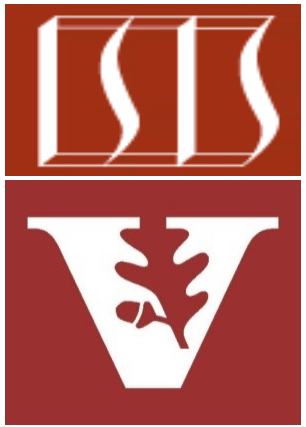
Avoiding Programming Hazards with Java Parallel Streams

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt



Professor of Computer Science

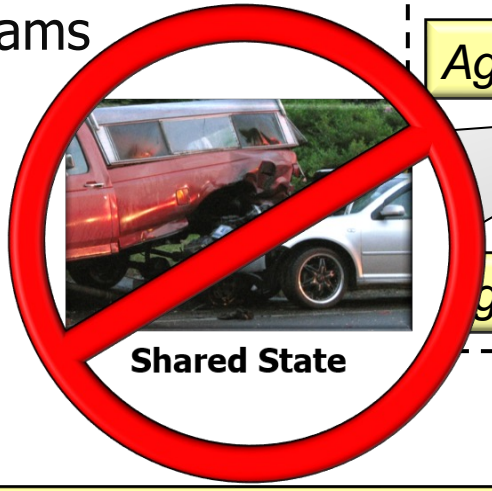
Institute for Software Integrated Systems

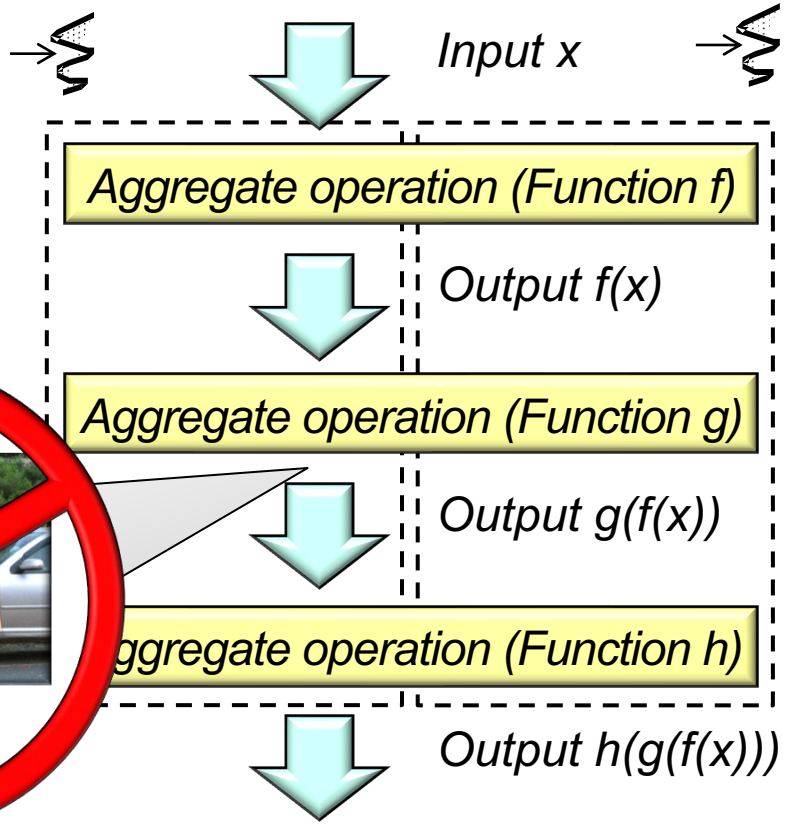
Vanderbilt University

Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

- Know how aggregate operations & functional programming features are applied seamlessly in parallel streams
- Be aware of how parallel stream phases work "under the hood"
- Recognize common programming hazards in Java parallel streams & how to avoid them





See earlier lesson on "*Java Streams: Avoiding Common Programming Mistakes*"

