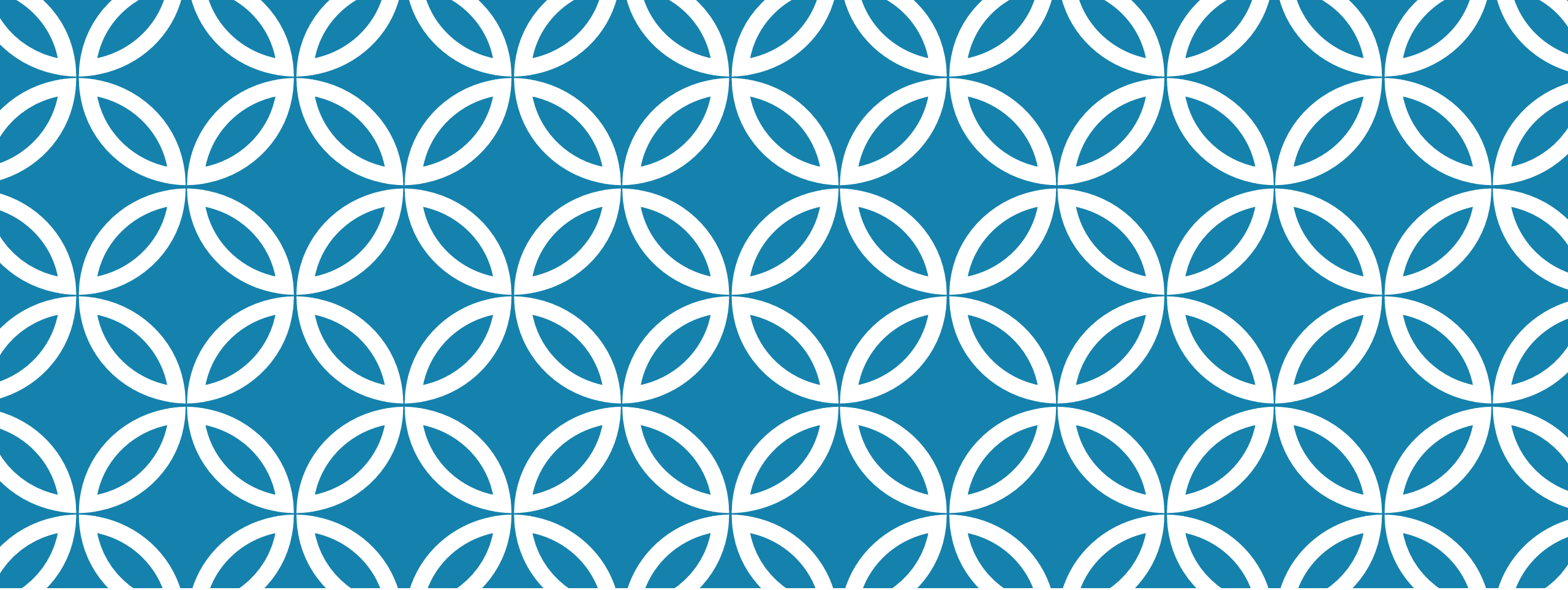


COLLECTING STREAMS

Chandreyee Chowdhury

THE COLLECTOR FUNCTIONS

- A Collector is specified by four functions that work together to accumulate entries into a mutable result container, and optionally perform a final transform on the result. They are:
  - creation of a new result container (supplier())
  - incorporating a new data element into a result container (accumulator())
  - combining two result containers into one (combiner())
  - performing an optional final transform on the container (finisher())
- Collectors also have a set of characteristics, such as Collector.Characteristics.CONCURRENT, that provide hints that can be used by a reduction implementation to provide better performance.

