스트림(Stream)이란

JDK 8부터 새롭게 추가된 기능으로 데이터 집합체를 반복적으로 처리한다.
스트림을 이용하면 다수의 스레드 코드를 구현하지 않아도 데이터를 병렬로 처리한다.
스트림은 스트림 데이터와 스트림 연산의 개념을 모두 포함한다.

컬렉션과 스트림 구분

- 컬렉션: 데이터의 공간적 집합체
  스트림: 데이터의 시간적 집합체
- 컬렉션: 데이터 원소의 효율적인 관리와 접근에 맞게 설계
  스트림: 데이터 원소에서 수행할 함수형 연산에 맞게 설계 (=> 원소에 직접 접근하거나 조작하는 수단을 제공하지 않음.)
- 스트림을 사용하면 코드가 간단해지고 오류 발생 확률이 줄어든다.
  (체이닝 기법처럼 메소드 뒤에 메소드가 오는 것!)

스트림 종류와 생성

종류

```
baseStream (객체 스트림: Stream, 숫자 스트림: IntStream, LongStream, DoubleStream)
```

숫자스트림과 객체 스트림 차이

- 숫자스트림: 평균, 합계를 반환(집계연산자)하는 메소드가 있고 데이터 스트림의 기본 통계 용약 내용을 나타내는 summaryStatistics() 메소드를 제공한다.
- 객체 스트림: 최종 연산이 Optional 타입을 반환 // 숫자 스트림: OptionalInt 타입,OptionalLong 타입, OptionalDouble 타입을 반환
- 객체 스트림으로 숫자 스트림을 만들 수 있지만 성능이 떨어지고 average나 sum을 사용할 수 없다.

```
IntStream => Stream<Integer> // 사용가능
```

생성

컬렉션으로 부터 스트림을 생성한다.

```
default Stream<E> stream()
default Stream<E> paralleStream()


list.stream() // 가능
```

배열로부터 스트림 생성 (각자에 맞춰 import를 해줘야 생성 가능)

- Arrays 클래스 제공

```
static IntStream stream(int[] array)
static IntStream of(int ... values)
```

- Stream 인터페이스 제공

```
static <T> Stream of (T ... values)
```

- IntStream 인터페이스 제공

```
static IntStream of (int ...values)
```

기타 데이터로부터 스트림 생성

- Random의 ints(), longs(), doubles()
- 숫자 스트림과 객체 스트림의 iterate(), generate()
- IntStream, LongStream의 range(), rangeClosed()
- 입출력 파일이나 폴더로부터도 스트림을 생성

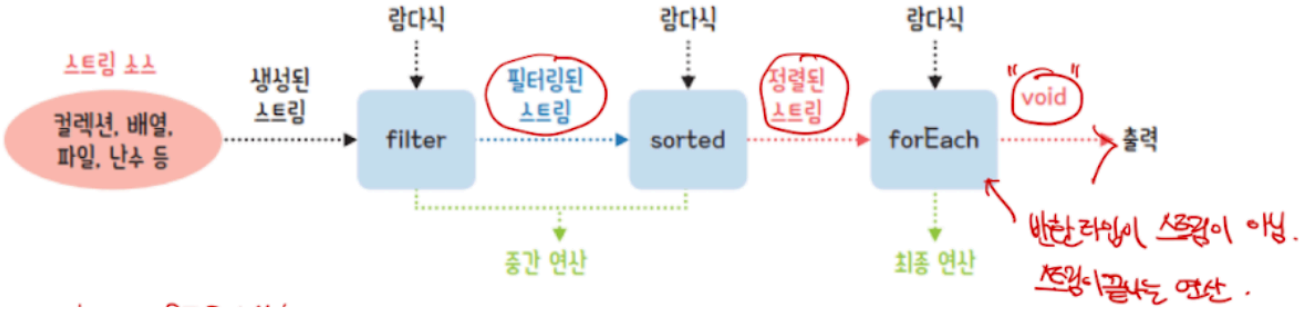
스트림 연산과 옵션 타입

스트림 파이프라인 (체이닝 기법)

스트림 연산의 결과가 Stream 타입이면 연속적으로 호출할 수 있다.
스트림 연산의 연속 호출은 여러개의 스트림이 연결되어 스트림 파이프라인을 형성한다.

