Data Analysis
(Stream and Parallel Processing 2)

Fall, 2020

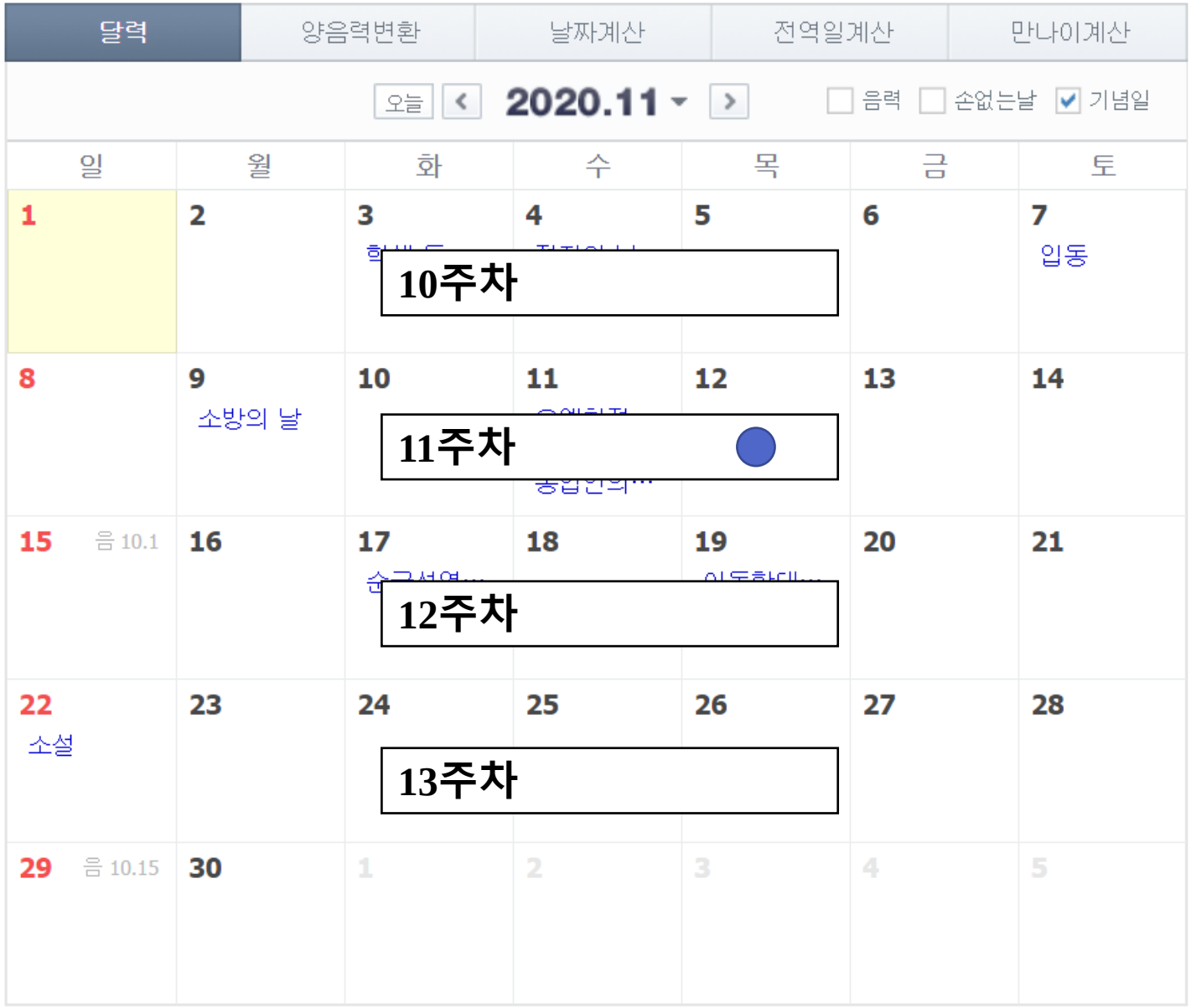
Calendar


달력
양음력변환
날짜계산
전역일계산
만나이계산
오늘
2020.09
음력
손없는날
기념일
일
월
화
수
목
금
토
30
31
1
소개
2
음 7.15
3
환경
세팅
4
지식재산…
5
6
7
백로
8
복습
1
9
10
복습
2
11
12
13
14
15
16
17
음 8.1
18
19
청년의 날
3주차
20
21
치매극복…
22
23
24
25
26
4주차
27
28
29
30
1
2
3
5주차

