



IN1010 våren 2022

Tirsdag 15. februar

Subklasser III: Interface - Grensesnitt + Litt om invarianter

Stein Gjessing



Dagens hovedtema

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

- Les notatet “Interface i Java” av Stein Gjessing

- To måter å bruke interface på (men det er mye av det samme)
 - 1) Multippel arv av oppførsel / roller
 - 2) Definere en oppførsel / rolle

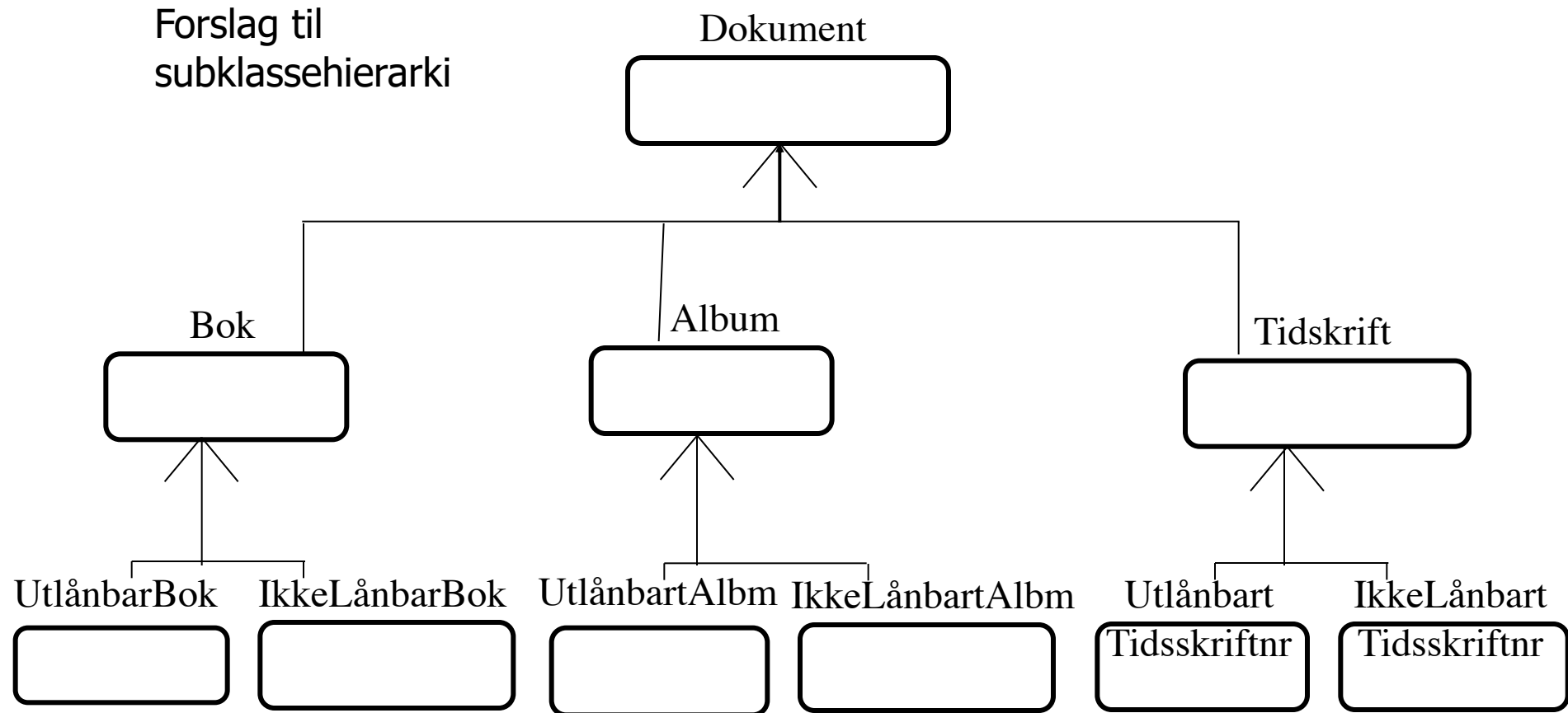
Multippel arv: Om å arve fra flere

- 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 mekanismen interface: Analyse av et bibliotek

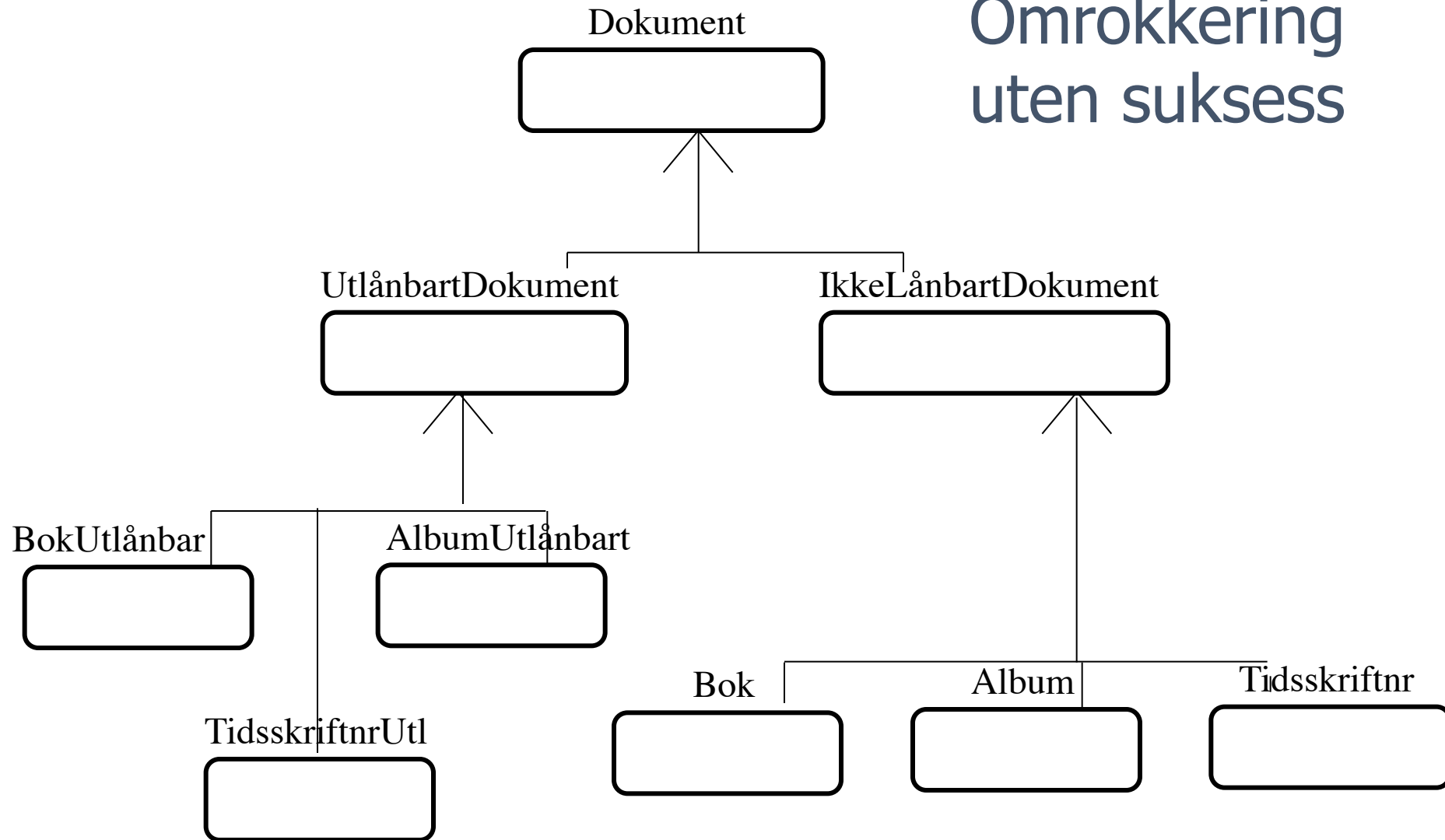
- Bøker, tidsskrifter, musikk, videoer, mikrofilmet materiale, antikvariske bøker, flerbindsverk, oppslagsverk, upubliserte skrifter, ...
- En del felles egenskaper
 - antall eksemplarer, hylleplass, identifikasjonskode (Dokument)
 - for det som kan lånes ut: Er utlånt ? , navn på låner, ... (TilUtlån)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

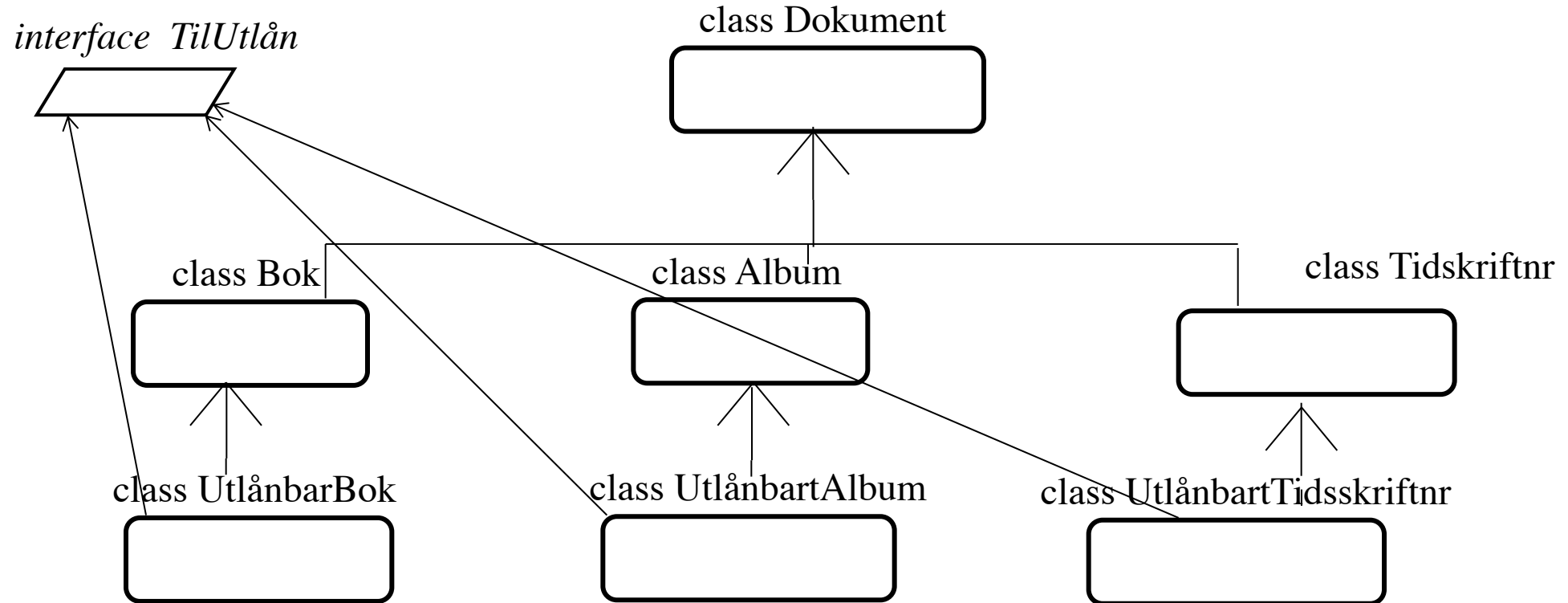
Tvilsomt begrepshierarki





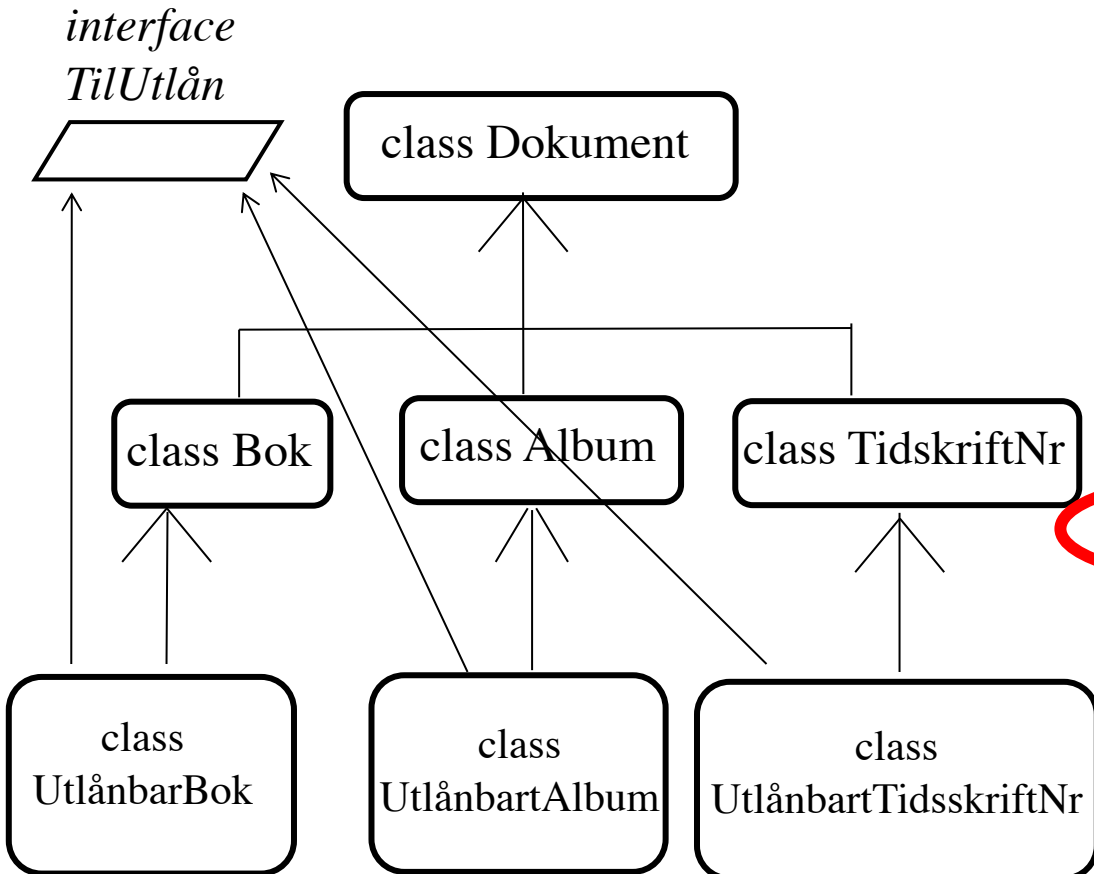
Omrokking uten suksess





- Felles egenskaper (rolle) på tvers av klassehierarkiet
- 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**

