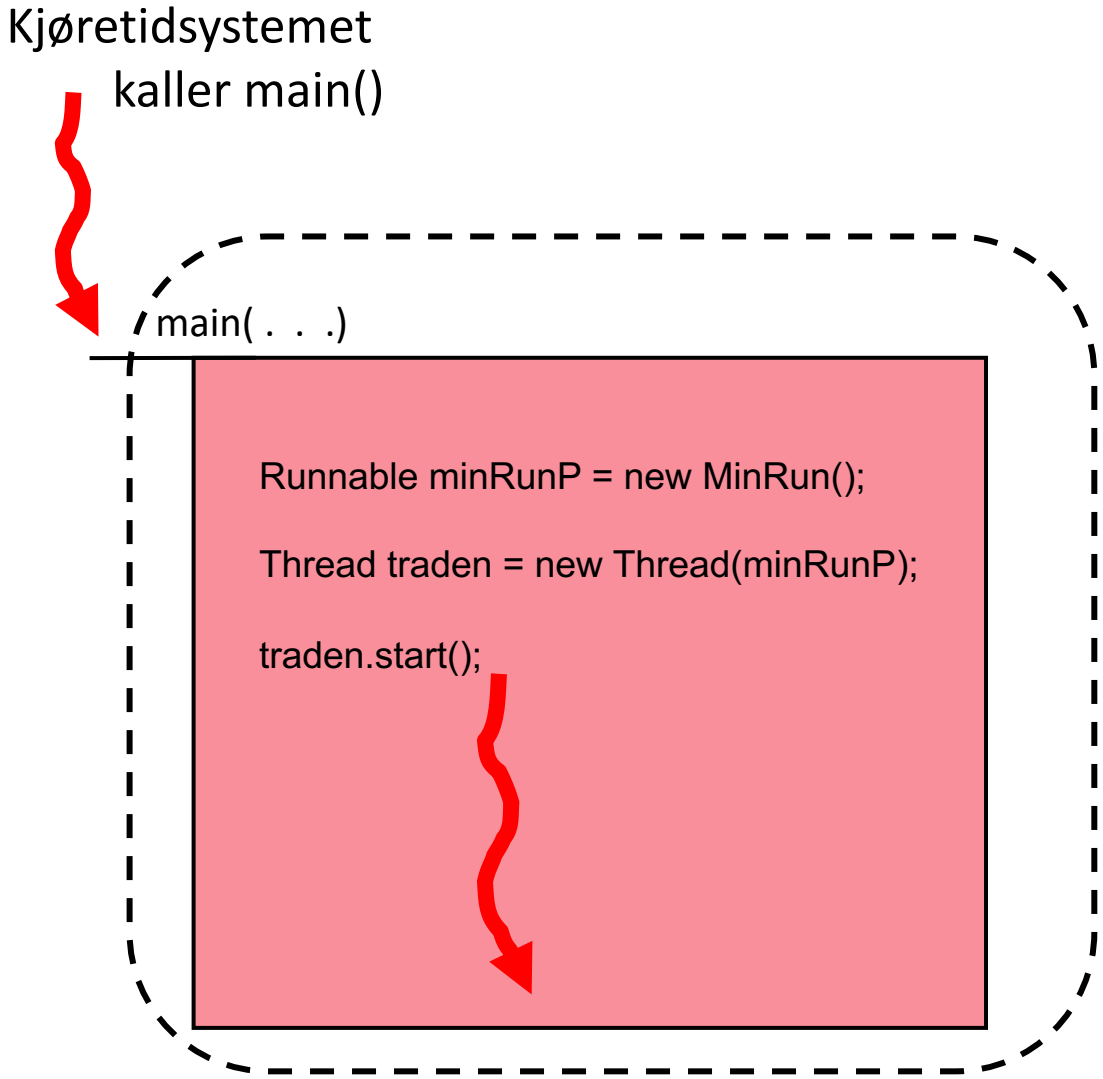
UNIVERSITETET I OSLO

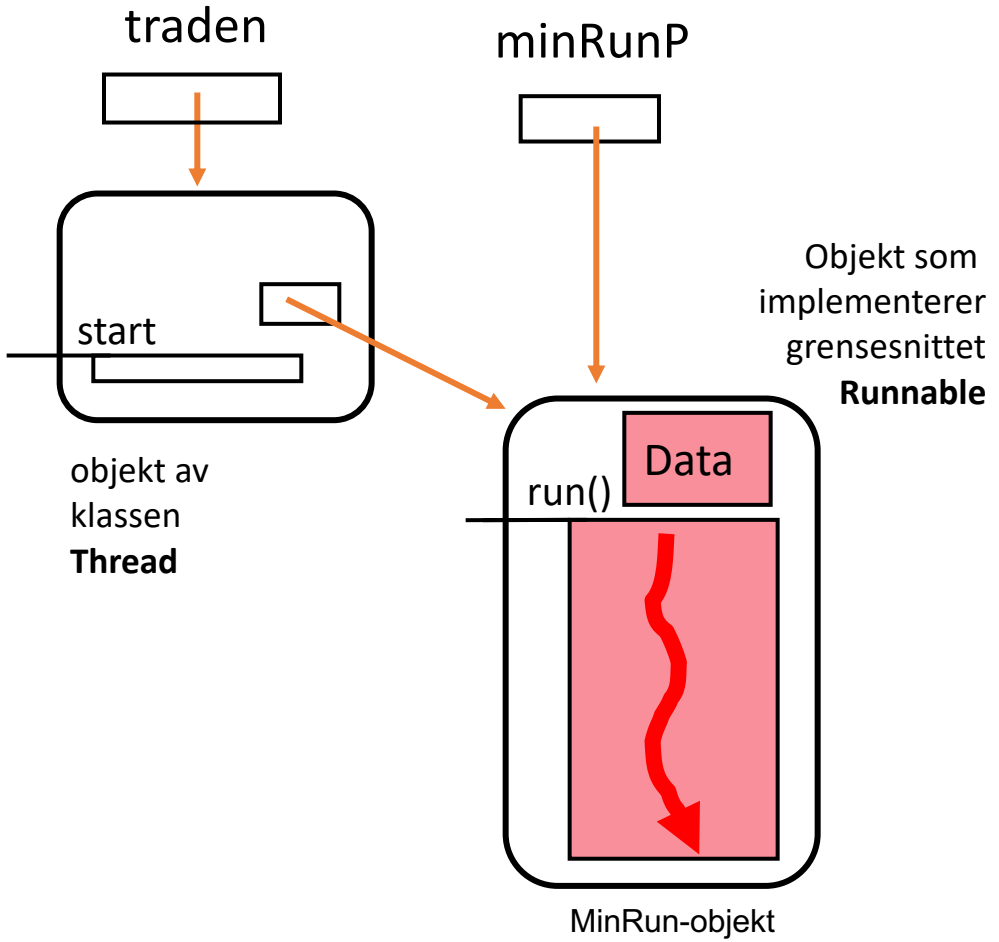
Institutt for informatikk

IN1010
Tråder II
29. mars 2022

Stein Gjessing

Forrige uke lærte vi:





Dere vet også: Tråder kommuniserer via felles data

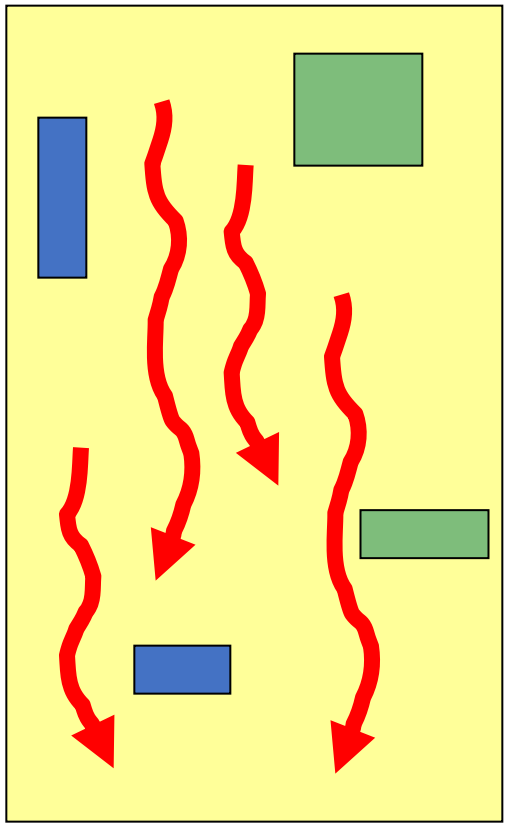
Felles data (grønne felt) må vanligvis bare aksesseres (lese og skrives i) av en tråd om gangen. Hvis ikke blir det kluss i dataene.

På figuren er det to (grønne) områder vi har problemer med (dvs. at to eller flere tråder kan risikere å manipulere data i disse områdene samtidig). Kode som aksesserer slike felles data må skrives inne i ***kritiske regioner*.**

De andre to områdene (blå data) inneholder data som vi enten vet bare kan leses (”immutable” data), eller vi vet at bare én tråd om gangen skriver i disse dataene. Immutable data er bra!

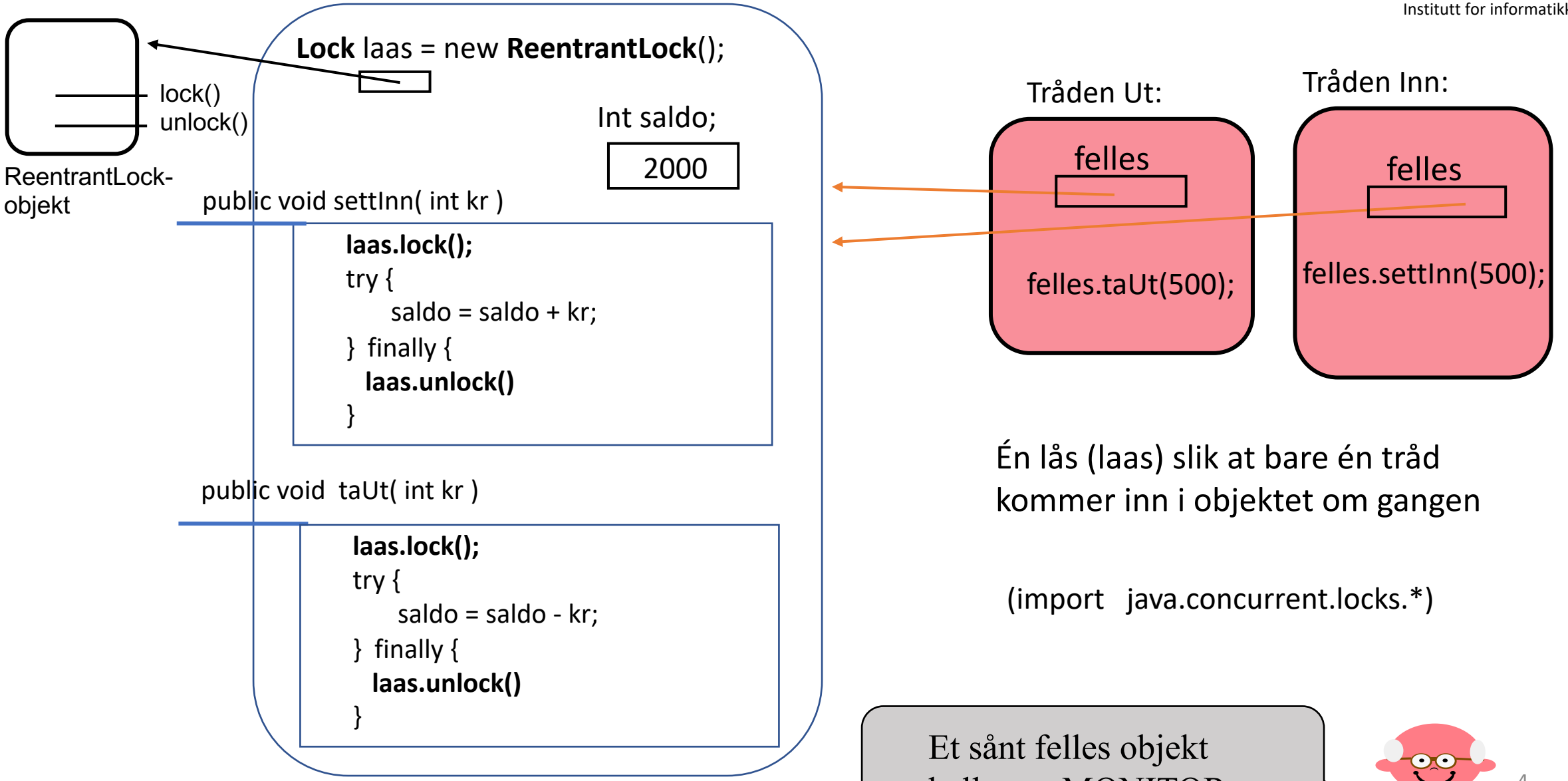
Felles data som skrives i

Felles data som alle bare leser (immutable)



Vårt Java-program med 4 tråder

Også forrige uke: Kritiske regioner



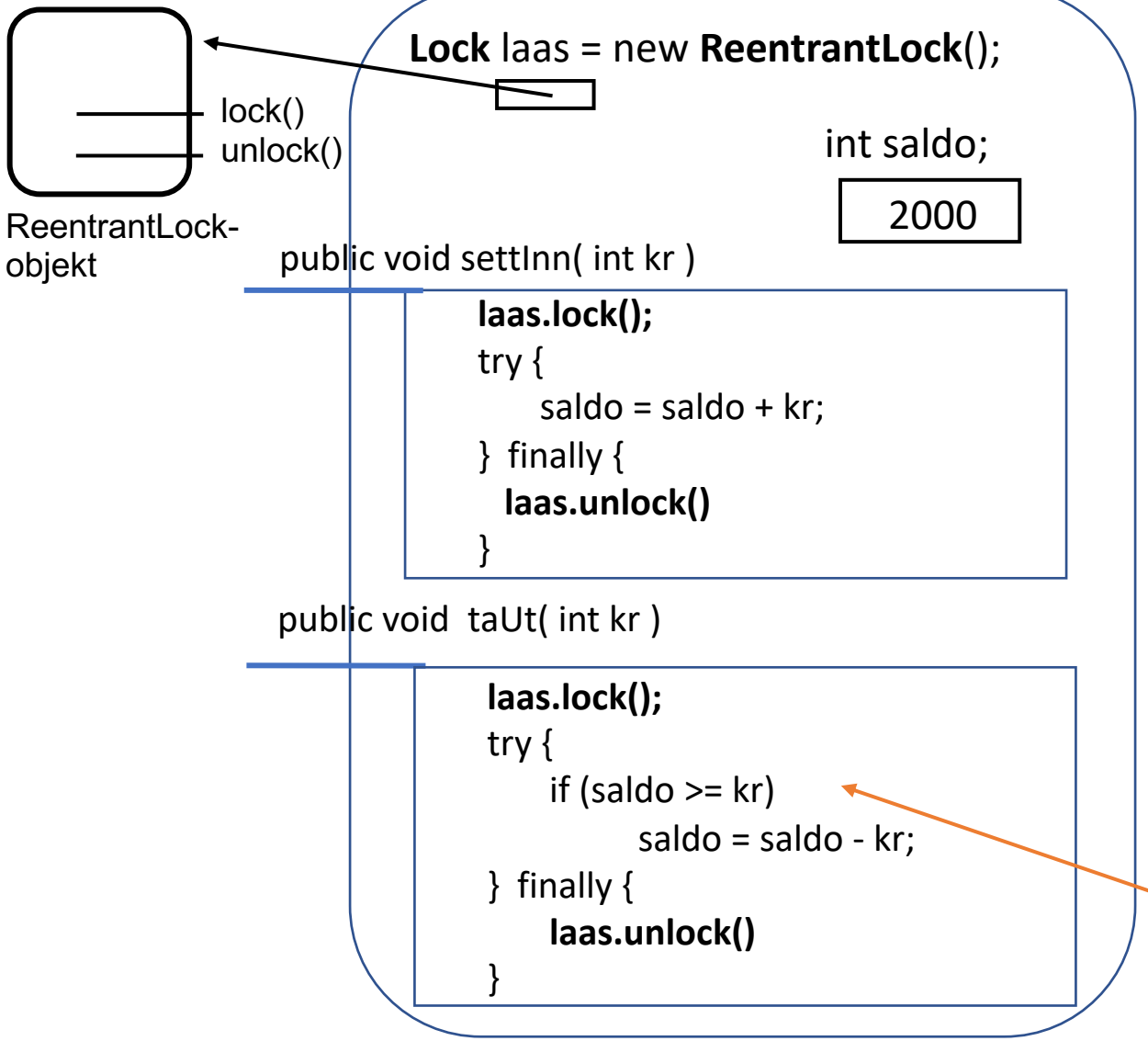
Én lås (laas) slik at bare én tråd kommer inn i objektet om gangen

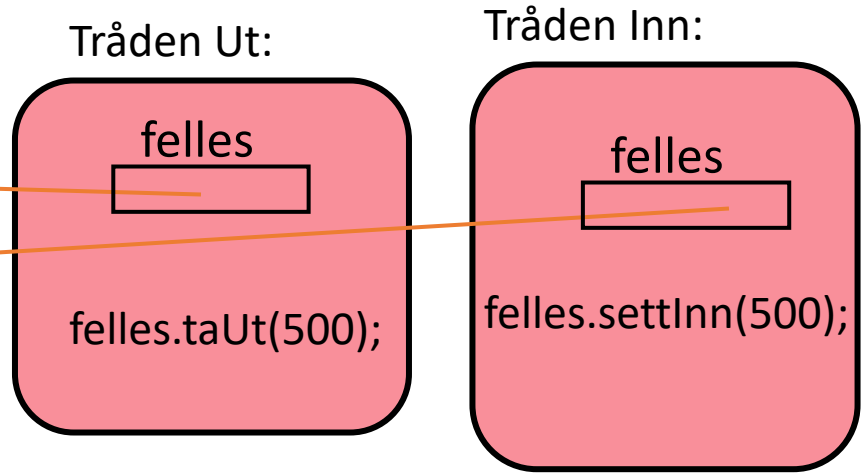
(import java.concurrent.locks.*)

Et sånt felles objekt kalles en MONITOR

Nytt I dag

taUt, men hva om det ikke er penger igjen på konto?