Calendar


달력
양음력변환
날짜계산
전역일계산
만나이계산
오늘
2020.10
음력
손없는날
기념일
일
월
화
수
목
금
토
27
28
29
30
1 음 8.15
추석
국군의 날
2
노인의 날
3
개천절
4
5
세계 한…
6
7
8
9
한글날
10
6주차
11
12
13
14
15
16
부마민주…
17 음 9.1
문화의 날
7주차
18
19
20
21
22
23
상강
24
국제연합일
8주차: 중간고사
25
독도의날
중양절
26
27
28
29
30
31 음 9.15
9주차

Calendar


달력
양음력변환
날짜계산
전역일계산
만나이계산
오늘
2020.11
음력
손없는날
기념일
일
월
화
수
목
금
토
1
2
3
4
5
6
7
입동
10주차
8
9
소방의 날
10
11
12
13
14
11주차
15
음 10.1
16
17
18
19
20
21
12주차
22
소설
23
24
25
26
27
28
13주차
29
음 10.15
30
1
2
3
4
5

Calendar


달력
양음력변환
날짜계산
전역일계산
만나이계산
오늘
2020.12
음력
손없는날
기념일
일
월
화
수
목
금
토
29
30
1
2
3
4
5
무역의 날
14주차
6
7
대설
8
9
10
11
12
15주차
13
14
15
음 11.1
16
17
18
19
16주차: 기말고사 주간
20
21
동지
22
23
24
25
성탄절
26
27
원자력의…
28
29
음 11.15
30
31
1
2

Table of Contents

- Parallel Stream
  - Reference
    - https://www.slideshare.net/dgomezg/parallel-streams-en-java-8
    - https://bmi.cchmc.org/sites/bmidrupalpbmi.chmcres.cchmc.org/files/admin/documents/events/Java%208%20Streams.pptx

(Parallel) Stream

- Stream
  - A convenient method to iterate over collections in a declarative way (Lambda expression)

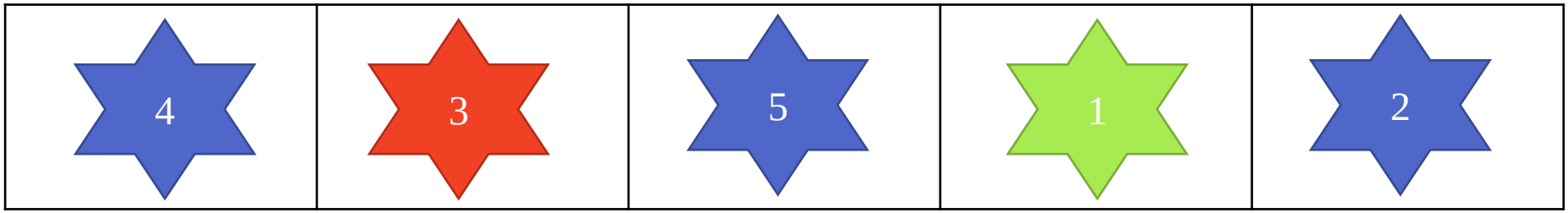
```
List<Integer> numbers = new ArrayList<Integer>();
for (int i = 0; i < 100; i++) {
numbers.add(i);
}

System.out.println(numbers.stream().filter(n -> n % 2 ==
0).collect(Collectors.toList()));
```

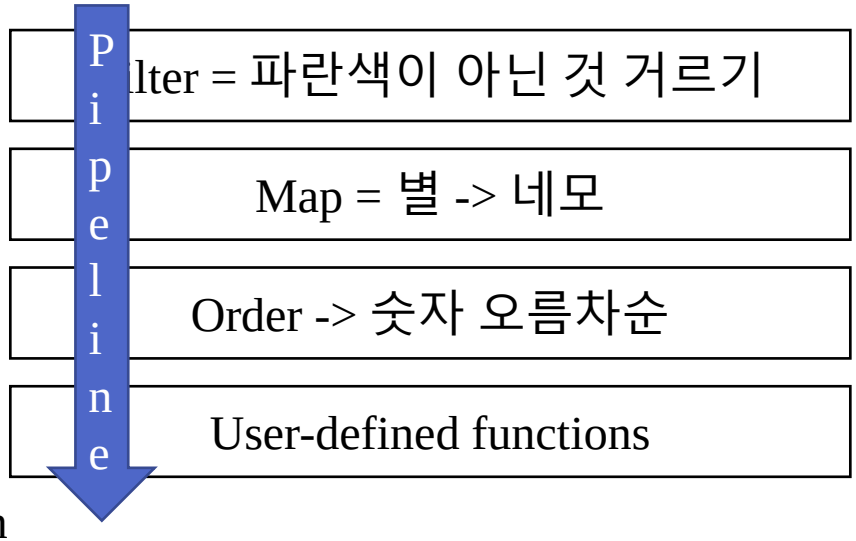
Filter odd numbers!

