아이템 47 반환 타입으로는 스트림보다 컬렉션이 낫다

🙂 배경

- 자바 7까지는 일련의 원소를 반환하는 메서드의 반환 타입으로 **Collection, Set, List 같은 컬렉션 인터페이스, Iterable, 배열**을 사용했다.

  ▼ 선택 우선순위

  1. 기본은 컬렉션 인터페이스
  2. foreach문에서만 쓰이거나 반환된 원소시퀀스가 일부 Collection 메서드를 구현할 수 없을 때 Iterable 인터페이스
  3. 반환 원소들이 기본 타입이거나 성능에 민감하면 배열

- 자바 8 스트림 개념이 등장하면서 원소시퀀스의 반환타입을 정하기 복잡해졌다.

🤔 Stream 은 반복(iteration) 을 지원하지 않는다.

⇒ **스트림과 반복을 알맞게 조합**해야 좋은 코드가 나온다.

- Stream 인터페이스는 Iterable 인터페이스가 정의한 추상 메서드를 전부 포함하고 있고, Iterable 인터페이스가 정의한 방식대로 동작한다.

```
public interface BaseStream<T, S extends BaseStream<T, S>>
        extends AutoCloseable {
    /**
     * Returns an iterator for the elements of this stream.
     *
     * <p>This is a <a href="package-summary.html#StreamOps">terminal
     * operation</a>.
     *
     * @return the element iterator for this stream
     */
    @NotNull Iterator<T> iterator();
```

  - 하지만 Stream이 Iterable을 확장(extend)하지 않기 때문에 for-each로 반복할 수 없다.

```
// ex1
for(ProcessHandle ph : ProcessHandle.allProcesses()::iterator){
}
// ex2
Stream<String> games = Stream.of("오버워치", "배틀그라운드", "롤");
for (String game: games::iterator) { }
// 자바 타입 추론의 한계로 컴파일 되지 않는다.
```

  - 위 코드는 컴파일 오류를 내는데, 메서드 참조를 매개변수화된 Iterable로 적절히 형변환 해줘야 한다.

```
// ex 1
for(ProcessHandle ph : (Iterable<ProcessHandle>)ProcessHandle.allProcesses()::iterator){}
// ex 2
for(String game : (Iterable<String>) games::iterator){System.out.println(game);}
```

  - 작동은 하지만 실전에서 쓰기에 너무 난잡하고 직관성이 떨어진다.

🔎 어댑터 를 사용해서 해결하자

▼ adapter pattern

- 한 클래스의 인터페이스를 클라이언트에서 사용하고자 하는 다른 인터페이스로 변환한다.
- 어댑터를 이용하면 인터페이스 호환성 문제 때문에 같이 쓸 수 없는 클래스들을 연결해서 쓸 수 있다.

- 자바의 타입 추론이 문맥을 잘 파악해 어댑터 메서드 안에서 따로 형변환 하지 않아도 된다.

```
public static <E> Iterable<E> iterableOf(Stream<E> stream){
  return stream::iterator;
}
```

- 이 어댑터를 사용하면 어떤 스트림도 foreach문으로 반복할 수 있다.

```
// ex 1
for(ProcessHandle p : iterableOf(ProcessHandle.allProcesses())){}
// ex 2
for(String game : Adapters.iterableOf(games)){System.out.println(game);}
```

- Stream을 Iterable로 바꾼것과 마찬가지로 그 반대도 구현해야한다.

```
public static <E> Steam<E> streamOf(Iterable<E> iterable){
  return StreamSupport.stream(iterable.spliterator(), false);
}
```

- 스프림 파이프 라인에서만 쓰인다면 → Stream 반환
- 반복문에서만 쓰인다면 → Iterable 반환
- 여러 상황을 고려해 두가지 방법 모두 구현해야 한다.
- **Collection** 인터페이스는 Iterable의 하위 타입이고 stream 메서드도 제공하므로 **반복과 스트림**을 동시에 지원한다.
  - 원소 시퀀스를 반환하는 공개 API 반환 타입에는 Collection이나 그 하위 타입을 쓰는게 일반적으로 최선이다.
    - Arrays → Arrays.asList (Iterable), Stream.of (Stream)

