Collection data with streams

Chapter 5

Grouping the hard pre Java 8 way

```
@Test
public void howToDoSomethingSimpleAsGroupingApplesByColour() {

    Map<String,List<Apple>> listsGroupedByColour= new HashMap<>();
    for (Apple apple : stock) {
        String colour = apple.getColour();
        if(listsGroupedByColour.get(colour)==null){
            ArrayList<Apple> fixedColourList = new ArrayList<>();
            listsGroupedByColour.put(colour,fixedColourList);
        }
        listsGroupedByColour.get(colour).add(apple);
    }

    Set<String> colours = listsGroupedByColour.keySet();

    printMap(listsGroupedByColour, colours);
}
```

The Java 8 way

```
@Test
public void groupApplesByColour() {
    Stream<Apple> appleStream = stock.stream();

    Map<String, List<Apple>> appleGroups =

    appleStream.collect(groupingBy(Apple::getColour));

    Set<String> colours = appleGroups.keySet();

    printMap(appleGroups, colours);

}
```

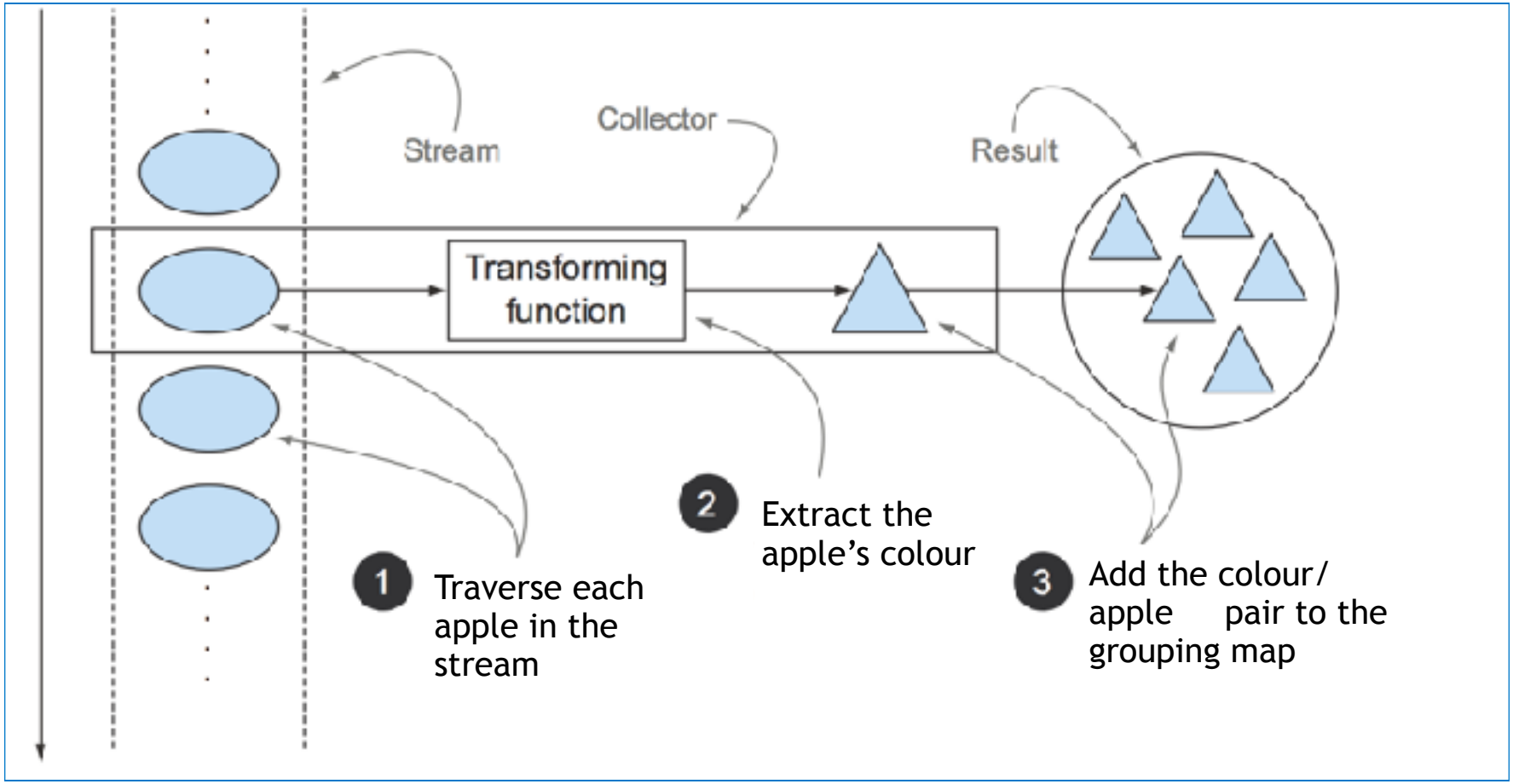
the Java 8 way

- Specify the what
- Not the how
- Better to read
- Better to maintain especially in multi level groupings

Collector in a nutshell

- The collect method expects an implementation of the Collector interface
- Collector is an interface of how to build a summary of elements in the stream
- the toList says: make a list of all the elements in the stream
- likewise groupingBy says make a map whose keys are colour buckets and whose values are Lists of apples of that colour

A picture of collection process



Collectors as advanced reductions

- invoking the collect method on a stream
- triggers a reduction operation
- parameterized by the Collector interface
- Typically the Collector does the following
  - applies a transformation (often identity)
  - and accumulates the result in a data structure

Predefined collectors

- The Collectors class defines collectors

- Fall into 3 different groups:
- Reducing and summarising to one value
- Grouping elements
- Partitioning elements

Reducing and summarising to one value

```
@Test
public void countTheNumberOfApplesInStock() throws Exception {

    Collector<Apple, ?, Long> appleCounter = Collectors.counting();
    Long numberOfApples = stock.stream().collect(appleCounter);

    assertThat(numberOfApples,is(4L));

    //Note: same can be achieved with:

    stock.stream().count();

    //Collectors.counting() will show it's usefulness
}
```

minBy and maxBy

