6 Collecting data with streams

This chapter covers

- Creating and using a collector with the `Collectors` class
- Reducing streams of data to a single value
- Summarization as a special case of reduction
- Grouping and partitioning data
- Developing your own custom collectors

- You can view Java 8 streams as fancy lazy iterators of sets of data.
- Java 8 streams support two types of operations
  - Intermediate operations such as filter or map
    - set up a pipeline of streams.
  - terminal operations such as count, findFirst, forEach, and reduce.
    - do consume from a stream
- collect is a reduction operation, like reduce, that takes as an argument various recipes for accumulating the elements of a stream into a summary result.
- Collection, Collector, and collect

- Using `collect` and collectors:
  - Group a list of transactions by currency to obtain the sum of the values of all transactions with that currency (returning a `Map<Currency, Integer>`)
  - Partition a list of transactions into two groups: expensive and not expensive (returning a `Map<Boolean, List<Transaction>>`)
  - Create multilevel groupings, such as grouping transactions by cities and then further categorizing by whether they're expensive or not (returning a `Map<String, Map<Boolean, List<Transaction>>>`)

Listing 6.1 Grouping transactions by currency in imperative style

Iterates the List of Transactions

Creates the Map where the grouped transaction will be accumulated

```
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);
}
```

Extracts the Transaction's currency

If there's no entry in the grouping Map for this currency, creates it

Adds the currently traversed Transaction to the List of Transactions with the same currency

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

6.1 Collectors in a nutshell

- The argument passed to the `collect` method is an implementation of the `Collector` interface.
  - **`toList`** and **`groupingBy`**
- *`Collectors`* as advanced reduction
  - Invoking the `collect` method on a stream triggers a reduction operation (parameterized by a `Collector`) on the elements of the stream itself.
  - Typically, the `Collector` applies a transforming function to the element.
- The implementation of the methods of the `Collector` interface defines how to perform a reduction operation on a stream .
- ```
  List<Transaction> transactions =
      transactionStream.collect(Collectors.toList());
  ```
  ```

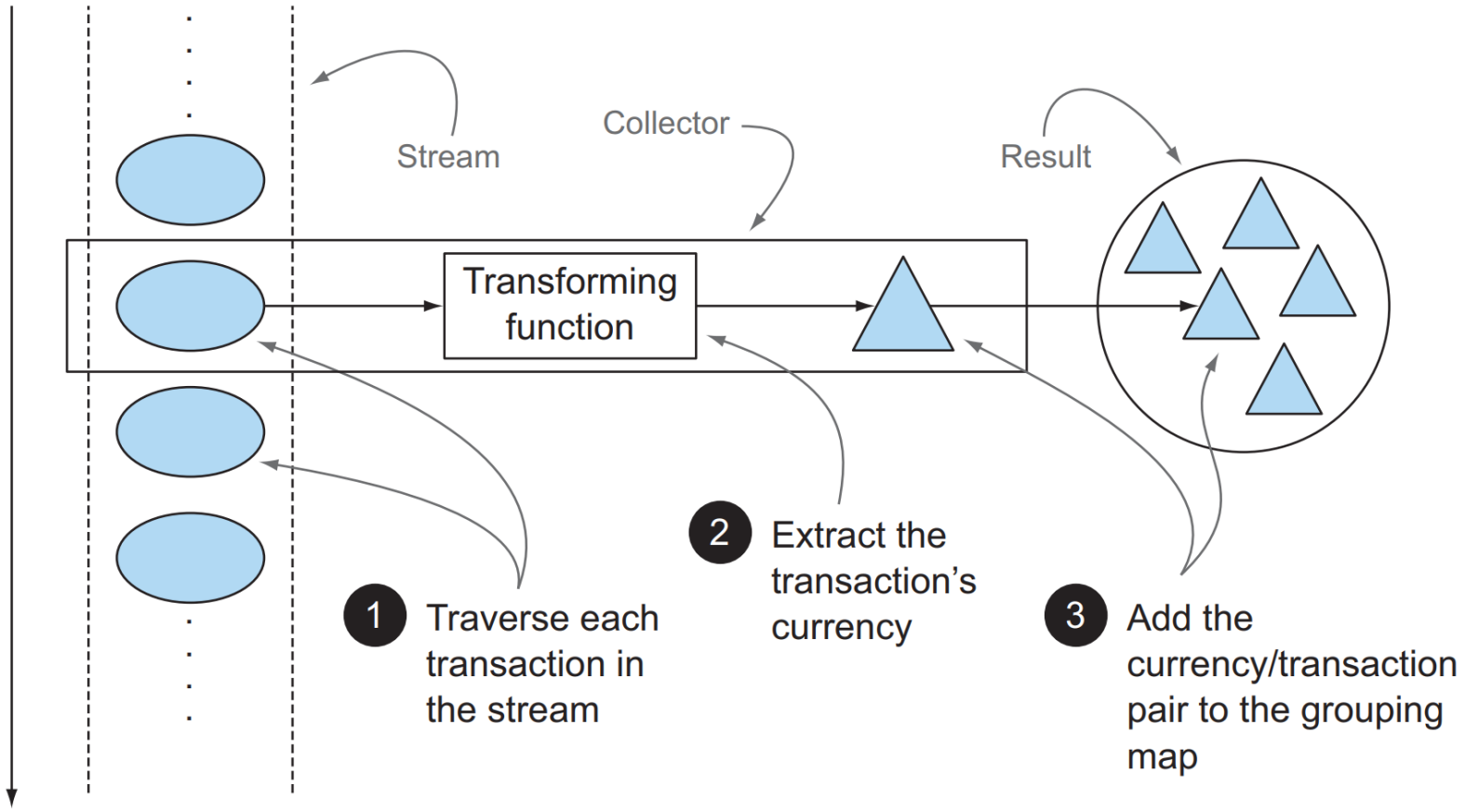


Figure 6.1 The reduction process grouping the transactions by currency

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

- The predefined collectors can be created form the factory methods (such as `groupingBy`) provided by the `Collectors` class.

6.2 Reducing and summarizing

- Count the number of dishes in the menu, using the collector returned by the `counting` factory method:

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

- You can write this far more directly as

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

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

Finding maximum and minimum in a stream of values

- `Collectors.maxBy` and `Collectors.minBy`
  - These two collectors take a `Comparator` as argument to compare the elements in the stream.

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


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

Summarization

- `Collectors.summingInt`
  - It accepts a function that maps an object into the `int`.
  - Find the total number of calories in your menu list with

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

- `Collectors.summingLong` and `Collectors.summingDouble`
- `Collectors.averagingInt`, `averagingLong` and `averagingDouble`

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

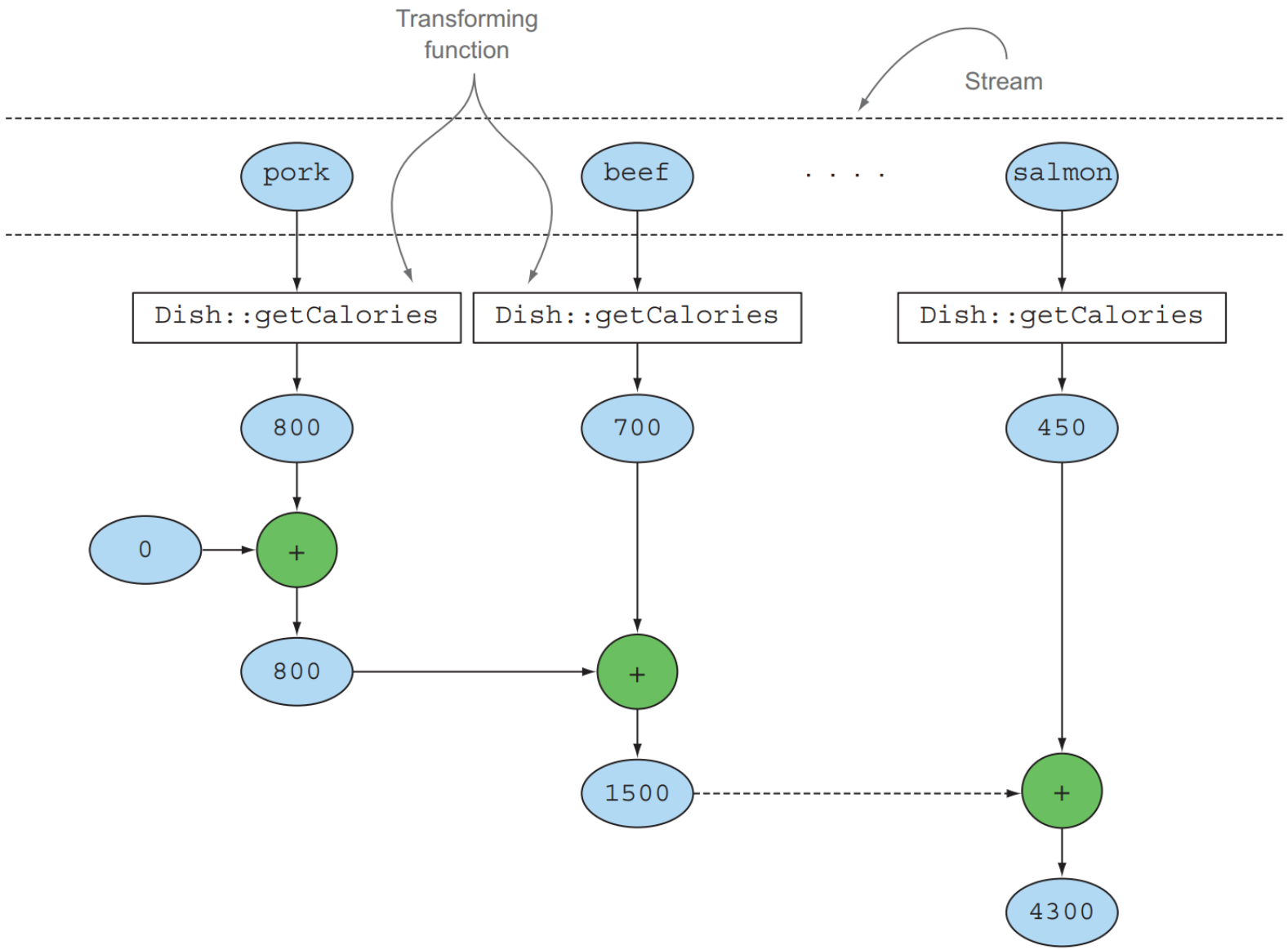


Figure 6.2 The aggregation process of the `summingInt` collector

- summarizingInt

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

- Printing the menuStatistics object:

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

- summarizingLong and summarizingDouble
  - LongSummaryStatistics
  - DoubleSummaryStatistics

Joining strings

- Concatenate the names of all the dishes in the menu as follows:

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

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

Both produce the string

