# 14

# Parallel Streams


ORACLE

# Objectives

After completing this lesson, you should be able to:

- Review the key characteristics of streams
- Contrast old style loop operations with streams
- Describe how to make a stream pipeline execute in parallel
- List the key assumptions needed to use a parallel pipeline
- Define reduction
- Describe why reduction requires an associative function
- Calculate a value using reduce
- Describe the process for decomposing and then merging work
- List the key performance considerations for parallel streams

# Streams Review

- Pipeline
  - Multiple streams passing data along
  - Operations can be lazy
  - Intermediate, Terminal, and Short-Circuit Terminal Operations
- Stream characteristics
  - Immutable
  - Once elements are consumed, they are no longer available from the stream.
  - Can be sequential (default) or **parallel**

# Old Style Collection Processing

```
15          double sum = 0;
16
17          for(Employee e:eList){
18              if(e.getState().equals("CO") &&
19                  e.getRole().equals(Role.EXECUTIVE)){
20                  e.printSummary();
21                  sum += e.getSalary();
22              }
23          }
24
25          System.out.printf("Total CO Executive Pay:
    $%,9.2f %n", sum);
```



There are a couple of key points that can be made about the above code.

- All elements in the collections must be iterated through every time.
- The code is more about "how" information is obtained and less about "what" the code is trying to accomplish.
- A mutator must be added to the loop to calculate the total.
- There is no easy way to parallelize this code.

## New Style Collection Processing

```
15          double result = eList.stream()
16              .filter(e -> e.getState().equals("CO"))
17              .filter(e -> e.getRole().equals(Role.EXECUTIVE))
18              .peek(e -> e.printSummary())
19              .mapToDouble(e -> e.getSalary())
20              .sum();
21
22          System.out.printf("Total CO Executive Pay: $%,9.2f %n", result);
```

- What are the advantages?
  - Code reads like a problem.
  - Acts on the data set
  - Operations can be lazy.
  - Operations can be serial or parallel.



There are also some key points worth pointing out for this piece of code as well.

- The code reads much more like a problem statement.
- No mutator is needed to get the final result.
- Using this approach provides more opportunity for lazy optimizations.
- This code can easily be parallelized.

# Stream Pipeline: Another Look

```
13     public static void main(String[] args) {
14
15         List<Employee> eList = Employee.createShortList();
16
17         Stream<Employee> s1 = eList.stream();
18
19         Stream<Employee> s2 = s1.filter(
20             e -> e.getState().equals("CO"));
21
22         Stream<Employee> s3 = s2.filter(
23             e -> e.getRole().equals(Role.EXECUTIVE));
24         Stream<Employee> s4 = s3.peek(e -> e.printSummary());
25         DoubleStream s5 = s4.mapToDouble(e -> e.getSalary());
26         double result = s5.sum();
27
28         System.out.printf("Total CO Executive Pay: $%,9.2f %n",
result);
29     }
```



So far all the examples have used lambda expressions and stream pipelines to perform the tasks. In the above example, the `Stream` class is used with regular Java statements to perform the same steps as those found in a pipeline. Even though the approach is possible, a stream pipeline seems like a much better solution.

# Styles Compared

| Imperative Programming | Streams |
| --- | --- |
| • Code deals with individual data items.<br>• Focused on how<br>• Code does not read like a problem.<br>• Steps mashed together<br>• Leaks extraneous details<br>• Inherently sequential | • Code deals with data set.<br>• Focused on what<br>• Code reads like a problem.<br>• Well factored<br>• No "garbage variables" (Temp variables leaked into scope)<br>• Code can be sequential or parallel. |

## Parallel Stream

- May provide better performance
  - Many chips and cores per machine
  - GPUs
- Map/Reduce in the small
- Fork/join is great, but too low level
  - A lot of boilerplate code
  - Stream uses fork/join under the hood
- Many factors affect performance
  - Data size, decomposition, packing, number of cores
- Unfortunately, not a magic bullet
  - Parallel is not always faster.



Making a stream run in parallel is pretty easy. Just call the `parallelStream` or `parallel` method in the stream. With that call, when the stream executes it uses all the processing cores available to the current JVM to perform the task.

