Lecture 19

Collecting Data with Streams

SOEN 6441, Summer 2018

René Witte
Department of Computer Science
and Software Engineering
Concordia University

Outline

René Witte

Concordia University

Grouping transactions by currency

René Witte
Concordia University

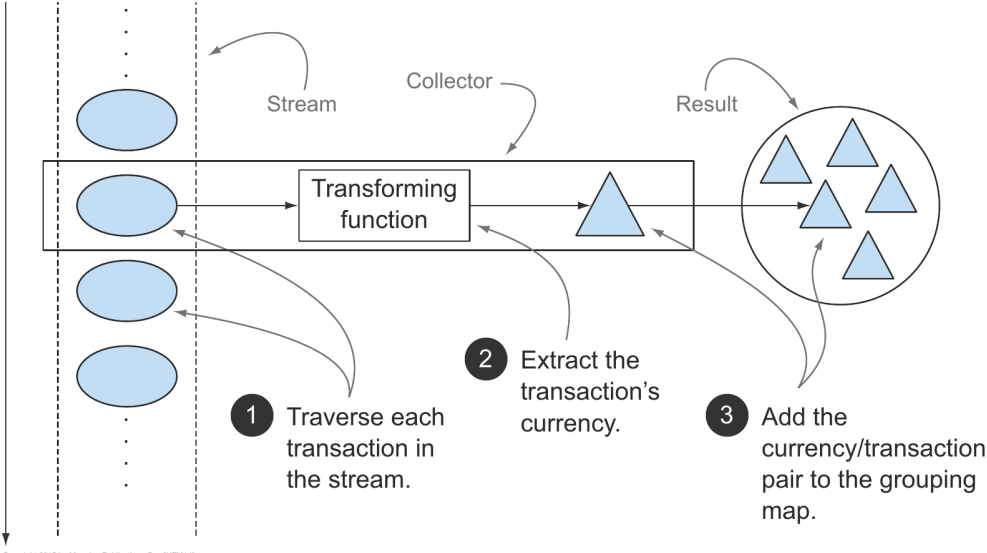
Java 7

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

Java 8

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

Collectors as advanced reductions



René Witte
Concordia University

Predefined Collectors

René Witte
Concordia University

Example: Collect `Stream` to `List`

```
List<Transaction> transactions =
    transactionStream.collect(Collectors.toList());
```

`Collectors` class

- Reducing and summarizing stream elements to a single value
- Grouping elements
- Partitioning elements

Example Data: Menu items

```
List<Dish> menu = Arrays.asList(
  new Dish("pork", false, 800, Dish.Type.MEAT),
  new Dish("beef", false, 700, Dish.Type.MEAT),
  new Dish("chicken", false, 400, Dish.Type.MEAT),
  new Dish("french␣fries", true, 530, Dish.Type.OTHER),
  new Dish("rice", true, 350, Dish.Type.OTHER),
  new Dish("season␣fruit", true, 120, Dish.Type.OTHER),
  new Dish("pizza", true, 550, Dish.Type.OTHER),
  new Dish("prawns", false, 300, Dish.Type.FISH),
  new Dish("salmon", false, 450, Dish.Type.FISH) );
```

René Witte
Concordia University

Example Class: Dish

```
public class Dish {
  private final String name;
  private final boolean vegetarian;
  private final int calories;
  private final Type type;

  public Dish(String name, boolean vegetarian, int calories, Type type) {
    this.name = name;
    this.vegetarian = vegetarian;
    this.calories = calories;
    this.type = type;
  }

  public String getName() { return name; }
  public boolean isVegetarian() { return vegetarian; }
  public int getCalories() { return calories; }
  public Type getType() { return type; }

  @Override
  public String toString() {
    return name;
  }

  public enum Type { MEAT, FISH, OTHER }
}
```

René Witte

Counting

René Witte

Count number of dishes in the menu

```
long howManyDishes = menu.stream().collect(Collectors.counting());
```

vs.

```
long howManyDishes = menu.stream().count();
```

Import the static factory methods

```
import static java.util.stream.Collectors.*;

long howManyDishes = menu.stream().collect(counting());
```

Finding maximum and minimum in a stream of values

René Witte
Concordia University

`maxBy` and `minBy`

Both take a `Comparator` as argument.

