Java Parallel Streams Internals: Demo'ing Collector Performance

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 parallel stream internals, e.g.
  - Know what can change & what can't
  - Partition a data source into "chunks"
  - Process chunks in parallel via the common fork-join pool
  - Configure the Java parallel stream common fork-join pool
  - Perform a reduction to combine partial results into a single result
  - Recognize key behaviors & differences of non-concurrent & concurrent collectors
  - Be aware of non-concurrent & concurrent collector APIs
  - Grok performance variance between concurrent & non-concurrent collectors

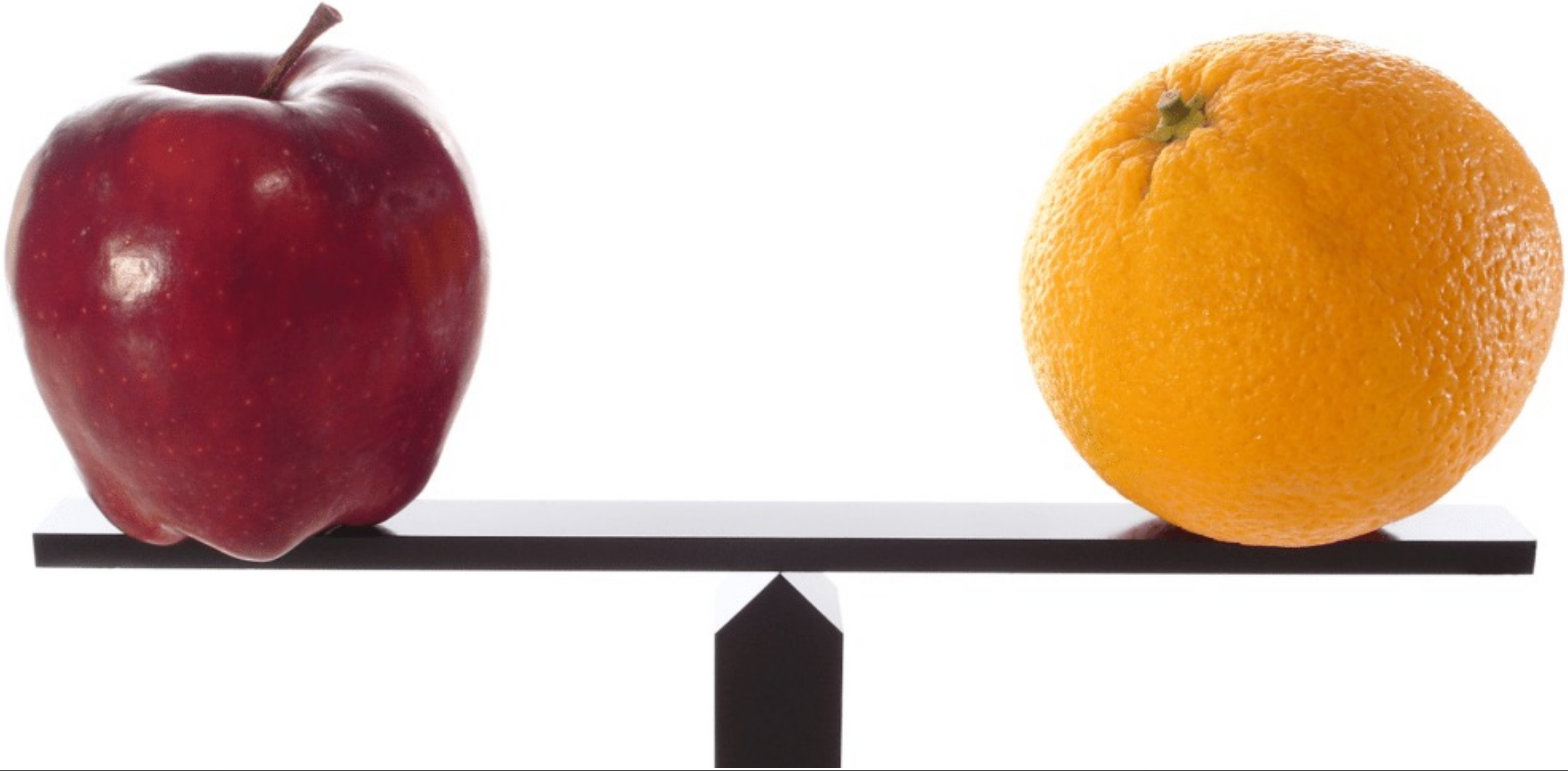
```
Starting collector tests for 1000 words..printing results
    21 msecs: sequential timeStreamCollectToSet()
    30 msecs: parallel timeStreamCollectToSet()
    39 msecs: sequential timeStreamCollectToConcurrentSet()
    59 msecs: parallel timeStreamCollectToConcurrentSet()
...
