

# IN1010 våren 2022

Tirsdag 8. februar

## Arv og subklasser - del 2:

### Polymorfi

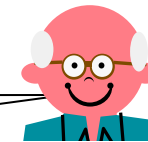
Konstruktører i subklasser

Stein Gjessing

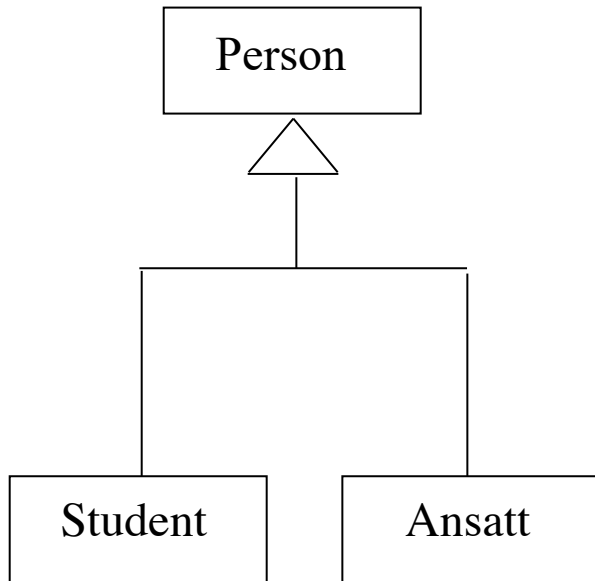
# Dagens tema

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

**"Polymorfi"**  
- fint navn, men det er  
ikke så vanskelig



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

navn   
tlfnr   
gyldigTlfnr()

Person-  
objekt

navn   
tlfnr   
gyldigTlfnr()

program   
byttProgram()

Student-  
objekt

navn   
tlfnr   
gyldigTlfnr()

lønnstrinn   
antallTimer   
lønnstillegg()

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;  
    }  
}
```

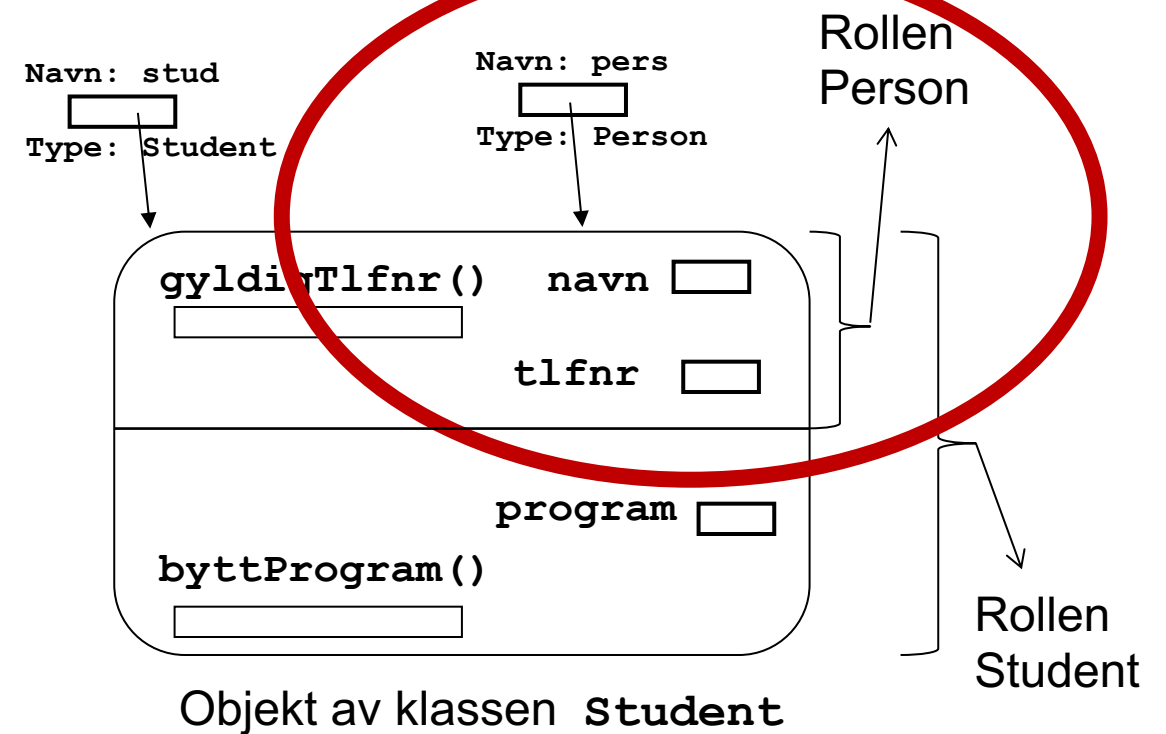
Java er et sterkt typet programmeringsspråk:  
I enden av en **Person pers** - peker vil vi alltid  
se et objekt med **Person**-egenskaper



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

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

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

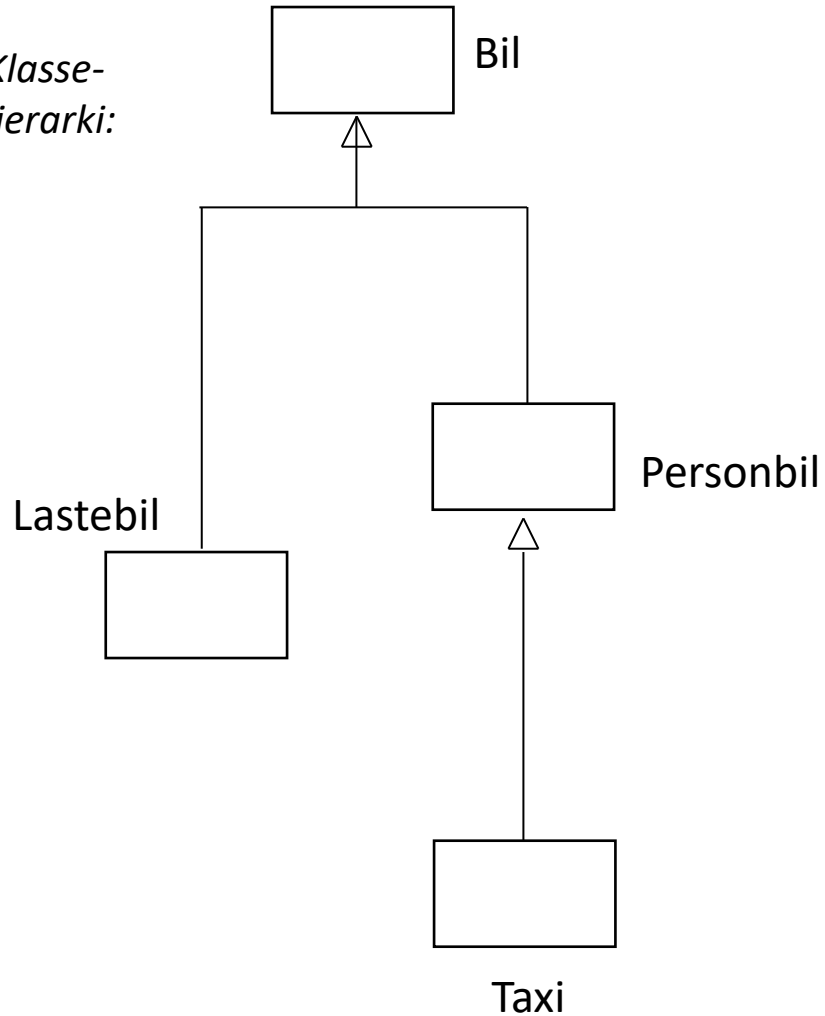


# Polymorfi - motivasjon

NYTT

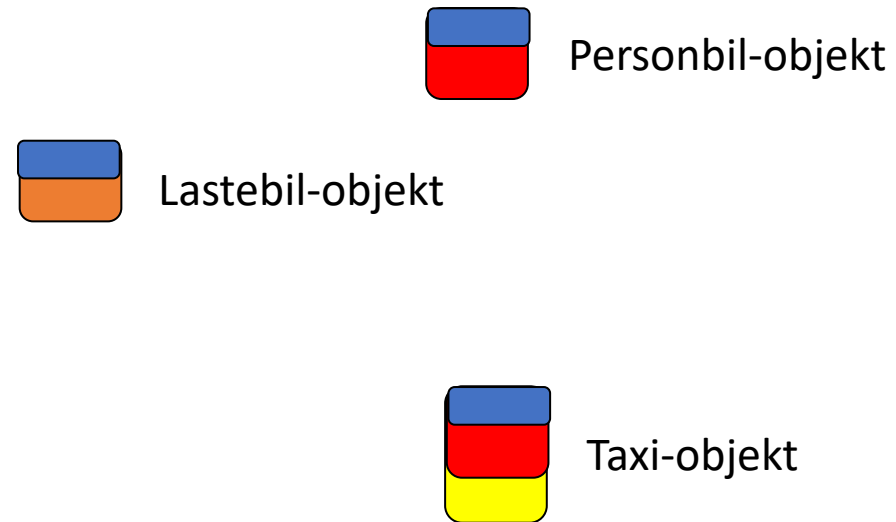
Alle biler skal betale skatt – men ikke like mye

Klasse-  
hierarki:

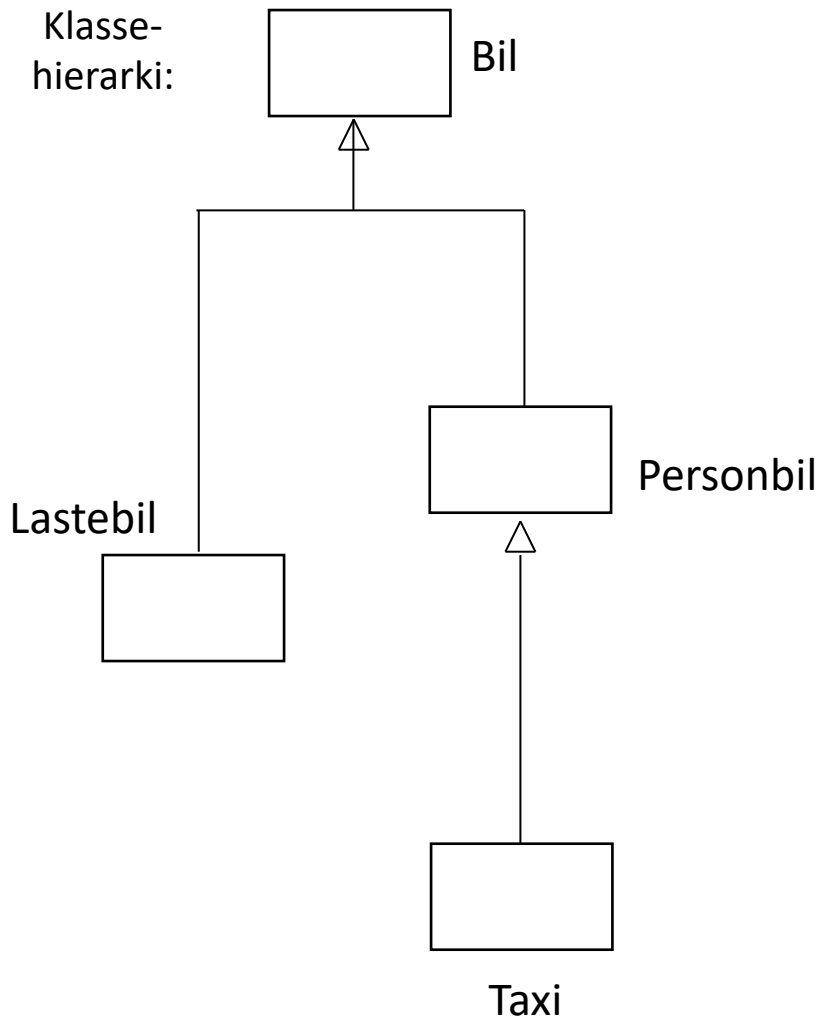


 Bil-objekt

*Eksempler på  
noen objekter*



## 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; }
    . . .
}

