IN1010 våren 2021

Tirsdag 26. januar

Arv og subklasser – del 1

Stein Gjessing

Når du har lært om subklasser kan du programmere med:

Første uke (i dag):

- Spesialisering (og generalisering)
- Klasse-hierarkier - arv
- Referanser (pekere) – sterk typing
- Nøkkelordet instanceof
- Konvertering av referanser
  - Klassen Object
- Abstrakte klasser

Andre uke (tirsdag 2. februar)

- Virtuelle metoder - polymorfi
  - Nøkkelordet super
- Gjenbruk av klasser og begreper
  - Ved sammensetning (komposisjon)
  - Ved arv
- Konstruktører

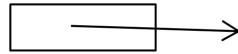
Tredje uke (tirsdag 9. februar)

- Interface

Først litt repetisjon

- En variabel i Java har et navn, en type og et innhold

Navn: dinTeller



Type: Counter

Navn: minTeller

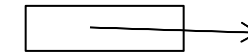
Type: Counter

Navn: sum

Type: int

Alle disse 4 variablene er på 32 bit (i en 32 bit arkitektur):

Navn: alleBiler



Type: Bil [ ]

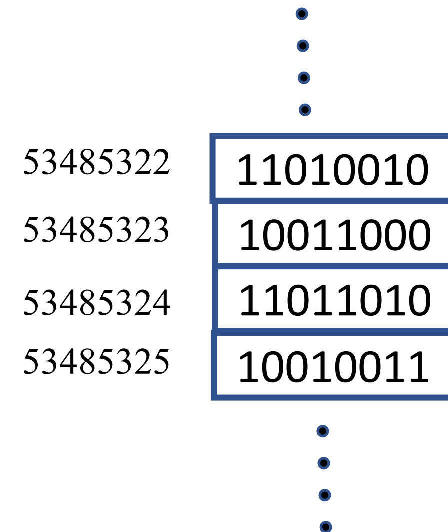
Scop-regler - synlighetsregler (engelsk scope)

En «static»-variabel lever så lenge programmet lever
En instansvariabel lever så lenge objektet lever
En variabel deklarert i en metode (inkl. parametrene) lever så lenge metodeinstansen lever
En variabel deklarert i en blokk lever så lenge programmet eksekverer inne i blokken

Variabler, verdier og typer

- 8 **primitive typer** med verdier som bruker plass i minnet:
  - boolean kan greie seg med en bit (0 for false og 1 for true)
    men kompilatoren velger å bruke én byte
  - **byte** 10011010
  - char: 2 byte
  - short: 2 byte
  - **int: 32 bit** 10010011 11011010 10011000 11010010
  - long: 64 bit
  - float: 32 bit
  - double: 64 bit

i RAM:



- **Referanse-typer er klassenavn*** (objekt-referanser): Så mange bit som det er i datamaskinens adresserom, 32 (eller 64) bit
  - Referanser til arrayer er typet med typen til innholdet + at det er en array f.eks. int [ ]

Uttrykk regnes ut til en verdi av en av disse typene.
Uttrykk forkommer i høyresider og i aktuelle parametre.

* Senere skal vi se at det også er interface-navn

Variabler / objekter

Navn: minTeller

Type: Counter

Navn: hansTeller

Type: Counter

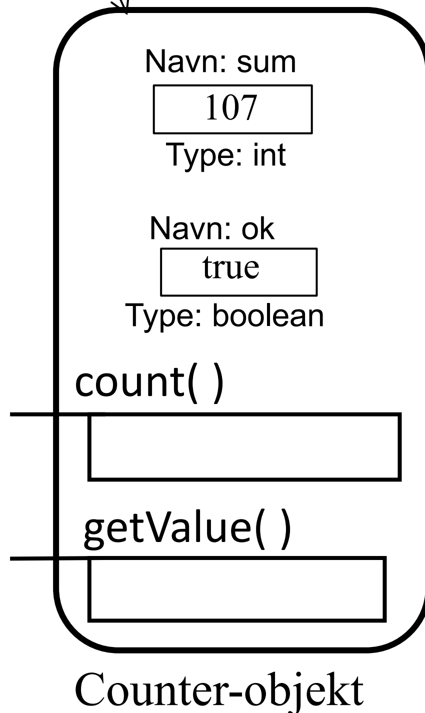
Navn: dinTeller

Type: Counter

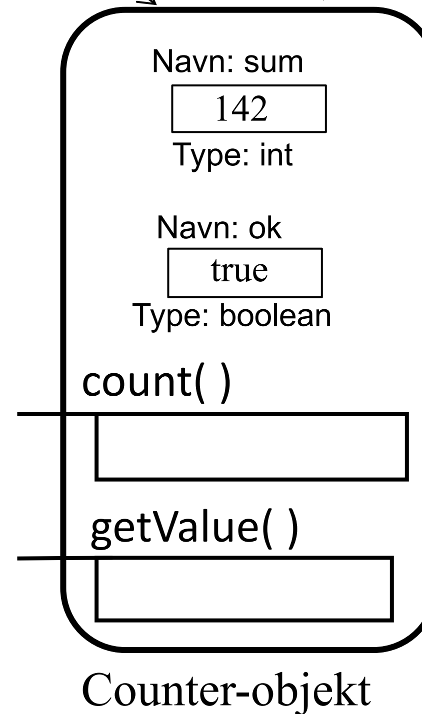
Navn: sum

249

Type: int



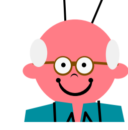
Counter-objekt



Counter-objekt

Når skal vi tenke og tegne datastrukturene i programmet ?

Og hvor nøyaktig skal vi da tegne?



