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
  - Configure the Java 8 parallel stream common fork-join pool
  - Avoid pool starvation & improve performance w/ ManagedBlocker
- Perform a reduction that combines partial results into a single result

Combining Results in a Parallel Stream
Combining Results in a Parallel Stream

- After the common fork-join pool finishes processing chunks their partial results are combined into a final result.

This discussion assumes a non-concurrent collector (more discussions follow).
Combining Results in a Parallel Stream

- After the common fork-join pool finishes processing chunks their partial results are combined into a final result.
- `join()` occurs in a single thread at each level.
  - i.e., the “parent”
Combining Results in a Parallel Stream

- After the common fork-join pool finishes processing chunks their partial results are combined into a final result.
- `join()` occurs in a single thread at each level, i.e., the “parent”

As a result, there’s typically no need for synchronizers during the joining.
Combining Results in a Parallel Stream

- Different terminal operations combine partial results in different ways

Understanding these differences is particularly important for parallel streams
Combining Results in a Parallel Stream

- Different terminal operations combine partial results in different ways, e.g.
  - `reduce()` creates a new immutable value

See [docs.oracle.com/javase/tutorial/essential/concurrency/immutable.html](https://docs.oracle.com/javase/tutorial/essential/concurrency/immutable.html)
Combining Results in a Parallel Stream

- Different terminal operations combine partial results in different ways, e.g.
  - `reduce()` creates a new immutable value

```java
long factorial(long n) {
    return LongStream
        .rangeClosed(1, n)
        .parallel()
        .reduce(1, (a, b) -> a * b, (a, b) -> a * b);
}
```

Combining Results in a Parallel Stream

- Different terminal operations combine partial results in different ways, e.g.
  - `reduce()` creates a new immutable value

```java
long factorial(long n) {
    return LongStream
        .rangeClosed(1, n)
        .parallel()
        .reduce(1, (a, b) -> a * b,
                (a, b) -> a * b);
}
```

- `reduce()` combines two immutable values (e.g., `long` or `Long`) & produces a new one
Combining Results in a Parallel Stream

- Different terminal operations combine partial results in different ways, e.g.
  - `reduce()` creates a new immutable value
  - `collect()` mutates an existing value

See greenteapress.com/thinkapjava/html/thinkjava011.html
Combining Results in a Parallel Stream

- Different terminal operations combine partial results in different ways, e.g.
  - `reduce()` creates a new immutable value
  - `collect()` mutates an existing value

```java
List<CharSequence> uniqueWords =
    getInput(sSHAKESPEARE, "\\s+")
    .parallelStream()
    ... .collect(toCollection(TreeSet::new));
```

See [github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex14](https://github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex14)
Combining Results in a Parallel Stream

- Different terminal operations combine partial results in different ways, e.g.
  - `reduce()` creates a new immutable value
  - `collect()` mutates an existing value

```java
List<CharSequence> uniqueWords =
    getInput(sSHAKESPEARE, "\s+")
    .parallelStream()
    ...
    .collect(toCollection(TreeSet::new));
```

Collecting results in a parallel stream involves processing sequentially:

1. First quarter of words
2. Second quarter of words
3. Third quarter of words
4. Fourth quarter of words

`collect()` mutates a container to accumulate the result it's producing.
Combining Results in a Parallel Stream

- Different terminal operations combine partial results in different ways, e.g.
  - `reduce()` creates a new immutable value
  - `collect()` mutates an existing value

List<CharSequence>

```java
List<CharSequence> uniqueWords =
    getInput(sSHAKESPEARE, "\s+")
    .parallelStream()
    ...
    .collect(ConcurrentHashSetCollector.toSet());
```

Concurrent collectors are different than non-concurrent collectors (covered later)
Combining Results in a Parallel Stream

• More discussion about `reduce()` vs. `collect()` appears online

See www.youtube.com/watch?v=oWIWEKNM5Aw
More discussion about reduce() vs. collect() appears online, e.g.

