Streams

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 8

Operating on collections

- We usually use an iterator to process a collection
    - Suppose we have split a text file as a list of words
    - We want to count the number of long words in the list

```
List<String> words = ....;
long count = 0;
for (String w : words) {
  if (w.length() > 10) {
    count++;
  }
}
```

Operating on collections

- We usually use an iterator to process a collection
    - Suppose we have split a text file as a list of words
    - We want to count the number of long words in the list
- An iterator generates all elements from a collection as a sequence

```
List<String> words = ....;
long count = 0;
for (String w : words) {
  if (w.length() > 10) {
    count++;
  }
}
```

Operating on collections

- We usually use an iterator to process a collection
    - Suppose we have split a text file as a list of words
    - We want to count the number of long words in the list

```
List<String> words = ....;
long count = 0;
for (String w : words) {
  if (w.length() > 10) {
    count++;
  }
}
```

- An iterator generates all elements from a collection as a sequence
- Alternative approach
    - Generate a stream of values from a collection
    - Operations transform input streams to output streams
    - Terminate with a result

```
long count = words.stream()
            .filter(w -> w.length() > 10)
            .count();
}
```

Why streams?

- Stream processing is declarative
  - Recall, declarative vs imperative
  - Focus on what to compute, rather than how

```
long count = words.stream()
            .filter(w -> w.length() > 10)
            .count();
}
```

Why streams?

- Stream processing is declarative
  - Recall, declarative vs imperative
  - Focus on what to compute, rather than how
- Processing can be parallelized
  - `filter()` and `count()` in parallel

```
long count = words.stream()
                  .filter(w -> w.length() > 10)
                  .count();
}
```

```
long count = words.parallelStream()
                  .filter(w -> w.length() > 10)
                  .count();
}
```

Why streams?

- Stream processing is declarative
  - Recall, declarative vs imperative
  - Focus on what to compute, rather than how
- Processing can be parallelized
  - `filter()` and `count()` in parallel
- Lazy evaluation is possible
  - Suppose we want first 10 long words
  - Stop generating the stream once we find 10 such words
  - Need not generate the entire stream in advance
  - Can even work, in principle, with infinite streams!

```
long count = words.stream()
            .filter(w -> w.length() > 10)
            .count();
}
```

```
long count = words.parallelStream()
            .filter(w -> w.length() > 10)
            .count();
}
```

Working with streams

- Create a stream

```
long count = words.stream()
            .filter(w -> w.length() > 10)
            .count();
}
```

```
long count = words.parallelStream()
            .filter(w -> w.length() > 10)
            .count();
}
```

Working with streams

- Create a stream
- Pass through intermediate operations that transform streams

```
long count = words.stream()
            .filter(w -> w.length() > 10)
            .count();
}
```

```
long count = words.parallelStream()
            .filter(w -> w.length() > 10)
            .count();
}
```

Working with streams

- Create a stream
- Pass through intermediate operations that transform streams
- Apply a terminal operation to get a result

```
long count = words.stream()
            .filter(w -> w.length() > 10)
            .count();
}
```

```
long count = words.parallelStream()
            .filter(w -> w.length() > 10)
            .count();
}
```

Working with streams

- Create a stream
- Pass through intermediate operations that transform streams
- Apply a terminal operation to get a result
- A stream does not store its elements
  - Elements stored in an underlying collection
  - Or generated by a function, on demand

```
long count = words.stream()
            .filter(w -> w.length() > 10)
            .count();
}
```

```
long count = words.parallelStream()
            .filter(w -> w.length() > 10)
            .count();
}
```

Working with streams

- Create a stream
- Pass through intermediate operations that transform streams
- Apply a terminal operation to get a result
- A stream does not store its elements
  - Elements stored in an underlying collection
  - Or generated by a function, on demand
- Stream operations are non-destructive
  - Input stream is untouched

```
long count = words.stream()
            .filter(w -> w.length() > 10)
            .count();
}
```

```
long count = words.parallelStream()
            .filter(w -> w.length() > 10)
            .count();
}
```

Creating streams

- Apply `stream()` to a collection
    - Part of `Collections` interface

```
List<String> wordlist = ...;
Stream<String> wordstream = wordlist.stream();
```

Creating streams

- Apply `stream()` to a collection
    - Part of `Collections` interface

```
List<String> wordlist = ...;
Stream<String> wordstream = wordlist.stream();
```

- Use static method `Stream.of()` for arrays

```
String[] wordarr = ...;
Stream<String> wordstream = Stream.of(wordarr);
```

