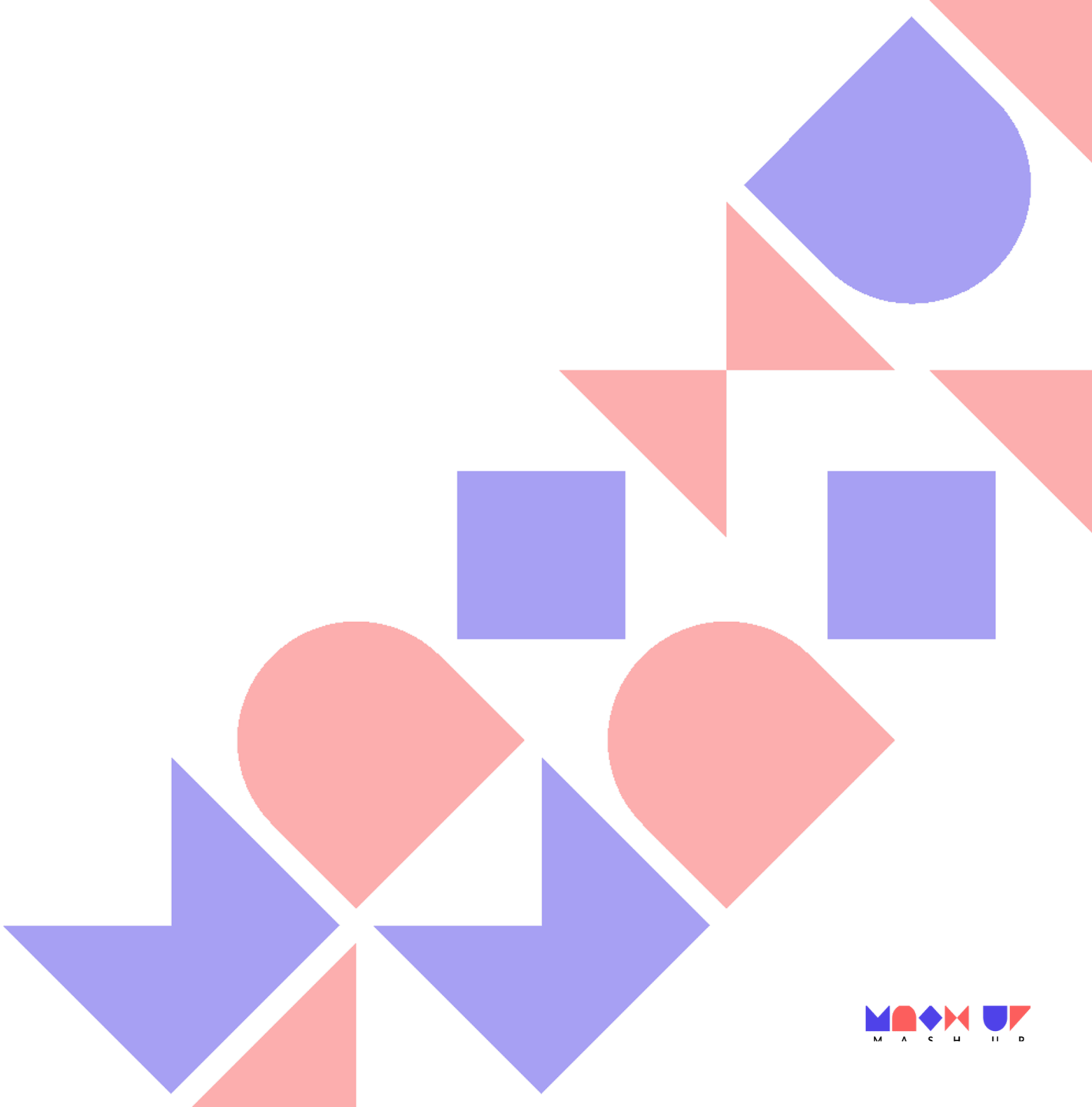
Java 8 추가 문법

박미현
김경훈



CONTENTS

Optional

Optional 이 생긴 이유?