Vi tar det en gang til, litt saktere: Dokumentene i et bibliotek



```
class Dokument { }

class Bok extends Dokument { }

class Album extends Dokument { }

class TidsskriftNr extends Dokument { }

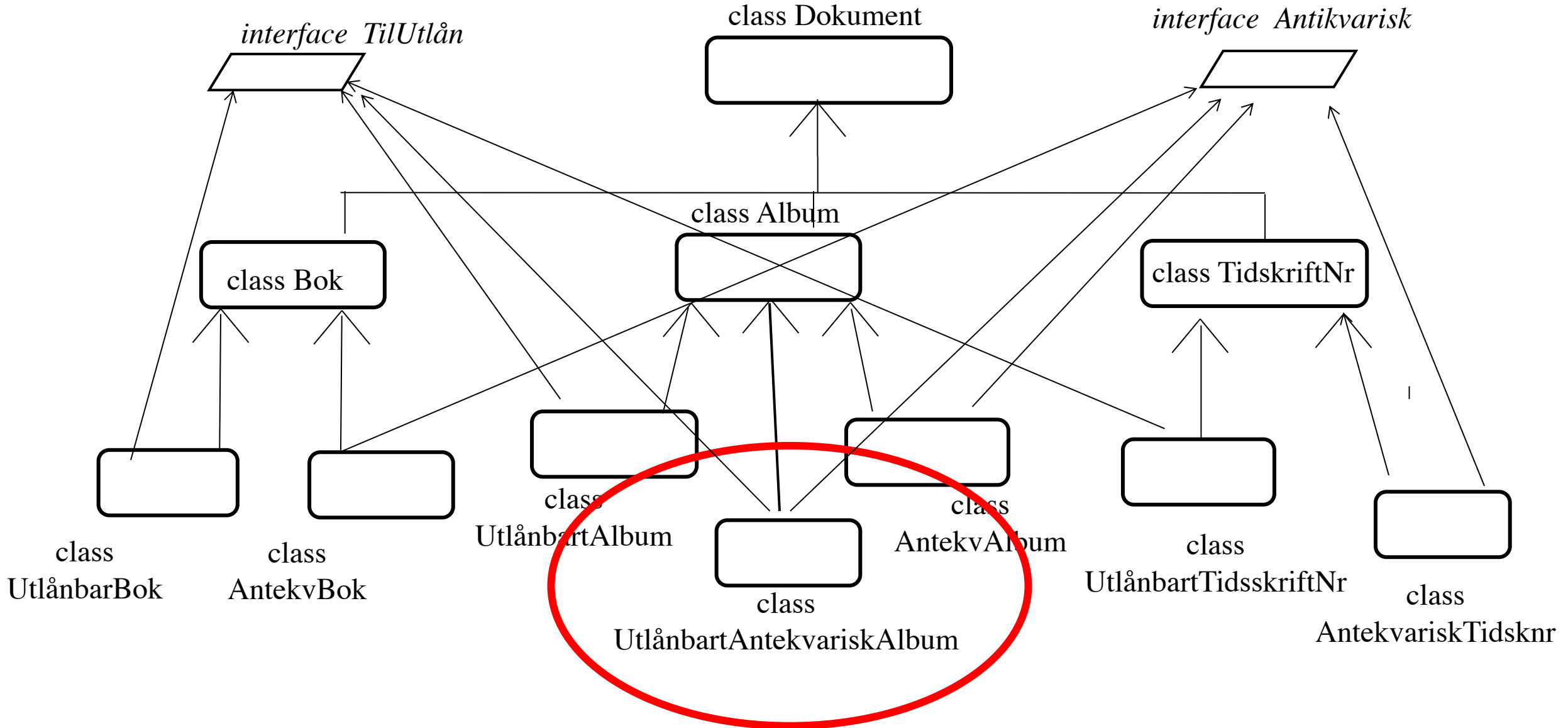
interface TilUtlån { }

class UtlånbarBok extends Bok implements TilUtlån { }

class UtlånbartAlbum extends Album implements TilUtlån { }

class UtlånbartTidsskriftNr extends TidsskriftNr implements TilUtlån { }
```


En klasse kan arve fra flere **interface**





Men hva arves fra et interface?

```
class Dokument { }

class Bok extends Dokument { }

class Album extends Dokument { }

class TidskriftNr extends Dokument { }

interface TilUtlån { }

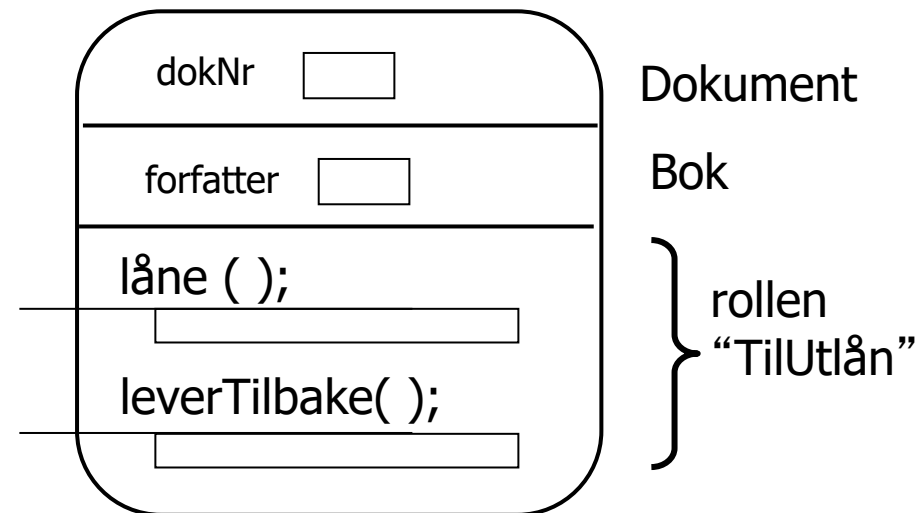
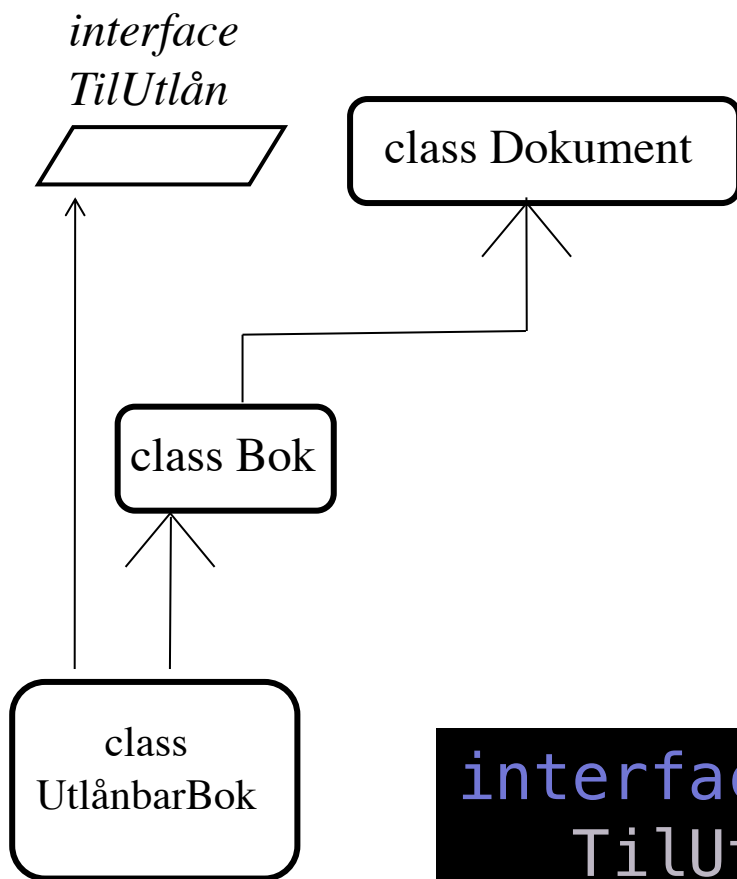
class UtlånbarBok extends Bok implements TilUtlån { }

class UtlånbartAlbum extends Album implements TilUtlån { }

class UtlånbartTidskriftNr extends TidskriftNr implements TilUtlån { }
```

Men hva arves fra et grensesnitt?

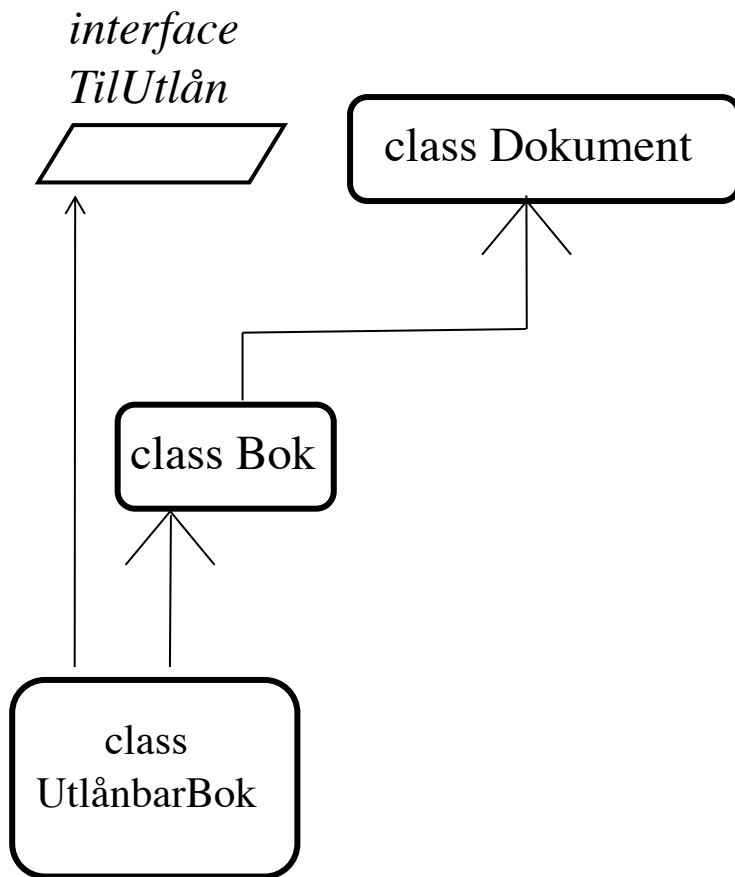
Svar: En rolle (metoder)



Et objekt av klassen
UtlånbarBok

```
interface TilUtlån {  
    TilUtlån låne(String låner);  
    void leverTilbake();  
}
```

Svar: En rolle (metoder – metodesignaturer)



```
class Dokument {final int dokNr;}

class Bok extends Dokument {final String forfatter;}

interface TilUtlån {
    TilUtlån låne(String låner);
    void leverTilbake();
}

class UtlånbarBok extends Dokument implements TilUtlån {
    private String låner = null;
    @Override
    public TilUtlån låne(String lnr) { . . . }
    @Override
    public void leverTilbake() { . . . }
}
```