```

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

objekt

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

---

protected int antPassasjer   
 public double skatt( ) {return pris ;}

Personbil-objekt

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

---

private double lastevekt   
 public double skatt( ) {return pris\*0.25;}

Lastebil-objekt

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

---

protected int antPassasjer   
 public double skatt( ) {return pris ;}

---

protected String loyveld   
 public double skatt( ) {return pris\*0.1;}

Taxi-objekt



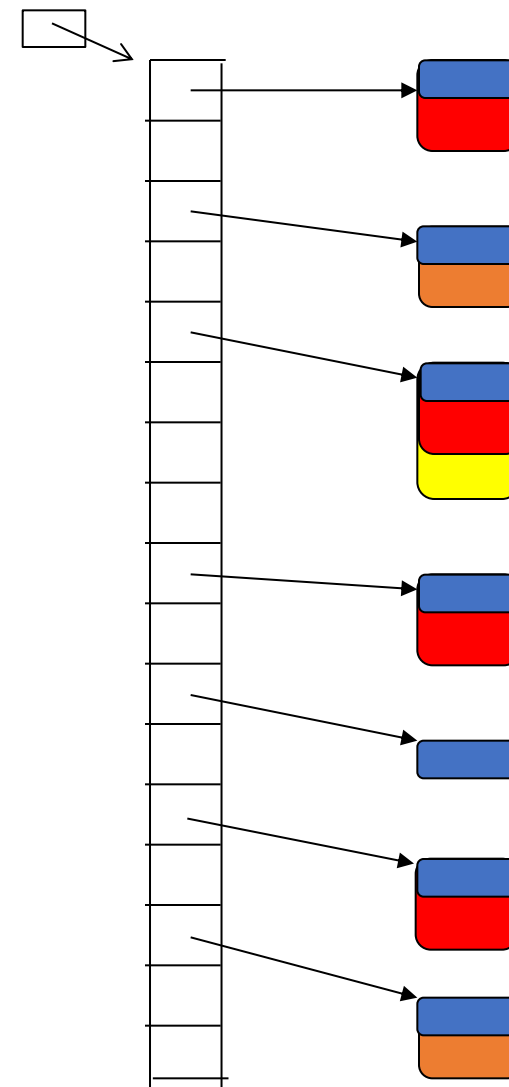
# Polymorfi - motivasjon

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());  
}
```

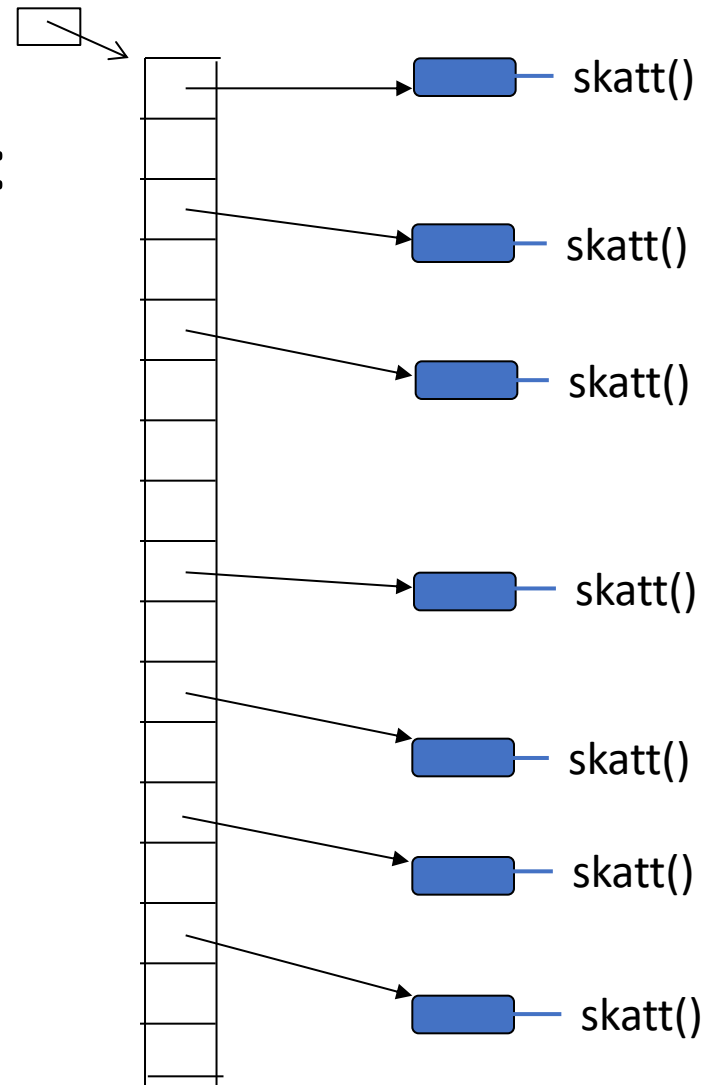
Bil[] bilTab



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());  
}
```

