UNIVERSITETET I OSLO

Institutt for informatikk

Velkommen til
IN1010 –Objektorientert programmering
Våren 2022

Idag:

1. time: Om IN1010

2. time (+ i morgen og to dobbelttimer neste uke):

Java for deg som kan Python

1

Siri Moe Jensen , Stein Gjessing, ++

Institutt for informatikk

Universitetet i Oslo

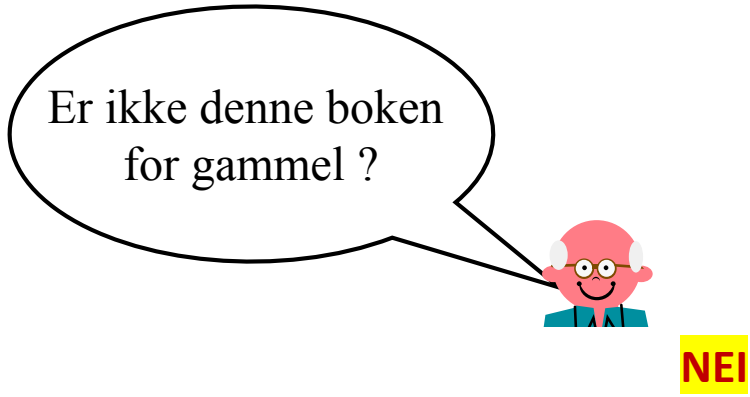
IN1010 –Objektorientert programmering

I IN1010 skal du lære:

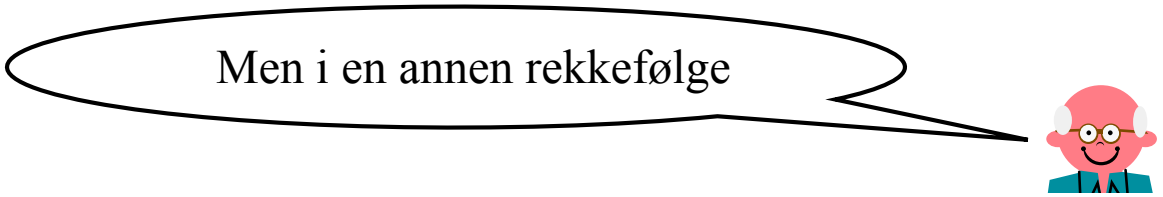
- Å løse et (middels stort) problem ved å lage et program
- Å lage gode (middels store) programmer
- Vite hva et godt program er
- Kunne bruke en del gode programmeringsteknikker
- Objektorientert programmering

Læreboka: Ikke kjøp, finnes i Leganto

- Horstmann: Big Java, Late objects 2013
- Kap 1 –8 Introduksjon til Java (IN1000 - men Java)

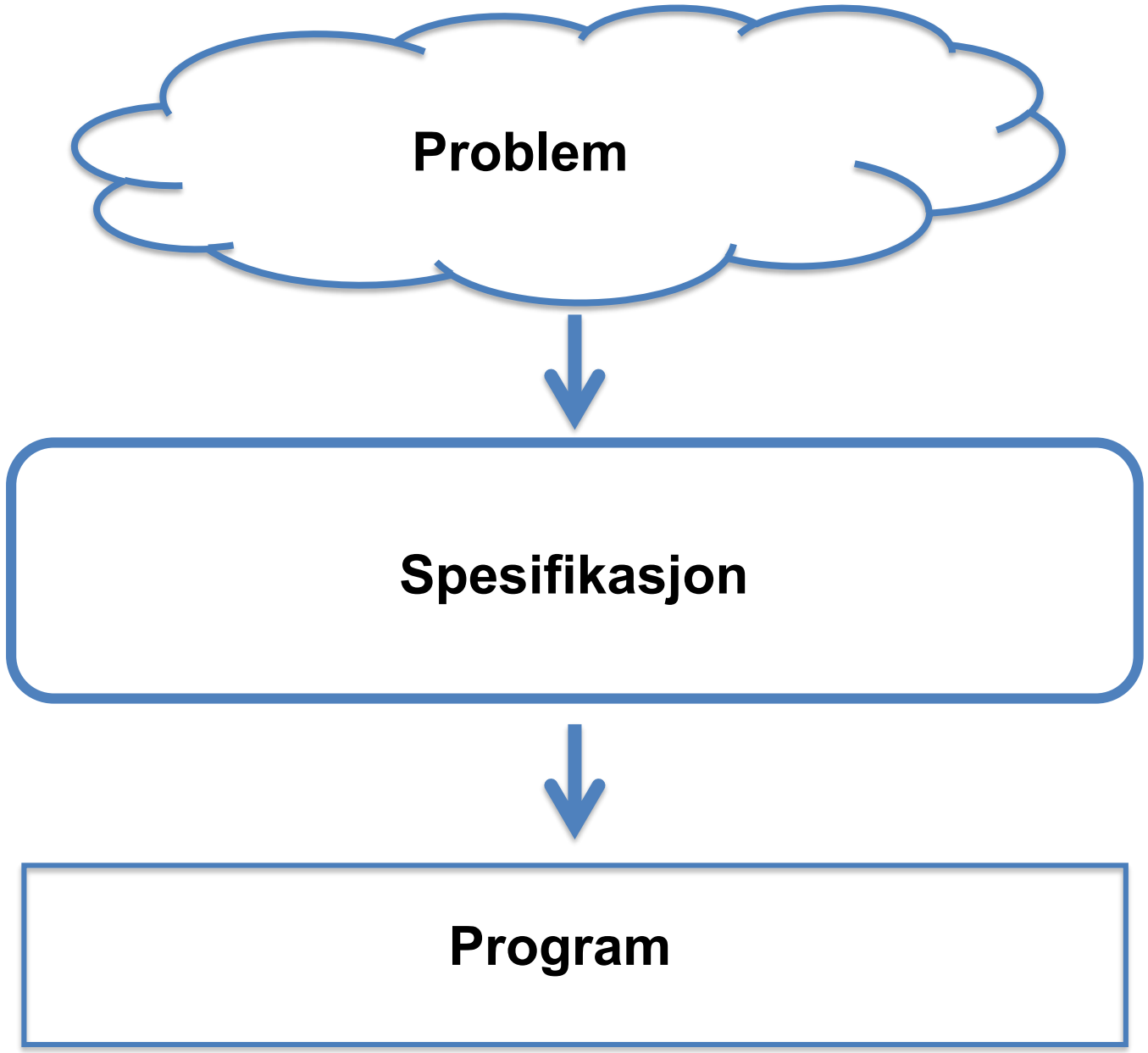


- Kap 9 – arv
- Kap 10 og 11 – GUI
- Kap 12 – design
- Kap 13 – rekursjon (ikke alt)
- Kap 14 – sortering og søking (bare som eksempelstoff)
- Kap 15 og 16 –datastrukturer (ikke alt)
- Kap 17 – ikke pensum (IN2010)
- Kap 18 – klasser med parametre
- Kap 20 – tråder



Objektorientert programmering

- Ble funnet opp i Oslo for 55 år siden av Ole-Johan Dahl og Kristen Nygaard.
  - Språket de fant opp heter Simula-67
  - C++, Smalltalk, Java, C#, . . . bygger på Simula
- Foreleserene i IN1010 (Siri, Dag, Stein) hadde Dahl og Nygaard som lærere
- Pensum er ikke først og fremst læreboka (Horstmann), men det som foreleses (lysarkene). Disse inneholder mye av det vi har lært av Dahl og Nygaard
- Ofte en annen rekkefølge / vinkling enn Horstmann


Problem
Spesifikasjon
Program

Eksemplene i IN1010

- Mange av de konseptene vi behandler er viktigst når programmene blir større
- Mange programmer i IN1010 er små
- Konseptene blir introdusert med små, enkle programmer
- Tenk på dette når dere lærer noe nytt
  (vi forelesere skal også minne dere på det)

- I noen obligatoriske oppgaver prøver vi å løse litt større problemer

I IN1010 skal du lære

- å programmere: dvs. å tenke seg hvordan oppgaven som programmet ditt skal løse, skal løses ved hjelp av **datastrukturer (objekter)** og **algoritmer** inne i datamaskinen
- å programmere: dvs. lage den koden som lager de **datastrukturene** og implementerer de **algoritmene** du har planlagt
- Derfor må du lære deg å visualisere eller **tegne datastrukturer**
  - for å planlegge programmet ditt
  - for å implementere programmet ditt
  - for å kommunisere om programmet med andre

IN1010 forkunnskapskrav

- Du kan løse små problemer i Python vha.
  - Variable og konstanter
    - Verdier typer / uttrykk
  - Tilordninger
  - Kontrollstrukturer (valg og løkker)
  - Lister/ ordbøker
  - Metoder/funksjoner (med parametre og returverdier)
  - Klasser
  - Objekter
    - Grensesnitt (public –non-public)

Denne uken (og neste)

- I dag første time: IN1010; "IN1010-kontrakten"
  - Undervisningstilbud og egne aktiviteter

- Andre time i dag, i morgen og neste uke: **Java for deg som kan Python**.
  Eget arbeid:
  - Les foilene
  - Les Horstmann kap 1 –8.
  - Les notatet "Litt om datastrukturer i Java "av Stein Gjessing
  - Gjør de to første ukenes Trix-oppgaver –bli "flytende" i Java
  - Løs og lever inn »Innleveringsoppgave 0»
  - Delta på grupper / seminarer
  - Delta på lab med egen veileder
  - Start første obligatoriske oppgave (frist mandag 7. februar) –2 ukers arbeid

Obligatoriske oppgaver

- Du må ha godkjent alle de 7 obligatoriske oppgavene for å ta eksamen
- Har du godkjent **alle** de obligatoriske oppgavene i IN1010 fra før (dvs. du kunne gått opp til eksamen) trenger du ikke gjøre noen oppgaver på nytt i år
  - Hvis ikke må du gjøre alle oppgavene i år
  - Hvis OK fra før: Automatisk inne i systemet
  - Alle oppgavene er forandret i forhold til 2021
    - Du må gjøre ÅRETS oppgaver.
- Hver oppgave er ca to ukes arbeid (ved siden av annet arbeid i IN1010)

Obligatoriske oppgaver

- Etter at et (eller flere) tema er forelest vil det (nesten alltid) bli gitt en obligatorisk oppgave med dette stoffet.
- Du får vanligvis to uker på deg til å løse oppgaven **etter** at stoffet er gjennomgått
  - Noen ganger bare én uke på deler av stoffet
- Men NB!: Dette betyr at du må jobbe med den obligatoriske oppgaven i det gamle stoffet SAMTIDIG med at du lærer annet nytt stoff.

Obligatoriske oppgaver
Innlevering annenhver mandag

Stort sett uavhengige oppgaver, men

- Oppgave 3 og 4 bygger på hverandre og på 2
  - Oppgave 4 skal gjøres i grupper

Å bruke 4 timer per uke på obligatoriske oppgaver er normert ut fra en gjennomsnittsstudent som får til en passe bra innlevering.

Obligatoriske oppgaver

- Skal løses individuelt
  - Diskuter gjerne ideer med andre, men **du skal skrive og forstå all koden selv**
  - Ikke lov å kopiere fra andre
- Du må lese regelverket for obligatoriske oppgaver (lenke fra semestersiden)
- Strengeste reaksjonsform ved fusk: Utestengning
- **Frist for innlevering**: Annenhver mandag kveld kl. 23:59

- Retting: Samretting eller tradisjonell retting
  - informasjon kommer på kurssiden

Eksamen

- Onsdag 8. juni 2022 kl 9 –13 i Silurveien
- Hvis forandringer pga.smittevern:
  - Kanskje hjemmeeksamen
  - Kanskje 6 timer onsdag 8. juni (9 –15)

- Tillatte hjelpemidler: Alle skriftlige

Arbeidet i IN1010 - ukehjulet

Undervisningstilbud >= 6 timer



Tirsdag

Mandag

Selvstendig arbeid - programmering, 7-8 timer

Teori, Trix, Obliger, Fysisk / Digital lab

Ressurser: Se semestersiden
Bøker, notater, lysark, opptak, kollokvier, nettforum, gruppelærere, ..

Om undervisningen

- Alle info og alle ressurser lenkes fra semestersiden
  - Bli kjent med den –bruk den!

- IN1010 –uka (unntatt denne første uka)
  - –Start med forelesning på tirsdag kl 14:15 –16:00
  - –To timer seminar ledet av gruppelærer der dere jobber med det nye stoffet sammen (tid avhenger av gruppe)
    - – Alt digitalt første to uker, senere noe mer fysisk tilstedeværelse (håper vi)
    - – Vi må lage rutiner for reservering av plass på fysiske grupper
  - –Lab (en til en hjelp digitalt i første omgang) –følg med på semestersiden
  - –Fellesøvelse (plenum) mandag kl 10:15 –12:00
    - Siste gjennomgang av det nye stoffet
  - Innleveringsfrist obliger annenhver mandag kl 23:59

"IN1-Kontrakten"

- Nytt stoff presenteres på forelesningen tirsdag.
  - Lysark legges ut på forhånd eller rett etter forelesningen
  - Vanligvis legger foreleserene ut opptak så fort som mulig
  - Nytt stoff presenteres grundig, først med små enkle eksempler
  - Du MÅ (vanligvis) ikke forberede deg på nytt stoff

  MEN:
  - DET ER DITT ANSVAR AT DU KAN FORRIGE UKES STOFF !!
    - Jobb med øvingsoppgaver i Trix
    - Bruk lærebok, lysark, opptak
    - Gå på grupper, lab, . . .

Nytt nettforum (Discourse)

- Spørsmål og svar om alt du lurer på om IN1010
- Du finner lenke på kurssiden (og i heading i denne pdf'en)
- Enkel og grei brukerveiledning ligger på semestersiden

  Alle må registrere seg selv ved å gå inn på lenken og logge inn via Feide!!

- Bruk felleskanalene for spørsmål som ikke er gruppe-spesifikke
- Ikke legg ut obligkode
- Alle må bidra med hjelp og diskusjon (du *kan* være anonym)

IN1010 er pilot for dette systemet på Ifi, som vi tror vil gi bra diskusjoner og rask hjelp - om alle bidrar!

Seminartimer med gruppelærer

- Seminartimer denne uken (19.-21.januar) og antakelig neste:
  - Alle seminartimer er digitale
  - Alle møter (digitalt) på tildelt gruppe
  - Om du likevel ikke kan komme på din gruppe:
    - Gi beskjed til gruppelærer om du ønsker å beholde plass (finn gruppelærer under Grupper)
    - Når undervisning er digital, er det mulig å delta på andre/ flere grupper
- Det vil åpnes for noe fysisk undervisning –følg med på semestersiden
  - Gruppe 10 og 13 er satt opp som digitale hele semesteret

- **Følg med på semestersiden** **ifi.uio.no -> IN1010**



- Neste time: Første intro til Java for deg som kan Python
- I morgen: Java med vekt på objektorientering

Java for deg som kan Python

IN1010

17.1.2022, 2. time

Siri Moe Jensen

Oversikt

- Hvorfor Java?
- Python Java
- Kom i gang

- I morgen: Klasser ved Stein Gjessing
- Neste uke: 2 * 2 timer forelesning om Java
- Deretter (uke 3): Arv

Innhold

Introduksjon til Java

- Hvorfor flere språk
- Hva er et objektorientert språk
- Java: Pluss og minus
- Hva trenger jeg for å programmere i Java i IN1010?

Java versus Python

- Et komplett program
- Variabler, verdier og typer
- Skrive til terminal

Språk lages med ulike utgangspunkt

Java (James Gosling/ Sun Microsystems; 1996)

Mål: It must

- be simple, object-oriented, and familiar
- be robust and secure
- be architecture-neutral and portable
- execute with high performance
- interpreted, threaded, and dynamic

Python (Guido van Rossum; 1989)

Filosofi

- Beautiful is better than ugly
- Explicit is better than implicit
- Simple is better than complex
- Complex is better than complicated
- Readability counts

Sentrale anvendelsesområder

Python

- Data Science & Data Analytics
- Kunstig intelligens (AI), dyp læring
- .. og mye mer

- Stadig vanligere:
  Introduksjonskurs i programmering

Java

- Enterprise Application utvikling
- Android app-utvikling
- ...og mye mer

Det finnes ikke ett "beste språk"

Mange faktorer avgjør

- Anvendelse (domene og oppgaver)
- Eksisterende arkitektur, kode og kompetanse
- Personlige preferanser
- Trender i business (vid forstand), samfunn og teknologi
- Teknologiutvikling, utvikling av språkene

Kontinuerlig utvikling

Men:

- Java og Python er begge blant topp 3 på de fleste rangeringer av programmeringsspråk: Antall søk, antall arbeidsannonser, popularitet blant utviklere, etc.

Hvorfor **lære** flere programmeringsspråk?

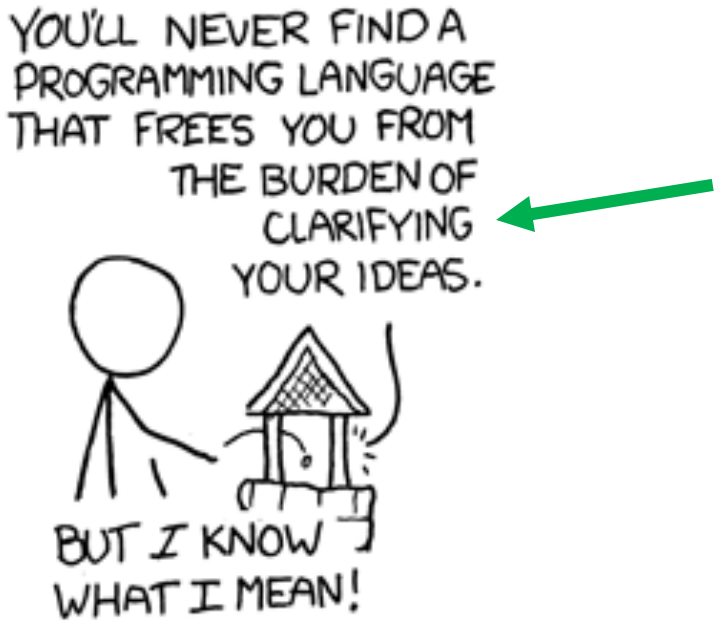
Python er et enkelt språk å komme i gang med, og relativt lett å lese

Men det betyr også at endel sentrale aspekter –og noen utfordringer - kamufleres eller unngås

I Java ser du mer direkte "hva som skjer" –og blir kjent med flere sentrale programmeringsmekanismer

Du vil høyst sannsynlig ha nytte av disse –og andre språk

- For bruk i videre studier eller jobb
- For å vite hva som er generelt –og spesielt for ett språk
- For å bli kjent med flere måter å "se" og løse oppgaver på
- For å lære språk nummer 3 og 4 enda raskere J



"Java og Python er objektorienterte språk"

- Hva betyr det?
  - ? Vi kan definere klasser og opprette objekter av disse *begge*
  - ? Vi kan bruke en objektorientert tankegang i modellering og design *begge*
  - ? Alle data er (egentlig) representert som objekter *Python*
  - ? All kode ligger i klasser, dvs vi må skrive minst én klasse *Java*

- Et Javaprogram har *all* kode inne i en eller flere klasser
  - Vi skal bruke *statiske* metoder i Java (finnes også i Python, men er ikke pensum i IN1000)
  - En statisk metode kan brukes selv om vi ikke har noen objekter av klassen –den hører ikke til ett objekt, men til klassen
  - Mer om dette i morgen

Kjøring av Java-programmer

- Du må installere Java Development Kit (JDK)
- Du trenger to programmer for å kjøre et Java-program
  - **javac** oversetter (kompilerer) .java-filen til en eller flere .class-filer
  - **java** utfører .class-filen(e)

```
$ javac Hei.java
$ java Hei
Hei!
$
```

```
class Hei {
public static void main(String[] arg) {
    System.out.println("Hei!");
}
```

Installasjon av Java: Se Tekniske ressurser og hjelp på semestersiden

Fra Python til Java

Gitt følgende Python program hvordan skriver og kjører vi (nesten) samme program i Java?

```
ant_programmer = 100
print("Dette er mitt " + str(ant_programmer) + ". Python-program")
```

Eksempel i Java (kommentarer i senere slides)

```
class Program {

    public static void main(String[] args) {

        int antallProgrammer;
        antallProgrammer = 1;

        System.out.println("Dette er mitt" + antallProgrammer + ". Java-program!");

        System.out.print("Denne linjen venter pa .println() kall eller programslutt"

    }
}
```

Innpakning

- I Java må alt ligge i klasser

- Utføring av programmet starter alltid i en metode som heter main –deklareres slik:

```
class Program {

    public static void main(String[] args) {

    }
}
```

- Filen må ha samme navn som klassen, her **Program.java**

Blokker og programsetninger

- I Java brukes krøllparenteser ({}) i stedet for innrykk.
- Alle Java-setninger avsluttes med semikolon (;).

```
if (antallProgrammer > 0) {
    System.out.println("Dette er mitt " + antallProgrammer + ". Java-program!");
}
```

- Linjeskift og innrykk betyr altså ikke noe i et Java-program:

```
        if (antallProgrammer
    > 0) {
            System.out.println
                    ("Dette er mitt " + antallProgrammer + ". Java-program!");
}
```

Angivelse av variabeltype

Python har *dynamisk typing* der typen er knyttet til verdiene ; derfor kan en variabel tilordnes verdier av en hvilken som helst type.

Java har *statisk typing* der både variablene og verdiene har type; variabler kan bare få verdier av den riktige typen.

Derfor: *I Java må alle variabler deklareres*.

```
int antallProgrammer = 1;
double pris;
pris = 3.25;
boolean ferdig = true, ok;

System.out.println("Pris = " + pris + " og ferdig = " + ferdig);
```

```
>java Program
Pris = 3.25 og ferdig = true
```

Javas viktigste typer

int	heltall (dvs 0, 255, -3, ...)
double	flyttall (dvs 0.0, 1.5, -22.873, ..)
boolean	logisk verdi (dvs **true**, **false**)
char	enkelttegn (dvs **'1'**, **'a'**, **'*'**, ...)

NB: enkel apostrof (fnutt)

Dessuten finnes klassen

String	tekst (dvs **"abcd"**, **""**, **"$50"**, **"146"**, ..)

NB: dobbel apostrof

if-tester kan ha flere alternativer

```
if (antallProgrammer== 0) {
    System.out.println("Dette er mitt null'te Java-program!");
} else if (antallProgrammer == 1) {
    System.out.println("Dette er mitt forste Java-program!");
} else {
    System.out.println("Dette er mitt " + antallProgrammer + "Java-program!");
}
```

Legg merke til:

- Java har parenteser rundt if-testen (i stedet for kolon etter)
- Java har ingen **elif** — i stedet skriver vi **else if**

Løkker

While-løkker

```
while (antallProgrammer < 10000) {
    // Fortsett å programmere i Java
    antallProgrammer += 1;
}
```

For-løkker med teller (erstatter ... in range(10000) i Python)

```
for (int i = 0; i<10000; i++) {
    // Skriv ett program til i Java
}
```

Utskrift

I java finnes to metoder for utskrift:

- System.out.print skriver ut parameteren (kun en parameter!)
  - Hvis det er flere elementer skjøtes de sammen med +
  - Eller man kan kalle System.out.print flere ganger
- System.out.println gjør det samme, men legger til linjeskift.




```
System.out.print("Navn: " + "Donald Duck");
System.out.println();
```

```
System.out.println("Navn: " + "Donald Duck");
```

Hvor mange feil finner du?

Feil.java

```
class Feil {
    public static void main (String[] args) {
        pris = 25;
        pris = 2*pris
        if pris > 200 {
            pris = 200;
        }
        else:
            pris = pris + 50;
        print("Pris: " + pris);
    }
}
```

Et komplett lite eksempel

FeilUtenFeil.java

```
class FeilUtenFeil {
    public static void main (String[] args) {
        int pris = 25;
        pris = 2*pris;
        if (pris > 200) {
            pris = 200;
        }
        else {
            pris = pris + 50;
        }
        System.out.println("Pris: " + pris);
    }
}
```

$ javac FeilUtenFeil.java

$ java FeilUtenFeil

Komme i gang

- Viktige lenker fra semestersiden (under Tekniske ressurser og hjelp og Uke1):
  - Installasjon og oppsett av Java
  - Oppkobling til og arbeid på Ifis servere
  - Java for menneskelige lesere (inkl deg selv) –Kodestil i Java

- Mer om programmering i Java denne uken og neste
  - Klasser og objektorientert programmering i morgen
  - Arrayer, metoder, input fra terminal og filhåndtering neste mandag
  - Mer om datastrukturer og representasjon neste tirsdag
  - Trening og flere detaljer i seminar og oppgaver inkl *Innlevering 0* (frist 24.1)!

Oppsummering

- Det finnes ikke noe beste språk for alle formål!
- Det er en styrke å kunne flere språk
- Alle informatikere må regne med å lære flere språk
- Flere språk gir dypere forståelse

- Det er egentlig bare én måte å lære et nytt programmeringsspråk på:
  - les dokumentasjonen (lærebok, notater, Java 8 API)
  - prøv selv med mange eksempler (evt i pythontutor –velg Java som språk)
  - om ikke alt fungerer som forventet, gjenta

IN1010 - våren 2022

Tirsdag 18. januar

Java
Verdier og variabler
Objekter, klasser og referanser
static

1

Stein Gjessing

Agenda

Rask intro til enkel innlesing og utskrift i terminalvindu (2 lysark)

- Mer i gruppene og neste mandag

Verdier og variabler i datamaskinen og i Java

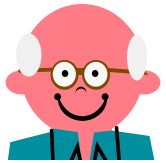
Objekter og klasser i Java

Lesing og skriving i terminalvinduet



```java
import java.util.Scanner;

class LesFraTerminal {
   public static void main (String [ ] args) {
      int alder;
      String navn, adresse;
      Scanner  minInn =  new Scanner (System.in);
      System.out.print(" Skriv adressen din: ");
      adresse = minInn.nextLine();
      System.out.print(" Skriv fornavnet ditt: ");
      navn = minInn.next();
      System.out.print(" Skriv alder: ");
      alder = minInn.nextInt();
      System.out.println( navn + ", du bor i " +
            adresse + " og er " + alder + " år" );
    }
}
```



Men pass på.
F.eks.
next() og
nextInt()
leser ikke
hele linjen

Mer i gruppene og mandag 24. januar
Midlertidig forklaring neste side

Midlertidig forklaring av noen detaljer

java.util.Scanner — klassen Scanner er definert i biblioteket util

new Scanner() — lager et nytt Scanner-objekt

Scanner minInn = new Scanner (System.in); — lager et nytt Scanner-objekt som leser fra terminalvindu OG setter variablen minInn til å referere dette objektet

adresse = minInn.nextLine(); — returnerer en referanse til en hel linje fra minInn (mer senere)

navn = minInn.next(); — returnerer en referanse til en tekst fra minInn (mer senere)

alder = minInn.nextInt(); — returnerer et heltall fra minInn (mer senere)

Google: Java API 8 Scanner

senere = neste mandag

IN1010: Objektorientert programmering

- Hva er et objekt ?
- Hva er en klasse ?

Men først: Hva er en variabel (i Java) ?

```
int antall;
int flere;

antall = 6;
flere = 4;
antall = antall + flere;
```

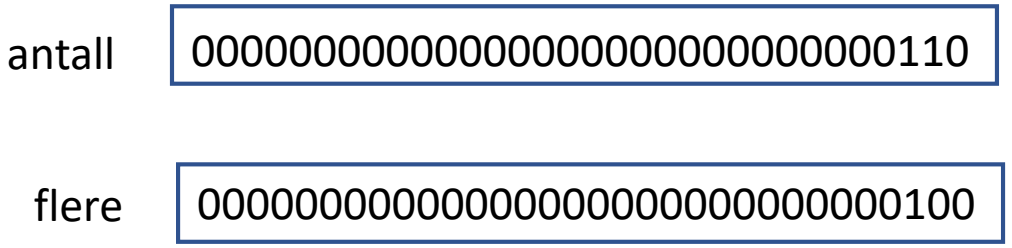
eller

```
int antall = 6;
int flere = 4;
antall = antall + flere;
```

En variabel i Java er

- Noen bit/byte i minnet som kan innholde en verdi
- Det stedet i minnet der disse bittene ligger har et navn, f.eks. antall eller flere

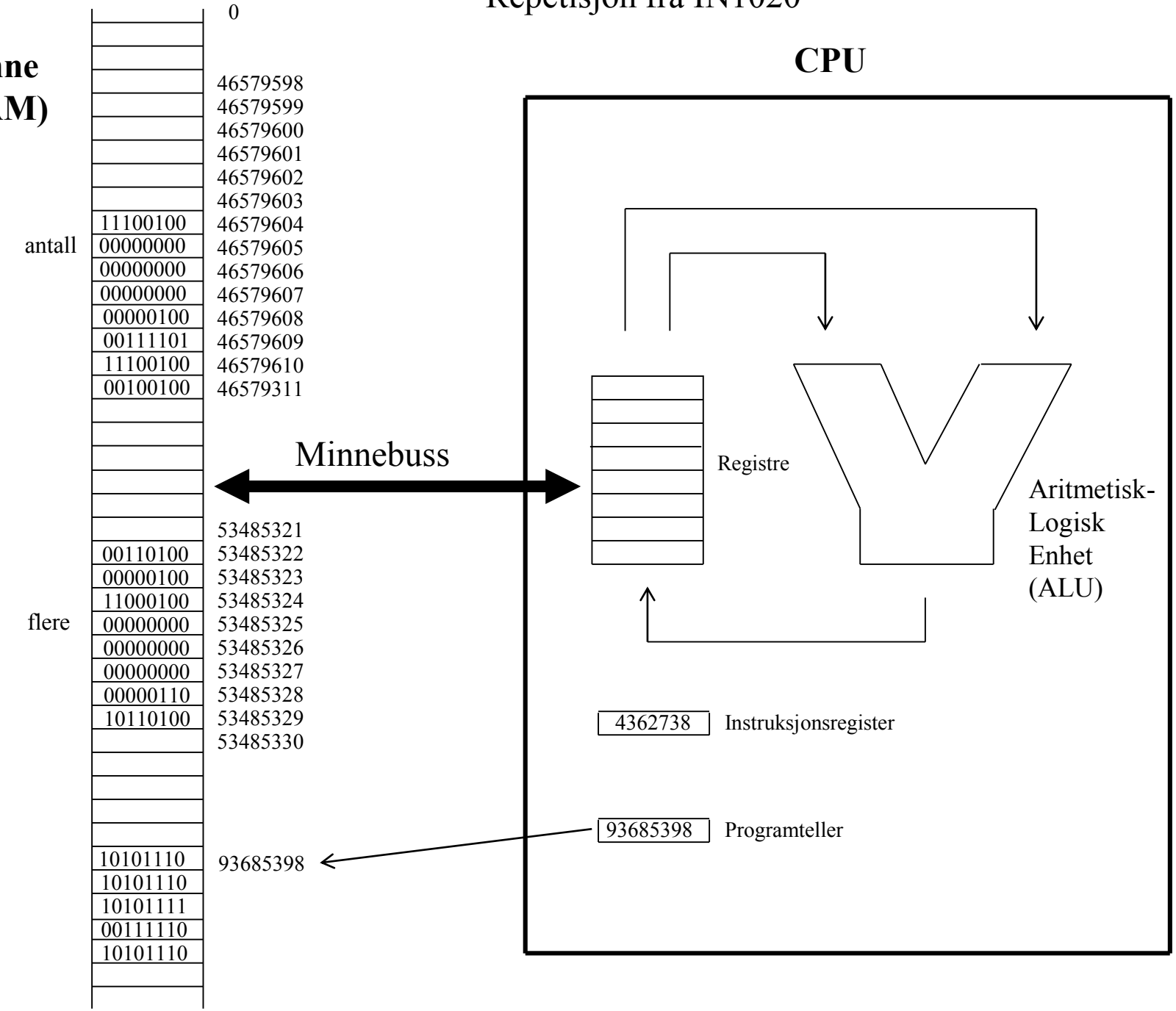
```
int antall = 6;
int flere = 4;
antall = antall + flere;
```



Repetisjon fra IN1020

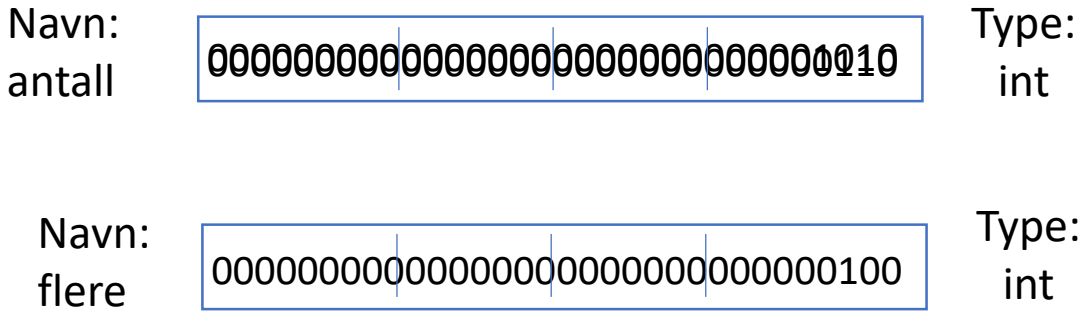
Minne (RAM)

Minne adresseres i byte (1 byte = 8 bit)



Variabler i Java

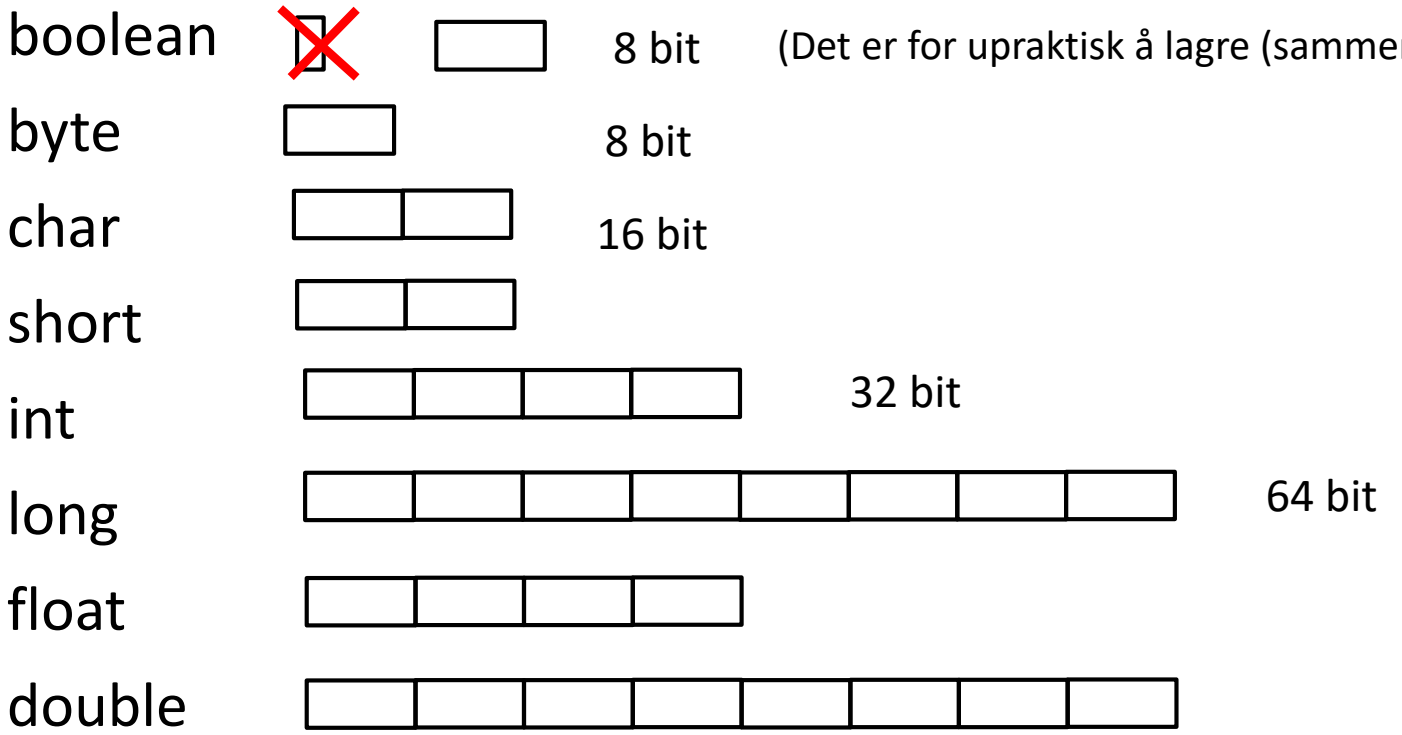

```
int antall = 6;
int flere = 4;
antall = antall + flere;
```



En variabel i Java er et **navngitt** sted i minnet som inneholder en **verdi** av en gitt **type**
Typen til en variabel kan ikke forandres (og heller ikker navnet kan forandres).
Innholdte til en variable forandres ved **tilordning**

Javas verdier

- 8 **primitive typer**: boolean, byte, char, short, int, long, float, double.

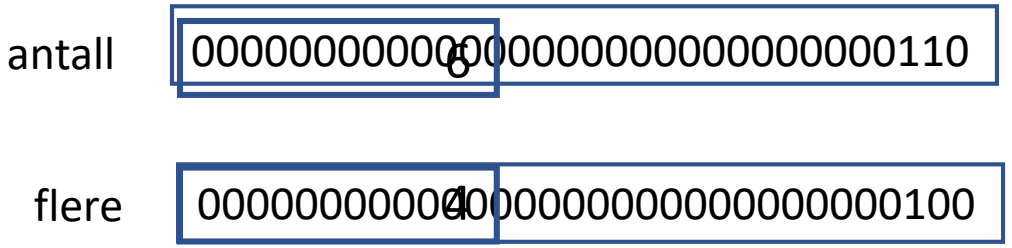


Det er disse verdiene som kan ligge i en variabel

Slik tegner vi en variabel i Java

- Noen bit/byte i minnet som kan innholde en verdi
- Det stedet i minnet der disse bittene ligger har et navn, f.eks. antall eller flere

```
int antall = 6;
int flere = 4;
antall = antall + flere;
```



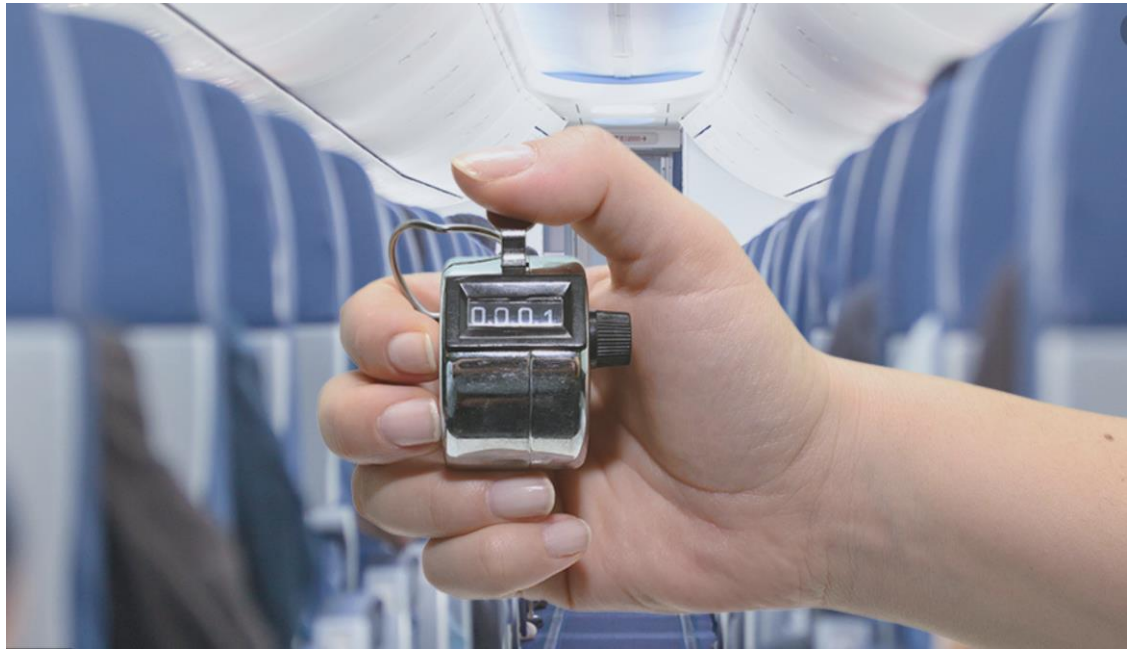
Nå, over til

Objektorientert programmering

- Hva er et objekt ?
- Hva er en klasse ?

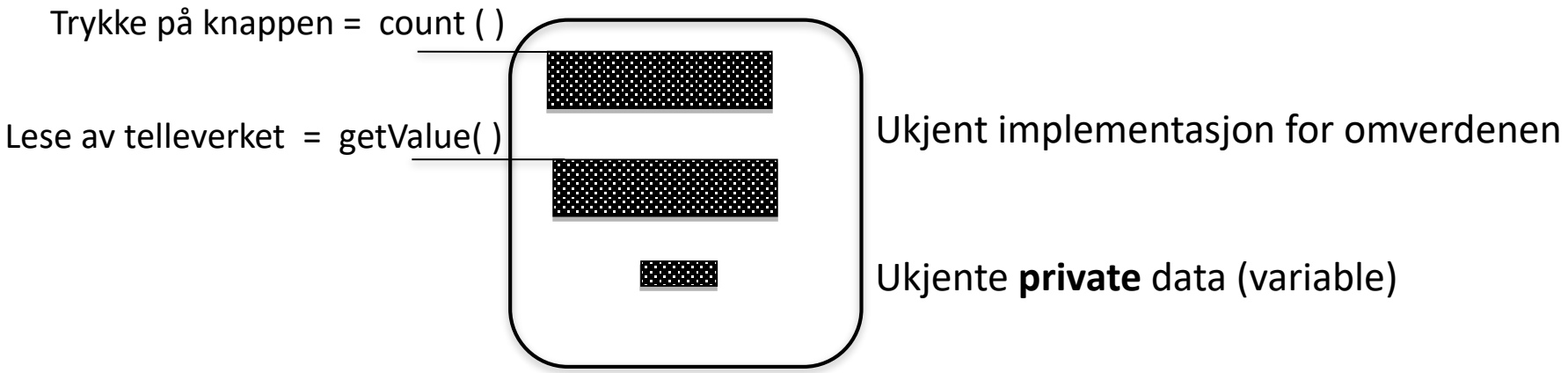
Objekter i Java: En teller (engelsk: counter)

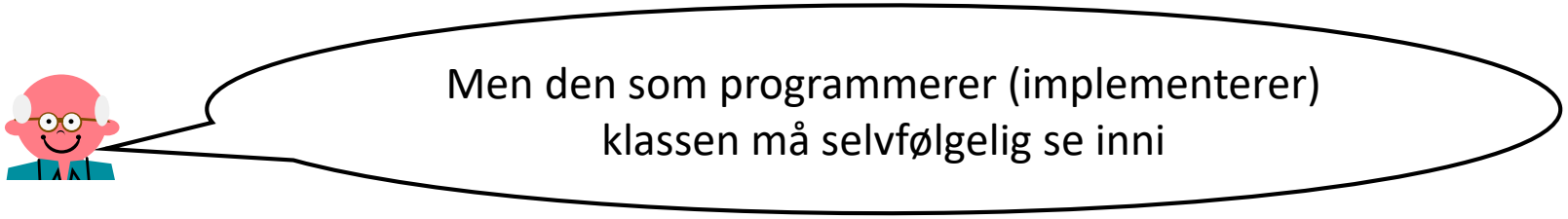
- Aller enkleste eksempel (Horstmann kap 8.2):
  - En teller (som f.eks. betjeningen på et fly bruker)
    - Tell én opp
    - Les av telleren
  - Starter på null
- Vi skal programmere en slik



Objekter i den virkelige verden modelleres av programmet inne i datamaskinen

Et objekt er en sort boks + grensesnittet mot omverdenen





En teller

Hvordan skal count() og getValue() virke

- Senere i IN1010 blir det fort mer kompliserte grensesnitt
  - Og vi skal da snakke mer om virkemåten (semantikken) til et grensesnitt.
- Men her er (skrivemåten til) grensesnittet opplagt:

```
public void count ( )
public int    getValue ( )
```

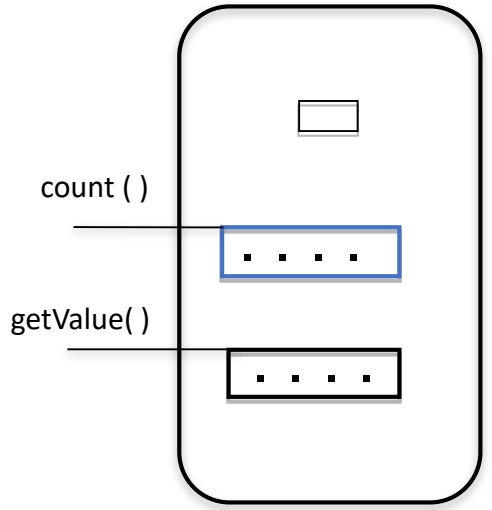
Hvordan skal count() og getValue() programmeres OG

Hvilke private data må være inne i et slikt teller-objekt?

Da må vi se for oss et teller-objekt

- Er det nok med èn variabel inne i objektet?

- Er det nok at
- count() teller opp denne verdien med èn og at
- getValue() leser av denne verdien ?

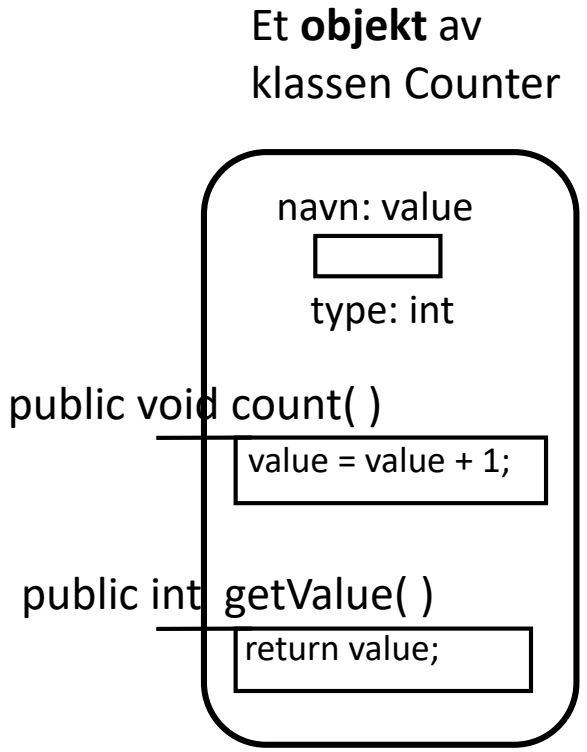




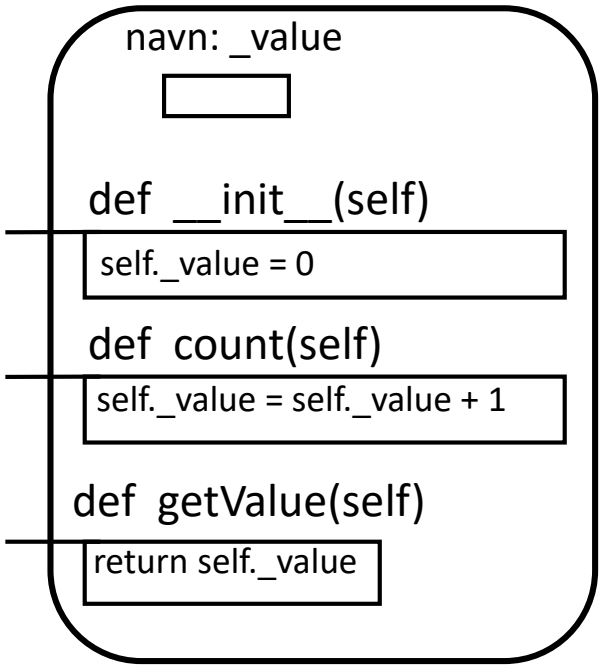
Det er dette som er å programmere !!
Men vanligvis er problemet vanskeligere !!

Et forslag til et teller-objekt

- Generelt innholder objekter
  - **Metoder –operasjoner - handlinger**
    - public (som regel)
    - men også private metoder
      - til bruk inne i objektet
  - **Variabler og konstanter** - “ D A T A ”
    - av de primitive typene eller referanser
    - som regel skjult for omverdenen –private (innkapsling)



Python



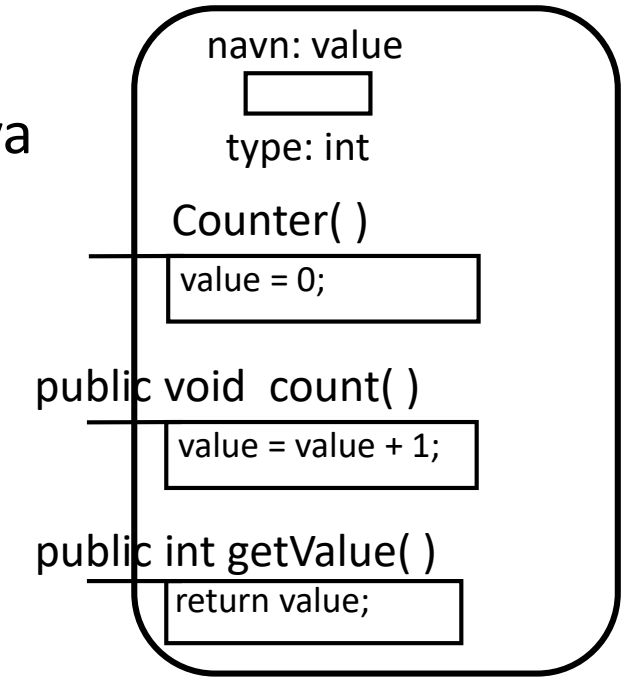

```
class Counter:

   def __init__(self):
     self. _value = 0

   def count(self):
      self._value = self. _value + 1

   def getValue(self):
      return self._value
```

Java

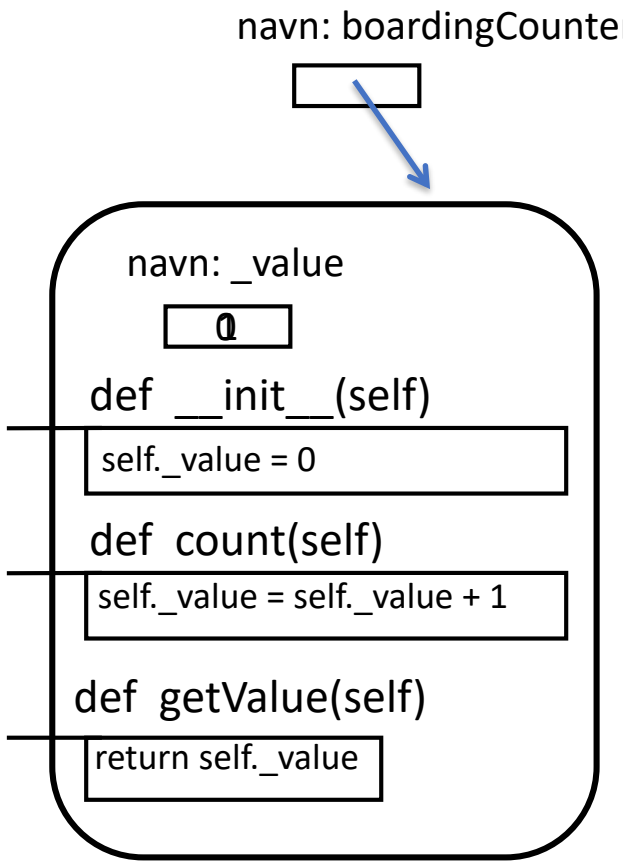



```
class Counter {
    private int value;
    public Counter() {
        value = 0;
    }
    public void count( ) {
        value = value + 1;
    }
    public int getValue( ) {
        return value;
    }
}
```

Neste skritt

1. Lage et pogram som deklarerer en Counter-klasse
2. La programmet opprette et Counter-objekt
3. Øk telleren i Counter-objektet med én
4. Les av telleren i Counter-objektet og lagre resultatet i en variabel i programmet

Python




```
boardingCounter = Counter ( );

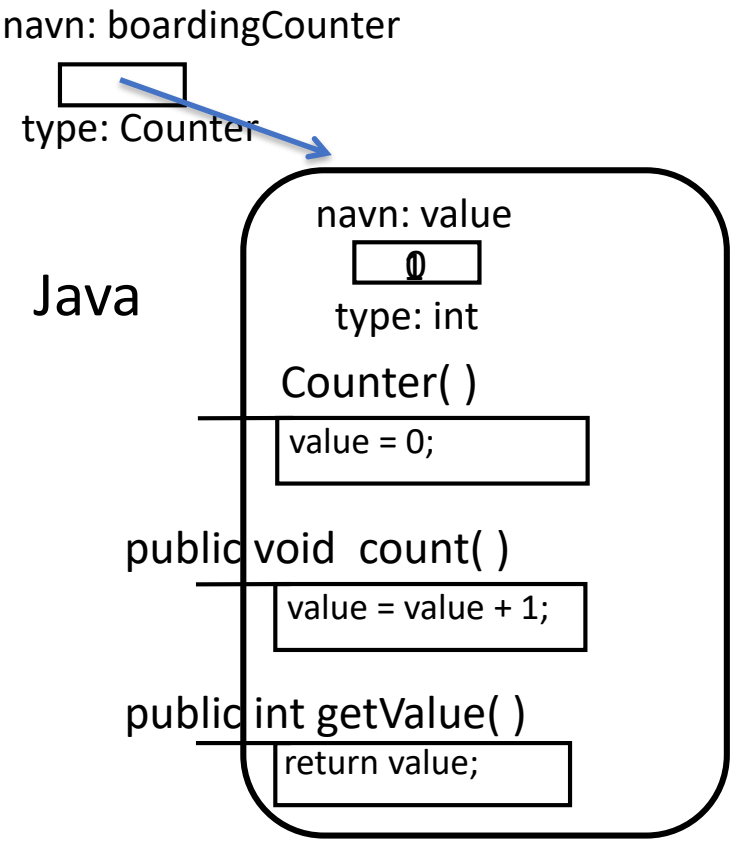
boardingCounter.count( );

tall = boardingCounter.getValue( );
```

navn: tall

1

Java




```
Counter boardingCounter = new Counter ( );

boardingCounter.count( );

int tall = boardingCounter.getValue( );
```

navn: tall

1

type: int

Nytt tema: Java og «static»

- En klasse er et mønster for å lage objekter

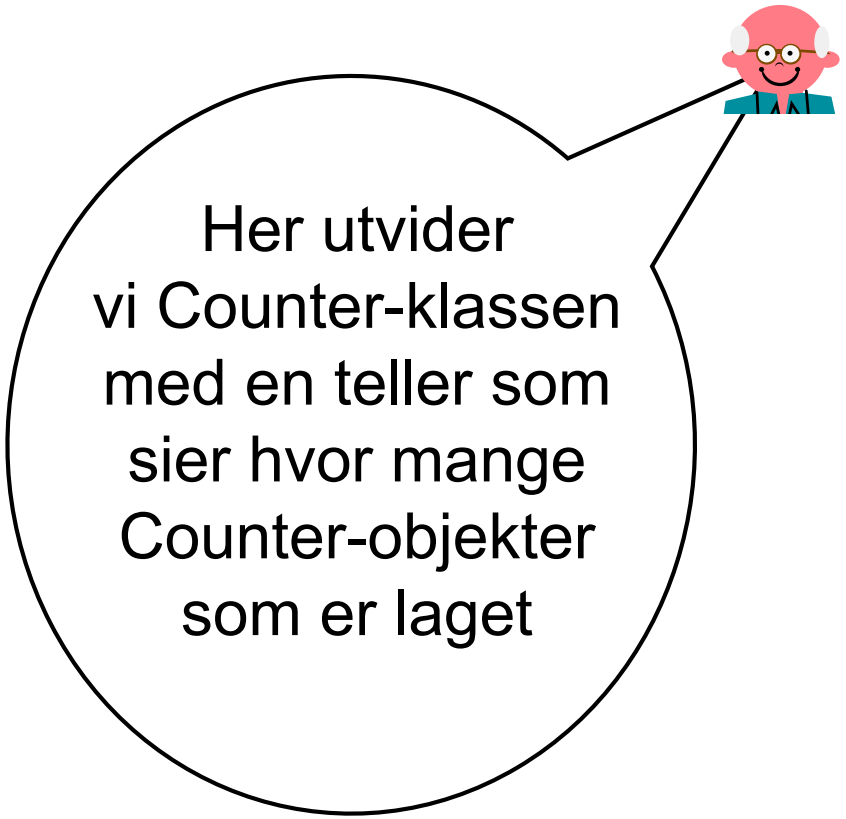
PLUSS:

- I Java kan vi deklarere egenskaper inne i en klasse som bare finnes èn gang
- Følgelig: Disse egenskapen blir IKKE gjentatt inne i hvert objekt
- Brukes når vi trenger felles egenskaper for ALLE objekter av samme klasse

- Ikke veldig viktig i IN1010, men noe vi bare må kunne

Java: **static**-egenskaper

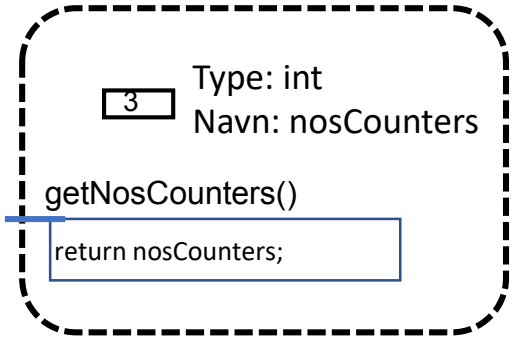

```
class Counter {
    private static int nosCounters =0;
    private int value;
    public Counter( ) {
        value = 0;
        nosCounters ++;
    }
    public static int getNosCounters ( ) {
            return nosCounters;
    }
    public void count( ) {
        value = value + 1;
    }
    public int getValue( ) {
        return value;
    }
}
```



static-egenskaper

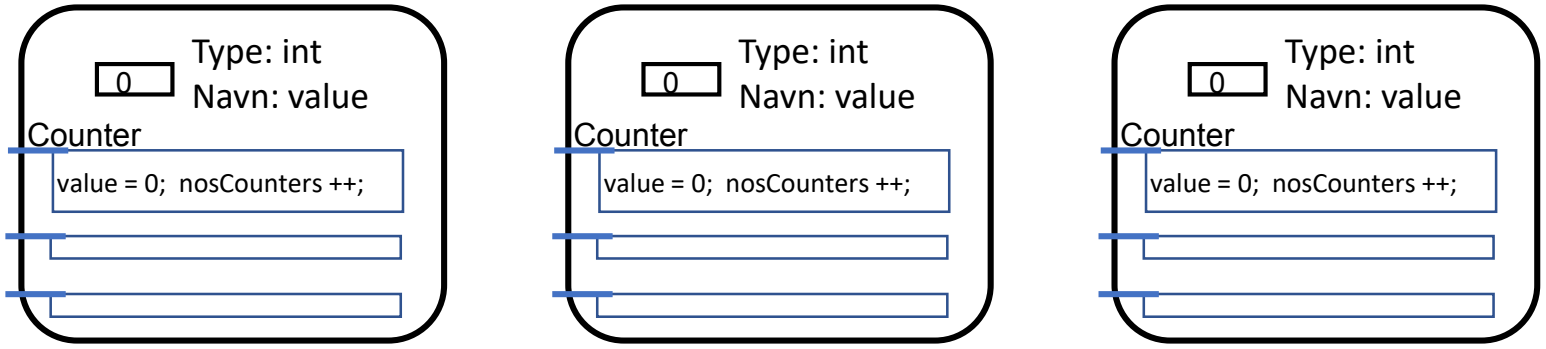
```
class Counter {
    private static int nosCounters =0;
    private int value;
    public Counter() {
        value = 0;
        nosCounters ++;
    }
    public static int getNosCounters ( ) {
        return nosCounters;
    }
    public void count( ) {
        value = value + 1;
    }
    public int getValue( ) {
        return value;
    }
}
```

Når programmet starter lages en *klasse-datastruktur* av "static-egenskapene" til alle klassene i programmet



De tegningene vi har sett hittil av variable, metoder, objekter og klassedatastrukturer kaller vi til sammen Java **datastrukturer**

Etter f.eks. 3 kall på new Counter() har vi 3 objekter (*objekt-datastruktur*):



Programmering er å løse problemer ved hjelp av datastrukturer

- Når vi skal løse et problem må vi tenke oss en **datastruktur** som løser problemet
- Selve løsningen av problemet er manipulering av denne datastrukturen (en **algoritme**)
- Når du skal løse et problem:
  - Tenk deg og eventuelt tegn først datastrukturen
  - Deretter kan du skrive koden (algoritmen) som lager og manipulerer datastrukturen og løser problemet

Hvor nøyaktig skal jeg tegne Java datastrukturer ?

