Overview of Java 8 Parallel Streams (Part 2)

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

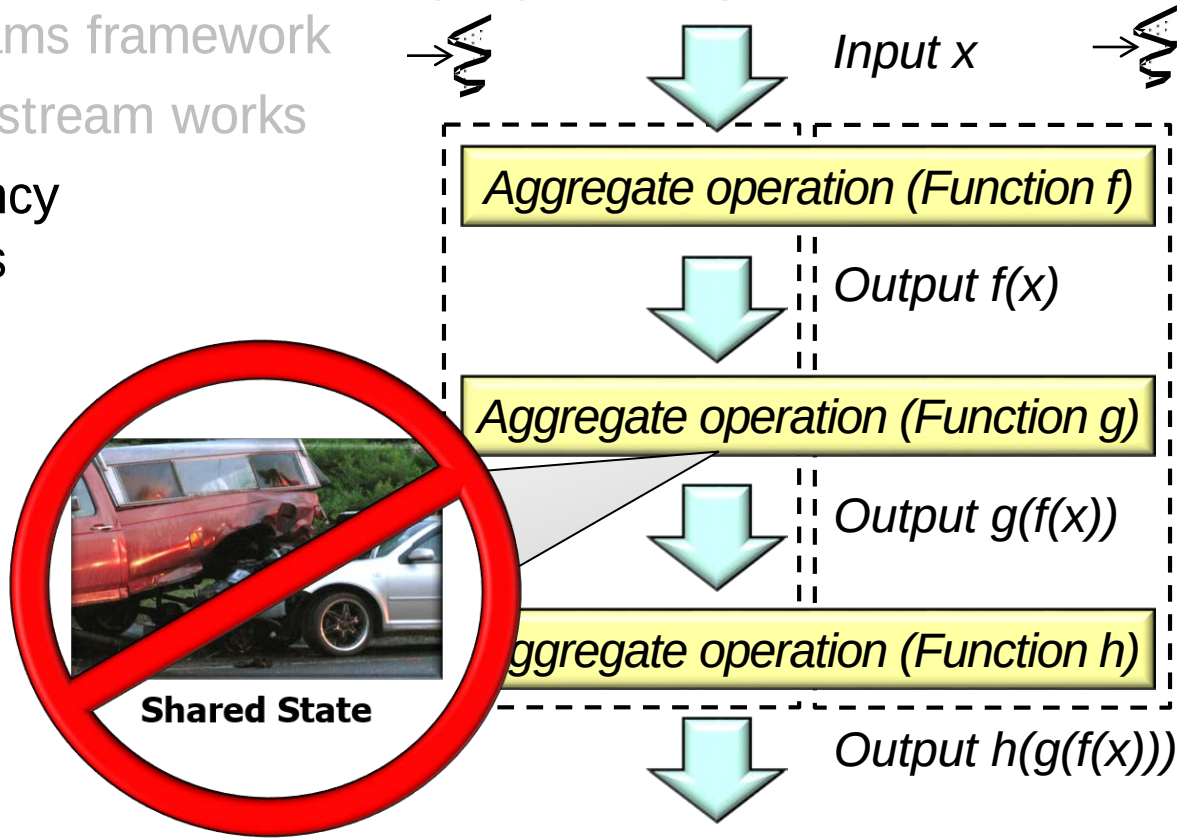
Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

- Know how aggregate operations & functional programming features are applied in the parallel streams framework
- Understand how a parallel stream works
- Be able to avoid concurrency hazards in parallel streams