Subklassen må selv *implementere* metodene

Svar: En rolle (metoder – metodesignaturer)

```
class Dokument {final int dokNr;}

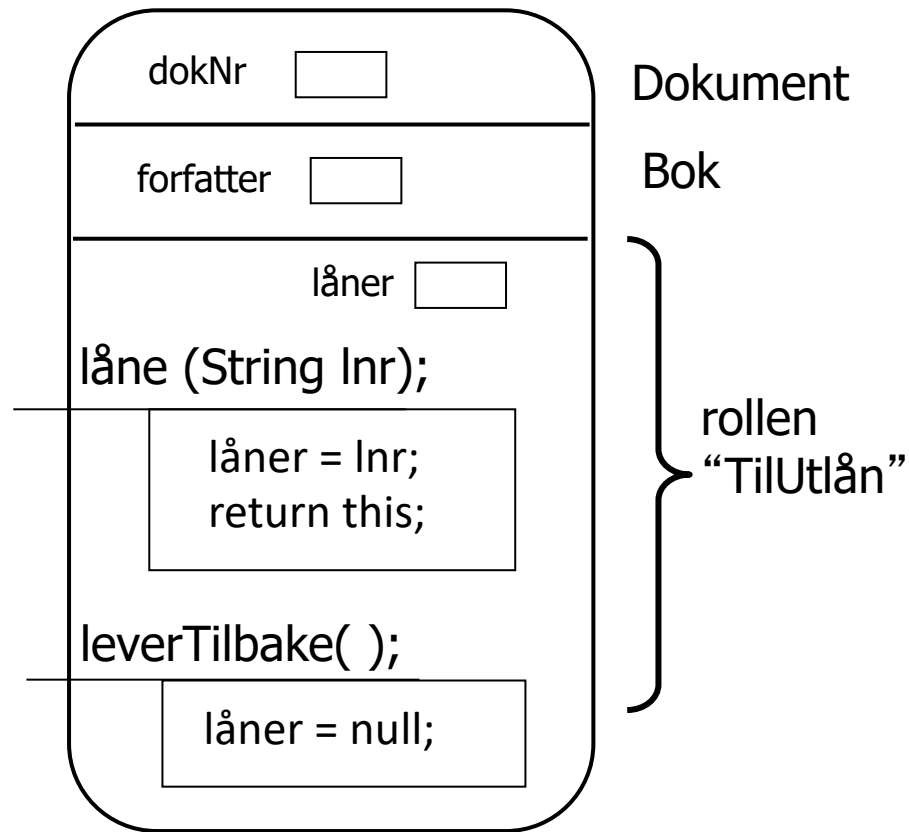
class Bok extends Dokument {final String forfatter;}

interface TilUtlån {
    TilUtlån låne(String låner);
    void leverTilbake();
}

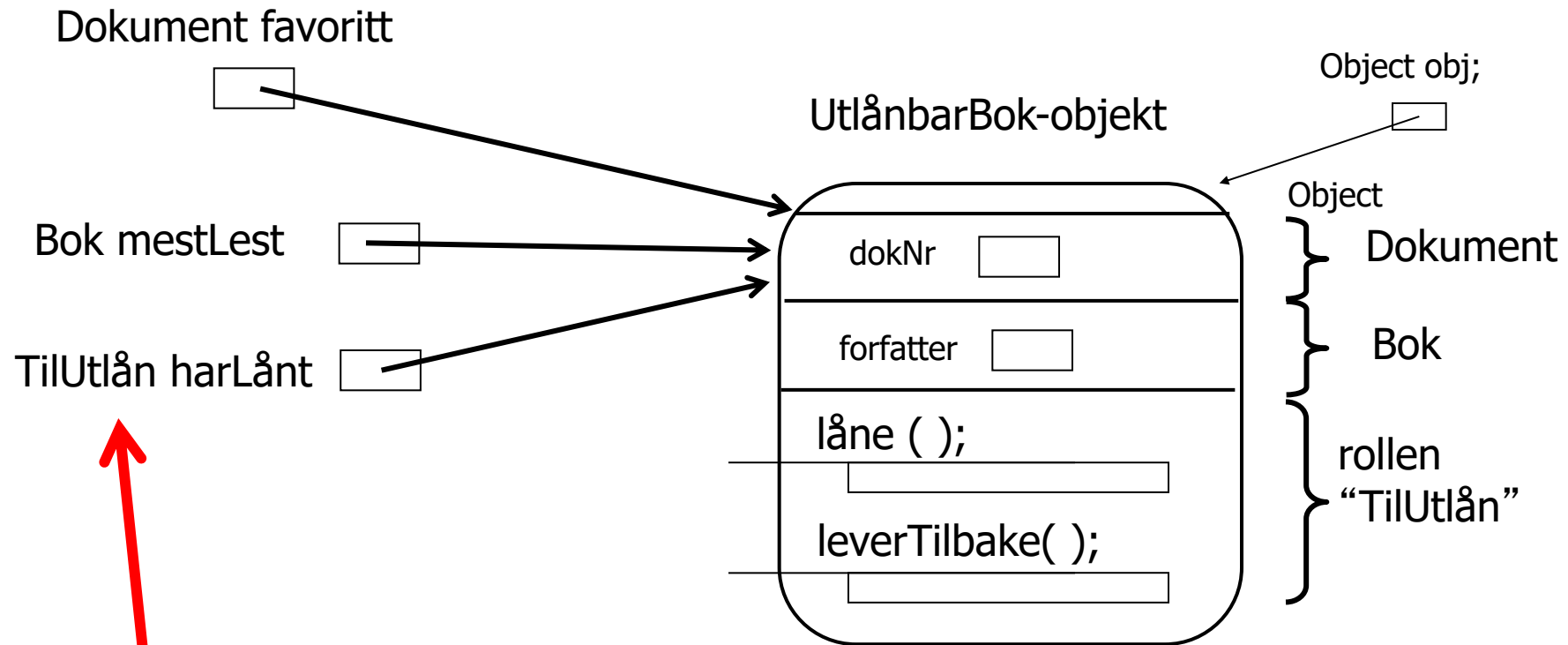
class UtlånbarBok extends Bok implements TilUtlån {
    private String låner = null;

    public TilUtlån låne(String lnr) {
        låner = lnr;
        return this;
    }

    public void leverTilbake() {låner = null;}
}
```



Et objekt av klassen
UtlånbarBok



Slik

NB! Vi kan ha pekere av interface-type

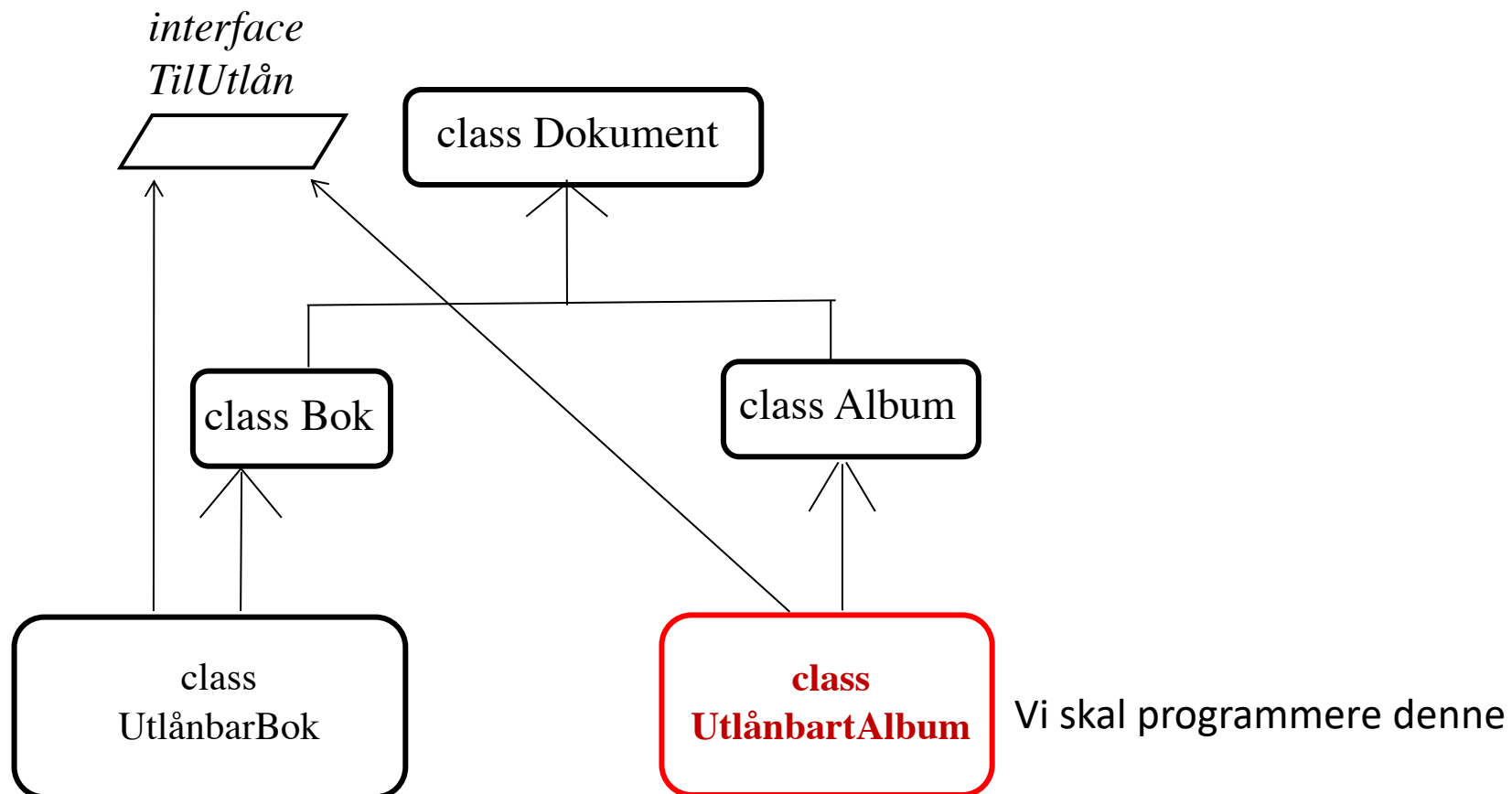
Et interface (grensesnitt) er (1):

- 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
 - Her: **Kan lånes ut (biler, bøker, festklær, ...)**
Antikvarisk (møbler, bøker,)
 - Kan delta i konkurranse (startnummer, resultat, ..
Mennesker, biler, hester kan delta i konkurranser)
 - Svømmedyktig (mennesker, fugler er svømmedyktige)
 - Sammenlignbar (Comparable) (I Java-biblioteket – kommer vi tilbake til)
 - ...

Et interface (grensesnitt) er (2):

- Et interface ligner en abstrakt klasse
- **Alle** metodene i et interface er abstrakte og polymorfe
- En interface inneholder **ingen** variabler eller annen datastruktur
(men litt annet som vi ikke bruker i IN1010)
- En klasse som arver egenskapene til et interface må selv putte inn kode i alle de abstrakte metodene (og deklarerer passende variabler 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
- Husk, med interface arves bare metodesignaturer
 - Implementasjon arves ikke



Kommentar lagt til etter forelesningen: Legg merke til hvordan Stein skriver Bok istedenfor Album som superklasse til UtlanbartAlbum i koden han skriver. Dette er en feil som blir rettet opp til slutt, men det er en programmeringsfeil som først ikke blir fanget opp av det lille testprogrammet. Moral: Programmet ditt kan lett inneholde feil som ikke blir oppdaget før du gjør meget grundig testing.



Nytt interface-eksempel

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();  
    double moms();  
}
```

Rollen Skattbar

```
interface Miljøvennlig {  
    int c02Utslipp ();  
    boolean svaneMerket ();  
}
```

Rollen Miljøvennlig



Tegning av interfacer i klassehierarkiet

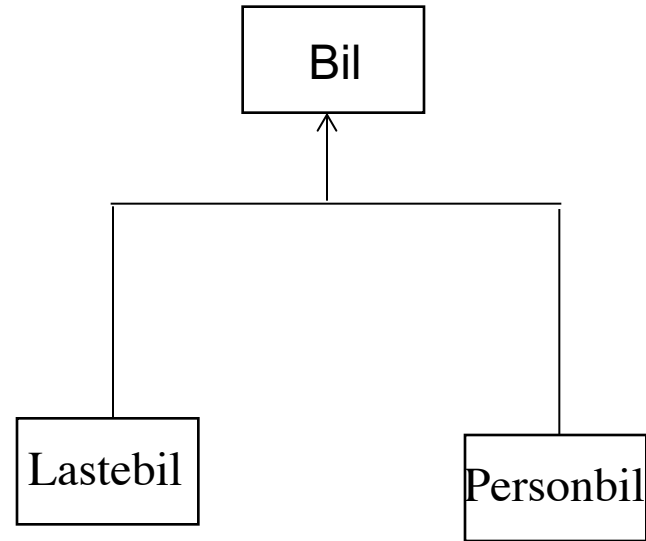
```
interface Skattbar {  
}
```

Skattbar