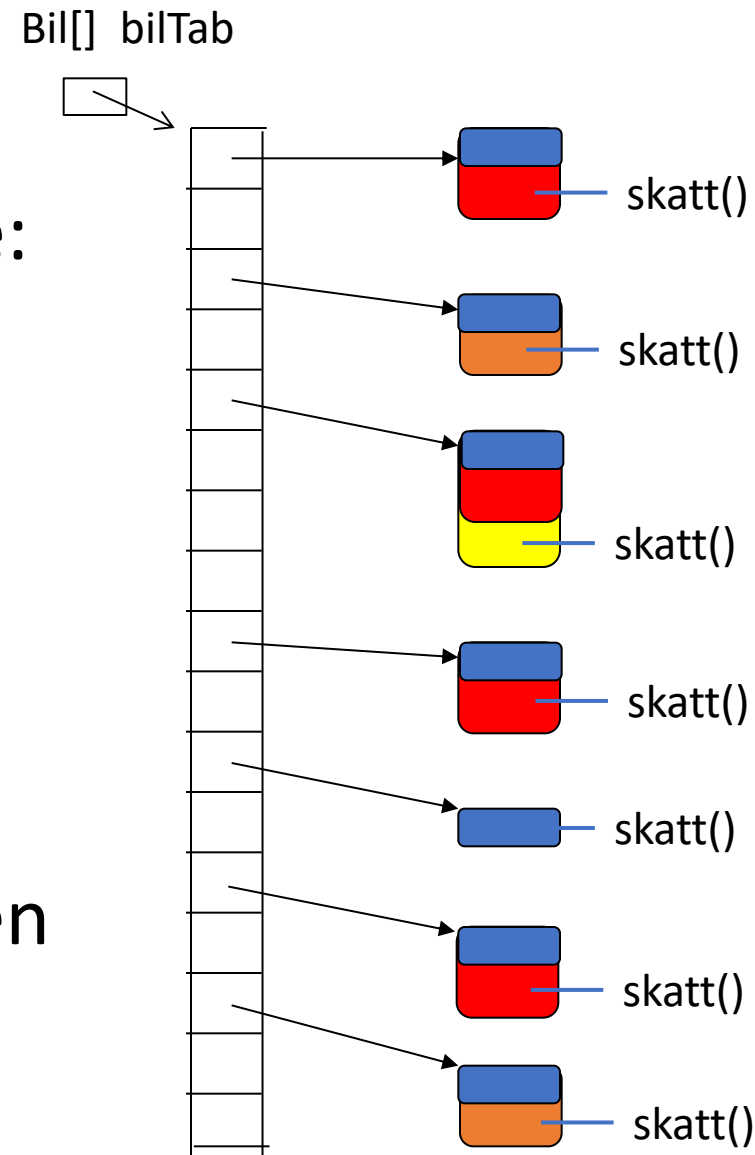
Bil[] bilTab



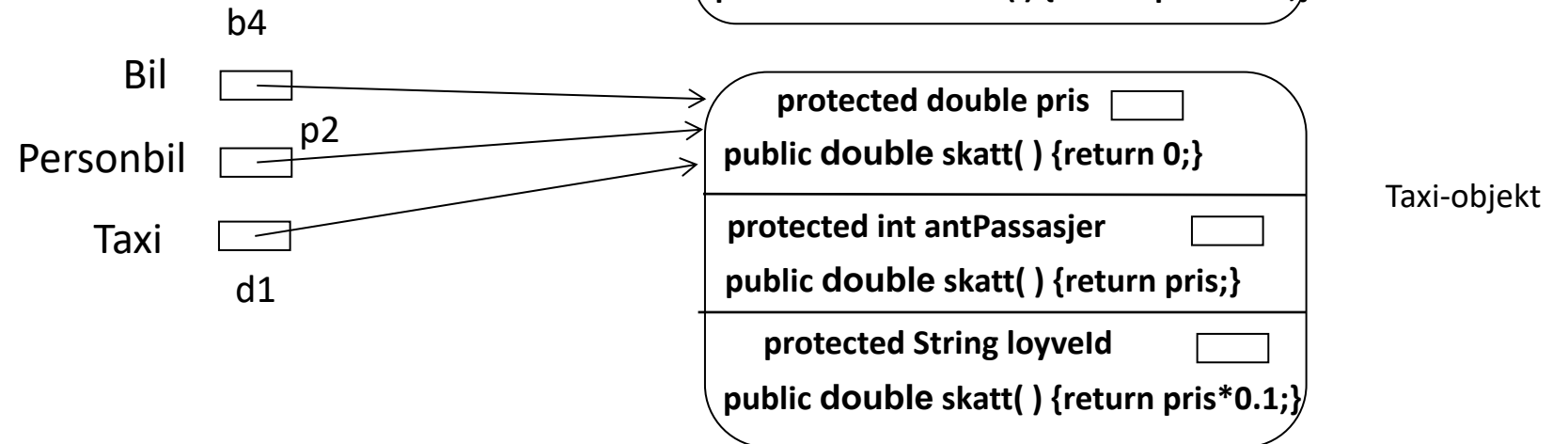
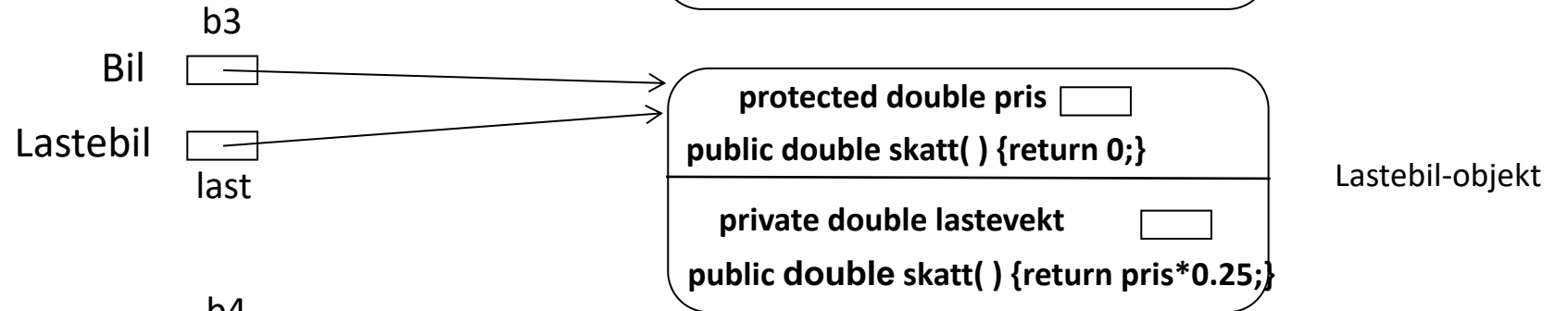
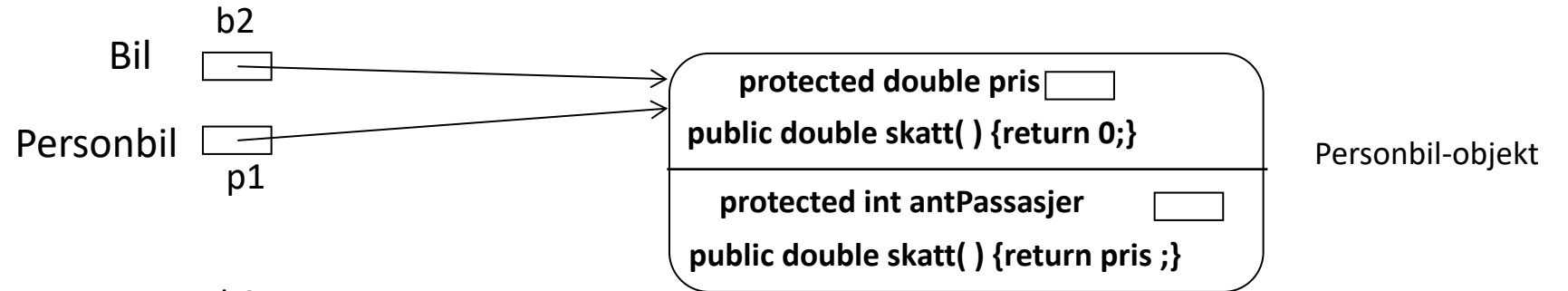
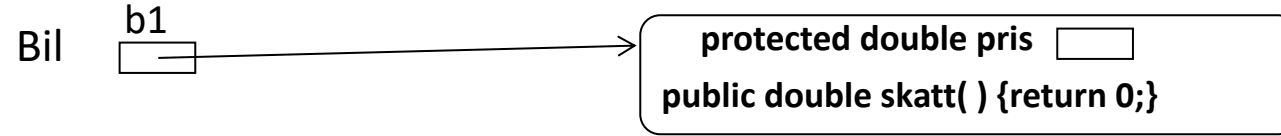
## Polymorfi gjør At kjøretidsystemt gjør dette:

```
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

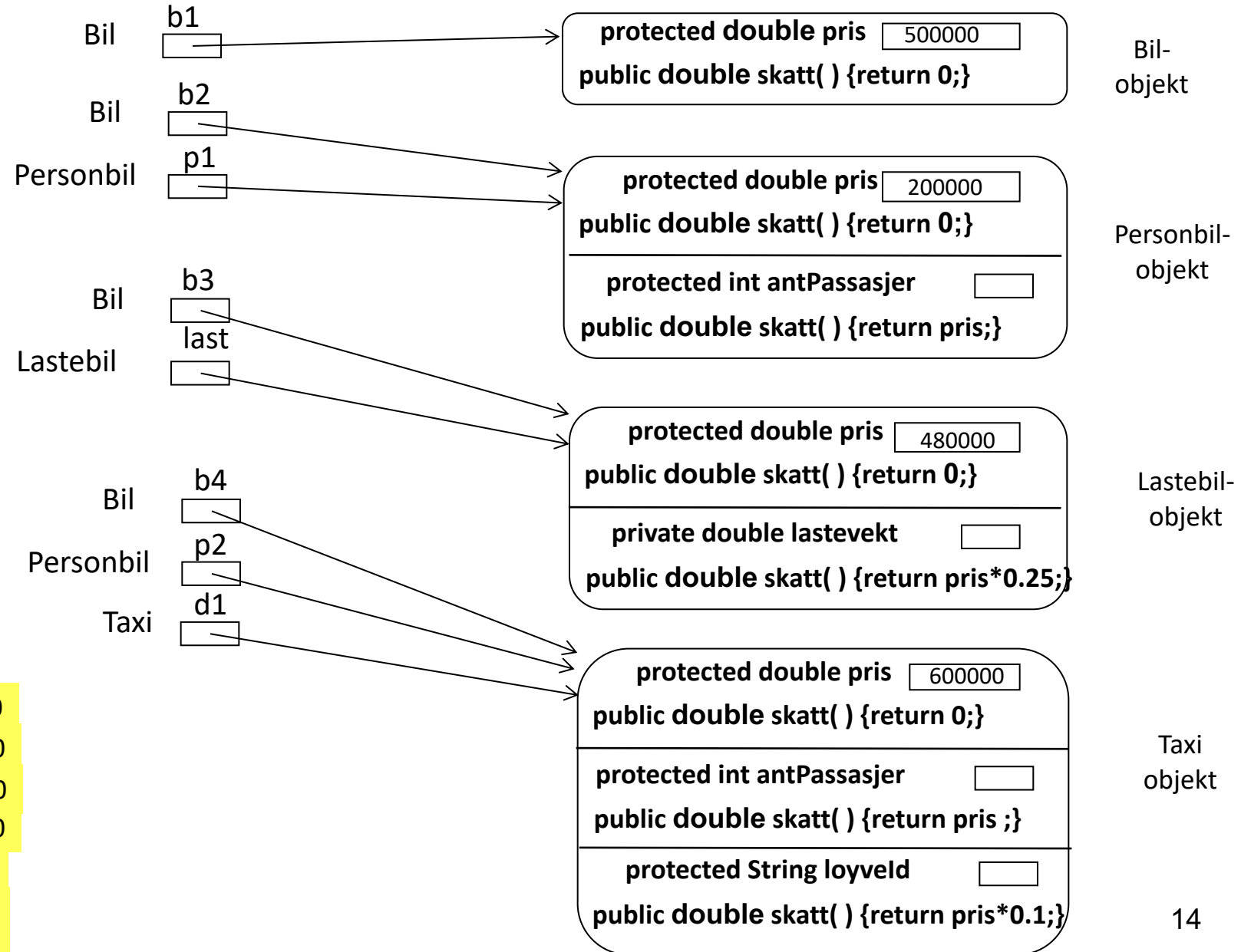
uavhengig av referansetypen som brukes for å komme til objektet



# 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 deklare 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 deklartert 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**



