Introduction to Java 8 Stream API

Sidlyarevich Vladislav

Contacts/samples

- https://github.com/kuksenko/jdk8-lambda-samples
- https://github.com/vlsidlyarevich/Stream-API-Examples
- https://www.youtube.com/watch?v=O8oN4KSZEXE
- https://www.youtube.com/watch?v=i0Jr2l3jrDA

Java 8 language features

- Lambdas and Functional Interfaces
- Interface’s Default and Static Methods
- Method References
- Repeating annotations

Java 8 language features

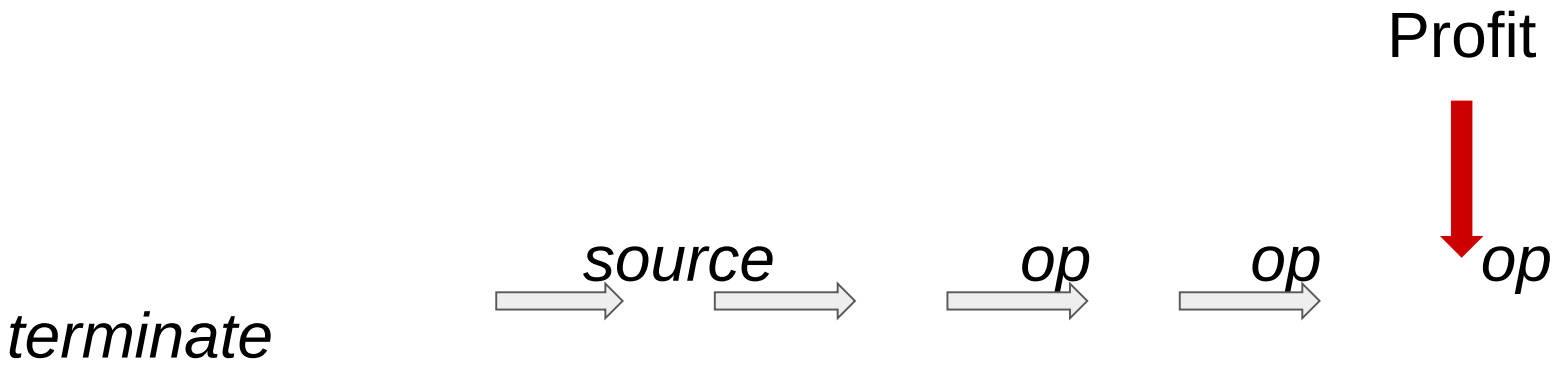
- Date/Time API (JSR 310)
- Nashorn JavaScript engine
- Base64
- Parallel Arrays
- Concurrency

Java 8 language features

STREAM API!

```
Set<Person> freshBlood = new HashSet<>();
for (Person person : team) {
        if (person.age <= 25) {
                freshBlood.add(person);
                }
        }
         List<Person> sortedFreshBlood = new ArrayList<>(freshBlood);
        Collections.sort(sortedFreshBlood, new Comparator<Person>() {
                @Override
                 public int compare(Person o1, Person o2) {
                return Integer.compare(o1.age, o2.age);
                }
        });
        for (Person person : sortedFreshBlood) {
                System.out.println(person.name);
        }
```

```
team.stream()
    .filter(person -> person.age <= 25)
    .collect(Collectors.toSet())
    .stream()
    .sorted(comparing(person -> person.age))
    .forEach(person -> System.out.println(person.name));
```


Profit
source
op
op
op
terminate

sources: collections, iterators, api’s

operations: filter, map, reduce, etc

sinks: collections, locals

What is Stream?

Multiplicity of values

Lazy

Single use

Not mutate the source

Ordered/Unordered

Parallel/Sequential

IntStream, DoubleStream, LongStream

What is Stream?

Sequence of elements − A stream provides a set of elements of specific type in a sequential manner. A stream gets/computes elements on demand. It never stores the elements.

Source − Stream takes Collections, Arrays, or I/O resources as input source.

Aggregate operations − Stream supports aggregate operations like filter, map, limit, reduce, find, match, and so on.

Pipelining − Most of the stream operations return stream itself so that their result can be pipelined. These operations are called intermediate operations and their function is to take input, process them, and return output to the target. collect() method is a terminal operation which is normally present at the end of the pipelining operation to mark the end of the stream.

Automatic iterations − Stream operations do the iterations internally over the source elements provided, in contrast to Collections where explicit iteration is required.

Stream pipeline

sources: **team, stream** *operations:* **filter, sorted, forEach**

sinks: **collect**

```
team.stream()
    .filter(person -> person.age <= 25)
     .collect(Collectors.toSet())
     .stream()
     .sorted(comparing(person -> person.age))
     .forEach(person -> System.out.println(person.name));
```

Sources

Collections

Popular API’s (For example Regex)

```
String sentence = "Java 8 Stream tutorial";
      Stream<String> regExpStream
            = Pattern.compile("\\w").splitAsStream(sentence);
```

Sources

Infinite

```
Stream iterateStream = Stream.iterate(0, n -> n + 1).limit(2);
```

Function

```
Stream generatedStream = Stream.generate(Math::random).limit(5L);
```

