Stream API

1. Stream 왜 쓰지?

- Java SE 8부터 추가된 스트림 API는 입출력의 스트림과는 전혀 다른 개념이다.
- 많은 양의 데이터를 저장하기 위해서 배열이나 컬렉션을 사용한다.
- 이렇게 저장된 데이터에 접근하기 위해서는 반복문이나 반복자(iterator)를 사용하여 매번 새로운 코드를 작성해야 한다.
- 하지만 이렇게 작성된 코드는 길이가 너무 길고 가독성도 떨어지며, 코드의 재사용이 거의 불가능하다.
- 즉, 데이터베이스의 쿼리와 같이 정형화된 처리 패턴을 가지지 못했기에 데이터마다 다른 방법으로 접근해야만 했다.
- 이러한 문제점을 극복하기 위해서 Java SE 8부터 스트림(stream) API를 도입한다.
- 스트림 API는 데이터를 추상화하여 다루므로, 다양한 방식으로 저장된 데이터를 읽고 쓰기 위한 공통된 방법을 제공한다.
- 따라서 스트림 API를 이용하면 배열이나 컬렉션뿐만 아니라 파일에 저장된 데이터도 모두 같은 방법으로 다룰 수 있게 된다.

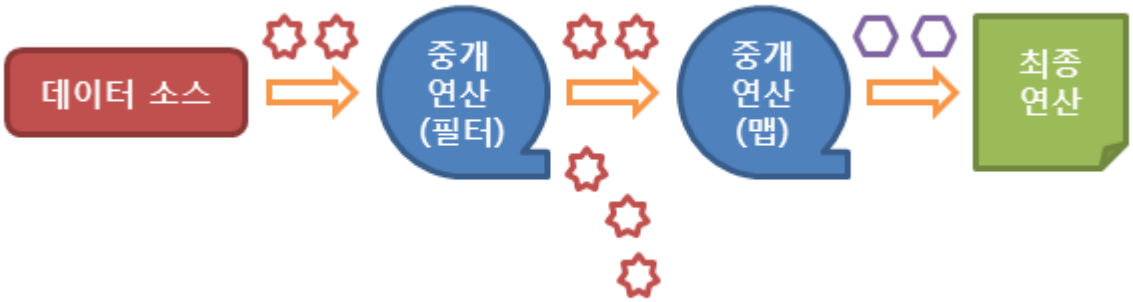
2. 스트림 API의 특징 – 컬렉션을 사용할 때랑 뭐가 다를까?

- 외부 반복을 통해 작업하는 컬렉션과는 달리 내부 반복(internal iteration)을 통해 작업을 수행한다.
  - 외부 반복 : 컬렉션 인터페이스를 사용하기 위해 사용자가 직접 반복해야 한다. 병렬성 관리를 스스로 해야 한다. -> Synchronized 처리
  - 내부 반복 : 반복을 알아서 처리해주고 결과 스트림 값을 어딘가에 저장해 준다. 데이터 표현과 하드웨어를 활용하는 병렬성 구현을 자동으로 선택.
- 재사용이 가능한 컬렉션과는 달리 재사용 불가.

- 원본 데이터 유지.
- 연산은 필터-맵(filter-map) 기반의 API를 사용하여 지연(lazy) 연산을 통해 성능을 최적화 한다.
- parallelStream() 메소드를 통한 손쉬운 병렬 처리를 지원한다.

3. 스트림 API의 동작 흐름

- 스트림의 생성
- 스트림의 중개 연산 (스트림의 변환)
- 스트림의 최종 연산 (스트림의 사용)
- 다음 그림은 자바 스트림 API가 동작하는 흐름을 나타낸다.



4. 스트림 어떻게 만들어?

1) 컬렉션

- 모든 컬렉션의 최고 상위 조상인 **Collection** 인터페이스에는 stream() 메소드가 정의되어 있다.
- 따라서 Collection 인터페이스를 구현한 **모든 List와 Set 컬렉션 클래스에서도 stream() 메소드로 스트림을 생성할 수 있다.**

- 또한, parallelStream() 메소드를 사용하면 병렬 처리가 가능한 스트림을 생성할 수 있다.

```
public void solution() {
        List<Integer> list = new ArrayList<Integer>();
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(4);

        Stream<Integer> stream = list.stream();
        stream.forEach(System.out::print); // 1 2 3 4
}
```

