UNIVERSITETET I OSLO

Institutt for informatikk

Kort versjon

IN1010 - Våren 2023
Parallellitet og tråder - del 1
20. mars 2023

Stein Gjessing

Horstmann kap 20.1 – 20.3

Oversikt

- Hva er parallelle programmer?
- Hvorfor parallelle programmer ?
- Hvordan kan dette skje på én kjerne/CPU/prosessor ?
- Hvordan kan dette skje med flere kjerner/CPU-er/prosessorer ?
- På engelsk: Parallel vs Concurrent computing
- Hva er en prosess? - Hva er en tråd?
- **Tråder i Java**
- **Hvordan bruker vi tråder**
- **Oppdateringsproblemet (race conditions)**
  - Samtidig oppdatering av felles data
    - Løsningen: Kritiske regioner og monitorer

Hvordan lage 5 papirfly?

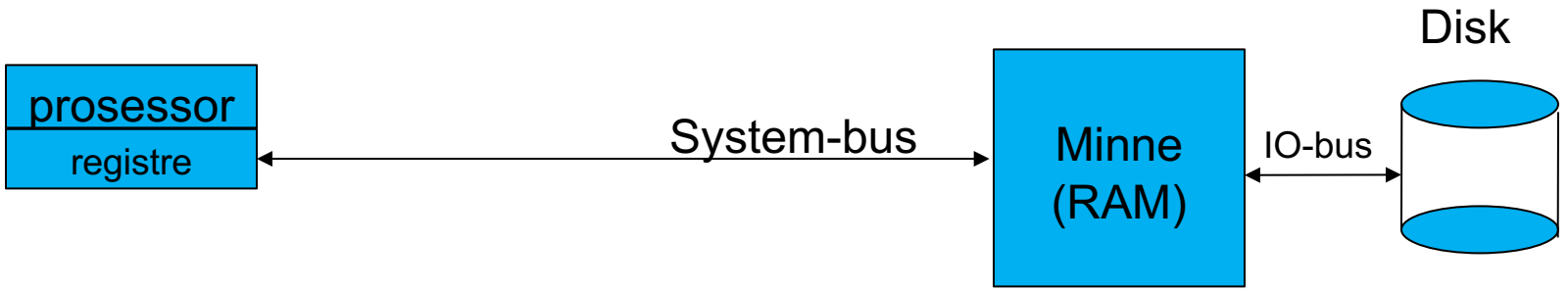
Hvorfor parallelle programmer ?

- **Naturlig parallellitet** i oppgaven:
  - Maskinen trenger å kjøre flere programmer samtidig for egen adminsitrasjon (filbehandling, nettverk, . . .) eller for en ellere flere brukerne (se film, chat, samtidig som du jobber . . .)
  - Ofte *må* mange ulike brukere jobbe (nesten) samtidig på *samme data* F.eks. et bestillingssystem: en kino, en flyavgang,…
  - Oppgaver som naturlig modelleres i flere parallelle deler
    - Bredde-parallellitet - Samlebånd-parallellitet

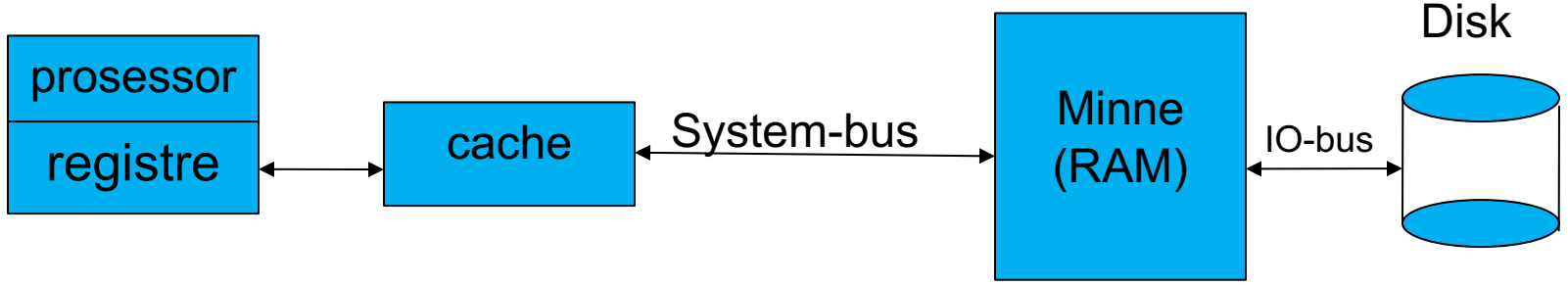
- **Ekte parallellitet - hastighetsøkning**
  - Når jeg simulerer deler av Internet bruker jeg så mange kjerner som mulig
  - Tunge beregningsoppgaver som ellers tar (for) lang tid å løse.
  - ”Dual-core”, "8-core", ”N-core”: Ekte parallellitet

Datamaskinarkitektur (Computer Architecture)

1. I gamle dager =
(nesten) fremdeles dagens abstrakte modell:

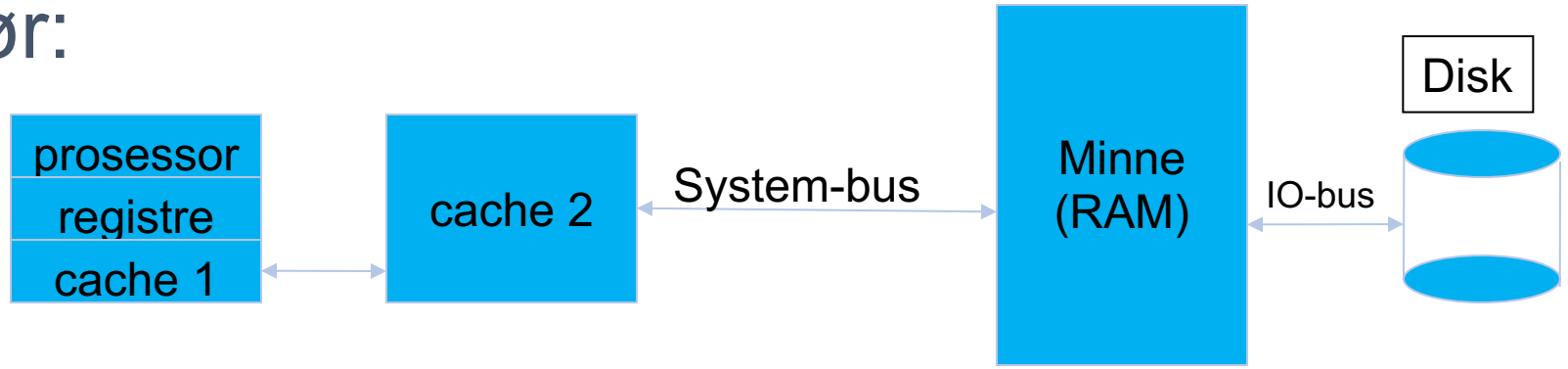


2. Før:

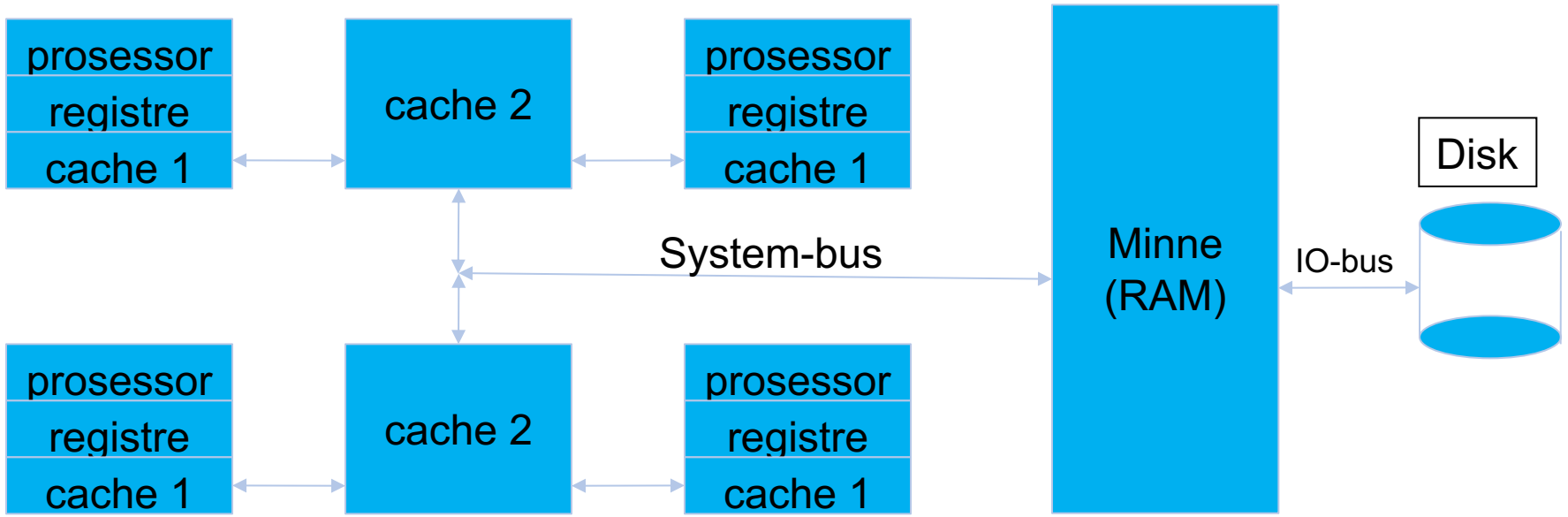


Datamaskinarkitektur (Computer Architecture)

