301AA - Advanced Programming
[AP-2017]

Lecturer: Andrea Corradini andrea@di.unipi.it
Tutor: Lillo Galletta galletta@di.unipi.it

Department of Computer Science, Pisa
Academic Year 2017/18

AP-2017-18: Streams in Java 8
Streams in Java 8

The `java.util.stream` package provides utilities to support functional-style operations on streams of values. **Streams** differ from **collections** in several ways:

- **No storage.** A stream is not a data structure that stores elements; instead, it conveys elements from a **source** (a data structure, an array, a generator function, an I/O channel, ...) through a **pipeline** of computational operations.

- **Functional in nature.** An operation on a stream produces a result, but does not modify its source.
Streams in Java 8 (cont’d)

- **Laziness-seeking.** Many stream operations, can be implemented lazily, exposing opportunities for optimization. Stream operations are divided into **intermediate** (stream-producing) operations and **terminal** (value- or side-effect-producing) operations. *Intermediate operations are always lazy.*
- **Possibly unbounded.** Collections have a finite size, streams need not. Short-circuiting operations such as `limit(n)` or `findFirst()` can allow computations on infinite streams to complete in finite time.
- **Consumable.** The elements of a stream are only visited once during the life of a stream. Like an `Iterator`, a new stream must be generated to revisit the same elements of the source.
Pipelines

• A typical pipeline contains
  – A source, producing (by need) the elements of the stream
  – Zero or more intermediate operations, producing streams
  – A terminal operation, producing side-effects or non-stream values

• Example of typical pattern: filter / map / reduce

```java
double average = listing  // collection of Person
    .stream()              // stream wrapper over a collection
    .filter(p -> p.getGender() == Person.Sex.MALE)  // filter
    .mapToInt(Person::getAge)       // extracts stream of ages
    .average()                      // computes average (reduce/fold)
    .getAsDouble();               // extracts result from OptionalDouble
```
Anatomy of the Stream Pipeline

- A Stream is processed through a pipeline of operations
- A Stream starts with a source
- **Intermediate methods** are performed on the Stream elements. These methods produce Streams and are not processed until the **terminal method** is called.
- The Stream is considered **consumed** when a terminal operation is invoked. No other operation can be performed on the Stream elements afterwards
- A Stream pipeline may contain some **short-circuit methods** (which could be intermediate or terminal methods) that cause the earlier intermediate methods to be processed only until the short-circuit method can be evaluated.
Stream sources

Streams can be obtained in a number of ways:

- From a **Collection** via the `stream()` and `parallelStream()` methods;
- From an array via `Arrays.stream(Object[])`;
- From static factory methods on the stream classes, such as `Stream.of(Object[])`, `IntStream.range(int, int)` or `Stream.iterate(Object, UnaryOperator)`;
- The lines of a file can be obtained from `BufferedReader.lines()`;
- Streams of file paths can be obtained from methods in `Files`;
- Streams of random numbers can be obtained from `Random.ints()`;
- Generators, like `generate` or `iterate`;
- Numerous other methods in the JDK...
Intermediate Operations

- An intermediate operation keeps a stream open for further operations. Intermediate operations are lazy.
- Several intermediate operations have arguments of **functional interfaces**, thus **lambdas** can be used.

```java
Stream<T> filter(Predicate<? super T> predicate)  // filter
IntStream mapToInt(ToIntFunction<? super T> mapper)  // map f:T -> int
<R> Stream<R> map(Function<? super T,? extends R> mapper)  // map f:T->R
Stream<T> peek(Consumer<? super T> action)  //performs action on elements
Stream<T> distinct()  // remove duplicates – stateful
Stream<T> sorted()  // sort elements of the stream – stateful
Stream<T> limit(long maxSize)  // truncate
Stream<T> skip(long n)  // skips first n elements
```
Using peek...

• **peek** does not affect the stream
• A typical use is for debugging

```java
IntStream.of(1, 2, 3, 4)
  .filter(e -> e > 2)
  .peek(e -> System.out.println("Filtered value: " + e))
  .map(e -> e * e)
  .peek(e -> System.out.println("Mapped value: " + e))
  .sum();
```
Terminal Operations

- A **terminal operation** must be the final operation on a stream. Once a terminal operation is invoked, the stream is consumed and is no longer usable.
- Typical: collect values in a data structure, reduce to a value, print or other side effects.

```java
void forEach(Consumer<? super T> action)

Object[] toArray()

T reduce(T identity, BinaryOperator<T> accumulator) // fold

Optional<T> reduce(BinaryOperator<T> accumulator) // fold

Optional<T> min(Comparator<? super T> comparator)

boolean allMatch(Predicate<? super T> predicate) // short-circuiting

boolean anyMatch(Predicate<? super T> predicate) // short-circuiting

Optional<T> findAny() // short-circuiting
```
From Reduce to Collect: Mutable Reduction

