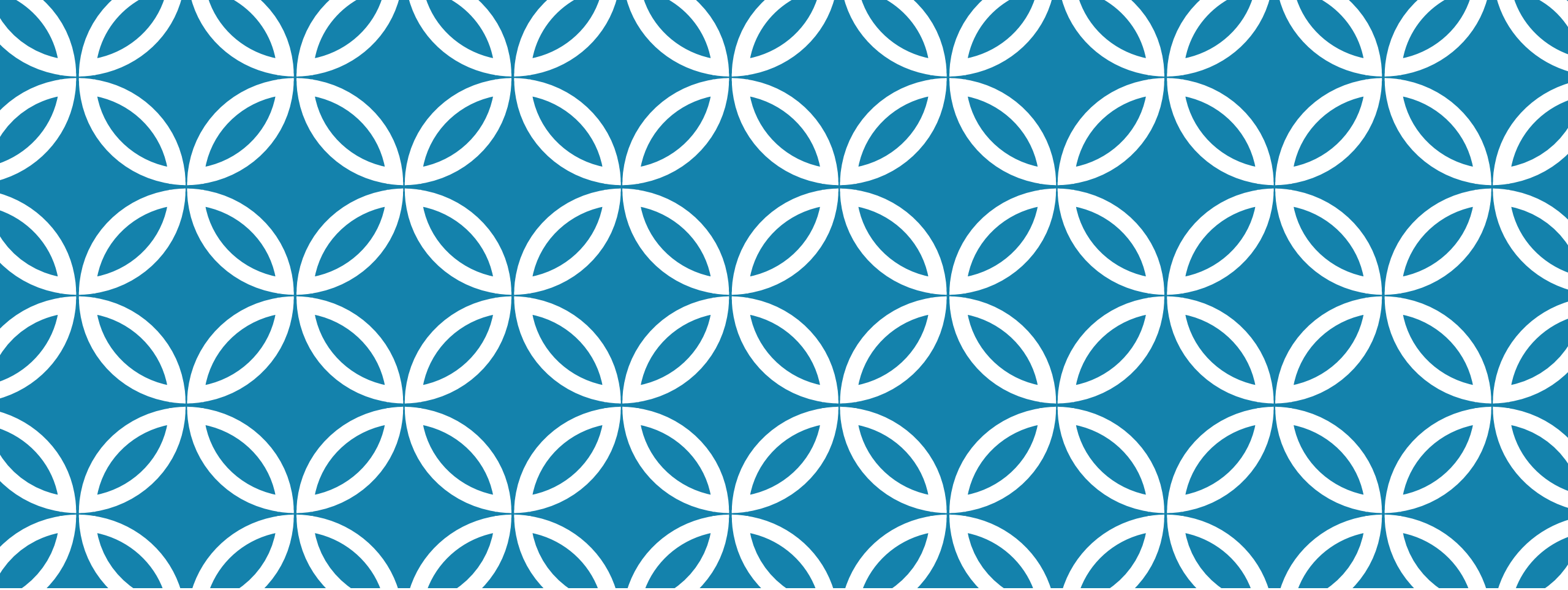


STREAM CREATION AND COLLECTING STREAMS

Chandreyee Chowdhury

NUMERIC STREAMS

Creating numeric streams

- `IntStream oneToHundred =IntStream.rangeClosed(1,100).filter(i%2==0)`
- `IntStream oneToNinetyNine = IntStream.range(1,100).filter(i%2==0)`

