UNIVERSITETET I OSLO

Institutt for informatikk

Foreløpig og lang 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

Hva er parallelle programmer - 1 ?

- På en datamaskin kjører mange programmer samtidig:
- **Virkelig samtidig**: To programmer får på samme tidspunkt utført hver sin instruksjon (må da ha flere kjerner/CPU-er/prosessorer)
  - Engelsk: Parallel computing
- **Tilsynelatende samtidig**: Maskinen (kjernen/CPU-en/prosessoren) skifter så raskt mellom programmene at du ikke merker det (flere ganger i sekundet).
  Maskinen utfører da noen få millioner instruksjoner for hvert program før den skynder seg til neste program.
  - Engelsk: Concurrent computing: Programmet/programmene kjører ekte eller bare tilsynelatende samtidig.
- På norsk har vi vanligvis ikke dette skille mellom ”parallel” og ”concurrent” – vi kaller alt parallelle eller samtidig programmer.
  Bruk: Virkelig/Ekte vs. Tilsynelatende parallellitet
- Uansett om programmene går virkelig eller tilsynelatende samtidig, må de behandles som om de går virkelig samtidig (fordi man ikke kan forutsi når maskinen skifter fra et program til et annet)

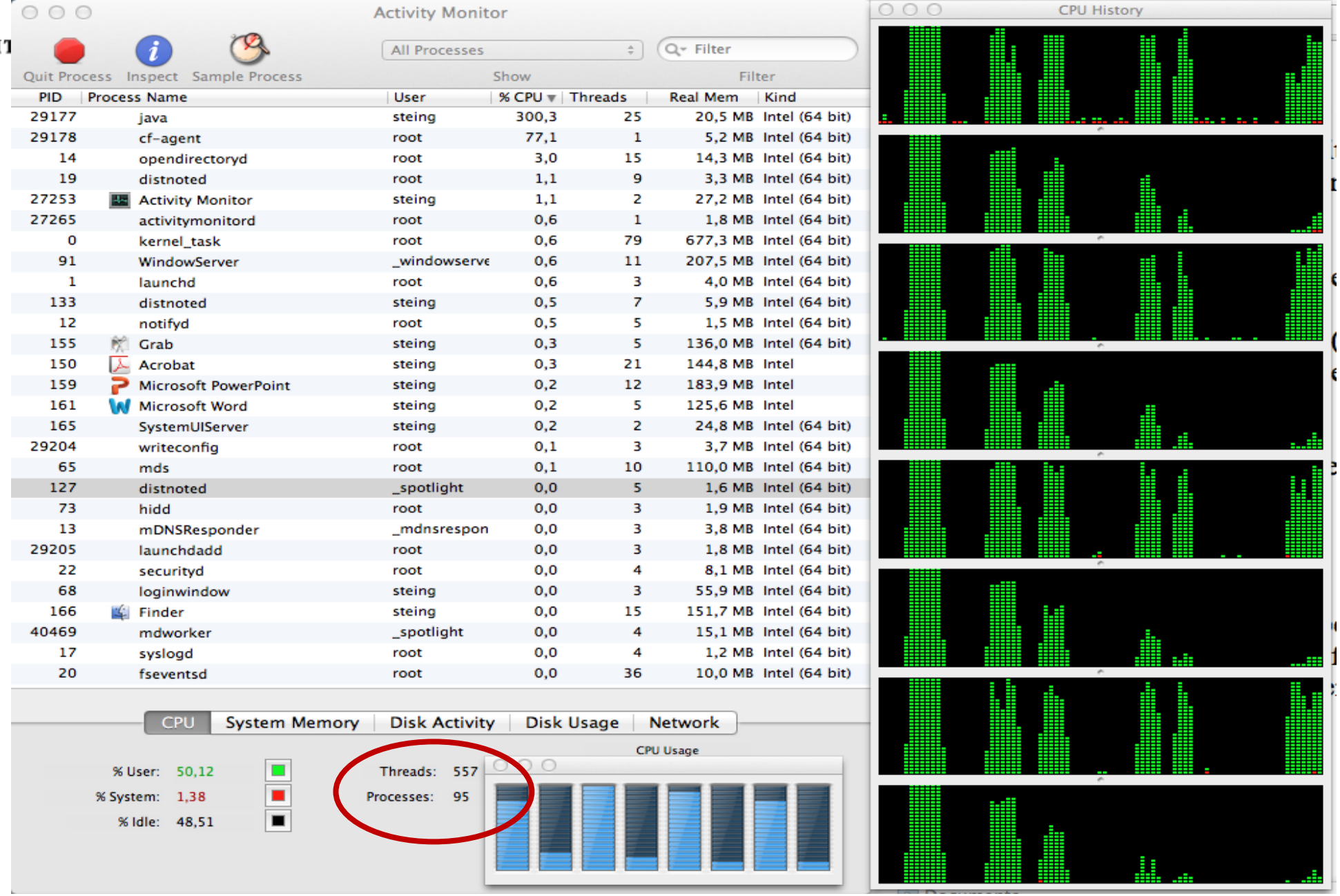
Hva er parallelle programmer – 2 ? (IN 1020)

- Administrasjonen av hvilke program som kjører, og hvordan de deler tiden og kjernene mellom seg, gjøres av **operativsystemet** (Unix, Windows, Mac-OS). Operativsystemet er et meget stort program og er det første programmet som startes i maskinen. Det har alle tillatelser, men skrur mange av disse tillatelsene av for vanlige programmer (eks: skrive direkte til disken eller til alle steder i primærlageret/RAM).
- På alle (?) operativsystemer i dag kan *flere* brukere være pålogget og kjøre sine programmer samtidig og *flere* uavhengige programmer kan kjøre samtidig
  - Datamaskinen kan skifte rask mellom alle de ulike programmene som er i gang i datamaskinen
    - Eks: Du kjører både et Java-program, skriver et brev i Word, får en melding og ser en video
    - De fleste mobiltelefoner og nettbrett har flere kjerner
  - Unntak:
    - en del svært enkle mobiltelefoner, nettbrett ol. ??
    - Mikrokontrollere – enkle embedded systems (f.eks. enkle senorer, IoT)

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 på *samme data* (nesten) samtidig. F.eks. et bestillingssystem: en kino, en flyavgang,… på en tjenermaskin
  - Oppgaver som naturlig modelleres i flere parallelle deler
    - Spesialtilfeller: Samlebånd. Bredde

- **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

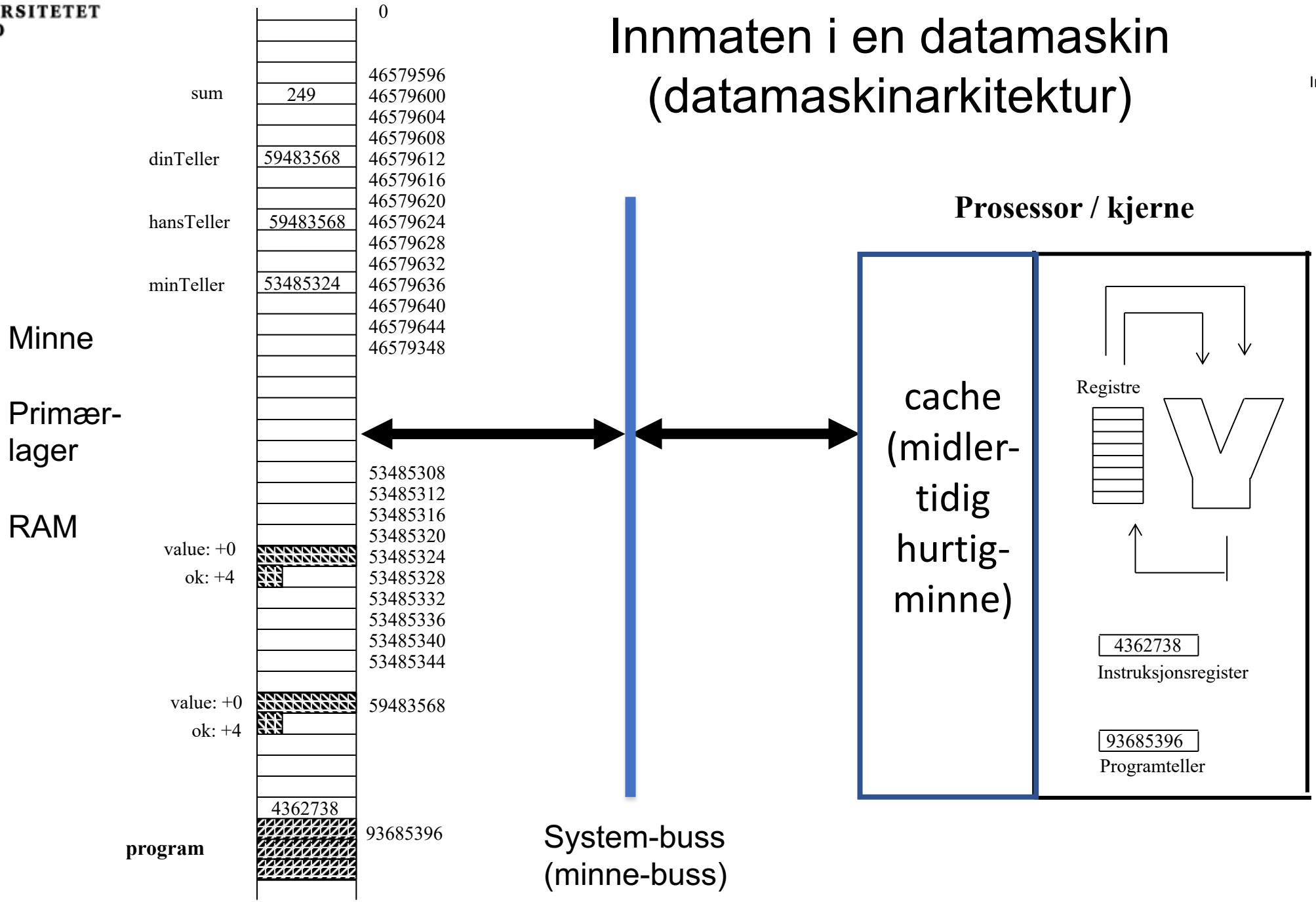


Maskin med 8 kjerner

UNIVERSITETET I OSLO

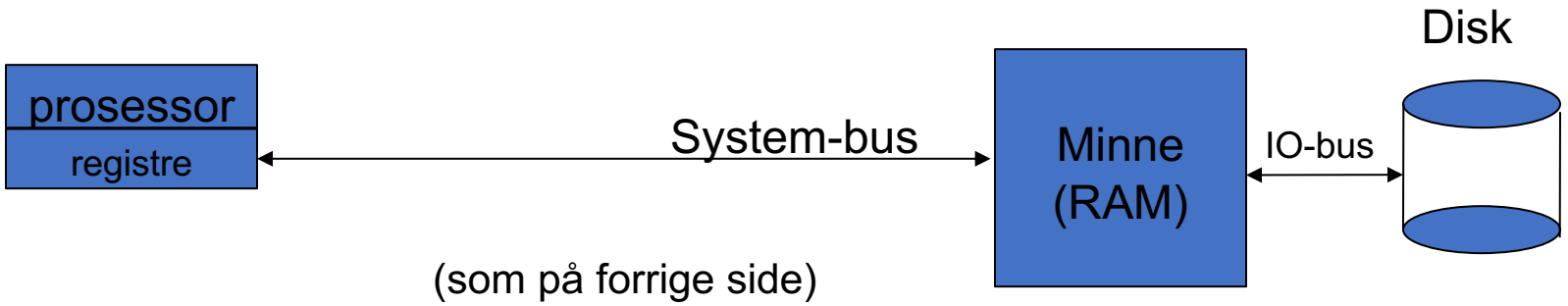
Innmaten i en datamaskin (datamaskinarkitektur)

Institutt for informatikk



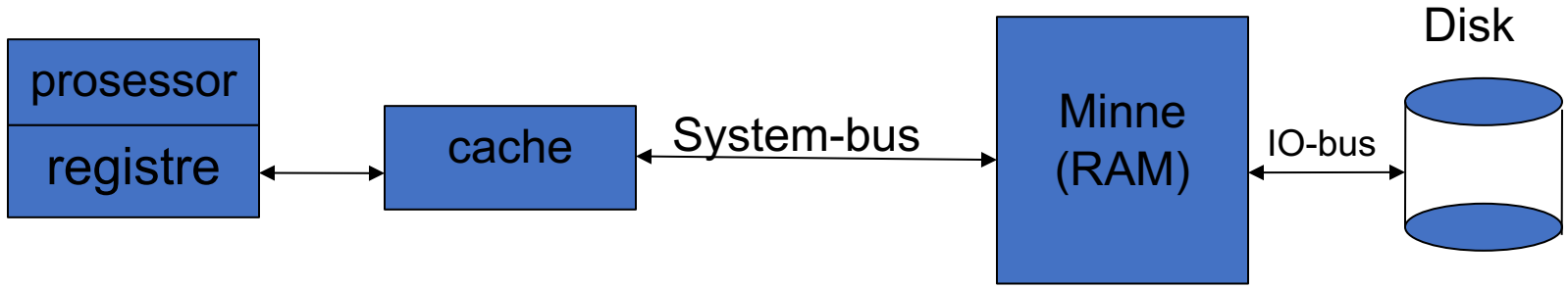
Datamaskinarkitektur (Computer Architecture)

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



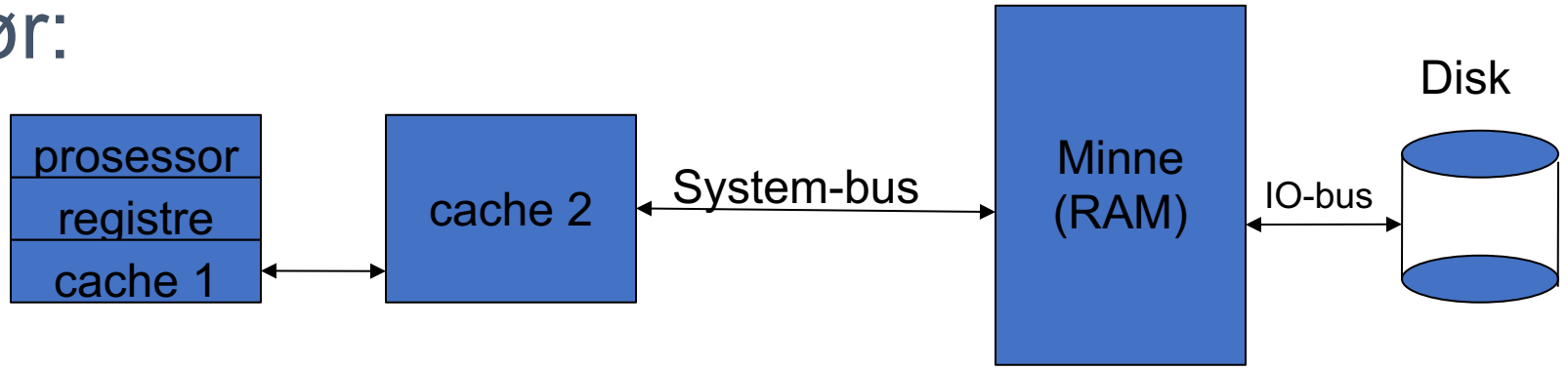
(som på forrige side)

2. Før:

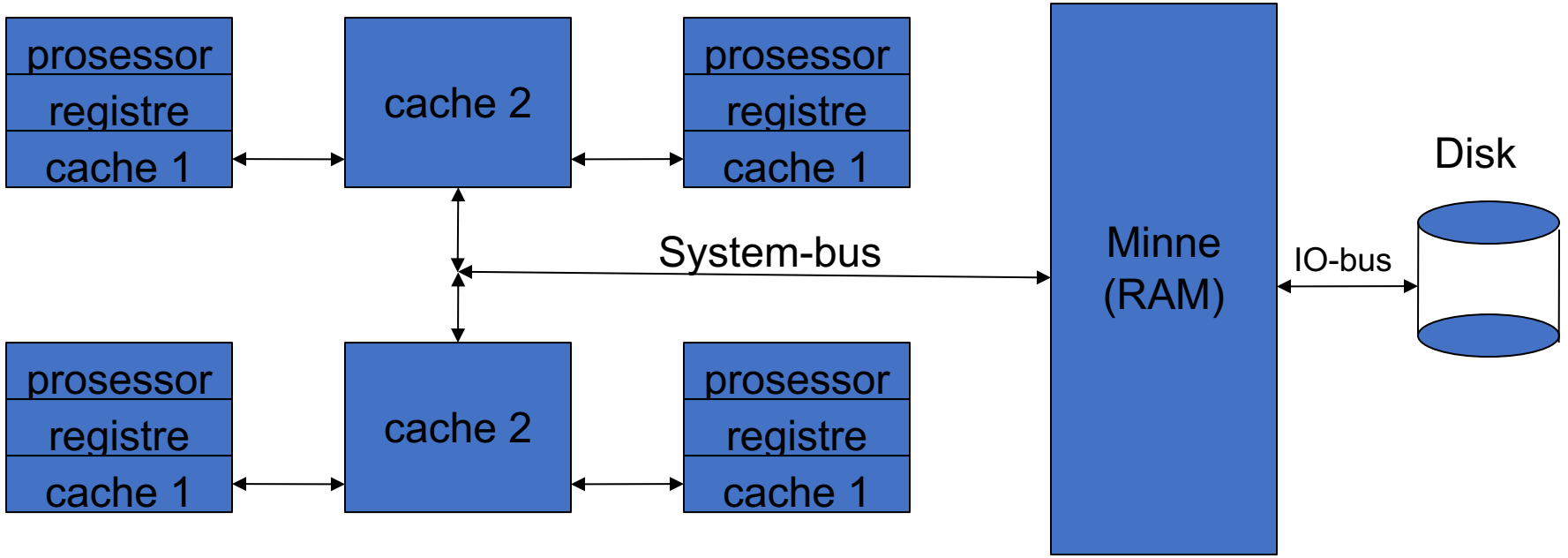


Datamaskinarkitektur (Computer Architecture)

