



UNIVERSITETET  
I OSLO



Institutt for informatikk

# IN1010 våren 2020

Onsdag 5. februar

## Arv og subklasser - del 2

Stein Gjessing  
Institutt for informatikk

# Dagens tema

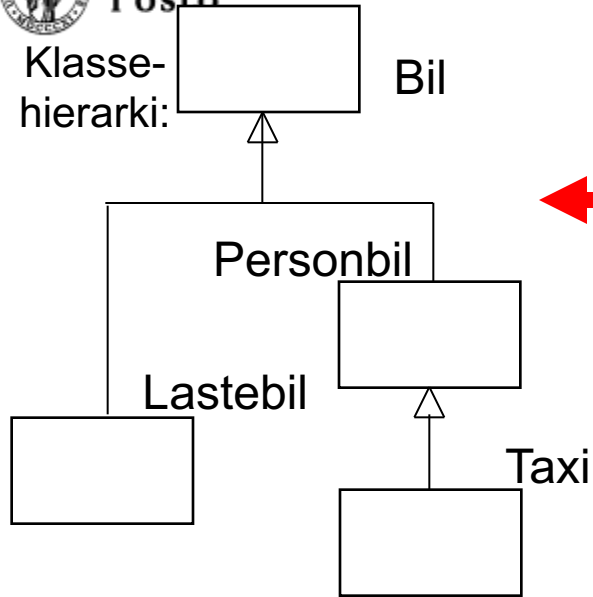
- **Virtuelle** metoder  
som er det samme som
- **Polymorfi**
- **Når bruker vi arv / når bruker vi komposisjon**
- **Konstruktører i subklasser**

”Virtuelle” – ”Polymorfi”  
- fine navn, men det er  
ikke så vanskelig



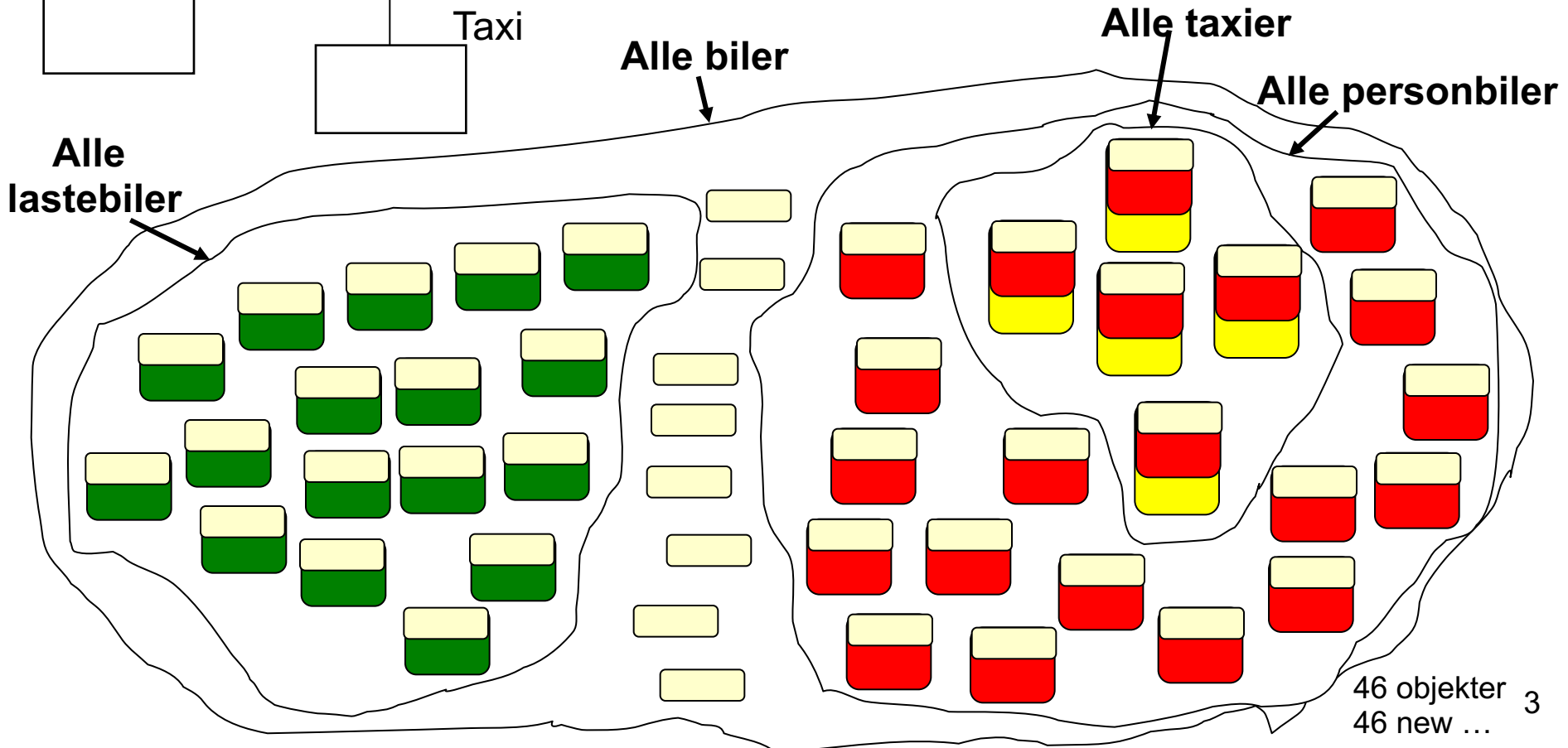
# Repetisjon: Subklasser

Klassehierarki:



```

class Bil { <lys beige egenskaper>}
class Personbil extends Bil { <røde egenskaper> }
class Lastebil extends Bil { <grønne egenskaper> }
class Taxi extends Personbil { <gule egenskaper> }
    
```



46 objekter 3  
46 new ...



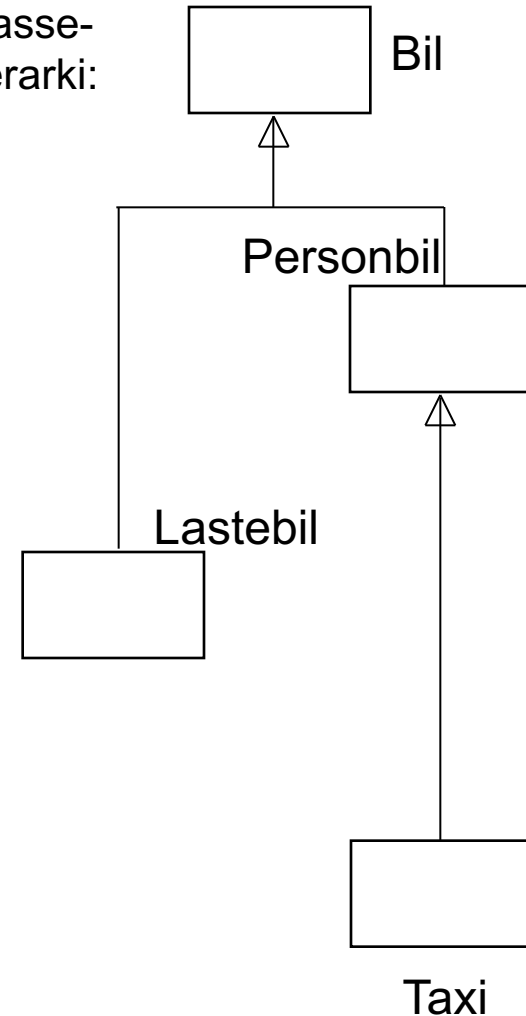
```
class Bil {  
    protected int pris;  
    public int skatt( ) {return pris * 0.25;}  
}
```

```
class Personbil extends Bil {  
    protected int antallPassasjer;  
    public int skatt( ) {return pris * 1;}  
}
```

```
class Lastebil extends Bil {  
    protected double lastevekt;  
    public int skatt ( ) {return pris * 0.5;}  
}
```

```
class Taxi extends Personbil {  
    protected String loyveld;  
    public int skatt ( ) {return pris * 0.25;}  
}
```

