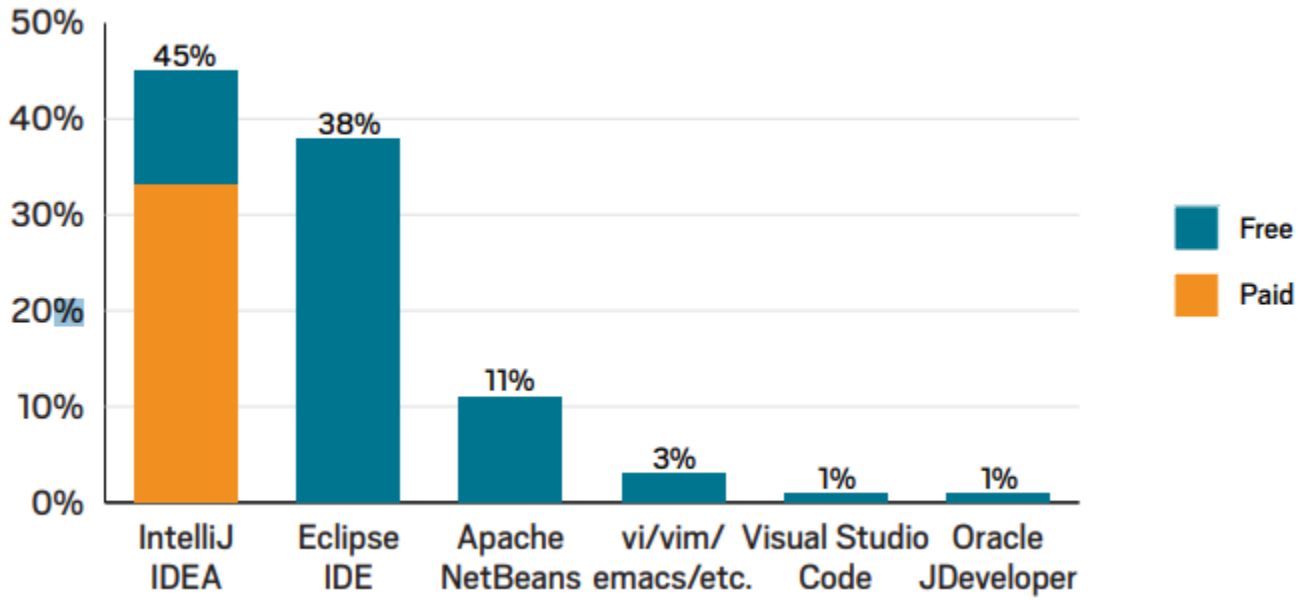


NOVEMBER/DECEMBER 2018

Stream

sanghyuck.na@lge.com

+Stream source operation

+Simple terminal operation

Stream[8]

