UNIVERSITETET I OSLO

Institutt for informatikk

IN1010 - Våren 2025
Parallellitet og tråder - del 1
20. mars 2025

Stein Gjessing

Horstmann kap 20.1 – 20.3

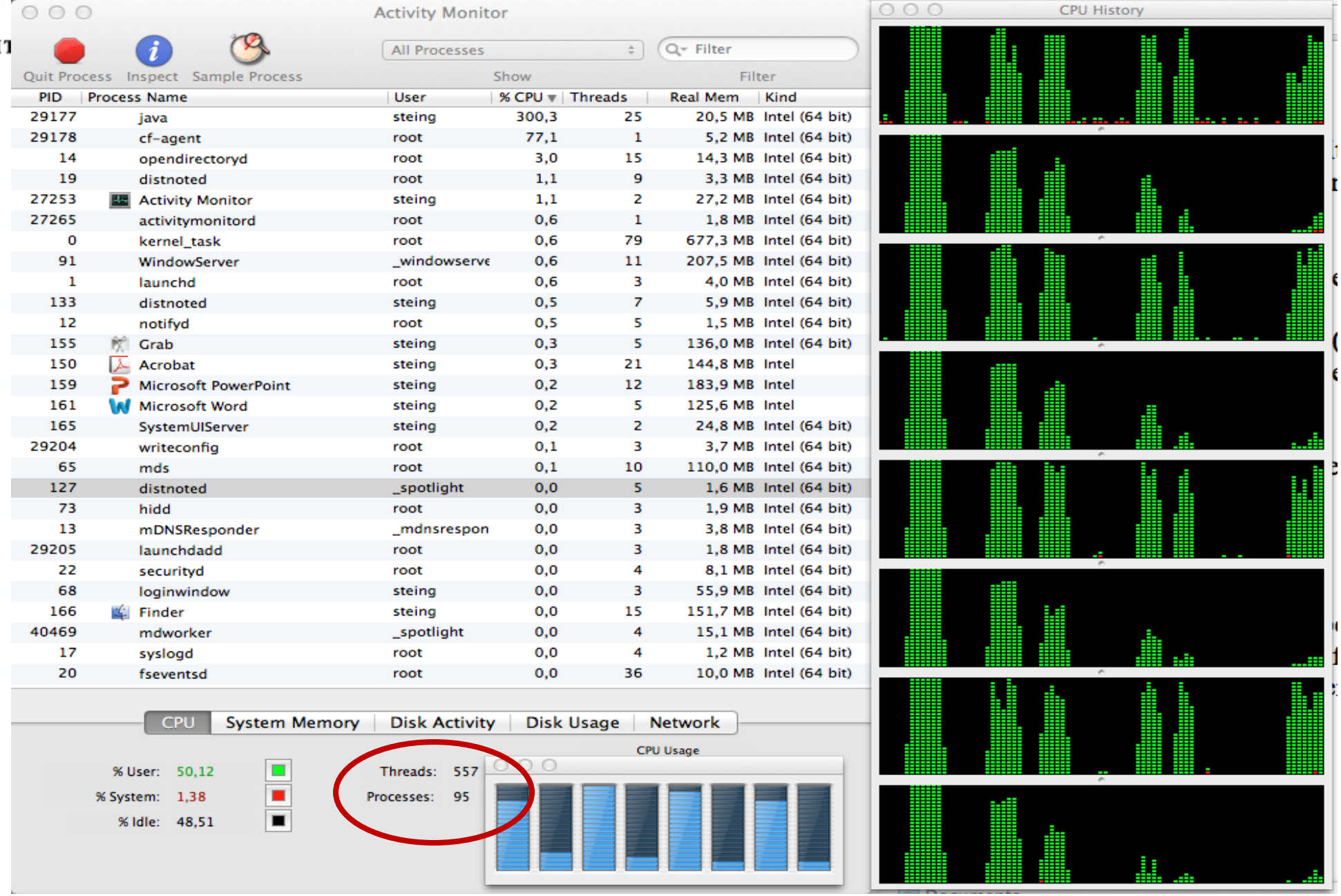
Oversikt

- 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
- Trådsikre datastrukturer

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 snakke, video, chat, samtidig som du jobber . . . .
    - en nett-butikk, en kino, en flyavgang . . . .
  - Oppgaver som naturlig modelleres i flere parallelle deler
    - Bredde-parallellitet - Samlebånd-parallellitet

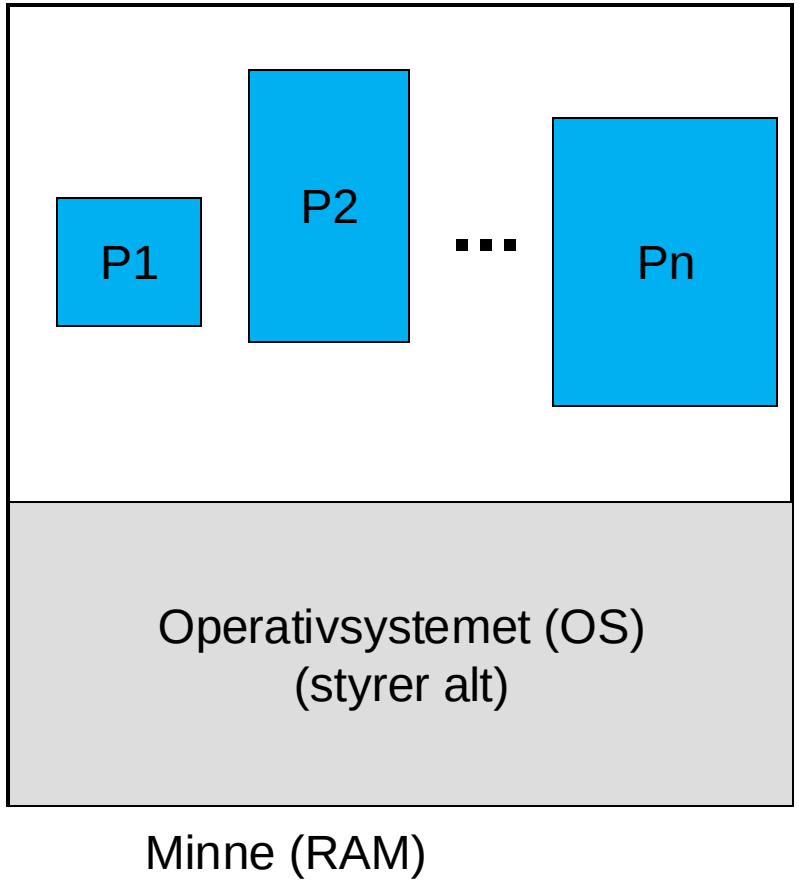
- **Ekte parallellitet - hastighetsøkning**
  - Når jeg simulerer deler av Internet bruker jeg så mange kjerner som mulig
  - Tunge beregningsoppgaver som ellers tar (for) lang tid å løse.
  - ”Dual-core”, "8-core", ”N-core”: Ekte parallellitet



Maskin med 8 kjerner

Parallellitet: prosesser og tråder (IN1020)

- 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.

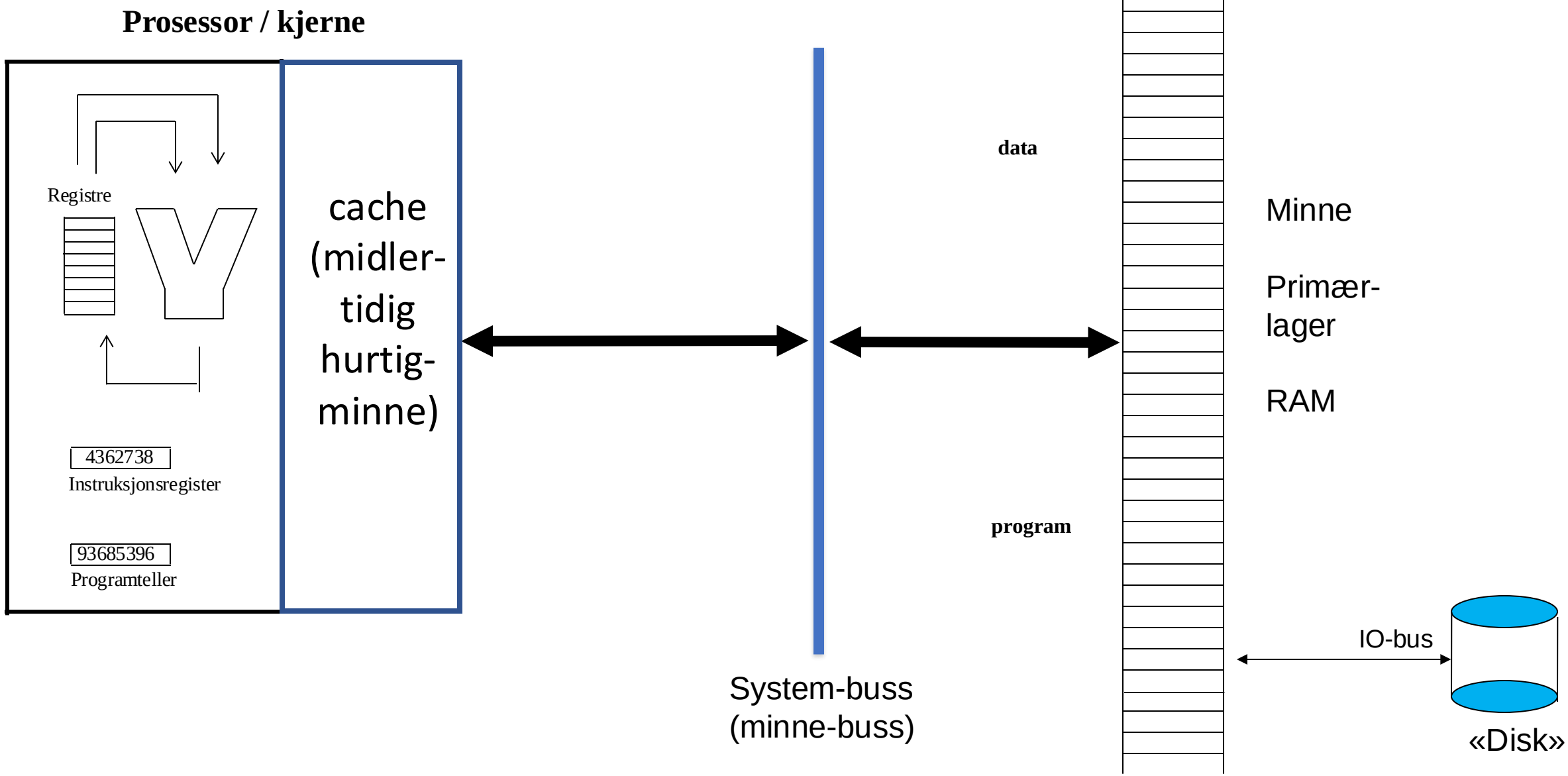


UNIVERSITETET I OSLO

Innmaten i en datamaskin

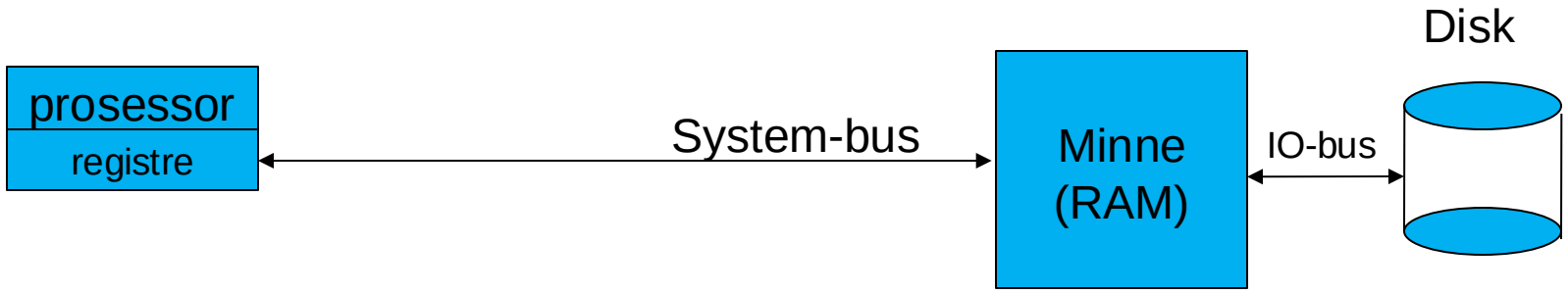
(datamaskinarkitektur – IN1020)

Institutt for informatikk

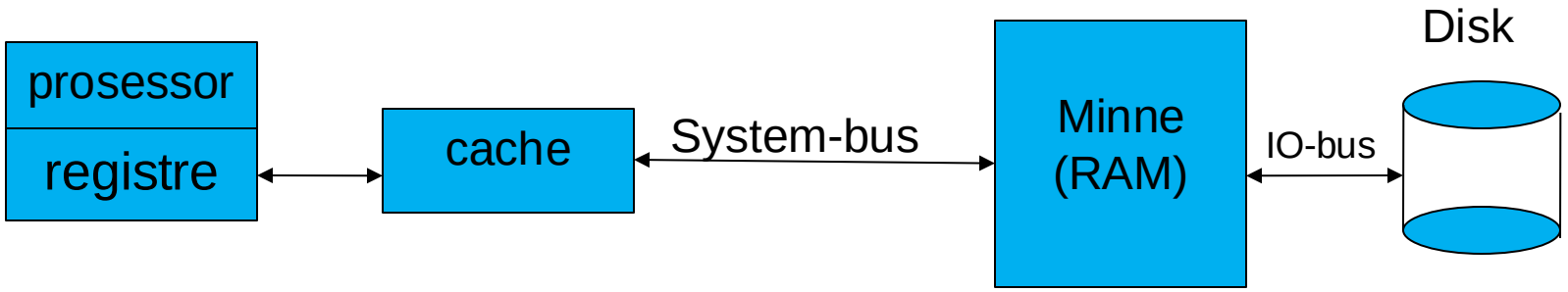


Datamaskinarkitektur (Computer Architecture)