Invariant:
0 <= saldo

If-testen sørger for at invarianten ikke brytes

Hva hvis det ikke er penger på konto

- Kanskje en annen kan sett inn penger
- I så fall: Kanskje vi kan ønsker å vente på at en annen setter inn penger

- Passiv eller aktiv venting (mer neste side)
  - Passsiv: Sett deg ned og vent til du vet det er penger på konto
  - Aktiv: Gå å sjekk med jevne mellomrom om det er penger på konto

Passiv venting er bedre enn aktiv venting

- Aktiv venting er vanligvis ikke smart
- Ikke rettferdig
- Det bruker ressurser unødvendig
  - CPU-kraft
  - Monitoren blokeres hver eneste gang tråden går inn i monitoren for å teste
- Bruk aktiv venting bare i spesielle tilfeller:
  - Når få som kunkurrerer og korte kritiske regioner
  - Når trådene ikke ødelegger (blokkerer) for andre tråder og systemet ikke kan bruke prosessorkapasiteten til noe annet

Dette er mest aktuelt når vi har (jfr. programmeringsmønstre) :

Produsenter og konsumenter

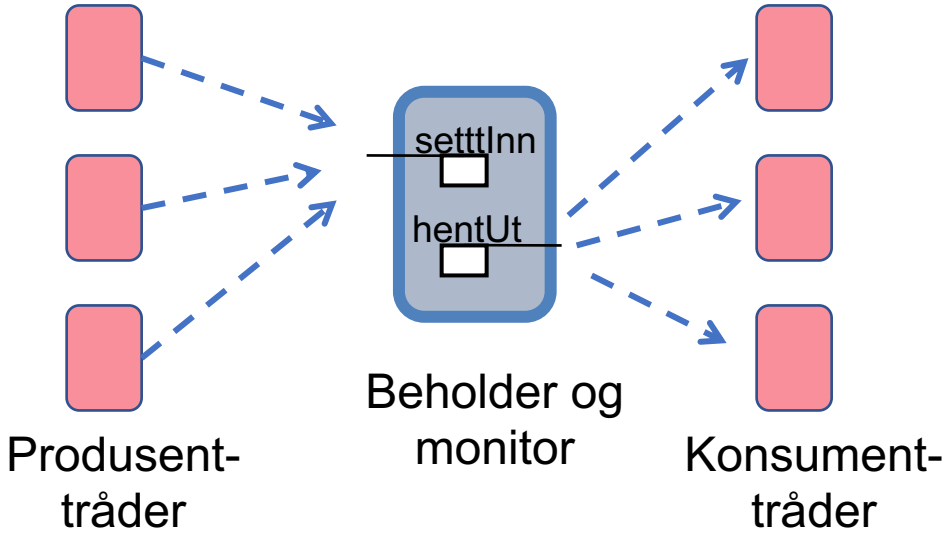
- Mange tråder samarbeider om å løse et problem

- Ofte har vi noen tråder som ***produserer*** noe
  - Tjener penger og setter dem inn på konto
  - Lager data/objekter og legger dem inn i en beholder
  - Lager nye bilder
  - . . . .

- Mens andre tråder ***konsumerer*** dette
  - Tar penger ut av en konto
  - Tar data/objekter ut av en beholder
  - Analyserer (deler av) bilder
  - . . . .

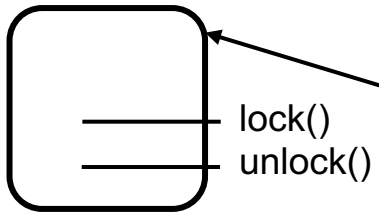
Produsenter og konsumenter - mønster

- Trådene som *produserer* noe (data) og legger disse i en beholder som også er en monitor
- Mens andre tråder *konsumerer* dette ved å hente data ut fra beholderen



En monitor som er/beskytter en beholder

UNIVERSITETET I OSLO
Institutt for informatikk

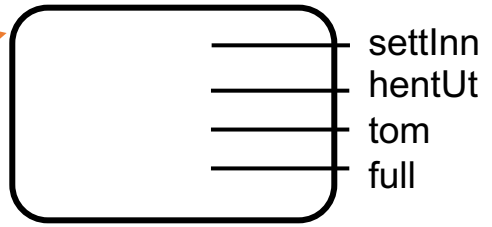


ReentrantLock-objekt

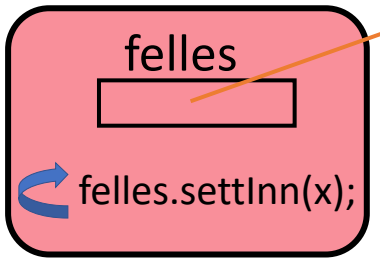
Lock laas = new **ReentrantLock**();

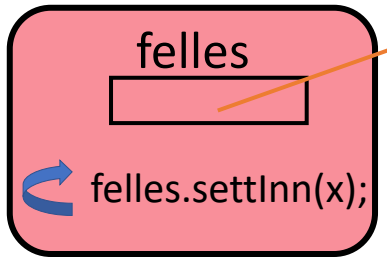
Beholder behold;

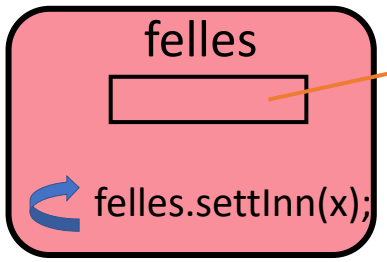
Objekt av klassen Beholder



(Beholderen kan også skrives direkte inne i monitoren)







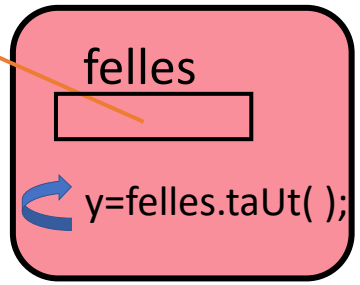
Produsent-tråder

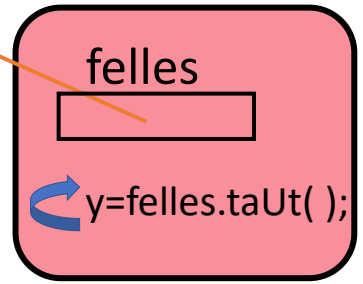
public void settInn (Ting tingen)

```
laas.lock();
try {
      behold.settInn(tingen);
}  finally {
   laas.unlock();
}
```

public Ting taUt ( )

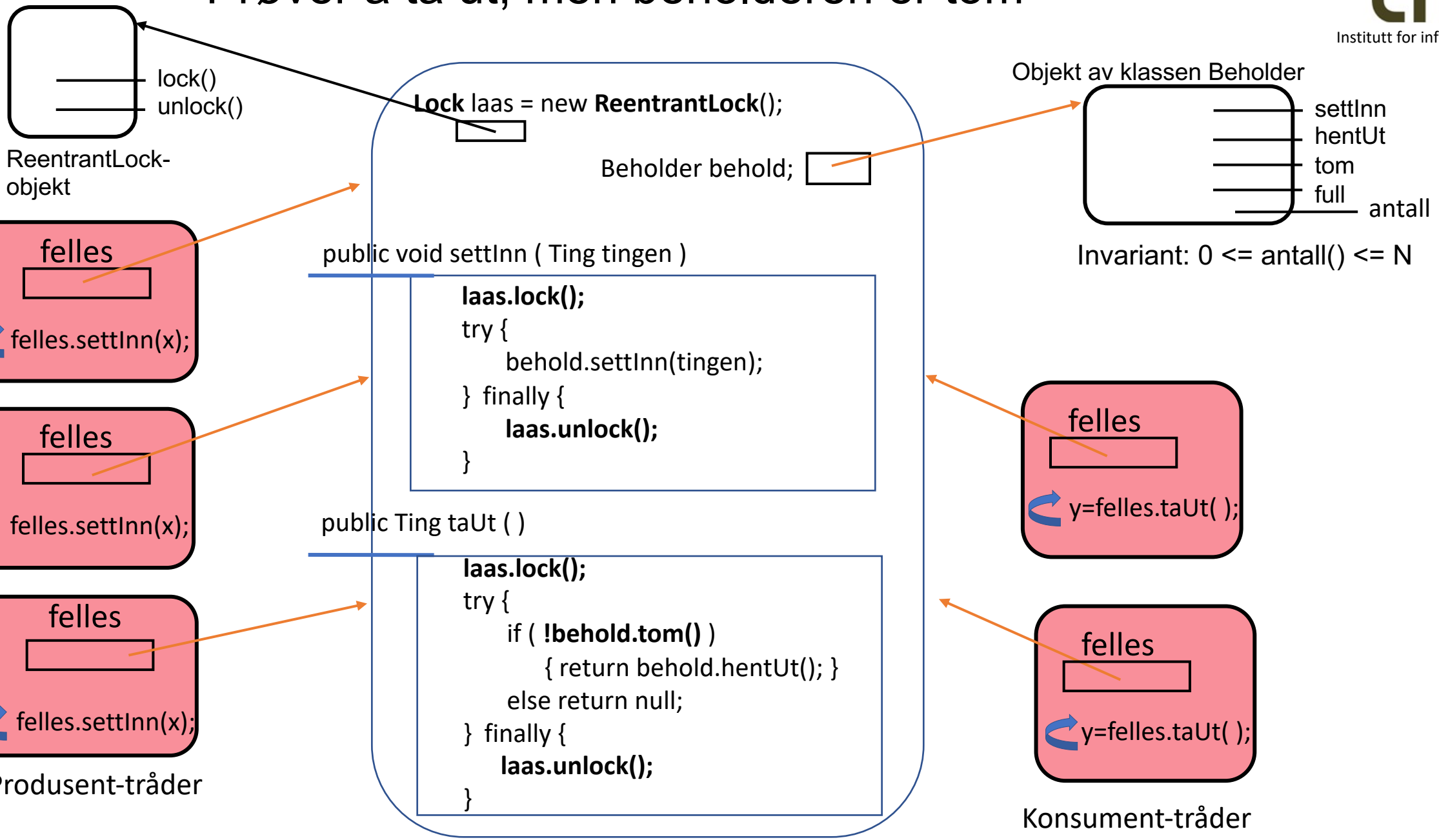
```
laas.lock();
try {
      return behold.hentUt();
}  finally {
   laas.unlock();
}
```





