

IN1010 våren 2019

Onsdag 15. mai

Rask repetisjon av subklasser og tråder (pluss μ nytt)

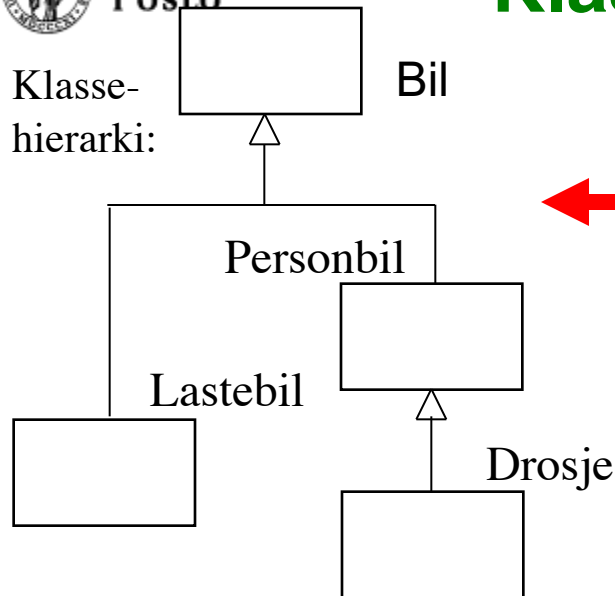
Stein Gjessing
Institutt for informatikk
Universitetet i Oslo

Innledning

- Dette er 41 lysark som det ikke er mulig å gå gjennom på én time
- Disse lysarkene representerer det jeg mener er det viktigste i pensum om subklasser og tråder
- Jeg kommer til å peke på noen få viktige poeng, og så er meningen at dere kan bruke lysarkene når dere skal repetere stoffet på egenhånd
- Lysarkene 27 – 31 handler om invarianter. Vi har ikke snakket mye om dette på forelesningene, men det er allikevel viktig, og de tre lysarkene om invarianter i løkker er helt nye

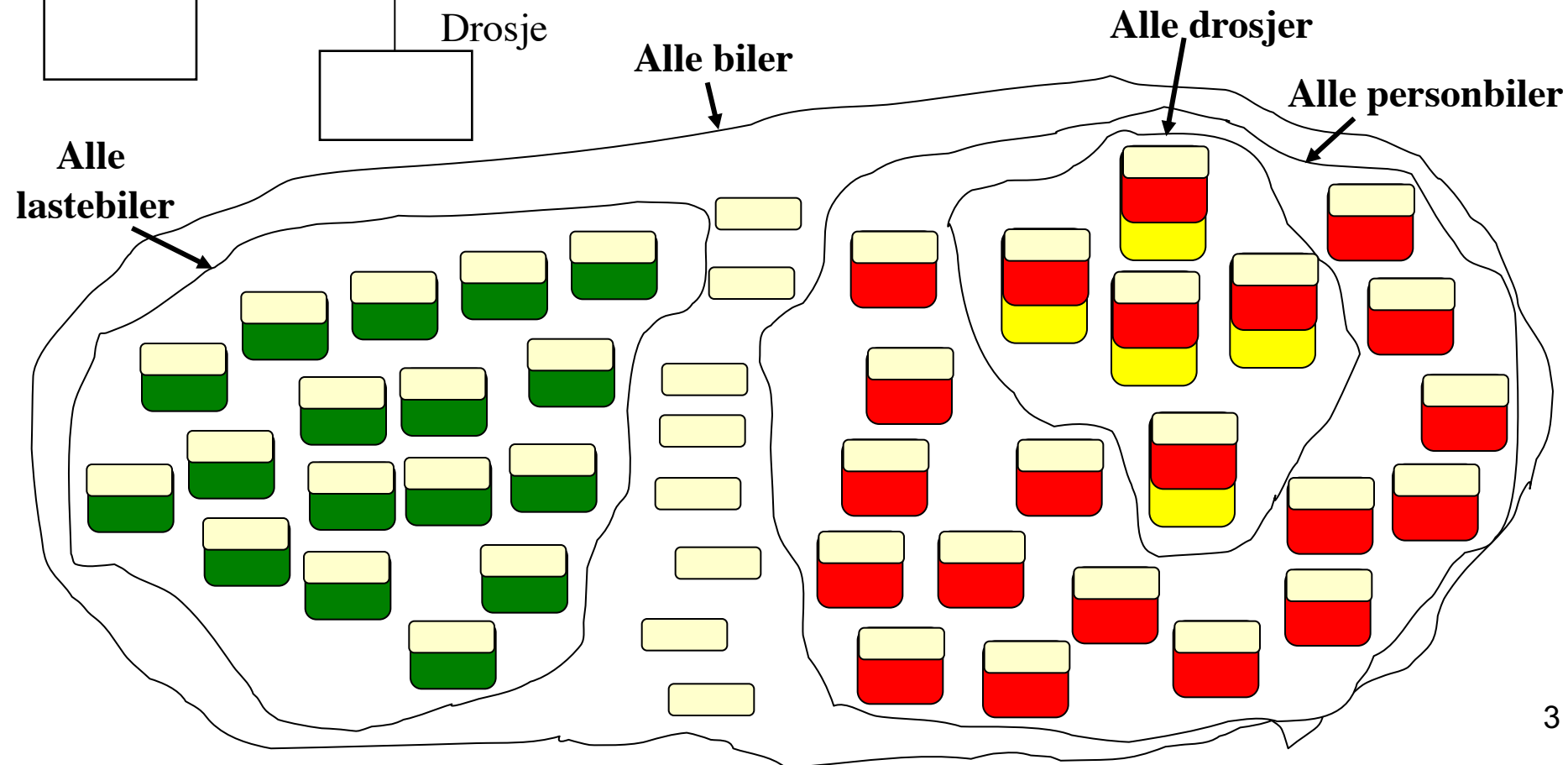
Klasser - Subklasser

Klasse-
hierarki:



```

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



Private og public i subklasser

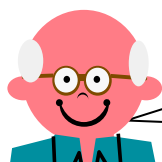
Private i en klasse gjør at ingen utenom denne klassen (og heller ikke subklasser) kan se denne egenskapen

Protected i en klasse gjør at alle subklasser kan se denne egenskapen
Men ingen utenfor klassen (bortsett fra i samme katalog/pakke)

Public er som før

```
class Person {
    protected String navn;
    protected int tlfnr;

    public boolean gyldigTlfnr() {
        return tlfnr >= 10000000 && tlfnr <= 99999999;
    }
}
```

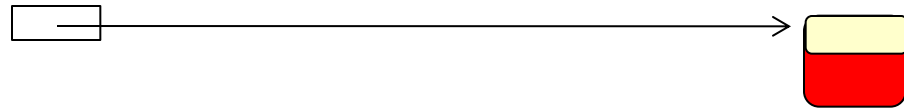


Nytt reservert ord i Java:
protected

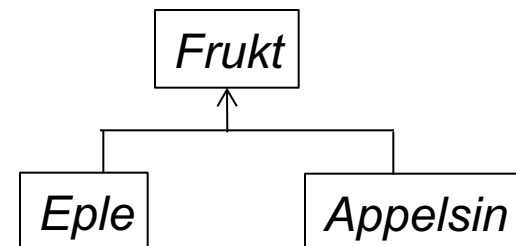
Hva slags objekt er dette?

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 { .. }
```



Konvertering mellom flere nivåer

```
MasterStudent master = new MasterStudent();
```

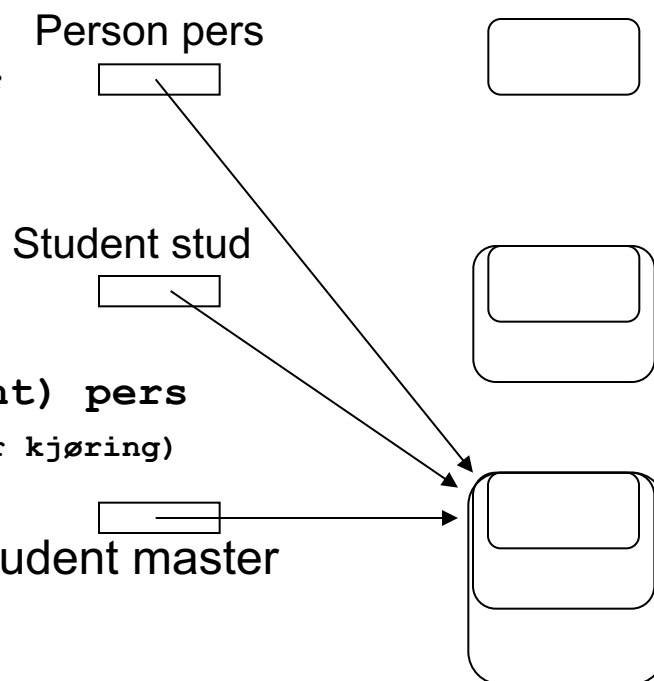


- **Konvertering oppover:**

```
Student stud = master;
Person pers = master;
```