UNIVERSITETET I OSLO

Institutt for informatikk

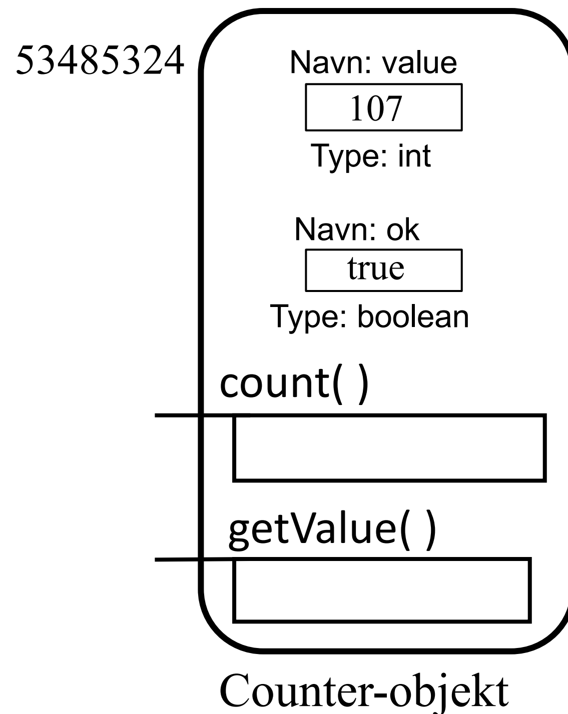
Referanser

Navn: minTeller
53485324
Type: Counter

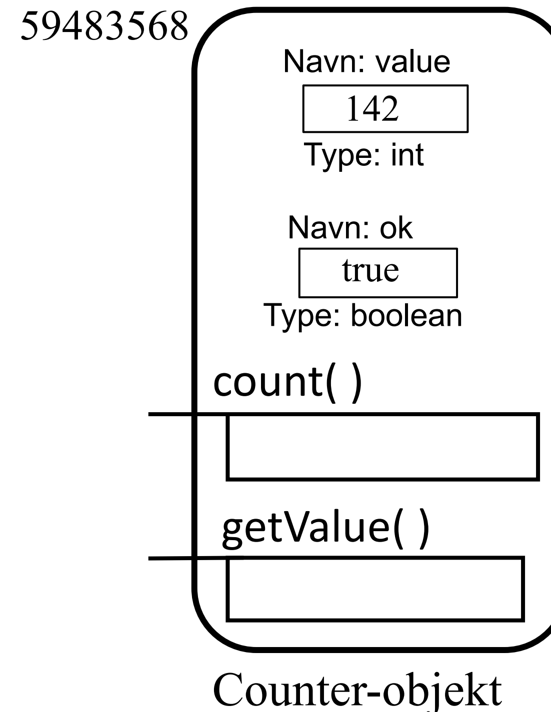
Navn: hansTeller
59483568
Type: Counter

Navn: dinTeller
59483568
Type: Counter

Navn: sum
249
Type: int



Counter-objekt

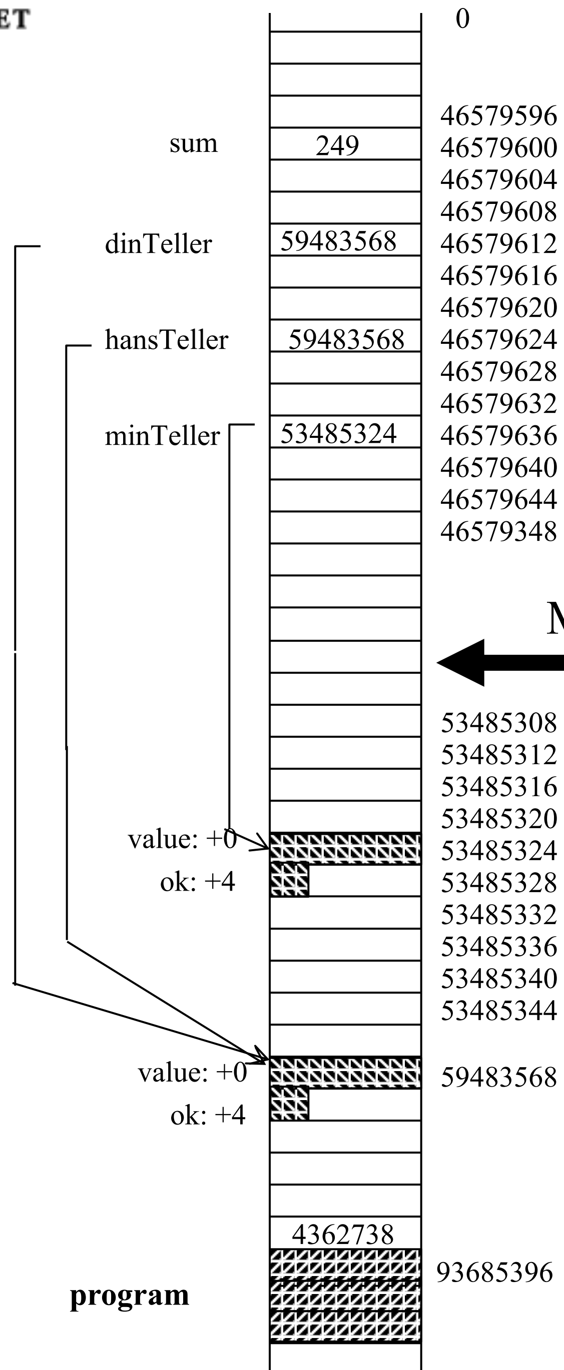


Counter-objekt

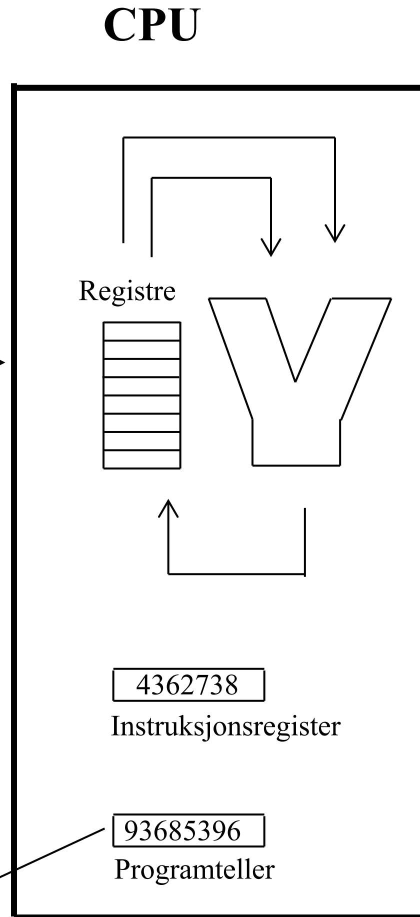
Dette er en tegning som bare er ment som en forklaring av referanser.

Kompilatoren og kjøretidsystemet setter av plass til variabler i minnet

Antar 32 bits adresser for referanser



Minnebuss



sum ligger i minnelokasjon (byte) 46579600, 46579601, 46579602 og 46579603

dinTeller ligger i minnelokasjon 46579612, 46579613, 46579614 og 46579615

hansTeller ligger i minnelokasjon 46579624, 46579625, 46579626 og 46579627

minTeller ligger i minnelokasjon 46579636, 46579637, 46579638 og 46579639

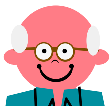
value ligger i starten av objektet, mens ok ligger 4 byte ut i objektet

UNIVERSITETET I OSLO

Institutt for informatikk

Siste "repetisjon" før subklasser

Når skal vi tenke og tegne datastrukturene i programmet ?



- Når du planlegger oppgaveløsningen og virkemåten til programmet
- Når du koder / programmerer
- Når du leter etter feil
- Når du vedlikeholder programmet
- Når du forklarer programmet for andre

- Når du skal lære deg å programmere

UNIVERSITETET I OSLO

Institutt for informatikk

Ukens tema: Arv og subklasser

- Objekter bruker vi til å modellere den virkelige verden (eller komponenter av programmet vårt) inne i datamaskinen.

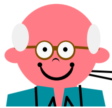
- Vi må alltid strukturere og arrangere begrepene våre og lage mer ryddige, oversiktlige og utvidbare komponenter, moduler og modeller

- Til dette bruker vi bl.a. arv og subklasser

Kortversjon: Definisjon av subklasser

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

```
class Kl { . . . }
class Sub extends Kl { . . . }
```



Nytt Java nøkkelord: extends

Navn på klasser og subklasser (Kl og Sub 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.

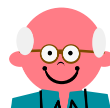
Alle personer skal behandles som . . . .

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 . . . .



Substantivmetoden

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)

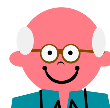
Tilbake til vårt system: Universitetsregisteret

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 . . . .

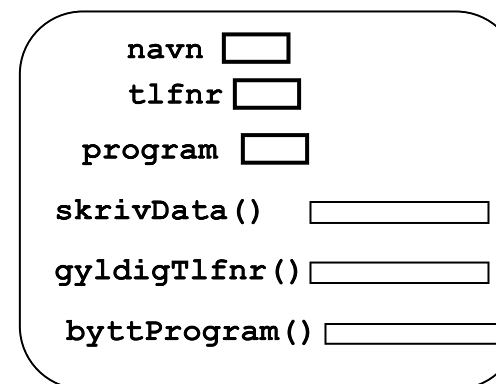
Kristen Nygaard



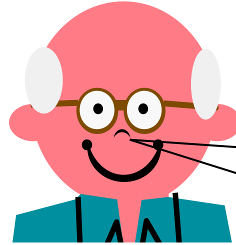
Hva er vårt perspektiv her?

Klassen Student


```java
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



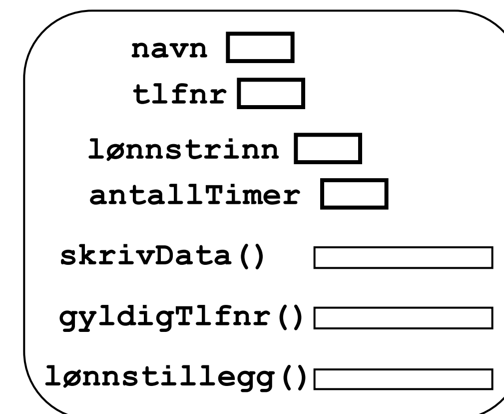
Hm –
på forrige lysark var det ingen egenskaper som var ”private” eller public” ?

OK –
Vi kommer tilbake til det på lysark 39

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

Klassen Ansatt

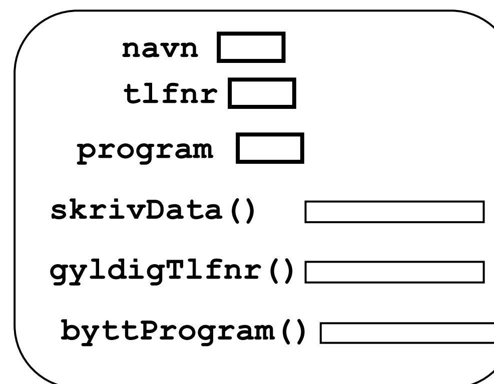
```java
class Ansatt {
    String navn;
    int tlfnr;
    int lønnstrinn;
    int antallTimer;

    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;
    }
}
```



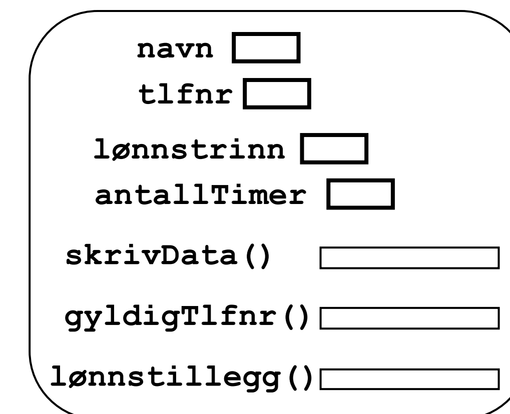
objekt av klassen Ansatt

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

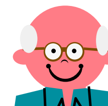
Kan samle det som er felles i en egen, mer generell, klasse

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


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

navn
tlfnr
gyldigTlfnr()

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

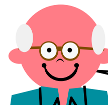
Kan nå gjøre Student og Ansatt til *subklasser* av Person:

```
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;
    }
}
```

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

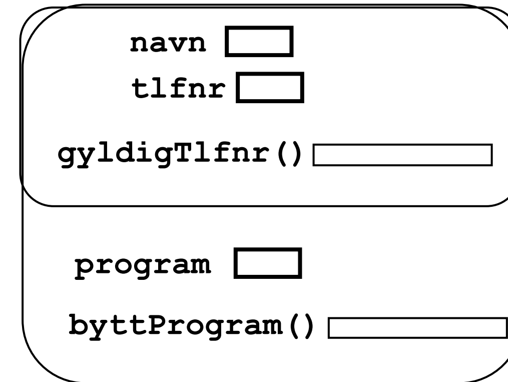
Hva med skrivData()?
- Kommer tilbake til denne…



Nytt Java nøkkelord: extends

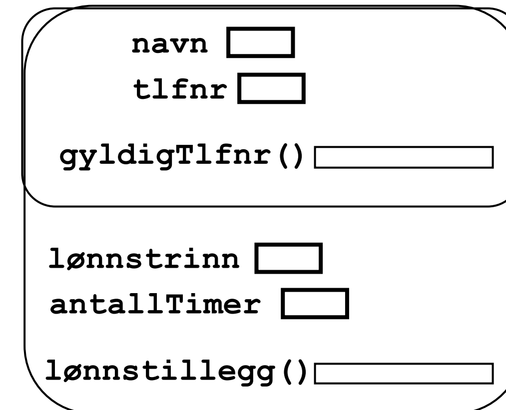
Student og Ansatt som subklasser av Person

