UNIVERSITETET I OSLO

Institutt for informatikk

INF1010 våren 2017

Torsdag 9. februar

Interface - Grensesnitt

og litt om generiske klasser og generiske interface hvis tid

Stein Gjessing

Institutt for informatikk

Dagens hovedtema

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

- Les notatet “Grensesnitt i Java” av Stein Gjessing

- To motivasjoner for interface
  1) Tydeliggjøre klassens public-metoder
  2) Multippel arv

Dagens tema 2 (en forsmak, hvis tid)

- Engelsk: Generic classes
- Norsk: Generiske klasser - klasser med type-parametre

- Neste uke:
  - Grundig om generiske klasser.
    Les notatet “Generiske klasser i Java” av Stein Gjessing

Multippel arv: Om å "arve" fra flere

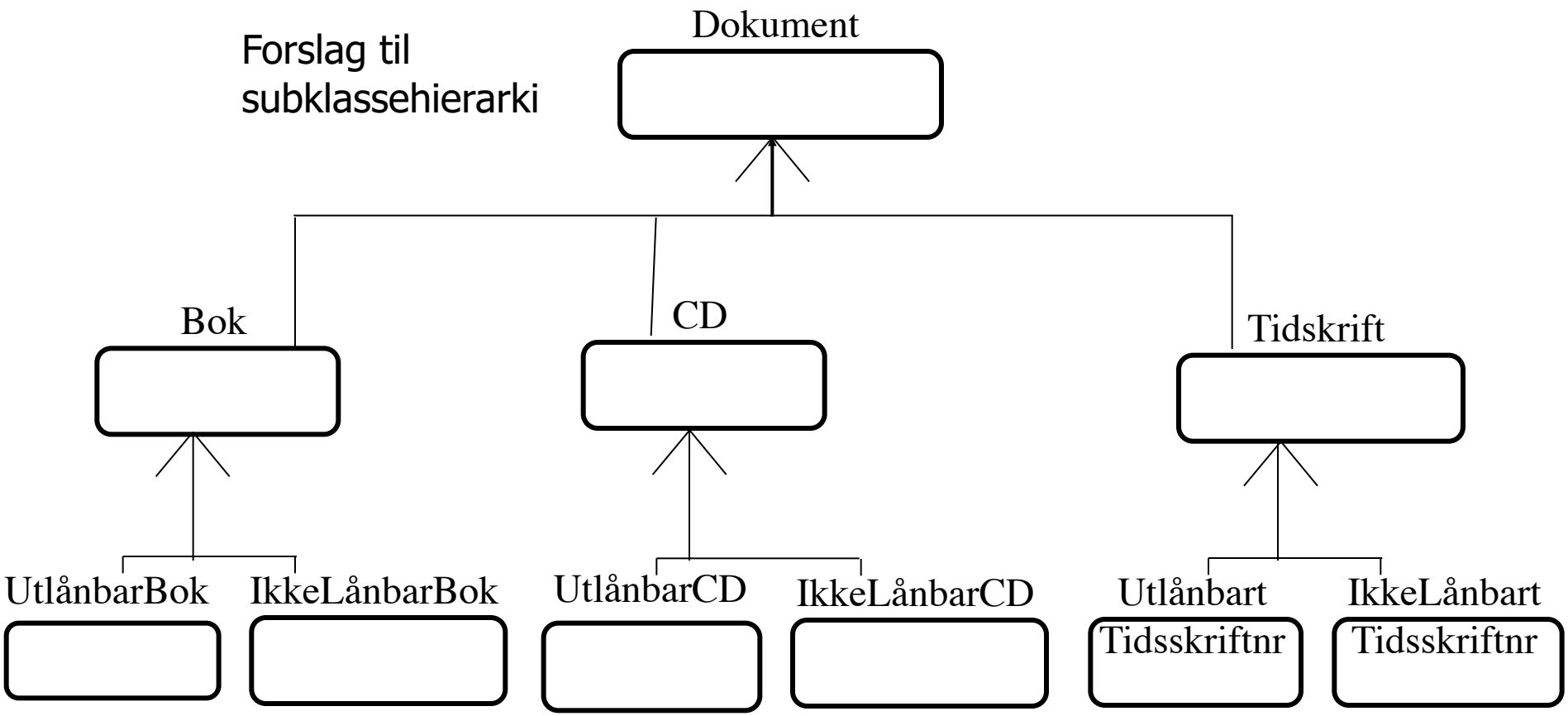
- I Java kan en klasse bare arve egenskapene til **én** annen klasse (en superklasse).
  - Dette gjør språket sikrere å bruke
- Hva skal vi gjøre hvis vi ønsker at et objekt skal inneholde mange forskjellige egenskaper fra forskjellige "superklasser" ?

- På de neste sidene:
  - Begrepshierarkiet i et bibliotek

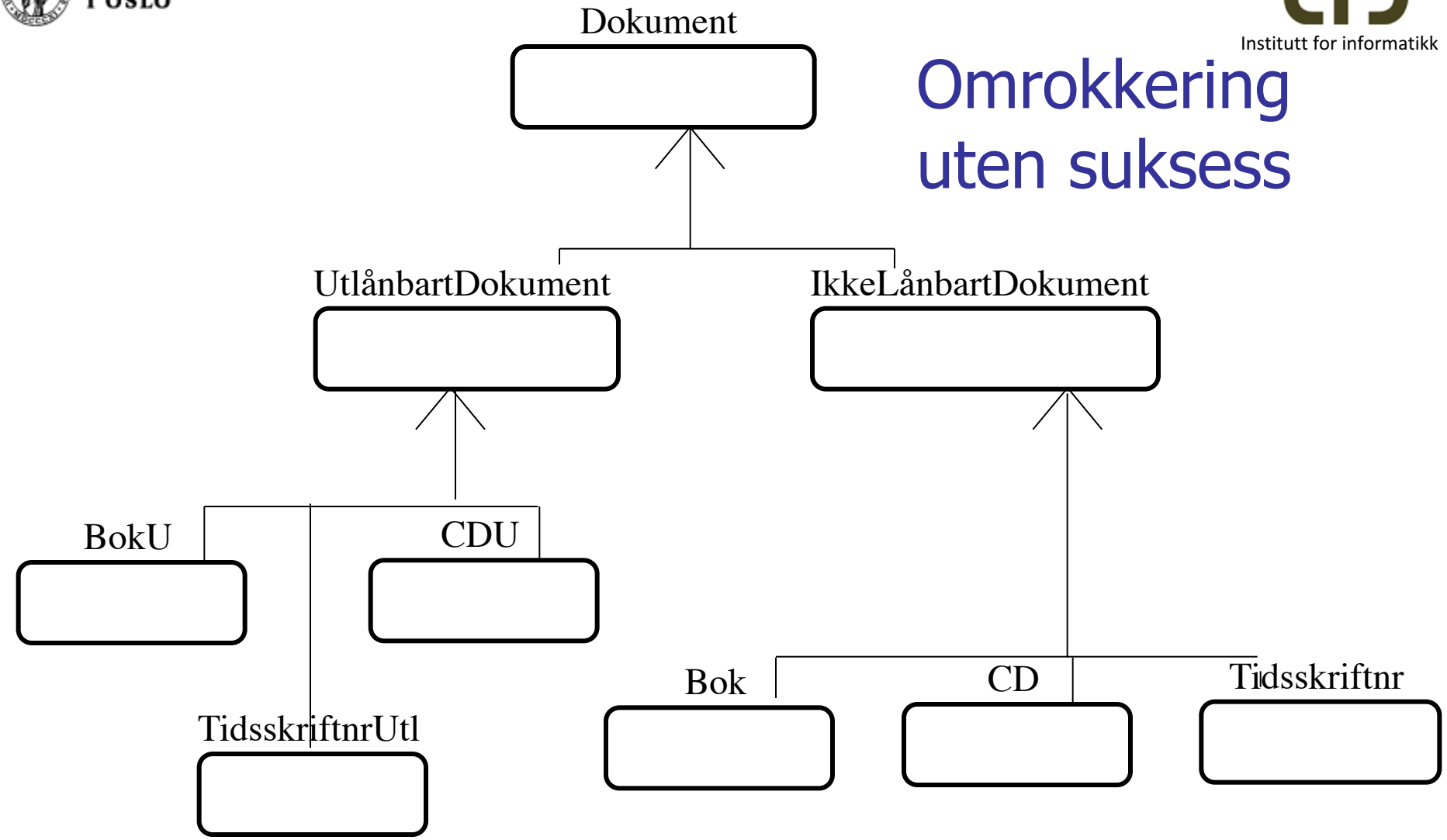
Motivasjon for begrepet interface: Analyse av bibliotek