- **Konvertering nedover:**

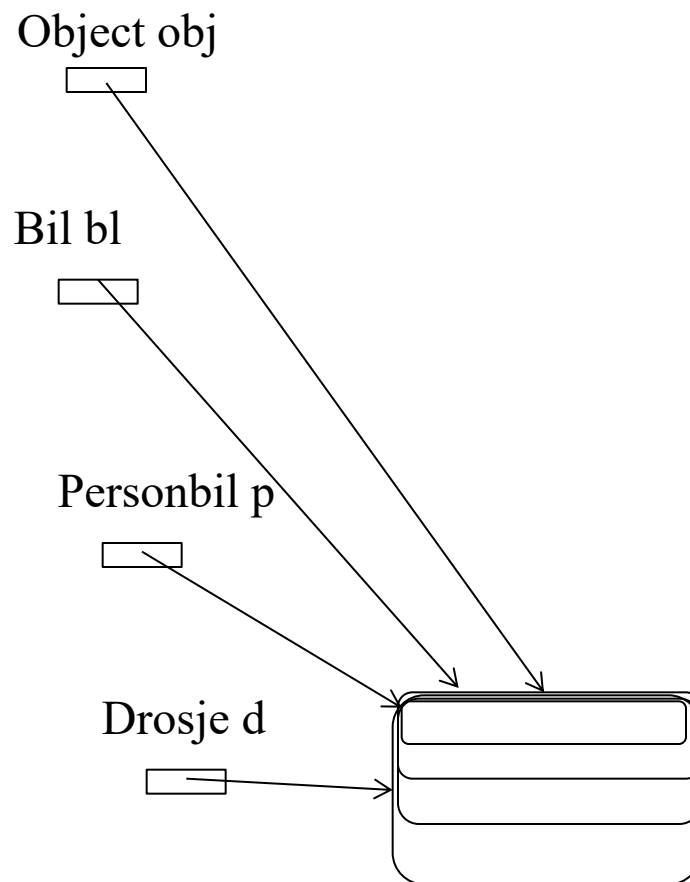
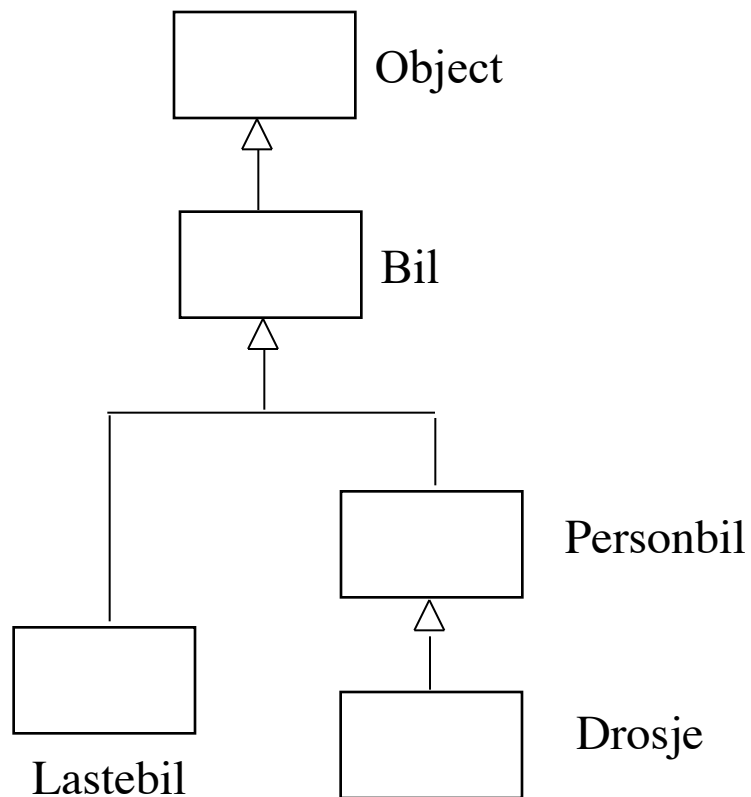
```
stud = (Student) pers
master = (MasterStudent) pers
(fordi dette krever kontroll under kjøring)
```



***Regel: "Alle referanser har lov til å peke bortover og nedover"
(men ikke "oppover")***

Eksempel

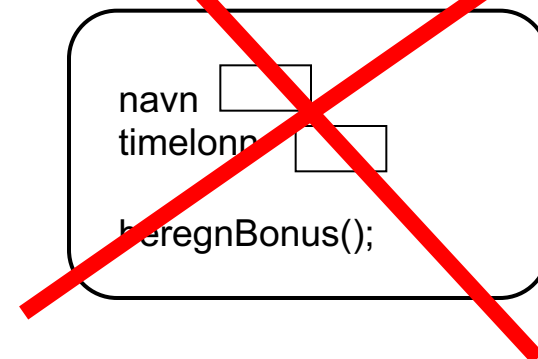
Klasshierarki:



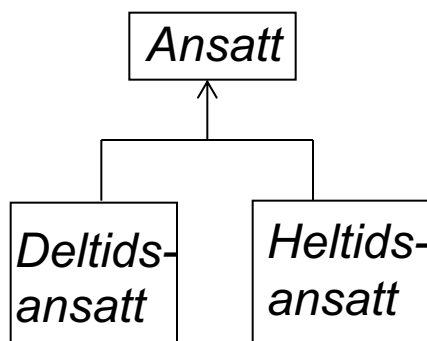
Abstrakte metoder og klasser

```
abstract class Ansatt {  
    protected String navn;  
    protected double timelonn;  
  
    public abstract double beregnBonus();  
}
```

Ikke lov å si
`new Ansatt()` !

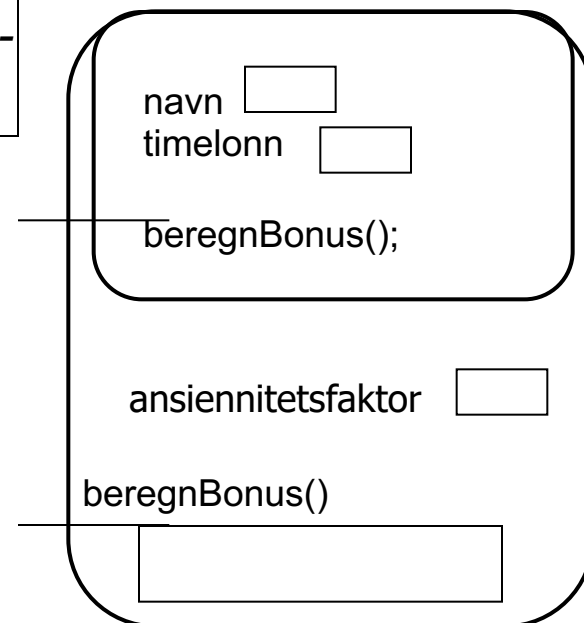


```
class Deltidsansatt extends Ansatt {  
    public double beregnBonus() {  
        return 0;  
    }  
}
```



```
class Heltidsansatt extends Ansatt {  
    protected int ansiennitetsfaktor;  
  
    public double beregnBonus() {  
        return timelonn* ansiennitetsfaktor;  
    }  
}
```