Learning Objectives in this Part of the Lesson

- Know how aggregate operations & functional programming features are applied seamlessly in parallel streams
- Be aware of how parallel stream phases work "under the hood"
- Recognize common programming hazards in Java parallel streams & how to avoid them, e.g.
  - Hazards with stateful lambda expressions

```
class BuggyFactorial2 {
  static long factorial(long n){
    return LongStream
      .rangeClosed(1, n)

      .parallel()

      .mapToObj(Mult::new)

      .reduce(new Mult(0L),
              Mult::multiply)

      .bigInteger();
} ...
```

Learning Objectives in this Part of the Lesson

- Know how aggregate operations & functional programming features are applied seamlessly in parallel streams
- Be aware of how parallel stream phases work "under the hood"
- Recognize common programming hazards in Java parallel streams & how to avoid them, e.g.
  - Hazards with stateful lambda expressions
  - Hazards from interference with the data source

```
List<Integer> list = IntStream
   .range(0, 10)
   .boxed()
   .collect(toCollection
                (LinkedList::new));
```

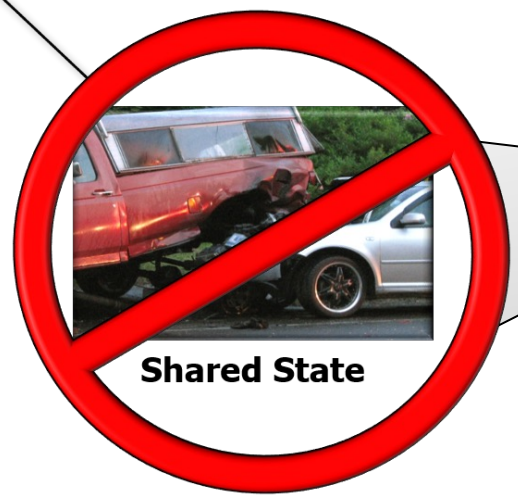
```
list
   .parallelStream()
   .peek(list::remove)
   .forEach(System.out::println);
```

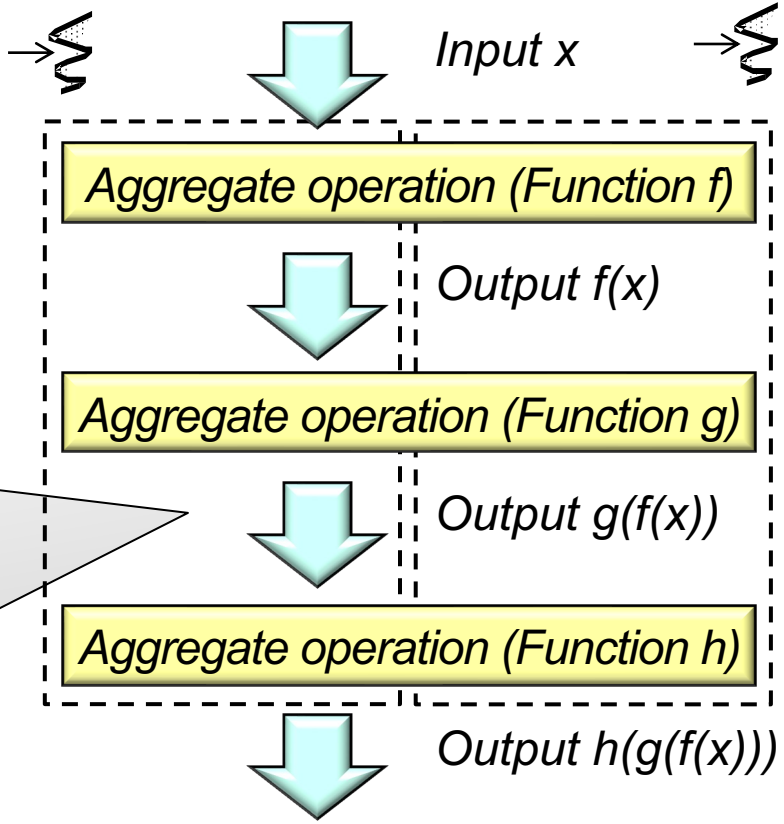
Avoiding Programming Hazards in Java Parallel Streams

Avoiding Programming Hazards in Java Parallel Streams

- The Java parallel streams framework assumes behaviors don't incur race conditions

