Java Parallel Streams Internals: Non-Concurrent & Concurrent Collectors (Part 1)

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

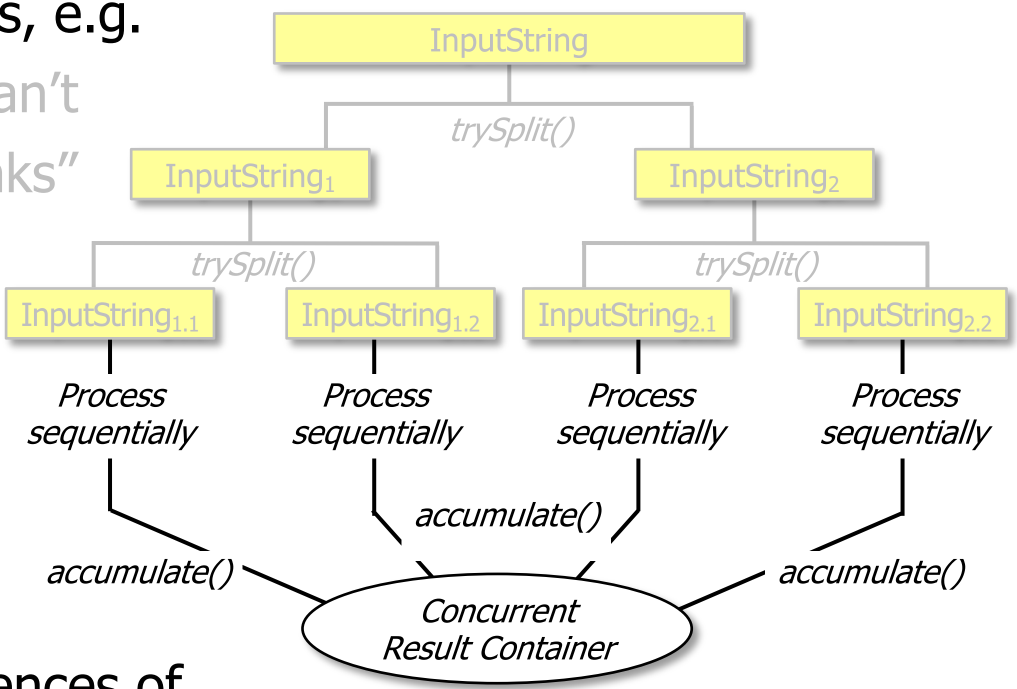
Professor of Computer Science

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



Overview of Concurrent & Non-Concurrent Collectors

Overview of Concurrent & Non-Concurrent Collectors

- Collector defines an interface whose implementations can accumulate input elements in a mutable result container

Interface Collector<T,A,R>

Type Parameters:

`T` - the type of input elements to the reduction operation

`A` - the mutable accumulation type of the reduction operation (often hidden as an implementation detail)

`R` - the result type of the reduction operation

```
public interface Collector<T,A,R>
```

A mutable reduction operation that accumulates input elements into a mutable result container, optionally transforming the accumulated result into a final representation after all input elements have been processed. Reduction operations can be performed either sequentially or in parallel.

Examples of mutable reduction operations include: accumulating elements into a `Collection`; concatenating strings using a `StringBuilder`; computing summary information about elements such as sum, min, max, or average; computing "pivot table" summaries such as "maximum valued transaction by seller", etc. The class `Collectors` provides implementations of many common mutable reductions.

A `Collector` is specified by four functions that work together to accumulate entries into a mutable result container, and optionally perform a final transform on the result. They are:

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

Overview of Concurrent & Non-Concurrent Collectors

- Collector implementations can either be concurrent or non-concurrent based on their characteristics

Enum Collector.Characteristics

java.lang.Object
    java.lang.Enum<Collector.Characteristics>
        java.util.stream.Collector.Characteristics

All Implemented Interfaces:

`Serializable, Comparable<Collector.Characteristics>`

Enclosing interface:

`Collector<T,A,R>`

```
public static enum Collector.Characteristics
extends Enum<Collector.Characteristics>
```

Characteristics indicating properties of a `Collector`, which can be used to optimize reduction implementations.

Enum Constant Summary

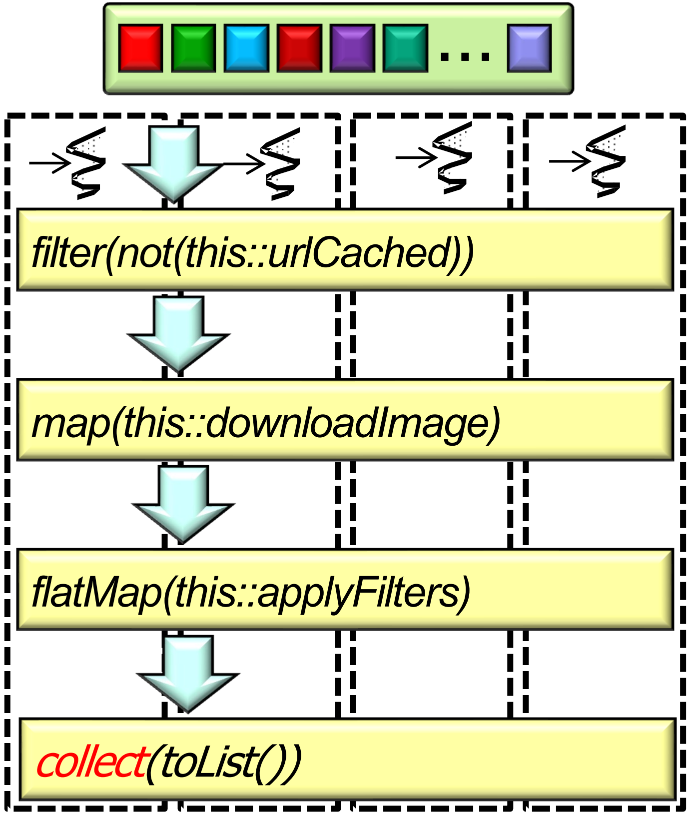
Enum Constants

Enum Constant and Description
`CONCURRENT` Indicates that this collector is *concurrent*, meaning that the result container can support the accumulator function being called concurrently with the same result container from multiple threads.
`IDENTITY_FINISH` Indicates that the finisher function is the identity function and can be elided.
`UNORDERED` Indicates that the collection operation does not commit to preserving the encounter order of input elements.