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

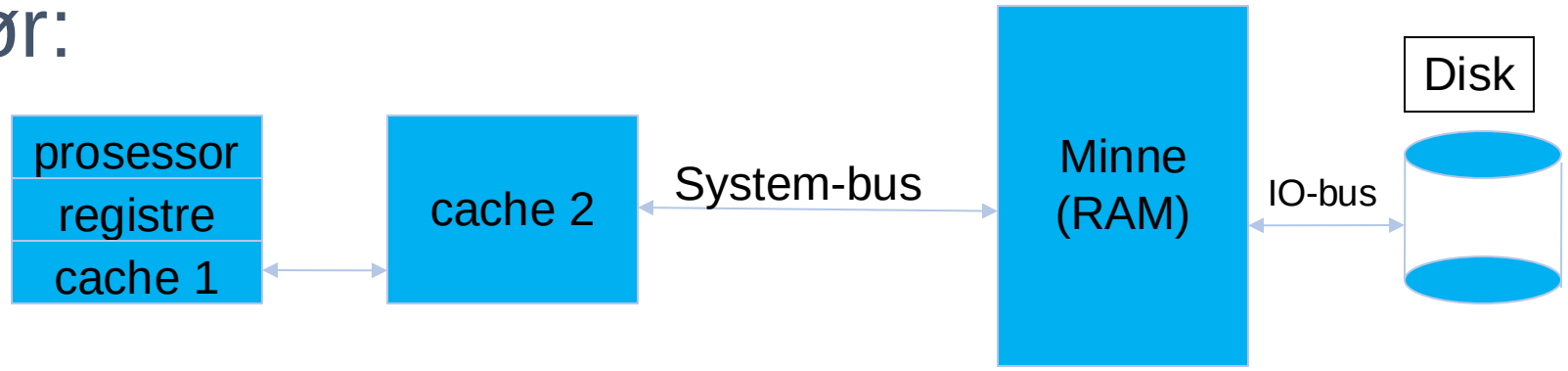


2. Før:

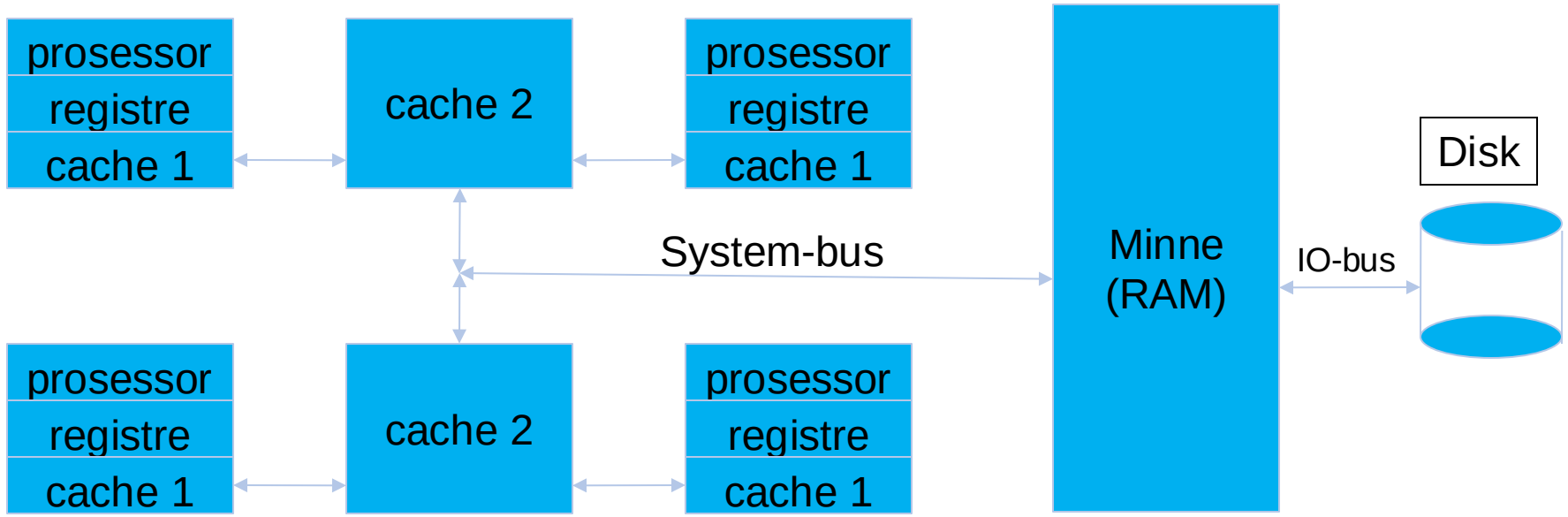


Datamaskinarkitektur (Computer Architecture)

3. Før:



4. Nå, f.eks:



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)

UNIVERSITETET I OSLO

Institutt for informatikk

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;