BUILDING STREAMS

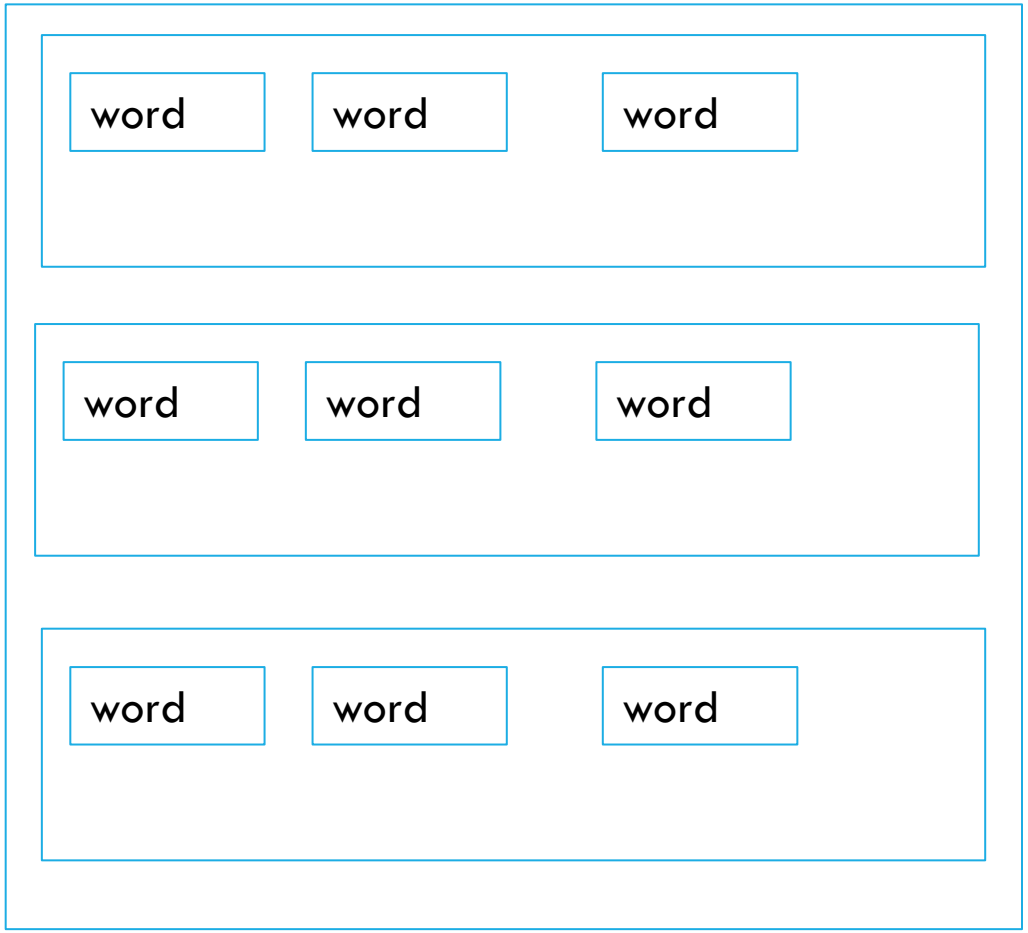
Static methods

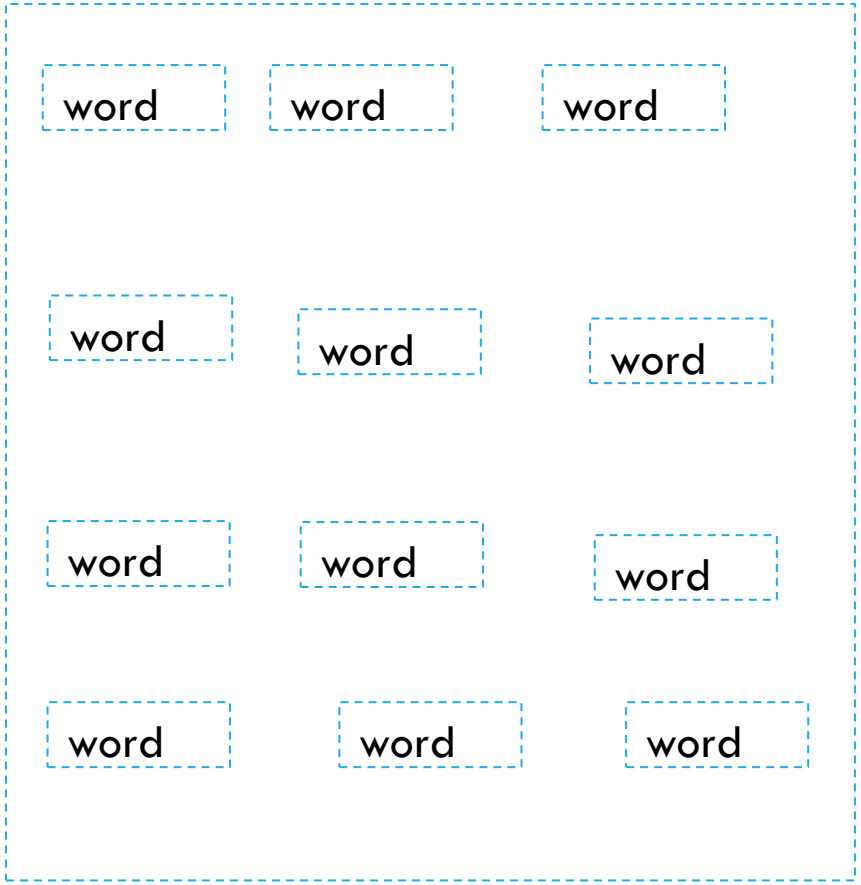
- Stream.of(“Kaushal”, “Bitanu”, “Titir”,”Subhayan”);
- Stream.empty()
- Arrays.stream(1,2,3,4)
- Str.chars()

- From files

STREAMS FROM FILES

```
long NoOfUniqueWords =0;
try{
    Stream<String> lines1=
            Files.lines(Paths.get("dataFile.txt"),
            Charset.defaultCharset());
```


word
word
word
word
word
word
word
word
word


word
word
word
word
word
word
word
word
word
word
word
word

```
NoOfUniqueWords =lines1.flatMap(lines2->
                        Arrays.stream(lines2.split(“ ")))
                       .distinct().count();
  System.out.println("1. Unique words are "
                          + NoOfUniqueWords);
            }catch(Exception e){}
```

INFINITE STREAMS

Iterate

- ***Stream.iterate(0, n -> n + 2)***.limit(10).forEach(System.out::println);
- Stream.of(1,2,3,4,5,6,7,8,9,10).?

Fibonacci number

Stream.iterate(new int[]{0, 1}, ???).limit(20)

.forEach(t -> System.out.println("(" + t[0] + "," + t[1] +")"));

INFINITE STREAMS

It takes a lambda of type Supplier<T> to provide new values

Stream.generate(Math::random)

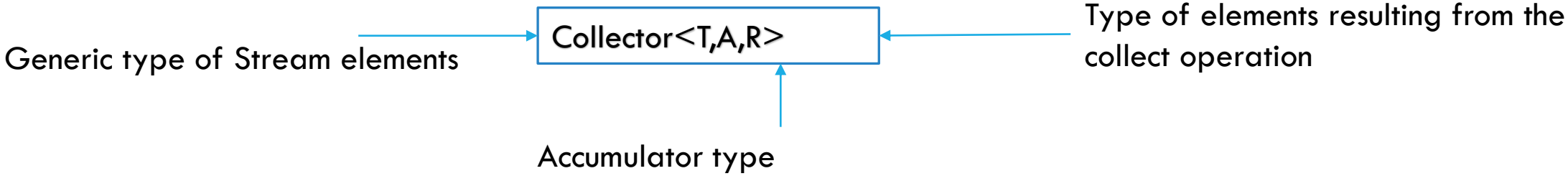
.limit(5)

.forEach(System.out::println);

