i) toList()
ii) toSet() > The exact implementation depends on the system. Thread safety is not maintained
iii) toCollection() – programmer specifies the exact collection
stream.collect(toCollection(TreeSet::new));

classmate
Date 03/02/17
Page

Collecting Data with Streams

Collect is a terminal operation that summarizes the stream while collecting the result. Collection, collector and collect are different.

(i) Group a list of transactions by currency to obtain the sum of the values of all transactions with that currency

currency is the key of the MAP → Map<Currency, List<Transaction>> transByCurrencies =
transactions.stream().collect(groupingBy
(Transactions::getCurrency));
↓
extracts currency from each transaction

(ii) Partition a list of transactions into two groups: expensive and not expensive
partitionBy takes a predicate

- Collector interface <T, A, R>
  - T → generic type of stream items
  - A → Accumulator type
  - R → type of object resulting from collect operation

Collector applies a transforming fn to the elements – eg in toList() it is the identity transformation, and accumulates the result in a data structure to form the final o/p.

PREDEFINED Collectors can be created from the factory methods provided by the Collectors class. Collector is an interface.

(i) reducing and summarizing stream elements to a single value.
(ii) grouping elements → groupingBy(...)
(iii) partitioning elements – a special case of grouping depending on a Predicate
partitioningBy(...)

(1) Reducing and summarizing – (a) Count the no of dishes in the menu:–

long countingDishes = menu.stream().collect(Collectors.counting());

import static j.u.s.Collectors.*;
so that Collectors.counting() can be replaced by counting()

menu.stream().count();

(b) maxBy(..) and minBy(..) take Comparators as argument

```
Map<Artist, Long> countAlbumsByArtist = tracks.stream()
        .collect(groupingBy(a -> a.getMainMusician(),
                            counting()));
```

partitions albums into buckets

downstream collector collects each bucket to build a Map

Collectors can be combined. Downstream collectors build part of the final result

```
Map<Artist, List<String>> nameOfAlbums = tracks.stream()
        .collect(groupingBy(Album::getMainMusician,
                 mapping(Album::getName, toList())));
```

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

```
enum CaloricLevel { DIET, NORMAL, FAT };
```

Fns passed to stream operations are generally side-effect free and its a good principle to document what side-effects we are willing to accept from fns passed as parameters, and 'none' is the best!

classmate Date 17/02/17 Page

First class fns can be used like other values — can be ~~used~~ passed as argument returned as results, and stored in data structures.

Currying

Any kind of unit conversion can be generalized as
1) Multiply by the conversion factor (f)
2) Adjust the baseline if relevant. (b)

Supplying f and b to convert each value in a similar manner (km → mi would be too tedius.

```
DoubleUnaryOperator convertCtoF = curriedConverter(9.0/5, 32);
"                   convertUSDtoGBP = curriedConverter(0.6, 0);
"                   convertKmtoMi = curriedConverter(0.6214, 0);
double gbp = convertUSDtoGBP.applyAsDouble(1000);
static DoubleUnaryOperator curriedConverter(double f, double d) {
    return (double x) -> x*f + b;
}
```

it contains a method - applyAsDouble

```
double gbp = convertUSDtoGBP.applyAsDouble(1000);
```

Currying is a technique where a function of of 2 arguments (x and y, say) is seen instead as a fn g of one argument that returns a fn also of one argument. The value returned by the latter fn is the same as the value of the original fn, $f(x,y) = (g(x))(y)$.

When some but not all arguments are passed, the fn is said to be partially applied.

```
public void composition() {
    Function<Integer, Integer> add3 = (a) -> a+3;
    Function<Integer, Integer> times2 = (a) -> a*2;   // default method
    Function<Integer, Integer> composedA = add3.compose(times2);
    "                          composedB = times2.compose(add3);
    S.o.p(composedA.apply(3));   // method
    S.o.p(composedB.apply(2));
}
p.s.v. main(...) {
    new Currying().composition();
}
```

add3.compose(a → a*2)

⇒ ~~(a) → a+3 (a) → a*2~~

a → ~~a → a+3~~ a + b*2

(x, y, z) → ✓

x → y → z → ✓

classmate
Date 28/02/17
Page

PARALLEL STREAMS

Sum of first n natural numbers :-

```
public static long seqSum(long n) {
    return Stream.iterate(1L, i -> i+1)
                 .limit(n).parallel()
                 .reduce(0L, Long::sum); }
```

→ difficult to divide and execute in parallel

stream.parallel() → uses ForkJoinPool that by default uses as many threads as there are processors → Runtime.getRuntime().availableProcessors()

```
.filter(...)
.sequential()
.map(...)
.parallel()
.reduce(...);
```

→ Last call wins and is applied to the pipeline globally

```
public static long ffSum(long n) {
    return LongStream.rangeClosed(1,n)
                     .parallel().reduce(0L, Long::sum);
```

→ (i) works with primitive numbers
(ii) produces ranges of numbers that can be split into independent chunks

- parallel execution may not guarantee correct result if applied on fns having side-effects (mutable states).
- e.g. limit(..), findFirst(...) are difficult to run in parallel but findAny() gives better performance.
- Data structures like ArrayList splits more efficiently than LinkedList. Fork/Join framework can be used where the system can decide to recursively fork unless the task is small enough. Finally, all the tasks are joined after computing the results.

Recursion

```
static long factorialRecursive(long n) {
    return n==1 ? 1 : n*factorialRecursive(n-1);
}
```

Tail Recursive (not in Java)

```
static long factorialHelper(long acc, long n) {
    return n==1 ? acc : factorialHelper(acc*n, n-1);
}
```

Stream version (side effect free)

```
static long factorialStreams(long n) {
    return LongStream.rangeClosed(1, n).reduce(1, (long a, long b) -> a*b);
}
```