Working with Streams – In Depth

Jesper de Jong
Software Architect

@jesperdj   www.jesperdj.com

Overview

Generating and building streams

Reduction and collection in detail

Grouping and partitioning

Parallel streams

Specialized streams

Generating and Building Streams

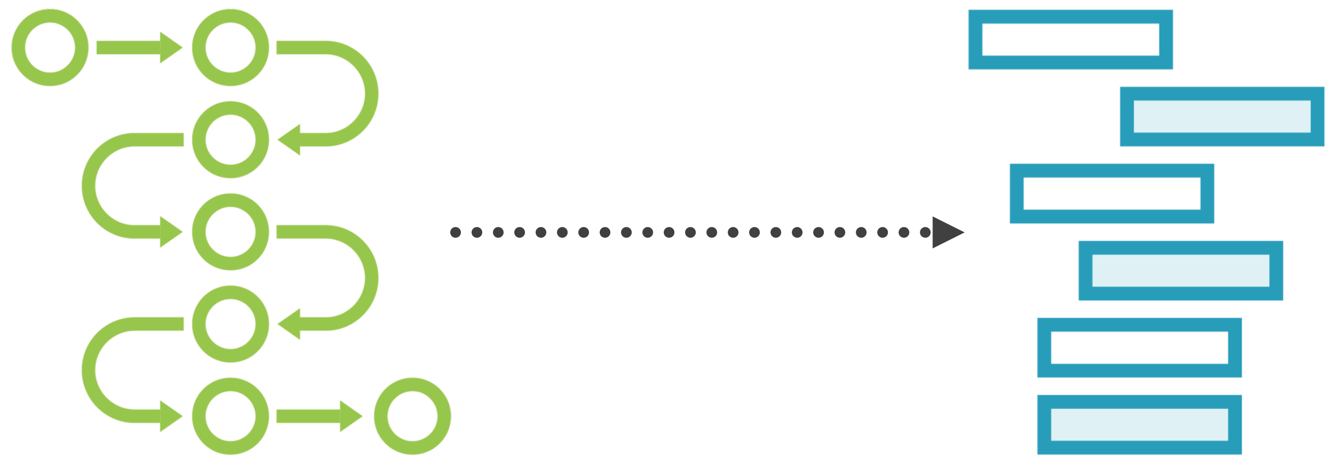
Reducing Streams in Detail

Collecting Streams in Detail

Collecting Streams

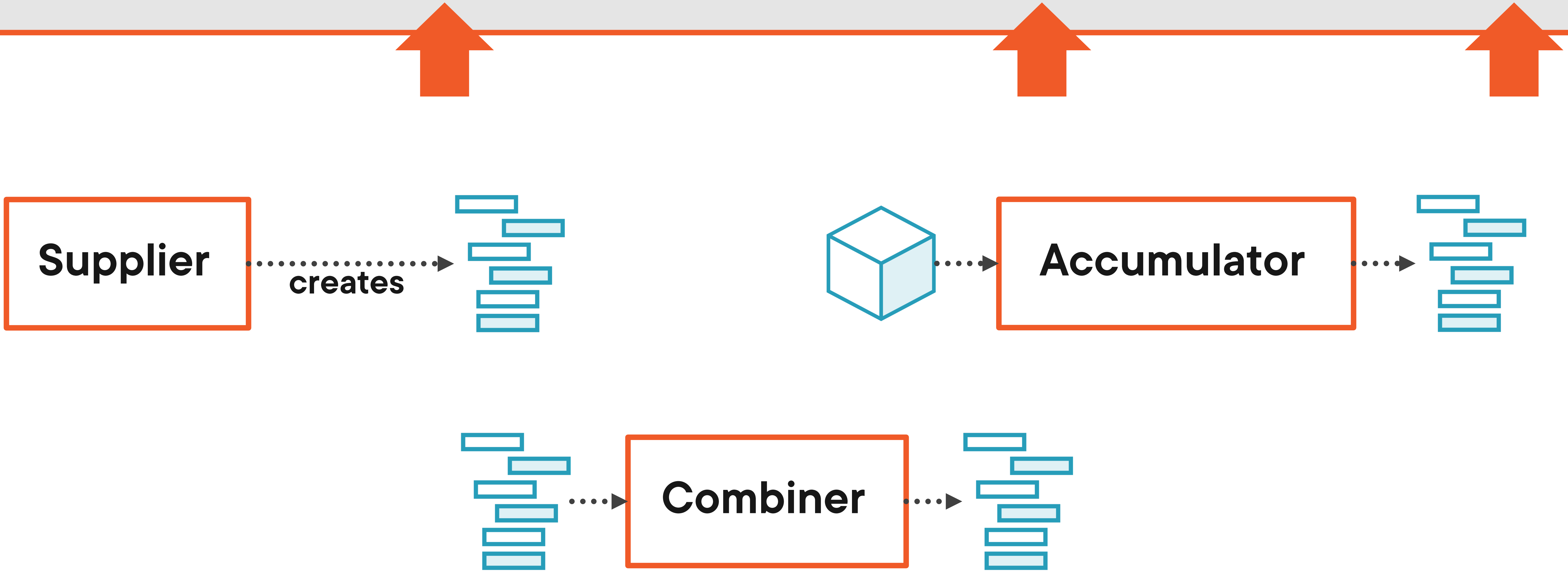
Collection = Mutable reduction