```
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;
    }
}
```



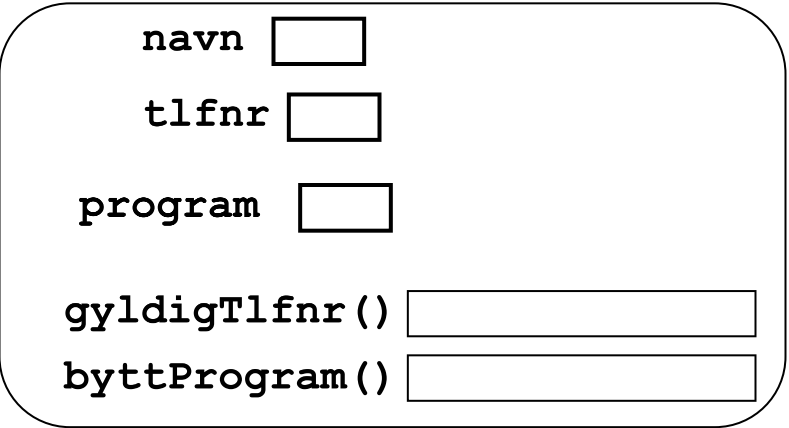
Ansatt-objekt

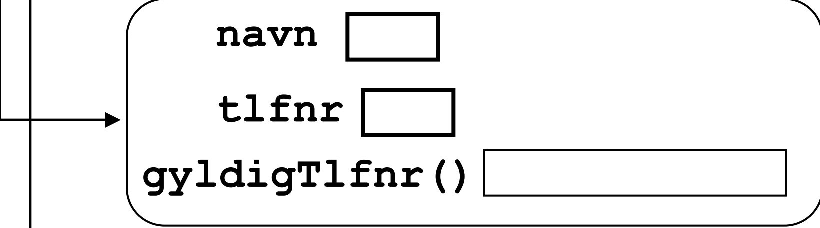
```
class Student {
    String navn;
    int tlfnr;
    String program;

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

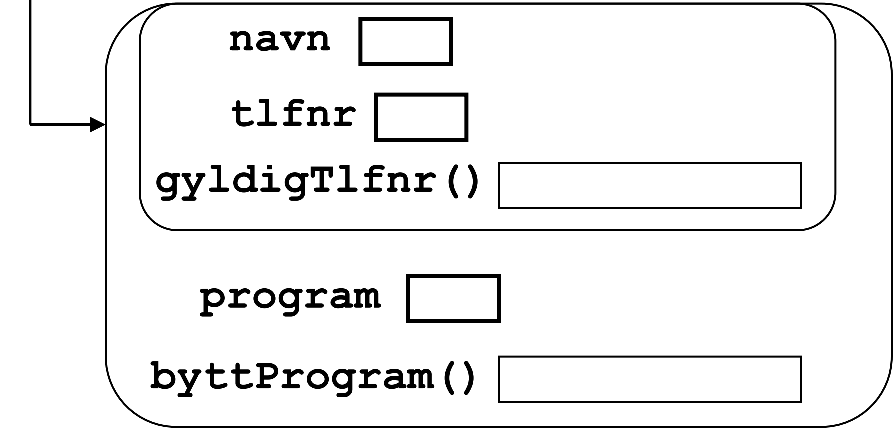
```
class Person {
    String navn;
    int tlfnr;

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

```
class Student extends Person {
    String program;

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

Eksempler på objekter av klassene







Bruk av en subklasse

UNIVERSITETET I OSLO

Institutt for informatikk

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

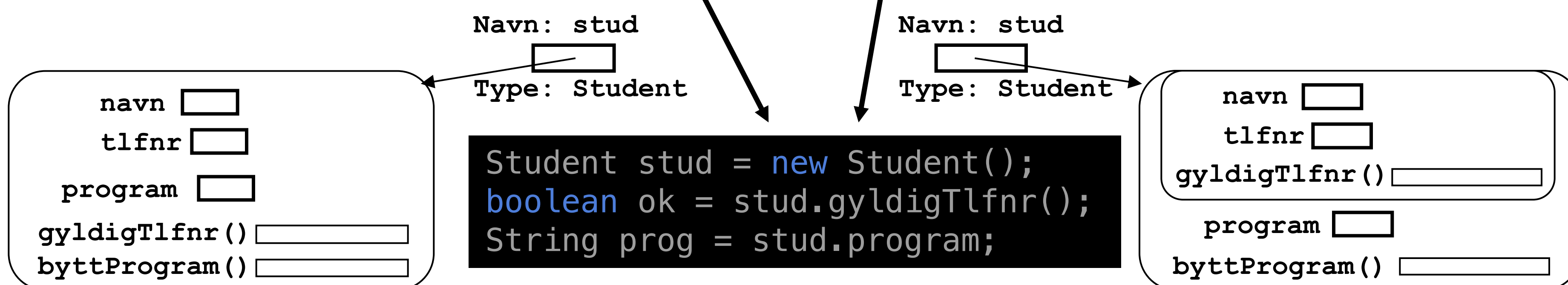
Før:

```
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) { . . .  }
}
```

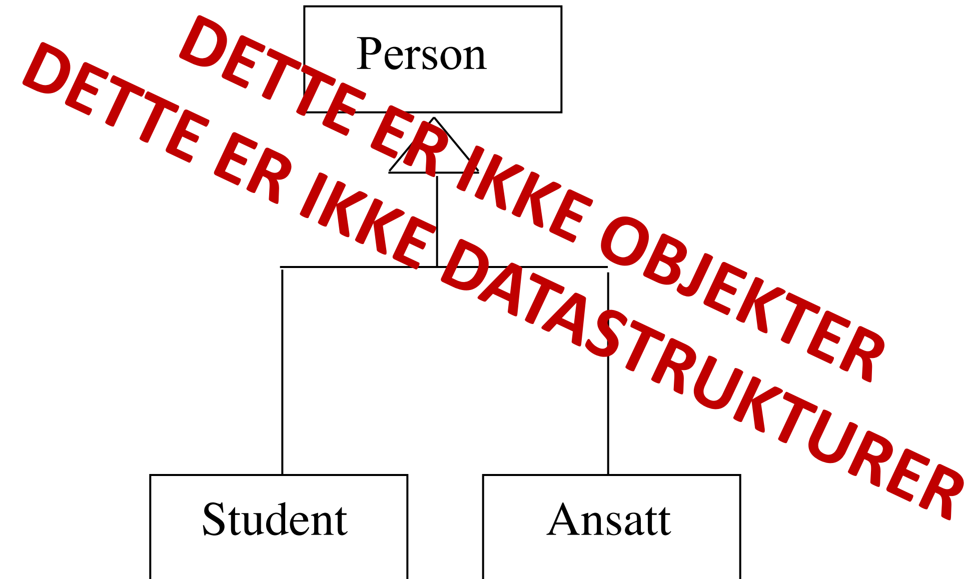


Tegning av subklassehierarki

```
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) { . . . }
}
```



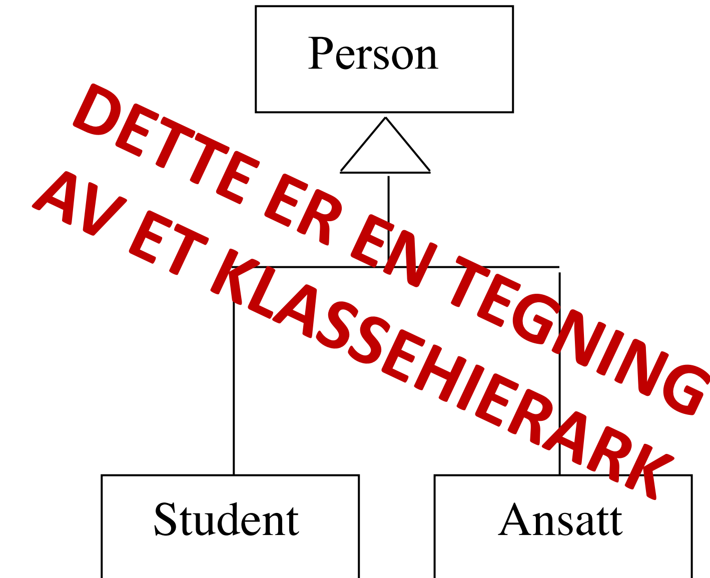
I IN1010 er riktige UML-tegninger ikke viktig (men subklassehierarkier er viktig)

Notasjon for **subklassehierarki**
(med bare det helt nødvendige)

```
class Person {

}
```

```
class Student extends Person {

}
```

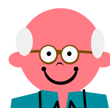
```
class Ansatt extends Person {

}
```



I IN1010 er riktig UML-notasjon ikke viktig (men subklassehierarkier er viktig)

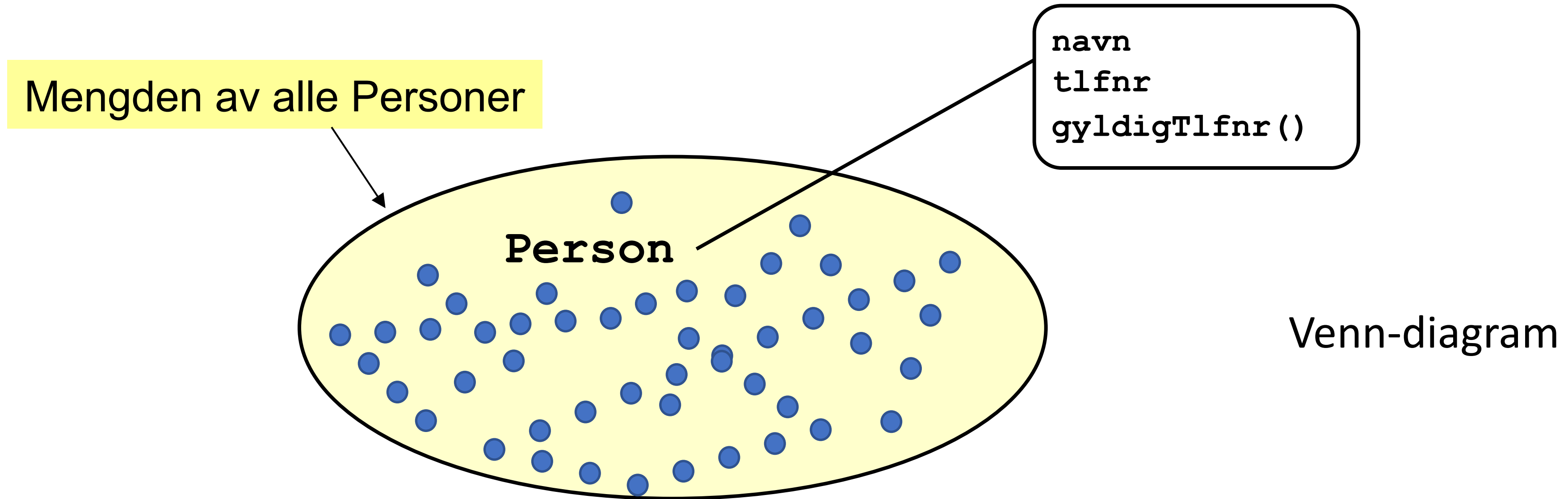
Igjen: Hva er en **subklasse**?