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

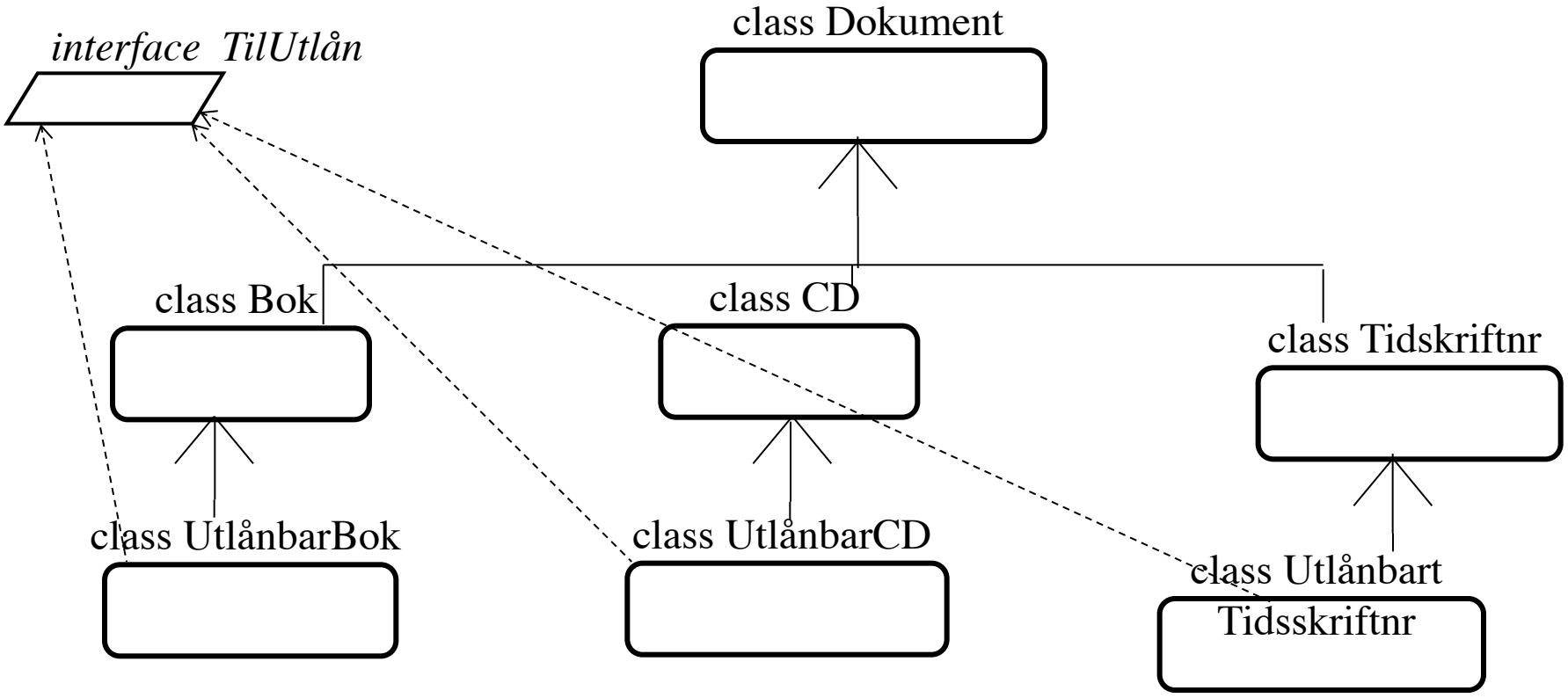
Tvilsomt begrepshierarki



Omrokkering uten suksess



Samle lik oppførsel: bruk **interface**



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

Et interface (grensesnitt) er:

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

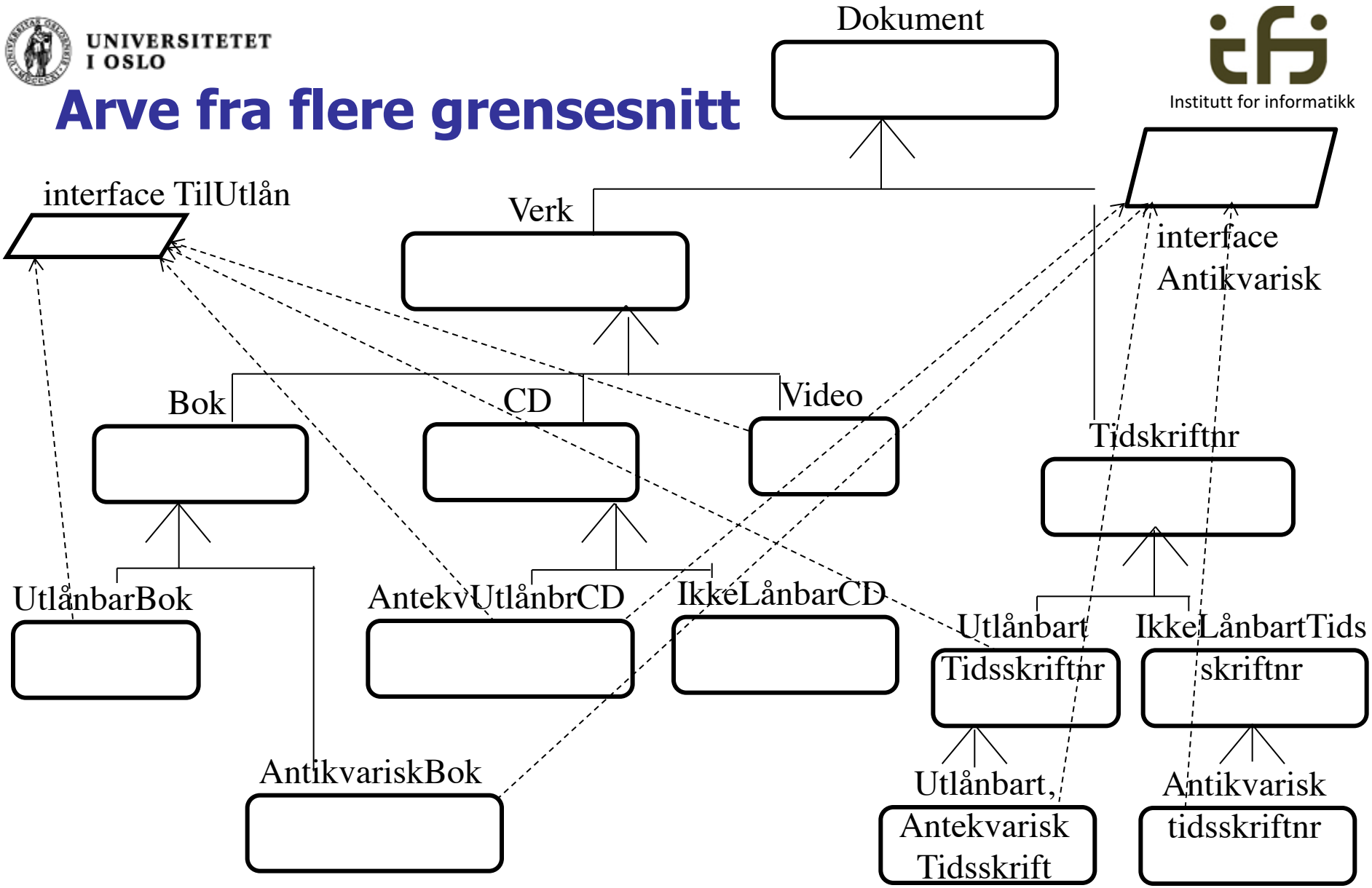
- For eksempel
  - Kan delta i konkurranse (startnummer, resultat, ..
    Mennesker, biler, hester kan delta i konkurranser)
  - Svømmedyktig (mennesker, fugler er svømmedyktige)
  - Her: Antikvarisk (møbler, bøker, ….)
    Kan lånes ut (biler, bøker, festklær, …)
  - Sammenlignbar (Comparable)
  - . . .

Hva er et interface ?

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

- Å arve (en samling metoder) = å spille en rolle

Arve fra flere grensesnitt



- En klasse kan tilføres et ubegrenset antall interface-er
- Dvs. en klasse kan spille et ubegrenset antall roller
- Felles egenskaper på tvers av klassehierarkiet

Nytt interface-eksempel

(Vi kommer tilbake til biblioteket senere)

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

```
interface Skattbar {
    double  toll();
    int momsSats();
}
```

interface istedenfor class

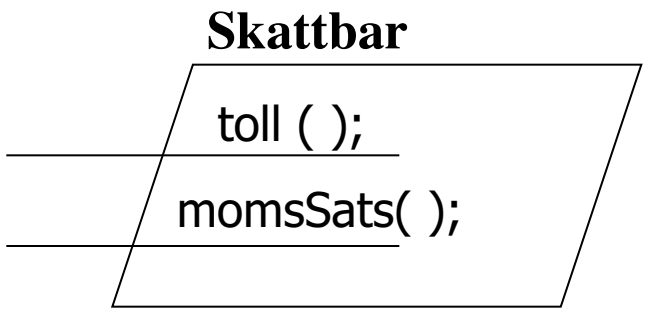
; istedenfor innmat i metodene

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

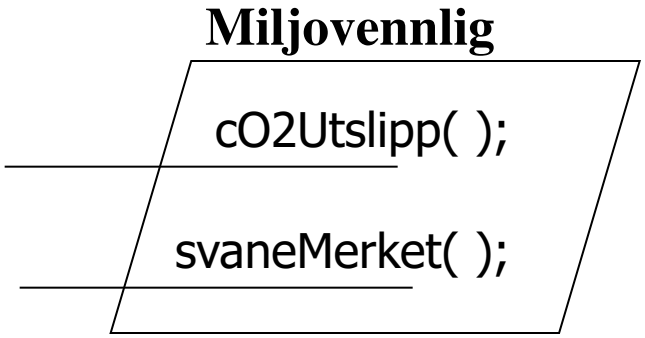
Nytt Java nøkkelord: **interface**

Vi tegner et interface slik (når vi tar med både navn og metoder)

```
interface Skattbar {
     double  toll();
     int momsSats();
}
```

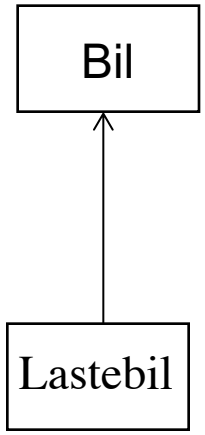



```
interface Miljovennlig {
     int cO2Utslipp ();
     boolean svaneMerket ();
}
```



Enkelt eksempel med bil-hierarkiet

```
class Bil { String regNr;}
class Lastebil extends Bil {double lasteVekt;}
```




```
interface Skattbar {
    double  toll( ) ;
    int momsSats() ;
}
```

