Berner Fachhochschule - Technik und Informatik

Object-Oriented Programming 2

Topic 3: Streams

Prof. Rolf Haenni & Prof. Annett Laube

FS 2016

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Outline

Introduction

Stream Sources

Intermediate Operations

Terminal Operations

Further Stream Examples

Outline

Introduction

Stream Sources

Intermediate Operations

Terminal Operations

Further Stream Examples

Java 8 Streams

- The concept of a stream has been introduced in Java 8 (not to confuse with input/output streams)
- A stream represents a sequence of elements (similar to a list), but does not allow positional access
- Each stream is linked to a source, but otherwise does not store any elements
- The main purpose of a stream is to iterate through the elements (similar to the `Iterable` interface) and to perform various operations on them
- In this way, processing of data can be performed in a declarative way (similar to SQL statements)

Example of Using Streams I

```
public class StreamExample {

  public static void main(String[] args) {

    List<String> list = Arrays.asList("a1", "a2", "b1", "c2",
        "c1", "d1");

    // Standard solution
    List<String> filteredList = new ArrayList<>();
    for (String s: list) {
      if (s.startsWith("c")) {
        filteredList.add(s.toUpperCase());
      }
    }
    Collections.sort(filteredList);
    for (String s: filteredList) {
      System.out.println(s); // prints C1 C2
```

Example of Using Streams II

```
    }

    // Solution with streams
    Stream<String> stream = list.stream();
    stream
      .filter(s -> s.startsWith("c"))
      .map(String::toUpperCase)
      .sorted()
      .forEach(System.out::println); // prints C1 C2
  }

}
```

Stream Pipeline

- The previous example shows a stream pipeline, which consists of three components:
  - → The source
  - → One or multiple intermediate operations
  - → A single terminal operation
- The return value of an intermediate operation is a stream (of the same or of another type)
- The return value of a terminal operation is not a stream
- A stream without a terminal operation is purely declarative, it does not consume any computational resources
  - → `stream.filter(s -> s.startsWith("c"))` does nothing

Stream Interface

- The generic interface `Stream<T>` is the core of the Java stream framework
- It provides multiple ...
  - → static methods for generating stream sources
  - → intermediate operations
  - → terminal operations
- Besides, there are three specialized classes `IntStream`, `LongStream`, and `DoubleStream` with additional operations for numbers (sum, min, max, average, ...)
- Note that `IntStream` does not implement `Stream<Integer>` (same for `LongStream` and `DoubleStream`)

Helper Interfaces and Classes I

- `Function<T,R>`: A function that accepts one argument and produces a result
  - → `R apply(T t)`
- `UnaryOperator<T>`: An operation on a single operand that produces a result of the same type
  - → `T apply(T t)`
- `BinaryOperator<T>`: An operation on two operands of the same type that produces a result of the same type
  - → `T apply(T t, T t)`
- `Predicate<T>`: A boolean-valued function of one argument
  - → `boolean test(T t)`:

Helper Interfaces and Classes II

- `Consumer<T>`: An operation that accepts a single input and returns no result
  - → `void accept(T t))`
- `Supplier<T>`: A supplier function of no arguments
  - → `T get()`
- `Optional<T>`: A container which may or may not contain a non-null value
  - → `boolean isPresent()`
  - → `T get()`
  - → `T orElse(T other)`
  - → `void ifPresent(Consumer<? super T> consumer)`

Properties of Streams

- Constant Memory: a stream is not a data structure that requires memory space to stores elements
- Functional: Operations on a stream produce a result, but do not modify its source
- Lazyness: Many stream operations can be implemented lazily, which generates opportunities for optimization
- No Bounds: While collections have a finite size, streams can be unbounded
- Consumable: The elements of a stream are only accessed once during the life of a stream
- Parallelism: Data processing can be executed in parallel, which can improve the performance

Outline

Introduction

Stream Sources

Intermediate Operations

Terminal Operations

Further Stream Examples

Stream Sources I

- From arrays
  - → `Arrays.stream(T[] array)`
  - → `Stream.of(T... values)`
- From collections (lists, sets, ...)
  - → `collection.stream()`
  - → `collection.parallelStream()`
- From a seed and update function
  - → `Stream.iterate(T seed, UnaryOperator<T> update)`
- From a supplier
  - → `Stream.generate(Supplier<T> supplier)`
- From a stream builder
  - → `Stream.builder().add(v1).add(v2). ... .build()`

Stream Sources II

- From a random generator
  - → `random.ints(int low, int high)`
  - → `random.doubles(double low, double high)`
- From strings
  - → `string.chars()`
- From files (text, jar, zip)
  - → `Files.lines(Path path)`
  - → `new JarFile(File file).stream()`
  - → `new ZipFile(File file).stream()`
- From directories
  - → `Files.list(Path path)`
  - → `Files.walk(Path path)`

Example of Stream Sources I

```
public class StreamSources {

  public static void main(String[] args) {

    Stream<Integer> stream;

    // From an array
    Integer[] array = new Integer[] { 1, 2, 3, 4, 5 };
    stream = Arrays.stream(array);
    stream = Stream.of(array);

    // From a collection (list)
    List<Integer> list = Arrays.asList(array);
    stream = list.stream();
    stream = list.parallelStream();
```

Example of Stream Sources II

```
// From a collection (set)
Set<Integer> set = new HashSet<>(list);
stream = set.stream();
stream = set.parallelStream();

// From a seed and an update function
stream = Stream.iterate(0, x -> x + 1);

// From a supplier
Random random = new Random();
stream = Stream.generate(() -> random.nextInt());

// From a stream builder
Stream.Builder<Integer> builder = Stream.builder();
builder.add(1);
builder.add(2).add(3).add(4).add(5);
stream = builder.build();
```