Java 8 Optional

```
class Person {
        private String name;

        // constructor, getter, setter method 생략
}

class House {
        private Person owner;
        private String address;

        // constructor, getter, setter method 생략
}
```

Java 8 Optional

```
void main() {
        House house = houseService.getRandomHouse();
// 이 메서드는 자신이 가지고 있는 집들 중 아무거나 반환합니다. null 은 나오지 않는다고 가정합니다.
        System.out.println(" owner : " + house.getOnwer().getName());
        System.out.println("address: " + house.getAddress());
}
```

```
void main() {
        House house = houseService.getRandomHouse();
        System.out.println(" owner : " + house.getOnwer().getName()); // java.lang.NullPointerException 발생!
        System.out.println("address: " + house.getAddress());
}
```

집에 주인과 주소가 있다면 OK
주인이 없는 집이 있다면 NPE 예외 발생!

대처법은?

Java 8 Optional

```
void main() {
        House house = houseService.getRandomHouse();
        if (house.getOwner() != null) {
                System.out.println("onwer: " + house.getOwner().getName());
        }
        System.out.println("address: " + house.getAddress());
}
```

그러면 새로운 요구 사항을 넣어봅시다. **주인은 있는데 주인의 이름이 없거나, 집 주소가 없는 경우에는 콘솔에 노출하지 않게 해주세요.** 라고 하는 요구 사항을 말이죠. 기존 코드를 활용해서 다시 작성해 봅시다.

Java 8 Optional

```
void main() {
        House house = houseService.getRandomHouse();
        if (house.getOwner() != null && house.getOwner().getName() != null) {
                System.out.println(" owner : " + house.getOwner().getName());
        }
        if (house.getAddress() != null) {
                System.out.println("address: " + house.getAddress());
        }
}
```

```
  void main() {
          House house = houseService.getRandomHouse();
          Optional.of(house)
      .map(House::getOwner)
      .map(Person::getName)
      .ifPresent(name -> System.out.println(" owner :" + name));

          Optional.of(house)
      .map(House::getAdress)
      .ifPresent(address -> System.out.println("address:" + address));  }
```

Java 8 Optional

Optional 객체 생성을 위한 메소드 종류

Optional.of

value가 null 인 경우 NPE 예외를 던집니다.
반드시 값이 있어야 하는 객체인 경우 해당 메서드를 사용하면 됩니다.

```
// 메서드 시그니처
public static <T> Optional<T> of(T value);
// 예제
Optional<String> opt = Optional.of("result");
```

Optional.ofNullable

value가 null 인 경우 비어있는 Optional 을 반환합니다.
값이 null 일수도 있는 것은 해당 메서드를 사용하면 됩니다.

```
// 메서드 시그니처
public static <T> Optional<T> ofNullable(T value);
// 예제
Optional<String> opt = Optional.ofNullable(null);
```

Java 8 Optional

Optinal.empty

비어있는 옵셔널 객체를 생성합니다. 반환할 값이 없는 경우에도 사용됩니다.

```
// 메서드 시그니처
public static<T> Optional<T> empty();
// 예제
Optional<String> emptyOpt = Optional.empty();
```

Java 8 Optional

Optional 중간 처리
옵셔널 객체를 생성한 후 사용 가능한 메서드입니다. 해당 메서드들은 다시 옵셔널을 반환하므로 메서드 체이닝을 통해 원하는 로직을 반복 삽입할 수 있습니다.

filter

predicate 값이 참이면 해당 필터를 통과시키고 거짓이면 통과 되지 않습니다.

```
// 메서드 시그니처
public Optional<T> filter(Predicate<? super T> predicate);
// 예제
Optional.of("True").filter((val) -> "True".eqauls(val)).orElse("NO DATA"); // "True"
Optional.of("False").filter((val) -> "True".eqauls(val)).orElse("NO DATA"); // "NO DATA"
```

Java 8 Optional

Map
mapper 함수를 통해 입력값을 다른 값으로 변환하는 메서드입니다.

```
// 메서드 시그니처
public<U> Optional<U> map(Function<? super T, ? extends U> mapper);
// 예제
Integer test =
Optional.of("1").map(Integer::valueOf).orElseThrow(NoSuchElementException::new);
// string to integer
```

