IN1010

Repetisjon Uke 16
Invarianter

(Basert på Lars Tveitos presentasjon)

16. mai, 2025

I dag

- Tilstander
- Betingelser
- Invarianter
- Debugging
- Dokumentasjon

Tilstand

Helt enkelt så er en tilstand et øyeblikksbilde. Men hvis vi har bare en tilstand, så skjer det ingenting. Det beskriver noe statisk og konstant. Altså noe som er uforanderlig (immutable).

Når vi programmerer så er vi ute etter å skape flere tilstander, og vi trenger også å vite hvordan vi går over til en ny tilstand.

Tilstand



En dør har normalt to tilstander, enten lukket eller åpen.

De følger hverandre, og vi kan si at døren har et "minne", den "vet" om den er åpen eller lukket. Og hver tilstand følger den forrige.

En tilstandsovergang, er hvordan vi endrer døren fra åpen til lukket, og motsatt.

Tilstand

Den vanligste måten vi tenker og skriver programmene kalles for imperativ programmering.

Programmet består av en sekvens av uttrykk. Hver "linje" i programmet vi skriver er en ny tilstand, og de beskriver en tilstandsovergang.

På engelsk kalles en program-"linje" for en *statement*, som kan oversettes som utsagn eller setning. Men en tilstand kalles for *state* på engelsk.

Tilstand

I Java så er hver program-setning enklest forklart som alt fram til semikolon.

Vi bruker en programteller (PC/program counter) for å angi hvor i programmet vi er, og hver program-setning får tildelt sitt nummer i sekvens.

(Mer teknisk korrekt, så er det i Java hver bytekode-instruksjon som telles)

Tilstand

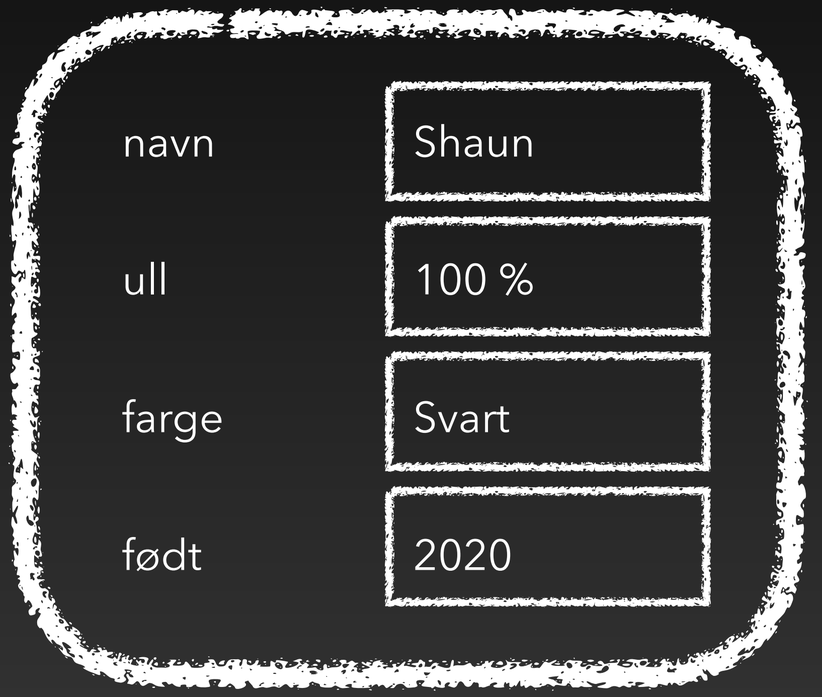
Hver tilstand i programmet beskriver alt som er relevant for hele programmet. Det kan fort bli mye.

En måte å begrense slik at vi kan gjøre det mer håndterbart, er å bruke objekter.

Tilstand i objekter

For å beskrive tilstanden til et objekt, så bruker vi instansvariablene.

Datastruktur-tegninger er gode måter for å forklare et objekts tilstand.



Sau-objekt

Feil i tilstandene

Hvis vi skriver kode som har feil syntaks, så vil Java-kompilatoren si fra med en gang.

Men logiske feil er det ikke like lett å finne ut av. Det kan føre til en kjøretidsfeil, som kan gi oss en indikasjon.

Som oftest får vi et resultat som ikke er det vi forventer.

Feil i tilstandene

Så hva gjør vi nå?

Vi må finne noen betingelser og faste holdepunkter i programmet vårt.

Vi må vite hvor tingenes tilstand er bra, og hvor det ikke er det.

Kontroll og oversikt

Ved hver tilstand gjennom programmet vårt, så må vi definere noen betingelser som må gjelde.

Det er variabler, egenskaper eller forhold som vi kan finne ut om kan være sant eller ikke.

Kontroll og oversikt

La oss ta døren fra eksempelet over

- En betingelse for at en dør er lukket, er at den går i ett med veggen.
- Hvis vi kan se hva som skjer på utsiden, så kan vi si at døren er åpen (døren har ikke vindu).

