Multicore Programming

Java Streams (Advanced)

Louis-Claude Canon
louis-claude.canon@univ-fcomte.fr

Bureau 414C

Master 1 computer science – Semester 8

Outline

Special Streams

Collectors

Summary and References

Outline

Special Streams
- Numeric Streams
- Building Streams
- Infinite Streams

Collectors

Summary and References

Primitive Stream Specializations

- `IntStream`, `DoubleStream` and `LongStream`.
- Additional methods for efficiency and convenience: `min`, `max`, `sum`, `average`, etc.
- Avoid unboxing and boxing (more efficient data representation).

Java: Boxing and Unboxing

- Collection of primitive type are forbidden in Java (e.g. `List<int>`).
- Boxing consists in encapsulating a primitive type into an object (e.g. `Integer`).
- Boxing and unboxing are performed automatically when needed.
- Working on boxed values is costly.

To and From a Primitive Stream

mapToInt:

```
int calories = menu.stream()
    .mapToInt(Dish::getCalories)
    .sum();
```

boxed:

```
Stream<Integer> stream = intStream.boxed();
```

Specialized Optionals

- `OptionalInt`
- `OptionalDouble`
- `OptionalLong`

Outline

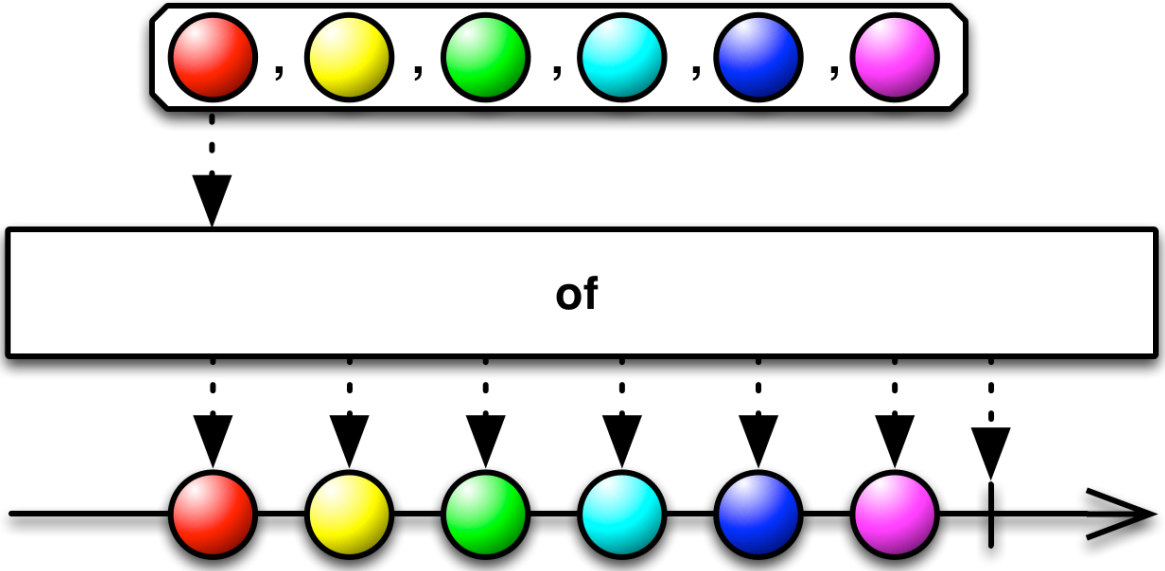
Special Streams
- Numeric Streams
- Building Streams
- Infinite Streams

Collectors

Summary and References

From Values

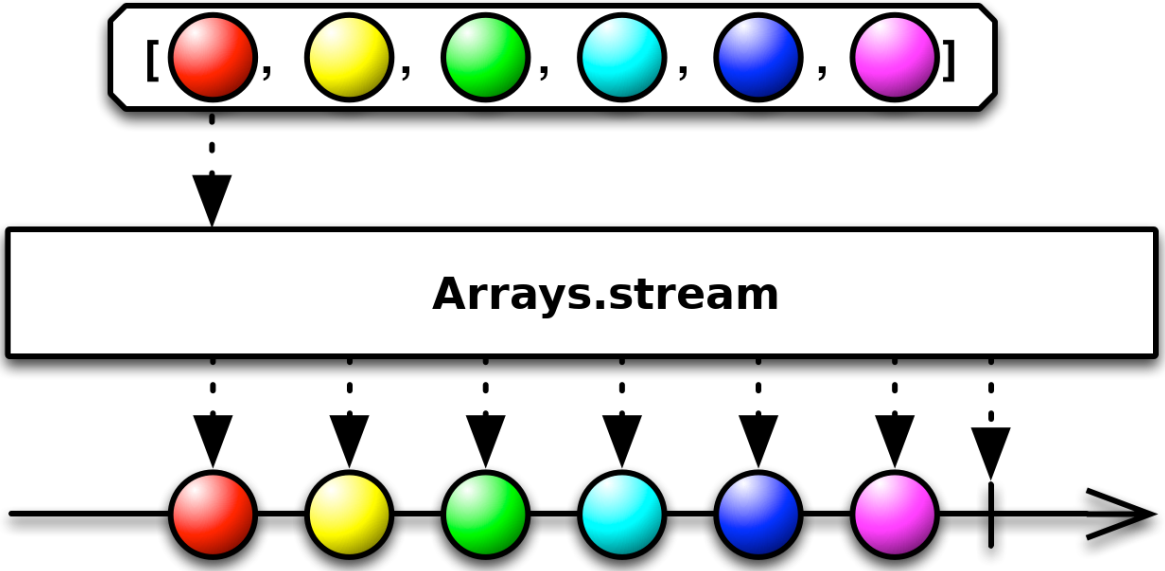
```
Stream<String> stream =
    Stream.of("Modern ", "Java ", "In ", "Action");
```