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

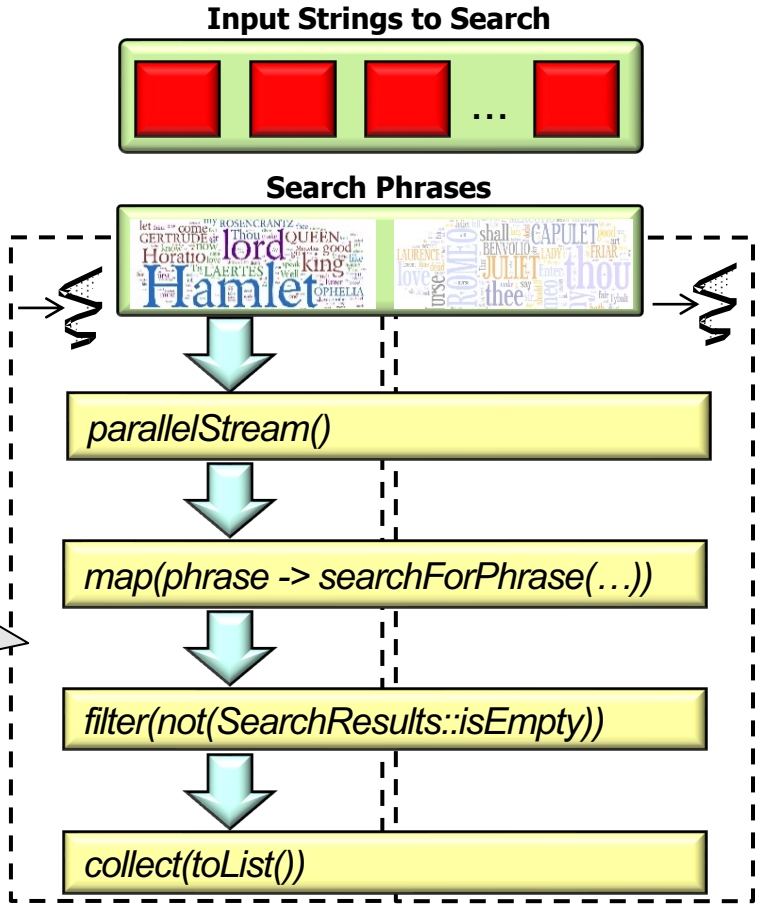




See en.wikipedia.org/wiki/Race_condition#Software

Avoiding Programming Hazards in Java Parallel Streams

- Thus avoid (or at least) minimize behaviors that have side-effects when programming parallel streams



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

Hazards with Stateful Lambda Expressions

Hazards with Stateful Lambda Expressions

- Avoid/minimize behaviors that use stateful lambda expressions
  - i.e., where results depend on shared mutable state

```
class BuggyFactorial2 {
  static class Mult {
    long mLong;

    Mult(long l)
    { mLong = l; }

    Mult multiply(Mult mult) {
        mLong *= mult.mLong;
      return this;
    }

    long longValue()
    { return mLong; }
  } ...
```

See docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#Statelessness

Hazards with Stateful Lambda Expressions

- Avoid/minimize behaviors that use stateful lambda expressions
  - i.e., where results depend on shared mutable state

```
class BuggyFactorial2 {
  static class Mult {
    long mLong;

    Mult(long l)
    { mLong = l; }

    Mult multiply(Mult mult) {
        mLong *= mult.mLong;
      return this;
    }

    long longValue()
    { return mLong; }
  } ...
```

This example demonstrates the problem

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

Hazards with Stateful Lambda Expressions

- Avoid/minimize behaviors that use stateful lambda expressions
  - i.e., where results depend on shared mutable state

Defines mutable state that's shared between threads in a parallel stream

```
class BuggyFactorial2 {
  static class Mult {
    long mLong;

    Mult(long l)
    { mLong = l; }

    Mult multiply(Mult mult) {
      mLong *= mult.mLong;
      return this;
    }

    long longValue()
    { return mLong; }
  } ...
```