Find the highest-calorie dish

```
Comparator<Dish> dishCaloriesComparator =
    Comparator.comparingInt(Dish::getCalories);

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

Summarization

René Witte
Concordia University

Summing values

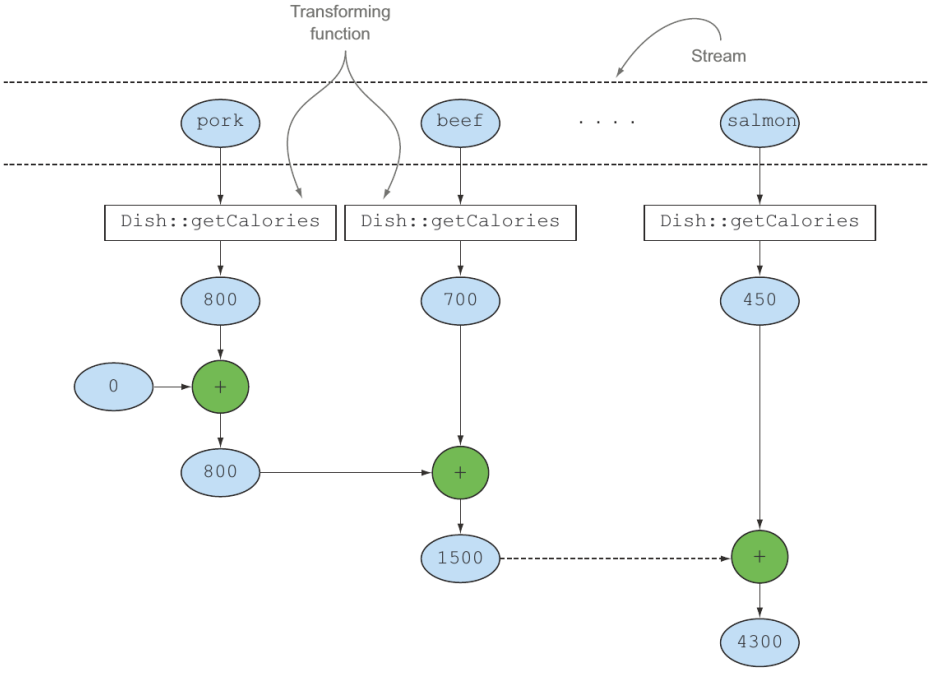
`Collectors.summingInt`

- Accepts function that maps object into `int`
- Returns collector that performs the summarization

Total number of calories in the menu

```
int totalCalories
    = menu.stream().collect(summingInt(Dish::getCalories));
```

The aggregation process of the `summingInt` collector



René Witte

Other Summarizations

Averaging

`averagingInt`, `averagingLong` and `averagingDouble`

Example

```
double avgCalories =
    menu.stream().collect(averagingInt(Dish::getCalories));
```

All-at-once summarizing

René Witte
Concordia University

Compute count, sum, minimum, average, and maximum

Use the `summarizingInt, summarizingLong, summarizingDouble` factory methods.

Example

```
IntSummaryStatistics menuStatistics =
    menu.stream().collect(summarizingInt(Dish::getCalories));
```

Result object

```
IntSummaryStatistics{
    count=9, sum=4300, min=120, average=477.777778, max=800}
```

Joining Strings

René Witte
Concordia University

Concatenate Strings

Use the `joining` factory method.

Example: concatenate all dish names

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

or (using `toString()` of `Dish`):

```
String shortMenu = menu.stream().collect(joining());
```

Result

```
porkbeefchickenfrench friesriceseason fruitpizzaprawnssalmon
```

Joining with Delimiter

René Witte

Overloaded `joining`

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

Result