Anatomy of (Parallel) Stream

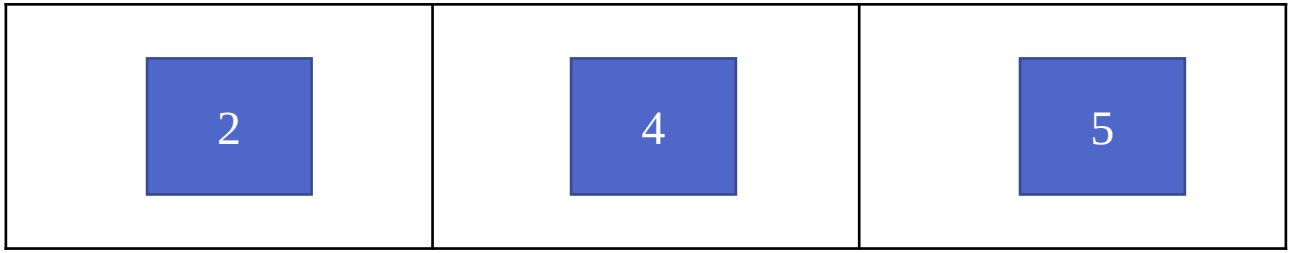
source



Intermediate Operations

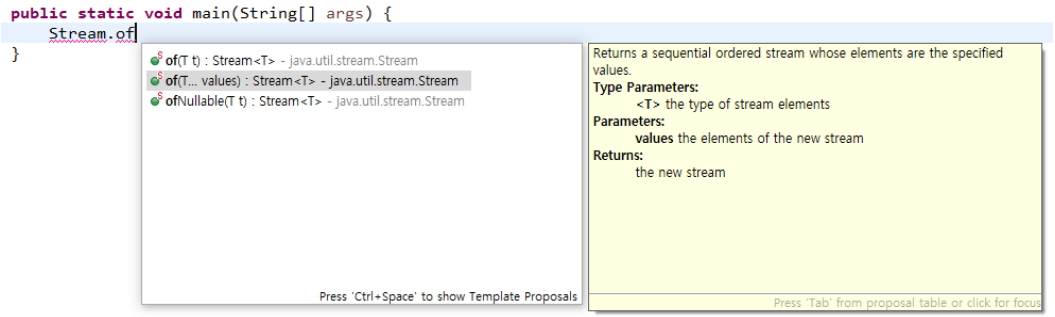


destination



Getting Streams

- From individual values
  - Stream.of(val1, val2, …)



- From array
  - Arrays.stream(someArray)

```
double[] doubleArray = new double[5];
doubleArray[0] = 1.1d;
doubleArray[1] = 2.0d;
doubleArray[2] = 3.2d;
doubleArray[3] = 4.5d;
doubleArray[4] = 2.5d;

Arrays.stream(doubleArray).forEach(e -> System.out.println(e));
```

Getting Streams

- From Collection
  - ArrayList

```
ArrayList<String> al = new ArrayList<String>();
al.add("Hello");
al.add("Data");
al.add("Analysis");
al.stream().forEach(e -> System.out.println(e));
```

  - HashSet

```
HashSet<Long> hs = new HashSet<Long>();
hs.add(1l);
hs.add(3l);
hs.add(3l);
hs.add(5l);
hs.stream().forEach(e -> System.out.println(e));
```

Getting Streams

- From Collection
  - HashMap

```
HashMap<Integer, String> hm = new HashMap<Integer, String>();
hm.put(5, "Hello");
hm.put(7, "Data");
hm.put(9, "Analysis");
hm.entrySet().stream().forEach(e ->
System.out.println(e.getKey() + " " + e.getValue()));
```

  - TreeSet?

  - TreeMap?

Functional Interfaces used in Stream

- Consumer<T>
  - Represents an operation that accepts a single input and returns no result
  - Functional method is void accept(T t)
  - Performs this operation on the given argument (T t)
- e.g., forEach method accepts implementation of Consumer

```
public class MyConsumer<E> implements Consumer<E> {
  @Override
  public void accept(E t) {
    System.out.println(t);
  }
}

hs.stream().forEach(new MyConsumer());
```

```
void java.util.stream.Stream.forEach(Consumer<? super Entry<Integer, String>> action)

Performs an action for each element of this stream.

This is a terminal operation.
```

Functional Interfaces used in Stream

- Predicate<T>
  - Represents a predicate (boolean-valued function) of one argument
  - Functional method is boolean Test(T t)
  - Evaluates this Predicate on the given input argument (T t)
  - Returns true if the input argument matches the predicate, otherwise false
- e.g., filter method accepts implementation of Predicate

```
public static void main(String[] args) {
    List<Integer> numbers = new ArrayList<Integer>();
    for (int i = 0; i < 100; i++) {
        numbers.add(i);
    }

    System.out.println(numbers.stream().filter(n -> n % 2 == 0).collect(Collectors.toList()));
}
```

Stream<Integer> java.util.stream.Stream.filter(Predicate<? super Integer> predicate)

Returns a stream consisting of the elements of this stream that match the given predicate.