```
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$
```



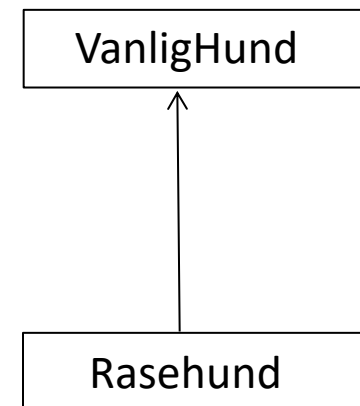
# Polymorfi: Nytt eksempel

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

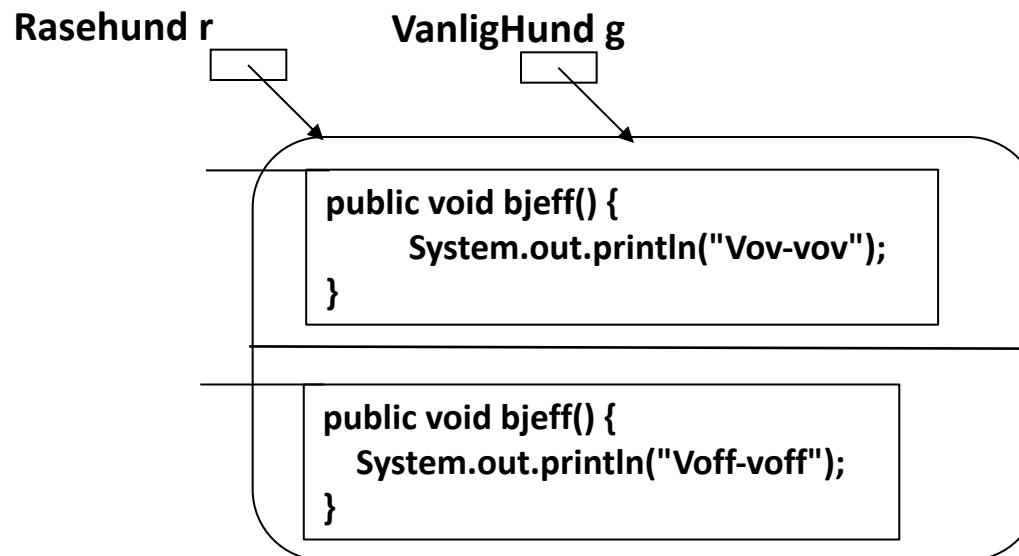
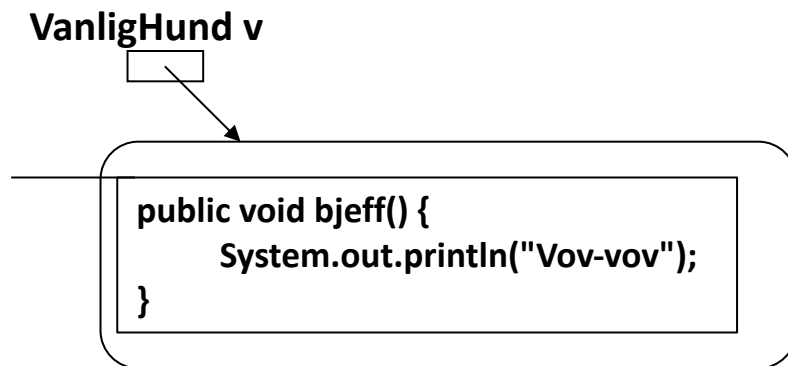
For objekter av typen  
VanligHund er det  
denne metoden som  
gjelder.

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

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

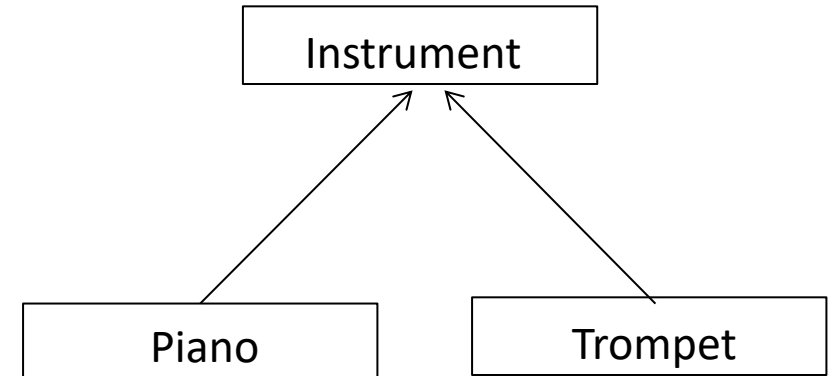
Instrument inst

```
void skrivDefinisjon() {
    System.out.println("Et instr . . .");
}
```

```
void skrivDefinisjon(String overskrift) {
    System.out.println("En trompet . . . ");
}
```

