Overview of Java 8 Foundations

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

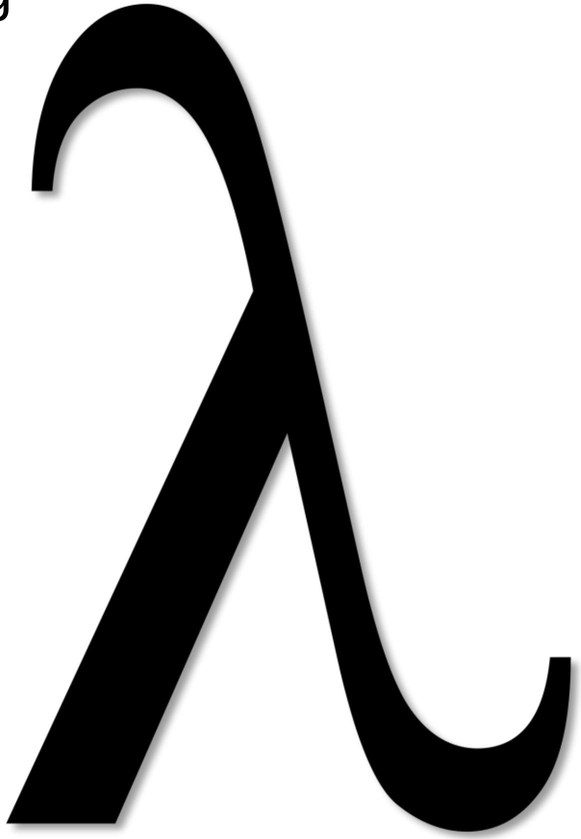
Professor of Computer Science

Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

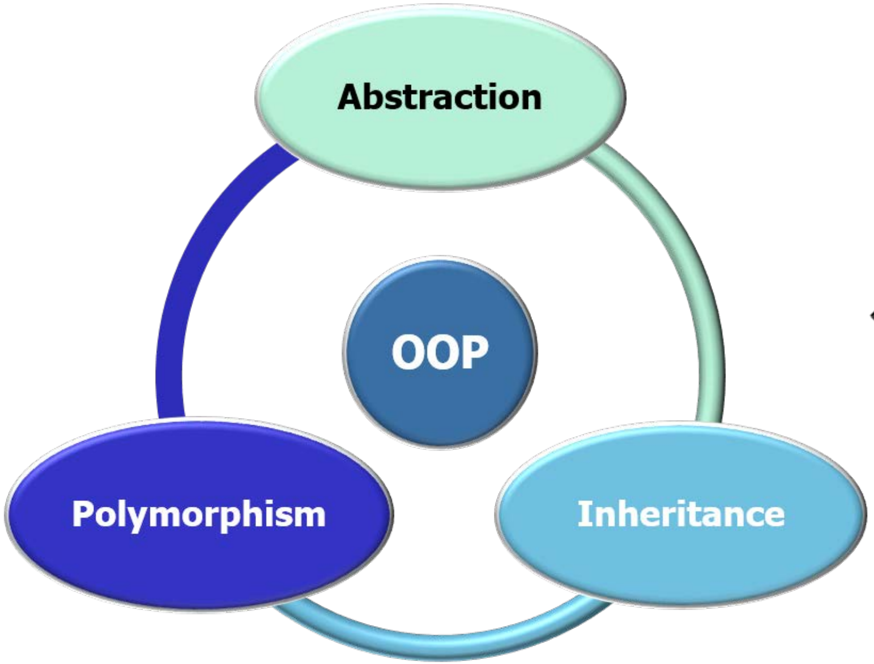
Learning Objectives in this Lesson

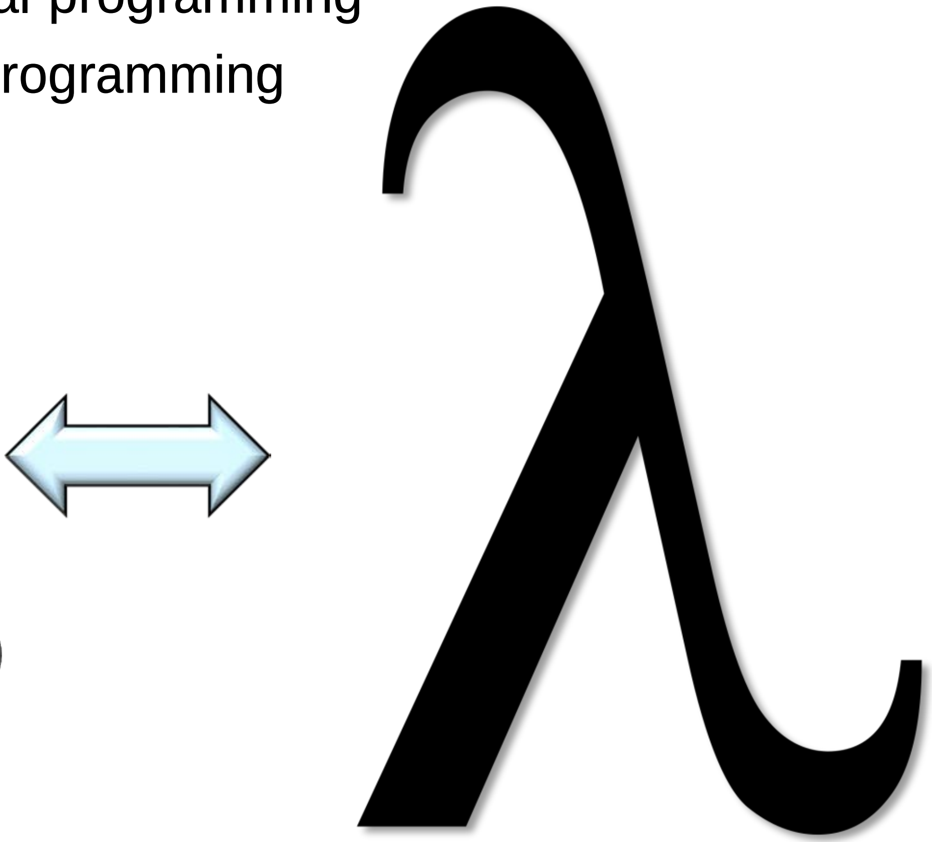
- Understand key aspects of functional programming



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

Setting the Context

Setting the Context

- This course is based on material we teach at Vanderbilt

CS 279. Software Engineering Project. Students work in teams to specify, design, implement, document, and test a nontrivial software project. The use of CASE (Computer-Assisted Software Engineering) tools is stressed. Prerequisite: CS 278. SPRING. [3]

CS 278. Principles of Software Engineering. The nature of software. The object-oriented paradigm. Software life-cycle models. Requirements, specification, design, implementation, documentation, and testing of software. Object-oriented analysis and design. Software maintenance. Prerequisite: CS 251. FALL. [3]

CS 251. Intermediate Software Design. High quality development and reuse of architectural patterns, design patterns, and software components. Theoretical and practical aspects of developing, documenting, testing, and applying reusable class libraries and object-oriented frameworks using object-oriented and component-based programming languages and tools. Prerequisite: CS 201. FALL, SPRING. [3]

CS 101. Programming and Problem Solving. An intensive introduction to algorithm development and problem solving on the computer. Structured problem definition, top down and modular algorithm design. Running, debugging, and testing programs. Program documentation. FALL, SPRING. [3]

CS 282. Principles of Operating Systems II. Projects involving modification of a current operating system. Lectures on memory management policies, including virtual memory. Protection and sharing of information, including general models for implementation of various degrees of sharing. Resource allocation in general, including deadlock detection and prevention strategies. Introduction to operating system performance measurement, for both efficiency and logical correctness. Two hours lecture and one hour laboratory. Prerequisite: CS 281. SPRING. [3]

CS 281. Principles of Operating Systems I. Resource allocation and control functions of operating systems. Scheduling of processes and processors. Concurrent processes and primitives for their synchronization. Use of parallel processes in designing operating system subsystems. Methods of implementing parallel processes on conventional computers. Virtual memory, paging, protection of shared and non-shared information. Structures of data files in secondary storage Security issues. Case studies. Prerequisite: CS 231, CS 251. FALL, SPRING. [3]

CS 201. Program Design and Data Structures. Continuation of CS 101. The study of elementary data structures, their associated algorithms and their application in problems; rigorous development of programming techniques and style; design and implementation of programs with multiple modules, using good data structures and good programming style. Prerequisite: CS 101. FALL, SPRING. [3]

See www.dre.vanderbilt.edu/~schmidt/courses.html

Setting the Context

- This course is based on material we teach at Vanderbilt

CS 279. Software Engineering Project. Students work in teams to specify, design, implement, document, and test a nontrivial software project. The use of CASE (Computer-Assisted Software Engineering) tools is stressed. Prerequisite: CS 27[illegible] SPRING. [3]