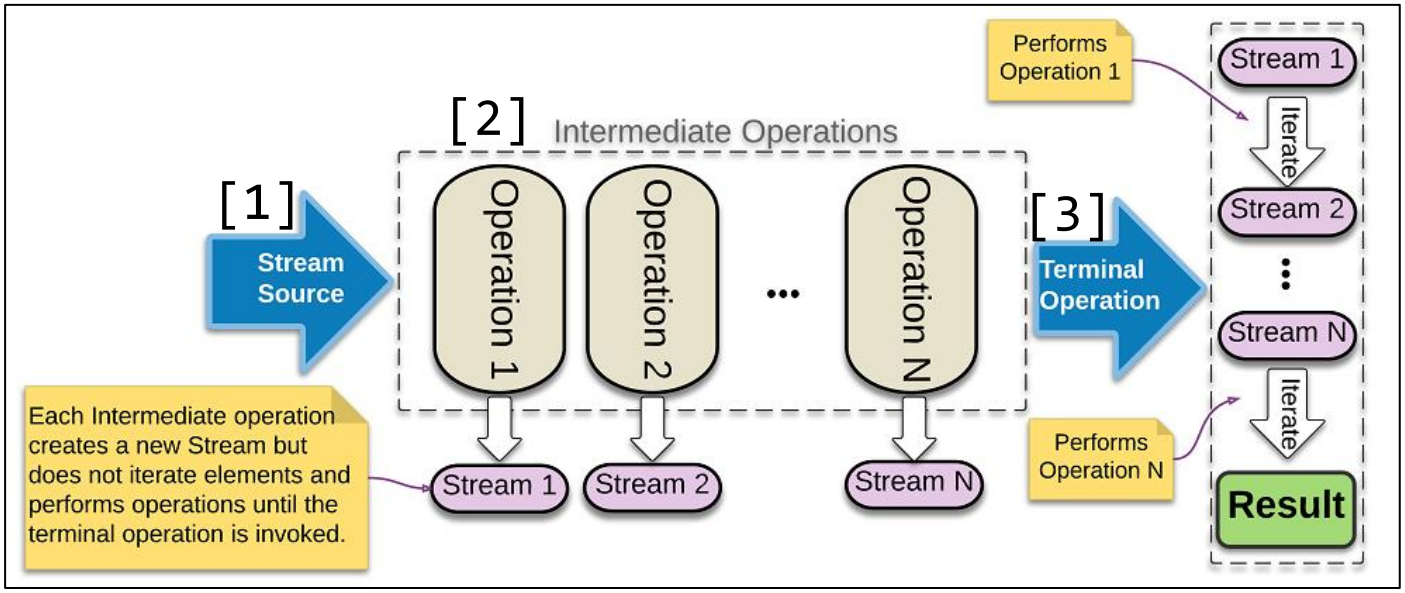
- 순차 + 병렬처리 및 집계연산 라이브러리
  - 재사용 불가객체로 Lambda expression를 이용해서 동작 설정
  - 주요연산자: Map, Reduce, Filter, Flat-map, Collect
- 주요 특징
  1. No storage: 임시변수 및 임시데이터를 유지 안 함
  2. Functional in nature: 원본을 변경하지 않음
  3. Laziness-seeking: 종료연산자 실행시점에 시작
  4. Possibly unbounded: 무한데이터 처리 가능
  5. Consumable: 입력 데이터는 단 한번 만 사용(접근)

```
double sum = 0;
for (int i=0; i<100; i++) {
    double j=Math.random();
    if (j > 50) {
        sum += j;
    }}
```

```
long sum = Stream
     .generate(Math::random)
    .limit(100)
    .filter(b -> b > 50)
    .mapToLong(Double::valueOf)
    .sum();
```

Workflow

- Major three factors in stream
  - Source + Intermediate operation + Terminal operation



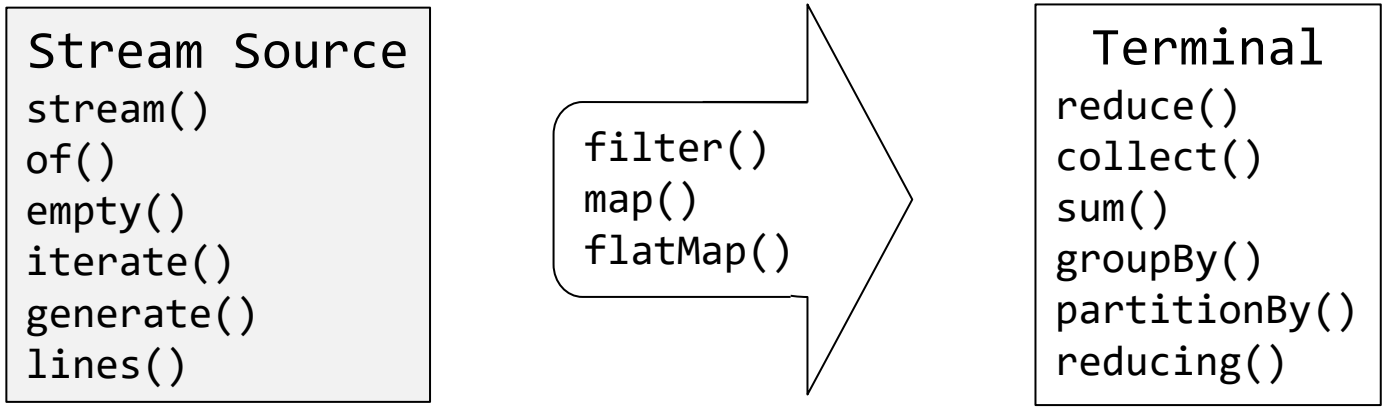
Stream Pipeline
Declaration
↓
Execution

```
long sum
[1]     = Arrays.stream(a)
[2]    .limit(100)
[2]    .filter(b -> b > 50)
[2]    .mapToLong(Long::valueOf)
[3]    .sum();
```