```
porkbeefchickenfrench friesriceseason fruitpizzap
```

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

will generate

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

Generalized summarization with reduction

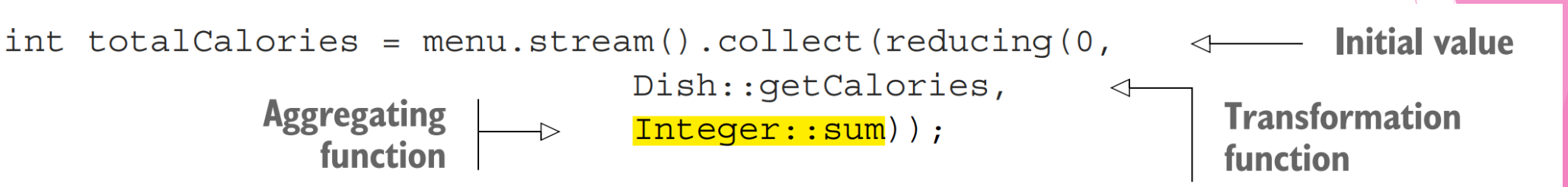
- The `Collectors.reducing` factory method is a generalization of all the collectors we've discussed so far.
- Calculate the total calories in your menu with a collector created from the `reducing` method as follows:

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

  - The first argument is the starting value of the reduction operation.
  - The second argument is used to transform a dish into an int.
  - The third argument is a `BinaryOperator` that aggregates two arguments into a single value of the same type.
-

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

Collection Framework Flexibility: Doing the same operation in different way



- The `counting` collector

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

- `int totalCalories =` **`Optional<Integer>`**
  `menu.stream().map(Dish::getCalories).reduce(Integer::sum).get();`
- `int totalCalories =`
  `menu.stream().mapToInt(Dish::getCalories).sum();`

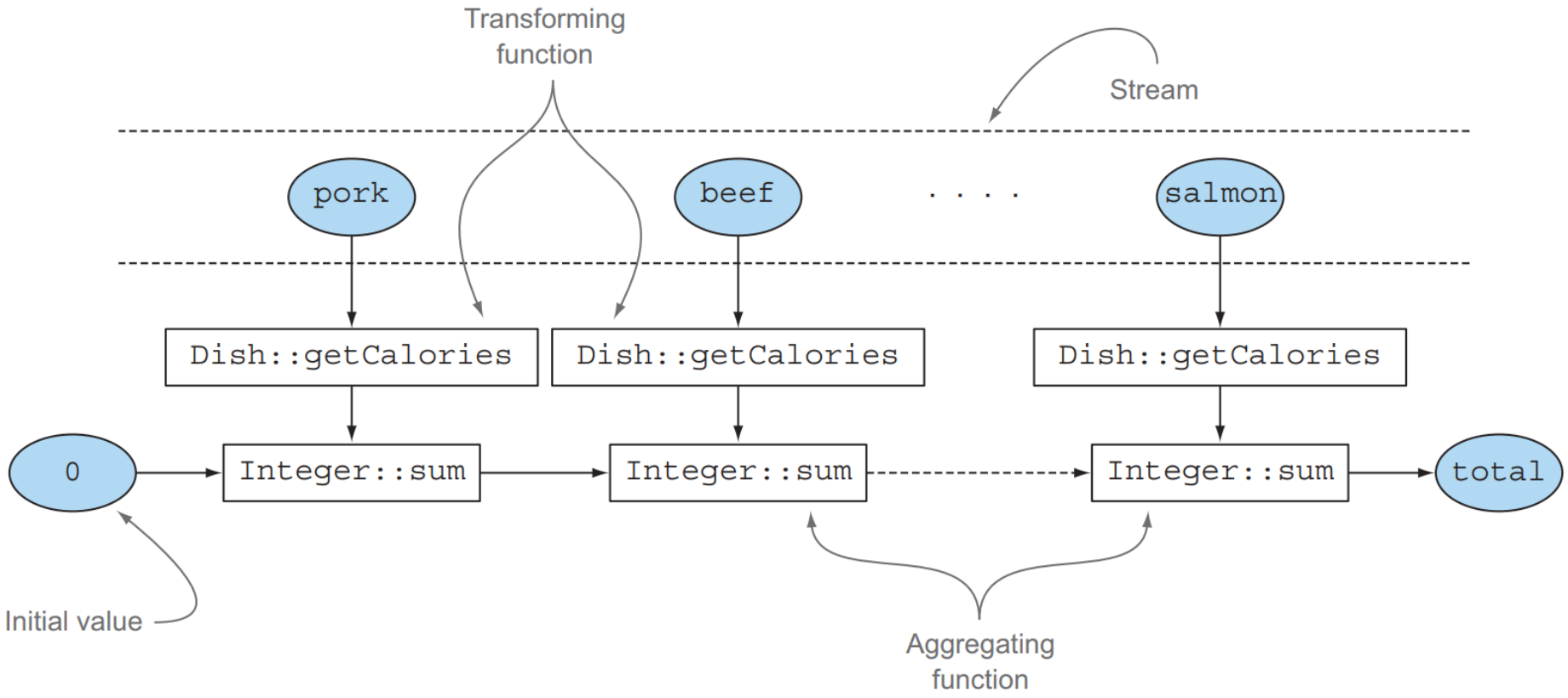


Figure 6.3 The reduction process calculating the total number of calories in the menu

6.3 Grouping

- `Collectors.groupingBy` factory method:

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

This will result in the following `Map`:

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

- `Dish::getType`
  - We call this `Function` a *classification* function specifically because it's used to classify the elements of the stream into different groups.

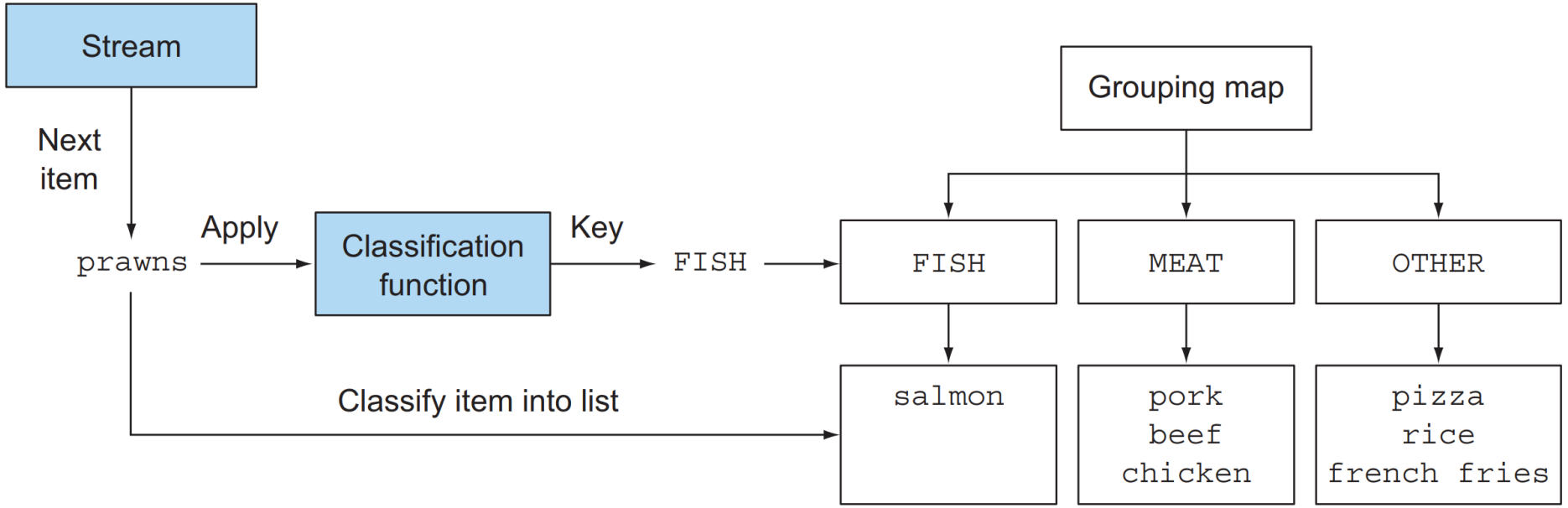


Figure 6.4 Classification of an item in the stream during the grouping process

