Terminal Operation

sanghyuck.na@lge.com

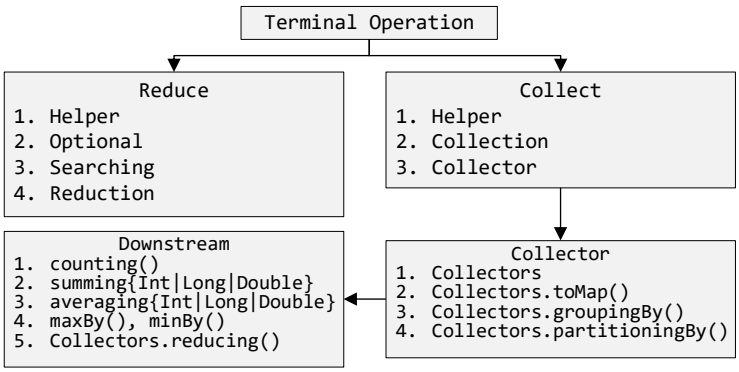
Terminal Operations
Reduce operations
Special Reduction
Optional
Stream Search

```
IntStream newIntStream() {return IntStream.of(1, 2, 3);};
Stream<Integer> newIntegerStream() {return newIntStream().boxed();};
```

1

Terminal operations

- 스트림 상태를 변화시키는 연산자
  - Autoclosable: 한번 실행된 Stream은 더 이상 재사용 될 수 없음. 동일한 데이터를 재 접근해야 한다면, 새로 만들어서 사용
  - Synchronized: pipeline의 데이터 순회와 Intermediate OP부터 Terminal OP처리를 모두 마쳐야 결과 반환
  - Spliterator: spliterator()로 순회방법에 따라 순회순서가 결정되어 Terminal OP에 전달
  - Terminal OP 분류: 동작방식에 따라 분류됨. 그 결과에 따라 분류하지 않음



2

Reduce operations(fold)

- 데이터 집계 연산자
  - 연속된 데이터를 결합 연산자를 반복해서 적용함으로 써 단일 객체로 결합
  - reduce(): Scalar(합계, 평균, 최대값, 최소값)
  - collect(): Collection
- Reduction helper 연산자
  - count(), max(), min()
  - sum(), avg()

```
int numbers[] = { 1, 2, 3 };
int sum = 0;
for (int x : numbers) {
    sum += x;
}
```

```
IntStream numbers = newIntStream();
int sum = numbers.sum();
```

```
IntStream numbers = newIntStream();
int sum = numbers.reduce(0, Integer::sum);
```

```
IntStream numbers = newIntStream();
int sum = numbers.reduce(0, (x, y) -> x + y);
```

3

Reduction Helper

- Stream
  - long count()
  - Optional<T> max(Comparator<? super T> comparator)
  - Optional<T> min(Comparator<? super T> comparator)
- Primitive Stream
  - OptionalInt min(), OptionalInt max()
  - int sum(), OptionalDouble average()

```
long cnts = newIntegerStream().count();
Optional<Integer> mins = newIntegerStream().min(Comparator.naturalOrder());
Optional<Integer> maxs = newIntegerStream().max(Comparator.reverseOrder());

long cntp = newIntStream().count();
OptionalInt minp = newIntStream().min();
OptionalInt maxp = newIntStream().max();
int sump = newIntStream().sum();
OptionalDouble avgp = newIntStream().average();
```

4

Optional[8]

- Nullable Object를 대표하는 객체
  - 가장 빈번하게 발생link하는 NullPointException 해결안으로 제안
  - 만약 Null이 아니라면 isPresent()와 get()는 value 반환, 그렇지 않은 경우 orElse() 사용
  - Value-based class로 비교 (연산자==) 은 값에 따라 결정. 예측할 수 없는 결과 발생가능
- Condition
  - boolean isPresent(), void ifPresent(consumer), void ifPresentOrElse()
- Stream
  - filter(), map(), flatMap()

```
Optional<Integer> mins
    = newIntegerStream().min(Comparator.naturalOrder());

if (mins.isPresent())
    System.out.println(mins.get());
mins.ifPresent(System.out::println);
mins.ifPresentOrElse(System.out::println, System.out::println);

mins.filter(m -> m > 0).map(String::valueOf)
    .ifPresent(System.out::println);
```