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

Overview of Concurrent & Non-Concurrent Collectors

- Collector implementations can either be concurrent or non-concurrent based on their characteristics
  - This distinction is only relevant for *parallel* streams



See *"Java Streams: Introducing Non-Concurrent Collectors"*

Overview of Concurrent & Non-Concurrent Collectors

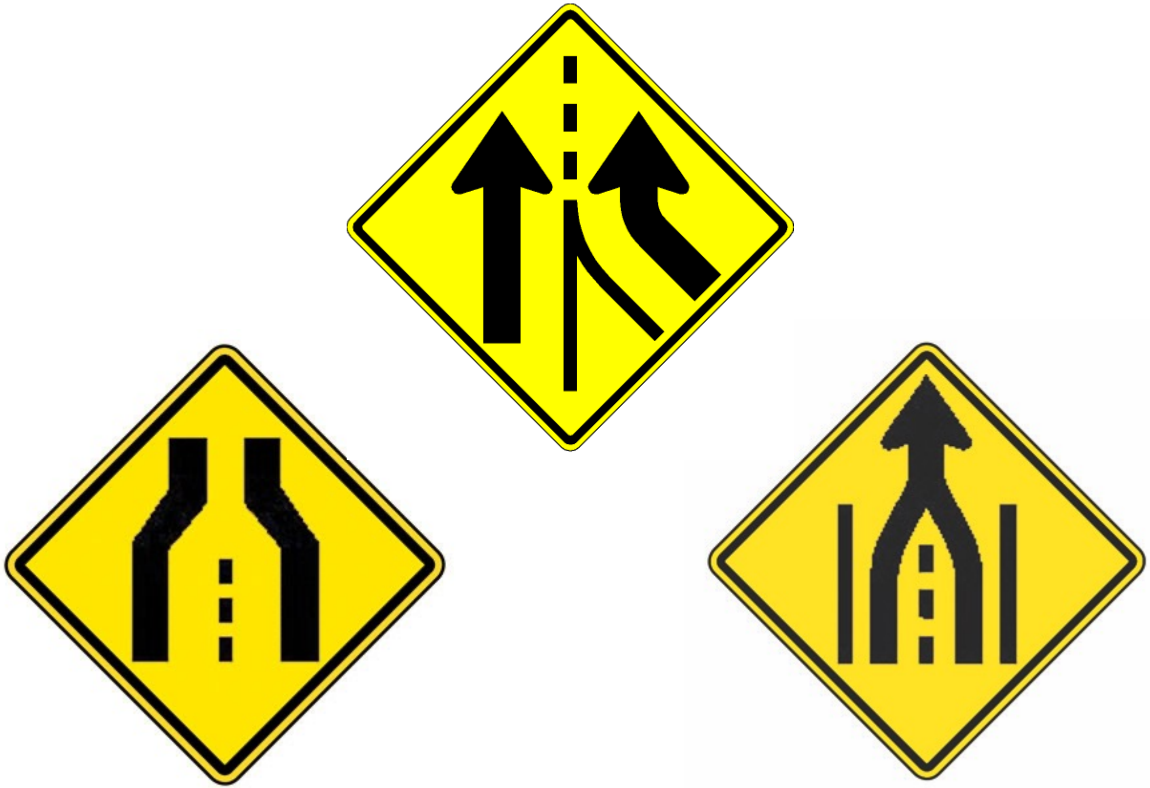
- Collector implementations can either be concurrent or non-concurrent based on their characteristics
  - This distinction is only relevant for *parallel* streams
  - A non-concurrent collector can be used for either a sequential stream or a parallel stream!

We just focus on parallel streams in this lesson

Structure & Functionality of Non-Concurrent Collectors

Structure & Functionality of Non-Concurrent Collectors

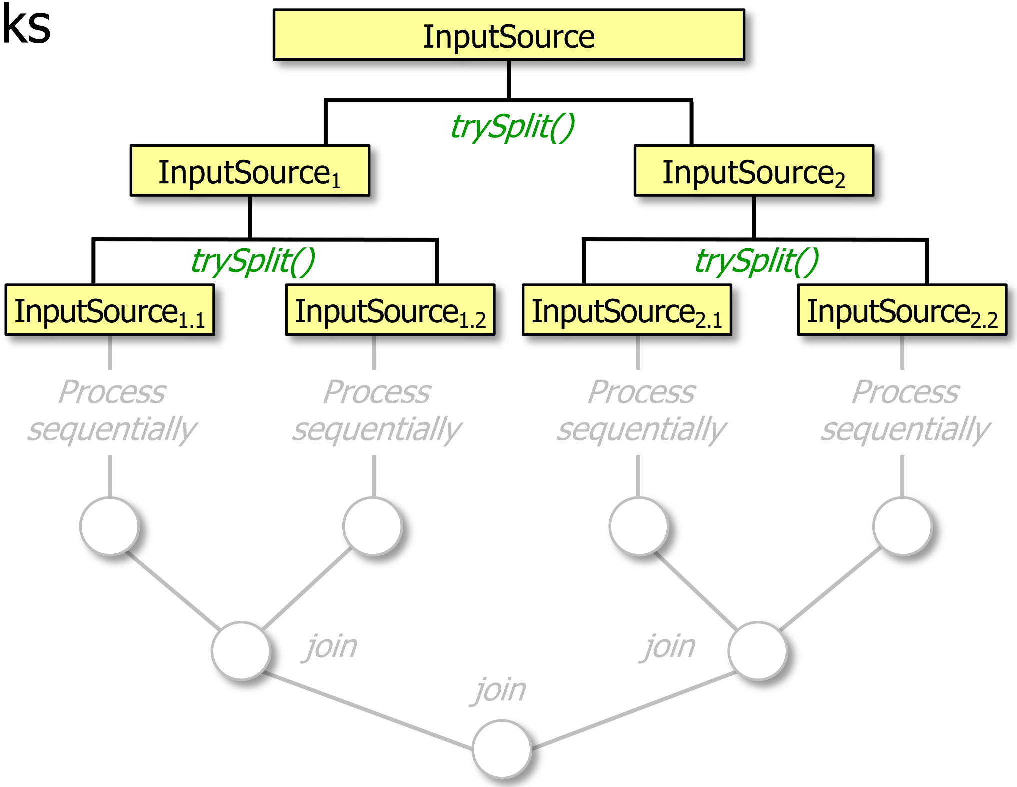
- A non-concurrent collector operates by merging sub-results



See stackoverflow.com/questions/22350288/parallel-streams-collectors-and-thread-safety

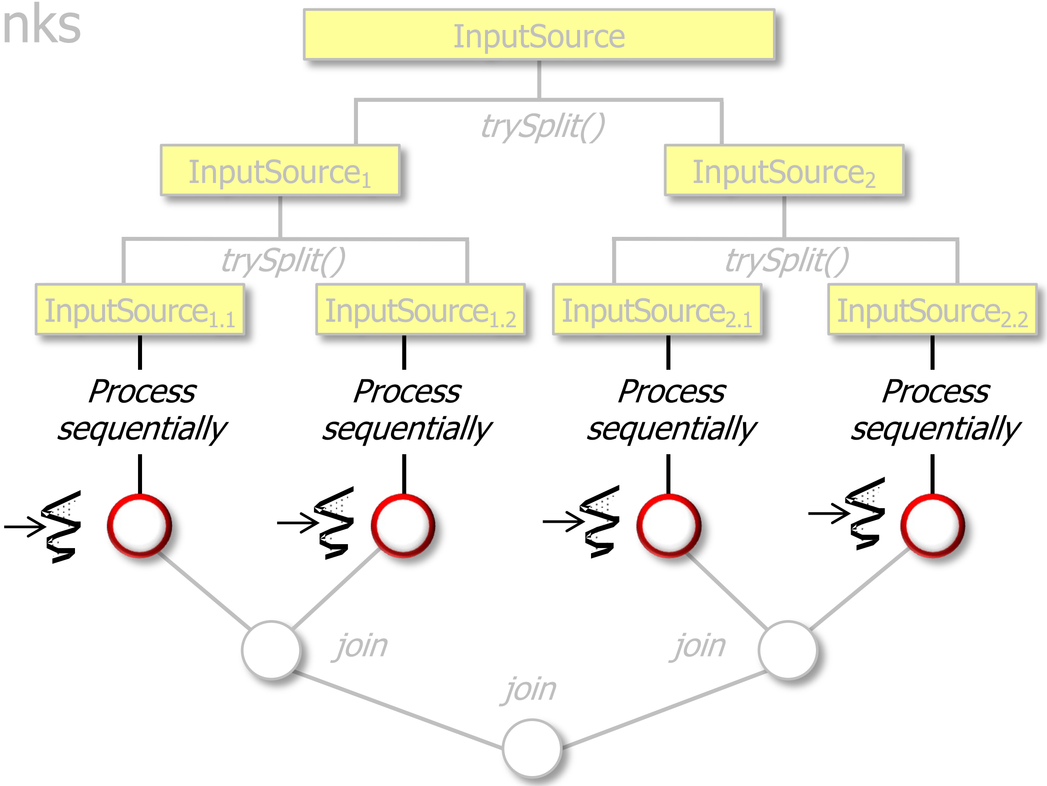
Structure & Functionality of Non-Concurrent Collectors