Streams under the hood

Stream source

분류	API
Stream factory	Stream.of(), Stream.iterator(), IntStream.range()
Arrays	stream(Object[])
Collection	stream(), parallelStream()
Random	Random.ints()
Files	Files.find(), Files.list(), Files.line()
BufferedReader	BufferedReader.lines()
Others	BitStet.stream(), Pattern.splitAsStream() JarFile.stream() …



Stream factory methods

- Static factory methods from Stream
  1. Empty Stream, Item sequence, From array
  2. Iterator, generate

```
Stream<T> iterate(T seed, UnaryOperator<T> f)[8]
Stream<T> iterate(T seed, Predicate<? super T> hasNext,
                          UnaryOperator<T> next)[9]
Stream<T> generate(Supplier<? extends T> s)
```

```
Stream<Integer> e = Stream.empty();
Stream<Integer> s = Stream.of(1, 2, 3);

Stream<String> a1 = Stream.of(Locale.getISOCountries());
Stream<String> a2 = Arrays.stream(Locale.getISOCountries());

Stream<Integer> i = Stream.iterate(0, i -> i + 1);
Stream<Integer> i2 = Stream.iterate(0, i -> i < 20, i -> i + 1);

Stream<String> g = Stream.generate(() -> "Echo");
Stream<Double> g2 = Stream.generate(Math::random);
```

From Arrays or Collection

- From Arrays

```
static <T> Stream<T> stream(T[] array)[8]
static <T> Stream<T> stream(T[] array,
        int startInclusive, int endExclusive)[8]
```

- From Collection

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

```
String[] array = Locale.getISOCountries();
Stream<String> sa2 = Arrays.stream(array);
Stream<String> sa3 = Arrays.stream(array, 0, 10);

Stream<Integer> fl = List.of(1, 2, 3).stream();
Stream<Integer> fs = Set.of(1, 2, 3).stream();
Stream<Entry<Integer, String>> fe
        = Map.of(1,"First", 2, "Second")
             .entrySet().stream();
```

도전하세요!

- Stream 만들기

```
Stream<String> ss = ?;
// 초기값 "It's me" 이 후엔 문자열 가장 뒤에 "+" 가 계속 추가
// "It’s me+", "It’s me++"
Stream<BigInteger> bs = ?; // 초기값은 2, 이후엔 x2

Stream<Double> rd = ?; // Math.random()호출하는 스트림

Stream<Integer> seq = ?;// 79, 68, 55, 59, 77로 구성된 스트림

double myds[] = new double[] {0.0466, 0.5751, 0.6599};
DoubleStream sa = ?; // 데이터가 Index [1], [2]로 구성된 스트림

List<Integer> ints = new ArrayList<>(){{ add(-1387513903);
add(164529915);}};
Stream<Integer> pis = ?; // List ints를 이용해서 병렬스트림 만들기
```

From Files

- File, Directory를 순회, 변경을 위한 스트림

```
Stream<Path> list(Path dir)8
Stream<String> lines(Path path)8
Stream<Path> walk(Path start, FileVisitOption… options)8
Stream<Path> find(Path start, int maxDepth,
                  BiPredicate<Path, BasicFileAttributes> matcher,
                  FileVisitOption… options)8
DirectoryStream<Path> Files.newDirectoryStream(Path dir)8
```

```
String fpath = "C:\\Windows\\System32\\drivers\\etc\\hosts";

try (Stream<String> lines = Files.lines(Paths.get(fpath))) {
    lines.forEach(System.out::println);
} catch (IOException e) {
    e.printStackTrace();
}
```