```
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;
        } ));
```

Manipulating grouped elements

- Filter only the caloric dishes with more than 500 calories.

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

This solution works but has a possibly relevant drawback. If you try to use it on the dishes in our menu, you will obtain a `Map` like the following:

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

- `filtering` factory method

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

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

- `mapping` factory method

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

- `filtering`

`flatMapping` factory method

```
Map<String, List<String>> dishTags = new HashMap<>();
dishTags.put("pork", asList("greasy", "salty"));
dishTags.put("beef", asList("salty", "roasted"));
dishTags.put("chicken", asList("fried", "crisp"));
dishTags.put("french fries", asList("greasy", "fried"));
dishTags.put("rice", asList("light", "natural"));
dishTags.put("season fruit", asList("fresh", "natural"));
dishTags.put("pizza", asList("tasty", "salty"));
dishTags.put("prawns", asList("tasty", "roasted"));
dishTags.put("salmon", asList("delicious", "fresh"));
```

```
Map<Dish.Type, Set<String>> dishNamesByType =
    menu.stream()
        .collect(groupingBy(Dish::getType,
                flatMapping(dish -> dishTags.get( dish.getName() ).stream(),
                        toSet())));
```

```
{MEAT=[salty, greasy, roasted, fried, crisp], FISH=[roasted, tasty, fresh,
     delicious], OTHER=[salty, greasy, natural, light, tasty, fresh, fried]}
```

Multilevel grouping

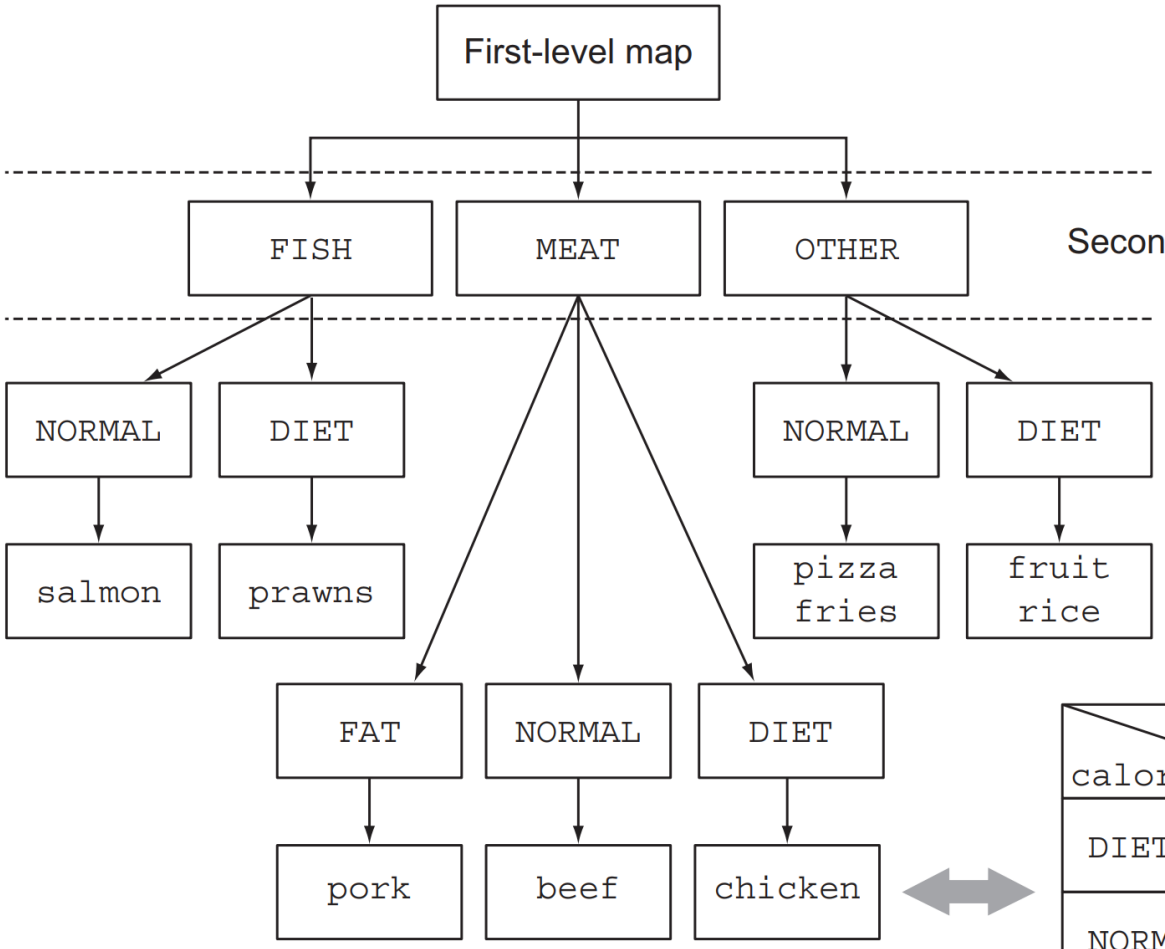
- A two level grouping

Listing 6.2 Multilevel grouping

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

The result of this two-level grouping is a two-level `Map` like the following:

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



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

Figure 6.5 Equivalence between *n*-level nested map and *n*-dimensional classification table

Collecting data in subgroups

- counting method

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

The result is the following Map:

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

- goupingBy(f)== groupingBy(f, toList())

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

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

Adapting the Collector to a Different Type

- `Collectors.collectingAndThen` method

Listing 6.3 Finding the highest-calorie dish in each subgroup

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

Classification function

Wrapped collector

Transformation function