```
interface Miljøvennlig {  
}
```

Miljøvennlig

Slik var bil-hierarkiet

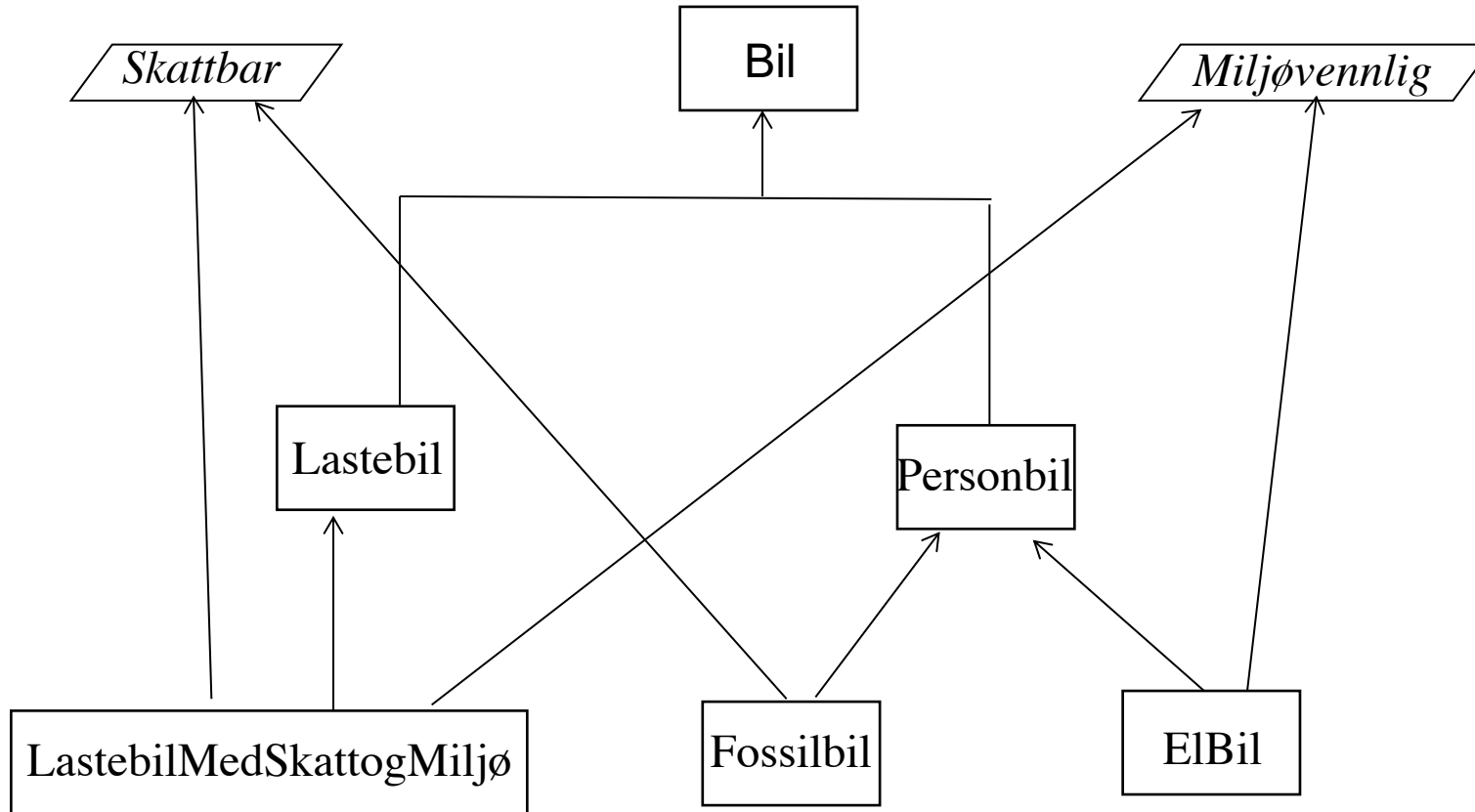


Mange biler må betale skatt og noen biler er miljøvennlige:

Skattbar

Miljøvennlig

Tre nye klasser som kan spille mange roller



Metodene må skrives på nytt hver gang de implementeres, men det er f.eks. OK når skatt i en lastebil er forskjellig fra skatt i en fossilbil



Om å implementere ett interface

rollen Personbil (i arv) fra klassehierarkiet

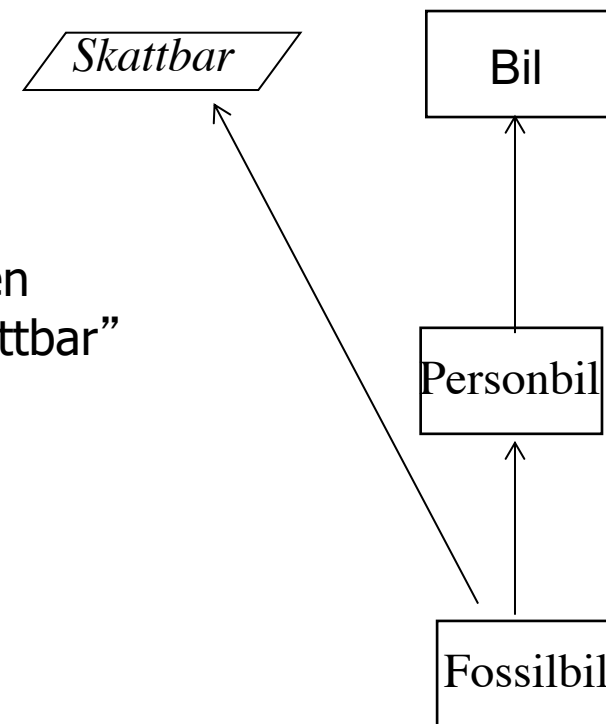
```
class Fossilbil extends Personbil implements Skattbar {
    protected double utslipp = 100;
    protected double innPris = 150000;
    @Override
    public double toll(){return (innPris*utslipp/200)* moms();}
    @Override
    public double moms( ){return 1.25;}
}
```

rollen
"Skattbar"

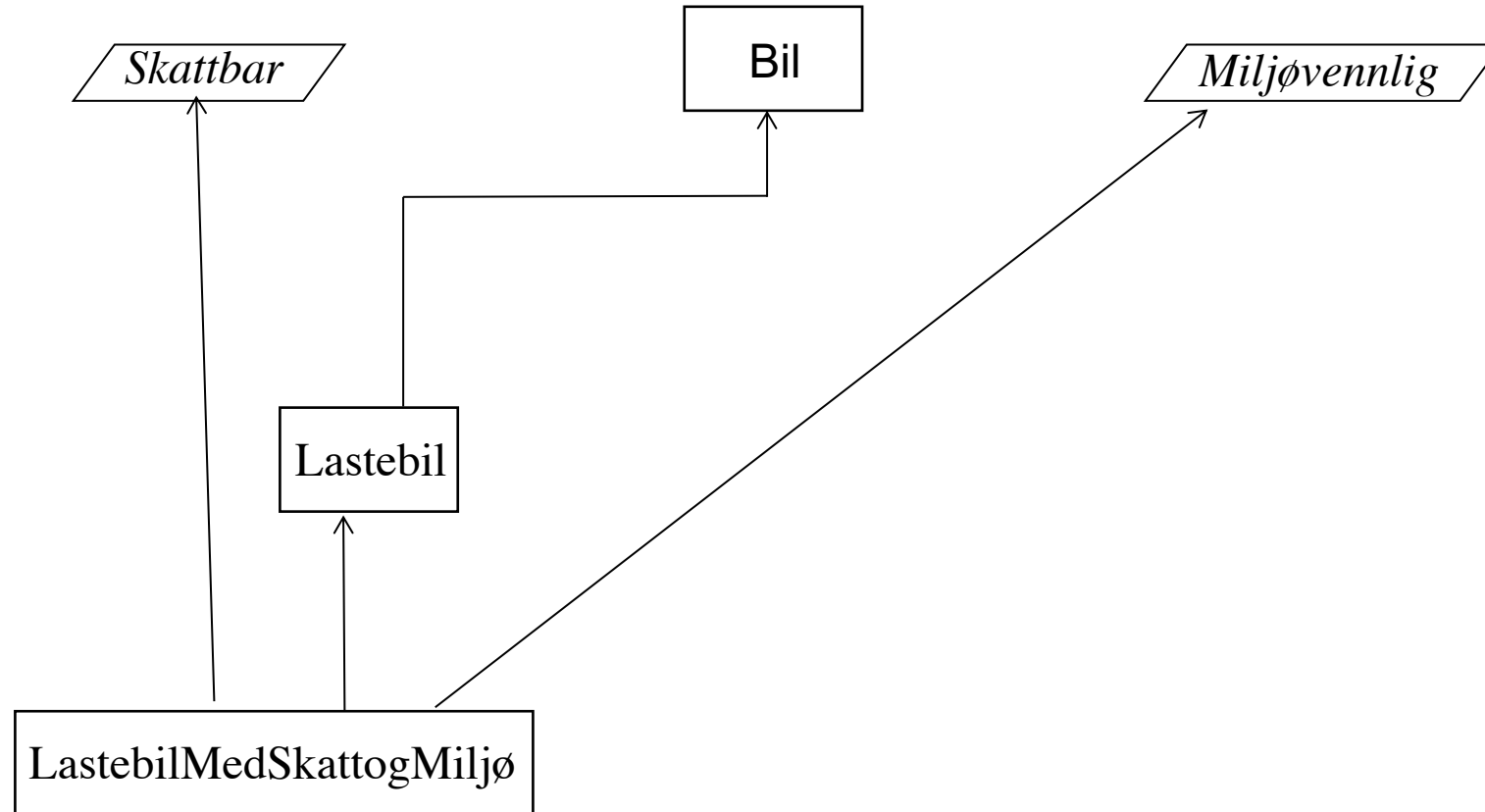
```
interface Skattbar {
    double toll();
    double moms();
}
```

```
class Bil {
    final String regNr;
}
```

```
class Personbil extends Bil {
    protected int antPass;
}
```



Om å implementere to interfacer



Om å implementere to interfacer

Lastebil-rollen i (arv fra) klassehierarkiet

```
class LastebilMedSkattogMiljø extends Lastebil implements Miljøvennlig, Skattbar {  
    protected double innkjøpspris = 200000;  
    protected int utslipp = 400;  
    @Override  
    public double toll(){return innkjøpspris*0.1;}  
    @Override  
    public double moms(){return 1.25;}  
    @Override  
    public int c02Utslipp(){return utslipp;}  
    @Override  
    public boolean svaneMerket(){return false;}  
}
```

rollen "Skattbar"

rollen "Miljøvennlig"

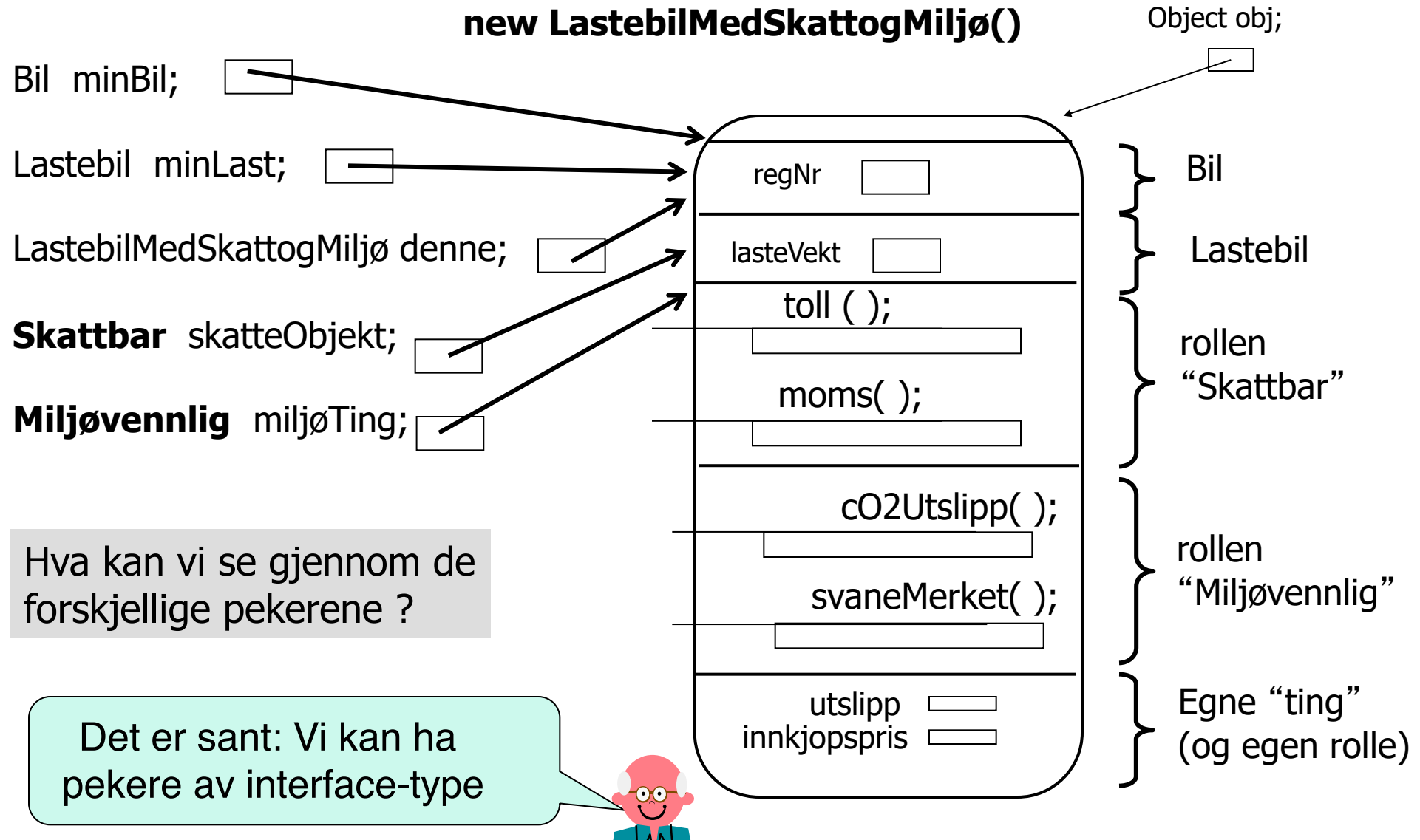
```
interface Skattbar {  
    double toll();  
    double moms();  
}
```

```
interface Miljøvennlig {  
    int co2Utslipp();  
    boolean svaneMerket();  
}
```