From Pattern

- 정규식으로 파싱 후 스트림 반환
  - stream static factory method

```
Pattern.splitAsStream(CharSequence input)[8]
```

```
String ip = "10.221.51.2";
Pattern.compile("\\.").splitAsStream(ip)
    .forEach(System.out::println);
```

Basic terminal operation – Stream.collect(C)

- Stream의 데이터E를 결과R타입으로 생성하는 연산자
  - Collector: 입력 데이터 T를 함수A를 적용하여 결과 R계산

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

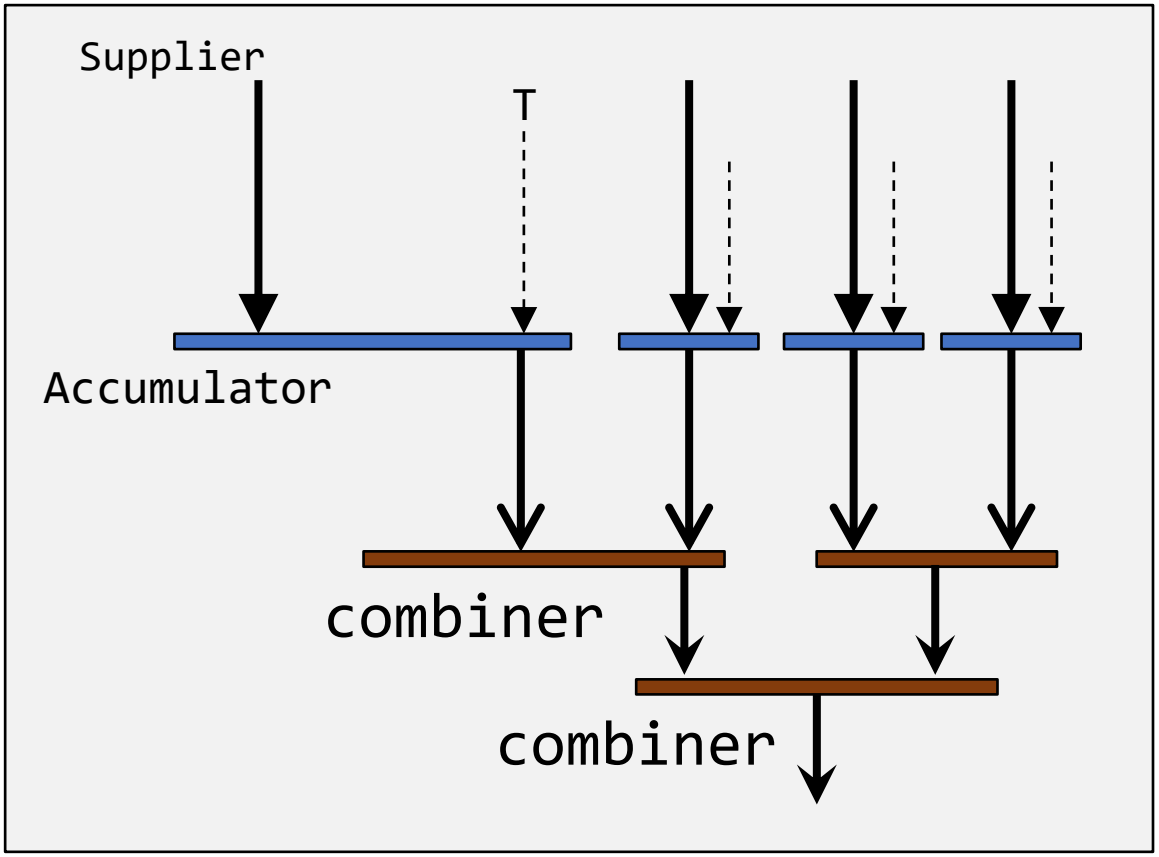


```
static <T> Collector<T,?,List<T>> Collectors.toList()
static <T> Collector<T,?,Set<T>> toSet()
static <T,K,U> Collector<T,?,Map<K,U>> toMap(…)
```

```
List<Integer> lst = Stream.of(1, 2, 3)
                          .collect(Collectors.toList());
Set<Integer> set = … .collect(Collectors.toSet());
```