The fork/join framework is used to break the work into smaller tasks, execute each task, and then recombine the results. But as you will see, much less code is needed to do this with streams than would be necessary if fork/join was coded by hand.

Remember, parallel is not always faster. For certain types of tasks, serial processing will produce better results.

# Using Parallel Streams: Collection

- Call from a Collection

```
15          double result = eList.parallelStream()
16              .filter(e -> e.getState().equals("CO"))
17              .filter(e -> e.getRole().equals(Role.EXECUTIVE))
18              .peek(e -> e.printSummary())
19              .mapToDouble(e -> e.getSalary())
20              .sum();
21
22          System.out.printf("Total CO Executive Pay: $%,9.2f %n", result);
```



This is an example of using the `parallelStream` method to make the stream pipeline parallel.

## Using Parallel Streams: From a Stream

```
27          result = eList.stream()
28              .filter(e -> e.getState().equals("CO"))
29              .filter(e -> e.getRole().equals(Role.EXECUTIVE))
30              .peek(e -> e.printSummary())
31              .mapToDouble(e -> e.getSalary())
32              .parallel()
33              .sum();
34
35          System.out.printf("Total CO Executive Pay: $%,9.2f %n", result);
```

- Specify with `.parallel` or `.sequential` (default is sequential)
- Choice applies to entire pipeline.
  - Last call wins
- Once again, the API doc is your friend.



This example uses the `parallel` method to make the stream pipeline parallel. Both the `sequential` and `parallel` methods may be called in a pipeline. **Whichever method is called last** will be applied to the stream.

## Pipelines Fine Print

- Stream pipelines are like Builders.
  - Add a bunch of intermediate operations and then execute
  - Cannot "branch" or "reuse" pipeline
- Do not modify the source during a query.
- Operation parameters must be stateless.
  - Do not access any state that might change.
  - **This enables correct operation sequentially or in parallel.**
- Best to banish side effects completely.



Your data should be immutable or read-only when used with stream pipelines. No changes to state should take place during a pipeline.

## Embrace Statelessness

```
17  List<Employee> newList02 = new ArrayList<>();
…
23    newList02 = eList.parallelStream() // Good Parallel
24              .filter(e -> e.getDept().equals("Eng"))
25              .collect(Collectors.toList());
```

- Mutate the stateless way.
  - The above is preferable.
  - It is designed to parallelize.



If you want to save the results after a pipeline completes, use the `collect` method and `Collectors` class as shown in the example. This method parallelizes well and treats the data in a stateless way.

Collectors are covered in much more detail in the lesson titled “Terminal Operations: Collectors.”

## Avoid Statefulness

```
15          List<Employee> eList = Employee.createShortList();
16          List<Employee> newList01 = new ArrayList<>();
17          List<Employee> newList02 = new ArrayList<>();
18
19          eList.parallelStream() // Not Parallel. Bad.
20              .filter(e -> e.getDept().equals("Eng"))
21              .forEach(e -> newList01.add(e));
```

- Temptation is to do the above.
  - **Do not do this. It does not parallelize.**



Stream pipeline results may be nondeterministic or incorrect if the behavioral parameters to the stream operations are stateful. A stateful lambda is one whose result depends on any state that might change during the execution of the stream pipeline.

**Note:** Do not write code like that shown in this example.

## Streams Are Deterministic for Most Part

```
14          List<Employee> eList = Employee.createShortList();
15
16          double r1 = eList.stream()
17              .filter(e -> e.getState().equals("CO"))
18              .mapToDouble(Employee::getSalary)
19              .sequential().sum();
20
21          double r2 = eList.stream()
22              .filter(e -> e.getState().equals("CO"))
23              .mapToDouble(Employee::getSalary)
24              .parallel().sum();
25
26      System.out.println("The same: " + (r1 == r2));
```

- Will the result be the same?



A deterministic algorithm is an algorithm that, given a particular input, will always produce the same output. The `sum` method is a great example as the order in which elements are combined does not matter. The result will be the same irrespective of the order in which elements are added.