THE COLLECTOR FUNCTIONS

- A sequential implementation of a reduction using a collector would
  - create a single result container using the supplier function
  - and invoke the accumulator function once for each input element.
- A parallel implementation would
  - partition the input
  - create a result container for each partition,
  - accumulate the contents of each partition into a subresult for that partition
  - use the combiner function to merge the subresults into a combined result
  - The combiner may fold state-returns a BinaryOperator

COLLECTING STREAMS

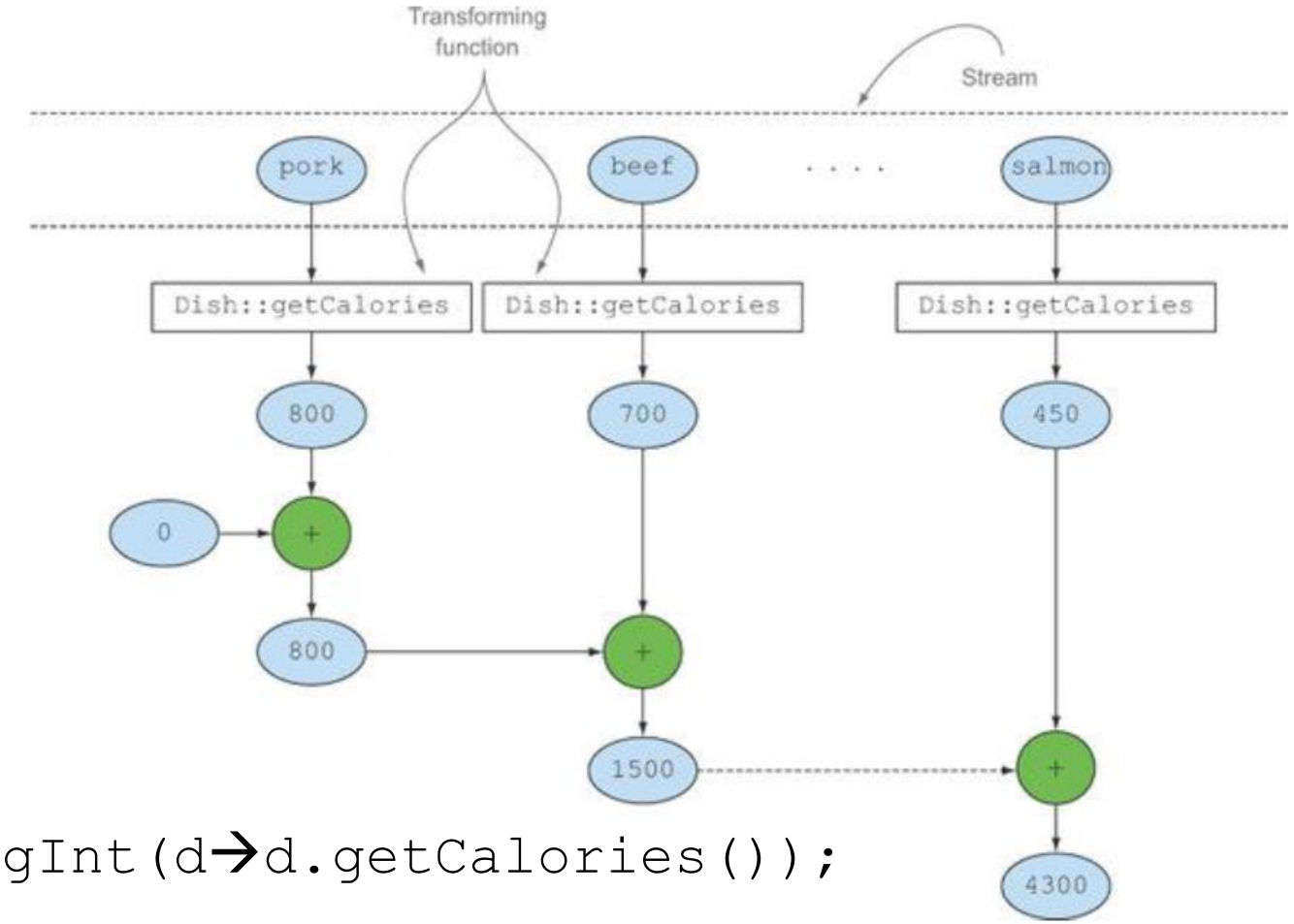
- Reducing and summarizing stream elements to a single value
- Grouping elements
- Partitioning elements