Java 8 Optional

flatMap
mapper 함수를 통해 입력값을 다른 값으로 변환하는 메서드입니다. map() 메서드와 다른점은 메서드 시그니처의 매개변수입니다. map() 에서는 제너릭으로 U를 정의했지만 flatMap() 에서는 제너릭으로 Optional(U)를 정의했습니다.
이것이 뜻하는 바는 flatMap() 메서드가 반환해야 하는 값은 Optional 이라는것입니다.

```
// 메서드 시그니처
public<U> Optional<U> flatMap(Function<? super T, Optional<U>> mapper);
// 예제
String result = Optional.of("result")
        .flatMap((val) -> Optional.of("good"))
        .get();
System.out.println(result); // print 'good’
```

Java 8 Optional

Optional 종단 처리
종단 처리라는 것은 옵셔널 객체의 체이닝을 끝낸다는 것입니다.

ifPresent
최종적으로 연산을 끝낸 후 값이 비어있지 않다면 입력값으로 주어집니다. 이 값을 가지고 원하는 작업을 수행하면 됩니다. 하지만 중간 연산을 하다 비어있는 옵셔널 객체를 받게 되면 .ifPresent() 메서드의 내용을 수행하지 않습니다.

```
// 메서드 시그니처
public void ifPresent(Consumer<? super T> consumer);
// 예제1
Optional.of("test").ifPresent((value) -> {
        // something to do
});
// 예제2 (ifPresent 미수행)
Optional.ofNullable(null).ifPresent((value) -> {
        // nothing to do
});
```

Java 8 Optional

isPresent
최종적으로 연산을 끝낸 후 객체가 존재하는지 여부를 판별합니다.

```
// 메서드 시그니처
public boolean isPresent();
// 예제
Optional.ofNullable("test").isPresent(); // true
Optional.ofNullable("test").filter((val) -> "result".eqauls(val)).isPresent(); // false
```

Get
최종적으로 연산을 끝낸 후 객체를 꺼냅니다. 이 때, 비어있는 옵셔널 객체였다면 예외가 발생합니다.

```
public T get();
// 예제
Optional.of("test").get(); // 'test'
Optional.ofNullable(null).get(); // NoSuchElementException!
```

Java 8 Optional

orElse
최종적으로 연산을 끝낸 후에도 옵셔널 객체가 비어있다면 기본값으로 제공할 객체를 지정합니다.

```
// 메서드 시그니처
public T orElse(T other);
// 예제
String result = Optional.ofNullable(null).orElse("default");
System.out.println(result); // print 'default'
```

orElseGet
최종적으로 연산을 끝낸 후에도 옵셔널 객체가 비어있다면 기본값으로
제공할 공급자 함수Supplier를 지정합니다.

```
// 메서드 시그니처
public T orElseGet(Supplier<? extends T> other);
// 예제
String result = Optional.ofNullable("input").filter("test"::equals).orElseGet(() -> "default");
System.out.println(result); // print 'default'
```

Java 8 Optional

orElseThrow
최종적으로 연산을 끝낸 후에도 옵셔널 객체가 비어있다면 예외 공급자 함수를 통해 예외를 발생시킵니다.

```
// 메서드 시그니처
public <X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)
throws X;
// 예제
Optional.ofNullable("input").filter("test"::equals).orElseThrow(NoSuchElementException::new)
;
```

orElse, orElseGet 중 무엇을 사용할까?

orElse 는 반환할 값을 그대로 받는 반면, *orElseGet* 은 *Supplier* 로 랩핑한 값을 인자로 받는다.
이는 함수를 전달받아 바로 값을 가져오지 않고lazy 필요할 때 값을 가져온다.
즉, *orElseGet* 은 null 일 경우에만 함수가 실행되면서 인스턴스화되지만, *orElse* 는 무조건 인스턴스화 된다.

람다
표현식

Java 8 람다 표현식

람다식이란?