Heltidsansatt objekt

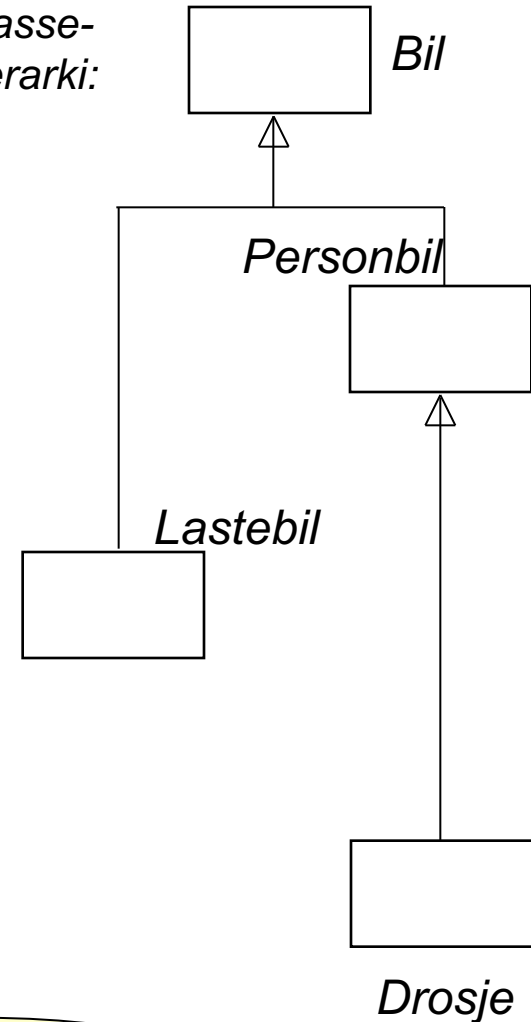




Polymorfi: eksempel

```
class Bil {  
    protected int pris;  
    public int skatt( ) {return pris;}  
}  
  
class Personbil extends Bil {  
    protected int antallPassasjer;  
    public int skatt( ) {return pris * 2;}  
}  
  
class Lastebil extends Bil {  
    protected double lastevekt;  
    public int skatt ( ) {return pris / 2;}  
}  
  
class Drosje extends Personbil {  
    protected String loyveld;  
    public int skatt ( ) {return pris / 4;}  
}
```

Klasse-
hierarki:



Se også lysark
nummer 19



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:

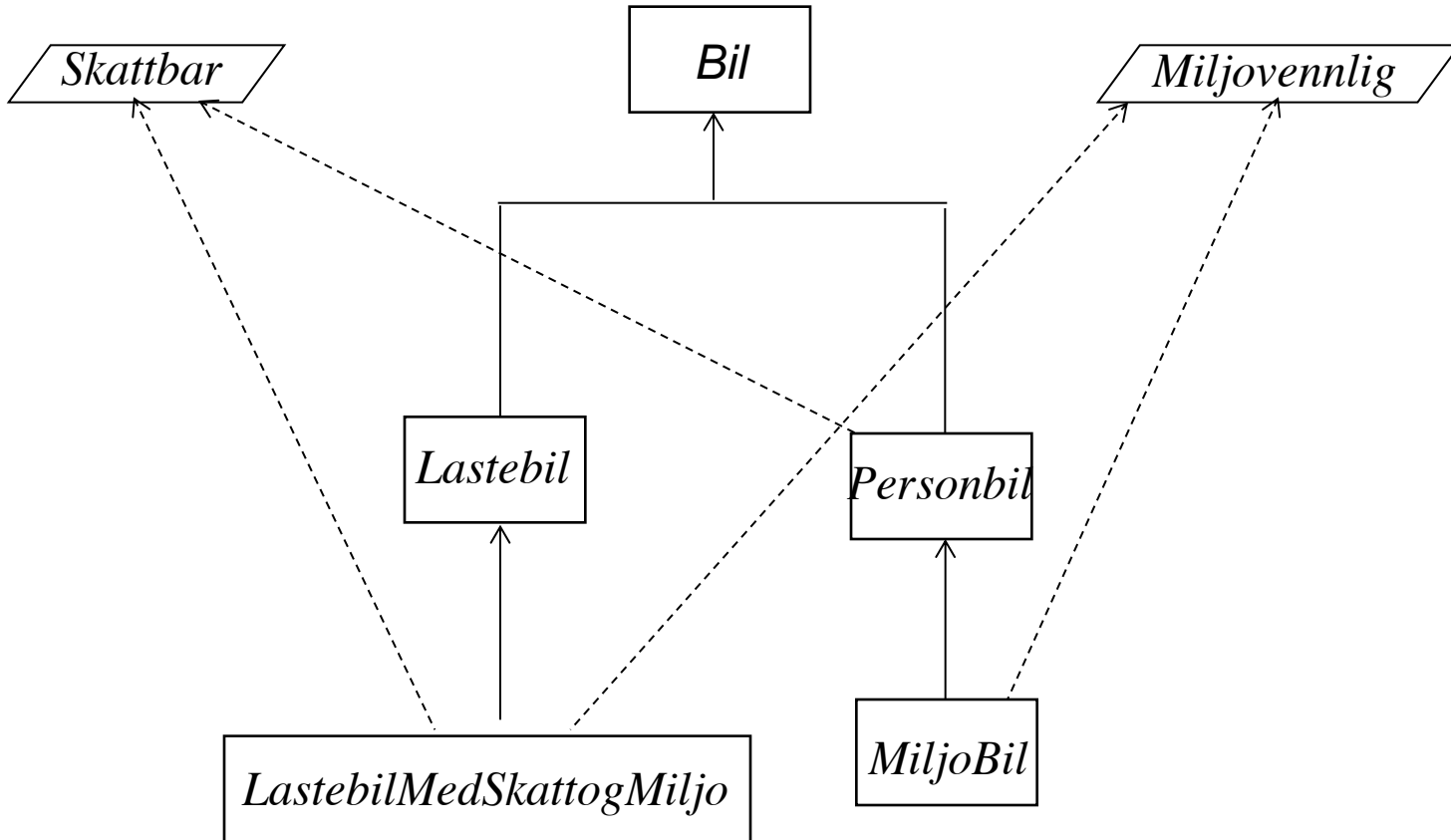
```
// I klassen Person:  
public void skrivData() {  
    System.out.println("Navn: " + navn);  
    System.out.println("Telefon: " + tlfnr);  
}  
  
// I klassen Student:  
public void skrivData() {  
    super.skrivData();  
    System.out.println("Studieprogram: " + program);  
}  
  
// Tilsvarende i klassen Ansatt:  
public void skrivData() {  
    super.skrivData();  
    System.out.println("Lønnstrinn: " + lønnstrinn);  
    System.out.println("Timer: " + antallTimer);  
}
```



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()`;

Interface



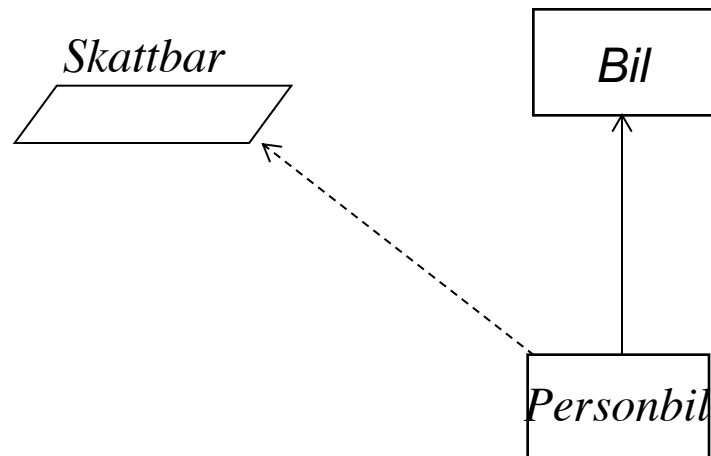
Implementere et interface: implements

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() ;  
}
```




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();}
}

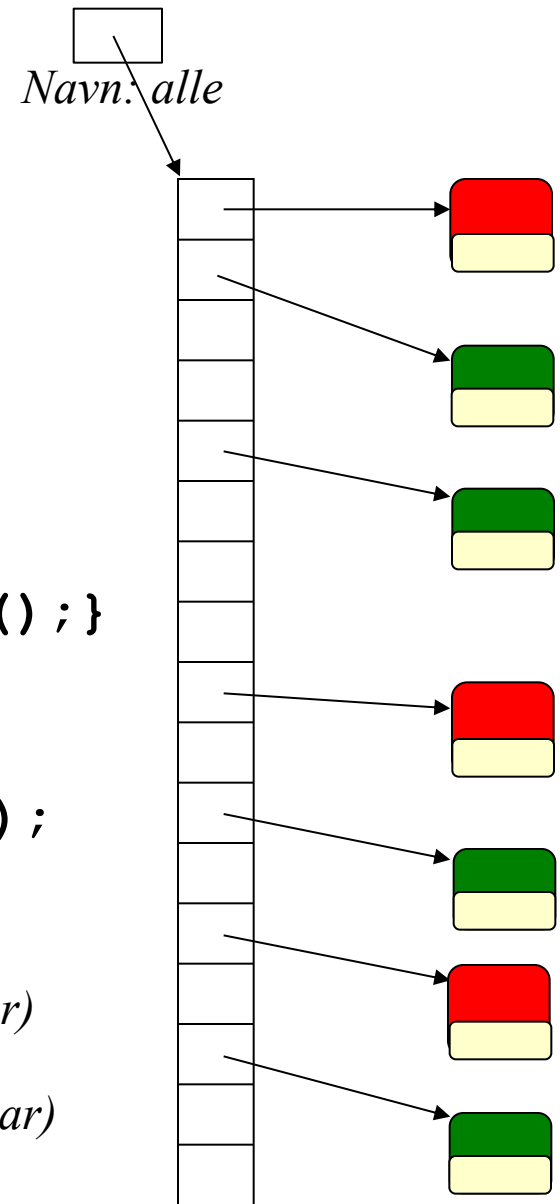
System.out.println("Total skatt: " +
                    totalSkatt);
```