ReactiveX

From Arrays

```
int[] numbers = {2, 3, 5, 7, 11, 13};
int sum = Arrays.stream(numbers).sum();
```



ReactiveX

From a Collection

```
List<Integer> numbers = Arrays.asList(1, 2, 1, 3, 3, 2, 4);
int sum = numbers.stream().mapToInt(Integer::intValue).sum();
```

From Numeric Ranges

```
IntStream evenNumbers = IntStream.rangeClosed(1, 100)
    .filter(n -> n % 2 == 0);
```

- `range` is exclusive
- `rangeClosed` is inclusive

From Files

```
long uniqueWords = 0;
try (Stream<String> lines =
        Files.lines(Paths.get("data.txt"),
                    Charset.defaultCharset())) {
    uniqueWords = lines
        .flatMap(line -> Arrays.stream(line.split(" ")))
        .distinct()
        .count();
} catch (IOException e) {
}
```

- Construction `try-with-resources` de Java 7.

From Nullable

From Java 9:

```
String homeValue = System.getProperty("home");
Stream<String> homeValueStream =
    homeValue == null ? Stream.empty() : Stream.of(value);
```

```
Stream<String> homeValueStream =
    Stream.ofNullable(System.getProperty("home"));
```

A stream with a null value differs from an empty stream.

Outline

Special Streams
- Numeric Streams
- Building Streams
- Infinite Streams

Collectors

Summary and References

iterate

```
Stream.iterate(0, n -> n + 2)
    .limit(10)
    .forEach(System.out::println);
```

```
IntStream.iterate(0, n -> n < 100, n -> n + 4)
    .forEach(System.out::println);
```

- Needs a short-circuiting operation to terminate.
- No call to `sorted`.

generate

```
Stream.generate(Math::random)
    .limit(5)
    .forEach(System.out::println);
```

- Requires a stateful (with a mutable state) supplier.

Outline

Special Streams

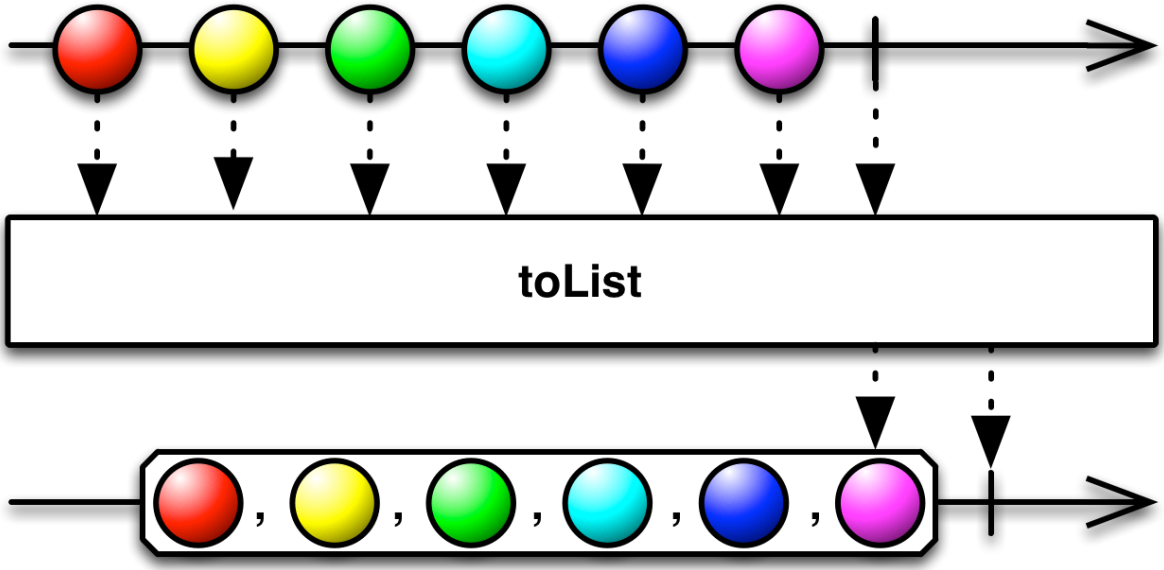
Collectors
- `collect`
- Reducing and Summarizing
- Other Single-Level Collectors
- Grouping and Partitioning
- Custom Collector

