UNIVERSITETET I OSLO

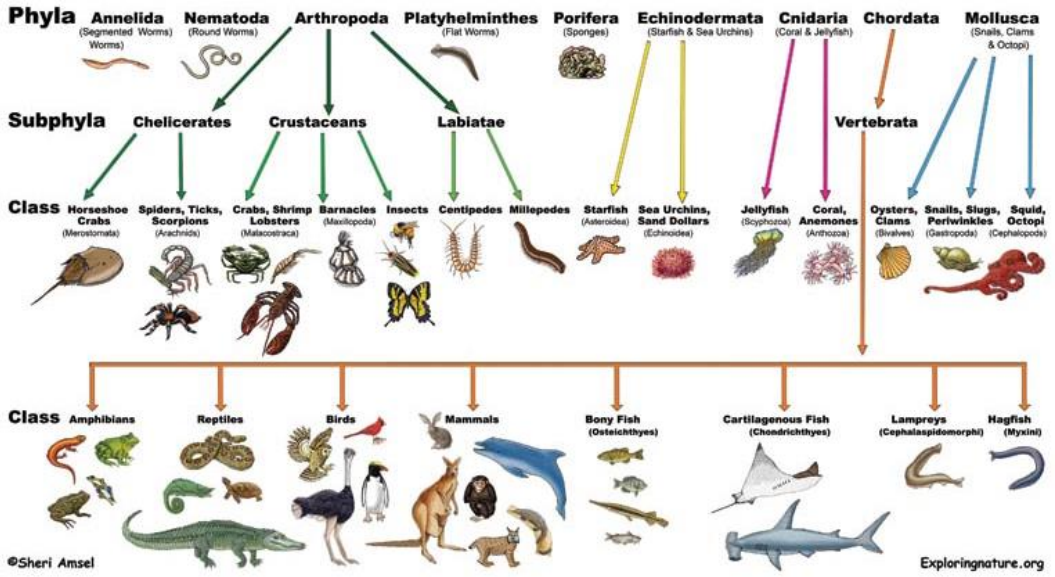
Institutt for informatikk

IN1010 våren 2024

Arv og subklasser – del 1

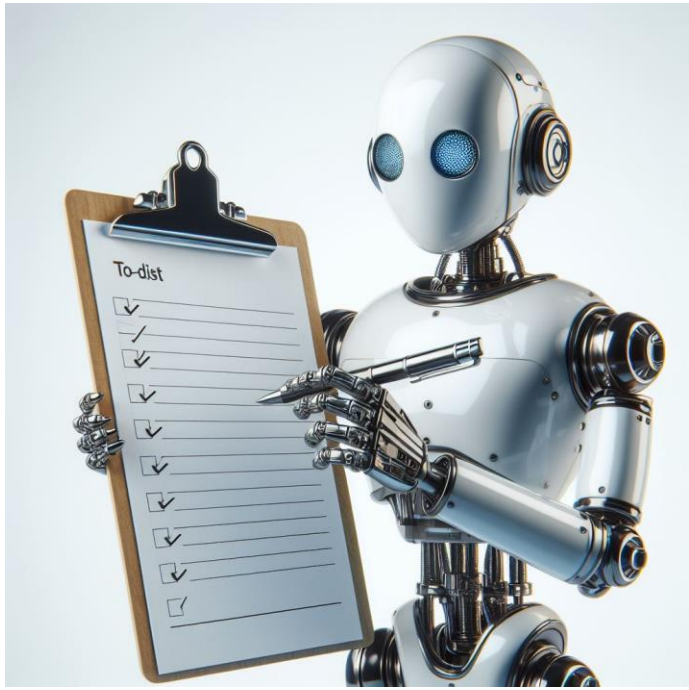
Eyvind W. Axelsen (eyvinda@ifi.uio.no)

Lysark basert på tidligere materiale laget av Stein Gjessing.



Plan for dagen

- Klasser og subklasser
  - Objekter av subklasser
- Referanser til objekter av subklasser
  - Casting / typekonvertering
- Klassen Object
- Abstrakte klasser



```java
public class GodEttermiddag {
  public static void main(String[] args) {
    System.out.println("Hei på deg!");
    System.out.println("Du bør kunne lese dette - sitter du for langt bak?");
  }
}
```

Hvem er jeg?

Eyvind W. Axelsen
(eyvinda@ifi.uio.no)

Førsteamanuensis II ved Ifi, UiO

- Det vil si, jeg er deltidsansatt
- Programmeringsteknologi-gruppa (10. etasje)
- Underviser *IN3040 – Programmeringsspråk* til høsten



CTO/IT-direktør ved Fürst Medisinsk Laboratorium

- Norges største medisinske laboratorium, > 500 ansatte
- > 12 000 pasientprøver per dag, > 200 000 analyser per dag
- Lager "ordentlige" programmer daglig i språk som C#, TypeScript og SQL







Ukens tema: Arv og subklasser I

- Når vi programmerer, så forsøker vi å representere en del av verden utenfor oss, gjerne i form av et spesifikt problem vi ønsker å løse

- Klasser kan brukes til å modellere og strukturere problemet (og løsningen!)
  - Merk at dette ikke handler om å skape virkelighetstro modeller av verden
  - Det finnes også mange andre måter å strukturere og modellere ting på

- Å programmere handler om å strukturere og arrangere begreper og lage ryddige, oversiktlige og utvidbare komponenter, moduler og modeller



Kortversjon: Definisjon av subklasser

- En klasse, her kalt `Klasse`, beskriver objekter med visse egenskaper
- En subklasse av `Klasse`, her 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 { /* ... */ }
```

Nytt Java-nøkkelord:
extends

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.



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.

«Substantivmetoden»



En lite side-kommentar

- 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 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.



Hva er vårt perspektiv her? Hva er det som er viktig for systemet *vi* skal lage?

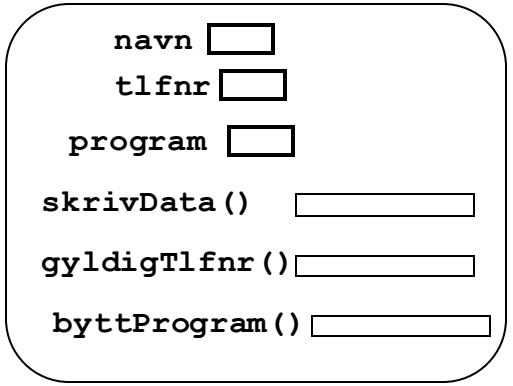
Klassen Student

```
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

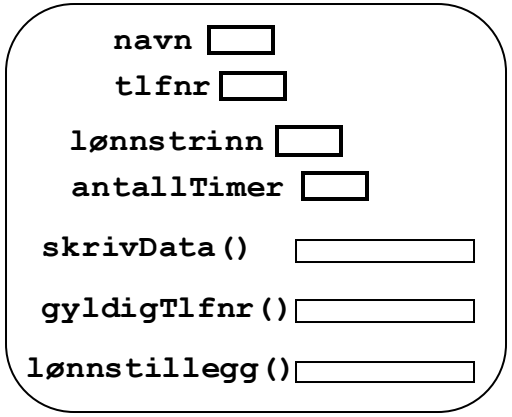
Klassen Ansatt

```
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;
    }

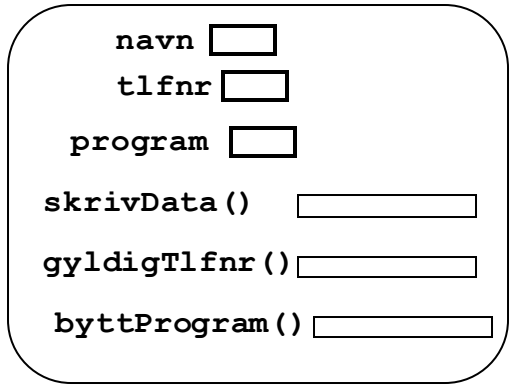
    void lønnstillegg(int tillegg) {
        lønnstrinn += tillegg;
    }
}
```



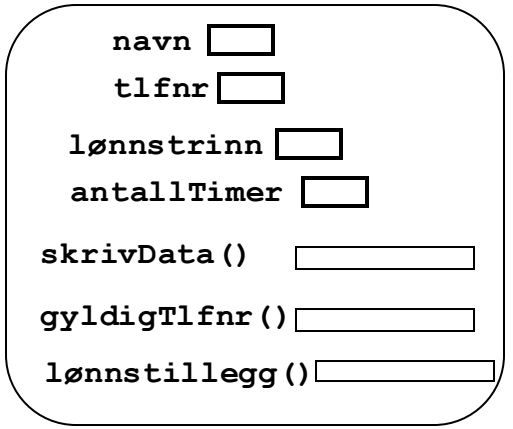
objekt av klassen Ansatt

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



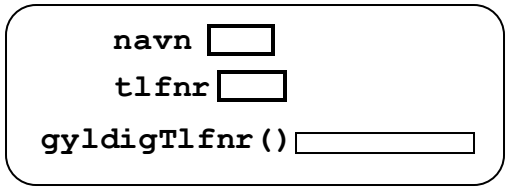
Ansatt-objekt

Klassen Person

Vi 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;
    }
}
```

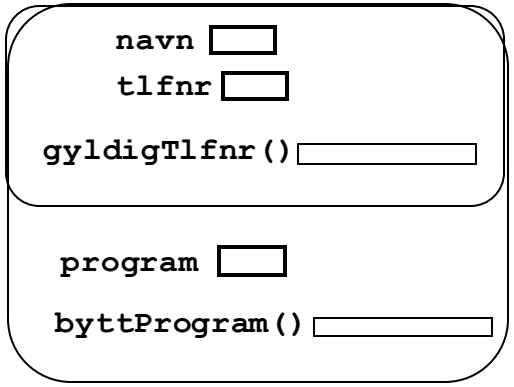


Klassen Person beskriver det som er felles for både Ansatt og Student

Student og Ansatt som subklasser av Person


