Stream API

Programski jezici II

Stream API

- Agregatna operacija je operacija koja se vrši nad svim elementima kolekcije. Pipeline je sekvenca agregatnih operacija. Pipeline se sastoji iz sljedećih komponenti:
  - izvor (eng. *source*) – izvor može biti kolekcija, niz, I/O operacija ili generatorska funkcija .
  - nula ili više međuoperacija (eng. *intermediate operations*) – međuoperacije uvijek kreiraju novi stream (koji metode i vraćaju kao rezultat svog izvršavanja).
  - terminalna operacija (eng. *terminal operation*) – terminalna operacija vraća rezultat koji nije stream, poput vrijednosti primitivnog tipa, kolekcije ili uopšte ne vraća rezultat. Nakon izvršavanja terminalne operacije stream se više ne može koristiti.
- Međuoperacije se mogu podijeliti na *stateless* i *stateful*.
  - *Stateless* operacije ne čuvaju stanje prethodno procesiranog elementa pri procesiranju tekućeg elementa, tj. svaki element može biti procesiran nezavisno od operacija nad drugim elementima.
  - *Stateful* operacije mogu koristiti stanje prethodno procesiranih elemenata pri procesiranju tekućeg elementa.
- Pored toga, potrebno je razlikovati i *short-circuiting međuoperacije. Short-circuiting* operacije su operacije koje pozvane nad beskonačnim stream-om mogu kreirati konačan stream.

Stream API

- Agregatne operacije, poput forEach, podsjećaju na iteratore. Ipak, postoje značajne razlike:
  - agregatne operacije koriste interno iteriranje – one ne sadrže metode poput next koja kao rezultat vraća sljedeći element u kolekciji. Ovo znači da programer definiše nad kojom kolekcijom se vrši iteracija, dok Java VM određuje kako se tačno vrši iteracija. Kod eksternog iteriranja (npr. primjenom iteratora) programer definiše oboje. Isto tako, eksterno iteriranje može obilaziti elemente kolekcije samo sekvencijalno, dok takvog ograničenja nema kod internog iteriranja.
  - agregatne operacije procesiraju elemente stream-a – ne procesiraju direktno elemente kolekcije. Iz tog razloga se i nazivaju stream operacijama.
  - agregatne operacije podržavaju rad s lambda izrazima kao parametrima – za većinu agregatnih operacija moguće je specificirati lambda izraze kao parametre. Ovo omogućava promjenu i prilagođavanje ponašanja agregatne operacije.

Stream API

- Stream je sekvenca elemenata koja podržava sekvencijalne i paralelne agregatne operacije. Iako podsjeća na kolekciju, postoje značajne razlike između stream-ova i kolekcija:
  - Za razliku od kolekcija, stream nije struktura podataka koja skladišti elemente. Umjesto toga, stream operiše sa vrijednostima iz izvora putem pipeline-a
  - Operacija nad stream-ovima ne mijenjaju njihov izvor, već uvijek vraćaju novi stream.
  - Operacije nad stream-ovima su *lazy* operacije kad god je to moguće. To znači da se iste ne izvršavaju dok god to nije potrebno. Na primjer, ako je potrebno pronaći samo prve dvije riječi čija je dužina veća ili jednaka 10, onda filter metoda neće nastaviti filtriranje nakon što pronađe drugu riječ koja zadovoljava uslov.
- BaseStream – osnovni interfejs Stream-ova, nasljeđuje ga Stream
- Pored generičkog Stream interfejsa, potrebno je razlikovati i sljedeće interfejse stream-ova:
  - IntStream – sekvenca vrijednosti primitivnog tipa int,
  - LongStream – sekvenca vrijednosti primitivnog tipa long,
  - DoubleStream – sekvenca vrijednosti primitivnog tipa double.
- Svi stream-ovi podržavaju sekvencijalne i paralelne agregatne operacije.

