Parallellitet og tråder (del 1)

IN1010, 24. mars 2026

Magnus Skjegstad

(basert på materiale fra Stein Gjessing)

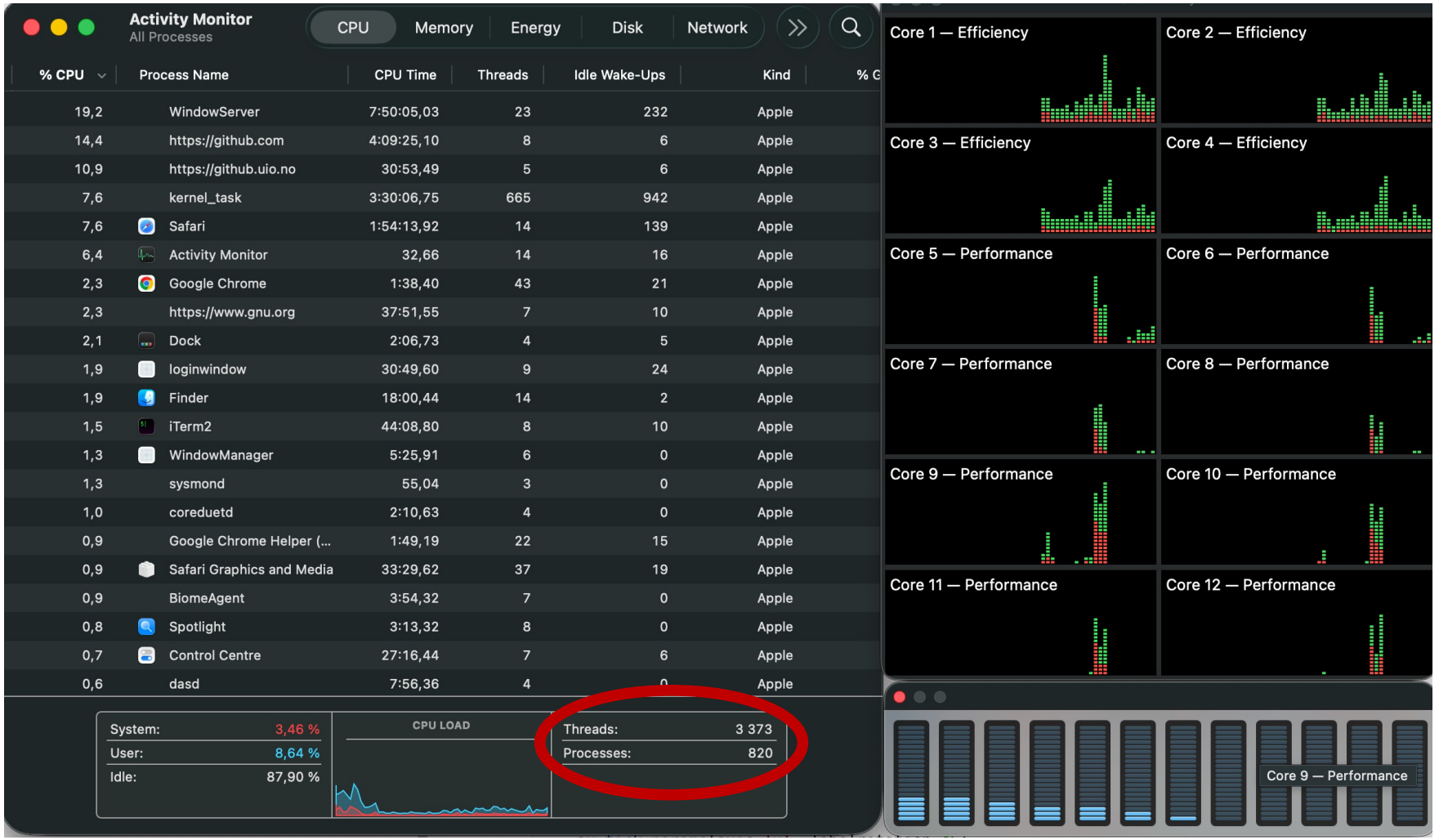
Horstmann kap 20.1 – 20.3 («Tråder i Java»)