 Rollen Skattbar

 Rollen Bil (untatt Skattbar)

 Rollen Ost (untatt Skattbar)

Type: Skattbar []



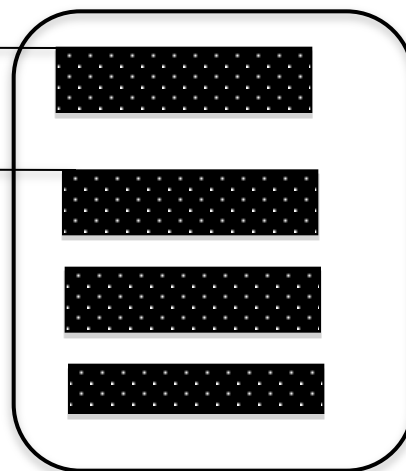
Veldig viktig og bra eksempel. Dagens rosin.



Objektorientering handler om å tydeliggjøre objektene public-metoder. Interface gjør det.

public void settInn(int tall)

public int taUt()

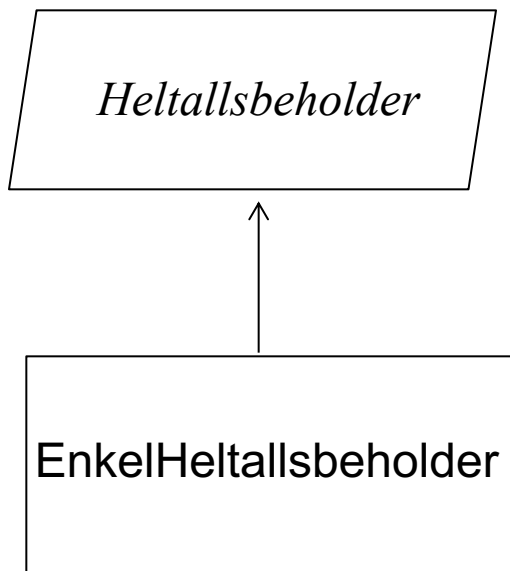


Ukjent implementasjon av metode

Ukjent implementasjon av metode

Ukjente **private** data og ukjente **private** metoder

Interface: Klassehierarki og Java-kode



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

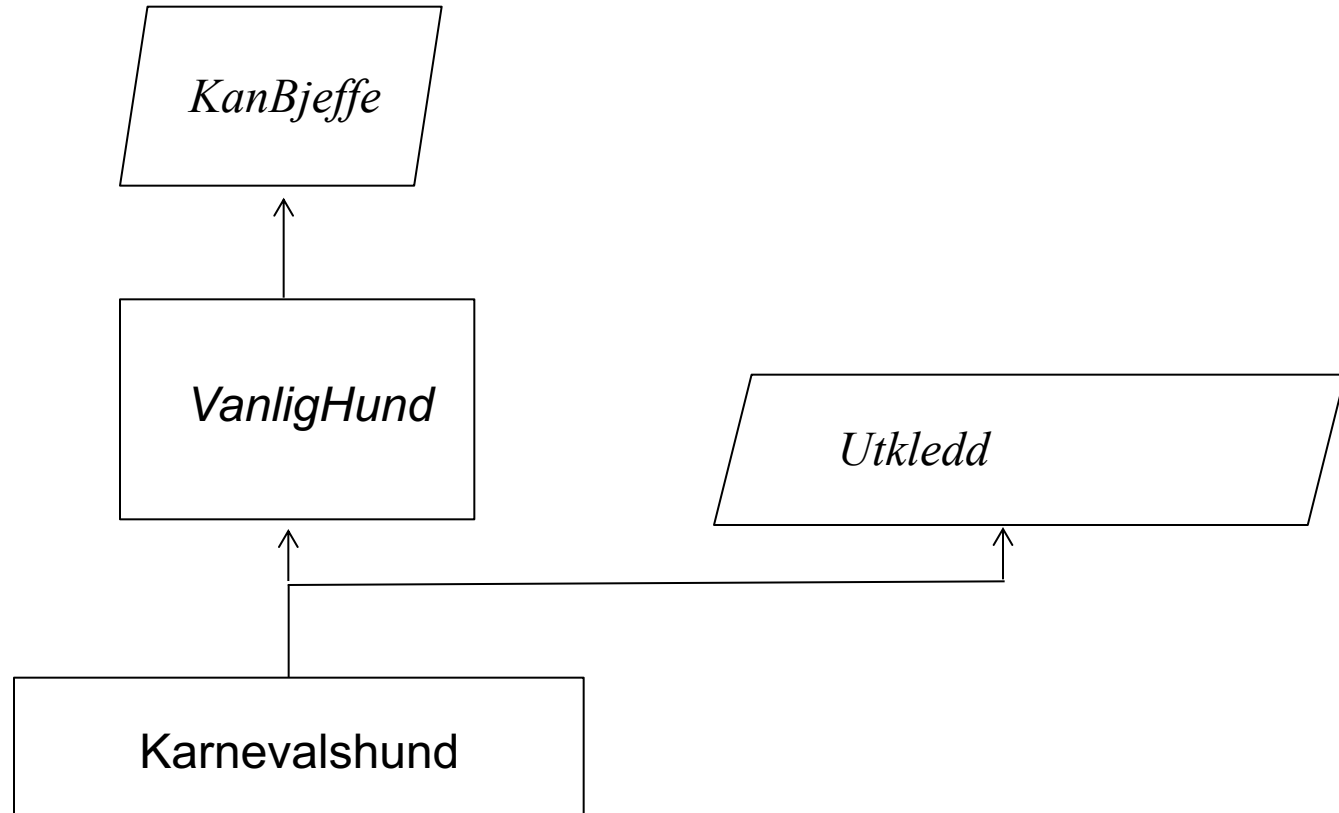
```
class EnkelHeltallsbeholder  
    implements Heltallsbeholder {  
    protected int [ ] tallene = new int [100];  
    protected int antall;  
    public void settInn(int tall) { . . . }  
    public int taUt( ) { . . . }  
}
```

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

Karnevalshund



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



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

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

```
class VanligHund implements KanBjefte {  
    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;  
    }  
}
```



Foto: AP



Generiske interface

Inteface med parametre

- På samme måte som klasser, kan interface lages med parametre.
- `class GeneriskBeholderTilEn <E> { . . . }`
- `interface Beholder <E> { . . . }`

new GeneriskBeholderTilEn<Bil>();

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



```
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;}  
}
```