3. Før:

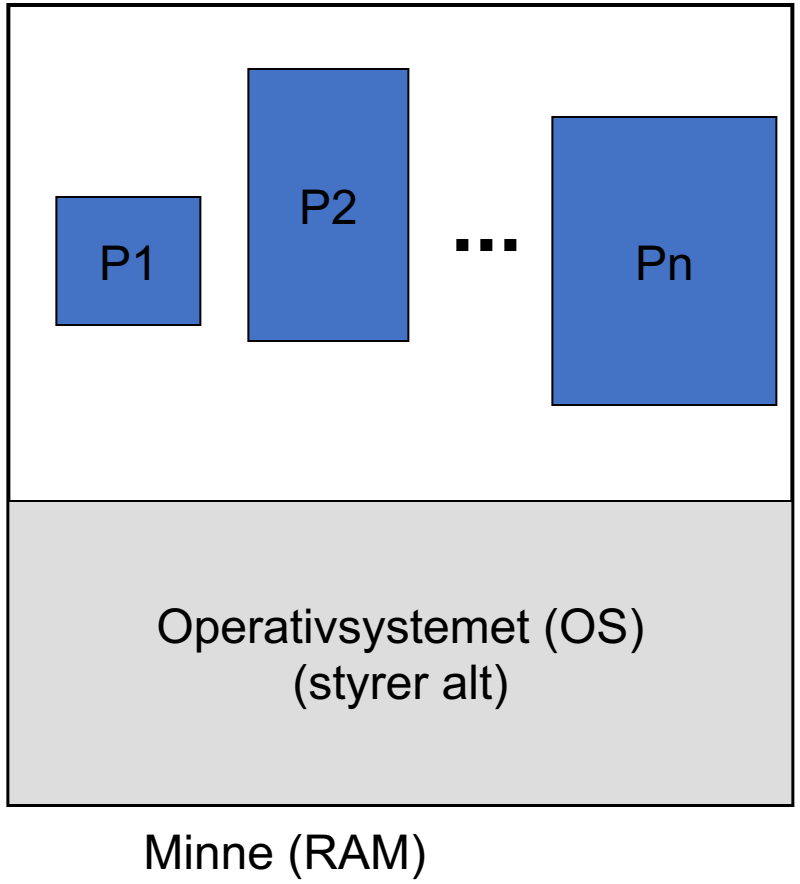


4. Nå, f.eks:



Parallellitet: prosesser og tråder

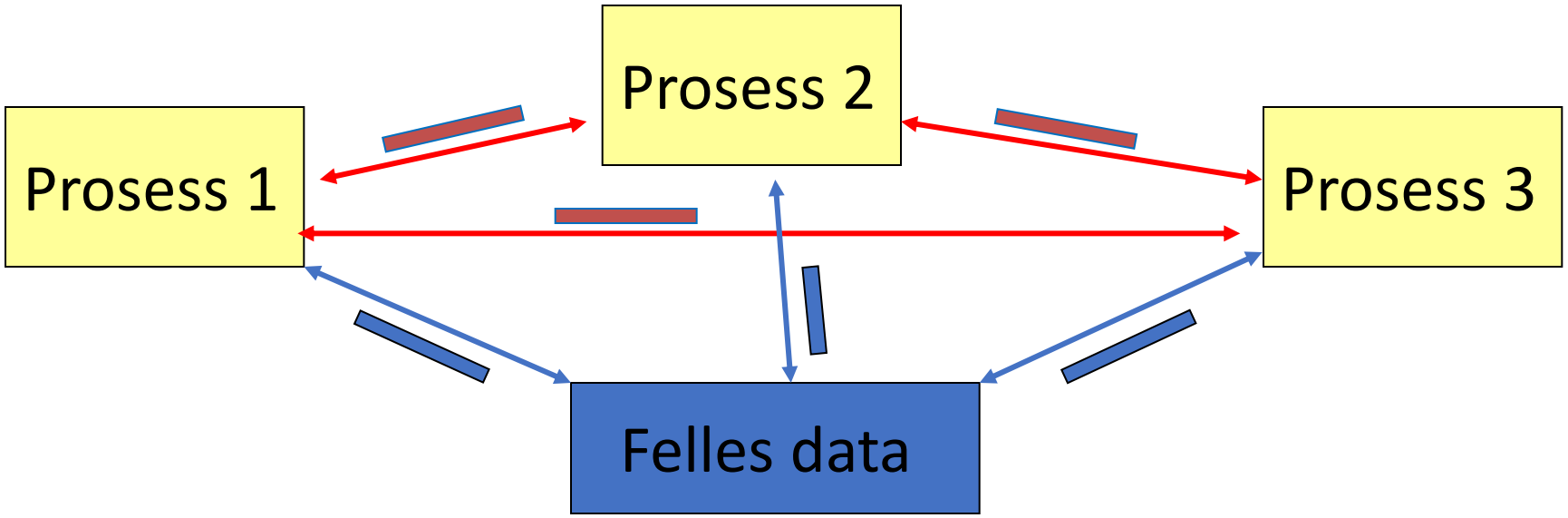
- Operativsystemet (OS) administrerer
  - Prosesser
  - (og delvis et antall tråder i hver prosess)
- Prosesser (P1, P2, ... , Pn)
  - En prosess er utføringen av et program
  - Er isolert fra hverandre, kan i utgangspunktet bare snakke til operativsystemet
  - Kan sende meldinger til andre prosesser 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.



Parallellprogrammering med prosesser

Parallellprogrammering vil si å løse en oppgave ved hjelp av programmer (eller programbiter) som utføres samtidig.

Samarbeidende prosesser sender meldinger til hverandre (røde piler)* **eller** leser og skriver i felles primærlager (blå piler) (men vanligvis ikke begge deler).



* Mest vanlig

Operativsystemets oppgave

(Anbefalt kurs: IN3000)

- Operativsystemet velger *hver gang* det kjører, blant de mange prosessene som er klare til å kjøre, hvilke som skal kjøres nå.
- ”Hver gang” betyr:
  - Det kommer noe data utenfra (tastatur, disk, nettverk, . . .)
  - Den innebygde klokka tikker (50 ganger i sekundet)
  - En prosess er (midlertidig) ferdig

  Da ’vekkes’ operativsystemet opp og overtar kontrollen fra (avbryter) den prosessen/ de prosessene som kjørte. Må huske hva de avbrutte prosessene var i ferd med å gjøre.
- Så bestemmer operativsystemet ’rettferdig’ hvilke prosesser som nå skal overta og kjøre litt:
  - De med høyest prioritet må slippe til oftest og mest
  - Ingen må vente ’alt for lenge’
  - Dersom ingen prosess ønsker å kjøre, har operativsystemet en egen tomgangsprosess som kan kjøre (går bare rundt i en tom løkke).

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 tråder er mye vanskeligere å skrive og teste/feilsøke enn bare én tråd i én prosess (som vi har gjort til nå)

Hva bruker vi tråder til ?

- Dele oppgavene i en prosess opp i naturlige del-oppgaver
- Alle slags tunge beregninger
  - Kunstig intelligens - maskinlæring - big data
  - Tolking og vising av video (animasjon) , tolking av tale, biler, ...
- Simuleringsmodeller: Etterape prosesser i virkeligheten for å finne ut statistikk/utfalet for f.eks. køer på et motorvei, variasjoner i biologiske systemer (simulere 100 000 laks fra fødsel til død) , datanettverk, hva skjer med et olje eller gassreservoar når oljen/gassen pumpes ut, krigføring (slag med tanks og fly),.... (men kvasiparallellitet er også vanlig (og enklere))
- Med flere prosessorer kan programmer med tråder gå raskere (”Dual/Quad/8/16/ …. -core”-prosessorer).
  - Også mulig med grafiske prosessorer
- Om å programmere med tråder og prosesser:
  - Prøv å finne det som naturlig er parallelt og uavhengig i oppgaven
  - Lag tråder eller prosesser som kommuniserer minst mulig med hverandre.

Tankemodell for tråder



```
public class Restaurant {
    int antBestilt;
    int antLaget = 0, antServert = 0; // tallerkenrertter
    int antKokker = 5, antServitører = 50;

    Restaurant(int ant) {
                    antBestilt = ant;
                    for (int i = 0; i < antKokker; i++) {
                        Kokk k = new Kokk(this, "Kokk nr. " + i);
                        k.start();
                    }

                    for (int i = 0; i < antServitører; i++) {
                        Servitor s = new Servitor(this, "S
                        s.start();
                    }

    }
    public static void main(String[] args) {
                    new Restaurant(Integer.parseInt(args[0]));
    }

    synchronized boolean kokkFerdig() {
                    return antLaget == antBestilt;
    }

    synchronized boolean servitørFerdig() {
                    return antServert == antBestilt;
    }

    synchronized boolean putTallerken(Kokk k) {
                    // Kokketråden blir eier av låsen.

                    while (antLaget - antServert > 2) {
                        /* så lenge det er minst 2 tallerkner
                         * som ikke er servert, skal kokken vente. */
                        try {
                                        wait(); /* Kokketråden gir fra seg
                                                 * låsen og sover til den
                                                 * blir vekket */
                        } catch (InterruptedException e) {}
                        // Kokketråden blir igjen eier av låsen
                    }

                    boolean ferdig = kokkFerdig();
                    if (!ferdig) {
                        antLaget++;
                        System.out.println(k.getName() + " laget nr: " + antLaget);
                    }

                    notify(); /* Si ifra til servitøren. */
                    return !ferdig;
    }

    synchronized boolean getTallerken(Servitor s) {
                    // Servitørtråden blir eier av låsen.

                    while (antLaget == antServert && !servitørFerdig()) {
                        /* så lenge kokken ikke har plassert
                         * en ny tallerken. Dermed skal
                         * servitøren vente. */
                        try {
                                        wait(); /* Servitørtråden gir fra seg
                                                 * låsen og sover til den
                                                 * blir vekket */
                        } catch(InterruptedException e) {}
                        // Servitørtråden blir igjen eier av låsen.
                    }

                    boolean ferdig = servitørFerdig();
                    if (!ferdig) {
                        antServert++;
                        System.out.println(s.getName() + " serverer nr: " + antServert);
                    }


                    notify(); /* si ifra til kokken */
                    return !ferdig;
    }
}

class Kokk extends Thread {
    Restaurant rest;

    Kokk(Restaurant rest, String navn) {
                    super(navn); // Denne tråden heter nå <navn>
                    this.rest = rest;
    }

    public void run() {
                    while (rest.putTallerken(this)) {
                        // levert tallerken.

                        try {
                                        sleep((long) (1000 * Math.random()));
                        } catch (InterruptedException e) {}
                    }
                    // Kokken er ferdig
    }
}

class Servitor extends Thread {
    Restaurant rest;

    Servitor(Restaurant rest, String navn) {
                    super(navn); // Denne tråden heter nå <navn>

    }

    public voi

                                    sleep((long) (1000 * Math.random()));
                    } catch (InterruptedException e) {}
                    }
                    // Servitøren er ferdig
```

Én tråd i programmet



```
public class Restaurant {
    int antBestilt;
    int antLaget = 0, antServert = 0; // tallerkenrertter
    int antKokker = 5, antServitører = 50;

    Restaurant(int ant) {
                    antBestilt = ant;
                    for (int i = 0; i < antKokker; i++) {
                        Kokk k = new Kokk(this, "Kokk nr. " + i);
                        k.start();
                    }

                    for (int i = 0; i < antServitører; i++) {
                        Servitor s = new Servitor(this, "Servitør nr. " + i);
                        s.start();
                    }


    public static void main(String[] args) {
                    new Restaurant(Integer.parseInt(args[0]));
    }
    synchronized boolean kokkFerdig() {
                    return antLaget == antBestilt;
    }

    synchronized boolean servitørFerdig() {
                    return antServert == antBestilt;
    }

    synchronized boolean putTallerken(Kokk k) {
                    // Kokketråden blir eier av låsen.

                    while (antLaget - antServert > 2) {
                        /* så lenge det er minst 2 tallerkner
                         * som ikke er servert, skal kokken vente. */
                        try {
                                        wait(); /* Kokketråden gir fra seg
                                                 * låsen og sover til den
                                                 * blir vekket */
                        } catch (InterruptedException e) {}
                        // Kokketråden blir igjen eier av låsen
                    }

                    boolean ferdig = kokkFerdig();
                    if (!ferdig) {
                        antLaget++;
                        System.out.println(k.getName() + " laget nr: " + antLaget);
                    }

                    notify(); /* Si ifra til servitøren. */
                    return !ferdig;
    }

    synchronized boolean getTallerken(Servitor s) {
                    // Servitørtråden blir eier av låsen.

                    while (antLaget == antServert && !servitørFerdig()) {
                        /* så lenge kokken ikke har plassert
                         * en ny tallerken. Dermed skal
                         * servitøren vente. */
                        try {
                                        wait(); /* Servitørtråden gir fra seg
                                                 * låsen og sover til den
                                                 * blir vekket */
                        } catch(InterruptedException e) {}
                        // Servitørtråden blir igjen eier av låsen.
                    }

                    boolean ferdig = servitørFerdig();
                    if (!ferdig) {
                        antServert++;
                        System.out.println(s.getName() + " serverer nr: " + antServert);
                    }


                    notify(); /* si ifra til kokken */
                    return !ferdig;
    }
}

class Kokk extends Thread {
    Restaurant rest;

    Kokk(Restaurant rest, String navn) {
                    super(navn); // Denne tråden heter nå <navn>
                    this.rest = rest;
    }
    public void run() {
                    while (rest.putTallerken(this)) {
                        // levert tallerken.

                        try {
                                        sleep((long) (1000 * Math.random()));
                        } catch (InterruptedException e) {}
                    }
                    // Kokken er ferdig
    }
}

class Servitor extends Thread {
    Restaurant rest;

    Servito

    }

    public void run() {
                    while (rest.getTallerken(this)) {
                        try {
                                        sleep((long) (1000 * Math.random()));
                        } catch (InterruptedException e) {}
                    }
                    // Servitøren er ferdig
```

To tråder i programmet

Tråder i Java

- Er innebygget i språket
- I ethvert program er det minst én tråd (den som starter og kjører **`main`**)
- GUI (Grafisk brukergrensesnitt) kjører en egen tråd (Event Dispatch Thread, EDT) (senere i IN1010)
- Fra en tråd kan vi starte flere andre, nye tråder
- En tråd starter som:
  - Lag et objekt av en klasse som også implementerer **interface Runnable** (kap 20.1 i Horstmann)
    - **run()** er eneste metode i grensesnittet
  - Gi dette objektet som parameter til klassen Thread, og kall start( ) på dette Thread-objektet
    - Komposisjon / Delegering: Thread-objektet delegerer til "Runnable"-klasse-objektet å utføre selve tråd-koden (i run-metoden).

Skjult lysark: Tråder i Java

- Er innebygget i språket
- I ethvert program er det minst én tråd (den som starter og kjører **`main`**)
- GUI kjører en egen tråd (Event Dispatch Thread, EDT) (senere i IN1010)
- Fra en tråd kan vi starte flere andre, nye tråder
- En tråd starter enten som:
  I. Lag et objekt av en klasse som også implementerer **interface Runnable** (kap 20.1 i Horstmann)
     - **run()** er eneste metode i grensesnittet
     - Gi dette objektet som parameter til klassen Thread, og kall start( ) på dette Thread-objektet
       - Litt mer komplisert enn metoden nedenfor.
       - Litt mer fleksibelt (kan da bytte ut objektet som inneholder run-metoden)

  II. Et objekt av en subklasse av **class Thread** som inneholder én polymorf metode **run()** som du skriver i subklassen (Horstmann: Programming tip 20.1).
     - Lag først et objekt av en subklasse til klassen Thread (new . . .)
     - Kall så metoden start() i objektet (ikke laget av deg , men arvet fra Thread).
     - start() sørger for at run(), som du selv har skrevet, vil bli kalt.
     - Thread implementerer grensesnittet Runnable (se over)

Tråder i Java

- Én viktig klasse og ett viktig interface

  class Thread { . . . }

  interface Runnable { . . . }

  Begge er definert i Java API

compact1, compact2, compact3
java.lang

Interface Runnable

All Known Subinterfaces:
RunnableFuture<V>, RunnableScheduledFuture<V>

All Known Implementing Classes:
AsyncBoxView.ChildState, ForkJoinWorkerThread, FutureTask, RenderableImageProducer, SwingWorker, Thread, TimerTask

Functional Interface:
This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

```
@FunctionalInterface
public interface Runnable
```

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called run.

This interface is designed to provide a common protocol for objects that wish to execute code while they are active. For example, Runnable is implemented by class Thread. Being active simply means that a thread has been started and has not yet been stopped.

In addition, Runnable provides the means for a class to be active while not subclassing Thread. A class that implements Runnable can run without subclassing Thread by instantiating a Thread instance and passing itself in as the target. In most cases, the Runnable interface should be used if you are only planning to override the run() method and no other Thread methods. This is important because classes should not be subclassed unless the programmer intends on modifying or enhancing the fundamental behavior of the class.

Since:
JDK1.0

See Also:
Thread, Callable

Method Summary

All Methods | Instance Methods | Abstract Methods

Modifier and Type	Method and Description
void	run() When an object implementing interface Runnable is used to create a thread, starting the thread causes the object's run method to be called in that separately executing thread.

Method Detail

run

```
void run()
```

When an object implementing interface Runnable is used to create a thread, starting the thread causes the object's run method to be called in that separately executing thread.

The general contract of the method run is that it may take any action whatsoever.

