Learning Objectives in this Part of the Lesson

- Understand parallel stream internals

Why Knowledge of Parallel Streams Matters
Learning Objectives in this Part of the Lesson

- Knowledge of (parallel) streams internals will make you a better Java 8 streams programmer!

See www.ibm.com/developerworks/library/j-java-streams-3-brian-goetz
Why Knowledge of Parallel Streams Matters

- Recall the 3 phases of Java 8 parallel streams

See docs.oracle.com/javase/tutorial/collectionsStreams/parallelism.html
Why Knowledge of Parallel Streams Matters

- Recall the 3 phases of Java 8 parallel streams
- *Splits* its elements into multiple chunks

---

**Stream factory operation ()**

**Input x**

**Intermediate operation (behavior f)**

**Output f(x)**

**Intermediate operation (behavior g)**

**Output g(f(x))**

**Terminal operation (reducer)**
Why Knowledge of Parallel Streams Matters

- Recall the 3 phases of Java 8 parallel streams
  - *Splits* its elements into multiple chunks
  - *Applies* processing on these chunks to run them in a thread pool independently

Stream factory operation ()

```
Input x
```

Intermediate operation (behavior f)

```
Output f(x)
```

Intermediate operation (behavior g)

```
Output g(f(x))
```

Terminal operation (reducer)
Why Knowledge of Parallel Streams Matters

• Recall the 3 phases of Java 8 parallel streams
  • *Splits* its elements into multiple chunks
  • *Applies* processing on these chunks to run them in a thread pool independently
• *Combines* partial results into a single result
Why Knowledge of Parallel Streams Matters

• Recall the 3 phases of Java 8 parallel streams
  • Splits its elements into multiple chunks
  • Applies processing on these chunks to run them in a thread pool independently
  • Combines partial results into a single result

GOD, grant me Serenity to ACCEPT the things I cannot change, Courage to CHANGE the things I can, and Wisdom to know the difference.

It’s important to which of these phases you can control & which you can’t!
Why Knowledge of Parallel Streams Matters

- A parallel stream’s splitting & thread pool mechanisms are often invisible

![Diagram showing stream operations:]

- **Input x**
- **Intermediate operation (behavior f)**: Output f(x)
- **Intermediate operation (behavior g)**: Output g(f(x))
- **Terminal operation (behavior h)**

Stream factory operation ()
Why Knowledge of Parallel Streams Matters

- A parallel stream’s splitting & thread pool mechanisms are often invisible, e.g.
- Java collections have predefined spliterators

```java
public interface Collection<E> {
    default Stream<E> stream() {
        return StreamSupport.stream(spliterator(), false);
    }
    default Spliterator<E> spliterator() {
        return Spliterators.spliterator(this, 0);
    }
}
```

See [blog.logentries.com/2015/10/java-8-introduction-to-parallelism-and-spliterator](blog.logentries.com/2015/10/java-8-introduction-to-parallelism-and-spliterator)
Why Knowledge of Parallel Streams Matters

- A parallel stream’s splitting & thread pool mechanisms are often invisible, e.g.
  - Java collections have predefined spliterators
  - There’s a common fork-join pool

See [www.baeldung.com/java-fork-join](http://www.baeldung.com/java-fork-join)
Why Knowledge of Parallel Streams Matters

• However, programmers can also customize the splitting & thread pool behavior

```java
public interface Spliterator<T> {
    boolean tryAdvance
        (Consumer<? Super T> action);
    Spliterator<T> trySplit();
    long estimateSize();
    int characteristics();
}

public interface ManagedBlocker {
    boolean block()
        throws InterruptedException;
    boolean isReleasable();
}
```

See Parts 2 & 4 of this lesson on “Java 8 Parallel Stream Internals”
Why Knowledge of Parallel Streams Matters

- The *order* in which chunks are processed is non-deterministic
Why Knowledge of Parallel Streams Matters

- The order in which chunks are processed is non-deterministic
- Programmers often have little/no control over how chunks are processed

---

Diagram:

- Stream factory operation ()
- Input x
- Intermediate operation (behavior f)
- Output f(x)
- Intermediate operation (behavior g)
- Output g(f(x))
- Terminal operation (reducer)
Why Knowledge of Parallel Streams Matters

- The order in which chunks are processed is non-deterministic
- Programmers often have little/no control over how chunks are processed
- Non-determinism is useful since it enables optimizations at multiple layers!

e.g., scheduling & execution of tasks via fork-join pool, JVM, hardware cores, etc.
The results of the processing are more deterministic.
Why Knowledge of Parallel Streams Matters

- The *results* of the processing are more deterministic
- Programmers can control how results are presented

Why Knowledge of Parallel Streams Matters

- The *results* of the processing are more deterministic
- Programmers can control how results are presented, e.g.
  - Order is maintained if the source is ordered & aggregate operations are obliged to maintain order

It does not matter whether the stream is parallel or sequential
The results of the processing are more deterministic.

Programmers can control how results are presented, e.g.,

- Order is maintained if the source is ordered & aggregate operations are obliged to maintain order
- e.g., ordered spliterators, ordered collection, static stream factory methods respect “encounter order”

See www.ibm.com/developerworks/library/j-java-streams-3-brian-goetz/index.html#eo
The results of the processing are more deterministic.

Programmers can control how results are presented, e.g.:
- Order is maintained if the source is ordered & aggregate operations are obliged to maintain order.
- Certain intermediate operations affect the ordering.
  - e.g., sorted(), unordered(), skip(), & limit()
Why Knowledge of Parallel Streams Matters

- The *results* of the processing are more deterministic
- Programmers can control how results are presented, e.g.
  - Order is maintained if the source is ordered & aggregate operations are obliged to maintain order
  - Certain intermediate operations effect the ordering
  - Certain terminal operations also effect the ordering
    - e.g., forEach() & forEachOrdered()
Why Knowledge of Parallel Streams Matters

- Intermediate operations are “lazy” & don’t run until a terminal operator is reached

```
Input x
Stream map(Behavior<...> mapper)
Output f(x)
Stream filter(Predicate<...> pred)
Output g(f(x))
R collect(Collector<...> collector)
Output h(g(f(x)))
```

Why Knowledge of Parallel Streams Matters

- Intermediate operations are “lazy” & don’t run until a terminal operator is reached.
- A stream pipeline is built by constructing a linked-list of the stream source & its intermediate operations.

Why Knowledge of Parallel Streams Matters

- Intermediate operations are “lazy” & don’t run until a terminal operator is reached
- A stream pipeline is built by constructing a linked-list of the stream source & its intermediate operations
- When the terminal operation is run, the stream implementation picks an execution plan
  - e.g., the plan is based on properties of the source & the aggregate operations

\[
\text{Input } x \quad \rightarrow \quad \text{Stream map}(\text{Behavior}<\ldots> \text{mapper}) \quad \rightarrow \quad \text{Output } f(x) \\
\quad \rightarrow \quad \text{Stream filter}(\text{Predicate}<\ldots> \text{pred}) \quad \rightarrow \quad \text{Output } g(f(x)) \\
\quad \rightarrow \quad R \text{ collect}(\text{Collector}<\ldots> \text{collector}) \quad \rightarrow \quad \text{Output } h(g(f(x)))
\]

End of Java 8 Parallel Stream Internals (Part 1)