A collection operation reduces a stream into a mutable result container

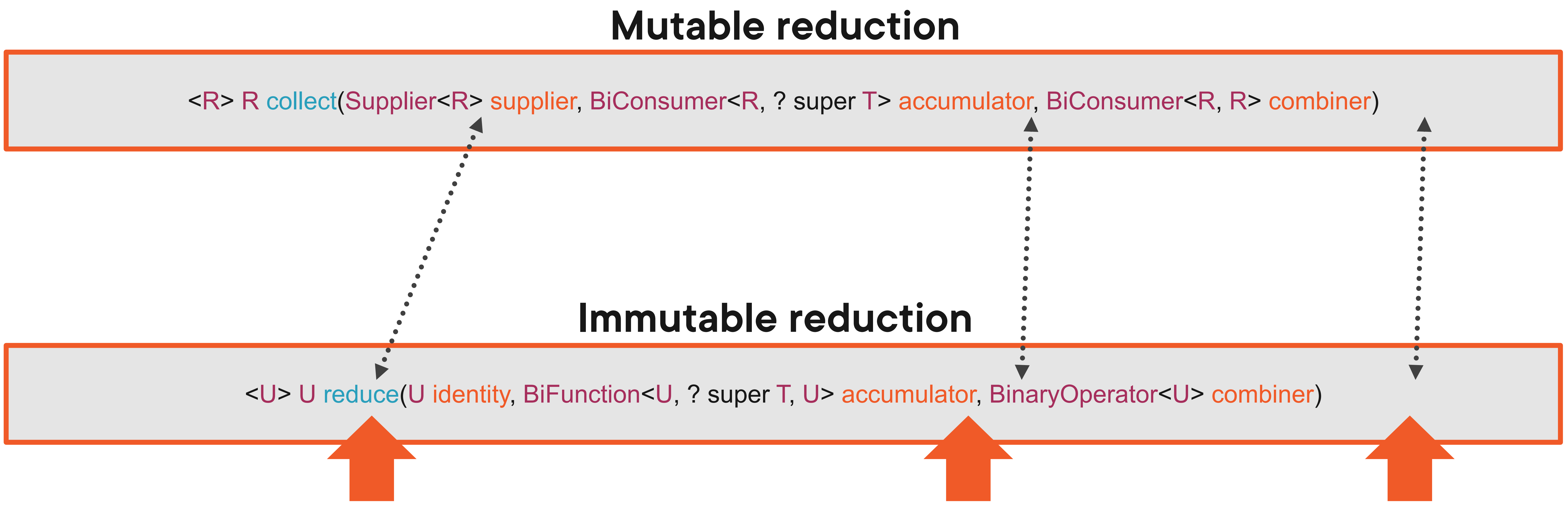


Collecting Streams

```
<R> R collect(Supplier<R> supplier, BiConsumer<R, ? super T> accumulator, BiConsumer<R, R> combiner)
```



Collection and Reduction



Mutable reduction

```
<R> R collect(Supplier<R> supplier, BiConsumer<R, ? super T> accumulator, BiConsumer<R, R> combiner)
```

Immutable reduction

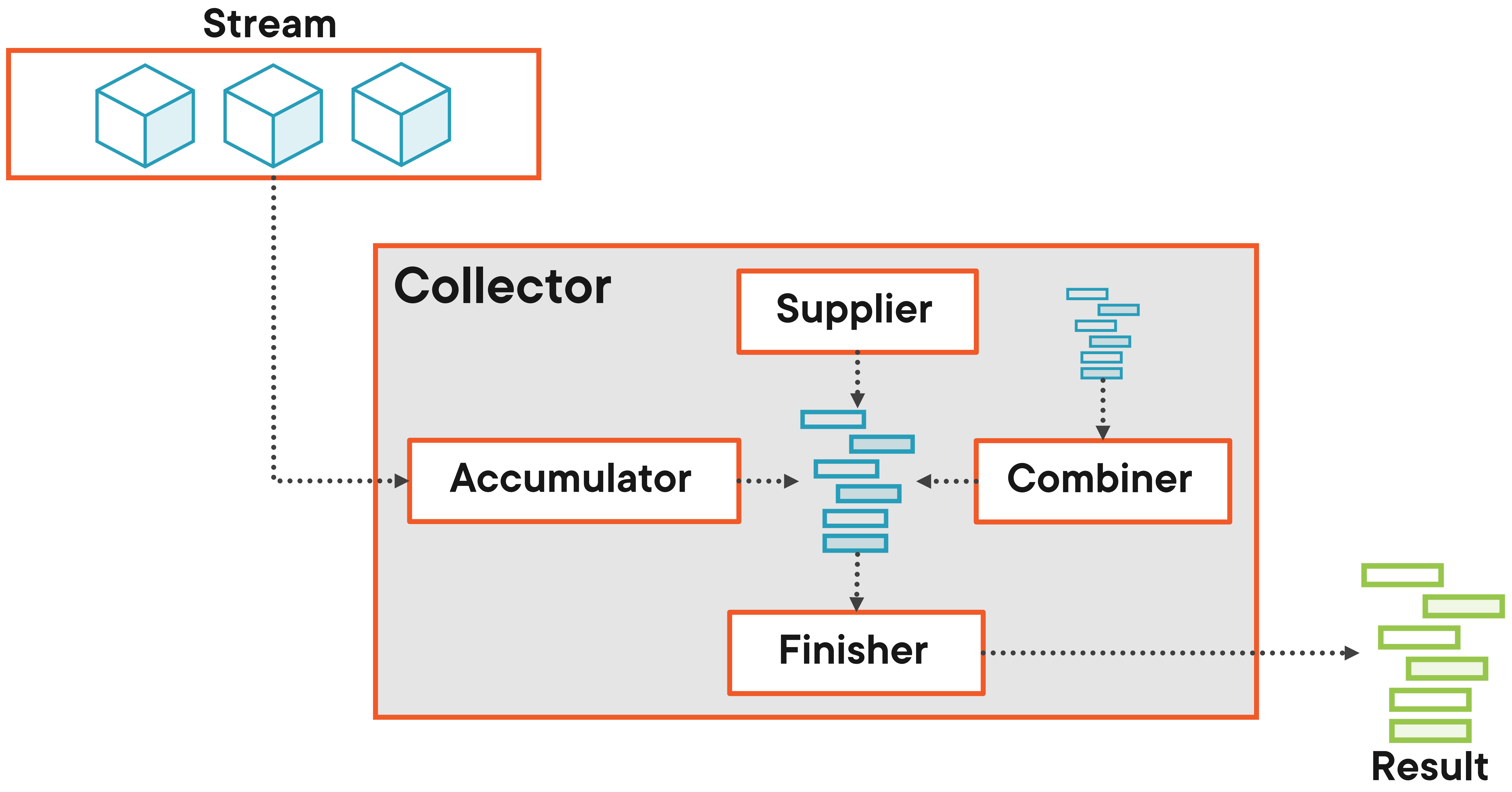
```
<U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)
```

