스트림 컬렉터

컬렉터의 장점!

스트림 + 컬렉터를 사용하지 않을시 multilevel로 그룹화를 수행할때 다중 루프를 사용해야해서 코드가 매우 복잡해진다! 다중 루프를 컬렉터를 사용하여 가독성을 크게 향상시킬수있다

명령형:

```
private static void groupImperatively() {
    Map<Currency, List<Transaction>> transactionsByCurrencies = new HashMap<>();
    for (Transaction transaction : transactions) {
      Currency currency = transaction.getCurrency();
      List<Transaction> transactionsForCurrency = transactionsByCurrencies.get(currency);
      if (transactionsForCurrency == null) {
        transactionsForCurrency = new ArrayList<>();
        transactionsByCurrencies.put(currency, transactionsForCurrency);
      }
      transactionsForCurrency.add(transaction);
    }

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

함수형:

```
private static void groupFunctionally() {
    Map<Currency, List<Transaction>> transactionsByCurrencies = transactions.stream()
        .collect(groupingBy(Transaction::getCurrency));
    System.out.println(transactionsByCurrencies);
}
```

그리고 이미 잘 알듯이 함수형 방식의의 또 다른 장점은 높은 수준의 **조합성과 재사용성**을 꼽을 수 있다!

미리 정의된 컬렉터

Collectors에는 미리 정의된 팩토리 메서드가 존재한다고 크게 세 가지로 구분할 수 있다:

- 스트림 요소를 하나의 값으로 리듀싱하고 요약
- 요소 그룹화 (groupingBy)
- 요소 분할 (partitioningBy)

리듀싱과 요약

숫자 카운팅 - **counting()** 활용

메뉴에서 요리 수를 계산하는 코드:

```
private static long howManyDishes() {
    return menu.stream().collect(counting());
}

System.out.println("Nr. of dishes: " + howManyDishes());
```

출력:

```
Nr. of dishes: 9
```

최댓값과 최솟값 검색 (Collectors.**maxBy** & Collectors.**minBy**)

칼로리가 높은 요리를 찾는 코드:

```
Comparator<Dish> dishCaloriesComparator = Comparator.comparing(Dish::getCalories);

Optional<Dish> mostCaloricDish = Dish.menu.stream()
  .collect(Collectors.maxBy(dishCaloriesComparator));
```

요약 연산 (summingInt, summingLong, summingDouble, averagingInt, averagingLong averagingDoubl)

summing 메서드들은 객체를 int로 매핑하는 함수를 인수로 받아서 전부 더한 값을 리턴해준다.

averaging은 평균을 리턴.

아래는 메뉴 리스트의 총 칼로리를 계산하는 코드이다

```
int totalCalories = menu.stream()
                        .collect(summingInt(Dish::getCalories));
```

그리고 count, sum, min, avergae, max 를 한번에 연산해주는 **summarizingInt, summarizingDouble, summarizingLong** 도 제공된다.

```
IntSummaryStatistics menuStatistics = menu.stream()
                                          .collect(summarizingInt(Dish::getCalories));
```

menuStatistics 출력값:

```
IntSummaryStatistics{count=9, sum=4300, min=120, average=477.777778, max=800}
```

문자열 연결 (joining)

문자열들을 연결 시켜주는 메서드, joining 메서드에 구분자를 인수로 줄수있다.

```
String shortMenu = menu.stream()
                       .map(Dish::getName)
                       .collect(joining(", "));
```

그룹화 (Grouping)

groupingBy 메서드는 그룹화하는 기준 함수를 인수로 받는다. 이 기준 함수를 **분류 함수 (classification function)** 라고 부른다.

아래는 메뉴를 타입별로 그룹화하는 코드이다:

```
private static Map<Dish.Type, List<Dish>> groupDishesByType() {
    return menu.stream().collect(groupingBy(Dish::getType));
}
```

그룹화된 요소 조작

요소를 그룹화 한 다음에 각 결과 그룹의 요소를 조작하려면 groupingBy 팩토리 메서드의 두번째 인수에 filtering 메서드를 넘겨주면 된다.

