JAVA 8

Optional and Streams

OPTIONAL

An Optional is a pipeline of operations to apply to a unique value if it exists.

Instantiation

```
// From no value, empty optional
Optional.empty()
// From non null reference
Optional.of(T ref); // ref cannot be null
// From possibly null reference
Optional.ofNullable(T ref); // ref can be null
```

Intermediate operations

```
// Filtering Optional<T>
Optional<T>
    Optional.filter(Predicate<T> p)

// Transforming Optional<T> to Optional<U>
Optional<U>
    Optional.map(Function<T, U> m)

// Transforming Optional<T> to Optional<U> when
// mapper function returns an Optional<U>
Optional<U>
    Optional.flatMap(Function<T, Optional<U>> m)
```

Terminal operations

To retrieve the value in Optional:

```
// throw NoSuchElementException if empty
T Optional.get()
// return valueIfEmpty if empty
T Optional.orElse(T valueIfEmpty)
// get supplier result if empty
T Optional.orElseGet(Supplier<T> s)
// throw supplier result if empty
T Optional.orElseThrow(Supplier<Throwable> s)

// Do something if Optional<T> is not empty
void Optional.ifPresent(Consumer<T> c)
```

Utilities

```
// Get if optional is empty or not
boolean Optional.isEmpty()
```

STREAMS

A stream is a pipeline of functions to apply to a possibly infinite amount of values.

Instantiation

```
// From no value, empty stream
Stream.empty()
// From one value
Stream.of(T value);
// From multiple values
Stream.of(T... values);
// Infinite unordered stream
Stream.generate(Supplier<T> s)
// Infinite ordered stream
Stream.iterate(T seed, UnaryOperator<T> op)
```

Intermediate operations

```
// Keep only distinct elements in Stream<T>
Stream<T> Stream.distinct()

// Counting elements in Stream<T>
long Stream.count()

// Taking only first n elements in Stream<T>
Stream<T> Stream.limit(int n)

// Skiping first n elements of Stream<T>
Stream<T> Stream.skip(int n)

// Filtering Stream<T>
Stream<T>
    Stream.filter(Predicate<T> p)

// Transforming Stream<T> to Stream<U>
Stream<U>
    Stream.map(Function<T, U> m)

// Transforming Stream<T> to Stream<U> when
// mapper function returns a Stream<U>
Stream<U>
    Stream.flatMap(Function<T, Stream<U>> m)

// Sort elements in Stream<T>
// Using elements natural order
Stream<T>
    Stream.sorted()
// Using different sort order
Stream<T>
    Stream.sorted(Comparator<T> c)
```

OPTIONAL AND STREAMS

Terminal operations

To test if:

```
// All elements of Stream<T> match a predicate:
boolean Stream.allMatch(Predicate<T> p)

// At least one element of Stream<T> match
// a predicate:
boolean Stream.anyMatch(Predicate<T> p)

// No element of Stream<T> match a predicate:
boolean Stream.noneMatch(Predicate<T> p)
```

To retrieve a value:

```
// Any element of Stream<T>
Optional<T>
    Stream.findAny()

// First element of Stream<T>
Optional<T>
    Stream.findFirst()

// Minimum of all elements of Stream<T>
Optional<T>
    Stream.min(Comparator<T> c)

// Maximum of all elements of Stream<T>
Optional<T>
    Stream.max(Comparator<T> c)

// Combine together elements of Stream<T>
// to get a unique value
// With an accumulator
Optional<T>
    Stream.reduce(BinaryOperator<T> o)
// With a seed and an accumulator
T
    Stream.reduce(T ident, BinaryOperator<T> o)
```

To collect elements:

```
// Into a List<T>
List<T>
    Stream.collect(Collectors.toList())

// Into a Set<T>
Set<T>
    Stream.collect(Collectors.toSet())

// Into another structure R with A the
// intermediate collector type
R
    Stream.collect(Collector<T,A,R> c)
```

To apply an operation on each element:

```
void Stream.forEach(Consumer<T> c)
```

Utilities

Transform into primitive streams:

```
// Using map
IntStream map(ToIntFunction<T> f)
LongStream map(ToLongFunction<T> f)
DoubleStream map(ToDoubleFunction<T> f)

// Using flatMap
IntStream flatMap(ToIntFunction<T> f)
LongStream flatMap(ToLongFunction<T> f)
DoubleStream flatMap(ToDoubleFunction<T> f)
```

Cause side effets for each element:

```
Stream<T> Stream.peek(Consumer<T> c)
```

Retrieve elements into an Array:

```
Object[] Stream.toArray()
A[] Stream.toArray(IntFunction<A[]> gen)
```

Be careful

Most of terminal operations can only be applied to finite streams. Any operation that needs to have all elements to be performed (counting, reducing, collecting, ...), when called on a infinite stream, will lead to an infinite loop.

EXAMPLES

```
Optional.of(user)
    .filter(User::hasEmail)
    .map(User::getEmail)
    .ifPresent(this::sendWelcomeEmail);
```

```
Stream.of(users)
    .filter(user -> !user.hasGuiColorPref())
    .peek(user -> user.setGuiColorPref(DEFAULT))
    .flatMap(user -> Stream.of(user)
        .filter(User::hasEmail)
        .map(User::getEmail))
    .forEach(this::sendGuiColorPrefWarnEmail);
```