Funksjonell programmering med Kotlin

Lars Tveito

Institutt for informatikk, Universitetet i Oslo
larstvei@ifi.uio.no

26. januar, 2024

Plan for timen

- Før pause skal vi snakke litt om:
  - hva funksjonell programmering er
  - anonyme funksjoner
  - høyereordens funksjoner
  - avveininger ved å programmere i en mer funksjonell stil
- Etter pause:
  - Liveprogrammering med *collections* i Kotlin

Programmeringsspråk

- Et programmeringsspråk brukes til å uttrykke hva en datamaskin skal gjøre
- Like fullt brukes de for å kommunisere idéer mellom mennesker
- Programmeringsspråk kategoriseres ofte som lavnivå, eller høynivå
  - Det referer til lavt eller høyt nivå av abstraksjon

Lavnivå programmeringsspråk

- Lavnivå språk ligger nærme språket datamaskinen tolker
- Gir svært god kontroll på nøyaktig hva datamaskinen skal gjøre
- Dette gjør programmene (potensielt) veldig raske!
- Programmene blir også fort større, mer kompliserte og vanskelige å teste
- Avstanden mellom språket du tenker i og språket du skriver i blir større

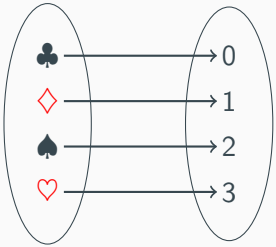
Høynivå programmeringsspråk

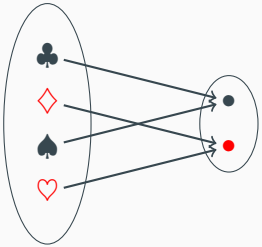
- I et høynivå programmeringsspråk jobber man på data som

Tall	Strenger	Lister	Mengder	Trær	Grafer
42	*"Hei!"*	$[1, 1, 2, 3]$	$\{0, 2, 4, 6\}$		

- Der representasjonen ligger nærme det som representeres
  - hvor den underliggende representasjonen *abstrahert* vekk
- I stedet for registere og minneadresser jobber man med variabler
- I stedet for å hoppe mellom navngitte kodelinjer jobber man med
  - Prosedyrer
  - Betingelser
  - Løkker

Funksjoner





- En *funksjon* returner samme output for et gitt input *hver gang*
- En funksjon kalles *utelukkende* for sin returverdi
- En funksjon har ingen sideeffekter
  - Den gjør ikke noe *mer* enn å returnere en verdi
- Fra et høyt nok abstraksjonsnivå er en funksjon og en ordbok (map) det samme

Funksjonell programmering

- I funksjonell programmering er funksjoner førsteklasses objekter
  - De kan sendes som argumenter til funksjoner
  - De kan returneres av funksjoner
  - *Funksjoner er verdier*
- Høyereordens funksjoner er funksjoner som
  - tar funksjoner som argument, eller
  - gir en funksjon som returverdi
- I *ren* funksjonell programmering gjøres *all* beregning med funksjoner
- I funksjonelle språk kan funksjoner uttrykkes uten å navngis
  - De kalles anonyme funksjoner eller *lambda* (eller λ)

Funksjonell programmering i Kotlin

- I Kotlin er funksjoner førsteklasses objekter
- Det lar deg programmere i en *funksjonell stil*
  - Men språket tvinger deg ikke til det
- Mange av idiomene er funksjonelle
  - (`https://kotlinlang.org/docs/idioms.html`)
- Det oppfordres til bruk av `val` over `var` (ikke-muterbare pekere)
- Det oppfordres til bruk av ikke-muterbare datastrukturer
- Standardbiblioteket for samlinger (Collections) er i stor grad funksjonell

Ikke-muterbar data

- En *funksjon* vil aldri mutere data, fordi å endre tilstand er en sideeffekt
- Dersom du skriver funksjonell kode vil det aldri ha noen hensikt å behandle muterbar data
- Fordelene med ikke-muterbare datastrukturer går hånd i hånd med fordelene med funksjonell programmering
- I Kotlin er datastrukturene `List`, `Set` og `Map` ikke-muterbare
  - men kommer i muterbare varianter `MutableList`, `MutableSet` og `MutableMap`

Eksempel: Fakultetsfunksjonen

```
fun factorial(n: Int): Int {
    var i = n
    var res = 1

    while (i > 0) {
        res *= i
        i--
    }

    return res
}
```

Vi starter med to muterbare `var` og en løkke

Eksempel: Fakultetsfunksjonen

```
fun factorial(n: Int): Int {
    var res = 1

    for (i in 1..n) {
        res *= i
    }

    return res
}
```

Vi kan bruke `i in 1..n` for å kvitte oss med én muterbar `var`

Eksempel: Fakultetsfunksjonen

```
fun factorial(n: Int): Int {
    if (n == 0) {
        return 1
    }
    return n * factorial(n - 1)
}
```

