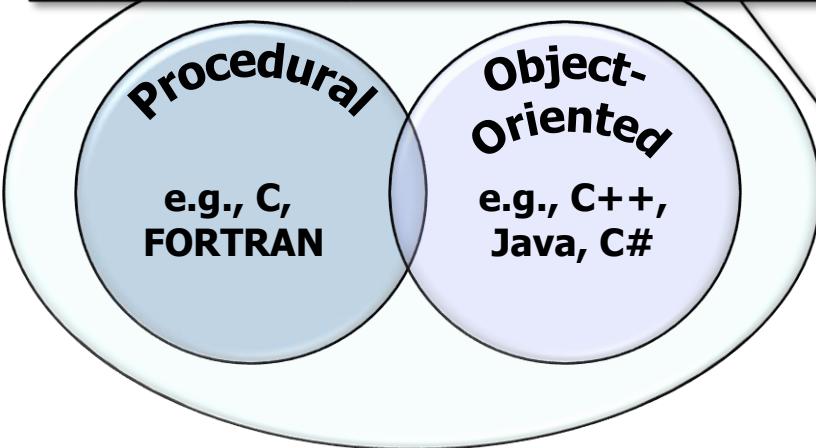


Overview of Programming Paradigms in Java 8

- Object-oriented programming is an “imperative” paradigm
 - e.g., a program consists of commands for the computer to perform

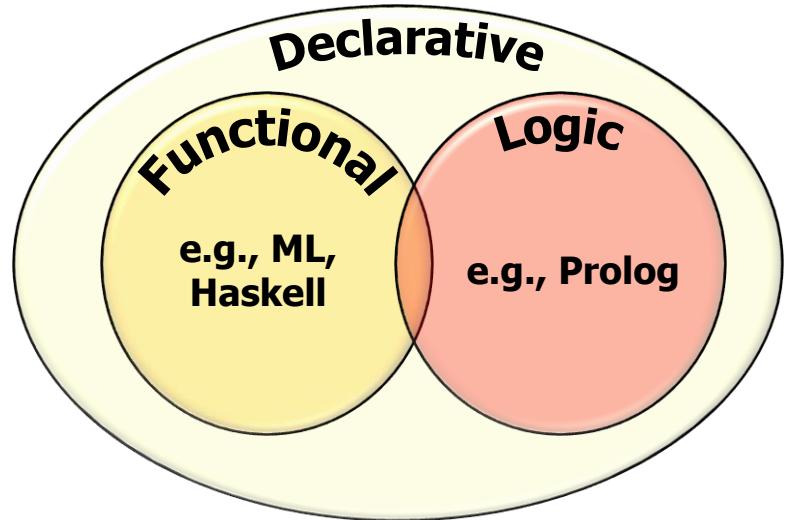
```
List<String> zap(List<String> lines,  
                  String omit) {  
  
    List<String> res =  
        new ArrayList<>();  
    for (String line : lines)  
        if (!omit.equals(line))  
            res.add(line);  
  
    return res;  
}
```

This inherently sequential code applies the Accumulator anti-pattern



Overview of Programming Paradigms in Java 8

- Conversely, functional programming is a “declarative” paradigm

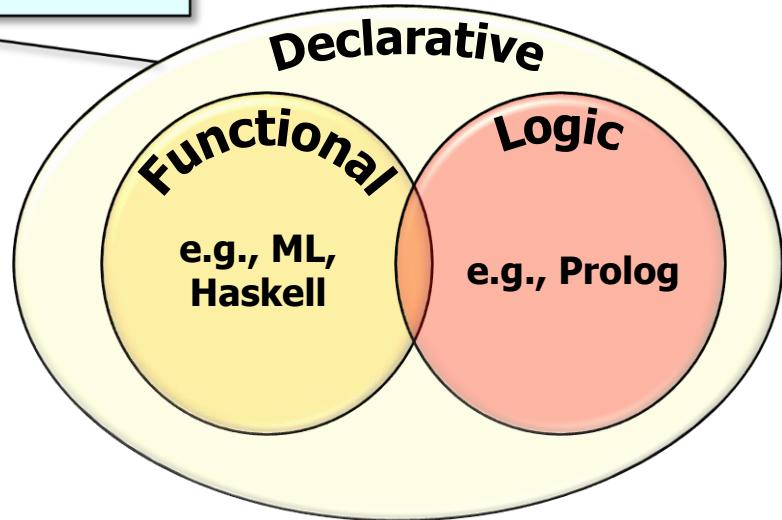


See en.wikipedia.org/wiki/Declarative_programming

Overview of Programming Paradigms in Java 8

- Conversely, functional programming is a “declarative” paradigm
 - e.g., a program expresses computational logic *without* describing control flow or explicit algorithmic steps

Declarative programming focuses on “what” computations to perform, not “how” to compute them

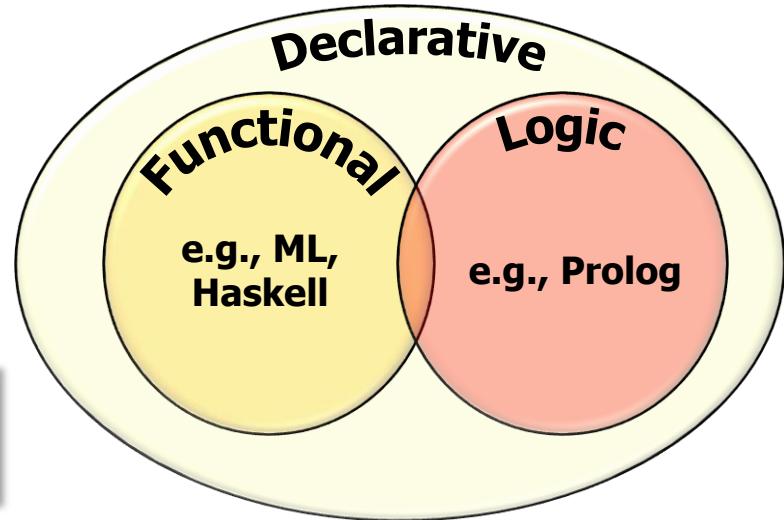


Overview of Programming Paradigms in Java 8

- Conversely, functional programming is a “declarative” paradigm
 - e.g., a program expresses computational logic *without* describing control flow or explicit algorithmic steps

```
List<String> zap(List<String> lines,  
                  String omit) {  
  
    return lines  
        .stream()  
        .filter(not(omit::equals))  
        .collect(toList());  
}
```

Declaratively remove a designated string from a list of strings

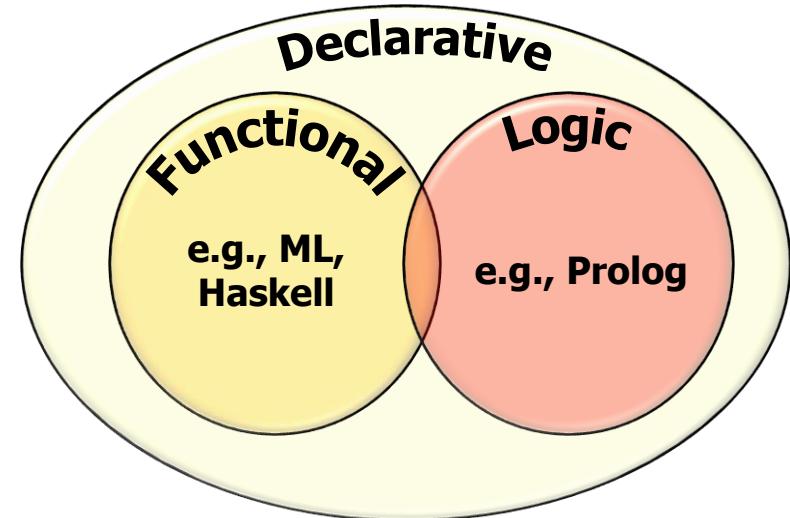


Overview of Programming Paradigms in Java 8

- Conversely, functional programming is a “declarative” paradigm
 - e.g., a program expresses computational logic *without* describing control flow or explicit algorithmic steps

```
List<String> zap(List<String> lines,  
                  String omit) {  
  
    return lines  
        .stream()  
        .filter(not(omit::equals))  
        .collect(toList());  
}
```

Note “fluent” programming style
with cascading method calls



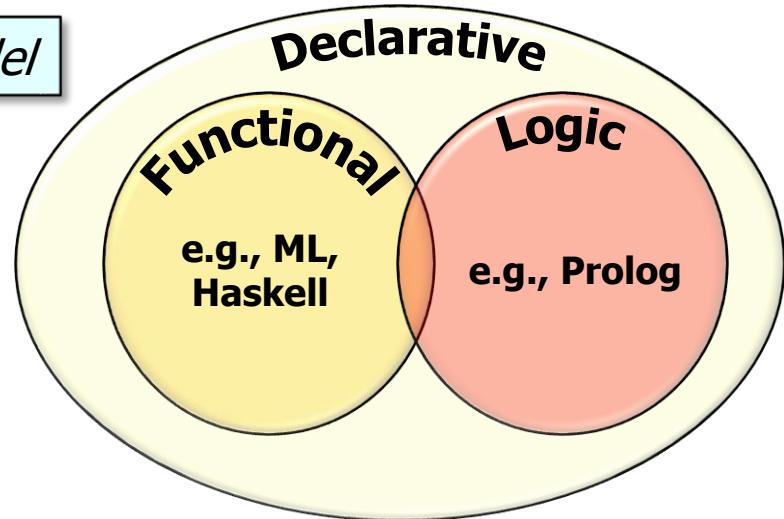
Overview of Programming Paradigms in Java 8

- Conversely, functional programming is a “declarative” paradigm
 - e.g., a program expresses computational logic *without* describing control flow or explicit algorithmic steps

```
List<String> zap(List<String> lines,  
                  String omit) {  
  
    return lines  
        .parallelStream()  
        .filter(not(omit::equals))  
        .collect(toList());  
}
```



Filter in parallel



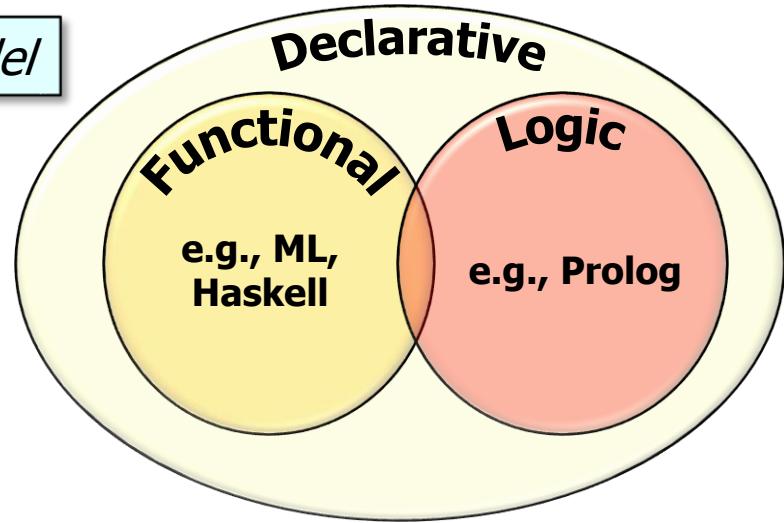
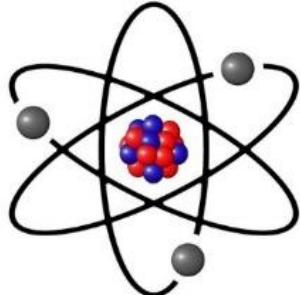
See docs.oracle.com/javase/tutorial/collections/streams/parallelism.html

Overview of Programming Paradigms in Java 8

- Conversely, functional programming is a “declarative” paradigm
 - e.g., a program expresses computational logic *without* describing control flow or explicit algorithmic steps

```
List<String> zap(List<String> lines,  
                  String omit) {  
  
    return lines  
        .parallelStream()  
        .filter(not(omit::equals))  
        .collect(toList());  
}
```

Filter in parallel

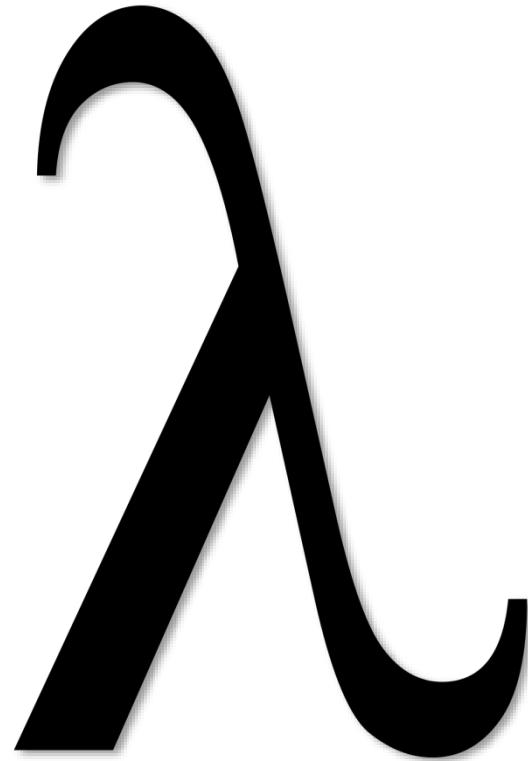


Code was parallelized with minuscule changes since it's declarative & stateless!

Overview of Functional Programming in Java 8

Overview of Functional Programming in Java 8

- Functional programming has its roots in lambda calculus

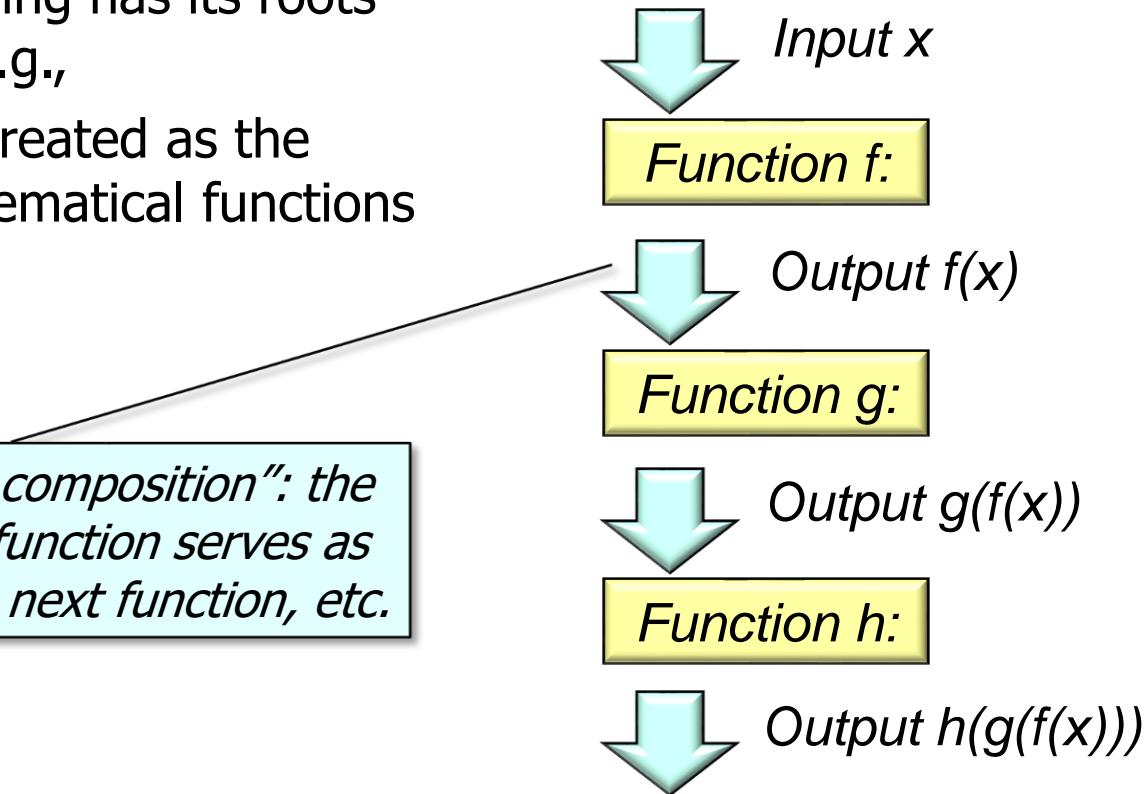


See en.wikipedia.org/wiki/Functional_programming

Overview of Functional Programming in Java 8

- Functional programming has its roots in lambda calculus, e.g.,
 - Computations are treated as the evaluation of mathematical functions

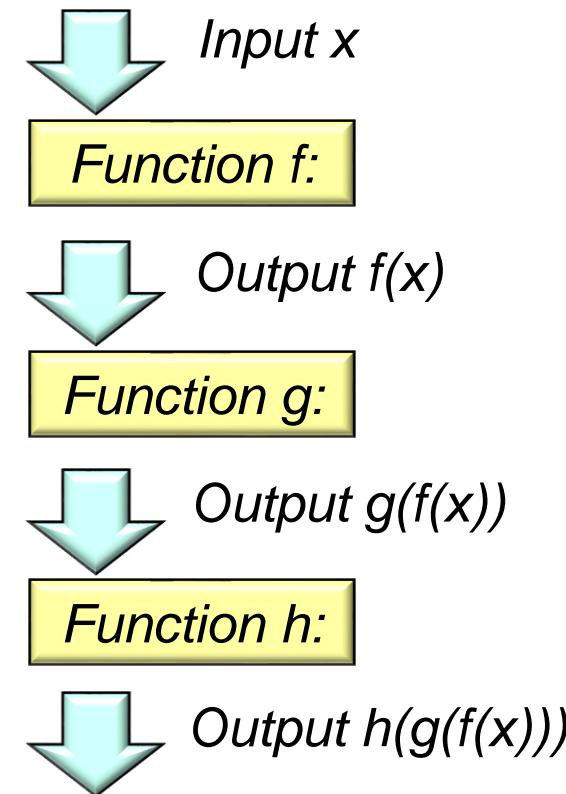
Note "function composition": the output of one function serves as the input to the next function, etc.



Overview of Functional Programming in Java 8

- Functional programming has its roots in lambda calculus, e.g.,
 - Computations are treated as the evaluation of mathematical functions

```
long factorial(long n) {  
    return LongStream  
        .rangeClosed(1, n)  
        .parallel()  
        .reduce(1, (a, b) -> a * b);  
}
```



Overview of Functional Programming in Java 8

- Functional programming has its roots in lambda calculus, e.g.,
 - Computations are treated as the evaluation of mathematical functions
 - Changing state & mutable shared data are discouraged to avoid various hazards



See [en.wikipedia.org/wiki/Side_effect_\(computer_science\)](https://en.wikipedia.org/wiki/Side_effect_(computer_science))

Overview of Functional Programming in Java 8

- Functional programming has its roots in lambda calculus, e.g.,
 - Computations are treated as the evaluation of mathematical functions
 - Changing state & mutable shared data are discouraged to avoid various hazards

```
long factorial(long n) {  
    Total t = new Total();  
    LongStream.rangeClosed(1, n)  
        .parallel()  
        .forEach(t::mult);  
  
    return t.mTotal;  
}
```

```
class Total {  
    public long mTotal = 1;  
  
    public void mult(long n)  
    { mTotal *= n; }  
}
```

Shared mutable state

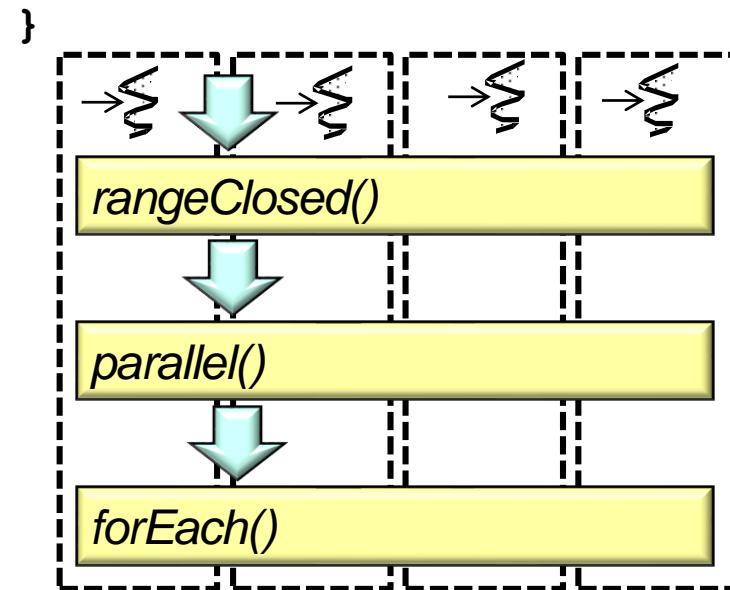


Overview of Functional Programming in Java 8

- Functional programming has its roots in lambda calculus, e.g.,
 - Computations are treated as the evaluation of mathematical functions
 - Changing state & mutable shared data are discouraged to avoid various hazards

```
long factorial(long n) {  
    Total t = new Total();  
    LongStream.rangeClosed(1, n)  
        Run in parallel    .parallel()  
        .forEach(t::mult);  
  
    return t.mTotal;  
}
```

```
class Total {  
    public long mTotal = 1;  
  
    public void mult(long n)  
    { mTotal *= n; }  
}
```



Overview of Functional Programming in Java 8

- Functional programming has its roots in lambda calculus, e.g.,
 - Computations are treated as the evaluation of mathematical functions
 - Changing state & mutable shared data are discouraged to avoid various hazards

```
long factorial(long n) {  
    Total t = new Total();  
    LongStream.rangeClosed(1, n)  
        .parallel()  
        .forEach(t::mult);  
  
    return t.mTotal;  
}
```

```
class Total {  
    public long mTotal = 1;  
  
    public void mult(long n)  
    { mTotal *= n; }  
}
```

Beware of race conditions!!!



See en.wikipedia.org/wiki/Race_condition#Software

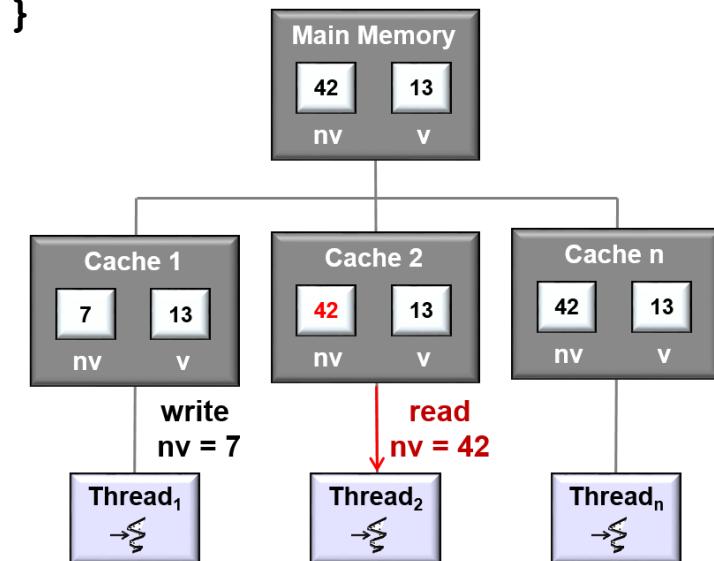
Overview of Functional Programming in Java 8

- Functional programming has its roots in lambda calculus, e.g.,
 - Computations are treated as the evaluation of mathematical functions
 - Changing state & mutable shared data are discouraged to avoid various hazards

```
long factorial(long n) {  
    Total t = new Total();  
    LongStream.rangeClosed(1, n)  
        .parallel()  
        .forEach(t::mult);  
  
    return t.mTotal;  
}
```

Beware of inconsistent memory visibility

```
class Total {  
    public long mTotal = 1;  
  
    public void mult(long n)  
    { mTotal *= n; }  
}
```



Overview of Functional Programming in Java 8

- Functional programming has its roots in lambda calculus, e.g.,
 - Computations are treated as the evaluation of mathematical functions
 - Changing state & mutable shared data are discouraged to avoid various hazards

```
long factorial(long n) {  
    Total t = new Total();  
    LongStream.rangeClosed(1, n)  
        .parallel()  
        .forEach(t::mult);  
  
    return t.mTotal;  
}
```

```
class Total {  
    public long mTotal = 1;  
  
    public void mult(long n)  
    { mTotal *= n; }  
}
```



***Only you can prevent
concurrency hazards!***

In Java *you* must avoid these hazards, i.e., the compiler & JVM won't save you..

Overview of Functional Programming in Java 8

- Functional programming has its roots in lambda calculus, e.g.,
 - Computations are treated as the evaluation of mathematical functions
 - Changing state & mutable shared data are discouraged to avoid various hazards
 - Instead, the focus is on “immutable” objects
 - The state of these objects cannot change after they are constructed



Overview of Functional Programming in Java 8

- Functional programming has its roots in lambda calculus, e.g.,
 - Computations are treated as the evaluation of mathematical functions
 - Changing state & mutable shared data are discouraged to avoid various hazards
 - Instead, the focus is on “immutable” objects
 - The state of these objects cannot change after they are constructed

```
final class String {  
    private final char value[];  
    ...  
  
    public String(String s) {  
        value = s;  
        ...  
    }  
  
    public int length() {  
        return value.length;  
    }  
    ...  
}
```

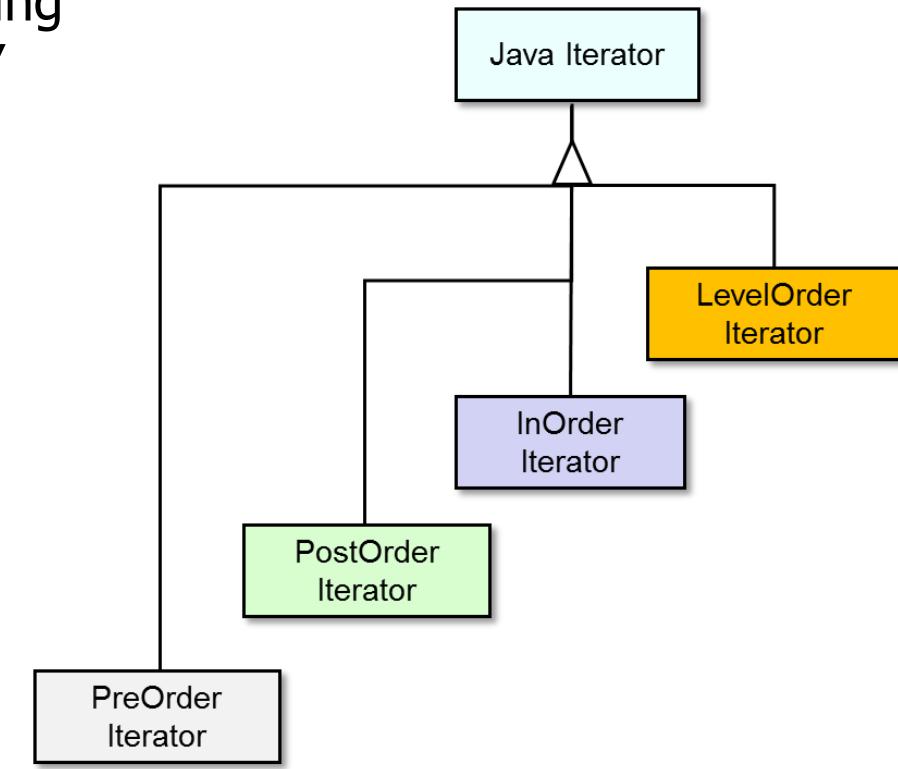
Overview of Functional Programming in Java 8

- Functional programming has its roots in lambda calculus, e.g.,
 - Computations are treated as the evaluation of mathematical functions
 - Changing state & mutable shared data are discouraged to avoid various hazards
 - Instead, the focus is on “immutable” objects
 - The state of these objects cannot change after they are constructed
 - e.g., final fields and/or only accessor methods

```
final class String {  
    private final char value[];  
    ...  
  
    public String(String s) {  
        value = s;  
        ...  
    }  
  
    public int length() {  
        return value.length;  
    }  
    ...  
}
```

Overview of Functional Programming in Java 8

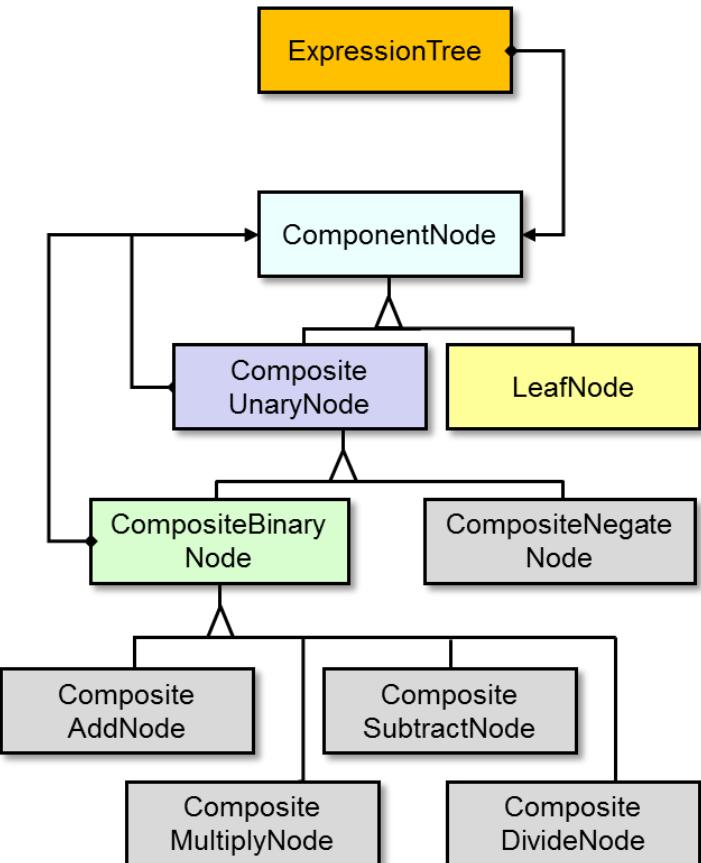
- In contrast, object-oriented programming employs “hierarchical data abstraction”



See en.wikipedia.org/wiki/Object-oriented_design

Overview of Functional Programming in Java 8

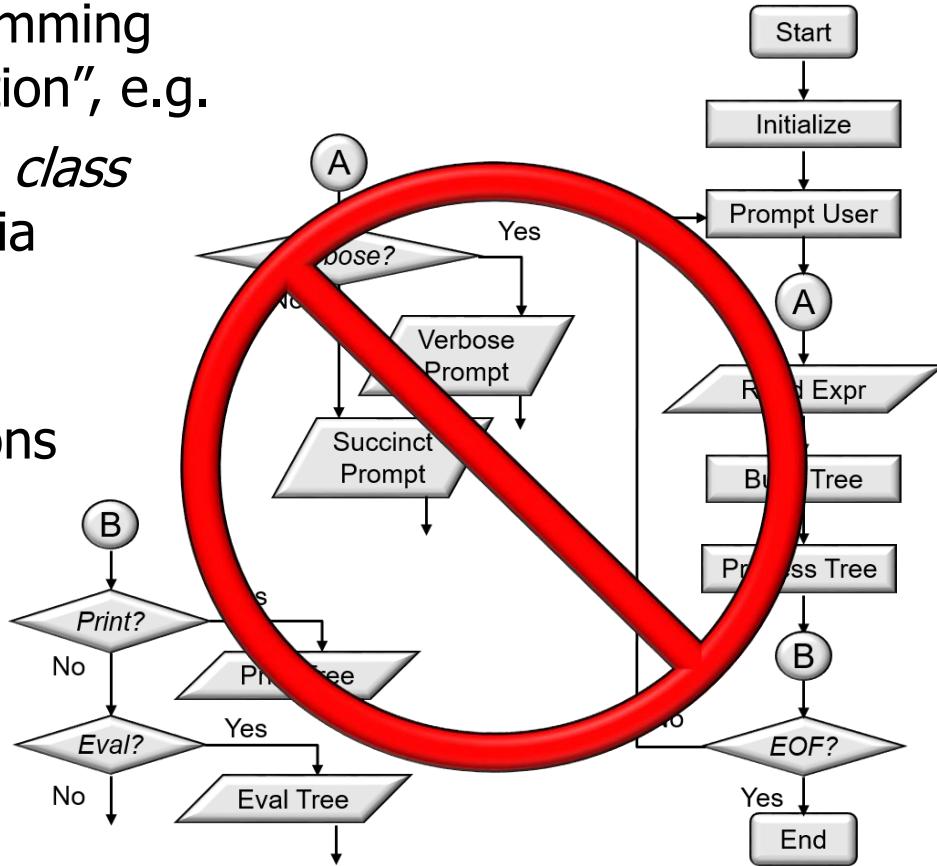
- In contrast, object-oriented programming employs “hierarchical data abstraction”, e.g.
 - Components are based on stable *class* roles & relationships extensible via inheritance & dynamic binding



See en.wikipedia.org/wiki/Object-oriented_programming

Overview of Functional Programming in Java 8

- In contrast, object-oriented programming employs “hierarchical data abstraction”, e.g.
 - Components are based on stable *class* roles & relationships extensible via inheritance & dynamic binding
 - Rather than by functions that correspond to algorithmic actions



Overview of Functional Programming in Java 8

- In contrast, object-oriented programming employs “hierarchical data abstraction”, e.g.
 - Components are based on stable *class* roles & relationships extensible via inheritance & dynamic binding
 - State is encapsulated by methods that perform imperative statements

```
Tree tree = ...;
Visitor printVisitor =
    makeVisitor(...);

for(Iterator<Tree> iter =
    tree.iterator();
    iter.hasNext();
    iter.next()
    .accept(printVisitor);
```

Overview of Functional Programming in Java 8

- In contrast, object-oriented programming employs “hierarchical data abstraction”, e.g.
 - Components are based on stable *class* roles & relationships extensible via inheritance & dynamic binding
 - State is encapsulated by methods that perform imperative statements
 - This state is often mutable



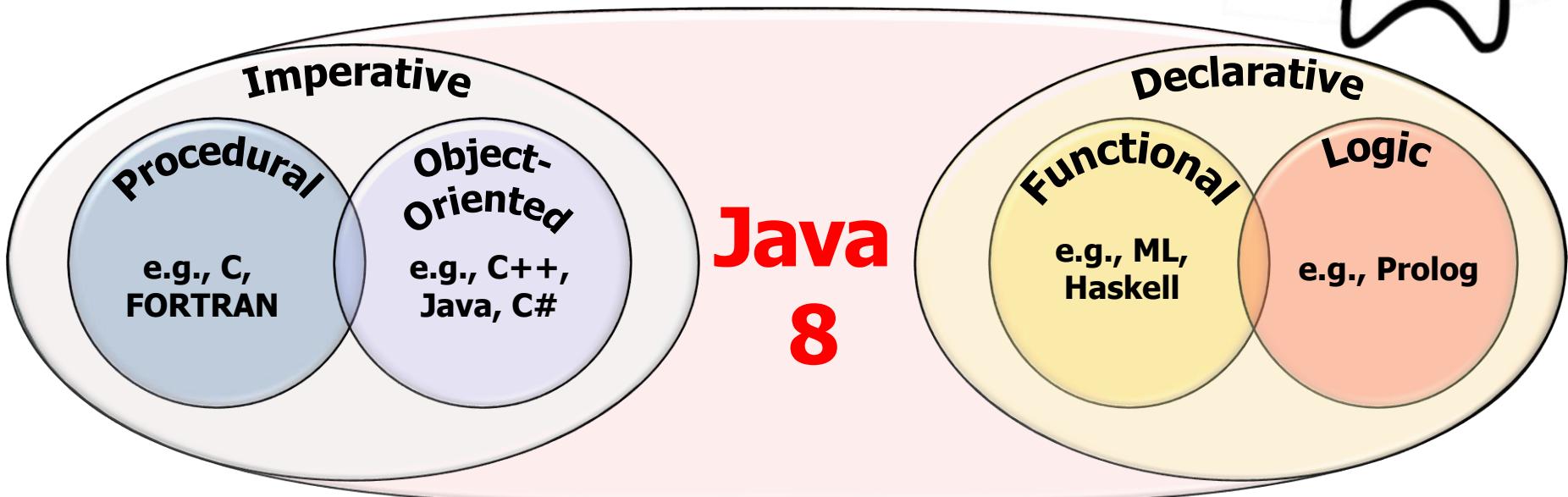
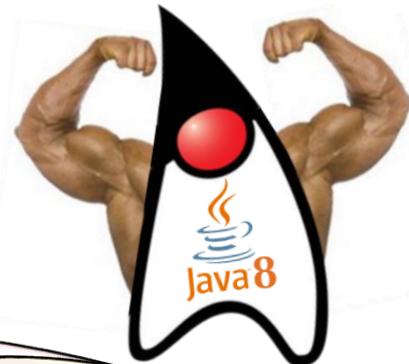
```
Tree tree = ...;  
Visitor printVisitor =  
    makeVisitor(...);  
  
for(Iterator<Tree> iter =  
    tree.iterator();  
    iter.hasNext();)  
    iter.next()  
        .accept(printVisitor);
```

*Access & update internal
state of the iterator*

Combining Object-Oriented (OO) & Functional Programming (FP) in Java 8

Benefits of Combining OO & FP in Java 8

- Java 8's combination of functional & object-oriented paradigms is powerful!



Benefits of Combining OO & FP in Java 8

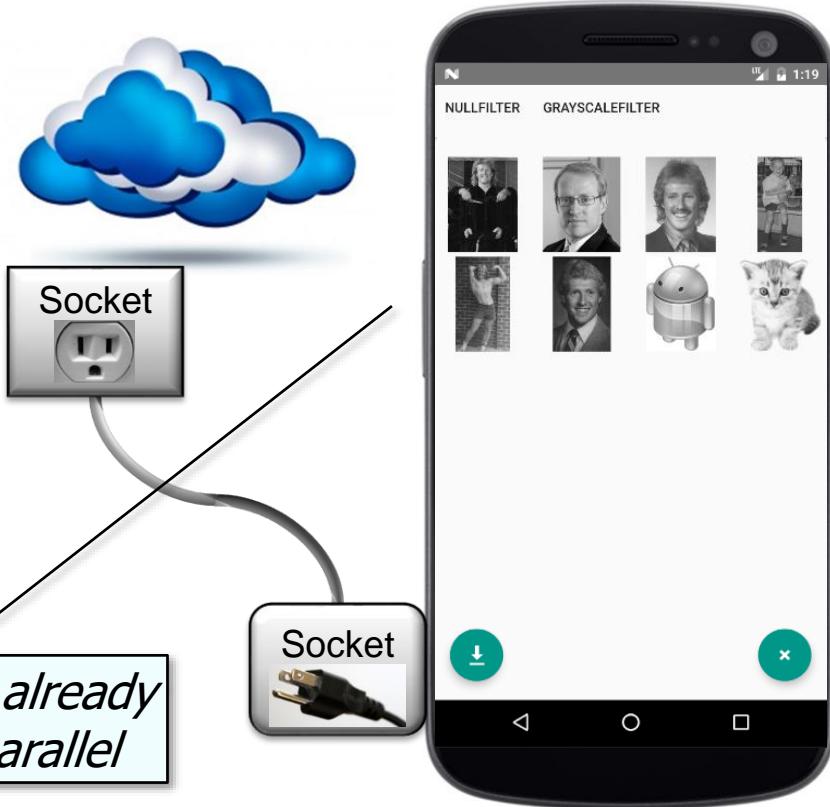
- Java 8's functional features help close the gap between a program's "domain intent" & its computations



See www.toptal.com/software/declarative-programming

Benefits of Combining OO & FP in Java 8

- Java 8's functional features help close the gap between a program's "domain intent" & its computations, e.g.,
 - Domain intent defines "what"

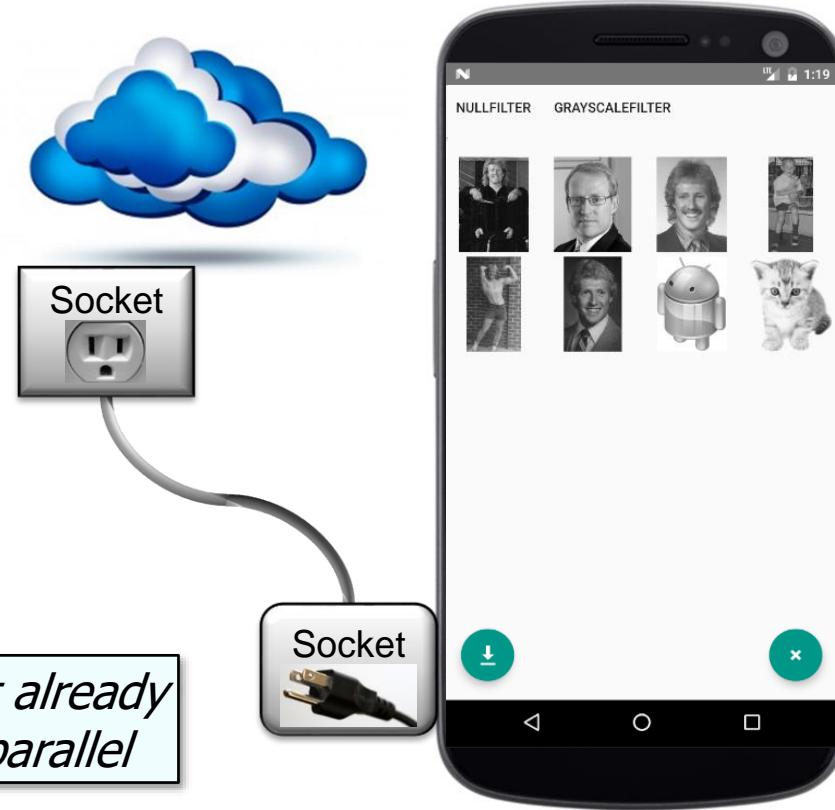


Benefits of Combining OO & FP in Java 8

- Java 8's functional features help close the gap between a program's "domain intent" & its computations, e.g.,
 - Domain intent defines "what"
 - Computations define "how"

```
List<Image> images = urls
    .parallelStream()
    .filter(not(this::urlCached))
    .map(this::downloadImage)
    .flatMap(this::applyFilters)
    .collect(toList());
```

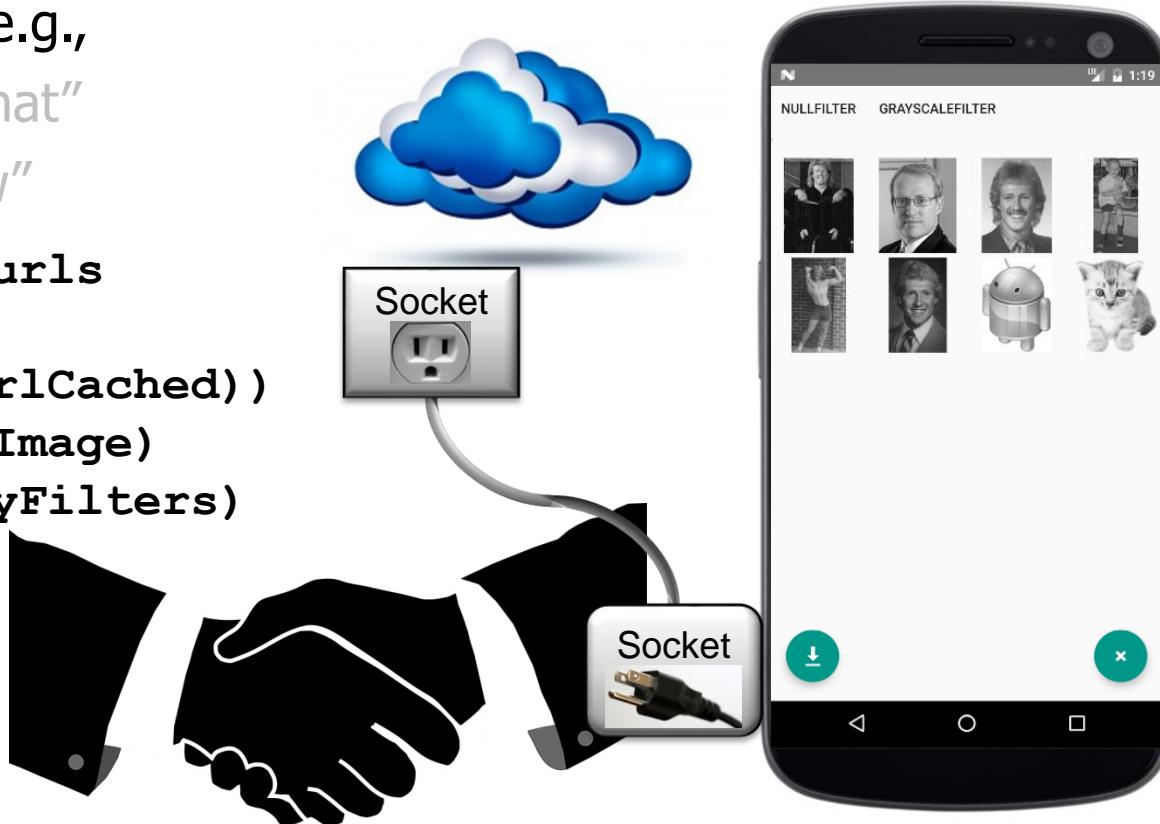
Process a list of URLs to images that aren't already cached & transform/store the images in parallel



Benefits of Combining OO & FP in Java 8

- Java 8's functional features help close the gap between a program's "domain intent" & its computations, e.g.,
 - Domain intent defines "what"
 - Computations define "how"

```
List<Image> images = urls
    .parallelStream()
    .filter(not(this::urlCached))
    .map(this::downloadImage)
    .flatMap(this::applyFilters)
    .collect(toList());
```

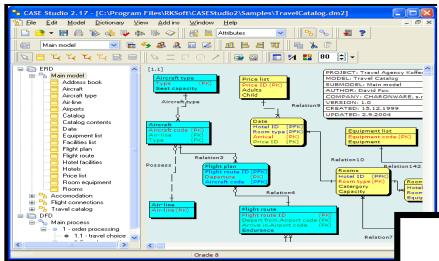


Java 8 functional programming features connect domain intent & computations

Benefits of Combining OO & FP in Java 8

- Likewise, Java 8's object-oriented features help to structure a program's software architecture

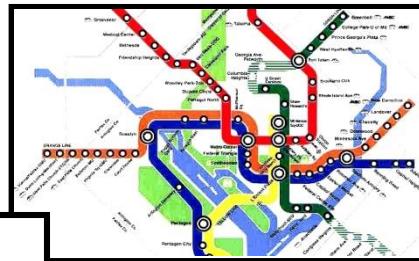
Logical View



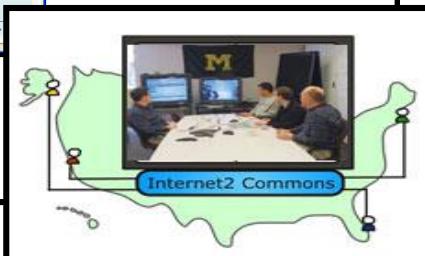
Physical View



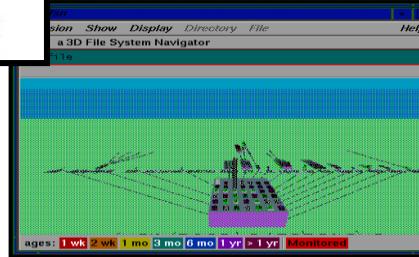
Process View



Use Case View



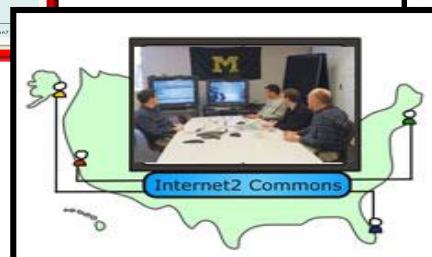
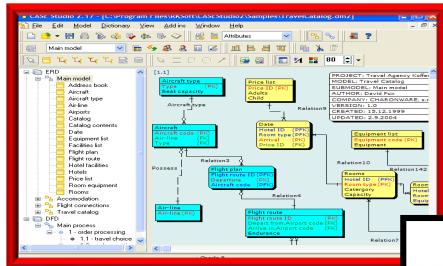
Development View



Benefits of Combining OO & FP in Java 8

- Likewise, Java 8's object-oriented features help to structure a program's software architecture

Logical View

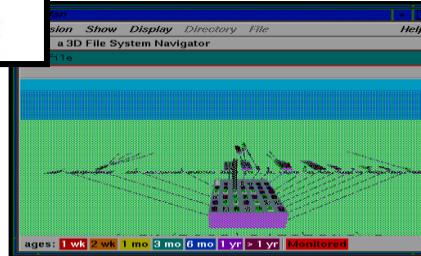


Process View



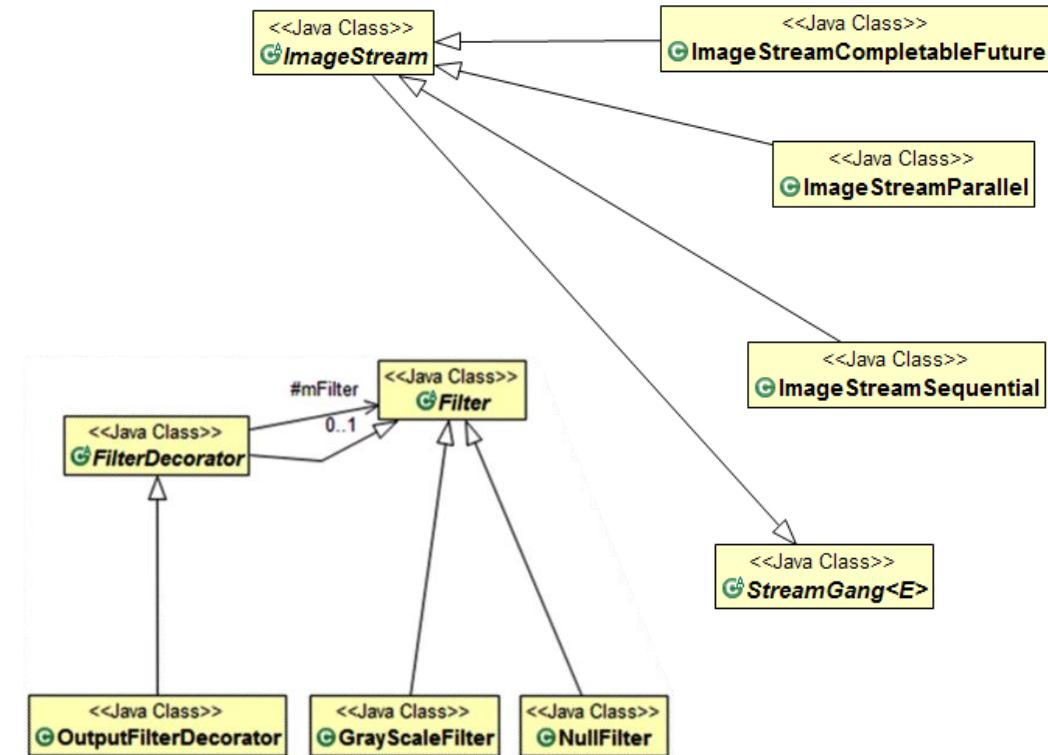
This view depicts key use-case realizations, sub-systems, packages, & classes that encompass architecturally important behavior

Development View



Benefits of Combining OO & FP in Java 8

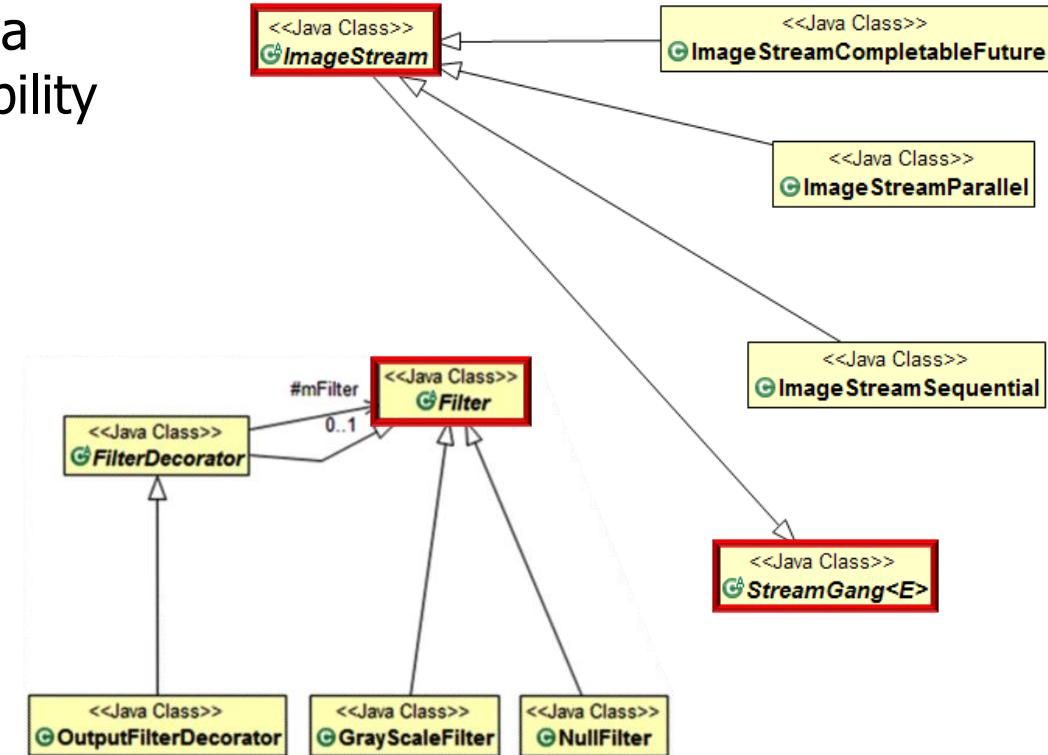
- e.g., consider the ImageStreamGang program



See github.com/douglasraigschmidt/LiveLessons/tree/master/ImageStreamGang

Benefits of Combining OO & FP in Java 8

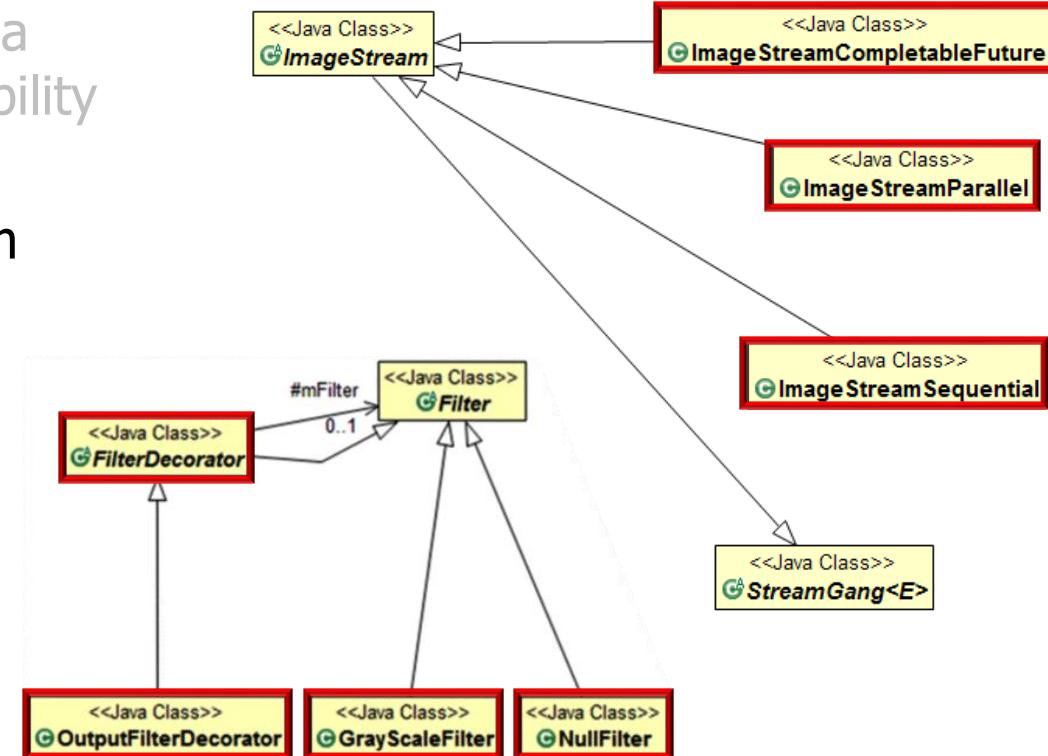
- e.g., consider the ImageStreamGang program
 - Common super classes provide a reusable foundation for extensibility



Benefits of Combining OO & FP in Java 8

- e.g., consider the ImageStreamGang program

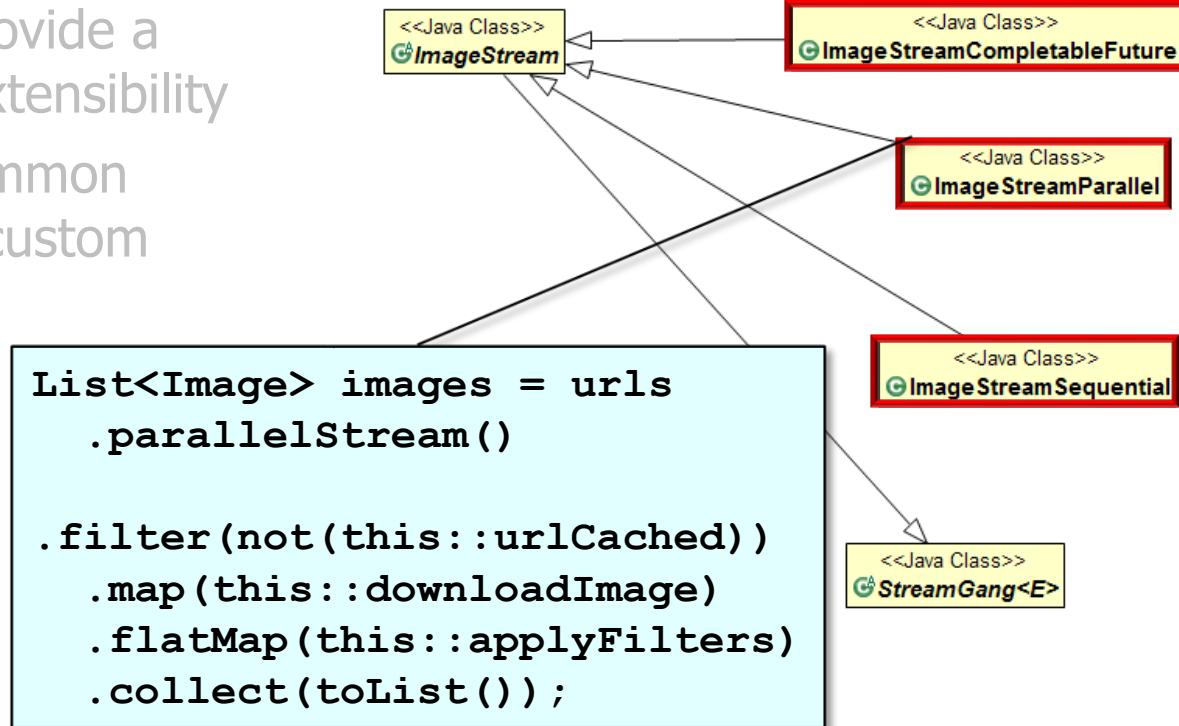
- Common super classes provide a reusable foundation for extensibility
- Subclasses extend the common classes to create various custom implementation strategies



Benefits of Combining OO & FP in Java 8

- e.g., consider the ImageStreamGang program

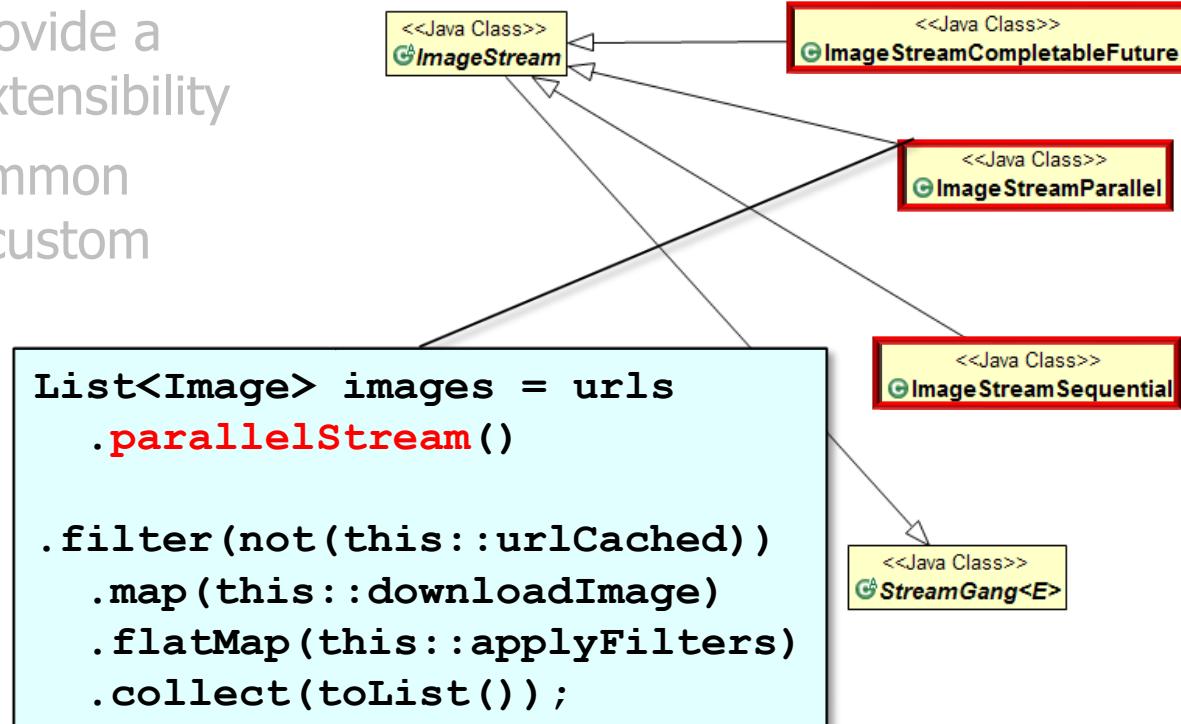
- Common super classes provide a reusable foundation for extensibility
- Subclasses extend the common classes to create various custom implementation strategies
- Java 8's FP features are most effective when used to simplify computations within the context of an OO software architecture



Benefits of Combining OO & FP in Java 8

- e.g., consider the ImageStreamGang program

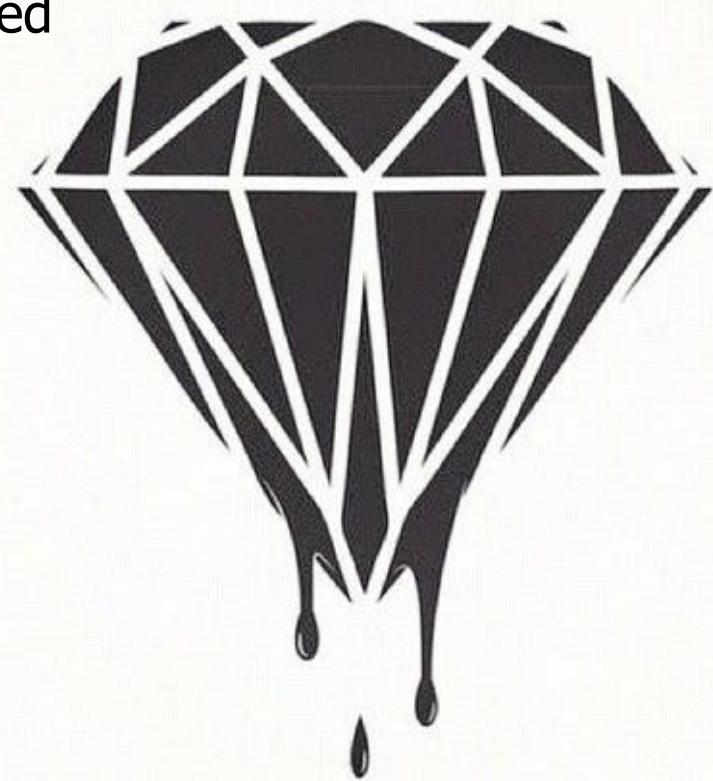
- Common super classes provide a reusable foundation for extensibility
- Subclasses extend the common classes to create various custom implementation strategies
- Java 8's FP features are most effective when used to simplify computations within the context of an OO software architecture
 - Especially concurrent & parallel computations



See docs.oracle.com/javase/tutorial/collections/stream/parallelism.html

Benefits of Combining OO & FP in Java 8

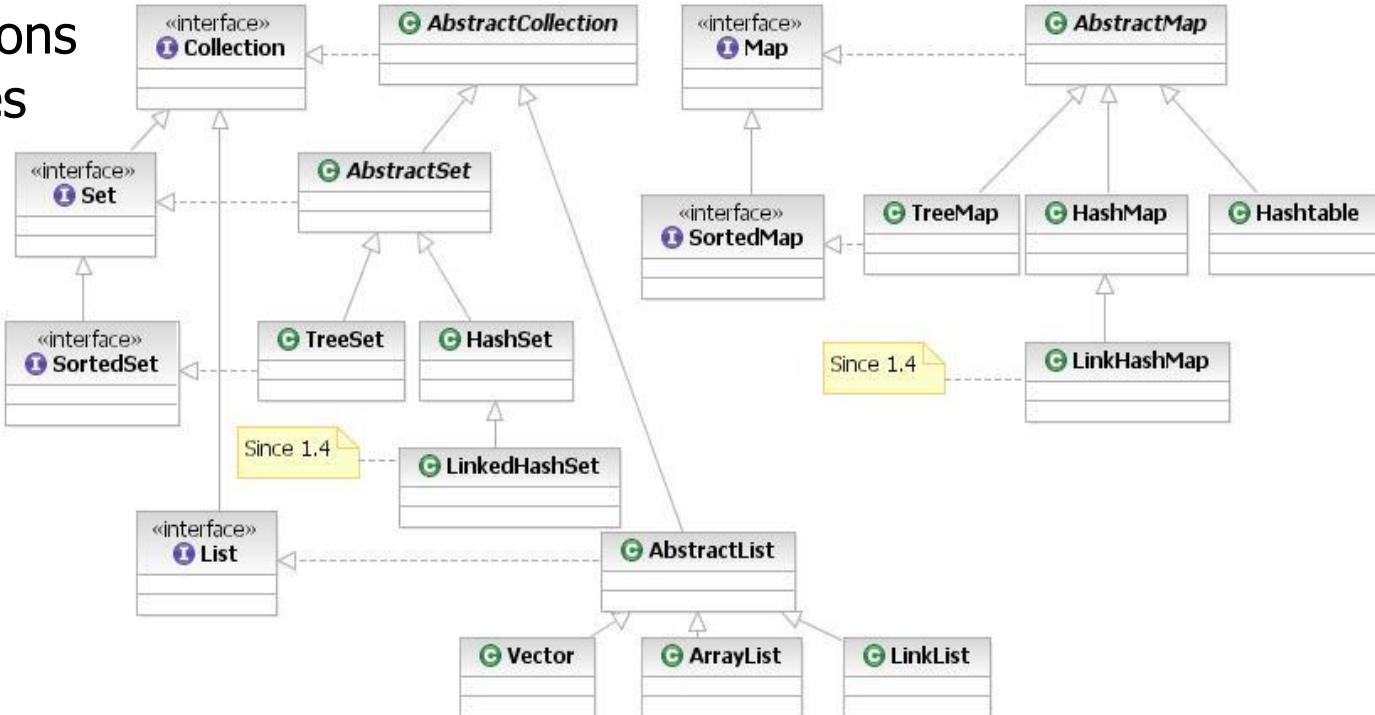
- Since Java 8 is a hybrid language, there are situations in which mutable changes to shared state are allowed/encouraged



See www.infoq.com/articles/How-Functional-is-Java-8

Benefits of Combining OO & FP in Java 8

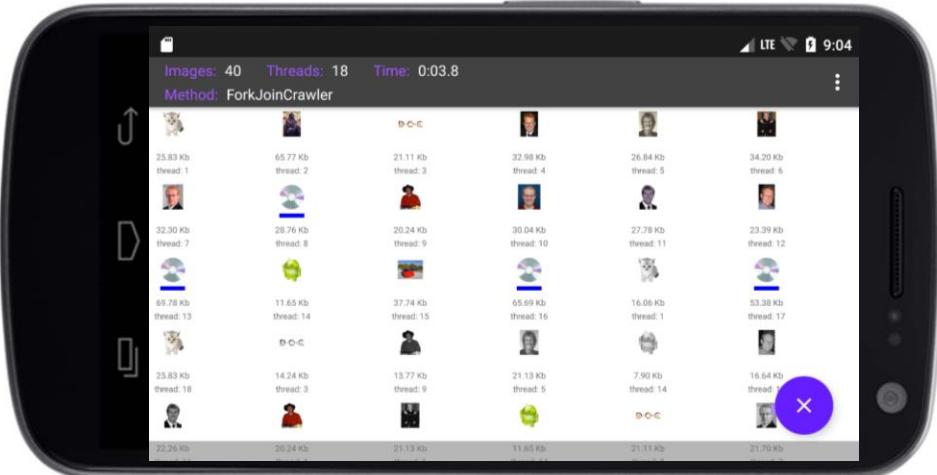
- Since Java 8 is a hybrid language, there are situations in which mutable changes to shared state are allowed/encouraged
 - e.g., Java collections framework classes



See docs.oracle.com/javase/8/docs/technotes/guides/collections/

Benefits of Combining OO & FP in Java 8

- However, you're usually better off by minimizing/avoiding the use of shared mutable state in *your* programs!!



End of Overview of Java 8 Foundations

Overview of Java 8 Lambda Expressions & Method References

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA



Learning Objectives in this Lesson

- Recognize foundational functional programming features in Java 8, e.g.,
 - Lambda expressions



Learning Objectives in this Lesson

- Recognize foundational functional programming features in Java 8, e.g.,
 - Lambda expressions
 - Method (& constructor) references



Learning Objectives in this Lesson

- Recognize foundational functional programming features in Java 8, e.g.,
 - Lambda expressions
 - Method (& constructor) references



Several concise examples are used to showcase foundational Java 8 features

Overview of Lambda Expressions

Overview of Lambda Expressions

- A *lambda expression* is an unnamed block of code (with optional parameters) that can be stored, passed around, & executed later

```
new Thread() ->  
    System.out.println("hello world"))  
    .start();
```

Overview of Lambda Expressions

- A *lambda expression* is an unnamed block of code (with optional parameters) that can be stored, passed around, & executed later, e.g.,

```
new Thread( () ->
```

This lambda expression takes no parameters, i.e., "()"

```
    System.out.println("hello world"))  
    .start();
```

Overview of Lambda Expressions

- A *lambda expression* is an unnamed block of code (with optional parameters) that can be stored, passed around, & executed later, e.g.,

```
new Thread( () ->  
    System.out.println("hello world"))  
.start();
```

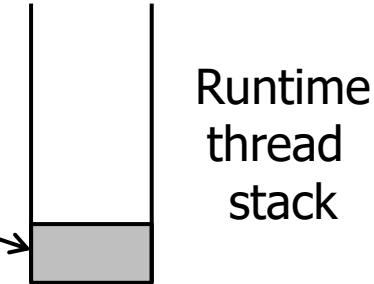
*It defines a computation that will
run in a separate Java thread*

Overview of Lambda Expressions

- A *lambda expression* is an unnamed block of code (with optional parameters) that can be stored, passed around, & executed later, e.g.,

```
new Thread(() ->  
    System.out.println("hello world"))  
.start();
```

*It defines a computation that will
run in a separate Java thread*



Overview of Lambda Expressions

- A *lambda expression* is an unnamed block of code (with optional parameters) that can be stored, passed around, & executed later, e.g.,

```
new Thread( () ->  
    System.out.println("hello world"))  
    .start();
```

```
Runnable r = () -> System.out.println("hello world");  
new Thread(r).start();
```

You can also store a lambda expression into
a variable & pass that variable to a method

Overview of Lambda Expressions

- A *lambda expression* is an unnamed block of code (with optional parameters) that can be stored, passed around, & executed later, e.g.,

```
new Thread() ->  
    System.out.println("hello world")  
    .start();
```

Lambda expressions are compact since they just focus on computation(s) to perform



Overview of Lambda Expressions

- A *lambda expression* is an unnamed block of code (with optional parameters) that can be stored, passed around, & executed later, e.g.,

```
new Thread( () ->  
    System.out.println("hello world") )  
.start();
```

vs

Conversely, this anonymous inner class requires more code to write each time

```
new Thread(new Runnable() {  
    public void run() {  
        System.out.println("hello world");  
    } }).start();
```



Overview of Lambda Expressions

- Lambda expressions can work with multiple parameters in a *much* more compact manner than anonymous inner classes

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
                      "Robert", "Michael", "Linda", "james", "mary"};
```

```
Arrays.sort(nameArray, new Comparator<String>() {
    public int compare(String s, String t) { return
        s.toLowerCase().compareTo(t.toLowerCase()); }});
```

VS

```
Arrays.sort(nameArray,
            (s, t) -> s.compareToIgnoreCase(t));
```

Overview of Lambda Expressions

- Lambda expressions can work with multiple parameters in a *much* more compact manner than anonymous inner classes, e.g.

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
    "Robert", "Michael", "Linda", "james", "mary"};
```

Array of names represented as strings

```
Arrays.sort(nameArray, new Comparator<String>() {
    public int compare(String s, String t) { return
        s.toLowerCase().compareTo(t.toLowerCase()); }});
```

VS

```
Arrays.sort(nameArray,
    (s, t) -> s.compareToIgnoreCase(t));
```

Overview of Lambda Expressions

- Lambda expressions can work with multiple parameters in a *much* more compact manner than anonymous inner classes, e.g.

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
    "Robert", "Michael", "Linda", "james", "mary"};
```

```
Arrays.sort(nameArray, new Comparator<String>() {
    public int compare(String s, String t) { return
        s.toLowerCase().compareTo(t.toLowerCase()); }});
```



*Extraneous syntax for
anonymous inner class*

Overview of Lambda Expressions

- Lambda expressions can work with multiple parameters in a *much* more compact manner than anonymous inner classes, e.g.

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
                      "Robert", "Michael", "Linda", "james", "mary"};
```

```
Arrays.sort(nameArray, new Comparator<String>() {
    public int compare(String s, String t) { return
        s.toLowerCase().compareTo(t.toLowerCase()); }});
```

VS

```
Arrays.sort(nameArray,
            (s, t) -> s.compareToIgnoreCase(t));
```



(s, t) is short for *(String s, String t)*, which leverages Java 8's type inference capabilities

See docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html

Overview of Lambda Expressions

- Lambda expressions can work with multiple parameters in a *much* more compact manner than anonymous inner classes, e.g.

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
    "Robert", "Michael", "Linda", "james", "mary"};
```

```
Arrays.sort(nameArray, new Comparator<String>() {
    public int compare(String s, String t) { return
        s.toLowerCase().compareTo(t.toLowerCase()); }});
```

VS

```
Arrays.sort(nameArray,
    (s, t) -> s.compareToIgnoreCase(t));
```



This lambda expression omits the method name & extraneous syntax

Overview of Lambda Expressions

- A lambda expression can access (effectively) final variables from the enclosing scope

```
int answer = 42;  
new Thread(() ->  
    System.out.println("The answer is " + answer))  
.start();
```

This lambda expression can access the value of "answer," which is an effectively final variable whose value never changes after it's initialized

Overview of Method References

Overview of Method References

- A method reference is a compact, easy-to-read handle for a method that already has a name

Kind	Syntax	Method Reference	Lambda Expression
1. Reference to a static method	ContainingClass:: staticMethodName	String::valueOf	s -> String.valueOf(s)
2. Reference to an instance method of a particular object	containingObject:: instanceMethodName	s::toString	() -> "string".toString()
3. Reference to instance method of an arbitrary object of a given type	ContainingType:: methodName	String::toString	s -> s.toString()
4. Reference to a constructor	ClassName::new	String::new	() -> new String()

See docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html

Overview of Method References

- A method reference is a compact, easy-to-read handle for a method that already has a name

Kind	Syntax	Method Reference	Lambda Expression
1. Reference to a static method	ContainingClass::staticMethodName	String::valueOf	s -> String.valueOf(s)
2. Reference to an instance method of a particular object	containingObject::instanceMethodName	s::toString	() -> "string".toString()
3. Reference to instance method of an arbitrary object of a given type	ContainingType::methodName	String::toString	s -> s.toString()
4. Reference to a constructor	ClassName::new	String::new	() -> new String()

It's shorthand syntax for a lambda expression that executes one method

Overview of Method References

- Method references are more compact than alternative mechanisms



Overview of Method References

- Method references are more compact than alternative mechanisms, e.g.,

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
                      "Robert", "Michael", "Linda", "james", "mary"};
```

```
Arrays.sort(nameArray, new Comparator<String>() {
    public int compare(String s, String t) { return
        s.toLowerCase().compareTo(t.toLowerCase()); }});
```

VS

```
Arrays.sort(nameArray,
            (s, t) -> s.compareToIgnoreCase(t));
```

VS

Method references are even more compact & readable

```
Arrays.sort(nameArray, String::compareToIgnoreCase);
```



Overview of Method References

- Method references are more compact than alternative mechanisms, e.g.,

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
                      "Robert", "Michael", "Linda", "james", "mary"};
```

```
Arrays.sort(nameArray, new Comparator<String>() {
    public int compare(String s, String t) { return
        s.toLowerCase().compareTo(t.toLowerCase()); }});
```

VS

```
Arrays.sort(nameArray,
            (s, t) -> s.compareToIgnoreCase(t));
```

VS

Method references also promote code reuse

```
Arrays.sort(nameArray, String::compareToIgnoreCase);
```



Overview of Method References

- Method references are more compact than alternative mechanisms, e.g.,

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
                      "Robert", "Michael", "Linda", "james", "mary"};
```

```
Arrays.sort(nameArray, new Comparator<String>() {
    public int compare(String s, String t) { return
        s.toLowerCase().compareTo(t.toLowerCase()); }});
```

VS

```
Arrays.sort(nameArray,
            (s, t) -> s.compareToIgnoreCase(t));
```



VS

```
Arrays.sort(nameArray, String::compareToIgnoreCase);
```



It's therefore good practice to use method references whenever you can!