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

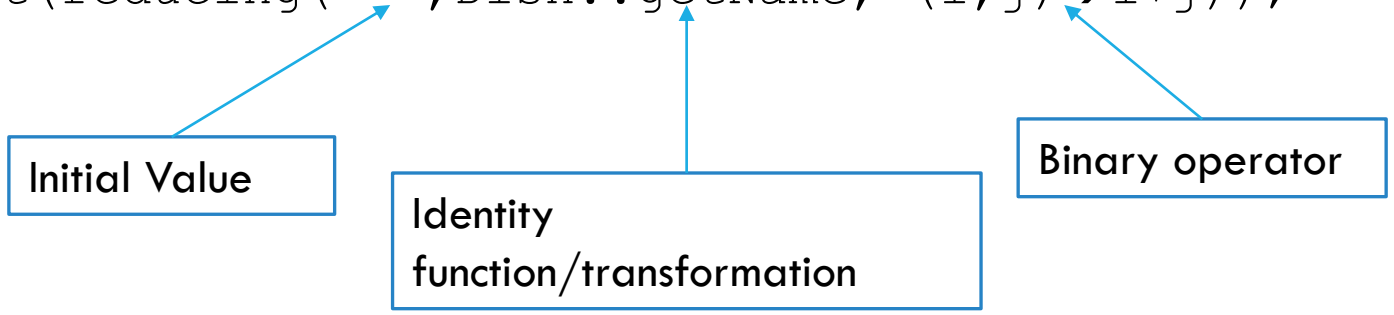


- Collectors.summingInt()
- `menu.stream().collect(summingInt(d→d.getCalories());`
- averagingInt()
- summarizingInt()
- IntSummaryStatistics

JOINING STRINGS

```
String results=menu.stream().filter(d→d.isVegetarian())
                            .map(d→d.getName())

                            .collect(Collectors.joining(“,”,”[“,”]”);
results=menu.stream().collect(reducing(“ “,Dish::getName, (i,j)→i+j));
```



`Joining()` internally makes use of a StringBuilder to append the generated strings into one