```
class Student extends Person {
    String program;

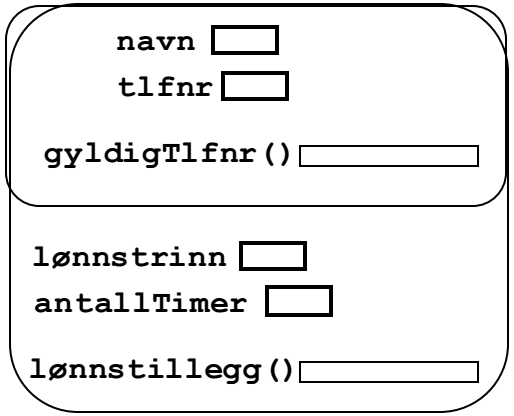
    void byttProgram(String nytt) {
      program = nytt;
    }
}
```



Student-objekt

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

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



Ansatt-objekt

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

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

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

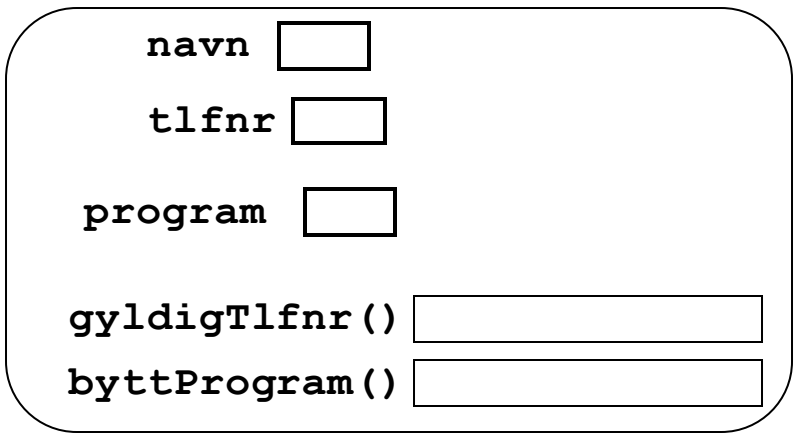
```java
class Person {
    String navn;
    int tlfnr;

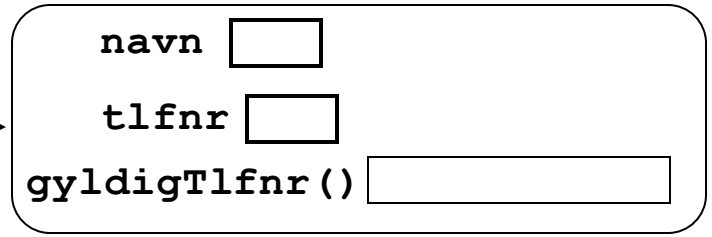
    boolean gyldigTlfnr() {
        return tlfnr >= 10000000 && tlfnr <= 99999999;
    }
}
```

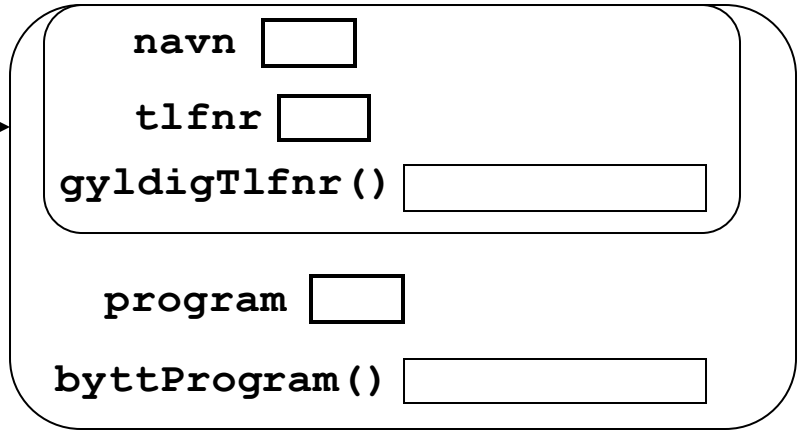
```java
class Student extends Person {
    String program;

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

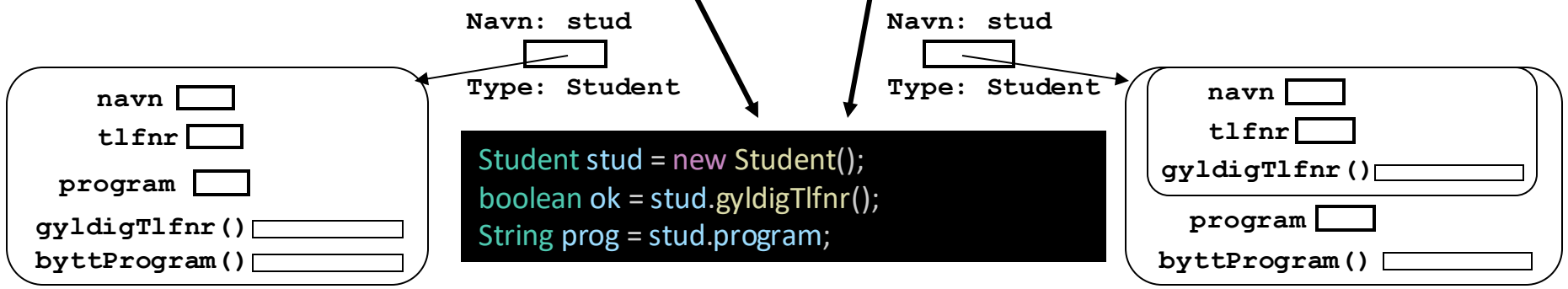
eller med bruk av subklasser (nå):

```
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;

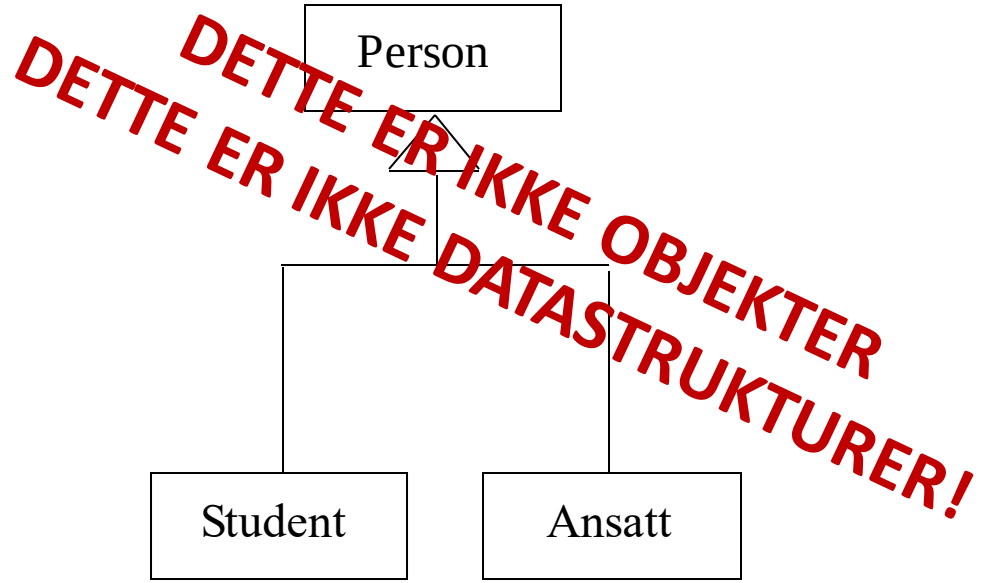
  boolean gyldigTlfnr() { ... }
}
```

```
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 {

  ...
}
```

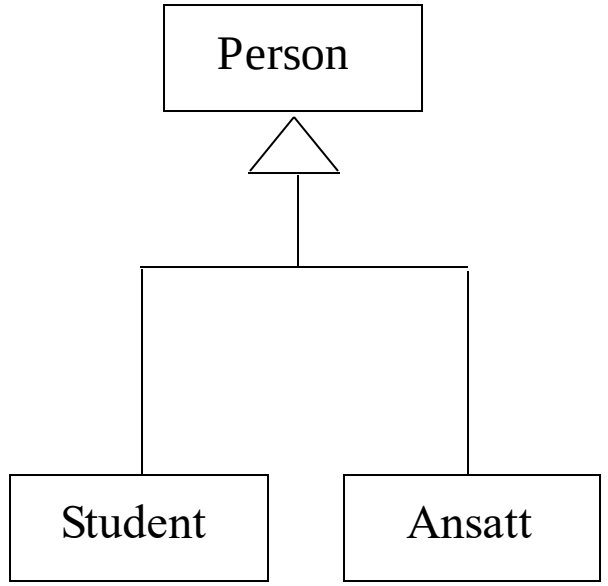
```
class Student extends Person {

  ...
}
```

```
class Ansatt extends Person {

  ...
}
```