CS 278. Principles of Software Engineering. The nature of software. The object-oriented paradigm. Software life-cycle models. Requirements, specification, design, implementation, documentation, and testing of software. Object-oriented analysis and design. Software maintenance. Prerequisite: CS 251. FALL. [3]

CS 251. Intermediate Software Design. High quality development and reuse of architectural patterns, design patterns, and software components. Theoretical and practical aspects of developing, documenting, testing, and applying reusable class libraries and object-oriented frameworks using object-oriented and component-based programming languages and tools. Prerequisite: CS 201. FALL, SPRING. [3]

CS 101. Programming and Problem Solving. An intensive introduction to algorithm development and problem solving on the computer. Structured problem definition, top down and modular algorithm design. Running, debugging, and testing programs. Program documentation. FALL, SPRING. [3]

CS 282. Principles of Operating Systems II. Projects involving modification of a current operating system. Lectures on memory management policies, including virtual memory. Protection and sharing of information, including general models for implementation of various degrees of sharing. Resource allocation in general, including deadlock detection and prevention strategies. Introduction to operating system performance measurement, for both efficiency and logical correctness. Two hours lecture and one hour laboratory. Prerequisite: CS 281. SPRING. [3]

CS 281. Principles of Operating Systems I. Resource allocation and control functions of operating systems. Scheduling of processes and processors. Concurrent processes and primitives for their synchronization. Use of parallel processes in designing operating system subsystems. Methods of implementing parallel processes on conventional computers. Virtual memory, paging, protection of shared and non-shared information. Structures of data files in secondary storage Security issues. Case studies. Prerequisite: CS 231, CS 251. FALL, SPRING. [3]

CS 201. Program Design and Data Structures. Continuation of CS 101. The study of elementary data structures, their associated algorithms and their application in problems; rigorous development of programming techniques and style; design and implementation of programs with multiple modules, using good data structures and good programming style. Prerequisite: CS 101. FALL, SPRING. [3]

Our Vanderbilt courses follow a particular sequence

Setting the Context

- However, I don’t know what you know or whether you’re prepared or not!

Setting the Context

- Therefore, we'll start with a brief overview of foundational Java 8 functional programming concepts & features
  - e.g., lambda expressions, method references, & functional interfaces

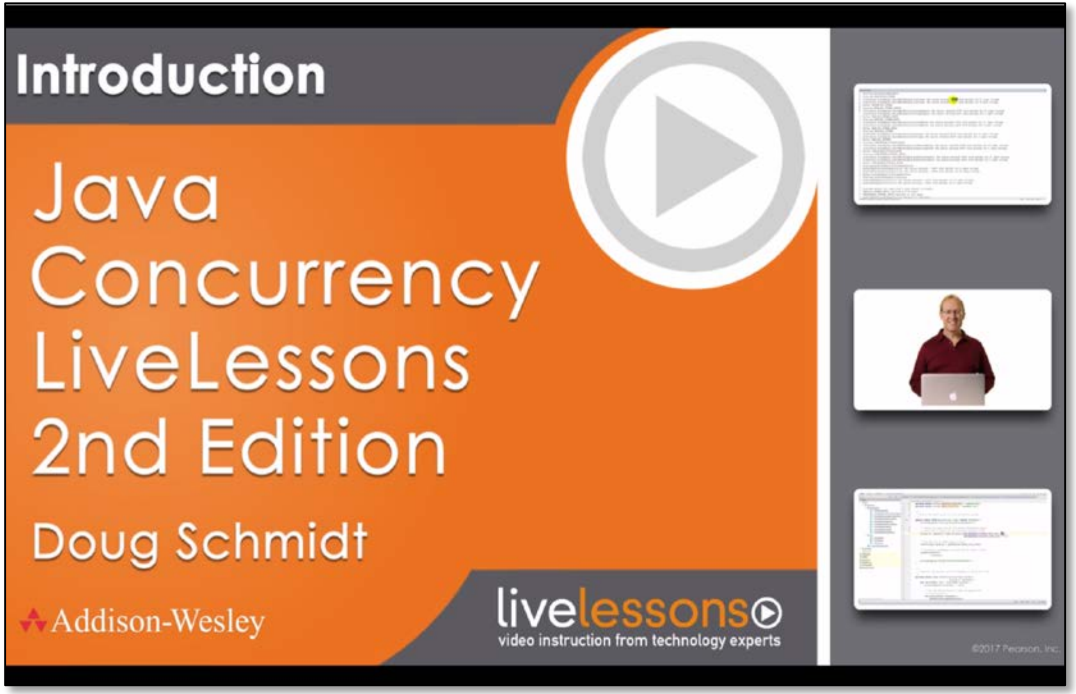
These features are the foundation for Java 8's concurrency/parallelism frameworks

Setting the Context

- We're going to cover this material quickly so we can focus on the Java 8 concurrency & parallelism frameworks

Setting the Context

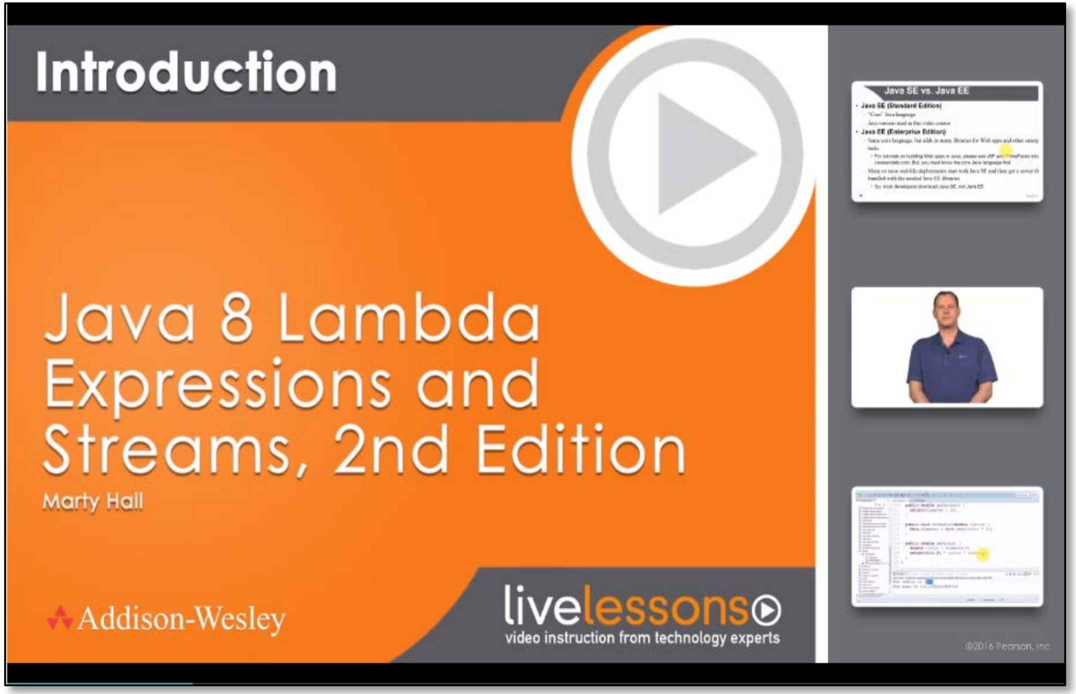
- My LiveLessons course on Java Concurrency covers concurrency & parallelism in Java 7 & Java 8



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

Setting the Context

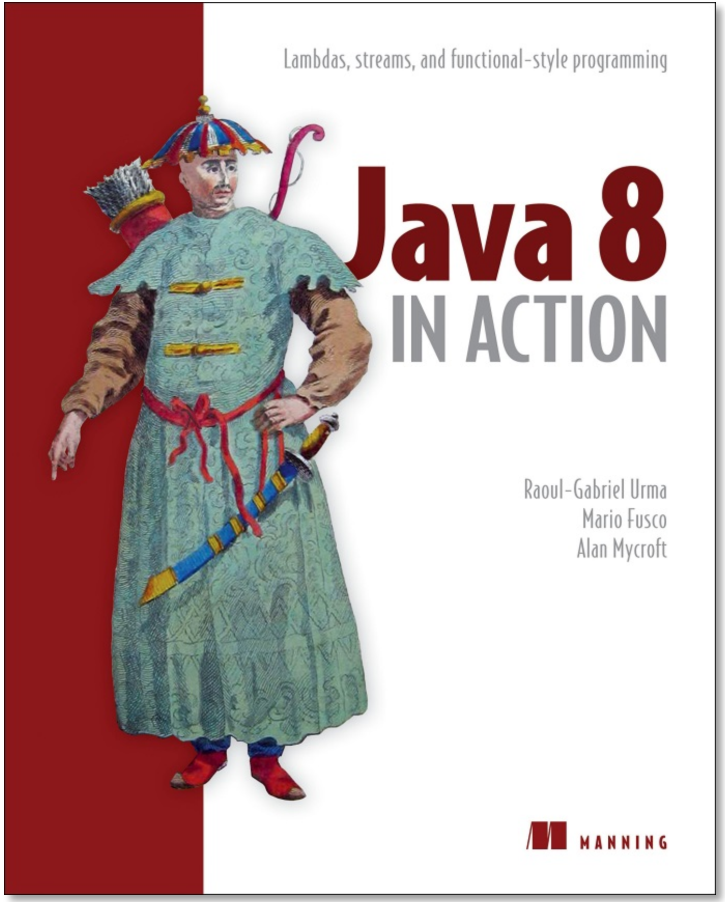
- Other LiveLessons courses cover Java 8 lambda expressions in even greater detail



See safari5.bvdep.com/video/programming/java/9780134383644

Setting the Context

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



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

Setting the Context

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

Processing Data with Java SE 8 Streams, Part 1
by Raoul-Gabriel Urma

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

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.

