When to Use Java Parallel Streams

Douglas C. Schmidt

Learning Objectives in This Lesson

- Know when to use parallel streams.

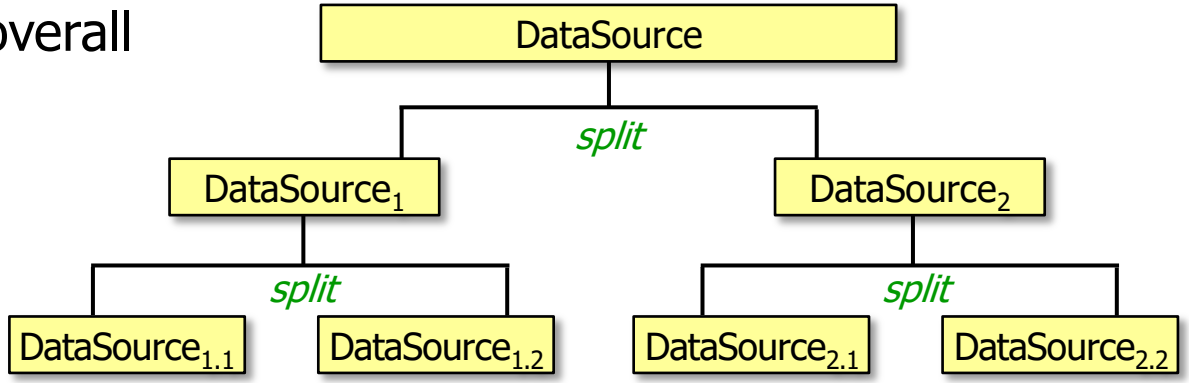
Parallelism Is Not a Panacea

Parallelism Is Not a Panacea

- A parallel program *always* does more work than a nonparallel program.

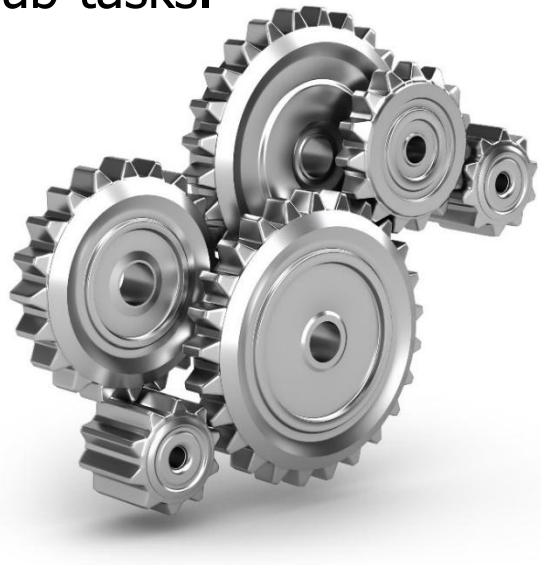
Parallelism Is Not a Panacea

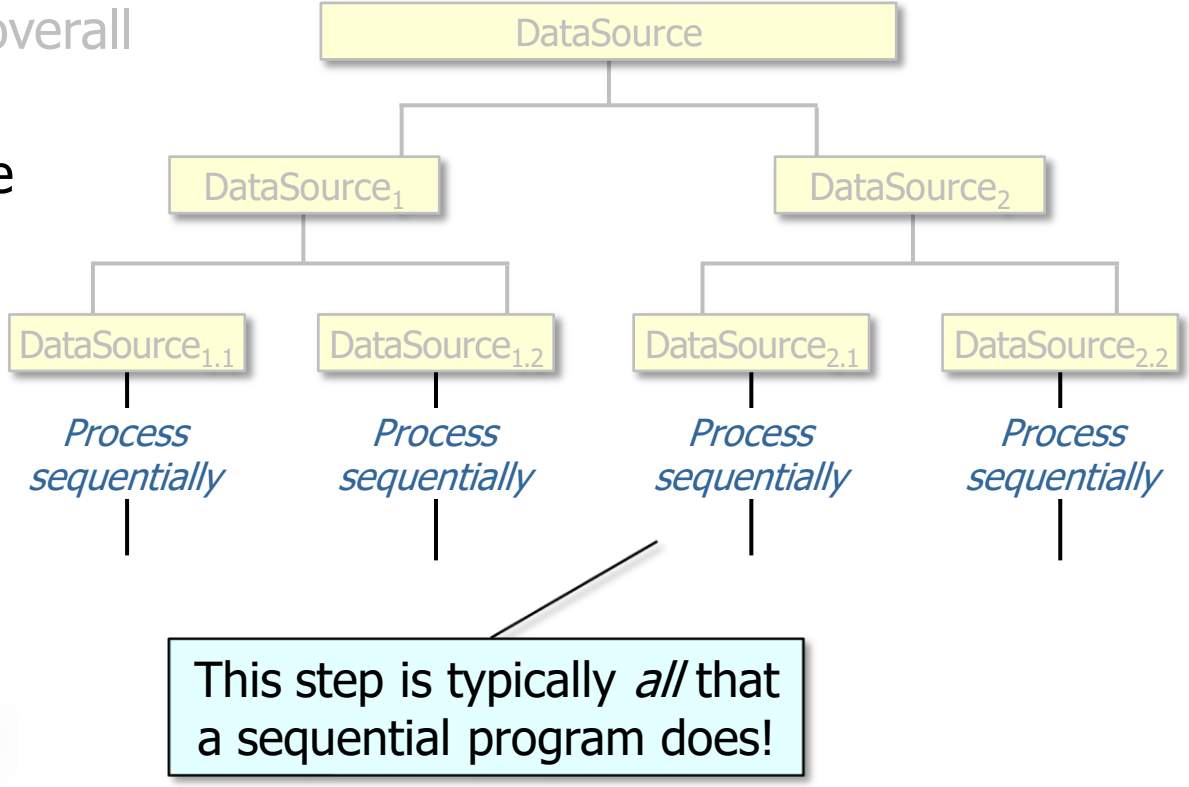
- A parallel program *always* does more work than a nonparallel program, e.g.
  1. It needs to partition the overall task into sub-tasks.



Parallelism Is Not a Panacea

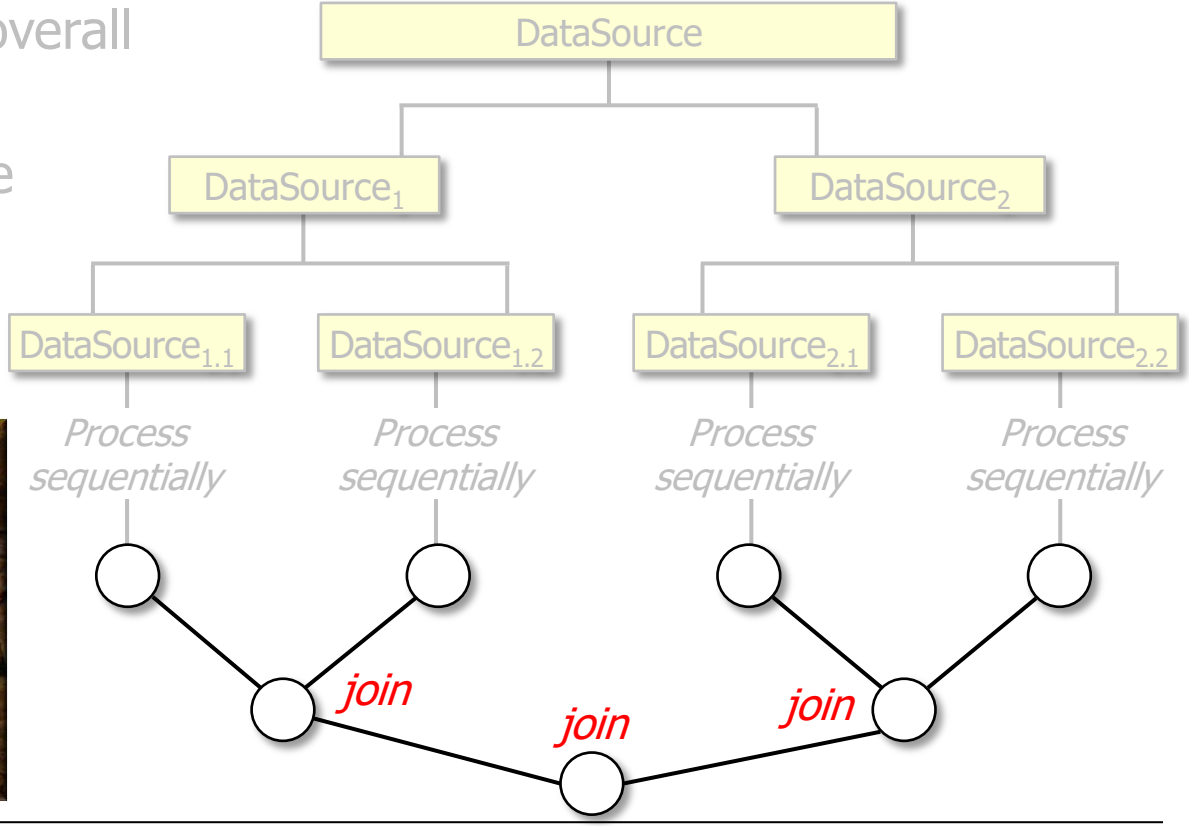
- A parallel program *always* does more work than a nonparallel program, e.g.
  1. It needs to partition the overall task into sub-tasks.
  2. It needs to process all the sub-tasks.





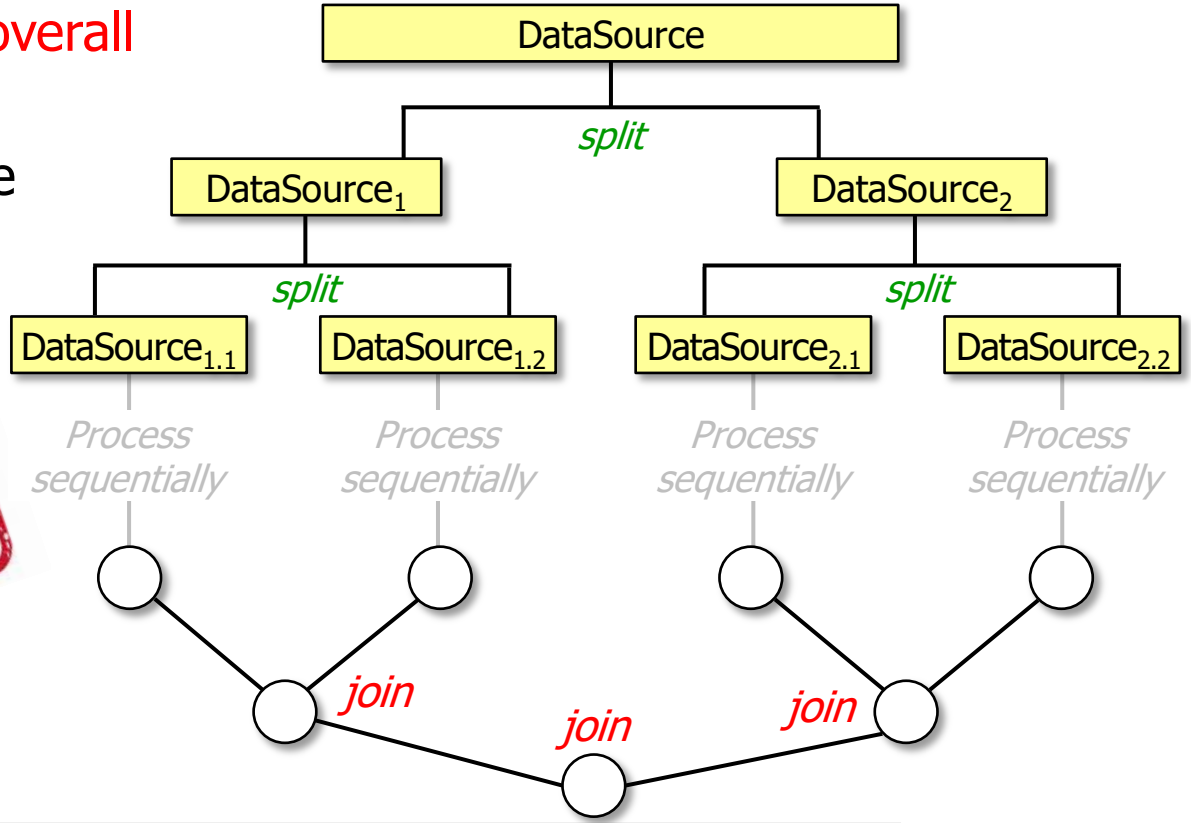
Parallelism Is Not a Panacea

- A parallel program *always* does more work than a nonparallel program, e.g.
  1. It needs to partition the overall task into sub-tasks.
  2. It needs to process all the sub-tasks.
  3. It needs to combine the sub-task results.



Parallelism Is Not a Panacea

- A parallel program *always* does more work than a nonparallel program, e.g.
  1. It needs to partition the overall task into sub-tasks.
  2. It needs to process all the sub-tasks.
  3. It needs to combine the sub-task results.



A sequential program needn't do steps one and three…

Parallelism Is Not a Panacea

- Java parallel streams are thus useful in some (but not all) conditions.

See gee.cs.oswego.edu/dl/html/StreamParallelGuidance.html

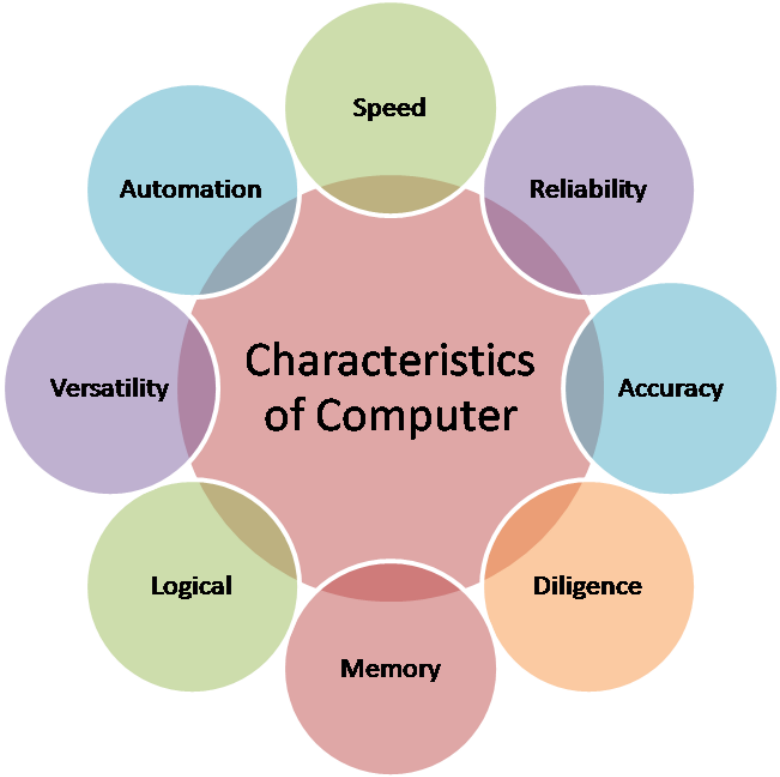
When to Use Java Parallel Streams

When to Use Java Parallel Streams

- Java parallel streams are most useful under certain conditions.

When to Use Java Parallel Streams

- Java parallel streams are most useful under certain conditions, e.g.
  - When behaviors have certain characteristics



When to Use Java Parallel Streams

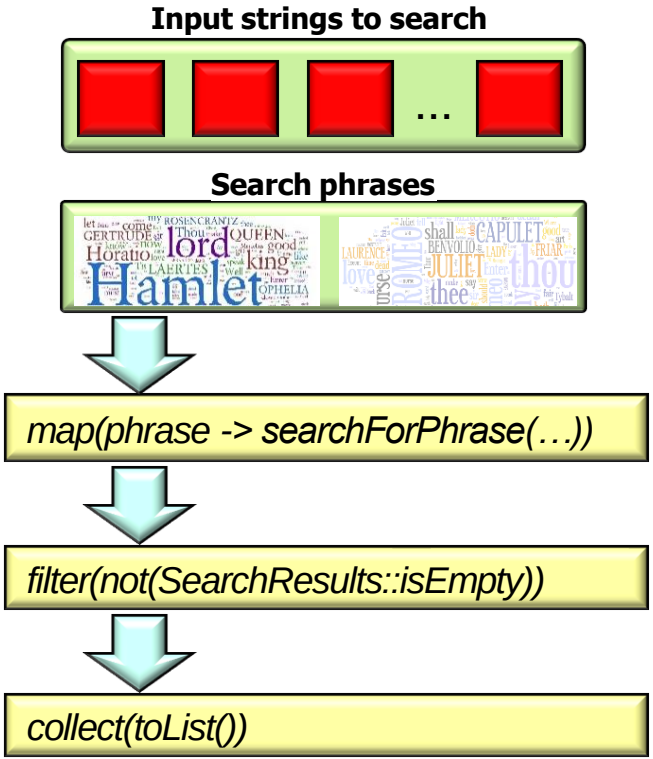
- Java parallel streams are most useful under certain conditions, e.g.
  - When behaviors have certain characteristics
    - Independent

"Embarrassingly parallel" tasks have little/no dependency or need for communication between tasks or for sharing results between them.

See en.wikipedia.org/wiki/Embarrassingly_parallel

When to Use Java Parallel Streams

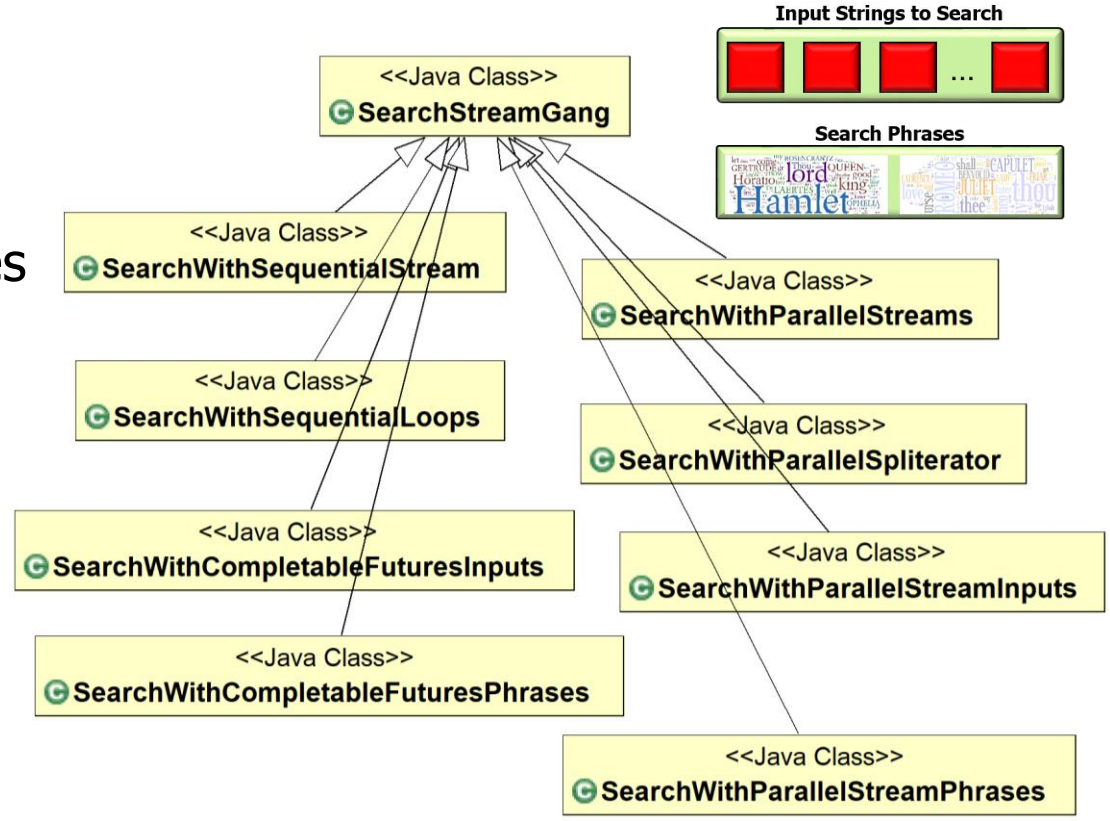
- Java parallel streams are most useful under certain conditions, e.g.
  - When behaviors have certain characteristics
    - Independent
      - e.g., searching for phrases in a list of input strings



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

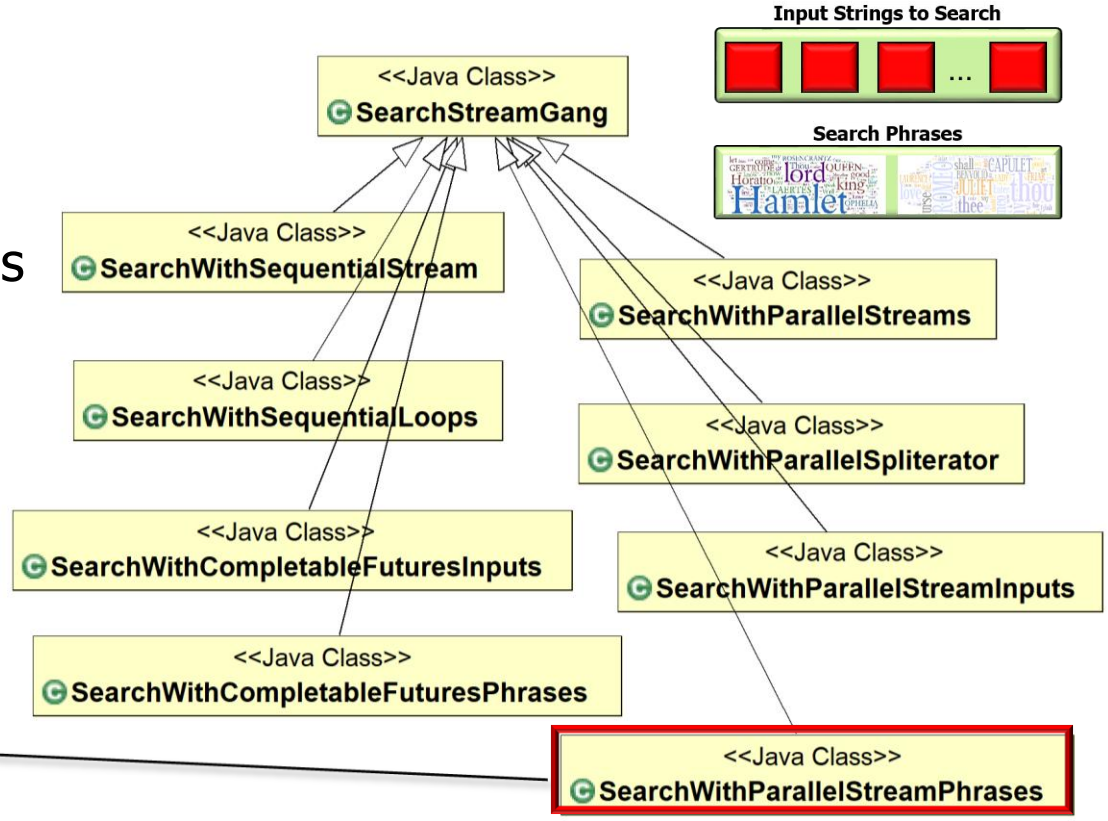
When to Use Java Parallel Streams

- Java parallel streams are most useful under certain conditions, e.g.
  - When behaviors have certain characteristics
    - Independent
      - e.g., searching for phrases in a list of input strings



When to Use Java Parallel Streams

- Java parallel streams are most useful under certain conditions, e.g.
  - When behaviors have certain characteristics
    - Independent
      - e.g., searching for phrases in a list of input strings

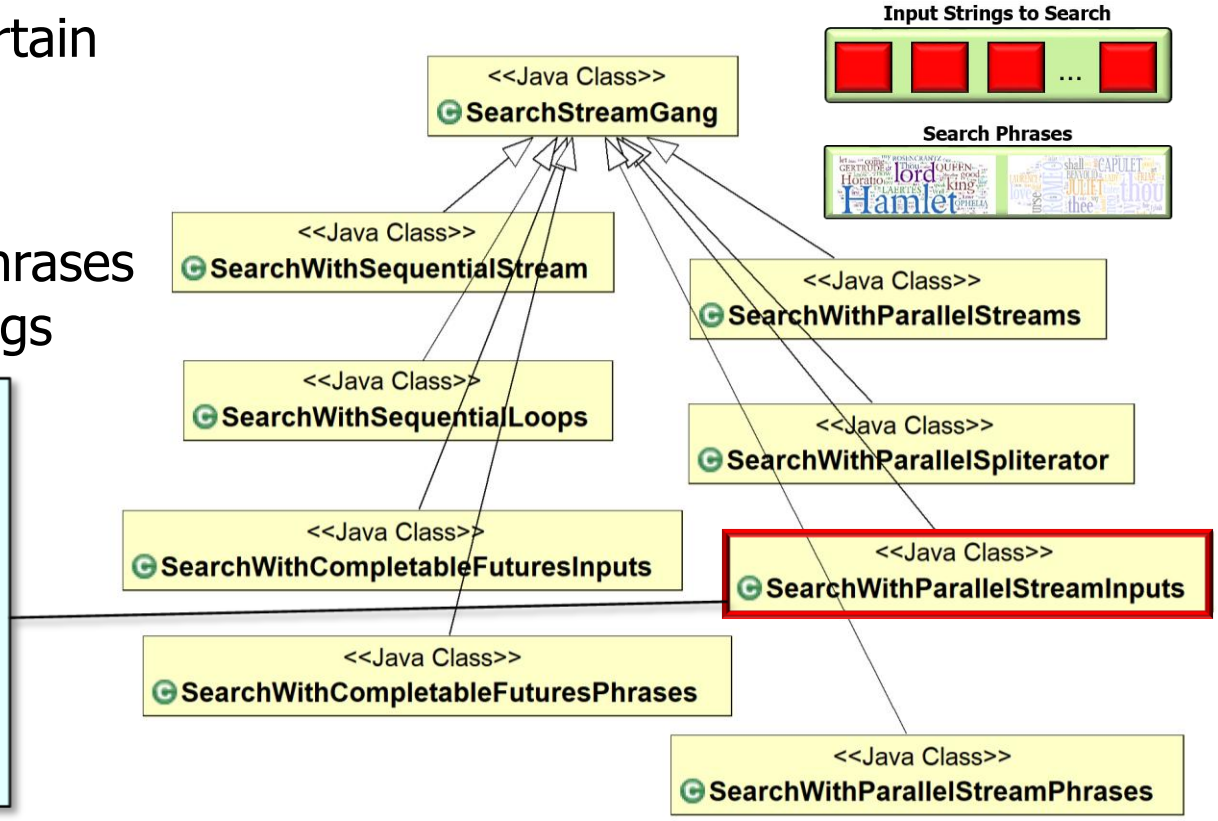


Parallel streams can
- *Search each phrase in parallel*
- *Search each input string in parallel*
- *Search chunks of each input string in parallel*

When to Use Java Parallel Streams

- Java parallel streams are most useful under certain conditions, e.g.
  - When behaviors have certain characteristics
    - Independent
      - e.g., searching for phrases in a list of input strings

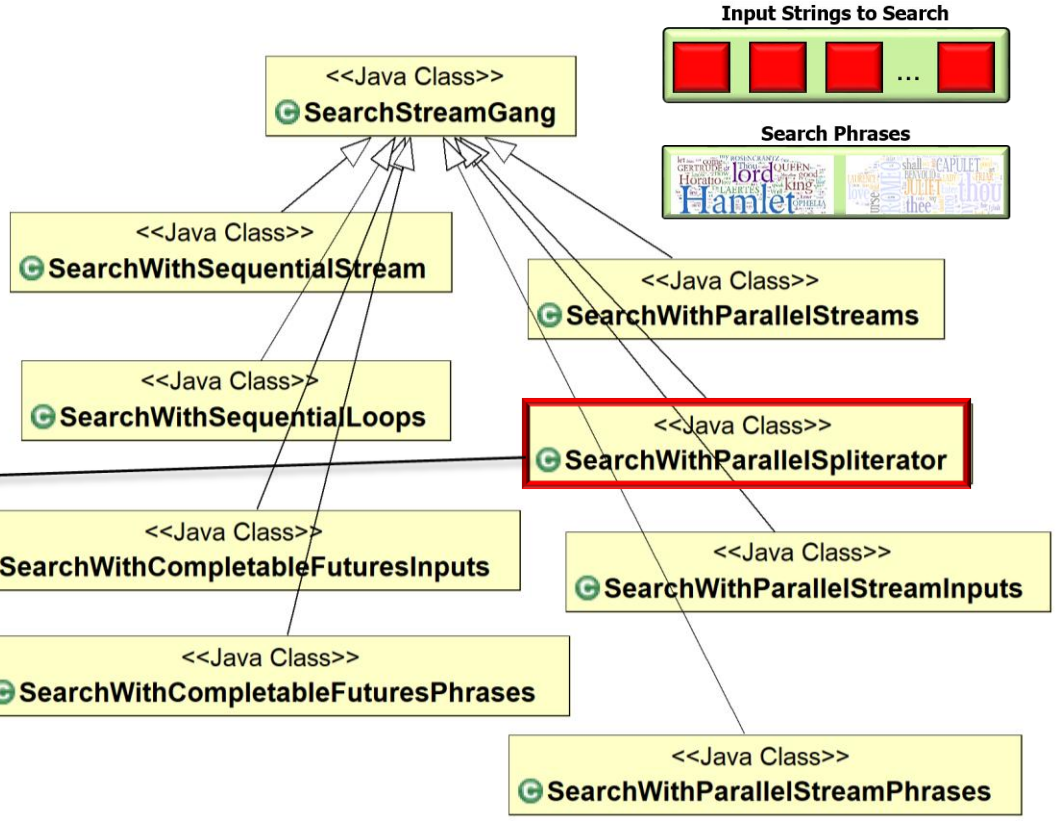
Parallel streams can
- *Search each phrase in parallel*
- *Search each input string in parallel*
- *Search chunks of each input string in parallel*



When to Use Java Parallel Streams

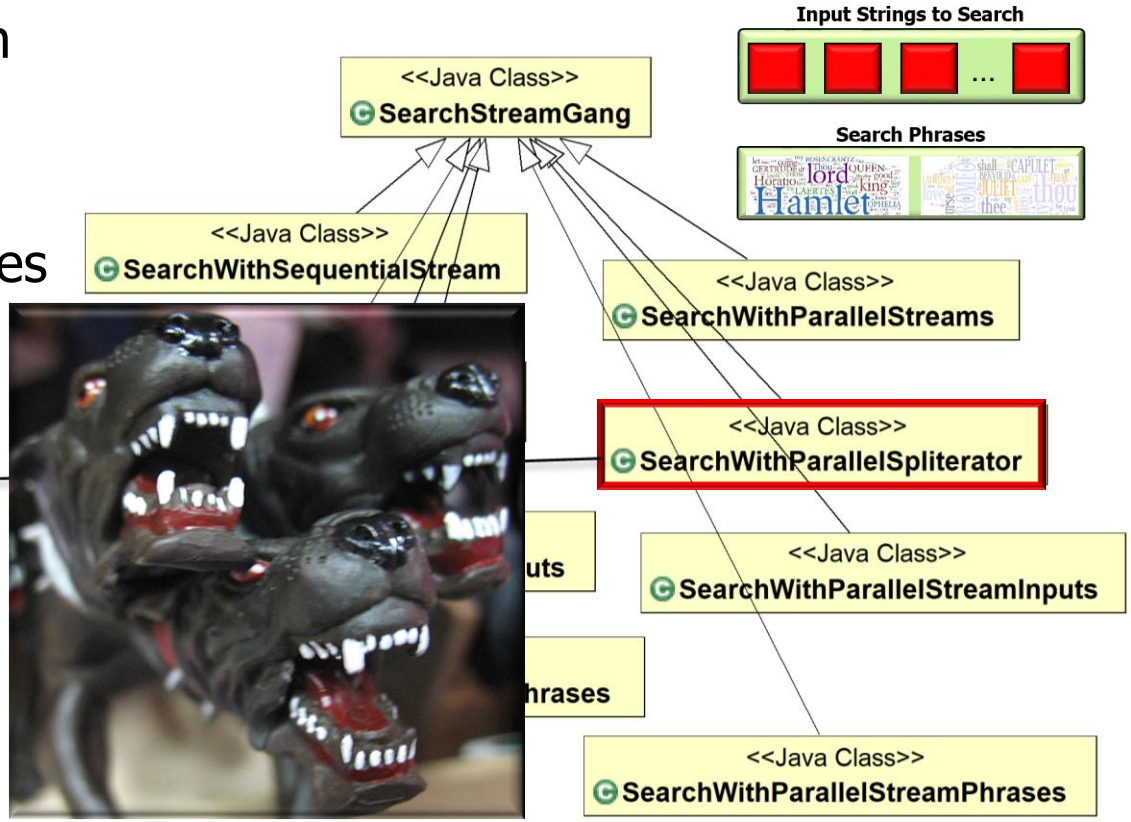
- Java parallel streams are most useful under certain conditions, e.g.
  - When behaviors have certain characteristics
    - Independent
      - e.g., searching for phrases in a list of input strings

Parallel streams can
- *Search each phrase in parallel*
- *Search each input string in parallel*
- *Search chunks of each input string in parallel*



When to Use Java Parallel Streams

- Java parallel streams are most useful under certain conditions, e.g.
  - When behaviors have certain characteristics
    - Independent
      - e.g., searching for phrases in a list of input strings

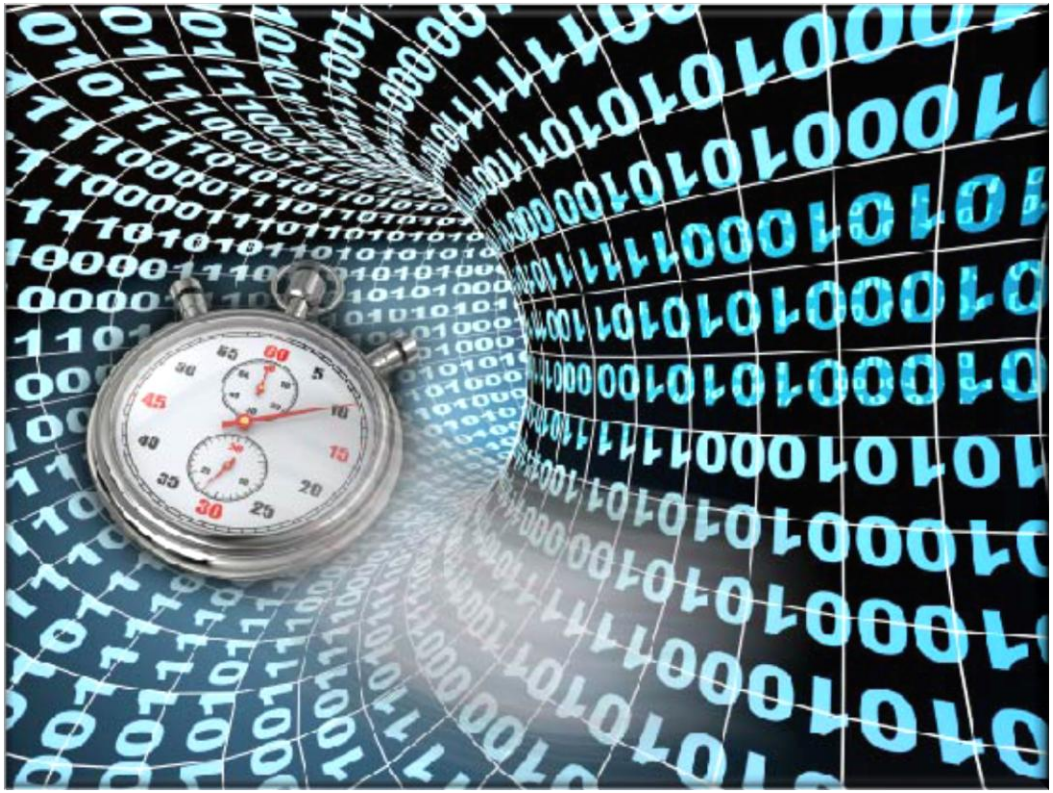


Parallel streams can
- *Search chunks of phrases in parallel*
- *Search chunks of input in parallel*
- *Search chunks of each input string in parallel*

SearchWithParallelSpliterator is the most aggressive parallelism strategy!

