12 Streamy

Streamy, česky by se dalo říci *proudy*, představují praktické rozhraní pro práci s kolekcemi. Na streamy můžeme nahlížet jako na posloupnost hodnot, kterou postupně transformujeme, dokud nedostaneme kýžený výsledek. Streamy mají několik zajímavých vlastností:

- jednotlivé operace pracující s daty jsou dány deklarativně (říkáme, co chceme udělat, nikoliv jak nebo v jakém pořadí),
- operace se provádí líně (hodnoty jsou spočítány, jen je-li to potřeba),
- operace pracující se streamy, převádí jeden stream na druhý (operace jsou vyjádřeny jako postupné volání metod),
- streamy nijak nemění obsah kolekcí, ze kterých berou hodnoty,
- je možné pracovat s nekonečnými streamy,
- je možné data zpracovávat paralelně.

1 Práce se streamy

Nejdříve si představíme práci se streamy, které pracují s obecnými objekty, tj. hodnotami, jež jsou potomky třídy `Object`.

1.1 Vytvoření streamu

Stream je možné vytvořit mnoha způsoby, např. prostým výčtem:

```
Stream<String> streamOfWords = Stream.of("alfa", "beta", "gama", "delta", "epsilon");
```

Nebo můžeme získat stream z kolekce zavoláním metody `stream()`.

```
List<String> words = List.of("alfa", "beta", "gama", "delta", "epsilon");
Stream<String> streamOfWords = words.stream();
```

Případně můžeme stream nechat generovat na základě nějaké funkce:

```
Stream<Integer> oddNumbers = Stream.iterate(1, n -> n + 2);
```

V tomto případě metoda `iterate` bere jako svůj první argument počáteční hodnotu (např. `1`) a pro získání další hodnoty aplikuje na předchozí hodnotu zadanou funkci (např. `n -> n + 2`). V tomto případě se bude jednat o nekonečný stream.

Co se ve streamu nachází, můžeme zjistit například pomocí metody `forEach`, které předáme funkci jednoho argumentu, která je zavolána pro každý prvek ze streamu.

```
streamOfWords.forEach(word -> System.out.println(word));
streamOfWords.forEach(System.out::println); // kratší verze s využitím odkazů na metody
```

Poznámka 1 *Pozor! Stream můžeme použít nanejvýš jednou. Pokud bychom chtěli stream projít dvakrát, je nutné vytvořit jeho novou verzi.*

1.2 Transformace streamu

Práce se streamy je postavena na tom, že transformujeme jeden stream na druhý. K tomu nám slouží řada metod, které třída `Stream` má. Zmiňme si ty nejčastěji používané.

- `filter(Predicate<T> predicate)` – vrací stream, kde jsou pouze hodnoty splňující zadanou podmínku,
- `map(Function<T, V> mapper)` – vrací stream, kde na každou hodnotu je aplikovaná zadaná funkce (transformuje jednu hodnotu na druhou),
- `skip(int n)` – vrací stream, kde je přeskočeno prvních `n` hodnot,
- `limit(int n)` – vrací stream, který není delší než `n` hodnot,
- `sorted()` – vrací stream, kde jsou hodnoty uspořádany (vyžaduje, aby třída implementovala rozhraní `Comparable`),
- `distinct()` – vrací stream, kde jsou odstraněny duplicitní hodnoty.

Jednotlivá volání metod nám opět vrátí stream:

```
streamOfWords.filter(word -> word.contains("e"))
streamOfWords.map(word -> word.toUpperCase())
streamOfWords.skip(1)
streamOfWords.limit(10)
streamOfWords.sorted()
streamOfWords.distinct()
```

S takto vráceným streamem můžeme dál pracovat, a to buď zavoláním další transformační metody (resp. metod) nebo například použitím `forEach`, jak ukazují následující příklady.

```
streamOfWords.filter(word -> word.contains("e")).forEach(System.out::println);

streamOfWords.filter(word -> word.contains("e"))
  .map(word -> word.toUpperCase())
  .forEach(System.out::println);

streamOfWords.filter(word -> word.contains("e"))
  .map(word -> word.toUpperCase())
  .sorted()
  .forEach(System.out::println);

streamOfWords.filter(word -> word.contains("e"))
  .map(word -> word.toUpperCase())
  .limit(3)
  .sorted()
  .forEach(System.out::println);
```

Všimněme si, že postupně voláme metody, které nám vrací stream, a vzniká nám takový „vláček" operací (volání metod). Tento vláček by šlo napsat na jeden řádek, ale pro větší srozumitelnost je vhodné psát každou operaci na samostatný řádek. Pokud bychom si vzali poslední příklad, kód by se četl:

1. nejdříve vyber slova, co obsahují písmeno „e",
2. pak převeď všechna písmena na velká,
3. vyber první tři hodnoty,
4. ty setřiď,
5. a vypiš je na standardní výstup.