See Also:
Thread.run()

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
Java™ Platform Standard Ed. 8
PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Institutt for informatikk

compact1, compact2, compact3

java.lang

Class Thread

java.lang.Object
    java.lang.Thread

All Implemented Interfaces:

Runnable

Direct Known Subclasses:

ForkJoinWorkerThread

```
public class Thread
extends Object
implements Runnable
```

A *thread* is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

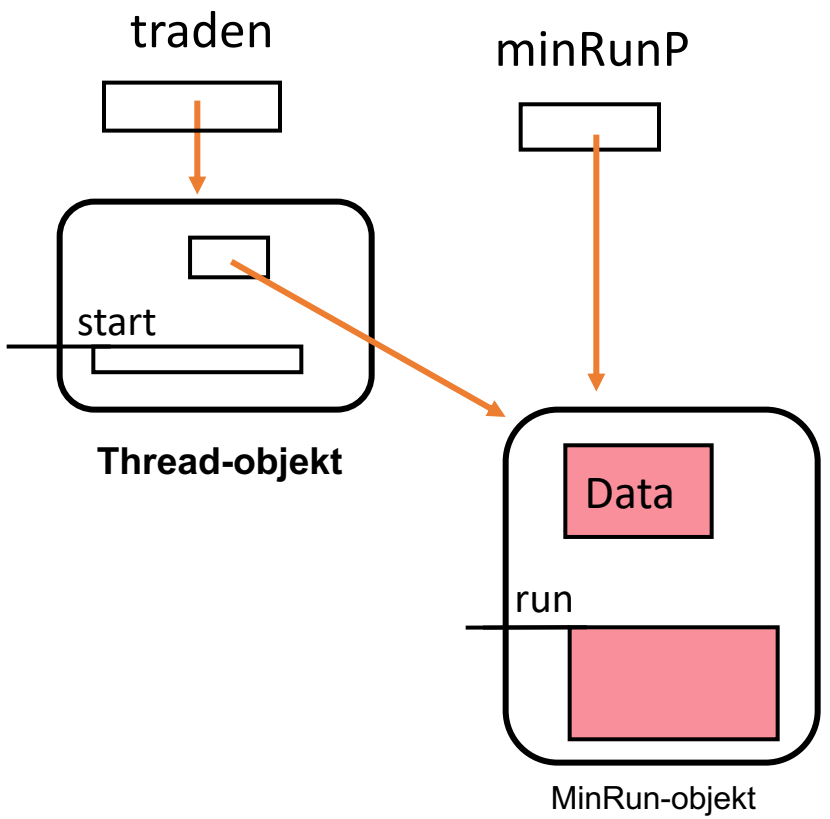
Tråder i Java


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

En tråd lages og startes opp slik:

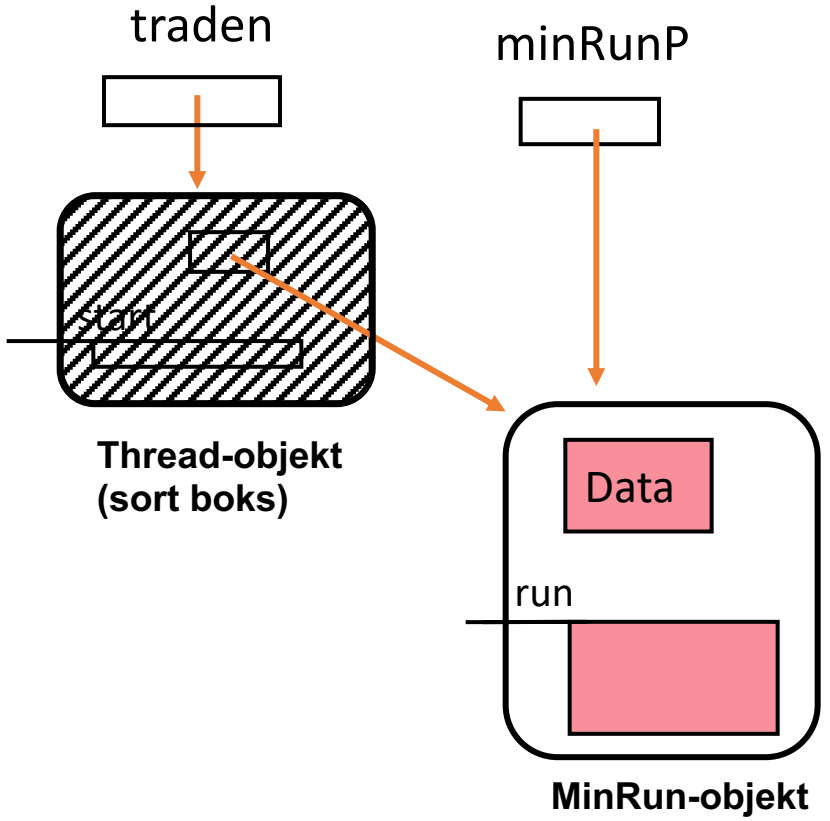
```
Runnable minRunP = new MinRun();
Thread traden = new Thread(minRunP);
traden.start( );
```





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).

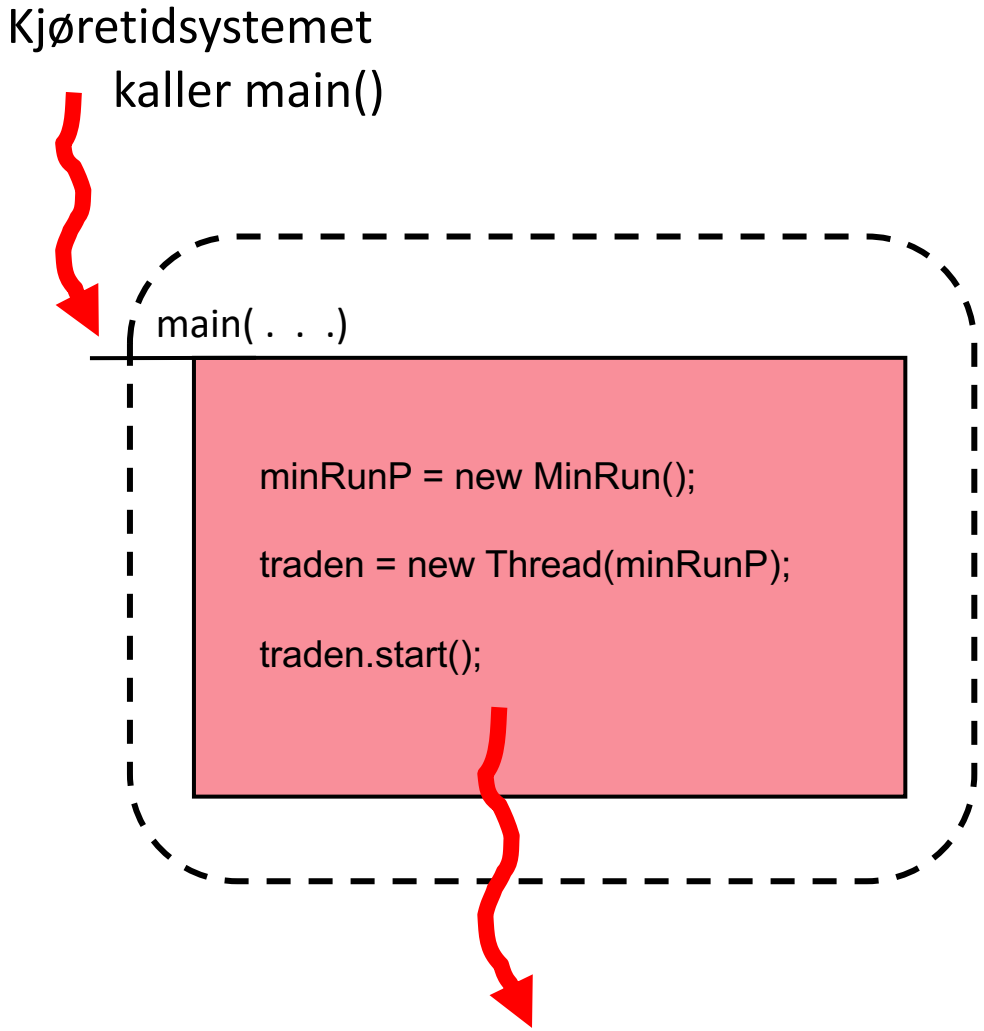
Tråder i Java

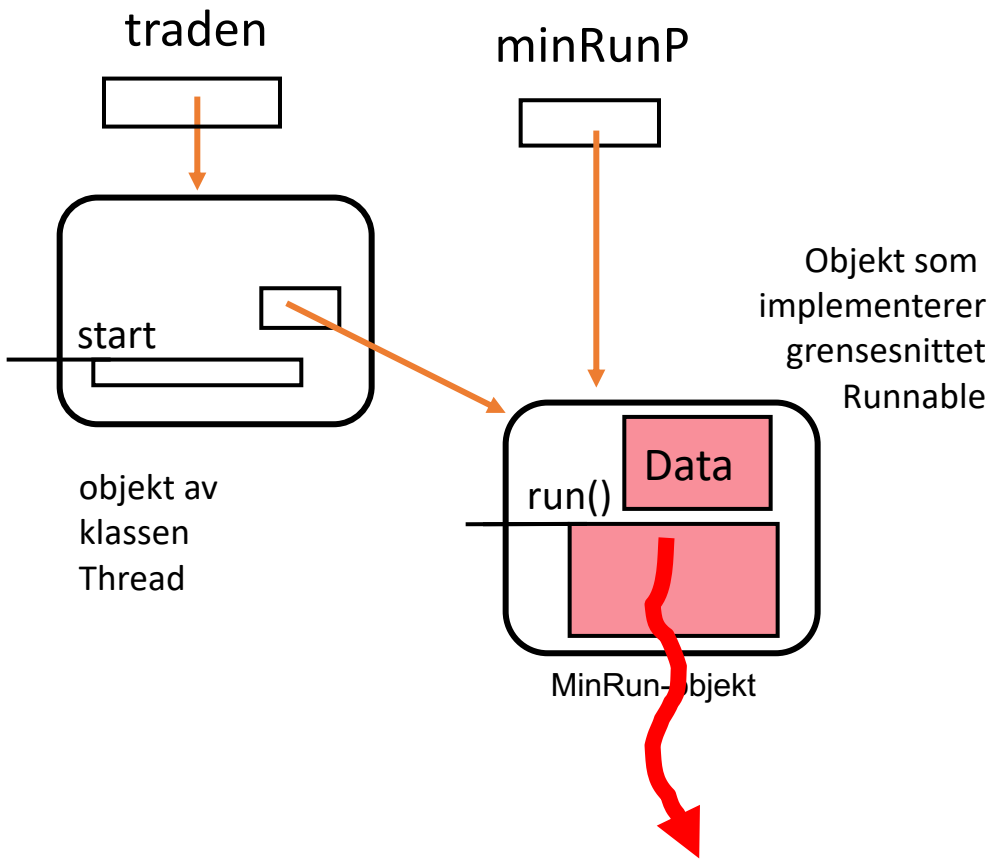


Delegering:
Thread-objektet delegerer til MinRun-objektet å utføre run()-metoden og med dette tråd-koden

```
Thread traden = new Thread(minRunP);
```

main()-tråden starter en ny tråd

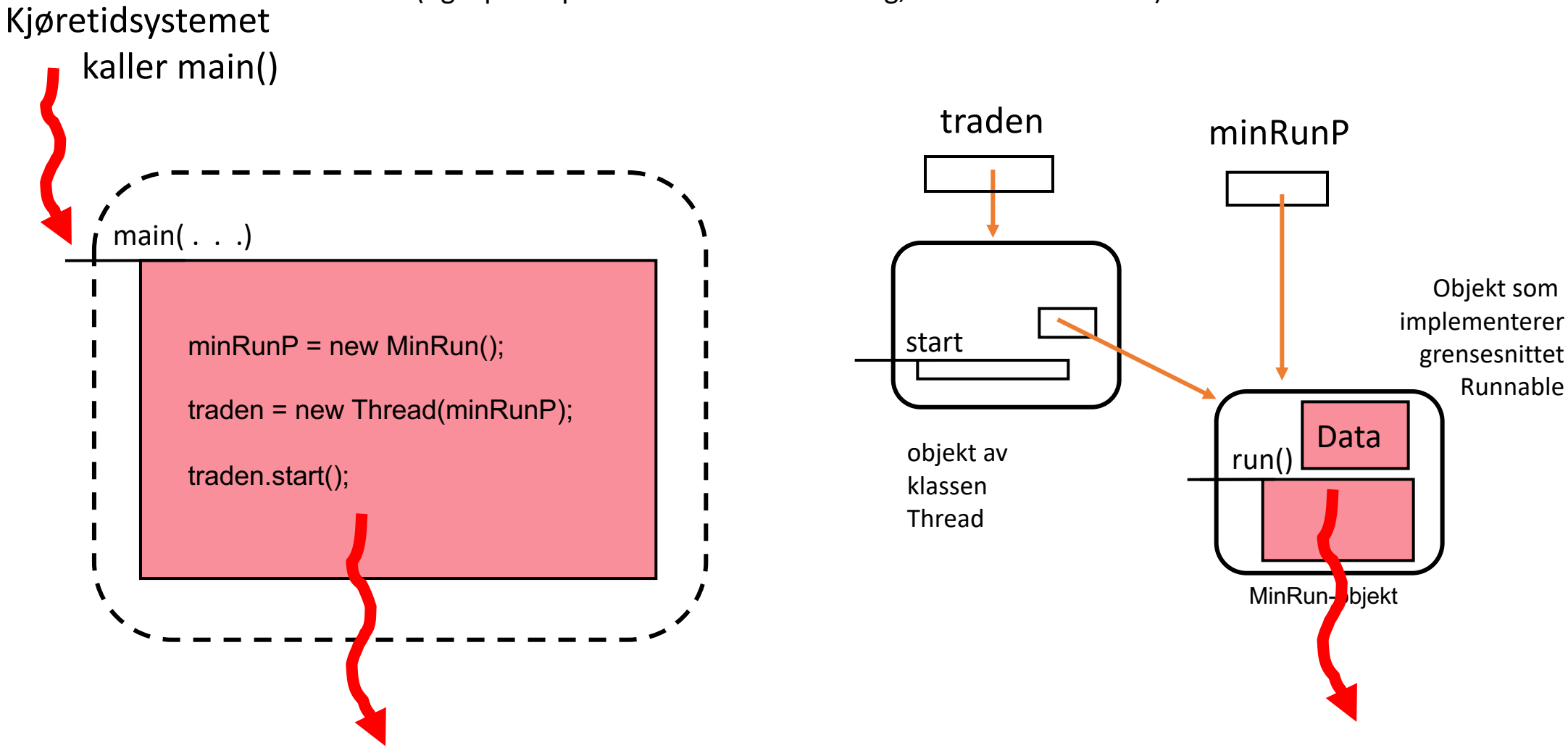




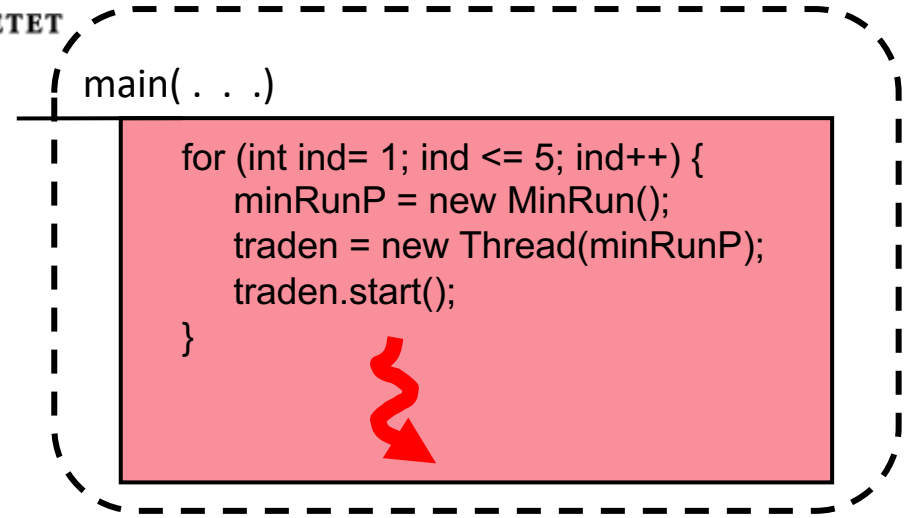
Det er det vi trenger å vite om tråder

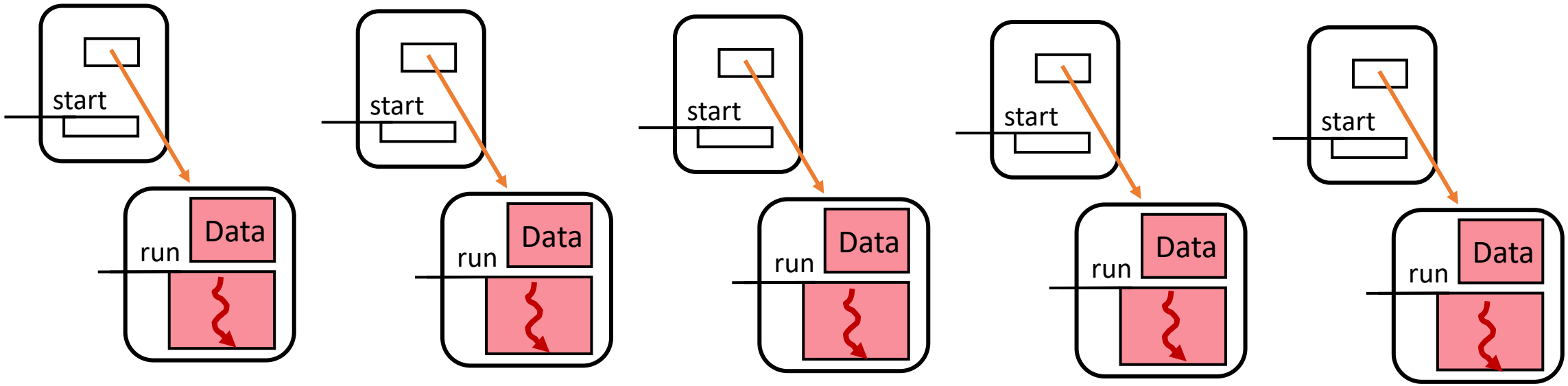
Nå er resten bare opp til fantasien ☺

(og å passe på at de samarbeider riktig, mer om det senere)