식별자 없이 실행 가능한 함수 표현식

자바에서 람다식을 사용하려면 다음과 같은 방법으로 사용이 가능합니다.

```
( parameters ) -> expression body
( parameters ) -> { expression body }
() -> { expression body }
() -> expression body
...
```

Java 8 람다 표현식

```
// Thread - traditional
new Thread(new Runnable() {
        @Override
        public void run() {
                System.out.println("Hello World.");
        }
}).start();
```

```
// Thread - Lambda Expression
new Thread(()->{
        System.out.println("Hello World.");
}).start();
```

Java 8 람다 표현식

장점 :

- 1) 코드의 라인수가 줄어든다

: 자바의 verbose(시끄럽고, 형식적인면)한 특성을 감소시킬수 있는 큰 장점이다.

- 2) 병렬 프로그래밍이 가능하다.

: iteration방식은 반복대상을 일일히 루프에서 지정하는 반면에,
함수형 프로그래밍에서는 반복대상을 사용자코드에서 직접 지정하지 않는다.
이로 인해 Collection API가 크게 효과적으로 개선되었다.

- 3) 메소드로 행동방식을 전달할수 있다.

: 자바 메소드로 값이나 객체를 생성하여 전달하여 전달하던 예전방식과 달리,
자바람다에서는 행동방식 그자체를 람다식으로 구현하여 전달한다.

- 4) 의도의 명확성

: 프로그래밍에서 중요한 요소중에 하나는 가독성이 높은 프로그램이다.
그러기 위해서는 코드에 작성자의 의도가 명확히 드러나도록 하는 것이 중요하다.
람다식을 사용하면 코드에서 드러내고자하는 개발자의 의도를 응축적이면서
추상화시켜서 나타낼수 있게 하기때문에 의도를 명확하게 하고,
버그가 적은 코드를 작성하게 하는데에 도움을 준다.

Java 8 람다 표현식

단점 :

- 1) 람다식의 호출을 위해 직접 메소드를 불러야 한다.

: 람다식을 생성하여 다른 함수의 파라미터로 전달하는 것은 자연스러우나,
람다식을 실행할때에는 인터페이스에 선언된 메소드를 호출하여야 한다.

- 2) 재귀 람다식의 호출이 까다롭다.

: 람다식안에서 자신을 다시 호출하기가 용이하지 않다.
람다식안에서는 람다식을 가리키는 변수를 참조할수가 없다.
배열등의 트릭을 사용하면 가능하기는 한다.

- 3) 클로저가 지원되지 않는다.

: 일반적인 함수형 프로그램에서는 람다식안에서 가리키는 외부 변수에 대해
클로져형태로서 외부변수의 라이프 사이클을 연장할수 있지만,
자바에서는 외부변수에 대해 사실상 final 형태로서만 참조할수 있다.

- 4) 함수 외부의 값을 변경한다.

: 순수함수형 프로그래밍 방식에서는 함수 내부의 동작은 항상 블랙박스이며
함수에 대해 동일한 입력을 하게 되면 동일한 출력을 하게 된다.
그러나 자바는 기본적으로 객체모델로서 함수안의 값이 다른 객체의 영향을 받는 경우,
같은 입력값에 대해 다른 출력값을 출력할수있다.
이러한 면은 수학적 프로그래밍, 병렬형 프로그래밍에 불리한 방식일수 있다.

Java 8 람다 표현식

Stream API

Stream은 자바 8부터 추가된 기능으로 **"컬렉션, 배열등의 저장 요소를 하나씩 참조하며 함수형 인터페이스(람다식)를 적용하며 반복적으로 처리할 수 있도록 해주는 기능"**이다.

먼저 Stream API를 사용하려면 stream을 얻어와야 합니다. 얻는 방법은 다음과 같습니다.

```
Arrays.asList(1,2,3).stream(); // (1)
Arrays.asList(1,2,3).parallelStream(); // (2)
```

Working Stream

실제로 얻어온 stream에 연산을 해봅시다. 주요하게 쓰이는 몇가지 API만 살펴봅시다.

forEach

stream의 요소를 순회해야 한다면 forEach를 활용할 수 있습니다.

```
Arrays.asList(1,2,3).stream()
        .forEach(System.out::println); // 1,2,3
```