Collection and Reduction

	collect(...)	reduce(...)
Accumulator	R, T → BiConsumer	U, T → BiFunction → U
Combiner	R, R → BiConsumer	U, U → BinaryOperator → U

Working with Collectors

Collector Functions



Working with Collectors

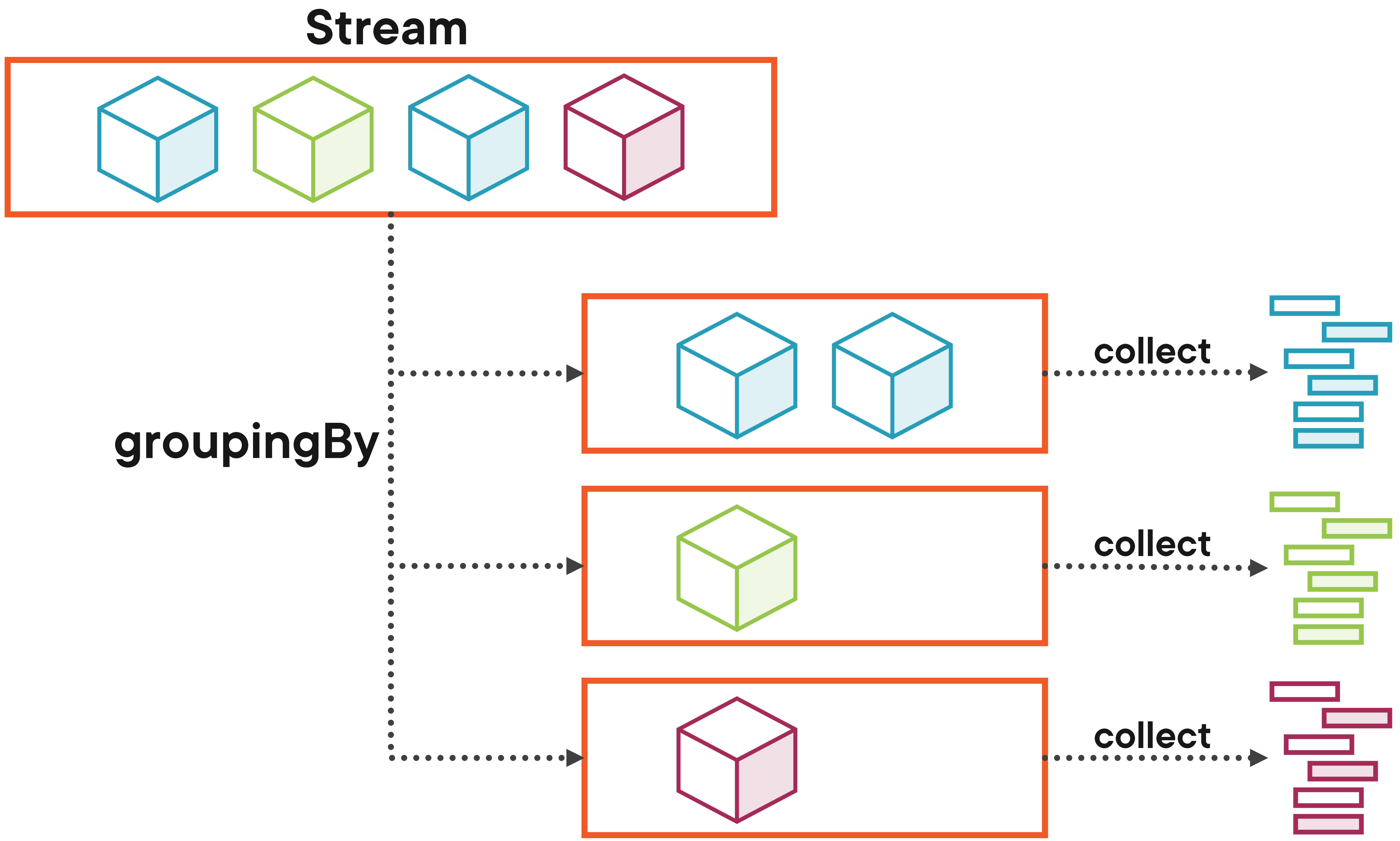
```
<R, A> R collect(Collector<? super T, A, R> collector)
```

Collectors

toList()
toSet()
toMap()
...