- A non-concurrent collector operates by merging sub-results
  - The input is partitioned into chunks



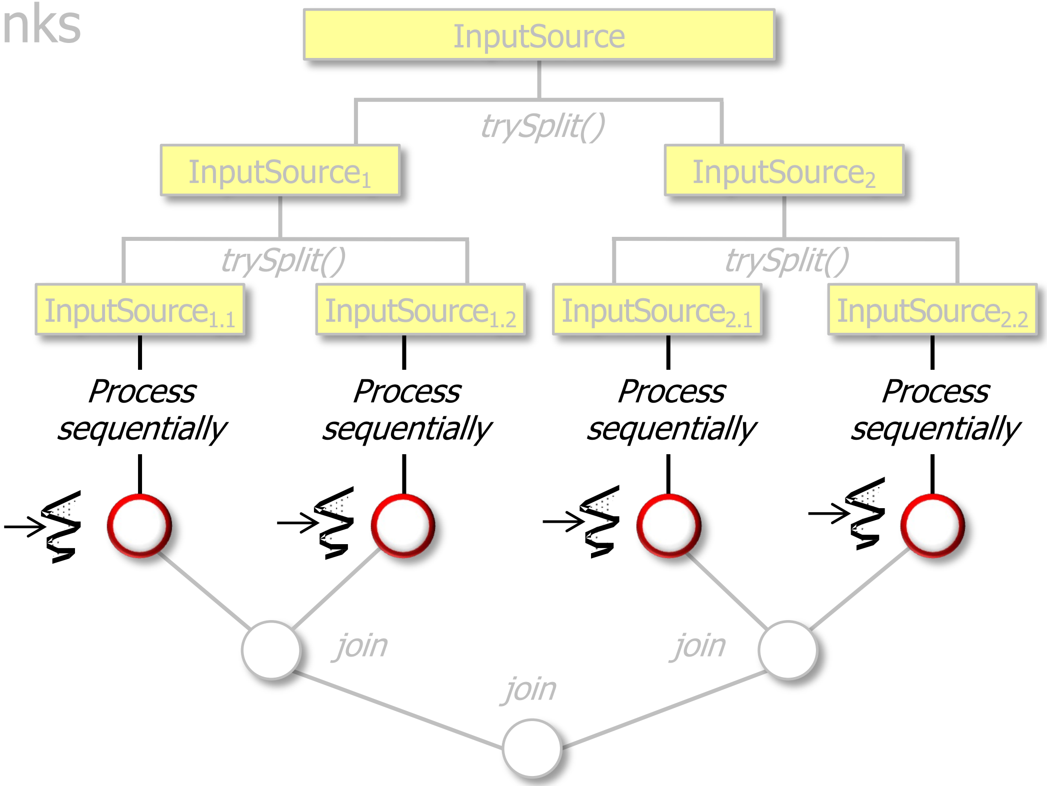
Structure & Functionality of Non-Concurrent Collectors

- A non-concurrent collector operates by merging sub-results
  - The input is partitioned into chunks
  - Each chunk runs in parallel in the common fork-join pool



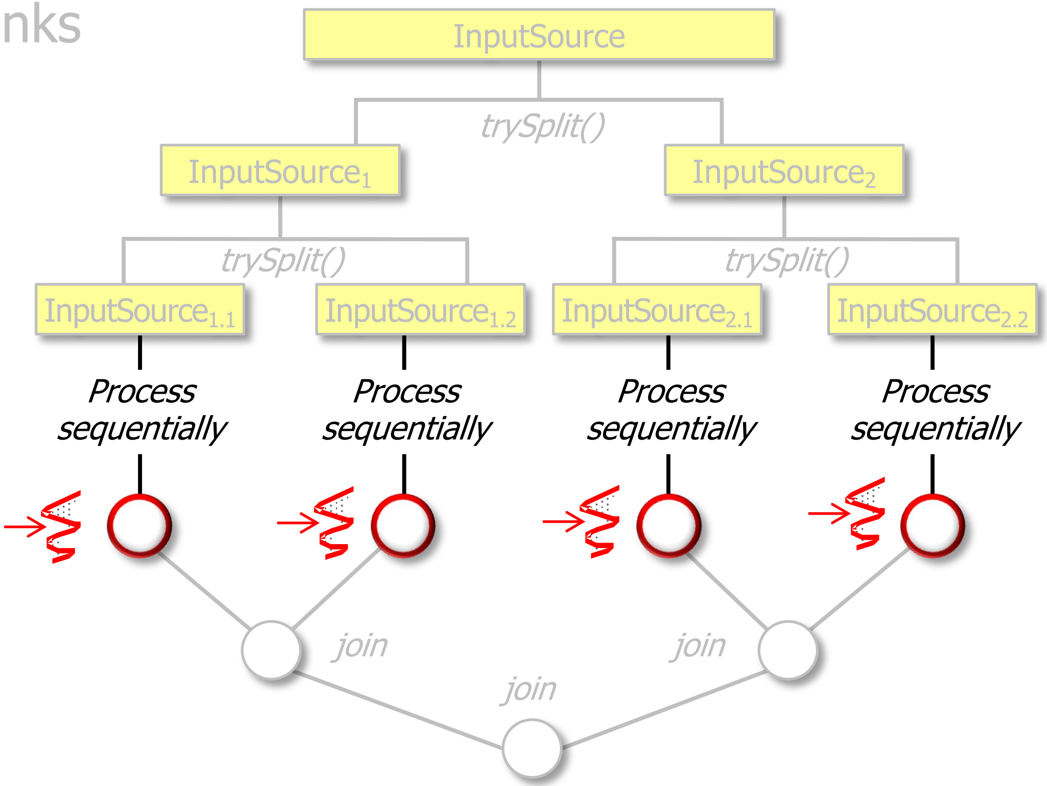
Structure & Functionality of Non-Concurrent Collectors

- A non-concurrent collector operates by merging sub-results
  - The input is partitioned into chunks
  - Each chunk runs in parallel in the common fork-join pool
  - Chunk sub-results are collected into an intermediate mutable result container
    - e.g., List, Set, Map, etc.