Summary and References

collect

- Collect the final result as the terminal operation.
- Require a `Collector` object as its parameter.
- A collector is designed to work on a stream.
- No relation to `Collection`.
- Several ways to define such collectors in class `Collectors`.

```
stream.collect(toList());
```



ReactiveX

Outline

Special Streams

Collectors
- `collect`
- Reducing and Summarizing
- Other Single-Level Collectors
- Grouping and Partitioning
- Custom Collector

Summary and References

maxBy and minBy

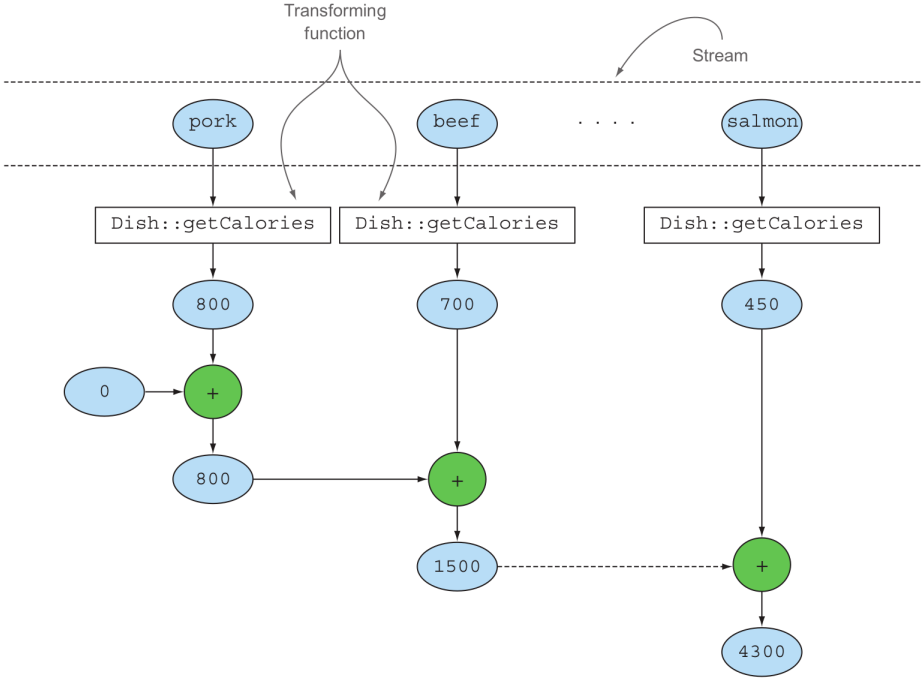
```
Comparator<Dish> dishCaloriesComparator =
    Comparator.comparing(Dish::getCalories);
Optional<Dish> mostCalorieDish = menu.stream()
    .collect(maxBy(dishCaloriesComparator));
```

Equivalence:

```
Optional<Dish> mostCalorieDish = menu.stream()
    .max(dishCaloriesComparator);
```

Summarizing: summingInt, counting, averagingInt and summarizingInt

```
int totalCalories = menu.stream()
    .collect(summingInt(Dish::getCalories));
long count = menu.stream()
    .collect(counting());
double avgCalories = menu.stream()
    .collect(averagingInt(Dish::getCalories));
IntSummaryStatistics menuStatistics = menu.stream()
    .collect(summarizingInt(Dish::getCalories));
```


Transforming function
Stream
pork
beef
. . . .
salmon
Dish::getCalories
Dish::getCalories
Dish::getCalories
800
700
450
0
+
800
+
1500
+
4300

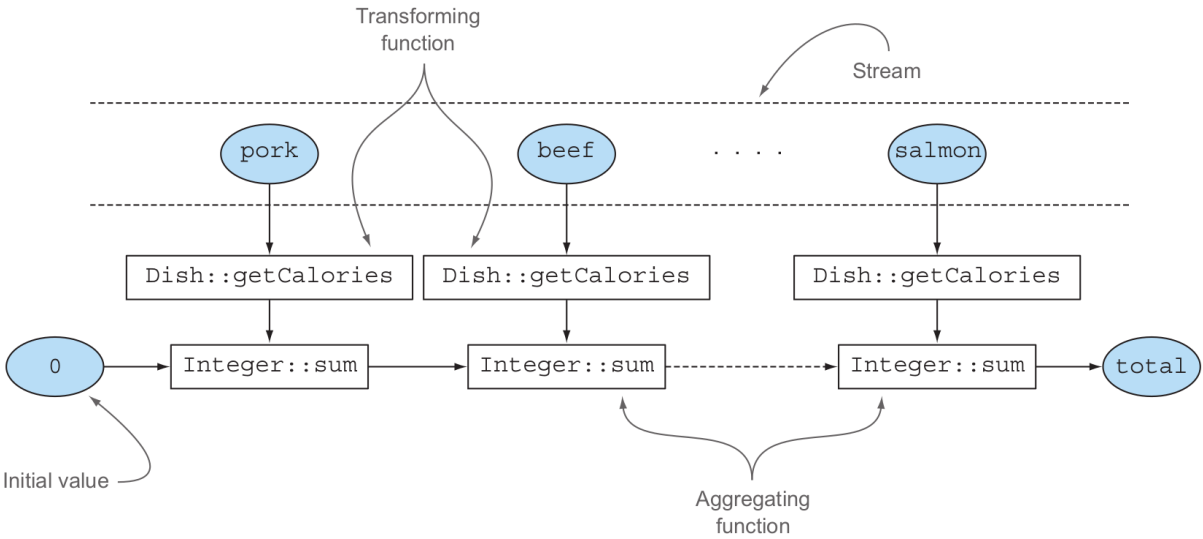
reducing

```
int totalCalories = menu.stream()
    .collect(reducing(0, Dish::getCalories,
                      Integer::sum));
```