Planen for i dag

- Litt bakgrunn først
  - Hva er parallelle programmer og hvorfor trenger vi dem?
  - Hvordan kan dette skje på én kjerne?
  - Hvordan kan dette skje på mange kjerner?
  - 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 kritiske data (løsning: kritiske regioner og monitorer)
    - Trådsikre datastrukturer

Hvorfor parallelle programmer?

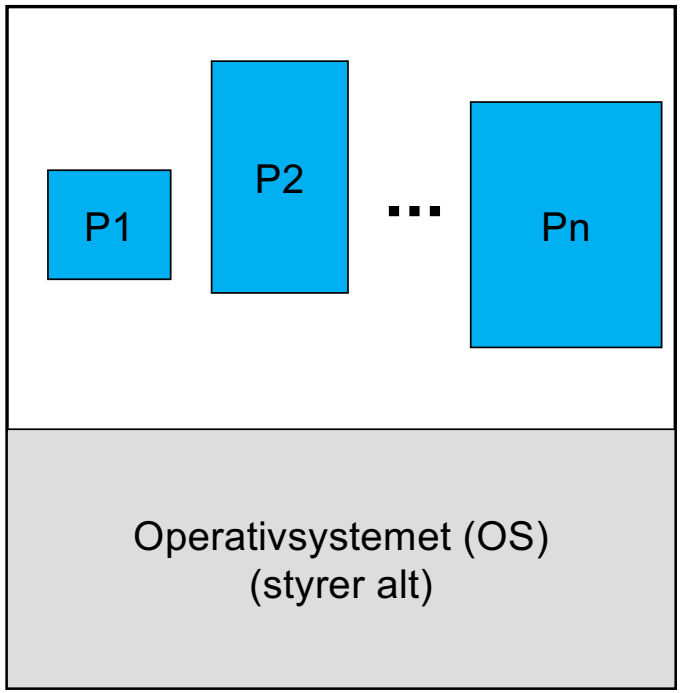
- **Naturlig parallellitet** i oppgaven
  - Maskinen må utføre administrative oppgaver samtidig (*lagre en fil, vise noe på skjermen, sende noe over nettverket*)
  - ... eller kjøre flere programmer samtidig for brukeren (*spille musikk og kjøre Java-kompilator mens man skriver programmer i vscode*)
  - Vente på at noe blir ledig / tilgjengelig (nettverk, filer etc)
  - Oppgaver som naturlig modelleres i parallelle deler
    - Bredde-parallellitet, samlebånd-parallellitet
- **Ekte parallellitet – hastighetsøkning**
  - Tunge beregningsoppgaver som ellers tar for langt tid å løse
  - Ekte parallellitet: CPU med flere kjerner (eller flere CPUer)
  - Multicore, dual-core, 8-core, n-core...



