

Objektorientert tankegang

TDT4100

uke 2 – forelesning 2

Modellere med Objekter

Å beskrive et system ved hjelp av objekter:

- Hvordan skal objektene være?
- Kan de settes sammen av enklere objekter?
- Er den en utvidet eller modifisert type av andre objekter?
- Hvordan samhandler objektene for å utføre systemets oppgaver/prosesser.

Objektene er “virkelige” i programmet, selv om de representerer noe abstract, og har:

- Identitet
- Tilstand
- Oppførsel.

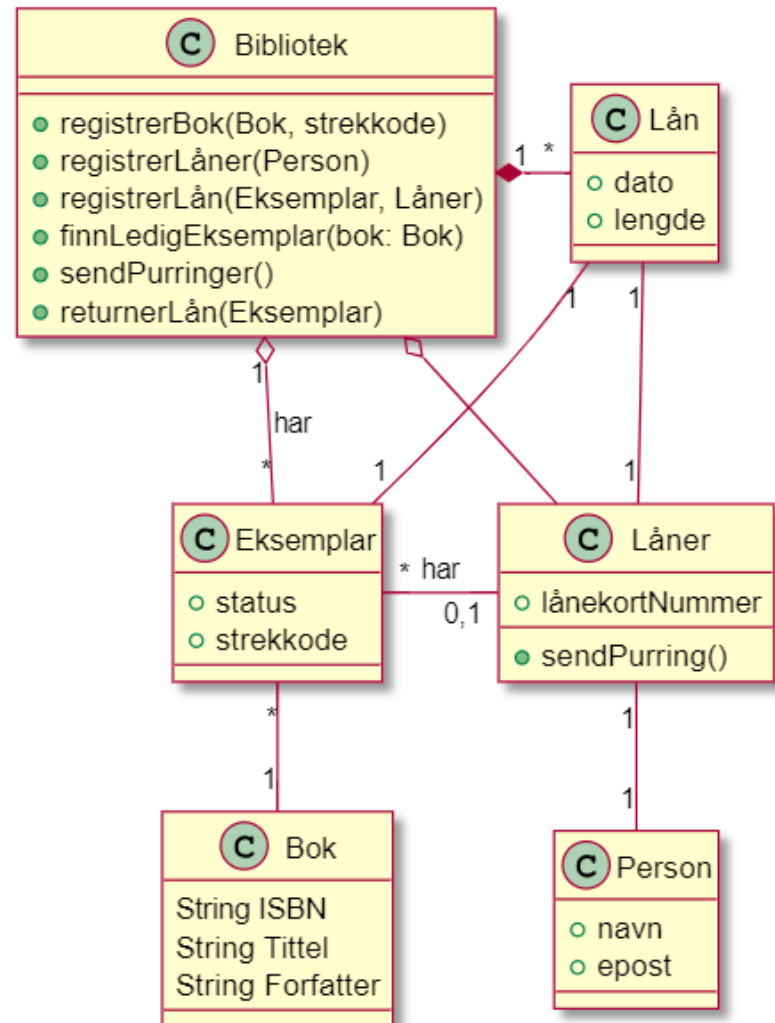
Som OO-utviklere må vi identifisere en hensiktsmessig modell og kunne implementere dette som kode.

Eksempel på modell – klassediagram (ufullstendig)

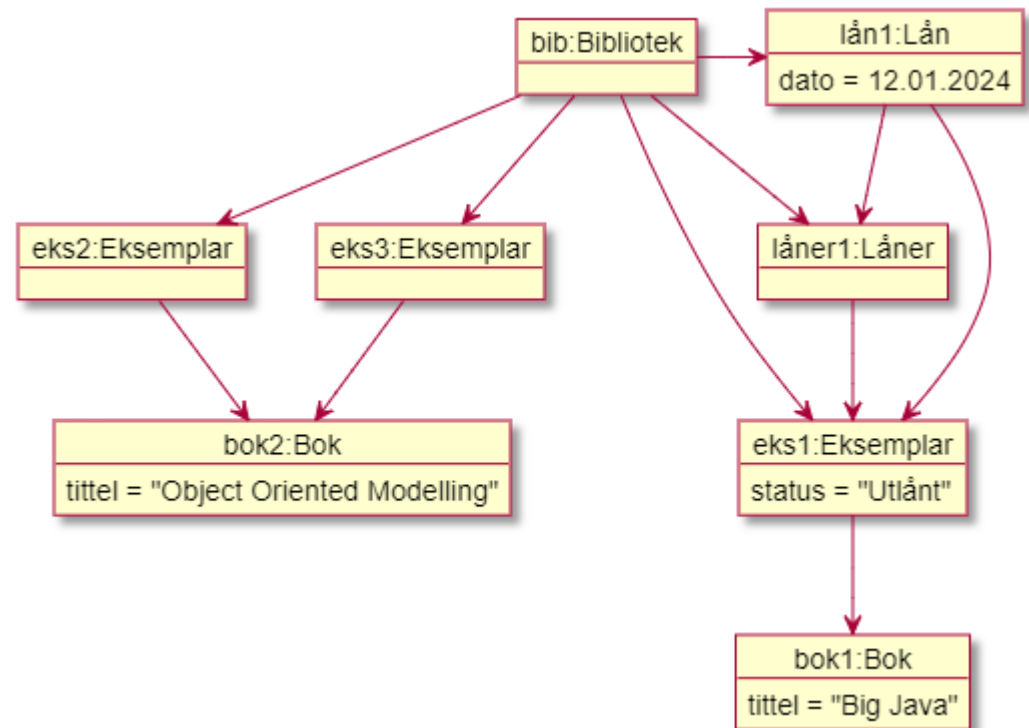
- Eksempel på klasser som definerer forskjellige typer objekter i en mulig modell
- For mer om klassediagrammer:

<https://www.ntnu.no/wiki/display/tdt4100/Klassediagrammer>

Klassediagram veldig enkelt Bibliotek



Objektdiagram



- Viser (noen av) objektene på et gitt tidspunkt.
- Hvert objekt er en «instans» av en klasse

Klasser og objekter i Java

Klasser definerer typer av objekter:

- Definerer felt for å holde tilstand
- Definerer metoder (funksjoner) som definerer oppførsel

Et objekt får vi ved å instansiere en klasse.

- Et objekt i Java er konkret, og har en unik identitet
- Hvert objekt får en egen kopi av hvert ikke-statiske feltene i klassen
- Metodene er felles for alle objektene av samme type (dvs. instansiert fra samme klasse)

NB!NB!NB! Hva er likhet?

Nevner det nå, dette kommer vi til å gjenta (lett å glemme):

Likhetsoperatoren (==) tester for objekt-identitet, dvs. er sann når vi har samme objekt på begge sider.

Ofte ønsker vi å test om INNHOLDET er likt. Det får vi vet å bruke en egen equals()-metode for klassen.

Dette gjelder ikke for de *primitive typene*. Disse er *int, byte, short, long, float, double, boolean and char*.

