Java Streams: Sequential vs. Parallel Streams

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

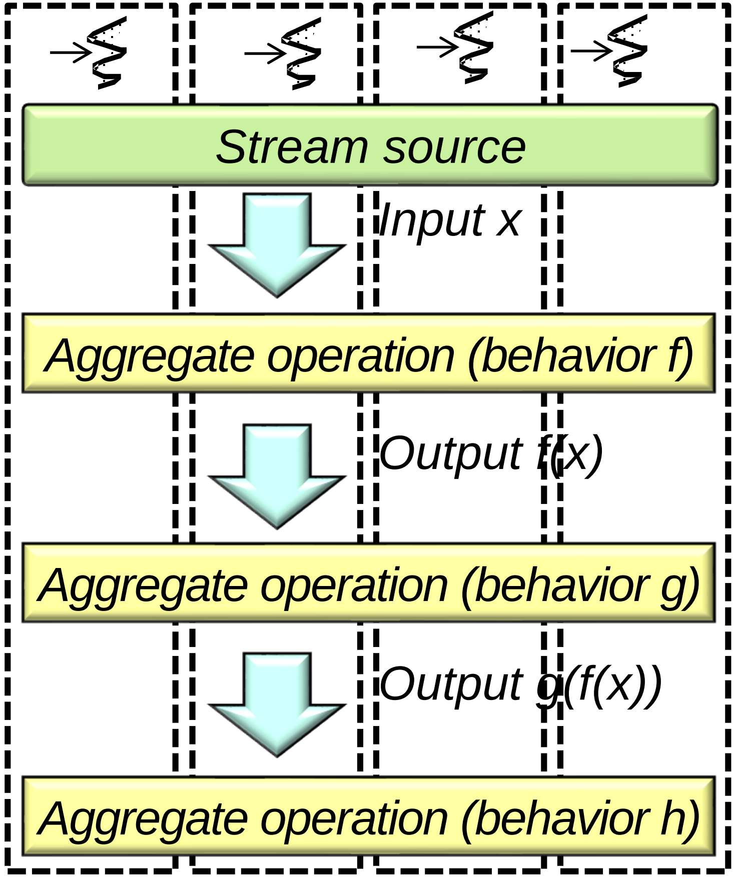
Professor of Computer Science

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

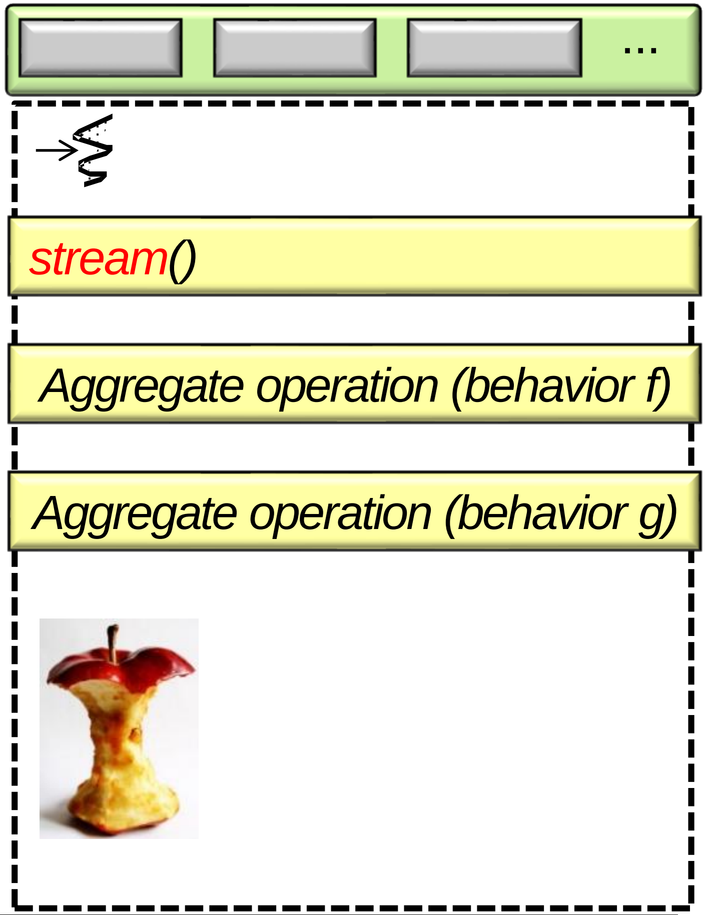

Stream source
Input x
Aggregate operation (behavior f)
Output f(x)
Aggregate operation (behavior g)
Output g(f(x))
Aggregate operation (behavior h)