Starting collector tests for 100000 words..printing results
   219 msecs: parallel timeStreamCollectToConcurrentSet()
   364 msecs: parallel timeStreamCollectToSet()
   657 msecs: sequential timeStreamCollectToSet()
   804 msecs: sequential timeStreamCollectToConcurrentSet()
Starting collector tests for 883311 words..printing results
  1782 msecs: parallel timeStreamCollectToConcurrentSet()
  3010 msecs: parallel timeStreamCollectToSet()
  6169 msecs: sequential timeStreamCollectToSet()
  7652 msecs: sequential timeStreamCollectToConcurrentSet()
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex36

Demonstrating Collector Performance

Demonstrating Collector Performance

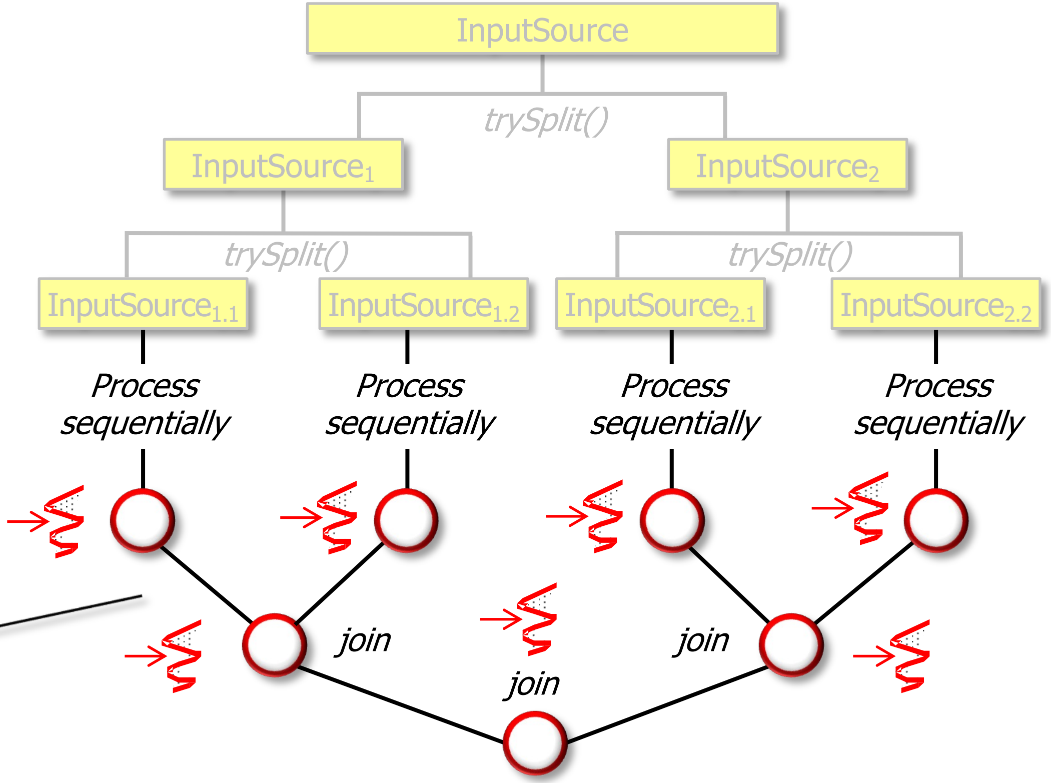
- Concurrent & non-concurrent collectors perform differently when used in parallel & sequential streams on different input sizes



See prior lessons on "*Java Parallel Streams Internals: Non-Concurrent and Concurrent Collectors*"

Demonstrating Collector Performance

- A non-concurrent collector operates by merging sub-results

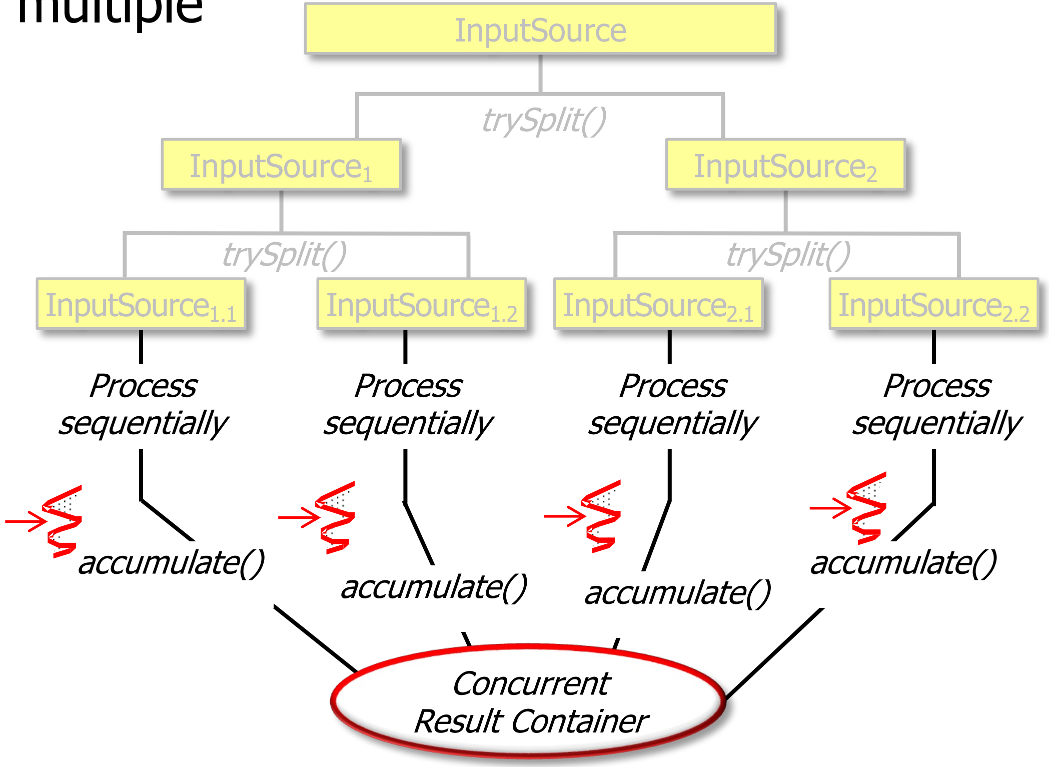


Different threads operate on different instances of the intermediate result containers

See docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html#toSet