Objekter og primitive typer

Primitive typer (`int`, `byte`, `short`, `long`, `float`, `double`, `Boolean`, `char`) lagres direkte i variable.

- Ved tilordning kopieres selve data, for det er det som lagres
- Likhetsoperatoren `==` sammenligner selve data

Alt annet er objekter, og variable lagrer referanser som peker til objekter

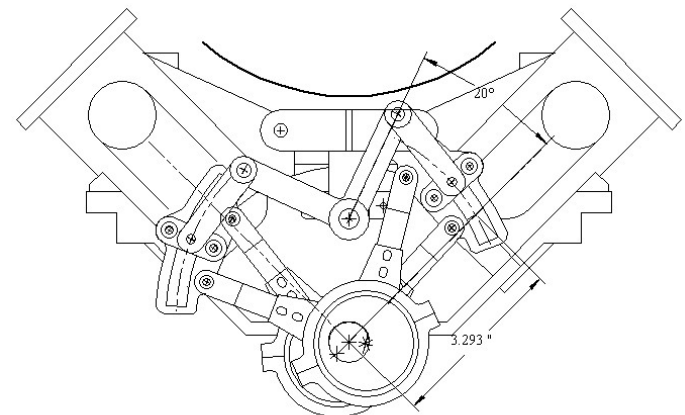
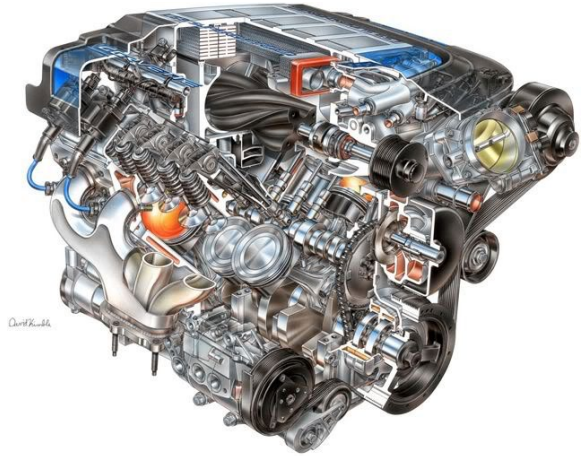
- Tilordning kopierer pekeren, så variable peker til samme data
- Likhetsoperatoren `==` sammenligner om det pekes til samme objekt.

Lage komplekse objekter

- Objekter vil ofte bygges opp av enklere objekter
- Dette kan gjøres på forskjellige måter
- Hva som er «riktig» måte, vil avhenge av situasjonen.
- Dette er et viktig tema for å lage gode objekt-orienterte modeller.

Motoranalogi

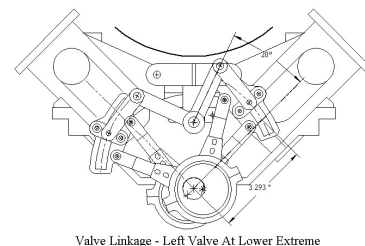
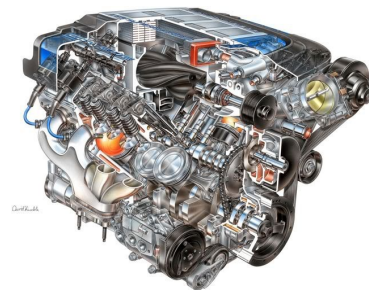
- Motor =
kjørende program
- Design
(maskintegning) =
programkode
 - Masse deler, og mulige oppførsler



Valve Linkage - Left Valve At Lower Extreme

Motor-analogi og OO

- System av deler / objekter
 - en motor består av mange deler
 - et kjørende program består av mange *objekter*
- Samvirkende deler / objekter
 - deler er koblet sammen og påvirker hverandre
 - objekter har *referanser* seg i mellom og *bruker* hverandre
- Designet / programkoden styrer tilvirkning og sammenkobling av delene
 - maskintegningen beskriver hvordan en del skal se ut og fungerer som en slags mal for å lage (en eller flere) av dem
 - en *klasse* beskriver oppførselen til en eller flere objekter, og er en slags *mal* for hvordan objektet er bygget opp



Viktige innsikter

- Kodene (tegningene) utformes for å gi det kjørende programmet (motoren) ønsket oppførsel (krav)
 - en må først tenke ut ønsket oppførsel, gjerne spesifisert som en test
 - skrive riktig programkode og teste den
- En må ha detaljert innsikt i prog.språket (fysikken)
 - objektorienterte programmeringsspråk generelt, og
 - Javaspråket og –maskineriet, spesielt
- Programmet (motoren) må testes mot krav
 - en må kjenne relevante brukstilfeller (typisk belastning), men også
 - teste mot spesialtilfeller (høy belastning, dårlig input/drivstoff, osv)
- Feilfinning må være systematisk
 - en må ha en hypotese om feil(situasjonen): hva skulle ha skjedd, og hva skjedde
 - debug-utskrift/logging (tilsvarende sensordata) har sine begrensninger
 - stopping i forkant av feil, trinnvis utførelse og inspeksjon av tilstand er viktig å lære

Hva gjør dette?

Kjenner dere igjen noe fra Python?

```
package counter;

public class DownCounter {

    int counter = 0;

    public DownCounter(int initCounter) {
        counter = initCounter;
    }

    public void countDown() {
        if (! isFinished()) {
            counter = counter - 1;
        }
    }

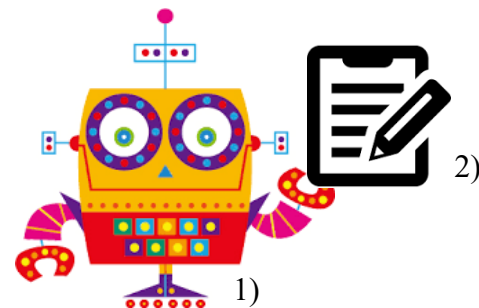
    public boolean isFinished() {
        return counter == 0;
    }

    public static void main(String[] args) {
        DownCounter dc1 = new DownCounter(2);
        System.out.println(dc1.isFinished());
        dc1.countDown();
    }
}
```

En enkel objekt-modell

- Objekt = robot med notatark

- roboten kan utføre et sett med *oppgaver*
- roboten har en notatblokk hvor den skriver ned nødvendig *data*
- roboten lages med et sett *innstillinger* (start-data)
- en kan ha mange roboter av samme *type*, med hver sine notatblokker (*data*)



1) <https://www.welovesolo.com/cute-cartoon-robot-colored-vector-set-12/>

2) <https://thenounproject.com/term/notepad/218406/>

- Objekt

- oppgaver = funksjoner, eller *metoder*
- data = variabler, eller *attributter*
- innstillinger = *start-verdier* for variabler

- Regler for oppførsel / virkemåte

- metodene er *funksjoner* som kjøres i *kontekst* av *variablene*, så metodene kan *bruke* (lese) og *oppdatere* (endre) variablene
- for hver oppgave må det defineres hvordan dataene brukes og oppdateres

