Introduktion Til Programmering Disposition 2
Functional Processing of Collections (Streams and Lambdas)
Kapitel 5

Joakim Iversen

December 19, 2024

Disposition:

1. Motivation
2. Streams
3. Lambdas

Noter:

KODEEKSEMPEL: ANIMAL POPULATION

Motivation

Der er de seneste par år blevet

Kapitel 5

Main Concepts

- Funktional Programmering
- lambdas
- Streams
- Pipelines

Hvor vi førhen brugte vores **for-each** og **While** løkke til at gennemløbe vores collections, er vi blevet introduceret til en "pænere" måde at gøre dette på ved brug af streams og lambdaer.

Lambdaer

Hoved ideen bag lambdaer er, hvor vi til en metode i stedet for at give den parametre nu kan give den en *Kode stump*. Så hvor vi før kun har set parametre som forskellige data typer, *int, strings, objekter*, kan vi nu bruge kode.

DETTE ER EN NY TILFØJELSE TIL JAVA, DER KOM I JAVA 8

Vores basis syntax for at gennemløbe en collection med lambdaer er lidt anderledes end da vi brugte løkker:

```
collection.doThisForEachElement(some code);
```

Dette er meget mere simpelt end at skulle lave en stor løkke med en conditions der skal mødes og så hvad der skal ske. Dette er nu skrevet in i vores *doThisForEachElement*. Vi kan sammenligne de to metoder med at skulle klippe en hel klasse. I den gamle metode - med løkker - beder vi læreren om at give os det første barn, hvilket vi så give en hårklipning. Derefter beder vi læreren om at sende os et nyt barn.

Når vi burger lambdaer skriver vi instrukserne ned for hvordan man klipper hår, giver det til læreren og beder dem om at klippe alle børnene. Vi er i bund og grund også ligeglad med hvordan læreren får det gjort, om de selv gør det eller giver opgaven videre til flere andre så flere børn kan blive klippet på samme tid.

På denne måde er vores liv blevet meget nemmere, da læreren gør det meste af arbejdet for os.

Der er nogle ting vi skal notere os omkring lambdaer der gør dem unike fra en normal funciton:

- Der er ikke noget public eller private keyword. (public af natur)
- Ingen returtype - Det kan compileren regne ud fra det endelige statement
- Der er ikke noget navn til en la,bda: starter med parameter listen
- en pil (->) deler paramter listen fra kroppen.

Basisk Lambda Syntax

```
(Sighting record) ->
{
    System.out.println(record.getDetails());
}
```

Lambdaer blive ofte brugt til små oneoff opgaver der ikke kræver kompleksiteten af en hel klasse.

Vi vil nu kigge på en mulighed for hvad vi kan indsætte ind i *doThisForEachElement* metoden vi nævnte. Det er nemlig at lave en for-each løkke med en lambda. Syntaxen for dette vil se sådan her ud:

For Each Løkke - Lambda Syntax

```
sightings.forEach(
    (Sighting record) ->
    {
        System.out.println(record.getDetails());
    }
);
```

Der er altså en *forEach* metode til lambdaer der gennemløber alle vores sightings og så gør det følgende ved dem. Her er det vigtigt at parameteren til *forEach* metoden skal være en lambda.

Selvom dette måske ikke umildbart ser "pænere" ud end en normal lykke er der nogle ting vi kan gøre, for at simplificere det endnu mere.

- Parameter typen (Denne er altid den samme som collection element)
- Hvis parameterlisten kun indeholder en parameter kan vi undlade '()'
- Hvis kroppen kun har en statement kan vi droppe ''

Vi vil altså kunne lave vores kode fra før om til

For Each Løkke - Lambda Syntax - Simplificeret

```
sightings.forEach(record -> System.out.println(record.getDetails()));
```

Og det er denne kode vi skal gå efter i vores kode.

Streams

forEach metoden vi lige har kigget på til ArrayList er et eksempel på en måde at gøre brug af streams. Streams minder meget om collections med med nogle meget vigtige forskelle:

- Stream elementer er ikke tilgået ved index, men normalt sekventialt
- Indhold og ordenen i streams kan ikke ændres - ændringer kræver at man laver en ny stream
- en stream kan muligvis blive uendelig

Streams er brugt til at ensgøre prosseser af forskellige dataset. Dette kunne være fra en collection, modtagene beskeder over et netværk, linjer af tekst fra en text fil, eller alle karaktere fra en String.

Streams tilføjer en anden stor mulighed til os hvilket er muligheden for sikker paralel processing. Dette kan være meget nyttigt når rækkefølgen hvor i elementer bliver behandlet er lige gyldig, eller prosesen med data afhænger drastigt af andre elementer.

Filters, maps, reductions

Ved brug af disse tre funktioner kan vi modifisere vores stream til næsten alle behov vi kunne have.

Filters: Filter funktionen tager en stream og filtrerer nogle af elementerne fra og returnere en ny stream uden disse elementer.

Maps: Map funktionen tager en stream og *"mapper"* elementerne fra den gamle stream til nye elementer i den nye stream. Den nye stream vil have det samme antal elementer, men typen og konteksten af hvert element kan være anderledes.

Reductions: Reduce funktionen tager alle elementer i en stream of reducere dem ned til et enkelt element. Dette kan være på forskellige præmiser. Det kan være ved at samle alle objekter sammen til et, vælge det mindste element eller noget helt andet. Så her starter vi altså med en helt stream, og ender op med et enkelt resultat.

Pipelines: Den fulde kraft af streams og streams funktioner bliver tilgægelige når man sætter flere sammen. Dettte kaldes også pipeline. Hvis vi fx ville finde hvor mange elefant sightings vi har, kan vi bruge et filter, map og reduce efter hinanden.

Først filtrere vi efter elefanter, dernæst mapper vi til antallet af elefanter vi har set og til sidst reducere vi ned til det samlede antal elefanter det er blevet set.