Invarianten

Nå så kan vi slå sammen de to betingelsene til en invariant.

Det skal ikke være mulig for at døren går i ett med veggen, og at vi samtidig kan se hva som skjer på baksiden.

Invarianter

Invarianter er påstander eller betingelser som alltid skal være sanne. Som oftest er det ganske åpenbare tilfeller.

Det betyr at hvis en invariant brytes, så har vi mest sannsynlig en feil.

Så vi kan ved ulike punkter i programmet vårt definere slike tilfeller, og på den måten få en ide om hvor feilen ligger.

Invarianter

Andre eksempler på invarianter

- Antall sauer på gården må være et positivt heltall
- Sidetall i en bok vil være i rekkefølge
- Summen av vinkler i en trekant er 180°
- På en rett vei, uten avkjørsler, vil antall biler inn være lik som ut
- Et barns foreldre er eldre enn barnet

Invarianter

Objekt-/klasseinvariant er invarianter som gjelder for alle instansene og hele objektet.

Vi må sikre at invarianten er sann i konstruktøren, og i hver metode gjelder de både inn og ut. Instansvariabler som er *public*, er vanskelig å sjekke, så det bør man unngå.

Argumentene til *public* metoder må sjekkes, og hvis de ikke kan håndteres så kaste et unntak som forteller hva feilen er.

Invarianter

Løkke-invarianter er invarianter som gjelder for en løkke. I for-løkker, er ofte det midterste feltet en invariant.

Men også målet med løkken vil kunne være en invariant.

Vi kan ha ulike betingelser som gjelder før løkken og andre etter. De kaller vi for–, og etterbetingelser. Samme gjelder også for metoder.

Invarianter

Invariantene kan hjelpe oss med å si hva som er feil, men det er også en måte å beskrive hva som er riktig måte å lage logikken på.

Men det er vel og bra, nå vet jeg hva som er rett og hva som er galt. Men fortsatt ikke nøyaktig hvor?

Debugging

Den enkleste men kanskje minst elegante måten for å finne ut om en invariant er brutt, eller hva tilstandene og betingelsene er:

I små programmer, og hvis du har et ganske klart avgrenset område, så går det helt fint. Men det kan også bli uoversiktlig.

Debugging

Utskrifter til terminalen kan også bli hengende igjen, og forvirre brukeren. Eller så kan det bli mye "støy" fra programmet ditt.

Debugging

I IN1000 var vi så vidt innom `assert`. Og det kan vi bruke i Java også. `assert` bruker vi får å teste om et boolskt uttrykk er sant, og hvis det ikke er det, så kastes et *AssertionError*-unntak.

Debugging

`assert` er ment som et verktøy for å hjelpe å finne feil, som betyr at ved vanlig kjøring så er alle linjene med `assert` ikke aktive

For å kjøre med assert aktivt så bruker du

```
$ java -ea ProgramKlasse
```

Det betyr også at du ikke skal bruke det for feilhåndtering.

Debugging

Når vi skriver klasser, så er en av idéene at vi inkapsler, men samtidig så har vi et grensesnitt som kan brukes av andre.

Det betyr at sjekk av argumentene til parameterne til metoder som er *public*, bør vi ikke bruke med assert. Feil i argumentene må brukeren av klassen få beskjed om, og det vil ikke skje hvis ikke `assert` er aktivt.

I private metoder kan man godt bruke det.

Debugging

En vanlig måte `assert` er brukt er ved "design-by-contract"-programmering. Der definerer man for–, og etterbetingelser i tillegg til invariantene, som man legger inn i en "kontrakt" mellom programmereren av klassen og den som bruker klassen.

`assert` "sikrer" at "kontrakten" blir fulgt, men også andre former for testing kan brukes

jdb

Java har en egen enkel debugger som følger med. Vi kan bruke den for å få kunnskap om tilstandene i programmet.

For å bruke den lokalt, så må vi først kompilere programmet vårt med -g slik at vi får nok informasjon fra jdb

```
$ javac -g Program.java
```

deretter kjører vi programmet på samme måte som før, men bruker jdb i stedet for java

```
$ jdb Program
```

jdb

Vi må da først finne et sted i programmet vi ønsker å stoppe, vi kan bruke metodenavn eller linjenummer slik

```
> stop at Program.main
> stop at Program:5
```

Og så starter vi programmet ved å skrive

```
> run
```

jdb

Da vil programmet kjøre helt til vi kommer til et stopp (breakpoint).

Her kan vi kjøre programmet skritt for skritt.