2) 배열

- 배열에 관한 스트림을 생성하기 위해 **Arrays** 클래스에는 다양한 형태의 stream() 메소드가 클래스 메소드로 정의되어 있다.
- 또한, 기본 타입인 int, long, double 형을 저장할 수 있는 배열에 관한 스트림이 별도로 정의되어 있다.
- 이러한 스트림은 java.util.stream 패키지의 IntStream, LongStream, DoubleStream 인터페이스로 각각 제공된다.

```
public void solution2() {
        String[] arr = {"호랑이", "형님", "재밌어!!!"};
        Stream<String> stream = Arrays.stream(arr);
        stream.forEach(x->System.out.print(x+" ")); // 호랑이 형님 재밌어!!!
}
```

3) 가변 매개변수

- Stream 클래스의 of() 메소드를 사용하면 가변 매개변수(variable parameter)를 전달받아 스트림을 생성할 수 있다.

```
public void solution3() {
   Stream<Double> stream = Stream.of(1.1, 1.2, 1.3);
   stream.forEach(System.out::println);  // 1.1  1.2  1.3
}
```

4) 지정된 범위의 연속된 정수

- 지정된 범위의 연속된 정수를 스트림으로 생성하기 위해 IntStream나 LongStream 인터페이스에는 range()와 rangeClosed() 메소드가 정의되어 있다.
- range() 메소드는 명시된 시작 정수를 포함하지만, 명시된 마지막 정수는 포함하지 않는 스트림을 생성한다.
- rangeClosed() 메소드는 명시된 시작 정수뿐만 아니라 명시된 마지막 정수까지도 포함하는 스트림을 생성한다.

```
public void solution4() {
        IntStream intStream = IntStream.range(1, 3);
        intStream.forEach(System.out::print);  // 1 2

        IntStream intStream2 = IntStream.rangeClosed(1, 3);
        intStream2.forEach(System.out::print);  // 1 2 3
}
```

5) 특정 타입의 난수들

- 특정 타입의 난수로 이루어진 스트림을 생성하기 위해 **Random 클래스에는 ints(), longs(), doubles()**와 같은 메소드가 정의되어 있다.
- 이 메소드들은 매개변수로 스트림의 크기를 long 타입으로 전달받을 수 있다.
- 이 메소드들은 만약 매개변수를 전달받지 않으면 크기가 정해지지 않은 무한 스트림(infinite stream)을 반환한다.
- 이 때에는 limit() 메소드를 사용하여 따로 스트림의 크기를 제한해야 한다.

```
public void solution5() {
        IntStream intStream = new Random().ints(4);
        intStream.forEach(System.out::println);
        // -1176164693
        // 1243258950
        // 1813706404
        // -396676677

        IntStream intStream2 = new Random().ints();
        intStream2.forEach(System.out::println);
        // -1176164693
        // ...
        // -396676677
}
```

6) 람다 표현식

- 람다 표현식을 매개변수로 전달받아 해당 람다 표현식에 의해 반환되는 값을 요소로 하는 무한 스트림을 생성하기 위해 Stream 클래스에는 iterate()와 generate() 메소드가 정의되어 있다.
- iterate() 메소드는 시드(seed)로 명시된 값을 람다 표현식에 사용하여 반환된 값을 다시 시드로 사용하는 방식으로 무한 스트림을 생성한다.
- 반면에 generate() 메소드는 매개변수가 없는 람다 표현식을 사용하여 반환된 값으로 무한 스트림을 생성한다..

```
public void solution6() {
        Stream<Integer> stream = Stream.iterate(2,n -> n+2).limit(3);
        stream.forEach(System.out::println);  // 2 4 6

        Stream<Integer> stream2 = Stream.generate(() -> 123);
        stream2.forEach(System.out::println);  // 123 … 123
    }
```

7) 파일

- 파일의 한 행(line)을 요소로 하는 스트림을 생성하기 위해 java.nio.file.Files 클래스에는 lines() 메소드가 정의되어 있다.
- 또한, java.io.BufferedReader 클래스의 lines() 메소드를 사용하면 파일뿐만 아니라 다른 입력으로부터도 데이터를 행(line) 단위로 읽어 올 수 있다.

```
public void solution7() {
        Stream<String> stream = Files.lines(Path path);
    }
```

5. 스트림의 중개 연산(intermediate operation)

- 스트림 API에 의해 생성된 초기 스트림은 중개 연산을 통해 또 다른 스트림으로 변환된다.