See radar.oreilly.com/2015/02/java-8-streams-api-and-parallelism.html

Sequential vs. Parallel Streams

Sequential vs. Parallel Streams

- Stream operations run sequentially

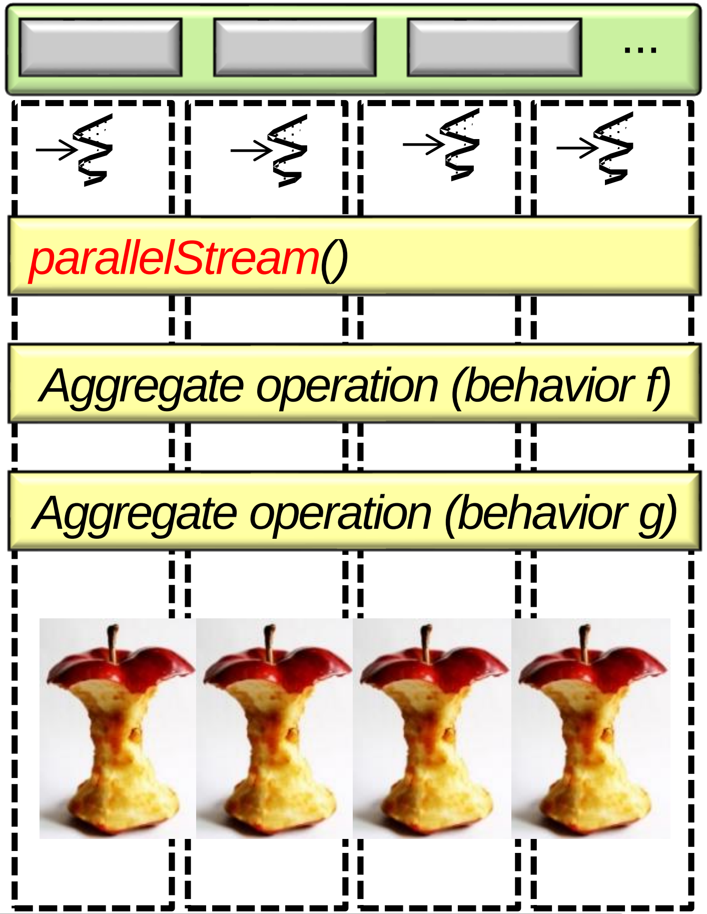

...
stream()
Aggregate operation (behavior f)
Aggregate operation (behavior g)

We'll cover sequential streams first

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

Sequential vs. Parallel Streams

- Stream operations run sequentially or in parallel

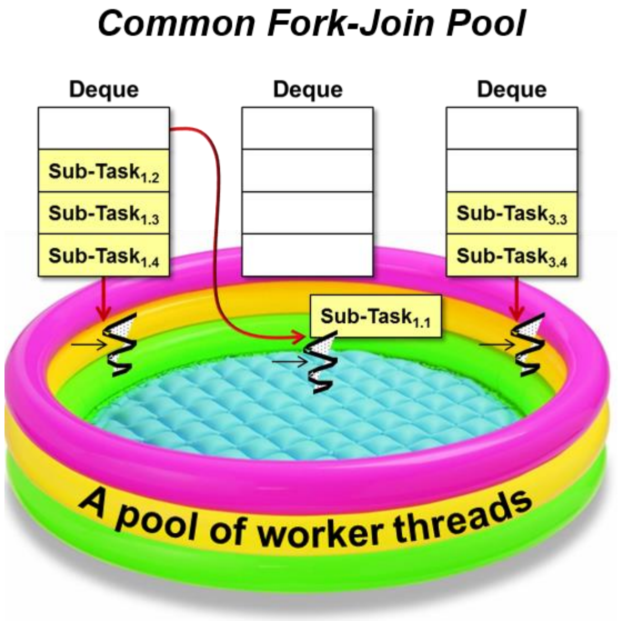

...
parallelStream()
Aggregate operation (behavior f)
Aggregate operation (behavior g)