```
@Test
public void findingMinAndMaxValues() throws Exception {

    Comparator<Apple> appleWeightComparator=
            (apple1,apple2)->apple1.getWeight()-apple2.getWeight();

    Optional<Apple> hwApple =
           stock.stream().collect(Collectors.maxBy(appleWeightComparator));

    Optional<Apple> lwApple =
           stock.stream().collect(Collectors.minBy(appleWeightComparator));

    hwApple.ifPresent(System.out::println);

    lwApple.ifPresent(System.out::println);

}
```

Summarization

```
@Test
public void countingTheTotalWeightOfTheStockOfApples() throws Exception {

    Integer totalWeight =
            stock.stream().collect(summingInt((apple)->apple.getWeight()));

    System.out.println(totalWeight);

    IntSummaryStatistics appleStatistics =
            stock.stream().collect(summarizingInt(Apple::getWeight));

    System.out.println(appleStatistics);
}
```

Output println: IntSummaryStatistics
{count=4, sum=550, min=80, average=137.500000, max=220}

Generalized summarization

```
@Test
public void generalSummarization() throws Exception {
    Integer totalWeightSummingInt =
            stock.stream().collect(summingInt((apple)->apple.getWeight()));

    //is special case of:

    Function<Apple,Integer> transformer;
    BinaryOperator<Integer> aggregator;
    Integer startValue;

    startValue=0;
    transformer=Apple::getWeight;
    aggregator=(i,j)-> i+j;
    Integer totalWeightGeneralizedReduction =

            stock.stream().collect(reducing(startValue,transformer,aggregator));

    assertThat(totalWeightSummingInt, is(totalWeightGeneralizedReduction));

}
```

Reducing example

```
@Test
public void reducingWithOnlyAnAggregator()  {

    Function<Apple,Apple> transformer;
    BinaryOperator<Apple> aggregator;
    Apple firstAppleInTheStream;

    firstAppleInTheStream=new Apple("dummy",0);
    aggregator=(apple1,apple2)->
                    apple1.getWeight()>apple2.getWeight()?apple1:apple2;
    transformer=(apple)-> apple;//indentity operation

    Apple heaviestAppleByReduction1 =
            stock.stream().collect(reducing(firstAppleInTheStream,
                                            transformer,aggregator
                                            )
                                   );

    //Nearly equal to reducing(aggregator)
    Optional<Apple> heaviestAppleByReduction2 =
         stock.stream().collect(reducing(aggregator));
}
```

