

## 6. 스트림론! 잡아라, collect()!

### 스트림으로 데이터 수집

#### 6.1 Collector란?

##### 기본 개념

```
Map<Currency, List<Transaction>> result =  
    transactions.stream().collect(  
        groupingBy(Transaction::getCurrency)  
    );
```

collect 메소드로 Collectors 인터페이스 구현을 전달한다.

Collector 인터페이스 구현은 스트림의 요소를 어떤 식으로 도출할지 지정한다.

다수준으로 그룹화를 수행할 때 명령형 프로그래밍과 함수형 프로그래밍의 차이점이 두드러진다.

#### 명령형 vs 함수형 프로그래밍

##### 명령형 프로그래밍

- **절차적 기술:** 어떻게 데이터를 그룹화할지 단계별로 기술
- **구현 방식:** 중첩 반복문과 조건문을 통해 그룹화 로직 구현
- **수동 조작:** 컬렉션을 수동적으로 생성하고 조작

##### 장점

- 로직이 명확하고 디버깅이 쉬우며 세세한 제어가 가능하다

##### 단점

- 가독성과 유지보수가 떨어지고 코드 중복이 많고 복잡도가 높다

##### 함수형 프로그래밍

- **선언적 기술:** 무엇을 할지 선언적으로 기술

##### 장점

- 간결하고 선언적이며 의도가 명확하다
- 가독성과 유지보수성이 높다

##### 단점

- 초보자가 읽기 어려울 수 있다

#### Collector vs Collection

## Collector

- 스트림의 결과를 수집하는 전략 객체 (Interface)
- 스트림의 처리 결과를 수집하거나 반환한다
- Stream API와 함께 사용한다
- 데이터를 저장하지 않으며 수집 방법만 정의한다

예시: `Collectors.toList()` , `Collectors.groupingBy()`

## Collection

- 데이터를 저장하는 자료구조 (Interface)
- 데이터를 보관하고 조작한다 (추가, 삭제 등)
- 일반적인 자료구조이다
- 내부에 데이터를 저장한다

예시: `List` , `Set` , `Map`

---

### 6.1.1 고급 리듀싱 기능을 수행하는 컬렉터




Stream에 `collect` 를 호출하면 스트림의 요소에 리듀싱 연산이 수행된다.

명령형 프로그래밍에서는 우리가 직접 구현해야 했던 작업이 자동으로 수행된다.

`Collectors` 인터페이스의 메소드를 어떻게 구현하느냐에 따라 어떤 리듀싱 연산을 수행할지 결정된다.

### 6.1.2 미리 정의된 컬렉터

`Collectors` 클래스에서 제공하는 메소드의 기능은 크게 3가지로 분류할 수 있다:

1.  스트림 요소를 하나의 값으로 리듀스하고 요약
2.  요소 그룹화
3.  요소 분할

---

## 6.2 리듀싱과 요약

스트림 데이터를 하나의 값 또는 통계정보로 줄이거나 요약

```
long howManydishes = menu.stream().collect(Collectors.counting());
```

```
long howManydishes = menu.stream().count();
```

- Stream API의 count()
- 단순히 전체 갯수 세는 용도

: 불필요한 과정 생략이 가능하다

```
Map<String, Long> nameCount = names.stream()  
    .collect(Collectors.groupingBy(  
        Function.identity(),  
        Collectors.counting()  
    ));
```

```
import static java.util.Collectors.counting;
```

```
Map<String, Long> nameCount = names.stream()  
    .collect(Collectors.groupingBy(  
        Function.identity(),  
        counting()  
    ));
```

- Collectors.counting() 같은 의미
- 그룹 별 갯수 세는데 사용

- 전체 요소 카운트: stream().count()
- 그룹별 요소 카운트: Collectors.counting() / counting()

### 🔍 6.2.1 스트림 값에서 최대, 최소 값 검색

Collectors.maxBy, Collectors.minBy 두 개의 메소드를 이용해 계산 가능하다.  
Comparator 를 인수로 받는다.

```
Comparator<Dish> maxComp = Comparator.comparingInt(Dish::getCalories);
Optional<Dish> result = menu.stream().collect(maxBy(maxComp));
```

💡 **Optional:** 만약 menu가 비어있다면 요리가 반환되지 않는다. 따라서 비어있을 수 있으니 Optional을 사용한다.

