

Streamy a lambda výrazy

Tomáš Pitner, Radek Ošlejšek, Marek Šabo

Lambda výrazy

Anonymní metody, umožňují elegantní zápis pomocí symbolu šipky.

- `i → i + 1`
 - inkrementuj číselní hodnotu
- `(Person p) → (p.getAge() >= 18)`
 - vrať `true`, je-li osoba dospělá
- levá část obsahuje vstupní parametry
- pravá část použití a návratovou hodnotu
- dříve realizováno pomocí rozhraní s jednou metodou

Kolekce typu Stream

Proud (`java.util.stream.Stream`) je homogenní lineární struktura prvků (např. seznam).

- hodí se pro **snadné řetězení více operací**
- existují 2 typy operací:
 - *intermediate operations* ("přechodné") — transformuje proud do dalšího proudu
 - *terminal operation* ("terminální", "koncová") — vyprodukuje výsledek nebo má vedlejší účinek
- jedná se o "lazy collections" — prvky se vyhodnotí, až když se zavolá terminální operace a použije se její výsledek



Neplést jsi se vstupně/výstupními proudy (`java.io.InputStream/OutputStream`)

Příklad

```
List<String> list = List.of("MUNI", "VUT", "X");
int count = list.stream()
    .filter(s -> s.length() > 1)
    .mapToInt(s -> s.length())
    .sum();

// count is 7
```

- kolekci transformujeme na proud
- použijeme pouze řetězce větší než 1
- transformujeme řetězec na přirozené číslo (délka řetězce)
- zavoláme terminální operaci, která čísla sečte

Vytvoření Stream

Stream obvykle vytváříme z prvků kolekce, pole, nebo vyjmenováním.

```
Stream<String> stringStream;

List<String> names = new ArrayList<>();
stringStream = names.stream();

String[] names = new String[] { "A", "B" };
stringStream = Stream.of(names);

stringStream = Stream.of("C", "D", "E");
```

Odkazy na metody

Lambda výraz, který pouze volá metodu, se dá zkrátit **vytvořením odkazu na (danou) metodu**.

- Zjištění délky řetězce `s` → `s.length()`:

```
String::length
```

- Vypsání prvků `s` → `System.out.println(s)`:

```
System.out::println
```

Příklad použití Stream

```
List<String> names = ...
names.stream()
    .map(String::toUpperCase)
    .forEach(System.out::println);
```

1. nejdříve vytvoří ze seznamu proud
2. pak každý řetězec převede pomocí `toUpperCase` (průběžná operace)
3. na závěr každý takto převedený řetězec vypíše (terminální operace)

Odpovídá sekvenční iteraci:

```
for(String name : names) {  
    System.out.println(name.toUpperCase());  
}
```

Přechodné metody třídy **Stream I**

Přechodné metody vrací proud, na který aplikují omezení:

- `Stream<T> distinct()`
 - bez duplicit (podle `equals`)
- `Stream<T> limit(long maxSize)`
 - proud obsahuje maximálně `maxSize` prvků
- `Stream<T> skip(long n)`
 - zahodí prvních `n` prvků
- `Stream<T> sorted()`
 - uspořádá podle přirozeného uspořádání

Přechodné metody třídy **Stream II**

- `Stream<T> filter(Predicate<? super T> predicate)`
 - `Predicate` = lambda výraz s jedním parametrem, vrací `boolean`
 - zahodí prvky, které nesplňují `predicate`
- `<R> Stream<R> map(Function<? super T,? extends R> mapper)`
 - `mapper` je funkce, která bere prvky typu `T` a vrací prvky typu `R`
 - může vracet stejný typ, např. `map(String::toUpperCase)`
- existují mapovací funkce `mapToInt`, `mapToLong`, `mapToDouble`
 - vracejí speciální `IntStream`, ..
 - obsahuje další funkce — např. `average()`

Terminální metody třídy **Stream I**

Terminální anebo ukončovací metody.