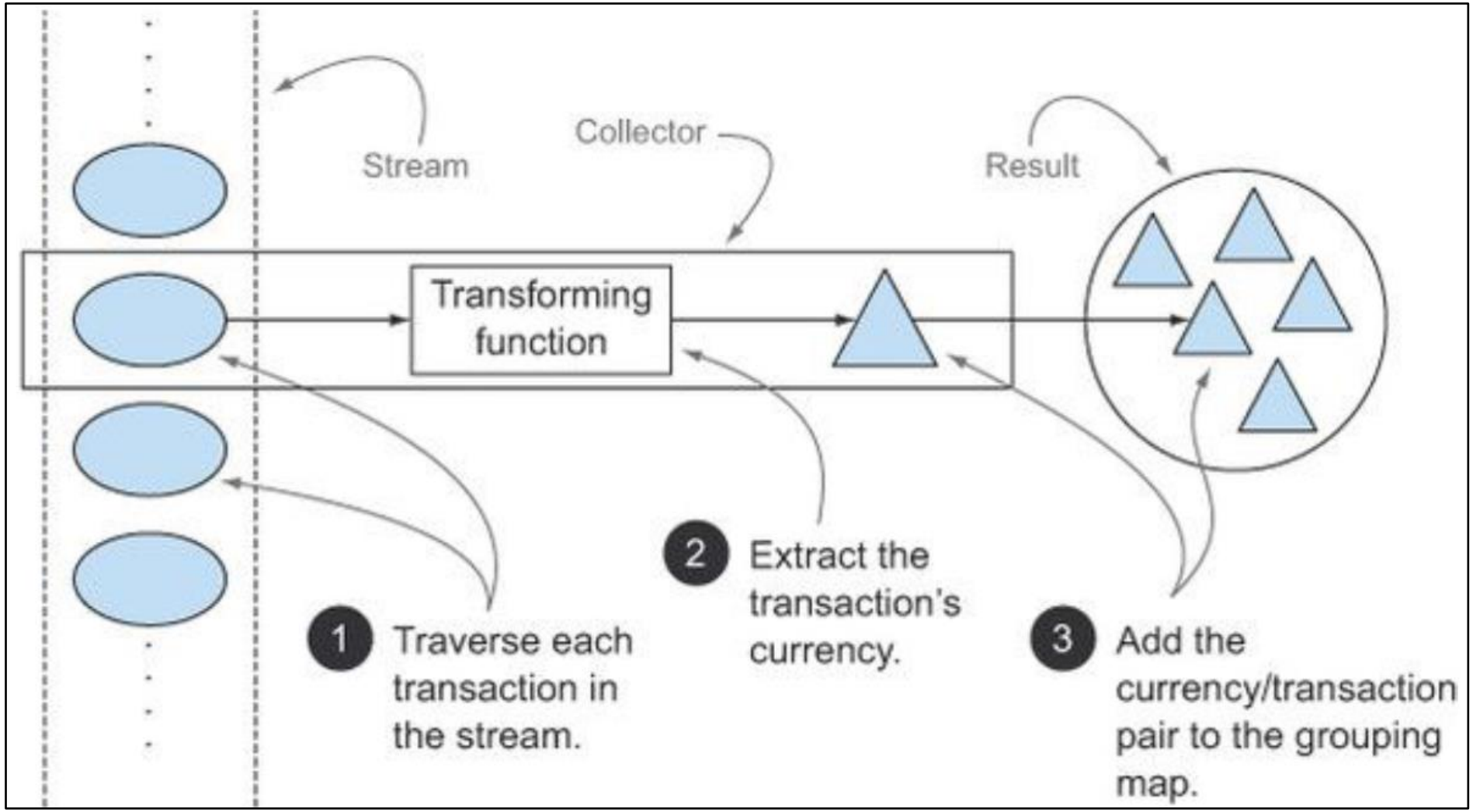
a supplier that’s stateful isn’t safe to use in parallel code

COLLECTING STREAMS

- Collect is a terminal operation that summarizes the stream while collecting the result
- collect(toList())
- Collection, collector and collect are different
- Collector interface
  - a general-purpose construct for producing complex values from streams.
  - Collector<T,A,R>



COLLECTING


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.

COLLECTORS

Collector applies a transforming function to the elements

For example, in toList() it is the identity transformation

Accumulates the results in a data structure

Predefined collectors can be created from the factory methods provided by the **Collectors** class

Collectors that are used are commonly statically imported from the java.util.stream.Collectors class

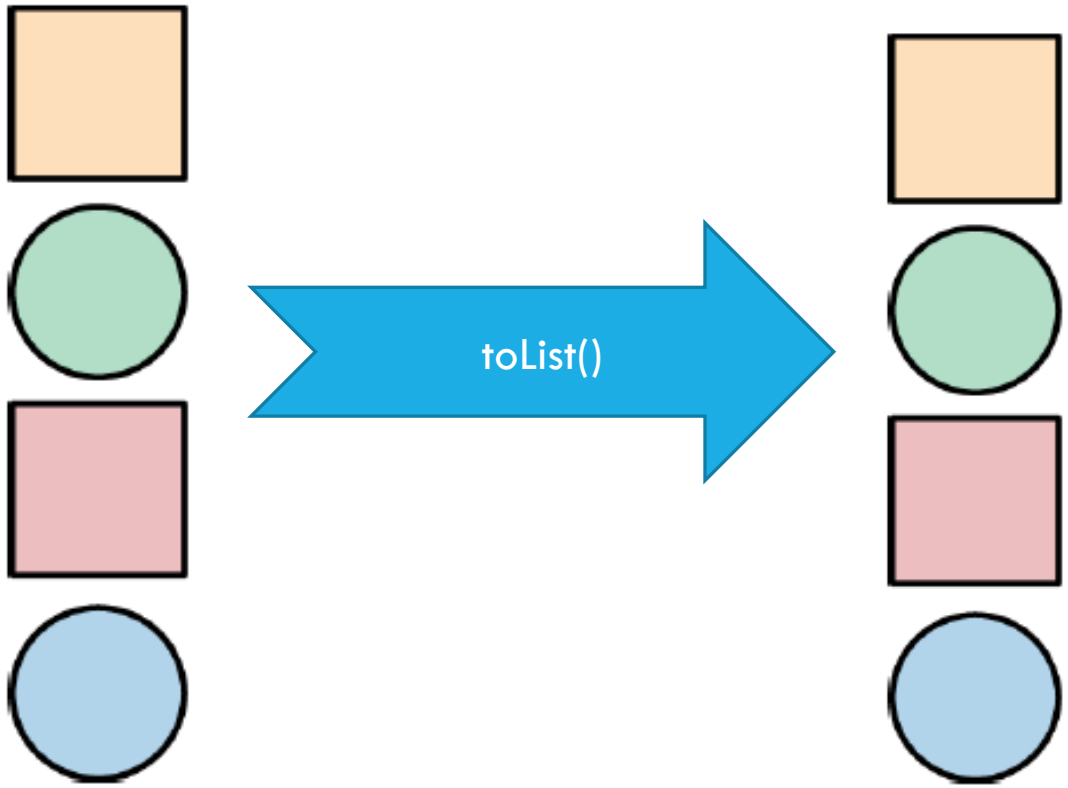
Normally when we create a collection, we specify the concrete type of the collection by calling the appropriate constructor

```
List<Artist> artists = new ArrayList<>();
```