This is an intermediate operation.

Parameters:

predicate a non-interfering, stateless predicate to apply to each element to determine if it should be included

Returns:

the new stream

Functional Interfaces used in Stream

- Supplier<T>
  - Represents a supplier of results
  - Functional method is T get()
  - Returns a result of type T
- e.g., Collectors.toCollection accepts implementation of Supplier

```java
/**
 * Represents a supplier of results.
 *
 * <p>There is no requirement that a new or distinct result be returned each
 * time the supplier is invoked.
 *
 * <p>This is a <a href="package-summary.html">functional interface</a>
 * whose functional method is {@link #get()}.
 *
 * @param <T> the type of results supplied by this supplier
 *
 * @since 1.8
 */
@FunctionalInterface
public interface Supplier<T> {

    /**
     * Gets a result.
     *
     * @return a result
     */
    T get();
}
```

```java
HashSet<Integer> arr = numbers.parallelStream().filter(n -> n % 2 == 0)
        .collect(Collectors.toCollection(HashSet<Integer>::new));

for (Integer elem : arr)
    System.out.println(elem)
```

<Integer, Collection<Integer>> Collector<Integer, ?, Collection<Integer>>
java.util.stream.Collectors.toCollection(Supplier<Collection<Integer>> collectionFactory)

Returns a `Collector` that accumulates the input elements into a new `Collection`, in encounter order. The `Collection` is created by the provided factory.

Type Parameters:
<T> the type of the input elements
<C> the type of the resulting `Collection`

Parameters:
collectionFactory a supplier providing a new empty `Collection` into which the results will be inserted

Returns:
a `Collector` which collects all the input elements into a `Collection`, in encounter order

Functional Interfaces used in Stream

- Function<T,R>
  - Represents a function that accepts one argument and produces a result
  - Functional method is R apply(T t)
  - Applies this function to the given argument (T t)
    - Returns the function result

java.util.function.Function<T, R>

Represents a function that accepts one argument and produces a result.

This is a functional interface whose functional method is `apply(Object)`.

Type Parameters:

<T> the type of the input to the function

<R> the type of the result of the function

- e.g., map accepts
  implementation of Function

```
public static void main(String[] args) {
    List<Integer> numbers = new ArrayList<Integer>();
    for(int i = 0 ; i < 100 ; i++) {
        numbers.add(i);
    }

    System.out.println(numbers.stream().map(e -> e+1).collect(Collectors.toList()));
}
```

<Integer> Stream<Integer> java.util.stream.Stream.map(Function<? super Integer, ? extends Integer> mapper)

Returns a stream consisting of the results of applying the given function to the elements of this stream.

This is an intermediate operation.

Type Parameters:

<R> The element type of the new stream

Parameters:

mapper a non-interfering, stateless function to apply to each element

Functional Interfaces used in Stream

- Consumer<T>
  - Represents an operation that accepts a single input and returns no result
  - Functional method is void accept(T t)
    - Performs this operation on the given argument (T t)
- E.g., forEach accepts implementation of Consumer

java.util.function.Consumer<T>

Represents an operation that accepts a single input argument and returns no result. Unlike most other functional interfaces, `Consumer` is expected to operate via side-effects.

This is a functional interface whose functional method is `accept(Object)`.

Type Parameters:
<T> the type of the input to the operation

```
public static void main(String[] args) {
    List<Integer> numbers = new ArrayList<Integer>();
    for(int i = 0 ; i < 100 ; i++) {
        numbers.add(i);
    }
    numbers.stream().forEach(e -> System.out.println(e));
}
```

void java.util.stream.Stream.forEach(Consumer<? super Integer> action)

Performs an action for each element of this stream.

This is a terminal operation.

The behavior of this operation is explicitly nondeterministic. For parallel stream pipelines, this operation does *not* guarantee to respect the encounter order of the stream, as doing so would sacrifice the benefit of parallelism. For any given element, the action may be performed at whatever time and in whatever thread the library chooses. If the action accesses shared state, it is responsible for providing the required synchronization.