    Servitor(Restaurant rest, String navn) {

    }

    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

- Én viktig klasse og ett viktig interface i Javas API

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

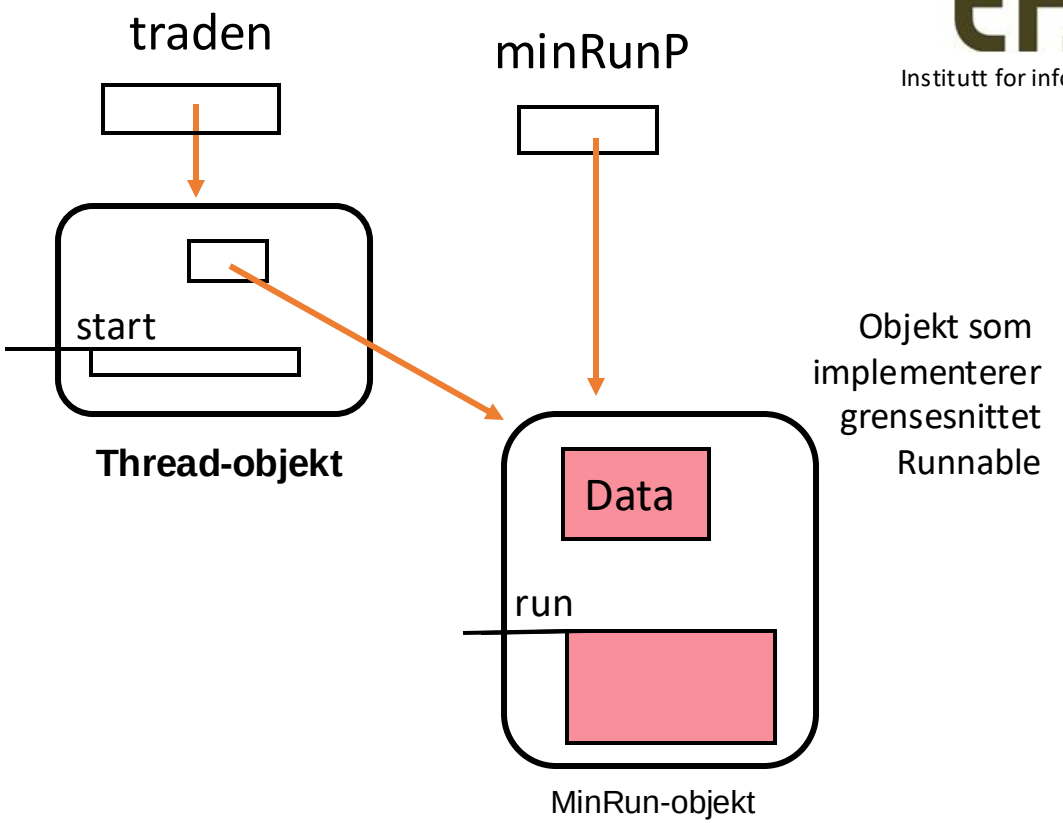
interface Runnable {
        void run();
}
```

Skriv det du ønsker tråden skal gjøre

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

Tråder i Java


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



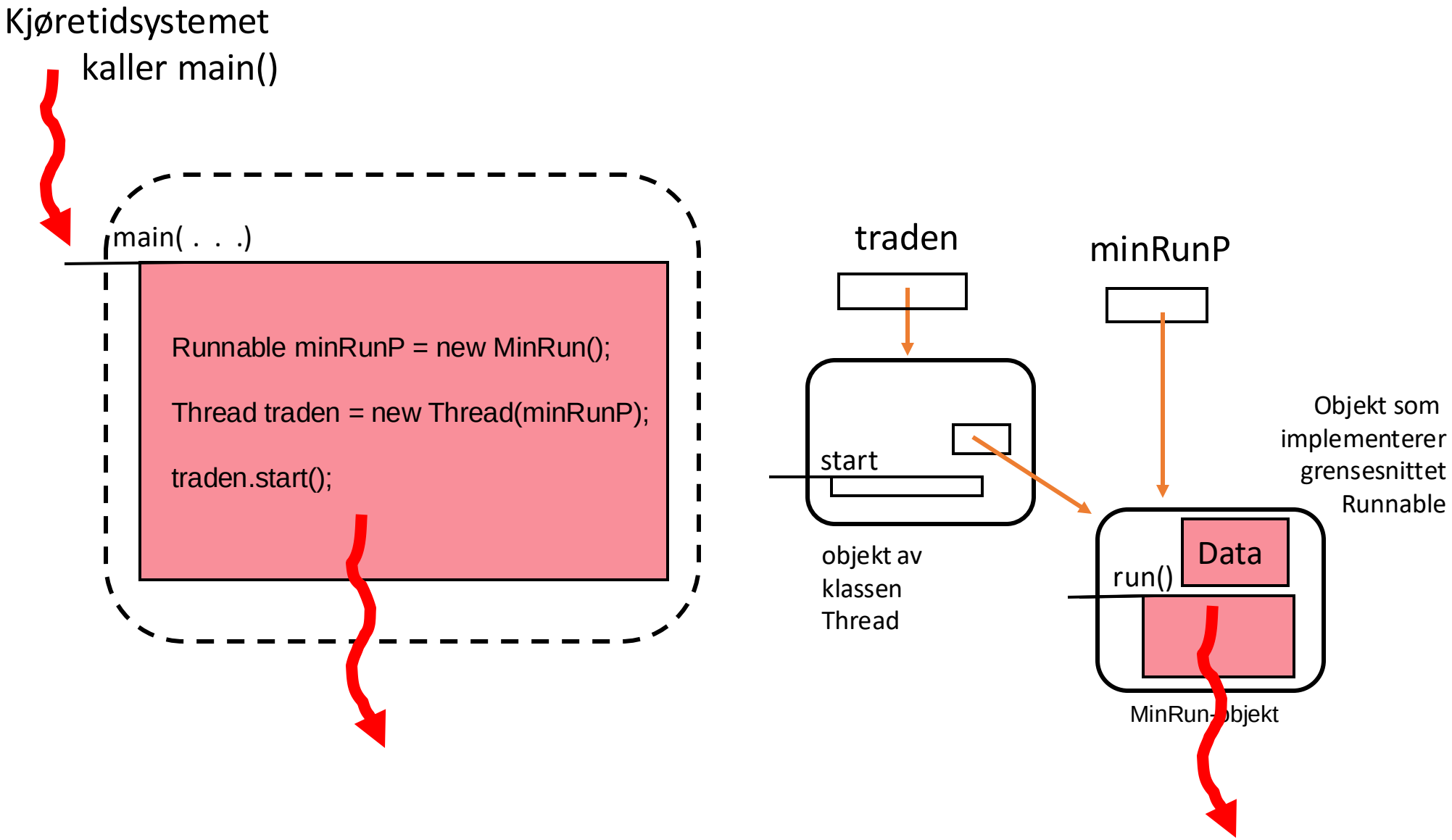
En tråd lages og startes opp slik (f.eks. i main()):

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



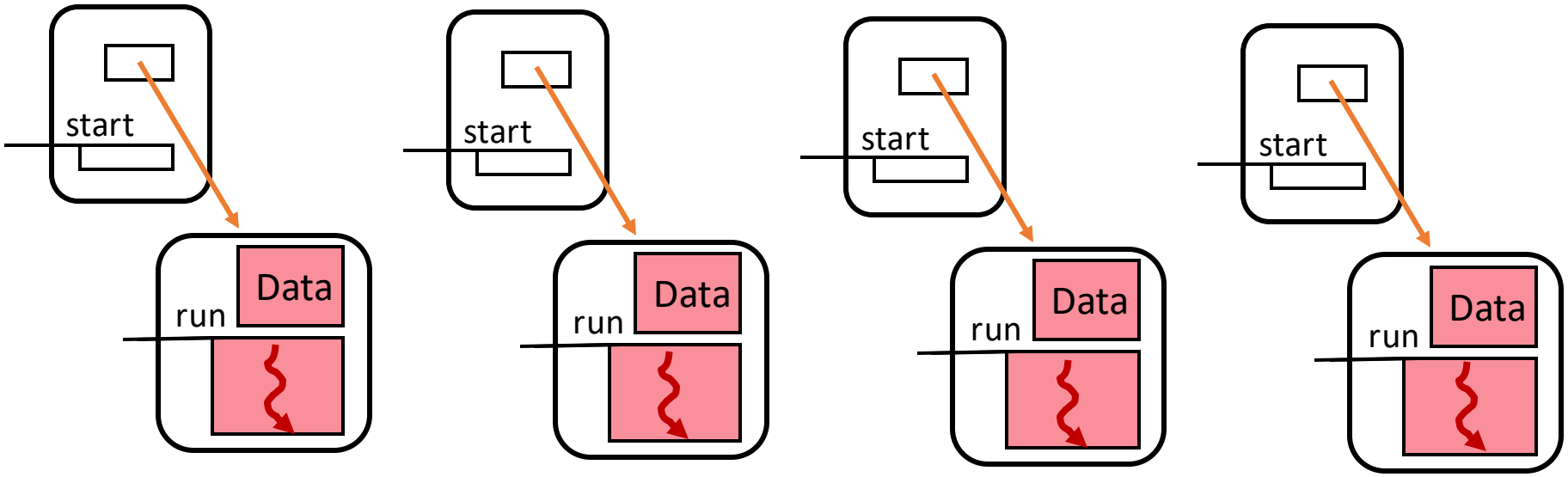
start( ) er en metode i Thread som må kalles opp for å få startet tråden. start-metoden vil igjen kalle metoden run (som vi selv programmerer).

Eksempel: main()-tråden starter én ny tråd



Eksempel:
main()-tråden starter 4 tråder

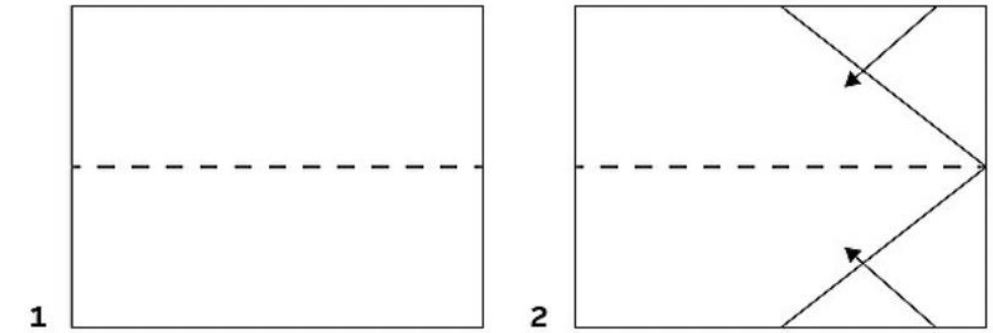
```
main( . . .)
    Runnable minRunP:
    Thread tråden;
    for (int ind= 1; ind <= 4; ind++) {
        minRunP = new MinRun();
        tråden = new Thread(minRunP);
        tråden.start();
    }
    . . .
```



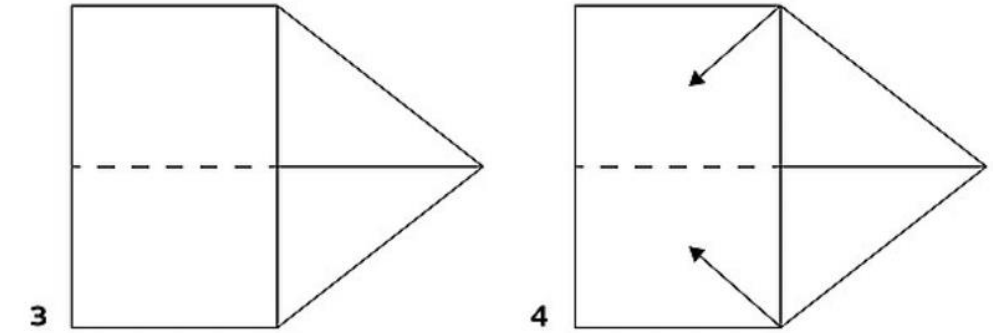
Fire steg for å lage et papirfly

Vi skal lage 4 papirfly i parallell

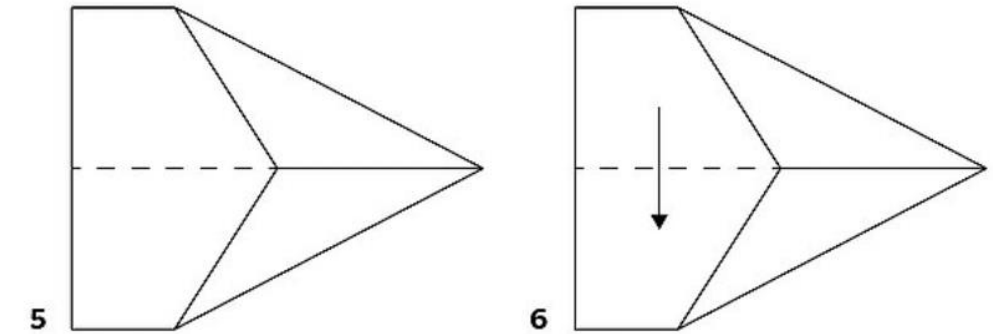
1. Parallellitet i bredde
2. Parallellitet i serie



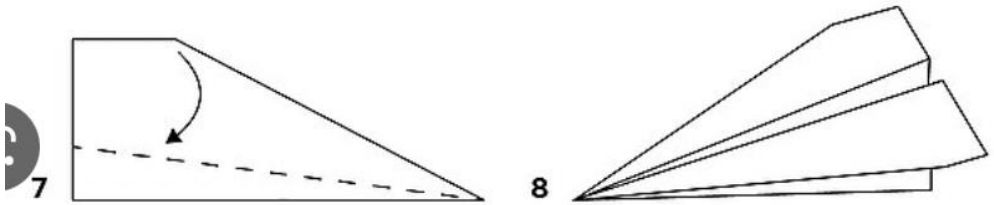
Steg 1



Steg 2



Steg 3

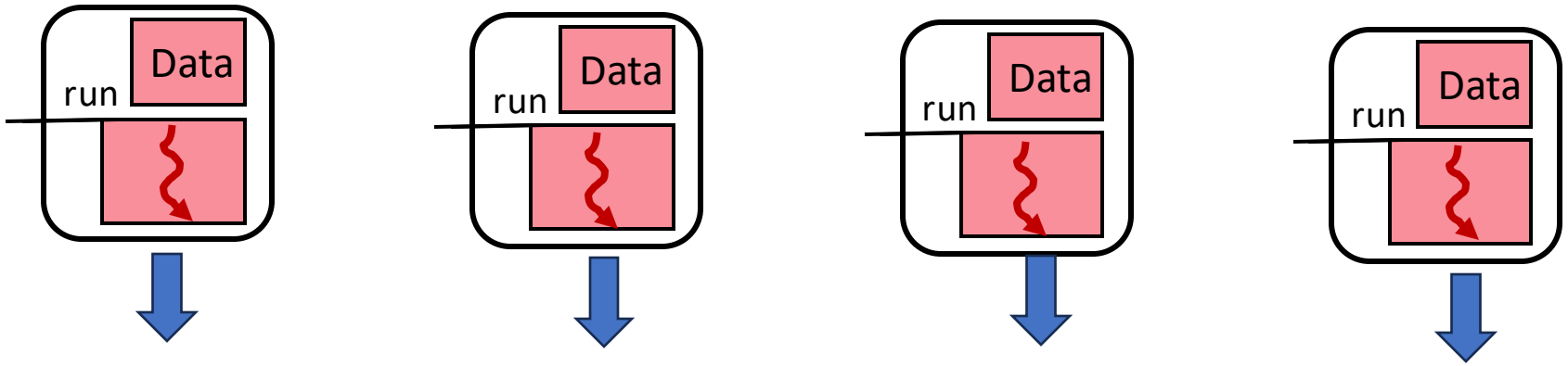


Steg 4

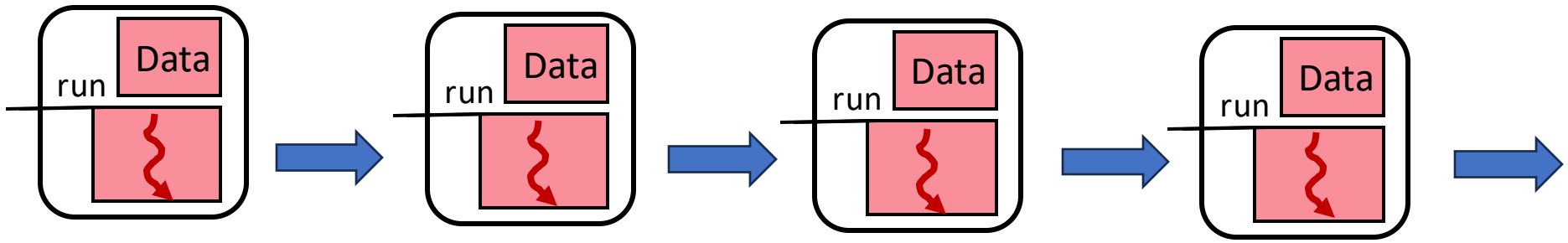
To hovedtyper parallellitet

- Parallellitet i bredden (alle gjør (nesten) det samme)

Mest vanlig i IN1010



- Sekvensiell prallellitet (samlebånd)



Sove

- En tråd kan sove et antall millisekunder; 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.”

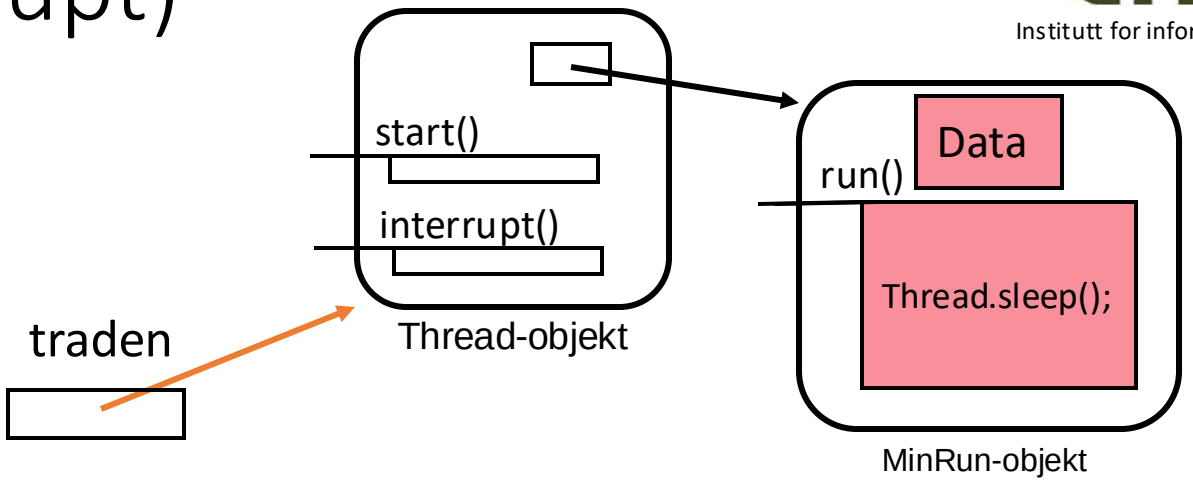
- Hvis noen avbryter denne tråden mens den sover skjer et unntak

```
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 avbrutt
  - 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 Stoppeklokke2 {
    public static void main (String[] arg) {
        Scanner tastatur = new Scanner(System.in);
        System.out.print("Trykk Return for å starte... ");
        tastatur.nextLine();

        Thread klokke = new Thread(new Klokke());
        klokke.start();

        System.out.print("Trykk Return for å stoppe...");
        tastatur.nextLine();
        klokke.interrupt();
        System.out.println("Takk for nå");
    }
}
```

