Collections (ArrayList, Maps, Sets and Iterators)

Kapitel 4 + 6

Joakim Iversen

December 18, 2024

Disposition:

1. Motivation
2. Collection
   (a) ArrayList
   (b) Maps
   (c) Set
3. Iterators

Detail Disposition:

1. ArrayList
2. Maps
3. Sets
4. Iterators
   (a) For-Each Loop
   (b) While Loop

Noter:

KODE EKSEMPEL: LIDT AF HVERT

Motivation:

Motivationen ved brug af ArrayList, Maps og Sets er muligheden for at opbevare en vilkårlig mængde objekter samlet, og kunne udfører gennemsøgninger på hele vores samling. Derudover har hver af disse collections deres egne egenskaber, hvilket gør det nemmere for os at finde og arbejde på vores objekter.

Kapitel 4

Main Concepts

1. Collections
2. Iterators
3. Loops

Collection abstraction heviser til et objection objekt. Disse er objekter der kan gemme et vilkårligt antal af andre elementer. Eksempler på dette kunne være *ArrayList*.

Det er vigtigt at notere her at bibliotek klasserne, som ArrayList, ikke er skrevet til et specifikt formål hvilket gør dem utrolig magtfulde da vi på denne måde selv kan implementere dem på den måde vi gerne vil.

Eksempel på brug af ArrayList:

```
import java.util.ArrayList;

public class MusicOrganizer {
    private ArrayList<String> files;

    public MusicOrganizer() {
        files = new ArrayList<>();
    }
}
```

Først linje viser hvordan vi får adgang til en bibliotek klasse i java ved brug af *import statement*. Vi gør **ArrayList** klassen fra **java.util** pakken, tilgængelig til den klassen vi improtere den i. Når vi først har importeret en klassen på denne måde, kan vi tilgå den, som om det var en af vores egne klasser.

Så støder vi på det næste, **<String>**. Dette er nødvendgit da *ArrayList* er en *general-purpose* klasse. Det vil sige at den ikke er begrænset til, hvad den kan holde. Men vi kan kun holde en type i det, og vi har brug for at oplyse om hvilken type det er vi egentlig gemmer i den. Så her oplyser vi om, at vi gemmer elementer af typen **String**.

Vi har altså derfor burg for at deklarer to typer når vi bruger collections:

1. Typen af collections (her ArrayList)
2. Typen af elementer vi vi gemme

Vi kalder klasser der kræver denne ekstra parameter for **generic classes**.

De er nul indekseret og hvis der fjernes et element rykke alle elementer "over" det fjerenede element ned. ergo en ArrayListe med 10 elementer har sidste indeks 9, og fjerner du indeks 5 bliver 6 til 5.

Processing collections

For at kunne gennemgå hele vores collection samling, bliver vi nød til at have nogle modificerbare kodestumper der automatisk ændre sig til at mængden der gemmer sig i vores collection. Dette kan vi gøre med fx. loops.

For-Each loop:

```
for(ElementType element : collection) {
    loop body
}
```

Dette er edn basale form a vores for-each løkke. Den tager en *ElementType* der er den type elementer vi har gemt i vores collection. Dernæst tager den en *element* hvilket bare er et variablenavn for det element vi er kommet til. Det sidste er den *collection* som vi vil gennemløbe.

Så i kontekst af MuicOrganizer hvor vi vil printe alle sang titler vi har i vores collection vil vi gøre sådan her:

```
public void listAllFiles() {
    for(String filename : files) {
        System.out.println(filename);
    }
}
```

Denne siger forhver filename = String i vores files collection, print dette filnavn. Ved brug af for-each kan vi også gøre flere ting da det muliggøre vi også kan søge igennem vores collection efter elementer med specifikke krav. Vi kunne fx lave en løkke der søger efter en specifik title ved at tilføje nogle få linjer til den tidligere kode

```
public void listAllFiles(String searchString) {
    for(String filename : files) {
        if(filename.contains(searchString)) {
            System.out.println(filename);
        }
    }
}
```

While loop:

```
while(boolean condition) {
    loop body
}
```

Antallet af gange et while loop gennemløber kan skiftes og er i styret af den bolske condition man giver dem. På denne måde kan man styre at løkken gennemløbes indtil et bestemt krav bliver mødt. Vi genavaluere vores bolske udtryk hver gang vi har været igennem løkken, og så snart det er mødt (vi får *true*) vil vi ikke kører gennem løkken mere.