5

Optional

- Else
  - Optional<T> **or**(supplier)
  - T orElse(T other), T orElseGet(supplier), T orElseThrow(exceptionSupplier)
- Factory
  - Optional<T> of(value), Optional<T> ofNullable(value), Optional<T> empty()

```
Optional<Integer> mins = newIntegerStream().min(Comparator.naturalOrder());

Optional<Integer> else1 = mins.or(() -> Optional.of(-1));
Integer else2 = mins.orElse(-1);
Integer else3 = mins.orElseGet(() -> -1);
Integer else4 = mins.orElseThrow();
Integer else5 = mins.orElseThrow(NullPointerException::new);

Optional<String> str = Optional.of("me");
Optional<String> str2 = Optional.empty();
Optional<String> str3 = Optional.ofNullable(null);
```

6

Stream Search

- 주어진 조건에 맞는 데이터를 검색하여 반환 Find
  - Optional<T> findAny(), Optional<T> findFirst()
- 주어진 조건에 맞는 데이터존재 유무 확인 Match
  - Stream에서 아이템이 주어진 predicate과 일치하는지 체크(Boolean)
  - boolean anyMatch(Predicate<? super T> predicate)
  - boolean allMatch(Predicate<? super T> predicate)
  - boolean noneMatch(Predicate<? super T> predicate)

```
newIntStream().findFirst().ifPresent(System.out::println);
newIntStream().findAny().ifPresent(System.out::println);

boolean r = newIntStream().anyMatch(i -> i == 1);
boolean r2 = newIntStream().allMatch(i -> i == 1);
boolean r3 = newIntStream().noneMatch(i -> i == 1);
```

7

도전하세요!

- 다음 주어진 Stream 에서 >0.2인 평균을 구하고, 화면에 출력하세요.
  - 단, 데이터가 존재하지 않는 경우 "None" 메시지를 화면에 출력하세요.

```
Random r = new Random();
DoubleStream ds = DoubleStream.generate(r::nextDouble)
    .limit(100);

DoubleStream ds2 = DoubleStream.generate(r::nextDouble)
    .limit(0);
```

- 다음 주어진 Stream<Integer> idata에서 최대값을 구하세요
  - 만약 그 결과 값이 > 23이라면 그 값에 *100을 하여 화면에 출력하세요

```
Random r = new Random();
Stream<Integer> s =
       IntStream.generate(r::nextInt).limit(100).boxed();
```

8

도전하세요!

- 다음 주어진 요구사항에 맞는 메소드 개발
  - Method Signature: OptionalDouble divide(double d)
  - Return value : 1 / d 의 연산 결과
  - 특이사항: d == 0 이라면 OptionalDouble.empty() 반환

```
divide(4).ifPresent(System.out::println);
```

9

정리

- Reduce Helper
  - count() sum() average() min() max()
- Optional
  - Null handling
- Searching
  - Finding
  - Matching

10

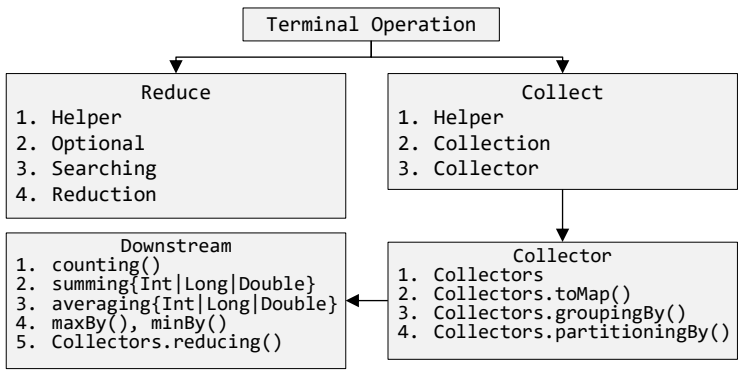
Reduce

sanghyuck.na@lge.com

```
IntStream newIntStream() {return IntStream.of(1, 2, 3);};
Stream<Integer> newStream() {return newIntStream().boxed();};
```

