Lambda izrazi

AUDITORNE VJEŽBE

Sadržaj

Primjeri jednostavnih lambda izraza

Najčešće operacije s tokovima

Sortiranje elemenata zbirke pomoću tokova

Primjeri korištenja referenci za metode

Korištenje paralelizma

Usporedba performansi sekvencijalnih i paralelnih tokova

Primjeri jednostavnih lambda izraza

- () -> System.out.println("Hello Lambda")
- x -> x + 10
- (int x, int y) -> { return x + y; }
- (String x, String y) -> x.length() – y.length()
- (String x) -> {
  listA.add(x);
  listB.remove(x);
  return listB.size();
  }

Primjeri jednostavnih lambda izraza

- Na primjer, kod Swing skupa alata za izgradnju grafičkih sučelja često se koristi sljedeći programski isječak koji gumbu dodjeljuje „listener”:

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        System.out.println("button clicked");
    }
});
```

- Korištenjem lambda izraza taj programski isječak izgleda ovako:

```
button.addActionListener(event -> System.out.println("button clicked!"));
```

Primjeri jednostavnih lambda izraza

- Umjesto prosljeđivanja objekta u funkciju, prosljeđuje se blok koda, odnosno funkcija bez imena
- „event” predstavlja naziv parametra i jednak je nazivu parametra koji se prosljeđuje anonimnoj ugniježđenoj klasi
- Znak „->” odvaja parametar od tijela lambda izraza i predstavlja dio programskog koda koji se izvodi kad se pritisne gumb
- Iako se kod lambda izraza ne mora deklarirati tip parametra „event”, kompajler određuje njegov tip korištenjem svog konteksta, odnosno iz „potpisa” metode „addActionListener”
- Tipovi parametara se kod lambda izraza ne moraju navoditi ako je očito o kojim tipovima se radi

Primjeri jednostavnih lambda izraza

- Lambda izrazi se mogu pisati i u sljedećim situacijama:

```
Runnable noArguments = () -> System.out.println("Hello World");

ActionListener oneArgument = event -> System.out.println("button clicked");

Runnable multiStatement = () -> {
    System.out.print("Hello");
    System.out.println(" World");
};

BinaryOperator<Long> add = (x, y) -> x + y;
BinaryOperator<Long> addExplicit = (Long x, Long y) -> x + y;
```

- Sučelje „Runnable” ima samo jednu metodu „public void run()” pa ni lambda izraz ne prima nikakve parametre

Najčešće operacije s tokovima

- **collect(Collectors.*toList*())**
  - Služi za generiranje liste iz vrijednosti unutar „Streama”
  - Spada u „eager” operacije
  - Primjer:

```
List<String> collected = Stream.of("a", "b", "c")
    .collect(Collectors.toList());
```

Najčešće operacije s tokovima

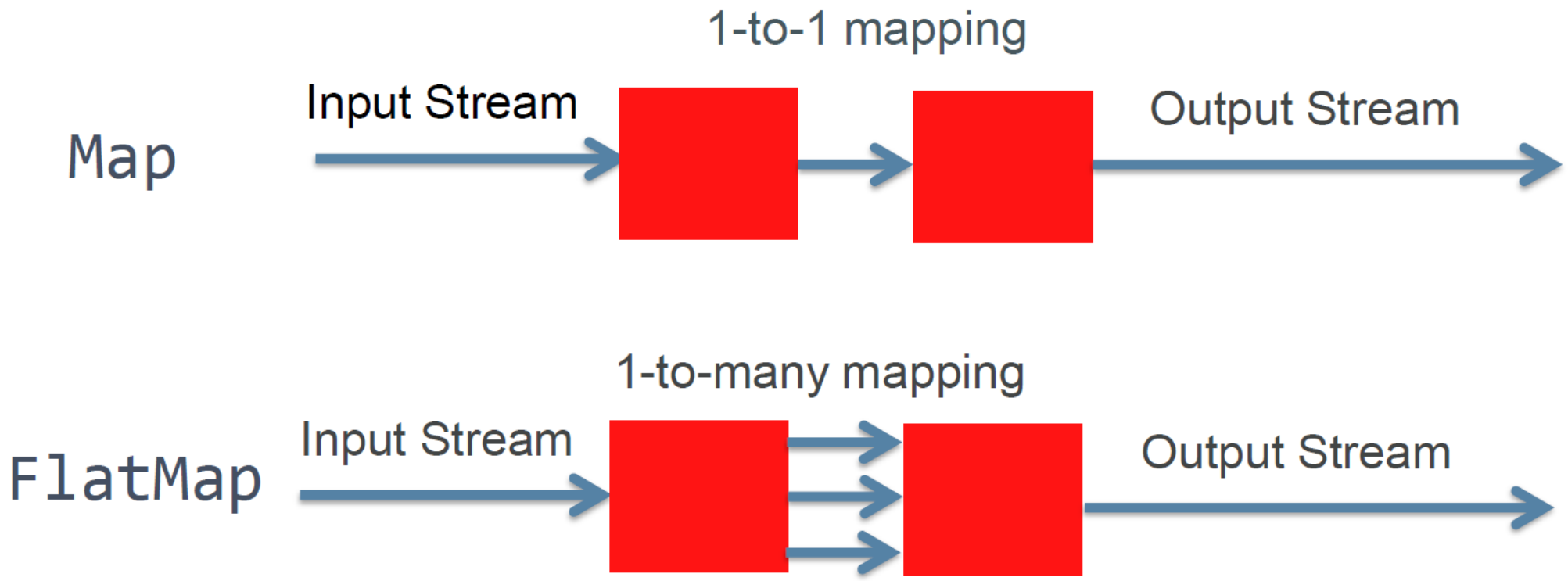
- **map**
  - „Mapira” jednu skupinu vrijednosti u drugu
  - Na primjer, moguće je pretvoriti sve „String” vrijednosti u „uppercase” vrijednosti:

```
List<String> collected = Stream.of("a", "b", "hello")
    .map(string -> string.toUpperCase())
    .collect(Collectors.toList());
```

```
List<String> output = reader
  .lines()
  .flatMap(line -> Stream.of(line.split(REGEXP)))
  .filter(word -> word.length() > 0)
  .collect(Collectors.toList());
```

Najčešće operacije s tokovima

- Može se koristiti za 1:1 ili 1:N mapiranje:



Najčešće operacije s tokovima

- **filter**
  - Služi za filtriranje vrijednosti prema zadanim kriterijima
  - Primjer:

```
List<String> beginningWithNumbers
    = Stream.of("a", "1abc", "abc1")
        .filter(value -> isDigit(value.charAt(0)))
        .collect(Collectors.toList());
```

Najčešće operacije s tokovima

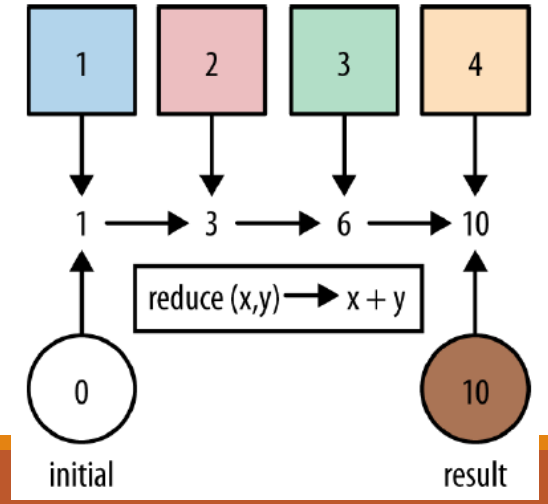
- **min i max**
  - Pronalaze najveću i najmanju vrijednost
  - Primjer:

```
Track shortestTrack = tracks.stream()
    .min(Comparator.comparing(track -> track.getLength()))
    .get();
```

Najčešće operacije s tokovima

- **reduce**
  - Koristi se kad je iz zbirke vrijednosti potrebno odrediti jednu vrijednost
  - Na primjer, sumu elemenata je moguće izračunati i na sljedeći način:

```
int count = Stream.of(1, 2, 3)
    .reduce(0, (acc, element) -> acc + element);
```



Najčešće operacije s tokovima

- **flatMap**
  - Koristi se u slučaju potrebe za zamjenom elemenata u toku i njihovo konkateniranje u zajedničku zbirku
  - Na primjer, sumu elemenata je moguće obaviti i na sljedeći način:

```
List<Integer> together = Stream.of(asList(1, 2), asList(3, 4))
    .flatMap(numbers -> numbers.stream())
    .collect(Collectors.toList());
assertEquals(asList(1, 2, 3, 4), together);
```

Ulančavanje operacija kod tokova

- Operacije nad elementima zbirki mogu se ulančavati pri čemu je prvo potrebno obaviti „lazy” operacije, a na kraju izvršiti jednu „eager” operaciju:

```
Set<String> origins = album.getMusicians()
    .filter(artist -> artist.getName().startsWith("The"))
    .map(artist -> artist.getNationality())
    .collect(toSet());
```

Sortiranje elemenata zbirke pomoću tokova

- Sortiranje elemenata se pomoću tokova može obaviti na sljedeći način:

```
Set<Integer> numbers = new HashSet<>(asList(4, 3, 2, 1));
List<Integer> sameOrder = numbers.stream()
    .sorted()
    .collect(Collectors.toList());
assertEquals(asList(1, 2, 3, 4), sameOrder);
```

Primjeri korištenja referenci za metode