En beskrivelse av programmet / problemet

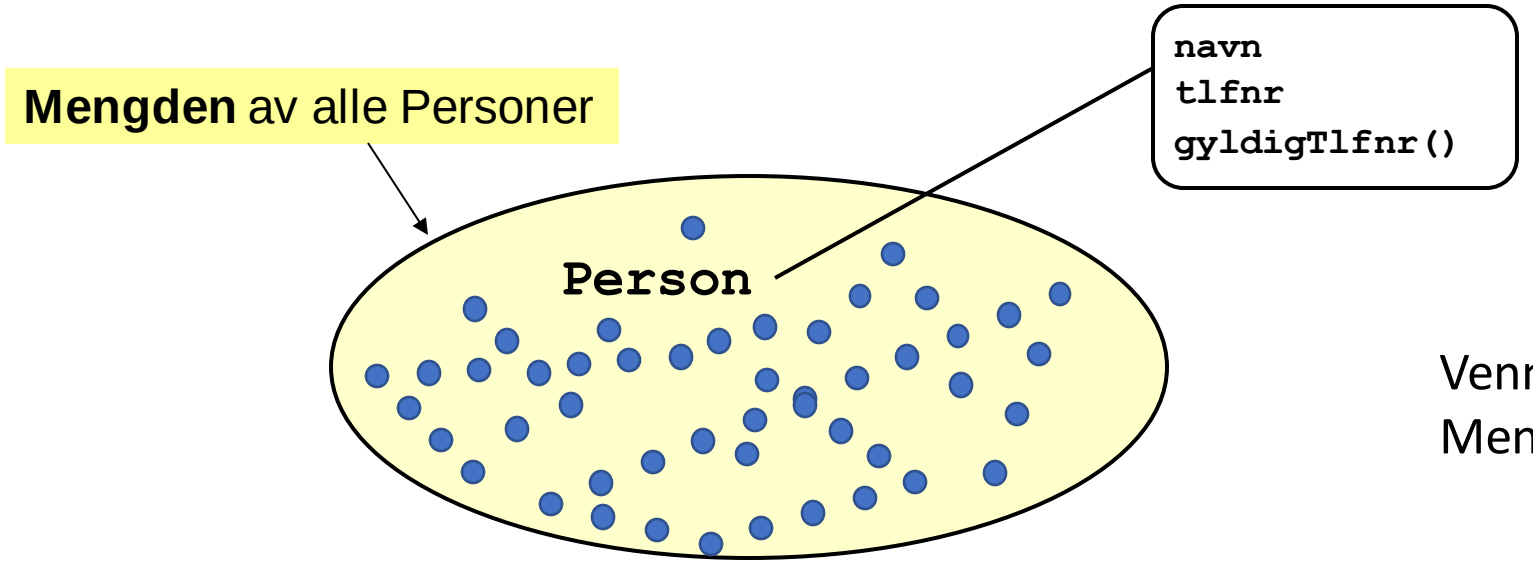
Igjen: Hva er en **subklasse**?

- 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

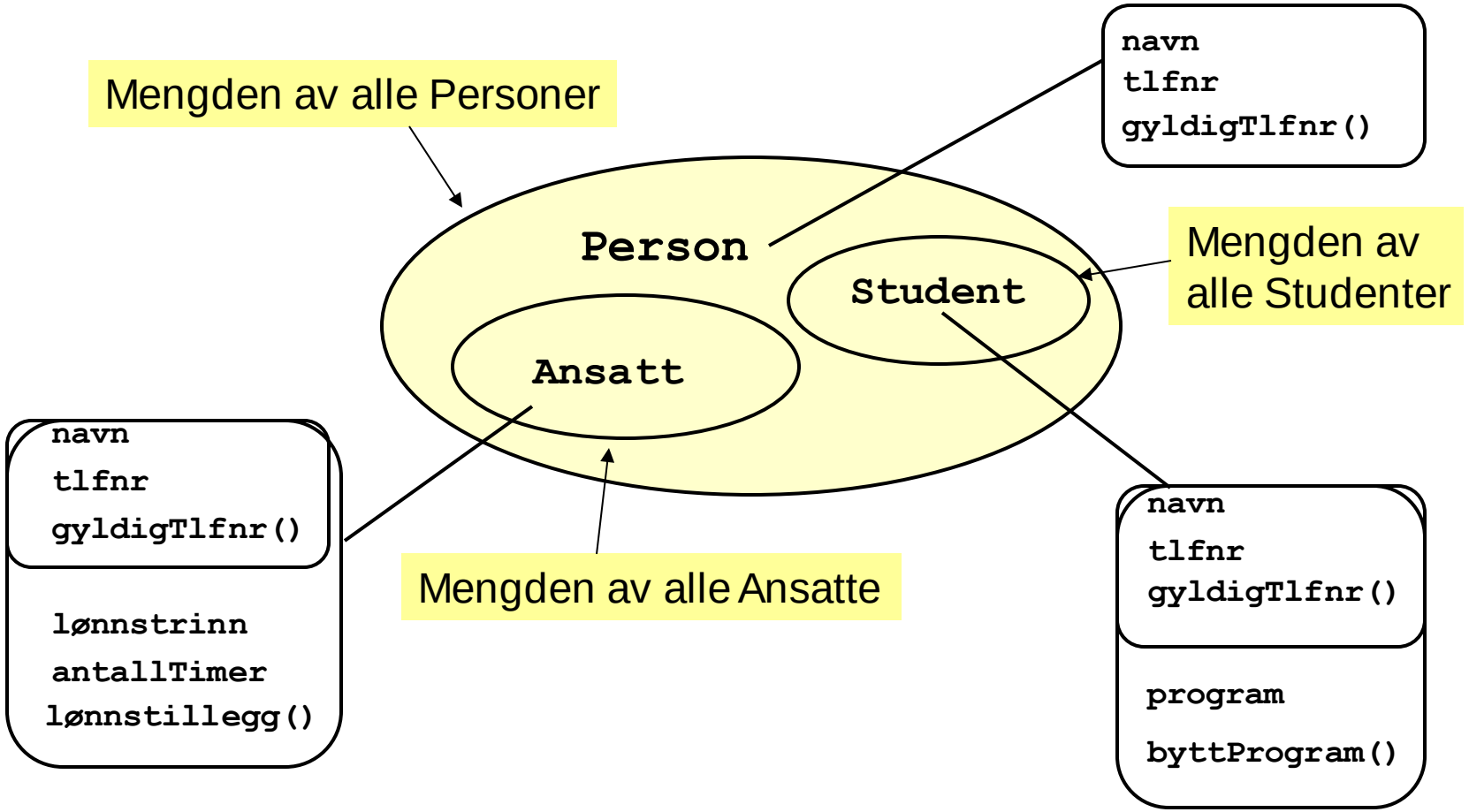
Alle Personene i programmet vårt



Venn-diagram
Mengde-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 {

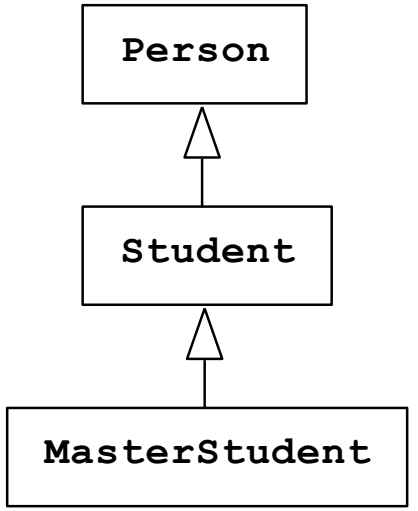
  ...
}
```

```
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 !

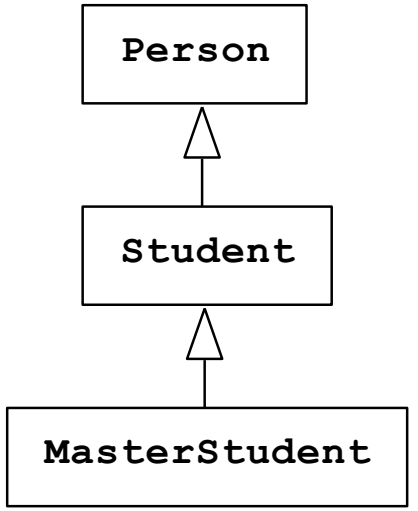
Klasse-hierarkier

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

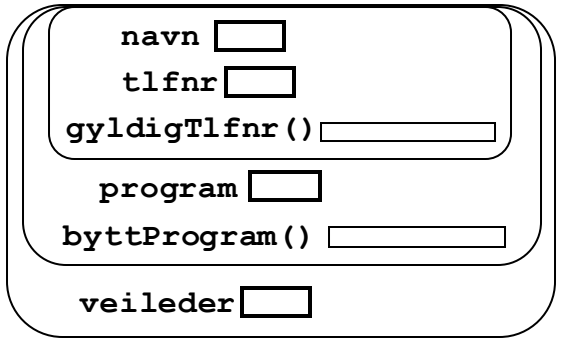
```
class Person {
  String navn;
  int tlfnr;
  boolean gyldigTlfnr() { ... }
}
```

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

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



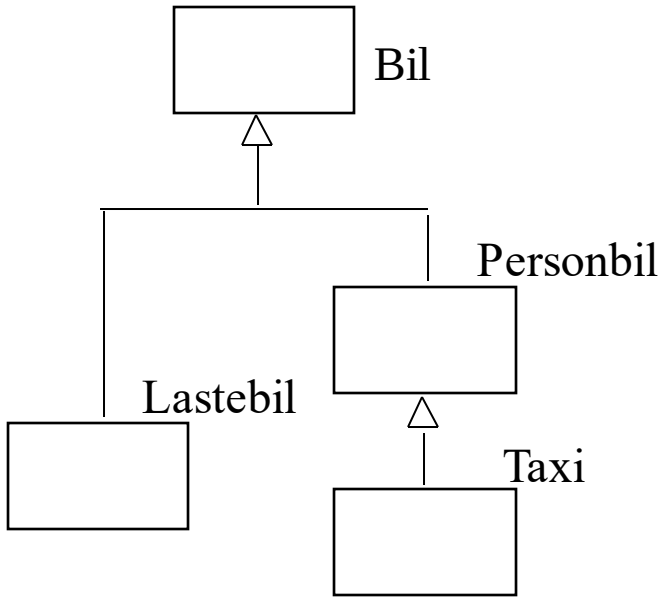
MasterStudent-objekt:



NB! KLASSE-HIERARKI, IKKE DATASTRUKTUR !