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

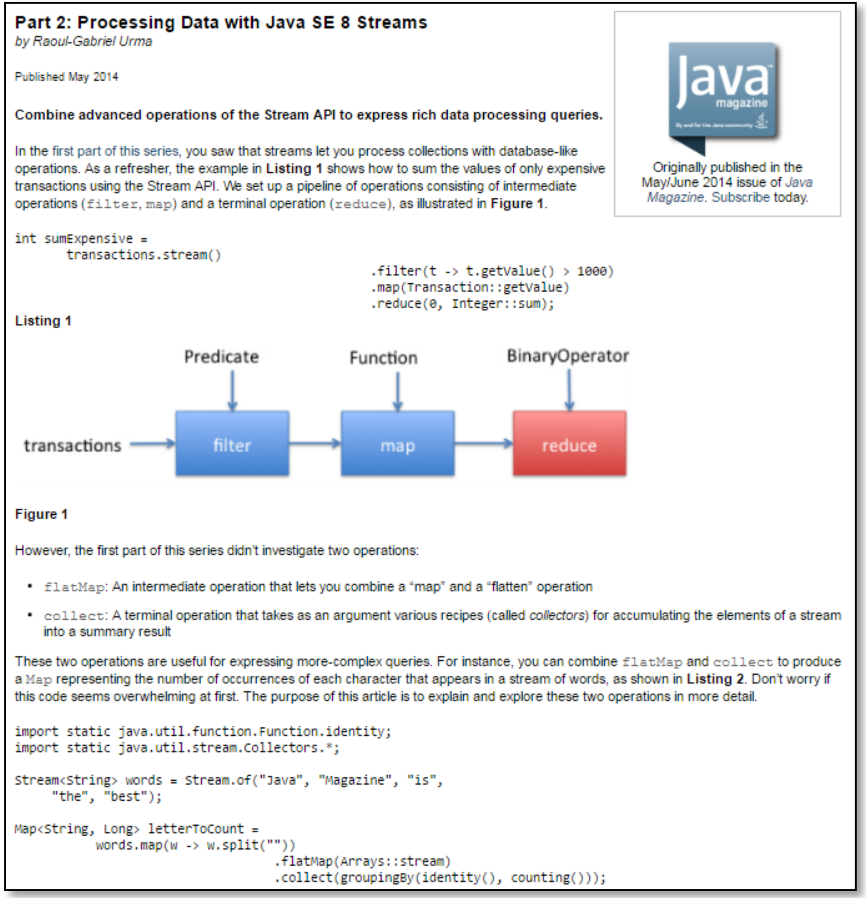
Before we explore in detail what you can do with streams, let's take a look at an example so you have a sense of the new programming style with Java SE 8 streams. Let's say we need to find all transactions of type `grocery` and return a list of transaction IDs sorted in decreasing order of transaction 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){
  transactionsIds.add(t.getId());
}
```

See www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html

Setting the Context

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



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.

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

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.

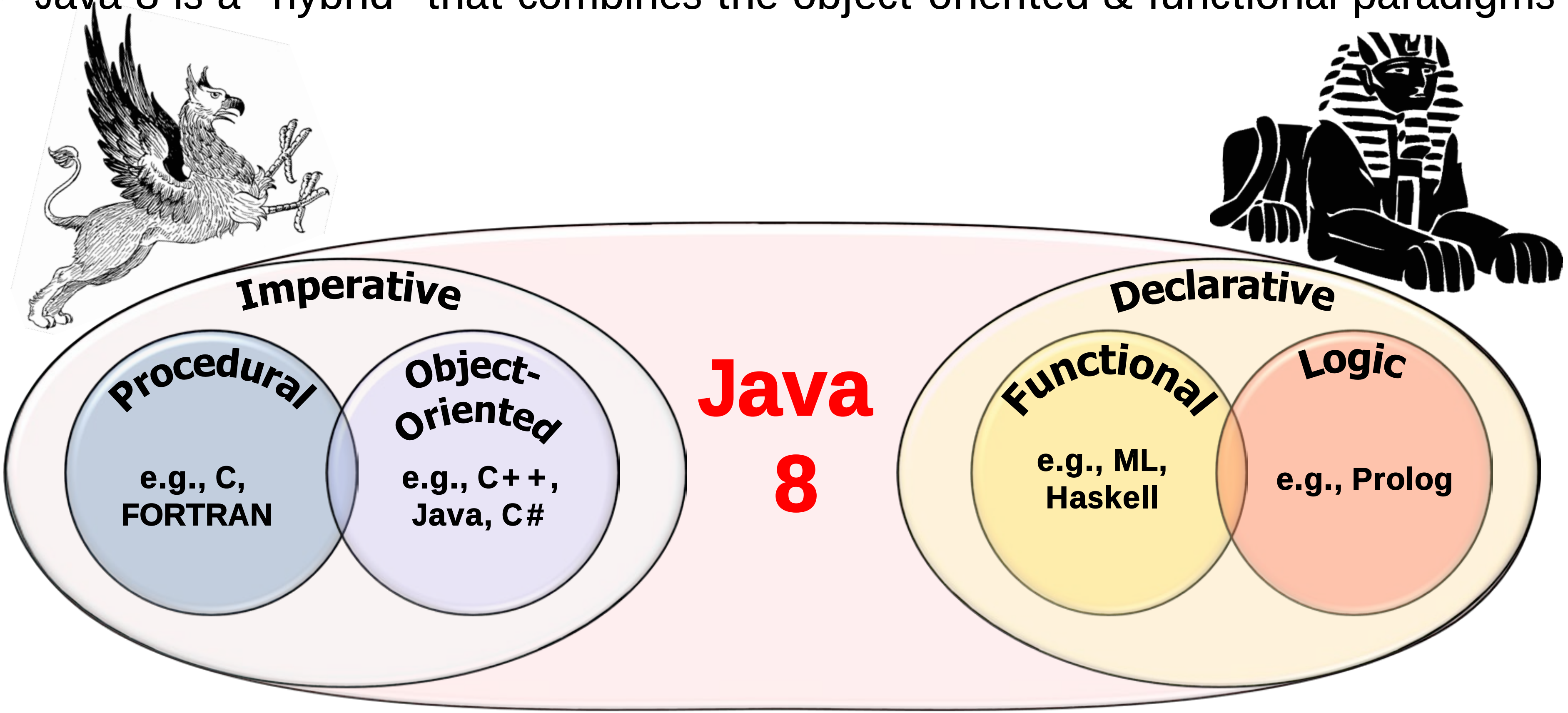
```
import static java.util.function.Function.identity;
import static java.util.stream.Collectors.*;