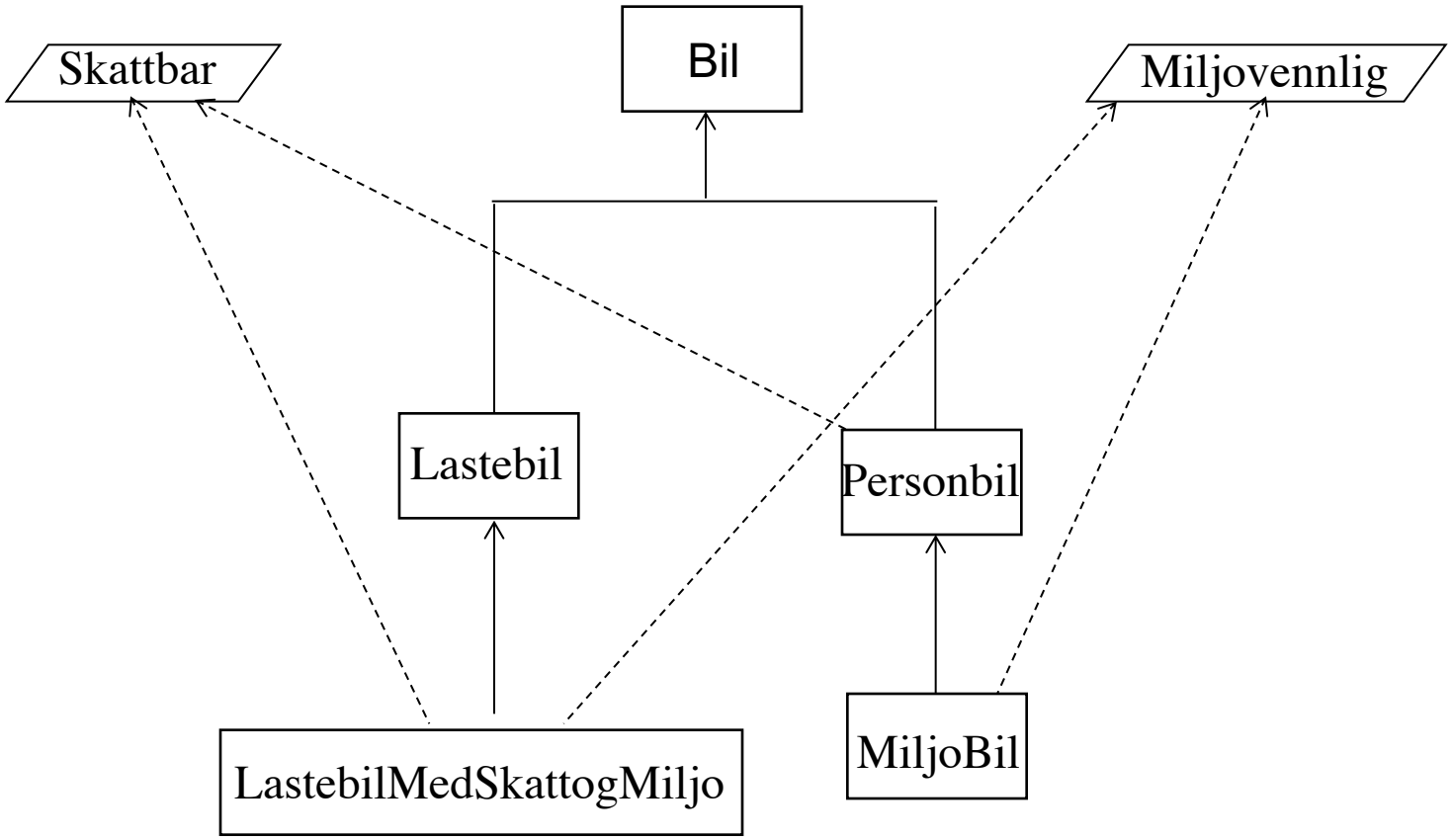
Skattbar

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

Miljovennlig

Metodene i et grensesnitt er veldig ”abstrakte”

Tre nye klasser som kan spille mange roller



Men metodene må (dessverre) skrives på nytt hver gang de brukes

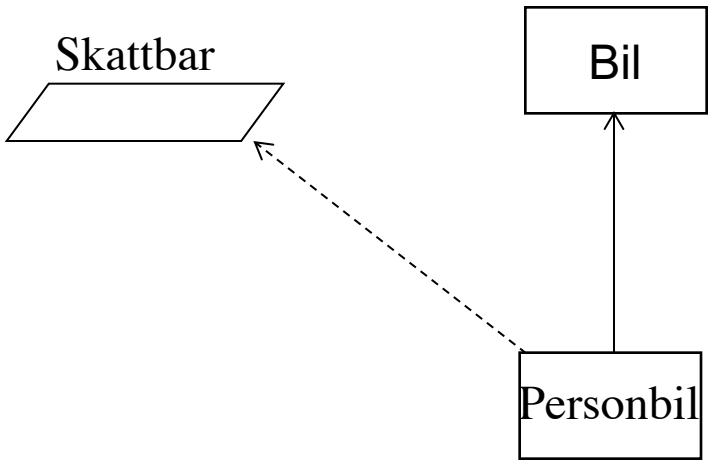
Reservert Java-ord: **implements** (1)

rollen Bil (i arv) fra klassehierarkiet

```
class Personbil extends Bil implements Skattbar {
    int antPass;
    double momsGrunnlag = 150000;
    public double toll( ){return momsGrunnlag*0.5;}
    public int momsSats( ){return 25;}
}
```

rollen "Skattbar"

```
class Bil {
    String regNr;
}

interface Skattbar {
    double  toll();
    int momsSats() ;
}
```



Nytt Java-ord: **implements** (2)

Rollene i (arv fra) klassehierarkiet

```
class MiljoBil extends Personbil implements Miljovennlig {
   int utslipp = 100;
   public int cO2Utslipp(){return utslipp;}
   public boolean svaneMerket(){return false;}
}
```

} rollen "Miljovennlig"

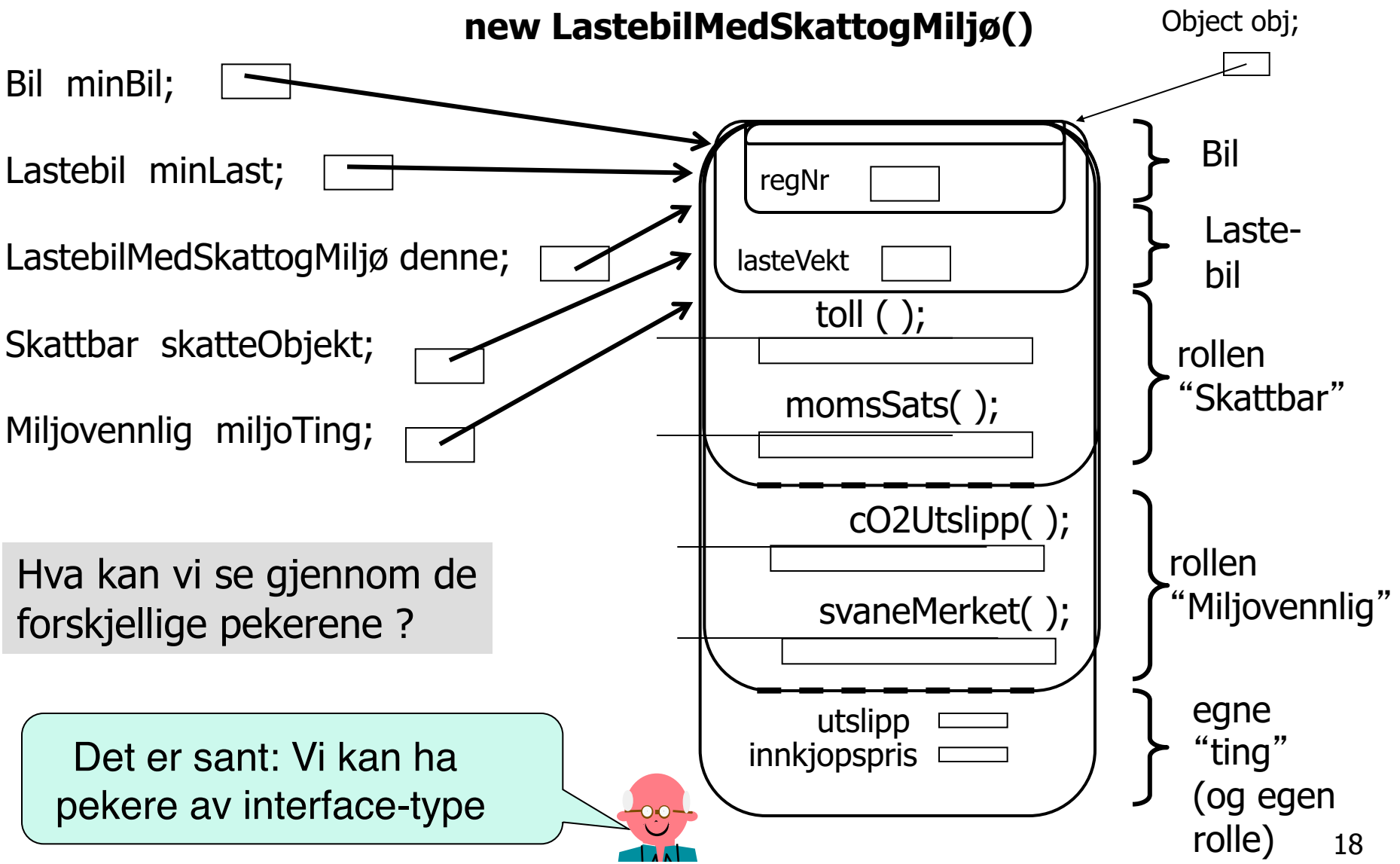
```
class LastebilMedSkattogMiljo extends Lastebil implements Skattbar,
Miljovennlig {
   double innkjopspris = 200000;
   int utslipp = 400;
   public double toll(){return innkjopspris*0.1;}
   public int momsSats(){return 25;}
   public int cO2Utslipp(){return utslipp;}
   public boolean svaneMerket(){return false;}
}
```

} rollen "Skattbar"

} rollen "Miljovennlig"

MiljoBil arver rollen Skatt fra Personbil

Et objekt og noen pekere



Hva kan vi se gjennom de forskjellige pekerene ?

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

Multippel arv

Rollene i (arv fra) klassehierarkiet

```
class MiljoBil extends Personbil implements Miljovennlig {
    int utslipp = 200;
    public int cO2Utslipp ( ) {return utslipp; }
    public boolean svaneMerket () { return false;}
}
```

} rollen "Miljovannlig"

```
class LastebilMedSkattogMiljo extends Lastebil implements Skattbar, Miljovennlig {
    double innkjopspris = 200000;
    int utslipp = 400;
    public double  toll( ) { return innkjopspris * 0.1; }
    public int momsSats( ) {return 20;}
    public int cO2Utslipp ( ) {return utslipp; }
    public boolean svaneMerket () { return false; }
}
```

} rollen "Skattbar"

} rollen "Miljovennlig"

MiljoBil arver rollen Skatt fra Personbil

Mer om grensesnitt (interface)

- Navnet på et interface kan brukes som typenavn når vi lager referanser (det så vi på side 18)
- Vitsen med et interfaceer å spesifisere **hva** som skal gjøres (ikke hvordan)
- Vanligvis er det flere implementasjoner av et interface (flere klasser implementerer det).
- Vi vet: En klasse kan implementerer (flere) interface samtidig som klassen også er subklasse av (bare) én annen klasse.
- En implementasjon (av et interface) skal kunne endres uten at resten av programmet behøver å endres.