느긋한 연산과 조급한 연산

- 느긋한 연산: 조급한 연산이 데이터 소스에게 원소를 요구할 때까지 아무 연산도 수행하지 않고 기다린다.
- 스트림의 최종연산은 조급한 연산이지만 중간 연산은 느긋한 연산이다.
- 최종 연산이 호출되기 전까지 중간 연산은 아무런 작업을 수행하지 않는다. (최종 연산은 값이 필요해!! -> 중간 연산이 실행 돼!)
- 스트림의 중간 연산이 느긋한 연산이니 다운로드 방식처럼 저장 공간이 따로 필요 없어 빅데이터나 무한 스트림에도 대응할 수 있다. (최종 연산이 되기 전에 실행되지 않으니 공간이 필요 없다!!)
- 최종 연산은 반환타입이 스트림이 아니다!! (반환 타입이 스트림이면 느긋한 연산!!)
- forEach 실행 -> sorted -> filter



병렬처리

- 멀티 코어 CPU를 제대로 활용해 데이터 소스를 병렬로 처리할 수 있도록 병렬 스트림도 지원한다.
- 병렬 스트림은 컬렉션 혹은 순차 스트림으로 부터 parallelStream() 혹은 parallel() 메소드를 호출해 획득한다.
- 스트림을 부분 스트림으로 분할하기 어렵거나 데이터 소스의 크기가 작거나 혹은 싱글 코어 CPU라면 오히려 성능이 나빠질 수 있다.
  => 예외를 제외하고는 병렬처리가 순차처리보다 시간이 빠르다.

옵션타입(JAVA8이후에나온개념)

- String는 Date는 상속관계가 아닌데 null을 사용할 수 있다.

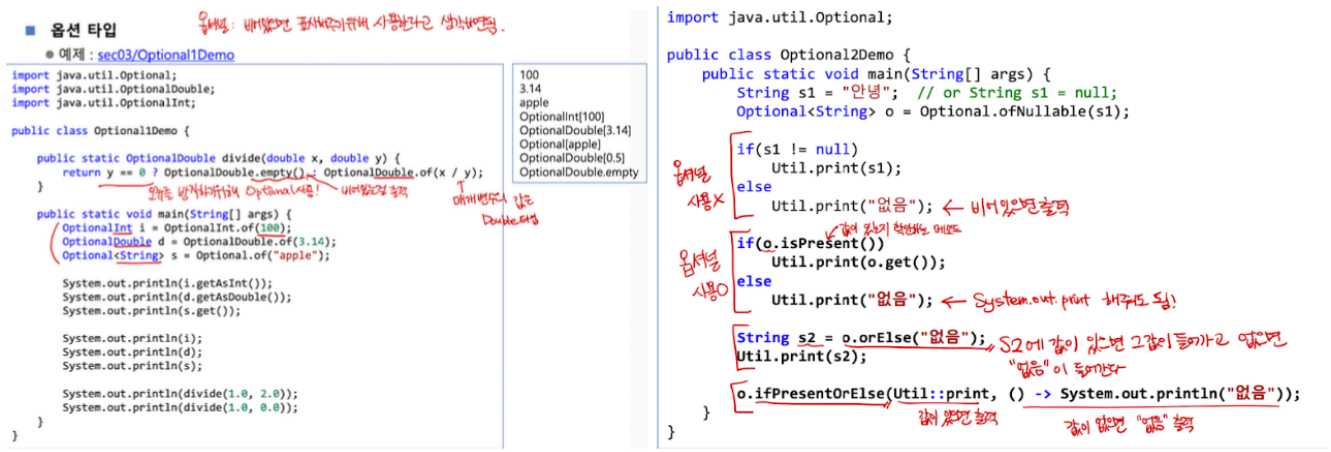
```
String s = null;
Date d = null;
```

- java.util 패키지 소속이다.
- Optional은 null을 사용하지 않고, 부재 값을 포함한 데이터를 저장하는 클래스이다.