Slik kan programmet lage 5 tråder





Sove

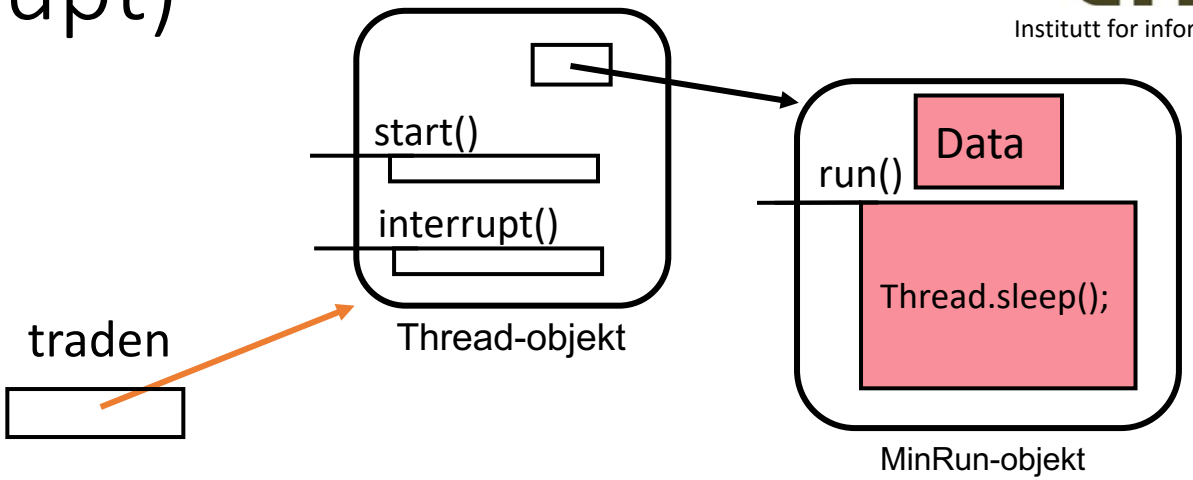
- En tråd kan sove et antall milli- sekunder; metode i klassen Thread:
  - static void **sleep** (long millis)
    ”Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.”
  - Andre tråder slipper da til og får kjøre
  - Hvis noen avbryter denne tråden mens den sover skjer et unntak (neste side)

```
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



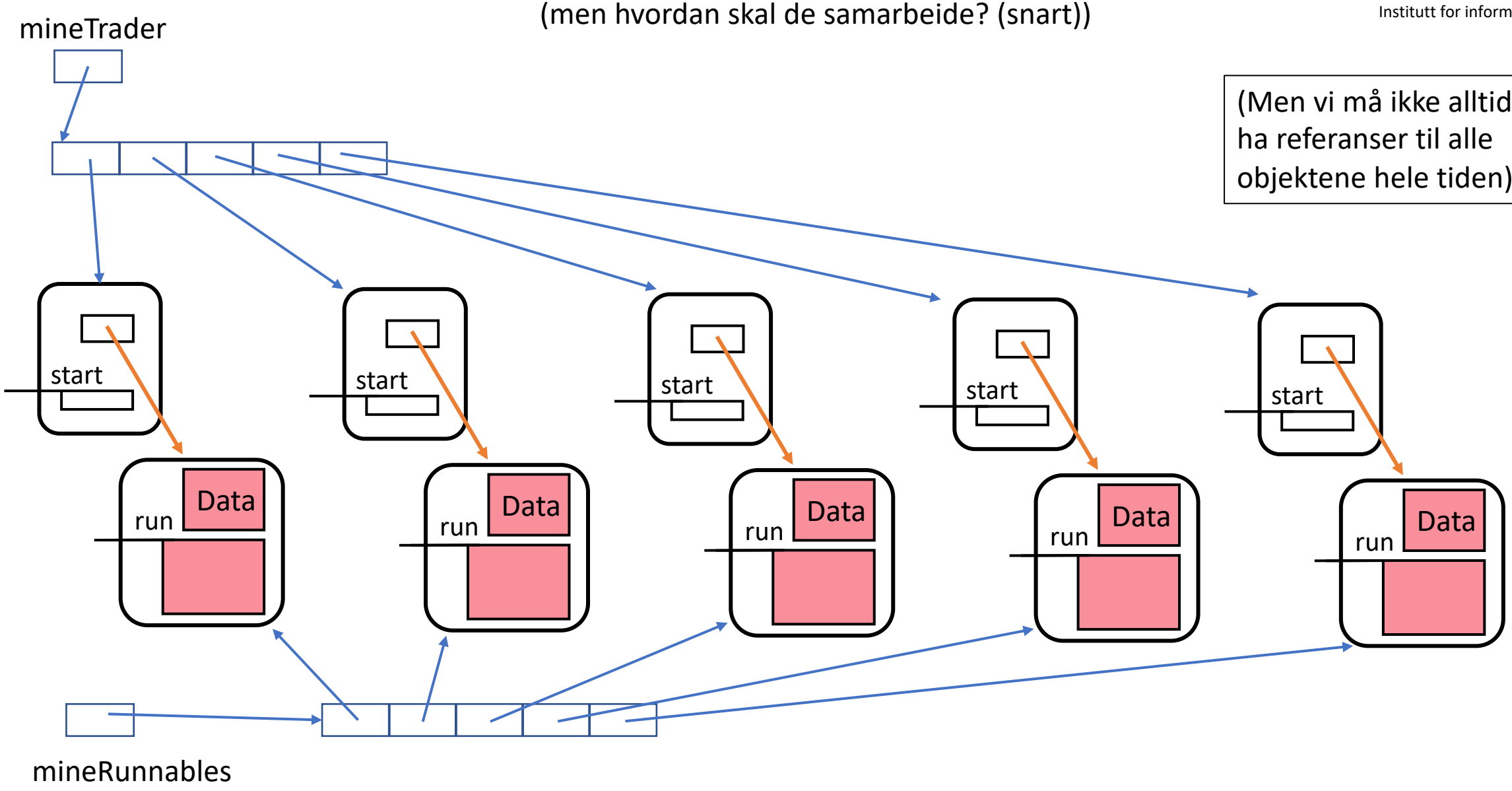
```java
import java.util.Scanner;