Konsument-tråder

Prøver å ta ut, men beholderen er tom

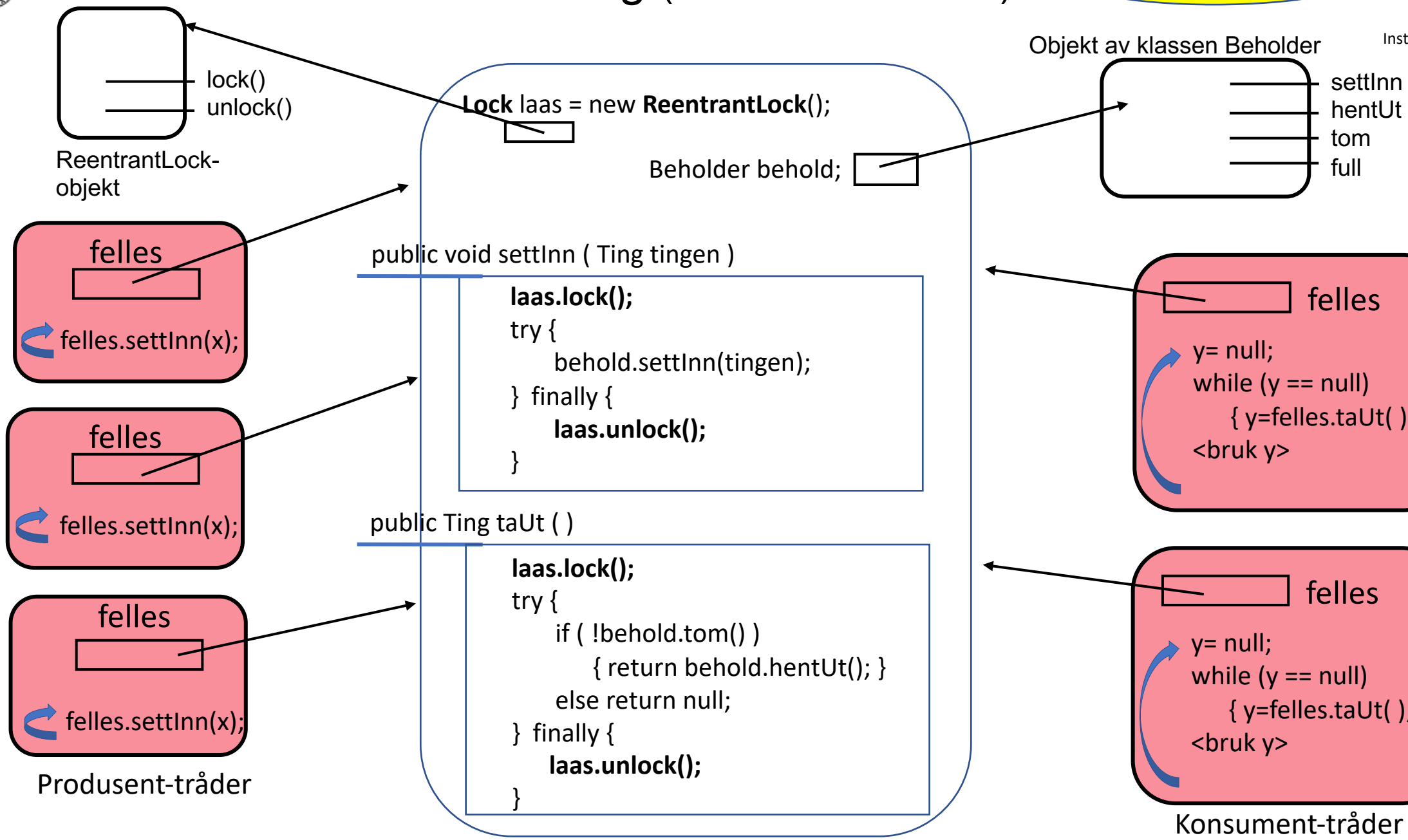


Hva hvis beholder er tom bare midlertidig

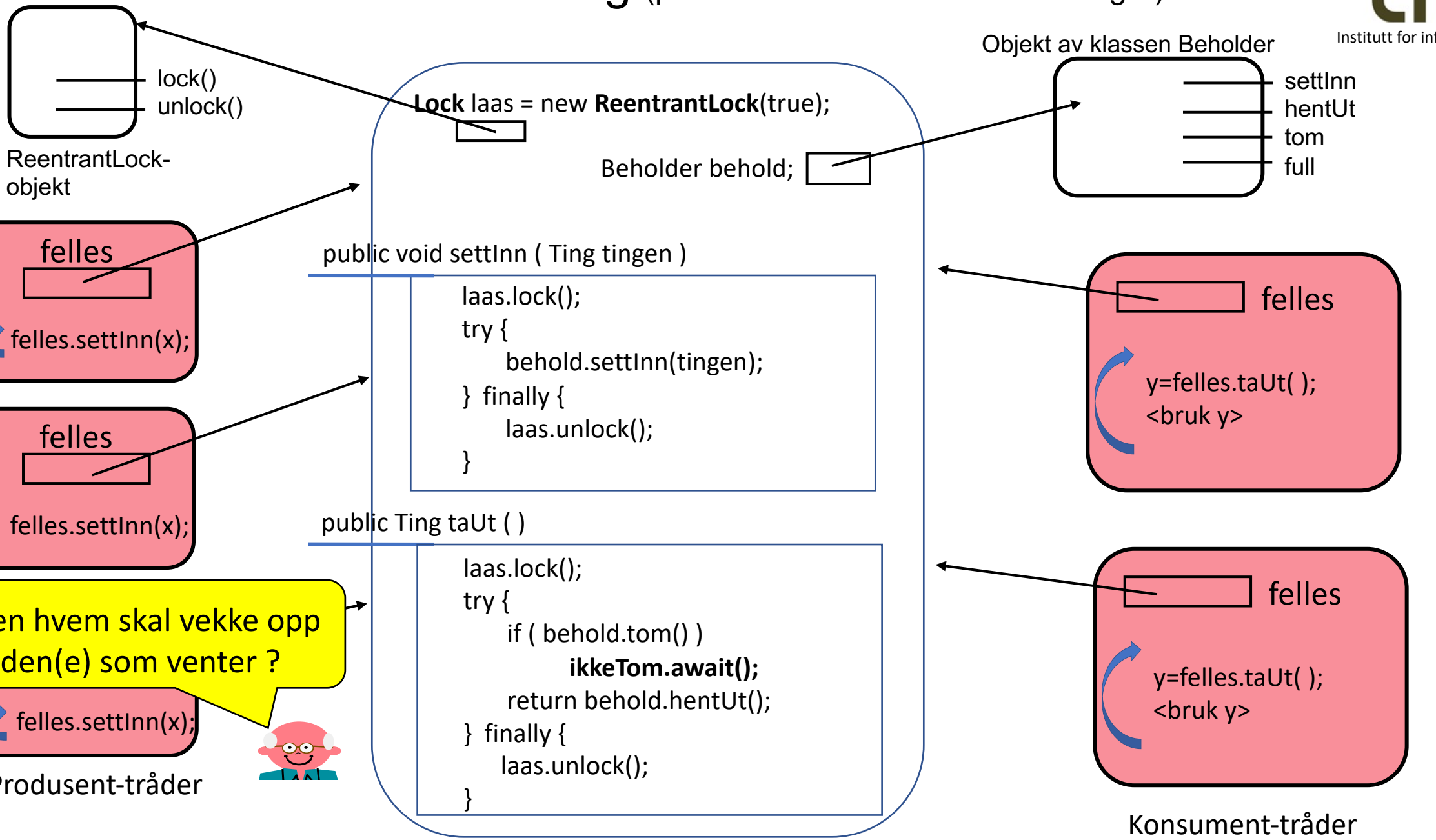
- Kanskje en produsent snart setter inn et nytt objekt
- I så fall: Kanskje vi kan ønsker å **vente** på dette

Aktiv venting (av konsumentene)

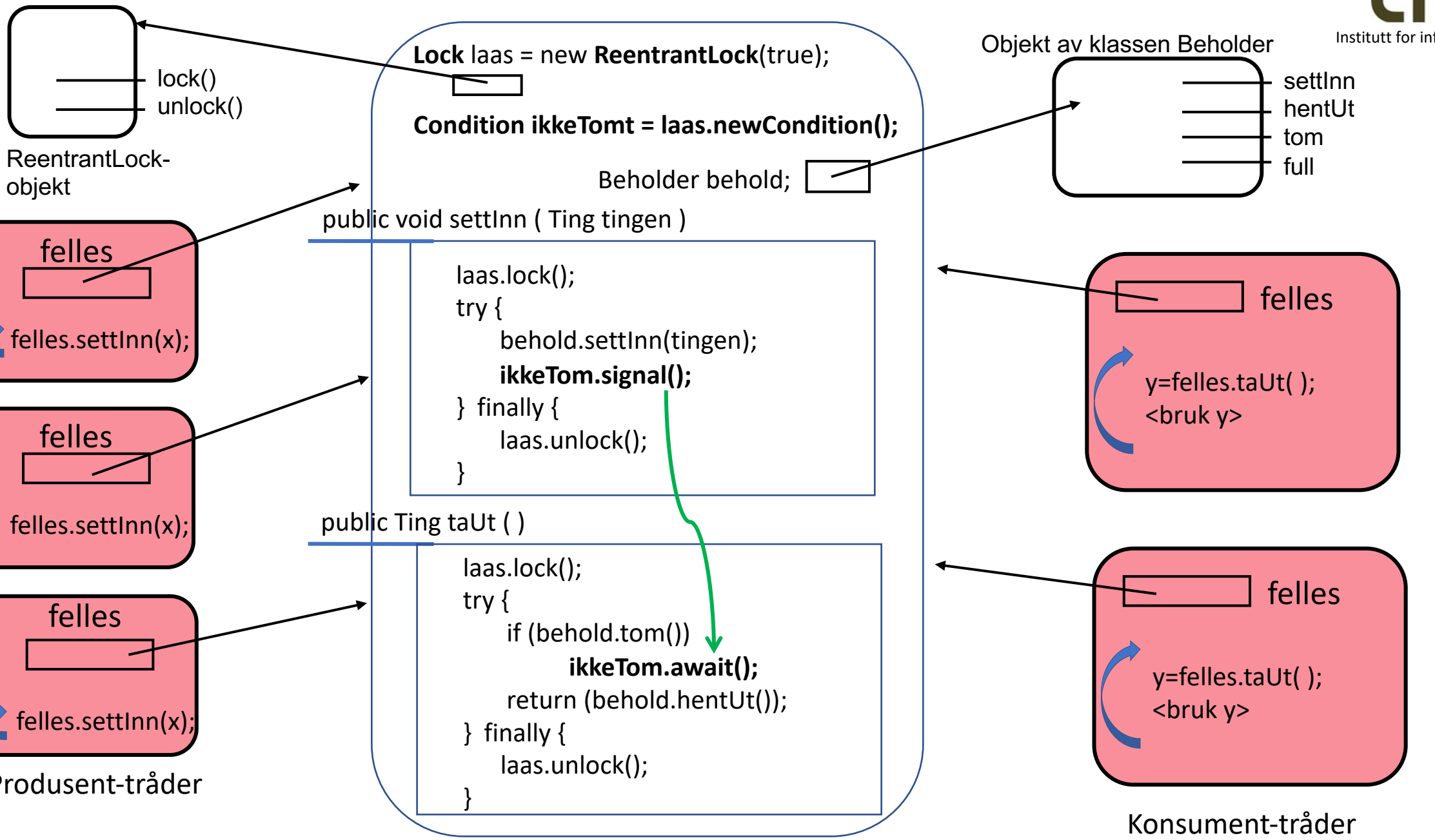
Ikke så bra



Passive venting (på at beholderen **ikke er tom** lenger)



Passive venting og signalering



Passiv venting, **køer bak kulissene**

Beholder behold;

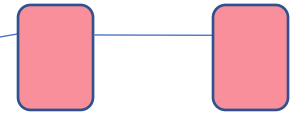
```
Lock laas = new ReentrantLock(true);
Condition ikkeTomt = laas.newCondition();
```

void settInn(Ting tingen) throws InterrupedException

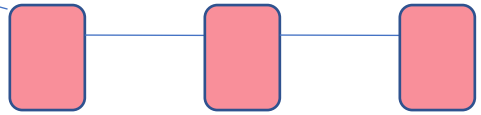
```
laas.lock();
try {
    behold.settInn(tingen);
    // det er lagt inn noe, så det er ikke tomt:
    ikkeTomt.signal();
} finally {
    laas.unlock();
}
```

Tingen taUt( ) throws InterrupedException

```
laas.lock();
try {
    if (behold.tom() )
        { ikkeTomt.await(); }
    return (behold.hentUt());
} finally {
    laas.unlock();
}
```



Tråder som venter på låsen for å få komme inn i monitoren



Tråder som venter på at bufferet ikke skal være tomt

Det er her de ligger og venter

Regler for robust programmering

Beholder behold;

```
Lock laas = new ReentrantLock(true);
Condition ikkeTomt = laas.newCondition();
```

void settInn (Ting tingen) throws InterrupedException

```
laas.lock();
try {
    behold.settInn(tingen);
    // det er lagt inn noe, så det er helt sikkert ikke tomt:
    ikkeTomt.signalAll();
} finally {
    laas.unlock();
}
```

Tingen taUt ( ) throws InterrupedException

```
laas.lock();
try {
    while ( behold.tom() )
            ikkeTomt.await();
    // nå er beholderen helt sikkert ikke tom;
    return behold.hentUt();
} finally {
    laas.unlock();
}
```

Her har vi skiftet ut if-testen i taUt() med en while-test.

Og vi har skiftet ut signal() med signalAll().

Dette gjør programmet mer robust.

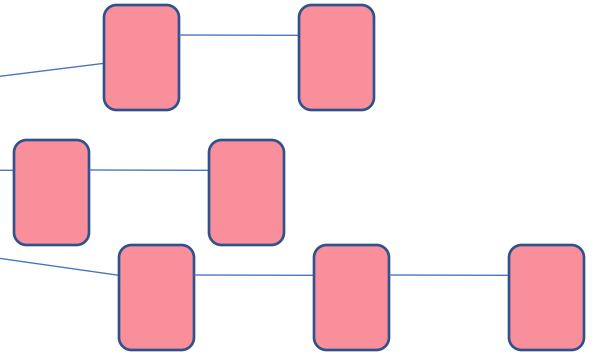
En metode signalerer når den TROR at invarianten ikke blir brutt, men det er den ventende metodens ansvar å sjekke (på nytt).

Se notatet om tråder.

(Signaleringen er bare et hint, og hint er bra i informatikk)

Beholder behold;

```
Lock laas = new ReentrantLock(true);
Condition ikkeFullt =  laas.newCondition();
Condition ikkeTomt = laas.newCondition();
```



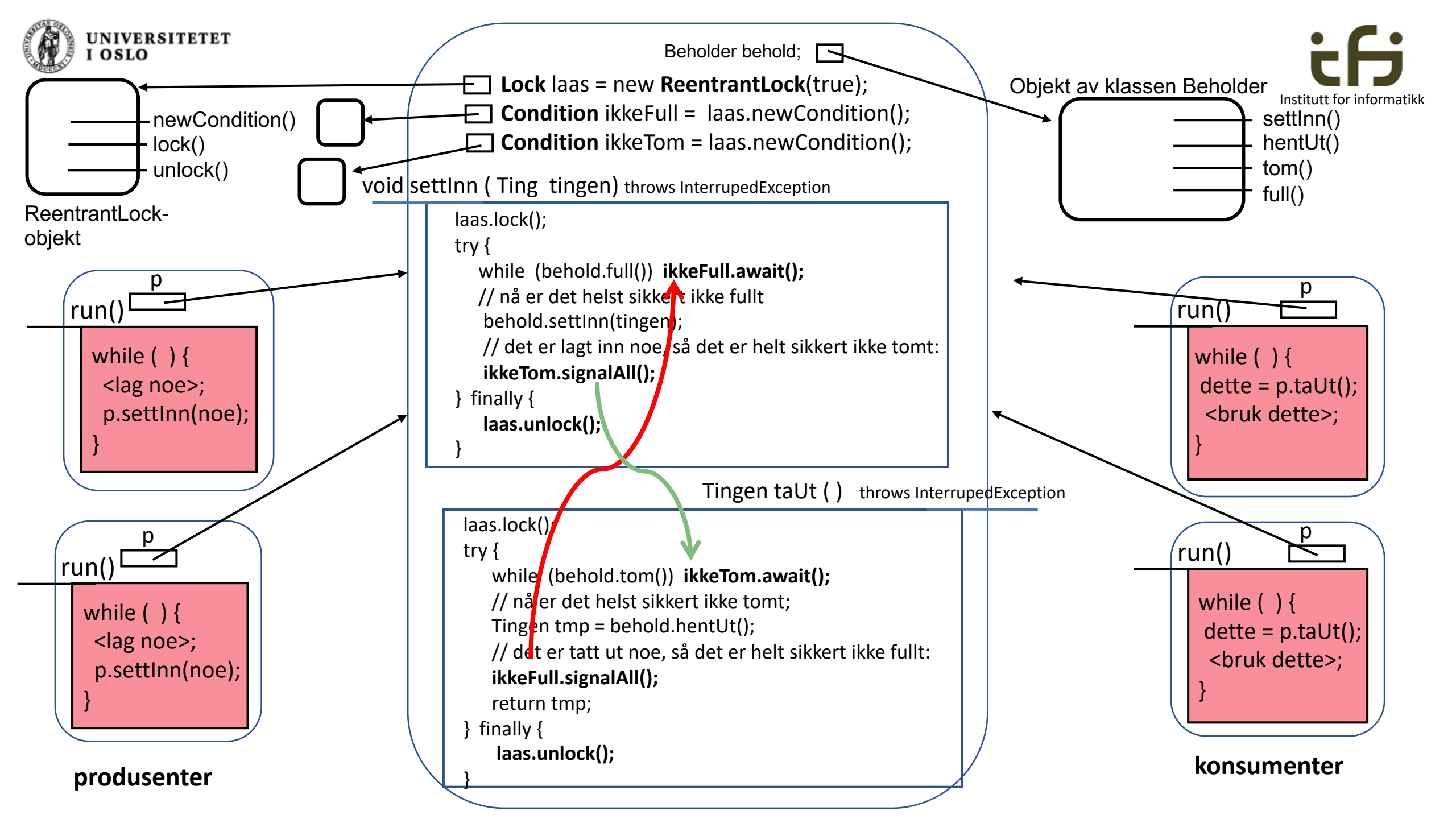
void settInn (Ting tingen) throws InterrupedException

```
laas.lock();
try {
    while  ( behold.full() )  ikkeFullt.await();
    // nå er det helst sikkert ikke fullt
    behold.settInn(tingen);
    // det er lagt inn noe, så det er ikke tomt:
    ikkeTomt.signalAll();
} finally {
    laas.unlock();
}
```

Tingen taUt ( ) throws InterrupedException

```
laas.lock();
try {
    while  (behold.tom())  ikkeTomt.await();
    // nå er det helst sikkert ikke tomt;
    Tingen tmp = beholder.hentUt();
    // det er det tatt ut noe, så det er ikke fullt:
    ikkeFullt.signalAll();
    return tmp;
} finally {
    laas.unlock();
}
```

En kø for selve låsen og en kø for hver betingelse (for hver condition)


UNIVERSITETET I OSLO
Institutt for informatikk
Beholder behold;
Lock laas = new ReentrantLock(true);
Condition ikkeFull = laas.newCondition();
Condition ikkeTom = laas.newCondition();
newCondition()
lock()
unlock()
ReentrantLock-objekt
Objekt av klassen Beholder
settInn()
hentUt()
tom()
full()
void settInn (Ting tingen) throws InterrupedException
laas.lock();
try {
while (behold.full()) ikkeFull.await();
// nå er det helst sikkert ikke fullt
behold.settInn(tingen);
// det er lagt inn noe, så det er helt sikkert ikke tomt:
ikkeTom.signalAll();
} finally {
laas.unlock();
}
Tingen taUt ( ) throws InterrupedException
laas.lock();
try {
while (behold.tom()) ikkeTom.await();
// nå er det helst sikkert ikke tomt;
Tingen tmp = behold.hentUt();
// det er tatt ut noe, så det er helt sikkert ikke fullt:
ikkeFull.signalAll();
return tmp;
} finally {
laas.unlock();
}
p
run()
while ( ) {
<lag noe>;
p.settInn(noe);
}
produsenter
p
run()
while ( ) {
dette = p.taUt();
<bruk dette>;
}
konsumenter

Java-biblioteket java.concurrent

Interface Lock:

import java.concurrent.locks.*

- void **lock**() Acquires the lock
- void **unlock**() Releases the lock.
- Condition **newCondition**() Returns a new **Condition** instance that is bound to this Lock instance.