- 값을 존재 여부에 따라 다양하게 처리할 수 있는 기능을 제공한다.
- 종류 : Optional, OptionalInt, OptionalLong, OptionalDouble
- Optional 및 OptionalInt 클래스가 제공하는 주요 연산

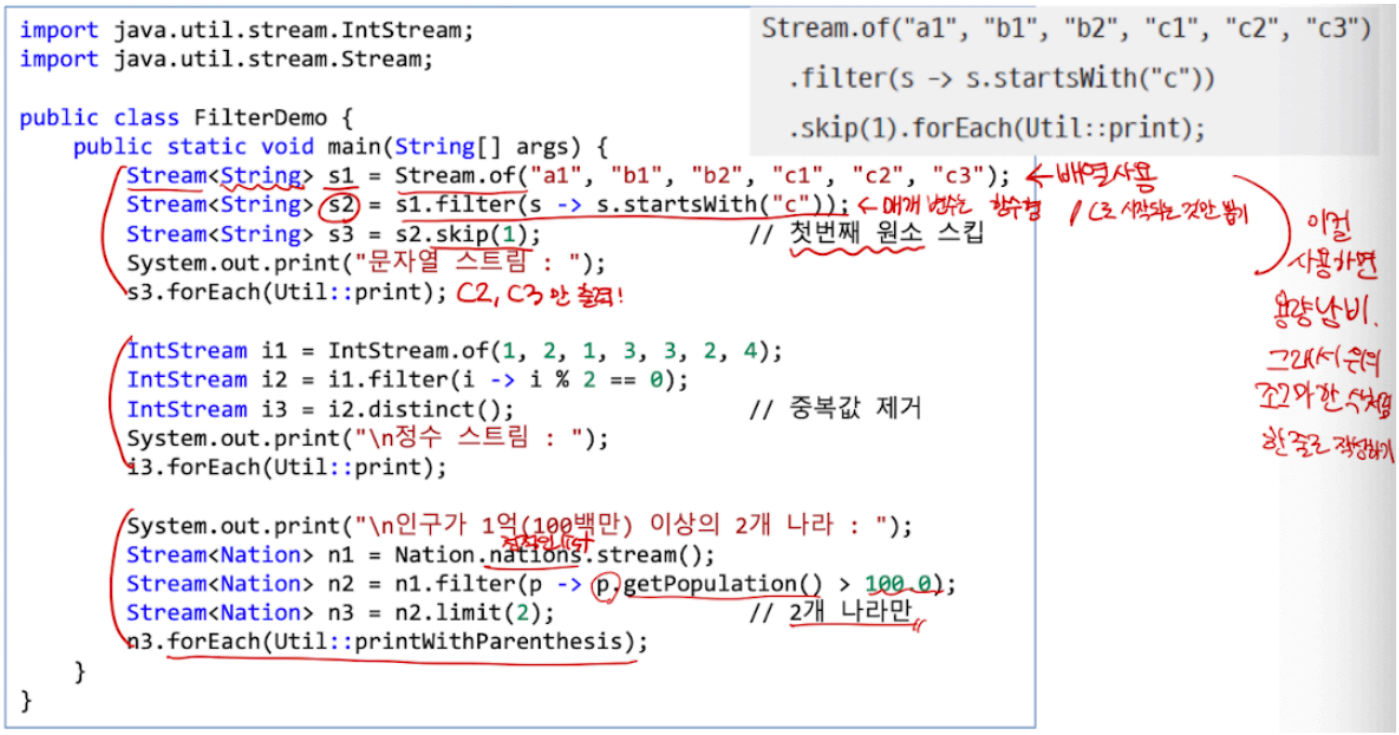
```
// Optional
static Optional empty()              // 빈 Optional 객체를 반환한다.
T get()                              // 값을 반환하며, 없다면 예외를 던진다.
boolean isPresent()                  // 값의 존재 여부를 반환한다.
void ifPresent(Consumer)             // 주어진 값을 가진 Optional타입으로 변환한다.
static Optional of(T)                // 주어진 값을 Optional을 반환한다. 만약 null이라면 빈
Optional을 반환한다.
T ofElse(T)                          // 값이 없을 때 디폴트 값을 지정한다.

// OptionalInt
int getAsInt()                       // 값을 반환하며, 없다면 예외를 던진다.
static OptionalInt of(int)           // 주어진 값을 가진 Optional타입으로 변환한다.
static Optional ofNullable(int)      // 주어진 값을 Optional을 반환한다. 만약 null이라면 빈
Optional을 반환한다.
int ofElse(int)                      // 값이 없을 때 디폴트 값을 지정한다.
```

- 예제



스트림 활용 - 필터링

입력된 스트림 원소 중 일부 원소를 걸러내는 중간 연산이다.
필터링 연산: filter(), distinct(), limit(), skip()