Vi kan bruke rekursjon for å kvitte oss med den siste muterbare `var`-en

Eksempel: Fakultetsfunksjonen

```
fun factorial(n: Int): Int =
    when (n) {
        0 -> 1
        else -> n * factorial(n - 1)
    }
```

Vi kan bruke when og få funksjonskroppen som ett enkelt uttrykk

Eksempel: Fakultetsfunksjonen

```
fun factorial(n: Int) = (1..n).fold(1) { acc, x -> acc * x }
```

Vi kan bruke en *høyereordens funksjon* `fold` med en λ og nesten bli kvitt all koden

Eksempel: Fakultetsfunksjonen

```
fun factorial(n: Int) = (1..n).fold(1, Int::times)
```

Til slutt kan vi bruke en navngitt funksjon `Int::times` med `fold`

Hva funksjonell programmering forenkler

- Funksjonell kode er enkel å teste
  - Alt funksjonen bruker er gitt som input
  - En funksjon gjør kun én ting (returnerer en verdi)
  - Dersom funksjonen er vanskelig å teste burde den kanskje deles opp i mindre funksjoner
- Funksjonell kode er enklere å resonnere om
  - Du kan bytte ut et sammensatt uttrykk med resultatet av uttrykket uten å endre programmet
- Det er enklere å debugge
  - Feilen vil være i kallkjeden som resulterte i feilen
- Samtidige (eng: concurrent) programmer er vesentlig enklere å få riktig
  - Ikke-muterbar data er trådsikker

Anonyme funksjoner i Kotlin

- Kotlin har en enkel syntaks for å uttrykke anonyme funksjoner
  - En funksjon som tar et argument og ganger det med seg selv:

    ```
    { x -> x * x }
    ```

  - En funksjon som tar to argumenter og summerer dem:

    ```
    { x, y -> x + y }
    ```

  - I en funksjon av ett argument kan argumentet refereres til som `it`

    ```
    { it * it }
    ```

- Hvis en funksjon forekommer som siste argument til en annen funksjon kan den skrives utenfor parentesen

```
lst.map({ it * it }) == lst.map { it * it }
```

Collections i Kotlin

- Kotlin har et rikt bibliotek av funksjoner på samlinger.
- Her er noen av dem:
  - `first` / `last`
  - `forEach`
  - `map`
  - `fold`
  - `filter`
  - `sortedBy`
  - `find` / `findLast`
  - `count`
  - `partition`
  - `zip` / `zipWithNext`
  - `...`

first / last

Henter ut første/siste element

```
val list = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9)

list.first()
// => 1

list.last()
// => 9
```

forEach

Utfører en funksjon på alle elementer

```
val list = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9)

list.forEach { println(it) }
// 1
// 2
// 3
// 4
// 5
// 6
// 7
// 8
// 9
```

map

Transformerer alle elementene med en gitt funksjon

```
val list = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9)

list.map { it * 2 }
// => [2, 4, 6, 8, 10, 12, 14, 16, 18]
```

filter

Beholder elementene som oppfyller et predikat

```
val list = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9)

list.filter { it % 2 == 0 }
// => [2, 4, 6, 8]
```

Et predikat er en funksjon som returnerer sant eller usant

fold

Tar en initiell verdi, og en funksjon som *kombinerer* elementene

```
val list = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9)

list.fold(0) { acc, x -> acc + x }
// => 45
```

sortedBy

Sorter basert på et kriterium, gitt som en funksjon

```
val list = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9)

list.sortedBy { it % 3 }
// => [3, 6, 9, 1, 4, 7, 2, 5, 8]
```

`find` / `findLast`

Finn det første/siste som oppfyller et predikat

```
val list = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9)

list.find { it % 2 == 0 }
// => 2

list.findLast { it % 2 == 0 }
// => 8
```

Kan ses på som en sammensetting av `filter` og `first` / `last`

count

Teller elementer som oppfyller et predikat

```
val list = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9)

list.count { it % 2 == 0 }
// => 4
```

Kan ses på som en sammensetting av `filter` og `size`

partition

Separerer elementene som oppfyller predikatet fra de som ikke gjør det

```
val list = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9)

list.partition { it % 2 == 0 }
// => ([2, 4, 6, 8], [1, 3, 5, 7, 9])
```

`zip` / `zipWithNext`

`zip` parrer elementer fra to lister, og `zipWithNext` parrer hvert element me det etterfølgende

```
val list = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9)

list.zipWithNext()
// => [(1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 7), (7, 8), (8, 9)]

list.zip(list.map { it * 2 })
// => [(1, 2), (2, 4), (3, 6), (4, 8), (5, 10), (6, 12), (7, 14), (8,
↪  16), (9, 18)]
```