Hazards with Stateful Lambda Expressions

- Avoid/minimize behaviors that use stateful lambda expressions
  - i.e., where results depend on shared mutable state
    - & where this state that may change in parallel execution of a pipeline

```
class BuggyFactorial2 {
  ...
  static long factorial(long n){
    return LongStream
      .rangeClosed(1, n)

      .parallel()

      .mapToObj(Mult::new)

      .reduce(new Mult(0L),
              Mult::multiply)

      .bigInteger();
  } ...
```

Incorrectly compute the factorial of param n using a parallel stream

Hazards with Stateful Lambda Expressions

- Avoid/minimize behaviors that use stateful lambda expressions
  - i.e., where results depend on shared mutable state
    - & where this state that may change in parallel execution of a pipeline

```
class BuggyFactorial2 {
  ...
  static long factorial(long n){
    return LongStream
      .rangeClosed(1, n)

      .parallel()

      .mapToObj(Mult::new)

      .reduce(new Mult(0L),
              Mult::multiply)

      .bigInteger();
  } ...
```

Race conditions may arise from the unsynchronized access to the mutable mLong field in Mult objects

This problem occurs even though we're using the reduce() terminal operation!

Hazards with Stateful Lambda Expressions

- Avoid/minimize behaviors that use stateful lambda expressions
  - i.e., where results depend on shared mutable state
    - & where this state that may change in parallel execution of a pipeline

```
class ParallelFactorial {
  static long factorial(long n){

    return LongStream
      .rangeClosed(1, n)

      .parallel()

      .reduce(1L,
              (x, y) -> x * y);
} ...
```

Using the reduce() terminal operation with immutable objects trivially addresses these problems!

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

Hazards from Interference with the Data Source

Hazards from Interference with the Data Source

- Also avoid behaviors that interfere with the data source
  - This occurs when source of stream is modified within the pipeline

```
List<Integer> list = IntStream
   .range(0, 10)
   .boxed()
   .collect(toCollection
                (LinkedList::new));
```


```
list
   .parallelStream()
   .peek(list::remove)
   .forEach(System.out::println);
```

See docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#NonInterference

Hazards from Interference with the Data Source

- Also avoid behaviors that interfere with the data source
  - This occurs when source of stream is modified within the pipeline

```
List<Integer> list = IntStream
   .range(0, 10)
   .boxed()
   .collect(toCollection
                (LinkedList::new));
```

Create a list of ten integers in range 0..9

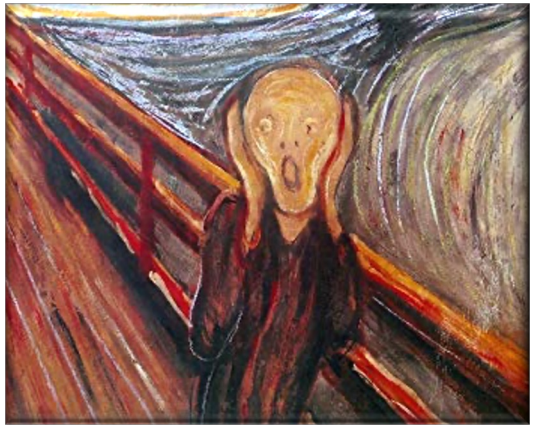
```
list
   .parallelStream()
   .peek(list::remove)
   .forEach(System.out::println);
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex11

Hazards from Interference with the Data Source

- Also avoid behaviors that interfere with the data source
  - This occurs when source of stream is modified within the pipeline

```
List<Integer> list = IntStream
   .range(0, 10)
   .boxed()
   .collect(toCollection
                (LinkedList::new));
```