Grensesnitt (interface) lærdom

- Et interface har bare
  - metodenavn med parametre, men ikke kode (husk ;)
  - konstanter (static final - eks static final int ANTALL = 4;)
- Bruker 'interface' i steden for 'class' før navnet
- Definerer en 'type' / 'rolle' som andre må implementere
- Meget nyttig, brukes mye ved distribuerte systemer og generelle programbiblioteker som Javas eget
- Ulempe: Koden/implementasjonen må gjøres mange ganger
- **Mer generelt kjent under navnet ADT** =Abstrakt DataType ,
  Vi definerer ***hva*** en ny datatype skal gjøre, ***ikke hvordan*** dette gjøres.
  - Det kan være mange mulige implementasjoner (=måter å skrive kode på) som lager en slik datatype.
  - Hva som er beste implementasjon må avgjøres etter hvilken bruk vi har.

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

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


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

Skriv i LastebilMedSkattOgMiljo

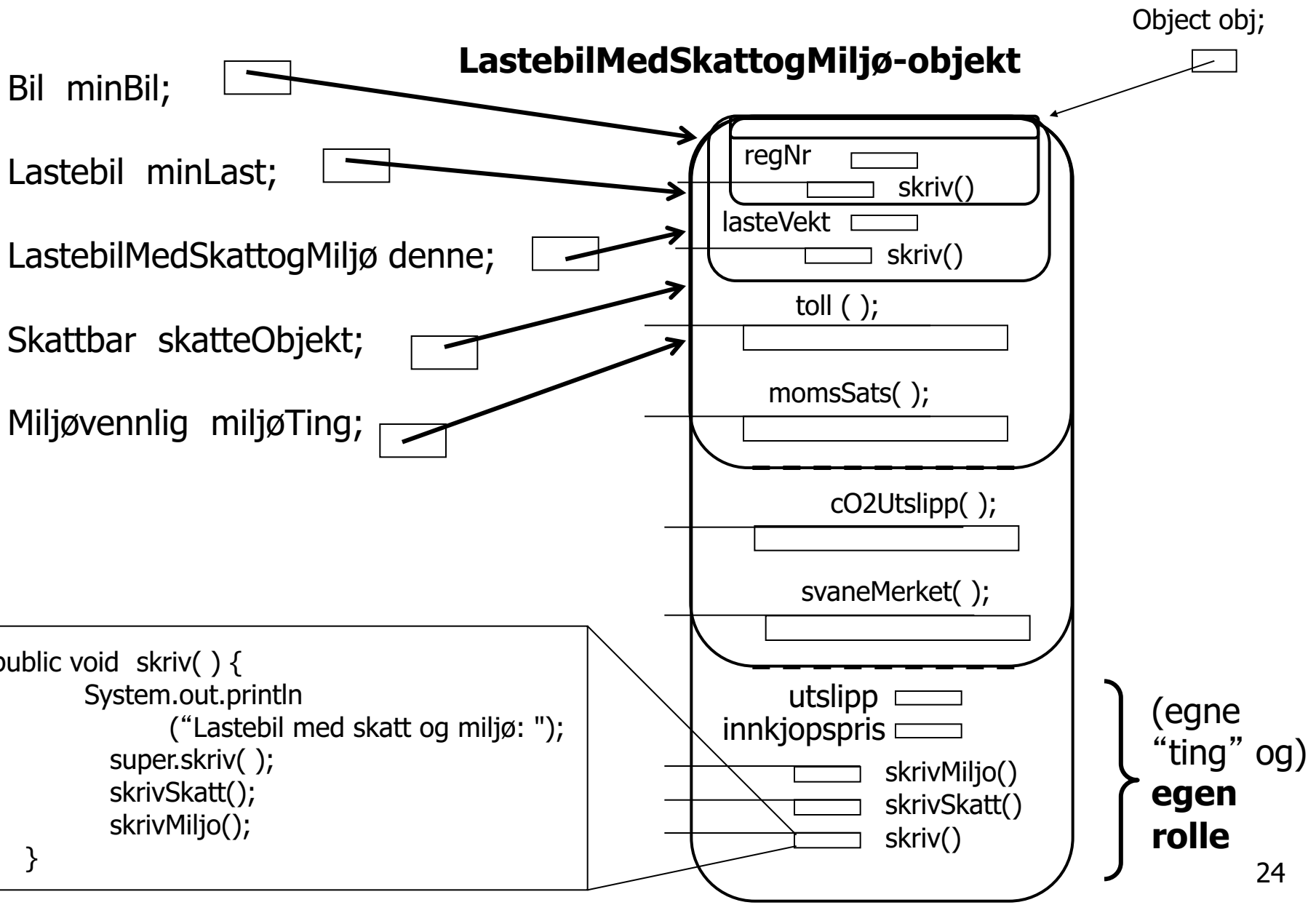

```
class LastebilMedSkattOgMiljo extends Lastebil implements Skattbar, Miljovennlig
{
   double innkjopspris = 200000;
   int utslipp = 400;
   public double  toll( ) { return innkjopspris * 0.1; }
   public int momsSats( ) {return 20;}
   public void skrivSkatt( ) {
                        System.out.println("Innkjøpspris " + innkjopspris); }
   public int cO2Utslipp ( ) {return utslipp; }
   public boolean svaneMerket () { return false; }
   public void skrivMiljo( ) { System.out.println("Utslipp " + utslipp); }

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

(Som før)

(Skattbar og Miljovennlig som før)

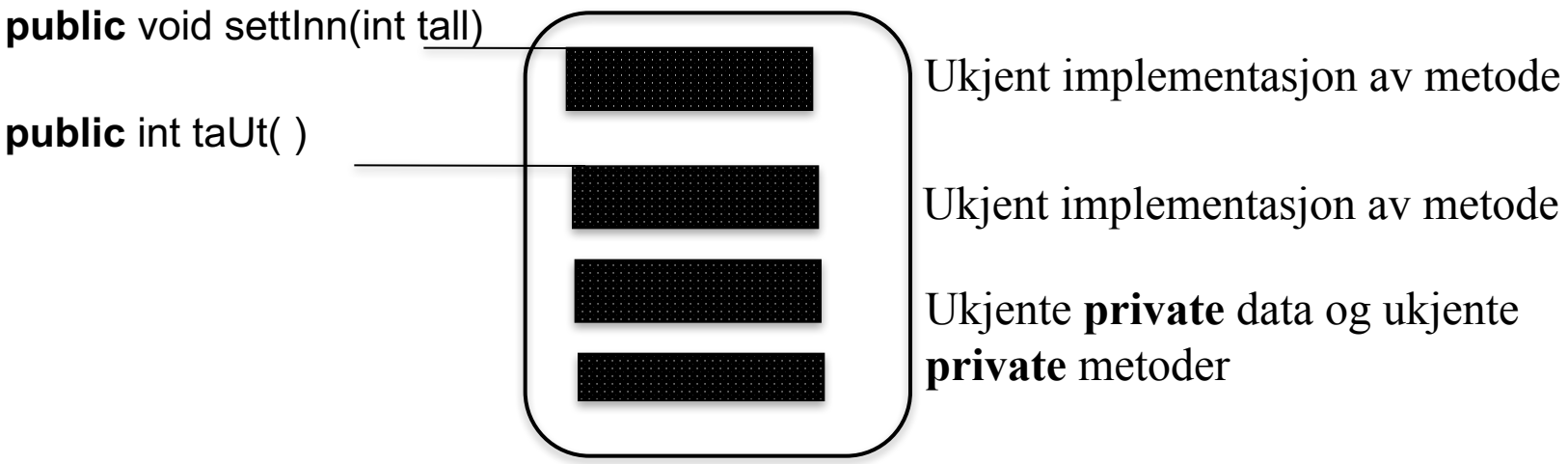
Det er ikke naturlig at Skatt og Miljo skal **kreve** en "skriv"-metode (?)

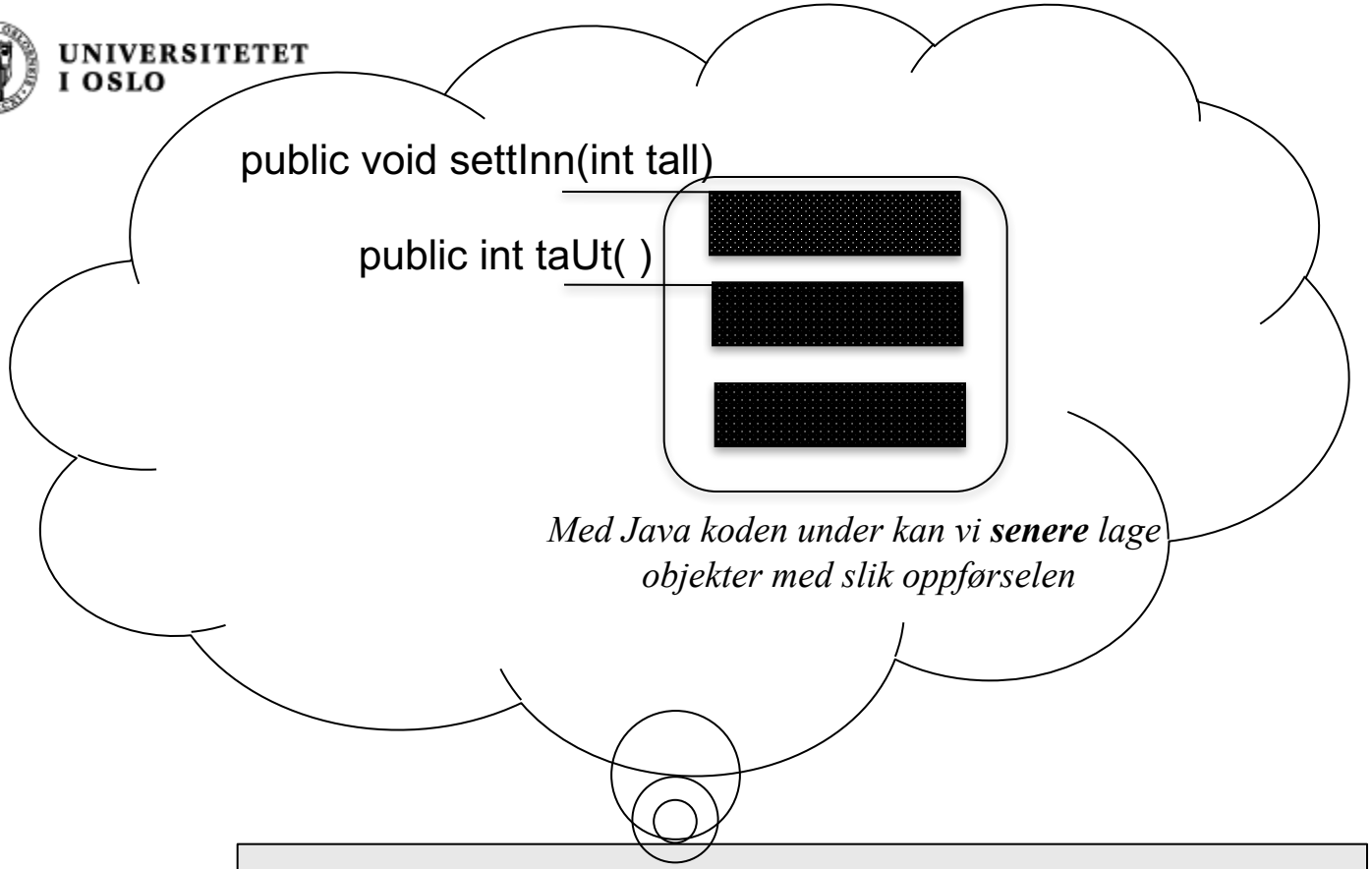
Skriv i LastebilMedSkattogMiljo



Objektorientering handler om å tydeliggjøre objektenes public-metoder

Husker dere forelesingen om enhetstesting:





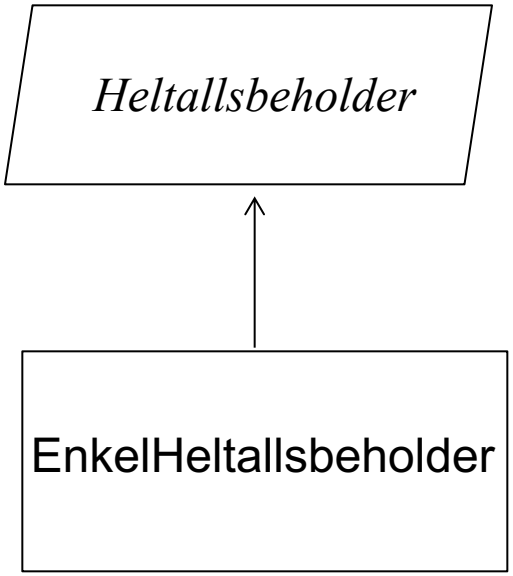
*Med Java koden under kan vi **senere** lage objekter med slik oppførselen*

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

Java kode

~~new Heltallsbeholder()~~

Interface: Klassehierarki og Java-kode



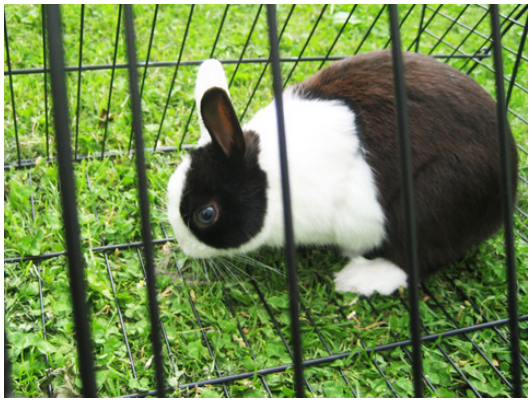