```java
class Klokke implements Runnable {
    @Override
    public void run () {
        int tid = 0;
        while (true) {
            System.out.print(tid + " ");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {return;}
            tid++;
        }
    }
}
```

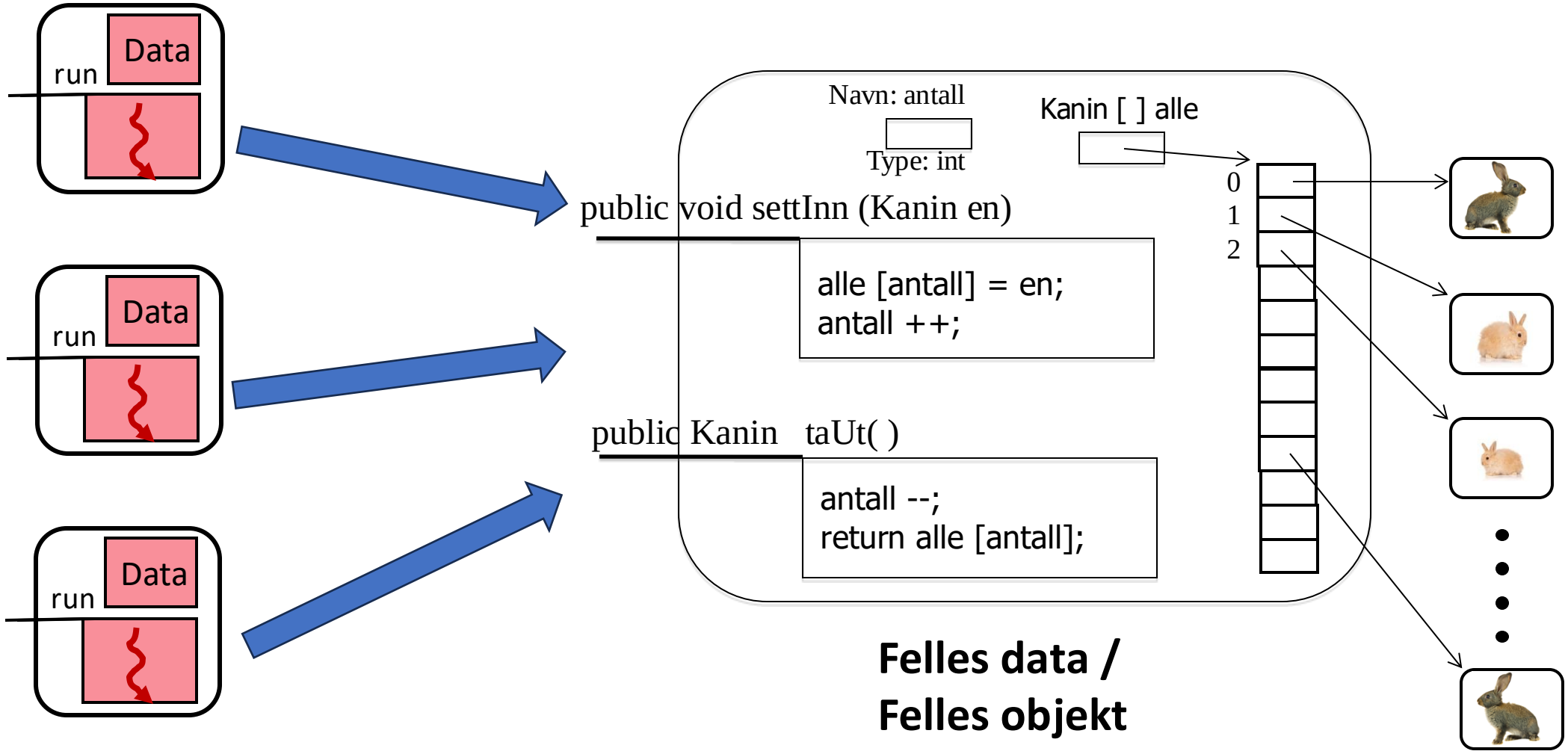
Kompiler og kjør denne koden selv. Gjør modifikasjoner.
Koden ligger på IN1010-siden
Her ligger også andre klokke-programmer

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

Hva skjer om flere tråder setter inn / og eller tar ut av en beholder samtidig?

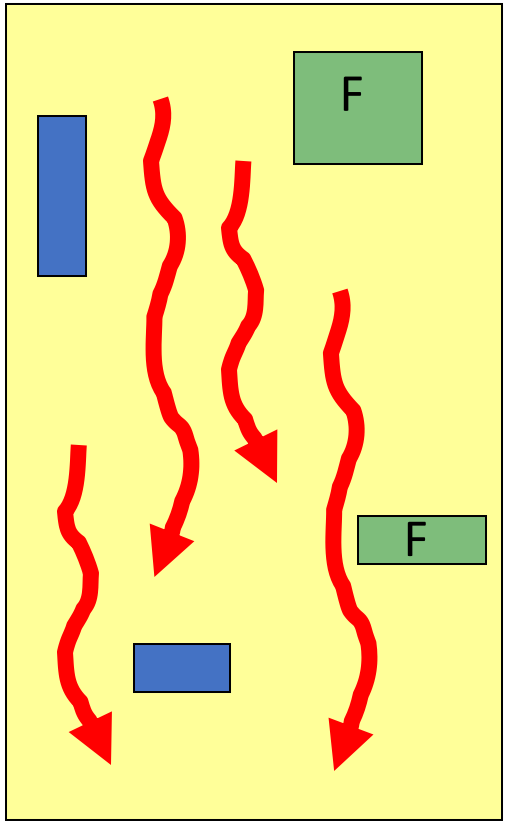


Felles data

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

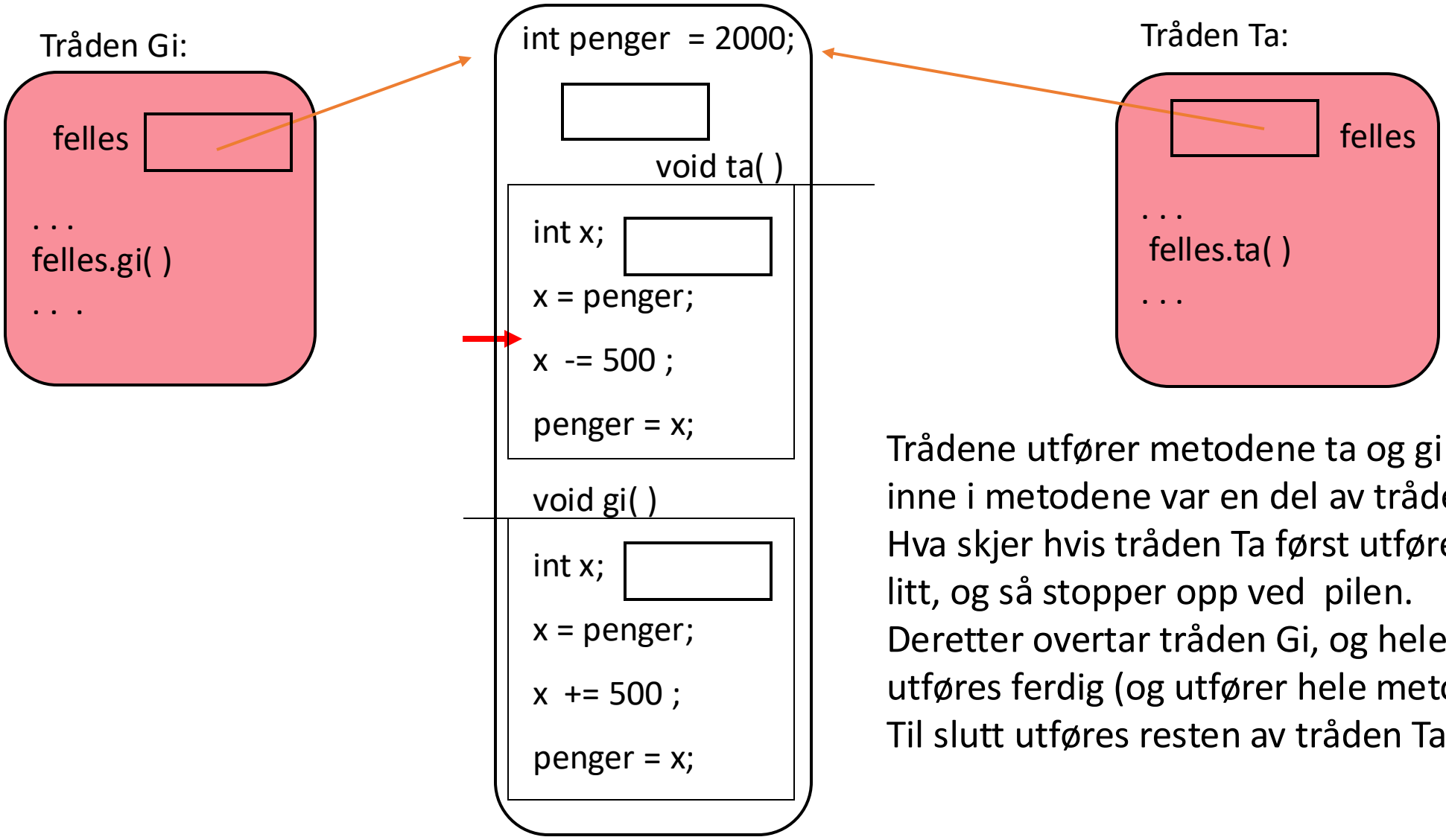
På figuren er det to områder vi har problemer med (dvs. at to eller flere tråder kan risikere å manipulere data i disse områdene samtidig). Disse to områdene er markert med F (for Felles). Kode som aksesserer felles data må skrives inne i en *kritisk region* (forklares snart).

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.



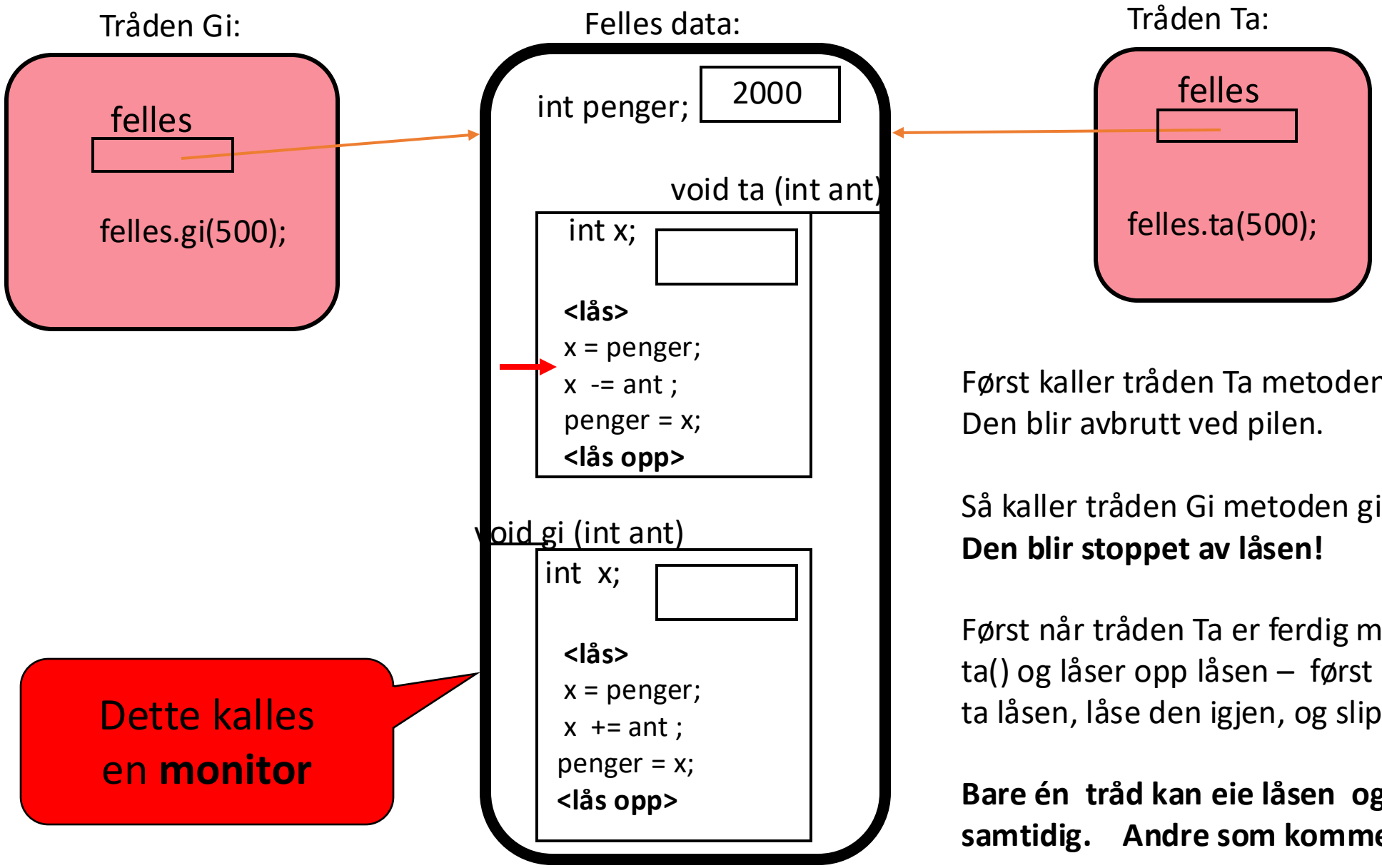
Vårt Java-program med 4 tråder

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 ta låsen, låse den igjen, og slippe inn i metoden gi().

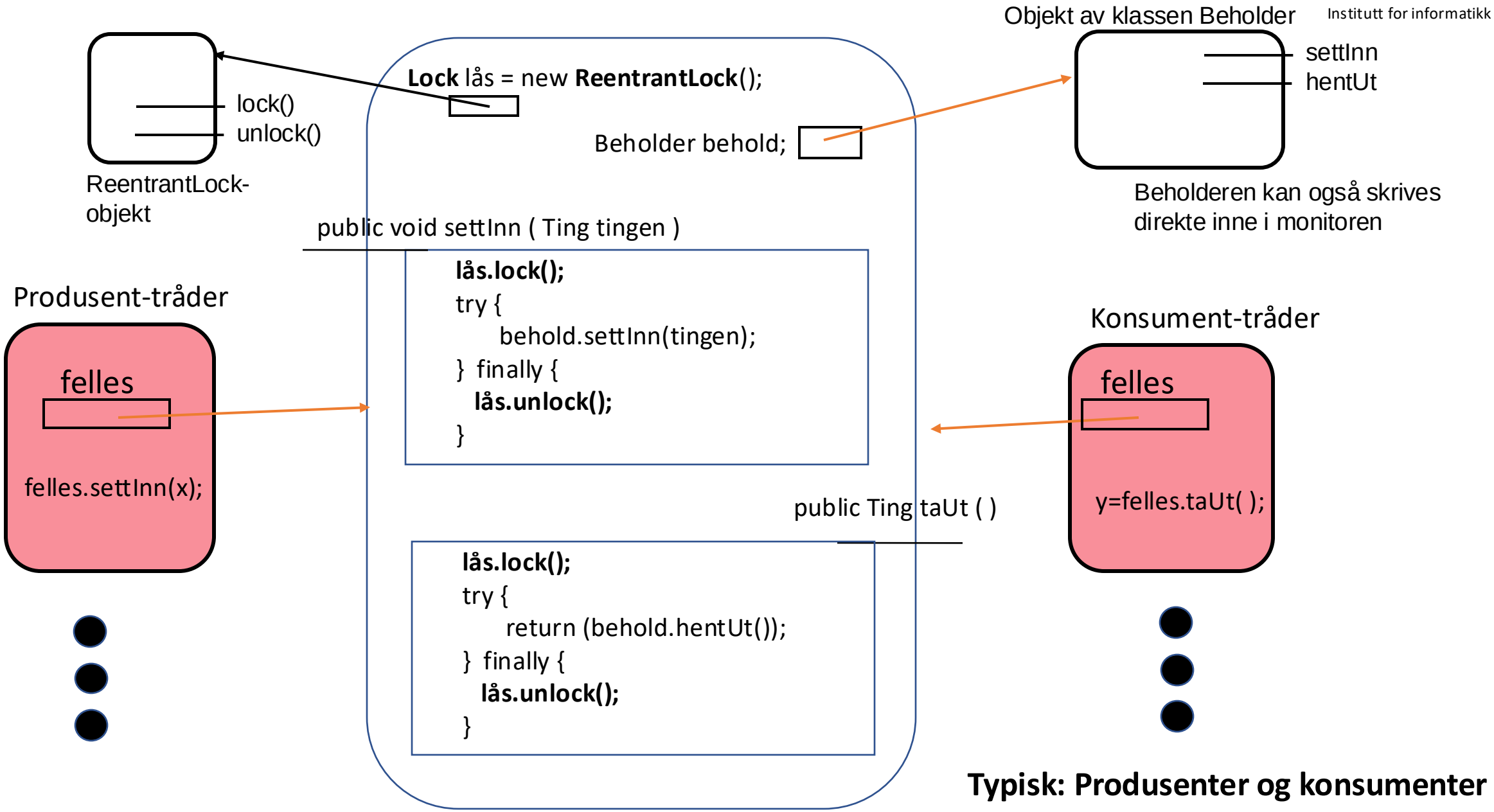
Bare én tråd kan eie låsen og de felles dataene samtidig. Andre som kommer må vente.

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** («Hoare monitor», CAR Hoare 1974)
- En monitor er **trådsikker**
- Til å lage metodene som kritiske regioner bruker vi en LÅS
  - Høyst en tråd er inne i en 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. cache og forskjellige kjerner).



En monitor som er/beskytter en beholder



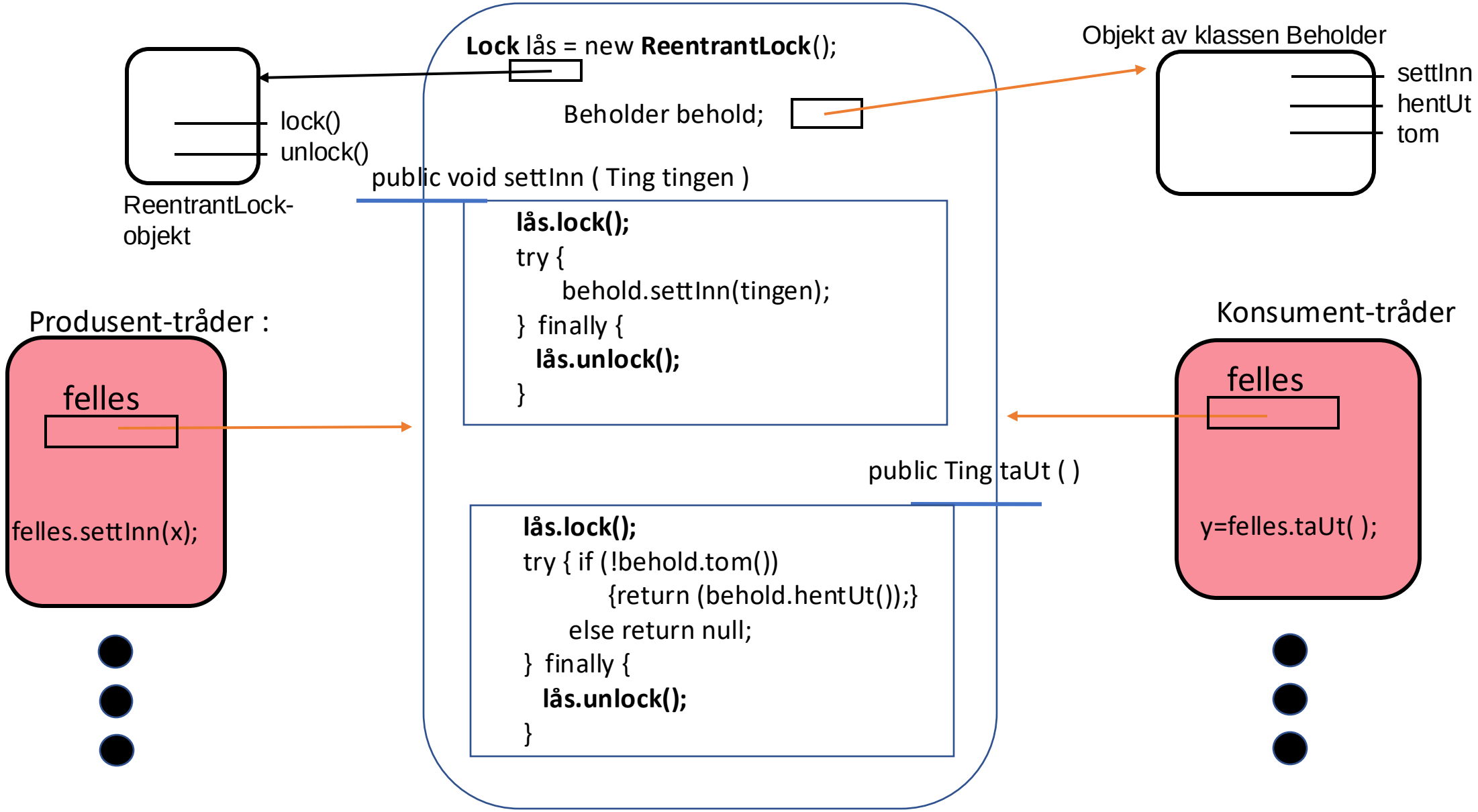
Typisk: Produsenter og konsumenter

Legg merke til bruken av finally