```
list
   .parallelStream()
   .peek(list::remove)
   .forEach(System.out::println);
```

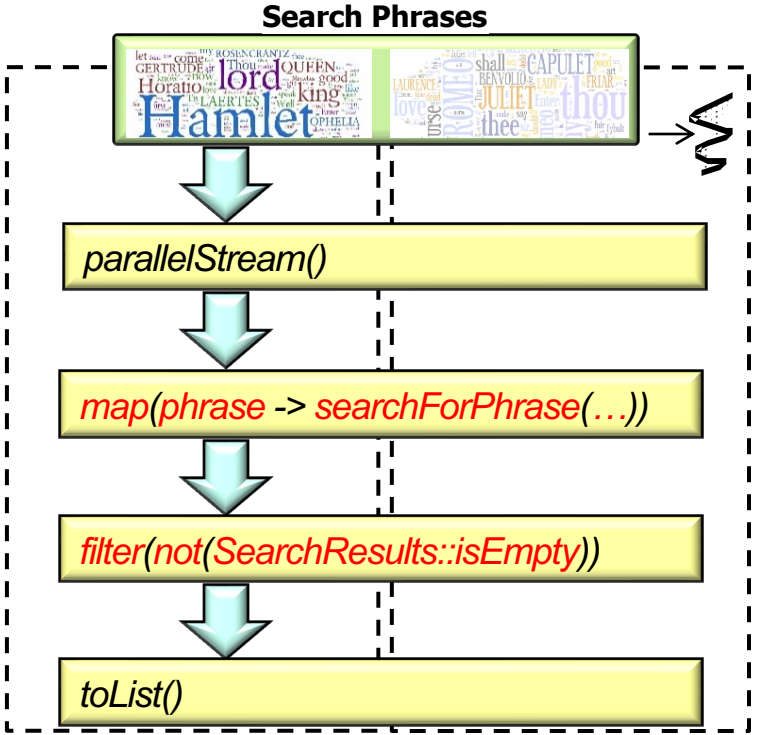
If a non-concurrent collection is modified while it's being operated on by the parallel stream the results will be chao & insanity!!

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

Avoiding Parallel Programming Hazards

Avoiding Parallel Programming Hazards

- Behaviors involving no shared state or side-effects are useful for parallel streams since they needn't be synchronized explicitly