3. Før:

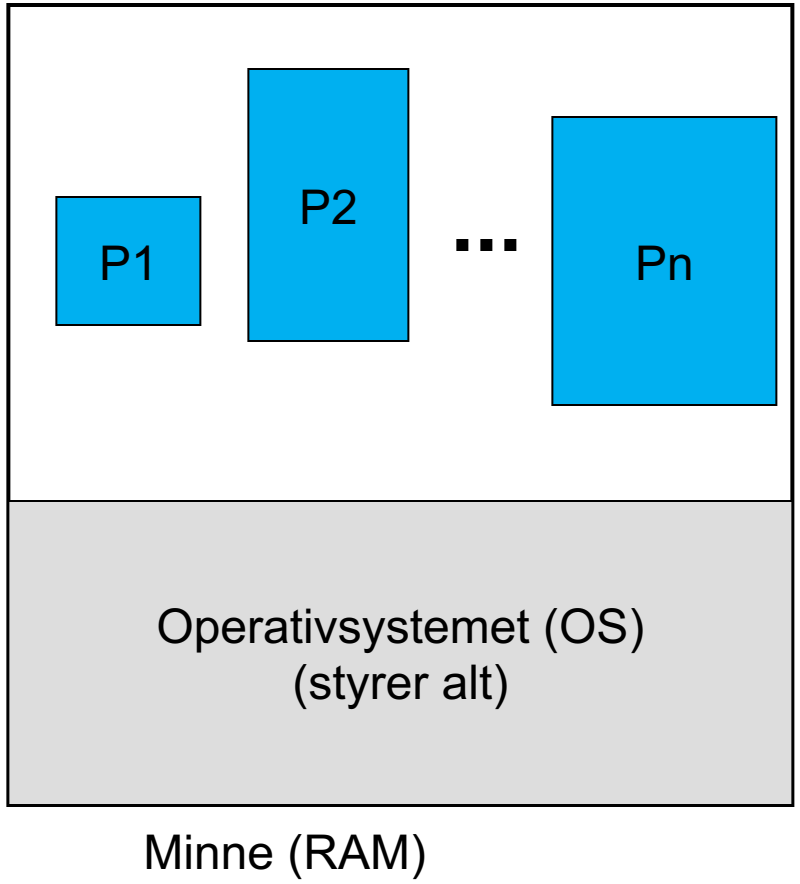


4. Nå, f.eks:



Parallellitet: prosesser og tråder

- Operativsystemet (OS) administrerer
  - Prosesser
- Prosesser (P1, P2, ... , Pn)
  - En prosess er utføringen av et program
  - Er isolert fra hverandre, kan i utgangspunktet bare snakke til operativsystemet
  - Kan sende meldinger til andre prosesser via operativsystemet
  - Eier hver sin del av hukommelsen
  - Eier hver sine filer,...
- Et program
  - Startes som én prosess (kan så evt. starte andre prosesser)
  - Kan startes som en prosess fra kommandolinjen
- En tråd
  - Er parallelle eksekveringer **inne i én prosess**
  - Alle tråder i en prosess deler prosessens del av hukommelsen (ser de samme variable og programkode)
  - Tråder er som ”små”-prosesser inne i en vanlig ”stor” prosess
  - Tråder kan også gå i ekte parallell
  - OS og kjøretidsystemet administrerer trådene.



Hvorfor fant man på tråder ?

- Vi har prosesser – hvorfor ikke bare bruke dem?
  - Det går greit, men litt tregt
  - Å skifte fra at en prosess til en annen tar om lag 20 000 instruksjoner
- Prosesser ble funnet på omlag 1960, tråder minst 20 år seinere.
- Tråder er som små prosesser inne i én prosess, og det er langt rasker å skifte fra en tråd til en annen tråd (tråder kalles ofte lettvektsprosesser)
- Prosesser kommuniserer via operativsystemet
- Tråder kommuniserer via felles data (inne i samme prosess)
- Ellers har tråder og prosesser omlag samme muligheter og problemer når man lager programmer
- Parallellprogrammering
  - = bruke ***flere*** prosesser og/eller ***flere*** tråder for å løse en programmeringsoppgave
  - Programmer med parallellitet er mye vanskeligere å skrive og teste/feilsøke enn bare én tråd (main-tråden) i én prosess (som vi har gjort til nå)

Tråder i Java

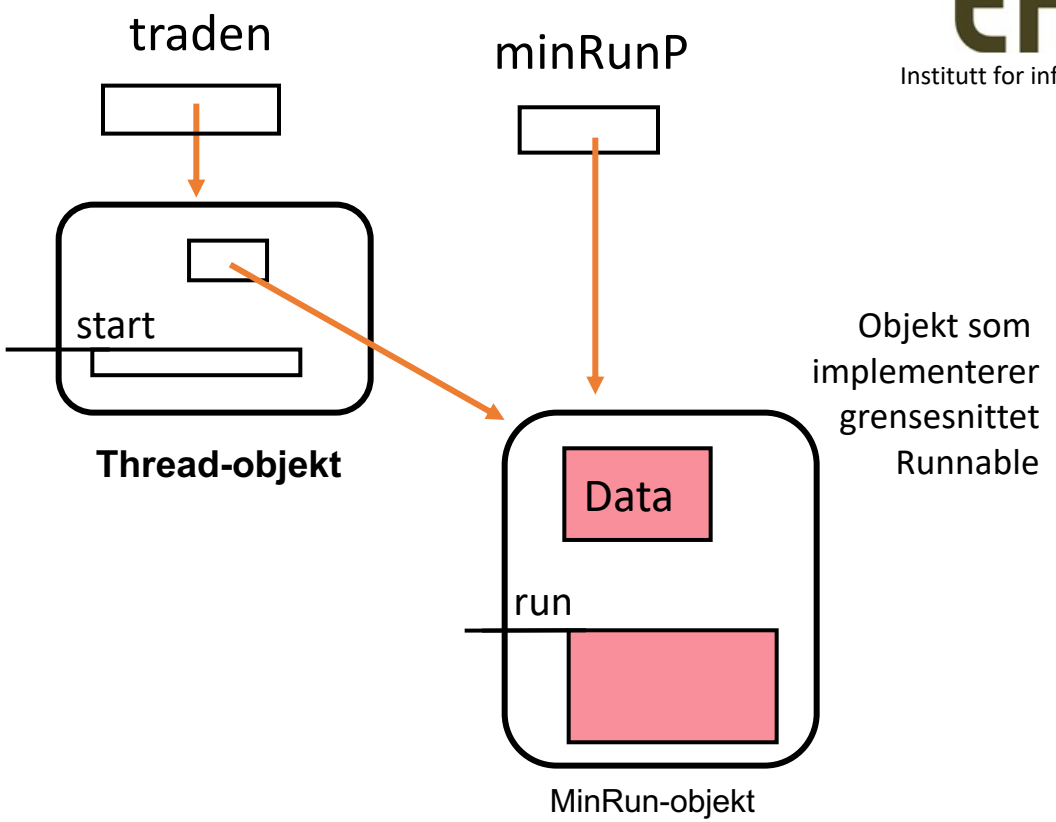
- Én viktig klasse og ett viktig interface i Javas API

```
class Thread {
        public void start() { . . . }
        . . .
}

interface Runnable {
     void run();
}
```

Tråder i Java


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



En tråd lages og startes opp slik:

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

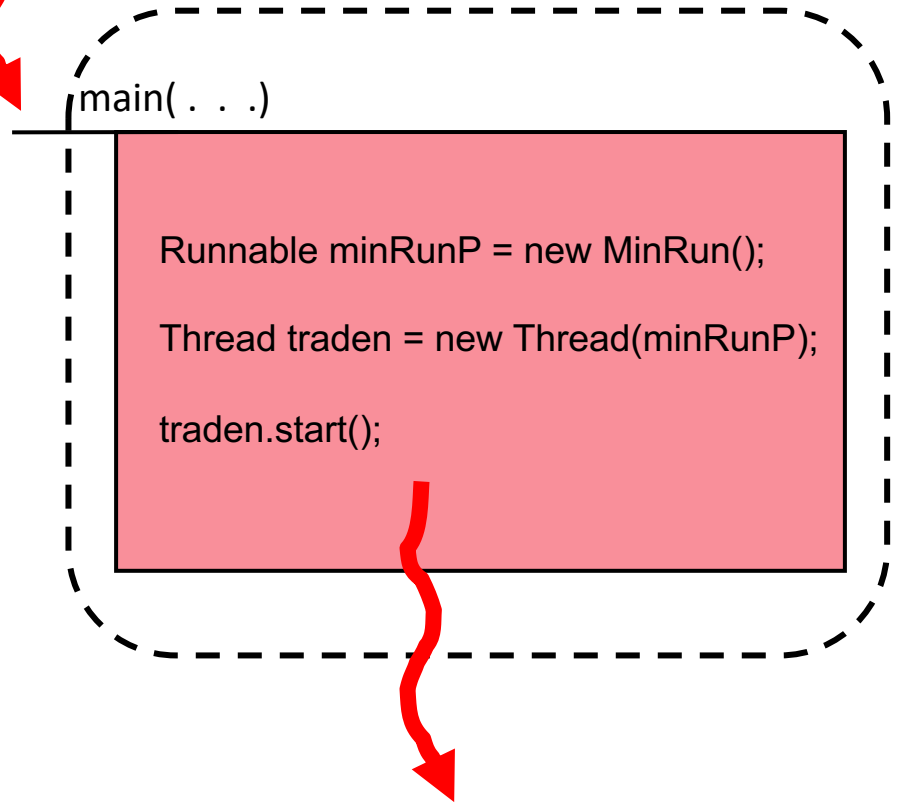
start( ) er en metode i Thread som må kalles opp for å få startet tråden. start-metoden vil igjen kalle metoden run (som vi selv programmerer).

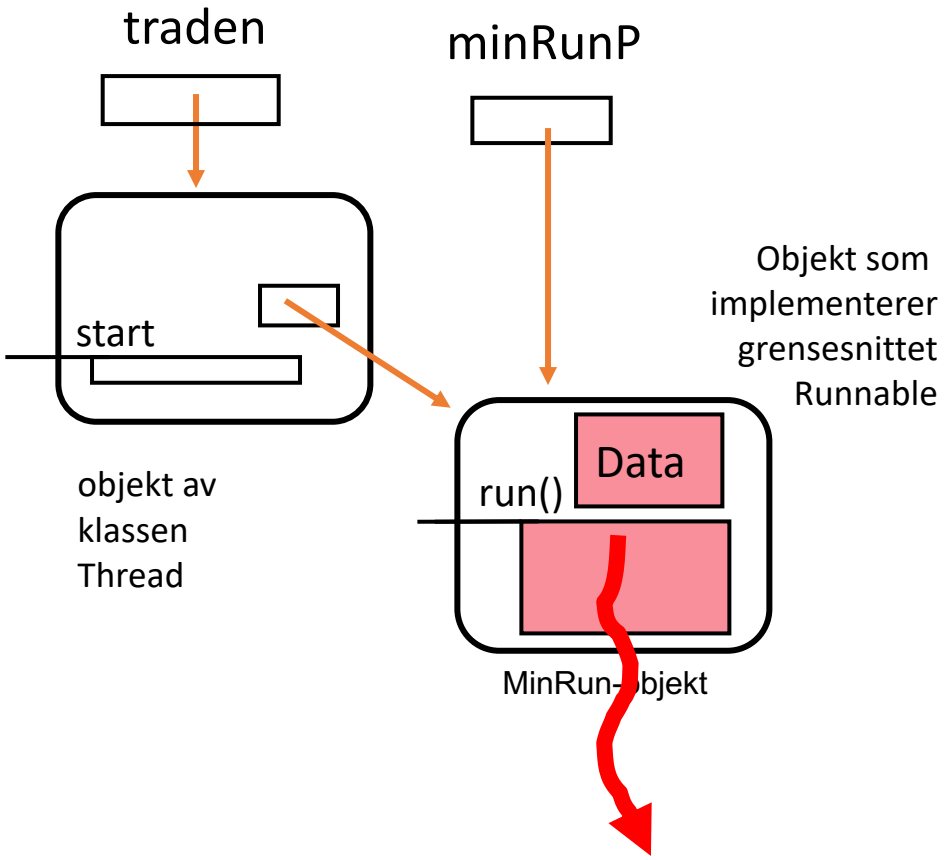
UNIVERSITETET I OSLO

Eksempel: main()-tråden starter én ny tråd

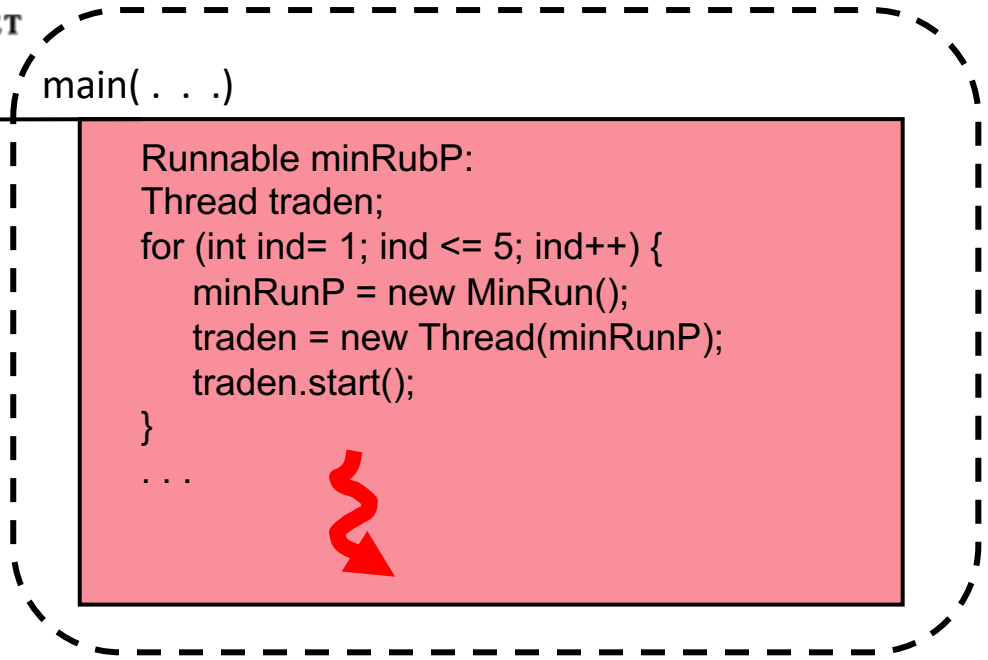
```
class MinRun implements Runnable {
    <Data>
    @Override
    public void run( ) {
       while (<mer å gjøre>) {
             <gjør noe>;
              . . .
       }
    }
}
```

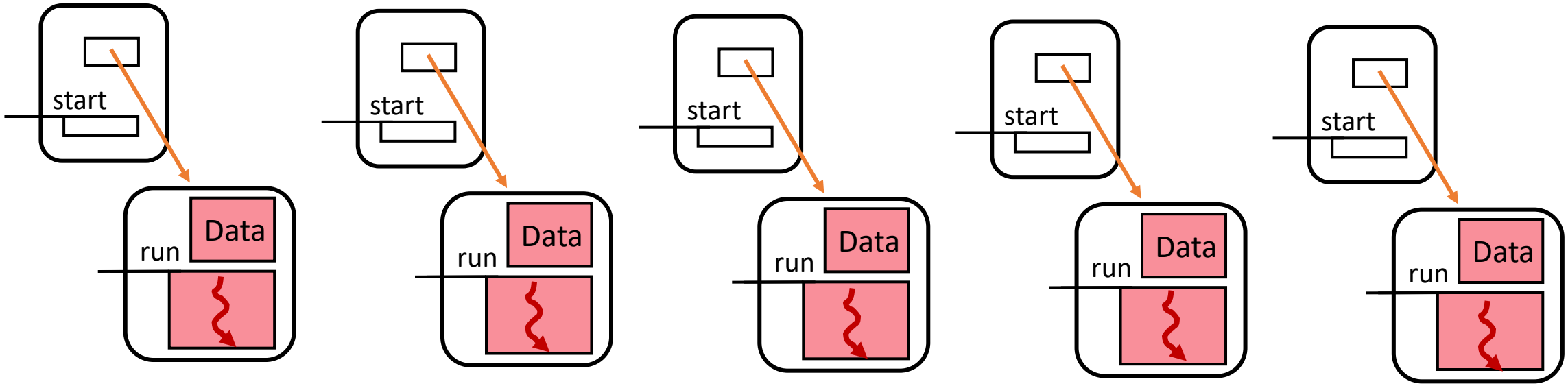
Kjøretidsystemet kaller main()