Collect versus reduce

- reduce is meant to combine 2 values and produce a new one
- i.e. reduce is an immutable reduction
- the collect method is meant to mutate a container that is supposed to accumulate the result

multiple ways to perform the same operation

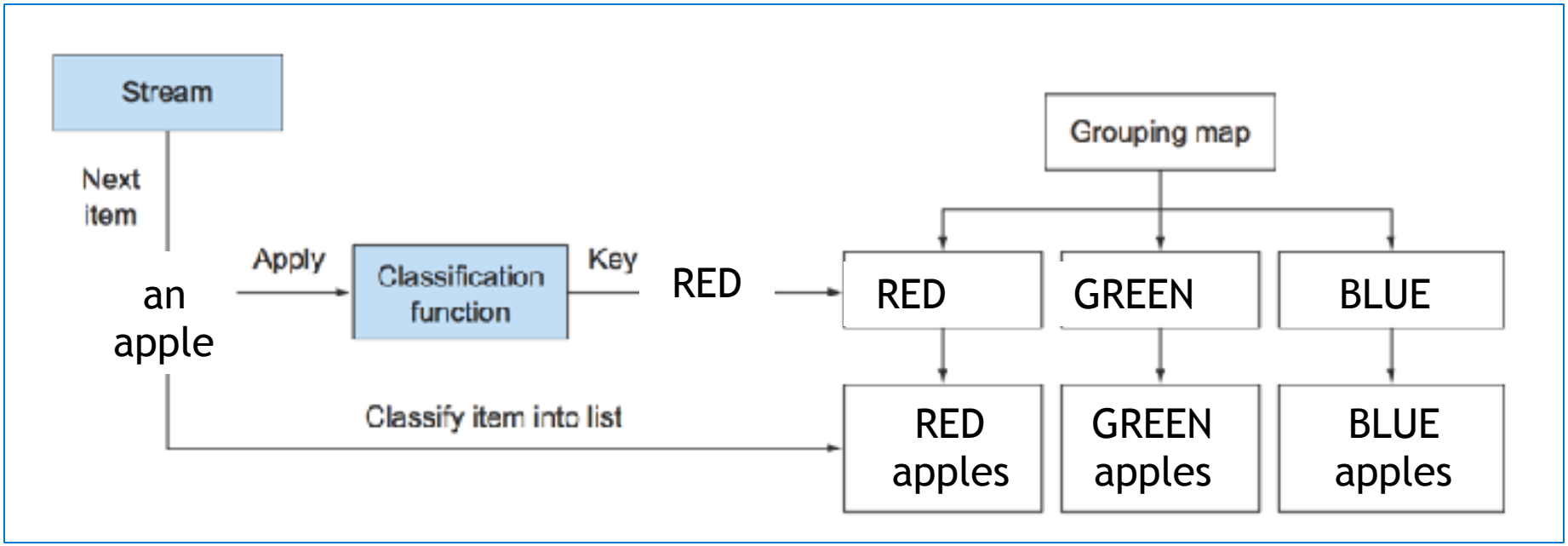
```
@Test
public void alternativesOfCounting() throws Exception {

    Integer x=0;
    BinaryOperator<Integer> accumulator=(i,j) -> i+j;
    Function<Apple,Integer> y=(a)->a.getWeight();
    Integer sumWay0 = stock.stream()
            .collect(reducing(x,y,accumulator));

    Integer sumWay1 = stock.stream()
            .collect(reducing(0, Apple::getWeight,Integer::sum));

    Integer sumWay2 = stock.stream()
            .map((a)-> a.getWeight()).reduce(accumulator).get();

    int sumWay3 = stock.stream().
            mapToInt(a->a.getWeight()).reduce((i,j)->i+j).get();

}
```

what to choose?