Structure & Functionality of Non-Concurrent Collectors

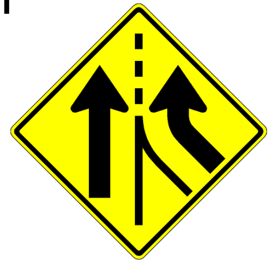
- A non-concurrent collector operates by merging sub-results
  - The input is partitioned into chunks
  - Each chunk runs in parallel in the common fork-join pool
  - Chunk sub-results are collected into an intermediate mutable result container
    - e.g., List, Set, Map, etc.

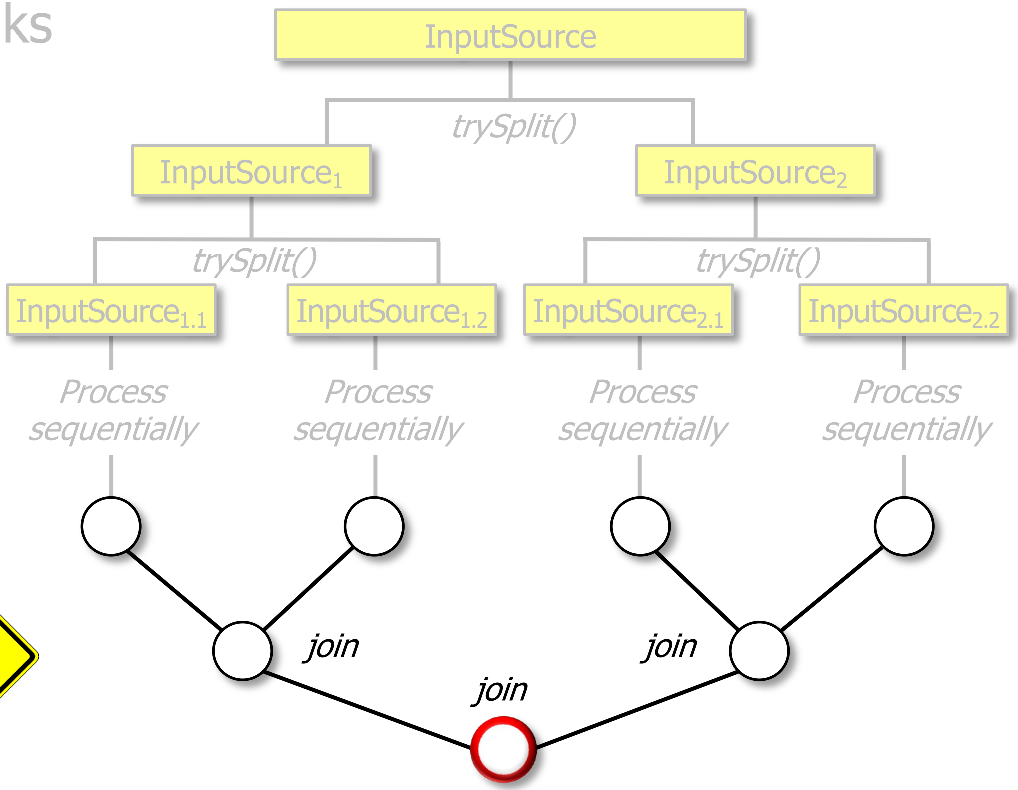


Different threads operate on different instances of intermediate result containers

Structure & Functionality of Non-Concurrent Collectors

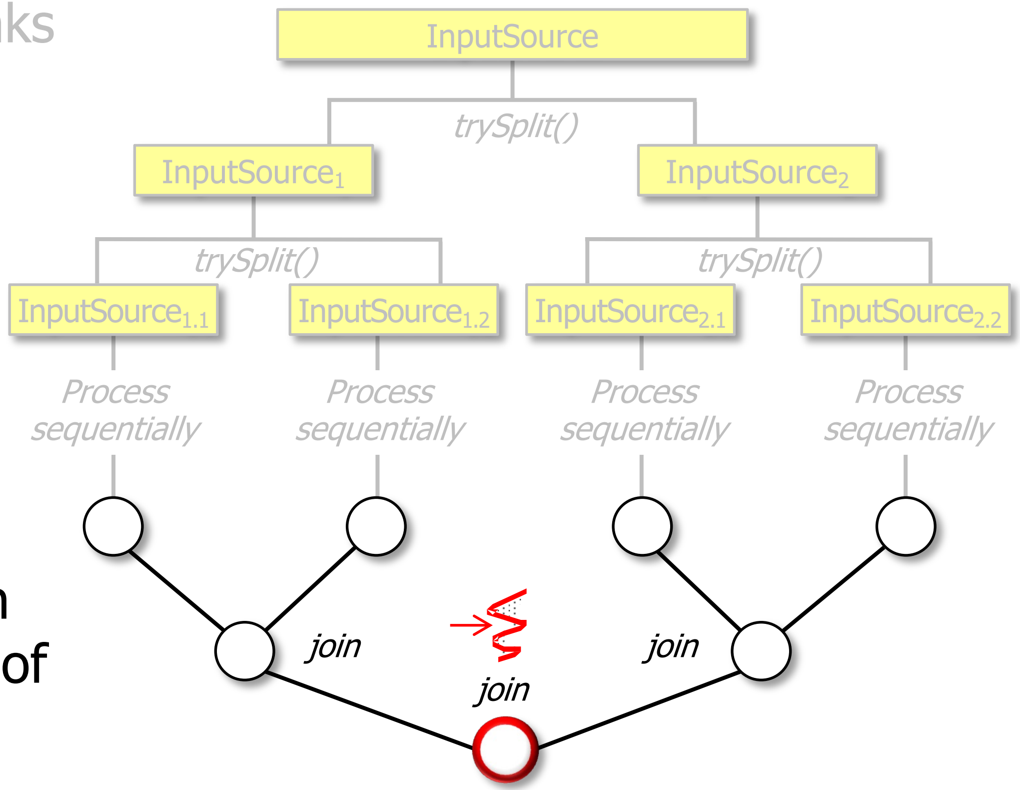
- A non-concurrent collector operates by merging sub-results
  - The input is partitioned into chunks
  - Each chunk runs in parallel in the common fork-join pool
  - Chunk sub-results are collected into an intermediate mutable result container
  - Sub-results are merged into one mutable result container