Always test w/a parallel stream to detect mistakes wrt mutable vs. immutable reductions

```java
void buggyStreamReduce (boolean parallel) {
    ...
    Stream<String> wordStream = allWords.stream();

    if (parallel)
        wordStream.parallel();

    String words = wordStream .reduce(new StringBuilder(),
        StringBuilder::append,
        StringBuilder::append) .toString();
}
```

See [github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex17](https://github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex17)
void buggyStreamReduce 
    (boolean parallel) {
    ... 
    Stream<String> wordStream = 
        allWords.stream();
    
    if (parallel)
        wordStream.parallel();
    
    String words = wordStream .reduce(new StringBuilder(), 
        StringBuilder::append, 
        StringBuilder::append) 
        .toString();

This code fails when parallel() is used since reduce() expects to do an "immutable" reduction

• More discussion about reduce() vs. collect() appears online, e.g.
• Always test w/a parallel stream to detect mistakes wrt mutable vs. immutable reductions
More discussion about `reduce()` vs. `collect()` appears online, e.g.

- Always test with a parallel stream to detect mistakes wrt mutable vs. immutable reductions

```java
void buggyStreamReduce(boolean parallel) {
    ...
    Stream<String> wordStream = allWords.stream();

    if (parallel)
        wordStream.parallel();

    String words = wordStream
        .reduce(new StringBuilder(),
                StringBuilder::append,
                StringBuilder::append)
        .toString();

    There are race conditions here since there's just one shared StringBuilder, which is not properly thread-safe..
```
Combining Results in a Parallel Stream

- More discussion about reduce() vs. collect() appears online, e.g.

- Always test w/ a parallel stream to detect mistakes wrt mutable vs. immutable reductions

```java
void buggyStreamReduce
  (boolean parallel) {
    ...
    Stream<String> wordStream = allWords.stream();
    if (parallel)
      wordStream.parallel();

    String words = wordStream
      .reduce(new StringBuilder(),
              StringBuilder::append,
              StringBuilder::append)
      .toString();
```

A stream can be dynamically switched to "parallel" mode!
Combining Results in a Parallel Stream

• More discussion about `reduce()` vs. `collect()` appears online, e.g.
  • Always test w/ a parallel stream to detect mistakes wrt mutable vs. immutable reductions
  • Beware of issues related to association & identity

```java
void testDifferenceReduce(...) {
    long difference = LongStream
        .rangeClosed(1, 100)
        .parallel()
        .reduce(0L,
                (x, y) -> x - y);
}

void testSum(long identity, ...) {
    long sum = LongStream
        .rangeClosed(1, 100)
        .reduce(identity,
                // Could use (x, y) -> x + y
                Math::addExact);
}
```

See [github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex17](https://github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex17)
More discussion about reduce() vs. collect() appears online, e.g.

- Always test w/a parallel stream to detect mistakes wrt mutable vs. immutable reductions

- Beware of issues related to association & identity

```java
void testDifferenceReduce(...) {
    long difference = LongStream
        .rangeClosed(1, 100)
        .parallel()
        .reduce(0L,
                (x, y) -> x - y);
}
```

```java
void testSum(long identity, ...) {
    long sum = LongStream
        .rangeClosed(1, 100)
        .reduce(identity,
                // Could use (x, y) -> x + y
                Math::addExact);
}
```

This code fails for a parallel stream since subtraction is not associative

• More discussion about `reduce()` vs. `collect()` appears online, e.g.
  • Always test w/ a parallel stream to detect mistakes wrt mutable vs. immutable reductions
  • Beware of issues related to association & identity

```java
void testDifferenceReduce(...) {
    long difference = LongStream
        .rangeClosed(1, 100)
        .parallel()
        .reduce(0L,
               (x, y) -> x - y);
}

void testSum(long identity, ...) {
    long sum = LongStream
        .rangeClosed(1, 100)
        .reduce(identity,
                 // Could use (x, y) -> x + y
                 Math::addExact);
}
```