- 이러한 중개 연산은 스트림을 전달받아 스트림을 반환하므로, 중개 연산은 연속으로 연결해서 사용할 수 있다.
- 또한, 스트림의 중개 연산은 필터-맵(filter-map) 기반의 API를 사용함으로 지연(lazy) 연산을 통해 성능을 최적화할 수 있다.
- 스트림 API에서 사용할 수 있는 대표적인 중개 연산과 그에 따른 메소드는 다음과 같다..

  1) 스트림 필터링 : filter(), distinct()

  2) 스트림 변환 : map(), flatMap()

  3) 스트림 제한 : limit(), skip()

  4) 스트림 정렬 : sorted()

1) 스트림 필터링

- filter() 메소드는 해당 스트림에서 주어진 조건(predicate)에 맞는 요소만으로 구성된 새로운 스트림을 반환한다.
- 또한, distinct() 메소드는 해당 스트림에서 중복된 요소가 제거된 새로운 스트림을 반환한다.
- distinct() 메소드는 내부적으로 Object 클래스의 equals() 메소드를 사용하여 요소의 중복을 비교한다.

```
public void solution() {
        Stream<Integer> stream = Stream.of(1,1,2,2,3,3);
        stream.distinct().forEach(System.out::println);  // 1 2 3

        Stream<Integer> stream2 = Stream.of(1,2,3,4,5,6);
        stream2.filter(e-> e%2!=0 ).forEach(System.out::println);  // 1 3 5
}
```

2) 스트림 변환

- map() 메소드는 해당 스트림의 요소들을 주어진 함수에 인수로 전달하여, 그 반환값들로 이루어진 새로운 스트림을 반환힌다.

- 만약 해당 스트림의 요소가 배열이라면, flatMap() 메소드를 사용하여 각 배열의 각 요소의 반환값을 하나로 합친 새로운 스트림을 얻을 수 있다.

```
public void solution2() {
        // stream 객체에서 숫자를 Integer 출력
        Stream<String> stream = Stream.of("일", "이이", "삼삼삼");
        stream.map(e -> e.length()).forEach(System.out::println);  // 1 2 3

       // stream 객체에서 새로운 stream 객체를 출력
        String[] arr = {"남자: 지용운, 최현웅, 이경호", "여자: 김수정, 송미정"};
        Stream<String> stream2 = Arrays.stream(arr);

       stream2.flatMap(s -> Stream.of(s.split(":")))
           .forEach(System.out::println);
        // 남자
        //  지용운, 최현웅, 이경호
        // 여자
        //  김수정, 송미정
}
```

3) 스트림 제한

- limit() 메소드는 해당 스트림의 첫 번째 요소부터 전달된 개수만큼의 요소만으로 이루어진 새로운 스트림을 반환한다.

- skip() 메소드는 해당 스트림의 첫 번째 요소부터 전달된 개수만큼의 요소를 제외한 나머지 요소만으로 이루어진 새로운 스트림을 반환한다.

```
public void solution3() {
        Stream<Integer> stream = Stream.of(1,2,3,4,5,6);
        stream.skip(3).forEach(System.out::println);  // 4 5 6

        Stream<Integer> stream2 = Stream.of(1,2,3,4,5,6);
        stream2.limit(2).forEach(System.out::println);  // 1 2

        // stream.limit(1).forEach(System.out::println);
        // stream 재사용 불가 : stream has already been operated upon or closed
        Stream<Integer> stream3 = Stream.of(1,2,3,4,5,6);
        stream3.skip(3).limit(1).forEach(System.out::println);  // 4
}
```

4) 스트림 정렬

- sorted() 메소드는 해당 스트림을 주어진 비교자(comparator)를 이용하여 정렬한다.
- 이때 비교자를 전달하지 않으면 기본적으로 사전 편찬 순(natural order)으로 정렬한다.

```
public void solution4() {
        Stream<String> stream = Stream.of("송대관", "송미정", "송골매");
        stream.sorted().forEach(System.out::println); // 송골매 송대관 송미정

        Stream<Integer> stream2 = Stream.of(3,2,1);
        stream2.sorted().forEach(System.out::println);  // 1 2 3
}
```

6. 스트림의 최종 연산(terminal operation)

- 중개 연산을 통해 변환된 스트림은 마지막으로 최종 연산을 통해 각 요소를 소모하여 결과를 표시힌다.
- 즉, 지연(lazy)되었던 모든 중개 연산들이 최종 연산 시에 모두 수행되는 것 이다.
- 해당 스트림은 더는 사용할 수 없게 된다.
- 대표적인 최종 연산과 그에 따른 메소드는 다음과 같다.