Functional Interfaces used in Stream

- BiFunction<T,U,R>
  - Represents an operation that accepts two arguments and produces a result
  - Functional method is R apply(T t, U u)
    - Applies this function to the given arguments (T t, U u)
    - Returns the function result

Functional Interfaces used in Stream

- BinaryOperator<T>
  - Extends BiFunction<T, U, R>
  - Represents an operation upon two operands of the same type, producing a result of the same type as the operands
  - Functional method is R BiFunction.apply(T t, U u)
    - Applies this function to the given arguments (T t, U u) where R,T and U are of the same type
    - Returns the function result
- E.g., reduce accepts BinaryOperator

Get Maximum

```
numbers.stream().reduce((e1, e2) -> e1 > e2 ? e1 : e2).ifPresent(e -> System.out.println(e));
```

Optional<Integer> java.util.stream.Stream.reduce(BinaryOperator<Integer> accumulator)

Performs a reduction on the elements of this stream, using an associative accumulation function, and returns an `Optional` describing the reduced value, if any. This is equivalent to:

```
boolean foundAny = false;
      T result = null;
      for (T element : this stream) {
          if (!foundAny) {
              foundAny = true;
              result = element;
          }
          else
              result = accumulator.apply(result, element);
      }
      return foundAny ? Optional.of(result) : Optional.empty();
  }
```

but is not constrained to execute sequentially.

The `accumulator` function must be an associative function.

This is a terminal operation.

Functional Interfaces used in Stream

- Comparator<T>
  - Compares its two arguments for order.
  - Functional method is int compareTo(T o1, T o2)
    - Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.
- E.g., max accepts Comparator

Get Maximum

```
List<Integer> numbers = new ArrayList<Integer>();
for (int i = 0; i < 100; i++) {
    numbers.add(i);
}
numbers.stream().max((e1, e2) -> {
    if(e1 > e2)
        return 1;
    else if(e1 == e2)
        return 0;
    else
        return -1;
}).ifPresent(e -> System.out.println(e));
```

Optional<Integer> java.util.stream.Stream.max(Comparator<? super Integer> comparator)

Returns the maximum element of this stream according to the provided `Comparator`. This is a special case of a reduction.

This is a terminal operation.

Parameters:

comparator a non-interfering, stateless `Comparator` to compare elements of this stream

Returns:

an `Optional` describing the maximum element of this stream, or an empty `Optional` if the stream is empty

Anatomy of the Stream Pipeline

- Elements in a stream go through a pipeline of operations
- A stream starts with a source data structure
- Intermediate methods (Lazy)
  - Start invoked when a terminal method invoked
  - e.g., map, filter, distinct, sorted, peek, limit, parallel

- Terminal methods (Eager)
  - Trigger the processing of a pipeline
  - The pipeline will close
  - e.g., forEach, toArray, reduce, collect, min, max, count, anyMatch, allMatch, noneMatch, findFirst, findAny, iterator

- Short-circuit methods (Eager)
  - Trigger the processing of a pipeline and wait next short-circuit or terminal methods
  - E.g., anyMatch, allMatch, noneMatch, findFirst, findAny, limit

Stream API Methods

- Void forEach(Consumer)
  - Easy way to loop over Stream elements
  - You supply a lambda for forEach and that lambda is called on each element of the Stream
  - Related peek method does the exact same thing, but returns the original Stream

- Practice #2
  - Get 1~100 of ArrayList
  - Print out all the even values

Stream API Methods

- Stream<T> map(Function)
  - Produces a new Stream that is the result of applying a Function to each element of original Stream

- Practice #3 using map and forEach
  - Get 1~100 of ArrayList
  - Increase each element by 1
  - Print out all the odd values

  - 1~100 이 있는 ArrayList
  - 각각의 요소를 1 증가시키고 (map)
  - 홀수만 출력하는 (forEach)
  - 프로그램 작성하세요

Stream API Methods

- Stream<T> map(Function)
  - Produces a new Stream that is the result of applying a Function to each element of original Stream

- Practice #4 using map and forEach
  - Get 1~100 of ArrayList
  - Cast int to char
  - Print out all the characters

  - 1~100의 수를 갖는 integer arraylist
  - Map을 이용해서 int -> char
  - 모든 캐릭터 출력 forEach

Stream API Methods

- Stream<T> filter(Predicate)
  - Produces a new Stream that contains only the elements of the original Stream that pass a given test

- Practice #5 using filter, and forEach
  - Get 1~100 of ArrayList
  - Filter odd number (홀수를 없애라)
  - Print out all the even numbers (짝수를 출력하기)

Stream API Methods

- Stream<T> filter(Predicate)
  - Produces a new Stream that contains only the elements of the original Stream that pass a given test

- Practice #6 using map, filter, and forEach
  - Get 1~100 of ArrayList
  - Append random (0 to 100) number to each element
  - Filter if the value is larger than 100
  - Print out all the numbers

nextInt(5)
0 1 2 3 4

1	2	3	4	5	6
0	30	25	64	33	58
1	32	28	68	38	64

  - 1~100 의 수를 갖는 ArrayList 가 있다.
  - 여기에 map을 통해서 각각의 요소에 0 ~ 100의 랜덤 수를 더하는 것
  - 100보다 크면 지운다 (filter)
  - 모두 출력한다. (forEach)

Stream API Methods

- Optional<T> findFirst()
  - Returns an Optional for the first entry in the Stream

- Practice #7 using map, filter, and forEach
  - Make 100000 random numbers from 0 to 1,0000,0000 to TreeSet
  - Print out first element (smallest)

  - 10만개의 수를 넣는다. 각각의 수는 0~1억 사이의 랜덤 수다
  - 이것을 TreeSet 에 넣는다.