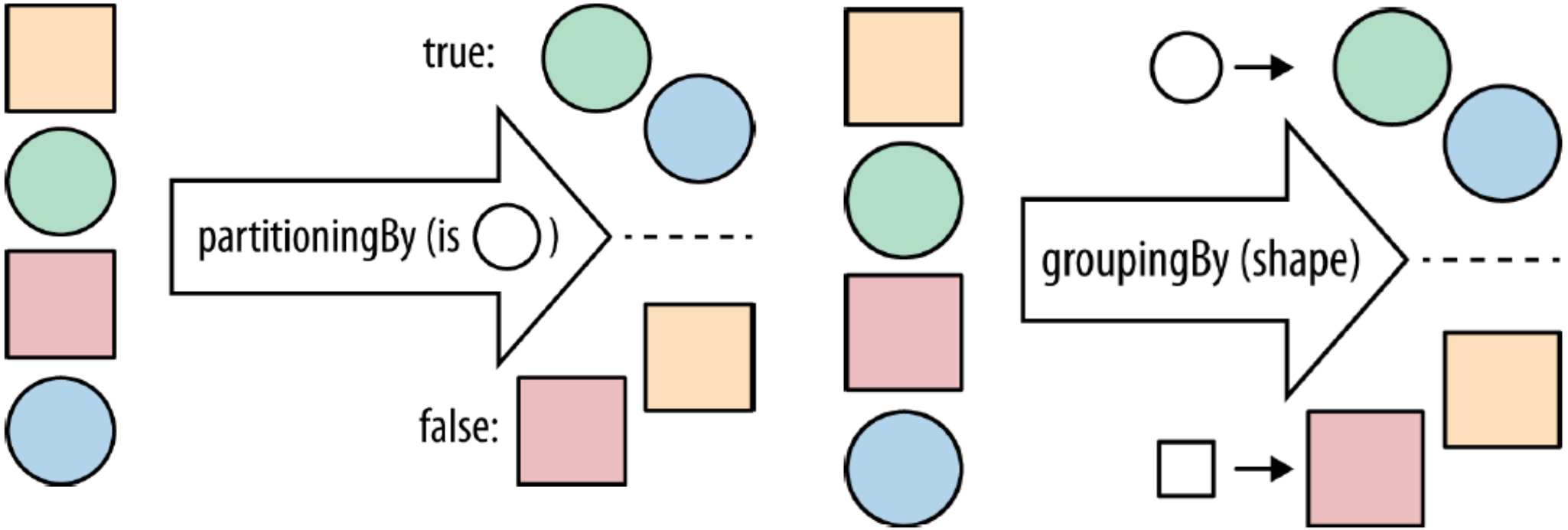
But when you're calling toList or toSet, you don't get to specify the concrete implementation of the List or Set

Under the hood, the streams library is picking an appropriate implementation for you