스트림 활용 - 매핑

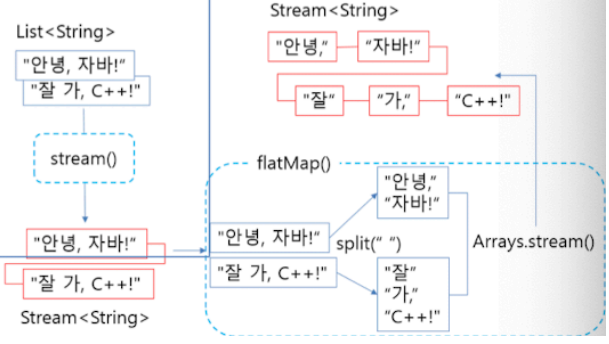
매개 값으로 제공된 람다식을 이용하여 입력 스트림의 객체 원소를 다른 타입 혹은 다른 객체 원소로 매핑한다.

ex) Map(), flatMap(), mapToObj(), mapToInt(), asLongStram(), boxed()

- 예제

```java
import java.util.stream.IntStream;
import java.util.stream.Stream;

public class Map1Demo {
    public static void main(String[] args) {
        Stream<String> s1 = Stream.of("a1", "b1", "b2", "c1", "c2");

        Stream<String> s2 = s1.map(String::toUpperCase);
        s2.forEach(Util::print);
        System.out.println();

        Stream<Integer> i1 = Stream.of(1, 2, 1, 3, 3, 2, 4);

        Stream<Integer> i2 = i1.map(i -> i * 2);
        i2.forEach(Util::print);
        System.out.println();

        Stream<String> s3 = Stream.of("a1", "a2", "a3");

        Stream<String> s4 = s3.map(s -> s.substring(1));

        IntStream i3 = s4.mapToInt(Integer::parseInt);

        Stream<String> s5 = i3.mapToObj(i -> "b" + i);

        s5.forEach(Util::print);
    }
}
```

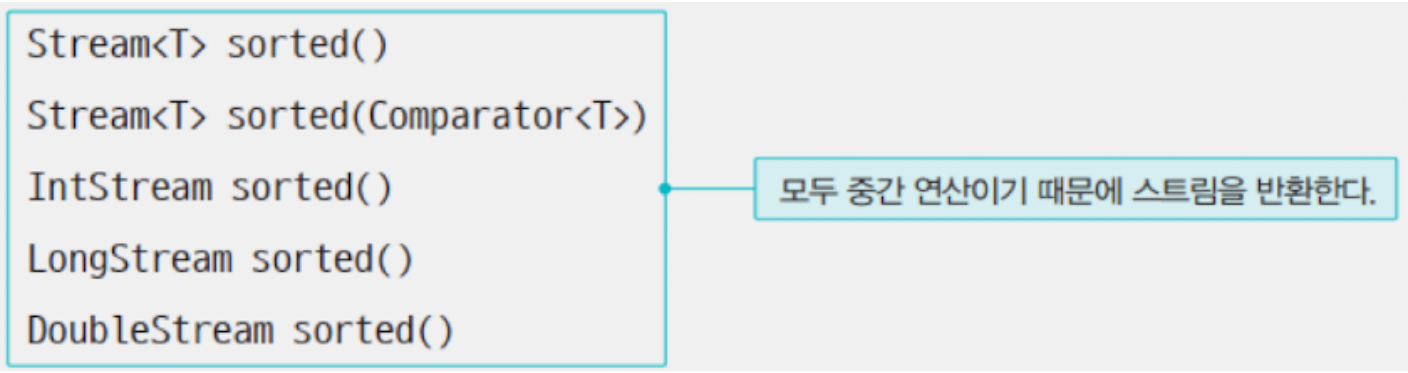
```java
import java.util.Arrays;
import java.util.List;
import java.util.stream.Stream;

public class Map3Demo {
    public static void main(String[] args) {
        List<String> list1 = List.of("안녕, 자바!", "잘 가, C++!");
        Stream<String> s1 = list1.stream();
        Stream<String> s2 = s1.flatMap(s -> Arrays.stream(s.split(" ")));
        s2.forEach(Util::printWithParenthesis);
        System.out.println();

        List<String> list2 = List.of("좋은 아침");
        List<String> list3 = List.of("안녕! 람다", "환영! 스트림");

        Stream<List<String>> s3 = Stream.of(list1, list2, list3);
        Stream<String> s4 = s3.flatMap(list -> {
            if (list.size() > 1)
                return list.stream();
            else
                return Stream.empty();
        });
        s4.forEach(Util::printWithParenthesis);
    }
}
```