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");
        }
    }
}
```

Mer om f.eks. 5 tråder som samarbeider om å løse en oppgave

(men hvordan skal de samarbeide? (snart))



(Men vi må ikke alltid ha referanser til alle objektene hele tiden)

Hva skjer når run() kaller andre metoder

f.eks. to tråder:

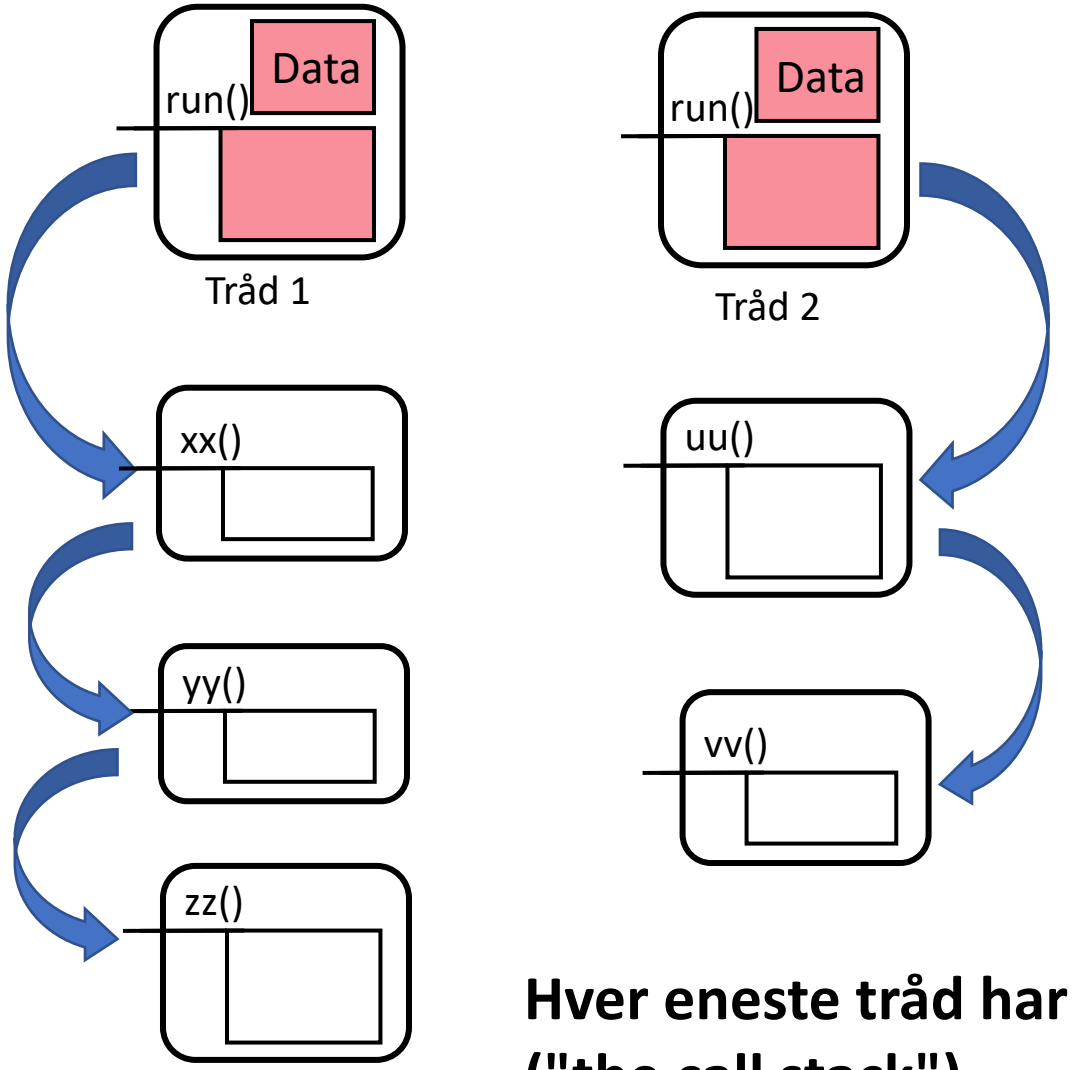
En stabel/stakk av metodekall:

run() i tråd 1 kaller

metoden xx() som kaller

metoden yy() som kaller

metoden zz() som . . .



En stabel/stakk av metodekall

run() i 2råd 2 kaller

metoden uu() som kaller

metoden vv() som . . .

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

Lokale variable i metoder (uavhengig av tråder)

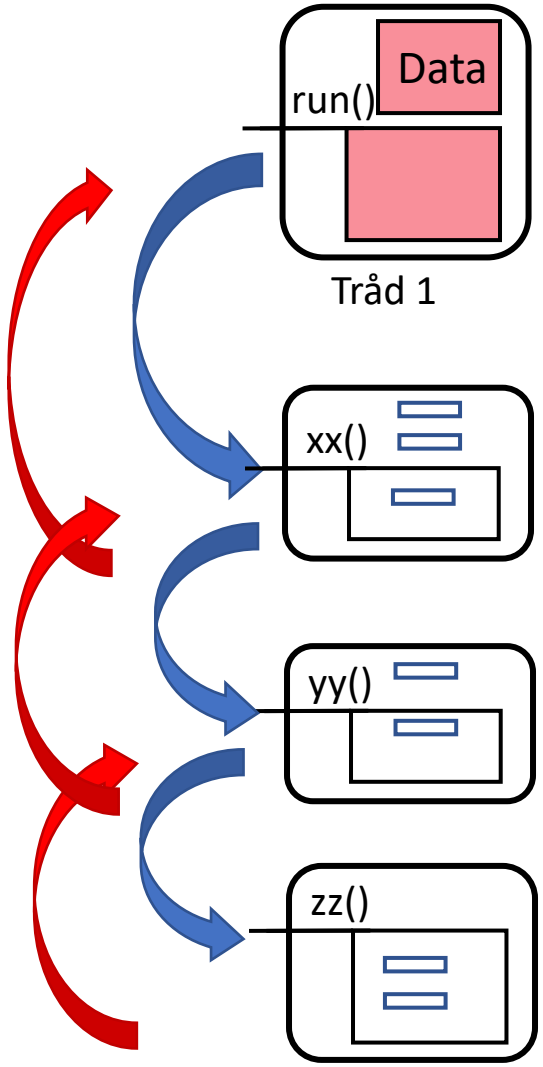
En stabel/stakk av metodekall:

run() i tråd 1, som terminerer.

metoden xx(), som returnerer til . .

metoden yy(), som returnerer til . .

metoden zz() returnerer til . .



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

Variable lokale i metoder legges på kall-stakken, tilhører metodeinstansen og kan ikke akseseres av andre trådere

Instansvariable (oftest private) er felles for alle metoder i objektet og kan følgelig aksesseres av alle metoder og alle tråder som kaller disse

Dette er ikke noe nytt med tråder, slik har det vært fra 23. januar (og det samme i Python og alle andre programmeringsspråk).

Avbrytbare eller ikke-avbrytbare prosesser og tråder.

Ikke så viktig

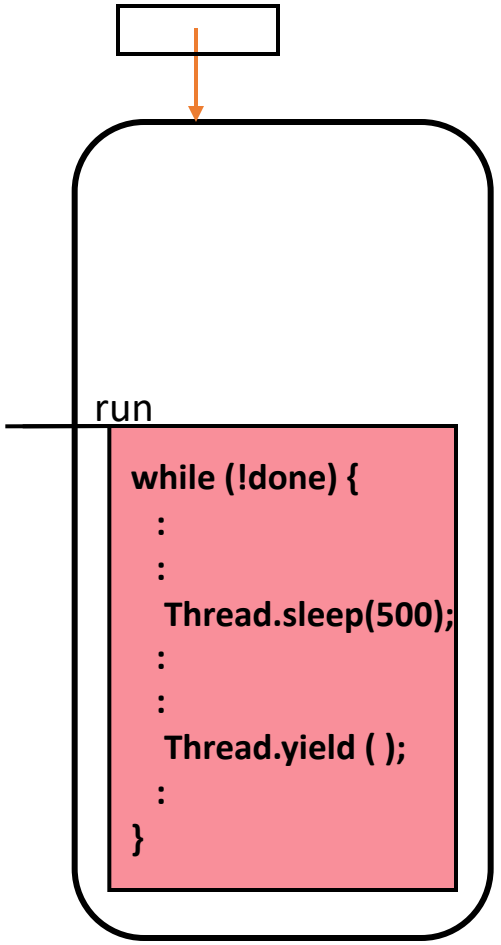
- Hvis en ***prosess*** (eller en tråd) prøver å gjøre en så lang beregning at andre prosesser (tråder) ikke slipper til, må den kanskje avbrytes.
- Alle skikkelige operativsystemer (Windows, MAC OS og Unix/linux) greier å avbryte både prosesser og tråder – f.eks. vha. klokka som sender avbruddssignal 50 ganger per sek.
- Gamle/enkle operativsystemer greide ikke det
  - Og fremdeles ikke enkle mobiltelefoner/lesebrett i dag
- Java-definisjonen sier at det er opp til hver implementasjon av kjøresystemet 'java' om tråder skal være avbrytbare eller ikke.

- Selv om vi har "mange-core" prosessorer vil en tråd ikke gjøre fornuftig arbeid hele tide. Vi bør ha 2 – 8 ganger så mange tråder som prosessorer ("core-er"), avhengig av hvor mye hver tråd trenger å vente (på andre tråder eller I/O)

Tråder kan avbryte seg selv

(ikke veldig viktig)

Vanligvis vil operativsystemet dele prosessoren mellom alle aktive tråder (tidsdeling - “time-slicing”), og altså kaste en tråd ut av prosessoren etter en tid (på engelsk: pre-emption).
run-metoden kan eksplisit la andre tråder slippe til
ved å si yield( );
Noen ganger det kan være ønskelig å la andre aktiviteter komme foran.
yield( ); slipper andre tråder til, og lar tråden som utfører yield vente midlertidig (men denne tråden blir igjen startet opp når det på ny er den sin tur).



Sove og vike

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

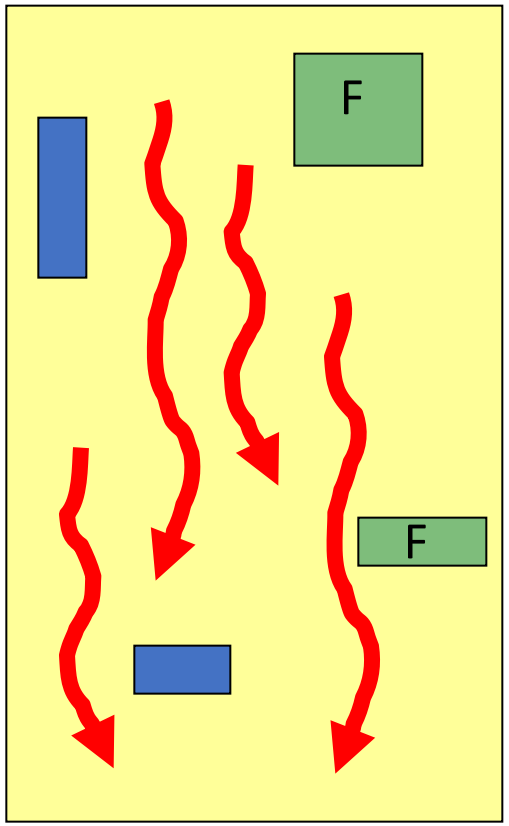
- Først: Mer intro og aller enkleste eksempel
- Så et større program (Kokk og servitør)

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.

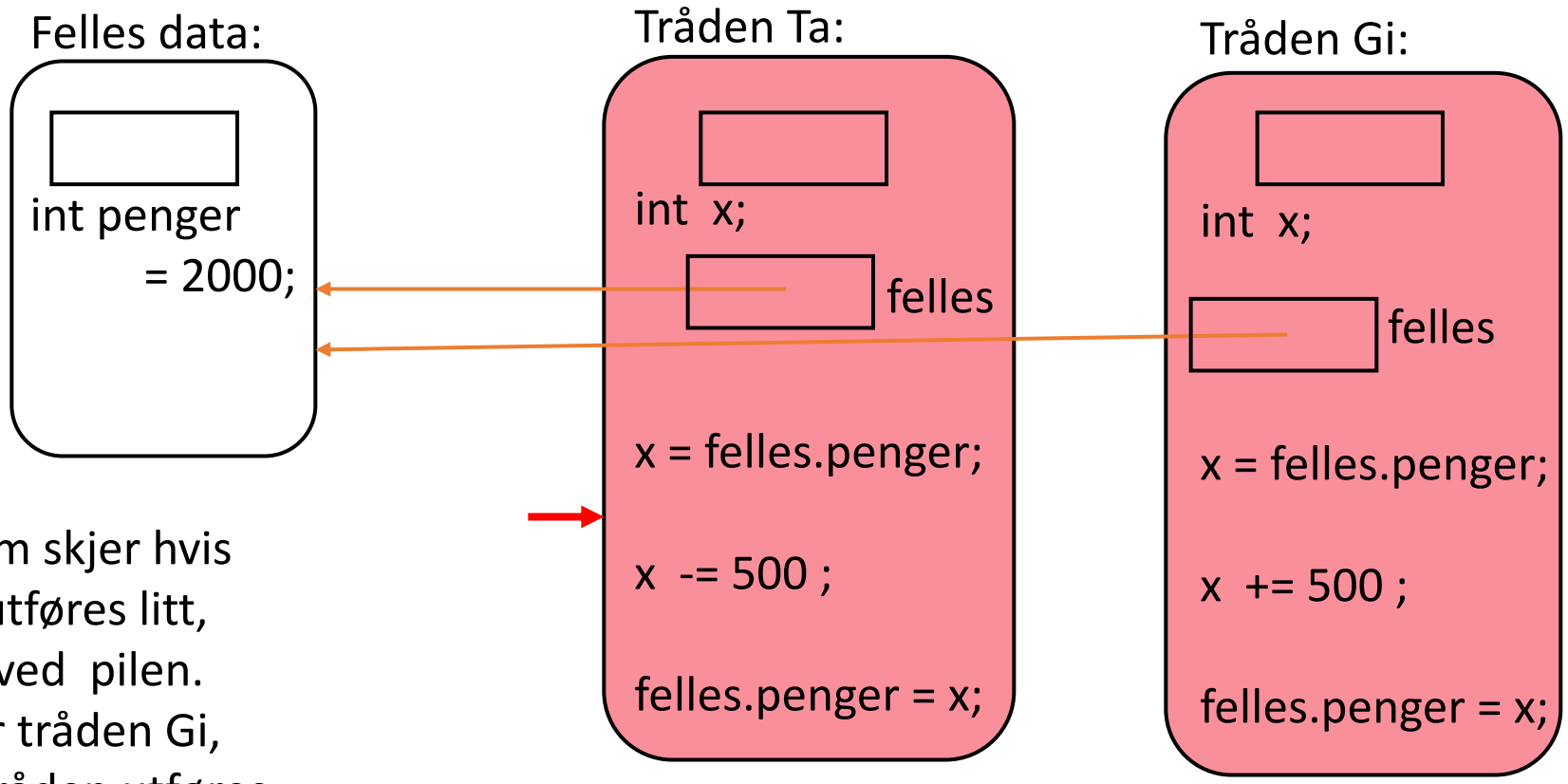
På figuren er det to områder vi har problemer med (dvs. at to eller flere tråder kan risikere å manipulere data i disse områdene samtidig). Disse to områdene er markert med F (for Felles). Kode som aksesserer felles data må skrives inne i en *kritisk region* (forklares snart).