Creating streams

- Apply `stream()` to a collection
  - Part of `Collections` interface
- Use static method `Stream.of()` for arrays
- Static method `Stream.generate()` generates a stream from a function
  - Provide a function that produces values on demand, with no argument

```
List<String> wordlist = ...;
Stream<String> wordstream = wordlist.stream();

String[] wordarr = ...;
Stream<String> wordstream = Stream.of(wordarr);

Stream<String> echos =
  Stream.generate(() -> "Echo");

Stream<Double> randomds =
  Stream.generate(Math::random);
```

Creating streams

- Apply `stream()` to a collection
    - Part of `Collections` interface
- Use static method `Stream.of()` for arrays
- Static method `Stream.generate()` generates a stream from a function
    - Provide a function that produces values on demand, with no argument
- `Stream.iterate()` — a stream of dependent values
    - Initial value, function to generate the next value from the previous one

```
List<String> wordlist = ...;
Stream<String> wordstream = wordlist.stream();

String[] wordarr = ...;
Stream<String> wordstream = Stream.of(wordarr);

Stream<String> echos =
  Stream.generate(() -> "Echo");

Stream<Double> randomds =
  Stream.generate(Math::random);

Stream<Integer> integers =
  Stream.iterate(0, n -> n+1)
```

Creating streams

- Apply `stream()` to a collection
    - Part of `Collections` interface
- Use static method `Stream.of()` for arrays
- Static method `Stream.generate()` generates a stream from a function
    - Provide a function that produces values on demand, with no argument
- `Stream.iterate()` — a stream of dependent values
    - Initial value, function to generate the next value from the previous one
    - Terminate using a predicate

```
List<String> wordlist = ...;
Stream<String> wordstream = wordlist.stream();

String[] wordarr = ...;
Stream<String> wordstream = Stream.of(wordarr);

Stream<String> echos =
  Stream.generate(() -> "Echo");

Stream<Double> randomds =
  Stream.generate(Math::random);

Stream<Integer> integers =
  Stream.iterate(0, n -> n+1)

Stream<Integer> integers =
  Stream.iterate(0, n -> n < 100, n -> n+1)
```

Processing streams

- `filter()` to select elements
  - Takes a predicate as argument
  - Filter out the long words

```
List<String> wordlist = ...;
Stream<String> longwords =
   wordlist.stream()
   .filter(w -> w.length() > 10);
```

Processing streams

- `filter()` to select elements
  - Takes a predicate as argument
  - Filter out the long words
- `map()` applies a function to each element in the stream.
  - Extract the first letter of each long word

```
List<String> wordlist = ...;
Stream<String> longwords =
   wordlist.stream()
   .filter(w -> w.length() > 10);
```

```
List<String> wordlist = ...;
Stream<String> startlongwords =
   wordlist.stream()
   .filter(w -> w.length() > 10)
   .map(s -> s.substring(0,1));
```

Processing streams

- filter() to select elements
  - Takes a predicate as argument
  - Filter out the long words
- map() applies a function to each element in the stream.
  - Extract the first letter of each long word
- What if map() function generates a list?
  - Suppose we have explode(s) that returns the list of letters in s
  - map() produces stream with nested lists

```
List<String> wordlist = ...;
Stream<String> longwords =
   wordlist.stream()
   .filter(w -> w.length() > 10);
```

```
List<String> wordlist = ...;
Stream<String> startlongwords =
   wordlist.stream()
   .filter(w -> w.length() > 10)
   .map(s -> s.substring(0,1));
```

```
List<String> wordlist = ...;
Stream<String> startlongwords =
   wordlist.stream()
   .filter(w -> w.length() > 10)
   .map(s -> explode(s));
```

Processing streams

- filter() to select elements
  - Takes a predicate as argument
  - Filter out the long words
- map() applies a function to each element in the stream.
  - Extract the first letter of each long word
- What if map() function generates a list?
  - Suppose we have explode(s) that returns the list of letters in s
  - map() produces stream with nested lists
- flatMap() flattens (collapses) nested list into a single stream

```
List<String> wordlist = ...;
Stream<String> longwords =
   wordlist.stream()
   .filter(w -> w.length() > 10);
```

```
List<String> wordlist = ...;
Stream<String> startlongwords =
   wordlist.stream()
   .filter(w -> w.length() > 10)
   .map(s -> s.substring(0,1));
```

```
List<String> wordlist = ...;
Stream<String> startlongwords =
   wordlist.stream()
   .filter(w -> w.length() > 10)
   .flatMap(s -> explode(s));
```

Stream transformations

- Make a stream finite — `limit(n)`
  - Generate 100 random numbers

```
Stream<Double> randomds =
  Stream.generate(Math::random).limit(100);
```