Java 8 람다 표현식

Map

stream의 개별 요소마다 연산을 할 수 있습니다.
아래의 코드는 리스트에 있는 요소의 제곱 연산을 합니다.

```
Arrays.asList(1,2,3).stream()
            .map(i -> i*i)
            .forEach(System.out::println); // 1,4,9
```

Limit

stream의 최초 요소부터 선언한 인덱스까지의 요소를 추출해 새로운 stream을 만듭니다.

```
Arrays.asList(1,2,3).stream()
                    .limit(1)
                    .forEach(System.out::println); // 1
```

Java 8 람다 표현식

Skip

stream의 최초 요소로부터 선언한 인덱스까지의 요소를 제외하고 새로운 stream을 만듭니다.

```
Arrays.asList(1,2,3).stream()
        .skip(1)
        .forEach(System.out::println); // 2,3
```

Filter

stream의 요소마다 비교를 하고 비교문을 만족하는 요소로만 구성된 stream을 반환합니다.

```
Arrays.asList(1,2,3).stream()
        .filter(i-> 2>=i)
        .forEach(System.out::println); // 1,2
```

flatMap

stream의 내부에 있는 객체들을 연결한 stream을 반환합니다.

```
Arrays.asList(Arrays.asList(1,2),Arrays.asList(3,4,5),Arrays.asList(6,7,8,9)).stream()
                                                    .flatMap(i -> i.stream())
                                                    .forEach(System.out::println); // 1,2,3,4,5,6,7,8,9
```

Reduce
stream을 단일 요소로 반환합니다.

```
Arrays.asList(1,2,3).stream()
                    .reduce((a,b)-> a-b)
                    .get(); // -4
```

우선, 첫번째 연산으로 1과 2가 선택되고 계산식은 앞의 값에서 뒤의 값을 빼는 것이기 때문에 결과는 -1이 됩니다.
그리고 이상태에서 -1과 3이 선택되고 계산식에 의해 -1-3이 되기 때문에 결과로 -4가 나옵니다.
뒤로 추가 요소가 있다면 차근차근 앞에서부터 차례대로 계산식에 맞춰 계산하면 됩니다.

Java 8 람다 표현식

Collection

아래의 코드들은 각각의 메소드로 콜렉션 객체를 만들어서 반환합니다.

```
Arrays.asList(1,2,3).stream()
                    .collect(Collectors.toList());
Arrays.asList(1,2,3).stream()
                    .iterator();
```

DATE API

기존 자바 Date API의 문제점

java.util.Calendar / java.util.Date

1. Mutable

Person p = new Person();

Date birthDay = p.getBirthDay();

birthDay.setTime();

```
public static void main(String[] args) {
    Calendar today = Calendar.getInstance(TimeZone.getTimeZone("UTC"));
    System.out.println("Tomorrow: " + tomorrow(today));

    System.out.println("Today: " + today.getTime());
}

private static Date tomorrow(Calendar today){

    today.add(Calendar.DATE,  amount: 1);
    return today.getTime();
}
```

```
Tomorrow: Sat Aug 29 23:38:21 KST 2020
Today: Sat Aug 29 23:38:21 KST 2020
```

기존 자바 Date API의 문제점

java.util.Calendar / java.util.Date

2. 헷갈리는 Month

MONTH

```
public static final int MONTH
```

Field number for `get` and `set` indicating the month. This is a calendar-specific value. The first month of the year in the Gregorian and Julian calendars is `JANUARY` which is 0; the last depends on the number of months in a year.

See Also:

`JANUARY`, `FEBRUARY`, `MARCH`, `APRIL`, `MAY`, `JUNE`, `JULY`, `AUGUST`, `SEPTEMBER`, `OCTOBER`, `NOVEMBER`, `DECEMBER`, `UNDECIMBER`, `Constant Field Values`