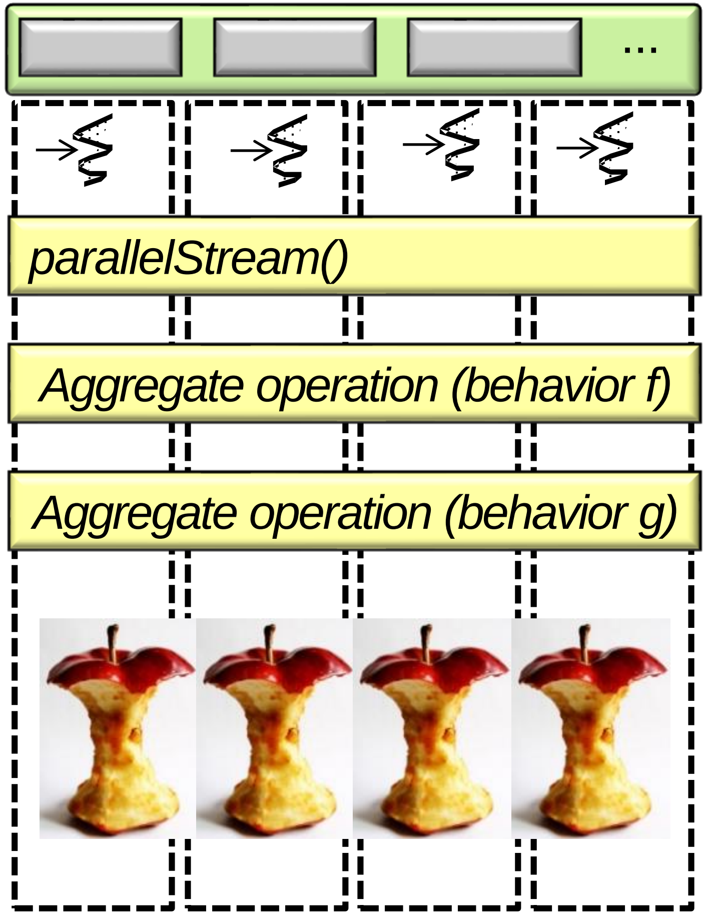
We'll cover parallel streams later

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

Sequential vs. Parallel Streams

- A parallel stream splits its elements into multiple chunk & uses the common fork-join pool to process these chunks independently



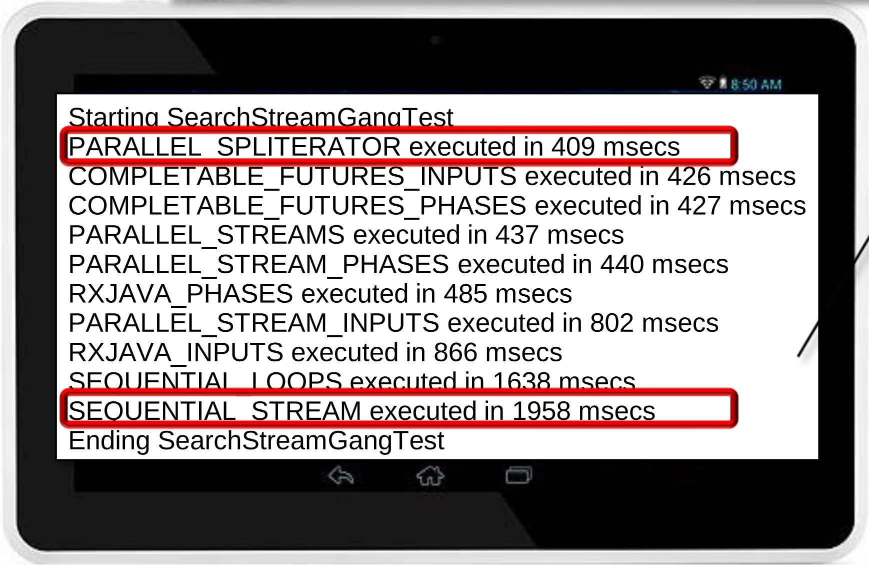

…
parallelStream()
Aggregate operation (behavior f)
Aggregate operation (behavior g)