Eksempel: nedtellingsobjekt

- Oppgaver

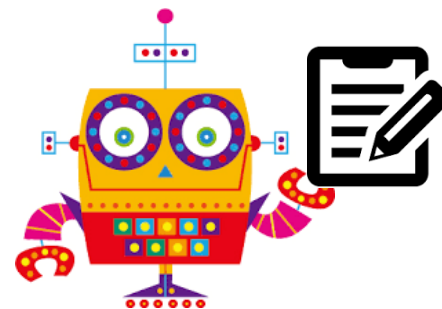
- telle ned ett trinn – `countDown()`
- si om en har telt helt ned – `isFinished()`

- Data

- `counter` - hvor mange trinn som gjenstår (teller ned)

- Innstillinger (start-tilstand)

- `counter` = hvor mange trinn som skal telles ned



- Regler for oppførsel

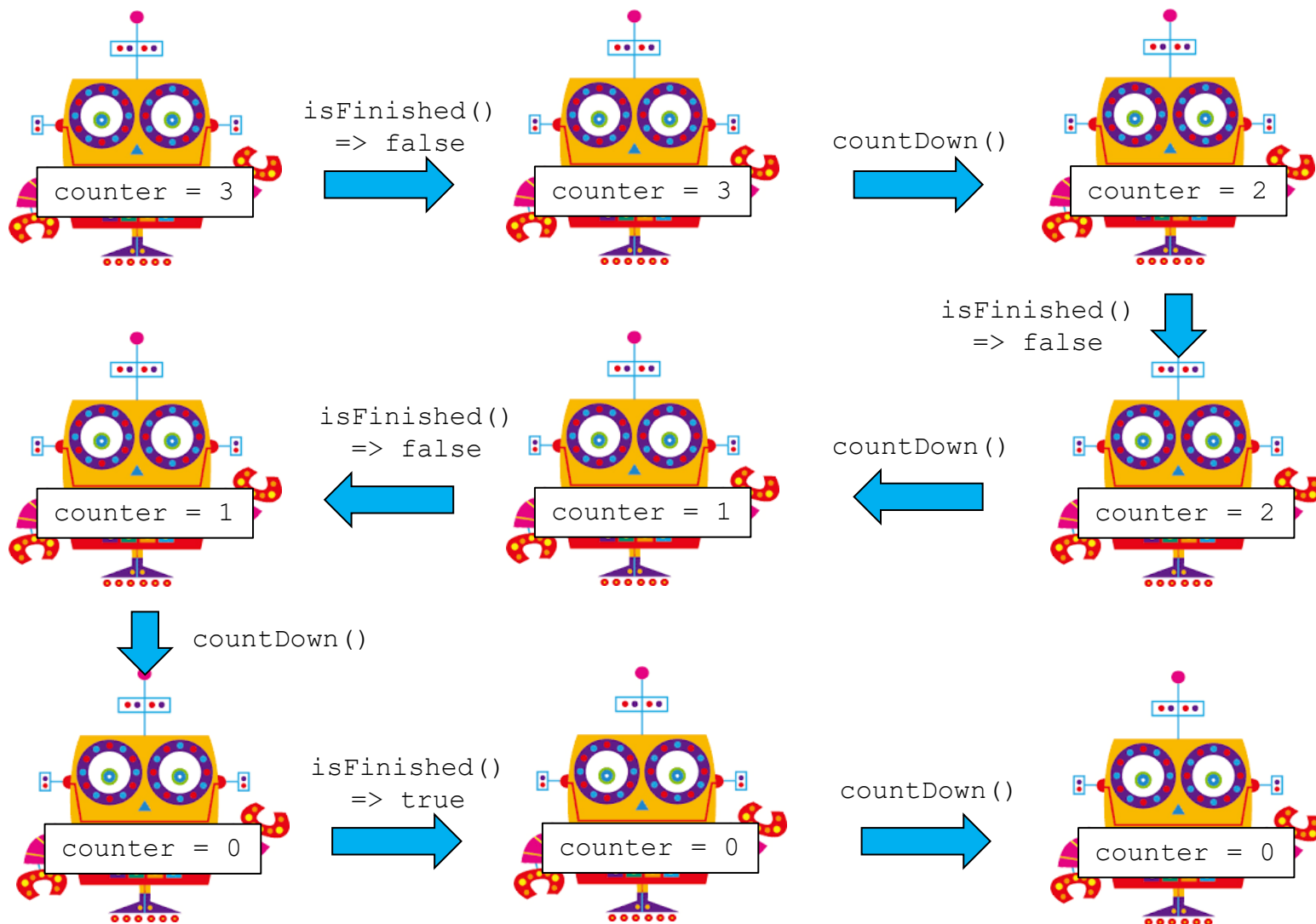
- `isFinished()` – sier om telleren er 0
- `countDown()` – minsker telleren ett trinn, hvis ikke grensen allerede er nådd

```
counter == 0
```

```
if (! isFinished())
    counter = counter - 1
```

Hvordan ville reglene blitt hvis en i stedet telte opp?

Eksempel på oppførsel



Java – prosedurelt?

- Det går an å programmere i Java uten å bruke objekter aktivt – men det er ikke formålet med dette kurset
- Mister formålet med objekt-orientering – dere har et halvt år på å lære nettopp det!

deklarasjon av klassen DownCounter

Java-koden

```
public class DownCounter {
```

```
    int counter; ← deklarasjon av objektvariabel/attributt/felt)
```

```
    DownCounter(int initCounter) { ← deklarasjon av  
        counter = initCounter;      init-funksjon/konstruktør  
    }
```

```
    boolean isFinished() { ← deklarasjon av  
        return counter == 0;    objektfunksjon/metode  
    }
```

```
    void countDown() { ← "ingen verdi"-  
        if (! isFinished()) {      typen  
            counter = counter - 1;  
        }  
    }
```

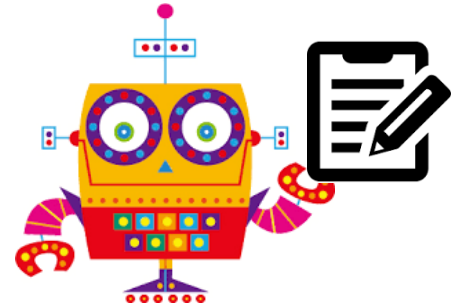
enkle setninger avsluttes med ;

kodeblokker
avgrenses med { },
inntrykk er "pynt"

"hvordan kjøres egentlig programmet?"