```
class Bil {  
    final String regNr;  
}
```

```
class Lastebil {  
    protected double lasteVekt;  
}
```


Et objekt og noen pekere



Enda mer om grensesnitt (interface)

- Navnet på et interface kan brukes som typenavn når vi lager referanser (det så vi på forrige side)
- Vitsen med et interface er 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 implementere (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.

- Vi har en **ny type**: interfacenavn:
«Skattbar» og «Miljøvennlig» er referansetyper

Interface lærdom

- Et interface har bare
 - metodenavn med parametre, men ikke kode (husk ;)
- 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
- **I informatikk, mer generelt kjent under navnet ADT = Abstrakt Data Type**
 - Vi definerer *hva* en ny datatype skal gjøre, *ikke hvordan* dette gjøres (derfor «Abstrakt»)
 - 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 {  
    protected String regNr;  
    public void skriv(){  
        System.out.println("Registreringsnummer: " + regNr);  
    }  
}
```

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

Skriv i LastebilMedSkattOgMiljo

```
class LastebilMedSkattOgMiljo extends Lastebil implements Skattbar, Miljøvennlig {
    protected double innkjopspris = 200000;
    protected int utslipp = 400;
    @Override
    public double toll( ) { return innkjopspris * 0.1; }
    @Override
    public double moms( ) {return 1.25;}
    public void skrivSkatt( ) {
        System.out.println("Innkjøpspris " + innkjopspris);
    }
    @Override
    public int c02Utslipp ( ) {return utslipp; }
    @Override
    public boolean svaneMerket ( ) { return false; }
    public void skrivMiljo( ) {
        System.out.println("Utslipp " + utslipp);
    }
    @Override
    public void skriv( ) {
        System.out.println("Lastebil med skatt og miljø: ");
        super.skriv( ); skrivSkatt(); skrivMiljo();
    }
}
```

} rollen
"Skattbar"

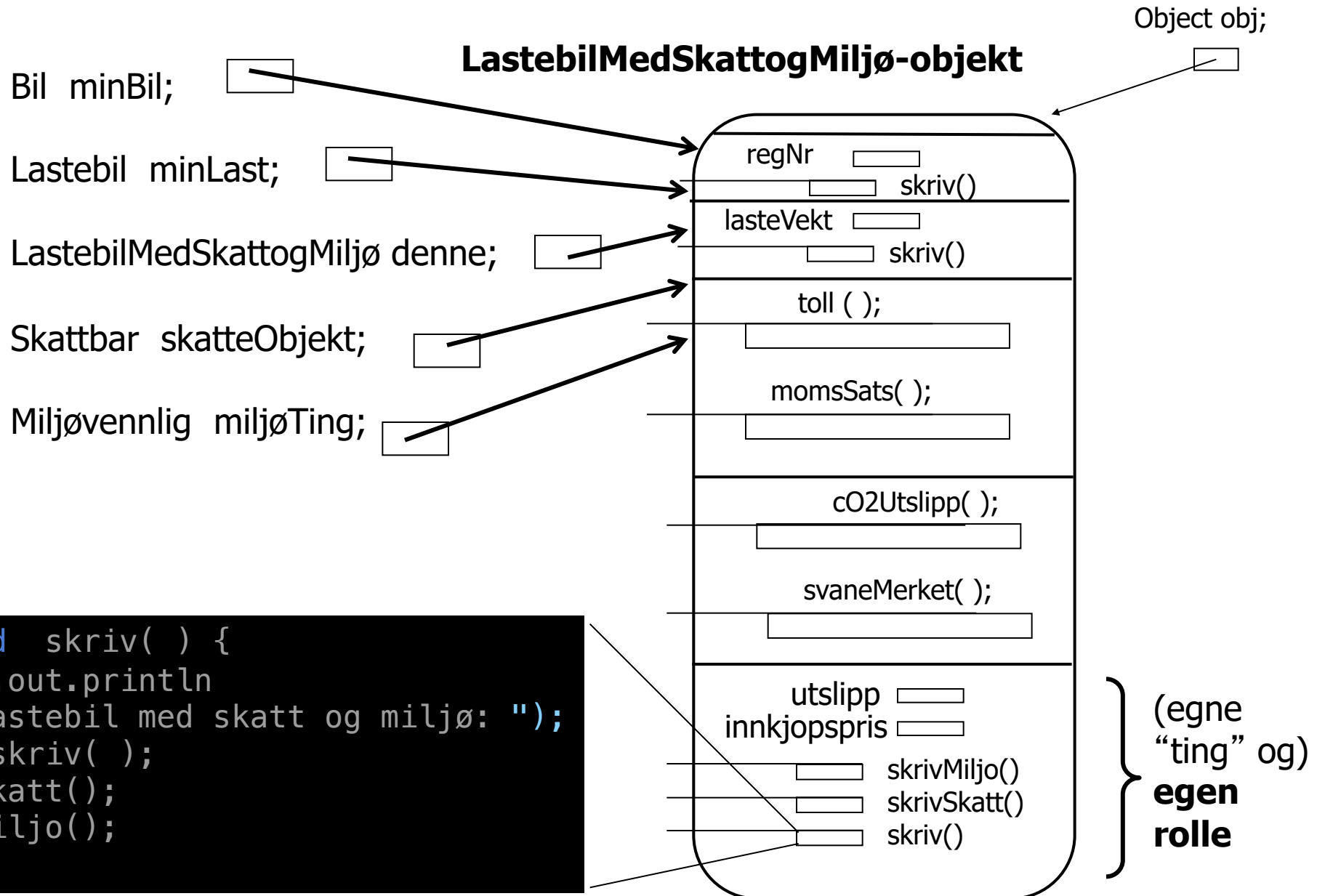
} rollen
"Miljøvennlig"

} Ny
"Skriv"-
rolle som
del av
LastebilMed. . .

(Skattbar og Miljøvennlig som før)

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

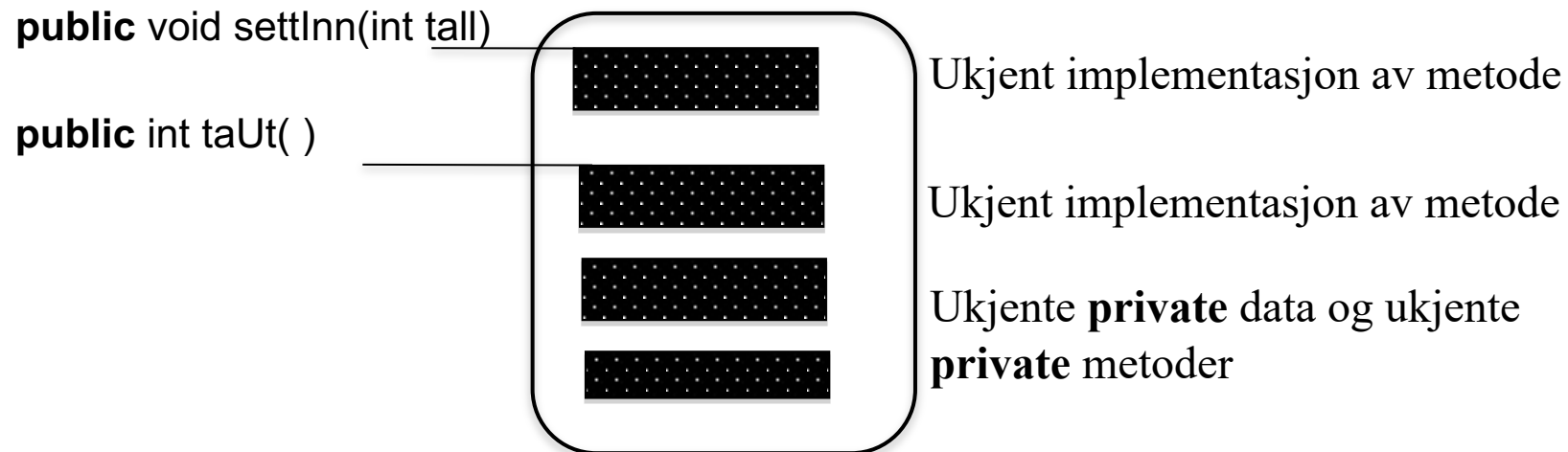
Skriv i LastebilMedSkattogMiljø

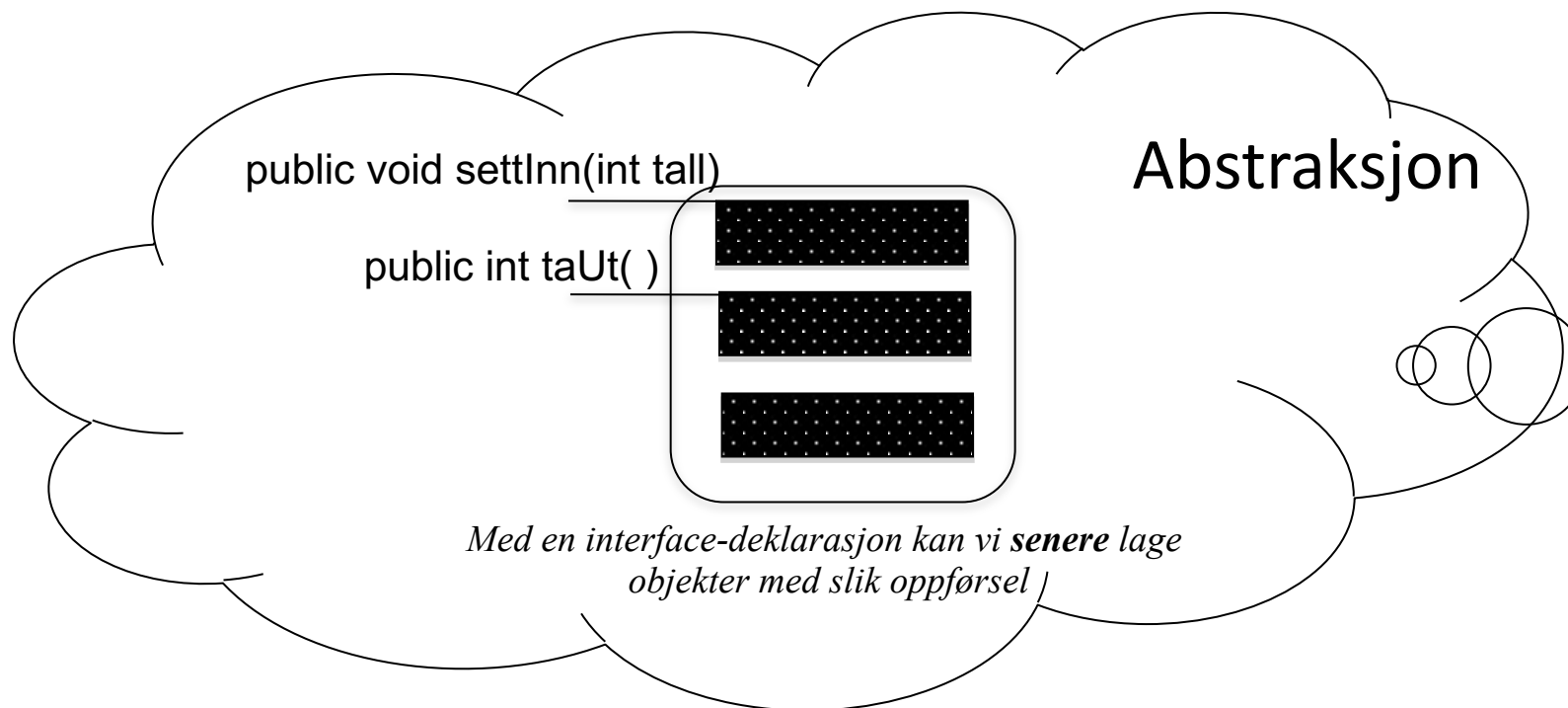


Interface som spesifikasjon av grensesnitt

Objektorientering handler om å tydeliggjøre objektene public-metoder – abstraksjon

Husker dere forelesingen om enhetstesting:

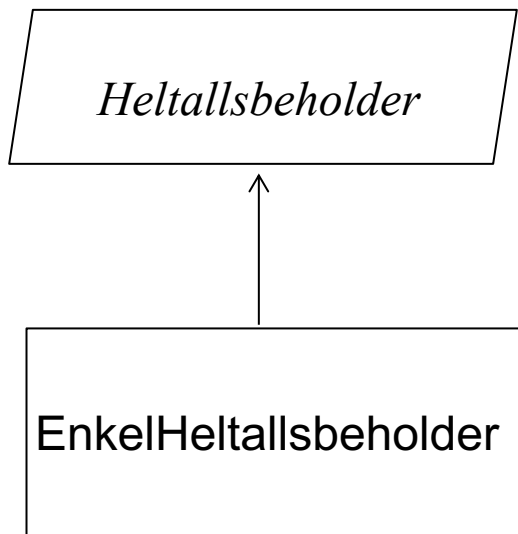




```
interface Heltallsbeholder {  
    void settInn(int tall);  
    int taUt( );  
}
```

~~new Heltallsbeholder()~~

Interface: Klassehierarki og Java-kode

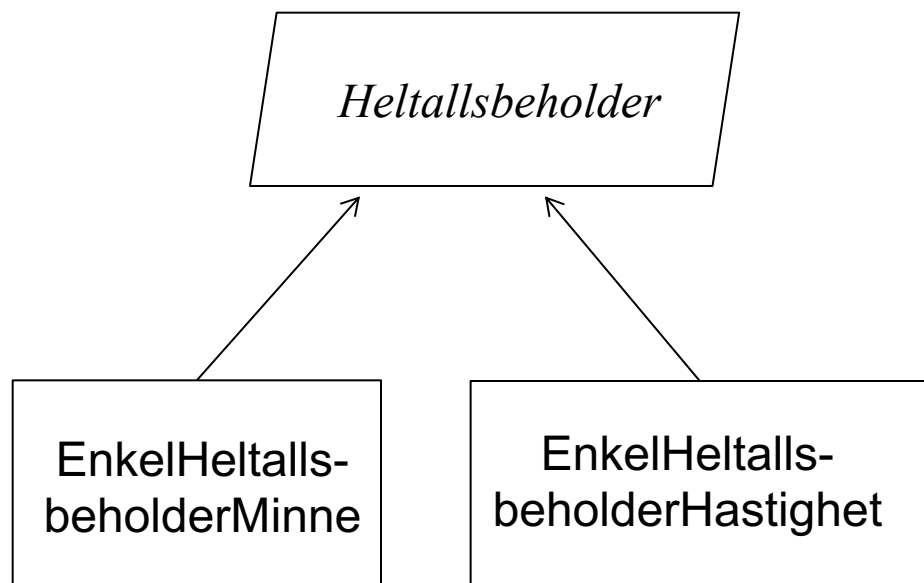


```
interface Heltallsbeholder {
    void settInn(int tall);
    int taUt( );
}
```

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

VIKTIG: Ett interface

Flere forskjellige implementasjoner



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

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

```
class EnkelHeltallsbeholderHastighet
    implements Heltallsbeholder {
    protected ArrayList . . . . .
    public void settInn(int tall) { . . }
    public int taUt( ) { . . . }
}
```

”Ikke-funksjonelle” forskjeller
på implementasjonene.
For eksempel hastighet,
minnebruk, . . .