- 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/variable/konstanter).
- En subklasse er altså en mer **spesialisert** utgave av klassen den bygger på.
- Klassen vi bygger på kalles en ***superklasse***.



Dette fant Ole-Johan Dahl og Kristen Nygaard på i ca. 1963, og laget programmeringsspråket Simula

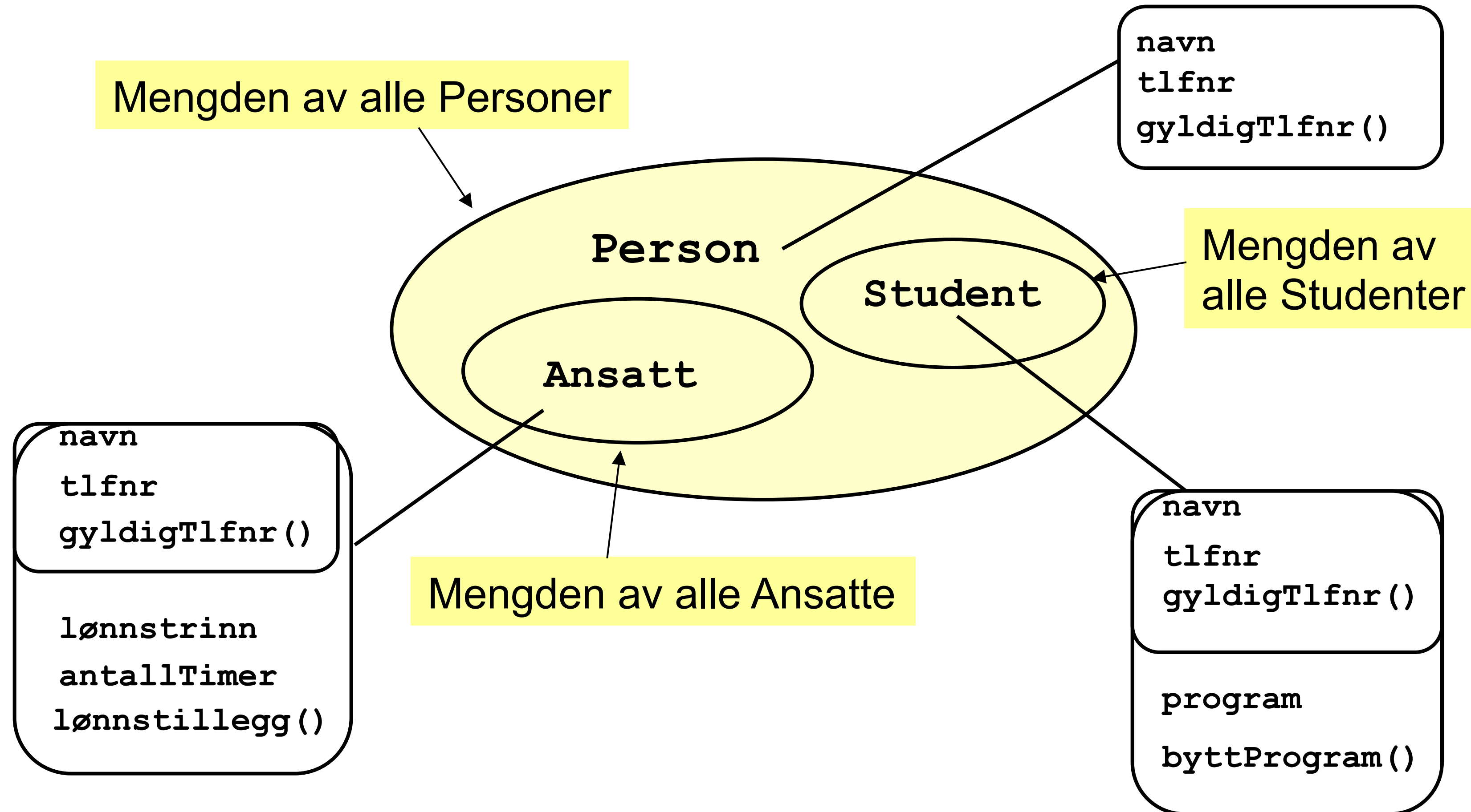
Alle Personene i programmet vårt



Venn-diagram

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

Spesialisering - Generalisering



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

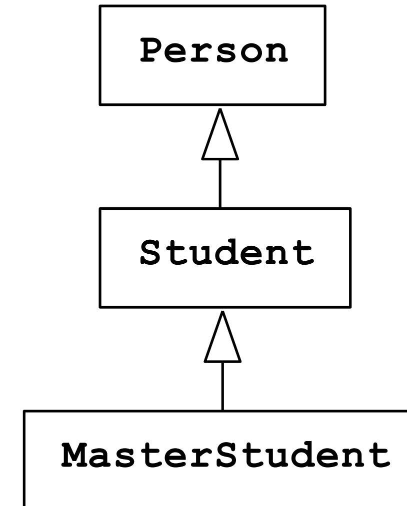
Klasse-hierarkier

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

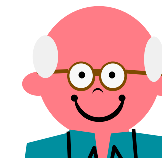
```
class Person {

}
```

```
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.



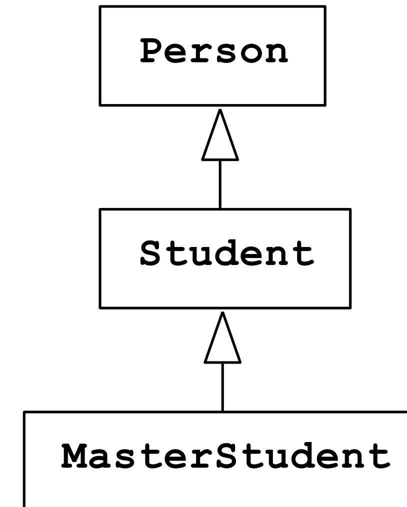
KLASSE-HIERARKI, IKKE DATASTRUKTUR !

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)

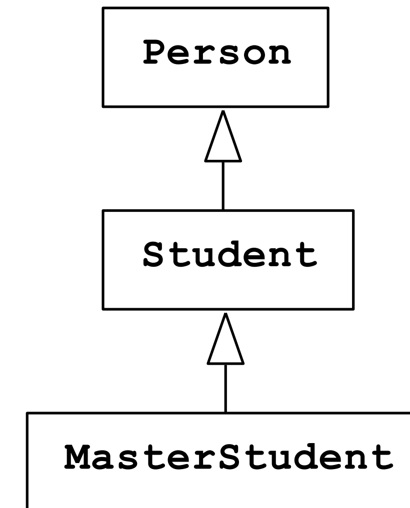
Klasse-hierarkier

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

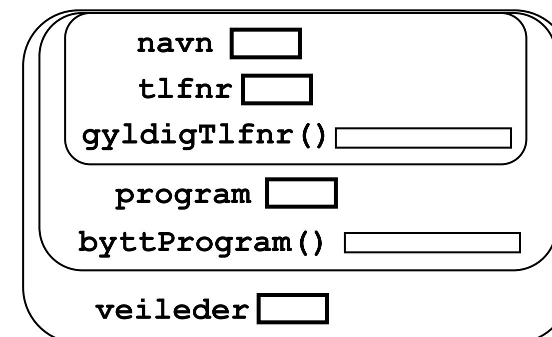
```
class Person {
    String navn;
    int tlfnr;
    boolean gyldigTlfnr() { . . . }
}
```

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

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



MasterStudent-objekt

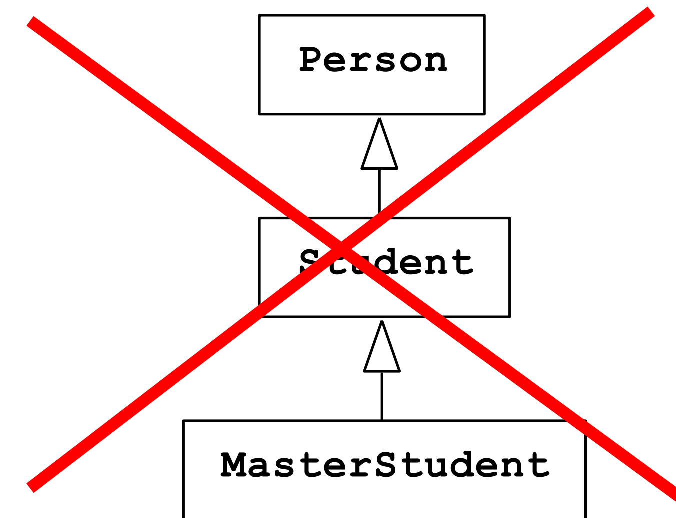


Når programmet utføres lages det datastrukturer / objekter

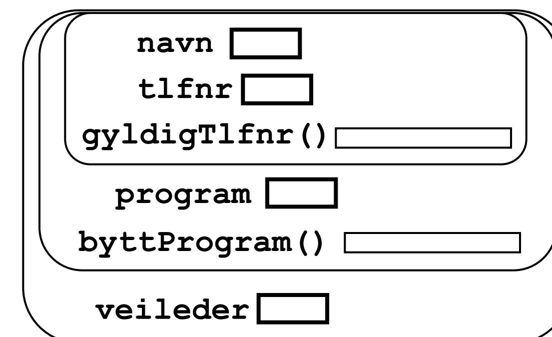
```
class Person {
    String navn;
    int tlfnr;
    boolean gyldigTlfnr() { . . . }
}
```

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

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

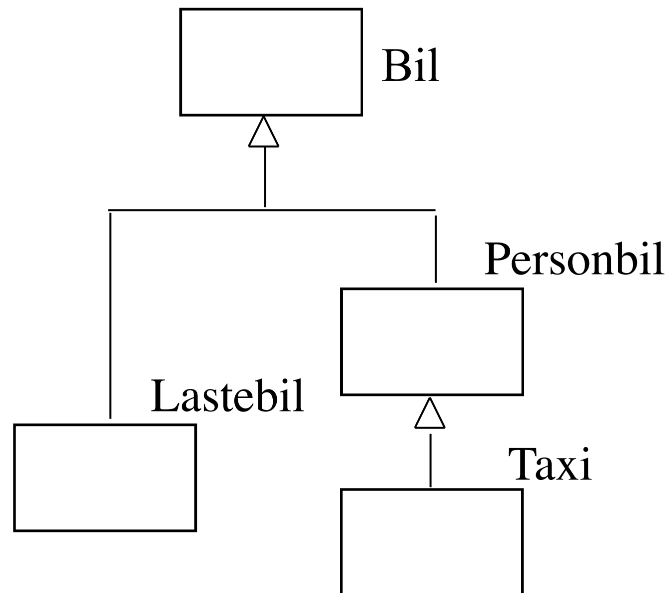


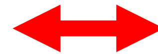
MasterStudent-objekt



Klasser - Subklasser

Klassehierarki:





class Bil { . . . }

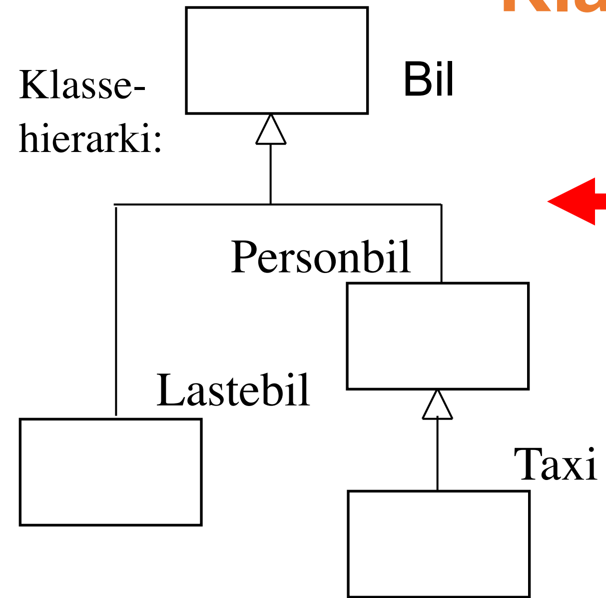
class Personbil extends Bil { . . . }