11

목차



12

Reduce

- 결과가 입력데이터와 동일한 타입 집계연산자
  - reduce() 3가지 연산자가 있으며 입력과 동일타입 결과값으로 귀결된 집계
- 연산자 파라미터
  - identity: 항등원, 초기값, 기본값, 데이터가 없는 경우 대표 값
  - accumulator: 입력파라미터는 중간결과값과 새로운 값으로 구성 된 연산자
  - combiner: 입력파라미터는 중간결과값 2개로 구성된 연산자로 집계결과 반환

```
Optional<T> reduce(BinaryOperator<T> accumulator)
T reduce(T identity, BinaryOperator<T> accumulator)
U reduce(U identity, BiFunction<U,? super T,U> accumulator,
       BinaryOperator<U> combiner)

interface BinaryOperator<T> extends BiFunction<T,T,T> {
    T apply(T t, T u);
}
```

13

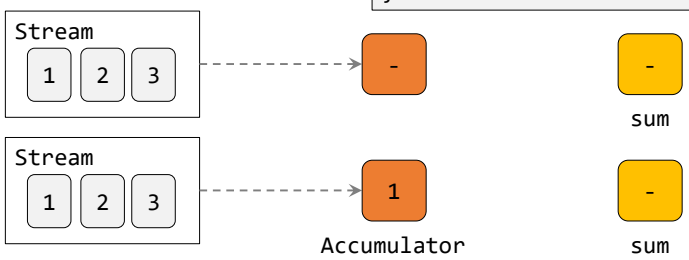
Operator Design

- accumulator

```
Stream<Integer> numbers = newStream();

Optional<Integer> sum = numbers.reduce((x, y) -> x + y);
```

```
int apply(int x, int y) {
     return x + y;
}
```

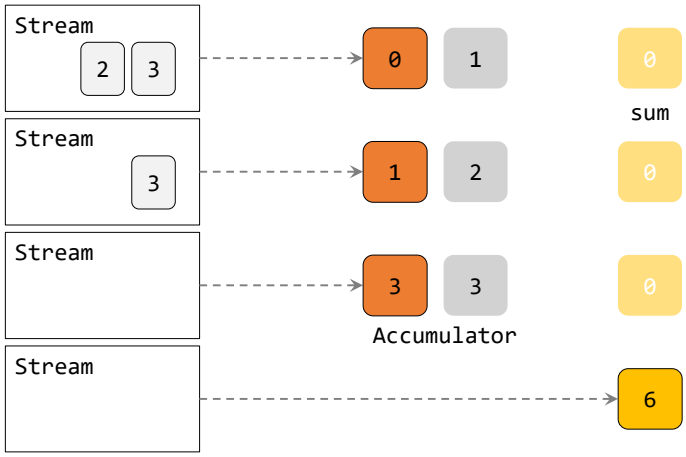


14

Operator Design

- identity, accumulator

```
Stream<Integer> numbers = newStream();
Integer sum = numbers.reduce(0, (x, y) -> x + y);
```



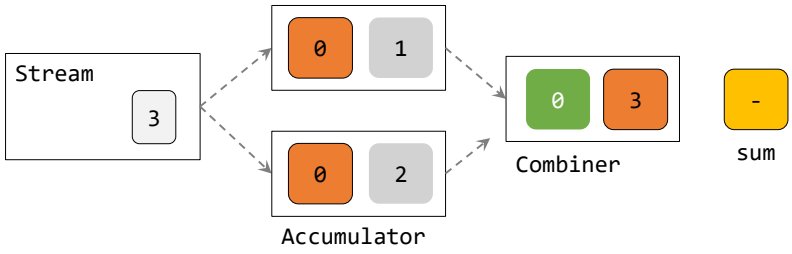
15

Operator Design

- combiner: 중간결과값 2개를 집계하여 결과값 반환

```
<U> U  reduce(U identity
              , BiFunction<U,? super T,U> accumulator
              , BinaryOperator<U> combiner)
combiner.apply(u, accumulator.apply(identity, t))
    == accumulator.apply(u, t)
```

```
Integer sum = newStream().reduce(0, (x, y) -> x + y ,
                                    (x, y) -> x + y);
```



16

도전하세요!