기존 자바 Date API의 문제점

java.util.Calendar / java.util.Date

3. 제멋대로 요일 상수

```
    public static void main(String[] args) {
        Calendar calendar = Calendar.getInstance();
        System.out.println(calendar.getTime());
        System.out.println("Calendar의 DAY_OF_WEEK : " + calendar.get(Calendar.DAY_OF_WEEK));
        System.out.println("Date의 DAY_OF_WEEK : " + calendar.getTime().getDay());
    }
}
```

```
"C:\Program Files\Java\jdk-12.0.2\bin\java.exe" -javaagent:C:\Users\hyoni\AppData\Local\JetBra
Sat Aug 29 03:08:17 KST 2020
Calendar의 DAY_OF_WEEK : 7
Date의 DAY_OF_WEEK : 6
```

기존 자바 Date API의 문제점

java.util.Calendar / java.util.Date

4. String타입의 TimeZone

```java
public class Java8Test {
    @Test
    public void wrongTimeZond() {
        TimeZone zone = TimeZone.getTimeZone("Seoul/Asia");
        assertThat(zone.getID()).isEqualTo("GMT");
    }
}
```

Java8Test.wrongTimeZond

Tests passed: 1 of 1 test – 284 ms

Test Results 284 ms

```
Testing started at 오전 12:56 ...

> Task :compileJava UP-TO-DATE
> Task :processResources UP-TO-DATE
> Task :classes UP-TO-DATE
> Task :compileTestJava
> Task :processTestResources UP-TO-DATE
> Task :testClasses

> Task :test
```

LocalDate / LocalTime

java.time.LocalDate / java.time.LocalTime

▷ 현대에서 사람들이 사용하는 시간 나타냄.

▷ 시간대 저장, 표현 X → 시간대 필요 없는 작업에 사용

▷ 직관적인 연산 메서드 지원

▷ 불변의 날짜-시간 객체

▷ 요일 클래스는 Enum상수 제공

▷ 잘못된 Month값을 가진 객체 생성 시점에서 IllegalFieldValueException

LocalDate / LocalTime

java.time.LocalDate / java.time.LocalTime

```
@org.junit.Test
public void test() {
    LocalDate date = IsoChronology.INSTANCE.date( prolepticYear: 2020,  month: 8,  dayOfMonth: 29);
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");
    LocalDate nextDate = date.plusDays(1);
    assertThat(nextDate.format(formatter)).isEqualTo("2020-08-30");
    assertThat(date.format(formatter)).isEqualTo("2020-08-29");
    assertThat(date.getDayOfWeek()).isEqualTo(DayOfWeek.SATURDAY);
}
}
```

```
Test.test
Tests passed: 1 of 1 test – 87 ms
Test Results  87 ms
Testing started at 오전 3:40 ...
```

ZonedDateTime

java.time.ZonedDateTime

▷ 시간대를 가진 날짜 시간 표현

▷2007-12-03T10:15:30+01:00 Europe/Paris

▷ LocalDateTime에 타임존 추가/제거 통한 상호변환 가능

▷ DateTimeFormatter사용하여 원하는 패턴으로 변환 가능

▷ 문자열로 상호변환 가능

▷ 잘못 지정된 시간대 ID에 ZoneRulesException

ZonedDateTime

java.time.ZonedDateTime

```java
public class Test {
    @org.junit.Test
    public void test() {
        ZonedDateTime nowSeoul = ZonedDateTime.now(ZoneId.of("Asia/Seoul"));
        System.out.println("Asia/Seoul : " + nowSeoul);
        LocalDateTime nowUTC = nowSeoul.withZoneSameInstant(ZoneId.of("UTC")).toLocalDateTime();
        System.out.println("UTC : " + nowUTC);
        System.out.println(nowSeoul.format(DateTimeFormatter.ofPattern("yyyy.MM.dd HH:mm:ss z")));
    }
}
```

```
Test.test
Tests passed: 1 of 1 test – 53 ms
Test Results 53 ms
> Task :dionysos-api:test

Asia/Seoul : 2020-08-29T04:10:08.042655400+09:00[Asia/Seoul]
UTC : 2020-08-28T19:10:08.042655400
2020.08.29 04:10:08 KST
```