```
{FISH=salmon, OTHER=pizza, MEAT=pork}
```

- Wrapped collector
  - `maxby`
- Transformation function
  - `Optional::get`

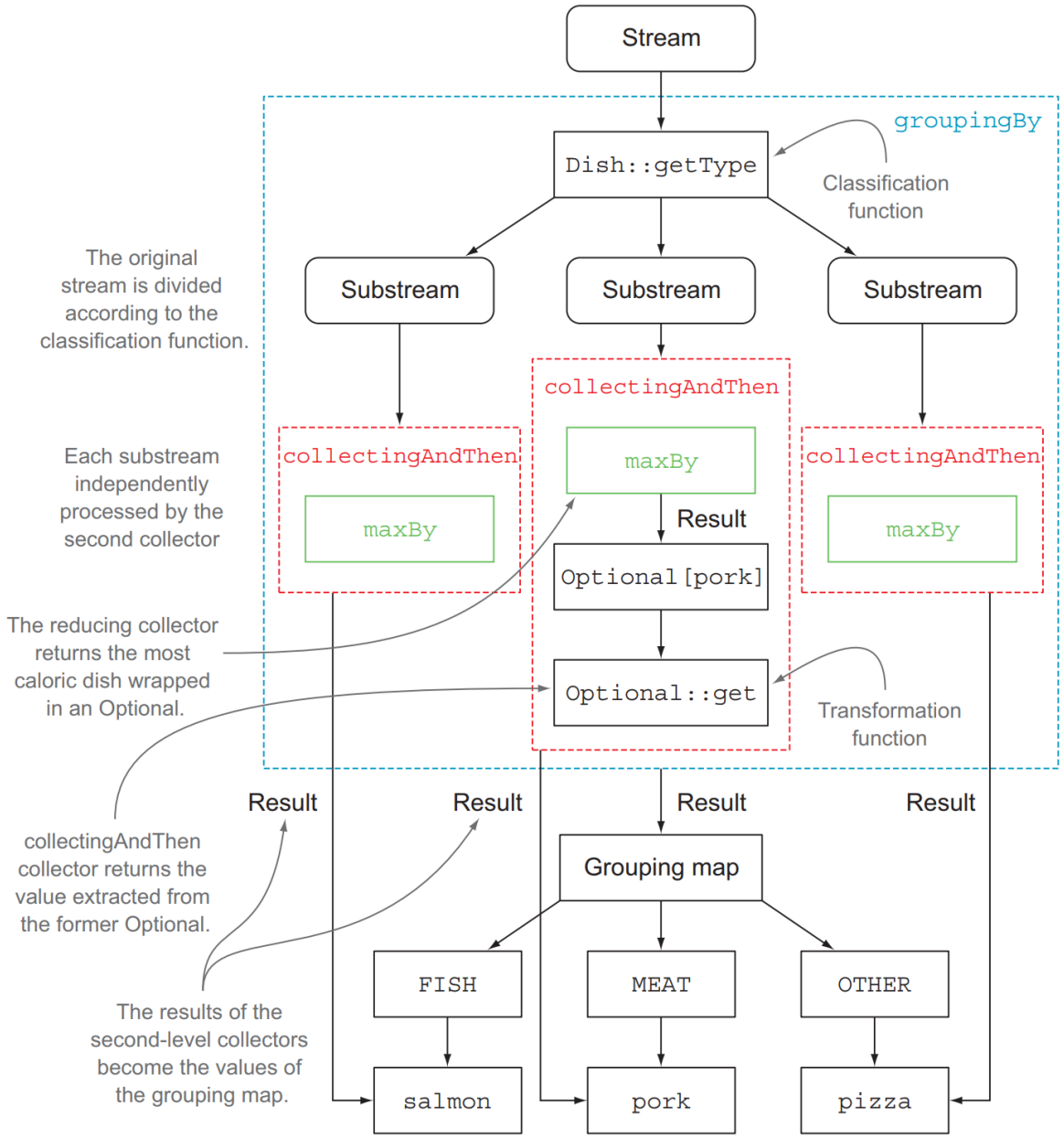


Figure 6.6 Combining the effect of multiple collectors by nesting one inside the other

Other Examples of Collectors used in conjunction with groupingBy

- `summingInt` method

```
Map<Dish.Type, Integer> totalCaloriesByType =
                menu.stream().collect(groupingBy(Dish::getType,
                        summingInt(Dish::getCalories)));
```

- `mapping` method

```
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() )));
```

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

`toCollection` method

- you can ask for a `HashSet` by passing a constructor reference to it:

```
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; },
    toCollection(HashSet::new) )));
```

6.4 Partitoning

- Partitioning is a special case of grouping: having a predicate called a *partitioning function* as a classification function.

Partitioning function

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

This will return the following Map:

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

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

```
List<Dish> vegetarianDishes =
                menu.stream().filter(Dish::isVegetarian).collect(toList());
```
```

- Find the most caloric dish among both vegetarian and nonvegetarian dishes:

```
Map<Boolean, Dish> mostCaloricPartitionedByVegetarian =
menu.stream().collect(
    partitioningBy(Dish::isVegetarian,
        collectingAndThen(maxBy(comparingInt(Dish::getCalories)),
                          Optional::get)));
```

That will produce the following result:

```
{false=pork, true=pizza}
```

Advantages of partitioning

- Partitioning has the advantage of keeping both lists of the stream elements, for which the application of the partitioning function returns `true` or `false`.

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

This will produce a two-level `Map`:

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

Partitioning numbers into prime and noprime

- A predicate that tests to see if a given candidate number is prime or not:

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

Generates a range of natural numbers starting from and including 2, up to but excluding candidate

Returns true if the candidate isn't divisible for any of the numbers in the stream

A simple optimization is to test only for factors less than or equal to the square root of the candidate:

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

- To partition the first *n* numbers into prime and nonprime:

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

Table 6.1 The main static factory methods of the `Collectors` class

Factory method	Returned type	Used to
`toList`	`List<T>`	Gather all the stream's items in a `List`.
Example use: `List<Dish> dishes = menuStream.collect(toList());`		
`toSet`	`Set<T>`	Gather all the stream's items in a `Set`, eliminating duplicates.
Example use: `Set<Dish> dishes = menuStream.collect(toSet());`		
`toCollection`	`Collection<T>`	Gather all the stream's items in the collection created by the provided supplier.
Example use: `Collection<Dish> dishes = menuStream.collect(toCollection(), ArrayList::new);`		

Factory method	Returned type	Used to
`counting`	`Long`	Count the number of items in the stream.
Example use: `longhowManyDishes = menuStream.collect(counting());`		
`summingInt`	`Integer`	Sum the values of an Integer property of the items in the stream.
Example use: `int totalCalories = menuStream.collect(summingInt(Dish::getCalories));`		
`averagingInt`	`Double`	Calculate the average value of an `Integer` property of the items in the stream.
Example use: `double avgCalories = menuStream.collect(averagingInt(Dish::getCalories));`		
`summarizingInt`	`IntSummaryStatistics`	Collect statistics regarding an `Integer` property of the items in the stream, such as the maximum, minimum, total, and average.
Example use: `IntSummaryStatistics menuStatistics = menuStream.collect(summarizingInt(Dish::getCalories));`		

Table 6.1 The main static factory methods of the `Collectors` class *(continued)*