class Lastebil extends Bil { . . . }

class Taxi extends Personbil { . . . }

33

Klasser - Subklasser



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

Alle biler

Alle taxier

Alle personbiler

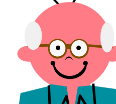
Alle lastebiler

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
   - Bra når vi skal lage et oversiktlig stort program

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

1 og 2 er klart viktigst



Mer om dette neste uke

35

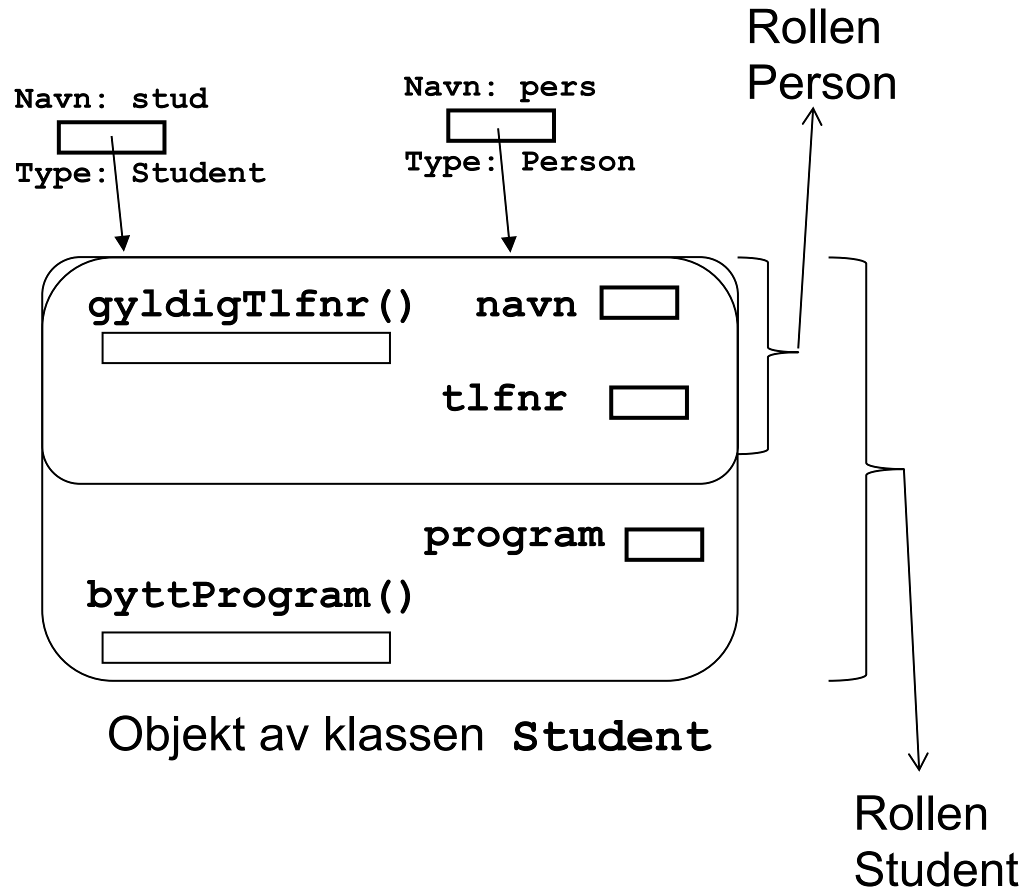
NYTT

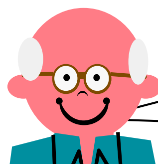
Ulike referansetyper (pekertyper)


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

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

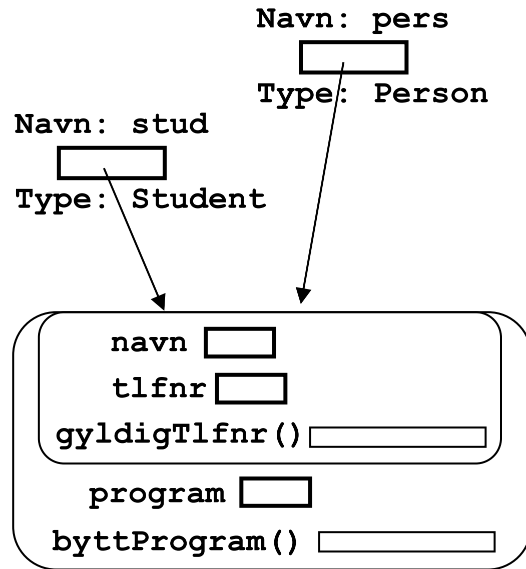
```
class Student extends Person {
    String program;
    void byttProgram(String nytt) { . . .  }
}
```





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

Ulike måter å se et objekt på



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

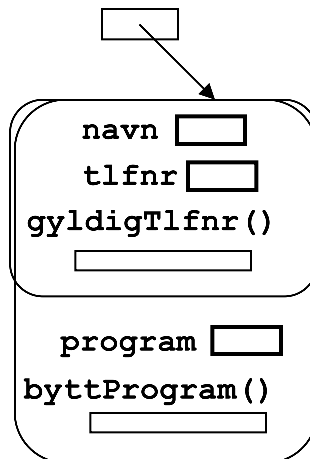
Student stud;
Person pers;
stud = new Student();
pers = stud;

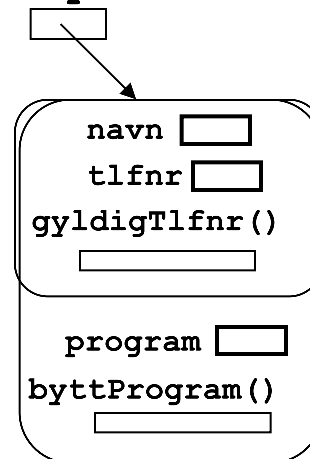
- Typen (klassen) til et objekt er uforanderlig.
  Et objekt kan likevel ***fremtre** for oss* på ulike måter
  Det kan spille forskjellige roller.
- Et objekt av klassen
  `class Student extends Person {...}`
  kan vi se på som et objekt av typen (klassen)
  - **`Person`**: da er egenskapene som er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

Eksempler


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

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





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");**~~

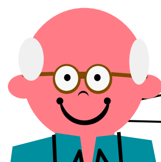
Endelig: private og public i subklasser

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**

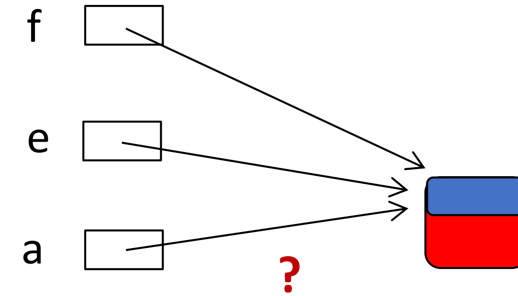
```
class Student extends Person {
    protected String program;