```
void  settInn (Ting tingen)  throws InterrupedException {
   lås.lock();
   try {
      :
      :
    }  finally {
     lås.unlock();
    }
}
```

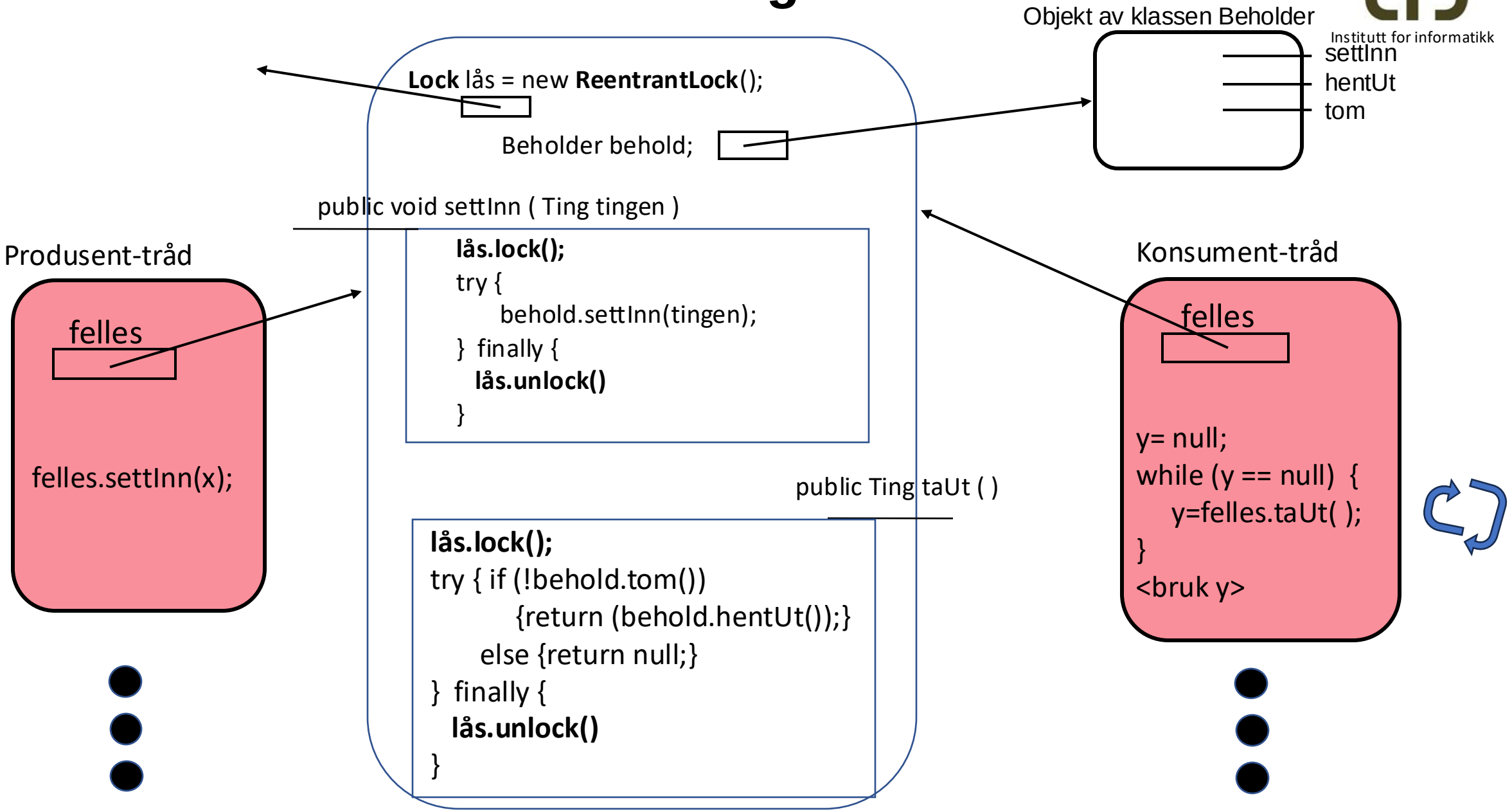
Da blir lås.unlock() **alltid** utført !!

Prøver å ta ut, men beholderen er tom



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

Aktiv venting



Passiv venting er bedre enn aktiv venting

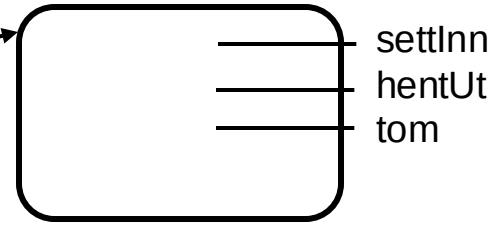
- Aktiv venting er vanligvis ikke smart
- Det bruker ressurser unødvendig
  - CPU-kraft
  - Monitoren blokkeres hver eneste gang tråden går inn i monitoren for å teste
- Bruk aktiv venting bare i spesielle tilfeller:
  - Få som kunkurrerer og korte kritiske regioner
  - Ødelegger (blokkerer) ikke for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

Passiv venting


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

Beholder behold;
```