```
pork, beef, chicken, french fries, rice,
season fruit, pizza, prawns, salmon
```

Generalized summarization with reduction

René Witte
Concordia University

`Collectors.reducing`

All previous collectors are specializations of `Collectors.reducing`:

- First argument: starting value (also value of empty stream)
- Second argument: function to transform object
- Third argument: `BinaryOperator` to sum two items into one

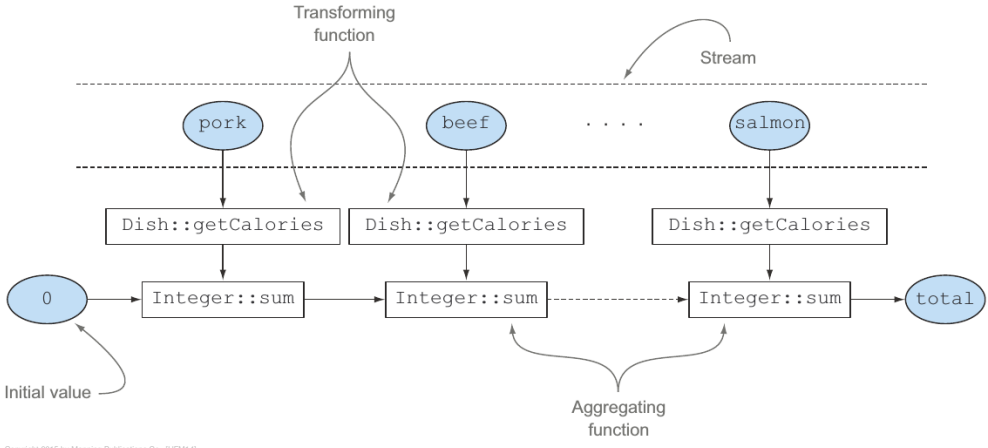
Example: Summing total calories

```
int totalCalories
    = menu.stream()
          .collect(reducing(0, Dish::getCalories, (i, j) -> i + j));
```

or (using `Integer::sum`):

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

The reduction process calculating the total number of calories in the menu



René Witte

`counting` implementation

```
public static <T> Collector<T, ?, Long> counting() {
  return reducing(0L, e -> 1L, Long::sum);
}
```

René Witte

One-argument `reducing`

René Witte

Example: find highest-calorie dish

```
Optional<Dish> mostCalorieDish =
    menu.stream()
        .collect(reducing((d1, d2)
            -> d1.getCalories() > d2.getCalories() ? d1 : d2));
```

Special case of three-argument version

- First argument is first element in stream
- Second argument is the *identity function*

reducing vs. reduce

René Witte
Concordia University

With reducing

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

With reduce

```
int totalCalories =
    menu.stream().map(Dish::getCalories).reduce(Integer::sum).get();
```

With stream specialization

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

Outline

René Witte
Concordia University

Grouping

René Witte

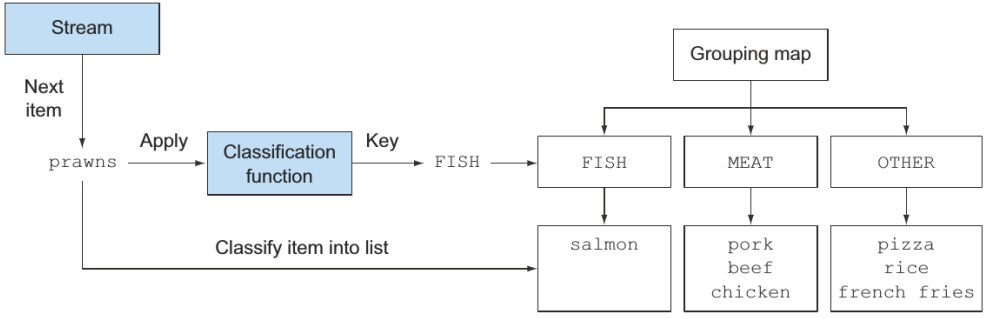
Task: Group dishes by type (`[MEAT, FISH, OTHER]`)

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

Result