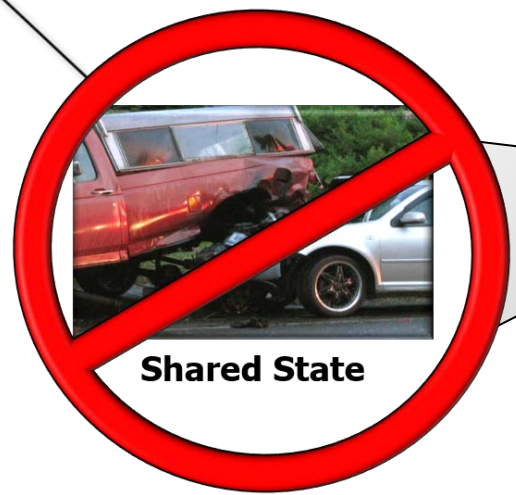


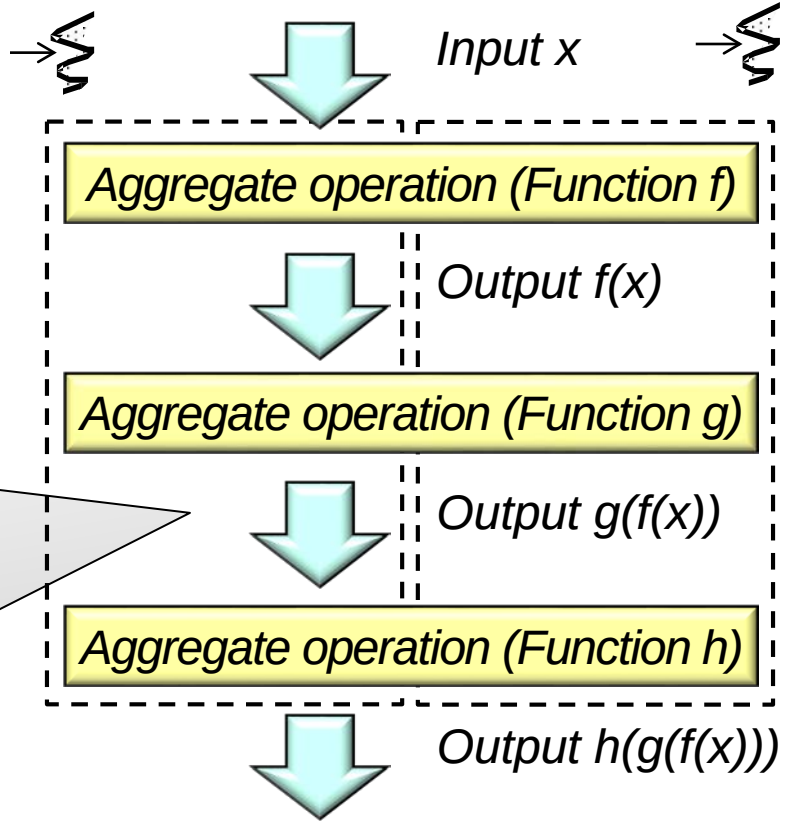
Avoiding Concurrency Hazards in Java 8 Parallel Streams