    1) 요소의 소모: reduce()

    2) 요소의 수집: collect()

    3) 요소의 출력: forEach()

    4) 요소의 검색: findFirst(), fintAny()

    5) 요소의 통계/연산: count(), min(), max(), sum(), average()

    6) 요소의 검사: anyMatch(), allMatch(), noneMatch()

1) 요소의 소모

- reduce() 메소드는 첫 번째와 두 번째 요소를 가지고 연산을 수행한 뒤, 그 결과와 세 번째 요소를 가지고 또 다시 연산을 수행한다.

- 이런 식으로 해당 스트림의 모든 요소를 소모하여 연산을 수행하고, 그 결과를 반환하게 된다.

- 또한, 인수로 초기값을 전달하면 초기값과 해당 스트림의 첫 번째 요소와 연산을 시작하며, 그 결과와 두 번째 요소를 가지고 계속해서 연산을 수행하게 된다.

```
public void solution() {
        Stream<String> stream = Stream.of("수정", "현웅", "미정");
        Optional<String> result = stream.reduce((s1,s2) -> s1+"♥"+s2);
        result.ifPresent(System.out::println);
       // 수정♥현웅♥미정

        Stream<String> stream2 = Stream.of("수정", "현웅", "미정");
        String result2 = stream2.reduce("아잉", (s1,s2) -> s1+"♥"+s2);
        System.out.println(result2);
       // 아잉♥수정♥현웅♥미정
    }
```

- Optinoal 갑툭튀 뭔가요?

  - Optional<T> 클래스는 Integer나 Double 클래스처럼 'T'타입의 객체를 포장해 주는 래퍼 클래스(Wrapper class) 이다.

  - 따라서 Optional 인스턴스는 모든 타입의 참조 변수를 저장할 수 있다.

  - 이러한 Optional 객체를 사용하면 예상치 못한 NullPointerException 예외를 제공되는 메소드로 간단히 회피할 수 있다.

  - 즉, 복잡한 조건문 없이도 널(null) 값으로 인해 발생하는 예외를 처리할 수 있다.

2) 요소의 수집

- collect() 메소드는 인수로 전달되는 Collectors 객체에 구현된 방법대로 스트림의 요소를 수집한다.

- 또한, Collectors 클래스에는 미리 정의된 다양한 방법이 클래스 메소드로 정의되어 있다.

- 이 외에도 사용자가 직접 Collector 인터페이스를 구현하여 자신만의 수집 방법을 정의할 수도 있다.

- 스트림 요소의 수집 용도별 사용할 수 있는 Collectors 메소드는 다음과 같다.

1) 스트림을 배열이나 컬렉션으로 변환 : toArray(), toCollection(), toList(), toSet(), toMap()

2) 요소의 통계와 연산 메소드와 같은 동작을 수행 : counting(), maxBy(), minBy(), summingInt(), averagingInt() 등

3) 요소의 소모와 같은 동작을 수행 : reducing(), joining()

4) 요소의 그룹화와 분할 : groupingBy(), partitioningBy()

```
// Stream -> List
    public void solution2() {
        Stream<String> stream = Stream.of("일본 ", "미쳤어? ", "응");

        List<String> list = stream.collect(Collectors.toList());

        Iterator<String> iterator = list.iterator();
        while (iterator.hasNext()) {
            System.out.print(iterator.next());
        }
        // 일본 미쳤어?응
}
```

```
// Stream -> 글자수별로 나누어 저장
public void solution3() {
    Stream<String> stream = Stream.of("일", "이이", "삼삼삼", "사사사사");

    Map<Boolean, List<String>> partition =
       stream.collect(Collectors.partitioningBy(
          s -> s.length()%2!=0
       ));

    List<String> oddLength = partition.get(true);
    for(String s : oddLength) {
        System.out.println(s);  // 일 삼삼삼
    }
    List<String> evenLength = partition.get(false);
    for(String s : evenLength) {
        System.out.println(s);  // 이이 사사사사
    }
}
```