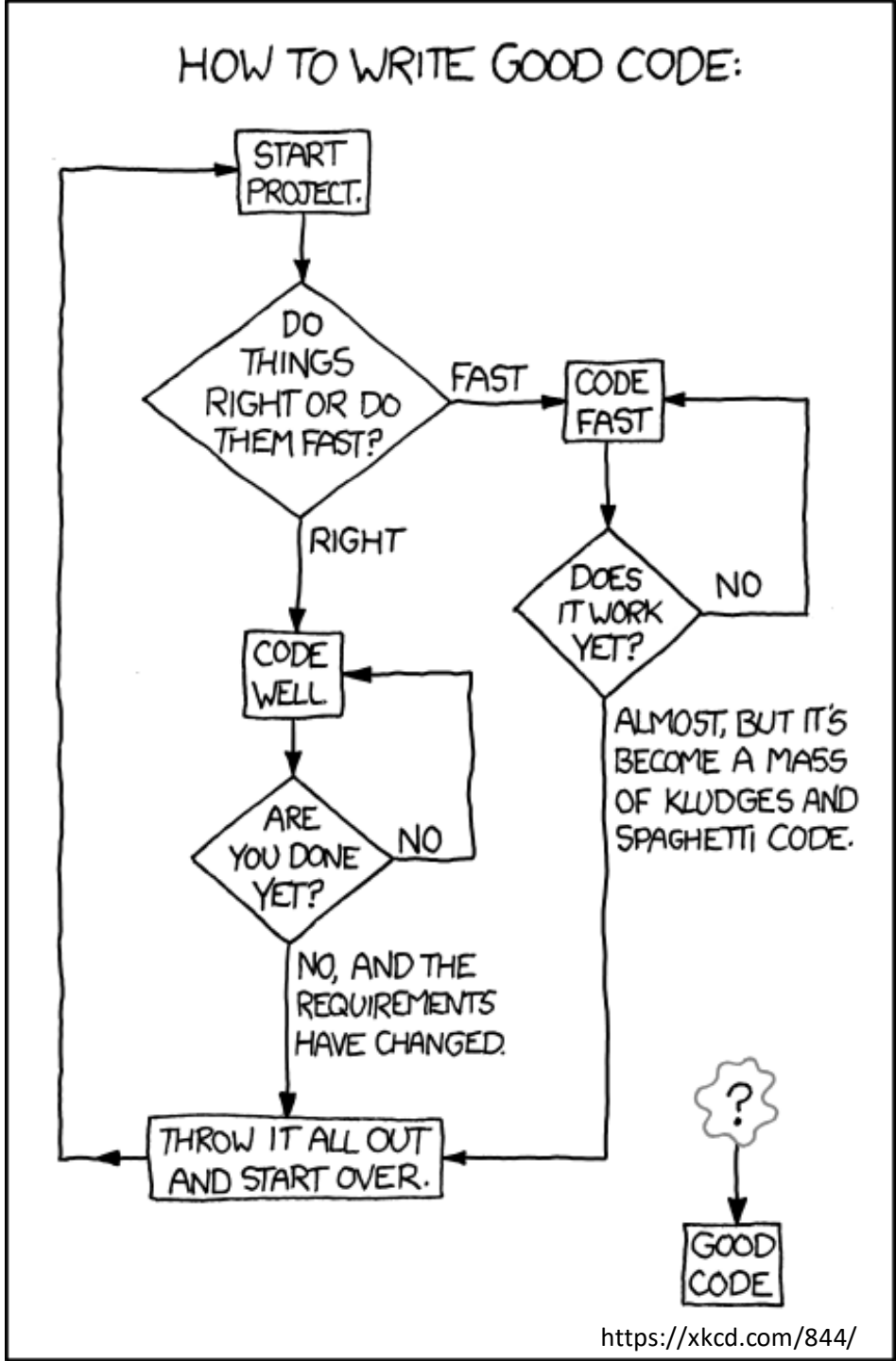
UNIVERSITETET I OSLO

IN1010 våren 2026

Arv og subklasser del 3: Grensesnitt i Java

Eyvind W. Axelsen (eyvinda@ifi.uio.no)
Lysark basert på materiale tidligere laget av Stein Gjessing



https://xkcd.com/844/

Dagens hovedtema

- Engelsk: Interface
- Norsk: Grensesnitt
- Java-nøkkelord: `interface`

- To måter å bruke interface på
  1) «Multippel arv» av roller
  2) Definere en oppførsel/rolle/tjeneste/kontrakt

Multippel arv: Om å arve fra flere

- I Java kan en klasse bare arve egenskapene til **én** annen klasse (én direkte superklasse)
  - Dette gjør språket enklere å bruke og enklere å implementere
- 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

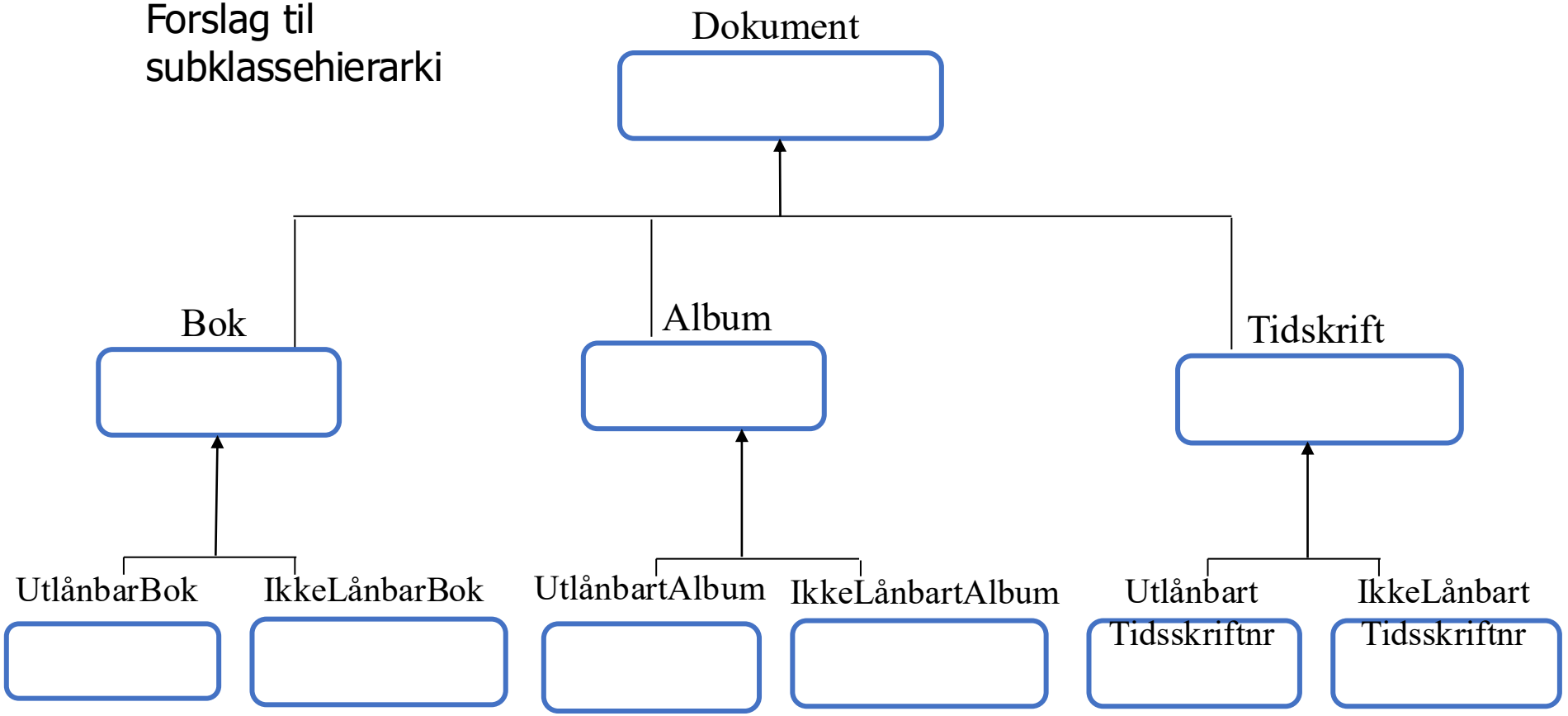
Motivasjon for mekanismen interface: Analyse av et bibliotek