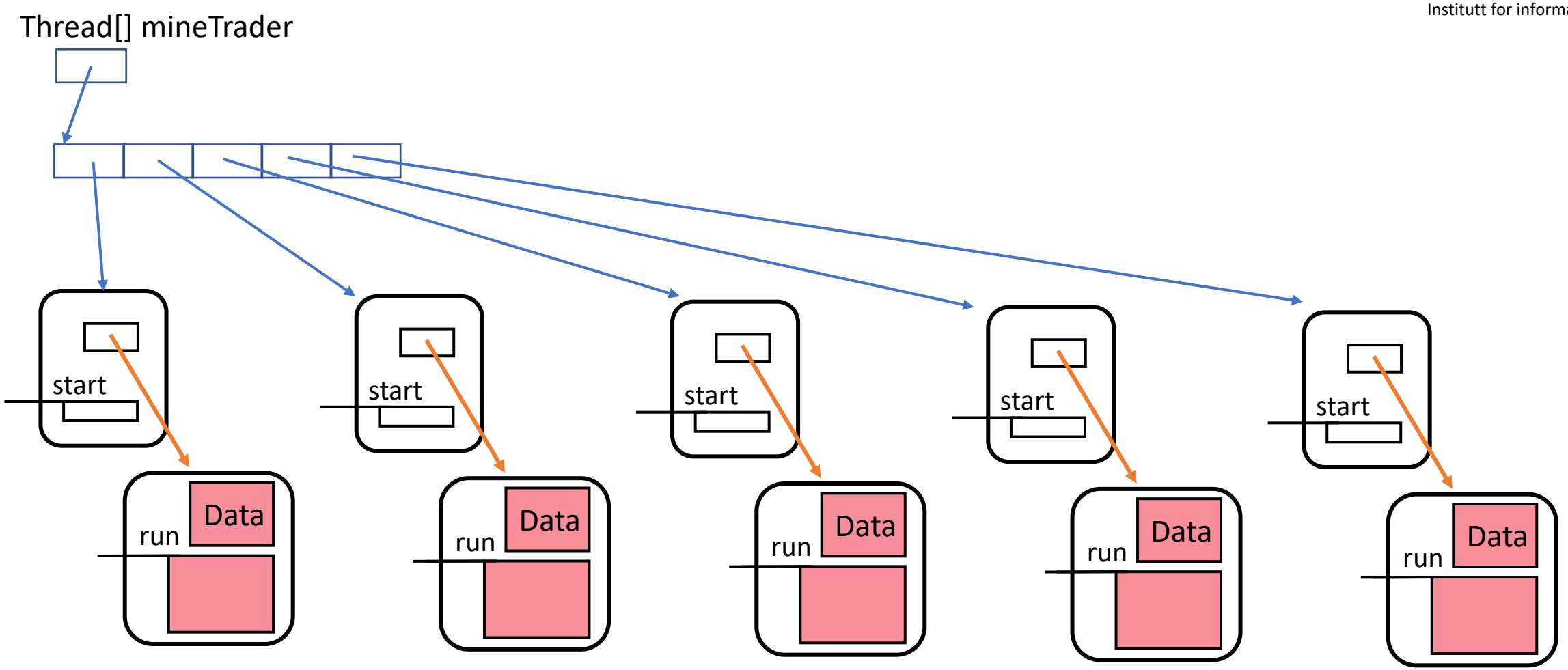


Eksempel:
main()-tråden starter 5 tråder





Noen ganger er det greit å ha referanser til alle trådene



Sove

- En tråd kan sove et antall milli- sekunder; metode i klassen Thread:
  - static void **sleep** (long millis)
    ”Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.”

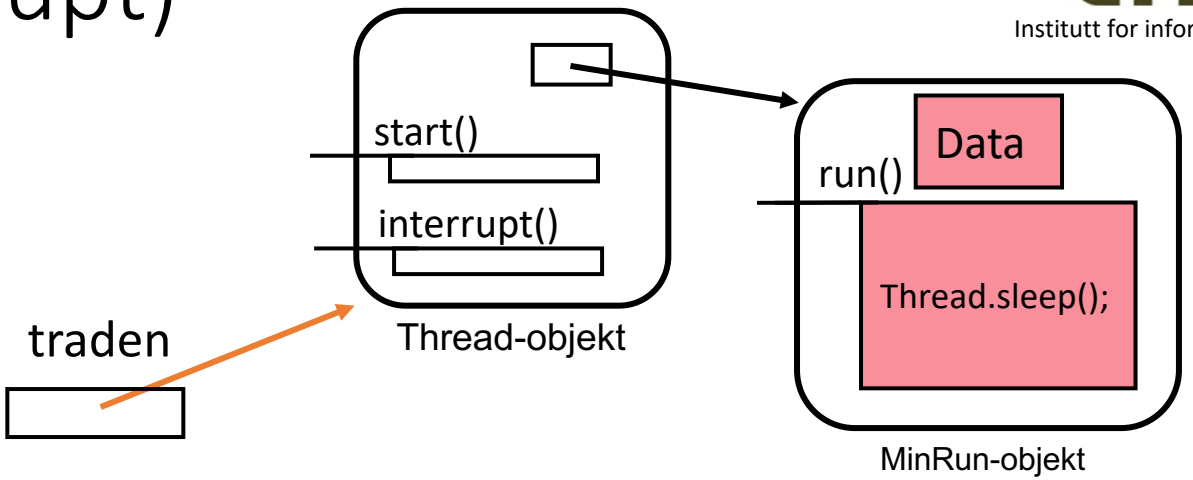
- Hvis noen avbryter denne tråden mens den sover skjer et unntak

```
try { Thread.sleep(1000); }  // sover ett sekund
     catch (InterruptedException e) { behandler avbrudd }

// Fortsetter her både etter avbrudd og når ferdig sovet
```

Avbrudd (interrupt)

- En tråd kan bli avrutt
  - void **interrupt** ()
    ” Interrupts this thread ”




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

traden.interrupt();
```

Her kaller programmet metoden interrupt() i Thread-objektet som variablen traden peker på.
Dette fører til at den tråden som dette Thread-objektet representerer blir avbrutt.

Hvis tråden ligger og venter / sover, vil den våkne opp av et InterruptedException (eksempel kommer nå)

En stoppeklokke

```java
import java.util.Scanner;