```
{FISH=[prawns, salmon],
 OTHER=[french fries, rice, season fruit, pizza],
 MEAT=[pork, beef, chicken]}
```

Classification of an item in the stream during the grouping process



René Witte

Grouping with custom lambda

René Witte
Concordia University

Group dishes into 'diet' types

```
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;
        }));
```

Multilevel grouping

René Witte
Concordia University

Two-argument `Collectors.groupingBy` factory method

- First argument: classification function
- Second argument: `collector`

Example: Group dished first by type, then by calories

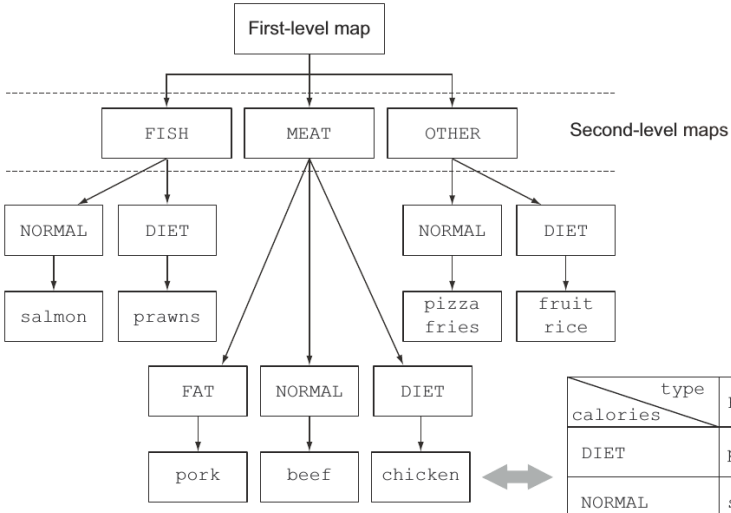
```
Map<Dish.Type, Map<CaloricLevel, List<Dish>>> dishesByTypeCaloricLevel =
  menu.stream()
      .collect(
        groupingBy(Dish::getType,
          groupingBy(dish -> {
            if (dish.getCalories() <= 400) return CaloricLevel.DIET;
            else if (dish.getCalories() <= 700) return CaloricLevel.NORMAL;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

Result

```
{MEAT={DIET=[chicken], NORMAL=[beef], FAT=[pork]},
 FISH={DIET=[prawns], NORMAL=[salmon]},
 OTHER={DIET=[rice, seasonal fruit], NORMAL=[french fries, pizza]}}
```

Equivalence between *n*-level nested map and *n*-dimensional classification table

René Witte



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

Collecting data in subgroups

René Witte
Concordia University

Second collector can be any type

```
Map<Dish.Type, Long> typesCount
    = menu.stream().collect(groupingBy(Dish::getType, counting()));
```

Result

```
{MEAT=3, FISH=2, OTHER=4}
```

Another example

Find highest-calorie dish, classified by type

```
Map<Dish.Type, Optional<Dish>> mostCaloricByType =
    menu.stream()
        .collect(groupingBy(Dish::getType,
                            maxBy(comparingInt(Dish::getCalories))));
```