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()));
```

See www.oracle.com/technetwork/articles/java/architect-streams-pt2-2227132.html

Overview of Functional Programming in Java 8

Overview of Functional Programming in Java 8

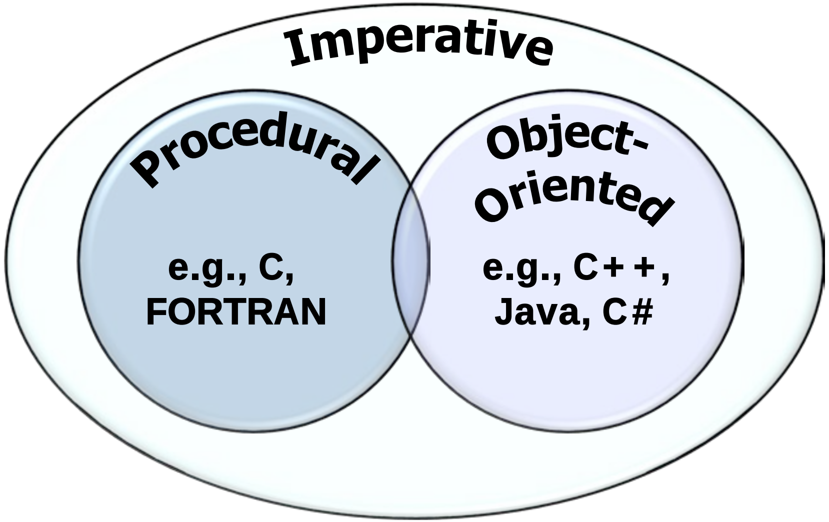
- Java 8 is a "hybrid" that combines the object-oriented & functional paradigms



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

Overview of Functional Programming in Java 8

- Object-oriented programming is an "imperative" paradigm

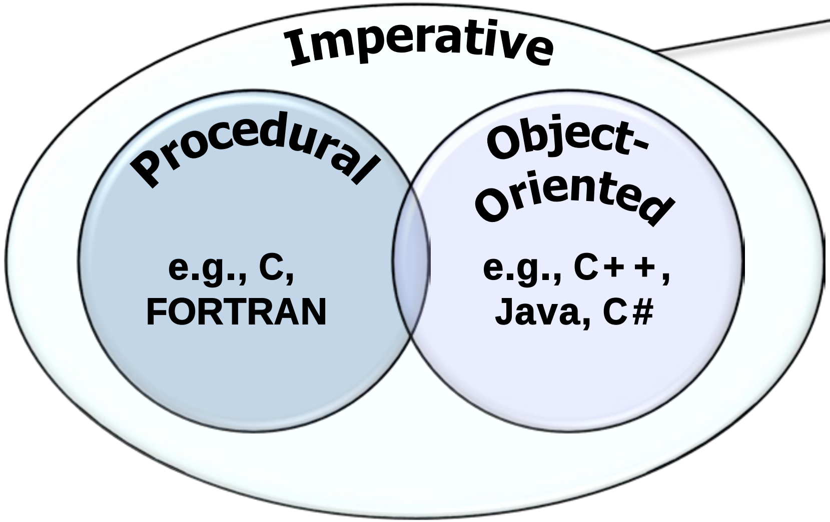


See en.wikipedia.org/wiki/Imperative_programming

Overview of Functional Programming in Java 8

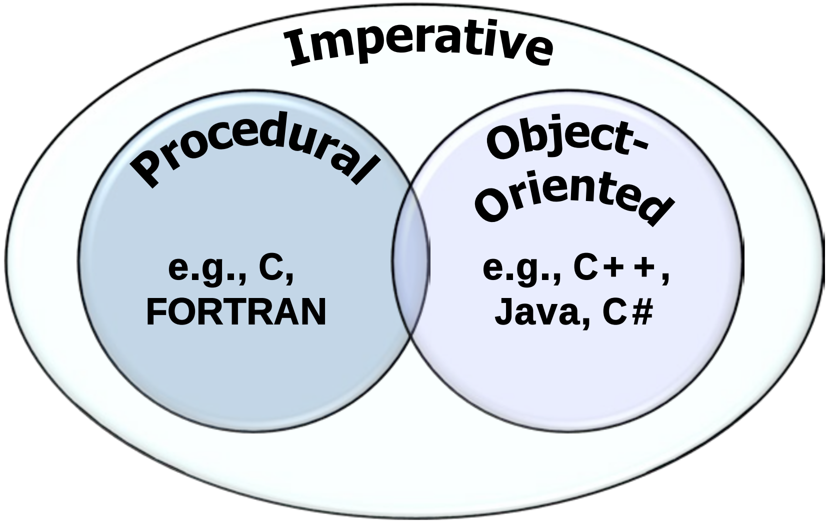
- Object-oriented programming is an "imperative" paradigm
  - e.g., a program consists of commands for the computer to perform

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



Overview of Functional Programming in Java 8

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



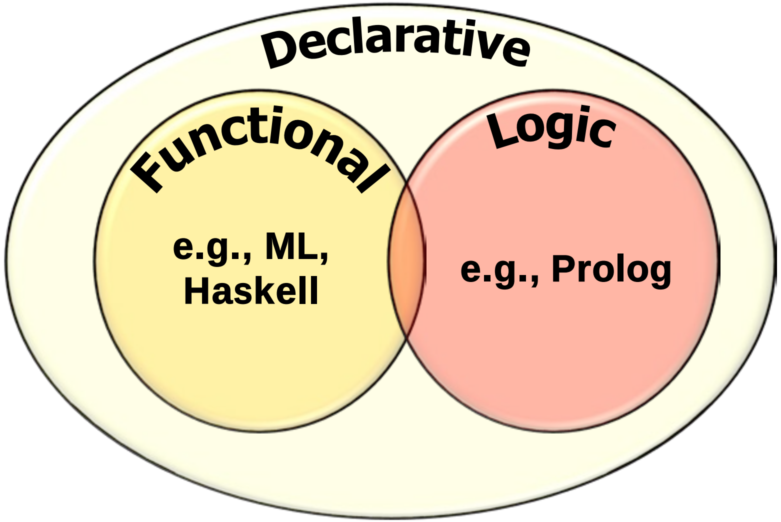

```
List<String> zap(List<String> lines,
                 String omit) {
  List<String> res =
    new ArrayList<>();
  for (String line : lines)
    if (!omit.equals(line))
      res.add(line);
  return res;
}
```

Imperatively remove a designated string from a list of strings

Note how this code is inherently sequential..

Overview of Functional Programming in Java 8

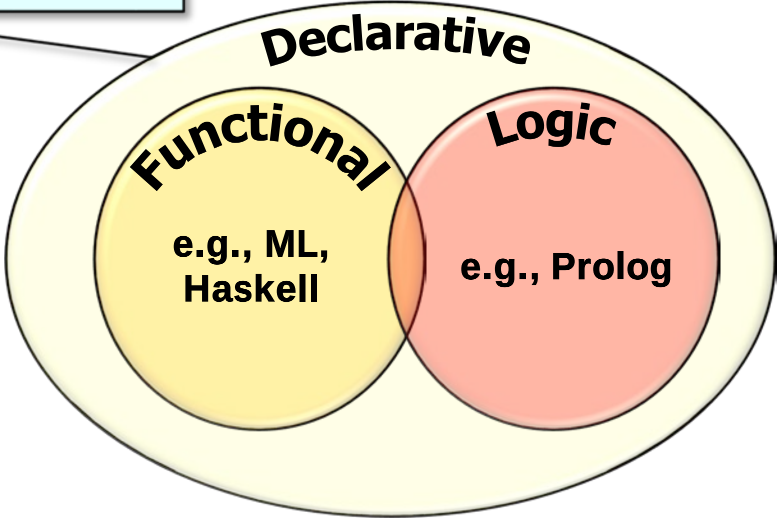
- Conversely, functional programming is a "declarative" paradigm



See en.wikipedia.org/wiki/Declarative_programming

- 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

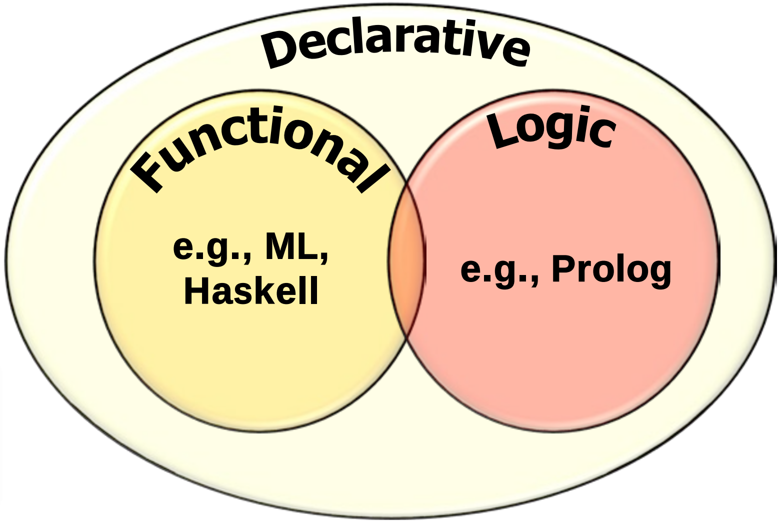


Overview of Functional Programming in Java 8

- 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(line ->
            !omit.equals(line))
    .collect(toList());
}
```