추가적으로 객체의 숫자 필드의 합계, 평균 등을 반환하는 연산에도 리듀싱 기능이 자주 사용된다. 이를 "요약"이라고 부른다.

## 6.2.2 요약 연산

Collectors 클래스는 Collectors.summingInt 라는 특별한 요약 팩토리 메소드를 지원한다.

### 팩토리 메소드

객체 생성 로직을 캡슐화한 메소드. 생성자 대신 static method나 인스턴스 메소드를 사용해 객체를 생성한다.

summingInt 는 객체를 int로 매핑하는 함수를 인수로 받는다.

summingInt 의 인수로 전달된 함수는 객체를 int로 매핑한 컬렉터로 변환한다.

summingInt 가 collect로 전달되면 요약 작업을 수행한다.

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

summingLong , summingDouble 메소드는 같은 방식으로 동작하며 각각 long 혹은 double 형식의 데이터로 요약한다.

합계 외에 평균값 계산 등의 연산도 요약 기능으로 제공한다.

Collectors.averagingInt , averagingLong , averagingDouble 등으로 다양한 형식을 제공한다.

```
double avgCalories = menu.stream().collect(averagingInt(Dish::getCalories));
```

하지만, 종종 2개 이상의 연산을 한 번에 수행해야 한다면 summarizingInt 를 사용한다.

이는 요소 수, 합계, 평균, 최대/최소 등을 계산해준다.

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

마찬가지로 summarizingLong , summarizingDouble , LongSummaryStatistics , DoubleSummaryStatistics 가 있다.

## 6.2.3 문자열 연결

joining() 팩토리 메소드를 이용하면 스트림의 각 객체에 toString 메소드를 호출해서 추출한 모든 문자열을 하나의 문자열로 연결해서 반환한다.

joining() 메소드는 내부적으로 StringBuilder 를 이용해 문자열을 하나로 만든다.

해당 클래스가 toString() 을 포함하고 있다면 각 문자열을 추출하는 과정을 생략할 수 있다.

## StringBuilder

내부에서 하나의 버퍼에 문자열을 계속 붙인다. 객체를 새로 생성하지 않아 빠르고 가볍다.

```
String shortMenu = menu.stream().collect(joining());
```

하지만 이렇게 작성하면 읽기 어렵게 전부 단순하게 연결된다.

- `joining(",")` 을 사용한다면 ","로 구분해서 보여준다 → 예: pork, beef, chicken
- `joining(", ", "[", "]")` 을 사용할 수 있다 → 예: [pork, beef, chicken]

## Collector와 Collectors의 차이점

- **Collector<T, A, R>** (Interface): Stream을 수집하는데 필요한 동작을 정의
- **Collectors** (Class): 다양한 Collector 구현체를 생성해주는 static 메소드 모음집

Collector는 "수집하는 규칙"을 정의한 인터페이스이고,  
Collectors는 그 인터페이스를 직접 구현하지 않는다.

대신, Collectors는 내부적으로 익명 클래스/람다를 이용해 Collector 객체를 만들어 리턴한다.

## 6.2.4 범용 리듀싱 요약 연산

모든 컬렉터는 `reducing` 팩토리 메소드로 정의 가능하다:

- `count()`, `summingInt()`, `IntSummaryStatistics()`, `joining()`

즉 범용 `Collectors.reducing()` 으로도 구현 가능하다.