```
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");
    }
}
```

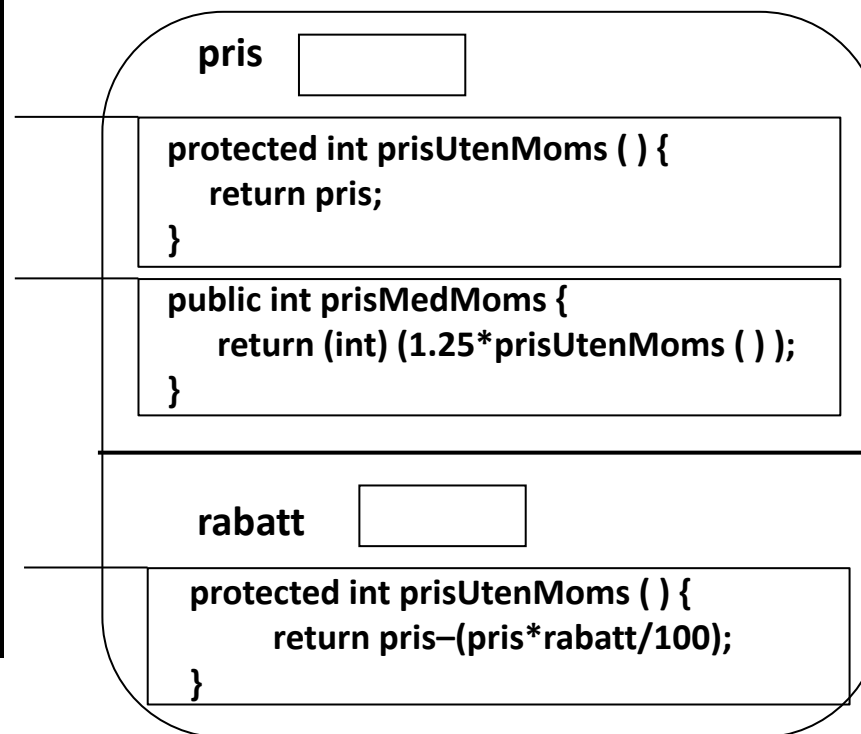
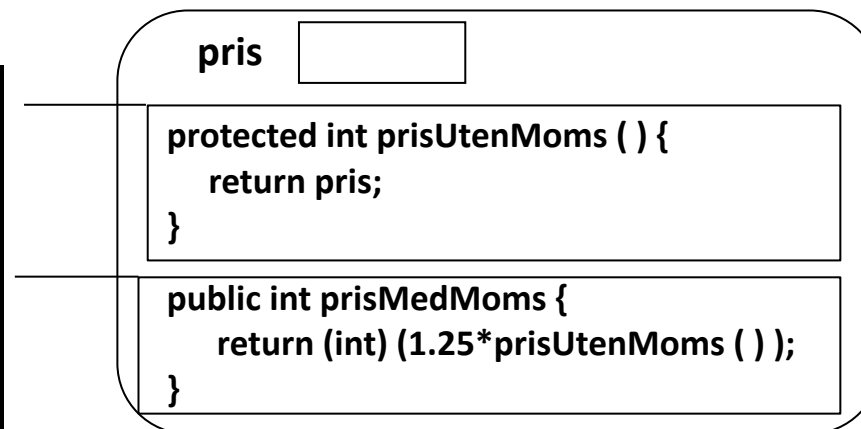
Objekt av  
klassen Trompet



# 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());
    }
}

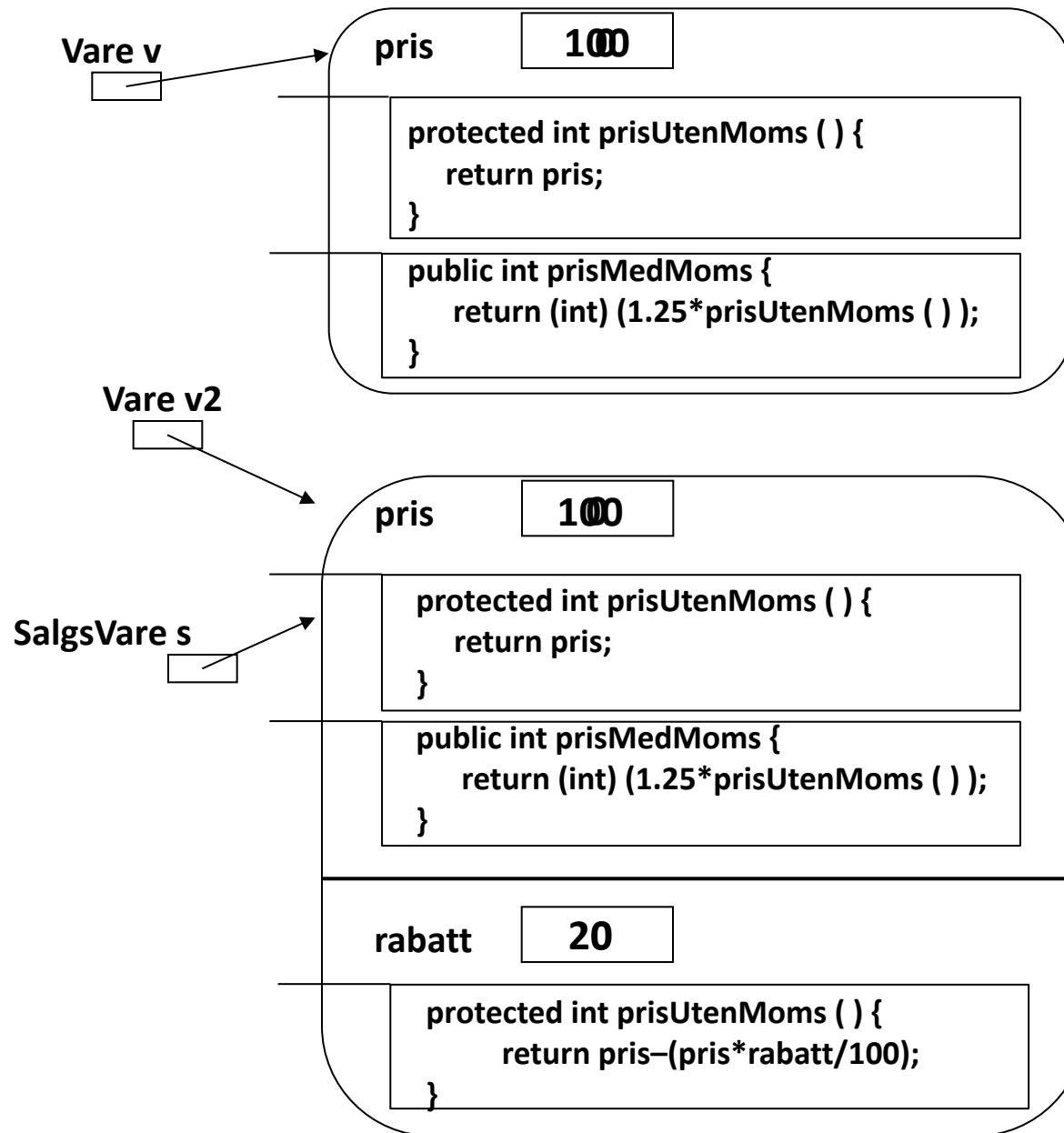
class SalgsVare extends Vare {
    protected int rabatt; // I prosent
    public void setRabatt(int r) {
        rabatt = r;
    }
    @Override
    protected int prisUtenMoms() {
        return (int) pris-(pris*rabatt/100);
    }
}
```



```
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 subclassene:

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

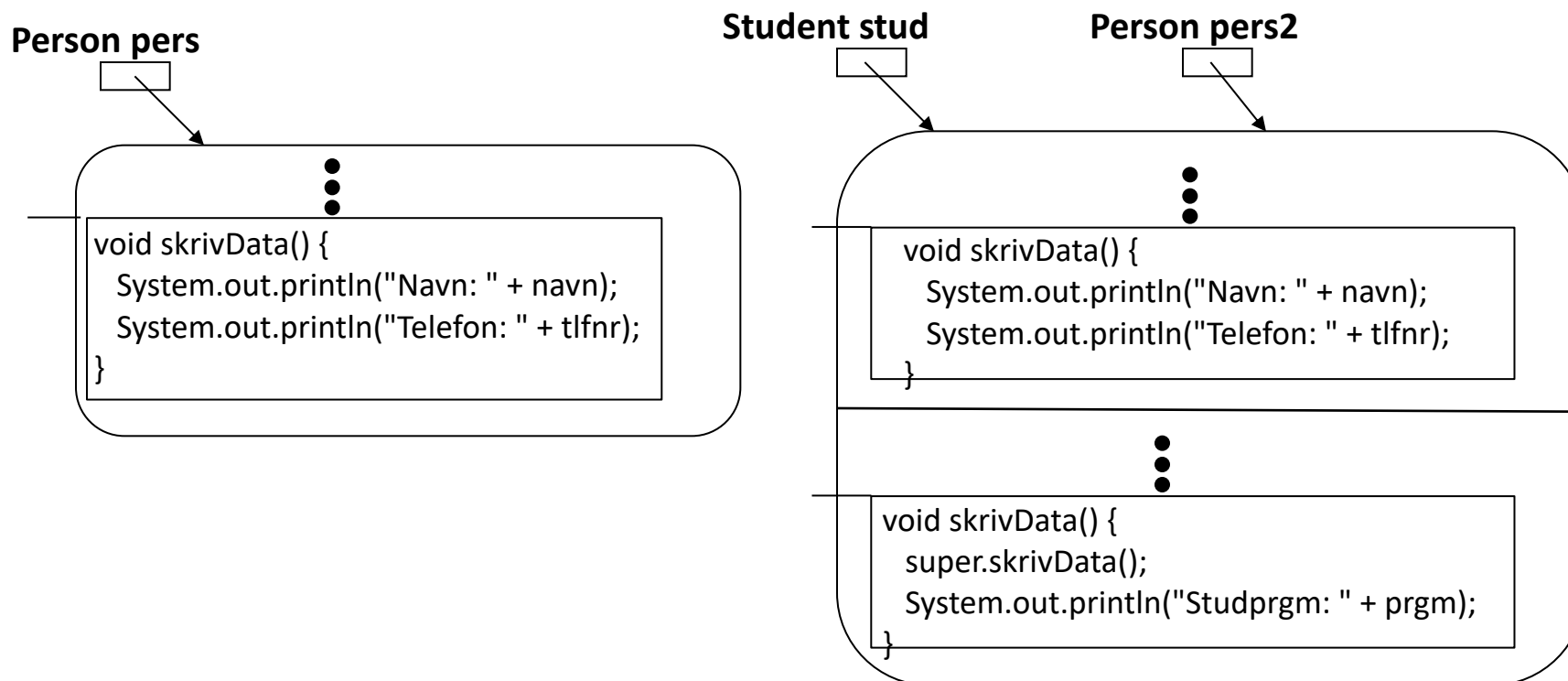
```
// I klassen Student:  
@Override  
public void skrivData() {  
    super.skrivData();  
    System.out.println("Studieprogram: " + program);  
}
```