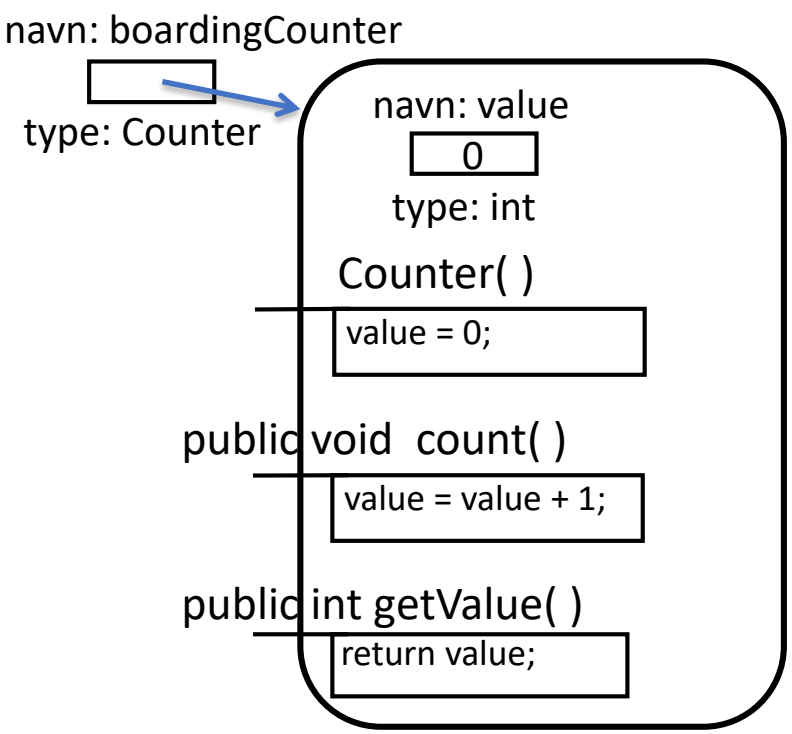
- Svar:

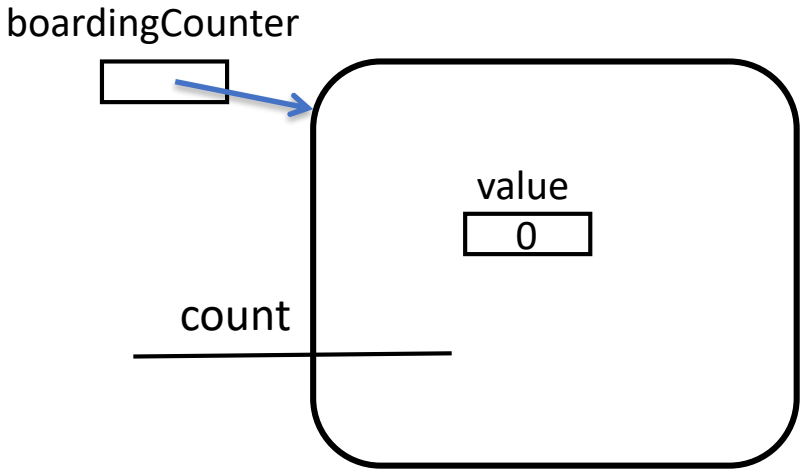
- Så nøyaktig som det er nødvendig for at du selv eller dem du samarbeider med skal skjønne hva som skjer med datastrukturen når programmet (algoritmen) utføres

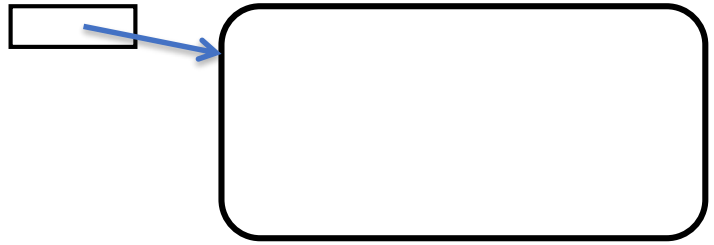
- Du må gjerne tegne det på en annen måte enn slik vi gjør i IN1010, men da er det ikke sikkert vi andre skjønner deg

Eksempel fra teller-programmet

Tre måter å tegne det samme objektet på:



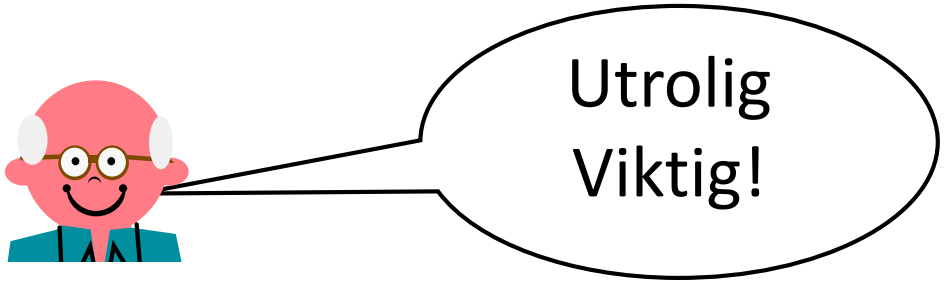




Tegn så mange detaljer at du selv skjønner hvordan objektene og variablene er/virker, og slik at du kan forklare hvordan algoritmen virker både for deg selv og for dem du samarbeidet med.
I et større program vil vi nok ha én tegning med grensesnitt og en annen med implementasjon.
Men referanser mellom objekter bør (nesten) alltid med.

Vi skiller mellom

- Klasse-deklarasjonen i programteksten. Den er et mønster som brukes både når klassedatastrukturen lages (i det programmet starter opp) og senere når nye objekter lages.
- Klasse-datastrukturen, dvs. den (statiske) datastrukturen som lages i det programmet starter.
- Objekt-datastrukturen (også kalt klasse-instanser, klasse-objekter eller bare objekter) som lages hver gang vi sier new.



Et litt større eksempel (hvis tid, ellers bare til egetstudium)

Hensikten med dette eksemplet er at dere skal lære Java.

Hvordan ser et Java-program ut?

Hva skjer når et Java-program kjører?

Et litt større eksempel

- Du har en venn som er bruktbilselger, og du skal hjelpe ham med å lage et program for å holde orden på hvor mange som er interessert i de enkelte bilene han har til salgs.
- Først tegner du to bil-objekter, så lager du et program som lager og bruker disse to bil-objektene.
  Dette programmet skal du senere utvide . . .
- Etter at du først tegnet datastrukturen og så programmerte en stund kom du fram til programmet på neste side
- Hvordan tenkte du?
- Hvordan virker dette programmet?

Program for salg av biler.


```
public class BilSalg{
  public static void main (String [ ] args) {
     int antallStein;
     Bil steinsT = new Bil ("Stein");
     Bil sirisO = new Bil ("Siri");
     steinsT.foresporsel ( );
     sirisO.foresporsel ( );
     steinsT.foresporsel ( );
     antallStein = steinsT.finnAntForesp();
     System.out.println ("Antall forespørsler på
          "Steins antallSteiner" +
          antallStein);
     System.out.println ("Antall forespørsler på
          "er BilfinnTotal( ) +
          "totalt" +
  }
}
```

```
class Bil {
   private static int total = 0;
   private String eier;
   private int antForesporsler = 0;

   public Bil (String navn) {
        eier = navn;
   }
   public static int finnTotal ( ) {
        return total;
    }
   public void foresporsel ( ) {
        antForesporsler ++;
        total ++;
    }
   public int finnAntForesp ( ) {
         return antForesporsler;
    }
}
```

BilSalg klassedatastruktur

public static void main (. . .)

navn: steinsT

type: Bil

navn: sirisO

type: Bil

navn: antallStein

type: int

2

```
int antallStein;
Bil steinsT  = new
Bil sirisO   = new
steinsT.foresporsel ( );
sirisO.foresporsel ( );
steinsT.foresporsel ( );
antallStein = steinsT.finnAntForesp();
System.out.println
    antallStein);
System.out.println
```

Stein

navn: eier
Type: String

navn: antForesporsler

1

type: int

void foresporsel( )

```
antForesporsler ++;
total ++;
```

int finnAntForesp( )

```
return (antForesporsler);
```

Bil(String navn)

```
eier = navn;
```

Bil-objekt

Bil klassedatastruktur

Navn: total

0

Type: int

finnTotal

```
return total;
```

Siri

navn: eier
Type: String

navn: antForesporsler

0

type: int

void foresporsel( )

```
antForesporsler ++;
total ++;
```

int finnAntForesp( )

```
return (antForesporsler);
```

Bil(String navn)

```
eier = navn;
```

Bil-objekt

Antall forespørsler på Steins Toyota er 2

Antall forespørsler totalt er nå 3

I dag har du lært

- Hva en verdi er
- Hva en variabel er
- Hva et objekt er
- Hva en klasse er
- Hva en referanse (= peker) er
- Hvordan vi visualiserer / tegner variabler, metoder og objekter
- Hva "static" betyr i Java og hvordan vi visualiserer / tegner klassedatastrukturer

- At å programmere er å finne på en datastruktur og lage et program som oppretter og manipulerer denne datastrukturen

Fra Python til Java

Uke 1 – dag 3 (IN1010-tidsregning)

Mandag 24. januar 2022

Beskjeder

Obliger

- Husk innlevering 0 (frist i kveld, nyttig å løse uansett levering)

Seminartimene (alle grupper) er digitale kommende uke

- du kan møte på all digital undervisning, men følg din gruppe om mulig
- informasjon om muligheter for å bytte til (annen) fysisk gruppe kommer ila uken (følg med på Beskjeder)

Læringsmål

Forrige uke og denne:
- Skrive og bruke klasser i Java
- Vite hvordan et Java-program representeres og utføres
  - fra programkode
  - fra øyeblikksbilder av datastrukturen

I dag
- Finne dokumentasjon på klasser og metoder i Java API

- Kunne lese og skrive til/ fra fil og terminal
- Manipulere og konvertere strenger
- Bruke arrayer og for-løkker

Oversikt

- Mentitest: Programrepresentasjon og –utførelse
- Lese inn data i Java –med diverse tilbehør
  - Java API (klassebiblioteket), packages, import og aksessmodifikatorer
  - Exceptions (unntakshåndtering)
  - Klassene File og Scanner
  - Lese ulike typer verdier
- Skrive til fil (PrintWriter)
- Tekststrenger (String), manipulering og konvertering
- Arrayer og for-løkker

Mentimeter

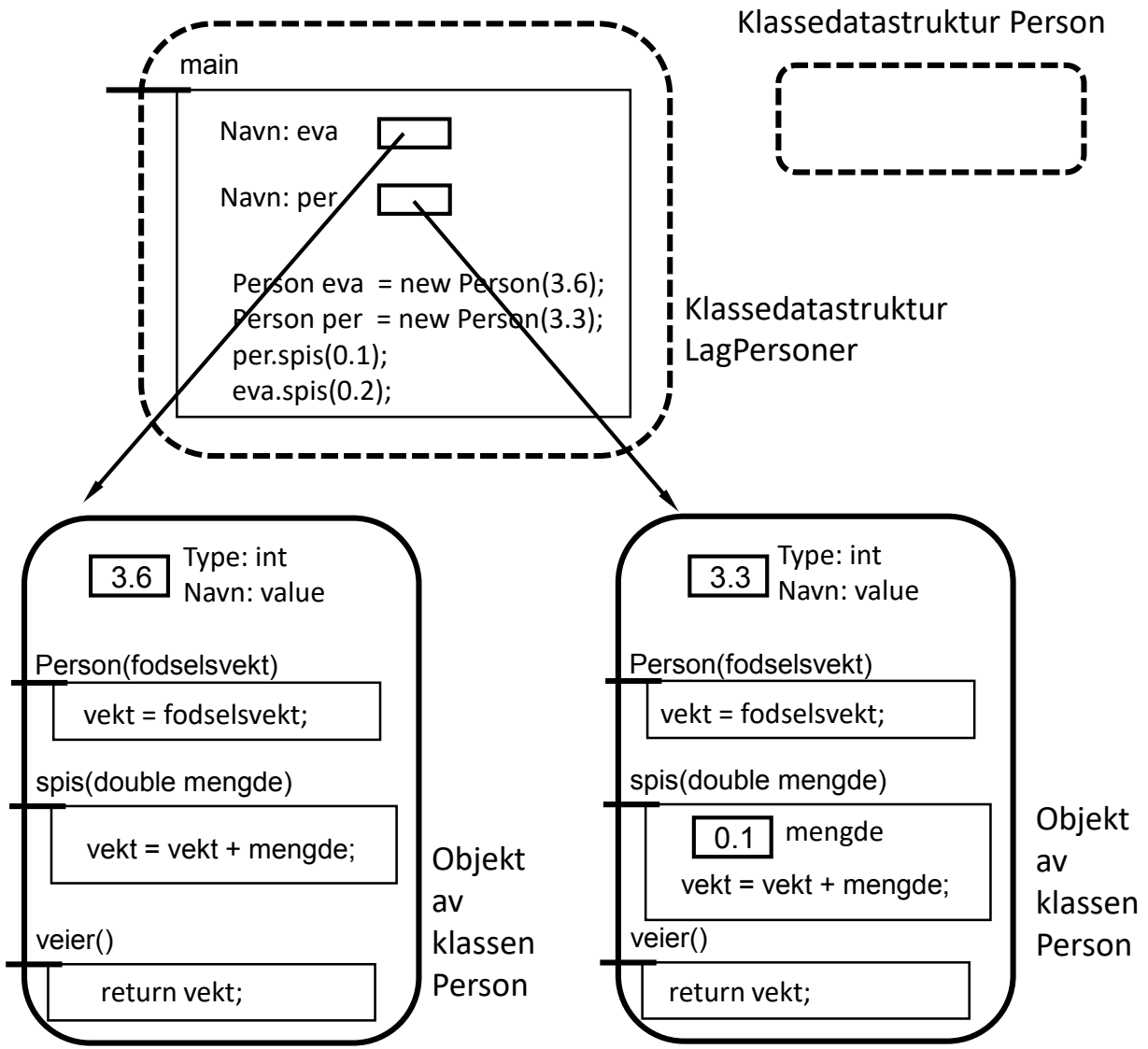
Gå til menti.com, bruk kode 4762 7793

Gå til menti.com, bruk kode 4762 7793

Øyeblikksbilde (tilstand) når Per er i ferd med å spise 100g

```
1  class Person {
2      private double vekt;
3      public Person(double fodselsvekt) {
4        vekt = fodselsvekt;
5      }
6      public void spis(double mengde) {
7              vekt = vekt + mengde;
8      }
9      public double veier() {
10       return vekt;
11     }
12 }
13
14 class LagPersoner {
15     public static void main(String[ ] arg) {
16        Person eva  = new Person(3.6);
17        Person per  = new Person(3.3);
18        per.spis(0.1);
19        eva.spis(0.2);
20     }
21 }
```

Har Per fått maten inn i munnen?
(Hvilken setning er neste til å bli utført?)



Java Development Kit

Innhold og bruk

Bli bedre kjent med Java

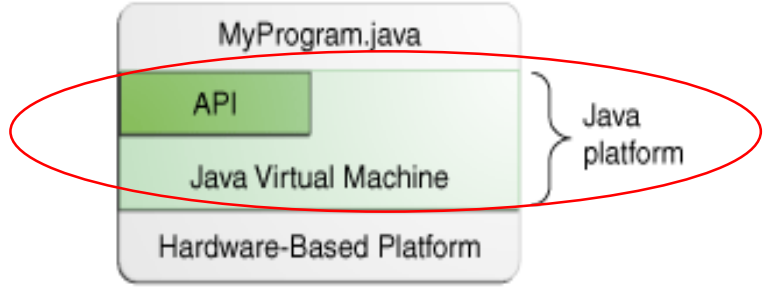
- Python og Java har ulike underliggende modeller og implementasjon, og noe ulik terminologi

- I IN1000 gikk vi ikke veldig dypt i "hva skjer under overflaten"
  - ikke nødvendig for å bruke mekanismene i IN1000 pensum
  - IN1000 pensum inkluderte ikke nødvendige konsepter
  - ville gjort overgangen til Java tyngre

- Men:
  - IN1010 pensumet gir grunnlag for, og krever, dypere forståelse
  - Dere skal bruke mange klasser i Java biblioteket (gjennom Java API)
  - Lær å bruke Java API dokumentasjon allerede nå

Java ~ Java Development Kit (JDK)

- JDK (Standard Edition –SE) inneholder alt du trenger for utvikling og kjøring av Java programmer:
  - Java språket/ kompilator
  - Java API (Application Programming Interface)/ klassebibliotekene
  - Java Runtime Environment (JRE) med Java Virtual Machine (JVM)
  - ++



- Stadig nye versjon av JDK. IN1010 versjon = subsett av Java 8 (=1.8)
  - Java 8 har alt vi trenger og bruker i IN1010
  - Kjører i alle nyere versjoner dvs på alle platformer
  - Fokus på grunnleggende mekanismer for videre påbygging
  - Lettere å gå til nyere versjoner enn motsatt vei

Hvordan finne svar - dokumentasjon

- Java 8 API (søk etter f eks "Java 8 API String" eller ".. Scanner")
  Dokumentasjon på klasser og metoder i Java
  - Hvilken pakke en klasse ligger i (for import)
  - Dokumentasjon på konstruktører og metoder:
    - hva gjør de, argumenter og returverdier

- Java Tutorials om du ønsker å gå "til kilden" for de store (og små) sammenhengene i språket Java

- Big Java inneholder alt dere trenger å vite om dagens temaer (og endel mer) –og gir stort sett gode forklaringer på riktig nivå

Lese fra fil

Se notatet: "Lesing og skriving i Java" på semestersiden

Navnestatistikk i Python

Hva er navnet på eldste person i en fil?

alder.txt

```
Petter 92
Kari 92
Emil 101
Katrine 99
```

finn_eldste_navn.py

```
fil = open("alder.txt", "r")
eldste_navn = "ingen"
maks_alder = 0
linje = fil.readline()
while linje != "":
    biter = linje.split()
    navn = biter[0]
    alder = int(biter[1])
    if alder > maks_alder:
        maks_alder = alder
        eldste_navn = navn
    linje = fil.readline()

print(eldste_navn)
fil.close()
```

finn_eldste_navn.py

```
fil = open("alder.txt", "r")
eldste_navn = "ingen"
maks_alder = 0
linje = fil.readline()
while linje != "":
    biter = linje.split()
    navn = biter[0]
    alder = int(biter[1])
    if alder > maks_alder:
        maks_alder = alder
        eldste_navn = navn
    linje = fil.readline()

print(eldste_navn)
fil.close()
```

FinnEldsteNavn. java

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;

class FinnEldsteNavn {
    public static void main(String[] args)
    throws FileNotFoundException {
        File fil = new File("alder.txt");
        Scanner sc = new Scanner(fil);
        String eldsteNavn = "ingen";
        int maksAlder = 0;
        while (sc.hasNextLine()) {
            String[] biter = sc.nextLine().split(" ");
            String navn = biter[0];     // Virker for alle navn?
            int alder = Integer.parseInt(biter[1]);
            if (alder > maksAlder) {
                maksAlder = alder;
                eldsteNavn = navn;
            }
        }
        System.out.println(eldsteNavn);
        sc.close();
    }
}
```

FinnEldsteNavn. java

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;

class FinnEldsteNavn {
    public static void main(String[] args)
    throws FileNotFoundException {
        File fil = new File("alder.txt");
        Scanner sc = new Scanner(fil);
        String eldsteNavn = "ingen";
        int maksAlder = -1;
        while (sc.hasNextLine()) {
            String[] biter = sc.nextLine().split(" ");
            String navn = biter[0];     // Virker for alle navn?
            int alder = Integer.parseInt(biter[1]);
            if (alder > maksAlder) {
                maksAlder = alder;
                eldsteNavn = navn;
            }
        }
        System.out.println(eldsteNavn);
        sc.close();
    }
}
```

Bedre: try –catch, se notat

Kaller .close() når Scanner-objektet er en fil

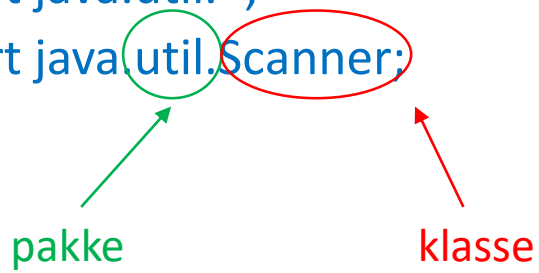
import: Hva skal programmet bruke?

Pakker og klasser, import

=> trenger vanligvis ikke tenke på dette

- All Java-kode hører til en klasse, som igjen hører til i en pakke (package)
  - Klasser uten pakke legges i en *unnamed* pakke
  - Disse er tilgjengelige for programmer i samme mappe (hvis filnavn = klassenavn)
- Pakken java.lang importeres alltid automatisk (her ligger for eksempel String-klassen)

- Andre pakker (eller enkeltklasser i pakker) må importeres eksplisitt
  - import java.util.*;
  - import java.util.Scanner;



Hvor ligger Scanner? Søk "java api 8 Scanner"

Tilgangskontroll: *Hvor* kan dette aksesseres?

Tilgangskontroll

- Alle klasser i programmet ditt ligger i samme mappe => de er i samme pakke

- Om du ikke angir tilgangsnivå er alt *package-private* (default tilgang)
  - Alle dine programmer i denne mappen har full tilgang til alt
  - Men: **main** må være **public** (fordi kjøresystemet til Java skal finne den)

- Innhold i klassen som du ønsker å beskytte (data og hjelpemetoder)
  - angis som **private** (kan bare aksesseres i klassen)

Innkapsling

- Mer å ta hensyn til og flere muligheter om programmene dine skal brukes av andre, eller gjenbrukes senere av deg selv –og når vi begynner å bruke arv

Feilhåndtering –"unntak"

Feil som oppstår under kjøring genererer unntak (exceptions) i Java (og i Python!)

Noen unntak *krever* Java at vi håndterer i programmene våre –blant annet ved åpning av filer

- krever import av det aktuelle unntaket
- unntak kan fanges og håndteres der det oppstår (med **`catch`**)
- .. eller "kastes" (med **`throws`**) fra metoden tilbake til kallstedet
- .. hver metode opp til og med main må da kaste videre *eller* fange unntaket

Beste håndtering:

- fange unntaket der vi vet mest mulig om hva som gikk galt (med `try ... catch`)
- gi en tilpasset feilmelding

Eksempler på `try ... catch` i notatet "Enkel lesing og skriving i Java" på semestersiden.

"Alt" om unntakshåndtering i Big Java, 7.4.

"Wrapper" klasser for primitive typer

Integer, Float, Boolean, Character, ++

Mer overhead, mer funksjonalitet om verdiene ligger i objekter med metoder

+ Noen ganger trenger vi at en verdi er et objekt (eksempler litt senere)

+ Disse klassene kan brukes som verktøykasser - med statiske metoder og konstanter som kan brukes uten å opprette et objekt av klassen. Eksempler fra **class Integer**:

  • Metoden **`parseInt`**

```
int alder = Integer.parseInt(biter[1]);          // konverterer String til int
```

  • Konstanten **`MAX_VALUE`**

```
int max = Integer.MAX_VALUE;                    //  høyeste verdi for integer
```

String

Konvertering til wrapper-klassens type

- Klassene Integer, Double, Boolean, Char, .. har metoder for å hente verdier av sine respektive typer fra en String

```
minStreng = " 1243 ";
minStreng = minStreng.trim();
tall = Integer.parseInt(minStreng);
```

- NB: argumentet må *kun* inneholde verdien som skal konverteres. Whitespace kan fjernes med String-metoden trim()

Konvertering til String

- Klassene Integer, Double, Boolean, Char, .. har alle en metode toString som tar et argument av respektiv type og returnerer den som en String

```
int tall = 1234;
String minStreng = Integer.toString(tall);

double flyttall = 12.34;
String minStreng2 = Double.toString(flyttall);

System.out.println("minStreng = " + minStreng);
System.out.println("minStreng2 = " + minStreng2);
```

- Merk at vi ikke bruker wrapper-klassene til å representere verdiene her –vi utnytter bare de statiske metodene i klassene til å konvertere primitive typer

Konvertering til String på ulike måter

Legge til en tom streng

```
double tall = 5.2;
String s = "" + tall;
```

Bruke Double klassens (statiske) metode toString

```
double tall = 5.2;
String s = Double.toString(tall);
```

Bruke String-klassens (statiske) metode valueOf

```
int heltall = 52;
String s1 = String.valueOf(heltall);
double flyttall = 5.2;
String s2 = String.valueOf(flyttall);
```

en metode for hver mulige argument-type: Overload

Bearbeiding av tekststrenger

- Klassen String inneholder mange nyttige metoder
  - `s.charAt(pos)` returner karakteren på posisjon pos i s
  - `s.equals(s2)` returnerer true hvis s og s2 er like tegn for tegn
  - `s.substring(3,5)` returnerer kopi av innhold i posisjon 3-4 som ny streng
  - `split`
  - `toLowerCase` og `toUpperCase`
  - `trim`

- Du finner alt om String i Java API

- Java metoder kan "overloades" => finnes i flere varianter:
  - Samme navn og type, men ulike parametere.
  - Den som passer med argumentene i kallet, er den som blir utført.

Klassen Scanner

"scanner" et buffer –fra en fil, fra terminal eller fra en String

Klassen Scanner

Scanner ligger i pakken **java.util**, som må importeres

Oppretter et objekt av klassen **Scanner**

Konstruktøren tar et argument som angir hvor du skal lese fra
- **System.in** (terminalen) eller
- et objekt av klassen **File** (om du skal lese fra fil) eller
- en tekststreng (**String**) (om du skal lese fra en streng))

To hovedmåter å lese inn data vha Scanner:
- en hel linje (som så skal lagres eller prosesseres videre)
- ett og ett "token" (vanligvis atskilt av blanke)

En linje av gangen som tekst

- I prinsippet det vi gjorde i Python i IN1000: Leser en og en linje inn i en String uten å bry oss om typer, eller om det er ett eller flere (eller ingen) ord

- Scanner-metoder for å lese linjevis:
  - `public boolean hasNextLine()`
  - `public String nextLine()`

- Leser forbi linjeskift, returner alt før linjeskift som en String*

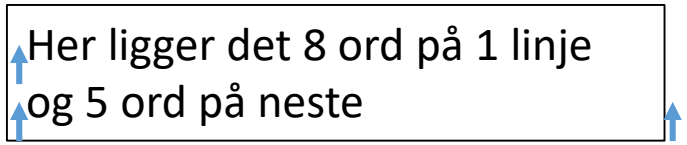
Ulikt Python (som tok med linjeskift når vi itererte over linjene i en fil)

- Kan siden bruke linjen hel, eller dele opp (som i eksempel med alder lengre oppe)

* "Kaster" linjeskiftet

Scanner har en posisjonspeker

- Jobber seg gjennom et buffer
- Har en *posisjonspeker* (pos) som holder rede på hva som er lest hittil
- *Leser* alltid fremover (og flytter da pos)
  - nextLine() // returnerer neste linje
- Men kan også *sjekke* hva som kommer - *uten* å flytte pos
  - hasNextLine() // returnerer `true` eller `false`: Er det flere linjer?



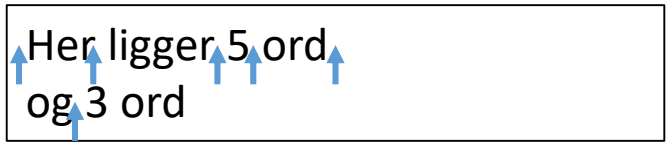
Å lese ett og ett "token" (ord)

- Et token er en sammenhengende tegnsekvens som avsluttes med whitespace* eller slutt på filen
- Kalles gjerne *ord* på norsk (kan være tall eller andre tegn)

- Viktige metoder for å behandle ord i klassen Scanner :

```
public boolean hasNext() // er det noen flere ord?
public String next()     // les og returner neste ord som en String
```

- `next` og `hasNext` hopper over innledende whitespace.



* whitespace: blank, tab, linjeskift

Å lese andre typer enn String

- Kan bruke Scanner-metoder for å teste neste ord:
  - `public boolean hasNextInt()` (sjekk om neste ord er int)
  - `public boolean hasNextDouble()` (sjekk om neste ord er double)

- Kan deretter lese med riktig metode, f eks:
  - `public int nextInt()` (les og returner neste ord som en int)
  - `public double nextDouble()` (les og returner neste ord som en double)

Test av Scanner metoder –hva skjer?

```
import java.util.Scanner;

public class LesTokens {
    public static void main(String[] args) {
      String test = "     1kjdfs\n834756 2.3";
      Scanner les = new Scanner(test);
      System.out.println(les.next());
      System.out.println(les.nextInt());
      System.out.println(les.hasNextInt());
      System.out.println(les.nextDouble());

    }
}
```

Utskrift:

1kjdfs

834756

false

2.3

Navn og alder (I)

```
import java.util.*;
class LesFraTermNavnAlder {
    public static void main (String [ ] args) {
        int alder;
        String navn;
        Scanner minInn = new Scanner (System.in);
        System.out.print(" Skriv navn: ");
        navn = minInn.nextLine();   // nextLine leser forbi linjeskift
        System.out.print(" Skriv alder: ");
        alder = minInn.nextInt();   // nextInt leser fra starten av neste
linje

        System.out.println(" Du heter " + navn + " og er " + alder + " aar.");
    }
}
```

Alder og navn (II)

```
import java.util.*;
class LesFraTermAlderNavn {
    public static void main (String [ ] args) {
        int alder;
        String navn;
        Scanner minInn = new Scanner (System.in);
        System.out.print(" Skriv alder: ");
        alder = minInn.nextInt();   // nextInt leser ikke forbi linjeskift
        System.out.print(" Skriv navn: ");
        navn = minInn.nextLine();   // nextLine leser forbi 1. linjeskift
        // og finner det linjeskiftet som "lå igjen" etter nextInt .
        // I variabelen navn legges det dermed bare en tom streng

        System.out.println(" Du heter " + navn + " og er " + alder + " aar.");
    }
}
```

Lese ett og ett ord fra fil

- Vi bruker **Scanner** som før
- Må først opprette et objekt av klassen **File** (ligger i **java.io**)
- Dette sendes som argument til nytt **Scanner**-objekt

```
class LesFraFilTokens {
    public static void main(String[] attr)
            throws FileNotFoundException {
        File minFil = new File("Handleliste.txt");
        Scanner lesFil = new Scanner(minFil);
        while (lesFil.hasNext()) {
            String vare = lesFil.next();
            System.out.println(vare);
        }
    }
}
```

Trening i fillesing og Scanner:
Skriv finnEldsteNavn med innlesing ord for ord

Skrive til fil

Å skrive til fil

```java
import java.io.PrintWriter;
import java.io.FileNotFoundException;

class SkrivTilFil{
    public static void main (String[] args)
    throws FileNotFoundException {
        PrintWriter utfil = new PrintWriter("utfil.txt");
        utfil.println("Linje 1");
        utfil.close();
    }
}
```

Array

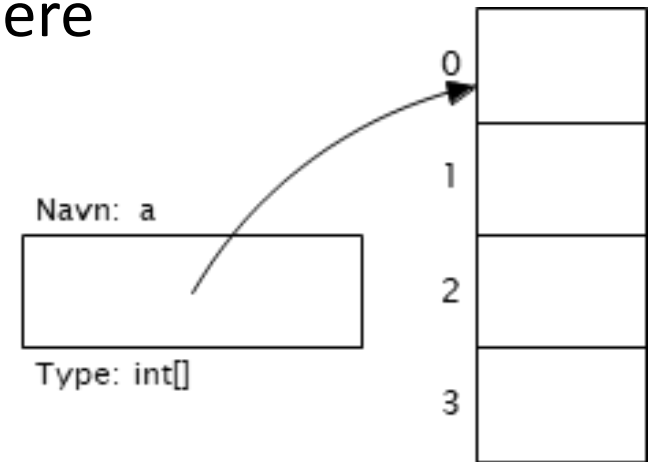
- I Java er array et alternativ til Pythons lister.
- En *array* er en datastruktur med mange elementer av samme type.

En array deklareres og opprettes slik:

```
int[] a = new int[4];
```

Array egenskaper

- Elementene er av samme type og lagres i etterfølgende celler i minnet.
- Dette gjør det til en effektiv struktur å aksessere

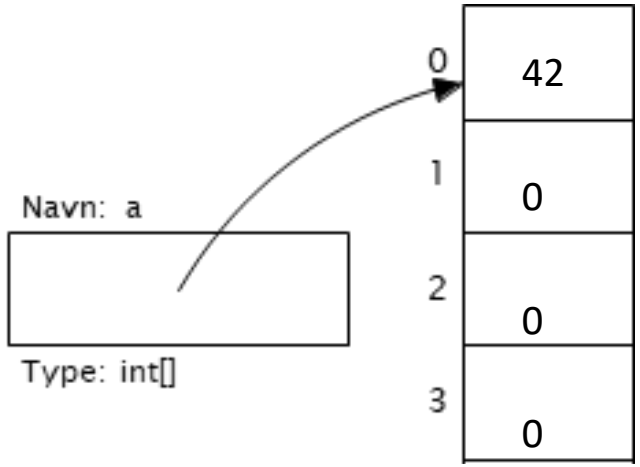


- Ikke en klasse - tilbyr ingen metoder!
- Ønsker vi en "smart" array må vi bruke en ArrayList (fra Java-biblioteket)

Array bruk

Opprette, legge inn, lese

```
int[] a = new int[4];
a[0] = 42;
int sum = 0;
for (int i=0; i<a.length; i++) {
   sum += a[i];
}
```



Innhold i en array

- Alle elementer må ha samme type, bestemmes av deklarasjonen av variabelen

```
double[] priser;    // Ikke opprettet array ennå!
boolean[] resultater;
```

- Kan være referanser til objekter av samme klasse

```
String[] emnekoder = new String[3]; //Array opprettet, med null-verdier
Person[] deltakere;
```

- Elementer kan være andre array-er (flerdimensjonal)

```
int[][] tabell = { {1,2,3}, {3,6,9}}; // Initialisert
System.out.println(tabell[0][0]);
System.out.println(tabell[2][0]);
```

Hva skjer her?

IndexErrorOutOfBounds

Gjennomløp av array: Summere tall

Merk bruken av argumenter fra kommandolinjen –disse mottar vi som en String array

Lengden av arrayen finner vi med .length
(NB: Ingen parenteser/ metodekall for arrayer!)

```
class SummerDisse2 {
    public static void main(String[] args) {



    }
}
```

Gjennomløp av array

```
class SummerDisse2 {
    public static void main(String[] args) {
        int sum = 0;
        for (int i=0; i<args.length; i++)
            sum += Integer.parseInt(args[i]);
        System.out.println(sum);
    }
}
```

```
M:\Ifi\Undervisning\IN1010 V2020\Forelesninger\uke2\Kode
>javac SummerDisse2.java

M:\Ifi\Undervisning\IN1010 V2020\Forelesninger\uke2\Kode
>java SummerDisse 10 10 10
30
```

for-each/ enhanced/ forenklet for-løkke

Ligner for-løkken i Python

Går gjennom alle verdier i en samling (her array)

```
class SummerDisse {
    public static void main(String[] args) {
        int sum = 0;
        for (String str : args)
            sum += Integer.parseInt(str);
        System.out.println(sum);
    }
}
```

Bruker "vanlig" for-løkke hvis:

- Hvis vi trenger indeks-verdien inne i løkken
- Hvis vi skal endre verdier i arrayen

Mer om arrayer, for-løkker, ...

Big Java –elektronisk tilgjengelig fra semestersiden under Pensum

- Forklaringer, tegninger, detaljer
- ... eksempler, tips inkludert "mønstere" for bruk

- Arrayer kapittel 6
  - Forenklet for-løkke i 6.2
- Løkker kapittel 4

Oppsummering

Slå opp klasser og metoder i Java 8 API

Bruk lærebok eller Java tutorials (Java 8) for mer detaljerte forklaringer

Oppskrifter på lese fra og skrive til fil og terminal i notat på semestersiden

Exceptions gir nyttig og noen ganger nødvendig håndtering av feilsituasjoner

Primitive typer kan pakkes inn i objekter av tilsvarende klasse Integer, Double, Boolean, .. Disse "wrapper"-klassene har også nyttige statiske metoder (og konstanter)

Bli kjent med String-klassen for manipulasjon og konvertering

Array er en effektiv, nyttig og veldig vanlig konstruksjon i mange språk

Java har to ulike former for for-løkker

UNIVERSITETET I OSLO

Institutt for informatikk

IN1010 våren 2022
25. januar

Objektorientering i Java

Om innkapsling og enhetstesting
Mer om arrayer og noen klasser som kan ta vare på objekter
Om parameteroverføring

Stein Gjessing

Agenda

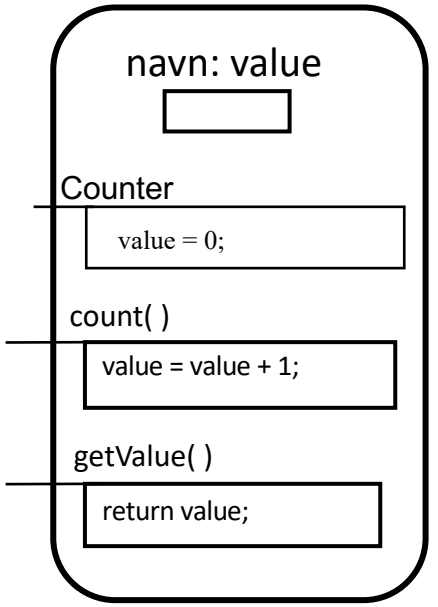
- Hvordan representeres objekter i minnet
- Mer om objekter og innkapsling
  - Enhetstesting
- Mer om arrayer
- Om beholdere i Javas bibliotek:
  - HashMap
  - ArrayList

UNIVERSITETET I OSLO

OO-programmering

Institutt for informatikk

- Objekter er levende (Kristen Nygaard)
- I noen OO-språk sender man meldinger mellom objekter

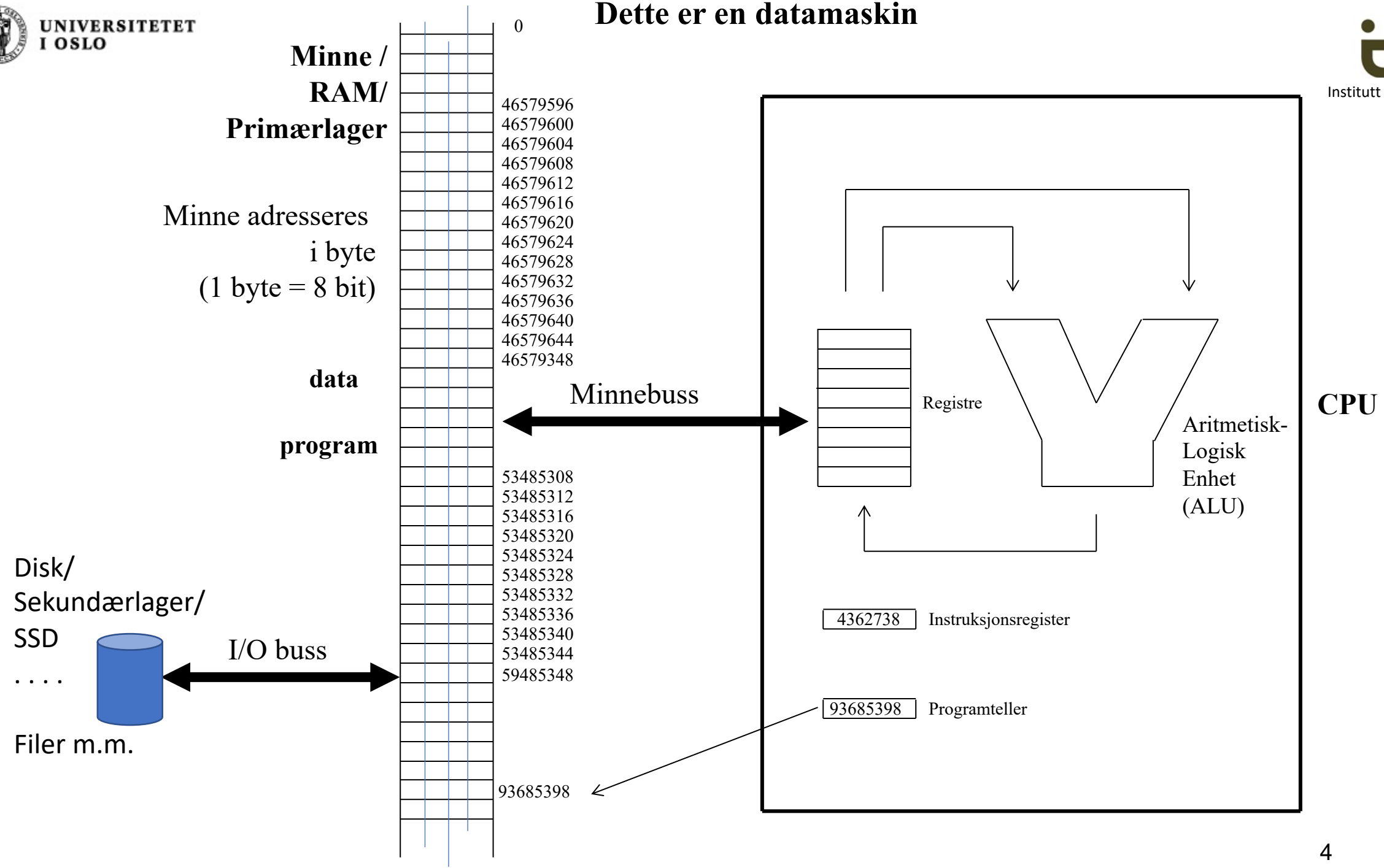


Et **objekt** av klassen Counter

- **Variable og konstanter - "DATA"**
  Kristen Nygaard: SUBSTANS
- **Metoder – handlinger**
- **Eksternt blikk: public-metoder**
- **Internt blikk: implementasjon**

```
class Counter {
    private int value;
    public Counter() {
        value = 0;
    }
    public void count( ) {
        value = value + 1;
    }
    public int getValue( ) {
        return value;
    }
}
```

Dette er en datamaskin



UNIVERSITETET I OSLO

Sist snakket vi om Javas primitive/basale **verdier**

Institutt for informatikk

boolean	~~1~~	00000001	8 bit (Det er for upraktisk å lagre (sammen med andre data), pakke ut og inn og teste ett bit)
byte	01001110		8 bit
char	0100111010110110		16 bit
short	0100100101101110		16 bit
int	01010010110110101000100101101110		32 bit
long	0101001011010011010110100010101001010110100110101000100101101110		64 bit
float	01010010110110101000100101101110		32 bit
double	0101001011010011010110100010101001010110100110101000100101101110		64 bit

Type: int Navn: tall

00000000000000000000000000001010

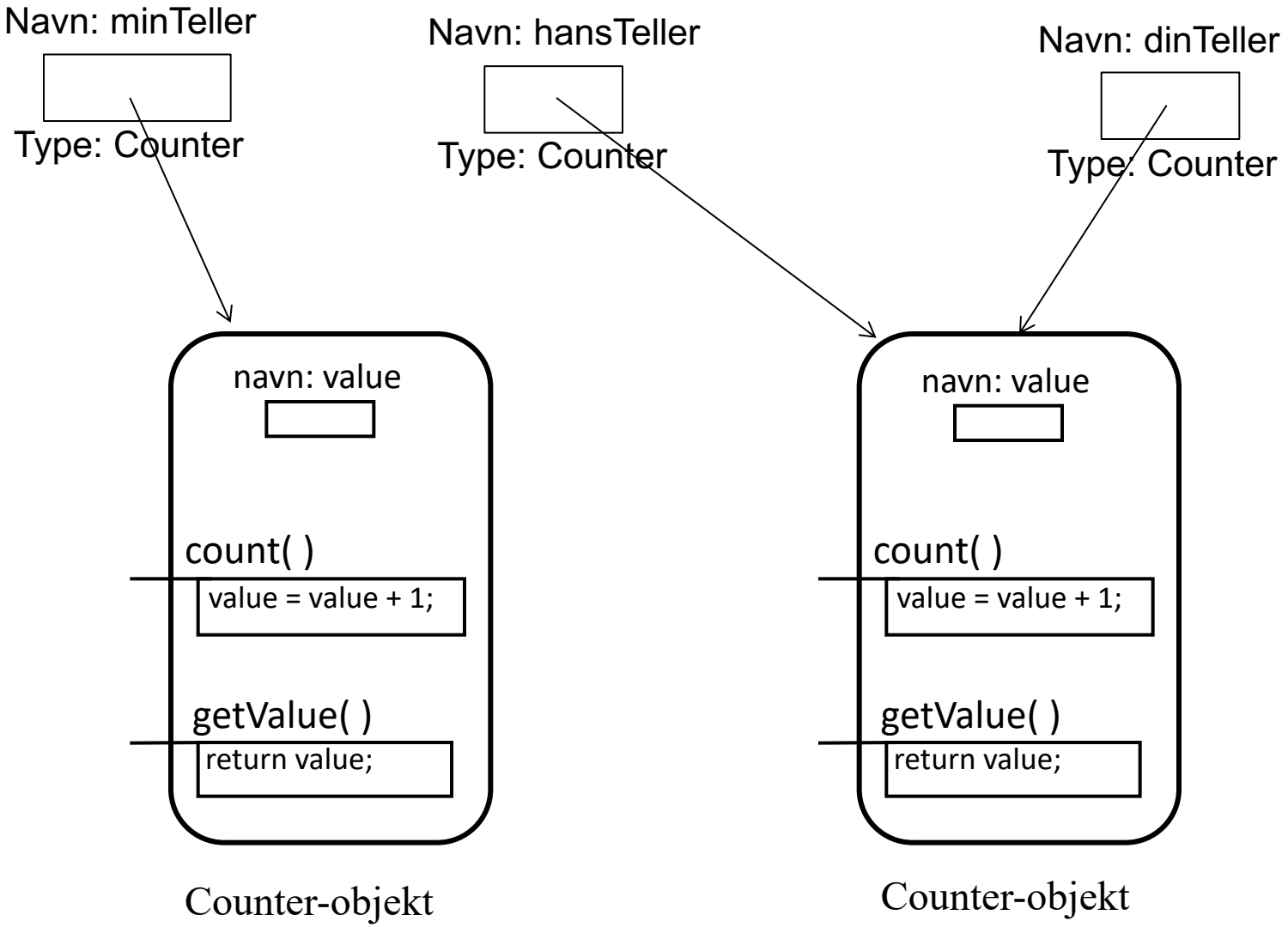
En **variabel** er et sted i minnet der det er lagret en slik **verdi**.

F.eks. deklarasjonen int tall = 10;

Java er **et sterkt typet språk:**

En variabel **har bare lov å inneholde en verdi av den deklarerte typen** ~~tall = 3.14~~

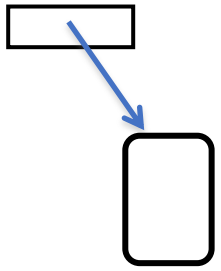
Nå skal vi snakke mer om referanseverdier



Hva er en referanse / peker

Python:

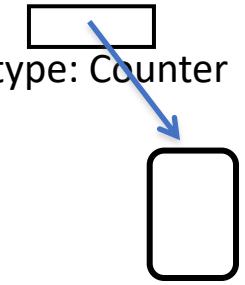
navn: minTeller



Java:

navn: minTeller

type: Counter



F.eks. deklarasjonen: Counter minTeller = new Counter();

En variabel: som innholder verdien:

navn: minTeller type: Counter

00000110010100000010000001100100000011001010000001000000 1100111

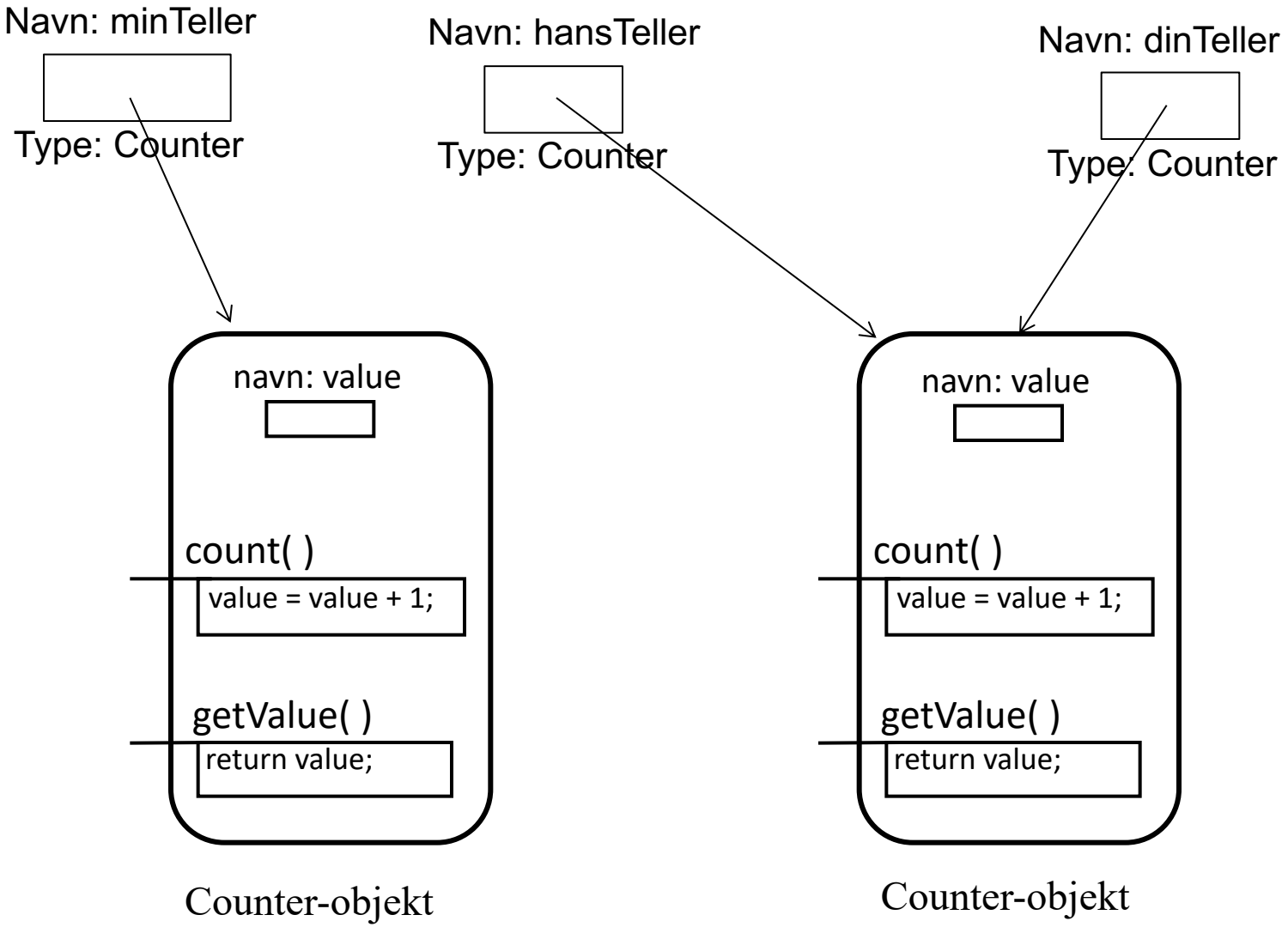
En variabel er et sted i minnet

00000110010100000010000001100100000011001010000001000000 1100111

En referanse(verdi) er en adresse til et sted i minnet der det ligger et objekt (eller en array)
I Java bruker vi *referanse* og *peker* synonymt.

Java er **et sterkt typet språk:**
Referanse-variablen i dette eksemplet **har bare lov å inneholde en peker til et** Counter-objekt

Mer om referanser



Referanser

Navn: minTeller

53485320

Type: Counter

Navn: hansTeller

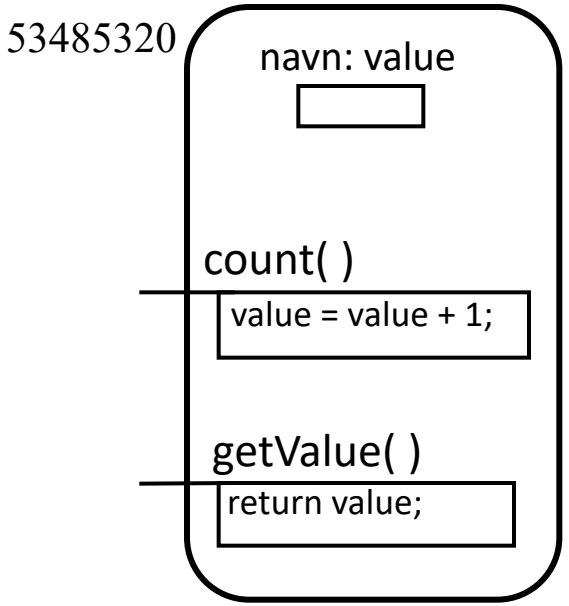
9483567

Type: Counter

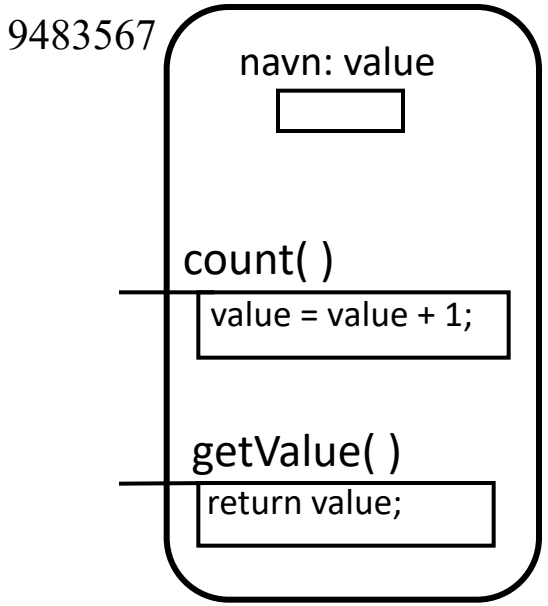
Navn: dinTeller

9483567

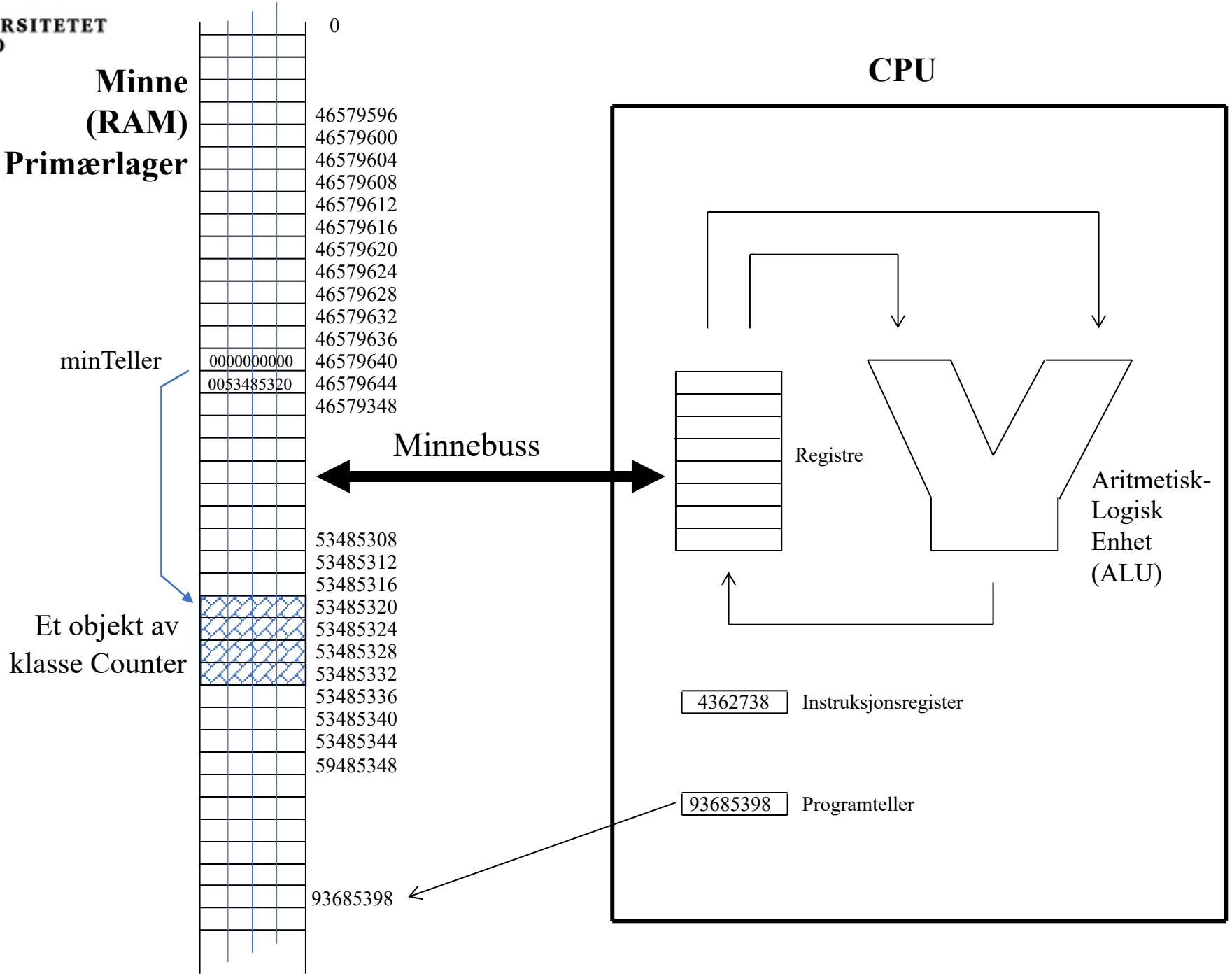
Type: Counter



Counter-objekt



Counter-objekt


Minne
(RAM)
Primærlager
0
46579596
46579600
46579604
46579608
46579612
46579616
46579620
46579624
46579628
46579632
46579636
minTeller
0000000000
46579640
0053485320
46579644
46579348
Minnebuss
53485308
53485312
53485316
53485320
53485324
53485328
53485332
53485336
53485340
53485344
59485348
Et objekt av
klasse Counter
93685398
CPU
Registre
Aritmetisk-
Logisk
Enhet
(ALU)
4362738
Instruksjonsregister
93685398
Programteller

Javas **uttrykk**

- **Uttrykk** (expressions) evalueres til en **verdi** av enten en
  - Primitiv verdi eller en
  - Referanseverdi

- De fleste uttrykk ser vi på høyresiden i en tilordning: tall = start + nyAntall;

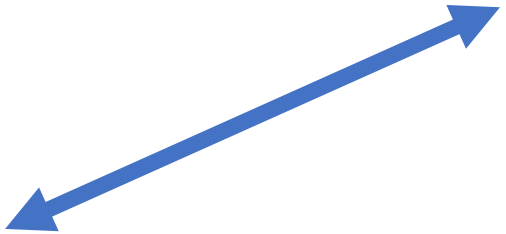
- Husk at en aktuell parameter (argument) er et uttrykk
  - som evalueres til en verdi av en gitt type
  - Denne verdien blir startverdien til den formelle parameteren (en lokal variabel)
  - Se til slutt i dag

Objektorienter (OO) programmering

Nå skal du lære om OO-programmering samtidig som du lærer Java

Hva er (objektorientert) programmering ?

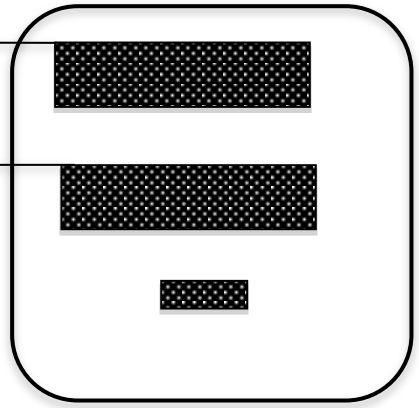
Forrige uke: Et objekt er en sort boks + *grensesnittet* mot omverdenen





Trykke på knappen = count ( )

Lese av telleverket = getValue( )

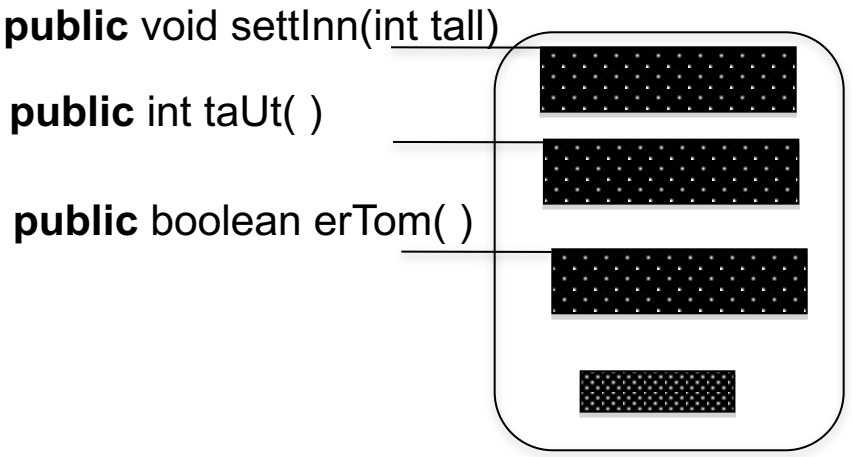


Ukjent implementasjon for omverdenen

Ukjente **private** data (variabler)

Et objekt kan også være en hjelpekonstruksjon i programmet

Et objekt som tar vare på ett tall
(kan bli en mer "intelligent variabel").



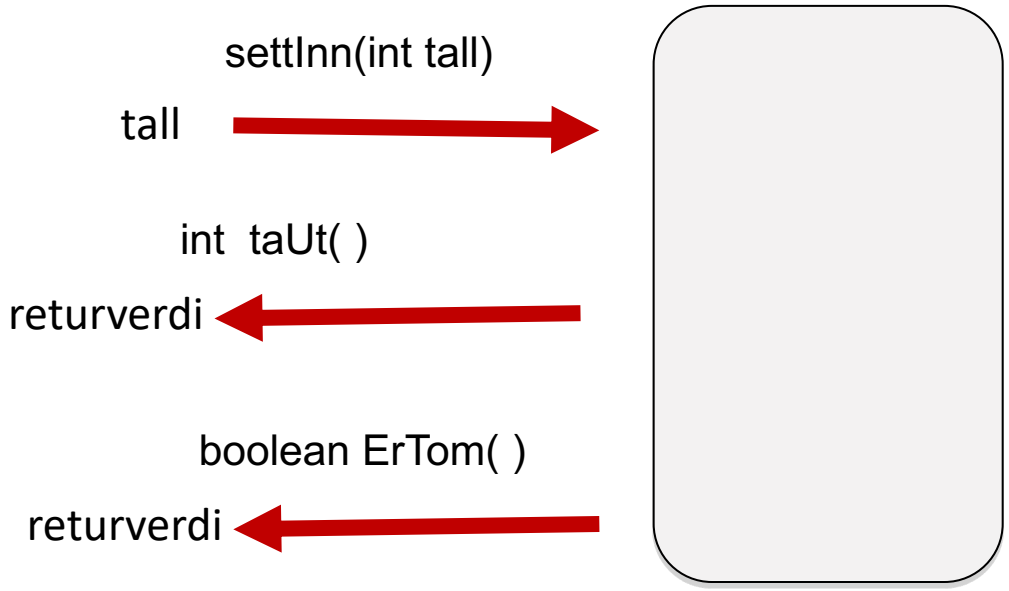
Ukjent implementasjon av metode

Ukjent implementasjon av metode

Ukjent implementasjon av metode

Ukjente **private** data og ukjente **private** metoder

En sort boks som tar vare på ett tall



Hvordan virker settInn hvis det er et tall der fra før?

Hvordan virker taUt hvis det ikke er noe tall i boksen?

. . .

Grensesnitt / tjenester

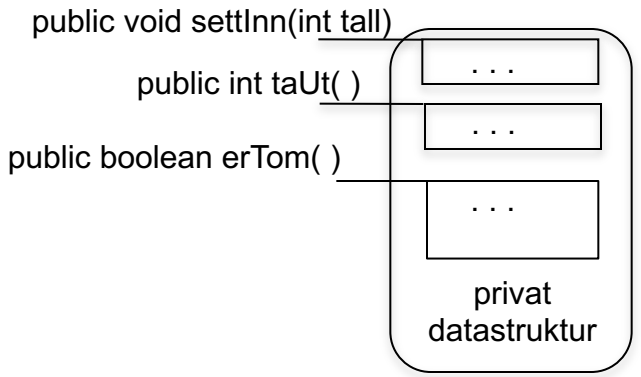
Hvilke metoder trengs ?
Hva er deres parametre ?
Hvordan skal disse metodene virke ?

Vi lager en klasse som vi kan lage objekter av:

```
class EnkelHeltallsbeholder {
    private . . . . ;

    public void settInn(int tall) {
        . . . . .
    }
    public int taUt( ) {
        . . . . .
    }
    public boolean erTom ( ) {
        . . . . .
    }
}
```

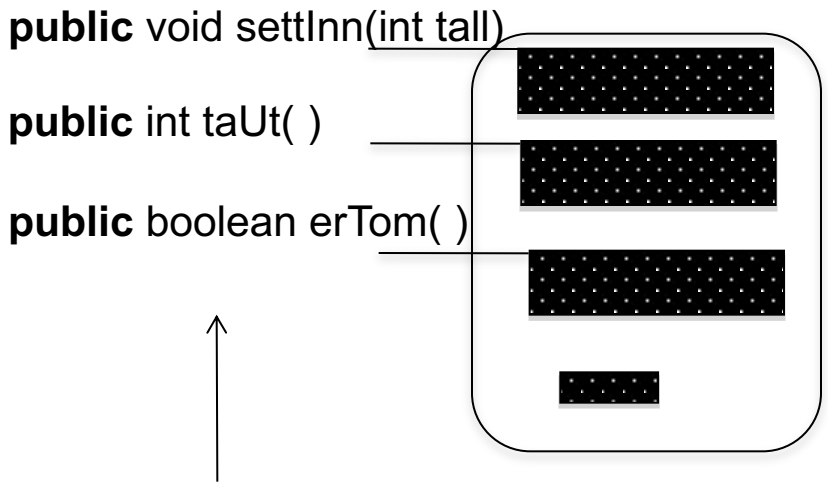
new EnkelHeltallsbeholder()
gir dette objektet:



Objekt av klassen
EnkelHeltallsbeholder

Denne koden kan *nesten* kompileres (og kjøres)!