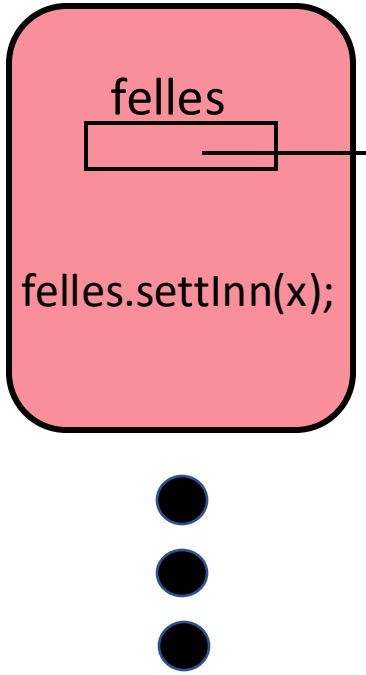
Objekt av klassen Beholder



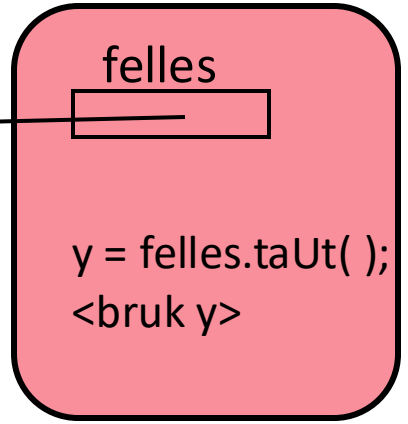

```
public void settInn ( Ting tingen )
    lås.lock();
    try {
        behold.settInn(tingen);
    }  finally {
      lås.unlock()
    }

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

Produsent-tråder



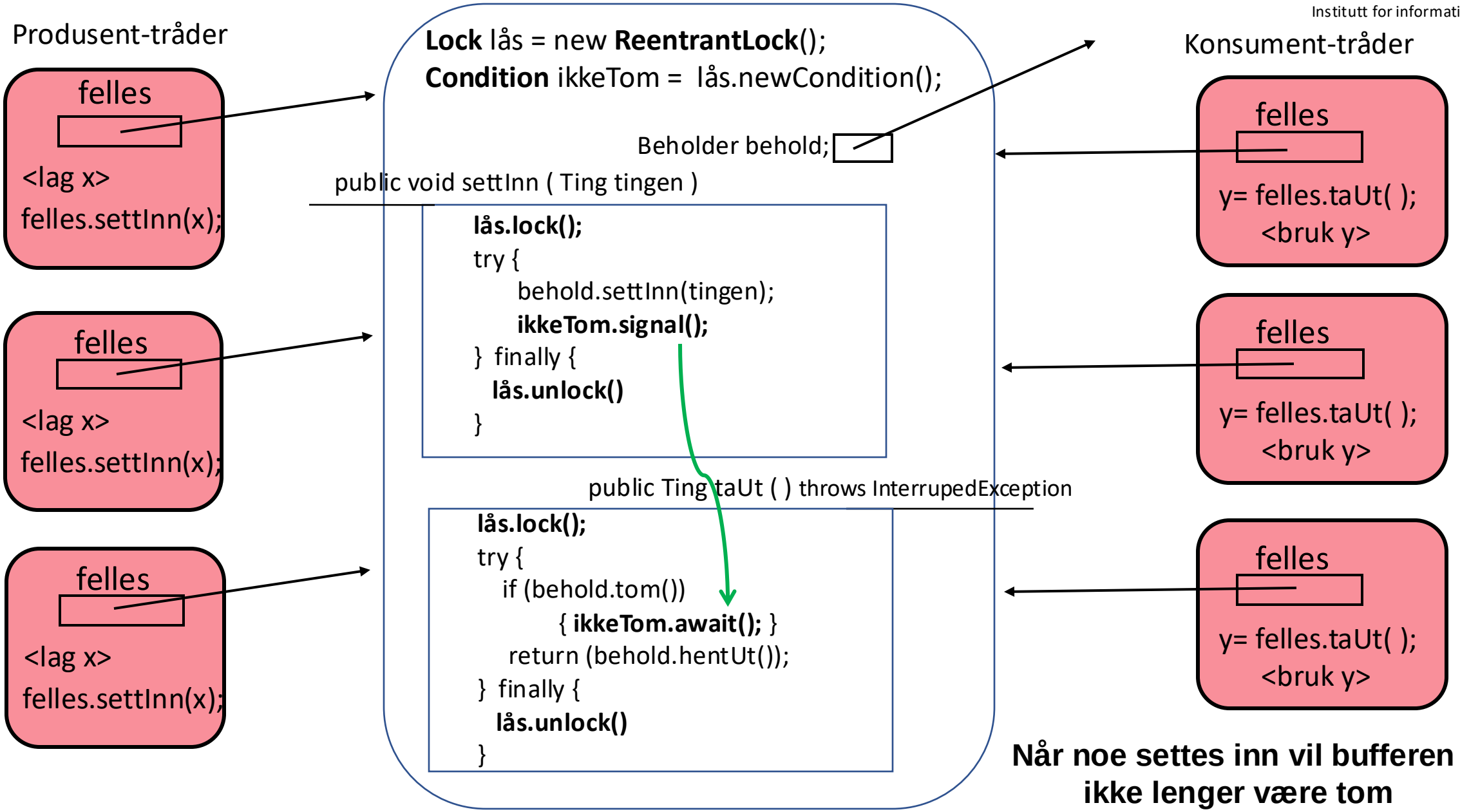
Konsument-tråder





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 lås = new **ReentrantLock**();

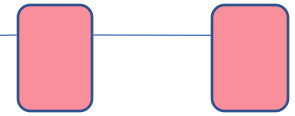
Condition ikkeTom = lås.newCondition();

void settInn (Ting tingen)

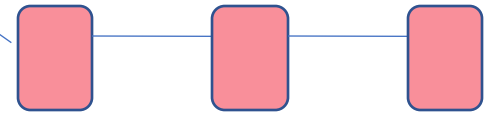
```
lås.lock();
try {
    behold.settInn(tingen);
    // det er lagt inn noe, så det er helt sikkert ikke tomt:
  ikkeTom.signal();
} finally {
  lås.unlock();
}
```

Tingen taUt ( ) throws InterrupedException

```
lås.lock();
try {
    if (behold.tom() ) ikkeTom.await();        ;
    return (behold.hentUt());
} finally {
  lås.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 lås = new ReentrantLock();
Condition ikkeFull =  lås.newCondition();
Condition ikkeTom = lås.newCondition();
```

```
void settInn ( Ting  tingen) throws InterrupedException
    lås.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 {
      lås.unlock();
    }
```

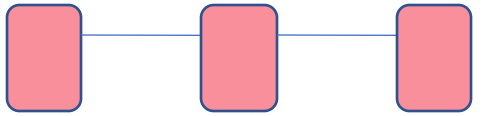
```
Tingen taUt ( )  throws InterrupedException
    lås.lock();
    try {
       if  (behold.tom())  ikkeTom.await();
       // nå er det helst sikkert ikke tomt;
      Tingen tmp = beholder.hentUt();
      // det er det tatt ut noe, så det er helt sikkert ikke fullt:
      ikkeFull.signal();
      return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



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



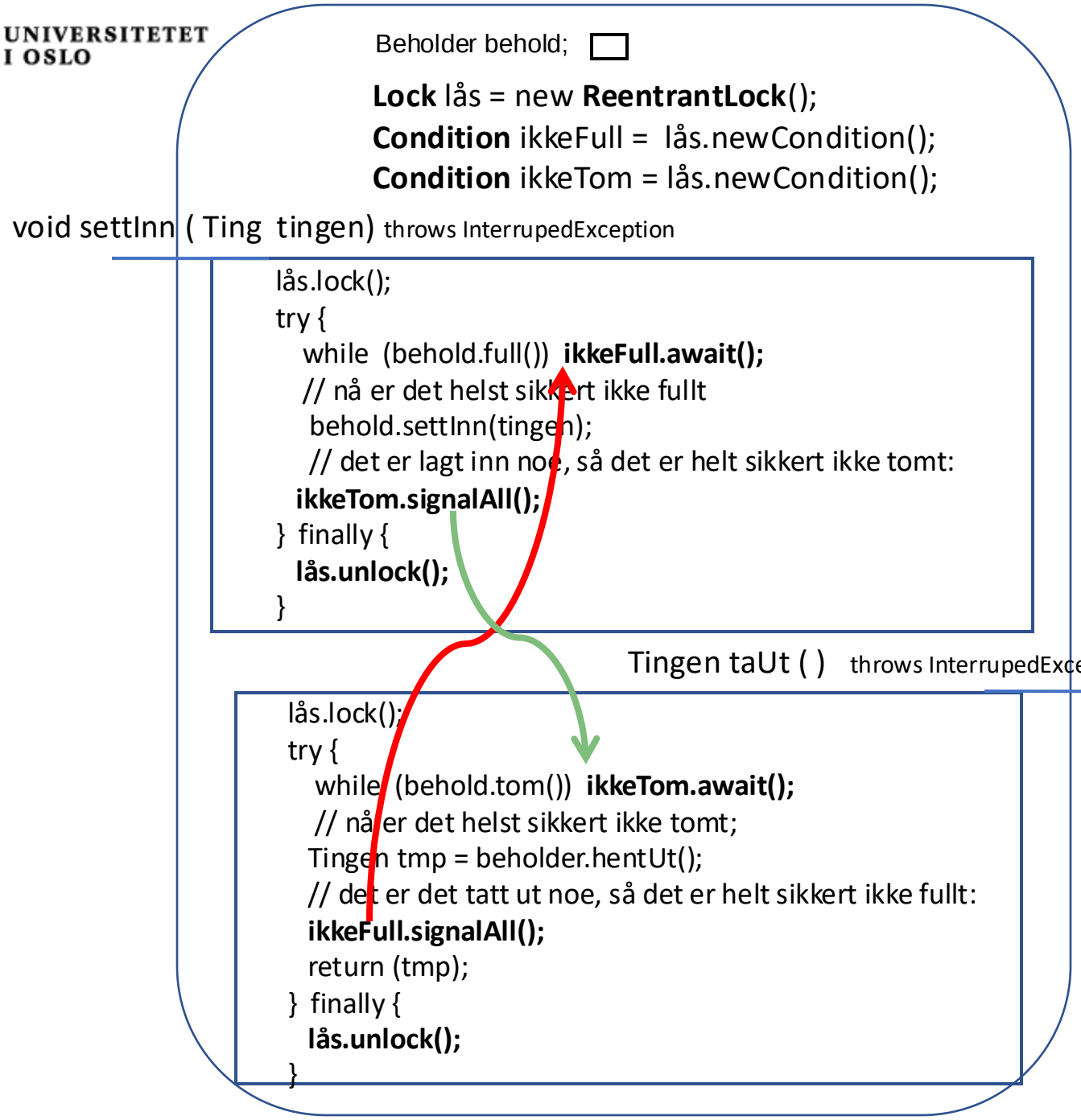
Tråder som venter på at bufferet ikke skal være fullt



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



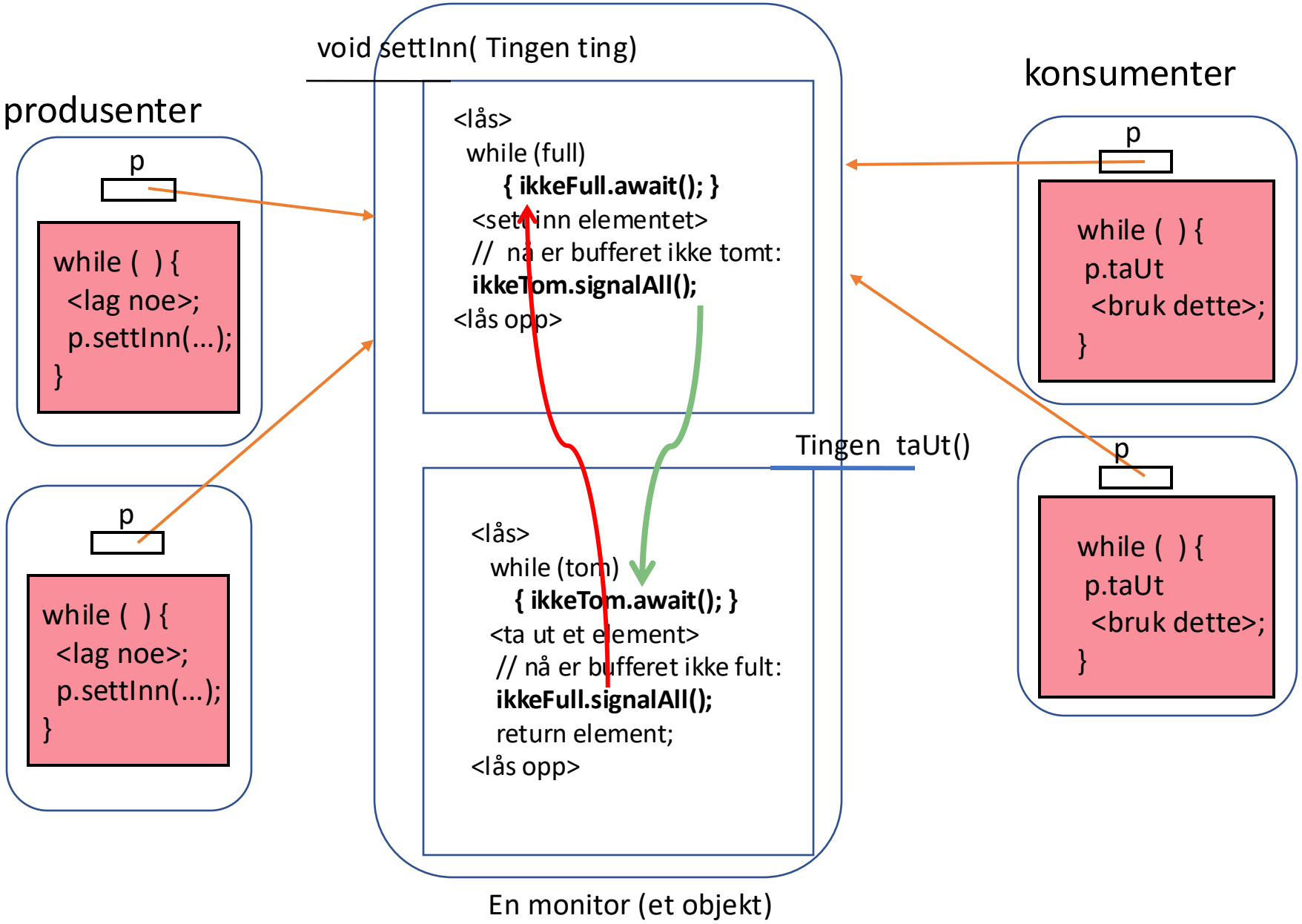
Her har vi skiftet ut if-testen i taUt() med en while-test.

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

Dette gjør programmet mer robust.

