Lecture

Streams in Java

Nguyen Anh Tu

October 23rd, 2019

NAZARBAYEV UNIVERSITY

Outline

- Introduce Java Lambda Expressions
- Java Streams

Streams API

- In addition to lambda expressions Java 8 also introduced the streams API.
- Allows for declarative manipulation of collections of data
- Compactly and transparently express iteration through a collection of data
- Collections have a method stream() that returns a stream from the collection
- Built in parallelization with parallellStream()
- Allows us to chain together a number of data-processing operations in a way that is concise and readable

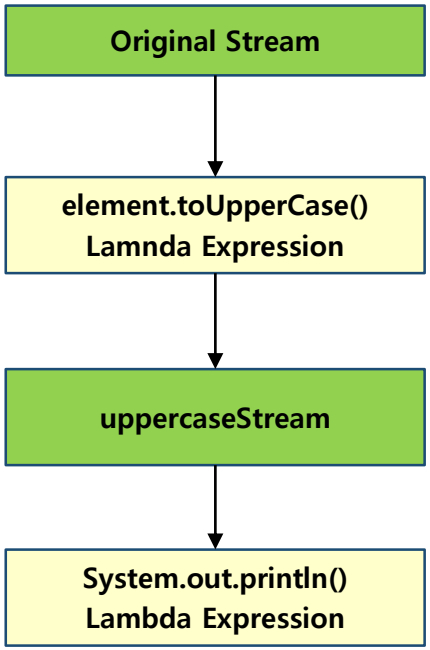
Stream creation

- There are many ways to obtain a Java Stream. One of the most common ways to obtain a Stream is from a Java Collection

```
List<String> items = new ArrayList<String>();

items.add("one");
items.add("two");
items.add("three");

Stream<String> stream = items.stream();
```



Stream<String> stream = items.stream();

Stream<String> upperStream = stream.map(…);

upperStream.forEach(…);

Intermediate and Terminal operations

- A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.
- Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined.
  - When you add a intermediate operation to a stream, you get a new stream back as result. The new stream represents the stream of elements resulting from the original stream with the intermediate operation applied
- Terminal operations mark the end of the stream and return the result.
  - The terminal operations of the Java Stream interface typicall return a single value.
  - Once the terminal operation is invoked on a Stream, the iteration of the Stream and any of the chained streams will get started.
  - Once the iteration is done, the result of the terminal operation is returned.

Intermediate and Terminal operations (cont'd)

```
// Intermediate Operations
Stream<T> filter(Predicate<T>);
<R> Stream<R> map(Function<T, R>);
IntStream mapToInt(ToIntFunction<T>);
LongStream mapToLong(ToLongFunction<T>);
DoubleStream mapToDouble(ToDoubleFunction<T>);
<R> Stream<R> flatMap(Function<T, Stream<R>>);
IntStream flatMapToInt(Function<T, IntStream>);
LongStream flatMapToLong(Function<T, LongStream>);
DoubleStream flatMapToDouble(Function<T, DoubleStream>);
Stream<T> distinct();
Stream<T> sorted();
Stream<T> sorted(Comparator<T>);
Stream<T> peek(Consumer<T>);
Stream<T> limit(long);
Stream<T> skip(long);
```

```
// Terminal Operations
void forEach(Consumer<T>);         // Ordered only for sequential streams
void forEachOrdered(Consumer<T>);  // Ordered if encounter order exists
Object[] toArray();
<A> A[] toArray(IntFunction<A[]> arrayAllocator);
T reduce(T, BinaryOperator<T>);
Optional<T> reduce(BinaryOperator<T>);
<U> U reduce(U, BiFunction<U, T, U>, BinaryOperator<U>);
<R, A> R collect(Collector<T, A, R>); // Mutable Reduction Operation
<R> R collect(Supplier<R>, BiConsumer<R, T>, BiConsumer<R, R>);
Optional<T> min(Comparator<T>);
Optional<T> max(Comparator<T>);
long count();
boolean anyMatch(Predicate<T>);
boolean allMatch(Predicate<T>);
boolean noneMatch(Predicate<T>);
Optional<T> findFirst();
Optional<T> findAny();
```

filter()

- The Java Stream filter() can be used to filter out elements from a Java Stream.
  - The filter method takes a Predicate which is called for each element in the stream.
  - If the element is to be included in the resulting Stream, the Predicate should return true. If the element should not be included, the Predicate should return false.
- Example:

```
Stream<String> longStringsStream = stream.filter((value) -> {
    return value.length() >= 3;
});
```

map()