This code fails if `identity` is not 0L

The “identity” of an OP is defined as “identity OP value == value”
Implementing Concurrent & Non-Concurrent Collectors
Implementing Concurrent & Non-Concurrent Collectors

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

![Interface Collector](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html)

**Interface Collector**<\text{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

```java
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](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html)
Collector implementations can either be non-concurrent or concurrent based on their characteristics.


**Enum Collector.Characteristics**

```java
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>

```java
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.
Implementing Concurrent & Non-Concurrent Collectors

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

See “Overview of Java 8 Streams (Part 4)” for non-concurrent collector implementation
Implementing Concurrent & Non-Concurrent Collectors

- Collector implementations can either be non-concurrent or 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’ll just focus on parallel streams in the subsequent discussion
Implementing Concurrent & Non-Concurrent Collectors

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

See stackoverflow.com/questions/22350288/parallel-streams-collectors-and-thread-safety
Implementing Concurrent & Non-Concurrent Collectors

- A non-concurrent collector operates by merging sub-results
- The input source is partitioned into chunks
Implementing Concurrent & Non-Concurrent Collectors

- A non-concurrent collector operates by merging sub-results
  - The input source is partitioned into chunks
  - Each chunk is collected into a result container
    - e.g., a list or a map

```
InputSource
  /        
trySplit() InputSource_1  trySplit() InputSource_2
    
trySplit() InputSource_1.1  trySplit() InputSource_2.1
    Process sequentially  Process sequentially

trySplit() InputSource_1.2  trySplit() InputSource_2.2
    Process sequentially  Process sequentially

    join

    join

    join

```
Implementing Concurrent & Non-Concurrent Collectors

- A non-concurrent collector operates by merging sub-results
  - The input source is partitioned into chunks
  - Each chunk is collected into a result container
    - e.g., a list or a map

Different threads operate on different instances of intermediate result containers
Implementing Concurrent & Non-Concurrent Collectors

- A non-concurrent collector operates by merging sub-results
  - The input source is partitioned into chunks
  - Each chunk is collected into a result container
  - These sub-results are then merged into a final mutable result container
  - Only one thread in the fork-join pool is used to merge any pair of intermediate results
Implementing Concurrent & Non-Concurrent Collectors

- A non-concurrent collector operates by merging sub-results
  - The input source is partitioned into chunks
  - Each chunk is collected into a result container
  - These sub-results are then merged into a final mutable result container
    - Only one thread in the fork-join pool is used to merge any pair of intermediate results

Thus there’s no need for any synchronizers in a non-concurrent collector
Implementing Concurrent & Non-Concurrent Collectors

• A non-concurrent collector operates by merging sub-results
  • The input source is partitioned into chunks
  • Each chunk is collected into a result container
  • These sub-results are then merged into a final mutable result container

This process is safe & order-preserving, but merging is costly for containers like maps & sets
Implementing Concurrent & Non-Concurrent Collectors

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

See stackoverflow.com/questions/22350288/parallel-streams-collectors-and-thread-safety
Implementing Concurrent & Non-Concurrent Collectors

- A concurrent collector creates one concurrent result container & inserts elements into it from multiple threads in a parallel stream
- As usual, the input source is partitioned into chunks
Implementing Concurrent & Non-Concurrent Collectors

- A concurrent collector creates one concurrent result container & inserts elements into it from multiple threads in a parallel stream
  - As usual, the input source is partitioned into chunks
  - Each chunk is collected into one concurrent result container
    - e.g., a concurrent map or set
Implementing Concurrent & Non-Concurrent Collectors

- A concurrent collector creates one concurrent result container & inserts elements into it from multiple threads in a parallel stream
- As usual, the input source is partitioned into chunks
- Each chunk is collected into one concurrent result container
  - e.g., a concurrent map or set

Different threads in a parallel stream share one concurrent result container
Implementing Concurrent & Non-Concurrent Collectors

- A concurrent collector creates one concurrent result container & inserts elements into it from multiple threads in a parallel stream.
  - As usual, the input source is partitioned into chunks.
  - Each chunk is collected into one concurrent result container.

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

Of course, encounter order is not preserved..
Implementing Concurrent & Non-Concurrent Collectors

- A concurrent collector may perform better than a non-concurrent collector if merging costs are high.