- 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

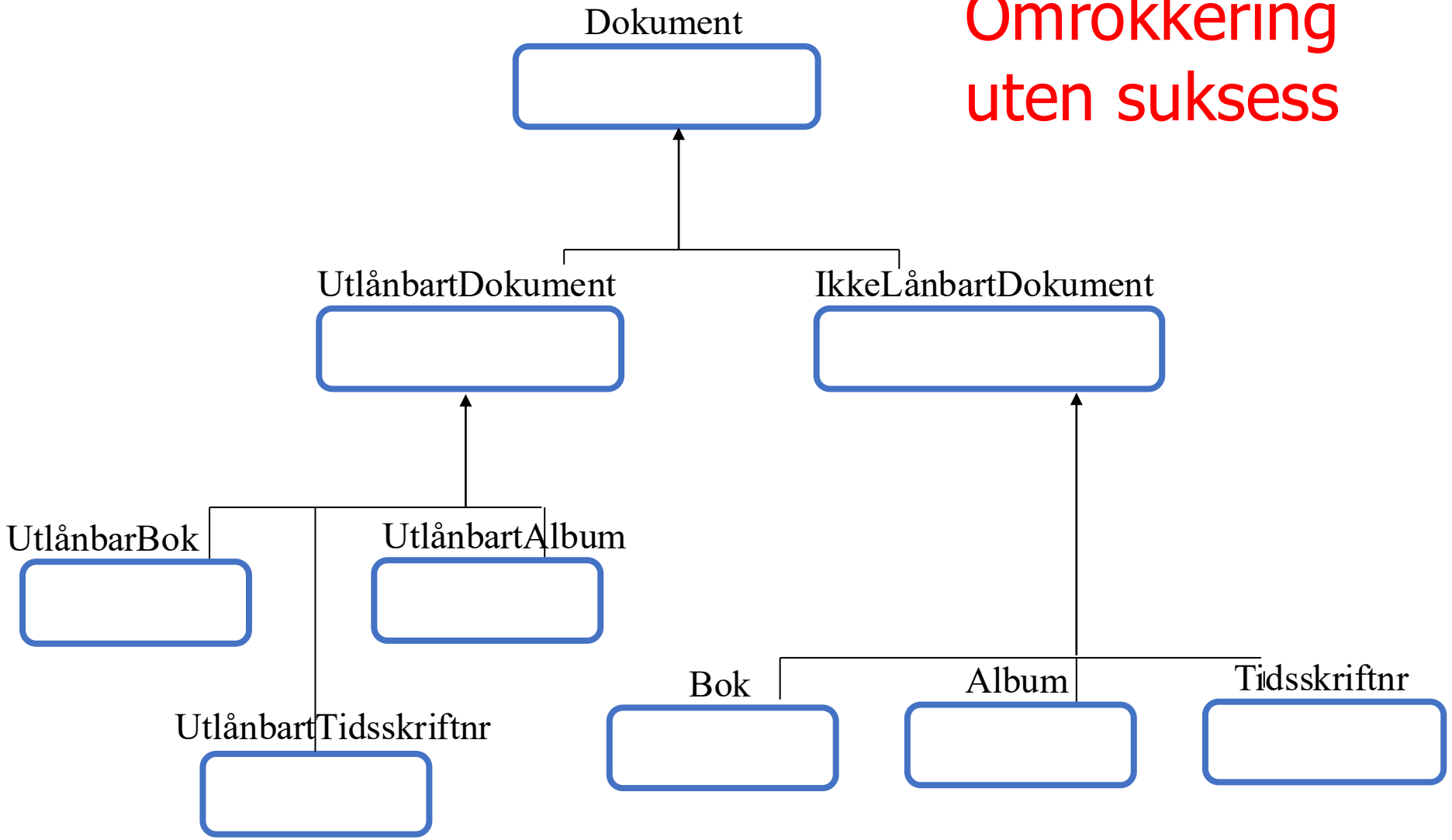


Tvilsomt begrepshierarki

Forslag til subklassehierarki

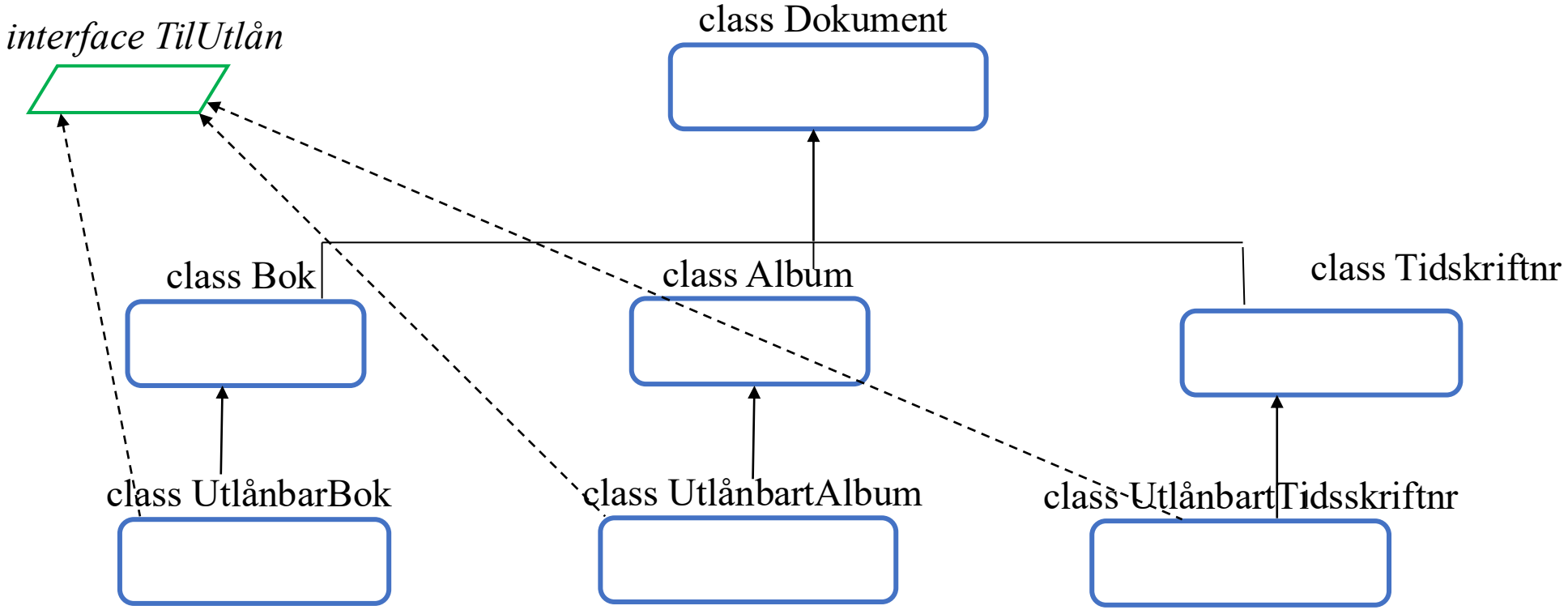


Omrokkering uten suksess



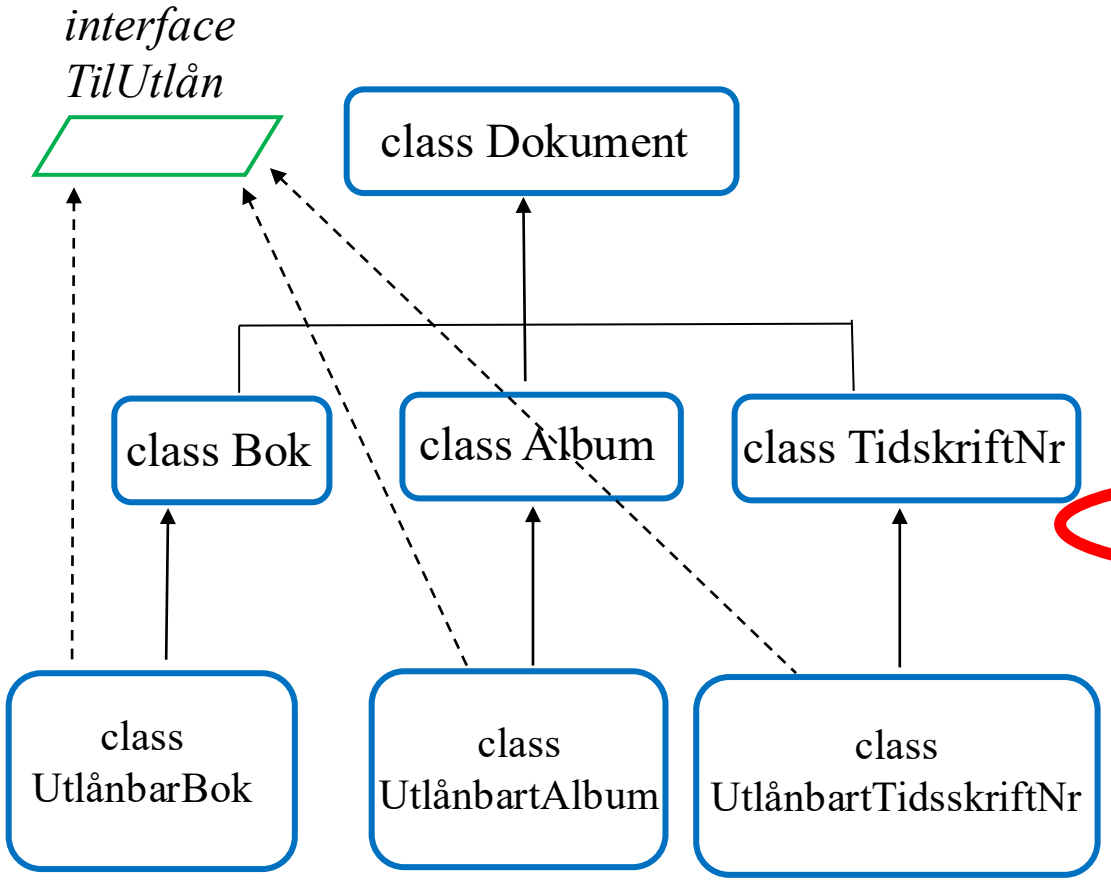
Samle lik oppførsel: bruk **interface**

NYTT



- 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 flere **roller**

Dokumentene i et bibliotek (en gang til)




```
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 TiskriftNr implements TilUtlån { }
```

Men hva kommer med fra et interface?


```
class Dokument { }

class Bok extends Dokument { }

class Album extends Dokument { }

class TidskriftNr extends Dokument { }

interface TilUtlån {...}

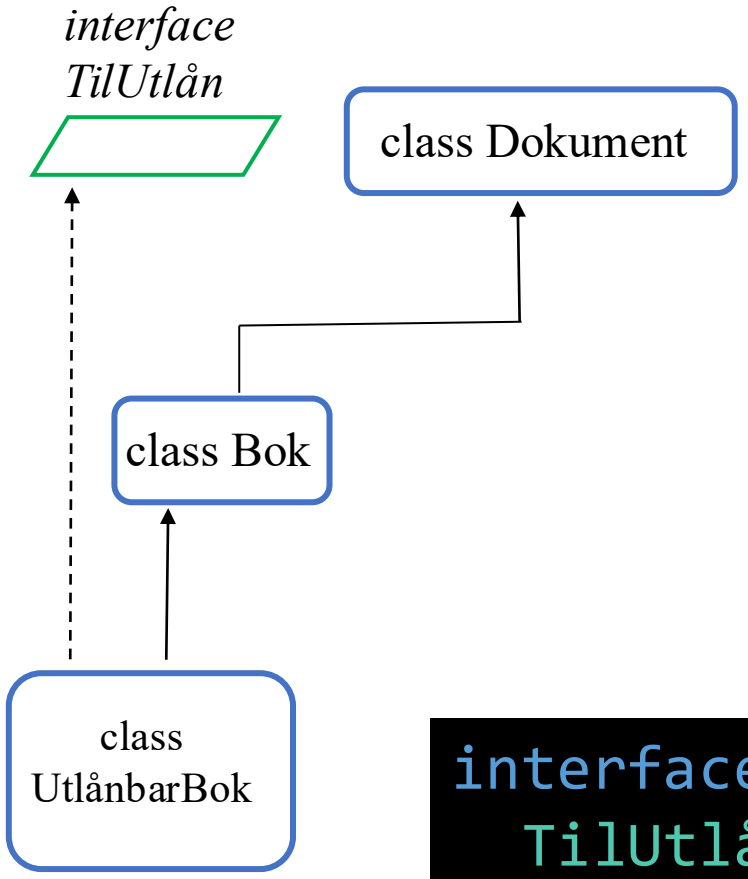
class UtlånbarBok extends Bok implements TilUtlån { }

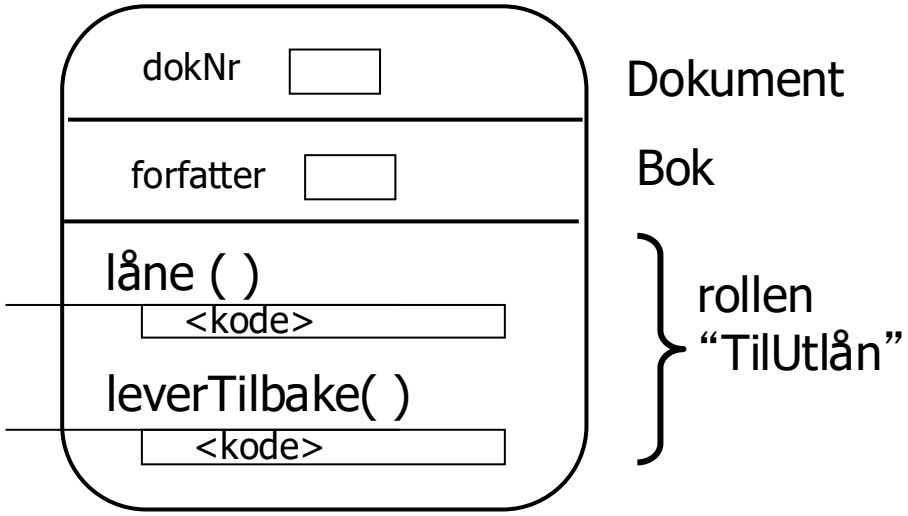
class UtlånbartAlbum extends Album implements TilUtlån { }

class UtlånbartTidskriftNr extends TidskriftNr implements TilUtlån { }
```

Men hva kommer med fra et interface?