- `long count()`
 - vrací počet prvků proudu
- `boolean allMatch(Predicate<? super T> predicate)`
 - vrací `true`, když všechny prvky splňují daný predikát
- `boolean anyMatch(Predicate<? super T> predicate)`
 - vrací `true`, když alespoň jeden prvek splňuje daný predikát

- `void forEach(Consumer<? super T> action)`
 - aplikuje `action` na každý prvek proudu
 - např. `forEach(System.out::println)`

Terminální metody třídy `Stream` II

- `<A> A[] toArray(IntFunction<A[]> generator)`
 - vytvoří pole daného typu a naplní jej prvky z proudu
 - `String[] stringArray = streamString.toArray(String[]::new);`
- `<R,A> R collect(Collector<? super T,A,R> collector)`
 - vytvoří kolekci daného typu a naplní jej prvky z proudu
 - `Collectors` je třída obsahující pouze statické metody
 - použití:

```
stream.collect(Collectors.toList());
stream.collect(Collectors.toCollection(TreeSet::new));
```

Příklad

```
int[] numbers = IntStream.range(0, 10) // 0 to 9
    .skip(2) // omit first 2 elements
    .limit(5) // take only first 5
    .map(x -> 2 * x) // double the values
    .toArray(); // make an array [4, 6, 8, 10, 12]

List<String> newList = list.stream()
    .distinct() // unique values
    .sorted() // ascending order
    .collect(Collectors.toList());

Set<String> set = Stream.of("an", "a", "the")
    .filter(s -> s.startsWith("a"))
    .collect(Collectors.toCollection(TreeSet::new));
// [a, an]
```

Možný výsledek `Optional`

- `Optional<T> findFirst()`
 - vrátí první prvek
- `Optional<T> findAny()`
 - vrátí nějaký prvek

- `Optional<T> max/min(Comparator<? super T> comparator)`
 - vrátí maximální/minimální prvek



V jazyce Haskell má `Optional` název `Maybe`.

Použití `Optional`

`Optional<T>` má metody:

- `boolean isPresent()` — `true`, jestli obsahuje hodnotu
- `T get()` — vrátí hodnotu, jestli neexistuje, vyhodí výjimku
- `T orElse(T other)` — jestli hodnota neexistuje, vrátí `other`

```
int result = List.of(1, 2, 3)
    .stream()
    .filter(num -> num > 4)
    .findAny()
    .orElse(0);
// result is 0
```

Paralelní a sekvenční proud

```
List<Integer> integerList = List.of(1, 2, 3, 4, 5);
integerList.parallelStream()
    .forEach(i -> System.out.print(i + " "));
// 3 5 4 2 1
```

```
List<Integer> integerList = List.of(1, 2, 3, 4, 5);
integerList.stream()
    .forEach(i -> System.out.print(i + " "));
// 1 2 3 4 5
```

Konverze na proud — příklad I

Jak konvertovat následující kód na proud?

```
Set<Person> owners = new HashSet<>();
for (Car c : cars) {
    owners.add(c.getOwner());
}
```

Hint `x` → `x.getOwner()` se dá zkrátit na `Car::getOwner`.

```
Set<Person> owners = cars.stream()
    .map(Car::getOwner)
    .collect(Collectors.toSet());
```

Konverze na proud — příklad II

Jak konvertovat následující kód na proud?

```
Set<Person> owners = new HashSet<>();
for (Car c : cars) {
    if (c.hasExpiredTicket()) owners.add(c.getOwner());
}
```

```
Set<Person> owners =
cars.stream()
    .filter(c -> c.hasExpiredTicket())
    .map(Car::getOwner)
    .collect(Collectors.toSet());
```

Funkcionální rozhraní

- Podpora v knihovnách (Java Core API)
- [funkcionální rozhraní](#) v balíku `java.util.functions`
- většinou jako generická (typově parametrizovaná) rozhraní, např.:

Predicate<T>

s jednou metodou `boolean test(T t)`

Supplier<T>

s jednou metodou `void get(T t)`

Consumer<T>

s jednou metodou `void accept(T t)`

Dokumentace

- Benjamin Winterberg: [Java 8 Stream Tutorial](#)
- Amit Phaltankar: [Understanding Java 8 Streams API](#)
- Oracle Java Documentation: [Lambda Expressions](#)