Factory method	Returned type	Used to
`joining`	`String`	Concatenate the strings resulting from the invocation of the `toString` method on each item of the stream.
Example use: `String shortMenu = menuStream.map(Dish::getName).collect(joining(", "));`		
`maxBy`	`Optional<T>`	An `Optional` wrapping the maximal element in this stream according to the given comparator or `Optional.empty()` if the stream is empty.
Example use: `Optional<Dish> fattest = menuStream.collect(maxBy(comparingInt(Dish::getCalories)));`		
`minBy`	`Optional<T>`	An `Optional` wrapping the minimal element in this stream according to the given comparator or `Optional.empty()` if the stream is empty.
Example use: `Optional<Dish> lightest = menuStream.collect(minBy(comparingInt(Dish::getCalories)));`		

Factory method	Returned type	Used to
`reducing`	The type produced by the reduction operation	Reduce the stream to a single value starting from an initial value used as accumulator and iteratively combining it with each item of the stream using a `BinaryOperator`.
Example use: `int totalCalories = menuStream.collect(reducing(0, Dish::getCalories, Integer::sum));`		
`collectingAndThen`	The type returned by the transforming function	Wrap another collector and apply a transformation function to its result.
Example use: `int howManyDishes = menuStream.collect(collectingAndThen(toList(), List::size));`		
`groupingBy`	`Map<K, List<T>>`	Group the items in the stream based on the value of one of their properties and use those values as keys in the resulting `Map`.
Example use: `Map<Dish.Type,List<Dish>> dishesByType = menuStream.collect(groupingBy(Dish::getType));`		
`partitioningBy`	`Map<Boolean, List<T>>`	Partition the items in the stream based on the result of the application of a predicate to each of them.
Example use: `Map<Boolean,List<Dish>> vegetarianDishes = menuStream.collect(partitioningBy(Dish::isVegetarian));`		

6.5 The Collector interface

- The `Collector` interface consists of a set of methods that provide a blueprint for how to implement specific reduction operations (collectors).
- `ToListCollector<T>` class
  - gathers all theelements of a Stream<T> into a List<T>