De andre to områdene inneholder data som vi enten vet bare kan leses (”immutable” data), eller vi vet at bare en tråd om gangen skriver i disse dataene. Immutable data er bra! En-om-gangen-resonementer er imidlertid farlige og ofte feilaktig. Hvis du er det minste i tvil så beskytte alle delte data som om de kan bli oppdatert samtidig.



Vårt Java-program med 4 tråder

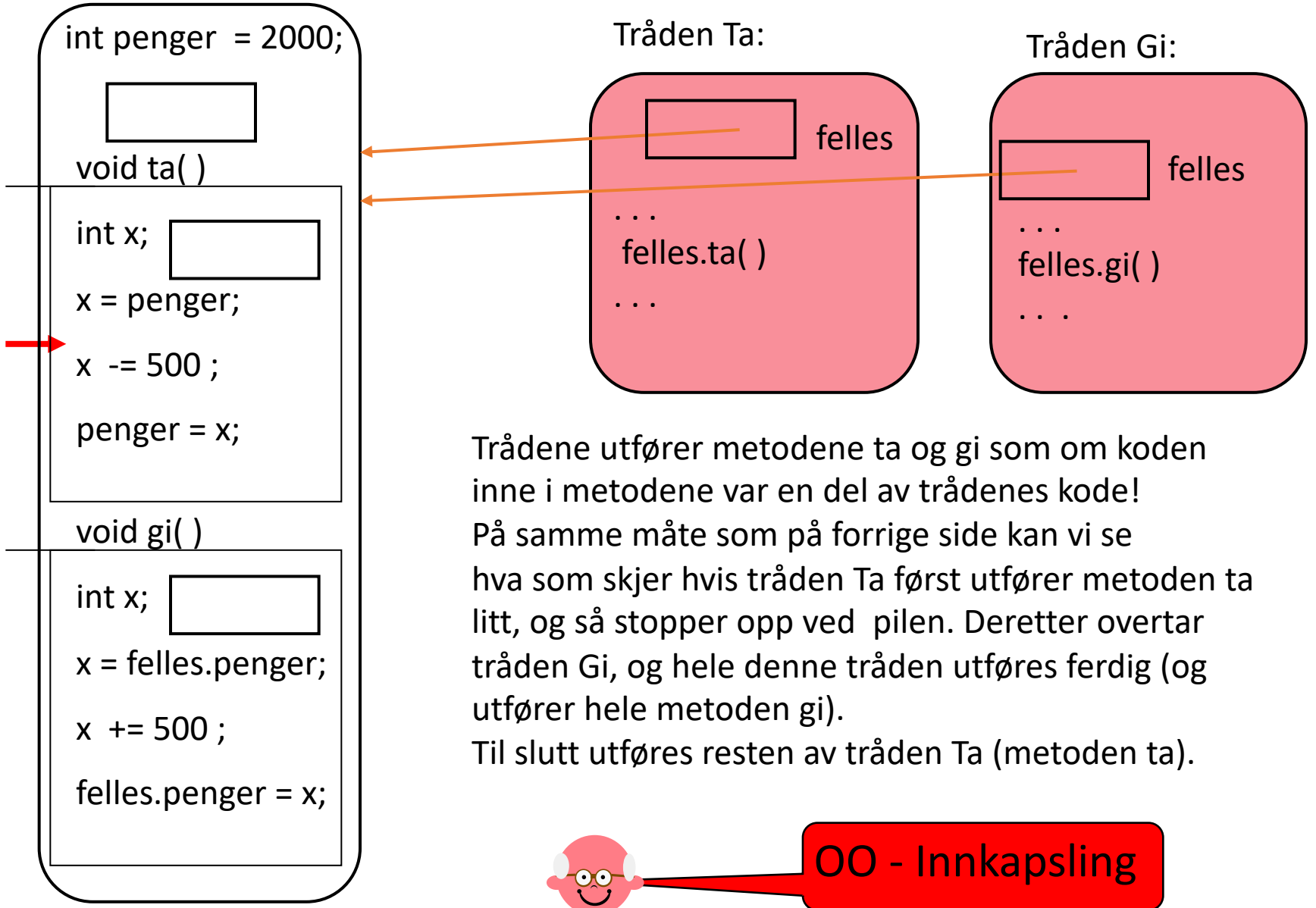
Enkelt eksempel på at to tråder kan ødelegge felles data



La oss se hva som skjer hvis tråden Ta først utføres litt, og stopper opp ved pilen. Deretter overtar tråden Gi, og hele denne tråden utføres ferdig. Til slutt utføres resten av tråden Ta.

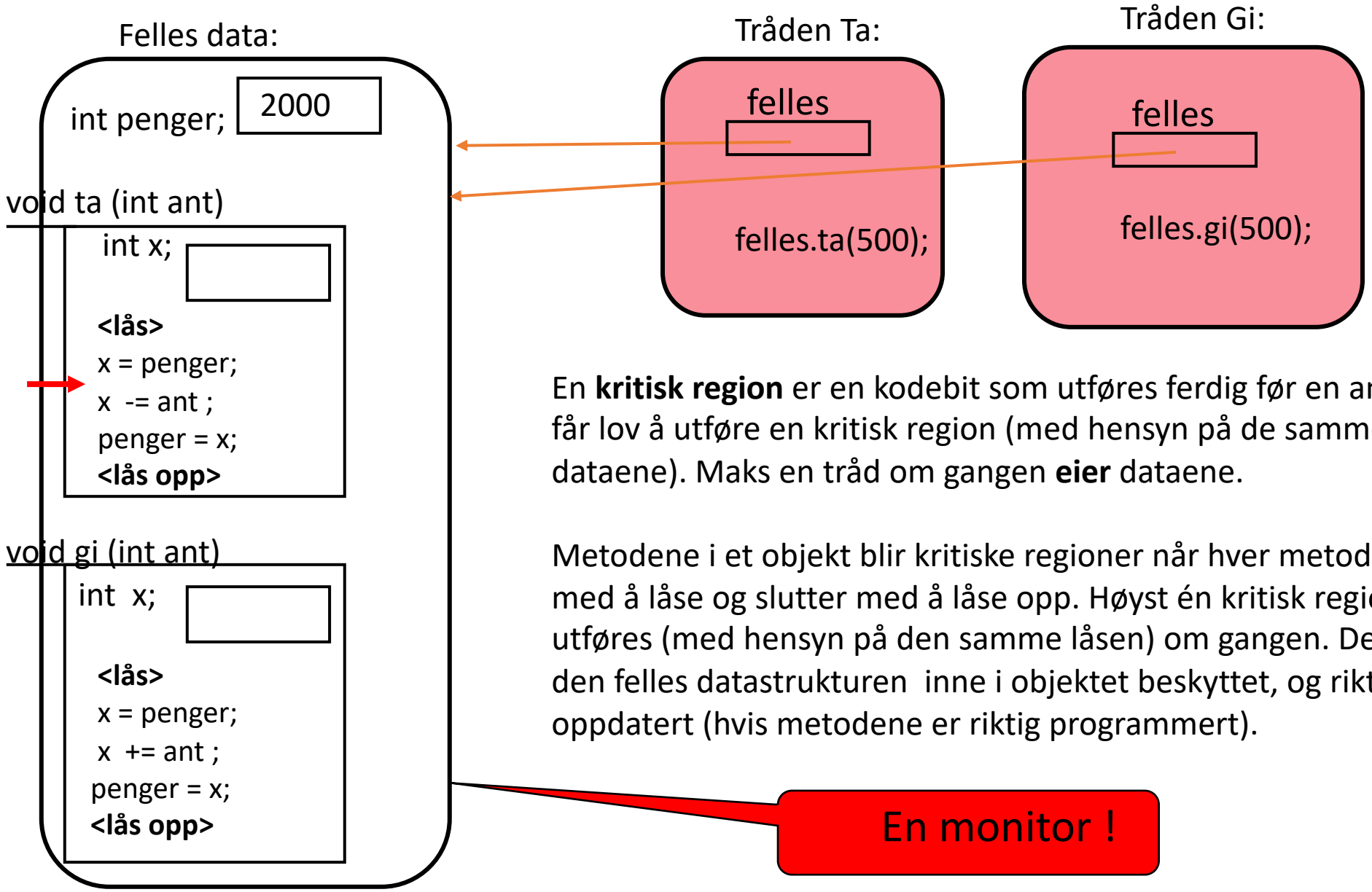


Samme – men objektorientert



Trådene utfører metodene ta og gi som om koden inne i metodene var en del av trådenes kode!
På samme måte som på forrige side kan vi se hva som 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).

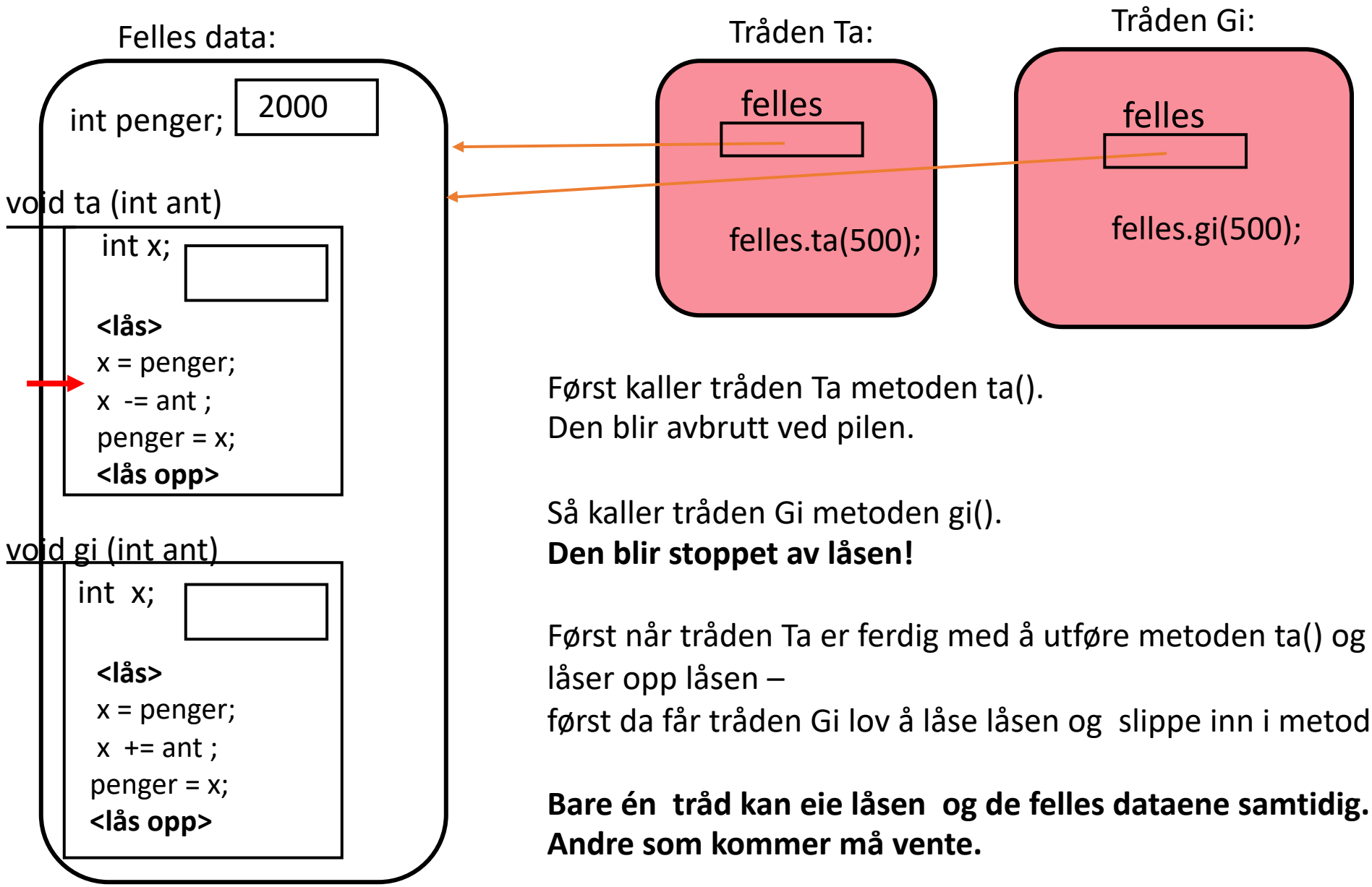
Vi ordner dette med **kritiske regioner**.



En **kritisk region** er en kodebit som utføres ferdig før en annen tråd får lov å utføre en kritisk region (med hensyn på de samme dataene). Maks en tråd om gangen **eier** dataene.

Metodene i et objekt blir kritiske regioner når hver metode starter med å låse og slutter med å låse opp. Høyst én kritisk region kan utføres (med hensyn på den samme låsen) om gangen. Derfor blir den felles datastrukturen inne i objektet beskyttet, og riktig oppdatert (hvis metodene er riktig programmert).

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().

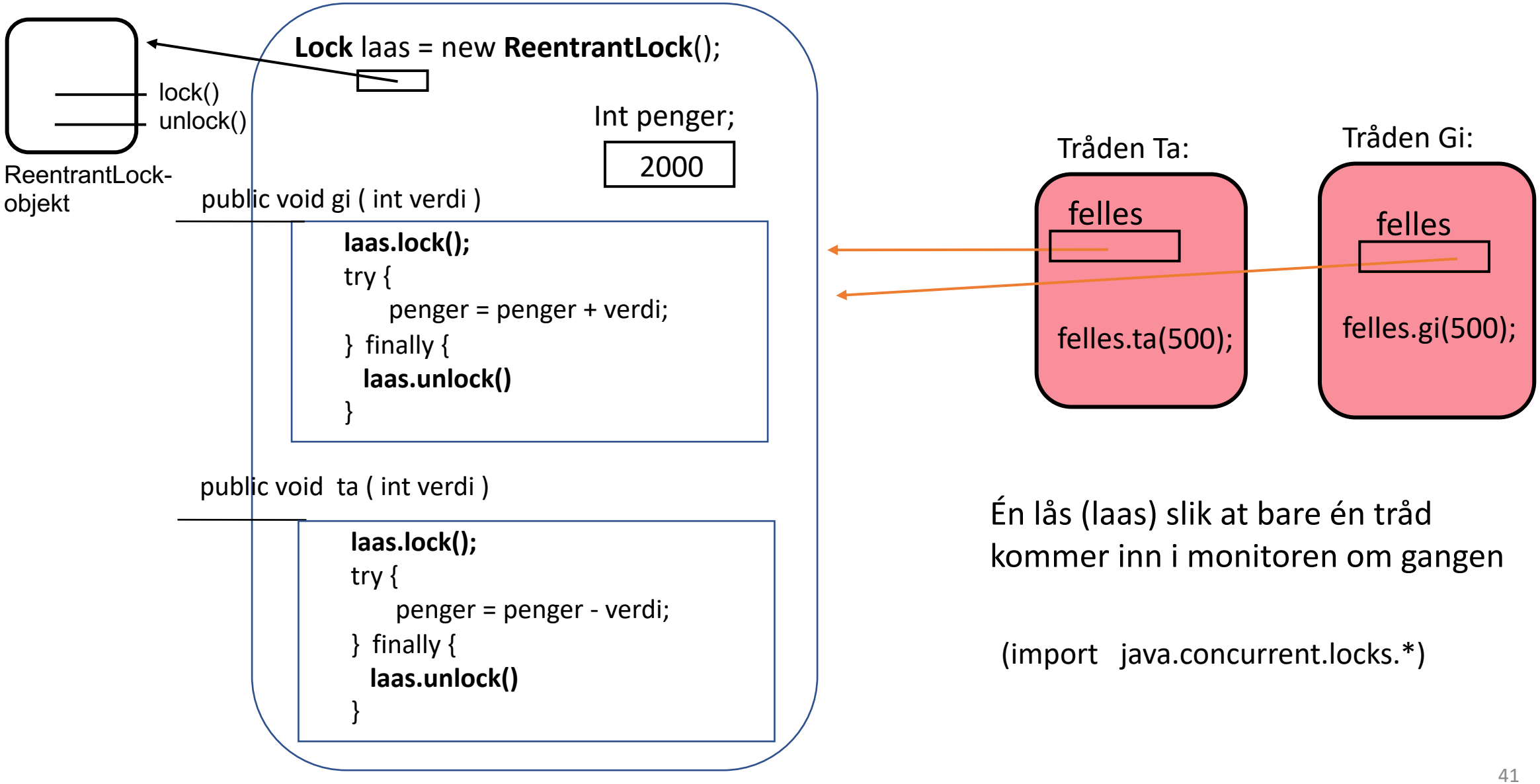
Bare én tråd kan eie låsen og de felles dataene samtidig.
Andre som kommer må vente.

Mer om kritiske regioner / monitorer

- Alle tråder har felles adresserom
- Vi må beskytte lesing og skriving av felles data i **kritisk regioner**
- I et OO språk er det naturlig å samle felles data som hører sammen i et objekt
- Metodene i objektet må da være **kritisk regioner** (med hensyn på dataene i objektet)
- Et slikt objekt kalles en **monitor**
- Til å lage metodene som kritiske regioner bruker vi en LÅS
  - Høyst en tråd er inne i noen av de låste metodene samtidig (låst med samme lås i samme objekt)
  - Én tråd får slippe til, de andre trådene må vente
  - Når den ene er ferdig, slipper de som venter til (en etter en)
  - Alle data i objektet blir skrevet skikkelig ned i primærlageret (RAM) mellom hver tråds tilgang (jfr. forskjellige kjerner).



Slik programmeres dette i Java



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

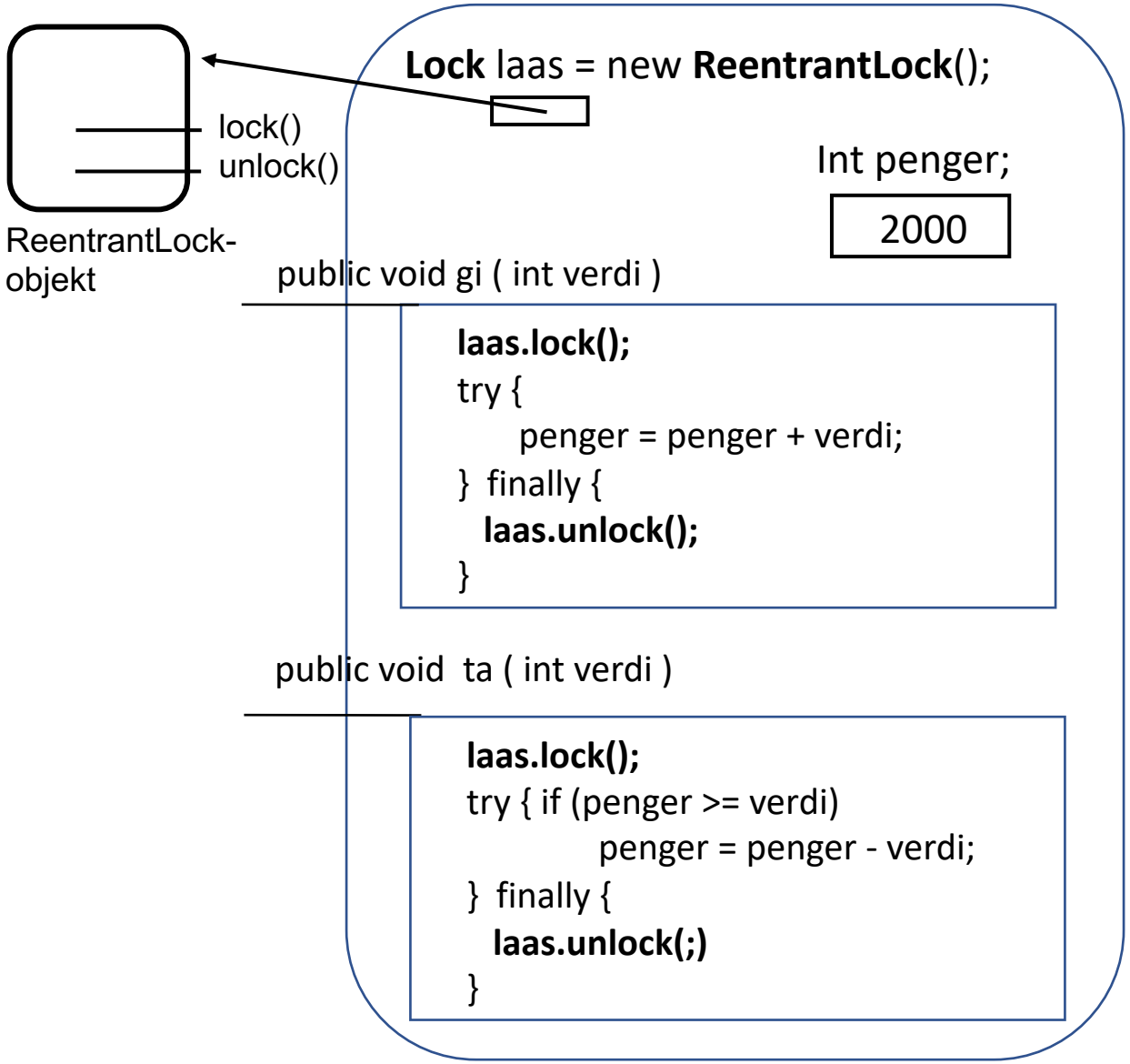
(import java.concurrent.locks.*)

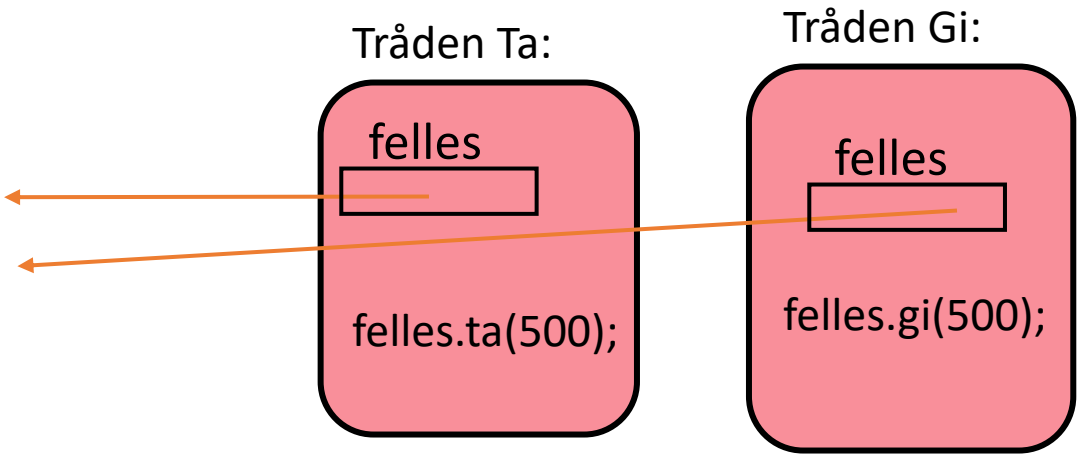
Legg merke til bruken av finally


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

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

Men hva om det ikke er penger igjen på konto?



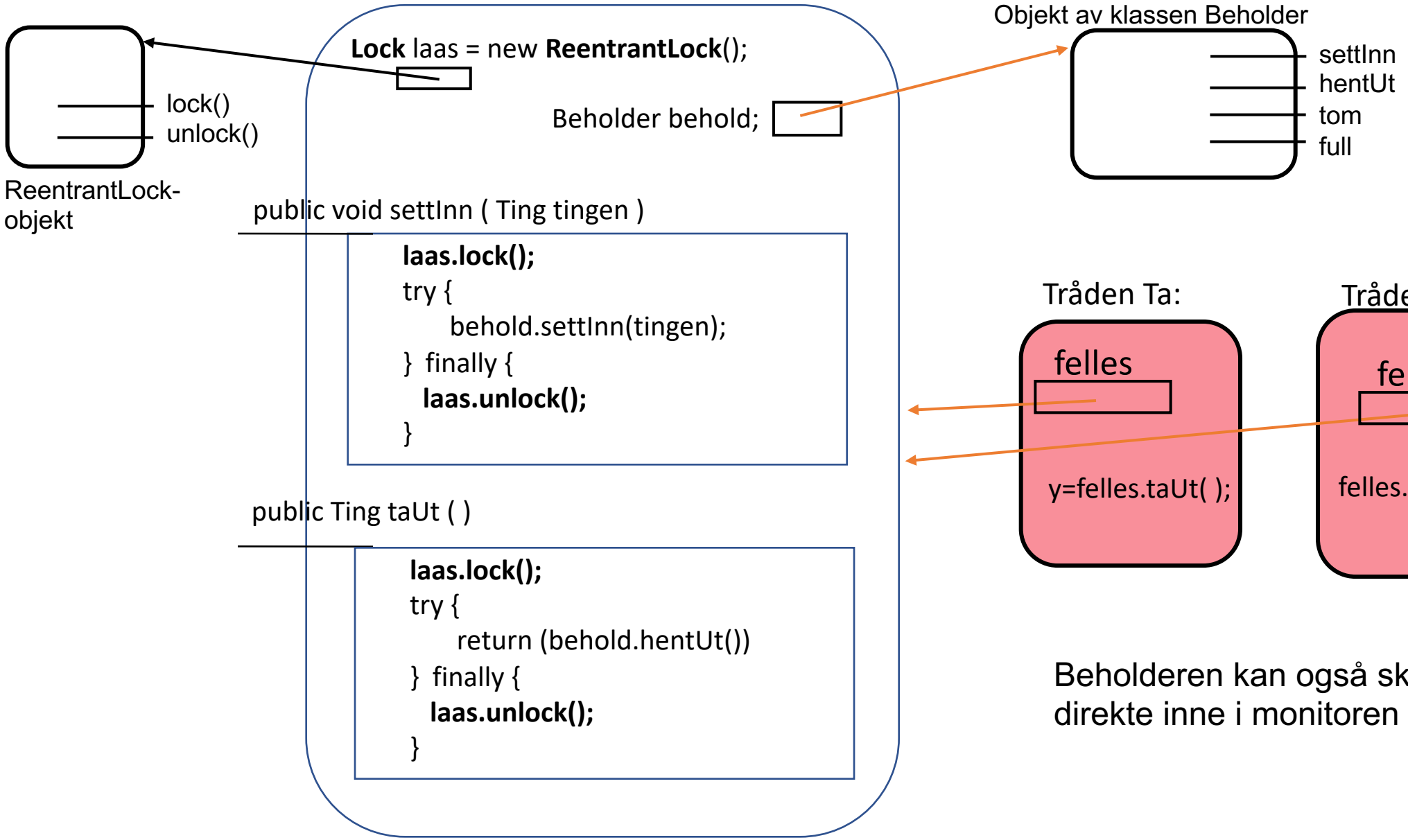


Én lås (laas) slik at bare én tråd kommer inn i monitoren om gangen

(import java.concurrent.locks.*)

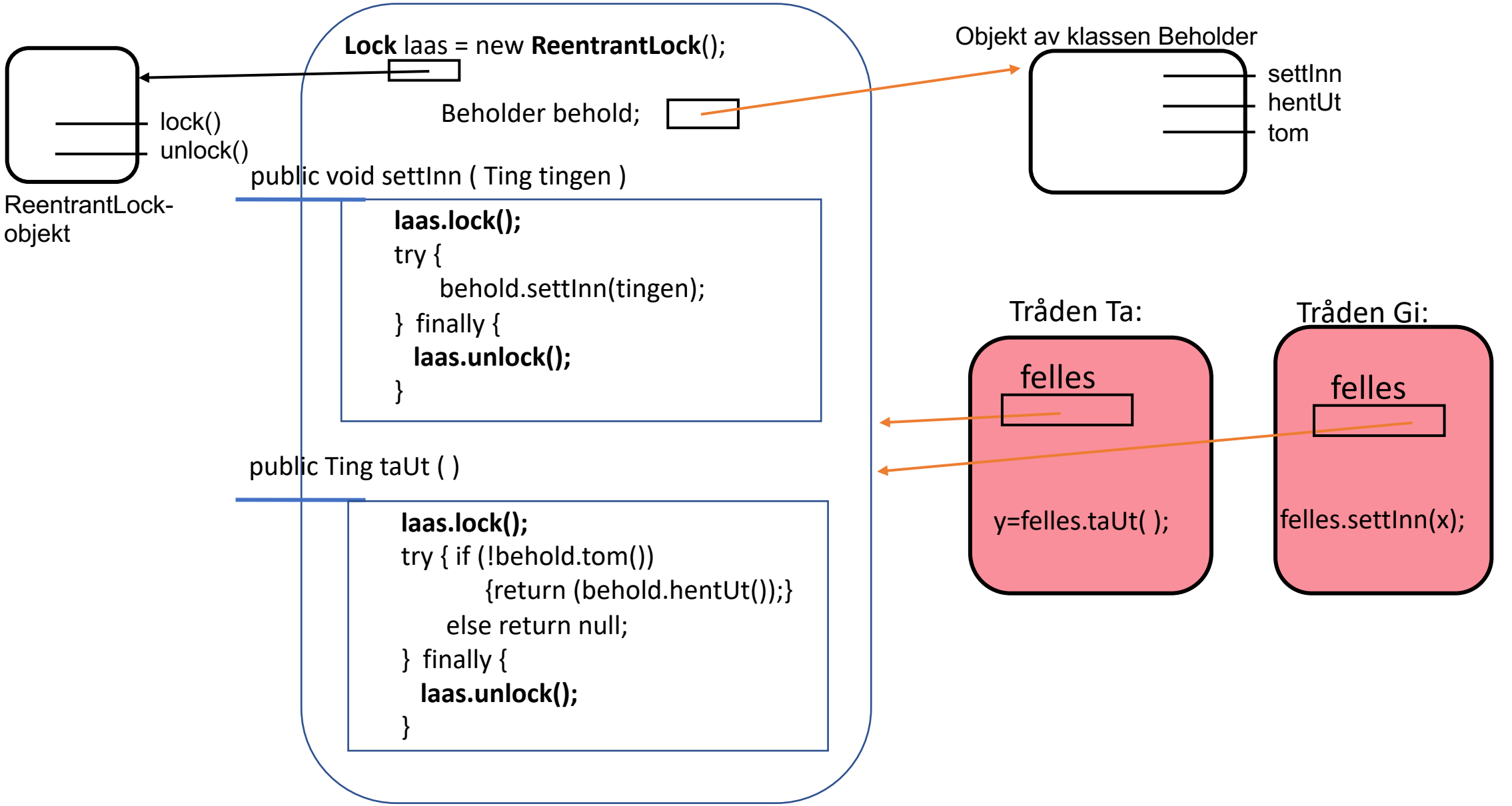
Et litt annet scenario:

En monitor som er/beskytter en beholder



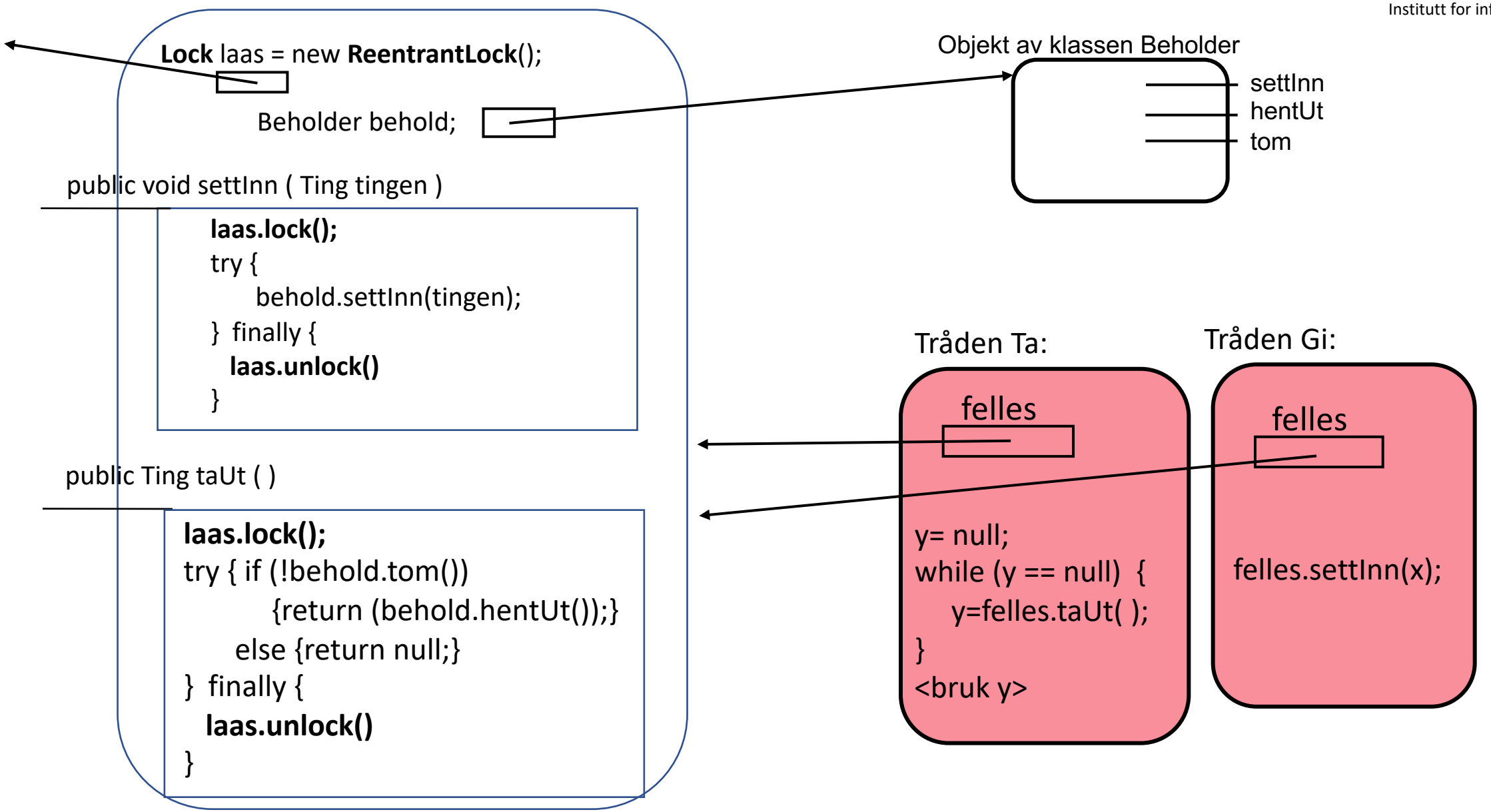
Beholderen kan også skrives direkte inne i monitoren

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



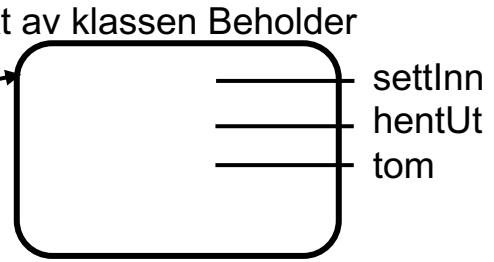
Passiv venting er bedre enn aktiv venting

- 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

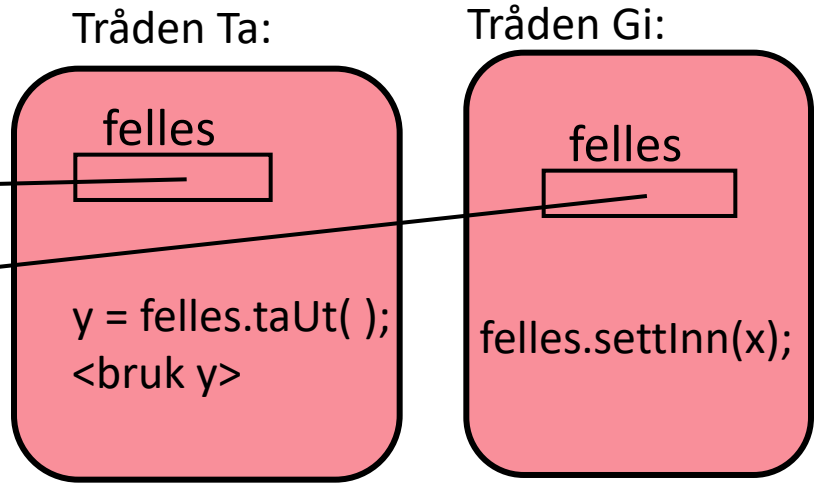