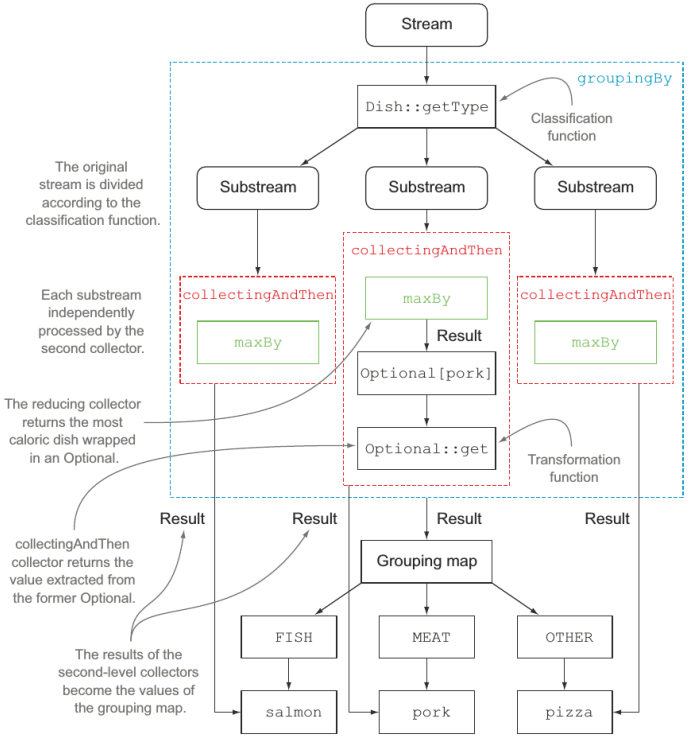
Result

```
{FISH=Optional[salmon], OTHER=Optional[pizza], MEAT=Optional[pork]}
```

Get rid of the `Optional` using `collectingAndThen`

```
Map<Dish.Type, Dish> mostCaloricByType =
    menu.stream()
        .collect(groupingBy(Dish::getType,
                            collectingAndThen(
                              maxBy(comparingInt(Dish::getCalories)),
                            Optional::get)));
```

René Witte



René Witte

Combine `groupingBy` and `mapping`

René Witte

`mapping` Collector

- First argument: transformation function
- Second argument: collector for result objects

Which `CaloricLevels` are available in the menu for each type of `Dish`?

```
Map<Dish.Type, Set<CaloricLevel>> caloricLevelsByType =
  menu.stream()
      .collect(groupingBy(Dish::getType, mapping(
        dish -> {
          if (dish.getCalories() <= 400) return CaloricLevel.DIET;
          else if (dish.getCalories() <= 700) return CaloricLevel.NORMAL;
          else return CaloricLevel.FAT;
        },
        toSet()
      )));
```

Result

```
{OTHER=[DIET, NORMAL], MEAT=[DIET, NORMAL, FAT], FISH=[DIET, NORMAL]}
```

Outline

René Witte
Concordia University

Partitioning

René Witte
Concordia University

Definition

Partitioning: grouping with a predicate, called partitioning function

Example: Partition menu into vegetarian and non-vegetarian dishes

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

Result

```
{false=[pork, beef, chicken, prawns, salmon],
 true=[french fries, rice, season fruit, pizza]}
```

Advantages of partitioning

René Witte
Concordia University

Partitioning vs. filtering

- With partitioning, we keep both `true` and `false` elements
- E.g., access vegetarian (`true`) results with

```
List<Dish> vegetarianDishes = partitionedMenu.get(true);
```

- Can combine with second collector, e.g.:

```
Map<Boolean, Map<Dish.Type, List<Dish>>> vegetarianDishesByType =
  menu.stream()
      .collect(partitioningBy(Dish::isVegetarian,
                              groupingBy(Dish::getType)));
```

resulting in:

```
{false={FISH=[prawns, salmon], MEAT=[pork, beef, chicken]},
 true={OTHER=[french fries, rice, season fruit, pizza]}}
```

Partitioning numbers into prime and nonprime

First try

```
public boolean isPrime(int candidate) {
  return IntStream.range(2, candidate)
                  .noneMatch(i -> candidate % i == 0);
}
```

Optimization: Only test for factors less than or equal to the square root

```
public boolean isPrime(int candidate) {
  int candidateRoot = (int) Math.sqrt((double) candidate);
  return IntStream.rangeClosed(2, candidateRoot)
                  .noneMatch(i -> candidate % i == 0);
}
```

Use `partitioningBy` collector

```
public Map<Boolean, List<Integer>> partitionPrimes(int n) {
  return IntStream.rangeClosed(2, n).boxed()
                  .collect(
                    partitioningBy(candidate -> isPrime(candidate)));
}
```