See dzone.com/articles/common-fork-join-pool-and-streams

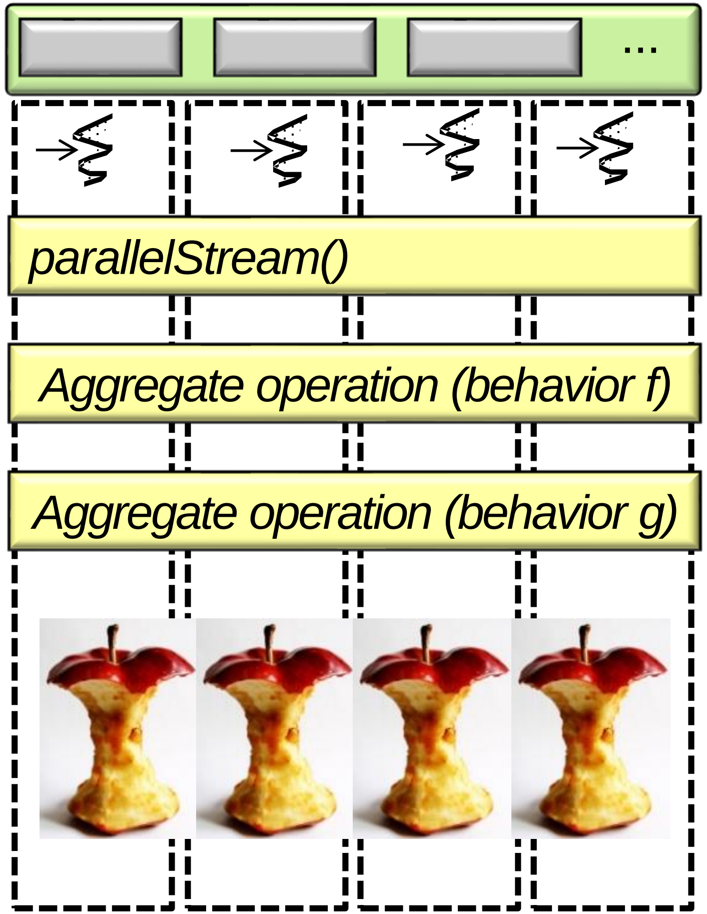
Sequential vs. Parallel Streams

- A parallel stream splits its elements into multiple chunk & uses the common fork-join pool to process these chunks independently

A parallel stream is often more efficient and scalable than a sequential stream.



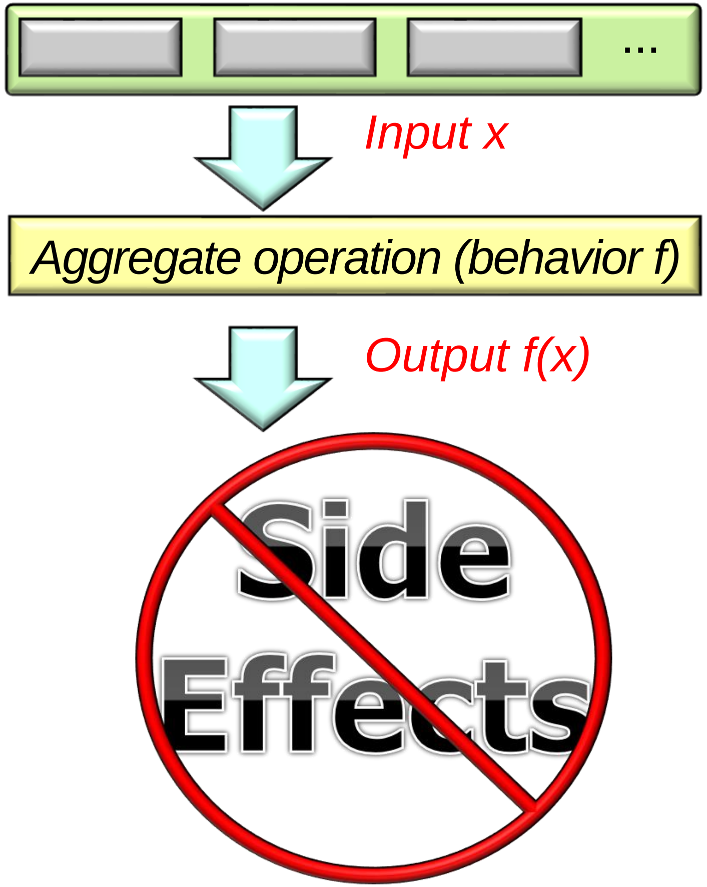
```
Starting SearchStreamGangTest
PARALLEL_SPLITERATOR executed in 409 msecs
COMPLETABLE_FUTURES_INPUTS executed in 426 msecs
COMPLETABLE_FUTURES_PHASES executed in 427 msecs
PARALLEL_STREAMS executed in 437 msecs
PARALLEL_STREAM_PHASES executed in 440 msecs
RXJAVA_PHASES executed in 485 msecs
PARALLEL_STREAM_INPUTS executed in 802 msecs
RXJAVA_INPUTS executed in 866 msecs
SEQUENTIAL_LOOPS executed in 1638 msecs
SEQUENTIAL_STREAM executed in 1958 msecs
Ending SearchStreamGangTest
```