```
interface Heltallsbeholder {
        public void settInn(int tall);
        public int taUt( );
}

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

Når en klasse implementerer et interface tegner vi det nesten på samme måte som en superklasse / subklasse. For å markere at "superklassen" ikke er det, men et interface, kan vi enten skrive "interface" i boksen, og/eller vi kan gjøre navnet på interfacet (og boksen ?) kursiv.

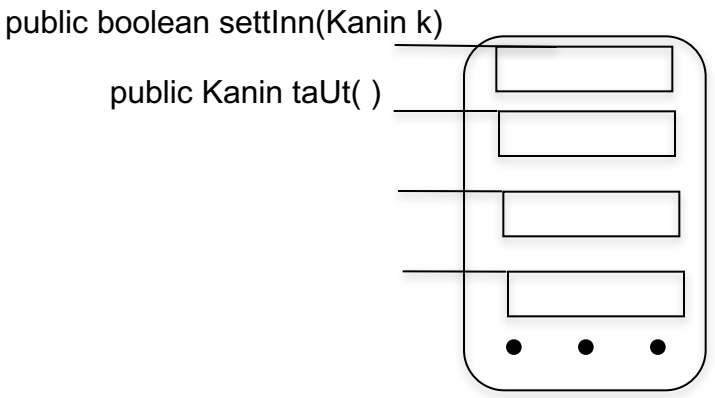
Engelsk: Interface **Norsk: Grensesnitt**

Vi kan også se på et kaninbur som et sted for kaninoppbevaring


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



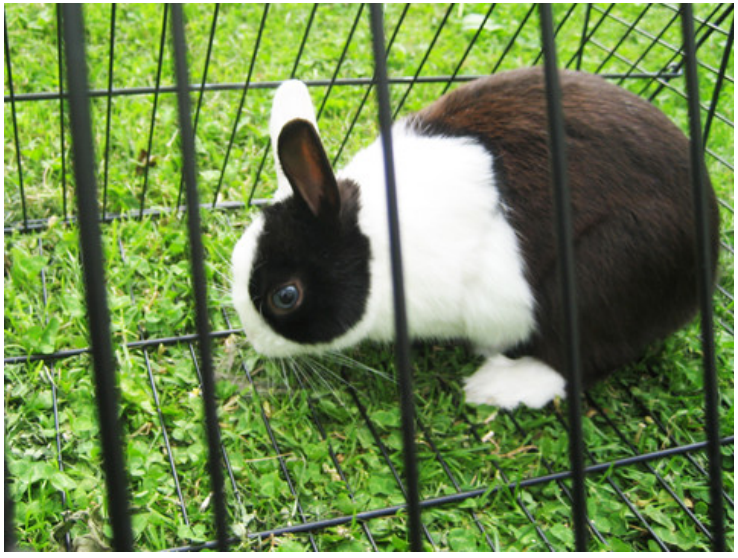
```
class Kaninbur  implements KaninOppbevaring{
    private Kanin denne = null;
    public boolean settInn(Kanin k) {
                        . . .
    }
    public Kanin taUt( ) {
                        . . .
    }
}
```



Et objekt av en klasse som implementerer grensesnittet KaninOppbevaring

Full kode


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

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



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

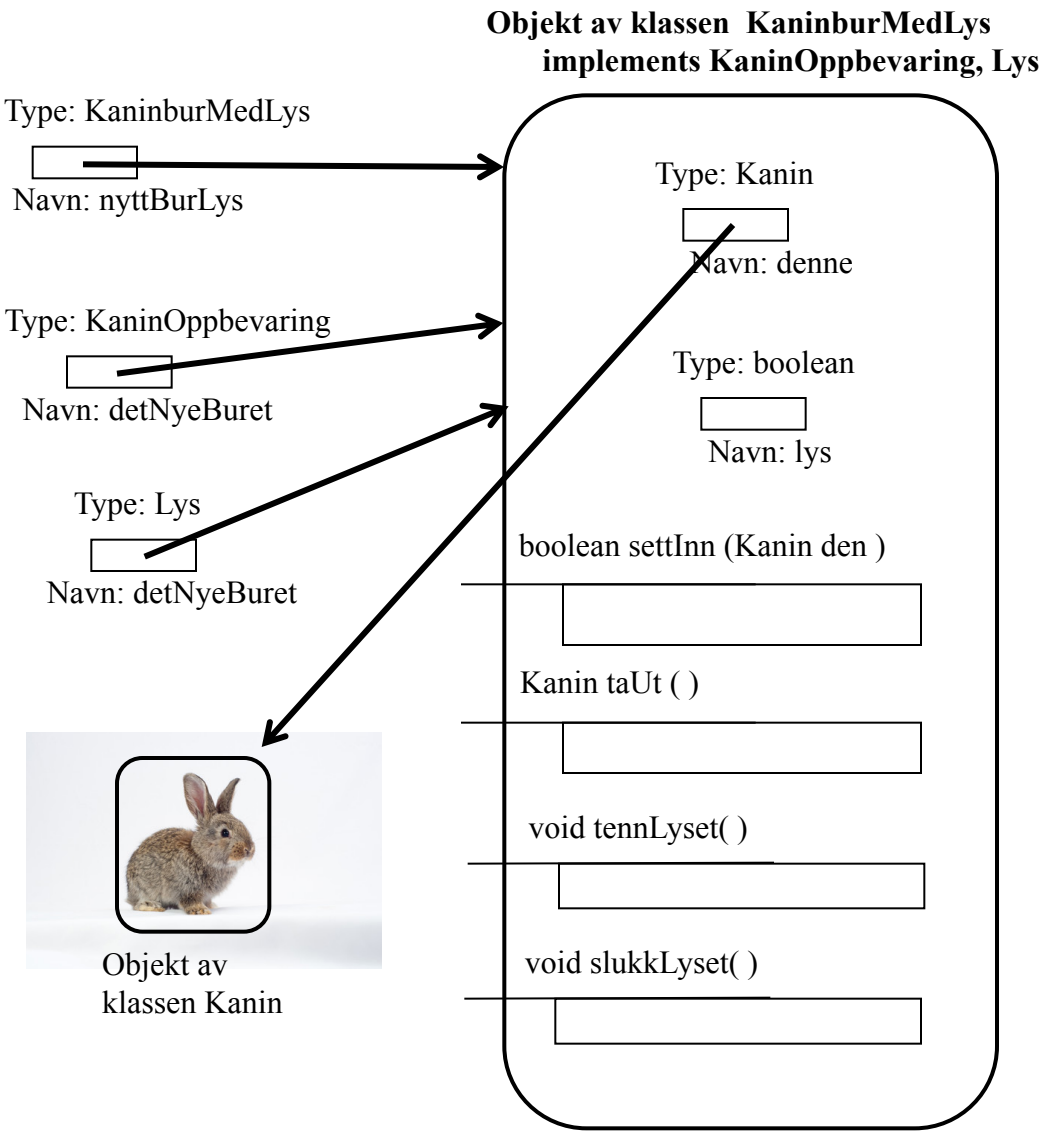
Vi kan lage kassen KaninburMedLys på denn måten: Én klasse – to grensesnitt


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

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

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

Ett objekt– to grensesnitt – tre briller




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

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

Vi kan se på objektet både med KaninburMedLys -briller og med KaninOppbevaring -briller og med Lys -briller