En metodes *signatur*

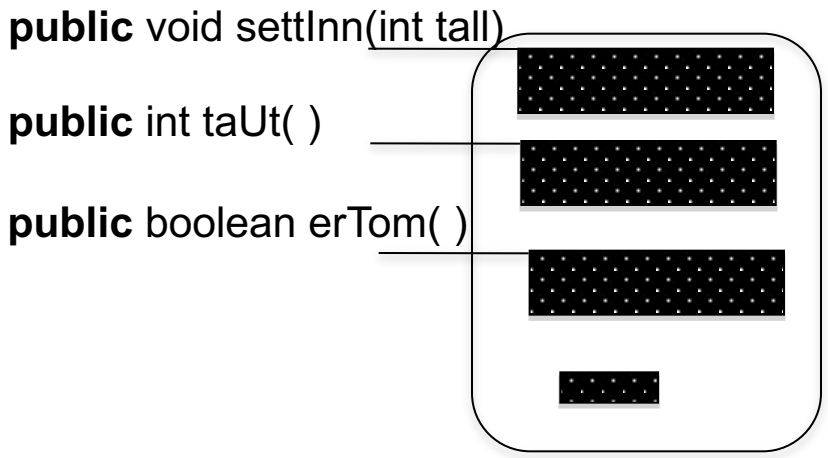


Dette kaller vi metodenes **signaturer** (skrivemåte, syntaks)

Signaturen til en metode er

- navnet på metoden
- typene, rekkefølgen og navnene til parametrene
- retur-typen (ikke i Java)

Metodenes **semantikk**

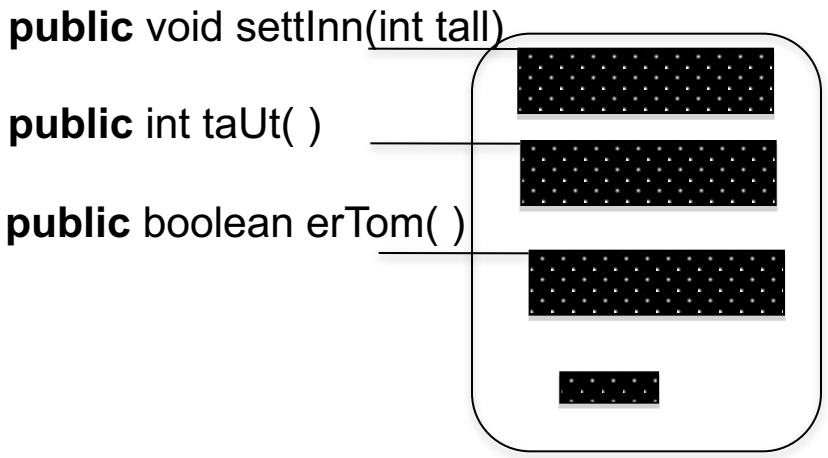


Hva gjør disse metodene? Hvordan virker de? Hva er sematikken til metodene? Hvilke tjenester tilbyr de?

Semantikk betyr virkemåte



Metodenes **semantikk**



Dette er spesifikasjonen av tjenestene objektet tilbyr, **dvs. utgangspunktet for programmeringen av klassen**

- Forslag til semantikk:
  - Metoden "settInn" gjør at objektet tar vare på tallet som er parameter til metode. Hvis det er et tall i objektet fra før, har metoden ingen virkning.
  - Metoden "taUt" tar ut av objektet det tallet som tidligere er satt inn. Metoden returnerer det tallet som slettes.
  - Metoden "erTom" returnerer sann om objektet er tomt, usann ellers.

- Informatikkens 3. lov: ☺

1. Først betemmer vi semantikken og signaturene til metodene
2. Deretter implementerer vi metodene
   samtidig som vi bestemmer oss for hva de private dataene skal være

Dette gjelder for alle programmeringsspråk - dette er ikke Java-spesifikt.

☺ Dette er en spøk. Informatikken har ikke nummererte lover

Testing -- Enhetstesting

- Når vi planlegger og skriver programmer prøver vi å overbevise oss selv (og dem vi skriver sammen med) at den koden vi skriver kommer til å utføre det vi ønsker (oppfylle semantikken)
- Men vi kommer alltid til å tenke og skrive feil
- Derfor må vi **teste** programmene våre
- Objektorientering / modularisering:
  - Teste et objekt eller en modul om gangen - **enhetstesting**
    - Sørg for at den er så riktig som mulig
  - Deretter kan vi test sammensettingen av objektene / modulene - **integrasjonstesting**

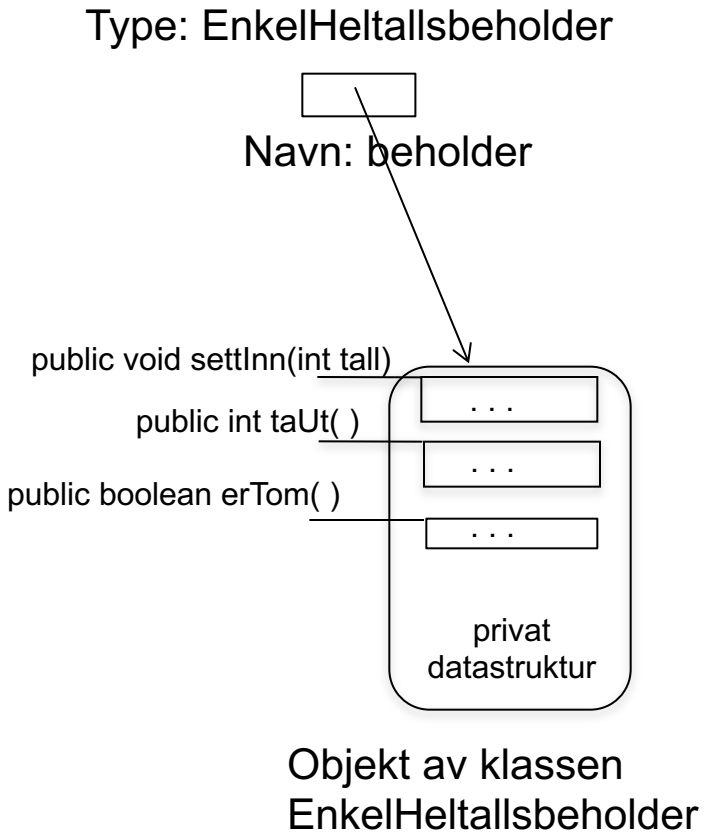
Mer om testing 5. april

Om å teste et objekt (som tar vare på ett tall)


```
class EnkelHeltallsbeholder {
    private . . . . ;

    public void settInn(int tall) {
        . . . . .
    }
    public int taUt( ) {
        . . . . .
    }
    public boolean erTom ( ) {
        . . . . .
    }
}
```

```
class TestAvBeholder {
    public static void main (String[ ] args){
        EnkelHeltallsbeholder beholder =
                    new EnkelHeltallsbeholder();
        . . .
        . . .
    }
}
```

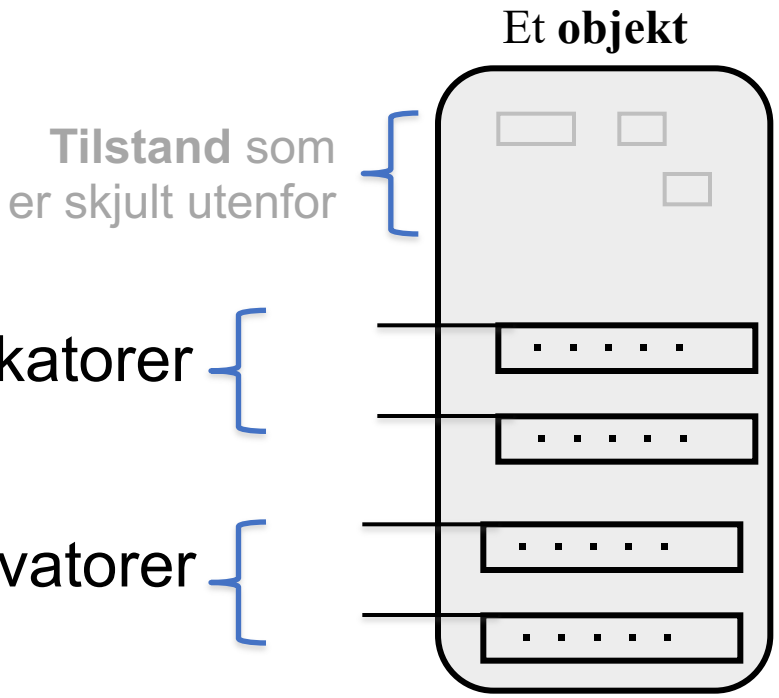


Når skal vi skrive testprogrammet?

Om du er alene?
Om dere er flere?

Modifikatorer og Observatorer

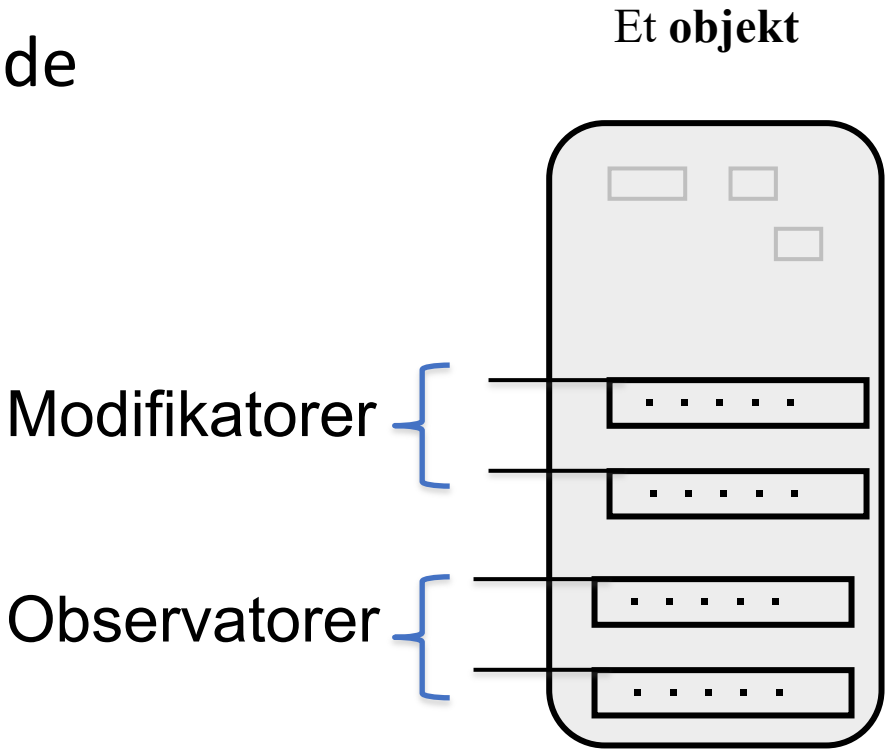
- En modifikator-metode forandrer tilstanden til et objekt
- En observator-metode leser av tilstanden uten å forandre den



Modifikator, f.eks. settInn(), Observator, f.eks. les()

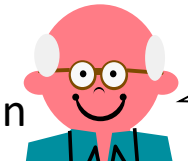
Testing - - Enhetstesting

- Når vi skal teste et objekt kan vi først kalle en observator-metode, så en modifikator-metode og så igjen observator-metoden og se om vi observerer det ønskede resultatet - den ønskede forandringen i **tilstanden**.



For de som er spesielt interessert:

Ole-Johan Dahl



Objektets semantikk kan beskrives av den historiske sekvensen av operasjoner på objektet

UNIVERSITETET I OSLO

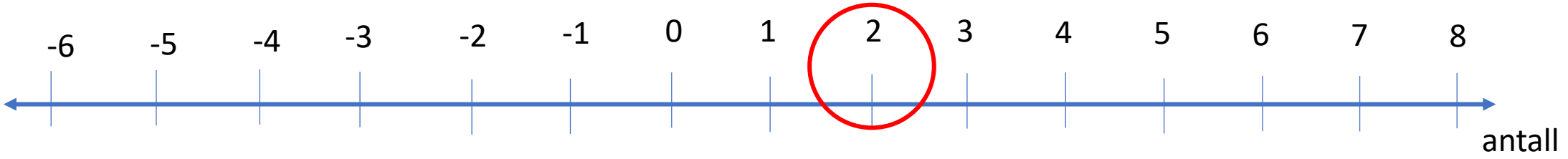
En liten digresjon:

Tillstanden (til et objekt) - tilstandsrommet

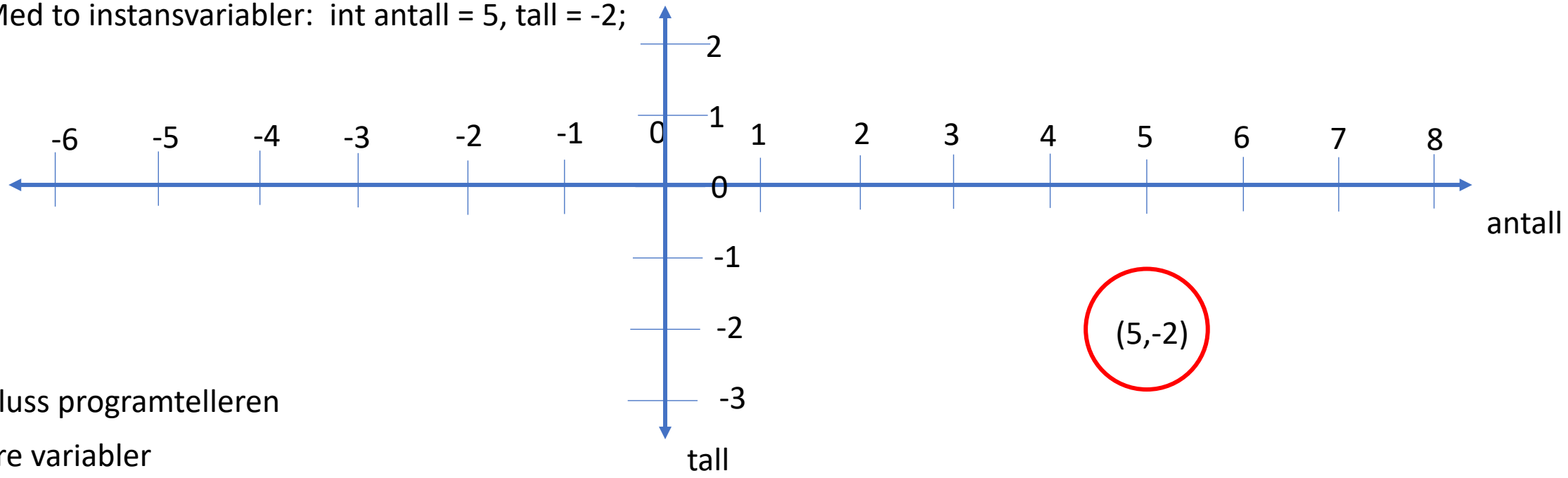
(Et litt matematisk syn)

Institutt for informatikk

- Med én instansvariabel: int antall = 2;



- Med to instansvariabler: int antall = 5, tall = -2;



- Pluss programtelleren
- Tre variabler
- Fire variabler . . . . .

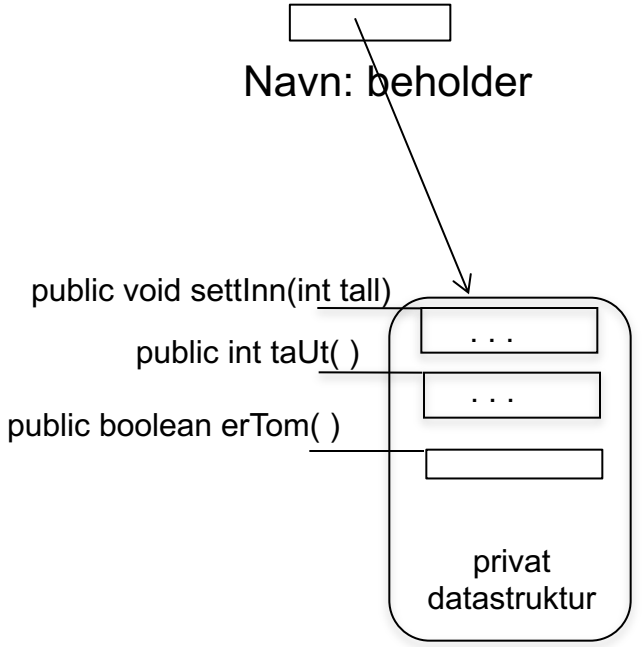
Veldig enkel test av beholder


```
class EnkelHeltallsbeholder {
    private int tallet;
    private boolean tom = true;
    public void settInn(int tall) {
        if (tom) tallet = tall;
    }
    public int taUt( ) {
        tom = true;
        return tallet;
    }
    public boolean erTom ( ) {
        return tom;
    }
}
```



```
class VeldigEnkelTestAvBeholder {
    public static void main (String[ ] args){
        EnkelHeltallsbeholder beholder =
                    new EnkelHeltallsbeholder();
        beholder.settInn(17);
        if (beholder.taUt() == 17)
                {System.out.println ("Riktig");}
        else {System.out.println("Feil");}
      }
}
```

Type: EnkelHeltallsbeholder



Objekt av klassen
EnkelHeltallsbeholder

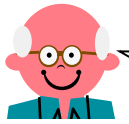
```
>java VeldigEnkelTestAvBeholder
Riktig
>
```

Enkel enhetstesting av heltallsbeholder


```
public class EnkelTestAvHeltallsbeholder {
   public static void main (String[ ] arg) {
       EnkelHeltallsbeholder beholder =
                 new EnkelHeltallsbeholder();
       beholder.settInn(17);
       if (beholder.taUt() == 17)
            {System.out.println ("Riktig 1");}
       else {System.out.println("Feil 1");}
       beholder.settInn(18);
       beholder.settInn(17);
       if (beholder.taUt() == 18)
            {System.out.println ("Riktig 2");}
       else {System.out.println("Feil 2");}
       if (beholder.erTom())
            {System.out.println ("Riktig 3");}
       else {System.out.println("Feil 3");}
       beholder.settInn(19);
       if (! beholder.erTom())
            {System.out.println ("Riktig 4");}
       else {System.out.println("Feil 4");}
   }
}
```



```
class EnkelHeltallsbeholder {
     private int tallet;
     private boolean tom = true;
     public void settInn(int tall) {
         if (tom) tallet = tall;
     }
     public int taUt( ) {
         tom = true;
         return tallet;
     }
     public boolean erTom ( ) {
         return tom;
     }
}
```



Test en (modifikator)metode om gangen
Lag gode tekster for testene
Lag egne test-metoder

```
>java EnkelTestAvHeltallsbeholder
Riktig 1
Feil 2
Riktig 3
Feil 4
>
```

Med Javadoc (og uten? feil)


```
/** Objekter av denne klassen tar vare på
 *  ett  heltall.
 *  Initielt er beholderen tom
 *
 * @author Stein Gjessing
 * versjon 21. januar 2022
 */
public class EnkelHeltallsbeholder {
      private boolean tom = true;
      private int tallet;
     /**
      * Gjør at objektet tar vare på tallet som
      * er parameter til metoden.
      * Hvis det allerede er lagret et tall i objektet,
      * dvs. at beholderen ikke er tom, har denne
      *  metoden ingen virkning
      *
      *@param tall  tallet som objektet skal
      *    ta vare på
      */
      public void settInn(int tall) {
            if (tom) tallet = tall;
            tom = false;
      }
```

```
      /**
      * Sjekkeren om objektet er tomt
      *
      @return objektet er tomt
      */
      public boolean erTom ( ) {
            return tom;
      }
      /**
      * Tar ut av objektet det tallet objektet
      *   tar vare på.
      * Om objektet alt er tomt, returneres en
      *  ubestemt verdi.
      * Etter dette kallet er objektet tomt.
      *
     @return tallet som tas ut. eller en
      *     ubestemt verdi om objektet er tomt
      */
       public int taUt( ) {
             tom = true;
             return tallet;
       }
}
```

UNIVERSITETET I OSLO

Javadoc resultat

PACKAGE CLASS TREE INDEX HELP

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD SEARCH:

Class Enkelheltallsbeholder

java.lang.Object
Enkelheltallsbeholder

```
public class Enkelheltallsbeholder
extends Object
```

Objekter av denne klassen tar vare på ett heltall. Initielt er beholderen tom

Constructor Summary

Constructors

Constructor	Description
`Enkelheltallsbeholder()`	

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method	Description
boolean	`erTom()`	Sjekkeren om objektet er tomt
void	`settInn(int tall)`	Gjør at objektet tar vare på tallet som er parameter til metoden.
int	`taUt()`	Tar ut av objektet det tallet objektet tar vare på.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Details

Enkelheltallsbeholder

```
public Enkelheltallsbeholder()
```

Method Details

settInn

```
public void settInn(int tall)
```

Gjør at objektet tar vare på tallet som er parameter til metoden. Hvis det allerede er lagret et tall i objektet, dvs. at beholderen ikke er tom, har denne metoden ingen virkning

Parameters:

`tall` - tallet som objektet skal ta vare på

erTom

```
public boolean erTom()
```

Sjekkeren om objektet er tomt

Returns:

objektet er tomt

taUt

```
public int taUt()
```

Tar ut av objektet det tallet objektet tar vare på. Om objektet alt er tomt, returneres en ubestemt verdi. Etter dette kallet er objektet tomt.

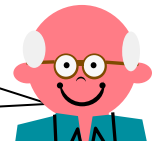
Returns:

tallet som tas ut. eller en ubestemt verdi om objektet er tomt

```
steing@eduroam-193-157-201-38 programmer % javadoc Enkelheltallsbeholder.java
Loading source file Enkelheltallsbeholder.java...
Constructing Javadoc information...
Building index for all the packages and classes...
Standard Doclet version 17.0.1+12
Building tree for all the packages and classes...
Generating ./Enkelheltallsbeholder.html...
Generating ./package-summary.html...
Generating ./package-tree.html...
Generating ./overview-tree.html...
Building index for all classes...
Generating ./allclasses-index.html...
Generating ./allpackages-index.html...
Generating ./index-all.html...
Generating ./index.html...
Generating ./help-doc.html...
steing@eduroam-193-157-201-38 programmer %
```

UNIVERSITETET I OSLO

Bare du, som er et menneske, kan sjekke at implementasjonen overholder de SEMANTISKE KRAVENE til metodene (?)



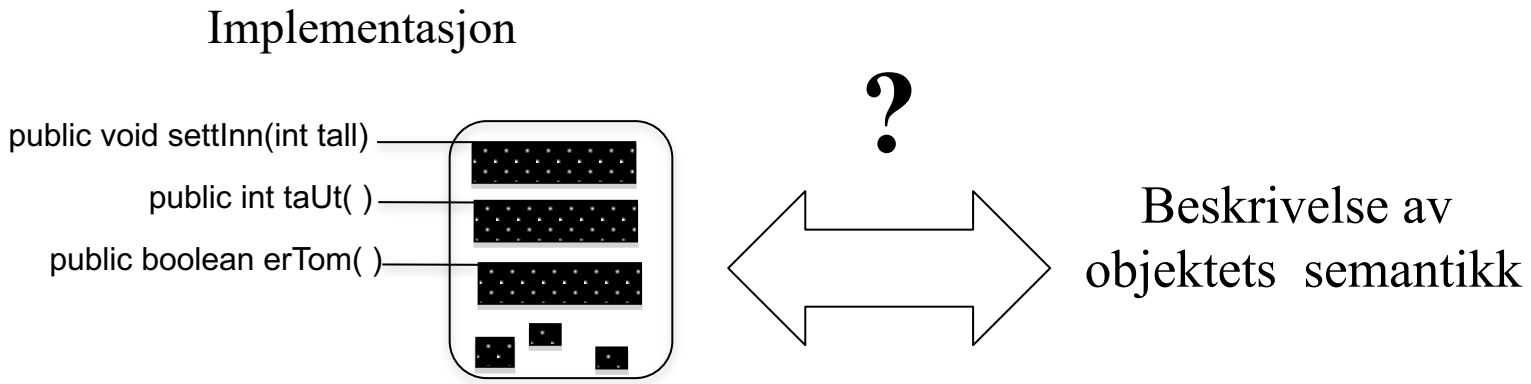
Institutt for informatikk

```
/** Objekter av denne klassen tar vare på
 *  ett  heltall.
 *  Initielt er beholderen tom
 *
 * @author Stein Gjessing
 * versjon 21. januar 2022
 */
public class Enkelheltallsbeholder {
      private boolean tom = true;
      private int tallet;
     /**
      * Gjør at objektet tar vare på tallet som
      * er parameter til metoden.
      * Hvis det allerede er lagret et tall i objektet,
      * dvs. at beholderen ikke er tom, har denne
      *  metoden ingen virkning
      *
      *@param tall  tallet som objektet skal
      *    ta vare på
      */
      public void settInn(int tall) {
            if (tom) tallet = tall;
            tom = false;
      }
      /**
      * Sjekkeren om objektet er tomt
      *
      @return objektet er tomt
      */
      public boolean erTom ( ) {
            return tom;
      }
      /**
      * Tar ut av objektet det tallet objektet
      *   tar vare på.
      * Om objektet alt er tomt, returneres en
      *  ubestemt verdi.
      * Etter dette kallet er objektet tomt.
      *
     @return tallet som tas ut. eller en
      *     ubestemt verdi om objektet er tomt
      */
       public int taUt( ) {
             tom = true;
             return tallet;
       }
}
```

Institutt for informatikk har 16 forskningsgrupper.

En av disse heter "Pålitelige systemer" (PSY).

Her arbeider de bl.a. med å formalisere disse sematiske kravene, slik at du kan få hjelp av datamaskinen til å sjekke at implementasjonen overholder de semantiske kravene. Litt mer på en senere forelesning.



De sematiske kravene kalles også en "kontrakt"
(mellom brukerene av objektet og objektet selv)

Flere eksempler på enhetstesting: Om å lage et kaninbur

```
class Kanin{
    private String navn;
    public Kanin(String nv) {
        navn = nv;
    }
    public String hentNavn() {
        return navn;
    }
}
```





```
class Kaninbur {
    private . . . ;
    public boolean settInn(Kanin k) {
        . . .
    }
    public Kanin taUt( ) {
        . . .
    }
}
```

Kaninburets tjenester / grensesnitt – Signaturene og semantikken til metodene i kaninburet

Signaturer (syntaks)

```
class Kaninbur {
        public boolean settInn(Kanin k) {. . .}
        public Kanin taUt( ) { . . . }
}
```

Semantikk (virkemåte):

- Hvis objektet er tomt vil metoden "settInn" gjøre at objektet tar vare på kaninen som er parameter til metoden, og metoden returnerer sann.
  Hvis objektet allerede inneholder en kanin gjør metoden ingen ting med objektet, og metoden returnerer usann.
- Metoden "taUt" tar ut kaninen som er i objektet og returnerer en peker til denne kaninen**.** Metoden returnerer null hvis objektet allerede er tomt.

Kaniner og kaninbur: Full kode


```
class Kanin{
    private String navn;
    public Kanin(String nv) {
        navn = nv;
    }
    public String hentNavn() {
        return navn;
    }
}
```



Men enda ikke et fullt program (mangler klasse med main).



```
class Kaninbur {
    private Kanin denne = null;
    public boolean settInn(Kanin k) {
        if (denne == null)  {
            denne = k;
            return true;
        }
        else return false;
    }
    public Kanin taUt( ) {
        Kanin k = denne;
        denne = null;
        return k;
    }
}
```

Enhetstesting av class Kaninbur

```
class Kanin{
    private String navn;
    public Kanin(String nv) {navn = nv;}
    public String hentNavn() {return navn; }
}
```

```
class Kaninbur {
    private Kanin denne = null;
    public boolean settInn(Kanin k) {
       if (denne == null)  {
          denne = k;
          return true;
       }
       else return false;
    }
    public Kanin taUt( ) {
        Kanin k = denne;
        denne = null;
        return k;
    }
}
```

Testprogram:

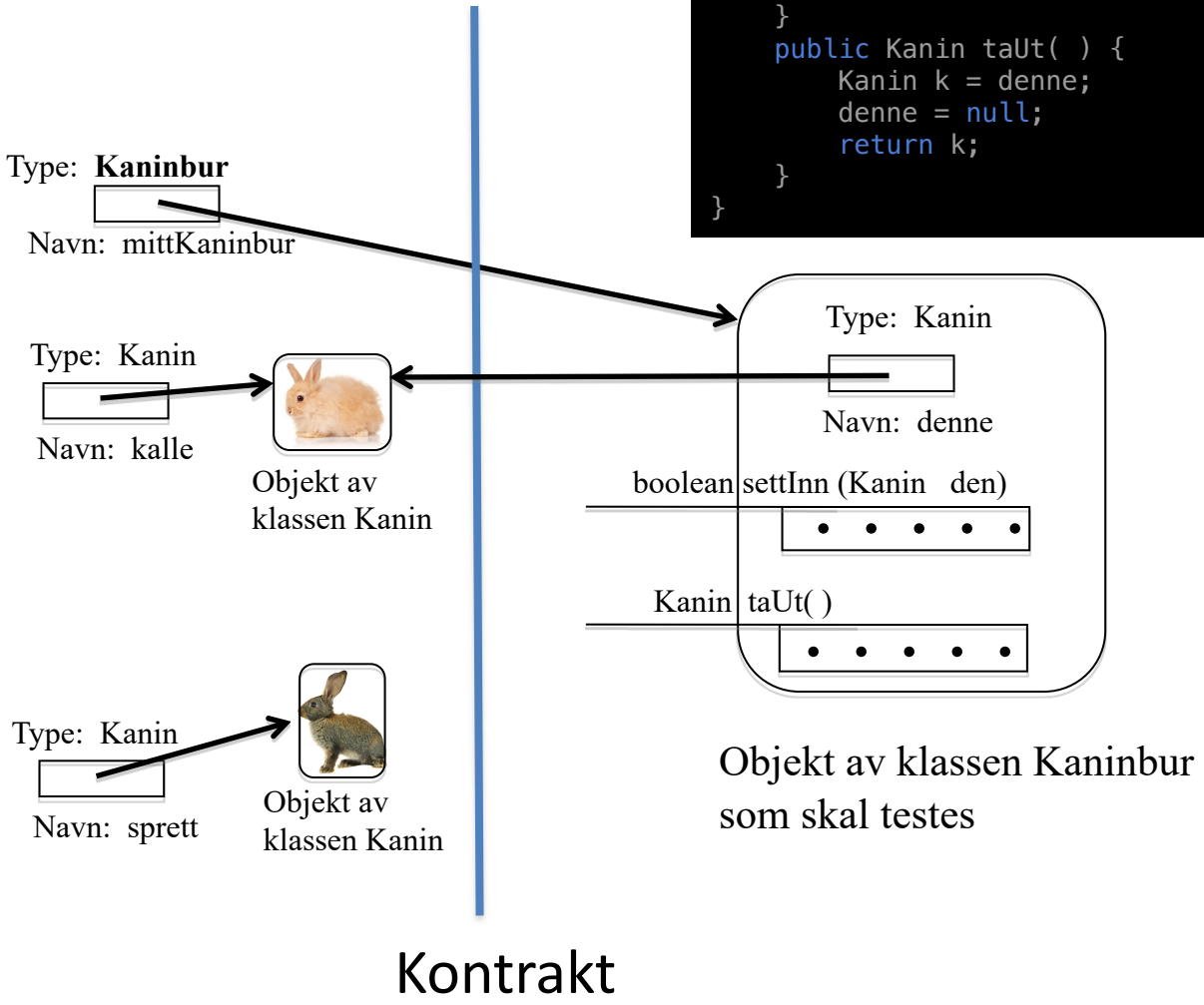
```
class Kanintest {
   public static void main ( String []args) {
      Kaninbur mittKaninbur = new Kaninbur( );




     Kanin kalle = new Kanin("Kalle");
     // Test å sette inn i tomt bur:
     boolean settInnOK = mittKaninbur.settInn(kalle);
     test("Test inn i tomt bur", settInnOK);


     Kanin sprett = new Kanin("Sprett");
     // Test å sette inn i fullt bur:
     boolean settInnOK = mittKaninbur.settInn(sprett);
     test("Test inn i fullt bur", !settInnOK);
     . . .
     . . .
     . . .

  }
  static void test(…) { . . . }
}
```



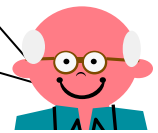
Om å lage et kaninbur til mange kaniner

3 observatorer

2 modifikatorer

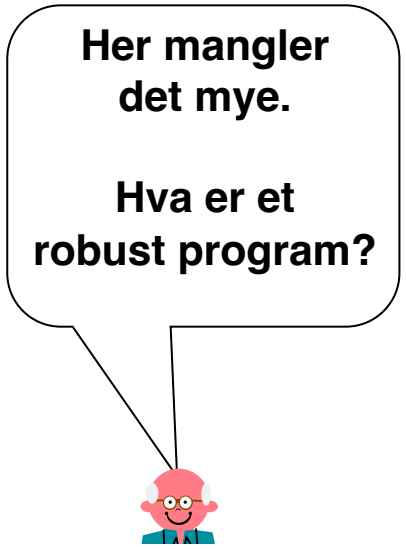
```
class    KaninGård {
   public boolean full( ) { . . . }
   public boolean tom ( ) { . . . }
   public Kanin  finnEn(String navn) { . . . }
   public void settInn (Kanin kn) { . . . }
   public void fjern(String navn) { . . . }
}
```

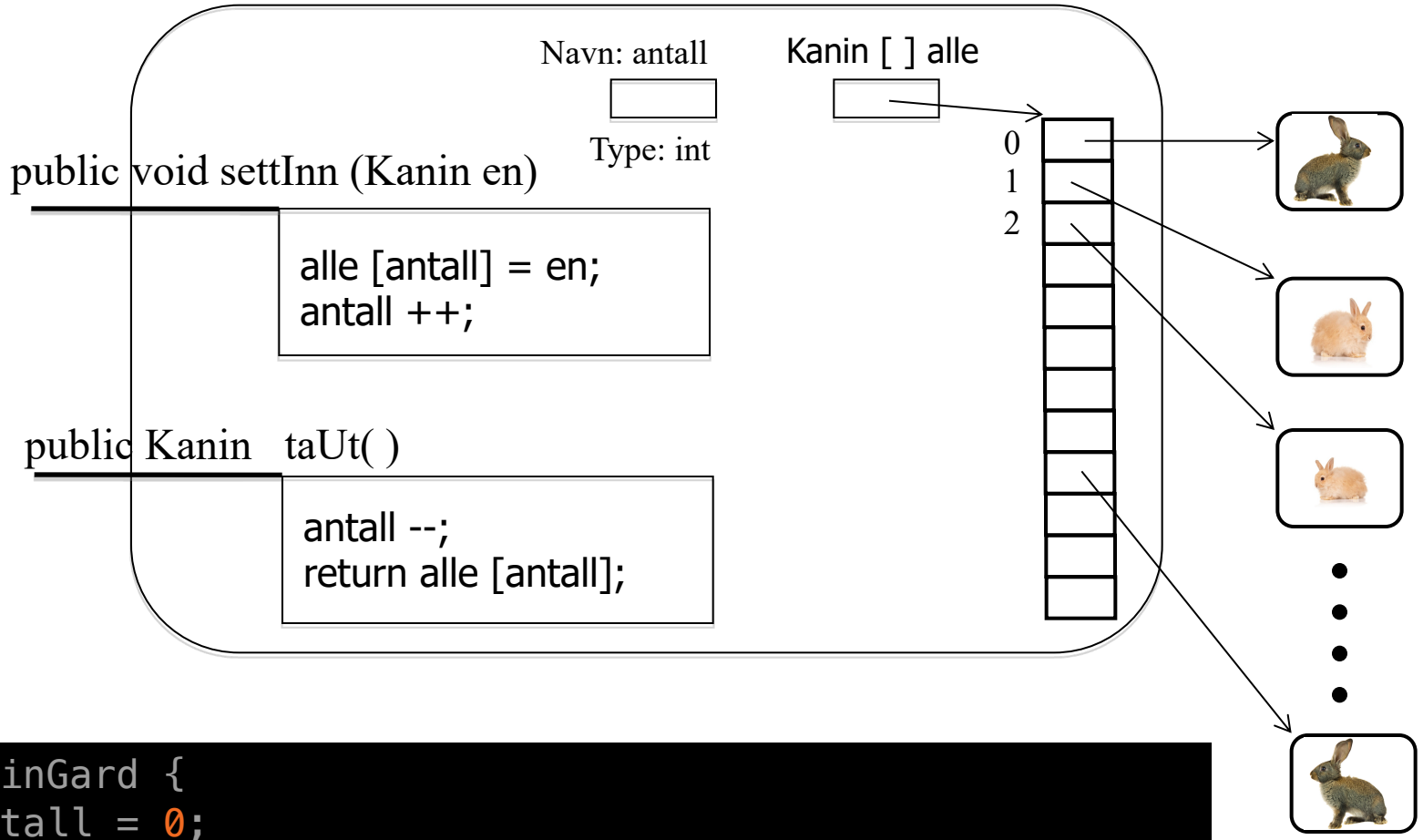
Men at reglen om at vi bare skal ha helt rene observator-metoder og helt rene modifikator-metoder er kanskje å drive det litt langt.
For eksempel public Kanin hentUt(String navn) { . . . }



Veldig forenklet kaningård med litt andre metoder på neste side

Objekt av klassen ForenkletKaninGård





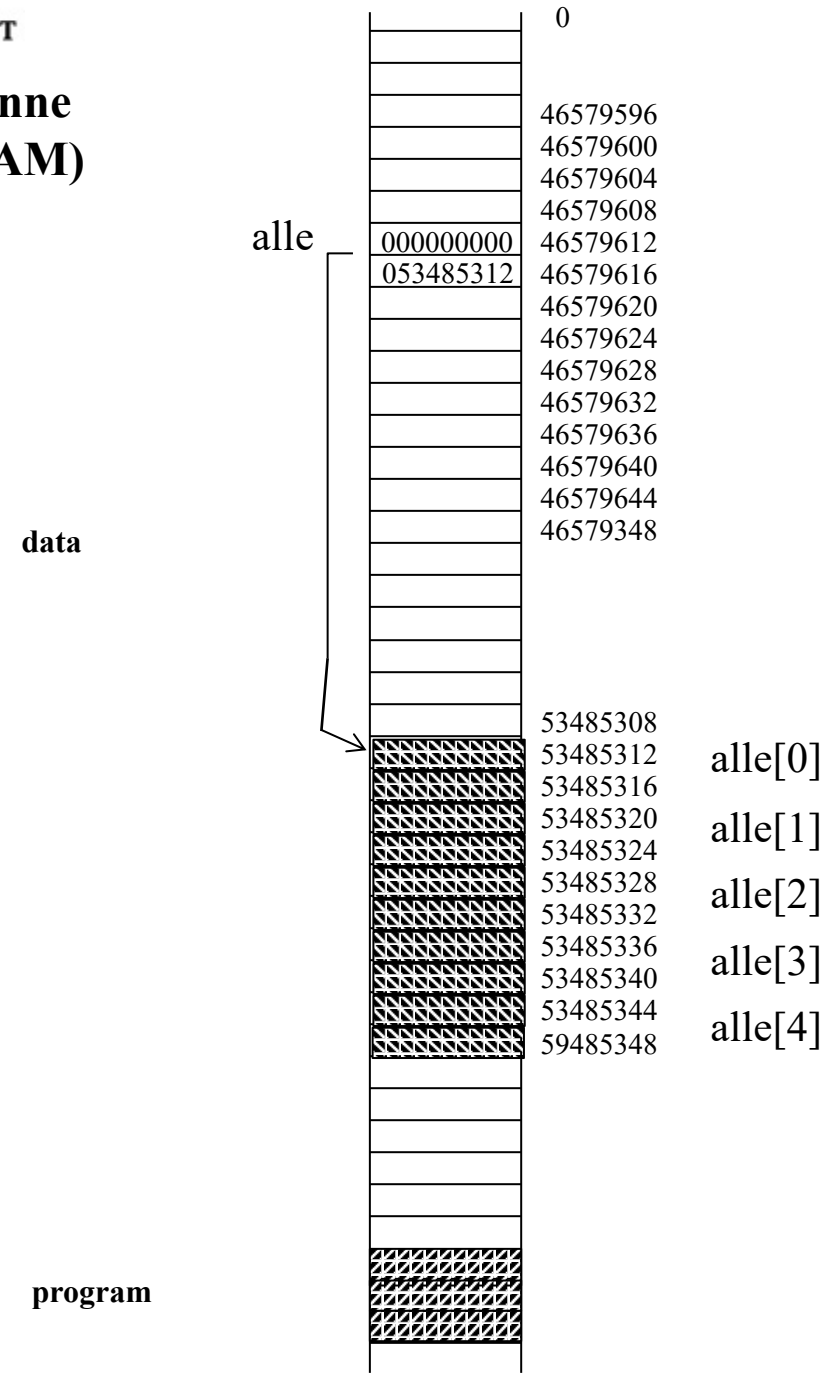

```
class ForenkletKaninGard {
    private int antall = 0;
    private Kanin [ ] alle = new Kanin[100];

    public void settInn(Kanin en) { alle[antall] = en; antall ++; }

    public Kanin taUt( ) { antall --; return alle[antall]; }
}
```

Oppgave:
skriv ferdig klassen KaninGård fra forrige side

Minne (RAM)



Array er iboende i datamaskinen og i Java

Oppslag (indeksering) i en array er veldig. fort gjort av datamaskinen

Husk: Minnet er byte-adresserbart

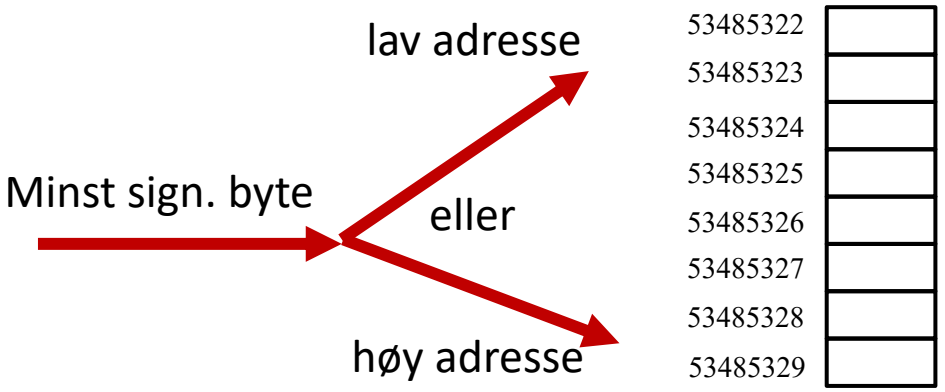
Her har jeg tegnet 4 byte = 32 bit på hver linje og alle er en array av referanser (á 64 bit)

For spesielt interesserte: Rekkefølgen i minnet, «endianness».

boolean	00000001
byte	01001110
char	0100111010110110
short	0100100101101110
int	01010010110110101000100101101110
long	0101001. . . . . .01001101010001001011011 10
float	01010010110110101000100101101110
double	01010010110. . . . . .11010100010010110111 0
referanse	0101000110100110 00100101101110

"Little endian"

i RAM:



"Big endian "

Du kan vel historien om “Gullivers reiser”?

To samlinger (beholdere/collections) fra Javas bibliotek:
ArrayList og **HashMap**

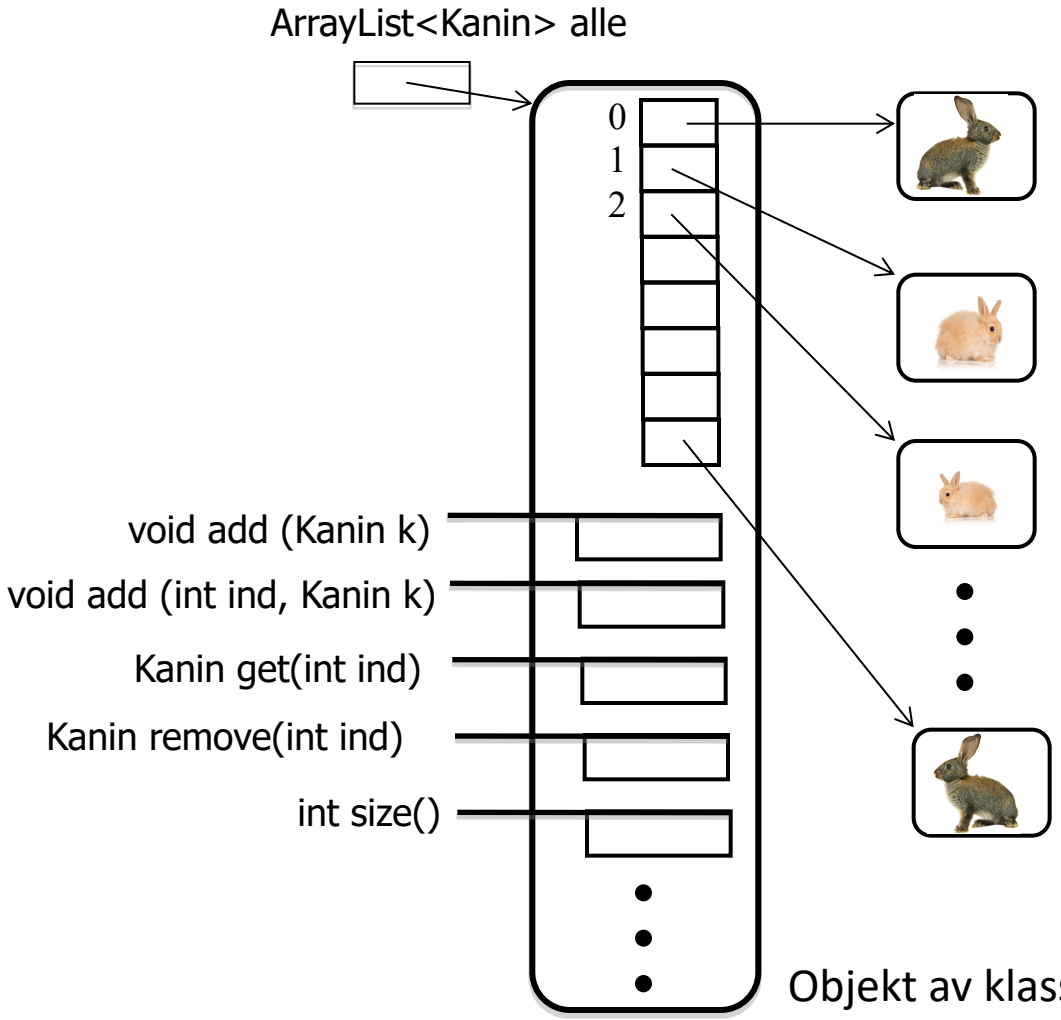
- ArrayList er en fleksibel array som utvider seg og trekker seg sammen etter behov
- ArrayList <Kaniner> mineKaniner = new ArrayList <Kaniner> ();
- Metoder: add, get, remove, . . . se Java-biblioteket

- HashMap er en samling der elementene identifiseres ved en nøkkel / navn
- HashMap<String,Kaniner> alleKaninene = new HashMap<String, Kaniner> ( );
- Metoder: put, get, remove, . . . se Java-bilblioteket

ArrayList

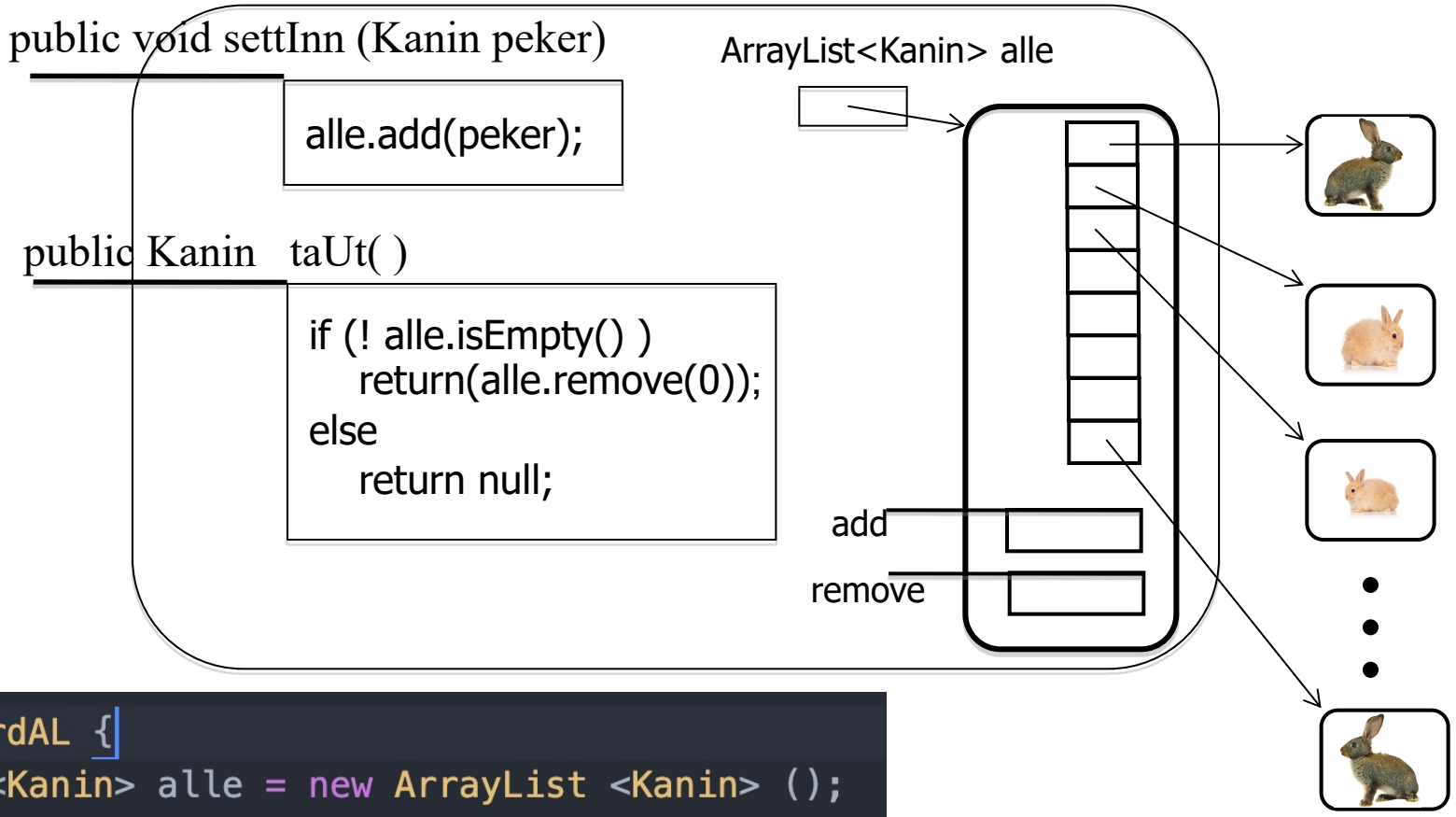
Deklarasjon:

```
ArrayList<Kanin> alle = new ArrayList< >( );
```



Objekt av klassen ArrayList

Objekt av klassen ForenkletKaninGardAL




```
class ForenkletKaninGardAL {
    private ArrayList <Kanin> alle = new ArrayList <Kanin> ();
    public void settInn(Kanin peker) {alle.add(peker);}
    public Kanin taUt() {
      if (! alle.isEmpty()) {return (alle.remove(0));}
      else  return null;
    }
}
```

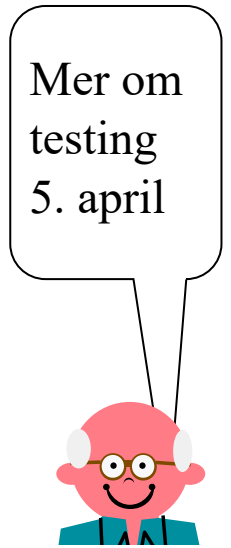



```java
import java.util.ArrayList;

class Kanin{
    private String navn;
    Kanin(String nv) {navn = nv;}
    public String hentNavn ( ) {return navn;}
}

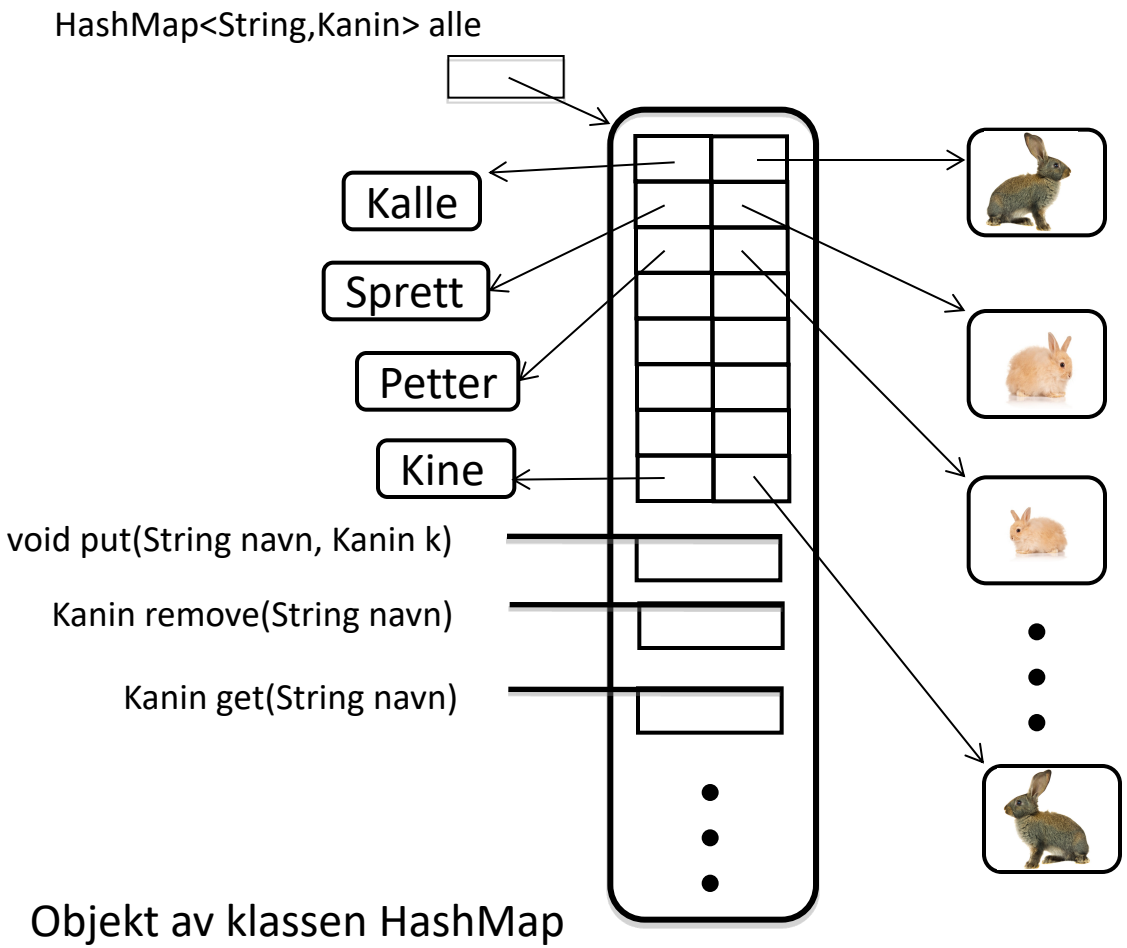
class ForenkletKaninGardAL {
    private ArrayList <Kanin> alle = new ArrayList <Kanin> ();
    public void settInn(Kanin peker) {alle.add(peker);}
    public Kanin taUt() {
      if (! alle.isEmpty()) {return (alle.remove(0));}
      else  return null;
    }
}
```

```java
class  KaningardTestArrayList {
  public static void main (String [ ] args) {
    ForenkletKaninGardAL mittKaninbur = new ForenkletKaninGardAL( );
    Kanin kalle = new Kanin("Kalle");
    mittKaninbur.settInn(kalle);
    Kanin sprett = new Kanin("Sprett");
    mittKaninbur.settInn(sprett);
    // test at først inn kommer først ut
    Kanin enKanin = mittKaninbur.taUt();
      test(((enKanin != null) && enKanin.hentNavn().equals("Kalle")),1);
    // test at neste inn nå kommer ut
    enKanin = mittKaninbur.taUt( );
      test(((enKanin != null) && enKanin.hentNavn().equals("Sprett")),2);
    // test at buret nå er tomt
    enKanin = mittKaninbur.taUt();
      test((enKanin == null),3);
  }
   static void test(boolean riktig, int testNr) {
      if (riktig) {
          System.out.println("Riktig test nummer " + testNr);
      }  else {
        System.out.println("Feil test nummer " + testNr);
    }
  }
}
```



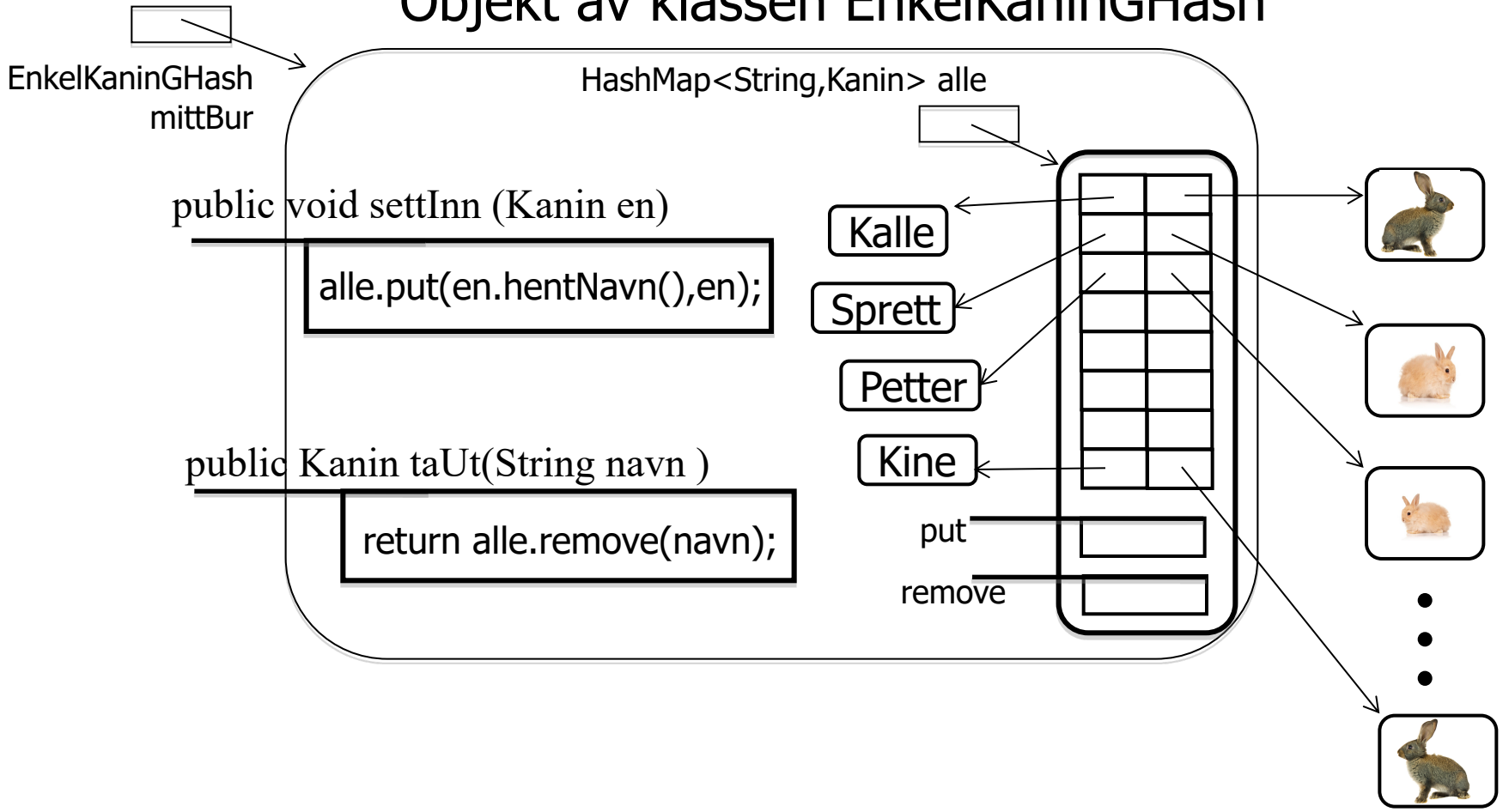
HashMap

Deklarasjon:
HashMap<String,Kanin> alle = new HashMap<String, Kanin>( );



Objekt av klassen HashMap

Objekt av klassen EnkelKaninGHash




```
class EnkelKaninGHash {
      private HashMap <String, Kanin> alle = new HashMap < >();
      public void settInn(Kanin en) {alle.put(en.hentNavn(), en);}
      public Kanin taUt(String navn ) { return alle.remove(navn); }
}
```




```java
import java.util.HashMap;

class Kanin{
  private String navn;
  Kanin(String nv) {navn = nv;}
  public String hentNavn() {return navn;}
}

class EnkelKaninGHash {
      private int antall = 0;
      private HashMap <String, Kanin> alle = new HashMap <String, Kanin>();
      public void settInn(Kanin peker) {alle.put(peker.hentNavn(), peker);}
      public Kanin taUt(String navn) {return alle.remove(navn); }
}
```

```java
class  KaningardTestHash {
  public static void main (String [ ] args) {
    EnkelKaninGHash mittBur = new EnkelKaninGHash();
    Kanin kalle = new Kanin("Kalle");
    mittBur.settInn(kalle);
    Kanin sprett = new Kanin("Sprett");
    mittBur.settInn(sprett);
    // Tester at en som er satt inn kan tas ut
    Kanin enKanin = mittBur.taUt("Kalle");
      test (((enKanin != null) &&  enKanin.hentNavn().equals("Kalle")), 1);
    // Tester at den som er tatt ut virkelig er ute
    enKanin = mittBur.taUt("Kallle");
      test ((enKanin == null),2);
    // . . .
  }
   static void test(boolean riktig, int testNr) {
      if (riktig) {
          System.out.println("Riktig  test nummer " + testNr);
      }  else {
        System.out.println("Feil test nummer " + testNr);
    }
  }
}
```

Enhetstesting - oppsummering

- Først: Tenk, tenk og tenk når du programmerer
- Små deler av programmet må testes før de settes sammen til et større program
- En slik liten del kan være et objekt / en klasse
- Intern testing vs. ekstern testing (grensesnitt vs. implementasjon)
- Du bør særlig teste grensetilfeller
  - F.eks: Ta ut av tom beholdere – sette inn i full beholder
- Du bør teste at alle metoder som forandrer tilstanden til objektet virker slik du ønsker – men umulig å teste alle tilfeller
    - Kanskje skrive ut alle private variable før og etter et slikt metodekall (toString())
- Du bør teste at alle observator-metodene observerer det du ønsker

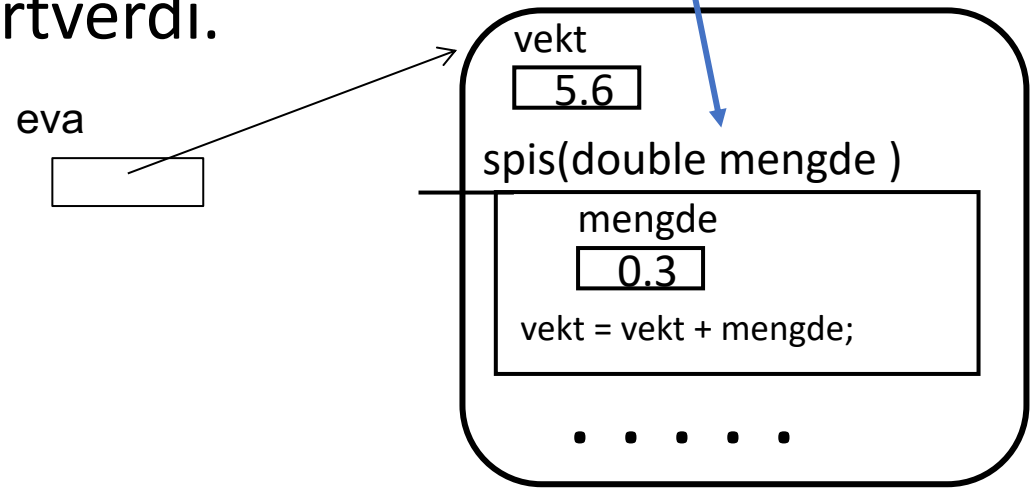
Parameteroverføring

- En parameter i en metodedeklarasjonen kalles en **formell parameter**
  - En del av signaturen til metoden
- Den **aktuelle parameteren** på kallstedet er et **uttrykk** som evalueres til en verdi i det metoden kalles

- Denne verdien tas med til metoden og det opprettes en **metodeinstans** som inneholder den formelle parameteren som lokal variabel og med denne verdien som startverdi.

```
double fisk = 0.2;
double salat = 0.1;

eva.spis(fisk+salat);
```



Parameteroverføring i Java

- Gjelder også for parametre til konstruktører
- Alltid "by value" (verdien av uttrykket tas med til metodeinstansen)
- Referanser/objekter blir da "by reference" (en referanseverdi tas med til metodeinstansen)

Med andre ord:

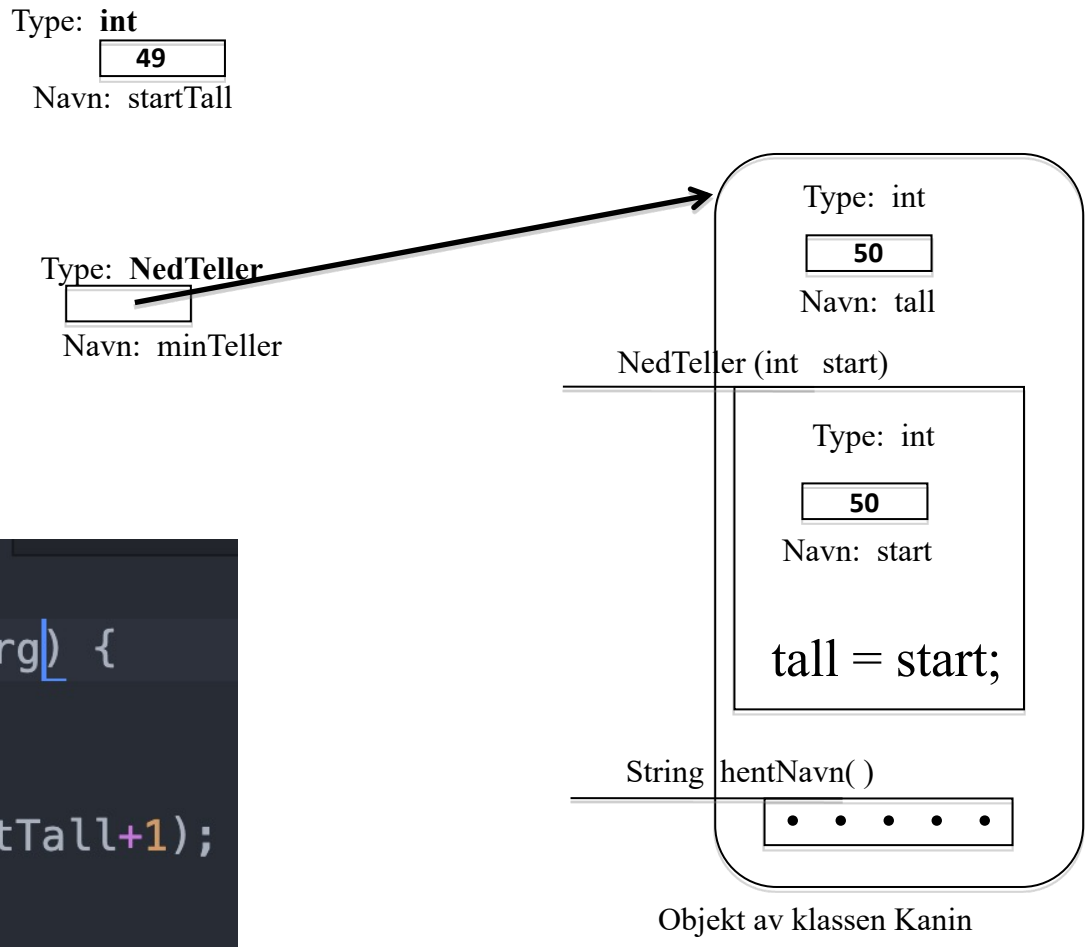
1. Regn ut verdien av uttrykket på kallstedet (aktuell parameter)
2. Opprett en metodeinnstans og ta med denne verdien til metoden
3. La den formelle parameteren i metoden bli en lokal variabel i metoden
4. Verdien som er tatt med blir startverdien til den lokale variablen

To eksempler til på de neste sidene. En konstruktør tar parametre akkurat som en vanlig metode.