Klasser - Subklasser

Klassehierarki:



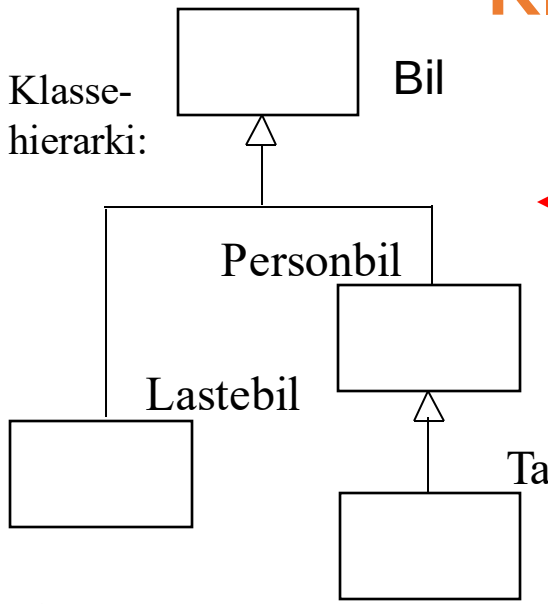

```
class Bil { ... }

class Personbil extends Bil { ... }

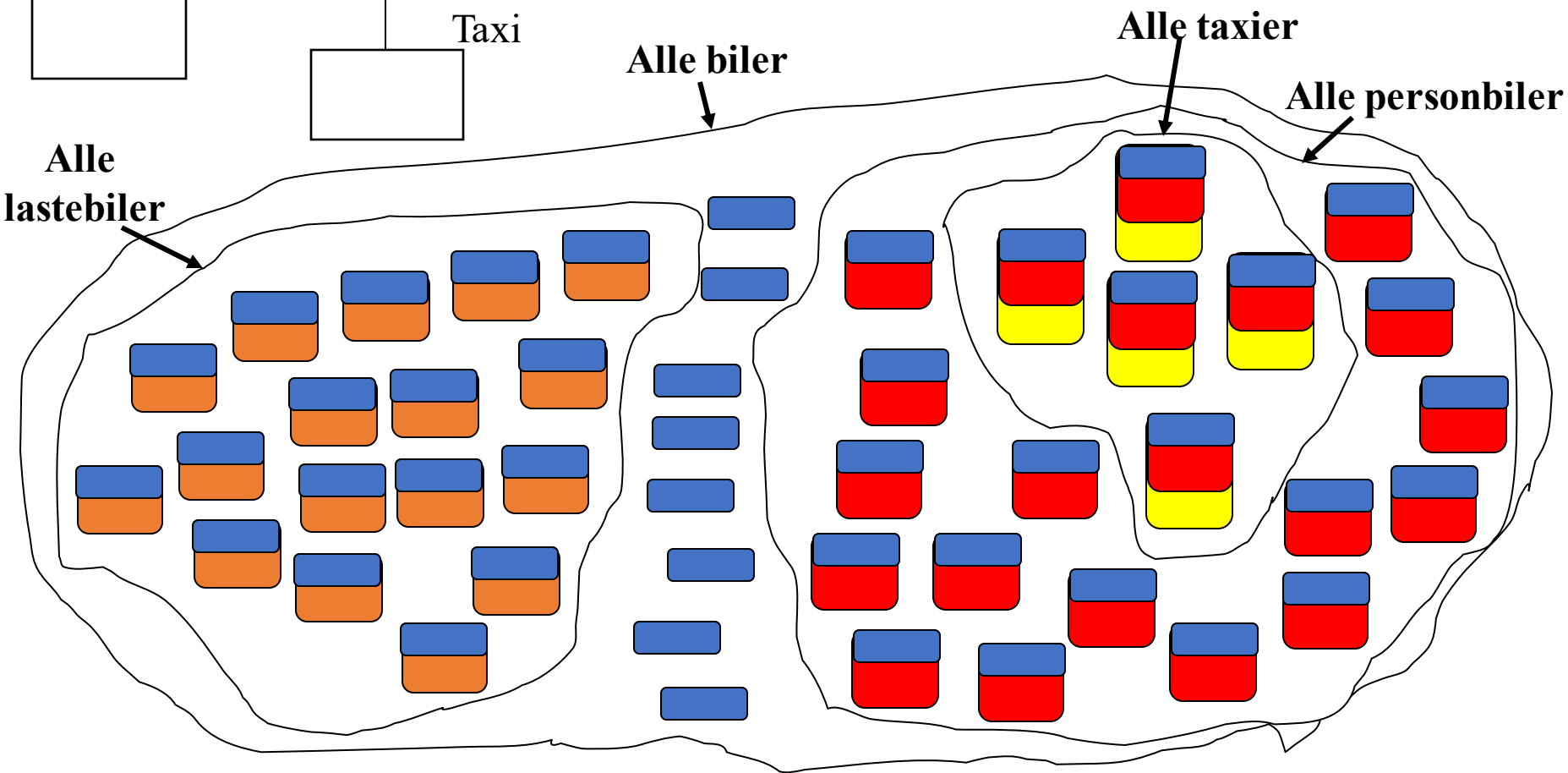
class Lastebil extends Bil { ... }

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

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?

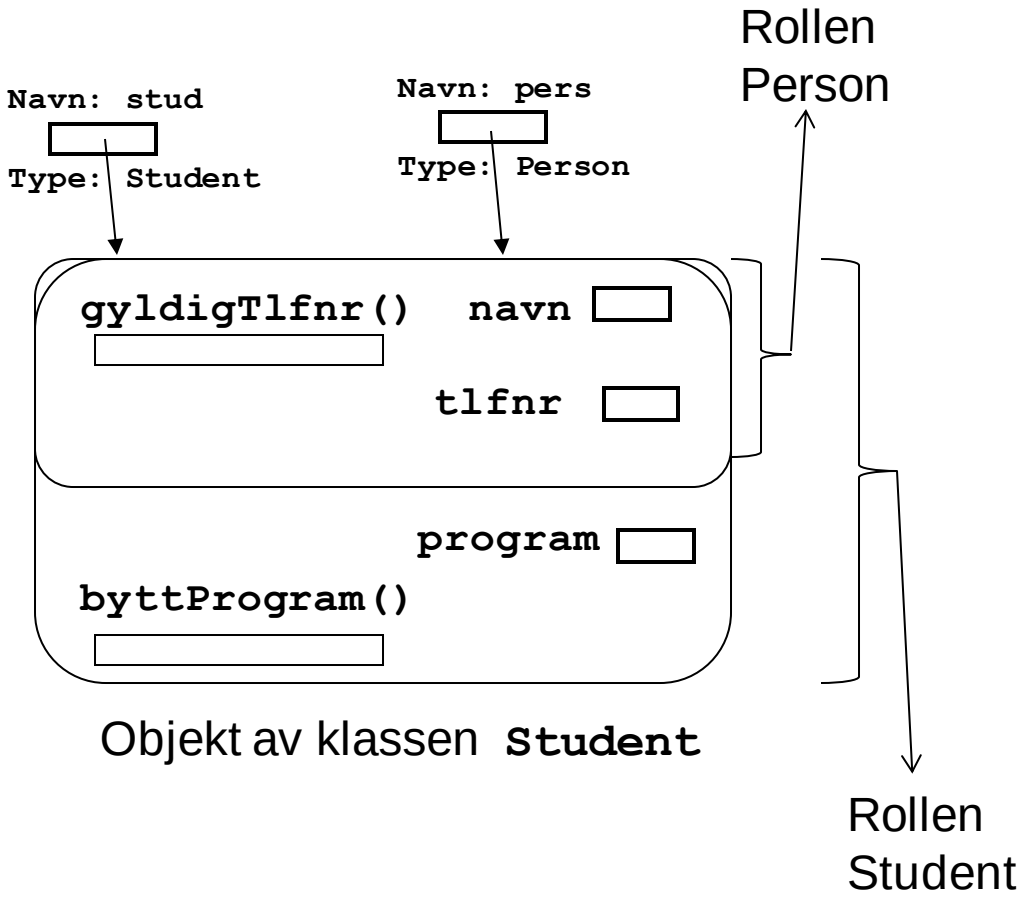
NYTT

Ulike referansetyper (pekertyper)

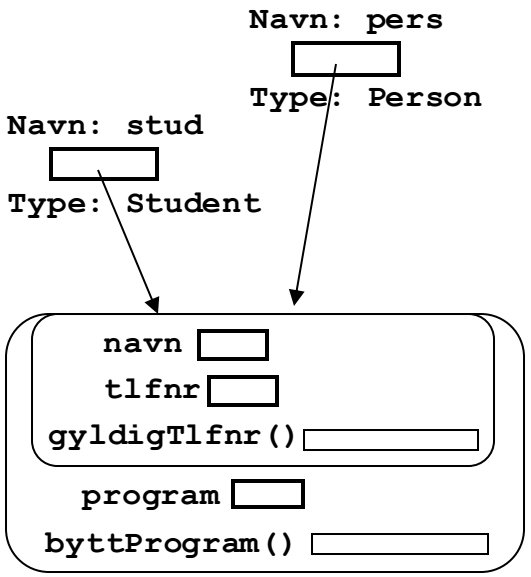
```
Student stud;
Person pers;
stud = new Student();
pers = stud;
```

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

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



Ulike måter å se et objekt på



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

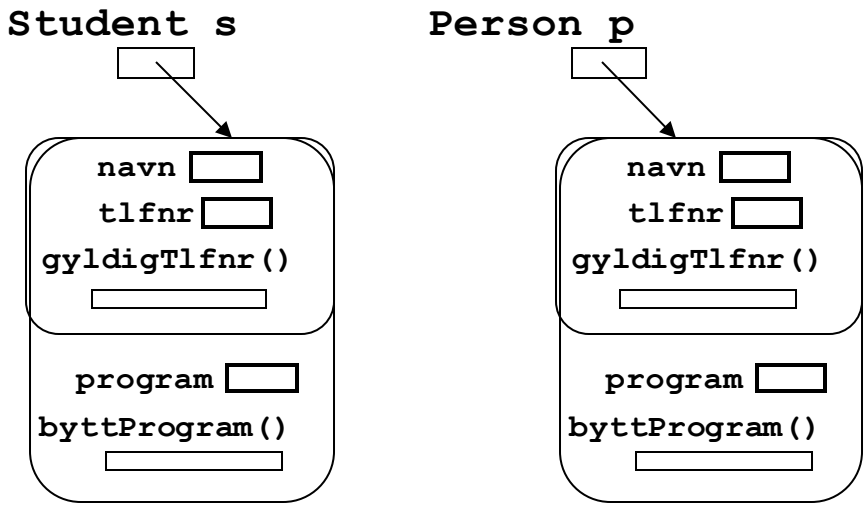
```
Student stud;
Person pers;
stud = new Student();
pers = stud;
```

- **Typen (klassen) til selve objektet er uforanderlig.**
  Et objekt kan likevel ***fremtre*** *for oss* på ulike måter
  Det kan spille forskjellige roller.
  - Når et objekt først er laget kan det aldri utvides eller krympes senere
- 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 unntak av polymorfe metoder, som vi skal lære om neste uke)

Flere 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");

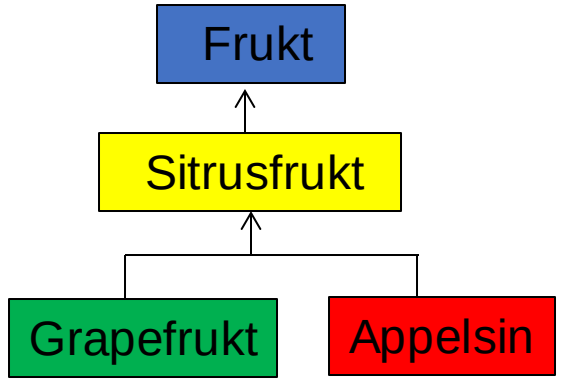

p.navn = ”Ole-Ivar";
. . . p.gyldigTlfnr();
p.program = "Matte";
p.byttProgram("Informatikk");
```