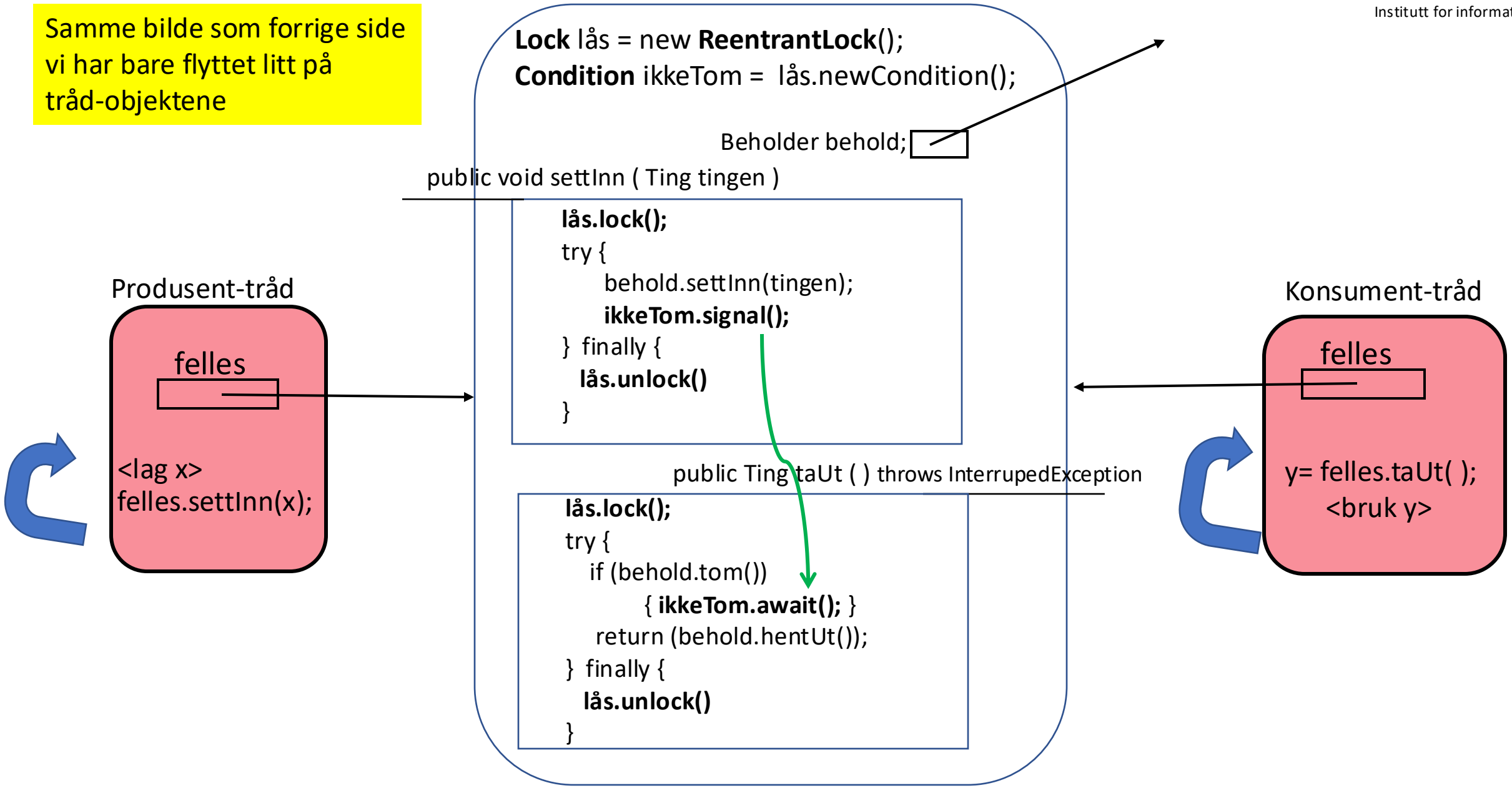
Se notatet om tråder.

Produsenter og konsumenter er et viktig eksempel



Produsent og konsument

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

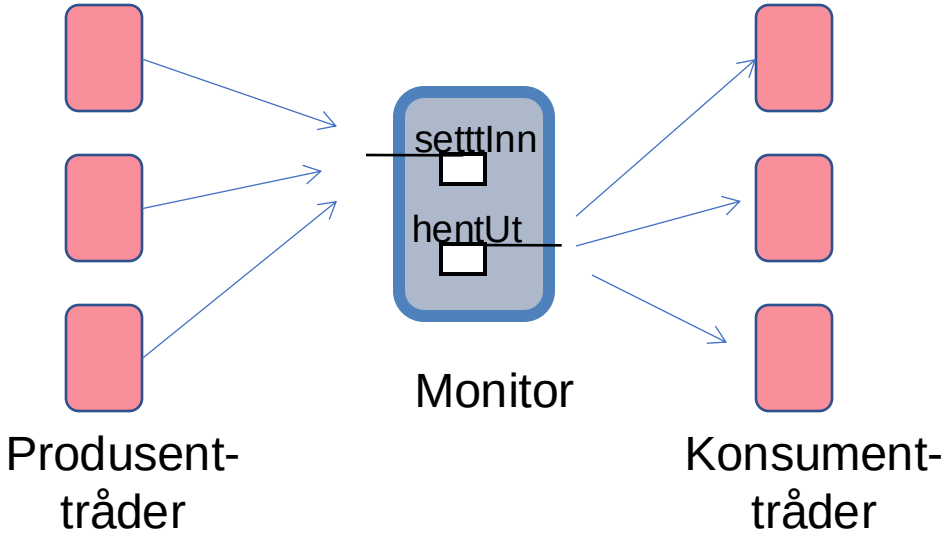


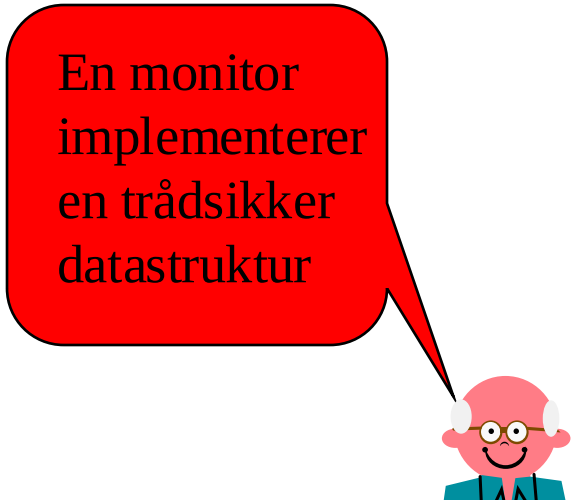
Hoved "take-away" i dag

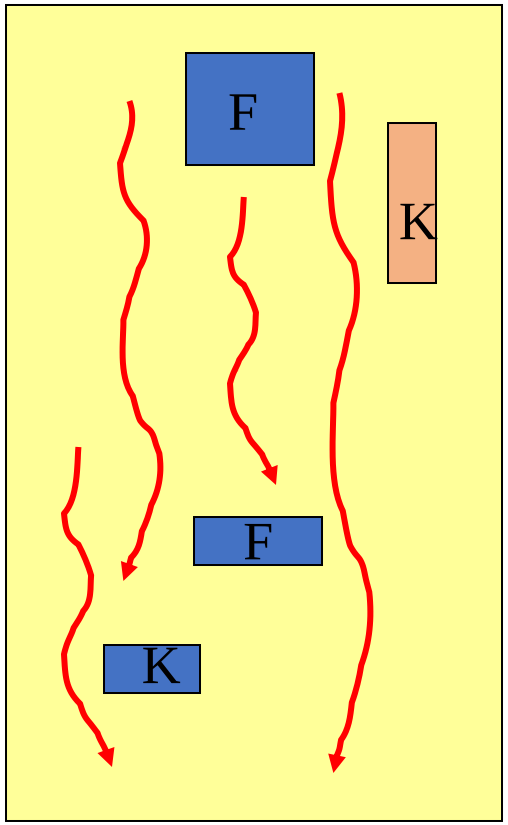
Du kan programmere parallelle aktiviteter i Java ved hjep av tråder.

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)
- Vi har lært hvordan tråder kommuniserer seg imellom ved hjelp av monitorer
- Vi har lært hvordan programmer venter i en monitor
  - og starter opp igjen de som venter
- 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.

Ekstrastoff

- Frekvenstabeller – eksempel (innlevering 5)
- Array av pekere til mange tråder
- Hvordan virker metodekall
- run()-metoder i subklasser av Thread
- Javas gamle ibeoende mekansimer for å lage monitorer.
- Eksempler på de to siste på IN1010-siden

Fletting av frekvenstabeller: De ti mest brukte navnene*

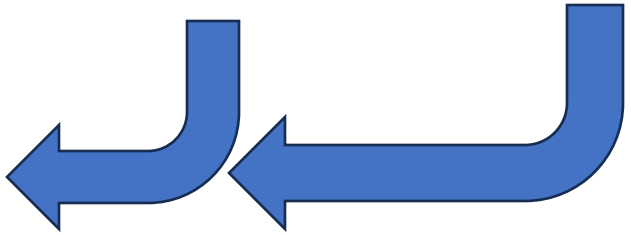
Institutt for informatikk

Norge	
Nora	1
Lukas	1
Oliver	1
Emma	1
Sofia	1
Ella	1
Emil	1
Jakob	1
Teodor	1
Lea	1

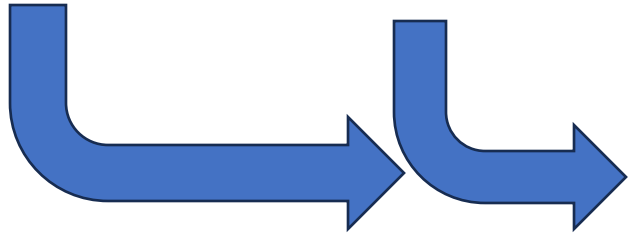
Sverige	
Astrid	1
Maja	1
William	1
Llam	1
Alma	1
Vera	1
Lea	1
Noha	1
Oliver	1
Lukas	1

Danmark	
Alma	1
Ida	1
Ella	1
Oscar	1
Oliver	1
Emil	1
Lukas	1
Emma	1
Nora	1
August	1

Finland	
Olivia	1
Leo	1
Väinö	1
Aada	1
Oliver	1
Lilja	1
Sofia	1
Ella	1
Emil	1
Toivo	1



Fllettes til en kombinert frekvenstabell



Fllettes til en kombinert frekvenstabell

Sverige og Norge	
Nora	1
Lukas	2
Oliver	2
Emma	1
Sofia	1
Ella	1
Emil	1
Jakob	1
Teodor	1
Lea	2
Astrid	1
Maja	1
William	1
Llam	1
Alma	1
Vera	1
Noha	1

Danmark og Finnland	
Alma	1
Ida	1
Ella	2
Oscar	1
Oliver	2
Emil	2
Lukas	1
Emma	1
Nora	1
August	1
Olivia	1
Leo	1
Väinö	1
Aada	1
Lilja	1
Sofia	1
Toivo	1

* Ikke helt pålitelige data

UNIVERSITETET I OSLO

Fletting av frekvenstabeller: De ti mest brukte navnene

Institutt for informatikk

Sverige og Norge	
Nora	1
Lukas	2
Oliver	2
Emma	1
Sofia	1
Ella	1
Emil	1
Jakob	1
Teodor	1
Lea	2
Astrid	1
Maja	1
William	1
LIam	1
Alma	1
Vera	1
Noha	1

Danmark og Finnland	
Alma	1
Ida	1
Ella	2
Oscar	1
Oliver	2
Emil	2
Lukas	1
Emma	1
Nora	1
August	1
Olivia	1
Leo	1
Väinö	1
Aada	1
Lilja	1
Sofia	1
Toivo	1

Fllettes til en kombinert frekvenstabell

Sverige, Norge, Danmark og Finnland

Nora	2
Lukas	3
Oliver	4
Emma	2
Sofia	2
Ella	3
Emil	3
Jakob	1
Teodor	1
Lea	2
Astrid	1
Maja	1
William	1
LIam	1
Alma	1
Vera	1
Noha	1
Alma	1
Ida	1
Oscar	1
August	1
Olivia	1
Leo	1
Väinö	1
Aada	1
Lilja	1
Toivo	1