Interface Condition:

- void **await**() Causes the current thread to wait on this Condition until it is signalled (or interrupted)
- void **signal**() Wakes up one waiting thread (on this Condition).
- void **signalAll**() Wakes up all the waiting threads (on this Condition)

- class ReentrantLock implements Lock

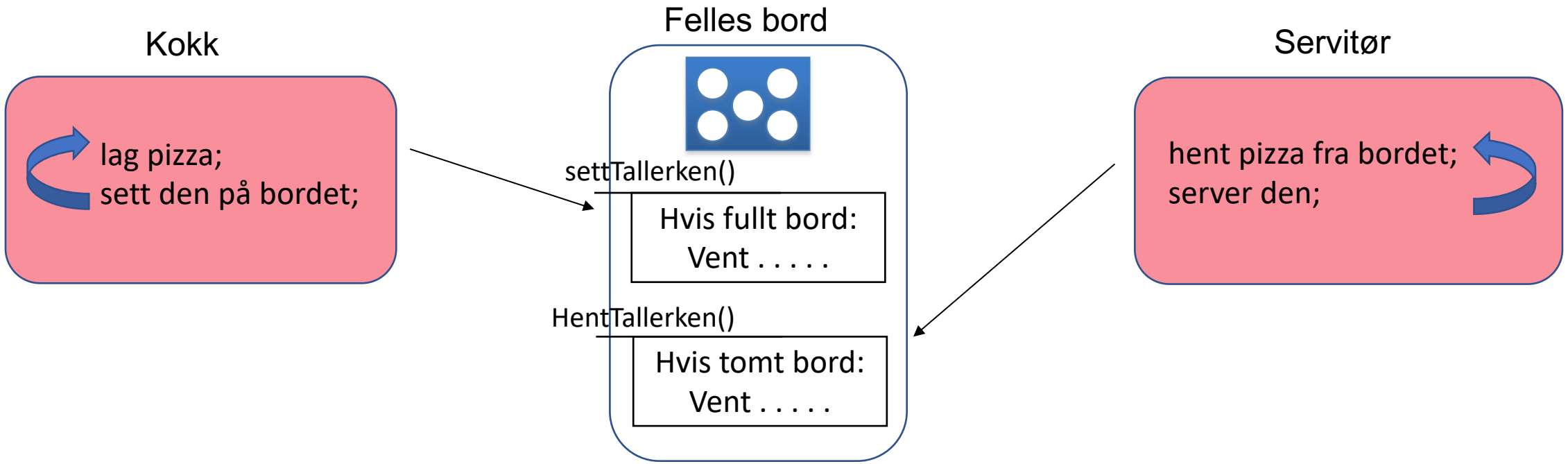
Et større litt større eksempel – kokk og servitør

- Kokken lager mat og setter en og en tallerken på et bord
- Servitøren tar en og en tallerken fra bordet og serverer
- Kokken må ikke sette mer enn BORD_KAPASITET tallerkener på bordet (maten blir kald)
- Servitøren kan selvsagt ikke servere mat som ikke er laget (bordet er tomt)

- Her: en kokk og en servitør – Oppgave: lag flere av hver.

Kokk og servitør

- Kokken må vente når det allerede er fem tallerkener på bodet
- Servitøren må vente når det ikke er laget noe mat (ingen tallerkener på bordet)
- Kokken må starte opp servitøren igjen når han har satt tallerken nr. 1 på bordet (eller alltid når han har satt en tallerken på bordet ?)
- Servitøren må starte opp kokken igjen når han tar tallerken nr. 5 fra bordet (eller alltid når han tar en tallerken fra bordet ?)



Slik virker programmet:

```
MacBook-Pro:programmer steing$ java RestaurantCM 5 20
Kokken lager tallerken nr: 1
Antall paa bordet: 1
Kelner serverer nr:1
Kokken lager tallerken nr: 2
Antall paa bordet: 1
Kelner serverer nr:2
Kokken lager tallerken nr: 3
Antall paa bordet: 1
Kelner serverer nr:3
Kokken lager tallerken nr: 4
Antall paa bordet: 1
Kelner serverer nr:4
Kokken lager tallerken nr: 5
Antall paa bordet: 1
Kokken lager tallerken nr: 6
Antall paa bordet: 2
Kokken lager tallerken nr: 7
Antall paa bordet: 3
Kelner serverer nr:5
Kokken lager tallerken nr: 8
Antall paa bordet: 3
Kokken lager tallerken nr: 9
Antall paa bordet: 4
Kokken lager tallerken nr: 10
Antall paa bordet: 5
Kelner serverer nr:6
Kokken lager tallerken nr: 11
Antall paa bordet: 5
Kelner serverer nr:7
Kokken lager tallerken nr: 12
```

Tror du at programmet nedenfor er feil ?

Hvis ja: Påvis en kjøring som gir en feilsituasjon

Tror du dette programmet er riktig ?

Hvis ja: Begrunn hvorfor det er riktig ?


```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class RestaurantCM {
    public static void main(String[] args) {
        int antallPlassBord = Integer.parseInt(args[0]);
        int antallRetter  = Integer.parseInt(args[1]);
        FellesBord bord = new FellesBord(antallPlassBord);
        Kokk  kokk = new Kokk(bord,antallRetter);
        new Thread(kokk).start();
        Servitor  servitor = new Servitor(bord,antallRetter);
        new Thread(servitor).start();
    }
}
```

```
class FellesBord {  // En monitor
    private Lock bordlas;
    private Condition ikkeTomtBord;
    private Condition ikkeFulltBord;
    private int antallPaBordet = 0;
    private final int BORD_KAPASITET;
    /* Invariant: 0 <= antallPaBordet <= BORD_KAPASITET */

    public FellesBord (int ant) {
        BORD_KAPASITET = ant;
        bordlas = new ReentrantLock();
        ikkeTomtBord = bordlas.newCondition();
        ikkeFulltBord = bordlas.newCondition();
    }
```

Vent på at bordet ikke er tomt lenger: ikkeTomtBord
Vent på at bordet ikke er fullt lenger: ikkeFulltBord

```
void settTallerken() throws InterruptedException {
    bordlas.lock();
    try {
        while (antallPaBordet >= BORD_KAPASITET) {
            /* sa lenge det er  BORD_KAPASITET tallerkner på bordet
                    må vi vente på at bordet ikke lenger er fullt */
            ikkeFulltBord.await();
        }
        // Na er antallPaBordet < BORD_KAPASITET
        antallPaBordet++;
        // Invarianten holder
        System.out.println("Antall paa bordet: " + antallPaBordet);
        // Si fra til den som tar tallerkener:
        ikkeTomtBord.signal();
    }
    finally {
        bordlas.unlock();
    }
}
```

```
    void hentTallerken() throws InterruptedException  {
        bordlas.lock();
        try {
            while (antallPaBordet == 0) {
               /* Sa lenge det ikke er noen talerkener pa bordet
                     må vi vente på at det ikke lenger er tomt */
               ikkeTomtBord.await();
            }
            // Na er antallPaBordet > 0
            antallPaBordet --;
            // Invarianten holder
            // si fra til den som setter tallerkener pa bordet:
            ikkeFulltBord.signal();
        }
        finally {
            bordlas.unlock();
        }
    }
} // slutt class FellesBord;
```

```
class Kokk implements Runnable {
    private FellesBord bord;
    private final int ANTALL;
    private int laget = 0;
    Kokk(FellesBord bord, int ant) {
        this.bord = bord;
        ANTALL = ant;
    }

    public void run() {
        try {
            while(ANTALL != laget) {
                laget ++;
                System.out.println("Kokken lager tallerken nr: " + laget);
                bord.settTallerken();
                Thread.sleep((long) (500 * Math.random()));
            }
        }
        catch (InterruptedException e) {
            System.out.println("Uventet stopp 1");
        }
    }
}
```

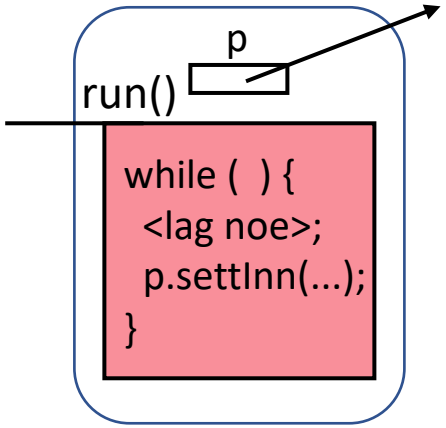
```
class Servitor implements Runnable {
      private FellesBord bord;
      private final int ANTALL;
      private int servert = 0;
      Servitor(FellesBord bord, int ant) {
          this.bord = bord;
          ANTALL = ant;
      }

      public void run() {
          try {
              while (ANTALL != servert)  {
                  bord.hentTallerken(); /* hent tallerken og server */
                  servert++;
                  System.out.println("Kelner serverer nr:" +  servert);
                  Thread.sleep((long) (1000 * Math.random()));
              }
          }
          catch (InterruptedException e) {
              System.out.println("Uventet stopp 2");
          }
      }
}
```

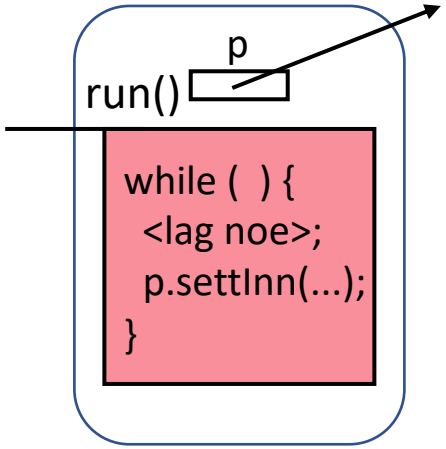
Andre alternativer for tråder og monitorer

Ikke pensum i IN1010

- Lage tråder ved hjelp av **subklasser** istedenfor delegering
  - Horstmann: Programming Tip 20.1
- Lage **monitorer** ved hjelp av **Javas originale mekanismer** istedenfor Doug Leas concurrent-bibliotek.
  - Horstmann: Special Topic 20.2
  - Se class Object (wait(), notify(), notifyAll()) og nøkkelordet synchronized
    - Alle objekter i Java har en iboende lås

Huskeregler for tråder





NB! Trådene/arbeiderene (run()-metodene) kaller bare metodene i monitorene som om monitoren var et vanlig objekt.

Ingen låsing, venting eller signalering i trådene utenom monitorene!

Alle låsing, venting og signalering skjer i inne monitorene

Men en tråd kan si Thread.sleep().
Og en tråd kan kalle interrupt() og join() i andre tråder

Pass også på (mest om monitorer)

- En monitor er et vanlig objekt som flere tråder har referanser til og som disse trådene derfor kan kalle metoder i.
- En monitor er ofte en beholder for utveksling av data mellom tråder.
- Låsing og opplåsing skjer bare i monitor-metoder
- Venting og signalering foregår alltid inne i monitor-metoder
- Ordet "monitor" betyr jo så mye, men når det er snakk om parrallell-programmering og tråder så er det definisjonen gitt her som gjelder
  - Funnet på av Tony Hoare i 1974

Om startverdier i tråder:

- Data overføres gjerne til run-metoden som parametre til konstruktøren (og videre til instansvariable) i den klassen run-metoden er inne i.

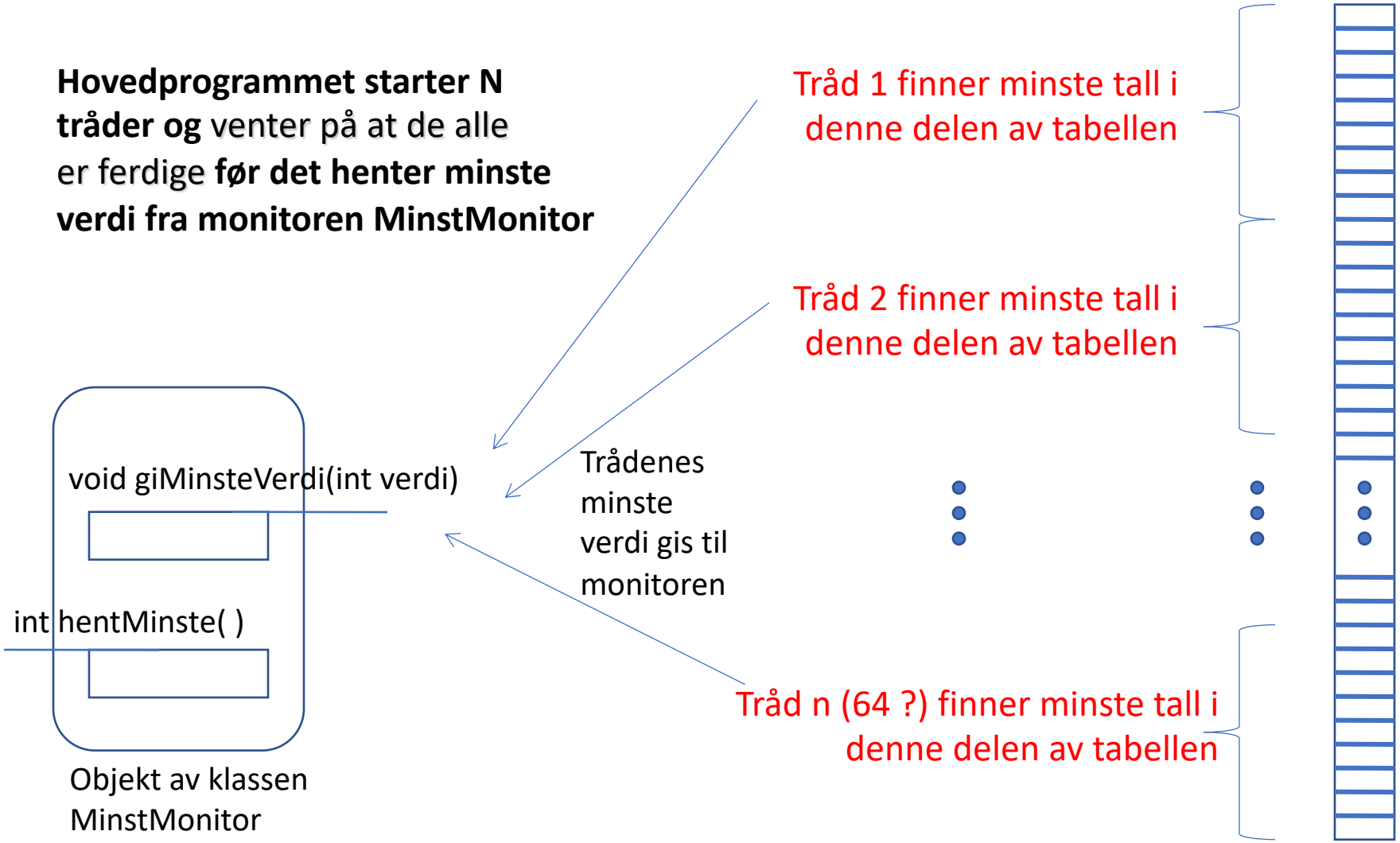
Nytt eksempel

- Finne minst tall i en array

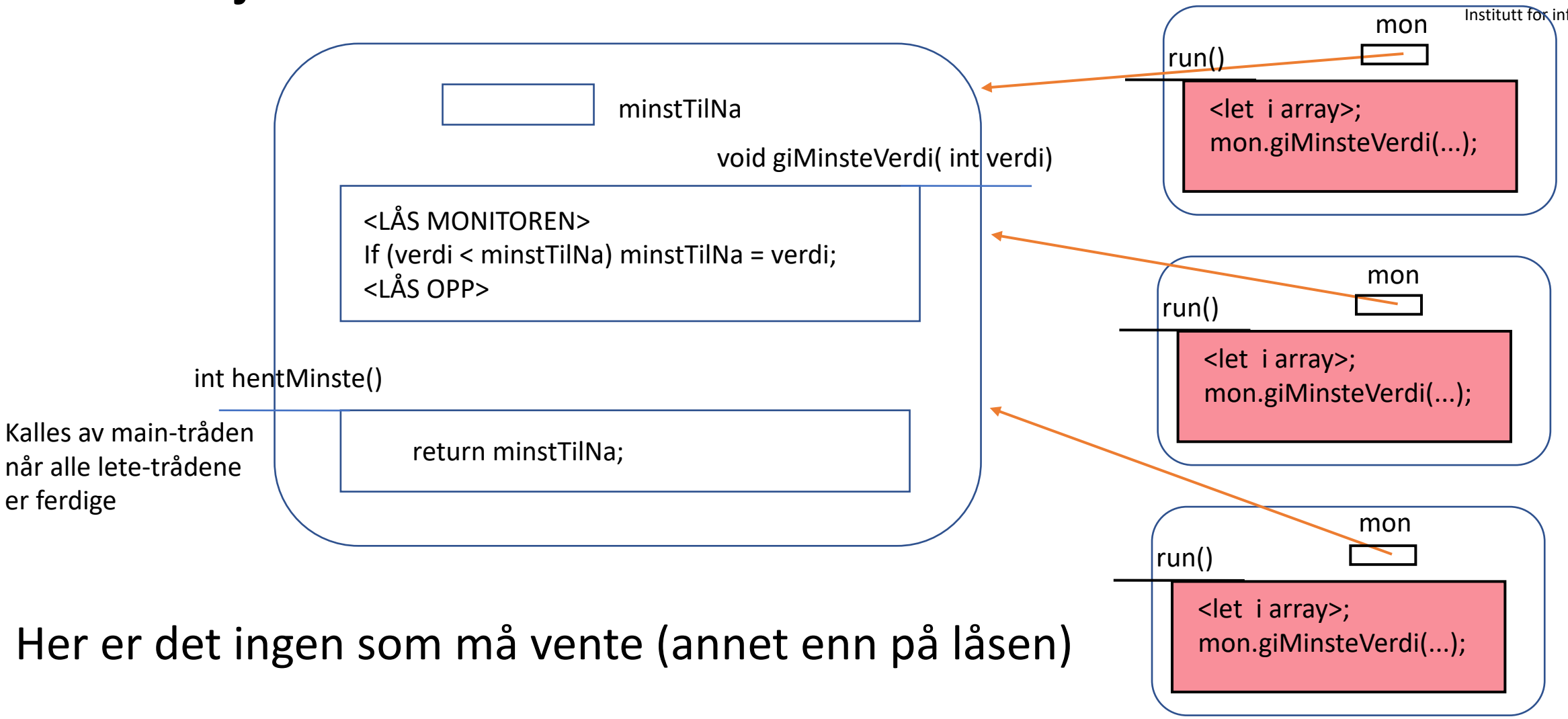
Eksempel på parallellisering: Finn minste tall i en tabell