- 다음 min()함수를 reduce로 바꾸세요

```
IntStream numbers = newIntStream();
OptionalInt min = numbers.min();
min.ifPresent(System.out::println);
```

```
Stream<Integer> numbers = newStream();
Optional<Integer> min = ?
```

17

도전하세요!

- 다음 합계 sum()을 reduce로 바꾸세요

```
IntStream numbers = newIntStream();
int sum = numbers.sum();
System.out.println(sum);
```

```
IntStream numbers = newIntStream();
Integer sum = ?
System.out.println(sum);
```

18

도전하세요!

- 메모리 크기는 MemoryPoolXBean으로 구할 수 있다.
  다음 코드의 getUsed()로 총 사용 메모리 크기(sum)를 구하세요

```
List<MemoryPoolMXBean> beans
        = ManagementFactory.getMemoryPoolMXBeans();
long sum = 0L;

for (MemoryPoolMXBean bean : beans) {
    MemoryUsage usage = bean.getUsage();
    long youngUsedMemory = usage.getUsed();
    sum += youngUsedMemory;
}
System.out.println(sum);
```

```
List<MemoryPoolMXBean> beans
        = ManagementFactory.getMemoryPoolMXBeans();
long sum = ?
System.out.println(sum);
```

19

정리

- Reduce
  - Optional<T> reduce(BinaryOperator<T> accumulator)
  - T reduce(T identity, BinaryOperator<T> accumulator)
  - <U> U reduce(U identity,
    BiFunction<U,? super T,U> accumulator,
    BinaryOperator<U> combiner)
- Identity, Accumulator, Combiner

20

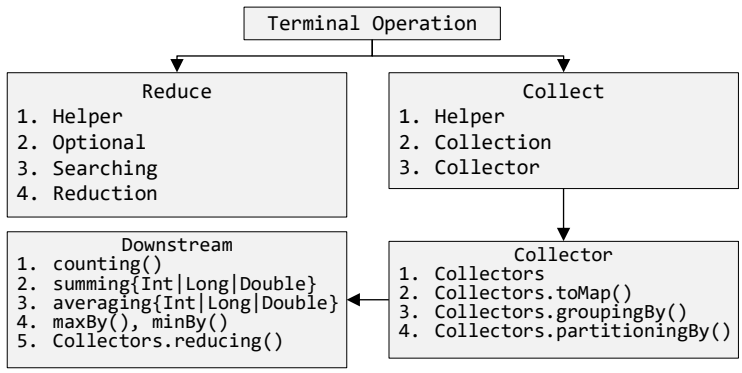
Collect

sanghyuck.na@lge.com

```
IntStream newIntStream() {return IntStream.of(1, 2, 3);};
Stream<Integer> newIntegerStream() {return newIntStream().boxed();};
```

21

목차



22

collect

- 입력 타입과 다른 타입으로 집계하는 연산자
  - 새로운 타입의 집계결과값 생성하는 연산자
  - 주로 입력데이터는 단일 Scalar값이고 결과는 Collection, Array
- 연산자 파라미터
  - supplier: reduce의 identity와 동일하지만 그 차이점은 입력타입과 다를 수 있음
  - accumulator, combiner: reduce 연산자와 동일

```
<R> R collect(Supplier<R> supplier,
       BiConsumer<R,? super T> accumulator,
       BiConsumer<R,R> combiner)

<R,A> R collect(Collector<? super T,A,R> collector)
```

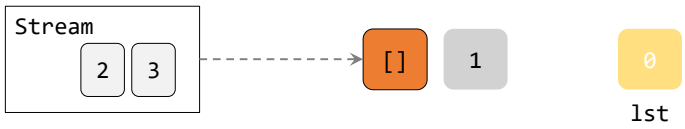
23

Operator Design

- supplier, accumulator

```
<R> R collect(Supplier<R> supplier,
       BiConsumer<R,? super T> accumulator,
       BiConsumer<R,R> combiner)

Stream<Integer> numbers = newIntegerStream();
List<Integer> lst = numbers.collect(
    () -> new ArrayList<Integer>(),
    (l, e) -> lp.add(e),
    (l, r) -> l.addAll(r));
```