```
private static Map<Dish.Type, List<Dish>> groupCaloricDishesByType() {
    //return menu.stream().filter(dish -> dish.getCalories() > 500).collect(groupingBy(Dish::getType));
    return menu.stream().collect(
        groupingBy(Dish::getType,
            filtering(dish -> dish.getCalories() > 500, toList())));
}
```

다수준 그룹화

외부, 내부 맵을 가진 다수준 그룹화를 하기 원한다면 groupingBy 메서드를 똑같이 외부, 내부로 중첩해서 사용하면 된다.

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

분할 함수

분할 함수는 boolean을 반환하기 때문에 partitioning을 사용하게 되면 리턴되는 맵의 키는 true 또는 false 이다.

아래 코드를 실행시:

```
private static Map<Boolean, List<Dish>> partitionByVegeterian() {
    return menu.stream().collect(partitioningBy(Dish::isVegetarian));
}
```

아래와 같은 맵이 출력된다.

```
{false=[pork, beef, chicken, prawns, salmon], true=[french fries, rice, season fruit, pizza]}
```

이렇게 true, false 두 가지 요소의 스트림 리스트를 모두 유지하고 있기 때문에 .get 을 사용하여 원하는 값만 추출할수 있다.

Collector 인터페이스

- Supplier
  - 작업 결과를 저장할 공간을 제공
- accumulator
  - 스트림의 요소를 수집할 방법을 제공
  - 스트림의 요소들을 supplier 메소드가 제공한 공간에 누적할 방법에 대해 정의
- combiner
  - 두 저장공간을 병합할 방법을 제공 (병렬)
  - 여러 스레드에 의해 처리된 결과를 어떻게 합칠 것인지 정의
  - 반드시 구현해야하는 클래스가 아니다, 스트림이 병렬로 처리될때만 실행된다.
- finisher
  - 결과를 최종적으로 변환할 방법을 제공
  - 변환이 필요 없다면 Function 함수형 인터페이스의 identity 함수를 반환하도록 구현
- characteristics
  - 컬렉터가 수행하는 작업의 속성에 대한 정보를 제공
  - 아래 속성 중 (enum) 해당하는 것을 Set 컬렉션에 담아서 반환
    - UNORDERED : 순서가 유지될 필요가 없다
    - CONCURRENT : 병렬 처리 가능
    - IDENTITY_FINISH : finisher 메소드가 Function.identity를 반환하는 경우

Function.identity

같은 인스턴스를 반환한다.

하지만 미세한 차이가 있는데 아래 예제를 보자.

```
Arrays.asList("a", "b", "c")
          .stream()
          .map(Function.identity()) // <- This,
          .map(str -> str)          // <- is the same as this.
          .collect(Collectors.toMap(
                      Function.identity(), // <-- And this,
                      str -> str));        // <-- is the same as this.
```

map()과 collect()에 써놓은대로 Function.identity() 와 str → str 은 '같은 값' 을 반환한다. 하지만 Function.identity()는 같은 인스턴스를 반환하고 str → str 은 새로운 인스턴스를 생성한다. 이유는 JVM이 모든 lambda expression 에 대해서 싱글톤을 고집하기 때문이다 (모든 lambda expresssion 마다 고유 인스턴스를 갖는다). 하지만 이 이유 때문에 str → str 을 사용하지 않을 이유는 전혀 없다고 생각한다. 이 정도 메모리 손해는 코드 가독성을 위해서 포기해도된다.

Custom Collector Example

```
public class StudentTripletCollector implements
        Collector<Student, List<Triplet<Integer, String, String>>, List<Triplet<Integer, String, String>>> {


    public static StudentTripletCollector toStudentsTriplesList() {
```

```
        return new StudentTripletCollector();
    }

    @Override
    public Supplier<List<Triplet<Integer, String, String>>> supplier() {
        return ArrayList::new;
    }

    @Override
    public BiConsumer<List<Triplet<Integer, String, String>>, Student> accumulator() {
        return (list, student) -> list.add(Triplet.with(student.getYear(), student.getFirstName(), student.getLastName()));
    }

    @Override
    public BinaryOperator<List<Triplet<Integer, String, String>>> combiner() {
        return (list1, list2) -> {
            list1.addAll(list2);
            return list1;
        };
    }

    @Override
    public Function<List<Triplet<Integer, String, String>>, List<Triplet<Integer, String, String>>> finisher() {
        return Collections::unmodifiableList;
    }

    @Override
    public Set<Characteristics> characteristics() {
        return Set.of(Characteristics.UNORDERED);
    }
}
```

Collections::unmodifiableList 가 Function.identity 대신 사용된 이유는 혹시나 이뤄질수도 있는 리스트 복사를 방지하기 위해서다. unmodifiableList() 메소드에서 리턴되는 리스트 reference는 read-only 용도로만 사용 가능하기 때문에 불필요한 복사가 이뤄지지 않는다.