Applying Method References in Practice

Applying Method References in Practice

- Method references can be used to print a collection or array in various ways

```
String[] nameArray = {"Barbara", "James", "Mary", "John",  
                     "Robert", "Michael", "Linda", "james", "mary"};
```

Array of names represented as strings

Applying Method References in Practice

- Method references can be used to print a collection or array in various ways

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
                      "Robert", "Michael", "Linda", "james", "mary"};
```

- System.out.println() can be used to print out an array

```
System.out.println(Arrays.asList(nameArray));
```

prints

```
[Barbara, James, Mary, John, Linda, Michael, Linda, james, mary]
```

Applying Method References in Practice

- Method references can be used to print a collection or array in various ways

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
                      "Robert", "Michael", "Linda", "james", "mary"};
```

- System.out.println() can be used to print out an array

```
System.out.println(Arrays.asList(nameArray));
```

prints

Returns a fixed-size list backed by the specified array

```
[Barbara, James, Mary, John, Linda, Michael, Linda, james, mary]
```

Applying Method References in Practice

- Method references can be used to print a collection or array in various ways

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
    "Robert", "Michael", "Linda", "james", "mary"};
```

- System.out.println() can be used to print out an array
- Java 8's forEach() method can be used to print out values of an array

See www.javaworld.com/article/2461744/java-language/java-language-iterating-over-collections-in-java-8.html

Applying Method References in Practice

- Method references can be used to print a collection or array in various ways

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
    "Robert", "Michael", "Linda", "james", "mary"};
```

- System.out.println() can be used to print out an array
- Java 8's forEach() method can be used to print out values of an array, e.g.
- In conjunction with a stream & method reference

```
Stream.of(nameArray).forEach(System.out::print);
```

prints

Factory method that creates a stream from an array

BarbaraJamesMaryJohnLindaMichaelLindajamesmary

Applying Method References in Practice

- Method references can be used to print a collection or array in various ways

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
    "Robert", "Michael", "Linda", "james", "mary"};
```

- System.out.println() can be used to print out an array
- Java 8's forEach() method can be used to print out values of an array, e.g.
 - In conjunction with a stream & method reference

```
Stream.of(nameArray).forEach(System.out::print);
```

prints

BarbaraJamesMaryJohnLindaMichaelLindajamesmary

Performs method reference action on each stream element

Applying Method References in Practice

- Method references can be used to print a collection or array in various ways

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
    "Robert", "Michael", "Linda", "james", "mary"};
```

- System.out.println() can be used to print out an array
- Java 8's forEach() method can be used to print out values of an array, e.g.
 - In conjunction with a stream & method reference
 - In conjunction with a collection (e.g., List)

```
Arrays.asList(nameArray).forEach(System.out::print);
```

prints

Converts array into a List

```
BarbaraJamesMaryJohnLindaMichaelLindaJamesMary
```

See docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#asList

Applying Method References in Practice

- Method references can be used to print a collection or array in various ways

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
                      "Robert", "Michael", "Linda", "james", "mary"};
```

- System.out.println() can be used to print out an array
- Java 8's forEach() method can be used to print out values of an array, e.g.
 - In conjunction with a stream & method reference
 - In conjunction with a collection (e.g., List)

```
Arrays.asList(nameArray).forEach(System.out::print);
```

prints

Performs method reference action on each list element

BarbaraJamesMaryJohnLindaMichaelLinda james mary

Applying Method References in Practice

- Method references can be used to print a collection or array in various ways

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
    "Robert", "Michael", "Linda", "james", "mary"};
```

- System.out.println() can be used to print out an array
- Java 8's forEach() method can be used to print out values of an array, e.g.
 - In conjunction with a stream & method reference
 - In conjunction with a collection (e.g., List)
 - forEach() on a stream differs slight from
forEach() on a collection



Applying Method References in Practice

- Method references can be used to print a collection or array in various ways

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
    "Robert", "Michael", "Linda", "james", "mary"};
```

- System.out.println() can be used to print out an array
- Java 8's forEach() method can be used to print out values of an array, e.g.
 - In conjunction with a stream & method reference
 - In conjunction with a collection (e.g., List)
 - forEach() on a stream differs slightly from forEach() on a collection
 - e.g., forEach() ordering is undefined on a stream, whereas it's defined for a collection



Implementing Closures with Lambda Expressions

Implementing Closures with Lambda Expressions

- Lambda expressions can implement (simplified) variants of “closures”

```
class ClosureExample {  
    private int mRes;  
  
    Thread makeThreadClosure(String s, int n) {  
        return new Thread(() ->  
            System.out.println(s + (mRes += n)));  
    }  
  
    ClosureExample() throws InterruptedException {  
        Thread t = makeThreadClosure("result = ", 10);  
        t.start(); t.join();  
    }  
}
```

Implementing Closures with Lambda Expressions

- Lambda expressions can implement (simplified) variants of “closures”
 - A closure is a method that has an environment w/at least 1 bound variable

```
class ClosureExample {  
    private int mRes;  
  
    Thread makeThreadClosure(String s, int n) {  
        return new Thread(() ->  
            System.out.println(s + (mRes += n)));  
    }  
  
    ClosureExample() throw InterruptedException {  
        Thread t = makeThreadClosure("result = ", 10);  
        t.start(); t.join();  
    }  
}
```

See [en.wikipedia.org/wiki/Closure_\(computer_programming\)](https://en.wikipedia.org/wiki/Closure_(computer_programming))

Implementing Closures with Lambda Expressions

- Lambda expressions can implement (simplified) variants of “closures”
 - A closure is a method that has an environment w/at least 1 bound variable

```
class ClosureExample {  
    private int mRes;
```

*This private field & method
params are bound variables*

```
    Thread makeThreadClosure(String s, int n) {  
        return new Thread(() ->  
            System.out.println(s + (mRes += n)));  
    }  
}
```

```
ClosureExample() throws InterruptedException {  
    Thread t = makeThreadClosure("result = ", 10);  
    t.start(); t.join();  
}  
}
```

A bound variable is name that has a *value*, such as a number

Implementing Closures with Lambda Expressions

- Lambda expressions can implement (simplified) variants of “closures”
 - A closure is a method that has an environment w/at least 1 bound variable

```
class ClosureExample {  
    private int mRes;
```

This lambda implements a closure that captures a private field & method params

```
    Thread makeThreadClosure(String s, int n) {  
        return new Thread(() ->  
            System.out.println(s + (mRes += n)));  
    }  
}
```

```
ClosureExample() throws InterruptedException {  
    Thread t = makeThreadClosure("result = ", 10);  
    t.start(); t.join();  
}  
}
```

See bruceeckel.github.io/2015/10/17/are-java-8-lambdas-closures

Implementing Closures with Lambda Expressions

- Lambda expressions can implement (simplified) variants of “closures”
 - A closure is a method that has an environment w/at least 1 bound variable

```
class ClosureExample {    Values of private fields can be updated in a lambda,  
but not params or local vars (which are read-only)
```

```
    private int mRes;  
  
    Thread makeThreadClosure(String s, int n) {  
        return new Thread(() ->  
            System.out.println(s + (mRes += n)));  
    }  
  
}
```

```
ClosureExample() throws InterruptedException {  
    Thread t = makeThreadClosure("result = ", 10);  
    t.start(); t.join();  
}  
}
```

Implementing Closures with Lambda Expressions

- Lambda expressions can implement (simplified) variants of “closures”
 - A closure is a method that has an environment w/at least 1 bound variable

```
class ClosureExample {  
    private int mRes;  
  
    Thread makeThreadClosure(String s, int n) {  
        return new Thread(() ->  
            System.out.println(s + (mRes += n)));  
    }  
  
    ClosureExample() throws InterruptedException {  
        Thread t = makeThreadClosure("result = ", 10);  
        t.start(); t.join();  
    }  
}
```

Constructor creates a closure & runs it in a background thread

End of Overview of Java 8 Lambda Expressions & Method References

Overview of Java 8 Functional Interfaces

Douglas C. Schmidt

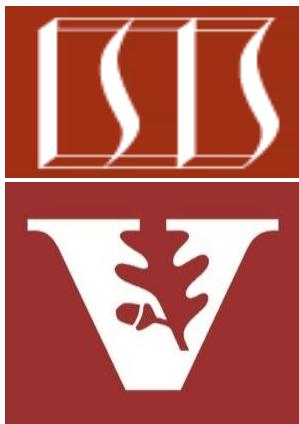
d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

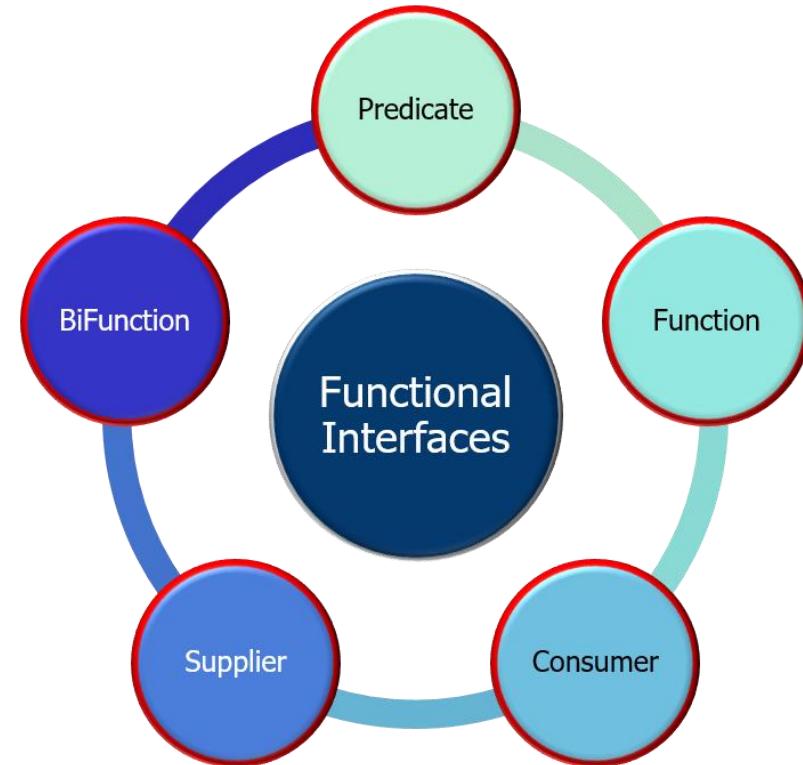
Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA



Learning Objectives in this Lesson

- Recognize foundational functional programming features in Java 8, e.g.,
 - Lambda expressions
 - Method & constructor references
 - Key functional interfaces



Learning Objectives in this Lesson

- Recognize foundational functional programming features in Java 8, e.g.,
 - Lambda expressions
 - Method & constructor references
 - Key functional interfaces



These features form the basis for Java 8's streams & concurrency/parallelism frameworks

Learning Objectives in this Lesson

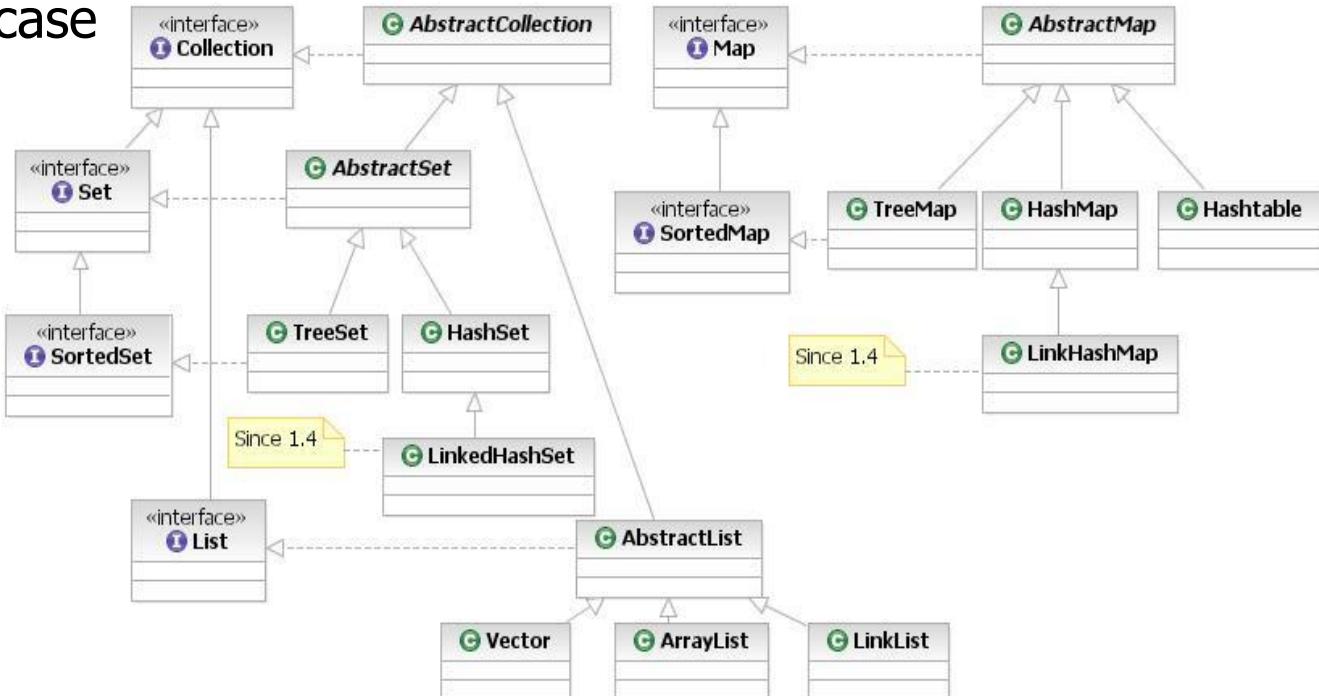
- Recognize foundational functional programming features in Java 8
- Understand how to apply these Java 8 features in concise example programs



See github.com/douglasraigschmidt/LiveLessons/tree/master/Java8

Learning Objectives in this Lesson

- Recognize foundational functional programming features in Java 8
- Understand how to apply these Java 8 features in concise example programs
 - The examples showcase the Java collections framework

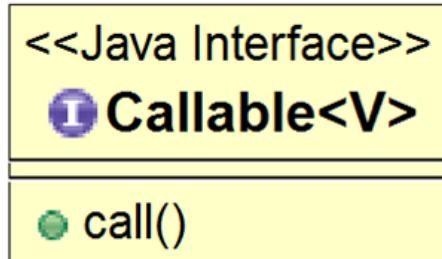
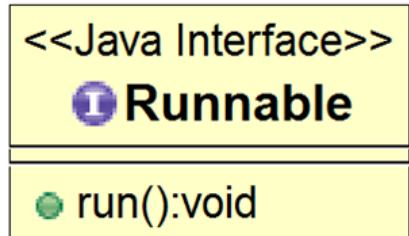


See docs.oracle.com/javase/8/docs/technotes/guides/collections

Overview of Common Functional Interfaces

Overview of Common Functional Interfaces

- A *functional interface* contains only one abstract method



See www.oreilly.com/learning/java-8-functional-interfaces

Overview of Common Functional Interfaces

- A functional interface is the type used for a parameter when a lambda expression or method reference is passed as an argument

```
<T> void runTest(Function<T, T> fact, T n) {  
    long startTime = System.nanoTime();  
    T result = fact.apply(n));  
    long stopTime = (System.nanoTime() - startTime) / 1_000_000;  
    ...  
}  
runTest(ParallelStreamFactorial::factorial, n);  
runTest(SequentialStreamFactorial::factorial, n);  
...
```

Overview of Common Functional Interfaces

- A functional interface is the type used for a parameter when a lambda expression or method reference is passed as an argument

```
<T> void runTest(Function<T, T> fact, T n) {  
    long startTime = System.nanoTime();  
    T result = fact.apply(n));  
    long stopTime = (System.nanoTime() - startTime) / 1_000_000;  
    ...  
}  
  
runTest(ParallelStreamFactorial::factorial, n);  
runTest(SequentialStreamFactorial::factorial, n);  
...
```

*Records & prints time taken
to compute 'n' factorial*

Overview of Common Functional Interfaces

- A functional interface is the type used for a parameter when a lambda expression or method reference is passed as an argument

```
<T> void runTest(Function<T, T> fact, T n)
    long startTime = System.nanoTime();
    T result = fact.apply(n));
    long stopTime = (System.nanoTime() - startTime) / 1_000_000;
    ...
}

runTest(ParallelStreamFactorial::factorial, n);
runTest(SequentialStreamFactorial::factorial, n);
...
```

'fact' parameterizes the factorial implementation

Overview of Common Functional Interfaces

- A functional interface is the type used for a parameter when a lambda expression or method reference is passed as an argument

```
<T> void runTest(Function<T, T> fact, T n) {  
    long startTime = System.nanoTime();  
    T result = fact.apply(n));  
    long stopTime = (System.nanoTime() - startTime) / 1_000_000;  
    ...  
}  
  
runTest(ParallelStreamFactorial::factorial, n);  
runTest(SequentialStreamFactorial::factorial, n);  
...
```

*Different factorial implementations can
be passed as params to runTest()*

Overview of Common Functional Interfaces

- A functional interface is the type used for a parameter when a lambda expression or method reference is passed as an argument

```
<T> void runTest(Function<T, T> fact, T n) {  
    long startTime = System.nanoTime();  
    T result = fact.apply(n));  
    long stopTime = (System.nanoTime() - startTime) / 1_000_000;  
    ...  
}  
runTest(ParallelStreamFactorial::factorial, n);
```

```
static BigInteger factorial(BigInteger n) {  
    return LongStream.rangeClosed(1, n)  
        .parallel()  
        .mapToObj(BigInteger::valueOf)  
        .reduce(BigInteger.ONE, BigInteger::multiply);  
}
```

Overview of Common Functional Interfaces

- Java 8 defines many types of functional interfaces

Package `java.util.function`

Functional interfaces provide target types for lambda expressions and method references.

See: [Description](#)

Interface Summary

Interface	Description
<code>BiConsumer<T,U></code>	Represents an operation that accepts two input arguments and returns no result.
<code>BiFunction<T,U,R></code>	Represents a function that accepts two arguments and produces a result.
<code>BinaryOperator<T></code>	Represents an operation upon two operands of the same type, producing a result of the same type as the operands.
<code>BiPredicate<T,U></code>	Represents a predicate (boolean-valued function) of two arguments.
<code>BooleanSupplier</code>	Represents a supplier of boolean-valued results.
<code>Consumer<T></code>	Represents an operation that accepts a single input argument and returns no result.
<code>DoubleBinaryOperator</code>	Represents an operation upon two double-valued operands and producing a double-valued result.
<code>DoubleConsumer</code>	Represents an operation that accepts a single double-valued argument and returns no result.
<code>DoubleFunction<R></code>	Represents a function that accepts a double-valued argument and produces a result.
<code>DoublePredicate</code>	Represents a predicate (boolean-valued function) of one double-valued argument.
<code>DoubleSupplier</code>	Represents a supplier of double-valued results.
<code>DoubleToIntFunction</code>	Represents a function that accepts a double-valued argument and produces an int-valued result.
<code>DoubleToLongFunction</code>	Represents a function that accepts a double-valued argument and produces a long-valued result.
<code>DoubleUnaryOperator</code>	Represents an operation on a single double-valued operand that produces a double-valued result.
<code>Function<T,R></code>	Represents a function that accepts one argument and produces a result.

Overview of Common Functional Interfaces

- Java 8 defines many types of functional interfaces
 - This list is large due to the need to support reference types & primitive types..

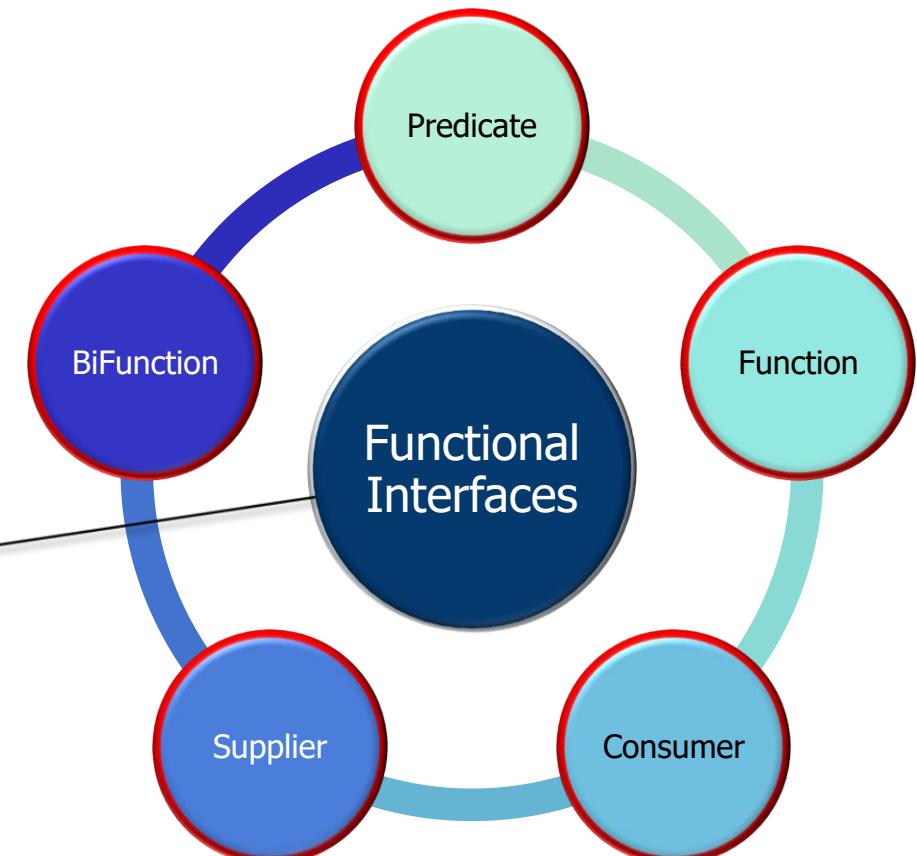
Interface Summary	
Interface	Description
<code>IntConsumer</code>	Represents an operation that accepts a single int-valued argument and returns no result.
<code>IntFunction<R></code>	Represents a function that accepts an int-valued argument and produces a result.
<code>IntPredicate</code>	Represents a predicate (boolean-valued function) of one int-valued argument.
<code>IntSupplier</code>	Represents a supplier of int-valued results.
<code>IntToDoubleFunction</code>	Represents a function that accepts an int-valued argument and produces a double-valued result.
<code>IntToLongFunction</code>	Represents a function that accepts an int-valued argument and produces a long-valued result.
<code>IntUnaryOperator</code>	Represents an operation on a single int-valued operand that produces an int-valued result.
<code>LongBinaryOperator</code>	Represents an operation upon two long-valued operands and producing a long-valued result.
<code>LongConsumer</code>	Represents an operation that accepts a single long-valued argument and returns no result.
<code>LongFunction<R></code>	Represents a function that accepts a long-valued argument and produces a result.
<code>LongPredicate</code>	Represents a predicate (boolean-valued function) of one long-valued argument.
<code>LongSupplier</code>	Represents a supplier of long-valued results.
<code>LongToDoubleFunction</code>	Represents a function that accepts a long-valued argument and produces a double-valued result.
<code>LongToIntFunction</code>	Represents a function that accepts a long-valued argument and produces an int-valued result.
<code>LongUnaryOperator</code>	Represents an operation on a single long-valued operand that produces a long-valued result.
<code>ObjDoubleConsumer<T></code>	Represents an operation that accepts an object-valued and a double-valued argument, and returns no result.
<code>ObjIntConsumer<T></code>	Represents an operation that accepts an object-valued and a int-valued argument, and returns no result.

See dzone.com/articles/whats-wrong-java-8-part-ii

Overview of Common Functional Interfaces

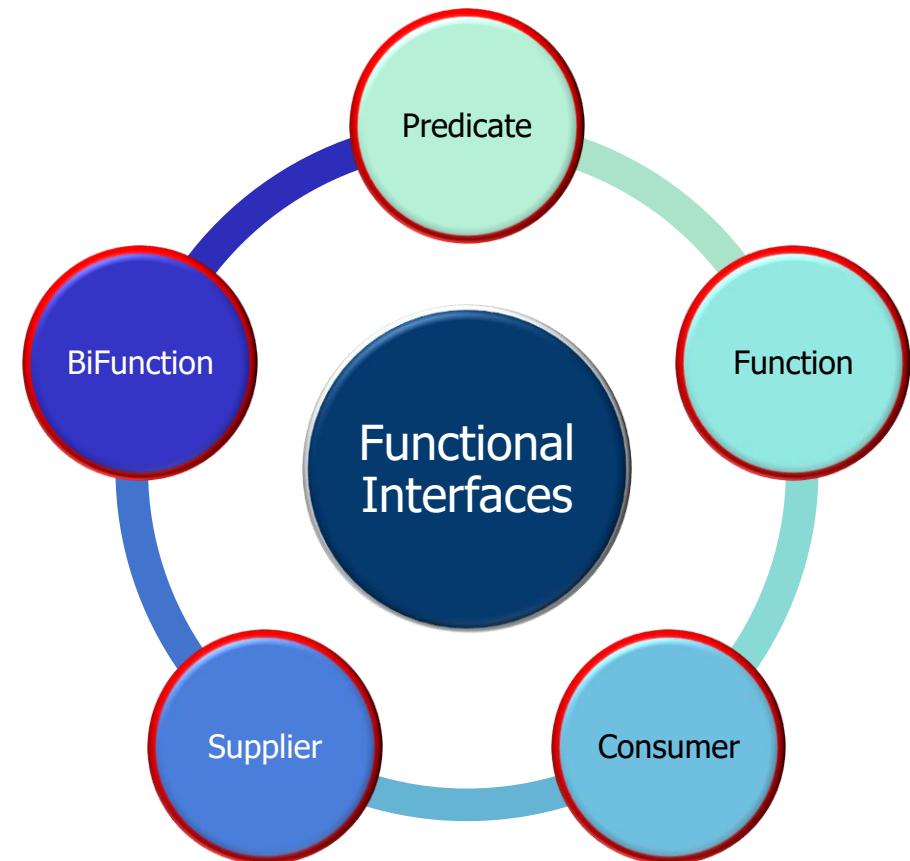
- Java 8 defines many types of functional interfaces
 - This list is large due to the need to support reference types & primitive types..

We focus on the most common types of functional interfaces



Overview of Common Functional Interfaces

- Java 8 defines many types of functional interfaces
 - This list is large due to the need to support reference types & primitive types..



All usages of functional interfaces in the upcoming examples are “stateless”!

Overview of Functional Interfaces: Predicate

Overview of Common Functional Interfaces: Predicate

- A *Predicate* performs a test that returns true or false, e.g.,
 - `public interface Predicate<T> { boolean test(T t); }`

Overview of Common Functional Interfaces: Predicate

- A *Predicate* performs a test that returns true or false, e.g.,
 - `public interface Predicate<T> { boolean test(T t); }`

Overview of Common Functional Interfaces: Predicate

- A *Predicate* performs a test that returns true or false, e.g.,

- ```
public interface Predicate<T> { boolean test(T t); }
```

```
Map<String, Integer> iqMap =
 new ConcurrentHashMap<String, Integer>() { {
 put("Larry", 100); put("Curly", 90); put("Moe", 110);
 }
};
```

*Create a map of "stooges" & their IQs!*

```
System.out.println(iqMap);
```

```
iqMap.entrySet().removeIf(entry -> entry.getValue() <= 100);
```

```
System.out.println(iqMap);
```



# Overview of Common Functional Interfaces: Predicate

- A *Predicate* performs a test that returns true or false, e.g.,

```
public interface Predicate<T> { boolean test(T t); }

Map<String, Integer> iqMap =
 new ConcurrentHashMap<String, Integer>() { {
 put("Larry", 100); put("Curly", 90); put("Moe", 110);
 }
};

System.out.println(iqMap);

iqMap.entrySet().removeIf(entry -> entry.getValue() <= 100);

System.out.println(iqMap);
```

*This predicate lambda deletes entries with iq <= 100*

# Overview of Common Functional Interfaces: Predicate

- A *Predicate* performs a test that returns true or false, e.g.,

```
• public interface Predicate<T> { boolean test(T t); }
```

```
Map<String, Integer> iqMap =
 new ConcurrentHashMap<String, Integer>() { {
 put("Larry", 100); put("Curly", 90); put("Moe", 110);
 }
};

System.out.println(iqMap);
```

*entry* is short for (*EntrySet<String, Integer> entry*), which leverages the Java 8 compiler's type inference capabilities

```
iqMap.entrySet().removeIf(entry -> entry.getValue() <= 100);

System.out.println(iqMap);
```

# Overview of Common Functional Interfaces: Predicate

- A *Predicate* performs a test that returns true or false, e.g.,

- ```
public interface Predicate<T> { boolean test(T t); }
```

```
interface Collection<E> {  
    ...  
    default boolean removeIf(Predicate<? super E> filter) {  
        ...  
        final Iterator<E> each = iterator();  
        while (each.hasNext()) {  
            if (filter.test(each.next())) {  
                each.remove();  
            }  
        }  
    }  
}
```

Here's how the `removeIf()` method uses the predicate passed to it

Overview of Common Functional Interfaces: Predicate

- A *Predicate* performs a test that returns true or false, e.g.,

```
public interface Predicate<T> { boolean test(T t); }

interface Collection<E> {
    ...
    default boolean removeIf(Predicate<? super E> filter) {
        ...
        final Iterator<E> each = iterator();
        while (each.hasNext()) {
            if (filter.test(each.next())))
                each.remove();
        ...
    }
}
```

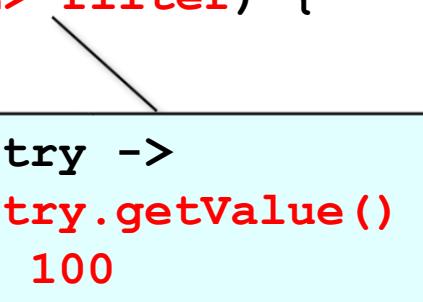
'super' is a *lower bounded* wildcard restricts the unknown type to be a specific type or a *super type* of that type

Overview of Common Functional Interfaces: Predicate

- A *Predicate* performs a test that returns true or false, e.g.,

```
public interface Predicate<T> { boolean test(T t); }

interface Collection<E> {
    ...
    default boolean removeIf(Predicate<? super E> filter) {
        ...
        final Iterator<E> each = iterator();
        while (each.hasNext()) {
            if (filter.test(each.next())) {
                each.remove();
            }
        }
    }
}
```



entry ->
entry.getValue()
<= 100

This predicate parameter is bound to the lambda expression passed to it

Overview of Common Functional Interfaces: Predicate

- A *Predicate* performs a test that returns true or false, e.g.,

- ```
public interface Predicate<T> { boolean test(T t); }
```

```
interface Collection<E> {
```

```
...
```

```
default boolean removeIf(Predicate<? super E> filter) {
```

```
...
```

```
final Iterator<E> each = iterator();
```

```
while (each.hasNext()) {
```

```
 if (filter.test(each.next())) {
```

```
 each.remove();
```

```
...
```

entry ->  
entry.getValue()  
<= 100

if (each.next().getValue() <= 100)

The 'entry' in the lambda predicate is replaced by the parameter to test()

---

# Overview of Functional Interfaces: Function

# Overview of Common Functional Interfaces: Function

---

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,
  - `public interface Function<T, R> { R apply(T t); }`

# Overview of Common Functional Interfaces: Function

---

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,
  - `public interface Function<T, R> { R apply(T t); }`

# Overview of Common Functional Interfaces: Function

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,

- ```
public interface Function<T, R> { R apply(T t); }
```

```
Map<Integer, Integer> primeCache =  
    new ConcurrentHashMap<>();
```

*This map caches the results
of prime # computations*

```
...
```

```
Long smallestFactor = primeCache.computeIfAbsent  
(primeCandidate, (key) -> primeChecker(key));
```

```
...
```

```
Integer primeChecker(Integer primeCandidate) {  
    ... // Returns 0 if a number is prime or the smallest  
        // factor if it's not prime  
}
```

Overview of Common Functional Interfaces: Function

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,

- ```
public interface Function<T, R> { R apply(T t); }
```

```
Map<Integer, Integer> primeCache =
 new ConcurrentHashMap<>();
```

*If key isn't already associated with a value, compute the value using the given mapping function & enter it into the map*

```
...
```

```
Long smallestFactor = primeCache.computeIfAbsent
(primeCandidate, (key) -> primeChecker(key));
```

```
...
```

```
Integer primeChecker(Integer primeCandidate) {
 ... // Returns 0 if a number is prime or the smallest
 // factor if it's not prime
}
```

# Overview of Common Functional Interfaces: Function

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,

- ```
public interface Function<T, R> { R apply(T t); }
```

```
Map<Integer, Integer> primeCache =  
    new ConcurrentHashMap<>();
```

*This method provides atomic
"check then act" semantics*

```
...
```

```
Long smallestFactor = primeCache.computeIfAbsent  
(primeCandidate, (key) -> primeChecker(key));
```

```
...
```

```
Integer primeChecker(Integer primeCandidate) {  
    ... // Returns 0 if a number is prime or the smallest  
        // factor if it's not prime  
}
```

Overview of Common Functional Interfaces: Function

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,

- ```
public interface Function<T, R> { R apply(T t); }
```

```
Map<Integer, Integer> primeCache =
 new ConcurrentHashMap<>();
```

*A lambda expression that calls a function*

```
...
Long smallestFactor = primeCache.computeIfAbsent
(primeCandidate, (key) -> primeChecker(key));
...
```

```
Integer primeChecker(Integer primeCandidate) {
 ... // Returns 0 if a number is prime or the smallest
 // factor if it's not prime
}
```

# Overview of Common Functional Interfaces: Function

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,

- ```
public interface Function<T, R> { R apply(T t); }
```

```
Map<Integer, Integer> primeCache =  
    new ConcurrentHashMap<>();
```

Could also be passed as a method reference

```
...  
Long smallestFactor = primeCache.computeIfAbsent  
(primeCandidate, this::primeChecker);  
...  
  
Integer primeChecker(Integer primeCandidate) {  
    ... // Returns 0 if a number is prime or the smallest  
    // factor if it's not prime  
}
```

Overview of Common Functional Interfaces: Function

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,

```
• public interface Function<T, R> { R apply(T t); }

class ConcurrentHashMap<K,V> ...
    public V computeIfAbsent(K key,
        Function<? super K, ? extends V> mappingFunction) {
    ...
    if ((f = tabAt(tab, i = (n - 1) & h)) == null)
    ...
    if ((val = mappingFunction.apply(key)) != null)
        node = new Node<K,V>(h, key, val, null);
    ...
}
```

Here's how the `computeIfAbsent()` method uses the function passed to it

Overview of Common Functional Interfaces: Function

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,

```
• public interface Function<T, R> { R apply(T t); }

class ConcurrentHashMap<K,V> ...
public V computeIfAbsent(K key,
    Function<? super K, ? extends V> mappingFunction) {
```

'super' is a *lower bounded* wildcard restricts the unknown type to be a specific type or a *super type* of that type

```
...
if ((f = tabAt(tab, i = (n - 1) & h)) == null)
    ...
if ((val = mappingFunction.apply(key)) != null)
    node = new Node<K,V>(h, key, val, null);
    ...
}
```

Overview of Common Functional Interfaces: Function

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,

```
• public interface Function<T, R> { R apply(T t); }

class ConcurrentHashMap<K,V> ...
public V computeIfAbsent(K key,
    Function<? super K, ? extends V> mappingFunction) {
    ...
    if ((f = tabAt(tab, i = (n - 1) & h)) == null)
        ...
    if ((val = mappingFunction.apply(key)) != null)
        node = new Node<K,V>(h, key, val, null);
    ...
}
```

'extends' is an *upper bounded* wildcard that restricts the unknown type to be a specific type or a subtype of that type

Overview of Common Functional Interfaces: Function

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,

```
• public interface Function<T, R> { R apply(T t); }

class ConcurrentHashMap<K,V> ...
public V computeIfAbsent(K key,
    Function<? super K, ? extends V> mappingFunction) {
```

'super' & 'extends' play different roles in Java generics

```
...
if ((f = tabAt(tab, i = (n - 1) & h)) == null)
    ...
if ((val = mappingFunction.apply(key)) != null)
    node = new Node<K,V>(h, key, val, null);
    ...
```

See [en.wikipedia.org/wiki/Generics_in_Java#Type wildcards](https://en.wikipedia.org/wiki/Generics_in_Java#Type_wildcards)

Overview of Common Functional Interfaces: Function

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,

```
• public interface Function<T, R> { R apply(T t); }

class ConcurrentHashMap<K,V> ...
    public V computeIfAbsent(K key,
        Function<? super K, ? extends V> mappingFunction) {
    ...
    if ((f = tabAt(tab, i = (n - 1) & h)) == null)
    ...
    if ((val = mappingFunction.apply(key)) != null)
        node = new Node<K,V>(h, key, val, null);
    ...
}
```

this::primeChecker

The function parameter is bound to this::primeChecker method reference

Overview of Common Functional Interfaces: Function

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,

```
• public interface Function<T, R> { R apply(T t); }

class ConcurrentHashMap<K,V> ...
public V computeIfAbsent(K key,
    Function<? super K, ? extends V> mappingFunction) {
```

```
    if ((val = primeChecker(key)) != null)
...
if ((f = tabAt(tab, i = (n - 1) & h)) == null)
...
if ((val = mappingFunction.apply(key)) != null)
    node = new Node<K,V>(h, key, val, null);
...
```

The `apply()` method is replaced with the `primeChecker()` lambda function

Overview of Common Functional Interfaces: Function

- This example also shows a *Function*, e.g.,

- ```
public interface Function<T, R> { R apply(T t); }
```

```
List<Thread> threads =
 new ArrayList<>(Arrays.asList(new Thread("Larry"),
 new Thread("Curly"),
 new Thread("Moe")));
```

*Create a list of threads named  
after the three stooges*

```
threads.forEach(System.out::println);
threads.sort(Comparator.comparing(Thread::getName));
threads.forEach(System.out::println);
```

# Overview of Common Functional Interfaces: Function

- This example also shows a *Function*, e.g.,

```
• public interface Function<T, R> { R apply(T t); }

List<Thread> threads =
 new ArrayList<>(Arrays.asList(new Thread("Larry"),
 new Thread("Curly"),
 new Thread("Moe")));
```

*A method reference to a Function used to sort threads by name*

```
threads.forEach(System.out::println);
threads.sort(Comparator.comparing(Thread::getName));
threads.forEach(System.out::println);
```

# Overview of Common Functional Interfaces: Function

- This example also shows a *Function*, e.g.,

```
• public interface Function<T, R> { R apply(T t); }

List<Thread> threads =
 new ArrayList<>(Arrays.asList(new Thread("Larry"),
 new Thread("Curly"),
 new Thread("Moe")));
```

*This method uses the Thread::getName method reference to impose a total ordering on some collection of objects*

```
threads.forEach(System.out::println);
threads.sort(Comparator.comparing(Thread::getName));
threads.forEach(System.out::println);
```

# Overview of Common Functional Interfaces: Function

- This example also shows a *Function*, e.g.,

```
• public interface Function<T, R> { R apply(T t); }

interface Comparator { Imposes a total ordering on a collection of objects
 ...
 static <T, U extends Comparable<? super U>> Comparator<T>
 comparing(Function<? super T, ? extends U> keyEx) {
 return ((c1, c2) ->
 keyEx.apply(c1)
 .compareTo(keyEx.apply(c2)); }
```

# Overview of Common Functional Interfaces: Function

- This example also shows a *Function*, e.g.,

```
public interface Function<T, R> { R apply(T t); }

interface Comparator {

 ...

 static <T, U extends Comparable<? super U>> Comparator<T>
 comparing(Function<? super T, ? extends U> keyEx) {
 return ((c1, c2) ->
 keyEx.apply(c1)
 .compareTo(keyEx.apply(c2)); }
```

The *comparing()* method is passed  
a *Function* parameter called *keyEx*

# Overview of Common Functional Interfaces: Function

- This example also shows a *Function*, e.g.,

```
• public interface Function<T, R> { R apply(T t); }

interface Comparator {
 ...
 static <T, U extends Comparable<? super U>> Comparator<T>
 comparing(Function<? super T, ? extends U> keyEx) {
 return ((c1, c2) ->
 keyEx.apply(c1)
 .compareTo(keyEx.apply(c2)); }
```

Thread::getName

The Thread::getName method reference is bound to the keyEx parameter

# Overview of Common Functional Interfaces: Function

- This example also shows a *Function*, e.g.,

```
• public interface Function<T, R> { R apply(T t); }

interface Comparator {
 ...
 static <T, U extends Comparable<? super U>> Comparator<T>
 comparing(Function<? super T, ? extends U> keyEx) {
 return ((c1, c2) ->
 keyEx.apply(c1)
 .compareTo(keyEx.apply(c2)); }
```

*c1 & c2 are thread objects  
being compared by sort()*

# Overview of Common Functional Interfaces: Function

- This example also shows a *Function*, e.g.,

```
public interface Function<T, R> { R apply(T t); }

interface Comparator {

 ...

 static <T, U extends Comparable<? super U>> Comparator<T>
 comparing(Function<? super T, ? extends U> keyEx) {
 return ((c1, c2) ->
 keyEx.apply(c1)
 .compareTo(keyEx.apply(c2)); }
```

The *apply()* method of the *keyEx* function is used to compare strings

# Overview of Common Functional Interfaces: Function

- This example also shows a *Function*, e.g.,

```
• public interface Function<T, R> { R apply(T t); }

interface Comparator {
 ...
 static <T, U extends Comparable<? super U>> Comparator<T>
 comparing(Function<? super T, ? extends U> keyEx) {
 return ((c1, c2) ->
 keyEx.apply(c1)
 .compareTo(keyEx.apply(c2)); }
```



```
c1.getName().compareTo(c2.getName())
```

The `Thread::getName` method reference is called to compare two thread names

---

# Overview of Functional Interfaces: BiFunction

# Overview of Common Functional Interfaces: BiFunction

- A *BiFunction* applies a computation on 2 parameters & returns a result, e.g.,
  - `public interface BiFunction<T, U, R> { R apply(T t, U u); }`

# Overview of Common Functional Interfaces: BiFunction

- A *BiFunction* applies a computation on 2 parameters & returns a result, e.g.,
  - `public interface BiFunction<T, U, R> { R apply(T t, U u); }`

# Overview of Common Functional Interfaces: BiFunction

- A *BiFunction* applies a computation on 2 parameters & returns a result, e.g.,

```
• public interface BiFunction<T, U, R> { R apply(T t, U u); }
```

```
Map<String, Integer> iqMap =
 new ConcurrentHashMap<String, Integer>() {
 { put("Larry", 100); put("Curly", 90); put("Moe", 110); }
 };
```

Create a map of "stooges" & their IQs!

```
for (Map.Entry<String, Integer> entry : iqMap.entrySet())
 entry.setValue(entry.getValue() - 50);
```

VS.

```
iqMap.replaceAll((k, v) -> v - 50);
```

See [github.com/douglasraigschmidt/LiveLessons/tree/master/Java8/ex4](https://github.com/douglasraigschmidt/LiveLessons/tree/master/Java8/ex4)

# Overview of Common Functional Interfaces: BiFunction

- A *BiFunction* applies a computation on 2 parameters & returns a result, e.g.,

```
• public interface BiFunction<T, U, R> { R apply(T t, U u); }
```

```
Map<String, Integer> iqMap =
 new ConcurrentHashMap<String, Integer>() {
 { put("Larry", 100); put("Curly", 90); put("Moe", 110); }
 };
```

```
for (Map.Entry<String, Integer> entry : iqMap.entrySet())
 entry.setValue(entry.getValue() - 50);
```

VS.

*Conventional way of subtracting 50 IQ points from each person in map*

```
iqMap.replaceAll((k, v) -> v - 50);
```

# Overview of Common Functional Interfaces: BiFunction

- A *BiFunction* applies a computation on 2 parameters & returns a result, e.g.,

```
• public interface BiFunction<T, U, R> { R apply(T t, U u); }
```

```
Map<String, Integer> iqMap =
 new ConcurrentHashMap<String, Integer>() {
 { put("Larry", 100); put("Curly", 90); put("Moe", 110); }
 };
```

```
for (Map.Entry<String, Integer> entry : iqMap.entrySet())
 entry.setValue(entry.getValue() - 50);
```

vs.

*BiFunction lambda subtracts 50 IQ points from each person in map*

```
iqMap.replaceAll((k, v) -> v - 50);
```

# Overview of Common Functional Interfaces: BiFunction

- A *BiFunction* applies a computation on 2 parameters & returns a result, e.g.,

```
• public interface BiFunction<T, U, R> { R apply(T t, U u); }
```

```
Map<String, Integer> iqMap =
 new ConcurrentHashMap<String, Integer>() {
 { put("Larry", 100); put("Curly", 90); put("Moe", 110); }
 };
```

```
for (Map.Entry<String, Integer> entry : iqMap.entrySet())
 entry.setValue(entry.getValue() - 50);
```

vs.

*Unlike Entry operations, replaceAll() operates in a thread-safe manner!*

```
iqMap.replaceAll((k, v) -> v - 50);
```

# Overview of Common Functional Interfaces: BiFunction

- A *BiFunction* applies a computation on 2 parameters & returns a result, e.g.,

```
• public interface BiFunction<T, U, R> { R apply(T t, U u); }

class ConcurrentHashMap<K,V> {

 ...
 public void replaceAll
 (BiFunction<? super K, ? super V, ? extends V> function) {
 ...
 for (Node<K,V> p; (p = it.advance()) != null;) {
 V oldValue = p.val;
 for (K key = p.key;;) {
 V newValue = function.apply(key, oldValue);
 ...
 replaceNode(key, newValue, oldValue)
 ...
 }
 }
 }
}
```

The replaceAll() method uses the bifunction passed to it in a thread-safe way

# Overview of Common Functional Interfaces: BiFunction

- A *BiFunction* applies a computation on 2 parameters & returns a result, e.g.,

```
• public interface BiFunction<T, U, R> { R apply(T t, U u); }

class ConcurrentHashMap<K,V> {
 ...
 public void replaceAll
 (BiFunction<? super K, ? super V, ? extends V> function) {
 ...
 for (Node<K,V> p; (p = it.advance()) != null;) {
 V oldValue = p.val;
 for (K key = p.key;;) {
 V newValue = function.apply(key, oldValue);
 ...
 replaceNode(key, newValue, oldValue)
 ...
 }
 }
 }
}
```

$(k, v) \rightarrow v - 50$



The bifunction parameter is bound to the lambda expression  $v - 50$

# Overview of Common Functional Interfaces: BiFunction

- A *BiFunction* applies a computation on 2 parameters & returns a result, e.g.,

```
• public interface BiFunction<T, U, R> { R apply(T t, U u); }
```

```
class ConcurrentHashMap<K,V> {
 ...
 public void replaceAll
 (BiFunction<? super K, ? super V, ? extends V> function) {
 ...
 for (Node<K,V> p; (p = it.advance()) != null;) {
 V oldValue = p.val;
 for (K key = p.key;;) {
 V newValue = function.apply(key, oldValue);
 ...
 replaceNode(key, newValue, oldValue)
 ...
 }
 }
}
```

(*k*, *v*) -> *v* - 50

V newValue =  
oldValue - 50

The apply() method is replaced by the *v* - 50 bifunction lambda

---

# Overview of Functional Interfaces: Supplier

# Overview of Common Functional Interfaces: Supplier

---

- A *Supplier* returns a value & takes no parameters, e.g.,
  - `public interface Supplier<T> { T get(); }`

# Overview of Common Functional Interfaces: Supplier

---

- A *Supplier* returns a value & takes no parameters, e.g.,
  - `public interface Supplier<T> { T get(); }`

# Overview of Common Functional Interfaces: Supplier

- A *Supplier* returns a value & takes no parameters, e.g.,

- ```
public interface Supplier<T> { T get(); }
```

```
String f1("62675744/15668936"); String f2("609136/913704");
```

```
CompletableFuture<BigFraction> future = CompletableFuture  
    .supplyAsync(() -> {
```

```
        BigFraction bf1 =  
            new BigFraction(f1);
```

```
        BigFraction bf2 =  
            new BigFraction(f2);
```

```
        return bf1.multiply(bf2);});
```



Define a supplier lambda to multiply two BigFractions

```
System.out.println(future.join().toMixedString());
```

Overview of Common Functional Interfaces: Supplier

- A *Supplier* returns a value & takes no parameters, e.g.,

- ```
public interface Supplier<T> { T get(); }
```

```
String f1("62675744/15668936"); String f2("609136/913704");
```

```
CompletableFuture<BigFraction> future = CompletableFuture
 .supplyAsync(() -> {
 BigFraction bf1 =
 new BigFraction(f1);
 BigFraction bf2 =
 new BigFraction(f2);
 return bf1.multiply(bf2); });

```

*Although get() takes no parameters (effectively) final values can be passed to a supplier lambda*

```
System.out.println(future.join().toMixedString());
```

# Overview of Common Functional Interfaces: Supplier

- A *Supplier* returns a value & takes no parameters, e.g.,

- ```
public interface Supplier<T> { T get(); }
```

```
String f1("62675744/15668936"); String f2("609136/913704");
```

```
CompletableFuture<BigFraction> future = CompletableFuture  
    .supplyAsync(() -> {  
        BigFraction bf1 =  
            new BigFraction(f1);  
        BigFraction bf2 =  
            new BigFraction(f2);  
  
        return bf1.multiply(bf2);});
```

supplyAsync() runs the supplier lambda in a thread residing in the common fork-join pool

```
System.out.println(future.join().toMixedString());
```

Overview of Common Functional Interfaces: Supplier

- A *Supplier* returns a value & takes no parameters, e.g.,

```
• public interface Supplier<T> { T get(); }

<U> CompletableFuture<U> supplyAsync(Supplier<U> supplier) {
    ...
    CompletableFuture<U> f =
        new CompletableFuture<U>();
    execAsync(ForkJoinPool.commonPool(),
              new AsyncSupply<U>(supplier, f));
    return f;
}
```

Here's how the supplyAsync() method uses the supplier passed to it

Overview of Common Functional Interfaces: Supplier

- A *Supplier* returns a value & takes no parameters, e.g.,

```
• public interface Supplier<T> { T get(); }

<U> CompletableFuture<U> supplyAsync(Supplier<U> supplier) {
    ...
    CompletableFuture<U> f =
        new CompletableFuture<U>();
    execAsync(ForkJoinPool.commonPool(),
              new AsyncSupply<U>(supplier, f));
}

return f;
}
```

...

```
() -> { ... return
          bf1.multiply(bf2);
      }
```

The supplier parameter is bound to the lambda passed to supplyAsync()

Overview of Common Functional Interfaces: Supplier

- A *Supplier* returns a value & takes no parameters, e.g.,

```
• public interface Supplier<T> { T get(); }

<U> CompletableFuture<U> supplyAsync(Supplier<U> supplier) {
    ...
    CompletableFuture<U> f =
        new CompletableFuture<U>();
    execAsync(ForkJoinPool.commonPool(),
              new AsyncSupply<U>(supplier, f));
    return f;
}
```

The supplier is enqueued for asynchronous execution in the common fork-join pool

Overview of Common Functional Interfaces: Supplier

- A *Supplier* returns a value & takes no parameters, e.g.,

- ```
public interface Supplier<T> { T get(); }
```

...

```
static final class AsyncSupply<U> extends Async {
 final Supplier<U> fn;
```

```
AsyncSupply(Supplier<U> fn, ...) { this.fn = fn; ... }
```

```
public final boolean exec() {
```

...

```
 U u = fn.get();
```

...

```
}
```

```
}
```

```
() -> { ... return
 bf1.multiply(bf2);
 }
```

The pool thread gets the original lambda from the supplier & runs it asynchronously

# Overview of Common Functional Interfaces: Supplier

---

- Here's another example of a *Supplier* that's used in the Optional class

```
• public interface Supplier<T> { T get(); }

Map<String, String> beingMap = new HashMap<String, String>()
{ { put("Demon", "Naughty"); put("Angel", "Nice"); } };

String being = ...;

Optional<String> disposition =
 Optional.ofNullable(beingMap.get(being));

System.out.println("disposition of "
 + being + " = "
 + disposition.orElseGet(() -> "unknown"));
```

# Overview of Common Functional Interfaces: Supplier

- Here's another example of a *Supplier* that's used in the Optional class

- ```
public interface Supplier<T> { T get(); }
```

```
Map<String, String> beingMap = new HashMap<String, String>()
{ { put("Demon", "Naughty"); put("Angel", "Nice"); } };
```

```
String being = ...;
```

Create a map that associates beings with their personality traits

```
Optional<String> disposition =
Optional.ofNullable(beingMap.get(being));
```

```
System.out.println("disposition of "
+ being + " = "
+ disposition.orElseGet(() -> "unknown"));
```

Overview of Common Functional Interfaces: Supplier

- Here's another example of a *Supplier* that's used in the `Optional` class

- ```
public interface Supplier<T> { T get(); }
```

```
Map<String, String> beingMap = new HashMap<String, String>()
{ { put("Demon", "Naughty"); put("Angel", "Nice"); } };
```

```
String being = ...;
```

*Get the name of a being from somewhere (e.g., prompt user)*

```
Optional<String> disposition =
Optional.ofNullable(beingMap.get(being));
```

```
System.out.println("disposition of "
+ being + " = "
+ disposition.orElseGet(() -> "unknown"));
```

# Overview of Common Functional Interfaces: Supplier

- Here's another example of a *Supplier* that's used in the `Optional` class

- ```
public interface Supplier<T> { T get(); }
```

```
Map<String, String> beingMap = new HashMap<String, String>()
{ { put("Demon", "Naughty"); put("Angel", "Nice"); } };
```

```
String being = ...;
```

*Return an optional describing the specified being
if non-null, otherwise returns an empty Optional*

```
Optional<String> disposition =
    Optional.ofNullable(beingMap.get(being));
```

```
System.out.println("disposition of "
    + being + " = "
    + disposition.orElseGet(() -> "unknown"));
```

Overview of Common Functional Interfaces: Supplier

- Here's another example of a *Supplier* that's used in the `Optional` class

```
• public interface Supplier<T> { T get(); }

Map<String, String> beingMap = new HashMap<String, String>()
{ { put("Demon", "Naughty"); put("Angel", "Nice"); } };
```

```
String being = ...;
```

```
Optional<String> disposition =
Optional.ofNullable(beingMap.get(being));
```

```
System.out.println("disposition of "
+ being + " = "
+ disposition.orElseGet(() -> "unknown"));
```

*A container object which may or
may not contain a non-null value*

Overview of Common Functional Interfaces: Supplier

- Here's another example of a *Supplier* that's used in the Optional class

```
• public interface Supplier<T> { T get(); }

Map<String, String> beingMap = new HashMap<String, String>()
{ { put("Demon", "Naughty"); put("Angel", "Nice"); } };
```

```
String being = ...;
```

```
Optional<String> disposition =
Optional.ofNullable(beingMap.get(being));
```

Returns value if being is non-null

```
System.out.println("disposition of "
+ being + " = "
+ disposition.orElseGet(() -> "unknown"));
```

Overview of Common Functional Interfaces: Supplier

- Here's another example of a *Supplier* that's used in the Optional class

```
• public interface Supplier<T> { T get(); }

Map<String, String> beingMap = new HashMap<String, String>()
{ { put("Demon", "Naughty"); put("Angel", "Nice"); } };
```

```
String being = ...;
```

```
Optional<String> disposition =
    Optional.ofNullable(beingMap.get(being));
```

```
System.out.println("disposition of "
    + being + " = "
    + disposition.orElseGet(() -> "unknown"));
```

Returns supplier lambda value if being is not found

Overview of Common Functional Interfaces: Supplier

- Here's another example of a *Supplier* that's used in the Optional class

```
• public interface Supplier<T> { T get(); }

class Optional<T> {
    ...
    public T orElseGet(Supplier<? extends T> other) {
        return value != null
            ? value
            : other.get();
    }
}
```

Here's how the orElseGet() method uses the Supplier passed to it

Overview of Common Functional Interfaces: Supplier

- Here's another example of a *Supplier* that's used in the Optional class

```
• public interface Supplier<T> { T get(); }

class Optional<T> {
    ...
    public T orElseGet(Supplier<? extends T> other) {
        return value != null
            ? value
            : other.get();
    }
}
```

() -> "unknown"



The string literal "unknown" is bound to the supplier lambda parameter

Overview of Common Functional Interfaces: Supplier

- Here's another example of a *Supplier* that's used in the Optional class

```
• public interface Supplier<T> { T get(); }

class Optional<T> {
    ...
    public T orElseGet(Supplier<? extends T> other) {
        return value != null
            ? value
            : other.get();
    }
}
```

`() -> "unknown"`

"unknown"

The string "unknown" returns by orElseGet() if the value is null

Overview of Common Functional Interfaces: Supplier

- A *Supplier* can also be used for a zero-param constructor reference e.g.,

```
• public interface Supplier<T> { T get(); }

class CrDemo implements Runnable {
    String mString;

    void zeroParamConstructorRef() {
        Supplier<CrDemo> factory = CrDemo::new;
        CrDemo crDemo = factory.get();
        crDemo.run();
    }

    void run() { System.out.println(mString); }
    ...
}
```

Overview of Common Functional Interfaces: Supplier

- A *Supplier* can also be used for a zero-param constructor reference e.g.,

```
• public interface Supplier<T> { T get(); }

class CrDemo implements Runnable {
    String mString;

    void zeroParamConstructorRef() {
        Supplier<CrDemo> factory = CrDemo::new;
        CrDemo crDemo = factory.get();
        crDemo.run();
    }
}

void run() { System.out.println(mString); }
...
```

Create a supplier that's initialized with a zero-param constructor reference for CrDemo

Overview of Common Functional Interfaces: Supplier

- A *Supplier* can also be used for a zero-param constructor reference e.g.,

```
• public interface Supplier<T> { T get(); }

class CrDemo implements Runnable {
    String mString;

    void zeroParamConstructorRef() {
        Supplier<CrDemo> factory = CrDemo::new;
        CrDemo crDemo = factory.get();
        crDemo.run();
    }
}

void run() { System.out.println(mString); }
...
```

get() creates a *CrDemo* object using a constructor reference for the *CrDemo* "default" constructor

Overview of Common Functional Interfaces: Supplier

- A *Supplier* can also be used for a zero-param constructor reference e.g.,

```
• public interface Supplier<T> { T get(); }

class CrDemo implements Runnable {
    String mString;

    void zeroParamConstructorRef() {
        Supplier<CrDemo> factory = CrDemo::new;
        CrDemo crDemo = factory.get();
        crDemo.run();
    }
}

void run() { System.out.println(mString); }
...
```

Call a method in CrDemo to print the result

Overview of Common Functional Interfaces: Supplier

- Constructor references simplify creation of parameterizable factory methods

- ```
public interface Supplier<T> { T get(); }
```

```
class CrDemo implements Runnable {
```

```
...
```

```
static class CrDemoEx extends CrDemo {
```

*This class extends CrDemo & overrides its run() method to uppercase the string*

```
public void run() {
```

```
 System.out.println(mString.toUpperCase());
```

```
}
```

```
}
```

```
...
```

# Overview of Common Functional Interfaces: Supplier

- Constructor references simplify creation of parameterizable factory methods

```
• public interface Supplier<T> { T get(); }

class CrDemo implements Runnable {

 ...

 static class CrDemoEx
 extends CrDemo {

 public void run() {
 System.out.println(mString.toUpperCase());
 }
 }
 ...
}
```

*Print the upper-cased value of mString*

# Overview of Common Functional Interfaces: Supplier

- Constructor references simplify creation of parameterizable factory methods

- ```
public interface Supplier<T> { T get(); }
```

```
class CrDemo implements Runnable {
```

```
    ...
```

```
    void zeroParamConstructorRefEx() {
```



Demonstrate how suppliers can be used as factories for multiple zero-parameter constructor references

```
Supplier<CrDemo> crDemoFactory = CrDemo::new;
```

```
Supplier<CrDemoEx> crDemoFactoryEx = CrDemoEx::new;
```

```
runDemo(crDemoFactory);
```

```
runDemo(crDemoFactoryEx);
```

```
}
```

```
...
```

Overview of Common Functional Interfaces: Supplier

- Constructor references simplify creation of parameterizable factory methods

- ```
public interface Supplier<T> { T get(); }
```

```
class CrDemo implements Runnable {
```

```
 ...
```

```
 void zeroParamConstructorRefEx() {
```

*Assign a constructor reference to a supplier that acts as a factory for a zero-param object of CrDemo/CrDemoEx*

```
Supplier<CrDemo> crDemoFactory = CrDemo::new;
```

```
Supplier<CrDemoEx> crDemoFactoryEx = CrDemoEx::new;
```

```
runDemo(crDemoFactory);
```

```
runDemo(crDemoFactoryEx);
```

```
}
```

```
...
```

# Overview of Common Functional Interfaces: Supplier

- Constructor references simplify creation of parameterizable factory methods

- ```
public interface Supplier<T> { T get(); }
```

```
class CrDemo implements Runnable {
```

```
    ...
```

```
    void zeroParamConstructorRefEx() {
```

```
        Supplier<CrDemo> crDemoFactory = CrDemo::new;
```

```
        Supplier<CrDemoEx> crDemoFactoryEx = CrDemoEx::new;
```

```
        runDemo(crDemoFactory);
```

```
        runDemo(crDemoFactoryEx);
```

```
}
```

```
    ...
```

This helper method invokes the given supplier to create a new object & call its run() method

Overview of Common Functional Interfaces: Supplier

- Constructor references simplify creation of parameterizable factory methods

```
• public interface Supplier<T> { T get(); }

class CrDemo implements Runnable {

    ...

    <T extends Runnable> void runDemo(Supplier<T> factory) {
        factory.get().run();
    }
    ...
}
```



Use the given factory to create a new object & call its run() method

Overview of Common Functional Interfaces: Supplier

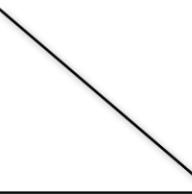
- Constructor references simplify creation of parameterizable factory methods

```
• public interface Supplier<T> { T get(); }

class CrDemo implements Runnable {

    ...

    <T extends Runnable> void runDemo(Supplier<T> factory) {
        factory.get().run();
    }
    ...
}
```



This call encapsulates details of the concrete constructor that's used to create an object!

Overview of Common Functional Interfaces: Supplier

- Arbitrary constructors w/params can also be supported in Java 8, e.g.

```
• public interface Supplier<T> { T get(); }

class CrDemo implements Runnable { ...

interface TriFactory<A, B, C, R> { R of(A a, B b, C c); }
```

Custom functional interfaces can be defined for arbitrary constructors w/params

```
void threeParamConstructorRef() {
    TriFactory<String, Integer, Long, CrDemo> factory =
        CrDemo::new;

    factory.of("The answer is ", 4, 2L).run();
}

CrDemo(String s, Integer i, Long l)
{ mString = s + i + l; } ...
```

This capability is unrelated to the Supplier interface..

Overview of Common Functional Interfaces: Supplier

- Arbitrary constructors w/params can also be supported in Java 8, e.g.

```
• public interface Supplier<T> { T get(); }

class CrDemo implements Runnable { ... }

interface TriFactory<A, B, C, R> { R of(A a, B b, C c); }
```

```
void threeParamConstructorRef() {
    TriFactory<String, Integer, Long, CrDemo> factory =
        CrDemo::new;

    factory.of("The answer is ", 4, 2L).run();
}
```

Create a factory that's initialized with a three-param constructor reference

```
CrDemo(String s, Integer i, Long l)
{ mString = s + i + l; } ...
```

Overview of Common Functional Interfaces: Supplier

- Arbitrary constructors w/params can also be supported in Java 8, e.g.

```
• public interface Supplier<T> { T get(); }

class CrDemo implements Runnable { ... }

interface TriFactory<A, B, C, R> { R of(A a, B b, C c); }
```

```
void threeParamConstructorRef() {
    TriFactory<String, Integer, Long, CrDemo> factory =
        CrDemo::new;

    factory.of("The answer is ", 4, 2L).run();
}

CrDemo(String s, Integer i, Long l)
{ mString = s + i + l; } ...
```

Create/print a 3-param
instance of CrDemo

Overview of Functional Interfaces: Consumer

Overview of Common Functional Interfaces: Consumer

- A *Consumer* accepts a parameter & returns no results, e.g.,
 - `public interface Consumer<T> { void accept(T t); }`

Overview of Common Functional Interfaces: Consumer

- A *Consumer* accepts a parameter & returns no results, e.g.,
 - `public interface Consumer<T> { void accept(T t); }`

Overview of Common Functional Interfaces: Consumer

- A *Consumer* accepts a parameter & returns no results, e.g.,

- ```
public interface Consumer<T> { void accept(T t); }
```

```
List<Thread> threads = Arrays.asList(new Thread("Larry"),
 new Thread("Curly"),
 new Thread("Moe"));
```

*Create a list of threads with  
names of the three stooges*

```
threads.forEach(System.out::println);
threads.sort(Comparator.comparing(Thread::getName));
threads.forEach(System.out::println);
```

# Overview of Common Functional Interfaces: Consumer

- A *Consumer* accepts a parameter & returns no results, e.g.,

```
• public interface Consumer<T> { void accept(T t); }
```

```
List<Thread> threads = Arrays.asList(new Thread("Larry"),
 new Thread("Curly"),
 new Thread("Moe"));
```

*Print out threads using forEach()*

```
threads.forEach(System.out::println);
threads.sort(Comparator.comparing(Thread::getName));
threads.forEach(System.out::println);
```

# Overview of Common Functional Interfaces: Consumer

---

- A *Consumer* accepts a parameter & returns no results, e.g.,

- ```
public interface Consumer<T> { void accept(T t); }

public interface Iterable<T> {

    ...

    default void forEach(Consumer<? super T> action) {
        for (T t : this) {
            action.accept(t);
        }
    }
}
```

Here's how the `forEach()` method uses the `Consumer` passed to it

Overview of Common Functional Interfaces: Consumer

- A *Consumer* accepts a parameter & returns no results, e.g.,

```
• public interface Consumer<T> { void accept(T t); }

public interface Iterable<T> {
    ...
    default void forEach(Consumer<? super T> action) {
        for (T t : this) {
            action.accept(t);
        }
    }
}
```

System.out::println

The consumer parameter is bound to the System.out::println method reference

Overview of Common Functional Interfaces: Consumer

- A *Consumer* accepts a parameter & returns no results, e.g.,

```
• public interface Consumer<T> { void accept(T t); }

public interface Iterable<T> {
    ...
    default void forEach(Consumer<? super T> action) {
        for (T t : this) {
            action.accept(t);
        }
    }
}
```



System.out.println(t)

A black arrow points from the word "action" in the code above to the red text "System.out.println(t)" in the callout box.

The accept() method is replaced by the call to System.out.println()

Other Properties of Functional Interfaces

Other Properties of Functional Interfaces

- Functional interfaces may also have default methods and/or static methods

```
interface Comparator<T> {
    int compare(T o1, T o2);

    boolean equals(Object obj);

    default Comparator<T> reversed()
    { return Collections.reverseOrder(this); }

    static <T extends Comparable<? super T>>
    Comparator<T> reverseOrder()
    { return Collections.reverseOrder(); }

    ...
}
```

Other Properties of Functional Interfaces

- Functional interfaces may also have default methods and/or static methods, e.g.

```
interface Comparator<T> {  
    int compare(T o1, T o2);  
  
    boolean equals(Object obj);
```

A comparison function that imposes a total ordering on some collection of objects

```
default Comparator<T> reversed()  
{ return Collections.reverseOrder(this); }
```

```
static <T extends Comparable<? super T>>  
Comparator<T> reverseOrder()  
{ return Collections.reverseOrder(); }
```

...

Other Properties of Functional Interfaces

- Functional interfaces may also have default methods and/or static methods, e.g.

```
interface Comparator<T> {  
    int compare(T o1, T o2);  
  
    boolean equals(Object obj);  
  
    default Comparator<T> reversed()  
    { return Collections.reverseOrder(this); }  
  
    static <T extends Comparable<? super T>>  
    Comparator<T> reverseOrder()  
    { return Collections.reverseOrder(); }  
    ...
```

*An abstract method in
this functional interface*

Other Properties of Functional Interfaces

- Functional interfaces may also have default methods and/or static methods, e.g.

```
interface Comparator<T> {  
    int compare(T o1, T o2);  
  
    boolean equals(Object obj);
```

A default method provides the default implementation, which can be overridden

```
default Comparator<T> reversed()  
{ return Collections.reverseOrder(this); }  
  
static <T extends Comparable<? super T>>  
Comparator<T> reverseOrder()  
{ return Collections.reverseOrder(); }  
...
```

Other Properties of Functional Interfaces

- Functional interfaces may also have default methods and/or static methods, e.g.

```
interface Comparator<T> {
    int compare(T o1, T o2);

    boolean equals(Object obj);

    default Comparator<T> reversed()
    { return Collections.reverseOrder(this); }

    static <T extends Comparable<? super T>>
    Comparator<T> reverseOrder()
    { return Collections.reverseOrder(); }
    ...
}
```

A static method provides the one-&-only implementation

Other Properties of Functional Interfaces

- Functional interfaces may also have default methods and/or static methods, e.g.