Declaratively remove a designated string from a list of strings

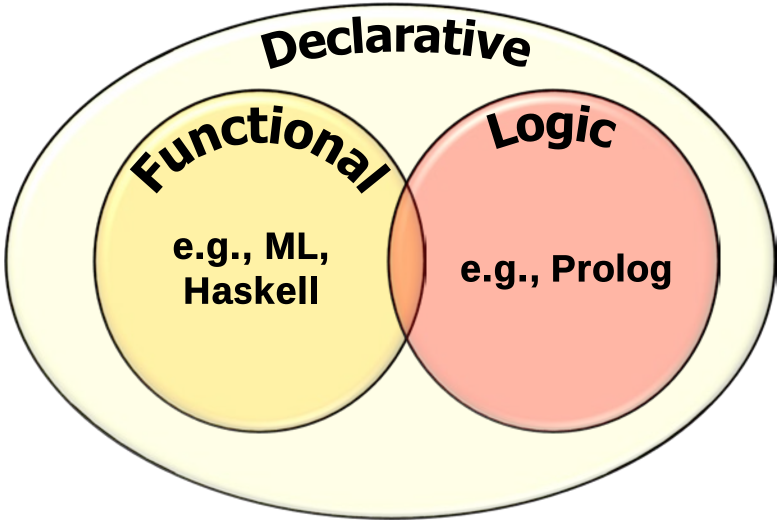


Overview of Functional Programming in Java 8

- 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(line ->
            !omit.equals(line))
    .collect(toList());
}
```

Note "fluent" programming style with cascading method calls



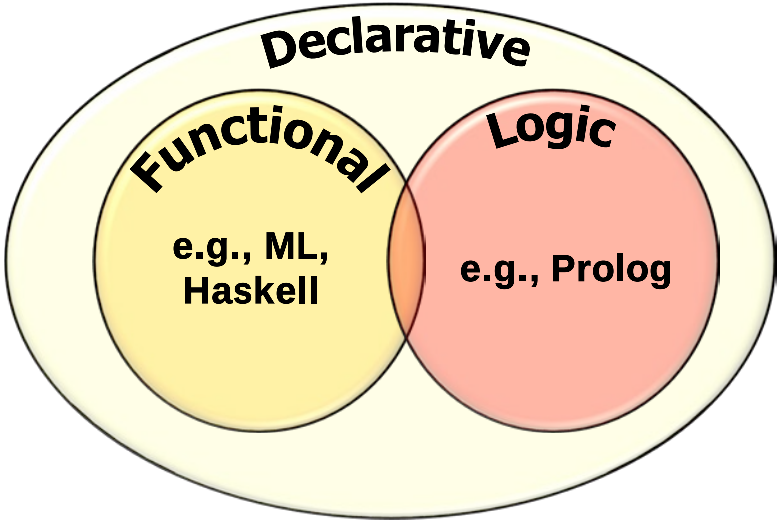
See en.wikipedia.org/wiki/Fluent_interface

Overview of Functional Programming 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(line ->
            !omit.equals(line))
    .collect(toList());
}
```

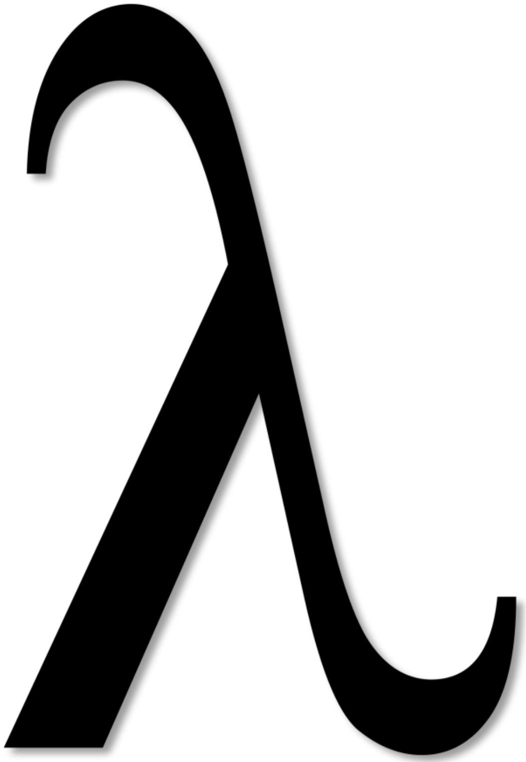
Perform the filtering in parallel



Note how this code is can be parallelized with minimal changes..

Overview of Functional Programming in Java 8

- Functional programming has its roots in lambda calculus

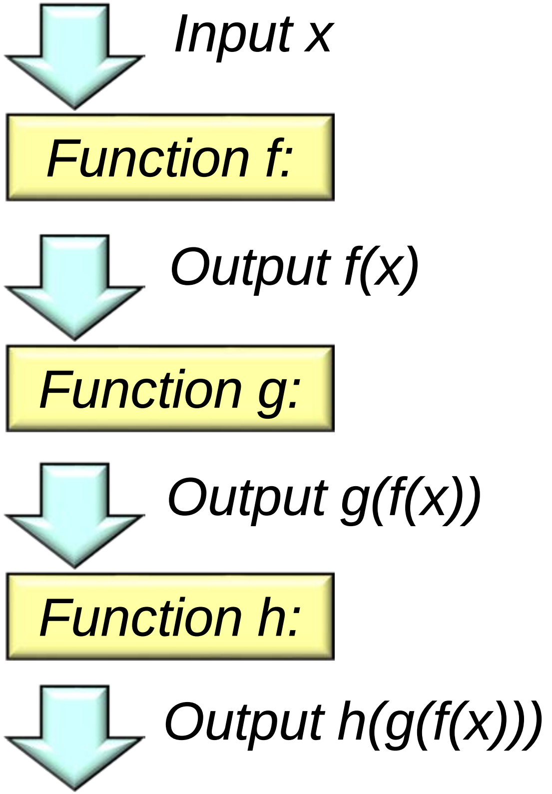


See en.wikipedia.org/wiki/Functional_programming

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

The output of one function serves as the input to the next function etc.

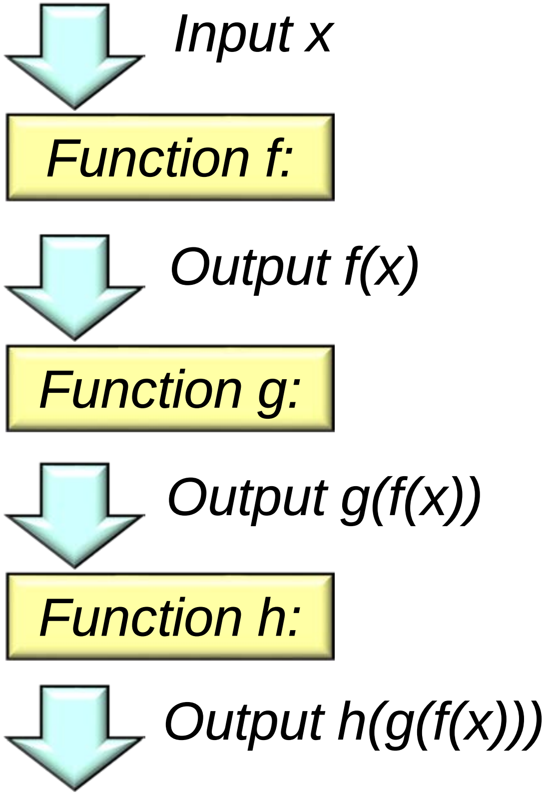


See en.wikipedia.org/wiki/Functional_programming#Pure_functions

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



See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex16

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 data are discouraged/avoided

See 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 data are discouraged/avoided

```
long parallelFactorial(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 github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex16

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 data are discouraged/avoided

```
long parallelFactorial(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 race conditions!

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

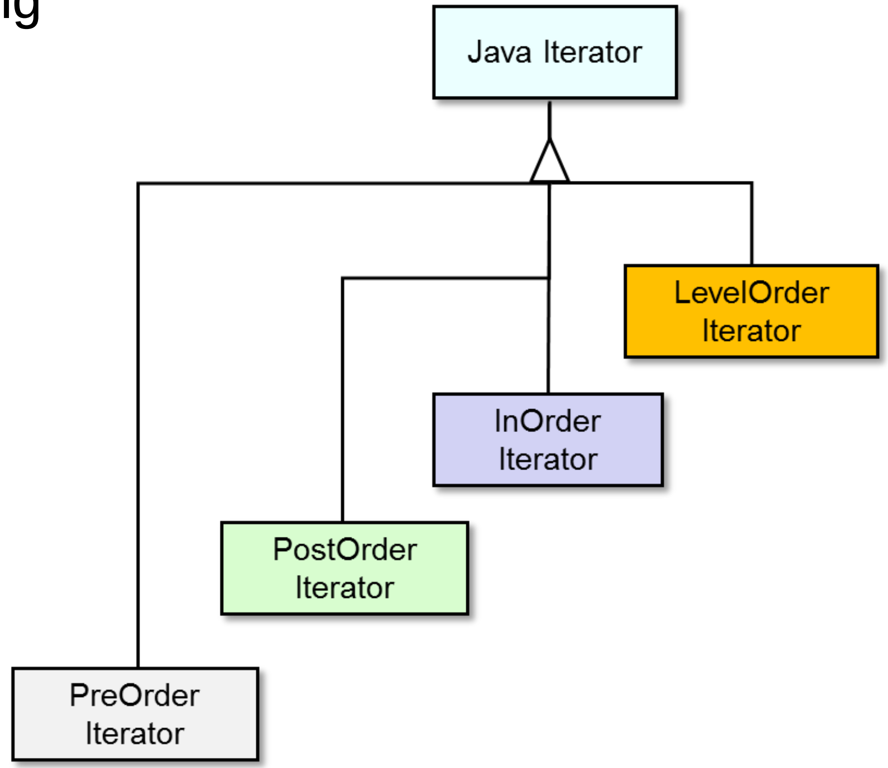
Overview of Functional Programming in Java 8