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

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

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

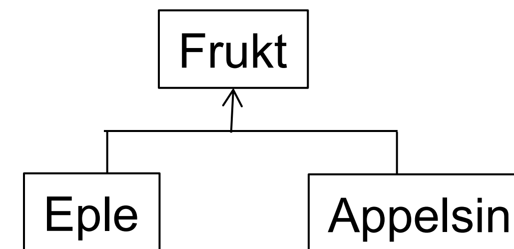
Om det hadde stått “private int antallTimer”, så ville ingen subklasser til Ansatt kunne se denne egenskapen

Tilordning av referanser

```
class LagFrukt {
    public static void main(String[] args){
       Frukt f;
       Eple e;
       Appelsin a;
       e = new Eple();
       f = e;
       a = f;  // ???
    }
}
```




```
class Frukt { .. }
class Eple extends Frukt { .. }
class Appelsin extends Frukt { .. }
```

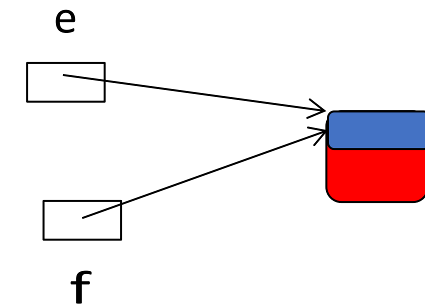


Klassehierarki

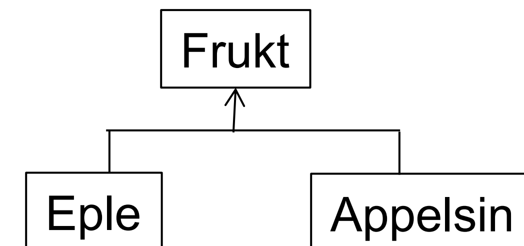
Hva slags objekt er dette?

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 {
    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 { .. }
```



Konvertering av referanser

- Anta at vi har:

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

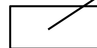
- Ved tilordningen

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

har vi en implisitt konvertering fra Student- til Person-referanse.

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

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



Dette kalles "casting" (class-cast på engelsk), typekonvertering på norsk. Medfører kjøretidstest.

Konvertering av referanser (forts.)

- Hva hvis vi isteden hadde hatt:

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

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

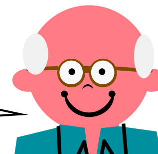
```
java.lang.ClassCastException
```

```
(fordi pers ikke peker på et objekt med alle
"Student" egenskapene)
```

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

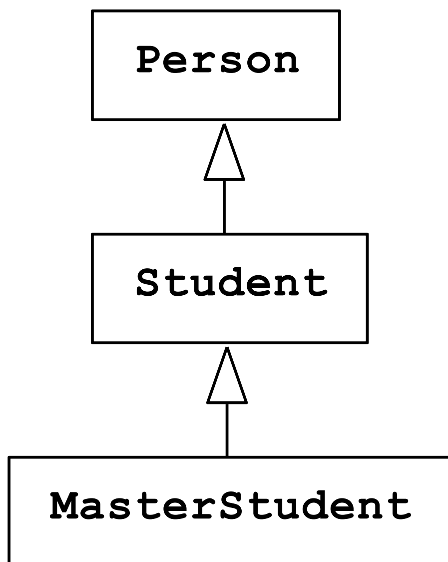
```
if (pers instanceof Student) {
    Student stud = (Student) pers;
}
```

Har objektet som pers peker på alle "Student"-egenskapene ?



Konvertering mellom flere nivåer

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



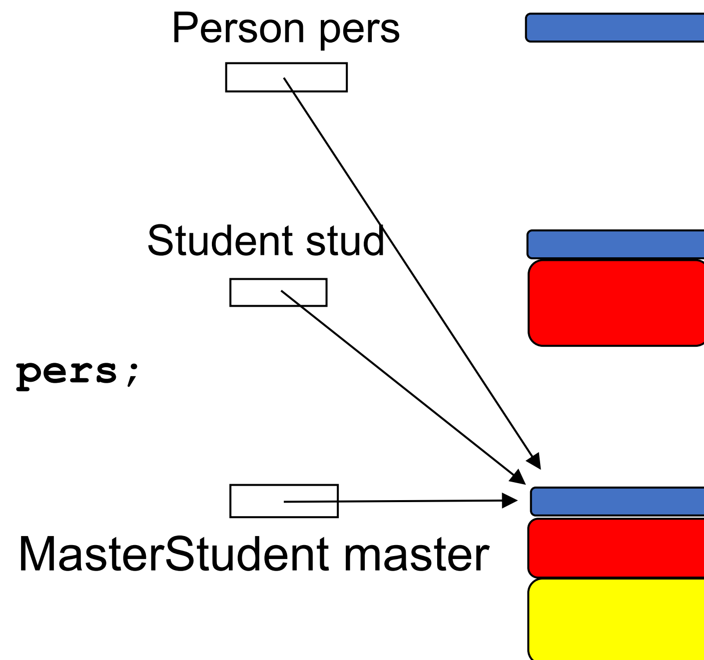
- **Konvertering oppover:**

```
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")

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

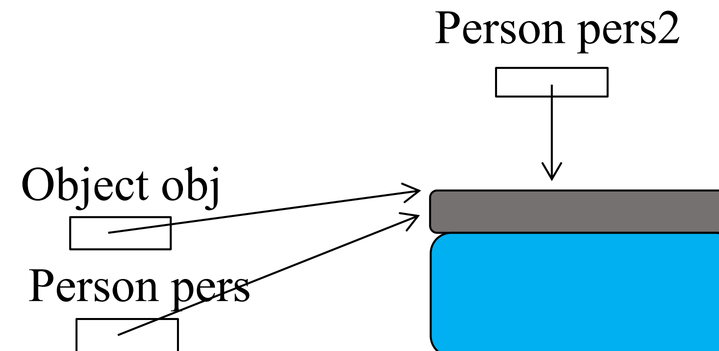
```
    class Person { … }
```

  så tolker Java dette som

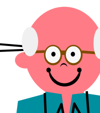
```
    class Person extends Object { … }
```