...
parallelStream()
Aggregate operation (behavior f)
Aggregate operation (behavior g)

Tests conducted on a quad-core Lenovo P50 with 32 Gbytes of RAM

Sequential vs. Parallel Streams

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

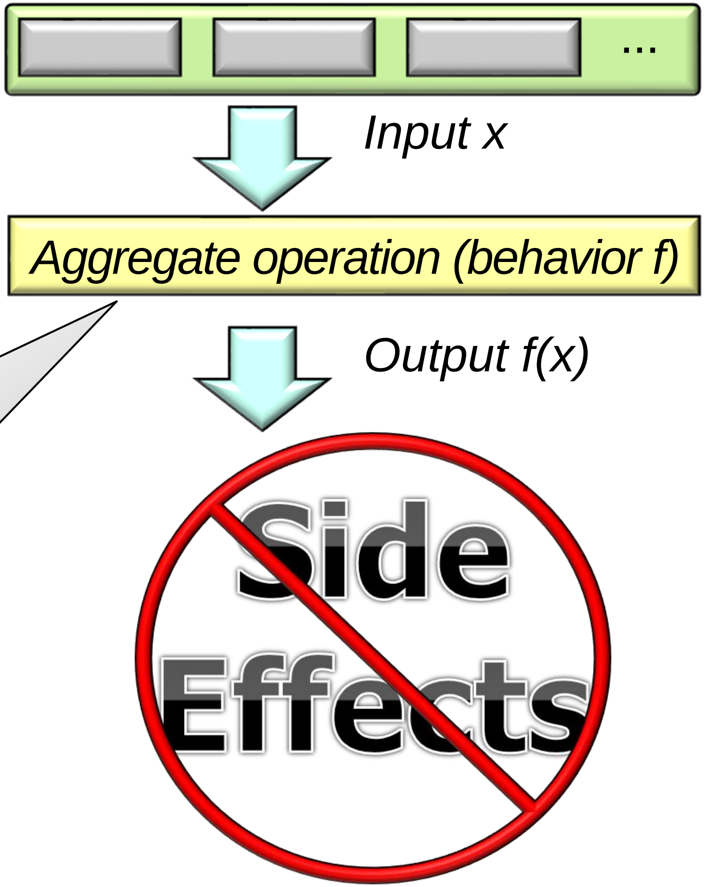


See en.wikipedia.org/wiki/Side_effect_(computer_science)