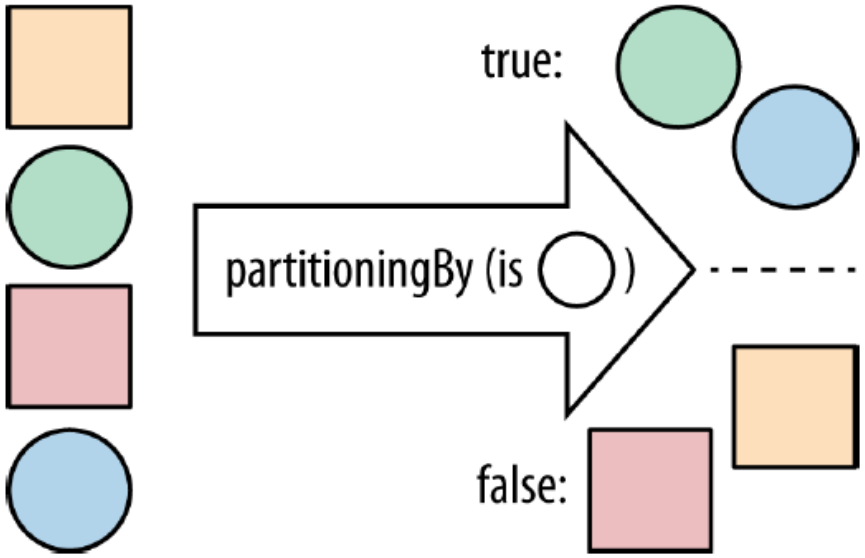
REDUCING

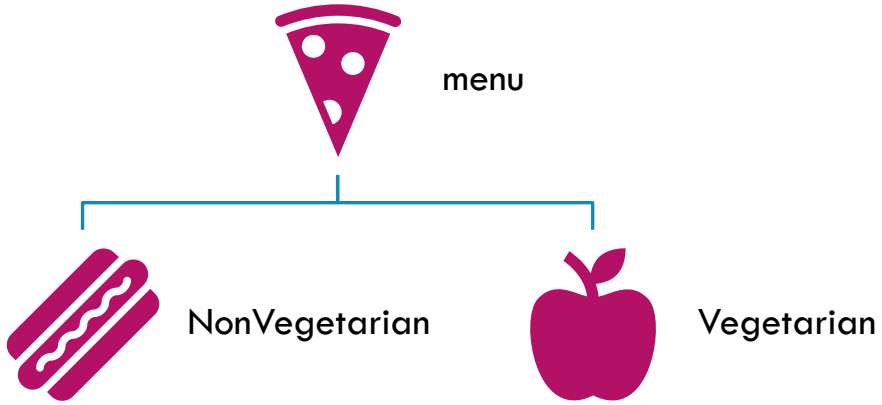
```
results=menu.stream().
                collect(reducing(" ",Dish::getName, (i,j)→i+j));

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

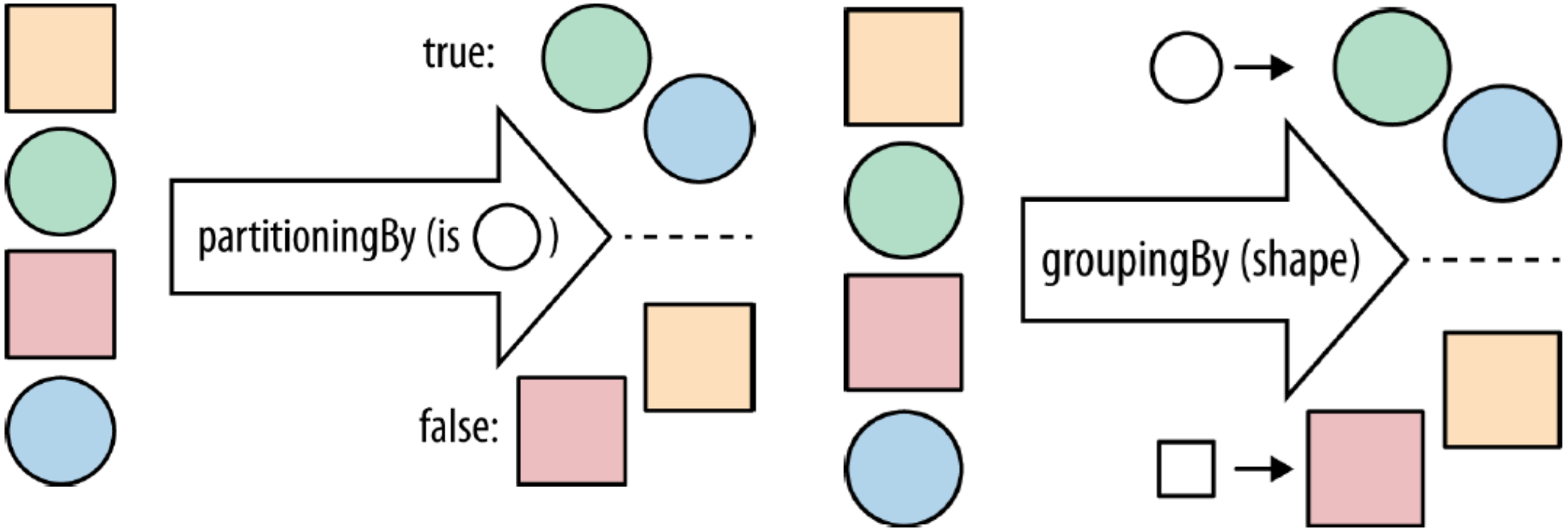
PARTITIONING