Grouping Stream Elements

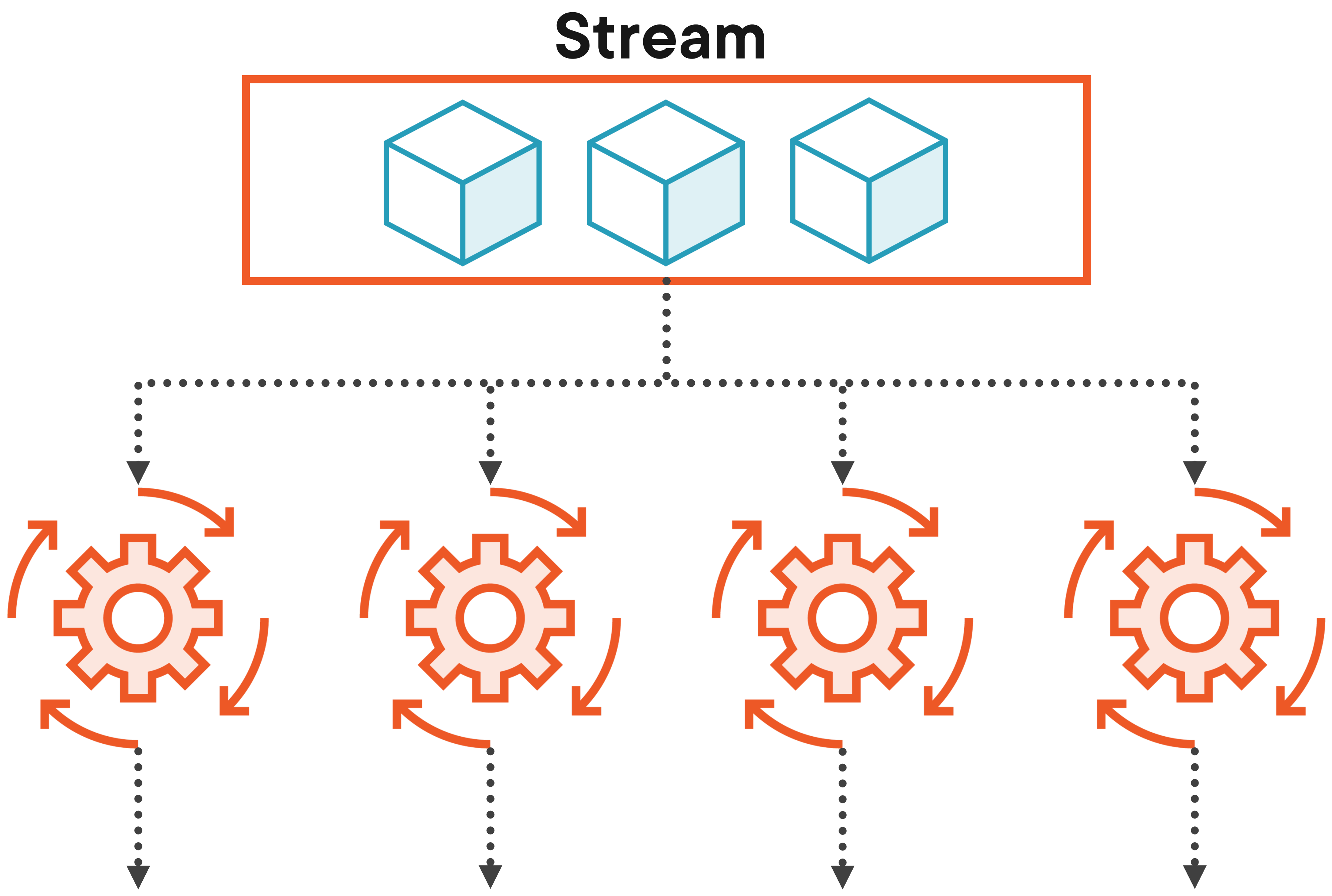
Grouping Stream Elements



Partitioning Stream Elements

Parallel Streams

Parallel Streams



Creating a Parallel Stream

```
List<String> names = products.parallelStream()
        .filter(product -> product.getCategory() == Category.FOOD)
        .map(Product::getName)
        .collect(Collectors.toList());
```

Internal vs External Iteration

External iteration

```
for (int i = 0; i < products.size(); i++) {
    Product p = products.get(i);
    System.out.println(product);
}
```

Internal iteration

```
products.parallelStream()
        .forEach(System.out::println);
```

Not a Magic Solution

Thread management and communication overhead

Only likely to be beneficial when limited by CPU

Measure if it is benificial for your use case

Grouping-By in Parallel Streams

```java
Map<Category, List<Product>> productsByCategory =
    products.stream().collect(
        Collectors.groupingBy(Product::getCategory));
```

```java
Map<Category, List<Product>> productsByCategory =
    products.parallelStream().collect(
        Collectors.groupingByConcurrent(Product::getCategory));
```