Flere eksempler: Parameteroverføring - int

```
class NedTeller {
  private int tall;
  public NedTeller(int start) {
        tall = start;
    }
    public void tellNed() {
      tall-- ;
    }
    public int hentTall()  {
      return tall;
    }
}
```

```
class BrukNedTeller {
    public static void main(String[ ] arg) {
          int startTall = 49;
          NedTeller  minTeller;
          minTeller = new NedTeller(startTall+1);
    }
}
```

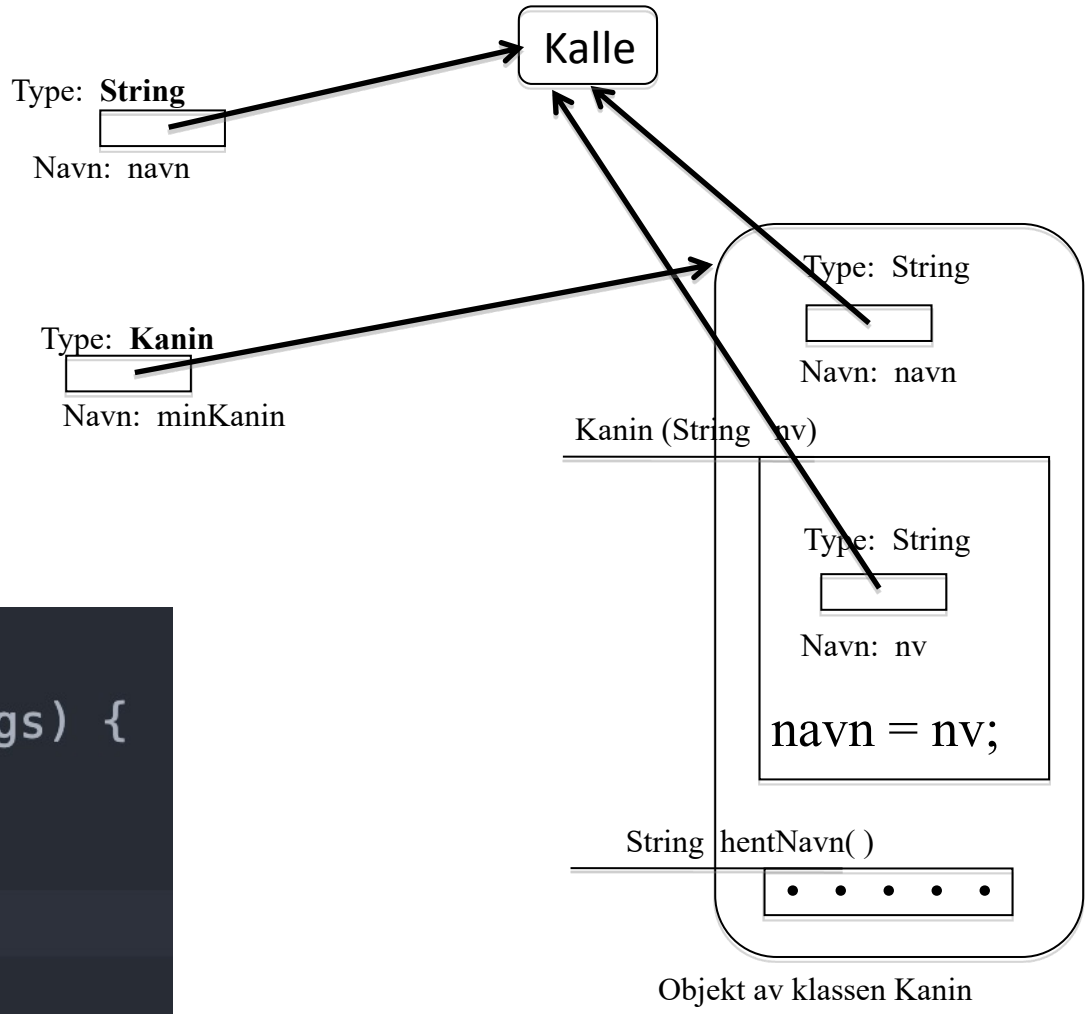


Nå har dere sett så mange klassedatastrukturer med main-metoden, så nå tegner vi det litt enklere

Parameteroverføring - referanse

```
class Kanin{
      private String navn;
      public Kanin(String nv) {
            navn = nv;
      }
      public String hentNavn()  {
          return navn;
      }
}
```

```
class BrukKanin {
    public static void main(String[]args) {
        String navn = "Kalle";
        Kanin minKanin;
        minKanin = new Kanin(navn);
    }
}
```



Nå har dere sett så mange klassedatastrukturer med main-metoden, så nå tegner vi det litt enklere

Oppsummering

- Enhetsestesting er viktig i programmering
  - Testing er viktig i tillegg til nøyaktig programmering.
- Nå kan dere «alt» om klasser og objekter (og typer og variabler)
  - Etter denne uken må dere kunne Java like bra som dere kan Python (Horstmann tom. kap 8)
- Dere kjenner til og kan bruke arrayer (innebygget i Java (og i datamaskiner)) og HashMap og ArrayList fra Javabiblioteket.

- Trøst: Neste uke begynner vi «sakte» på subklasser og arv.

Enda en trøst (?)

Jo før du blir hengende etter -

jo mer tid har du til å ta igjen det tapte



IN1010 våren 2022

Tirsdag 1. februar

Arv og subklasser – del 1

Stein Gjessing.

Agenda

- Subklasser
  - Objekter av subklasser
- Referanser til objekter av subklasser
  - Casting / typekonvertering (sterk typing)
  - “instanceof”
- class Object
- Abstrakte klasser

Ukens tema: Arv og subklasser I

- Objekter bruker vi til å modellere den virkelige verden inne i datamaskinen.
  - Pluss strukturere selve programmet vår (programvarearkitektur)
- Å programmere vil si å strukturere og arrangere begreper og lage ryddige, oversiktlige og utvidbare komponenter, moduler og modeller
- Til dette bruker vi bl.a. arv og subklasser
- Noen problemer som kan løses med subklasser kan også løses (bedre ?) på andre måter
  - Bla. a. ved hjelp av delegering (minner om komposisjon)
  - Mer om dette senere
  - Men uansett må dere være eksperter på subklasser

Kortversjon: Definisjon av subklasser

- En klasse, kalt Klasse, beskriver objekter med visse felles egenskaper
- En subklasse av Klasse, kalt Subklasse, beskriver objekter som har de samme egenskapene (som beskrevet av Klasse), men i tillegg er Subklasse-objektene noe mer, de har flere og / eller mer spesielle egenskaper . . .

```
class Klasse { . . . }

class Subklasse extends Klasse { . . . }
```



Navn på klasser og subklasser (Klasse og Subklasse her) bør best mulig beskrive hva klassene representerer

Eksempel: Universitetsregister

I et mini-system for Universitetet i Oslo skal alle studenter registreres med navn og telefonnummer (åtte siffer), samt hvilket studieprogram de er tatt opp til. Det skal være mulig for studenter å bytte program.

Systemet skal også inneholde informasjon om de ansatte ved universitetet, nemlig navn, telefonnummer, lønnstrinn og antall arbeidstimer per uke. Teknisk-administrativt ansatte har en arbeidsuke på 37,5 timer, mens vitenskapelig ansatte har 40-timers arbeidsuke.

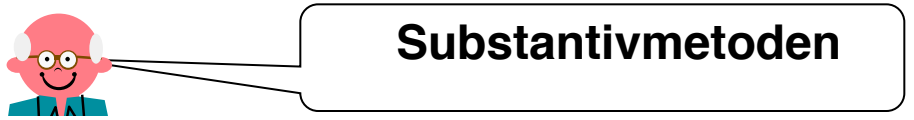
Alle personer skal behandles som . . . . . .

Eksempel: Universitetsregister

I et mini-system for Universitetet i Oslo skal alle studenter registreres med navn og telefonnummer (åtte siffer), samt hvilket studieprogram de er tatt opp til. Det skal være mulig for studenter å bytte program.

Systemet skal også inneholde informasjon om de ansatte ved universitetet, nemlig navn, telefonnummer, lønnstrinn og antall arbeidstimer per uke. Teknisk-administrativt ansatte har en arbeidsuke på 37,5 timer, mens vitenskapelig ansatte har 40-timers arbeidsuke.

Alle personer skal behandles som . . . . . .



Et lite sidesprang

- Når vi skal lage et program må vi velge hva og hvilke deler av den virkelige verden vi skal modellere inne i datamaskinen
- Vi tar bare med det vi trenger
- Vi modellerer bare de aspektene av den virkelige verden som vi trenger for å løse programmets oppgave
- Dette kaller vi gjerne for vårt *perspektiv* på systemet som vi modellerer
- Kristen Nygaard:
  - Et perspektiv er ikke nøytralt (etikk og informatikk)

Modeller av båter



Modeller av fly



Hva er hensikten med modellen?
Hvor nøyaktig skal vi modellere?



Tilbake til vårt system: Universitetsregisteret

I et mini-system for Universitetet i Oslo skal alle studenter registreres med navn og telefonnummer (åtte siffer), samt hvilket studieprogram de er tatt opp til. Det skal være mulig for studenter å bytte program.

Systemet skal også inneholde informasjon om de ansatte ved universitetet, nemlig navn, telefonnummer, lønnstrinn og antall arbeidstimer per uke. Teknisk-administrativt ansatte har en arbeidsuke på 37,5 timer, mens vitenskapelig ansatte har 40-timers arbeidsuke.

Alle personer skal behandles som . . . . . .

Kristen Nygaard



Hva er vårt perspektiv her?

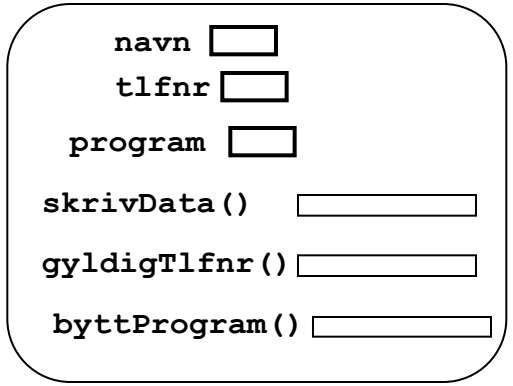
Klassen Student

```
class Student {
    String navn;
    int tlfnr;
    String program;

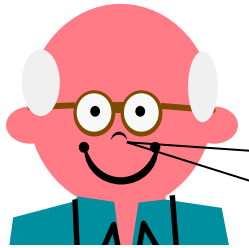
    void skrivData() {
        System.out.println("Navn: " + navn);
        System.out.println("Telefon: " + tlfnr);
        System.out.println("Studieprogram: " + program);
    }

    boolean gyldigTlfnr() {
        return tlfnr >= 10000000 && tlfnr <= 99999999;
    }

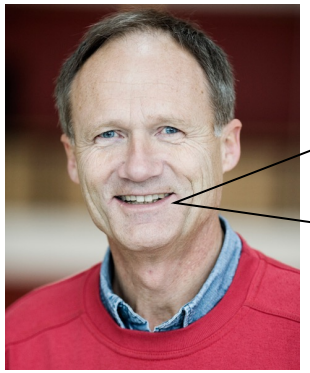
    void byttProgram(String nytt) {
        program = nytt;
    }
}
```



objekt av klassen Student



Hm –
på forrige lysark var det ingen egenskaper som var ”private” eller public” ?



OK –
Vi kommer tilbake til det på lysark 48

Og det er jo slik at om det ikke står noe, så er egenskapene synlige i hele fil-katalogen (package private)

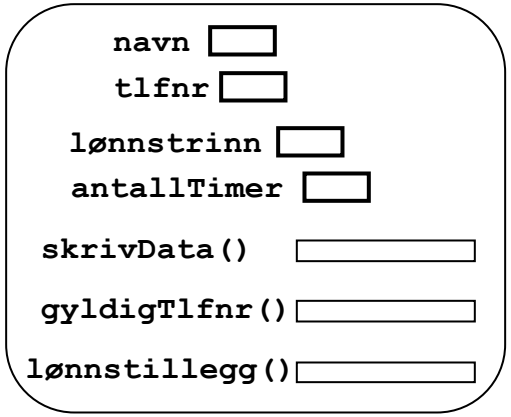
Klassen Ansatt

```
class Ansatt {
    String navn;
    int tlfnr;
    int lønnstrinn;
    int antallTimer;

    void skrivData() {
        System.out.println("Navn: " + navn);
        System.out.println("Telefon: " + tlfnr);
        System.out.println("Lønnstrinn: " + lønnstrinn);
        System.out.println("Timer: " + antallTimer);
    }

    boolean gyldigTlfnr() {
        return tlfnr >= 10000000 && tlfnr <= 99999999;
    }

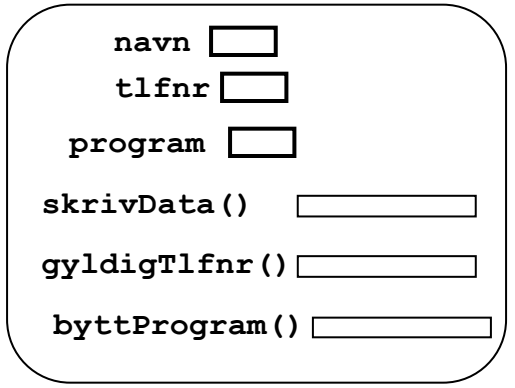
    void lønnstillegg(int tillegg) {
        lønnstrinn += tillegg;
    }
}
```



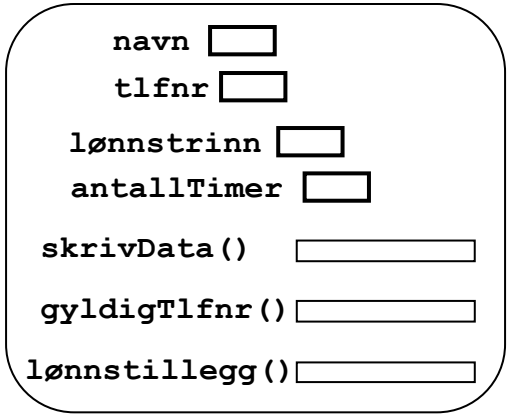
objekt av klassen Ansatt

Student vs Ansatt

- Felles variable:
  - **navn**, **tlfnr**
- Egne variable:
  - Student: **program**
  - Ansatt: **lønnstrinn**, **antallTimer**
- Felles metoder:
  - **gyldigTlfnr()**
- Lignende metoder:
  - **skrivData()**
- Egne metoder:
  - Student: **byttProgram(String nytt)**
  - Ansatt: **lønnstillegg(int tillegg)**



Student-objekt



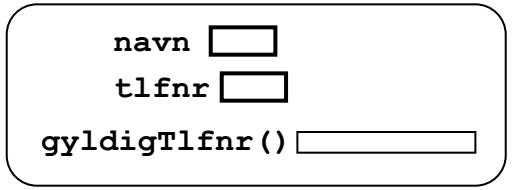
Ansatt-objekt

Klassen Person

Kan samle det som er **felles** i en egen, mer generell, klasse

```
class Person {
    String navn;
    int tlfnr;

    boolean gyldigTlfnr() {
        return tlfnr >= 10000000 && tlfnr <= 99999999;
    }
}
```



Klassen Person beskriver alt som er felles for studenter og ansatte

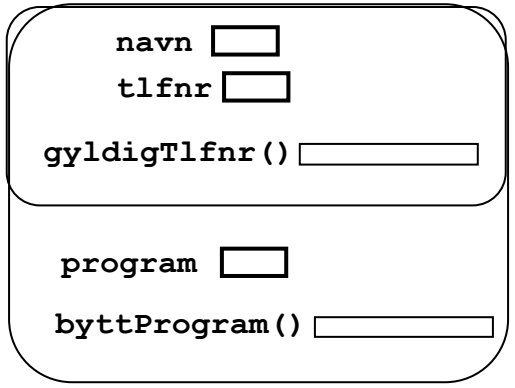
Og det er kanskje ikke helt unaturlig



Student og Ansatt som subklasser av Person


```
class Student extends Person {
    String program;

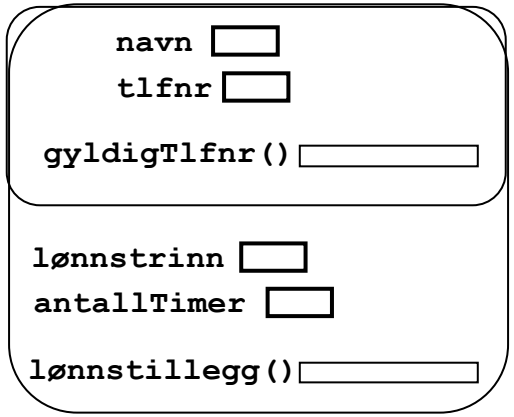
    void byttProgram(String nytt) {
        program = nytt;
    }
}
```



Student-objekt

```
class Ansatt extends Person {
    int lønnstrinn;
    int antallTimer;

    void lønnstillegg(int tillegg) {
        lønnstrinn += tillegg;
    }
}
```



Ansatt-objekt

Student og Ansatt som subklasser av Person


```
class Student extends Person {
    String program;

    void byttProgram(String nytt) {
        program = nytt;
    }
}
```

extends
angir at klassene Student og Ansatt er subklasser (= utvidelser) av klassen Person.

```
class Ansatt extends Person {
    int lønnstrinn;
    int antallTimer;

    void lønnstillegg(int tillegg) {
        lønnstrinn += tillegg;
    }
}
```

Nytt Java nøkkelord:
extends



Hva med skrivData()?
- Kommer tilbake til denne…

```
class Student {
    String navn;
    int tlfnr;
    String program;

    boolean gyldigTlfnr() { . . . }
    void byttProgram(String nytt) { . . . }
}
```

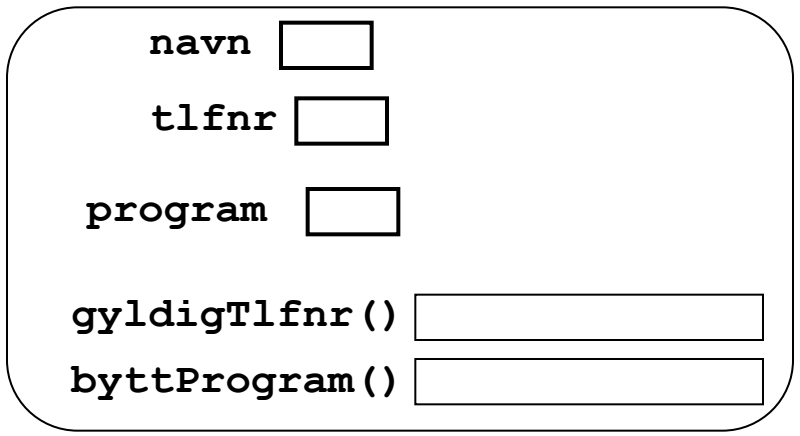
```
class Person {
    String navn;
    int tlfnr;

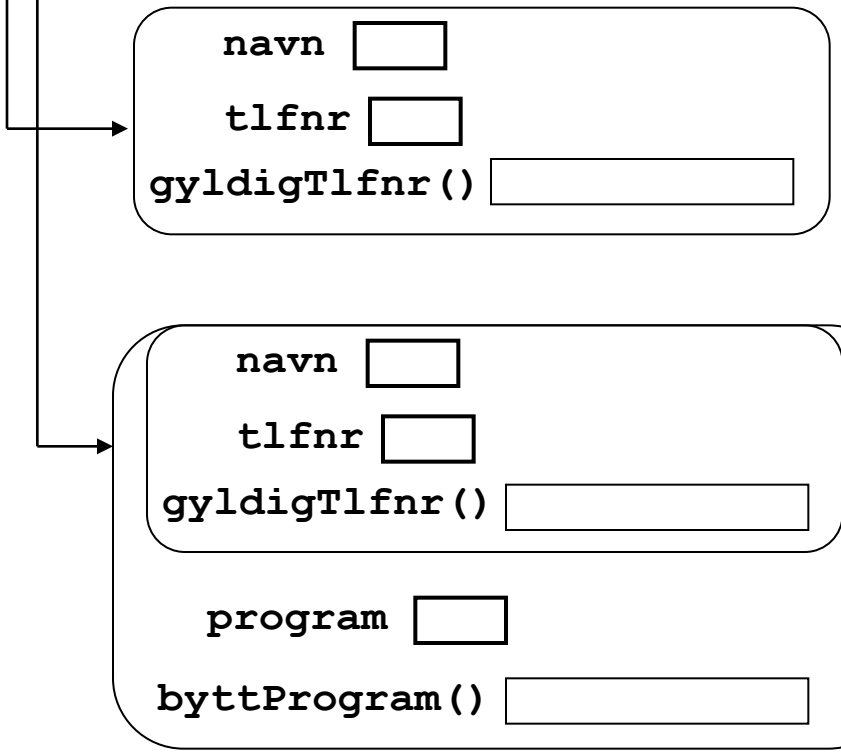
    boolean gyldigTlfnr() { . . . }
}
```

```
class Student extends Person {
    String program;

    void byttProgram(String nytt) { . . .  }
}
```

Eksempler på objekter av klassene





Bruk av en subklasse

Vi kan bruke variable og metoder i en subklasse på samme måte som om vi hadde definert alt i én klasse:

Før:

```
class Student {
    String navn;
    int tlfnr;
    String program;

    boolean gyldigTlfnr() { . . . }
    void byttProgram(String nytt) { . . . }
}
```

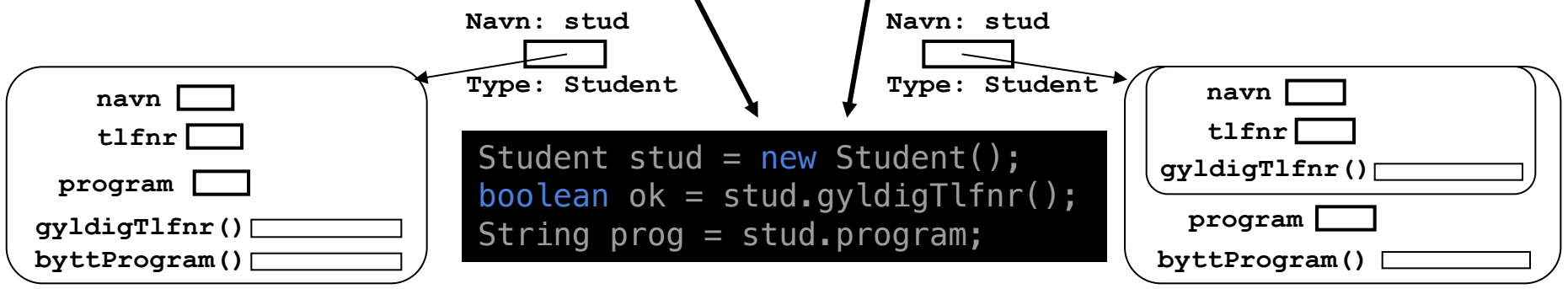
eller med bruk av subklasser (nå):

```
class Person {
    String navn;
    int tlfnr;

    boolean gyldigTlfnr() { . . . }
}
```

```
class Student extends Person {
    String program;

    void byttProgram(String nytt) { . . .  }
}
```



Nytt

Tegning av subklassehierarki


```
class Person {
    String navn;
    int tlfnr;

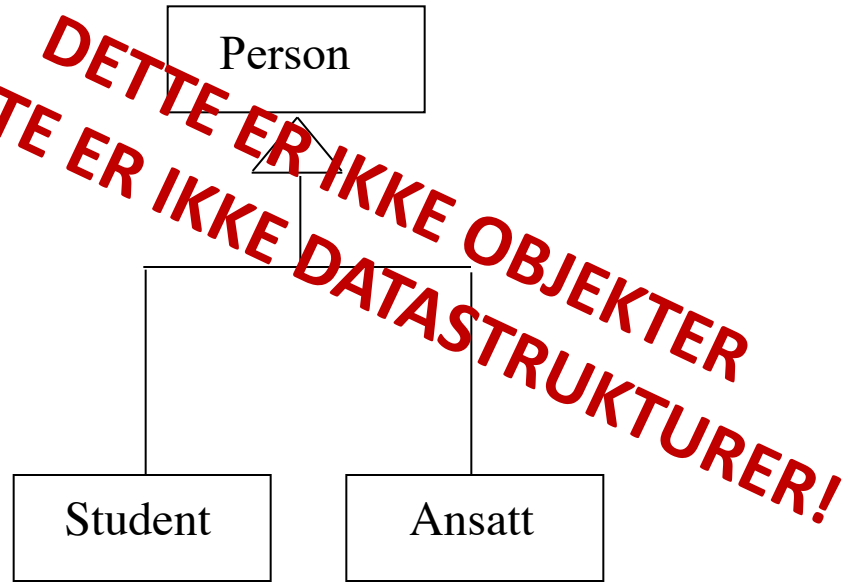
    boolean gyldigTlfnr() { . . . }
}
```

```
class Student extends Person {
    String program;

    void byttProgram(String nytt) { . . .  }
}
```

```
class Ansatt extends Person {
    int lønnstrinn;
    int antallTimer;

    void lønnstillegg(int tillegg) { . . . }
}
```



Det er en tegning av strukturen i problemet
Det er en tegning av strukturen i programmet

I IN1010 er riktige UML-tegninger ikke viktig (men subklassehierarkier er viktig)

Notasjon for **subklassehierarki**

```
class Person {

}
```

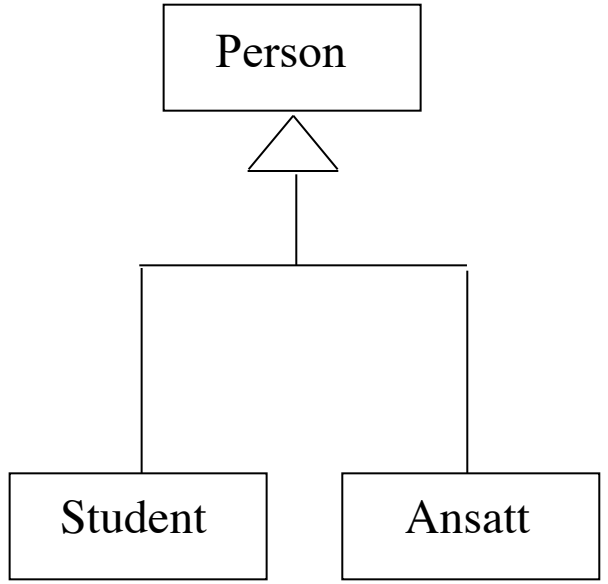
```
class Student extends Person {

}
```

```
class Ansatt extends Person {

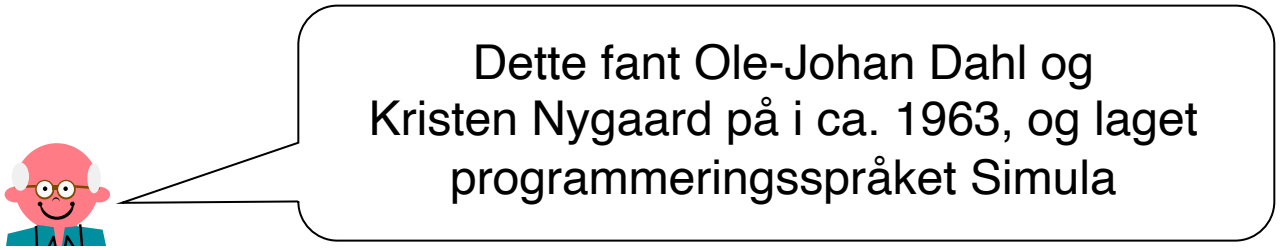
}
```



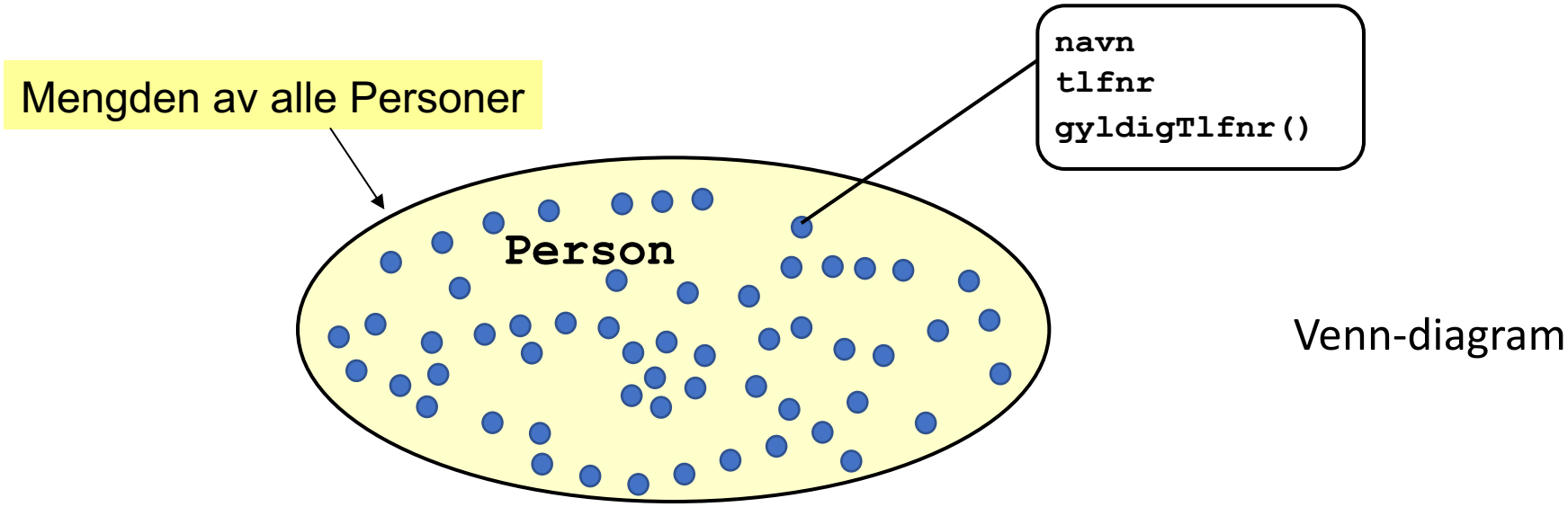


Subklasser og superklasser

- En subklasse er en klasse som bygger på en allerede spesifisert klasse, og som dermed ***arver*** dennes egenskaper i tillegg til å utvide med egne egenskaper (metoder/variabler/konstanter).
- En subklasse er altså en mer **spesialisert** utgave av klassen den bygger på.
- Klassen vi bygger på kalles en ***superklasse***.



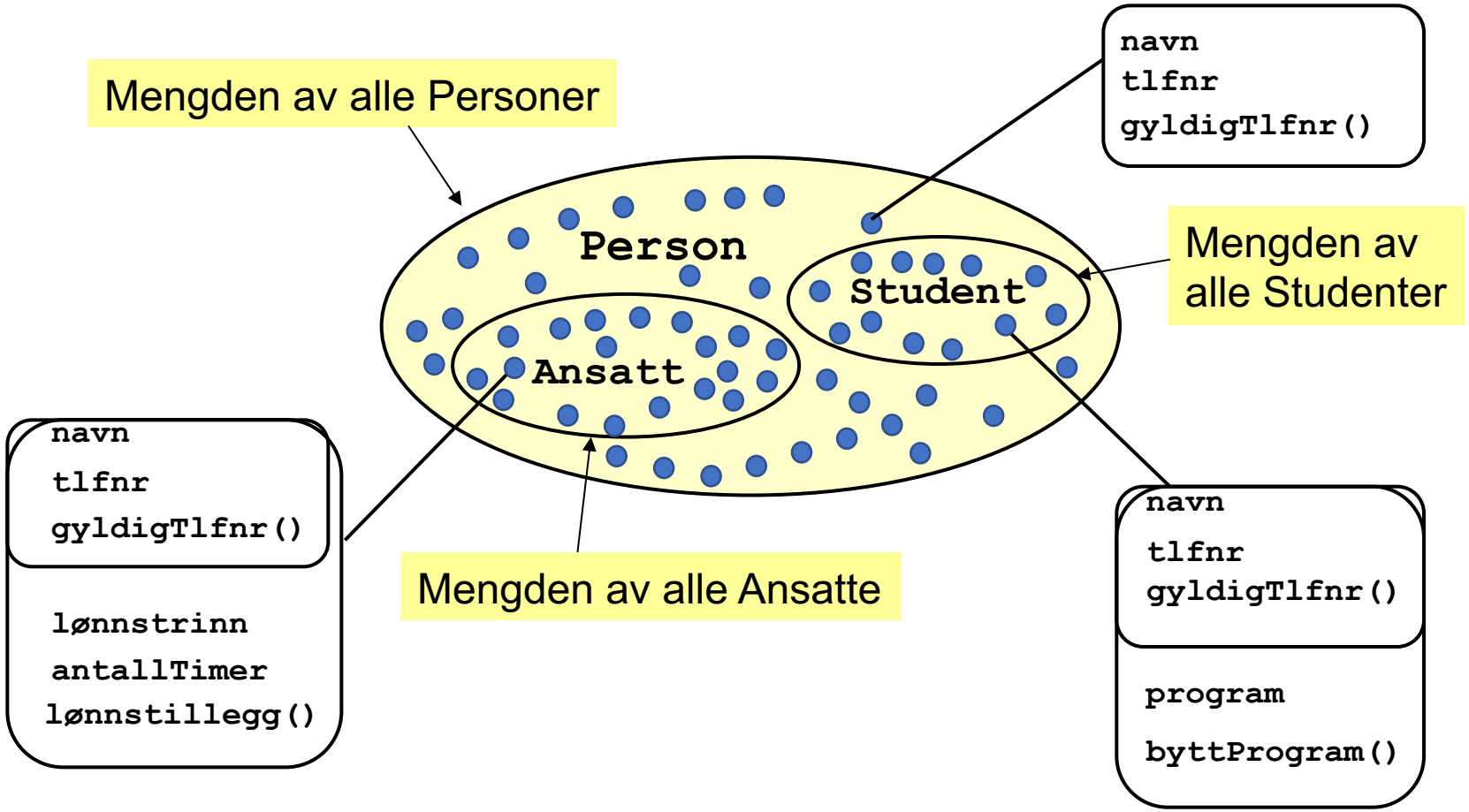
Alle Personene i programmet vårt



Venn-diagram

- Alle de blå prikkene er alle personer vi modellerer (i vårt system, i "vår verden")
- Alle de blå prikkene er alle personer som er med i programmet vårt
- Alle de blå prikkene er alle Person-objektene i programmet vårt
- Alle disse blå prikkene har de samme egenskapene og de er definert av **klassen Person**

Spesialisering - Generalisering



Sub-klasse ~ Sub-mengde (del-mengde)

Klasse-hierarkier

Det er mulig å definere subklasser av en subklasse (etc.):

```
class Person {

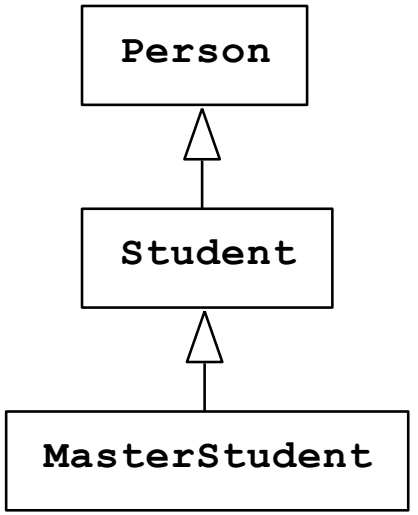
}
```

```
class Student extends Person {

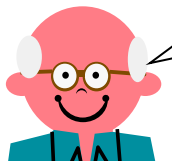
}
```

```
class MasterStudent extends Student {

}
```



Obs: Her er MasterStudent en subklasse av både Student og Person, og arver egenskaper fra begge disse.



NB! KLASSE-HIERARKI, IKKE DATASTRUKTUR !

Klasse-hierarkier


```
class Person {

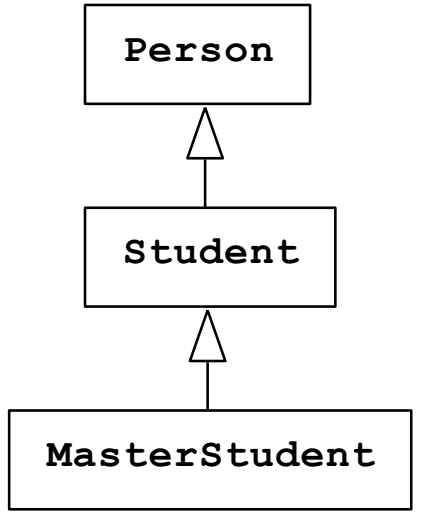
}
```

```
class Student extends Person {

}
```

```
class MasterStudent extends Student {

}
```



- Først modellerer vi virkeligheten og lager et klassehierarki.
- Deretter lager vi programmet.
- Programmets subklassestruktur og klassehierarkiet uttrykker det samme.

- Når programmet utføres er tegningen av klassehierarkiet glemt (men det avspeiles jo av programmet)

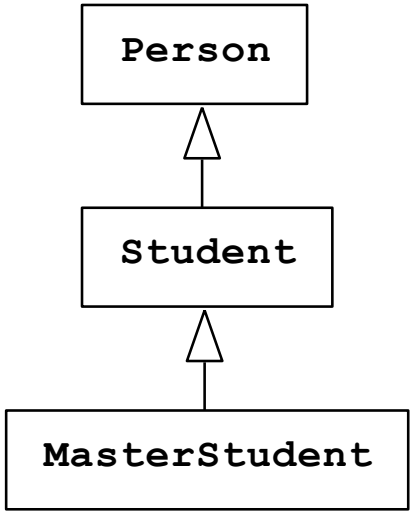
Klasse-hierarkier

Det er mulig å definere subklasser av en subklasse (etc.):

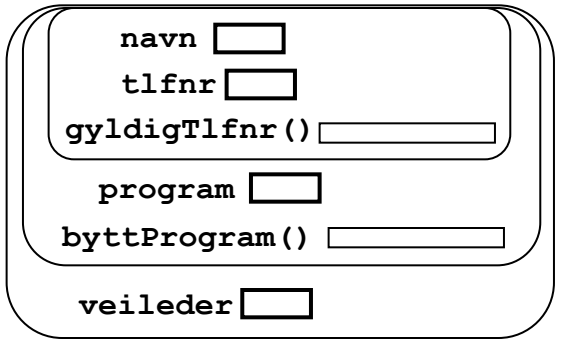
```
class Person {
    String navn;
    int tlfnr;
    boolean gyldigTlfnr() { . . . }
}
```

```
class Student extends Person {
    String program;
    void byttProgram(String nytt) { . . .  }
}
```

```
class MasterStudent extends Student {
    String veileder;
}
```



MasterStudent-objekt

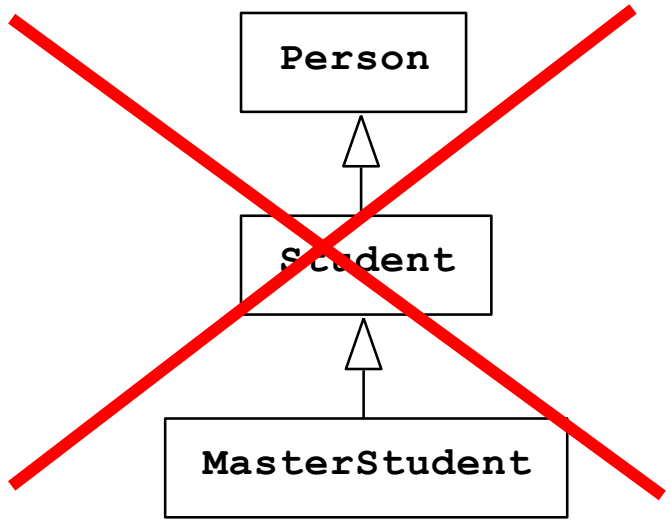


Når programmet utføres lages det datastrukturer / objekter

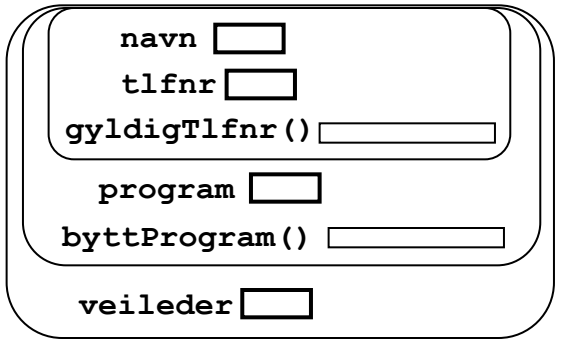

```
class Person {
    String navn;
    int tlfnr;
    boolean gyldigTlfnr() { . . . }
}
```

```
class Student extends Person {
    String program;
    void byttProgram(String nytt) { . . .  }
}
```

```
class MasterStudent extends Student {
    String veileder;
}
```

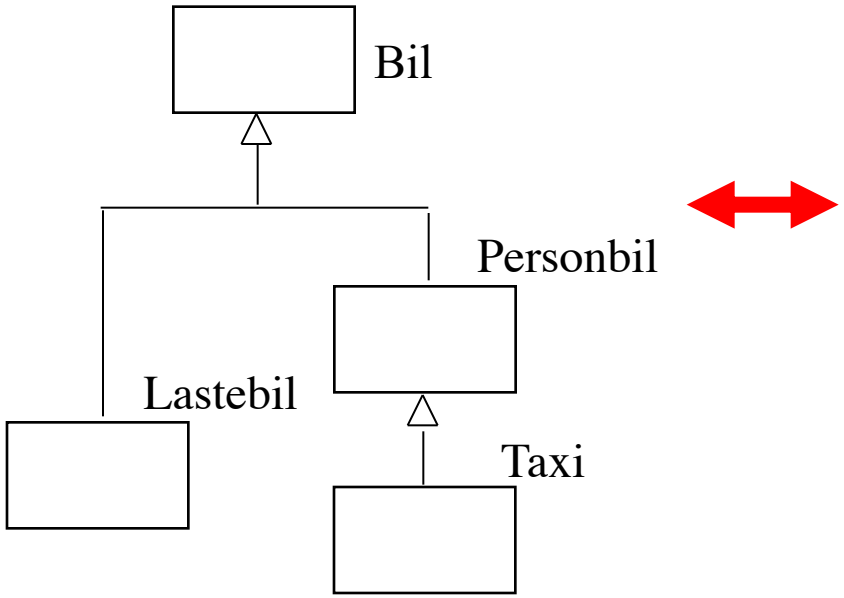


MasterStudent-objekt



Klasser - Subklasser

Klassehierarki:



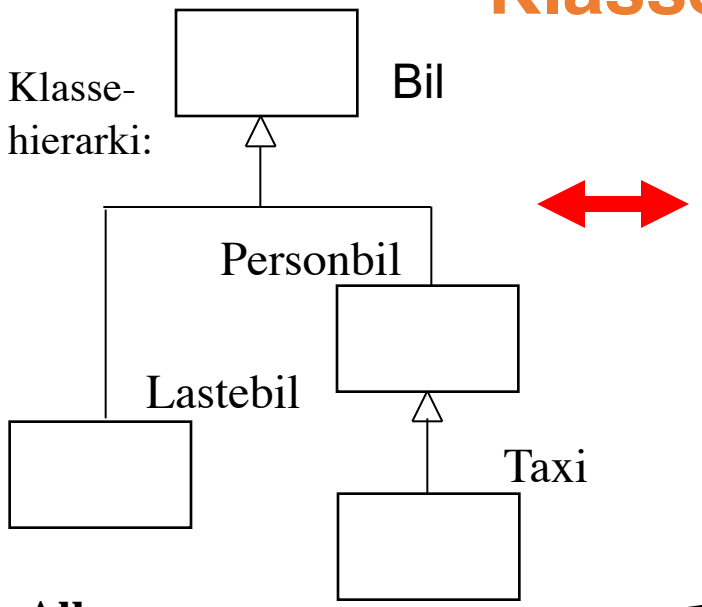

```
class Bil { . . . }

class Personbil extends Bil { . . . }

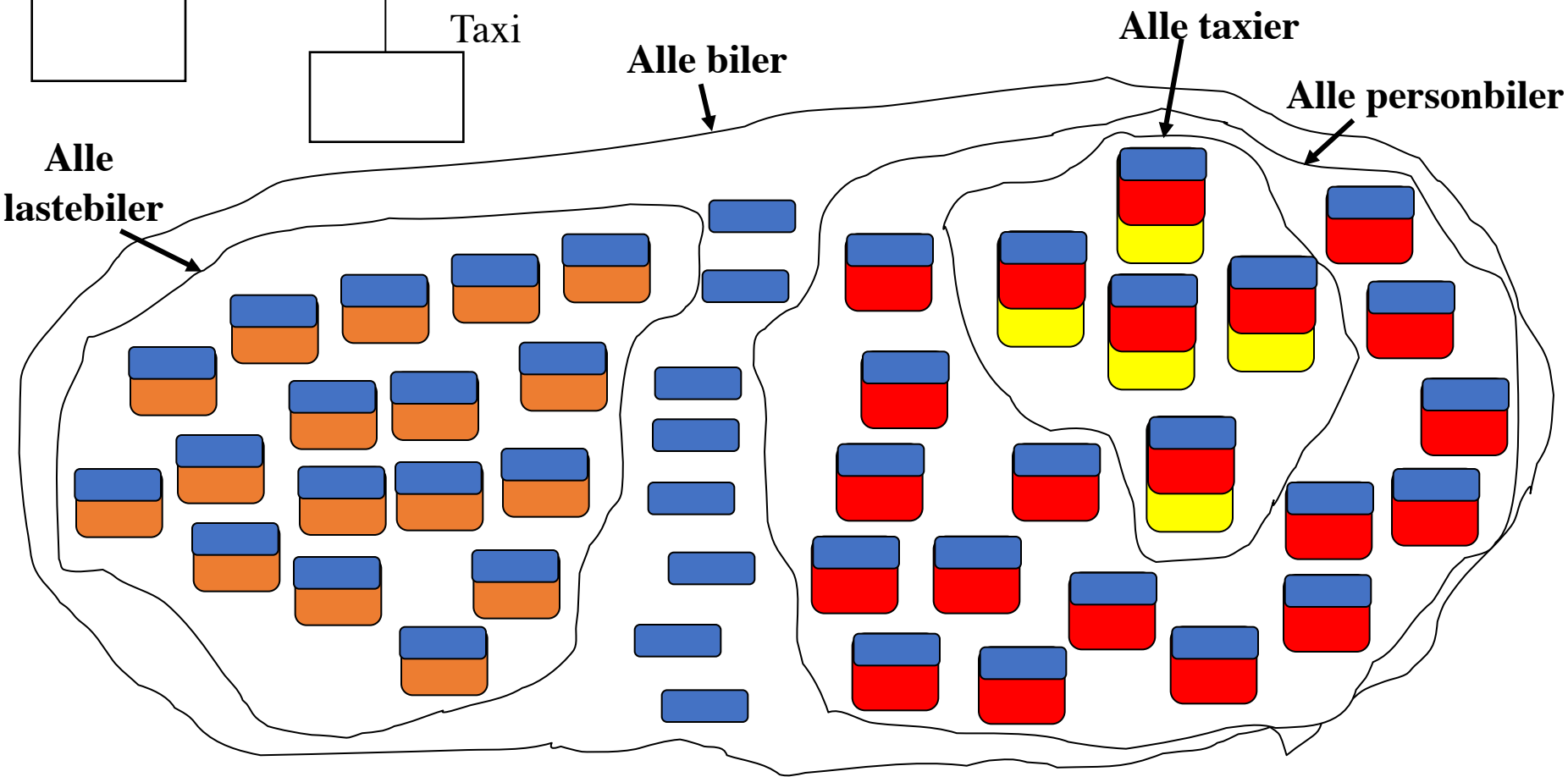
class Lastebil extends Bil { . . . }

class Taxi extends Personbil { . . . }
```

Klasser - Subklasser



class Bil { <blå egenskaper>}
class Personbil extends Bil { <røde egenskaper> }
class Lastebil extends Bil { <brune egenskaper> }
class Taxi extends Personbil { <gule egenskaper> }



Hvorfor bruker vi subklasser?

1. Klasser og subklasser avspeiler **virkeligheten**
   - Bra når vi skal modellere virkeligheten i et datasystem
2. Klasser og subklasser avspeiler **arkitekturen** til datasystemet / dataprogrammet
   - Se f.eks. Javas API
   - Bra når vi skal lage et oversiktlig stort program

3. Klasser og subklasser kan brukes til å forenkle og gjøre programmer mer forståelig, og spare arbeid:
   Gjenbruk av programdeler
   - ”Bottom up” – programmering
     - Lage verktøy
   - ”Top down” programmering
     - Postulere verktøy

Mer om dette neste uke



Referanser og subklasser

- Nå vet vi litt om subklasser
- Vi vet bl.a. at objekter av subklasser inneholder flere “deler”

- Java er et sterkt typet språk!

  - Hva har det å si for referanser til slike oppdelte subklasse-objekter?

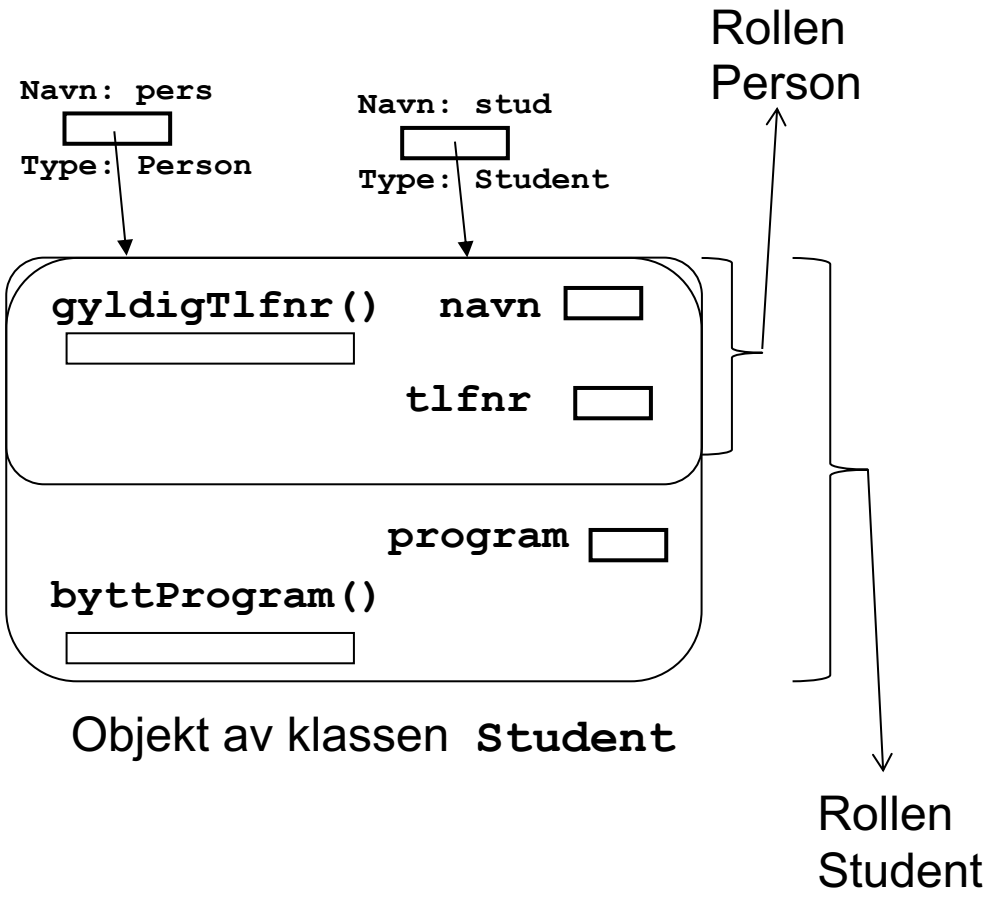
NYTT

Ulike referansetyper (pekertyper)

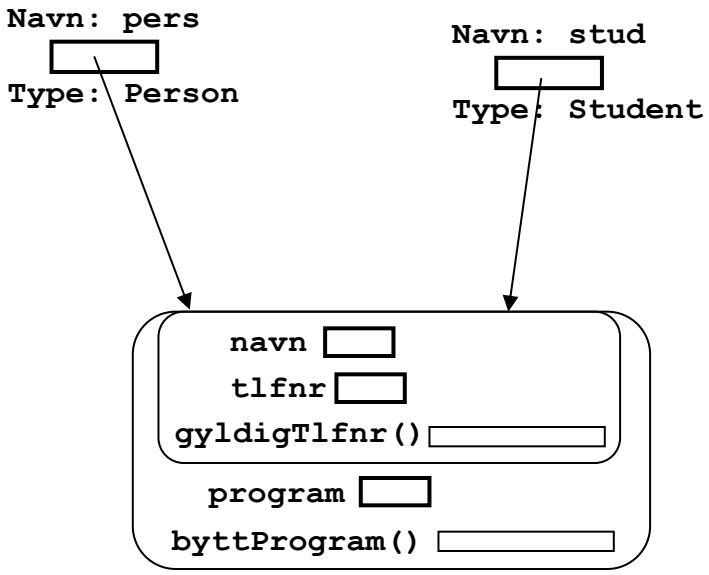
```
Person pers;
Student stud;
stud = new Student();
pers = stud;
```

```
class Person {
    String navn;
    int tlfnr;
    boolean gyldigTlfnr() { . . . }
}
```

```
class Student extends Person {
    String program;
    void byttProgram(String nytt) { . . .  }
}
```



Ulike måter å se et objekt på



Typen (klassen) til hele dette objektet er **Student**

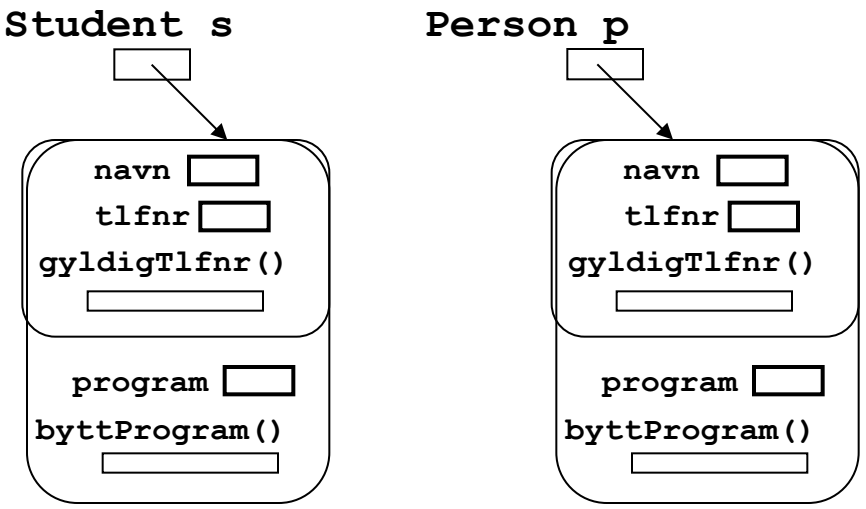
```
Person pers;
Student stud;
stud = new Student();
pers = stud;
```

- **Typen (klassen) til selve objektet er uforanderlig.**
  Et objekt kan likevel ***fremtre*** *for oss* på ulike måter
  Det kan spille forskjellige roller.
- Et objekt av klassen
  `class Student extends Person {...}`
  kan vi se på som et objekt av typen (klassen)
  - **`Person`**: da er egenskapene som er spesielle for Student ikke synlige (men de er der fortsatt!).
  - **`Student`**: da er både Person- og Student-egenskapene synlige for oss.
- Det er *referanseverdiens (pekerens) type* som avgjør hvordan objektet fremtrer.
  (med unntak av polymorfe metoder, som vi skal lære om neste uke)

Flere eksempler


```java
class Person {
    String navn;
    int tlfnr;
    boolean gyldigTlfnr() { . . . }
}
```

```java
class Student extends Person {
    String program;
    void byttProgram(String nytt) { . . .  }
}
```



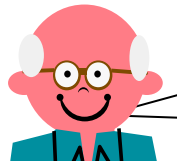

```java
// Anta

Student s = new Student();
Person  p = new Student();

// Hvilke av følgende uttrykk er nå lovlige?

s.navn = ”Ole-Morten";
. . . s.gyldigTlfnr();
s.program = "Matte";
s.byttProgram("Informatikk");

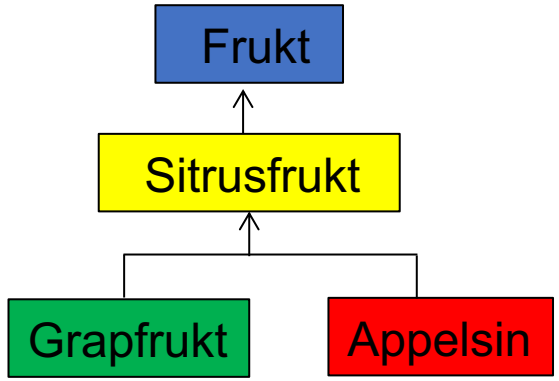

p.navn = ”Ole-Ivar";
. . . p.gyldigTlfnr();
p.program = "Matte";
p.byttProgram("Informtaikk");
```



forskjellige referansetyper =
forskjellige **roller** =
forskjellige **briller**

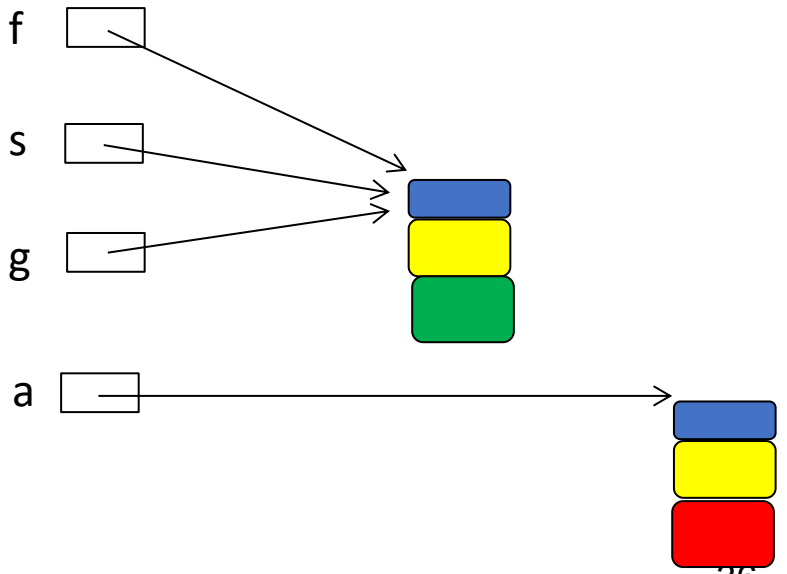
Regel om referanser – Sterk typing


```
class Frukt { .. }
class Sitrusfrukt extends Frukt { .. }
class Grapefrukt extends Sitrusfrukt { .. }
class Appelsin extends Sitrusfrukt { .. }
```



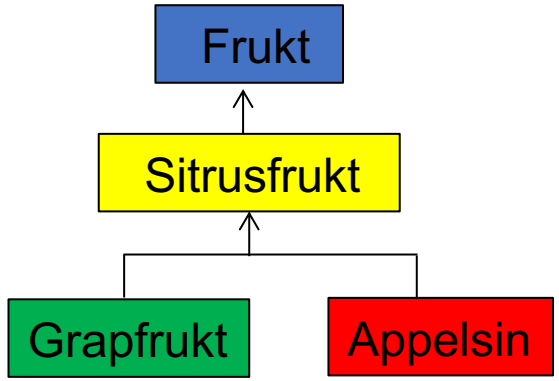
Klassehierarki

En referanse av typen P har bare lov å peke på objekter som har P-egenskaper



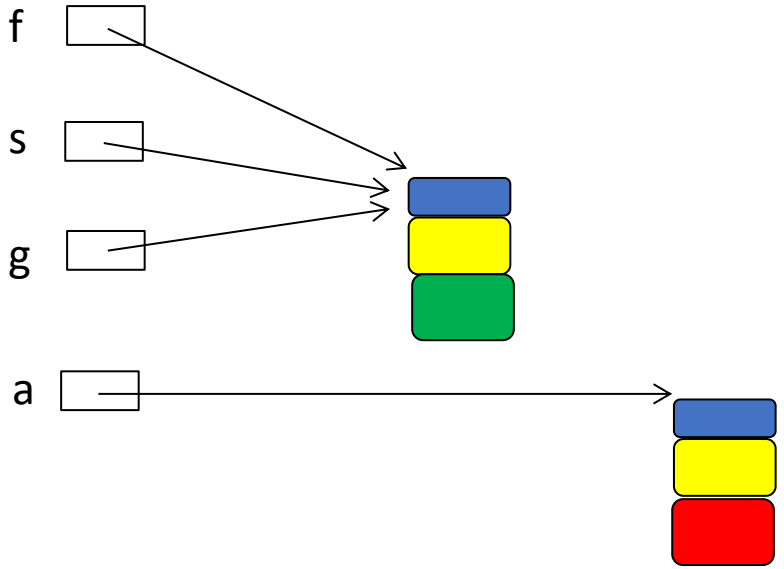
Quiz om referanser til subklasseobjekter


```
class Frukt { .. }
class Sitrusfrukt extends Frukt { .. }
class Grapefrukt extends Sitrusfrukt { .. }
class Appelsin extends Sitrusfrukt { .. }
```



Klassehierarki

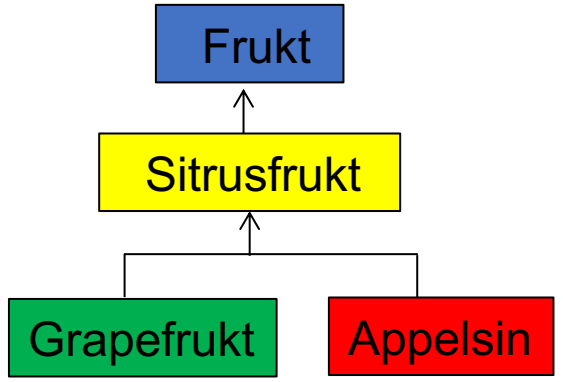
```
Frukt f;
Sitrusfrukt s;
Grapefrukt g;
Appelsin a;
// Hva er lov
g = new Grapefrukt();
f = g;
s = g;
a = new Appelsin();
```



Menti-quiz

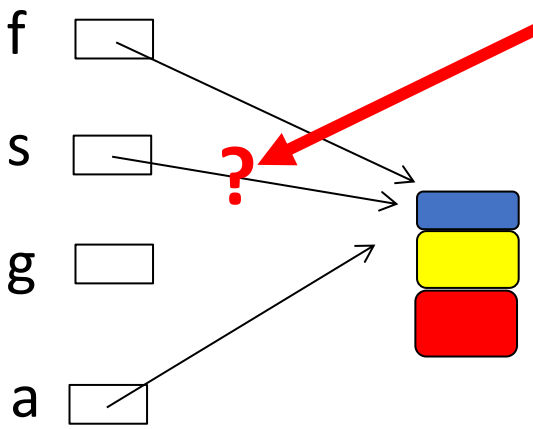
Tilbake til svaret på oppgave 3


```
Frukt f;
Sitrusfrukt s;
Grapefrukt g;
Appelsin a;
// Hva er lov:
a = new Appelsin();
f = a;
g = f;
s = f;
```



Klassehierarki

Men dette kan vi rette på



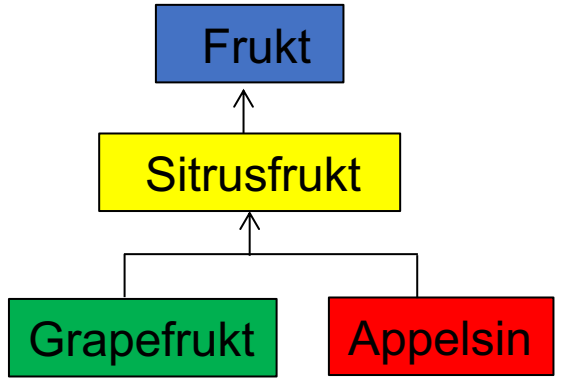
s = f; // Ikke lov

Men f peker jo på et objekt som innholder Sitrusfrukt-egenskaper, så dette burde jo være I orden?