Demonstrating Collector Performance

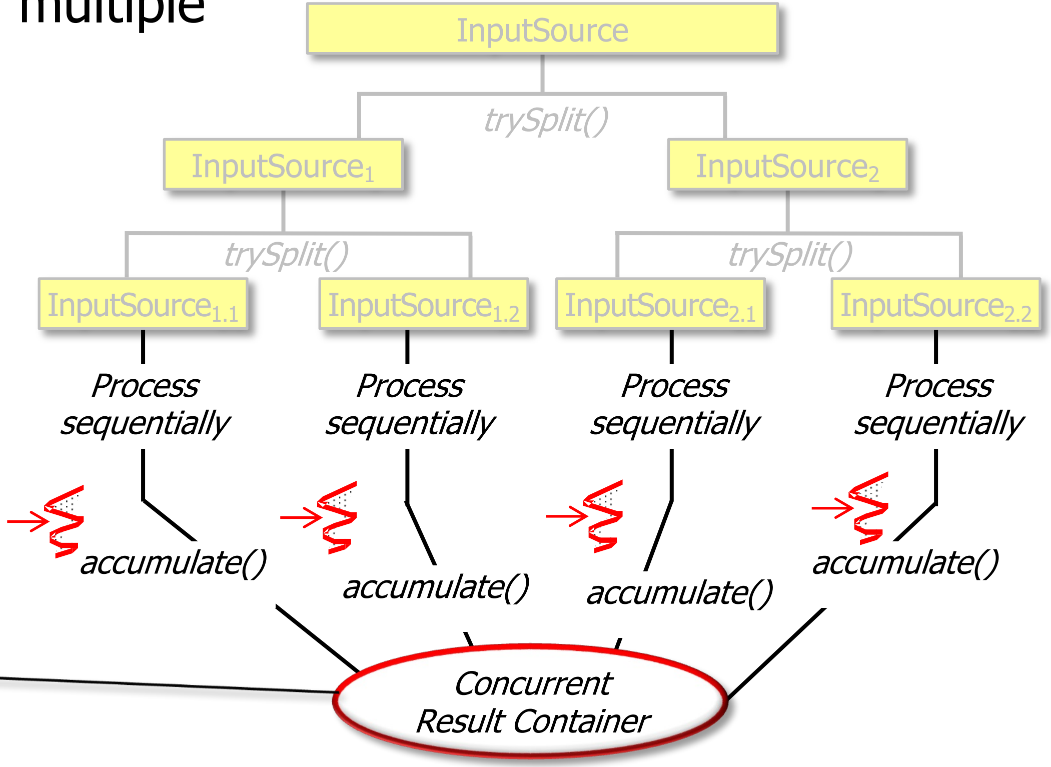
- A concurrent collector creates one concurrent mutable result container & accumulates elements into it from multiple threads in a parallel stream



See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.KeySetView.html

Demonstrating Collector Performance

- A concurrent collector creates one concurrent mutable result container & accumulates elements into it from multiple threads in a parallel stream



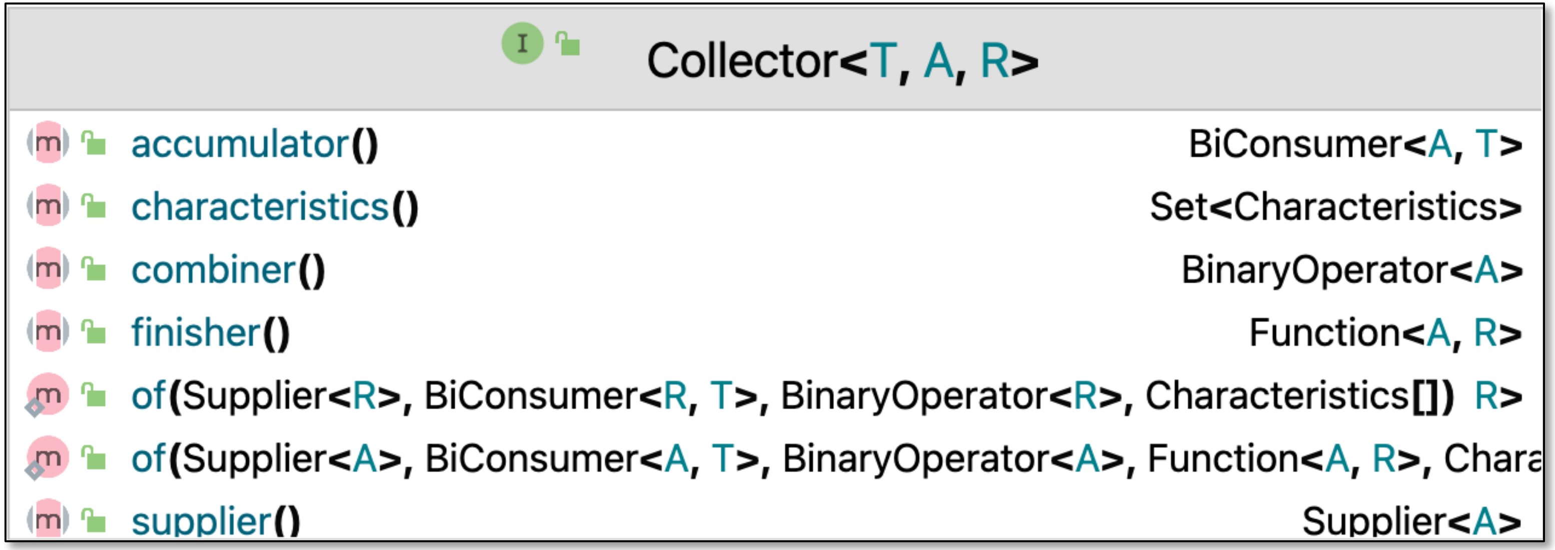
Thus there's no need to merge any intermediate sub-results!