```
interface Comparator<T> {  
    int compare(T o1, T o2);  
  
    boolean equals(Object obj);
```

An abstract method that overrides a public java.lang.Object method does not count as part of the interface's abstract method count

```
default Comparator<T> reversed()  
{ return Collections.reverseOrder(this); }  
  
static <T extends Comparable<? super T>>  
Comparator<T> reverseOrder()  
{ return Collections.reverseOrder(); }  
...
```

End of Overview of Java 8 Functional Interfaces

Overview of Java 8 Streams (Part 1)

Douglas C. Schmidt

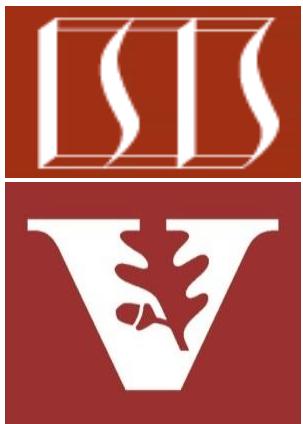
d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

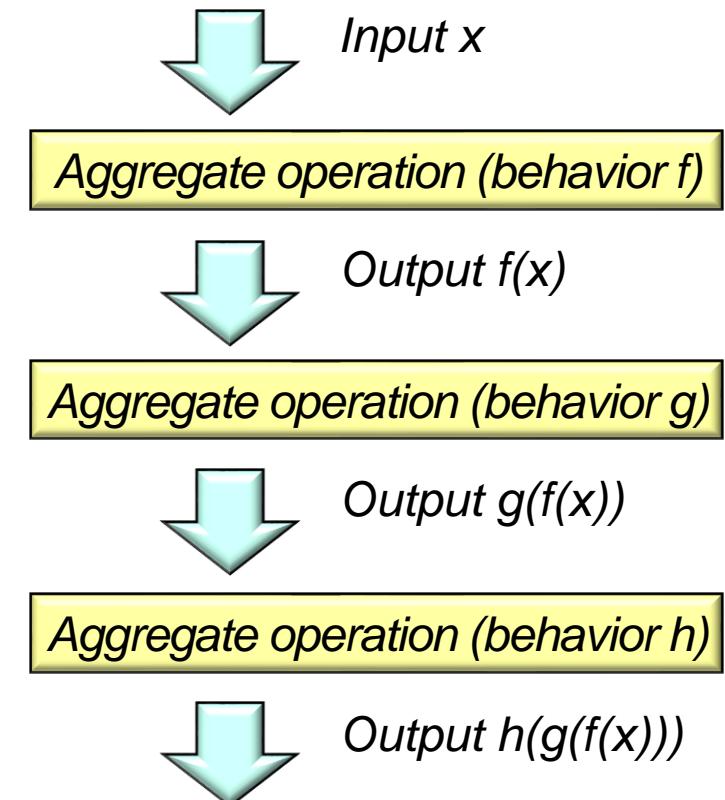
Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA



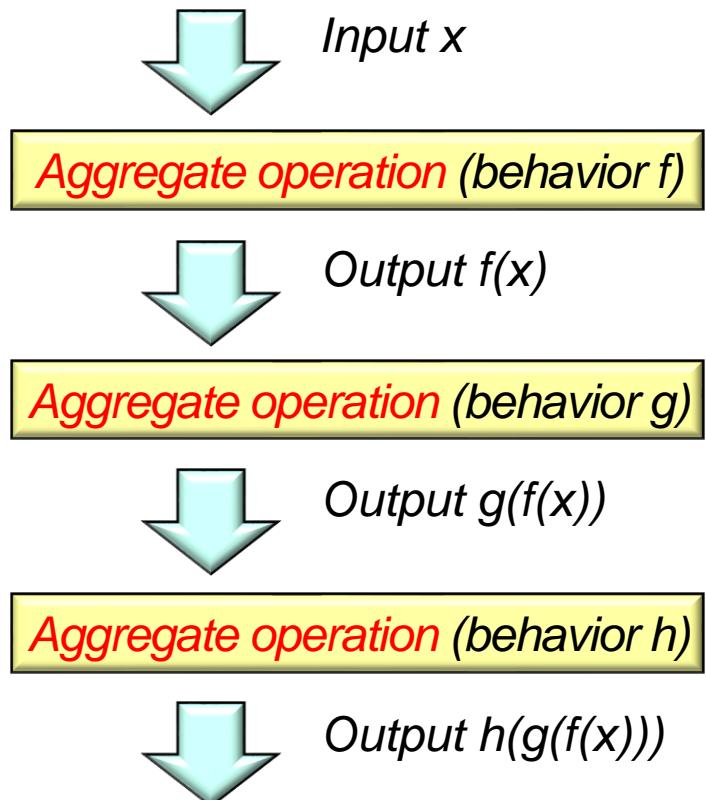
Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of Java 8 streams



Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of Java 8 streams, e.g.,
 - Fundamentals of streams

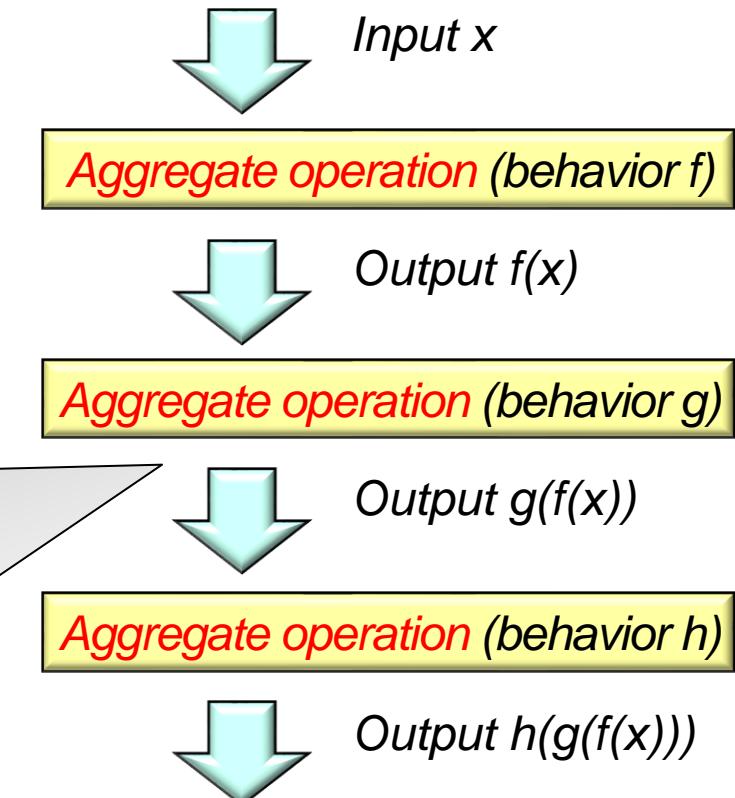


Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of Java 8 streams, e.g.,
 - Fundamentals of streams
 - We'll use an example program to illustrate key concepts

Stream

```
.of("horatio",
    "laertes",
    "Hamlet", ...)
.filter(s -> toLowerCase
        (s.charAt(0)) == 'h')
.map(this::capitalize)
.sorted()
.forEach(System.out::println);
```



Overview of Java 8 Streams

Overview of Java 8 Streams

- Java 8 streams are a framework in the Java class library that provides several key benefits to programs



What's New in JDK 8

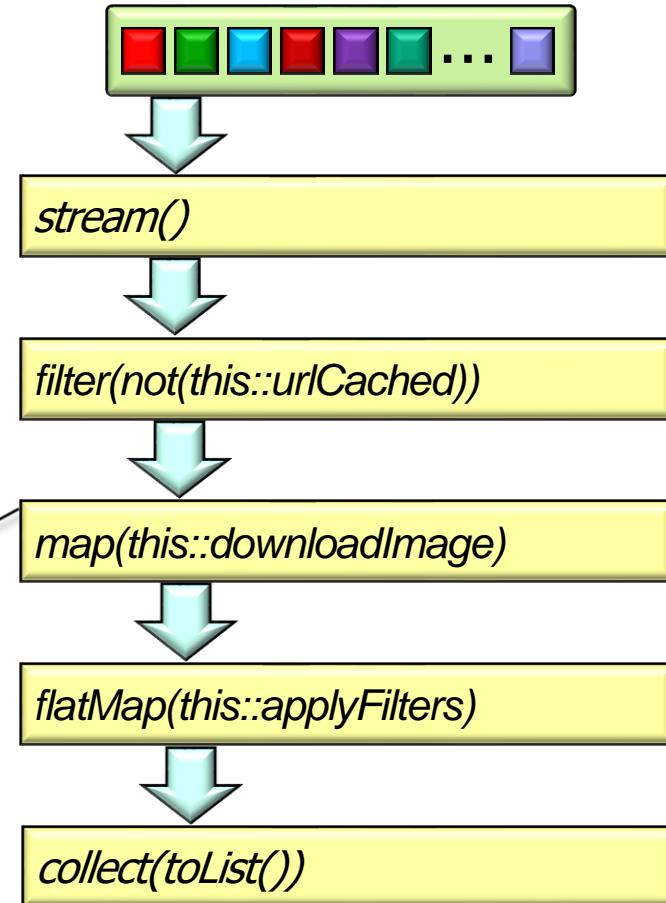
Java Platform, Standard Edition 8 is a major feature release. This document summarizes features and enhancements in Java SE 8 and in JDK 8, Oracle's implementation of Java SE 8. Click the component name for a more detailed description of the enhancements for that component.

- Java Programming Language
 - Lambda Expressions, a new language feature, has been introduced in this release. They enable you to treat functionality as a method argument, or code as data. Lambda expressions let you express instances of single-method interfaces (referred to as functional interfaces) more compactly.
 - Method references provide easy-to-read lambda expressions for methods that already have a name.
 - Default methods enable new functionality to be added to the interfaces of libraries and ensure binary compatibility with code written for older versions of those interfaces.
 - Repeating Annotations provide the ability to apply the same annotation type more than once to the same declaration or type use.
 - Type Annotations provide the ability to apply an annotation anywhere a type is used, not just on a declaration. Used with a pluggable type system, this feature enables improved type checking of your code.
 - Improved type inference.
 - Method parameter reflection.
- Collections
 - Classes in the new `java.util.stream` package provide a Stream API to support functional-style operations on streams of elements. The Stream API is integrated into the Collections API, which enables bulk operations on collections, such as sequential or parallel map-reduce transformations.
 - Performance Improvement for HashMaps with Key Collisions

See docs.oracle.com/javase/tutorial/collectionsstreams

Overview of Java 8 Streams

- Java 8 streams are a framework in the Java class library that provides several key benefits to programs, e.g.
 - Manipulate flows of data declaratively via function composition

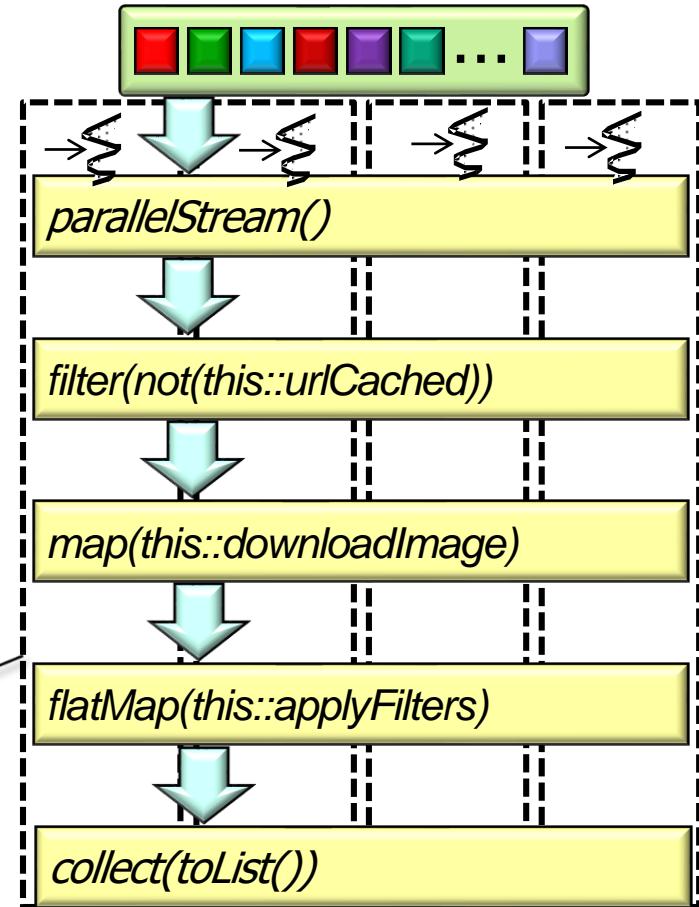


*This stream expresses **what** operations to perform, not **how** to perform them*

Overview of Java 8 Streams

- Java 8 streams are a framework in the Java class library that provides several key benefits to programs, e.g.
 - Manipulate flows of data declaratively via function composition
 - Enable transparent parallelization without the need to write any multi-threaded code

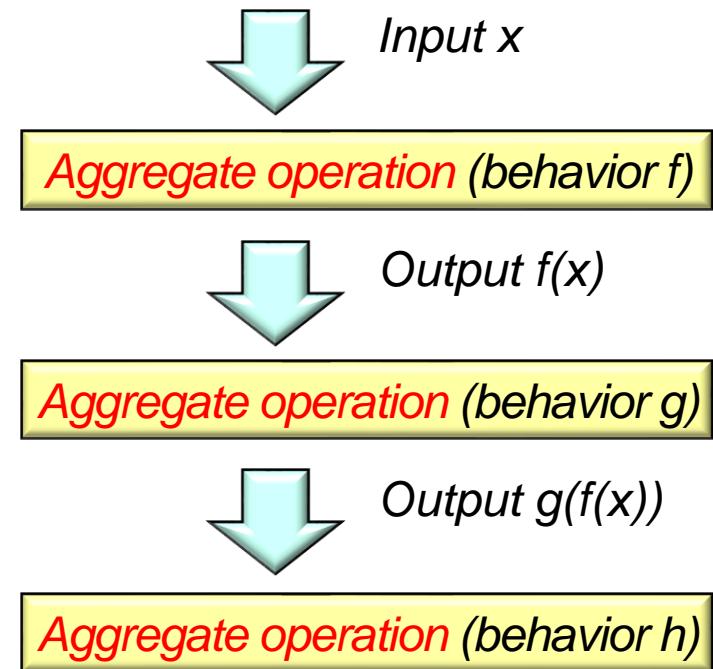
The data elements in this stream are automatically mapped to processor cores



See docs.oracle.com/javase/tutorial/collections/stream/parallelism.html

Overview of Java 8 Streams

- A stream is a pipeline of aggregate operations that process a sequence of elements (aka, “values” or “data”)

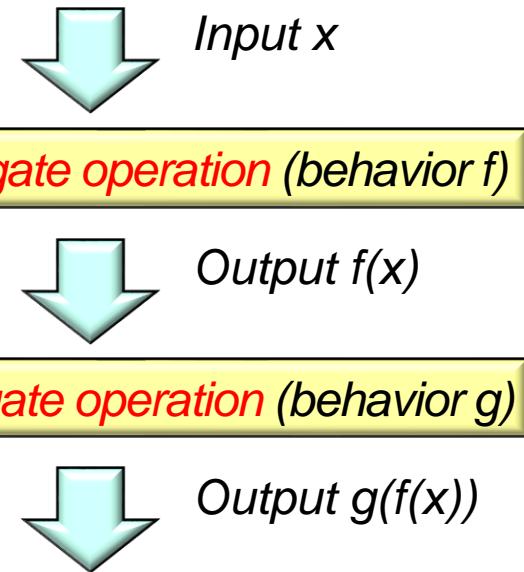


Overview of Java 8 Streams

- A stream is a pipeline of aggregate operations that process a sequence of elements (aka, "values" or "data")

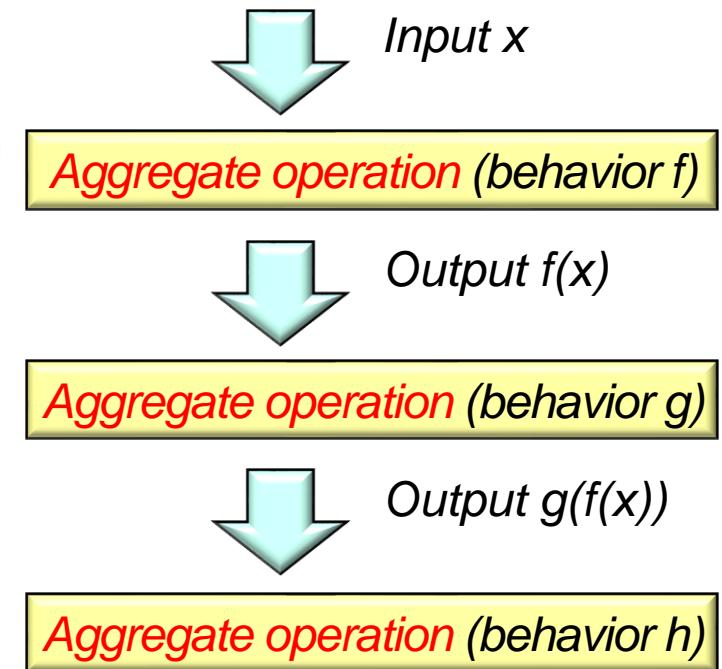


An aggregate operation is a higher-order function that applies a "behavior" parameter to every element in a stream



Overview of Java 8 Streams

- A stream is a pipeline of aggregate operations that process a sequence of elements (aka, “values” or “data”)



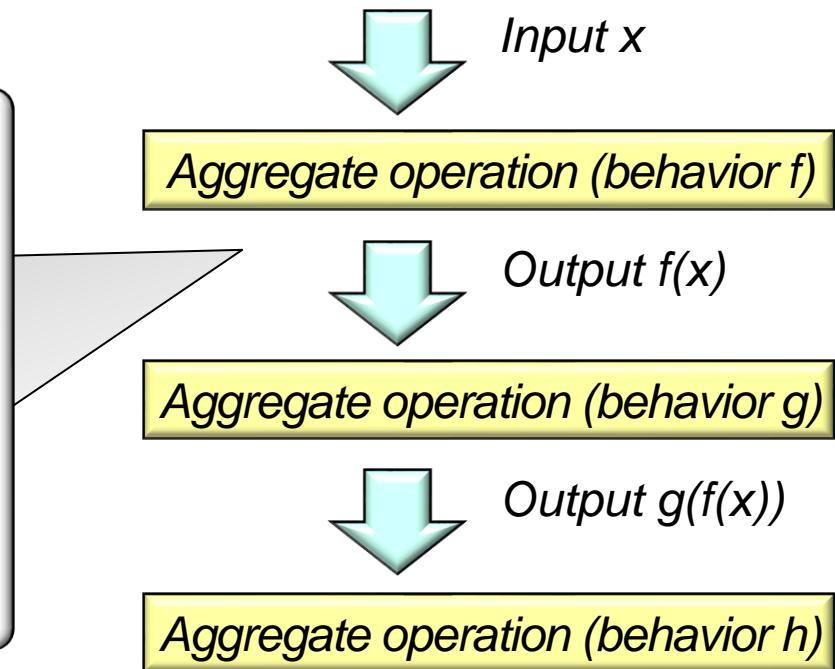
A stream is conceptually unbounded, though it's often bounded by practical constraints

Overview of Java 8 Streams

- A stream is a pipeline of aggregate operations that process a sequence of elements (aka, "values" or "data")

Stream

```
.of("horatio",
    "laertes",
    "Hamlet", ...)
.filter(s -> toLowerCase
        (s.charAt(0)) == 'h')
.map(this::capitalize)
.sorted()
.forEach(System.out::println);
```



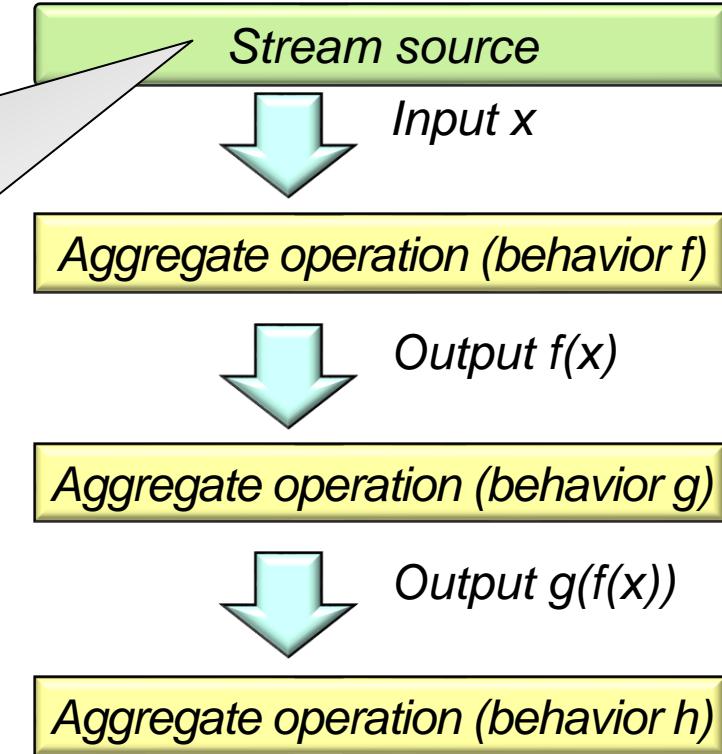
Overview of Java 8 Streams

- A stream is created via a factory method

Stream

```
.of("horatio",
    "laertes",
    "Hamlet", ...)
```

...

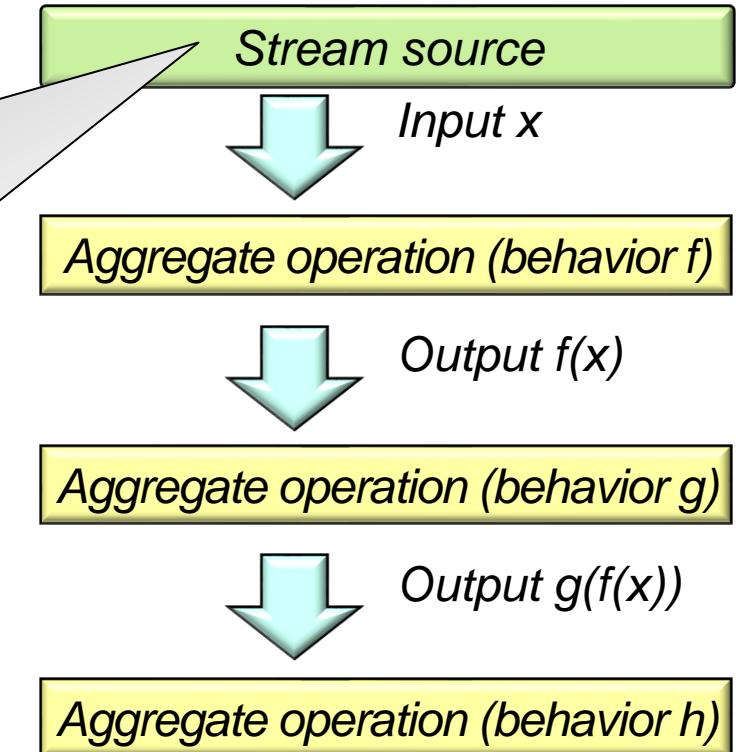
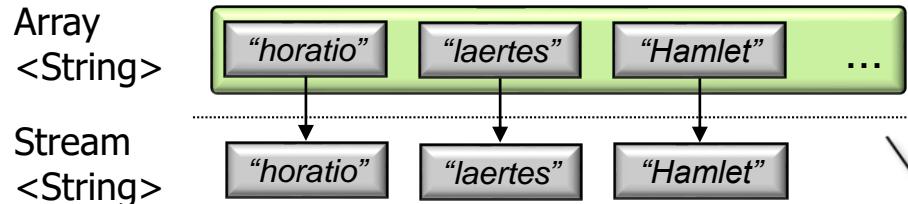


See en.wikipedia.org/wiki/Factory_method_pattern

Overview of Java 8 Streams

- A stream is created via a factory method

```
Stream  
.of ("horatio",  
       "laertes",  
       "Hamlet", ...) ...
```



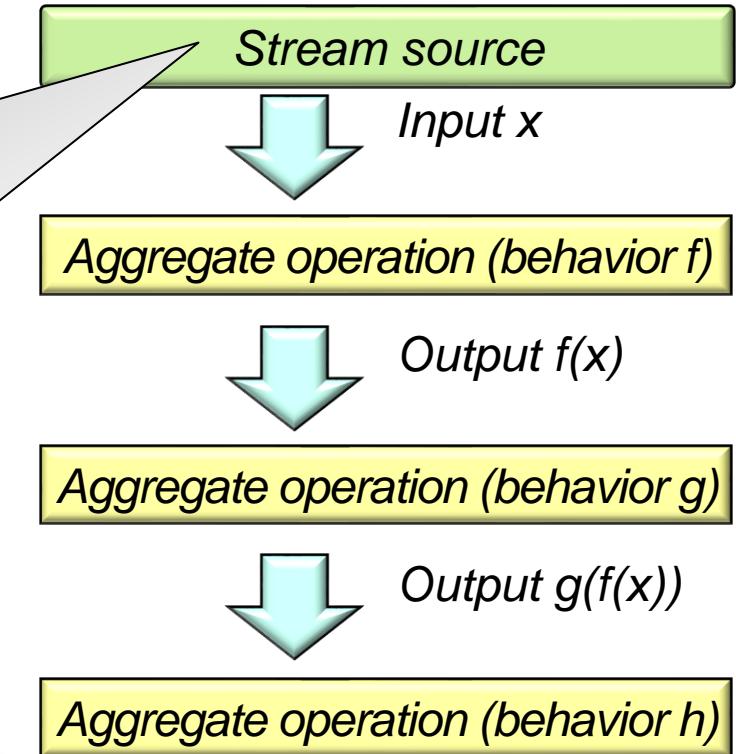
The of() factory method converts an array of T into a stream of T

See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#of

Overview of Java 8 Streams

- A stream is created via a factory method

```
List<String> l1 = ...;  
List<String> l2 = ...;  
List<String> l3 = ...;  
  
Stream  
.of(l1, l2, l3)  
.flatMap(List::stream)  
...  
.forEach(System.out::println);
```

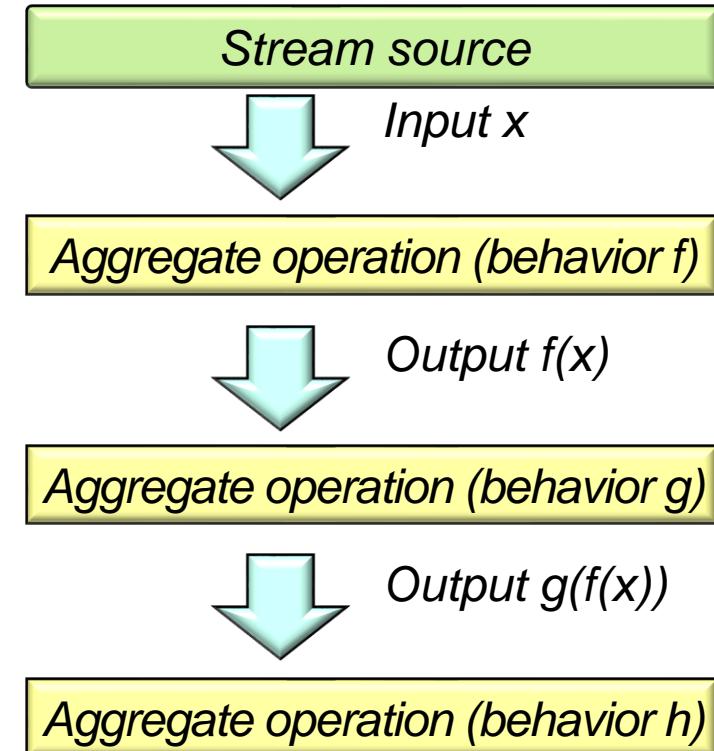


of() is flexible, especially when combined with other aggregate operations

Overview of Java 8 Streams

- A stream is created via a factory method

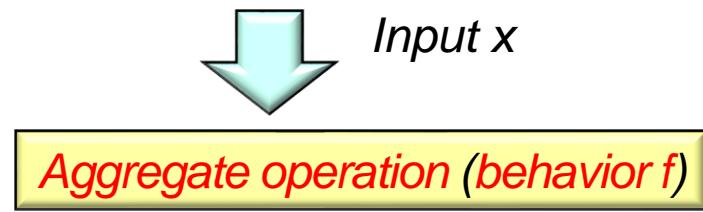
```
collection.stream()  
collection.parallelStream()  
Pattern.compile(...).splitAsStream()  
Stream.of(value1,... ,valueN)  
Arrays.stream(array)  
Arrays.stream(array, start, end)  
Files.lines(file_path)  
"string".chars()  
Stream.builder().add(...).build()  
Stream.generate(generate_expression)  
Files.list(file_path)  
Files.find(file_path, max_depth, mathcher)  
Stream.generate(iterator::next)  
Stream.iterate(init_value, generate_expression)  
StreamSupport.stream(iterable.splitter(), false)  
...
```



There are many other factory methods that create streams

Overview of Java 8 Streams

- An aggregate operation performs a *behavior* on each element in a stream



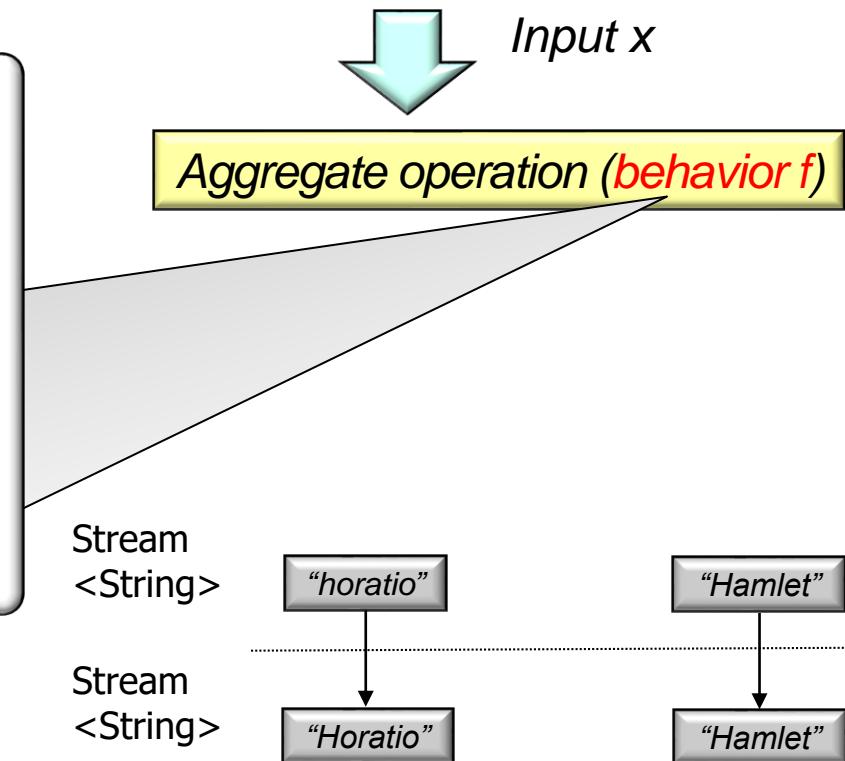
A behavior is implemented by a lambda expression or method reference

Overview of Java 8 Streams

- An aggregate operation performs a *behavior* on each element in a stream

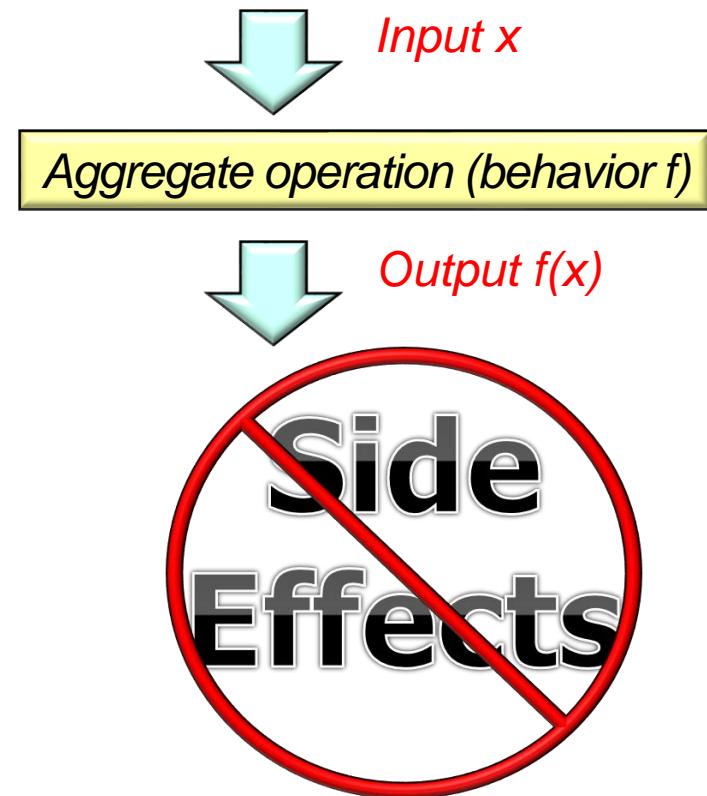
```
Stream
.of("horatio",
     "laertes",
     "Hamlet", ...)
.filter(s -> toLowerCase
        (s.charAt(0)) == 'h')
.map(this::capitalize)
.sorted()
.forEach(System.out::println);
```

Method reference



Overview of Java 8 Streams

- An aggregate operation performs a *behavior* on each element in a stream
 - Ideally, a behavior's output in a stream depends only on its input arguments

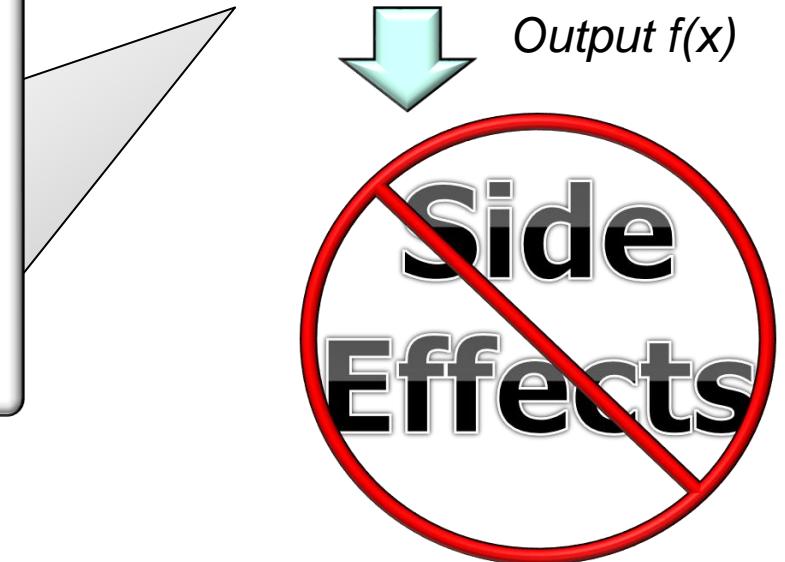
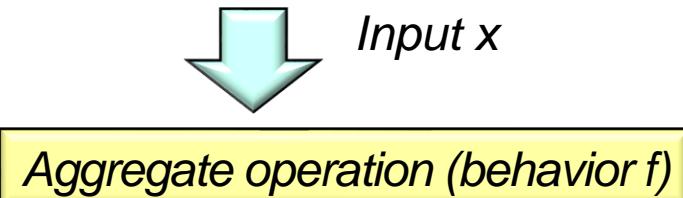


See [en.wikipedia.org/wiki/Side_effect_\(computer_science\)](https://en.wikipedia.org/wiki/Side_effect_(computer_science))

Overview of Java 8 Streams

- An aggregate operation performs a *behavior* on each element in a stream
 - Ideally, a behavior's output in a stream depends only on its input arguments

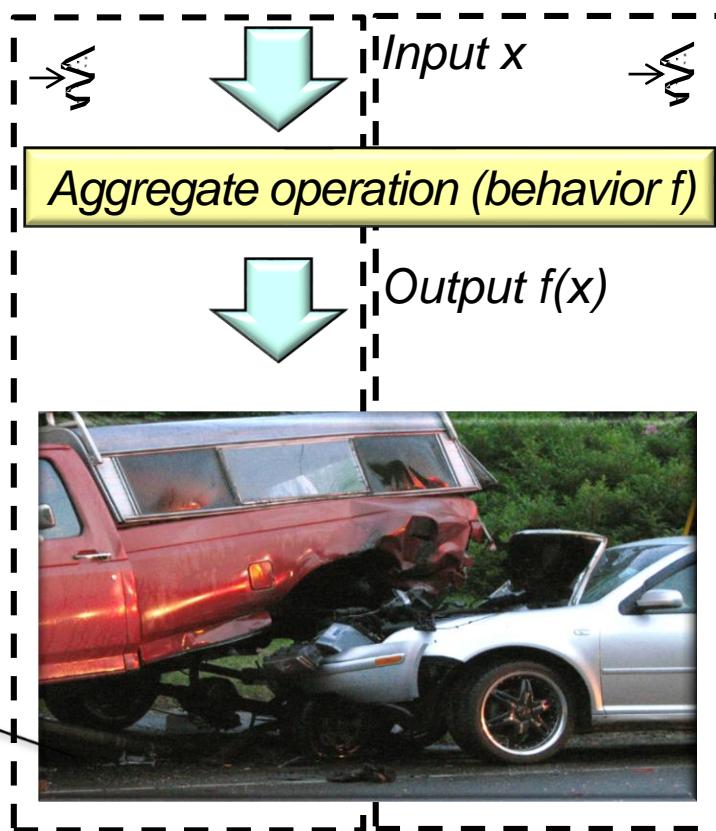
```
String capitalize(String s) {  
    if (s.length() == 0)  
        return s;  
    return s.substring(0, 1)  
        .toUpperCase()  
        + s.substring(1)  
        .toLowerCase();
```



Overview of Java 8 Streams

- An aggregate operation performs a *behavior* on each element in a stream
 - Ideally, a behavior's output in a stream depends only on its input arguments
 - Behaviors with side-effects likely incur race conditions in parallel streams

Race conditions arise in software when an application depends on the sequence or timing of threads for it to operate properly



See en.wikipedia.org/wiki/Race_condition#Software

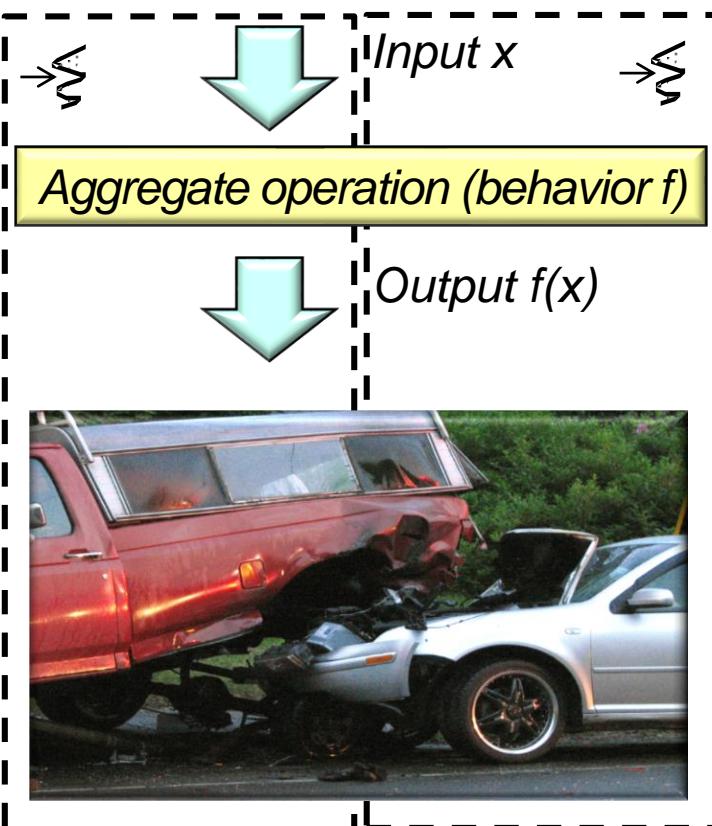
Overview of Java 8 Streams

- An aggregate operation performs a *behavior* on each element in a stream
 - Ideally, a behavior's output in a stream depends only on its input arguments
 - Behaviors with side-effects likely incur race conditions in parallel streams



ONLY YOU

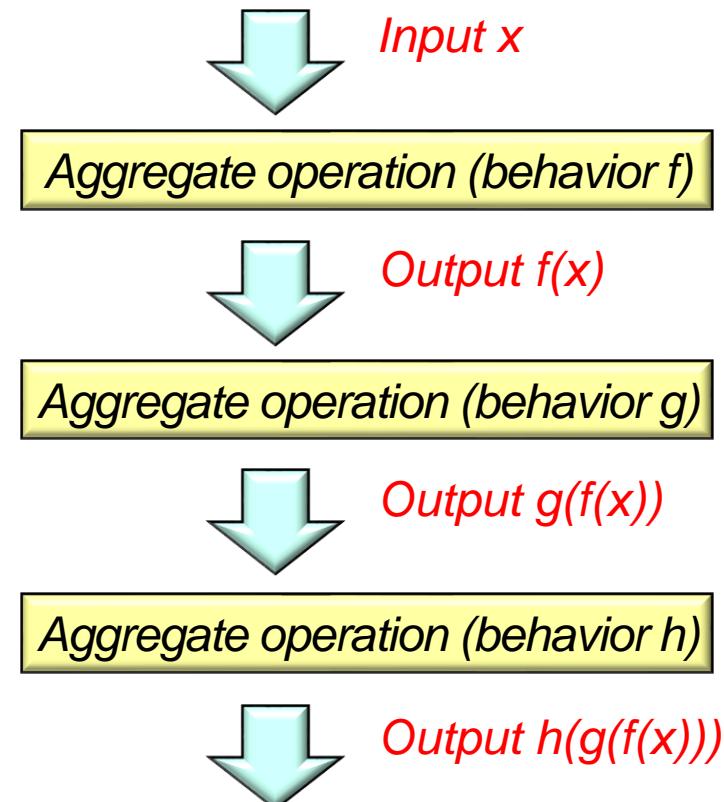
***Only you can
prevent
race conditions!***



In Java you must avoid race conditions, i.e., the compiler & JVM won't save you..

Overview of Java 8 Streams

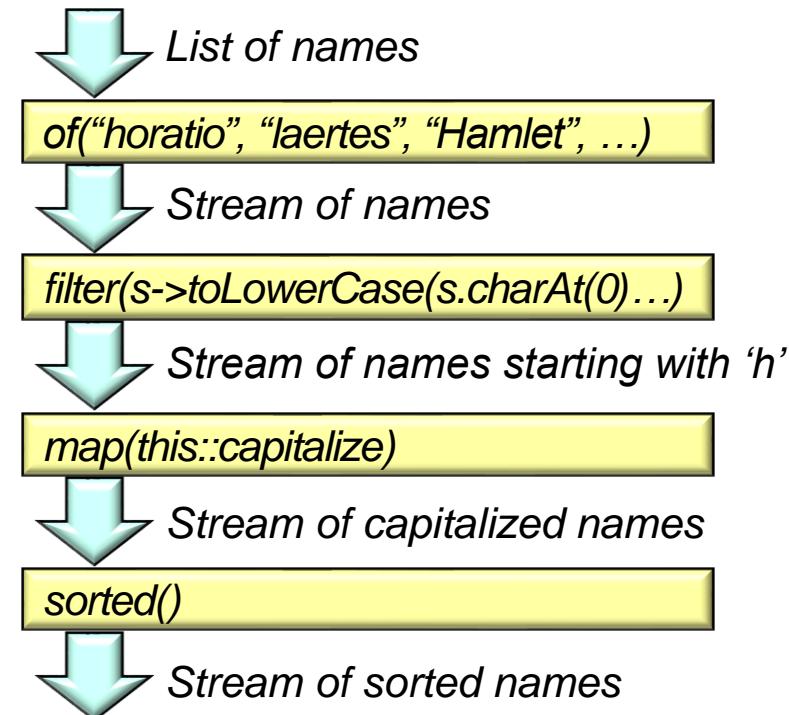
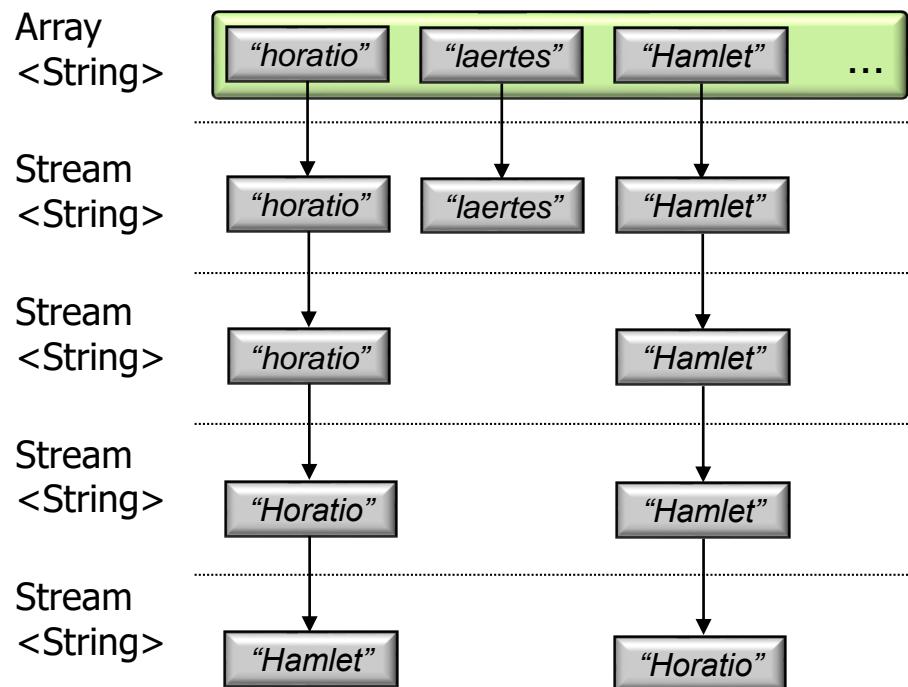
- Streams enhance flexibility by forming a “processing pipeline” that chains multiple aggregate operations together



See [en.wikipedia.org/wiki/Pipeline_\(software\)](https://en.wikipedia.org/wiki/Pipeline_(software))

Overview of Java 8 Streams

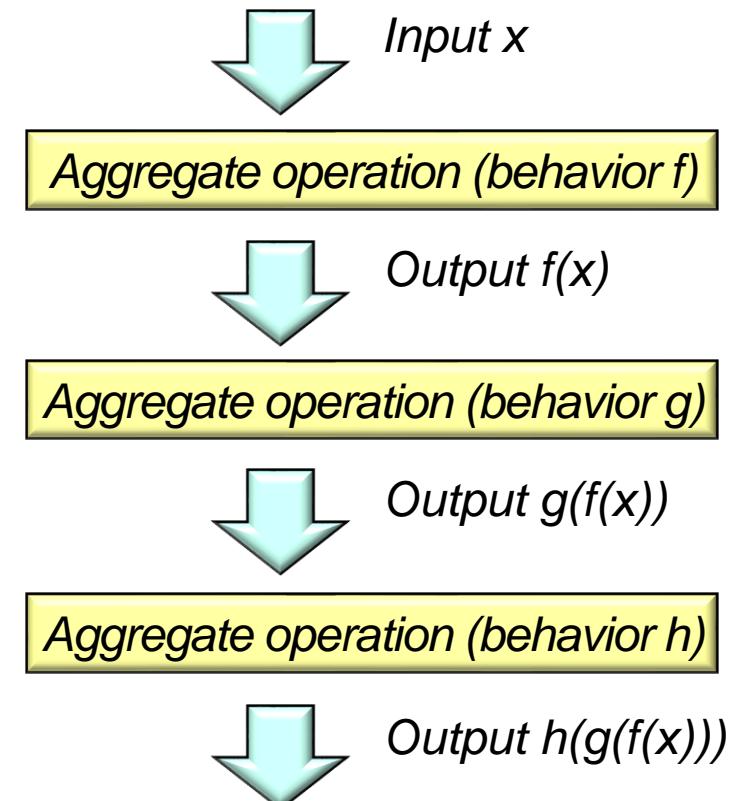
- Streams enhance flexibility by forming a “processing pipeline” that chains multiple aggregate operations together



Each aggregate operation in the pipeline can filter and/or transform the stream

Overview of Java 8 Streams

- A stream holds no non-transient storage



Overview of Java 8 Streams

- Every stream works very similarly



Overview of Java 8 Streams

- Every stream works very similarly
 - Starts with a source of data



Stream

```
.of("horatio",
    "laertes",
    "Hamlet", ...)  
...
```

e.g., a Java **array**, collection, generator function, or input channel

Overview of Java 8 Streams

- Every stream works very similarly
 - Starts with a source of data



```
List<String> characters =  
    Arrays.asList("horatio",  
                 "laertes",  
                 "Hamlet", ...);
```

```
characters  
.stream()  
...
```

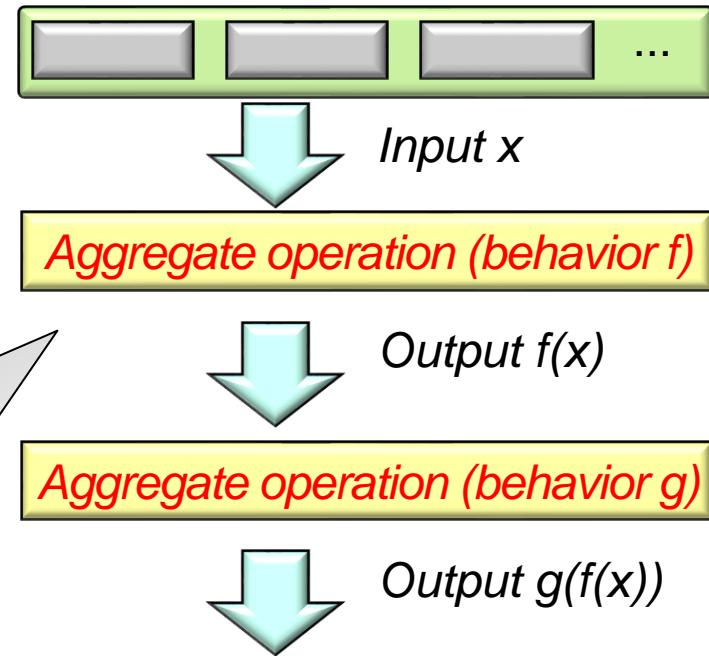
e.g., a Java array, collection, generator function, or input channel

Overview of Java 8 Streams

- Every stream works very similarly
 - Starts with a source of data
 - Processes data through a pipeline of intermediate operations that each map an input stream to an output stream

Stream

```
.of("horatio", "laertes",
    "Hamlet", ...)
.filter(s -> toLowerCase
        (s.charAt(0)) == 'h')
.map(this::capitalize)
.sorted()
...
```

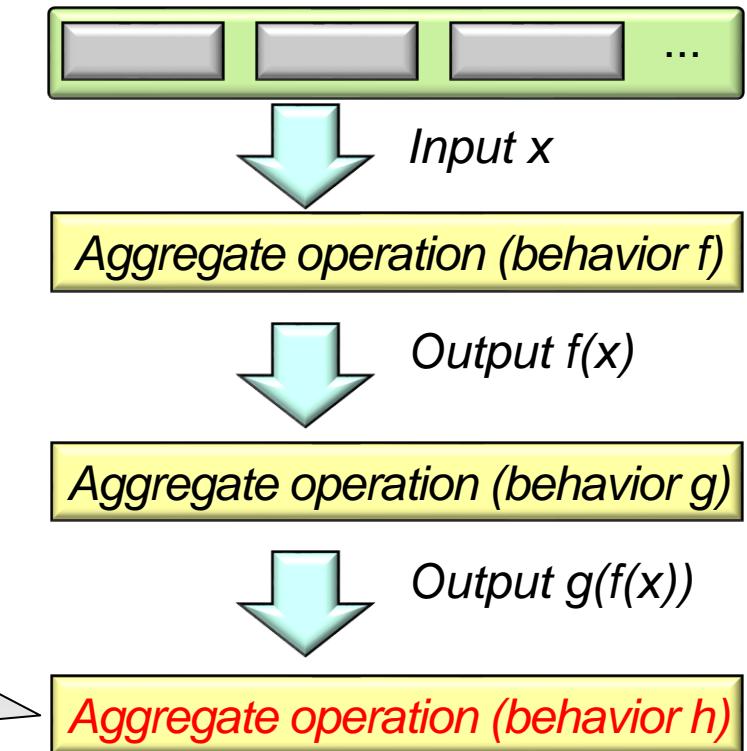


Examples of intermediate operations include filter(), map(), & flatMap()

Overview of Java 8 Streams

- Every stream works very similarly
 - Starts with a source of data
 - Processes data through a pipeline of intermediate operations that map an input stream to an output stream
 - Finishes with a terminal operation that yields a non-stream result

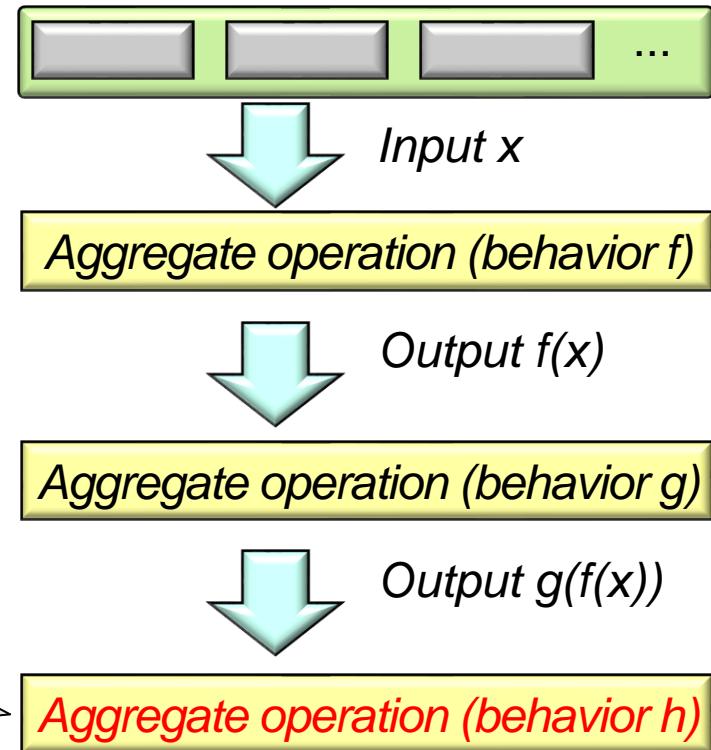
```
...  
.filter(s -> toLowerCase  
        (s.charAt(0)) == 'h')  
.map(this::capitalize)  
.sorted()  
.forEach(System.out::println);
```



Overview of Java 8 Streams

- Every stream works very similarly
 - Starts with a source of data
 - Processes data through a pipeline of intermediate operations that each map an input stream to an output stream
 - Finishes with a terminal operation that yields a non-stream result

```
...  
.filter(s -> toLowerCase  
        (s.charAt(0)) == 'h')  
.map(this::capitalize)  
.sorted()  
.forEach(System.out::println);
```



A terminal operation triggers processing of intermediate operations in a stream

Overview of Java 8 Streams

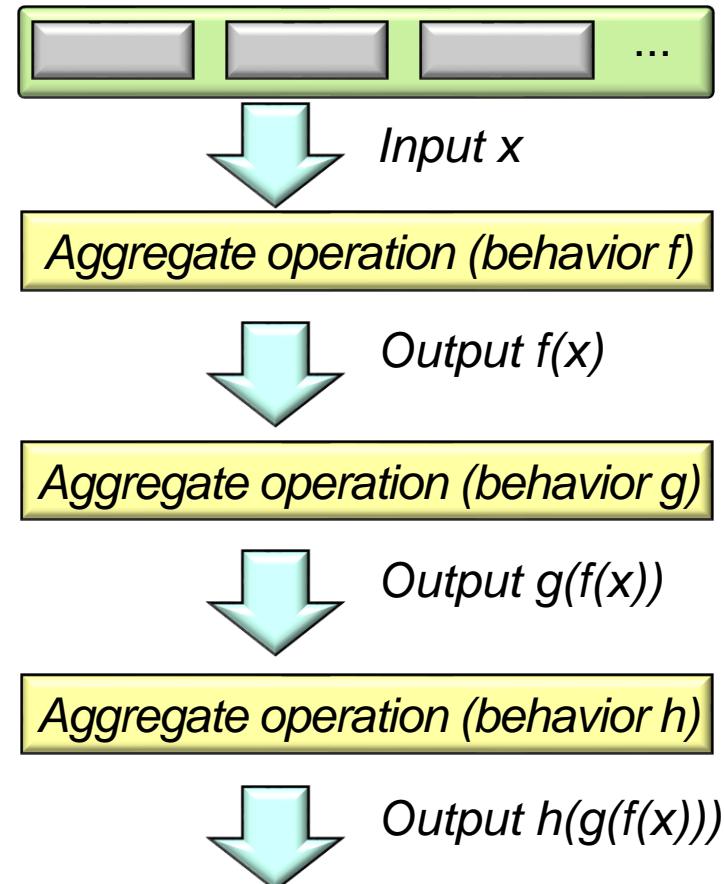
- Every stream works very similarly
 - Starts with a source of data
 - Processes data through a pipeline of intermediate operations that each map an input stream to an output stream
 - Finishes with a terminal operation that yields a non-stream result



Each stream *must* have one (& only one) terminal operation

Overview of Java 8 Streams

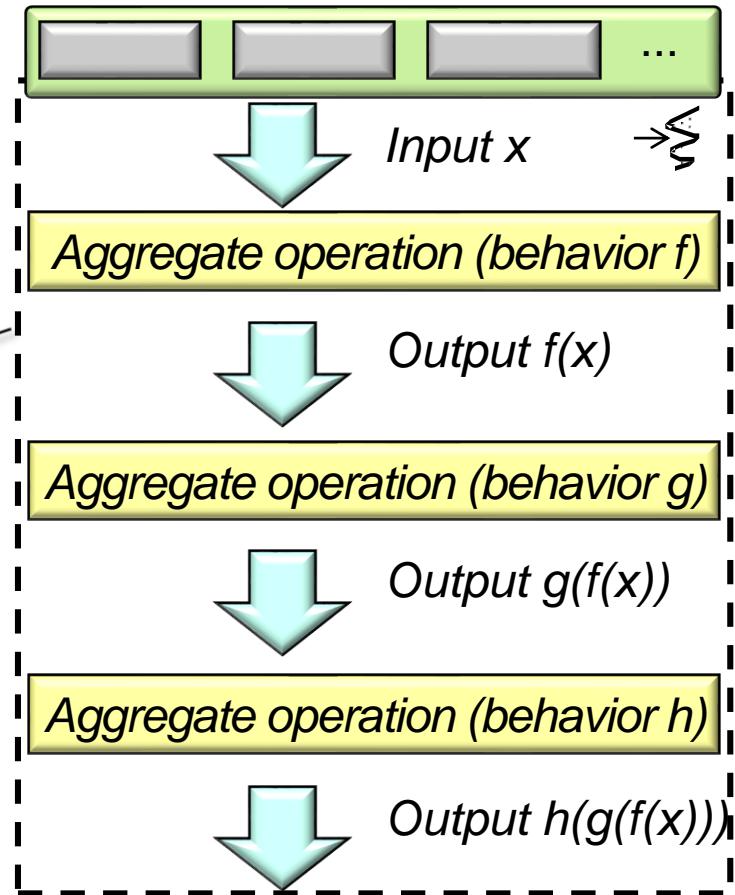
- Each aggregate operation in a stream runs its behavior sequentially by default



Overview of Java 8 Streams

- Each aggregate operation in a stream runs its behavior sequentially by default
 - i.e., one at a time in a single thread

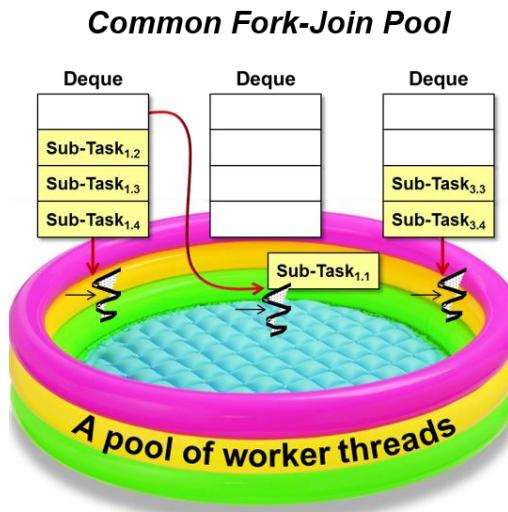
We'll cover sequential streams first



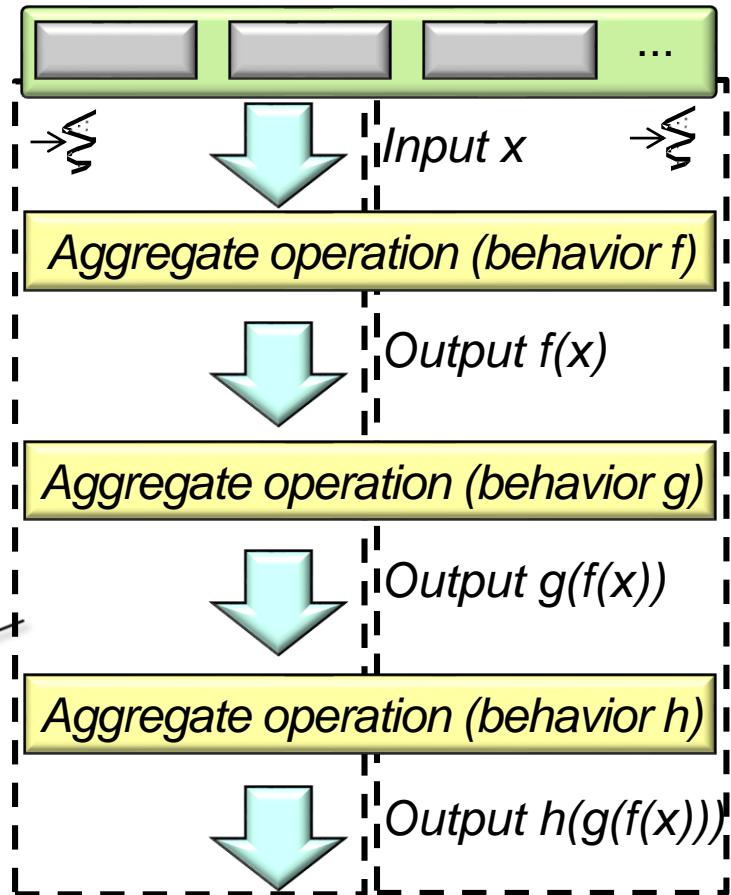
See docs.oracle.com/javase/tutorial/collections/streams

Overview of Java 8 Streams

- A Java 8 parallel stream splits its elements into multiple chunks & uses a common fork-join pool to process the chunks independently



We'll cover parallel streams later



See docs.oracle.com/javase/tutorial/collections/stream/parallelism.html

End of Overview of Java 8 Streams (Part 1)

Overview of Java 8 Streams (Part 2)

Douglas C. Schmidt

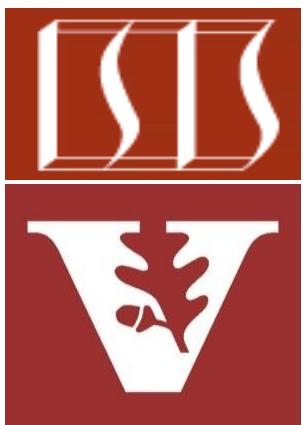
d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

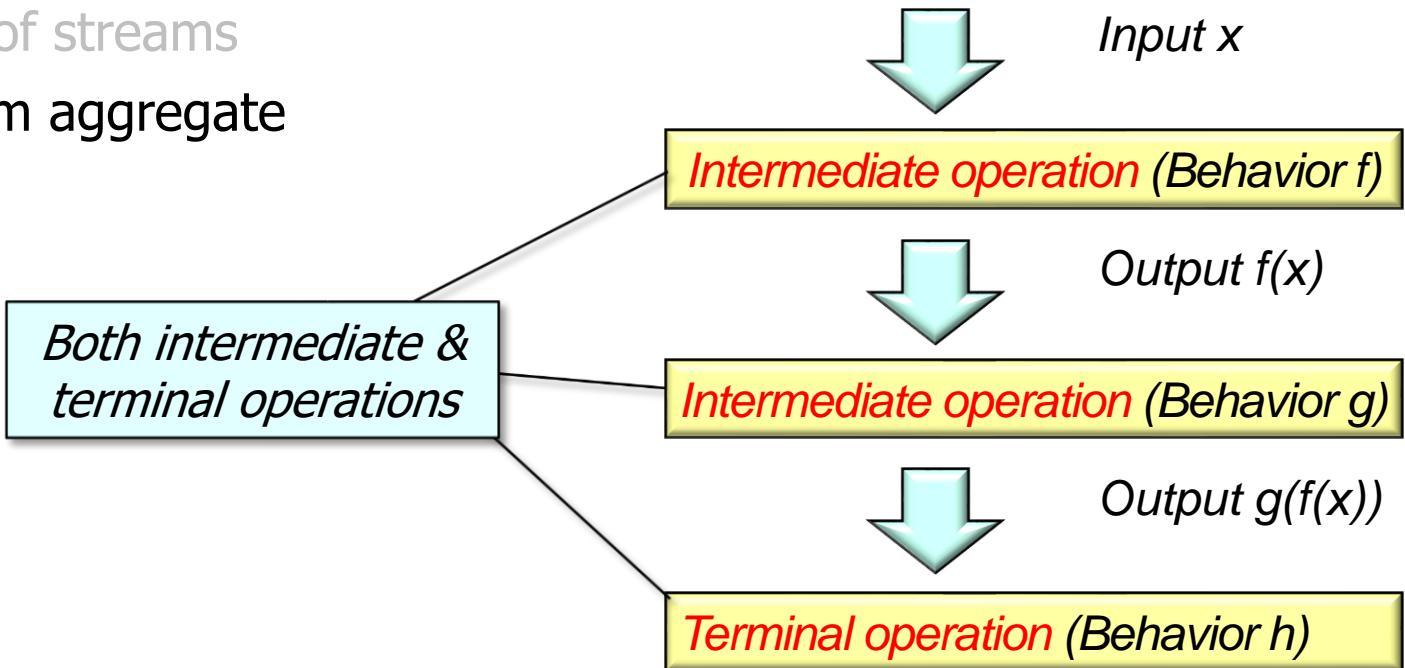
Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA



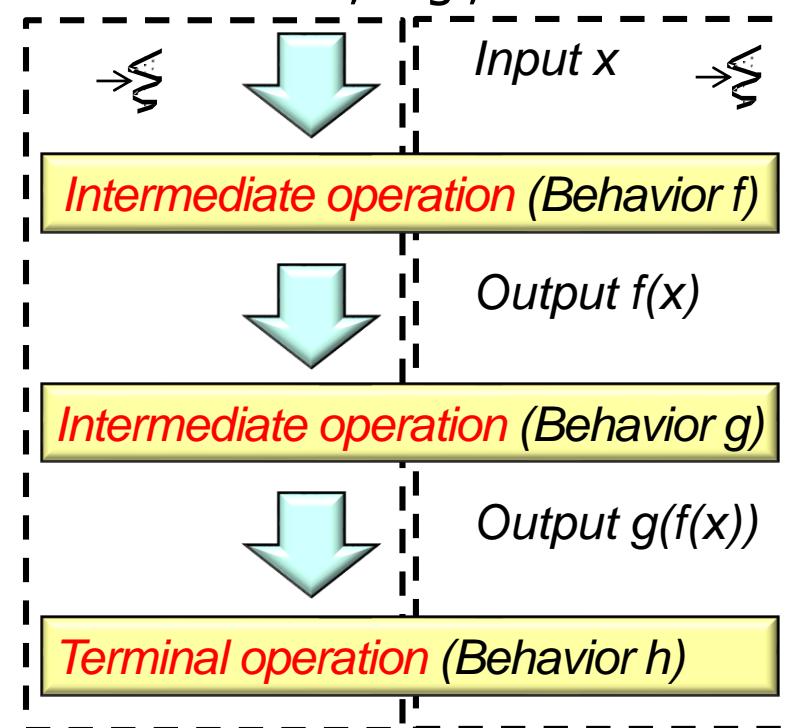
Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of Java 8 streams, e.g.,
 - Fundamentals of streams
 - Common stream aggregate operations



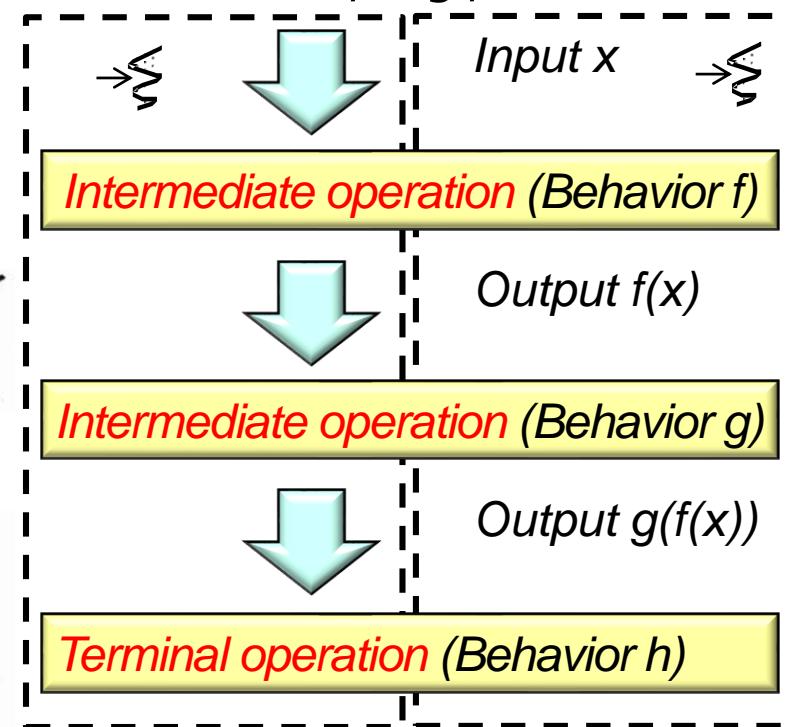
Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of Java 8 streams, e.g.,
 - Fundamentals of streams
 - Common stream aggregate operations
 - These operations apply to both sequential & parallel streams



Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of Java 8 streams, e.g.,
 - Fundamentals of streams
 - Common stream aggregate operations
 - These operations apply to both sequential & parallel streams

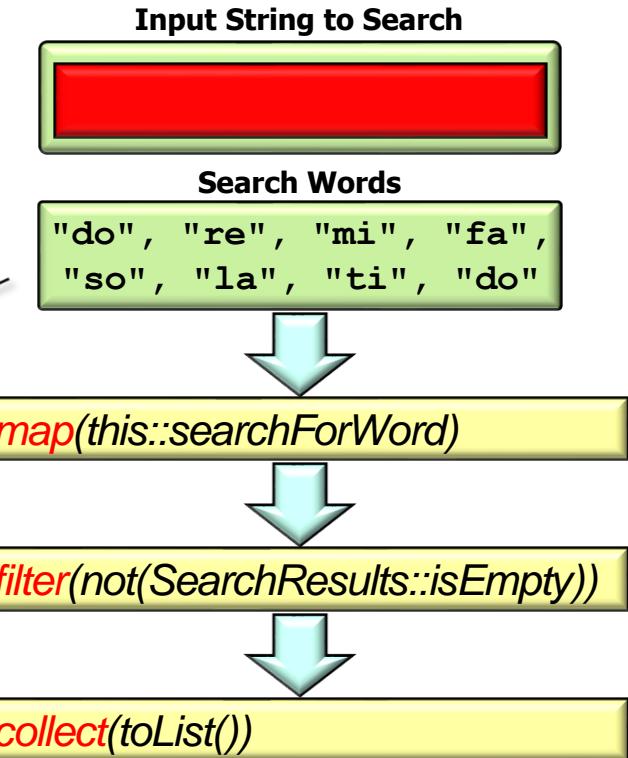


Being a good streams programmer makes you a better parallel streams programmer

Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of Java 8 streams, e.g.,
 - Fundamentals of streams
 - Common stream aggregate operations
 - These operations apply to both sequential & parallel streams

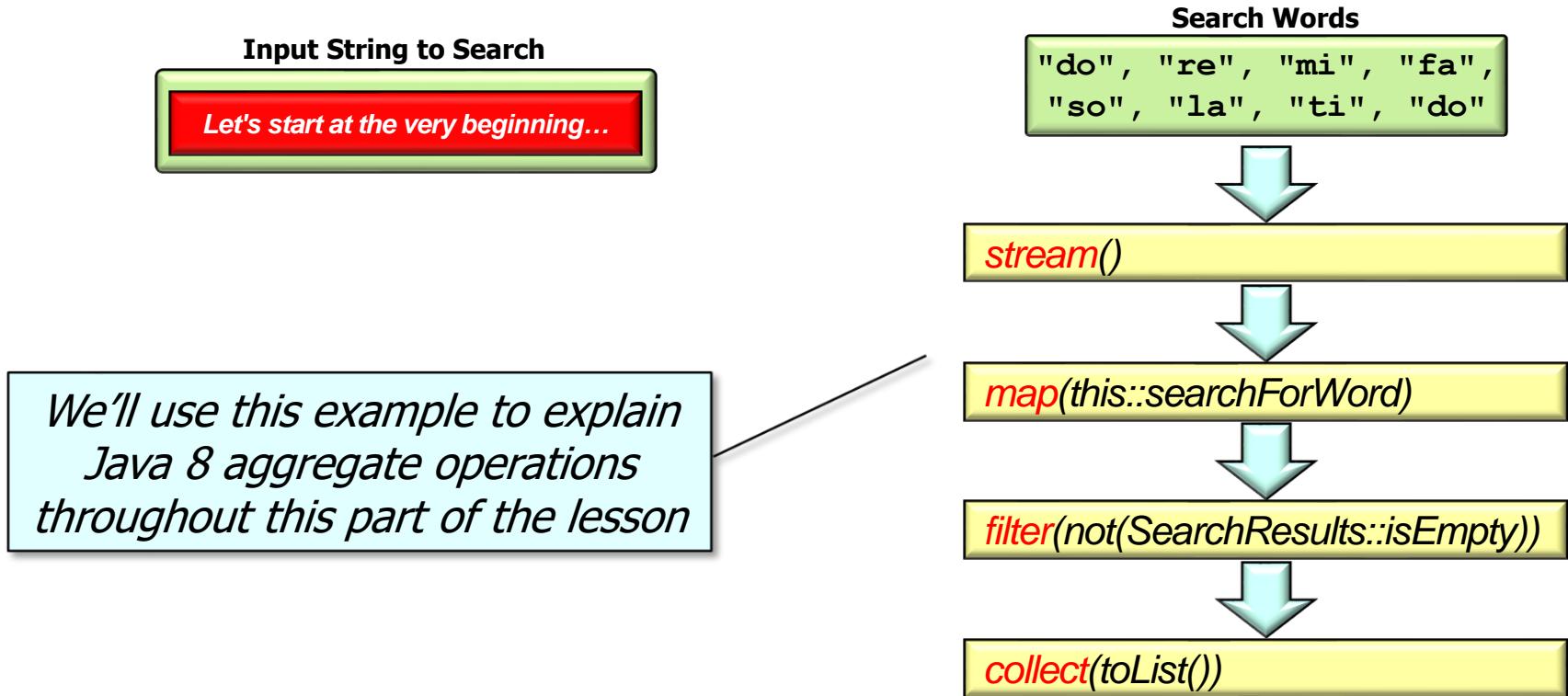
We'll use a simple sequential stream example to explain common Java 8 aggregate operations



Overview of SimpleSearch Stream Example

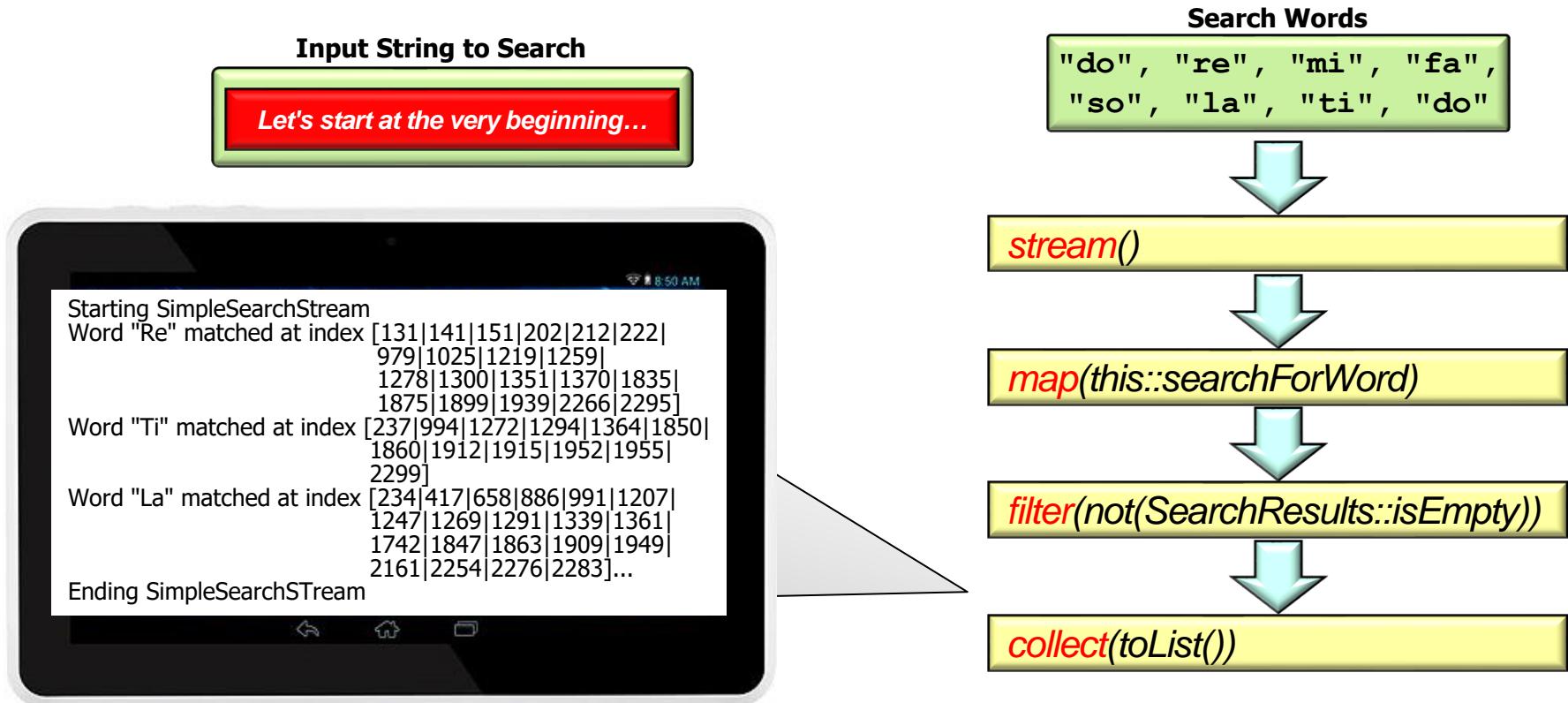
Overview of SimpleSearchStream Example

- This example finds words in an input string



Overview of SimpleSearchStream Example

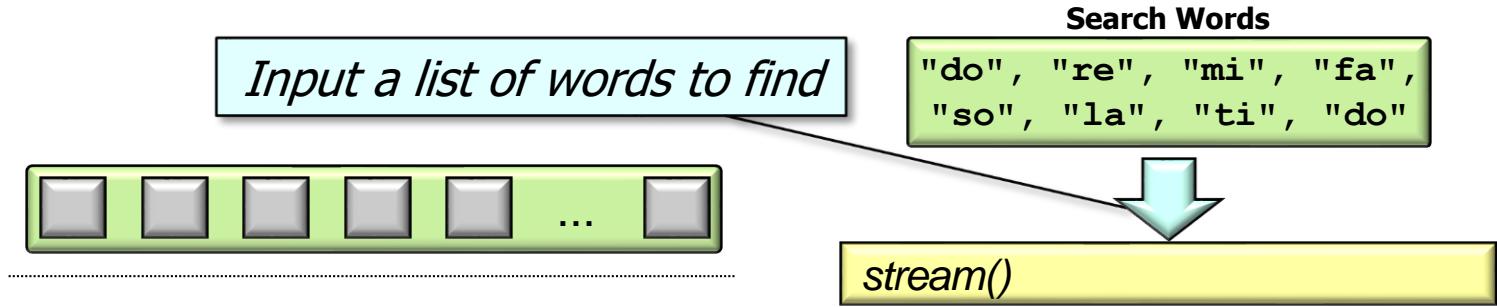
- This example finds words in an input string



Overview of SimpleSearchStream Example

- This example finds words in an input string

List
<String>



Overview of SimpleSearchStream Example

- This example finds words in an input string

List
<String>



Search Words