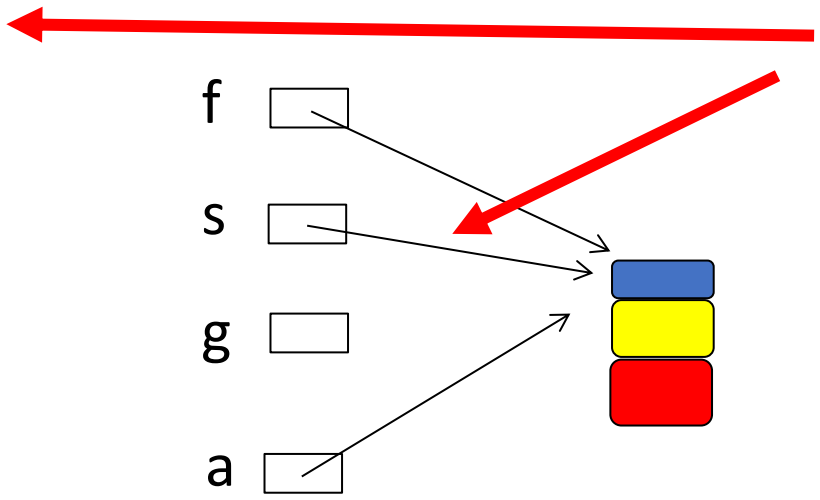
Men det vet ikke kompilatoren !!!

Løsning: La kjøretidsystemet sjekke


```
Frukt f;
Sitrusfrukt s;
Grapefrukt g;
Appelsin a;
// Hva er lov:
a = new Appelsin();
f = a;
g = f;
s = (Sitrusfrukt)f;
```



Klassehierarki

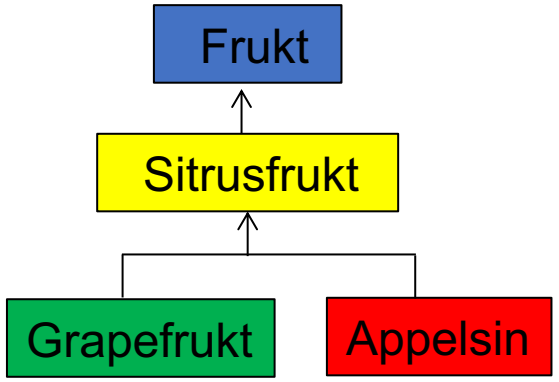


s = f; // Ikke lov
S = (Sitrusfrukt) f; // Er lov

Dette blir en kontroll under kjøring at f virkelig peker på et objekt med Sitrusfrukt-egenskaper

Løsning:
La kjøretidsystemet sjekke og **konvertere** om OK

```
Frukt f;
Sitrusfrukt s;
Grapefrukt g;
Appelsin a;
// Hva er lov:
a = new Appelsin();
f = a;
g = f;
s = (Sitrusfrukt)f;
```



Klassehierarki

s = (Sitrusfrukt) f;

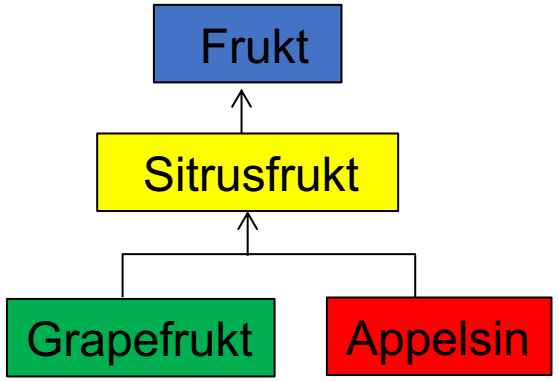
Svaret på uttrykket f er en verdi av typen Frukt.

Denne verdien kan vi **typekonvertere / "caste"** til en verdi av typen Sitrusfrukt.

Etter dette kan vi legge en verdi av riktig type ned i variable s.

Løsning: La kjøretidsystemet sjekke


```
Frukt f;
Sitrusfrukt s;
Grapefrukt g;
Appelsin a;
// Hva er lov:
f = new Frukt();
s = (Sitrusfrukt)f;
```

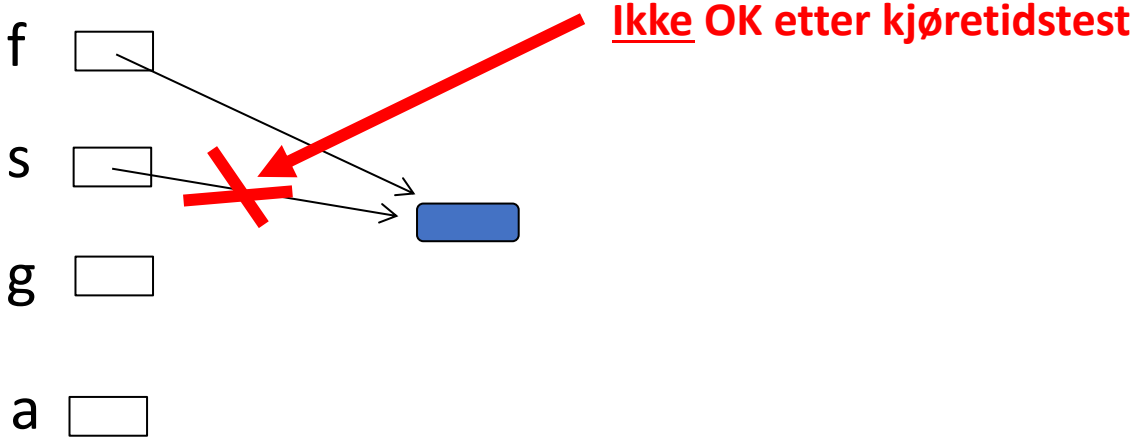


Klassehierarki

Dette blir en kontroll under kjøring om f virkelig peker på et objekt med Sitrusfrukt-egenskaper

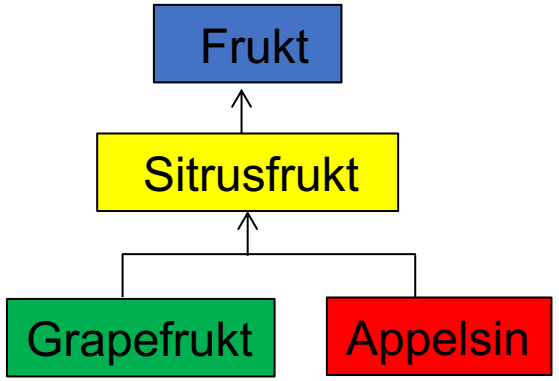
Og her er **det** ikke OK: Programmet stopper

Bruk: **ClassCastException**



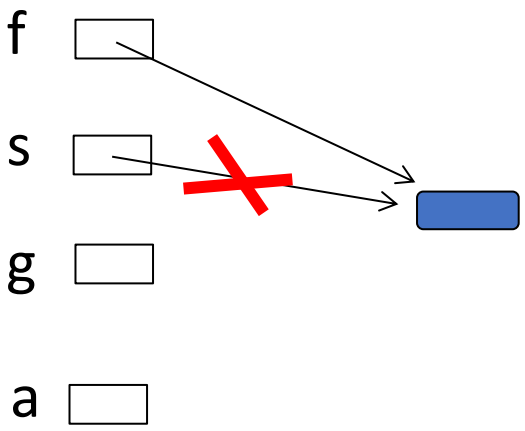
Løsning: instanceof -nytt Java nøkkelord


```
Frukt f;
Sitrusfrukt s;
Grapefrukt g;
Appelsin a;
// Hva er lov:
f = new Sitrusrukt();
 . . .
if (f instanceof Sitrusfrukt)
    { s = (Sitrusfrukt)f; }
```



Klassehierarki

If-testen regnes ut til «false» og da vil tilordningen til s ikke bli utført

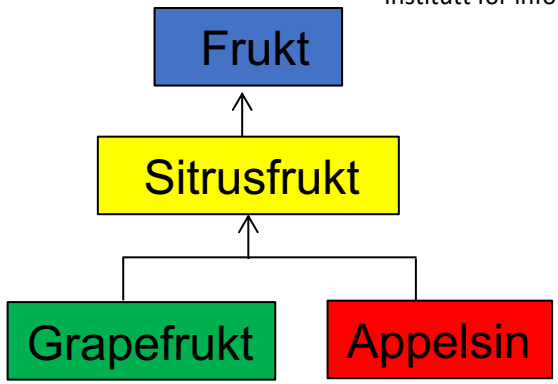


Hva slags objekt er dette?

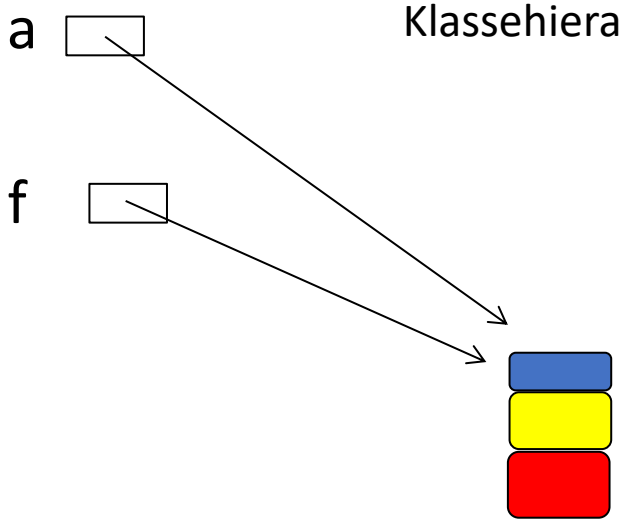
Ved inngangen til metoden vet referansevariablen f ikke annet enn at den peker på en Frukt.

Men hva slags objekt er det virkelig.

```
class TestFrukt {
    public static void main(String[] args) {
        Appelsin a = new Appelsin();
        skrivUt(a);
     }
     static void skrivUt(Frukt f) {
        if (f instanceof Grapefrukt)
           System.out.println("Dette er en grapefrukt");
        else if (f instanceof Appelsin)
        System.out.println("Dette er en appelsin!");
     }
}
```



Klassehierarki



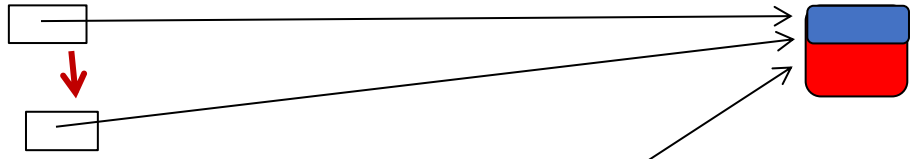
Flere eksempler på konvertering av referanser

- Anta at vi har:

```
class Student extends Person {…}
Student stud = new Student();
```

- Ved tilordningen

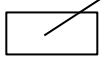
```
Person pers;
pers = stud;
```



har vi en **implisitt typekonvertering** fra Student- til Person-referanse.

- Hvis vi nå ønsker å få tak i de spesielle Student-egenskapene, må vi foreta en **eksplisitt typekonvertering** tilbake til Student igjen:

```
Student stud2 = (Student) pers;
```

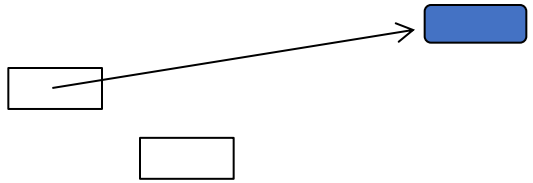


Typekonvertering: "Casting" på engelsk

Flere eksempler på konvertering (forts.)

- Hva hvis vi har:

```
Person pers = new Person();
Student stud = (Student) pers;
```

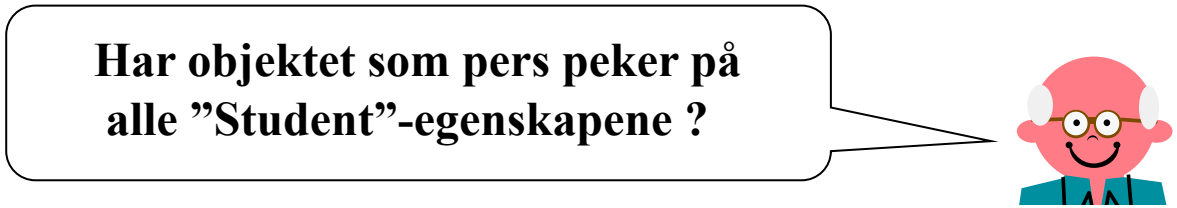


- Dette godkjennes av kompilatoren, men ved kjøring går det galt, og vi får feilmeldingen

```
java.lang.ClassCastException
(fordi pers ikke peker på et objekt med alle "Studentegenskapene)
```

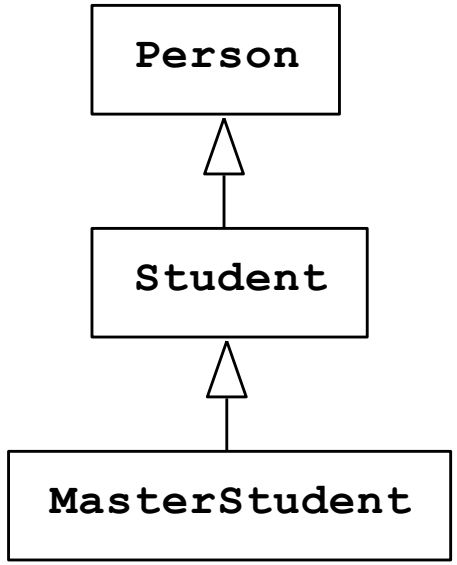
- For å unngå denne feilen, bør **instanceof** brukes:

```
if (pers instanceof Student) {
    Student stud = (Student) pers;
}
```



Konvertering mellom flere nivåer

```
MasterStudent master = new MasterStudent();
```



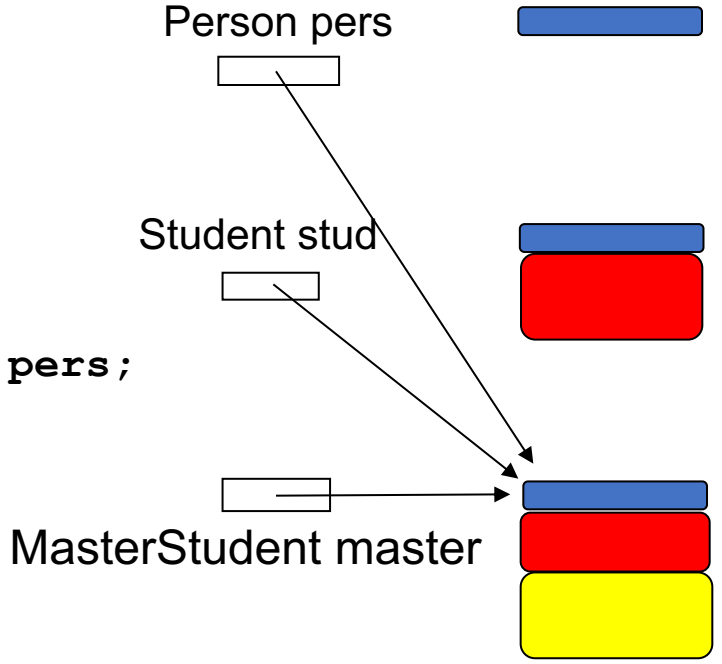
- **Konvertering oppover:**

```
Student stud = master;
Person pers = master;
```

- **Konvertering nedover:**

```
stud = (Student) pers;
master = (MasterStudent) pers;
```

(Dette krever kontroll under kjøring)



Regel: ” Alle referanser har lov til å peke bortover og nedover”

(men ikke ”oppover”)

Endelig: private og public i subklasser

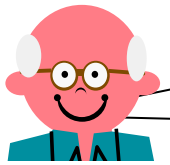
private gjør at ingen subklasser kan se denne egenskapen

protected gjør at alle subklasser kan se denne egenskapen
Men ingen utenfor klassen (bortsett fra i samme katalog/pakke)

public er som før

```
class Person {
    protected String navn;
    protected int tlfnr;


    public boolean gyldigTlfnr() {
        return tlfnr >= 10000000 && tlfnr <= 99999999;
    }
}
```



Nytt reservert ord / nøkkelord i Java:
protected

Student og Ansatt med **protected**

```
class Student extends Person {
    protected String program;

    public void byttProgram(String nytt) {
       program = nytt;
    }
}
```

```
class Ansatt extends Person {
    protected int lønnstrinn;
    protected int antallTimer;

    public void lønnstillegg(int tillegg) {
       lønnstrinn += tillegg;
    }
}
```

Om det hadde stått “private String program”, så ville ingen subklasser til Student kunne se denne egenskapen

Klassen Object

- class Object {. . . } er alle klassers mor (alle klassers superklasse)
- D.v.s. at alle klasser i Java er subklasser av klassen Object. Når vi skriver

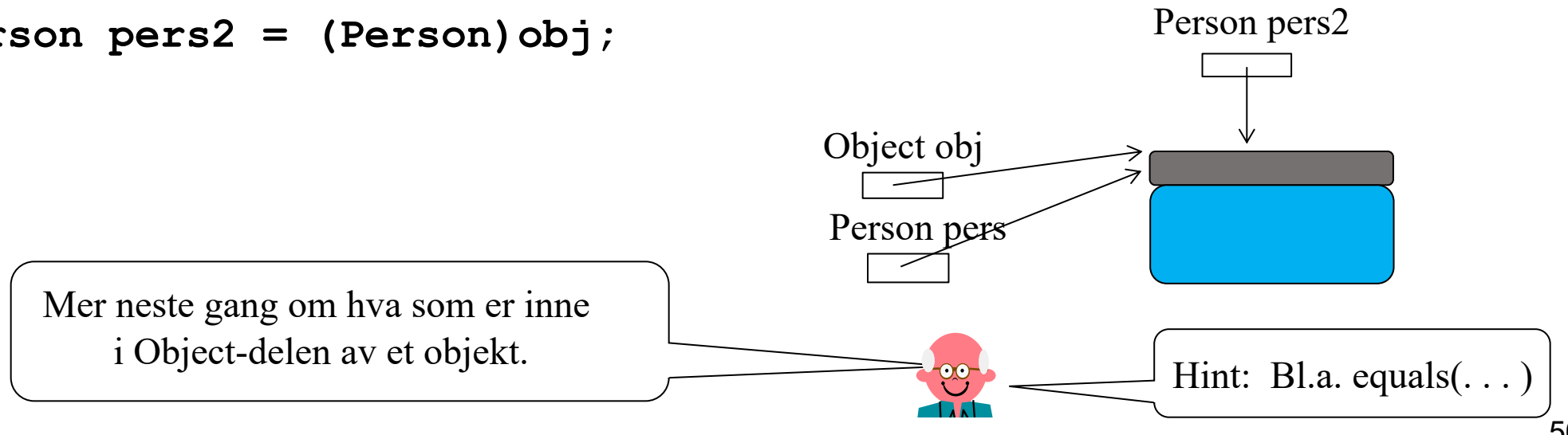
```
class Person { … }
```

  så tolker Java dette som

```
class Person extends Object { … }
```

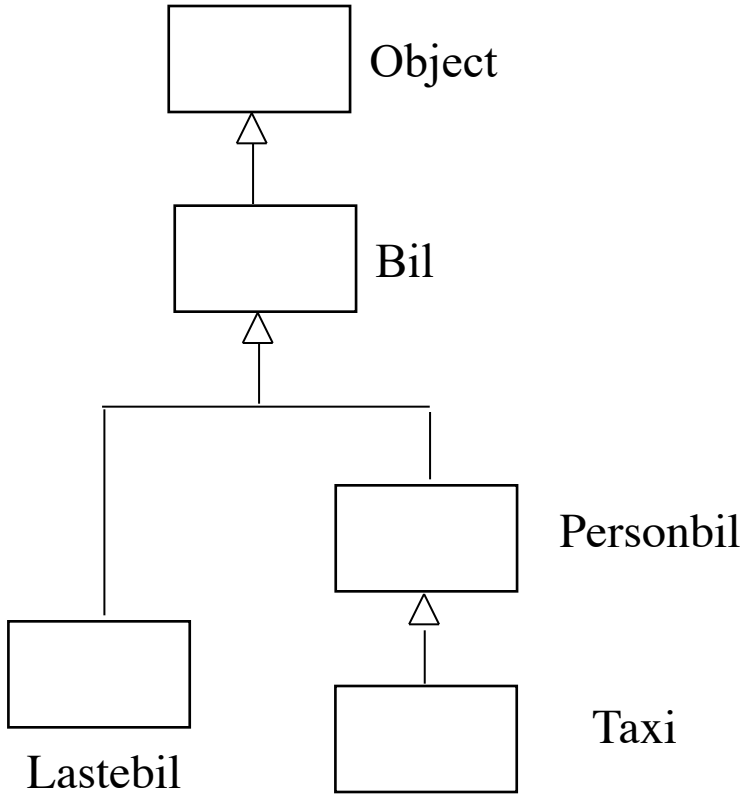
- Dermed kan en referanse av typen Object peke på et hvilket som helst objekt:

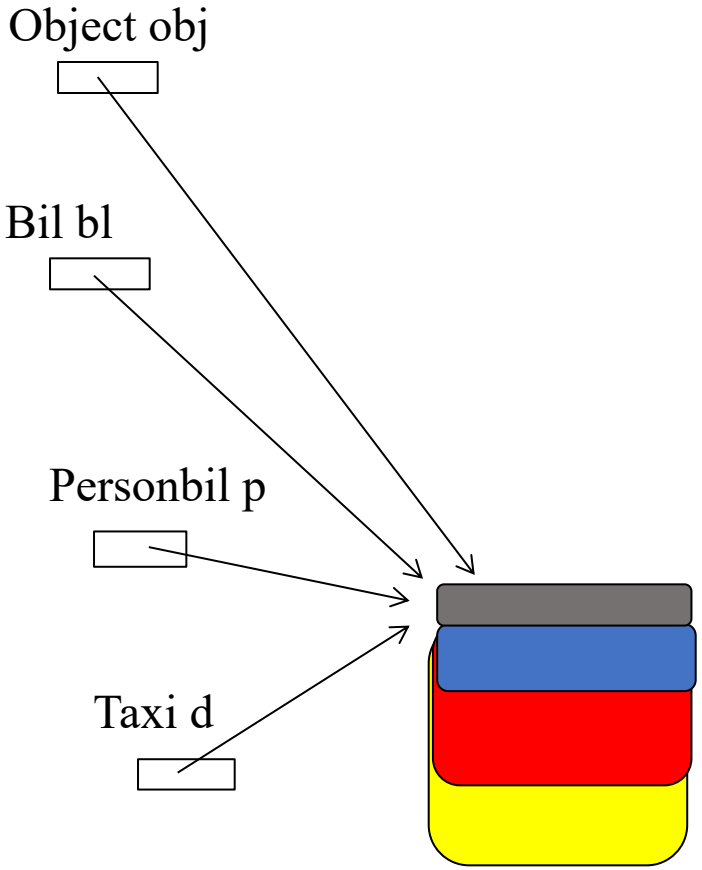
```
Person pers = new Person();
Object obj = pers;
Person pers2 = (Person)obj;
```



class Object - eksempel

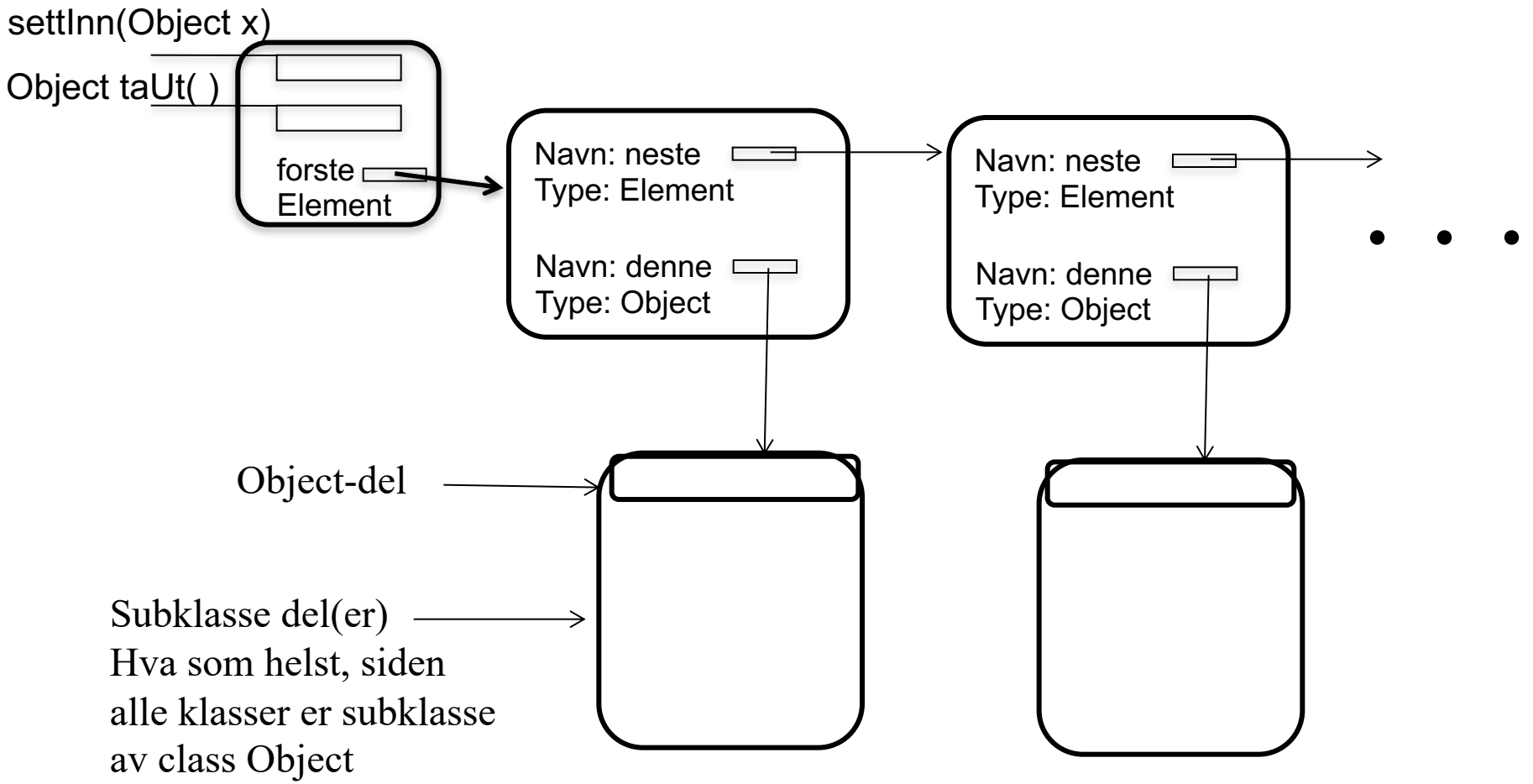
Klassehierarki:





En lenket liste med noder

Dette venter vi noen uker med



Konstanter i Java

- En konstant i Java deklareres med «final» og kan ikke endres etter at den er initialisert:

```
class KonstantDemoKlasse {
    protected final int objektId;
    KonstantDemoKlasse (int objektId) {
            this.objektId = objektId;
    }
}
```

Helt på siden av dette temaet (men har med scop å gjøre):

«this» er en referanse til objektet som koden er inne i.

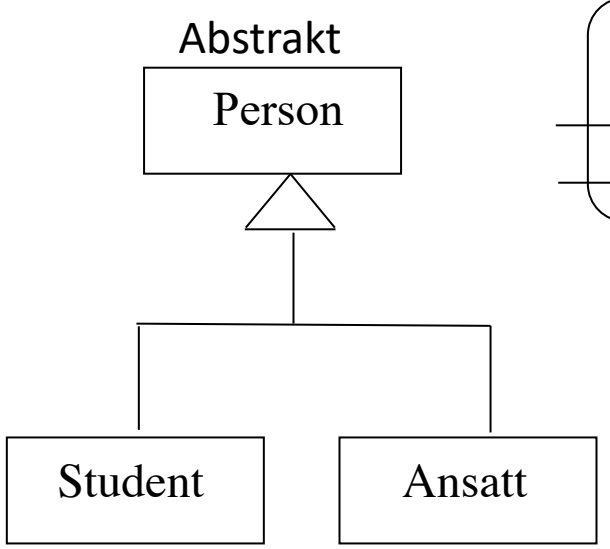
Bare når parameteren og instansvariabelen har samme navn brukes «this» i konstruktører!

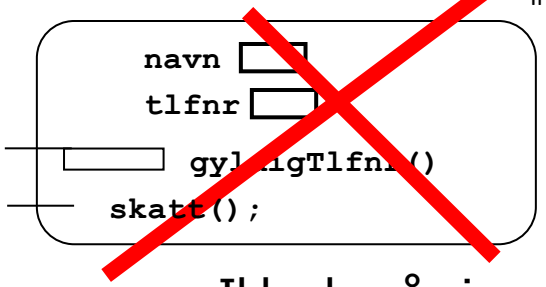
```
class KonstantDemoKlasse {
    protected final int objektId;
    KonstantDemoKlasse (int obId) {
            objektId = obId;
    }
}
```

Uten "this" fordi parameter og instansvariabel har forskjellig navn

Abstrakte metoder og abstrakte klasser


```
abstract class Person {
    protected String navn;
    protected int tlfnr;
    public abstract boolean skatt();
    public boolean gyldigTlfnr() { . . . }
}
```



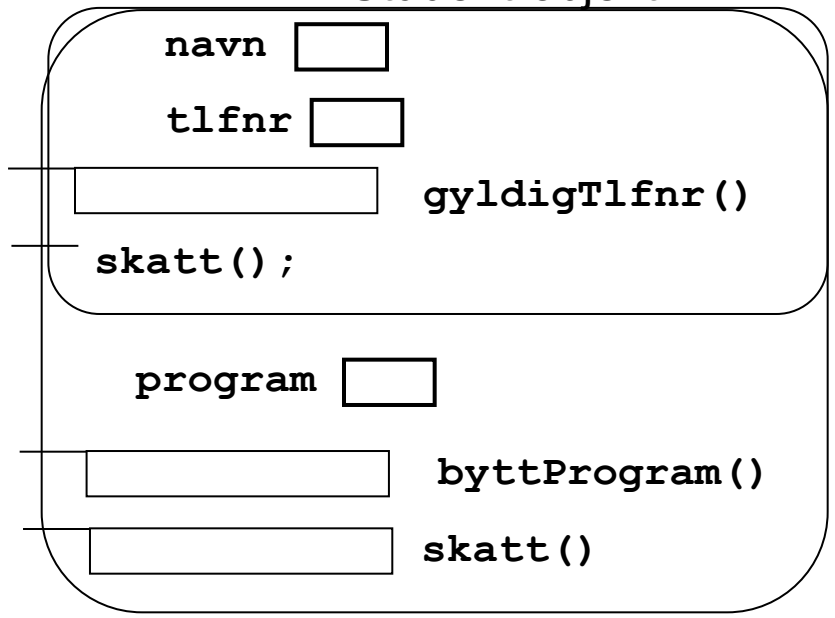


Ikke lov å si
new Person() !

```
class Student extends Person {
    protected String program;
    public  boolean skatt() {return 0;}
    void byttProgram(String nytt) { . . .  }
}
```

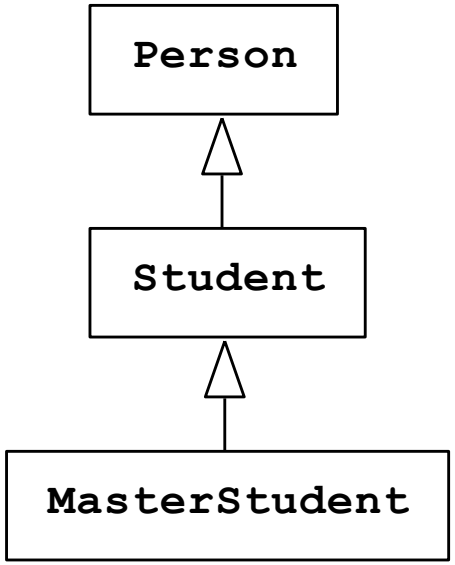
```
class Ansatt extends Person {
    protected int lønnstrinn;
    protected int antallTimer;
    public  boolean skatt() {return 100000;}
    publicvoid lønnstillegg(int tillegg) { ...}
}
```

Eksempel på et
Student-objekt



Mer forklaring de to neste ukene

Programmering med subklasser
Hoved-"take away " i dag

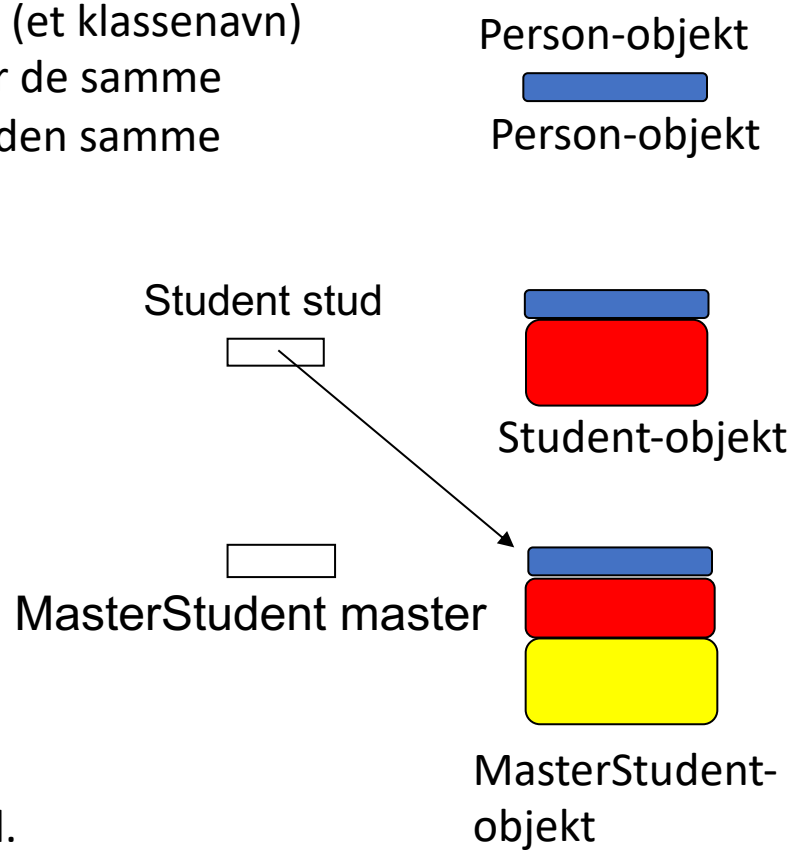


En peker (en referanseverdi) har en type (et klassenavn) og den vil bare peke på et objekt som har de samme (eller flere) egenskapene (objektet er av den samme klassen eller av en subklasse)

Tror du at objektet har flere egenskaper enn typen til pekeren tilsier, kan du teste dette med **instanceof**,
og du kan konvertere verdien til en subklassetype ved "casting", f.eks.:

master = (MasterStudent) stud;

Var dette ikke riktig får du en kjøretidsfeil.
Fang «ClassCastException»s.



I dag har vi lært

- Subklasser - extends
- Generalisering - spesialisering
- Subklasser - submengder
- Referanseverdier og referansevariabler av subklassetyper / subklassenavn
- Tilordninger mellom referanser - opp og ned i klassehierarkiet
- Test på objektets egenskaper: instanceof
- class Object
- Abstrakte classer og metoder – abstract

UNIVERSITETET I OSLO

Institutt for informatikk

IN1010 våren 2022

Tirsdag 8. februar

Arv og subklasser - del 2:

Polymorfi

Konstruktører i subklasser

Stein Gjessing

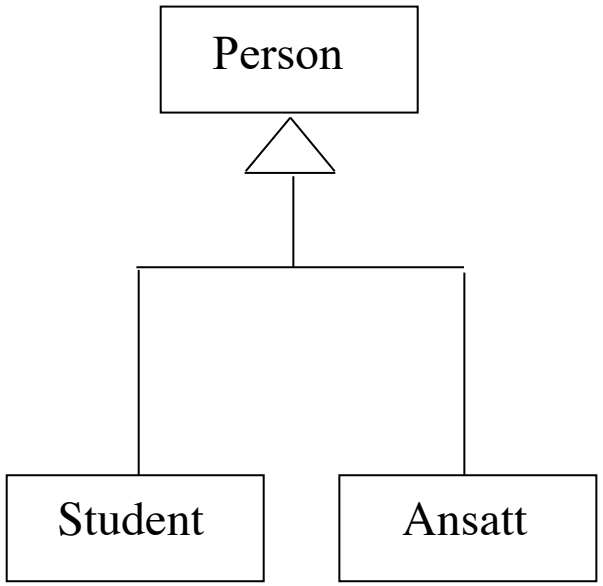
Dagens tema

- **Polymorfi**
- Når bruker vi arv / når bruker vi komposisjon
- **Konstruktører i subklasser**



Klassehierarki vs. programkode vs. datastrukturer

Når du planlegger programmet



Når du planlegger og kjører programmet

```
class Person {
    String navn;
    int tlfnr;
    boolean gyldigTlfnr() { . . . }
}
```

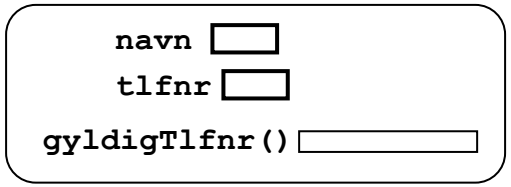
```
class Student extends Person {
    String program;

    void byttProgram(String nytt) {
        program = nytt;
    }
}
```

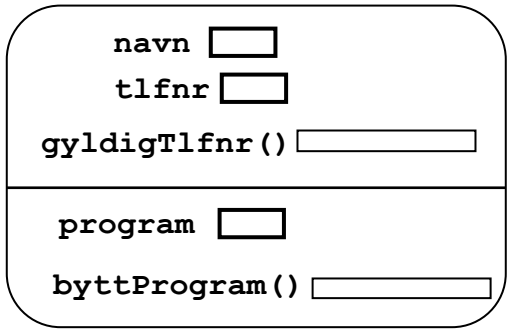
```
class Ansatt extends Person {
    int lønnstrinn;
    int antallTimer;

    void lønnstillegg(int tillegg) {
        lønnstrinn += tillegg;
    }
}
```

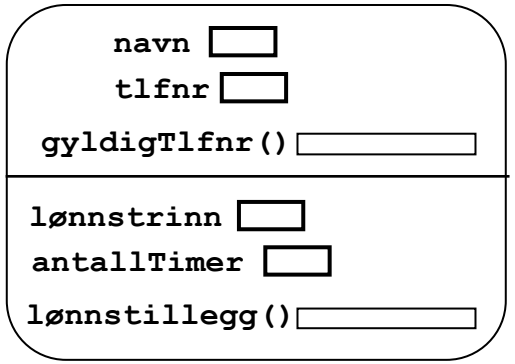
Når du (planlegger og) kjører programmet



Person-objekt



Student-objekt



Ansatt-objekt

Superklasser og subklasser

Superklasse →

```
class Person {
    String navn;
    int tlfnr;
    boolean gyldigTlfnr() { . . . }
}
```

```
class Student extends Person {
    String program;

    void byttProgram(String nytt) {
        program = nytt;
    }
}
```

← Subklasse og Superklasse

Subklasse →

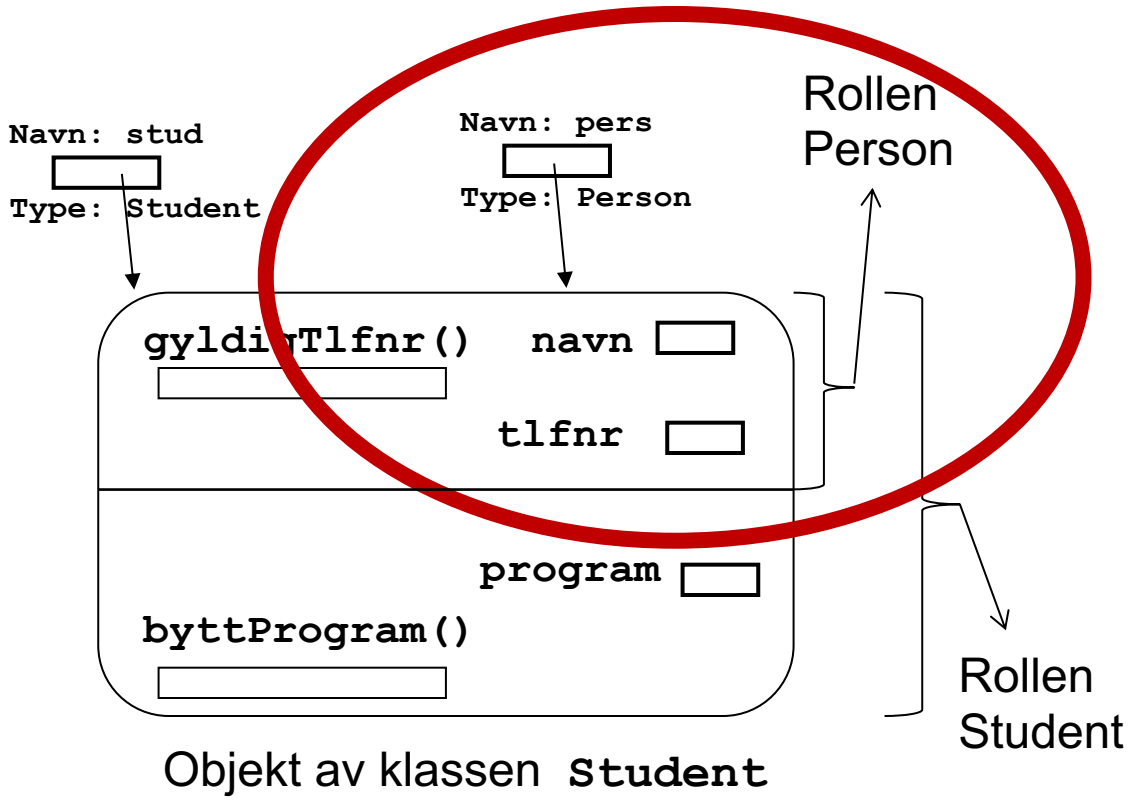
```
class Ansatt extends Person {
    int lønnstrinn;
    int antallTimer;

    void lønnstillegg(int tillegg) {
        lønnstrinn += tillegg;
    }
}
```

Java er et sterkt typet programmeringsspråk:
I enden av en **Person pers -** peker vil vi alltid se et objekt med **Person**-egenskaper



```
Student stud;
Person pers;
stud = new Student();
pers = stud;
```



Objekt av klassen **Student**

```
class Person {
    String navn;
    int tlfnr;
    boolean gyldigTlfnr() { . . . }
}
```

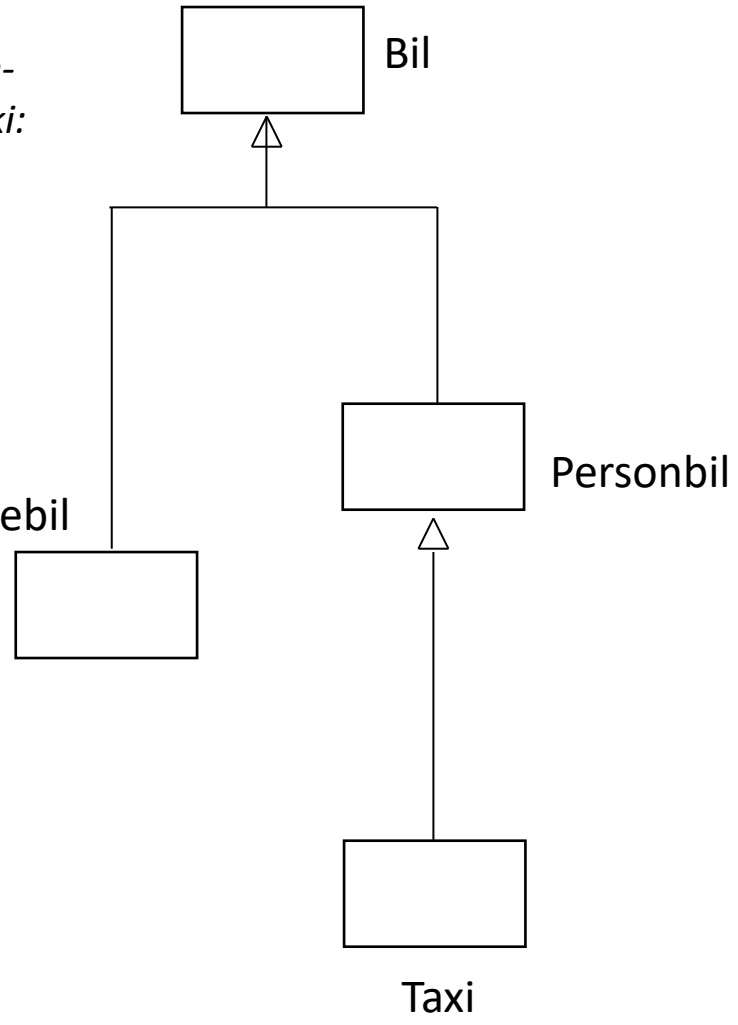
```
class Student extends Person {
    String program;
    void byttProgram(String nytt) { . . .  }
}
```

Polymorfi - motivasjon

NYTT

Alle biler skal betale skatt – men ikke like mye

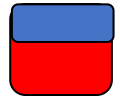
Klasse-hierarki:





Bil-objekt

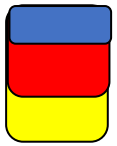
Eksempler på noen objekter



Personbil-objekt



Lastebil-objekt

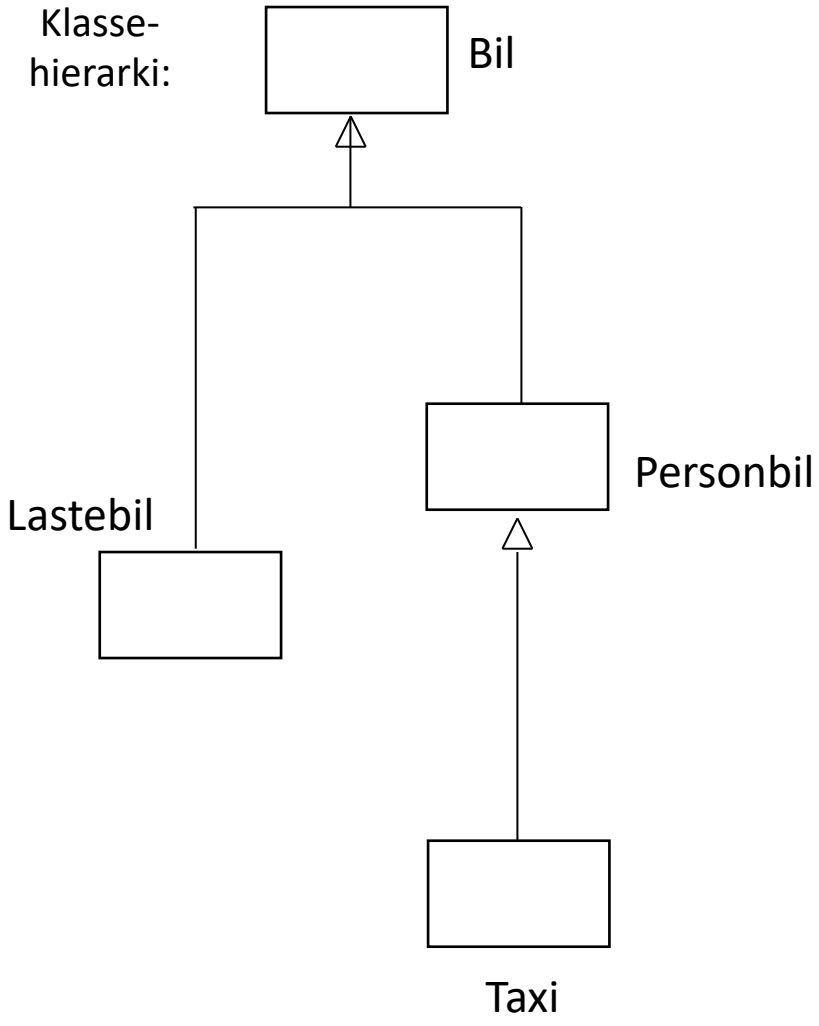


Taxi-objekt

Polymorfi

NYTT

Alle biler betaler skatt – men ikke like mye




```
class Bil {
    protected double pris;
    protected String regNr;
    public double skatt(){return 0;}
}

class Personbil extends Bil {
    protected int antPass;
    public double skatt( ){return pris;}
    . . .
}

class Lastebil extends Bil {
    protected double lasteVekt;
    public  double skatt(){return pris * 0.25;}
    . . .
}

class Taxi extends Personbil {
    protected int loyveNr;
    public double skatt(){return pris * 0.1; }
    . . .
}
```

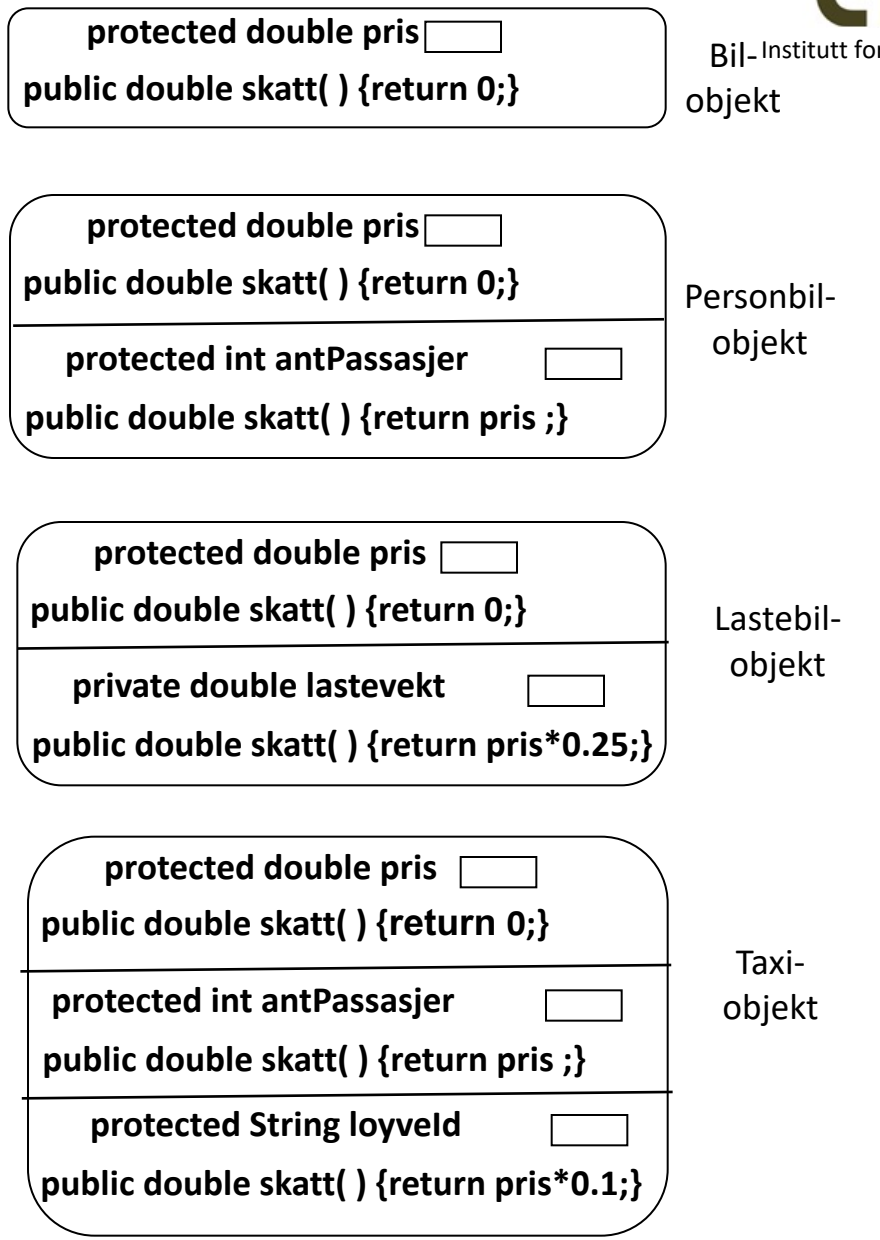
Polymorfi


```
class Bil {
    protected double pris;
    protected String regNr;
    public double skatt(){return 0;}
}

class Personbil extends Bil {
    protected int antPass;
    public double skatt( ){return pris;}
    . . .
}

class Lastebil extends Bil {
    protected double lasteVekt;
    public  double skatt(){return pris * 0.25;}
    . . .
}

class Taxi extends Personbil {
    protected int loyveNr;
    public double skatt(){return pris * 0.1; }
    . . .
}
```

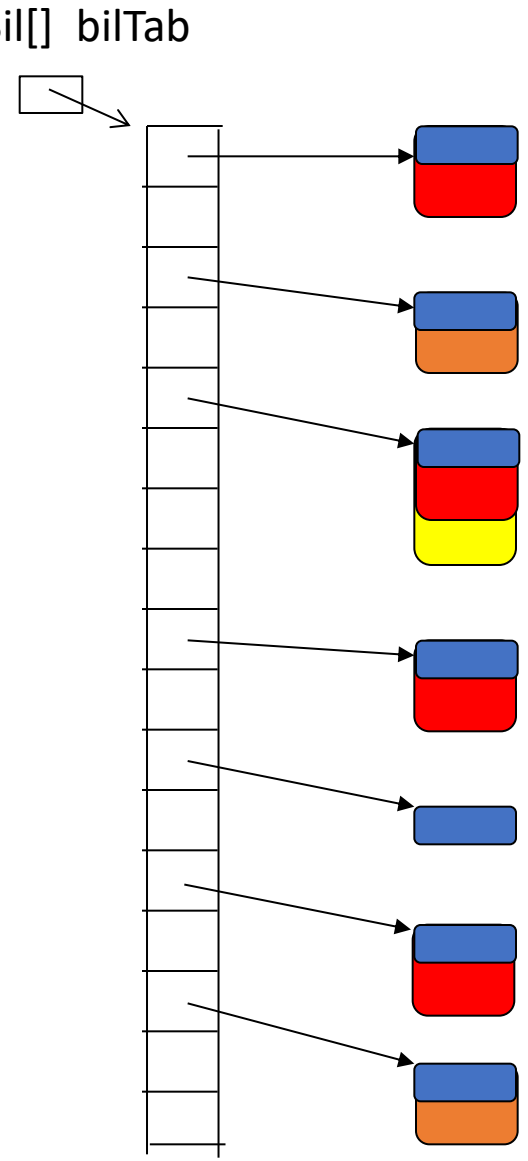


Polymorfi - motivasjon

Vi har en «database» med mange biler

Vi ønsker å skrive ut hva hver enkelt bil i databasen vår skal betale i skatt

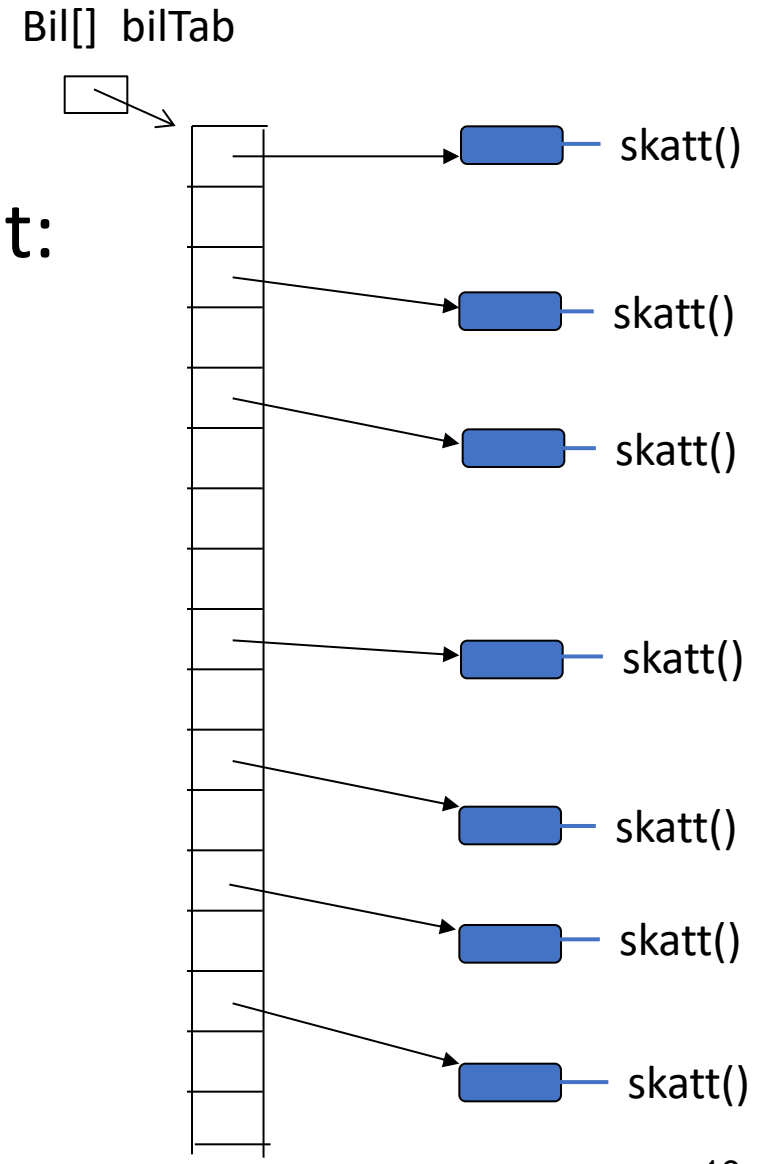
```
Bil[] bilTab = new Bil[1000];
 . . .
// Lag objekter og fyll inn i arrayen
 . . .
for (Bil b: bilTab) {
    System.out.println("Skatt: " + b.skatt();
}
```