예시:

```
int a = menu.stream().collect(
```

3개의 인수를 받는다

`reducing(`

`0, Dish::getCalories, (i,j) -> (i+j));`

스트림 시작 값, 인수 없을 때는 반환 값

정수로 변환할 때 사용하는 함수

두 항목을 하나의 값으로 더하는 `BinaryOperator`

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

## collect와 reduce

### collect

- 도출하려는 결과를 누적 컨테이너를 바꾸도록 설계된 메소드

### reduce

- 두 값을 하나로 도출하는 불변형 연산

람다 표현식 대신 `Integer` 클래스의 `sum` 메소드 참조를 사용할 수 있다.

```
.collect(reducing(
    0, Dish::getCalories, Integer::sum));
```

## 6.3 그룹화

어떤 기준으로 묶어 `Map<key, Collection<value>>` 형태로 재조직한다.

팩토리 메소드인 `Collectors.groupingBy` 를 이용해 쉽게 그룹화할 수 있다.

```
Map<Dish.Type, List<Dish>> dishesByType =
    menu.stream().collect(groupingBy(Dish::getType));
```

출력:

```
{FISH=[prawns, salmon], OTHER=[french fries, rice, pizza], MEAT=[...]}
```

`Dish.Type` 과 일치하는 모든 요리를 추출하는 함수를 `groupingBy` 메소드로 전달한다.

하지만 복잡한 분류 기준이 필요한 상황에서는 메소드 참조를 분류함수로 사용할 수 없다.

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

### 6.3.1 그룹화된 요소 조작

요소를 그룹화한 다음에는 그룹의 요소를 조작하는 연산이 필요하다.

예를 들어 500칼로리를 넘는 요리만 필터링한다고 가정하자.

이때 프레디케이트를 이용해 해결할 수 있다고 생각한다.

```
Map<Dish.Type, List<Dish>> result = menu.stream()
    .filter(dish -> dish.getCalories() > 500)
    .collect(groupingBy(Dish::getType));
```

**✗ 문제점:**

프레디케이트를 만족하는 데이터가 없다면 결과 맵에서 해당 키 자체가 사라진다.

`OTHER=[pizza, ...]`, `MEAT=[pork, ...]` 그럼 `FISH`는 어디에 있을까?

이는 `Collectors` 안에 필터를 이용하여 해결할 수 있다.

```
.collect(groupingBy(
    Dish::getType,
    filtering(d -> d.getCalories() > 500, toList())
));
```

`filtering`: `Collectors` 클래스의 또 다른 정적 팩토리 메소드로 프레디케이트를 인수로 받는다. 각 그룹의 요소와 필터링된 요소를 재그룹화한다.

**결과:** OTHER=[pizza, ...], MEAT=[pork, ...], FISH=[]

추가적으로 매핑함수를 이용해 요소를 변환하는 작업이 있다.

`Collectors` 클래스는 매핑함수와 각 항목에 적용된 함수를 모으는데 사용하는 또 다른 컬렉터를 인수로 받는 `mapping` 메소드를 제공한다.

```
.collect(groupingBy(
    Dish::getType, mapping(Dish::getName, toList())
));
```

결과 맵의 각 그룹이 문자열 리스트라면?

`groupingBy` 와 연계해 세 번째 컬렉터를 사용해 일반 맵이 아닌 `flatMap` 변환을 수행할 수 있다.

```
Map<String, List<String>> dishTags = new HashMap<>();
dishTags.put("pork", asList("greasy", "salty"));
// ...

Map<Dish.Type, Set<String>> result = menu.stream()
    .collect(
        groupingBy(Dish::getType,
            flatMapping(dish -> dishTags.get(dish.getName())
                .stream(), toSet())));
```

두 수준의 리스트를 한 수준으로 평면화: `flatMap` 수행

`flatMap` 연산결과를 수집해서 집합으로 그룹화해 중복 태그를 제거한다.

## 6.3.2 다수준 그룹화

두 인수를 받는 팩토리 메소드 `Collectors.groupingBy` 를 이용해 항목을 다수준으로 그룹화할 수 있다. 일반적인 분류함수와 컬렉터를 인수로 받는다.