```
"do", "re", "mi", "fa",
"so", "la", "ti", "do"
```

stream()

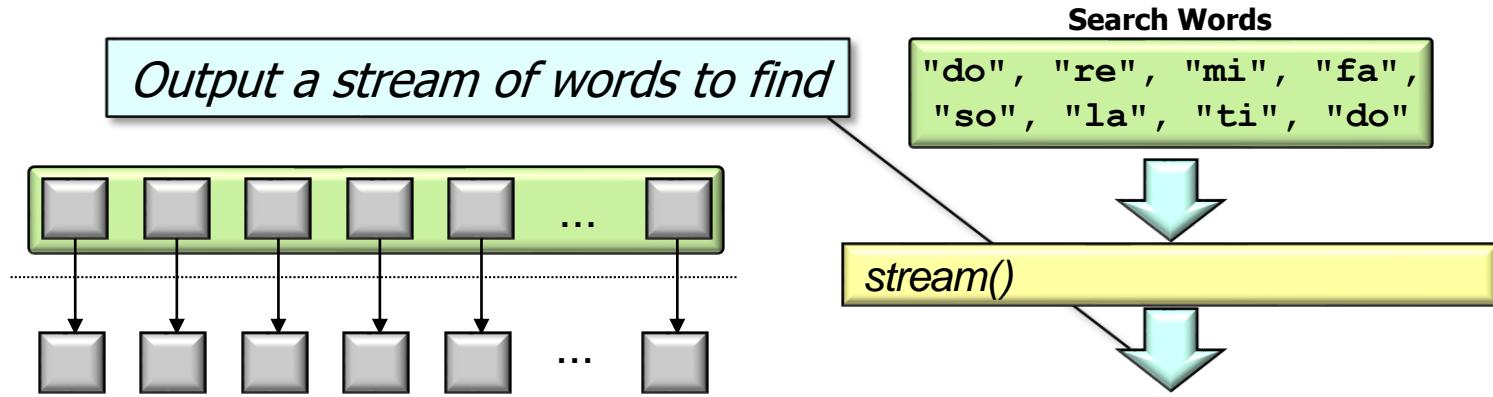
Convert collection to a (sequential) stream

Overview of SimpleSearchStream Example

- This example finds words in an input string

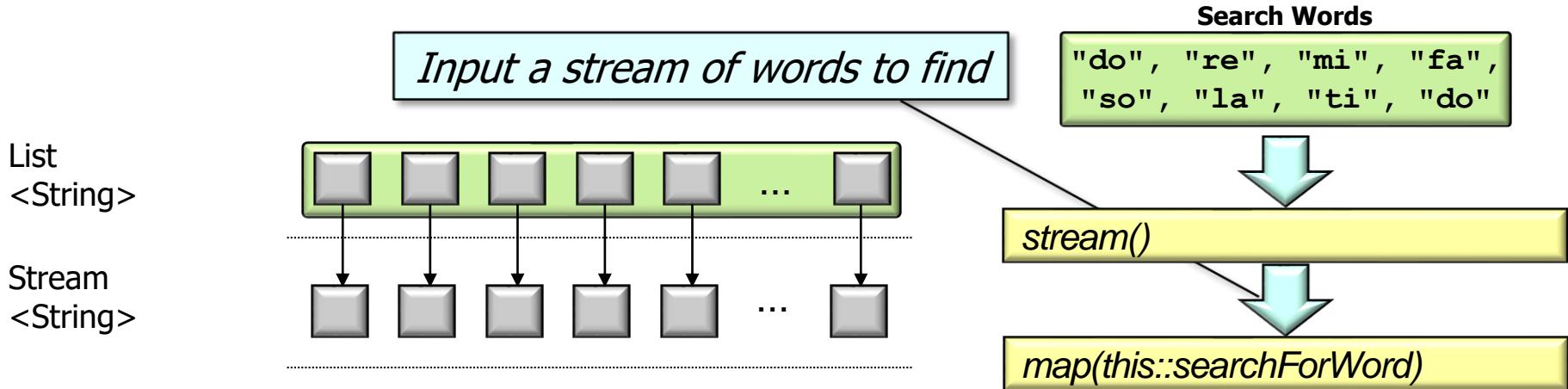
List
<String>

Stream
<String>



Overview of SimpleSearchStream Example

- This example finds words in an input string

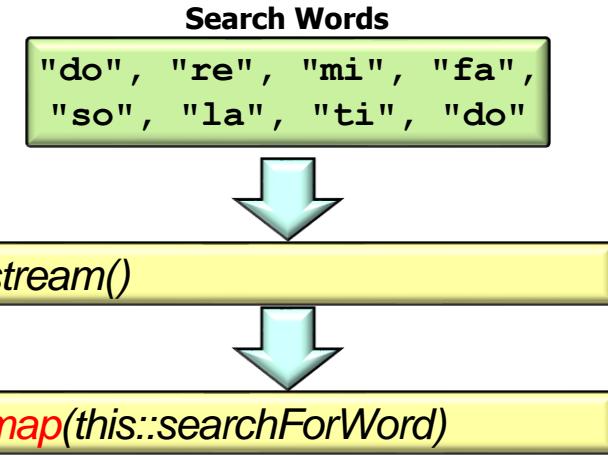
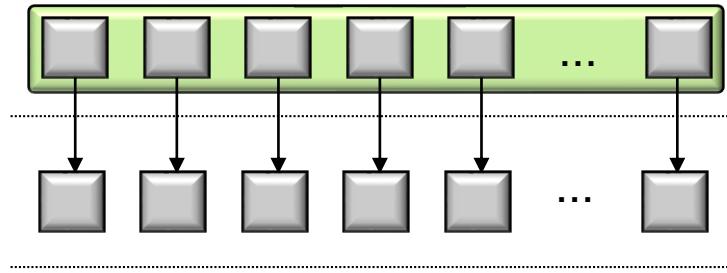


Overview of SimpleSearchStream Example

- This example finds words in an input string

List
<String>

Stream
<String>



Search for the word in the input string

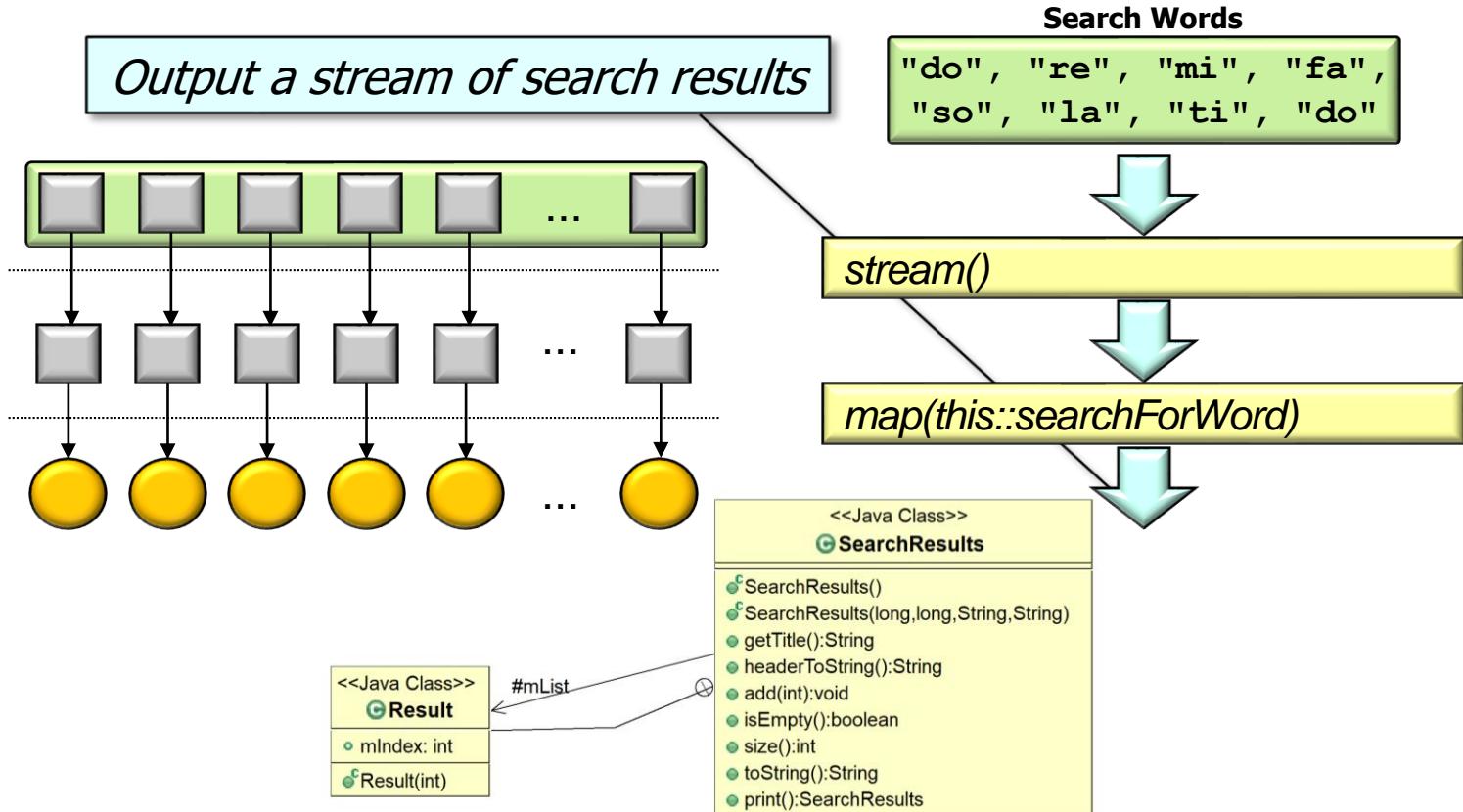
Overview of SimpleSearchStream Example

- This example finds words in an input string

List
<String>

Stream
<String>

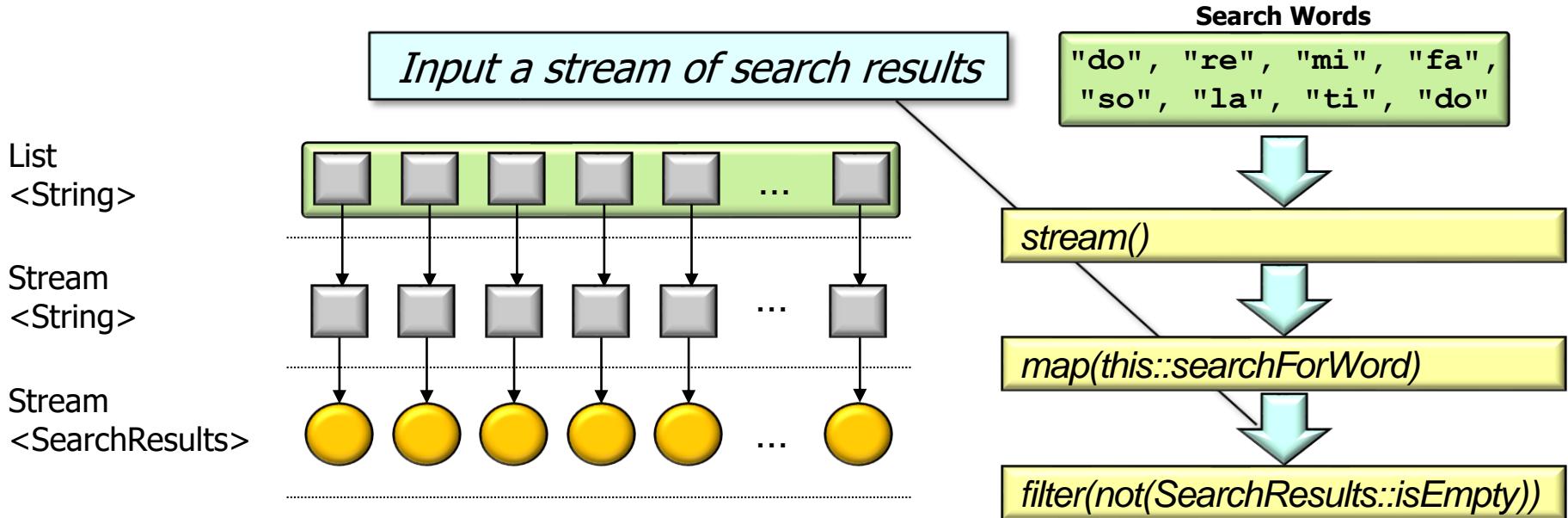
Stream
<SearchResults>



SearchResults stores # of times a word appeared in the input string

Overview of SimpleSearchStream Example

- This example finds words in an input string



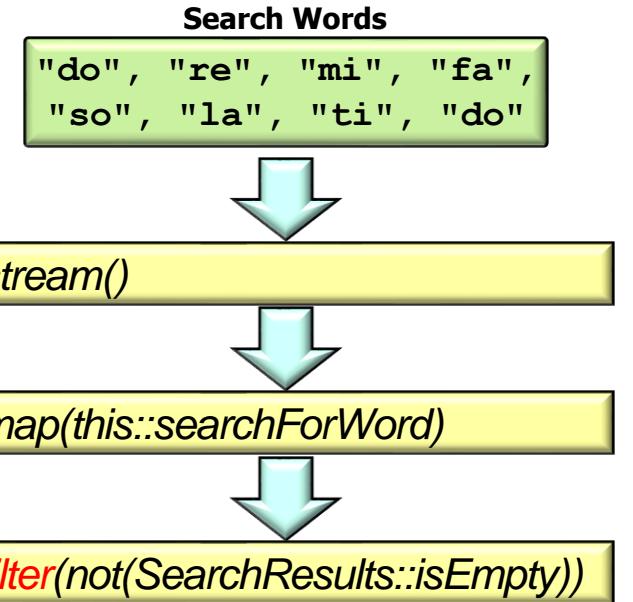
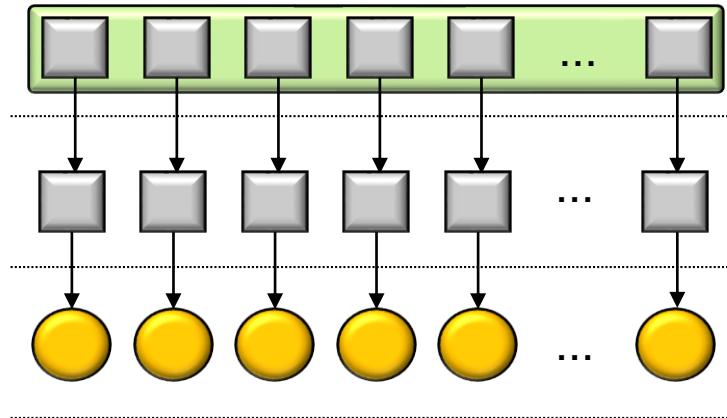
Overview of SimpleSearchStream Example

- This example finds words in an input string

List
<String>

Stream
<String>

Stream
<SearchResults>



Remove empty search results from the stream

Overview of SimpleSearchStream Example

- This example finds words in an input string

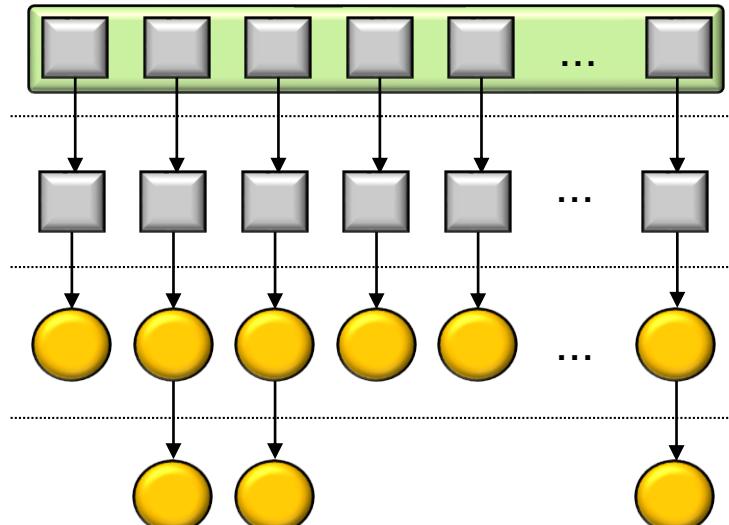
Output a stream of non-empty search results

List
<String>

Stream
<String>

Stream
<SearchResults>

Stream
<SearchResults>



Search Words
"do", "re", "mi", "fa",
"so", "la", "ti", "do"

stream()

map(this::searchForWord)

filter(not(SearchResults::isEmpty))

Overview of SimpleSearchStream Example

- This example finds words in an input string

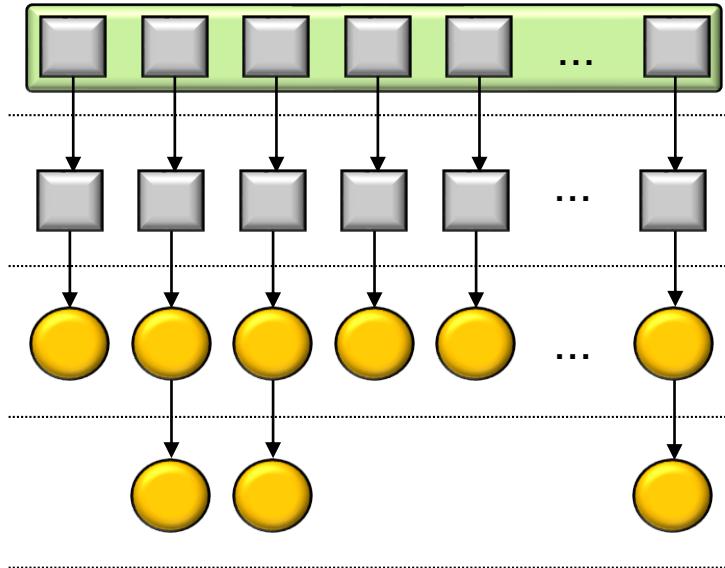
List
<String>

Stream
<String>

Stream
<SearchResults>

Stream
<SearchResults>

Input a stream of non-empty search results



Search Words
"do", "re", "mi", "fa",
"so", "la", "ti", "do"

stream()

map(this::searchForWord)

filter(not(SearchResults::isEmpty))

collect(toList())

Overview of SimpleSearchStream Example

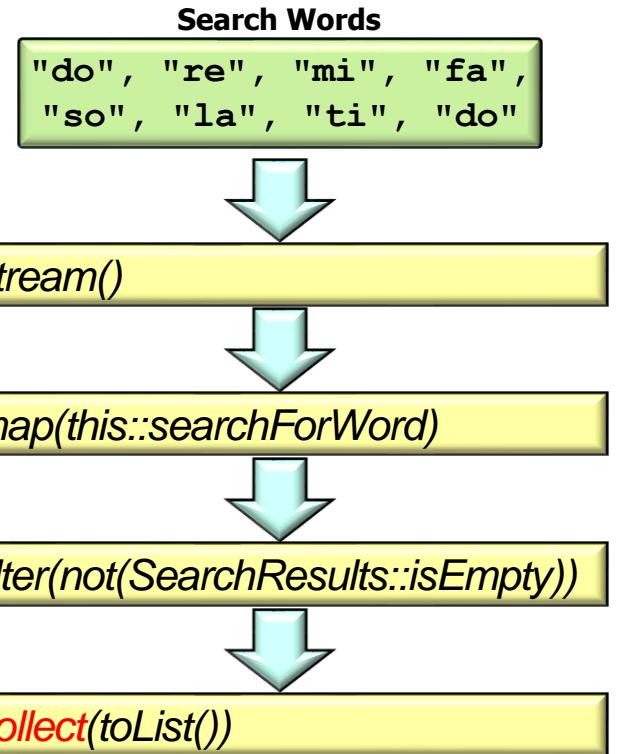
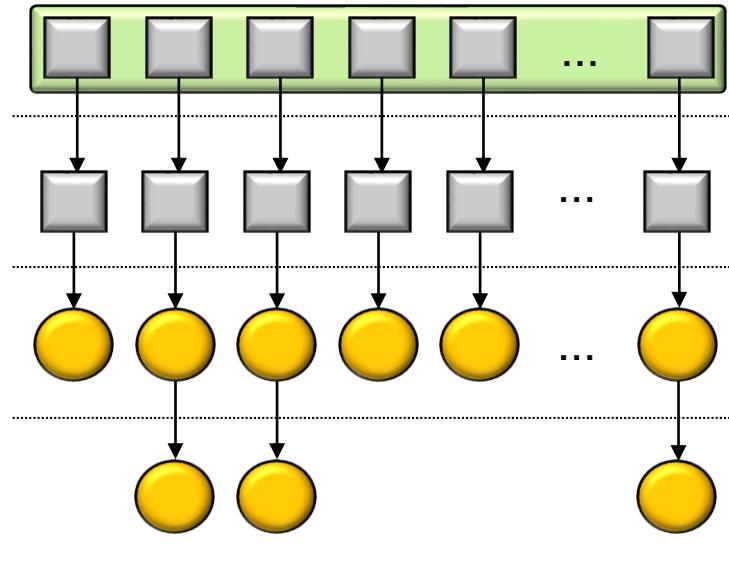
- This example finds words in an input string

List
<String>

Stream
<String>

Stream
<SearchResults>

Stream
<SearchResults>



Trigger intermediate operation processing

Overview of SimpleSearchStream Example

- This example finds words in an input string

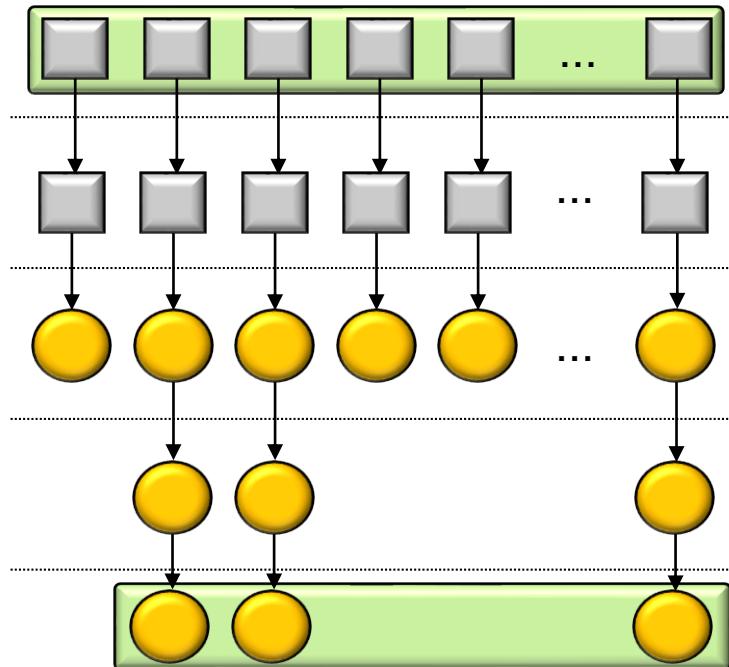
List
<String>

Stream
<String>

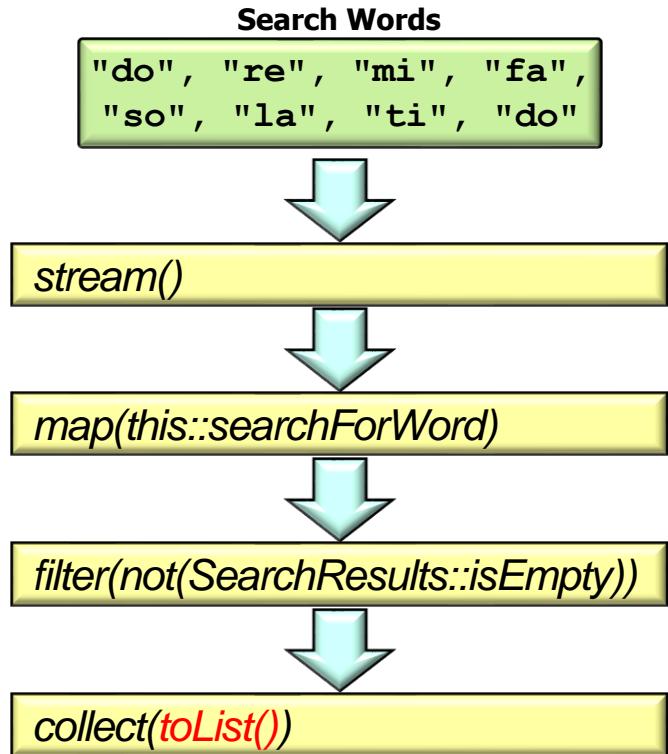
Stream
<SearchResults>

Stream
<SearchResults>

List
<SearchResults>



Return a list of search results



Overview of SimpleSearchStream Example

- The “physical” processing of a stream differs from the “logical” model
 - i.e., each element is “pulled” from the source through each aggregate operation

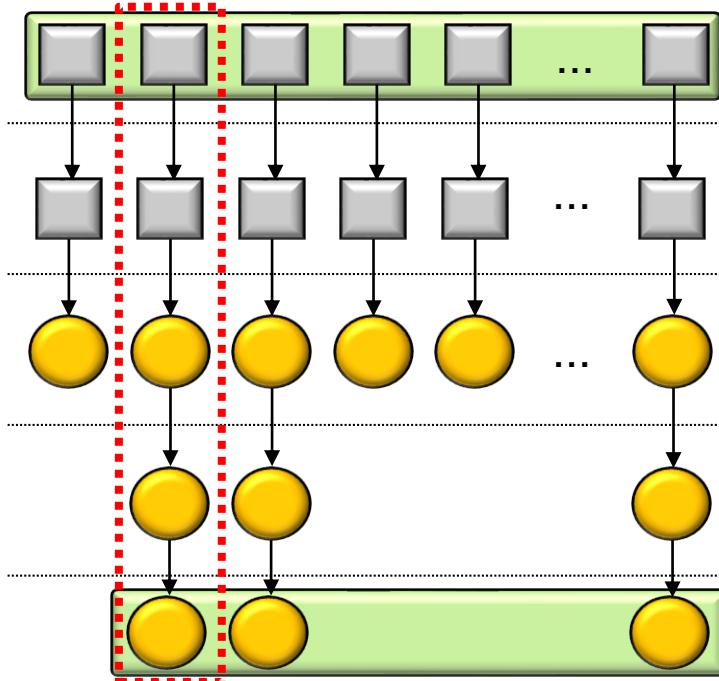
List
<String>

Stream
<String>

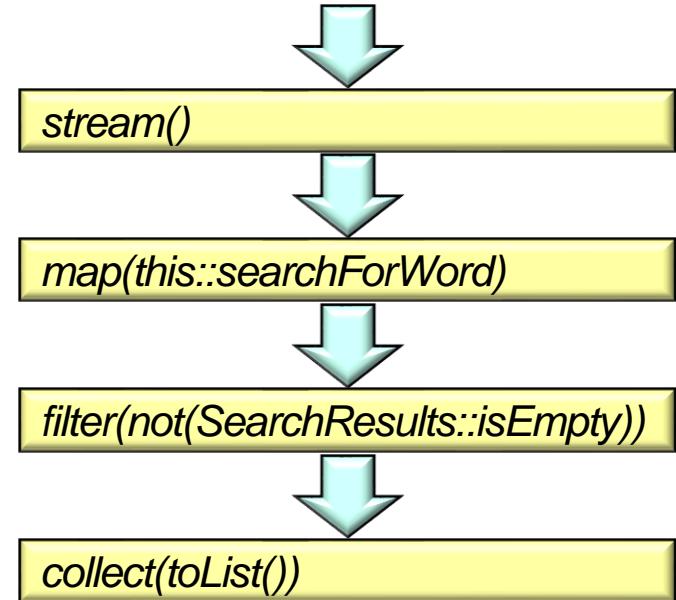
Stream
<SearchResults>

Stream
<SearchResults>

List
<SearchResults>



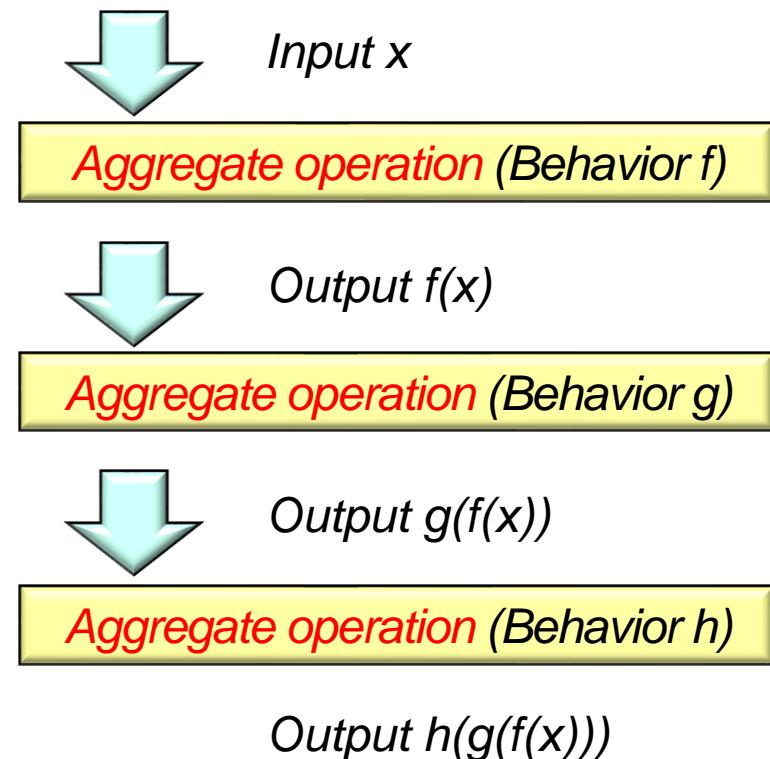
Search Words
"do", "re", "mi", "fa",
"so", "la", "ti", "do"



Overview of Common Stream Aggregate Operations

Overview of Common Stream Aggregate Operations

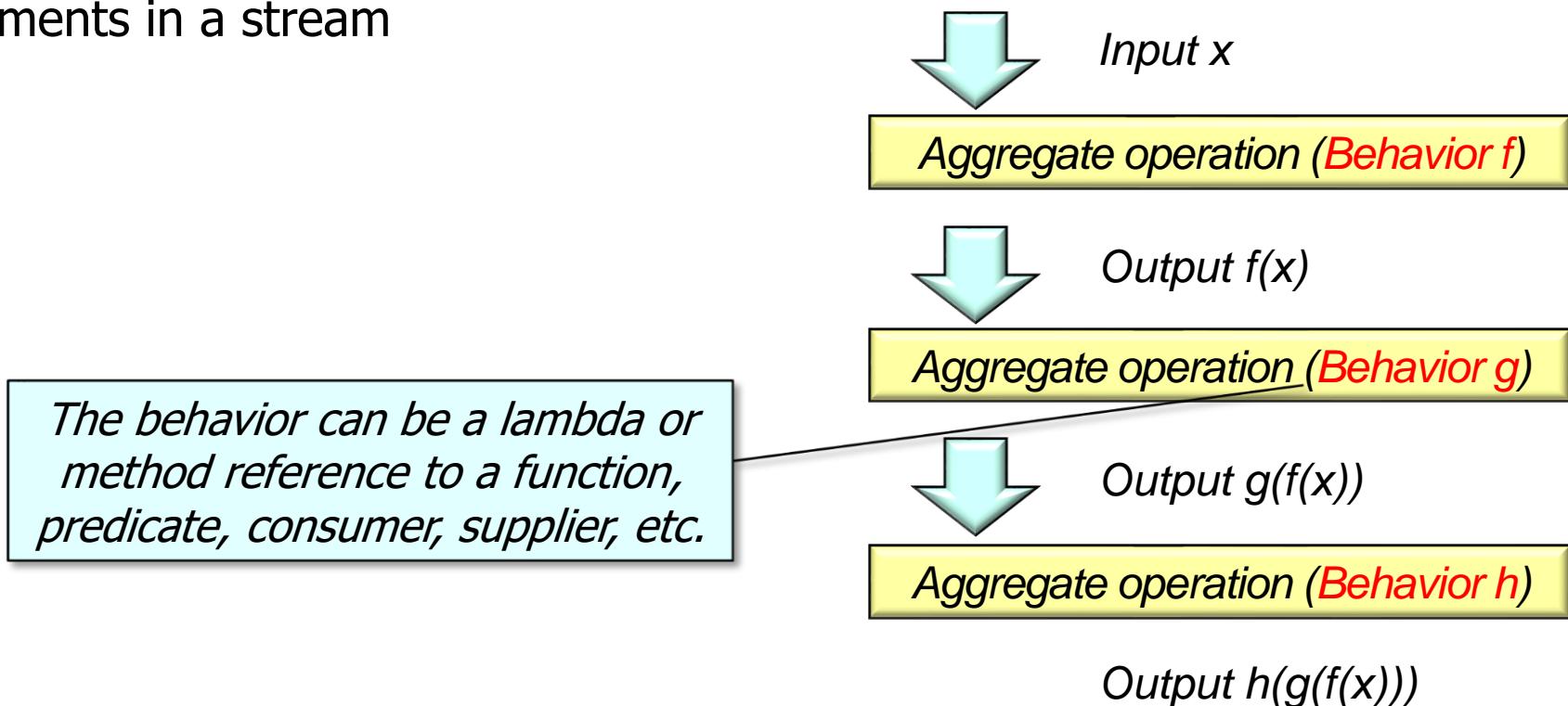
- An aggregate operation is a higher-order function that applies a “behavior” on elements in a stream



See en.wikipedia.org/wiki/Higher-order_function

Overview of Common Stream Aggregate Operations

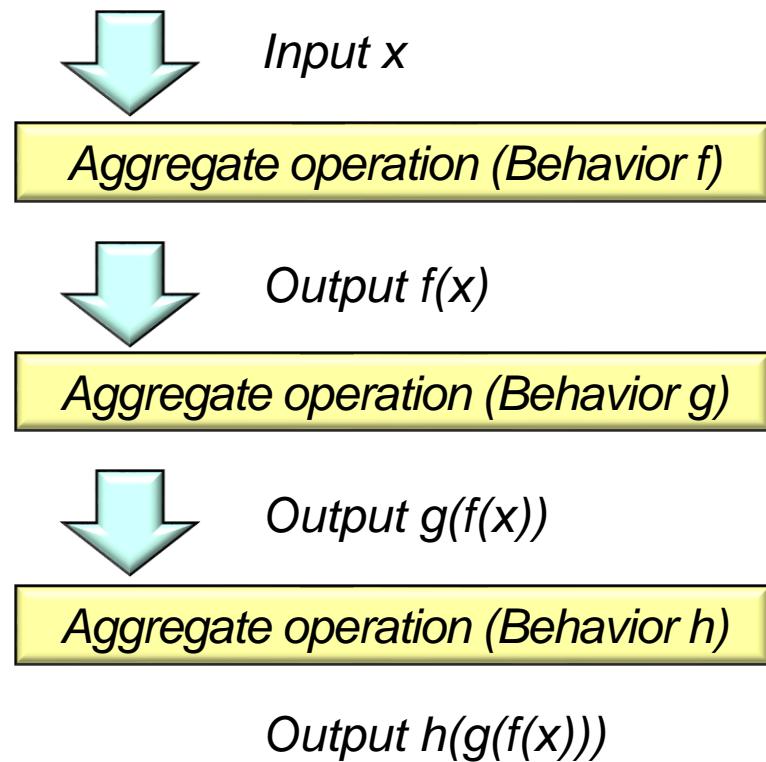
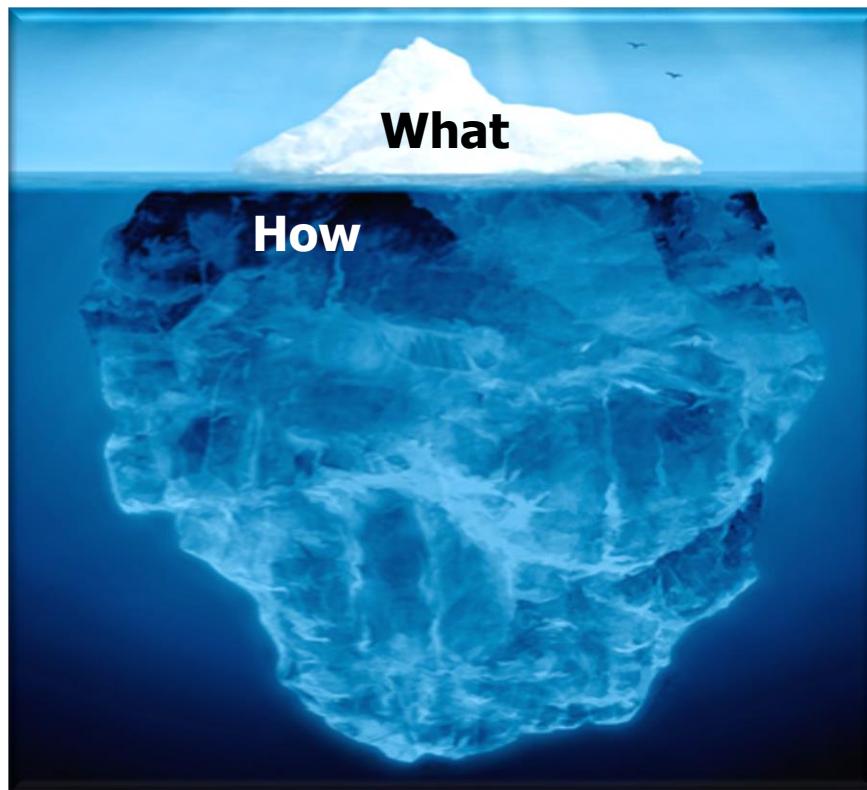
- An aggregate operation is a higher-order function that applies a “behavior” on elements in a stream



See en.wikipedia.org/wiki/Higher-order_function

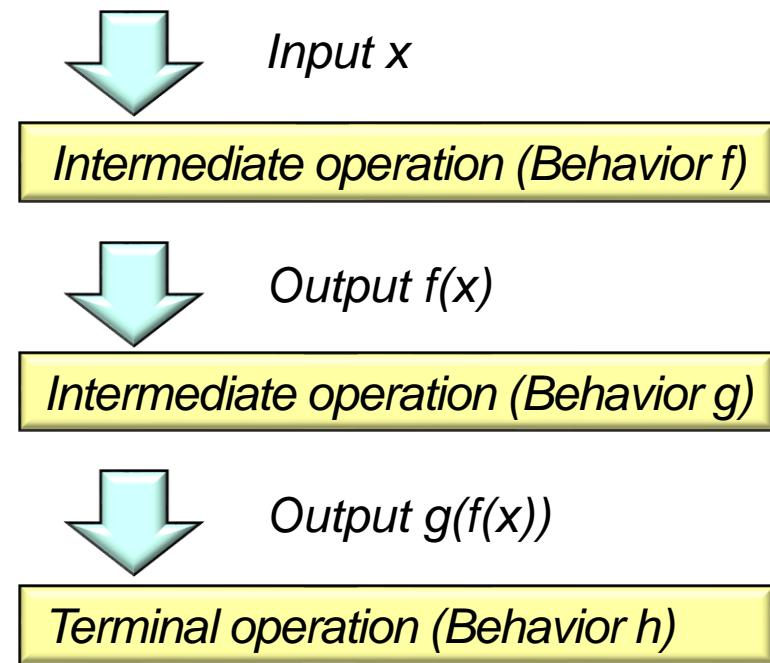
Overview of Common Stream Aggregate Operations

- Aggregate operations focus on “what” (declarative), *not* “how” (imperative)



Overview of Common Stream Aggregate Operations

- There are two types of aggregate operations

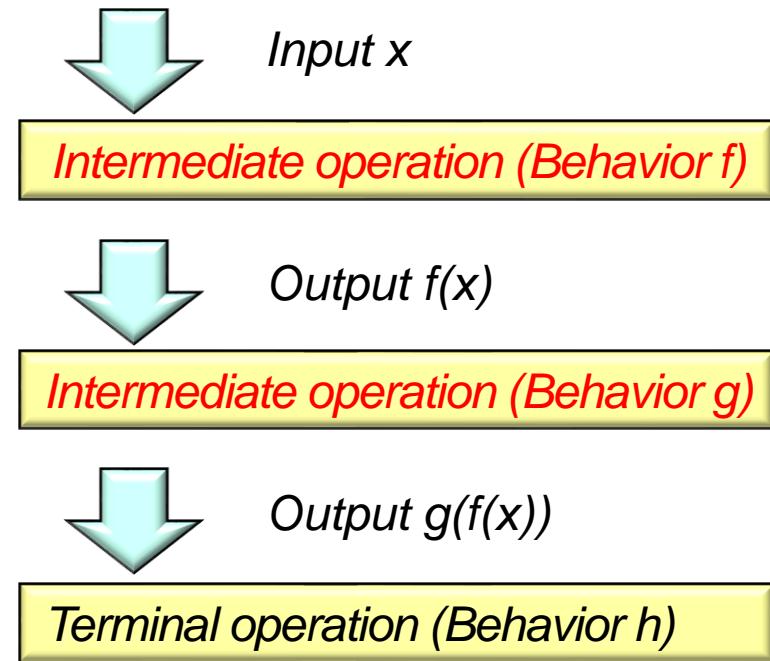


Overview of Common Stream Aggregate Operations

- There are two types of aggregate operations

- **Intermediate operations**

- Process elements in their input stream & yield an output stream
 - e.g., filter(), map(), flatMap(), etc.



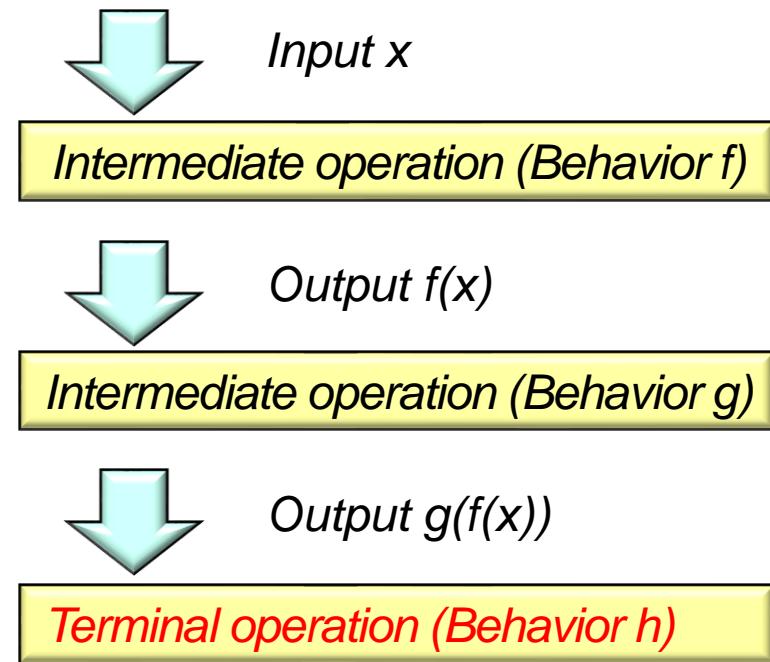
Overview of Common Stream Aggregate Operations

- There are two types of aggregate operations

- **Intermediate operations**

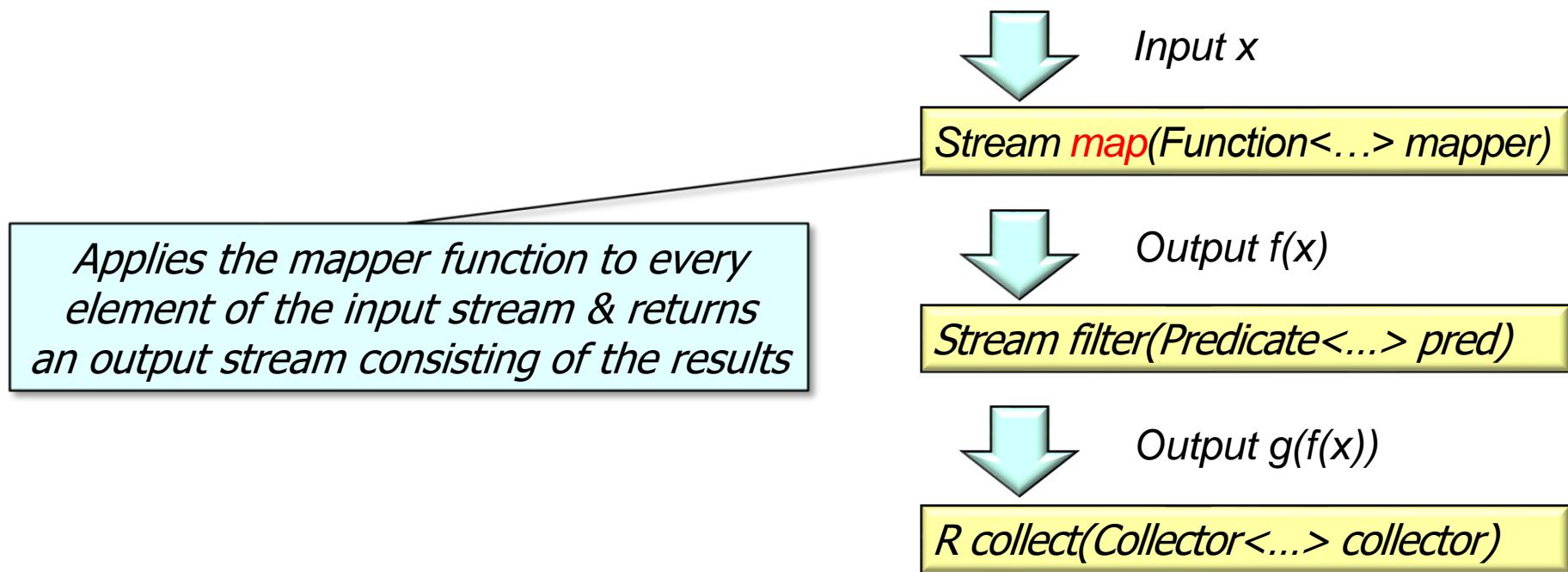
- **Terminal operations**

- Trigger intermediate operations & produce a non-stream result
 - e.g., `forEach()`, `reduce()`, `collect()`, etc.



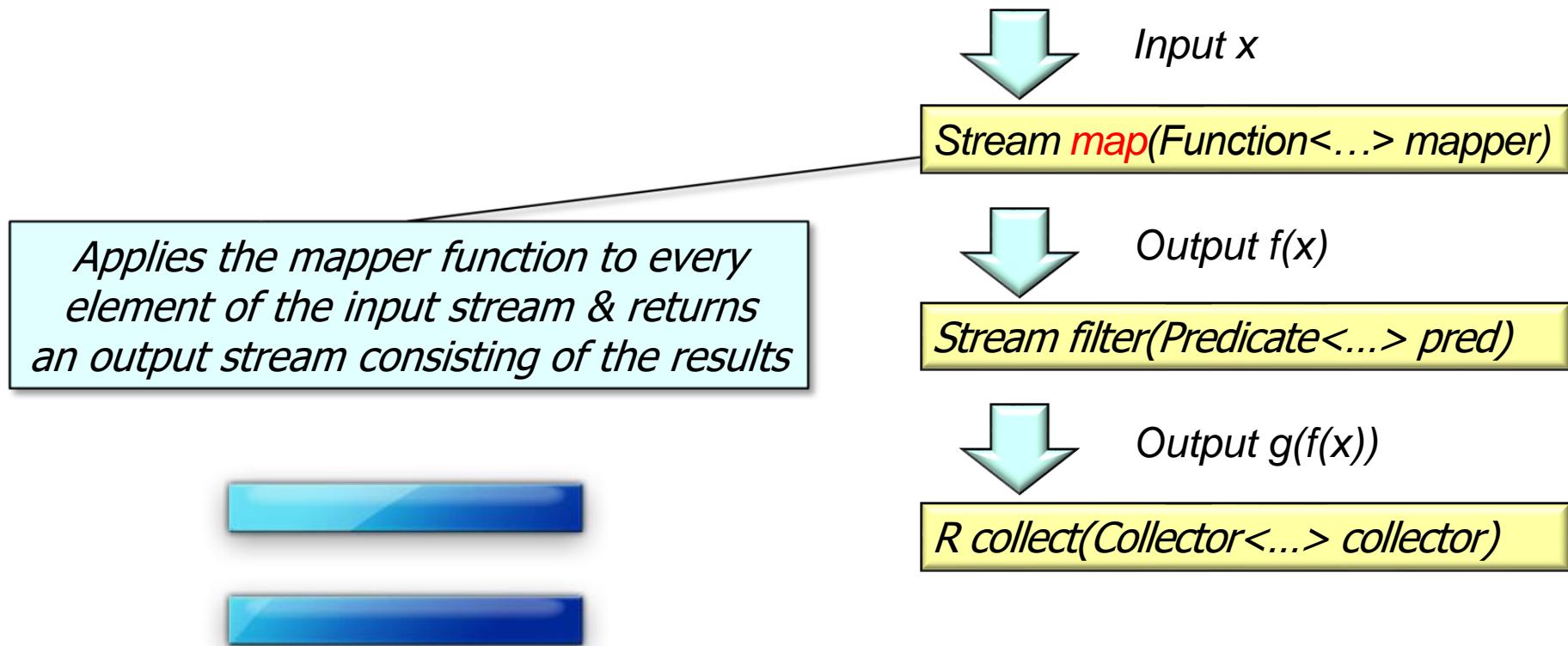
Overview of Common Stream Aggregate Operations

- Overview of the map() intermediate operation



Overview of Common Stream Aggregate Operations

- Overview of the map() intermediate operation



The # of output stream elements matches the # of input stream elements

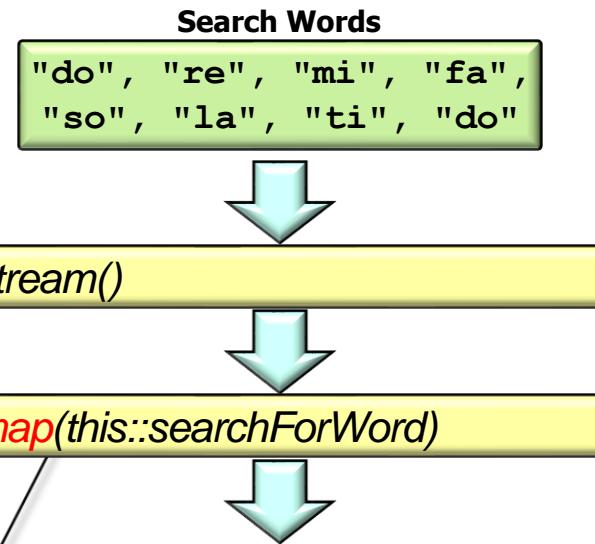
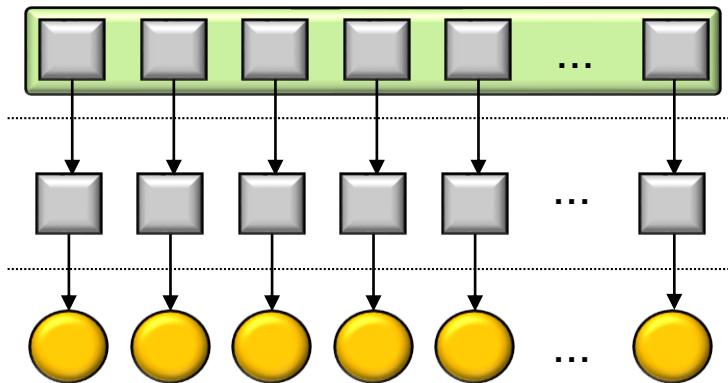
Overview of Common Stream Aggregate Operations

- Overview of the map() intermediate operation

List
<String>

Stream
<String>

Stream
<SearchResults>



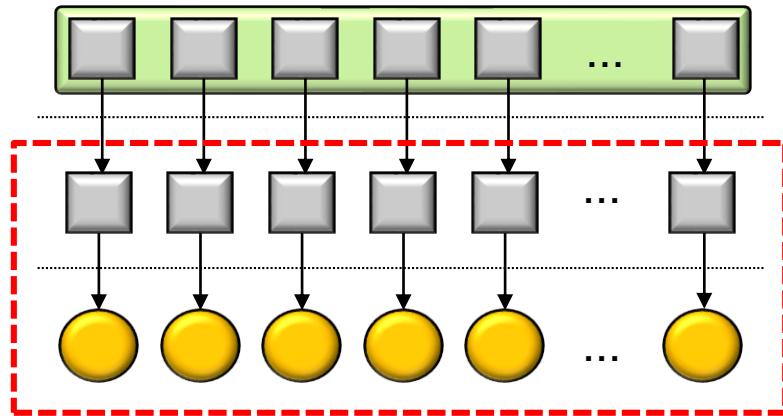
Overview of Common Stream Aggregate Operations

- Overview of the map() intermediate operation

List
<String>

Stream
<String>

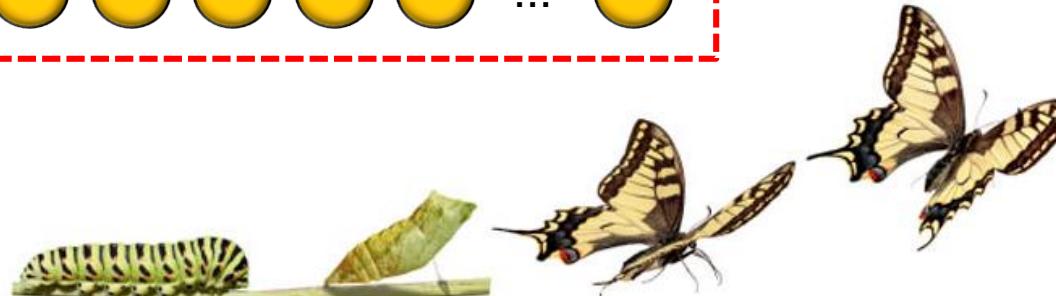
Stream
<SearchResults>



Search Words
"do", "re", "mi", "fa",
"so", "la", "ti", "do"

stream()

map(this::searchForWord)

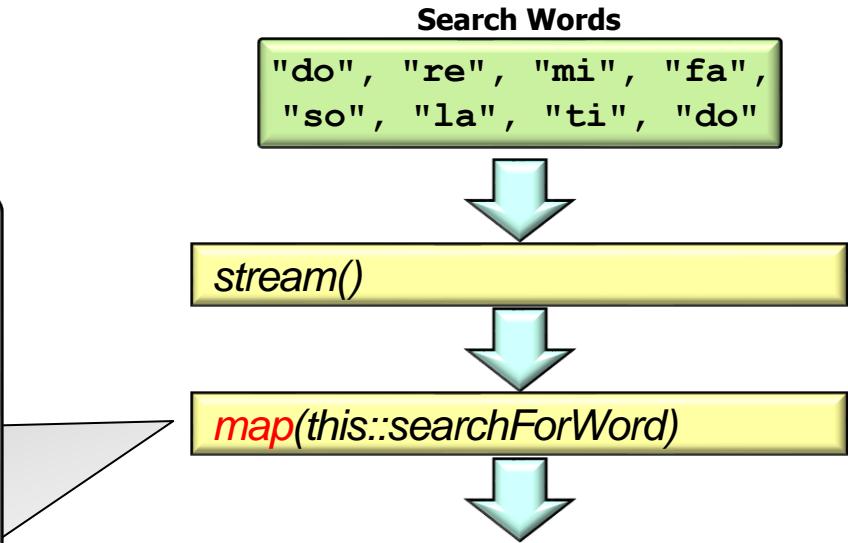


map() may transform the type of elements it processes

Overview of Common Stream Aggregate Operations

- Overview of the map() intermediate operation

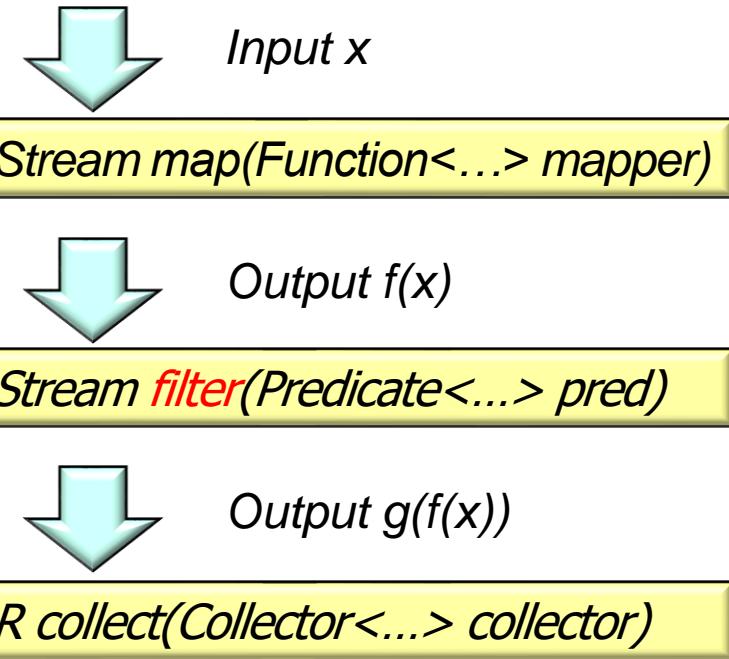
```
List<SearchResults> results =  
    wordsToFind  
        .stream()  
        .map(this::searchForWord)  
        .filter(not  
            (SearchResults::isEmpty))  
        .collect(toList());
```



Overview of Common Stream Aggregate Operations

- Overview of the filter() intermediate operation

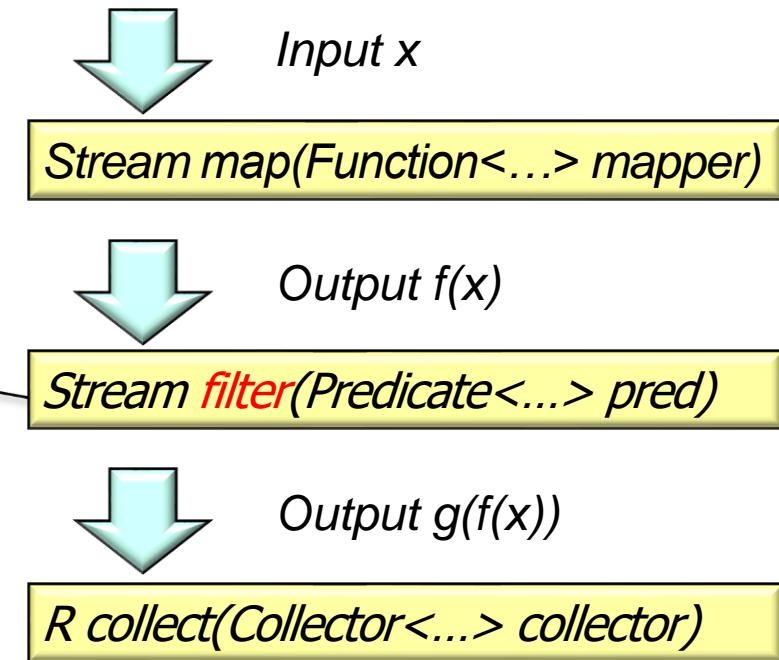
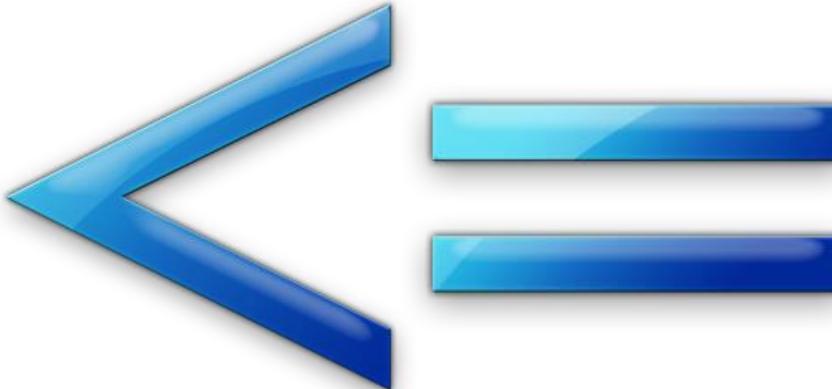
Tests the given predicate against each element of the input stream & returns an output stream consisting only of the elements that match the predicate



Overview of Common Stream Aggregate Operations

- Overview of the filter() intermediate operation

Tests the given predicate against each element of the input stream & returns an output stream consisting only of the elements that match the predicate



The # of output stream elements may be less than the # of input stream elements

Overview of Common Stream Aggregate Operations

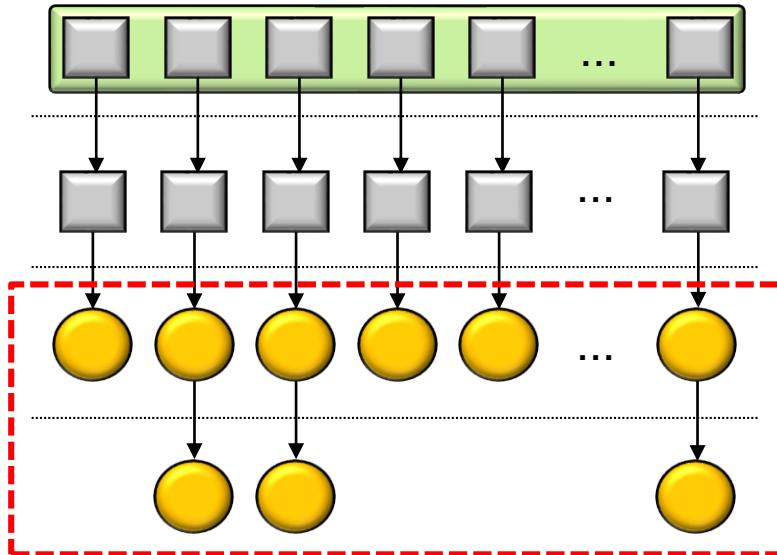
- Overview of the filter() intermediate operation

List
<String>

Stream
<String>

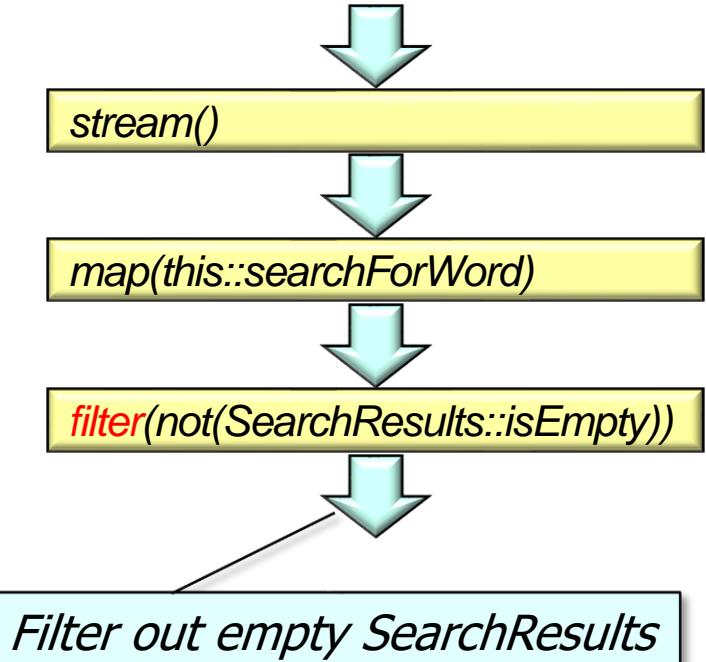
Stream
<SearchResults>

Stream
<SearchResults>



Search Words

```
"do", "re", "mi", "fa",
"so", "la", "ti", "do"
```

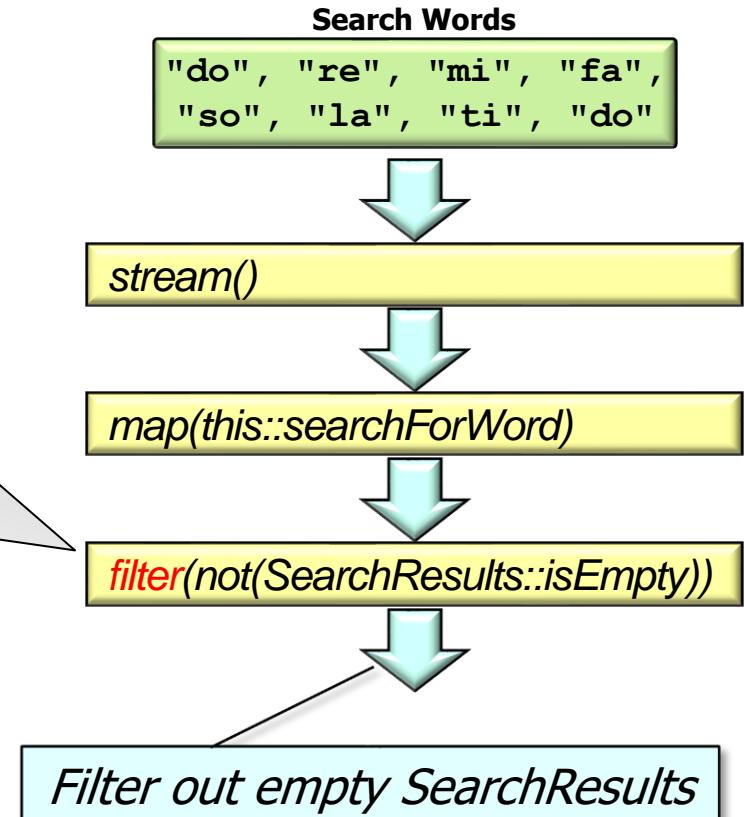


The filter() intermediate operation *can't* change the type of elements it processes

Overview of Common Stream Aggregate Operations

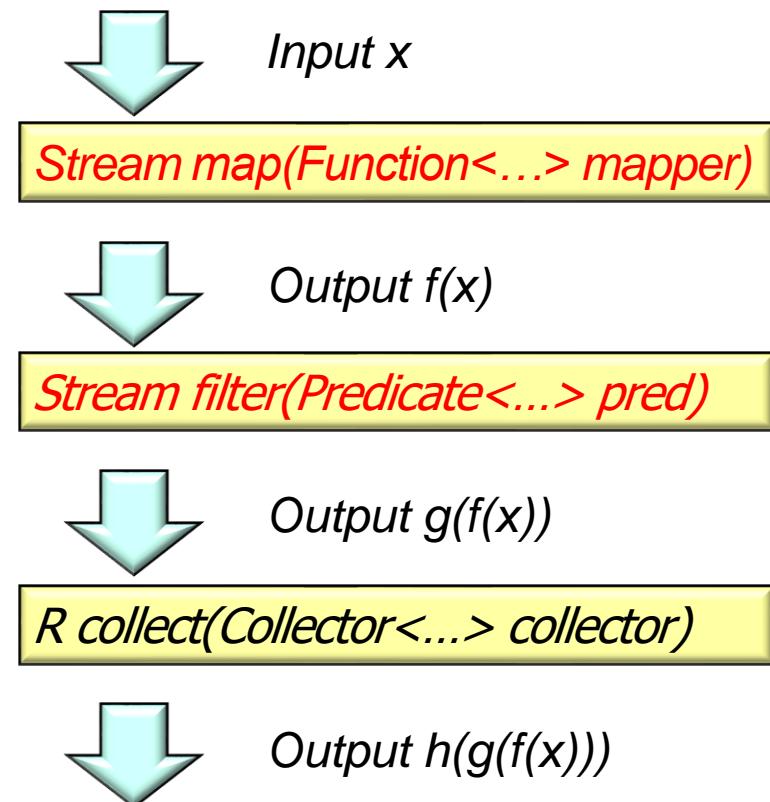
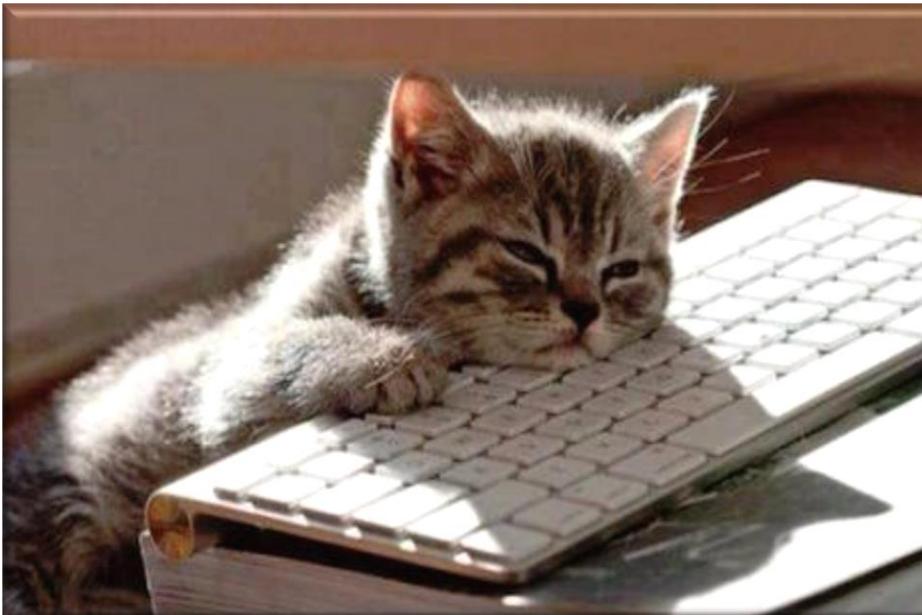
- Overview of the filter() intermediate operation

```
List<SearchResults> results =  
    wordsToFind  
        .stream()  
        .map(this::searchForWord)  
        .filter(not  
            (SearchResults::isEmpty))  
        .collect(toList());
```



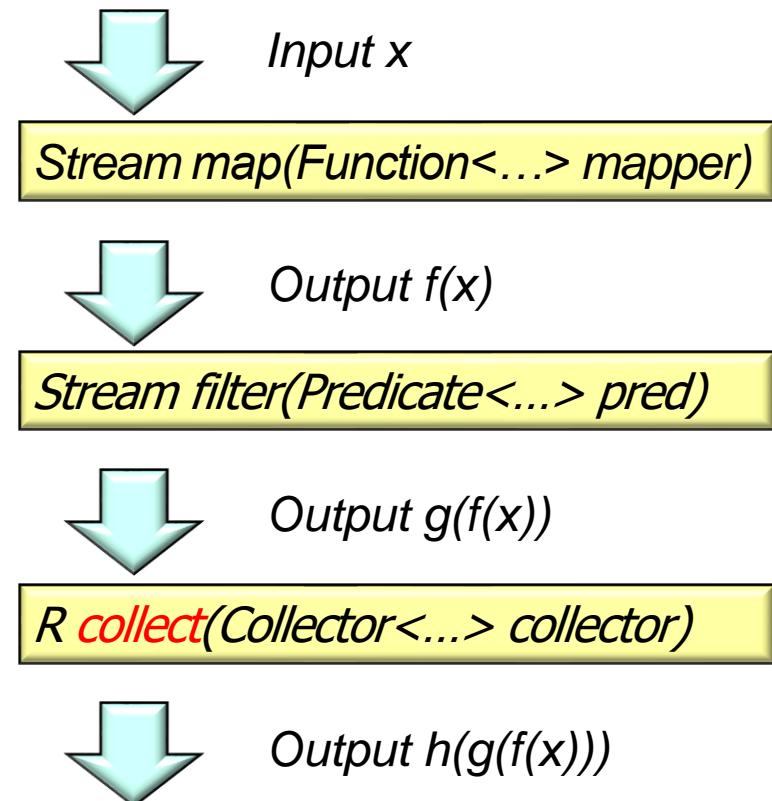
Overview of Common Stream Aggregate Operations

- Intermediate operations are “lazy” & don’t run until a terminal operator is reached



Overview of Common Stream Aggregate Operations

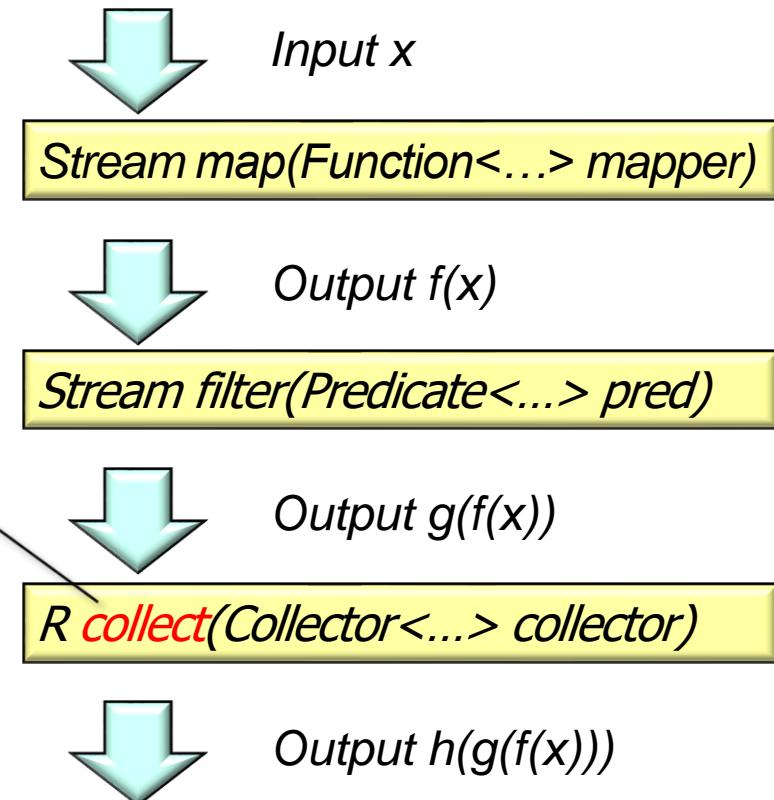
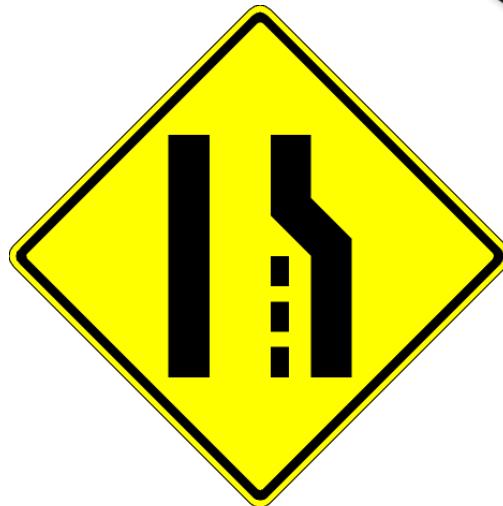
- A terminal operation triggers intermediate operation processing



Overview of Common Stream Aggregate Operations

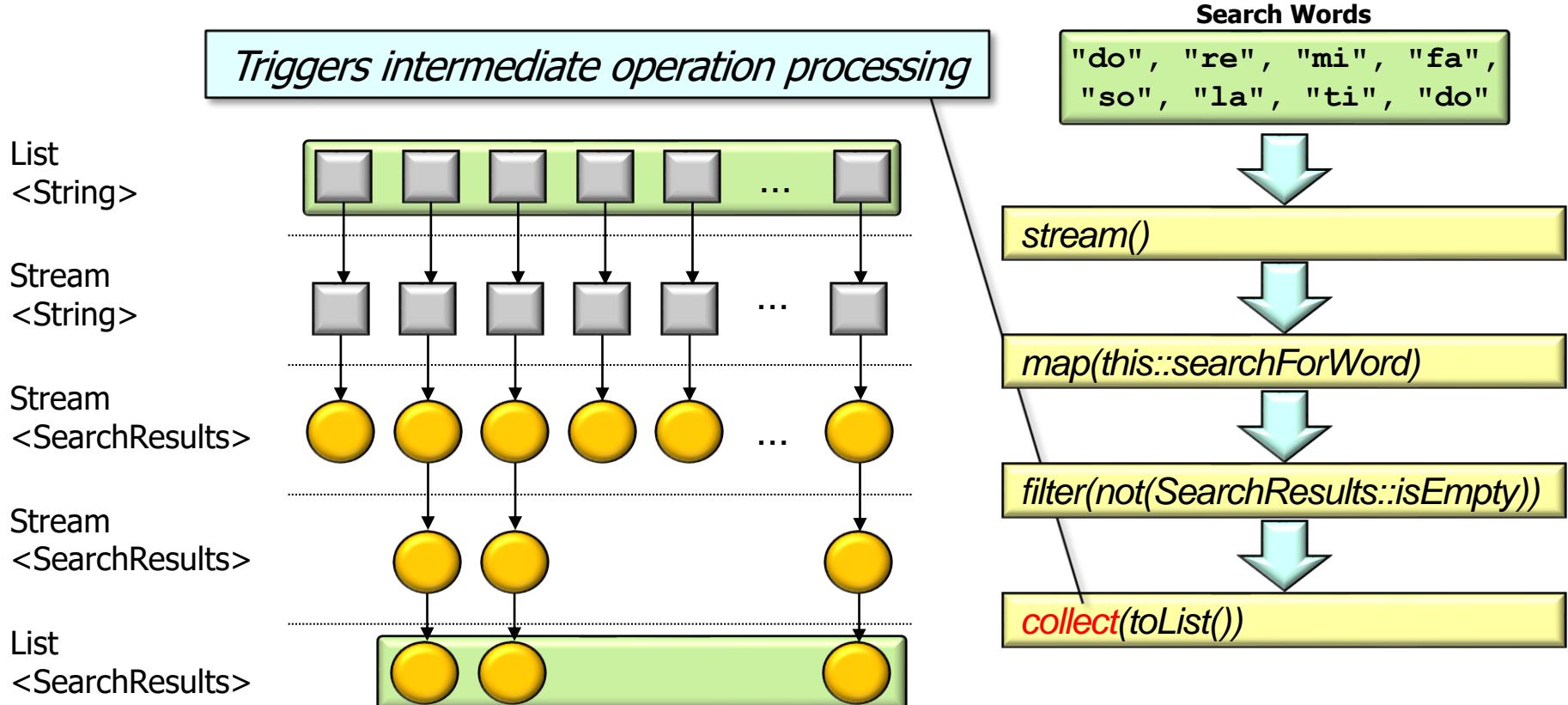
- Overview of the collect() terminal operation

This terminal operation uses a collector to perform a reduction on the elements of its input stream & returns the results of the reduction



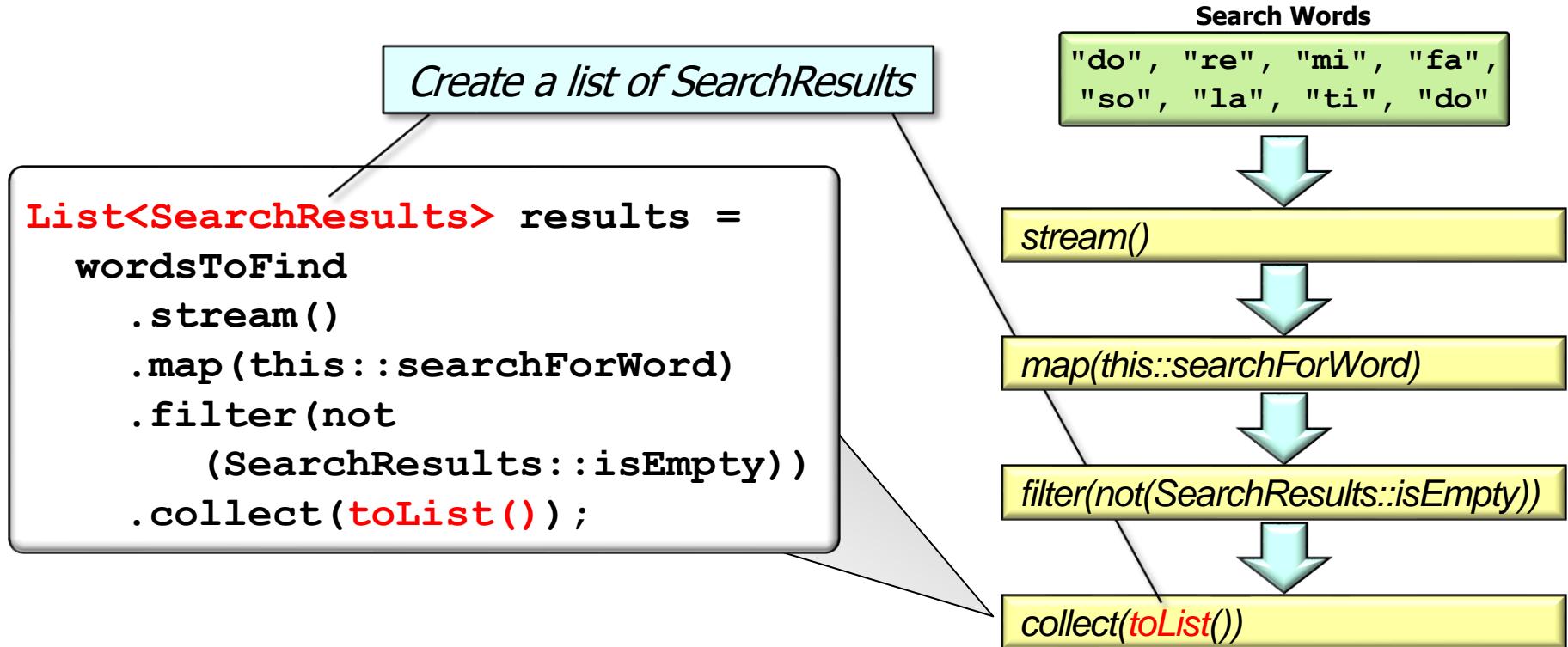
Overview of Common Stream Aggregate Operations

- Overview of the collect() terminal operation



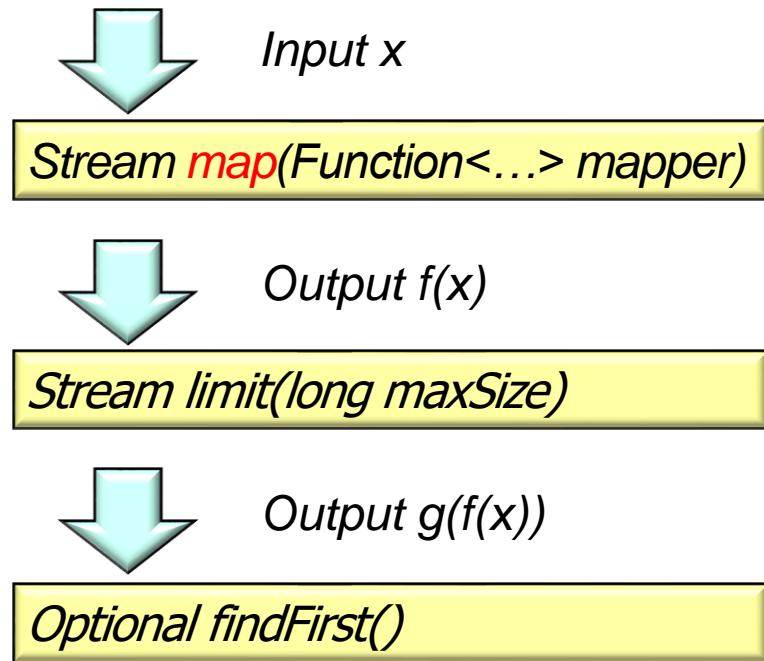
Overview of Common Stream Aggregate Operations

- Overview of the collect() terminal operation



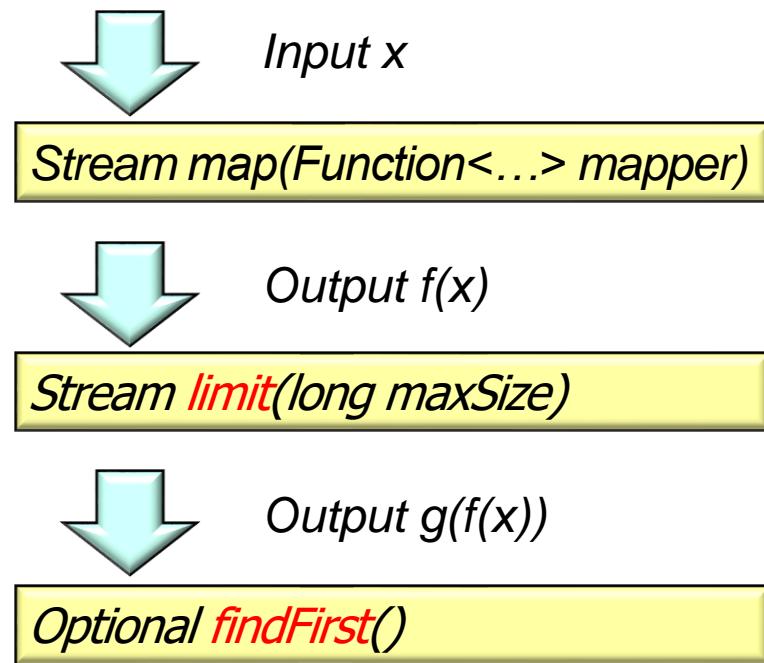
Overview of Common Stream Aggregate Operations

- An aggregate operation *may* process all elements in a stream, e.g.
 - map() processes all of the elements in its input stream



Overview of Common Stream Aggregate Operations

- An aggregate operation *may* process all elements in a stream, e.g.
 - `map()` processes all of the elements in its input stream
 - `limit()` & `findFirst()` are “short-circuit” operations that halt further processing after their condition is reached



End of Overview of Java 8 Streams (Part 2)

Overview of Java 8 Streams (Part 3)

Douglas C. Schmidt

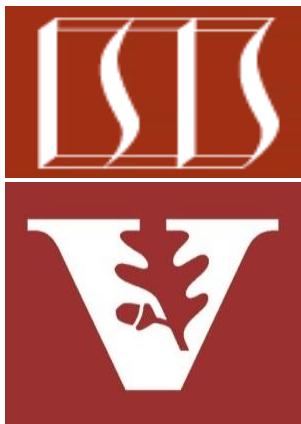
d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA



Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of Java 8 streams, e.g.,
 - Fundamentals of streams
 - Common stream aggregate operations
 - “Splittable iterators” (Spliterators)

Interface Spliterator<T>

Type Parameters:

T - the type of elements returned by this Spliterator

All Known Subinterfaces:

Spliterator.OfDouble, Spliterator.OfInt, Spliterator.OfLong,
Spliterator.OfPrimitive<T,T_CONS,T_SPLITR>

All Known Implementing Classes:

Spliterators.AbstractDoubleSpliterator,
Spliterators.AbstractIntSpliterator,
Spliterators.AbstractLongSpliterator,
Spliterators.AbstractSpliterator

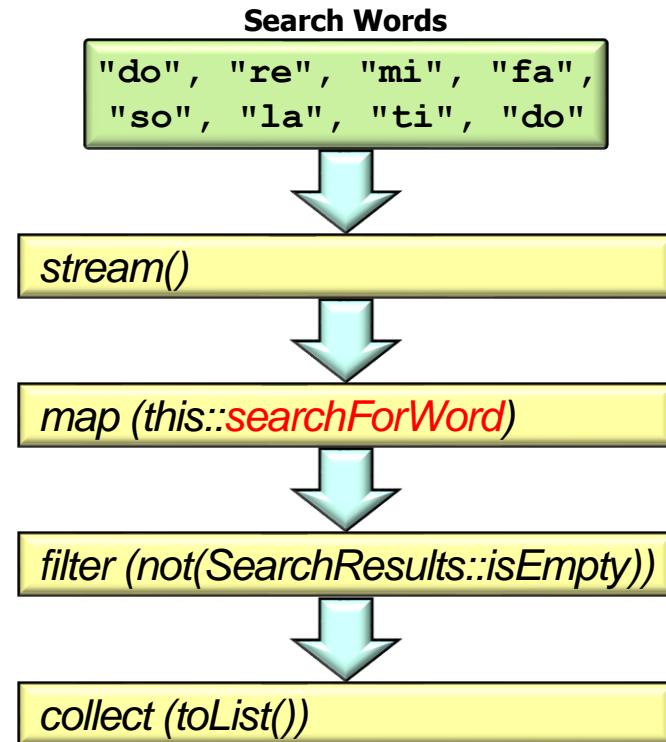
public interface **Spliterator<T>**

An object for traversing and partitioning elements of a source. The source of elements covered by a Spliterator could be, for example, an array, a Collection, an IO channel, or a generator function.

See docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html

Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of Java 8 streams, e.g.,
 - Fundamentals of streams
 - Common stream aggregate operations
 - “Splittable iterators” (Spliterators)
 - We’ll show how a Spliterator is used in the SimpleSearchStream



Overview of the Java Spliterator

Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8

Interface Spliterator<T>

Type Parameters:

T - the type of elements returned by this Spliterator

All Known Subinterfaces:

Spliterator.OfDouble, Spliterator.OfInt, Spliterator.OfLong,
Spliterator.OfPrimitive<T,T_CONS,T_SPLITR>

All Known Implementing Classes:

Spliterators.AbstractDoubleSpliterator,
Spliterators.AbstractIntSpliterator,
Spliterators.AbstractLongSpliterator,
Spliterators.AbstractSpliterator

public interface Spliterator<T>

An object for traversing and partitioning elements of a source. The source of elements covered by a Spliterator could be, for example, an array, a Collection, an IO channel, or a generator function.

A Spliterator may traverse elements individually (`tryAdvance()`) or sequentially in bulk (`forEachRemaining()`).

See docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html

Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8
 - *Iterator* – It can be used to traverse elements of a source
 - e.g., a collection, array, etc.