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

Listing 6.4 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();
}
```

- In this listing, the following definitions apply:
  - `T` is the generic type of the items in the stream to be collected.
  - `A` is the type of the accumulator, the object on which the partial result will be accumulated during the collection process.
  - `R` is the type of the object (typically, but not always, the collection) resulting from the collect operation.

Making sense of the methods declared by Collector interface

- Each of the first four methods returns a function that will be invoked by the `collect` method.
- The fifth one, `characteristics`, provides a set of characteristics that's a list of hints used by the `collect` method itself to know which optimizations (for example, parallelization) it's allowed to employ while performing the reduction operation.

Making a new result container: the Supplier method

- The `supplier` method has to return a `Supplier` of an empty accumulator.
- In our `ToListCollector` the `supplier` will then return an empty `List`, as follows:

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

Note that you could also pass a constructor reference:

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

Adding an element to a result container: the Accumulator method

- The `accumulator` method returns the function that performs the reduction operation.
- For `ToListCollector`, this function merely has to add the current item to the list containing the already traversed ones:

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

You could instead use a method reference, which is more concise:

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

Applying the final transformation to the result container: the Finisher method

- The `finisher` method has to return a function that's invoked at the end of the accumulation process, after having completely traversed the stream, in order to transform the accumulator object into the final result of the whole collection operation.
- There's no need to perform a transformation, so the finisher method has to return the identity function:

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

Merging two result containers: the Combiner method