Svar: En rolle (metodesignaturer)



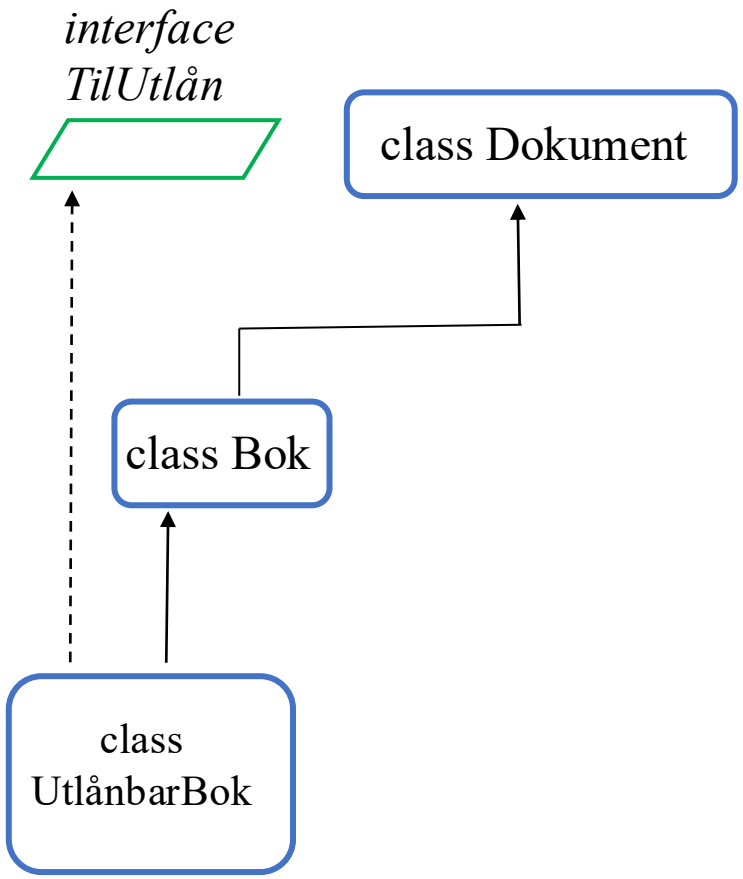


Et objekt av klassen UtlånbarBok

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

Men hva kommer med fra et interface?

Svar: En rolle (metodesignaturer)




```java
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 . . . ;

  @Override
  public TilUtlån låne(String lnr) { ... }

  @Override
  public void leverTilbake() { ... }
}
```

Klassen må *selv implementere* metodene

Men hva kommer med fra et interface?

Svar: En rolle (metodesignaturer)


```
class Dokument {final int dokNr;}

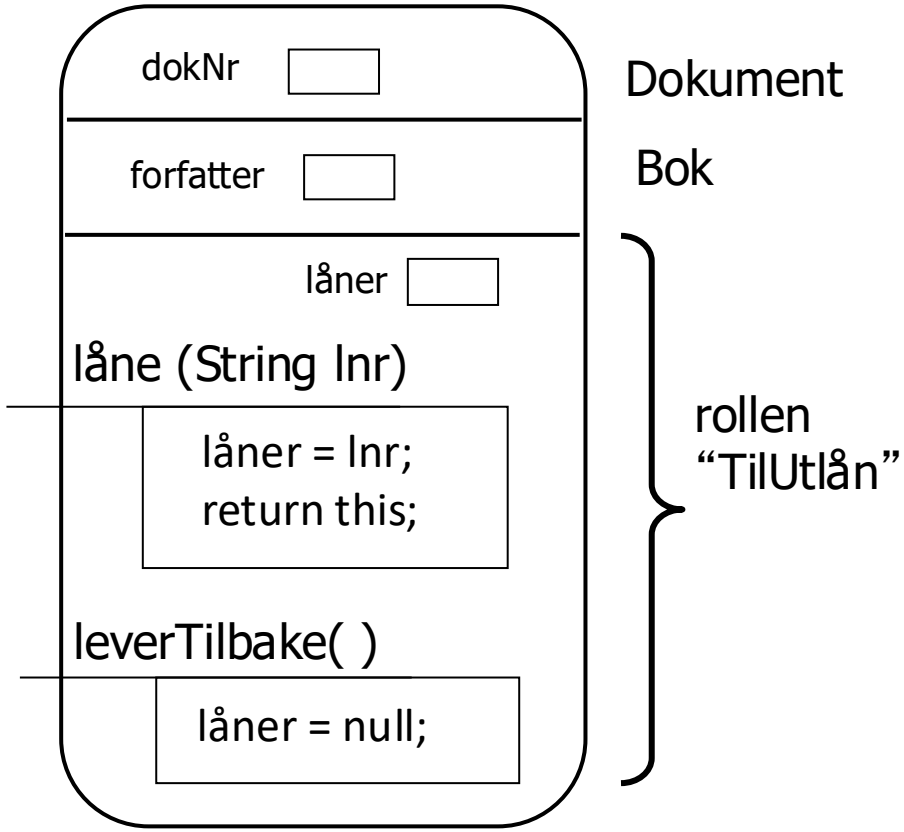
class Bok extends Dokument {final String forfatter;}

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

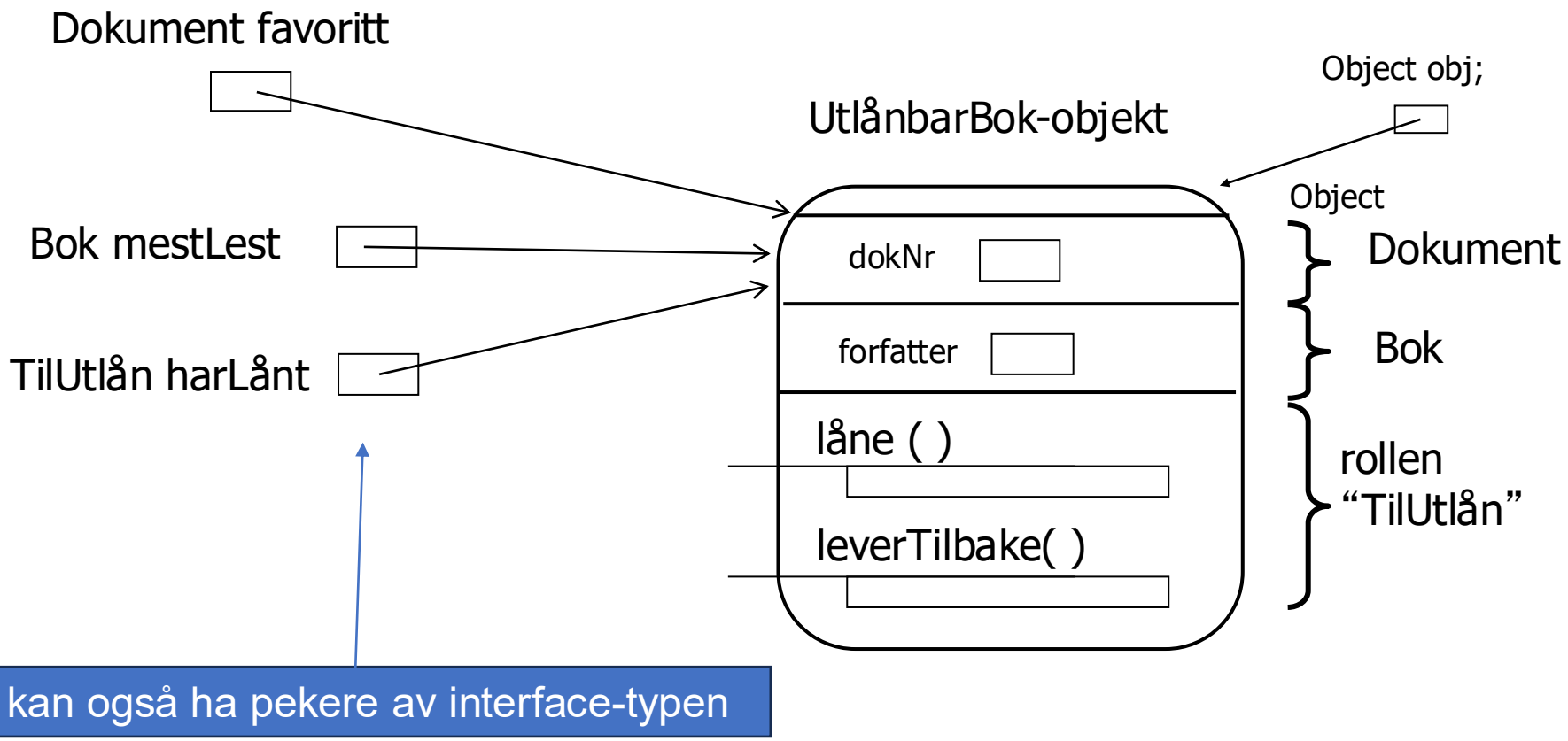

class UtlånbarBok extends Bok implements TilUtlån {
   private String låner = null;

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

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



Et objekt av klassen UtlånbarBok


Dokument favoritt
Object obj;
UtlånbarBok-objekt
Object
Bok mestLest
dokNr
Dokument
forfatter
Bok
TilUtlån harLånt
låne ( )
rollen
"TilUtlån"
leverTilbake( )
Vi kan også ha pekere av interface-typen

Et interface (grensesnitt) er:

- En samling egenskaper (en rolle)
- …som mange forskjellige "ting" av forskjellige typer kan anta

For eksempel

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

- Et interface ligner en abstrakt klasse
  - Metodene i et interface er abstrakte og polymorfe
  - De har ingen implementasjon
- Et interface inneholder **ingen** variable
- En klasse som implementerer et interface må selv putte inn kode i alle metodene
- En klasse kan implementere mange interface
- Å implementere (en samling metoder) = å spille en rolle
- Husk: ved å implementere et interface får du bare med metodesignaturer
  - (Det finnes noe som kalles “default”-metoder i interface, men vi fokuserer ikke på det i IN1010)



Nytt interface-eksempel

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

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

Rollen Skattbar

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

Rollen Miljøvennlig

Tegning av interfacer i klassehierarkiet