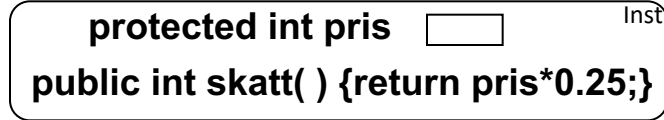
Klasse-  
hierarki:





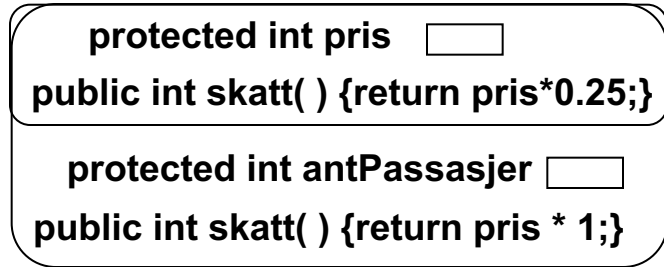
# Polymorfi: eksempel

```
class Bil {
    protected int pris;
    public int skatt( ) {return pris * 0.25;}
}
```



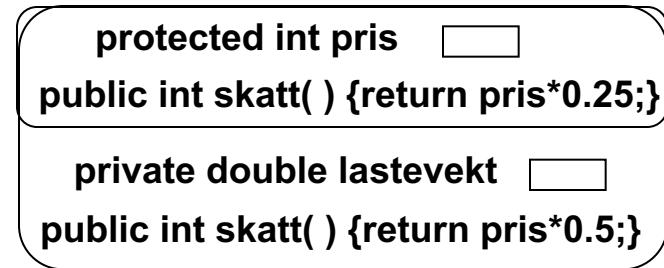
Bil-objekt

```
class Personbil extends Bil {
    protected int antallPassasjer;
    public int skatt( ) {return pris * 1;}
}
```



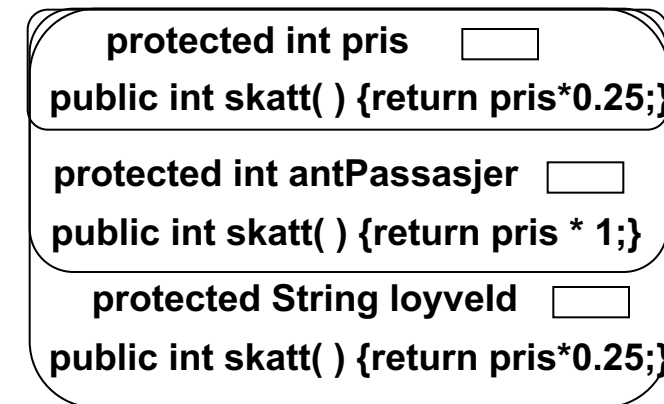
Personbil-objekt

```
class Lastebil extends Bil {
    protected double lastevekt;
    public int skatt ( ) {return pris * 0.5;}
}
```



Lastebil-objekt

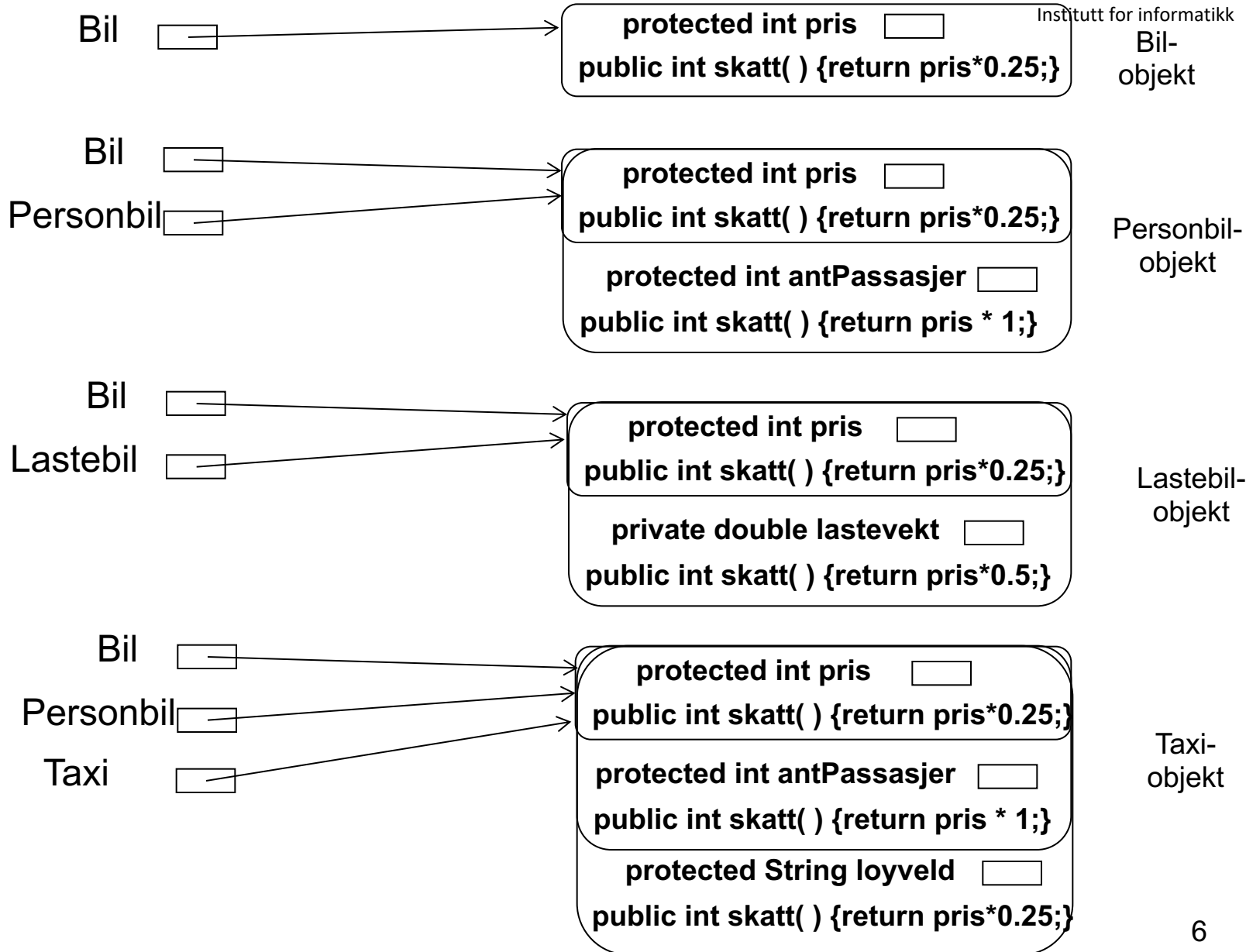
```
class Taxi extends Personbil {
    protected String loyveld;
    public int skatt ( ) {return pris *0.25;}
}
```