COLLECTING STREAM ELEMENTS



COLLECTING STREAM ELEMENTS



COLLECTING STREAMS

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

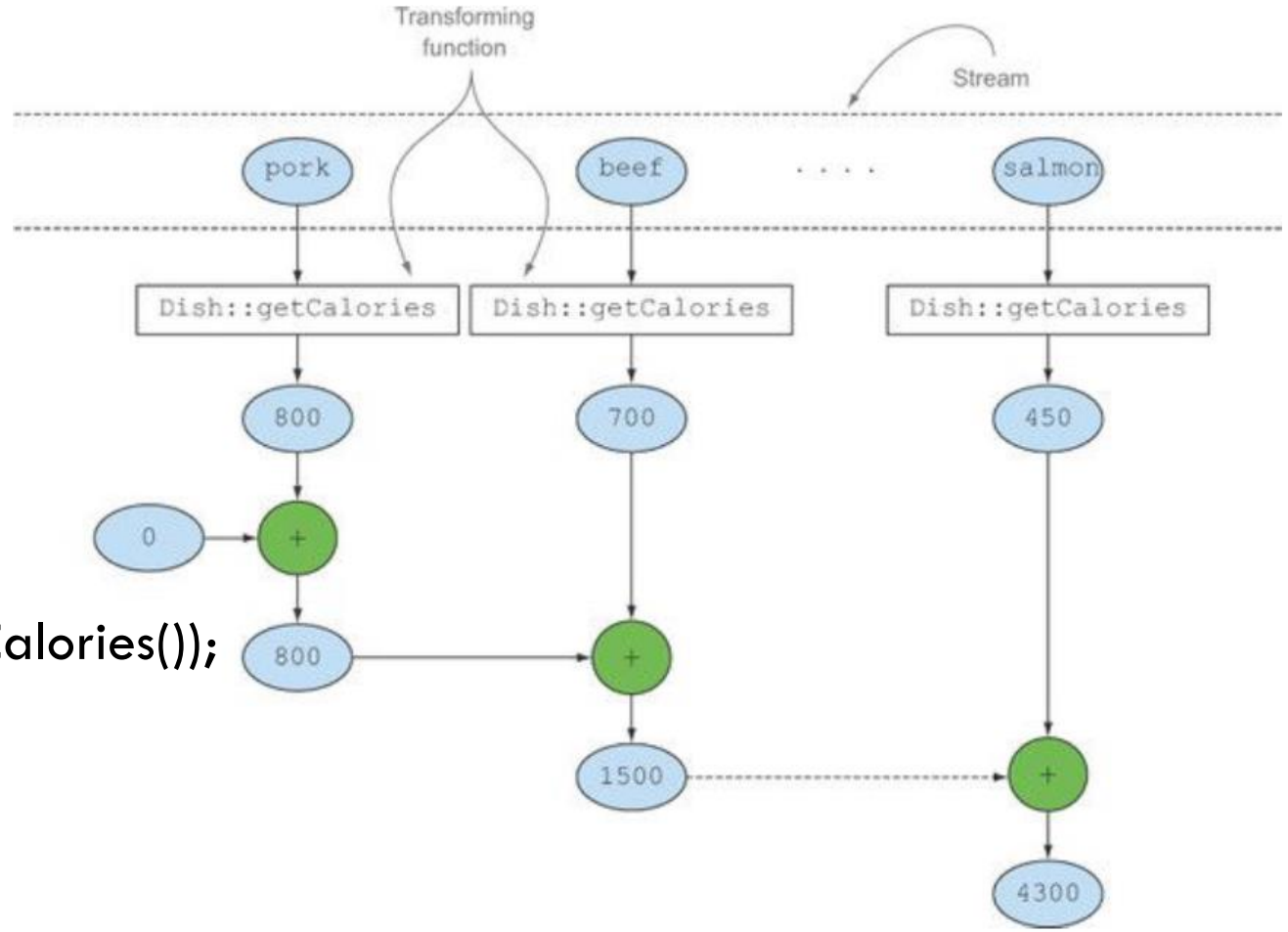
REDUCING AND SUMMARIZING

- Count the no of menu items
  - Collectors.counting()
- long countingDish=menu.stream().collect(Collectors.counting());
- maxBy() and minBy()

- Comparator<Dish> dishCaloriesComp=Comparator.comparing(x→x.getCalories());
- Optional<Dish> TastyDish=menu.stream().collect(maxBy(dishCaloriesComp));

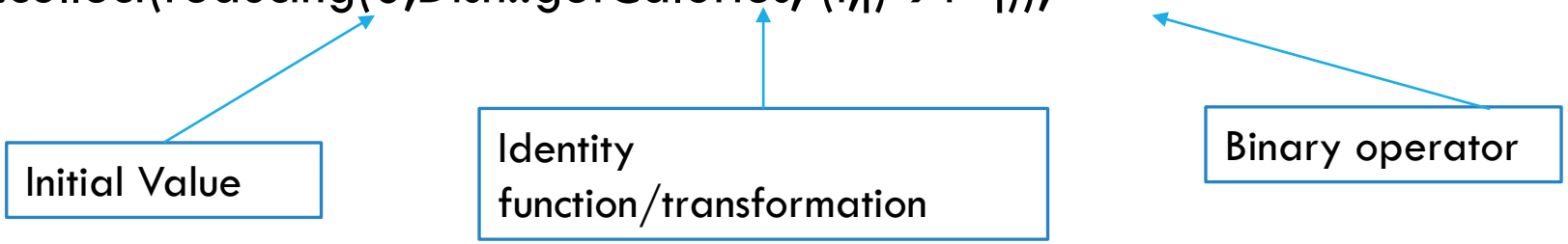
SUMMARIZING

- Collectors.summingInt()
- menu.stream().collect(summingInt(d→d.getCalories());
- averagingInt()
- summarizingInt()
- IntSummaryStatistics



JOINING STRINGS

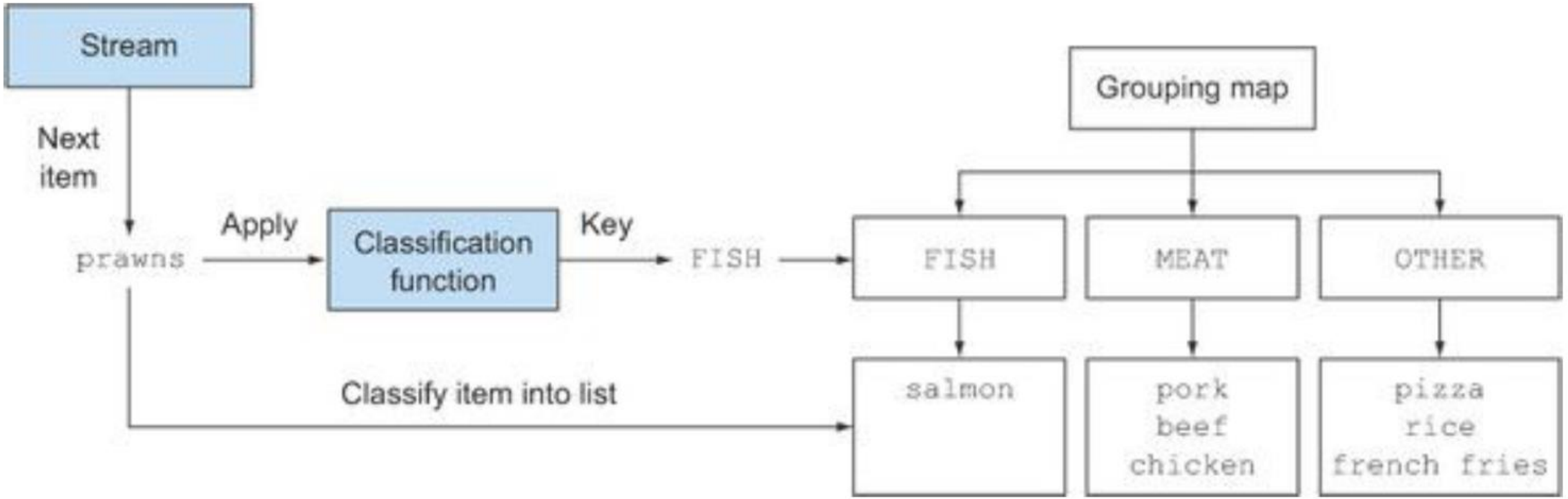
```
String results=menu.stream().filter(d→d.isVegetarian())
                            .map(d→d.getName())
                            .collect(Collectors.joining(“,”,”[“,”]”);
results=menu.stream().collect(reducing(0,Dish::getCalories, (i,j)→i+j));
```



When executed in parallel, multiple intermediate results may be instantiated, populated, and merged so as to maintain isolation of mutable data structures.