(samme teknikk for å finne sum / gjennomsnitt)

Hovedprogrammet starter N tråder og venter på at de alle er ferdige **før det henter minste verdi fra monitoren MinstMonitor**



Objekt av klassen MinstMonitor

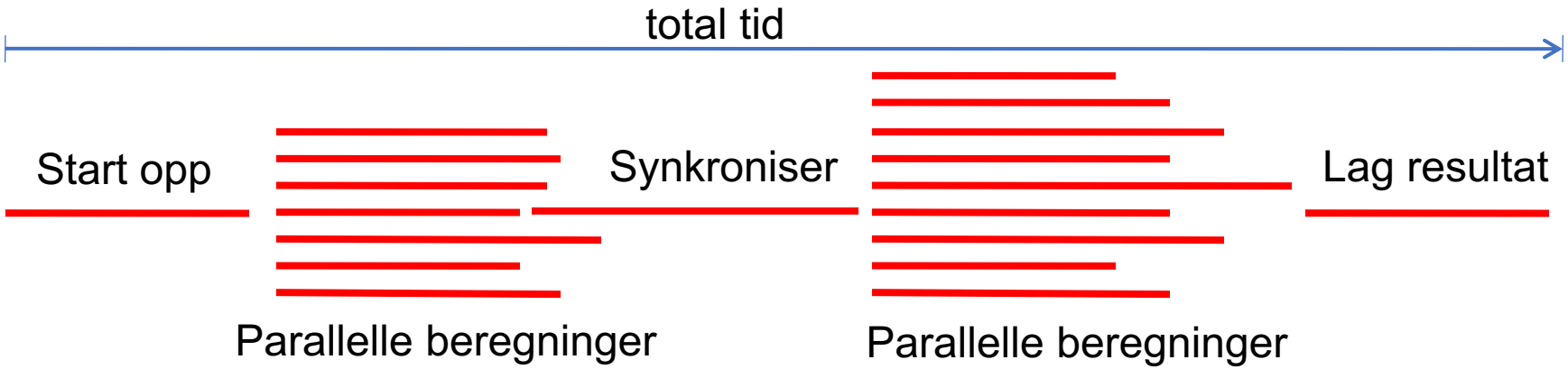


Her er det ingen som må vente (annet enn på låsen)

Men hvordan skal vi vite når alle lete-trådene er ferdige ?

Amdahls lov

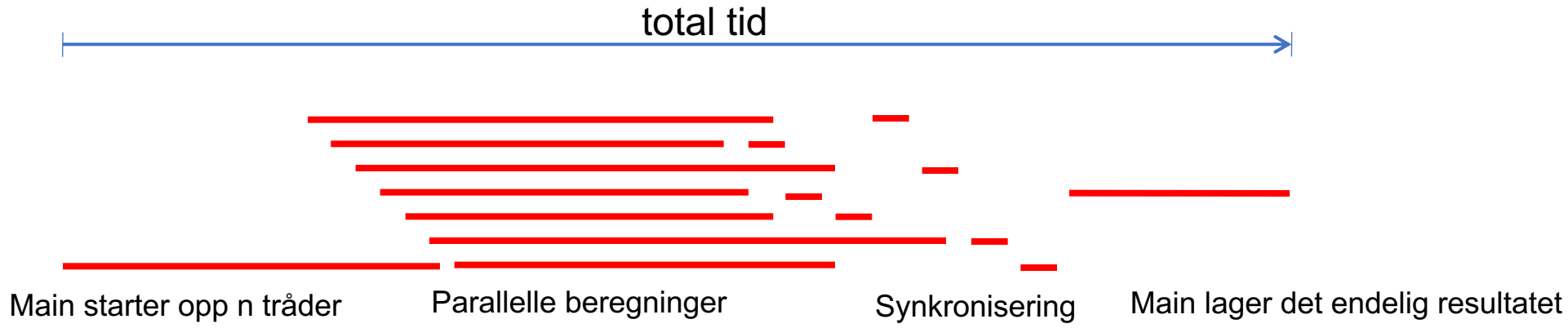
- En beregning delt opp i parallell går fortere jo mer uavhengig delene er
- **Amdahls lov**:
  - Totaltiden er
    - tiden i parallell +
    - tiden det tar å kommunisere / synkronisere/ gjøre felles oppgaver
  - Tiden det tar å synkronisere er ikke parallelliserbar (hjelper ikke med flere prosessorer)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



Du skal ikke pugge Amdahls lov, du skal bare skjønne den

Amdahls lov og ”finn minste tall”

- Totaltid:
  - Tiden det tar å lage og sette i gang 64 tråder
    - (kan det gjøres i parallell ?) (log $_2$ 64 = 6) *
  - Tiden det tar å finne et minste tall i min del av tabellen
  - Til slutt i monitoren: En og en tråd må teste sitt resultat
    - (Kan det gjøres i parallell ?)

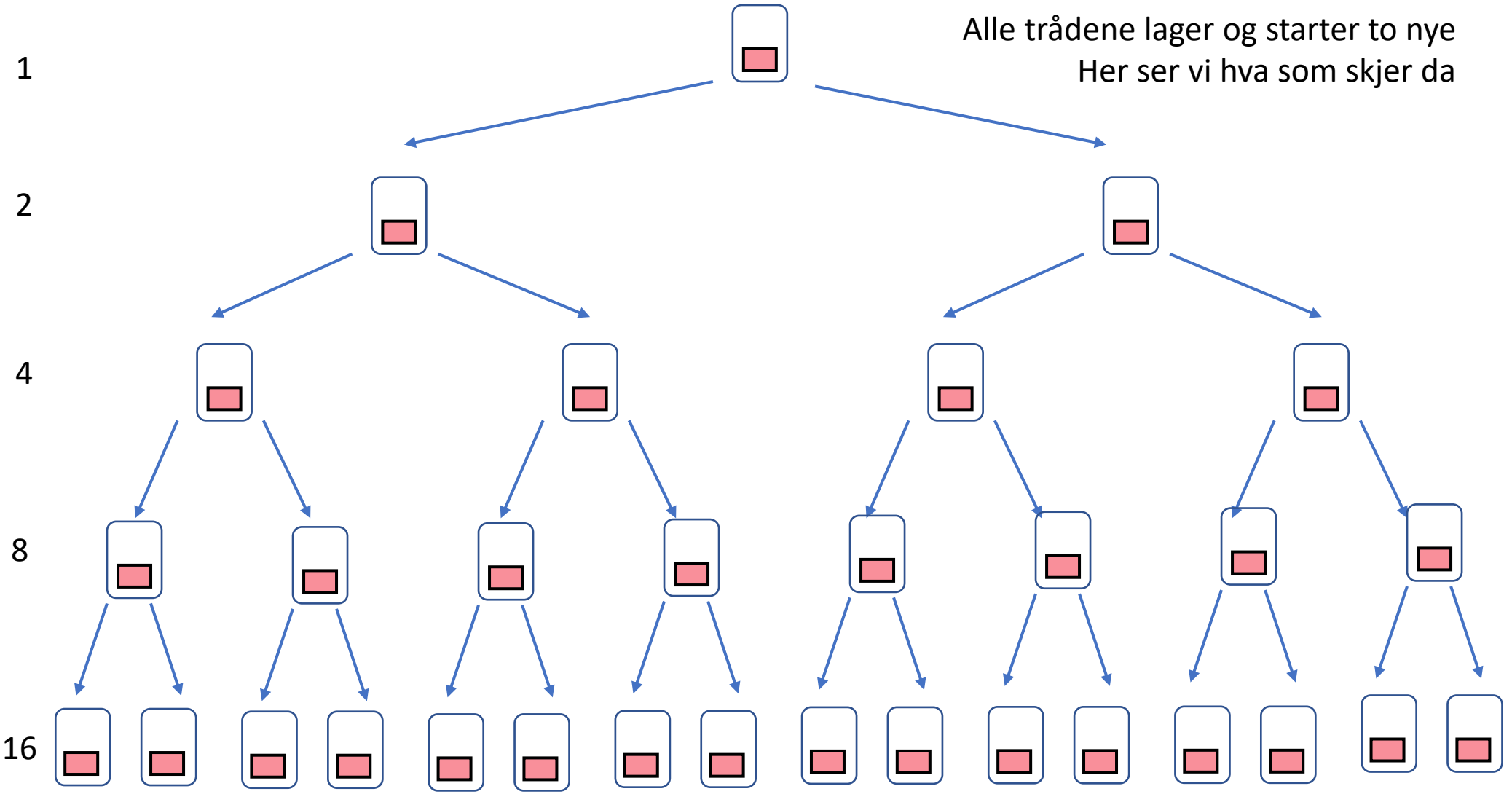


* Start med én tråd, vha. doblinger 6 ganger har vi 64 tråder, dvs, $2^6 = 64$, og $\log_2 64 = 6$

2, 4, 8, 16, 32, 64

Trær er populære i informatikk:

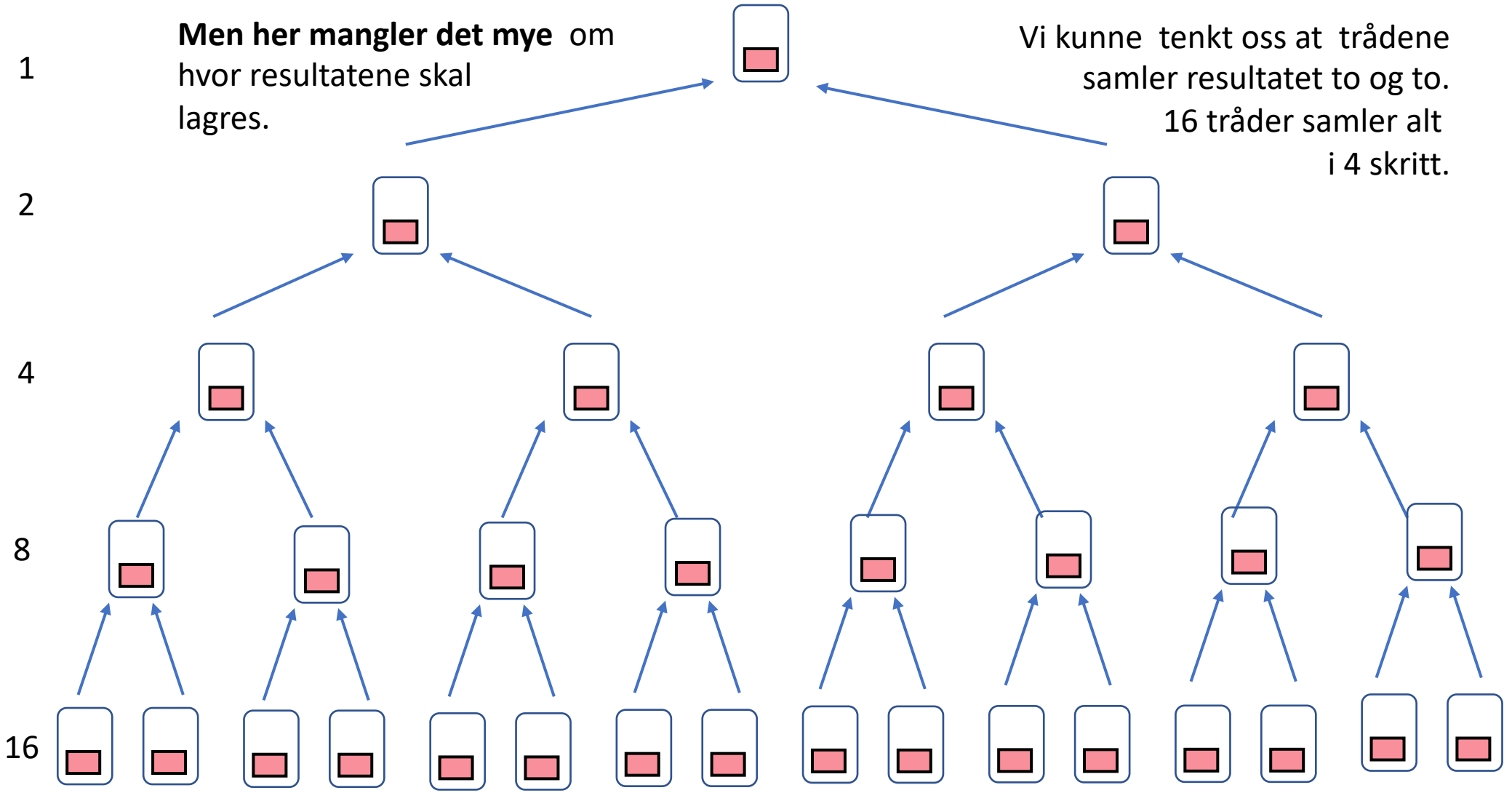
Alle trådene lager og starter to nye
Her ser vi hva som skjer da



Trær er populære i informatikk:

Men her mangler det mye om hvor resultatene skal lagres.

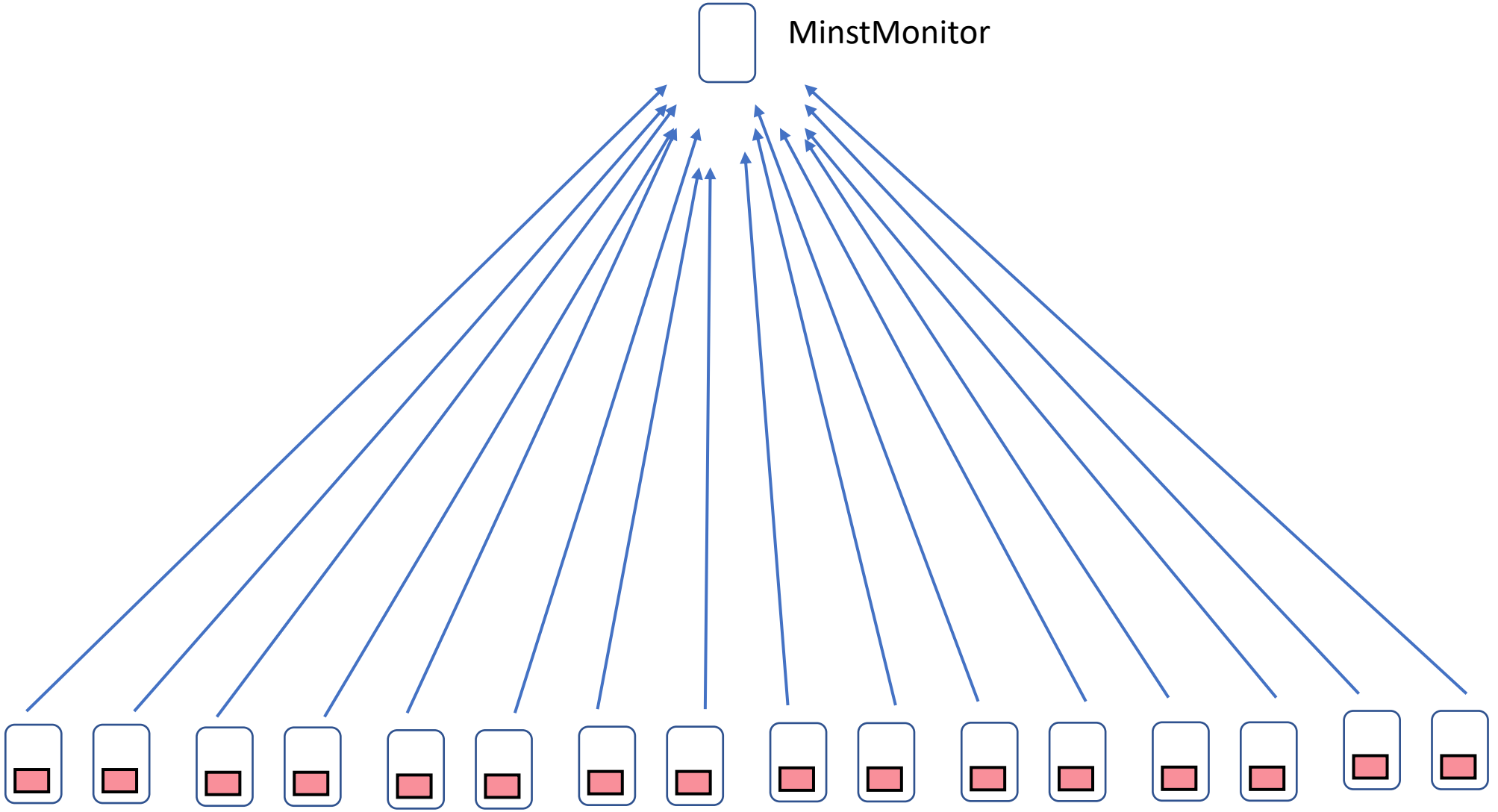
Vi kunne tenkt oss at trådene samler resultatet to og to. 16 tråder samler alt i 4 skritt.