Eksempel: gjennomsnittsobjekt

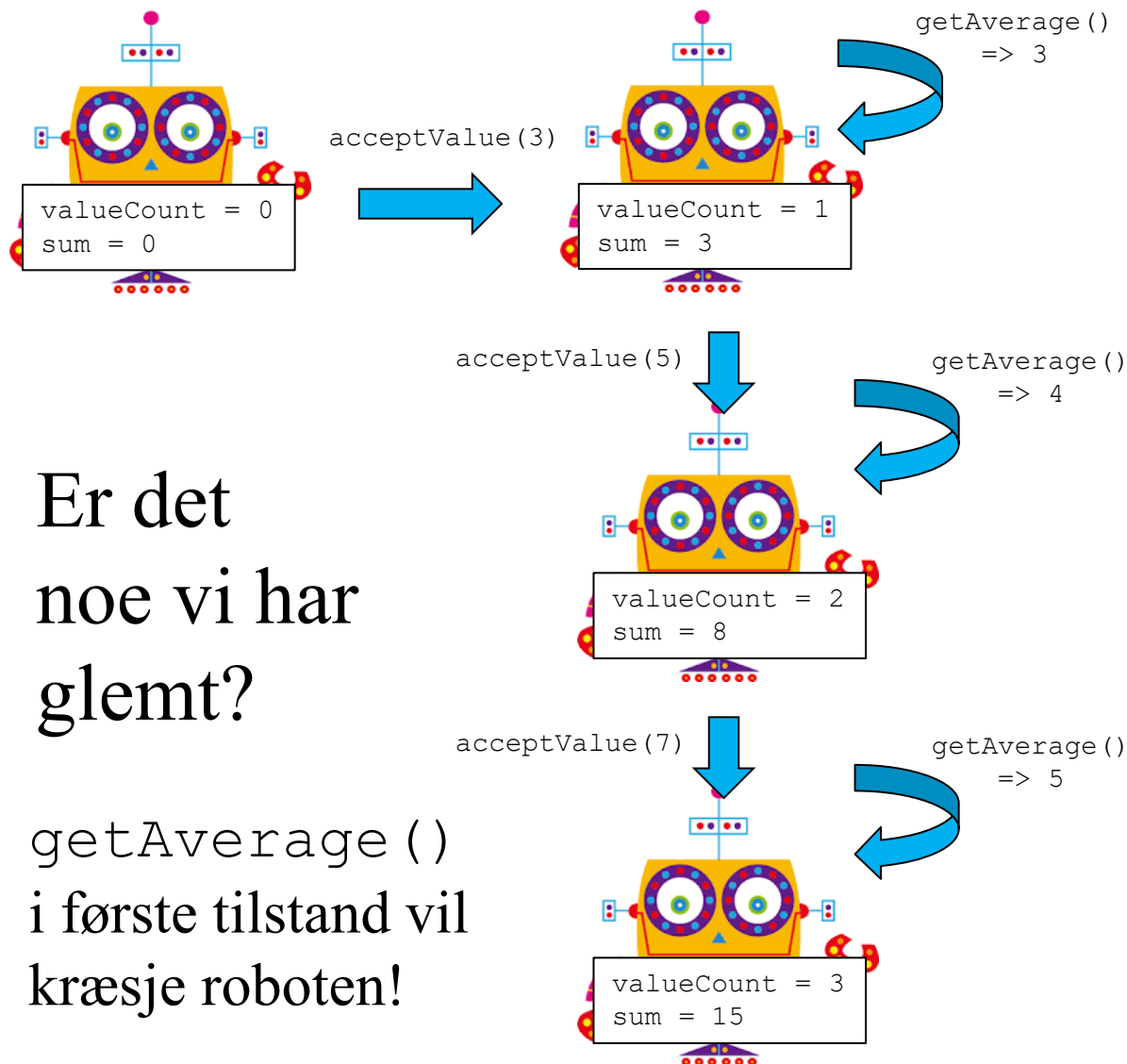
- Oppgaver
 - ta i mot et nytt tall – `acceptValue (value)`
 - gi oss gjennomsnittet – `getAverage ()`
- Data
 - `valueCount` – antall mottatte verdier
 - `sum` – summen av verdiene som (hittil) er mottatt
- Innstillinger (*starttilstand* for nye objekter)
 - `valueCount = 0, sum = 0`
- Regler for oppførsel
 - `acceptValue (value)` – øker teller og sum
 - `getAverage ()` – beregner og returnerer gjennomsnittet



```
valueCount = valueCount + 1
sum = sum + value
```

```
return sum / valueCount
```

Eksempel på oppførsel



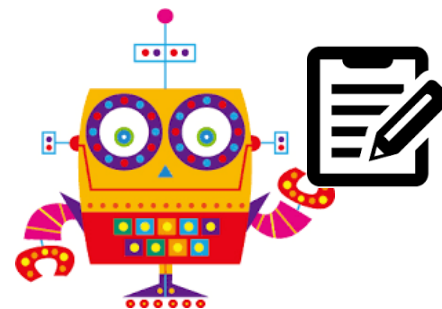
La oss programmere Average-klassen

Hvordan virker Random

- Lag en klasse Random100
- Metode getRandom100 returnerer et tilfeldig heltall i intervallet [0, 99]

Eksempel: random-objekt

- Oppgaver
 - gi ut et (nytt) tilfeldig heltall mellom 0 og 100 (99) – `random()`
- Data
 - ?
- Innstillinger
 - ? = ?
- Regler for oppførsel
 - Her trenger vi ikke finne opp hjulet.
 - Bruke java sin random-generator