Oppsummering

*Hva brukes **interface** til?*



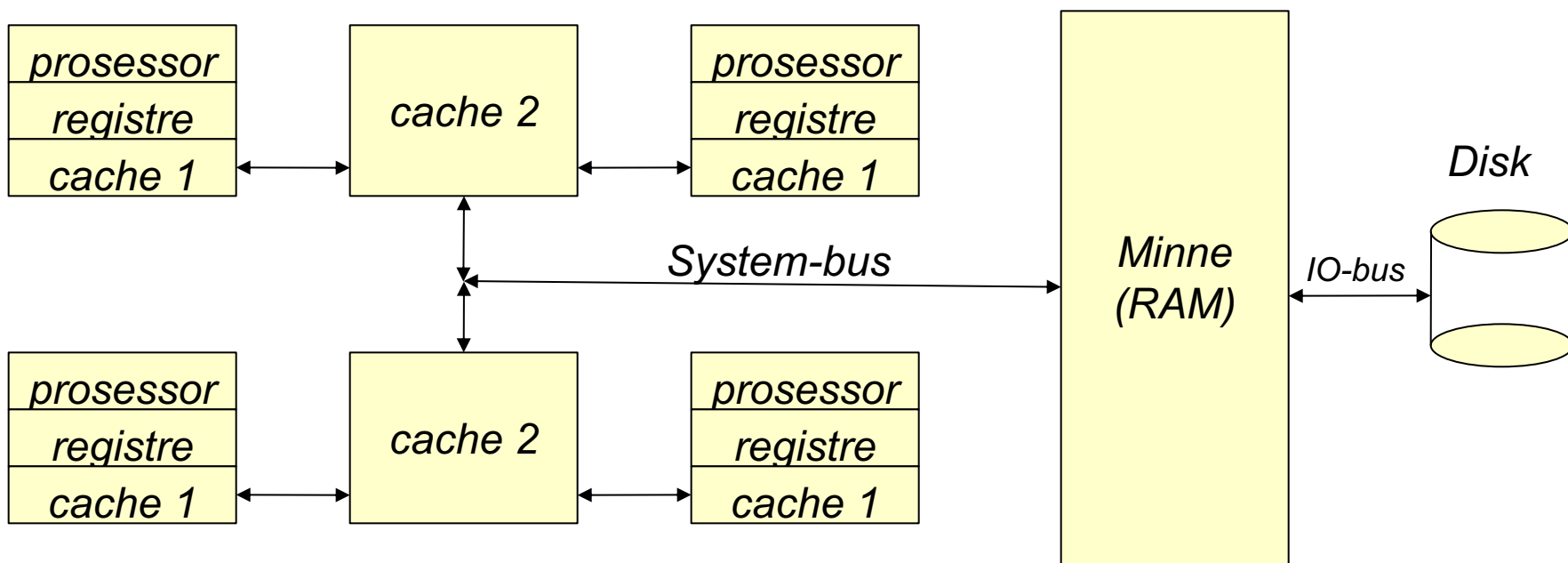
Vet hjelp av interface kan forskjellige klasser og objekter ha det samme grensesnittet. Dette er en fordel når vi skal beskrive objekter med felles egenskaper.

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

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

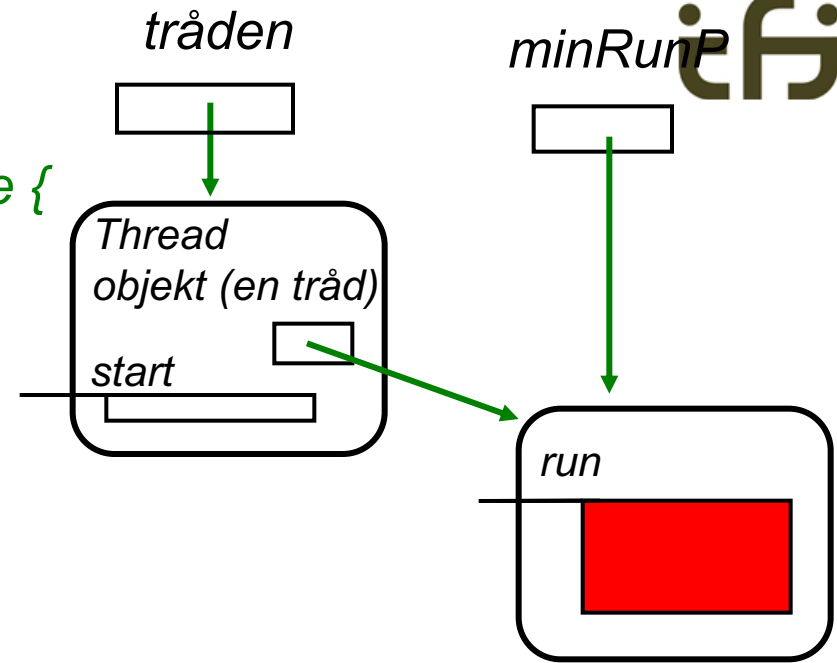
Tråder

Datamaskinarkitektur



Tråder i Java

```
class MinRun implements Runnable {  
    <datastruktur>  
    public void run( ) {  
        while ( <mer å gjøre> ) {  
            <gjør noe>;  
            ....  
        }  
    }  
}
```



En tråd lages og startes opp slik:

```
Runnable minRunP = new MinRun();  
Thread tråden = new Thread(minRunP);  
tråden.start( );
```



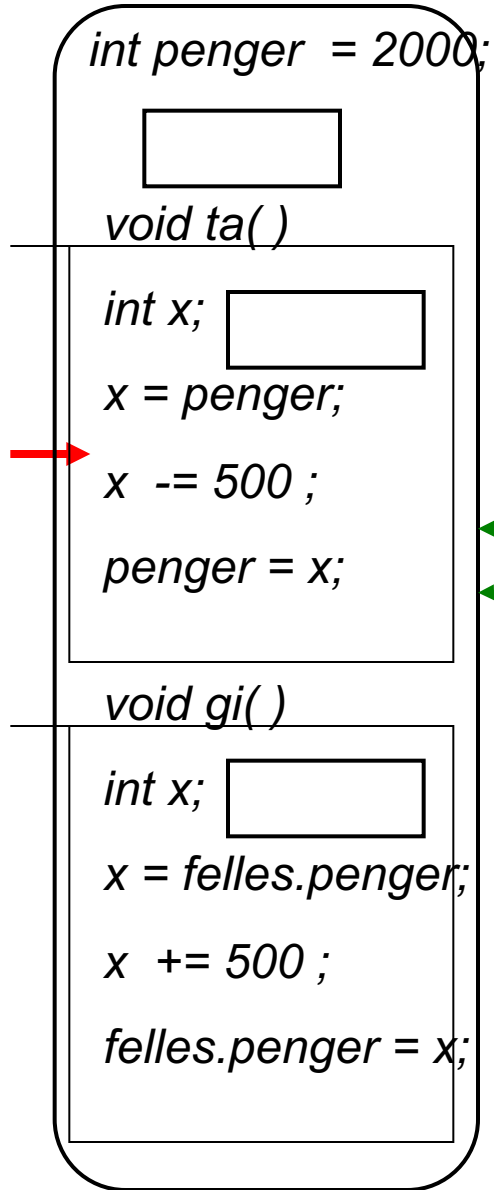
Her går den nye og den gamle tråden (dette programmet), videre hver for seg

start() er en metode i Thread som må kalles opp for å få startet tråden. start-metoden vil igjen kalle metoden run (som vi selv programmerer).

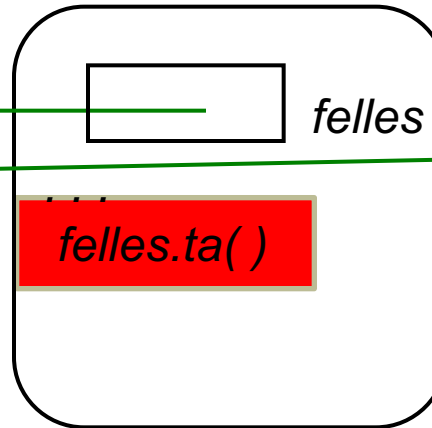


Felles data:

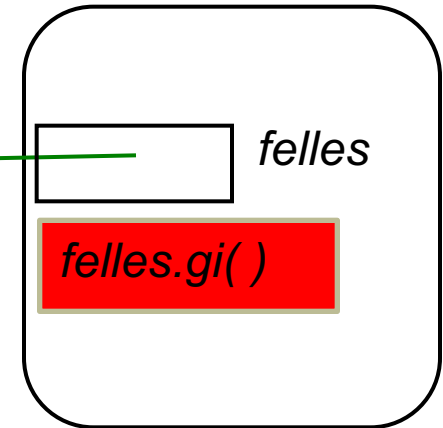
Enkelt eksempel på at to tråder kan ødelegge felles data



Tråden Ta:



Tråden Gi:




```
Lock laas = new ReentrantLock();
```

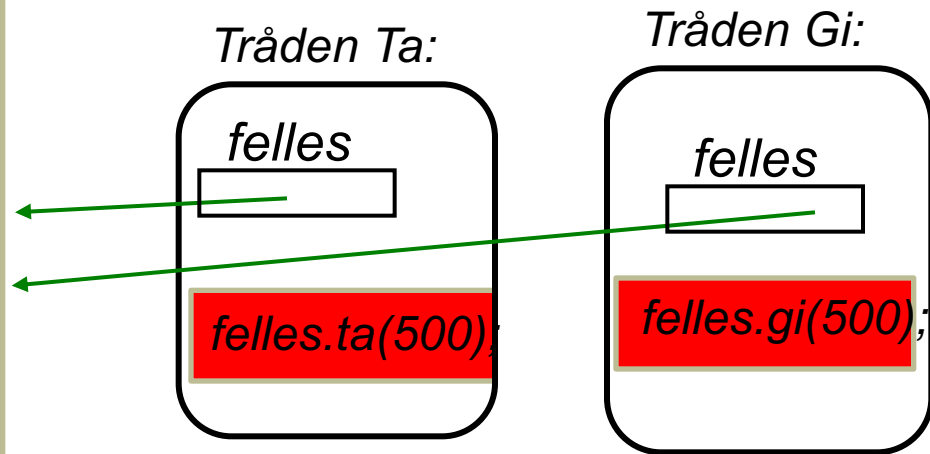
```
Int penger; 2000
```

```
public void gi ( int verdi ) throws InterruptedException
```

```
    laas.lock();  
    try {  
        penger = penger + verdi;  
    } finally {  
        laas.unlock()  
    }  
}
```

```
public void ta ( int verdi ) throws InterruptedException
```

```
    laas.lock();  
    try {  
        penger = penger - verdi;  
    } finally {  
        laas.unlock()  
    }  
}
```