På forrige bilde ("Her mangel det mye")

- For spesielt interesserte (meget avansert bruk av pensum i IN1010):
  - Det er mulig for en klasse som implementerer Runnable å være en monitor også
  - Når treet lages kan en nye tråd få med en referanse til sine mor.
  - Når resultatet er laget av hver av de to barna kan resultatet sendes til moren, men siden det er to barn må metoden som tar i mot resultatet være en kritisk region og objektet må være en monitor.
  - Moren må vente på at begge barna har lagt inn et resultat før hun sender sitt resultate til sin mor igjen, osv.

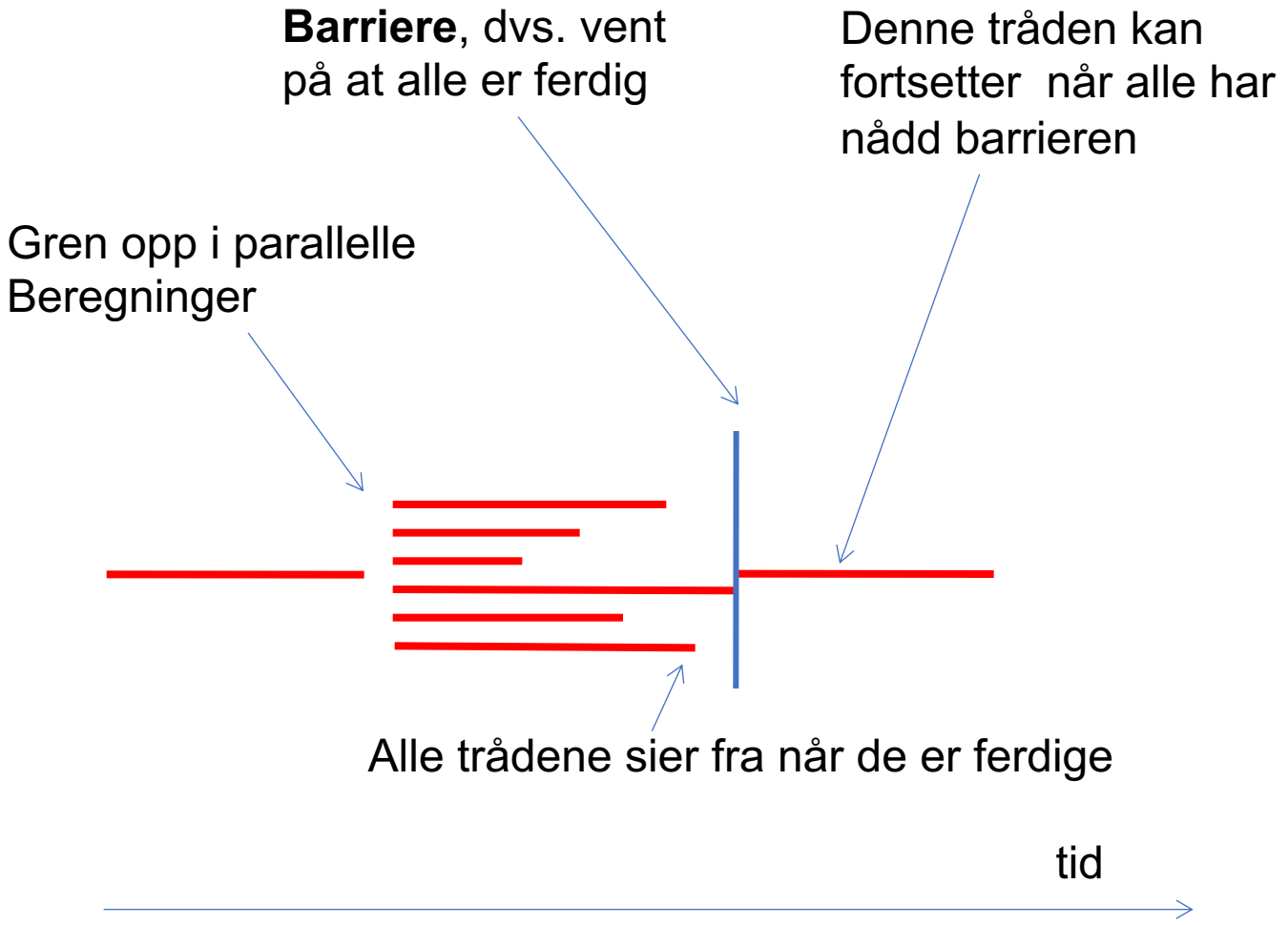
Men vi skal gjøre det enklere for oss selv:



Men, tilbake til "finn minste tall i en stor array" ved hjelp av mange (64?) tråder

- Hvordan kan vi vite når alle lete-trådene er ferdige og resultatet ligger ferdig i monitoren?

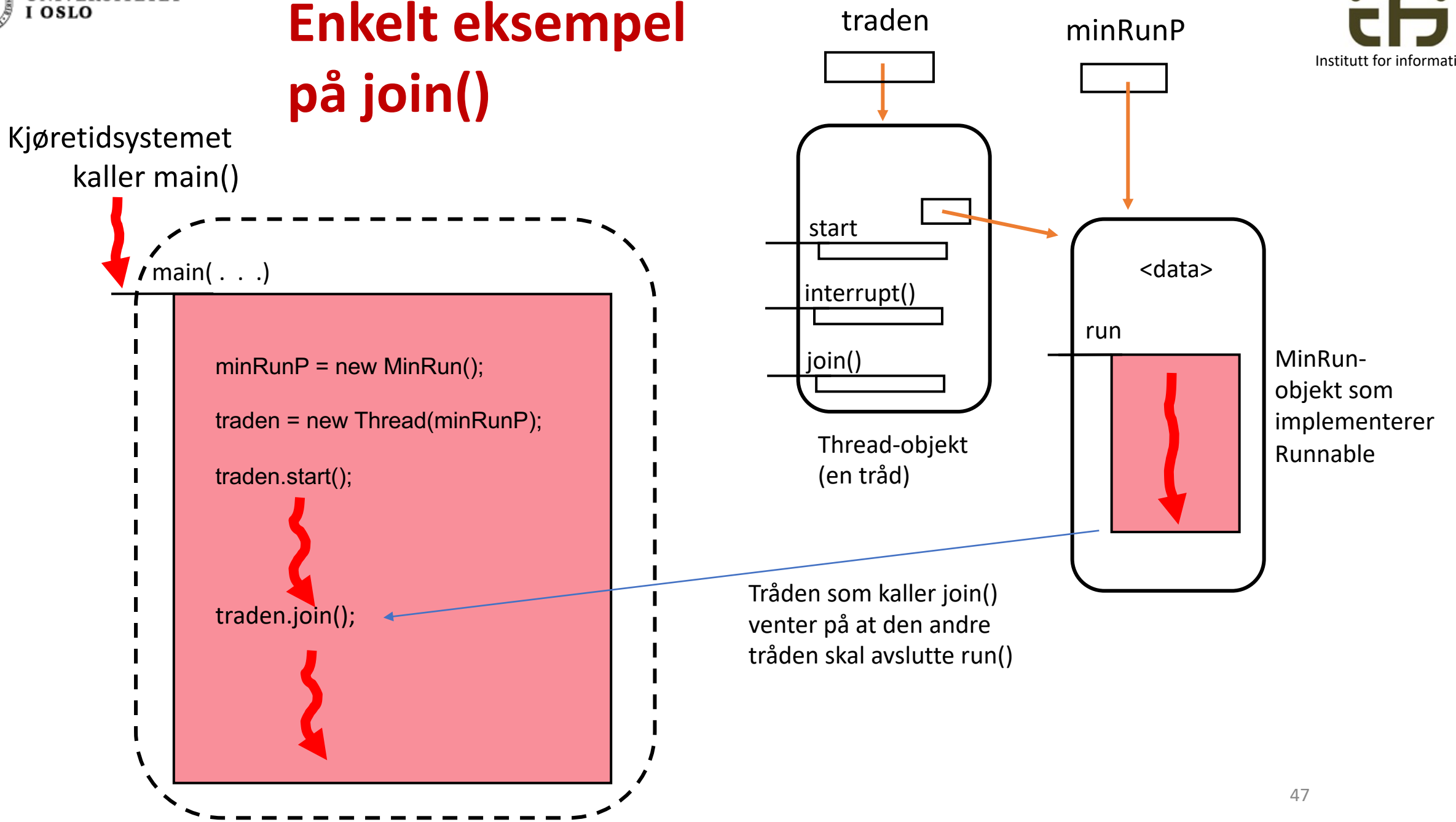
Barrierer i parallellprogrammering



Gafling og møting (fork and join)

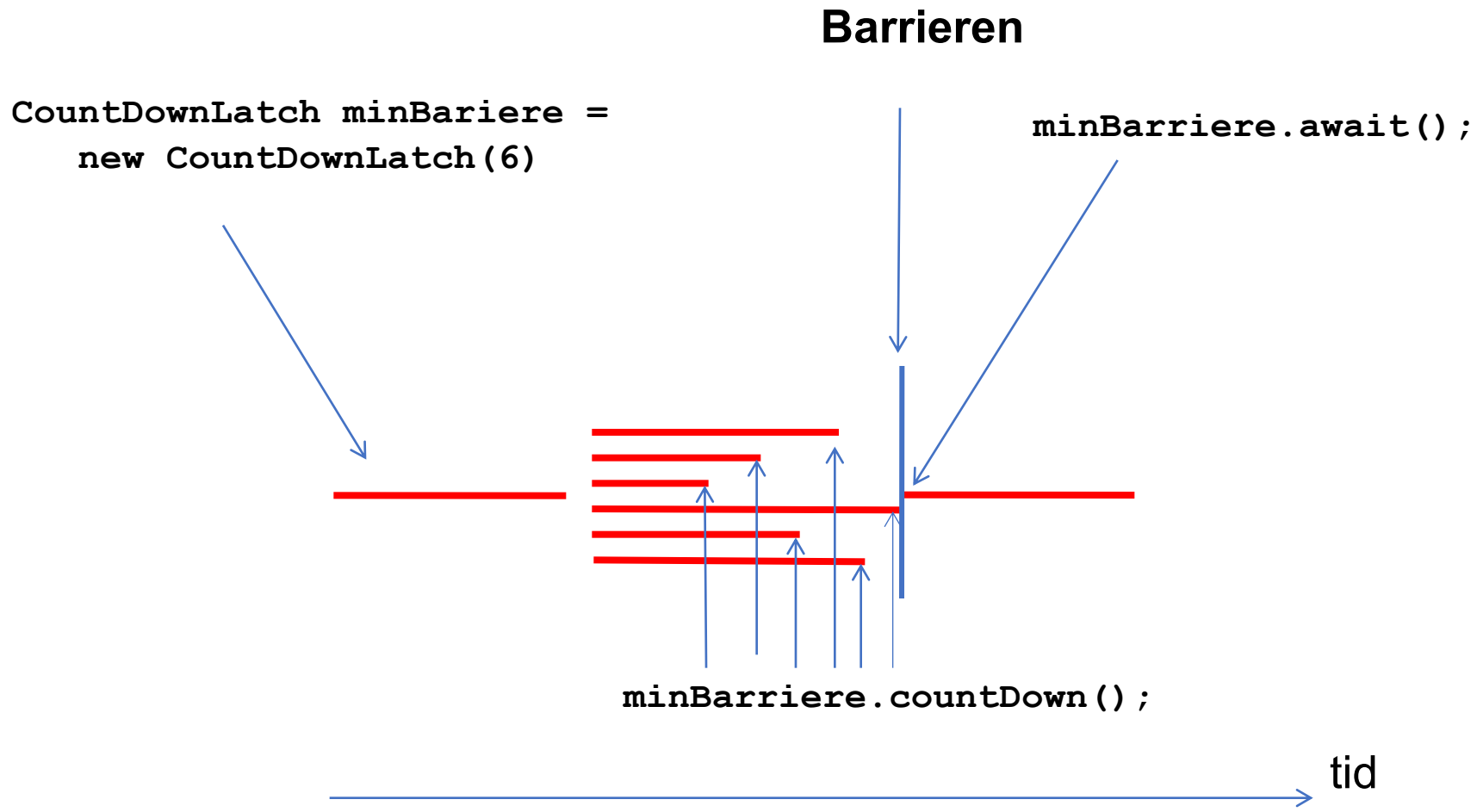
- I parallellprogrammering og på engelsk brukes "fork" om å lage en ny tråd eller en ny prosess
- Jeg tror ikke det brukes noen norsk oversettelse (å gafle? forgrening?)
- Det finnnes ikke noen fork-mekanisme i Java
  - Du må selv lage en ny tråd, og eventulet gå i løkke for å lage flere
- Når tråder terminerer kalles det gjerne join (møtes ?)
  - Dag snakket om join() forrige uke
  - Her kan programmet som startet opp alle trådene vente på hver og en med join()

Enkelt eksempel på join()