Man skal være opmærksom på, at det er muligt og lave en løkke der aldrig vil ende. Dette vil give os en fejl i vores program. Det er derfor vigtigt at være opmærksom på at de while-løkker man laver, har en ende. vi vil kunne kigge igennem vores collection på denne måde:

```
int index = 0;
while(index < file.size()) {
    String filename = files.get(index);
    System.out.println(filename);
    index++
}
```

Dette vil kigge igennem alle vores elementer i vores collection, og sikre sig at vi kommer ud af løkken så længe collection ikke har en uendelig størrelse. På denne måde vil vi også, ligesom ved for-each, gennemsøge vores collection efter et bestemt element.

Iterator type

Dette er en tredje metode til at gennemgå en collection, og ligger på en måde i mellem *for-each* og *while* løkkerne. Den gør brug af en while løkke at udfører iterationen og et *iterator objekt* istedet for et tal til at olde tryr på indekset i listen.

!OBS! Der er forskel på **Iterator** med stort 'I', som er en Java klasse, og **iterator** med lille 'i', som er et metode kald.

Vi vil kunne bruge en iterator på vores ArrayList på denne måde

```
import java.util.ArrayList;
import java.util.Iterator;

Iterator<elementType> it = meCollection.iterator();
while(it.hasNext()) {
    call it.next() to get the next element
    do something with that element
}
```

En Iterator har kun fire metoder, hvor to bliver brugt til at iterare over en collection:

1. hasNext
2. next

Ingem af disse tager en parameter, men begge har en ikke-void returtype, så de bliver brugt som et udtryk.

I eksemplet bruger vi først *iterator* metoden til at få vores **Iterator** objekt - tag note af at dette også er af en generisk type, så vi skal give den en elementType.

Det smarte ved at bruge vores iterator object er at vi ikke skal forholde os til et indeks nummer, men iteratoren selv kan tjekke om der er flere objekter.

For at forstå en iterator er det vigtigt at forstå metoden **next**. Den returnere det næste element og rækker derefter "forbi" dette element, på den måde vil det næste kald til **next** metoden give os det efterfølgende element.

En af de ting der er sværer at gøre ved en collection ved brug af løkker er at slette da dette rykker på vores indeks nummerering. Her er det en fordel af bruge vores iterator object. Både en ArrayListe og Iterator har en remove method, men når vi skal fjerne under gennemgang, er det en fordel at bruge iteratoren. Hvor at ArrayListen "mister" styring over hvilket objekt den er kommet til, huske Iteratr objekter nemlig ordenen iforhold til vores collection.

Vi skal dog bruge et while loop under dette, da vi i en for-each ikke har en iterator til at arbejde sammen med den.

```
Iterator<Track> it = tracks.iterator();
while(it.hasNext()) {
    Track t = it.next();
    String artist = t.getArtist();
    if(artist.equals(artistToRemove)) {
        it-remove();
    }
}
```

Kapitel 6

Main Concepts

1. using library classes
2. reading documentation
3. writing documentation

Fra kapitel 6 omhandler sprøgsmålet kun Maps og Sets. Det er derfor dette vi vil have fokus på i denne tekst - OBS, der vil være noter til resten af kapitelet i bunden, da dette også vil være tilgægeligt for spørgsmål.

Maps:

Maps er en collection af nøgler/værdier (Key/values) par. Dette er ligesom en ArrayList er størrelsen dynamisk, men i modsætning er det ikke en samling af et objekt. Men et samling af to objekter, et *nøgle-objekt* og et *værdi-objekt*.

Dette giver os mulighed for ikke at slå op i vores map med et indeks, men med en specifik nøgle.

Eksempel på dette kunne være en kontaktliste i en telefon. Her slåer du et navn op for at få et telefon nummer. Vi bruger ikke et indeks - placeringen af kontakten i listen - til at finde nummeret.

Det kan gøres virkelig nemt at slå noget op i et map, i kontekst af en kontaktliste, kan det gøres ved at alfabetisk sortere navne. En omvendt søgning er muligt, hvor du vil finde en person ved at søge efter et nummer, men det er vildt svært og tage lang tid. Det er derfor ikke ideelt.

EKSEMPEL PÅ ET MAP KUNNE VÆRE HASHMAP

Sets:

I standard biblioteket til java er der allerede implementerede forskellige **set**.

De forskellige collections i java er meget ens. Så når du forstår en enkelt af dem, og hvordan du bruger dem, forstå du langt en af vejen også de andre. En liste holder foreksempel den orden som de bliver langt ind i listen. Et set derimod, holde rikke styr på ordenen, men det sikre sig vi kun har en enkelt af hvert objekt. Så hvis vi har et set af tal, kan vi ikke have flere 1-taller.

EKSEMPEL PÅ ET SET KUNNE VÆRE HASHSET