Na tomto příkladu můžeme vidět, jak jednoduše a přehledně lze zapsat relativně komplikovanou úlohu.

1.3 Koncové operace se streamy

V předchozích příkladech, pokud jsme chtěli zjistit, jak vypadá obsah streamu, použili jsme operaci `forEach` a výpis na standardní výstup. To je však vhodný postup jen pro prvotní seznámení. Pro serióznější práci, potřebujeme mít něco, co nám umožní převést stream do podoby, se kterou bude možné pracovat běžnými, nám známými, prostředky.

Například můžeme získat iterátor s pomocí metody `iterator()`.[1] Stream můžeme převést na seznam pomocí metody `toList()` nebo pole pomocí `toArray`.

[1] Viz předchozí seminář. Stream však nemůžeme použít v konstrukci `for`-each, protože třída `Stream` neimplementuje rozhraní `Iterable`. To by vyžadovalo opakované použití streamu, což není povolené.

```
streamOfWords.toList();
streamOfWords.toArray(String[]::new);
```

V případě ukládání streamu do pole je nutné předat funkci, která nám vytvoří pole dané velikosti. Nabízí se proto použití odkazu na konstruktor pole.

Metody `toList` a `toArray` představují základní operace, která nám převedou stream na typy, se kterými se dá dále pracovat. Avšak můžeme narazit na to, že chceme stream transformovat na nějaký jiný typ nebo do jiné formy. K tomu slouží obecná metoda `collect`, které se předá objekt typu `Collector`, který ze streamu vytvoří výsledný objekt. Řada prakticky použitelných implementací třídy `Collector` je nachystána ve třídě `Collectors`, představme si některé z nich.

- `Collectors.toSet()` – vrací množinu hodnotu,
- `Collectors.joining()` – sloučí hodnoty do jednoho řetězce,
- `Collectors.groupingBy(Function<T, K> classifier)` – rozdělí hodnoty do skupin podle zadaného klíče,
- `Collectors.counting()` – vrací počet hodnot,[2]
- `Collectors.averagingInt(Function<T, Integer> mapper)` – vrací průměrnou hodnotu pro všechny hodnoty ve streamu.

Příklady použití těchto metod by mohly vypadat následovně:

```
Set<String> wordSet = streamOfWords.collect(Collectors.toSet());
String words        = streamOfWords.collect(Collectors.joining(", "));
long count          = streamOfWords.collect(Collectors.counting());
Map<Object, List<String>> wordsByLength = streamOfWords.collect(
  Collectors.groupingBy(word -> word.length()));
Double averageLength = streamOfWords.collect(
  Collectors.averagingInt(word -> word.length()));
```

Dále se můžeme setkat se situacemi, kdy nás zajímá, jestli existuje nějaká hodnota ve streamu splňující zadanou podmínku, nebo zda nějaká podmínka platí pro všechny hodnoty. K tomuto účelu mají streamy metody (i) `anyMatch(Predicate<T>)`, (ii) `allMatch(Predicate<T>)` a (iii) `noneMatch(Predicate<T>)`, které vrátí `true`, pokud (i) nějaká hodnota splňuje zadanou podmínku, (ii) všechny hodnoty splňují zadanou podmínku, (iii) žádná hodnota nesplňuje podmínku.

```
streamOfWords.anyMatch(word -> word.contains("x"));
streamOfWords.allMatch(word -> word.contains("x"));
streamOfWords.noneMatch(word -> word.contains("x"));
```

[2] Lze použít i metodu `Stream.count()`.

V případě, že nás zajímá jen jedna hodnota ze streamu, máme k dispozici metody `findAny()` a `findFirst()`, které vrátí libovolnou hodnotu ze streamu nebo první hodnotu ze streamu.