- Generalization of previous summarizing operations.
- First apply map function (e.g. `getCalories`), and then perform the reduction operation (e.g. `sum`).

Equivalence:

```
int totalCalories = menu.stream()
    .mapToInt(Dish::getCalories)
    .reduce(0, Integer::sum);
```


Transforming function
Stream
pork
beef
. . . .
salmon
Dish::getCalories
Dish::getCalories
Dish::getCalories
0
Integer::sum
Integer::sum
Integer::sum
total
Initial value
Aggregating function

joining

```
String shortMenu = menu.stream()
    .map(Dish::getName)
    .collect(joining(", "));
```

Equivalence:

```
String shortMenu = menu.stream()
    .map(Dish::getName)
    .reduce((a, b) -> a + ", " + b)
    .orElse("")
```

Outline

Special Streams

Collectors
- collect
- Reducing and Summarizing
- Other Single-Level Collectors
- Grouping and Partitioning
- Custom Collector

Summary and References

filtering

```
List<Dish> caloricDishes = menu.stream()
    .collect(filtering(dish -> dish.getCalories() > 500,
        toList()));
```

Result:

```
[french fries, pizza, pork, beef]
```

Equivalence:

```
List<Dish> caloricDishes = menu.stream()
    .filter(dish -> dish.getCalories() > 500)
    .toList();
```

mapping

```
List<String> dishNames = menu.stream()
   .collect(mapping(Dish::getName,
       toList()));
```

- `flatMapping` when generating a stream for each object instead of an object only (merge the streams).

Equivalence:

```
List<String> dishNames = menu.stream()
    .map(Dish::getName)
    .toList();
```

collectingAndThen

To apply a specific function on the result of the collection operation:

```
Dish mostCaloric = menu.stream()
    .collect(collectingAndThen(
        maxBy(comparing(Dish::getCalories)),
        Optional::get));
```

Equivalence:

```
Dish mostCaloric = menu.stream()
    .max(comparing(Dish::getCalories))
    .get();
```

Intermediate operation, transformation and `collect`

`reducing` is similar to `reduce` except for the following:

- `reducing` can be used with multi-level reduction (see grouping and partitioning).
- The resulting object can be mutable:
  - `collect` builds incrementally a mutable object (more efficient for strings thanks to `StringBuilder`).
  - `reduce` cannot work on a mutable result object because it can be parallelized and all threads would then share the same initial value and work on the same accumulator.

Outline

Special Streams

Collectors
- `collect`
- Reducing and Summarizing
- Other Single-Level Collectors
- Grouping and Partitioning
- Custom Collector

