Common 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

- Understand the structure & functionality of Java streams, e.g.,
  - Fundamentals of streams
  - Benefits of streams
  - Creating a stream
  - Aggregate operations in a stream
  - Applying streams in practice
  - Sequential vs. parallel streams
- Common programming hazards of parallel streams
Common Programming Hazards for Parallel Streams
Common Programming Hazards for Parallel Streams

- Ideally, a behavior’s output in a stream depends only on its input arguments.

See [en.wikipedia.org/wiki/Side_effect_(computer_science)](en.wikipedia.org/wiki/Side_effect_(computer_science))
Ideally, a behavior’s output in a stream depends only on its input arguments.

```java
String capitalize(String s) {
    if (s.length() == 0)
        return s;
    return s.substring(0, 1).toUpperCase()
        + s.substring(1).toLowerCase();
}
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex12
Common Programming Hazards for Parallel Streams

- Ideally, a behavior’s output in a stream depends only on its input arguments.
- Behaviors with side-effects can incur race conditions in parallel streams.

Race conditions arise in software when an application depends on the sequence or timing of threads for it to operate properly.

See en.wikipedia.org/wiki/Race_condition#Software
Common Programming Hazards for Parallel Streams

- Ideally, a behavior’s output in a stream depends only on its input arguments.
- Behaviors with side-effects can incur race conditions in parallel streams, e.g.

```java
class Total {
    public long mTotal = 1;

    public void mult(long n) {
        mTotal *= n;
    }
}
```

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

A buggy attempt to compute the 'nth' factorial in parallel.
Common Programming Hazards for Parallel Streams

• Ideally, a behavior’s output in a stream depends only on its input arguments
• Behaviors with side-effects can incur race conditions in parallel streams, e.g.

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

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

Shared mutable state

See henrikeichenhardt.blogspot.com/2013/06/why-shared-mutable-state-is-root-of-all.html
Common Programming Hazards for Parallel Streams

- Ideally, a behavior’s output in a stream depends only on its input arguments.
- Behaviors with side-effects can incur race conditions in parallel streams, e.g.

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

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

Generate a range of values from 1..n
Common Programming Hazards for Parallel Streams

- Ideally, a behavior’s output in a stream depends only on its input arguments.
- Behaviors with side-effects can incur race conditions in parallel streams, e.g.

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

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

![Diagram showing `rangeClosed`, `parallel`, and `forEach` methods in a flowchart with a note "Run in parallel".]

- Ideally, a behavior’s output in a stream depends only on its input arguments.
- Behaviors with side-effects can incur race conditions in parallel streams, e.g.

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

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

![Diagram showing `rangeClosed`, `parallel`, and `forEach` methods in a flowchart with a note "Run in parallel".]

- Ideally, a behavior’s output in a stream depends only on its input arguments.
- Behaviors with side-effects can incur race conditions in parallel streams, e.g.

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

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

![Diagram showing `rangeClosed`, `parallel`, and `forEach` methods in a flowchart with a note "Run in parallel".]

- Ideally, a behavior’s output in a stream depends only on its input arguments.
- Behaviors with side-effects can incur race conditions in parallel streams, e.g.

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

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

![Diagram showing `rangeClosed`, `parallel`, and `forEach` methods in a flowchart with a note "Run in parallel".]

- Ideally, a behavior’s output in a stream depends only on its input arguments.
- Behaviors with side-effects can incur race conditions in parallel streams, e.g.

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

```java
class Total {
    public long mTotal = 1;
    public void mult(long n) {
        mTotal *= n;
    }
}
```
• Ideally, a behavior’s output in a stream depends only on its input arguments
• Behaviors with side-effects can incur race conditions in parallel streams, e.g.

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

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

**Multiply the running total w/the latest value**
```java
long factorial(long n) {
    Total t = new Total();
    LongStream.rangeClosed(1, n).parallel().forEach(t::mult);
    return t.mTotal;
}

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

Beware of race conditions!!!

See [en.wikipedia.org/wiki/Race_condition#Software](en.wikipedia.org/wiki/Race_condition#Software)
Common Programming Hazards for Parallel Streams

- Ideally, a behavior’s output in a stream depends only on its input arguments.
- Behaviors with side-effects can incur race conditions in parallel streams, e.g.

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

```java
class Total {
    public long mTotal = 1;

    public void mult(long n) {
        mTotal *= n;
    }
}
```

Beware of inconsistent memory visibility.

Common Programming Hazards for Parallel Streams

- Ideally, a behavior’s output in a stream depends only on its input arguments
- Behaviors with side-effects can incur race conditions in parallel streams, e.g.

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

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

Only you can prevent concurrency hazards!

In Java you must avoid these hazards, i.e., the compiler & JVM won’t save you..
End of Common Programming Hazards of Java Parallel Streams