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



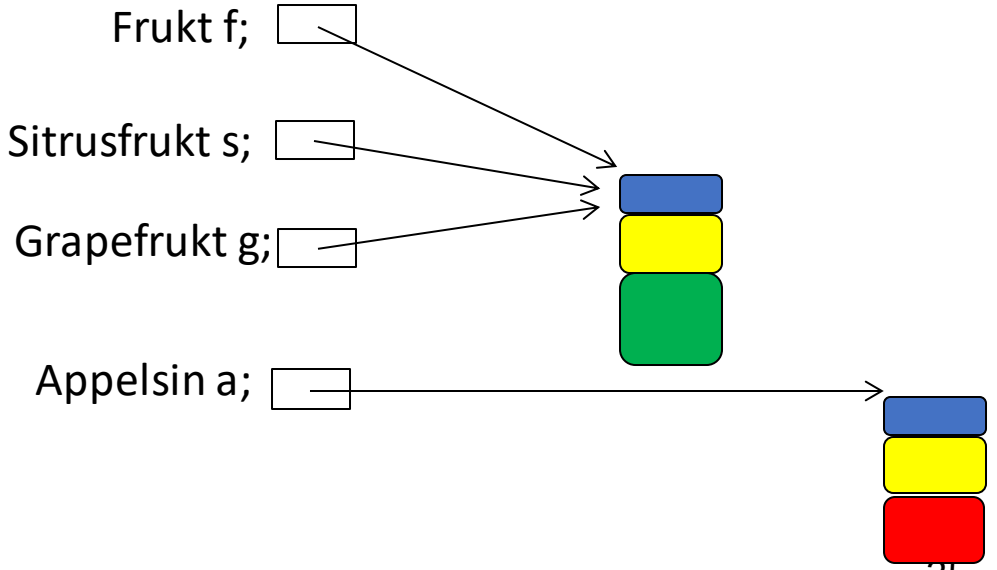
Regel om referanser – Sterk typing


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



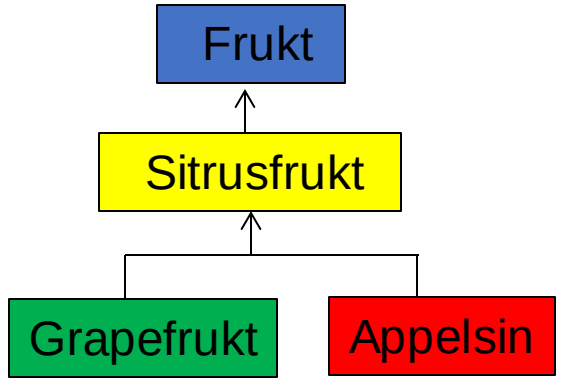
Klassehierarki

En referanse av typen P har bare lov å peke på objekter som har P-egenskaper, og som stammer fra P eller en subklasse av P



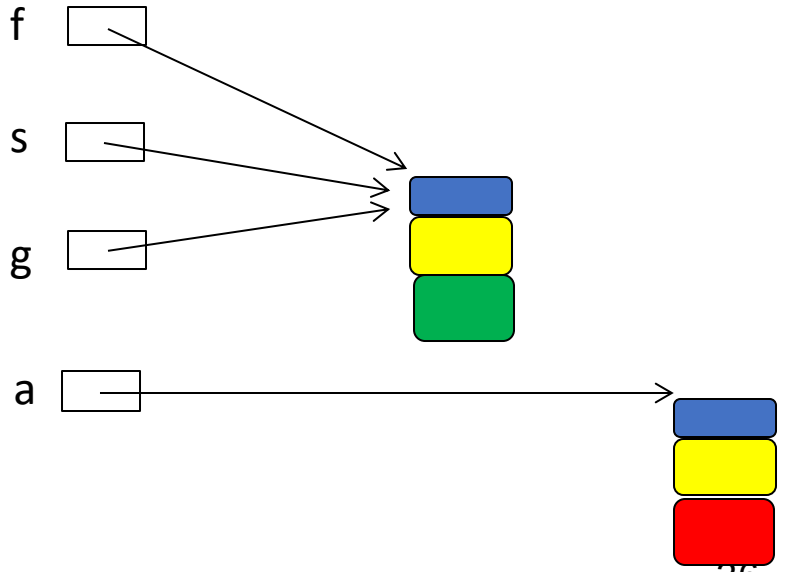
Quiz om referanser til subklasseobjekter


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



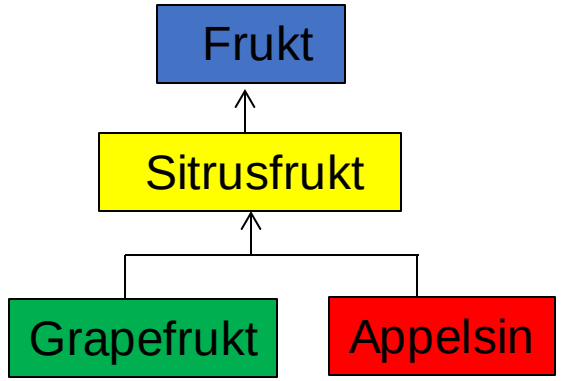
Klassehierarki

```
Frukt f;
Sitrusfrukt s;
Grapefrukt g;
Appelsin a;
// Lovlige tilordninger:
g = new Grapefrukt();
f = g;
s = g;
a = new Appelsin();
```



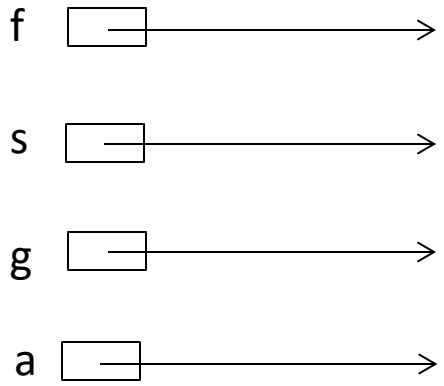
Quiz om referanser til subklasseobjekter