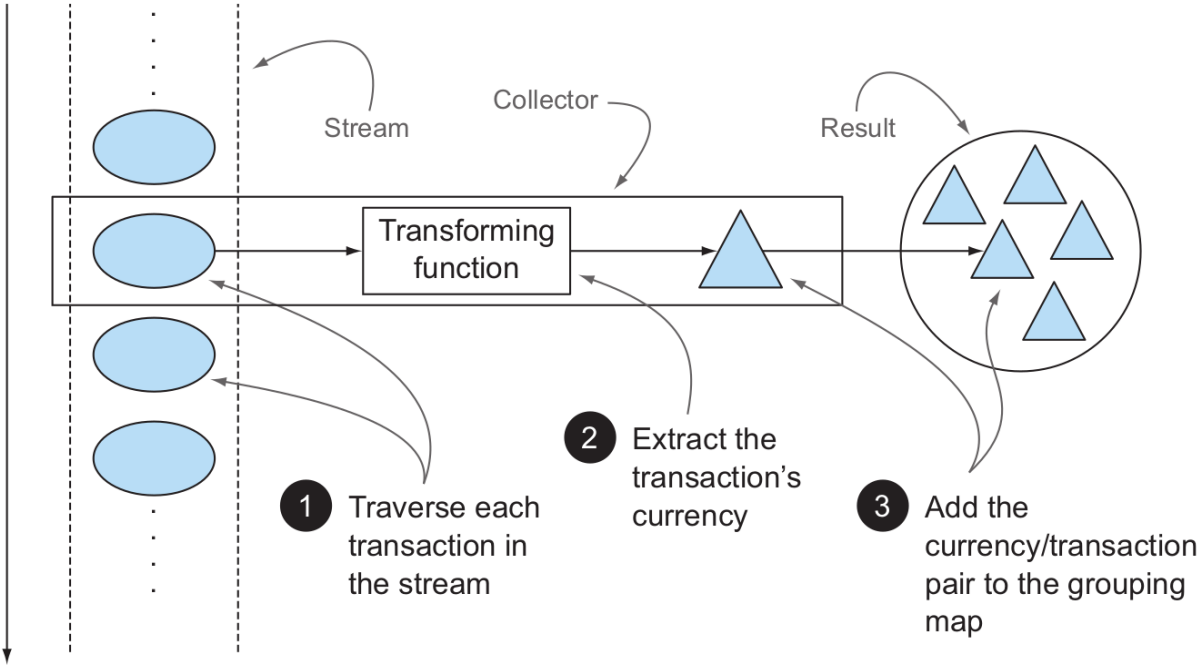
Summary and References

Example without Streams

```
Map<Currency, List<Transaction>> transByCurr =
    new HashMap<>();
for (Transaction transaction : transactions) {
    Currency currency = transaction.getCurrency();
    List<Transaction> transForCurr =
        transByCurr.get(currency);
    if (transForCurr == null) {
        transForCurr = new ArrayList<>();
        transByCurr.put(currency, transForCurr);
    }
    transForCurr.add(transaction);
}
```

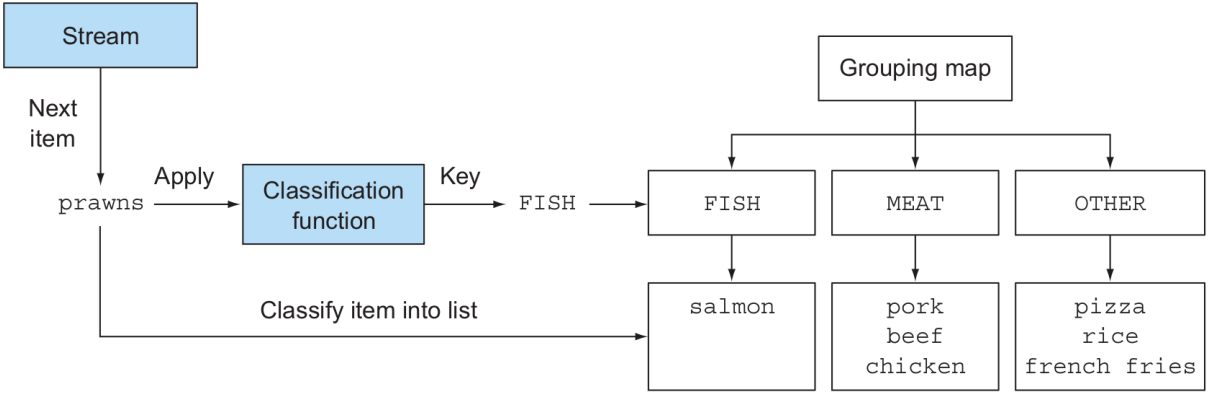
groupingBy

```
Map<Currency, List<Transaction>> transByCurr =
    transactions.stream()
        .collect(groupingBy(Transaction::getCurrency));
```


Stream
Collector
Result
Transforming function
1 Traverse each transaction in the stream
2 Extract the transaction's currency
3 Add the currency/transaction pair to the grouping map

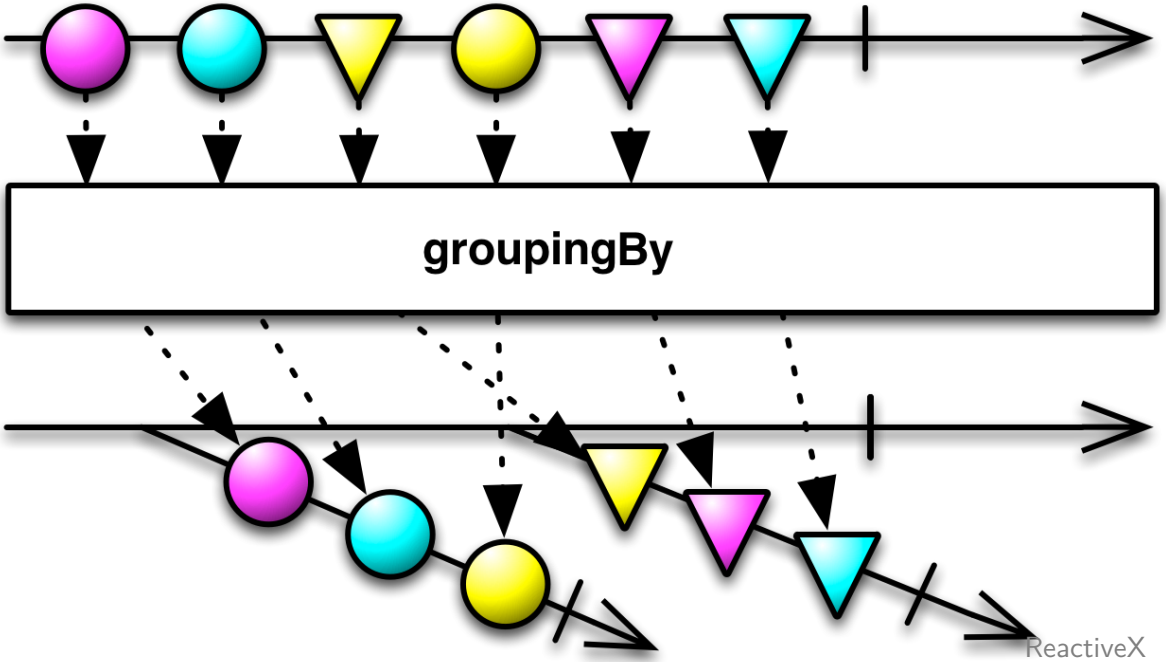
Second Example

```
Map<String, List<Dish>> dishesByType =
    menu.stream()
        .collect(groupingBy(Dish::getType));
```