- The Java Stream map() method converts (maps) an element to another object.
  - For instance, if you had a list of strings it could convert each string to lowercase, uppercase, or to a substring of the original string, or something completely else
- Example:

```
List<String> list = new ArrayList<String>();
Stream<String> stream = list.stream();

Stream<String> streamMapped = stream.map((value) -> value.toUpperCase());
```

collect()

- The Java Stream collect() method is a terminal operation that starts the internal iteration of elements, and collects the elements in the stream in a collection or object of some kind

- Example:

```
List<String> stringList = new ArrayList<String>();

stringList.add("One flew over the cuckoo's nest");
stringList.add("To kill a muckingbird");
stringList.add("Gone with the wind");

Stream<String> stream = stringList.stream();

List<String> stringsAsUppercaseList = stream
.map(value -> value.toUpperCase())
.collect(Collectors.toList());

System.out.println(stringsAsUppercaseList);
```

reduce()

- The Java Stream reduce() method is a terminal operation that can reduce all elements in the stream to a single element.
  - *reduce* operation applies a Binary operator to each element in the stream where the first argument to the operator is the return value of the previous application and second argument is the current stream element.
- Example:

```
// Creating list of integers
List<Integer> array = Arrays.asList(-2, 0, 4, 6, 8);

// Finding sum of all elements
int sum = array.stream().reduce(0, (element1, element2) -> element1 + element2);

// Displaying sum of all elements
System.out.println("The sum of all elements is " + sum);
```

When to use a parallel stream

- When operations are independent, and
- Either or both:
  - Operations are computationally expensive
  - Operations are applied to many elements of efficiently splittable data structures
- **Always measure before and after parallelizing!**

Streams Example

- Suppose you want to chain together the two operations of applying a conditional to the objects in a collection and then returning a new collection with only the true results

- Notice here that filter is part of the streams API and

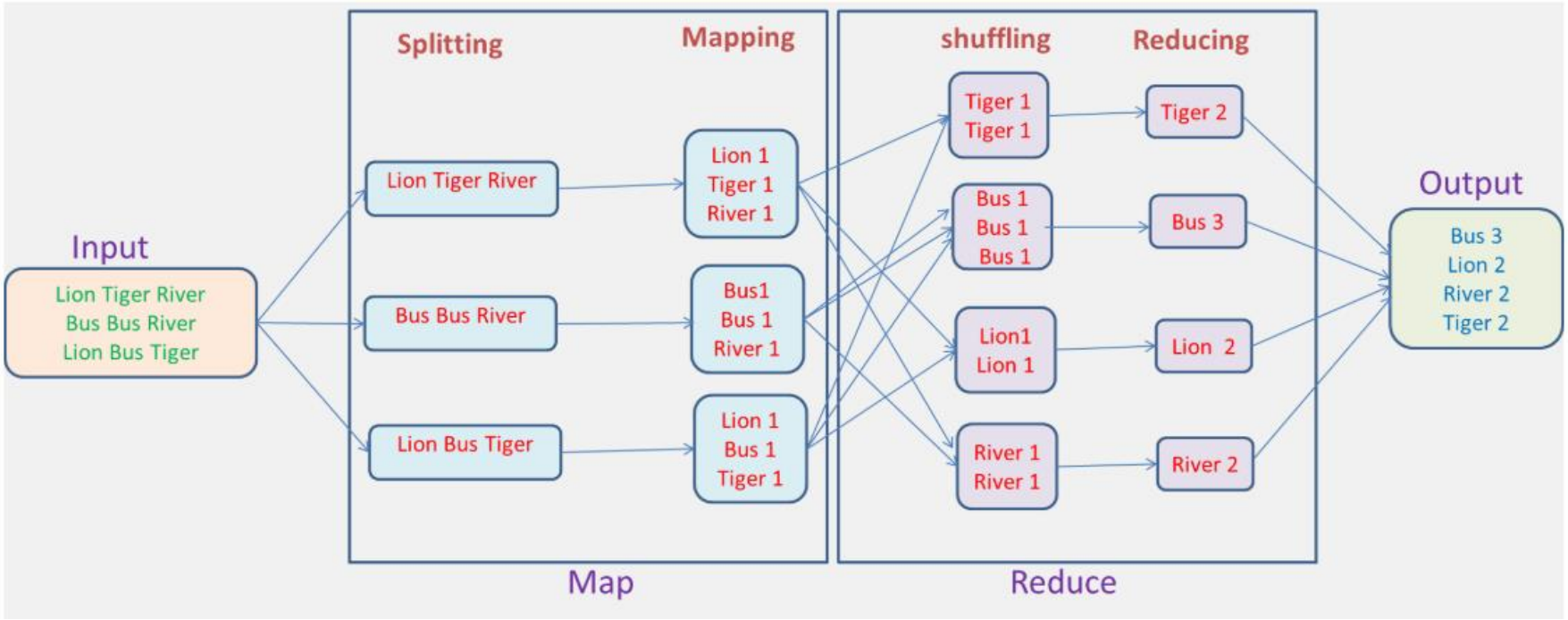
```
List<String> users = Arrays.asList("alice","bob","Charlie");
filteredUsers = users.stream()
                     .filter((String s) -> s.length() > 3)
                     .collect(Collectors.toList());
```

