Learning Objectives in this Part of the Lesson

- Understand parallel stream internals

See www.ibm.com/developerworks/library/j-java-streams-3-brian-goetz
Learning Objectives in this Part of the Lesson

- Understand parallel stream internals, e.g.
- Know what can change & what can’t

Grant me the Serenity to accept the things I cannot change. The Courage to change the things I can and the Wisdom to know the difference.
Why Knowledge of Parallel Streams Matters
Why Knowledge of Parallel Streams Matters

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

When performance is critical, it's important to understand how streams work internally.

Why Knowledge of Parallel Streams Matters

- Recall the 3 phases of a Java 8 parallel stream

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

• Recall the 3 phases of a Java 8 parallel stream
  • *Splits* its elements into multiple chunks

```
Input x

Intermediate operation (behavior f)

Output f(x)

Intermediate operation (behavior g)

Output g(f(x))

Terminal operation (reducer)
```

Stream factory operation ()
Why Knowledge of Parallel Streams Matters

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

- Recall the 3 phases of a Java 8 parallel stream
  - 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 a Java 8 parallel stream
  - **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!
Parallel Stream Splitting & Thread Pool Mechanisms
Parallel Stream Splitting & Thread Pool Mechanisms

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

```
Stream factory operation ()

Input x

Intermediate operation (behavior f)

Output f(x)

Intermediate operation (behavior g)

Output g(f(x))

Terminal operation (behavior h)
```
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
A parallel stream’s splitting & thread pool mechanisms are often invisible, e.g.

- Java collections have predefined spliterators
- A common fork-join pool is used by default

See [www.baeldung.com/java-fork-join](http://www.baeldung.com/java-fork-join)
Parallel Stream Splitting & Thread Pool Mechanisms

- However, programmers can customize the behavior of splitting & thread pools

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

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

See Parts 2 & 4 of this lesson on “Java 8 Parallel Stream Internals”
Parallel Stream Ordering
The order in which chunks are processed is non-deterministic.
The order in which chunks are processed is non-deterministic.

Programmers have little/no control over how chunks are processed.

Parallel Stream Ordering

Input $x$

Intermediate operation (behavior $f$)

Output $f(x)$

Intermediate operation (behavior $g$)

Output $g(f(x))$

Terminal operation (reducer)

Stream factory operation ()
Parallel Stream Ordering

- The *order* in which chunks are processed is non-deterministic
- Programmers 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
Parallel Stream Ordering

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

Parallel Stream Ordering

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

*It doesn’t matter whether the stream is parallel or sequential*

Parallel Stream Ordering

- The results of the processing are more deterministic
- Programmers can control how results are presented
  - Order is maintained if the source is ordered & the aggregate operations used are obliged to maintain order
- Ordered spliterators, ordered collections, & static stream factory methods respect “encounter order”

```java
List<Integer> list = Arrays.asList(1, 2, ...);
Integer[] doubledList = list
  .parallelStream()
  .filter(x -> x % 2 == 0)
  .map(x -> x * 2)
  .toArray(Integer[]::new);
```

The encounter order is [1, 2, 3, 4, ...] since list is ordered

See [github.com/douglas craigschmidt/LiveLessons/tree/master/Java8/ex21](https://github.com/douglas craigschmidt/LiveLessons/tree/master/Java8/ex21)
Parallel Stream Ordering

- The *results* of the processing are more deterministic
- Programmers can control how results are presented
- Order is maintained if the source is ordered & the aggregate operations used are obliged to maintain order
- Ordered spliterators, ordered collections, & static stream factory methods respect “encounter order”

List<Integer> list = Arrays.asList(1, 2, ...);

Integer[] doubledList = list
    .parallelStream()
    .filter(x -> x % 2 == 0)
    .map(x -> x * 2)
    .toArray(Integer[]::new);

*The result must be [2, 4, ...]*

The results of the processing are more deterministic

Programmers can control how results are presented

Order is maintained if the source is ordered & the aggregate operations used are obliged to maintain order