`Collectors.groupingBy` 를 중첩해 한 번에 2개(혹은 그 이상)의 기준으로 데이터를 계층적으로 묶는 기법이다.

- **2수준:** `Map<K1, Map<K2, List<T>>>`
- **3수준:** `Map<K1, Map<K2, Map<K3, List<T>>>>`

첫 번째 키 → 두 번째 키 → ... → 요소들로 트리를 타고 내려간다.

```
Map<Dish.Type, Map<CaloricLevel, List<Dish>>> result = menu.stream()
    .collect(
```

```

        groupingBy(Dish::getType,
            groupingBy(dish -> {
                if(dish.getCalories() <= 400) return CaloricLevel.DIET;
                else if(dish.getCalories() <= 700) return CaloricLevel.NORMAL;
                else return CaloricLevel.FAT;
            })
        )
    );

```

결과:

```

MEAT={DIET=[chicken], NORMAL=[beef], FAT=[pork]},
FISH={DIET=...},
OTHER=...

```

n수준 그룹화의 결과는 n수준 트리 구조로 표현되는 n수준 맵이 된다.

### 6.3.3 서브그룹으로 데이터 수집

사실 첫 번째 `groupingBy` 로 넘겨주는 컬렉터의 형식은 제한이 없다.  
예를 들어 두 번째 인수로 `counting` 컬렉터를 전달할 수 있다.

```

Map<Dish.Type, Long> result = menu.stream().collect(
    groupingBy(Dish::getType, counting())
);

```

분류함수 1개의 인수를 갖는 `groupingBy(f)` 는 `groupingBy(f, toList())` 의 축약형이다.  
이로 인해 가장 높은 값을 찾는 것도 구현 가능하다.

```

Map<Dish.Type, Optional<Dish>> result =
    menu.stream().collect(
        groupingBy(
            Dish::getType,
            maxBy(comparingInt(Dish::getCalories))
        )
    );

```

#### Note

`maxBy` 가 생성하는 컬렉터의 결과 형식에 따라 맵의 값이 `Optional` 형식이 되었다.  
실제 메뉴의 요리 중 `Optional.empty()` 를 값으로 갖는 요리는 존재하지 않는다.  
존재하지 않는 요리의 키는 맵에 추가되지 않는다.

`groupingBy` 컬렉터는 첫 번째 요소를 찾은 이후에야 그룹화 맵에 새로운 키를 게으르게 추가한다.  
리듀싱 컬렉터가 반환하는 형식을 사용하는 상황이므로 굳이 `Optional` 래퍼를 사용할 필요가 없다.

### 컬렉터의 결과를 다른 형식에 적용



마지막 그룹화 연산에서 모든 값을 Optional로 감쌀 필요가 없다.

Collectors.collectingAndThen 으로 컬렉터가 결과를 다른 형식으로 활용할 수 있다.

```
Map<Dish.Type, Dish> result =
    menu.stream().collect(
        groupingBy(Dish::getType, // 분류 함수
            collectingAndThen(
                maxBy(comparingInt(Dish::getCalories)), // 감싸인 컬렉터
                Optional::get
            )
        )
    );
```

팩토리 메소드 `collectingAndThen` 은 적용할 컬렉터와 변환함수를 인수로 받아 다른 컬렉터를 반환한다. 반환되는 컬렉터는 기존 컬렉터의 래퍼 역할을 하며 `collect`의 마지막 과정에서 변환함수로 자신이 반환하는 값을 매핑한다.

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

- 첫 번째 인자로 기본 컬렉터 수행
- 그 결과에 대해 함수를 한 번 더 적용 → 수집 후 변환

## groupingBy와 함께 사용하는 다른 컬렉터 예제

일반적으로 스트림에서 같은 그룹으로 분류된 모든 요소에 리듀싱 작업을 수행할 때 `groupingBy` 에 두 번째 인수로 전달한 컬렉터를 사용한다.

```
Map<Dish.Type, Integer> total = menu.stream()
    .collect(
        groupingBy(
            Dish::getType,
            summingInt(Dish::getCalories)
        )
    );
```

이 외에도 `mapping` 메소드로 만들어진 컬렉터도 `groupingBy` 와 자주 사용된다.

```
Map<Dish.Type, Set<CaloricLevel>> result = 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())
        )
    );
```

mapping 메소드에 전달한 변환함수가 Dish를 CaloricLevel로 매핑한다.  
CaloricLevel 결과 스트림은 toSet 컬렉터로 전달되며 집합으로 스트림의 요소를 누적한다.