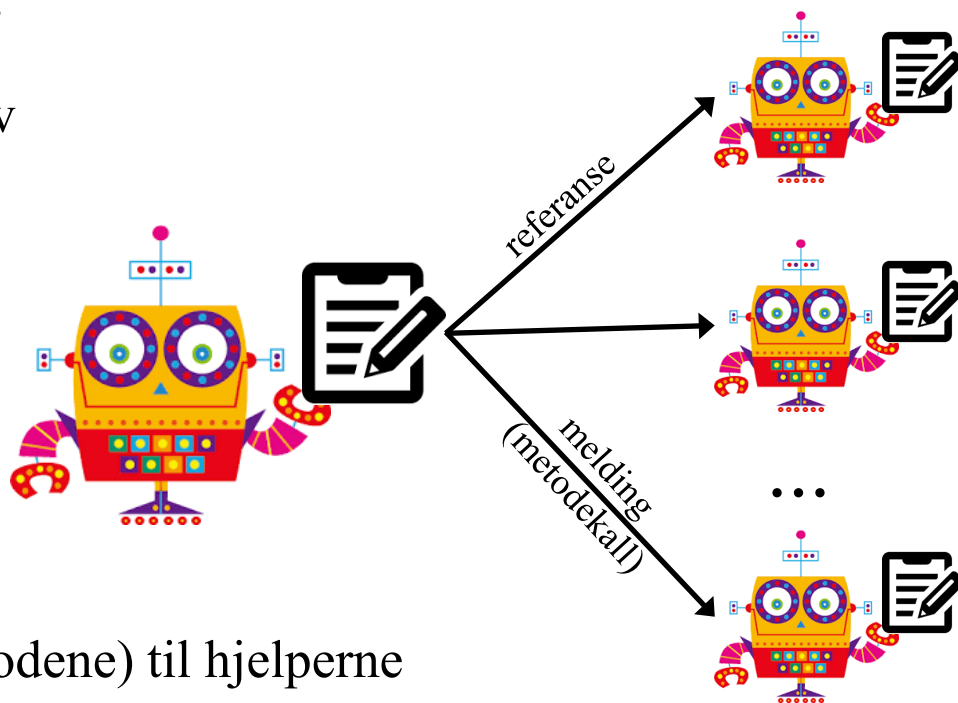


https://en.wikipedia.org/wiki/List_of_random_number_generators

<http://docs.oracle.com/javase/6/docs/api/java/util/Random.html>

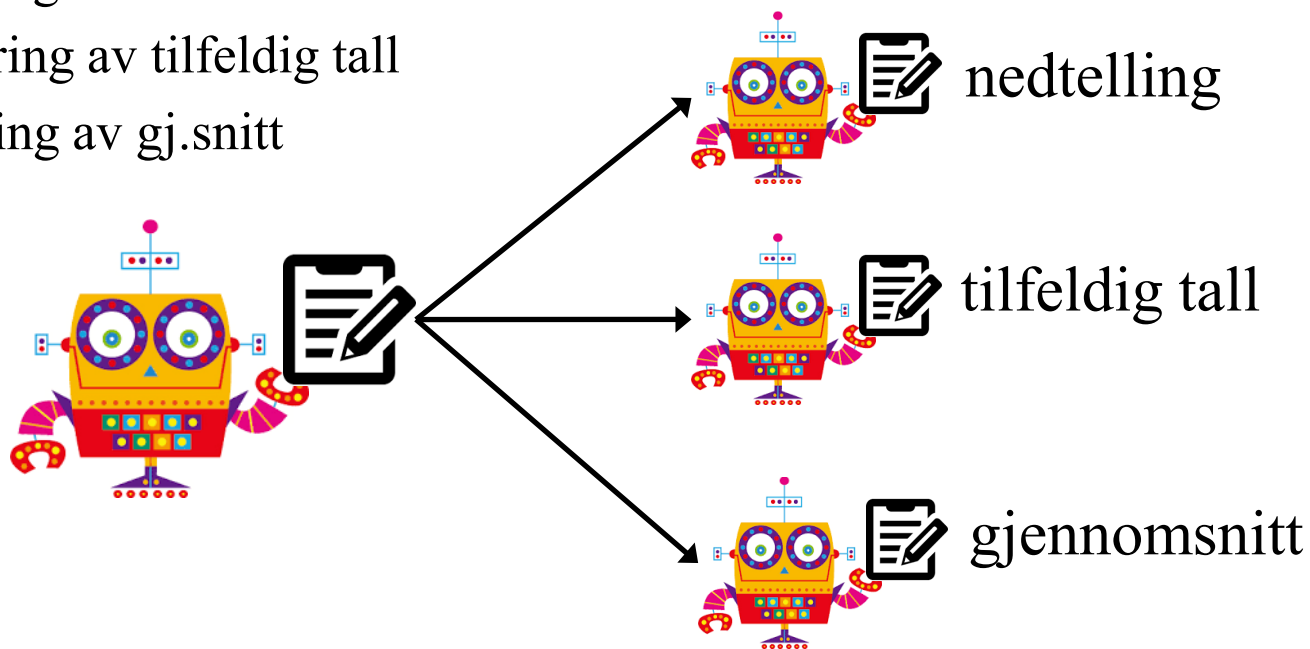
Eksempel: koordinator-objekt

- Et system består av samhandlende objekter, med ulike *roller*
 - hjelpere, som styres av
 - koordinatore
- Samhandling
 - koordinatoren har *referanser* (piler) til hjelpene
 - koordinatoren sender meldinger (kaller metodene) til hjelpene
- **Programobjektet** er objektet som koordinerer (øverst i hierarkiet)!



Eksempel: beregning av gjennomsnitt av tilfeldig tall

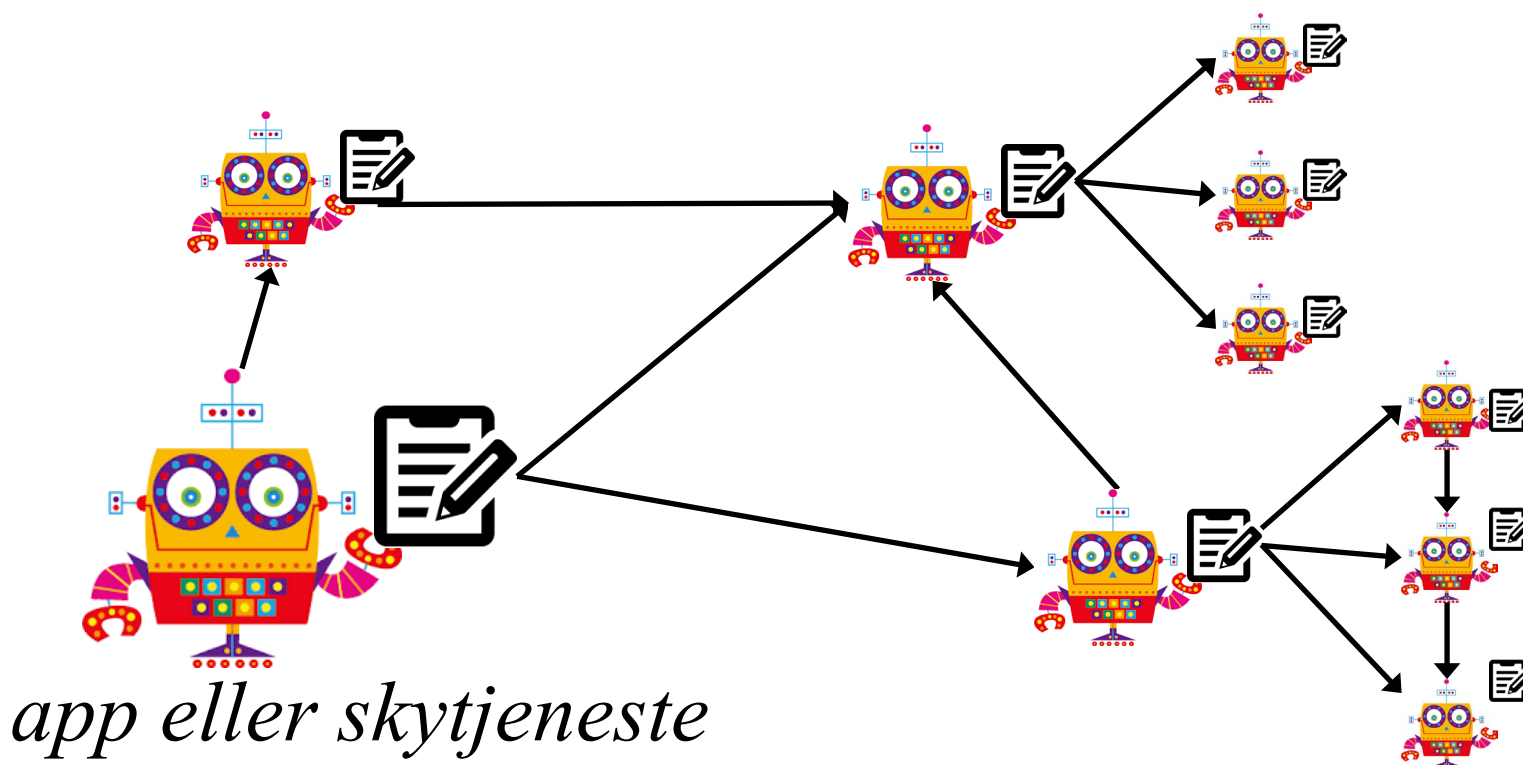
- Én koordinator
- Hjelpere for
 - nedtelling
 - generering av tilfeldig tall
 - beregning av gj.snitt



La oss programmere klassen!