## Some Are Not Deterministic

```
14          List<Employee> eList = Employee.createShortList();
15
16          Optional<Employee> e1 = eList.stream()
17              .filter(e -> e.getRole().equals(Role.EXECUTIVE))
18              .sequential().findAny();
19
20          Optional<Employee> e2 = eList.stream()
21              .filter(e -> e.getRole().equals(Role.EXECUTIVE))
22              .parallel().findAny();
23
24          System.out.println("The same: " +
25              e1.get().getEmail().equals(e2.get().getEmail()));
```

- Will the result be the same?
  - In this case, maybe not.



The larger the data set, the more likely the two code blocks will produce a different result. The parallel stream does not search the data sequentially. Consequently, it is possible it will find a different element that meets the criteria first.

# Reduction

- Reduction
  - An operation that takes a sequence of input elements and combines them into a single summary result by repeated application of a combining operation.
  - Implemented with the `reduce()` method
- Example: `sum` is a reduction with a base value of 0 and a combining function of +.
  - $((((0 + a_1) + a_2) + ...) + a_n)$
  - `.sum()` is equivalent to reduce `(0, (a, b) -> a +b )`
  - `(0, (sum, element) -> sum + element)`



Reduction is an operation that takes a sequence of input elements and combines them into a single summary result by repeated application of a combining operation. The `sum` method for the `Stream` class is an application of reduction.

## Reduction Fine Print

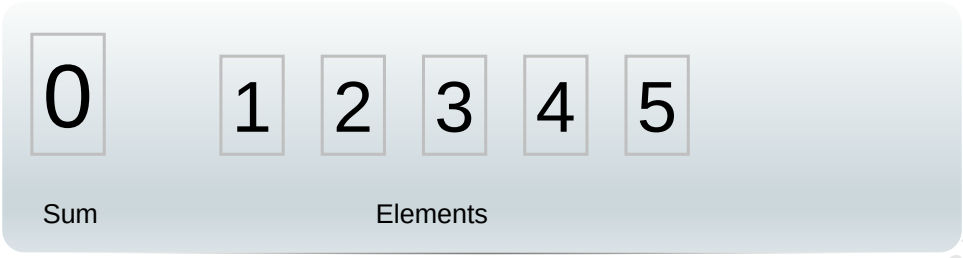
- If the combining function is associative, reduction parallelizes cleanly.
  - Associative means the order does not matter.
  - The result is the same irrespective of the order used to combine elements.
- Examples of: sum, min, max, average, count
  - `.count()` is equivalent to `.map(e -> 1).sum()`.
- **Warning:** If you pass a nonassociative function to `reduce`, you will get the wrong answer. The function must be associative.



As the above text points out, a reduction can only be preformed on an associate function, in effect, a function where order does not matter. If the function is not associative, you will get the wrong result.

## Reduction: Example

```
18          int r2 = IntStream.rangeClosed(1, 5).parallel()
19              .reduce(0, (sum, element) -> sum + element);
20
21          System.out.println("Result: " + r2);
```

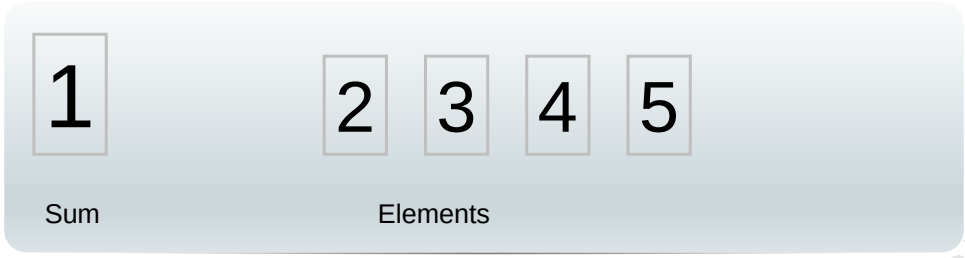





Note that the integer value of 0 is passed into the reduce method. This is called the *identity* value. It represents the starting value for the reduce function and the default return value if there are no members in the reduction.