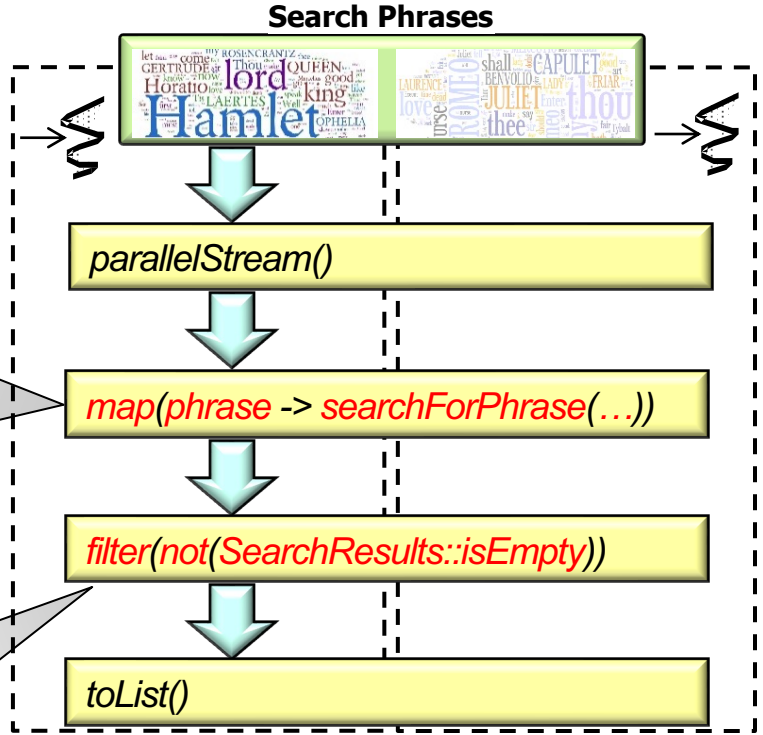
See henrikeichenhardt.blogspot.com/2013/06/why-shared-mutable-state-is-root-of-all.html

Avoiding Parallel Programming Hazards

- Behaviors involving no shared state or side-effects are useful for parallel streams since they needn't be synchronized explicitly
  - e.g., Java lambda expressions & method references that are "pure functions"

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

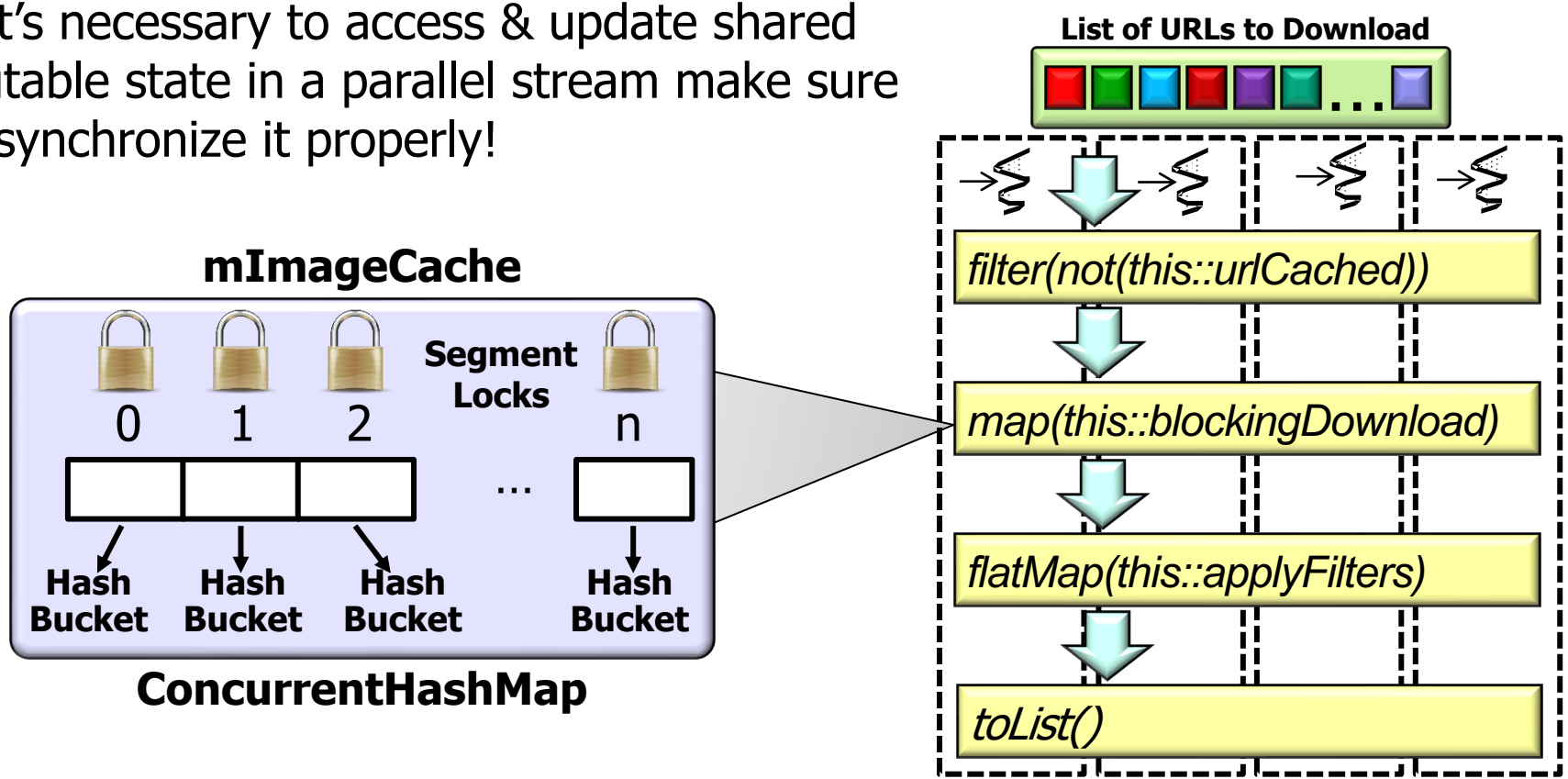
```
return mList.size() == 0;
```



See en.wikipedia.org/wiki/Pure_function