```
<<Java Interface>>
Collector<T,A,R>

- supplier(): Supplier<A>
- accumulator(): BiConsumer<A,T>
- combiner(): BinaryOperator<A>
- finisher(): Function<A,R>
- characteristics(): Set<Characteristics>
```

```
<<Java Class>>
ConcurrentHashSetCollector<T>

- ConcurrentHashSetCollector()
- supplier(): Supplier<ConcurrentHashSet<T>>
- accumulator(): BiConsumer<ConcurrentHashSet<T>,T>
- combiner(): BinaryOperator<ConcurrentHashSet<T>>
- finisher(): Function<ConcurrentHashSet<T>,ConcurrentHashSet<T>>
- characteristics(): Set
- toSet(): Collector<E,?,ConcurrentHashSet<E>>
```

See [github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex14](https://github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex14)
Implementing Concurrent & Non-Concurrent Collectors

- A concurrent collector may perform better than a non-concurrent collector if merging costs are high
- e.g., for a highly optimized result container like ConcurrentHashMap vs. merging HashMaps

Implementing Concurrent & Non-Concurrent Collectors

- The Collector interface defines three generic types

  ```java
  public interface Collector<T, A, R> {
      Supplier<A> supplier();
      BiConsumer<A, T> accumulator();
      BinaryOperator<A> combiner();
      Function<A, R> finisher();
      Set<Characteristics> characteristics();
  }
  ```

See [www.baeldung.com/java-8-collectors](http://www.baeldung.com/java-8-collectors)
Implementing Concurrent & Non-Concurrent Collectors

- The Collector interface defines three generic types
  - $T$ – The type of objects available in the stream
    - e.g., Integer, String, etc.

<<Java Interface>>

```java
public interface Collector<T, A, R> {
    Supplier<A> supplier();
    BiConsumer<A, T> accumulator();
    BinaryOperator<A> combiner();
    Function<A, R> finisher();
    Set<Characteristics> characteristics();
}
```
Implementing Concurrent & Non-Concurrent Collectors

- The Collector interface defines three generic types
  - \( T \)
  - \( A \) – The type of a mutable accumulator object for collection
    - e.g., ConcurrentHashSet or ArrayList of \( T \)

<<Java Interface>>

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

- supplier(): Supplier\(<A>\)
- accumulator(): BiConsumer\(<A,T>\)
- combiner(): BinaryOperator\(<A>\)
- finisher(): Function\(<A,R>\)
- characteristics(): Set\(<Characteristics>\)
Implementing Concurrent & Non-Concurrent Collectors

- The Collector interface defines three generic types
  - T
  - A
  - R – The type of a final result
  - e.g., ConcurrentHashSet or ArrayList of T

```
<<Java Interface>>
Collector<T,A,R>

- supplier(): Supplier<A>
- accumulator(): BiConsumer<A,T>
- combiner(): BinaryOperator<A>
- finisher(): Function<A,R>
- characteristics(): Set<Characteristics>
```
Implementing Concurrent & Non-Concurrent Collectors

- Five methods are defined in the Collector interface

```
<<Java Interface>>
Collector<T,A,R>

- supplier(): Supplier<A>
- accumulator(): BiConsumer<A,T>
- combiner(): BinaryOperator<A>
- finisher(): Function<A,R>
- characteristics(): Set<Characteristics>
```
Implementing Concurrent & Non-Concurrent Collectors

- Five methods are defined in the Collector interface
  - `characteristics()` – provides a stream with additional information used for internal optimizations, e.g.
    - UNORDERED
    - The collector need not preserve the encounter order