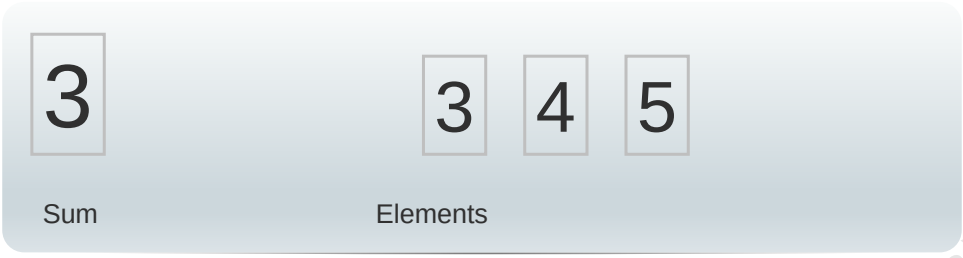
# Reduction: Example

```
18          int r2 = IntStream.rangeClosed(1, 5).parallel()
19              .reduce(0, (sum, element) -> sum + element);
20
21          System.out.println("Result: " + r2);
```



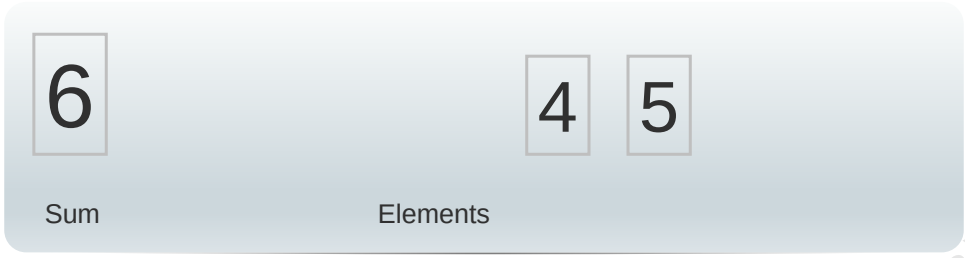
# Reduction: Example

```
18          int r2 = IntStream.rangeClosed(1, 5).parallel()
19              .reduce(0, (sum, element) -> sum + element);
20
21          System.out.println("Result: " + r2);
```



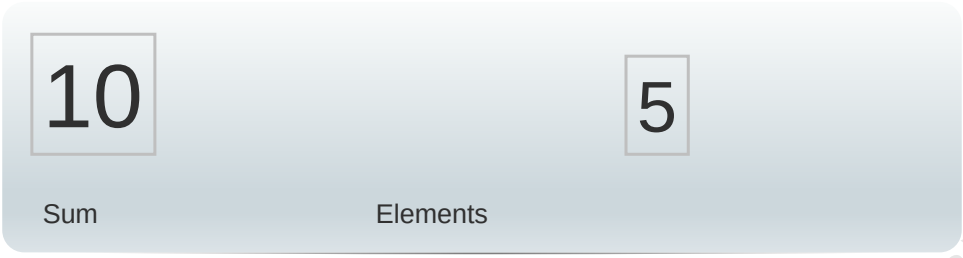
# Reduction: Example

```
18          int r2 = IntStream.rangeClosed(1, 5).parallel()
19              .reduce(0, (sum, element) -> sum + element);
20
21          System.out.println("Result: " + r2);
```



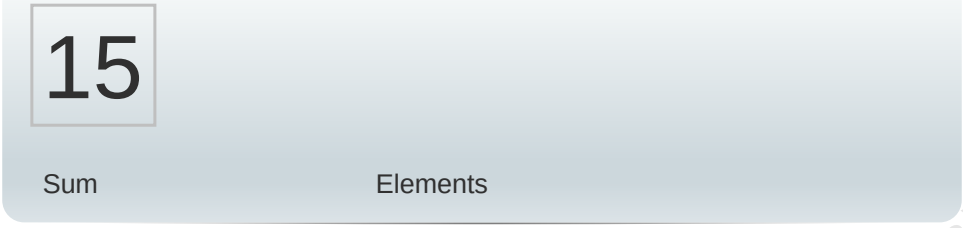
# Reduction: Example

```
18          int r2 = IntStream.rangeClosed(1, 5).parallel()
19              .reduce(0, (sum, element) -> sum + element);
20
21          System.out.println("Result: " + r2);
```