Stream
Next item
prawns
Apply
Classification function
Key
FISH
Grouping map
FISH
MEAT
OTHER
salmon
pork
beef
chicken
pizza
rice
french fries
Classify item into list

Grouping without Classification Function

```
public enum CaloricLevel { DIET, NORMAL, FAT }
Map<CaloricLevel, List<Dish>> dishesByCaloricLevel =
    menu.stream()
        .collect(groupingBy(dish -> {
                if (dish.getCalories() <= 400)
                    return CaloricLevel.DIET;
                else if (dish.getCalories() <= 700)
                    return CaloricLevel.NORMAL;
                else return CaloricLevel.FAT;
            } ));
```


groupingBy
ReactiveX

filtering

```
Map<String, List<Dish>> caloricDishesByType =
    menu.stream()
        .collect(groupingBy(Dish::getType,
            filtering(dish -> dish.getCalories() > 500,
                toList())));
```

Result:

```
{OTHER=[french fries, pizza],
    MEAT=[pork, beef],
    FISH=[]}
```

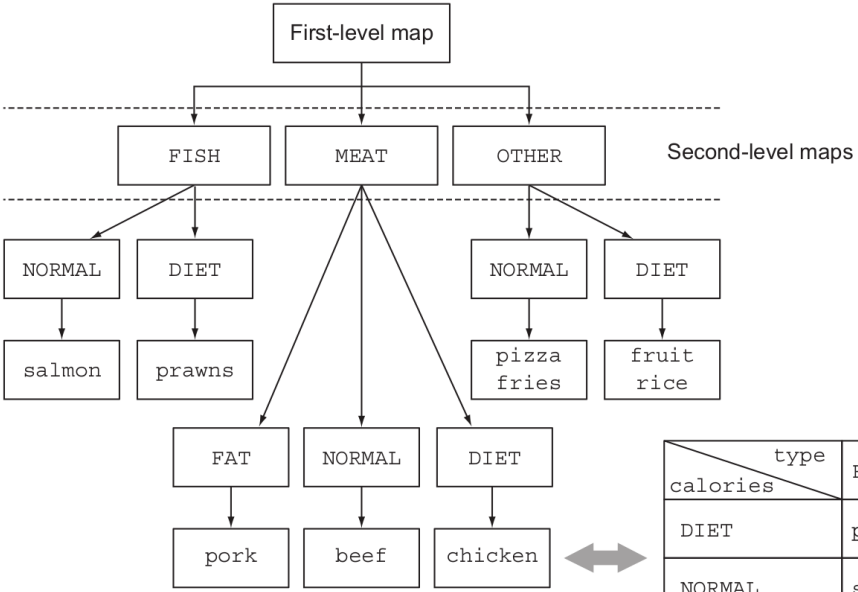
Note the empty group (unachievable with `filter`).

mapping

```
Map<String, List<String>> dishNamesByType =
    menu.stream()
        .collect(groupingBy(Dish::getType,
            mapping(Dish::getName, toList())));
```

Multilevel Grouping

```
Map<String, Map<CaloricLevel, List<Dish>>> res =
    menu.stream().collect(
        groupingBy(Dish::getType,
            groupingBy(dish -> {
                    if (dish.getCalories() <= 400)
                        return CaloricLevel.DIET;
                    else if (dish.getCalories() <= 700)
                        return CaloricLevel.NORMAL;
                    else return CaloricLevel.FAT;
                } )
        )
    );
```



calories \ type	FISH	MEAT	OTHER
DIET	prawns	chicken	fruit rice
NORMAL	salmon	beef	pizza fries
FAT		pork	

partitioningBy

```
Map<Boolean, List<Dish>> partitionedMenu =
    menu.stream()
        .collect(partitioningBy(Dish::isVegetarian));
```

- Always generate lists for both `true` and `false`.
- Special map optimized for two values.