When to Use Java Parallel Streams

- Java parallel streams are most useful under certain conditions, e.g.
  - When behaviors have certain characteristics
    - Independent
    - Computationally expensive
      - e.g., when behavior(s) applied to each element take a "long time" to run



See www.ibm.com/developerworks/library/j-java-streams-5-brian-goetz

When to Use Java Parallel Streams

- Java parallel streams are most useful under certain conditions, e.g.
  - When behaviors have certain characteristics
    - Independent
    - Computationally expensive
    - Applied to many elements of data sources

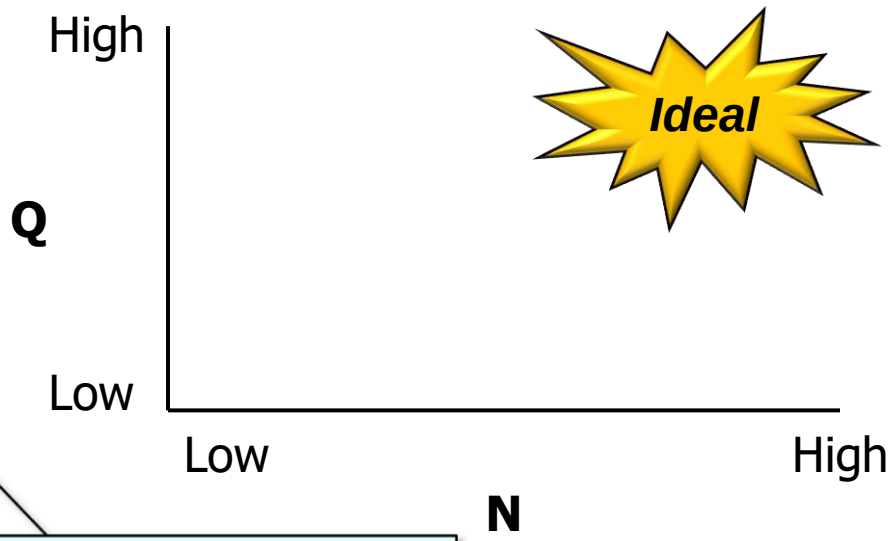
See www.ibm.com/developerworks/library/j-java-streams-5-brian-goetz

When to Use Java Parallel Streams

- Java parallel streams are most useful under certain conditions, e.g.
  - When behaviors have certain characteristics
    - Independent
    - Computationally expensive
    - Applied to many elements of data sources
      - Where these sources can be split efficiently/evenly

When to Use Java Parallel Streams

- Java parallel streams are most useful under certain conditions, e.g.
  - When behaviors have certain characteristics
    - Independent
    - Computationally expensive
    - Applied to many elements of data sources



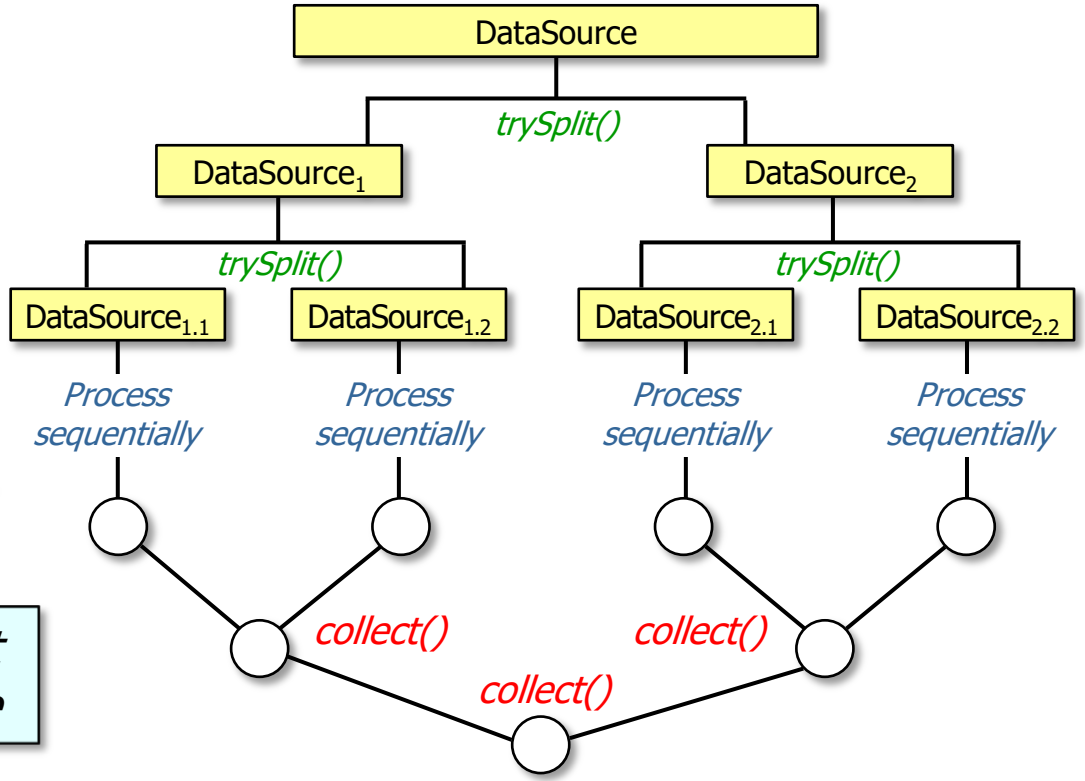
The "NQ" model

- *N is the number of data elements to process per thread.*
- *Q quantifies how CPU-intensive the processing is.*

See on-sw-integration.epischel.de/2016/08/05/parallel-stream-processing-with-java-8-stream-api

When to Use Java Parallel Streams

- Java parallel streams are most useful under certain conditions, e.g.
  - When behaviors have certain characteristics
    - Independent
    - Computationally expensive
    - Applied to many elements of data sources

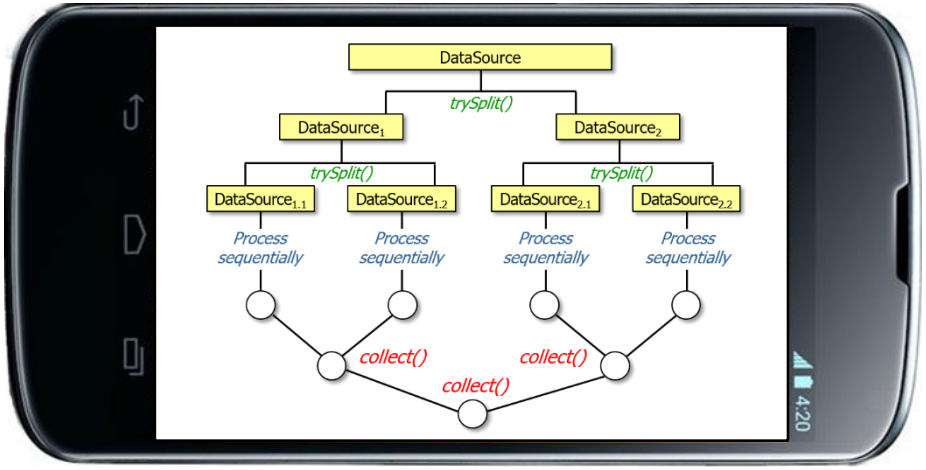


e.g., *searching for phrases that match in works of Shakespeare*

See github.com/douglascraigschmidt/LiveLessons/blob/master/SearchStreamGang

When to Use Java Parallel Streams

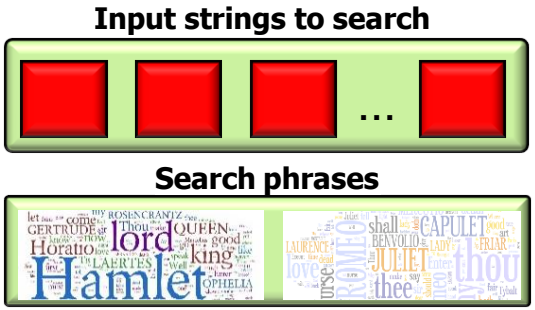
- Java parallel streams are most useful under certain conditions, e.g.
  - When behaviors have certain characteristics
  - If there are multiple cores

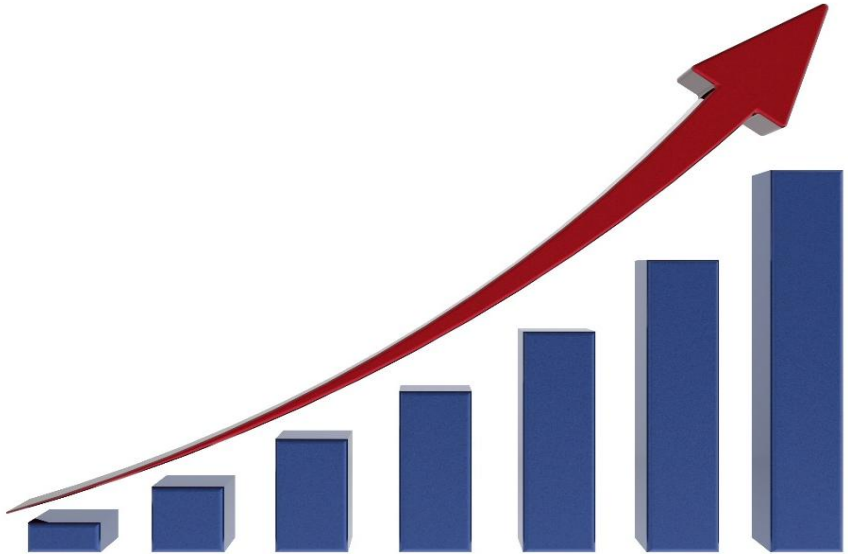


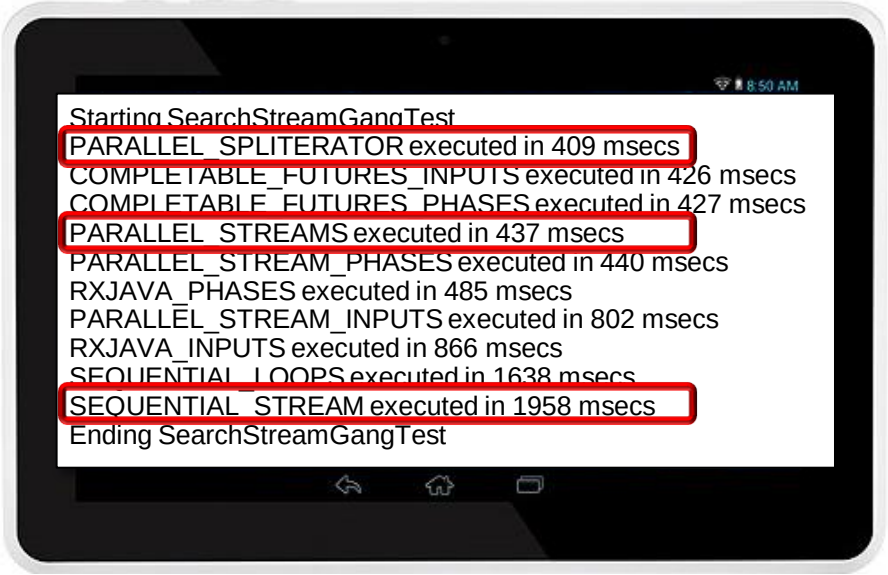
See blog.oio.de/2016/01/22/parallel-stream-processing-in-java-8-performance-of-sequential-vs-parallel-stream-processing

When to Use Java Parallel Streams

- Under the right conditions Java parallel streams can scale up nicely on multicore and many-core processors.







See www.infoq.com/presentations/parallel-java-se-8

The End

When Not to Use Java Parallel Streams

Douglas C. Schmidt

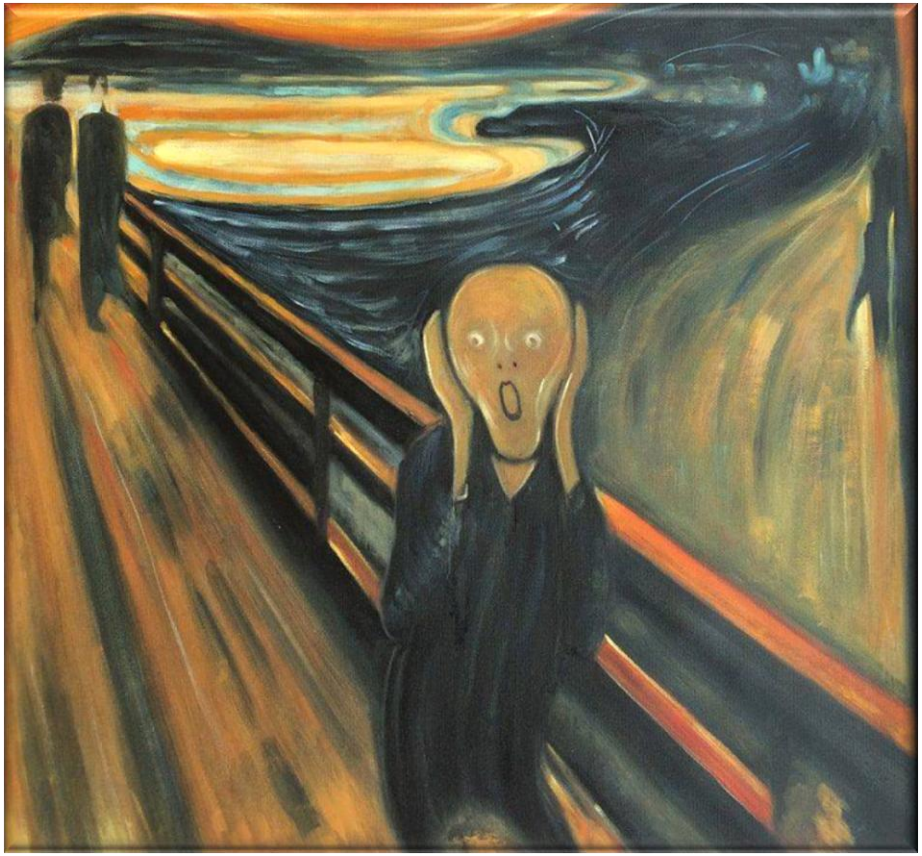
Learning Objectives in This Lesson

- Know when to use parallel streams.
  - And when *not* to use parallel streams.

When Not to Use Java Parallel Streams

When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs.



See www.ibm.com/developerworks/library/j-java-streams-5-brian-goetz

When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly.

```
List<CharSequence> arrayAllWords =
  TestDataFactory.getInput
    (sSHAKESPEARE_WORKS, "\\s+");

List<CharSequence> listAllWords =
  new LinkedList<>(arrayAllWords);


arrayAllWords.parallelStream()
             ...;

listAllWords.parallelStream()
             ...;
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex14

When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly.

Make an ArrayList that contains all words in the works of Shakespeare.

```
List<CharSequence> arrayAllWords =
  TestDataFactory.getInput
    (sSHAKESPEARE_WORKS, "\\s+");

List<CharSequence> listAllWords =
  new LinkedList<>(arrayAllWords);


arrayAllWords.parallelStream()
                  ...;

listAllWords.parallelStream()
                  ...;
```

When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly.

```
List<CharSequence> arrayAllWords =
  TestDataFactory.getInput
    (sSHAKESPEARE_WORKS, "\\s+");

List<CharSequence> listAllWords =
  new LinkedList<>(arrayAllWords);


arrayAllWords.parallelStream()
             ...;

listAllWords.parallelStream()
            ...;
```

Make a LinkedList that contains all words in the works of Shakespeare.

LinkedList doesn't split evenly or efficiently compared with ArrayList.

When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly.

```
Starting spliterator tests for 100000
words....printing results
    599 msecs: ArrayList parallel
    701 msecs: LinkedList parallel

Starting spliterator tests for 883311
words....printing results
   5718 msecs: ArrayList parallel
  31226 msecs: LinkedList parallel
```

```
List<CharSequence> arrayAllWords =
  TestDataFactory.getInput
    (sSHAKESPEARE_WORKS, "\\s+");

List<CharSequence> listAllWords =
  new LinkedList<>(arrayAllWords);


arrayAllWords.parallelStream()
             ...;

listAllWords.parallelStream()
            ...;
```

The ArrayList parallel stream is much faster than the LinkedList parallel stream.

See earlier lesson on "*Java Parallel Stream Internals: Demo'ing Spliterator Performance*"

When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly.

The ArrayList Spliterator runs in O(1) constant time.

```
class ArrayListSpliterator {
  ...
ArrayListSpliterator<E>
  trySplit() {
  int hi = getFence(), lo =
    index, mid = (lo + hi) >>> 1;
  return lo >= mid
       ? null
       : new
       ArrayListSpliterator<E>
        (list, lo, index = mid,
         expectedModCount);
}
...
```

See openjdk/8u40-b25/java/util/ArrayList.java

When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly.

```
class ArrayListSpliterator {
  ...
  ArrayListSpliterator<E>
    trySplit() {
    int hi = getFence(), lo =
      index, mid = (lo + hi) >>> 1;
    return lo >= mid
           ? null
           : new
           ArrayListSpliterator<E>
            (list, lo, index = mid,
             expectedModCount);
  }
  ...
```

Compute the midpoint efficiently.

When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly.

```
class ArrayListSpliterator {
  ...
  ArrayListSpliterator<E>
    trySplit() {
    int hi = getFence(), lo =
      index, mid = (lo + hi) >>> 1;
    return lo >= mid
         ? null
         : new
         ArrayListSpliterator<E>
          (list, lo, index = mid,
           expectedModCount);
}
...
```

Split the array list evenly without copying the data.

When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly.

```
class LLSpliterator {
  ...
  public Spliterator<E> trySplit(){
    ...
    int n = batch + BATCH_UNIT;
    ...
    Object[] a = new Object[n];
    int j = 0;
    do { a[j++] = p.item; }
    while ((p = p.next) != null
           && j < n);
    ...
    return Spliterators
      .spliterator(a, 0, j,
                   Spliterator.ORDERED);
```

The LinkedList Spliterator runs in O(n) linear time.

See openjdk/8-b132/java/util/LinkedList.java

When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly.

Create a fixed-size chunk.

```
class LLSpliterator {
  ...
  public Spliterator<E> trySplit(){
    ...
    int n = batch + BATCH_UNIT;
    ...
    Object[] a = new Object[n];
    int j = 0;
    do { a[j++] = p.item; }
    while ((p = p.next) != null
           && j < n);
    ...
    return Spliterators
      .spliterator(a, 0, j,
                   Spliterator.ORDERED);
```

When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly.

```
class LLSpliterator {
  ...
  public Spliterator<E> trySplit(){
    ...
    int n = batch + BATCH_UNIT;
    ...
    Object[] a = new Object[n];
    int j = 0;
    do { a[j++] = p.item; }
    while ((p = p.next) != null
           && j < n);
    ...
    return Spliterators
      .spliterator(a, 0, j,
                   Spliterator.ORDERED);
```

Copy data into the chunk.

When Not to Use Java Parallel Streams

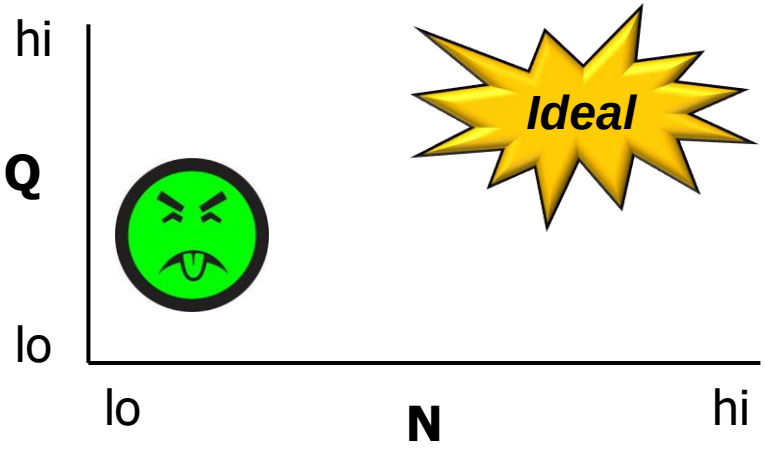
- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly.

```
class LLSpliterator {
  ...
  public Spliterator<E> trySplit(){
    ...
    int n = batch + BATCH_UNIT;
    ...
    Object[] a = new Object[n];
    int j = 0;
    do { a[j++] = p.item; }
    while ((p = p.next) != null
           && j < n);
    ...
    return Spliterators
      .spliterator(a, 0, j,
                   Spliterator.ORDERED);
```

Create a new Spliterator that covers the chunk.

When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly.
  - The startup costs of parallelism overwhelm the amount of data.




```
class ParallelStreamFactorial {
  BigInteger factorial(long n) {
    return LongStream
      .rangeClosed(1, n)
      .parallel() ...
      .reduce(BigInteger.ONE,
              BigInteger::multiply);
  ...

class SequentialStreamFactorial {
  BigInteger factorial(long n) {
    return LongStream
      .rangeClosed(1, n) ...
      .reduce(BigInteger.ONE,
              BigInteger::multiply);
  ...
```

See previous lesson on "*When to Use Parallel Streams*"

When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly.
  - The startup costs of parallelism overwhelm the amount of data.

The overhead of creating a parallel stream is greater than the benefits of parallelism for small values of "n."

```
class ParallelStreamFactorial {
  BigInteger factorial(long n) {
    return LongStream
      .rangeClosed(1, n)
      .parallel() ...
      .reduce(BigInteger.ONE,
              BigInteger::multiply);
  ...

class SequentialStreamFactorial {
  BigInteger factorial(long n) {
    return LongStream
      .rangeClosed(1, n) ...
      .reduce(BigInteger.ONE,
              BigInteger::multiply);
  ...
```

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

When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly.
  - The startup costs of parallelism overwhelm the amount of data.

If "n" is small, then this parallel solution will be inefficient.

```
class ParallelStreamFactorial {
  BigInteger factorial(long n) {
    return LongStream
      .rangeClosed(1, n)
      .parallel() ...
      .reduce(BigInteger.ONE,
              BigInteger::multiply);
  ...

class SequentialStreamFactorial {
  BigInteger factorial(long n) {
    return LongStream
      .rangeClosed(1, n) ...
      .reduce(BigInteger.ONE,
              BigInteger::multiply);
  ...
```

When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly.
  - The startup costs of parallelism overwhelm the amount of data.

If "n" is small, then this sequential solution will be more efficient.

```
class ParallelStreamFactorial {
  BigInteger factorial(long n) {
    return LongStream
      .rangeClosed(1, n)
      .parallel() ...
      .reduce(BigInteger.ONE,
              BigInteger::multiply);
  ...

class SequentialStreamFactorial {
  BigInteger factorial(long n) {
    return LongStream
      .rangeClosed(1, n) ...
      .reduce(BigInteger.ONE,
              BigInteger::multiply);
  ...
```

When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly.
  - The startup costs of parallelism overwhelm the amount of data.
  - Combining partial results is costly.

```
List<CharSequence> allWords =
  new LinkedList<>
    (TestDataFactory.getInput
       (sSHAKESPEARE_DATA_FILE,
        "\\s+"));
...

Set<CharSequence> uniqueWords =
  allWords
    .parallelStream()
    ...
    .collect(toCollection
                  (TreeSet::new));
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex14

When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly.
  - The startup costs of parallelism overwhelm the amount of data.
  - Combining partial results is costly.

```
List<CharSequence> allWords =
  new LinkedList<>
    (TestDataFactory.getInput
       (sSHAKESPEARE_DATA_FILE,
        "\\s+"));
...

Set<CharSequence> uniqueWords =
  allWords
    .parallelStream()
    ...
    .collect(toCollection
                  (TreeSet::new));
```

A linked list of all words in the complete works of Shakespeare.

When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly.
  - The startup costs of parallelism overwhelm the amount of data.
  - Combining partial results is costly.

```
List<CharSequence> allWords =
  new LinkedList<>
    (TestDataFactory.getInput
       (sSHAKESPEARE_DATA_FILE,
        "\\s+"));
...

Set<CharSequence> uniqueWords =
  allWords
    .parallelStream()
    ...
    .collect(toCollection
                  (TreeSet::new));
```

Performance will be poor due to the overhead of combining partial results for a set in a parallel stream.

When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly.
  - The startup costs of parallelism overwhelm the amount of data.
  - Combining partial results is costly.

```
List<CharSequence> allWords =
  new LinkedList<>
    (TestDataFactory.getInput
       (sSHAKESPEARE_DATA_FILE,
        "\\s+"));
...

Set<CharSequence> uniqueWords =
  allWords
    .parallelStream()
    ...
    .collect(toCollection
                  (TreeSet::new));
```

Combining costs can be alleviated if the amount of work performed per element is rather large (i.e., the "NQ model").

See www.ibm.com/developerworks/library/j-java-streams-5-brian-goetz

When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly.
  - The startup costs of parallelism overwhelm the amount of data.
  - Combining partial results is costly.

```
List<CharSequence> allWords =
  new LinkedList<>
    (TestDataFactory.getInput
       (sSHAKESPEARE_DATA_FILE,
        "\\s+"));
...

Set<CharSequence> uniqueWords =
  allWords
    .parallelStream()
    ...
    .collect(toSet()));
```

A concurrent collector can also be used to optimize the reduction phase.

See Java8/ex14/src/main/java/utils/ConcurrentHashSetCollector.java

When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly.
  - The startup costs of parallelism overwhelm the amount of data.
  - Combining partial results is costly.

```
Starting collector tests for 100000 words..printing results
   219 msecs: parallel timeStreamCollectToConcurrentSet()
   364 msecs: parallel timeStreamCollectToSet()
   657 msecs: sequential timeStreamCollectToSet()
   804 msecs: sequential timeStreamCollectToConcurrentSet()
Starting collector tests for 883311 words..printing results
  1782 msecs: parallel timeStreamCollectToConcurrentSet()
  3010 msecs: parallel timeStreamCollectToSet()
  6169 msecs: sequential timeStreamCollectToSet()
  7652 msecs: sequential timeStreamCollectToConcurrentSet()
```

```
List<CharSequence> allWords =
  new LinkedList<>
    (TestDataFactory.getInput
       (sSHAKESPEARE_DATA_FILE,
        "\\s+"));
...

Set<CharSequence> uniqueWords =
  allWords
    .parallelStream()
    ...
    .collect(toSet()));
```

Concurrent collector scales much better than non-concurrent collector

See previous earlier lesson on "*Java Parallel Stream Internals: Demo'ing Collector Performance*"

When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly.
  - The startup costs of parallelism overwhelm the amount of data.
  - Combining partial results is costly.
  - Some streams operations don't sufficiently exploit parallelism.

```
List<Double> result = Stream
   .iterate(2, i -> i + 1)
   .parallel()
   .filter(this::isEven)
   .limit(n)
   .map(this::findSQRT)
   .collect(toList());
```

```
List<Double> result = LongStream
   .range(2, (n * 2) + 1)
   .parallel()
   .filter(this::isEven)
   .mapToObj(this::findSQRT)
   .collect(toList());
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex15

When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly.
  - The startup costs of parallelism overwhelm the amount of data.
  - Combining partial results is costly.
  - Some streams operations don't sufficiently exploit parallelism.

Create a list containing square root of the first "n" even numbers.

```
List<Double> result = Stream
   .iterate(2, i -> i + 1)
   .parallel()
   .filter(this::isEven)
   .limit(n)
   .map(this::findSQRT)
   .collect(toList());
```

```
List<Double> result = LongStream
   .range(2, (n * 2) + 1)
   .parallel()
   .filter(this::isEven)
   .mapToObj(this::findSQRT)
   .collect(toList());
```

When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly.
  - The startup costs of parallelism overwhelm the amount of data.
  - Combining partial results is costly.
  - Some streams operations don't sufficiently exploit parallelism.

```
List<Double> result = Stream
   .iterate(2, i -> i + 1)
   .parallel()
   .filter(this::isEven)
   .limit(n)
   .map(this::findSQRT)
   .collect(toList());

List<Double> result = LongStream
   .range(2, (n * 2) + 1)
   .parallel()
   .filter(this::isEven)
   .mapToObj(this::findSQRT)
   .collect(toList());
```

Stream.iterate() and limit() split and parallelize poorly since iterate creates an ordered stream . . .

See www.java2novice.com/java-8/streams/limit-method-example

When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly.
  - The startup costs of parallelism overwhelm the amount of data.
  - Combining partial results is costly.
  - Some streams operations don't sufficiently exploit parallelism.

Create a list containing square root of the first "n" even numbers.

```
List<Double> result = Stream
   .iterate(2, i -> i + 1)
   .parallel()
   .filter(this::isEven)
   .limit(n)
   .map(this::findSQRT)
   .collect(toList());
```

```
List<Double> result = LongStream
   .range(2, (n * 2) + 1)
   .parallel()
   .filter(this::isEven)
   .mapToObj(this::findSQRT)
   .collect(toList());
```

When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly.
  - The startup costs of parallelism overwhelm the amount of data.
  - Combining partial results is costly.
  - Some streams operations don't sufficiently exploit parallelism.

```
List<Double> result = Stream
   .iterate(2, i -> i + 1)
   .parallel()
   .filter(this::isEven)
   .limit(n)
   .map(this::findSQRT)
   .collect(toList());
```

```
List<Double> result = LongStream
   .range(2, (n * 2) + 1)
   .parallel()
   .filter(this::isEven)
   .mapToObj(this::findSQRT)
   .collect(toList());
```

LongStream.range() splits nicely and thus runs efficiently in parallel.

When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly.
  - The startup costs of parallelism overwhelm the amount of data.
  - Combining partial results is costly.
  - Some streams operations don't sufficiently exploit parallelism.
  - There aren't many/any cores.

Older computing devices just have a single core, which limits available parallelism.

When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly.
  - The startup costs of parallelism overwhelm the amount of data.
  - Combining partial results is costly.
  - Some streams operations don't sufficiently exploit parallelism.
  - There aren't many/any cores.
  - No built-in means to shut down processing of a parallel stream.

When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly.
  - The startup costs of parallelism overwhelm the amount of data.
  - Combining partial results is costly.
  - Some streams operations don't sufficiently exploit parallelism.
  - There aren't many/any cores.
  - No built-in means to shut down processing of a parallel stream.

```
private static volatile
  boolean mCancelled;
```

Define a static volatile flag.

```
Image downloadImage(Cache.Item
                    item) {
  if (mCancelled)
    throw new
      CancellationException
        ("Canceling crawl.");
  ...
```

When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly.
  - The startup costs of parallelism overwhelm the amount of data.
  - Combining partial results is costly.
  - Some streams operations don't sufficiently exploit parallelism.
  - There aren't many/any cores.
  - No built-in means to shut down processing of a parallel stream.

```
private static volatile
  boolean mCancelled;