- Suggestion
  - explore the largest number of solutions
  - choose most specialised and general enough to solve the problem at hand
  - this goes often hand in hand with readability and performance

- In our example
  - calculate total weight of stock prefer last example
  - it’s concise and no (un)boxing overhead

groupingBy


Stream
Next item
an apple
Apply
Classification function
Key
RED
Grouping map
RED
GREEN
BLUE
Classify item into list
RED apples
GREEN apples
BLUE apples

groupingBy an expression

```
@Test
public void itsTimeToGroup() throws Exception {

    Function<Apple, String> classifier = Apple::getColour;

    Collector<Apple, ?, Map<String, List<Apple>>> collector =
                                                    groupingBy(classifier);

    Map<String, List<Apple>> groups = stock.stream().collect(collector);

    Set<String> keys = groups.keySet();

    for (String colour : keys) {
        for (Apple apple : groups.get(colour)) {
            System.out.println(apple);
        }
    }
}
```

groupingBy an expression

```
@Test
public void groupByExpression() throws Exception {

    Function<Apple, Integer> classifier =
                    (apple)->(apple.getWeight())/100 ;

    Collector<Apple, ?, Map<Integer, List<Apple>>> collector =
                    groupingBy(classifier);

    Map<Integer, List<Apple>> groups = stock.stream().collect(collector);

    Set<Integer> keys = groups.keySet();

    System.out.println(groups);
}
```

multilevel grouping

```
@Test
public void groupingByWeightAndForEachWeightGroupGroupingByColour(){

    Function<Apple, String> colourClassifier = Apple::getColour;

    Collector<Apple, ?, Map<String, List<Apple>>> colourCollector
                            = groupingBy(colourClassifier);

    Function<Apple, Integer> weightClassifier
                            = (apple)->(apple.getWeight())/100 ;

    Collector<Apple, ?, Map<Integer, Map<String, List<Apple>>>> multiCollector
                            = groupingBy(weightClassifier, colourCollector);
    Map<Integer, Map<String, List<Apple>>> groups
                            = stock.stream().collect(multiCollector);

    System.out.println(groups);
}
```

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

weight colour	GREEN	RED
100 gram	new Apple(110,GREEN) new Apple(150,GREEN)	new Apple(150,RED) new Apple(180,RED)
200 gram	new Apple(210,GREEN)	

Using different type of collector as 2nd element

```
@Test
public void groupingByWeightAndCountingApplesInEachWeightGroup(){

    Function<Apple, Long> weightClassifier = (apple)->(apple.getWeight())/100L ;
    //Note: reducing(x,y,z);
    // x is start value
    // y is transformer
    // z is aggregator
    Collector<Apple,?,Long> counter= reducing(0L,(apple)-> 1L,(i,j)->i+j);
    // equivalent to:
    counter=Collectors.counting();

    Collector<Apple, ?, Map<Long, Long>> weightCollector
                                = groupingBy(weightClassifier,counter);
    Map<Long, Long> groups
                                 = stock.stream().collect(weightCollector);

    System.out.println(groups);
}
```

Note: Apple versus Optional<Apple>

```
@Test
public void groupingByWeightAndFindMostHeavyAppleInEachGroup(){

    Function<Apple, Long> weightClassifier = (apple)->(apple.getWeight())/100L ;
    //Note: reducing(x,y,z);
    // x is start value,y is transformer,z is aggregator
    Collector<Apple,?,Apple> heavyAppleSearcher=
             reducing(stock.get(0),(a)-> a,
                                (a1,a2)->a1.getWeight()>a2.getWeight()?a1:a2);
    // nearly equivalent to:
    Comparator<Apple> comparator=(a1,a2)-> a1.getWeight()-a2.getWeight();
    Collector<Apple, ?, Optional<Apple>> heavyAppleSearcher1 =
                                                  Collectors.maxBy(comparator);

    Collector<Apple, ?, Map<Long, Optional<Apple>>> weightCollector
                               = groupingBy(weightClassifier,heavyAppleSearcher1);
    Map<Long, Optional<Apple>> groups
                                = stock.stream().collect(weightCollector);
    System.out.println(groups);
}
```

Optional<Apple> incidentally used

- Optional<Apple>
  - not expressing absence of value
  - incidentally there because of the signature of the reduce operation maxBy
  - the groupingBy collector lazily adds a key to the map when it finds one
  - the absence of a key is not noticed by design