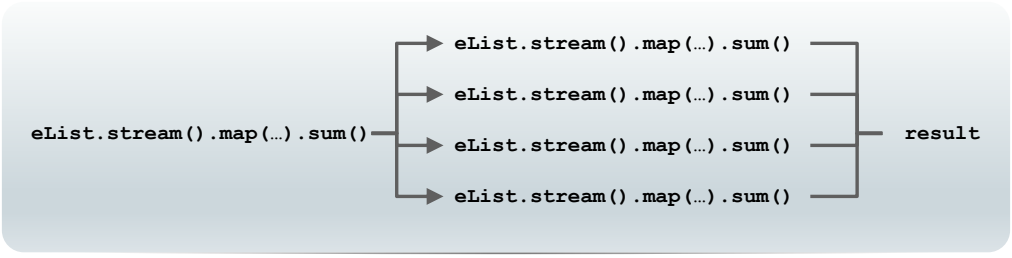
# Reduction: Example

```
18          int r2 = IntStream.rangeClosed(1, 5).parallel()
19              .reduce(0, (sum, element) -> sum + element);
20
21          System.out.println("Result: " + r2);
```




# A Look Under the Hood

- Pipeline decomposed into subpipelines
  - Each subpipeline produces a subresult.
  - Subresults are combined into final result.

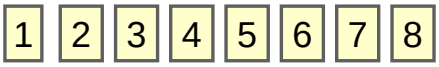





This picture shows how the sum is first decomposed into smaller steps. The results are then combined to produce a result.

## Illustrating Parallel Execution

```
18          int r2 = IntStream.rangeClosed(1, 8).parallel()
19              .reduce(0, (sum, element) -> sum + element);
```

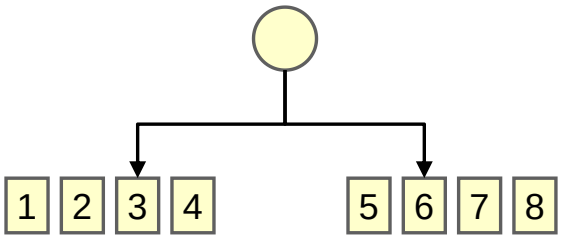





In the steps that follow, the data set above is summed. The steps of decomposition and then combination are shown in detail. Note that for this operation, the order of operations does not matter.

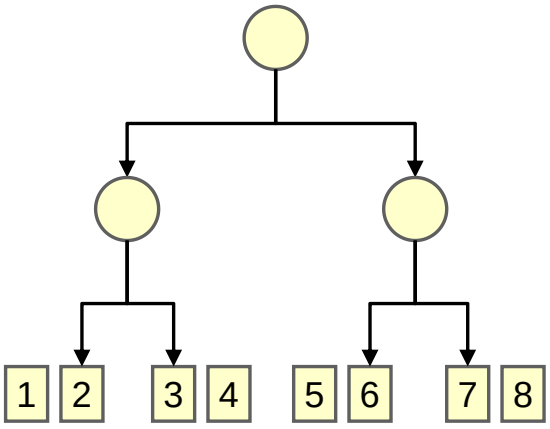
# Illustrating Parallel Execution

```
18          int r2 = IntStream.rangeClosed(1, 8).parallel()
19              .reduce(0, (sum, element) -> sum + element);
```



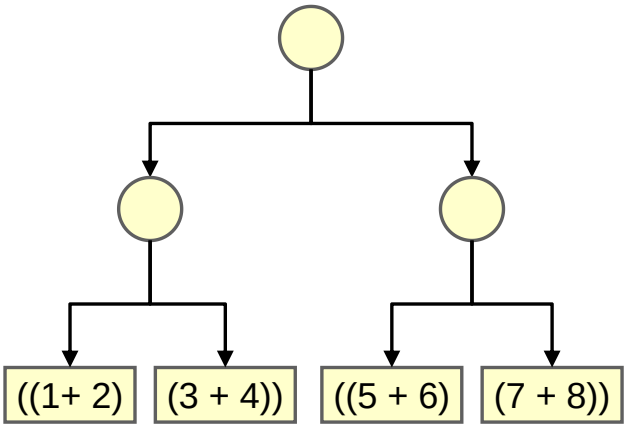
# Illustrating Parallel Execution

```
18          int r2 = IntStream.rangeClosed(1, 8).parallel()
19              .reduce(0, (sum, element) -> sum + element);
```