```
interface Skattbar {

}
```

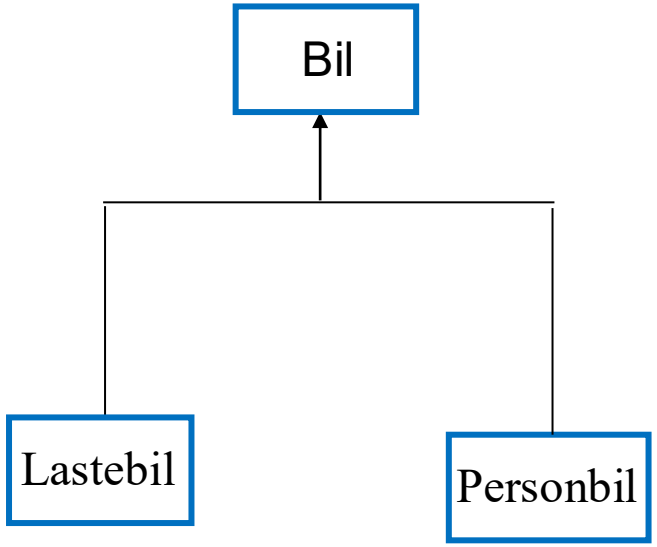



```
interface Miljøvennlig {

}
```



Slik var bil-hierarkiet

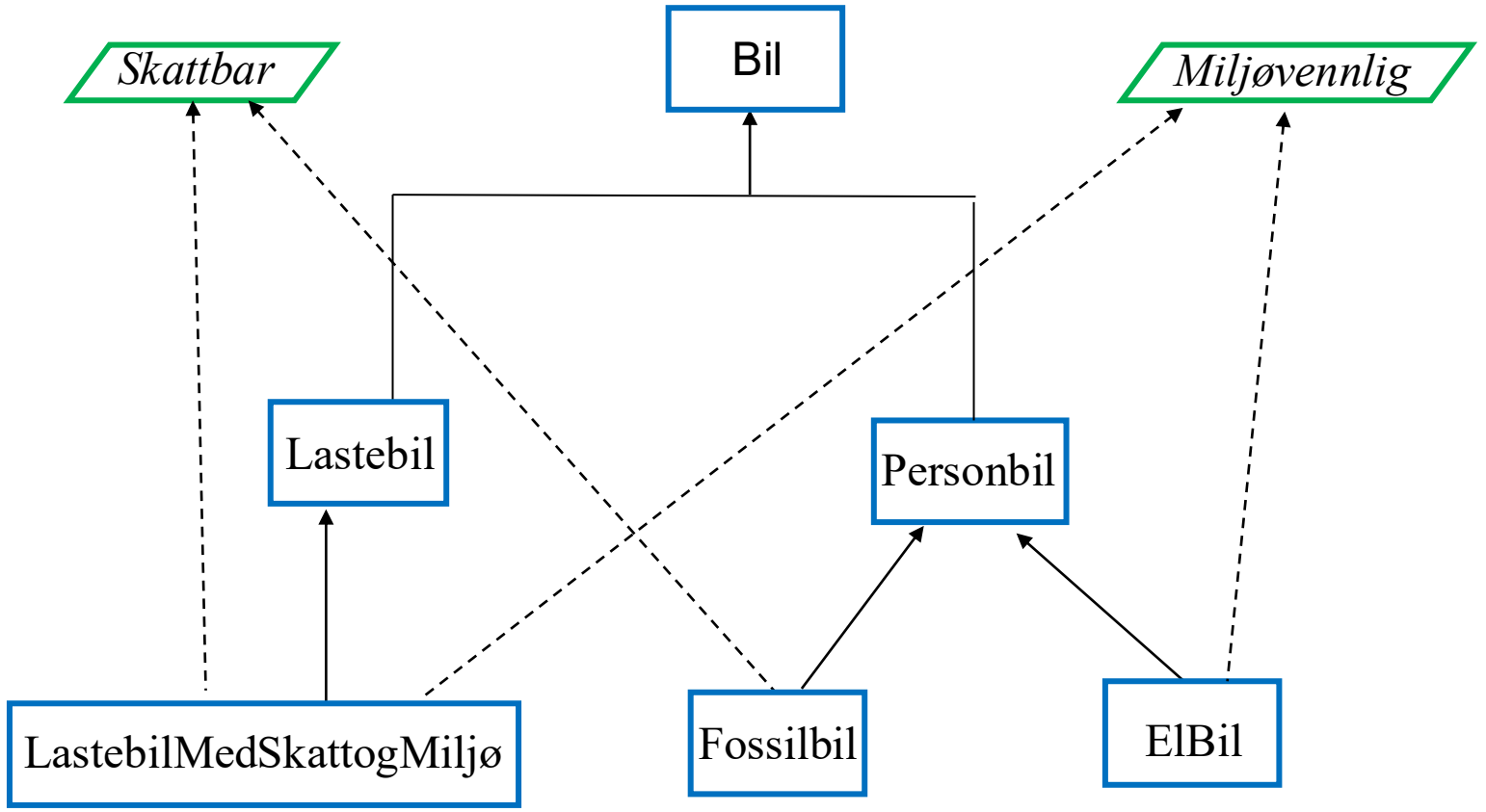


Mange biler må betale skatt og noen biler er miljøvennlige:

Skattbar

Miljøvennlig

Tre nye klasser som kan spille mange roller



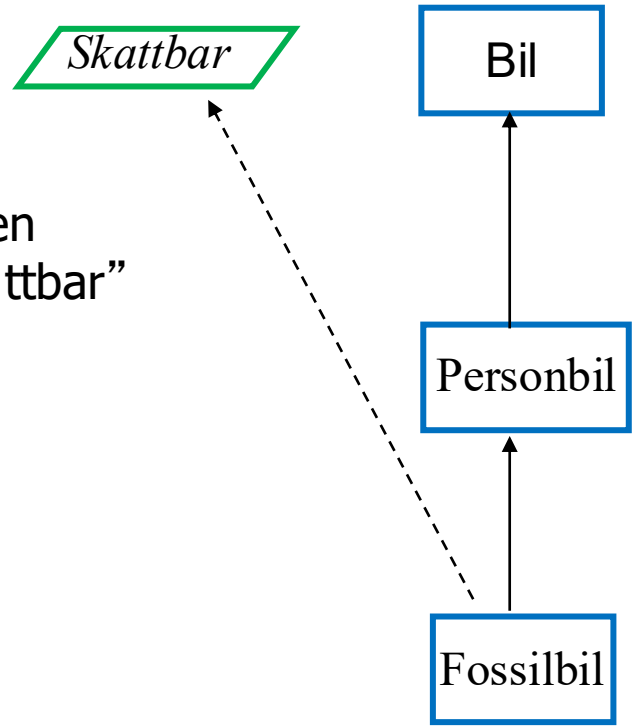
Metodene må skrives på nytt hver gang de implementeres. Her er for eksempel skatt i en lastebil forskjellig fra skatt i en fossilbil.

Mer om å implementere et 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"

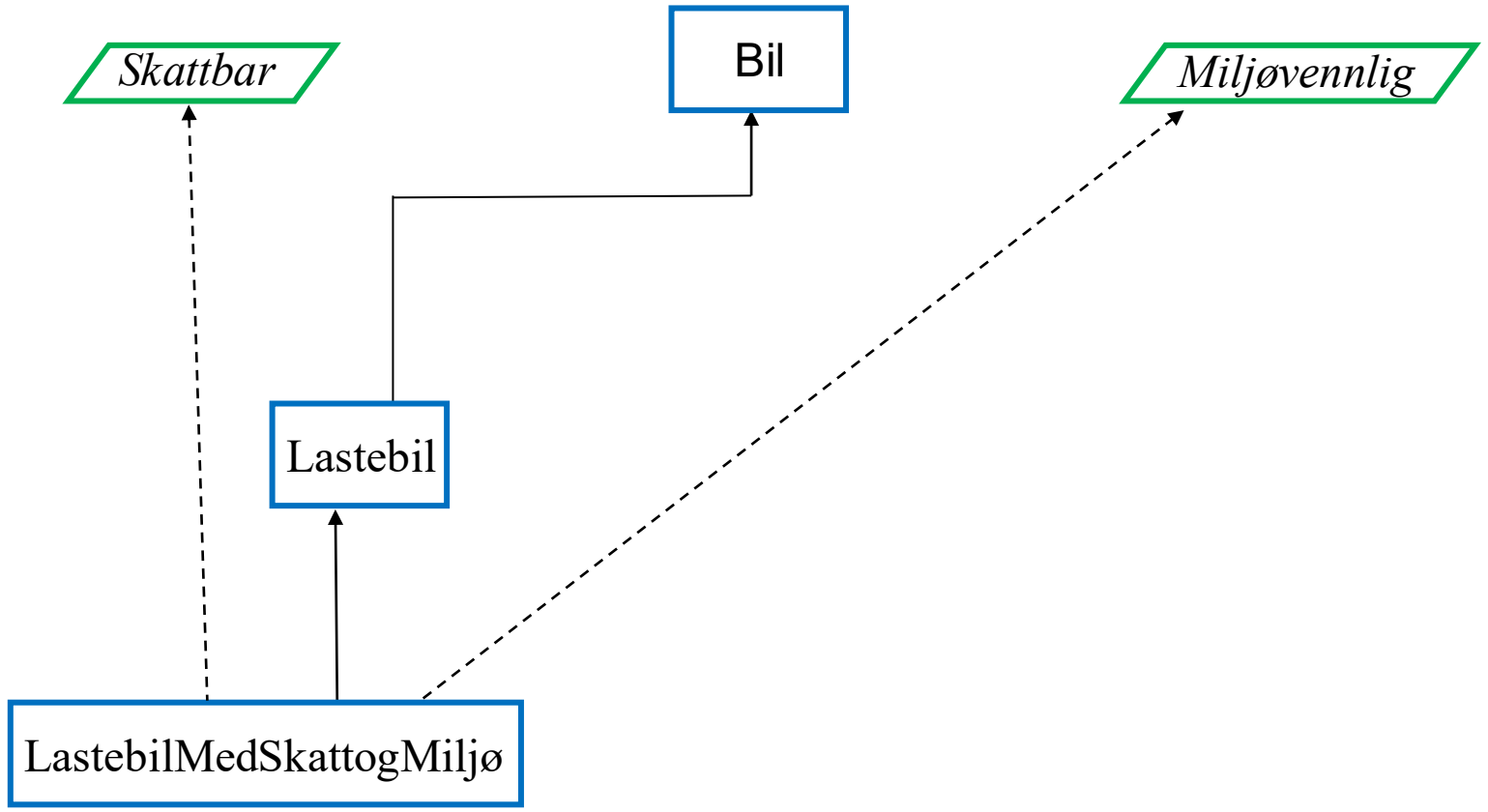



```
interface Skattbar {
    double toll();
    double moms();
}
```

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

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

Om å implementere to interfacer



UNIVERSITETET I OSLO

Om å implementere to interfacer

Institutt for informatikk