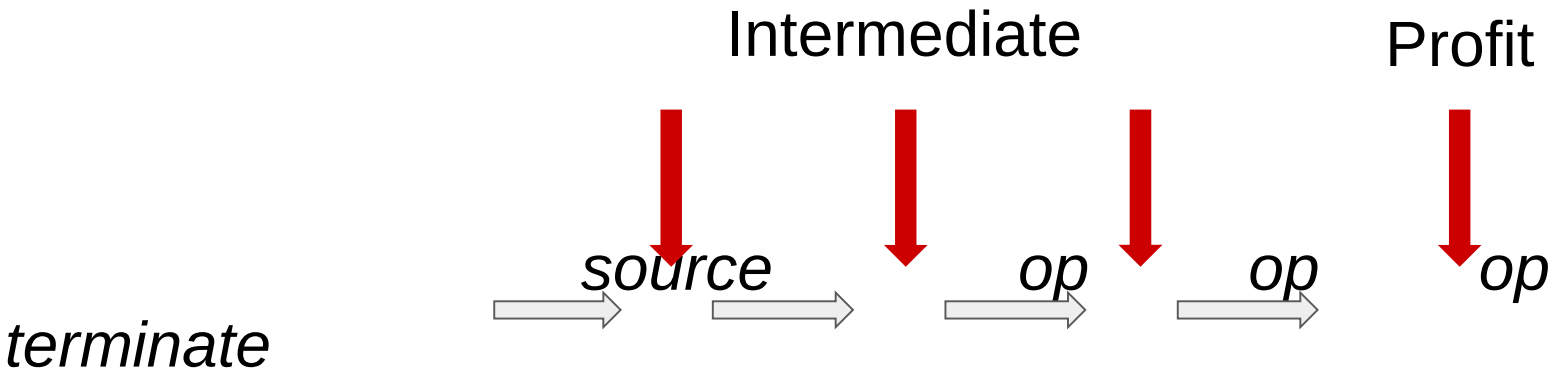
```
List<String> arrList = new ArrayList<>();
Stream<String> arrListStream = arrList.stream();  //sized, ordered


List<String> linkedList = new LinkedList<>();
Stream<String> linkedListStream = linkedList.stream();  //sized, ordered


Set<String> hashSet = new HashSet<>();
Stream<String> hashSetStream = hashSet.stream();  //sized, distinct


Set<String> linkedHashSet = new LinkedHashSet<>();
Stream<String> linkedHashSetStream = linkedHashSet.stream();  //sized, distinct, ordered


Set<String> treeSet = new TreeSet<>();
Stream<String> treeSetStream = treeSet.stream();  //sized, distinct, sorted, ordered
```


Intermediate
Profit
source
op
op
op
terminate

- Stream<S> s.distinct();
- Stream<S> s.filter(Predicate <S>);
- Stream<T> s.map(Function<S, T>);
- Stream<T> s.flatMap(Function<S, Stream<T>>);
- Stream<S> s.peek(Consumer<S>)
- Stream<S> s.sorted()
- Stream<S> s.limit(long);
- Stream<S> s.skip(long);

- Stream<S> s.distinct();
- Stream<S> s.filter(Predicate <S>);
- Stream<T> s.map(Function<S, T>);
- Stream<T> s.map(Function<S, Stream<T>>);
- Stream<S> s.peek(Consumer<S>)
- Stream<S> s.sorted()
- Stream<S> s.limit(long);
- Stream<S> s.skip(long);

- Stream<S> s.unordered();
- Stream<S> s.parallel();
- Stream<S> s.sequential();

Terminal operations = Profit

Terminal operations

iteration: forEach, iterator

search: findFirst, findAny

check: allMatch, anyMatch, noneMatch

aggregation: reduction, collectors

Short-circuiting

All find*

All match*

limit

```
int number = Stream.iterate(1, n -> n * 2)
                                        .filter(n -> n % 1024 == 0)
                        .findFirst().get();
```

Examples

```
List<Transaction> groceryTransactions = new Arraylist<>();
for(Transaction t: transactions) {
        if(t.getType() == Transaction.GROCERY) {
                groceryTransactions.add(t);
        }
}
Collections.sort(groceryTransactions, new Comparator() {
        public int compare(Transaction t1, Transaction t2) {
                return t2.getValue().compareTo(t1.getValue());
        }
});
List<Integer> transactionIds = new ArrayList<>();
for(Transaction t: groceryTransactions) {
        transactionsIds.add(t.getId());
}
```

Examples

```
List<Transaction> groceryTransactions = new Arraylist<>();

transactions.Stream()
                .filter(t.getType() == Transaction.GROCERY)
```

Examples

```
List<Transaction> groceryTransactions = new Arraylist<>();

transactions.Stream()
                .filter(t.getType() == Transaction.GROCERY)
                .sorted(comparing(t -> t.getValue)
```

Examples

```
List<Transaction> groceryTransactions = new Arraylist<>();

transactions.Stream()
                .filter(t.getType() == Transaction.GROCERY)
                .sorted(comparing(t -> t.getValue).reversed)
```

Examples

```
List<Transaction> groceryTransactions = new Arraylist<>();

transactions.Stream()
                .filter(t.getType() == Transaction.GROCERY)
                .sorted(comparing(t -> t.getValue).reversed)
                .map(transaction -> transaction.getId())
```

Examples

```
List<Transaction> groceryTransactions = new Arraylist<>();

transactions.Stream()
                .filter(t.getType() == Transaction.GROCERY)
                .sorted(comparing(t -> t.getValue).reversed)
                .map(transaction -> transaction.getId())
                .collect(Collectors.toList())
```

Examples

```
List<Transaction> groceryTransactions = new Arraylist<>();

transactions.Stream()
                .filter(t.getType() == Transaction.GROCERY)
                .sorted(comparing(Transaction::getValue).reversed)
                .map(Transaction::getId())
                .collect(Collectors.toList())
```

Thanks for your attention!