Image downloadImage(Cache.Item
                      item) {
  if (mCancelled)
    throw new
      CancellationException
        ("Canceling crawl.");
  ...
```

Before downloading the next image, check for cancellation and throw an exception if cancelled.

The End

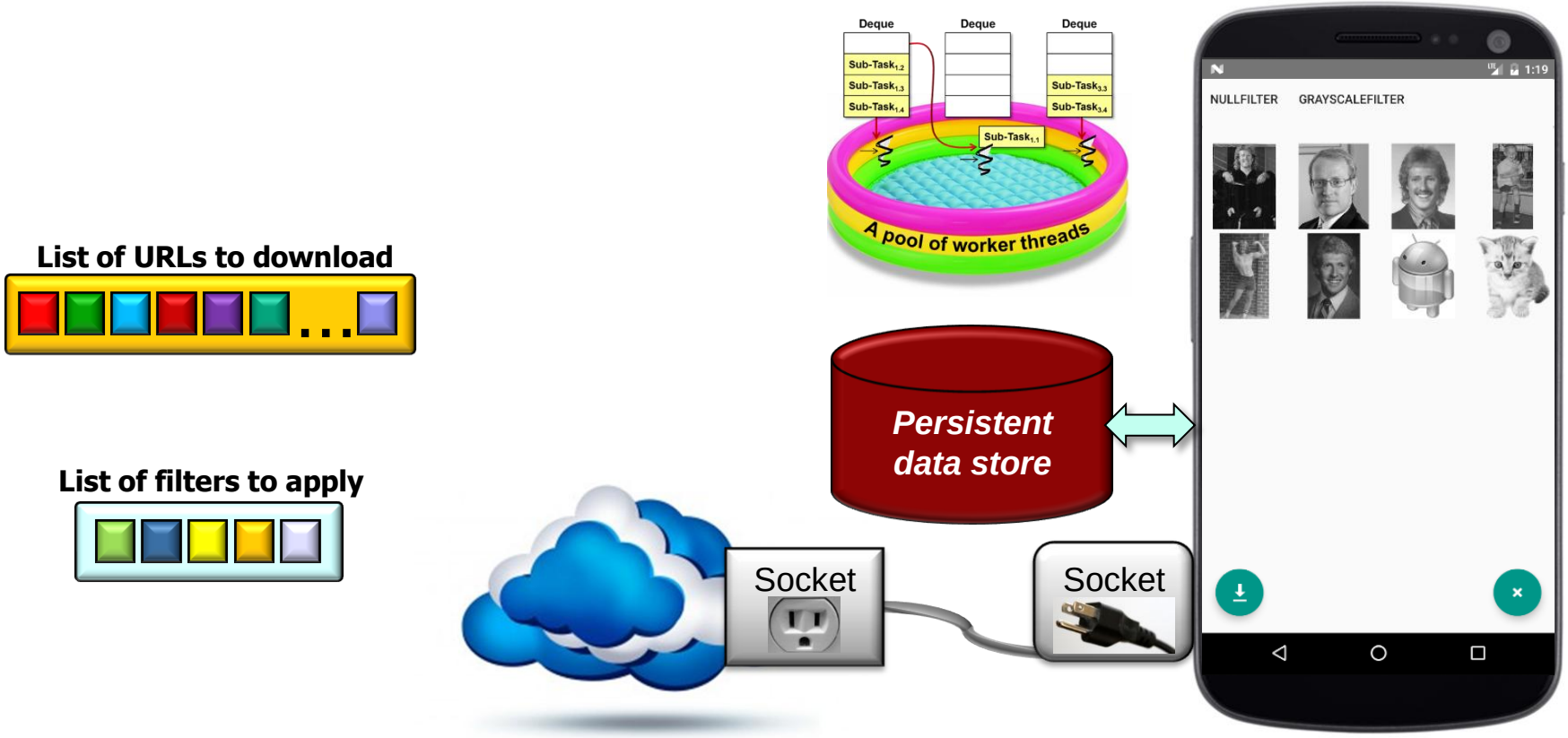
Java Parallel ImageStreamGang Example

Introduction

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

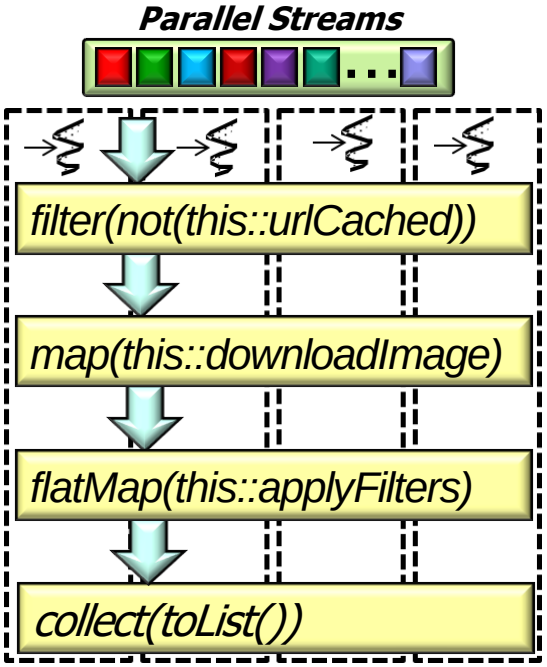
- Recognize the structure and functionality of the ImageStreamGang app.

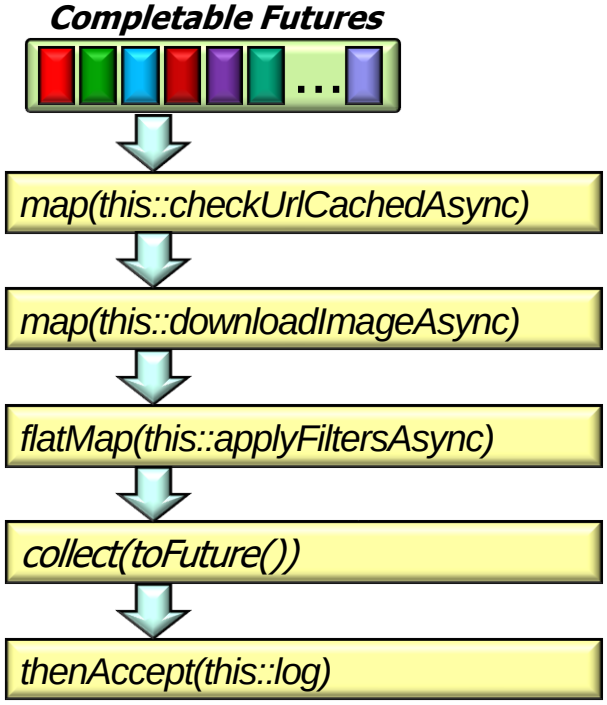


See github.com/douglascraigschmidt/LiveLessons/tree/master/ImageStreamGang

Learning Objectives in This Part of the Lesson

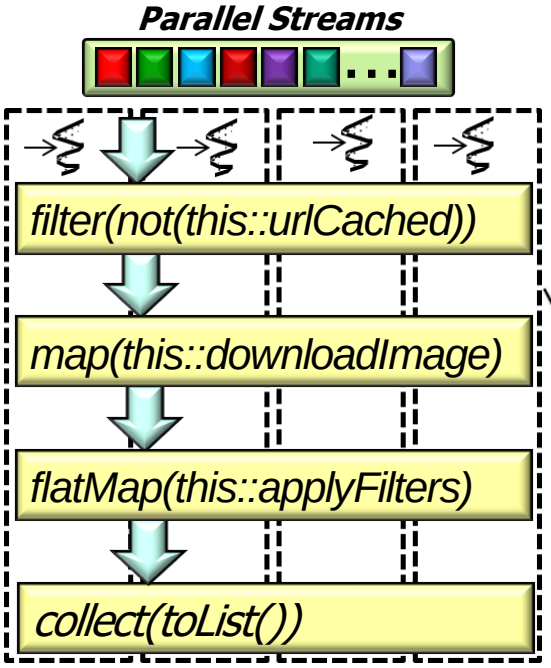
- Recognize the structure and functionality of the ImageStreamGang app.
  - It applies several Java parallelism frameworks.



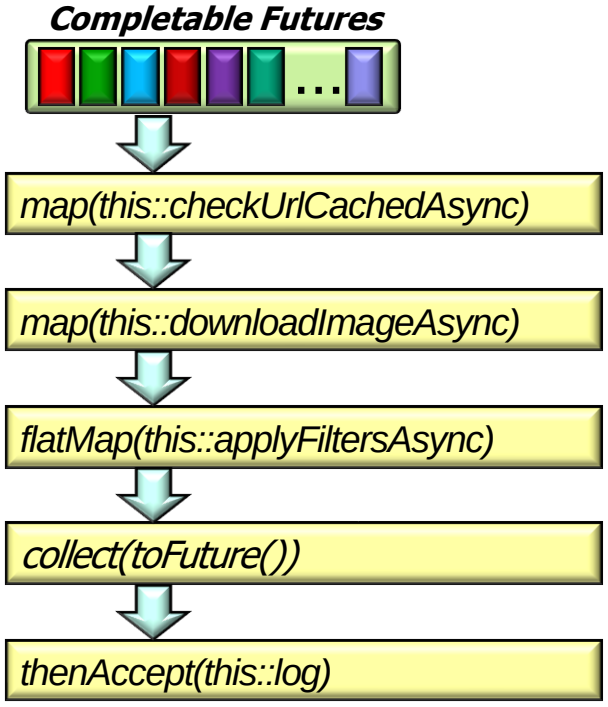


Learning Objectives in This Part of the Lesson

- Recognize the structure and functionality of the ImageStreamGang app.
  - It applies several Java parallelism frameworks.



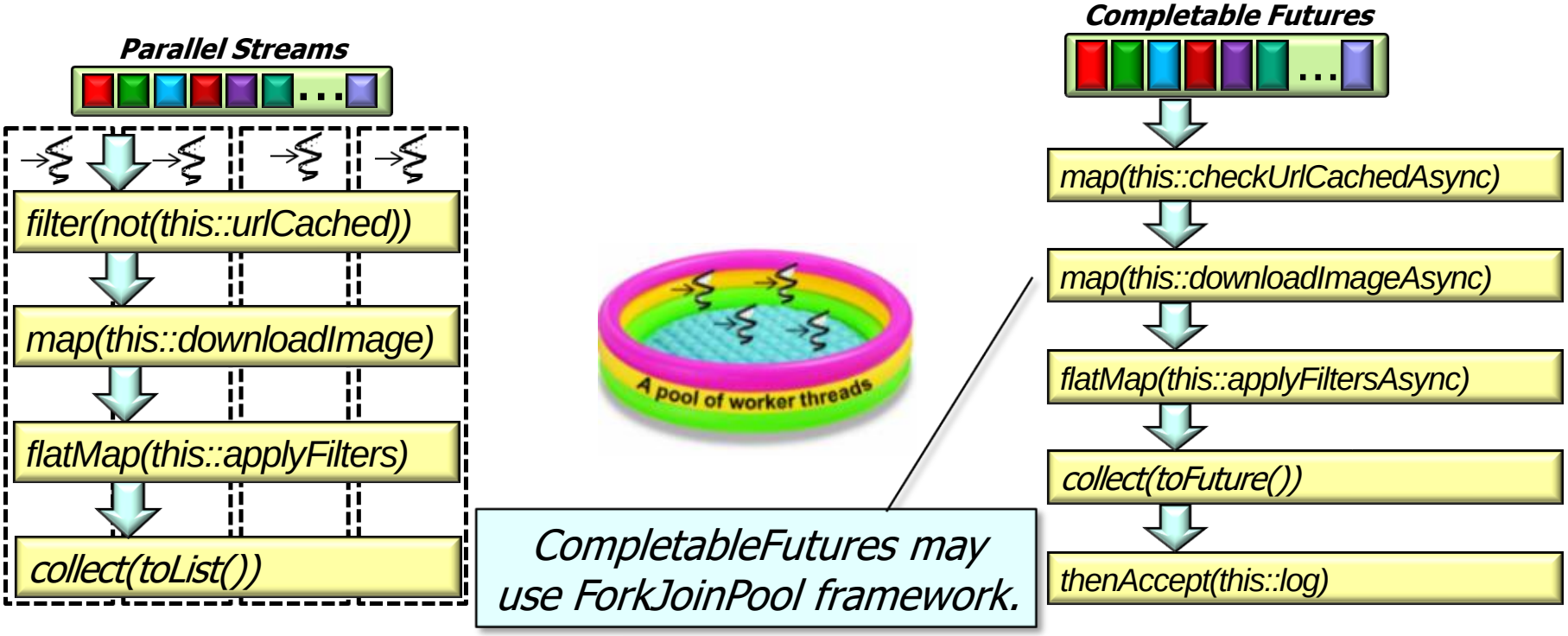
Parallel streams must use ForkJoinPool framework.



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

Learning Objectives in This Part of the Lesson

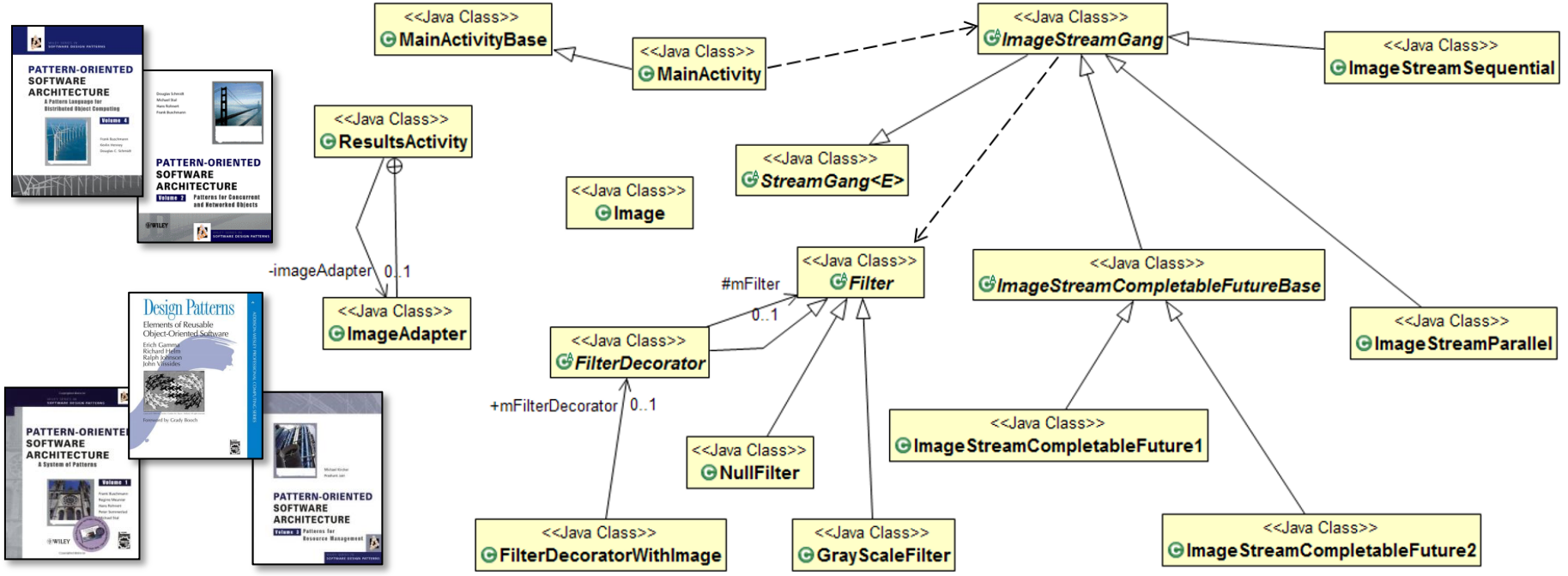
- Recognize the structure and functionality of the ImageStreamGang app.
  - It applies several Java parallelism frameworks.



See www.nurkiewicz.com/2013/05/java-8-definitive-guide-to.html

Learning Objectives in This Part of the Lesson

- Recognize the structure and functionality of the ImageStreamGang app.
  - It applies several Java parallelism frameworks.
  - The focus is on integrating object-oriented and functional programming paradigms.

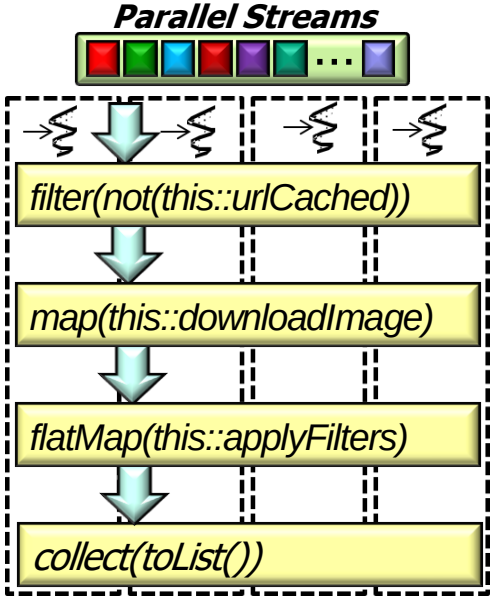


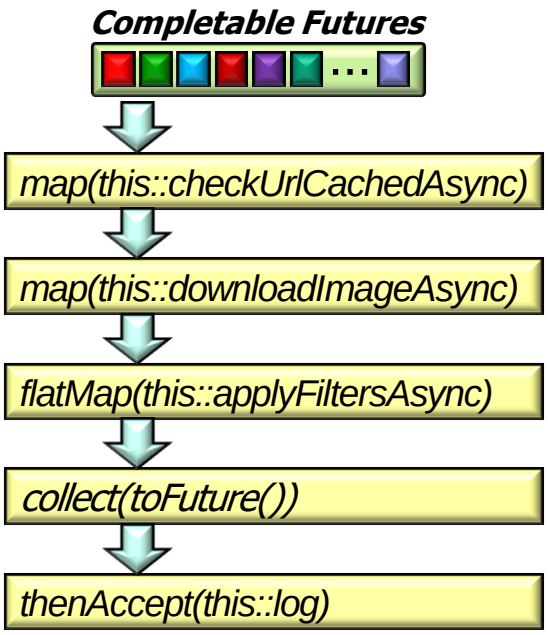
Patterns are used to emphasize key roles and responsibilities in the app's design.

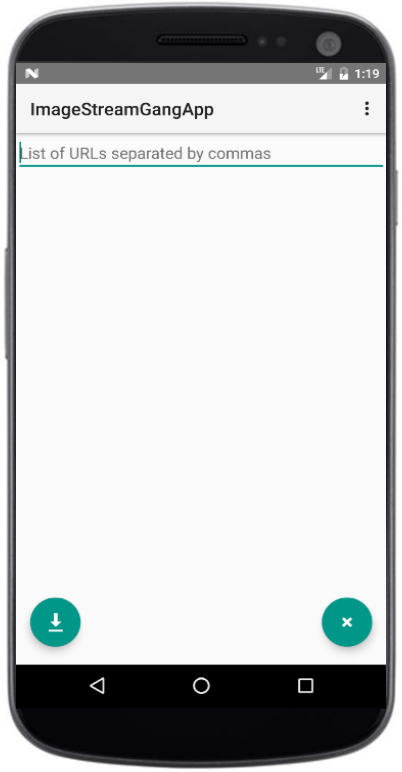
Overview of the Pattern-Oriented ImageStreamGang App

Overview of the Pattern-Oriented ImageStreamGang App

- This app combines streams and CompletableFutures with the StreamGang framework to download, transform, store, and display images.



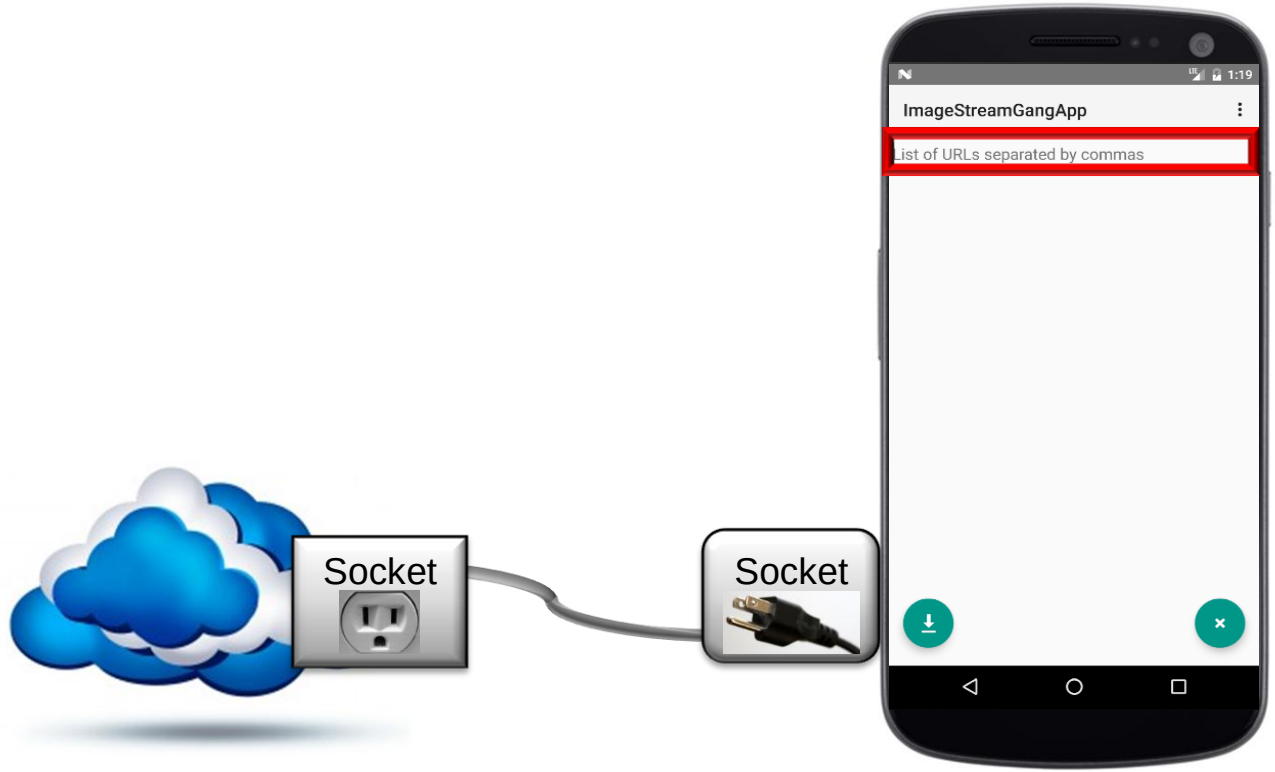




See github.com/douglascraigschmidt/LiveLessons/tree/master/ImageStreamGang

Overview of the Pattern-Oriented ImageStreamGang App

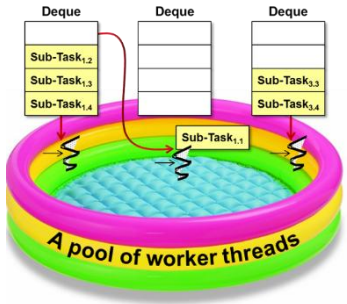
- This app combines streams and CompletableFutures with the StreamGang framework to download, transform, store, and display images, for example.

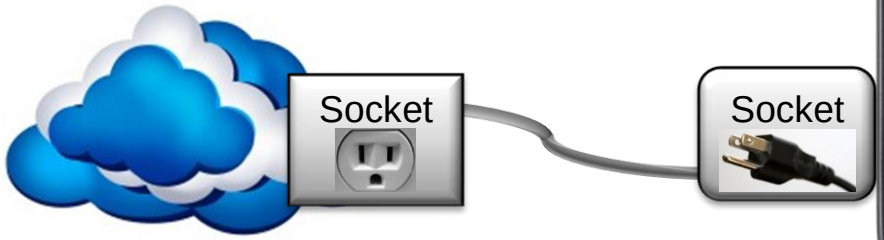


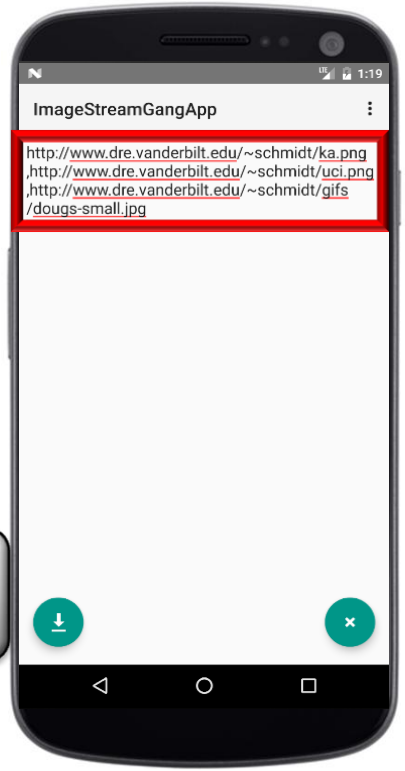
Prompt user for list of URLs to download.

Overview of the Pattern-Oriented ImageStreamGang App

- This app combines streams and CompletableFutures with the StreamGang framework to download, transform, store, and display images, for example.



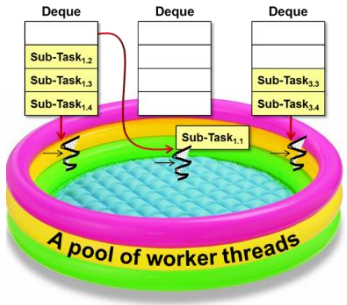


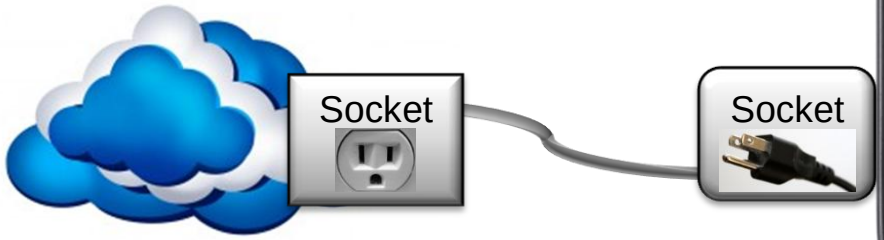


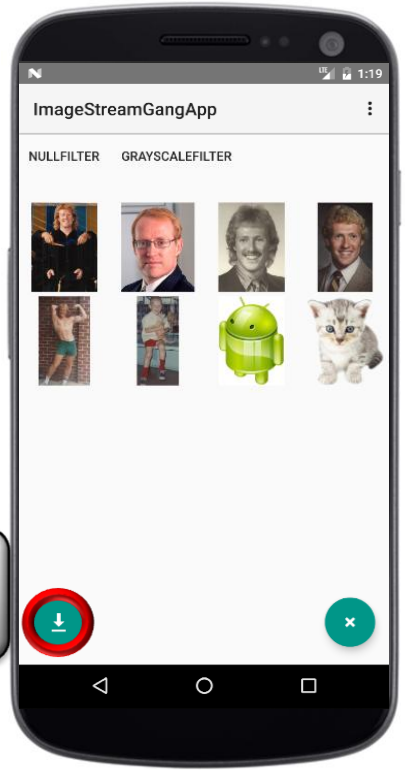
User supplies the list of URLs to download.

Overview of the Pattern-Oriented ImageStreamGang App

- This app combines streams and CompletableFutures with the StreamGang framework to download, transform, store, and display images, for example.



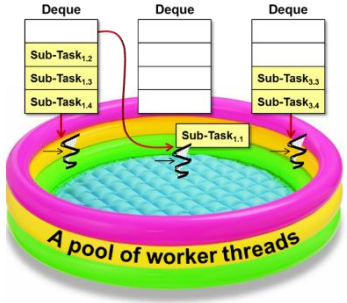


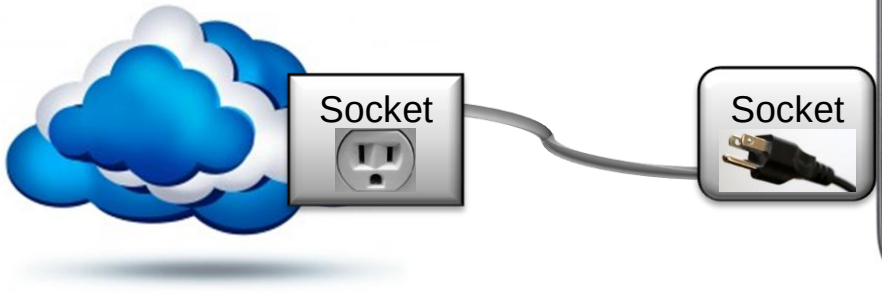


Download images via one or more threads.

Overview of the Pattern-Oriented ImageStreamGang App

- This app combines streams and CompletableFutures with the StreamGang framework to download, transform, store, and display images, for example.

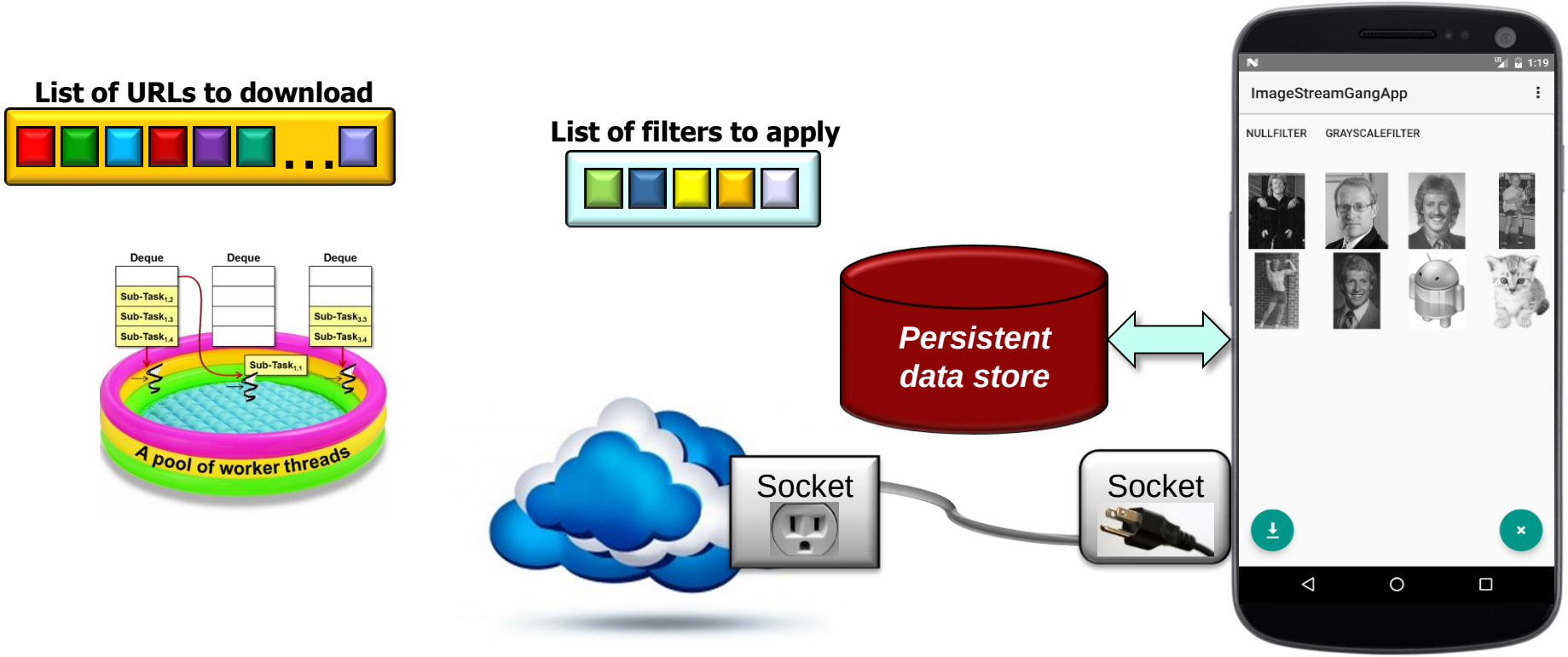




Apply filters to transform downloaded images.

Overview of the Pattern-Oriented ImageStreamGang App

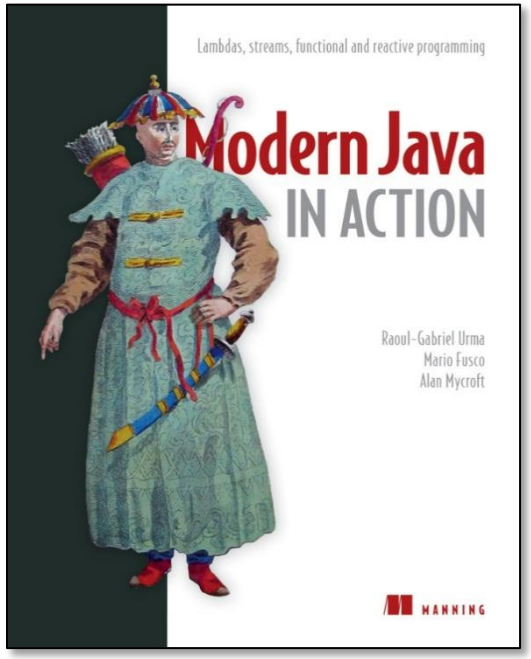
- This app combines streams and CompletableFutures with the StreamGang framework to download, transform, store, and display images, for example.



Output filtered images to persistent storage.

Overview of the Pattern-Oriented ImageStreamGang App

- The ImageStreamGang app applies a range of modern Java features.



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

Overview of the Pattern-Oriented ImageStreamGang App

- The ImageStreamGang app applies a range of modern Java features, e.g.
  - Sequential and parallel streams

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

We'll cover parallel streams now.

Overview of the Pattern-Oriented ImageStreamGang App

- The ImageStreamGang app applies a range of modern Java features, e.g.
  - Sequential and parallel streams
  - CompletableFutures

```
getInput()
  .stream()
  .map(this::checkUrlCachedAsync)
  .map(this::downloadImageAsync)
  .flatMap(this::applyFiltersAsync)
  .collect(toFuture())
  .thenAccept
    (stream ->
     log(stream.flatMap(Optional::stream),
         urls.size()))
  .join();
```

We'll cover CompletableFutures later.

Overview of the Pattern-Oriented ImageStreamGang App

- The ImageStreamGang app applies a range of modern Java features, e.g.
  - Sequential and parallel streams
  - CompletableFutures
  - Lambda expressions and method references

```
Runnable mCompletionHook =
   () -> MainActivity.this.runOnUiThread
      (this::goToResultActivity);
```

We covered these foundational Java features earlier.

Overview of the Pattern-Oriented ImageStreamGang App

- The ImageStreamGang app applies a range of modern Java features, e.g.
  - Sequential and parallel streams
  - CompletableFutures
  - Lambda expressions and method references

```
Runnable mCompletionHook =
   () -> MainActivity.this.runOnUiThread
      (this::goToResultActivity);
```

versus