- Lage ti tilfeldige tall opp til 100
- Skrive ut gjennomsnittet av dem
- Bruke Average, DownCounter, Random

Komplekse systemer krever kompleks samhandling



Kunsten er å dele et system opp i
“naturlige” objekter med “ryddig” samhandling

Men hvordan “utføres” et objekt?

- Et objekt må lages først
 - `new` <klassenavn>(... argumenter til *konstruktør*...)
- Siden så utføres metoder
 - <objekt-ref>.<metodenavn>(... argumenter ...)
 - mens metoden kjøres, så har objektet “kontrollen”
- Kontrast til prosedyre-orientert kode
 - kjøres ofte fra topp til bunn, og er så ferdig, eller...
- Java har sin **main**-metode, se **DownCounter**-eksempel...

JShell

- Lar en skrive inn Java-snutter
 - variabel-deklarasjoner
 - enkle og sammensatte uttrykk og setninger
 - variabelreferanser
 - metodekall
 - **if**, **while** og **for**
 - (re)deklarasjoner av metoder og til og med hele klasser
- Støtter såkalt “completion”
 - kan foreslå variabler og metoder, basert på kontekst
- Laget for enkel utprøving av kode og skripting (kan lagre og laste inn kode)

Utprøving av objekter med JShell

- JShell - interaktiv utprøving av Java-kode

```
[dhcp-110-148:examples hal$ jshell  
| Welcome to JShell -- Version 9.0.1  
| For an introduction type: /help intro
```

```
[jshell> String s = "Java er gøy!"  
s ==> "Java er gøy!"
```

```
[jshell> s.substring(8)  
$2 ==> "gøy!"
```

```
[jshell> java.util.Random rand = new java.util.Random()  
rand ==> java.util.Random@31dc339b
```

```
[jshell> rand.n  
nextBoolean()      nextBytes(      nextDouble()      nextFloat()      nextGaussian()      nextInt(
```

```
[jshell> rand.nextInt(10)  
$4 ==> 2
```

```
[jshell> rand.nextInt(10)  
$5 ==> 0
```

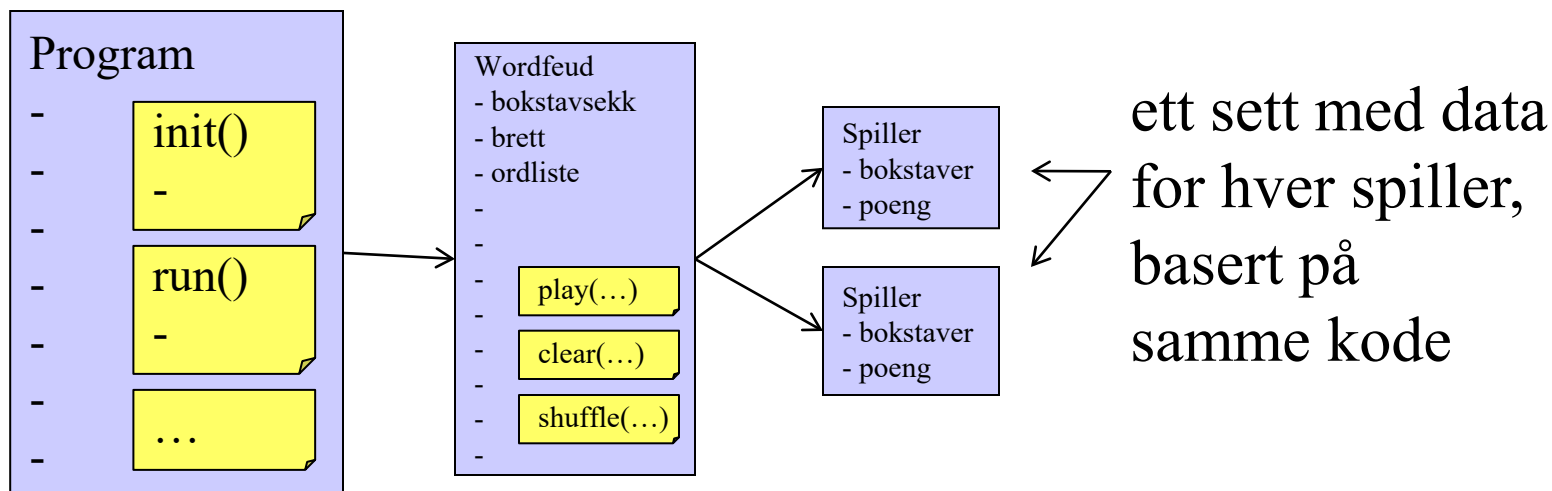
```
[jshell>
```

Designproblemet

- Hvordan dele et system opp i objekter?
 - Hvordan dele opp totaloppførselen og -tilstanden i mindre enheter?
 - Hva er kriterier for en god oppdeling?
- Erfaring og magefølelse
 - objekter det er lett å sette navn på
 - gjenkjennbare oppgaver og strukturer
 - noen oppdelinger virker ryddigere enn andre
 - oppdelingen må ikke representere virkeligheten
- Konvensjoner og mønstre
 - selv unike problemstillinger har gjenkjennbare delproblemer som allerede har kjente løsninger
 - såkalte ”design patterns” (standardteknikker) er viktig innen objektorientering