```
// 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).



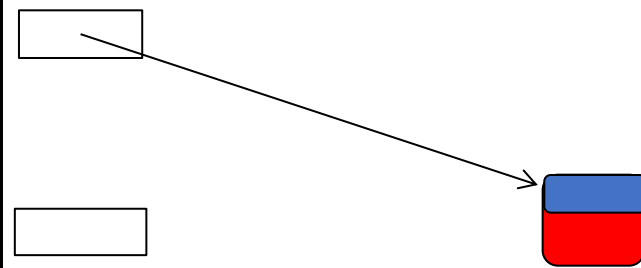
# Omdefinering av variable

- 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...

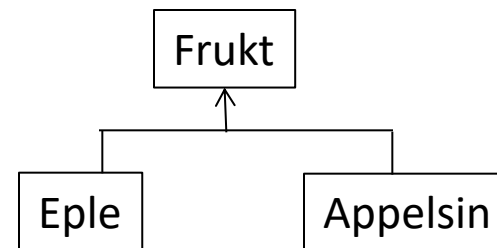
# Husk:

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"



Bruk heller  
polymorfi  
hvis mulig

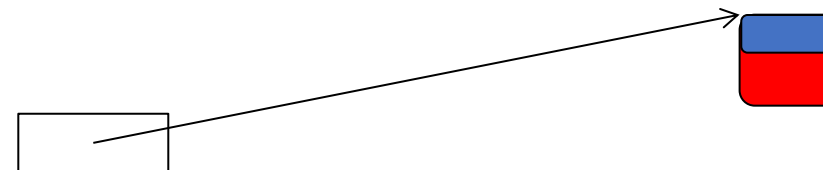
Bedre program:

```
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!");
    }
}
```

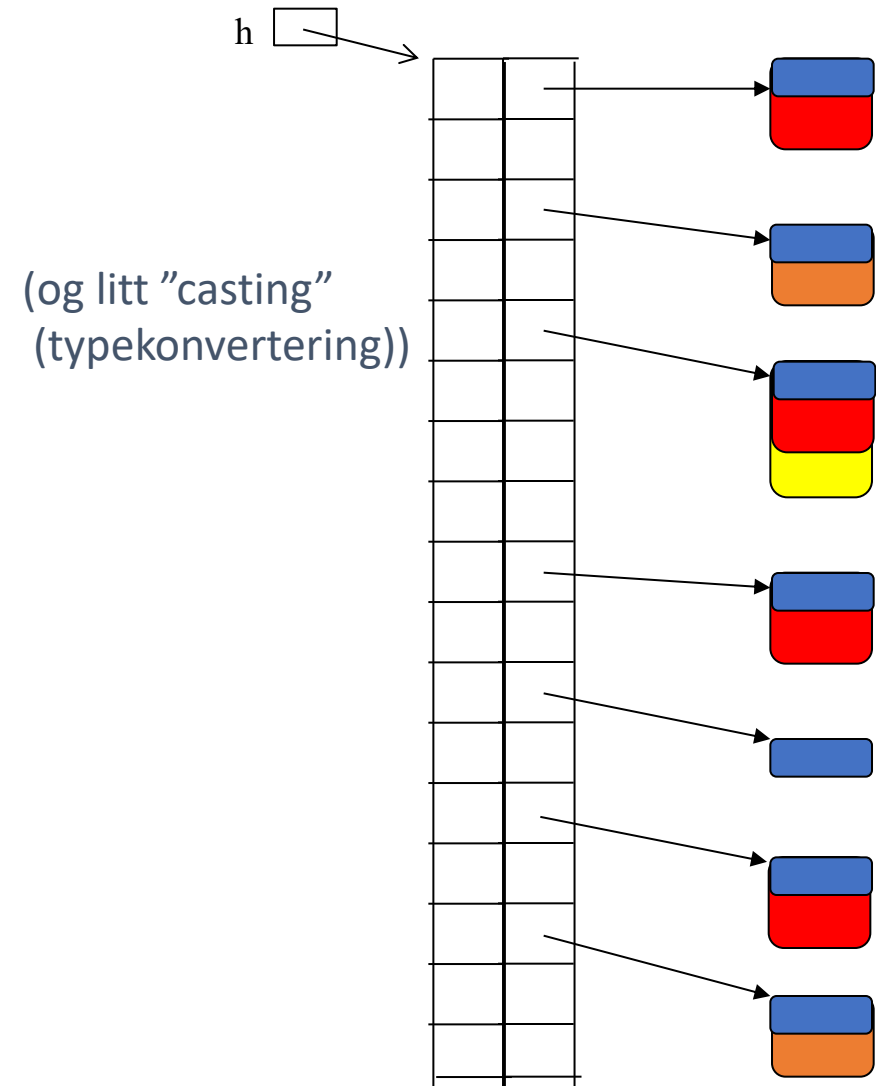


Istedenfor å teste hvilken klasse objektet er av, ber vi objektet gjøre jobben selv. Objektet vet jo best selv hva det er.

```
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;}  
}
```

# Biler og mer bruk av "instanceof" god



# 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 forståelig, og spare arbeid:  
**Gjenbruk av programdeler**
  - "Bottom up" – programmering
    - Lage verktøy
  - "Top down" programmering
    - Postulere verktøy

Nå skal vi se  
litt på 3



# 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 biblioteksklasser)
  - 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