스트림 활용 - 정렬

입력된 스트림 원소 전체를 정렬하는 중간 연산으로 distinct() 연산과 마찬가지로 버퍼가 필요하다.

ex)



- 예제

```
import java.util.Comparator;
import java.util.stream.Stream;

public class SortedDemo {
    public static void main(String[] args) {
        Stream<String> s1 = Stream.of("d2", "a2", "b1", "b3", "c");
        Stream<String> s2 = s1.sorted();
        s2.forEach(Util::print);

        System.out.print("\n국가 이름 순서 : ");
        Stream<Nation> n1 = Nation.nations.stream();
        Stream<Nation> n2 = n1.sorted(Comparator.comparing(Nation::getName));
        Stream<String> s3 = n2.map(x -> x.getName());
        s3.forEach(Util::printWithParenthesis);

        System.out.print("\n국가 GDP 순서 : ");
        Stream<Nation> n3 = Nation.nations.stream();
        Stream<Nation> n4 = n3.sorted(Comparator.comparing(Nation::getGdpRank));
        Stream<String> s4 = n4.map(Nation::getName);
        s4.forEach(Util::printWithParenthesis);
    }
}
```

```
a2 b1 b3 c d2
국가 이름 순서 : (China) (Morocco) (New Zealand) (Philiphine) (ROK) (Sri Lanka) (USA) (United Kingdom)
국가 GDP 순서 : (USA) (China) (United Kingdom) (ROK) (Philiphine) (New Zealand) (Morocco) (Sri Lanka)
```

스트림 활용 - 매칭과 검색

특정 속성과 일치되는 스트림 원소의 존재 여부를 조사하거나 검색하는데 사용되는 스트림의 최종 연산이다.
ex)

```
boolean allMatch(Predicate<? super T> predicate)
boolean anyMatch(Predicate<? super T> predicate)
boolean noneMatch(Predicate<? super T> predicate)
Optional<T> findAny()
Optional<T> findFirst()
```

- 예제

■ **매칭과 검색**

● 예제 : sec04/MatchDemo

```
true
false
false
모든 국가의 인구가 1억이 넘지 않는다.
ROK
New Zealand
```

```
import java.util.Optional;
import java.util.stream.IntStream;
import java.util.stream.Stream;

public class MatchDemo {
    public static void main(String[] args) {
        boolean b1 = Stream.of("a1", "b2", "c3").anyMatch(s -> s.startsWith("c"));
        System.out.println(b1);

        boolean b2 = IntStream.of(10, 20, 30).allMatch(p -> p % 3 == 0);
        System.out.println(b2);

        boolean b3 = IntStream.of(1, 2, 3).noneMatch(p -> p == 3);
        System.out.println(b3);

        if (Nation.nations.stream().allMatch(d -> d.getPopulation() > 100.0))
            System.out.println("모든 국가의 인구가 1억이 넘는다.");
        else
            System.out.println("모든 국가의 인구가 1억이 넘지 않는다.");

        Optional<Nation> nation = Nation.nations.stream().findFirst();
        nation.ifPresentOrElse(Util::print, () -> System.out.print("없음."));
        System.out.println();

        nation = Nation.nations.stream().filter(Nation::isIsland).findAny();
        nation.ifPresent(Util::print);
    }
}
```

스트림 활용 - 루핑과 단순 집계

루핑은 전체 원소를 반복하는 연산이다.

- 루핑의 종류

forEach() : 최종연산이다.
peek() : 중간연산이다.

단순집계는 스트림을 사용하여 스트림 원소의 개수, 합계, 평균값, 최댓값, 최솟값 등과 같은 하나의 값을 도출하는 최종 연산이다.

- 단순 집계 연산의 종류

count(), sum(), average(), max(), min() 등이 있다.