Kompilatoren ser dette:

og tror at datastrukturen ser slik ut:

```
Bil[] bilTab = new Bil[1000];
 . . .
// Lag objekter og fyll inn i arrayen
 . . .
for (Bil b: bilTab) {
    System.out.println("Skatt: " + b.skatt();
}
```

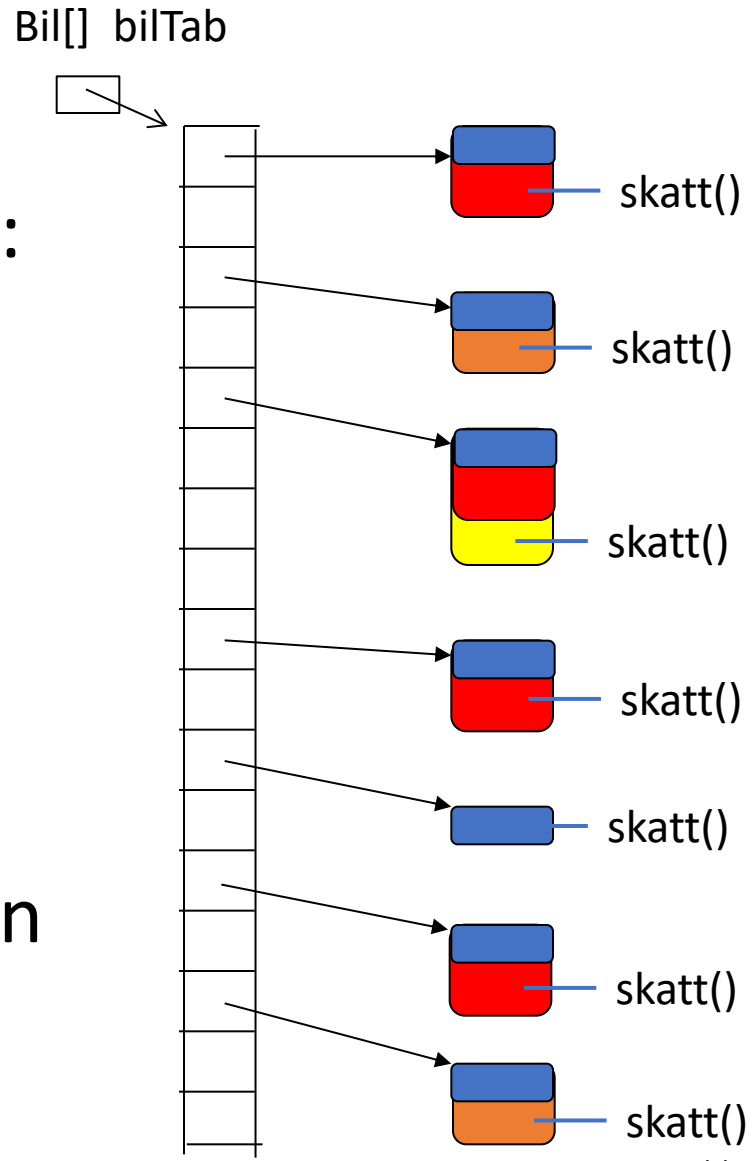


Polymorfi gjør

At kjøretidsystemt gjør dette:

```
Bil[] bilTab = new Bil[1000];
 . . .
// Lag objekter og fyll inn i arrayen
 . . .
for (Bil b: bilTab) {
    System.out.println("Skatt: " + b.skatt();
}
```

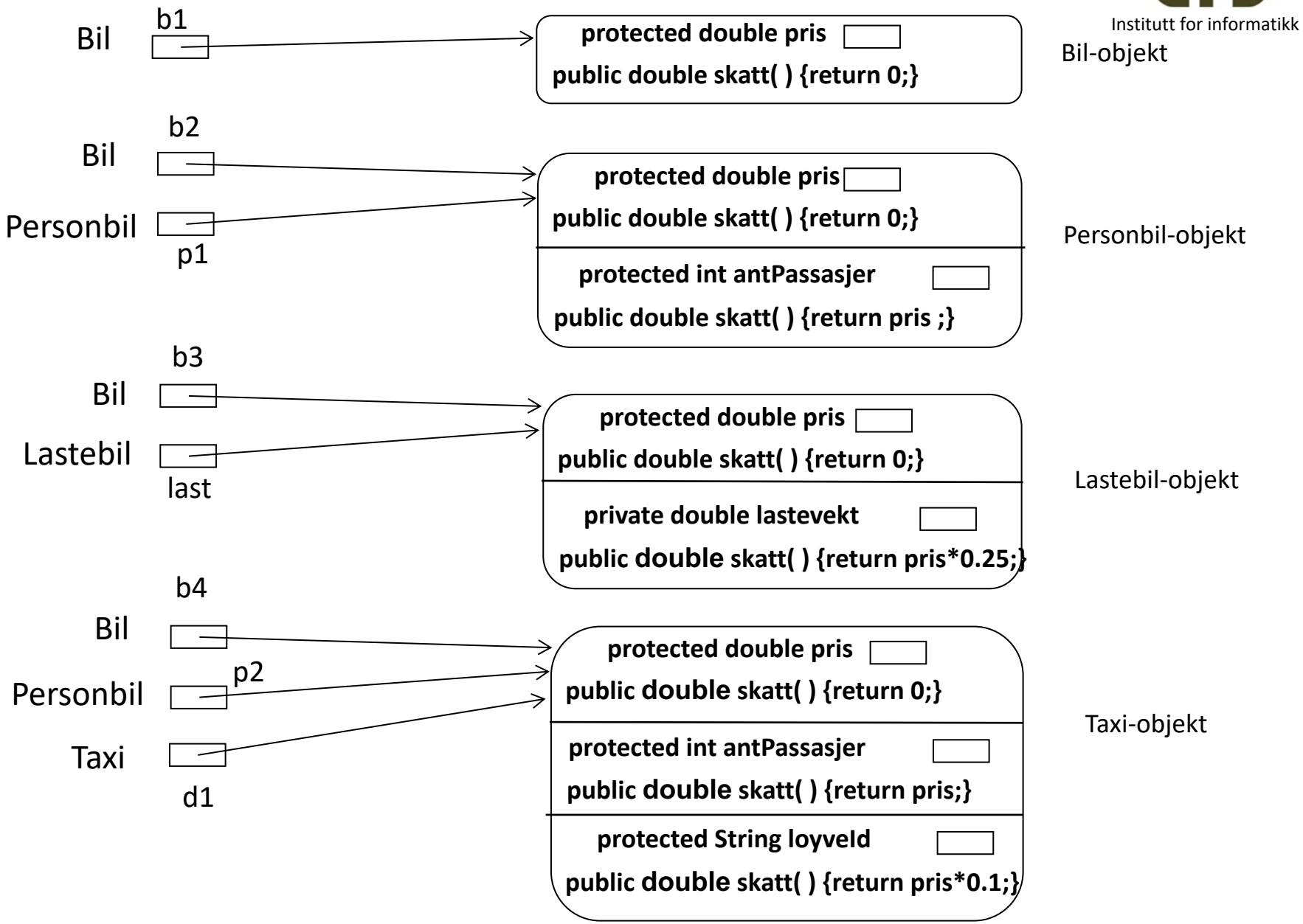
dvs. kjøretidsystemt bruker metoden i den nederste subklassen - det objektet «virkelig» er:



Polymorfi

Derfor:
Alle betaler uansett den skatten som er best definert av **objektets** type

uavhengig av referansetypen som brukes for å komme til objektet

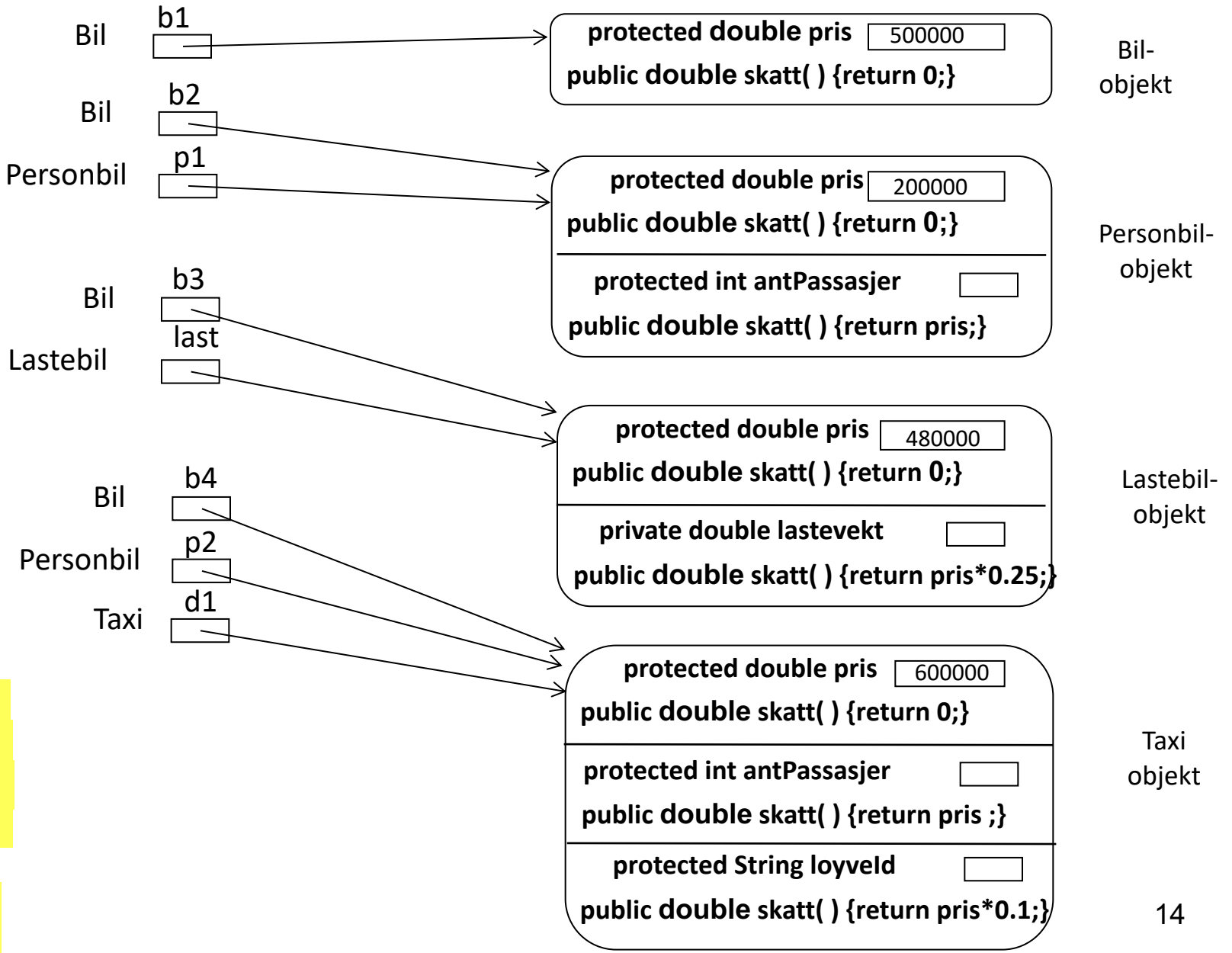


Redefinering av metoder - polymorfi

- Vi har tidligere sett at med subklasser kan vi utvide en eksisterende klasse med nye metoder (og nye variabler)
- En subklasse kan også deklarere en metode med ***samme signatur*** som en metode i superklassen, men med ulikt innhold i subklassen.
- Den nye metoden i subklassen vil redefinere (omdefinere, erstatte, overskrive, overstyre) (engelsk: override) metoden som er definert i superklassen
- Metoder som kan redefineres på denne måten kalles ***polymorfe metoder***
  (også kalt *virtuelle* metoder)
- I Java er alle metoder polymorfe, så sant de ikke er deklarert med **final**

- **Kompilatoren** bestemmer **at** det er lov, **kjøretidsystemet hvilken** metode som utføres, og det er den metoden som er lengst nede i objektets klassehierarki som utføres.

Polymorfi: eksempel



Hva blir:

b1.skatt()	0
b2.skatt()	200000
p1.skatt()	200000
b3.skatt()	120000
last.skatt()	120000
b4.skatt()	60000
p2.skatt()	60000
d1.skatt()	60000

```
class Bil {
    protected double pris;
    protected String regNr;
    public double skatt(){return 0;}
}

class Personbil extends Bil {
    protected int antPass;
    @Override
    public double skatt( ){return pris;}
    . . .
}

class Lastebil extends Bil {
    protected double lasteVekt;
    @Override
    public  double skatt(){return pris * 0.25;}
     . . .
}

class Taxi extends Personbil {
    protected int loyveNr;
    @Override
    public double skatt(){return pris *0.1; }
     . . .
}
```

Bruk annotasjonen @Override

Bruk annotasjonen:

@Override

```java
class Bil {
    protected double pris;
    protected String regNr;
    public double skatt(){return 0;}
}

class Personbil extends Bil {
    protected int antPass;
    @Override
    public double skatt( ){return pris;}
    . . .
}

class Lastebil extends Bil {
    protected double lasteVekt;
    @Override
    public  double skatt(int i){return pris * 0.25;}
     . . .
}

class Taxi extends Personbil {
    protected int loyveNr;
    @Override
    public double skatt(){return pris *0.1; }
     . . .
}
```

Kompilatoren gir feilmelding:

```
MacBook-Pro:programmer steing$ javac BrukHashMap.java
BrukHashMap.java:32: error: method does not override or implement a method from a supertype
        @Override
        ^
1 error
MacBook-Pro:programmer steing$
```

Polymorfi: Nytt eksempel

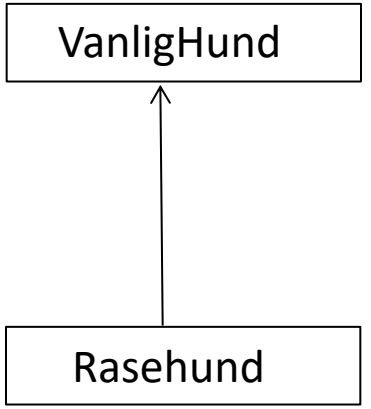

```java
class VanligHund {
   // ...
   public void bjeff() {
      System.out.println("Vov-vov");
   }
}



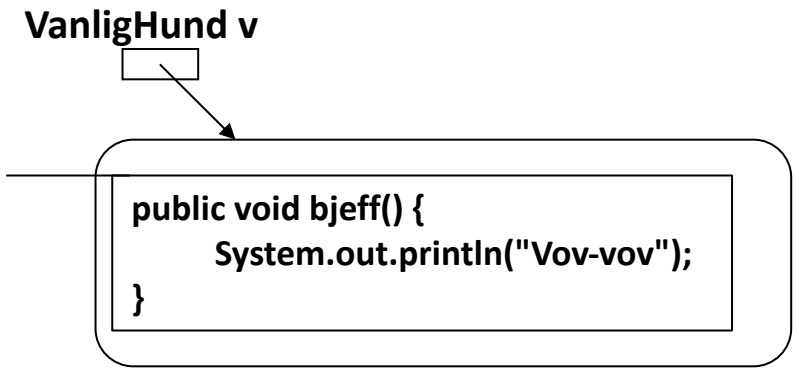
class Rasehund extends VanligHund {
   // ...
   @Override
   public void bjeff() {
      System.out.println("Voff-voff");
   }
}
```

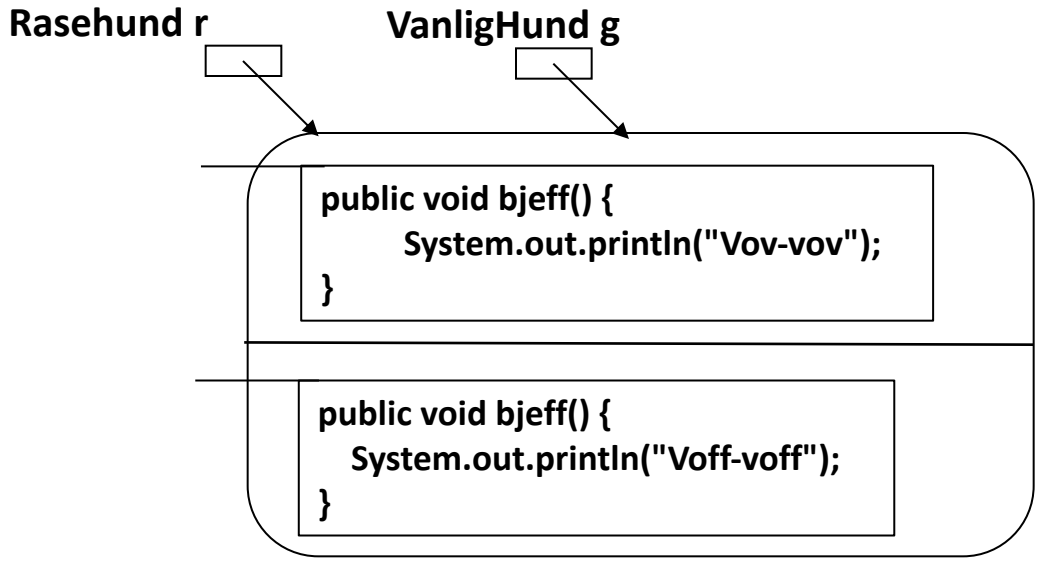
For objekter av typen VanligHund er det denne metoden som gjelder.

For objekter av typen Rasehund er det denne metoden som gjelder.



Polymorfi: eksempel





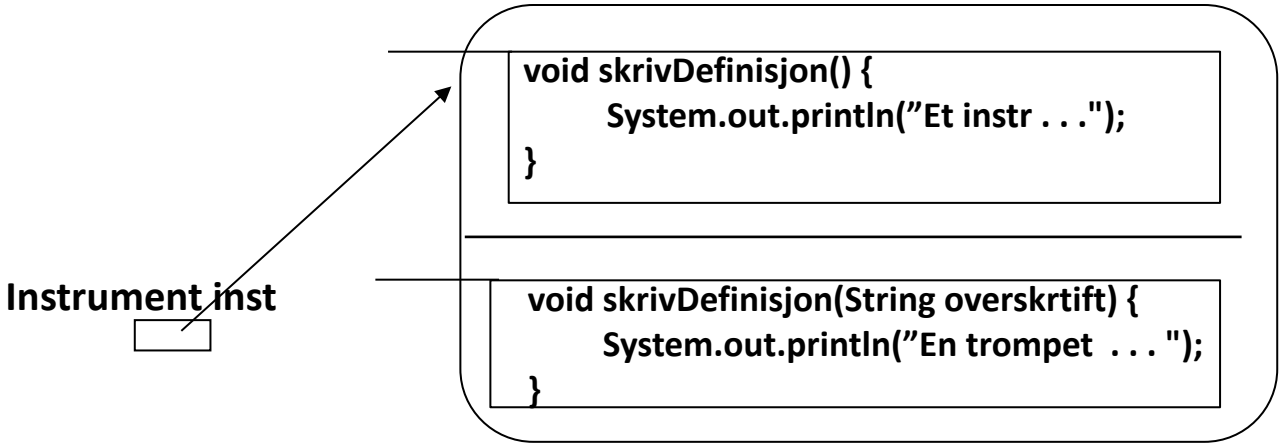
Anta dette programmet:

```
VanligHund v = new VanligHund();
Rasehund r = new Rasehund();
VanligHund g = r;
```

Hva skrives ut ved hvert av kallene:

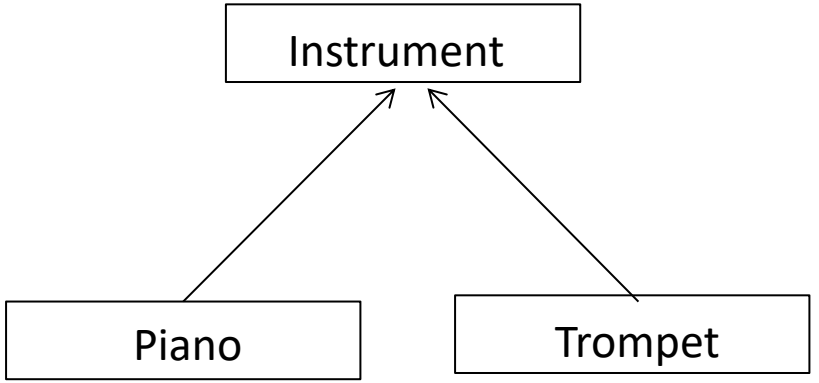
v.bjeff();	**Vov-vov**
r.bjeff();	**Voff-voff**
g.bjeff();	**Voff-voff**

Polymorfi – eksempel




```
class Musikk{
  public static void main (String[] args) {
  Instrument  inst = new Trompet();
     inst.skrivDefinisjon();
  }
}
class Instrument {
  public void skrivDefinisjon() {
    System.out.println("Et instrument er noe man kan spille på");
  }
}
class Piano extends Instrument {
  public void skrivDefinisjon () {
    System.out.println("Et piano er et strengeinstrument");
  }
}
class Trompet extends Instrument {
  public void skrivDefinisjon (String overskrift) {
    System.out.println(overskrift);
    System.out.println("En trompet er et blåseinstrument");
  }
}
```

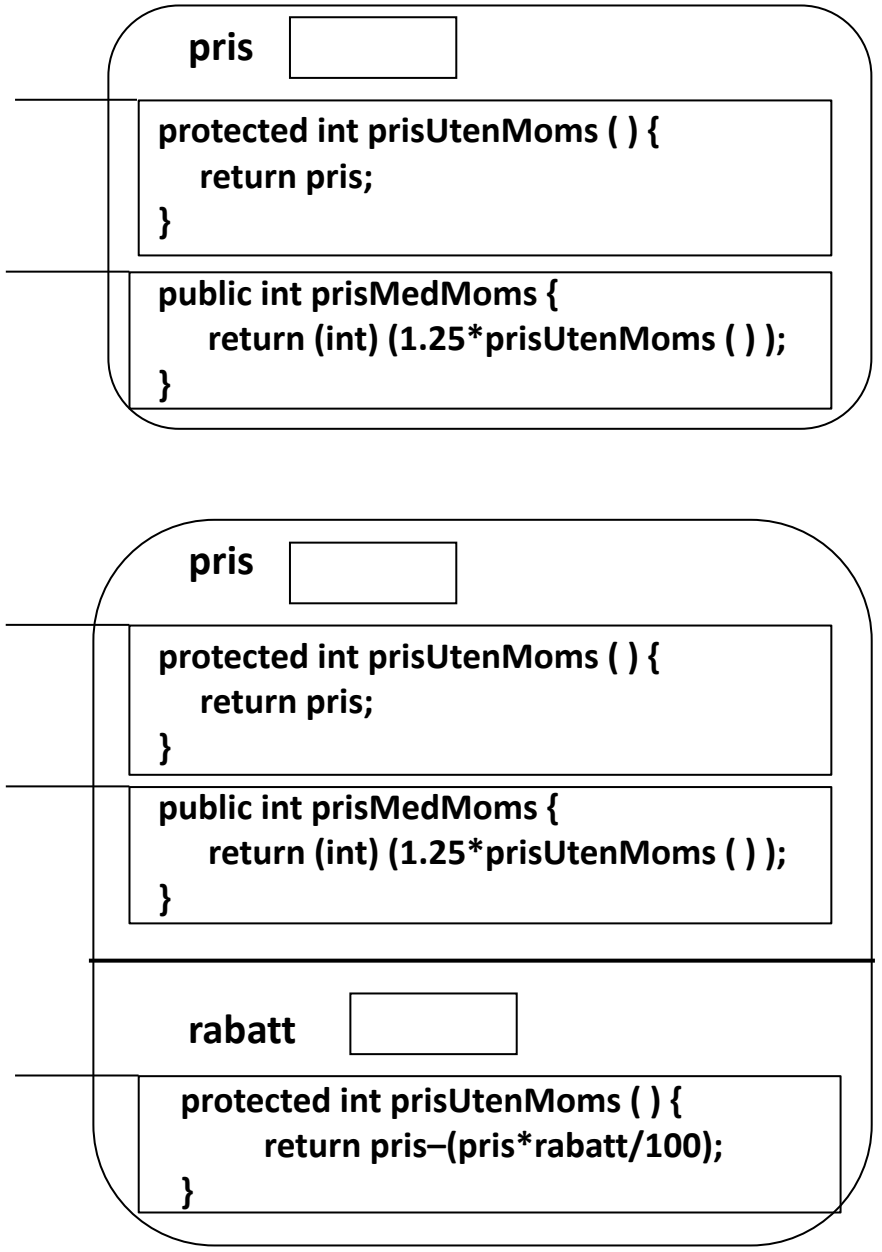
Objekt av klassen Trompet



Flere polymorfe metoder


```
class Vare {
    protected int pris;
    public void setPris(int p){pris = p;}
    protected int prisUtenMoms() {
        return pris;
    }
    public int prisMedMoms() {
        return (int) (1.25*prisUtenMoms());
    }
}

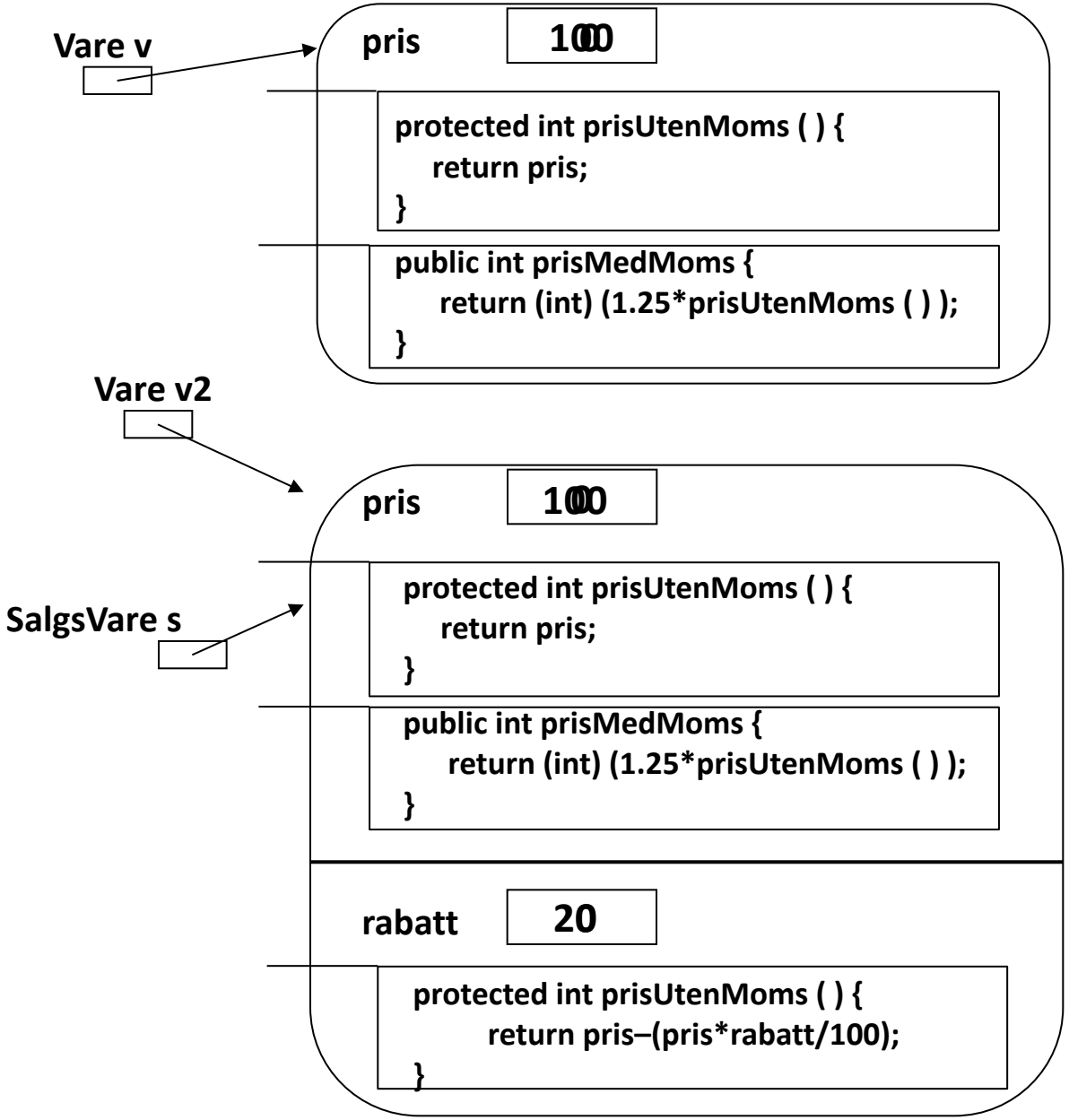
class SalgsVare extends Vare {
    protected int rabatt; // I prosent
    public void setRabatt(int r) {
        rabatt = r;
    }
    @Override
    protected int prisUtenMoms() {
        return (int) pris-(pris*rabatt/100);
    }
}
```



```
Vare v = new Vare();
v.setPris(100);
SalgsVare s =
     new SalgsVare();
s.setPris(100);
s.setRabatt(20);
Vare v2 = s;
```

Hva blir nå:

```
v.prisMedMoms();
s.prisMedMoms();
v2.prisMedMoms();
```



Polymorfi: skrivData

- I universitets-eksemplet så vi at klassene Student og Ansatt (før vi hadde lært om subklasser) hadde nesten like skrivData-metoder:

```
    // I klassen Student:
    @Override
    public void skrivData() {
        System.out.println("Navn: " + navn);
        System.out.println("Telefon: " + tlfnr);
        System.out.println("Studieprogram: " + program);
    }
```

```
    // I klassen Ansatt:
    @Override
    public void skrivData() {
        System.out.println("Navn: " + navn);
        System.out.println("Telefon: " + tlfnr);
        System.out.println("Lønnstrinn: " + lønnstrinn);
        System.out.println("Timer: " + antallTimer);
}
```

Nøkkelordet **super**

NYTT

Nøkkelordet **super** brukes til å aksessere variable / metoder i objektets superklasse. Dette kan vi bruke til å la superklassen Person ha en generell **skrivData**, som så kalles i subklassene:

```
// I klassen Person:
public void skrivData() {
    System.out.println("Navn: " + navn);
    System.out.println("Telefon: " + tlfnr);
}
```

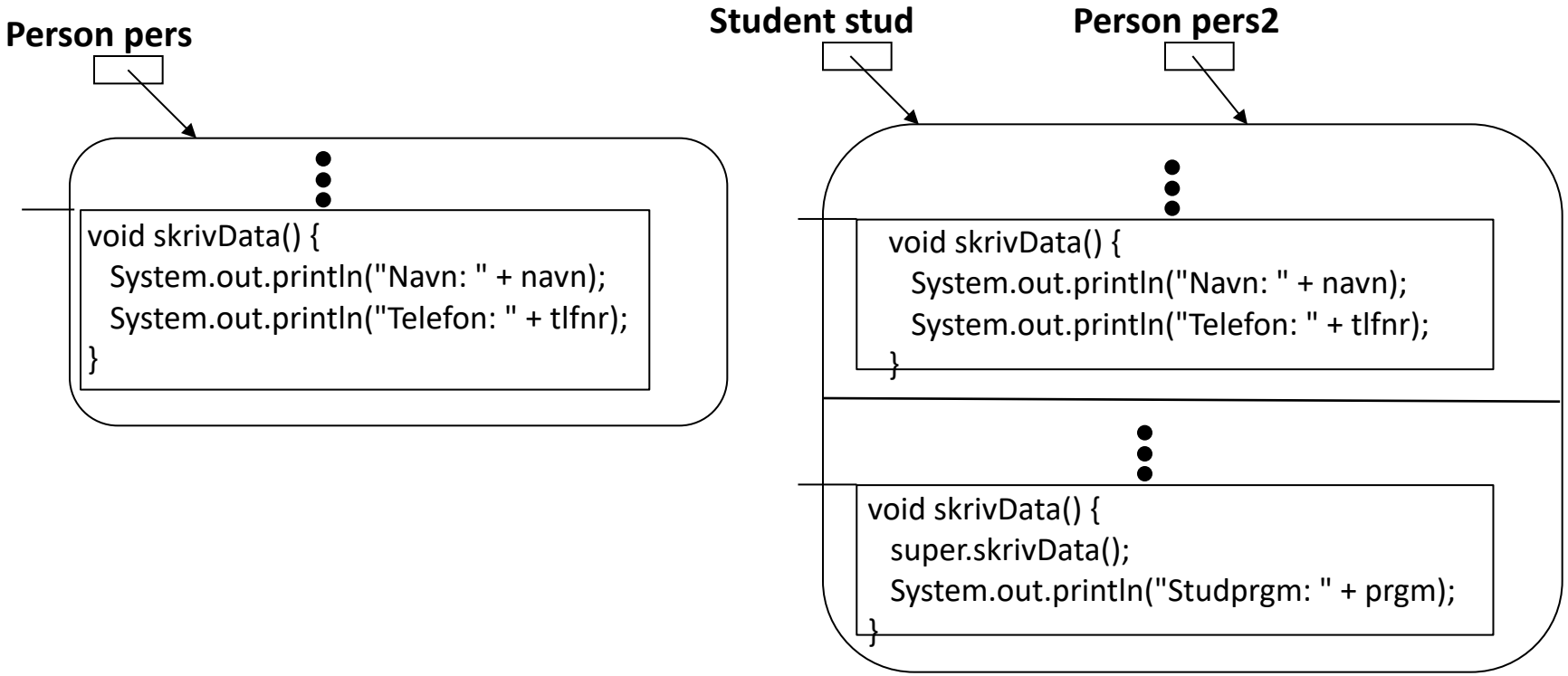
```
// I klassen Student:
@Override
public void skrivData() {
    super.skrivData();
    System.out.println("Studieprogram: " + program);
}
```

```
// Tilsvarende i klassen Ansatt:
@Override
public void skrivData() {
    super.skrivData();
    System.out.println("Lønnstrinn: " + lønnstrinn);
    System.out.println("Timer: " + antallTimer);
}
```

```
class StudentRegister {
  public static void main(String [] args) {
    Student stud = new Student();
    Person pers  = new Person();

    stud.skrivData();    // Her brukes definisjonen i Student
    pers.skrivData();    // Her brukes definisjonen i Person

    Person pers2 = stud;
    pers2.skrivData();  // Hvilken definisjon benyttes her?
   }
}
```



Overlaste vs. redefinere
Eng.: Overload vs. override

- Det er vanlig å ha flere definisjoner av en metode med samme navn, men med forskjellige parametre (**forskjellige signaturer**)
  f.eks. System.out.print(19); og System.out.print("Hei");
- Dette kalles overload på engelsk, norsk: overlasting
- Sett fra Java kunne metodene gjerne hatt helt forskjellige navn.
- Det er pga. at programmet skal leses og forstås (og metodene huskes) av andre mennesker at vi bruker det samme navnet

- Overlasting har ikke noe med polymorfi å gjøre !!
- Overlasting bestemmes helt og holdent av kompilatoren (og det er ikke noe å bestemme, det er helt vanlige metoder).

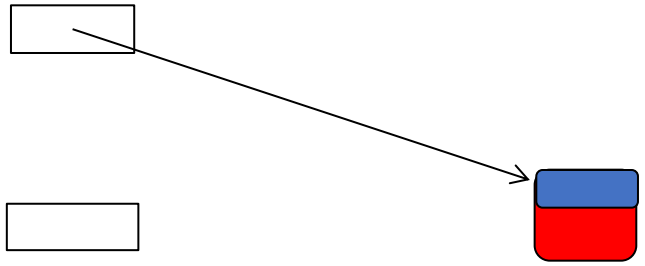
Omdefinering av **variable**

- En subklasse kan også redefinere (skyggelegge) variable som er definert i superklassen.
- MEN: Dette bør IKKE brukes!!!
  - Sjelden nødvendig
  - Reduserer lesbarheten
  - Kan føre til uventet oppførsel
- Og mer trenger dere ikke å vite om det…

Husk:

Den boolske operatoren **instanceof** hjelper oss å finne ut av hvilken klasse et gitt objekt er, noe som er nyttig i mange tilfeller:

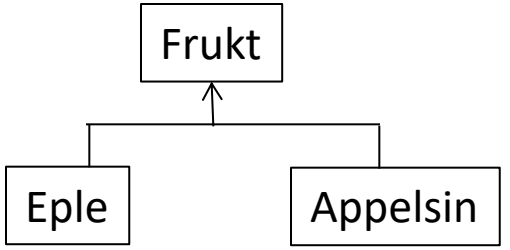
```
class TestFrukt {
  public static void main(String[] args) {
     Eple e = new Eple();
     skrivUt(e);
  }
  static void skrivUt(Frukt f)  {
     if (f instanceof Eple)
         System.out.println("Dette er et eple!”);
     else if (f instanceof Appelsin)
         System.out.println("Dette er en appelsin!”);
  }
}
```



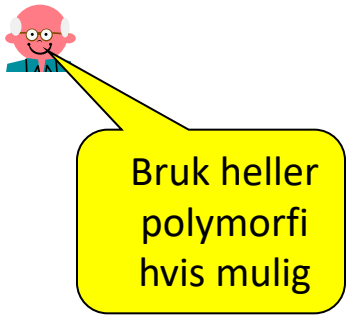
class Frukt { .. }

class Eple extends Frukt { .. }

class Appelsin extends Frukt { .. }



Men: Prøv å unngå ”instanceof”



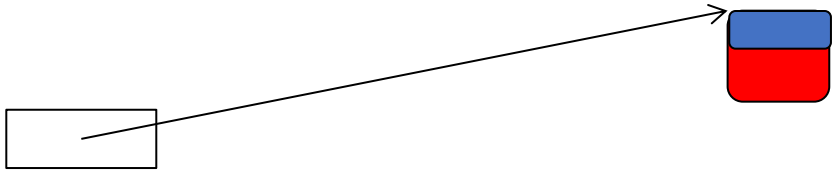
Bedre program:

```
class TestFrukt2 {
    public static void main(String[] args) {
        Frukt2 f = new Eple2();
        f.skrivUt( );
    }
}

abstract class Frukt2 {
        abstract public void skrivUt( );
}

class Eple2 extends Frukt2 {
    @Override
    public void skrivUt( ) {
        System.out.println("Jeg er et eple!");
    }
}

class Appelsin2 extends Frukt2 {
    @Override
    public void skrivUt( ) {
        System.out.println("Jeg er en appelsin!");
    }
}
```



Istedenfor å teste hvilken klasse objektet er av, ber vi objektet gjøre jobben selv.
Objektet vet jo best selv hva det er.

Biler og mer bruk av "instanceof"

god

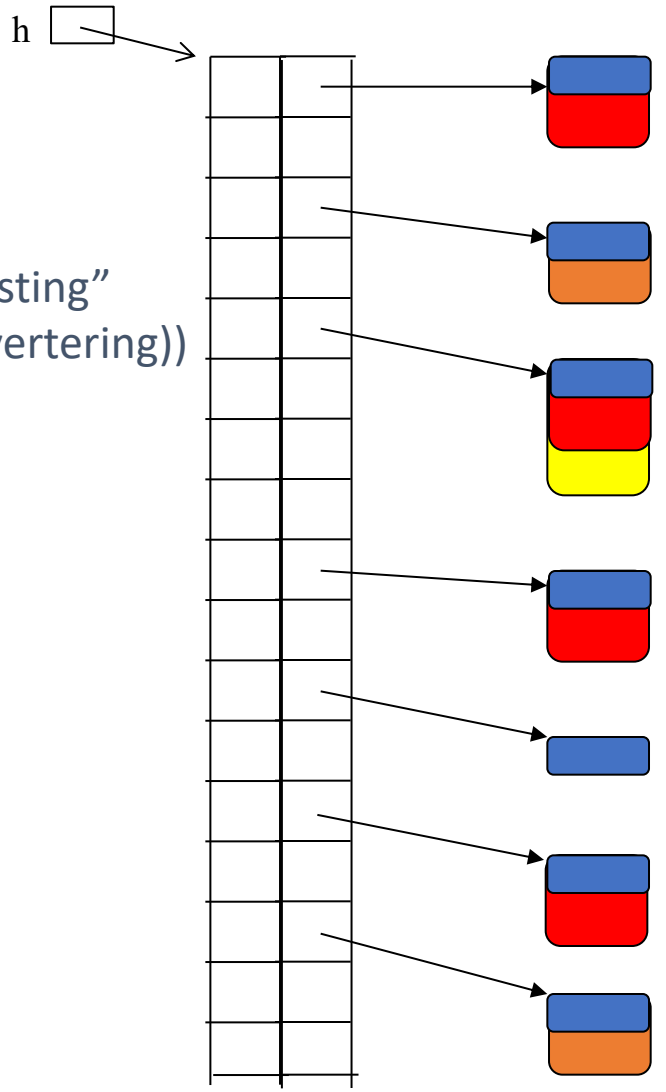

```
HashMap <String, Bil> h;
h = new HashMap <String, Bil> ( );
// Fyll inn forskjellige biler i hashmap-en
for (Bil b: h.values()) {
    // kall på polymorf metode:
    double  skatt = b.skatt();
    if (b instanceof Personbil) {
        Personbil pb = (Personbil) b;
        int pas = pb.passasjerAnt();
    } else if (b instanceof Lastebil) {
        Lastebil ls = (Lastebil) b;
        double lv = ls.hentLast();
    }
}
```

```
class Bil {
    protected double pris;
    public double skatt(){return 0;}
}

class Personbil extends Bil {
    protected int antPass;
    @Override
    public double skatt( ){return pris;}
    public int passasjerAnt( ){return antPass;}
}

class Lastebil extends Bil {
    protected double lasteVekt;
    @Override
    public  double skatt(){return pris * 0.25;}
    public int hentLast( ){return lasteVekt;}
}
```

(og litt "casting" (typekonvertering))



Hvorfor bruker vi subklasser?

Repetisjon

1. Klasser og subklasser avspeiler **virkeligheten**
   - Bra når vi skal modellere virkeligheten i et datasystem
2. Klasser og subklasser avspeiler **arkitekturen** til datasystemet / dataprogrammet
   - Bra når vi skal lage et oversiktlig stort program

3. Klasser og subklasser kan brukes til å forenkle og gjøre programmer mer forståelig, og spare arbeid:
   Gjenbruk av programdeler
   - ”Bottom up” – programmering
     - Lage verktøy
   - ”Top down” programmering
     - Postulere verktøy



Gjenbruk av deler av programmer

- Viktig å ikke måtte skrive ny kode hver gang man skal programmere noe nytt
  - Gjenbruk mest mulig av kode du har skrevet før
- Lag kode med henblikk på et den skal brukes (til noe liknende) senere
  - Lag biblioteker
- Bruk andres bibliotek
- Javas eget bibliotek

- Strukturering av kode (”gjenbruk” i samme program)
- IN1000: Gjenbruk av metoder og klasser

Gjenbruk ved hjelp av klasser / subklasser

- Ved ***sammensetning*** / ***komposisjon***
  - i Oblig 1 - pensum i IN100
  - Deklarer referanser til objekter av klasser du har skrevet før (eller bibliotheksklasser)
  - Lag objekter av disse klassen
  - Kall på metoder i disse klassene
- Ved ***arv*** (nytt i IN1010):
  - Lag en ny klasse som utvider den eksisterende klassen (spesielt viktig ved litt større klasser)
  - Føy til ekstra variabler og metoder

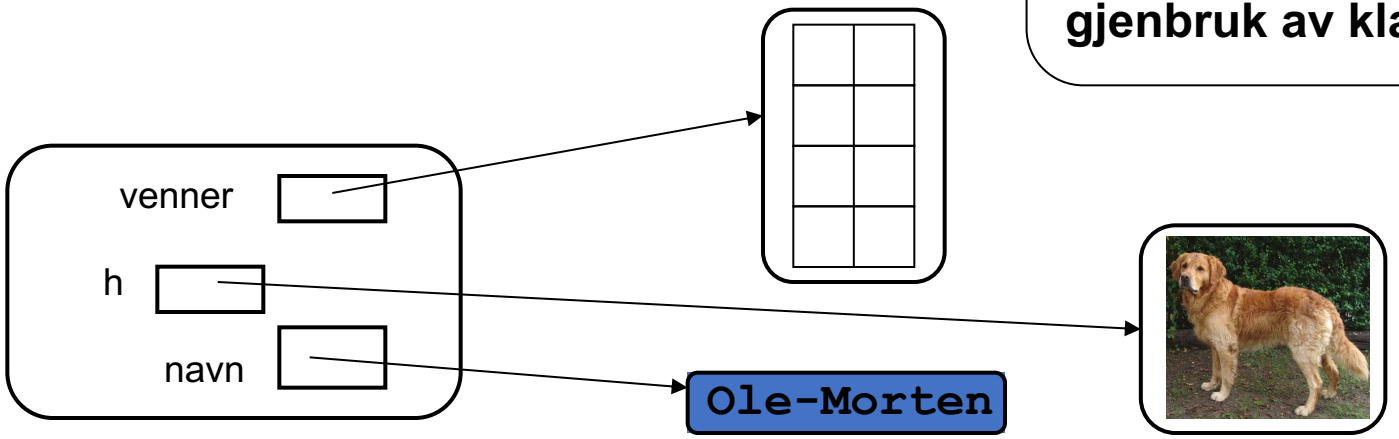
Gjenbruk ved sammensetning / komposisjon

Omtrent som i oblig 1. Ikke noe nytt

```
class Demoklasse {
    HashMap<String,Person> venner = new HashMap<String,Person>();
    Husdyr h = new Hund("Passopp");
    String navn = "Ole-Morten";


    /* + Diverse metoder */
}
```

'venner', 'h' og 'navn' er deklarert som referansevariable som peker på objekter av andre klasser **som allerede eksisterer / gjenbruk av klassedefinisjonene**



Gjenbruk ved arv

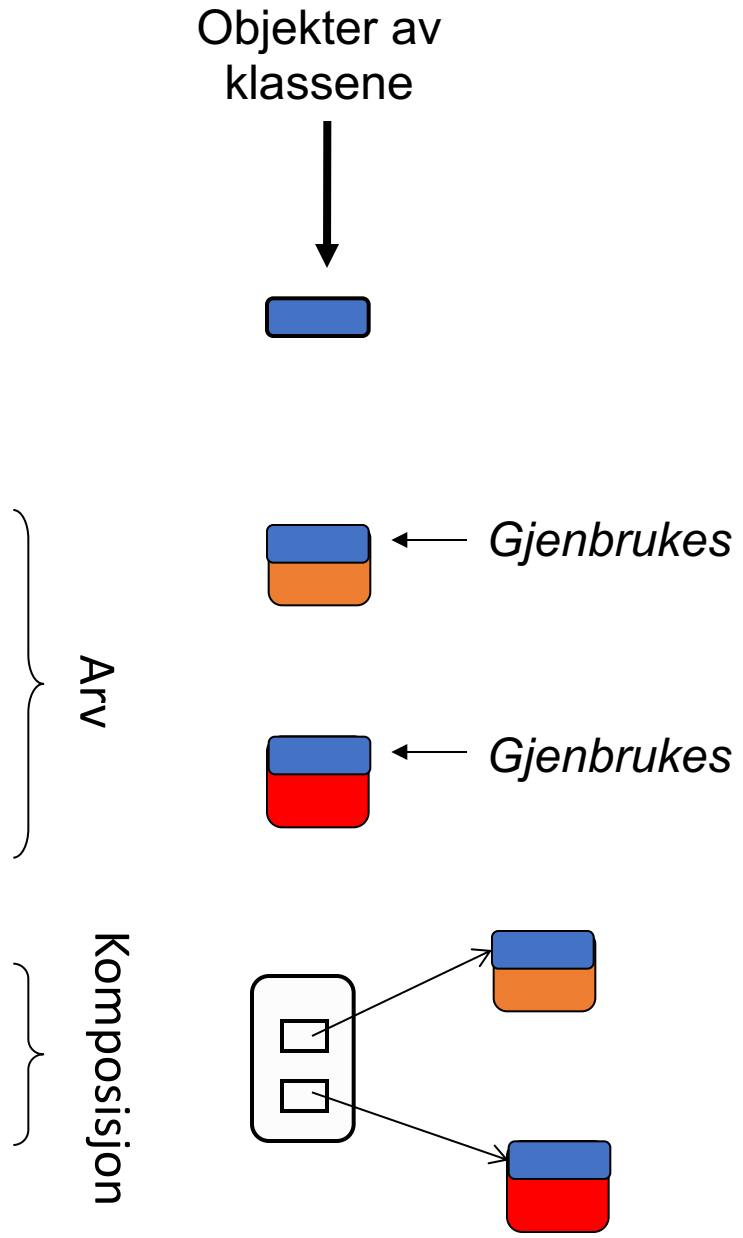
(domenet er et bibliotek)


```
class Bok {
   protected String tittel, forfatter;
}

class Fagbok extends Bok {
   protected double dewey;
}

class Skjønnlitterærbok extends Bok {
   protected String sjanger;
}

class Bibliotek {
   protected Bok b1 = new Fagbok();
   protected Bok b2 = new Skjønnlitterærbok();
}
```



Når skal vi bruke arv?

- Generelt: Ved *er-en* relasjon mellom objektene.
  - En Student er en Person
  - En Ansatt er en Person
- Hva med relasjonene
  - roman – bok?
    - En roman *er en* bok (arv).
  - kapittel – bok?
    - Et kapittel *er ikke* en bok, men et kapittel *er en del av* en bok, og en bok *har/består av* kapitler (sammensetning)
- Relasjoner som *har-en* og *består-av* skal ikke modelleres som subklasser, men ved hjelp av sammensetning / komposisjon (som datafelt (konstanter/variable)).

Arv vs. delegering kommer vi tilbake til



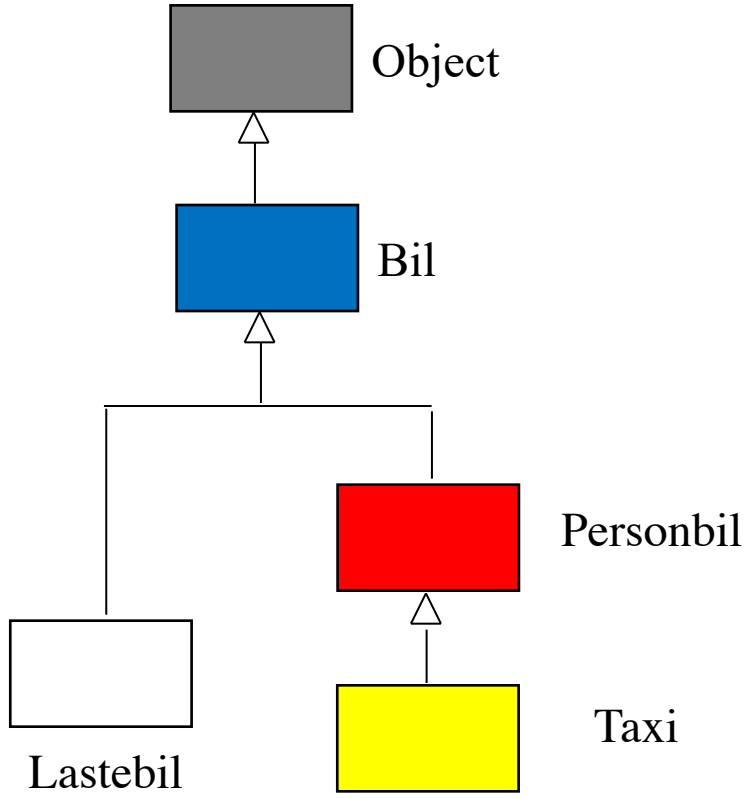
Menti kode: 5575 3366

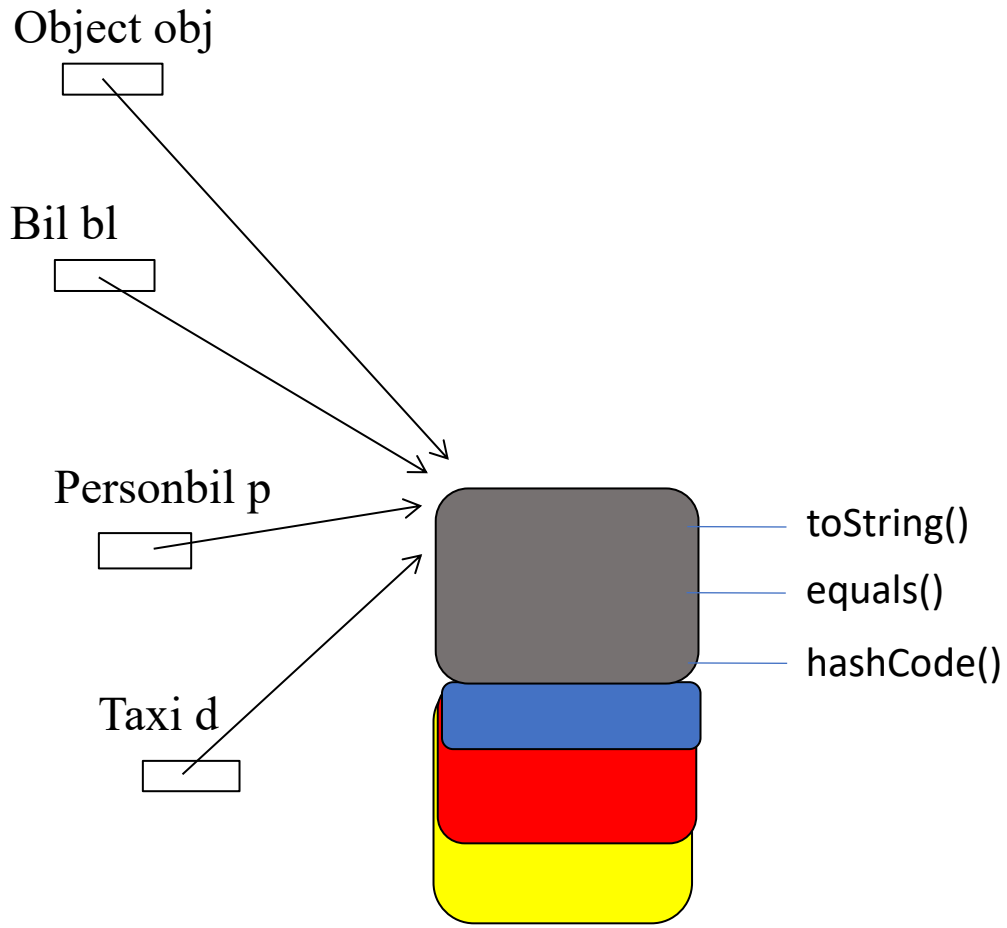
Menti – oppgave

Hvor er det naturlig å bruke komposisjon og hvor er det naturlig med arv ?

class Object

Klassehierarki:





Object: toString og equals

- Klassen Object inneholder bl.a. tre viktige metoder:
  - String toString( )
    returnerer en String-representasjon av objektet
  - boolean equals(Object o)
    sjekker om to objekter er like (i Object det samme som pekerlikhet)
  - int hashCode( )
    returnerer en hash-verdi av objektet
- Disse metodene kan man så selv redefinere til å gjøre noe mer fornuftig.
- Poenget er at en bruker av en klasse vet at disse metodene *alltid* vil være definert (pga. polymorfi)

Eksempel på toString og equals


```
class Punkt {
   protected int x, y;
   Punkt(int x0, int y0) {
      x = x0;
      y = y0;
   }
}
```

Anta:

```
Punkt p1 = new Punkt(3,4);
Punkt p2 = new Punkt(3,4);

Punkt2 q1 = new Punkt2(3,4);
Punkt2 q2 = new Punkt2(3,4);
```

Hva blir nå:

```
p1.toString();
```
Punkt@4d591d15

```
p1.equals(p2);
```
false

```
q1.toString();
```
x = 3 y = 4

```
q1.equals(q2);
```
true

```
class Punkt2 {
   protected int x, y;
   Punkt2(int x0, int y0) {
      x = x0;  y = y0;
   }
   @Override
   public String toString() {
      return ("x = "+x+" y = "+y);
   }
   @Override
   public boolean equals(Object o) {
       if (!(o instanceof Punkt2))
          return false;
       Punkt2 p = (Punkt2) o;
       return x == p.x && y == p.y;
   }
}
```

(4D591D15 er hash-koden til objektet i det heksadesimale (16-)tall-systemet = 1297685781 i 10-tall-systemet)

Konstruktører

Ikke noe nytt her

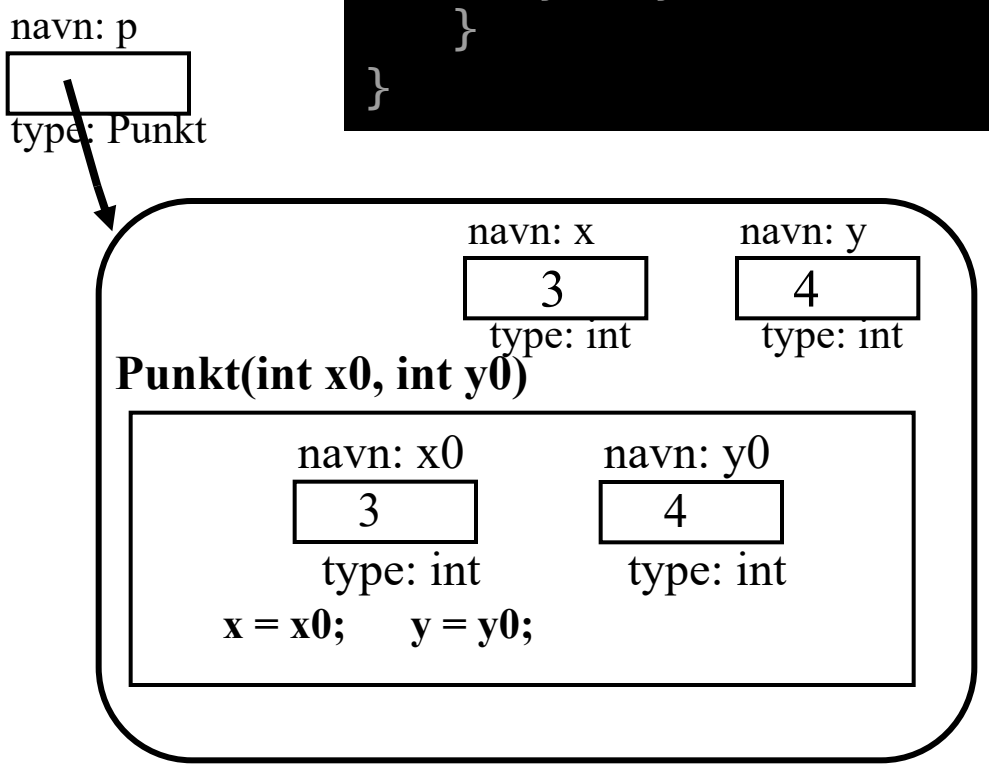
Bruk av konstruktører når vi opererer med "enkle" klasser er ganske ukomplisert. Når vi skriver

```
Punkt p  = new Punkt(3,4);
```

skjer følgende:

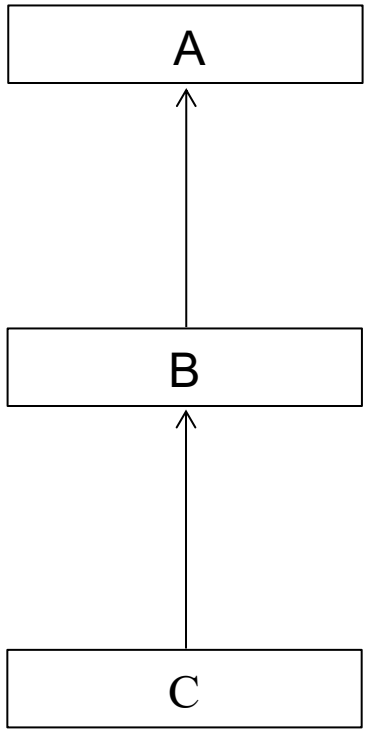
```
class Punkt {
   protected int x, y;
   Punkt(int x0, int y0) {
      x = x0;
      y = y0;
   }
}
```

1. Det settes av plass i intern-minnet til et objekt av klassen Punkt og til referansevariablen p.
2. Variablene x og y blir opprettet inne i objektet (instansvariable)
3. Konstruktør-metoden blir kalt med x0=3 og y0=4.
4. Etter at konstruktøren har satt x=3 og y=4, blir verdien av høyresiden i tilordningen
   Punkt p = new Punkt(3,4)
   adressen (en referanse, peker) til det nye objektet.
5. Tilordningen Punkt p = … utføres, dvs p settes lik adressen / referansen til objektet.



Nytt

Konstruktører blir noe mer komplisert når vi opererer med arv



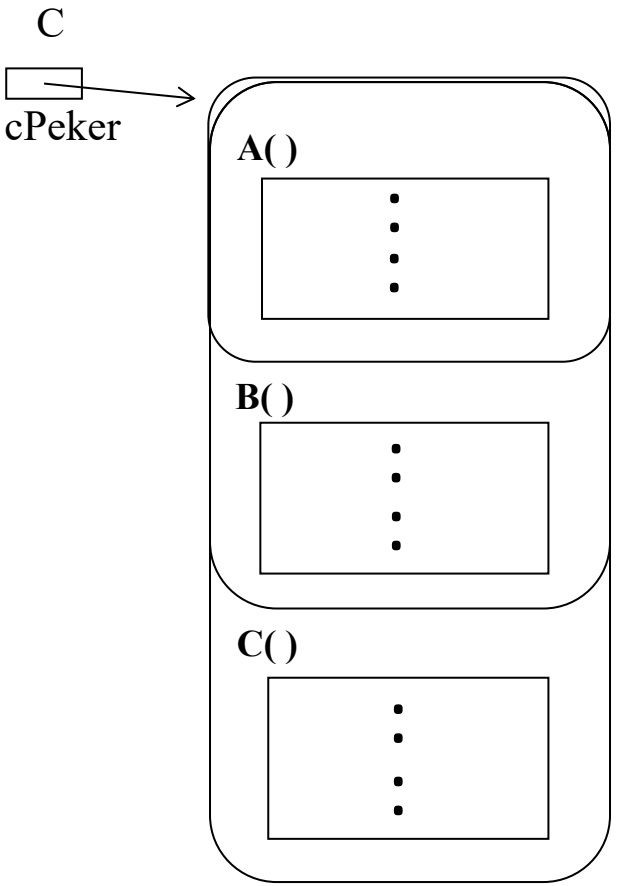
- Anta at vi har deklarert tre klasser:

```
class A { … }
class B extends A { … }
class C extends B { … }
```

- Hvilken / hvilke konstruktør utføres hvis vi skriver

```
C cPeker = new C();
```

?