- 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;
```



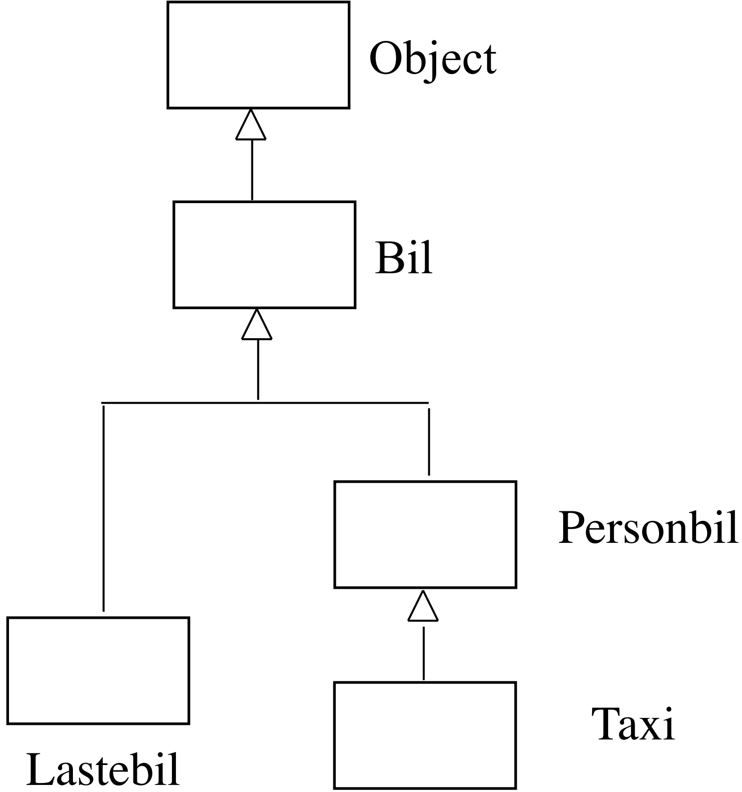
Mer neste gang om hva som er inne i Object-delen av et objekt.

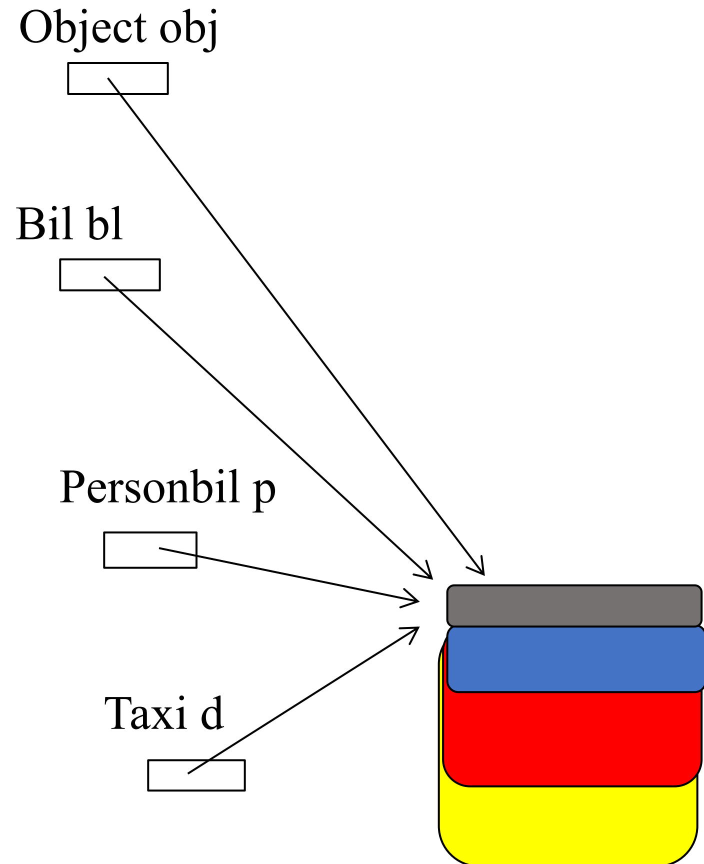


Hint: Bl.a. equals(. . .)

class Object - eksempel

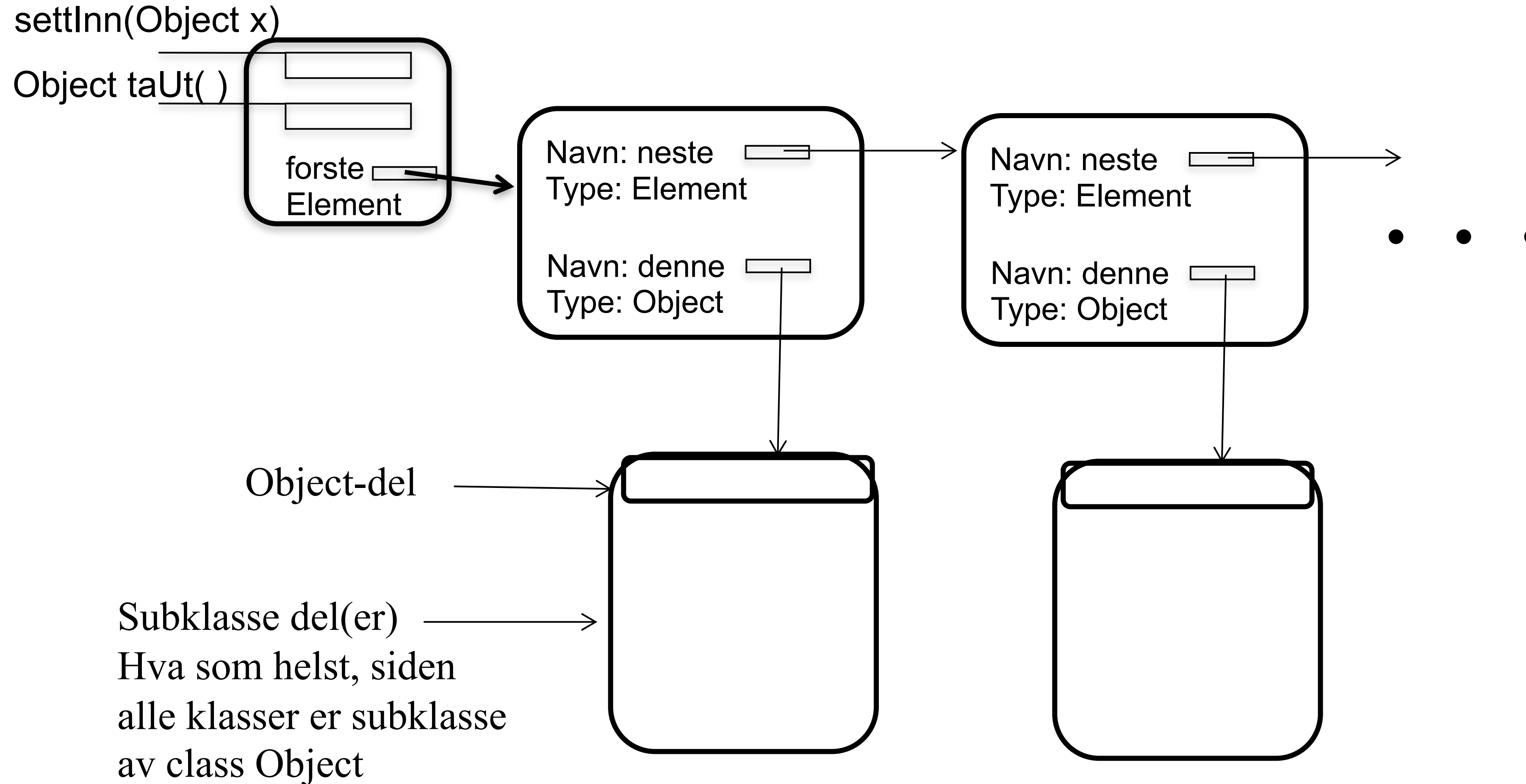
Klassehierarki:





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.

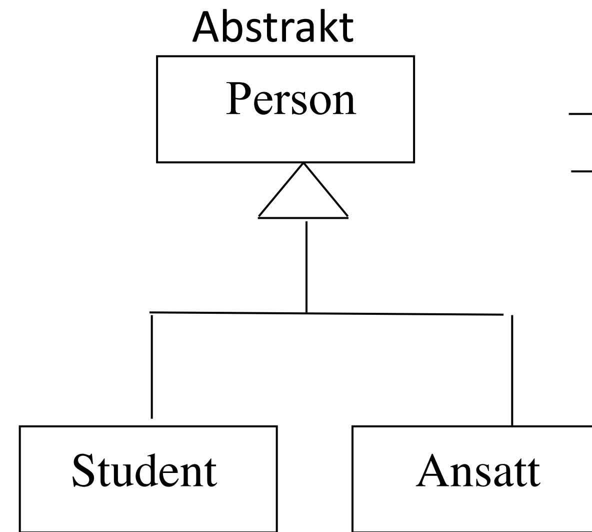
Bare når parameteren og instansvariabelen har samme navn brukes «this» i konstruktører!

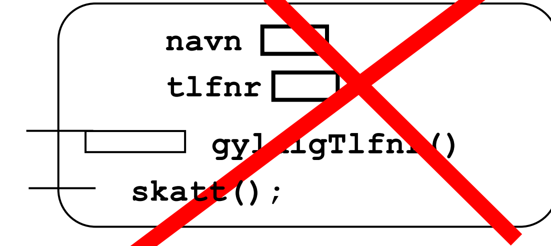
```
class KonstantDemoKlasse {
    protected final int objektId;
    KonstantDemoKlasse (int obId) {
            objektId = obId;
    }
}
```

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() { . . . }
}
```

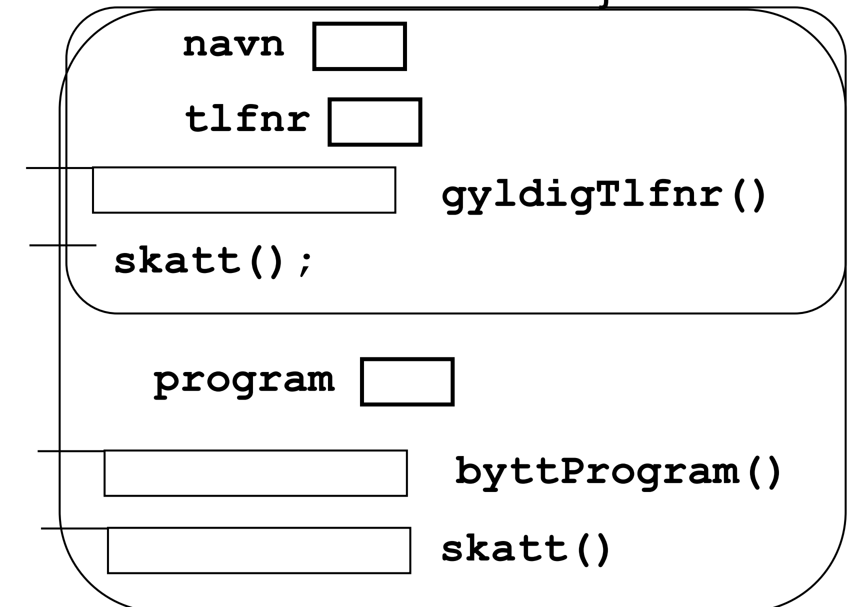




Ikke lov å si
new Person() !

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

Eksempel på et
Student-objekt




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

Mer forklaring neste uke

I/O og unntaksbehandling

Du kan behandle unntaket selv (til høyre)

```
import java.io.*;

class FilTest1 {
    public static void main (String [ ] args)
             throws FileNotFoundException {

        PrintWriter  filut =
            new PrintWriter ("minutfil.txt");
        filut.println( "Utskrift " + 17 );
        filut.close( );
    }
}
```

```
import java.io.*;

class FilTest2 {
    public static void main (String [ ] args) {
        try {
            PrintWriter  filut =
                   new PrintWriter ("minutfil.txt");
            filut.println("Utskrift " + 17 );
            filut.close( );
        }
        catch (FileNotFoundException f) {
            System.out.println ("Fant ikke filen");
        }
    }
}
```

To nye nøkkelord: **try** og **catch**

UNIVERSITETET I OSLO

Institutt for informatikk

Generelt om unntaksbehandling

- Mye kode kan feile og feilaktige situasjoner (unntak) kan oppstå.

Kode som kan feile ***kan*** **-** og som oftest ***må*** - vi legge inn i "try" og fange i "catch"

try { <kodes som kan feile> }

catch (. . .) { <behandle feilen> }

```
try {
    PrintWriter  filut =
             new PrintWriter ("minutfil.txt");
    filut.println("Utskrift " + 17 );
    filut.close( );
}
catch (FileNotFoundException f) {
     System.out.println ("Fant ikke filen");
}
```

Feiler koden blir denne blokken utført med feilobjektet som f peker på som parameter

UNIVERSITETET I OSLO

Fem Java nøkkelord

Institutt for informatikk

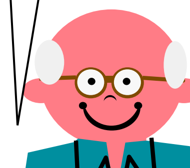
- **try** - Står foran en blokk som er usikker dvs. der det kan oppstå et unntak
- **catch** - Står foran en blokk som behandler et unntak.
  Har en peker til et unntaksobjekt som parameter
- **finally** - blir alltid utført (mer senere)
- **throw** - Starter å kaste et unntak
- **throws** - Kaster et unntak videre
  Brukes i overskriften på en metode som ikke selv vil behandle et unntak
- **Viktigst bruk:**

```
try {
      <kode som kan feile>
}
catch (Unntaksklasse u) {
   <behandle  unntaket,  u peker på et objekt som beskriver  unntaket>
}
```

class Exception


```
try {
        <kode som kan feile>
}
catch (Unntaksklasse u) {
     <behandle  unntaket,  u peker på et objekt som beskriver  unntaket>
}
```

Se neste side



"Unntaksklasse" er en subklasse av klassen Exception.