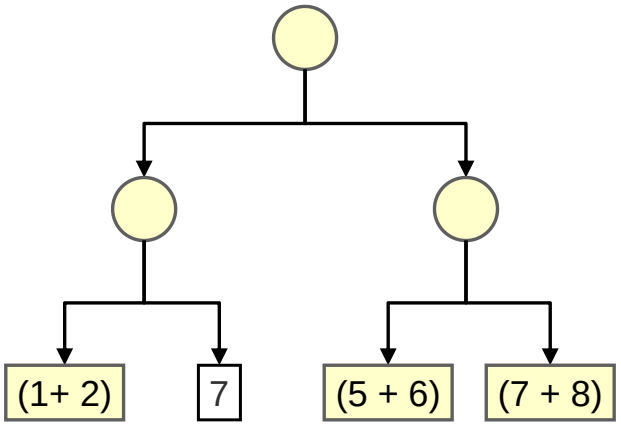
# Illustrating Parallel Execution

```
18          int r2 = IntStream.rangeClosed(1, 8).parallel()
19              .reduce(0, (sum, element) -> sum + element);
```



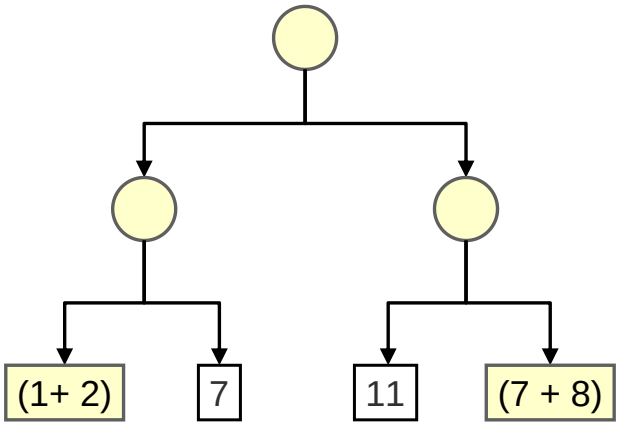
# Illustrating Parallel Execution

```
18          int r2 = IntStream.rangeClosed(1, 8).parallel()
19              .reduce(0, (sum, element) -> sum + element);
```



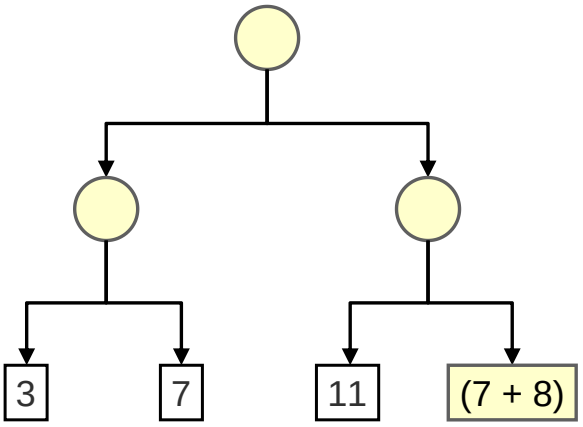
# Illustrating Parallel Execution

```
18          int r2 = IntStream.rangeClosed(1, 8).parallel()
19              .reduce(0, (sum, element) -> sum + element);
```



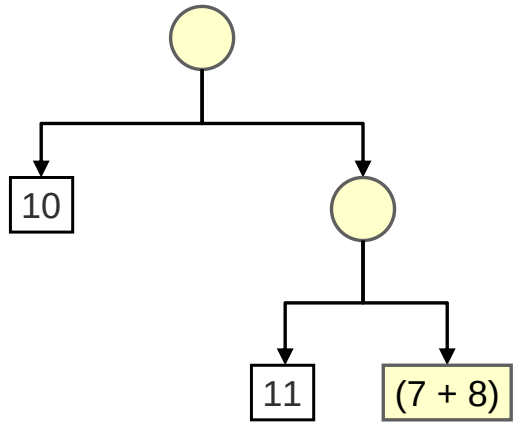
# Illustrating Parallel Execution

```
18          int r2 = IntStream.rangeClosed(1, 8).parallel()
19              .reduce(0, (sum, element) -> sum + element);
```