```
Runnable mCompletionHook = new Runnable() {
    public void run() {
    MainActivity.this.runOnUiThread
     (new Runnable() { public void run()
       { goToResultActivity(); } }); }};
```

Overview of Patterns Applied in the ImageStreamGang App

Overview of Patterns Applied in the ImageStreamGang App

- "Gang-of-Four" and POSA patterns are applied to enhance its framework-based architecture.

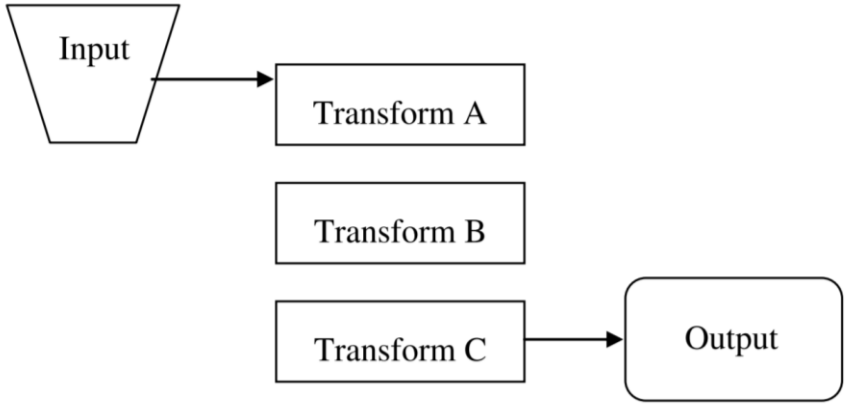
See en.wikipedia.org/wiki/Design_Patterns & www.dre.vanderbilt.edu/~schmidt/POSA

Overview of Patterns Applied in the ImageStreamGang App

- Some patterns are essential to its design.

Overview of Patterns Applied in the ImageStreamGang App

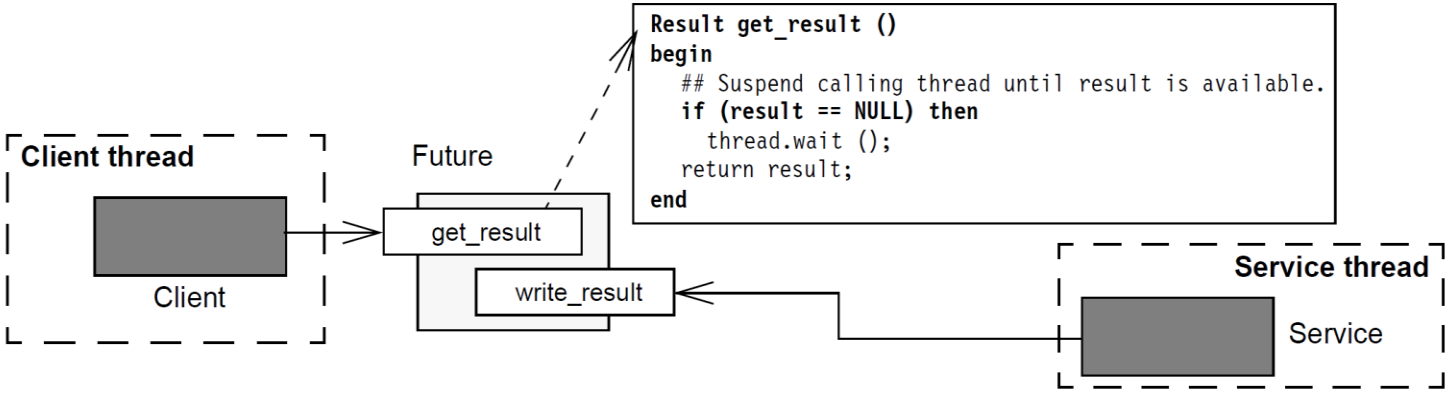
- Some patterns are essential to its design.
  - *Pipes and filters*
    - Divide application's tasks into several self-contained data processing steps and connect these steps via intermediate data buffers to a data processing pipeline



See www.hillside.net/plop/2011/papers/B-10-Hanmer.pdf

Overview of Patterns Applied in the ImageStreamGang App

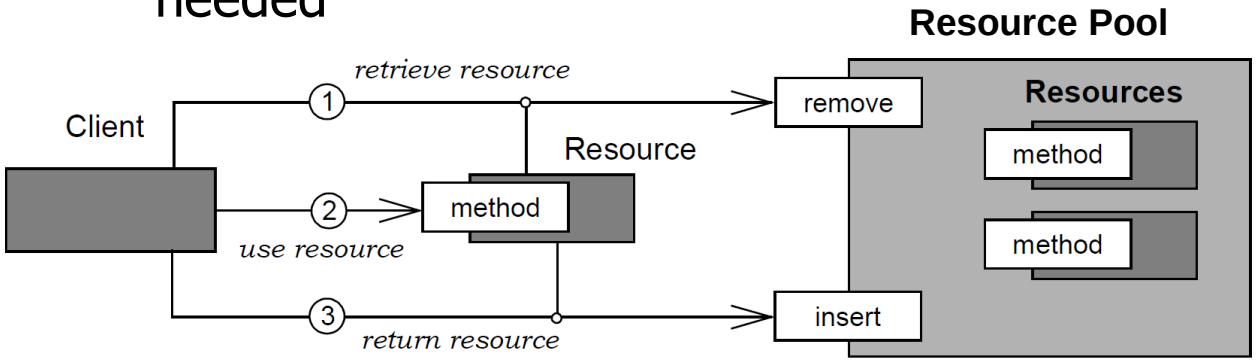
- Some patterns are essential to its design.
  - *Future*
    - Provides a proxy to a client when it invokes a service to keep track of the state of the service's concurrent computation and only returns a value to the client when the computation completes



See en.wikipedia.org/wiki/Futures_and_promises

Overview of Patterns Applied in the ImageStreamGang App

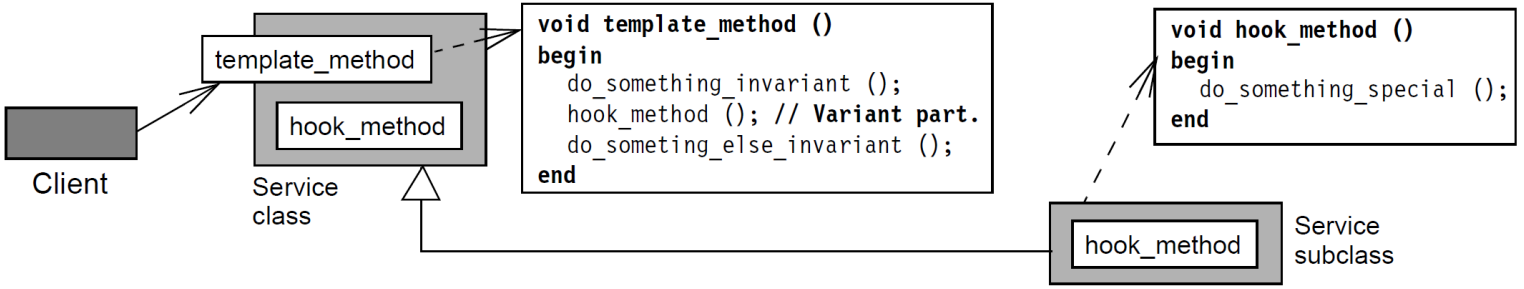
- Some patterns are essential to its design.
  - *ResourcePool*
    - Prevents expensive acquisition and release of resources by recycling resources no longer needed

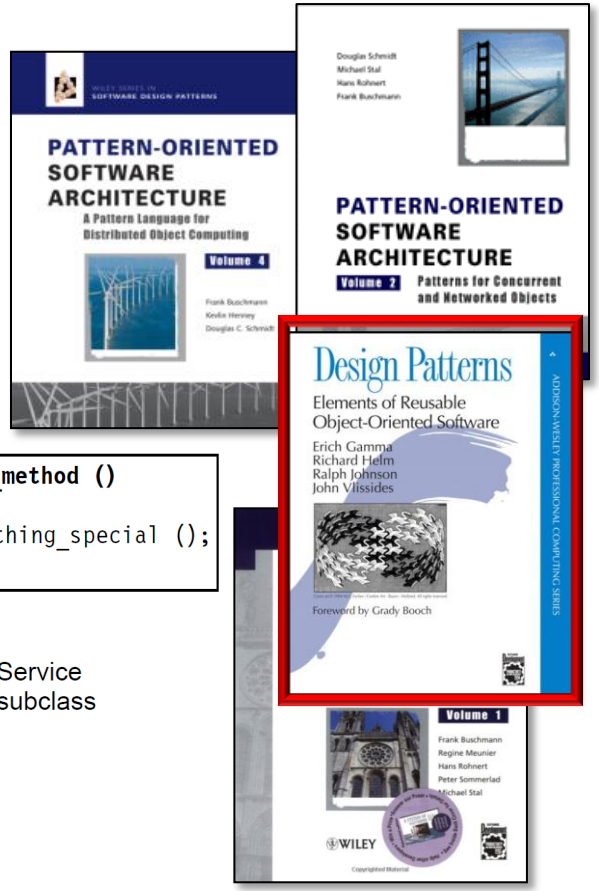


See kircher-schwanninger.de/michael/publications/Pooling.pdf

Overview of Patterns Applied in the ImageStreamGang App

- Some patterns are essential to its design.
  - *Template method*
    - Defines the overall structure of a method, while allowing subclasses to refine, or redefine, certain steps

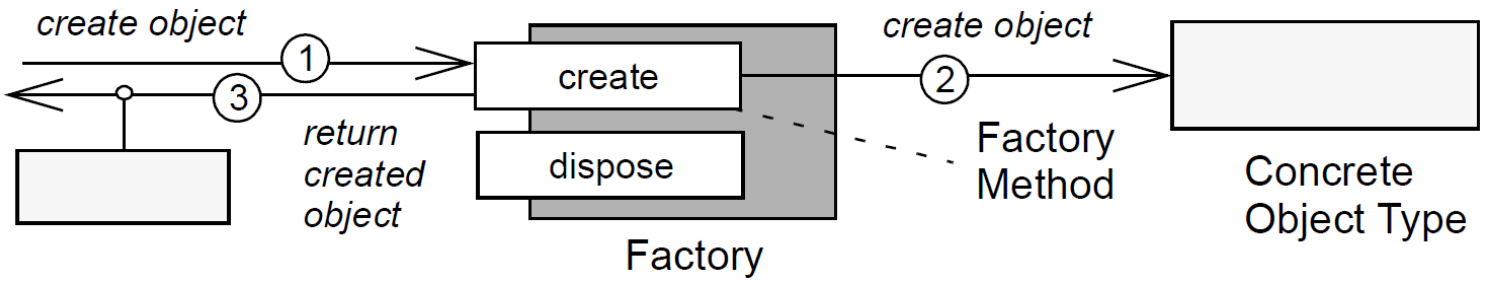




See en.wikipedia.org/wiki/Template_method_pattern

Overview of Patterns Applied in the ImageStreamGang App

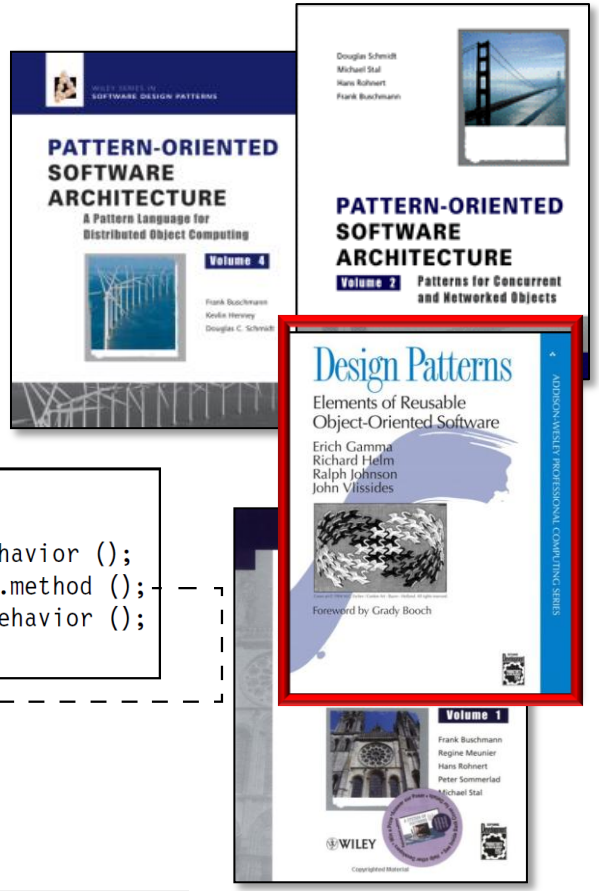
- Some patterns are essential to its design.
  - *Factory method*
    - Encapsulate the concrete details of object creation inside a factory method, rather than letting clients create the object themselves

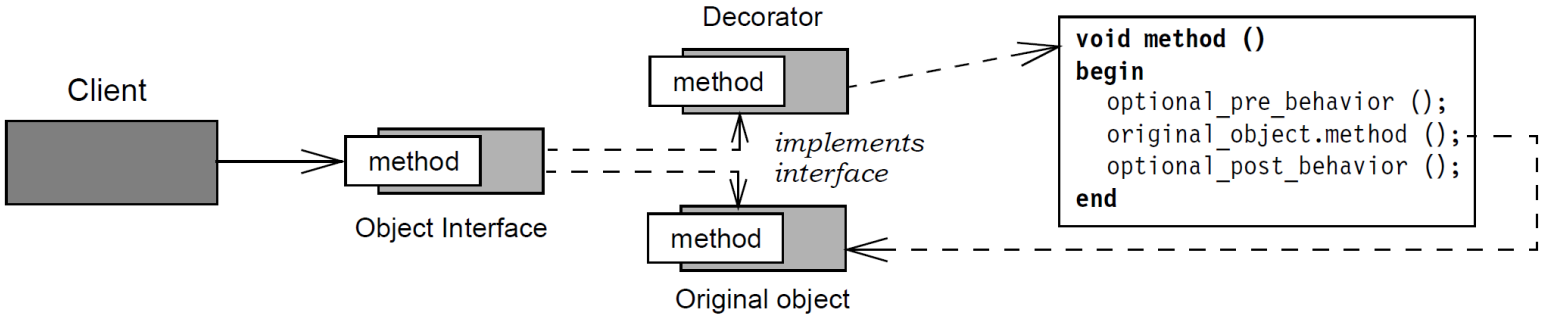


See en.wikipedia.org/wiki/Factory_method_pattern

Overview of Patterns Applied in the ImageStreamGang App

- Some patterns are essential to its design.
  - *Decorator*
    - Allows behavior to be added to an individual object, dynamically, without affecting the behavior of other objects from the same class

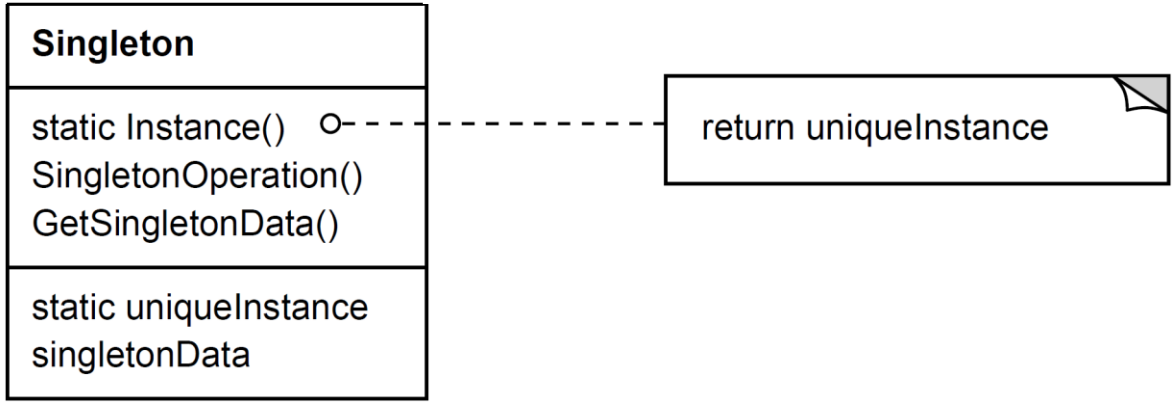




See en.wikipedia.org/wiki/Decorator_pattern

Overview of Patterns Applied in the ImageStreamGang App

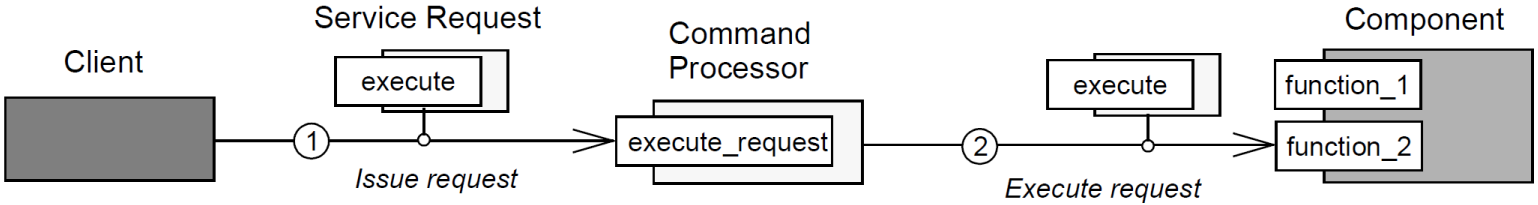
- Other patterns are also applied.
  - *Singleton*
    - Ensures a class has only one instance and provide a global point of access to it



See en.wikipedia.org/wiki/Singleton_pattern

Overview of Patterns Applied in the ImageStreamGang App

- Other patterns are also applied.
  - *CommandProcessor*
    - Packages a piece of application functionality—as well as its parameterization in an object—to make it usable in another context, such as later in time or in a different thread

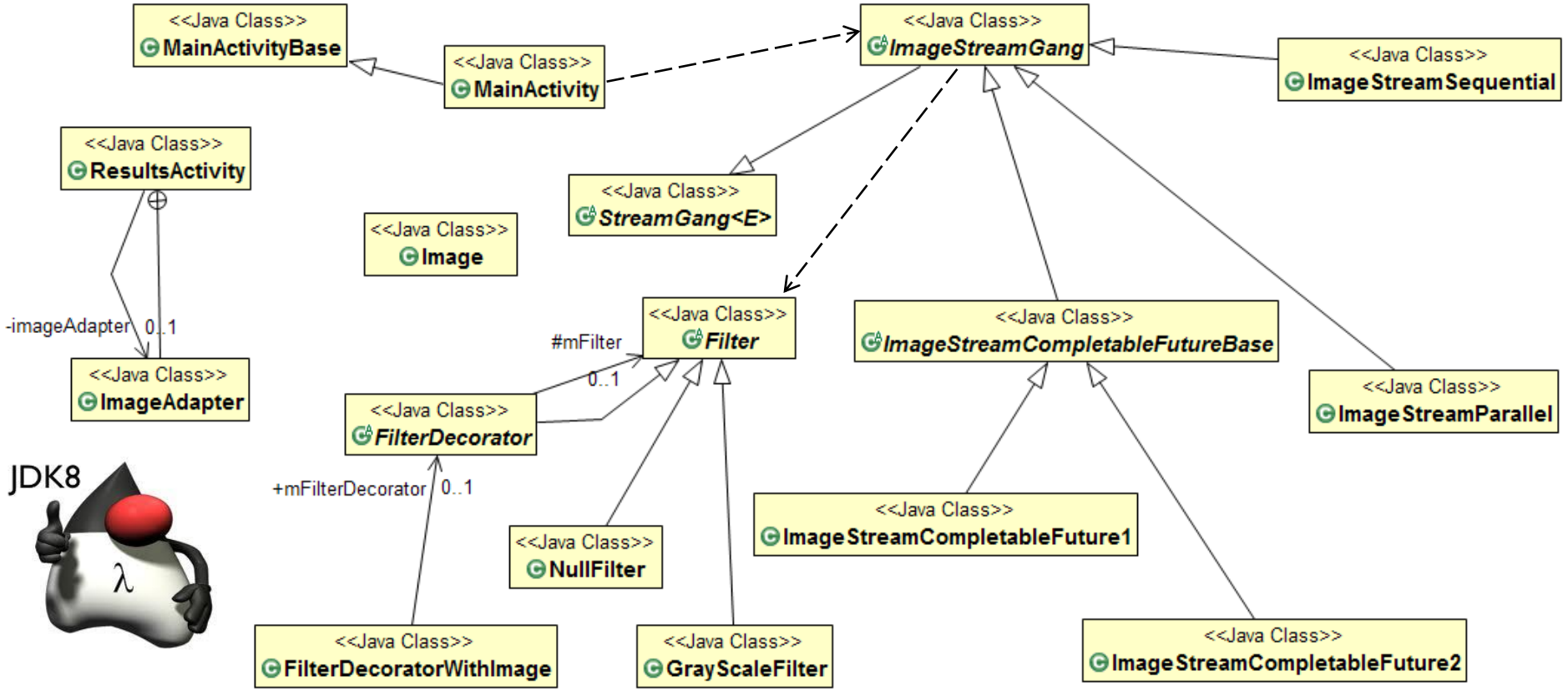


See www.dre.vanderbilt.edu/~schmidt/CommandProcessor.pdf

Strategy for Understanding the ImageStreamGang App

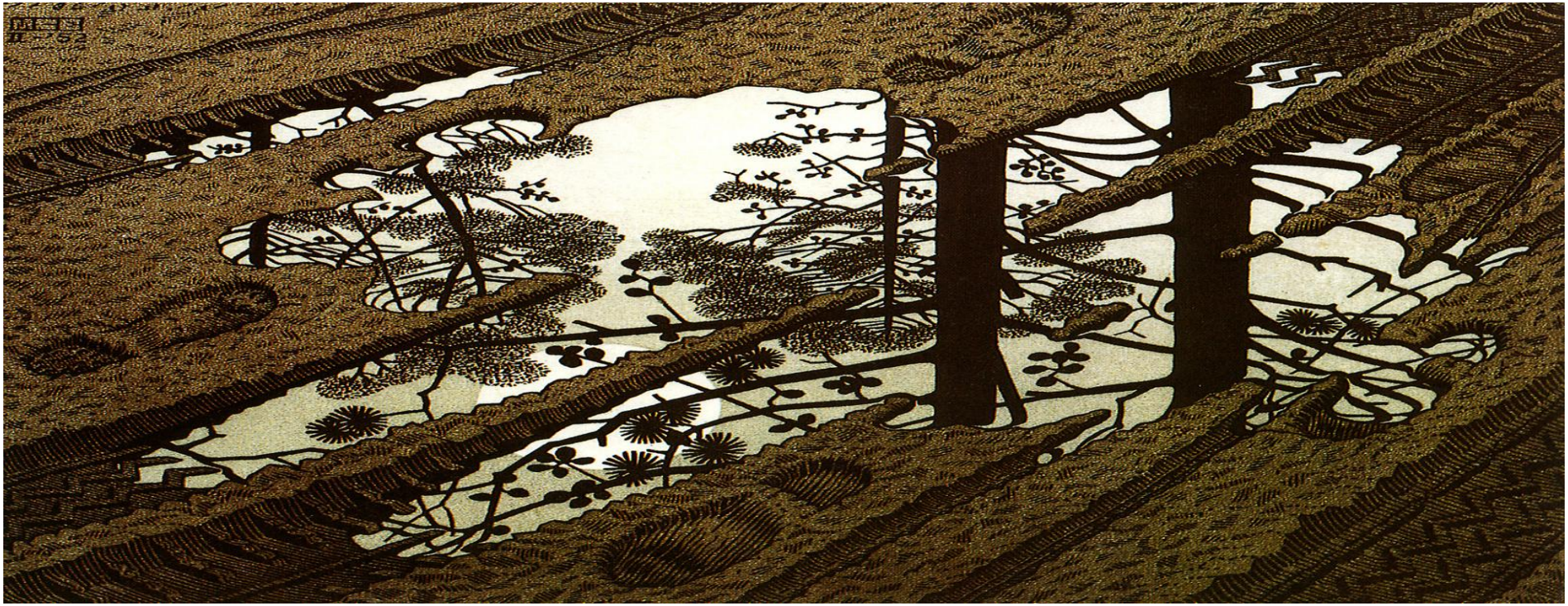
Strategy for Understanding the ImageStreamGang App

- This app is complicated and contains many classes.



Strategy for Understanding the ImageStreamGang App

- This app is complicated and contains many classes.
  - We therefore analyze it from various perspectives.



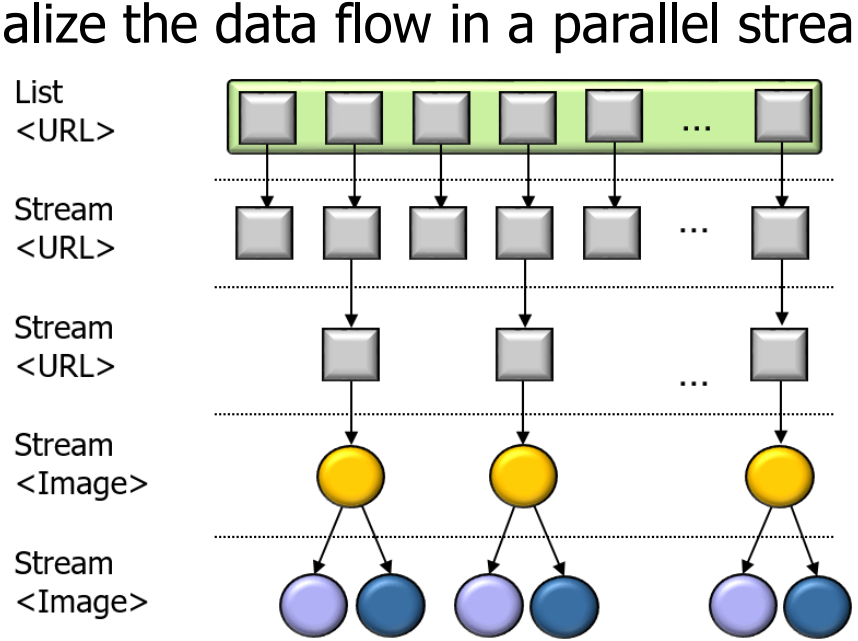
Including pattern-oriented design, data flows, and detailed code walk-throughs

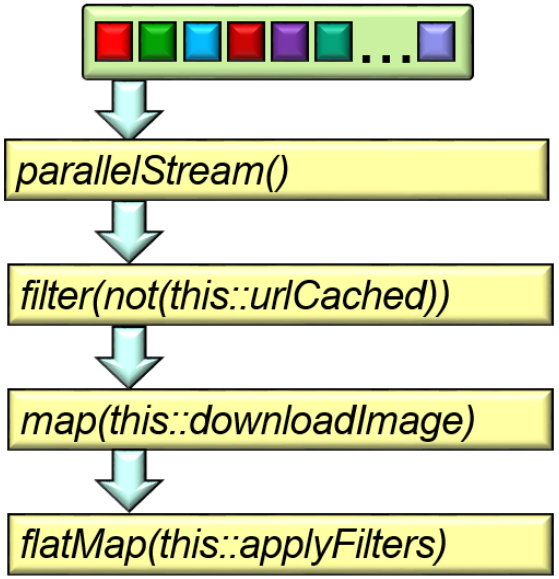
Strategy for Understanding the ImageStreamGang App

- This app is complicated and contains many classes.
  - We therefore analyze it from various perspectives.
  - Watch this entire lesson carefully to understand how it all works.

Strategy for Understanding the ImageStreamGang App

- This app is complicated and contains many classes.
  - We therefore analyze it from various perspectives.
  - Watch this entire lesson carefully to understand how it all works.
  - Visualize the data flow in a parallel stream.





Strategy for Understanding the ImageStreamGang App

- This app is complicated and contains many classes.
  - We therefore analyze it from various perspectives.
  - Watch this entire lesson carefully to understand how it all works.