Ordered spliterators, ordered collections, & static stream factory methods respect “encounter order”

Unordered collections don’t need to respect “encounter order”

Set<Integer> set = new HashSet<>(Arrays.asList(1, 2, ...);

A HashSet is unordered

Integer[] doubledSet = set.parallelStream()
.filter(x -> x % 2 == 0)
.map(x -> x * 2)
.toArray(Integer[]::new);

Parallel Stream Ordering

• The results of the processing are more deterministic
• Programmers can control how results are presented
  • Order is maintained if the source is ordered & the aggregate operations used are obliged to maintain order
  • Ordered spliterators, ordered collections, & static stream factory methods respect “encounter order”
• Unordered collections don’t need to respect “encounter order”

```
Set<Integer> set = new HashSet<>(Arrays.asList(1, 2, ...));
Integer[] doubledSet = set.parallelStream()
  .filter(x -> x % 2 == 0)
  .map(x -> x * 2)
  .toArray(Integer[]::new);
```

This code runs faster since encounter order need not be maintained

The results of the processing are more deterministic.

Programmers can control how results are presented.

Order is maintained if the source is ordered & the aggregate operations used are obliged to maintain order.

Certain intermediate operations effect ordering behavior.
The results of the processing are more deterministic:

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

The result must be `[2, 4, ...]`, but the code is slow due to `limit()` & `distinct()` "stateful" semantics in parallel streams.

```java
List<Integer> list = Arrays.asList(1, 2, ...);
Integer[] doubledList = list
  .parallelStream()
  .distinct()
  .filter(x -> x % 2 == 0)
  .map(x -> x * 2)
  .limit(sOutputLimit)
  .toArray(Integer[]::new);
```

Parallel Stream Ordering

- The *results* of the processing are more deterministic
- Programmers can control how results are presented
  - Order is maintained if the source is ordered & the aggregate operations used are obliged to maintain order
  - Certain intermediate operations effect ordering behavior
    - e.g., sorted(), unordered(), skip(), & limit()

```java
List<Integer> list = Arrays.asList(1, 2, ...);
Integer[] doubledList = list
  .parallelStream()
  .unordered()
  .distinct()
  .filter(x -> x % 2 == 0)
  .map(x -> x * 2)
  .limit(sOutputLimit)
  .toArray(Integer[]::new);
```

This code runs faster since stream is unordered & thus limit() & distinct() incur less overhead

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex21
Parallel Stream Ordering

- The *results* of the processing are more deterministic
- Programmers can control how results are presented
  - Order is maintained if the source is ordered & the aggregate operations used are obliged to maintain order
  - Certain intermediate operations effect ordering behavior
  - Certain terminal operations also effect ordering behavior
Parallel Stream Ordering

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

```java
List<Integer> list = Arrays.asList(1, 2, ...);
ConcurrentLinkedQueue<Integer> queue = new ConcurrentLinkedQueue<>();
list
    .parallelStream()
    .distinct()
    .filter(x -> x % 2 == 0)
    .map(x -> x * 2)
    .limit(sOutputLimit)
    .forEachOrdered(queue::add);
```

The results of the processing are more deterministic.

Programmers can control how results are presented.

Order is maintained if the source is ordered & the aggregate operations used are obliged to maintain order.

Certain intermediate operations effect ordering behavior.

Certain terminal operations also effect ordering behavior.

E.g., forEachOrdered() & forEach()

```java
List<Integer> list = Arrays.asList(1, 2, ...);
ConcurrentLinkedQueue<Integer> queue = new ConcurrentLinkedQueue<>();
list
  .parallelStream()
  .distinct()
  .filter(x -> x % 2 == 0)
  .map(x -> x * 2)
  .limit(sOutputLimit)
  .forEach(queue::add);
```

End of Java 8 Parallel Stream Internals (Part 1)