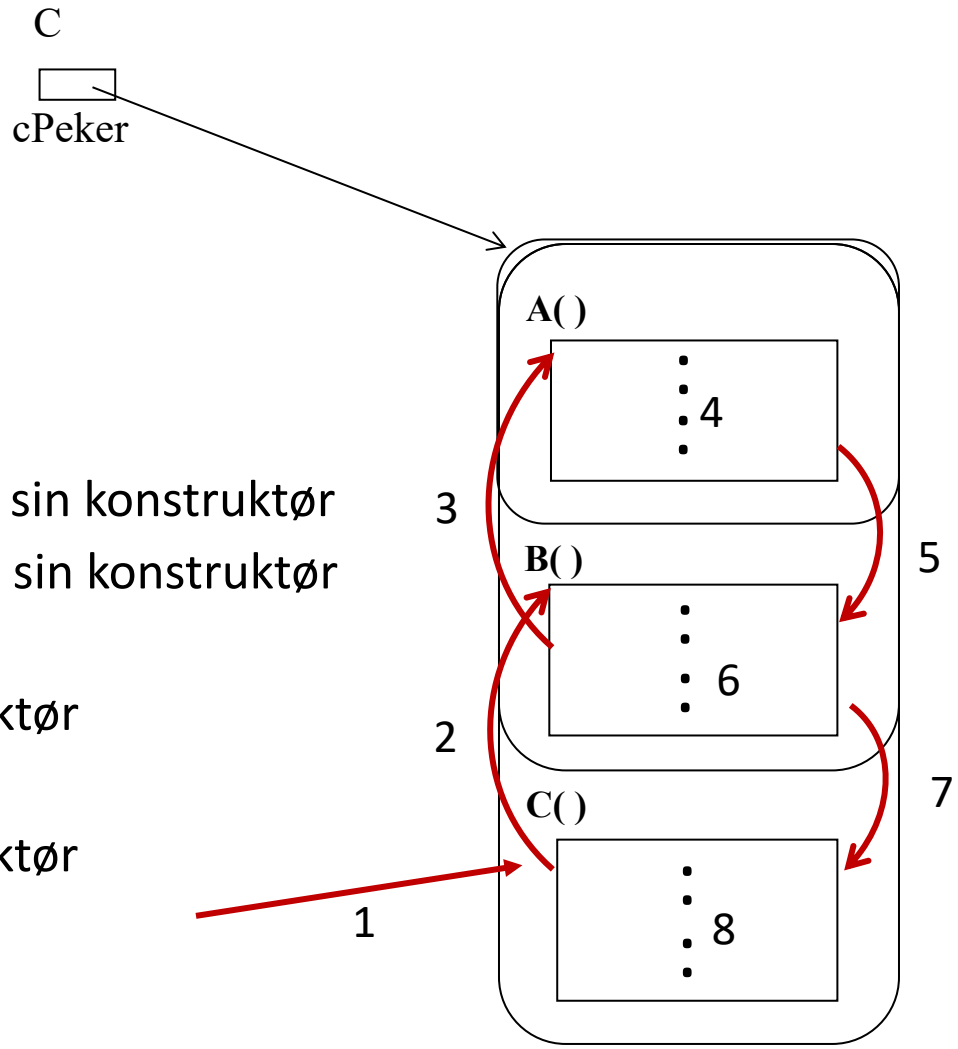


Konstruktører og arv virker slik:

- Anta at vi har deklarert tre klasser:

```
class A { … }
class B extends A { … }
class C extends B { … }
```

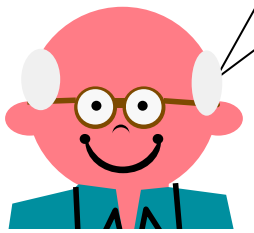
- Når vi skriver **`new C()`** skjer følgende:
  1. Konstruktøren til C kalles (som vanlig)
  2. Konstruktøren til C starter med å kalle på B sin konstruktør
  3. Konstruktøren til B starter med å kalle på A sin konstruktør
  4. Så utføres A sin konstruktør
  5. Kontrollen kommer tilbake til B sin konstruktør
  6. Så uføres B sin konstruktør
  7. Kontrollen kommer tilbake til C sin konstruktør
  8. Så uføres C sin konstruktør



Kall på superklassens-konstruktør

- Superklassens konstruktør kalles fra en subklasse ved å si:
  - `super();`
    - vil kalle på en konstruktør uten parametre
  - `super(5, "test");`
    - om vi vil kalle på en konstruktør med to parametre (int og String)
- Et kall på super **må** legges **helt i begynnelsen av** konstruktøren.
- Kaller man ikke super eksplisitt, vil Java **selv legge inn kall på super( )** helt først i konstruktøren når programmet kompileres.
- Hvis en klasse ikke har noen konstruktør, legger Java inn en tom konstruktør med kallet super();

NB! NB!
Det er forskjell på
super.
og
super(. . .)



Eksempel 1

Anta at vi har følgende klasser:

```
class Person {
    proteceted String fødselsnr;

    Person() {
       fødselsnr = "12030167890";
    }
}

class Student extends Person {
    protected int studID;

    Student() {...}
}
```

Anta to konstruktører:

```
Student() {
   super();
   studID = 42;
}
```

eller:

```
Student() {
   studID = 42;
}
```

Disse to er helt ekvivalente!

Hva skjer hvis Student ikke har noen konstruktør : ?

```
class Student extends Person {
    protected int studID = 42;
}
```

Eksempel 2

Anta at vi har følgende klasser:

```
class Person {
    protected String fødselsnr;

    Person(String fnr) {
        fødselsnr = fnr;
    }
}

class Student extends Person {
    protected int studID;

    Student() {...}
}
```

Her er noen forslag til konstruktører:

```
Student() {
    studID = 42;
}
```

```
Student() {
    super("12030145787");
    studID = 42;
}
```

```
Student(String nr){
    super(nr);
    studID = 42;
}
```

```
Student(String nr, int id){
    super(nr);
    studID = id;
}
```

😊

Eksempel 3



```
class Bygning {
   Bygning() {
      System.out.println("Bygning");
   }
}

class Bolighus extends Bygning {
   Bolighus() {
      System.out.println("Bolighus");
   }
}

class Blokk extends Bolighus {
   Blokk() {
      System.out.println("Blokk");
   }

   public static void main(String[] args) {
      new Blokk();
   }
}
```

Hva blir utskriften fra dette programmet?

Når programmet kompileres


```
class Bygning {
   Bygning() {
      System.out.println("Bygning");
   }
}

class Bolighus extends Bygning {
   Bolighus() {
      System.out.println("Bolighus");
   }
}

class Blokk extends Bolighus {
   Blokk() {
      System.out.println("Blokk");
   }

   public static void main(String[] args) {
      new Blokk();
   }
}
```

Java føyer selv på 'super()' i disse tre konstruktørene før programmet utføres

Når programmet utføres


```
class Bygning {
   Bygning() {
      super( );
      System.out.println("Bygning");
   }
}

class Bolighus extends Bygning {
   Bolighus() {
      super( );
      System.out.println("Bolighus");
   }
}

class Blokk extends Bolighus {
   Blokk() {
      super( );
      System.out.println("Blokk");
   }

   public static void main(String[] args) {
      new Blokk();
   }
}
```

5.

Til Object sin konstruktør

4.

3.

2.

Her starter eksekveringen

1.

Når programmet utføres (forts.)


```
class Bygning {
   Bygning() {
      super( );
      System.out.println("Bygning");
   }
}

class Bolighus extends Bygning {
   Bolighus() {
      super( );
      System.out.println("Bolighus");
   }
}

class Blokk extends Bolighus {
   Blokk() {
      super( );
      System.out.println("Blokk");
   }

   public static void main(String[] args) {
      new Blokk();
   }
}
```

6.

Tilbake fra Object sin konstruktør

7.

Nå er **Bygning** skrevet ut

8.

Nå er **Bolighus** skrevet ut

9.

Nå er **Blokk** skrevet ut

Eksempel 4


```
class Bygning {
   Bygning() {
      System.out.println("Bygning");
   }
}

class Bolighus extends Bygning {
   Bolighus(int i) {
      System.out.println("Bolighus nr " + i);
   }
}

class Blokk extends Bolighus {
   Blokk() {
      System.out.println("Blokk");
   }

   public static void main(String[] args) {
      new Blokk();
   }
}
```

Hva skjer i dette eksemplet?

Merk: Konstruktøren i klassen Bolighus har nå en parameter.

Når programmet kompileres


```
class Bygning {
   Bygning() {
      super( );
      System.out.println("Bygning");
   }
}

class Bolighus extends Bygning {
   Bolighus(int i) {
      super( );
      System.out.println("Bolighus");
   }
}

class Blokk extends Bolighus {
   Blokk() {
      super( );
      System.out.println("Blokk");
   }

   public static void main(String[] args) {
      new Blokk();
   }
}
```

Java legger igjen til kall på super() i alle konstruktørene.

Men: Super-kallet matcher ikke Bolighus-konstruktøren i antall parametre!

Mulige løsninger:

1. Selv legge til kall på super, med argument, i kontruktøren i Blokk.
2. Legge til en tom konstruktør i Bolighus.

Flere konstruktører


```
class Bygning {
   Bygning() {
      System.out.println("Bygning");
   }
}

class Bolighus extends Bygning {
   Bolighus() {
      System.out.println("Bolighus");
   }
   Bolighus(int i) {
      System.out.println("Bolighus nr " + i);
   }
}

class Blokk extends Bolighus {
   Blokk() {
      System.out.println("Blokk");
   }

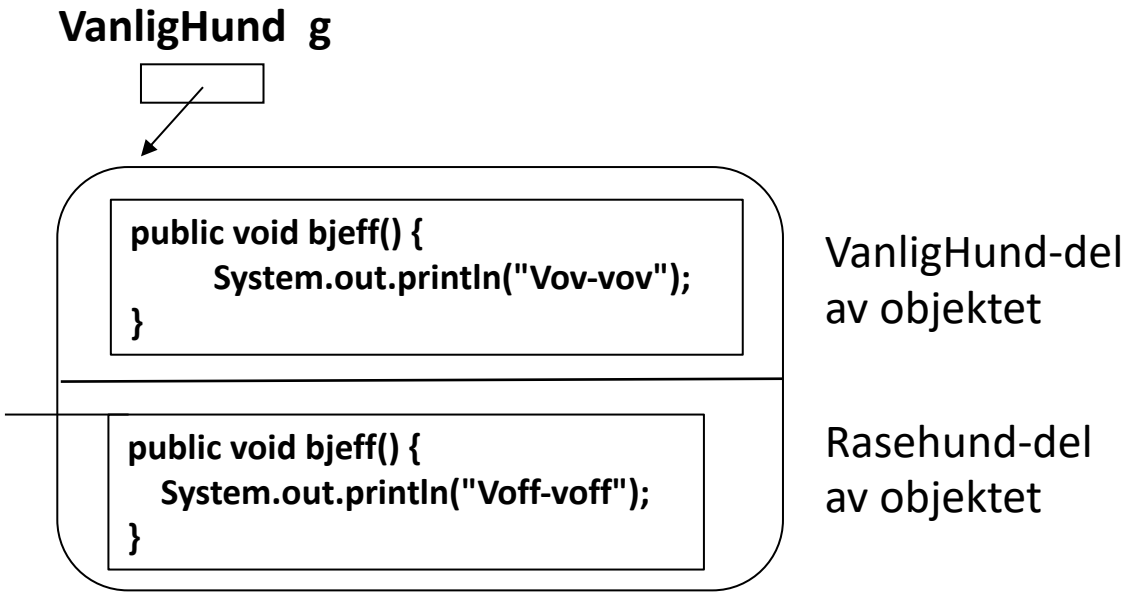
   public static void main(String[] args) {
      new Blokk();
   }
}
```

Da brukes den konstruktøren som her de samme parametrene som på kallstedet (samme som overlasting)

Hoved "take-away" i dag (polymorfi)



Dette er en rasehund i virkeligheten
(i alle fall et bilde av en)



Dette er et rasehundobjekt
inne i datamaskinen

g.bjeff( ) gir Voff-voff

Oppsummering

- Polymorfi lar objekter gjøre det de "egentlig" kan:
  - At kallet kan utføres bestemmes av kompilatoren
  - Hvilken metode som utføres bestemmes av kjøretidsystemet
    - Det er den metoden i objektet som er definert dypest i klassehierarkiet som utføres
- Metodene må da ha samme signatur
  - Dette kalles redefinering – engelsk: override
- Samme navn - ulik signatur – kalles overlasting (engelsk: overload)
  - Sett fra Java kunne de like gjerne hatt forskjellig navn
- Det er forskjell på super. og super( )
- Pass på konstruktører i subklasser (reglene på lysark 44)

UNIVERSITETET I OSLO

Institutt for informatikk

IN1010 våren 2022

Tirsdag 15. februar

Subklasser III: Interface - Grensesnitt

+ Litt om invarianter

Stein Gjessing

Dagens hovedtema

- Engelsk: Interface (også et Java-nøkkelord)
- Norsk: Grensesnitt

- Les notatet “Interface i Java” av Stein Gjessing

- To måter å bruke interface på (men det er mye av det samme)
  1) Multippel arv av oppførsel / roller
  2) Definere en oppførsel / rolle

Multippel arv: Om å arve fra flere

- I Java kan en klasse bare arve egenskapene til **én** annen klasse (en superklasse).
  - Dette gjør språket sikrere å bruke
- Hva skal vi gjøre hvis vi ønsker at et objekt skal inneholde mange forskjellige egenskaper fra forskjellige "superklasser" ?

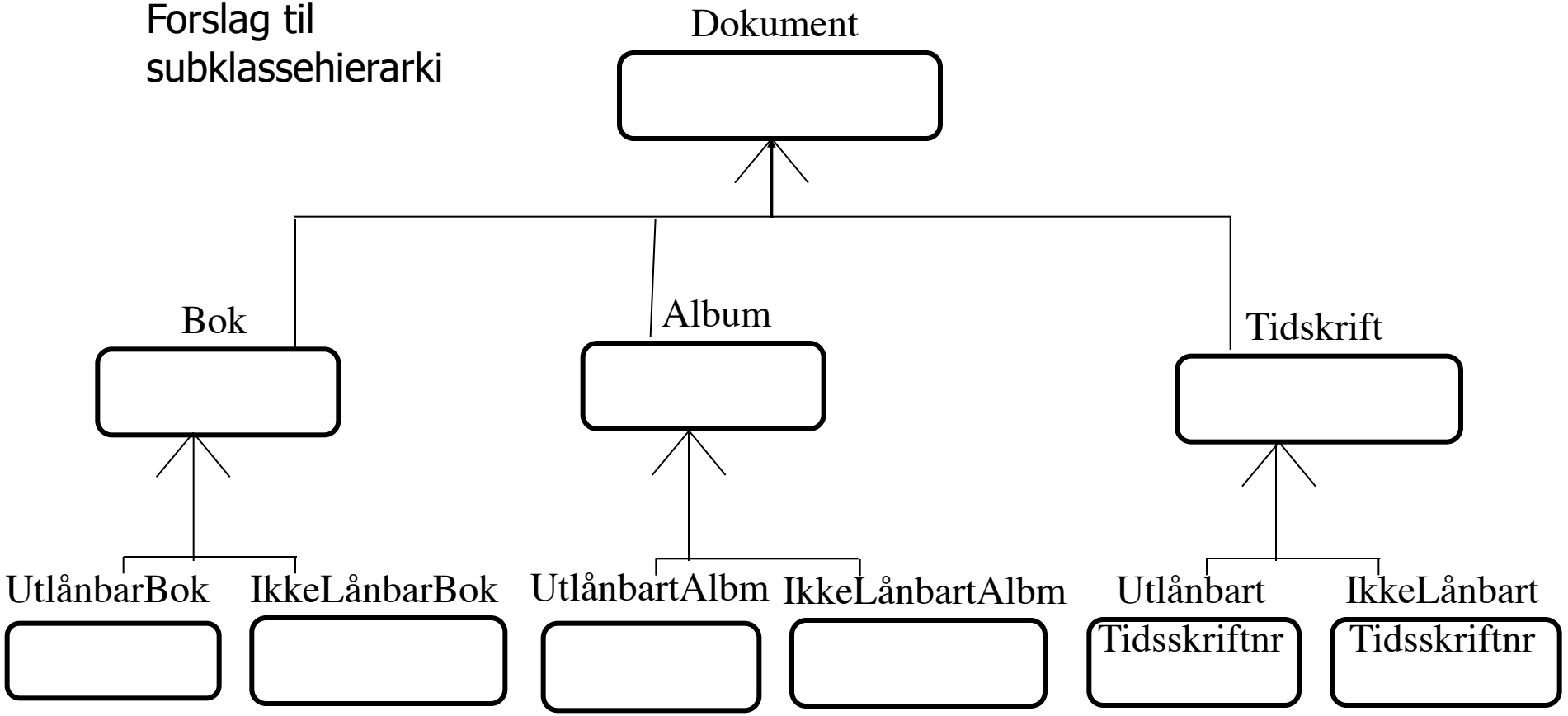
- På de neste sidene:
  - Begrepshierarkiet i et bibliotek

Motivasjon for mekanismen interface: Analyse av et bibliotek

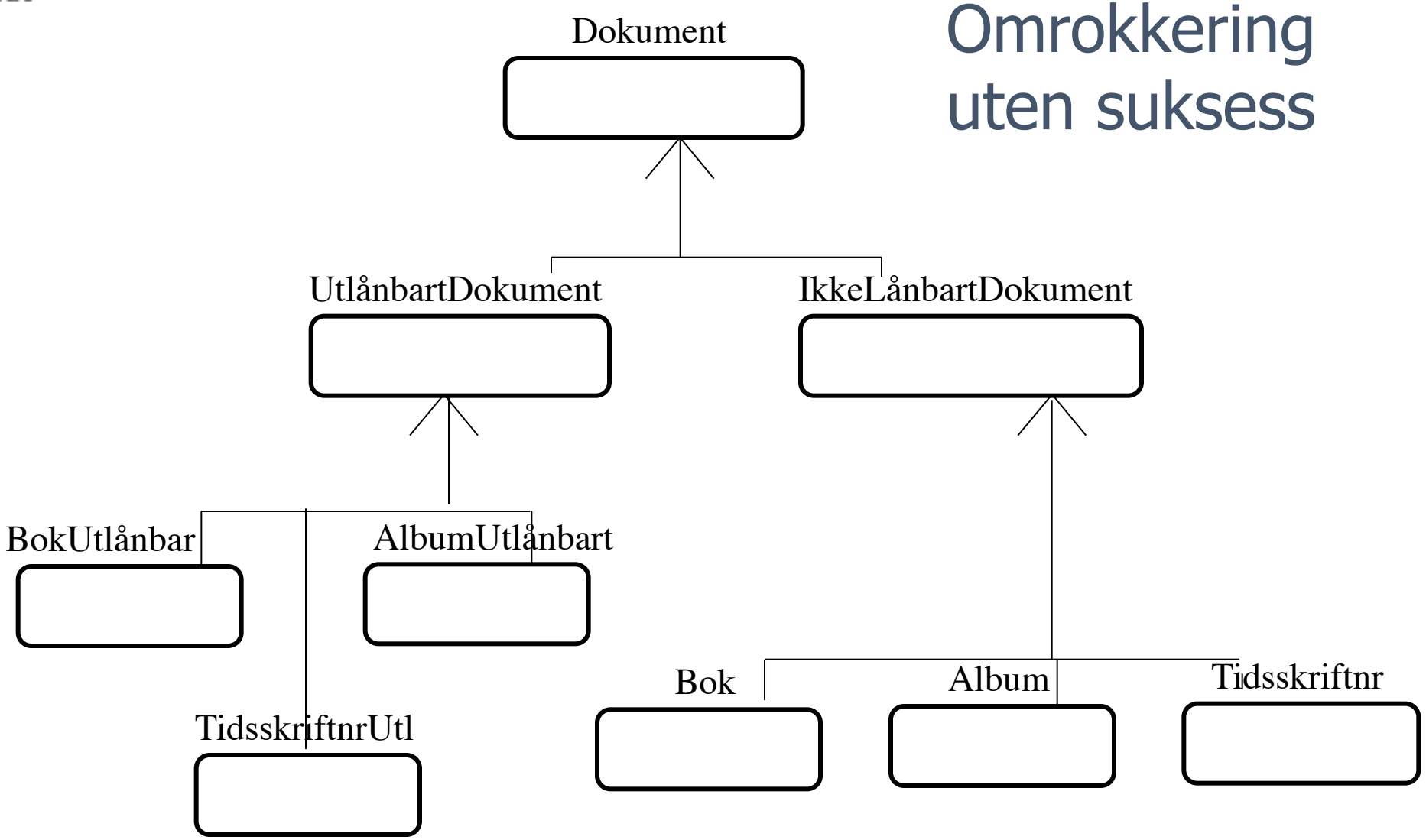
- Bøker, tidsskrifter, musikk, videoer, mikrofilmet materiale, antikvariske bøker, flerbindsverk, oppslagsverk, upubliserte skrifter, …
- En del felles egenskaper
  - antall eksemplarer, hylleplass, identifikasjonskode (Dokument)
  - for det som kan lånes ut: Er utlånt ? , navn på låner, ... (TilUtlån)
  - for det som er antikvarisk: Verdi, forskringssum, ... (Antikvarisk)
- Spesielle egenskaper:
  - Bok: Forfatter, tittel, forlag
  - Tidsskriftnummer: Årgang, nummer, utgiver
  - Album: Tittel, artist, komponist, musikkforlag

Tvilsomt begrepshierarki

Forslag til subklassehierarki

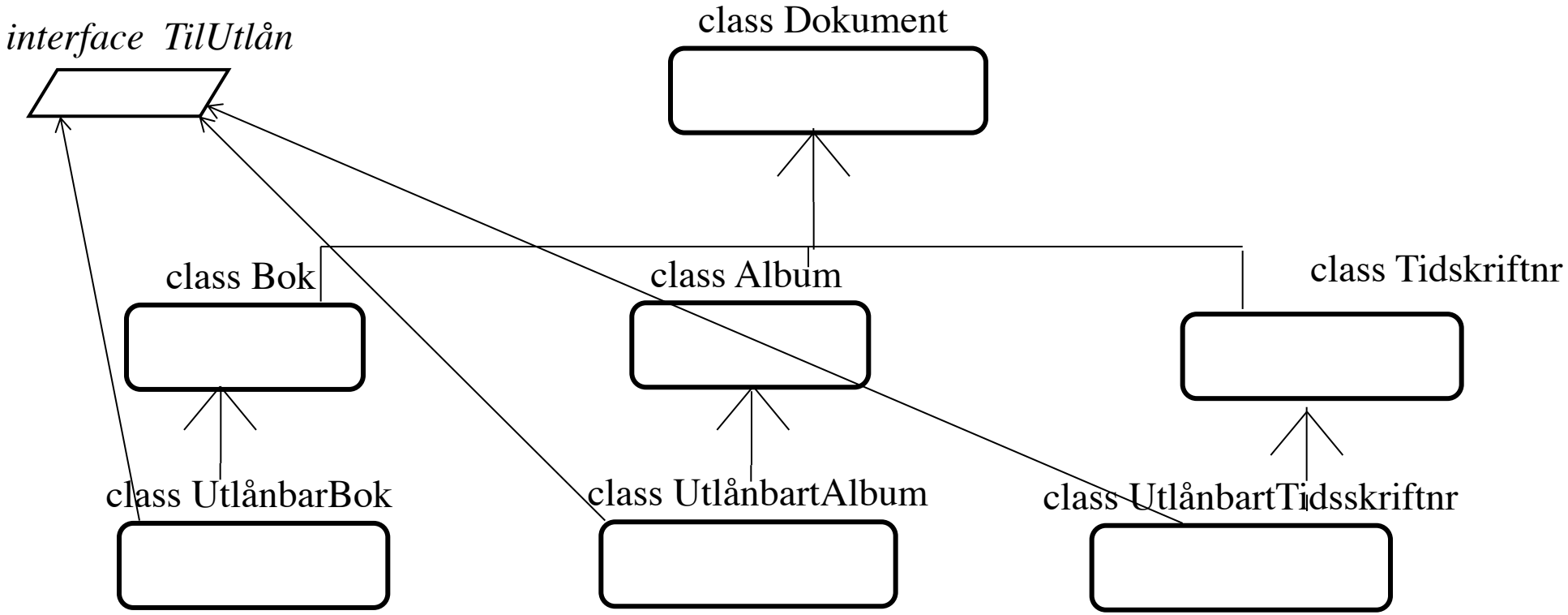


Omrokkering uten suksess



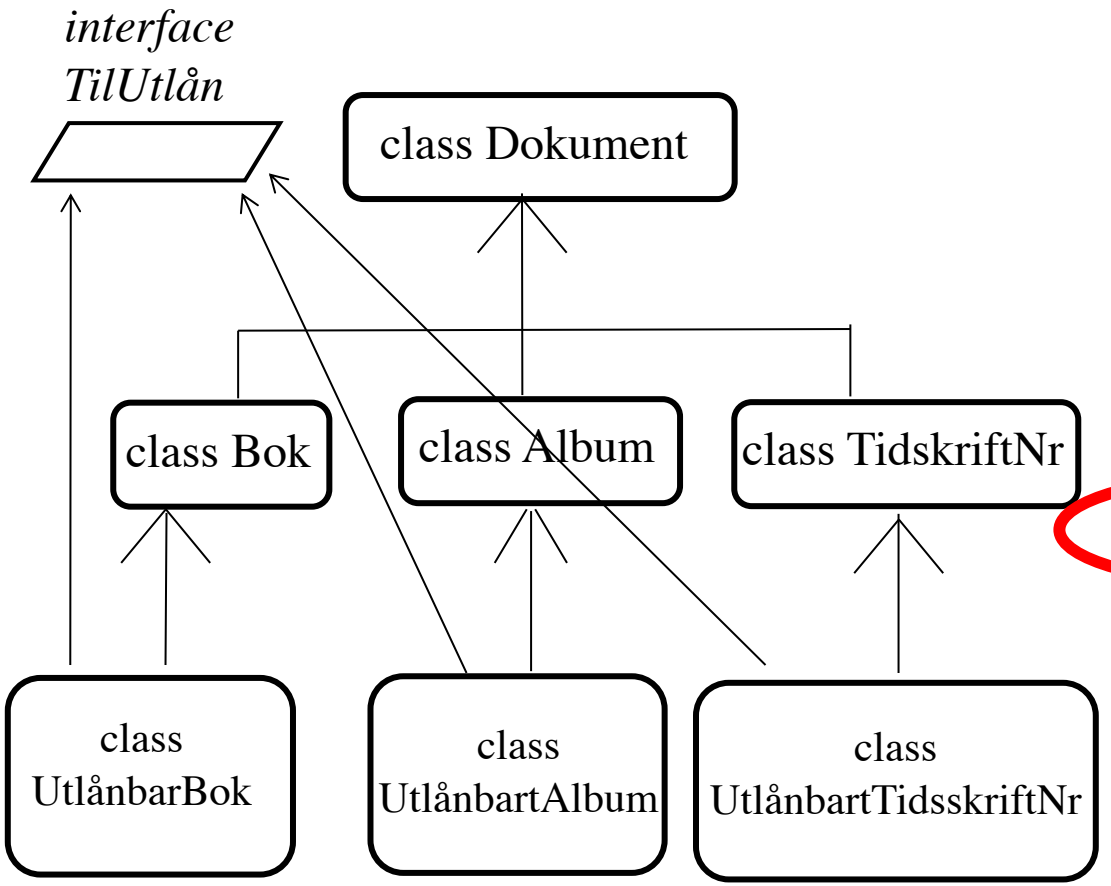
Samle lik oppførsel: bruk **interface**

NYTT



- Felles egenskaper (rolle) på tvers av klassehierarkiet
- En klasse kan tilføres et (eller flere) interface
  - i tillegg til å arve egenskapene i klassehierarkiet
- Dvs. en klasse kan spille to (eller flere) **roller**

Vi tar det en gang til, litt saktere: Dokumentene i et bibliotek




```
class Dokument {   }

class Bok extends Dokument  {   }

class Album extends Dokument {    }

class  TidskriftNr extends Dokument {   }

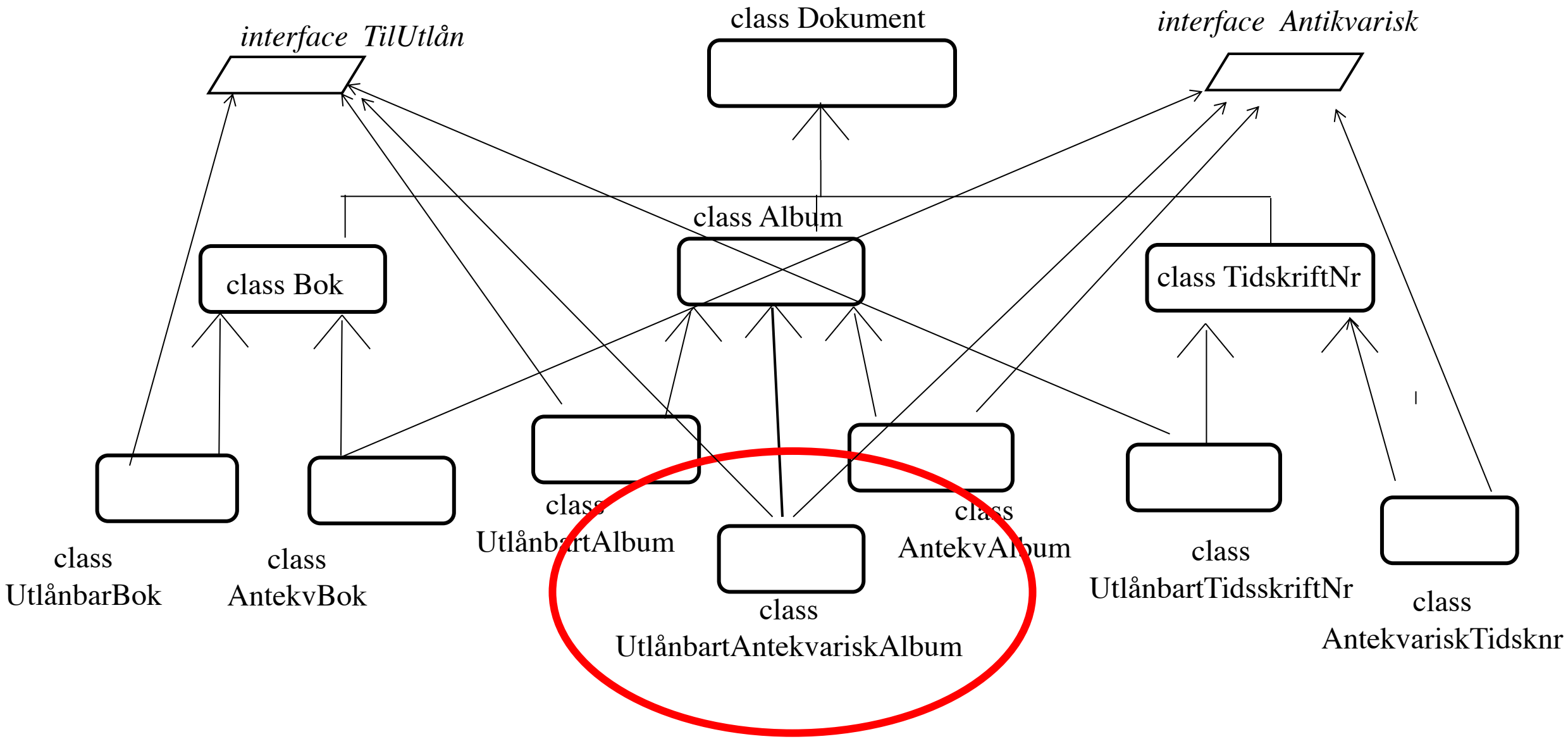
interface TilUtlån {    }

class UtlånbarBok extends Bok implements TilUtlån {  }

class UtlånbartAlbum extends Album implements TilUtlån {  }

class UtlånbartTidskriftNr extends TiskriftNr implements TilUtlån { }
```

En klasse kan arve fra flere **interface**



Men hva arves fra et interface?


```
class Dokument {   }

class Bok extends Dokument  {   }

class Album extends Dokument {    }

class  TidskriftNr extends Dokument {   }

interface TilUtlån {    }

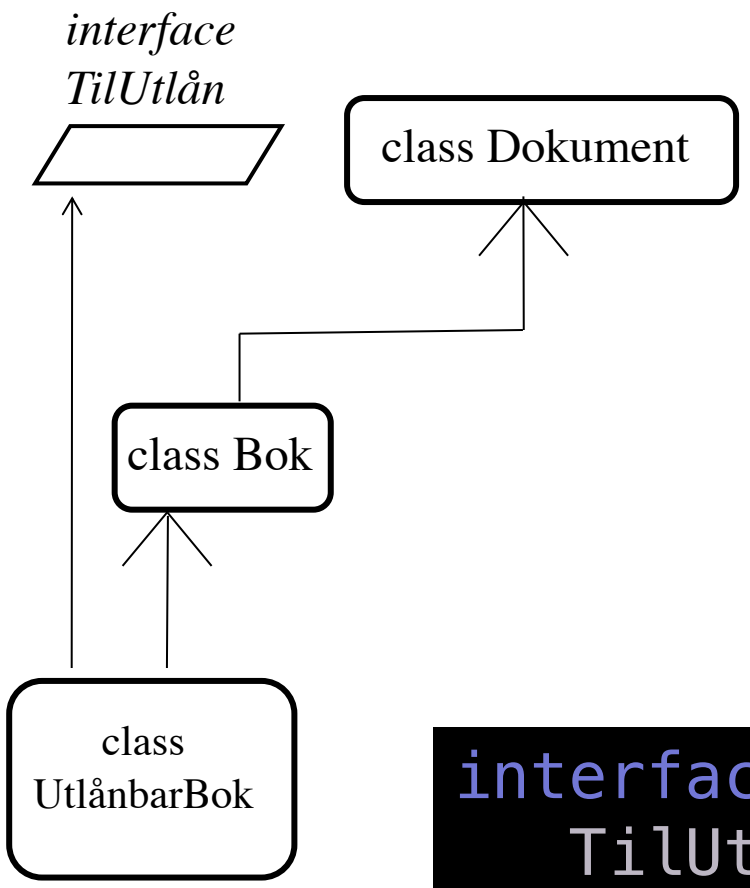
class UtlånbarBok extends Bok implements TilUtlån {  }

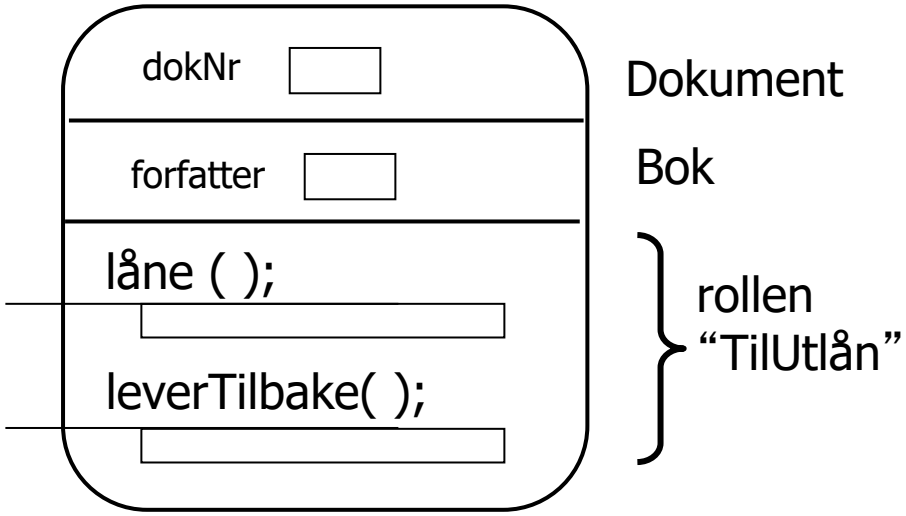
class UtlånbartAlbum extends Album implements TilUtlån {  }

class UtlånbartTidskriftNr extends TiskriftNr implements TilUtlån { }
```

Men hva arves fra et grensesnitt?

Svar: En rolle (metoder)



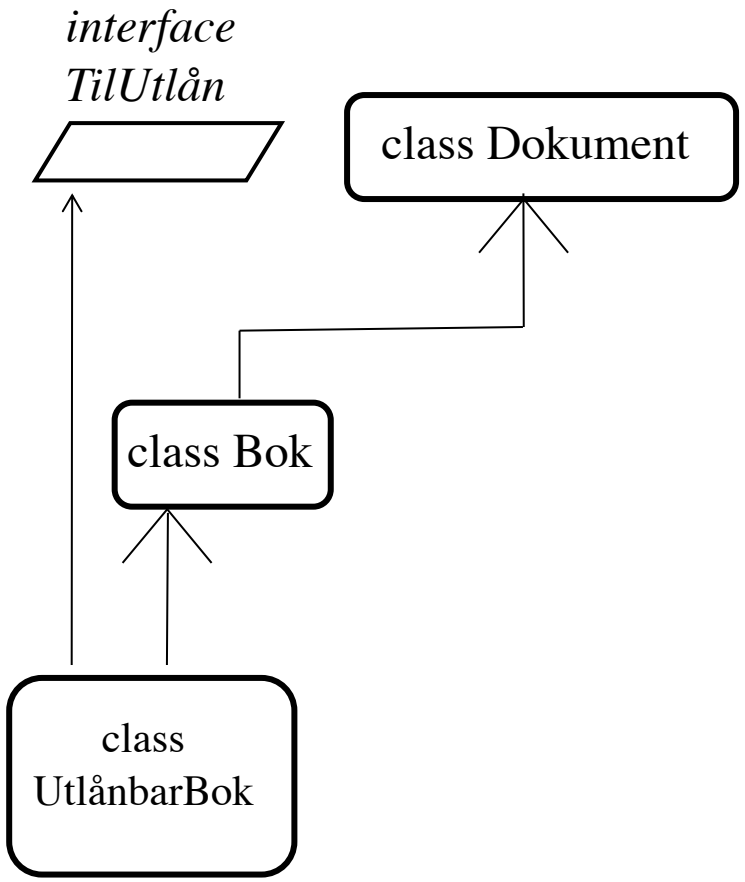


Et objekt av klassen UtlånbarBok

```
interface TilUtlån {
   TilUtlån låne(String låner);
   void leverTilbake();
}
```

Men hva arves fra et grensesnitt?

Svar: En rolle (metoder – metodesignaturer)




```
class Dokument {final int dokNr;}

class Bok extends Dokument {final String forfatter;}

interface TilUtlån {
   TilUtlån låne(String låner);
   void leverTilbake();
}


class UtlånbarBok extends Dokument implements TilUtlån {
   private String låner = null;
   @Override
   public TilUtlån låne(String lnr) { . . . }
   @Override
   public void leverTilbake() { . . . }
}
```



Subklassen må selv *implementere* metodene

Men hva arves fra et grensesnitt?

Svar: En rolle (metoder – metodesignaturer)


```
class Dokument {final int dokNr;}

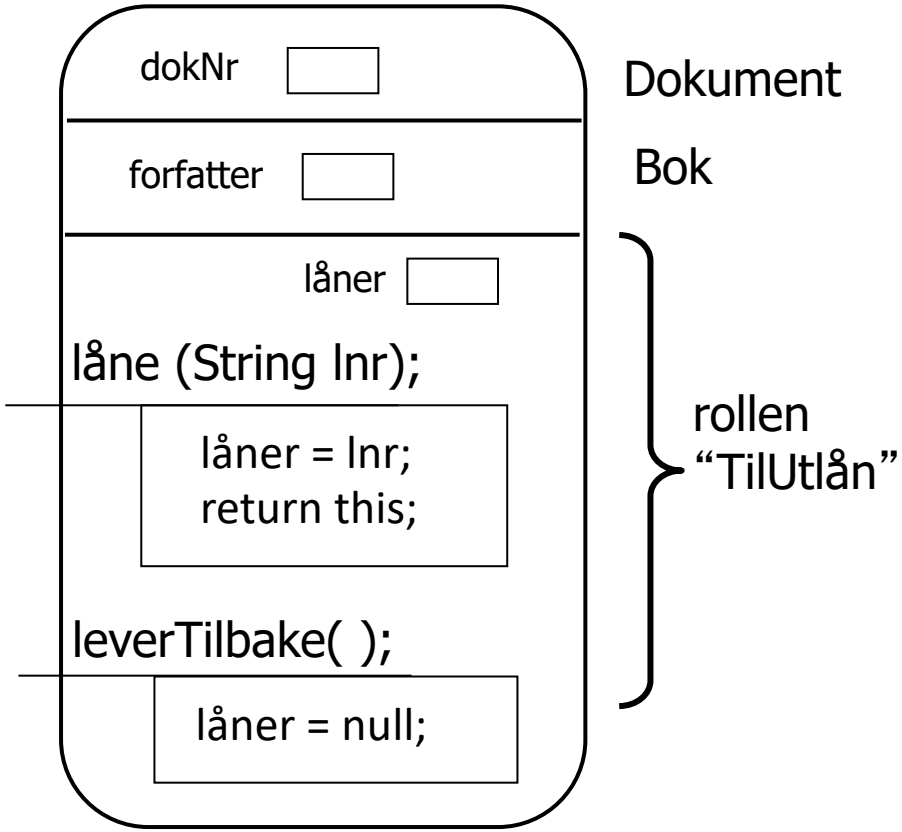

class Bok extends Dokument {final String forfatter;}


interface TilUtlån {
   TilUtlån låne(String låner);
   void leverTilbake();
}

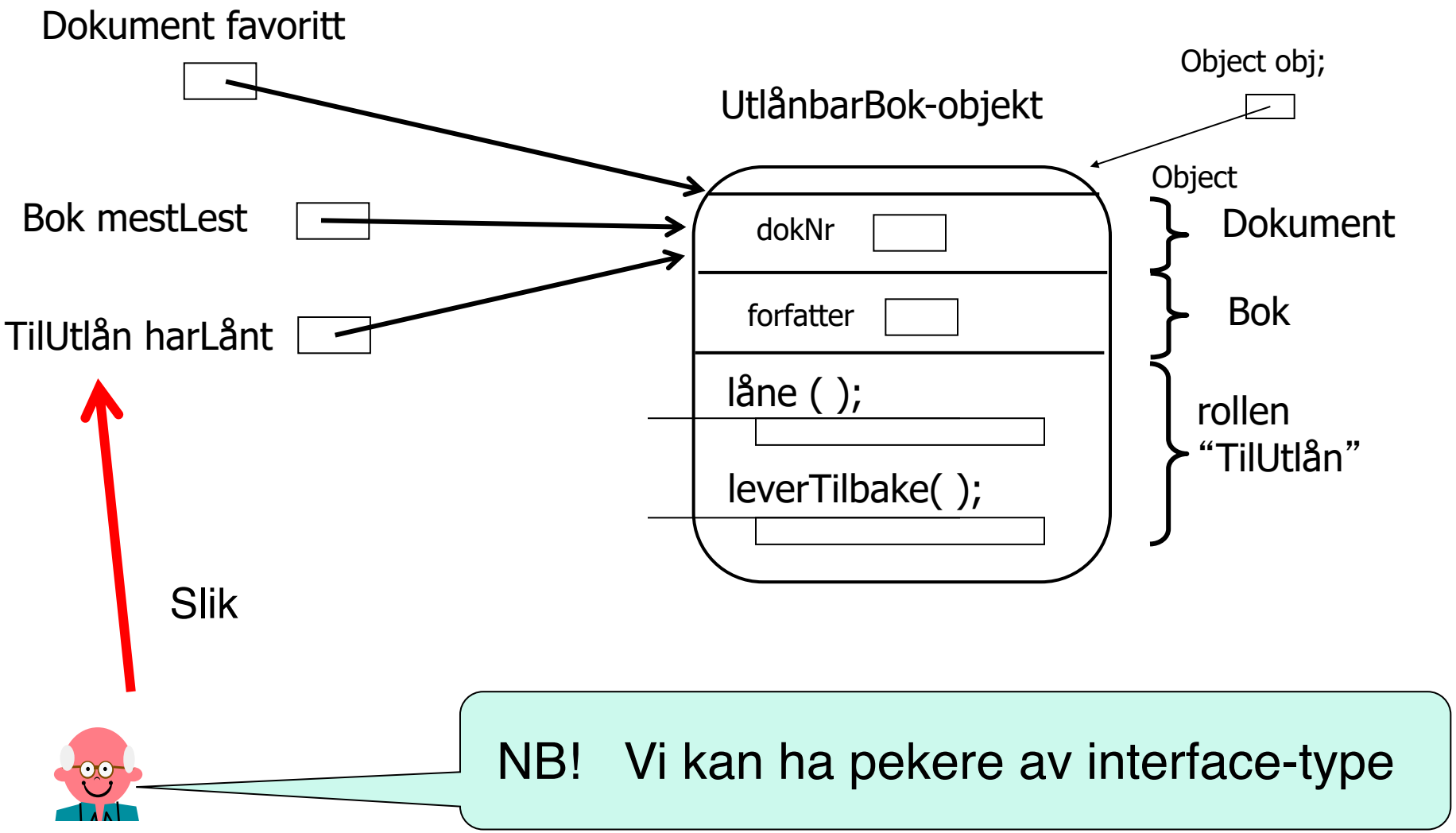

class UtlånbarBok extends Bok implements TilUtlån {
   private String låner = null;

   public TilUtlån låne(String lnr) {
      låner = lnr;
      return this;
   }

   public void leverTilbake() {låner = null;}
}
```



Et objekt av klassen UtlånbarBok


Dokument favoritt
Object obj;
UtlånbarBok-objekt
Object
Bok mestLest
dokNr
Dokument
forfatter
Bok
TilUtlån harLånt
låne ( );
rollen
"TilUtlån"
leverTilbake( );
Slik
NB! Vi kan ha pekere av interface-type

Et interface (grensesnitt) er (1):

- En samling egenskaper (en rolle) som ikke naturlig hører hjemme i et arve-hierarki
- En samling egenskaper som mange forskjellige "ting" av forskjellige typer kan anta
- En klasse kan arve egenskapene til mange grensesnitt (men bare en klasse)

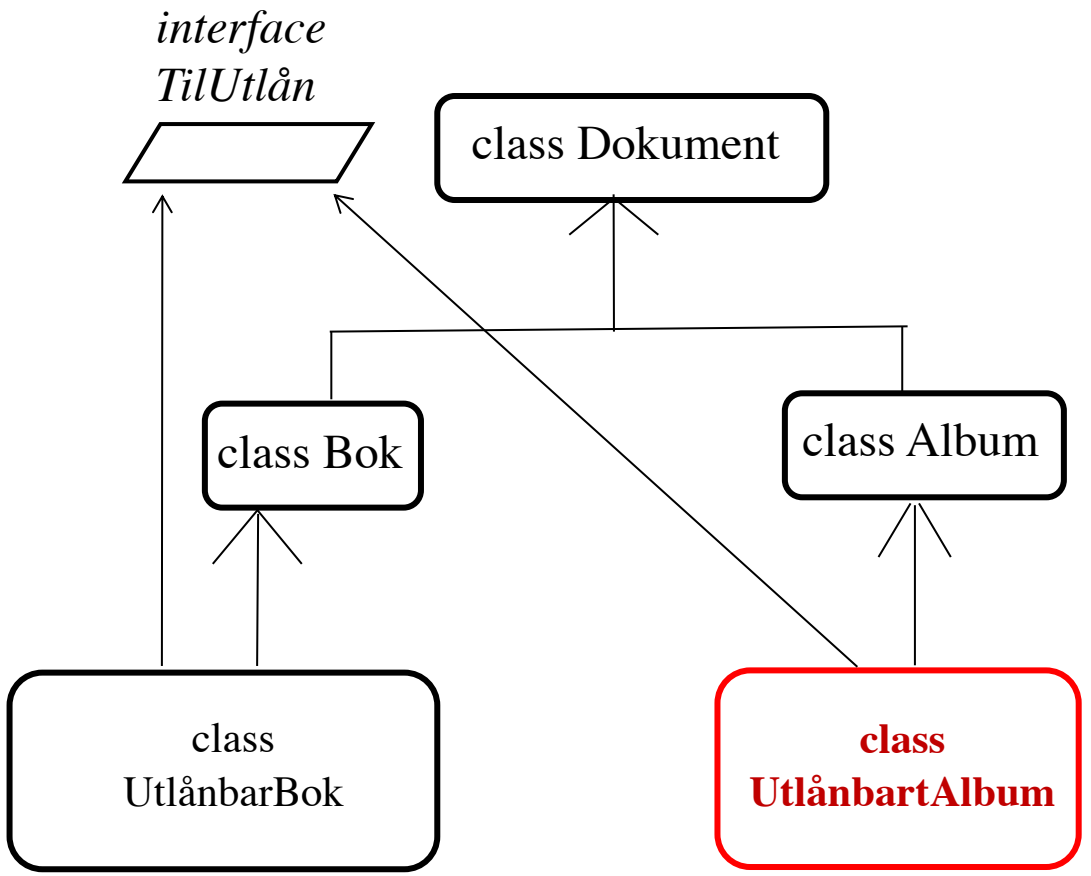
- For eksempel
  - Her: **Kan lånes ut (biler, bøker, festklær, …)**
    Antikvarisk (møbler, bøker, ….)
  - Kan delta i konkurranse (startnummer, resultat, ..
    Mennesker, biler, hester kan delta i konkurranser)
  - Svømmedyktig (mennesker, fugler er svømmedyktige)
  - Sammenlignbar (Comparable) (I Java-biblioteket – kommer vi tilbake til)
  - . . .

Et interface (grensesnitt) er (2):

- Et interface ligner en abstrakt klasse
- **Alle** metodene i et interface er abstrakte og polymorfe
- En interface inneholder **ingen** variabler eller annen datastruktur
  (men litt annet som vi ikke bruker i IN1010)
- En klasse som arver egenskapene til et interface må selv putte inn kode i alle de abstrakte metodene (og deklarere passende variabler som disse metodene bruker for å gjøre jobben sin).
- En klasse kan arve egenskapene til mange grensesnitt (men bare en klasse)

- Å arve (en samling metoder) = å spille en rolle
- Husk, med interface arves bare metodesignaturer
  - Implementasjon arves ikke

På IN1010s hjemmeside: BrukBibliotek.java



Vi skal programmere denne

Kommentar lagt til etter forelesningen: Legg merke til hvordan Stein skriver Bok istedenfor Album som superklasse til UtlånbartAlbum i koden han skriver. Dette er en feil som blir rettet opp til slutt, men det er en programmeringsfeil som først ikke blir fanget opp av det lille testprogrammet. Moral: Programmet ditt kan lett innholde feil som ikke blir oppdaget før du gjør meget grundig testing.

Nytt interface-eksempel

Hvis vi ønsker at noen objekter også skal kunne spille rollene (ha / arve egenskapene)
"**Skattbar**" (en ting vi må skatte av) og
"**Miljøvennlig**" (en ting som er miljøvennelig) kan vi ha:

```
interface Skattbar {
    double  toll();
    double moms();
}
```

Rollen Skattbar

```
interface Miljovennlig {
    int cO2Utslipp ();
    boolean svaneMerket ();
}
```

Rollen Miljøvennlig

Tegning av interfacer i klassehierarkiet


```
interface Skattbar {

}
```

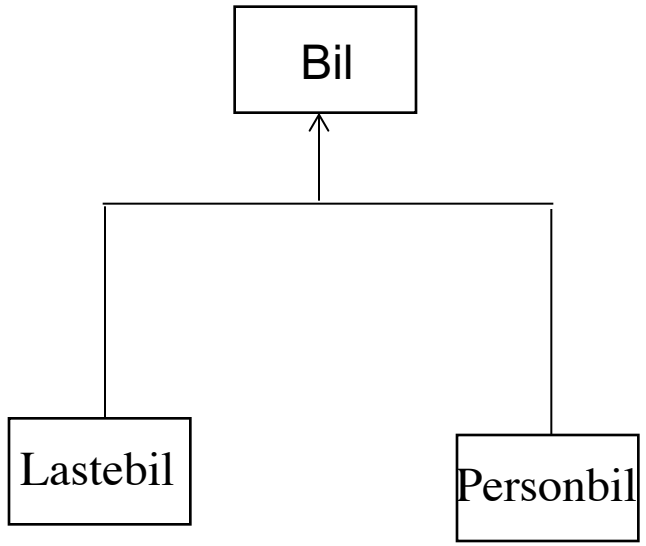



```
interface Miljøvennlig {

}
```



Slik var bil-hierarkiet

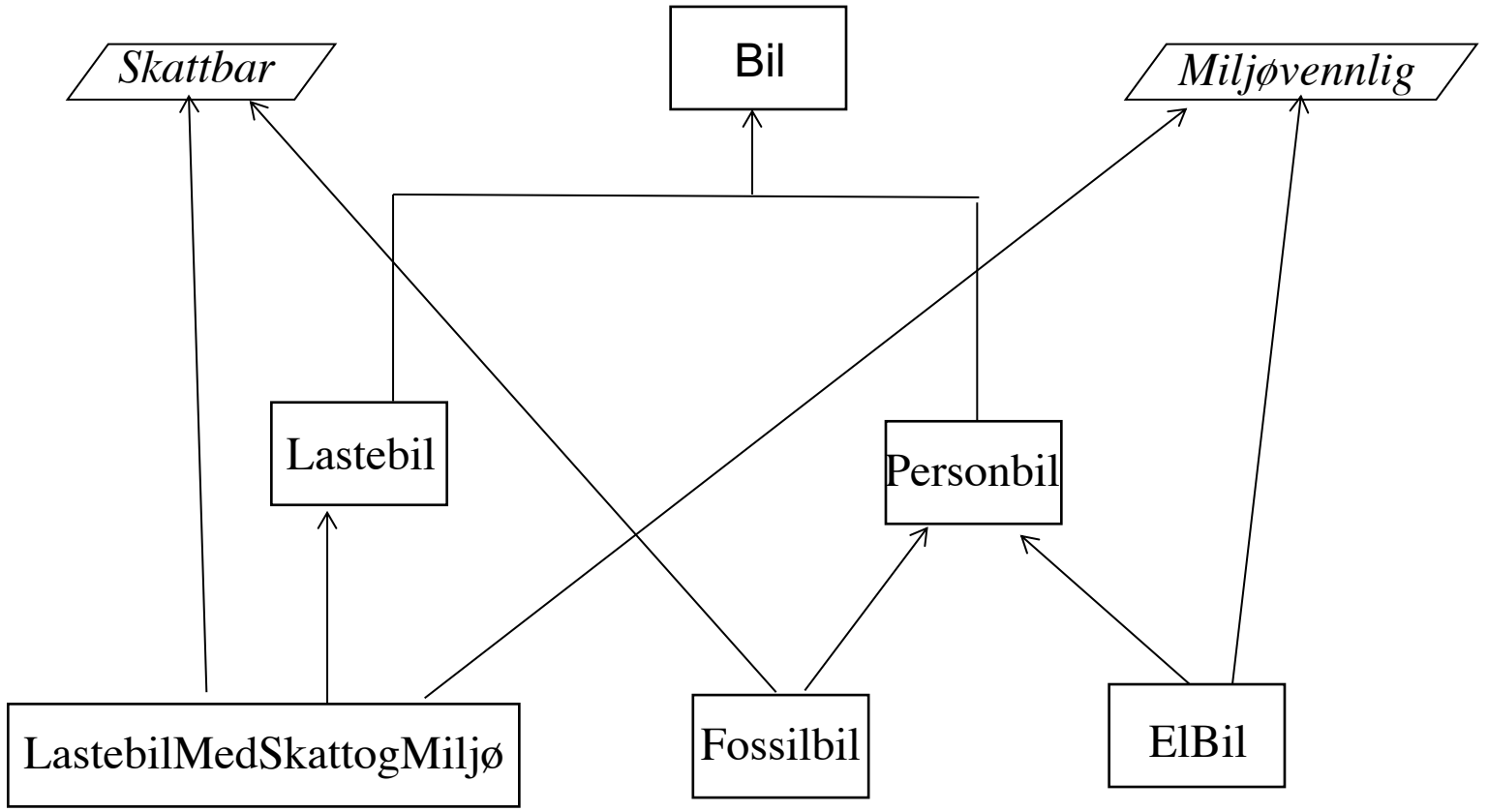


Mange biler må betale skatt og noen biler er miljøvennlige:

Skattbar

Miljøvennlig

Tre nye klasser som kan spille mange roller



Metodene må skrives på nytt hver gang de implementeres,
men det er f.eks. OK når skatt i en lastebil er forskjellig fra skatt i en fossilbil

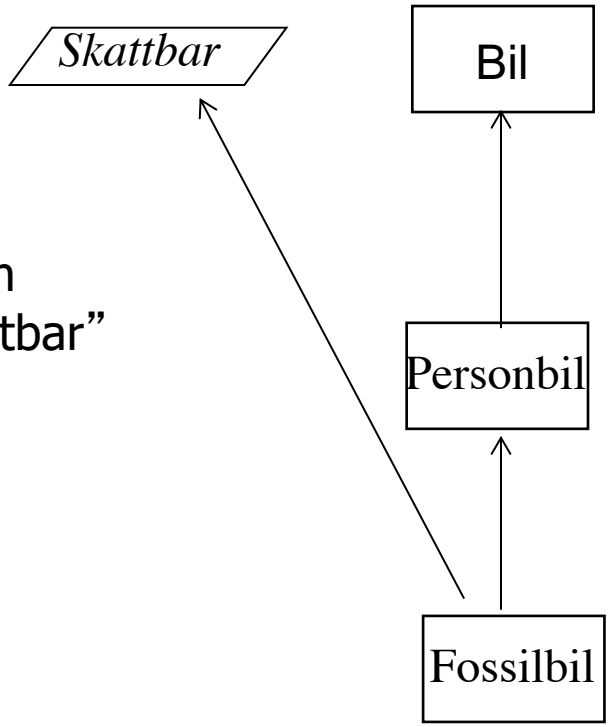


Om å implementere ett interface

rollen Personbil (i arv) fra klassehierarkiet

```
class Fossilbil extends Personbil implements Skattbar {
   protected double utslipp = 100;
   protected double innPris = 150000;
   @Override
   public double toll(){return (innPris*utslipp/200)* moms();}
   @Override
   public double moms( ){return 1.25;}
}
```

rollen “Skattbar”

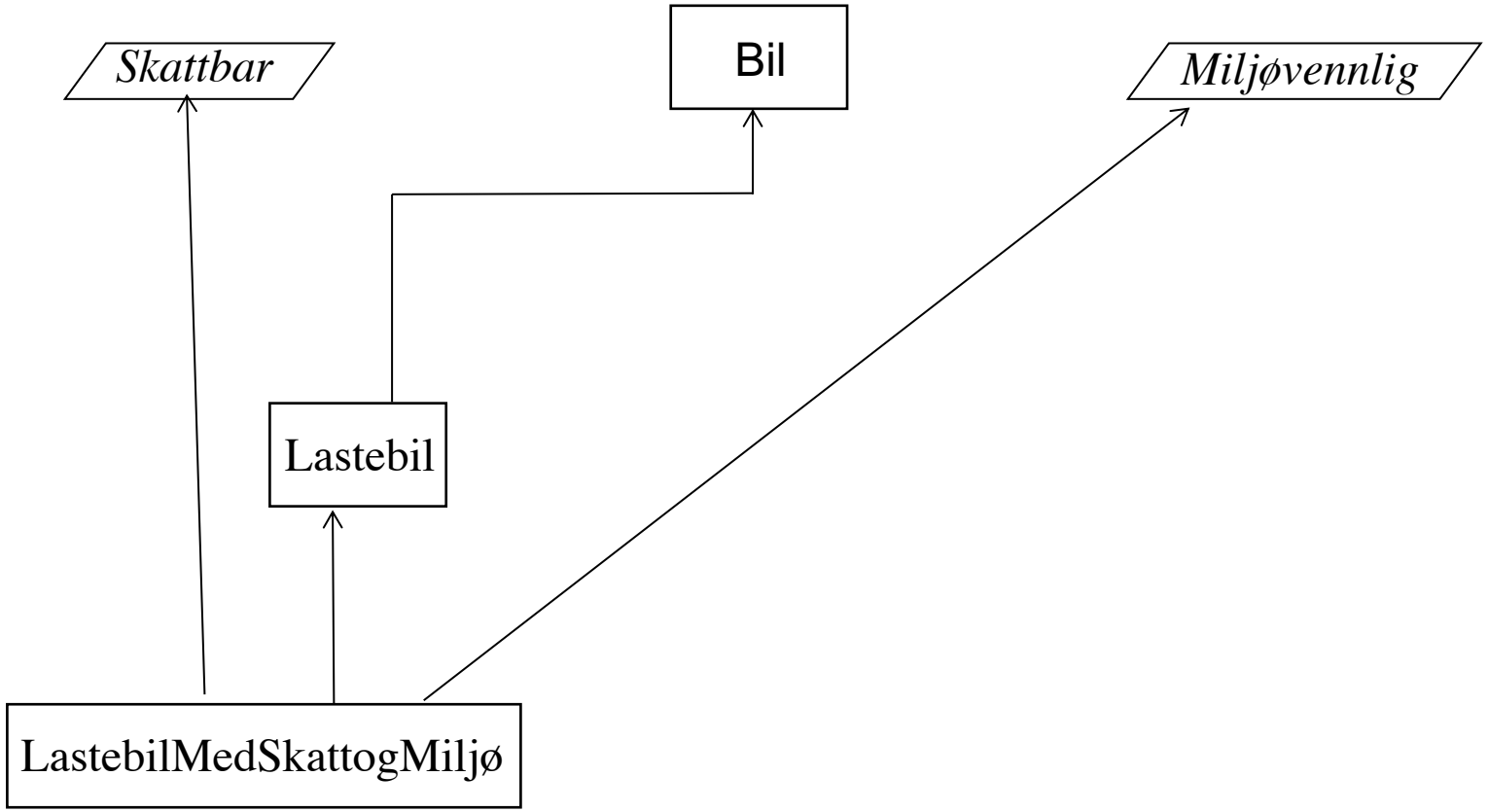



```
interface Skattbar {
    double toll();
    double moms();
}
```

```
class Bil {
    final String regNr;
}
```

```
class Personbil extends Bil {
    protected  int antPass;
}
```

Om å implementere to interfacer



UNIVERSITETET I OSLO

Om å implementere to interfacer

Institutt for informatikk

Lastebil-rollen i (arv fra) klassehierarkiet

```
class LastebilMedSkattogMiljø extends Lastebil implements Miljøvennlig, Skattbar {
   protected double innkjøpspris = 200000;
   protected int utslipp = 400;
   @Override
   public double toll(){return innkjøpspris*0.1;}
   @Override
   public double moms(){return 1.25;}
   @Override
   public int cO2Utslipp(){return utslipp;}
   @Override
   public boolean svaneMerket(){return false;}
}
```

rollen "Skattbar"

rollen "Miljøvennlig"

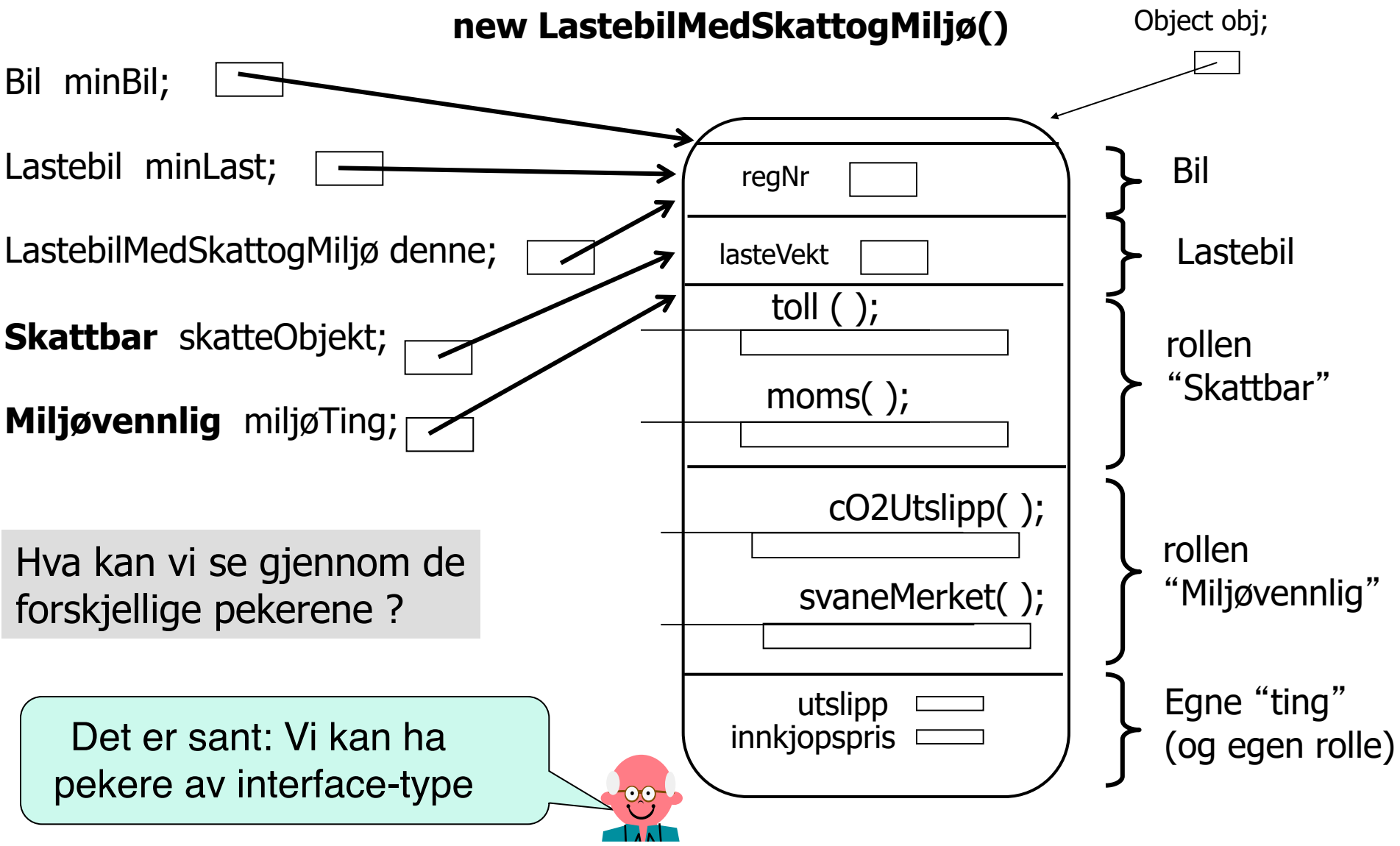
```
interface Skattbar {
    double toll();
    double moms();
}
```

```
interface Miljøvennlig {
   int co2Utslipp();
   boolean svaneMerket();
}
```

```
class Bil {
    final String regNr;
}
```

```
class Lastebil {
    protected  double lasteVekt;
}
```

Et objekt og noen pekere



Hva kan vi se gjennom de forskjellige pekerene ?