```
List<String> quote = Arrays.asList("This ", "above ", "all- ",  
        "to ", "thine ", "own ",  
        "self ", "be ", "true", ",\n",  
        ...);  
  
for (Spliterator<String> s =  
        quote.spliterator();  
    s.tryAdvance(System.out::print)  
        != false;  
)  
    continue;
```

Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8
 - *Iterator* – It can be used to traverse elements of a source
 - e.g., a collection, array, etc.

The source is an array/list of strings

```
List<String> quote = Arrays.asList  
        ("This ", "above ", "all- ",  
         "to ", "thine ", "own ",  
         "self ", "be ", "true", ",\n",  
         ...);  
  
for (Spliterator<String> s =  
        quote.spliterator();  
        s.tryAdvance(System.out::print)  
        != false;  
    )  
    continue;
```

Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8
 - *Iterator* – It can be used to traverse elements of a source
 - e.g., a collection, array, etc.

```
List<String> quote = Arrays.asList  
        ("This ", "above ", "all- ",  
         "to ", "thine ", "own ",  
         "self ", "be ", "true", ",\n",  
         ...);  
  
for (Spliterator<String> s =  
        quote.spliterator();  
        s.tryAdvance(System.out::print)  
        != false;  
        )  
    continue;
```

Create a spliterator for
the entire array/list

Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8
 - *Iterator* – It can be used to traverse elements of a source
 - e.g., a collection, array, etc.

```
List<String> quote = Arrays.asList  
        ("This ", "above ", "all- ",  
         "to ", "thine ", "own ",  
         "self ", "be ", "true", ",\n",  
         ...);  
  
for (Spliterator<String> s =  
        quote.spliterator();  
        s.tryAdvance(System.out::print)  
        != false;  
    )  
    continue;
```

tryAdvance() combines
the *hasNext()* & *next()*
methods of *Iterator*

Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8
 - *Iterator* – It can be used to traverse elements of a source
 - e.g., a collection, array, etc.

```
List<String> quote = Arrays.asList  
        ("This ", "above ", "all- ",  
         "to ", "thine ", "own ",  
         "self ", "be ", "true", ",\n",  
         ...);  
  
for (Spliterator<String> s =  
        quote.spliterator();  
        s.tryAdvance(System.out::print)  
        != false;  
        )  
    continue;
```

Print value of each string in the quote

Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8
 - *Iterator* – It can be used to traverse elements of a source
 - *Split* – It can also partition all elements of a source

```
List<String> quote = Arrays.asList("This ", "above ", "all- ",  
        "to ", "thine ", "own ",  
        "self ", "be ", "true", ",\n",  
        ...);  
  
Spliterator<String> secondHalf =  
        quote.spliterator();  
Spliterator<String> firstHalf =  
        secondHalf.trySplit();  
  
firstHalf.forEachRemaining  
        (System.out::print);  
secondHalf.forEachRemaining  
        (System.out::print);
```

Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8
 - *Iterator* – It can be used to traverse elements of a source
 - *Split* – It can also partition all elements of a source

Create a spliterator for the entire array/list

```
List<String> quote = Arrays.asList("This ", "above ", "all- ",  
        "to ", "thine ", "own ",  
        "self ", "be ", "true", ",\n",  
        ...);  
  
Spliterator<String> secondHalf =  
        quote.spliterator();  
Spliterator<String> firstHalf =  
        secondHalf.trySplit();  
  
firstHalf.forEachRemaining  
        (System.out::print);  
secondHalf.forEachRemaining  
        (System.out::print);
```

Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8
 - Iterator* – It can be used to traverse elements of a source
 - Split* – It can also partition all elements of a source

trySplit() returns a spliterator covering elements that will no longer be covered by the invoking spliterator

```
List<String> quote = Arrays.asList("This ", "above ", "all- ",  
        "to ", "thine ", "own ",  
        "self ", "be ", "true", ",\n",  
        ...);  
  
Spliterator<String> secondHalf =  
        quote.spliterator();  
Spliterator<String> firstHalf =  
        secondHalf.trySplit();  
  
firstHalf.forEachRemaining  
        (System.out::print);  
secondHalf.forEachRemaining  
        (System.out::print);
```

Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8
 - *Iterator* – It can be used to traverse elements of a source
 - *Split* – It can also partition all elements of a source



```
List<String> quote = Arrays.asList("This ", "above ", "all- ",  
        "to ", "thine ", "own ",  
        "self ", "be ", "true", ",\n",  
        ...);  
  
Spliterator<String> secondHalf =  
        quotespliterator();  
Spliterator<String> firstHalf =  
        secondHalf.trySplit();  
  
firstHalf.forEachRemaining  
        (System.out::print);  
secondHalf.forEachRemaining  
        (System.out::print);
```

Ideally a spliterator splits the original input source in half!

Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8
 - Iterator* – It can be used to traverse elements of a source
 - Split* – It can also partition all elements of a source

Performs the action for each element in the spliterator

```
List<String> quote = Arrays.asList("This ", "above ", "all- ",  
        "to ", "thine ", "own ",  
        "self ", "be ", "true", ",\n",  
        ...);
```

```
Spliterator<String> secondHalf =  
        quote.spliterator();  
Spliterator<String> firstHalf =  
        secondHalf.trySplit();  
  
firstHalf.forEachRemaining  
        (System.out::print);  
secondHalf.forEachRemaining  
        (System.out::print);
```

Overview of the Java Spliterator

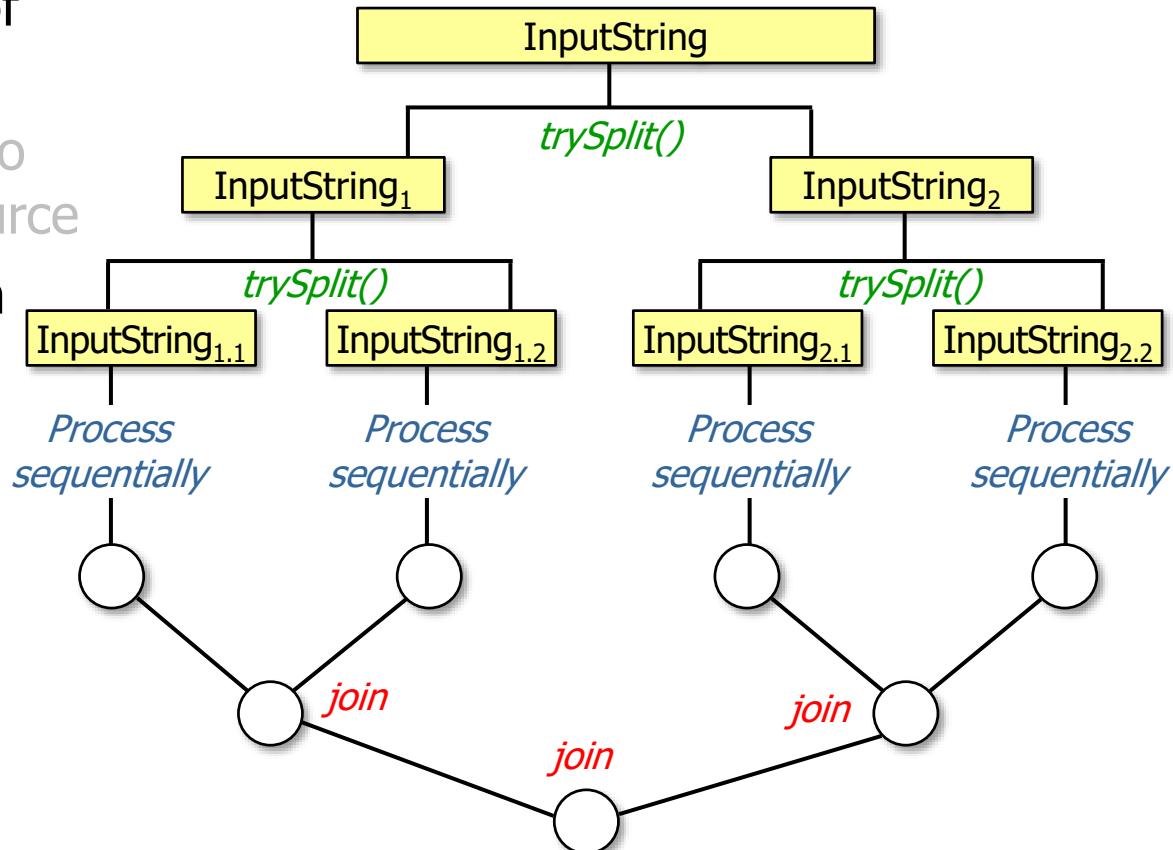
- A Spliterator is a new type of "splittable iterator" in Java 8
 - Iterator* – It can be used to traverse elements of a source
 - Split* – It can also partition all elements of a source

Print value of each string in the quote

```
List<String> quote = Arrays.asList("This ", "above ", "all- ",  
        "to ", "thine ", "own ",  
        "self ", "be ", "true", ",\n",  
        ...);  
  
Spliterator<String> secondHalf =  
        quote.spliterator();  
Spliterator<String> firstHalf =  
        secondHalf.trySplit();  
  
firstHalf.forEachRemaining  
        (System.out::print);  
secondHalf.forEachRemaining  
        (System.out::print);
```

Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8
 - *Iterator* – It can be used to traverse elements of a source
 - *Split* – It can also partition all elements of a source
 - Mostly used with Java 8 parallel streams



Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8
 - *Iterator* – It can be used to traverse elements of a source
 - *Split* – It can also partition all elements of a source



Interface Spliterator<T>

Type Parameters:

T - the type of elements returned by this Spliterator

All Known Subinterfaces:

Spliterator.OfDouble, Spliterator.OfInt, Spliterator.OfLong,
Spliterator.OfPrimitive<T,T_CONS,T_SPLITR>

All Known Implementing Classes:

Spliterators.AbstractDoubleSpliterator,
Spliterators.AbstractIntSpliterator,
Spliterators.AbstractLongSpliterator,
Spliterators.AbstractSpliterator

public interface Spliterator<T>

An object for traversing and partitioning elements of a source. The source of elements covered by a Spliterator could be, for example, an array, a Collection, an IO channel, or a generator function.

A Spliterator may traverse elements individually (`tryAdvance()`) or sequentially in bulk (`forEachRemaining()`).

We'll focus on traversal now & on partitioning after covering parallel streams

Overview of the Java Spliterator

- The StreamSupport.stream() factory method creates a new sequential or parallel stream from a Spliterator

stream

```
public static <T> Stream<T> stream(Spliterator<T> spliterator,  
                                     boolean parallel)
```

Creates a new sequential or parallel Stream from a Spliterator.

The spliterator is only traversed, split, or queried for estimated size after the terminal operation of the stream pipeline commences.

It is strongly recommended the spliterator report a characteristic of IMMUTABLE or CONCURRENT, or be late-binding. Otherwise, stream(java.util.function.Supplier, int, boolean) should be used to reduce the scope of potential interference with the source. See Non-Interference for more details.

Type Parameters:

T - the type of stream elements

Parameters:

spliterator - a Spliterator describing the stream elements

parallel - if true then the returned stream is a parallel stream; if false the returned stream is a sequential stream.

Returns:

a new sequential or parallel Stream

Overview of the Java Spliterator

- The StreamSupport.stream() factory method creates a new sequential or parallel stream from a spliterator
 - e.g., the Collection interface defines two default methods using this capability

```
public interface Collection<E>
    extends Iterable<E> {
    ...
    default Stream<E> stream() {
        return StreamSupport
            .stream(spliterator(),
                    false);
    }

    default Stream<E>
    parallelStream() {
        return StreamSupport
            .stream(spliterator(),
                    true);
    }
}
```

See jdk8/jdk8/jdk/file/tip/src/share/classes/java/util/Collection.java

Overview of the Java Spliterator

- The StreamSupport.stream() factory method creates a new sequential or parallel stream from a spliterator
 - e.g., the Collection interface defines two default methods using this capability

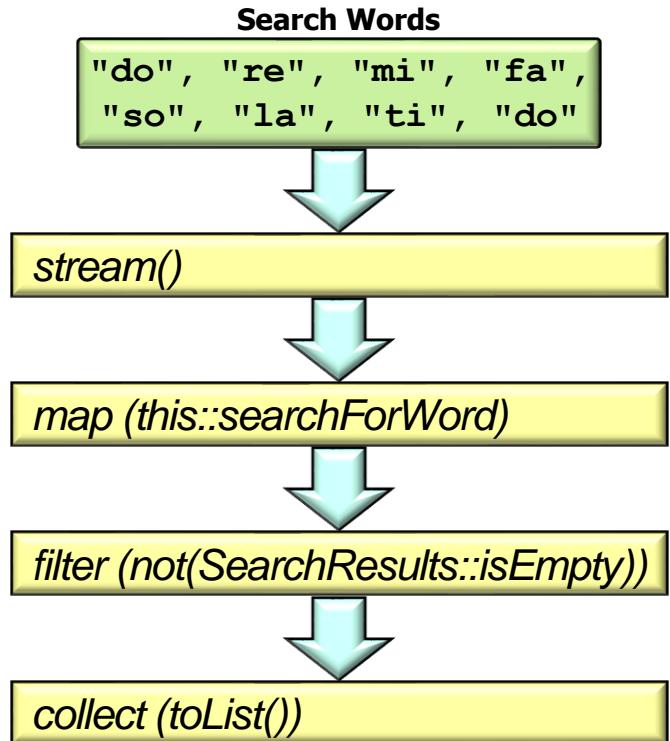
The 'false' parameter creates a sequential stream, whereas 'true' creates a parallel stream

```
public interface Collection<E>
    extends Iterable<E> {
    ...
    default Stream<E> stream() {
        return StreamSupport
            .stream(spliterator(),
                    false);
    }
    default Stream<E>
    parallelStream() {
        return StreamSupport
            .stream(spliterator(),
                    true);
    }
}
```

Using Java Spliterator in SimpleSearchStream

Using Java Splitter in SimpleSearchStream

- The SimpleSearchStream program uses a sequential splitter

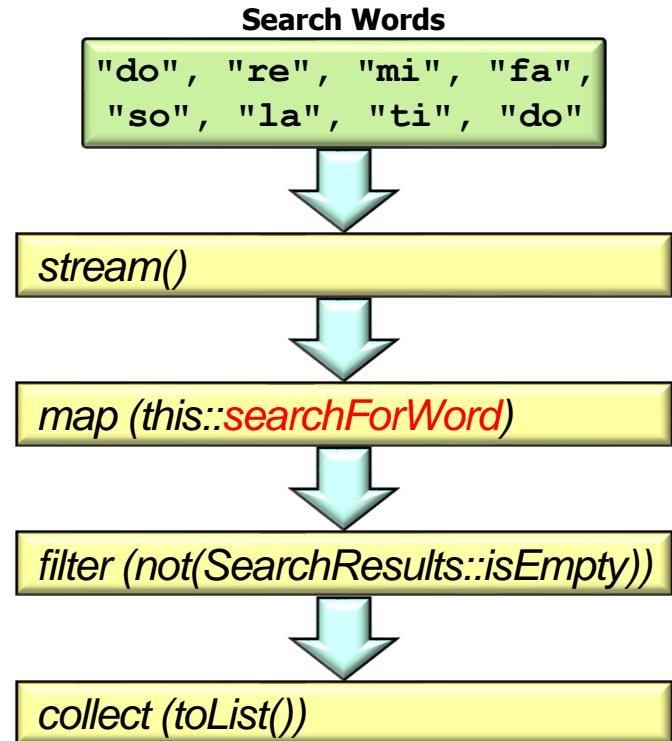


Using Java Spliterator in SimpleSearchStream

- searchForWord() uses the spliterator to find all instances of a word in the input & return a list of all the SearchResults

SearchResults **searchForWord**

```
(String word) {  
    return new SearchResults  
        (..., word, ..., StreamSupport  
            .stream(new WordMatchSpliterator  
                (mInput, word),  
                false)  
        .collect(toList()));  
}
```



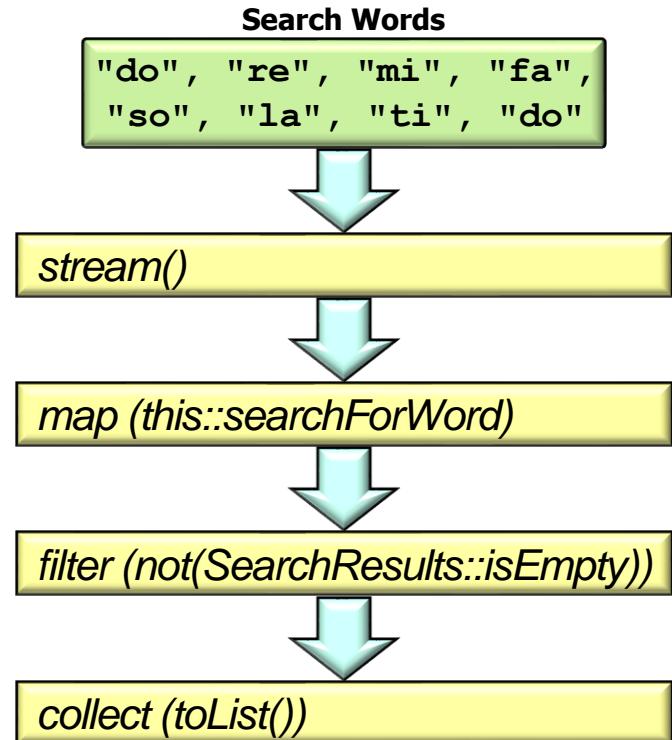
See [SimpleSearchStream/src/main/java/search/WordSearcher.java](#)

Using Java Spliterator in SimpleSearchStream

- searchForWord() uses the spliterator to find all instances of a word in the input & return a list of all the SearchResults

```
SearchResults searchForWord  
          (String word) {  
    return new SearchResults  
    (..., word, ..., StreamSupport  
     .stream(new WordMatchSpliterator  
             (mInput, word),  
             false)  
     .collect(toList()));  
}
```

StreamSupport.stream() creates a sequential stream via the WordMatchSpliterator class

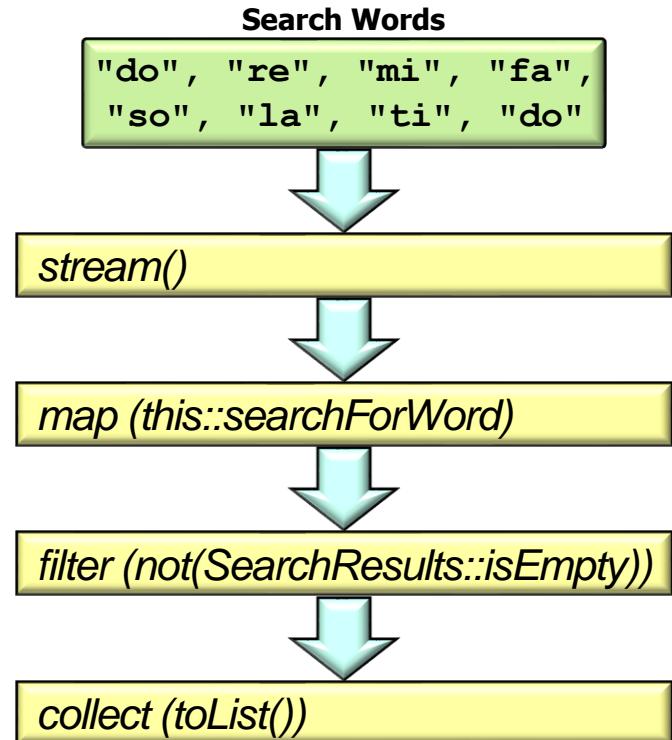


Using Java Spliterator in SimpleSearchStream

- searchForWord() uses the spliterator to find all instances of a word in the input & return a list of all the SearchResults

```
SearchResults searchForWord  
          (String word) {  
    return new SearchResults  
    (..., word, ..., StreamSupport  
     .stream(new WordMatchSpliterator  
             (mInput, word),  
             false)  
     .collect(toList()));  
}
```

*This stream is collected into a list
of SearchResults.Result objects*



Using Java Spliterator in SimpleSearchStream

- WordMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a word appears in an input string

```
class WordMatchSpliterator
    extends Spliterators.AbstractSpliterator<Result> {
private final Matcher mWordMatcher;

public WordMatchSpliterator(String input, String word) {
    ...
    String regexWord = "\\b" + word.trim() + "\\b";
    mWordMatcher =
        Pattern.compile(regexWord,
                        Pattern.CASE_INSENSITIVE)
        .matcher(input);
}
```

See [SimpleSearchStream/src/main/java/search/WordMatchSpliterator.java](#)

Using Java Spliterator in SimpleSearchStream

- WordMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a word appears in an input string

```
class WordMatchSpliterator
    extends Spliterators.AbstractSpliterator<Result> {
private final Matcher mWordMatcher;

public WordMatchSpliterator(String input, String word) {
    ...
    String regexWord = "\\b" + word.trim() + "\\b";
    mWordMatcher =
        Pattern.compile(regexWord,
                        Pattern.CASE_INSENSITIVE)
        .matcher(input);
}
```

*Create a regex that
matches only a "word"*

Using Java Spliterator in SimpleSearchStream

- WordMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a word appears in an input string

```
class WordMatchSpliterator
    extends Spliterators.AbstractSpliterator<Result> {
private final Matcher mWordMatcher;

public WordMatchSpliterator(String input, String word) {
    ...
    String regexWord = "\\b" + word.trim() + "\\b";
    mWordMatcher =
        Pattern.compile(regexWord,
                        Pattern.CASE_INSENSITIVE)
            .matcher(input);
}
```

Compile the regex & create a matcher for the input string

See docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html

Using Java Spliterator in SimpleSearchStream

- WordMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a word appears in an input string

```
class WordMatchSpliterator
    extends Spliterators.AbstractSpliterator<Result> {
    ...
    public boolean tryAdvance(Consumer<? super Result> action) {
        if (!mWordMatcher.find())
            return false;
        else {
            action.accept(new Result(mWordMatcher.start()));
            return true;
        }
    }
}
```

Called by the Java 8 streams framework to attempt to advance the spliterator by one word match

Using Java Spliterator in SimpleSearchStream

- WordMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a word appears in an input string

```
class WordMatchSpliterator
    extends Spliterators.AbstractSpliterator<Result> {
    ...
    public boolean tryAdvance(Consumer<? super Result> action) {
        if (!mWordMatcher.find())
            return false;
        else {
            action.accept(new Result(mWordMatcher.start()));
            return true;
        }
    }
}
```

Passes the result (if any) back "by reference" to the streams framework

Using Java Spliterator in SimpleSearchStream

- WordMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a word appears in an input string

```
class WordMatchSpliterator
    extends Spliterators.AbstractSpliterator<Result> {
    ...
    public boolean tryAdvance(Consumer<? super Result> action) {
        if (!mWordMatcher.find())
            return false;
        else {
            action.accept(new Result(mWordMatcher.start()));
            return true;
        }
    }
}
```

*Check if any remaining phrases
in the input match the regex*

Using Java Spliterator in SimpleSearchStream

- WordMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a word appears in an input string

```
class WordMatchSpliterator
    extends Spliterators.AbstractSpliterator<Result> {
    ...
    public boolean tryAdvance(Consumer<? super Result> action) {
        if (!mWordMatcher.find())
            return false;
        else {
            action.accept(new Result(mWordMatcher.start()));
            return true;
        }
    }
}
```

Inform the streams framework to cease calling tryAdvance() if there's no match

Using Java Spliterator in SimpleSearchStream

- WordMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a word appears in an input string

```
class WordMatchSpliterator
    extends Spliterators.AbstractSpliterator<Result> {
    ...
    public boolean tryAdvance(Consumer<? super Result> action) {
        if (!mWordMatcher.find())
            return false;
        else {
            action.accept(new Result(mWordMatcher.start()));
            return true;
        }
    }
}
```



accept() stores the index in the input string where the match occurred, which is returned to the streams framework

Using Java Spliterator in SimpleSearchStream

- WordMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a word appears in an input string

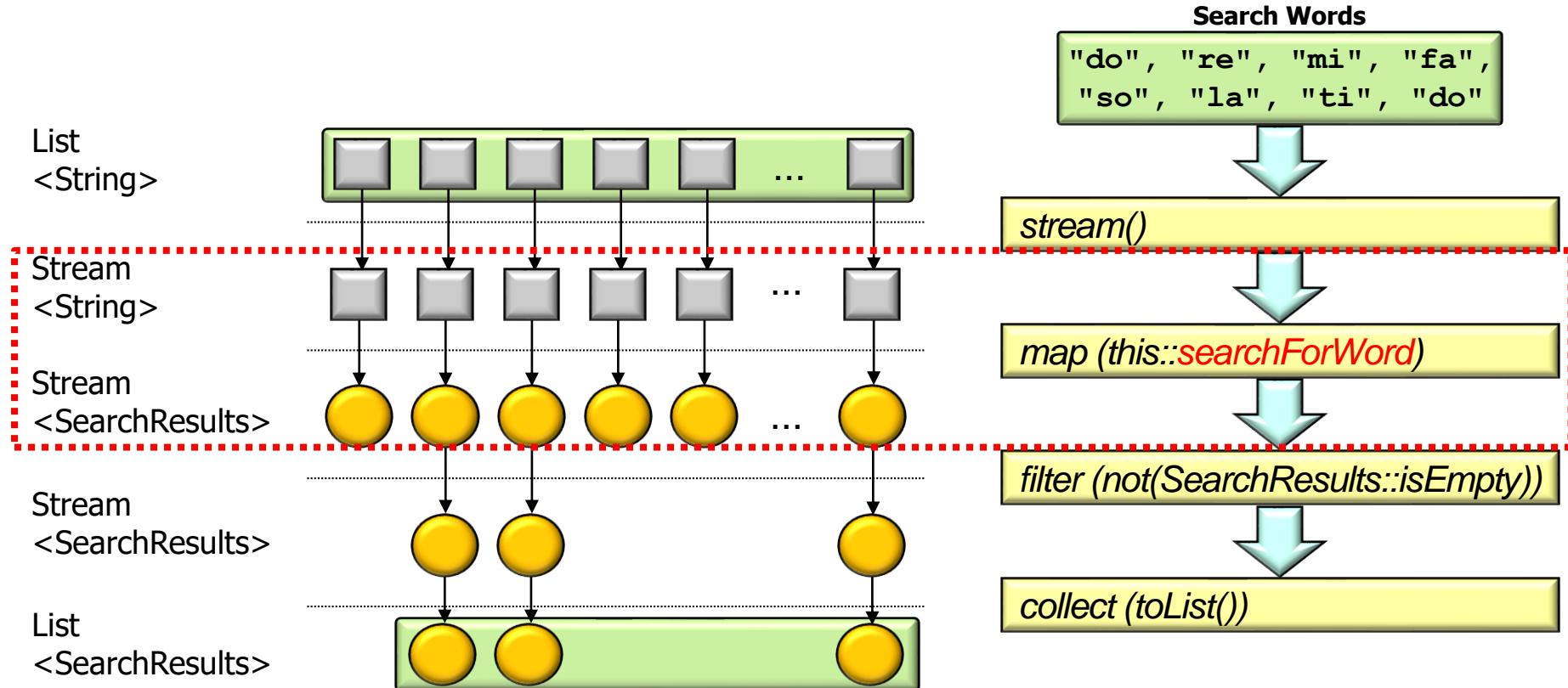
```
class WordMatchSpliterator
    extends Spliterators.AbstractSpliterator<Result> {
    ...
    public boolean tryAdvance(Consumer<? super Result> action) {
        if (!mWordMatcher.find())
            return false;

        else {
            action.accept(new Result(mWordMatcher.start()));
            return true;
        }
    }
}
```

*Inform the streams framework
to continue calling tryAdvance()*

Using Java Splitter in SimpleSearchStream

- Here's the output that searchForWord() & WordMatchSpliterator produce



End of Overview of Java 8 Streams (Part 3)

Overview of Java 8 Streams (Part 4)

Douglas C. Schmidt

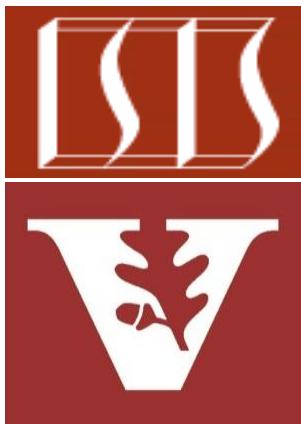
d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

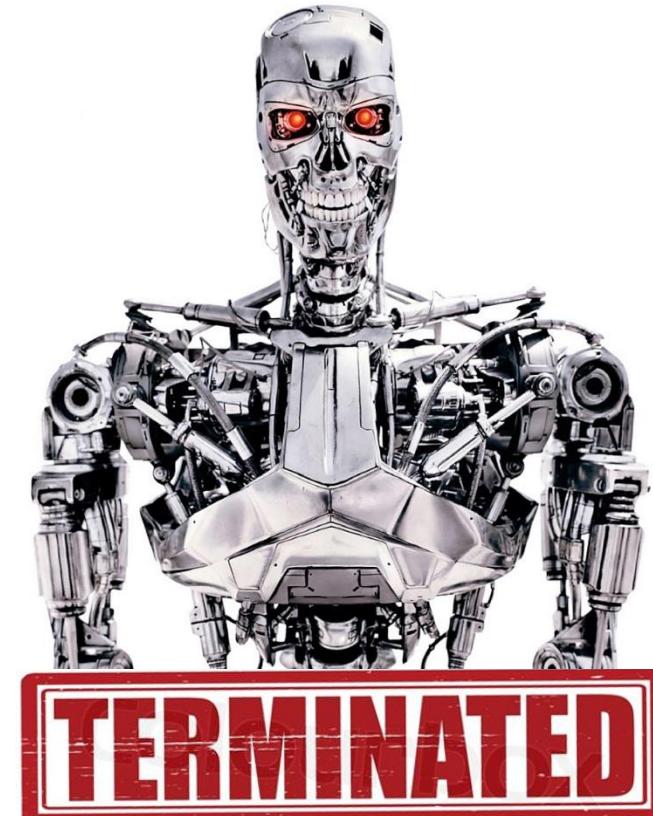
Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA



Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of Java 8 streams, e.g.,
 - Fundamentals of streams
 - Common stream aggregate operations
 - “Splittable iterators” (Spliterators)
 - Terminating a stream



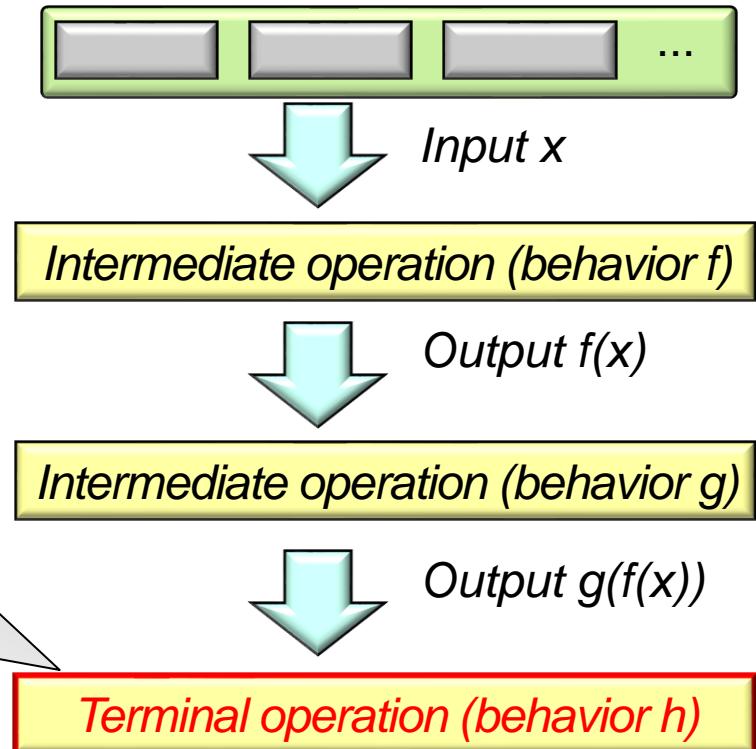
Terminating a Stream

Terminating a Stream

- Every stream finishes with a terminal operation that yields a non-stream result

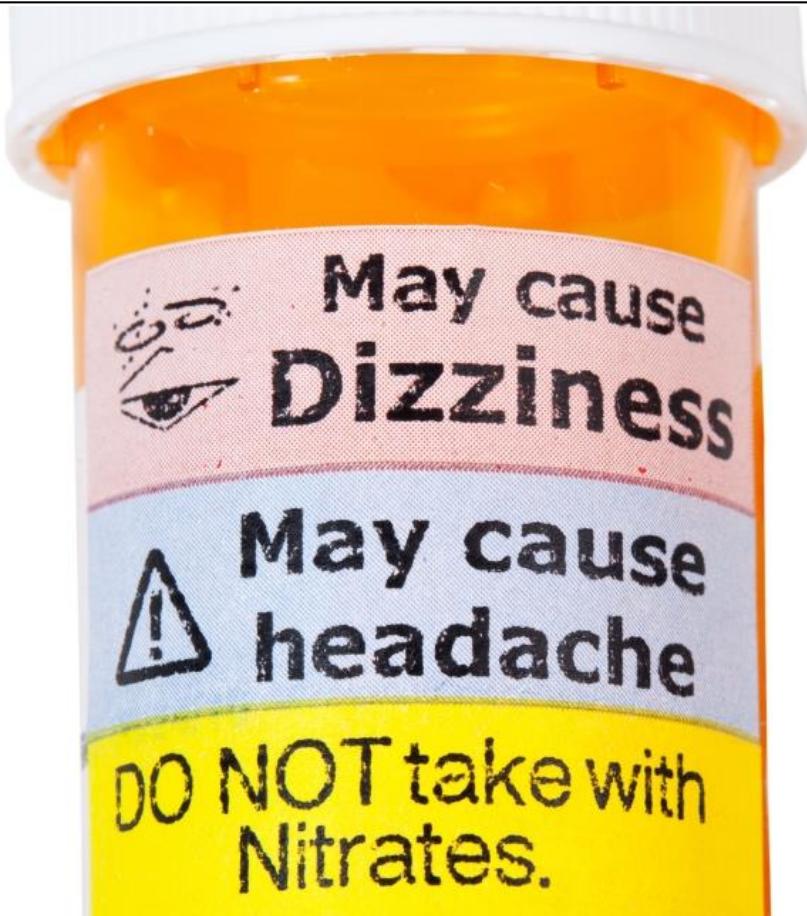
Stream

```
.of("horatio",
    "laertes",
    "Hamlet", ...)
.filter(s -> toLowerCase
        (s.charAt(0)) == 'h')
.map(this::capitalize)
.sorted()
.forEach(System.out::println);
```



Terminating a Stream

- Every stream finishes with a terminal operation that yields a non-stream result, e.g.
 - No value at all
 - i.e., only side-effects!



Terminating a Stream

- Every stream finishes with a terminal operation that yields a non-stream result, e.g.
 - No value at all
 - The result of a reduction operation



See docs.oracle.com/javase/tutorial/collectionsstreams/reduction.html

Terminating a Stream

- Several terminal operations return no value at all

```
void runForEach() {  
    ...  
  
    Stream  
        .of("horatio", "laertes",  
             "Hamlet", ...)  
        .filter(s -> toLowerCase  
                (s.charAt(0)) == 'h')  
        .map(this::capitalize)  
        .sorted()  
        .forEach  
            (System.out::println);  
    ...  
}
```

Terminating a Stream

- Several terminal operations return no value at all

```
void runForEach() {  
    ...  
  
    Stream  
        .of("horatio", "laertes",  
             "Hamlet", ...)  
        .filter(s -> toLowerCase  
                (s.charAt(0)) == 'h')  
        .map(this::capitalize)  
        .sorted()  
        .forEach  
            (System.out::println);  
    ...  
}
```

Performs the designated action on each element of this stream

Terminating a Stream

- Several terminal operations return no value at all

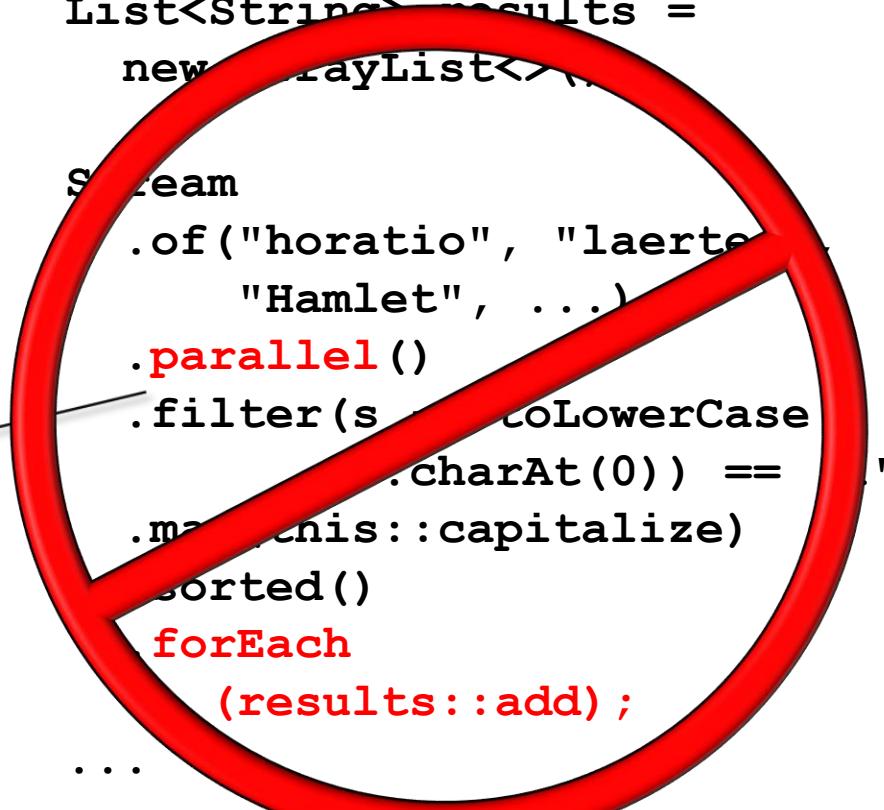
```
void runForEach() {  
    List<String> results =  
        new ArrayList<>();  
  
    Stream  
        .of("horatio", "laertes",  
            "Hamlet", ...)  
        .filter(s -> toLowerCase  
            (s.charAt(0)) == 'h')  
        .map(this::capitalize)  
        .sorted()  
        .forEach  
            (results::add);  
    ...  
}
```

The lambda passed to forEach()
is allowed to have side-effects

Terminating a Stream

- Several terminal operations return no value at all

```
void runForEach() {  
    List<String> results =  
        new ArrayList<>();  
  
    Stream  
        .of("horatio", "laertes",  
            "Hamlet", ...)  
        .parallel()  
        .filter(s -> s.toLowerCase()  
            .charAt(0) == 'h')  
        .map(s -> s.toUpperCase())  
        .sorted()  
        .forEach  
            (results::add);  
    ...  
}
```



Avoid using forEach() with side-effects in a parallel stream!!!

Terminating a Stream

- Other terminal operations return the result of a reduction operation

```
void runCollectTo*() {  
    List<String> characters =  
        Arrays.asList("horatio",  
                      "laertes",  
                      "Hamlet", ...);  
  
    ...<String> results =  
        characters  
            .stream()  
            .filter(s ->  
                toLowerCase(...) == 'h')  
            .map(this::capitalize)  
            .sorted()  
            .collect(...); ...  
}
```

Terminating a Stream

- Other terminal operations return the result of a reduction operation, e.g.
 - a collection

Performs a mutable reduction on all elements of this stream using some type of collector

```
void runCollectTo*() {  
    List<String> characters =  
        Arrays.asList("horatio",  
                      "laertes",  
                      "Hamlet", ...);  
  
    ...<String> results =  
    characters  
        .stream()  
        .filter(s ->  
           toLowerCase(...) == 'h')  
        .map(this::capitalize)  
        .sorted()  
        .collect(...); ...
```

Terminating a Stream

- Other terminal operations return the result of a reduction operation, e.g.
 - a collection

A collector performs reduction operations, e.g., summarizing elements according to various criteria, accumulating elements into various types of collections, etc.

```
void runCollectTo*() {  
    List<String> characters =  
        Arrays.asList("horatio",  
                      "laertes",  
                      "Hamlet", ...);  
  
    ...<String> results =  
        characters  
            .stream()  
            .filter(s ->  
               toLowerCase(...) == 'h')  
            .map(this::capitalize)  
            .sorted()  
            .collect(...); ...
```

Terminating a Stream

- Other terminal operations return the result of a reduction operation, e.g.
 - a collection

```
void runCollectToList() {  
    List<String> characters =  
        Arrays.asList("horatio",  
                     "laertes",  
                     "Hamlet, ...");  
  
    List<String> results =  
        characters  
            .stream()  
            .filter(s ->  
                toLowerCase(...) == 'h')  
            .map(this::capitalize)  
            .sorted()  
            .collect(toList()); ...  
}
```

Collect the results into a ArrayList

Terminating a Stream

- Other terminal operations return the result of a reduction operation, e.g.
 - a collection

```
void runCollectToSet() {  
    List<String> characters =  
        Arrays.asList("horatio",  
                     "laertes",  
                     "Hamlet", ...);  
  
    Set<String> results =  
        characters  
            .stream()  
            .filter(s ->  
                toLowerCase(...).=='h')  
            .map(this::capitalize)  
            .collect(toSet()); ...  
}
```

*Collect the results into a HashSet,
which has no duplicate entries*

Terminating a Stream

- Other terminal operations return the result of a reduction operation, e.g.
 - a collection

```
void runCollectToMap() {  
    List<String> characters =  
        Arrays.asList("horatio",  
                     "laertes",  
                     "Hamlet", ...);  
  
    Map<String, Integer> results =  
        characters  
            .stream()  
            .filter(s ->  
                toLowerCase(...).=='h')  
            .map(this::capitalize)  
            .collect(toMap(identity(),  
                           String::length,  
                           Integer::sum));  
    ...  
}
```

Collect results into a HashMap, along with the length of (merged duplicate) entries

Terminating a Stream

- Other terminal operations return the result of a reduction operation, e.g.
 - a collection

```
void runCollectGroupingBy() {  
    List<String> characters =  
        Arrays.asList("horatio",  
                     "laertes",  
                     "Hamlet", ...);  
  
    Map<String, Long> results =  
        ...  
        .collect  
            (groupingBy  
                (identity(),  
                 TreeMap::new,  
                 summingLong  
                     (String::length)));  
    ...  
}
```

Collect the results into a TreeMap by grouping elements according to name (key) & name length (value)

Terminating a Stream

- Other terminal operations return the result of a reduction operation, e.g.
 - a collection



groupingBy() partitions a stream via a "classifier" function, e.g., by the identity() function that always returns its input argument

```
void runCollectGroupingBy() {  
    List<String> characters =  
        Arrays.asList("horatio",  
                     "laertes",  
                     "Hamlet", ...);  
  
    Map<String, Long> results =  
        ...  
        .collect  
            (groupingBy  
                (identity(),  
                 TreeMap::new,  
                 summingLong  
                     (String::length)));  
    ...  
}
```

Terminating a Stream

- Other terminal operations return the result of a reduction operation, e.g.
 - a collection

```
void runCollectGroupingBy() {  
    List<String> characters =  
        Arrays.asList("horatio",  
                     "laertes",  
                     "Hamlet", ...);  
  
    Map<String, Long> results =  
        ...  
        .collect  
            (groupingBy  
                (identity(),  
                 TreeMap::new,  
                 summingLong  
                     (String::length)));  
    ...  
}
```

A factory supplier is used to create the type of map

Terminating a Stream

- Other terminal operations return the result of a reduction operation, e.g.
 - a collection

A "downstream collector" defines a collector applied by the Java runtime to the results of an earlier collector

```
void runCollectGroupingBy() {  
    List<String> characters =  
        Arrays.asList("horatio",  
                     "laertes",  
                     "Hamlet", ...);  
  
    Map<String, Long> results =  
        ...  
        .collect  
            (groupingBy  
                (identity(),  
                 TreeMap::new,  
                 summingLong  
                     (String::length)));  
    ...  
}
```

Terminating a Stream

- Other terminal operations return the result of a reduction operation, e.g.
 - a collection

Convert a string into a stream via regular expression splitting!

```
void runCollectReduce() {  
    Map<String, Long>  
        matchingCharactersMap =  
            Pattern.compile(",")  
                .splitAsStream  
                    ("horatio,Hamlet,...")  
...  
    .collect  
        (groupingBy  
            (identity(),  
            TreeMap::new,  
            summingLong  
                (String::length))) ;
```

Terminating a Stream

- Other terminal operations return the result of a reduction operation, e.g.
 - a collection

Collect the results into a TreeMap by grouping elements according to name (key) & name length (value)

```
void runCollectReduce() {  
    Map<String, Long>  
    matchingCharactersMap =  
    Pattern.compile(",")  
        .splitAsStream  
        ("horatio,Hamlet,...")  
        ...  
        .collect  
        (groupingBy  
            (identity(),  
            TreeMap::new,  
            summingLong  
                (String::length)));
```

Terminating a Stream

- Other terminal operations return the result of a reduction operation, e.g.
 - a collection
 - a primitive value

```
void runCollectReduce () {  
    Map<String, Long>  
        matchingCharactersMap =  
            ...  
  
    long countOfNameLengths =  
        matchingCharactersMap  
            .values()  
            .stream()  
            .reduce(0L,  
                    (x, y) -> x + y);
```

Sum up the lengths of all character names in Hamlet

Terminating a Stream

- Other terminal operations return the result of a reduction operation, e.g.

- a collection
- a primitive value

0 is the "identity," i.e., the initial value of the reduction & the default result if there are no elements in the stream

```
void runCollectReduce() {  
    Map<String, Long>  
        matchingCharactersMap =  
        ...  
  
    long countOfNameLengths =  
        matchingCharactersMap  
            .values()  
            .stream()  
            .reduce(0L,  
                    (x, y) -> x + y);  
}
```

Terminating a Stream

- Other terminal operations return the result of a reduction operation, e.g.

- a collection
- a primitive value

```
void runCollectReduce () {  
    Map<String, Long>  
        matchingCharactersMap =  
        ...  
  
    long countOfNameLengths =  
        matchingCharactersMap  
            .values()  
            .stream()  
            .reduce(0L,  
                    (x, y) -> x + y);
```

This lambda is the "accumulator," which is a stateless function that combines two values into a single (immutable) "reduced" value

Terminating a Stream

- Other terminal operations return the result of a reduction operation, e.g.

- a collection
- a primitive value

```
void runCollectReduce() {  
    Map<String, Long>  
        matchingCharactersMap =  
        ...  
  
    long countOfNameLengths =  
        matchingCharactersMap  
            .values()  
            .parallelStream()  
            .reduce(0L,  
                    (x, y) -> x + y,  
                    (x, y) -> x + y);
```

There's a 3 parameter "map/reduce" version of reduce() that's used in parallel streams

Terminating a Stream

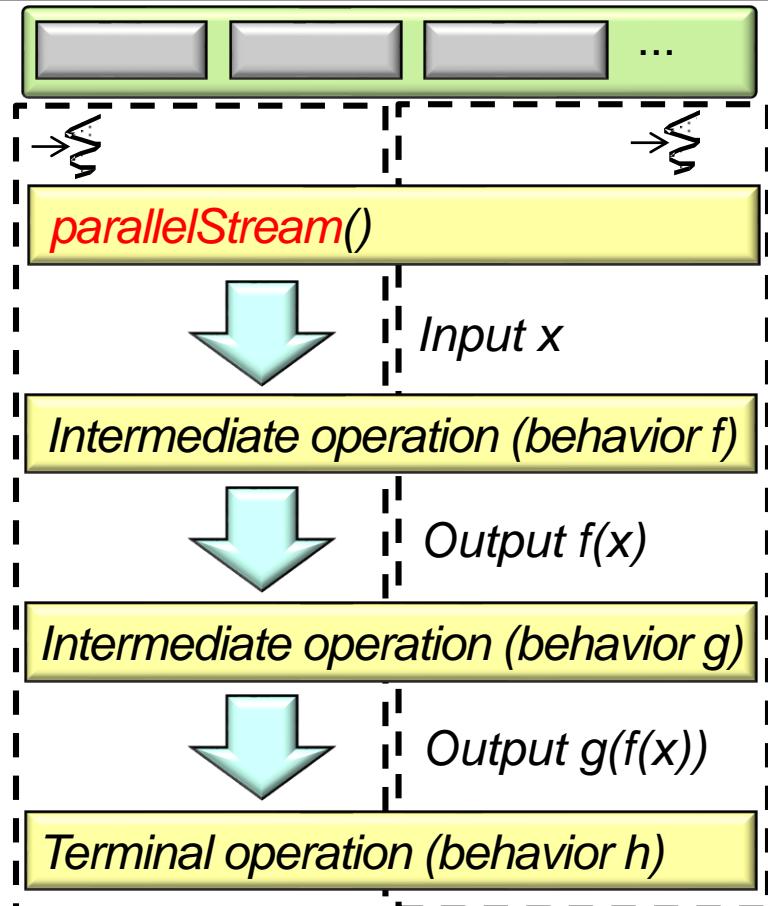
- Other terminal operations return the result of a reduction operation, e.g.
 - a collection
 - a primitive value

```
void runCollectReduce() {  
    Map<String, Long>  
        matchingCharactersMap =  
            ...  
  
    long countOfNameLengths =  
        matchingCharactersMap  
            .values()  
            .stream()  
            .sum()
```

There's a sum() method that simplifies the use of reduce()

Terminating a Stream

- Other terminal operations return the result of a reduction operation, e.g.
 - a collection
 - a primitive value



collect() & reduce() terminal operations work seamlessly with parallel streams

End of Overview of Java 8 Streams (Part 4)

Overview of Java 8 Streams (Part 5)

Douglas C. Schmidt

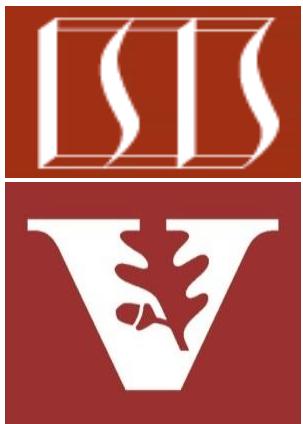
d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA



Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of Java 8 streams, e.g.,
 - Fundamentals of streams
 - Common stream aggregate operations
 - “Splittable iterators” (Spliterators)
 - Terminating a stream
 - Implementing non-concurrent collectors for sequential streams

Interface Collector<T,A,R>

Type Parameters:

T - the type of input elements to the reduction operation

A - the mutable accumulation type of the reduction operation (often hidden as an implementation detail)

R - the result type of the reduction operation

public interface Collector<T,A,R>

A mutable reduction operation that accumulates input elements into a mutable result container, optionally transforming the accumulated result into a final representation after all input elements have been processed. Reduction operations can be performed either sequentially or in parallel.

Examples of mutable reduction operations include: accumulating elements into a Collection; concatenating strings using a StringBuilder; computing summary information about elements such as sum, min, max, or average; computing "pivot table" summaries such as "maximum valued transaction by seller", etc. The class Collectors provides implementations of many common mutable reductions.

A Collector is specified by four functions that work together to accumulate entries into a mutable result container, and optionally perform a final transform on the result. They are:

See docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html

Implementing a Non-Concurrent Collector

Implementing a Non-Concurrent Collector

- A collector is used to terminate a stream

```
void runCollectToList() {  
    List<String> characters = Arrays  
        .asList("horatio", "laertes",  
                "Hamlet, ...);
```

```
List<String> results =  
    characters  
        .stream()  
        .filter(s ->  
            toLowerCase(...) == 'h')  
        .map(this::capitalize)  
        .sorted()  
        .collect(toList()); ...
```

Collect the results into a ArrayList

Implementing a Non-Concurrent Collector

- A collector is used to terminate a stream
 - Collector defines an interface whose implementations can accumulate input elements in a mutable result container

Interface Collector<T,A,R>

Type Parameters:

T - the type of input elements to the reduction operation

A - the mutable accumulation type of the reduction operation (often hidden as an implementation detail)

R - the result type of the reduction operation

public interface Collector<T,A,R>

A mutable reduction operation that accumulates input elements into a mutable result container, optionally transforming the accumulated result into a final representation after all input elements have been processed. Reduction operations can be performed either sequentially or in parallel.

Examples of mutable reduction operations include: accumulating elements into a Collection; concatenating strings using a StringBuilder; computing summary information about elements such as sum, min, max, or average; computing "pivot table" summaries such as "maximum valued transaction by seller", etc. The class Collectors provides implementations of many common mutable reductions.

A Collector is specified by four functions that work together to accumulate entries into a mutable result container, and optionally perform a final transform on the result. They are:

Implementing a Non-Concurrent Collector

- Collector implementations can either be non-concurrent or concurrent based on their characteristics

Enum Collector.Characteristics

java.lang.Object
java.lang.Enum<Collector.Characteristics>
java.util.stream.Collector.Characteristics

All Implemented Interfaces:

Serializable, Comparable<Collector.Characteristics>

Enclosing Interface:

Collector<T,A,R>

public static enum Collector.Characteristics
extends Enum<Collector.Characteristics>

Characteristics indicating properties of a Collector, which can be used to optimize reduction implementations.

Enum Constant Summary

Enum Constants

Enum Constant and Description

CONCURRENT

Indicates that this collector is *concurrent*, meaning that the result container can support the accumulator function being called concurrently with the same result container from multiple threads.

IDENTITY_FINISH

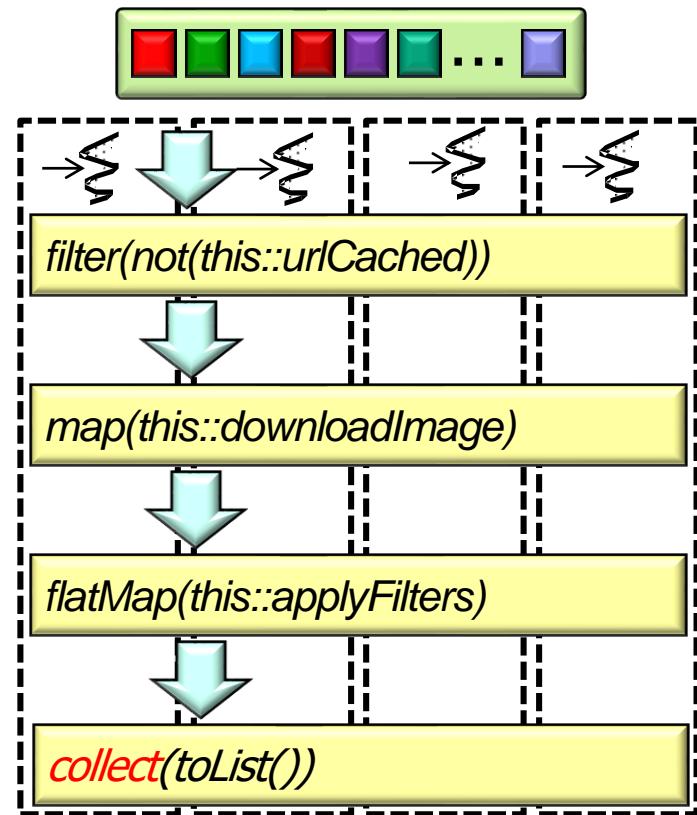
Indicates that the finisher function is the identity function and can be elided.

UNORDERED

Indicates that the collection operation does not commit to preserving the encounter order of input elements.

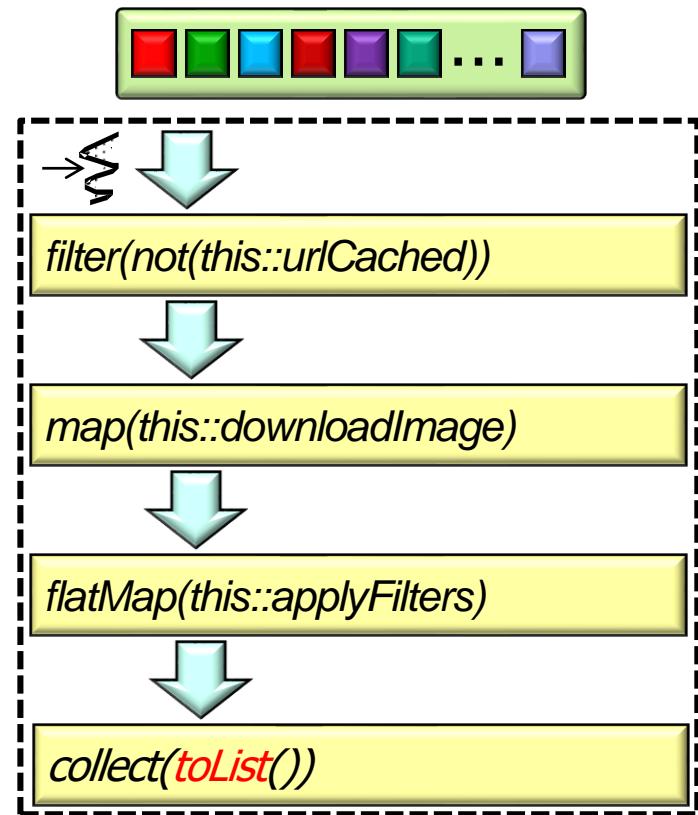
Implementing a Non-Concurrent Collector

- Collector implementations can either be non-concurrent or concurrent based on their characteristics
 - This distinction is only relevant for *parallel* streams



Implementing a Non-Concurrent Collector

- Collector implementations can either be non-concurrent or concurrent based on their characteristics
 - This distinction is only relevant for *parallel* streams
 - Our focus here is on non-concurrent collectors for sequential streams



Non-concurrent & concurrent collectors for parallel streams are covered later

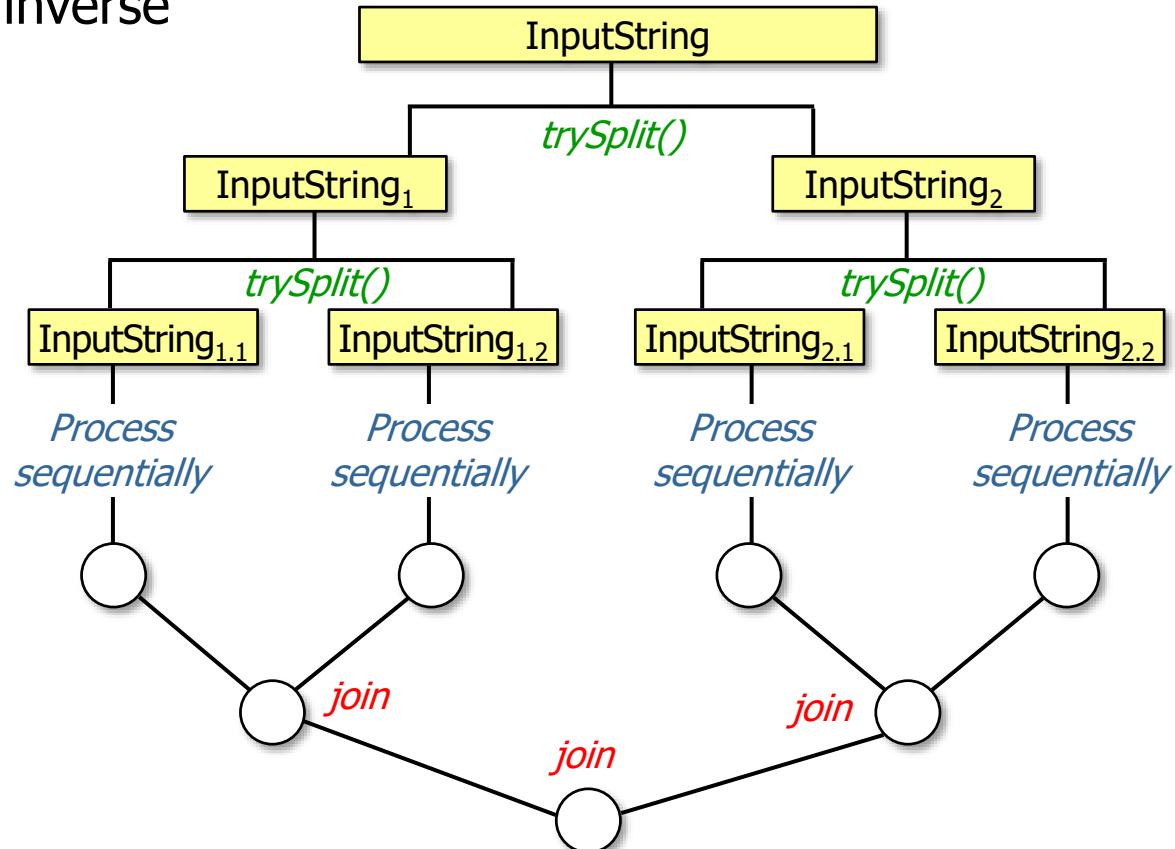
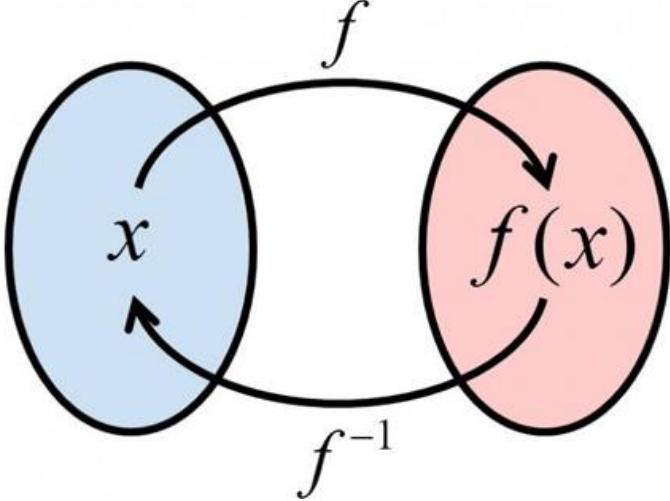
Implementing a Non-Concurrent Collector

- A non-concurrent collector for a sequential stream simply accumulates elements into a mutable result container



Implementing a Non-Concurrent Collector

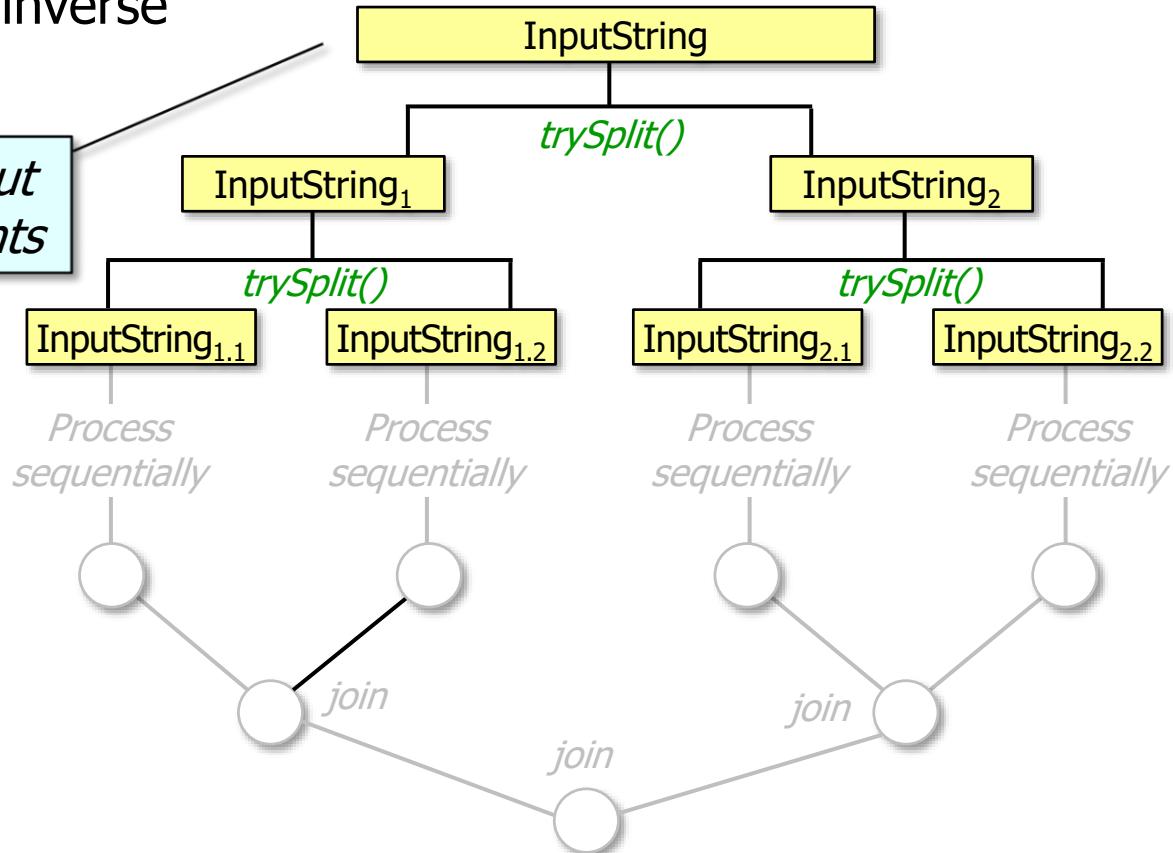
- A collector is essentially the inverse of a splitterator



Implementing a Non-Concurrent Collector

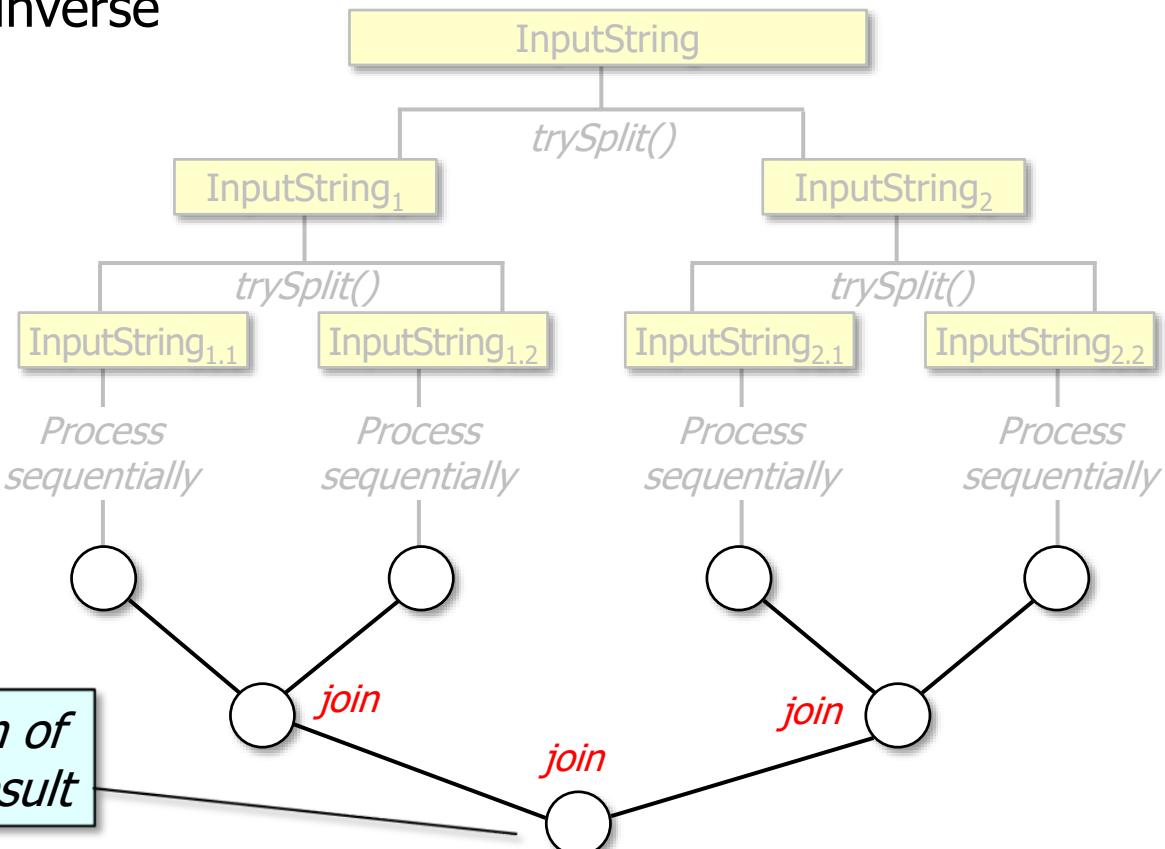
- A collector is essentially the inverse of a spliterator

A spliterator splits a single input source into a stream of elements



Implementing a Non-Concurrent Collector

- A collector is essentially the inverse of a splitterator



Implementing a Non-Concurrent Collector

- The Collector interface defines three generic types



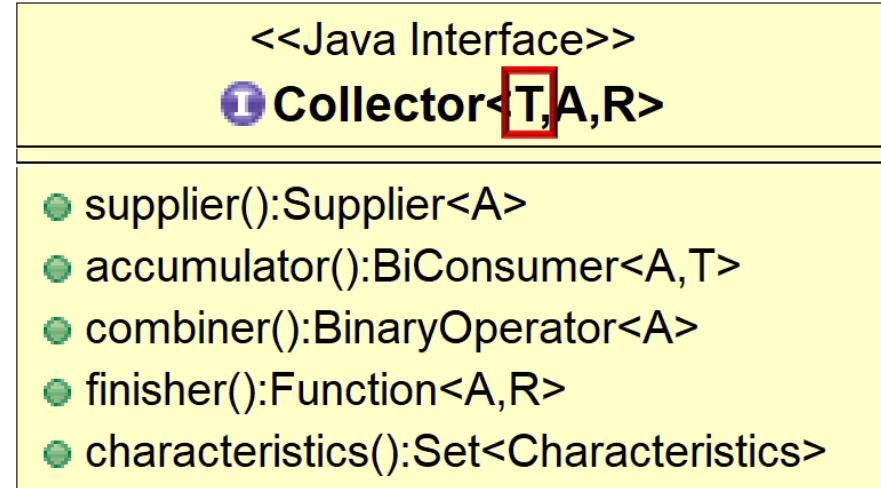
<<Java Interface>>

Collector<T,A,R>

- supplier():Supplier<A>
- accumulator():BiConsumer<A,T>
- combiner():BinaryOperator<A>
- finisher():Function<A,R>
- characteristics():Set<Characteristics>

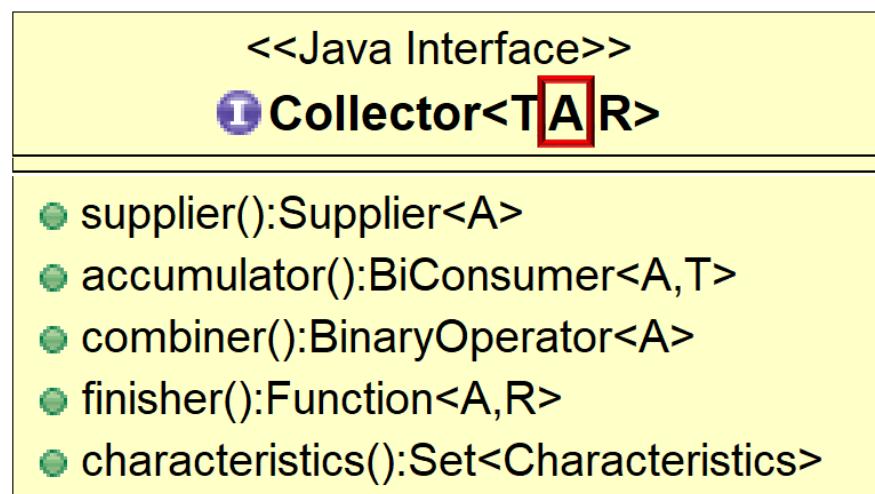
Implementing a Non-Concurrent Collector

- The Collector interface defines three generic types
 - **T** - The type of objects available in the stream
 - e.g., Integer, String, etc.



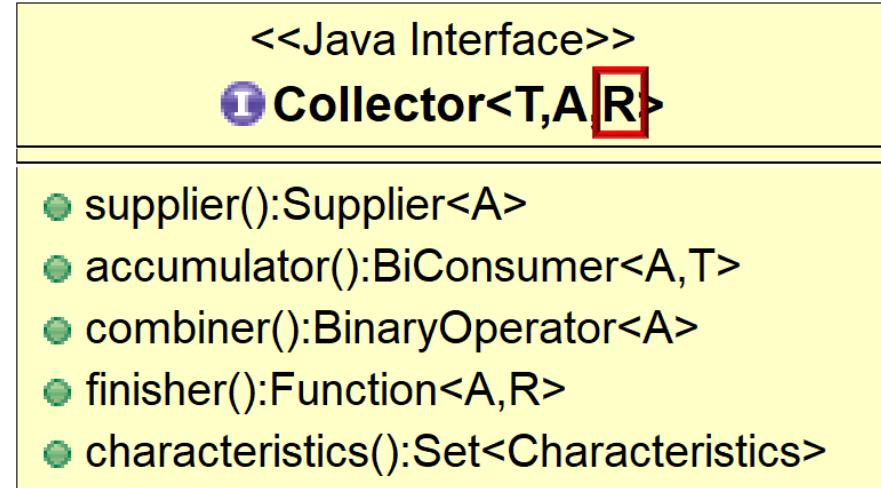
Implementing a Non-Concurrent Collector

- The Collector interface defines three generic types
 - **T**
 - **A** – The type of a mutable accumulator object for collection
 - e.g., List of T (implemented via ArrayList, LinkedList, etc.)



Implementing a Non-Concurrent Collector

- The Collector interface defines three generic types
 - T
 - A
 - R – The type of a final result
 - e.g., List of T



Implementing a Non-Concurrent Collector

- Five methods are defined in the Collector interface



<<Java Interface>>

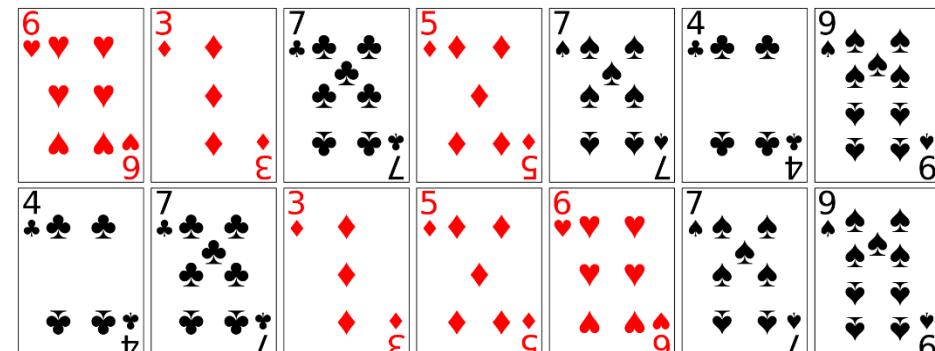
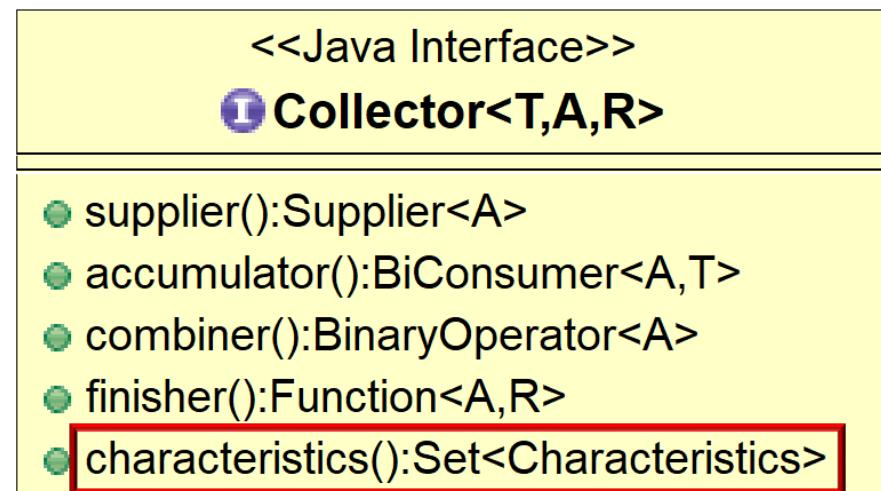
I Collector<T,A,R>

- supplier():Supplier<A>
- accumulator():BiConsumer<A,T>
- combiner():BinaryOperator<A>
- finisher():Function<A,R>
- characteristics():Set<Characteristics>

Again, this discussion assumes we're implementing a *non-concurrent* collector

Implementing a Non-Concurrent Collector

- Five methods are defined in the Collector interface
 - **characteristics()** - provides a stream with additional information used for internal optimizations, e.g.
 - UNORDERED
 - The collector need not preserve the encounter order



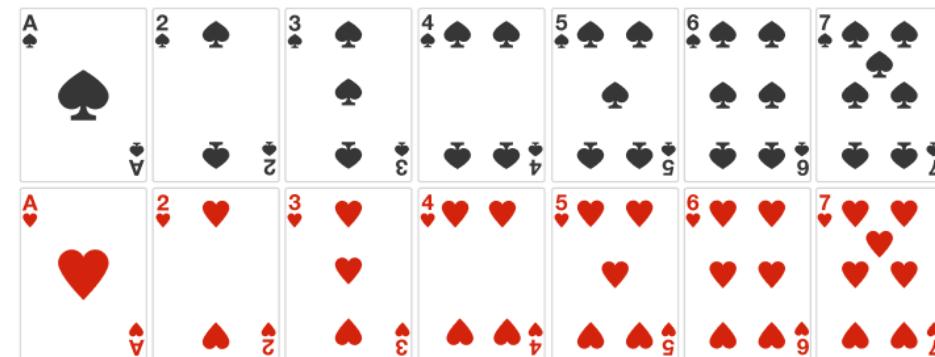
Implementing a Non-Concurrent Collector

- Five methods are defined in the Collector interface
 - **characteristics()** - provides a stream with additional information used for internal optimizations, e.g.
 - UNORDERED
 - The collector need not preserve the encounter order

<<Java Interface>>

I Collector<T,A,R>

- supplier():Supplier<A>
- accumulator():BiConsumer<A,T>
- combiner():BinaryOperator<A>
- finisher():Function<A,R>
- characteristics():Set<Characteristics>



A collector may preserve encounter order if it incurs no additional overhead

Implementing a Non-Concurrent Collector

- Five methods are defined in the Collector interface
 - **characteristics()** - provides a stream with additional information used for internal optimizations, e.g.
 - UNORDERED
 - IDENTITY_FINISH
 - The finisher() is the identity function so it can be a no-op
 - e.g., finisher() just returns null

<<Java Interface>>	
 Collector<T,A,R>	
• supplier():Supplier<A>	
• accumulator():BiConsumer<A,T>	
• combiner():BinaryOperator<A>	
• finisher():Function<A,R>	
• characteristics():Set<Characteristics>	



Implementing a Non-Concurrent Collector

- Five methods are defined in the Collector interface
 - **characteristics()** - provides a stream with additional information used for internal optimizations, e.g.
 - UNORDERED
 - IDENTITY_FINISH
 - CONCURRENT
 - The accumulator() method is called concurrently on the result container

<<Java Interface>>

I Collector<T,A,R>

- supplier():Supplier<A>
- accumulator():BiConsumer<A,T>
- combiner():BinaryOperator<A>
- finisher():Function<A,R>
- characteristics():Set<Characteristics>

The mutable result container must be synchronized!!



Implementing a Non-Concurrent Collector

- Five methods are defined in the Collector interface
 - **characteristics()** - provides a stream with additional information used for internal optimizations, e.g.
 - UNORDERED
 - IDENTITY_FINISH
 - CONCURRENT
 - The accumulator() method is called concurrently on the result container

<<Java Interface>>

I Collector<T,A,R>

- supplier():Supplier<A>
- accumulator():BiConsumer<A,T>
- combiner():BinaryOperator<A>
- finisher():Function<A,R>
- characteristics():Set<Characteristics>



We're focusing on a non-concurrent collector, which doesn't enable CONCURRENT

Implementing a Non-Concurrent Collector

- Five methods are defined in the Collector interface
 - **characteristics()** - provides a stream with additional information used for internal optimizations, e.g.

Any/all characteristics can be set using EnumSet.of()