Collector in Action – Aggregation

- Mutable reduction operation
  - Supplier: Result container 생성
  - Accumulator: Element + Container 결합
  - Combiner: Container + Container 결합



Basic terminal operation – Stream.collect(S,B,B)

- 연속된 입력R를 결과R타입으로 생성하는 연산자
  - 입력값과 결과 값의 타입 동일 R==R

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

```
List<Integer> lst = Stream.of(1, 2, 3)
    .collect(
        () -> new LinkedList<>(),
        (l, e) -> l.add(e),
        (l, b) -> l.addAll(b)
);
```

정리

- Stream
  - Source
  - Files
  - Patterns
- Collection 만들기

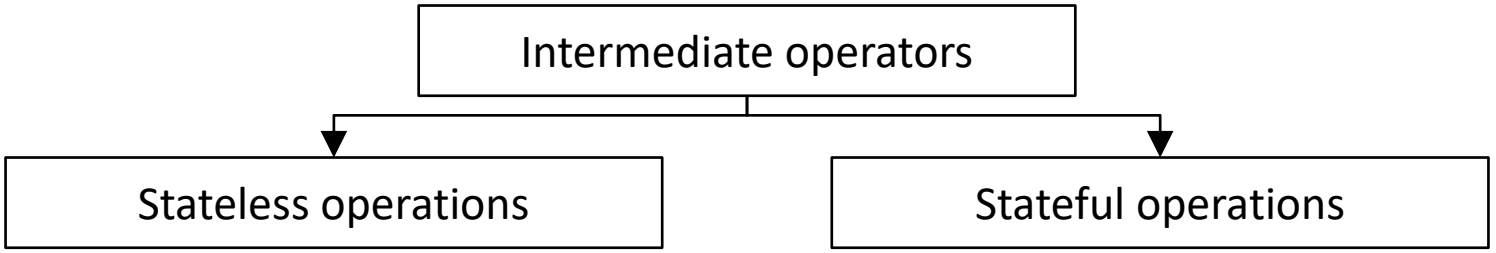
Intermediate operations

sanghyuck.na@lge.com

+ filter(), + map(), + flatMap()

Intermediate Operation

- 새로운 스트림을 생성하는 연산자
  - Lazy allows: Terminal operation호출까지 IO는 실행안 됨
  - Short-circuiting: 데이터/연산자를 모두 확인하지 않고 처리가능



API	분류
`Stream<T> filter(Predicate<… T> predicate)`	Stateless
`Stream<R> map(Function<… T, … R> mapper)`	Stateless
`Stream<R> flatMap(Function<…T, Stream<…R>> mapper)`	Stateless
`Stream<T> limit(long maxSize)`	Stateless
`Stream<T> skip(long n)`	Stateless
`Stream<T> peek(Consumer<? super T> action)`	Stateless
`Stream<T> concat(Stream<…> a, Stream<…> b)`	Stateless
`Stream<T> distinct()`	Stateful
`Stream<T> sorted()`	Stateful

Filter

- 표현식(Predicate)의 결과가 true인 데이터로 구성하는 Stream 생성
  - 입/출력 데이터 타입: T

```
Stream<T> filter(Predicate<? super T> predicate)
```

```
interface Predicate<T> {
    boolean test(T t);
}
```

```
Predicate<Integer> p = i -> i > 1;
Stream <Integer> s = Stream.of(1, 2, 3).filter(p)
```

Filter – Logical

- 논리 부정

```
Predicate<T> negate()
Predicate<T> not(Predicate<? super T> target)[11]
```

- 논리 연산

```
Predicate<T> and(Predicate<? super T> other)
Predicate<T> or(Predicate<? super T> other)
```

```
Predicate<Integer> p = i -> i > 1;
Stream.of(1, 2, 3).filter(p.negate());

Stream.of(1, 2, 3).filter(Predicate.not(p));

Stream.of(1, 2, 3).filter(p.and(e -> e > 2));
Stream.of(1, 2, 3).filter(p.or(e -> e == 1));
```

도전하세요!