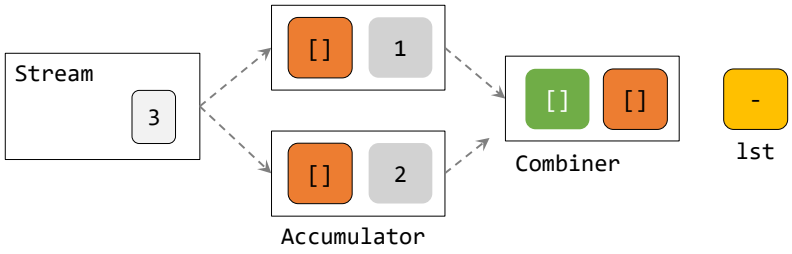
24

Operator Design

- combiner

```
<R> R collect(Supplier<R> supplier,
        BiConsumer<R,? super T> accumulator,
        BiConsumer<R,R> combiner)

Stream<Integer> numbers = newStream();
List<Integer> lst = numbers.collect(
    () -> new ArrayList<Integer>(),
    (tmp, e) -> tmp.add(e),
    (l, r) -> l.addAll(r));
```



25

도전하세요!

- 다음 출력결과를 참고하여 스트림을 HashMap으로 변형하세요

```
Stream<Integer> numbers = newIntegerStream();
HashMap<Integer, String> map = ?;
System.out.println(map);
```

```
{1:"my-value=1", 2:"my-value=2", 3:"my-value=3"}
```

26

Collector

- 입력값(supplier, accumulator, combiner) 3개를 캡슐화 한 클래스
  - T: Reduction연산의 입력데이터 타입
  - A: 수정가능한 수집 mutable accumulation 타입
  - R: Reduction연산의 결과데이터타입

```
<R,A> R collect(Collector<? super T,A,R> collector)

public interface Collector<T,A,R> {
    Supplier<A> supplier()
    BiConsumer<A, T> accumulator()
    BinaryOperator<A> combiner()

    Function<A, R> finisher()
    Set<Characteristics> characteristics()
}
```

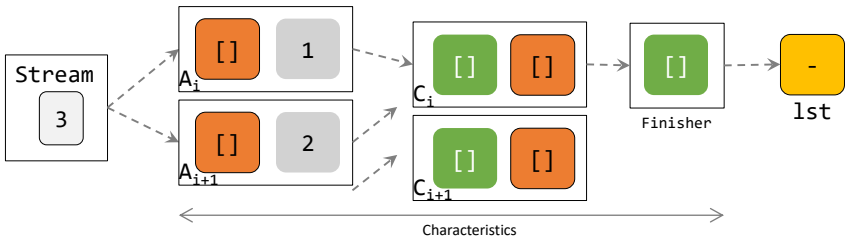
```
참고: <R> R collect(Supplier<R> supplier,
                    BiConsumer<R,? super T> accumulator,
                    BiConsumer<R,R> combiner)
```

27

Collector

- Supplier<A> supplier(): 수정가능한 새로운 컨테이너를 생성하여 반환
- BiConsumer<A, T> accumulator(): 단일 값과 중간결과를 집계
- BinaryOperator<A> combiner(): 두개 중간결과값을 집계
- Function<A, R> finisher(): 중간결과타입을 최종결과타입으로 변형하는 함수
- Set<Characteristics> characteristics() Collector동작 제어

```
Stream<Integer> numbers = newIntegerStream();
List<Integer> lst = numbers.collect(
    new Collector<Integer, List<Integer>, List<Integer>>(){});
```



28

Collector

- CONCURRENT
  - 다중쓰레드(Multithread)는 Accumulator함수에서 동일 중간결과컨테이너를 동시에 접근하고 사용하도록(Shared workspace) 동작
  - Accumulator, Multithread, Shared workspace
- UNORDERED
  - 컬렉션연산은 입력데이터의 데이터순서를 출력에서 보장하지 않음
  - Associative, Commutative
- IDENTITY_FINISH
  - finisher 함수는 항등 함수로 이 옵션은 finisher() 호출이 생략될 수 있음
  - Finisher, identity function

```
Collections.unmodifiableSet(EnumSet.of(
                Characteristics.CONCURRENT,
                Characteristics.IDENTITY_FINISH,
                Characteristics.UNORDERED));
```

