Streams

Streams and Pipelines

A stream is a sequence of elements.

- Unlike a collection, it is not a data structure that stores elements.
- Unlike an iterator, streams do not allow modification of the underlying source

A stream carries values from a source through a pipeline.
A pipeline contains the following components:

- A source: This could be a collection, an array, a generator function, or an I/O channel.
- Zero or more intermediate operations. An intermediate operation, such as filter, produces a new stream
- A terminal operation. A terminal operation, such as forEach, produces a non-stream result, such as a primitive value (like a double value), a collection, or in the case of forEach, no value at all.

Stream Example: How Many Mustaches?

Consider this simple example from SuperTroopers.java:

```
long mustaches = troopers.stream().filter(Trooper::hasMustache).count();
System.out.println("Mustaches: " + mustaches);
```

- `troopers.stream()` is the *source*
- `.filter(Trooper::hasMustache)` is an *intermediate operation*
- `.count()` is the *terminal operation*

The terminal operation yields a new value which results from applying all the intermediate operations and finally the terminal operation to the source.
Let's look at this simple stream in detail.

Stream.filter

Here's the signature of the filter method in Stream<T>:

```
Stream<T> filter(Predicate<? super T> predicate)
```

- filter takes a Predicate<? super T> predicate)

Here's Predicate<T>

```
public interface Predicate<T> {
    default Predicate<T> and(Predicate<? super T> other) { ... }

    static <T> Predicate<T> isEqual(Object targetRef) { ... }

    default Predicate<T> negate() { ... }

    default Predicate<T> or(Predicate<? super T> other) { ... }

    boolean test(T t);
}
```

What is the single abstract method (SAM) in Predicate<T>?

A Predicate<T> Class

A Predicate<T> is an object that has a boolean test(T t) method.
Here's a Predicate<T> class:

```
public class HasMustache implements Predicate<Trooper> {

    public boolean test(Trooper t) {
        return t.hasMustache();
    }
}
```

An object of the HasMustache class is a Predicate<T>.
Here's the same Predicate<T> class as an anonymous inner class:

```
new Predicate<Trooper>() {
    public boolean test(Trooper t) {
        return t.hasMustache();
    }
}
```

A `Predicate<T>` Lambda Expression

Here's the same `Predicate<T>` class as a lambda expression (in the context of the filter method – why is that important?):

```
troopers.stream().filter(t -> t.hasMustache()).count();
```

Here's the lambda expression above as a method reference:

```
troopers.stream().filter(Trooper::hasMustache).count();
```

`Trooper::hasMustache` supplies the body of the `Predicate<T>` `test(T t)` method by calling `t.hasMustache()`.

Play with StreamTroopers.java to see these `Predicate<T>` implementations in action.

A Bigger Stream Example: WordCount Pipeline

Consider this example from `WordCount`:

```
Set<String> stopWords = new HashSet<>(Arrays.asList(
    "a", "an", "and", "are", "as", "be", "by", "is", "in", "of",
    "for", "from", "not", "to", "the", "that", "this", "with", "which"
));
wc.wordCounts.entrySet().stream()
    .filter(entry -> !stopWords.contains(entry.getKey().toLowerCase()))
    .sorted((e1, e2) -> e1.getValue() - e2.getValue())
    .forEach(entry ->
        System.out.printf("%s occurs %d times%n", entry.getKey(),
                                                  entry.getValue()));
```

This code does the same tasks we did before with classes and for loops.

WordCount Pipeline - Stop Words

```
Set<String> stopWords = new HashSet<>(Arrays.asList(
    "a", "an", "and", "are", "as", "be", "by", "is", "in", "of",
    "for", "from", "not", "to", "the", "that", "this", "with", "which"
));
```

- Every document has information-carrying words and grammatical words that carry no information, like prepositions, verbs like to be or have, pronouns
- In document processing we call these non-information-carrying words *stop words*

Here we've implemented a naiive and terribly incomplete stop words list. BTW, why a `HashSet`?

WordCount Pipeline - filter

Consider this example from ~WordCount~s:

```
Set<String> stopWords = new HashSet<>(Arrays.asList(
    "a", "an", "and", "are", "as", "be", "by", "is", "in", "of",
    "for", "from", "not", "to", "the", "that", "this", "with", "which"
));
wc.wordCounts.entrySet().stream()
    .filter(entry -> !stopWords.contains(entry.getKey().toLowerCase()))
    .sorted((e1, e2) -> e1.getValue() - e2.getValue())
    .forEach(entry ->
        System.out.printf("%s occurs %d times%n", entry.getKey(),
                                                  entry.getValue()));
```

The filter operation takes a predicate function.

- A predicate function returns a `boolean`
- If predicate function returns `true`, element is retained in the stream

Notice that we're also normalizing words to lower case.

WordCount Pipeline - sorted

Consider this example from ~WordCount~s:

```
Set<String> stopWords = new HashSet<>(Arrays.asList(
    "a", "an", "and", "are", "as", "be", "by", "is", "in", "of",
    "for", "from", "not", "to", "the", "that", "this", "with", "which"
));
wc.wordCounts.entrySet().stream()
    .filter(entry -> !stopWords.contains(entry.getKey().toLowerCase()))
    .sorted((e1, e2) -> e1.getValue() - e2.getValue())
    .forEach(entry ->
        System.out.printf("%s occurs %d times%n", entry.getKey(),
                                                  entry.getValue()));
```

The sortd operation takes a `Comparator` that defines the ordering over the stream's elements.

WordCount Pipeline - forEach

Consider this example from ~WordCount~s:

```
Set<String> stopWords = new HashSet<>(Arrays.asList(
    "a", "an", "and", "are", "as", "be", "by", "is", "in", "of",
    "for", "from", "not", "to", "the", "that", "this", "with", "which"
));
wc.wordCounts.entrySet().stream()
    .filter(entry -> !stopWords.contains(entry.getKey().toLowerCase()))
    .sorted((e1, e2) -> e1.getValue() - e2.getValue())
    .forEach(entry ->
        System.out.printf("%s occurs %d times%n", entry.getKey(),
                                          entry.getValue()));
```

`forach` is the terminal operation.

- Called for its effect - no return value

The underlying source is not modified.

Homework: Make the WordCount Pipeline Clearer

Notice that we use anonymous lambda expressions in our WordCOunt pipeline:

```
wc.wordCounts.entrySet().stream()
    .filter(entry -> !stopWords.contains(entry.getKey().toLowerCase()))
    .sorted((e1, e2) -> e1.getValue() - e2.getValue())
    .forEach(entry ->
        System.out.printf("%s occurs %d times%n", entry.getKey(),
                                                  entry.getValue()));
```

- Functional-style code can easily become hard to read.
- You can improve readability by introducing intermediate helper variables with informative names.

Rewrite the WordCount pipeline with intermediate helper variables so that the pipeline is easy to understand. You'll need to look up these aggregate operations in the Java API to get the types for these variables.