En lås (laas) slik at bare en tråd
kommer inn i monitoren om gangen

```
(import java.concurrent.locks.*)
```



Legg merke til bruken av finally

```
void putInn (int verdi) throws InterruptedException {  
    laas.lock();  
    try {  
        :  
        :  
    } finally {  
        laas.unlock();  
    }  
}
```

Da blir `laas.unlock()` alltid utført !!



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.

Invarianter i løkker

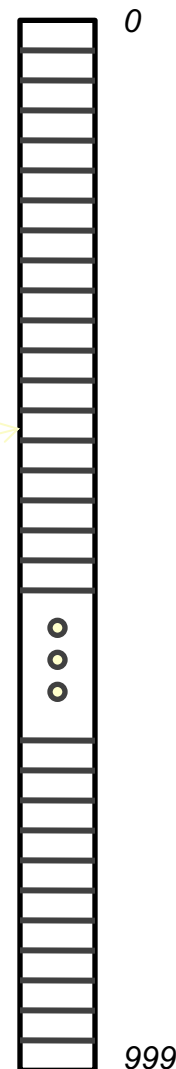
Eksempel: Finne minste verdi i tabell

minstTilNaa

minstTilNaa inneholder minste verdi i området fra og med tabell [0] til og med tabell[indeks]

= Inv(indeks):

indeks



Induksjons-basis: $indeks = 0$

Induksjons-skritt: $indeks = indeks + 1$

Resultat: $indeks = 999$:

minstTilNaa inneholder minste verdi i området fra og med tabell [0] til og med tabell[999]



Invarianter i løkker

Eksempel: Finne minste verdi i tabell

// Vi vet ingenting annet enn at tabell [0] til og med
// tabell[999] inneholder tall. Vi skal finne det minste

```
int minstTilNaa = tabell[0];
```

```
// minstTilNaa inneholder minste verdi i området  
// fra og med tabell [0] til og med tabell[0]
```

```
for (indeks = 1; indeks < 1000; indeks ++) {
```

```
// minstTilNaa inneholder minste verdi i området  
// fra og med tabell [0] til og med tabell[indeks-1]
```

```
if (minstTilNaa > tabell [indeks] ) minstTilNaa = tabell[indeks]
```

```
// minst TilNaa inneholder minste verdi i området  
// fra og med tabell [0] til og med tabell[indeks]
```

```
}
```

```
// Nå er indeks == 1000
```

```
// minstTilNaa inneholder minste verdi i området  
// fra og med tabell [0] til og med tabell[indeks-1]
```

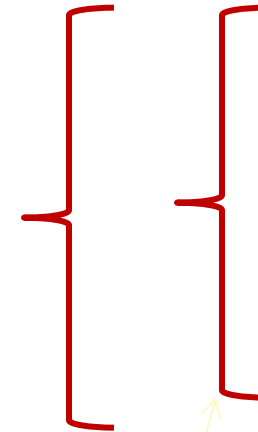
```
// Da følger:
```

```
// minstTilNaa inneholder minste verdi i området  
// fra og med tabell[0] til og med tabell[999] !!!!!
```


minstTilNaa

Inv(0)

Induksjons-
basis



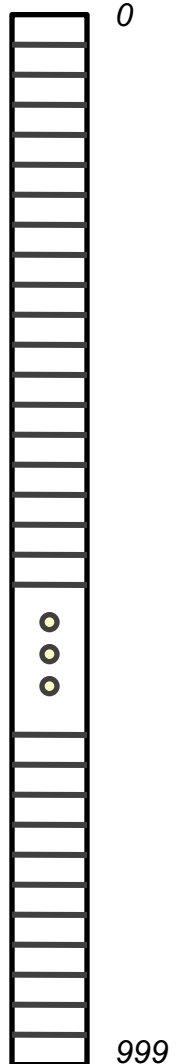
Inv(indeks-1)

Inv(indeks)

Hvis Inv(indeks) er sant og
vi utfører: indeks ++ så er
Inv(indeks-1) sant etterpå!

Induksjons-
skritt

resultat





Invarianter i løkker

Eksempel: Finne minste verdi i tabell


minstTilNaa

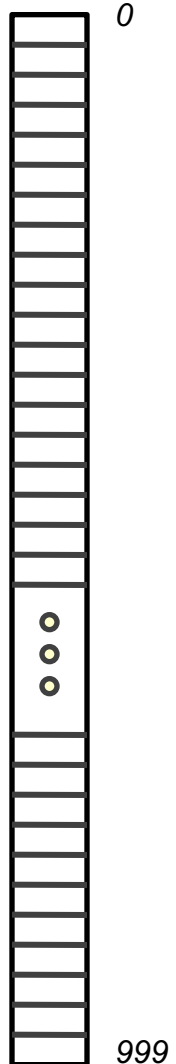


*// Vi vet ingenting annet enn at tabell [0] til og med
// tabell[999] inneholder tall. Vi skal finne det minste*

For-betingelse. Engelsk: Pre-condition

***Bak-betingelse.
Engelsk: Post-condition (Post-assertion)***

*// minstTilNaa inneholder minste verdi i området
// fra og med tabell[0] til og med tabell[999] !!!!!*



Invarianter i objekter (class invariante)

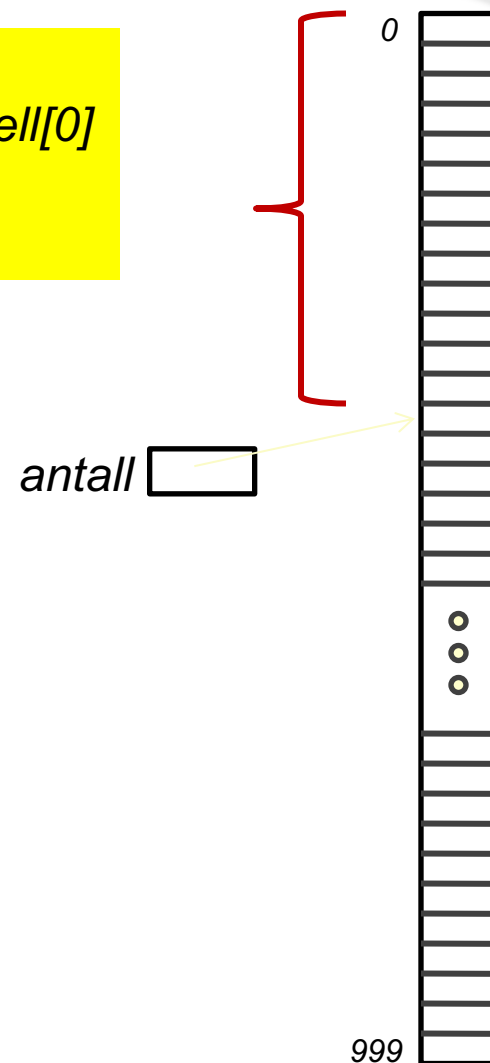
Dette har vi
snakket om

Invariant:
Alle dataene vi lagrer ligger i tabell[0]
til og med tabell [antall - 1] og
 $0 \leq \text{antall} \leq 1000$