29

Collector

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

@Override public BiConsumer<List<Integer>, Integer> accumulator() {
    return (lst, item) -> lst.add(item);
}

@Override public BinaryOperator<List<Integer>> combiner() {
    return (llst, rlst) -> { llst.addAll(rlst); return llst; };
}

@Override public Function<List<Integer>, List<Integer>> finisher() {
    return Function.identity();
}

@Override public Set<Characteristics> characteristics() {
    return Collections.unmodifiableSet(
        EnumSet.of(Characteristics.CONCURRENT,
        Characteristics.IDENTITY_FINISH,Characteristics.UNORDERED));
}
```

30

Collector

- Collector.of()
  - 주어진 supplier, accumulator, combiner함수를 반환하는 새로운 Collector 반환
  - 기본함수는 IDENTITY_FINISH 포함

```
static <T,R> Collector<T,R,R>    of(Supplier<R> supplier,
    BiConsumer<R,T> accumulator, BinaryOperator<R> combiner,
    Collector.Characteristics... characteristics)

static <T,A,R> Collector<T,A,R> of(Supplier<A> supplier,
    BiConsumer<A,T> accumulator, BinaryOperator<A> combiner,
    Function<A,R> finisher,
    Collector.Characteristics... characteristics)
```

```
ArrayList<Integer> lst = numbers.collect(
  Collector.of(ArrayList::new, ArrayList::add,
      (llst, rlst) -> {llst.addAll(rlst);return llst;},
      Characteristics.CONCURRENT,Characteristics.IDENTITY_FINISH,
        Characteristics.UNORDERED));
```

31

Collectors.*

- static Collector factory
  - 자주 사용하는 Reduction Operation 구현

```
<T> Collector<T,?,List<T>> toList();
<T,C extends Collection<T>> Collector<T,?,C> toCollection
    (Supplier<C> collectionFactory);
Collector<CharSequence,?,String> joining();
<T> Collector<T,?,Double> averagingDouble
    (ToDoubleFunction<? super T> mapper);
<T> Collector<T,?,IntSummaryStatistics> summarizingInt
    (ToIntFunction<? super T> mapper);
```

```
var lst = newIntegerStream().collect(Collectors.toList());
var set = newIntegerStream()
        .collect(Collectors.toCollection(TreeSet::new));
var joi = newIntegerStream().map(String::valueOf)
        .collect(Collectors.joining());
int total = newIntegerStream()
        .collect(Collectors.summingInt(Integer::intValue));
```

34

Collectors.summarizingInt

- 단순 데이터통계함수
  - long getCount(), double getAverae(), long/double getSum()
  - int/long/double getMax(), int/long/double getMin()

```
IntSummaryStatistics stat = newIntegerStream().collect(
        Collectors.summarizingInt(Integer::intValue));

IntSummaryStatistics stats = newIntStream().collect(
        IntSummaryStatistics::new,
        IntSummaryStatistics::accept,
        IntSummaryStatistics::combine);

stats.getAverage();
```

https://goo.gl/iUjbX8 LongSummaryStatistics
https://goo.gl/YhTn8B DoubleSummaryStatistics

35

도전하세요!

- 고객 이름을 단일 문자열로 합치세요 seq: Collectors.joining()를 사용하세요
- 고객 포인트의 총합을 구하세요 sum: Collectors.summingInt()를 사용하세요

```
class Customer {
    String name; int points;
    Customer(String name, int points) {
        this.name = name; this.points = points;
    }
    int getPoints() { return this.points; }
    String getName() { return this.name; }
}
List<Customer> customers = List.of(new Customer("John P.", 15),
    new Customer("Sarah M.", 200),
    new Customer("Charles B.", 150), new Customer("Mary T.", 1));

String seq = ?;
Integer sum= ?;
```

```
John P.Sarah M.Charles B.Mary T.
366
```

36

도전하세요!

- Memory Pool로 현재 시스템의 메모리사용량에 대해 총 개수, 평균, 총합, 최대/최소 값을 구하세요
  - 단 결과 값의 타입은 double로 합니다

```
List<MemoryPoolMXBean> beans
        = ManagementFactory.getMemoryPoolMXBeans();

MemoryUsage usage = beans.get(0).getUsage();
long youngUsedMemory = usage.getUsed();
```