Demonstrating Collector Performance

- The ex36 example showcases the different in performance of two collectors

Demonstrating Collector Performance

- The ex36 example showcases the different in performance of two collectors
  - Various Set collectors defined by the Java Collectors utility class

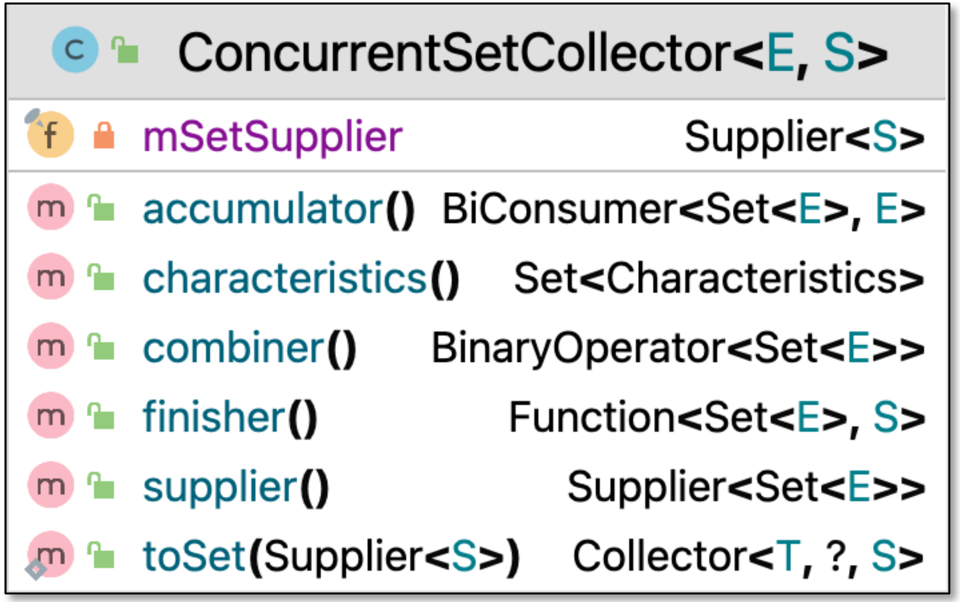


```
Collector<T, A, R>
accumulator()                                                          BiConsumer<A, T>
characteristics()                                                      Set<Characteristics>
combiner()                                                             BinaryOperator<A>
finisher()                                                             Function<A, R>
of(Supplier<R>, BiConsumer<R, T>, BinaryOperator<R>, Characteristics[])  R>
of(Supplier<A>, BiConsumer<A, T>, BinaryOperator<A>, Function<A, R>, Chara
supplier()                                                             Supplier<A>
```

See docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html

Demonstrating Collector Performance

- The ex36 example showcases the different in performance of two collectors
  - Various Set collectors defined by the Java Collectors utility class
  - The ConcurrentSetCollector



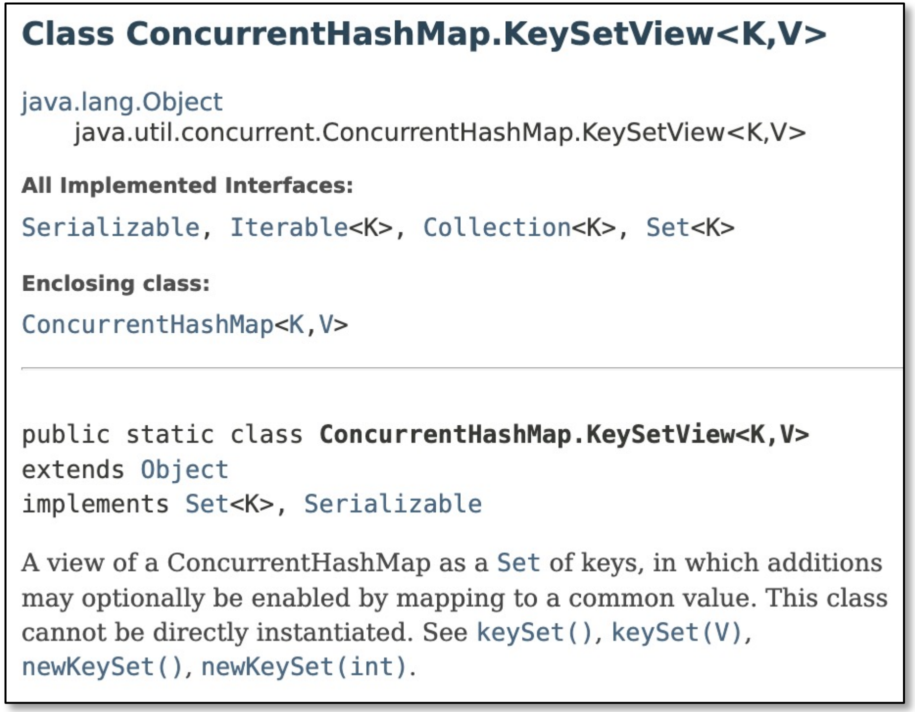
ConcurrentSetCollector<E, S>

mSetSupplier	Supplier<S>
accumulator()	BiConsumer<Set<E>, E>
characteristics()	Set<Characteristics>
combiner()	BinaryOperator<Set<E>>
finisher()	Function<Set<E>, S>
supplier()	Supplier<Set<E>>
toSet(Supplier<S>)	Collector<T, ?, S>

See Java8/ex36/src/main/java/utils/ConcurrentSetCollector.java