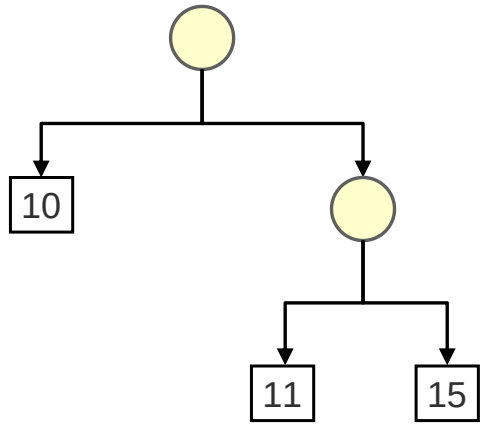
# Illustrating Parallel Execution

```
18          int r2 = IntStream.rangeClosed(1, 8).parallel()
19              .reduce(0, (sum, element) -> sum + element);
```



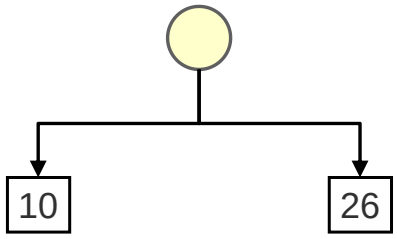
# Illustrating Parallel Execution

```
18          int r2 = IntStream.rangeClosed(1, 8).parallel()
19              .reduce(0, (sum, element) -> sum + element);
```



# Illustrating Parallel Execution

```
18          int r2 = IntStream.rangeClosed(1, 8).parallel()
19              .reduce(0, (sum, element) -> sum + element);
```




# Illustrating Parallel Execution

```
18          int r2 = IntStream.rangeClosed(1, 8).parallel()
19              .reduce(0, (sum, element) -> sum + element);
```

36

## Performance

- Do not assume parallel is always faster.
  - Parallel is not always the right solution.
  - Sometimes parallel is slower than sequential.
- Qualitative considerations
  - Does the stream source decompose well?
  - Do terminal operations have a cheap or expensive merge operation?
  - What are stream characteristics?
    - Filters change size, for example.
- Primitive streams are provided for performance.
  - Boxing/Unboxing negatively impacts performance.



As with any code, test and verify that a particular approach works as intended. As stated previously, associative functions decompose well and make good candidates for parallel processing. But operations that do not meet this criteria may perform better when processed sequentially.

# A Simple Performance Model

N = Size of the source data set

Q = Cost per element through the pipeline

N * Q ~= Cost of the pipeline

- Larger N*Q -> Higher change of good parallel performance
- Easier to know N than Q
- You can reason qualitatively about Q.
  - Simple pipeline example
    - N > 10K. Q=1
    - Reduction using sum
  - Complex pipelines might
    - Contain filters
    - Contain limit operation
    - Complex reduction using `groupingBy()`



As the slide points out, the larger the data set, the more likely parallel processing is going to show an improvement in performance. Some other observations:

- A system needs to have at least four cores available to the JVM before you will see any substantial difference in performance.
- As a general guideline, a data set should contain more than 10,000 items before showing a difference in performance.
- Any operations or complex operations that cause threads to block will have a negative impact on performance.

# Summary

In this lesson, you should have learned how to:

- Review the key characteristics of streams
- Contrast old style loop operations with streams
- Describe how to make a stream pipeline execute in parallel
- List the key assumptions needed to use a parallel pipeline
- Define reduction
- Describe why reduction requires an associative function
- Calculate a value using reduce
- Describe the process for decomposing and then merging work
- List the key performance considerations for parallel streams

# Practice 14: Overview

This practice covers the following topics:

- Practice 14-1: Calculating total sales without a pipeline
- Practice 14-2: Calculating sales totals using Parallel Streams
- Practice 14-3: Calculating sales totals using Parallel Streams and Reduce