class TradKlokke {
    public static void main (String [] args) {
        Scanner minInn = new Scanner (System.in);
        Runnable minRun = new RunStoppekl();
        Thread traden  = new Thread(minRun);
        System.out.println(" Stoppeklokke");
        System.out.println(
               " Tast CR for å stoppe og starte");
        minInn.nextLine();
        traden.start();
        minInn.nextLine();
        traden.interrupt();
        System.out.println(" Takk for naa  "   );
    }
}
```

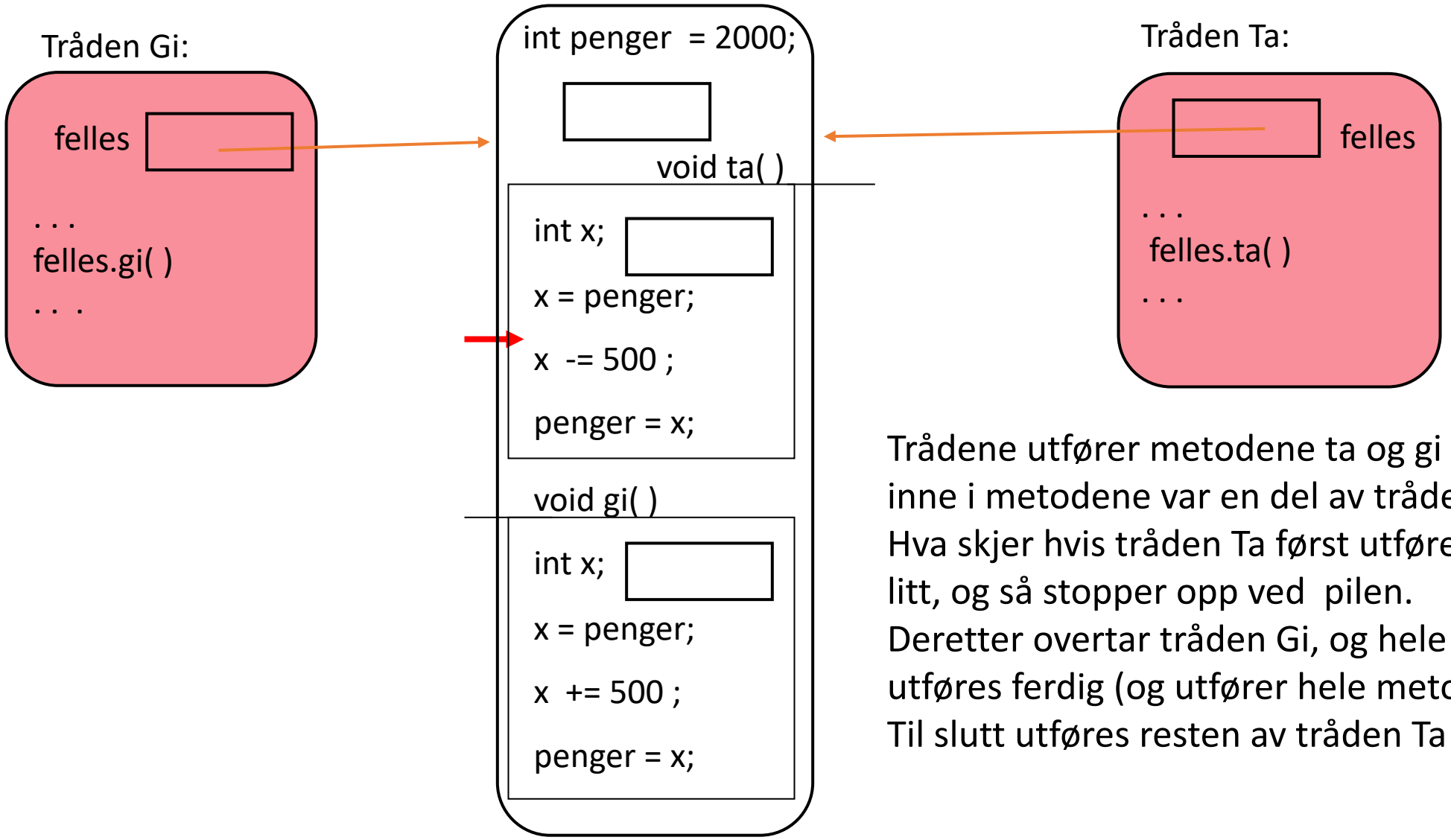
```java
class RunStoppekl implements Runnable {
    public void run() {
        int klokka = 0;
        try {
            while (true) {
                Thread.sleep(1000);
                System.out.println(klokka);
                klokka ++;
            }
        }
        catch (InterruptedException i) {
            System.out.println(
                    " Klokka er ferdig");
        }
    }
}
```

Samarbeid mellom tråder / Felles data / Oppdateringsproblemet

Om å passe på at tråder samarbeider riktig
Horstmann kap 20.3

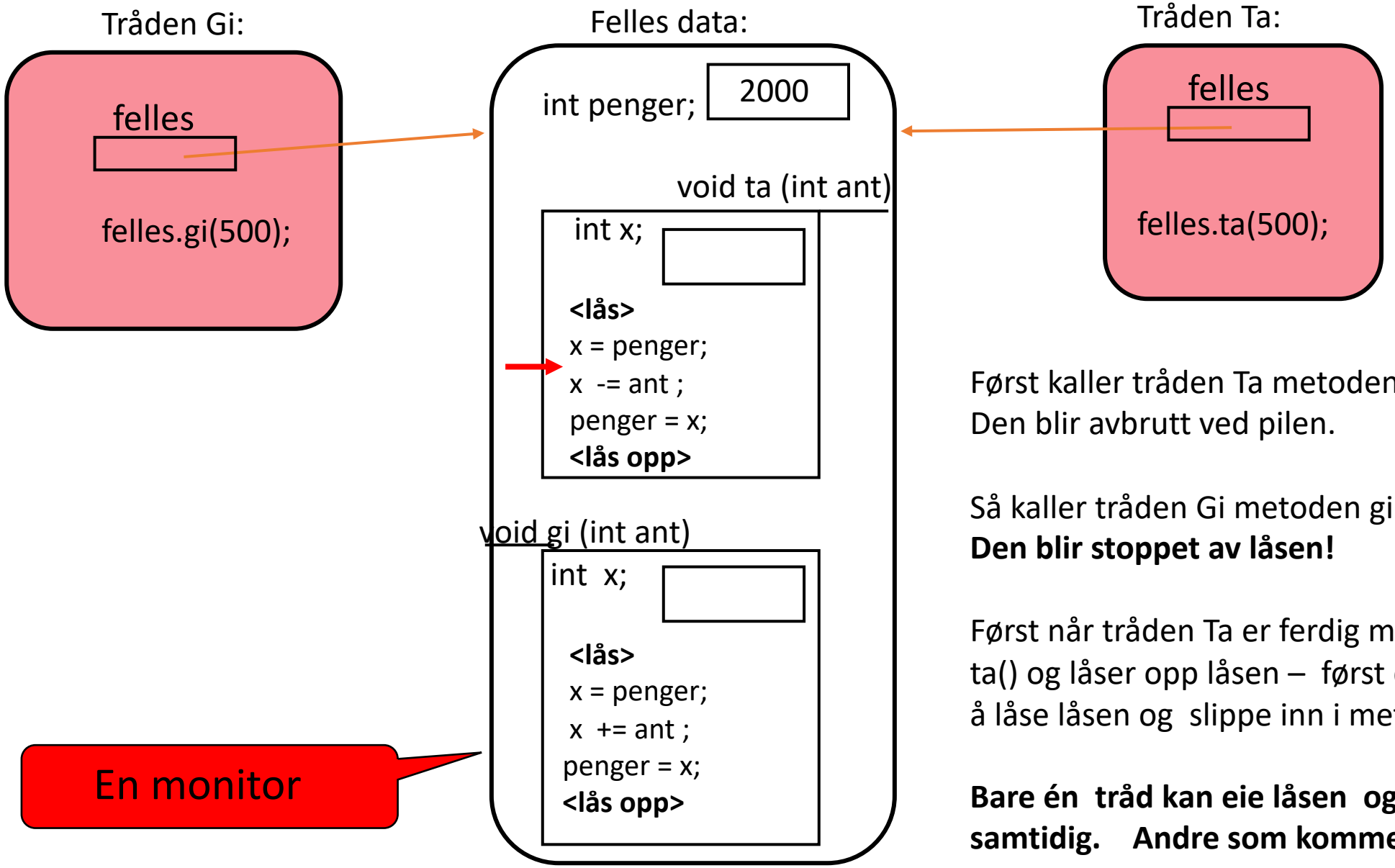
- **Samtidig oppdatering av felles data**
- Løsningen: Kritiske regioner

Standardeksempelet på at to tråder kan ødelegge felles data



Trådene utfører metodene ta og gi som om koden inne i metodene var en del av trådenes kode!
Hva skjer hvis tråden Ta først utfører metoden ta litt, og så stopper opp ved pilen.
Deretter overtar tråden Gi, og hele denne tråden utføres ferdig (og utfører hele metoden gi).