Wordfeud-eksempel

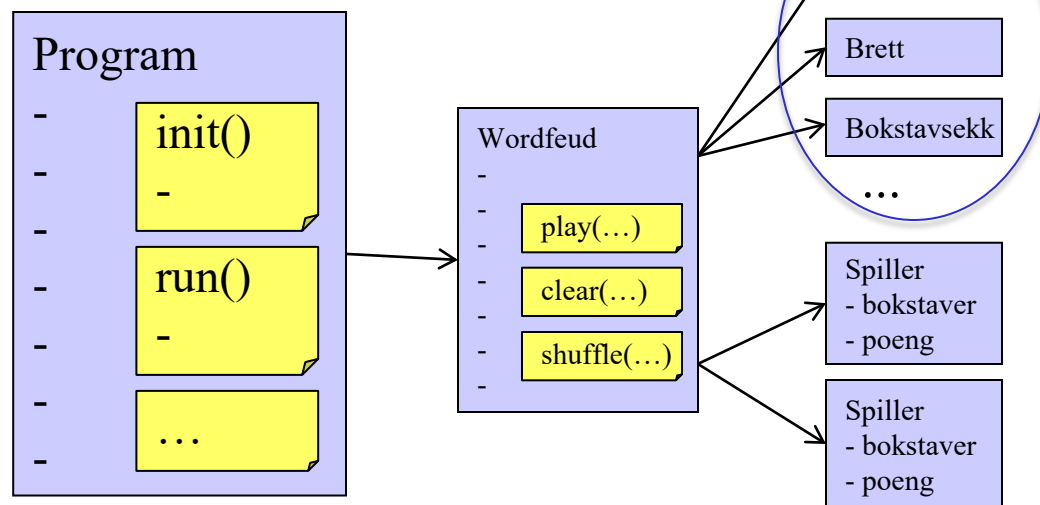
- Wordfeud
 - Deler opp i notater/objekter for spillbrettet og spillerne
 - Hovedprogrammet styrer det hele



Wordfeud-eksempel

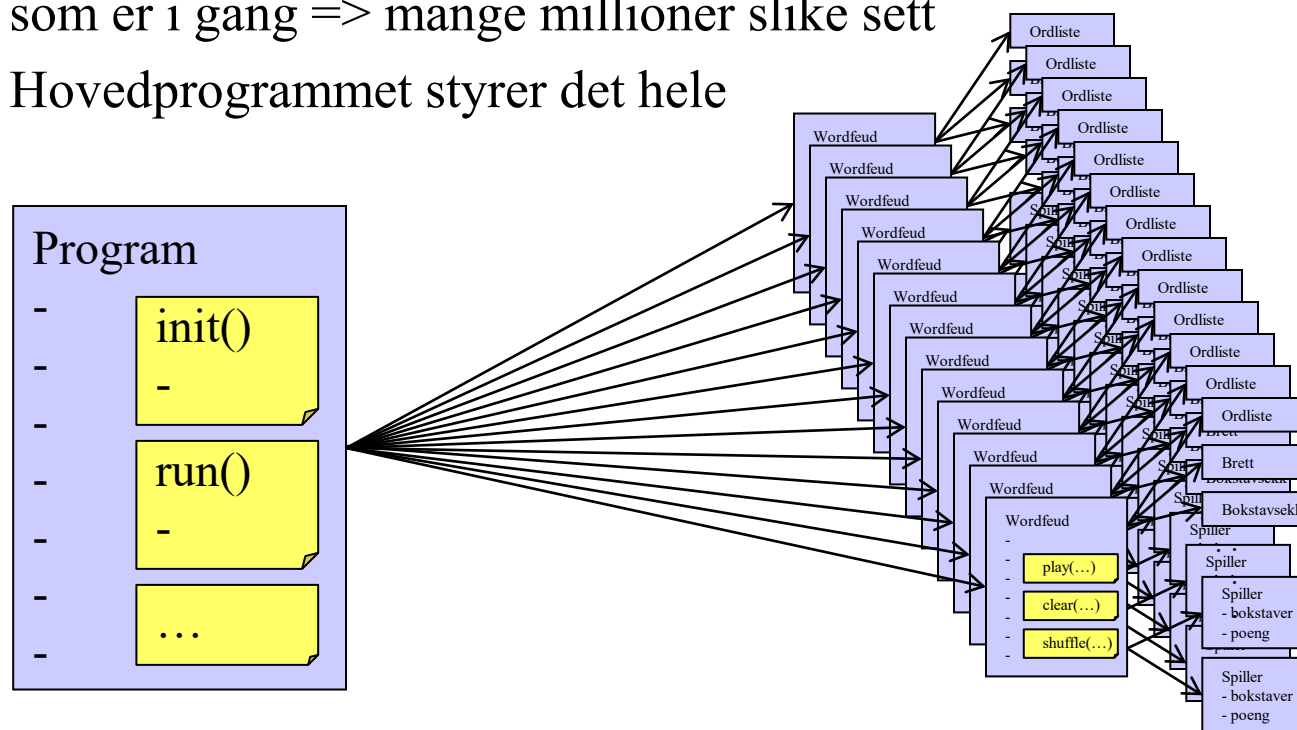
- Wordfeud

- Deler opp i objekter for spillbrettet og spillerne
- Hjelpeobjekter for ulike spillelementer
- Hovedprogrammet styrer det hele



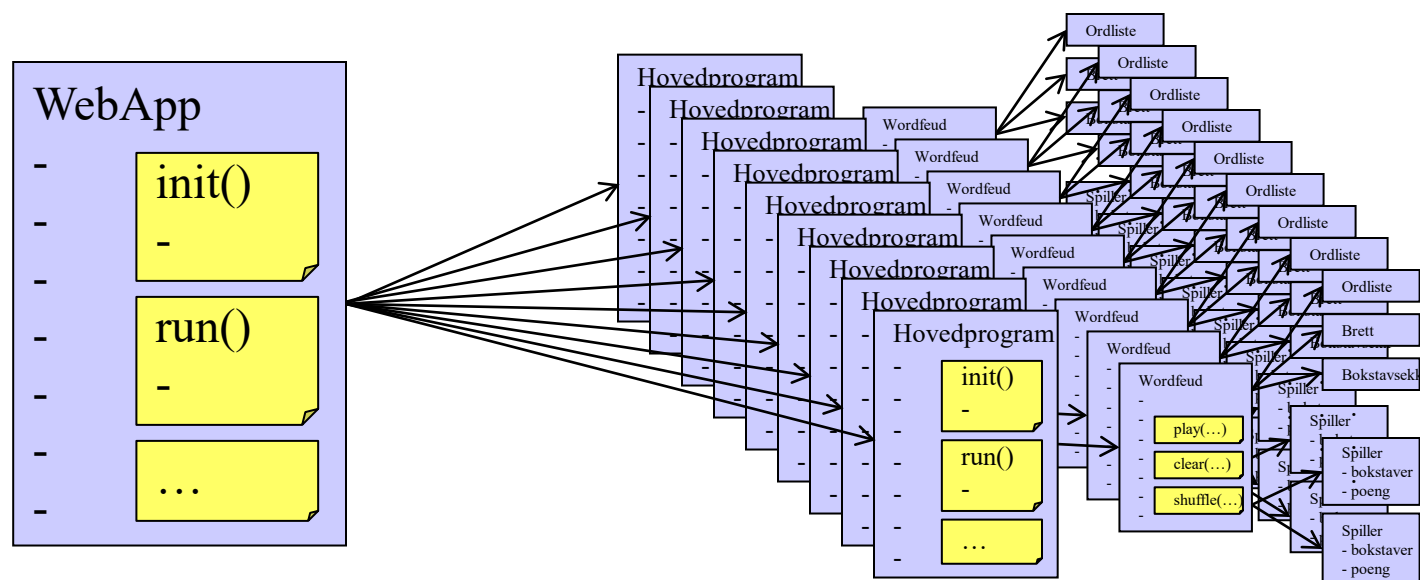
Wordfeud-eksempel

- Wordfeud på nettet
 - Ett wordfeud-objekt med tilhørende underobjekter for hvert spill som er i gang => mange millioner slike sett
 - Hovedprogrammet styrer det hele