Forskjellige briller = forskjellige roller

Én klasse – to grensesnitt: Full kode


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

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

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

Flere eksempler: En klasse – mange grensesnitt

```
class Hund {protected double vekt;}

interface KanBjeffe{
      void bjeff();
}
interface Utkledd {
      int antallFarger();
}

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

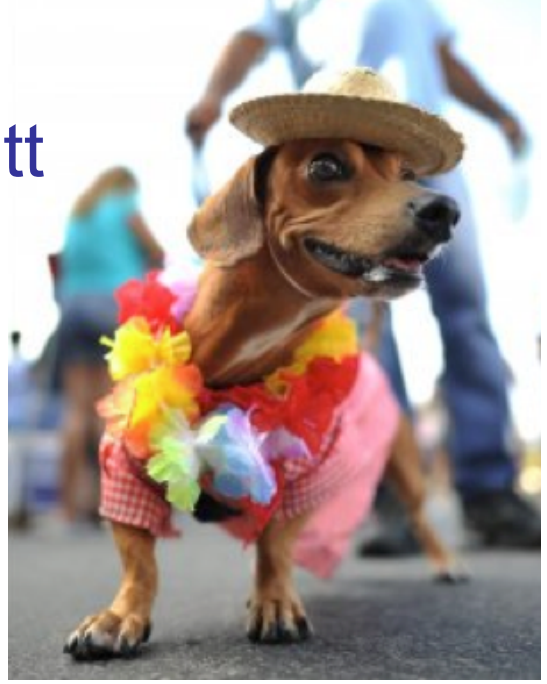
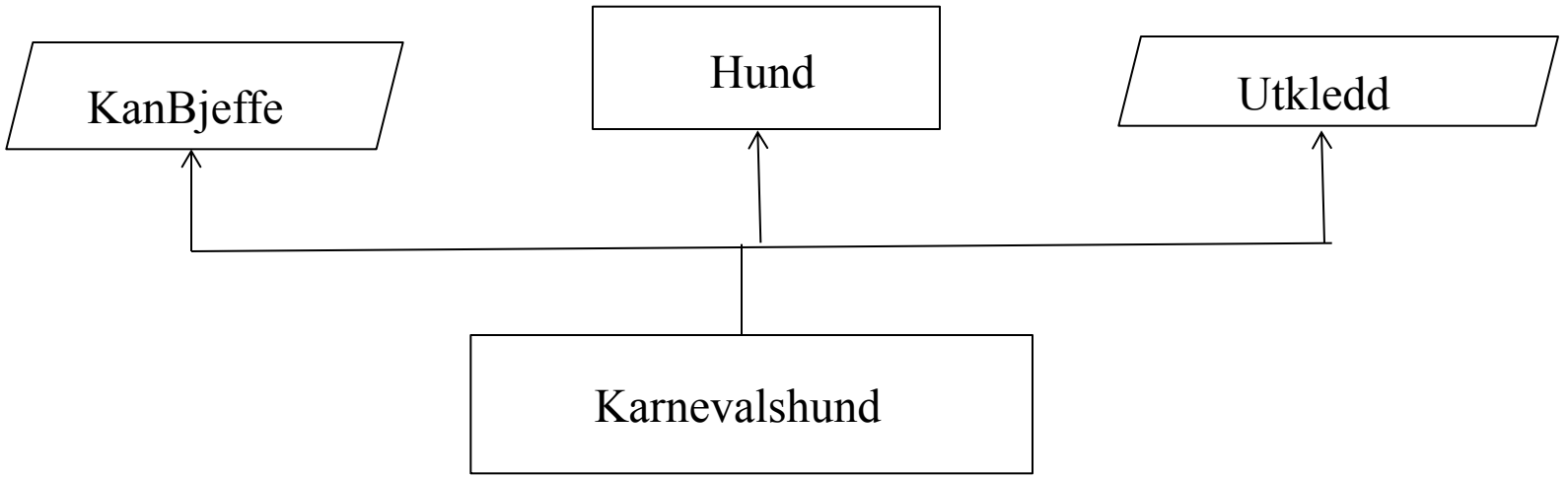


Foto: AP

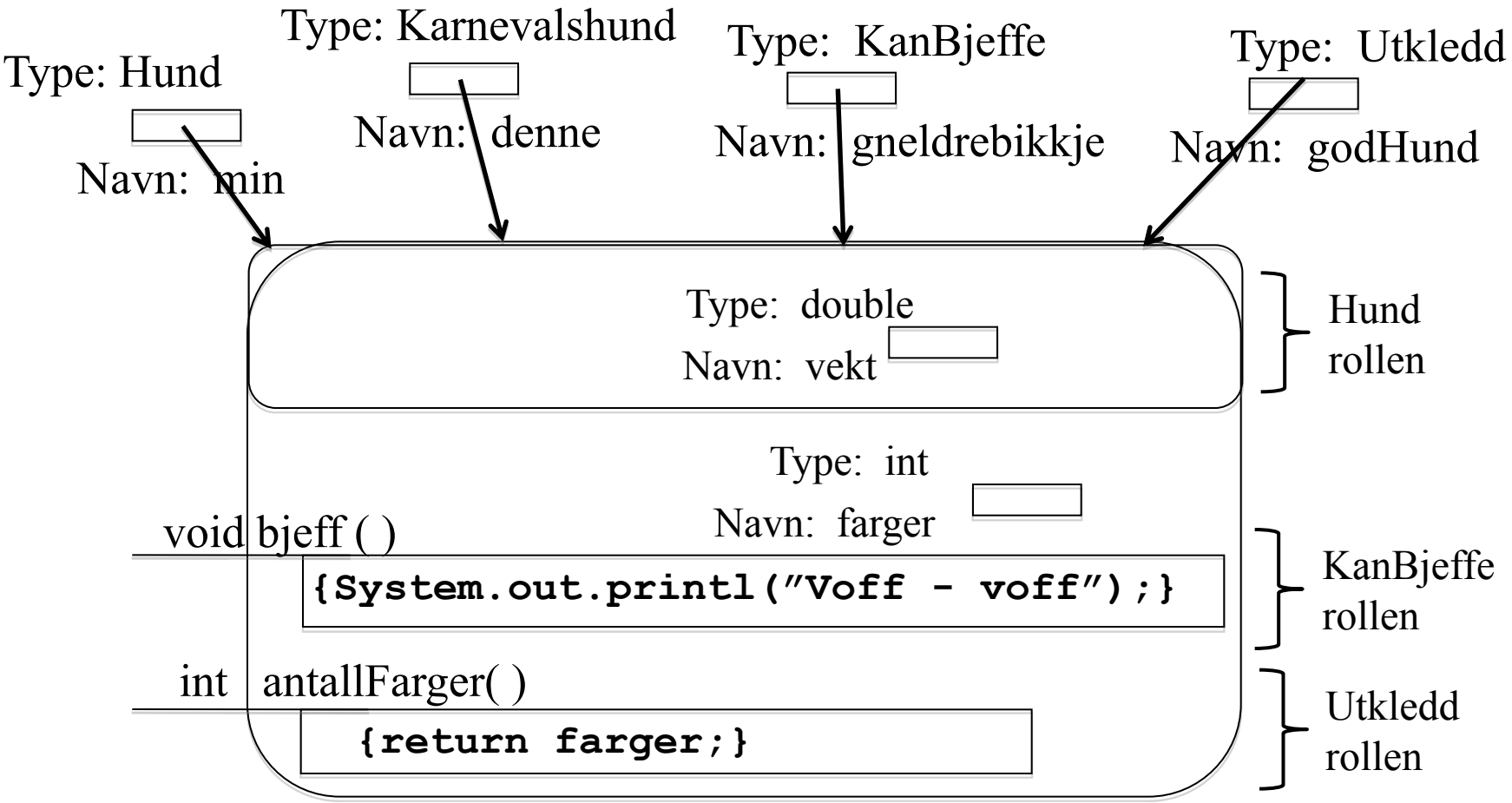
To (eller flere) grensesnitt = to (eller flere) roller



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

Multippel arv

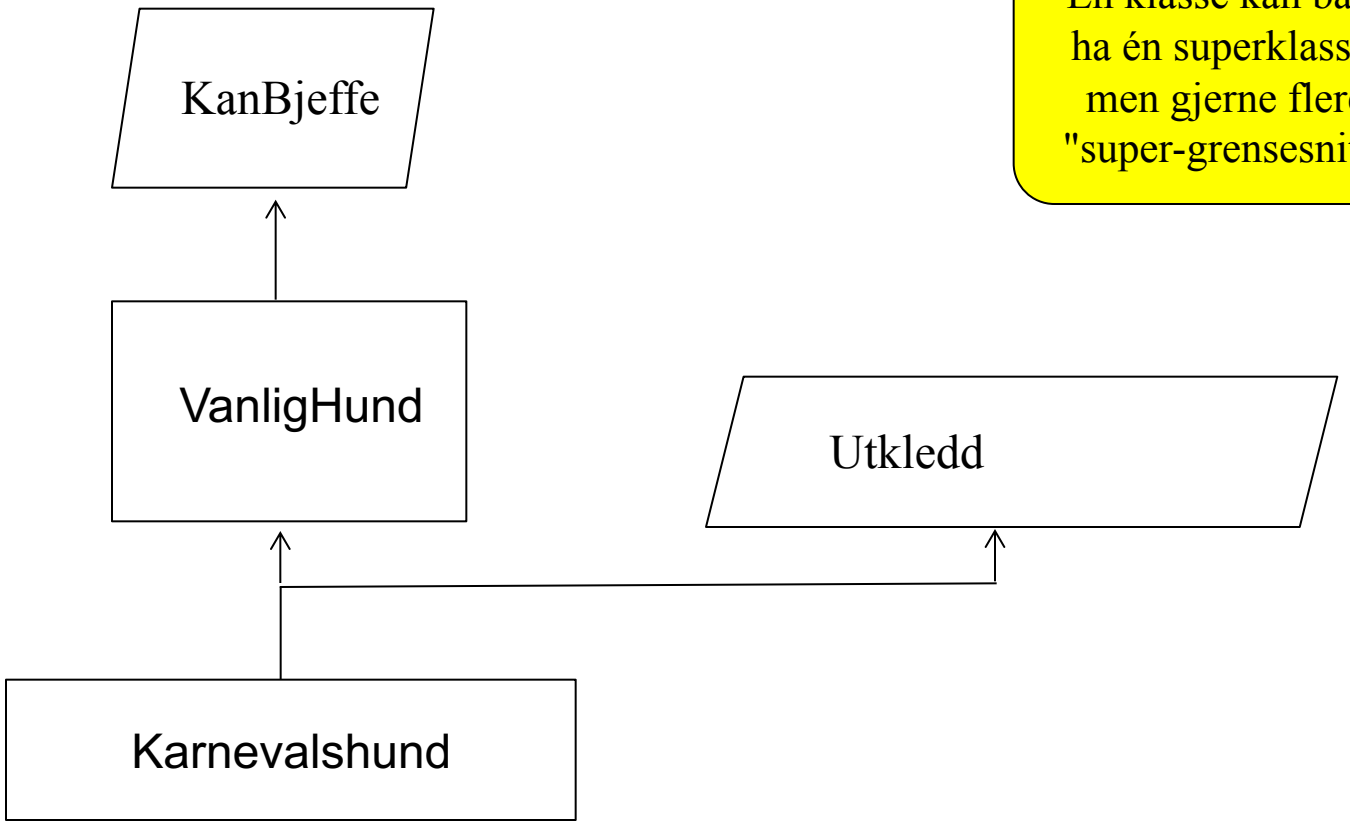
```
Karnevalshund   passopp   = new Karnevalshund( ):
Hund min                  = passopp;
KanBjeffe   gneldrebikkje = passopp;
Utkledd     godHunden     = passopp;
```



Objekt av klassen Karnevalshund

Eller kanskje bedre:

En klasse kan bare ha én superklasse, men gjerne flere "super-grensesnitt"



Nytt
(kanskje bedre hierarki)

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

Foto: AP

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

interface Utkledd {
        int antallFarger();
}

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

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

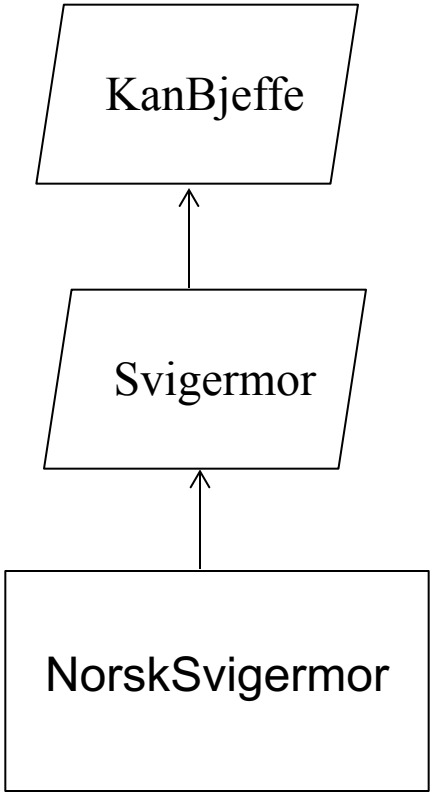
Enda et eksempel :


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

interface Svigermor{
        boolean oKPaaBesok();
}

class NorskSvigermor extends KanBjeffe, Svigermor {
       boolean hyggelig = false;
       public void bjeff( ) {
              System.out.println("Uff – uff");
       }
       public boolean oKPaaBesok() {
              return hyggelig;
       }
}
```