```
Map<Boolean,List<Dish>> mapResults=
menu.stream().collect(partitioningBy(d→d.isVegetarian()));
```





COLLECTING STREAM ELEMENTS



GROUPING

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

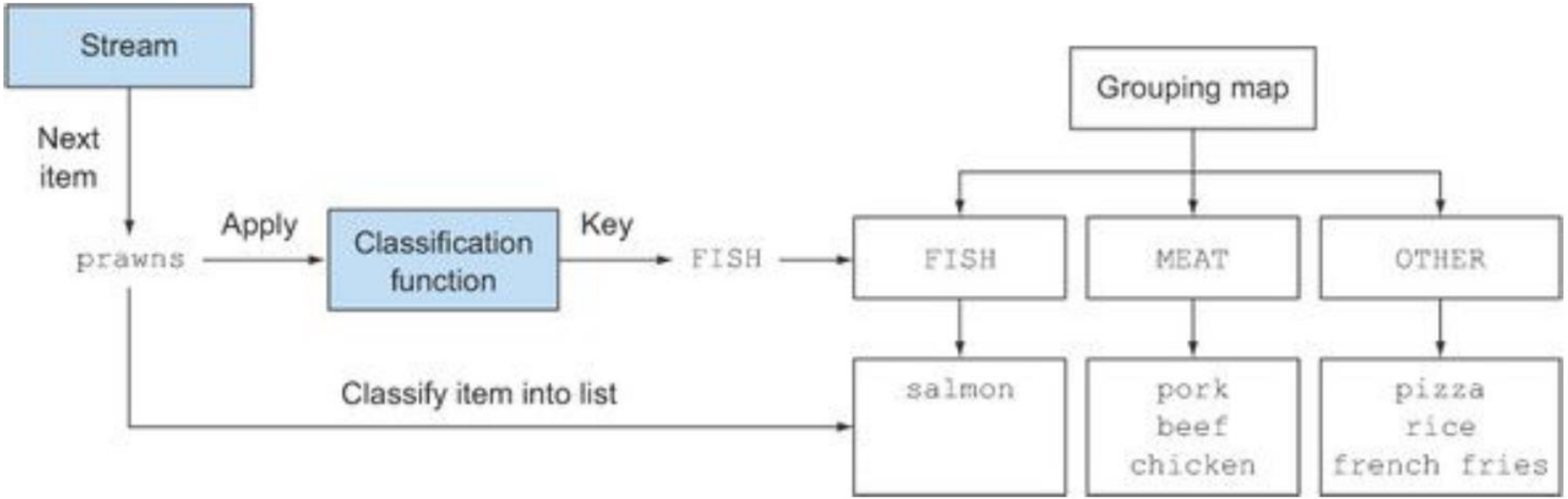
We call this Function a *classification* function because it's used to classify the elements of the stream into different groups.

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 }
```

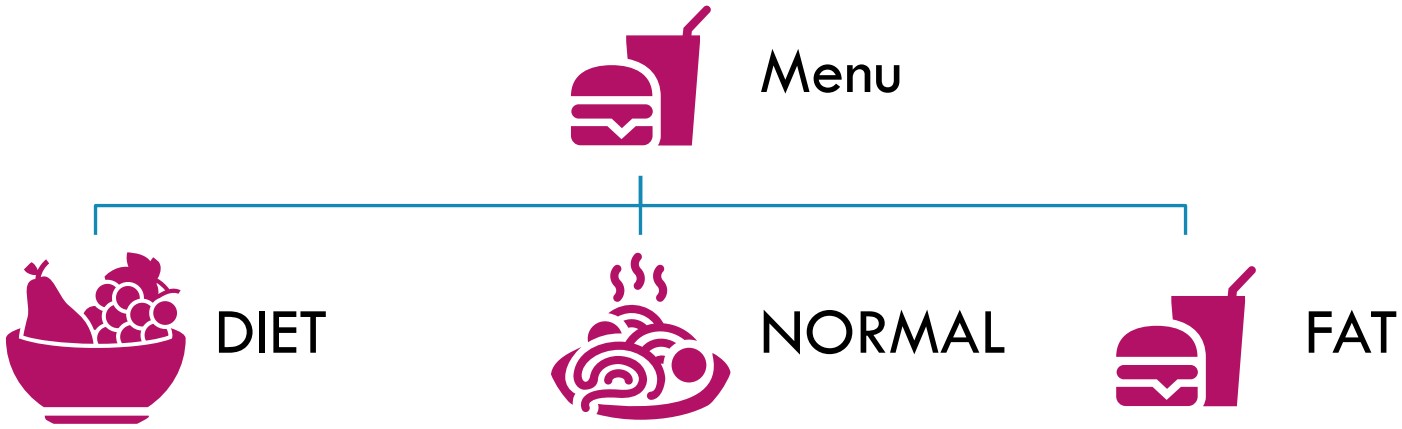
GROUPING


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

GROUPING

It isn't always possible to use a method reference as a classification function, because you may wish to classify using something more complex than a simple property accessor

public enum Category { DIET, NORMAL, FAT }



EXTRACTING GROUP-WISE FEATURES

- 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
- The regular one-argument groupingBy(f), where f is the classification function, is in reality just shorthand for groupingBy(f, toList()).

MULTILEVEL COLLECTION

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

GROUPWISE FEATURES

- {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]}`
- groupingBy works in terms of “buckets.”
- The first groupingBy creates a bucket for each key. You then collect the elements in each bucket with the downstream collector
- 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

DO YOU REMEMBER?