## 6.4 분할

분할은 **분할함수**라 불리는 프레디케이트를 분류함수로 사용하는 특수한 그룹화 기능이다.  
불리언을 반환하므로 맵의 키 형식은 Boolean이다.

**예시:** 채식과 채식이 아닌 요리 구분

```
Map<Boolean, List<Dish>> result = menu.stream()
    .collect(partitioningBy(Dish::isVegetarian));

List<Dish> vegetarianDishes = result.get(true); // 참 값의 키로 데이터 얻기
```

물론 이전에 사용한 프레디케이트로 필터링한 다음 결과를 얻을 수 있다.

### 6.4.1 분할의 장점

참, 거짓 두 가지 요소의 스트림 리스트를 모두 유지한다는 것이 분할의 장점이다.  
이전에 거짓 키를 이용해서 채식이 아닌 데이터를 얻을 수 있었다.  
다음 예제처럼 컬렉터를 두 번째 인수로 전달할 수 있는 오버로드된 버전의 partitioningBy 메소드도 있다.

```
Map<Boolean, Map<Dish.Type, List<Dish>>> result = menu.stream()
    .collect(
        partitioningBy(Dish::isVegetarian,
            groupingBy(Dish::getType)
        )
    );
```

또한 각각의 그룹에서 가장 칼로리가 높은 것도 찾을 수 있다.

```
Map<Boolean, Dish> result = menu.stream()
    .collect(
        partitioningBy(Dish::isVegetarian,
            collectingAndThen(
                maxBy(comparingInt(Dish::getCalories)),
                Optional::get
            )
        )
    );
```

## 6.5 Collector 인터페이스

다시 한번 Collector와 Collectors를 정리해보자.

## Collector vs Collectors 정리

### Collector

- `java.util.stream.Collectors<T, A, R>` 인터페이스
- 스트림을 어떻게 접어서 최종 결과를 만들지에 대한 계약
- 타입 매개변수: T(입력요소), A(중간 누산기), R(최종결과)
- 구성요소: supplier, accumulator, combiner, finisher, characteristics

### Collectors

- `java.util.stream.Collectors` 정적 메소드 모음
- `toList()`, `toSet()`, `joining()`, `groupingBy()`, `partitioningBy()`, ...
- 보통 `import static java.util.stream.Collectors.*;`

Collector 인터페이스는 리듀싱 연산(즉, 컬렉터)을 어떻게 구현할지 제공하는 메소드 집합으로 구성된다.

```
public interface Collector<T, A, R> {  
    Supplier<A> supplier();  
    BiConsumer<A, T> accumulator();  
    Function<A, R> finisher();  
    BinaryOperator<A> combiner();  
    Set<Characteristics> characteristics();  
}
```

- **T**: 스트림 원소 타입
- **A**: 중간 누적 컨테이너
- **R**: 최종 결과 타입

예를 들어 `Stream<T>` 의 모든 요소를 `List<T>` 로 수집하는 `ToListCollector<T>` 라는 클래스를 구현할 수 있다.

```
public class ToListCollector<T> implements Collector<T, List<T>, List<T>>
```

스트림 → 가변 컨테이너에 모으면서 → 최종 결과를 반환하는 리듀싱 과정을 캡슐화한 전략

## 6.5.1 Collector 인터페이스의 메소드 살펴보기

### 1 Supplier: 새로운 결과 컨테이너 만들기

빈 결과로 이루어진 Supplier를 반환해야 한다.