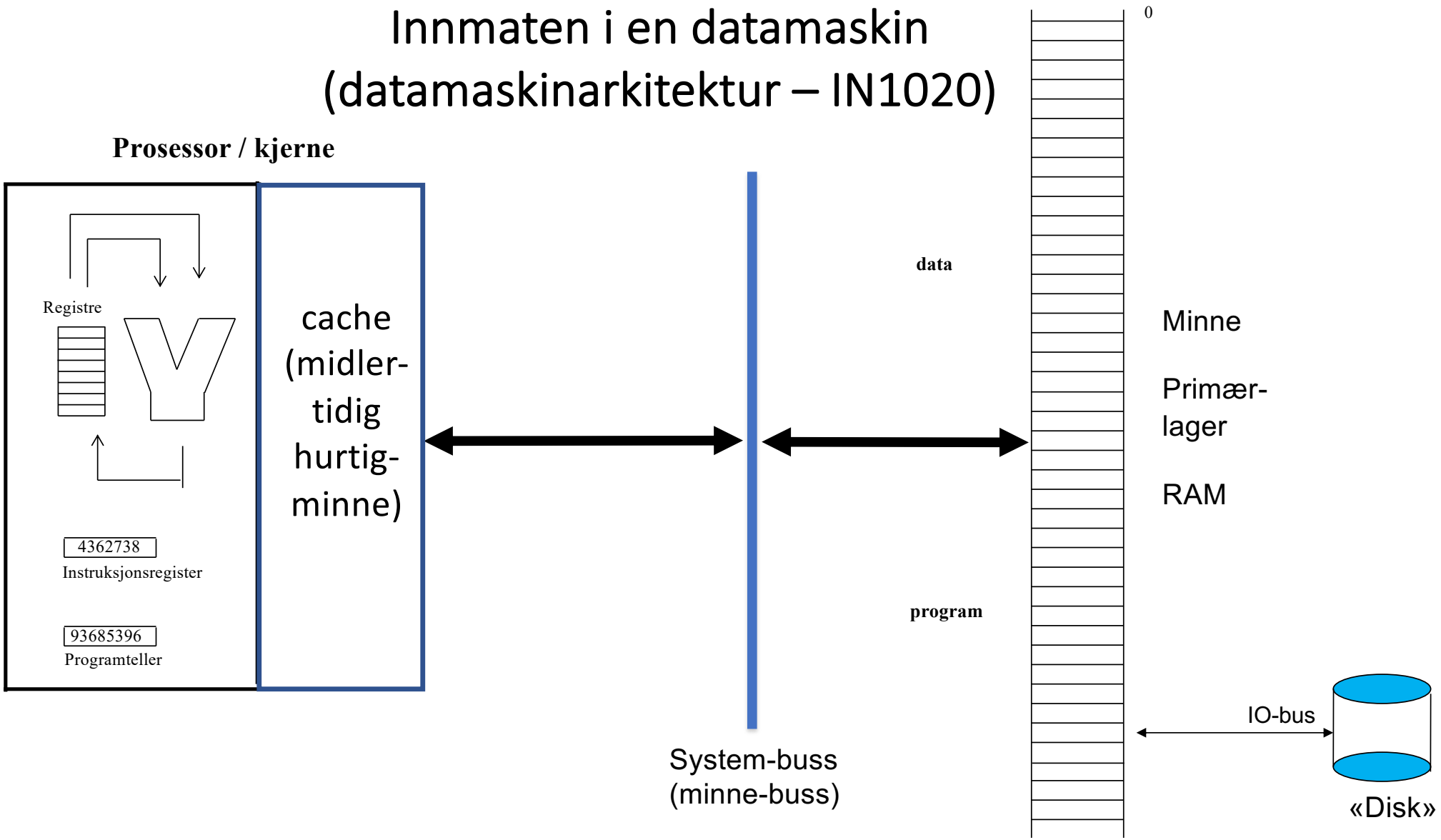
Maskin med 12 kjerner

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.



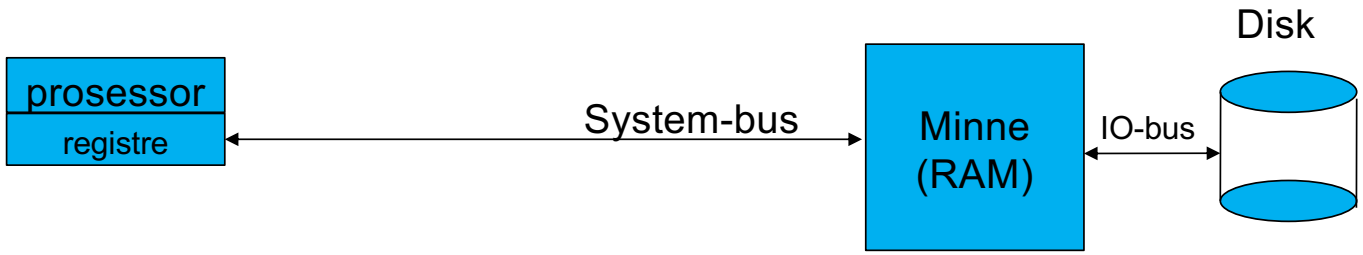
Innmaten i en datamaskin (datamaskinarkitektur – IN1020)