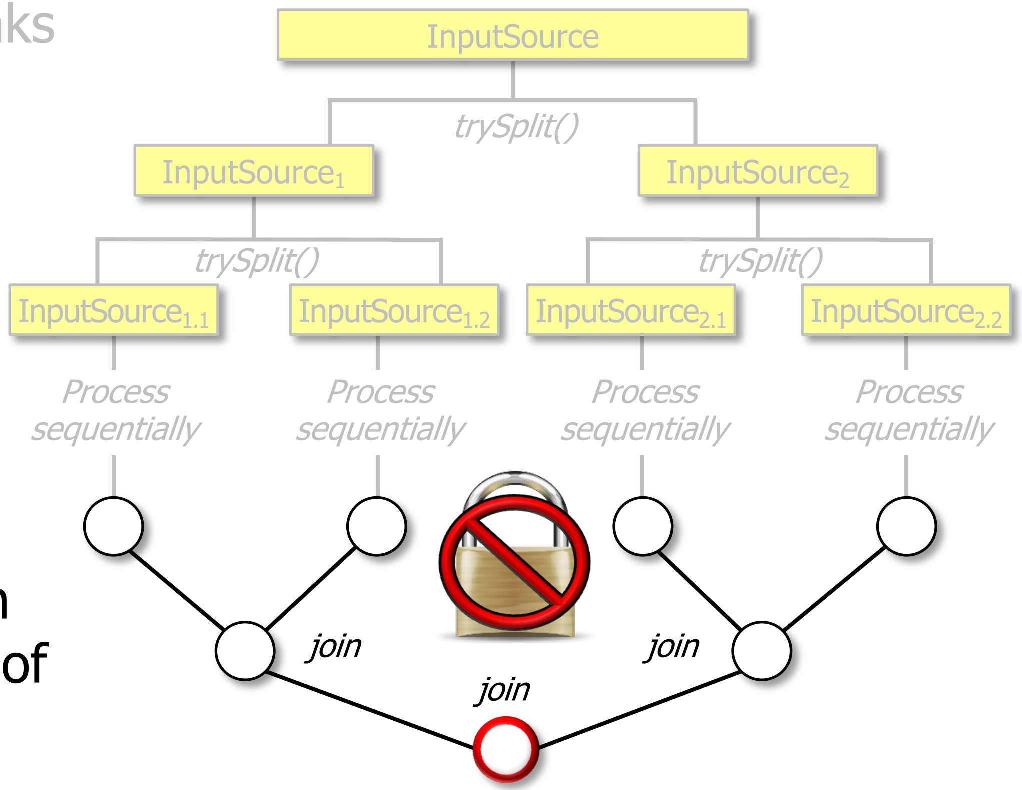
Structure & Functionality of Non-Concurrent Collectors

- A non-concurrent collector operates by merging sub-results
  - The input is partitioned into chunks
  - Each chunk runs in parallel in the common fork-join pool
  - Chunk sub-results are collected into an intermediate mutable result container
  - Sub-results are merged into one mutable result container
    - Only one thread in the fork-join pool is used to merge any pair of intermediate sub-results



Structure & Functionality of Non-Concurrent Collectors

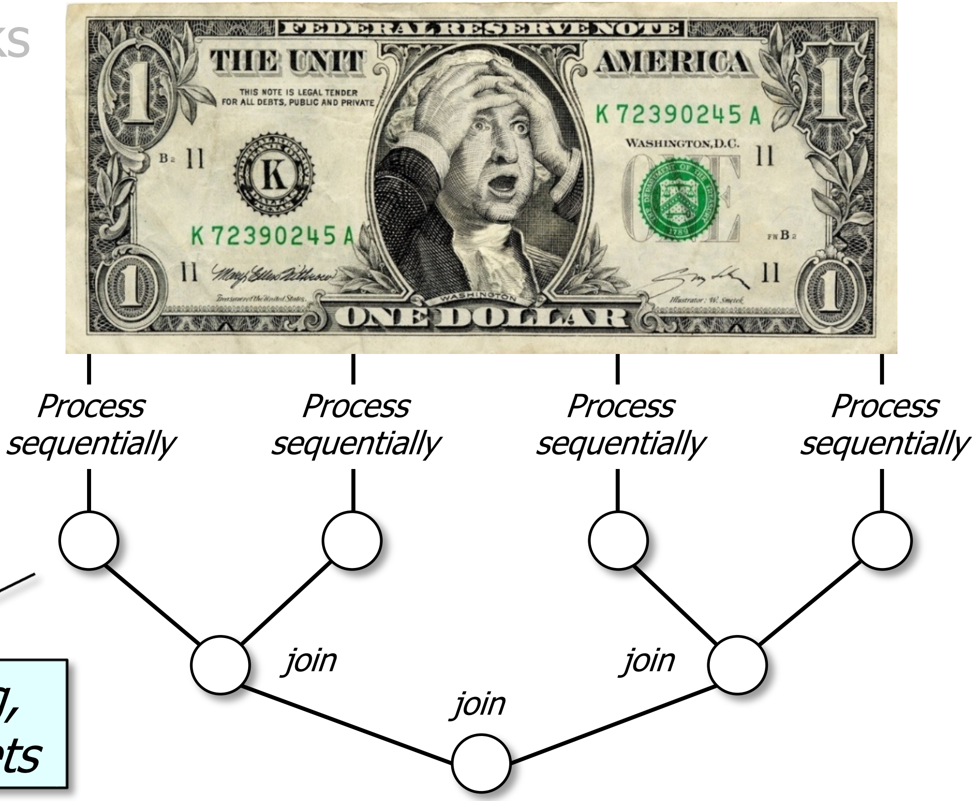
- A non-concurrent collector operates by merging sub-results
  - The input is partitioned into chunks
  - Each chunk runs in parallel in the common fork-join pool
  - Chunk sub-results are collected into an intermediate mutable result container
  - Sub-results are merged into one mutable result container
    - Only one thread in the fork-join pool is used to merge any pair of intermediate sub-results



Thus there's no need for any synchronizers in a non-concurrent collector

Structure & Functionality of Non-Concurrent Collectors

- A non-concurrent collector operates by merging sub-results
  - The input is partitioned into chunks
  - Each chunk runs in parallel in the common fork-join pool
  - Chunk sub-results are collected into an intermediate mutable result container
  - Sub-results are merged into one mutable result container



This process is safe & order-preserving, but costly for containers like maps & sets

Structure & Functionality of Concurrent Collectors

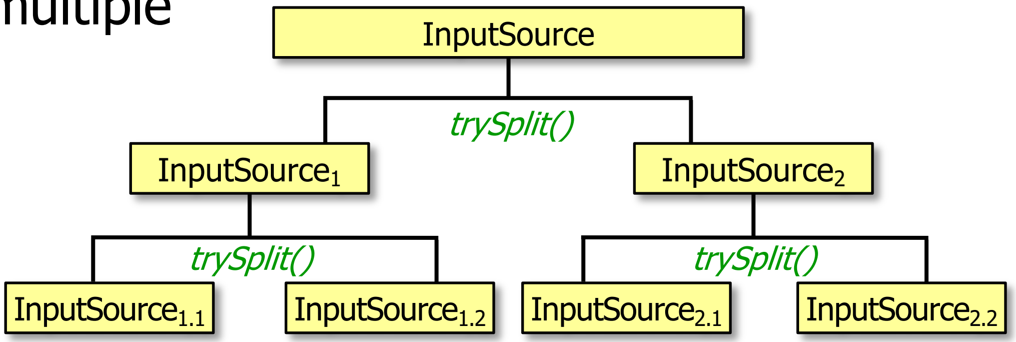
Structure & Functionality of Concurrent Collectors

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

See stackoverflow.com/questions/22350288/parallel-streams-collectors-and-thread-safety