Outline

Special Streams

Collectors
- `collect`
- Reducing and Summarizing
- Other Single-Level Collectors
- Grouping and Partitioning
- Custom Collector

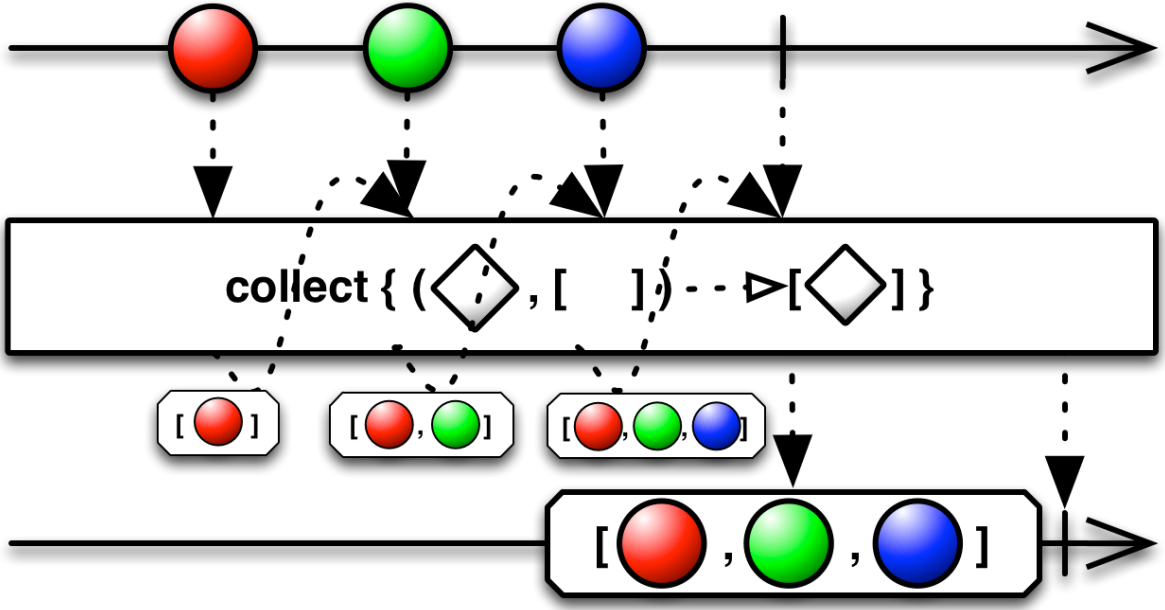
Summary and References

Collector Interface

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

- T stream element
- A accumulator element (temporary/intermediate result)
- R final returned element

Example in the following slides: `toList` (`A` and `R` are both `List<T>`).


collect { (, [ ]) - - ▷[ ] }
[ ]
[,]
[, ,]
[, ,]
ReactiveX

Supplier

Initial value for the accumulator:

```
public Supplier<List<T>> supplier() {
    return ArrayList<T>::new;
}
```

Accumulator

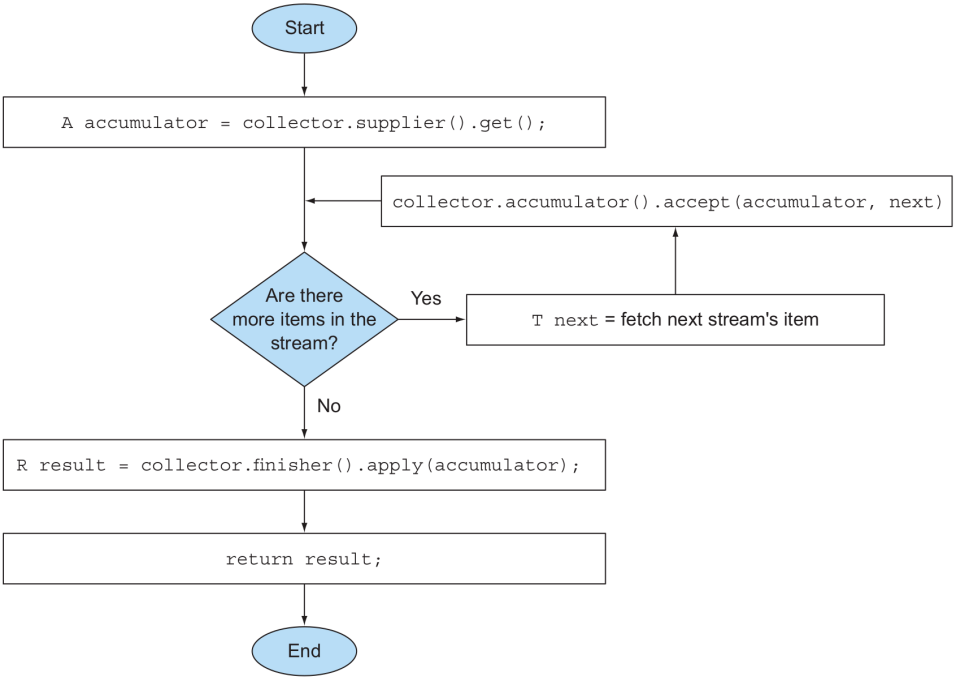
Take an element and merge it with the accumulator:

```
public BiConsumer<List<T>, T> accumulator() {
    return List::add;
}
```