```
Lock laas = new ReentrantLock();
Condition ikkeTom =  laas.newCondition();

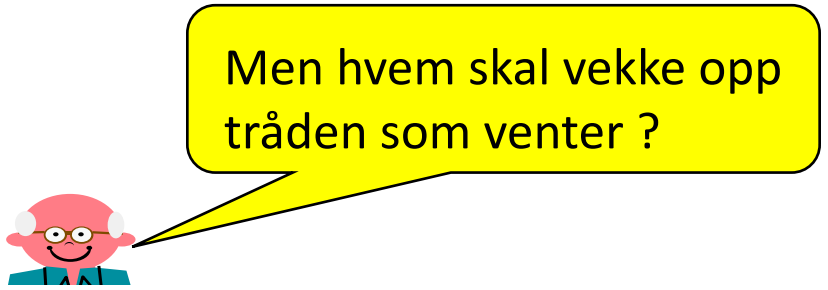
Beholder behold;
```




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

public Ting taUt ( ) throws InterrupedException
    laas.lock();
    try {
       if (behold.tom())
              { ikkeTom.await(); }
        return (behold.hentUt());
    }  finally {
      laas.unlock()
    }
```





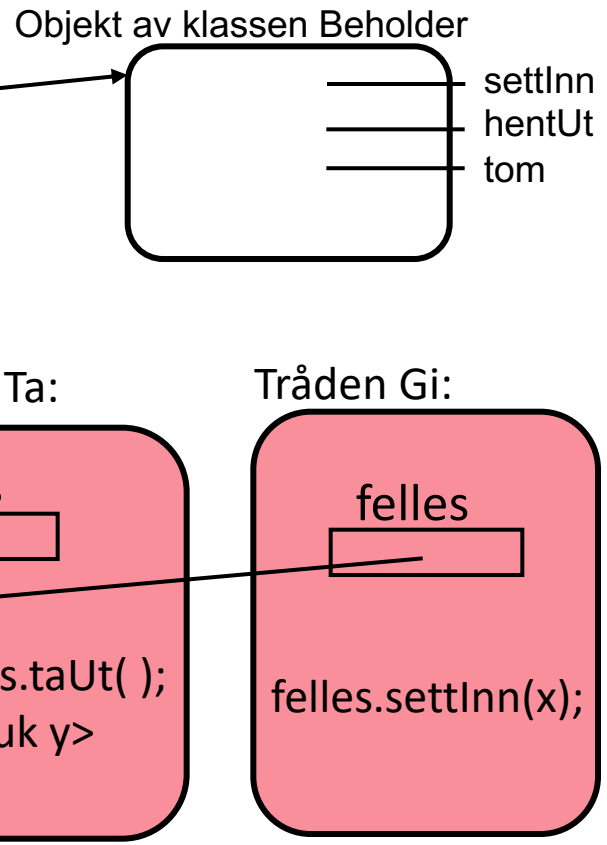
Passiv venting og signalering


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

Beholder behold;

public void settInn ( Ting tingen )
    laas.lock();
    try {
        behold.settInn(tingen);
        ikkeTom.signal();
    }  finally {
      laas.unlock()
    }

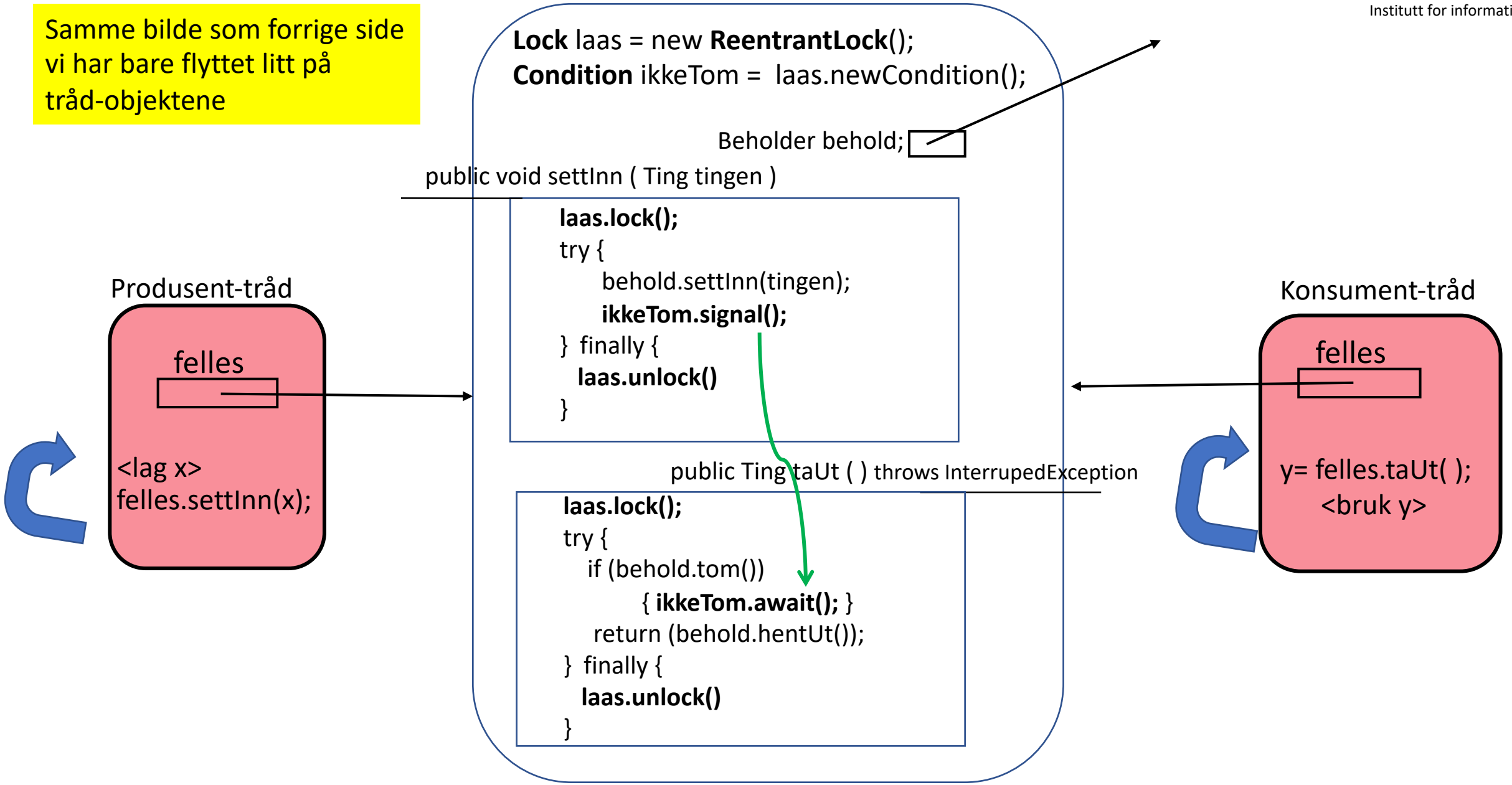
public Ting taUt ( ) throws InterruptedException
    laas.lock();
    try {
       if (behold.tom())
              { ikkeTom.await(); }
        return (behold.hentUt());
    }  finally {
      laas.unlock()
    }
```



Når noe settes inn vil bufferen ikke lenger være tom

Produsent og konsument

Samme bilde som forrige side vi har bare flyttet litt på tråd-objektene



Java-biblioteket java.concurrent

import java.concurrent.locks.*

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 er en lås og en fabrikk som lager objekter som implementerer grensesnittet Condition (på denne låsen)

Passiv venting, køer

Beholder behold;

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

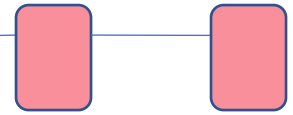
Condition ikkeTom = laas.newCondition();

void settInn (Ting tingen) throws InterrupedException

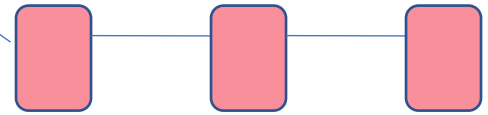
```
laas.lock();
try {
    behold.settInn(tingen);
    // det er lagt inn noe, så det er helt sikkert ikke tomt:
  ikkeTom.signal();
} finally {
  laas.unlock();
}
```

Tingen taUt ( ) throws InterrupedException

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



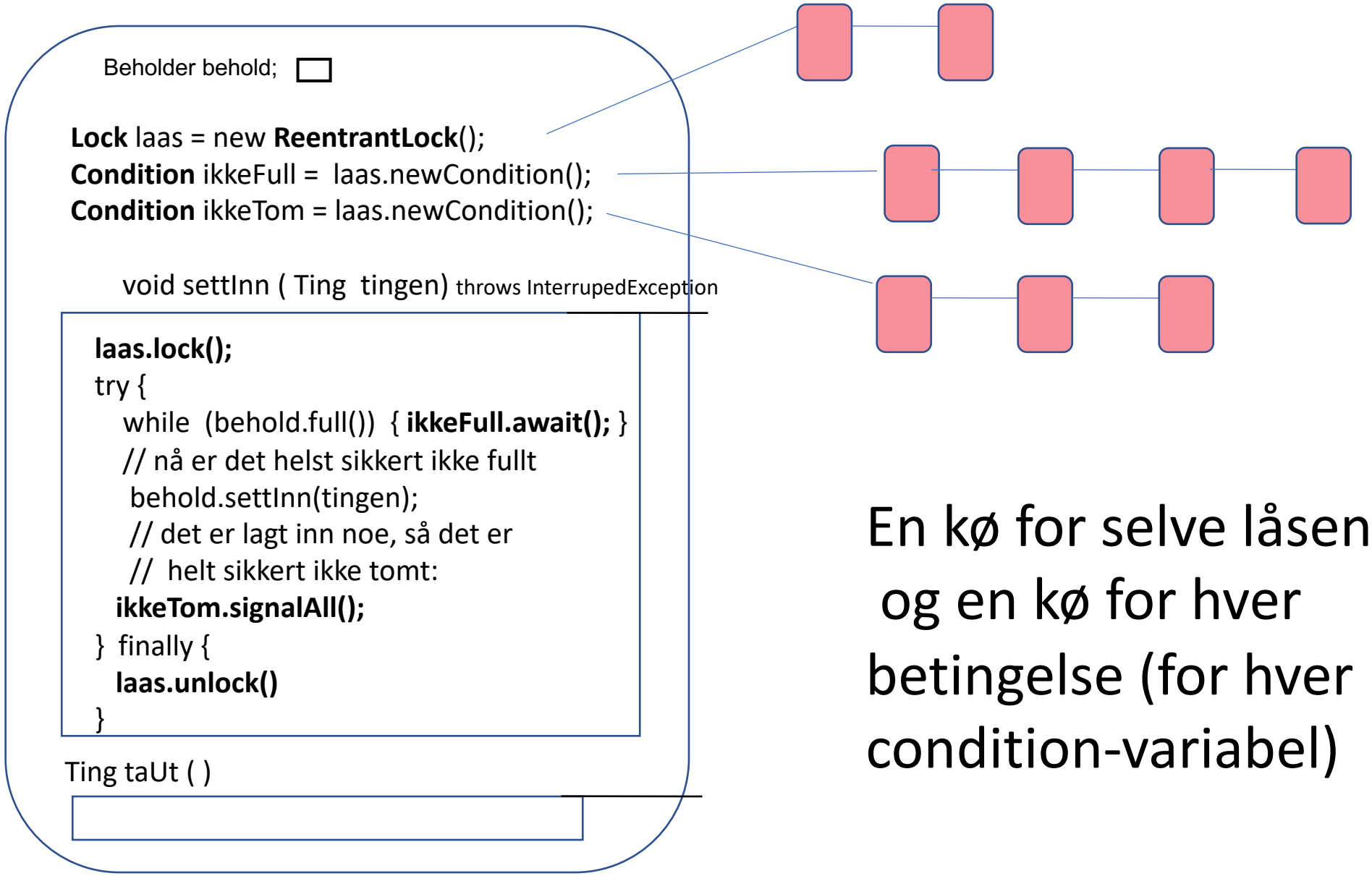
Tråder som venter på låsen for å få komme inn første gang



Tråder som venter på at bufferet ikke skal være tomt

En monitor (et objekt)

settInn() må også vente (på at det er plass)



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

Vente både på tom og full monitor

Beholder behold;

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

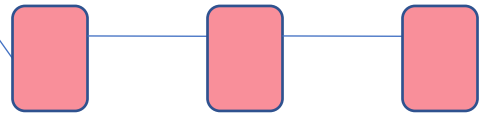
Tråder som venter på låsen for å få komme inn første gang



Tråder som venter på at bufferet ikke skal være tomt



Tråder som venter på at bufferet ikke skal være fullt



void setInn(Ting tingen)

```
laas.lock();
if  (antall == 1000)  ikkeFull.await();
antall++;
behold.settInn(tingen);
ikkeTom.signal();
laas.unlock();
```

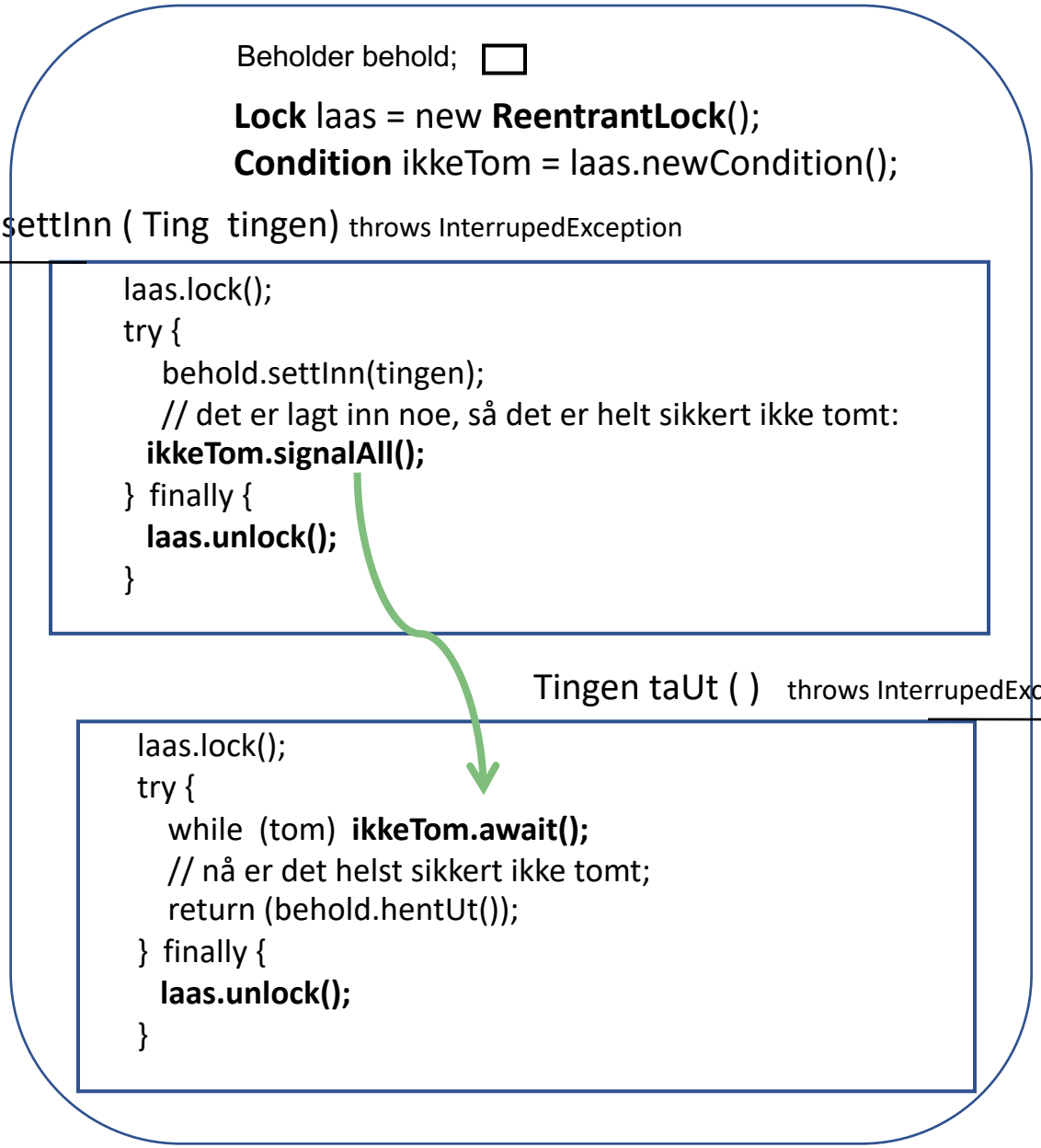
Ting taUt()