Stream API Methods

- Optional<T> Class
  - A container which may or may not contain a non-null value
  - Common methods
    - isPresent(): return true if value is present
    - get(): return value if present
    - orElse(T other): returns value if present, or other
    - ifPresent(Consumer): runs the lambda if value is present

Stream API Methods

- Object[] toArray(Supplier)
  - Reads the Stream of elements into a an array

- Practice #8 using toArray
  - Make 100000 random numbers from 0 to 100000000 to TreeSet
  - Get Integer[] from TreeSet
  - Print out each element (not stream api)

Stream API Methods

- Object[] toArray(IntFunction)
  - Reads the Stream of elements into a an array

- Practice #8 using toArray
  - Create Email Class
  - Make an email of source and destination with 100000 random numbers from 0 to 100000000 to ArrayList
  - Get Email[] from ArrayList with toArray
  - Print out each element (not stream api)

Stream API Methods

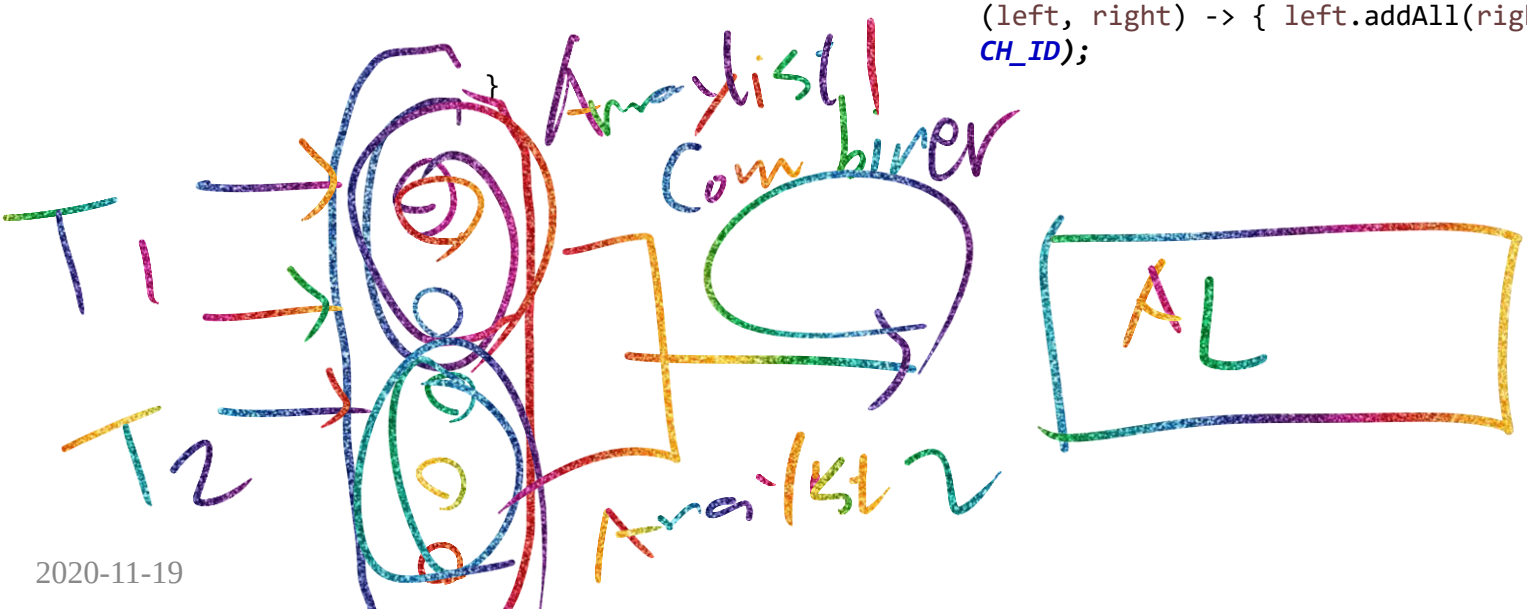
- List<T> Stream.collect(Collectors.toList())
  - Reads the Stream of elements into a List or any other collection