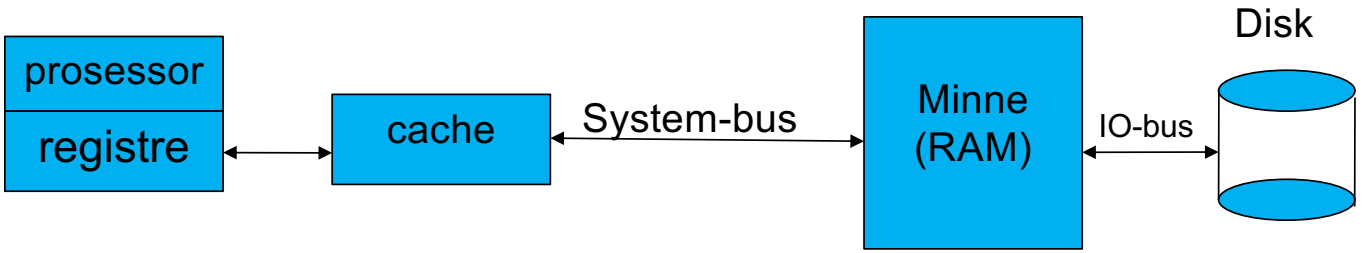
Om ”Dual-core”, "8-core", ”N-core” Ekte parallellitet (neste side)

Datamaskinarkitektur (Computer Architecture)

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



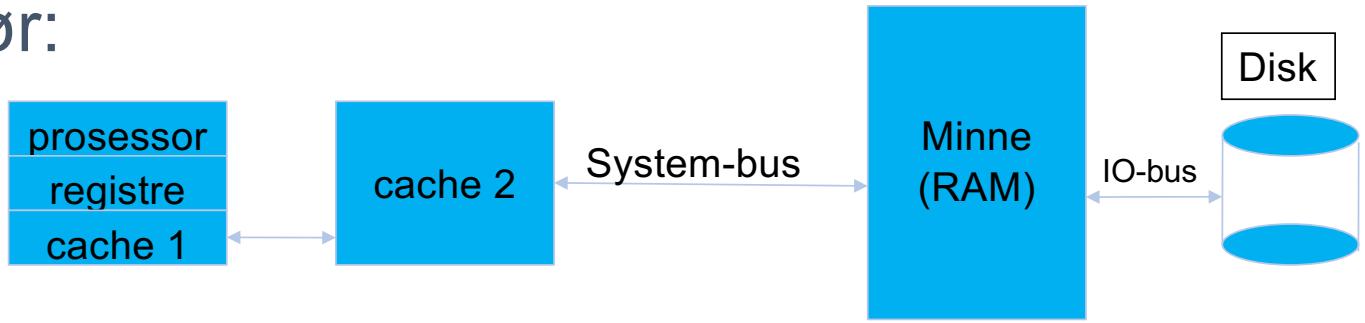
2. Før:



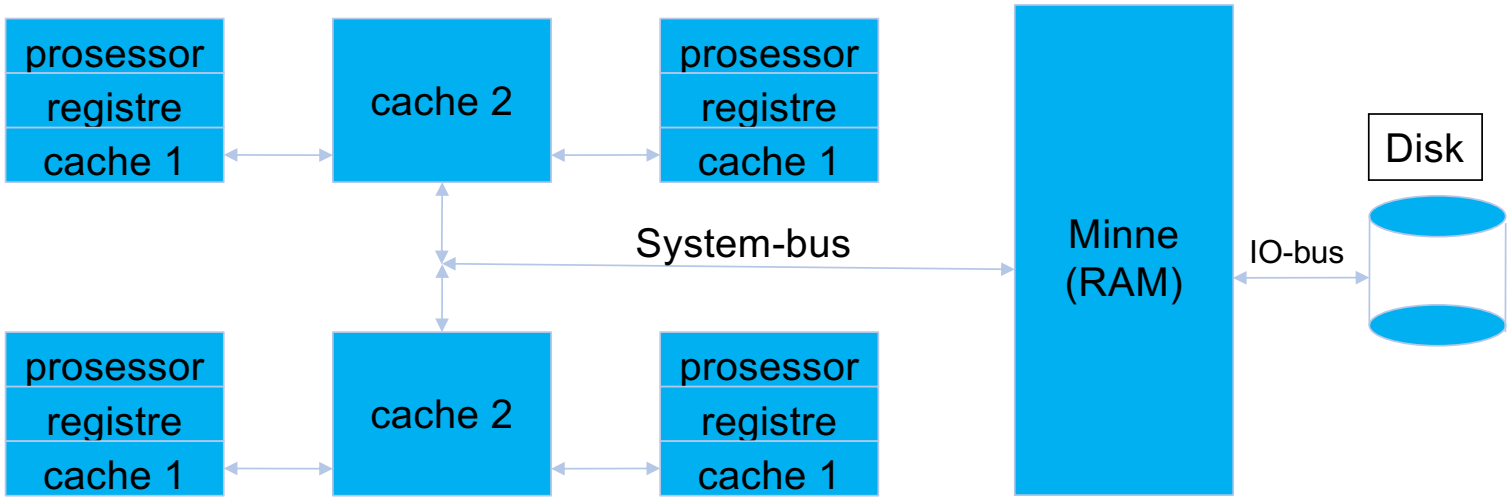
”Dual-core”, "8-core", ”N-core” Ekte parallellitet (nederst)

Datamaskinarkitektur (Computer Architecture)