Specialized Streams

Specialized Standard Functional Interfaces

XFunction<R>	XPredicate
XToYFunction	XConsumer
ToXFunction<T>	ObjXConsumer<T>
ToXBiFunction<T,U>	XSupplier
XUnaryOperator	XBinaryOperator

X, Y = Int, Long, Double

Extra: BooleanSupplier

Specialized Streams

IntStream

LongStream

DoubleStream

Course Summary

Working with Lambda Expressions

A lambda expression is an anonymous method

A lambda expression implements a functional interface

A functional interface has a single abstract method

Working with Lambda Expressions

Lambda Expression Syntax

```
(parameters) -> { body }
```

Capturing Variables

```
BigDecimal priceLimit = new BigDecimal("5.00");

Predicate<Product> isCheap =
    product -> product.getPrice().compareTo(priceLimit) < 0;
```

The meaning of "this" and "super" in a lambda expression

Working with checked exceptions in a lambda expression

Method References

Use a **method reference** instead of a lambda expression

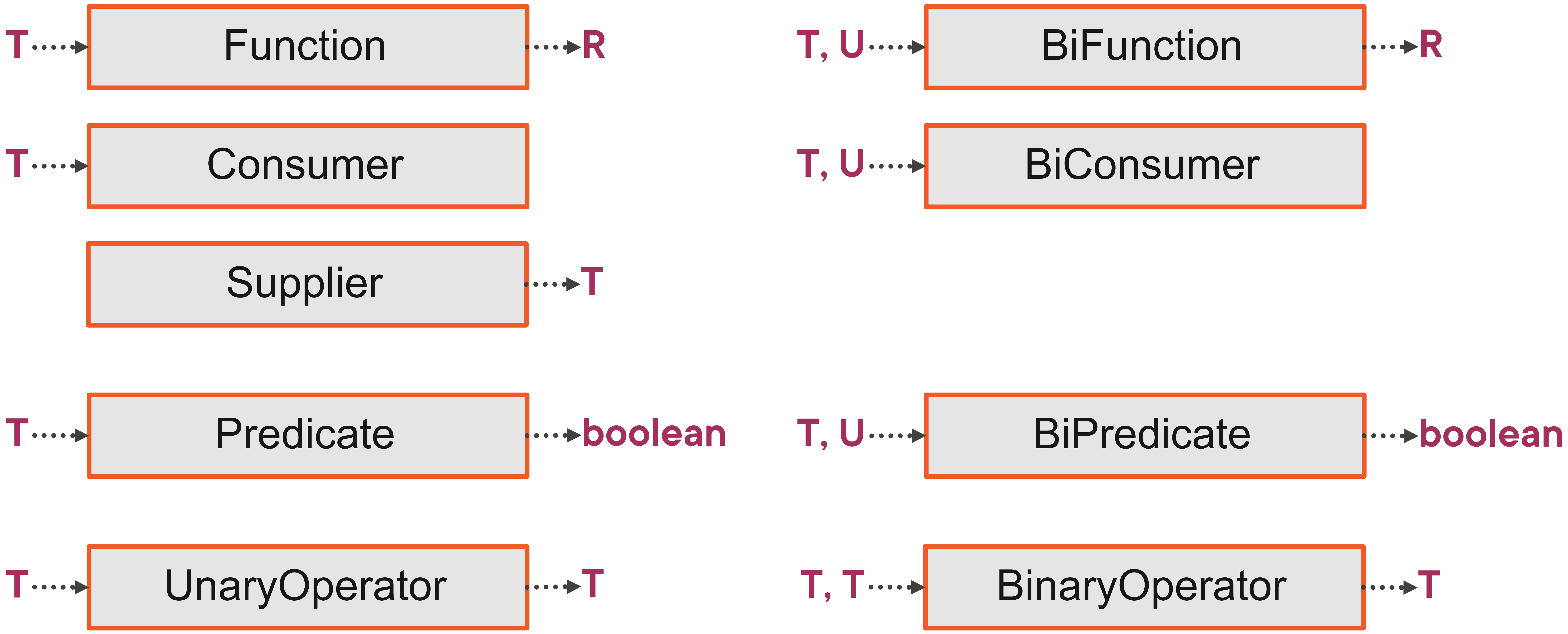
A method reference **implements a functional interface**

Refers to a **static** or **non-static method** or a **constructor**

Functional Interfaces

```
@FunctionalInterface
interface ProductFilter {
    boolean test(Product product);
}
```

Common Standard Functional Interfaces



Specialized Standard Functional Interfaces

XFunction<R>	XPredicate
XToYFunction	XConsumer
ToXFunction<T>	ObjXConsumer<T>
ToXBiFunction<T,U>	XSupplier
XUnaryOperator	XBinaryOperator

X, Y **= Int, Long, Double**

Extra: BooleanSupplier

Working with Streams – The Basics

```
products.stream()
```

Intermediate operations

```
        .filter(product -> product.getCategory() == Category.FOOD)
        .map(Product::getName)
```

Terminal operation

```
        .forEach(System.out::println);
```

Stream processing is lazy

Differences between Streams and Collections

Collection	Stream
Stores elements in a data structure	Does not store elements
Eager evaluation	Lazy evaluation
Imperative programming	Functional programming
Do modify the collection	Does not modify its source
Can be iterated multiple times	Iterating consumes the stream
Never infinite	May be infinite