- The `combiner` method, the last of the four methods that return a function used by the reduction operation, defines how the accumulators resulting from the reduction of different subparts of the stream are combined when the subparts are processed in parallel.
- The addition of this fourth method allows a parallel reduction of the stream.
- In the `toList` case:

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

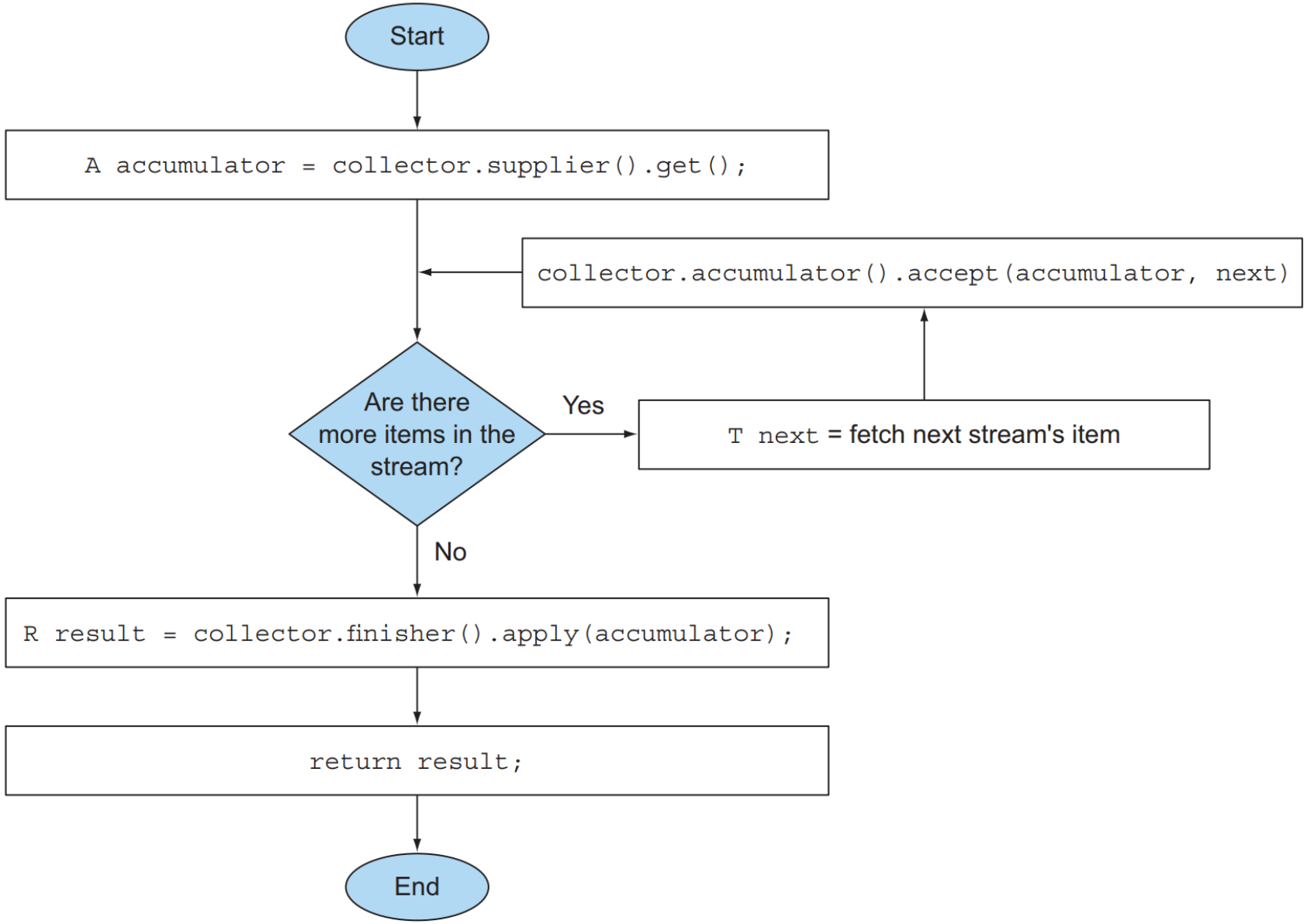


Figure 6.7 Logical steps of the sequential reduction process

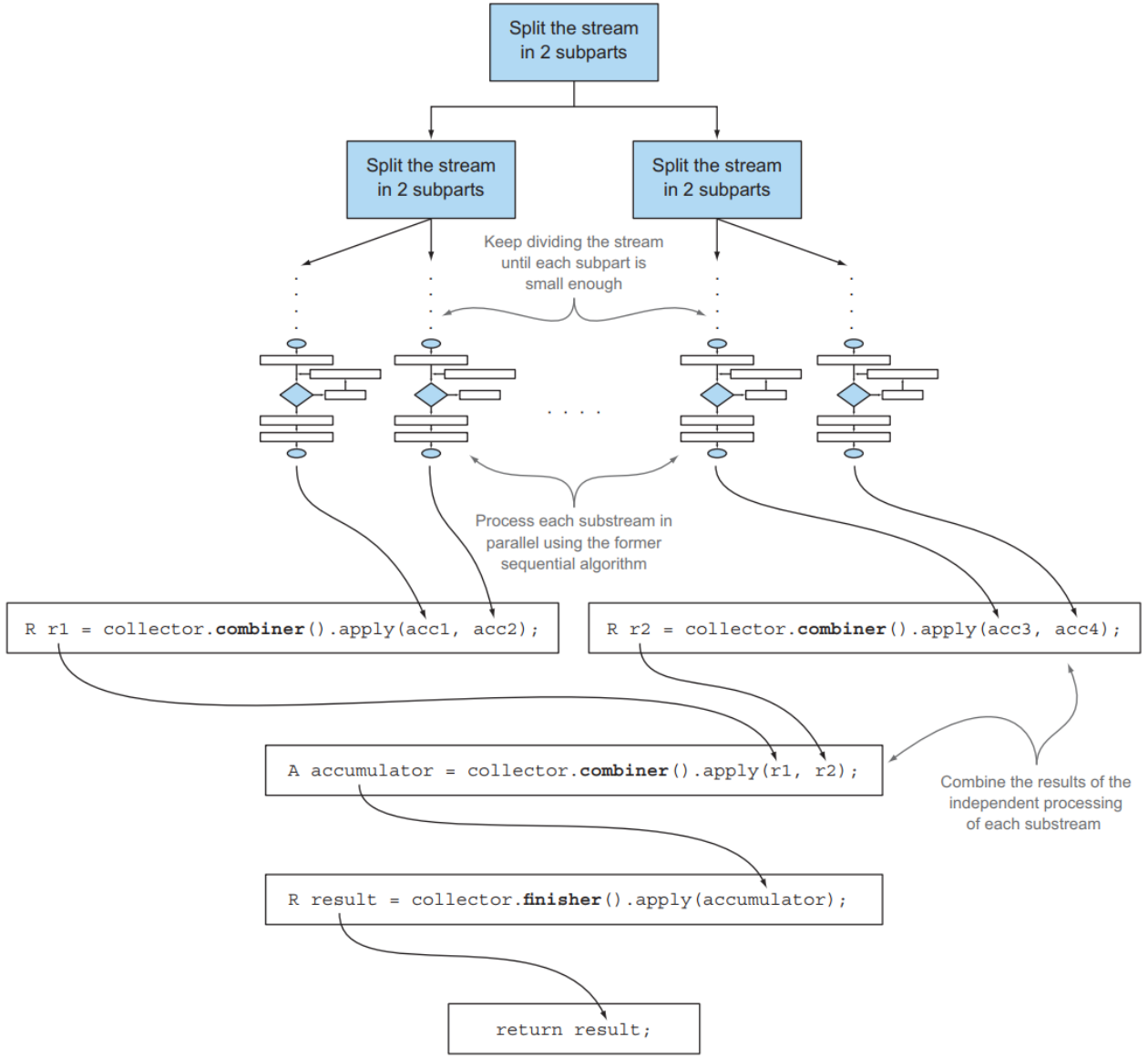


Figure 6.8 Parallelizing the reduction process using the `combiner` method

The Characteristics method

- The last method, `characteristics`, returns an immutable set of `Characteristics`, defining the behavior of the collector.
- `Characteristics` is an enumeration containing three items:
  - UNORDERED
  - CONCURRENT
  - IDENTITY_FINISH

Putting them all together

Listing 6.5 The `ToListCollector`

```
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));
    }
}
```

Creates the collection operation starting point

Accumulates the traversed item, modifying the accumulator in place

Identifies function

Modifies the first accumulator, combining it with the content of the second one

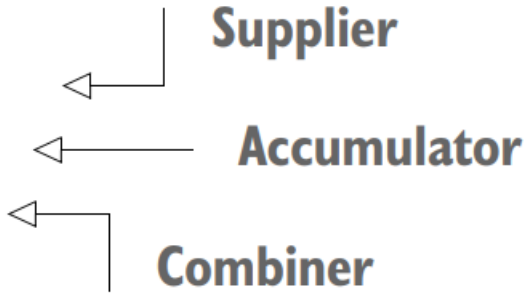
Returns the modified first accumulator

Flags the collector as IDENTITY_FINISH and CONCURRENT

Performing a custom collector without creating a Collector implementation

- Streams has an overloaded `collect` method accepting the three other functions—`supplier`, `accumulator`, and `combiner`.
- It's possible to collect in a `List` all the items in a stream of dishes, as follows:

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



6.6 Developing your own collector for better performance

Listing 6.6 Partitioning the first *n* natural numbers into primes and nonprimes

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

- An Improvement:

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

Divide only by prime numbers

- One possible optimization is to test only if the candidate number is divisible by prime numbers.
- You can limit the test to only the prime numbers found before the current candidate.
- `isPrime` method