```
Set characteristics () {  
    return Collections.unmodifiableSet  
        (EnumSet.of(Collector.Characteristics.CONCURRENT,  
                    Collector.Characteristics.UNORDERED,  
                    Collector.Characteristics.IDENTITY_FINISH)) ;  
}
```

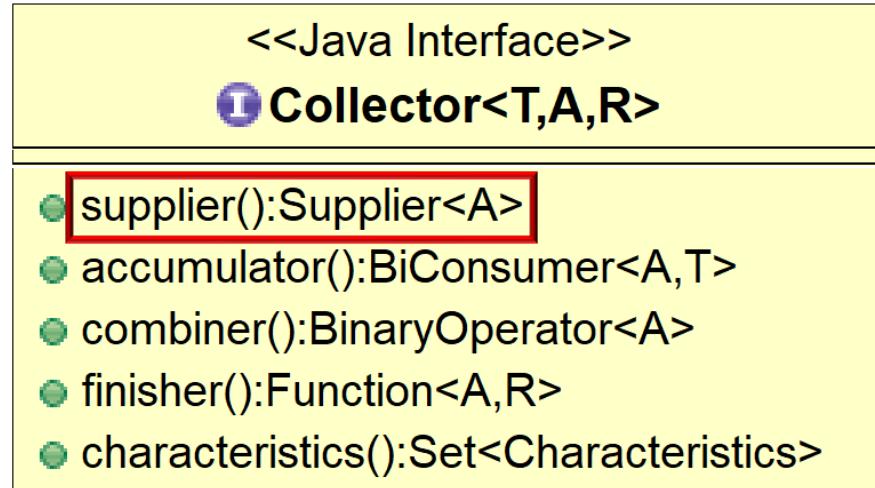
<<Java Interface>>
I Collector<T,A,R>

- supplier():Supplier<A>
- accumulator():BiConsumer<A,T>
- combiner():BinaryOperator<A>
- finisher():Function<A,R>
- characteristics():Set<Characteristics>

See docs.oracle.com/javase/8/docs/api/java/util/EnumSet.html

Implementing a Non-Concurrent Collector

- Five methods are defined in the Collector interface
 - **characteristics()**
 - **supplier()** - returns a supplier that acts as a factory to generate an empty result container



Implementing a Non-Concurrent Collector

- Five methods are defined in the Collector interface
 - **characteristics()**
 - **supplier()** - returns a supplier that acts as a factory to generate an empty result container, e.g.

```
Supplier<List> supplier() {  
    return ArrayList::new;  
}
```

<<Java Interface>>

I Collector<T,A,R>

- **supplier():Supplier<A>**
- **accumulator():BiConsumer<A,T>**
- **combiner():BinaryOperator<A>**
- **finisher():Function<A,R>**
- **characteristics():Set<Characteristics>**

Implementing a Non-Concurrent Collector

- Five methods are defined in the Collector interface
 - **characteristics()**
 - **supplier()**
 - **accumulator()** - returns a bi-consumer that adds a new element to an existing result container, e.g.

```
BiConsumer<List, Integer> accumulator() {  
    return List::add;  
}
```

<<Java Interface>>	
	Collector<T,A,R>
●	supplier():Supplier<A>
●	accumulator():BiConsumer<A,T>
●	combiner():BinaryOperator<A>
●	finisher():Function<A,R>
●	characteristics():Set<Characteristics>

A non-concurrent collector needs no synchronization



Implementing a Non-Concurrent Collector

- Five methods are defined in the Collector interface
 - **characteristics()**
 - **supplier()**
 - **accumulator()**
 - **combiner()** - returns a binary operator that merges two result containers together, e.g.

```
BinaryOperator<List> combiner() {  
    return (one, another) -> {  
        one.addAll(another);  
        return one;  
    };  
}
```

<<Java Interface>>

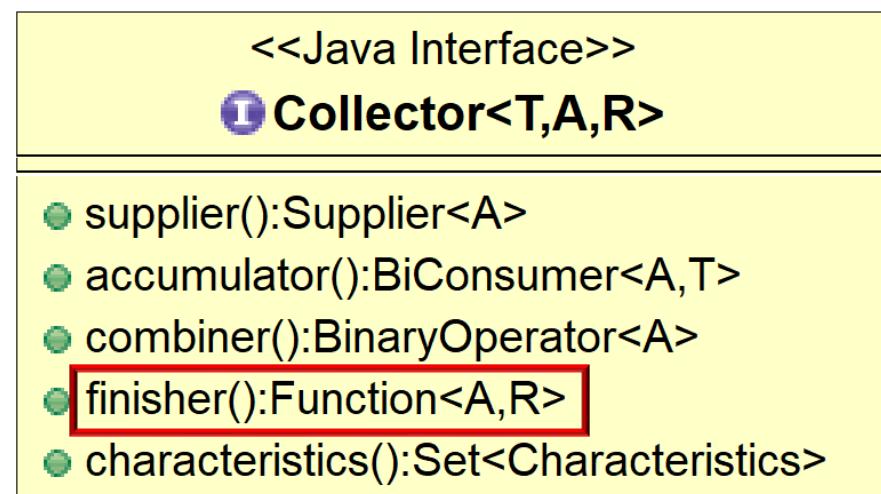
I Collector<T,A,R>

- supplier():Supplier<A>
- accumulator():BiConsumer<A,T>
- **combiner():BinaryOperator<A>**
- finisher():Function<A,R>
- characteristics():Set<Characteristics>

This combiner() will not be called for a sequential stream..

Implementing a Non-Concurrent Collector

- Five methods are defined in the Collector interface
 - **characteristics()**
 - **supplier()**
 - **accumulator()**
 - **combiner()**
 - **finisher()** - returns a function that converts the result container to final result type, e.g.
 - `return Function.identity()`



Implementing a Non-Concurrent Collector

- Five methods are defined in the Collector interface
 - **characteristics()**
 - **supplier()**
 - **accumulator()**
 - **combiner()**
 - **finisher()** - returns a function that converts the result container to final result type, e.g.
 - `return Function.identity()`
 - `return null;`

<<Java Interface>>

I Collector<T,A,R>

- supplier():Supplier<A>
- accumulator():BiConsumer<A,T>
- combiner():BinaryOperator<A>
- finisher():Function<A,R>
- characteristics():Set<Characteristics>



Should be a no-op if IDENTITY_FINISH characteristic is set

Implementing a Non-Concurrent Collector

- Five methods are defined in the Collector interface
 - **characteristics()**
 - **supplier()**
 - **accumulator()**
 - **combiner()**
 - **finisher()** - returns a function that converts the result container to final result type, e.g.
 - `return Function.identity();`
 - `return null;`

```
Stream
      .generate(() ->
          makeBigFraction
              (new Random(), false))
      .limit(sMAX_FRACTIONS)

      .map(reduceAndMultiplyFraction)
      .collect(FuturesCollector
          .toFuture()))
      .thenAccept
          (this::sortAndPrintList);
```

finisher() can also be much more interesting!

See [Java8/ex19/src/main/java/utils/FuturesCollector.java](https://github.com/Java8/ex19/blob/main/src/main/java/utils/FuturesCollector.java)

Implementing a Non-Concurrent Collector

- Collectors is a utility class whose factory methods create collectors for common collection types

Class Collectors

`java.lang.Object`
`java.util.stream.Collectors`

```
public final class Collectors
extends Object
```

Implementations of `Collector` that implement various useful reduction operations, such as accumulating elements into collections, summarizing elements according to various criteria, etc.

The following are examples of using the predefined collectors to perform common mutable reduction tasks:

Implementing a Non-Concurrent Collector

- Collectors is a utility class whose factory methods create collectors for common collection types
 - A utility class is a final class having only static methods, no state, & a private constructor

```
<<Java Class>>
Collectors

Collectors()
toCollection(Supplier<C>):Collector<T,?>,C>
toList():Collector<T,?,List<T>>
toSet():Collector<T,?,Set<T>>
joining():Collector<CharSequence,?,String>
joining(CharSequence):Collector<CharSequence,?,String>
joining(CharSequence,CharSequence,CharSequence):Collector<CharSequence,?,String>
mapping(Function<? super T,? extends U>,Collector<? super U,A,R>):Collector<T,?,R>
collectingAndThen(Collector<T,A,R>,Function<R,RR>):Collector<T,A,RR>
counting():Collector<T,?,Long>
minBy(Comparator<? super T>):Collector<T,?,Optional<T>>
maxBy(Comparator<? super T>):Collector<T,?,Optional<T>>
summingInt(TolIntFunction<? super T>):Collector<T,?,Integer>
summingLong(ToLongFunction<? super T>):Collector<T,?,Long>
summingDouble(ToDoubleFunction<? super T>):Collector<T,?,Double>
averagingInt(TolIntFunction<? super T>):Collector<T,?,Double>
averagingLong(ToLongFunction<? super T>):Collector<T,?,Double>
averagingDouble(ToDoubleFunction<? super T>):Collector<T,?,Double>
reducing(T,BinaryOperator<T>):Collector<T,?,T>
reducing(BinaryOperator<T>):Collector<T,?,Optional<T>>
reducing(U,Function<? super T,? extends U>,BinaryOperator<U>):Collector<T,?,U>
groupingBy(Function<? super T,? extends K>):Collector<T,?,Map<K,List<T>>>
toMap(Function<? super T,? extends K>,Function<? super T,? extends U>):Collector<T,?,Map<K,U>>
summarizingInt(TolIntFunction<? super T>):Collector<T,?,IntSummaryStatistics>
summarizingLong(ToLongFunction<? super T>):Collector<T,?,LongSummaryStatistics>
summarizingDouble(ToDoubleFunction<? super T>):Collector<T,?,DoubleSummaryStatistics>
```

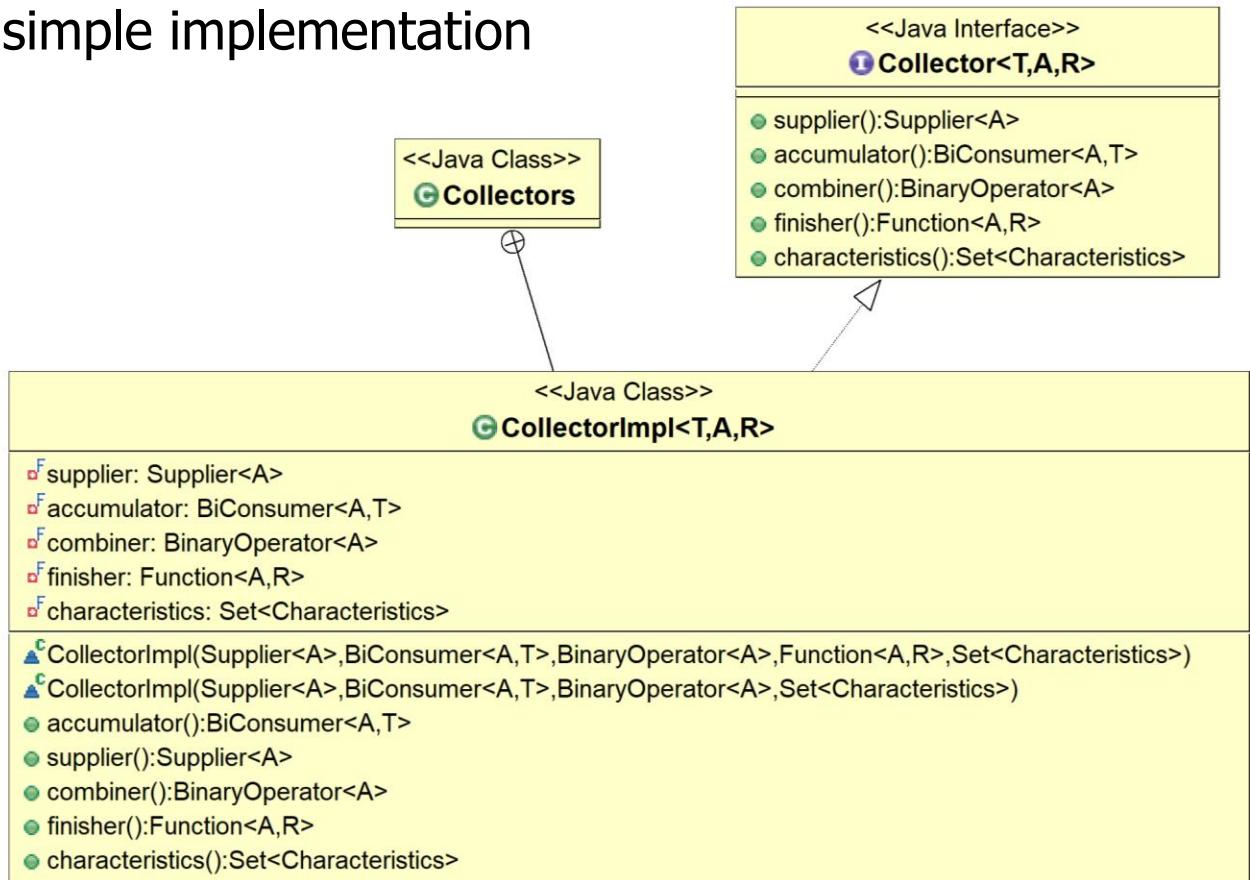
Implementing a Non-Concurrent Collector

- Collectors is a utility class whose factory methods create collectors for common collection types
 - A utility class is a final class having only static methods, no state, & a private constructor
 - `toList()` uses `CollectorImpl` to return a Collector that accumulates input elements into a new (ArrayList)

```
final class Collectors {  
    ...  
    public static <T> Collector  
        <T, ?, List<T>>  
    toList() {  
        return new CollectorImpl<>  
            ((Supplier<List<T>>)  
            ArrayList::new,  
            List::add,  
            (left, right) -> {  
                left.addAll(right);  
                return left;  
            },  
            CH_ID);  
    } ...
```

Implementing a Non-Concurrent Collector

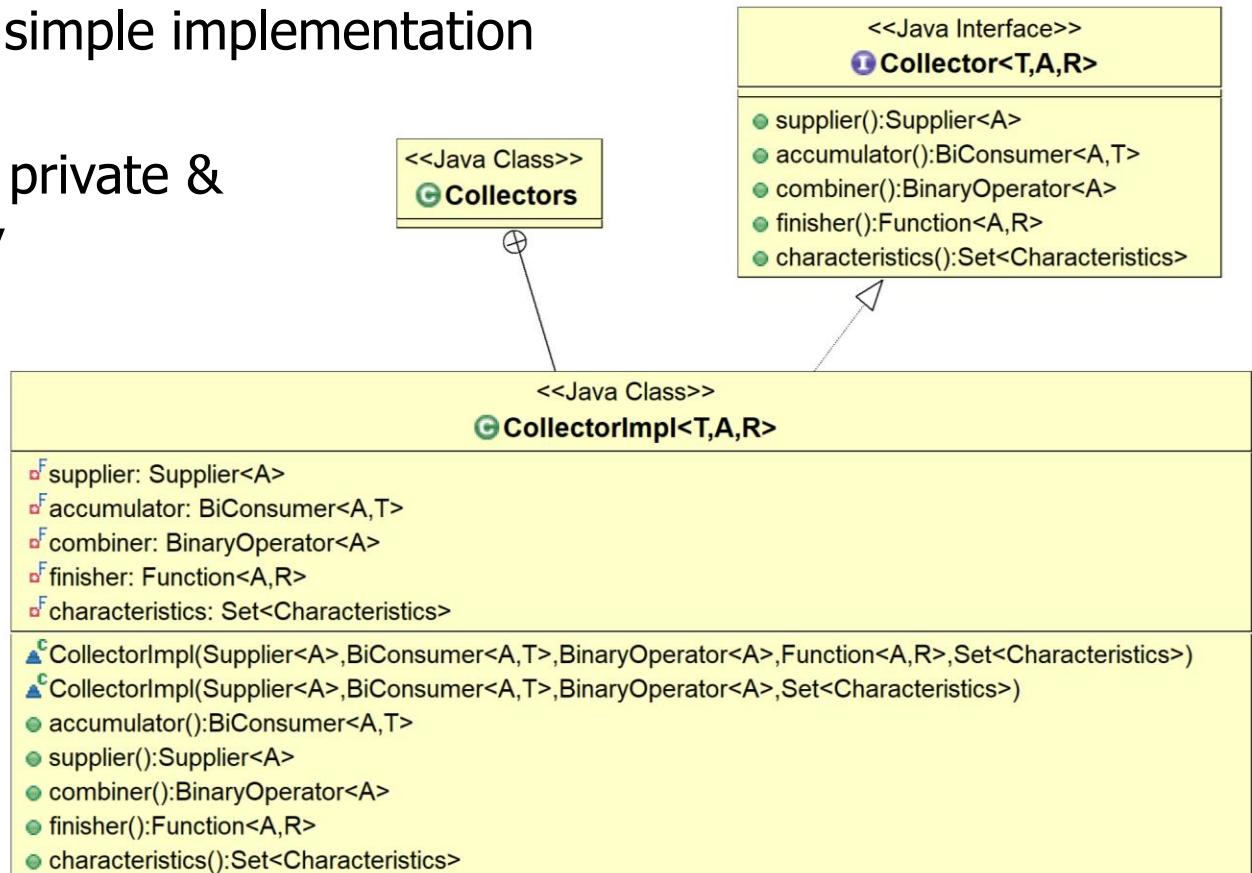
- CollectorImpl defines a simple implementation class for a Collector



See openjdk/8-b132/java/util/stream/Collectors.java#Collectors.CollectorImpl

Implementing a Non-Concurrent Collector

- CollectorImpl defines a simple implementation class for a Collector
 - However, this class is private & is only used internally



PRIVATE

Implementing a Non-Concurrent Collector

- `Collector.of()` defines a simple (public) factory method that implements a `Collector` using the (private) `CollectorImpl`

```
interface Collector<T, A, R> {  
    ...  
    static<T, R> Collector<T, R, R> of  
        (Supplier<R> supplier,  
         BiConsumer<R, T> accumulator,  
         BinaryOperator<R> combiner,  
         Characteristics... chars) {  
    ...  
    return new Collectors  
        .CollectorImpl<>  
        (supplier,  
         accumulator,  
         combiner,  
         chars);  
}
```

Implementing a Non-Concurrent Collector

- Collector.of() can also implement custom collectors that have pithy lambdas

```
public String toString() {  
    ...  
    mList.stream()  
        .collect(Collector.of(() -> new StringJoiner("|"),  
                           (j, r) -> j.add(r.toString()),  
                           StringJoiner::merge,  
                           StringJoiner::toString)); ...  
}
```



SearchResults's custom collector formats itself

See [SimpleSearchStream/src/main/java/search/SearchResults.java](#)

Implementing a Non-Concurrent Collector

- Complex custom collectors should implement the Collector interface



<<Java Interface>>

I Collector<T,A,R>

- supplier():Supplier<A>
- accumulator():BiConsumer<A,T>
- combiner():BinaryOperator<A>
- finisher():Function<A,R>
- characteristics():Set<Characteristics>



<<Java Class>>

G FuturesCollector<T>

- FuturesCollector()
- supplier():Supplier<List<CompletableFuture<T>>>
- accumulator():BiConsumer<List<CompletableFuture<T>>,CompletableFuture<T>>
- combiner():BinaryOperator<List<CompletableFuture<T>>>
- finisher():Function<List<CompletableFuture<T>>,CompletableFuture<List<T>>>
- characteristics():Set
- toFuture():Collector<CompletableFuture<T>,?,CompletableFuture<List<T>>>

See [Java8/ex19/src/main/java/utils/FuturesCollector.java](#)

Implementing a Non-Concurrent Collector

- More information on implementing custom collectors is available online

The screenshot shows a video player interface. At the top left is the logo for "iDays GÖTEBORG". Below it is the title "STREAMS IN JAVA 8 (PART 02/02): REDUCE VS COLLEC" and the subtitle "BREV. 1 / DAY 2 / 9 MARCH 2016 / 15:30-16:15" followed by "Angelika Langer, Angelika Langer Training & Consulting". To the right of the title is a video frame showing a woman speaking at a podium. On the right side of the screen is a slide titled "example – accumulator" containing the following Java code:

```
public void accumulate(String nextLine) {  
    if (nextLine != null) {  
        int indexOfLastEntry = result.size()-1;  
        if (indexOfLastEntry < 0) {  
            result.add(indexOfLastEntry+1,nextLine);  
        } else {  
            String current = result.get(indexOfLastEntry);  
            if (current.length() == 0)  
                result.add(indexOfLastEntry+1, nextLine);  
            else {  
                char endChar = current.charAt(current.length()-1);  
                if (endChar == '\\')  
                    result.set(indexOfLastEntry, current.substring  
                                (0, current.length()-1) + nextLine);  
                else  
                    result.add(indexOfLastEntry+1, nextLine);  
            } }  
}
```

At the bottom of the slide, there is a copyright notice: "© Copyright 2000-2016 by Angelika Langer & Klaus Klett. All Rights Reserved. http://www.angelikalanger.com/ last update: 19/3/2016, 15:05". To the right of the code, the text "reduce/collect (77)" is visible. The video player interface includes a play button, a progress bar showing "31:28 / 51:11", and various control icons like CC, HD, and volume.

See www.youtube.com/watch?v=H7VbRz9aj7c

End of Overview of Java 8 Streams (Part 5)

Java 8 Sequential Search Stream Gang

Example (Part 1)

Douglas C. Schmidt

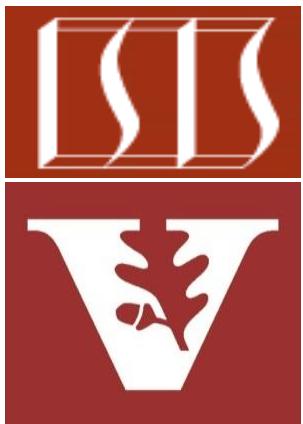
d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

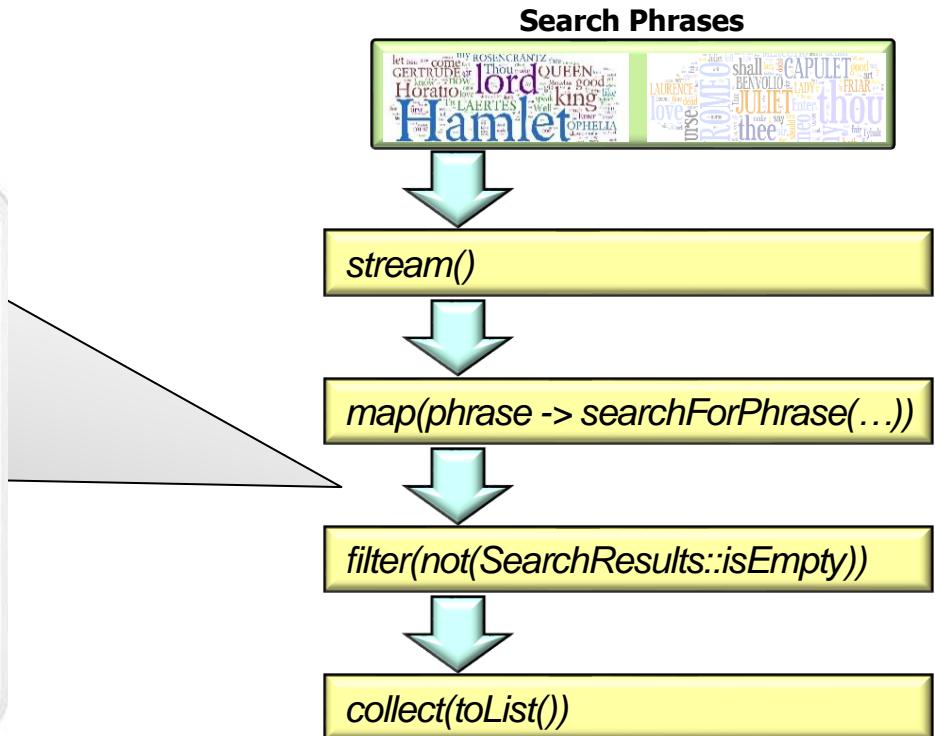
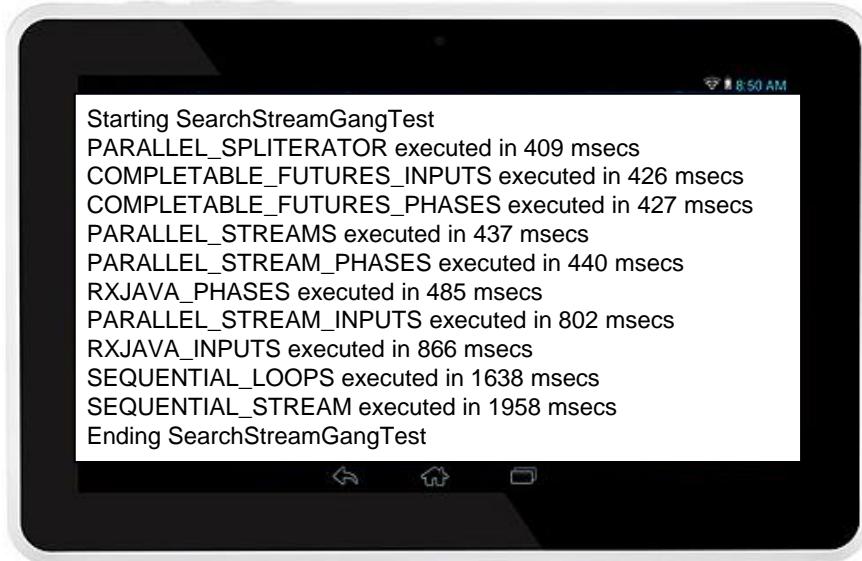
Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA



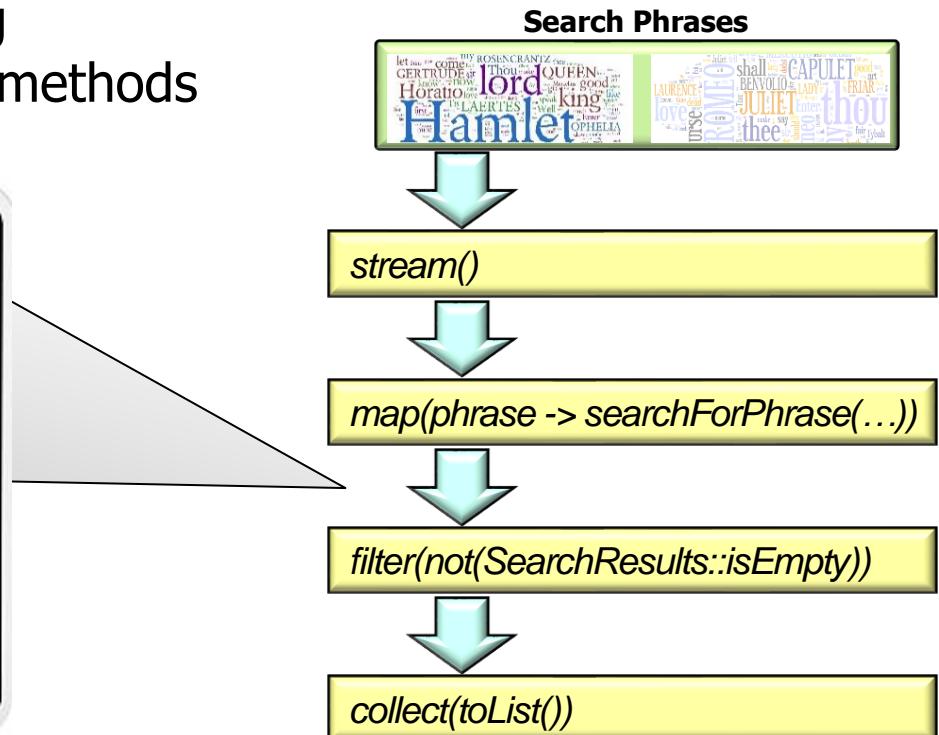
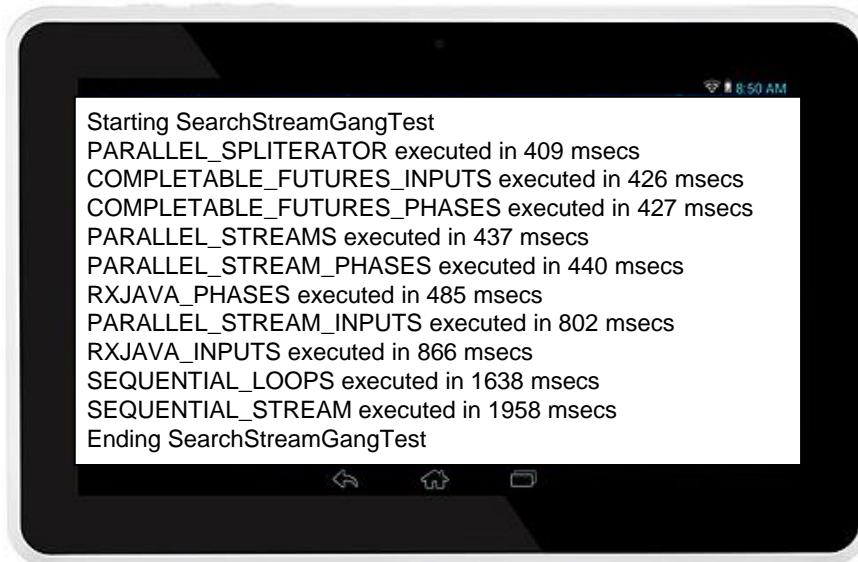
Learning Objectives in this Part of the Lesson

- Know how to apply sequential streams to the SearchStreamGang program



Learning Objectives in this Part of the Lesson

- Know how to apply sequential streams to the SearchStreamGang program
 - Understand the SearchStreamGang processStream() & processInput() methods



Learning Objectives in this Part of the Lesson

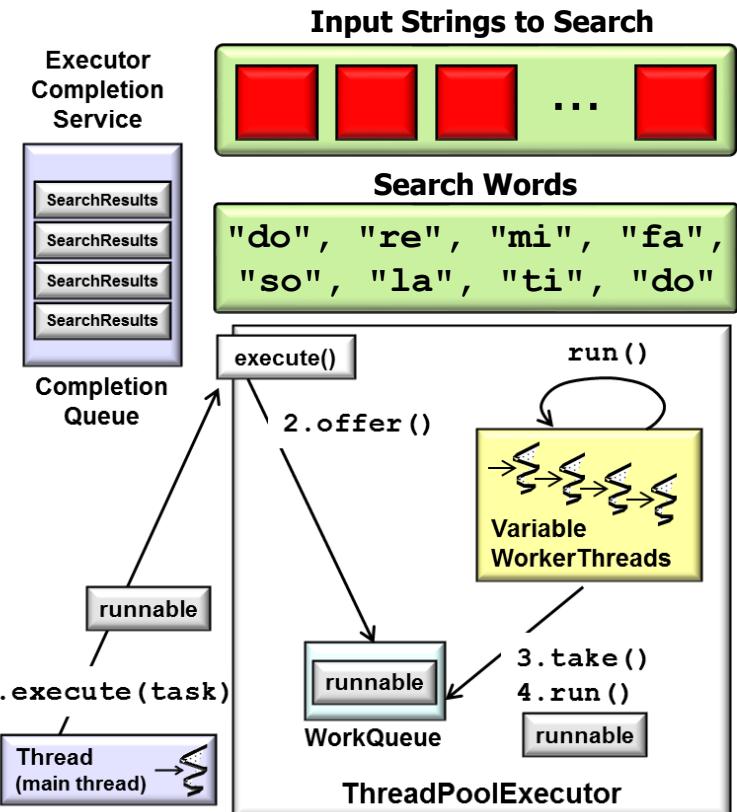
- Know how to apply sequential streams to the SearchStreamGang program
 - Understand the SearchStreamGang processStream() & processInput() methods
 - This program is more interesting than the SimpleSearchStream program



Overview of SearchStreamGang

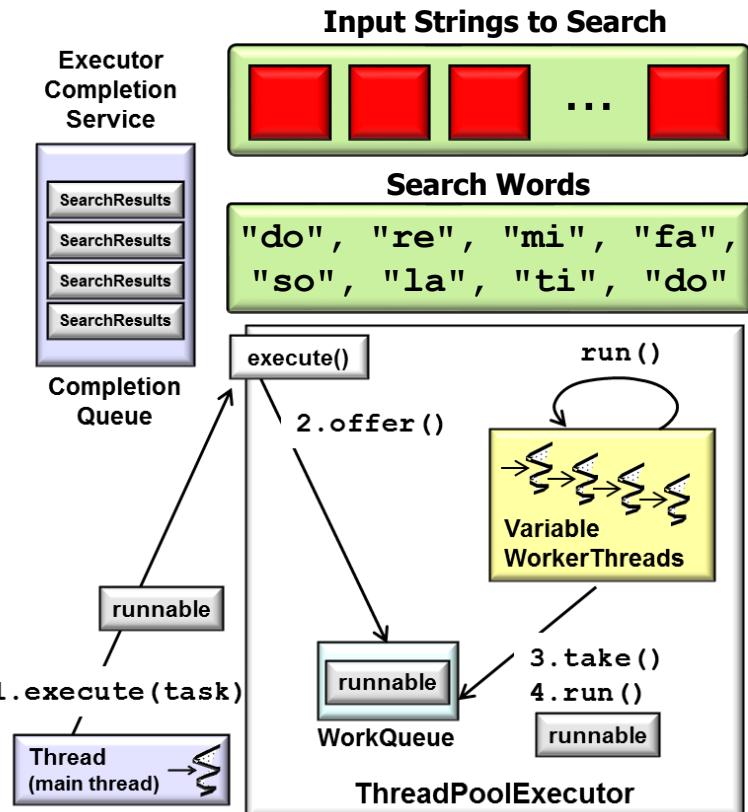
Overview of SearchStreamGang

- SearchStreamGang is a Java 8 revision of SearchTaskGang



Overview of SearchStreamGang

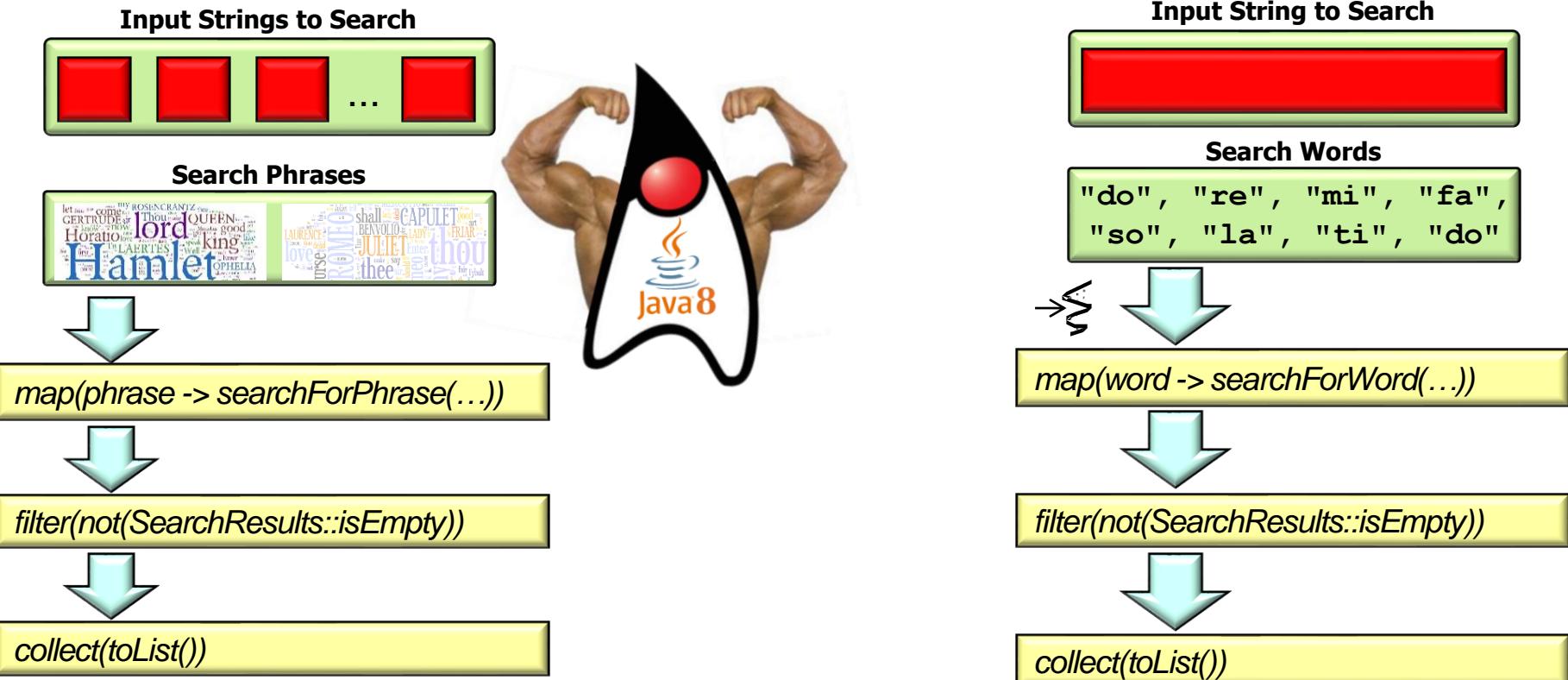
- SearchStreamGang is a Java 8 revision of SearchTaskGang
 - SearchTaskGang showcases the Java executor framework for tasks that are “embarrassingly parallel”



e.g., Executor, Executor Service, Executor Completion Service

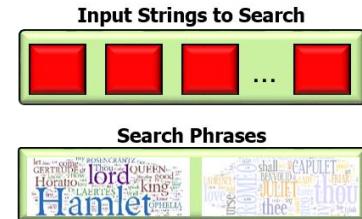
Overview of SearchStreamGang

- SearchStreamGang is a more powerful revision of SimpleSearchStream



Overview of SearchStreamGang

- SearchStreamGang is a more powerful revision of SimpleSearchStream, e.g.
 - It uses regular expressions to find phrases in works of Shakespeare



The Complete Works of William Shakespeare



Welcome to the Web's first edition of the Complete Works of William Shakespeare. This site has offered Shakespeare's plays and poetry to the Internet community since 1993.

For other Shakespeare resources, visit the [Mr. William Shakespeare and the Internet](#) Web site.

The original electronic source for this server was the Complete Moby(tm) Shakespeare. The HTML versions of the plays provided here are placed in the public domain.

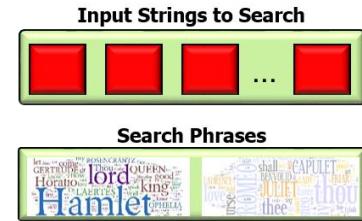
[Older news items](#)

See shakespeare.mit.edu

Overview of SearchStreamGang

- SearchStreamGang is a more powerful revision of SimpleSearchStream, e.g.
 - It uses regular expressions to find phrases in works of Shakespeare

"...
My liege, and madam, to expostulate
What majesty should be, what duty is,
Why day is day, night is night, and time is time.
Were nothing but to waste night, day, and time.
Therefore, since **brevity is the soul of wit**,
And tediousness the limbs and outward flourishes,
I will be brief. ..."



"Brevity is the soul of wit"

A phrase can match anywhere within a line

Overview of SearchStreamGang

- SearchStreamGang is a more powerful revision of SimpleSearchStream, e.g.
 - It uses regular expressions to find phrases in works of Shakespeare

"...
What's in a name? That which we call a rose
By any other name would smell as sweet.
So Romeo would, were he not Romeo call'd,
Retain that dear perfection which he owes
Without that title. ..."

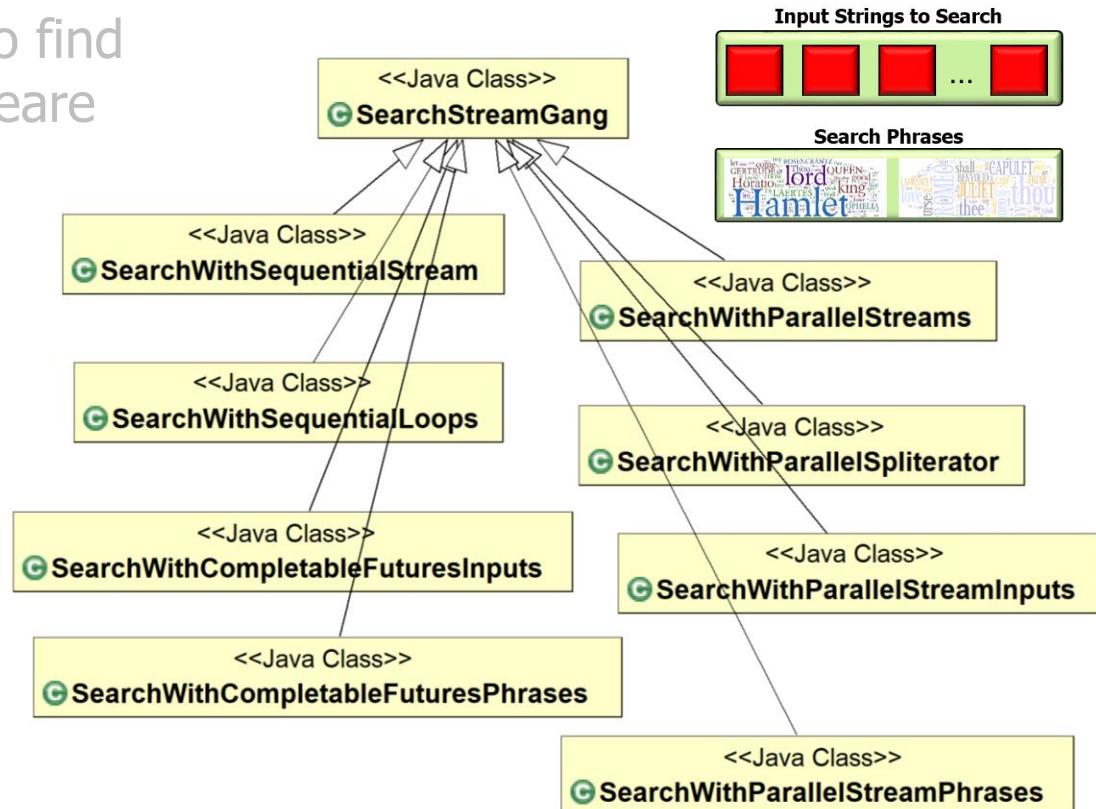


"What's in a name? That which we call a rose
By any other name would smell as sweet."

The phrases can also match across multiple lines

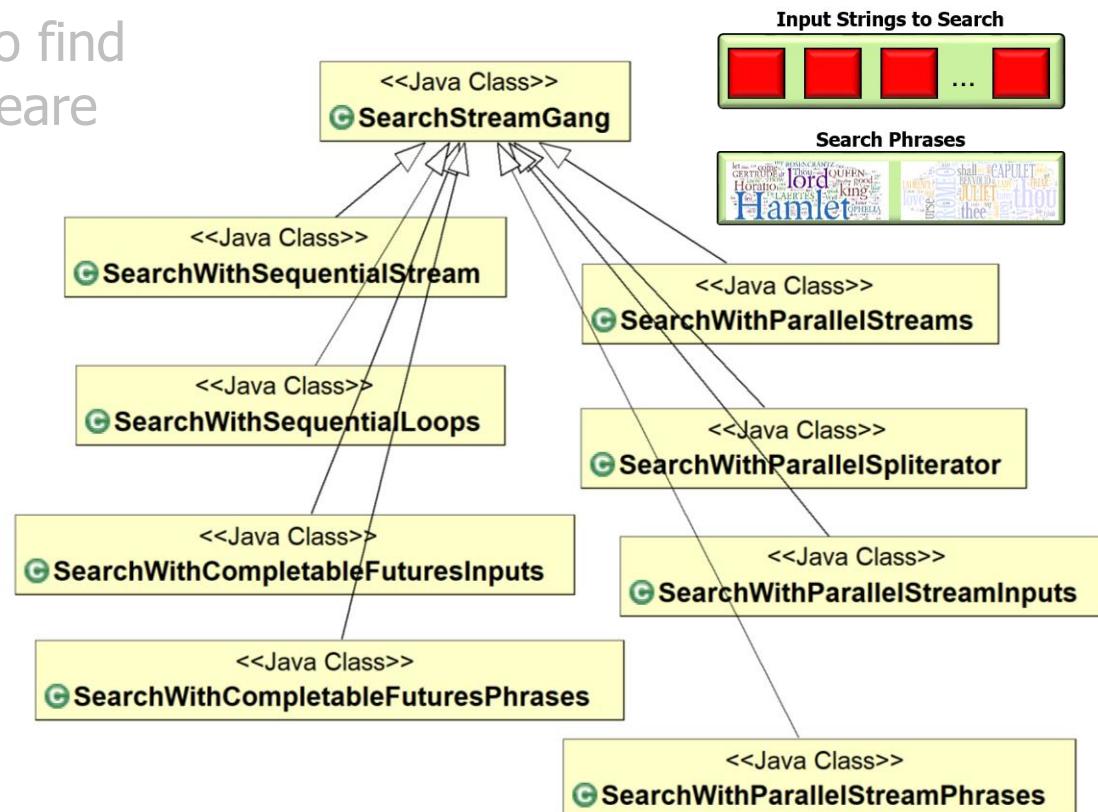
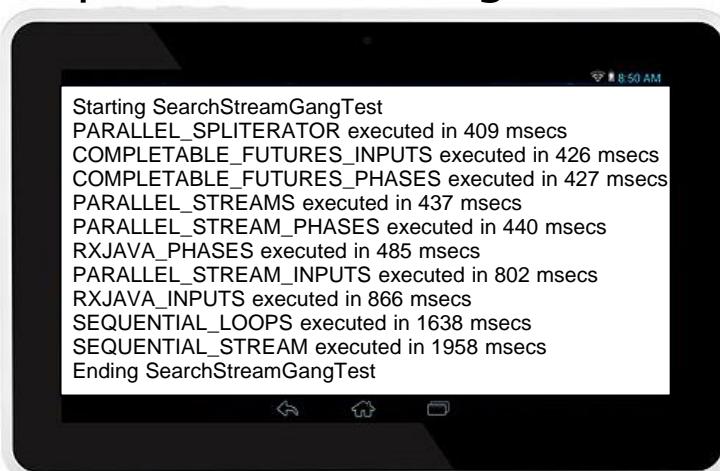
Overview of SearchStreamGang

- SearchStreamGang is a more powerful revision of SimpleSearchStream, e.g.
 - It uses regular expressions to find phrases in works of Shakespeare
 - It defines a framework for Java 8 concurrency & parallelism strategies



Overview of SearchStreamGang

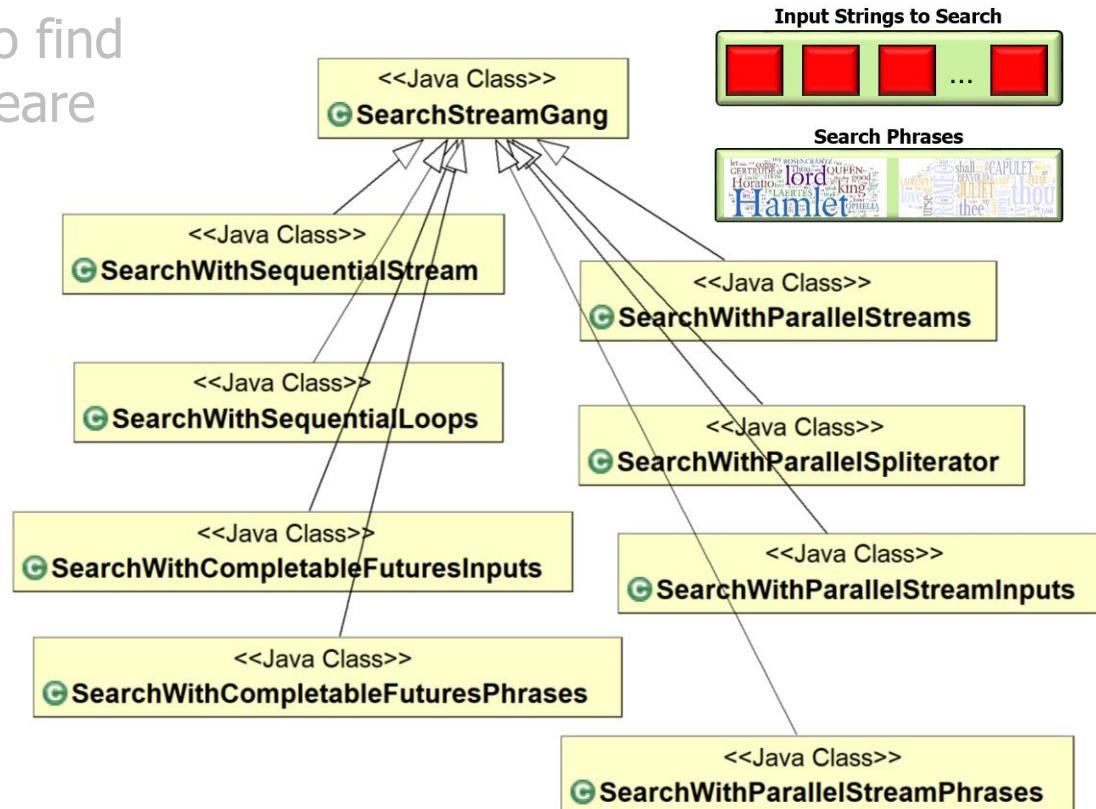
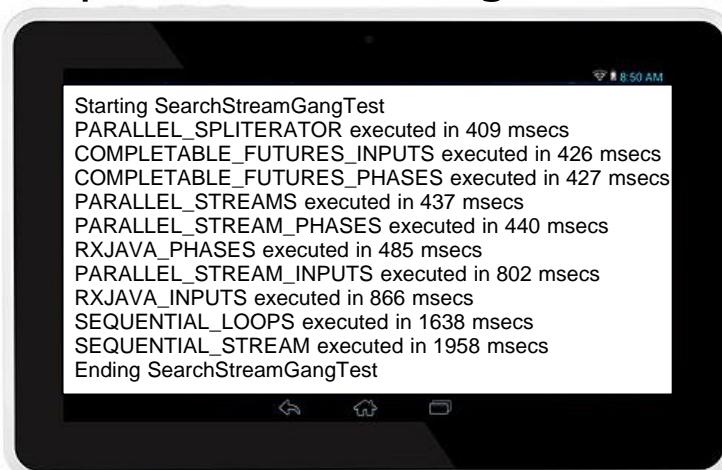
- SearchStreamGang is a more powerful revision of SimpleSearchStream, e.g.
 - It uses regular expressions to find phrases in works of Shakespeare
 - It defines a framework for Java 8 concurrency & parallelism strategies



This framework enables “apples-to-apples” performance comparisons

Overview of SearchStreamGang

- SearchStreamGang is a more powerful revision of SimpleSearchStream, e.g.
 - It uses regular expressions to find phrases in works of Shakespeare
 - It defines a framework for Java 8 concurrency & parallelism strategies



We'll cover Java 8 concurrency/parallel strategies after covering sequential streams

Applying Sequential Streams to SearchStreamGang

Applying Sequential Streams to SearchStreamGang

- We show aggregate operations in the SearchStreamGang's processStream() & processInput() methods

<<Java Class>>

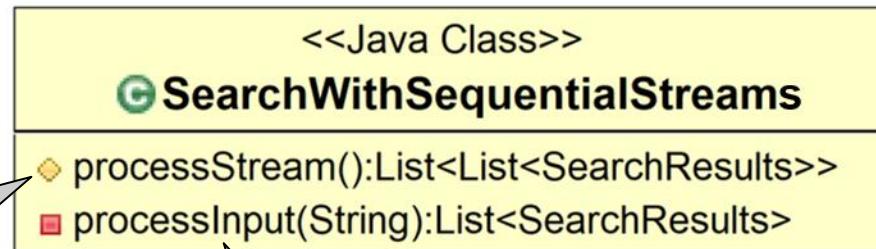
 **SearchWithSequentialStreams**

- ◆ processStream():List<List<SearchResults>>
- processInput(String):List<SearchResults>

Applying Sequential Streams to SearchStreamGang

- We show aggregate operations in the SearchStreamGang's processStream() & processInput() methods

```
getInput()
    .stream()
    .map(this::processInput)
    .collect(toList());
```



```
return mPhrasesToFind
    .stream()
    .map(phrase -> searchForPhrase(phrase, input, title, false))
    .filter(not(SearchResults::isEmpty))
    .collect(toList());
```

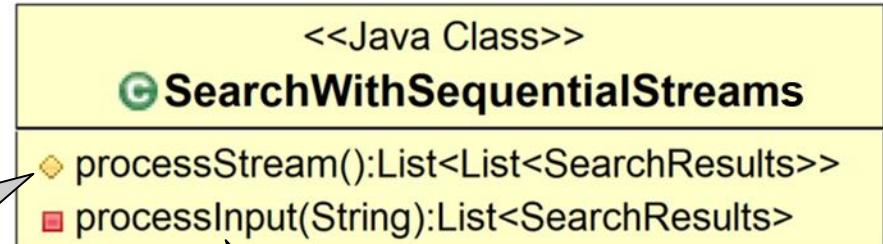
Applying Sequential Streams to SearchStreamGang

- We show aggregate operations in the SearchStreamGang's processStream() & processInput() methods

```
getInput()  
    .stream()  
    .map(this::processInput)  
    .collect(toList());
```