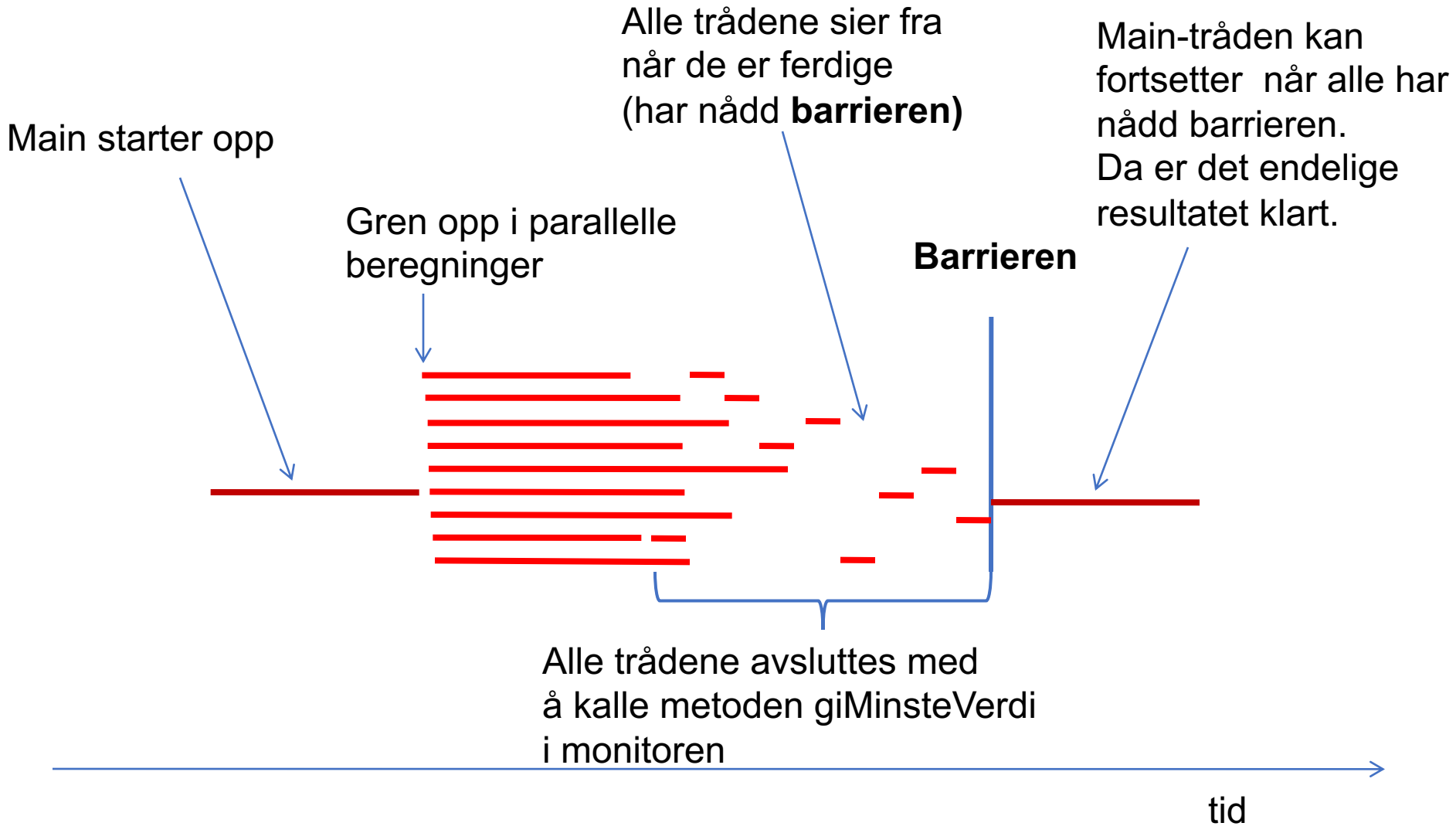


Barrierer i Java


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

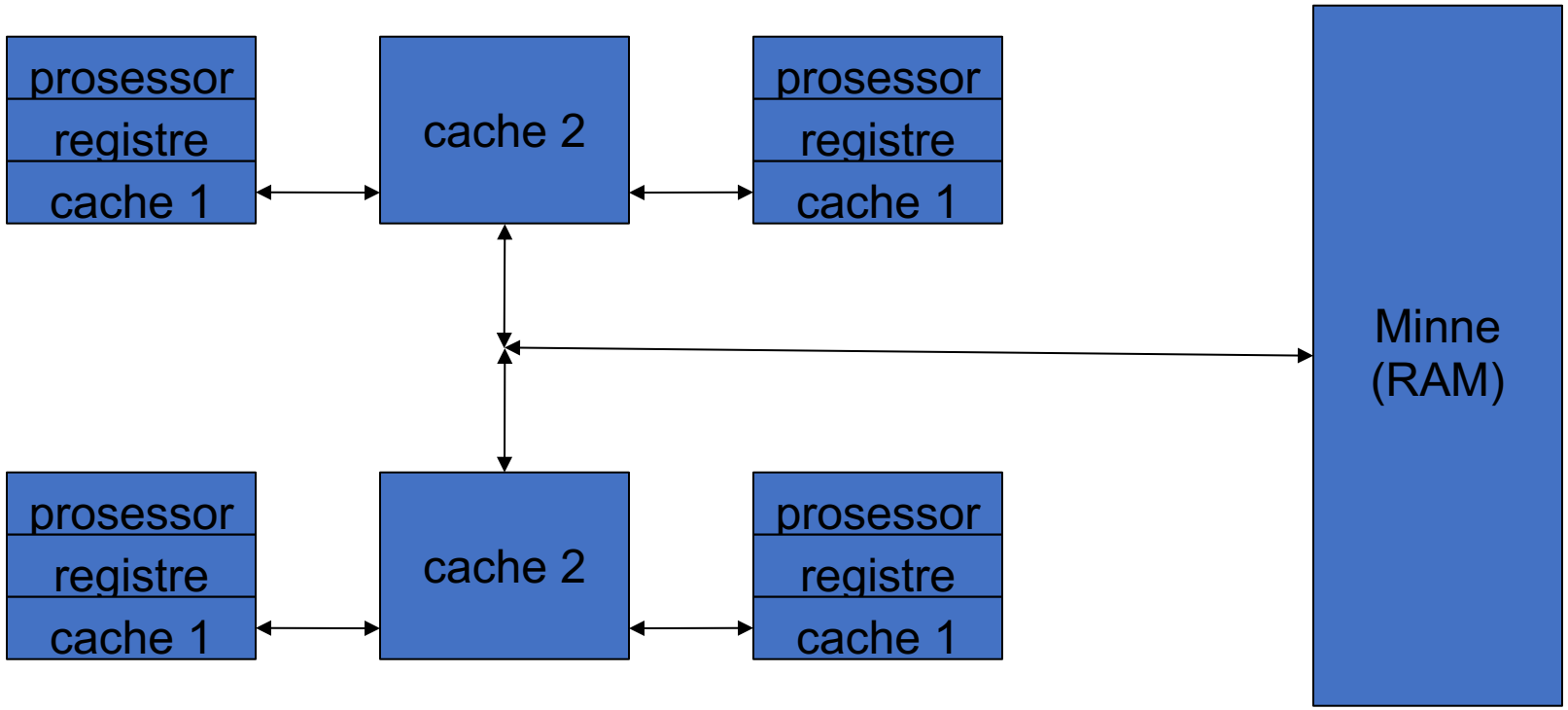


Barrieren i dette eksemplet



Hvorfor tar synkronisering mellom tråder tid ?

Maskinarkitektur, f.eks. 4 kjerner



For de mest interesserte: “Javas minnemodell”

Volatile variable

En variabel som er deklarert volatile caches ikke
(og oppbevares ikke i registre
(lenger enn helt nødvending))

- En volatil variable skrives helt tilbake i primærlageret.

boolean stopp = false;

En tråd:

```
stopp = true;
```

En annen tråd:

```
while(! stopp) {
    . . .
}
```

Må da deklarere:

volatile boolean stopp=false;

Synkronisering og felles variable i en monitor

- This means that any memory operations which were visible to a thread before exiting a synchronized block are visible to any thread after it enters a synchronized block protected by the same monitor, since all the memory operations happen before the release, and the release happens before the acquire.

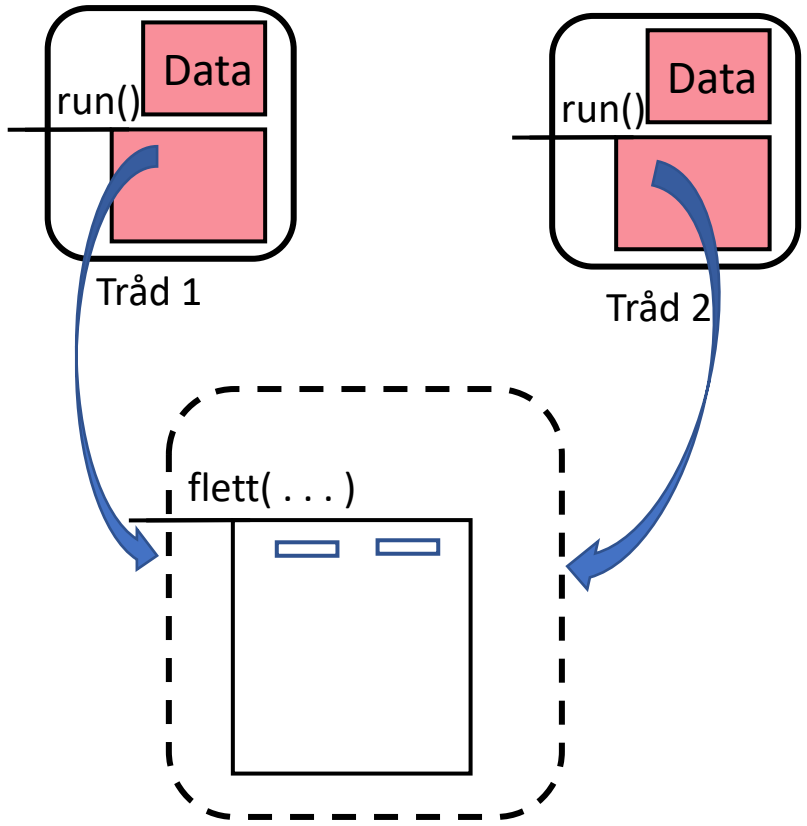
From: JSR 133 (Java Memory Model) FAQ
Jeremy Manson and Brian Goetz, February 2004

Litt på siden av pensum i IN1010.
Mest ment som en forsmak for de som er mer interessert.

Fordelen med konstanter (immutable objects)

- Konstanter kan det aldri skrives til
- Minnemodellen for konstanter blir derfor veldig enkel.
- Prøv å ha mest mulig konstanter når data skal deles mellom tråder.
- Eksempel: Geografiske data, observasjoner
- Hvis du ønsker å gjøre en forandring:
  - Kast det gamle objektet og lag et nytt.

Oblig 5 og bruk av samme metode



Om du programmerer flettingen av hashmaper i en metode som bare bruker lokale variable, så kan alle tråder som vil kalle og bruke denne metoden samtidig gå i ekte parallell.

Derfor kan en static-metode brukes av alle trådene dine uten å gå i bena på hverandre.

Husk: Hver eneste tråd har sin egen stabel av metodekall ("the call stack")

Nytt tema
(men fortsatt tråder)

Vranglås
Engelsk: Deadlock

Horstmann kap. 20.5

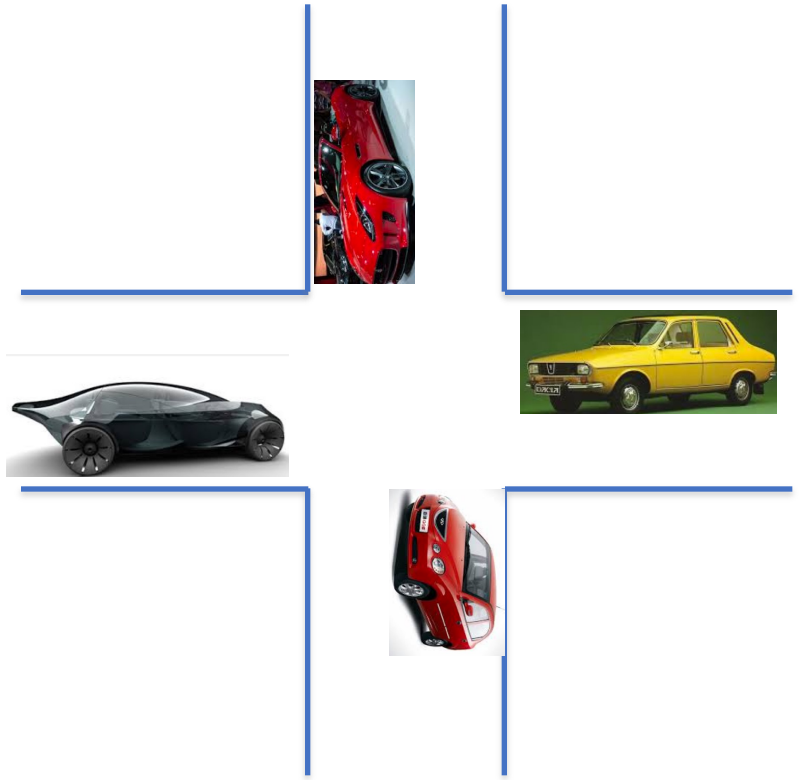
55

Vranglås (deadlock)

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

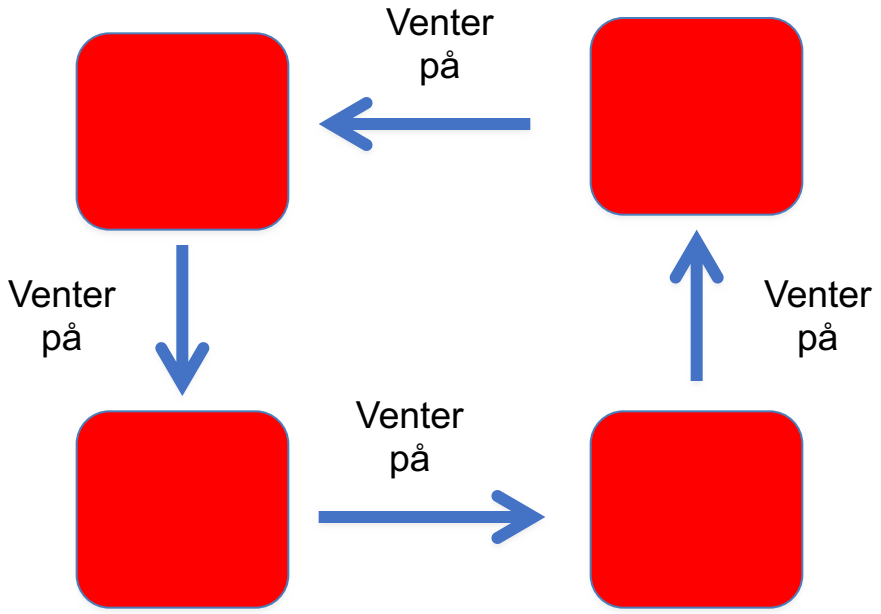
    ->

    Alle stopper = VRANGLÅS



Vranglås betyr

- Flere tråder venter på hverandre
- Syklisk venting



Vranglås kan oppstå når flere tråder kjemper om felles ressurser

- En eller flere felles ressurser ønskes av mer enn en tråd
- Hvis en tråd først tar en ressurs og deretter en annen . . .

