Streams

Using Intermediate Operations

Intermediate Operations

- produces a stream as a result
- can deal with infinite streams

  (by returning another infinite stream)
- can be omitted in a pipeline

  (unlike source and terminal operations)

```java
// filtering

Stream<String> names = Stream.of("John", "George", "Ben");

names.filter(s -> s.startsWith("G")).forEach(System.out::println);
```

source *intermediate operation* *terminal operation*

```
George
```

argument of `filter()` is `Predicate`

```
// removing duplicates

Stream<String> names = Stream.of("John", "John", "John", "Ben");

names.distinct().forEach(System.out::println);
```

intermediate operation

```
John
Ben
```

```
// restricting by position
```

infinite stream of numbers starting from 1

```
Stream<Integer> numbers = Stream.iterate(1, n -> n + 1);

names.skip(3)
     .limit(4)
     .forEach(System.out::print);
```

create a stream by skipping first 3 elements from the source stream

create a stream using first 4 elements from the previous stream

terminate a stream

```
4567
```

```java
// mapping using map()

Stream<String> names = Stream.of("John", "George", "Ben");

names.map(s -> s.length()).forEach(System.out::print);
```

argument of `map()` is `Function` (equivalent: `String::length`)

creates one-to-one mapping from elements in source stream to a new stream

```
463
```

```java
// mapping using flatMap()

List<String> zero = List.of();

List<String> one = List.of("John");

List<String> two = List.of("George", "Ben");

Stream<List<String>> names = Stream.of(zero, one, two);

names.flatMap(m -> m.stream()).forEach(System.out::println);
```

argument of `flatMap()` is `Function`

removes the empty list, and changes all elements to be at the top level of the stream

```
John
George
Ben
```

```
// sorting

Stream<String> names = Stream.of("John", "George", "Benedict");

names.sorted().forEach(System.out::print);

  => BenedictGeorgeJohn


// we can provide Comparator as an argument, e.g.

Stream<String> myNames = Stream.of("John", "George", "Benedict");

myNames.sorted(Comparator.comparingInt(String::length))

       .forEach(System.out::print);

  => JohnGeorgeBenedict
```

```
// peek()

Stream<String> names = Stream.of("John", "George", "Ben");

long count = names.filter(s -> s.startsWith("G"))

                  .count();

System.out.println(count);

  => 1


// if we want to see what's going on in the pipeline:

long count = names.filter(s -> s.startsWith("G"))

                  .peek(System.out::println)

                  .count();

System.out.println(count);
```

argument of peek() is Consumer

```
George
1
```