Til slutt utføres resten av tråden Ta (metoden ta).

Vi ordner dette med **kritiske regioner**.



Først kaller tråden Ta metoden ta().
Den blir avbrutt ved pilen.

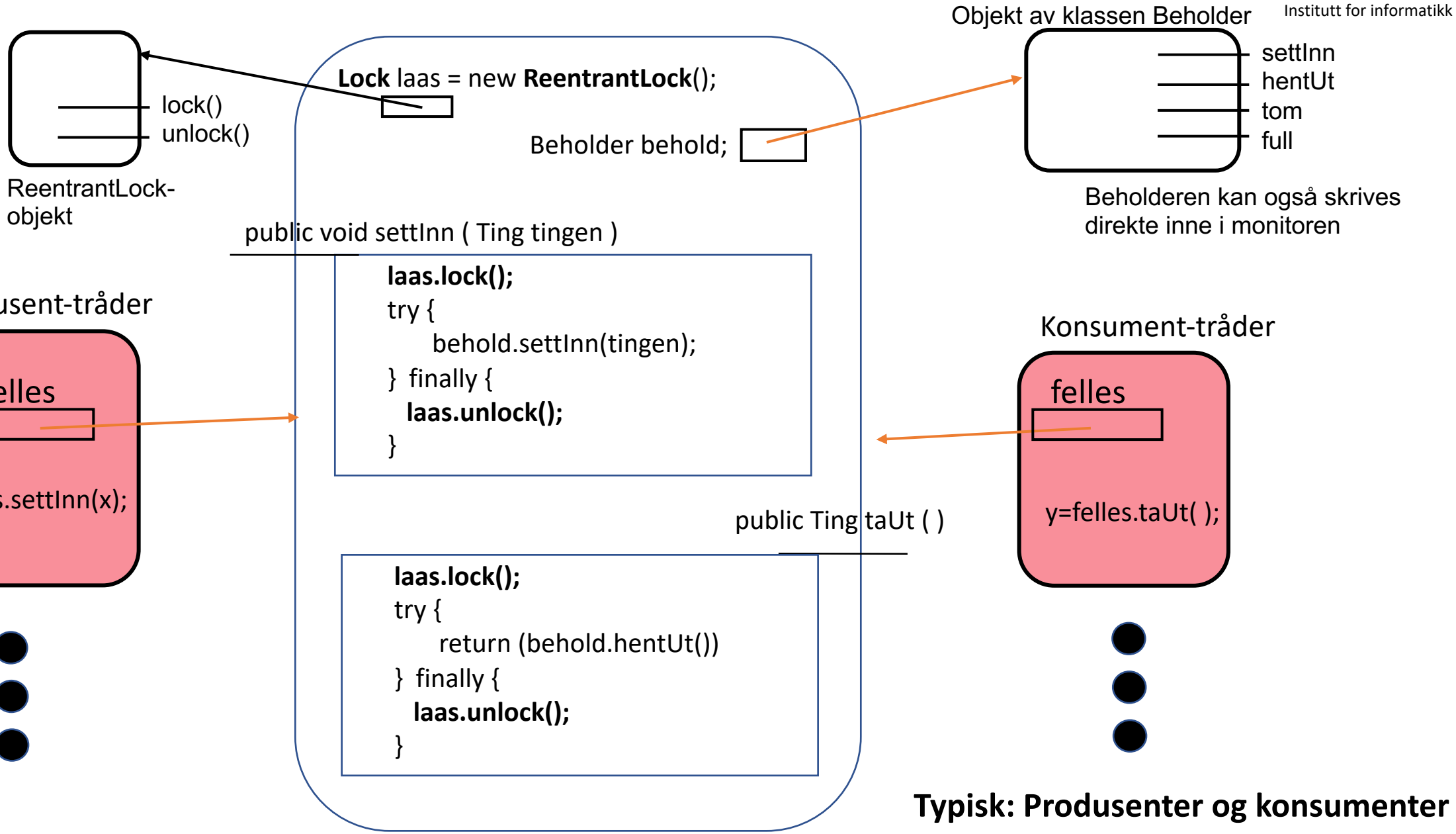
Så kaller tråden Gi metoden gi().
Den blir stoppet av låsen!

Først når tråden Ta er ferdig med å utføre metoden ta() og låser opp låsen – først da får tråden Gi lov å låse låsen og slippe inn i metoden gi().

Bare én tråd kan eie låsen og de felles dataene samtidig. Andre som kommer må vente.

Flyfabrikker og flyselskaper

En monitor som er/beskytter en beholder



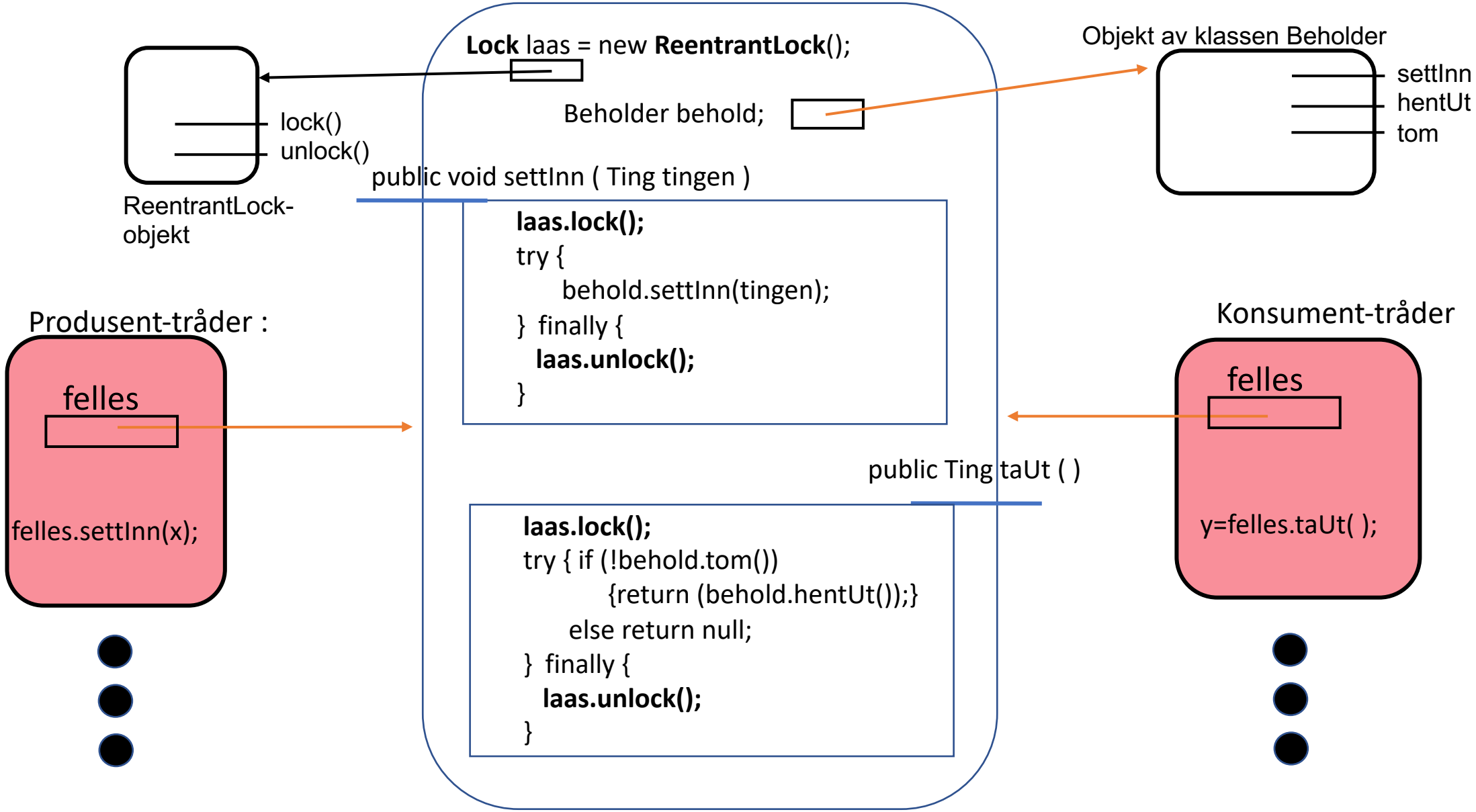
Typisk: Produsenter og konsumenter

Legg merke til bruken av finally


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

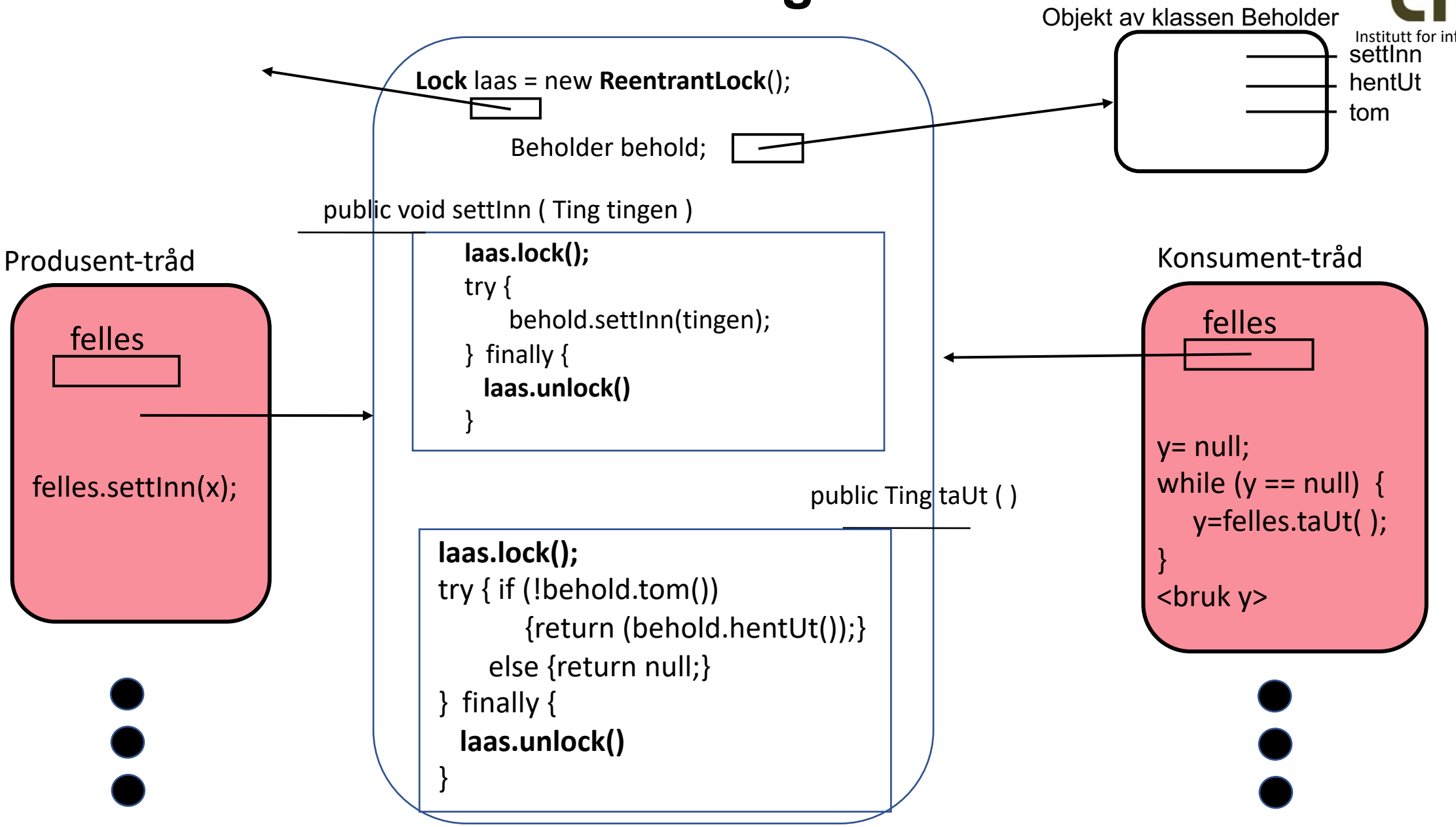
Da blir laas.unlock() **alltid** utført !!

Prøver å ta ut, men beholderen er tom



- Men kanskje det at beholderen er tom er veldig midlertid
- Kanskje kan vi vente på et en annen tråd legger inn noe i beholderen

Aktiv venting



Passiv venting er bedre enn aktiv venting

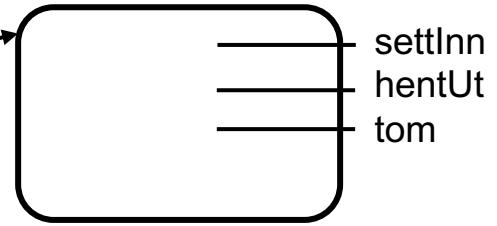
- Aktiv venting er vanligvis ikke smart
- Det bruker ressurser unødvendig
  - CPU-kraft
  - Monitoren blokkeres hver eneste gang tråden går inn i monitoren for å teste
- Bruk aktiv venting bare i spesielle tilfeller:
  - Få som kunkurrerer og korte kritiske regioner
  - Ødleger (blokkerer) ikke for andre og kan ikke bruke prosessorkapasiteten til noe annet