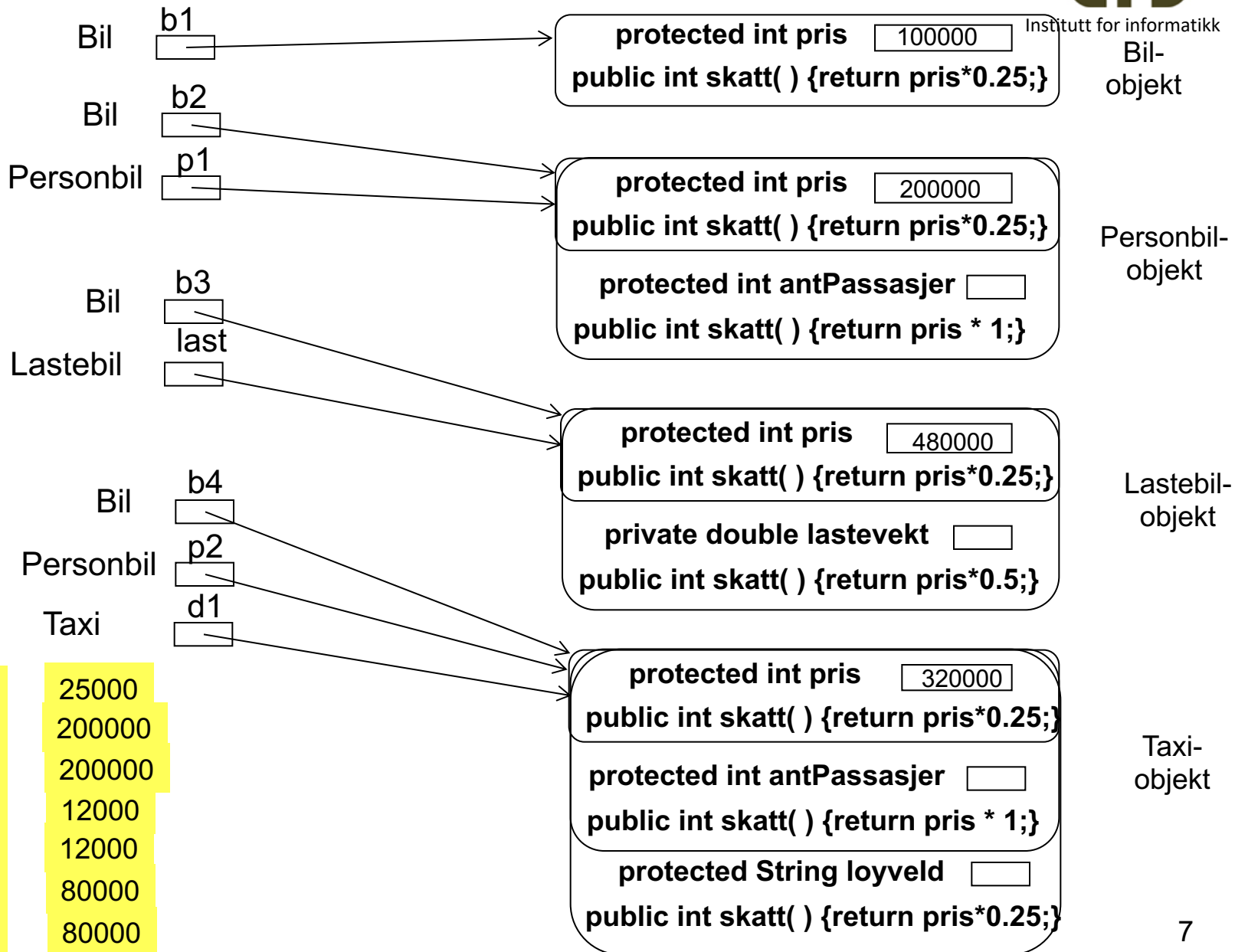
Taxi-objekt



# Polymorfi: eksempel



# Polymorfi: eksempel



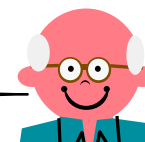
Hva blir:

b1.skatt()	25000
p1.skatt()	200000
b2.skatt()	200000
last.skatt()	12000
b3.skatt()	12000
b4.skatt()	80000
p2.skatt()	80000
d1.skatt()	80000

# Omdefinering av metoder - polymorfi

- Vi har sett at med subklasser kan vi utvide en eksisterende klasse med nye metoder (og nye variable)
- En subklasse kan også definere en metode med *samme signatur* som en metode i superklassen, men med ulikt innhold.
- Den nye metoden vil omdefinere (erstatte) metoden som er definert i superklassen
- Metoder som kan omdefineres på denne måten kalles *virtuelle metoder*
- I Java er alle metoder virtuelle, så sant de ikke er deklartert med **final**

**Virtuelle metoder = polymorfi**







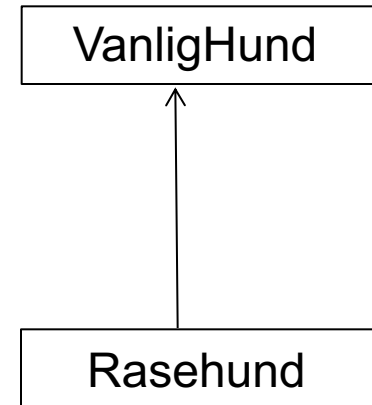
# Polymorfi: Nytt eksempel

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

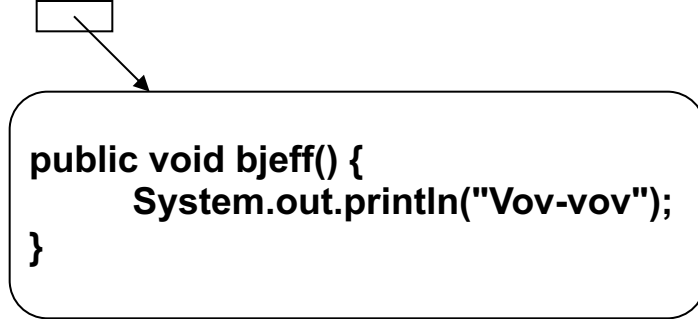
For objekter av typen  
VanligHund er det  
denne metoden som  
gjelder.

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

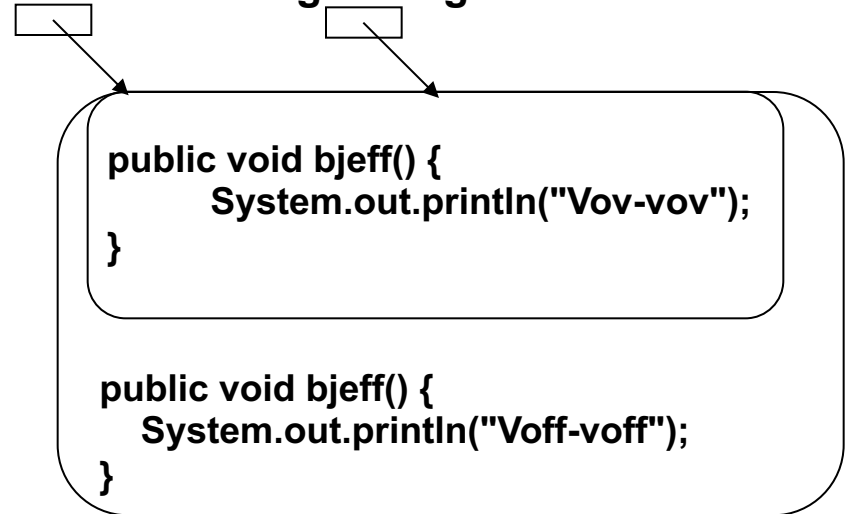
For objekter av typen  
Rasehund er det  
denne metoden som  
gjelder.