- Practice #9 using map and **Collectors.toList**
  - Make 0 – 99 ArrayList
  - Increase each element by 2
  - Collect arrayList into List

<List<Integer>, Object> List<Integer> java.util.stream.Stream.collect(Collector<? super Integer, Object, List<Integer>> collector)

Performs a mutable reduction operation on the elements of this stream using a Collector. A Collector encapsulates the functions used as arguments to collect(Supplier, BiConsumer, BiConsumer), allowing for reuse of collection strategies and composition of collect operations such as multiple-level grouping or partitioning.

```
Collector<T, ?, List<T>> toList() {
        return new CollectorImpl<>((Supplier<List<T>>) ArrayList::new, List::add,
                                   (left, right) -> { left.addAll(right); return left; },
                                   CH_ID);
}
```



Stream API Methods

- Set<T> collect(Collectors.toSet())
  - Reads the Stream of elements into a Set or any other collection

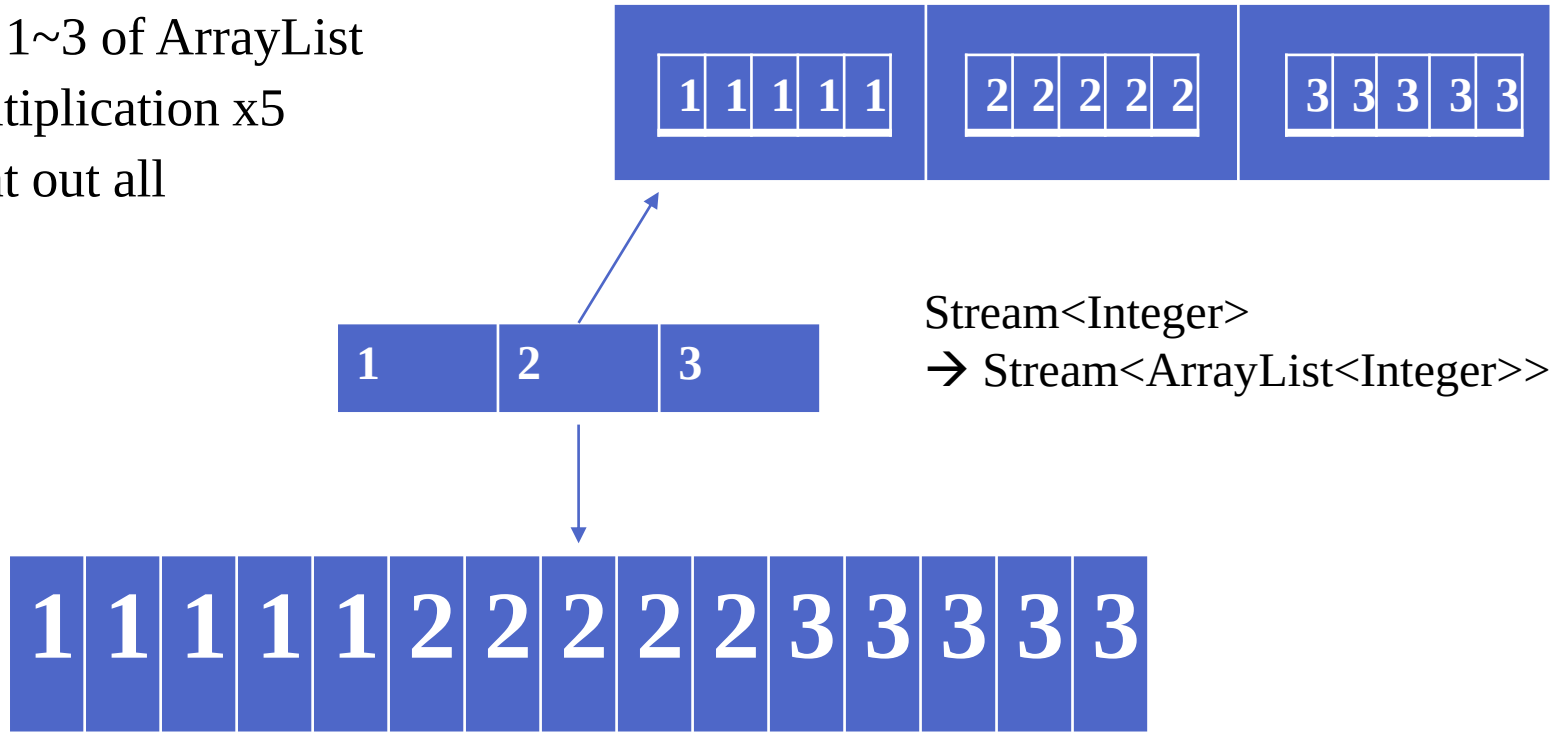
- Practice #10 using map and Collectors.toSet
  - Make 0 – 99 ArrayList
  - Map e to e%10
  - Collect arrayList into Set

```
public static <T>
Collector<T, ?, Set<T>> toSet() {
    return new CollectorImpl<>((Supplier<Set<T>>) HashSet::new, Set::add,
                               (left, right) -> {
                                   if (left.size() < right.size()) {
                                       right.addAll(left); return right;
                                   } else {
                                       left.addAll(right); return left;
                                   }
                               },
                               CH_UNORDERED_ID);
}
```