```
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 cO2Utslipp(){return utslipp;}
   @Override
   public boolean svaneMerket(){return false;}
}
```

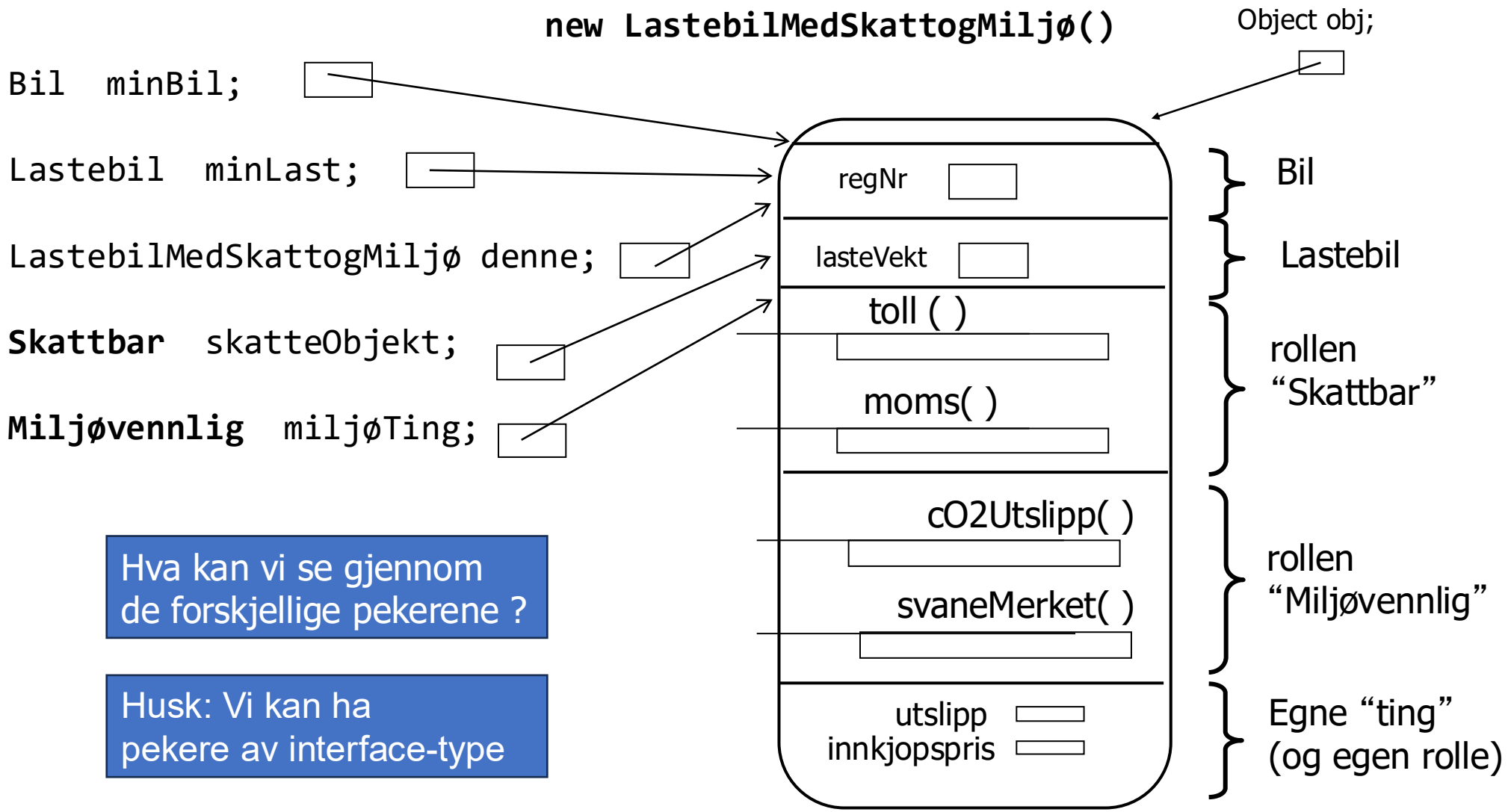
```
interface Skattbar {
    double toll();
    double moms();
}
```

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

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

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

Et objekt og noen pekere



Hva kan vi se gjennom de forskjellige pekerene ?

Husk: Vi kan ha pekere av interface-type

Interface kan brukes som typer i programmet

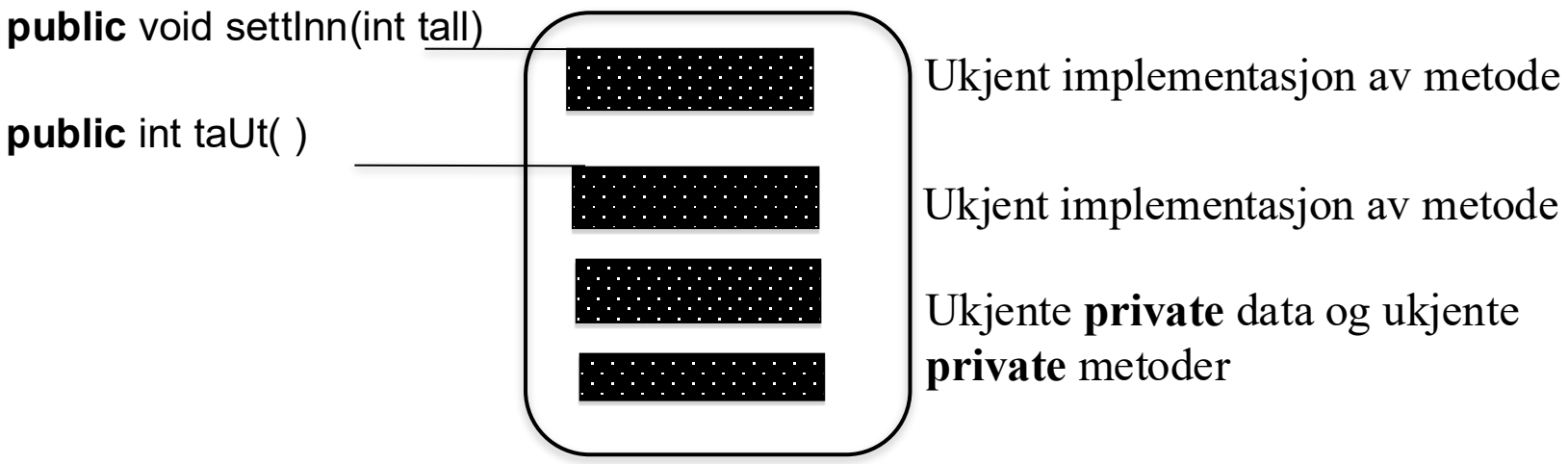
- Navnet på et interface kan brukes som typenavn når vi lager referanser
- Poenget med et interface er å spesifisere **hva** som skal gjøres (ikke hvordan)
- Ofte er det flere implementasjoner av et interface (flere klasser implementerer det)
- En klasse kan implementere (flere) interface samtidig som 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

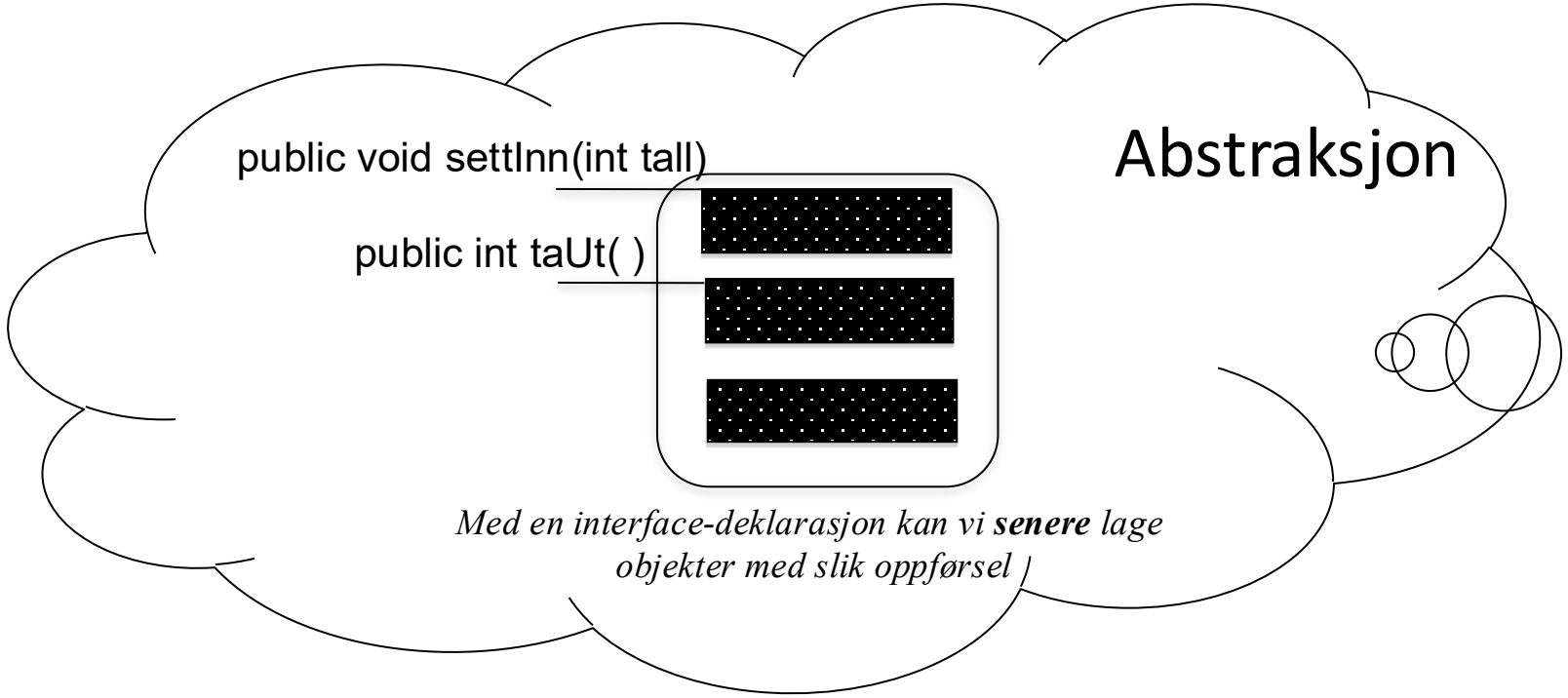
Menti-quiz om interface

Interface som spesifikasjon

Objektorientering handler mye om *innkapsling* og *abstraksjon*



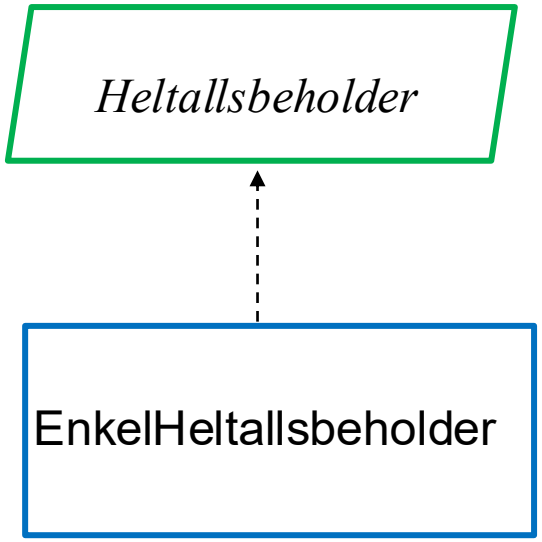
Objekt som kan ta vare på heltall (heltallsbeholder)




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

~~new Heltallsbeholder()~~

Interface: Klassehierarki og Java-kode



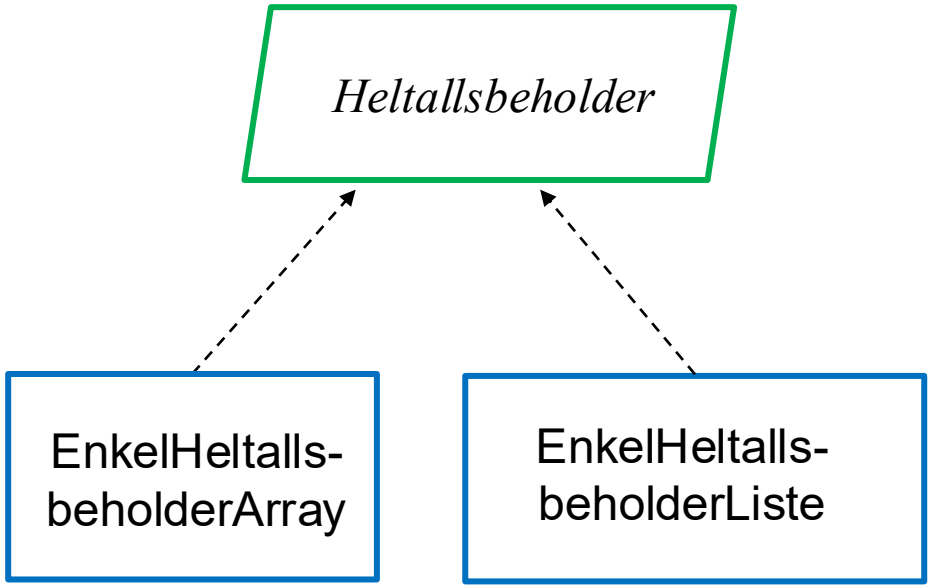

```java
interface Heltallsbeholder {
  void settInn(int tall);
  int taUt( );
}
```

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

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

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

VIKTIG

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( );
```

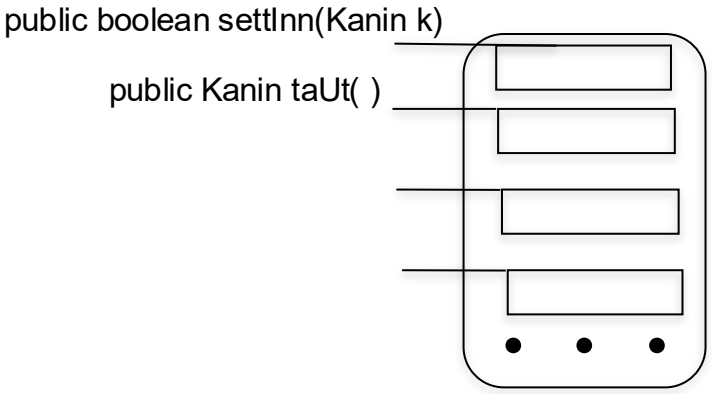
Resten av programmet er uforandret

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() {
      ...
   }
}
```





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;
    }
    return false;
  }
  @Override
  public Kanin taUt() {
    Kanin k = denne;
    denne = null;
    return k;
  }
}
```