Avoiding Concurrency Hazards in Java 8 Parallel Streams

- The Java 8 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

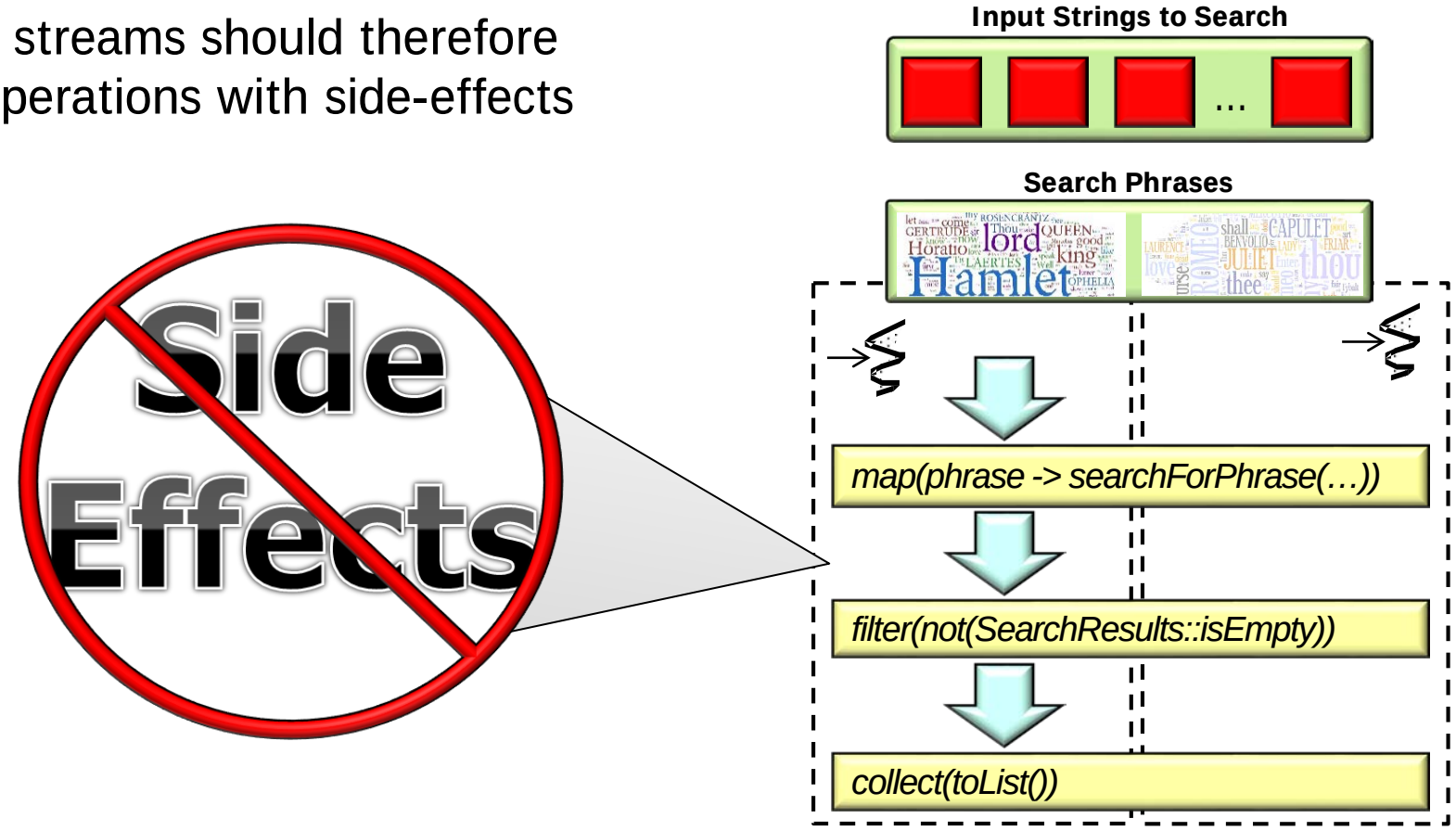




See en.wikipedia.org/wiki/Race_condition#Software

Avoiding Concurrency Hazards in Java 8 Parallel Streams

- Parallel streams should therefore avoid operations with side-effects



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

Avoiding Concurrency Hazards in Java 8 Parallel Streams

- Parallel streams should therefore avoid operations with side-effects, e.g.
  - *Stateful lambda expressions*
    - Where results depends on shared mutable state

```
class BuggyFactorial {
  static class Total {
    long mTotal = 1;
    void multiply(long n)
    { mTotal *= n; }
  }

  static long factorial(long n){
    Total t = new Total();
    LongStream
      .rangeClosed(1, n)
      .parallel()
      .forEach(t::multiply);

    return t.mTotal;
  } ...
```

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

Avoiding Concurrency Hazards in Java 8 Parallel Streams

- Parallel streams should therefore avoid operations with side-effects, e.g.
  - *Stateful lambda expressions*
    - Where results depends on shared mutable state
      - i.e., state that may change in parallel execution of a pipeline

```
class BuggyFactorial {
  static class Total {
    long mTotal = 1;
    void multiply(long n)
    { mTotal *= n; }
  }

  static long factorial(long n){
    Total t = new Total();
    LongStream
      .rangeClosed(1, n)
      .parallel()
      .forEach(t::multiply);

    return t.mTotal;
  } ...
```

Avoiding Concurrency Hazards in Java 8 Parallel Streams

- Parallel streams should therefore avoid operations with side-effects, e.g.
  - *Stateful lambda expressions*
    - Where results depends on shared mutable state
      - i.e., state that may change in parallel execution of a pipeline

Race conditions can arise due to the unsynchronized access to mTotal field

```
class BuggyFactorial {
  static class Total {
    long mTotal = 1;
    void multiply(long n)
    { mTotal *= n; }
  }

  static long factorial(long n){
    Total t = new Total();
    LongStream
      .rangeClosed(1, n)
      .parallel()
      .forEach(t::multiply);

    return t.mTotal;
  } ...
```

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

Avoiding Concurrency Hazards in Java 8 Parallel Streams

- Parallel streams should therefore avoid operations with side-effects, e.g.
  - *Stateful lambda expressions*
  - *Interference w/the data source*
    - Occurs when source of stream is modified within the pipeline


```
List<Integer> list = IntStream
   .range(0, 10)
   .boxed()
   .collect(toList());

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

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

Avoiding Concurrency Hazards in Java 8 Parallel Streams

- Parallel streams should therefore avoid operations with side-effects, e.g.
  - *Stateful lambda expressions*
  - *Interference w/the data source*
    - Occurs when source of stream is modified within the pipeline

```
List<Integer> list = IntStream
   .range(0, 10)
   .boxed()
   .collect(toList());

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

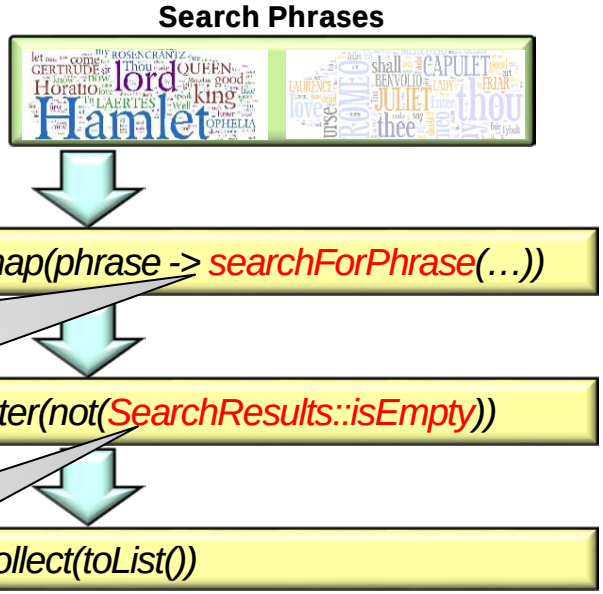
Aggregate operations enable parallelism with non-thread-safe collections provided the collection is not modified while it's being operated on..

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

Avoiding Concurrency Hazards in Java 8 Parallel Streams

- Java 8 lambda expressions & method references containing no shared state are useful for parallel streams since they needn't be explicitly synchronized

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




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

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