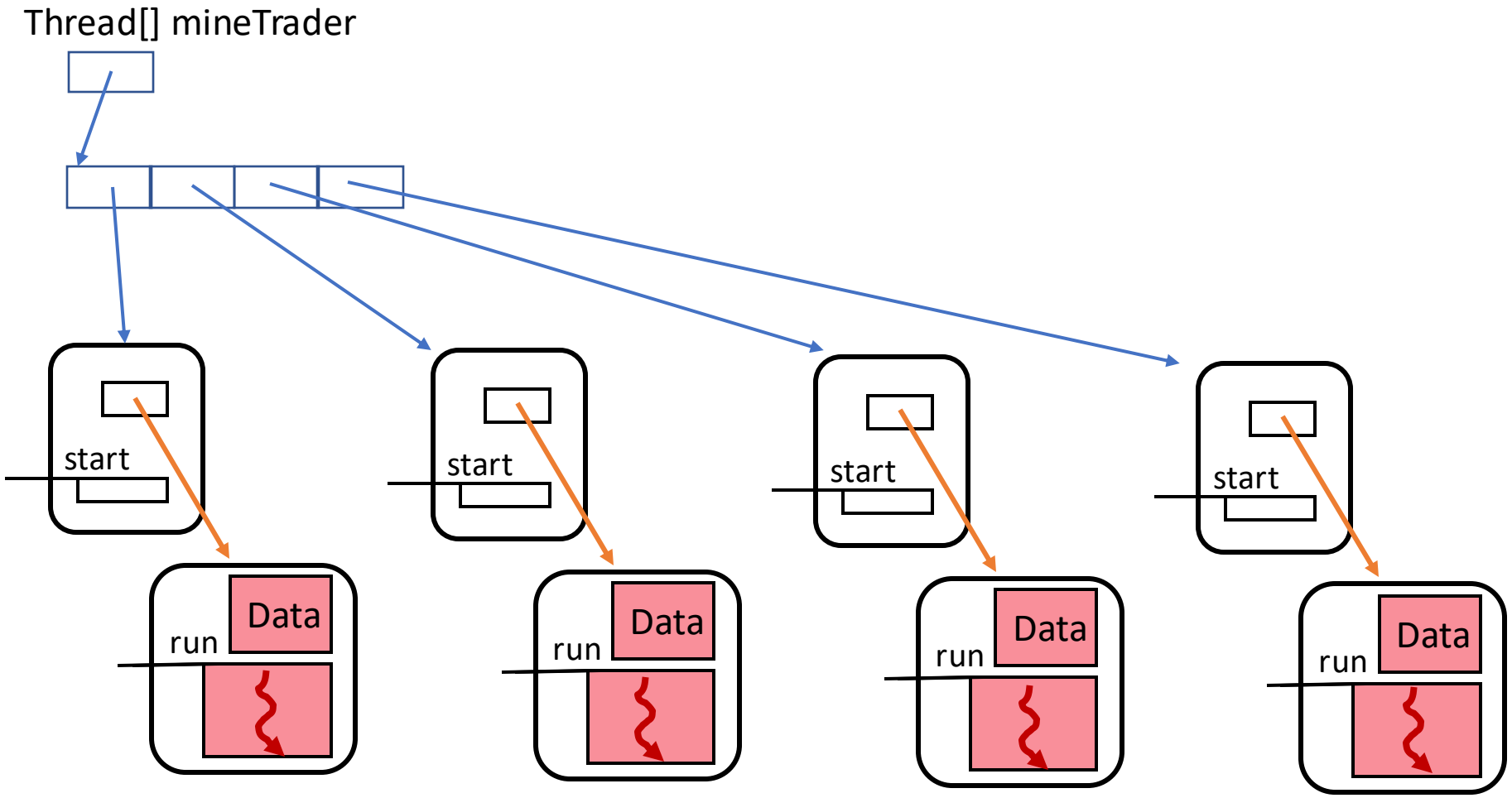
Det vil bl.a. si:

Det er bare ett navn som er blant de ti mest brukte i alle de fire landene, og det er Oliver

Tre navn er blant de ti mest brukte i tre av landene (Lukas, Ella og Emil)

Osv.

Noen ganger er det greit å ha referanser til alle trådene



Hva skjer når run() kaller andre metoder

f.eks. to tråder:

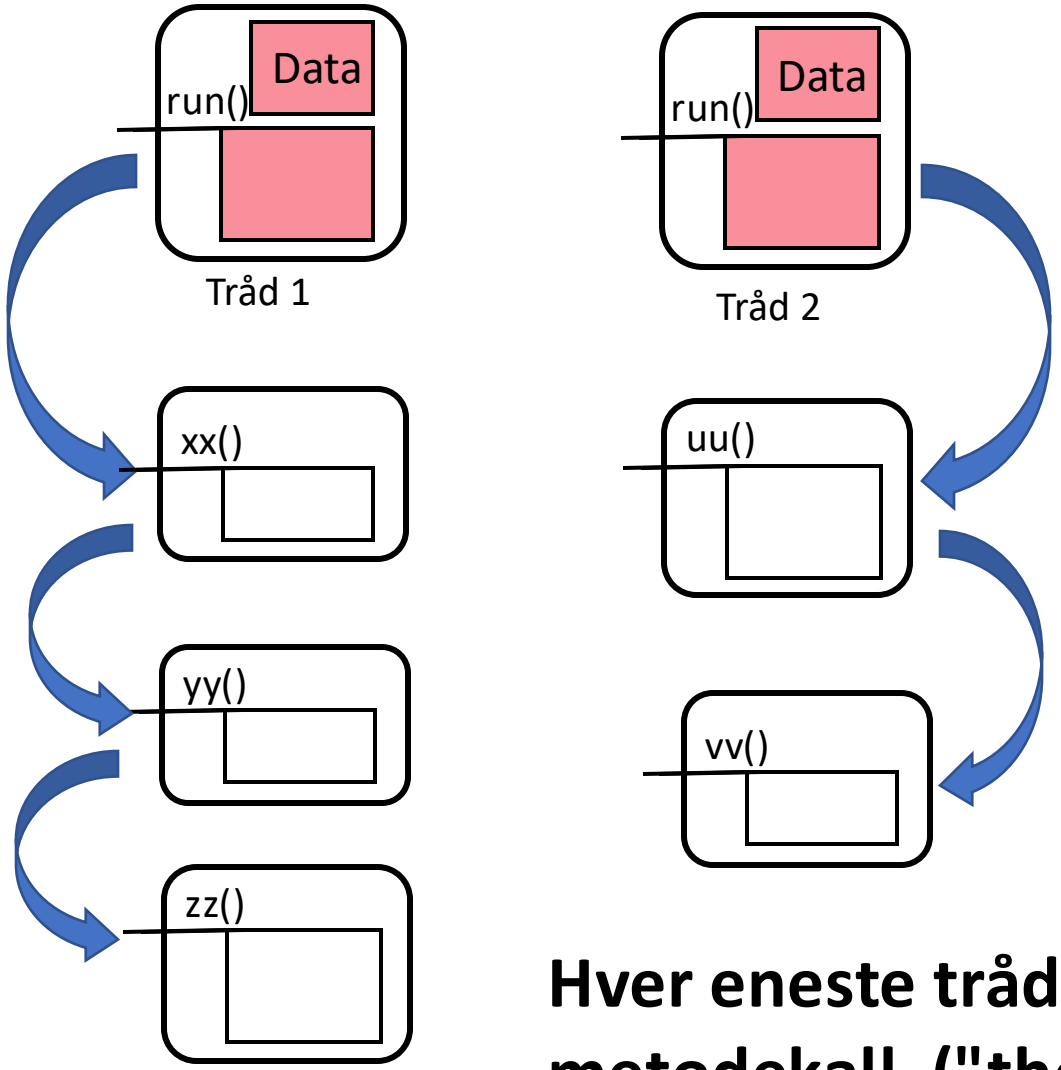
En stabel/stakk av metodekall:

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)

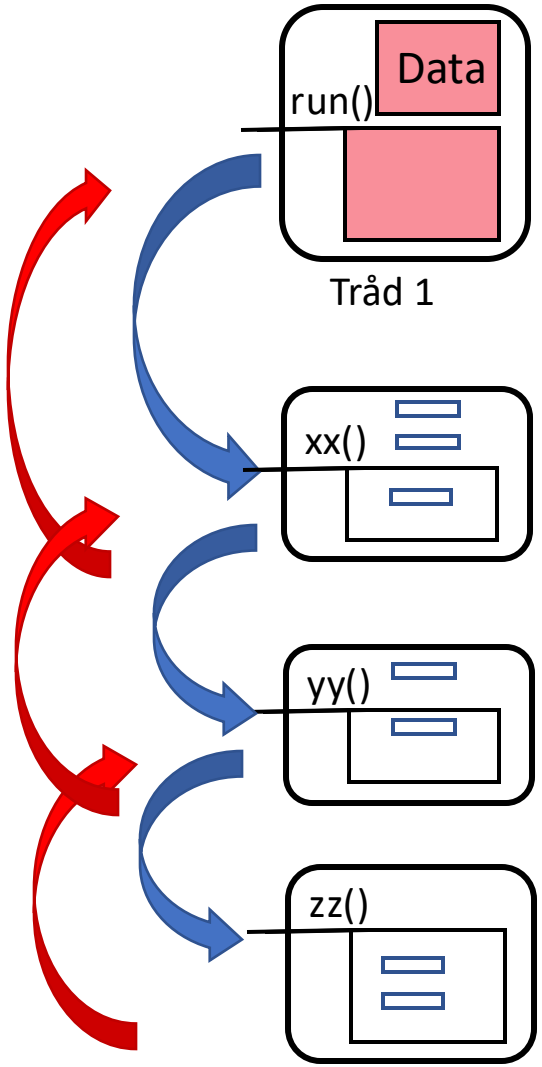
En stabel/stakk av metodekall:

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 20. januar (og det samme i Python og alle andre programmeringsspråk).

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

Et større litt større eksempel – kokk og servitør

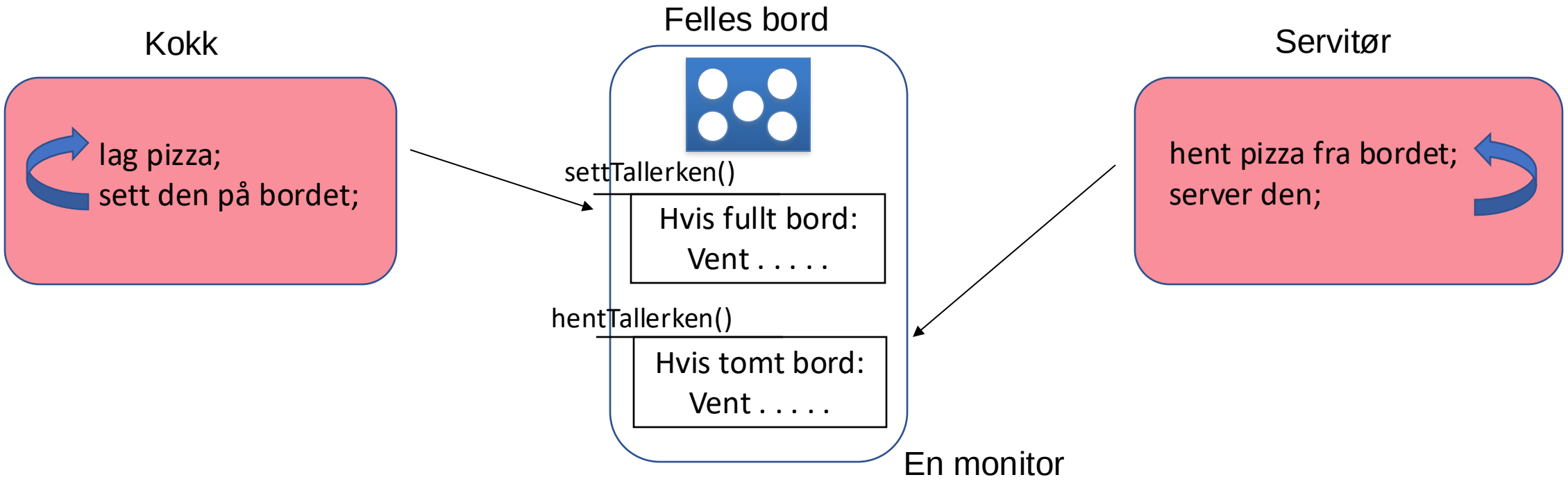
- Kokken lager mat og setter en og en tallerken på et bord
- Servitøren tar en og en tallerken fra bordet og serverer
- Kokken må ikke sette mer enn BORD_KAPASITET tallerkener på bordet (maten blir kald)
- Servitøren kan selvsagt ikke servere mat som ikke er laget (bordet er tomt)

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

- Tre varianter på IN1010-siden
  - Restaurant.java — Pensum i IN1010
  - RestaurantJavaOld.java — Ikke Pensum i IN1010 (Horstmann: Special Topic 20.2)
  - RestaurantSubklassing.java — Ikke Pensum i IN1010 (Horstmann: Programming Tip 20.1)

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:

Kode på IN1010-siden:

Restaurant. java
RestaurantJavaOld.java
RestaurantSubklassing.java

```
steing@MacBookPro programmer % java Restaurant 5 20
Kokken lager tallerken nr: 1
Antall på bordet: 1
Kelner serverer nr:1
Kokken lager tallerken nr: 2
Antall på bordet: 1
Kokken lager tallerken nr: 3
Antall på bordet: 2
Kokken lager tallerken nr: 4
Antall på bordet: 3
Kelner serverer nr:2
Kokken lager tallerken nr: 5
Antall på bordet: 3
Kelner serverer nr:3
Kokken lager tallerken nr: 6
Antall på bordet: 3
Kokken lager tallerken nr: 7
Antall på bordet: 4
Kelner serverer nr:4
Kokken lager tallerken nr: 8
Antall på bordet: 4
Kokken lager tallerken nr: 9
Antall på bordet: 5
Kelner serverer nr:5
Kokken lager tallerken nr: 10
Antall på bordet: 5
Kokken lager tallerken nr: 11
Kelner serverer nr:6
Antall på bordet: 5
Kokken lager tallerken nr: 12
```

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.
  - En god måte, og slik det gjøres i Horstmann (presentert i dag)
  - Slik det gjøres innebygget i Java (notify og wait, ikke pensum i IN1010)
  - Se neste sider, programeksemplene og "Notat om tråder" på IN1010-siden

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

Om å teste parallelle programmer

Et vanlig sekvensielt program er (vanligvis) **deterministisk**
Det gir samme resultat hver gang du kjører det

Parallelle programmer er **ikke-deterministiske**
Neste umulig å teste parallelle programmer fordi de oppfører seg forskjellig fra gang til gang.
Du får forskjellige (tidsavhengige) svar.
En feil kan materialisere seg først etter en million kjøringer
Jfr. Stoppeklokke3.java

Tror du programmet ditt er riktig ?
Hvis ja: Begrunn hvorfor det er riktig ?
Du må resonere om hvorfor programmet er riktig (ikke teste)

Testing viser bare feil – ikke fravær av feil