```
<<Java Interface>>

Collector<T,A,R>

- supplier(): Supplier<A>
- accumulator(): BiConsumer<A,T>
- combiner(): BinaryOperator<A>
- finisher(): Function<A,R>
- characteristics(): Set<Characteristics>
```

A concurrent collector *should* be UNORDERED, but a non-concurrent collector *can* be ORDERED.
Implementing Concurrent & Non-Concurrent Collectors

- Five methods are defined in the Collector interface
  - `characteristics()` – provides a stream with additional information used for internal optimizations, e.g.
    - UNORDERED
  - `IDENTIFY_FINISH`
    - The finisher() is the identity function so it can be a no-op
      - e.g. finisher() just returns null

```
<<Java Interface>>

Collector<T,A,R>

- supplier(): Supplier<A>
- accumulator(): BiConsumer<A,T>
- combiner(): BinaryOperator<A>
- finisher(): Function<A,R>
- characteristics(): Set<Characteristics>
```

A concurrent collector should be `IDENTIFY_FINISH`, whereas a non-concurrent collector could be...
Implementing Concurrent & Non-Concurrent Collectors

- Five methods are defined in the Collector interface
  - `characteristics()` – provides a stream with additional information used for internal optimizations, e.g.
    - UNORDERED
    - IDENTIFY_FINISH
  - CONCURRENT
    - The accumulator() method is called concurrently on the result container
    - Naturally, the mutable result container must be synchronized!!

```
<<Java Interface>>
Collector<T,A,R>

- supplier(): Supplier<A>
- accumulator(): BiConsumer<A,T>
- combiner(): BinaryOperator<A>
- finisher(): Function<A,R>
- characteristics(): Set<Characteristics>
```

A concurrent collector *should* be CONCURRENT, but a non-concurrent collector should not be!
Implementing Concurrent & Non-Concurrent Collectors

- Five methods are defined in the Collector interface
  - `characteristics()` – provides a stream with additional information used for internal optimizations, e.g.
    - UNORDERED
    - IDENTIFY_FINISH
  - CONCURRENT
    - The accumulator() method is called concurrently on the result container
    - The combiner() method is a no-op since it’s not called at all

<<Java Interface>>
Collector<T,A,R>

- supplier(): Supplier<A>
- accumulator(): BiConsumer<A,T>
- combiner(): BinaryOperator<A>
- finisher(): Function<A,R>
- characteristics(): Set<Characteristics>
Implementing Concurrent & Non-Concurrent Collectors

- Five methods are defined in the Collector interface
  - `characteristics()` – provides a stream with additional information used for internal optimizations, e.g.
    - UNORDERED
    - IDENTIFY_FINISH
  - CONCURRENT

A non-concurrent collector can be used with either sequential or parallel streams
Five methods are defined in the Collector interface:

- `characteristics()` – provides a stream with additional information used for internal optimizations, e.g.
  - UNORDERED
  - IDENTIFY_FINISH
  - CONCURRENT