- 예제

```
package sec04;

import java.util.Comparator;
import java.util.Optional;
import java.util.stream.IntStream;
import java.util.stream.Stream;

public class LoopAggregateDemo {
    public static void main(String[] args) {
        Stream<Nation> sn =
Nation.nations.stream().peek(Util::printWithParenthesis);
        System.out.println("어디 나타날까?");
        Optional<Nation> on =
sn.max(Comparator.comparing(Nation::getPopulation));
        System.out.println();
        System.out.println(on.get());

        System.out.println(IntStream.of(5, 1, 2, 3).min().getAsInt());

        sn = Nation.nations.stream();
        System.out.println(sn.count());
    }
}
```

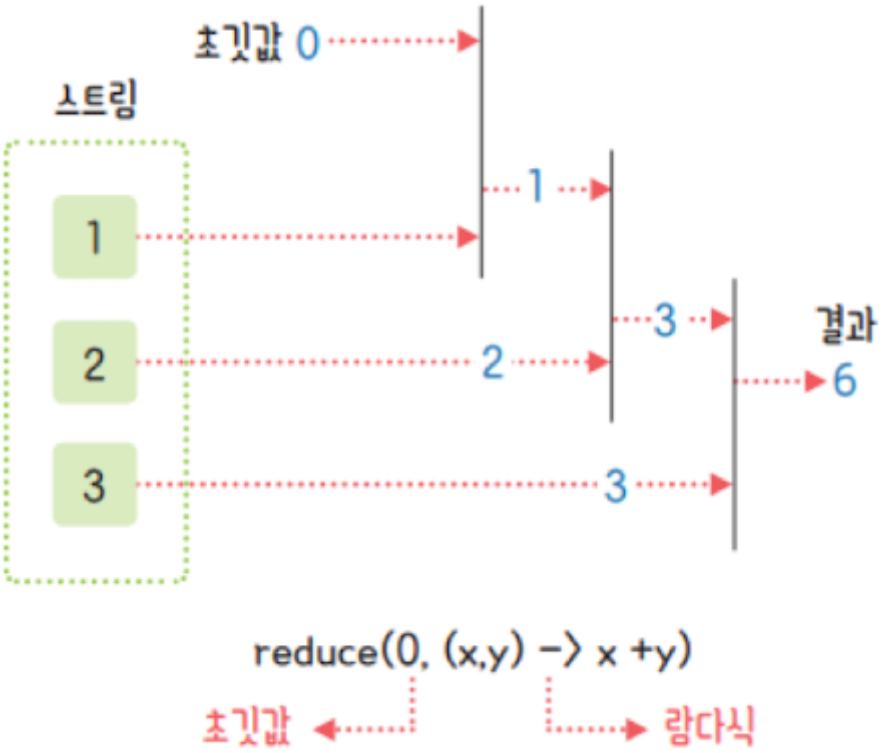
```
어디 나타날까?
(ROK) (New Zealand) (USA) (China) (Philiphine) (United Kingdom) (Sri Lanka) (Morocco)
China
1
8
```

스트림을 이용한 집계와 수집

리듀싱 연산

원소 개수, 합계, 평균값 등과 같은 단순 집계가 아니라면 자바 API가 제공하는 기본 집계 연산으로 감당할 수 없다.
-> 이를 위해 Stream 인터페이스는 람다식을 사용해 복합적인 집계 결과를 도출할 수 있도록 리듀싱 기능을 제공한다.

리듀싱 기능 : 스트림 원소를 단 하나의 값으로 축약시키는 연산이다.



리듀싱 연산

메서드	의미
Optional reduce(BinaryOperator) OptionalInt reduce(IntBinaryOperator)	2개 원소를 조합해서 1개 값으로 축약하는 과정을 반복하여 집계한 결과를 반환한다.
T reduce(T, BinaryOperator) int reduce(int, IntBinaryOperator)	첫 번째 인숫값을 초깃값으로 제공한다는 것을 제외하면 위 메서드와 같다.