- Functional programming has its roots in lambda calculus, e.g.,
  - Computations are treated as the evaluation of mathematical functions
  - Changing state & mutable data are discouraged/avoided
  - Instead, the focus is on "immutable" objects
    - i.e., objects whose state cannot change after they are constructed

See docs.oracle.com/javase/tutorial/essential/concurrency/immutable.html

Overview of Functional Programming in Java 8

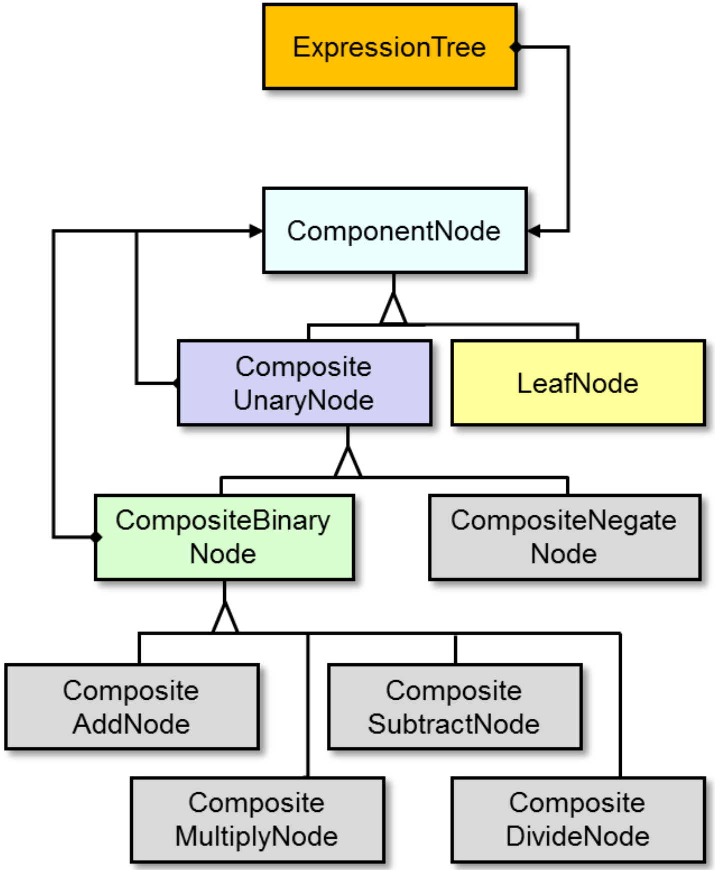
- In contrast, object-oriented programming employs "hierarchical data abstraction"



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

Overview of Functional Programming in Java 8

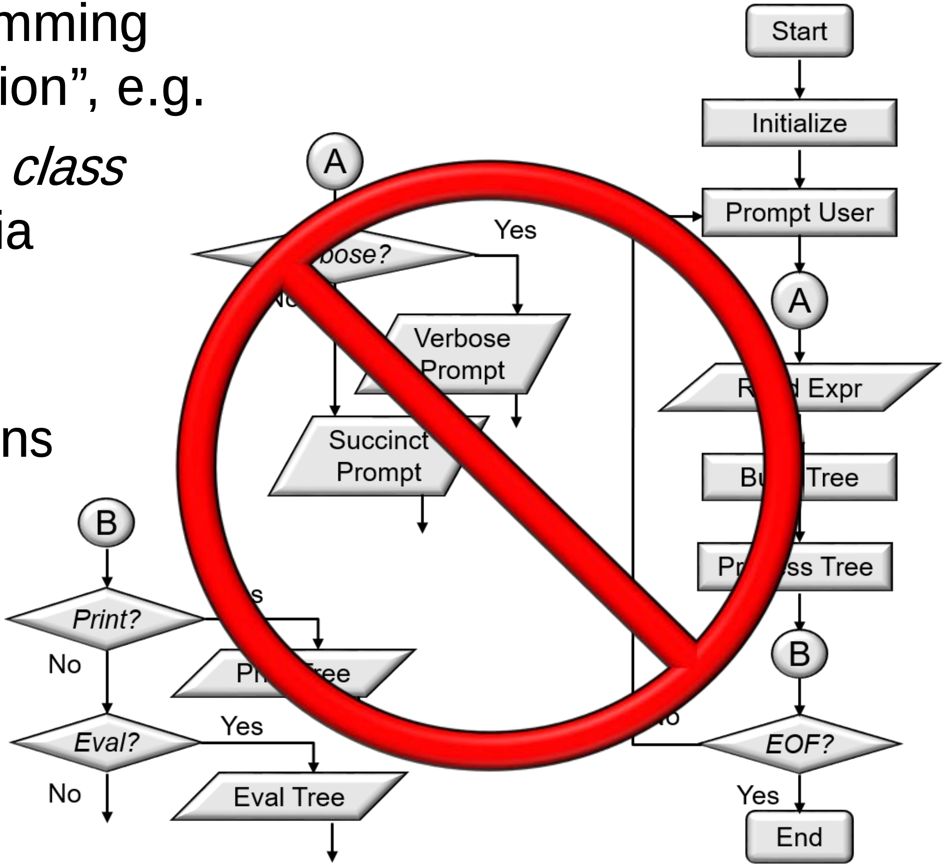
- In contrast, object-oriented programming employs "hierarchical data abstraction", e.g.
  - Components are based on stable *class* roles & relationships extensible via inheritance & dynamic binding



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



See www.drdobbs.com/windows/software-complexity-bringing-order-to-ch/199901062

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);
```

See en.wikipedia.org/wiki/Imperative_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
  - State is encapsulated by methods that perform imperative statements
    - This state is often mutable

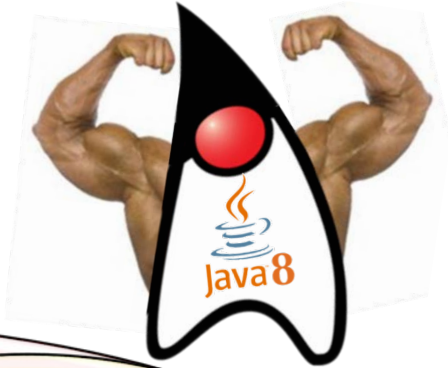
```
Tree tree = ...;
Visitor printVisitor =
    makeVisitor(...);

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

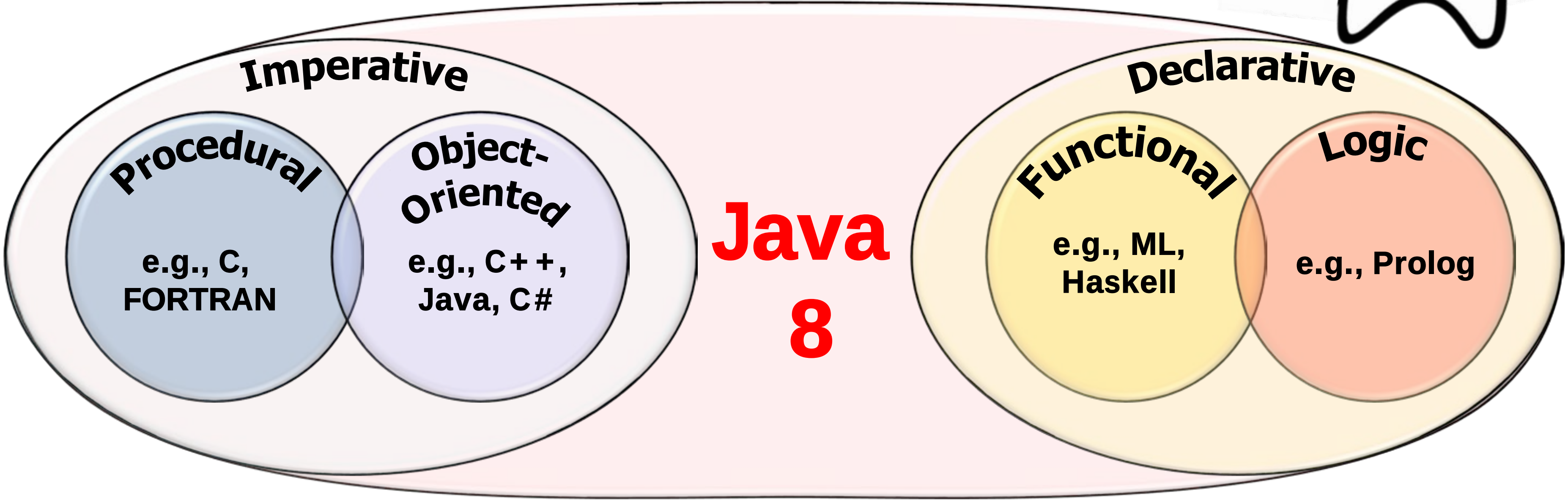
See en.wikipedia.org/wiki/Imperative_programming