```
laas.lock();
If (antall == 0)  ikkeTom.await();
antall --;
ikkeFull.signal();
return (behold.hentUt());
laas.unlock();
```



Disse køene finnes bak kulissene

Regler for robust programmering



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

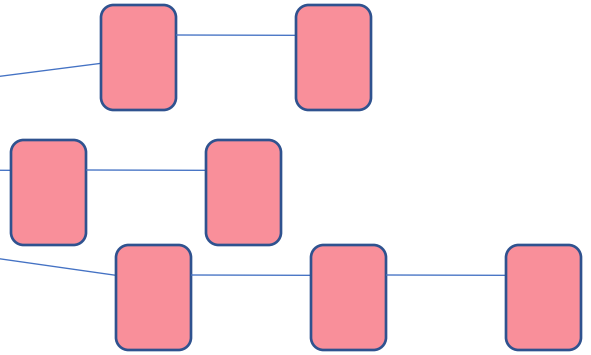
Og vi har skiftet ut signal() med signalAll().

Dette gjør programmet mer robust.

Se notatet om tråder.

Beholder behold;

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



void settInn (Ting tingen) throws InterrupedException

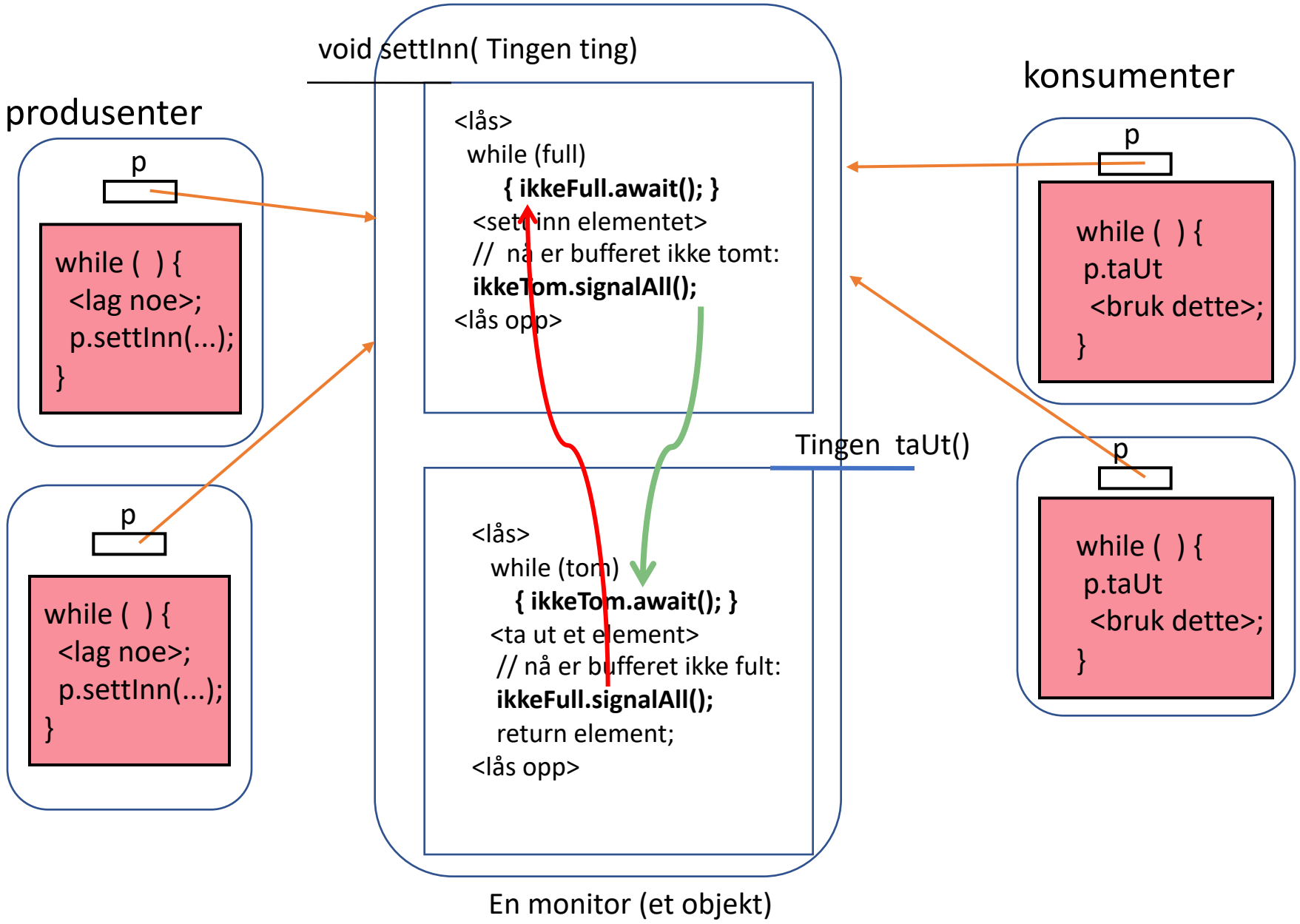
```
laas.lock();
try {
   while (behold.full()) ikkeFull.await();
   // nå er det helst sikkert ikke fullt
    behold.settInn(tingen);
    // det er lagt inn noe, så det er helt sikkert ikke tomt:
  ikkeTom.signalAll();
} finally {
  laas.unlock();
}
```

Tingen taUt ( ) throws InterrupedException

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

Med begge metodene skrevet ut

Produsenter og konsumenter er et viktig eksempel



Et større litt større eksempel – kokk og servitør

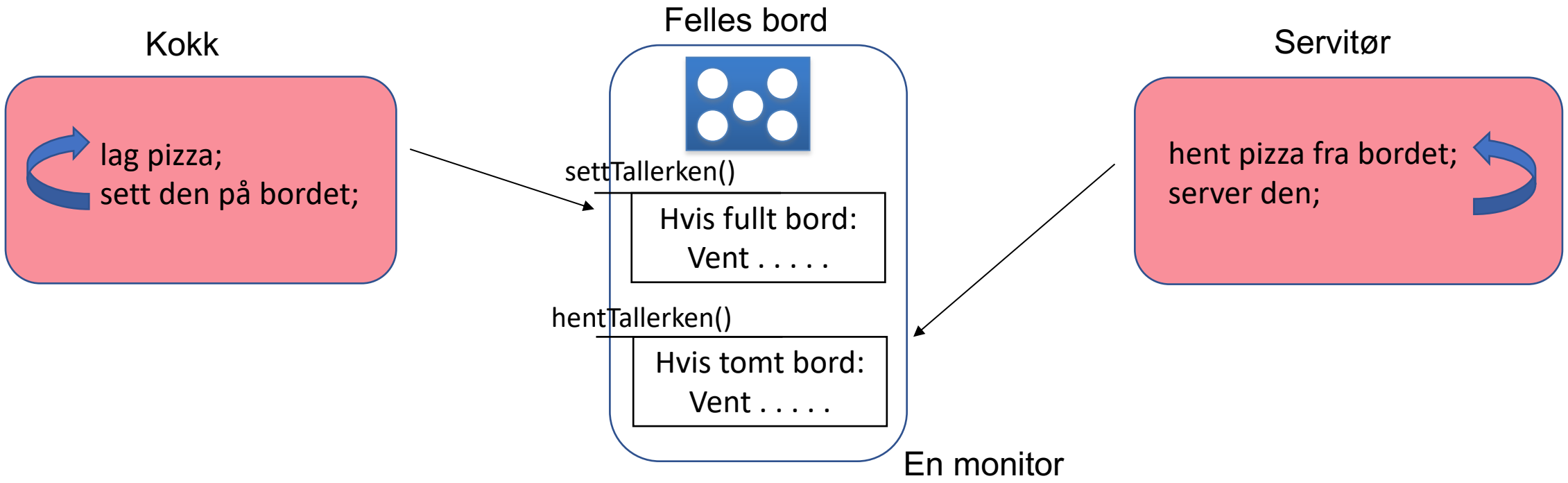
- Kokken lager mat og setter en og en tallerken på et bord
- Servitøren tar en og en tallerken fra bordet og serverer
- Kokken må ikke sette mer enn BORD_KAPASITET tallerkener på bordet (maten blir kald)
- Servitøren kan selvsagt ikke servere mat som ikke er laget (bordet er tomt)

- 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();
    }
```

```
void settTallerken() throws InterruptedException {
    bordlas.lock();
    try {
        while (antallPaBordet >= BORD_KAPASITET) {
            /* sa lenge det er  BORD_KAPASITET tallerkner
                    pa bordet er det ikke lov a sette pa flere. */
            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 er det ikke lov a ta en */
               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;
      }

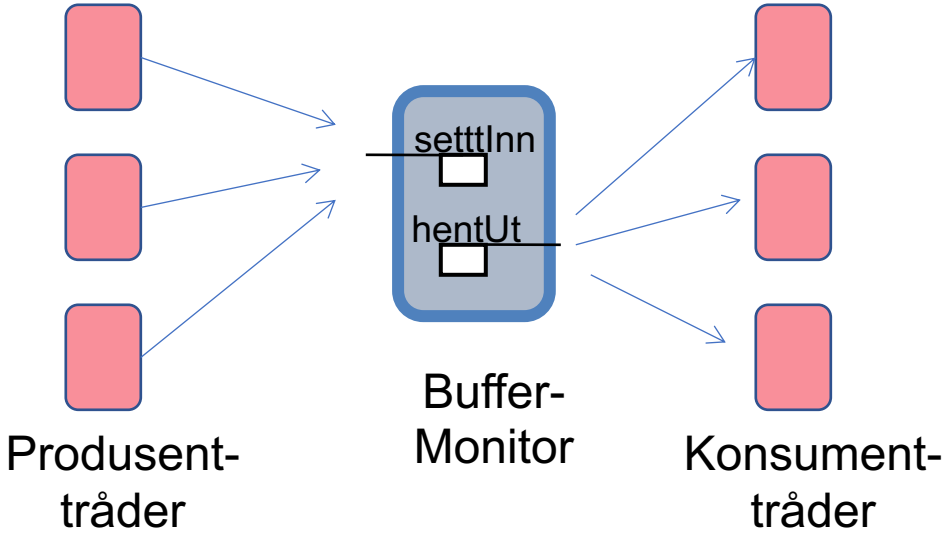
      public void run() {
          try {
              while (ANTALL != servert)  {
                  bord.hentTallerken(); /* hent tallerken og server */
                  servert++;
                  System.out.println("Kelner serverer nr:" +  servert);
                  Thread.sleep((long) (1000 * Math.random()));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

Hoved "take-away" i dag

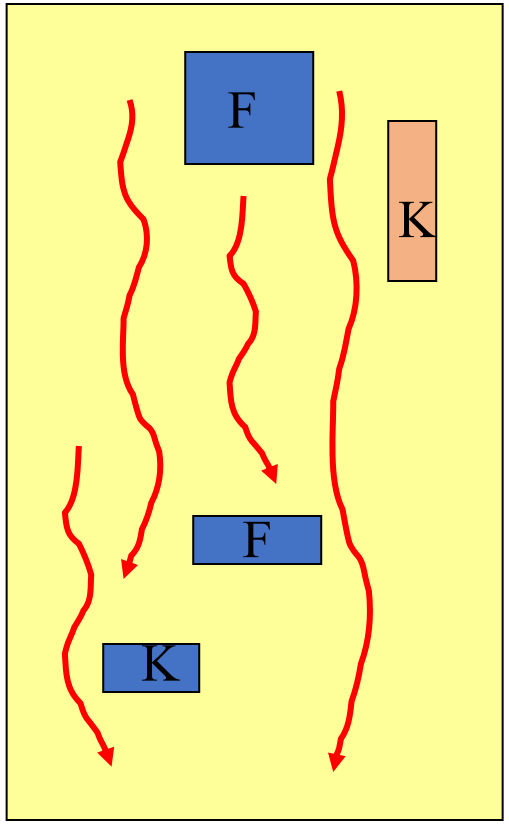
Du kan programmere parallelle aktiviteter i Java ved hjep av tråder.

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)

Oppsummering
Vi har lært:

- Hva tråder brukes til
- Hvordan vi lager tråder 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 resoneere om programmets tilstand
  - Nesten umulig å teste nok tilfeller
    - Tidsavhengige feil kan vise seg først etter mange år.

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

Enda mer om monitorer, kritiske regioner og Big Java

- Kritiske regioner (ekskusiv tilgang til felles data) kan løses på mange måter
- Det er bl. a. noe som heter *semaforer* (P og V)
  - og aktiv venting (har vi snakket litt om) (spin locks)
- I IN1010 bruker vi *monitorer* fordi det er mest objektorientert.
- Venting og signalering i monitorer kan gjøres på forskjellige måter.
  - I dag: En god måte, og slik det gjøres i Horstmann
  - Ekstrastoff: Slik det gjøres innebygget i Java (notify og wait, ikke pensum i IN1010)

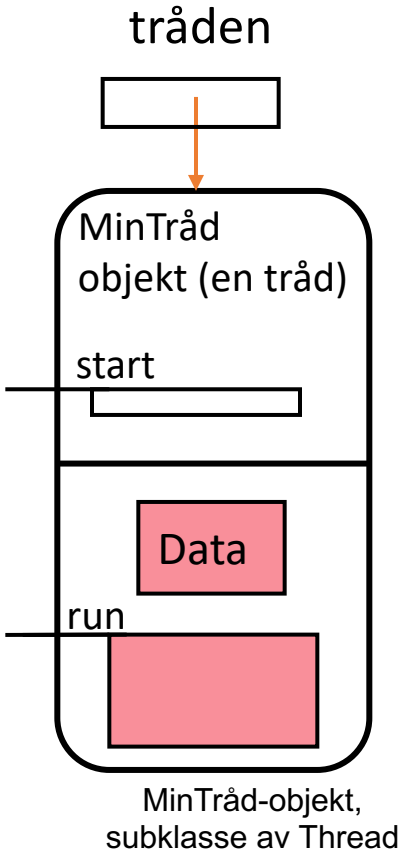
Tråder i Java II: Subklasse av Thread

(Subklassing istedenfor delegering)

I Javas bibliotek:

```
class Thread implements Runnable {
   . . .
   public void run () {
      <se Java API class Thread>
   }
}
```

```
class MinTrad extends Thread {
    <datastruktur>
    @Override
    public void run( ) {
       while (<mer å gjøre>) {
            <gjør noe>;
             . . .
       }
    }
}
```



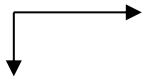
MinTråd-objekt,
subklasse av Thread

Tråder i Java II: Subklasse av Thread

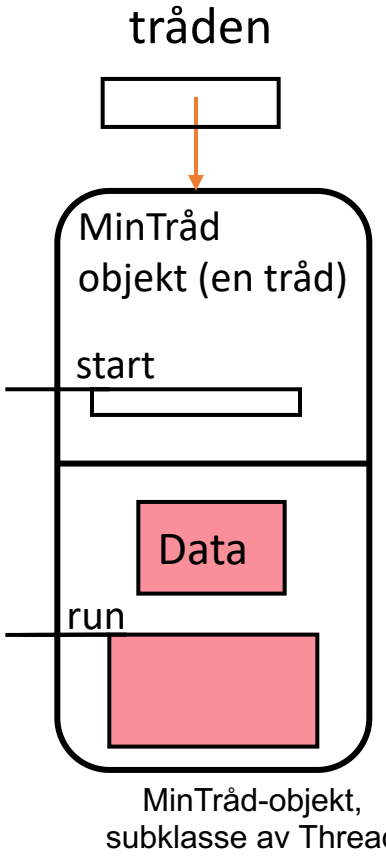

```
class MinTrad extends Thread {
    <datastruktur>
    @Override
    public void run( ) {
       while (<mer å gjøre>) {
            <gjør noe>;
             . . .
       }
   }
}
```

En tråd lages og startes opp slik:

```
Thread minTrad = new MinTrad();
minTrad.start( );
```



(Subklassing istedenfor delegering)

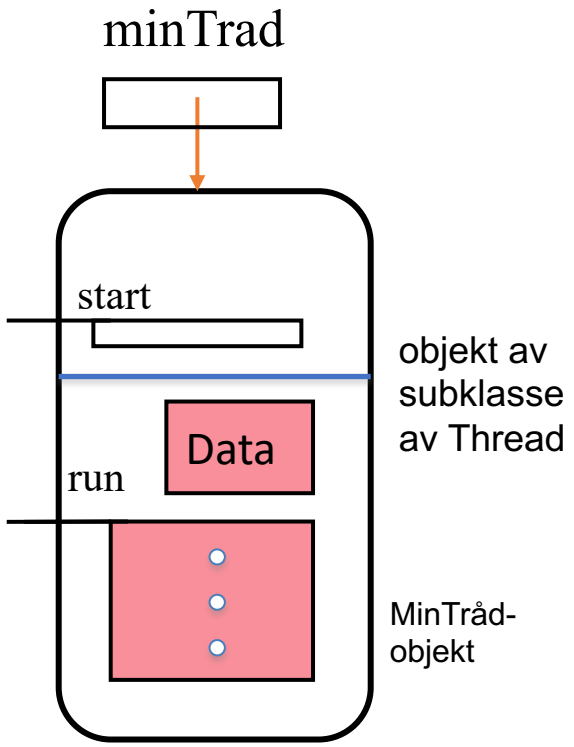


MinTråd-objekt,
subklasse av Thread

Tråder II: Subklasse av Thread

(Ikke noe nytt her)

```
Thread minTrad = new MinTrad();
minTrad.start( );
```



Subklassing istedenfor delegering

Sove (i Thread)

- Når run()-metoden er i klassen Thread (subklassing) er sleep()-metoden i scopet, og programmet og tråden kan da sove slik:

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

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

En stoppeklokke (med subklasse)



```
import java.util.Scanner;

class TradKlokkeSub {
    public static void main (String [] args) {
        Scanner minInn = new Scanner (System.in);
        Thread traden = new TradKlokke();
        System.out.println(" Stoppeklokke (subkl)");
        System.out.println(
               " Tast CR for å stoppe og starte");
        minInn.nextLine();

        traden.start();
        minInn.nextLine();

        traden.interrupt();
        System.out.println(" Takk for naa  "   );
    }
}
```

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

Om Javas iboende monitor-mekanismer

- Da Java ble designet ble det bygget monitor-mekansimer inn i språket:
- Alle objekter har en lås
- Metoder kan ha modifikatoren *synchronized* som gjør at metoden blir en kritisk region (som beskyttes av objektets lås)
- To metoder i klassen Object lar en tråd vente og våkne opp igjen:
  - wait()
  - notify()
  - notifyAll()

wait(), notify() og notifyAll() i class Object

- Hvorfor ikke bruke dette ?
- Fordi:
  - Gjør at det blir ÉN kø av passivt ventende tråder per objekt.
    - Pluss en kø for å komme inn i objektet første gang (venter på synchronized)
  - Ingen løfter om hvilken rekkefølge ventende tråder blir startet opp.
    - Kan lett bli "livelock" (deadlock -> vranglås, livelock -> alltid forbigått?)
    - Kan til og med ta en ny istedenfor en som venter i "wait".
- Brukes bak kulissene av Doug Leas concurrency-bibliotek.
  - Java har én iboende lås per objekt

Restaurant Java Original


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

```
class FellesBord {  // En monitor
    private int antallPaBordet = 0;
    private final int BORD_KAPASITET;
    /* Invariant: 0 <= antallPaBordet <= BORD_KAPASITET */

    public FellesBord (int ant) {
        BORD_KAPASITET = ant;
    }
```

```
synchronized void settTallerken() throws InterruptedException {
    // Invarianten holder
    while (antallPaBordet >= BORD_KAPASITET) {
        /* sa lenge det er  BORD_KAPASITET tallerkner
                    pa bordet er det ikke lov aa sette pa flere. */
        wait();
    }
    // Invarianten holder og naa er antallPaBordet < BORD_KAPASITET
    antallPaBordet++;
    System.out.println("Antall paa bordet: " + antallPaBordet);
    // Invarianten holder og antallPaBord > 0;
    // Si fra til de som tar tallerkener:
    notifyAll();
}
```

```
    synchronized void hentTallerken() throws InterruptedException {
        // Invarianten holder
        while (antallPaBordet == 0) {
            /* Sa lenge det ikke er noen talerkener pa
                    bordet er det ikke lov aa ta en */
            wait();
        }
        //Invarianten holder og naa er antallPaBordet > 0
        antallPaBordet --;
        // Innvarianten holder og antallPaBordet < BORD_KAPASITET
        // Si fra til de som setter tallerkener pa bordet:
        notifyAll();
    }
}
```

```
class Kokk extends Thread {
    private FellesBord bord;
    private final int ANTALL;
    private int laget = 0;
    public  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();
                sleep((long) (500 * Math.random()));
             }
         }
         catch (InterruptedException e) {
             System.out.println("Uventet stopp 1");
         }
    }
}
```

```
class Servitor extends Thread {
    private FellesBord bord;
    private final int ANTALL;
    private int servert = 0;
    public Servitor(FellesBord bord, int ant) {
        this.bord = bord;
        ANTALL = ant;
    }
    public void run() {
       try {
           while (ANTALL != servert)  {
               bord.hentTallerken();
               servert++;
               System.out.println("Kelner serverer nr:" +  servert);
               sleep((long) (1000 * Math.random()));
           }
       }
       catch (InterruptedException e) {
            System.out.println("Uventet stopp 2");
       }
   }
}
```