Stream API

- Stream-ovi imaju pogodnije metode nego liste
  - forEach, filter, map, reduce, min, sorted, distinct, limit, etc.
- Stream-ovi imaju korisne osobine koje listama nedostaju
  - Stream-ovi su „moćniji“, brži i memorijski efikasniji nego liste
  - Najvažnije osobine:
    - Lazy evaluacija
    - Automatska paralelizacija
    - Infinite (unbounded) stream-ovi
- Stream-ovi ne skladište podatke
  - Oni su programski wrapper-i oko postojećih izvora podataka

Stream API

- Interface Stream<T>

- boolean allMatch(Predicate<? super T> predicate)
- boolean anyMatch(Predicate<? super T> predicate)
- static <T> Stream.Builder<T> builder()
- <R,A> R collect(Collector<? super T,A,R> collector)
- <R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)
- static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)
- long count()
- Stream<T> distinct()
- static <T> Stream<T> empty()
- Stream<T> filter(Predicate<? super T> predicate)
- Optional<T> findAny()
- Optional<T> ()
- <R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)
- DoubleStream flatMapToDouble(Function<? super T,? extends DoubleStream> mapper)
- IntStream flatMapToInt(Function<? super T,? extends IntStream> mapper)
- LongStream flatMapToLong(Function<? super T,? extends LongStream> mapper)
- void forEach(Consumer<? super T> action)
- void forEachOrdered(Consumer<? super T> action)

Stream API

- static <T> Stream<T> generate(Supplier<T> s)
- static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)
- Stream<T> limit(long maxSize)
- <R> Stream<R> map(Function<? super T,? extends R> mapper)
- DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)
- IntStream mapToInt(ToIntFunction<? super T> mapper)
- LongStream mapToLong(ToLongFunction<? super T> mapper)
- Optional<T> max(Comparator<? super T> comparator)
- Optional<T> min(Comparator<? super T> comparator)
- boolean noneMatch(Predicate<? super T> predicate)
- static <T> Stream<T> of(T... values)
- static <T> Stream<T> of(T t)
- Stream<T> peek(Consumer<? super T> action)
- Optional<T> reduce(BinaryOperator<T> accumulator)
- T reduce(T identity, BinaryOperator<T> accumulator)
- <U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)
- Stream<T> skip(long n)
- Stream<T> sorted()
- Stream<T> sorted(Comparator<? super T> comparator)
- Object[] toArray()
- <A> A[] toArray(IntFunction<A[]> generator)

Stream API

```
package org.unibl.etf.streams;

import java.math.BigInteger;
import java.util.stream.Stream;

public class StreamCreation {
    public static void main(String[] args) {
        Stream<String> s1 = Stream.of("value");
        Stream<String> s2 = Stream.of("value 1", "value 2");
        Stream<String> s3 = Stream.empty();
        Stream<String> s4 = Stream.generate(() -> "element");
        Stream<Double> s5 = Stream.generate(Math::random);
        Stream<BigInteger> s6 = Stream.iterate(BigInteger.ZERO,
n -> n.intValue() < 100, n -> n.add(BigInteger.ONE));
        Stream<BigInteger> s7 = Stream.iterate(BigInteger.ZERO,
n -> n.add(BigInteger.ONE));
        Stream<String> s8 = Stream.ofNullable(null);
        Stream.Builder<String> builder = Stream.builder();
        Stream<String> s9 = sb.add("a").add("b").build();
    }
}
```

- s4, s5 i s7 referenciraju beskonačne stream-ove
- s3 i s8 referenciraju prazne stream/ove

Stream API

- pored metoda interfejsa Stream, postoje i metode drugih tipova koje se mogu koristiti za potrebe kreiranja stream-ova
- interfejs Collection ima dvije metode koje kreiraju Stream objekat:
  - metoda stream koja vraća sekvencijalni stream čiji je izvor data kolekcija i
  - metoda parallelStream koja vraća paralelni stream čiji je izvor data kolekcija. Ideja sa paralelnim stream-om jeste dobijanje istog rezultata, kao da je korišćen sekvencijalni stream, uz paralelizaciju koja je automatska. Odgovornost programera je da paralelnom stream-u proslijedi samo thread-safe operacije.