For å kjøre neste kommando:

```
> next
```

Det samme men som også vil kjøre skritt for skritt i metodene som blir kalt

```
> step
```

jdb

Vi kan skrive ut verdien av et uttrykk

```
> print antallSauer
```

og vi kan gi ny verdi til en parameter

```
> set antallSauer = 12
```

for all informasjon, f.eks om et array eller objekt så kan vi bruke

```
> dump sauene
```

og alle lokale variabler

```
> locals
```

jdb

Når vi vil fortsette kjøringen fram til neste stopp, så bruker vi

```
> cont
```

eller hvis vi bare vil kjøre ferdig i metoden vi er i

```
> step up
```

og for flere og mer informasjon om kommandoer

```
> help
```

jdb

Tips til å gjøre det litt lettere

For å kjøre en kommando ved hvert stopp eller hopp, f.eks locals

```
> monitor locals
```

Hvis du vil repetere siste kommando

```
> !!
```

alternativt så kan du sette

```
> repeat on
```

som vil gjøre at du bare kan trykke enter for å repetere siste

jdb

Hvis du alltid skriver de samme kommandoene, så kan du lage en fil som heter jdb.ini eller .jdbrc i mappen du kjører fra. Der lister du opp de kommandoene du vil skal kjøres med en gang jdb starter.

Dokumentasjon av koden

Vi skriver kode i høynivåspråk som Java fordi vi og andre som leser koden skal forstå hva som skjer. Datamaskinen kan ikke Java, for at den skal forstå det så må programmet oversettes og kompileres.

Dokumentasjon av koden

Det viktigste er å skrive kommentarer der vi har viktige logiske tilfeller, og hvor store eller uventede endringer i tilstandene.

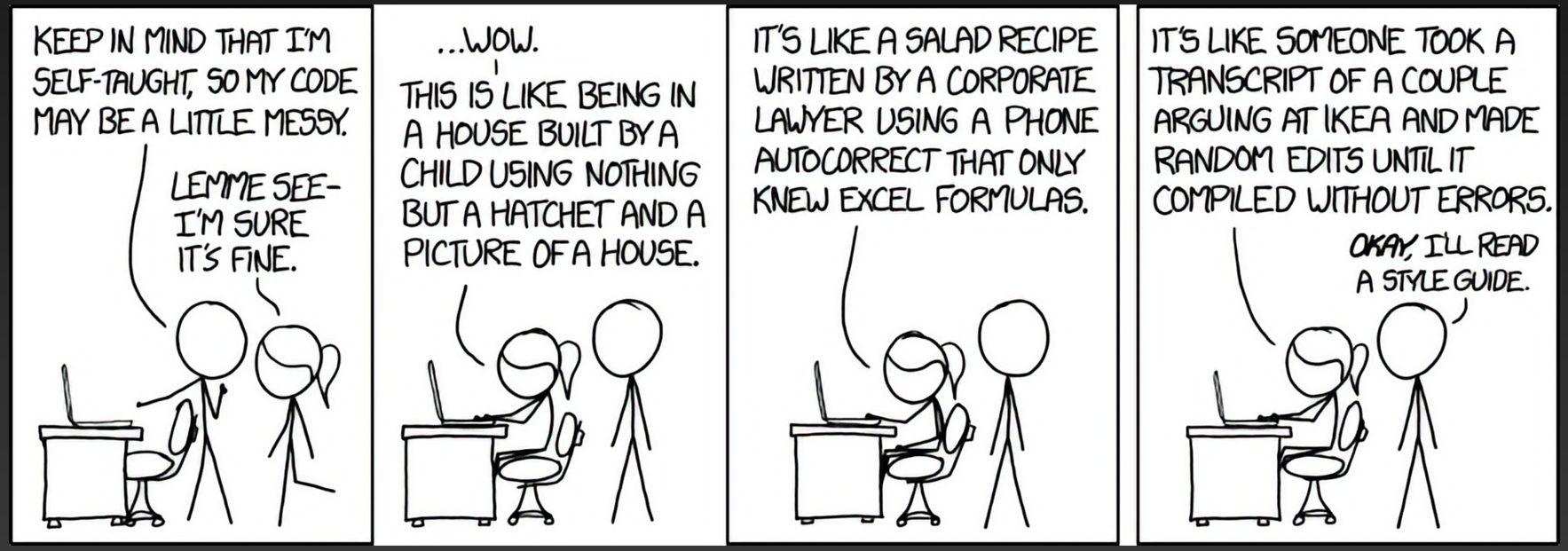
Invarianter og betingelser bør også kommenteres.

Variabel–, klasse–, metode–, og andre navn bør si noe om hva de er eller hva de gjør. Da vil koden vår forklare seg selv, og vi trenger kun skrive kommentarer for det som er viktigst.

Men unngå å skriv kommentarer som bare repeterer kodeuttrykket

Dokumentasjon av koden

Java har som de fleste andre språk, en spesiell kodestil som bør følges. Det finnes noen varianter, og jobber man sammen bli enig om felles stil.



javadoc

For å dokumentere kode på en detaljert og god måte så kan man bruke javadoc. Den lager dokumentasjonen ut fra kommentarene, og sier fra om noe mangler.