Stream Operations

Filtering and Transforming

```
filter()
map()
flatMap()
```

Searching

```
findFirst()
findAny()
anyMatch()
allMatch()
noneMatch()
```

Reducing and Collecting

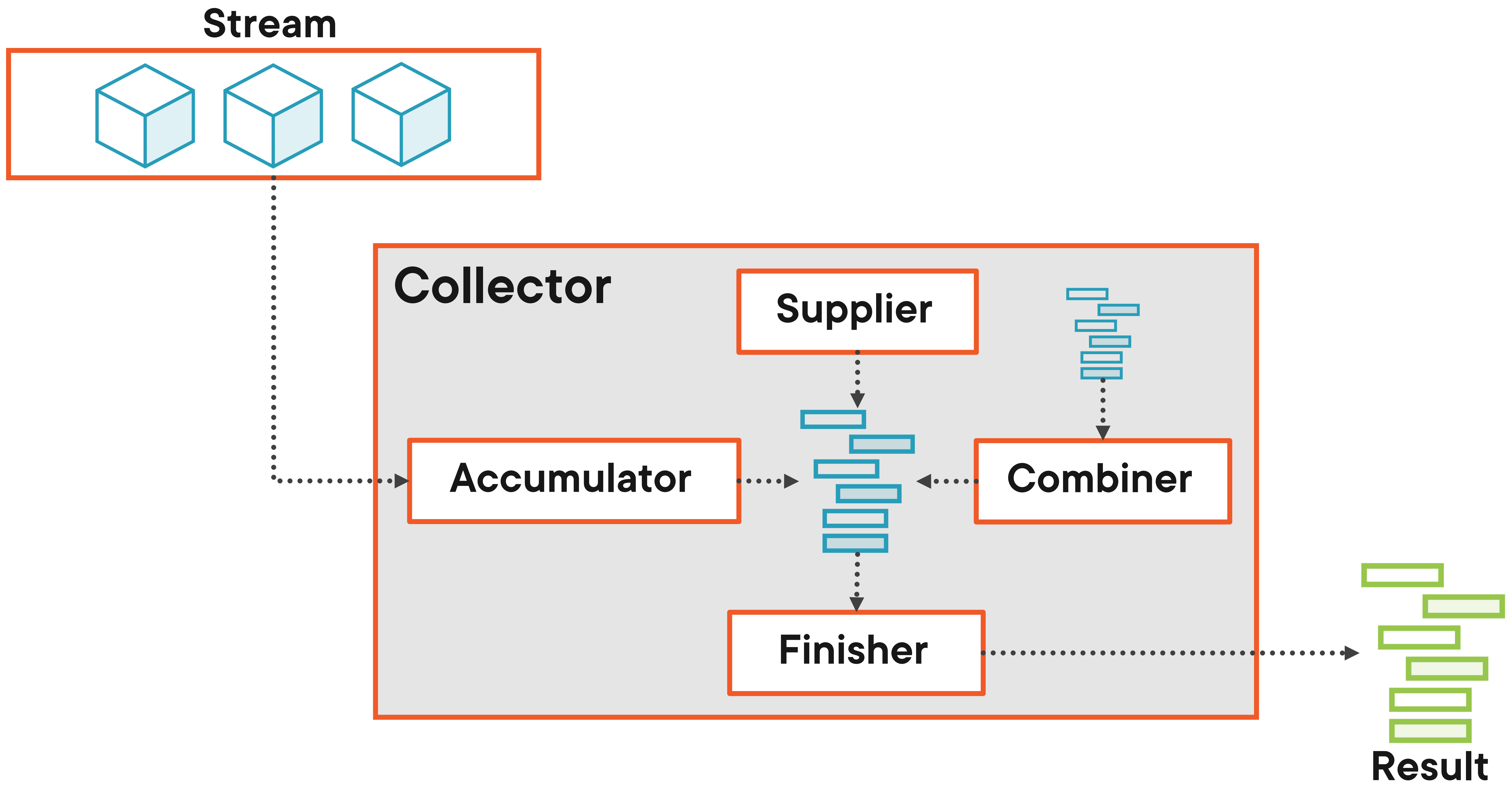
```
collect(Collectors.toList())
collect(Collectors.joining())
```