Stream

Stream

java.util.stream

▷ Stream = ‘데이터의 흐름‘

▷ 장점

§ 배열, 컬렉션 인스턴스에 함수 조합 + 결과 필터링 하여 가공된 결과

§ 람다 활용하여 간결한 코드

§ multi-threading 통해 많은 요소를 빠르게 처리

▷ 생성 → 매핑 → 필터링 → 결과 만들기 → 결과물

Stream

생성하기 → 가공하기 → 결과 만들기

▷ 스트림 생성하기

- 배열 스트림

  -stream(T[] array)

  -stream(T[] array, int startIdx, int endIdx)

- 컬렉션 스트림

  - 컬렉션인스턴스.stream();

  - 컬렉션인스턴스.parallelStream();

Stream

생성하기 → 가공하기 → 결과 만들기

- Stream.generate(Supplier<T> s)
- Stream.iterate(final T seed, final UnaryOperator<T> f)
    - seed : 초기값
    - f : 초기값을 다루는 람다

Stream

생성하기 → 가공하기 → 결과 만들기

▷ Filter : 스트림 내 요소 하나씩 평가해 걸러내기

-Stream<String> stream = cities.stream().filter(city -> city.contains(“a”));

▷ Map : 스트림 내 요소 하나씩 특정 값으로 변환

-Stream<String> stream = cities.stream().map(String::toUpperCase);

-Stream<Integer> stream = cities.stream().map(City::getPpl);

Stream

생성하기 → 가공하기 → 결과 만들기

▷ Sorted : 정렬

-Stream<String> stream = cities.stream().sorted();

-Stream<String> stream =
cities.stream().sorted(Comparator.reverseOrder);

▷ Peek : 스트림 내 요소 각각 대상 연산 수행

-int sum = IntStream.of(3, 24).peek(System.out,println).sum();

Stream

생성하기 → 가공하기 → 결과 만들기

▷ Calculating

- count, sum, …

- min, max, …

▷ Reduction

- 인자 1개

```
OptionalInt reduced =
IntStream.range(1, 4)
.reduce((a, b) -> {return
Integer.sum(a, b);};
```

- 인자 2개

```
Int reduced =
IntStream.range(1, 4)
.reduce(10, Integer::sum);
```

- 인자 3개

```
Integer reduced =
IntStream.of(1,2,3)
.reduce(10, Integer::sum,
(a, b) ->
{System.out.println(“Hi”)
return a+b;});
```

Stream

생성하기 → 가공하기 → 결과 만들기

▷ Collect : collect메소드는 Collector타입 인자 받아 처리

- .collect(Collectors.toList()) / Collectors.joining()
- groupingBy()

▷ Matching

-anyMatch : 하나라도 조건 만족하는 요소 있는가

-allMatch : 모두 조건을 만족하는가

-noneMatch : 모두 조건을 불만족하는가

Stream

Lazy Invocation

▷ 지연 처리

▷ 동작 순서

- 한 요소가 모든 파이프라인 거쳐서 결과 만들어내고 다른 요소 동작

▷ 최종 연산이 실행될 때, 중개 연산들의 실행이 강제된다.

▷ 장점

- 미리 계산하여 반복적으로 순회하는 일을 방지

Stream

Lazy Invocation

```
private static long counter;
private static void addOne() {
    ++counter;
}
public static void main(String[] args) {
    List<String> list = Arrays.asList("Seoul", "JeJu", "DokDo");
    Stream<String> stream = list.stream().filter(name -> {addOne(); return name.contains("J");});
    System.out.println(counter);
}
```

```
"C:\Program Files\Java\jdk-1
0
```

```
public static void main(String[] args) {
    List<String> list = Arrays.asList("Seoul", "JeJu", "DokDo");
    List<String> list2 = list.stream().filter(name -> {addOne(); return name.contains("J");})
            .collect(Collectors.toList());
    System.out.println(counter);
}
```

```
"C:\Program Files
3
```

Stream

스트림 재사용 불가

▷ 스트림은 요소를 보관하지 않는다.

(스트림은 저장된 데이터 꺼내 처리하는 용도!)

▷ 최종 연산을 하지 않으면 하나의 인스턴스로 계속 사용가능

▷ 최종 연산 하는 순간 스트림이 닫힌다 → 재사용 불가!

- Runtime Exception 발생

THANKS FOR WATCHING

감사합니다