Overview of Java Parallel Streams: Avoiding Programming Hazards

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
- Be aware of how parallel stream phases work “under the hood”
- Recognize now to avoid programming hazards in parallel streams

\[ \text{Input } x \]

\[ \text{Aggregate operation (Function } f) \]

\[ \text{Output } f(x) \]

\[ \text{Aggregate operation (Function } g) \]

\[ \text{Output } g(f(x)) \]

\[ \text{Aggregate operation (Function } h) \]

\[ \text{Output } h(g(f(x))) \]

\[ \text{Shared State} \]
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](en.wikipedia.org/wiki/Race_condition#Software)
Avoiding Programming Hazards in Java Parallel Streams

- Parallel streams should therefore avoid behaviors with side-effects

See docs.oracle.com/javase/tutorial/collections/streams/parallelism.html#side_effects
Avoiding Programming Hazards in Java Parallel Streams

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

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

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

See [docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#Statelessness](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#Statelessness)
Parallel streams should therefore avoid behaviors with side-effects, e.g.:

- **Stateful lambda expressions**
  - Where results depend on shared mutable state
  - i.e., state that may change in parallel execution of a pipeline

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

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

See [github.com/douglasraigschmidt/LiveLessons/tree/master/Java8/ex16](https://github.com/douglasraigschmidt/LiveLessons/tree/master/Java8/ex16)
• Parallel streams should therefore avoid behaviors with side-effects, e.g.
• **Stateful lambda expressions**
• Where results depend on shared mutable state
  • i.e., state that may change in parallel execution of a pipeline

**Incorrectly compute the factorial of param n using a parallel stream**
Parallel streams should therefore avoid behaviors with side-effects, e.g.

*Stateful lambda expressions*

- Where results depend on shared mutable state
- i.e., state that may change in parallel execution of a pipeline

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

*Define mutable state that’s shared between threads in parallel stream*
Avoiding Programming Hazards in Java Parallel Streams

- Parallel streams should therefore avoid behaviors with side-effects, e.g.
  - **Stateful lambda expressions**
    - Where results depend 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 mult(long n) {
            mTotal *= n;
        }
    }

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

*Race conditions & inconsistent memory visibility may arise from the unsynchronized access to mTotal field*
Avoiding Programming Hazards in Java Parallel Streams

- Parallel streams should therefore avoid behaviors 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(toCollection(ArrayList::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]
Avoiding Programming Hazards in Java Parallel Streams

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

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

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

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

See [github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex11](github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex11)
Avoiding Programming Hazards in Java Parallel Streams

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

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

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

*If a non-concurrent collection is modified while it’s being operated on the results will be chao & insanity!!*

See [docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#peek](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#peek)
Behaviors involving no shared state or side-effects are useful for parallel streams since they needn’t be synchronized explicitly.

Avoiding Programming Hazards in Java Parallel Streams

```java
parallelStream()
    .map(phrase -> searchForPhrase(...))
    .filter(not(SearchResults::isEmpty))
    .collect(toList());
```

See henrikeichenhardt.blogspot.com/2013/06/why-shared-mutable-state-is-root-of-all.html
Behaviors involving no shared state or side-effects are useful for parallel streams since they needn’t be synchronized explicitly.

For example, Java lambda expressions & method references that are “pure functions”

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

Avoiding Programming Hazards in Java Parallel Streams

See [en.wikipedia.org/wiki/Pure_function](en.wikipedia.org/wiki/Pure_function)
Avoiding Programming Hazards in Java Parallel Streams

- 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](docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html)
End of Overview of Java
Parallel Streams: Avoiding Programming Hazards