- To get rid of the Optional
  - change the type returned
  - use collectiongAndThen

collectingAndThen to change the type returned

```
@Test
public void findMostHeavyAppleInEachWeightGroupAndThenChangeTheTypeReturned(){

    Comparator<Apple> comparator=(a1,a2)-> a1.getWeight()-a2.getWeight();
    Collector<Apple, ?, Optional<Apple>> heavyAppleSearcher1 = maxBy(comparator);

    Collector<Apple, ?, Optional<Apple>> downstream=heavyAppleSearcher1;
    Function<Optional<Apple>,Apple> finisher=(optApple)->optApple.get();

    Collector<Apple, ?, Apple> collectorAndTypeChanger =
                                        collectingAndThen(downstream, finisher);

    Function<Apple, Long> weightClassifier = (apple)->(apple.getWeight())/100L ;

    Collector<Apple, ?, Map<Long, Apple>> weightCollector
                        = groupingBy(weightClassifier,collectorAndTypeChanger);

    Map<Long, Apple> groups = stock.stream().collect(weightCollector);

    System.out.println(groups);
}
```

About the second argument to the groupingBy factory method

- The collector passed as a second argument
  - performs a further reduction operation on all the elements in the stream classified into the same group

mapping: transform elements before collecting

```
@Test public void mappingElementsInAGroup(){

    Function<Apple, Long> weightClassifier = (apple)->(apple.getWeight())/100L ;

    Collector<Apple,?,Set<CaloricLevel>> mapper=
             mapping((Apple apple)-> {
                                          if(apple.getWeight()<120) {
                                              return CaloricLevel.DIET;
                                          }else if(apple.getWeight()<180) {
                                              return CaloricLevel.NORMAL;
                                          }else {
                                              return CaloricLevel.HEAVY;
                                          }
                                      },toSet());
    Collector<Apple, ?, Map<Long, Set<CaloricLevel>>> weightCollector
                                = groupingBy(weightClassifier,mapper);
    Map<Long, Set<CaloricLevel>> groups
                                 = stock.stream().collect(weightCollector);
    System.out.println(groups);
}
```

Notes about former slide

- Use the toSet method to remove duplicates
- When you want to specify the type of Set returned use the following syntax

```
Supplier<HashSet<CaloricLevel>> collectionFactory=HashSet::new;

Collector<Apple,?,HashSet<CaloricLevel>> mapper=
         mapping((Apple apple)-> {
                                        if(apple.getWeight()<120) {
                                             return CaloricLevel.DIET;
                                        }else if(apple.getWeight()<180) {
                                             return CaloricLevel.NORMAL;
                                        }else {
                                             return CaloricLevel.HEAVY;
                                        }

                                   },
                                   toCollection(collectionFactory));
```

Partitioning: a special case of groupingBy

- having a predicate as a partitioning function that serves as a classifier
- result in 2 groups
  - group for which predicate returns true and
  - a group for which predicate return false

```
@Test
public void partitionApplesInRedAndNonRed() {

    Predicate<Apple> applePredicate=
            (apple)-> "RED".equalsIgnoreCase(apple.getColour());

    Map<Boolean, List<Apple>> collourGroups =
            stock.stream().collect(partitioningBy(applePredicate));

    System.out.println(collourGroups);
}
```

Divide numbers in prime and non prime numbers

```
public boolean isPrime(int n) {
    int upperLimit=(int)Math.sqrt(n);
    //Note: start at 2, everything is divisible by 1
    IntStream rangeClosed = IntStream.rangeClosed(2,upperLimit);
    IntPredicate intPredicate =(i)-> (n %i)==0;
    return (rangeClosed.noneMatch(intPredicate ));
}


@Test
public void partitionARangeOfValuesINPrimeAndNonPrime() throws Exception {
    final int n=1000;
    IntStream range = IntStream.range(1, n);

    Predicate<Integer> numberPredicate=i ->isPrime(i);
    Map<Boolean, List<Integer>> primeAndNonPrimeGroups =
            range.boxed().collect(partitioningBy(numberPredicate));
    System.out.println(primeAndNonPrimeGroups);
}
```

See static members of Collectors class for collectors