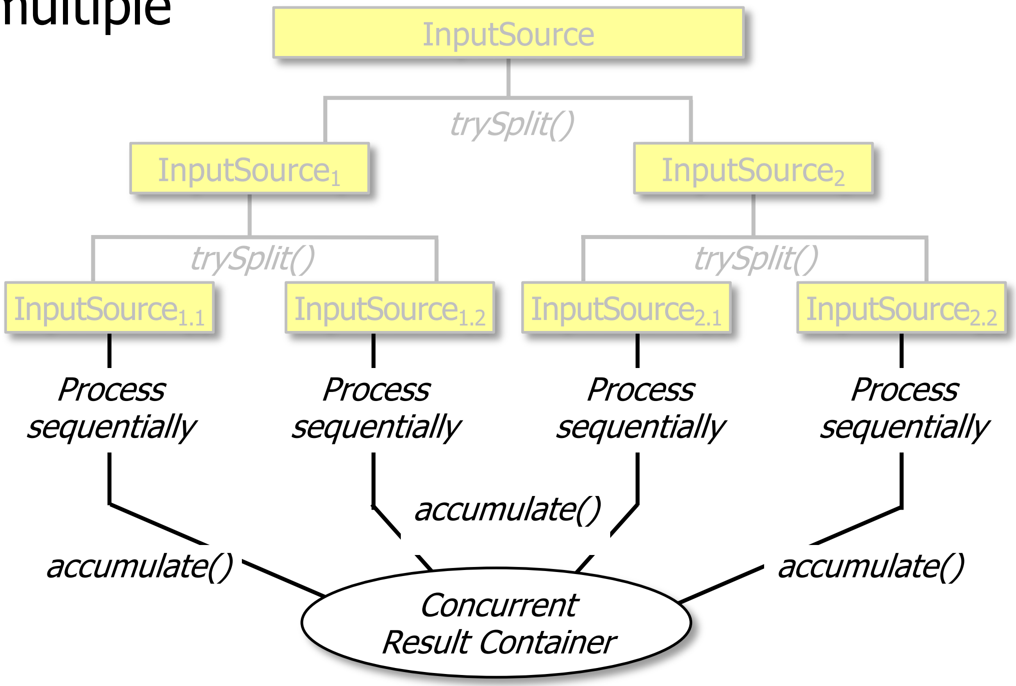
Structure & Functionality of Concurrent Collectors

- A concurrent collector creates one concurrent mutable result container & accumulates elements into it from multiple threads in a parallel stream
  - As usual, the input is partitioned into chunks



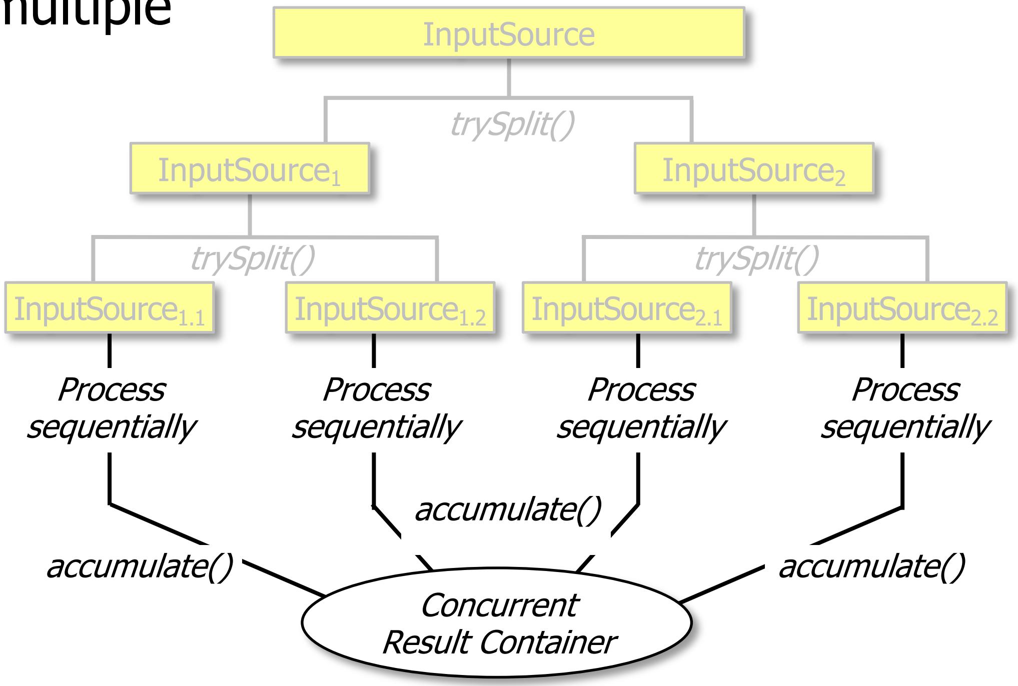
Structure & Functionality of Concurrent Collectors

- A concurrent collector creates one concurrent mutable result container & accumulates elements into it from multiple threads in a parallel stream
  - As usual, the input is partitioned into chunks
  - Each chunk runs in parallel in the common fork-join pool



Structure & Functionality of Concurrent Collectors

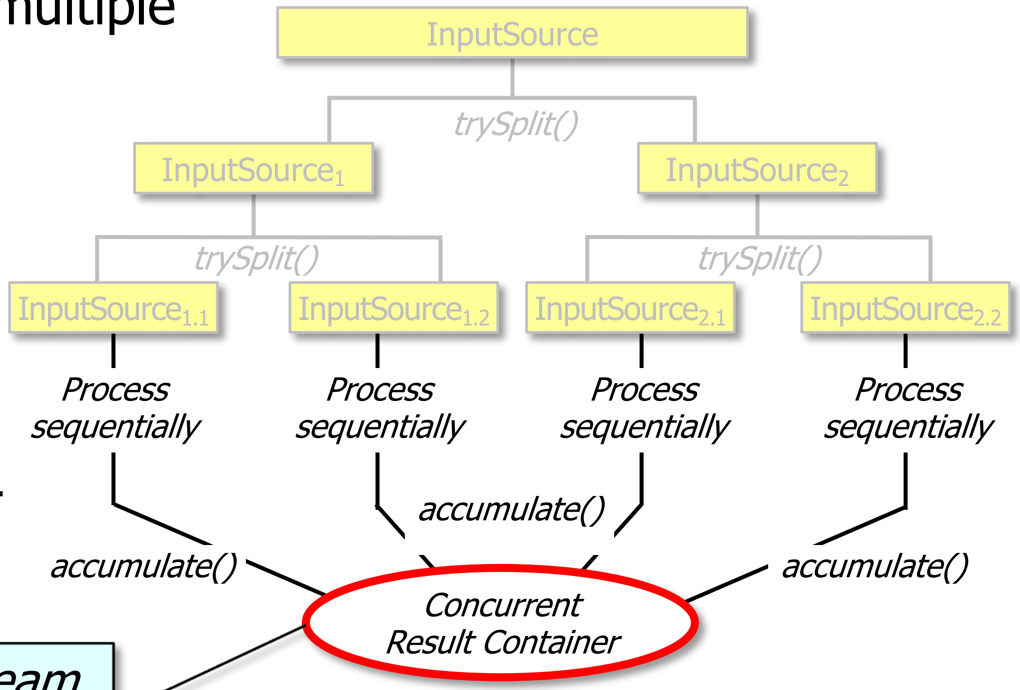
- A concurrent collector creates one concurrent mutable result container & accumulates elements into it from multiple threads in a parallel stream
  - As usual, the input is partitioned into chunks
  - Each chunk runs in parallel in the common fork-join pool
  - Chunk sub-results are collected into one mutable result container
    - e.g., a concurrent collection