  - Visualize the data flow in a parallel stream.
  - Run/read the code to see how it all works.

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

The End

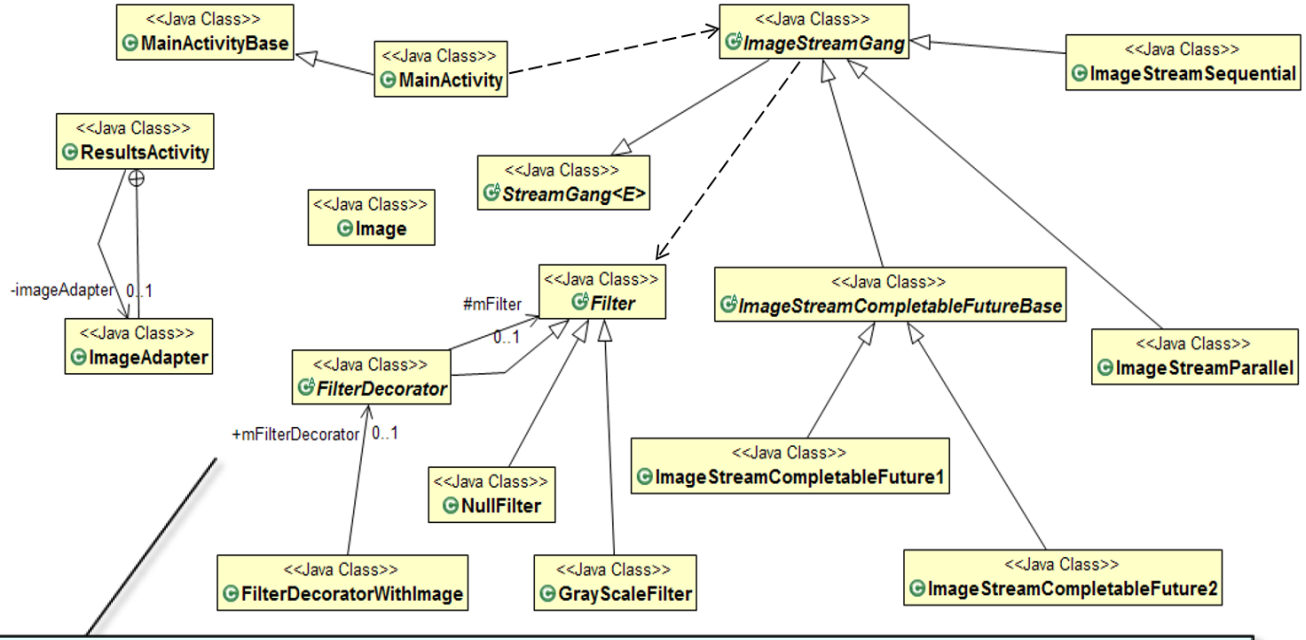
Java Parallel ImageStreamGang Example

Structure and Functionality

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Recognize the structure and functionality of the ImageStreamGang app.
  - It applies several Java parallelism frameworks.
  - The focus is on integrating object-oriented and functional programming paradigms.

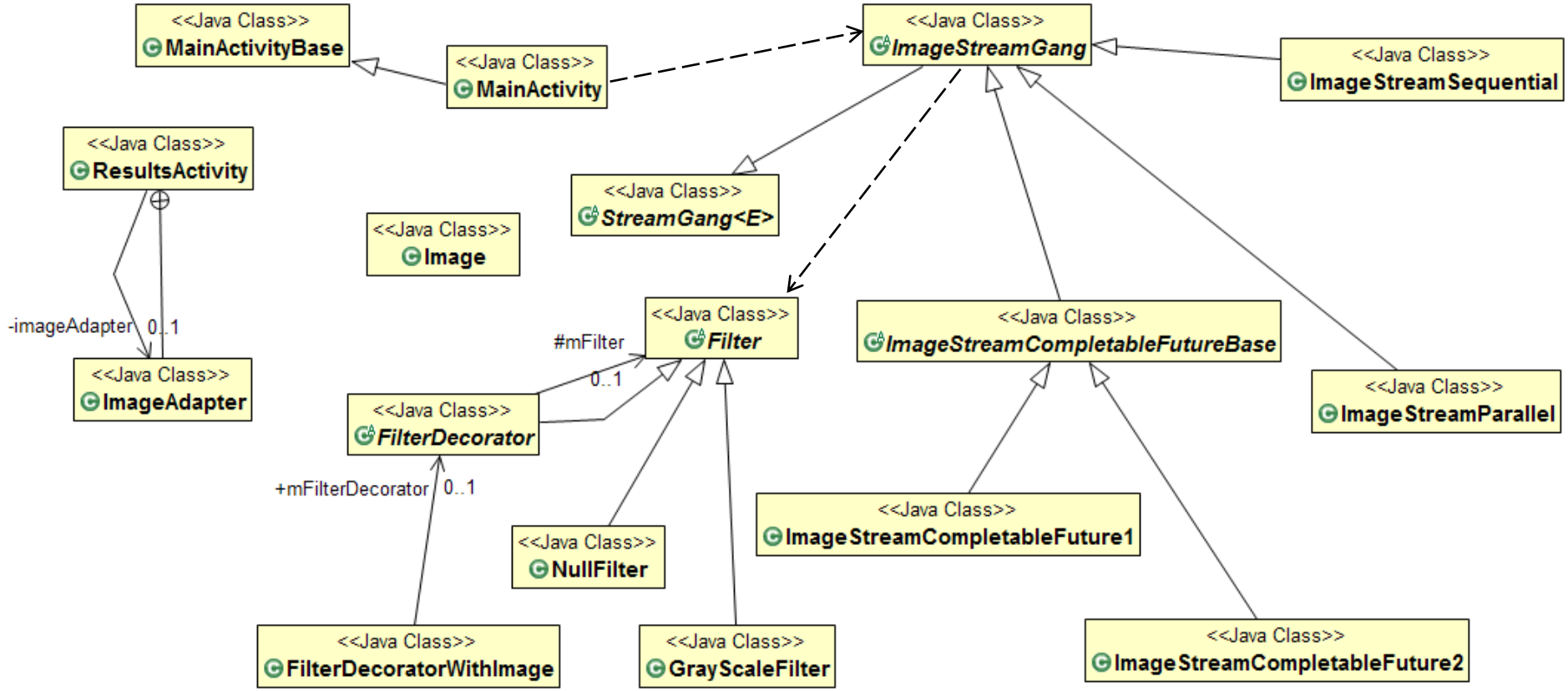


This design shows the synergy between object-oriented and functional programming.

The Structure of the ImageStreamGang App

The Structure of the ImageStreamGang App

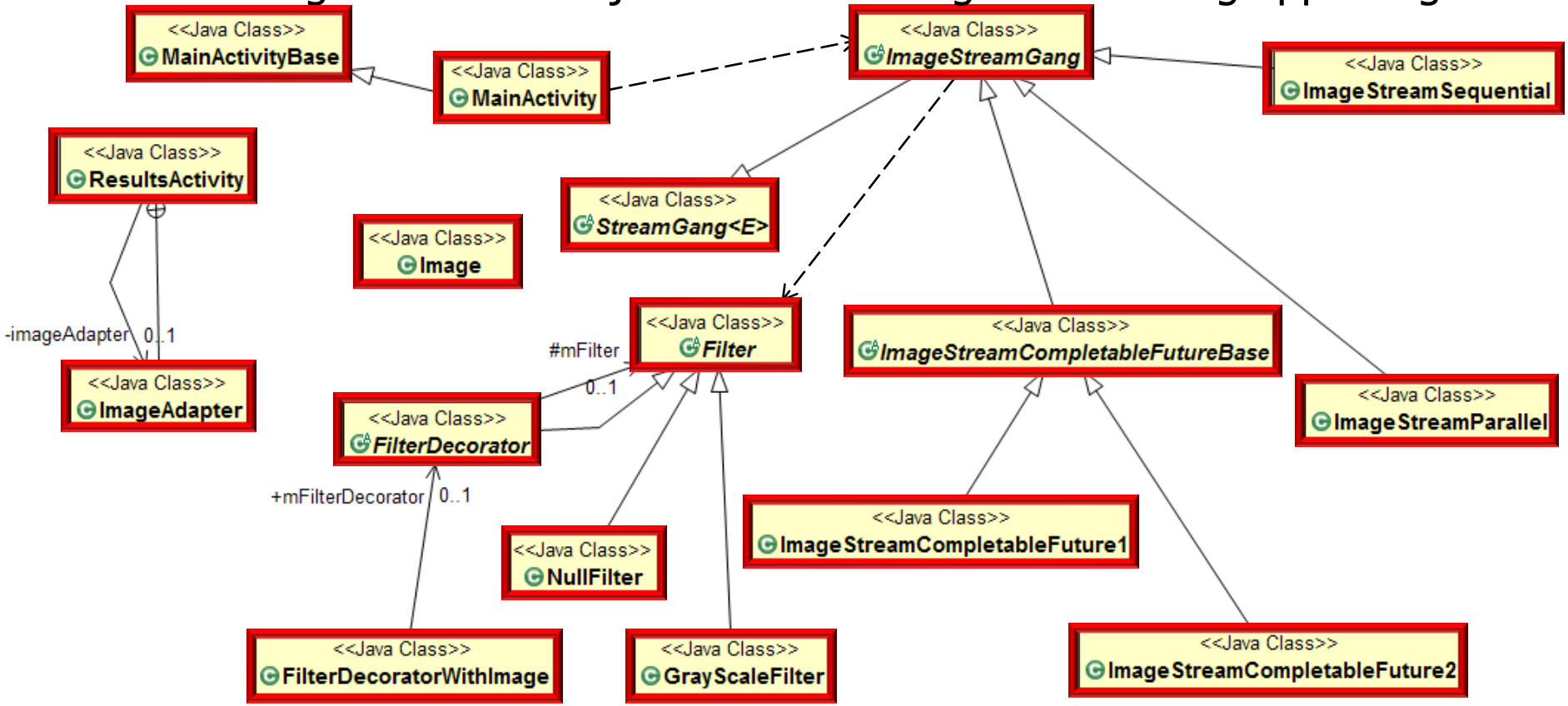
- UML class diagram for the object-oriented ImageStreamGang app design



See en.wikipedia.org/wiki/Unified_Modeling_Language

The Structure of the ImageStreamGang App

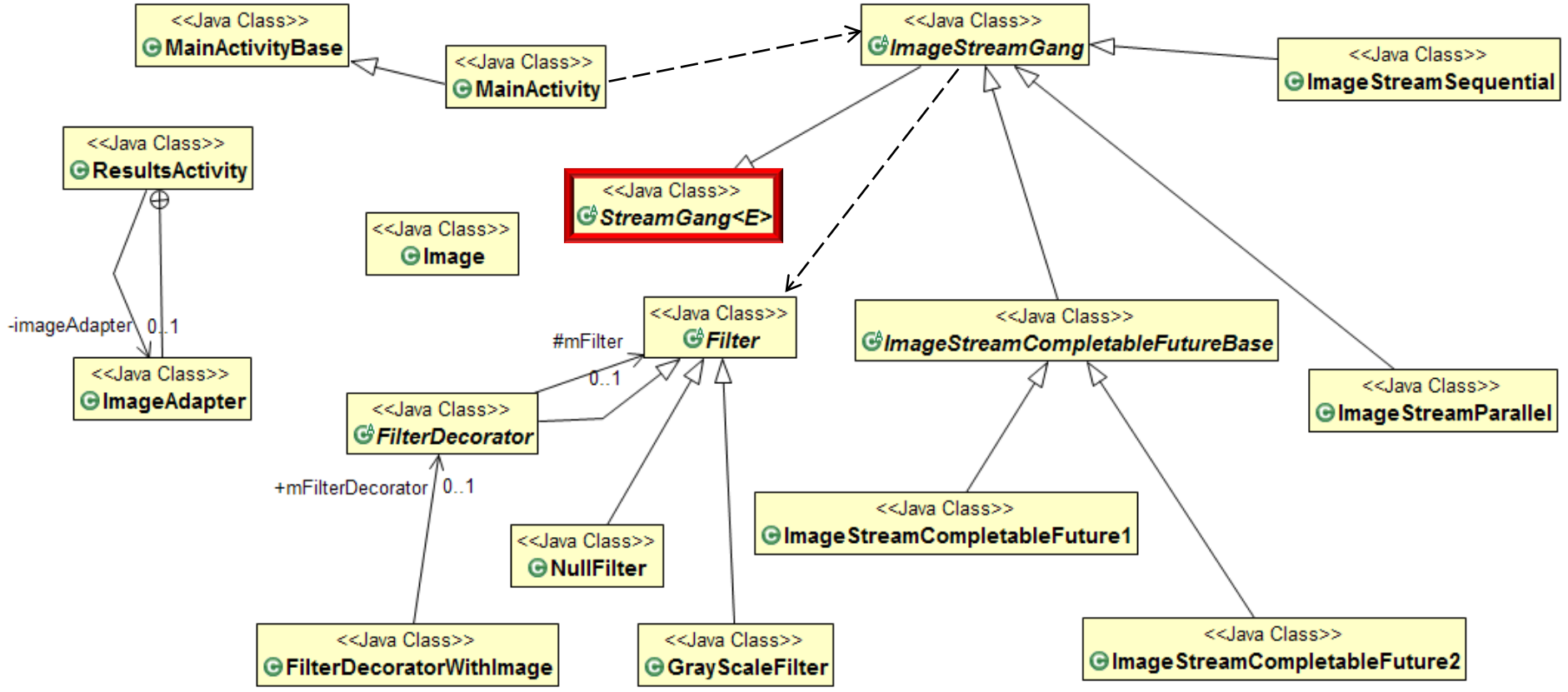
- UML class diagram for the object-oriented ImageStreamGang app design



These classes apply Java features to image downloading and processing.

The Structure of the ImageStreamGang App

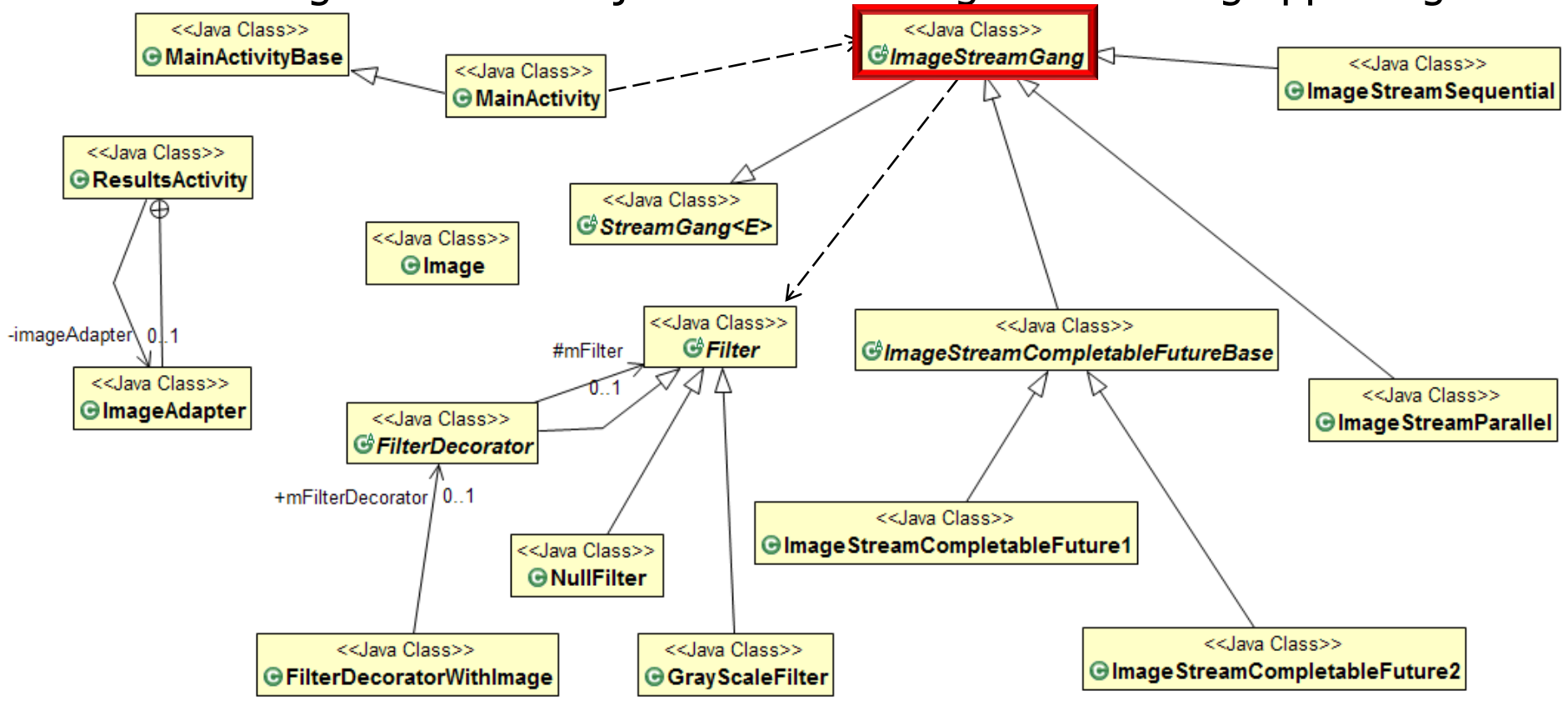
- UML class diagram for the object-oriented ImageStreamGang app design



A framework for initiating streams that process input from a list of elements

The Structure of the ImageStreamGang App

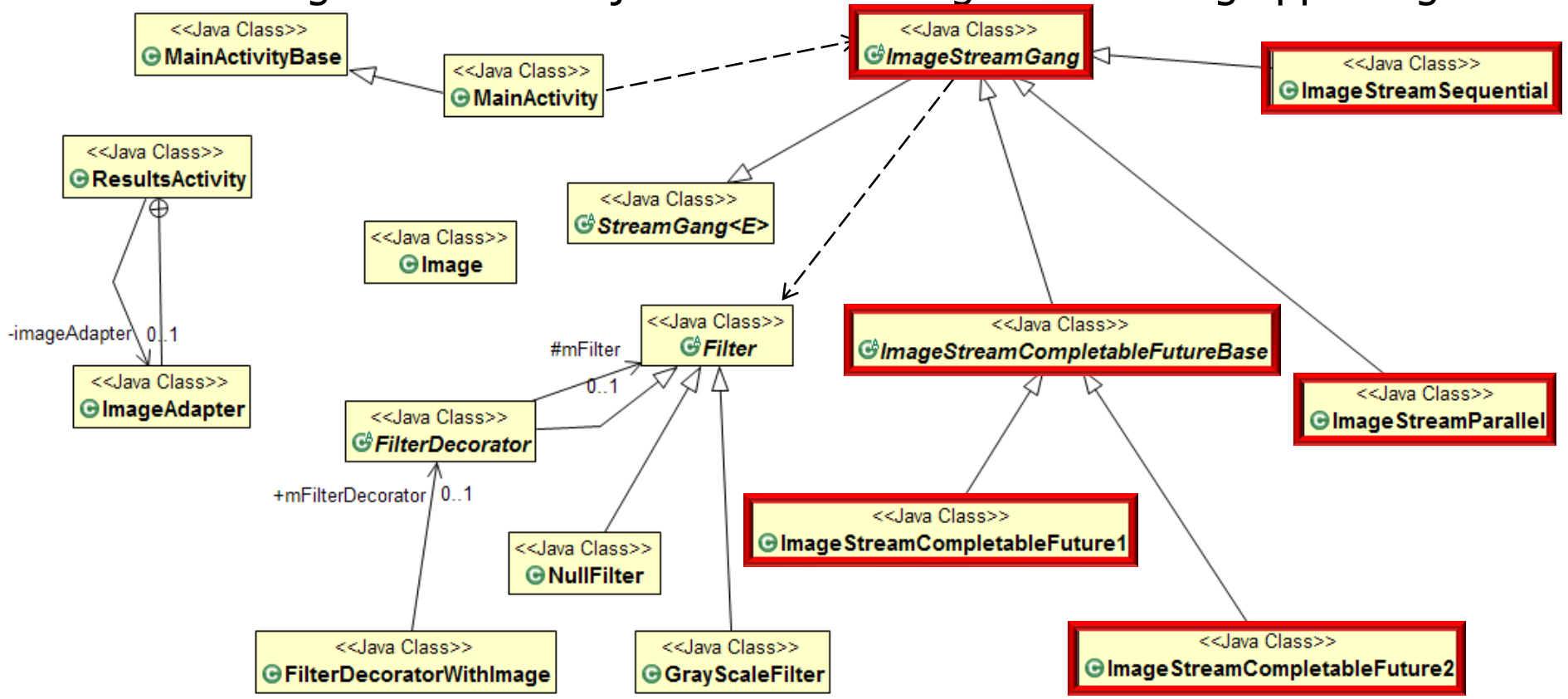
- UML class diagram for the object-oriented ImageStreamGang app design



Customizes the StreamGang framework to download and process images . . .

The Structure of the ImageStreamGang App

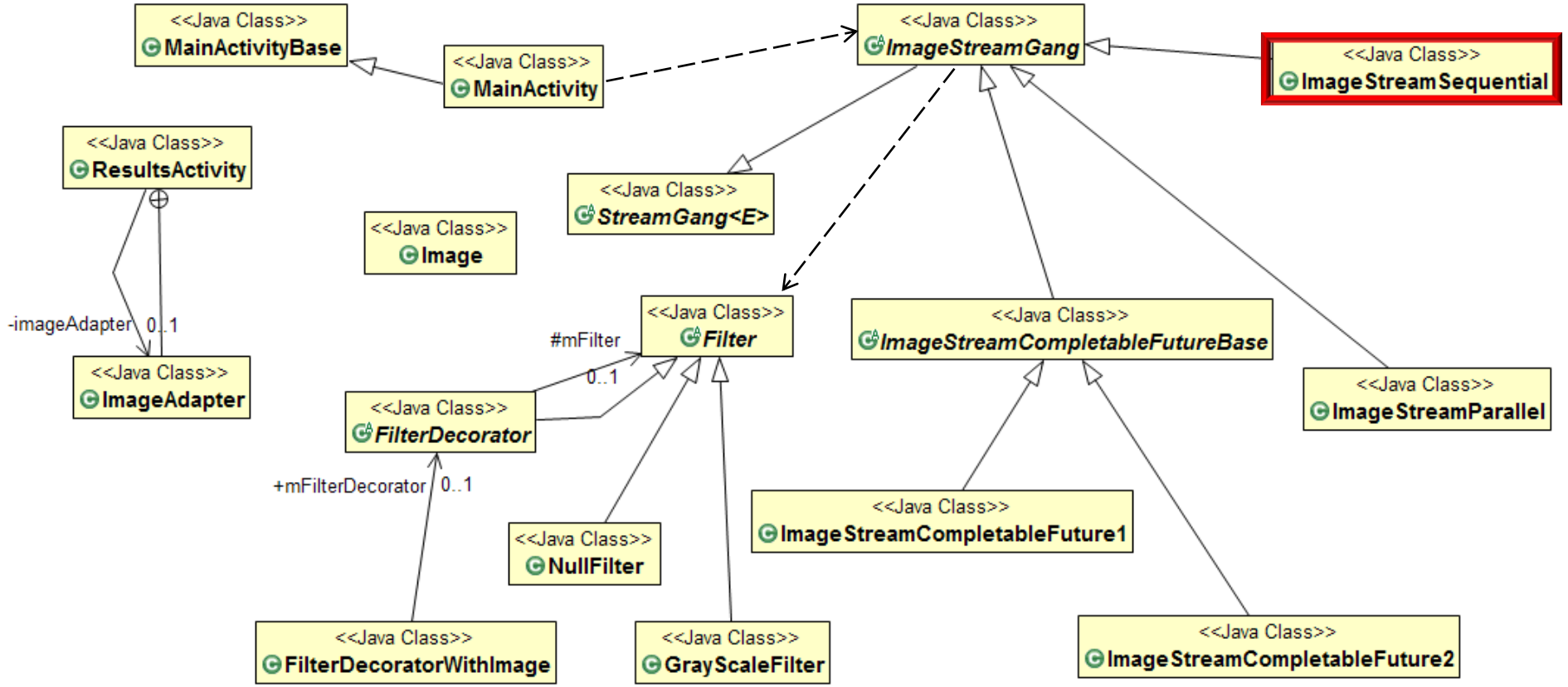
- UML class diagram for the object-oriented ImageStreamGang app design



. . . based on different Java concurrency and parallelism frameworks

The Structure of the ImageStreamGang App

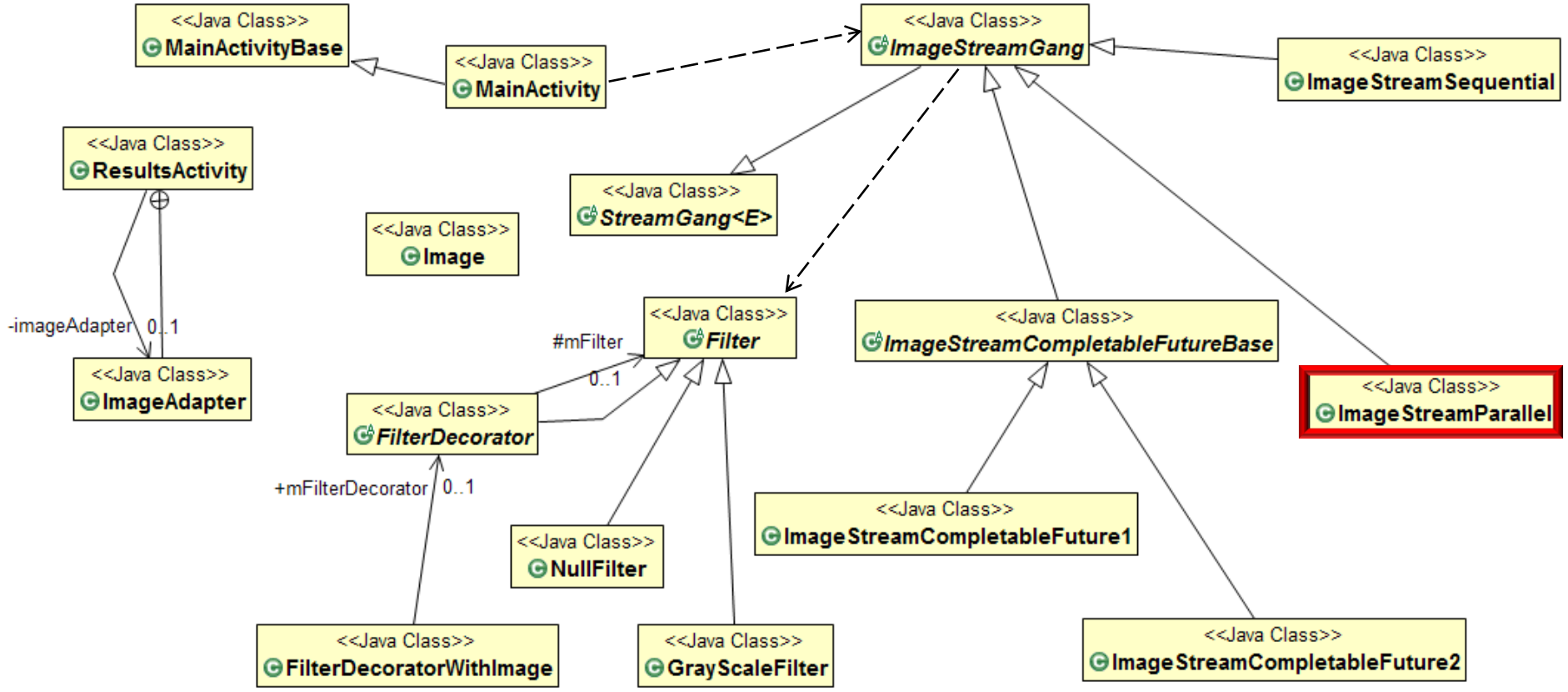
- UML class diagram for the object-oriented ImageStreamGang app design



Uses Java streams to download and filter images sequentially

The Structure of the ImageStreamGang App

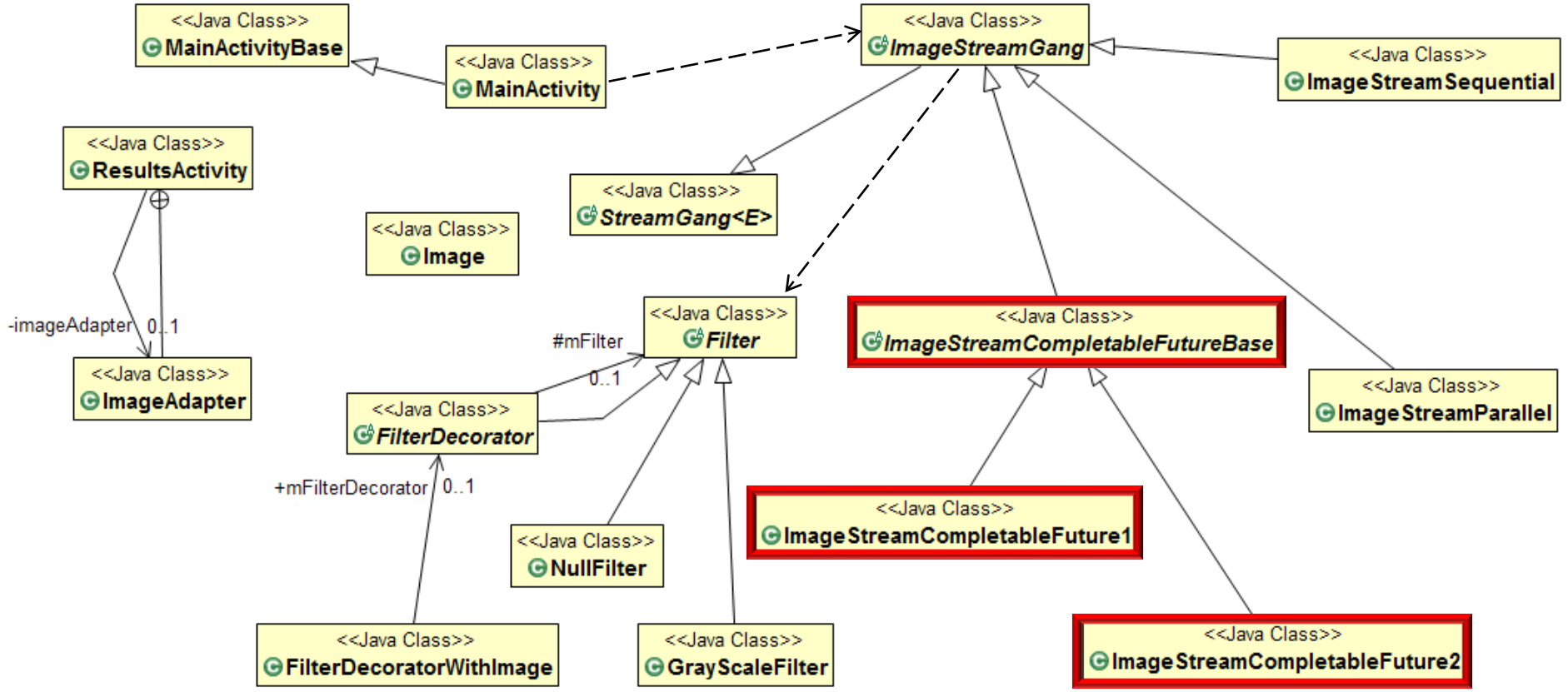
- UML class diagram for the object-oriented ImageStreamGang app design



Uses Java parallel streams to download and filter images concurrently

The Structure of the ImageStreamGang App

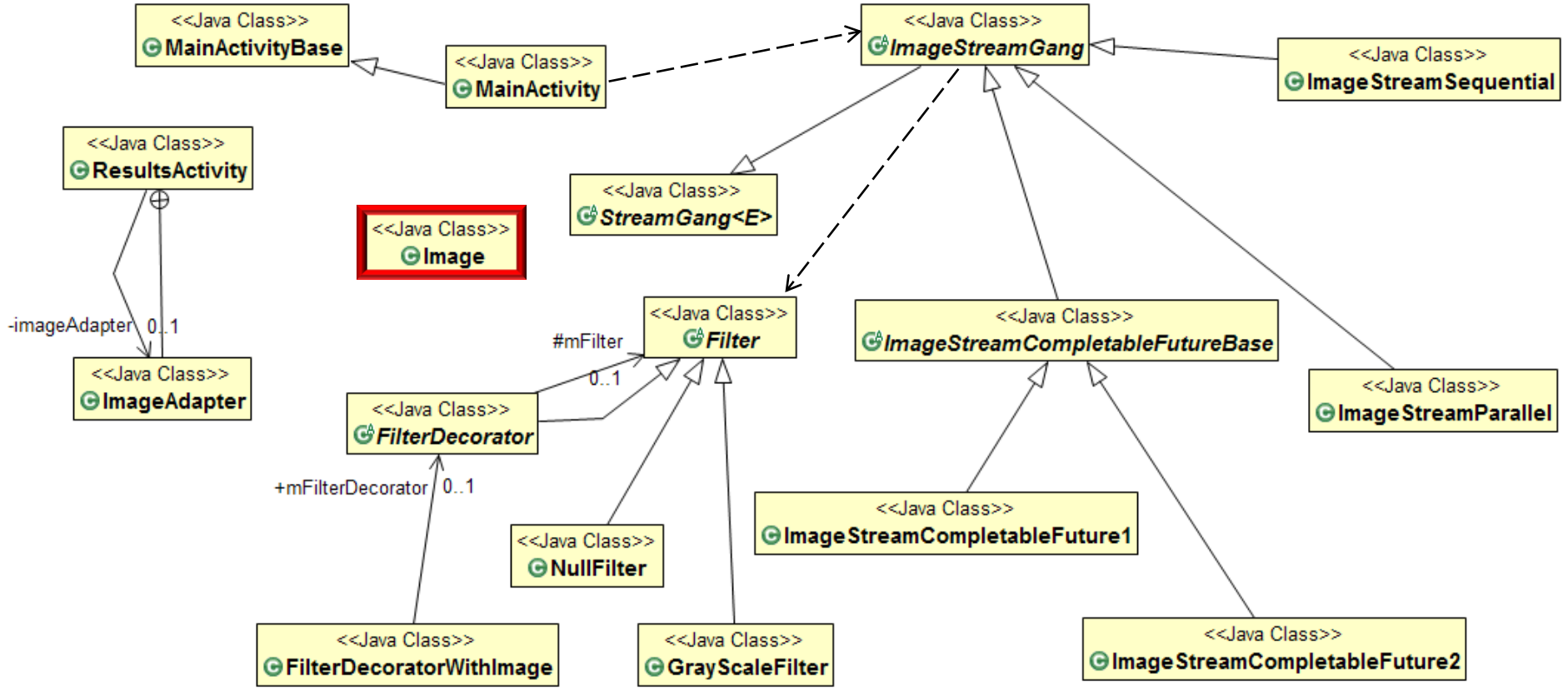
- UML class diagram for the object-oriented ImageStreamGang app design



Uses Java CompletableFutures to download and filter images asynchronously

The Structure of the ImageStreamGang App

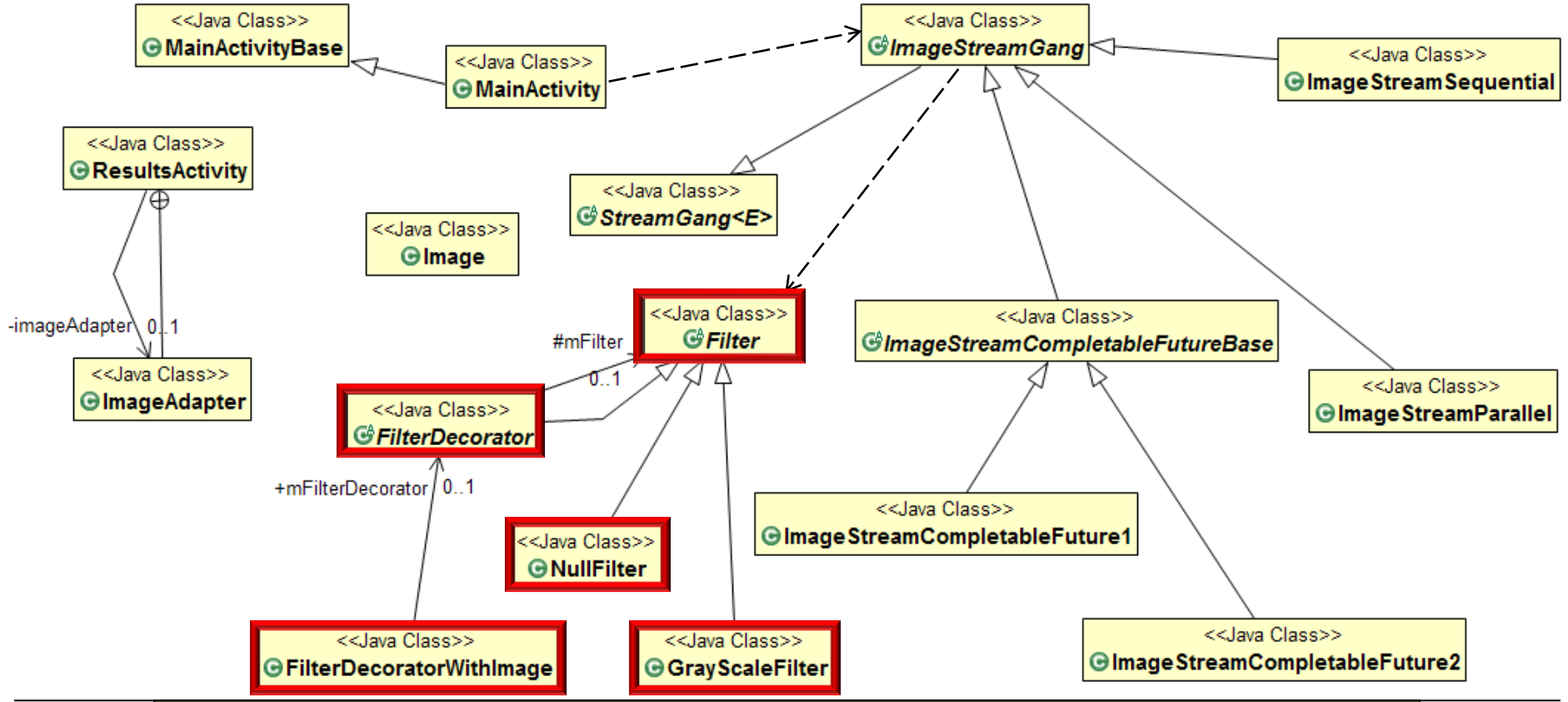
- UML class diagram for the object-oriented ImageStreamGang app design



Stores image metadata and provides methods for common image-/file-related tasks

The Structure of the ImageStreamGang App

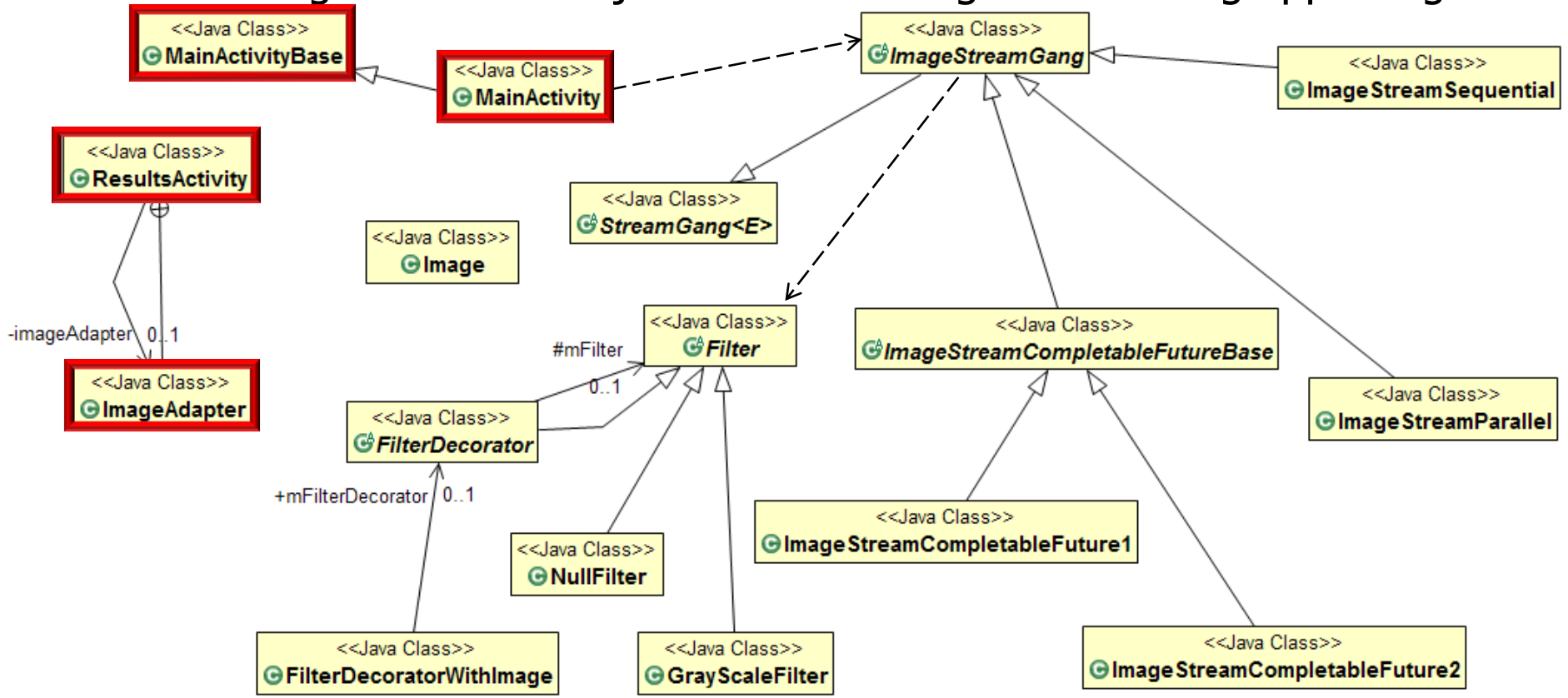
- UML class diagram for the object-oriented ImageStreamGang app design



This class hierarchy applies operations to filter and store images.

The Structure of the ImageStreamGang App

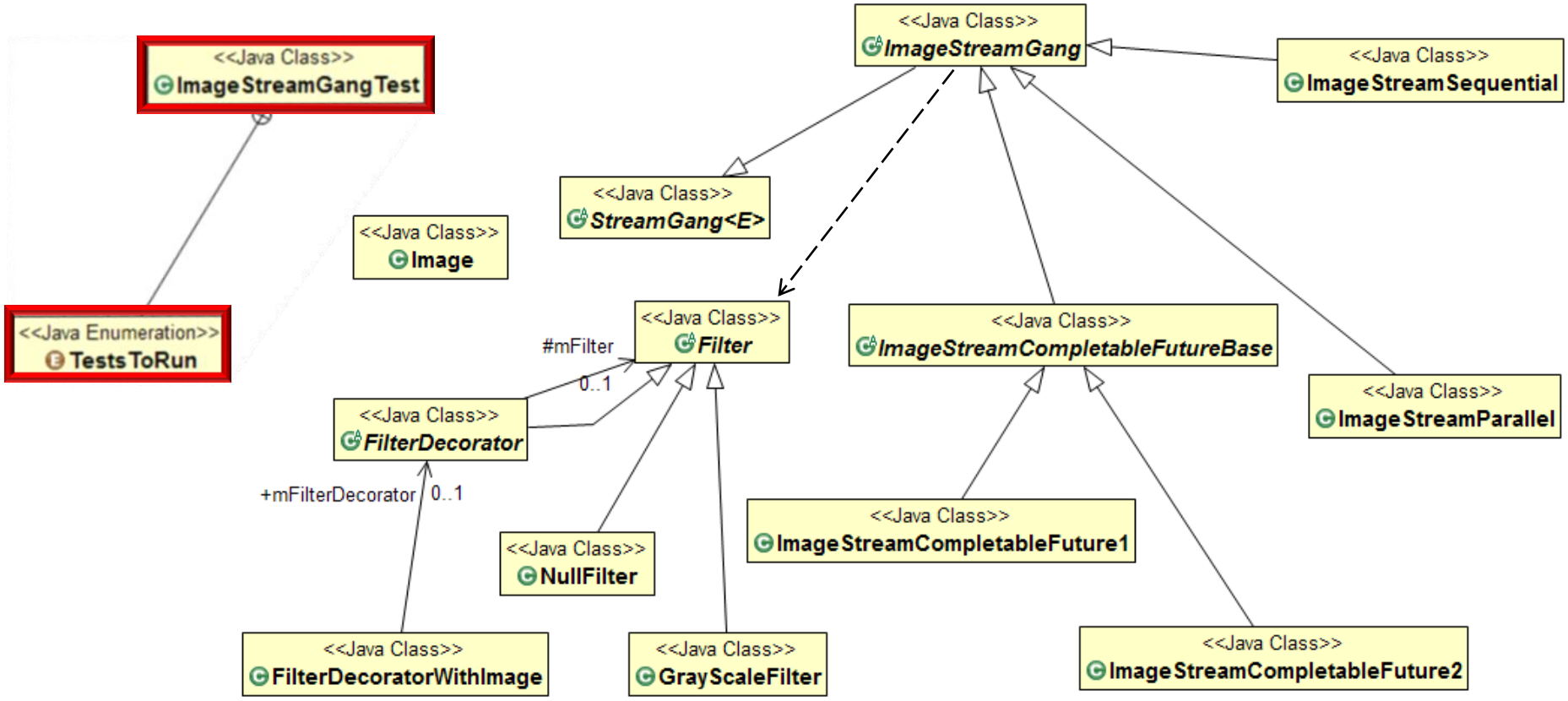
- UML class diagram for the object-oriented ImageStreamGang app design



Provides the user interface for an Android app

The Structure of the ImageStreamGang App

- UML class diagram for the object-oriented ImageStreamGang app design



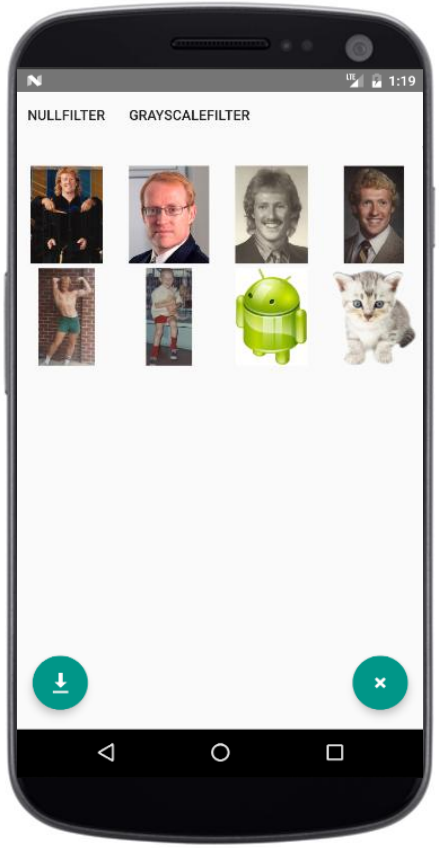
There's a Java console version of ImageStreamGang that shares most of the code.

Running the ImageStreamGang App

Running the ImageStreamGang App

Starting ImageStreamGangTest
Printing 4 results for input file 1 from fastest to slowest
COMPLETABLE_FUTURES_1 executed in 312 msecs
COMPLETABLE_FUTURES_2 executed in 335 msecs
PARALLEL_STREAM executed in 428 msecs
SEQUENTIAL_STREAM executed in 981 msecs

Printing 4 results for input file 2 from fastest to slowest
COMPLETABLE_FUTURES_2 executed in 82 msecs
COMPLETABLE_FUTURES_1 executed in 83 msecs
PARALLEL_STREAM executed in 102 msecs
SEQUENTIAL_STREAM executed in 251 msecs
Ending ImageStreamGangTest



Tests conducted on a 2.6 GHz six-core Lenovo P52 with 64 GB of RAM

Java Parallel ImageStreamGang Example:
Structure and Functionality

The End

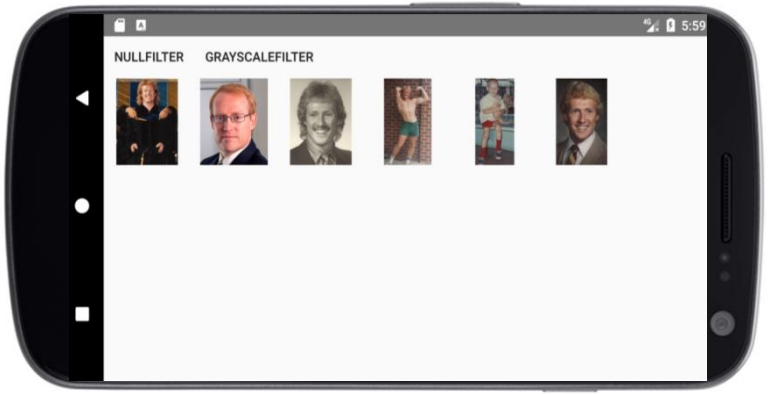
Java Parallel ImageStreamGang Example

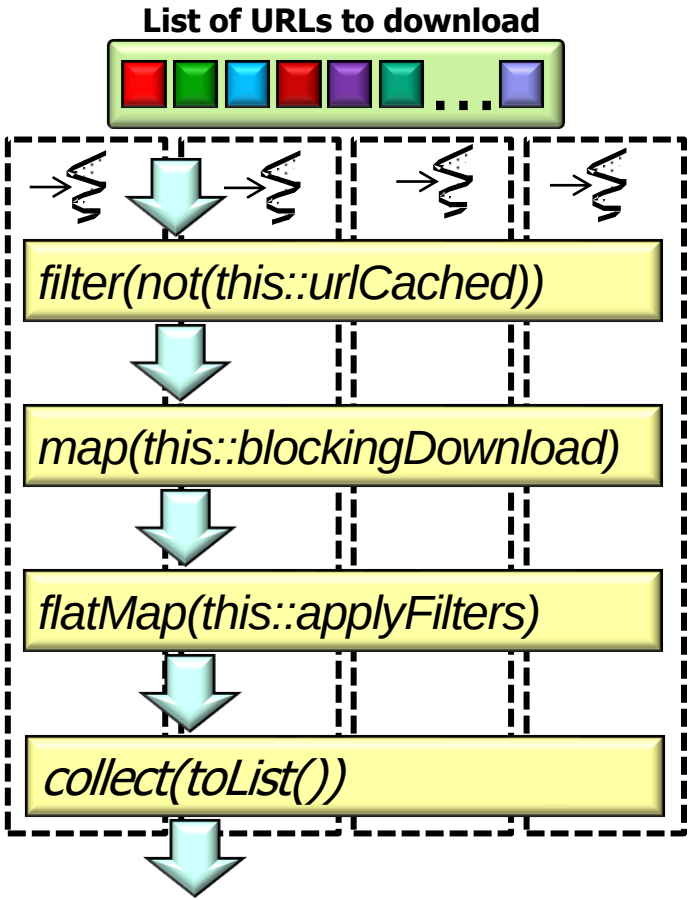
Visualizing Behaviors

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Recognize the structure/functionality of the ImageStreamGang app.
- Know how Java parallel streams are applied to the ImageStreamGang app.

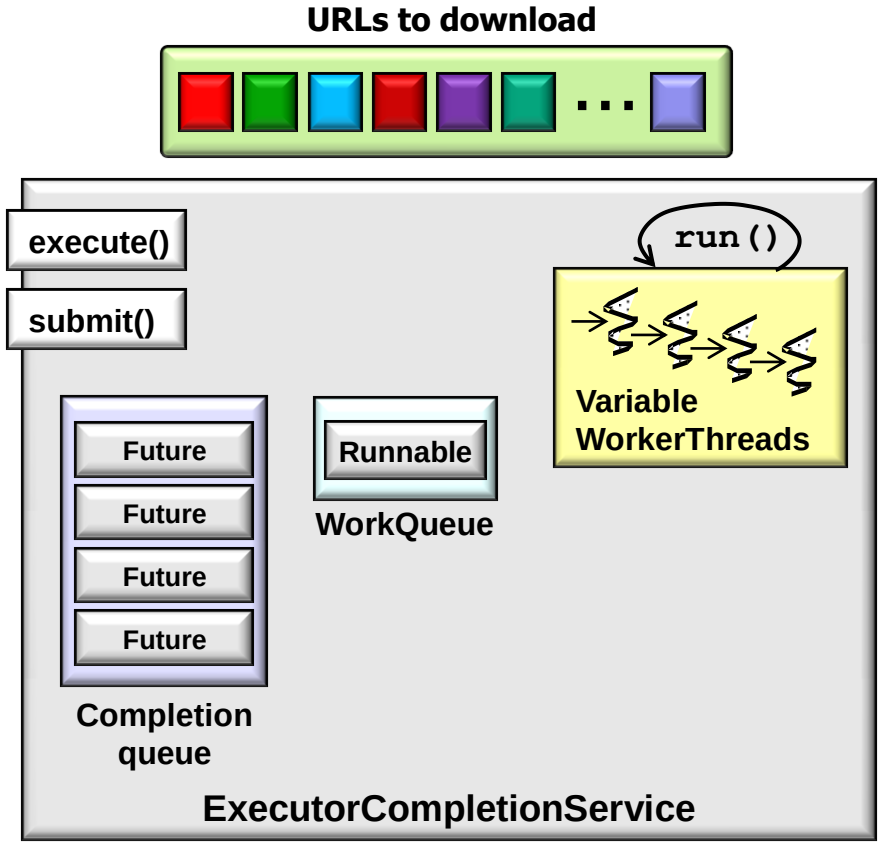




See github.com/douglascraigschmidt/LiveLessons/blob/master/ImageStreamGang

Learning Objectives in This Part of the Lesson

- Recognize the structure/functionality of the ImageStreamGang app.
- Know how Java parallel streams are applied to the ImageStreamGang app.
  - This app enhances ImageTaskGang.

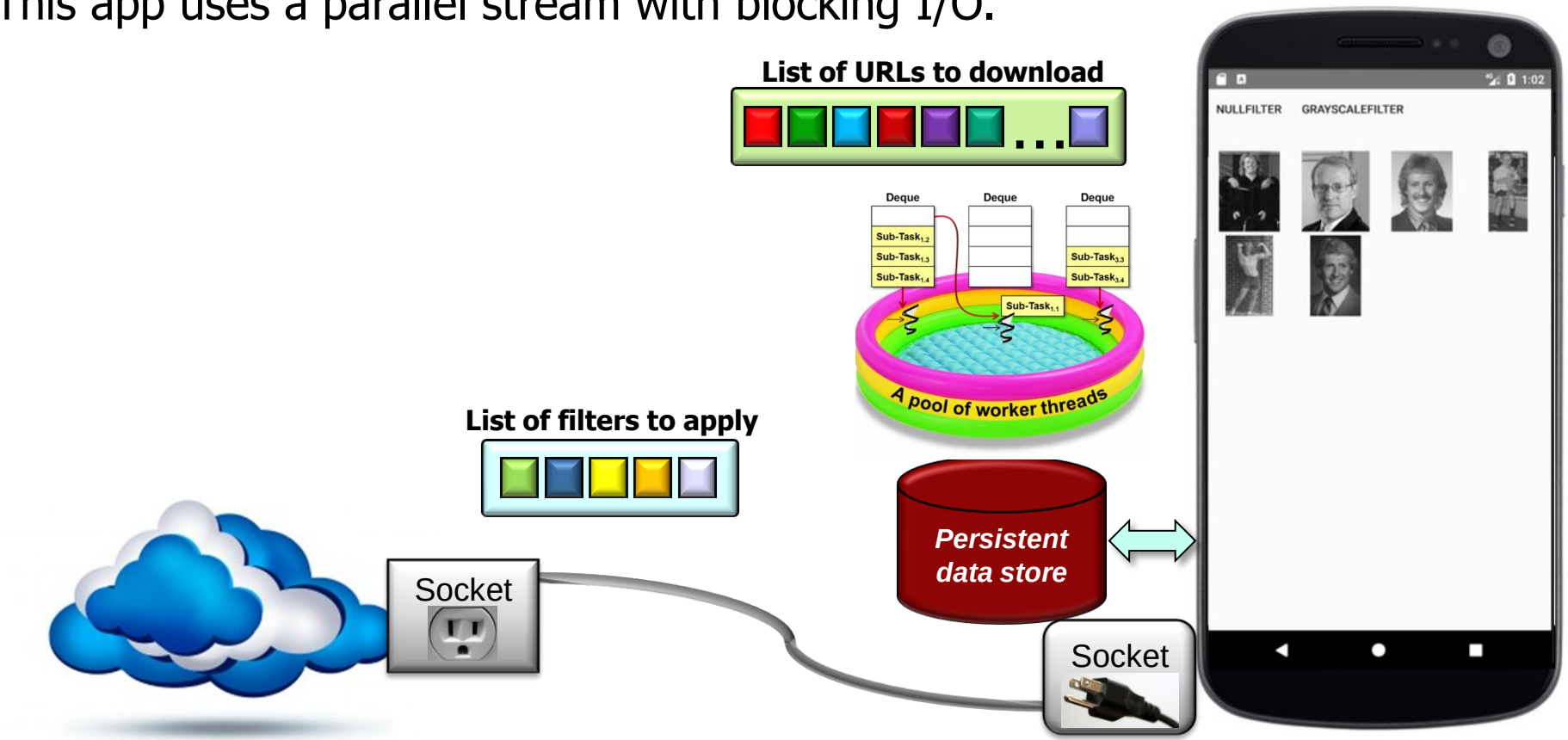