René Witte

Outline

René Witte

Concordia University

René Witte
Concordia University

The `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();
}
```

Here,

- `T` generic type of the items in the stream to be collected
- `A` type of the accumulator object
- `R` type of the resulting object (e.g., collection)

Example: `ToListCollector<T>`

```
public class ToListCollector<T> implements Collector<T,List<T>,List<T>>
```

Making a new result container

René Witte
Concordia University

The `supplier()` method

Returns `Supplier` of an empty result
(used as empty accumulator during collection)

Example: In `ToListCollector<T>`

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

or just

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

Adding an element to a result container

René Witte
Concordia University

The `accumulator()` method

Returns the function that performs the reduction operation on elements (*acc*, *n*) of the stream (changing the internal state of the accumulator)

Example: In `ToListCollector<T>`

```
public BiConsumer<List<T>, T> accumulator() {
    return (list, item) -> list.add(item);
}
```

or just

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

Applying the final transformation to the result container

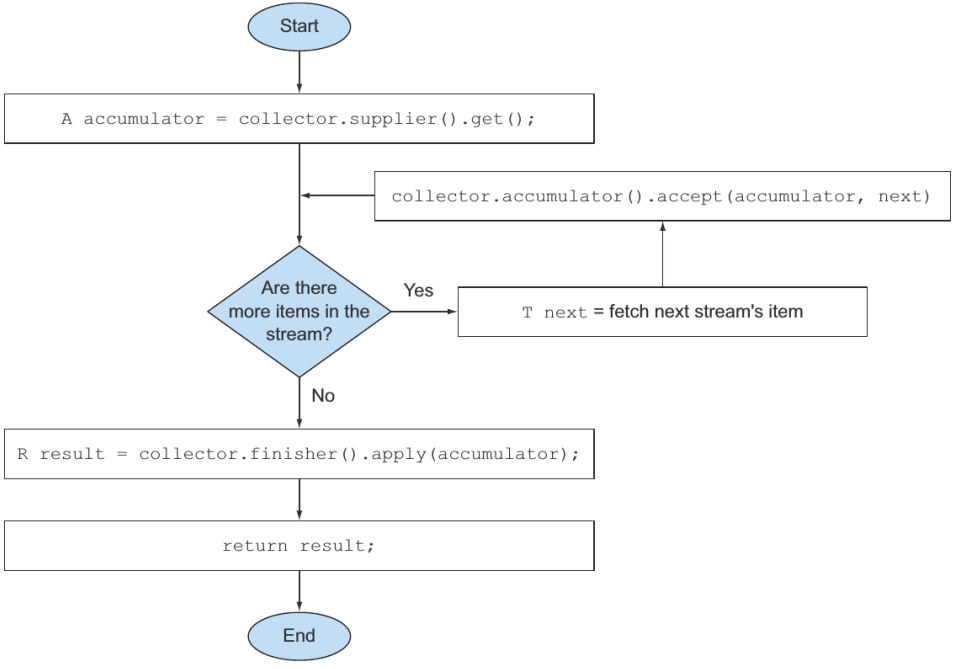
The `finisher()` method

Return the function invoked at the end in order to transform the accumulator object into the final result of the whole collection operation (can be the same as accumulator).

Example: In `ToListCollector<T>`

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

Logical steps of the sequential reduction process



René Witte

Merging two result containers

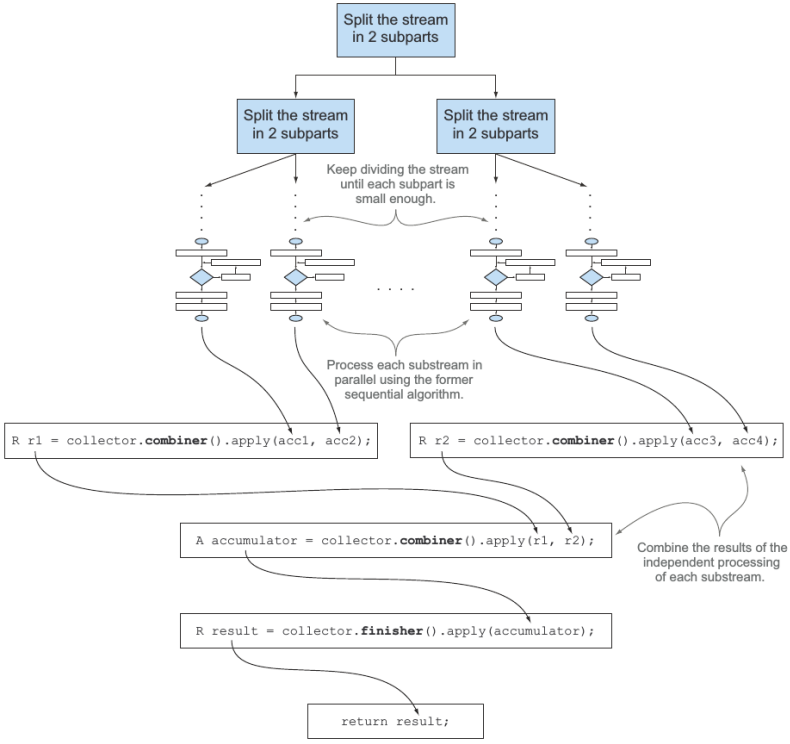
René Witte
Concordia University

The `combiner()` method

Defines how subparts of the stream are combined when processed in parallel (allowing parallel reduction of streams)

Example: In `ToListCollector<T>`

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



René Witte

The Characteristics Method

René Witte
Concordia University

Characteristics

The method `characteristics` returns an immutable `Set<Characteristics>`, used in optimizing parallel operations:

`UNORDERED` — The result of the reduction isn't affected by the order in which the items in the stream are traversed and accumulated

`CONCURRENT` — The `accumulator` function can be called concurrently from multiple threads, and then this collector can perform a parallel reduction of the stream.

`IDENTITY_FINISH` — Finisher method is the identity one, can be omitted and simply do an unchecked cast from the accumulator `A` to the result `R`

Putting them all together