Map and Reduce - Example

```
int n = (int) toks.stream().map((String x) -> x.toUpperCase())
                  .filter((String x) -> x.contains("O"))
                  .mapToInt((String x) -> x.length())
                  .reduce(0,(a,b) -> a+b);
```

- We can also perform a reduction using a binary function

MapReduce- word count example



https://data-flair.training/blogs/map-only-job-in-hadoop-mapreduce/

Custom Sort

- Oftentimes, we want to sort a collection using a custom comparison function
- Comparator<T> is an example of a functional interface
  - defines a method in compare(T o1, T o2)
- Collections.sort is a static method that sorts a collection in place

```
String[] a = {"Alice", "Bob", "Mary", "Joe", "Alexander"};
List<String> l = Arrays.asList(a);

Collections.sort(l,
  (u,v) -> u.substring(1).compareTo(v.substring(1))
);

// Prints Mary, Alexander, Alice, Bob, Joe
l.stream().forEach(System.out::println);
```

Method References

```
Stream<String> l = Files.lines(Paths.get("lambtest.txt"));
l.forEach(System.out::println);
```

- *use case*: where a lambda function calls an existing method
- Succinct way to generate function objects
- Most method references refer to static methods

Method references (cont'd)

- An instance method of a particular object (*bound*)
  - objectRef::methodName
- An instance method whose receiver is unspecified (*unbound*)
  - ClassName::instanceMethodName
  - The resulting function has an extra argument for the receiver
- A static method
  - ClassName::staticMethodName
- A constructor
  - ClassName::new

Method reference examples

Kind	Examples
Bound instance method	createBicyclesList().stream() .sorted(bikeFrameSizeComparator :: compare)
Unbound instance method	numbers.stream().sorted(Integer::compareTo)
Static method	Value.forEach(Math :: cos)
Constructor	LinkedHashSet<String> :: new
Array constructor	String[] :: new

Variance (Computer Science)

- Sometimes we create new classes (types) from existing classes (types)
  - For example, if Person is a class then List<Person> is a new class: lists of people
  - The angle bracket notation indicates a *generic*
  - Generics allow us to use types as parameters in the definition of new types
- The question is how should types defined by generics be related through inheritance
- *Variance* refers to the different possibilities

Variance (Example)

- Suppose that B and A are classes and that B extends A
  - Write this as B ≤ A
- Intuitively, we would like to believe that List<B> ≤ List<A> (especially if the lists are immutable)
  - This would be an example of a *covariant* type constructor, that is the type constructor maintains the order relation ≤
  - This is *not* the case in Java
  - By default the relationship between B and A does not imply anything about the list types
- A *contravariant* type constructor would reverse the order relation
  - B ≤ A → MyType<A> ≤ MyType<B>
- What would be an example of a contravariant type constructor?

Function Types

- With Java lambda expressions, we have seen examples of “function types”
  - The type of functions from A and B, (A → B)
  - In a functional programming language (Haskell, OCaml, Scala etc.) it is usually much easier to define these *higher kinded types*
  - Lambda expressions and Functional Interfaces gives us an approximation of these concepts in Java
- Function types are contravariant in the type of the argument
  - Suppose that B ≤ A
  - Then (A → C) ≤ (B → C)
- Imagine all the predicates on the type Cat (Cat → Boolean), then imagine the predicates on the type Animal

Functors

- Terminology comes from a branch of mathematics called *Category Theory*
  - All about abstract reasoning with diagrams
- Category Theory is closely related to programming languages, proof theory, and logic via the *Curry-Howard* correspondence
- A *Functor* is a map between categories (maps over the objects of the category and the arrows) that preserves composition
- In our case, we can think of Functors as type constructors that we can map over
  - If *f* : A → B then, for a functor *F* we have, *F*(*f*) : *F*(A) → *F*(B)
  - This is usually called map or fmap
  - Example, lists