- 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));`

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

EXTRACTING GROUP FEATURES

- Mapping can also be done

```
albums.collect(groupingBy(Album::get
MainMusician,
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
- This method takes two arguments: a function transforming the elements in a stream and a further collector accumulating the objects resulting from this transformation.

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

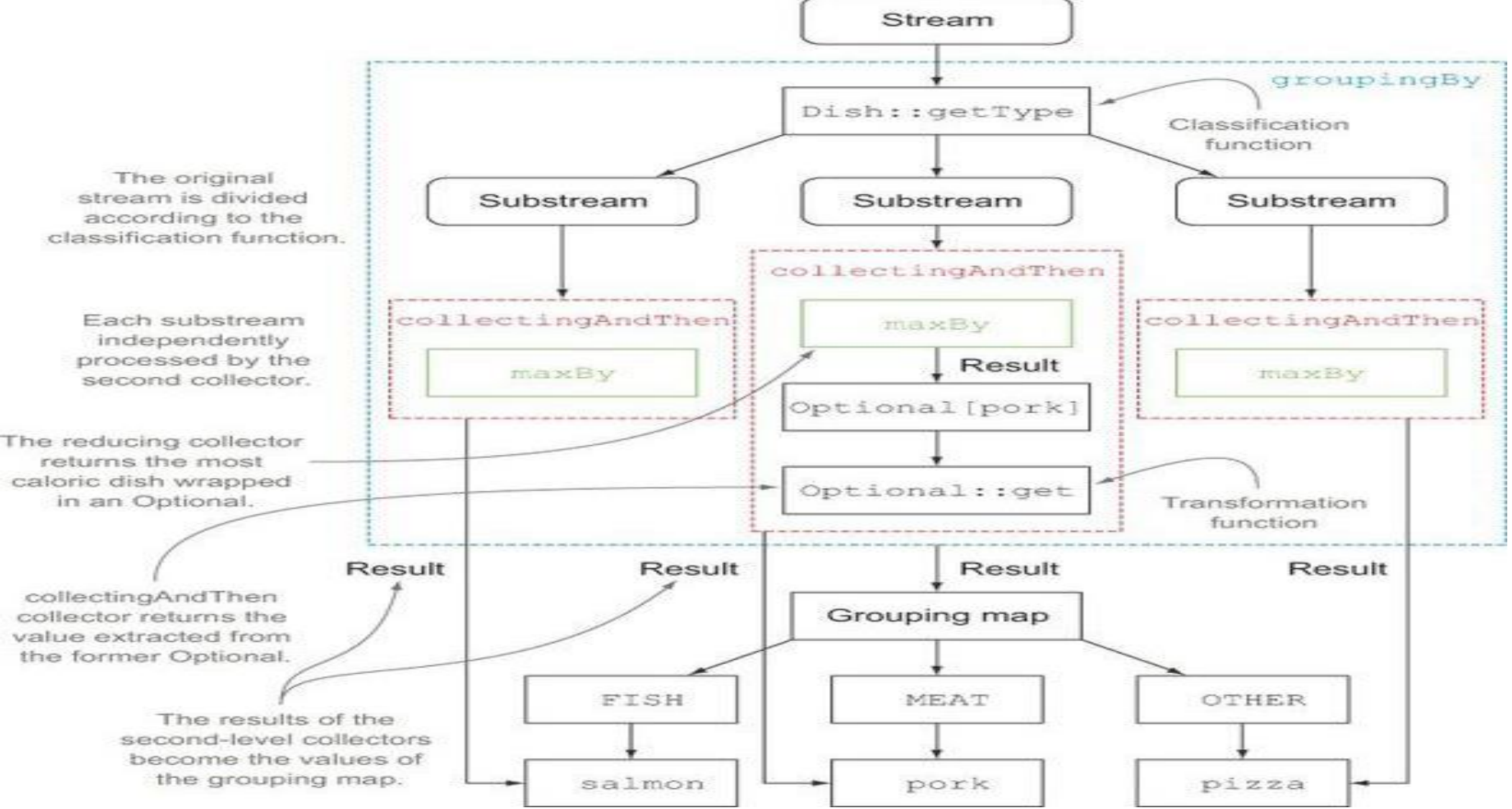
COLLECTING AND THEN WRAPPING