Avoiding Parallel Programming Hazards

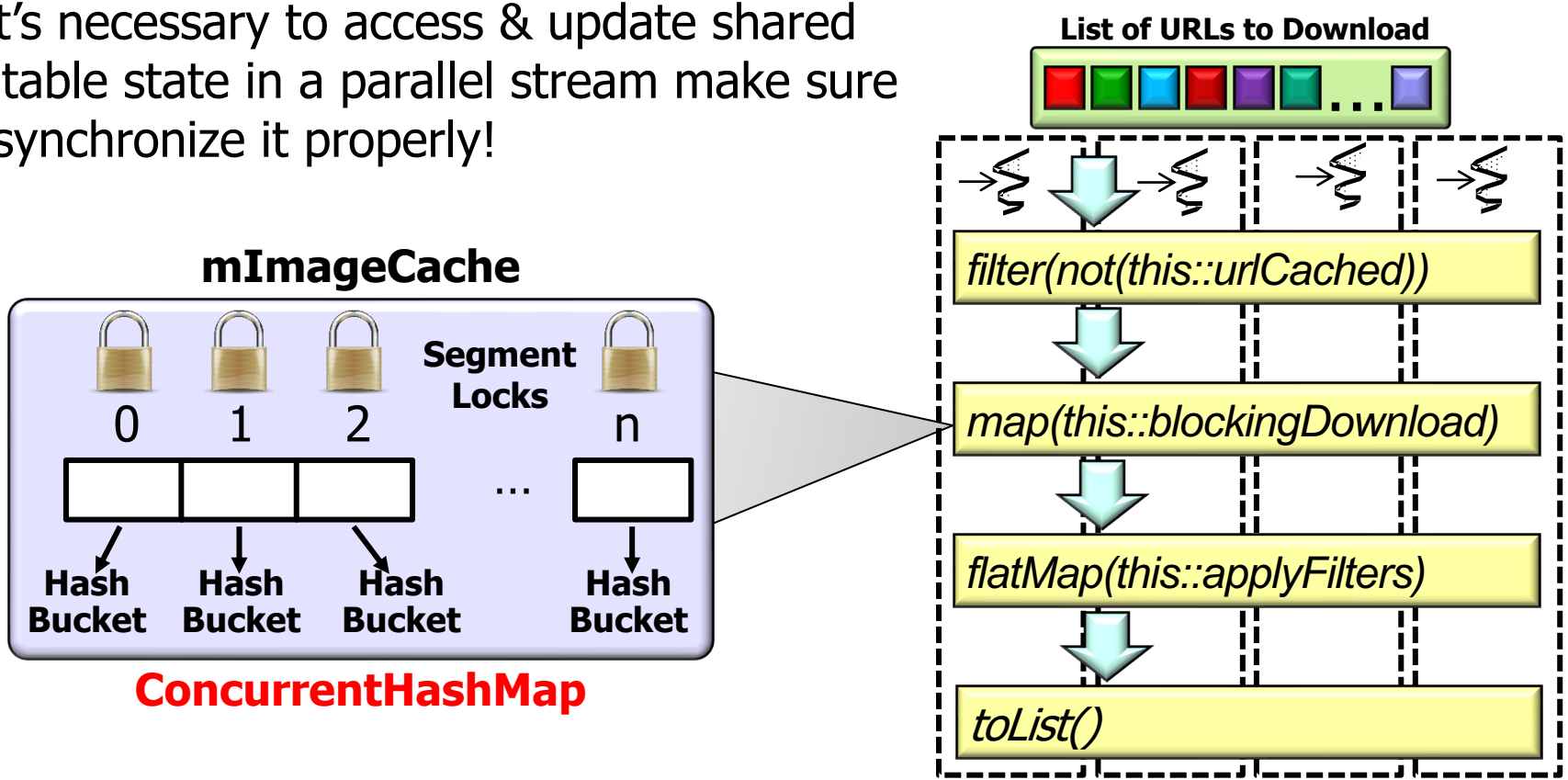
- If it's necessary to access & update shared mutable state in a parallel stream make sure to synchronize it properly!



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

Avoiding Parallel Programming Hazards

- If it's necessary to access & update shared mutable state in a parallel stream make sure to synchronize it properly!



See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html

End of Avoiding Programming Hazards with Java Parallel Streams