Stream transformations

- Make a stream finite — `limit(n)`
    - Generate 100 random numbers
- Skip `n` elements — `skip(n)`
    - Discard first 10 random numbers

```
Stream<Double> randomds =
  Stream.generate(Math::random).limit(100);

Stream<Double> randomds =
  Stream.generate(Math::random).skip(10);
```

Stream transformations

- Make a stream finite — `limit(n)`
  - Generate 100 random numbers
- Skip `n` elements — `skip(n)`
  - Discard first 10 random numbers
- Stop when element matches a criterion — `takeWhile()`
  - Stop with number smaller than 0.5

```
Stream<Double> randomds =
  Stream.generate(Math::random).limit(100);

Stream<Double> randomds =
  Stream.generate(Math::random).skip(10);

Stream<Double> randomds =
  Stream.generate(Math::random)
        .takeWhile(n -> n >= 0.5);
```

Stream transformations

- Make a stream finite — `limit(n)`
    - Generate 100 random numbers
- Skip `n` elements — `skip(n)`
    - Discard first 10 random numbers
- Stop when element matches a criterion — `takeWhile()`
    - Stop with number smaller than 0.5
- Start after element matches a criterion — `dropWhile()`
    - Start after number larger than 0.05

```
Stream<Double> randomds =
  Stream.generate(Math::random).limit(100);
```

```
Stream<Double> randomds =
  Stream.generate(Math::random).skip(10);
```

```
Stream<Double> randomds =
  Stream.generate(Math::random)
        .takeWhile(n -> n >= 0.5);
```

```
Stream<Double> randomds =
  Stream.generate(Math::random)
        .dropWhile(n -> n <= 0.05);
```

Stream transformations

- Make a stream finite — `limit(n)`
  - Generate 100 random numbers
- Skip `n` elements — `skip(n)`
  - Discard first 10 random numbers
- Stop when element matches a criterion — `takeWhile()`
  - Stop with number smaller than 0.5
- Start after element matches a criterion — `dropWhile()`
  - Start after number larger than 0.05
- Can also combine streams, extract distinct elements, sort, . . .

```
Stream<Double> randomds =
  Stream.generate(Math::random).limit(100);
```

```
Stream<Double> randomds =
  Stream.generate(Math::random).skip(10);
```

```
Stream<Double> randomds =
  Stream.generate(Math::random)
        .takeWhile(n -> n >= 0.5);
```

```
Stream<Double> randomds =
  Stream.generate(Math::random)
        .dropWhile(n -> n <= 0.05);
```

Reducing a stream to a result

- Number of elements — `count()`
  - Count random numbers larger than 0.1

```
long countrand =
  Stream.generate(Math::random)
        .limit(100).
        .filter(n -> n > 0.1)
        .count();
```

Reducing a stream to a result

- Number of elements — `count()`
  - Count random numbers larger than 0.1
- Largest and smallest values seen
  - `max()` and `min()`
  - Requires a comparison function

```
long countrand =
  Stream.generate(Math::random)
        .limit(100).
        .filter(n -> n > 0.1)
        .count();

Optional<Double> maxrand =
  Stream.generate(Math::random)
        .limit(10)
        .max(Double::compareTo);
```

Reducing a stream to a result

- Number of elements — `count()`
  - Count random numbers larger than 0.1
- Largest and smallest values seen
  - `max()` and `min()`
  - Requires a comparison function
  - What happens if the stream is empty?
    Return value is optional type — later

```
long countrand =
  Stream.generate(Math::random)
        .limit(100).
        .filter(n -> n > 0.1)
        .count();

Optional<Double> maxrand =
  Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n < 0.001)
        .max(Double::compareTo);
```

Reducing a stream to a result

- Number of elements — `count()`
  - Count random numbers larger than 0.1
- Largest and smallest values seen
  - `max()` and `min()`
  - Requires a comparison function
  - What happens if the stream is empty?
    Return value is optional type — later
- First element — `findFirst()`
  - First random number above 0.999
  - Again, deal with empty stream
- And more . . .

```
long countrand =
  Stream.generate(Math::random)
        .limit(100).
        .filter(n -> n > 0.1)
        .count();

Optional<Double> maxrand =
  Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n < 0.001)
        .max(Double::compareTo);

Optional<Double> firstrand =
  Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n > 0.999)
        .findFirst();
```

Streams

- We can view a collection as a stream of elements
- Process the stream rather than use an iterator
- Declarative way of computing over collections — popular in functional programming
- Create a stream, transform it, reduce it to a result
- Can create a stream from any collection, or generate from a function
- Stream transformations are non-destructive: filter, map, limit to a finite number, skip elements, . . .
- Various functions to reduce to a result — deal with empty streams