"Unntaksklasse" er enten en klasse fra Javas bibliotek   eller

det er en klasse vi har deklarert selv (MinBeholderFull på neste side)

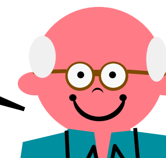
Unntak kan oppstå i egen kode


```
try {
      <Når programmet oppdager at noe er galt,
        f.eks. at en beholder er full:>
         throw new MinBeholderFull( );
   . . . .
   . . . .
}
catch (MinBeholderFull  unt) {
      < Unntaksbehandling.
        Dette hoppes over  når intet
        unormalt/galt/feil har hendt >
}

< her fortsetter programmet
  både etter normal utføring og etter
  behandling av eventuelle unntak >
```

På forhånd har vi deklarert:

```
class MinBeholderFull extends Exception {

}
```

throw
Nå bestemmer vi selv at
et unntak skal oppstå



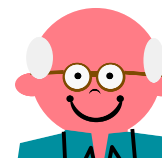
Eksempler på senere forelesninger, grupper og plenum

Når unntak oppstår i en annen metode (og ikke fanges og behandles der)


```
try { . . . .
      p.settInn ( );
 . . . .
}
catch (MinBeholderFull  unt) {
  < Unntaksbehandling.
     Dette hoppes over
     når intet unormalt
     har hendt >
}
< her fortsetter programmet
   både etter normal utføring og
   etter behandling av
   eventuelle unntak>
```

kall på metoden settInn

```
void settInn( ) throws MinBeholderFull {
      . . . .

    < oppdager en feil: >
    throw new MinBeholderFull ( ) ;

    < Normal retur: >
}
```

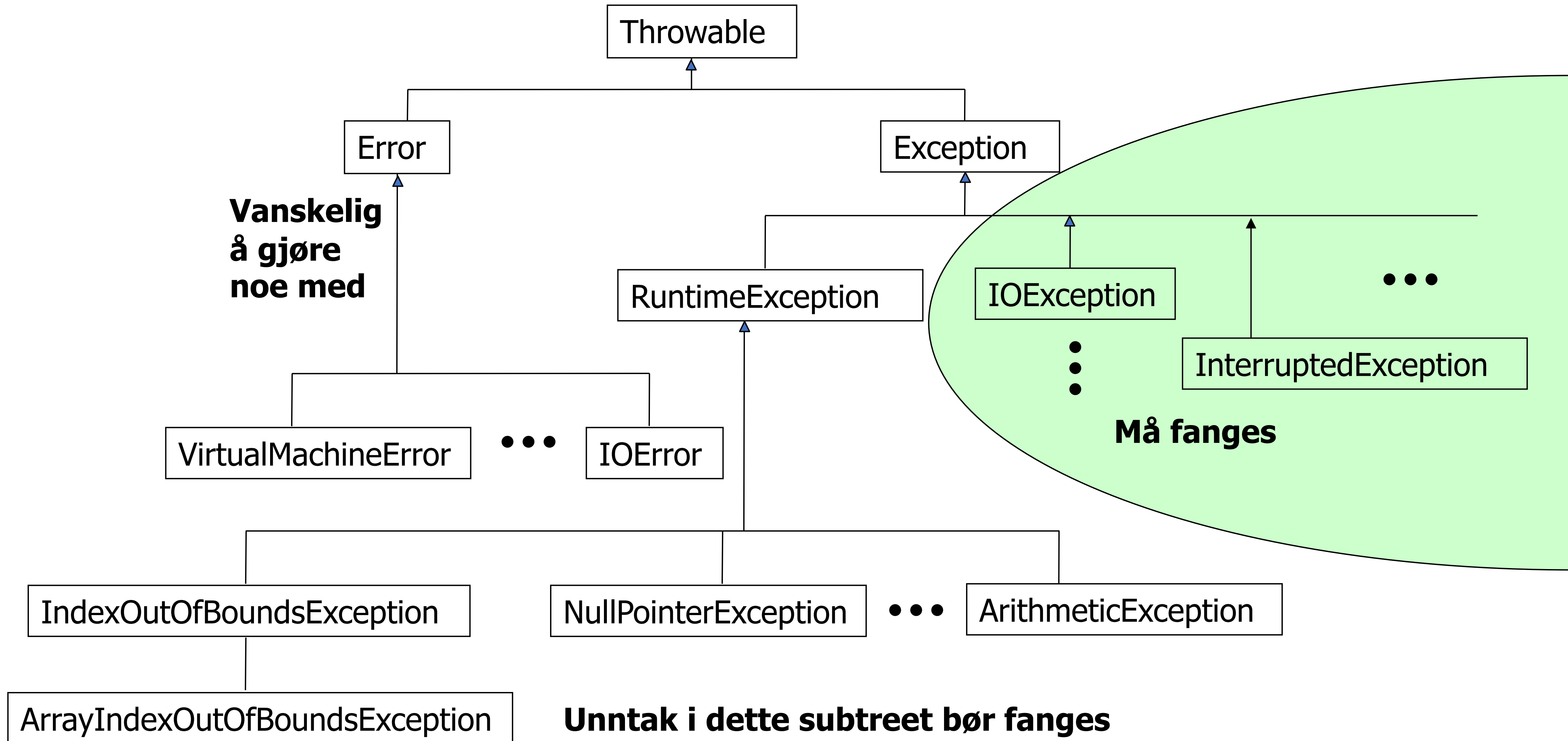


Metoden feiler kanskje fordi kontrakten for kall på metoden ikke ble oppfylt.

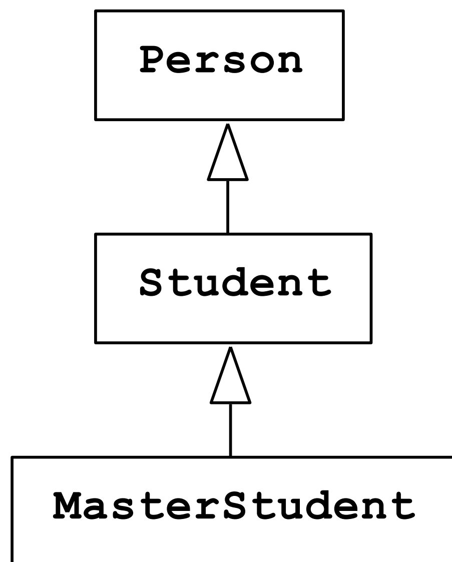
UNIVERSITETET I OSLO

Java-bibliotekets klassehierarki for unntak

Institutt for informatikk



Programmering med subklasser Hoved-"take away"

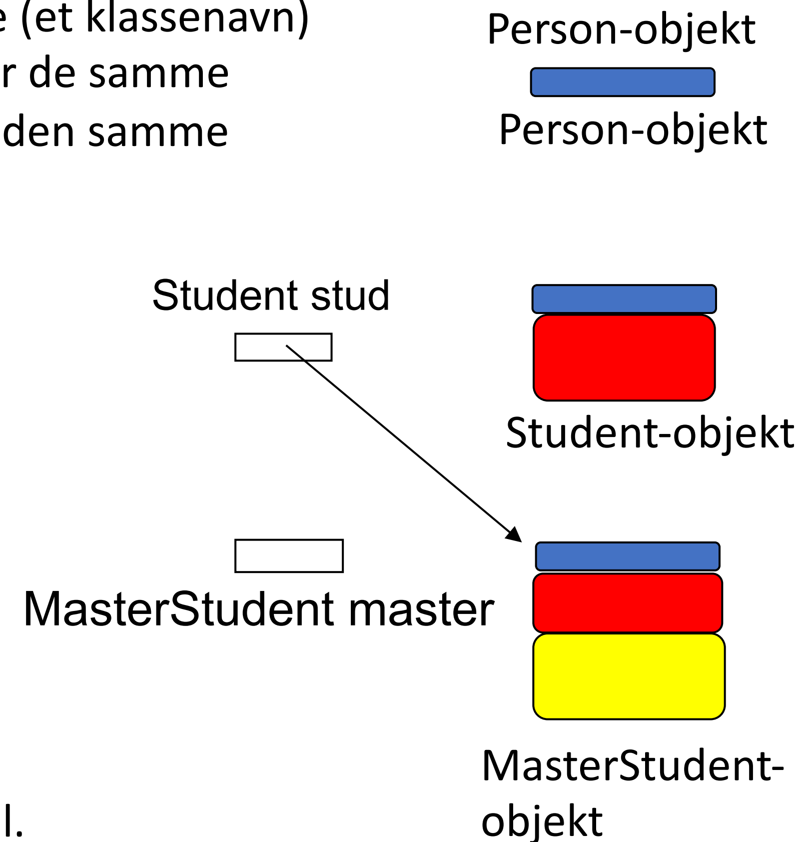


En peker (en referanseverdi) har en type (et klassenavn) og den vil bare peke på et objekt som har de samme (eller flere) egenskapene (objektet er av den samme klassen eller av en subklasse)

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.



I dag har vi lært

- Subklasser - extends
- Generalisering - spesialisering
- Subklasser - submengder
- Referanser av subklassetyper / subklassenavn
- Tilordninger mellom referanser - opp og ned i klassehierarkiet
- Test på objektets egenskaper: instanceof
- class Object
- Abstrakte classer og metoder – abstract
- Unntaksbehandling - try – catch – throw – throws
- Egendefinerte unntaksklasser