Finisher

Convert the accumulator to the final result:

```
public Function<List<T>, List<T>> finisher() {
    return Function.identity();
}
```

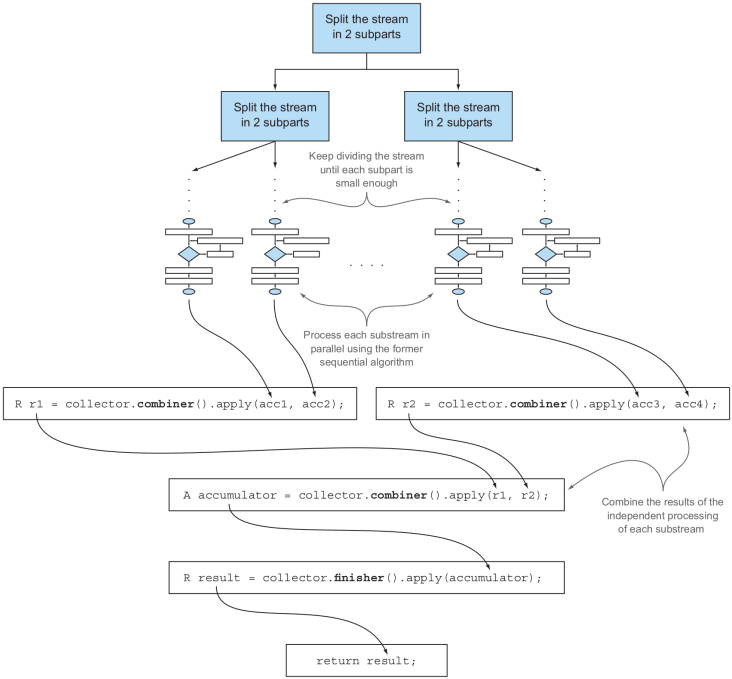

Start
A accumulator = collector.supplier().get();
collector.accumulator().accept(accumulator, next)
Are there more items in the stream?
Yes
T next = fetch next stream's item
No
R result = collector.finisher().apply(accumulator);
return result;
End

Combiner

Combine two accumulators:

```
public BinaryOperator<List<T>> combiner() {
    return (list1, list2) -> {
        list1.addAll(list2);
        return list1;
    }
}
```

- Enable the parallelization: each core processes a part of the stream, which leads to a set of accumulators that are combined.


Split the stream in 2 subparts
Split the stream in 2 subparts
Split the stream in 2 subparts
Keep dividing the stream until each subpart is small enough
Process each substream in parallel using the former sequential algorithm
R r1 = collector.combiner().apply(acc1, acc2);
R r2 = collector.combiner().apply(acc3, acc4);
A accumulator = collector.combiner().apply(r1, r2);
Combine the results of the independent processing of each substream
R result = collector.finisher().apply(accumulator);
return result;

Characteristics

UNORDERED do not commit to preserve order

CONCURRENT can be parallelized (on all streams if UNORDERED, on unordered streams only otherwise)

IDENTITY_FINISH finisher is the identity function

```
public Set<Characteristics> characteristics() {
    return Collections.unmodifiableSet(EnumSet.of(
        IDENTITY_FINISH, CONCURRENT));
}
```

Alternative

```
List<Dish> dishes = menuStream
    .collect(ArrayList<Dish>::new,
        List::add,
        List::addAll);
```

Collector	argument	chaining
`toList/toSet`		
`minBy/maxBy`	`Comparator`	
`summingInt`	`keyExtractor`	
`averagingInt`	`keyExtractor`	
`summarizingInt`	`keyExtractor`	
`counting`		
`reducing`	...	
`joining`	`String`	
`filtering`	`Predicate`	✓
`mapping`	`Function`	✓
`flatMapping`	`Function`	✓
`collectingAndThen`	`Function`	✓
`groupingBy`	`Function`	✓
`partitioningBy`	`Predicate`	✓

Outline

Special Streams

Collectors

Summary and References

Official Documentation

- Documentation of interface `Collector`
- Documentation of class `Collectors`

Demonstration

Compute the sum of all numbers between 1 and n which square is below 20:

```
IntStream.rangeClosed(1, n)
    .filter(i -> i < 20)
    .sum()
```

Special Streams

- Primitive specializations of streams: `IntStream`, `DoubleStream`, and `LongStream`.
- Other sources: from values, arrays, files, etc.
- Infinite stream: `iterate` and `generate` with short-circuiting.

Collectors

- `collect` accumulates the elements of a stream into a summary result.
- Collectors compose to create multilevel groupings, partitions, and reductions (`groupingBy`, `partitioningBy`, etc.).
- Custom collectors must implement the methods defined in the `Collector` interface.