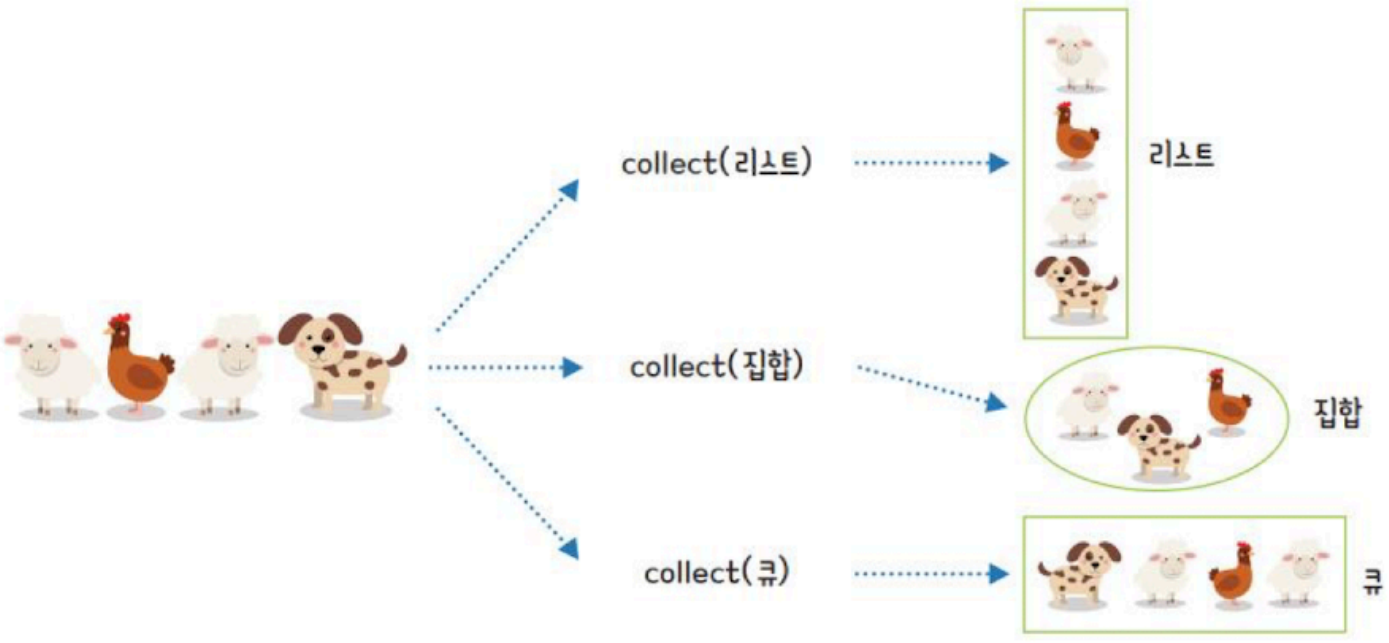
- 예제

```java
import java.util.List;
import java.util.Optional;
import java.util.OptionalInt;

public class Reduce1Demo {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(3, 4, 5, 1, 2);

        int sum1 = numbers.stream().reduce(0, (a, b) -> a + b);

        int sum2 = numbers.stream().reduce(0, Integer::sum);

        int mul1 = numbers.stream().reduce(1, (a, b) -> a * b);

        System.out.println(sum1);
        System.out.println(sum2);
        System.out.println(mul1);

        Optional<Integer> sum3 = numbers.stream().reduce(Integer::sum);

        OptionalInt sum4 = numbers.stream().mapToInt(x -> x.intValue()).reduce(Integer::sum);

        Optional<Integer> mul2 = numbers.stream().reduce((a, b) -> a * b);

        System.out.println(sum3.get());
        System.out.println(sum4.getAsInt());
        mul2.ifPresent(Util::print);
    }
}
```

```
15
15
120
15
15
120
```

컬렉터 연산 : 수집 및 요약

컬렉터는 원소를 어떤 컬렉션에 수집할 것인지 결정하며 다음과 같은 역할을 수행한다.
-원소를 컬렉션이나 StringBuilder와 같은 컨테이너에 수집한다.
-원소를 구분자와 같은 문자와 합칠 수 있다.
-원소를 그룹핑해 새로운 컬렉션을 구성한다.



- Collectors 클래스가 제공하는 주요 정적 메소드

메서드	의미
Collector averagingInt(ToIntFunction)	정수 원소에 대한 평균을 도출한다.
Collector counting( )	원소 개수를 헤아린다.
Collector joining(CharSequence)	문자열 원소를 연결하여 하나의 문자열로 결합한다.
Collector mapping(Function, Collector)	원소를 매핑한 후 컬렉터로 수집한다.
Collector maxBy(Comparator)	원소의 최댓값을 구한다.
Collector minBy(Comparator)	원소의 최솟값을 구한다.
Collector summingInt(ToIntFunction)	원소의 숫자 필드의 합계, 평균 등을 요약한다.
Collector toList( )	원소를 List에 저장한다.
Collector toSet( )	원소를 Set에 저장한다.
Collector toCollection(Supplier)	원소를 Supplier가 제공한 컬렉션에 저장한다.
Collector toMap(Function, Function)	원소를 키-값으로 Map에 저장한다.