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"

Download images that aren't already cached from a list of URLs & process/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(urlCached()))
   .map(this::downloadImage)
   .flatMap(this::applyFilters)
   .collect(toList());
```

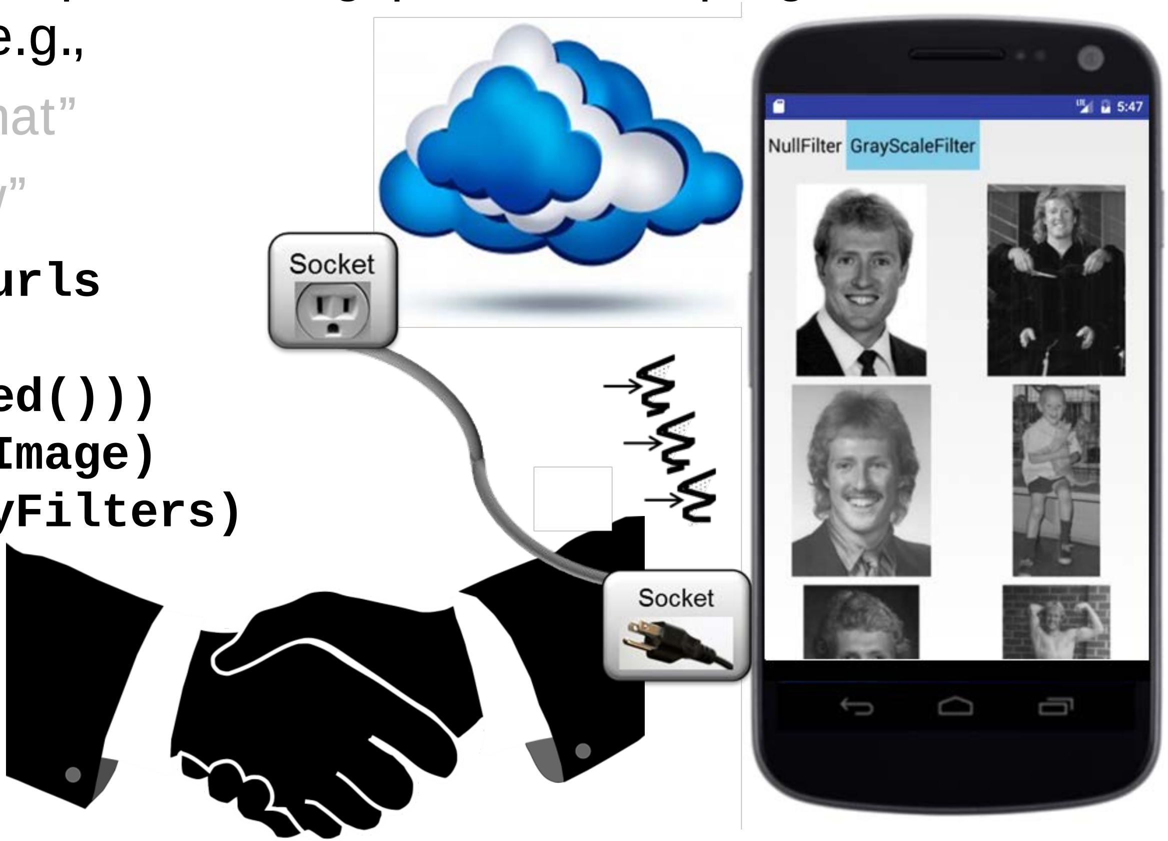
Download images that aren't already cached from a list of URLs & process/store the images in parallel

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

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"

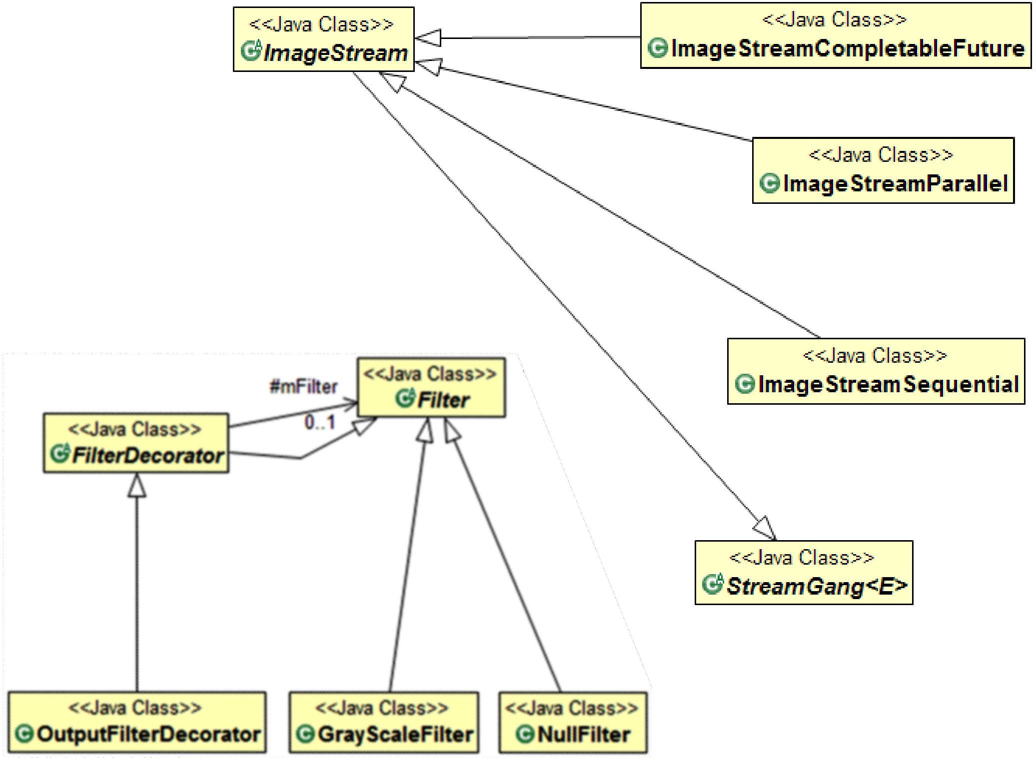
```
List<Image> images = urls
   .parallelStream()
   .filter(not(urlCached()))
   .map(this::downloadImage)
   .flatMap(this::applyFilters)
   .collect(toList());
```



Java 8 functional programming features connect domain intent & computation

Benefits of Combining OO & FP in Java 8

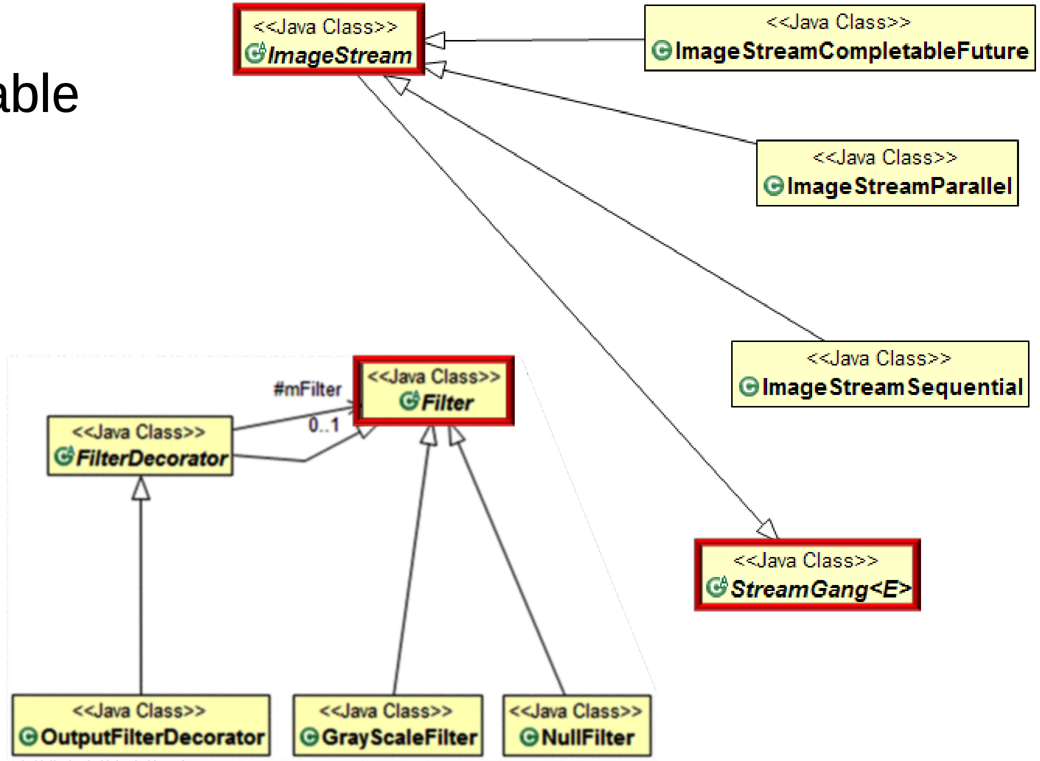
- Likewise, Java 8's object-oriented features help to structure a program's software architecture



See en.wikipedia.org/wiki/Software_architecture

Benefits of Combining OO & FP in Java 8

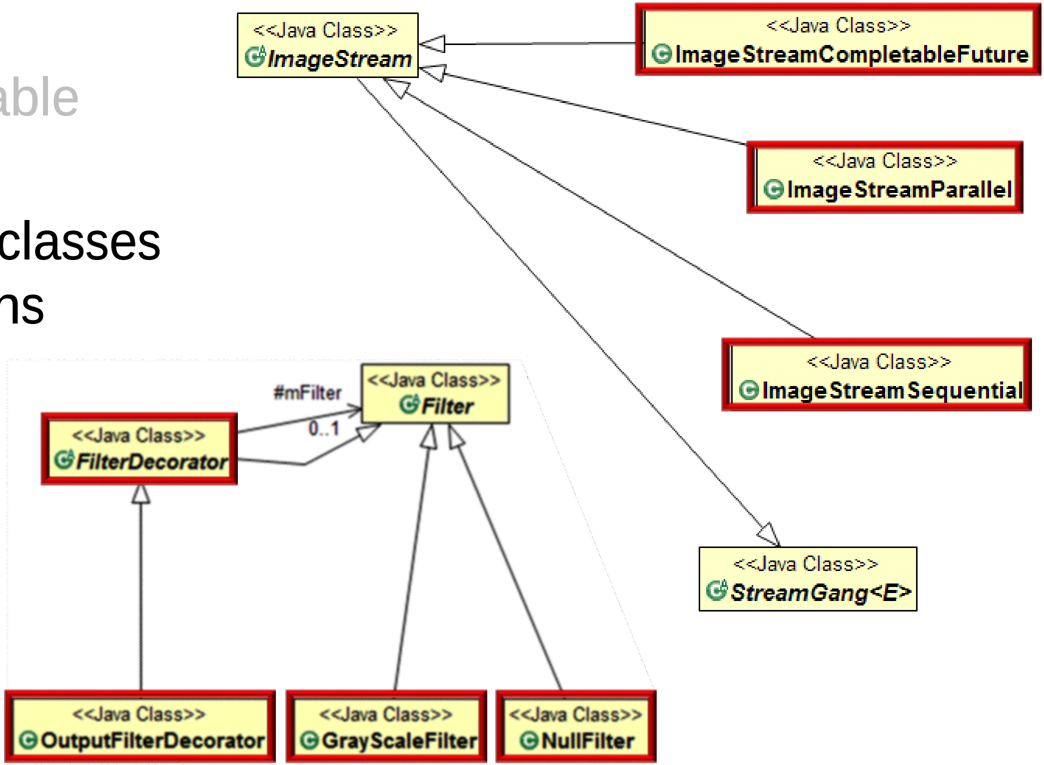
- Likewise, Java 8's object-oriented features help to structure a program's software architecture, e.g.,
  - Common classes provide a reusable foundation for extensibility



See www.dre.vanderbilt.edu/~schmidt/PDF/Commonality_Variability.pdf

Benefits of Combining OO & FP in Java 8

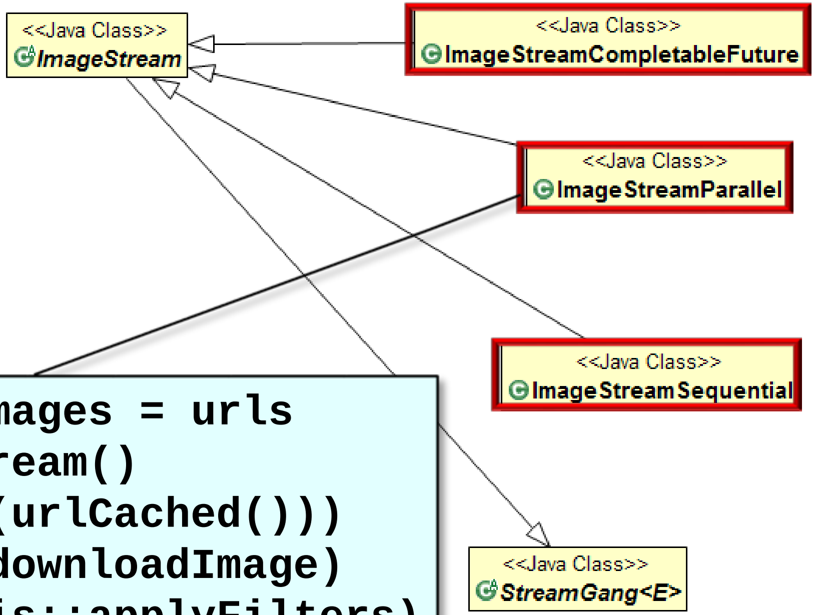