VanligHund v



Rasehund r



Anta dette programmet:

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

Hva skrives ut ved hvert av kallene:

```
v.bjeff();  
r.bjeff();  
g.bjeff();
```

```
Vov-vov  
Voff-voff  
Voff-voff
```

Instrument inst



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

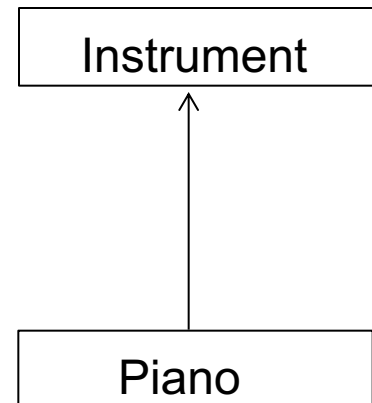
```
void skrivDefinisjon() {  
    System.out.println("Et piano ...");  
}
```

Oppgave:

Hva skrives ut når programmet

Musikk.java kjøres?

```
class Musikk {  
    public static void main (String[] args) {  
        Instrument inst = new Piano();  
        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");  
    }  
}
```





Instrument inst



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

```
void skrivDefinisjon(String overskrift) {
    System.out.println( . . . );
    System.out.println( "Et piano . . );
}
```

## Musikk versjon 2

Hva skjer i dette tilfellet?

```
class Musikk {
    public static void main (String[] args) {
        Instrument inst = new Piano();
        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 (String overskrift) {
        System.out.println(overskrift);
        System.out.println("Et piano er et strengeinstrument");
    }
}
```



Instrument inst



```
void skrivDefinisjon(String overskrift) {  
    System.out.println( . . . );  
    System.out.println( "Et instrument . . . );  
}
```

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

## Musikk versjon 3

```
class Musikk {  
    public static void main (String[] args) {  
        Instrument inst = new Piano();  
        inst.skrivDefinisjon();  
    }  
}  
  
class Instrument {  
    public void skrivDefinisjon(String overskrift) {  
        System.out.println(overskrift);  
        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");  
    }  
}
```

Hva skjer a ?

Programmet  
lar seg ikke  
oversette



# Musikk-eksemplene: Lærdom

- Når vi ser på et objekt via en superklasse-peker, mister vi vanligvis tilgang til metoder og variable som er definert i subklassen.
- Dersom en metode i subklassen også er definert (**med samme signatur**) i superklassen har vi likevel tilgang via superklasse-pekeren, fordi objektets "dypeste" metode brukes.  
Slike metoder kalles virtuelle metoder, og denne mekanismen kalles polymorfi.
- Det som er relevant er derfor *hvilke* metoder som finnes i superklassen (med hvilke parametre), men *ikke* nødvendigvis *innholdet* i metodene.

Samme signatur = samme navn og nøyaktig samme parametre (ikke inkl. returtype i Java, men Java protesterer hvis gal returtype)



# Flere virtuelle 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;
    }
    protected int prisUtenMoms() {
        return pris-(pris*rabatt/100);
    }
}
```

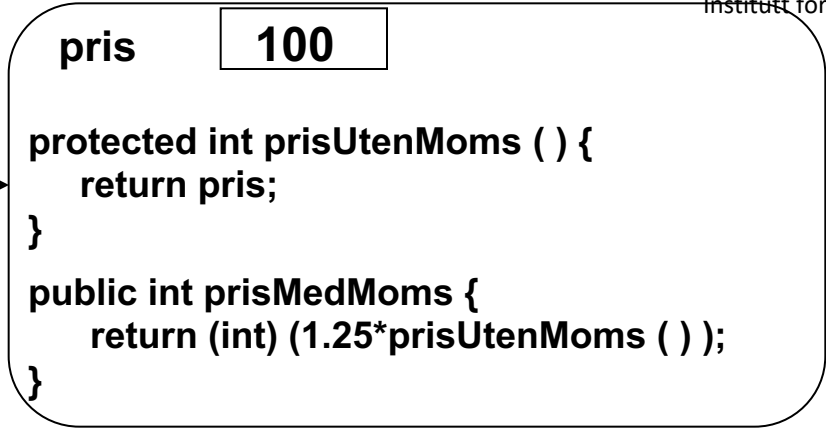
```
Vare v = new Vare();  
v.setPris(100);
```

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

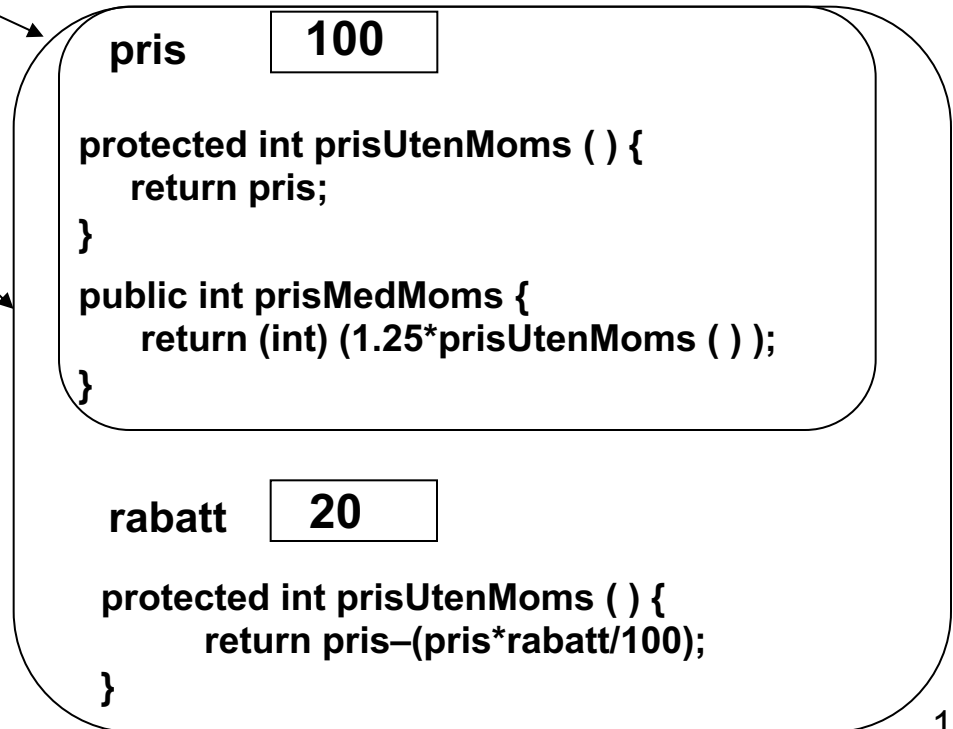
Hva blir nå:

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

Vare v



Vare v2



SalgsVare s







# Polymorfi: skrivData

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

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

// I klassen Ansatt:
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**

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

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

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

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

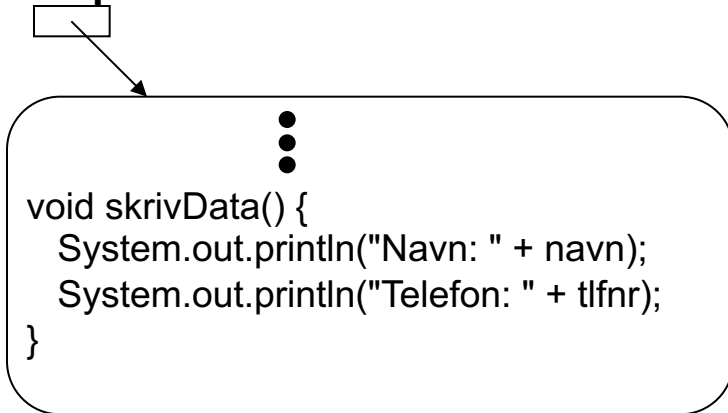