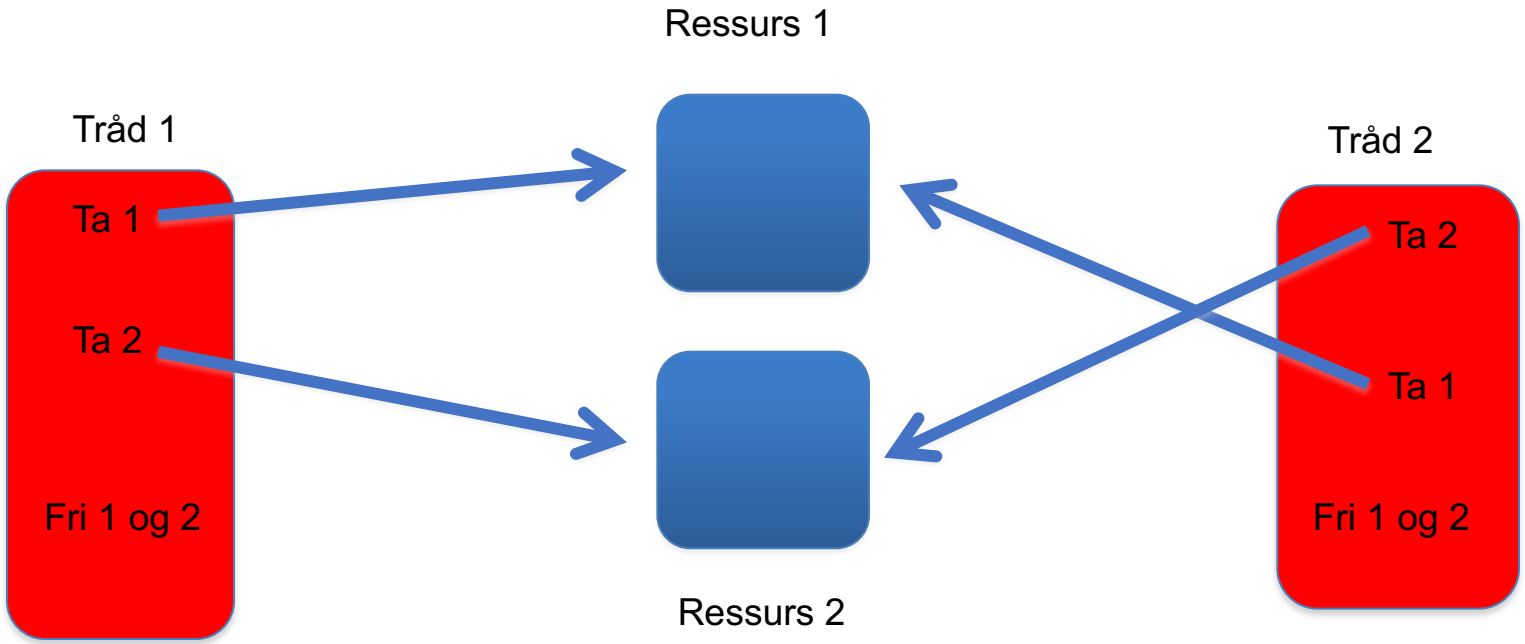
Unngå vranglås

1. Ta bare én ressurs
2. Ta alle på en gang, 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

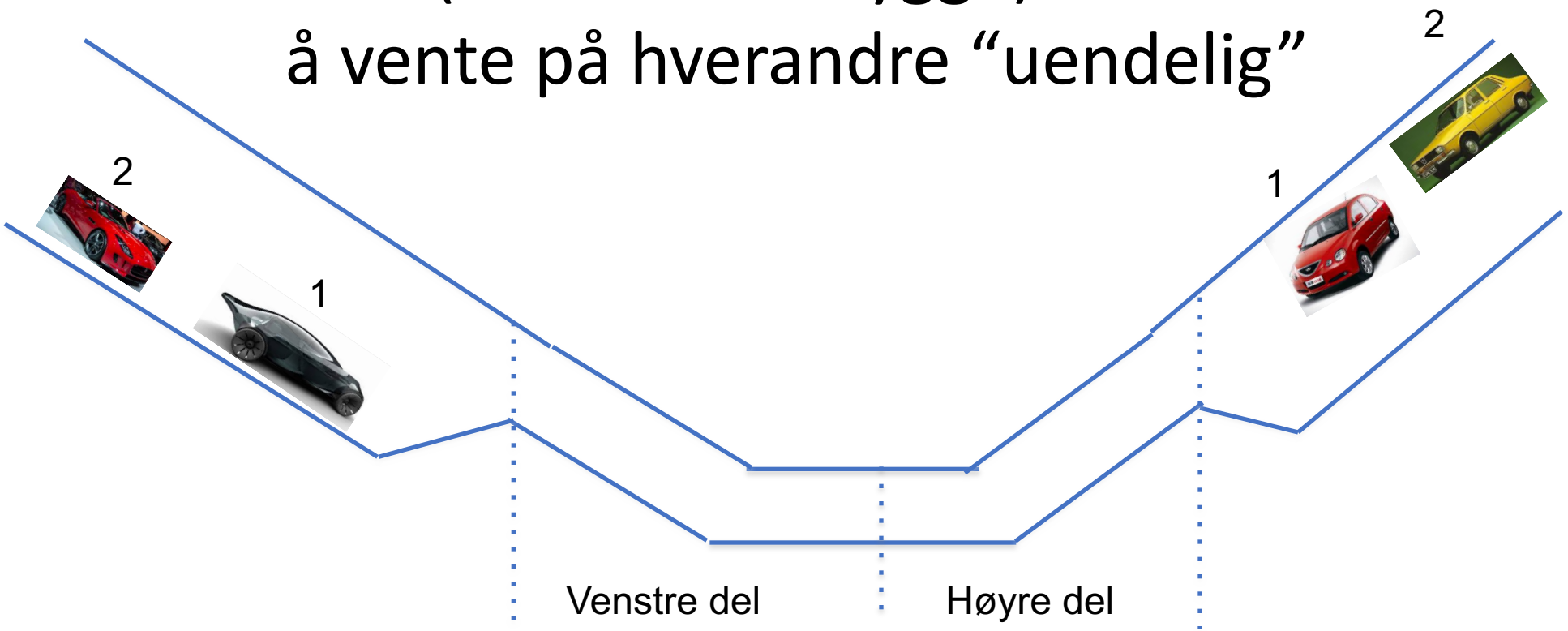
Enkleste eksempel på vranglås

- To tråder
- To ressurser som tas i forskjellig rekkefølge



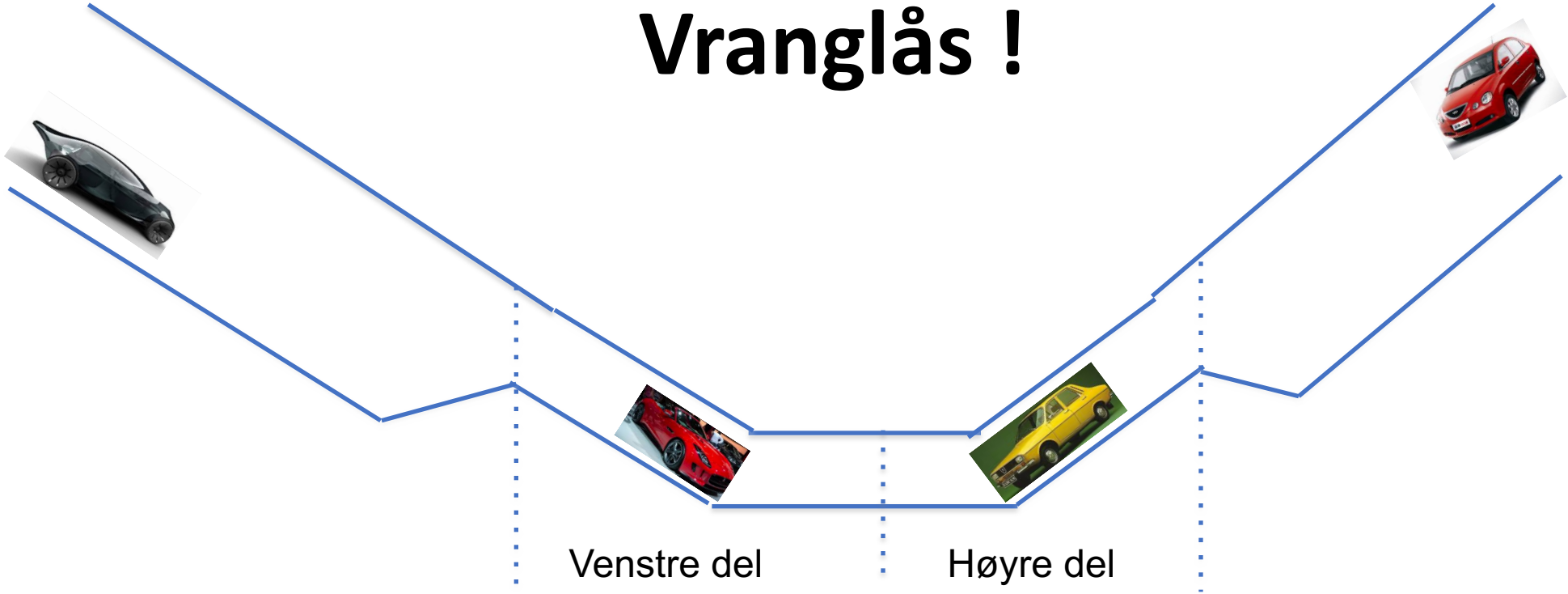
Enkleste eksempel på vranglås:

To biler (som ikke vil rygge) kan risikere å vente på hverandre "uendelig"



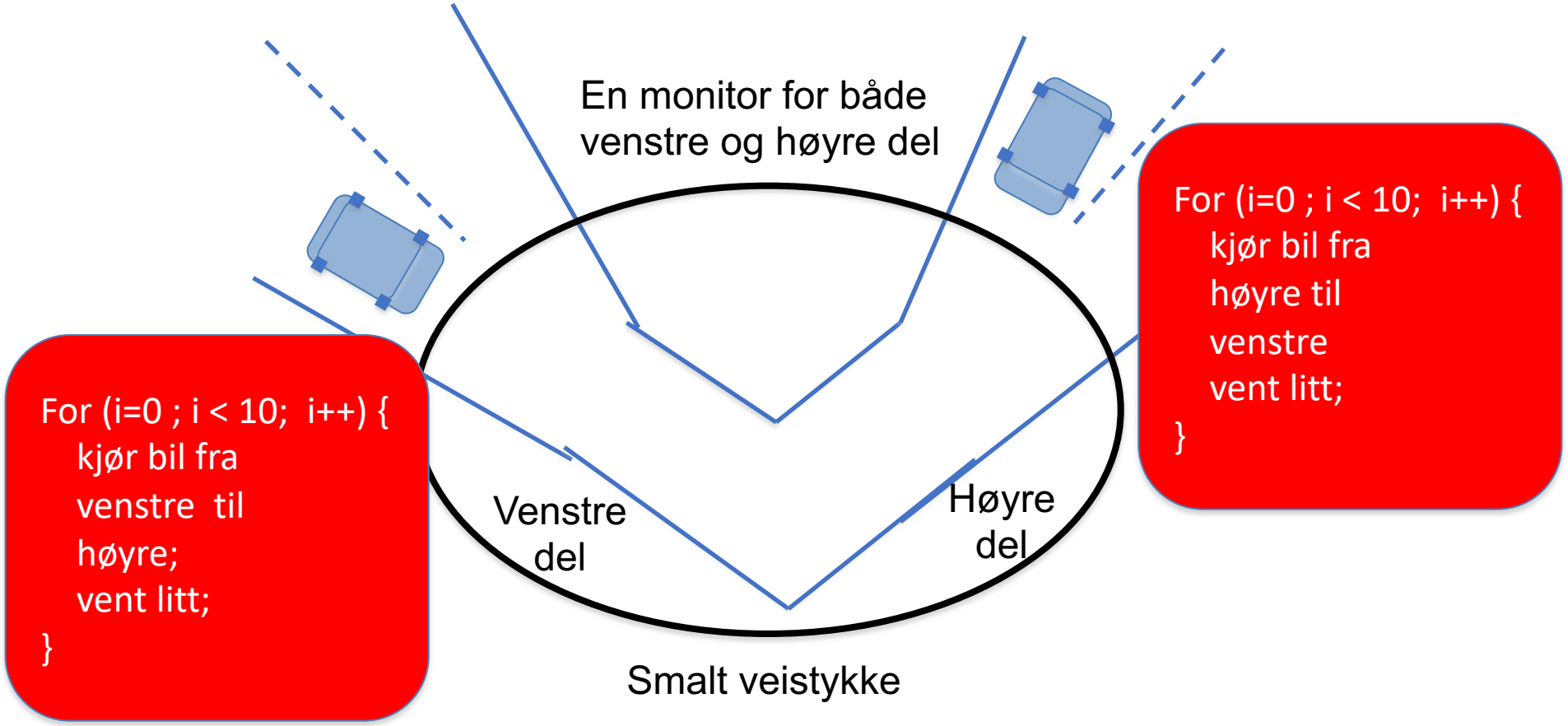
Smalt veistykke, to biler
kan ikke passere hverandre.
Bilene kan bare se den første delen.

Vranglås !

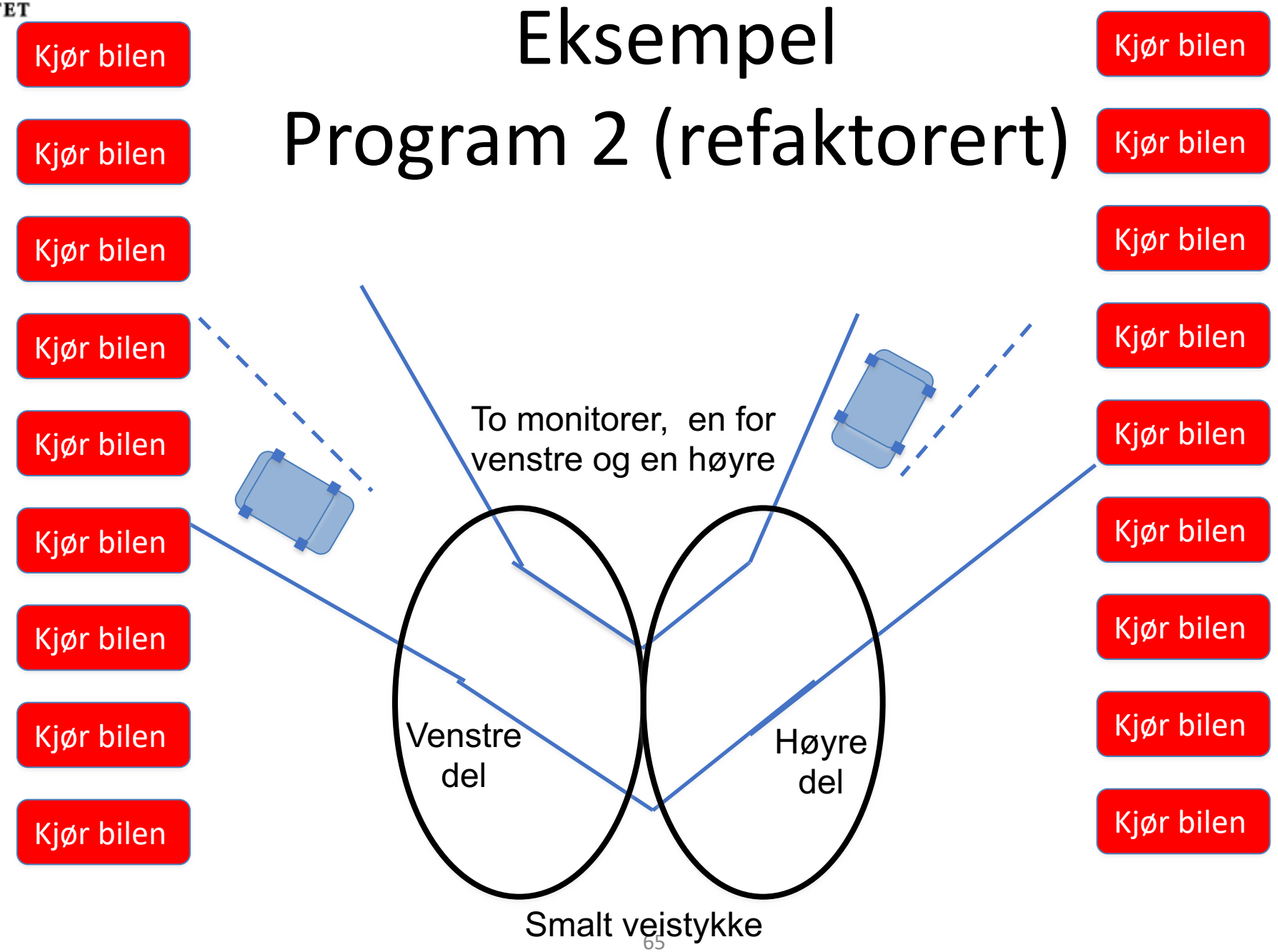


Det smale veistykket består av to ressurser, og begge bilene venter på at den andre bilen skal bli ferdig med å bruke sin ressurs (de har tatt ressursene i forskjellig rekkefølge)

Eksempel Program 1



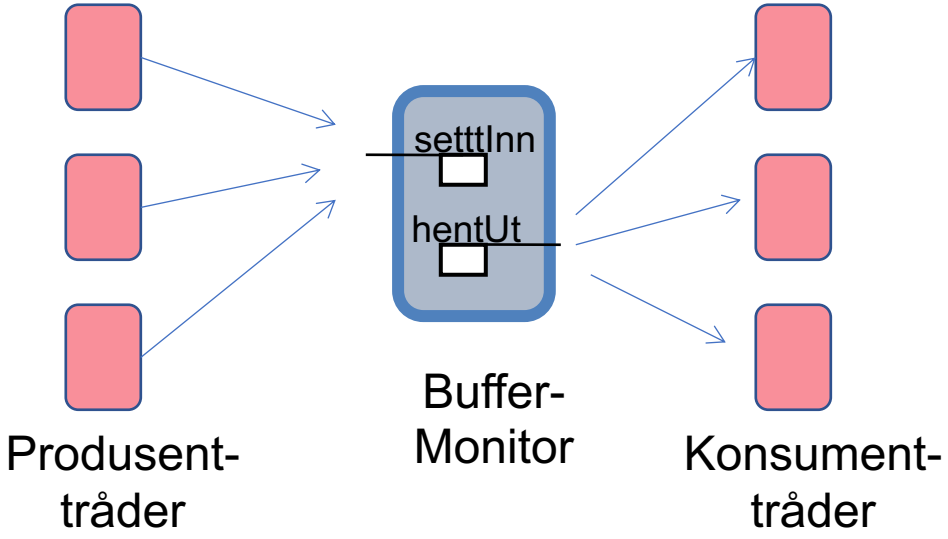
Eksempel
Program 2 (refaktorert)

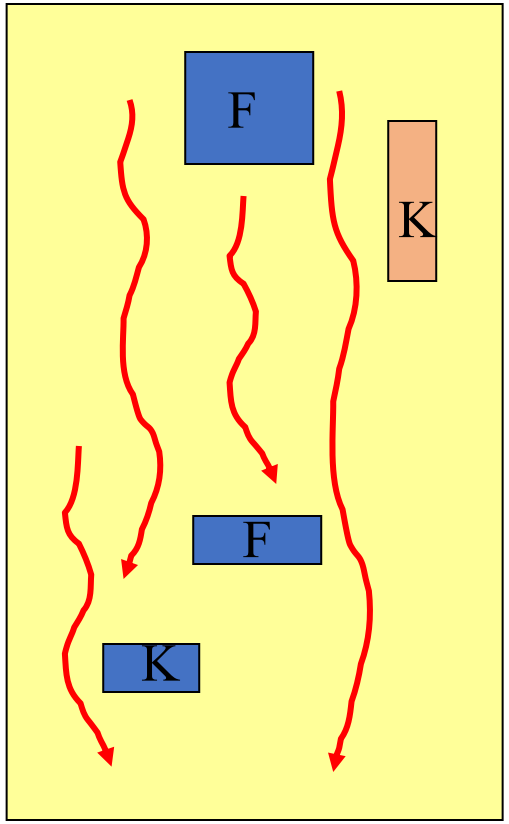


Hoved "take-away" i dag

Du kan programmere parallelle aktiviteter i Java ved hjep av tråder.

Tråder deler adresserom.
Et felles objekt kalles en **monitor**. Metodene i en monitor er *kritiske regioner.*





K: konstante data (immutable)

Oppsummering: Vi har lært:

- Hvordan tråder kommuniserer seg imellom ved hjelp av monitorer
- Hvordan programmer venter i en monitor
  - og starter opp igjen de som venter
  - Låsing og venting og signalering (på betingelser) skjer bare inne i monitorer.
- At det er utfordrende å programmere parallelle aktiviteter
  - Viktig å resonere om programmets tilstand.
    - Bruk invarianter
  - Nesten umulig å teste nok tilfeller
    - Tidsavhengige feil kan vise seg først etter mange år.
- Amdahls lov sier at mer parallellitet er bedre, og at synkronisering koster
- Vranglås kan forekomme i parallellprogrammering.
  - Vranglås kan unngås på forskjellige måter