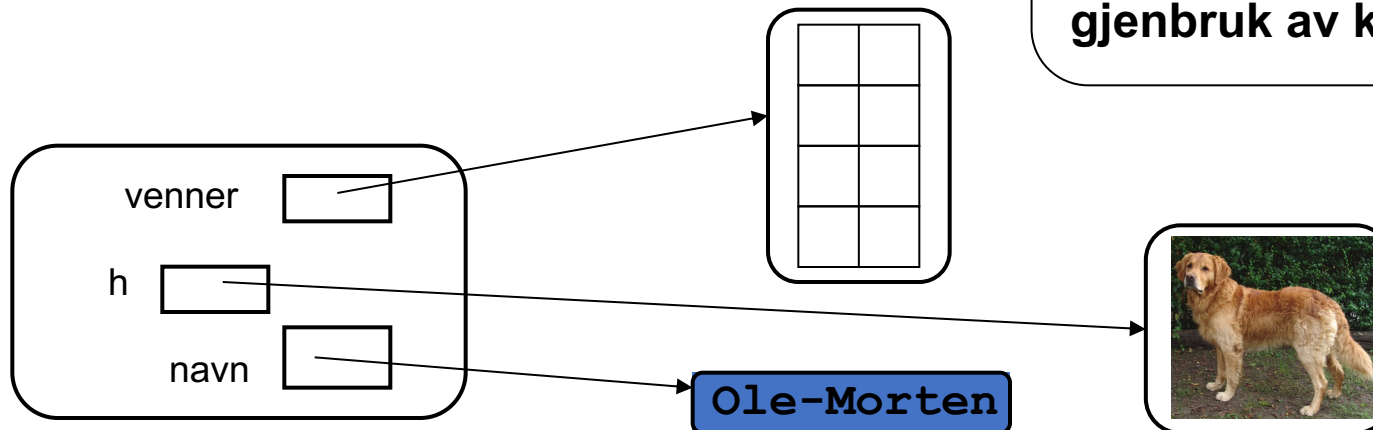


# Gjenbruk ved sammensetning / komposisjon

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 deklartert 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();  
}
```

Arv

Komposisjon

Objekter av  
klassene



← Gjenbrukes



← Gjenbrukes



# 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**



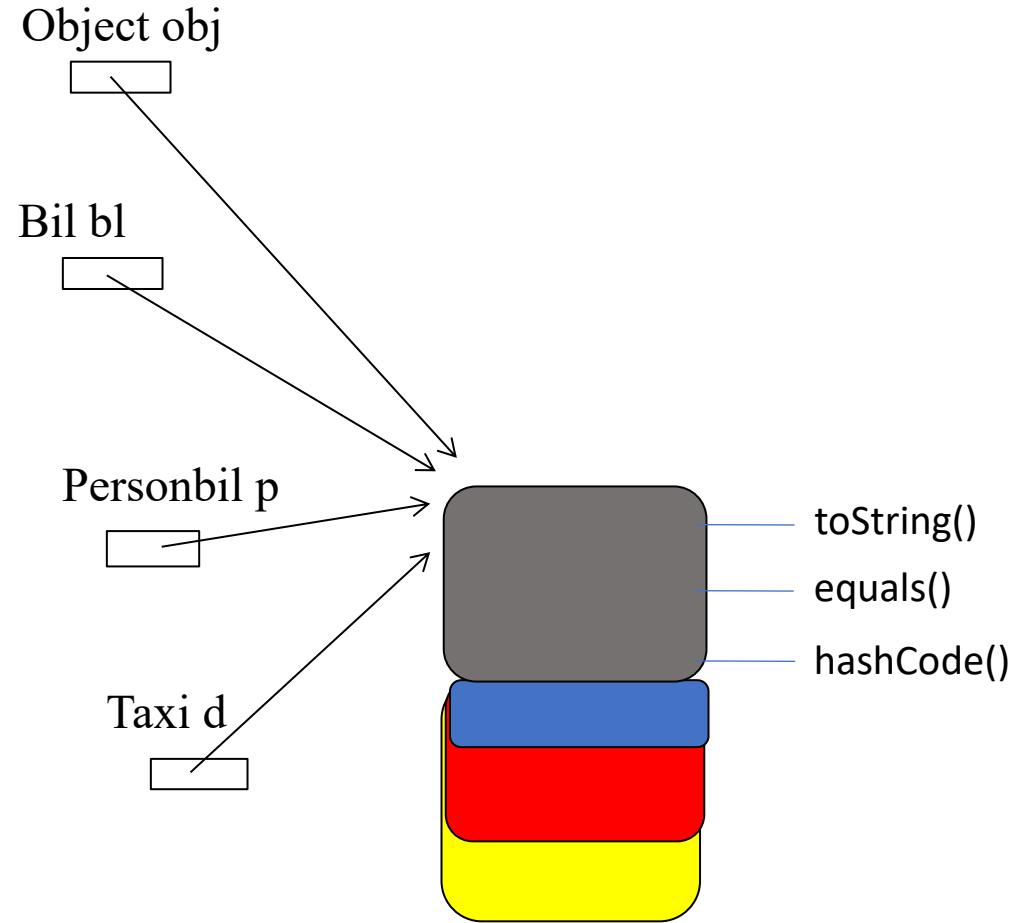
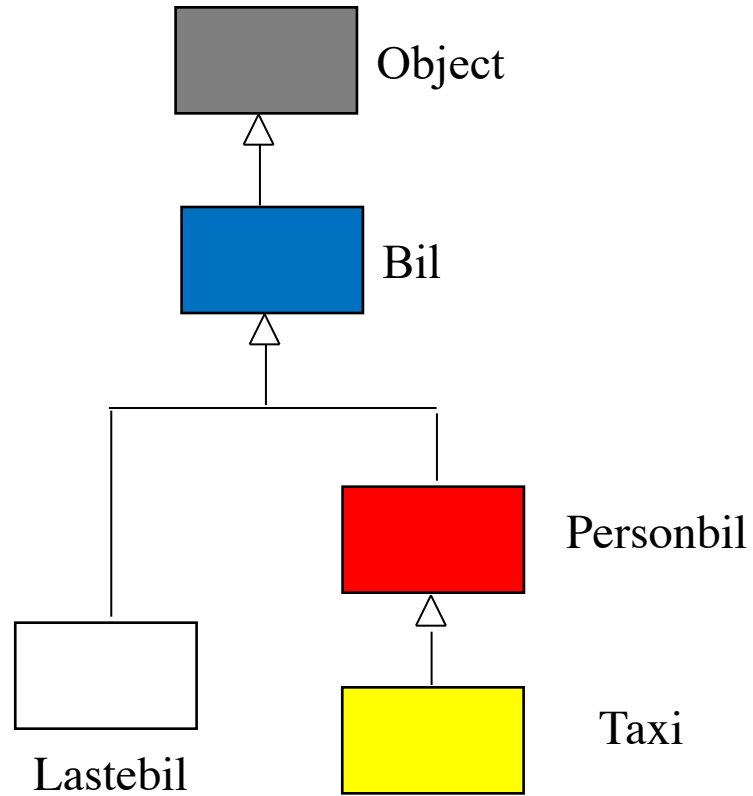


## Menti – oppgave

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 pekerlighet)
  - 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;  
    }  
}
```

# Konstruktører

Ikke noe nytt her

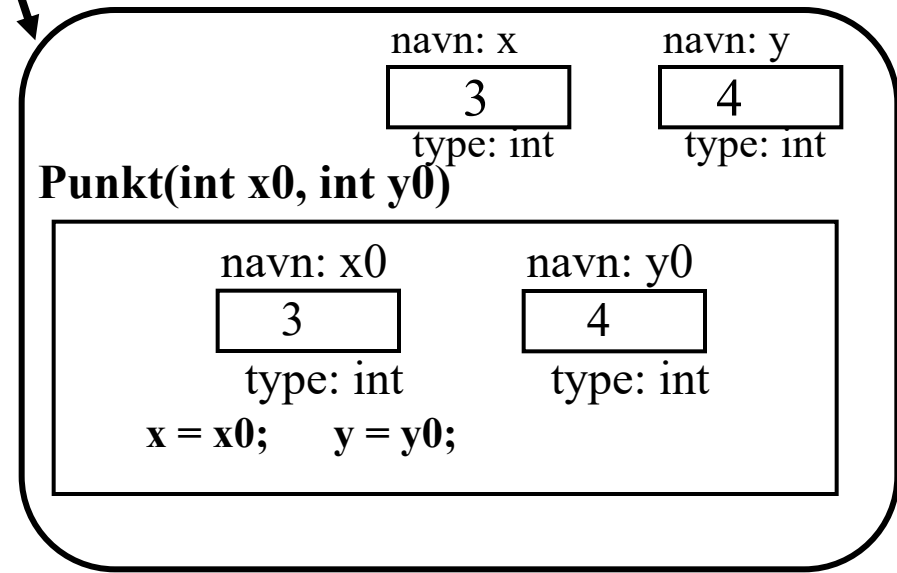
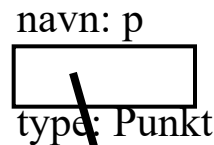
Bruk av konstruktører når vi opererer med "enkle" klasser er ganske ukomplisert. Når vi skriver

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

skjer følgende:

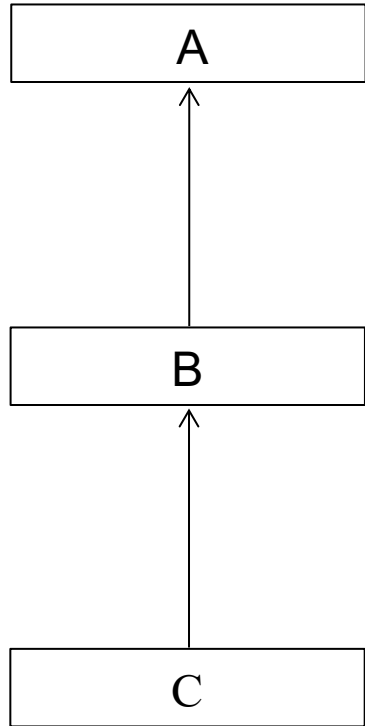
1. Det settes av plass i intern-minnet til et objekt av klassen Punkt og til referansevariablen p.
2. Variablene x og y blir opprettet inne i objektet (instansvariable)
3. Konstruktør-metoden blir kalt med x0=3 og y0=4.
4. Etter at konstruktøren har satt x=3 og y=4, blir verdien av høyresiden i tilordningen  
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.