Zastavme se u návrátové hodnoty těchto metod, kterým je datový typ `Optional<T>`.

1.4 Třída `Optional<T>`

Objekt třídy `Optional<T>` může obsahovat buď jednu hodnotu typu `T`, nebo příznak, že žádnou hodnotu neobsahuje. Smyslem této třídy je bezpečně zpracovat situace, kdy nemáme k dispozici hodnotu, kterou bychom mohli vrátit, což se v případě metod `findFirst` nebo `findAny` může stát. Díky této třídě máme garantované, že obě metody vždy vrátí objekt (třídy `Optional<T>`) a my s tímto objektem můžeme pracovat. Například pomocí metody `isPresent()` jsme schopni zjistit, jestli je nějaká hodnota k dispozici, a pomocí metody `get()` hodnotu získat. Pomocí metody `orElse` může určit hodnotu, která se vrátí, pokud objekt neobsahuje hodnotu, apod.

V minulosti bylo běžnou praxí, že v případě, kdy nebylo možné vrátit hodnotu, vrátil se příznak `null`. Takové chování je sice snadno naprogramovatelné, ale může být zdrojem potenciálních chyb. Pokud si programátor neuvědomí, že návrátová hodnota může být `null`, může se za běhu objevit výjimka `NullPointerException`. Avšak pokud metoda vrací hodnoty typu `Optional<T>`, je programátor explicitně upozorněn na to, že hodnota, kterou si vyžádal nemusí existovat a musí se s tím vypořádat.

Způsobů, jak se s absencí hodnoty vypořádat, je celá řada. Několik z nich nastiňuje následující kód.

```
Optional<String> anyWord = streamOfWords.findAny();
if (anyWord.isPresent()) {
  System.out.println("Nějaké slovo: " + anyWord.get());
}
anyWord.ifPresent(word -> System.out.println("Nějaké slovo: " + word));
System.out.println("Nějaké slovo: " + anyWord.orElse("N/A"));
```

Poznámka 2 *Na hodnotu typu* `Optional<T>` *můžeme nahlížet jako na seznam, který má nejvýše jeden prvek. Proto třída* `Optional<T>` *nabízí metody jako* `map` *nebo* `filter`*, které umožňují bezpečně provádět transformace této hodnoty.*

Poznámka 3 *Hodnotu typu* `Optional<T>` *vrací i metody* `Stream.max` *a* `Stream.min`*, kdy se může stát, že stream je prázdný, a proto není možné vrátit žádnou hodnotu.*

Pokud bychom chtěli vytvořit vlastní hodnotu typu `Optional<T>`, máme k tomu dvě metody `Optional.of(T value)` a `Optional.empty()`, kdy první metoda vrací hodnotu zabalenou do třídy `Optional<T>` a druhá vrací prazdný objekt.

1.5 Streamy primitivních číselných datových typů

Pro práci se streamy, které obsahují čísla máme k dispozici specializované třídy `IntStream`, `LongStream` a `DoubleStream`, které kromě toho, že nepotřebují autoboxing a unboxing, obsahují metody pro pohodlnější

práci s čísly, jako `sum`, `average`, `min`, `max`, jak ukazují následující příklady.

```
IntStream.range(0, 10).sum()
IntStream.range(0, 10).map(x -> x * 2).average()
```

Mezi specializovanými streamy a streamy pracujícími s objekty, je možné přecházet pomocí metod `mapToObj`, `mapToInt` apod. Jak nastiňují následující příklady.

```
streamOfWords.mapToInt(word -> word.length()).average();
IntStream.of(1, 42, 255)
  .mapToObj(n -> Integer.toHexString(n))
  .collect(Collectors.joining(", "));
```

1.6 Paralelní zpracování

Data ve streamu je možné zpracovávat i paralelně. Slouží k tomu metoda `parallel()`, která vrátí stream zpracovávající data ve více vláknech. Použití je opravdu velice jednoduché.

```
streamOfWords.parallel()
  .filter(word -> word.contains("e"))
  .map(word -> word.toUpperCase())
  .toList();
```

Nutno dodat, že přínos paralelního zpracování streamu se projeví až při větších objemech dat, případně pokud jednotlivé operace typu `map` nebo `filter` trvají delší dobu.