Stream API Methods

- Stream<T> map(Function)
  - Produces a new Stream that is the result of applying a Function to each element of original Stream

- Practice #11 (motivating example)
  - Get 1~3 of ArrayList
  - Multiplication x5
  - Print out all



Stream<Integer>
→ Stream<ArrayList<Integer>>


1
2
3
4
5
11111
22222
33333
44444
55555

Stream API Methods

- Stream<T> flatMap(Function)
  - Produces a new Stream that is the result of applying a Function to each element of original Stream

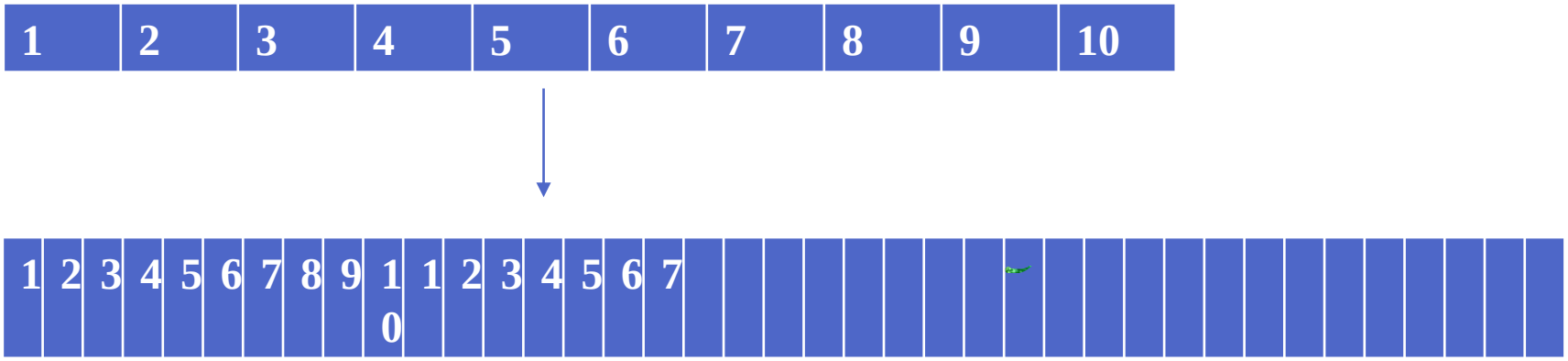
- Practice #12 using flatMap, toList
  - Get 1~10 of ArrayList
  - Multiplication x5
  - Get flattened list!

```
flatMap(e -> {
```

<Integer> Stream<Integer> java.util.stream.Stream.flatMap(Function<? super Integer, ? extends Stream<? extends Integer>> mapper)

Returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.)

This is an intermediate operation.



Stream API Methods

- Stream<T> Stream.limit(long maxSize)
  - Limit(n) returns a stream of the first n elements

- Practice #13 using limit, collect(toList)
  - Get 1~1000 of ArrayList
  - Limit the number of element to 100
  - Print out

Stream API Methods

- Stream<T> skip(long n)
  - skip(n) returns a stream starting with element n

- Practice #14 using skip and toList
  - Get 1~1000 of ArrayList
  - Skip the first 900 elements
  - toList

Stream API Methods

- Stream<T> sorted()
- Stream<T> sorted(Comparator)
  - Returns a stream consisting of the elements of this stream, sorted according to the provided Comparator

- Practice #15 using sorted
  - Get 0~999의 값을 갖는 100 난수를 HashSet에 넣는다
  - Stream -> Sort the set
  - Collect them to List
  - Print out

Stream API Methods

- Stream<T> distinct()
  - Returns a stream consisting of the distinct elements of this stream
  - Remove redundancy

- Practice #16 using distinct
  - Get 0 to 99 of ArrayList
  - Map each element to its %5
  - Use distinct
  - Collect them to list
  - Print out

Stream API Methods

- long count()
  - Returns the count of elements in the Stream

- Practice #17 using sorted
  - Get 100 random 1~1000 of HashSet
  - Print out the number of elements

Wrap-up

- Stream and Parallel Processing 1 - Basic