- 예제

● 예제 : sec05/Collect1Demo

```
인구 평균 : 244.6
나라 개수 : 8
4개 나라(방법 1) : ROK-New Zealand-USA-China
4개 나라(방법 2) : ROK+New Zealand+USA+China
최대 인구 나라의 인구 수 : Optional[1355.7]
IntSummaryStatistics{count=8, sum=227, min=1, average=28.375000, max=63
```

```java
import java.util.IntSummaryStatistics;
import java.util.Optional;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class Collect1Demo {
    public static void main(String[] args) {
        Stream<Nation> sn = Nation.nations.stream();
        Double avg = sn.collect(Collectors.averagingDouble(Nation::getPopulation));
        System.out.println("인구 평균 : " + avg);

        sn = Nation.nations.stream();
        Long num = sn.collect(Collectors.counting());
        System.out.println("나라 개수 : " + num);

        sn = Nation.nations.stream();
        String name1 = sn.limit(4).map(Nation::getName).collect(Collectors.joining("-"));
        System.out.println("4개 나라(방법 1) : " + name1);

        sn = Nation.nations.stream();
        String name2 = sn.limit(4).collect(Collectors.mapping(Nation::getName, Collectors.joining("+")));
        System.out.println("4개 나라(방법 2) : " + name2);

        sn = Nation.nations.stream();
        Optional<Double> max = sn.map(Nation::getPopulation).collect(Collectors.maxBy(Double::compare));
        System.out.println("최대 인구 나라의 인구 수 : " + max);

        sn = Nation.nations.stream();
        IntSummaryStatistics sta = sn.collect(Collectors.summarizingInt(x -> x.getGdpRank()));
        System.out.println(sta);
    }
}
```

스트림 원소를 그룹핑하여 새로운 컬렉션 구성

그룹핑
-키와 값의 쌍으로 이루어진 map 원소를 수집한다.
-두 번째 매개 값인 컬렉터가 없으면 list타입이다.
-groupingBy() 연산으로 수집된 map 타입을 결과로 도출한다.

```
Collector groupingBy(Function);
Collector groupingBy(Function, Collector);
```

- 예제

■ **그룹핑**

● 예제 : sec05/GroupingDemo

```
{LAND=[ROK, USA, China], ISLAND=[New Zealand]}
{LAND=3, ISLAND=1}
{LAND=ROK#USA#China, ISLAND=New Zealand}
```

```java
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class GroupingDemo {
    public static void main(String[] args) {
        Stream<Nation> sn = Nation.nations.stream().limit(4);
        Map<Nation.Type, List<Nation>> m1 = sn
                .collect(Collectors.groupingBy(Nation::getType));
        System.out.println(m1);

        sn = Nation.nations.stream().limit(4);
        Map<Nation.Type, Long> m2 = sn
                .collect(Collectors.groupingBy(Nation::getType, Collectors.counting()));
        System.out.println(m2);

        sn = Nation.nations.stream().limit(4);
        Map<Nation.Type, String> m3 = sn.collect(
                Collectors.groupingBy(Nation::getType,
                        Collectors.mapping(Nation::getName,
                                Collectors.joining("#"))));
        System.out.println(m3);
    }
}
```

파티셔닝

조건에 따라 그룹핑한다.

```
Collector partitioningBy(Predicate);
Collector partitioningBy(Predicate, Collector);
```

- 예제

파티셔닝

- 예제 : sec05/PartitioningDemo

```
{false=[New Zealand], true=[ROK, USA, China]}
{false=1, true=3}
{false=New Zealand, true=ROK#USA#China}
```

```java
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class PartitioningDemo {
    public static void main(String[] args) {
        Stream<Nation> sn = Nation.nations.stream().limit(4);
        Map<Boolean, List<Nation>> m1 = sn
                .collect(Collectors.partitioningBy
                        (p -> p.getType() == Nation.Type.LAND));
        System.out.println(m1);

        sn = Nation.nations.stream().limit(4);
        Map<Boolean, Long> m2 = sn
                .collect(Collectors.partitioningBy(
                        p -> p.getType() == Nation.Type.LAND,
                        Collectors.counting()));
        System.out.println(m2);

        sn = Nation.nations.stream().limit(4);
        Map<Boolean, String> m3 = sn
                .collect(Collectors.partitioningBy(
                        p -> p.getType() == Nation.Type.LAND,
                        Collectors.mapping(Nation::getName,
                                Collectors.joining("#"))));
        System.out.println(m3);
    }
}
```