즉, Supplier는 수집과정에서 빈 누적자 인스턴스를 만드는 파라미터가 없는 함수이다.

누적자를 반환하는 컬렉터에서는 빈 누적자가 비어있는 스트림의 수집 과정의 결과가 될 수 있다.

```
public Supplier<List<T>> supplier() {  
    return () -> new ArrayList<T>();  
}
```

```

}

// 또는
public Supplier<List<T>> supplier() {
    return ArrayList::new;
}

```

- 새로운 가변 컨테이너
- 스트림 파이프라인이 시작될 때 스레드마다 1번씩

### 가변 컨테이너

리듀싱 과정에서 중간 상태를 업데이트해야 하는 객체

예: List → 원소 추가, Map → put, ...

## 2 Accumulator: 결과 컨테이너에 요소 추가하기

리듀싱 연산을 수행하는 함수를 반환한다.

Stream에서 n번째 요소를 탐색할 때 두 인수, 즉 누적자와 n번째 요소를 함수에 적용한다.

함수의 반환값은 void이며, 요소를 탐색하면서 적용하는 함수에 의해 누적자 내부 상태가 바뀌므로 누적자가 어떤 값일지 단정할 수 없다.

```

public BiConsumer<List<T>, T> accumulator() {
    return (list, item) -> list.add(item); // return List::add와 같다
}

```

여기서 List<T>를 A라고 보자.

요소 T를 컨테이너 A에 추가하는 동작으로 각 요소를 만날 때마다 호출한다.

## 3 Finisher: 최종 반환값을 결과 컨테이너로 적용

스트림 탐색을 끝내고 누적자 객체를 최종 결과로 변환하면서 누적 과정을 끝낼 때 호출할 함수를 반환한다.

```

public Function<List<T>, List<T>> finisher() {
    return Function.identity();
}

```

- List<T> ⇒ A
- List<T> ⇒ R

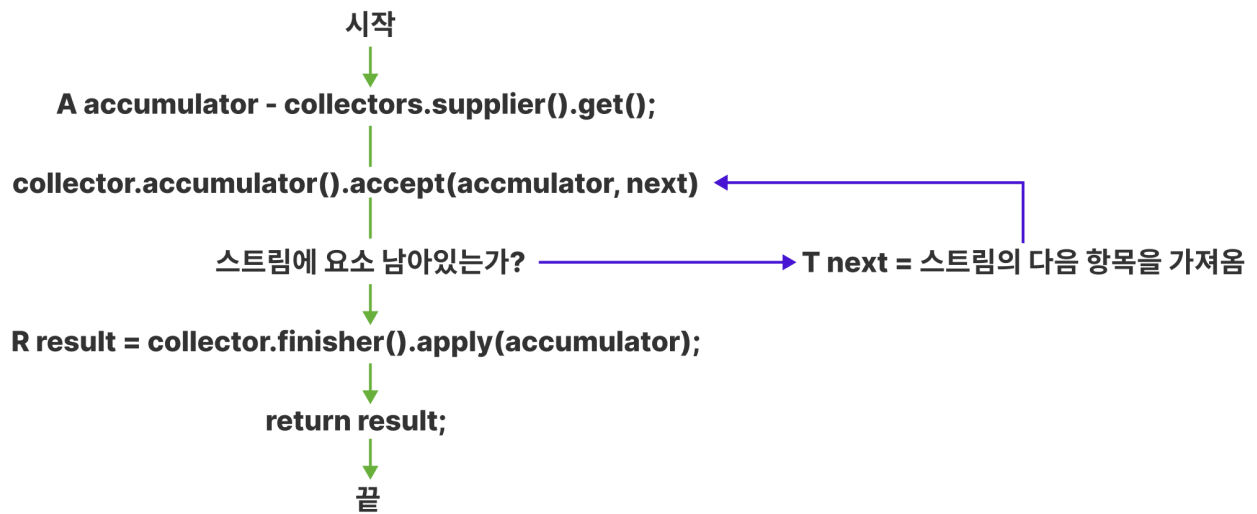
컨테이너 A → 최종결과 R로 반환

모든 누적/병합이 끝난 마지막 한 번

```

Supplier  —————> Accumulator  —————> Finisher
: 초기 누적 컨테이너 생성      : Stream 요소 T를 A에 누적      : 누적된 A를 최종 결과 R로
변환