GROUPING

```
menu.stream().collect(groupingBy(d→d.getType()))
```



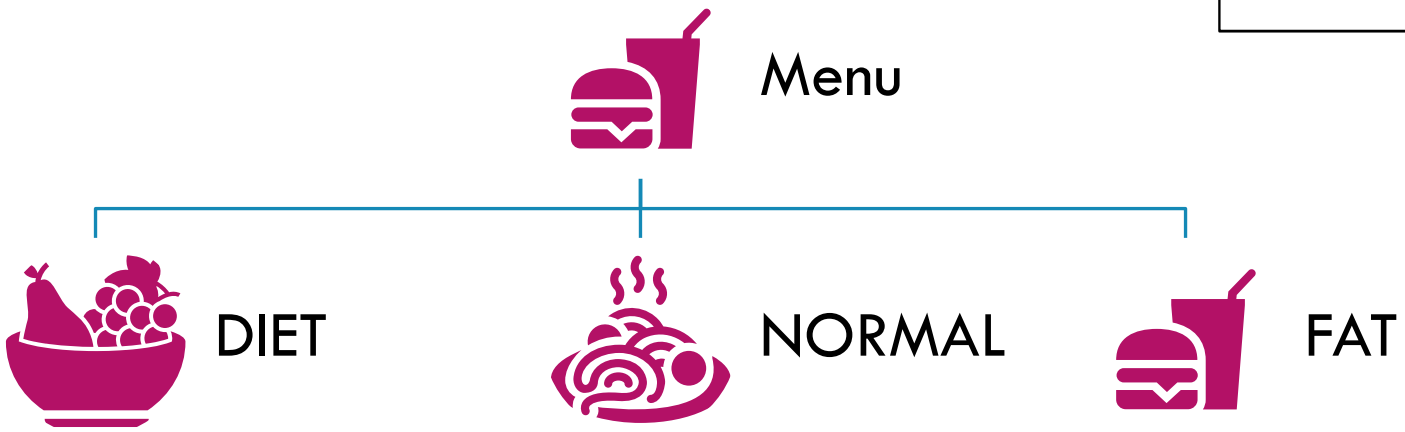
GROUPING

public enum Category { DIET, NORMAL, FAT }

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);
public String getName();
public boolean isVegetarian();
public int getCalories();
public Type getType();
public String toString();
public enum Type { MEAT, FISH, OTHER }
```

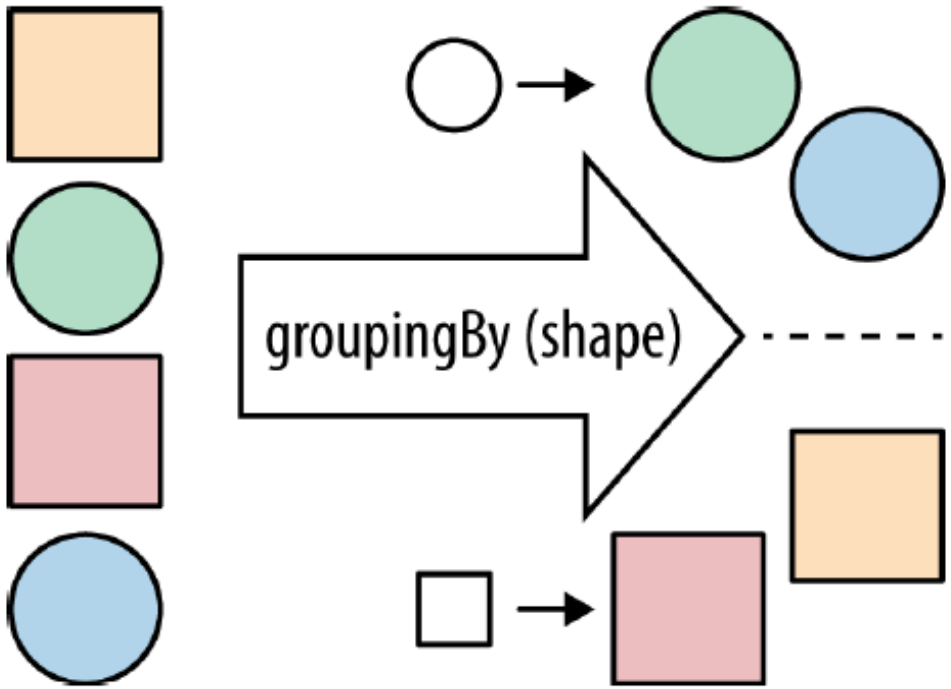


GROUPING

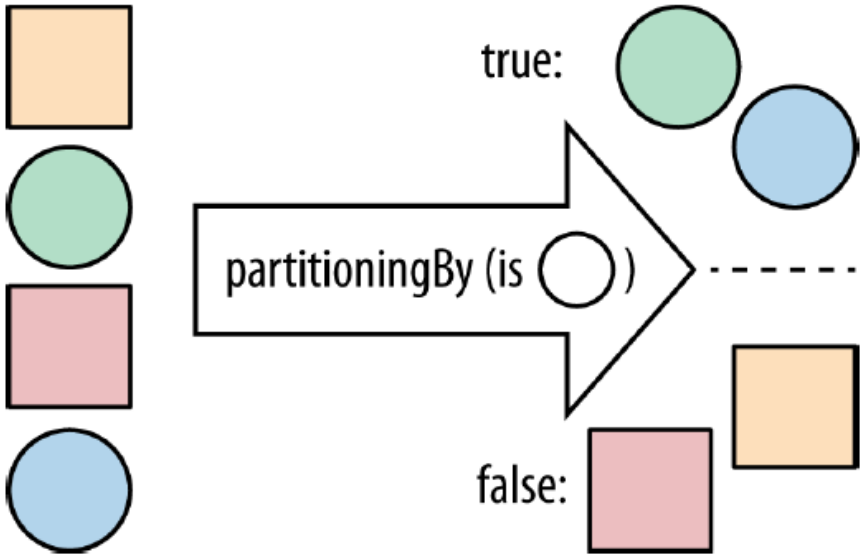
```
public enum Category { DIET, NORMAL, FAT }