🔎 시퀀스의 크기가 크다면 `전용 컬렉션`을 구현하자

- 단지 컬렉션을 반환한다는 이유로 덩치 큰 시퀀스를 메모리에 올려서는 안된다.
  - 표현을 간결하게 할 수 있다면 **전용 컬렉션을 구현**하는 방안을 검토하자
  - 예를 들어 주어진 집합의 멱집합(한집합의 모든 부분집합을 원소로 하는 집합)을 반환하는 상황에서
    - 원소의 개수가 n 개면 멱집합의 원소 개수는 2^n -1 개가 된다.

```
public class PowerSet {
// 입력 집합의 멱집합을 전용 컬렉션에 담아 반환한다.
    public static final <E> Collection<Set<E>> of(Set<E> s) {
        List<E> src = new ArrayList<>(s);
        // Collection.size()는 int(int의 최대 값은 2^31-1) 를 반환하기 때문에
        if (src.size() > 30)
            throw new IllegalArgumentException("Set too big " + s);
        return new AbstractList<Set<E>>() {
            @Override public int size() {
                return 1 << src.size(); // 멱집합의 크기는 2를 원래 집합의 원소의 수만큼 거듭제곱한 것과 같다.
            }

            @Override public boolean contains(Object o) {
                return o instanceof Set && src.containsAll((Set)o);
            }

            @Override public Set<E> get(int index) {
                Set<E> result = new HashSet<>();
                for (int i = 0; index != 0; i++, index >>= 1)
                    if ((index & 1) == 1)
                        result.add(src.get(i));
                return result;
            }
        };
    }

    public static void main(String[] args) {
        Set s = new HashSet(Arrays.asList(args));
        System.out.println(PowerSet.of(s));
    }
}
```

- 집합 원소의 수가 30을 넘으면 예외를 던지는데, 이는 Stream 이나 Iterable처럼 size 고려가 필요없는 것이 아니기 때문에 Collection을 쓸 때의 단점이다.

🤔 Stream이 나을 때도 있다

- AbstractCollection을 활용해 Collection 구현체를 작성할 때는 contains 와 size 만 더 구현하면 된다.
    - contains와 size를 구현하는게 불가능할 때는 Collection 보다 Stream이나 Iterable을 반환하는 편이 낫다.

🤔 for 반복문 vs stream

▼ IntStream, rangeClosed, range, flatMap, mapToObj

- InStream : int 를 스트림으로 다룰 수 있도록 한다.
- rangeClosed(1,5); 1,2,3,4,5 에 대한 int 스트림 생성
- range(1,5) 1,2,3,4에 대한 int 스트림 생성
- flatMap : 스트림의 형태가 배열과 같을 때, 모든 원소를 단일 원소 스트림으로 반환할 수 있다.
- mapToObj : 객체 스트림으로 변환
- subList : List<E> subList(int fromIndex, int toIndex);

```
public class SubLists {
    private static <E> Stream<List<E>> prefixes(List<E> list) {
        return IntStream.rangeClosed(1, list.size())
                .mapToObj(end -> list.subList(0, end));
    }
// (a,b,c)의 prefixes :  (a), (a,b),(a,b,c)

    private static <E> Stream<List<E>> suffixes(List<E> list) {
        return IntStream.range(0, list.size())
                .mapToObj(start -> list.subList(start, list.size()));
    }
// (a) 의 suffixes : (a)
// (a,b)의 suffixes : (a,b), (a)
// (a,b,c)의 suffixes :(a,b,c), (b,c), (c)

// 입력 리스트의 모든 부분 리스트를 스트림으로 반환한다.
// -> 단 3줄이면 충분하지만 입력 리스트 크기의 거듭제곱만큼 메모리를 차지 한다.  -> 좋은 방식은 아니다.
    public static <E> Stream<List<E>> of(List<E> list) {
        return Stream.concat(Stream.of(Collections.emptyList()),
                prefixes(list).flatMap(SubLists::suffixes));
    }
// stream.concat메서드는 반환되는 스트림에 빈 리스트를 추가하며,
// flatMap메서드(아이템 45)는 모든 프리픽스의 모든 서픽스로 구성된 하나의 스트림을 만든다.

    public static void main(String[] args) {
        List<String> list = Arrays.asList(args);
        SubLists.of(list).forEach(System.out::println);
    }
}
```

```
for (int start = 0; start < src.size(); start++) {
    for (int end = start + 1; end <= src.size(); end++) {
        System.out.println(src.subList(start, end));
    }
}
```

```
public static <E> Stream<List<E>> of(List<E> list) {
    return IntStream.range(0, list.size())
            .mapToObj(start ->
                    IntStream.rangeClosed(start + 1, list.size())
                            .mapToObj(end -> list.subList(start, end)))
            .flatMap(x -> x);
}
```

- 빈 리스트는 concat을 사용하더가 rangeClosed 호출을 1→Math.signum(start)로 고치면 된다
- for 반복문 보다 간결하지만, 읽기에는 더 안좋다.

정리

- Stream이나 Iterable을 리턴하는 API에서는 stream→Iterable, Iterable → Stream 으로 변환해주는 어댑터를 사용해야 한다.
    - 하지만 어댑터는 클라이언트 코드를 어수선하게 만들고 (약 2.3배) 더 느리다.

- 컬렉션을 반환할 수 있다면 컬렉션으로 반환한다.
- 원소 시퀀스를 반환하는 메서드를 작성할 때는 Stream, Iterator를 모두 지원할 수 있게 작성해야한다.
- 원소의 갯수가 많으면, 전용 컬렉션을 리턴하는 방법을 고려하자.
- 나중에 Stream 인터페이스가 Iterable을 지원하면 그땐 안심하고 Stream을 반환하면 된다.