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



Klassehierarki

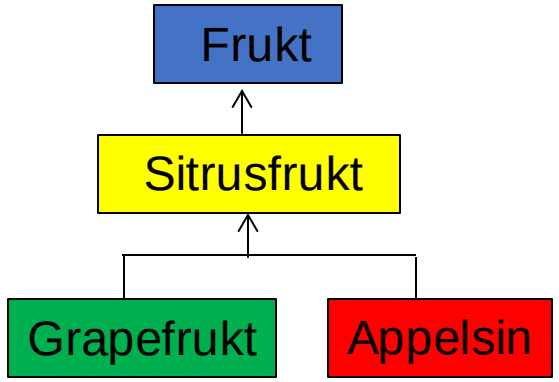
```
Frukt f;
Sitrusfrukt s;
Grapefrukt g;
Appelsin a;
// Hva er lov?
g = ...
f = ...
s = ...
a = ...
```



Menti-quiz

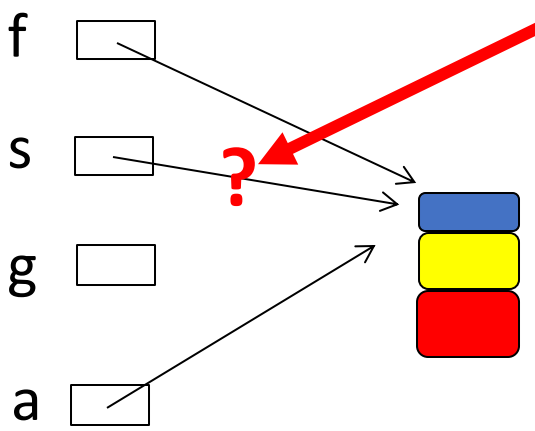
Tilbake til svaret på oppgave 3


```
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



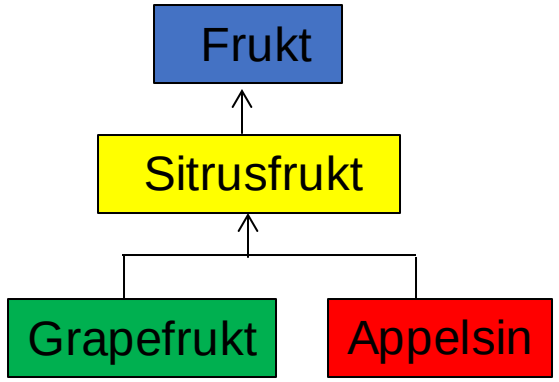
s = f; // Ikke lov

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

Men det vet ikke kompilatoren!

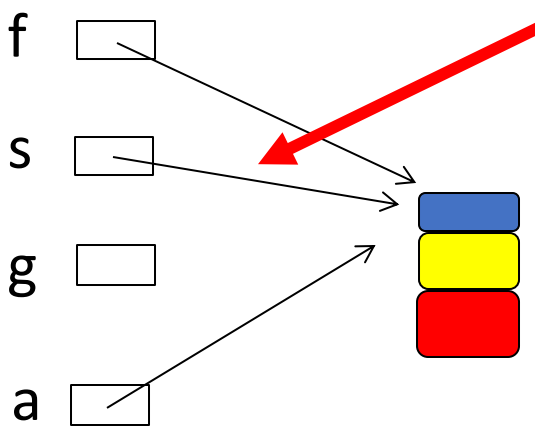
Løsning: La kjøretidsystemet sjekke


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



Klassehierarki

OK etter kjøretidstest

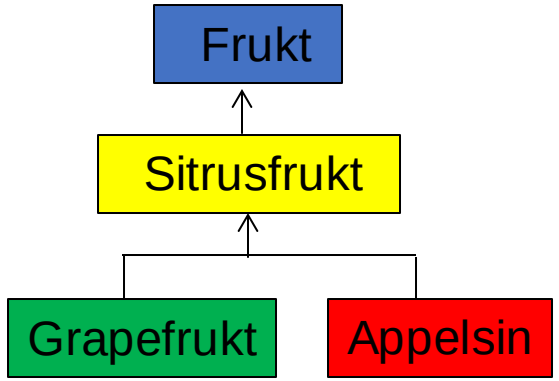


s = f; // Ikke lov
S = (Sitrusfrukt) f; // Er lov

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;

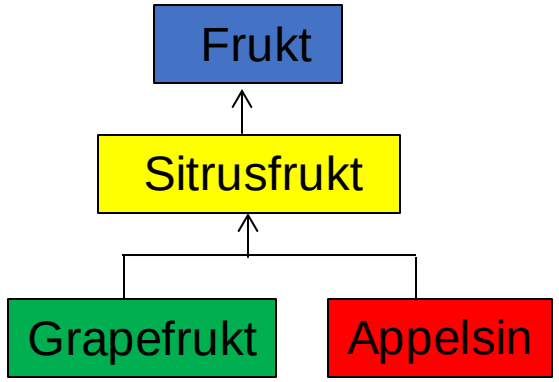
Svaret på uttrykket f er en verdi av typen Frukt.

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

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

Løsning: La kjøretidsystemet sjekke


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

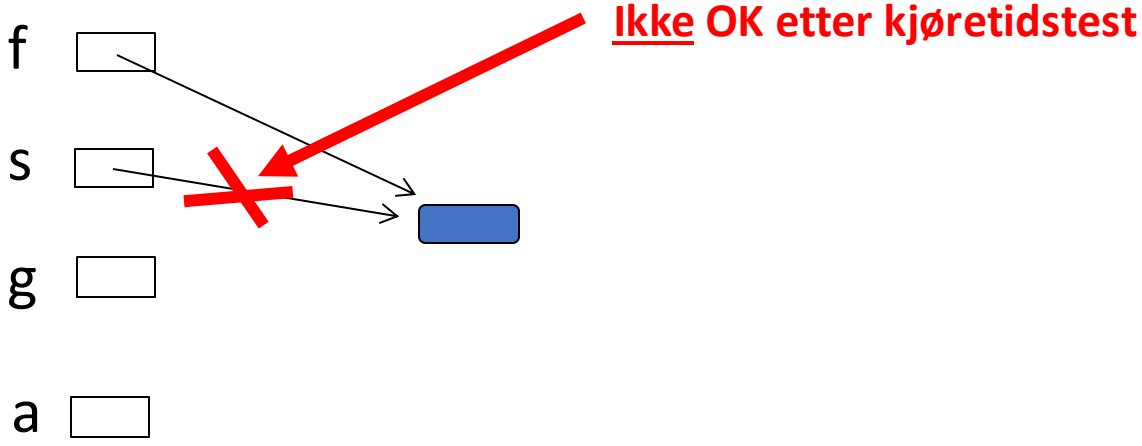


Klassehierarki

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

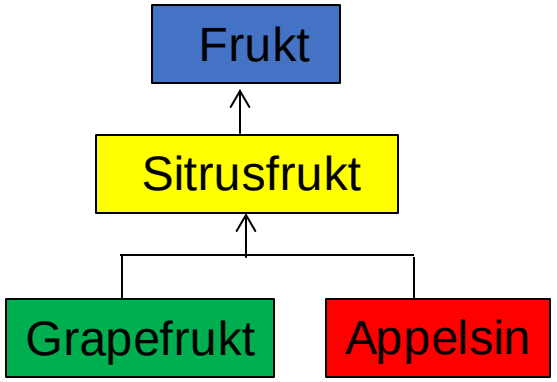
Og her er det ikke OK: Programmet stopper

Resultat: **ClassCastException**



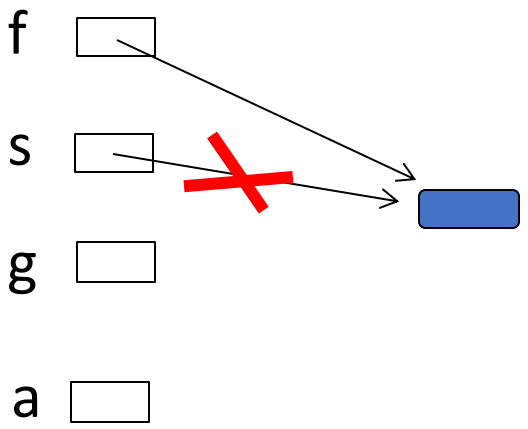
Løsning: instanceof -nytt Java nøkkelord


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



Klassehierarki

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



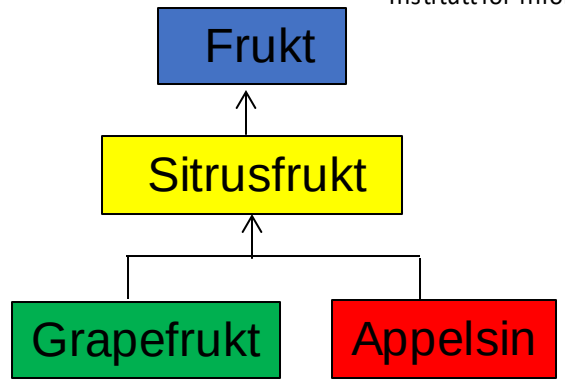
Hva slags objekt er dette?

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

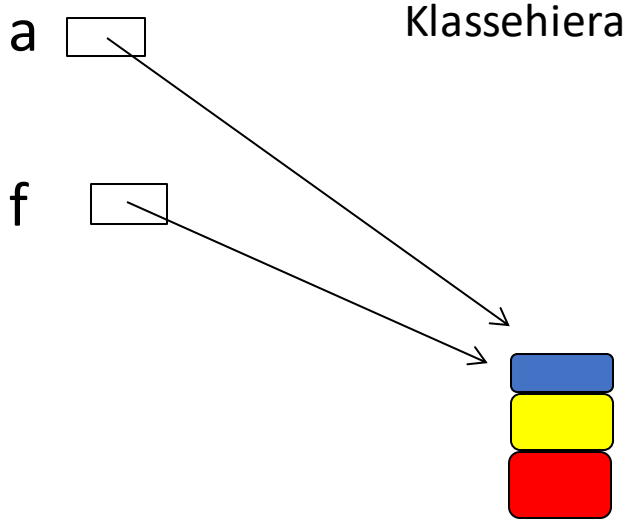
Men hva slags objekt er det virkelig?