See docs.oracle.com/javase/tutorial/essential/concurrency/collections.html

Structure & Functionality of Concurrent Collectors

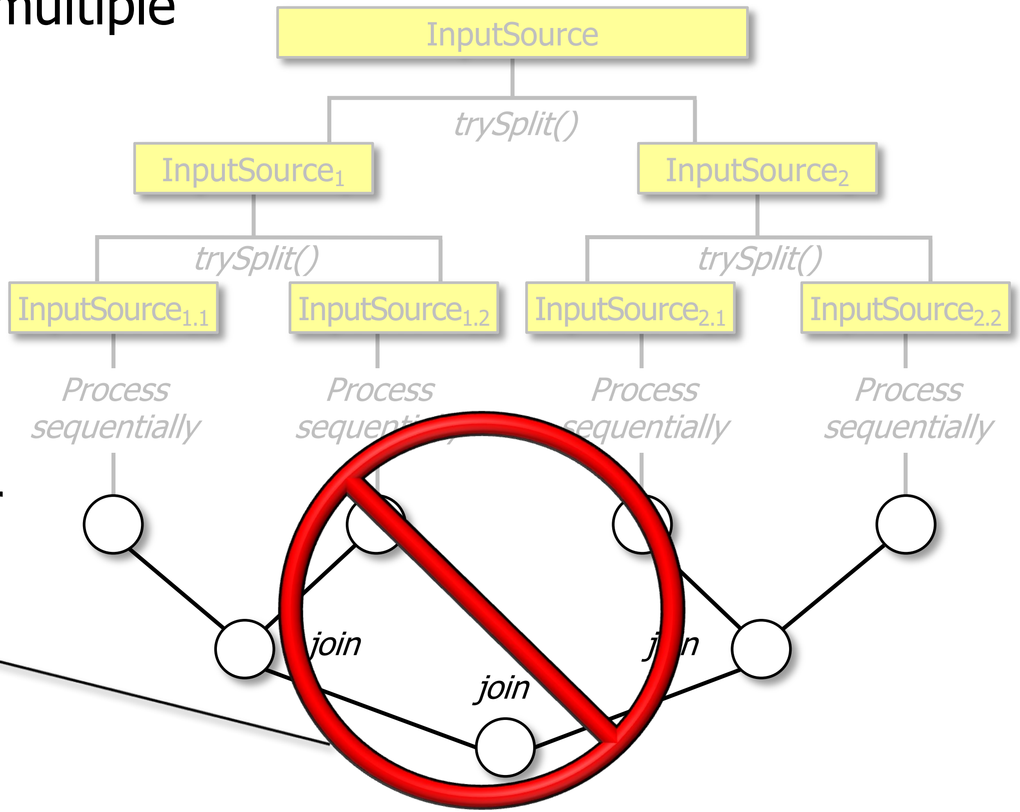
- A concurrent collector creates one concurrent mutable result container & accumulates elements into it from multiple threads in a parallel stream
  - As usual, the input is partitioned into chunks
  - Each chunk runs in parallel in the common fork-join pool
  - Chunk sub-results are collected into one mutable result container
    - e.g., a concurrent collection



Different threads in a parallel stream share one concurrent result container

Structure & Functionality of Concurrent Collectors

- A concurrent collector creates one concurrent mutable result container & accumulates elements into it from multiple threads in a parallel stream
  - As usual, the input is partitioned into chunks
  - Each chunk runs in parallel in the common fork-join pool
  - Chunk sub-results are collected into one mutable result container

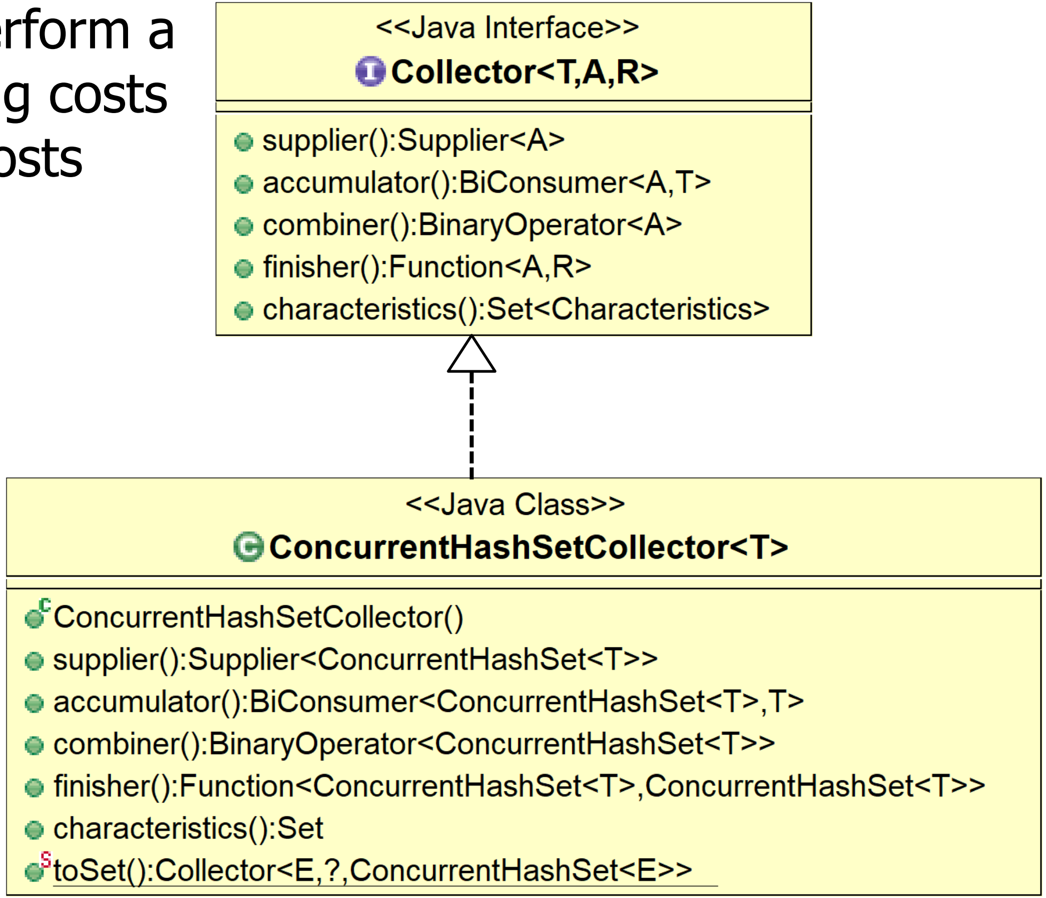


Thus there's no need to merge any intermediate sub-results!

Of course, encounter order is not preserved & synchronization is required..