See github.com/douglascraigschmidt/LiveLessons/tree/master/ImageTaskGangApplication

Overview of Parallel Streams in ImageStreamGang

Overview of Parallel Streams in ImageStreamGang

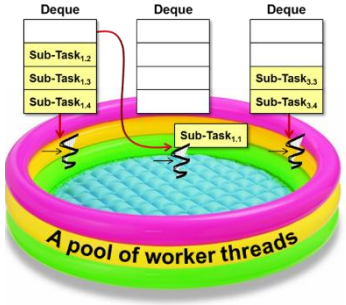
- This app uses a parallel stream with blocking I/O.



See github.com/douglascraigschmidt/LiveLessons/blob/master/ImageStreamGang/AndroidGUI

Overview of Parallel Streams in ImageStreamGang

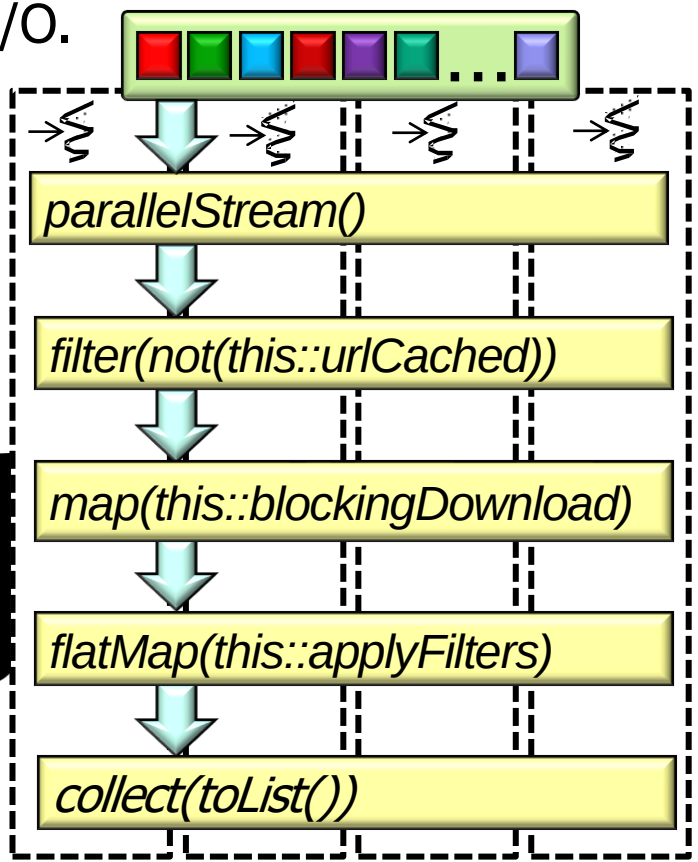
- This app uses a parallel stream with blocking I/O.
  - Ignore cached images
  - Download non-cached images
  - Apply list of filters to each image
  - Store filtered images in the file system
  - Display images to the user



Combines Java object-oriented and functional programming features

Overview of Parallel Streams in ImageStreamGang

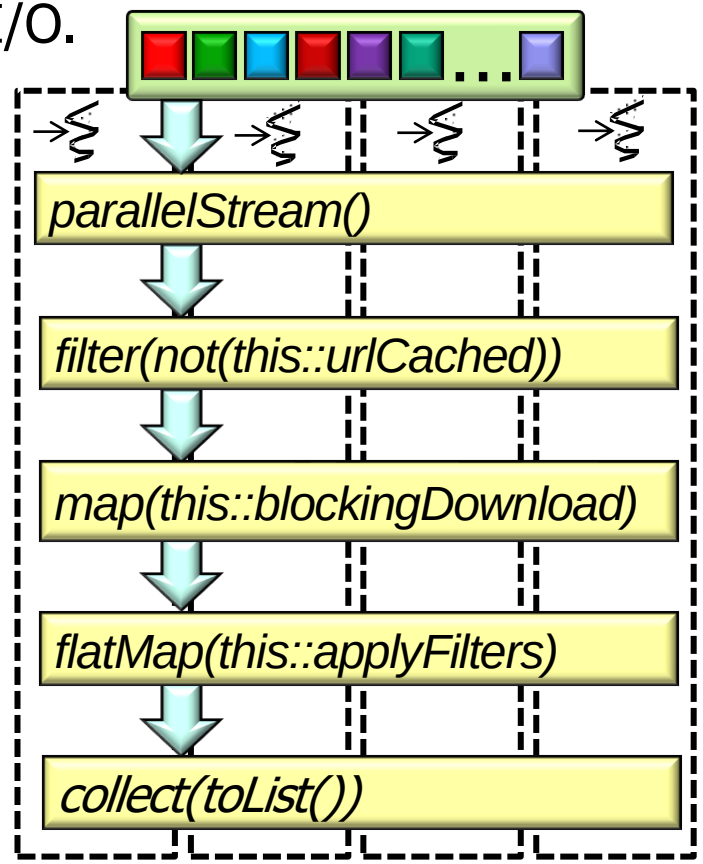
- This app uses a parallel stream with blocking I/O.
  - Ignore cached images
  - Download non-cached images
  - Apply list of filters to each image
  - Store filtered images in the file system
  - Display images to the user



Declarative stream pipeline closely aligns with the app description.

Overview of Parallel Streams in ImageStreamGang

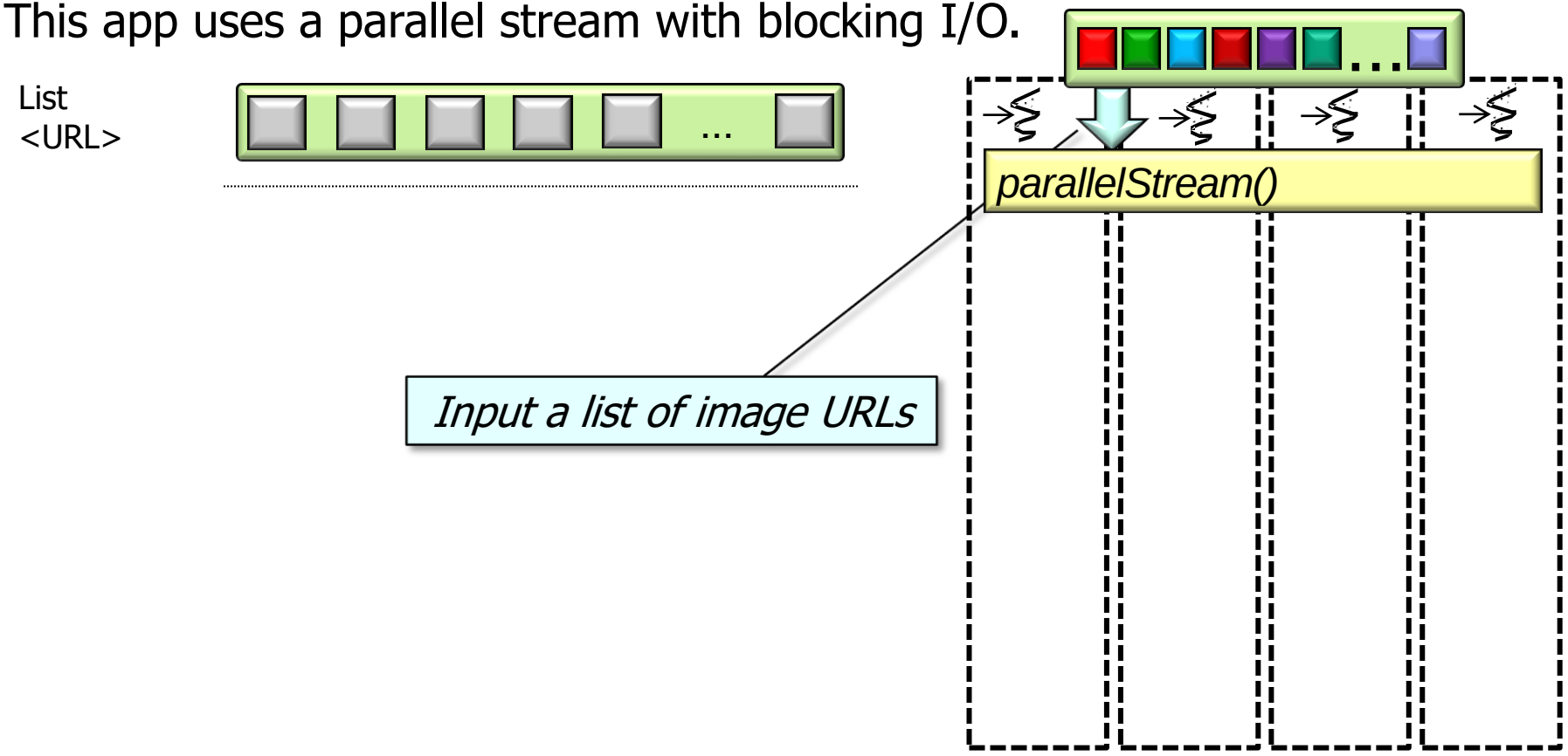
- This app uses a parallel stream with blocking I/O.



Closes gap between domain intent and computations that implement the intent

Overview of Parallel Streams in ImageStreamGang

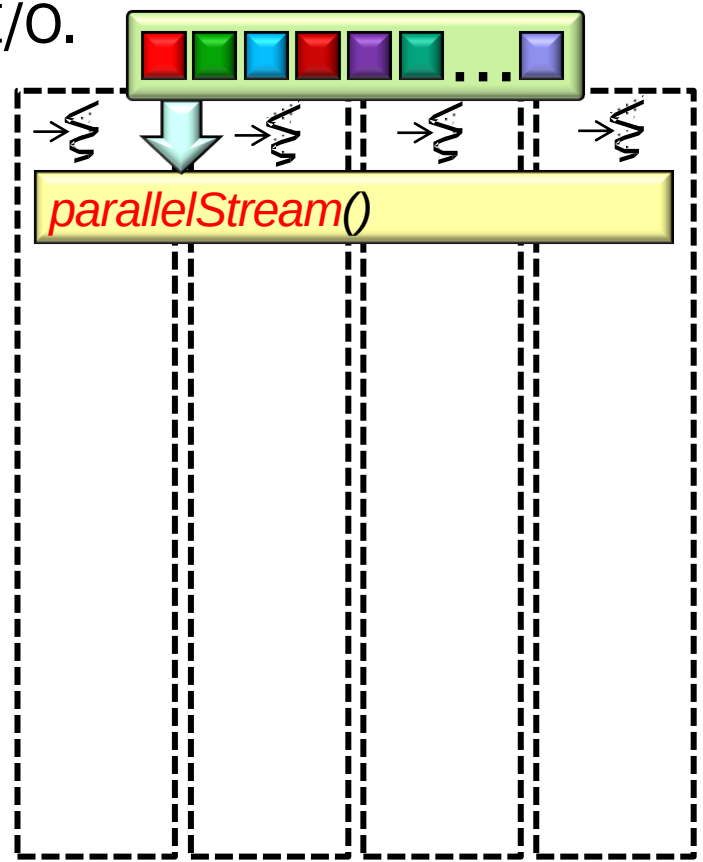
- This app uses a parallel stream with blocking I/O.



Overview of Parallel Streams in ImageStreamGang

- This app uses a parallel stream with blocking I/O.

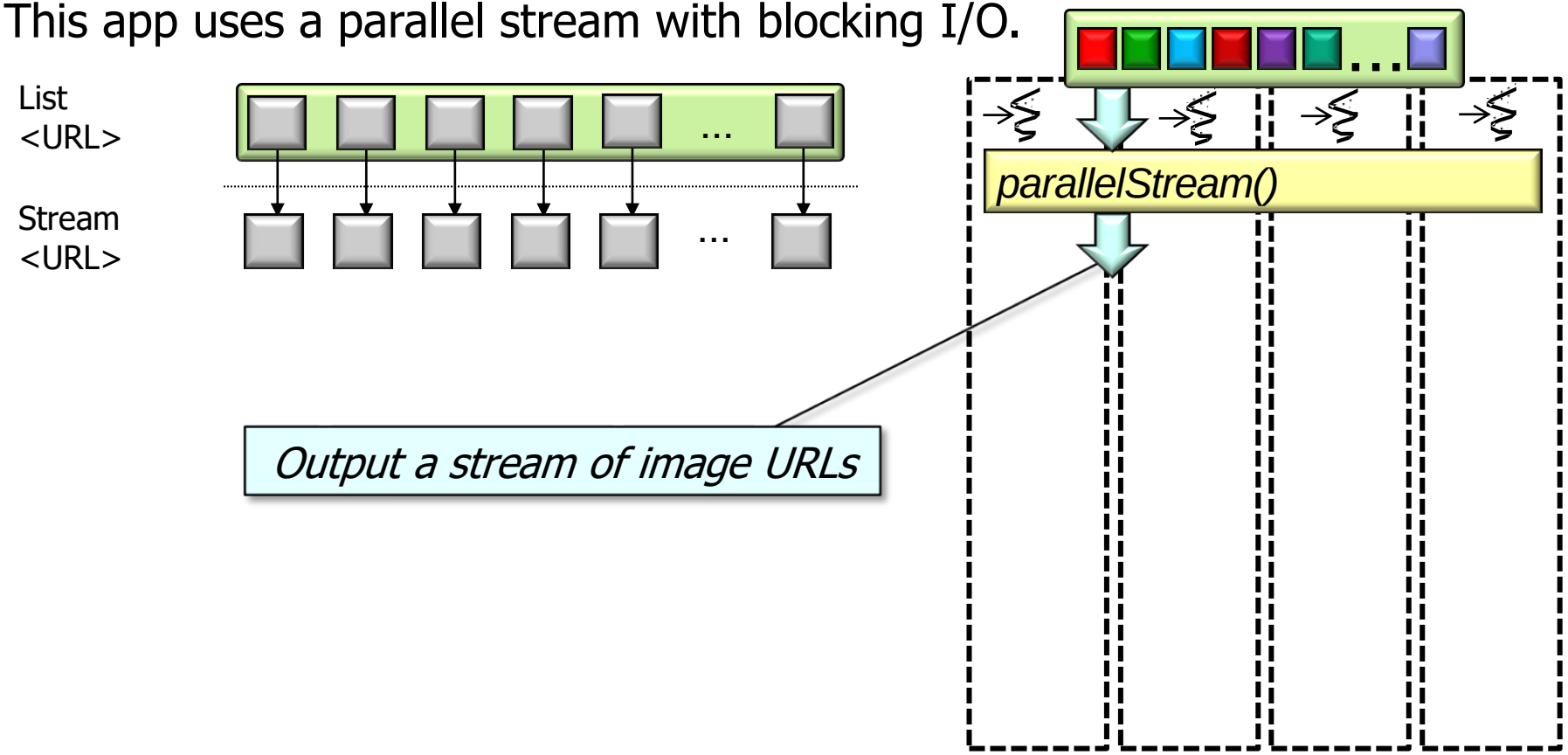
List
<URL>



Convert collection to a parallel stream

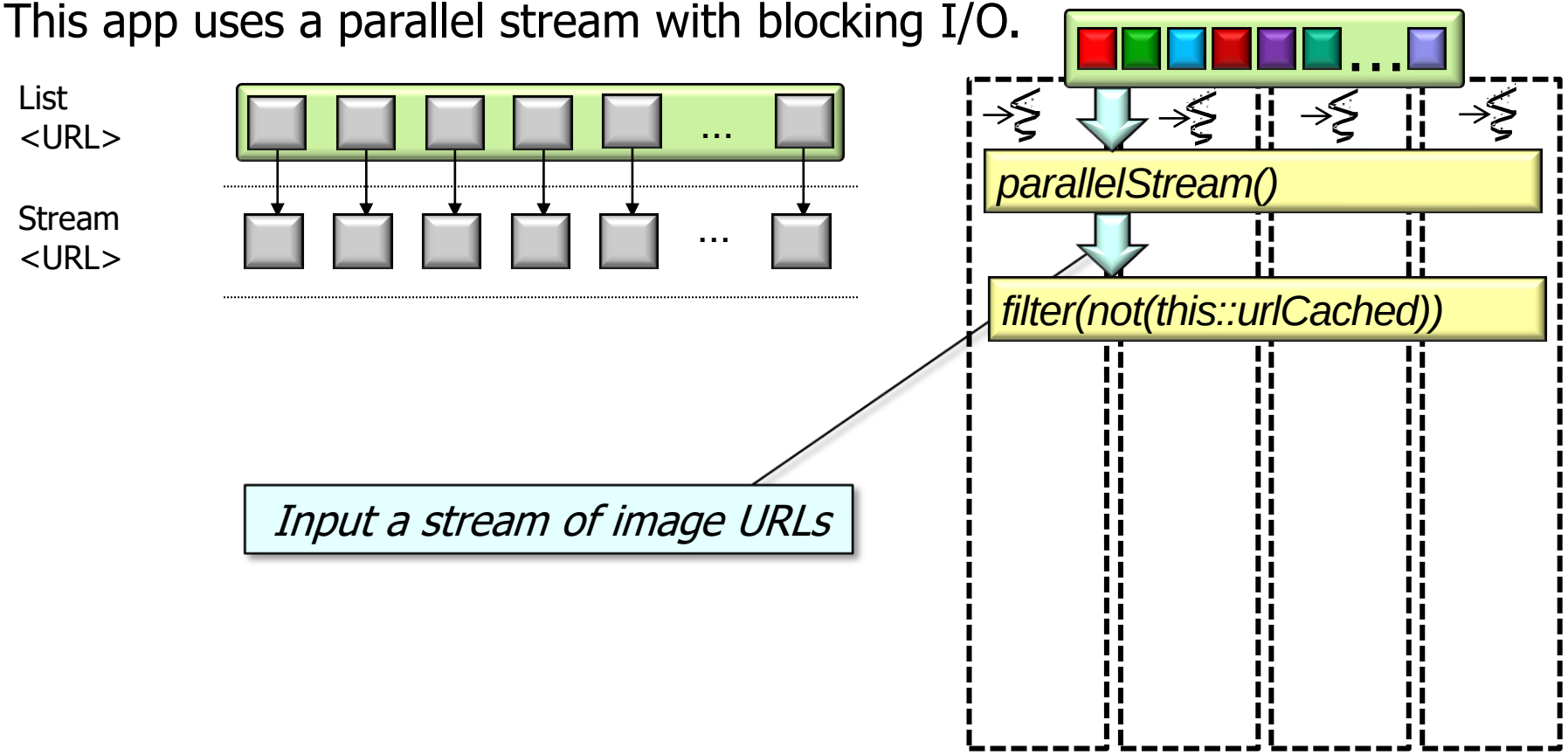
Overview of Parallel Streams in ImageStreamGang

- This app uses a parallel stream with blocking I/O.



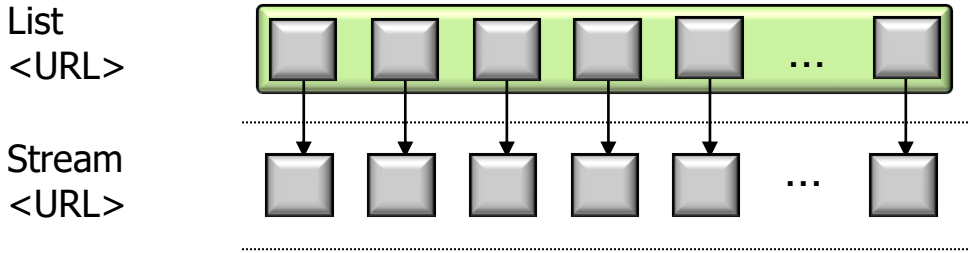
Overview of Parallel Streams in ImageStreamGang

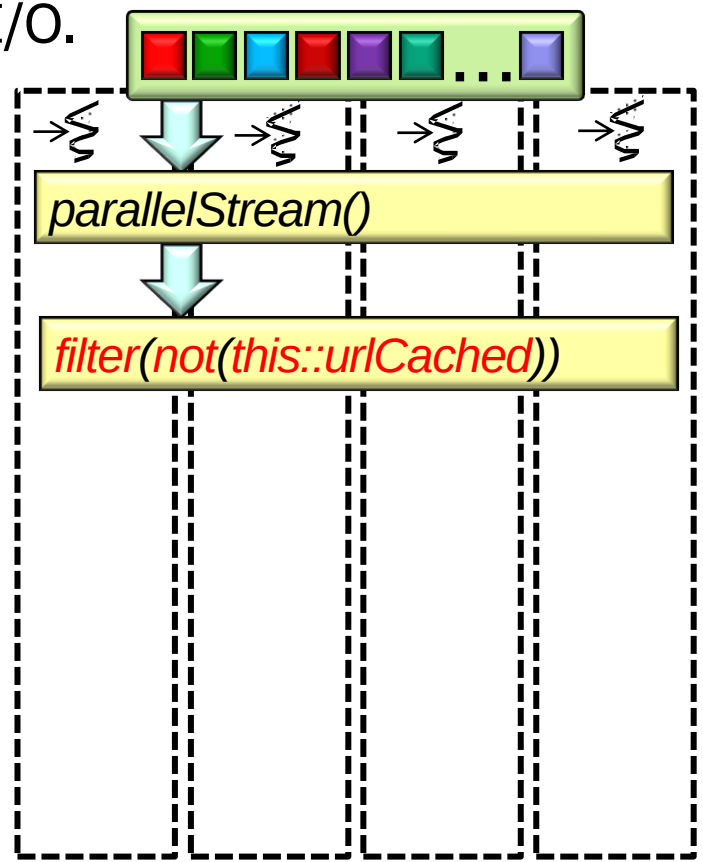
- This app uses a parallel stream with blocking I/O.



Overview of Parallel Streams in ImageStreamGang

- This app uses a parallel stream with blocking I/O.

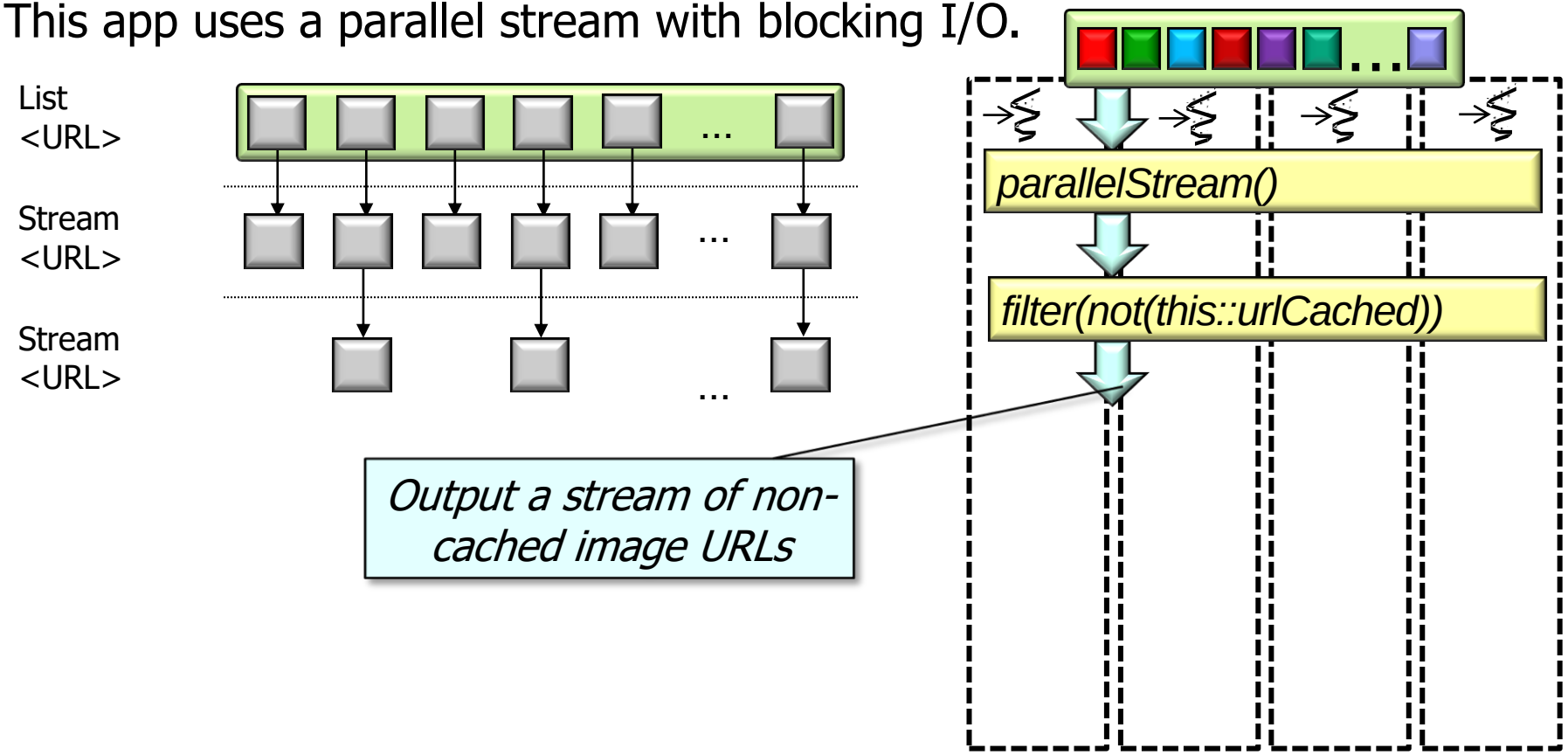




Filter() ignores cached images.

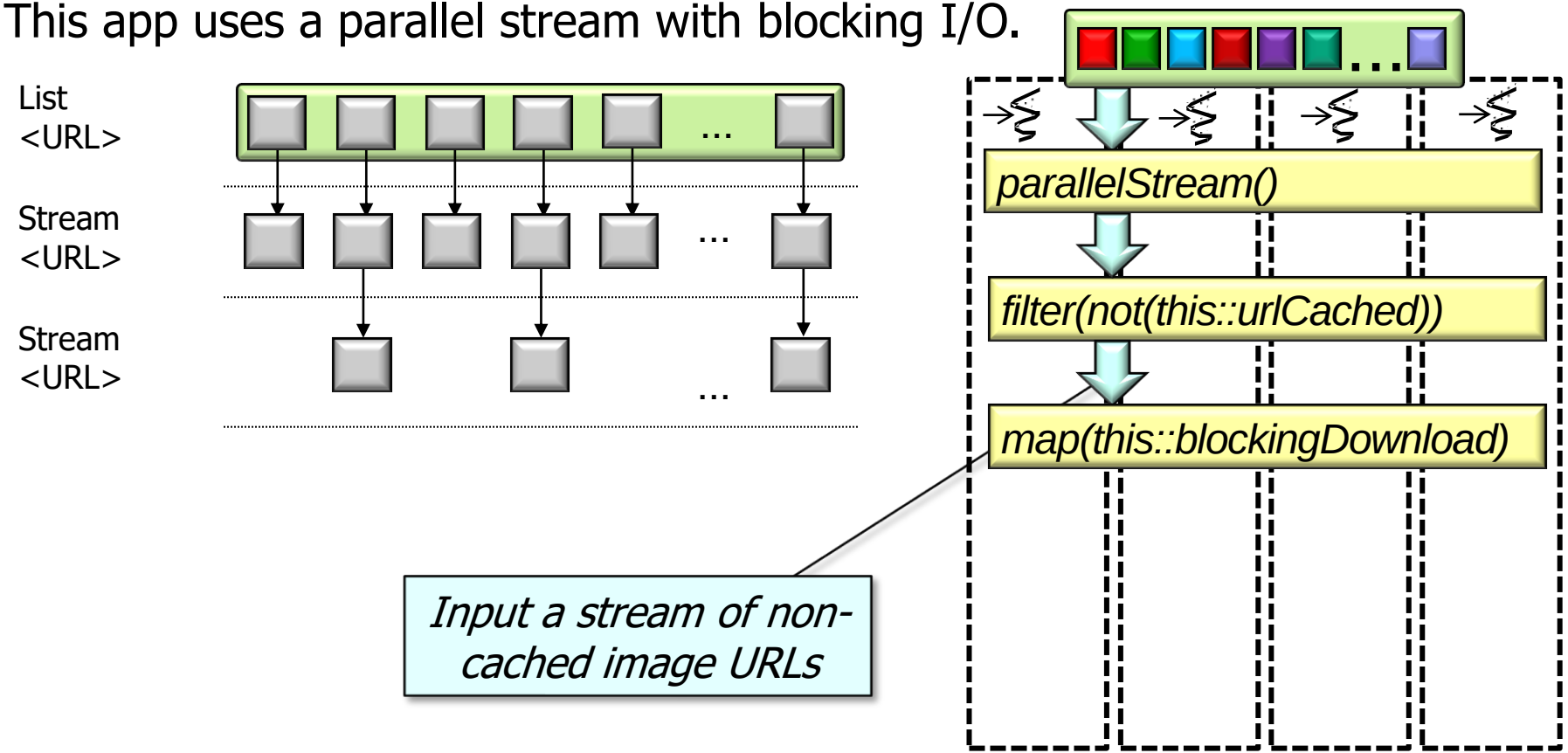
Overview of Parallel Streams in ImageStreamGang

- This app uses a parallel stream with blocking I/O.



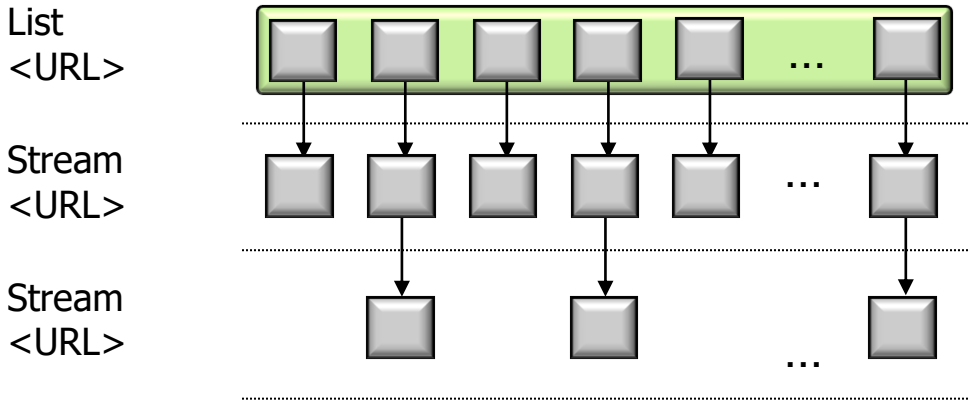
Overview of Parallel Streams in ImageStreamGang

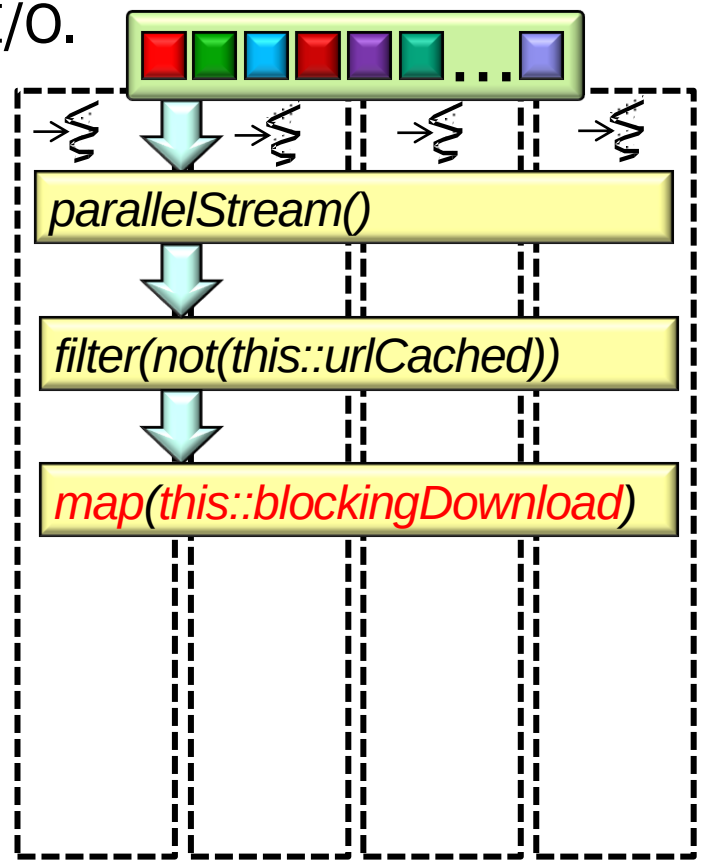
- This app uses a parallel stream with blocking I/O.



Overview of Parallel Streams in ImageStreamGang

- This app uses a parallel stream with blocking I/O.

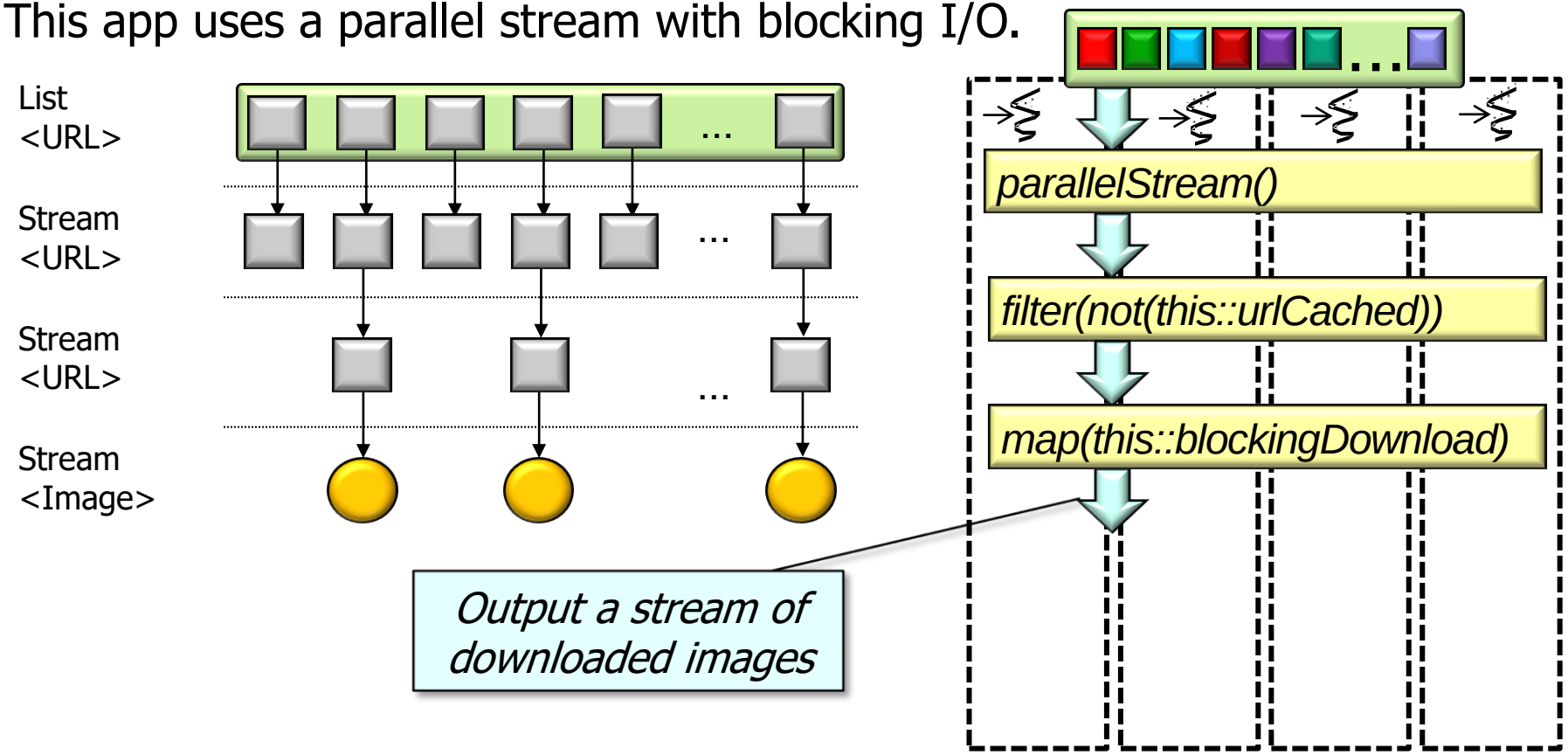




Download non-cached images in parallel

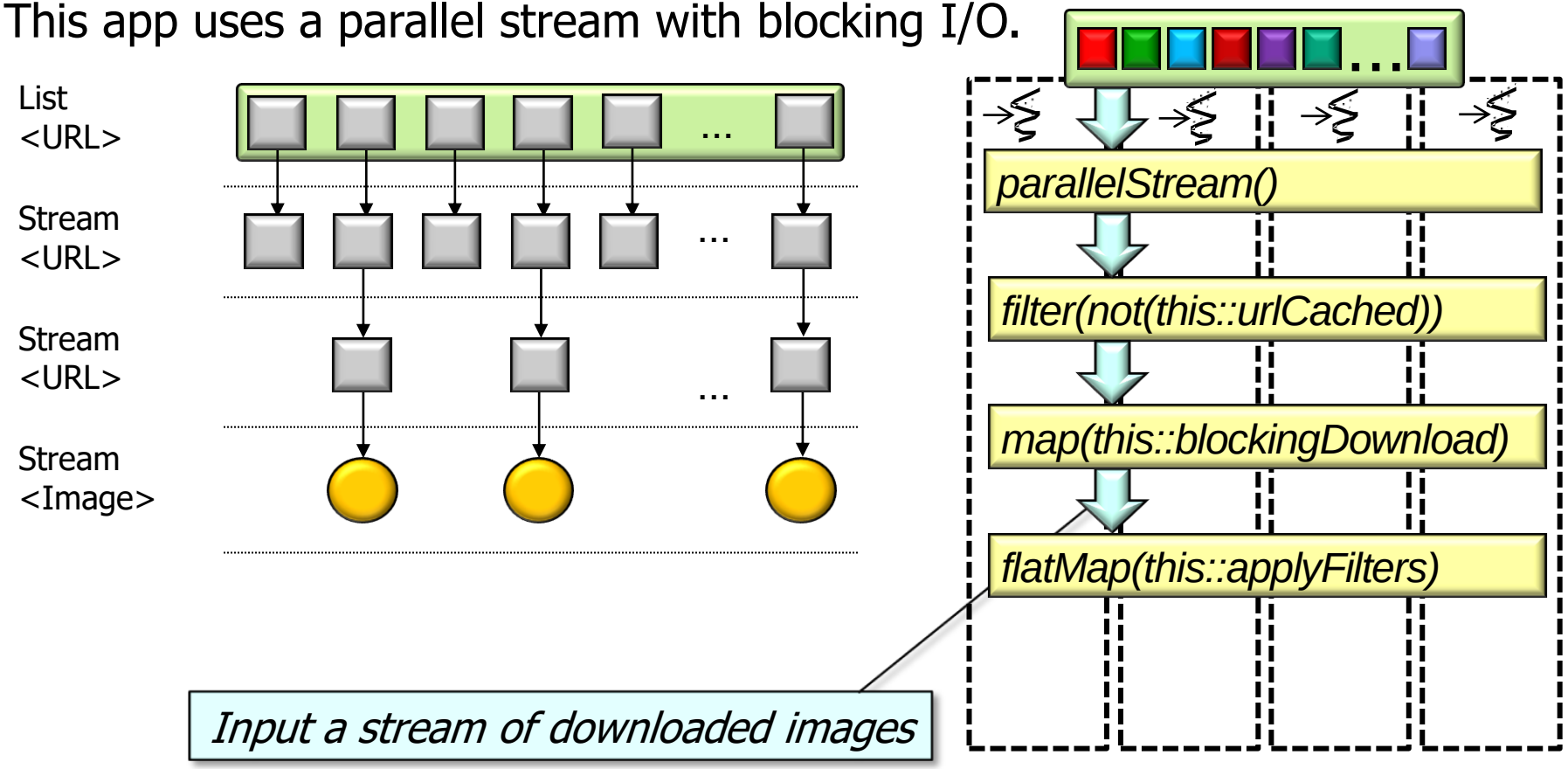
Overview of Parallel Streams in ImageStreamGang

- This app uses a parallel stream with blocking I/O.



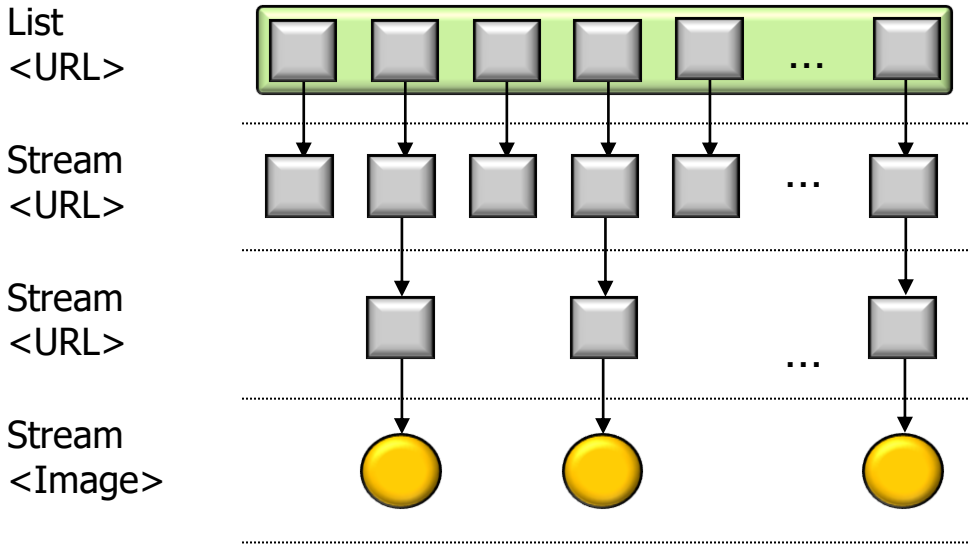
Overview of Parallel Streams in ImageStreamGang

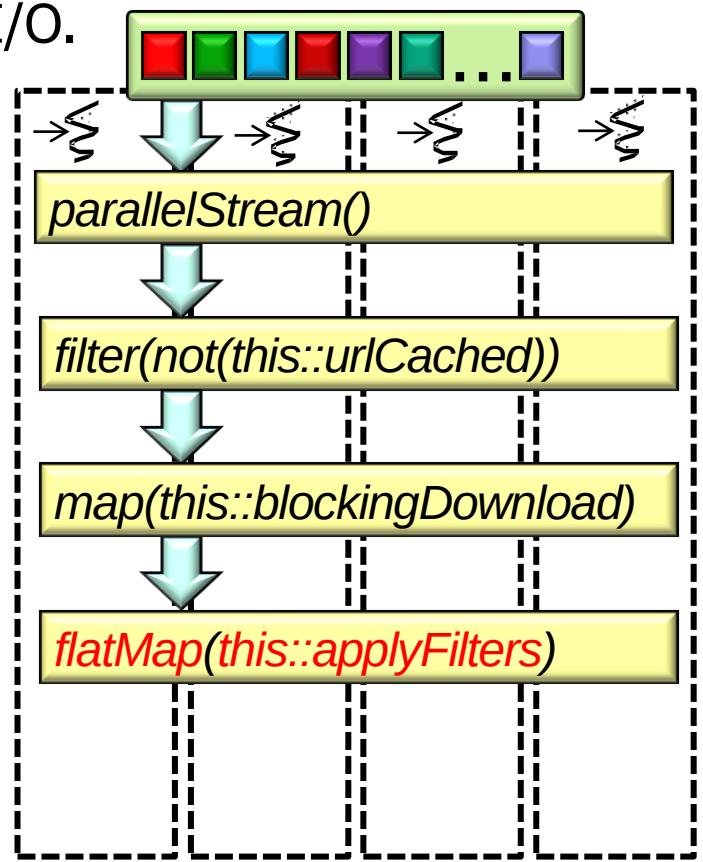
- This app uses a parallel stream with blocking I/O.



Overview of Parallel Streams in ImageStreamGang

- This app uses a parallel stream with blocking I/O.

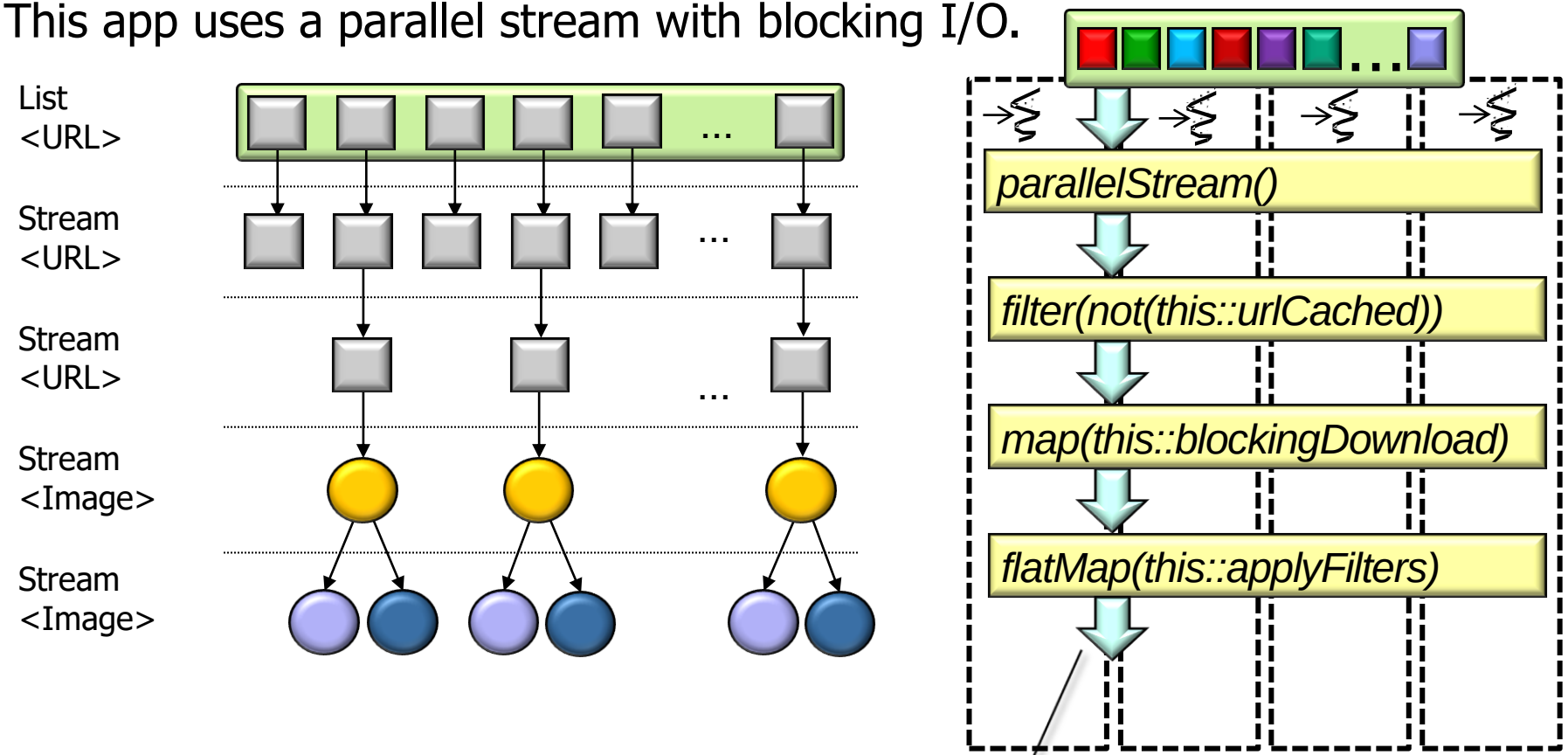




Apply filters to each image in parallel and store filtered images in file system

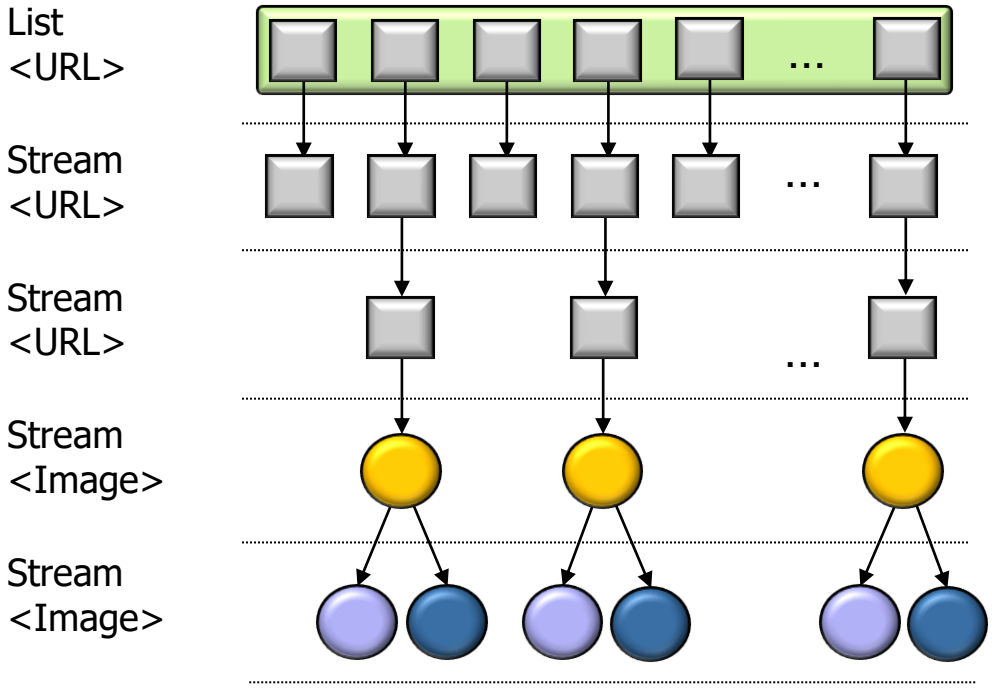
Overview of Parallel Streams in ImageStreamGang

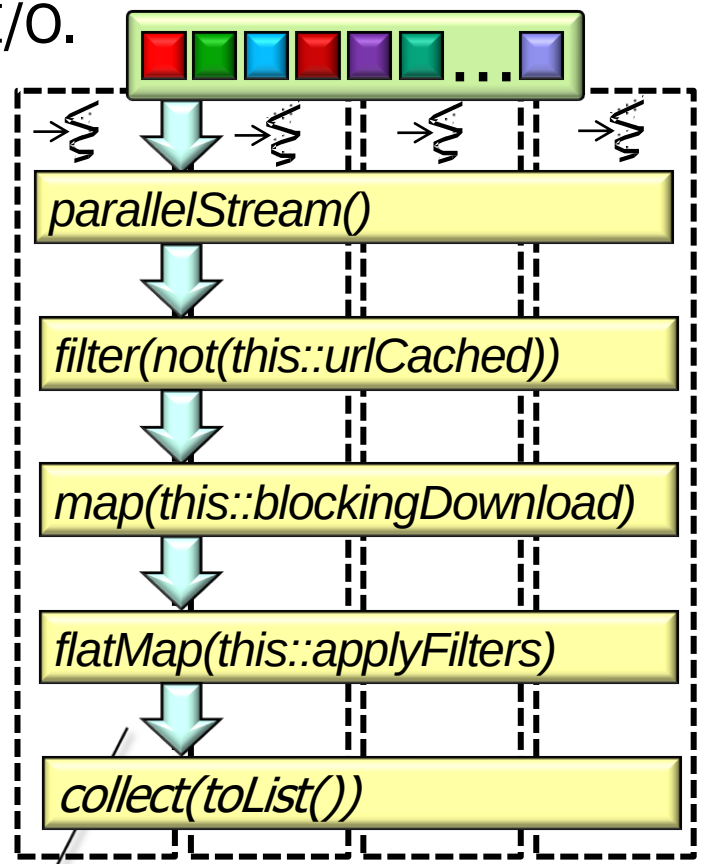
- This app uses a parallel stream with blocking I/O.



Overview of Parallel Streams in ImageStreamGang

- This app uses a parallel stream with blocking I/O.

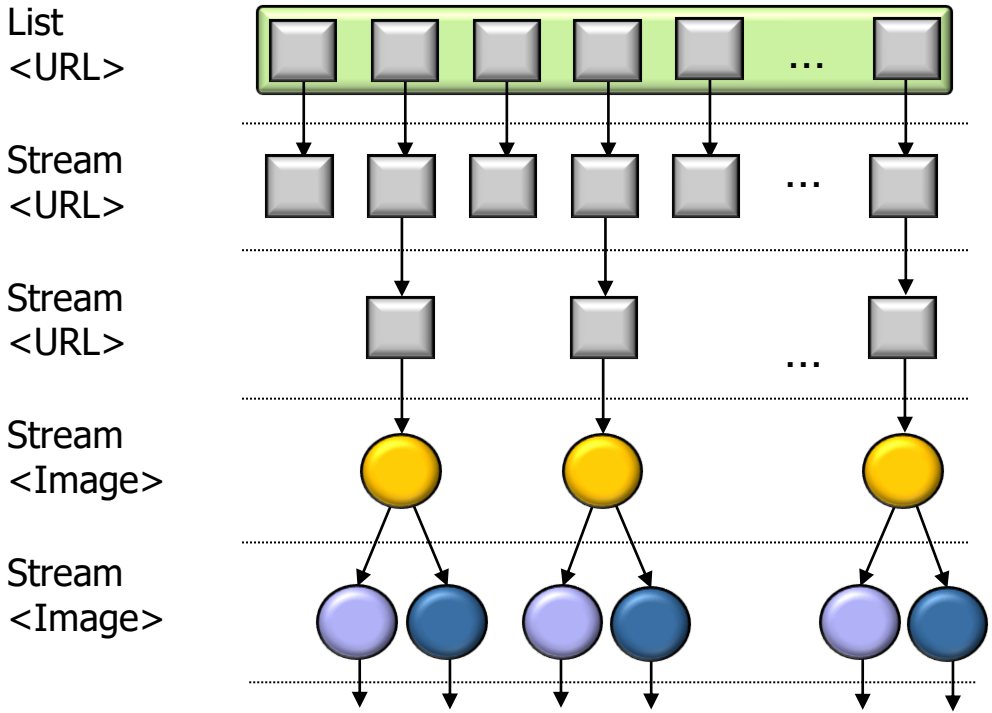


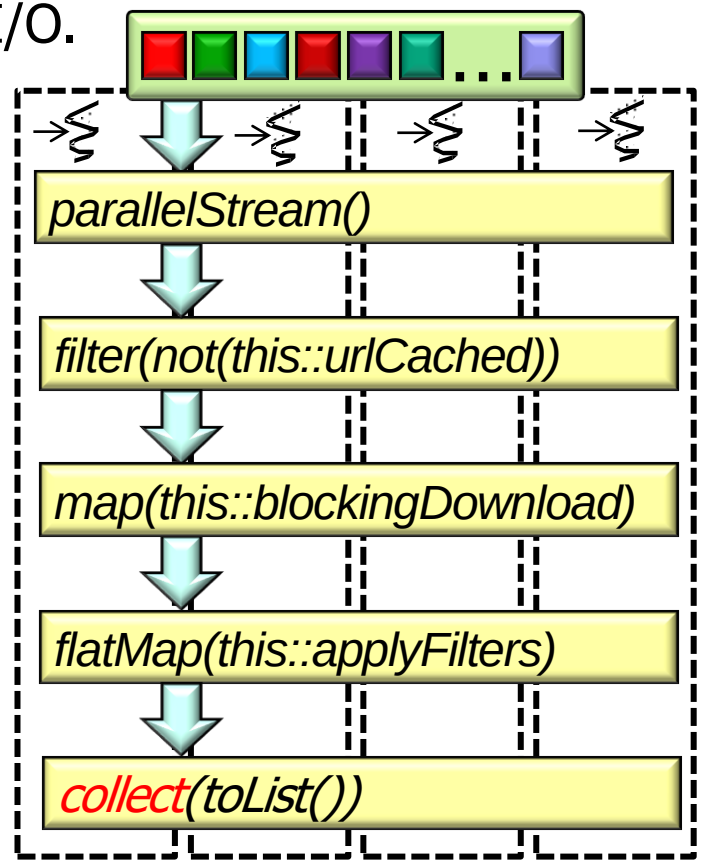


Input a stream of filtered and stored images

Overview of Parallel Streams in ImageStreamGang

- This app uses a parallel stream with blocking I/O.

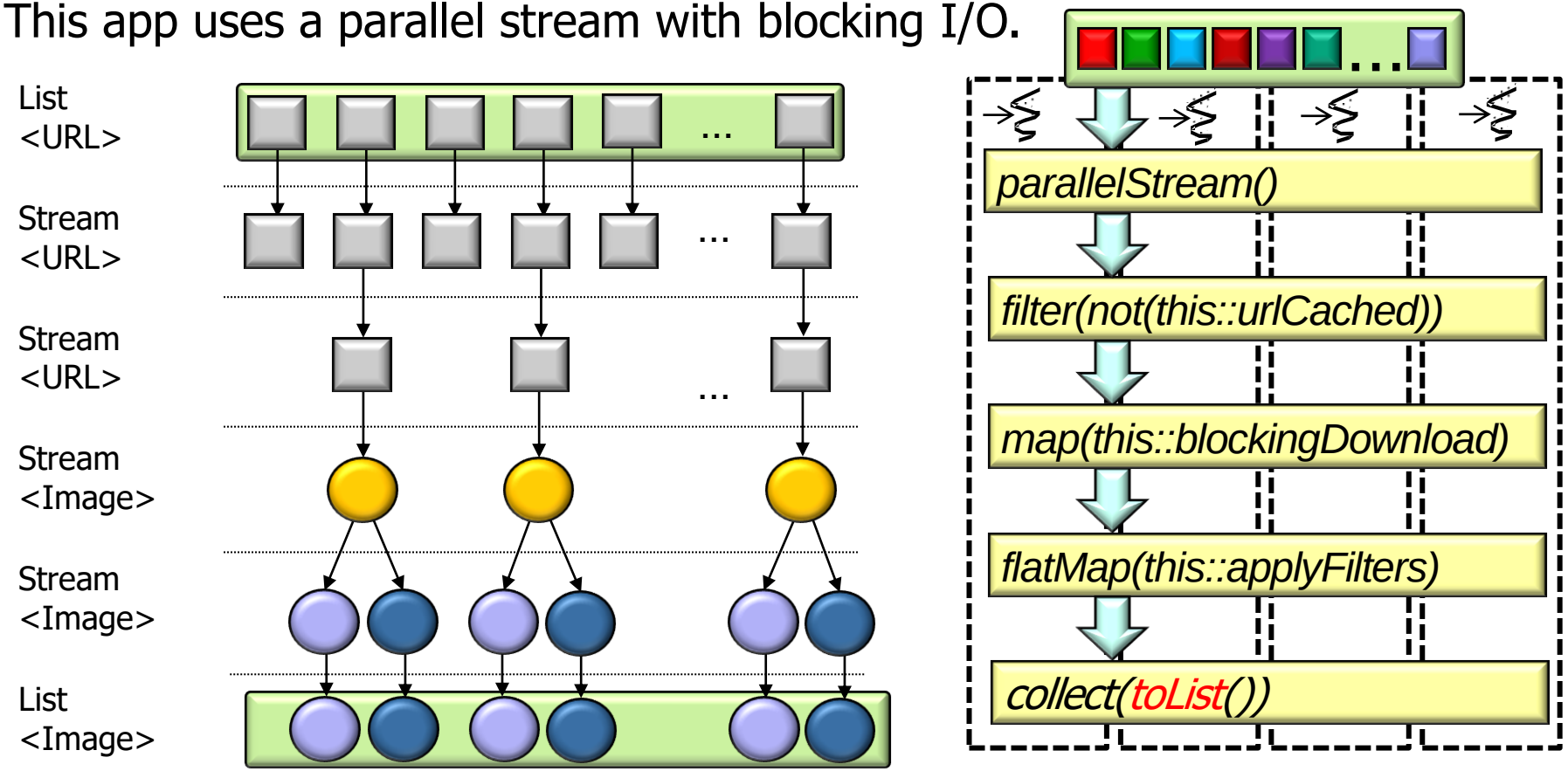




Trigger intermediate operation processing

Overview of Parallel Streams in ImageStreamGang

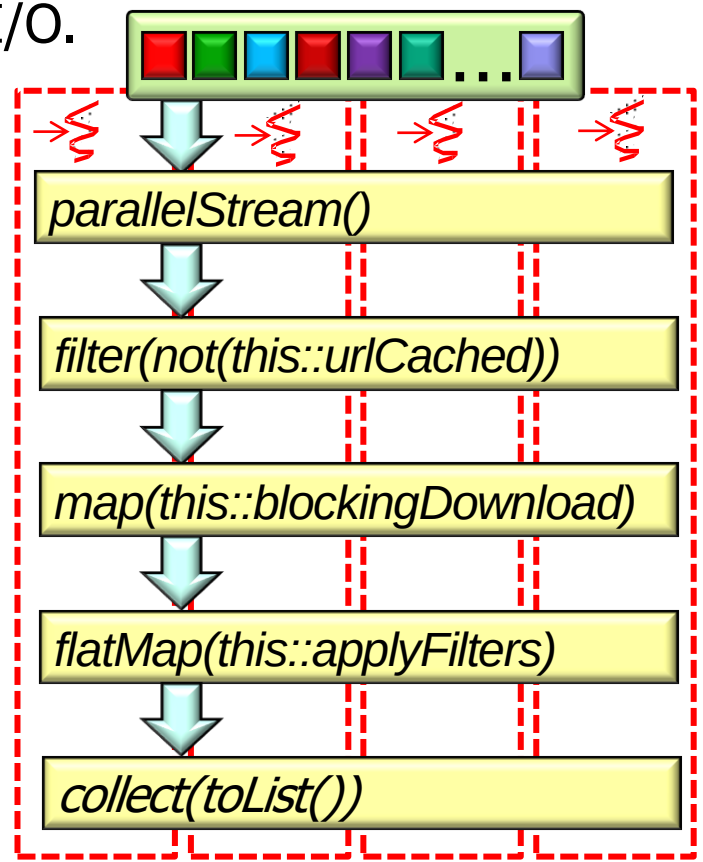
- This app uses a parallel stream with blocking I/O.



Return a list of filtered and stored images

Overview of Parallel Streams in ImageStreamGang

- This app uses a parallel stream with blocking I/O.
  - Ignore cached images
  - Download non-cached images
  - Apply list of filters to each image
  - Store filtered images in the file system
  - Display images to the user (after triggering stream processing)



The Java streams framework orchestrates all these steps in parallel.

The End

Java Parallel ImageStreamGang Example

Implementing Behaviors

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Recognize the structure/functionality of the ImageStreamGang app.
- Know how Java parallel streams are applied to the ImageStreamGang app.
- Understand the parallel streams implementation of ImageStreamGang.