```
return mPhrasesToFind  
    .stream()  
    .map(phrase -> searchForPhrase(phrase, input, title, false))  
    .filter(not(SearchResults::isEmpty))  
    .collect(toList());
```

i.e., the map(), filter(), & collect() aggregate operations



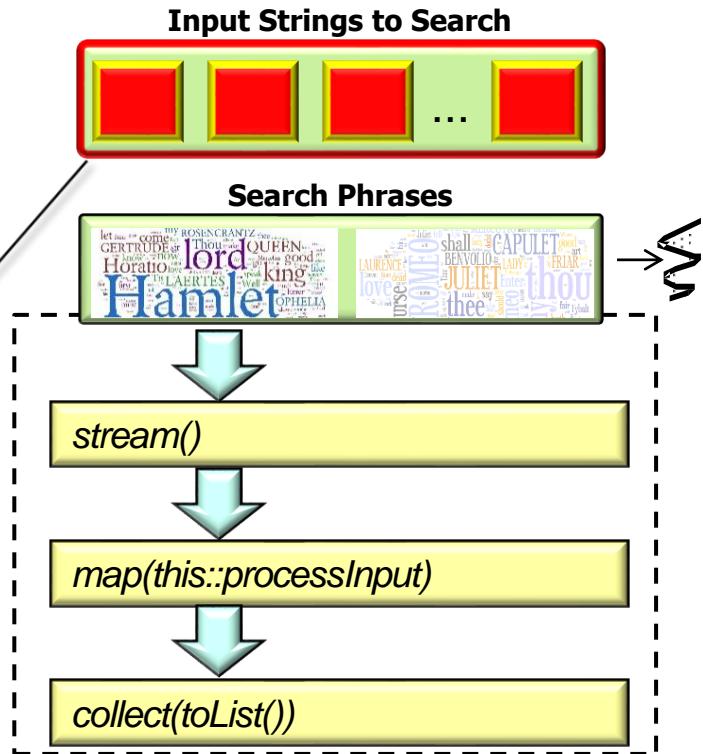
Applying Sequential Streams to SearchStreamGang

- We show aggregate operations in the SearchStreamGang's `processStream()` & `processInput()` methods

- processStream()**

- Uses a sequential stream to search a list of input strings in one thread

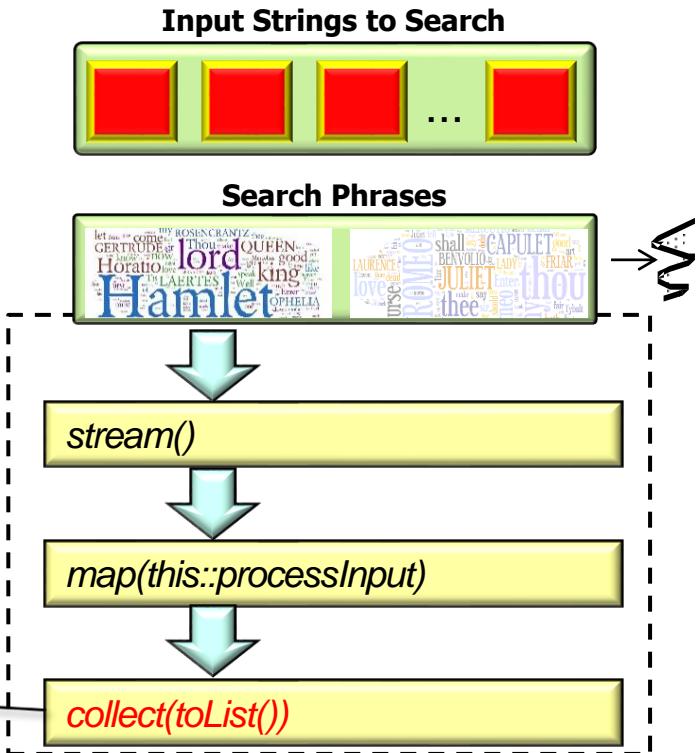
Each input string contains a work of Shakespeare (e.g., Hamlet, MacBeth, etc.)



Applying Sequential Streams to SearchStreamGang

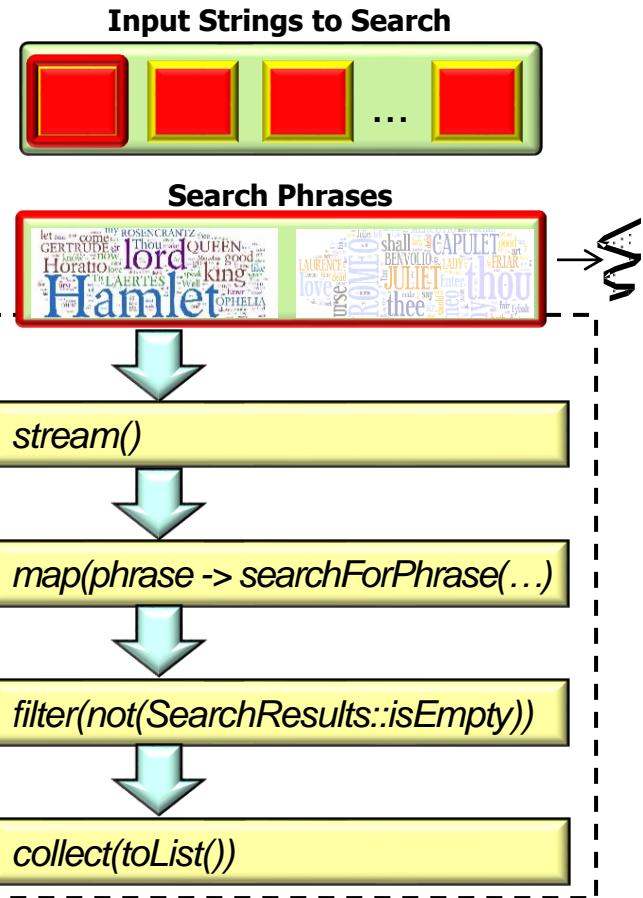
- We show aggregate operations in the SearchStreamGang's `processStream()` & `processInput()` methods
 - processStream()**
 - Uses a sequential stream to search a list of input strings in one thread

Returns a list of lists of `SearchResults`



Applying Sequential Streams to SearchStreamGang

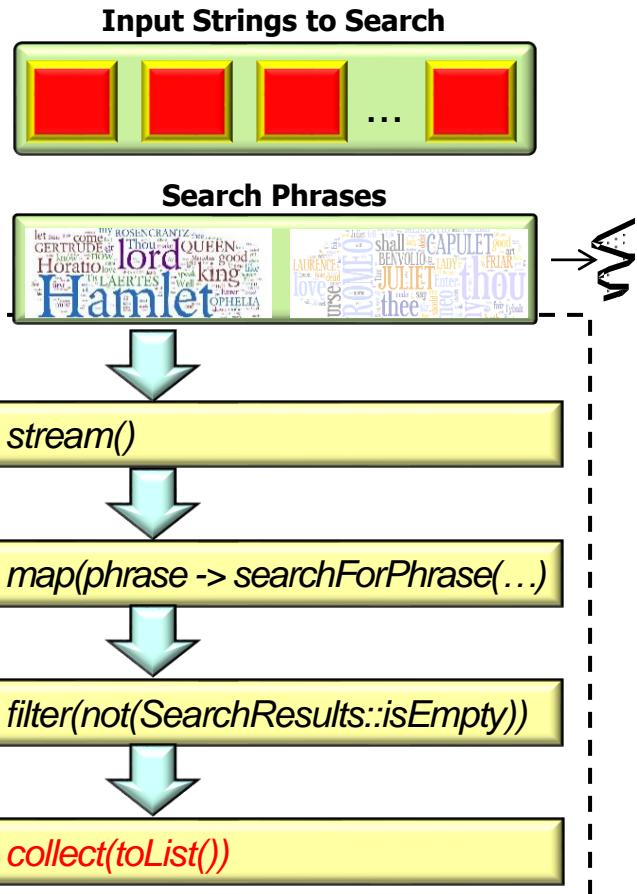
- We show aggregate operations in the SearchStreamGang's `processStream()` & `processInput()` methods
 - processStream()**
 - processInput()**
 - Uses a sequential stream to search a given input string & locate all the occurrences of phrases in one thread



Applying Sequential Streams to SearchStreamGang

- We show aggregate operations in the SearchStreamGang's `processStream()` & `processInput()` methods
 - processStream()**
 - processInput()**
 - Uses a sequential stream to search a given input string & locate all the occurrences of phrases in one thread

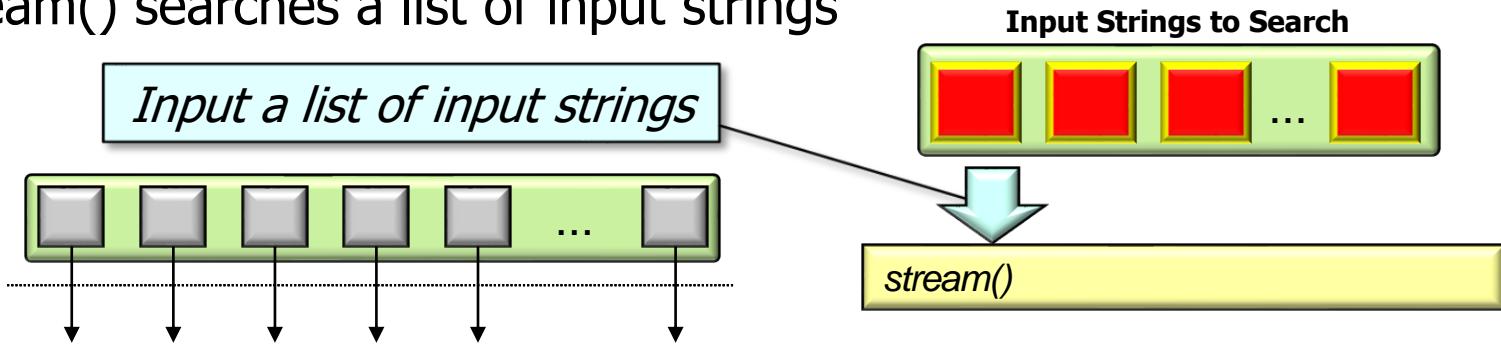
Returns a list of SearchResults



Visualizing processStream() & processInput()

Visualizing processStream() & processInput()

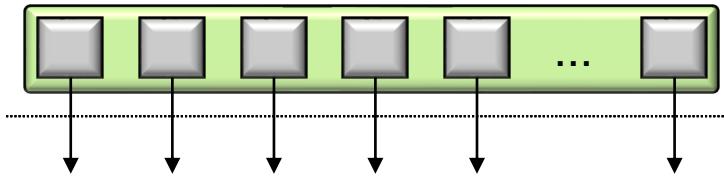
- processStream() searches a list of input strings



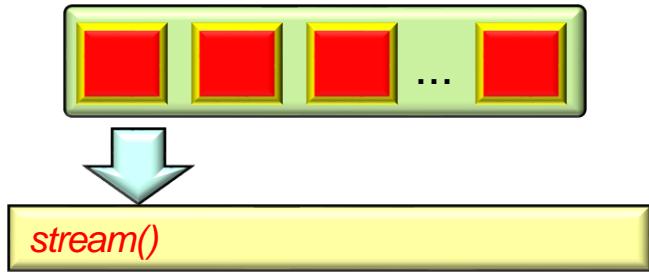
Visualizing processStream() & processInput()

- processStream() searches a list of input strings

List
<String>



Input Strings to Search



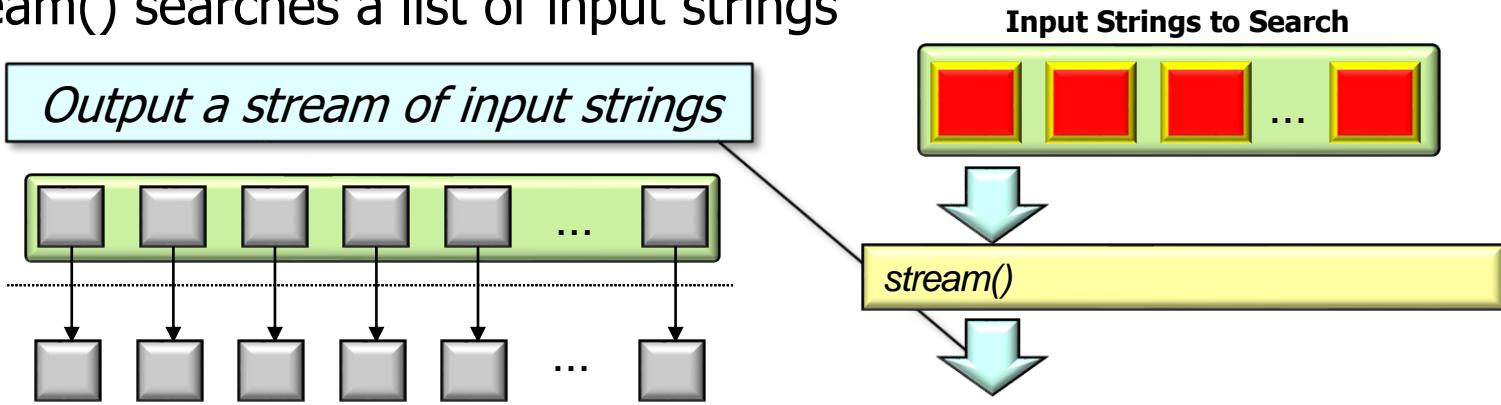
Convert collection to a (sequential) stream

Visualizing processStream() & processInput()

- processStream() searches a list of input strings

List
<String>

Stream
<String>

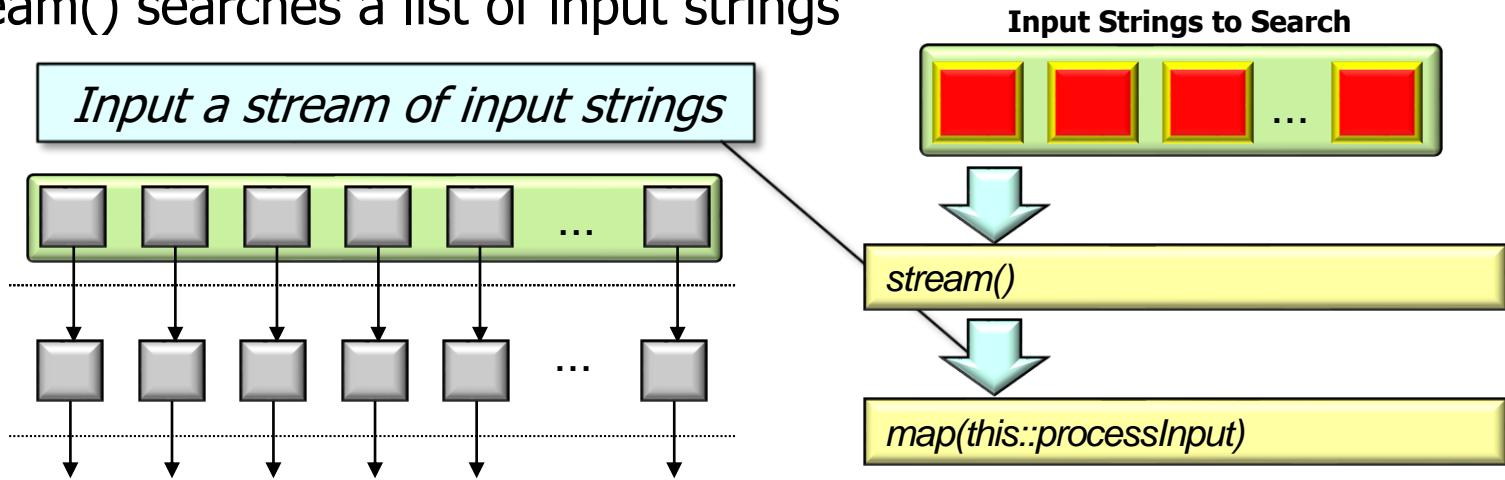


Visualizing processStream() & processInput()

- processStream() searches a list of input strings

List
<String>

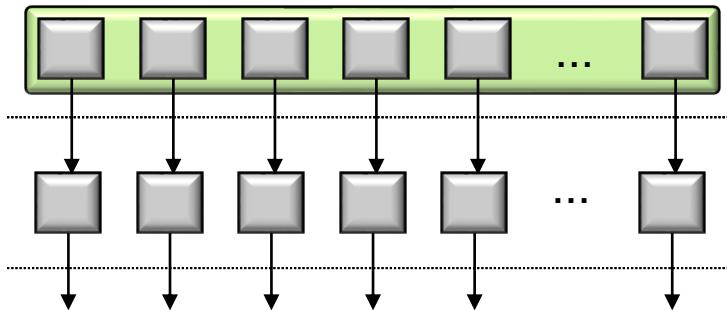
Stream
<String>



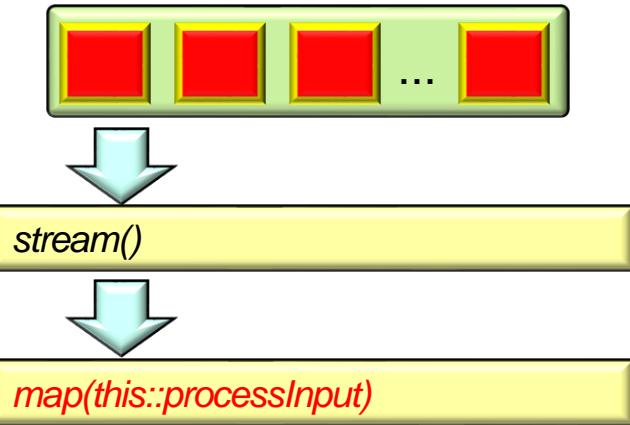
Visualizing processStream() & processInput()

- processStream() searches a list of input strings

List
<String>



Input Strings to Search



Call processInput() to search for phrases in each input string

Visualizing processStream() & processInput()

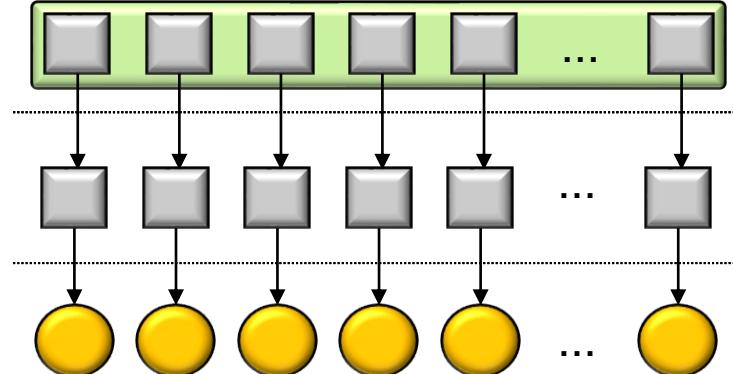
- processStream() searches a list of input strings

List
<String>

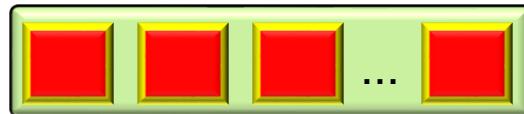
Stream
<String>

Stream<List
<SearchResults>>

Output a stream of lists of search results



Input Strings to Search

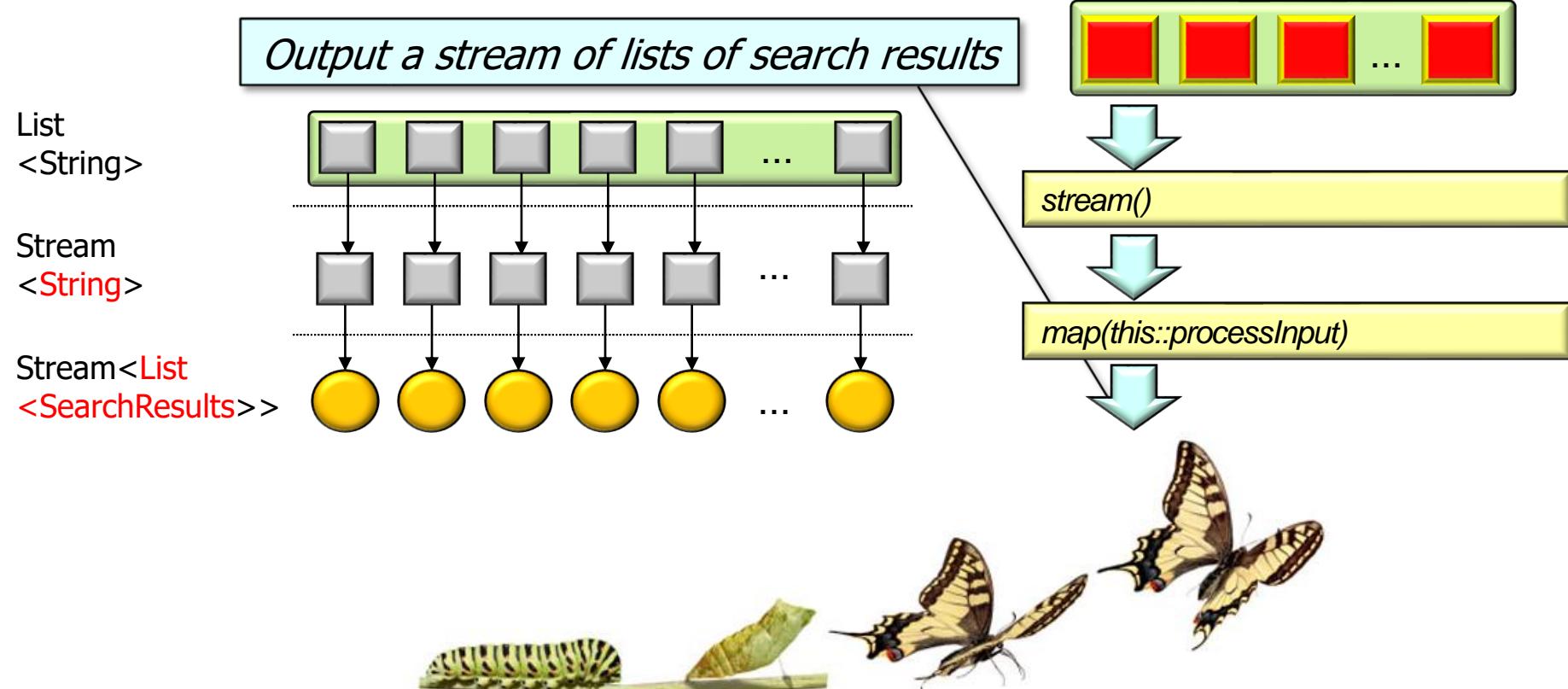


stream()

map(this::processInput)

Visualizing processStream() & processInput()

- processStream() searches a list of input strings



map() also transforms its input stream type into a different output stream type

Visualizing processStream() & processInput()

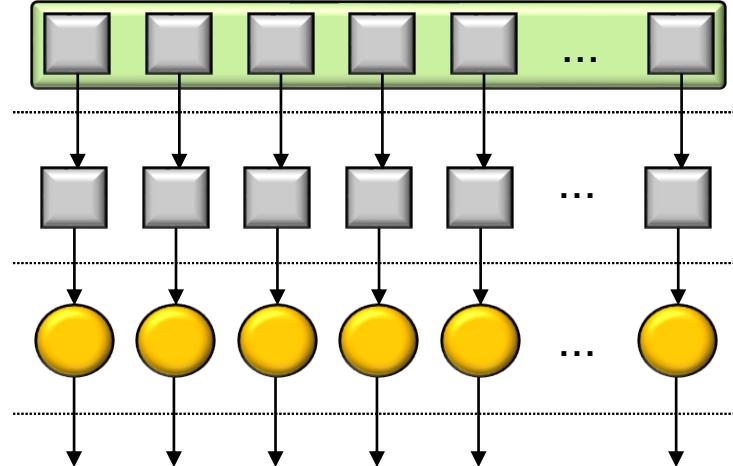
- processStream() searches a list of input strings

List
<String>

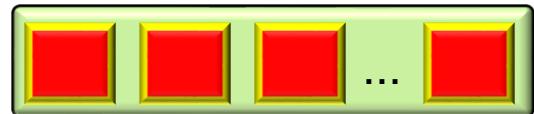
Stream
<String>

Stream<List
<SearchResults>>

Input a stream of lists of search results



Input Strings to Search



stream()



map(this::processInput)

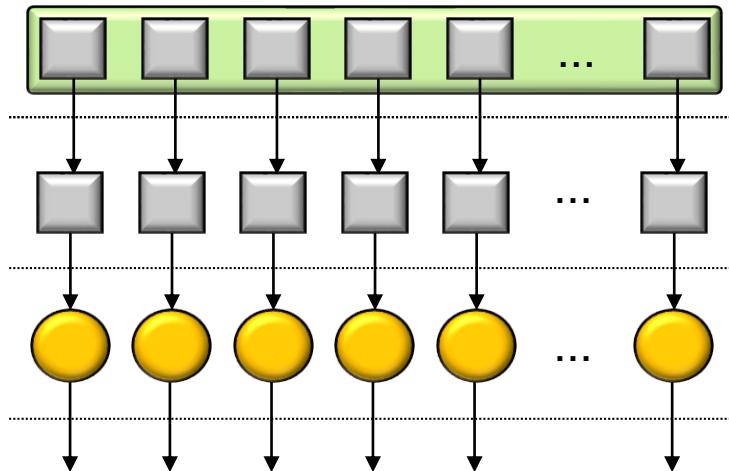


collect(toList())

Visualizing processStream() & processInput()

- processStream() searches a list of input strings

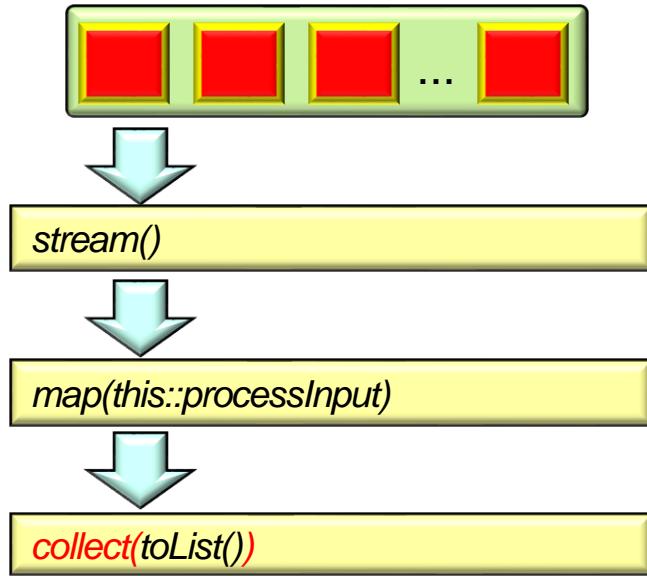
List
<String>



Stream
<String>

Stream<List
<SearchResults>>

Input Strings to Search

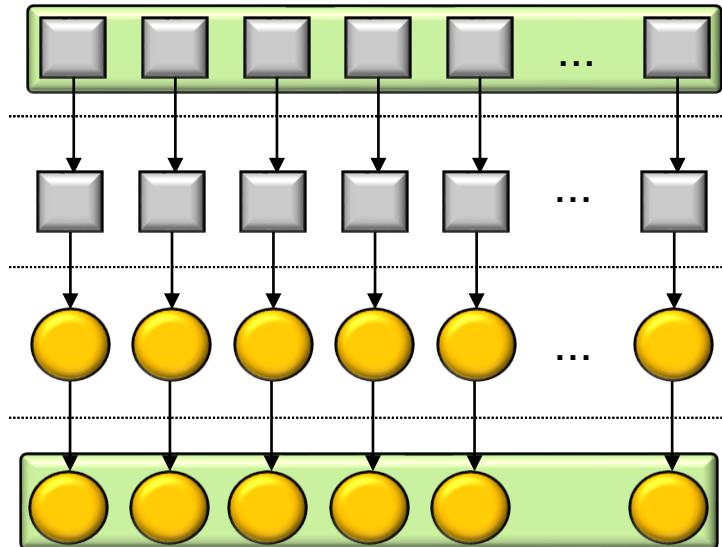


Trigger intermediate operation processing

Visualizing processStream() & processInput()

- processStream() searches a list of input strings

List
<String>

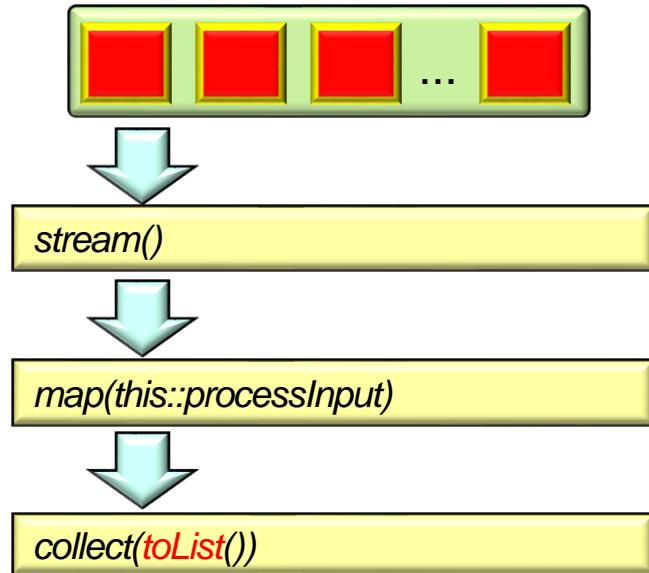


Stream
<String>

Stream<List
<SearchResults>>

List<List
<SearchResults>>

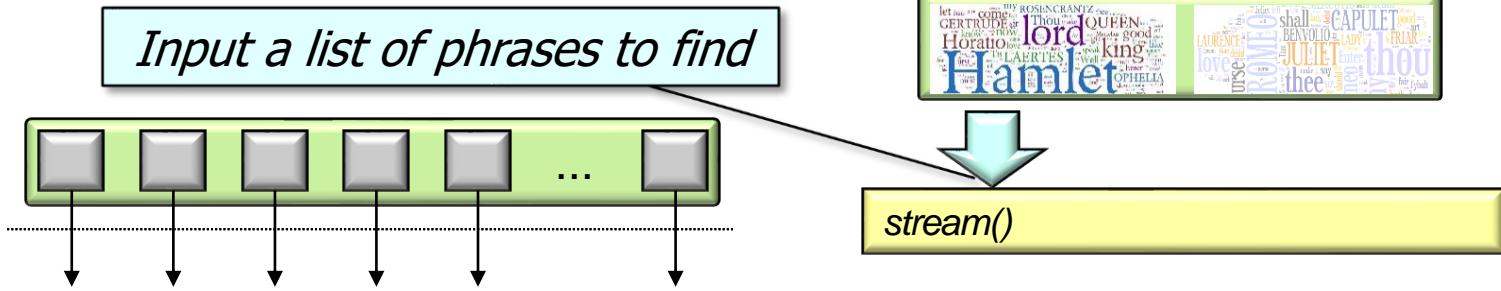
Input Strings to Search



Return a list of lists of search results based on “encounter order”

Visualizing processStream() & processInput()

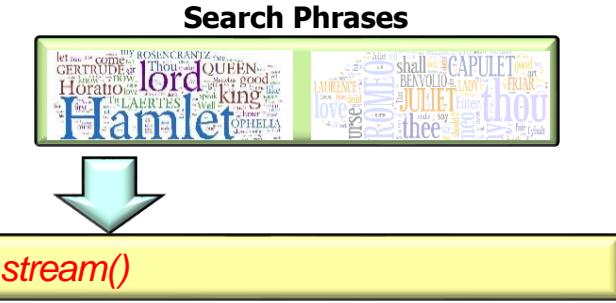
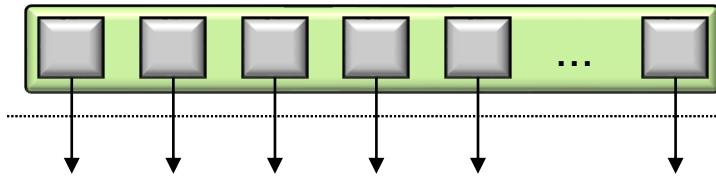
- processInput() finds phrases in an input string



Visualizing processStream() & processInput()

- processInput() finds phrases in an input string

List
<String>



Convert collection to a (sequential) stream

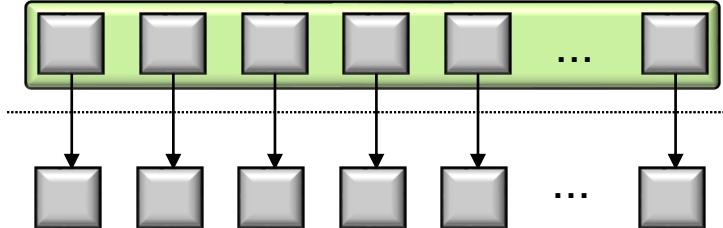
Visualizing processStream() & processInput()

- processInput() finds phrases in an input string

List
<String>

Stream
<String>

Output a stream of phrases to find



Search Phrases



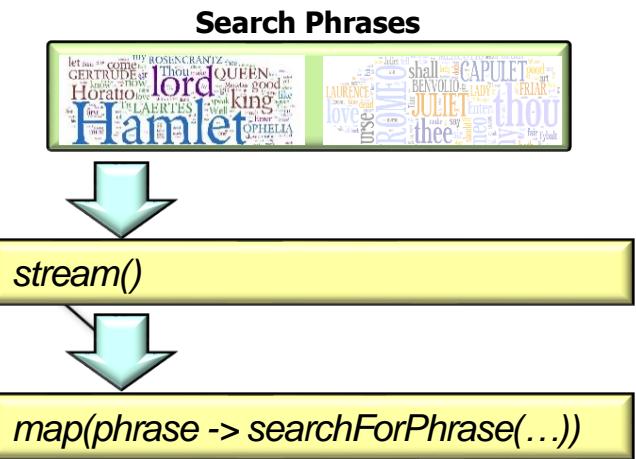
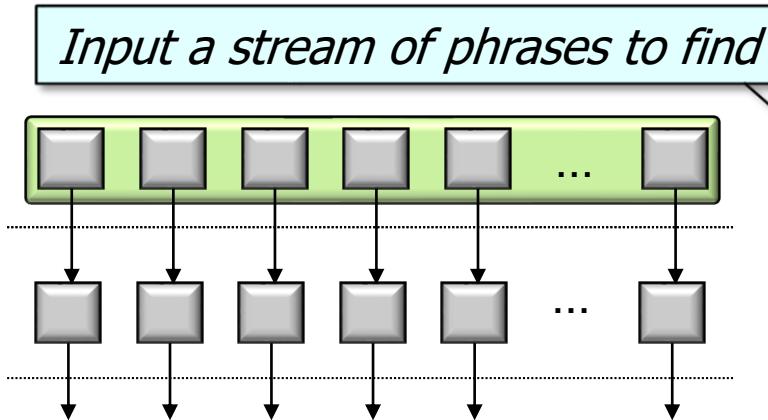
stream()

Visualizing processStream() & processInput()

- processInput() finds phrases in an input string

List
<String>

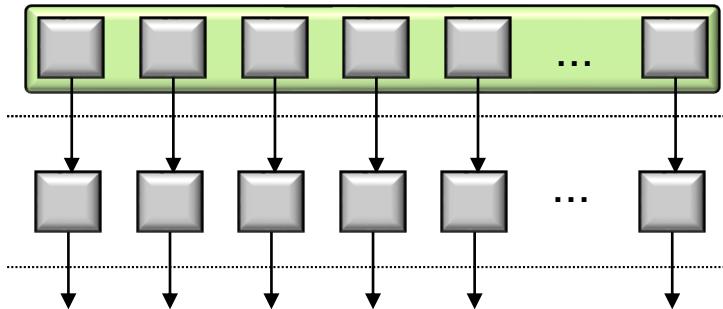
Stream
<String>



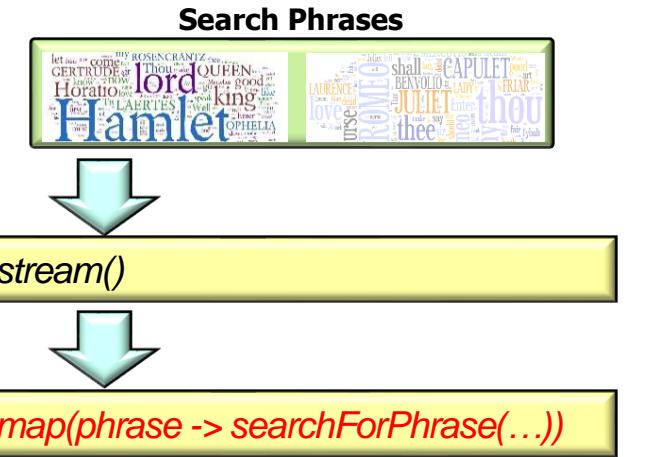
Visualizing processStream() & processInput()

- processInput() finds phrases in an input string

List
<String>



Stream
<String>



Search for the phrase in each input string

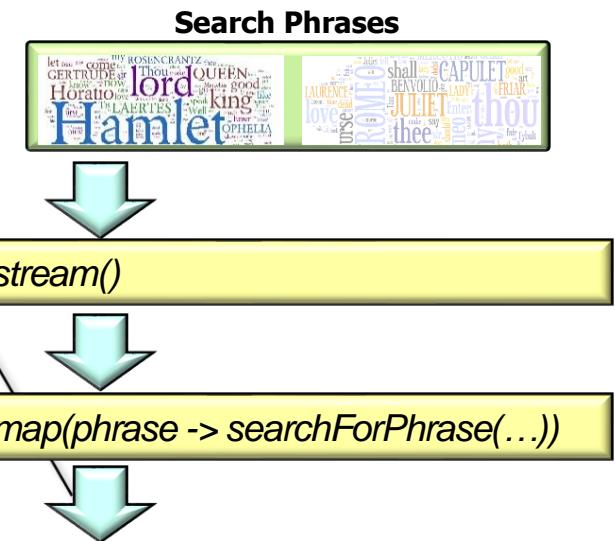
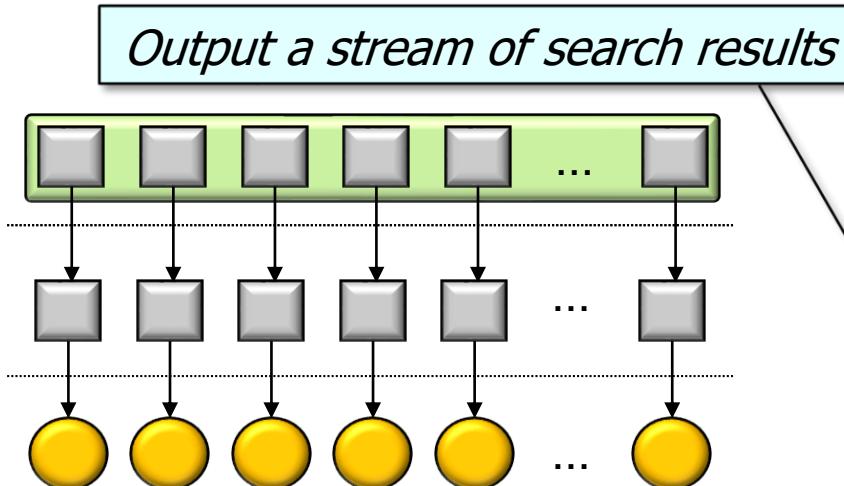
Visualizing processStream() & processInput()

- processInput() finds phrases in an input string

List
<String>

Stream
<String>

Stream
<SearchResults>



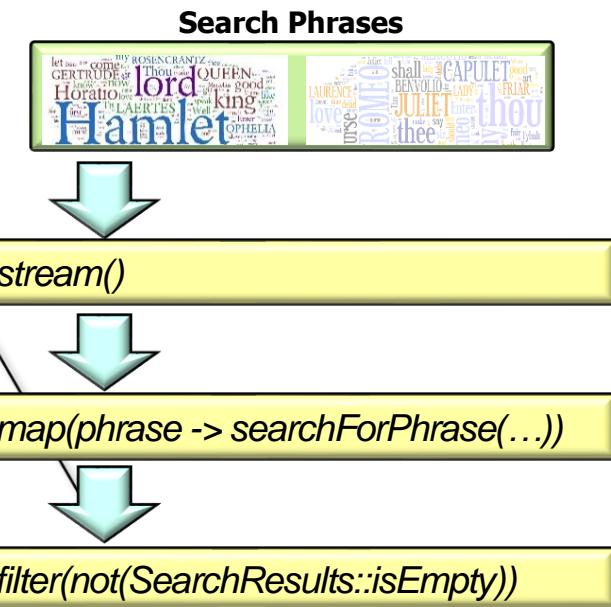
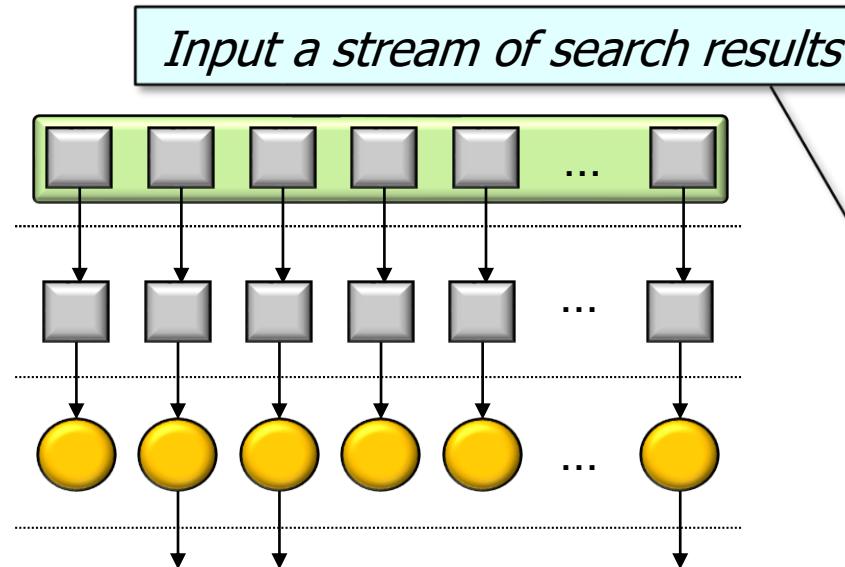
Visualizing processStream() & processInput()

- processInput() finds phrases in an input string

List
<String>

Stream
<String>

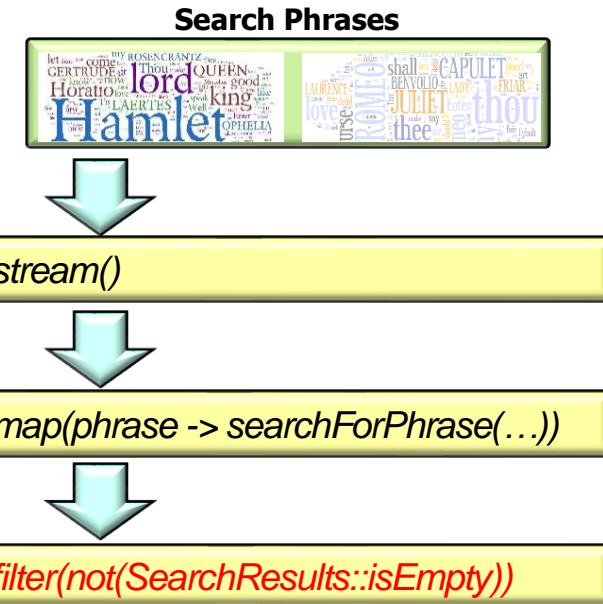
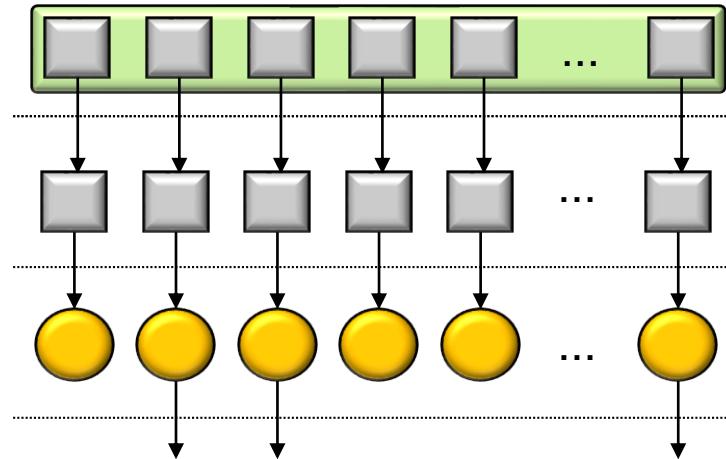
Stream
<SearchResults>



Visualizing processStream() & processInput()

- processInput() finds phrases in an input string

List
<String>



Remove empty search results from the stream

Visualizing processStream() & processInput()

- processInput() finds phrases in an input string

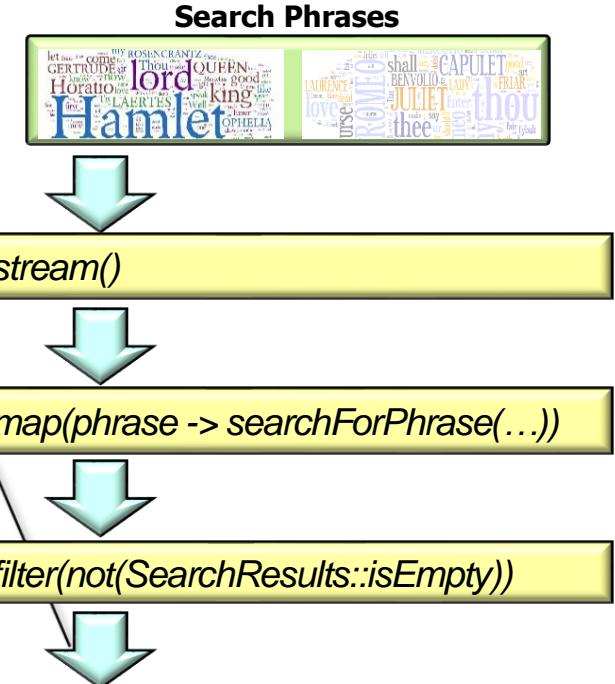
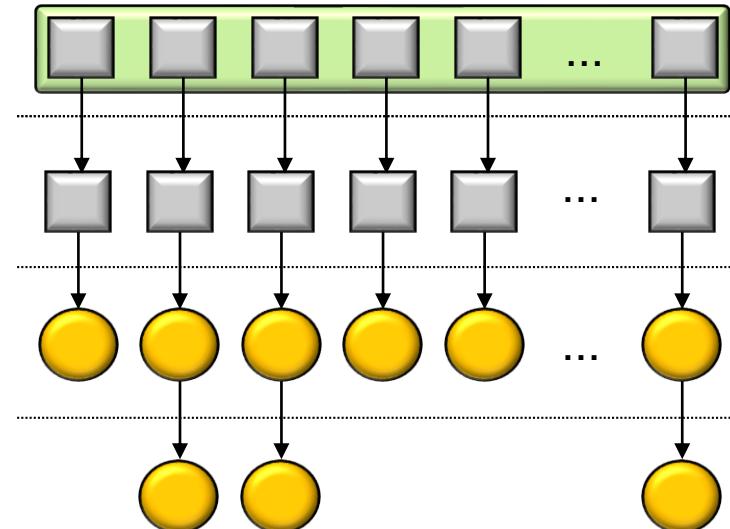
Output a stream of non-empty search results

List
<String>

Stream
<String>

Stream
<SearchResults>

Stream
<SearchResults>



Visualizing processStream() & processInput()

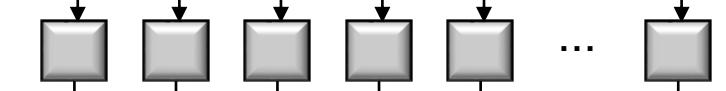
- processInput() finds phrases in an input string

Input a stream of non-empty search results

List
<String>



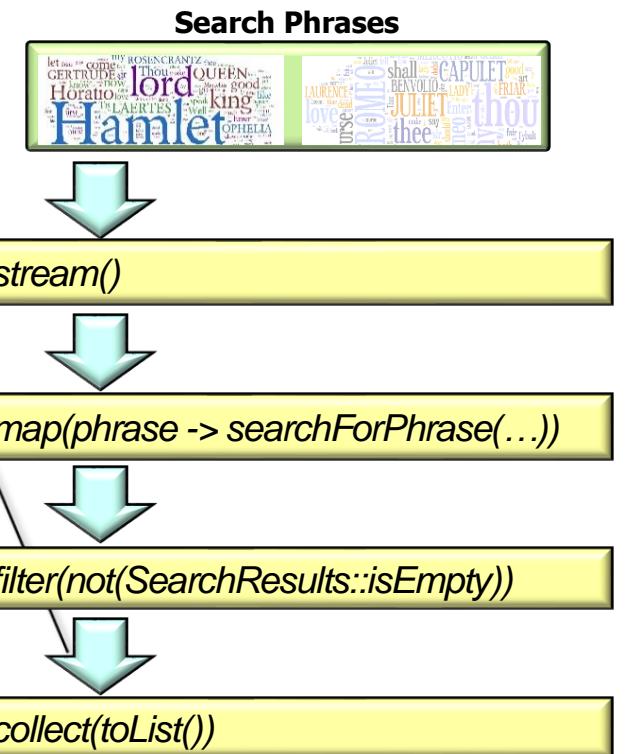
Stream
<String>



Stream
<SearchResults>



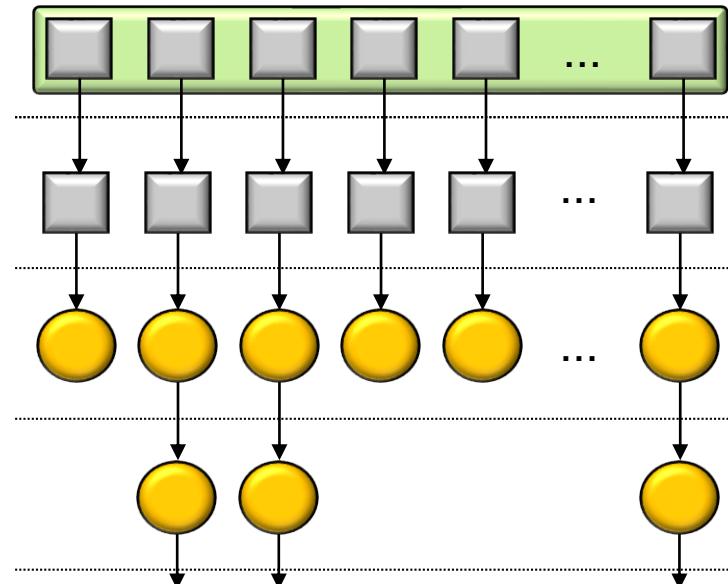
Stream
<SearchResults>



Visualizing processStream() & processInput()

- processInput() finds phrases in an input string

List
<String>

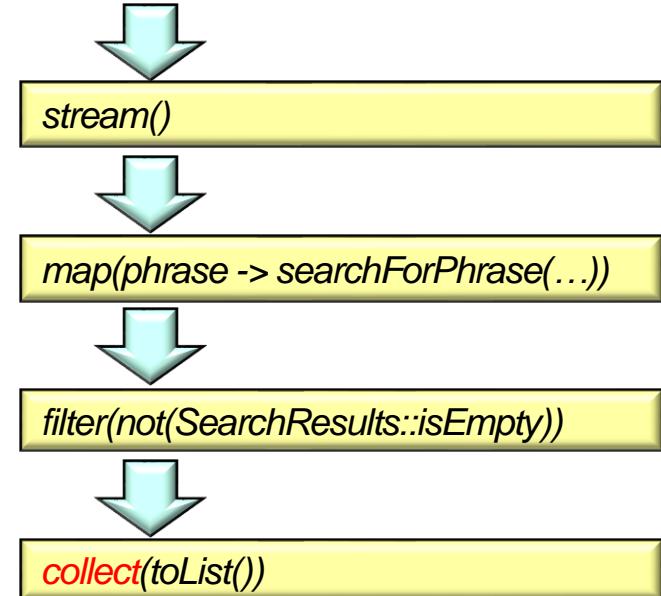


Stream
<String>

Stream
<SearchResults>

Stream
<SearchResults>

Search Phrases



Trigger intermediate operation processing

Visualizing processStream() & processInput()

- processInput() finds phrases in an input string

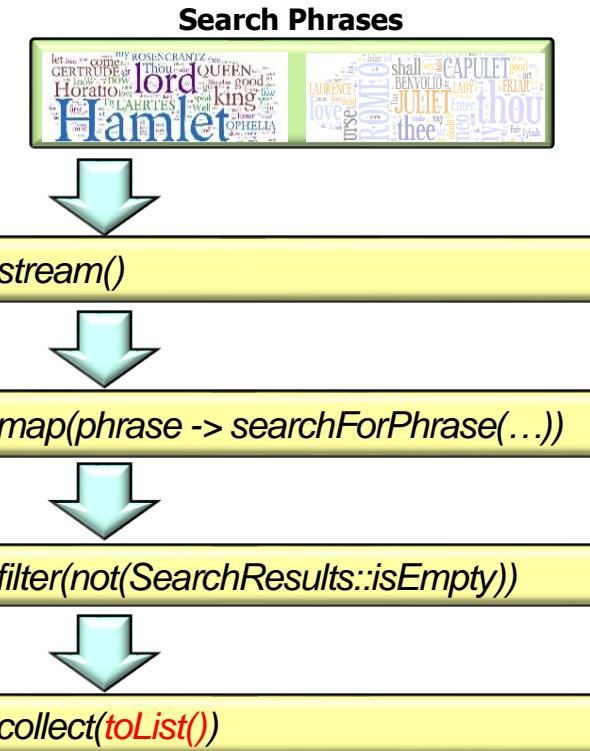
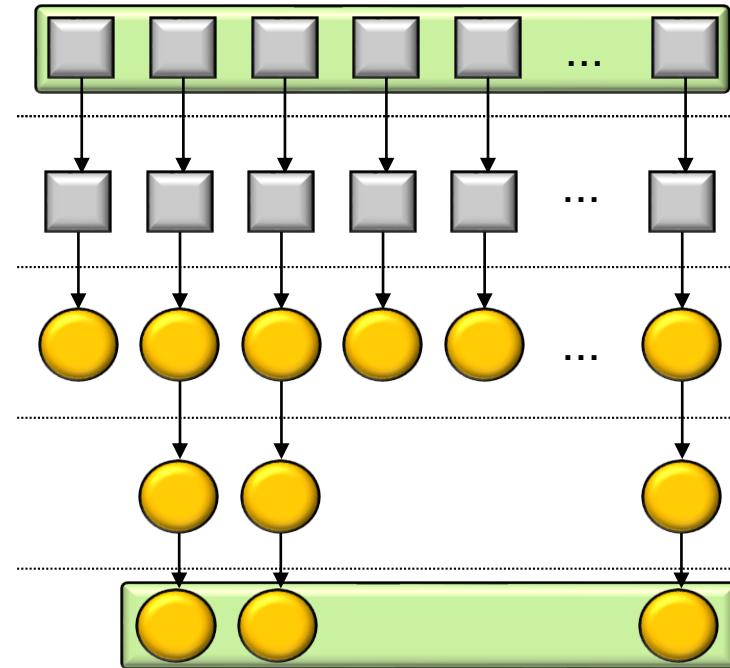
List
<String>

Stream
<String>

Stream
<SearchResults>

Stream
<SearchResults>

List
<SearchResults>



Return a (possibly empty) list of search results in encounter order

Visualizing processStream() & processInput()

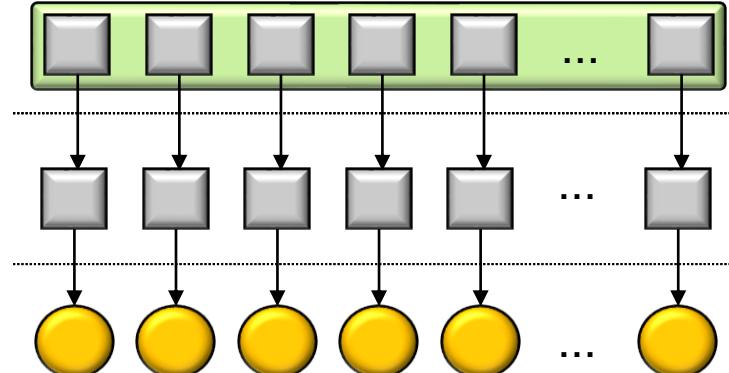
- processStream() searches a list of input strings

List
<String>

Stream
<String>

Stream<List
<SearchResults>>

Output a stream of lists of search results



Input Strings to Search

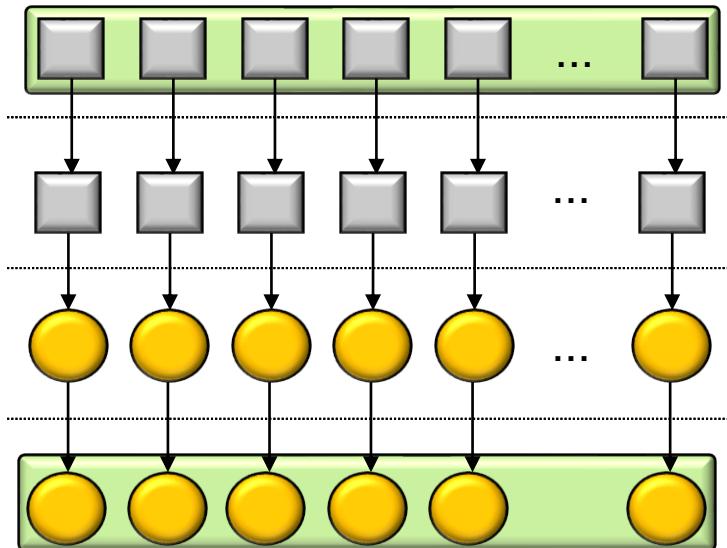


Some lists of search results may be empty if no phrases match an input string

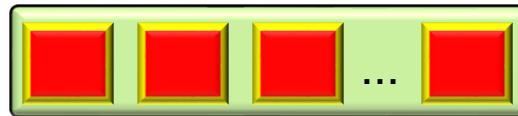
Visualizing processStream() & processInput()

- processStream() searches a list of input strings

List
<String>



Input Strings to Search



stream()



map(this::processInput)



collect(toList())

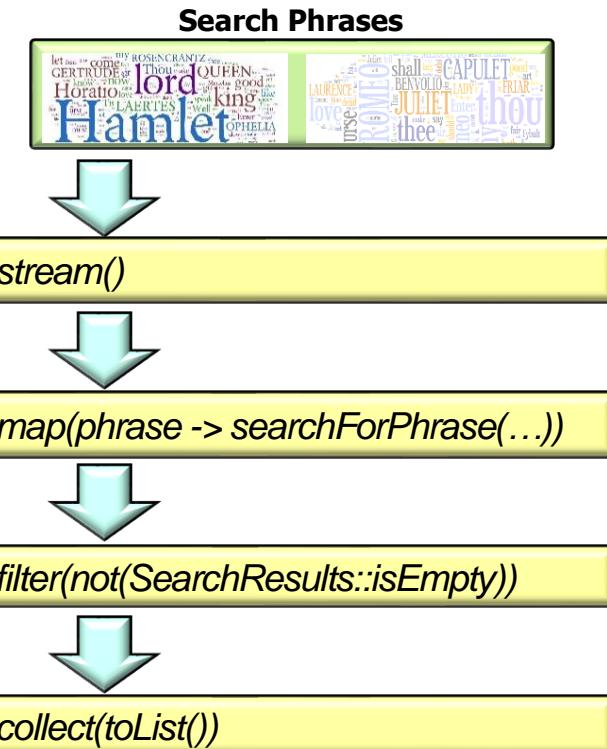
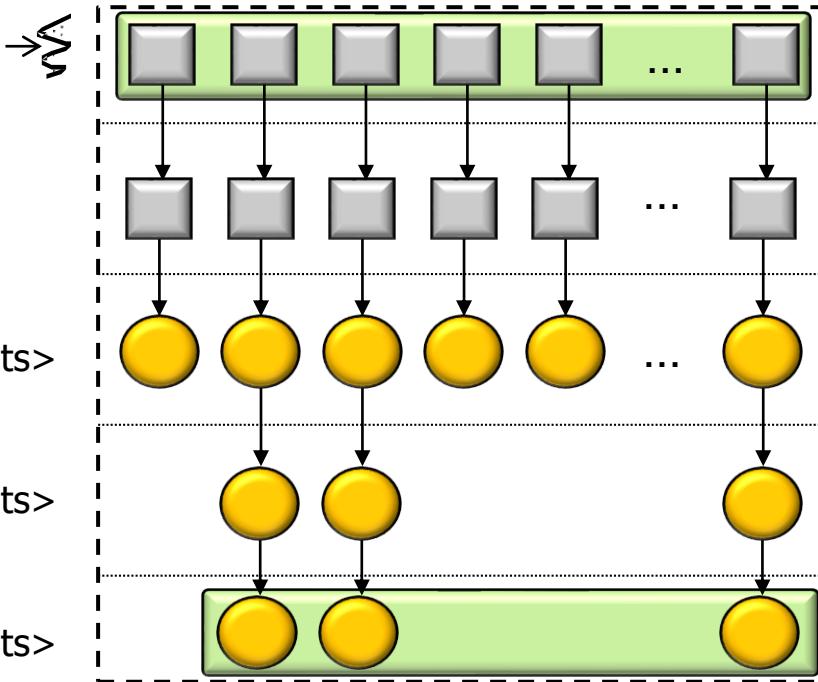
The printPhrases() method handles empty "list of search results" later

See "Java 8 Sequential SearchStreamGang Example (Part 2)"

Visualizing processStream() & processInput()

- We focus on sequential streams with one thread

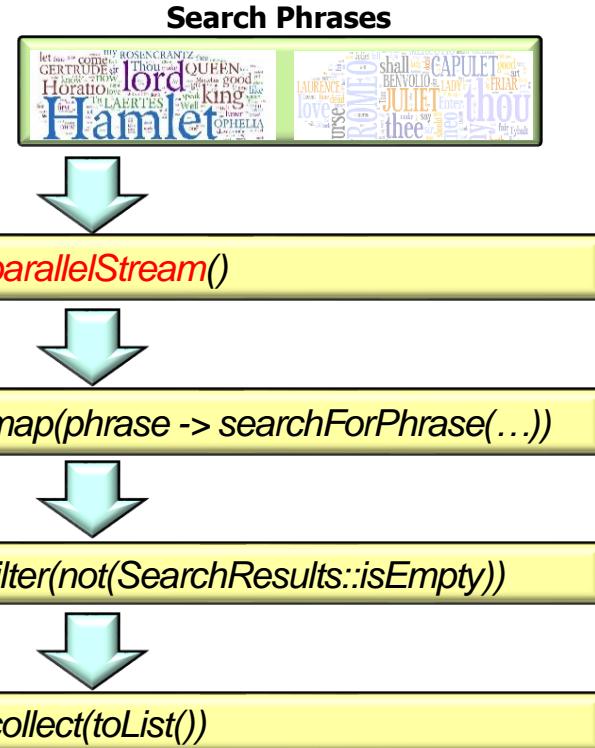
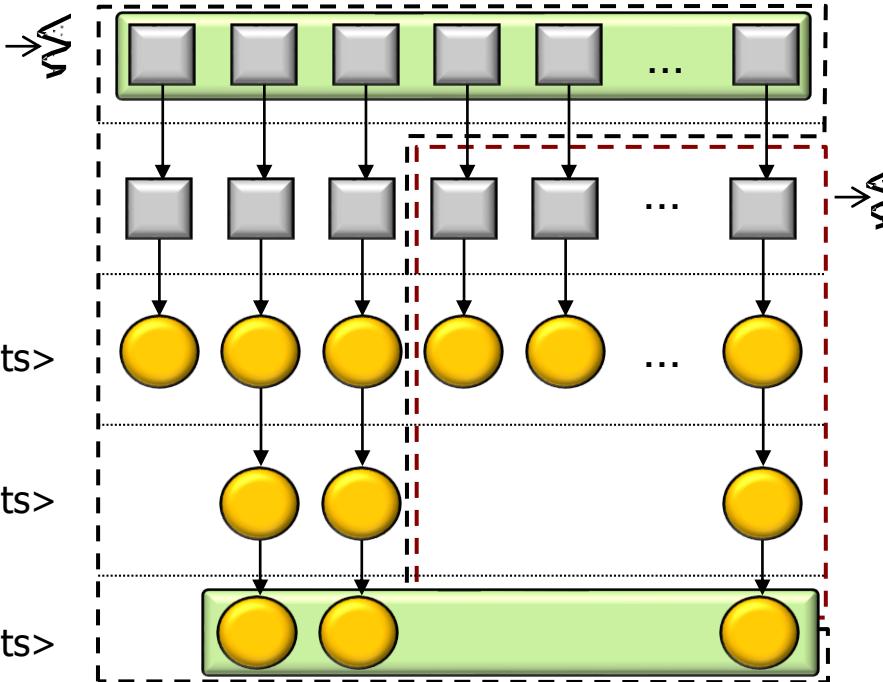
List
<String>
→
Stream
<String>
Stream
<SearchResults>
Stream
<SearchResults>
List
<SearchResults>



Visualizing processStream() & processInput()

- We focus on sequential streams with one thread
 - We'll cover parallel streams later

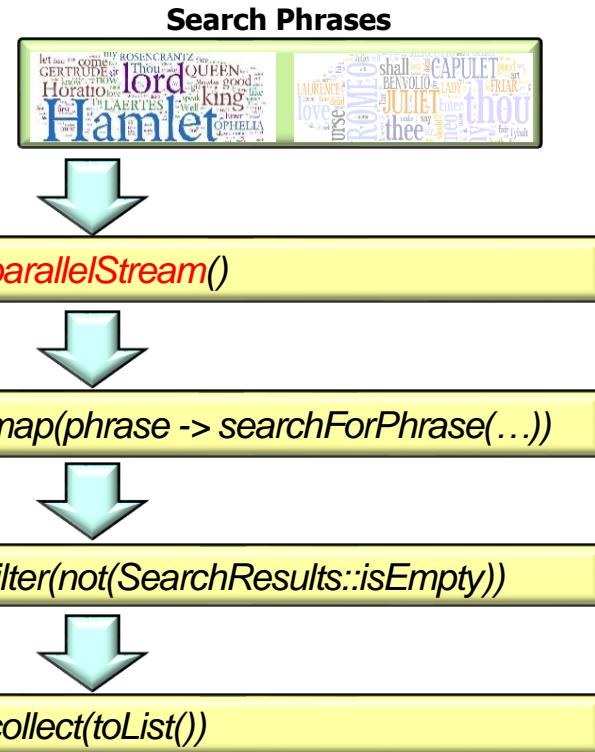
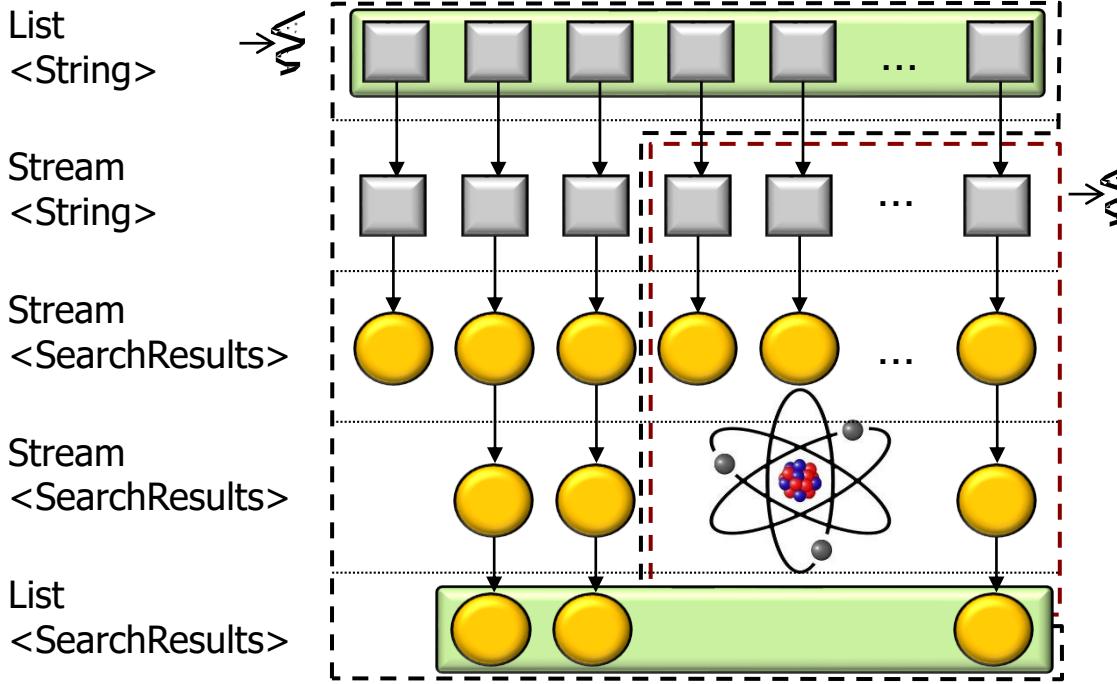
List
<String>
→
Stream
<String>
Stream
<SearchResults>
Stream
<SearchResults>
List
<SearchResults>



See "*Overview of Java 8 Parallel Streams*"

Visualizing processStream() & processInput()

- We focus on sequential streams with one thread
 - We'll cover parallel streams later



Minuscule changes are needed to transition from sequential to parallel streams!

Implementing processStream() as a Sequential Stream

Implementing processStream() as a Sequential Stream

- processStream() sequentially searches for phrases in lists of input “strings”

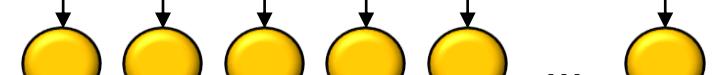
List
<CharSequence>



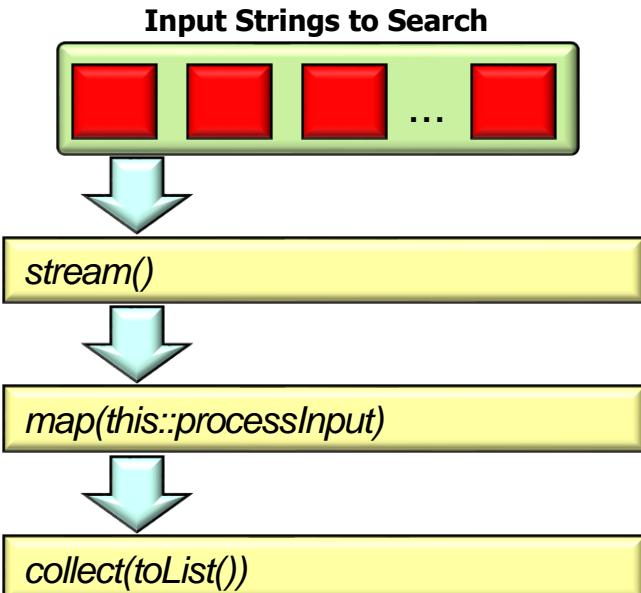
Stream
<CharSequence>



Stream<List
<SearchResults>>



List<List
<SearchResults>>



Implementing processStream() as a Sequential Stream

- processStream() sequentially searches for phrases in lists of input “strings”

```
protected List<List<SearchResults>> processStream() {  
    List<CharSequence> inputList = getInput();  
  
    return inputList  
        .stream()  
  
        .map(this::processInput)  
  
        .collect(toList());  
}
```

Implementing processStream() as a Sequential Stream

- processStream() sequentially searches for phrases in lists of input “strings”

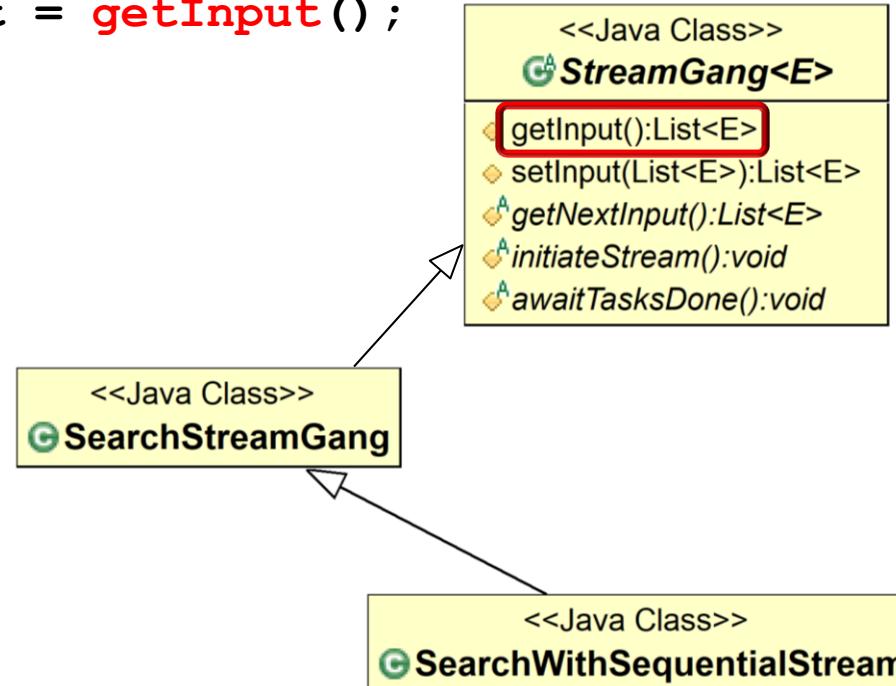
```
protected List<List<SearchResults>> processStream() {  
    List<CharSequence> inputList = getInput();  
  
    return inputList  
        .stream()  
        .map(this::processInput)  
  
        .collect(toList());  
}
```

*Get list of strings containing
all works of Shakespeare*

Implementing processStream() as a Sequential Stream

- processStream() sequentially searches for phrases in lists of input “strings”

```
protected List<List<SearchResults>> processStream() {  
    List<CharSequence> inputList = getInput();  
  
    return inputList  
        .stream()  
  
        .map(this::processInput)  
  
        .collect(toList());  
}
```



The `getInput()` method is defined in the `StreamGang` framework

Implementing processStream() as a Sequential Stream

- processStream() sequentially searches for phrases in lists of input “strings”

```
protected List<List<SearchResults>> processStream() {  
    List<CharSequence> inputList = getInput();  
  
    return inputList  
        .stream()  
        .map(this::processInput)  
  
        .collect(toList());  
}
```

CharSequence optimizes subSequence() to avoid memory copies (cf. String substring())

Implementing processStream() as a Sequential Stream

- processStream() sequentially searches for phrases in lists of input “strings”

```
protected List<List<SearchResults>> processStream() {  
    List<CharSequence> inputList = getInput();  
  
    return inputList  
        .stream()  
  
        .map(this::processInput)  
  
        .collect(toList());  
}
```

Returns a list of lists of search results denoting how many times a search phrase appeared in each input string

Implementing processStream() as a Sequential Stream

- processStream() sequentially searches for phrases in lists of input “strings”

```
protected List<List<SearchResults>> processStream() {  
    List<CharSequence> inputList = getInput();  
  
    return inputList  
        .stream()  
  
        .map(this::processInput)  
  
        .collect(toList());  
}
```

We'll later show how flatMap() "flattens" List<List<SearchResults>> into a stream of SearchResults



See “Java 8 Sequential SearchStreamGang Example (Part 2)”

Implementing processStream() as a Sequential Stream

- processStream() sequentially searches for phrases in lists of input “strings”

```
protected List<List<SearchResults>> processStream() {  
    List<CharSequence> inputList = getInput();  
  
    return inputList  
        .stream()  
        .map(this::processInput)  
  
        .collect(toList());  
}
```

*Stores # of times a phrase
appeared in an input string*

<<Java Class>>
G SearchResults

mThreadId: long
mWord: String
mTitle: String
mCycle: long
SearchResults()
SearchResults(long,long,String,String)
getTitle():String
headerToString():String
add(int):void
isEmpty():boolean
size():int
toString():String
print():SearchResults

<<Java Class>>
G Result

mIndex: int
Result(int)

Implementing processStream() as a Sequential Stream

- processStream() sequentially searches for phrases in lists of input “strings”

```
protected List<List<SearchResults>> processStream() {  
    List<CharSequence> inputList = getInput();  
  
    return inputList  
        .stream()  
  
        .map(this::processInput)  
  
        .collect(toList());  
}
```

processStream() is implemented via a sequential stream pipeline

Implementing processStream() as a Sequential Stream

- processStream() sequentially searches for phrases in lists of input “strings”

```
protected List<List<SearchResults>> processStream() {  
    List<CharSequence> inputList = getInput();  
  
    return inputList  
        .stream()  
        .map(this::processInput)  
  
        .collect(toList());  
}
```

*This factory method converts
the input list into a stream*

The stream() factory method uses StreamSupport.stream(splitter(), false)

Implementing processStream() as a Sequential Stream

- processStream() sequentially searches for phrases in lists of input “strings”

```
protected List<List<SearchResults>> processStream() {  
    List<CharSequence> inputList = getInput();  
  
    return inputList  
        .stream()  
  
        .map(this::processInput)  
  
        .collect(toList());  
}
```



Applying processInput() method reference to each input in the stream

Implementing processStream() as a Sequential Stream

- processStream() sequentially searches for phrases in lists of input “strings”

```
protected List<List<SearchResults>> processStream() {  
    List<CharSequence> inputList = getInput();  
  
    return inputList  
        .stream()  
  
        .map(this::processInput)  
  
        .collect(toList());  
}
```

processInput() returns a list of SearchResults—one list for each input string

Implementing processStream() as a Sequential Stream

- processStream() sequentially searches for phrases in lists of input “strings”

```
protected List<List<SearchResults>> processStream() {  
    List<CharSequence> inputList = getInput();  
  
    return inputList  
        .stream()  
  
        .map(this::processInput)  
  
        .collect(toList());  
}
```

This terminal operation triggers intermediate operation processing

Implementing processStream() as a Sequential Stream

- processStream() sequentially searches for phrases in lists of input “strings”

```
protected List<List<SearchResults>> processStream() {  
    List<CharSequence> inputList = getInput();  
  
    return inputList  
        .stream()  
  
        .map(this::processInput)  
  
        .collect(toList());  
}
```

Yields a list (of lists) of search results

Implementing processStream() as a Sequential Stream

- processStream() sequentially searches for phrases in lists of input “strings”

```
protected List<List<SearchResults>> processStream() {  
    List<CharSequence> inputList = getInput();  
  
    return inputList  
        .stream()  
        .map(this::processInput)  
  
        .collect(toList());  
}
```

Returns a list of lists of search results denoting how many times a search phrase appeared in each input string

Implementing processInput() as a Sequential Stream

Implementing processInput() as a Sequential Stream

- processInput() searches an input string for all occurrences of phrases to find

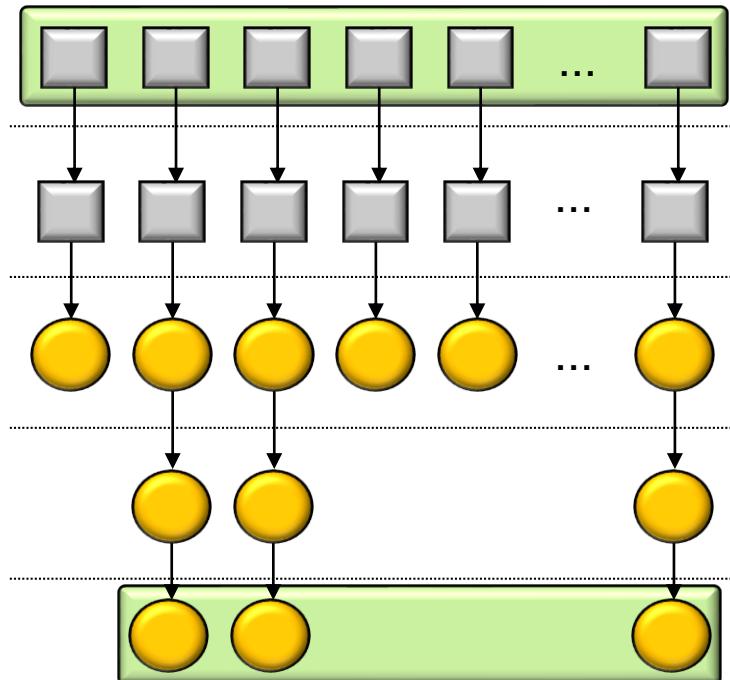
List
<String>

Stream
<String>

Stream
<SearchResults>

Stream
<SearchResults>

List
<SearchResults>



Implementing processInput() as a Sequential Stream

- processInput() searches an input string for all occurrences of phrases to find

```
List<SearchResults> processInput(CharSequence inputSeq) {  
    String title = getTitle(inputSeq);  
    CharSequence input = inputSeq.subSequence(...);  
  
    List<SearchResults> results = mPhrasesToFind  
        .stream()  
        .map(phrase  
            -> searchForPhrase(phrase, input, title, false))  
        .filter(not(SearchResults::isEmpty))  
  
        .collect(toList());  
    return results;  
}
```

Implementing processInput() as a Sequential Stream

- processInput() searches an input string for all occurrences of phrases to find

```
List<SearchResults> processInput(CharSequence inputSeq) {  
    String title = getTitle(inputSeq);  
    CharSequence input = inputSeq.subSequence(...);
```

```
List<SearchResults> results = mPhrasesToFind  
    .stream()  
    .map(phrase  
        -> searchForPhrase(phrase, input,  
    .filter(not(SearchResults::isEmpty))  
    .collect(toList());  
return results;  
}
```

The input is a section of a text file managed by the test driver program

Implementing processInput() as a Sequential Stream

- processInput() searches an input string for all occurrences of phrases to find

```
List<SearchResults> processInput(CharSequence inputSeq) {  
    String title = getTitle(inputSeq);  
    CharSequence input = inputSeq.subSequence(...);
```

The input string is split into two parts

```
List<SearchResults> results = mPhrasesToFind  
    .stream()  
    .map(phrase  
        -> searchForPhrase(phrase, input, title, false))  
    .filter(not(SearchResults::isEmpty))  
  
    .collect(toList());  
return results;  
}
```

Implementing processInput() as a Sequential Stream

- processInput() searches an input string for all occurrences of phrases to find

```
List<SearchResults> processInput(CharSequence inputSeq) {  
    String title = getTitle(inputSeq);  
    CharSequence input = inputSeq.subSequence(...);
```

subSequence() is used to avoid memory copying overhead for substrings

```
List<SearchResults> results = mPhrasesToFind  
    .stream()  
    .map(phrase  
        -> searchForPhrase(phrase, input, title, false))  
    .filter(not(SearchResults::isEmpty))  
  
    .collect(toList());  
return results;  
}
```

See [SearchStreamGang/src/main/java/livelessons/utils/SharedString.java](https://github.com/SearchStreamGang/src/main/java/livelessons/utils/SharedString.java)

Implementing processInput() as a Sequential Stream

- processInput() searches an input string for all occurrences of phrases to find

```
List<SearchResults> processInput(CharSequence inputSeq) {  
    String title = getTitle(inputSeq);  
    CharSequence input = inputSeq.subSequence(...);
```

```
List<SearchResults> results = mPhrasesToFind  
    .stream() → Convert a list of phrases into a stream  
    .map(phrase  
        -> searchForPhrase(phrase, input, title, false))  
    .filter(not(SearchResults::isEmpty))  
  
    .collect(toList());  
return results;  
}
```

Implementing processInput() as a Sequential Stream

- processInput() searches an input string for all occurrences of phrases to find

```
List<SearchResults> processInput(CharSequence inputSeq) {  
    String title = getTitle(inputSeq);  
    CharSequence input = inputSeq.subSequence(...);
```

```
List<SearchResults> results = mPhrasesToFind  
    .stream()  
    .map(phrase  
        -> searchForPhrase(phrase, input, title, false))  
    .filter(not(SearchResults::isEmpty))  
    .collect(toList());  
return results;  
}
```

Apply this function lambda to all phrases in input stream & return an output stream of SearchResults

Part 2 of this lesson shows the implementation of this method

Implementing processInput() as a Sequential Stream

- processInput() searches an input string for all occurrences of phrases to find

```
List<SearchResults> processInput(CharSequence inputSeq) {  
    String title = getTitle(inputSeq);  
    CharSequence input = inputSeq.subSequence(...);
```

```
List<SearchResults> results = mPhrasesToFind  
    .stream()  
    .map(phrase  
        -> searchForPhrase(phrase, input, title, false))  
    .filter(not(SearchResults::isEmpty))  
    .collect(toList());  
return results;  
}
```

Returns output stream containing non-empty SearchResults from input stream

Implementing processInput() as a Sequential Stream

- processInput() searches an input string for all occurrences of phrases to find

```
List<SearchResults> processInput(CharSequence inputSeq) {  
    String title = getTitle(inputSeq);  
    CharSequence input = inputSeq.subSequence(...);
```

```
List<SearchResults> results = mPhrasesToFind  
    .stream()  
    .map(phrase  
        -> searchForPhrase(phrase, input, title, false))  
    .filter(not(SearchResults::isEmpty))  
    .collect(toList());  
return results;  
}
```

*Note use of a method reference
& a negator predicate lambda*

See [SearchStreamGang/src/main/java/livelessons/utils/StreamsUtils.java](#)

Implementing processInput() as a Sequential Stream

- processInput() searches an input string for all occurrences of phrases to find

```
List<SearchResults> processInput(CharSequence inputSeq) {  
    String title = getTitle(inputSeq);  
    CharSequence input = inputSeq.subSequence(...);
```

```
List<SearchResults> results = mPhrasesToFind  
    .stream()  
    .map(phrase  
        -> searchForPhrase(phrase, input, title, false))  
    .filter(result -> result.size() > 0)  
    .collect(toList());  
return results;  
}
```

*Another approach using
a lambda expression*

Implementing processInput() as a Sequential Stream

- processInput() searches an input string for all occurrences of phrases to find

```
List<SearchResults> processInput(CharSequence inputSeq) {  
    String title = getTitle(inputSeq);  
    CharSequence input = inputSeq.subSequence(...);
```

```
List<SearchResults> results = mPhrasesToFind  
    .stream()  
    .map(phrase  
        -> searchForPhrase(phrase, input, title, false))  
    .filter(not(SearchResults::isEmpty))  
  
    .collect(toList());  
return results;  
}
```

These are both intermediate operations

Implementing processInput() as a Sequential Stream

- processInput() searches an input string for all occurrences of phrases to find

```
List<SearchResults> processInput(CharSequence inputSeq) {  
    String title = getTitle(inputSeq);  
    CharSequence input = inputSeq.subSequence(...);
```

```
List<SearchResults> results = mPhrasesToFind  
    .stream()  
    .map(phrase  
        -> searchForPhrase(phrase, input, title, false))  
    .filter(not(SearchResults::isEmpty))  
  
    .collect(toList());  
return results;  
}
```

This terminal operation triggers intermediate operation processing & yields a list result

Implementing processInput() as a Sequential Stream

- processInput() searches an input string for all occurrences of phrases to find

```
List<SearchResults> processInput(CharSequence inputSeq) {  
    String title = getTitle(inputSeq);  
    CharSequence input = inputSeq.subSequence(...);
```

```
List<SearchResults> results = mPhrasesToFind  
    .stream()  
    .map(phrase  
        -> searchForPhrase(phrase, input, title, false))  
    .filter(not(SearchResults::isEmpty))  
  
    .collect(toList());  
return results;  
}
```

This terminal operation triggers intermediate operation processing & yields a list result

Implementing processInput() as a Sequential Stream

- processInput() searches an input string for all occurrences of phrases to find

```
List<SearchResults> processInput(CharSequence inputSeq) {  
    String title = getTitle(inputSeq);  
    CharSequence input = inputSeq.subSequence(...);
```

```
List<SearchResults> results = mPhrasesToFind  
    .stream()  
    .map(phrase  
        -> searchForPhrase(phrase, input, title, false))  
    .filter(not(SearchResults::isEmpty))  
  
    .collect(toList());  
return results;
```

}

The list result is returned back to the map() operation in processStream()

End of Java 8 Sequential SearchStreamGang Example (Part 1)

Java 8 Sequential SearchStreamGang

Example (Part 2)

Douglas C. Schmidt

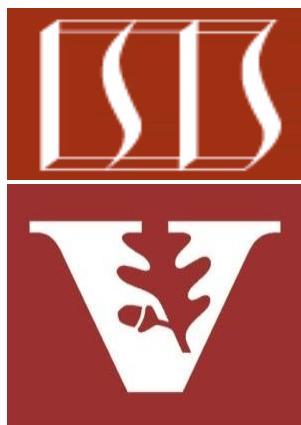
d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA



Learning Objectives in this Part of the Lesson

- Know how to apply sequential streams to the SearchStreamGang program
 - Understand the SearchStreamGang printPhrases() method