Kan da lett bytte ut implementasjonen:

```
class BeregnEttEllerAnnet {  
    . . .  
    Heltallsbeholder hBeholder = new EnkelHeltallsbeholderHastighet( );  
    . . .  
    . . .  
    . . .  
    hBeholder.settInn(7);  
    int x = hBeholder.taUt();  
    . . .  
    hBeholder.settInn(13);  
    hBeholder.settInn(7102);  
    hBeholder.settInn(14);  
    . . .  
    int y = hBeholder.taUt();  
    . . .  
}
```

Kan byttes ut med

```
new EnkelHeltallsbeholderMinne( );
```

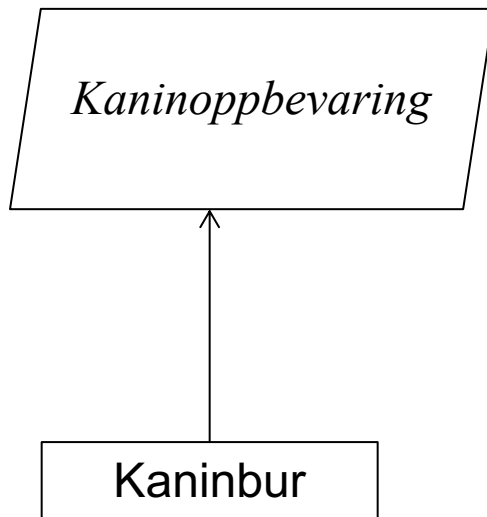
Resten av programmet er uforandret

Interface for å tydeliggjøre "public-metodene":

Vi kan se på et kaninbur som et sted for kaninoppbevaring



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



Kaninbur som et sted for kaninoppbevaring

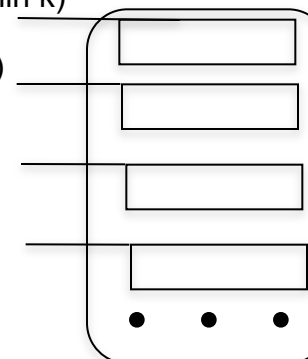
```
interface KaninOppbevaring {  
    public boolean settInn(Kanin k);  
    public Kanin taUt( );  
}
```

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



public boolean settInn(Kanin k)

public Kanin taUt()



Et objekt av en klasse som
implementerer grensesnittet
KaninOppbevaring

Full kode

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

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

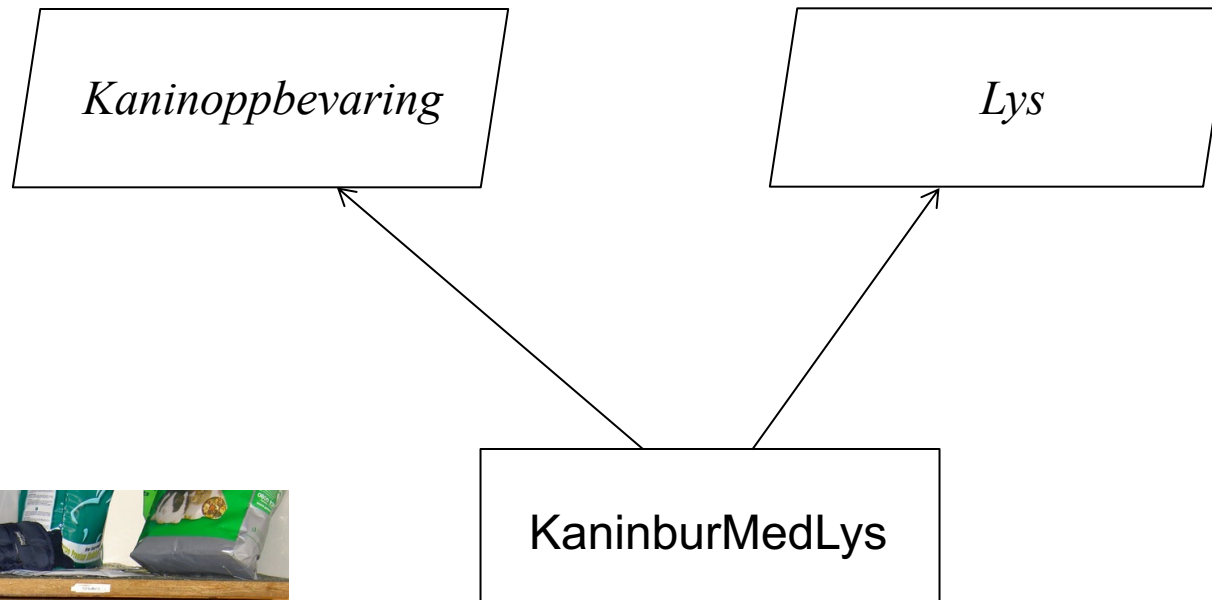


```
class Kaninbur implements KaninOppbevaring {  
    private Kanin denne = null;  
    @Override  
    public boolean settInn(Kanin k) {  
        if (denne == null) {  
            denne = k;  
            return true;  
        }  
        else return false;  
    }  
    @Override  
    public Kanin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

KaninburMedLys



Når en kanin vil ha lys på om natten:



Lys: Et sted/rom hvor det er lys som kan slås på og av

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



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

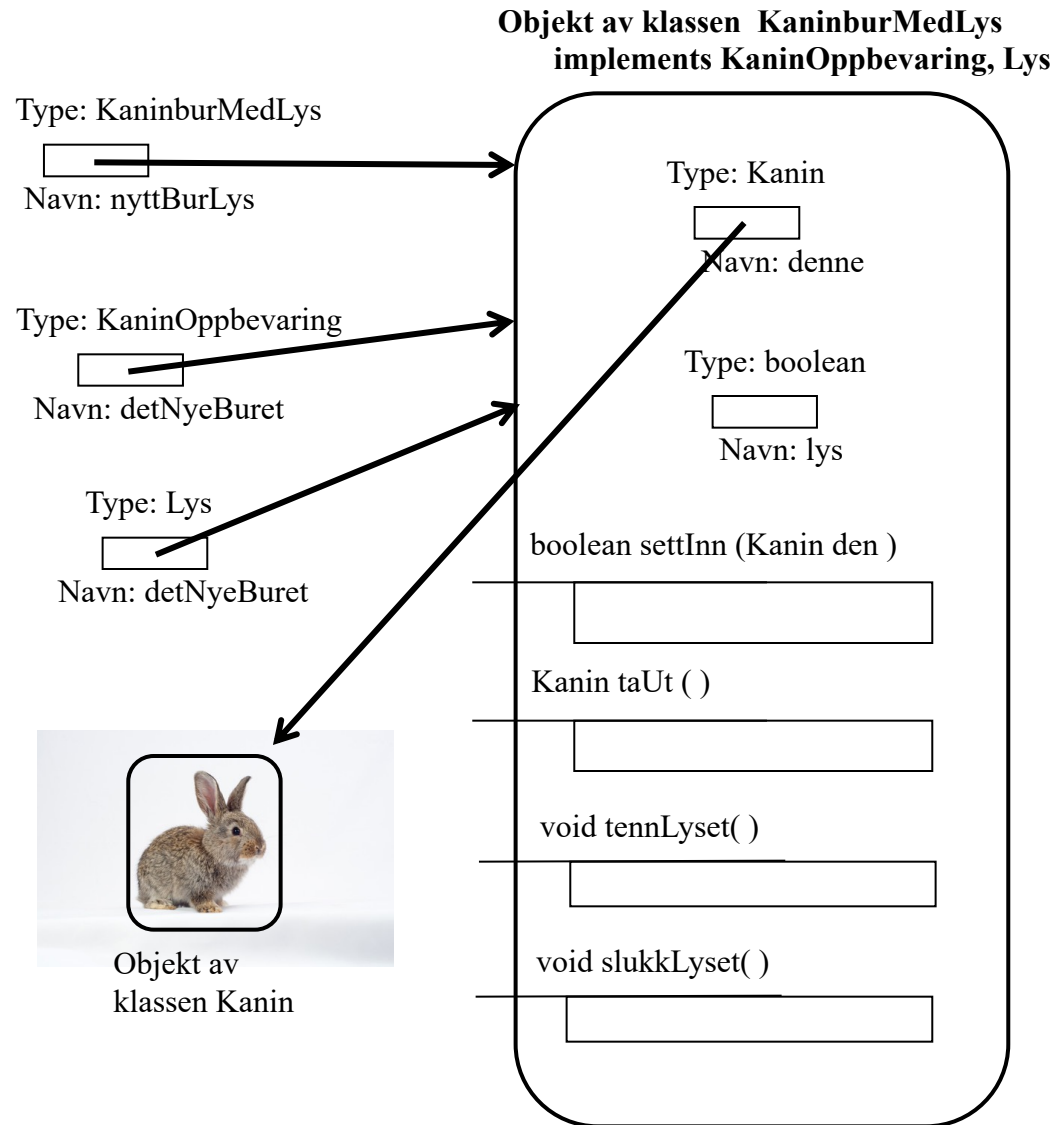
```
class KaninburMedLys implements
    KaninOppbevaring, Lys {
    private boolean lys = false;
    private Kanin denne = null;
    @Override
    public boolean settInn(Kanin k) {
        . . .
    }
    @Override
    public Kanin taUt( ) {
        . . .
    }
    @Override
    public void tennLyset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

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

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



Étt 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;
    @Override
    public boolean settInn(Kanin k) {
        if (denne == null) {
            denne = k;
            return true;
        }
        else {return false;}
    }
    @Override
    public Kanin taUt( ) {
        Kanin k = denne;
        denne = null;
        return k;
    }
    @Override
    public void tennLyset() {lys = true;}
    @Override
    public void slukkLyset(){lys = false;}
}
```

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

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

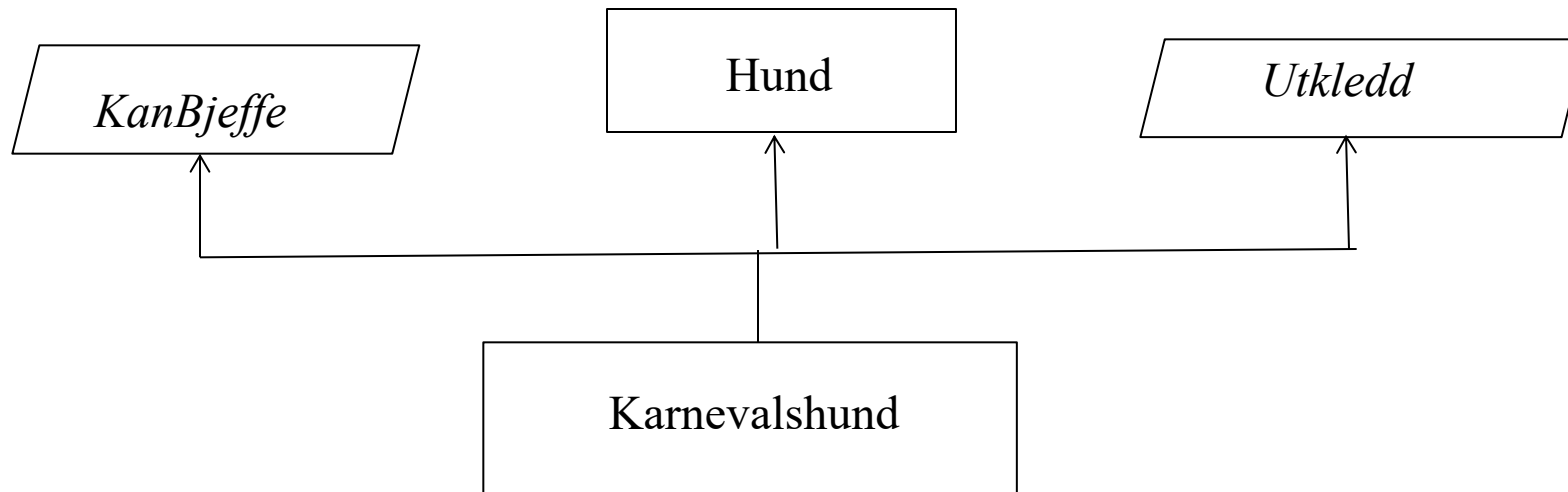


Karnevalshund

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



Foto: AP



Flere eksempler: En klasse – mange grensesnitt