- "K" 또는 "C" 로 시작하는 문자열 개수를 구하세요

```
List<String> in = List.of(Locale.getISOCountries());

long cnt = in.stream().???.count();
```

- "mkyong"가 아닌 아이템 개수를 구하세요

```
List<String> lines = Arrays.asList(
    "spring", "node", "mkyong");

long cnt = lines.stream().???.count();
```

Map

- 입력데이터T 를 R로 변환하는 Stream 생성
  - T: 입력, R: 리턴타입

```
<R> Stream<R> map(Function<? super T
                           , ? extends R> mapper)
```

```
interface Function<T, R> {
        R apply(T t)
}
```

```
Function<String, Integer> f = (s)->Integer.parseInt(s);
Stream <Integer> s = Stream.of("1", "2", "3").map(f);
```

Map In Primitive

- Non-boxing

```
Stream<R> map(Function<? super T,? extends R> mapper)
```

```
IntStream
        mapToInt(ToIntFunction<? super T> mapper)

LongStream
        mapToLong(ToLongFunction<? super T> mapper)

DoubleStream
        mapToDouble(ToDoubleFunction<? super T> mapper)
```

Map operators

- 동치(Equivalence)
- 전치(preprocessing)
- 후치(postprocessing)

```
<T> Function<T,T> identity()
<V> Function<V,R> compose(Function<? super V,? extends T> before)
<V> Function<T,V> andThen(Function<? super R,? extends V> after)
```

```
Function<Integer, Integer> identity = (i)-> i;
Stream.of(1, 2, 3).map(identity)

Function<Integer, String> m = i -> String.valueOf(i);
Stream.of(1, 2, 3).map(m.compose(i -> i + 100));
Stream.of(1, 2, 3).map(m.andThen(s -> s + 100));
```

도전 하세요!

- 다음 리스트customers에서 적립금(points)의 총 평균을 구하세요

```
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));

OptionalDouble avg = customers.stream().??????.average();
avg.ifPresent(System.out::println);
```

FlatMap

- 입력데이터T 에서 R에 대한 스트림 Stream<R>을 반환하는 스트림 생성
  - 차원 축소(flattening) 목적으로 빈번히 사용
    - Stream<Stream<Integer>> → Stream<Integer>
    - Stream<Integer[]> → Stream<Integer>
    - Stream<List<Integer[]>> → Stream<Integer[]>

```
<R> Stream<R> flatMap(Function<? super T,
                ? extends Stream<? extends R>> mapper)
```

```
List<Integer> alst = List.of(1, 2);
List<Integer> blst = List.of(3, 4);

Stream<List<Integer>> all = Stream.of(alst, blst);
Stream<Integer> s = all.flatMap(e-> e.stream());
```

```
Integer[][] arrays = new Integer[][] {
    { 1, 2, 3, 4, 5 }, { 3, 4, 5, 6 }};
Stream<Integer[]> s = Stream.of(arrays);
Stream<Integer> t = s.flatMap(o -> Arrays.stream(o));
```

FlatMap In Primitive Stream

- Non-boxing

```
<R> Stream<R> flatMap(Function<? super T
            ,? extends Stream<? extends R>> mapper)
```

```
DoubleStream flatMapToDouble(Function<? super T
            ,? extends DoubleStream> mapper)

IntStream flatMapToInt(Function<? super T
             ,? extends IntStream> mapper)

LongStream flatMapToLong(Function<? super T
            ,? extends LongStream> mapper)
```

도전하세요!

- 문자열로 구성되어 리스트 strs로 단어로 분할한 리스트wordsLists 만들기

```
List<String> strs = List.of(
    “Imagine driving down the highway at 70 miles “,
    “per hour, when suddenly the wheel turns hard right. “,
    “You crash. And it was because someone hacked your car.”);

List<String> wordsList = strs.????.collect(Collectors.toList());
```

정리

- filter()
- map()
- flatMap()