```
class StudentRegister {
    public static void main(String [] args) {
        Student stud = new Student();
        Person pers = new Person();

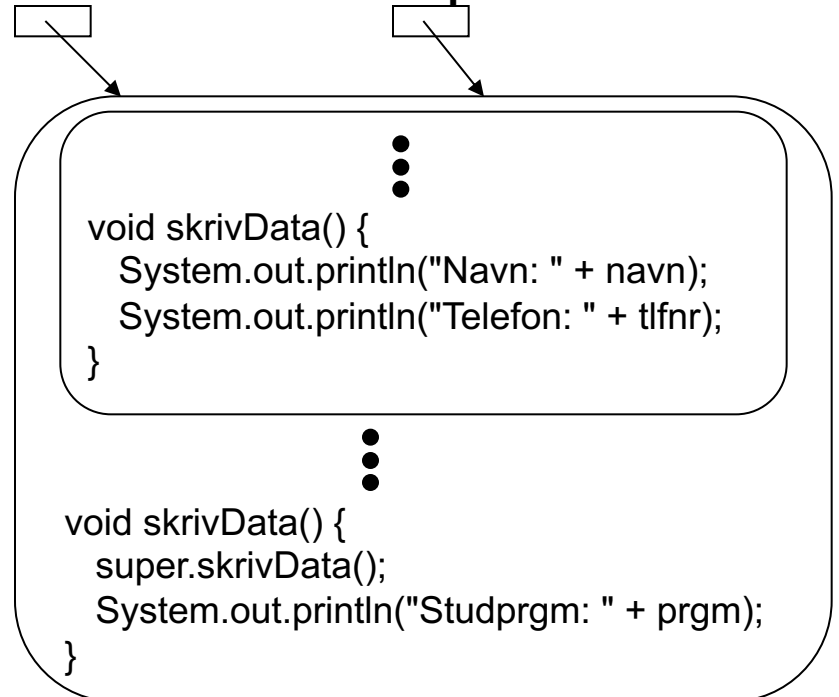
        stud.skrivData(); // Her brukes definisjonen i Student
        pers.skrivData(); // Her brukes definisjonen i Person

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

Person pers



Student stud



**Regel:** Det er *objekttypen*, ikke referansetypen, som avgjør hvilken definisjon som gjelder når en metode er virtuell.

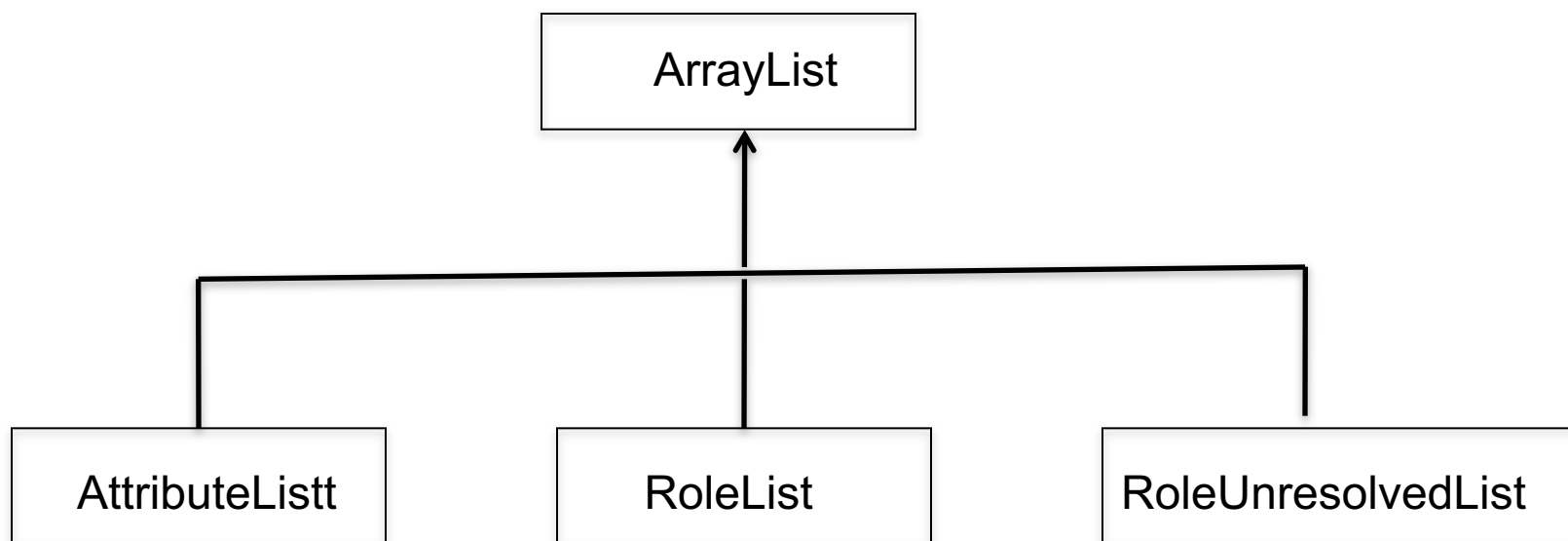


# Omdefinering av **variable**

- En subklasse kan også omdefinere (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...