```
class Hund {protected double vekt;}
```

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

```
interface Utkledd {  
    int antallFarger();  
}
```

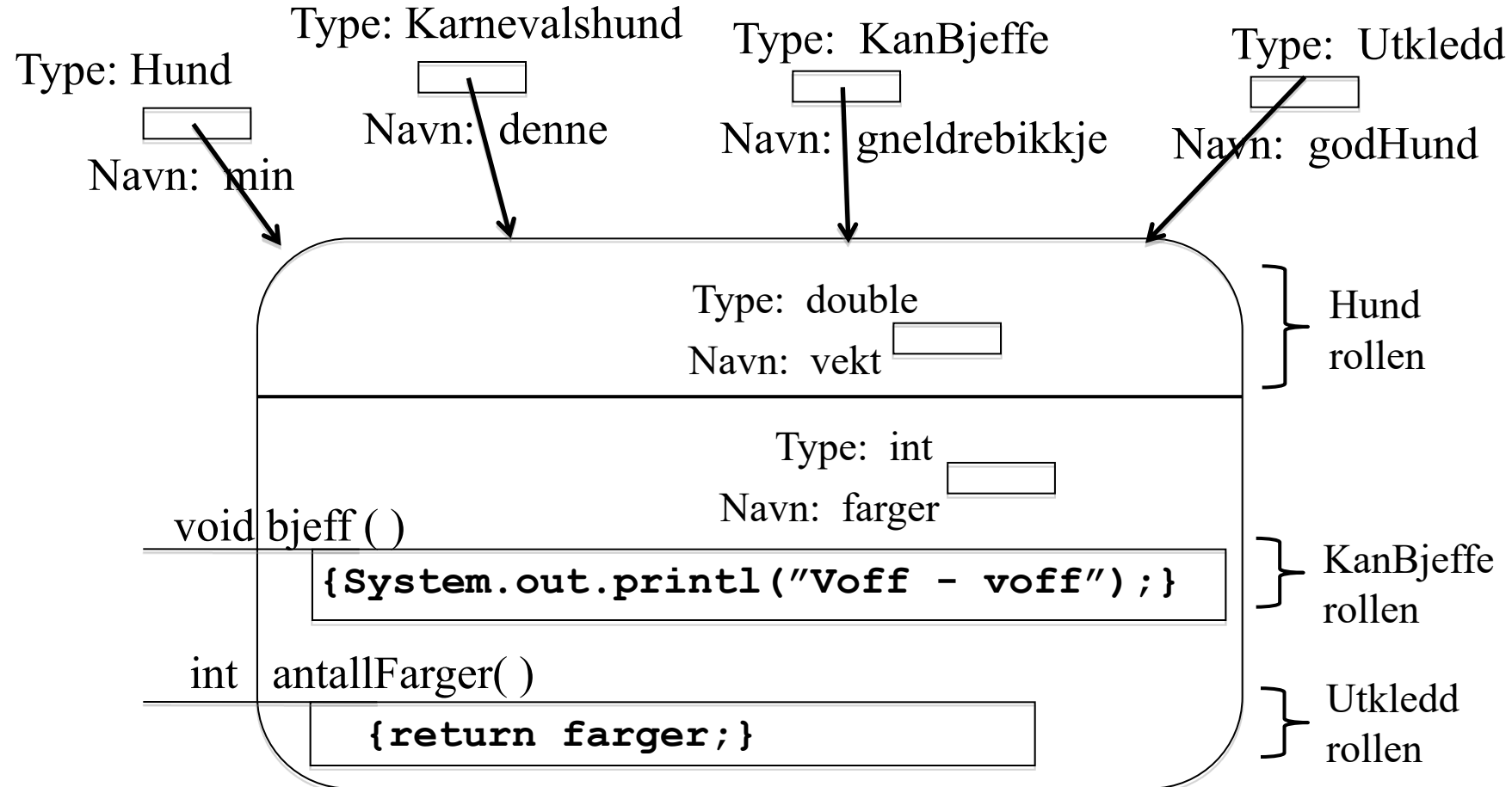
```
class Karnevalshund extends Hund implements KanBjefte, Utkledd {  
    protected int farger;  
    public Karnevalshund (int frg) { farger = frg; }  
    @Override  
    public void bjeff( ) {  
        System.out.println("Voff - voff");  
    }  
    @Override  
    public int antallFarger() { 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

Foto: AP



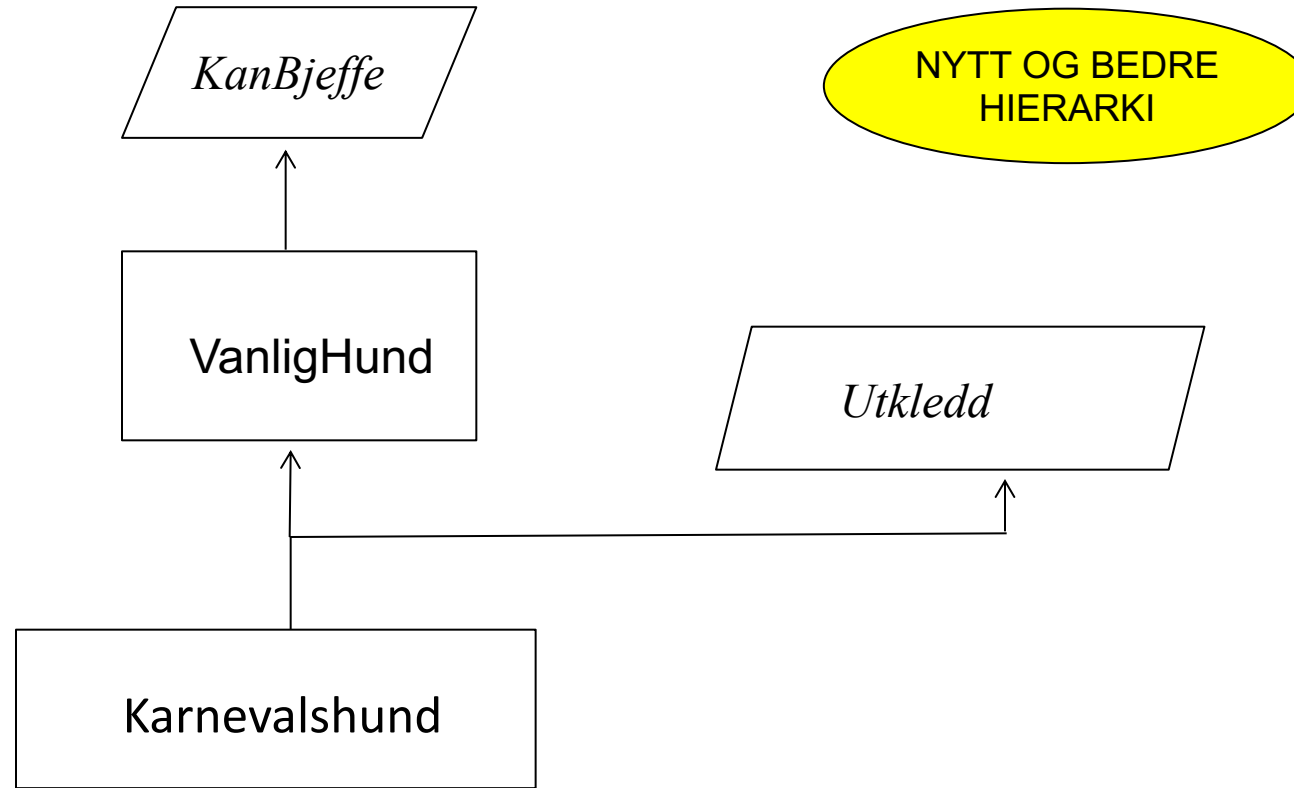
```

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



Objekt av klassen Karnevalshund

Eller kanskje bedre:



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

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

```
interface Utkledd {  
    int antallFarger();  
}
```

```
class VanligHund implements KanBjefte {  
    @Override  
    public void bjeff() {  
        System.out.println("Vov-vov");  
    }  
}
```

```
class Karnevalshund extends VanligHund implements Utkledd {  
    protected int farger;  
    public Karnevalshund (int frg) {  
        farger = frg;  
    }  
    @Override  
    public int antallFarger() {  
        return farger;  
    }  
}
```



Foto: AP

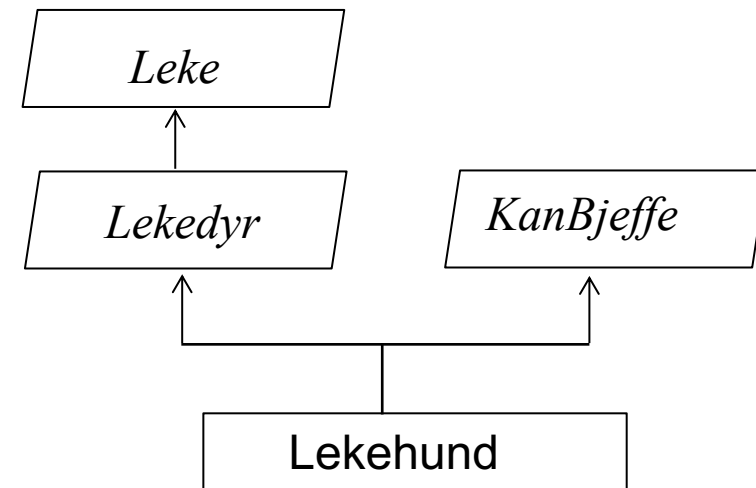
```
interface Leke {  
    String type();  
}
```

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

NYTT

```
interface Lekedyr extends Leke {  
    int hoyde();  
    boolean mykPels();  
}
```

```
class Lekehund implements Lekedyr, KanBjeffe {  
    int hoyde; boolean myk;  
    Lekehund(int h, boolean myk) {  
        hoyde = h; this.myk = myk;  
    }  
    @Override  
    public String type() { return "Hund";}  
    @Override  
    public int hoyde () {return hoyde;}  
    @Override  
    public boolean mykPels () {return myk;}  
    @Override  
    public void bjeff() {System.out.println("Vov-vov");}  
}
```



BATTERIES INCLUDED

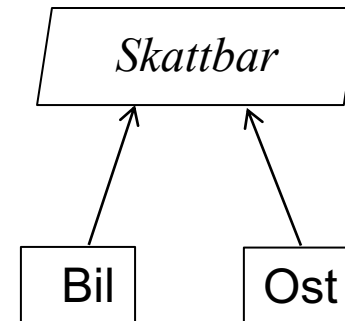
Tilbake til skatteetaten: Både biler og ost skal skattlegges

Ett grensesnitt, flere klasser.

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

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

class Ost implements Skattbar { // Ost: 200% skatt
    protected int importprisPrKg;
    protected . . .
    public Ost ( . . . ) { . . . }
    @Override
    public int skatt( ){return . . . * 2;}
}
```



Dette eksemplet har store likheter med Skatteetaten-eksemplet forrige uke

Hva er likt?

Hva er forskjellig?

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 totalSkatt = 0;  
totalSkatt = totalSkatt + minBS.skatt();  
totalSkatt = totalSkatt + minOS.skatt();
```



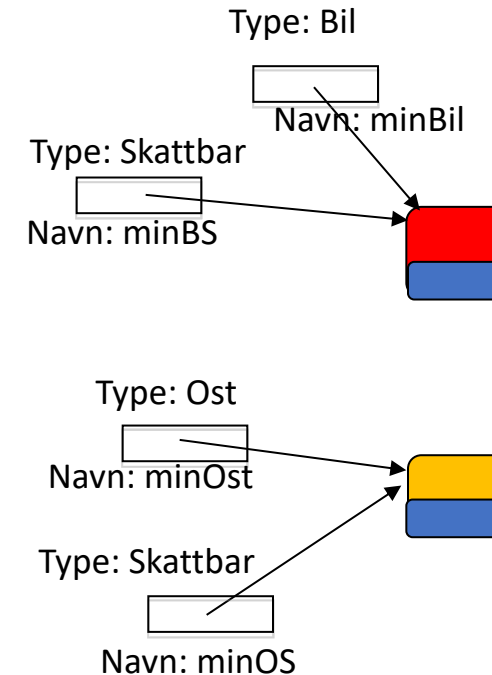
Rollen Skatt



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



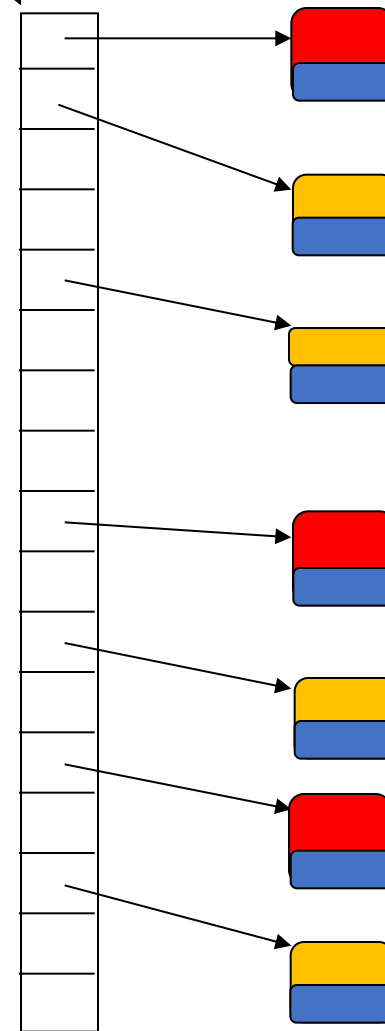
Rollen Bil (untatt Skatt)



Rollen Ost (untatt Skatt)

Type: Skattbar []

Navn: alle



Klassene Bil og Ost – Full kode

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

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

class Ost implements Skattbar { // Ost: 200% skatt
    protected int importprisPrKg;
    protected int antKg;
    public Ost (int kgPris, int mengde) {
        importprisPrKg = antKg; antKg = mengde;
    }
    @Override
    public int skatt( ){return importprisPrKg*antKg*2;}
}
```



Litt om invarianter og om å lage gode programmer

Mer den 10. mai.