```
void printPhrases(List<List<SearchResults>>
                    listOfListOfSearchResults) {
    Map<String, List<SearchResults>> resultsMap =
        listOfListOfSearchResults
            .stream()
            .flatMap(List::stream)
            .collect(groupingBy(SearchResults::getTitle)) ;

    resultsMap.forEach((key, value) -> {
        System.out.println("Title \"" + key + "\" contained");
        value.forEach(SearchResults::print);
    });
}
```

Visualizing printPhrases()

Visualizing printPhrases()

- SearchStreamGang.printPhrases() displays phrases associated with each play

...

Title "The Tragedy of Hamlet, Prince of Denmark" contained

"It shall be so. Madness in great ones must not unwatch'd go." at [89594]

"Give every man thine ear, but few thy voice" at [26207]

"There is nothing either good or bad but thinking makes it so" at [62609]

"To be, or not to be- that is the question" at [83061]

"Neither a borrower nor a lender be" at [26556]

"This above all- to thine own self be true, And it must follow, as the night the day,
Thou canst not then be false to any man" at [26693]

"Frailty, thy name is woman" at [17233]

"The lady doth protest too much, methinks" at [102267]

"Get thee to a nunnery" at [86071|86953]

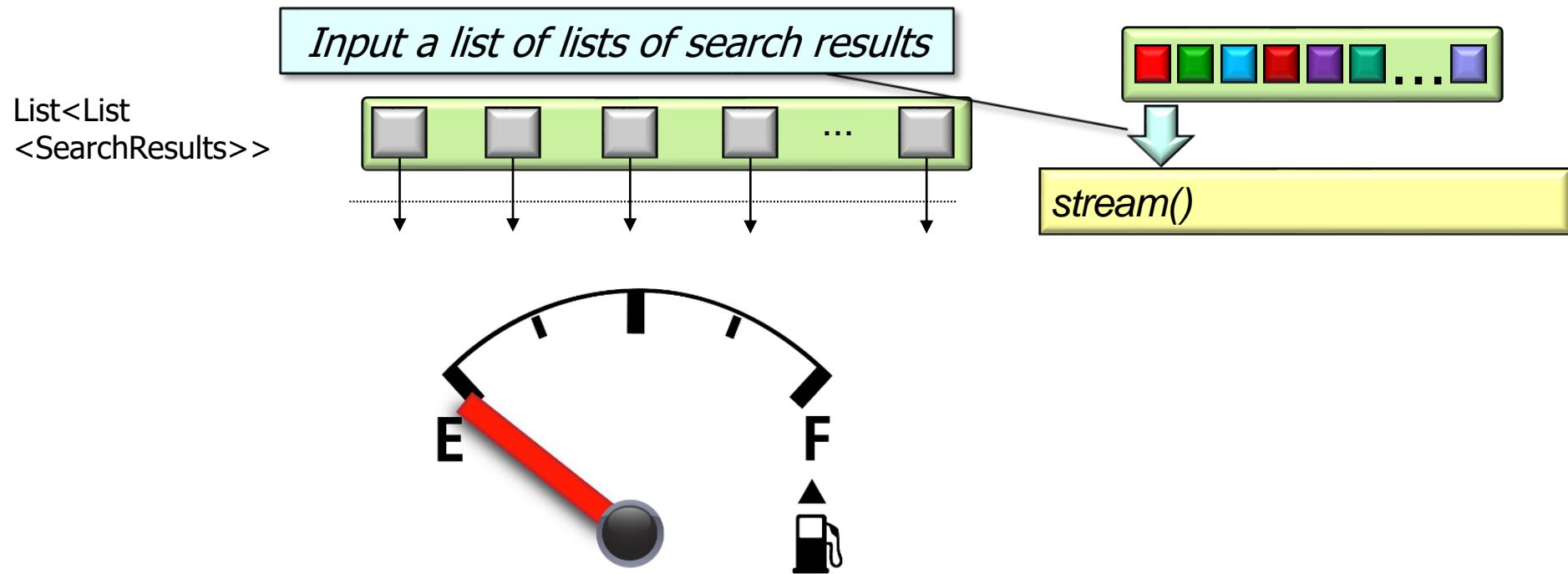
"Brevity is the soul of wit" at [54747]

...

This method shows the flatMap() & collect(groupingBy()) aggregate operations

Visualizing printPhrases()

- printPhrases() uses a stream that converts a list of lists of search results into a map that associates phrases with the works where they were found

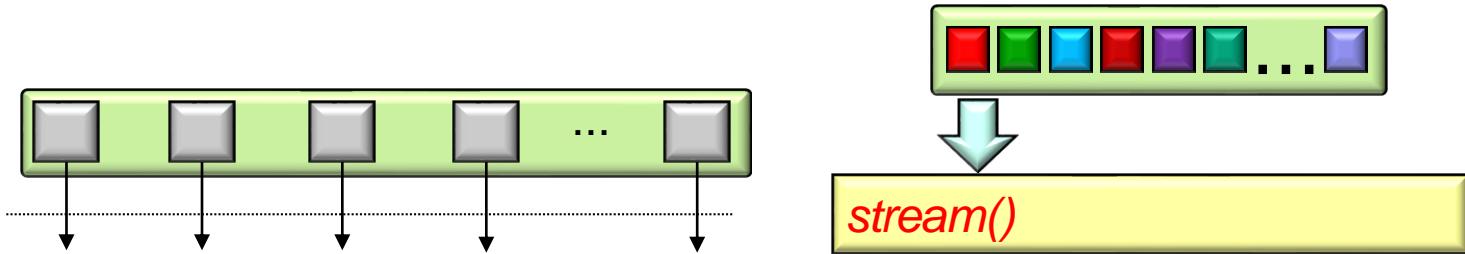


Some lists of search results may be empty if no phrases matched an input string

Visualizing printPhrases()

- printPhrases() uses a stream that converts a list of lists of search results into a map that associates phrases with the works where they were found

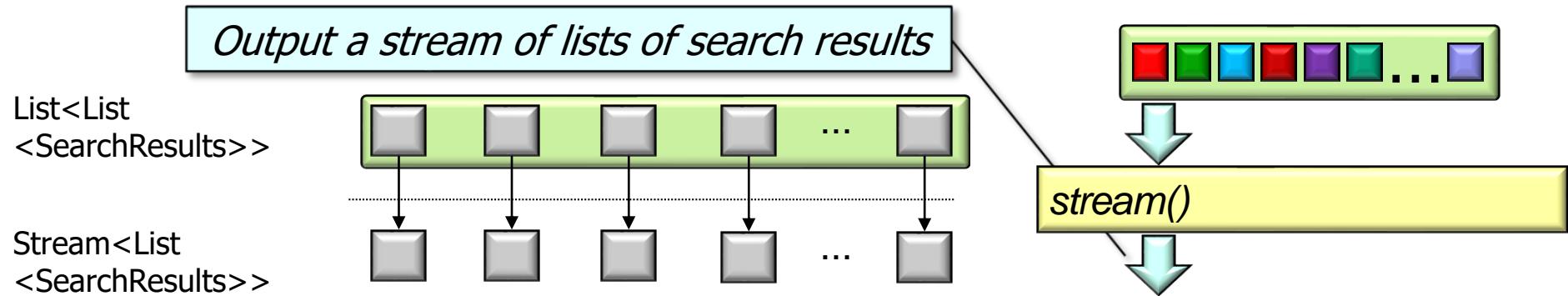
List<List
<SearchResults>>



Convert list to a (sequential) stream of lists of search results

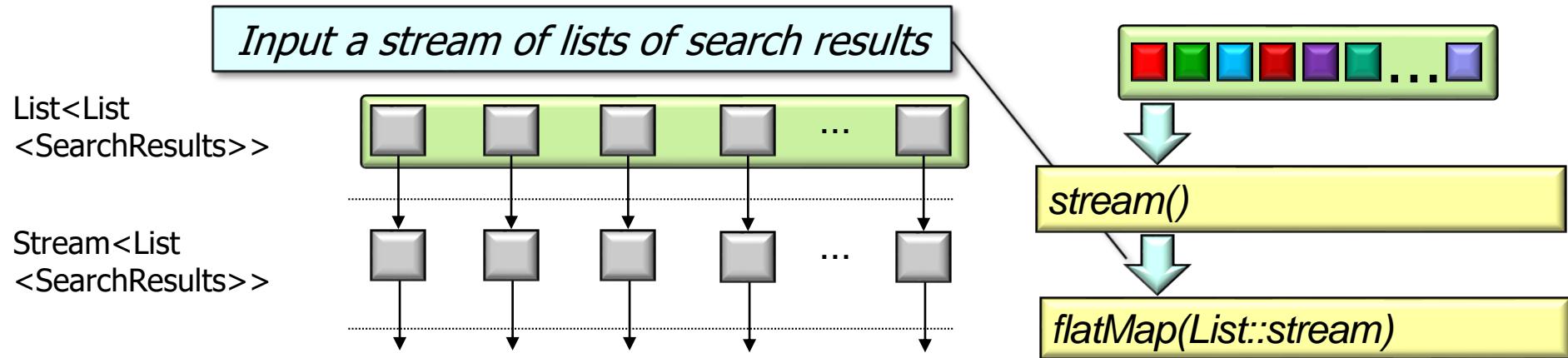
Visualizing printPhrases()

- printPhrases() uses a stream that converts a list of lists of search results into a map that associates phrases with the works where they were found



Visualizing printPhrases()

- printPhrases() uses a stream that converts a list of lists of search results into a map that associates phrases with the works where they were found

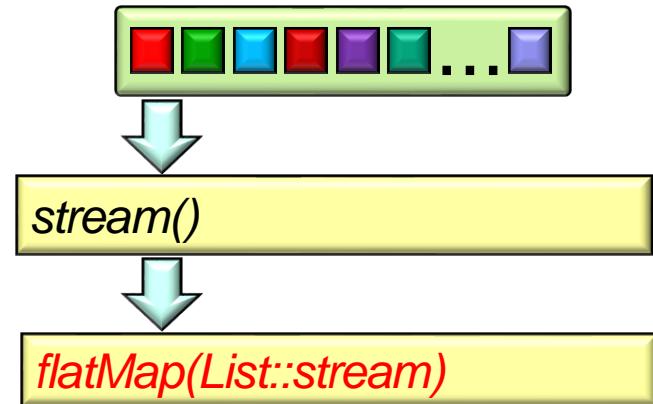
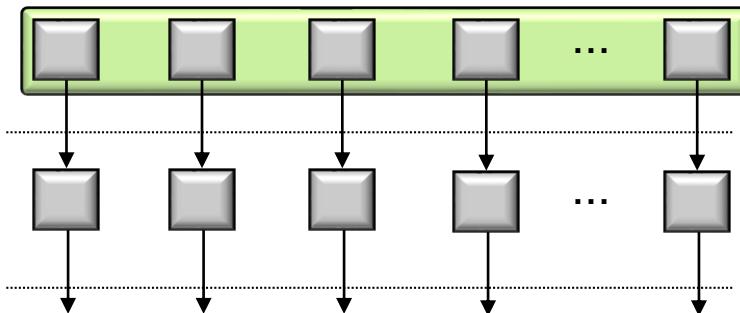


Visualizing printPhrases()

- printPhrases() uses a stream that converts a list of lists of search results into a map that associates phrases with the works where they were found

List<List
<SearchResults>>

Stream<List
<SearchResults>>

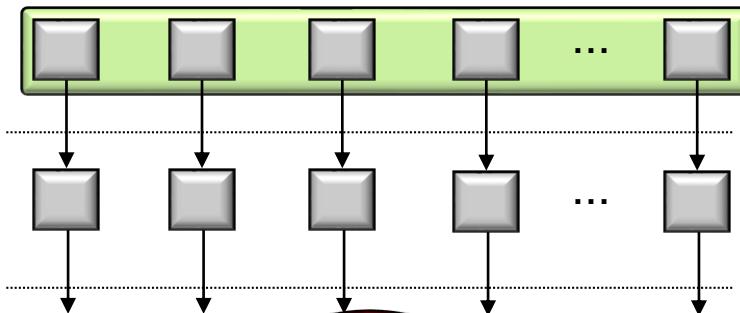


Flatten the stream of lists of search results to a stream of search results

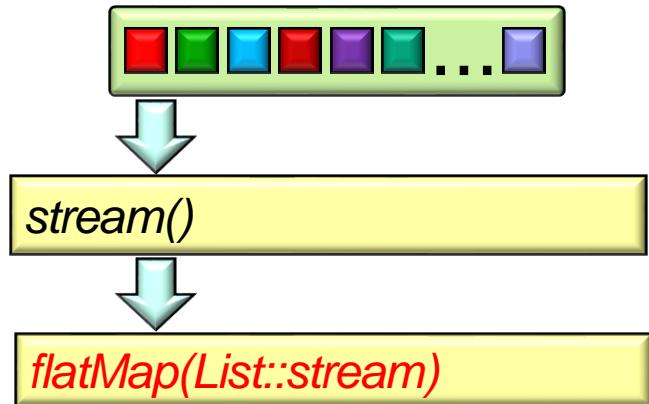
Visualizing printPhrases()

- printPhrases() uses a stream that converts a list of lists of search results into a map that associates phrases with the works where they were found

List<List
<SearchResults>>



Stream<List
<SearchResults>>



flatMap() removes empty lists

Empty lists will occur if no phrases match in an input string (Shakespeare work)

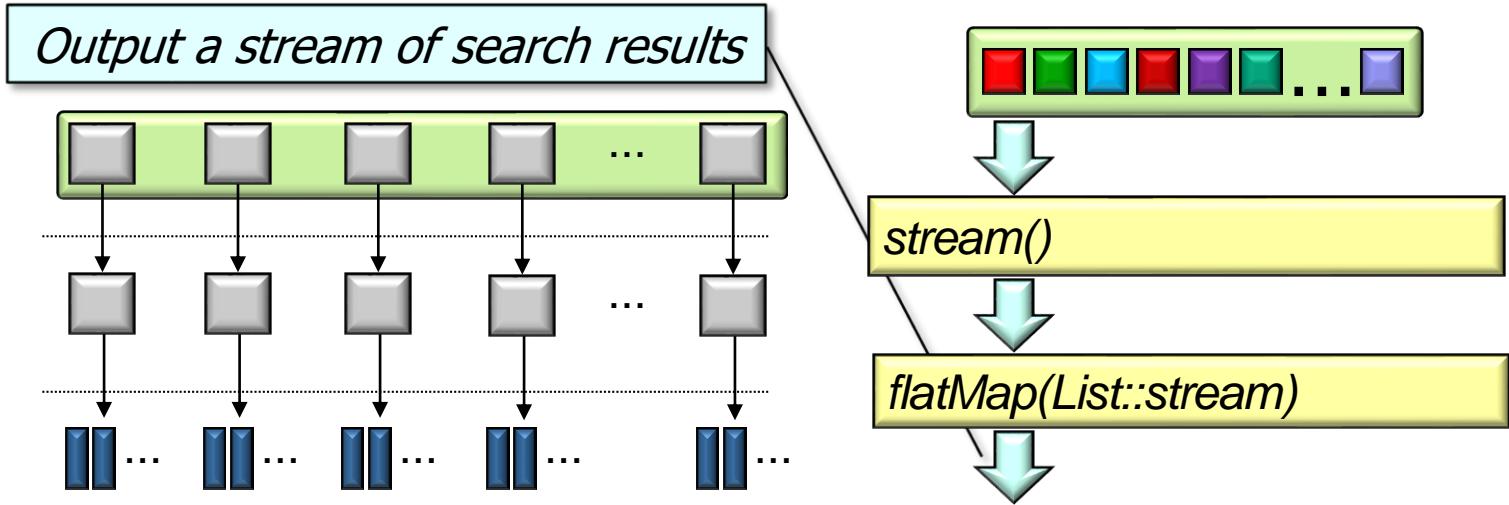
Visualizing printPhrases()

- printPhrases() uses a stream that converts a list of lists of search results into a map that associates phrases with the works where they were found

List<List
<SearchResults>>

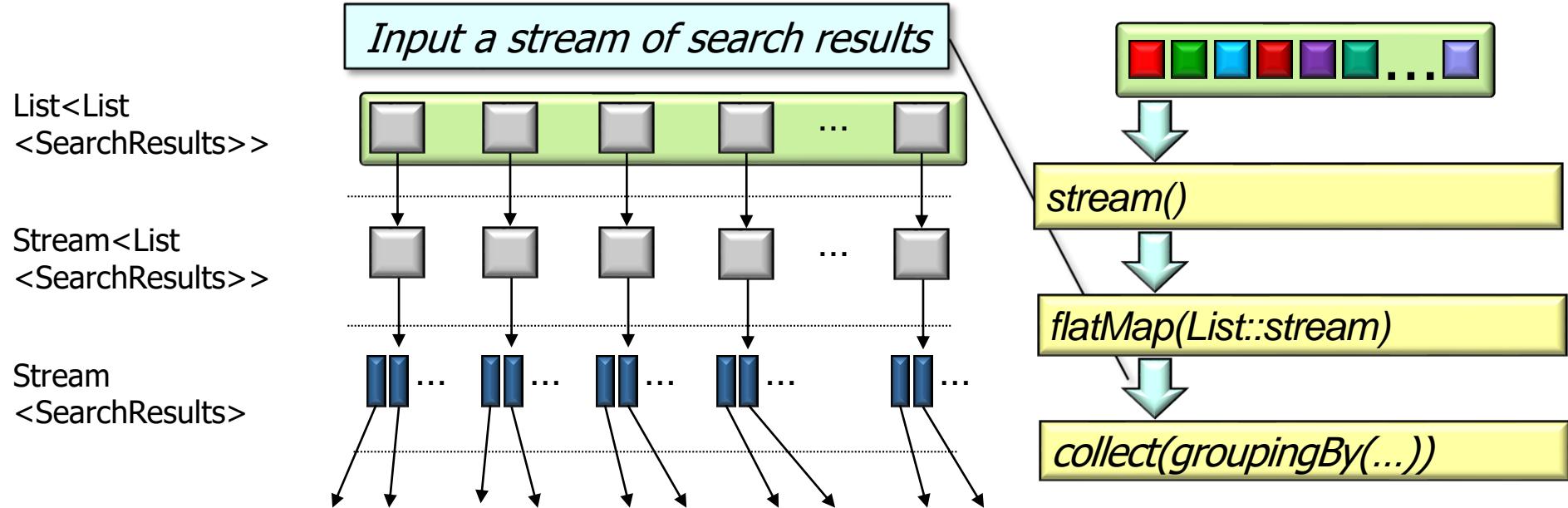
Stream<List
<SearchResults>>

Stream
<SearchResults>



Visualizing printPhrases()

- printPhrases() uses a stream that converts a list of lists of search results into a map that associates phrases with the works where they were found



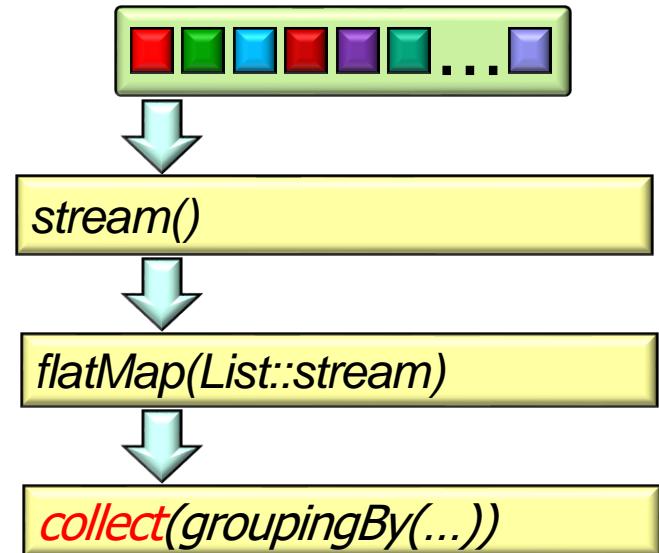
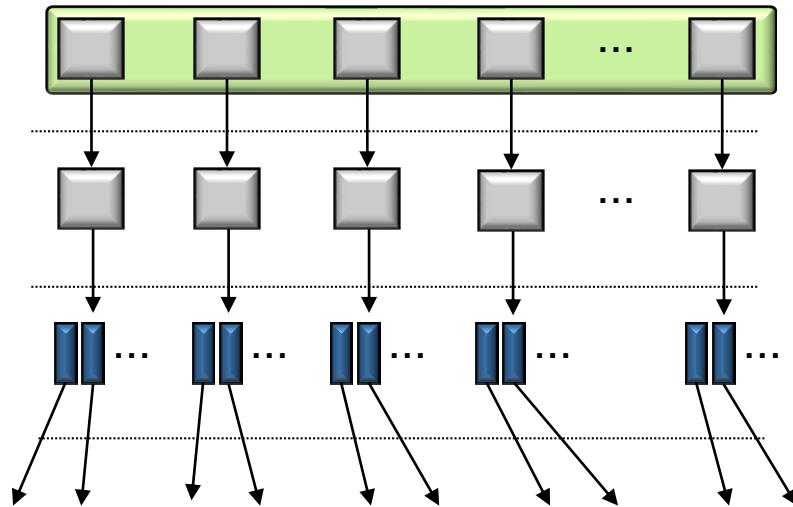
Visualizing printPhrases()

- printPhrases() uses a stream that converts a list of lists of search results into a map that associates phrases with the works where they were found

List<List
<SearchResults>>

Stream<List
<SearchResults>>

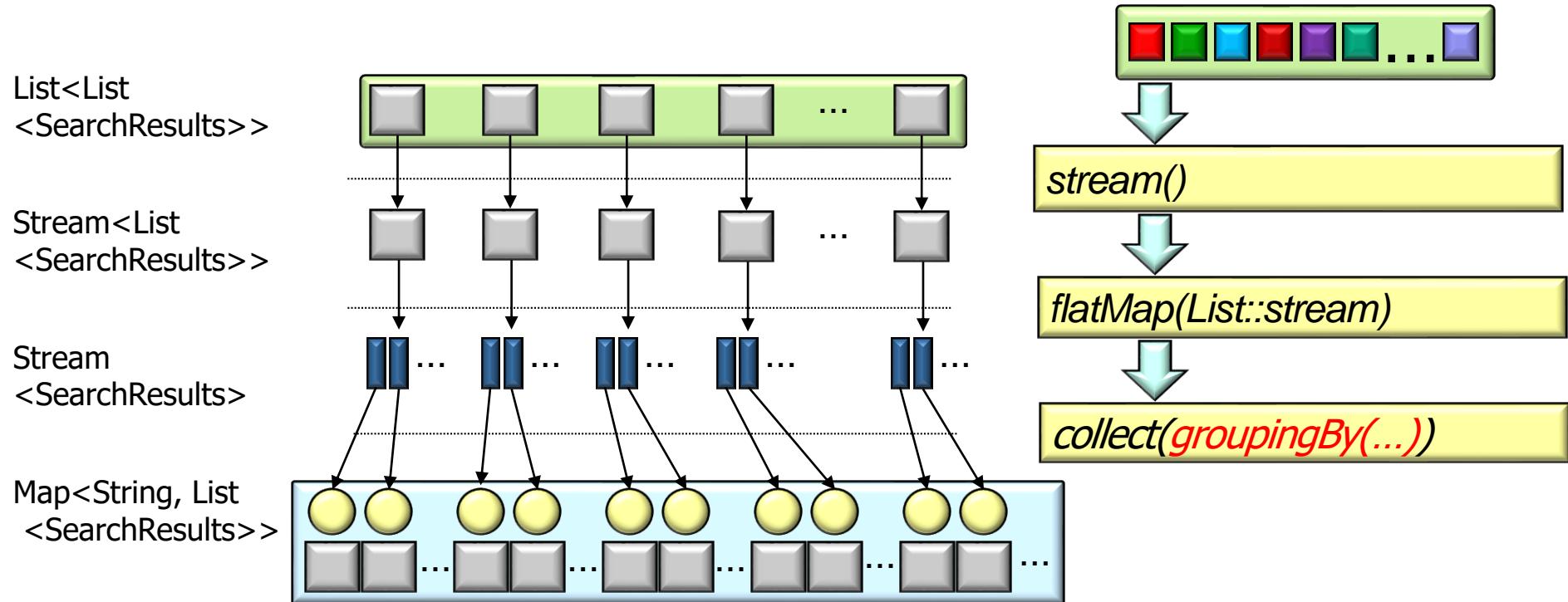
Stream
<SearchResults>



Trigger intermediate operation processing

Visualizing printPhrases()

- printPhrases() uses a stream that converts a list of lists of search results into a map that associates phrases with the works where they were found



Create a map that groups phrases according to the works where they appear

Implementing printPhrases() as a Sequential Stream

Implementing printPhrases() as a Sequential Stream

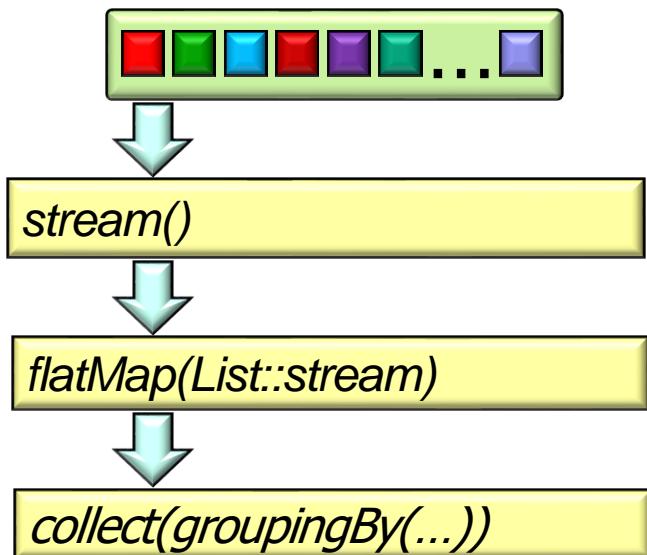
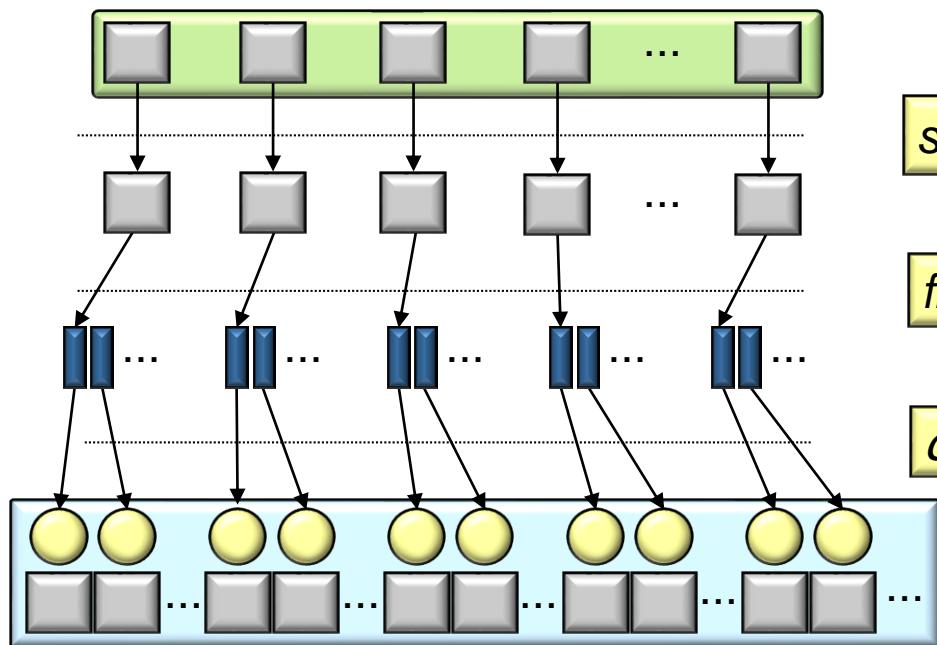
- printPhrases() uses a stream to converts a list of lists of search results to a map that associates phrases found in the input with the works where they were found

List<List
<SearchResults>>

Stream<List
<SearchResults>>

Stream
<SearchResults>

Map<String, List
<SearchResults>>



Implementing printPhrases() as a Sequential Stream

- printPhrases() uses a stream to display phrases associated with each play

```
void printPhrases(List<List<SearchResults>> listOfListOfResults) {  
    Map<String, List<SearchResults>> map = listOfListOfResults  
        .stream()  
  
        .flatMap(List::stream)  
  
        .collect(groupingBy(SearchResults::getTitle));  
  
    map.forEach((key, value) -> {  
        System.out.println("Title \""  
            + key  
            + "\" contained");  
        value.forEach(SearchResults::print);});  
}
```

See [SearchStreamGang/src/main/java/livelessons/streamgangs/SearchStreamGang.java](#)

Implementing printPhrases() as a Sequential Stream

- printPhrases() uses a stream to display phrases associated with each play

```
void printPhrases(List<List<SearchResults>> listOfListOfResults) {  
    Map<String, List<SearchResults>> map = listOfListOfResults  
        .stream()  
        .flatMap(List::stream)  
        .collect(groupingBy(SearchResults::getTitle));  
  
    map.forEach((key, value) -> {  
        System.out.println("Title \"  
            + key  
            + "\" contained");  
        value.forEach(SearchResults::print);});  
}
```

*Converts the list of lists of search results
into a stream of lists of search results*

Implementing printPhrases() as a Sequential Stream

- printPhrases() uses a stream to display phrases associated with each play

```
void printPhrases(List<List<SearchResults>> listOfListOfResults) {  
    Map<String, List<SearchResults>> map = listOfListOfResults  
        .stream()  
        .flatMap(List::stream)  
        .collect(groupingBy(SearchResults::getTitle));  
  
    map.forEach((key, value) -> {  
        System.out.println("Title \"'"  
            + key  
            + "\\" contained");  
        value.forEach(SearchResults::print);});  
}
```

Returns an output stream that flattens the stream of lists into a stream of search results (any/all empty lists are ignored)



Implementing printPhrases() as a Sequential Stream

- printPhrases() uses a stream to display phrases associated with each play

```
void printPhrases(List<List<SearchResults>> listOfListOfResults) {  
    Map<String, List<SearchResults>> map = listOfListOfResults  
        .stream()  
  
        .flatMap(List::stream)  
  
        .collect(groupingBy(SearchResults::getTitle));  
  
    map.forEach((key, value) -> {  
        System.out.println("Title \""  
            + key  
            + "\" contained");  
        value.forEach(SearchResults::print);});  
}
```



of output stream elements may differ from the # of input stream elements

Implementing printPhrases() as a Sequential Stream

- printPhrases() uses a stream to display phrases associated with each play

```
void printPhrases(List<List<SearchResults>> listOfListOfResults) {  
    Map<String, List<SearchResults>> map = listOfListOfResults  
        .stream()  
  
        .flatMap(List::stream)  
        .collect(groupingBy(SearchResults::getTitle));  
  
    map.forEach((key, value) -> {  
        System.out.println("Title \"  
            + key  
            + "\" contained");  
        value.forEach(SearchResults::print);});  
}
```



flatMap() can also transforms its input stream type into a different output stream type

Implementing printPhrases() as a Sequential Stream

- printPhrases() uses a stream to display phrases associated with each play

```
void printPhrases(List<List<SearchResults>> listOfListOfResults) {  
    Map<String, List<SearchResults>> map = listOfListOfResults  
        .stream()  
        .flatMap(List::stream)  
        .collect(Collectors.groupingBy(SearchResults::getTitle));  
  
    map.forEach((key, value) -> {  
        System.out.println("Title \"  
            + key  
            + "\" contained");  
        value.forEach(SearchResults::print);});  
}
```

Groups elements via to a classification function & return results in a Map



Implementing printPhrases() as a Sequential Stream

- printPhrases() uses a stream to display phrases associated with each play

```
void printPhrases(List<List<SearchResults>> listOfListOfResults) {  
    Map<String, List<SearchResults>> map = listOfListOfResults  
        .stream()  
        .flatMap(List::stream)  
        .collect(groupingBy(SearchResults::getTitle));  
  
    map.forEach((key, value) -> {  
        System.out.println("Title \"  
            + key  
            + "\" contained");  
        value.forEach(SearchResults::print);});  
}
```

*Associates phrases found in input
with titles where they were found*

Implementing printPhrases() as a Sequential Stream

- printPhrases() uses a stream to display phrases associated with each play

```
void printPhrases(List<List<SearchResults>> listOfListOfResults) {  
    Map<String, List<SearchResults>> map = listOfListOfResults  
        .stream()  
  
        .flatMap(List::stream)  
  
        .collect(groupingBy(SearchResults::getTitle));  
  
    map.forEach((key, value) -> {  
        System.out.println("Title \"'"  
            + key  
            + "\" contained");  
        value.forEach(SearchResults::print);});  
}
```



Displays titles (keys) & phrases (values) in map

See docs.oracle.com/javase/8/docs/api/java/util/Map.html#forEach

End of Java 8 Sequential SearchStreamGang Example (Part 2)

Java 8 Sequential SearchStreamGang

Example (Part 3)

Douglas C. Schmidt

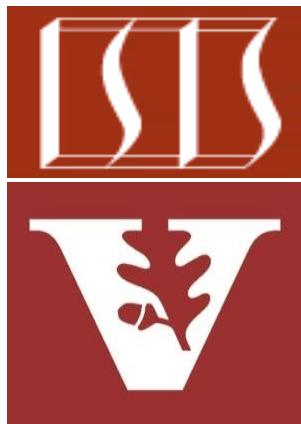
d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

Institute for Software
Integrated Systems

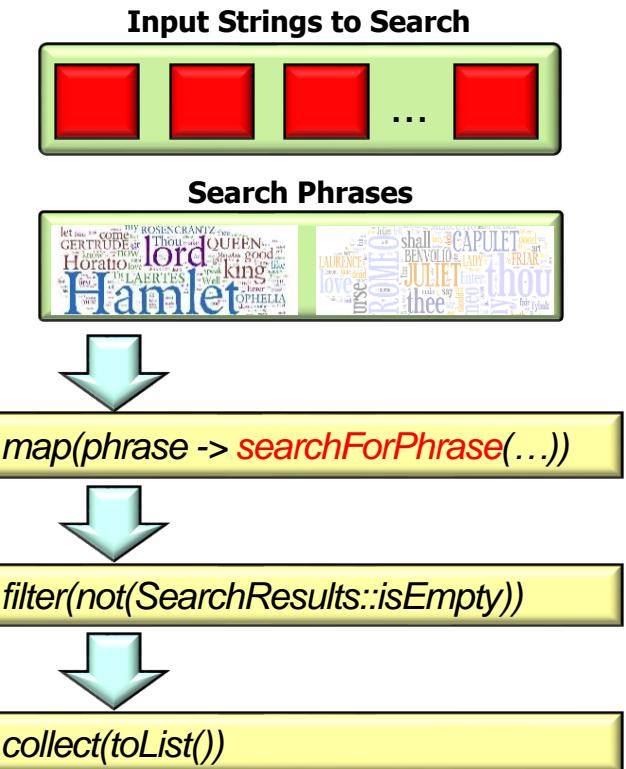
Vanderbilt University
Nashville, Tennessee, USA



Learning Objectives in this Part of the Lesson

- Know how to apply sequential streams to the SearchStreamGang program
- Recognize how a Spliterator is used in SearchWithSequentialStreams

```
SearchResults searchForPhrase  
  (String phrase, CharSequence input,  
   String title, boolean parallel) {  
  return new SearchResults  
    (... , phrase, ... , StreamSupport  
     .stream(new PhraseMatchSpliterator  
              (input, phrase),  
              parallel)  
     .collect(toList()));  
}
```



Learning Objectives in this Part of the Lesson

- Know how to apply sequential streams to the SearchStreamGang program
- Recognize how a Spliterator is used in SearchWithSequentialStreams
- Understand the pros & cons of the SearchWithSequentialStreams class

<<Java Class>>

G SearchWithSequentialStreams

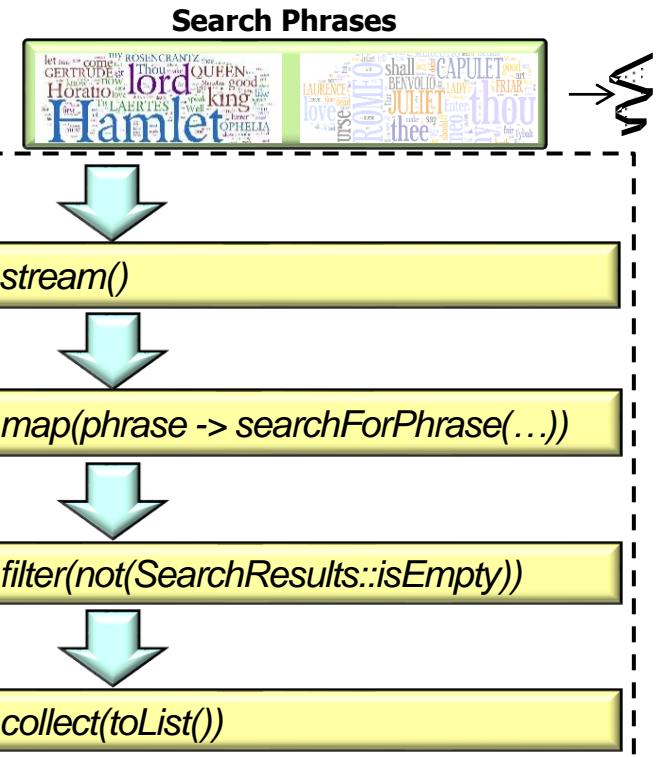
- ◆ processStream():List<List<SearchResults>>
- processInput(String):List<SearchResults>



Using Java Spliterator in SearchStreamGang

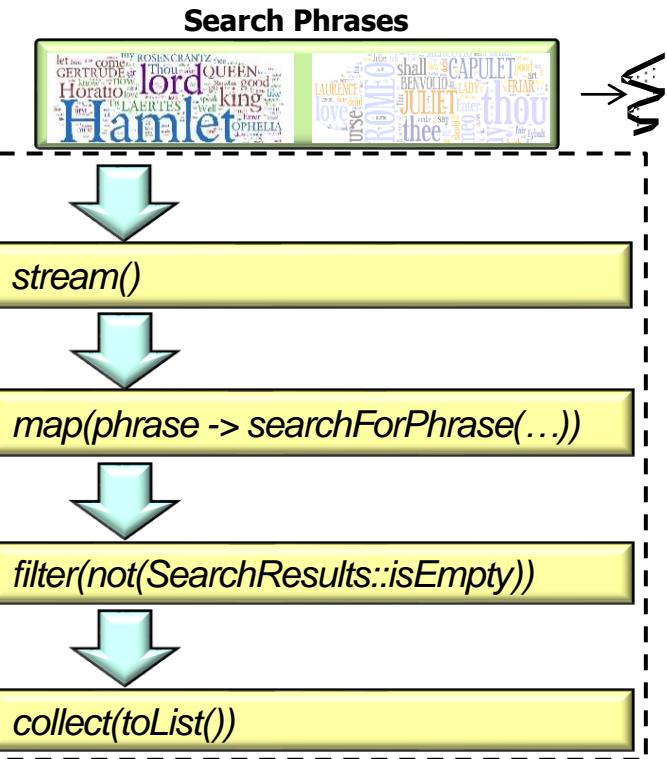
Using Java Spliterator in SearchStreamGang

- SearchStreamGang uses PhraseMatchSpliterator that works for both sequential & parallel streams



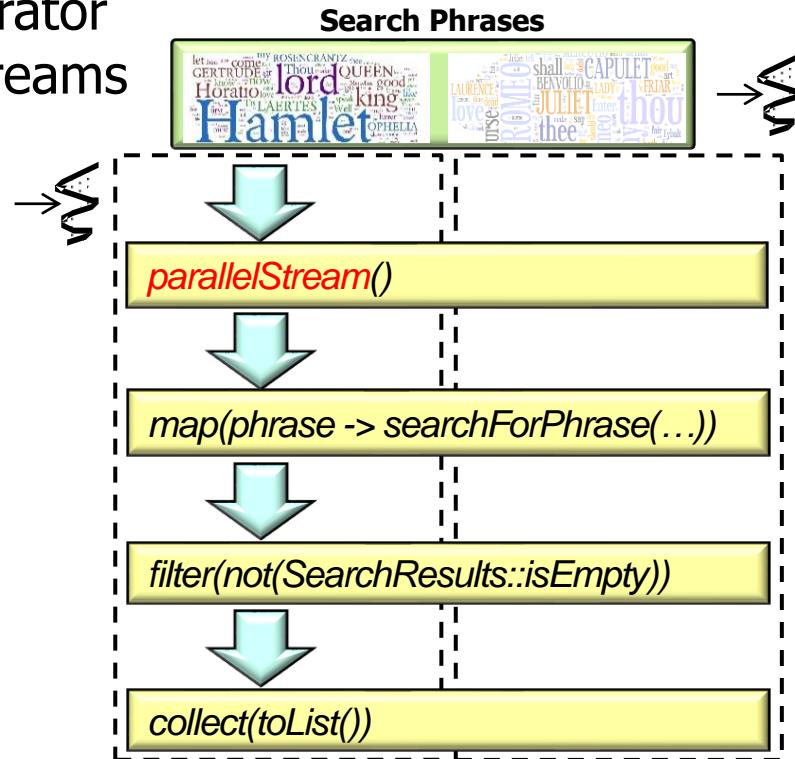
Using Java Spliterator in SearchStreamGang

- SearchStreamGang uses PhraseMatchSpliterator that works for both sequential & parallel streams
 - We focus on the sequential portions now



Using Java Spliterator in SearchStreamGang

- SearchStreamGang uses PhraseMatchSpliterator that works for both sequential & parallel streams
 - We focus on the sequential portions now
 - We'll cover the parallel portions later



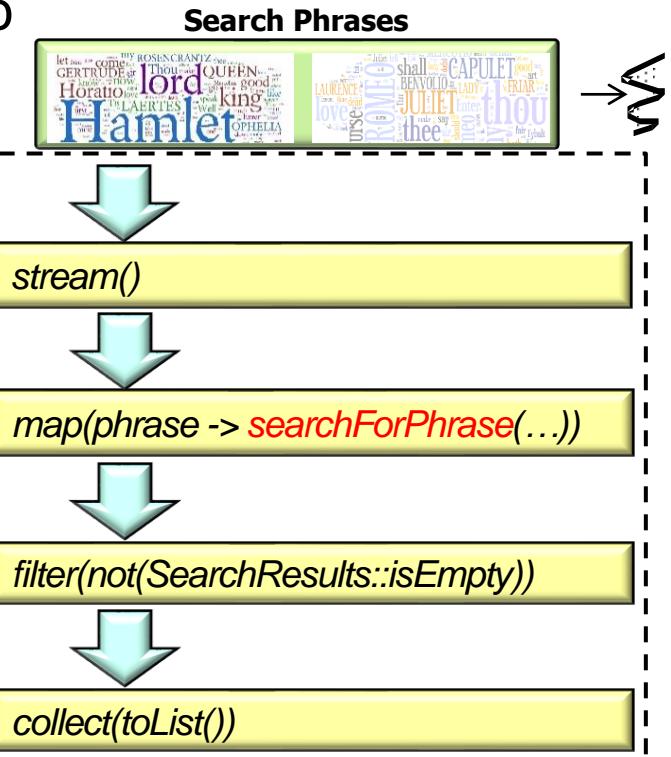
See "Java 8 Parallel SearchStreamGang Example (Part 2)"

Using Java Spliterator in SearchStreamGang

- searchForPhrase() uses PhraseMatchSpliterator to find all phrases in input & return SearchResults

SearchResults **searchForPhrase**

```
(String phrase, CharSequence input,  
 String title, boolean parallel) {  
 return new SearchResults  
 (... , phrase, ... , StreamSupport  
 .stream(new PhraseMatchSpliterator  
 (mInput, word),  
 parallel)  
 .collect(toList()));  
 }
```

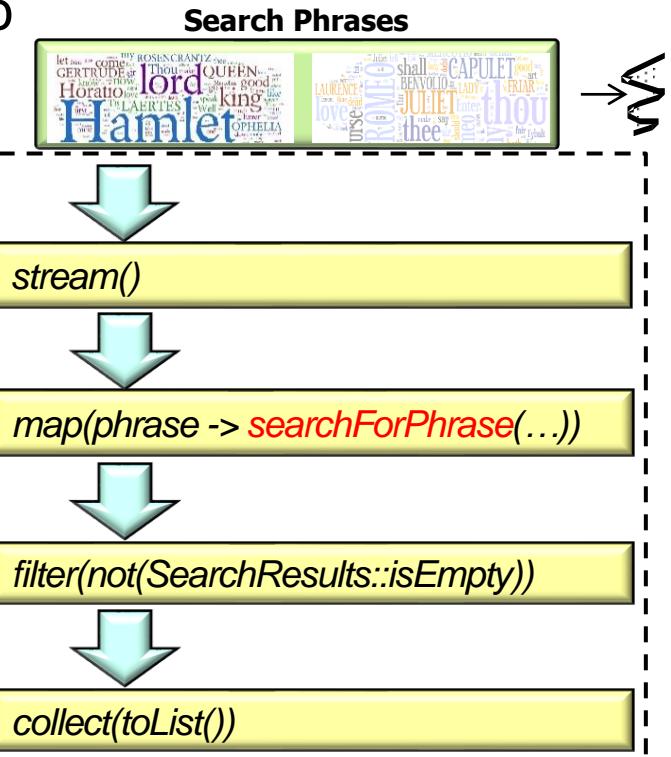


Using Java Spliterator in SearchStreamGang

- searchForPhrase() uses PhraseMatchSpliterator to find all phrases in input & return SearchResults

```
SearchResults searchForPhrase  
    (String phrase, CharSequence input,  
     String title, boolean parallel) {  
    return new SearchResults  
        (... , phrase, ... , StreamSupport  
            .stream(new PhraseMatchSpliterator  
                (input, phrase),  
                parallel)  
            .collect(toList()));  
}
```

StreamSupport.stream() creates a sequential or parallel stream via PhraseMatchSpliterator

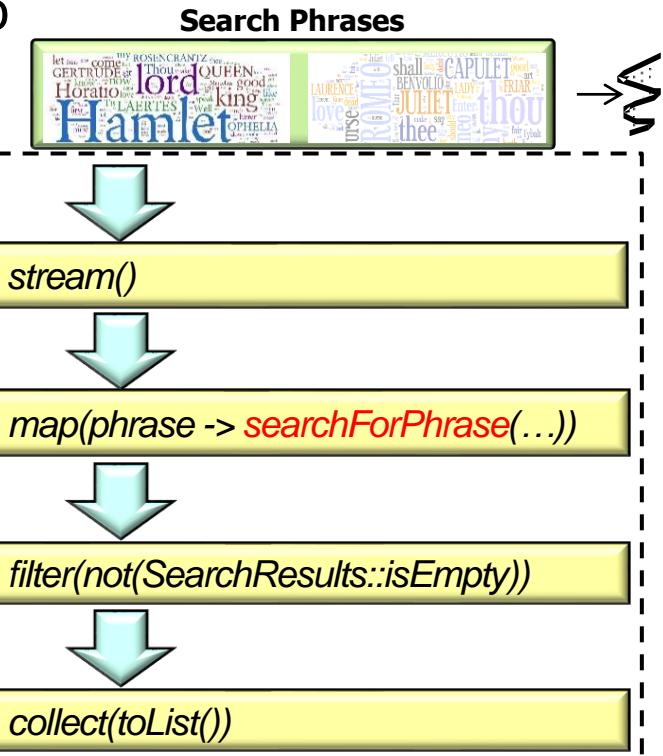


Using Java Spliterator in SearchStreamGang

- searchForPhrase() uses PhraseMatchSpliterator to find all phrases in input & return SearchResults

```
SearchResults searchForPhrase  
  (String phrase, CharSequence input,  
   String title, boolean parallel) {  
  return new SearchResults  
    (... , phrase, ... , StreamSupport  
     .stream(new PhraseMatchSpliterator  
              (input, phrase),  
              parallel)  
     .collect(toList()));  
}
```

For SearchWithSequentialStreams "parallel" is false, so we'll use a sequential splitter



Using Java Spliterator in SearchStreamGang

- Here's the input/output of PhraseMatchSpliterator for SearchWithSequentialStreams

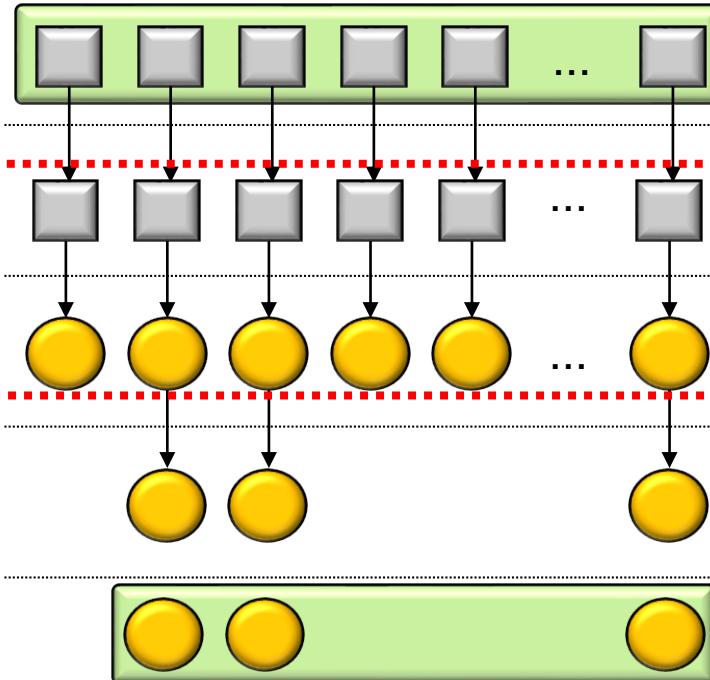
List
<String>

Stream
<String>

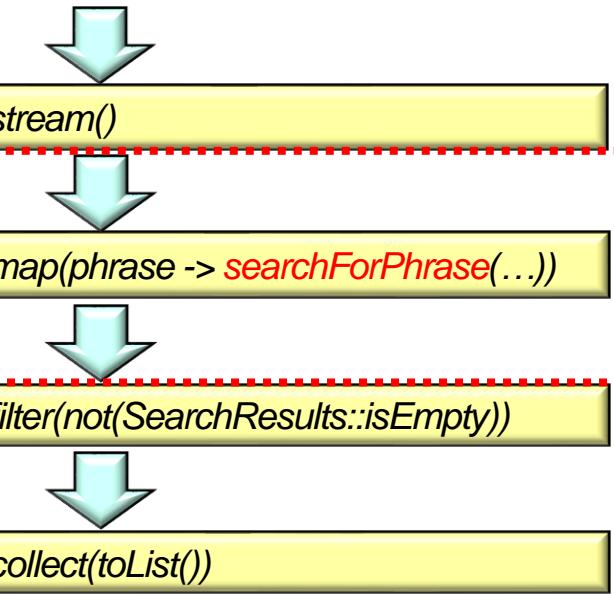
Stream
<SearchResults>

Stream
<SearchResults>

List
<SearchResults>



Search Phrases



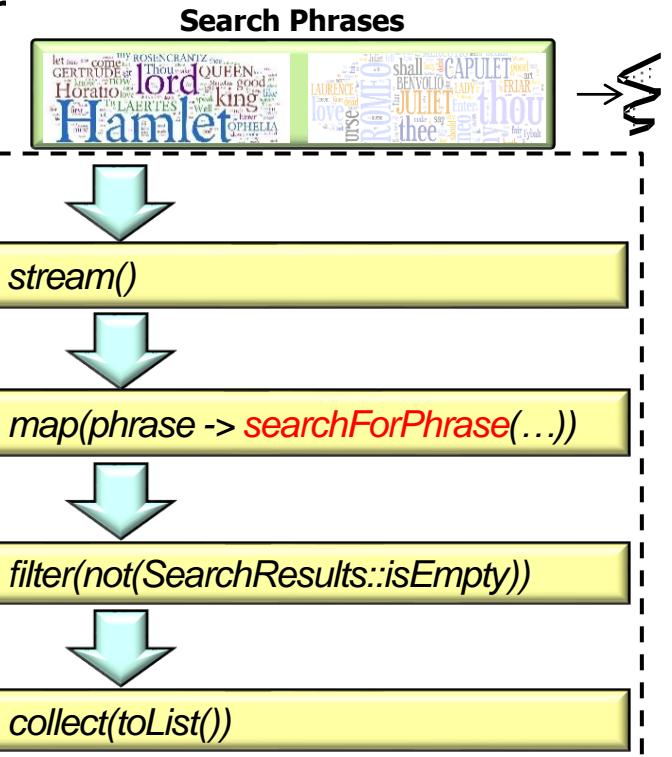
Using Java Spliterator in SearchStreamGang

- Here's the input/output of PhraseMatchSpliterator for SearchWithSequentialStreams

"...

My liege, and madam, to expostulate
What majesty should be, what duty is,
Why day is day, night is night, and time is time.
Were nothing but to waste night, day, and time.
Therefore, since **brevity is the soul of wit**,
And tediousness the limbs and outward flourishes,
I will be brief. ..."

*"Brevity is the soul of wit"
matches at index [54739]*



Using Java Spliterator in SearchStreamGang

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a phrase appears in an input string

```
class PhraseMatchSpliterator implements Spliterator<Result> {  
    private CharSequence mInput;  
  
    private final String mPhrase;  
  
    private final Pattern mPattern;  
  
    private Matcher mPhraseMatcher;  
  
    private final int mMinSplitSize;  
  
    private int mOffset = 0;  
  
    ...
```

See [SearchStreamGang/src/main/java/livelessons/utils/PhraseMatchSpliterator.java](#)

Using Java Spliterator in SearchStreamGang

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a phrase appears in an input string

```
class PhraseMatchSpliterator implements Spliterator<Result> {  
    private CharSequence mInput;  
  
    private final String mPhrase;  
  
    private final Pattern mPattern;  
  
    private Matcher mPhraseMatcher;  
  
    private final int mMinSplitSize;  
  
    private int mOffset = 0;  
  
    ...
```

Spliterator is an interface that defines eight methods, including tryAdvance() & trySplit()

Using Java Spliterator in SearchStreamGang

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a phrase appears in an input string

```
class PhraseMatchSpliterator implements Spliterator<Result> {  
    private CharSequence mInput;  
  
    private final String mPhrase;  
  
    private final Pattern mPattern;  
  
    private Matcher mPhraseMatcher;  
  
    private final int mMinSplitSize;  
  
    private int mOffset = 0;  
  
    ...
```

These fields implement Phrase MatchSpliterator for both of the sequential & parallel use-cases

Some fields are updated in the trySplit() method, which is why they aren't final

Using Java Spliterator in SearchStreamGang

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a phrase appears in an input string

```
class PhraseMatchSpliterator implements Spliterator<Result> {  
    private CharSequence mInput;  
  
    private final String mPhrase;  
  
    private final Pattern mPattern;  
  
    private Matcher mPhraseMatcher;  
  
    private final int mMinSplitSize;  
  
    private int mOffset = 0;  
  
    ...
```

*Contains a single
work of Shakespeare*

Using Java Spliterator in SearchStreamGang

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a phrase appears in an input string

```
class PhraseMatchSpliterator implements Spliterator<Result> {  
    private CharSequence mInput;  
  
    private final String mPhrase;  
  
    private final Pattern mPattern;  
  
    private Matcher mPhraseMatcher;  
  
    private final int mMinSplitSize;  
  
    private int mOffset = 0;  
  
    ...
```

Contains the phrase to search for in the work

Using Java Spliterator in SearchStreamGang

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a phrase appears in an input string

```
class PhraseMatchSpliterator implements Spliterator<Result> {  
    private CharSequence mInput;  
  
    private final String mPhrase;  
  
    private final Pattern mPattern;  
  
    private Matcher mPhraseMatcher;  
  
    private final int mMinSplitSize;  
  
    private int mOffset = 0;  
  
    ...
```

Contains the regular expression representation of the phrase

See docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html

Using Java Spliterator in SearchStreamGang

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a phrase appears in an input string

```
class PhraseMatchSpliterator implements Spliterator<Result> {  
    private CharSequence mInput;  
  
    private final String mPhrase;  
  
    private final Pattern mPattern;  
  
    private Matcher mPhraseMatcher;  
    private final int mMinSplitSize;  
    private int mOffset = 0;  
    ...
```

*Contains a matcher that searches
for the phrase in the input*

Using Java Spliterator in SearchStreamGang

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a phrase appears in an input string

```
class PhraseMatchSpliterator implements Spliterator<Result> {  
    private CharSequence mInput;  
  
    private final String mPhrase;  
  
    private final Pattern mPattern;  
  
    private Matcher mPhraseMatcher;  
    private final int mMinSplitSize;  
    Dictates the minimum size to perform a split  
    private int mOffset = 0;  
    ...
```

Using Java Spliterator in SearchStreamGang

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a phrase appears in an input string

```
class PhraseMatchSpliterator implements Spliterator<Result> {  
    ...  
    PhraseMatchSpliterator(CharSequence input, String phrase) {  
        String regexPhrase = "\b" + phrase.trim().replaceAll  
            ("\\s+", "\\\\b\\\\\\s+\\\\b")  
            + "\\b"; ...  
  
        mPattern = Pattern.compile(regexPhrase,  
            Pattern.CASE_INSENSITIVE | Pattern.DOTALL);  
        mPhraseMatcher = mPattern.matcher(input);  
        mInput = input; mPhrase = phrase;  
        mMinSplitSize = input.length() / 2;  
    } ...
```

See docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html

Using Java Spliterator in SearchStreamGang

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a phrase appears in an input string

```
class PhraseMatchSpliterator implements Spliterator<Result> {  
    ...  
    PhraseMatchSpliterator(CharSequence input, String phrase) {  
        String regexPhrase = "\\b" + phrase.trim().replaceAll  
            ("\\s+", "\\\\b\\\\\\s+\\\\\\b")  
            + "\\b"; ...  
  
        mPattern = Pattern.compile(regexPhrase,  
            Pattern.CASE_INSENSITIVE | Pattern.DOTALL);  
        mPhraseMatcher = mPattern.matcher(input);  
        mInput = input; mPhrase = phrase;  
        mMinSplitSize = input.length() / 2;  
    } ...  
}
```

A matcher is created to search the input for the regex pattern

See docs.oracle.com/javase/8/docs/api/java/util/regex/Matcher.html

Using Java Spliterator in SearchStreamGang

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a phrase appears in an input string

```
class PhraseMatchSpliterator implements Spliterator<Result> {  
    ...  
    PhraseMatchSpliterator(CharSequence input, String phrase) {  
        String regexPhrase = "\b" + phrase.trim().replaceAll  
            ("\\s+", "\\\\b\\\\\\s+\\\\b")  
            + "\\b"; ...  
  
        mPattern = Pattern.compile(regexPhrase,  
            Pattern.CASE_INSENSITIVE | Pattern.DOTALL);  
        mPhraseMatcher = mPattern.matcher(input);  
        mInput = input; mPhrase = phrase;  
        mMinSplitSize = input.length() / 2; Define the min split size  
    } ...
```

Using Java Spliterator in SearchStreamGang

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a phrase appears in an input string

```
class PhraseMatchSpliterator implements Spliterator<Result> {  
    ...  
    boolean tryAdvance(Consumer<? super Result> action) {  
        if (!mPhraseMatcher.find())  
            return false;  
  
        else {  
            action.accept(new Result  
                (mOffset + mPhraseMatcher.start()));  
            return true;  
        }  
    }  
    ...
```

Called by the Java 8 streams framework to attempt to advance the spliterator by one word match

Using Java Spliterator in SearchStreamGang

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a phrase appears in an input string

```
class PhraseMatchSpliterator implements Spliterator<Result> {  
    ...  
    boolean tryAdvance(Consumer<? super Result> action) {  
        if (!mPhraseMatcher.find())  
            return false;  
  
        else {  
            action.accept(new Result  
                (mOffset + mPhraseMatcher.start()));  
            return true;  
        }  
    }  
    ...
```

Passes the result (if any) back "by reference" to the streams framework

Using Java Spliterator in SearchStreamGang

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a phrase appears in an input string

```
class PhraseMatchSpliterator implements Spliterator<Result> {  
    ...  
    boolean tryAdvance(Consumer<? super Result> action) {  
        if (!mPhraseMatcher.find())  
            return false;  
        else {  
            action.accept(new Result  
                (mOffset + mPhraseMatcher.start()));  
            return true;  
        }  
    }  
    ...
```

*Check if any remaining phrases
in the input match the regex*

Using Java Spliterator in SearchStreamGang

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a phrase appears in an input string

```
class PhraseMatchSpliterator implements Spliterator<Result> {  
    ...  
    boolean tryAdvance(Consumer<? super Result> action) {  
        if (!mPhraseMatcher.find())  
            return false;  
        else {  
            action.accept(new Result  
                (mOffset + mPhraseMatcher.start()));  
            return true;  
        }  
    }  
    ...
```

Inform the streams framework to cease calling tryAdvance() if there's no match

Using Java Spliterator in SearchStreamGang

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a phrase appears in an input string

```
class PhraseMatchSpliterator implements Spliterator<Result> {  
    ...  
    boolean tryAdvance(Consumer<? super Result> action) {  
        if (!mPhraseMatcher.find())  
            return false;  
  
        else {  
            action.accept(new Result  
                (mOffset + mPhraseMatcher.start()));  
            return true;  
        }  
    }  
    ...  
}
```

accept() stores the index in the input string where the match occurred, which is returned to the streams framework

Using Java Spliterator in SearchStreamGang

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a phrase appears in an input string

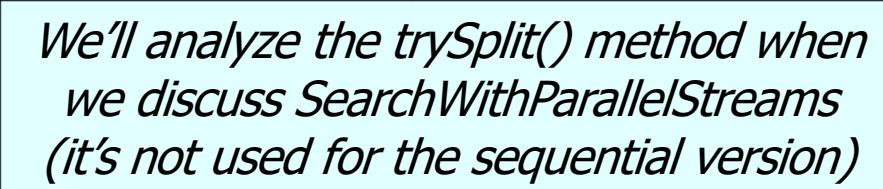
```
class PhraseMatchSpliterator implements Spliterator<Result> {  
    ...  
    boolean tryAdvance(Consumer<? super Result> action) {  
        if (!mPhraseMatcher.find())  
            return false;  
  
        else {  
            action.accept(new Result  
                (mOffset + mPhraseMatcher.start()));  
            return true;  
        }  
    }  
    ...
```

*Inform the streams framework
to continue calling tryAdvance()*

Using Java Spliterator in SearchStreamGang

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a phrase appears in an input string

```
class PhraseMatchSpliterator implements Spliterator<Result> {  
    ...  
    public Spliterator<SearchResults.Result> trySplit() {  
        ...  
    }  
    ...
```



*We'll analyze the `trySplit()` method when
we discuss `SearchWithParallelStreams`
(it's not used for the sequential version)*

Pros of the SearchWith SequentialStreams Class

Pros of the SearchWithSequentialStreams Class

- There are several benefits with this sequential streams implementation

```
List<SearchResults> processInput(CharSequence inputSeq) {  
    String title = getTitle(inputString);  
    CharSequence input = inputSeq.subSequence(...);  
  
    List<SearchResults> results = mPhrasesToFind  
        .stream()  
        .map(phrase -> searchForPhrase  
            (phrase, input, title, false))  
  
        .filter(not(SearchResults::isEmpty))  
  
        .collect(toList());  
    return results;  
}
```



Pros of the SearchWithSequentialStreams Class

- There are several benefits with this sequential streams implementation

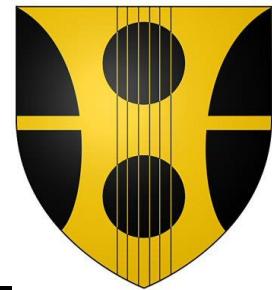
```
List<SearchResults> processInput(CharSequence inputSeq) {  
    String title = getTitle(inputString);  
    CharSequence input = inputSeq.subSequence(...);  
  
    List<SearchResults> results = mPhrasesToFind  
        .stream()  
        .map(phrase -> searchForPhrase  
            (phrase, input, title, false))  
  
        .filter(not(SearchResults::isEmpty))  
        .collect(toList());  
    return results;  
}
```

Streams use "internal" iterators versus "external" iterators used by collections

Pros of the SearchWithSequentialStreams Class

- There are several benefits with this sequential streams implementation

```
List<SearchResults> processInput(CharSequence inputSeq) {  
    String title = getTitle(inputString);  
    CharSequence input = inputSeq.subSequence(...);  
  
    List<SearchResults> results = mPhrasesToFind  
        .stream()  
        .map(phrase -> searchForPhrase  
            (phrase, input, title, false))  
  
        .filter(not(SearchResults::isEmpty))  
  
        .collect(toList());  
    return results;  
}
```



Internal iterators shield programs from streams processing implementation details

Pros of the SearchWithSequentialStreams Class

- There are several benefits with this sequential streams implementation

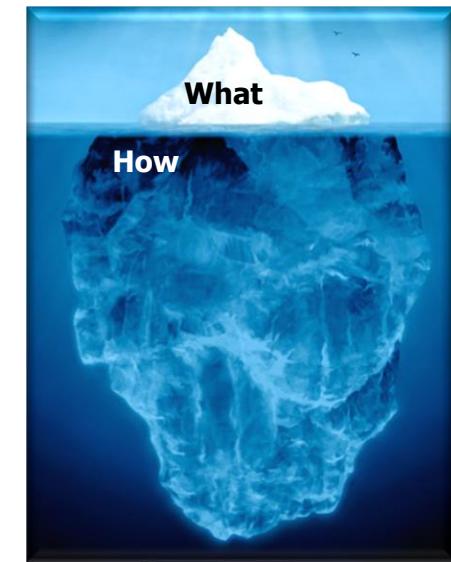
```
List<SearchResults> processInput(CharSequence inputSeq) {  
    String title = getTitle(inputString);  
    CharSequence input = inputSeq.subSequence(...);  
  
    List<SearchResults> results = mPhrasesToFind  
        .stream()  
        .map(phrase -> searchForPhrase  
            (phrase, input, title, false))  
  
        .filter(not(SearchResults::isEmpty))  
        .collect(toList());  
    return results;  
}
```

This pipeline is declarative since it's a series of transformations performed by aggregate operations

Pros of the SearchWithSequentialStreams Class

- There are several benefits with this sequential streams implementation

```
List<SearchResults> processInput(CharSequence inputSeq) {  
    String title = getTitle(inputString);  
    CharSequence input = inputSeq.subSequence(...);  
  
    List<SearchResults> results = mPhrasesToFind  
        .stream()  
        .map(phrase -> searchForPhrase  
            (phrase, input, title, false))  
  
        .filter(not(SearchResults::isEmpty))  
  
        .collect(toList());  
    return results;  
}
```



Focus on “what” operations to perform, rather than on “how” they’re implemented

Pros of the SearchWithSequentialStreams Class

- There are several benefits with this sequential streams implementation

```
List<SearchResults> processInput(CharSequence inputSeq) {  
    String title = getTitle(inputString);  
    CharSequence input = inputSeq.subSequence(...);  
  
    List<SearchResults> results = mPhrasesToFind  
        .stream()  
        .map(phrase -> searchForPhrase  
            (phrase, input, title, false))  
  
        .filter(not(SearchResults::isEmpty))  
  
        .collect(toList());  
    return results;  
}
```



These lambda functions have no side-effects

Pros of the SearchWithSequentialStreams Class

- There are several benefits with this sequential streams implementation

```
List<SearchResults> processInput(CharSequence inputSeq) {  
    String title = getTitle(inputString);  
    CharSequence input = inputSeq.subSequence(...);  
  
    List<SearchResults> results = mPhrasesToFind  
        .stream()  
        .map(phrase -> searchForPhrase  
            (phrase, input, title, false))  
  
        .filter(not(SearchResults::isEmpty))  
  
        .collect(toList());  
    return results;  
}
```

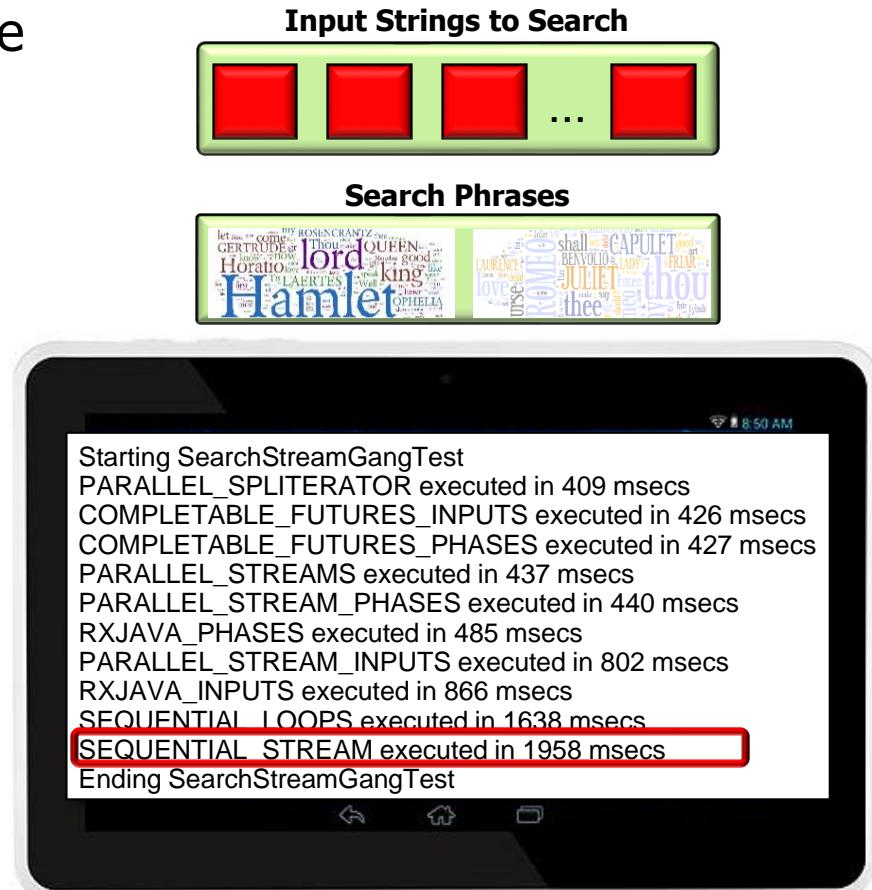
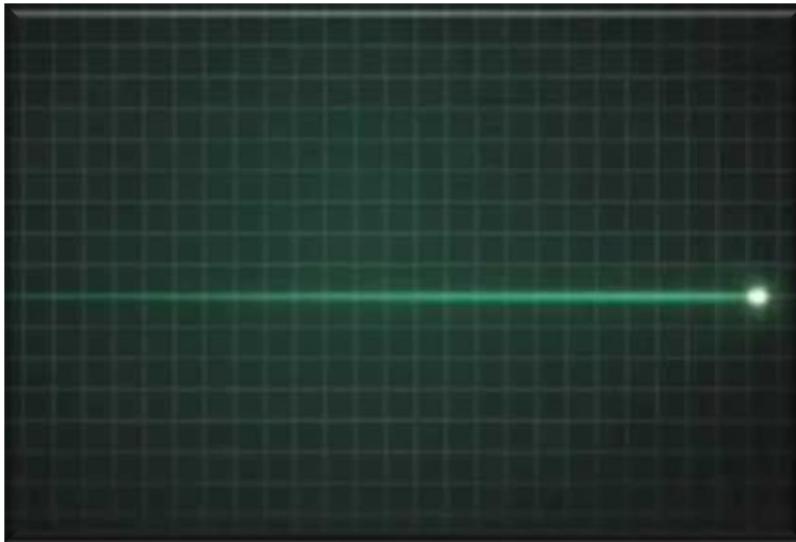


No side-effects makes it easier to reason about behavior & enables optimization

Cons of the SearchWith SequentialStreams Class

Cons of the SearchWithSequentialStreams Class

- The sequential implementation can't take advantage of multi-core processors



Tests conducted on a quad-core Lenovo P50 with 32 Gbytes of RAM

Cons of the SearchWithSequentialStreams Class

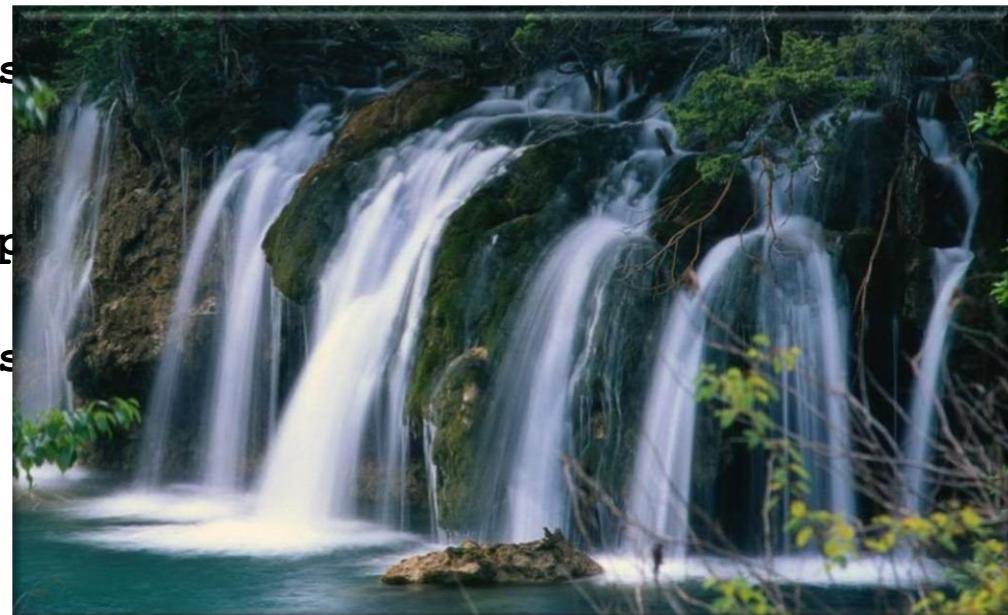
- This class only used a few Java 8 aggregate operations

```
List<SearchResults> processInput(CharSequence inputSeq) {  
    String title = getTitle(inputString);  
    CharSequence input = inputSeq.subSequence(...);  
  
    List<SearchResults> results = mPhrasesToFind  
        .stream()  
        .map(phrase  
            -> searchForPhrase(phrase, input, title))  
  
        .filter(not(SearchResults::isEmpty))  
  
        .collect(toList());  
    return results; ...}
```

Cons of the SearchWithSequentialStreams Class

- This class only used a few Java 8 aggregate operations

```
List<SearchResults> processInput(CharSequence inputSeq) {  
    String title = getTitle(inputString);  
    CharSequence input = inputSeq.subSequence(...);  
  
    List<SearchResults> results =  
        .stream()  
        .map(phrase  
            -> searchForPhrase(p...  
  
        .filter(not(SearchResults.  
            .collect(toList());  
return results; ...
```



However, these aggregate operations are also useful for parallel streams

Cons of the SearchWithSequentialStreams Class

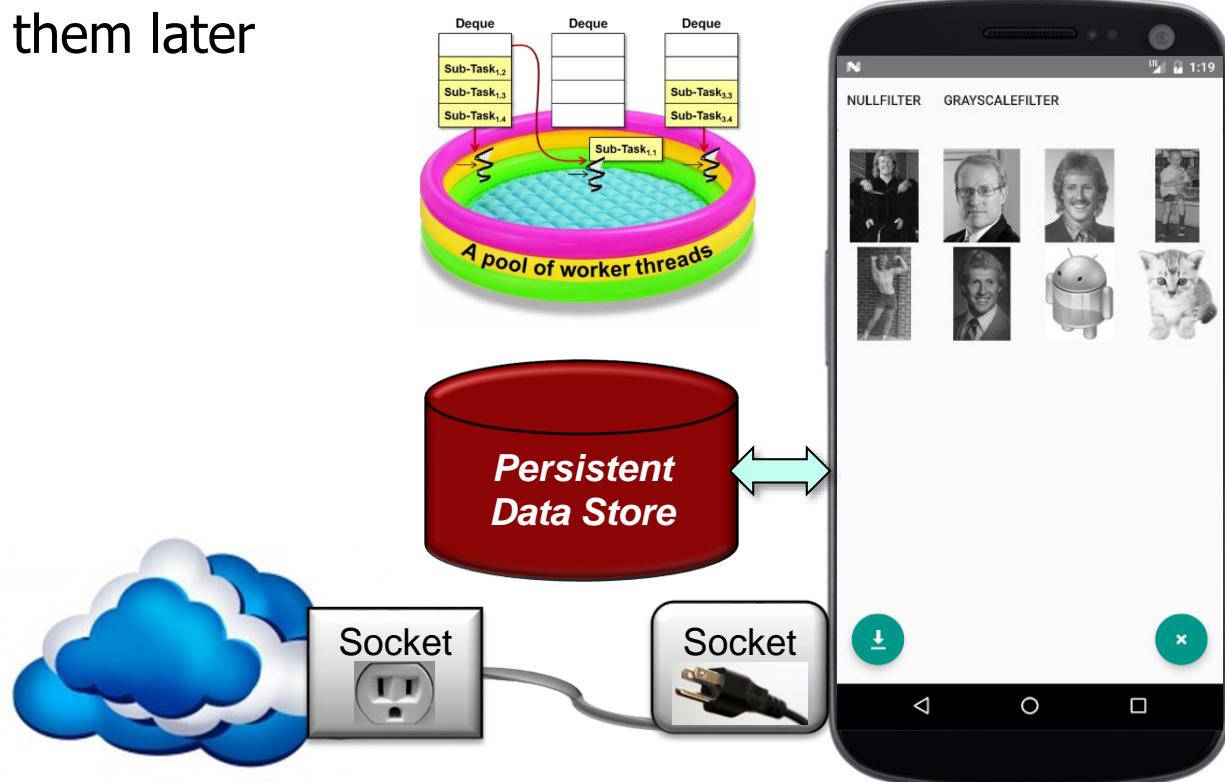
- Many other aggregate operations are part of the Java 8 stream API

Modifier and Type	Method and Description
boolean	<code>allMatch(Predicate<? super T> predicate)</code> Returns whether all elements of this stream match the provided predicate.
boolean	<code>anyMatch(Predicate<? super T> predicate)</code> Returns whether any elements of this stream match the provided predicate.
static <T> Stream.Builder<T>	<code>builder()</code> Returns a builder for a Stream.
<R,A> R	<code>collect(Collector<? super T,A,R> collector)</code> Performs a mutable reduction operation on the elements of this stream using a Collector.
<R> R	<code>collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)</code> Performs a mutable reduction operation on the elements of this stream.
static <T> Stream<T>	<code>concat(Stream<? extends T> a, Stream<? extends T> b)</code> Creates a lazily concatenated stream whose elements are all the elements of the first stream followed by all the elements of the second stream.
long	<code>count()</code> Returns the count of elements in this stream.
Stream<T>	<code>distinct()</code> Returns a stream consisting of the distinct elements (according to <code>Object.equals(Object)</code>) of this stream.
static <T> Stream<T>	<code>empty()</code> Returns an empty sequential Stream.
Stream<T>	<code>filter(Predicate<? super T> predicate)</code> Returns a stream consisting of the elements of this stream that match the given predicate.
Optional<T>	<code>findAny()</code> Returns an <code>Optional</code> describing some element of the stream, or an empty <code>Optional</code> if the stream is empty.
Optional<T>	<code>findFirst()</code> Returns an <code>Optional</code> describing the first element of this stream, or an empty <code>Optional</code> if the stream is empty.
<R> Stream<R>	<code>flatMap(Function<? super T,? extends Stream<? extends R>> mapper)</code> Returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.

See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

Cons of the SearchWithSequentialStreams Class

- Many other aggregate operations are part of the Java 8 stream API
 - We'll cover more of them later



See "Java 8 Parallel ImageStreamGang Example"

End of Java 8 Sequential SearchStreamGang Example (Part 3)