```
void processStream() {
  List<URL> urls = getInput();

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

  System.out.println(TAG
        + "Image(s) filtered = "
        + filteredImages.size());
}
```

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

Implementing a Parallel Stream in ImageStreamGang

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java.

```
void processStream() {
  List<URL> urls = getInput();

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

  System.out.println(TAG
        + "Image(s) filtered = "
        + filteredImages.size());
}
```

See imagestreamgang/streams/ImageStreamParallel.java

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java.

Get a list of URLs

```
void processStream() {
  List<URL> urls = getInput();

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

  System.out.println(TAG
        + "Image(s) filtered = "
        + filteredImages.size());
}
```

GetInput() is defined by the underlying StreamGang framework.

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java.

Convert a collection into a parallel stream

```
void processStream() {
  List<URL> urls = getInput();

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

  System.out.println(TAG
        + "Image(s) filtered = "
        + filteredImages.size());
}
```

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java.

Return an output stream consisting of the URLs in the input stream that are not already cached

```
void processStream() {
  List<URL> urls = getInput();

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

  System.out.println(TAG
        + "Image(s) filtered = "
        + filteredImages.size());
}
```

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

Implementing a Parallel Stream in ImageStreamGang

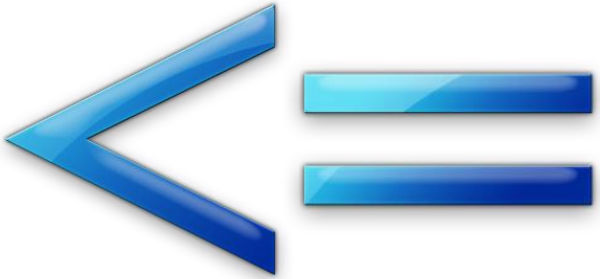
- We focus on processStream() in ImageStreamParallel.java.

```
void processStream() {
  List<URL> urls = getInput();

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

  System.out.println(TAG
        + "Image(s) filtered = "
        + filteredImages.size());
}
```

Return an output stream consisting of the URLs in the input stream that are not already cached



# of output stream elements will be <= # of input stream elements

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java.

```
void processStream() {
  List<URL> urls = getInput();

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

  System.out.println(TAG
        + "Image(s) filtered = "
        + filteredImages.size());
}
```

```
boolean urlCached(URL url) {
  return mFilters
    .stream()
    .filter(filter ->
       urlCached(url,
          filter.getName()))
    .count() > 0;
}
```

Determine whether this URL has been downloaded to an image and had filters applied to it yet

See imagestreamgang/streams/ImageStreamGang.java

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java.

```
void processStream() {
  List<URL> urls = getInput();

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

  System.out.println(TAG
        + "Image(s) filtered = "
        + filteredImages.size());
}
```

```
boolean urlCached(URL url,
          String filterName) {
  File file =
    new File(getPath(),
             filterName);

  File imageFile =
    new File(file,
          getNameForUrl(url));

  return imageFile.exists();
}
```

Check if a file with this name already exists

See imagestreamgang/streams/ImageStreamGang.java

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java.


```
void processStream() {
  List<URL> urls = getInput();

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

  System.out.println(TAG
        + "Image(s) filtered = "
        + filteredImages.size());
}
```

There are clearly better ways of implementing an image cache!

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java.

Return an output stream consisting of the images that were downloaded from the URLs in the input stream

```
void processStream() {
  List<URL> urls = getInput();

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

  System.out.println(TAG
        + "Image(s) filtered = "
        + filteredImages.size());
}
```

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

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java.

```
void processStream() {
  List<URL> urls = getInput();

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

  System.out.println(TAG
        + "Image(s) filtered = "
        + filteredImages.size());
}
```

Return an output stream consisting of the images that were downloaded from the URLs in the input stream

# of output stream elements must match the # of input stream elements

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java.

```
void processStream() {
  List<URL> urls = getInput();

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

  System.out.println(TAG
        + "Image(s) filtered = "
        + filteredImages.size());
}
```

```
Image blockingDownload
                (URL url) {
  return BlockingTask
   .callInManagedBlock
     (() ->
      downloadImage(url));
}
```

Downloads content from a URL and converts it into an image

See imagestreamgang/streams/ImageStreamParallel.java

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java.

```
void processStream() {
  List<URL> urls = getInput();

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

  System.out.println(TAG
        + "Image(s) filtered = "
        + filteredImages.size());
}
```

```
Image blockingDownload
                (URL url) {
  return BlockingTask
   .callInManagedBlock
     (() ->
      downloadImage(url));
}
```

Uses a "managed blocker" to ensure sufficient threads are in the common ForkJoinPool

See lesson on "*The Java ForkJoinPool: Applying the ManagedBlocker Interface.*"

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java.

```
void processStream() {
  List<URL> urls = getInput();

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

  System.out.println(TAG
        + "Image(s) filtered = "
        + filteredImages.size());
}
```

```
Image blockingDownload
                (URL url) {
  return BlockingTask
   .callInManagedBlock
     (() ->
      downloadImage(url));
}
```

I/O-bound tasks on an N-core CPU typically run best with N(1+WT/ST) threads (WT = wait time & ST = service time).*

See www.ibm.com/developerworks/library/j-jtp0730

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java.

```
void processStream() {
  List<URL> urls = getInput();

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

  System.out.println(TAG
        + "Image(s) filtered = "
        + filteredImages.size());
}
```

Return an output stream containing the results of applying a list of filters to each image in the input stream and storing the results in the file system

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

Implementing a Parallel Stream in ImageStreamGang

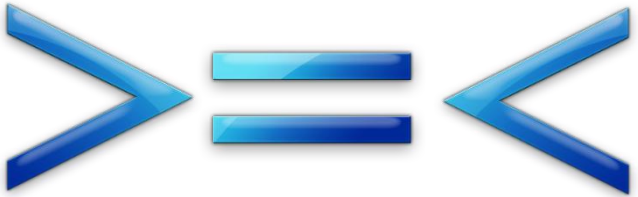
- We focus on processStream() in ImageStreamParallel.java.

```
void processStream() {
  List<URL> urls = getInput();

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

  System.out.println(TAG
        + "Image(s) filtered = "
        + filteredImages.size());
}
```

Return an output stream containing the results of applying a list of filters to each image in the input stream and storing the results in the file system



# of output stream elements may differ from the # of input stream elements

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java.

```
void processStream() {
  List<URL> urls = getInput();

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

System.out.println(TAG
       + "Image(s) filtered = "
       + filteredImages.size());
}
```

```
Stream<Image> applyFilters
                    (Image image) {
  return mFilters
    .parallelStream()
    .map(filter ->
         makeFilterWithImage
          (filter,
           image).run())
}
```

Apply all filters to an image in parallel and store on the device

See imagestreamgang/streams/ImageStreamParallel.java

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java.

Collect() is a "reduction" operation that combines elements into one result.

```
void processStream() {
  List<URL> urls = getInput();

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

  System.out.println(TAG
        + "Image(s) filtered = "
        + filteredImages.size());
}
```

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

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java.

Trigger all intermediate operations

```
void processStream() {
  List<URL> urls = getInput();

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

  System.out.println(TAG
        + "Image(s) filtered = "
        + filteredImages.size());
}
```

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java.

```
void processStream() {
  List<URL> urls = getInput();

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

  System.out.println(TAG
        + "Image(s) filtered = "
        + filteredImages.size());
}
```

Create a list containing all the filtered and stored images

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java.

```
void processStream() {
  List<URL> urls = getInput();

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

  System.out.println(TAG
          + "Image(s) filtered = "
          + filteredImages.size());
}
```

Logs the # of images that were downloaded, filtered, and stored.

The End

Java Parallel ImageStreamGang Example

Evaluating Pros and Cons

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Recognize the structure/functionality of the ImageStreamGang app.
- Know how Java parallel streams are applied to the ImageStreamGang app.
- Understand the parallel streams implementation of ImageStreamGang.
- Be aware of the pros and cons of the parallel streams solution.

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

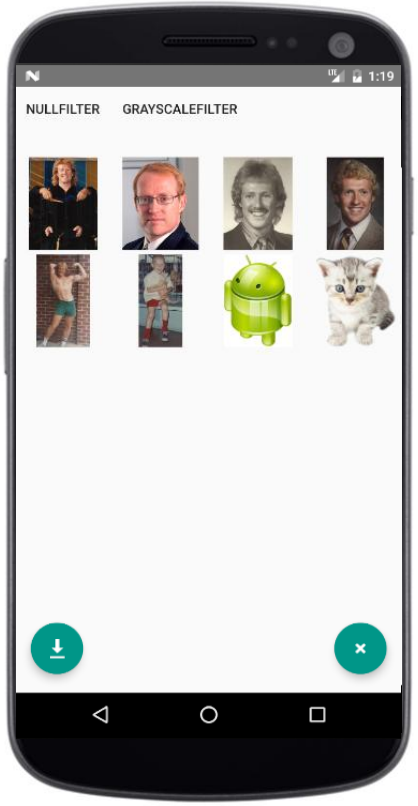
Pros of the Java Parallel Streams Solution

Pros of the Java Parallel Streams Solution

- The parallel stream version is faster than the sequential streams version.

Starting ImageStreamGangTest
Printing 4 results for input file 1 from fastest to slowest
COMPLETABLE_FUTURES_1 executed in 312 msecs
COMPLETABLE_FUTURES_2 executed in 335 msecs
PARALLEL_STREAM executed in 428 msecs
SEQUENTIAL_STREAM executed in 981 msecs

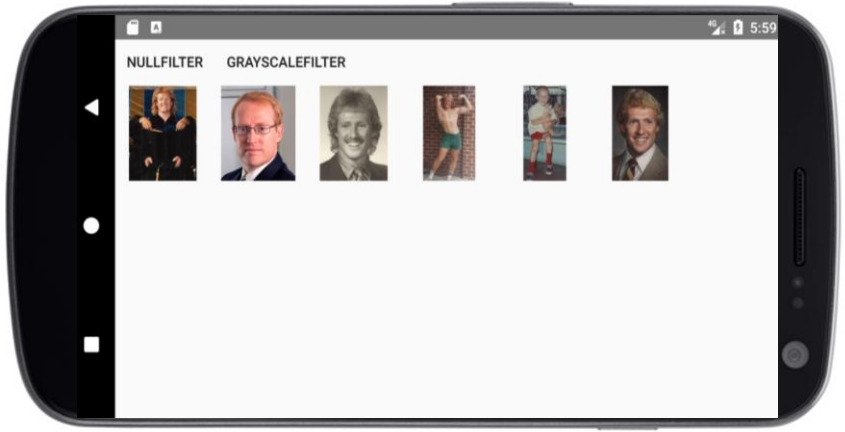
Printing 4 results for input file 2 from fastest to slowest
COMPLETABLE_FUTURES_2 executed in 82 msecs
COMPLETABLE_FUTURES_1 executed in 83 msecs
PARALLEL_STREAM executed in 102 msecs
SEQUENTIAL_STREAM executed in 251 msecs
Ending ImageStreamGangTest

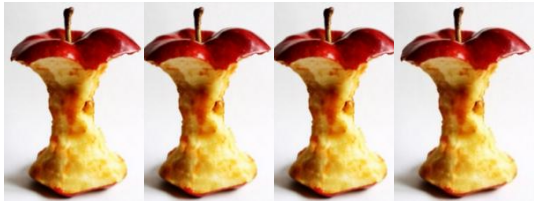


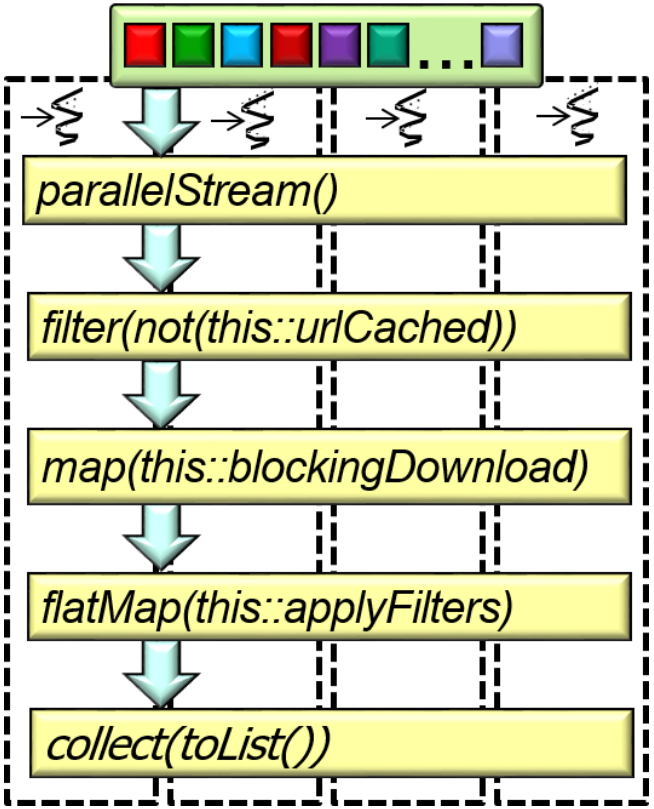
Six-core 2.6 GHz Windows Intel computer with 64 GB RAM

Pros of the Java Parallel Streams Solution

- The parallel stream version is faster than the sequential streams version.
  - e.g., images are downloaded and processed in parallel on multiple cores.







Pros of the Java Parallel Streams Solution

- The solution is relatively straightforward to understand.


```
void processStream() {
  List<URL> urls = getInput();

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

  System.out.println(TAG
        + "Image(s) filtered = "
        + filteredImages.size());
}
```

Pros of the Java Parallel Streams Solution

- The solution is relatively straightforward to understand, e.g.
  - The behaviors map cleanly onto the domain intent.

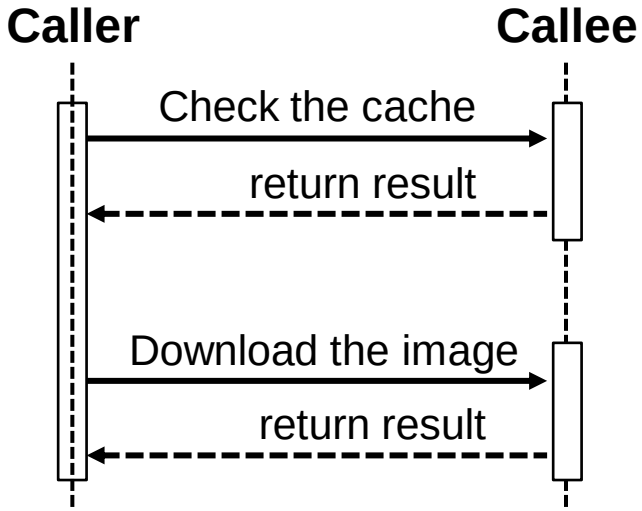

```
void processStream() {
  List<URL> urls = getInput();

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

  System.out.println(TAG
        + "Image(s) filtered = "
        + filteredImages.size());
}
```

Pros of the Java Parallel Streams Solution

- The solution is relatively straightforward to understand, e.g.
  - The behaviors map cleanly onto the domain intent.
  - Behaviors are all synchronous.



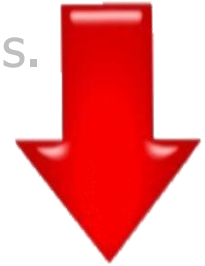

```
void processStream() {
  List<URL> urls = getInput();

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

  System.out.println(TAG
        + "Image(s) filtered = "
        + filteredImages.size());
}
```

Pros of the Java Parallel Streams Solution

- The solution is relatively straightforward to understand, e.g.
  - The behaviors map cleanly onto the domain intent.
  - Behaviors are all synchronous.
  - The flow of control can be read "linearly."



```
void processStream() {
  List<URL> urls = getInput();

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

  System.out.println(TAG
        + "Image(s) filtered = "
        + filteredImages.size());
}
```

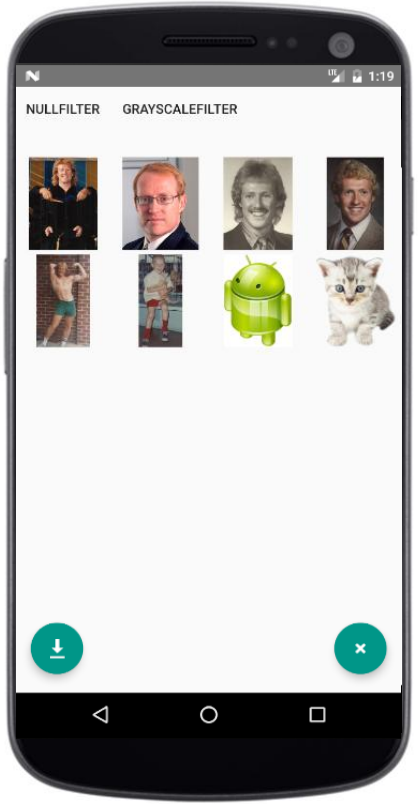
Cons of the Java Parallel Streams Solution

Cons of the Java Parallel Streams Solution

- The CompletableFutures versions are faster than the parallel streams version.

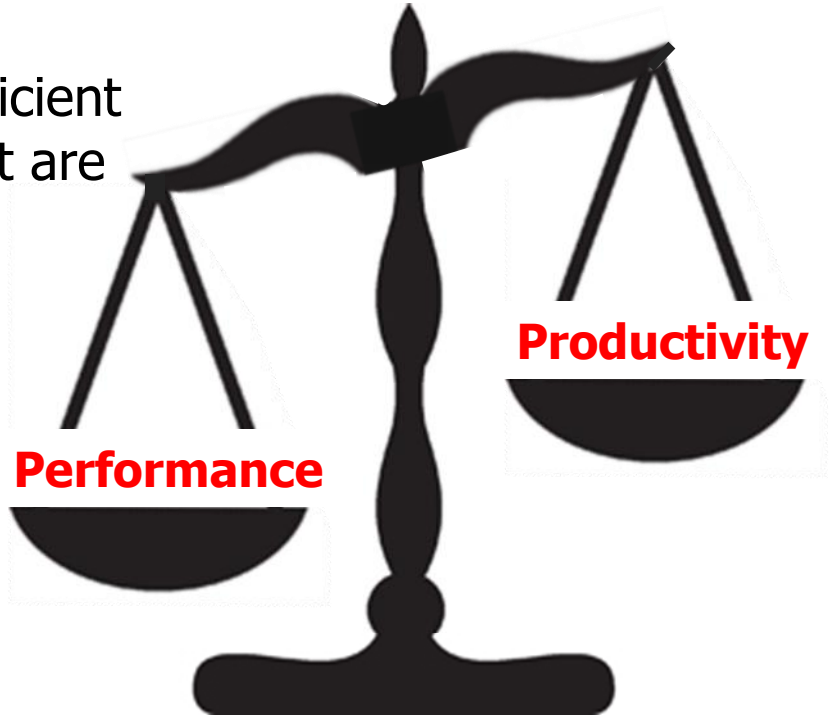
Starting ImageStreamGangTest
Printing 4 results for input file 1 from fastest to slowest
COMPLETABLE_FUTURES_1 executed in 312 msecs
COMPLETABLE_FUTURES_2 executed in 335 msecs
PARALLEL_STREAM executed in 428 msecs
SEQUENTIAL_STREAM executed in 981 msecs

Printing 4 results for input file 2 from fastest to slowest
COMPLETABLE_FUTURES_2 executed in 82 msecs
COMPLETABLE_FUTURES_1 executed in 83 msecs
PARALLEL_STREAM executed in 102 msecs
SEQUENTIAL_STREAM executed in 251 msecs
Ending ImageStreamGangTest



Cons of the Java Parallel Streams Solution

- In general, there's a trade-off between computing performance and programmer productivity when choosing amongst Java parallelism frameworks.
  - e.g., CompletableFutures are more efficient and scalable than parallel streams, but are somewhat harder to program.



Java Parallel ImageStreamGang Example:
Evaluating Pros and Cons

The End

Java Parallel Streams

Evaluating the Pros

Douglas C. Schmidt

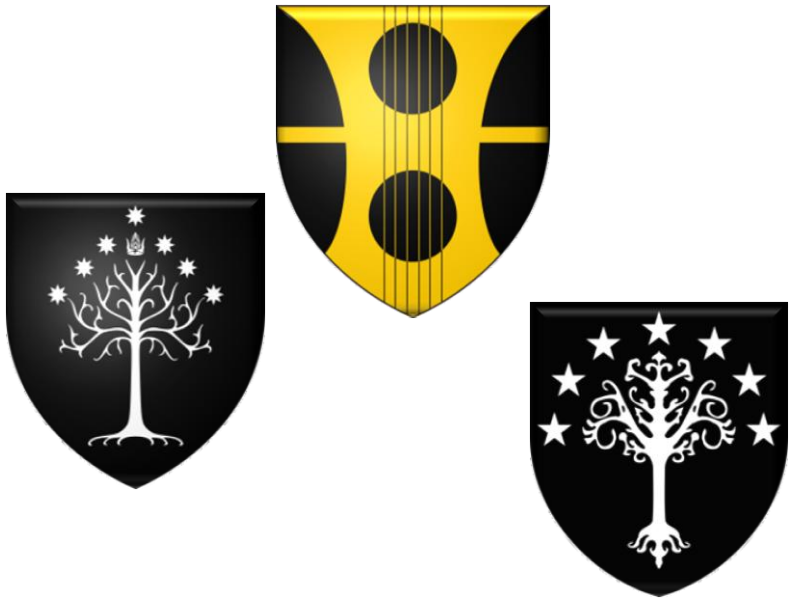
Learning Objectives in This Lesson

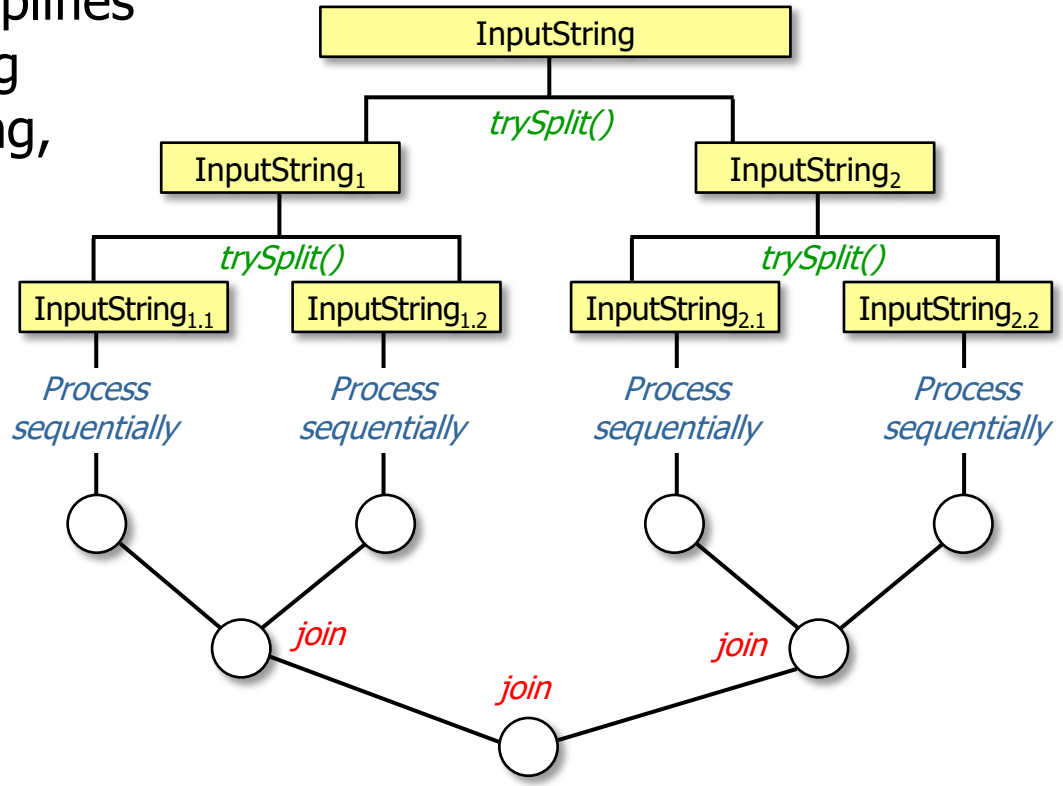
- Evaluate the pros of Java parallel streams.

Pros of Java Parallel Streams

Pros of Java Parallel Streams

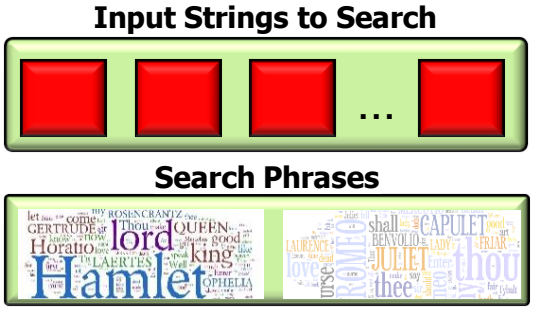
- The Java streams framework simplifies parallel programming by shielding developers from details of splitting, applying, and combining results.

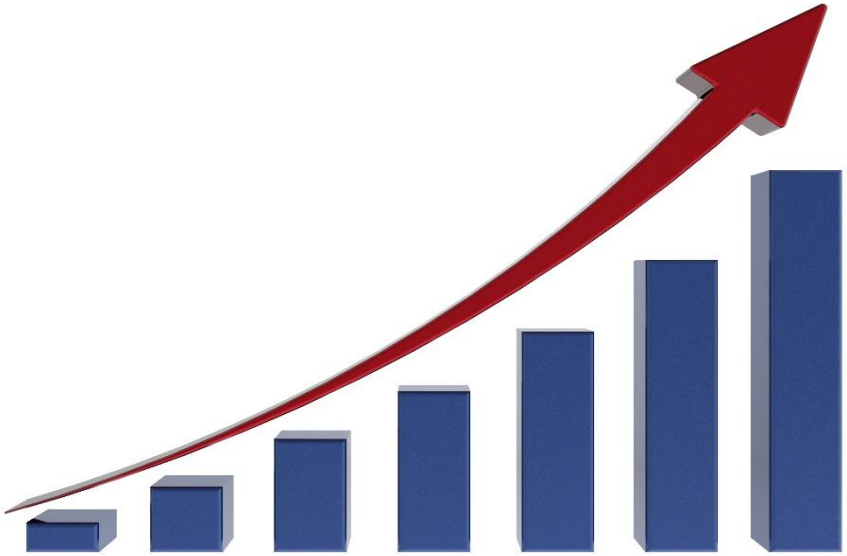


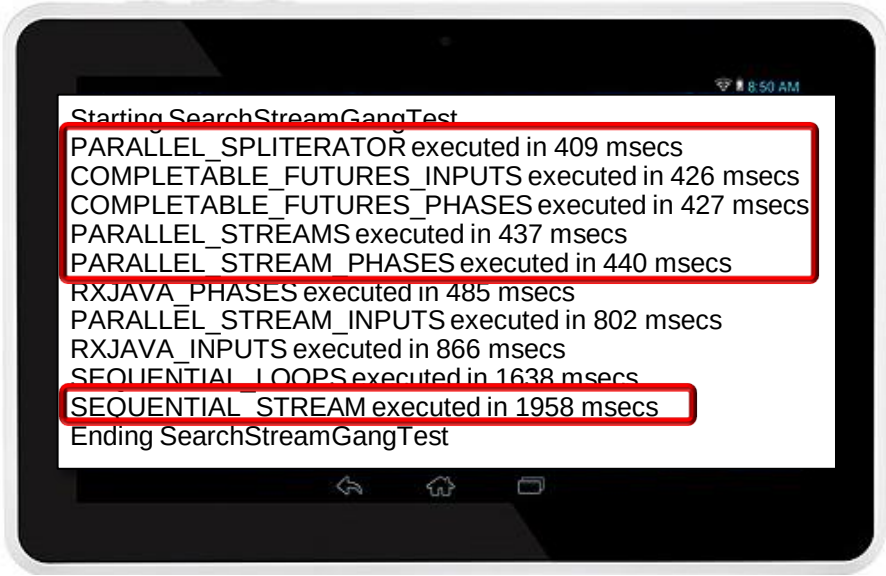


Pros of Java Parallel Streams

- Parallel stream implementations are often (much) faster and more scalable than sequential (stream and loops) implementations.

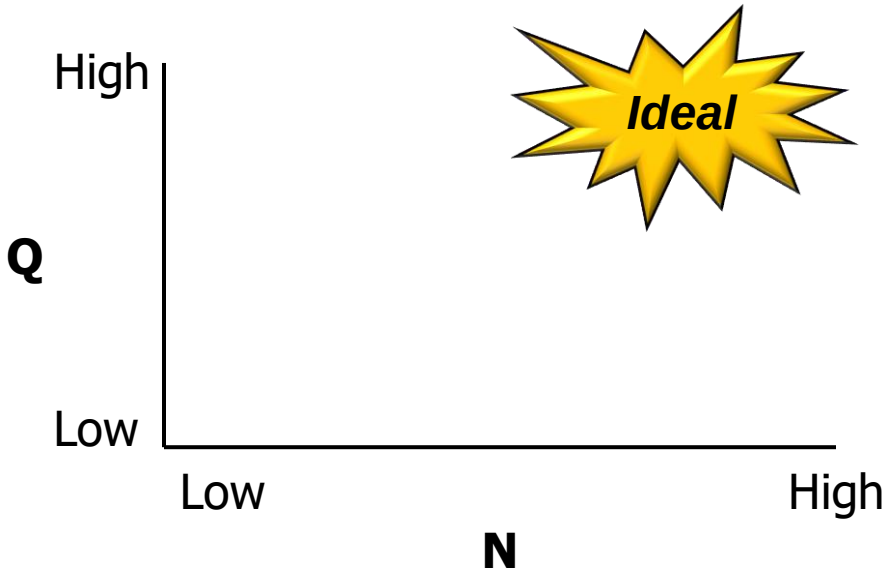






Pros of Java Parallel Streams

- The performance speedup is a largely a function of the partitioning strategy for the input (N), the amount of work performed (Q), and the # of cores.



The NQ model

- *N is the # of data elements to process per thread.*
- *Q quantifies how CPU-intensive the processing is.*

Pros of Java Parallel Streams

- Apps often don't need explicit synchronization or threading.

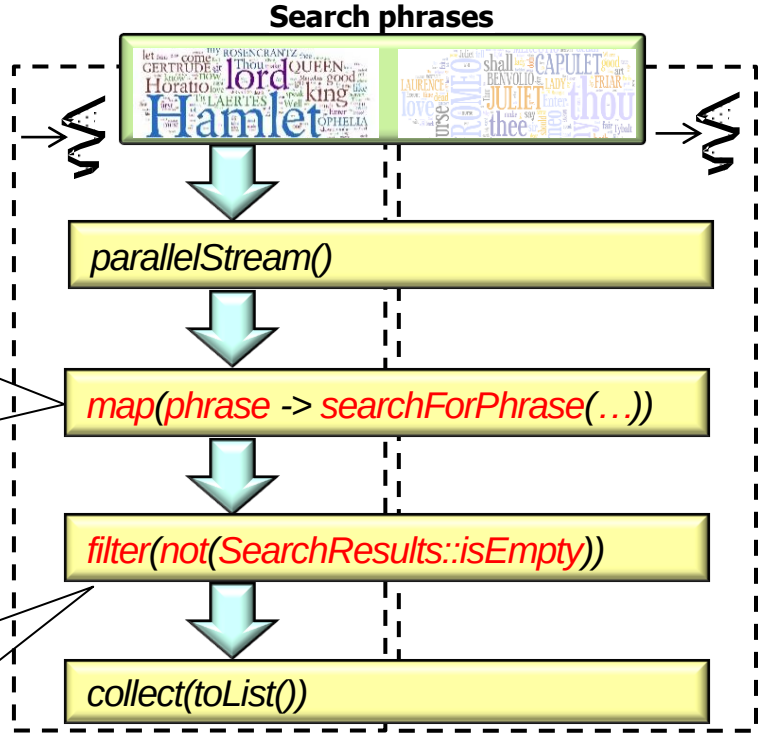
Alleviates many accidental and inherent complexities of concurrency/parallelism

Pros of Java Parallel Streams

- Apps often don't need explicit synchronization or threading.
  - Stateless behaviors alleviate the need to access shared mutable state.