Map<Category,List<Dish>>
```

```
menu.stream().collect(groupingBy(d→{if(d.getCalories()<=400)
                return Category.DIET;
            else if(dish.getCalories()<=700)
                return Category.NORMAL;
            else
                return Category.FAT;
        }));
```

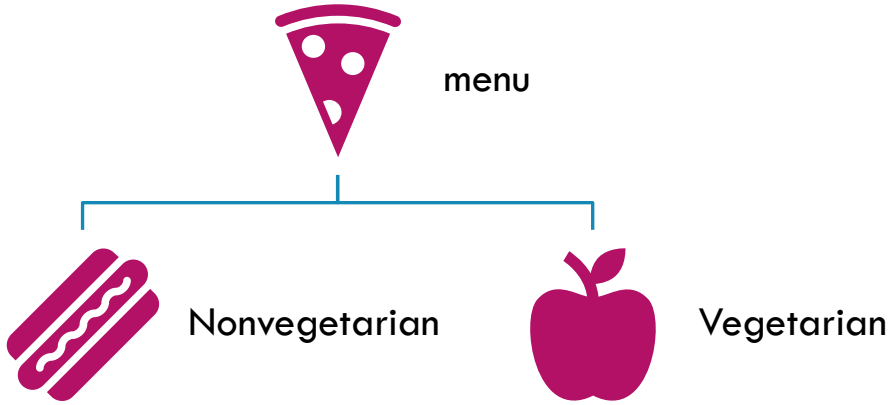


PARTITIONING



menu.stream().collect(partitioningBy(d→d.isVegetarian()));

Map<Boolean,List<Dish>> mapResults

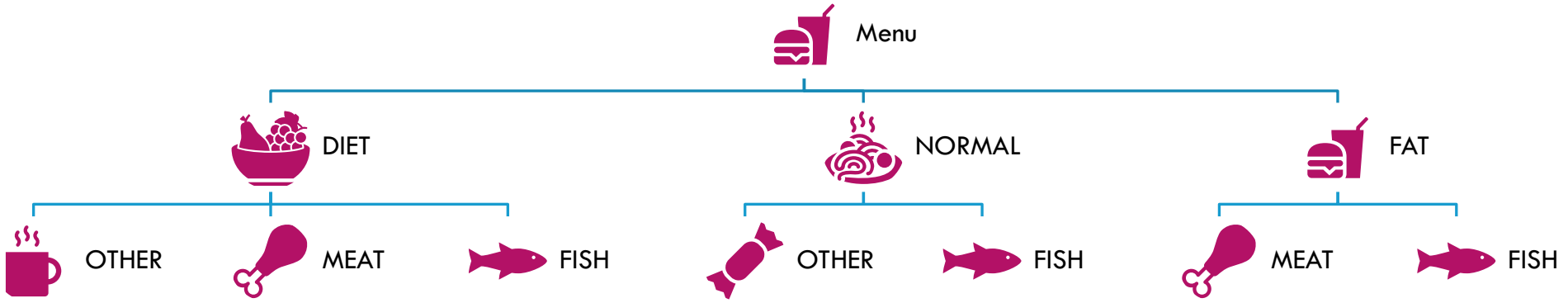


MULTILEVEL COLLECTION

Multilevel grouping by using a collector created with a two-argument version of the Collectors.groupingBy factory method

It accepts a second argument of type collector besides the usual classification function

To perform a two-level grouping, you can pass an inner groupingBy to the outer groupingBy

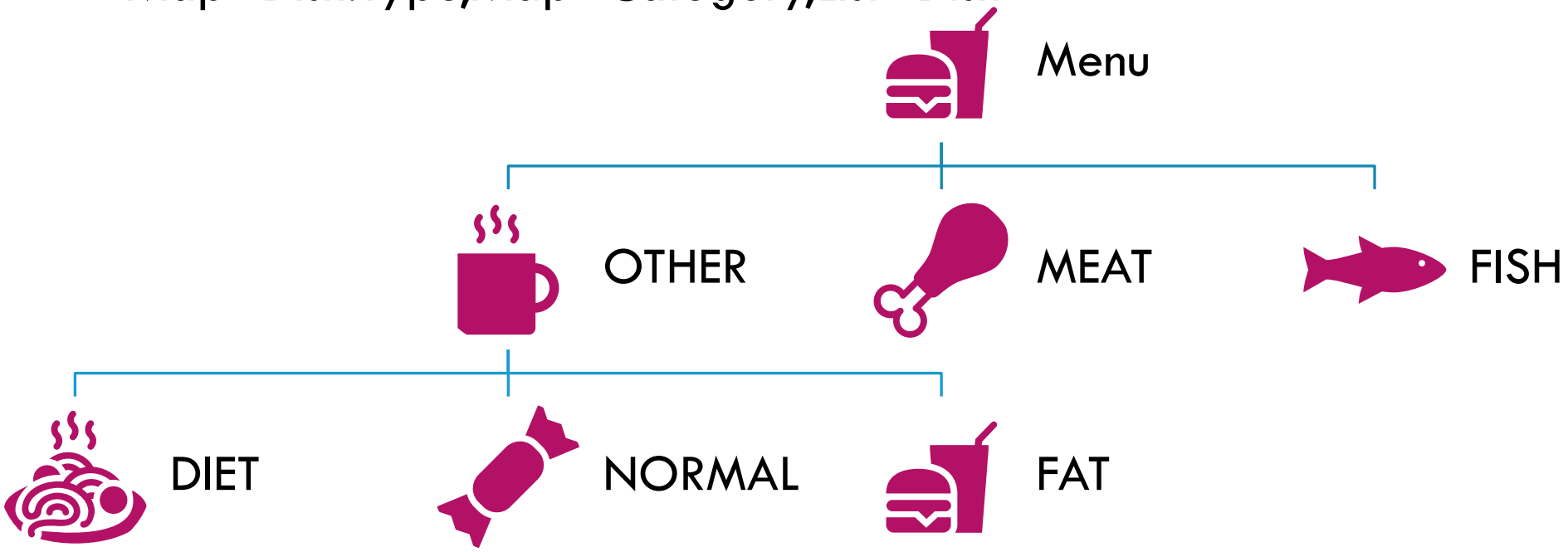


MULTILEVEL GROUPING

```
Map<Category,Map<Dish.Type,List<Dish>>> dishesByCalorie
menu.stream().collect(groupingBy(dish->{if(dish.getCalories()<=400)
                                          return Category.DIET;
                                       else if(dish.getCalories()<=700)
                                          return Category.NORMAL;
                                       else
                                          return Category.FAT;},
                                       groupingBy(d2->d2.getType())));
```

MULTILEVEL GROUPING

Map<Dish.Type,Map<Category,List<Dish>>>



How to achieve *n*-level groupings

MULTILEVEL COLLECTION

- Second level collector may not always subgroup
- Reducing and summarizing stream elements to a single value
- Grouping elements
- Partitioning elements

MULTILEVEL COLLECTION

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

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

highest-calorie Dish for a given type:

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

Each bucket gets associated with the key provided by the classifier function

The groupingBy operation then uses the downstream collector to collect each bucket and makes a map of the results

COLLECTING

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