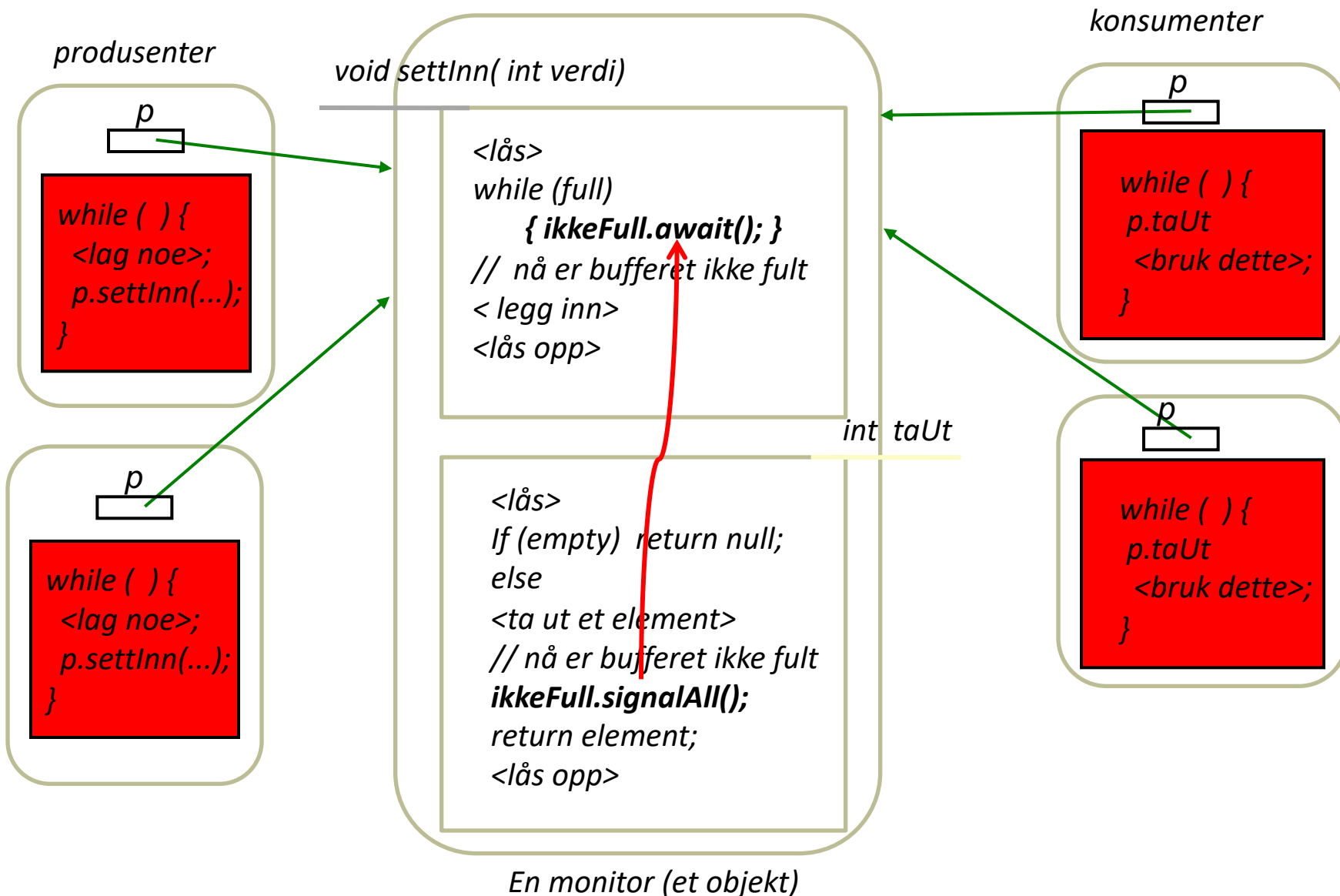
```
settInn(x) {  
    if (antall == 1000) return ;  
    antall ++;  
    tabell[antall-1] = x;  
}
```

```
taUt () {  
    if (antall == 0) return null;  
    antall --;  
    return (tabell[antall]);  
}
```

Overbevis deg
(og andre) om at
metodene bevarer
Invarianten !!!!!!!
(og at den er sann
ved oppstart) !!!!!



Tråder: Produsenter og konsumenter





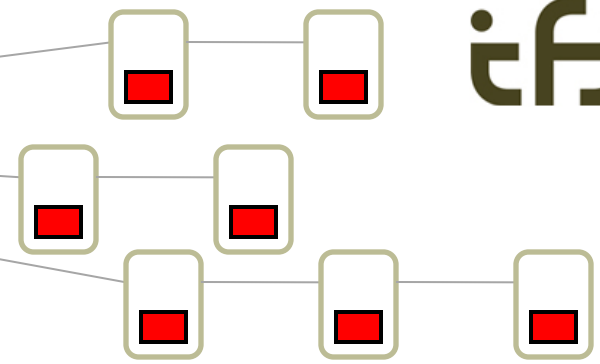
```
Lock laas = new ReentrantLock();  
Condition ikkeFull = laas.newCondition();  
Condition ikkeTom = laas.newCondition();
```

```
void settInn ( int verdi) throws InterruptedException
```

```
laas.lock();  
try {  
    while (full) ikkeFull.await();  
    // nå er det helst sikkert ikke fullt  
    :  
    // det er lagt inn noe, så det er helt sikkert ikke tomt:  
    ikkeTom.signalAll();  
} finally {  
    laas.unlock(),  
}
```

```
int taUt ( ) throws InterruptedException
```

```
laas.lock();  
try {  
    while (tom) ikkeTom.await();  
    // nå er det helst sikkert ikke tomt;  
    :  
    // det er det tatt ut noe, så det er helt sikkert ikke fullt:  
    ikkeFull.signalAll();  
} finally {  
    laas.unlock();  
}
```



Ofte har vi flere grunner til å vente i en monitor

Barrierer i Java

```
import java.util.concurrent.*;
```

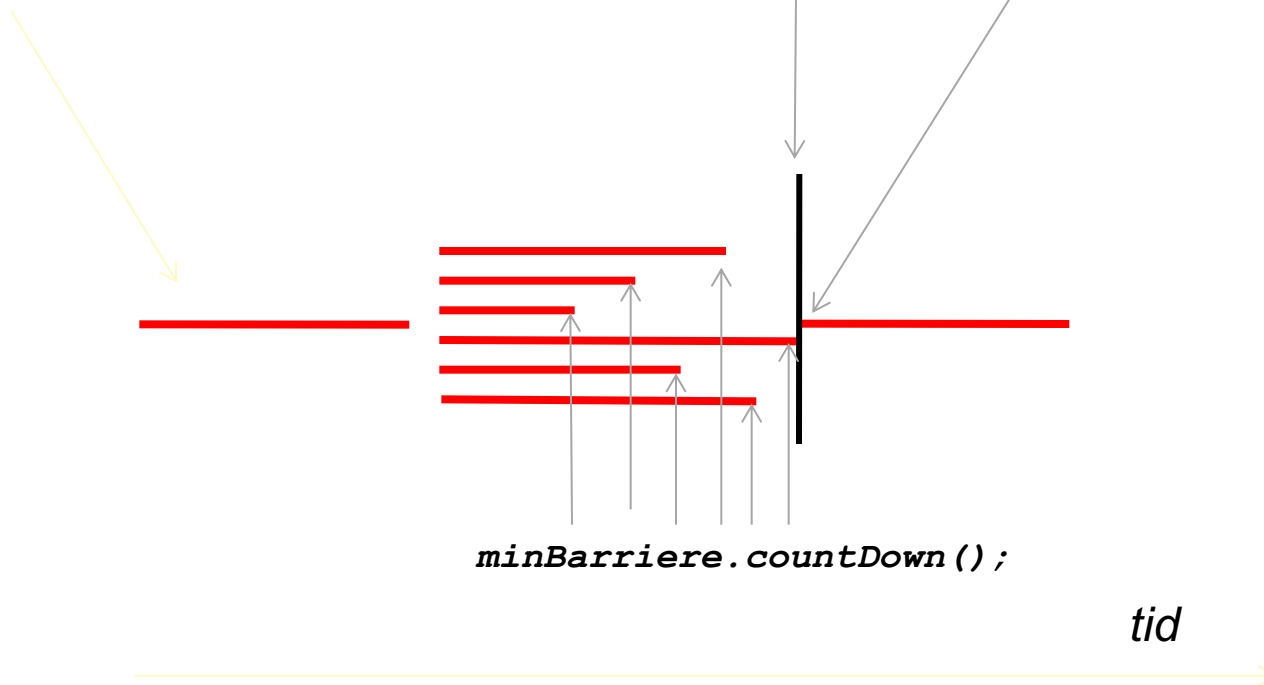
```
CountDownLatch minBarriere =  
    new CountDownLatch(6)
```