Det er sant: Vi kan ha pekere av interface-type

Enda mer om grensesnitt (interface)

- Navnet på et interface kan brukes som typenavn når vi lager referanser (det så vi på forrige side)
- Vitsen med et interface er å spesifisere **hva** som skal gjøres (ikke hvordan)
- Vanligvis er det flere implementasjoner av et interface (flere klasser implementerer det).
- Vi vet: En klasse kan implementere (flere) interface samtidig som klassen også er subklasse av (bare) én annen klasse.
- En implementasjon (av et interface) skal kunne endres uten at resten av programmet behøver å endres.

- Vi har en **ny type**: interfacenavn:
  «Skattbar» og «Miljøvennlig» er referansetyper

Interface lærdom

- Et interface har bare
  - metodenavn med parametre, men ikke kode (husk ;)
- Definerer en ’type’ / ’rolle’ som andre må implementere
- Meget nyttig, brukes mye ved distribuerte systemer og generelle programbiblioteker som Javas eget
- Ulempe: Koden/implementasjonen må gjøres mange ganger

- **I informatikk, mer generelt kjent under navnet ADT** = Abstrakt Data Type
  - Vi definerer ***hva*** en ny datatype skal gjøre, ***ikke hvordan*** dette gjøres (derfor «Abstrakt»)
  - Det kan være mange mulige implementasjoner (=måter å skrive kode på) som lager en slik datatype.
  - Hva som er beste implementasjon må avgjøres etter hvilken bruk vi har.

Ekstra eksempler:
Mer om Biler og Lastebiler:
Legg til metoder for å skrive ut på skjerm:

```
class Bil {
    protected String regNr;
    public void skriv(){
        System.out.println(“Registreringsnummer: " + regNr);
    }
}
```

```
class Lastebil extends Bil {
    double lasteVekt;
    @Override
    public void skriv (){
        super.skriv();
        System.out.println("Lastevekt: " + lasteVekt);
    }
}
```

Skriv i LastebilMedSkattOgMiljo


```
class LastebilMedSkattOgMiljo extends Lastebil implements Skattbar, Miljøvennlig {
    protected double innkjopspris = 200000;
    protected int utslipp = 400;
    @Override
    public double  toll( ) { return innkjopspris * 0.1; }
    @Override
    public double moms( ) {return 1.25;}
    public void skrivSkatt( ) {
         System.out.println("Innkjøpspris " + innkjopspris);
    }
    @Override
    public int cO2Utslipp ( ) {return utslipp; }
    @Override
    public boolean svaneMerket () { return false; }
    public void skrivMiljo( ) {
         System.out.println("Utslipp " + utslipp);
    }
    @Override
    public void  skriv( ) {
          System.out.println("Lastebil med skatt og miljø: ");
          super.skriv( );   skrivSkatt();   skrivMiljo();
    }
}
```

rollen "Skattbar"

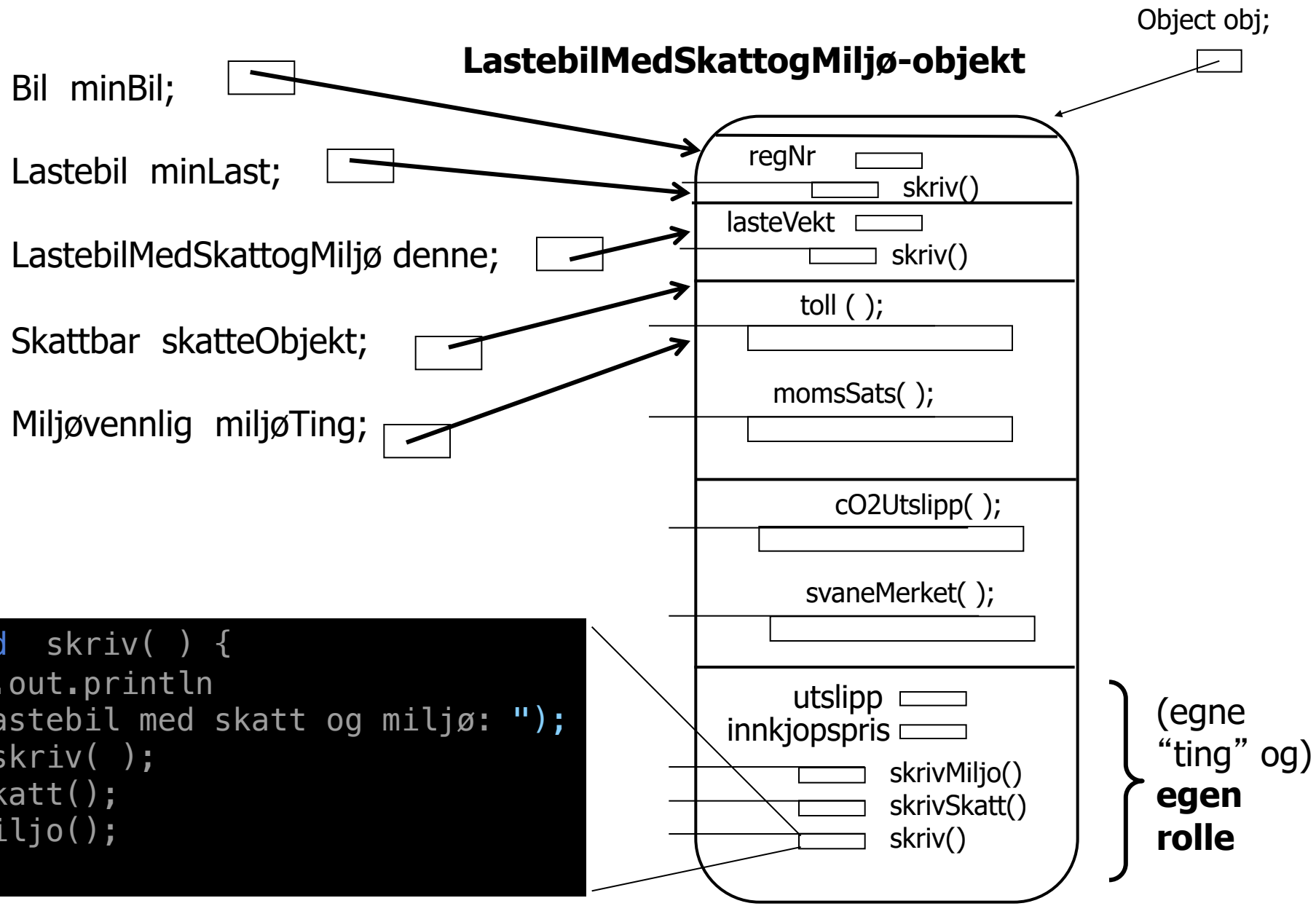
rollen "Miljøvennlig"

Ny "Skriv"-rolle som del av LastebilMed. . .

(Skattbar og Miljøvennlig som før)

Det er ikke naturlig at Skatt og Miljo skal **kreve** en "skriv"-metode (?)

Skriv i LastebilMedSkattogMiljø



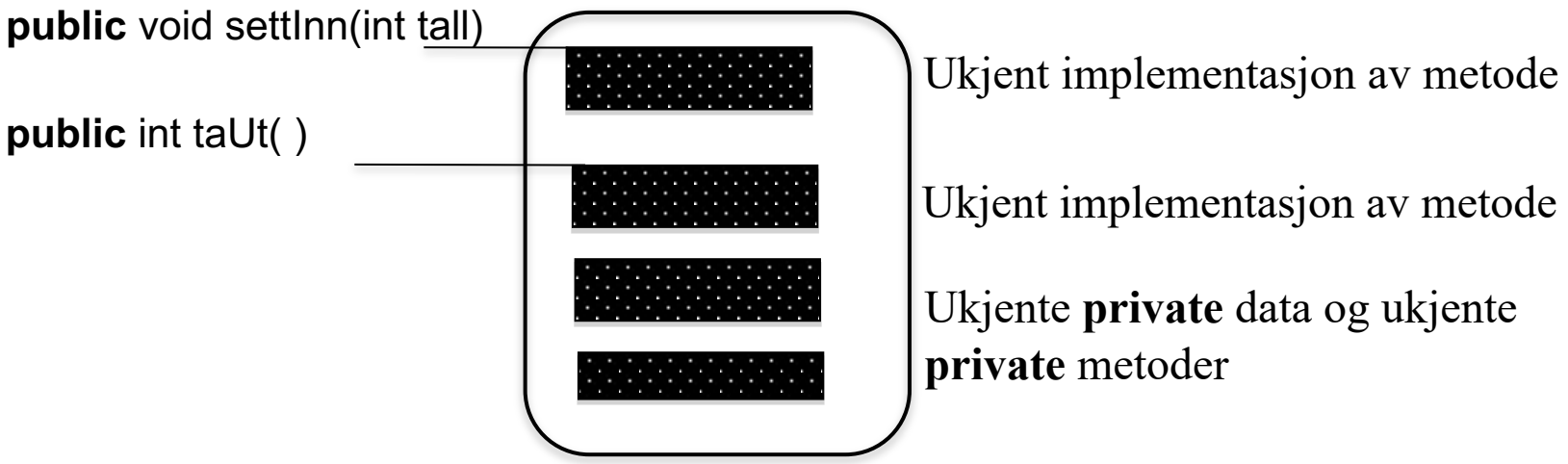

```
public void  skriv( ) {
    System.out.println
       ("Lastebil med skatt og miljø: ");
    super.skriv( );
    skrivSkatt();
    skrivMiljo();
}
```

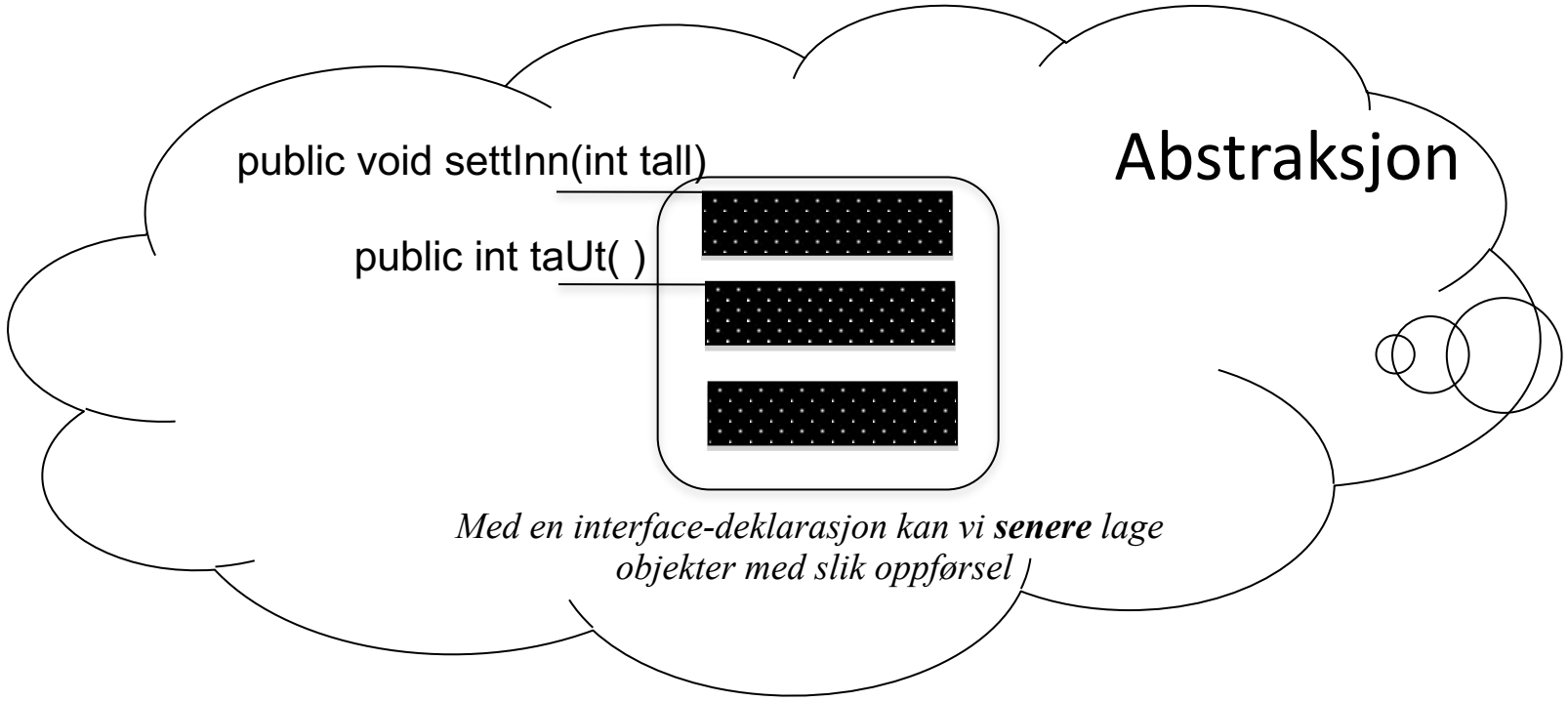
(Delvis) ny bruke av interface

Interface som spesifikasjon av grensesnitt

Objektorientering handler om å tydeliggjøre objektenes public-metoder – abstraksjon

Husker dere forelesingen om enhetstesting:

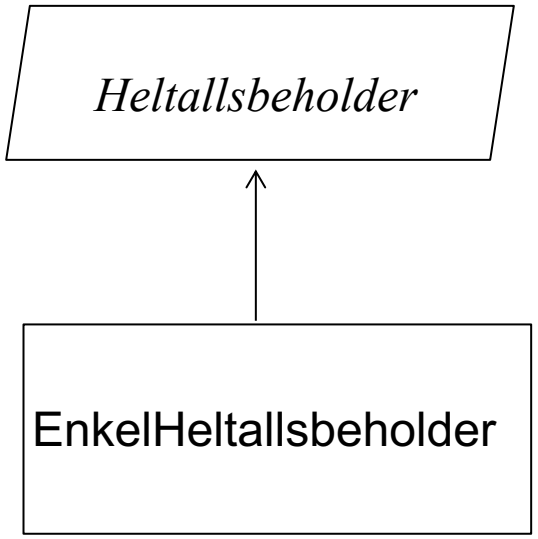





```
interface Heltallsbeholder {
    void settInn(int tall);
    int taUt( );
}
```

~~new Heltallsbeholder()~~

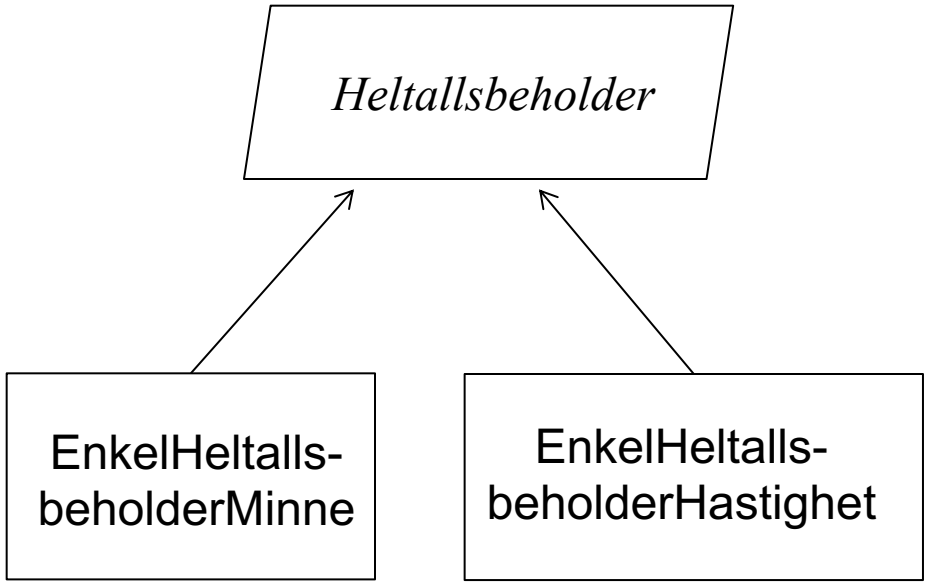
Interface: Klassehierarki og Java-kode




```
interface Heltallsbeholder {
   void settInn(int tall);
   int taUt( );
}
```

```
class EnkelHeltallsbeholder implements Heltallsbeholder {
    protected int [ ] tallene = new int [100];
    protected int antall;
    @Override
    public void settInn(int tall) { . . . }
    @Override
    public int taUt( ) { . . . }
}
```

VIKTIG: Ett interface
Flere forskjellige implementasjoner




```
interface Heltallsbeholder {
   void settInn(int tall);
   int taUt( );
}
```

```
class EnkelHeltallsbeholderMinne
                implements Heltallsbeholder {
   protected int [ ] tallene = new int [100];
   protected int antall;
   public void settInn(int tall) { . . . }
   public int taUt( ) { . . . }
}
```

”Ikke-funksjonelle” forskjeller på implementasjonene. For eksempel hastighet, minnebruk, . . .



```
class EnkelHeltallsbeholderHastighet
            implements Heltallsbeholder {
   protected ArrayList . . . . .
   public void settInn(int tall) { . . }
   public int taUt( ) { . . . }
}
```

VIKTIG

Kan da lett bytte ut implementasjonen:

```
class BeregnEttEllerAnnet {
    . . .
    Heltallsbeholder hBeholder = new EnkelHeltallsbeholderHastighet( );
     . . .
     . . .
     . . .
    hBeholder.settInn(7);
    int x = hBeholder.taUt();
     . . .
    hBeholder.settInn(13);
    hBeholder.settInn(7102);
    hBeholder.settInn(14);
     . . .
    int y = hBeholder.taUt();
     . . .
}
```

Kan byttes ut med

```
new EnkelHeltallsbeholderMinne( );
```

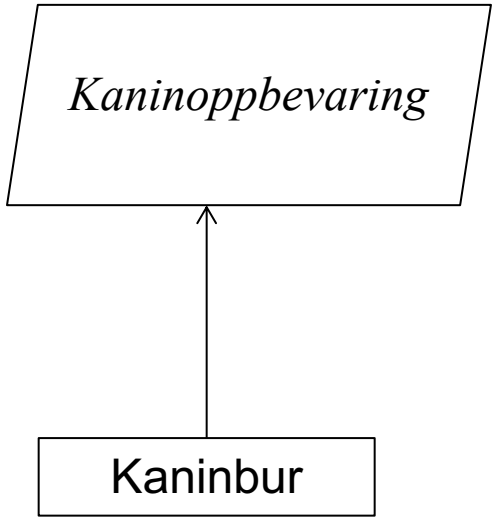
Resten av programmet er uforandret

Interface for å tydeliggjøre ”public-metodene”:

Vi kan se på et kaninbur som et sted for kaninoppbevaring



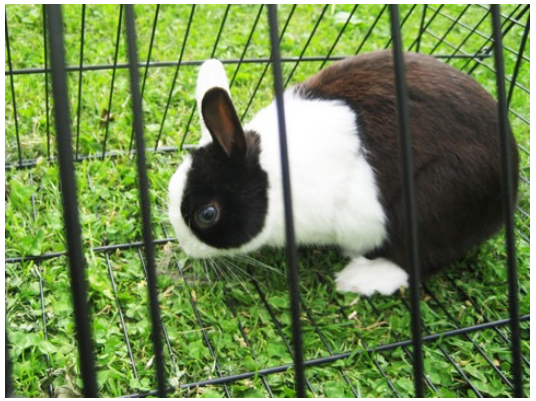
```
interface KaninOppbevaring {
    public boolean settInn(Kanin k);
    public Kanin taUt( );
}
```



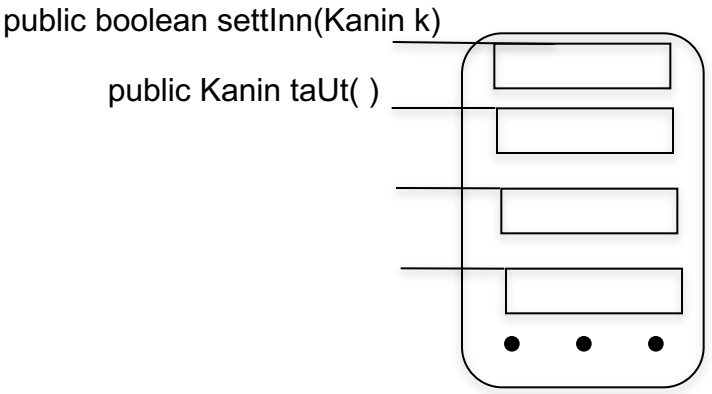


Kaninbur som et sted for kaninoppbevaring


```
interface KaninOppbevaring {
    public boolean settInn(Kanin k);
    public Kanin taUt( );
}
```



```
class Kaninbur implements KaninOppbevaring {
    private Kanin denne = null;
    @Override
    public boolean settInn(Kanin k) {
        . . .
    }
    @Override
    public Kanin taUt( ) {
       . . .
    }
}
```





Et objekt av en klasse som implementerer grensesnittet KaninOppbevaring

Full kode


```
class Kanin {
    private  String navn;
    public Kanin(String nv) {navn = nv;}
}
```



```
interface KaninOppbevaring {
    public boolean settInn(Kanin k);
    public Kanin taUt( );
}
```

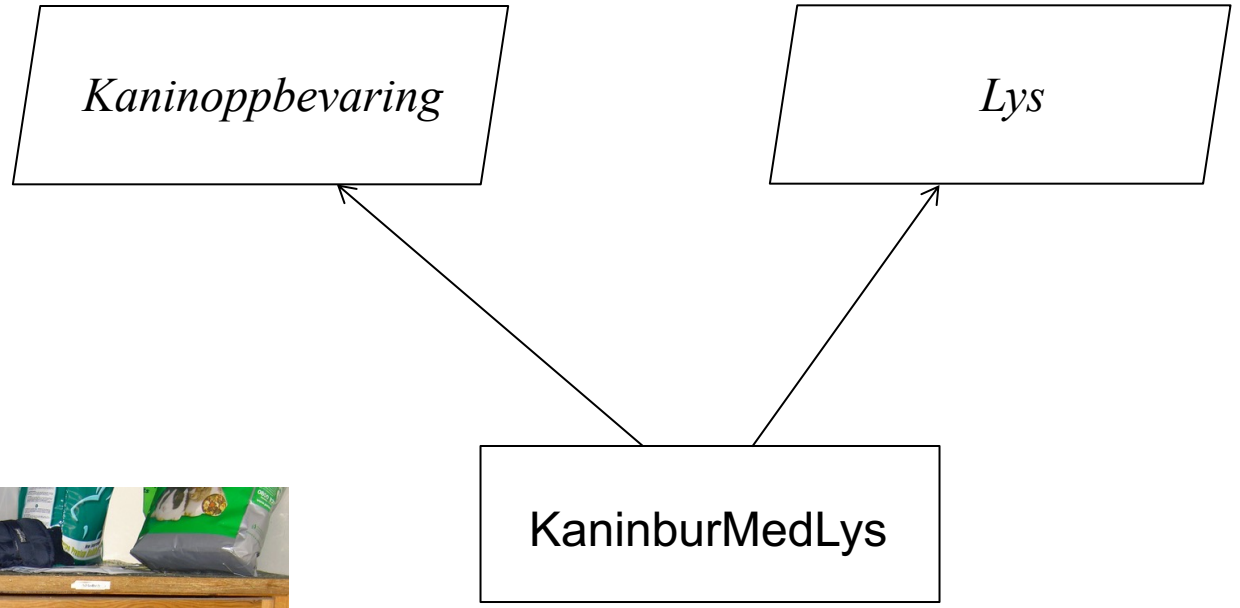


```
class Kaninbur implements KaninOppbevaring {
    private Kanin denne = null;
    @Override
    public boolean settInn(Kanin k) {
        if (denne == null)  {
            denne = k;
            return true;
        }
        else return false;
    }
    @Override
    public Kanin taUt( ) {
        Kanin k = denne;
        denne = null;
        return k;
    }
}
```

KaninburMedLys



Når en kanin vil ha lys på om natten:



Lys: Et sted/rom hvor det er lys som kan slås på og av

```
interface Lys {
    public void tennLyset ( );
    public void slukkLyset ( );
}
```



Vi kan lage kassen KaninburMedLys på denne måten: Én klasse – to grensesnitt

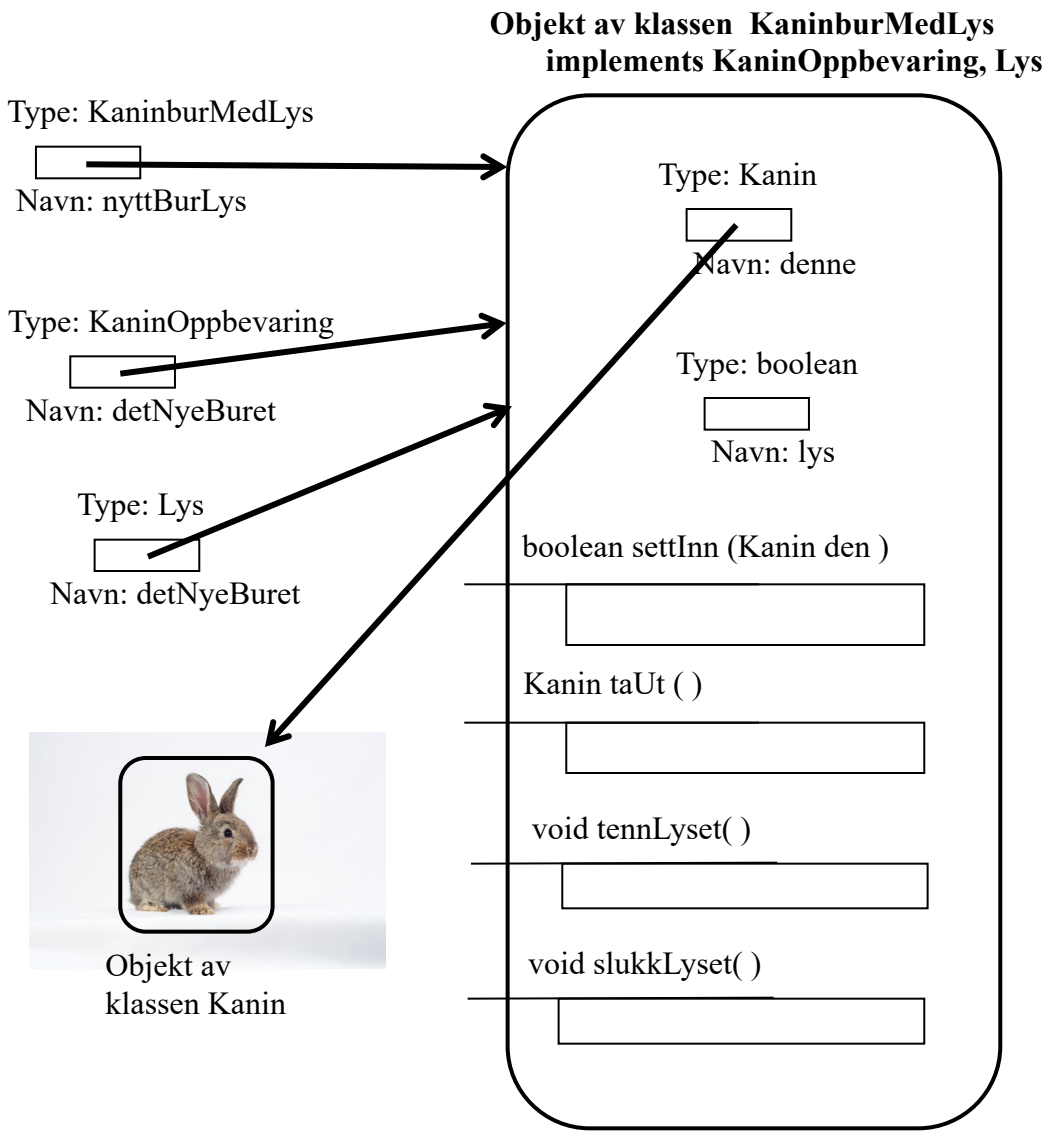

```
class KaninburMedLys implements
                      KaninOppbevaring, Lys {
    private boolean lys = false;
    private Kanin denne = null;
    @Override
    public boolean settInn(Kanin k) {
       . . .
    }
    @Override
    public Kanin taUt( ) {
       . . .
    }
    @Override
    public void tennLyset ( )  {lys = true;}
    @Override
    public void slukkLyset ( ) {lys = false;}
}
```

```
interface KaninOppbevaring {
    public boolean settInn(Kanin k);
    public Kanin taUt( );
}
```

```
interface Lys {
    public void tennLyset ( );
    public void slukkLyset ( );
}
```



Ett objekt– to grensesnitt – tre briller




```
interface KaninOppbevaring {
   public boolean settInn(Kanin k);
   public Kanin taUt( );
}
```

```
interface Lys {
     public void tennLyset ( );
     public void slukkLyset ( );
}
```



Forskjellige briller = forskjellige roller

Én klasse – to grensesnitt: Full kode


```
class KaninburMedLys implements
                    KaninOppbevaring, Lys {
    private boolean lys = false;
    private Kanin denne = null;
    @Override
    public boolean settInn(Kanin k) {
        if (denne == null)  {
            denne = k;
            return true;
        }
        else {return false;}
    }
    @Override
    public Kanin taUt( ) {
        Kanin k = denne;
        denne = null;
        return k;
    }
    @Override
    public void tennLyset() {lys = true;}
    @Override
    public void slukkLyset(){lys = false;}
}
```

```
interface KaninOppbevaring {
    public boolean settInn(Kanin k);
    public Kanin taUt( );
}
```

```
interface Lys {
    public void tennLyset ( );
    public void slukkLyset ( );
}
```

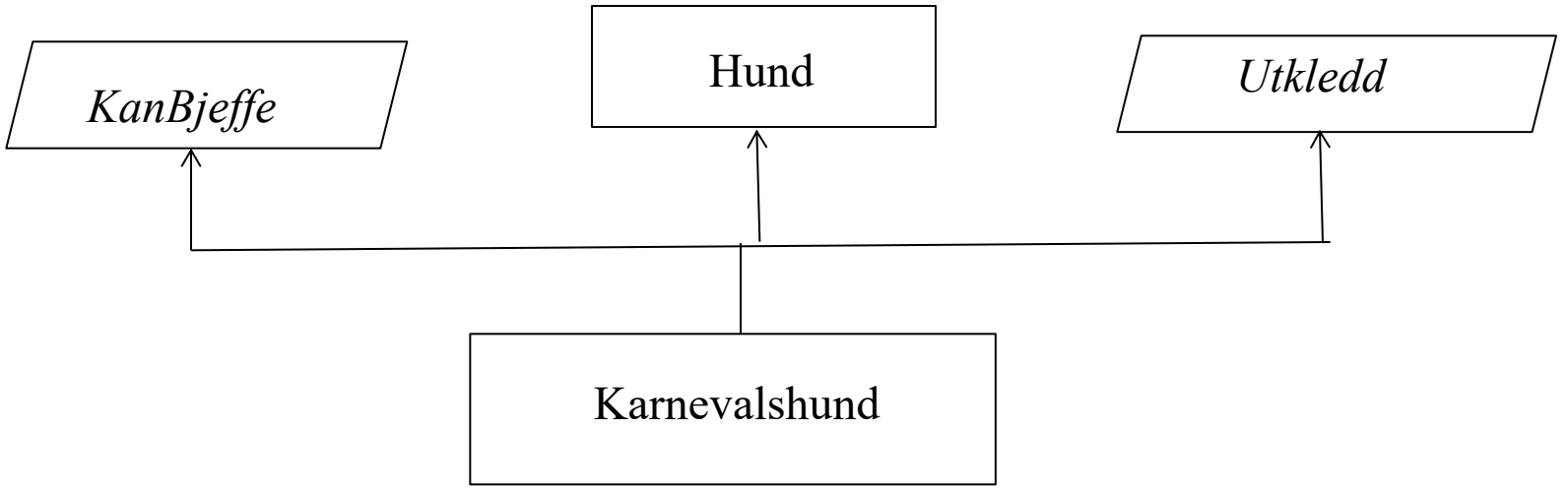


Karnevalshund



Foto: AP

To (eller flere) grensesnitt
= to (eller flere) roller



Flere eksempler: En klasse – mange grensesnitt


```
class Hund {protected double vekt;}
```

```
interface KanBjeffe{
      void bjeff();
}
```

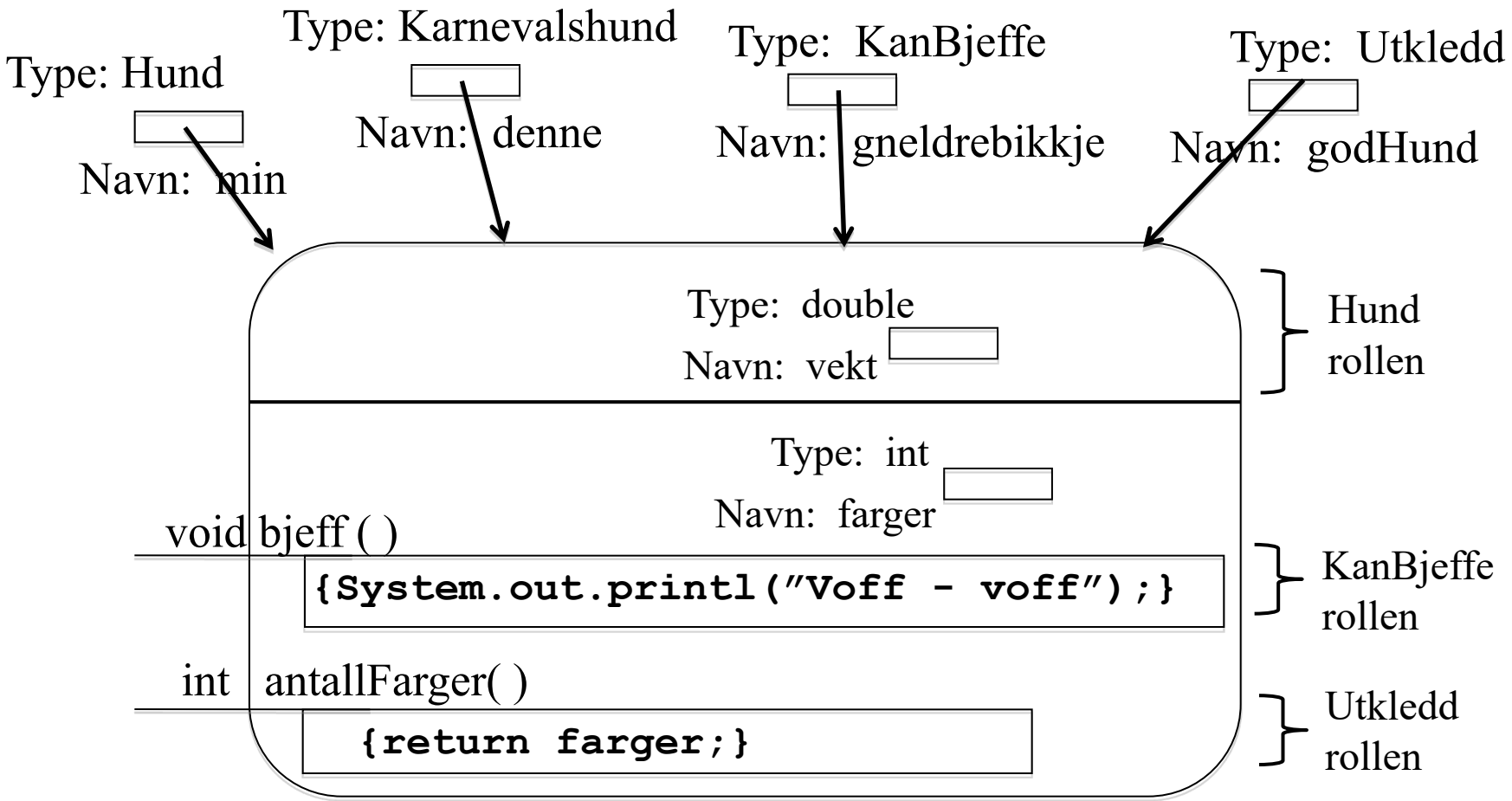
```
interface Utkledd {
      int antallFarger();
}
```

Foto: AP



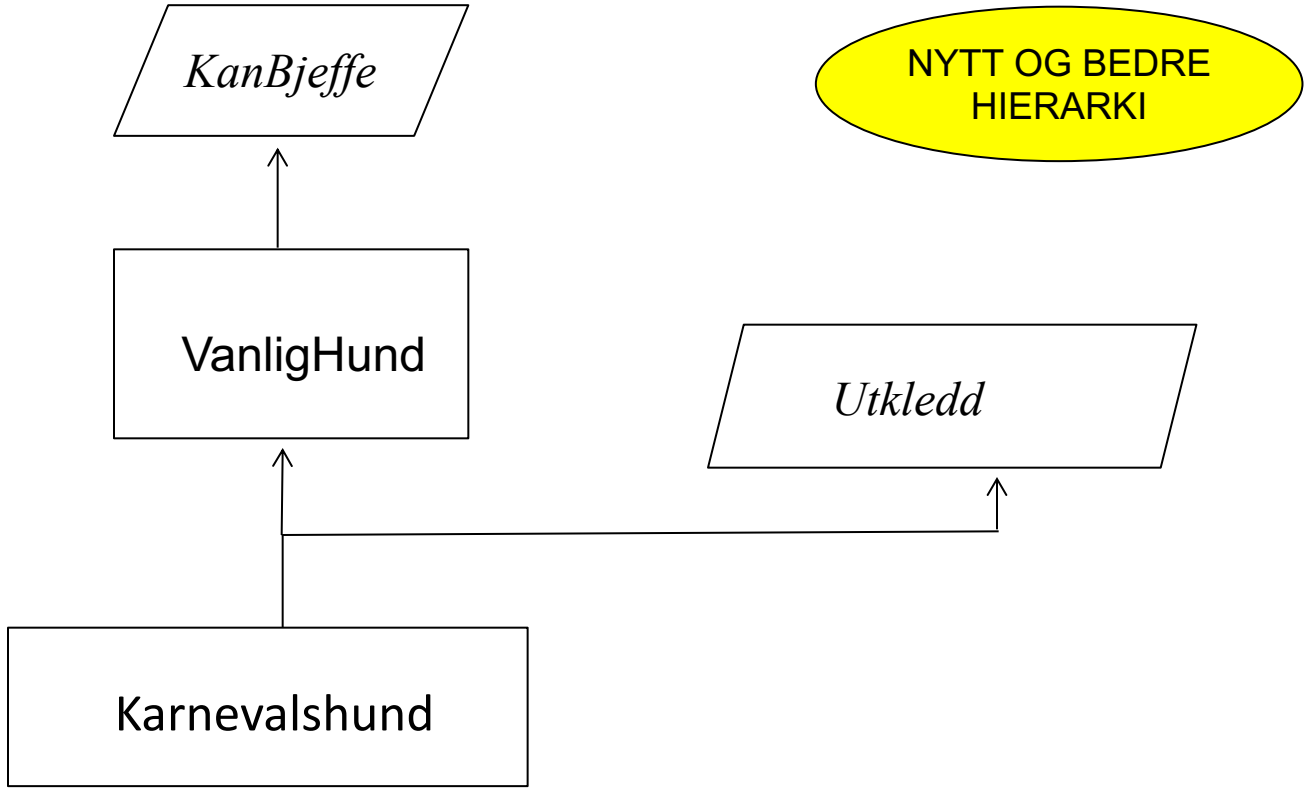
```
class Karnevalshund extends Hund implements KanBjeffe, Utkledd {
    protected int farger;
    public Karnevalshund (int frg) { farger = frg; }
    @Override
    public void bjeff( ) {
        System.out.printl("Voff – voff");
    }
    @Override
    public int antallFarger() {
        return farger;
    }
}
```

```
Karnevalshund   passopp  = new Karnevalshund( ):
Hund            min      = passopp;
KanBjeffe       gneldrebikkje = passopp;
Utkledd         godHunden = passopp;
```



Objekt av klassen Karnevalshund

Eller kanskje bedre:



Denne figuren avspeiler
”interface”-ene og ”class”-ene på neste siden

```
interface KanBjeffe{
      void bjeff();
}
```

```
interface Utkledd {
      int antallFarger();
}
```

```
class VanligHund implements KanBjeffe {
    @Override
    public void bjeff() {
        System.out.println("Vov-vov");
    }
}
```



Foto: AP

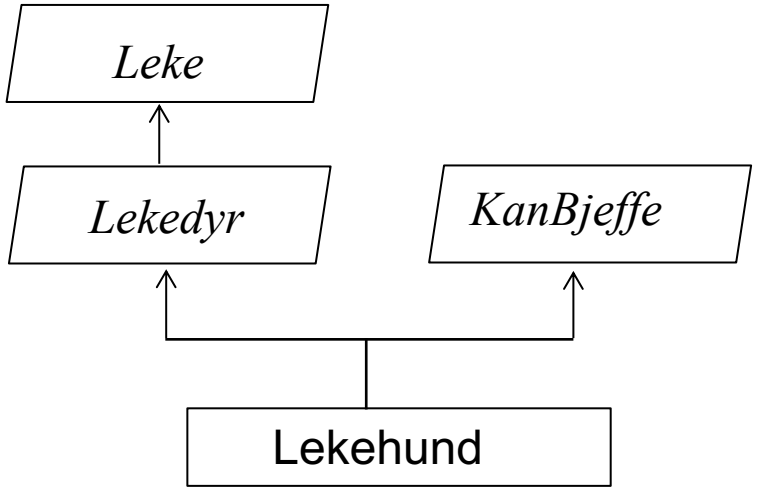
```
class Karnevalshund extends VanligHund implements Utkledd {
    protected int farger;
    public Karnevallshund (int frg) {
        farger = frg;
    }
    @Override
    public int antallFarger() {
        return farger;
    }
}
```

```
interface Leke {
    String type();
}
```

```
interface KanBjeffe{
    void bjeff();
}
```


NYTT

```
interface Lekedyr extends Leke {
    int hoyde();
    boolean mykPels();
}
```




```
class Lekehund implements Lekedyr, KanBjeffe {
    int hoyde;  boolean myk;
    Lekehund(int h, boolean myk) {
        hoyde = h;  this.myk = myk;
    }
    @Override
    public String type() { return "Hund";}
    @Override
    public  int hoyde () {return hoyde;}
    @Override
    public boolean mykPels () {return myk;}
    @Override
    public void bjeff() {System.out.println("Vov-vov");}
}
```



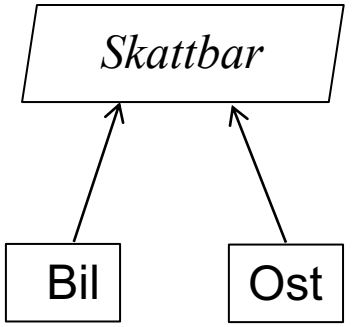
BATTERIES INCLUDED

Tilbake til skatteetaten: Både biler og ost skal skattlegges
Ett grensesnitt, flere klasser.


```
interface Skattbar{                  // Skatt på importerte varer
    int skatt();
}

class Bil implements Skattbar {  // Bil: 100% skatt
    protected . . .
    protected int importpris;
    public Bil ( . . . ) { . . . }
    @Override
    public int skatt( ){return importpris;}
    . . .
}

class Ost  implements Skattbar {  // Ost: 200% skatt
    protected int importprisPrKg;
    protected  . . .
    public Ost (. . . ) { . . . }
    @Override
    public int skatt( ){return . . .  * 2;}
}
```



Dette eksemplet har store likheter med Skatteetaten-eksemplet forrige uke

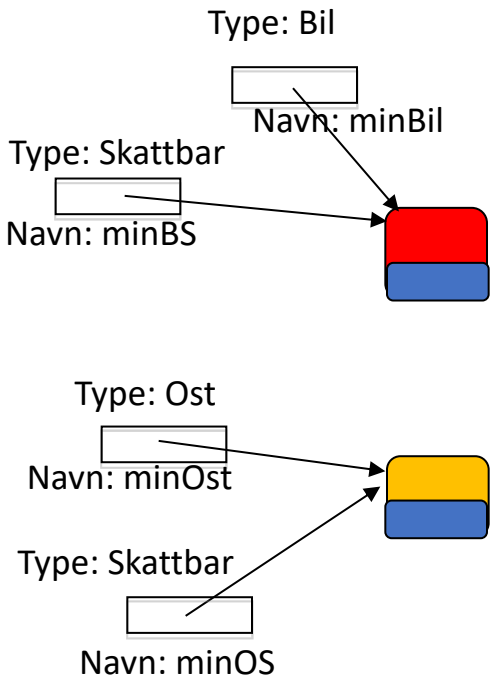
Hva er likt?

Hva er forskjellig?

Legg merke til at metoden skatt er implementert på forskjellige måter i Bil og Ost.


```
Bil minBil = new Bil ("BP12345", 100000);
Skattbar minBS = minBil;
Ost minOst = new Ost(100, 2);
Skattbar minOS = minOst;

int totalSkatt = 0;
totalSkatt = totalSkatt + minBS.skatt();
totalSkatt = totalSkatt + minOS.skatt();
```



Rollen Skatt

Rollen Bil (untatt Skatt)

Rollen Ost (untatt Skatt)

Generalisering på neste side

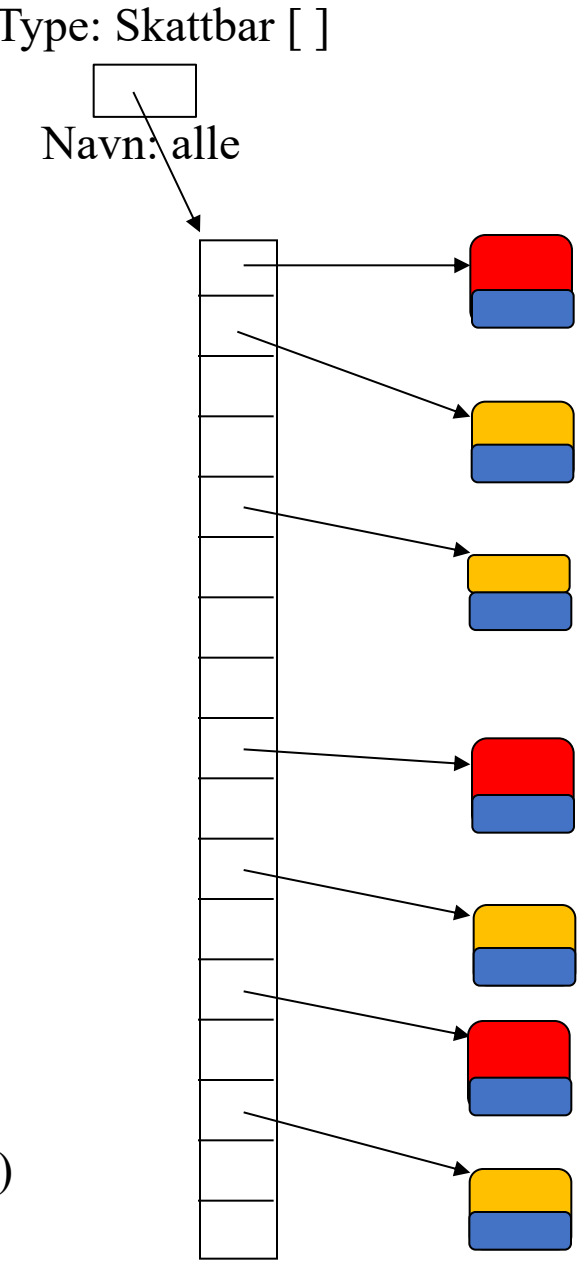
VIKTIG EKSEMPEL

Samlet import-skatt


```
Skattbar[ ] alle = new Skattbar [100];
alle[0] = new Bil(”DK12345”, 150000);
alle[1] = new Ost(20,5000);
. . .
. . .
int totalSkatt = 0;

for (Skattbar den: alle) {
    if (den != null)
        totalSkatt = totalSkatt + den.skatt();
}

System.out.println(”Total skatt: ” + totalSkatt);
```



Rollen Skatt

Rollen Bil (untatt Skatt)

Rollen Ost (untatt Skatt)

Klassene Bil og Ost – Full kode


```
interface Skattbar{                  // Skatt på importerte varer
    int skatt();
}

class Bil implements Skattbar {  // Bil: 100% skatt
    final public String regNr;
    protected int importpris;
    public Bil (String reg, int imppris) {
        regNr = reg; importpris = imppris;
    }
    @Override
    public int skatt( ){return importpris;}
}

class Ost implements Skattbar {  // Ost: 200% skatt
    protected int importprisPrKg;
    protected int antKg;
    public Ost (int kgPris, int mengde) {
        importprisPrKg = antKg; antKg = mengde;
    }
    @Override
    public int skatt( ){return importprisPrKg*antKg*2;}
}
```

Litt om invarianter og om å lage gode programmer

Mer den 10. mai.

Først:

Norsk: Antagelser / Betingelser
Engelsk: Assertions / Conditions

- En beskrivelse av tilstanden på et bestemt sted i programmet kalles en antagelse eller en betingelse, f.eks.:

```
int tall = 0;
while (tall < 10) {
    tall++;
}
// Nå er !(tall < 10)
// dvs.   tall >= 10
```

tall >= 10 kalles en antagelse eller en betingelse

Et godt program inneholder kommentarer med gyldige antagelser om tilstanden til variablene i programmet på dette stedet i koden

WIKIPEDIA:
Invariant (computer science)

- In computer science, an **invariant** is a condition that can be relied upon to be true during execution of a program, or during some portion of it. It is a logical assertion that is held to always be true during a certain phase of execution. For example, a **loop invariant** is a condition that is true at the beginning and end of every execution of a loop.

  . . . . .
- Programmers often use assertions in their code to make invariants explicit. Some object oriented programming languages have a special syntax for specifying **class invariants**.

"Loop invariants" skal vi ikke snakke om nå men den 10. mai.

Invarianter i objekter

- Ofte kalt (noe unøyaktig) “class invariants”
- En invariant på datastrukturen i et objekt hjelper oss å programmere (public) metodene bedre og mer oversiktlig
- En invariant på datastrukturen i et objekt sier oss noe om instansvariablenes verdier, forholdet mellom verdiene til instansvariablene og generelt om datastrukturen inne i objektet
- En invariant på datastrukturen i et objekt forklarer oppgaven til instansvariablene og begrenser verdiene de kan ha
  - Litt slik som typen til en variabel gjør

Som vanlig: Et veldig enkelt eksempel:

Kaningård

Navn: antall

3

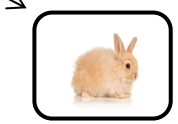
Type: int

Kanin [ ] alle

0
1
2
3

MAXANT - 1







```
public void settInn (Kanin en)
{
    alle [antall] = en;
    antall ++;
}
```

```
public Kanin  taUt( )
{
    antall --;
    return alle [antall];
}
```

```
public  Kaningard ( )
{
    antall = 0;
}
```



Ikke særlig robust program

La oss se på kode:

- BrukKanin.java

Invarianten holder i det objektet opprettes OG alle metodene bevarer innvarianten -> Da holder invarianten alltid

public void settInn (Kanin en)

Pre-condition: Invarianten gjelder

```
if (antall < MAXANT) {
        alle [antall] = en;
        antall ++;
}
```

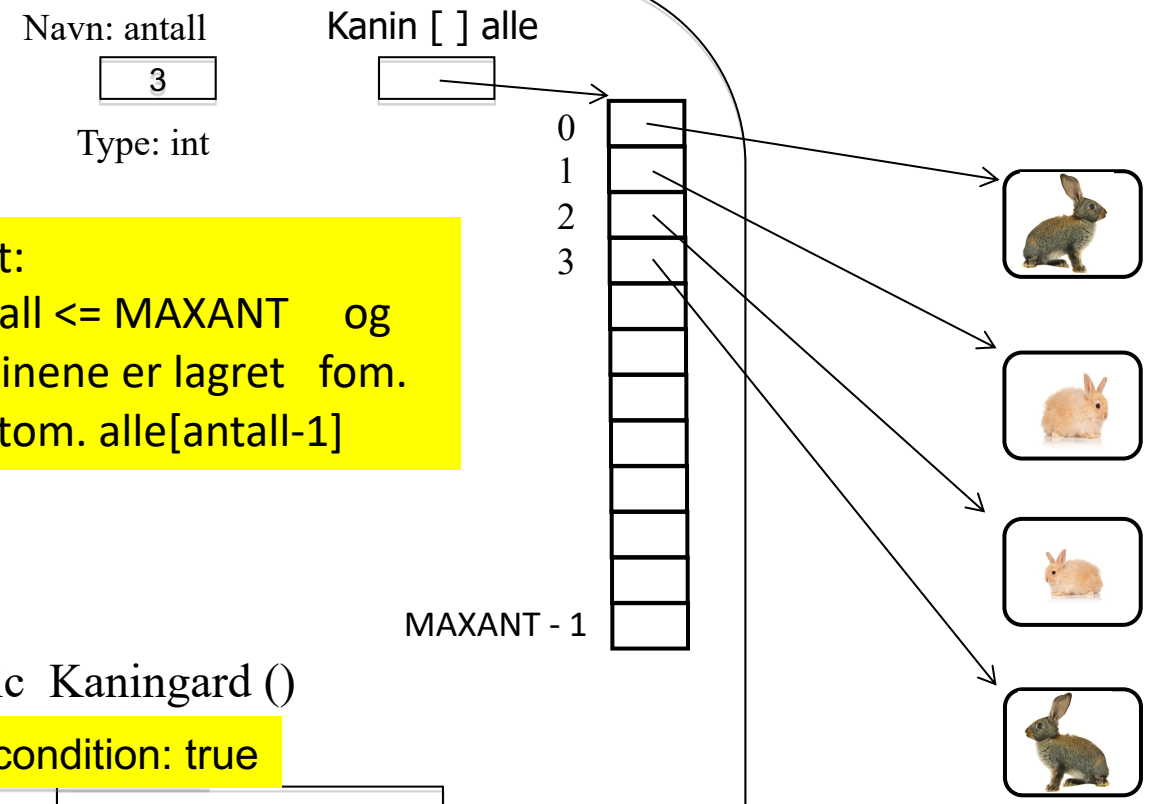
Post-condition: Invarianten gjelder

public Kanin taUt( )

Pre-condition: Invarianten gjelder

```
if (antall > 0)  {
        antall --;
        return alle [antall];
} else return null;
```

Post-condition: Invarianten gjelder



Invariant:
0 <= antall <= MAXANT og
alle kaninene er lagret fom.
alle[0] tom. alle[antall-1]

public Kaningard ()

Pre-condition: true

```
antall = 0;
```

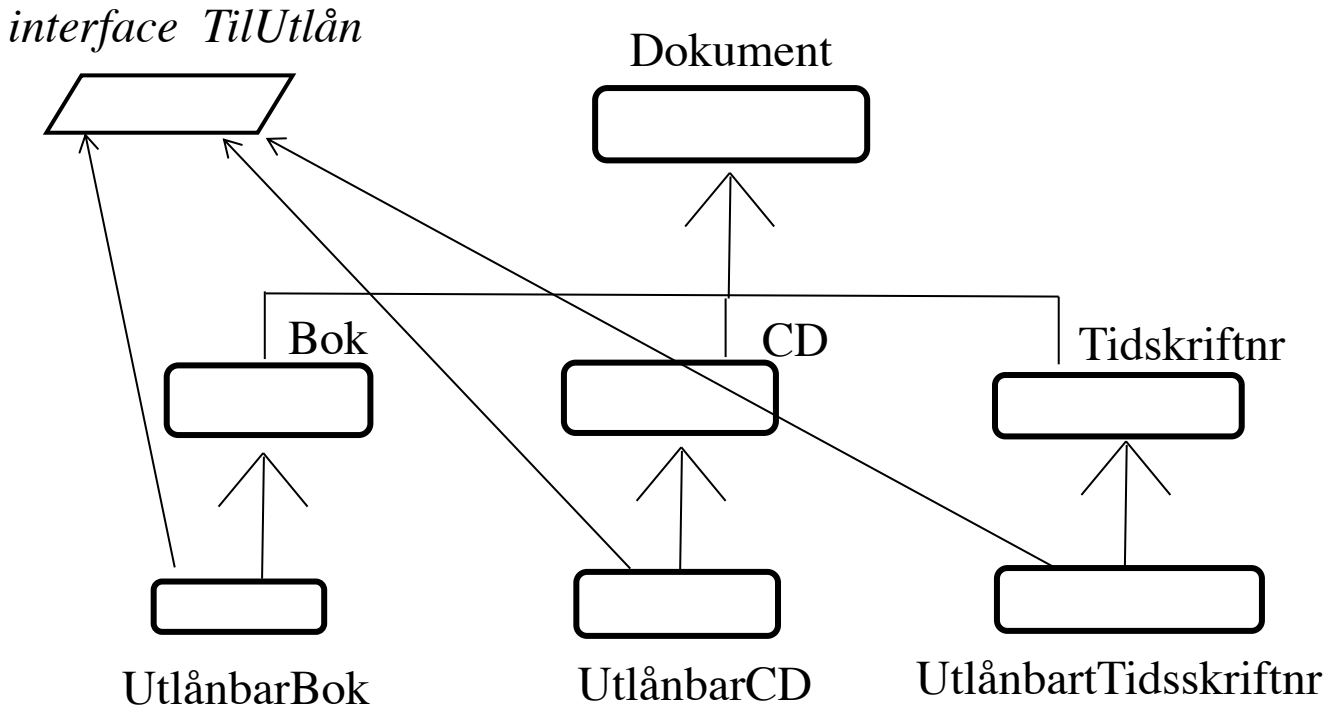
Post-condition: Invarianten gjelder

(Hva gjør denne kaninen her ?)

Hoved "take-away" i dag

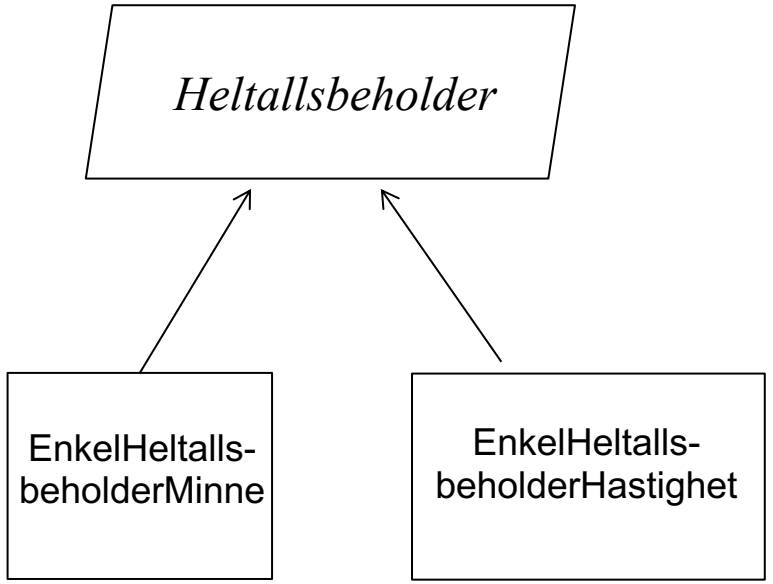
- To hoved-grunner til å bruke interface:

Multippel arv og samme oppførsel på tvers av klasser



Forskjellig / ukjent implementasjon men samme oppførsel

```
Heltallsbeholder hBeholder = new . . .
```



+ Både Biler og Oster er *Skattbare*

Oppsummering om interface

- Java har en mekaniske "interface" som
- Tydeliggjør og definerer en (implementerendes) klasses grensesnitt
  - Abstraksjon / Innkapsling / Skjuling av detaljer
- Kan brukes til multippel arv av oppførsel
  - Men alle metodene må implementeres på nytt
  - Så Java har ikke multippel arv av kode
- Er en "rolle" på linje med klasser og subklasser
- Kan brukes som en referansetype
- Alle egenskapene til et interface er **metoder** (metodesignaturer)
  - ; istedenfor { . . . } bak signaturen / overskriften til metoden

Eksempel på toString og equals

```
class Punkt1 {
   protected int x, y;
   Punkt1(int x0, int y0) {
      x = x0;
      y = y0;
   }
}
```

```
class Punkt2 {
   protected int x, y;
   Punkt2(int x0, int y0) {
      x = x0;  y = y0;
   }
   @Override
   public String toString() {
      return ("x = "+x+"   y = "+y);
   }
   @Override
   public boolean equals(Object o) {
       if (!(o instanceof Punkt2))
          return false;
       Punkt2 p = (Punkt2) o;
       return x == p.x && y == p.y;
   }
}
```

```
Punkt p1 = new Punkt1(3,4);
Punkt p2 = new Punkt1(3,4);

p1.toString();  Punkt1@4d591d15

p1.equals(p2);  false
```

```
Punkt2 q1 = new Punkt2(3,4);
Punkt2 q2 = new Punkt2(3,4);

q1.toString();    x = 3   y = 4

q1.equals(q2);    true
```

(4D591D15 er hash-koden til objektet i det heksadesimale (16-)tall-systemet = 1297685781 i 10-tall-systemet)