Oppgave: Tegn opp et objekt av klassen NorskSvigermor og tre pekere av forskjellig type.
Hvilke roller kan dette objektet spille ?
Hva ser vi ved hjelp av de forskjellige pekerene?

Alternativ svigemor: Arv fra grensesnitt til grensesnitt

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

interface Svigermor extends KanBjeffe {
        boolean oKPaaBesok();
}


class NorskSvigermor implements Svigermor {
         protected boolean hyggelig = false;
         public void bjeff( ) {
                  System.out.println("Uff – uff");
         }
        public boolean oKPaaBesok() {
                  return hyggelig;
         }
}
```

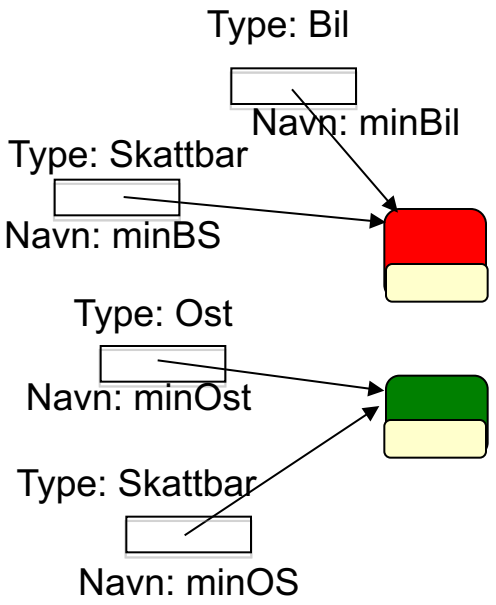


Ett grensesnitt, flere klasser: Eksempel: Både biler og ost skal skattlegges


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

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

class Ost  implements Skattbar {  // Ost: 200% skatt
        private int importprisPrKg;
        private int antKg;
        Ost (. . . ) {
                . . .
        }
        public int skatt( ){return importprisPrKg*antKg*2.00;}
}
```

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

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

int bilskatt = minBil.skatt();
int osteskatt = minOst.skatt();
int skatt = bilskatt + osteskatt;

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



Rollen Skatt

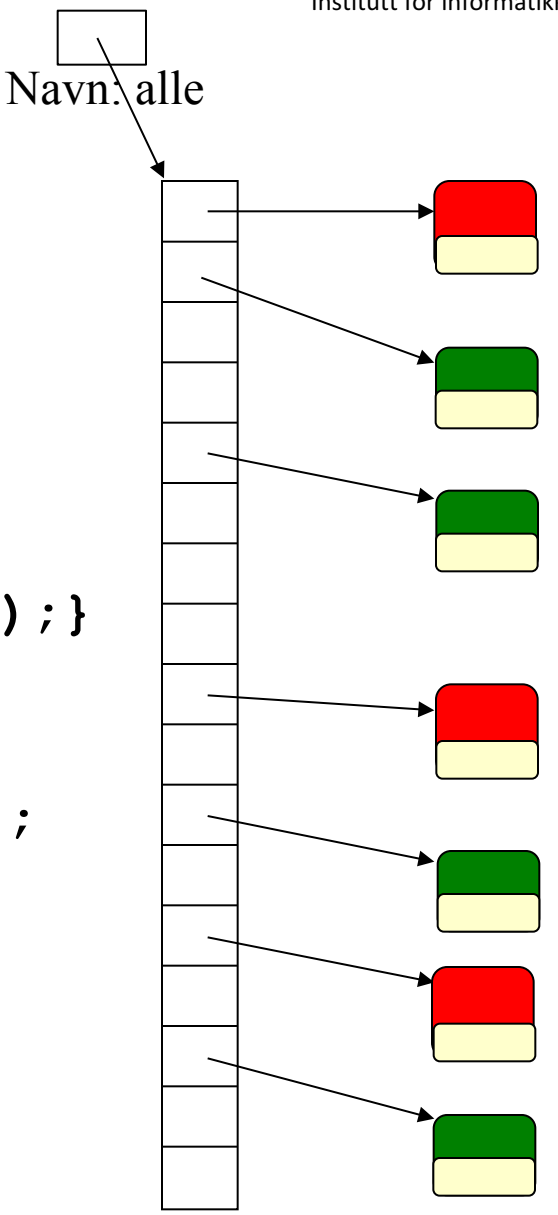
Rollen Bil (untatt Skatt)

Rollen Ost (untatt Skatt)

Generalisering på neste side

Samlet import-skatt


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

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

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



Rollen Skatt

Rollen Bil (untatt Skatt)

Rollen Ost (untatt Skatt)

Veldig viktig og bra eksempel. Dagens rosin.

Bil og Ost – Full kode


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

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

class Ost  implements Skattbar {  // Ost: 200% skatt
        private int importprisPrKg;
        private int antKg;
        Ost (int kgPris, int mengde) {
                importprisPrKg = antKg; antKg = mengde;
        }
        public int skatt( ){return importprisPrKg*antKg*2.00;}
}
```

Generiske klasser

- Se egne lysark om generiske klasser

Generiske interface
Inteface med parametre

- På samme måte som klasser, kan interface lages med parametre.
- FØR så vi: class GeneriskBeholderTilEn <E> { . . . }
- NÅ skal vi lære: interface Beholder <E> { . . . }

```
new GeneriskBeholderTilEn<Bil>( );
```

~~`new Beholder<Bil>( );`~~

Før:

```
class GeneriskBeholderTilEn <T> {
        T   denne;
        public void settInn (T  en) { denne = en;}
        public T  taUt ( ) {return denne;}
}
```

Nå:

```
interface Beholder <T> {
        public void settInn (T  en);
        public T  taUt ( );
}



class GeneriskBeholderTilEn <T> implements  Beholder <T> {
        T   denne;
        public void settInn (T  en) { denne = en;}
        public T  taUt ( ) {return denne;}
}
```

Generisk grensesnitt

Mer grensesnitt med parametre (Generiske grensesnitt)

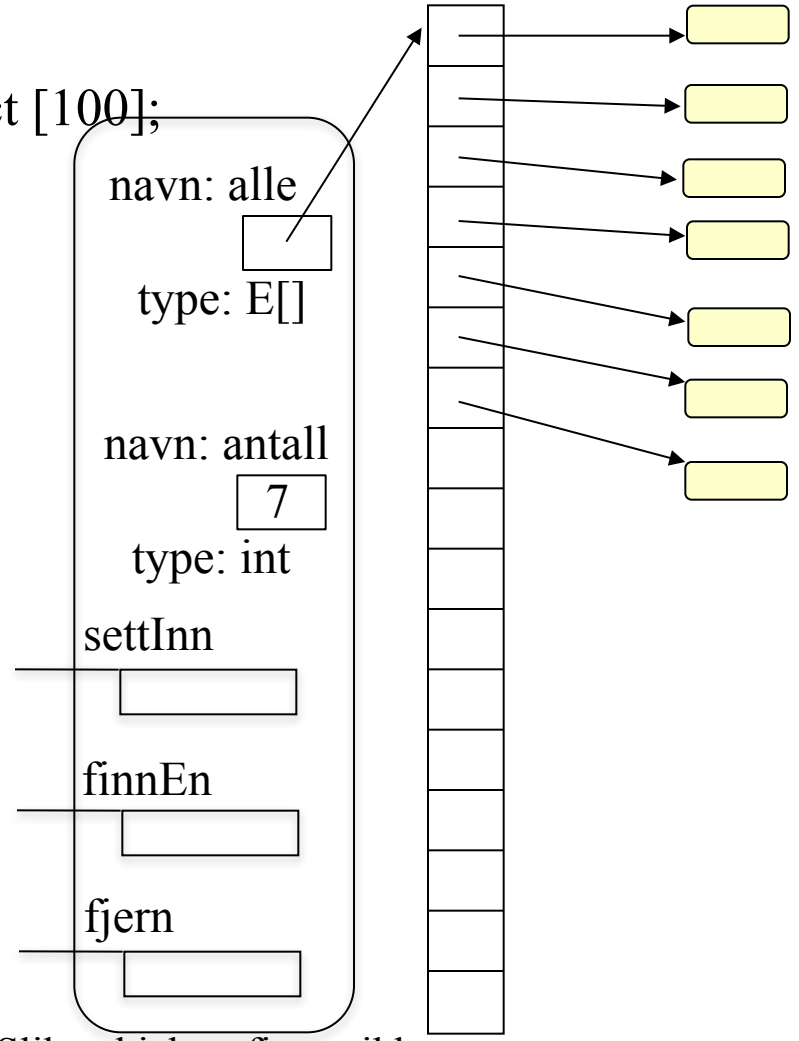