```
class TestFrukt {
  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



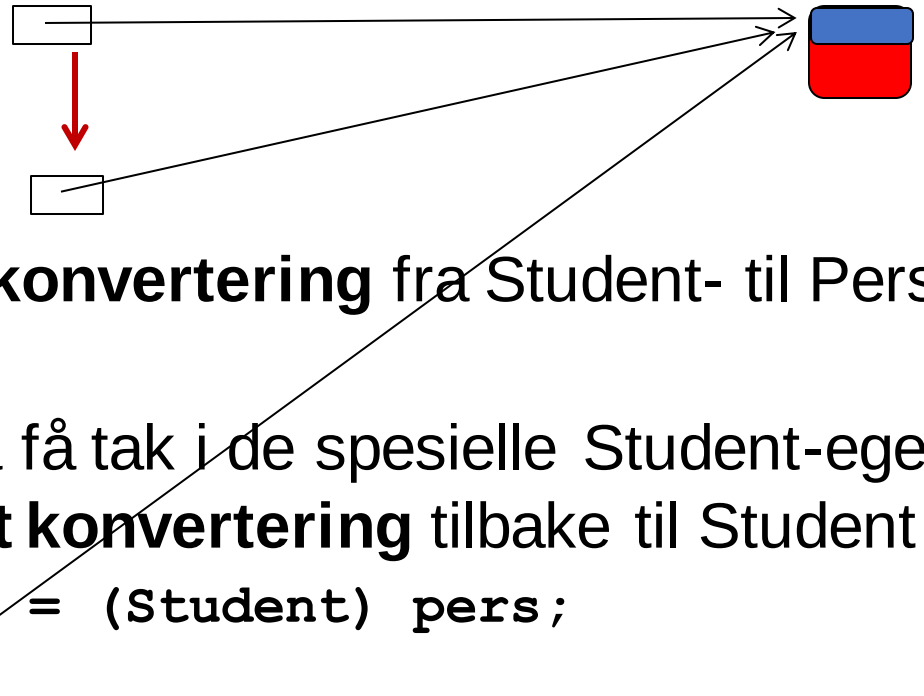
Flere eksempler på konvertering av referanser

- Anta at vi har:

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

- Ved tilordningen

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



har vi en **implisitt 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:

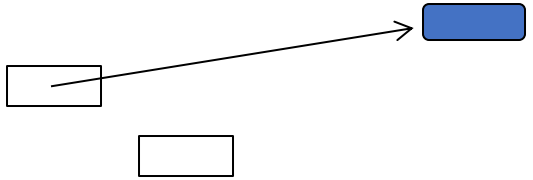
```
Student stud2 = (Student) pers;
```

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

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»)

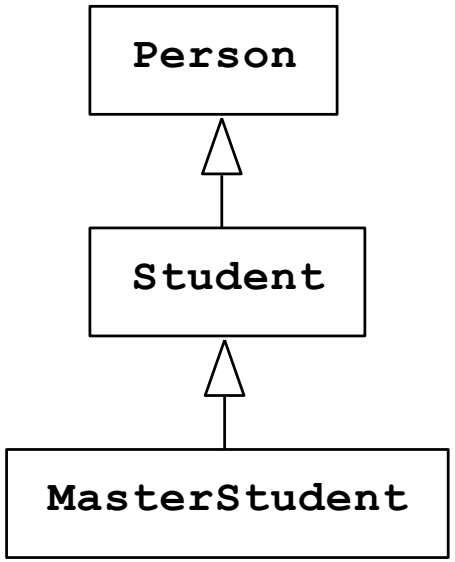
- Igjen: for å unngå denne feilen, bør **instanceof** brukes:

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

`pers instanceof Student` ? **Har objektet som pers peker på alle "Student"-egenskapene, og er det en student?**

Konvertering mellom flere nivåer

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



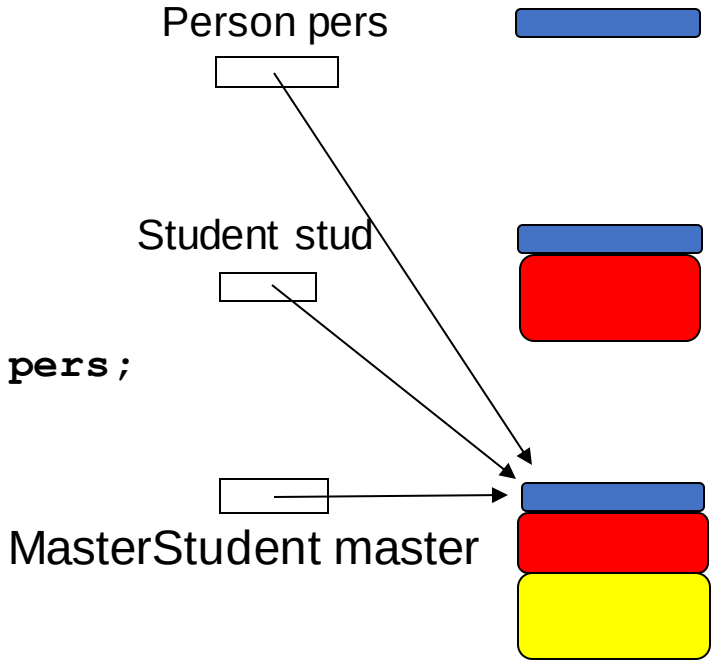
- **Konvertering oppover:**

```
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»)

Modifikatorer 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 { … }
  ```

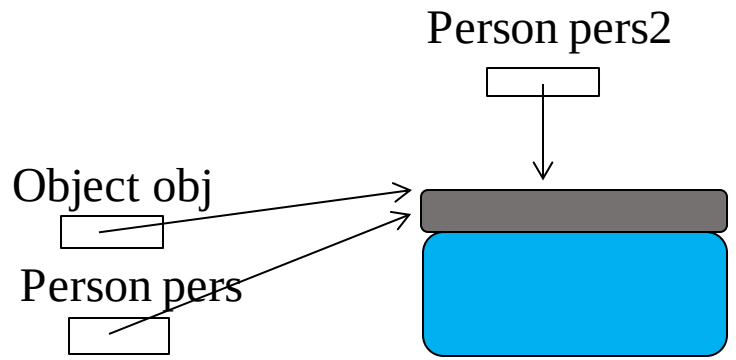
  så tolker Java dette som

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

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

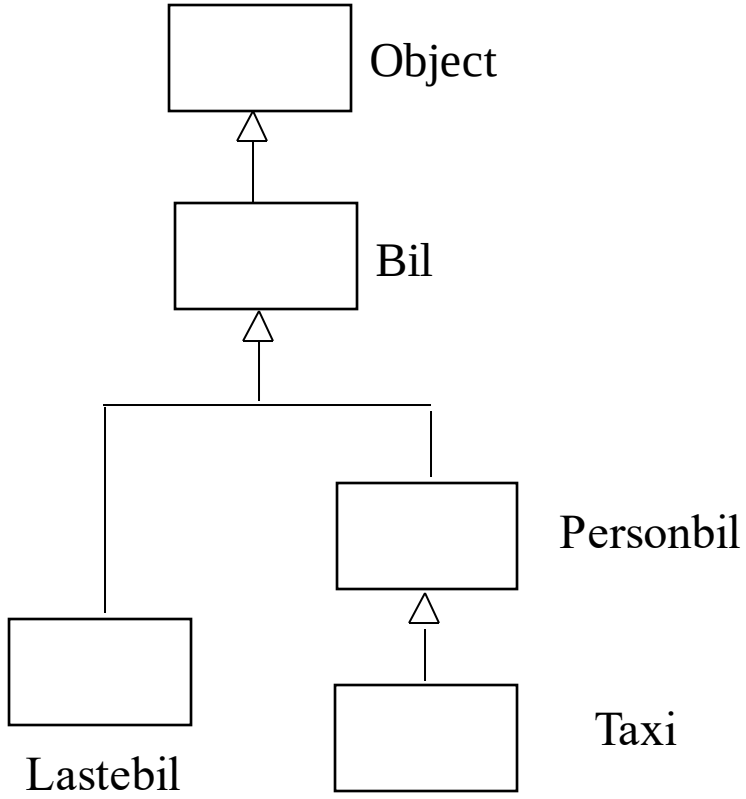
```
Person pers = new Person();
Object obj = pers;
Person pers2 = (Person)obj;
```