Interface for å definere ”public-metodene”:

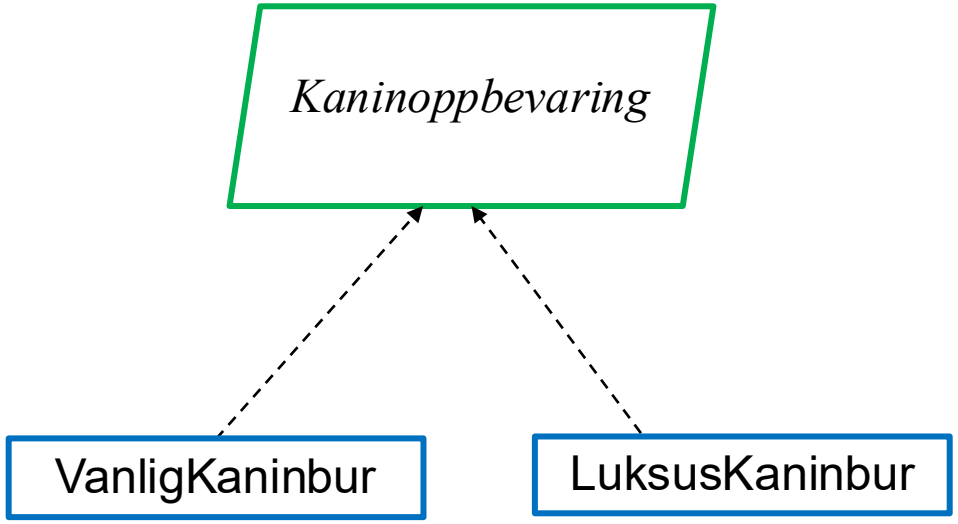
Vi kan lage forskjellige implementasjoner



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





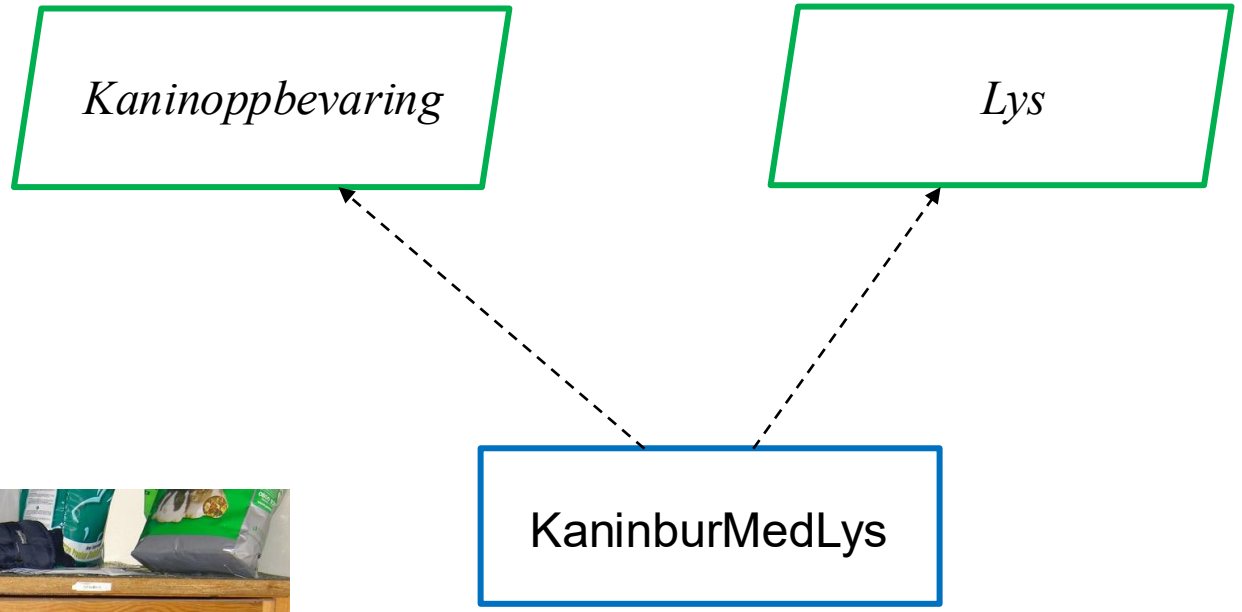




KaninburMedLys



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;

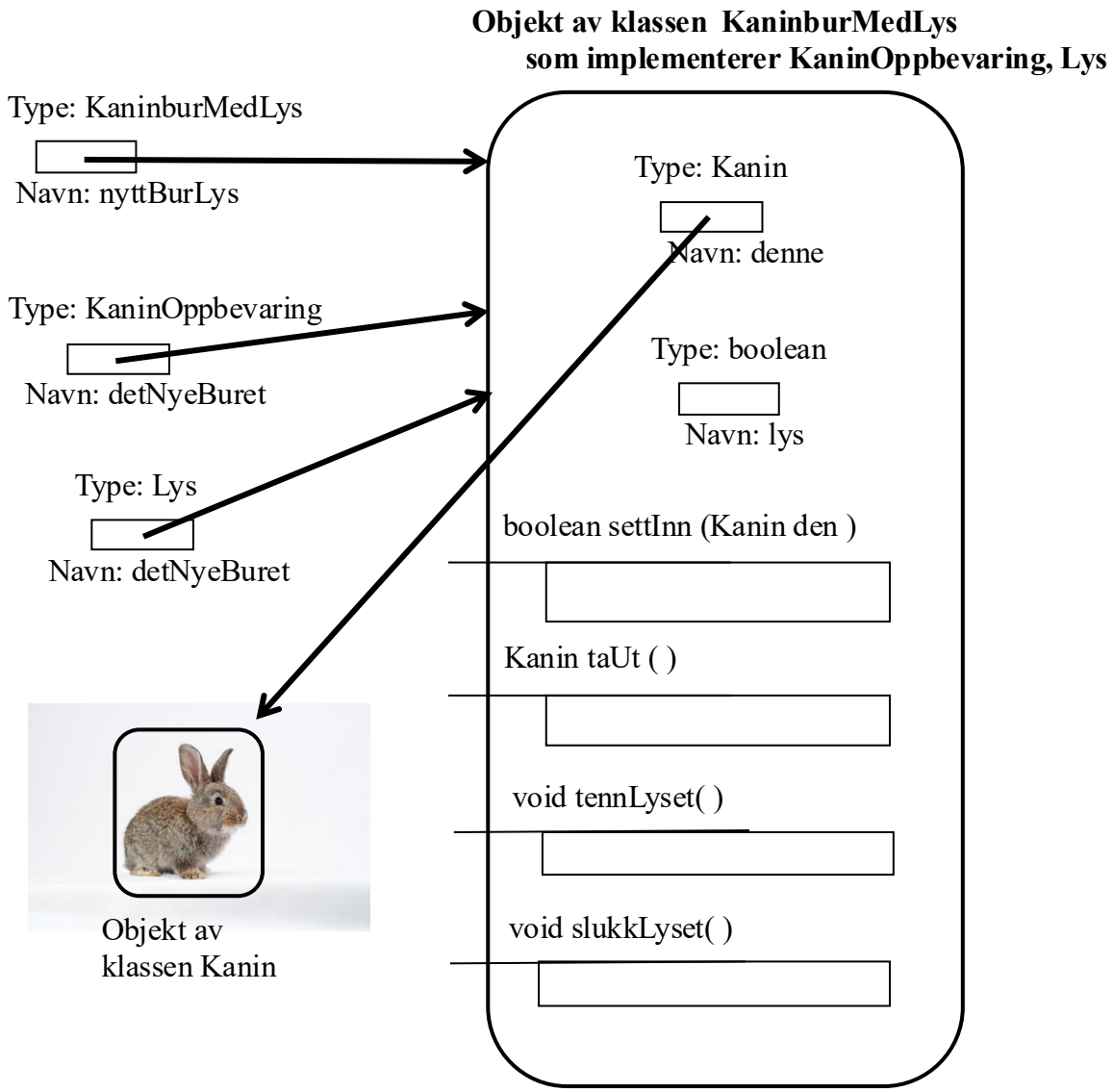
  @Override
  public boolean settInn(Kanin k) {
    ...
  }
  @Override
  public Kanin taUt() {
    ...
  }
  @Override
  public void tennLyset()  {lys = true;}
  @Override
  public void slukkLyset() {lys = false;}
}
```

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

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



Ett objekt– to grensesnitt – tre briller




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