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



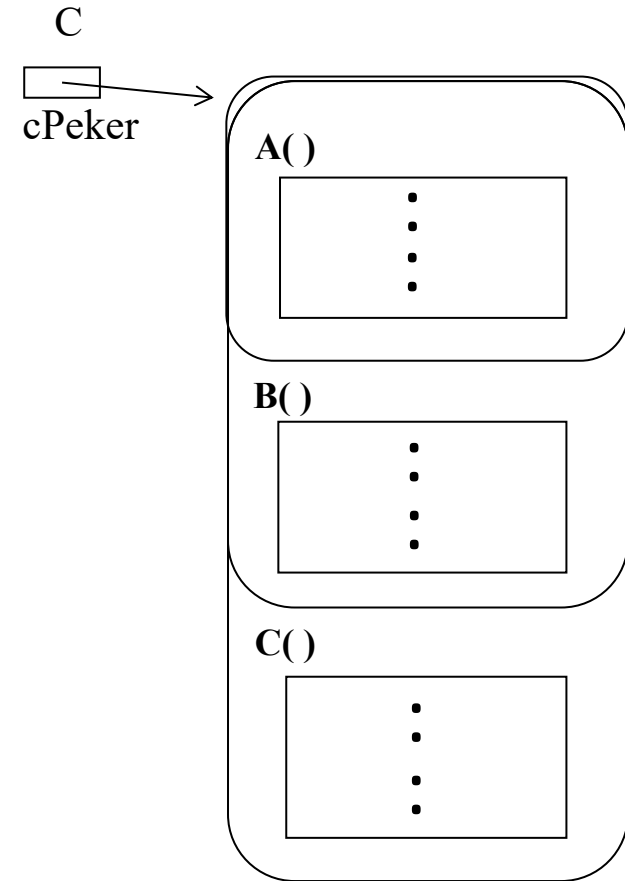


# Konstruktører blir noe mer komplisert når vi opererer med arv



- Anta at vi har deklartert tre klasser:  
`class A { ... }`  
`class B extends A { ... }`  
`class C extends B { ... }`
- Hvilken / hvilke konstruktør utføres hvis vi skriver

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



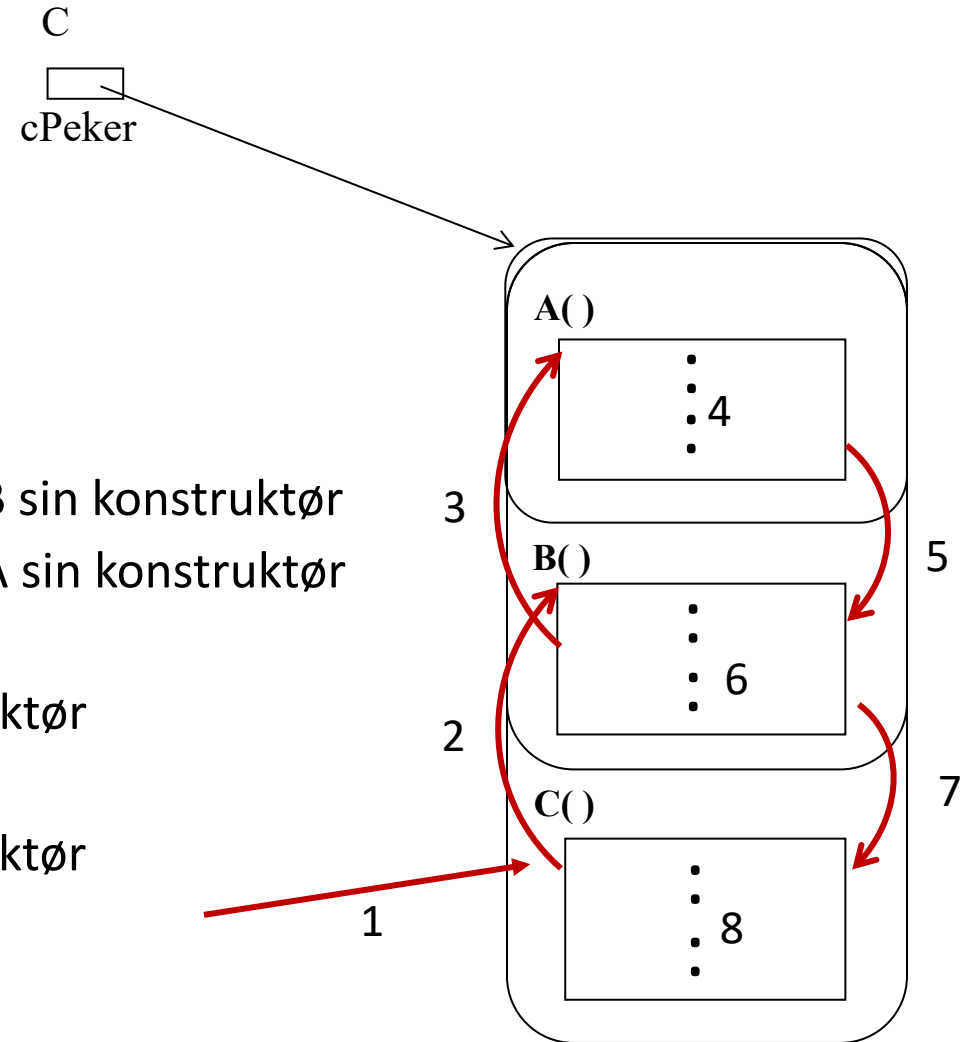
# Konstruktører og arv virker slik:

- Anta at vi har deklartert 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 utføres B sin konstruktør
7. Kontrollen kommer tilbake til C sin konstruktør
8. Så utføres C sin konstruktør

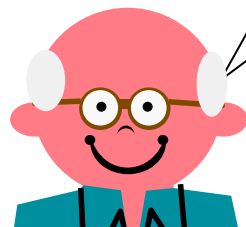


# Kall på superklassens-konstruktør

- 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 {  
    protected 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

Her er noen forslag  
til konstruktører:

Anta at vi har følgende klasser:

```
class Person {  
    protected String fødselsnr;  
  
    Person(String fnr) {  
        fødselsnr = fnr;  
    }  
}  
  
class Student extends Person {  
    protected int studID;  
  
    Student() {...}  
}
```

```
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();  
}
```

4.

5.

3.

2.

Til Object sin konstruktør

Her starter eksekveringen

1.

# Når programmet utføres (forts.)

Tilbake fra Object  
sin konstruktør

7.

Nå er  
**Bygning**  
skrevet ut

8.

Nå er  
**Bolighus**  
skrevet ut

9.

Nå er **Blokk**  
skrevet ut

```
class Bygning {
    Bygning() {
        super( ); ← 6.
        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();
}
```

# 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 konstruktøren 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();
    }
}
```

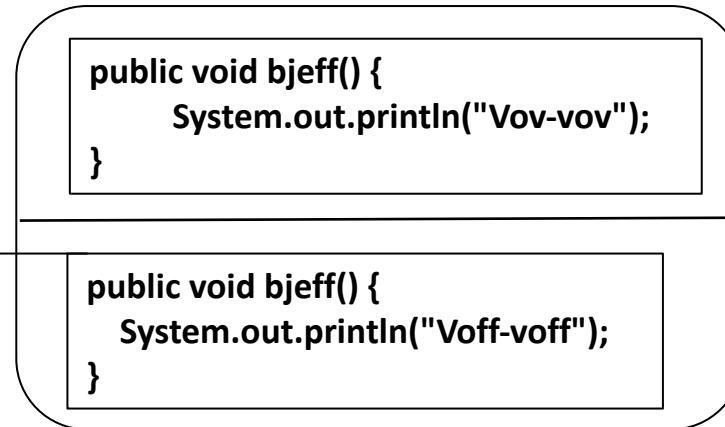
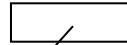
Da brukes den konstruktøren som her de samme parametrene som på kallstedet (samme som overlasting)

# Hoved "take-away" i dag (polymorfi)



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

VanligHund g



VanligHund-del  
av objektet

Rasehund-del  
av objektet

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 redefinerings – engelsk: override
- Samme navn - ulik signatur – kalles overlastering (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)