Structure & Functionality of Concurrent Collectors

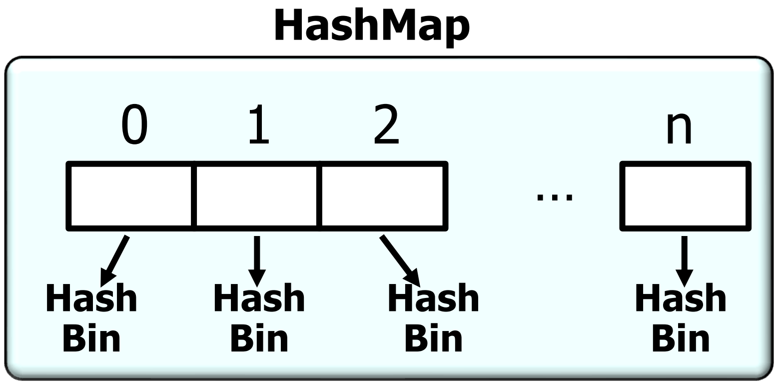
- A concurrent collector *may* out-perform a non-concurrent collector *if* merging costs are higher than synchronization costs

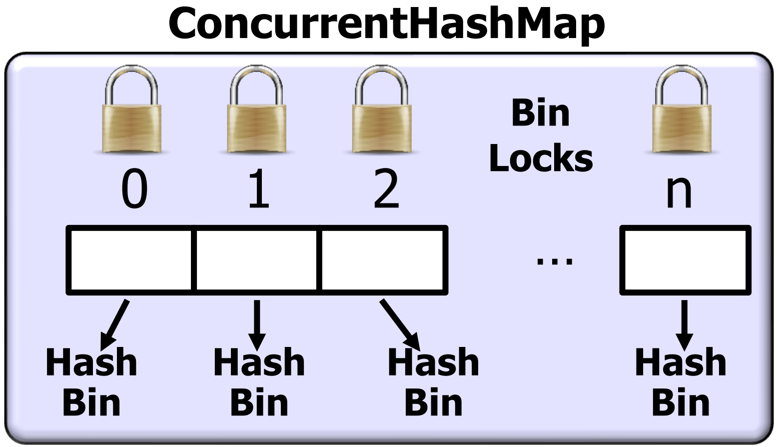


See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex34

Structure & Functionality of Concurrent Collectors

- A concurrent collector *may* out-perform a non-concurrent collector *if* merging costs are higher than synchronization costs
  - Highly optimized result containers like ConcurrentHashMap may be more efficient than merging HashMaps

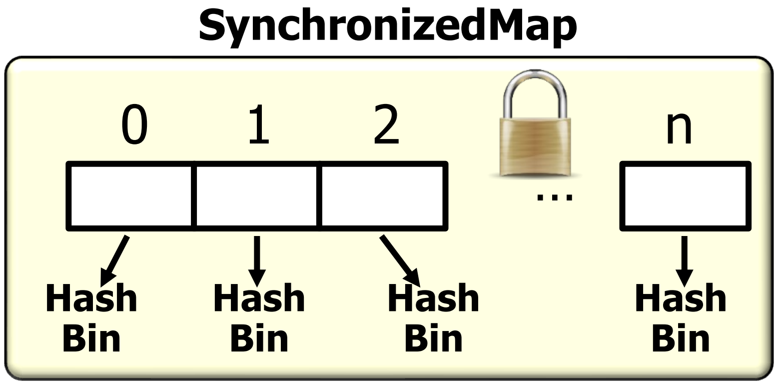


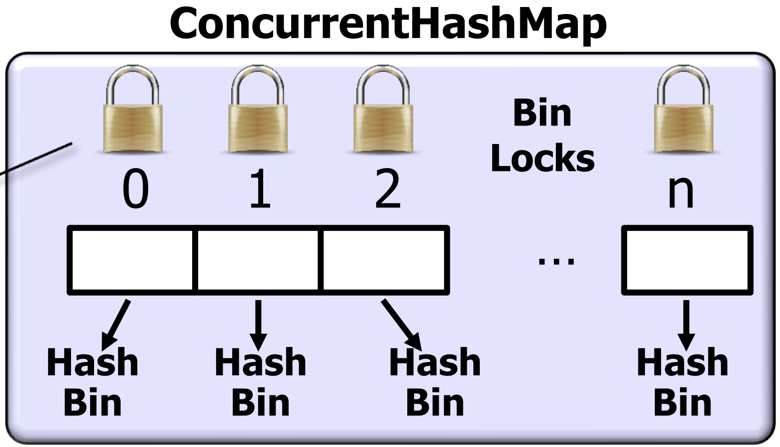


See codepumpkin.com/hashtable-vs-synchronizedmap-vs-concurrenthashmap

Structure & Functionality of Concurrent Collectors

- A concurrent collector *may* out-perform a non-concurrent collector *if* merging costs are higher than synchronization costs
  - Highly optimized result containers like ConcurrentHashMap may be more efficient than merging HashMaps
  - ConcurrentHashMap is also more efficient than a SynchronizedMap



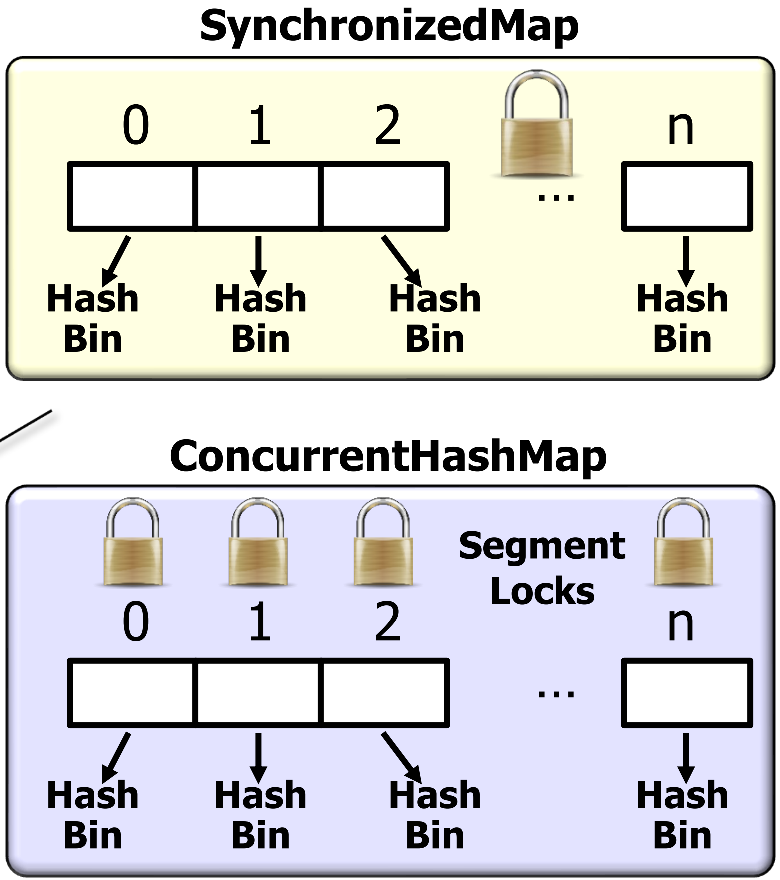


Contention is low due to use of multiple locks

See www.quora.com/What-is-the-difference-between-synchronize-and-concurrent-collection-in-Java

Structure & Functionality of Concurrent Collectors

- A concurrent collector *may* out-perform a non-concurrent collector *if* merging costs are higher than synchronization costs
  - Highly optimized result containers like ConcurrentHashMap may be more efficient than merging HashMaps
  - ConcurrentHashMap is also more efficient than a SynchronizedMap



In contrast, SynchronizedMap uses just one lock

See www.quora.com/What-is-the-difference-between-synchronize-and-concurrent-collection-in-Java

End of Java Parallel Streams Internals: Non-Concurrent & Concurrent Collectors (Part 1)