```java
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;
    }
    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;}
}
```

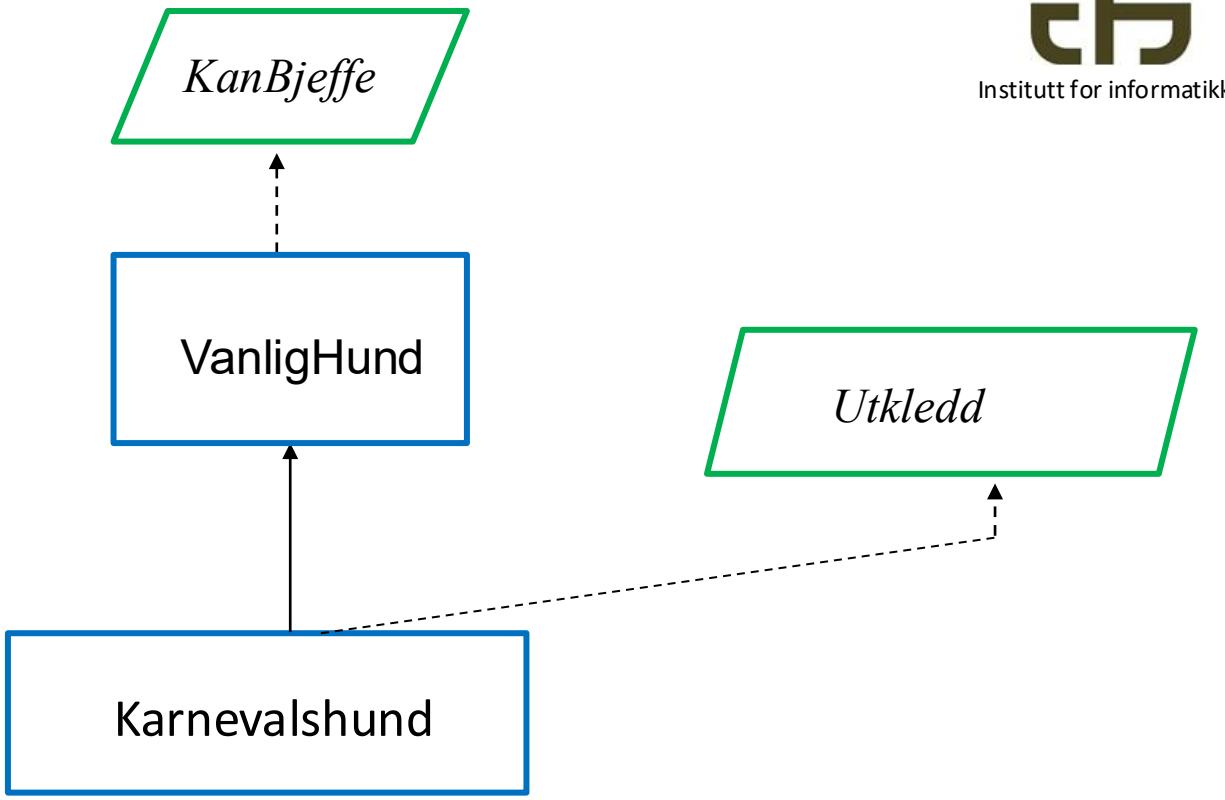
```
interface KaninOppbevaring {
  public boolean settInn(Kanin k);
  public Kanin taUt( );
}
```

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



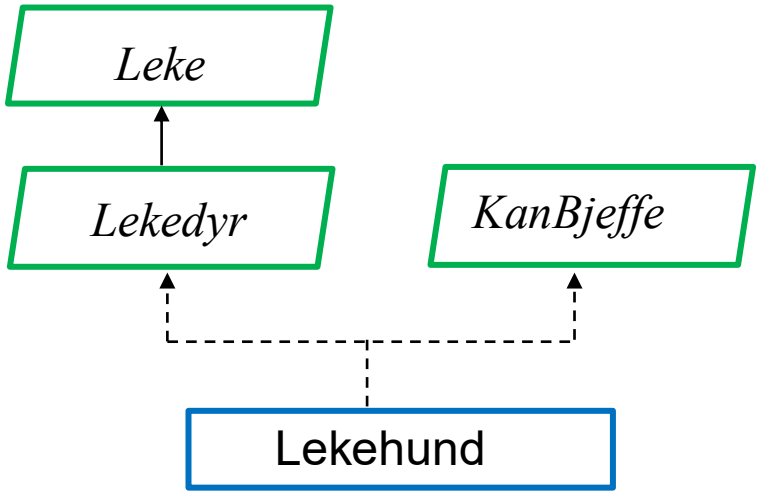


Institutt for informatikk



> Hvis tid: Live-koding!





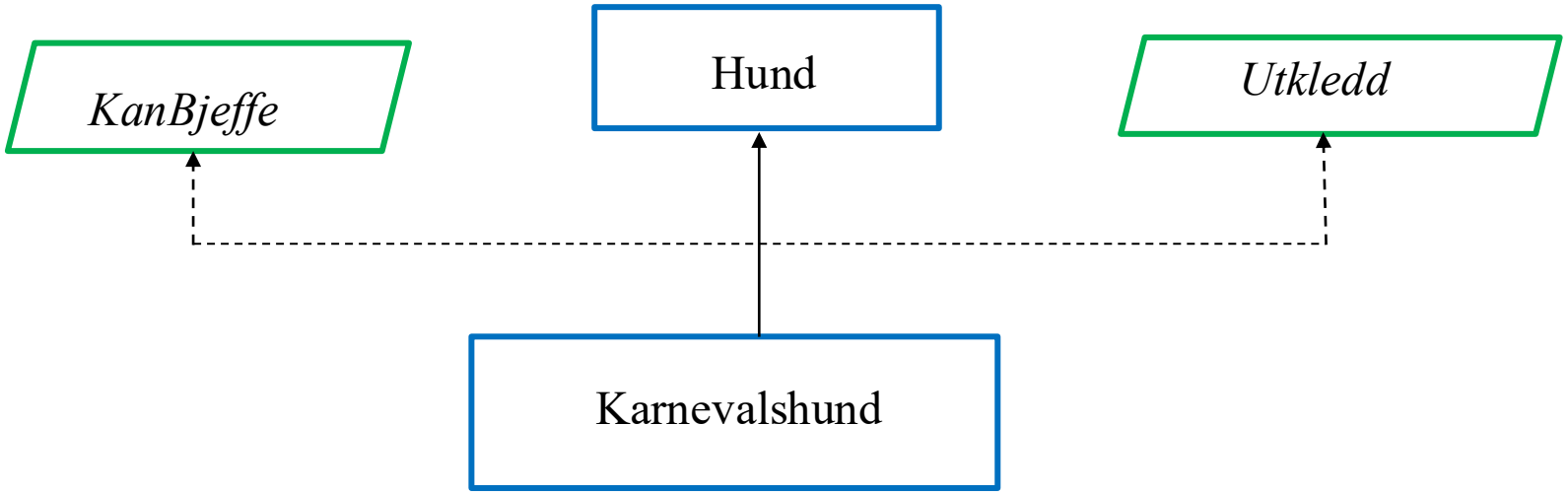
> Hvis tid: Live-koding!

Karnevalshund



Foto: AP

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



Flere eksempler: En klasse – mange grensesnitt


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

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

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

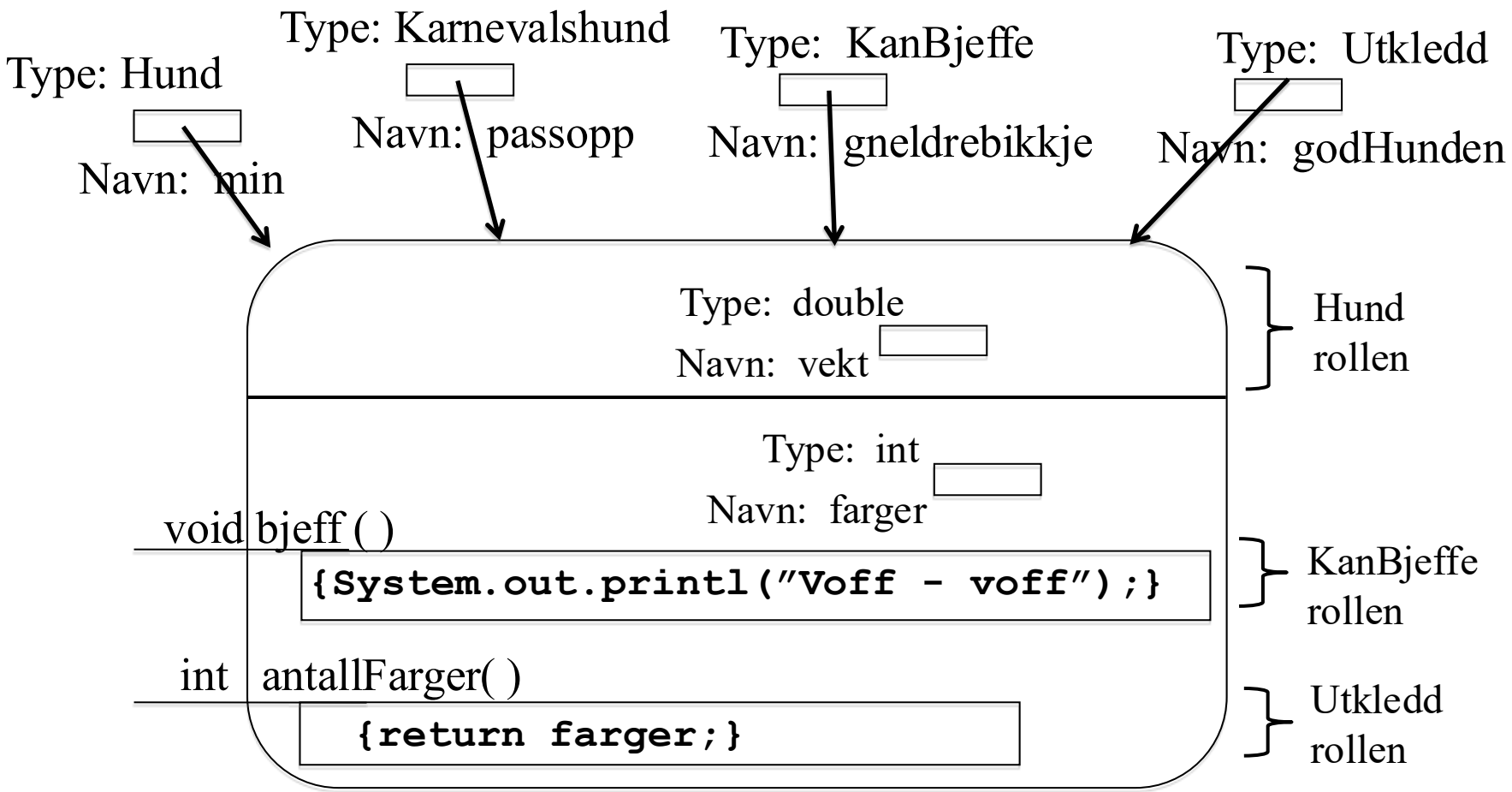
Foto: AP



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

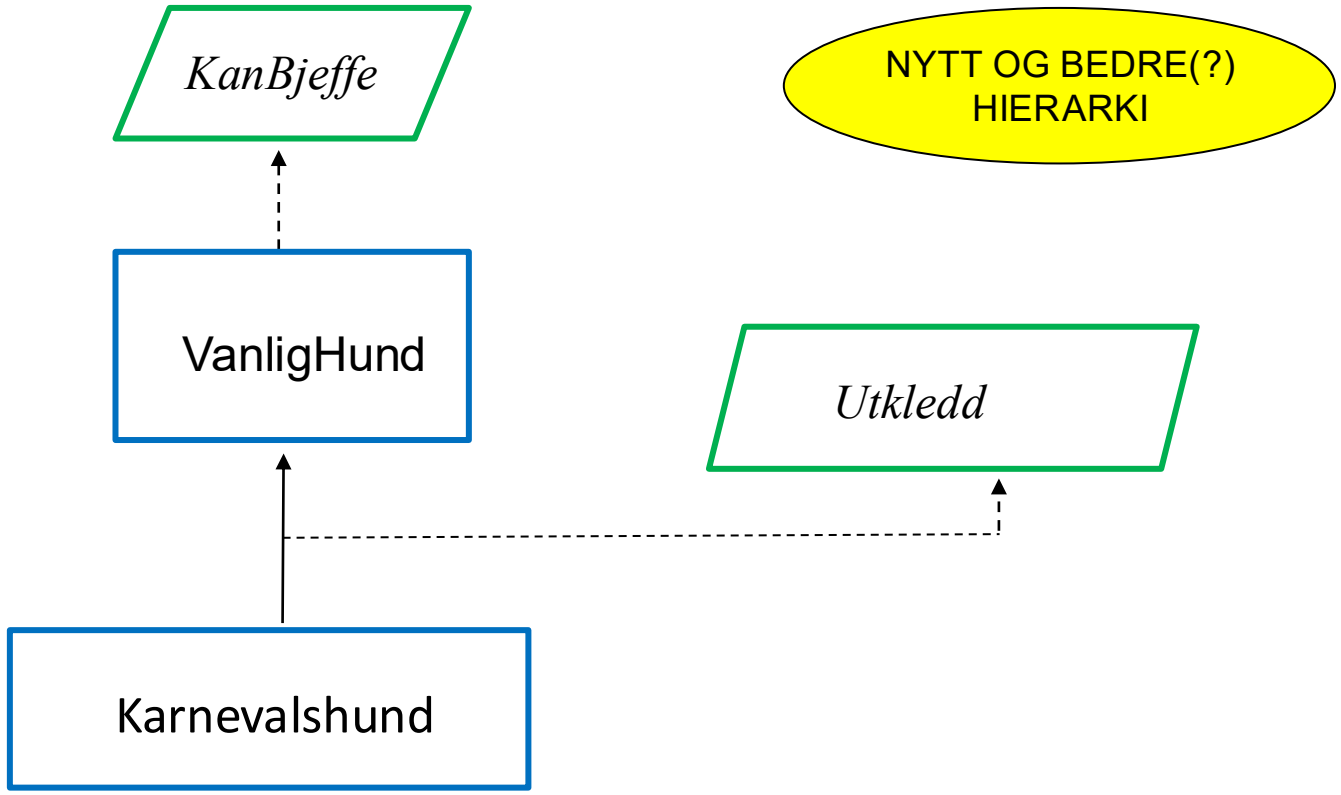
```
Karnevalshund passopp = new Karnevalshund( );

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

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

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

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



Foto: AP

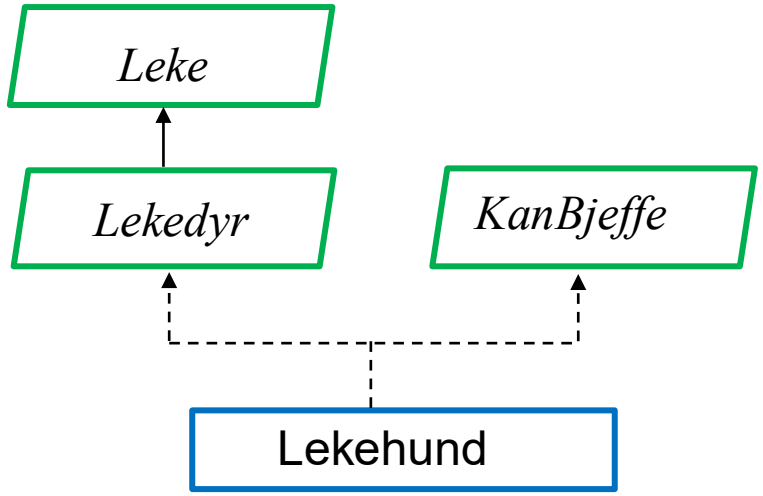
```java
class Karnevalshund extends VanligHund implements Utkledd {
  protected int farger
  public Karnevallshund (int frg) {
    farger = frg;
  }
  @Override
  public int antallFarger() {
    return farger;
  }
}
```

```
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"); }
}
```



> Ferdig med live-koding, tilbake til slides

Tilbake til skatteetaten: Både biler og ost skal skattlegges
Ett grensesnitt, flere klasser.