```
menu.stream().collect(groupingBy(d->d.getType(),
maxBy(Comparator.comparingInt(d->d.getCalories()))))
```

```
Map<Dish.Type,Optional<Dish>>
```

The values in this Map are Optionals because this is the resulting type of the collector generated

by the maxBy factory method

if there's no Dish in the menu for a given type, that type won't have an Optional.empty() as value; it won't be present at all as a key in the Map

The groupingBy collector lazily adds a new key in the grouping Map only the first time it finds an element in the stream

MULTILEVEL COLLECTION

Mapping can also be done

```
albums.collect(groupingBy(Album::getMainMusician, mapping(Album::getName, toList())));
```

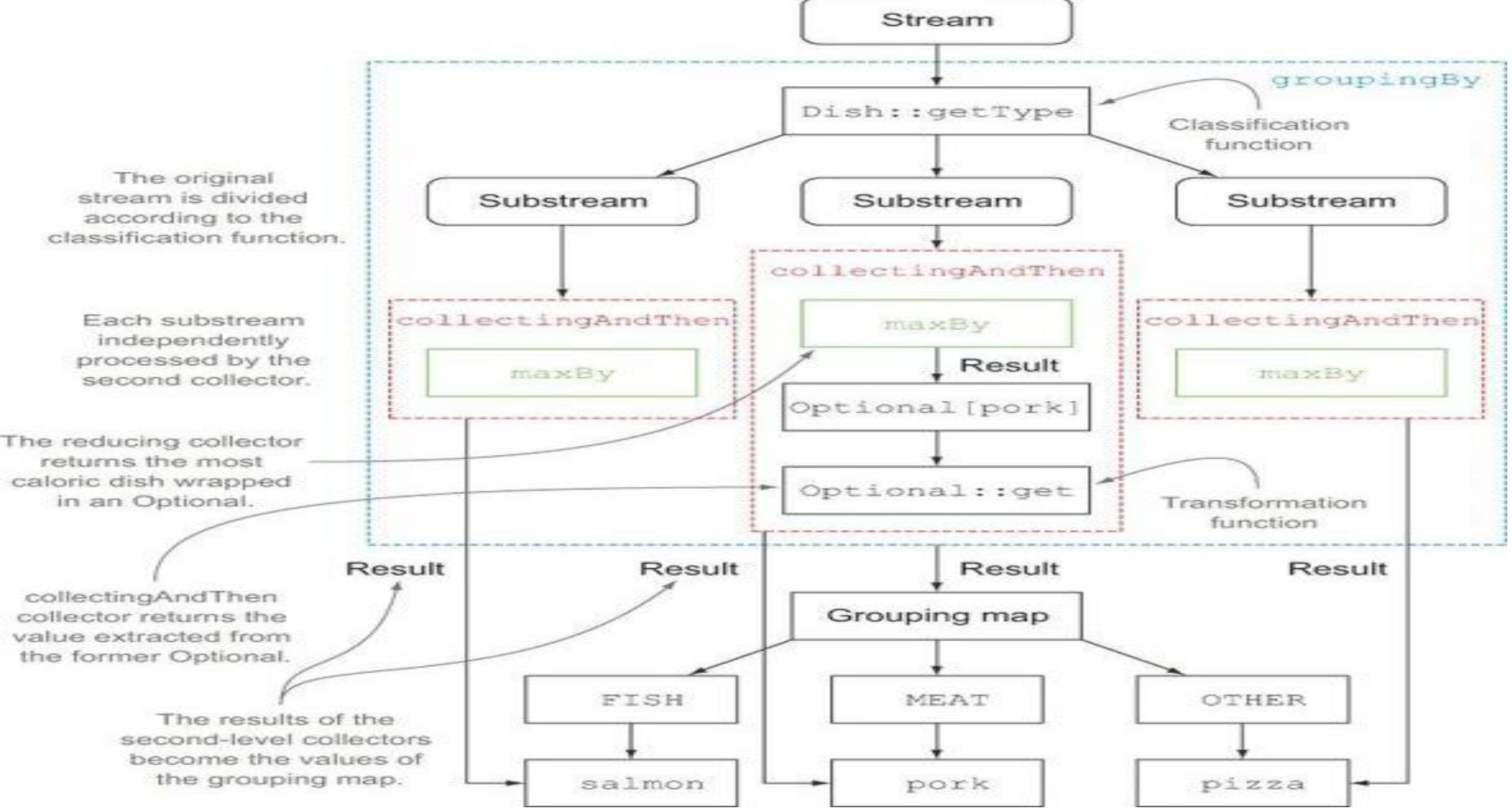
In the same way that a collector is a recipe for building a final value, a downstream collector is a recipe for building a part of that value, which is then used by the main collector

COLLECTING AND THEN WRAPPING

```
Map<Dish.Type,Dish> result4=menu.stream().collect(groupingBy(d->d.getType(), collectingAndThen(maxBy(Comparator.comparingInt(d->d.getCalories())), s->s.get())))
```

This factory method takes two arguments, the collector to be adapted and a transformation function, and returns another collector

This additional collector acts as a wrapper for the old one and maps the value it returns using the transformation function as the last step of the collect operation


Stream
groupingBy
Dish::getType
Classification function
The original stream is divided according to the classification function.
Substream
Substream
Substream
collectingAndThen
collectingAndThen
maxBy
collectingAndThen
maxBy
Each substream independently processed by the second collector.
maxBy
Result
Optional[pork]
The reducing collector returns the most caloric dish wrapped in an Optional.
Optional::get
Transformation function
Result
Result
Result
Result
collectingAndThen collector returns the value extracted from the former Optional.
Grouping map
FISH
MEAT
OTHER
The results of the second-level collectors become the values of the grouping map.
salmon
pork
pizza

ANY TYPE OF COLLECTION

```
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) )));
```