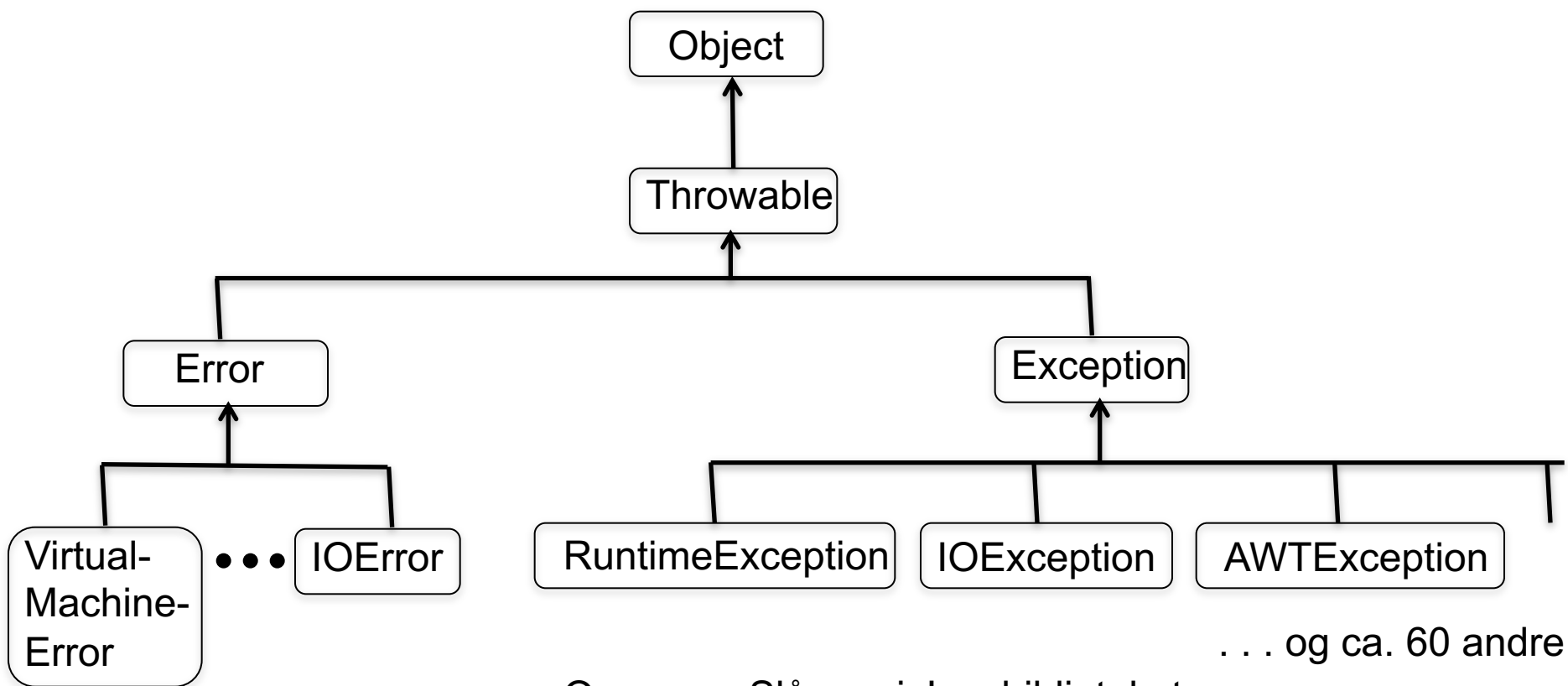


# *Klassehierarki: Et eksempel fra Java-biblioteket*



Oppgave: Slå opp på ArrayList i Javabiblioteket og se at de tre klassene er direkte subklasser av ArrayList (direkte naboer i klassehierarkiet)  
**(“Direct Known Subclasses”)**

# Klassehierarki: Også fra Java-biblioteket: Feil-klasser

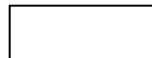
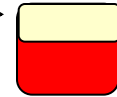
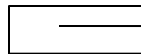


Oppgave: Slå opp i Javabiblioteket

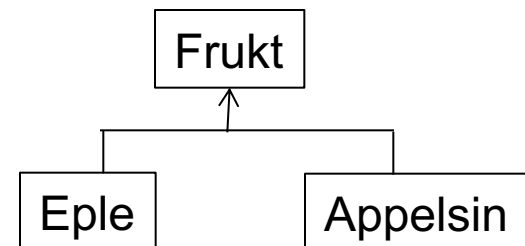
# Repetisjon:

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"

Istedenfor å teste hvilken klasse objektet er av,  
la objektet gjøre jobben selv:

Bedre program:

Bruk heller  
polymorfi  
hvis mulig

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



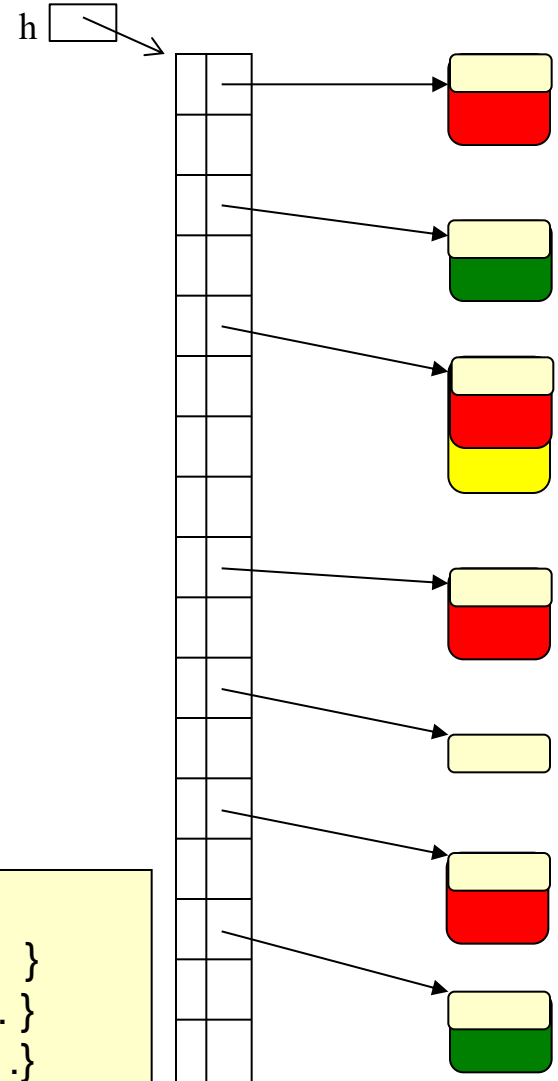
```
abstract class Frukt2 {  
    abstract public void skrivUt( );  
}  
  
class Eple2 extends Frukt2 {  
    public void skrivUt( ) {  
        System.out.println("Jeg er et eple!");  
    }  
}  
  
class Appelsin2 extends Frukt2 {  
    public void skrivUt( ) {  
        System.out.println("Jeg er en appelsin!");  
    }  
}
```



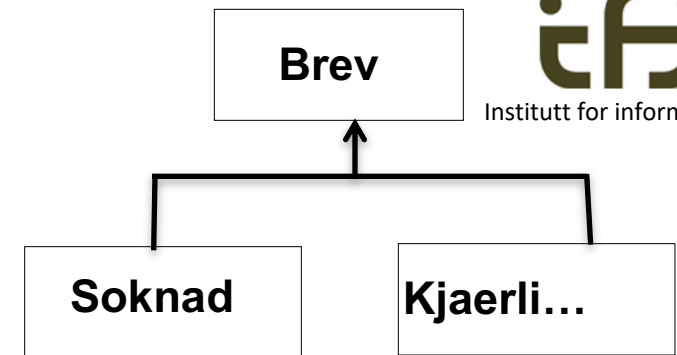
# Biler og mer bruk av "instanceof"

```
HashMap <String, Bil> h;  
h = new HashMap <String, Bil> ( );  
.  
.  
for (Bil b: h.values()) {  
    String nr = b.regNr;  
    // kall på virtuell metode:  
    b.skatt( );  
    if (b instanceof Personbil) {  
        Personbil pb = (Personbil) b;  
        int pas = pb.hentAntPass();  
    } else {  
        if (b instanceof Lastebil) {  
            Lastebil ls = (Lastebil) b;  
            double lv = ls.hentLasteVekt();  
        }  
    }  
}
```