```
List<String> myList = ...
myList.forEach(s -> System.out.println(s));
```

```
myList.forEach(System.out::println);
```

```
List<String> myList = ...
myList.replaceAll(s -> s.toUpperCase());
```

```
myList.replaceAll(String::toUpperCase);
```

Korištenje paralelizma

- Kako bi se maksimalno iskoristila snaga višejezgrenog procesora kod obrađivanja podataka unutar zbirki, moguće je koristiti „paralelne tokove”
- Razlika između „serijskih” i „paralelnih” tokova očituje se samo u pozivu metode „parallelStream” umjesto „stream”:

```
public int serialArraySum() {
    return albums.stream()
        .flatMap(Album::getTracks)
        .mapToInt(Track::getLength)
        .sum();
}
```

```
public int parallelArraySum() {
return albums.parallelStream()
    .flatMap(Album::getTracks)
    .mapToInt(Track::getLength)
    .sum();
}
```

Usporedba performansi sekvencijalnih i paralelnih tokova

- Korištenjem različitih vrsta tokova moguće je dobiti znatna ubrzanja u izvođenju operacija nad velikim količinama podataka unutar zbirki
- Na primjer, ako je zadano polje „long” vrijednosti od 10.000.000 elemenata i po njemu se pretražuju najveća, najmanja i srednja vrijednost, potrebno je proći po svim elementima kako bi se dobili rezultati
- Kod takvih operacija moguće je koristiti odvojeno izvođenje operacija, sekvencijalne i paralelne tokove

Usporedba performansi sekvencijalnih i paralelnih tokova

```
public class StreamStatisticsComparison
{
   public static void main(String[] args)
   {
      SecureRandom random = new SecureRandom();

      // create array of random long values
      long[] values = random.longs(10_000_000, 1, 1001).toArray();

      // perform calculcations separately
      Instant separateStart = Instant.now();
      long count = Arrays.stream(values).count();
      long sum = Arrays.stream(values).sum();
      long min = Arrays.stream(values).min().getAsLong();
      long max = Arrays.stream(values).max().getAsLong();
      double average = Arrays.stream(values).average().getAsDouble();
      Instant separateEnd = Instant.now();
```

Usporedba performansi sekvencijalnih i paralelnih tokova

```
// display results
System.out.println("Calculations performed separately");
System.out.printf("    count: %,d%n", count);
System.out.printf("      sum: %,d%n", sum);
System.out.printf("      min: %,d%n", min);
System.out.printf("      max: %,d%n", max);
System.out.printf("  average: %f%n", average);
System.out.printf("Total time in milliseconds: %d%n%n",
   Duration.between(separateStart, separateEnd).toMillis());

// time sum operation with sequential stream
LongStream stream1 = Arrays.stream(values);
System.out.println("Calculating statistics on sequential stream");
Instant sequentialStart = Instant.now();
LongSummaryStatistics results1 = stream1.summaryStatistics();
Instant sequentialEnd = Instant.now();
```

Usporedba performansi sekvencijalnih i paralelnih tokova

```
   // display results
   displayStatistics(results1);
   System.out.printf("Total time in milliseconds: %d%n%n",
      Duration.between(sequentialStart, sequentialEnd).toMillis());

   // time sum operation with parallel stream
   LongStream stream2 = Arrays.stream(values).parallel();
   System.out.println("Calculating statistics on parallel stream");
   Instant parallelStart = Instant.now();
   LongSummaryStatistics results2 = stream2.summaryStatistics();
   Instant parallelEnd = Instant.now();

   // display results
   displayStatistics(results1);
   System.out.printf("Total time in milliseconds: %d%n%n",
      Duration.between(parallelStart, parallelEnd).toMillis());
}
```

Usporedba performansi sekvencijalnih i paralelnih tokova

```
// display's LongSummaryStatistics values
   private static void displayStatistics(LongSummaryStatistics stats)
   {
      System.out.println("Statistics");
      System.out.printf("    count: %,d%n", stats.getCount());
      System.out.printf("      sum: %,d%n", stats.getSum());
      System.out.printf("      min: %,d%n", stats.getMin());
      System.out.printf("      max: %,d%n", stats.getMax());
      System.out.printf("  average: %f%n", stats.getAverage());
   }
} // end class StreamStatisticsComparison
```

Usporedba performansi sekvencijalnih i paralelnih tokova

```
Calculations performed separately
    count: 10,000,000
      sum: 5,006,045,362
      min: 1
      max: 1,000
  average: 500.604536
Total time in milliseconds: 188
```

```
Calculating statistics on parallel stream
Statistics
    count: 10,000,000
      sum: 5,006,045,362
      min: 1
      max: 1,000
  average: 500.604536
Total time in milliseconds: 47
```

```
Calculating statistics on sequential
stream
Statistics
    count: 10,000,000
      sum: 5,006,045,362
      min: 1
      max: 1,000
  average: 500.604536
Total time in milliseconds: 62
```

Pitanja?