```
interface EnkelStorBeholder <E> {
        public void settInn (E elem);
        public E  finnEn( );
        public void fjern( );
}
```

Her ønsker vi å lage et grensesnitt til en beholder som kan ta vare på mange elementer (mange objekter av klassen E).

Men hva er sematikken til metodene / til grensesnittet?
Hvilken semantikk ønsker vi egentlig?
Hva ønsker vi å kreve av klassene som implementerer dette grensesnittet?

Forslag til implementasjon

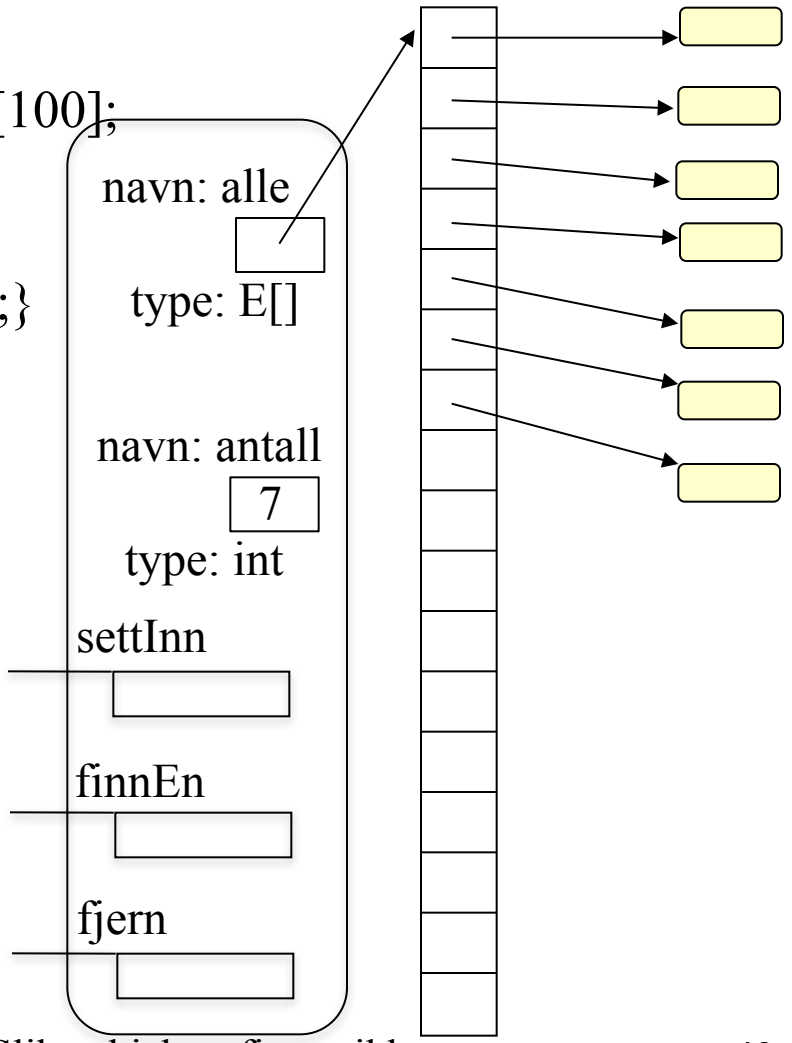

```
class KonkretEnkelStorBeholder <E>
                    implements EnkelStorBeholder<E>
{
        private E [ ] alle = (E [ ] ) new Object [100];
        private int antall = 0;
        public boolean settInn(E  elem) {
                . . .
                . . .
                . . .
                . . .
        }
        public E finnEn( ) {
                . . .
                . . .
        }
        public void  fjern( ) {
                . . .
        }
}
```



Slike objekter finnes ikke
(må gi E en verdi først)

Implementasjon: Full kode


```
class KonkretEnkelStorBeholder <E>
                implements EnkelStorBeholder<E>
{
        private E [ ] alle = (E [ ] ) new Object [100];
        private int antall = 0;
        public boolean settInn(E  elem) {
                if (antall ==100) {return false;}
                alle[antall] = elem;
                antall ++;
                return true;
        }
        public E finnEn( ) {
                if (antall == 0) {return null;}
                return alle[antall-1];
        }
        public void  fjern( ) {
                if(antall != 0) {antall -- ;}
        }
}
```

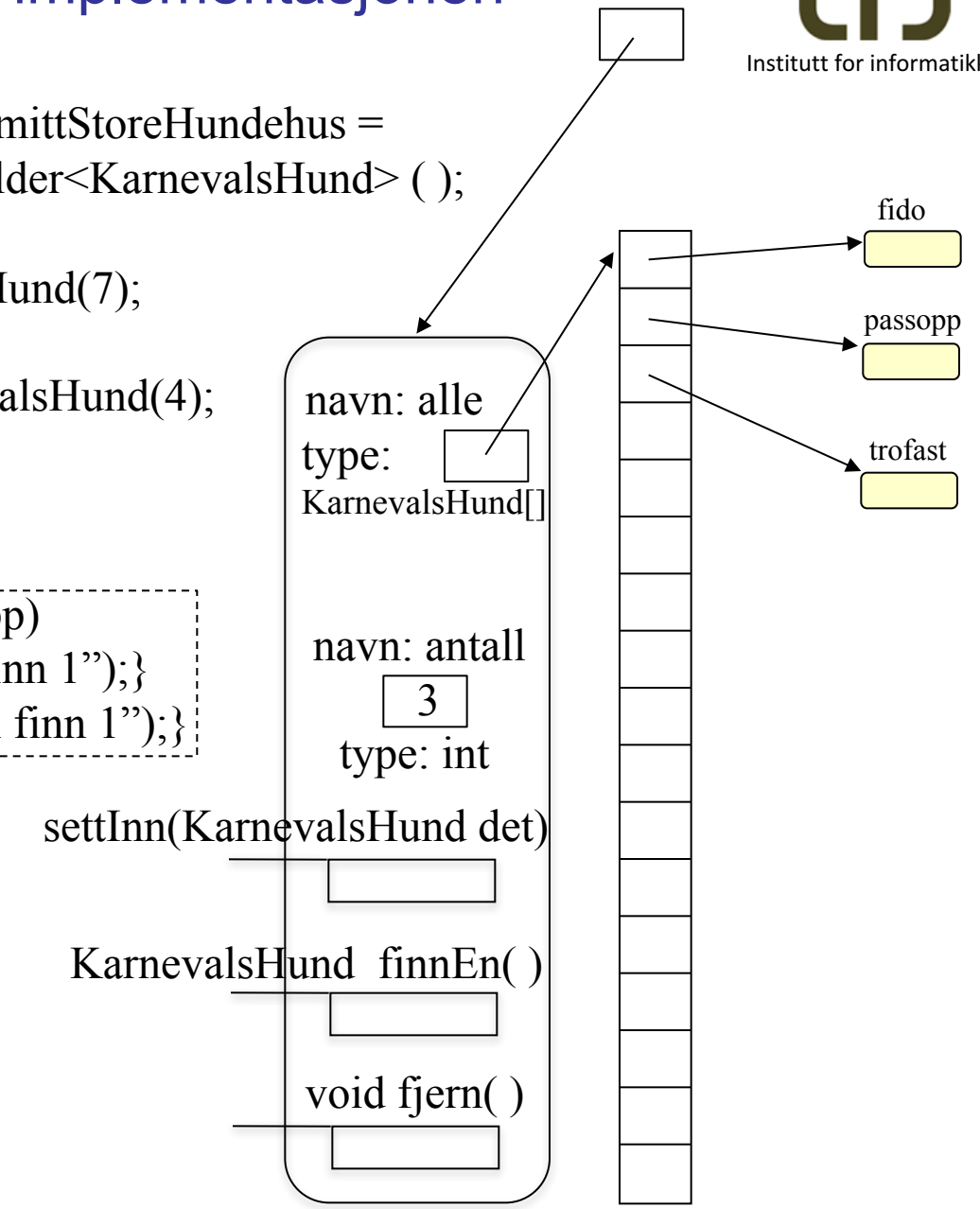


Slike objekter finnes ikke
(må gi E en verdi først)

Bruk av denne implementasjonen


```
EnkelStorBeholder<KarnevalsHund>   mittStoreHundehus =
            new KonkretEnkelStorBeholder<KarnevalsHund> ( );

KarnevalsHund fido =  new KarnevalsHund(7);
mittStoreHundehus.settInn(fido);
KarnevalsHund passopp =  new KarnevalsHund(4);
mittStoreHundehus.settInn(passopp);
KarnevalsHund  enHund =
            mittStoreHundehus.finnEn( );
if (enHund == fido || enHund == passopp)
            {System.out.println("Riktig finn 1");}
            else {System.out.println("Feil finn 1");}
```

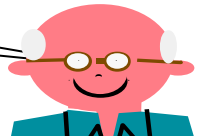
• • •



Slike objekter finnes

Enda en gang:

Hva brukes **interface** til?



Vet hjelp av interface kan forskjellige klasser og objekter ha det samme grensesnittet. Dette er en fordel når vi skal beskrive objekter med felles egenskaper.

Et interface kalles gjerne også en **rolle** (som en subklasse)

- Noen objekter kan spille flere forskjellig roller (snart: multippel arv)
- Forskjellige objekter kan implementere samme rolle på forskjellige måter
  - innkapsling = skjuling av detaljer

MYE MER I EKSEMPLER HER OG SENERE