```java
return Collections.unmodifiableSet
   (EnumSet.of(Collector.Characteristics.CONCURRENT,
               Collector.Characteristics.UNORDERED,
               Collector.Characteristics.IDENTITY_FINISH));
```

See [github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex14](https://github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex14)
Implementing Concurrent & Non-Concurrent Collectors

• Five methods are defined in the Collector interface
  • characteristics()
  • supplier() – returns a supplier instance that generates an empty result container

<<Java Interface>>

Collector<T,A,R>

- supplier(): Supplier<A>
- accumulator(): BiConsumer<A,T>
- combiner(): BinaryOperator<A>
- finisher(): Function<A,R>
- characteristics(): Set<Characteristics>
Implementing Concurrent & Non-Concurrent Collectors

• Five methods are defined in the Collector interface
  • characteristics()
  • supplier() – returns a supplier instance that generates an empty result container, e.g.
    • return ArrayList::new

A non-concurrent collector has a different result container for each thread in a parallel stream.
Implementing Concurrent & Non-Concurrent Collectors

- Five methods are defined in the Collector interface
  - `characteristics()`
  - `supplier()` – returns a supplier instance that generates an empty result container, e.g.
    - `return ArrayList::new`
    - `return ConcurrentHashMap::new`

A concurrent collector has one result container shared by each thread in a parallel stream.
Implementing Concurrent & Non-Concurrent Collectors

- Five methods are defined in the Collector interface
  - characteristics()
  - supplier()
  - **accumulator()** – returns a biconsumer that adds a new element to result container
Implementing Concurrent & Non-Concurrent Collectors

- Five methods are defined in the Collector interface
  - characteristics()
  - supplier()
  - **accumulator()** – returns a biconsumer that adds a new element to result container, e.g.
    - return `ArrayList::add`

<table>
<thead>
<tr>
<th>Collector&lt;T,A,R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>supplier():Supplier&lt;A&gt;</td>
</tr>
<tr>
<td><strong>accumulator():BiConsumer&lt;A,T&gt;</strong></td>
</tr>
<tr>
<td>combiner():BinaryOperator&lt;A&gt;</td>
</tr>
<tr>
<td>finisher():Function&lt;A,R&gt;</td>
</tr>
<tr>
<td>characteristics():Set&lt;Characteristics&gt;</td>
</tr>
</tbody>
</table>

A non-concurrent collector’s methods should *not* be synchronized!
Implementing Concurrent & Non-Concurrent Collectors

- Five methods are defined in the Collector interface
  - characteristics()
  - supplier()
  - **accumulator()** – returns a biconsumer that adds a new element to result container, e.g.
    - return ArrayList::add
    - return ConcurrentHashMap::add

A concurrent collector’s methods *must* be synchronized!
Implementing Concurrent & Non-Concurrent Collectors

- Five methods are defined in the Collector interface
  - `characteristics()`
  - `supplier()`
  - `accumulator()`
  - `combiner()` – returns a function that merges two result containers together

```
java
Collector<T, A, R>

- supplier(): Supplier<A>
- accumulator(): BiConsumer<A, T>
- combiner(): BinaryOperator<A>
- finisher(): Function<A, R>
- characteristics(): Set<Characteristics>
```
Implementing Concurrent & Non-Concurrent Collectors

• Five methods are defined in the Collector interface
  • characteristics()
  • supplier()
  • accumulator()
  • combiner() – returns a function that merges two result containers together, e.g.
    • return (one, another) -> {
      one.addAll(another); return one;
    }
Implementing Concurrent & Non-Concurrent Collectors

Five methods are defined in the Collector interface:

- characteristics()
- supplier()
- accumulator()
- combiner() – returns a function that merges two result containers together, e.g.
  - return (one, another) -> {
    one.addAll(another); return one;
  }
  - return null

The combiner() method is not called when CONCURRENT is set.
Five methods are defined in the Collector interface:

- `characteristics()`
- `supplier()`
- `accumulator()`
- `combiner()`
- `finisher()` – returns a function that converts the result container to final result type

**Java Interface**

```java
Collector<T,A,R>
```

- `supplier()`: `Supplier<A>`
- `accumulator()`: `BiConsumer<A,T>`
- `combiner()`: `BinaryOperator<A>`
- `finisher()`: `Function<A,R>`
- `characteristics()`: `Set<Characteristics>`
Implementing Concurrent & Non-Concurrent Collectors

• Five methods are defined in the Collector interface
  • characteristics()
  • supplier()
  • accumulator()
  • combiner()
  • finisher() – returns a function that converts the result container to final result type, e.g.
    • `Function.identity()` or something much more interesting!

<<Java Interface>>

```
1 Collector<T,A,R>

supplier():Supplier<A>
accumulator():BiConsumer<A,T>
combiner():BinaryOperator<A>
finisher():Function<A,R>
characteristics():Set<Characteristics>
```
Implementing Concurrent & Non-Concurrent Collectors

- Five methods are defined in the Collector interface
  - `characteristics()`
  - `supplier()`
  - `accumulator()`
  - `combiner()`
  - `finisher()` – returns a function that converts the result container to final result type, e.g.
    - `Function.identity()`
    - `return null`

The finisher() method is not called when IDENTITY_FINISHER is set.
End of Java 8 Parallel Stream Internals (Part 5)