Barrieren

```
minBarriere.await();
```

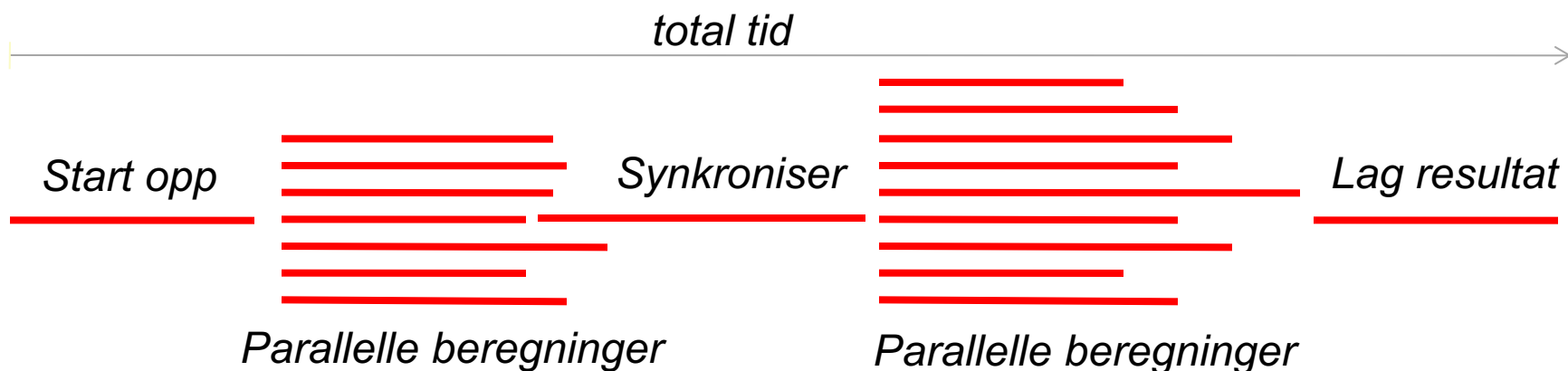
```
minBarriere.countDown();
```

tid



Amdahls lov

- En beregning delt opp i parallell går fortere jo mer uavhengig delene er
- **Amdahls lov:**
 - Totaltiden er
 - tiden i parallell +
 - tiden det tar å kommunisere / synkronisere/ gjøre felles oppgaver
 - Tiden det tar å synkronisere er ikke parallelliserbar (hjelper ikke med flere prosessorer)
 - Men du kan være smart og lage synkroniseringen så kort eller mellom så få tråder som mulig



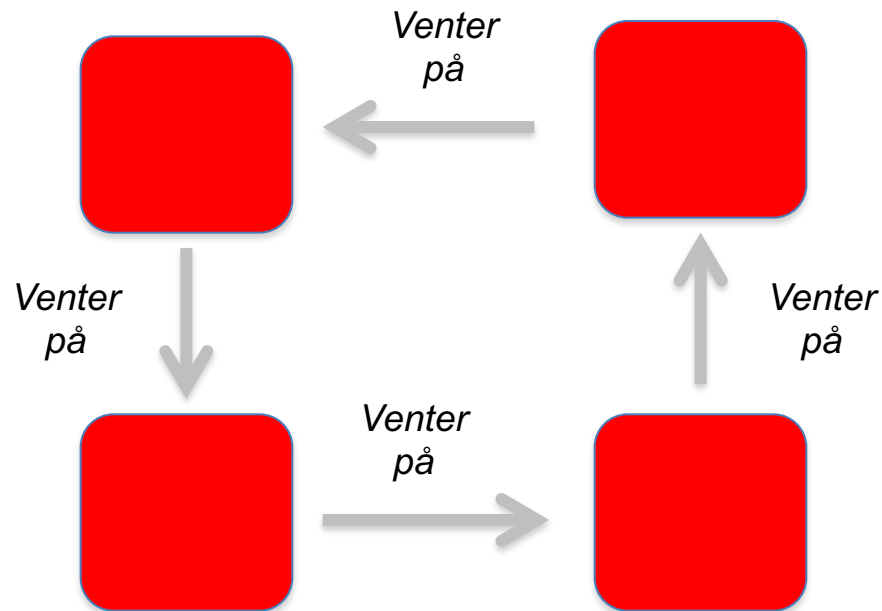
Vranglås (deadlock)

- Vranglås skjer når flere tråder venter på hverandre i en sykel:
- Eksempel
 - Veikryss:
alle bilene skal stoppe for biler fra høyre ->
Alle stopper = VRANGLÅS



Vranglås betyr

- Flere tråder venter på hverandre
- Syklisk venting

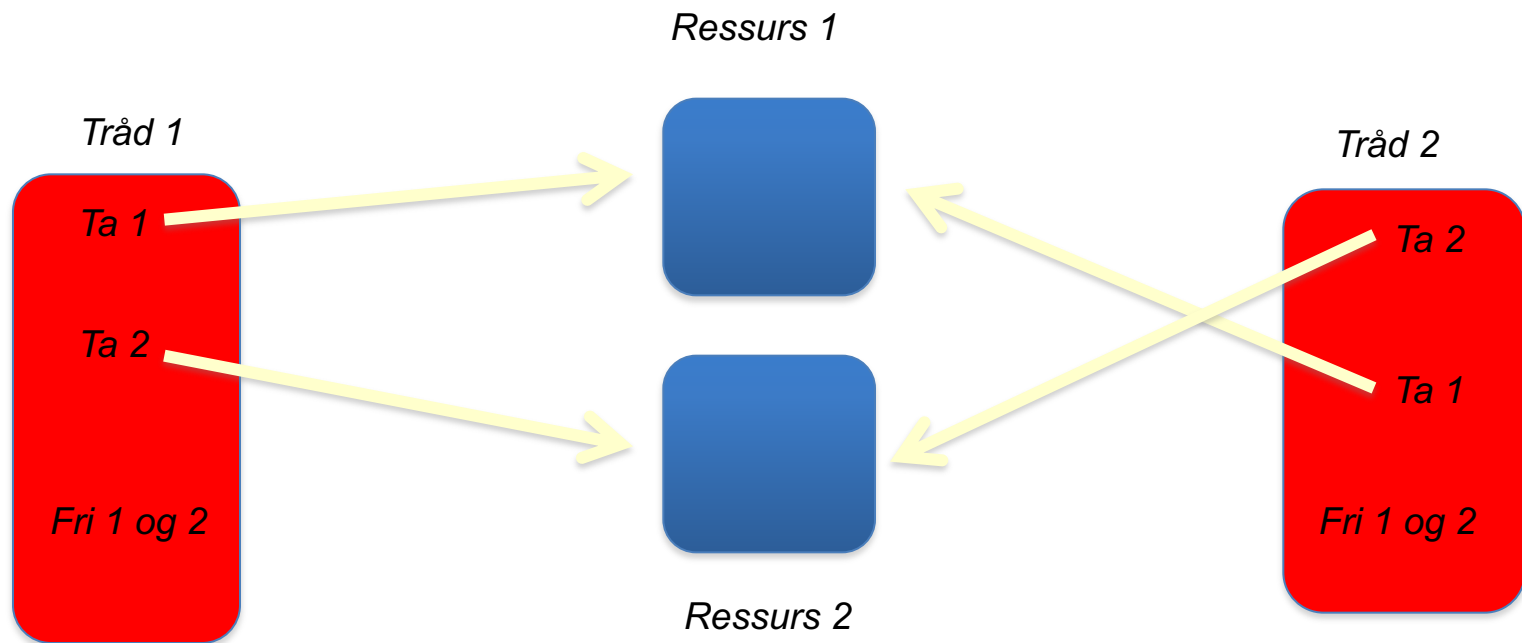


Unngå vranglås

1. Ta bare en ressurs
 2. Ta alle eller ingen
 3. Alle tråder tar alle ressurser i samme rekkefølge
- Hvis vranglås har oppstått:
 - Fri en og en ressurs til det ikke lenger er vranglås

Enkleste eksempel på vranglås

- To tråder
- To ressurser som tas i forskjellig rekkefølge



Enkleste eksempel på vranglås: 2

*To biler kan risikere å
vente på hverandre*



*Smalt veistykke, to biler
kan ikke passere hverandre.
Bilene kan bare se den første delen.*

Program

Kjør bilen

Kjør bilen

Kjør bilen

Kjør bilen

Kjør bilen

Kjør bilen

Kjør bilen

Kjør bilen

Kjør bilen

Kjør bilen

Kjør bilen

Kjør bilen

Kjør bilen

Kjør bilen

Kjør bilen

Kjør bilen

Kjør bilen

Kjør bilen