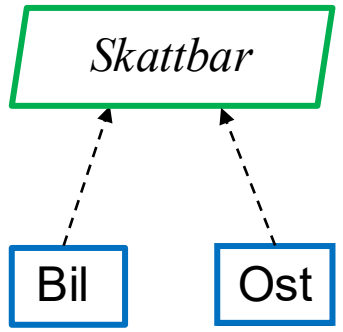

```
interface Skattbar{                  // Skatt på importerte varer
  int skatt();
}

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

  @Override
  public int skatt(){ return importpris; }
  ...
}

class Ost  implements Skattbar {  // Ost: 200% skatt
   protected int importprisPrKg;
   public Ost (...) { ... }

   @Override
   public int skatt(){ return ... * 2; }
}
```



Dette eksemplet har store likheter med Skatte-eksemplet fra forrige uke

Hva er likt?

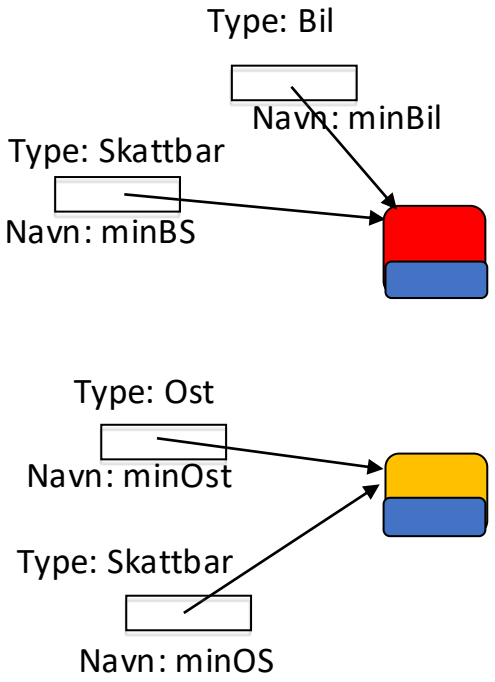
Hva er forskjellig?

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)

VIKTIG EKSEMPEL

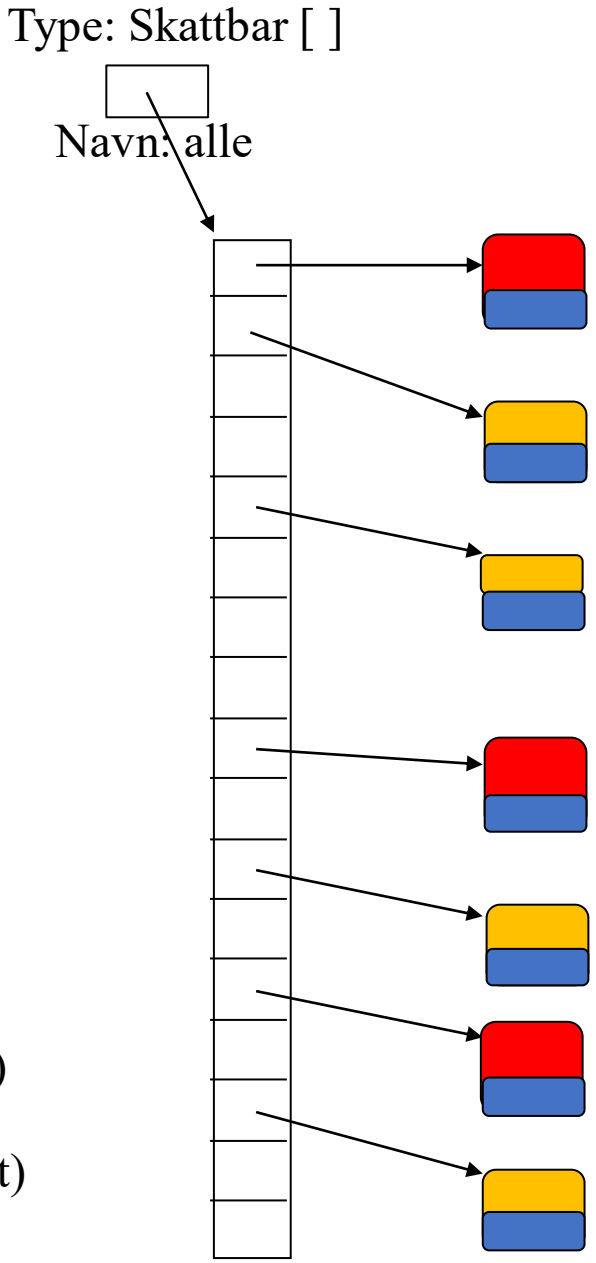
Samlet import-skatt


```
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)

Klassene Bil og Ost – Full kode


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

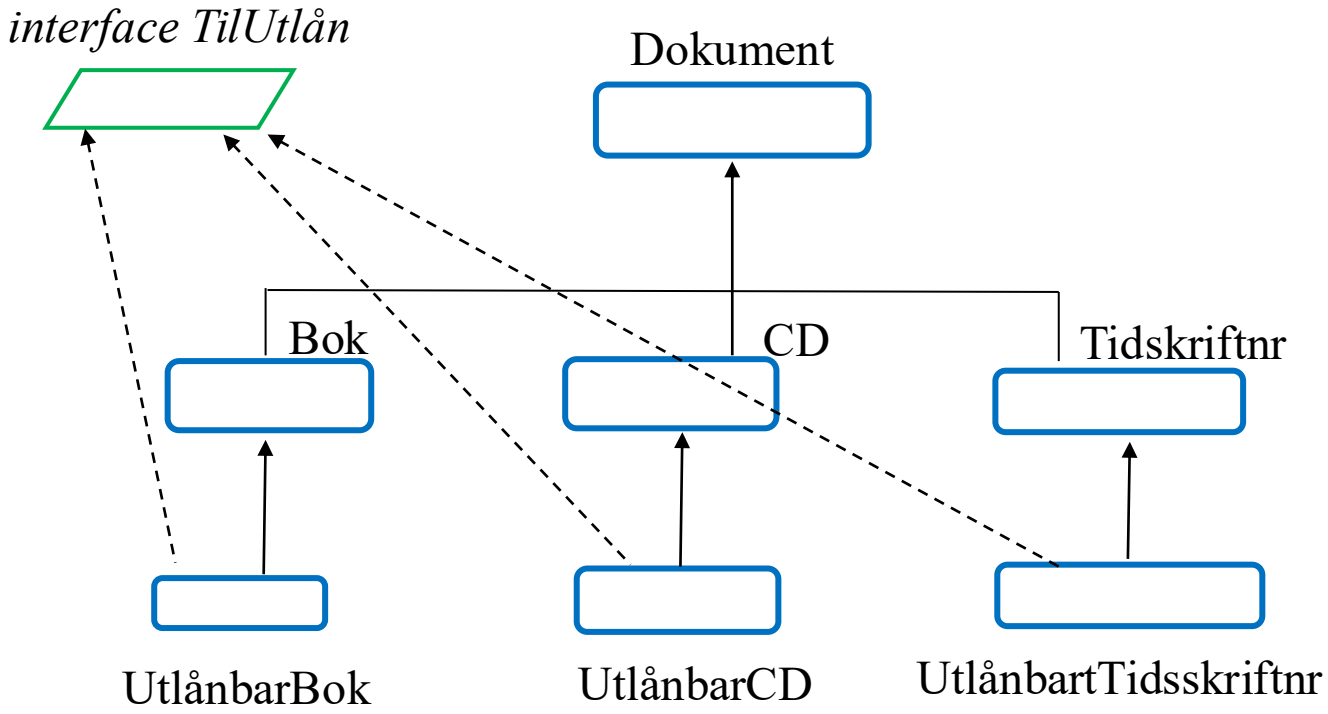
class Bil implements Skattbar {  // Bil: 100% skatt
  final public String regNr;
  protected int importpris;
  public Bil (String reg, int imppris) {
    regNr = reg; importpris = imppris;
  }
  @Override
  public int skatt(){ return importpris; }
}

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; }
}
```

Hoved "take-away" i dag

To hoved-grunner til å bruke interface:

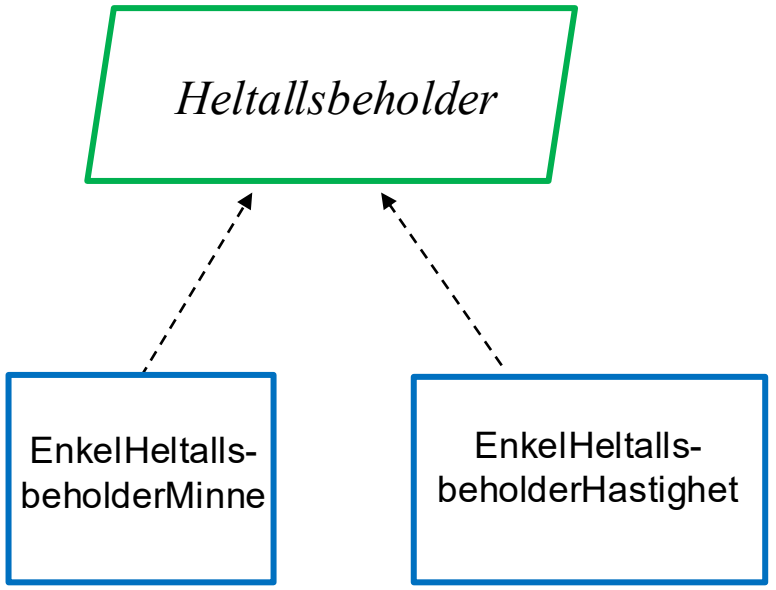
Flere klasser på tvers av et arvehierarki kan spille samme rolle



+ Både Biler og Oster er *Skattbare*

Forskjellig/ukjent implementasjon men samme grensesnitt: abstraksjon

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



Oppsummering om interface

Java har en mekanisme og et nøkkelord `interface` som:

- Definerer et grensesnitt i form av en rolle eller en kontrakt
  - Abstraksjon / Innkapsling / Skjuling av detaljer
  - Helt konkret er grensesnittet en samling metodesignaturer
    - Bruk ; istedenfor { . . . } bak signaturen til metoden når du skriver et interface
- Kan brukes til «multippel arv» av roller
  - Men alle metodene må implementeres på nytt
  - Så Java har (i utgangspunktet) ikke multippel arv av kode
- Kan brukes som en referansetype

Neste uke:
beholdere med Siri!