```
import java.util.*;
import java.util.function.*;
import java.util.stream.Collector;
import static java.util.stream.Collector.Characteristics.*;

public class ToListCollector<T> implements Collector<T, List<T>, List<T>> {
  @Override
  public Supplier<List<T>> supplier() {
    return ArrayList::new;
  }
  @Override
  public BiConsumer<List<T>, T> accumulator() {
    return List::add;
  }
  @Override
  public Function<List<T>, List<T>> finisher() {
    return Function.identity();
  }
  @Override
  public BinaryOperator<List<T>> combiner() {
    return (list1, list2) -> {
      list1.addAll(list2);
      return list1;
    };
  }
  @Override
  public Set<Characteristics> characteristics() {
    return Collections.unmodifiableSet(EnumSet.of(
      IDENTITY_FINISH, CONCURRENT));
  }
}
```

René Witte

Custom `collect` without creating a `Collector` implementation

René Witte
Concordia University

Overloaded `collect` method

Special case for `IDENTITY_FINISH`:

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

Example

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

Summary

René Witte
Concordia University

Collecting Data with Streams

- `collect` is a terminal operation that takes as argument various recipes (called collectors) for accumulating the elements of a stream into a summary result
- Predefined collectors include reducing and summarizing stream elements into a single value (e.g., minimum, maximum, or average)
- Predefined collectors let you group elements of a stream with `groupingBy` and partition elements of a stream with `partitioningBy`
- Collectors compose effectively to create multilevel groupings, partitions, and reductions
- You can develop your own collectors by implementing the methods defined in the `Collector` interface

Outline

Reading Material

René Witte

Required

- [UFM14, Chapter 6] (Collecting data with streams)

Supplemental

- [War14, Chapter 5] (Advanced Collections and Collectors)

References

[UFM14] Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft.
Java 8 in Action: Lambdas, streams, and functional-style programming.
Manning Publications, 2014.
https://www.manning.com/books/java-8-in-action.

[War14] Richard Warburton.
Java 8 Lambdas.
O'Reilly, 2014.