Example of Stream Sources III

```
    // From a random generator
    IntStream ints = new Random().ints(1, 100);
    DoubleStream doubles = new Random().doubles(0.0, 1.0);

    // From a string
    IntStream chars = "12345".chars();

    // From a text file
    Path path = FileSystems.getDefault().getPath("src/topic03"
        , "numbers.txt");
    try {
      Stream<String> lines = Files.lines(path);
      lines.forEach(System.out::println);
    } catch (IOException e) { }
  }

}
```

Outline

Introduction

Stream Sources

Intermediate Operations

Terminal Operations

Further Stream Examples

Length-Preserving Operations

- Apply a function to each element
  - → `Stream<R> map(Function<? super T,? extends R> f)`
  - → `IntStream mapToInt(ToIntFunction<? super T> f)`
    (same for `LongStream` and `DoubleStream`)
- Sort the elements
  - → `Stream<T> sorted()`
  - → `Stream<T> sorted(Comparator<? super T> comparator)`
- Perform an action on each element (no terminal operation)
  - → `Stream<T> peek(Consumer<? super T> action)`

Length-Reducing Operations

- Remove duplicates
  - → `Stream<T> distinct()`
- Remove prefix or suffix
  - → `Stream<T> limit(long maxLength)`
  - → `Stream<T> skip(long offset)`
- Remove element that do not match the given predicate
  - → `Stream<T> filter(Predicate<? super T> predicate)`

Outline

Introduction

Stream Sources

Intermediate Operations

Terminal Operations

Further Stream Examples

ForEach Iteration

- Perform an action for each element
  - → `void forEach(Consumer<? super T> action)`
- Mostly used for side-effects
- This operation is similar to the intermediate operation `peek(action)`, but `forEach(action)` terminates the stream
- For parallel stream pipelines, this operation does not guarantee to respect the order of the elements

Processing Students I

```
public class ParallelStream {

  public static void main(String[] args) {

    Stream.iterate(1, x -> x+1).limit(10)
      .forEach(System.out::println);
    // prints 1 2 3 4 5 6 7 8 9 10

    Stream.iterate(1, x -> x+1).limit(10)
      .parallel()
      .forEach(System.out::println);
    // prints for example 7 6 3 1 4 8 2 5 10 9
  }
}
```

Predicates and Queries

- Check if all/some/no elements match a predicate
  - → `boolean allMatch(Predicate<? super T> predicate)`
  - → `boolean anyMatch(Predicate<? super T> predicate)`
  - → `boolean noneMatch(Predicate<? super T> predicate)`
- Find an element that matches a predicate
  - → `Optional<T> findFirst()`
  - → `Optional<T> findAny()`
- Find the minimal/maximal element
  - → `Optional<T> min(Comparator<? super T> comparator)`
  - → `Optional<T> max(Comparator<? super T> comparator)`
- Count all elements
  - → `long count()`

Collecting Elements

- Collect all elements into an array
  - → `Object[] toArray()`
- Collect all elements into an object of type `R` using a instance of the class `Collector<T,A,R>`
  - → `R collect(Collector<? super T,A,R> collector)`
- The most common collectors are implemented in the class `Collectors`
  - → `Collector<T,?,List<T>> toList()`
  - → `Collector<T,?,Set<T>> toSet()`
  - → `Collector<String,?,String> joining()`
  - → `Collector<String,?,String> joining(String delim)`

Reducing Elements

- Apply an associative accumulation function f pairwise to all elements $e_1, \ldots, e_n$ of a stream and return a single value
- Example: $n = 5$

$$f(f(f(f(e_1, e_2), e_3), e_4), e_5)$$

- For $n = 1$, e_1 is returned
- For $n = 0$, the return value depends on the method in use:
  - → `Optional<T> reduce(BinaryOperator<T> f)`
  - → `T reduce(T identity, BinaryOperator<T> f)`

Outline

Introduction

Stream Sources

Intermediate Operations

Terminal Operations

Further Stream Examples

Processing Students I

```
public class Student {

  private String firstName;
  private String lastName;
  private int semester;
  private double averageGrade;

  private static Random random = new Random();

  private static String[] firstNames = {"Tom", "Ben", "Joe", "
      Andy", "Pete", "Sam"};
  private static String[] lastNames = {"Smith", "Miller", "
      Jordan", "Wright", "Bush"};

  public Student() {
    this.firstName = firstNames[random.nextInt(firstNames.
        length)];
```

Processing Students II

```
    this.lastName = lastNames[random.nextInt(lastNames.length)
        ];
    this.semester = random.nextInt(10) + 1;
    this.averageGrade = 5*random.nextDouble() + 1;
  }

  public static void main(String[] args) {

    // Compute a string containing a sorted list of 10 first/
        last names of students in 6th semester in upper-case
    String names = Stream.generate(Student::new)
      .filter(s -> s.semester == 6)
      .limit(10)
      .peek(s -> System.out.println(s.semester)) // for
          testing
      .sorted(Comparator.comparing((Student s) -> s.lastName).
          thenComparing((Student s) -> s.firstName))
      .map(s -> s.firstName + " " + s.lastName)
```

Processing Students III

```
      .map(String::toUpperCase)
      .collect(Collectors.joining(", "));

    System.out.println(names);

    // Print the average of 100 rounded grades of students
        with average grade >= 4.0
    Stream.generate(Student::new)
      .limit(100)
      .filter(s -> s.averageGrade >= 4.0)
      .mapToDouble(s -> s.averageGrade)
      .map(d -> Math.round(d))
      .average()
      .ifPresent(System.out::println);
  }

}
```