Passiv venting

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

Beholder behold;

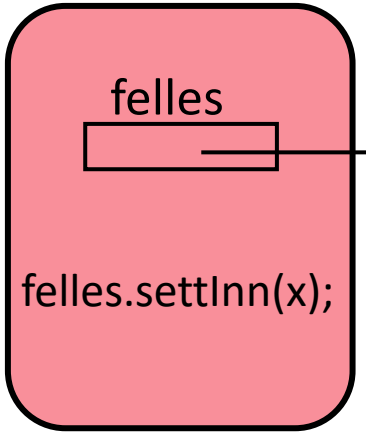
Objekt av klassen Beholder



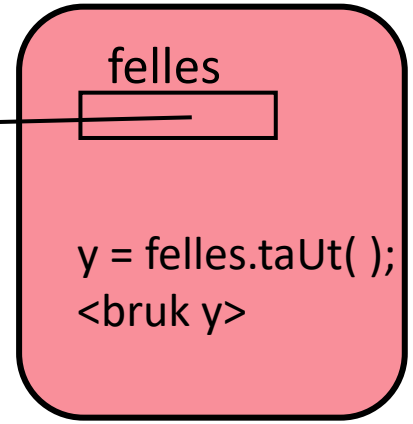
public void settInn (Ting tingen)

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

Produsent-tråder



Konsument-tråder

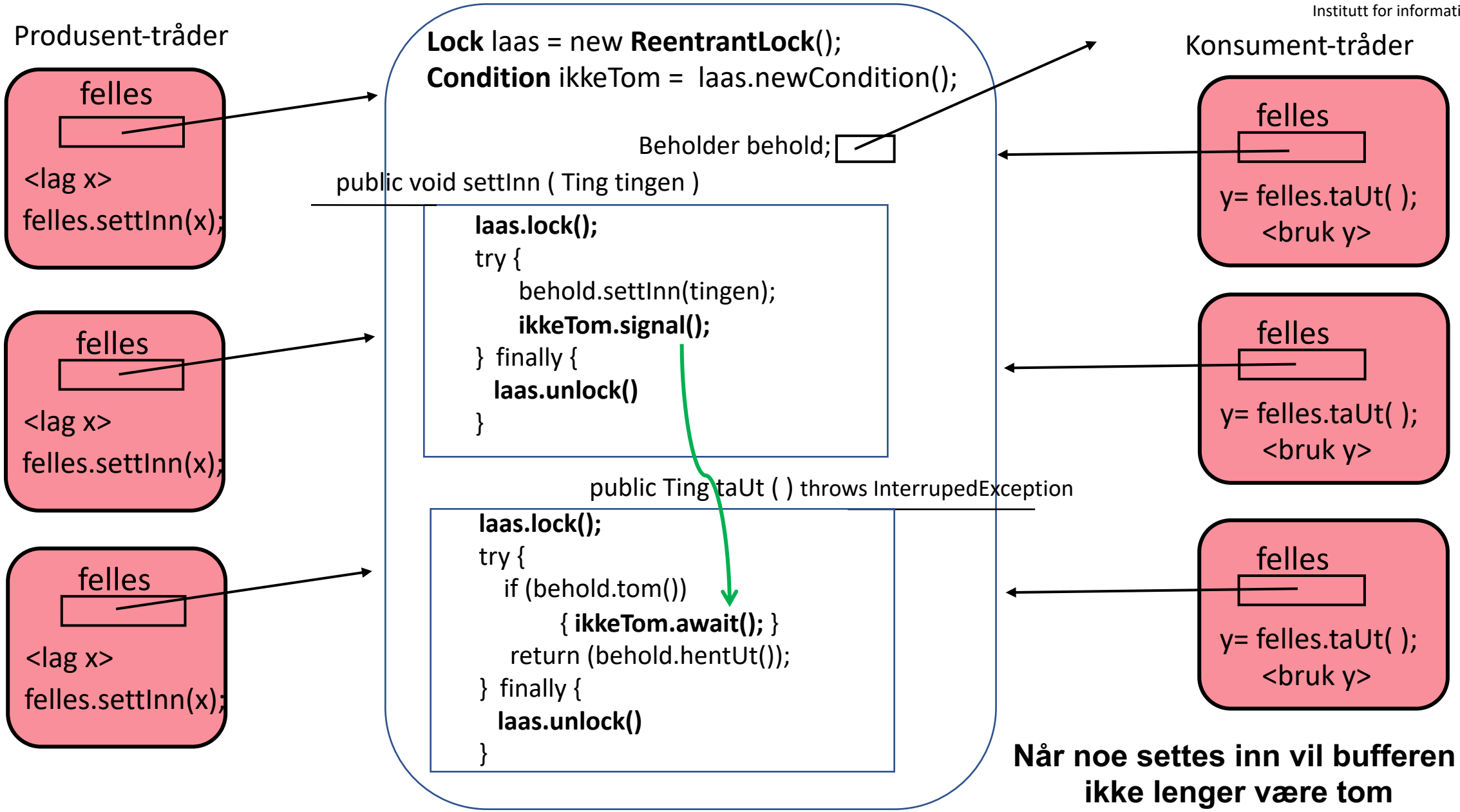


public Ting taUt ( ) throws InterrupedException

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

Men hvem skal vekke opp tråden som venter ?

Passiv venting og signalering



Når noe settes inn vil bufferen ikke lenger være tom

Passiv venting, køer

Beholder behold;

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

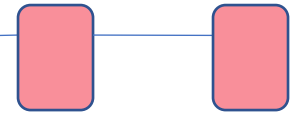
Condition ikkeTom = laas.newCondition();

void settInn (Ting tingen)

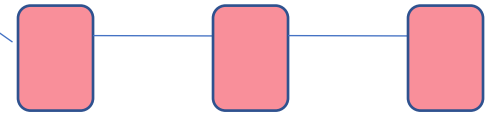
```
laas.lock();
try {
    behold.settInn(tingen);
    // det er lagt inn noe, så det er helt sikkert ikke tomt:
  ikkeTom.signal();
} finally {
  laas.unlock();
}
```

Tingen taUt ( ) throws InterrupedException

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



Tråder som venter på låsen for å få komme inn første gang



Tråder som venter på at bufferet ikke skal være tomt

En monitor (et objekt)

Beholder behold;

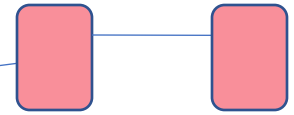
Lock laas = new **ReentrantLock**();
Condition ikkeFull = laas.newCondition();
Condition ikkeTom = laas.newCondition();

void settInn (Ting tingen) throws InterrupedException