Først:

Norsk: Antagelser / Betingelser

Engelsk: Assertions / Conditions

- En beskrivelse av tilstanden på et bestemt sted i programmet kalles en antagelse eller en betingelse, f.eks.:

```
int tall = 0;
while (tall < 10) {
    tall++;
}
// Nå er !(tall < 10)
// dvs.    tall >= 10
```

tall >= 10 kalles en antagelse eller en betingelse

Et godt program inneholder kommentarer med gyldige antagelser om tilstanden til variablene i programmet på dette stedet i koden

WIKIPEDIA: Invariant (computer science)

- In [computer science](#), an **invariant** is a condition that can be relied upon to be true during execution of a program, or during some portion of it. It is a [logical assertion](#) that is held to always be true during a certain phase of execution. For example, a [loop invariant](#) is a condition that is true at the beginning and end of every execution of a loop.
.....
- Programmers often use [assertions](#) in their code to make invariants explicit. Some [object oriented programming languages](#) have a special syntax for specifying [class invariants](#).

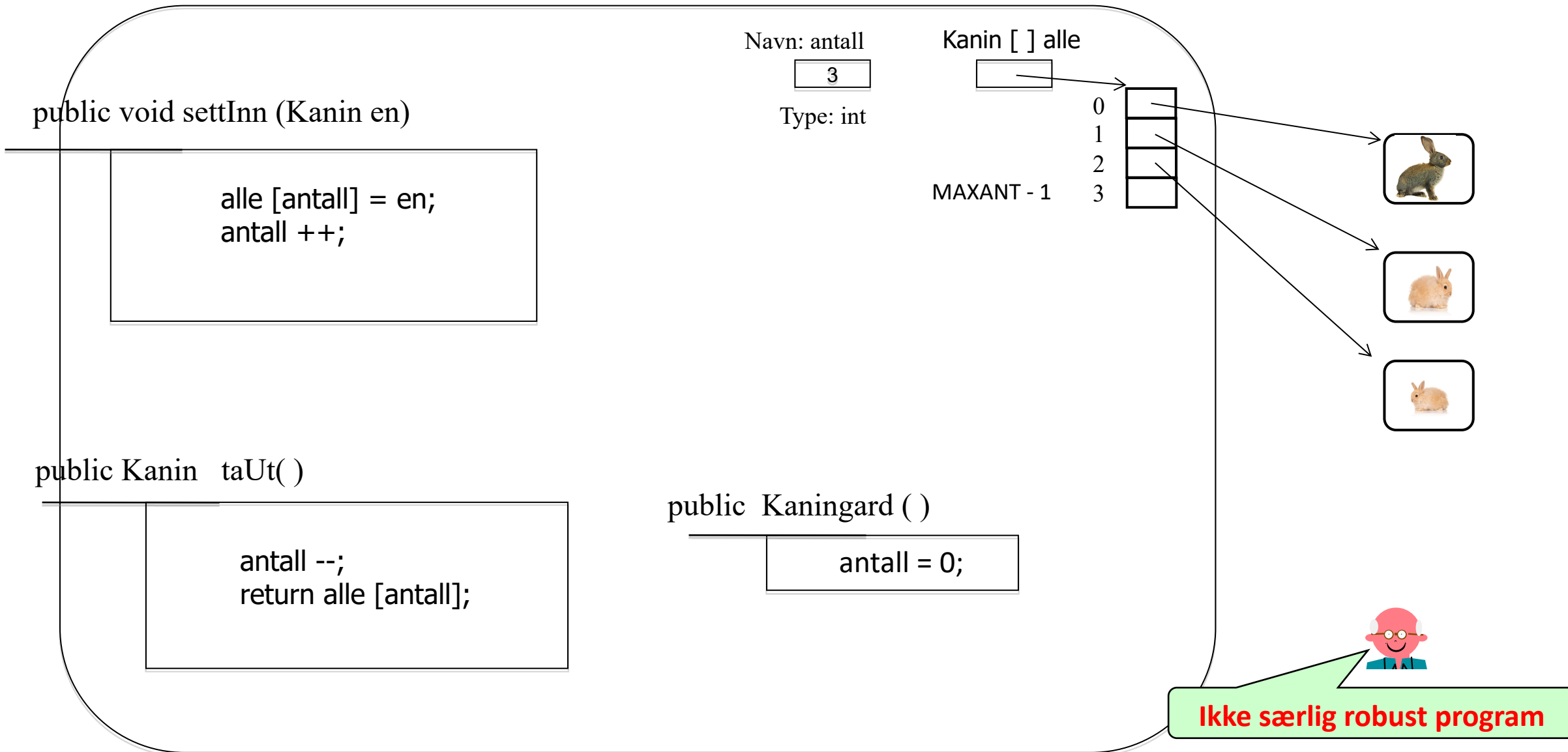
“Loop invariants” skal vi ikke snakke om nå men den 10. mai.



Invarianter i objekter

- Ofte kalt (noe unøyaktig) “class invariants”
- En invariant på datastrukturen i et objekt hjelper oss å programmere (public) metodene bedre og mer oversiktlig
- En invariant på datastrukturen i et objekt sier oss noe om instansvariablenes verdier, forholdet mellom verdiene til instansvariablene og generelt om datastrukturen inne i objektet
- En invariant på datastrukturen i et objekt forklarer oppgaven til instansvariablene og begrenser verdiene de kan ha
 - Litt slik som typen til en variabel gjør

Kaningård





La oss se på kode:

- BrukKanin.java

Invarianten holder i det objektet opprettes OG alle metodene bevarer invarianten -> Da holder invarianten alltid

public void settInn (Kanin en)

Pre-condition: Invarianten gjelder

```
if (antall < MAXANT) {
    alle [antall] = en;
    antall ++;
}
```

Post-condition: Invarianten gjelder

public Kanin taUt()

Pre-condition: Invarianten gjelder

```
if (antall > 0) {
    antall --;
    return alle [antall];
} else return null;
```

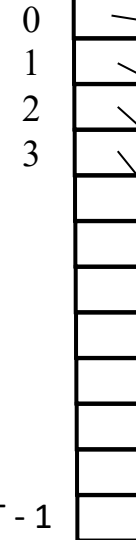
Post-condition: Invarianten gjelder

Navn: antall

3

Type: int

Kanin [] alle



Invariant:
 $0 \leq \text{antall} \leq \text{MAXANT}$ og
 alle kaninene er lagret fom.
 $\text{alle}[0]$ tom. $\text{alle}[\text{antall}-1]$

public Kaningard ()

Pre-condition: true

antall = 0;

Post-condition: Invarianten gjelder

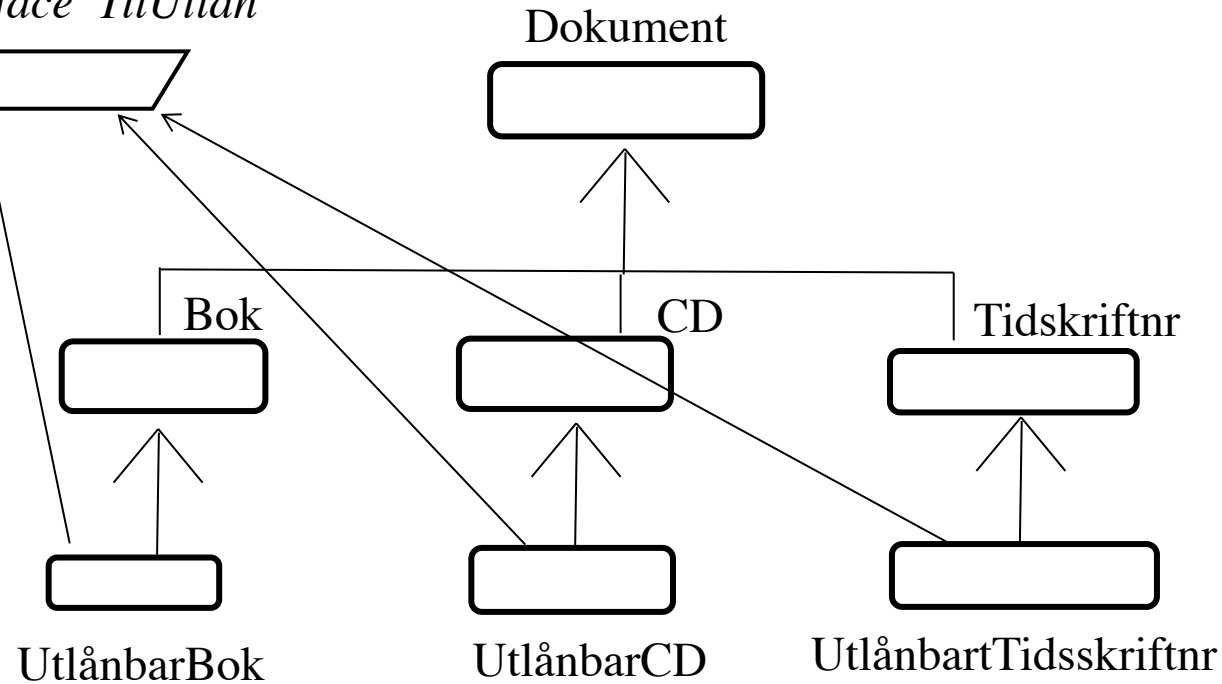
(Hva gjør denne kaninen her ?)

Hoved "take-away" i dag

- To hoved-grunner til å bruke interface:

Multipel arv og samme oppførsel på tvers av klasser

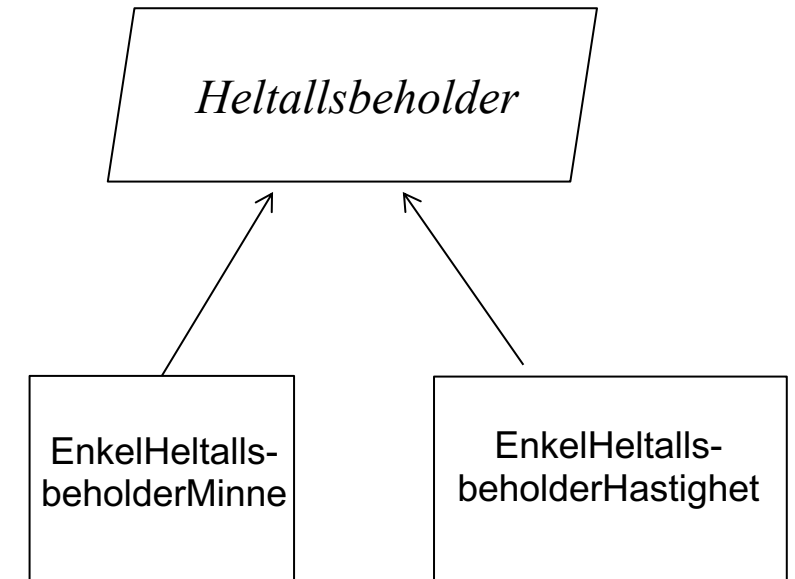
interface TilUtlån



+ Både Biler og Oster er *Skattbare*

Forskjellig / ukjent implementasjon men samme oppførsel

```
Heltallsbeholder hBeholder = new . . .
```





Oppsummering om interface

- Java har en mekaniske "interface" som
- Tydeliggjør og definerer en (implementerendes) klasses grensesnitt
 - Abstraksjon / Innkapsling / Skjuling av detaljer
- Kan brukes til multippel arv av oppførsel
 - Men alle metodene må implementeres på nytt
 - Så Java har ikke multippel arv av kode
- Er en "rolle" på linje med klasser og subklasser
- Kan brukes som en referansetype
- Alle egenskapene til et interface er **metoder** (metodesignaturer)
 - ; istedenfor { . . . } bak signaturen / overskriften til metoden



Eksempel på toString og equals

```
class Punkt1 {  
    protected int x, y;  
    Punkt1(int x0, int y0) {  
        x = x0;  
        y = y0;  
    }  
}
```

```
Punkt p1 = new Punkt1(3,4);  
Punkt p2 = new Punkt1(3,4);
```

```
p1.toString(); Punkt1@4d591d15
```

```
p1.equals(p2); false
```

```
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;  
    }  
}
```

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

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

```
q1.equals(q2); true
```