Reducing and Collecting in Detail

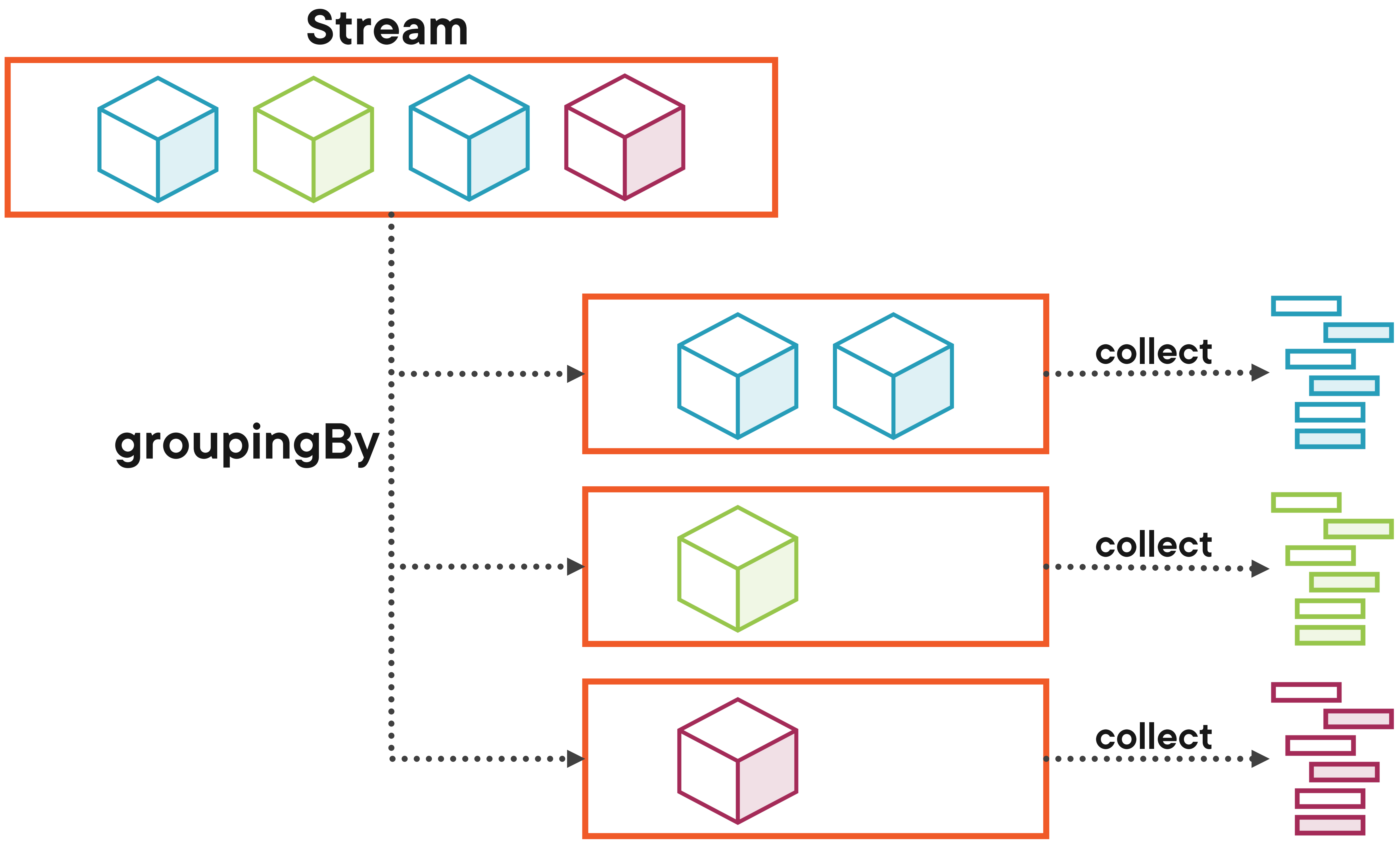
`reduce()` **Immutable reduction**

`collect()` **Mutable reduction**

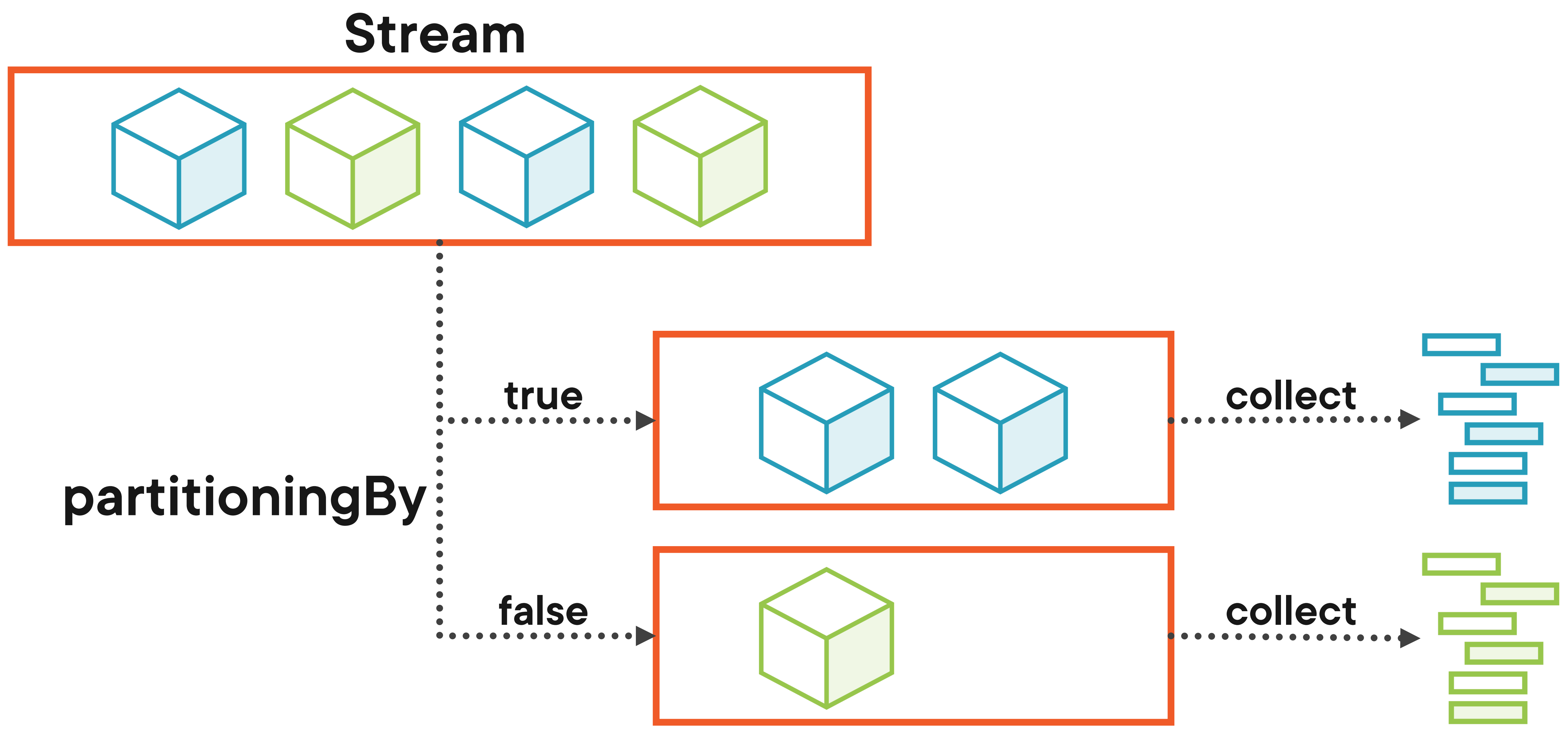
Collector Functions



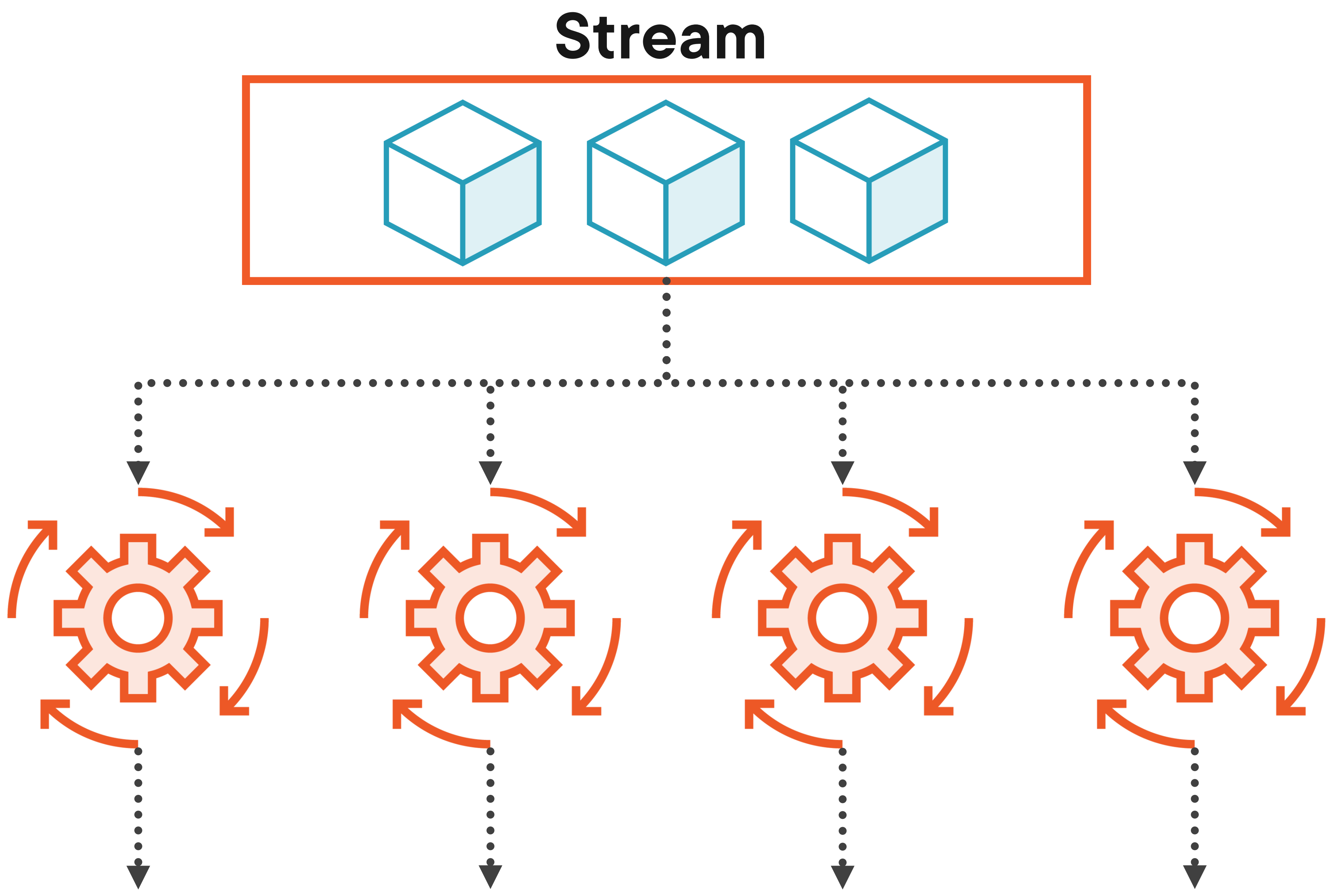
Grouping Stream Elements



Partitioning Stream Elements



Parallel Streams



Specialized Streams

IntStream

LongStream

DoubleStream

Lambda Expressions	Functional Interfaces	Method References
Streams	Stream Operations	Reduction and Collection
Collectors	Grouping and Partitioning	Parallel Streams