```
laas.lock();
try {
   if (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.signal();
} finally {
  laas.unlock();
}
```

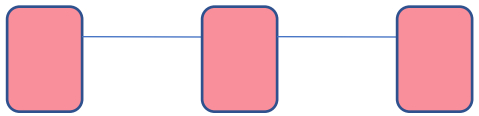
Tingen taUt ( ) throws InterrupedException

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



Tråder som venter på låsen for å få komme inn første gang

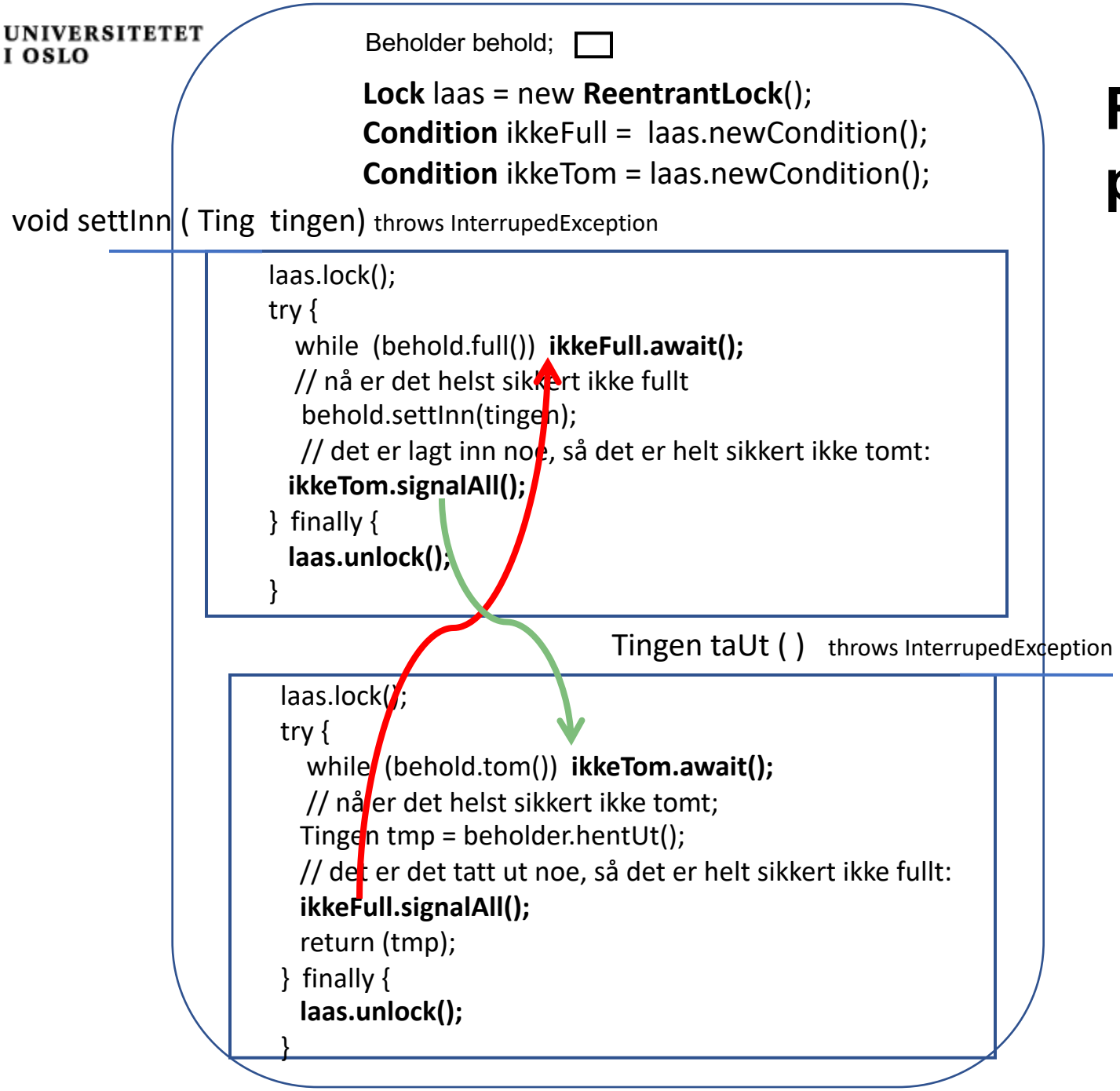
Tråder som venter på at bufferet ikke skal være fullt



Tråder som venter på at bufferet ikke skal være tomt

Monitor med metoder som venter både på tom og full buffer

Regler for robust programmering



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.

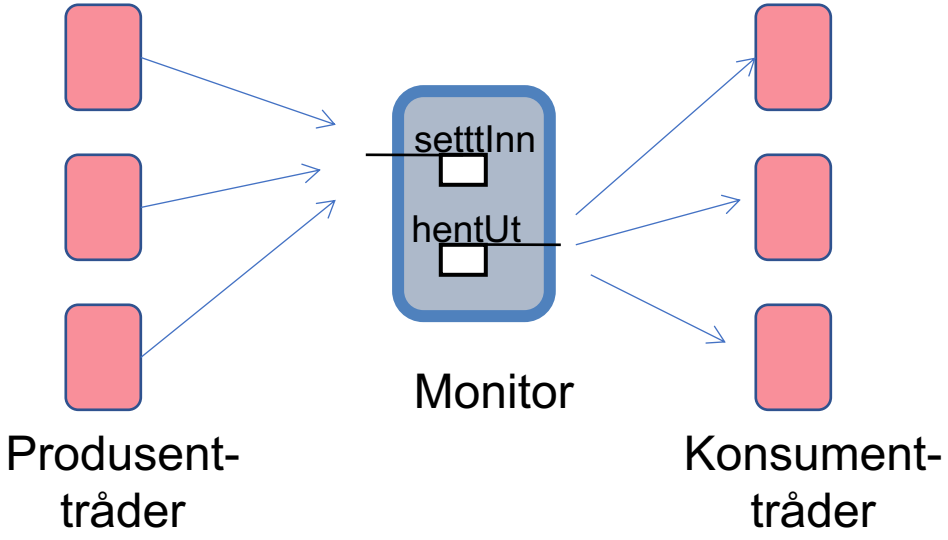
Se notatet om tråder.

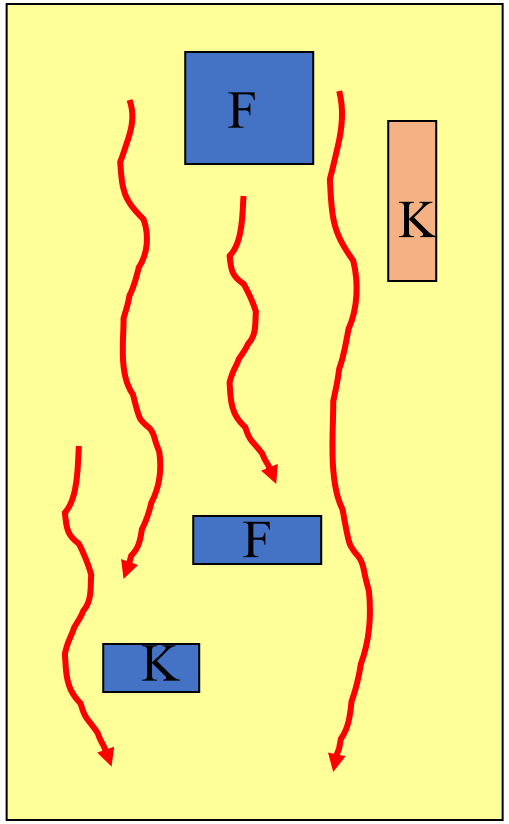
Hoved "take-away" i dag

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

Tråder deler adresserom. Felles data (blå felt, F) må vanligvis bare aksesseres (lese eller skrives i) av en tråd om gangen. Hvis ikke blir det kluss i dataene.

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





K: konstante data (immutable)
Konstante data kan leses uten å bruke kritiske regioner

Oppsummering
Vi har lært:

- Hvorfor og hvordan vi lager parallelle programmer (i Java)
- Hvordan tråder kommuniserer seg imellom ved hjelp av monitorer
- Hvordan programmer venter i en monitor
  - og starter opp igjen de som venter
- At det er utfordrende å programmere parallelle aktiviteter
  - Viktig å resonere om programmets tilstand
  - Umulig å teste nok tilfeller
    - Tidsavhengige feil kan vise seg først etter mange år.