```
count=8, sum=34289560.000000, min=0.000000,
average=4286195.000000, max=22020096.000000}
```

39

도전하세요!

- 다음 평균 average()함수를 collect로 바꾸세요
  - 난이도 : 상

```
IntStream numbers = newIntStream();
OptionalDouble avg = numbers.average();
avg.ifPresent(System.out::println);
```

```
Stream<Integer> numbers = newIntegerStream();
OptionalDouble od = numbers.collect(
    ???
);

od.ifPresent(System.out::println);
```

40

정리

- Collect
  - <R> R collect(Supplier<R> supplier,
    - BiConsumer<R,? super T> accumulator,
    - BiConsumer<R,R> combiner)
  - <R,A> R collect(Collector<? super T,A,R> collector)
- Collector
  - supplier, accumulator, combiner, finisher, Characteristics
  - of()
- Collectors
  - <T> Collector<T,?,List<T>> toList()<T,C extends Collection<T>>
  - Collector<T,?,C>toCollection(Supplier<C> collectionFactory)
  - Collector<CharSequence,?,String> joining()
  - <T> Collector<T,?,Double> averagingDouble(ToDoubleFunction<? super T> mapper)
  - <T> Collector<T,?,IntSummaryStatistics> summarizingInt(ToIntFunction<? super T> mapper)
- 남은 것
  - minBy(), maxBy(), counting(), collectingAndThen() teeing()

41

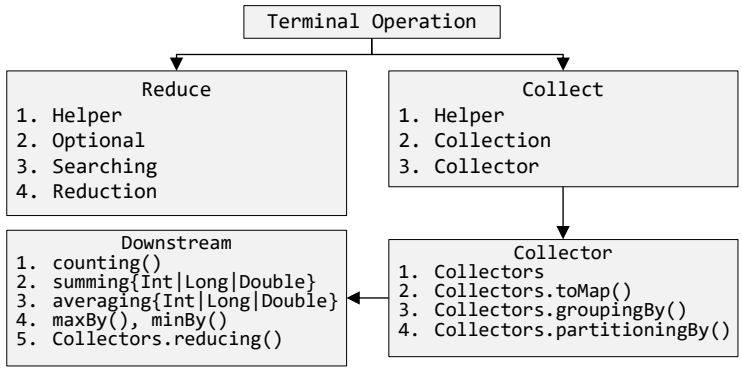
Collectors.toMap

sanghyuck.na@lge.com

```
static class Employee {String name;int score;public String getName() { return
name; }public int getScore() { return score; }public Employee(String name, int
score) {this.name = name;this.score = score;}static Employee of(String name, int
score) {return new Employee(name, score);}} Stream<Employee> newEmployeeStream()
{return Stream.of(Employee.of("google", 40),Employee.of("yahooe",
20),Employee.of("naver", 30));}
```

42

목차

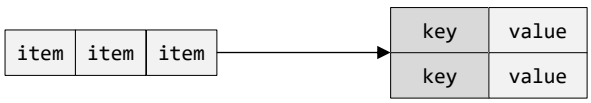


43

Collectors.toMap

- 맵으로 집계하는 Collector 생성
  - 주어진 맵핑함수를 각 입력데이터에 적용하여 Key와 Value로 구성된 Map으로 집계하는 Collector 반환

```
<T,K,U> Collector<T,?,Map<K,U>>
        toMap(Function<? super T,? extends K> keyMapper
            , Function<? super T,? extends U> valueMapper)
<T,K,U> Collector<T,?,Map<K,U>>
        toMap(Function<? super T,? extends K> keyMapper
            , Function<? super T,? extends U> valueMapper
            , BinaryOperator<U> mergeFunction)
<T,K,U,M extends Map<K,U>> Collector<T,?,M>
        toMap(Function<? super T,? extends K> keyMapper
            , Function<? super T,? extends U> valueMapper
            , BinaryOperator<U> mergeFunction
            , Supplier<M> mapSupplier)
```



44

Collectors.toMap

- keyMapper, valueMapper
  - key 와 Value 맵핑 함수 지정
  - Key 중복발생 시 IlegalsStateException발생하고 Type, Mutability, Serializability, Thread-Safe를 보장하지 않은 Map 반환