```
public static boolean isPrime(List<Integer> primes, int candidate) {
    return primes.stream().noneMatch(i -> candidate % i == 0);
}
public static boolean isPrime(List<Integer> primes, int candidate){
    int candidateRoot = (int) Math.sqrt((double) candidate);
    return primes.stream()
                 .takeWhile(i -> i <= candidateRoot)
                 .noneMatch(i -> candidate % i == 0);
}
```

- First, you need to declare a new class that implements the `Collector` interface.
- Then, you need to develop the five methods required by the `Collector` interface.
- Step 1: Defining the Collector Class Signature

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

```
public class PrimeNumbersCollector
             implements Collector<Integer,
                                  Map<Boolean, List<Integer>>,
                                  Map<Boolean, List<Integer>>>
```

The type of the elements in the stream

The type of the result of the collect operation

The type of the accumulator

Step 2: Implement the Reduction Process

- The `supplier` method has to return a function that when invoked creates the accumulator:

```
public Supplier<Map<Boolean, List<Integer>>> supplier() {
    return () -> new HashMap<Boolean, List<Integer>>() {{
        put(true, new ArrayList<Integer>());
        put(false, new ArrayList<Integer>());
    }};
}
```

- The `accumulator` contains the prime numbers found so far:

```
public BiConsumer<Map<Boolean, List<Integer>>, Integer> accumulator() {
    return (Map<Boolean, List<Integer>> acc, Integer candidate) -> {
            acc.get( isPrime(acc.get(true), candidate) )
               .add(candidate);
    };
}
```

Gets the list of prime or nonprime numbers depending on the result of isPrime

Adds the candidate to the appropriate list

Step 3: Making the Collector Work in Parallel

```
public BinaryOperator<Map<Boolean, List<Integer>>> combiner() {
    return (Map<Boolean, List<Integer>> map1,
            Map<Boolean, List<Integer>> map2) -> {
                map1.get(true).addAll(map2.get(true));
                map1.get(false).addAll(map2.get(false));
                return map1;
    };
}
```

- Note that in reality this collector can't be used in parallel, because the algorithm is inherently sequential.

Step 4: The Finisher Method and the Collector's Characteristic Method

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

As for the characteristic method, we already said that it's neither `CONCURRENT` nor `UNORDERED` but is `IDENTITY_FINISH`:

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

Listing 6.7 The `PrimeNumbersCollector`

```
public class PrimeNumbersCollector
    implements Collector<Integer,
            Map<Boolean, List<Integer>>,
            Map<Boolean, List<Integer>>> {
    @Override
    public Supplier<Map<Boolean, List<Integer>>> supplier() {
        return () -> new HashMap<Boolean, List<Integer>>() {{
              put(true, new ArrayList<Integer>());
              put(false, new ArrayList<Integer>());
            }};
    }
    @Override
    public BiConsumer<Map<Boolean, List<Integer>>, Integer> accumulator() {
        return (Map<Boolean, List<Integer>> acc, Integer candidate) -> {
            acc.get( isPrime( acc.get(true),
                 candidate) )
                 .add(candidate);
        };
    }
```

Starts the collection process with a Map containing two empty Lists

Passes to the isPrime method the list of already found primes

Gets from the Map the list of prime or nonprime numbers, according to what the isPrime method returned, and adds to it the current candidate

```
        @Override
        public BinaryOperator<Map<Boolean, List<Integer>>> combiner() {
                return (Map<Boolean, List<Integer>> map1,
                        Map<Boolean, List<Integer>> map2) -> {
                                map1.get(true).addAll(map2.get(true));
                                map1.get(false).addAll(map2.get(false));
                                return map1;
                        };
        }
        @Override
        public Function<Map<Boolean, List<Integer>>,
                        Map<Boolean, List<Integer>>> finisher() {
                                return Function.identity();

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

Merges the second Map into the first one

No transformation necessary at the end of the collection process, so terminate it with the identity function

This collector is IDENTITY_FINISH but neither UNORDERED nor CONCURRENT because it relies on the fact that prime numbers are discovered in sequence.

You can now use this new custom collector in place of the former one created with the `partitioningBy` factory method in section 6.4 and obtain exactly the same result:

```
public Map<Boolean, List<Integer>>
                        partitionPrimesWithCustomCollector(int n) {
        return IntStream.rangeClosed(2, n).boxed()
                        .collect(new PrimeNumbersCollector());
}
```

Comparing Collectors' Performance

- Java Microbenchmark Harness (JMH)

```
public class CollectorHarness {
    public static void main(String[] args) {
        long fastest = Long.MAX_VALUE;
        for (int i = 0; i < 10; i++) {
            long start = System.nanoTime();
            partitionPrimes(1_000_000);
            long duration = (System.nanoTime() - start) / 1_000_000;
            if (duration < fastest) fastest = duration;
        }
        System.out.println(
            "Fastest execution done in " + fastest + " msecs");
    }
}
```

Runs the test 10 times

Partitions the first million natural numbers into primes and nonprimes

The duration in milliseconds

Checks if this execution is the fastest one

- Running it on an Intel i5 2.4 GHz, it prints the following result:

```
Fastest execution done in 4716 msecs
Fastest execution done in 3201 msecs
```

```
public Map<Boolean, List<Integer>> partitionPrimesWithCustomCollector
             (int n) {
    IntStream.rangeClosed(2, n).boxed()
        .collect(
                    () -> new HashMap<Boolean, List<Integer>>() {{
                       put(true, new ArrayList<Integer>());
                       put(false, new ArrayList<Integer>());
                    }},
             (acc, candidate) -> {
                acc.get( isPrime(acc.get(true), candidate) )
                   .add(candidate);
            },
              (map1, map2) -> {
                map1.get(true).addAll(map2.get(true));
                map1.get(false).addAll(map2.get(false));
            });
}
```

Supplier

Accumulator

Combiner