```
return new SearchResults
   (Thread.currentThread().getId(),
    currentCycle(), phrase, title,
    StreamSupport
      .stream(new PhraseMatchSpliterator
                    (input, phrase),
              parallel)
      .collect(toList()));
```

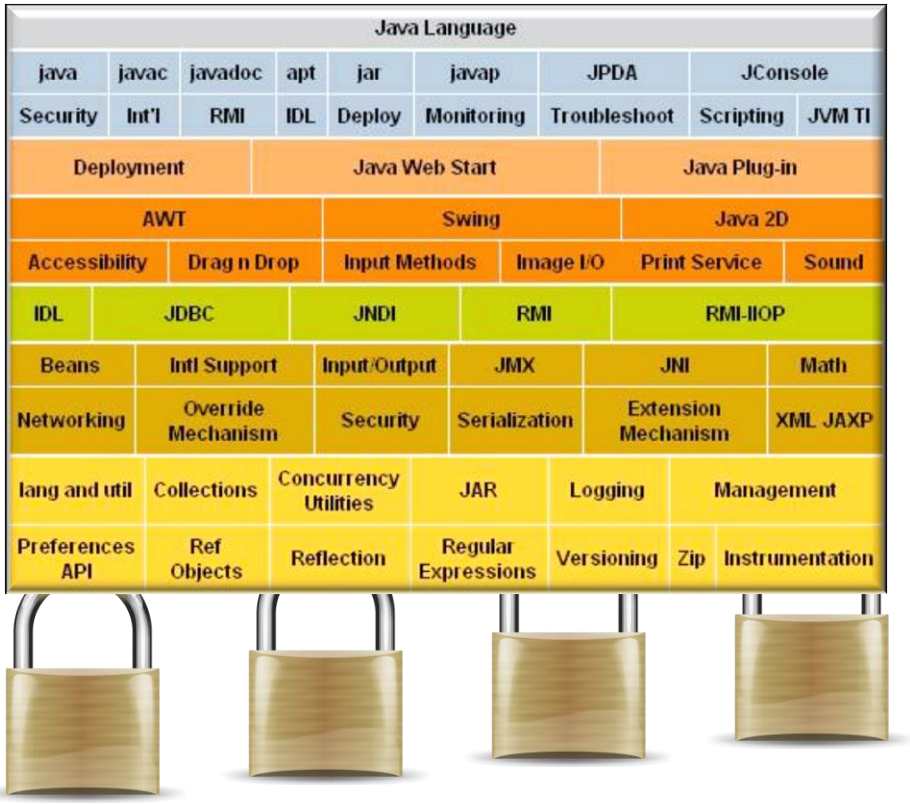
```
return mList.size() == 0;
```



See en.wikipedia.org/wiki/Pure_function

Pros of Java Parallel Streams

- Apps often don't need explicit synchronization or threading.
  - Stateless behaviors alleviate the need to access shared mutable state.
  - Java class library handles locking needed to protect shared mutable state.



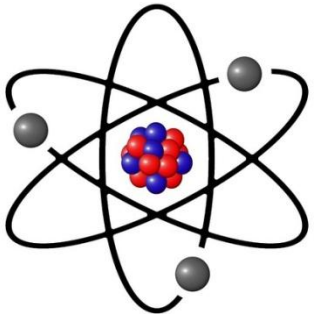
See docs.oracle.com/javase/tutorial/essential/concurrency/collections.html

Pros of Java Parallel Streams

- Streams ensure that the structure of sequential and parallel code is the same.

```
List<List<SearchResults>>
                processStream() {
  return getInput()
    .stream()
    .map(this::processInput)
    .collect(toList());
}
```

```
List<List<SearchResults>>
                processStream() {
  return getInput()
    .parallelStream()
    .map(this::processInput)
    .collect(toList());
}
```



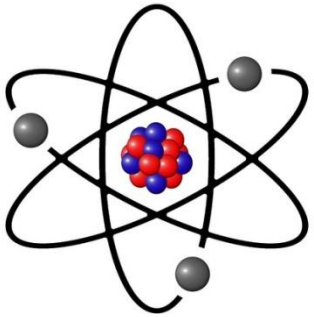
Converting sequential to parallel streams only requires minuscule changes!

Pros of Java Parallel Streams

- Streams ensure that the structure of sequential and parallel code is the same.

```
List<SearchResults> results =
  mPhrasesToFind
    .parallelStream()
    .map(phase ->
         searchForPhrase(...,
                         false))
    .filter(not(SearchResults
                ::isEmpty))
    .collect(toList());
```

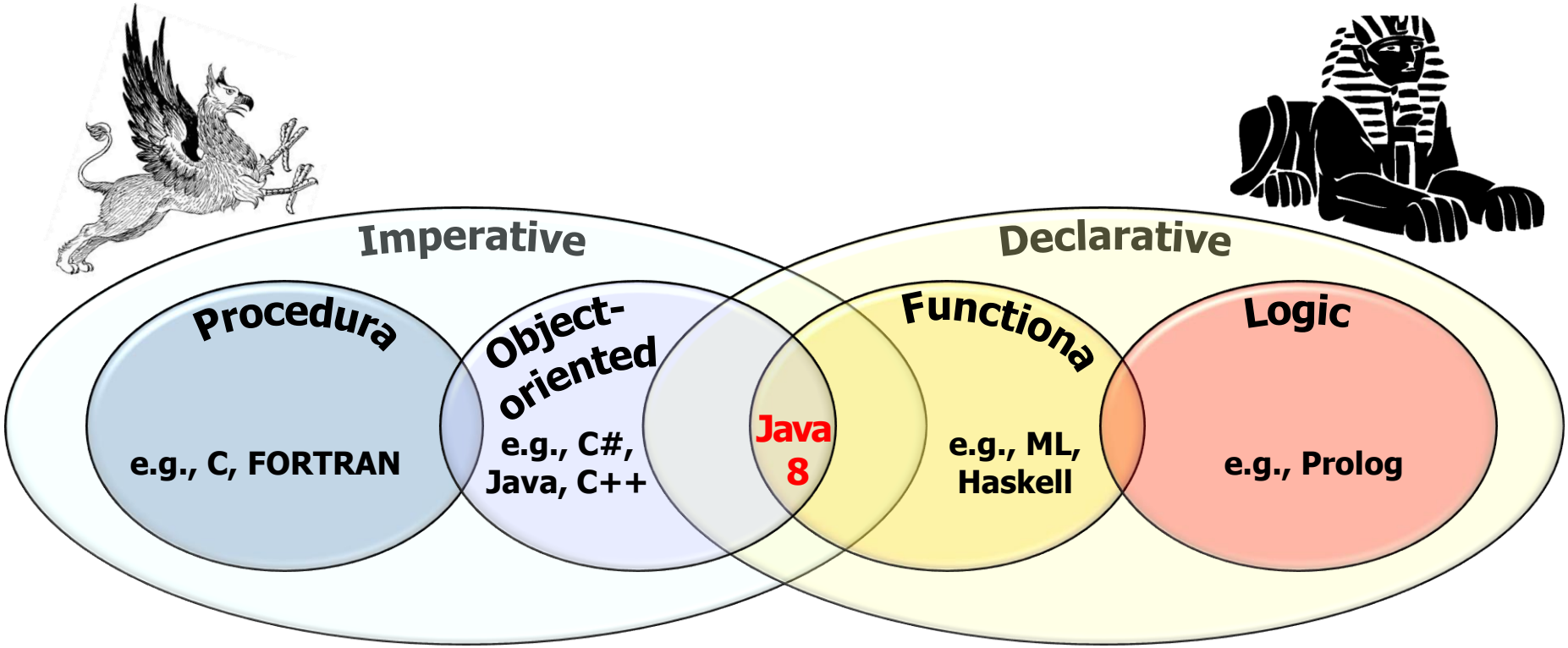
```
List<SearchResults> results =
  mPhrasesToFind
    .parallelStream()
    .map(phase ->
         searchForPhrase(...,
                         true))
    .filter(not(SearchResults
                ::isEmpty))
    .collect(toList());
```



Converting sequential to parallel streams only requires minuscule changes!

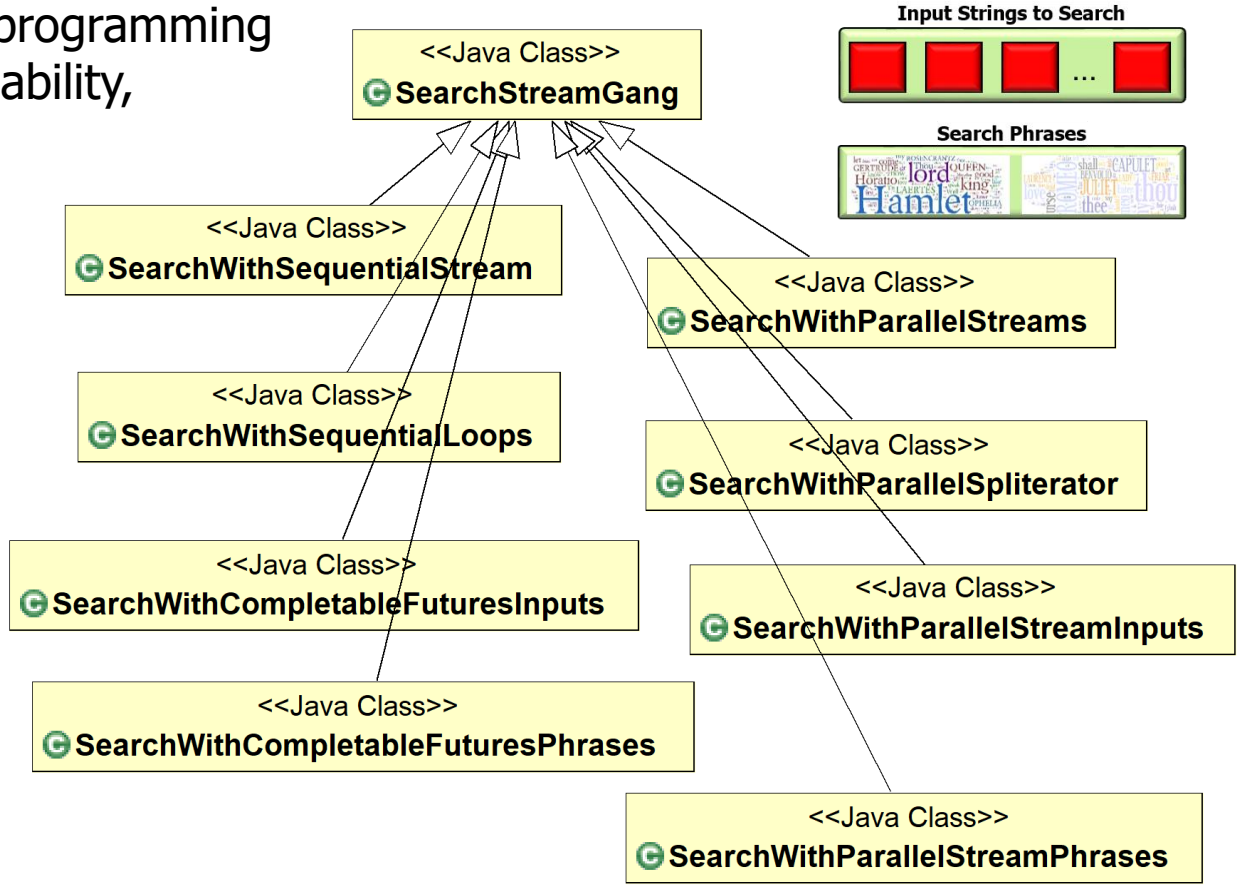
Pros of Java Parallel Streams

- Examples show synergies between functional and object-oriented programming.



Pros of Java Parallel Streams

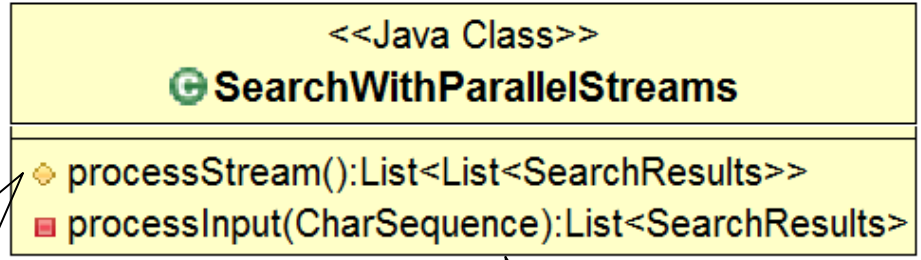
- Object-oriented design and programming features simplify understandability, reusability, and extensibility.



Object-oriented techniques emphasize systematic reuse of *structure.*

Pros of Java Parallel Streams

- Implementing object-oriented hook methods with functional programming features helps to close gap between domain intent and computations.




```
getInput()
   .parallelStream()
   .map(this::processInput)
   .collect(toList());
```

```
return mPhrasesToFind
   .parallelStream()
   .map(phrase -> searchForPhrase(phrase, input, title, false))
   .filter(not(SearchResults::isEmpty)
   .collect(toList());
```

The End

Java Parallel Streams

Evaluating the Cons

Douglas C. Schmidt

Learning Objectives in This Lesson

- Evaluate the cons of Java parallel streams.

Cons of Java Parallel Streams

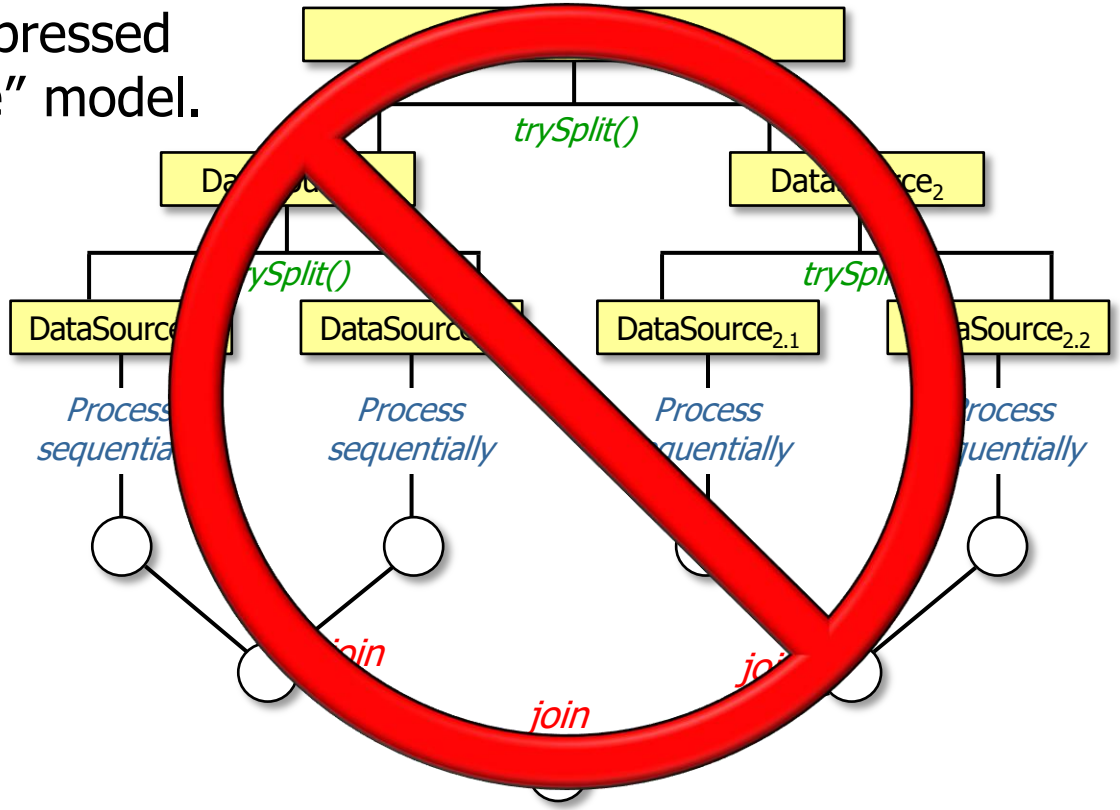
Cons of Java Parallel Streams

- There are some limitations with Java parallel streams.

The Java parallel streams framework is not all unicorns and rainbows!

Cons of Java Parallel Streams

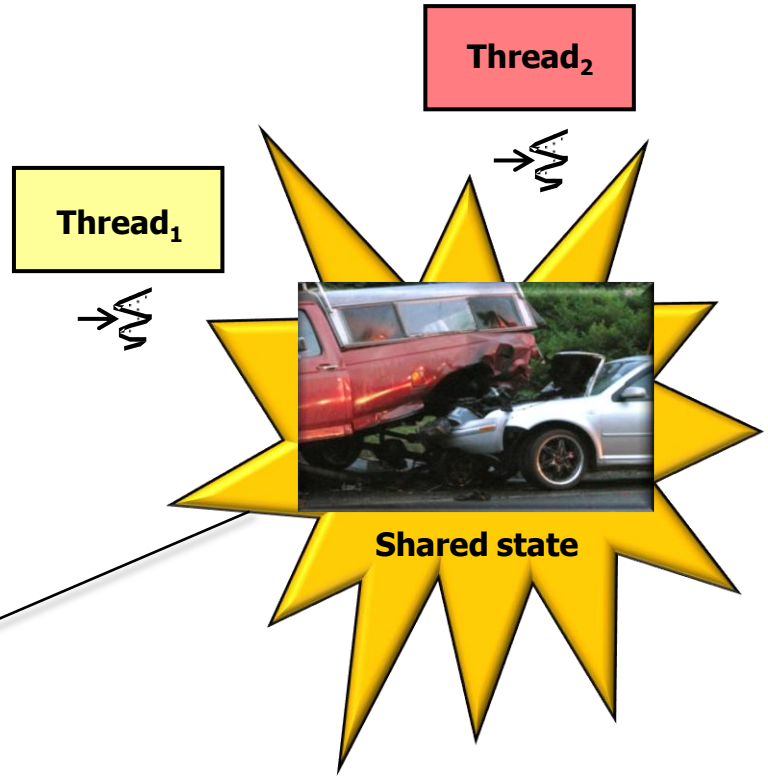
- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model.



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

Cons of Java Parallel Streams

- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model.
  - If behaviors aren't stateless and thread-safe, race conditions may occur.



Race conditions occur when a program depends on the sequence or timing of threads for it to operate properly.

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

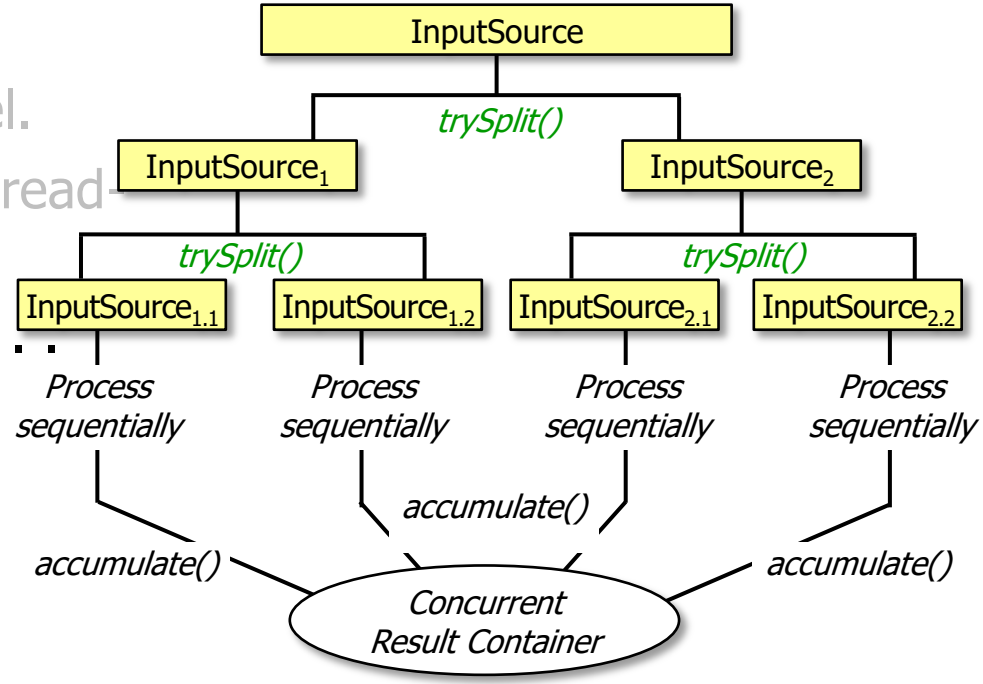
Cons of Java Parallel Streams

- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model.
  - If behaviors aren't stateless and thread-safe, race conditions may occur.
  - Parallel spliterators may be tricky . . .

See lesson on "*Java SearchWithParallelSpliterator Example: trySplit()*."

Cons of Java Parallel Streams

- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model.
  - If behaviors aren't stateless and thread-safe, race conditions may occur.
  - Parallel spliterators may be tricky . . .
    - Concurrent collectors are easier.



See lesson on "*Java Parallel Stream Internals: Nonconcurrent and Concurrent Collectors.*"

Cons of Java Parallel Streams

- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model.
  - If behaviors aren't stateless and thread-safe, race conditions may occur.
  - Parallel spliterators may be tricky . . .
  - All parallel streams share a common ForkJoinPool.

See dzone.com/articles/think-twice-using-java-8

Cons of Java Parallel Streams

- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model.
  - If behaviors aren't stateless and thread-safe, race conditions may occur.
  - Parallel spliterators may be tricky . . .
  - All parallel streams share a common ForkJoinPool.
    - Java CompletableFutures don't have this limitation.

See dzone.com/articles/think-twice-using-java-8

Cons of Java Parallel Streams

- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model.
  - If behaviors aren't stateless and thread-safe, race conditions may occur.
  - Parallel spliterators may be tricky . . .
  - All parallel streams share a common ForkJoinPool.
    - Java CompletableFutures don't have this limitation.
    - It's important to know how to apply ManagedBlockers.

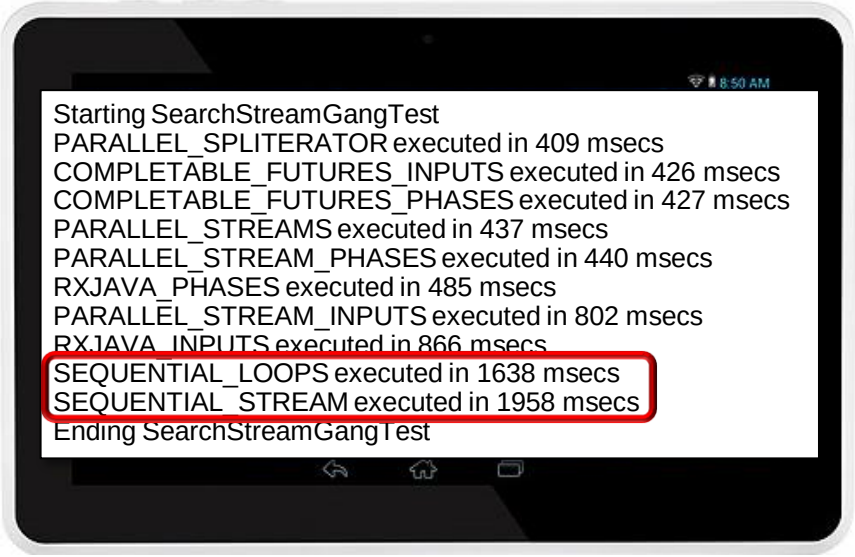
See "*The Java ForkJoinPool: Applying the ManagedBlocker Interface.*"

Cons of Java Parallel Streams

- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model.
  - If behaviors aren't stateless and thread-safe, race conditions may occur.
  - Parallel spliterators may be tricky . . .
  - All parallel streams share a common ForkJoinPool.
  - Streams incur some overhead.

Cons of Java Parallel Streams

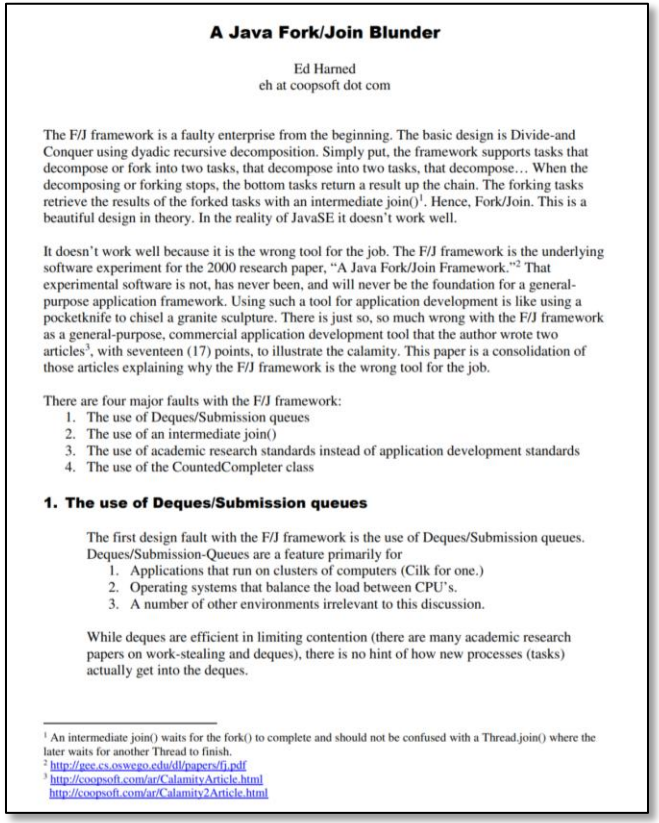
- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model.
  - If behaviors aren't stateless and thread-safe, race conditions may occur.
  - Parallel spliterators may be tricky . . .
  - All parallel streams share a common ForkJoinPool.
  - Streams incur some overhead, e.g.
    - Splitting and combining overhead



See blog.jooq.org/2015/12/08/3-reasons-why-you-shouldnt-replace-your-for-loops-by-stream-foreach

Cons of Java Parallel Streams

- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model.
  - If behaviors aren't stateless and thread-safe, race conditions may occur.
  - Parallel spliterators may be tricky . . .
  - All parallel streams share a common ForkJoinPool.
  - Streams incur some overhead, e.g.
    - Splitting and combining overhead
    - Fork/join framework



A Java Fork/Join Blunder

Ed Harned
eh at coopsoft dot com

The F/J framework is a faulty enterprise from the beginning. The basic design is Divide-and Conquer using dyadic recursive decomposition. Simply put, the framework supports tasks that decompose or fork into two tasks, that decompose into two tasks, that decompose… When the decomposing or forking stops, the bottom tasks return a result up the chain. The forking tasks retrieve the results of the forked tasks with an intermediate join()[1]. Hence, Fork/Join. This is a beautiful design in theory. In the reality of JavaSE it doesn't work well.

It doesn't work well because it is the wrong tool for the job. The F/J framework is the underlying software experiment for the 2000 research paper, "A Java Fork/Join Framework."[2] That experimental software is not, has never been, and will never be the foundation for a general-purpose application framework. Using such a tool for application development is like using a pocketknife to chisel a granite sculpture. There is just so, so much wrong with the F/J framework as a general-purpose, commercial application development tool that the author wrote two articles[3], with seventeen (17) points, to illustrate the calamity. This paper is a consolidation of those articles explaining why the F/J framework is the wrong tool for the job.

There are four major faults with the F/J framework:
1. The use of Deques/Submission queues
2. The use of an intermediate join()
3. The use of academic research standards instead of application development standards
4. The use of the CountedCompleter class

1. The use of Deques/Submission queues

The first design fault with the F/J framework is the use of Deques/Submission queues. Deques/Submission-Queues are a feature primarily for
1. Applications that run on clusters of computers (Cilk for one.)
2. Operating systems that balance the load between CPU's.
3. A number of other environments irrelevant to this discussion.

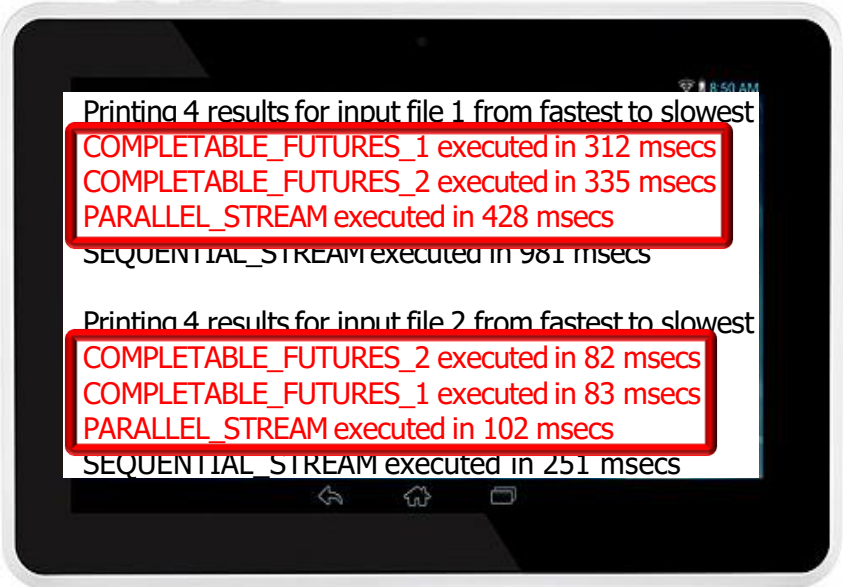
While deques are efficient in limiting contention (there are many academic research papers on work-stealing and deques), there is no hint of how new processes (tasks) actually get into the deques.

[1] An intermediate join() waits for the fork() to complete and should not be confused with a Thread.join() where the later waits for another Thread to finish.
[2] http://gee.cs.oswego.edu/dl/papers/fj.pdf
[3] http://coopsoft.com/ar/CalamityArticle.html
http://coopsoft.com/ar/Calamity2Article.html

See coopsoft.com/dl/Blunder.pdf

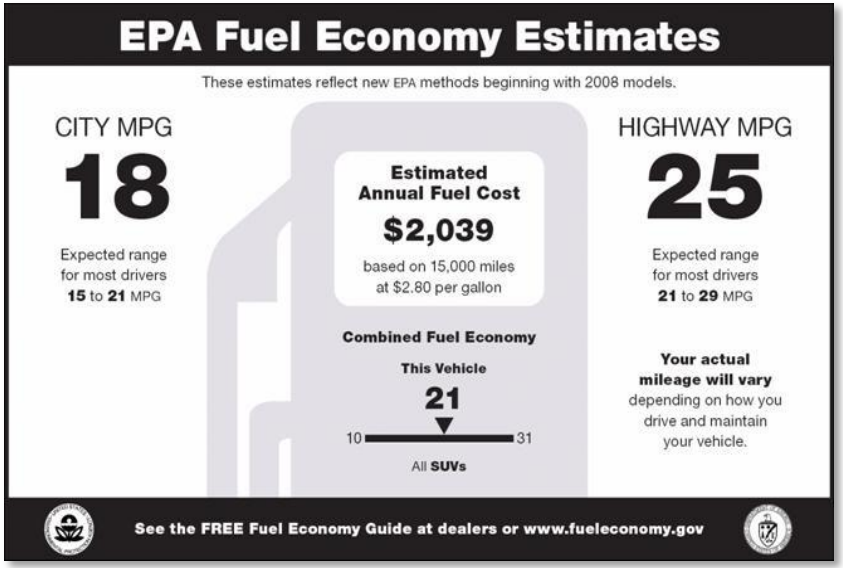
Cons of Java Parallel Streams

- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model.
  - If behaviors aren't stateless and thread-safe, race conditions may occur.
  - Parallel spliterators may be tricky . . .
  - All parallel streams share a common ForkJoinPool.
  - Streams incur some overhead, e.g.
    - Splitting and combining overhead
    - Fork/join framework
    - Java CompletableFutures may be more efficient and scalable



Cons of Java Parallel Streams

- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model.
  - If behaviors aren't stateless and thread-safe, race conditions may occur.
  - Parallel spliterators may be tricky . . .
  - All parallel streams share a common ForkJoinPool.
  - Streams incur some overhead, e.g.
    - Splitting and combining overhead
    - Fork/join framework
    - Java CompletableFutures may be more efficient and scalable



Naturally, your mileage may vary.

Cons of Java Parallel Streams

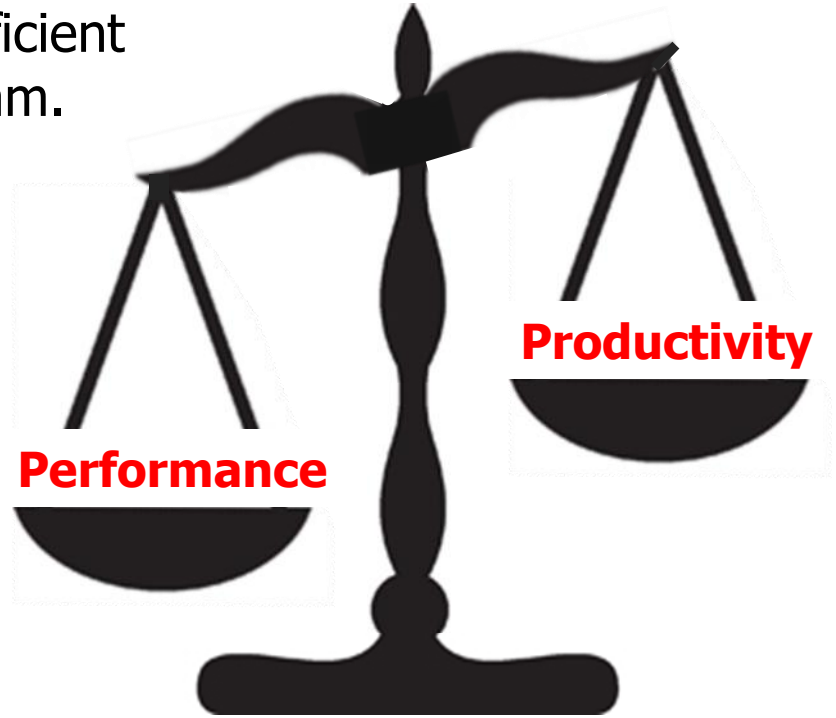
- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model.
  - If behaviors aren't stateless and thread-safe, race conditions may occur.
  - Parallel spliterators may be tricky . . .
  - All parallel streams share a common ForkJoinPool.
  - Streams incur some overhead.
  - There's no substitute for benchmarking!

See java-performance.info/jmh

Wrapping Up Java Parallel Streams

Wrapping Up Java Parallel Streams

- In general, there's a trade-off between computing performance and programmer productivity when choosing amongst these frameworks.
  - e.g., CompletableFutures are more efficient and scalable, but are harder to program.



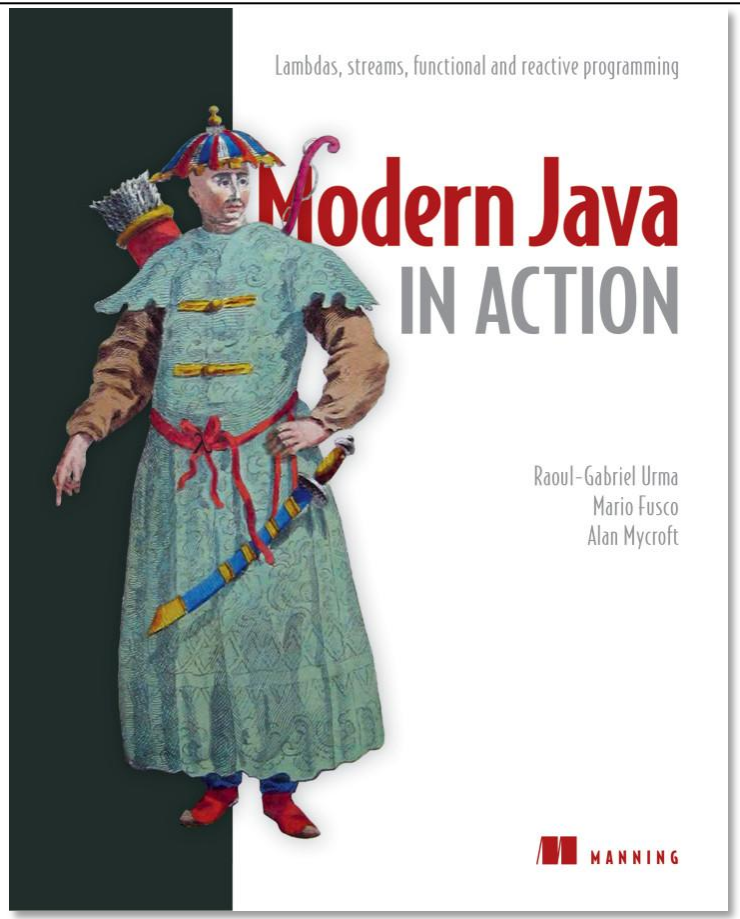
Wrapping Up Java Parallel Streams

- In general, however, the pros of Java parallel streams far outweigh the cons for many use cases!

See www.ibm.com/developerworks/library/j-jvmc2

Wrapping Up Java Parallel Streams

- Good coverage of parallel streams appears in the book "Modern Java in Action."



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

The End