Sequential vs. Parallel Streams

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

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

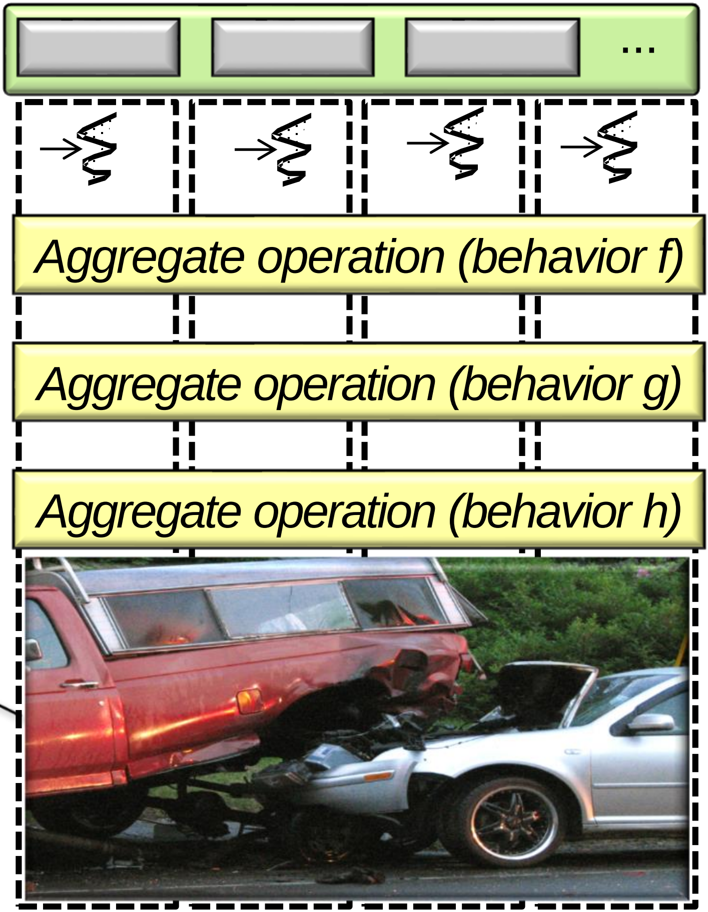


See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex12

Sequential vs. 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*


…
Aggregate operation (behavior f)
Aggregate operation (behavior g)
Aggregate operation (behavior h)

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

Sequential vs. 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.

```
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; }
}
```

A buggy attempt to compute the 'nth' factorial in parallel

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

Sequential vs. 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.

```
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; }
}
```

Shared mutable state

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

Sequential vs. Parallel Streams

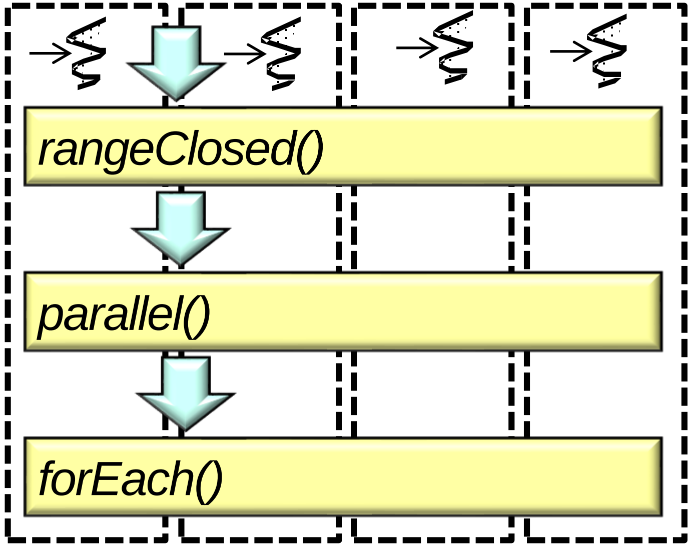
- Ideally, a behavior's output in a stream depends only on its input arguments
  - Behaviors with side-effects can incur race conditions in parallel streams, e.g.

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

Run in parallel

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

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



Sequential vs. 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.

```
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

Sequential vs. Parallel Streams

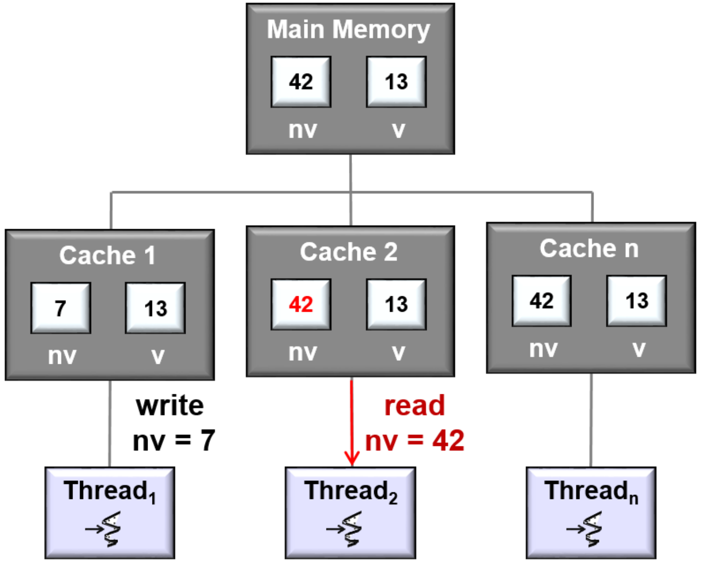
- Ideally, a behavior's output in a stream depends only on its input arguments
  - Behaviors with side-effects can incur race conditions in parallel streams, e.g.

```
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 inconsistent memory visibility



See jeremymanson.blogspot.com/2007/08/atomicity-visibility-and-ordering.html

Sequential vs. 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.

```
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; }
}
```

Only you can prevent concurrency hazards!

In Java *you* must avoid these hazards, i.e., the compiler & JVM won't save you..

End of Java Streams: Sequential vs. Parallel Streams