```
Collector<T,?,Map<K,U>> toMap(
Function<? super T,? extends K> keyMapper,
Function<? super T,? extends U> valueMapper)
```

```
Map<Employee, Integer> em = newEmployeeStream()
    .collect(Collectors.toMap(Function.identity(),
    Employee::getScore));
```

Employee
<google, 40>
<yahoo, 20>
<naver, 30>

key Mapper

value Mapper

Employee	Score
<google, 40>	40
<yahoo, 20>	20
<naver, 30>	30

45

Collectors.toMap

- mergeFunction
  - Key 중복발생 시 mergeFunction를 호출하여 value 2개 중에 선택

```
<T,K,U> Collector<T,?,Map<K,U>> toMap(
    Function<? super T,? extends K> keyMapper,
    Function<? super T,? extends U> valueMapper,
    BinaryOperator<U> mergeFunction)
```

```
Map<String, Integer> em = newEmployeeStream2()
    .collect(Collectors.toMap(Employee::getName,
        Employee::getScore, (v1, v2) -> v1 + v2));
```

Employee
<google, 40>
<yahoo, 20>
<naver, 30>
<naver, 20>

key Mapper

value Mapper

merge Function

Name	Score
google	40
yahoo	20
naver	**50**

```
Stream<Employee> newEmployeeStream2() {return Stream.of(Employee.of("google", 40),Employee.of("yahooe", 20),Employee.of("naver", 30),Employee.of("naver", 20));}
```

47

도전하세요!

- Key 충돌 발생 시 가장 작은 값을 선택하는 정책을 구현하세요

```
Map<Employee, Integer> em = newEmployeeStream2()
    .collect(Collectors.toMap(Function.identity(),
        Employee::getScore, ? );
```

Employee
<google, 40>
<yahoo, 20>
<naver, 30>
<naver, 20>

Name	Score
google	40
yahoo	20
naver	20

48

도전하세요!

- 직원 데이터 점수가 key고, 이름목록이 value인 Transactional data를 만드세요

```
Map<Integer, HashSet<String>> em = newEmployeeStream2()
    .collect( ? );
```

Employee
<google, 40>
<yahoo, 20>
<naver, 30>
<naver, 20>

Score	HashSet<Name>
20	naver, yahoo
30	naver
40	google

49

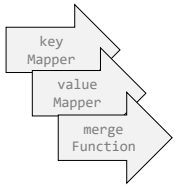
Collectors.toMap

- mapSupplier
  – 최종 결과 Map의 상세타입 지정

```
<T,K,U,M extends Map<K,U>> Collector<T,?,M> toMap(
    Function<? super T,? extends K> keyMapper,
    Function<? super T,? extends U> valueMapper,
    BinaryOperator<U> mergeFunction, Supplier<M> mapSupplier)
```

```
TreeMap<Employee, Integer> em = newEmployeeStream2()
    .collect(Collectors.toMap(Employee::getScore,
        Employee::getScore, (v1, v2) -> v1 + v2,
        TreeMap::new);
```

Entry
<google, 40>
<yahoo, 20>
<naver, 30>
<naver, 20>



Name	Score
google	40
yahoo	20
naver	50

50

toConcurrentMap

- 동시 집계
  – Unordered: 교환적(Associative)이고 결합적인(Commutative) 연산으로 동작
  – Concurrent: Accumulator연산 시 공유되는 임시중간결과 저장소

```
<T,K,U> Collector<T,?,ConcurrentMap<K,U>>
        toConcurrentMap(Function<? super T,? extends K> keyMapper
                , Function<? super T,? extends U> valueMapper)

<T,K,U> Collector<T,?,ConcurrentMap<K,U>>
        toConcurrentMap(Function<? super T,? extends K> keyMapper
                , Function<? super T,? extends U> valueMapper
                , BinaryOperator<U> mergeFunction)

<T,K,U,M extends ConcurrentMap<K,U>> Collector<T,?,M>
        toConcurrentMap(Function<? super T,? extends K> keyMapper
                , Function<? super T,? extends U> valueMapper
                , BinaryOperator<U> mergeFunction
                , Supplier<M> mapSupplier)
```

51

정리

- toMap
  - toConcurrentMap

52