- Suppose we want to concatenate a stream of strings.
- The following works but is highly inefficient (it builds one new string for each element):

  ```java
  String concatenated = listOfStrings.stream()
  .reduce("", String::concat);
  ```

- Better to “accumulate” the elements in a mutable object (a StringBuilder, a collection, ...)
- The **mutable reduction operation** is called `collect()`. It requires three functions:
  - a **supplier** function to construct new instances of the result container,
  - an **accumulator** function to incorporate an input element into a result container,
  - a **combining function** to merge the contents of one result container into another.

  ```java
  <R> R collect( Supplier<R> supplier,
  BiConsumer<R, ? super T> accumulator,
  BiConsumer<R, R> combiner);
  ```
Mutable reductions: examples

• Collecting the String representations of the elements in a stream into an ArrayList:

```java
ArrayList<String> strings = new ArrayList<>();
for (T element : stream) {
    strings.add(element.toString());
}
```

```java
ArrayList<String> strings =
    stream.collect(() -> new ArrayList<>(), //Supplier
    (c, e) -> c.add(e.toString()), // Accumulator
    (c1, c2) -> c1.addAll(c2)); //Combining
```

```java
List<String> strings = stream.map(Object::toString)
    .collect(ArrayList::new, ArrayList::add,ArrayList::addAll);
```
Mutable reductions: Collectors

• Method `collect` can also be invoked with a `Collector` argument:

```java
<R,A> R collect(Collector<? super T,A,R> collector)
```

• A `Collector` encapsulates the functions used as arguments to `collect(Supplier, BiConsumer, BiConsumer)`, allowing for reuse of collection strategies and composition of collect operations.

The following will accumulate strings into an `ArrayList`:

```java
List<String> asList = stringStream.collect(Collectors.toList());
```

The following will classify Person objects by city:

```java
Map<String, List<Person>> peopleByCity = personStream.collect(Collectors.groupingBy(Person::getCity));
```
Infinite Streams

• Streams wrapping collections are finite
• Infinite streams can be generated with:
  – iterate
  – generate

static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)

// Example: summing first 10 elements of an infinite stream
int sum = Stream.iterate(0,x -> x+1).limit(10).reduce(0,(x,s) -> x+s);

static <T> Stream<T> generate(Supplier<T> s)

// Example: printing 10 random numbers
Stream.generate(Math::random).limit(10).forEach(System.out::println);
Parallelism

• Streams facilitate parallel execution
• Stream operations can execute either in serial (default) or in parallel

```java
double average = persons //average age of all male
.parallelStream() // members in parallel
.filter(p -> p.getGender() == Person.Sex.MALE)
.mapToInt(Person::getAge)
.average()
.getAsDouble();
```

• The runtime support takes care of using multithreading for parallel execution, in a transparent way
• If operations don’t have side-effects, thread-safety is guaranteed even if non-thread-safe collections are used (e.g.: ArrayList)
Parallelism (2)

• Concurrent mutable reduction supported for parallel streams
  – Suitable methods of Collector

• Order of processing stream elements depends on serial/parallel execution and intermediate operations

```java
Integer[] intArray = {1, 2, 3, 4, 5, 6, 7, 8};
List<Integer> listOfIntegers = new ArrayList<>(Arrays.asList(intArray));
listOfIntegers.stream()
    .forEach(e -> System.out.print(e + " "));
// prints: 1 2 3 4 5 6 7 8
listOfIntegers.parallelStream()
    .forEach(e -> System.out.print(e + " "));
// may print: 3 4 1 6 2 5 7 8
```
Streams from collections

• A stream wrapping a collection uses a **Splitterator** over the collection

• This is the parallel analogue of an **Iterator**: it describes a (possibly infinite) collection of elements with support for
  – sequentially advancing,
  – *applying an action* to the next or to all remaining elements
  – splitting off some portion of the input into another splitterator which can be processed in parallel.

• At the lowest level, all streams are driven by a splitterator.
Critical issues

• Non-interference
  – Behavioural parameters (like lambdas) of stream operations should not affect the source (*non-interfering behaviour*)
  – Risk of ConcurrentModificationExceptions, even if in single thread

• Stateless behaviours
  – Statless behaviour for intermediate operations is encouraged, as it facilitates parallelism, and functional style, thus maintenance

• Parallelism and thread safety
  – For parallel streams with side-effects, ensuring thread safety is the programmers’ responsibility
try {
    List<String> listOfStrings =
        new ArrayList<>(Arrays.asList("one", "two"));
    // This will fail as the peek operation will attempt to add the
    // string "three" to the source after the terminal operation has
    // commenced.
    String concatenatedString = listOfStrings
        .stream()
        // Don't do this! Interference occurs here.
        .peek(s -> listOfStrings.add("three"))
        .reduce((a, b) -> a + " " + b)
        .get();
    System.out.println("Concatenated string: " + concatenatedString);
} catch (Exception e) {
    System.out.println("Exception caught: " + e.toString());
}