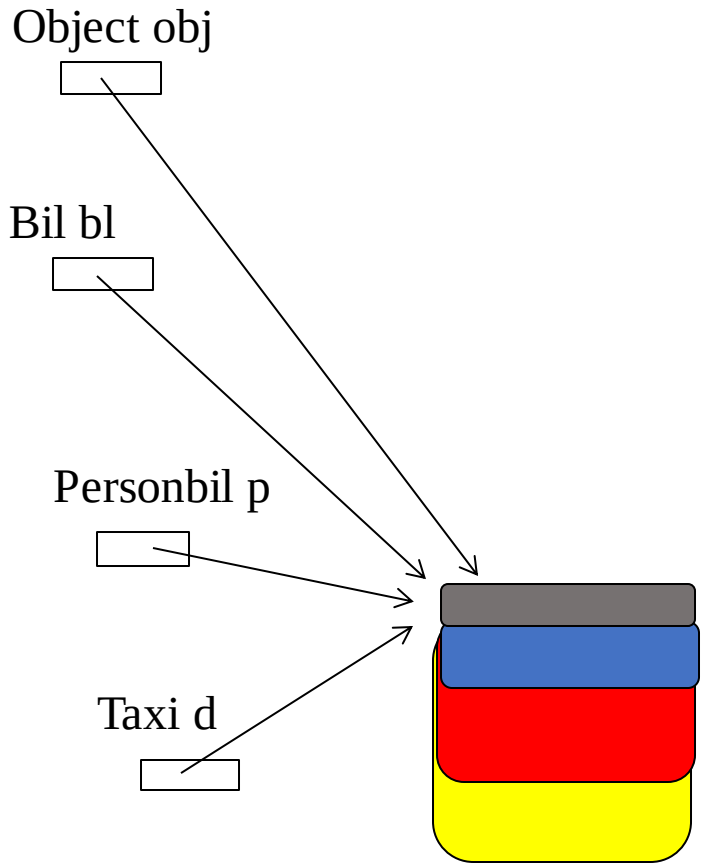
Mer neste gang om hva som er inne i Object-delen av et objekt.



class Object - eksempel

Klassehierarki:





Konstanter i Java

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

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

Helt på siden av dette temaet (men har med skop 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/metoder!

```
class KonstDemoKlasse {
  protected final int objektId;
  KonstDemoKlasse (int objId) {
    objektId = objId;
  }
}
```

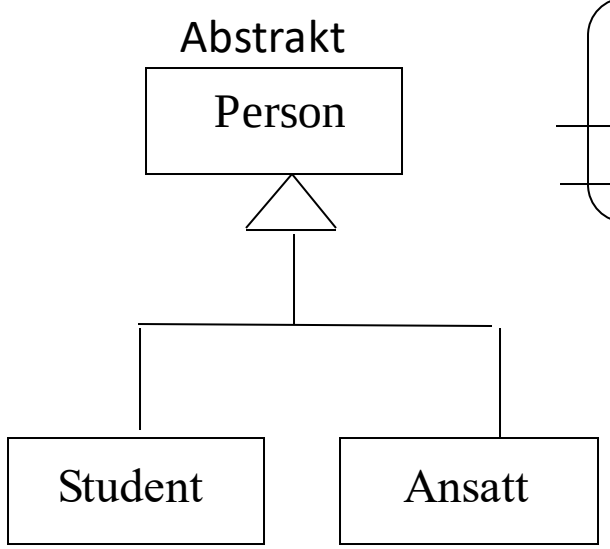
Uten "this" fordi parameter og instansvariabel har forskjellig navn

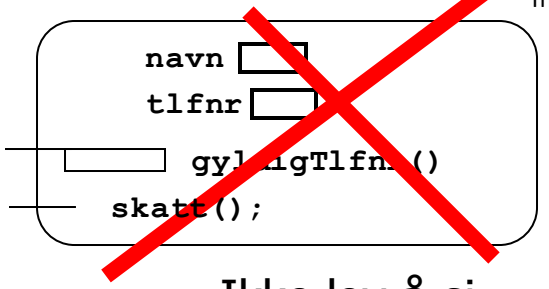
Abstrakte metoder og abstrakte klasser


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

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

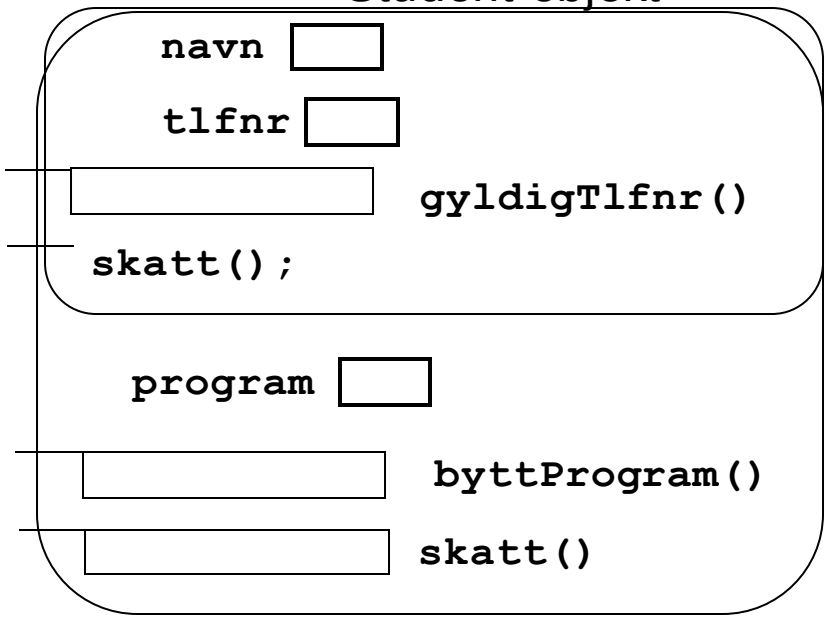
```
class Ansatt extends Person {
  protected int lønnstrinn;
  protected int antallTimer;
  public boolean skatt() {return 100000;}
  public void lønnstillegg(int tillegg) {...}
}
```





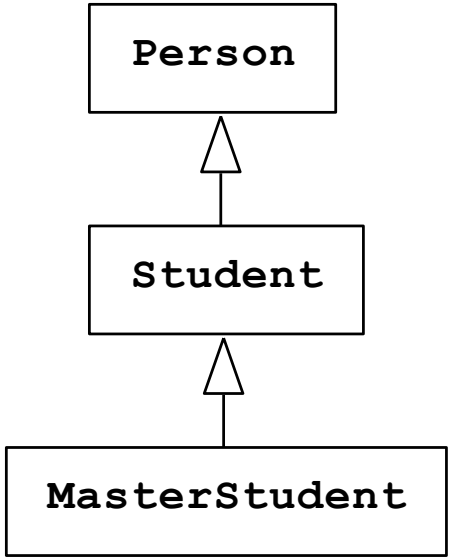
Ikke lov å si
new Person() !

Eksempel på et
Student-objekt



Mer forklaring neste uke

Programmering med subklasser
Hoved-"take away" i dag



Alle studenter er personer.

Alle masterstudenter er studenter og personer.

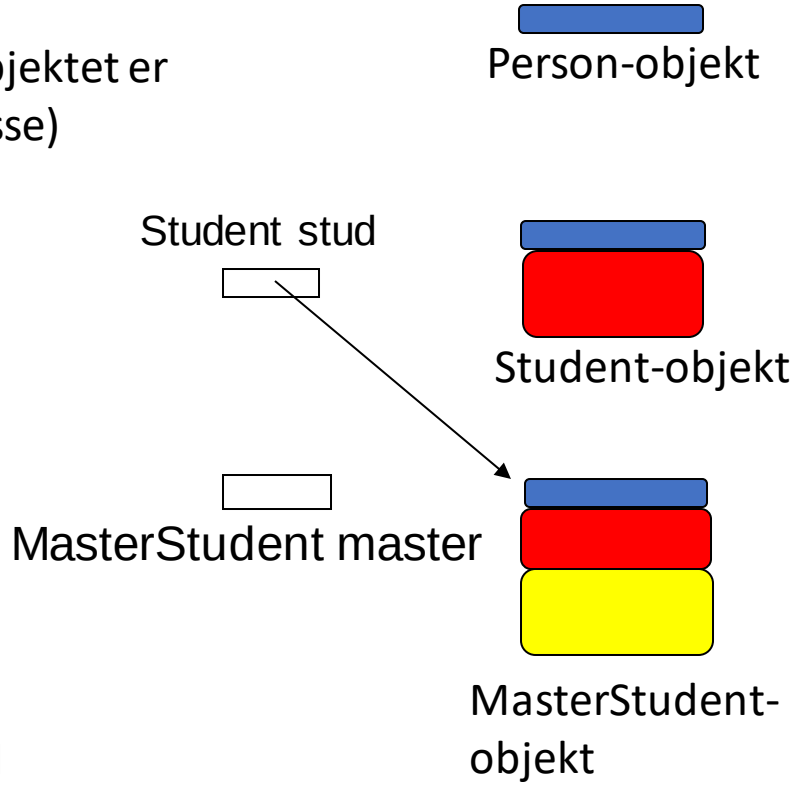
Men: Ikke alle personer er studenter.

En peker har en type (et klassenavn) og den vil bare peke på et objekt som har de samme (eller flere) egenskapene som klassen (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 form av «ClassCastException».



I dag har vi lært

- Subklasser
  - Nøkkelordet extends i Java
- Subklasser - submengder
- Referanser av subklassetyper
- Tilordninger mellom referanser - opp og ned i klassehierarkiet
- Test på objektets type
  - Nøkkelordet instanceof i Java
- Klassen Object
- Abstrakte klasser og metoder
  - Nøkkelordet abstract i Java