- Likewise, Java 8's object-oriented features help to structure a program's software architecture, e.g.,
  - Common classes provide a reusable foundation for extensibility
  - Subclasses extend the common classes to create various custom solutions



See www.dre.vanderbilt.edu/~schmidt/PDF/Commonality_Variability.pdf

Benefits of Combining OO & FP in Java 8

- Likewise, Java 8's object-oriented features help to structure a program's software architecture, e.g.,
  - Common classes provide a reusable foundation for extensibility
  - Subclasses extend the common classes to create various custom solutions
  - Java 8's FP features are most effective when used to simplify computations within the context of an OO software architecture

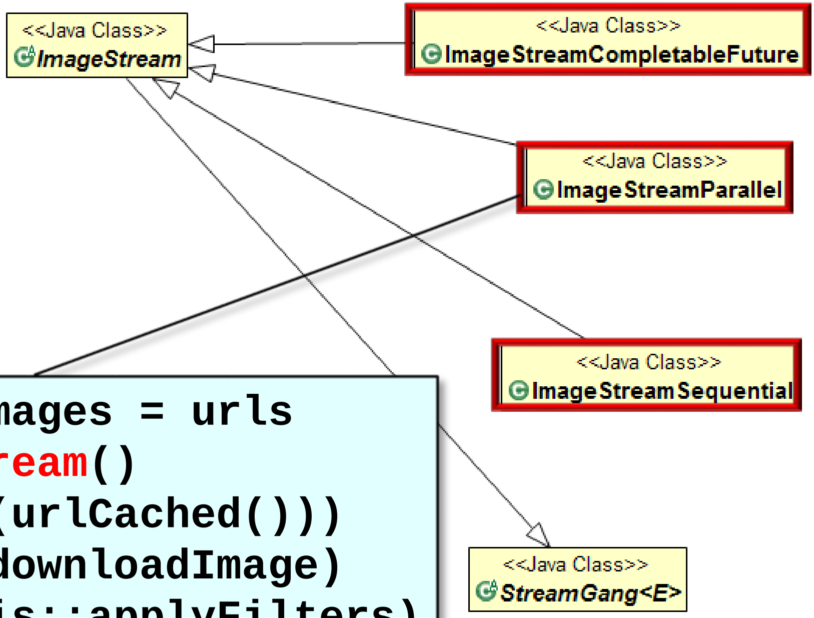



```
List<Image> images = urls
   .parallelStream()
   .filter(not(urlCached()))
   .map(this::downloadImage)
   .flatMap(this::applyFilters)
   .collect(toList());
```

See www.drdobbs.com/jvm/lambda-expressions-in-java-8/240166764

Benefits of Combining OO & FP in Java 8

- Likewise, Java 8's object-oriented features help to structure a program's software architecture, e.g.,
  - Common classes provide a reusable foundation for extensibility
  - Subclasses extend the common classes to create various custom solutions
  - 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




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

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

End of Overview of Java 8 Foundations