Java 8 Map-Returning Stream Operations Cheat Sheet

1️⃣ Collectors.groupingBy()

Purpose: Group elements by a classifier function.

```
List<String> fruits = Arrays.asList("apple", "banana", "apricot", "blueberry",
"avocado");
Map<Character, List<String>> grouped = fruits.stream()
        .collect(Collectors.groupingBy(f -> f.charAt(0)));
System.out.println(grouped);
// Output: {a=[apple, apricot, avocado], b=[banana, blueberry]}
```

With counting:

```
Map<Character, Long> countMap = fruits.stream()
        .collect(Collectors.groupingBy(f -> f.charAt(0),
Collectors.counting()));
System.out.println(countMap);
// Output: {a=3, b=2}
```

With mapping:

```
Map<Character, List<String>> upperMap = fruits.stream()
        .collect(Collectors.groupingBy(f -> f.charAt(0),
                Collectors.mapping(String::toUpperCase, Collectors.toList())));
System.out.println(upperMap);
// Output: {a=[APPLE, APRICOT, AVOCADO], b=[BANANA, BLUEBERRY]}
```

2️⃣ Collectors.partitioningBy()

Purpose: Partition elements based on a boolean predicate.

```
List<Integer> nums = Arrays.asList(3, 5, 2, 8, 6);
Map<Boolean, List<Integer>> partitioned = nums.stream()
        .collect(Collectors.partitioningBy(n -> n % 2 == 0));
System.out.println(partitioned);
// Output: {false=[3, 5], true=[2, 8, 6]}
```

With counting:

```
Map<Boolean, Long> countPartition = nums.stream()
        .collect(Collectors.partitioningBy(n -> n % 2 == 0,
Collectors.counting()));
System.out.println(countPartition);
// Output: {false=2, true=3}
```

With mapping (squares):

```
Map<Boolean, List<Integer>> squaredPartition = nums.stream()
        .collect(Collectors.partitioningBy(n -> n % 2 == 0,
                Collectors.mapping(n -> n * n, Collectors.toList())));
System.out.println(squaredPartition);
// Output: {false=[9, 25], true=[4, 64, 36]}
```

3️⃣ Collectors.mapping()

Purpose: Transform elements before collecting.

```
Map<Character, List<String>> upperMap = fruits.stream()
        .collect(Collectors.groupingBy(f -> f.charAt(0),
                Collectors.mapping(String::toUpperCase, Collectors.toList())));
System.out.println(upperMap);
// Output: {a=[APPLE, APRICOT, AVOCADO], b=[BANANA, BLUEBERRY]}
```

4️⃣ Collectors.toMap()

Purpose: Convert stream elements to a Map<K,V>.

```
Map<Character, String> fruitMap = fruits.stream()
        .collect(Collectors.toMap(f -> f.charAt(0), f -> f, (existing,
replacement) -> existing));
System.out.println(fruitMap);
// Output: {a=apple, b=banana}
```

5️⃣ Collectors.counting()

```
Map<Character, Long> countMap = fruits.stream()
        .collect(Collectors.groupingBy(f -> f.charAt(0),
Collectors.counting()));
System.out.println(countMap);
// Output: {a=3, b=2}
```

6️⃣ Collectors.summingInt()/averagingInt()

```
class Product { String category; int price;
    Product(String c, int p){category=c; price=p;}
}
List<Product> products = Arrays.asList(
    new Product("A", 100),
    new Product("B", 200),
    new Product("A", 150)
);

Map<String, Integer> totalByCategory = products.stream()
        .collect(Collectors.groupingBy(p -> p.category, Collectors.summingInt(p
-> p.price)));
System.out.println(totalByCategory);
// Output: {A=250, B=200}

Map<String, Double> avgByCategory = products.stream()
        .collect(Collectors.groupingBy(p -> p.category,
Collectors.averagingInt(p -> p.price)));
System.out.println(avgByCategory);
// Output: {A=125.0, B=200.0}
```

7️⃣ Collectors.maxBy()/minBy()

```
Map<String, Optional<Product>> maxPrice = products.stream()
        .collect(Collectors.groupingBy(p -> p.category,
                Collectors.maxBy(Comparator.comparingInt(p -> p.price))));
System.out.println(maxPrice);
// Output: {A=Optional[Product@...], B=Optional[Product@...]}
```

8️⃣ Summary Table

Collector / Operation	Return Type	Key Type / Value Type	Notes
groupingBy(classifier)	Map<K, List<T>>	K = classifier, V = List<T>	Simple grouping
groupingBy(classifier, downstream)	Map<K, R>	R = downstream result	counting, mapping, etc.
partitioningBy(predicate)	Map<Boolean, List<T>>	Boolean / List<T>	Only 2 keys
partitioningBy(predicate, downstream)	Map<Boolean, R>	R = downstream result	counting, mapping, etc.
mapping(mapper, downstream)	Depends on downstream	-	Transform elements
toMap(keyMapper, valueMapper)	Map<K, V>	-	Throws exception if duplicate key
counting()	Long	-	Counts elements
summingInt()/ summingDouble()	Numeric	-	Sum property
averagingInt()/ averagingDouble()	Double	-	Average property
maxBy()/minBy()	Optional<T>	-	Max/min by comparator