```
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

```
collectingAndThen(Collector<T,A,R> downstream, Function<R,RR> finisher)
```

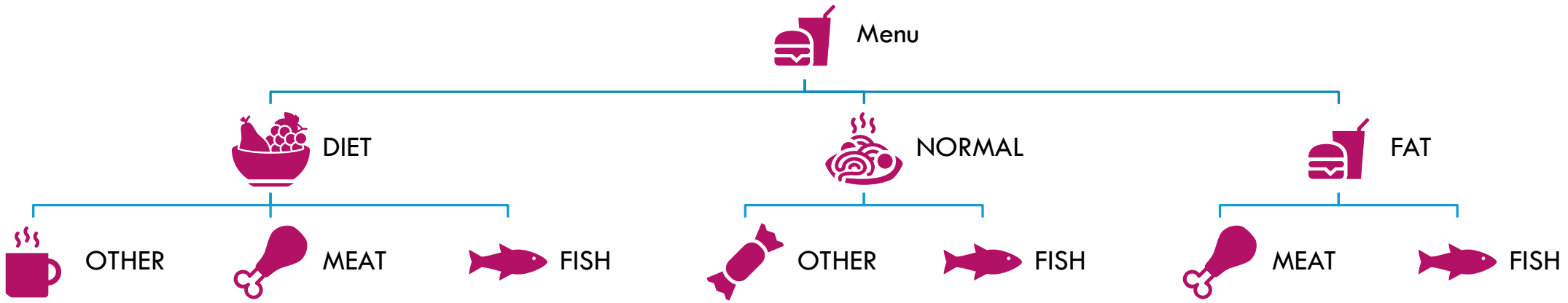

Stream
groupingBy
Dish::getType
Classification function
The original stream is divided according to the classification function.
Substream
Substream
Substream
collectingAndThen
collectingAndThen
maxBy
collectingAndThen
maxBy
Each substream independently processed by the second collector.
maxBy
Result
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) )));
```

EXTRACTING GROUP-WISE FEATURES

- The regular one-argument groupingBy(f), where f is the classification function, is in reality just shorthand for groupingBy(f, toList()).
- 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())));
```

MORE GROUPNGS

```
groupingBy(Function<T, K> classifier, Collector<T,A,D> downstream)
```