Stream API

```
public static void main(String[] args) {
    ArrayList<String> list = new ArrayList<>();
    list.add("one");
    list.add("two");
    Stream<String> s1 = list.stream();
    Stream<String> s2 = list.parallelStream();
    String[] array = new String[]{"one", "two"};
    Stream<String> s3 = Arrays.stream(array);
    IntStream s4 = "abc".chars();
    Path path = Paths.get("test.txt");
    try(Stream<String> lines = Files.lines(path)) {

    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Stream API

- Međuoperacije

- Stream<T> filter(Predicate<? super T> predicate)
- Stream<T> distinct()
- Stream<T> limit(long maxSize)
- Stream<T> skip(long n)
- <R> Stream<R> map(Function<? super T,? extends R> mapper)
- IntStream mapToInt(ToIntFunction<? super T> mapper)
- LongStream mapToLong(ToLongFunction<? super T> mapper)
- DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)
- <R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)
- IntStream flatMapToInt(Function<? super T,? extends IntStream> mapper)
- LongStream flatMapToLong(Function<? super T,? extends LongStream> mapper)
- DoubleStream flatMapToDouble(Function<? super T,? extends DoubleStream> mapper)
- Stream<T> sorted()
- Stream<T> sorted(Comparator<? super T> comparator)
- Stream<T> peek(Consumer<? super T> action)

Stream API

- Terminalne operacije

- void forEach(Consumer<? super T> action)
- void forEachOrdered(Consumer<? super T> action)
- Object[] toArray()
- <A> A[] toArray(IntFunction<A[]> generator)
- T reduce(T identity, BinaryOperator<T> accumulator)
- Optional<T> reduce(BinaryOperator<T> accumulator)
- <U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)
- <R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)
- <R,A> R collect(Collector<? super T,A,R> collector)
- Optional<T> min(Comparator<? super T> comparator)
- Optional<T> max(Comparator<? super T> comparator)
- long count()
- boolean anyMatch(Predicate<? super T> predicate)
- boolean allMatch(Predicate<? super T> predicate)
- boolean noneMatch(Predicate<? super T> predicate)
- Optional<T> findFirst()
- Optional<T> findAny()

Stream API

- Karakteristike stream-ova

- Stream-ovi ne skladište podatke
- Ne modifikuju strukturu na koju su povezani
- Dizajnirani za Lambde – sve Stream operacije koriste Lambde kao argumente
- Ne podržavaju indeksiran pristup
  - Moguće je zahtijevati prvi element, ali ne i drugi, treći, itd.
- Mogu dati izlaz u formi List-e ili niza
- Lazy
  - Većina Stream operacija se odlaže dok se ne utvrdi koliko podataka je potrebno
- Paralelizacija
  - Ako je Stream paralelan – operacija se mogu odvijati u paraleli
- Mogu biti *unbounded*

Stream API

- Karakteristike stream-ova

- Stream-ovi ne skladište podatke
- Ne modifikuju strukturu na koju su povezani
- Dizajnirani za Lambde – sve Stream operacije koriste Lambde kao argumente
- Ne podržavaju indeksiran pristup
  - Moguće je zahtijevati prvi element, ali ne i drugi, treći, itd.
- Mogu dati izlaz u formi List-e ili niza
- Lazy
  - Većina Stream operacija se odlaže dok se ne utvrdi koliko podataka je potrebno
- Paralelizacija
  - Ako je Stream paralelan – operacija se mogu odvijati u paraleli
- Mogu biti *unbounded*

Stream API

- for-each

```
for(Employee e: empList) {
e.setSalary(e.getSalary() * 11/10);
}
```

- forEach

```
empList().stream().forEach(e ->
e.setSalary(e.getSalary() * 11/10));

empList().stream().parallel().forEach(e ->
e.setSalary(e.getSalary() * 11/10));
```

Stream API

- Map
  - Transformacija Stream-a propuštanjem elemenata kroz Function

```
Stream<String> stream = list.stream();
stream.map(e -> e + e).forEach(e ->
System.out.println(e));
```

Stream API

- Reduce
  - proces kombinovanja svih elemenata stream-a u cilju kreiranja jedne vrijednosti (koja predstavlja rezultat operacije redukcije)

```
Stream<Integer> stream = list.stream();
Optional<Integer> result =
stream.stream().distinct().filter(e -> e >
100).reduce((a, b) -> a + b);
```

Stream API

- Collect
  - proces kreiranja kolekcije na osnovu Stream-a

```
List<Product> products = Arrays.asList(new Product("Shoes A", "Running Shoes", 20), new Product("Shoes B", "Running Shoes", 55), new Product("Shoes C", "Soccer", 30));
List<String> productsList = products.stream().map(Product::getName).collect(Collectors.toList());
```

Stream API

- Filter
  - Zadržavanje elemenata koji zadovoljavaju određeni Predicate

```
Stream<Integer> intStream = intList.stream();
intStream.filter(e -> e > 3).forEach(System.out::println);
```

Stream API

- findFirst
  - Vraća prvi element Stream-a ako postoji

```
Optional<Integer> optiInt = intStream.filter(e -> e > 8).findFirst();
if(optiInt.isPresent()){
                System.out.println(optiInt.get());
}
```

Stream API

- orElse
  - Vraća prvi element ne Stream-a postoji možemo vratiti predefinisani

```
Integer optiInt = intStream.filter(e -> e > 8).findFirst().orElse(0);
System.out.println(optiInt);
```