(og litt "casting" (typekonvertering))



```
class Bil { String regNr; public void skatt(){...} ... }  
class Personbil extends Bil { int antPass; public void skatt(){...} ... }  
class Lastebil extends Bil { double lasteVekt; p.. void skatt(){...} ... }  
class Drosje extends Personbil { int LøyveNr; p.. void skatt(){...} ... }
```



- Anta at vi har deklarasjonene  
**class Brev { ... }**  
**class Soknad extends Brev { ... }**  
**class Kjaerlighetsbrev extends Brev { ... }**
- Avgjør hvilke av følgende uttrykk som er lovlige:

	Lovlig	Ulovlig
<b>Soknad s1 = new Soknad();</b>	✓	
<b>Soknad s2 = new Brev();</b>		✓
<b>Brev b1 = new Soknad();</b>	✓	
<b>Brev b2 = (Brev) new Soknad();</b>	✓	
<b>Soknad s3 = new Kjaerlighetsbrev();</b>		✓
<b>Soknad s4 = (Soknad) new Kjaerlighetsbrev();</b>		✓
<b>Brev b3 = (Soknad) new Brev();</b>		✓

# 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





- 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

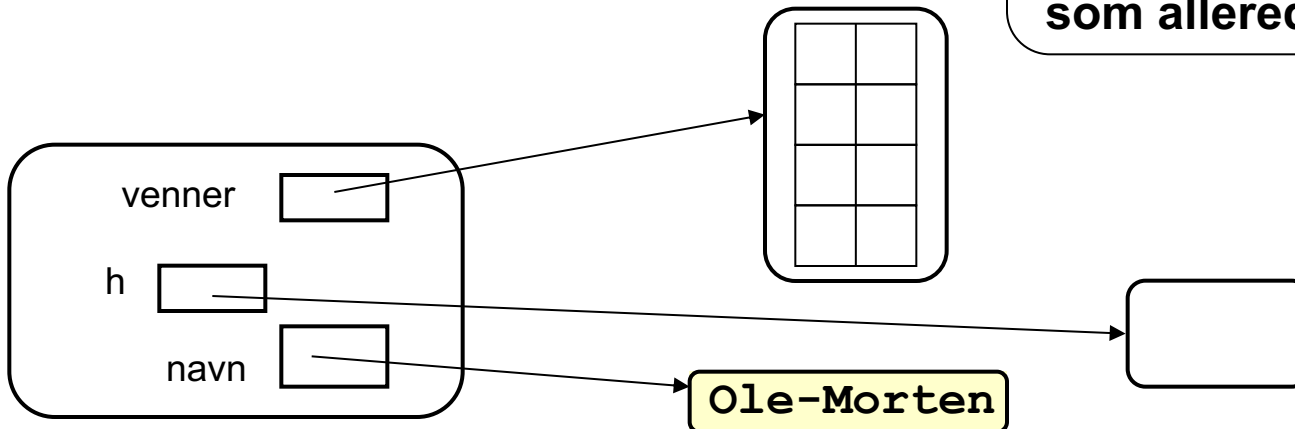
- Ved **sammensetning** (komposisjon)
  - (i Oblig 1, pensum i IN1000):
    - Deklarer referanser til objekter av klasser du har skrevet før (eller biblioteksklasser)
    - Lag objekter av disse klassene
    - Kall på metoder i disse klassene
- Ved **arv** (nytt i IN1010):
  - Lag en ny klasse som utvider den eksisterende klassen  
(spesielt viktig ved litt større klasser)
  - Føy til ekstra variable 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 ved arv

(domenet er et bibliotek)

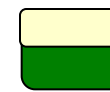
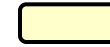
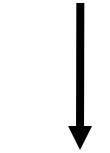
```
class Bok {
    protected String tittel, forfatter;
}

class Fagbok extends Bok {
    protected double dewey;
}

class Skjønnlitterærbok extends Bok {
    protected String sjanger;
}

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

Objekter av  
klassene



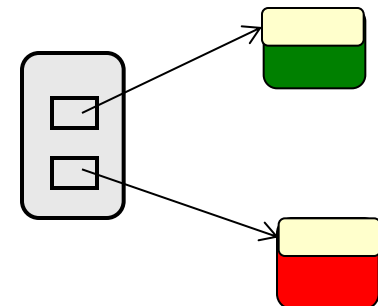
← *Gjenbrukes*



← *Gjenbrukes*

Arv

Komposisjon



# 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 (som datafelt (konstanter/variable)).

**Arv vs. delegering**





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

Relasjon mellom	Komposisjon	Arv
vare - varelager	✓	
nyhetskanal - kanal		✓
person - personregister	✓	
cd - spor (sanger)	✓	
PC - datamaskin		✓
gaupe - rovdyr		✓
fly - transportmiddel		✓
motor - bil	✓	



# Object: toString og equals

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

# Eksempel på toString og equals

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

Anta:

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

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

Hva blir nå:

```
p1.toString(); Punkt@f5da06
p1.equals(p2); false
```

```
class Punkt2 {
    protected int x, y;
    Punkt2(int x0, int y0) {
        x = x0; y = y0;
    }

    public String toString() {
        return ("x = "+x+" y = "+y);
    }

    public boolean equals(Object o) {
        if (!(o instanceof Punkt2))
            return false;
        Punkt2 p = (Punkt2) o;
        return x == p.x && y == p.y;
    }
}
```

```
q1.toString(); x = 3 y = 4
q1.equals(q2); true
```

# Konstruktører

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.

navn: p  
  
type: Punkt

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

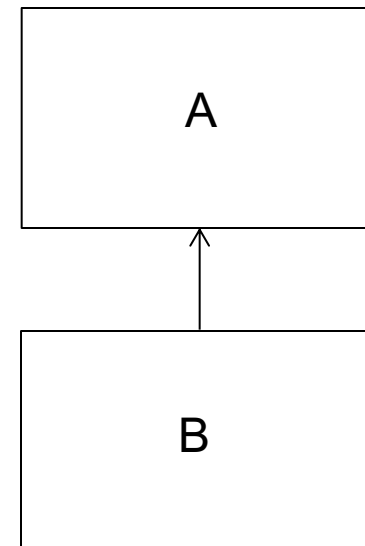
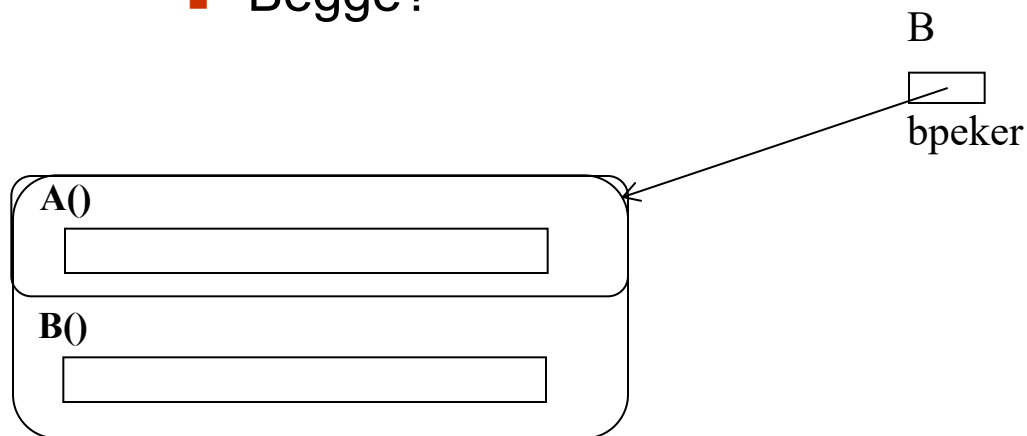
navn: x  3 type: int  
navn: y  4 type: int  
**Punkt(int x0, int y0)**

navn: x0  3 type: int  
navn: y0  4 type: int  
x = x0; y = y0;

# Konstruktører og arv

Det blir noe mer komplisert når vi opererer med arv:

- Anta at vi har definert en subklasse  
`class B extends A { ... }`
- Hvilken konstruktør utføres hvis vi skriver  
`B bpeker = new B();`
  - Konstruktøren i klassen A?
  - Konstruktøren i klassen B?
  - Begge?



# Konstruktører og arv (forts.)

- Anta at vi har deklarerert 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, som utføres
  6. Kontrollen kommer tilbake til C sin konstruktør, som utføres

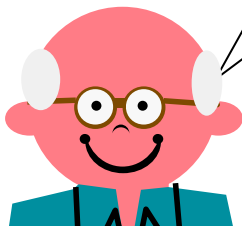


# Kall på super-konstruktøren

- Superklassens konstruktør kan 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!**  
**Det er forskjell på**  
**super.**  
**og**  
**super( . . . )**





# Eksempel 1

Anta at vi har følgende klasser:

```
class Person {
    protected String fødselsnr;

    Person() {
        fødselsnr = "12034567890";
    }
}

class Student extends Person {
    protected int studID;

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

Anta to konstruktører:

```
Student() {
    super();
    studID = 0;
}
```

eller:

```
Student() {
    studID = 0;
}
```

Disse to er helt ekvivalente!

Hva skjer hvis Student ikke har noen konstruktør : ?

```
class Student extends Person {
    int studID = 0;
}
```

Svar: det går bra



## Eksempel 2

Her er fire 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 = 0;
}
```

```
Student() {
    super("12345");
    studID = 0;
}
```

```
Student(String nr) {
    super(nr);
    studID = 17;
}
```

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

Hvilke virker?  
Diskuter!



# 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

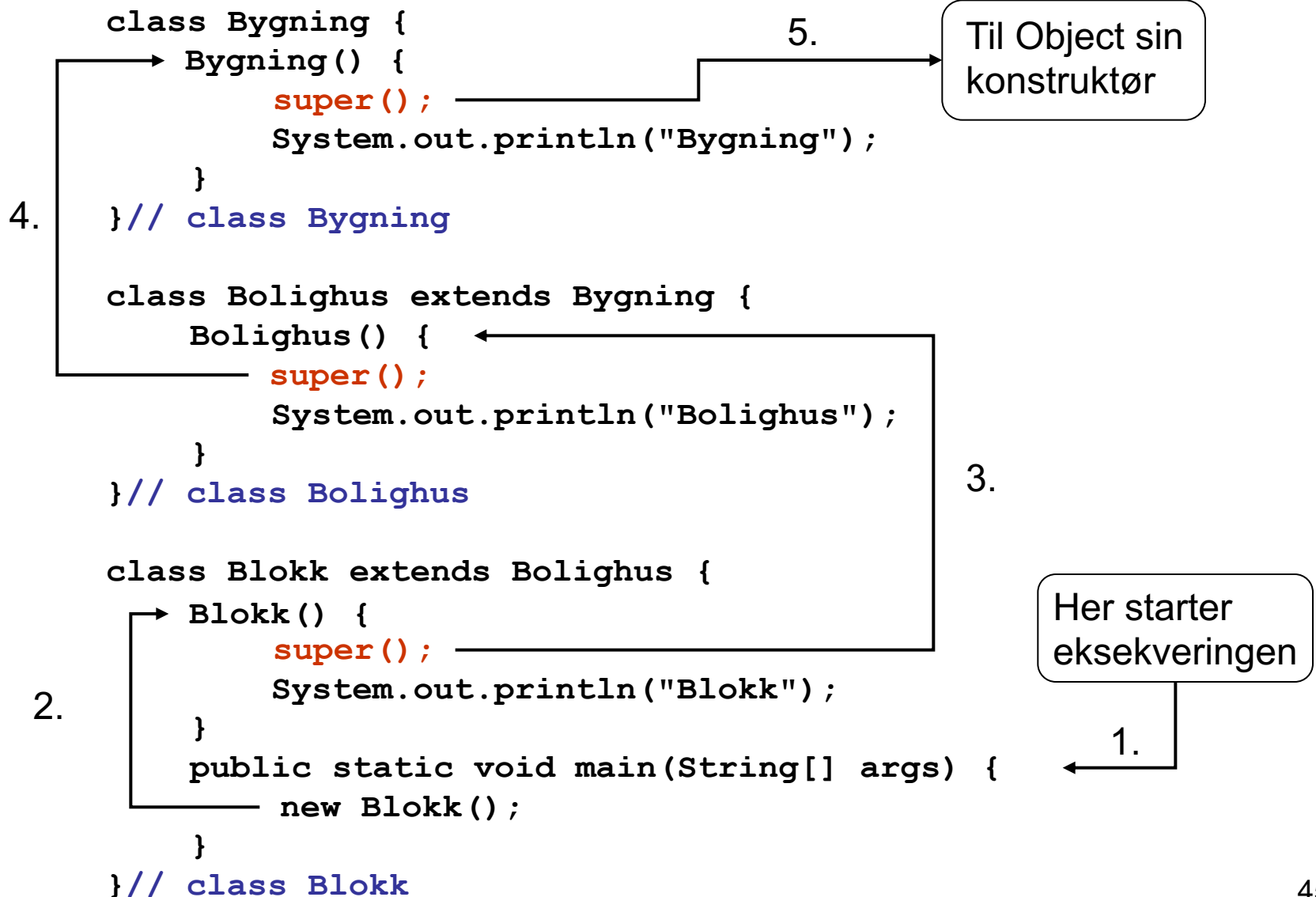
```
class Bygning {
    Bygning() {
        super();
        System.out.println("Bygning");
    }
} // class Bygning

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

class Blokk extends Bolighus {
    Blokk() {
        super();
        System.out.println("Blokk");
    }
    public static void main(String[] args) {
        new Blokk();
    }
} // class Blokk
```

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

# Når programmet utføres



# Når programmet utføres (forts.)

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

6.

Tilbake fra Object  
sin konstruktør

7.

Nå er  
Bygning  
skrevet ut

```
}
} // class Bygning

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

8.

Nå er  
Bolighus  
skrevet ut

```
}
} // class Bolighus

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

9.

Nå er Blokk  
skrevet ut

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

Merk:  
Konstruktøren i klassen Bolighus har nå en parameter.



# Når programmet kompileres

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

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

class Blokk extends Bolighus {
    Blokk() {
        super();
        System.out.println("Blokk");
    }
    public static void main(String[] args) {
        new Blokk();
    }
} // class Blokk
```

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

Men: Kallet matcher ikke  
metoden i antall parametre!

Mulige løsninger:

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