Demonstrating Collector Performance

- The ex36 example showcases the different in performance of two collectors
  - Various Set collectors defined by the Java Collectors utility class
  - The ConcurrentSetCollector
    - Applied in conjunction with ConcurrentHashMap. KeySetView



Class ConcurrentHashMap.KeySetView<K,V>

java.lang.Object
    java.util.concurrent.ConcurrentHashMap.KeySetView<K,V>

All Implemented Interfaces:

`Serializable`, `Iterable<K>`, `Collection<K>`, `Set<K>`

Enclosing class:

`ConcurrentHashMap<K,V>`

```
public static class ConcurrentHashMap.KeySetView<K,V>
extends Object
implements Set<K>, Serializable
```

A view of a ConcurrentHashMap as a Set of keys, in which additions may optionally be enabled by mapping to a common value. This class cannot be directly instantiated. See `keySet()`, `keySet(V)`, `newKeySet()`, `newKeySet(int)`.

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.KeySetView.html

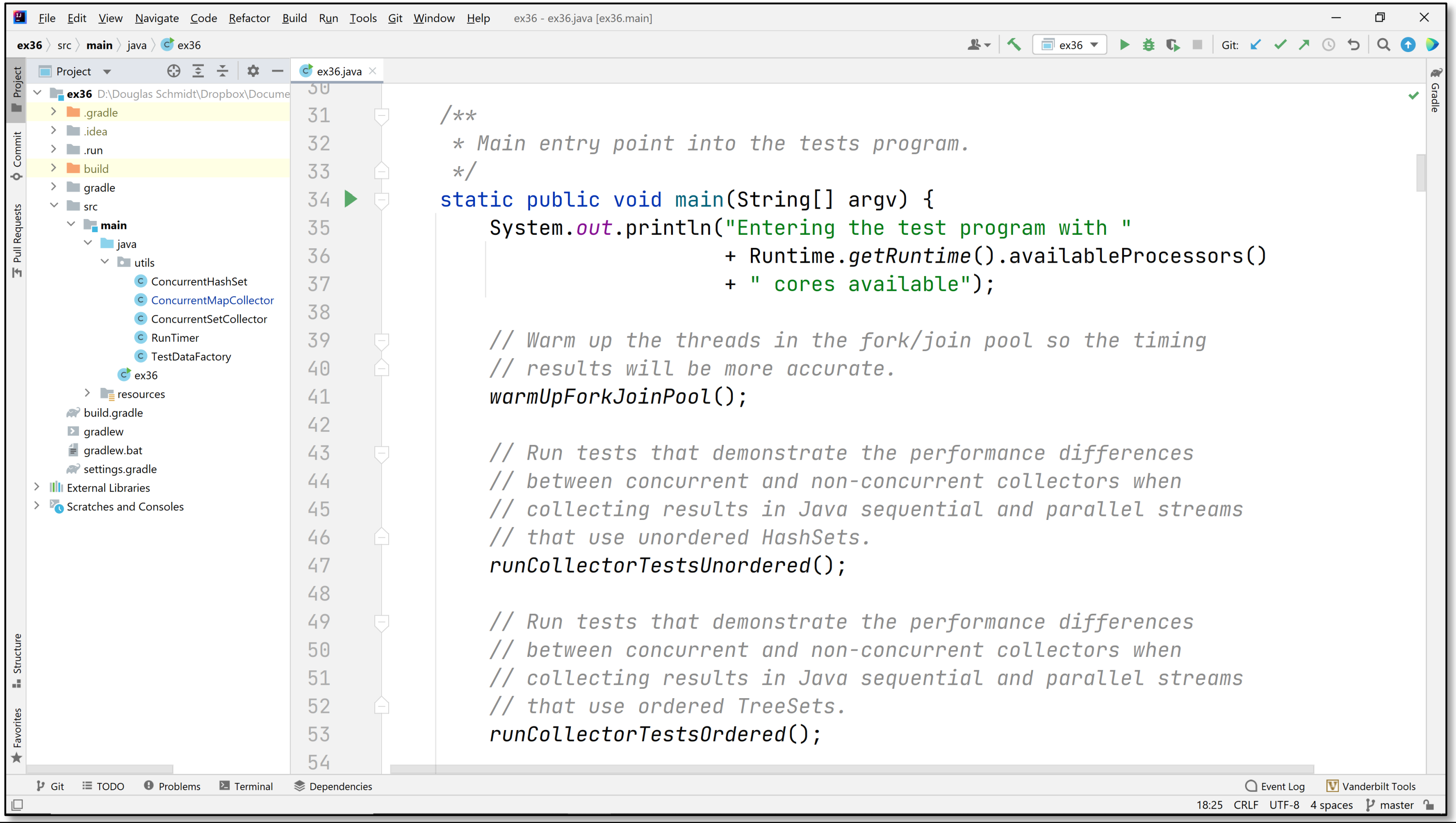
Demonstrating Collector Performance

- Results show collector differences become more significant as input grows

```
Starting collector tests for 1000 words..printing results
    21 msecs: sequential timeStreamCollectToSet()
    30 msecs: parallel timeStreamCollectToSet()
    39 msecs: sequential timeStreamCollectToConcurrentSet()
    59 msecs: parallel timeStreamCollectToConcurrentSet()
...
Starting collector tests for 100000 words....printing results
   219 msecs: parallel timeStreamCollectToConcurrentSet()
   364 msecs: parallel timeStreamCollectToSet()
   657 msecs: sequential timeStreamCollectToSet()