How to achieve *n*-level groupings

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

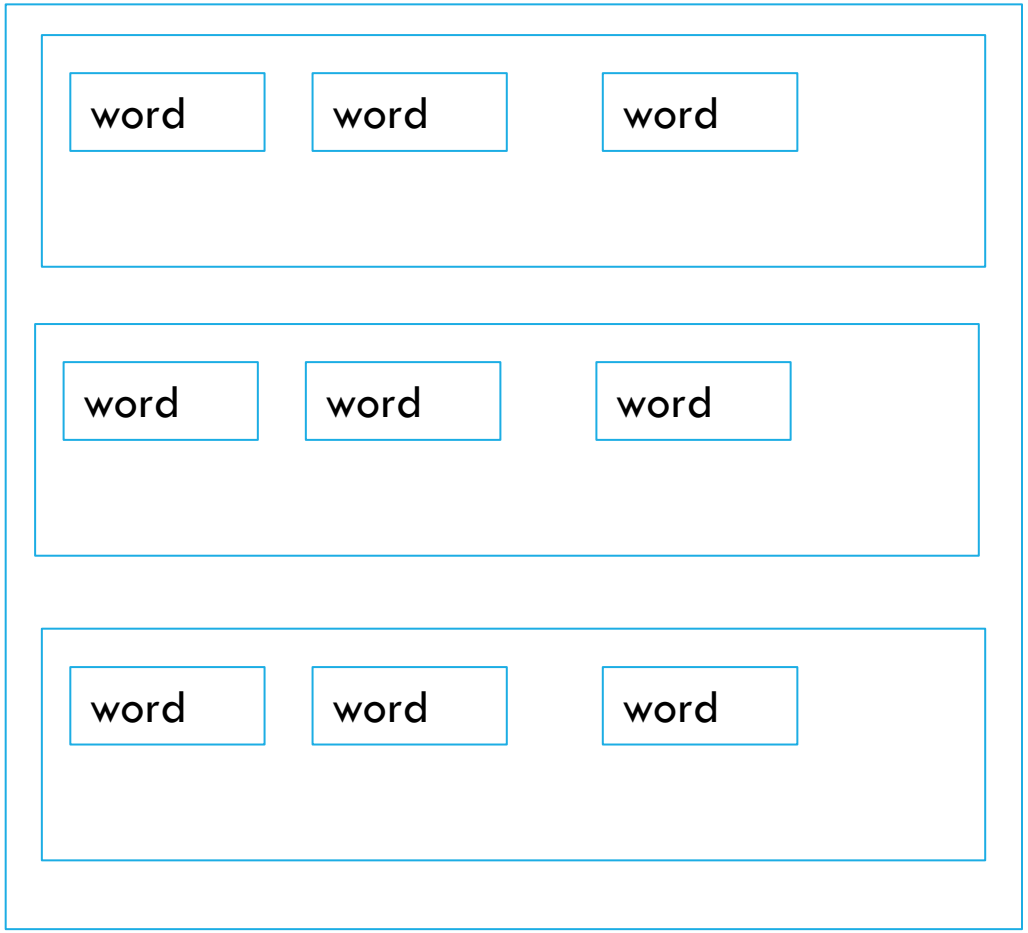
Static methods

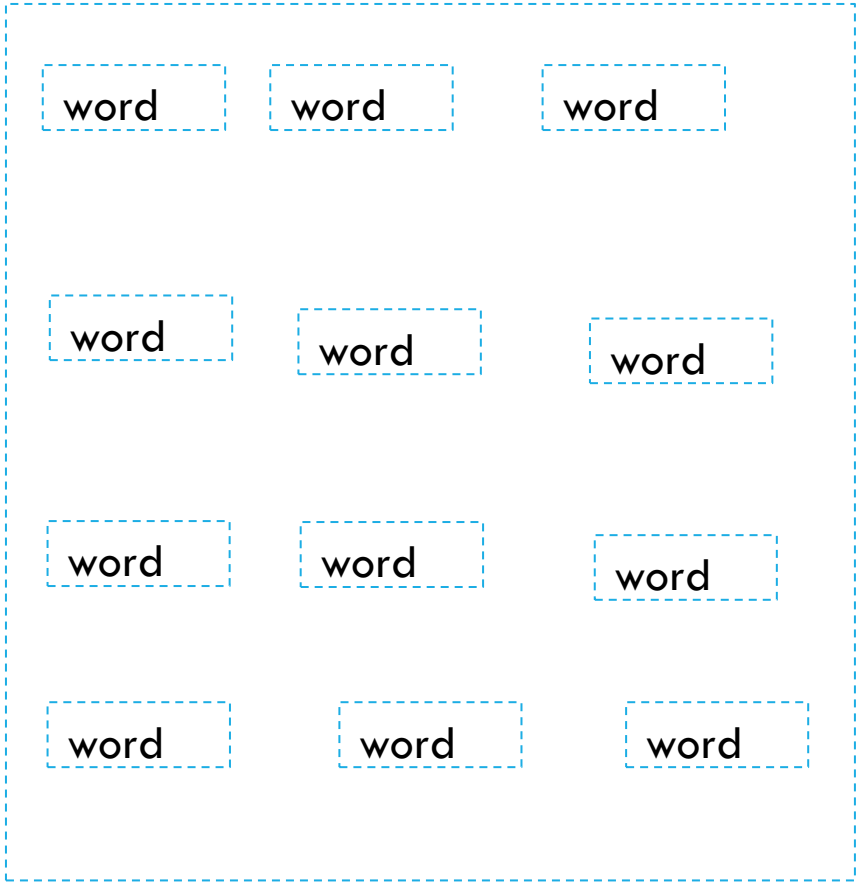
- Stream.of(“ ”, “”, “ ”,” ”);
- Stream.empty()
- Arrays.stream(1,2,3,4)
- Str.chars()

- 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){}
```

EXAMPLES

- Identify and list the distinct letters;
- Group it’s words into three categories depending on word length-2-letter words, 3-letter words and more than 3 letter words.

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