```



#### 4 Combiner: 두 결과 컨테이너 병합

서로 다른 서브파트를 병렬로 처리할 때 누적자가 이 결과를 어떻게 처리할지 정의한다.

```

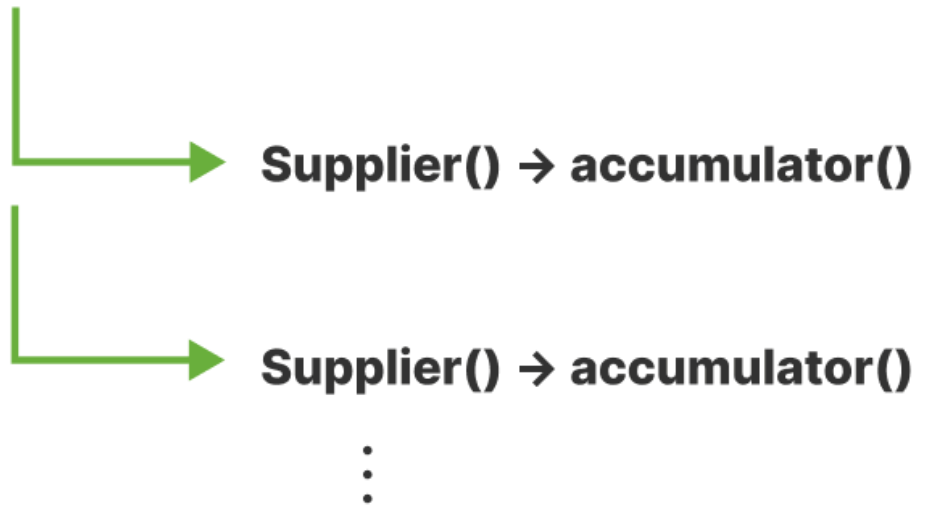
public BinaryOperator<List<T>> combiner() {
    return (list1, list2) -> {
        list1.addAll(list2);
        return list1;
    };
}

```

`List<T> ⇒ A`

- 두 컨테이너 A를 하나로 합치는 연산(결합법칙 필요)
- 주로 병렬 수집에서 서브 결과를 합칠 때 사용
- 병렬 처리로 쪼개 모은 결과를 하나로 합쳐야 최종 결과

# Stream<T>



**combiner() 병합**

**finisher() 최종 결과**

위를 이용하면 스트림의 리듀싱을 병렬로 수행할 수 있다.

스트림의 리듀싱을 병렬로 수행할 때 "포크/조인 프레임워크"와 Spliterator를 사용한다.

## 5 Characteristics: 동작 특성들을 Set 형태로 변환

컬렉터의 연산을 정의하는 Characteristics 형식의 불변 집합을 반환한다.

스트림을 병렬로 리듀싱할 것인지 그리고 병렬로 리듀스한다면 어떤 최적화를 선택해야 할지 힌트를 제공한다.

### 🔴 UNORDERED

리듀싱 결과는 스트림 요소의 방문 순서나 누적 순서에 영향받지 않는다.

### 🔴 CONCURRENT

다중 스레드에서 accumulator 함수를 동시에 호출할 수 있다. 이 컬렉터는 스트림의 병렬 리듀싱을 수행할 수 있다.

컬렉터의 플래그에 UNORDERED를 함께 설정하지 않았다면 데이터 소스가 정렬되어있지 않은 상황에서만 병렬 리듀싱을 수행할 수 있다.

## IDENTITY\_FINISH

finisher 메소드가 반환하는 함수는 단순히 identity를 적용할 뿐이므로 이를 생략할 수 있다.