   804 msecs: sequential timeStreamCollectToConcurrentSet()
Starting collector tests for 883311 words....printing results
  1782 msecs: parallel timeStreamCollectToConcurrentSet()
  3010 msecs: parallel timeStreamCollectToSet()
  6169 msecs: sequential timeStreamCollectToSet()
  7652 msecs: sequential timeStreamCollectToConcurrentSet()
```

See upcoming lessons on "*When [Not] to Use Parallel Streams*"

Demonstrating Collector Performance



```java
    /**
     * Main entry point into the tests program.
     */
    static public void main(String[] argv) {
        System.out.println("Entering the test program with "
                           + Runtime.getRuntime().availableProcessors()
                           + " cores available");

        // Warm up the threads in the fork/join pool so the timing
        // results will be more accurate.
        warmUpForkJoinPool();

        // Run tests that demonstrate the performance differences
        // between concurrent and non-concurrent collectors when
        // collecting results in Java sequential and parallel streams
        // that use unordered HashSets.
        runCollectorTestsUnordered();

        // Run tests that demonstrate the performance differences
        // between concurrent and non-concurrent collectors when
        // collecting results in Java sequential and parallel streams
        // that use ordered TreeSets.
        runCollectorTestsOrdered();
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex36

End of Java Parallel Streams Internals: Demo‘ing Collector Performance