Wordfeud-eksempel

- Wordfeud på nettet
 - Ett wordfeud-objekt med tilhørende underobjekter for hvert spill som er i gang => mange millioner slike sett
 - Hovedprogrammet styrer det hele



Matlagings-analogi

- **Utførelse** i kontekst av **hva det utføres** på: Matvarer uten utførelse eller utførelse uten matvarer gir lite mening

Kyllingfilet med poteter i tomatsaus



Under 20 min Enkel

Del Legg til Mobil Skriv ut

Det er hverdager det er flest av. Her er en god hverdagsmiddag med kyllingfilet som er ferdig på ca. 20 minutter.

Porsjoner OK

Ingredienser

2 stk kyllingfilet
1 ss margarin til steking
1 stk løk
2 båt finhakket hvitløk
1 boks hermetiske tomater
1 boks tomatpuré
¼ ts salt
¼ ts pepper
¼ ts sukker
1 ts tørket oregano
4 stk potet i biter
1 stk brokkoli

Slik gjør du:

1. Stek kyllingfiletene i en stekepanne på middels varme i 2 minutter på hver side. Legg over lokk og etterstek i 6-8 minutter.
2. Stek løk og hvitløk på middels varme til løken blir blank. Tilsett hermetiske tomater, tomatpuré og oregano, og la alt surre på svak varme i ca. 15 minutter. Smak til med salt, pepper og litt sukker.
3. Kok potetbitene ca. 10 minutter, ha dem i tomatsausen og la alt bli gjennomvarmt. Del brokkolien i buketter og kok dem så vidt møre i litt vann.

Data + kode = sant!

- Kode utføres alltid i kontekst av data
 - Hvilke trinn må løsningsprosessen deles opp i?
 - Hvilke data trengs for å representere et problem?
 - Hvilke data trengs for å støtte løsningsprosessen?

- Matlagingsanalogi:
 - kjøkkenet er stedet hvor matvarene er og som arbeidet foregår på



Data + kode = sant!

- Data deles opp etter
 - hvilke data som uløselig hører sammen.
F.eks. hører for- og etternavn sammen, men er løst knyttet til fødselsdato (som består av dag, måned og år)
 - hvilke data som trengs på samme tid, til samme formål?
 - Den beste oppdelingen er ikke alltid basert på vår reelle verden
- Matlagingsanalogi:
 - Alle ingrediensene i én (del)rett samles.
 - Alle ingrediensene som skal has oppi samtidig samles.

Objektorientert vs. Prosedyreorientert

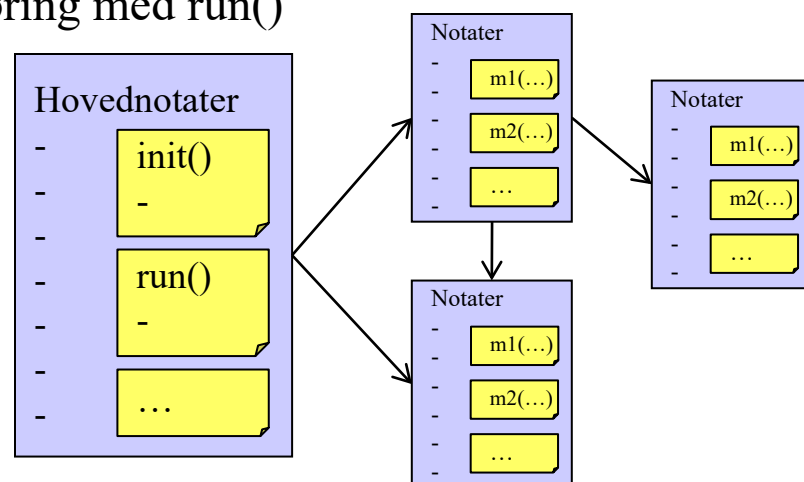
- En objektorientert tilnærming
 - grupperer data etter bruksmønster
 - samler beskrivelsen av data og instruksjoner
 - instruksjonene utføres alltid i kontekst av dataene
- En prosedyreorientert tilnærming
 - fokuserer i større grad på behandlingen enn dataene
 - har svake mekanismer for å dele et program opp i små, gjenbrukbare enheter
- Matvare-analogien
 - OO: Én arbeidsbenk pr. (del)rett, med sine matvarer og deloppskrift, som koordineres av en sjefskokk som samler de ferdige rettene på sitt bord.
 - PO: Alle matvarer på én benk, ordnet sammen med verktøyet som behandler dem.

Klasse vs. objekt

- Klassene er oppskriften/programkoden
 - hver klasse utgjør én deloppskrift
 - programklassen koordinerer deloppskriftene
- Objektene utgjør tilstanden til det kjørende programmet
 - objektene tilsvarer matvarene og oppskriften som en utfører, på vei til å bli en matrett
- Klassen beskriver
 - hvordan objektene ser ut ved oppstart
 - hvordan objektene endrer tilstand over tid, når metodene blir utført

Program vs. klasse

- Et program er et spesialtilfelle av det en generelt kaller *klasser*
- Klasser er den fundamentale programvare-enheten: data + kode
- En klasse er et program dersom det
 1. er klassen som startes ved kjøring
 2. har en eller flere standard-metoder som håndterer programmets *livssyklus*, f.eks. initialisering med `init()`, kjøring med `run()`
- Programklassen må selv “rigge opp” de andre notatarkene



Diagrammer

- Diagrammer hjelper oss å forstå
 - tilstand ved kjøretid
 - koden ved å vise innholdet i klasser og sammenhenger mellom dem
- Objektdiagrammer
 - viser hvilke notatark/objekter som finnes underveis i kjøringen av programmet og hvilke verdier som variablene (attributtene) har
 - koblingen (piler) til andre objekter
- Klassediagrammer
 - viser klasser med variabler (counter), konstruktører og metoder (isFinished() og countDown())
 - kobling til andre klasser

```
downCounter1: DownCounter
counter = 5
```

```

classDiagram
    class DownCounter {
        +int counter
        +DownCounter(int initCounter)
        +boolean isFinished()
        +void countDown()
    }
  
```

Hva har vi lært så langt?

- Data grupperes i “notatark” som fungerer som kontekst for utførelse av kode/metoder
- Apper består av
 - *fasade, gjerne beskrevet med FXML i egen fil*
 - kontroller-objekt(er), som bygger bro til
 - intern tilstand, én eller flere objekter
- Diagrammer illustrerer både koden og tilstanden under kjøring