3. Før:

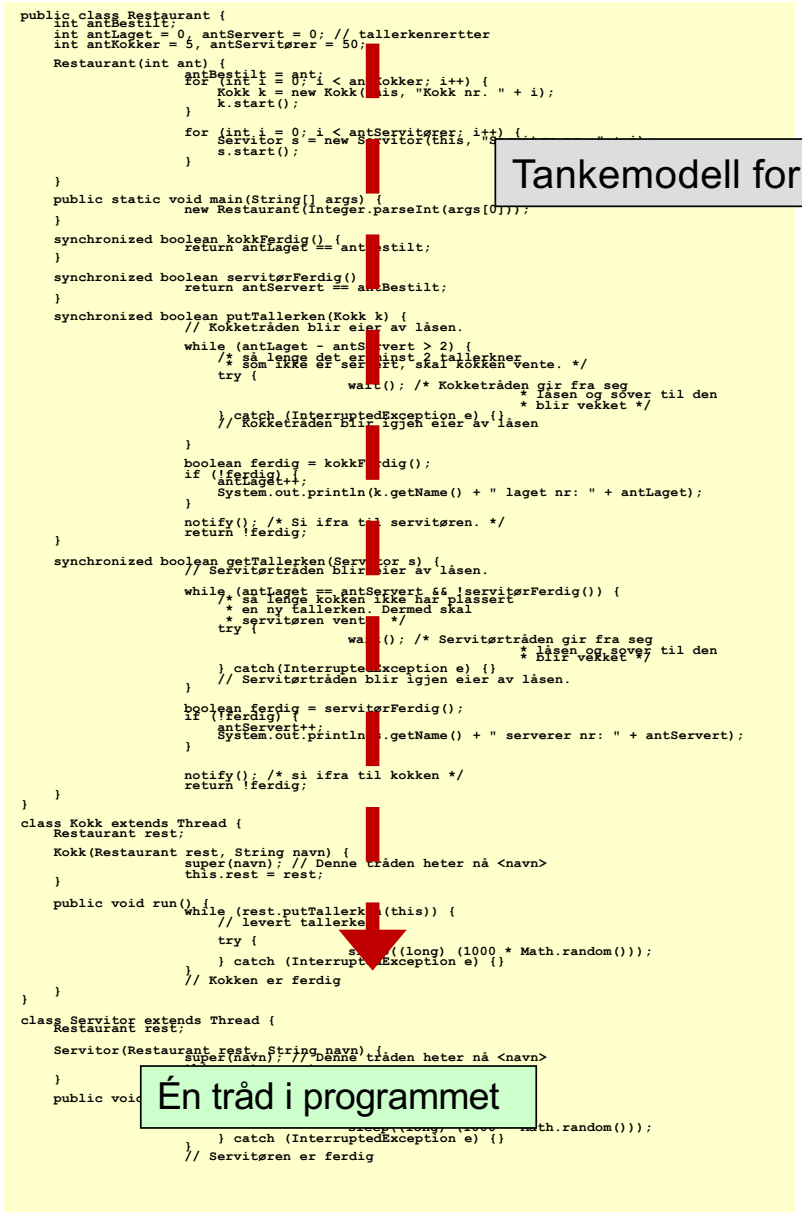


4. Nå, f.eks:



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)



Tankemodell for tråder

Én tråd i programmet



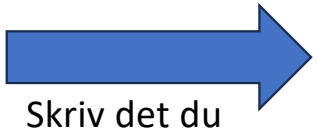
To tråder i programmet

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();
}
```

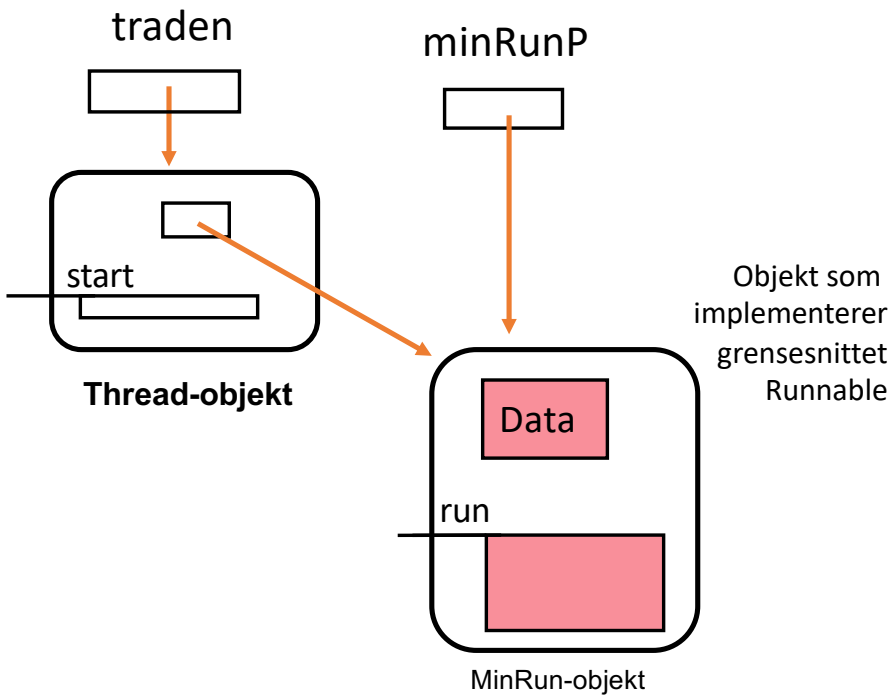


Skriv det du ønsker tråden skal gjøre

```
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>;
              . . .
       }
   }
}
```



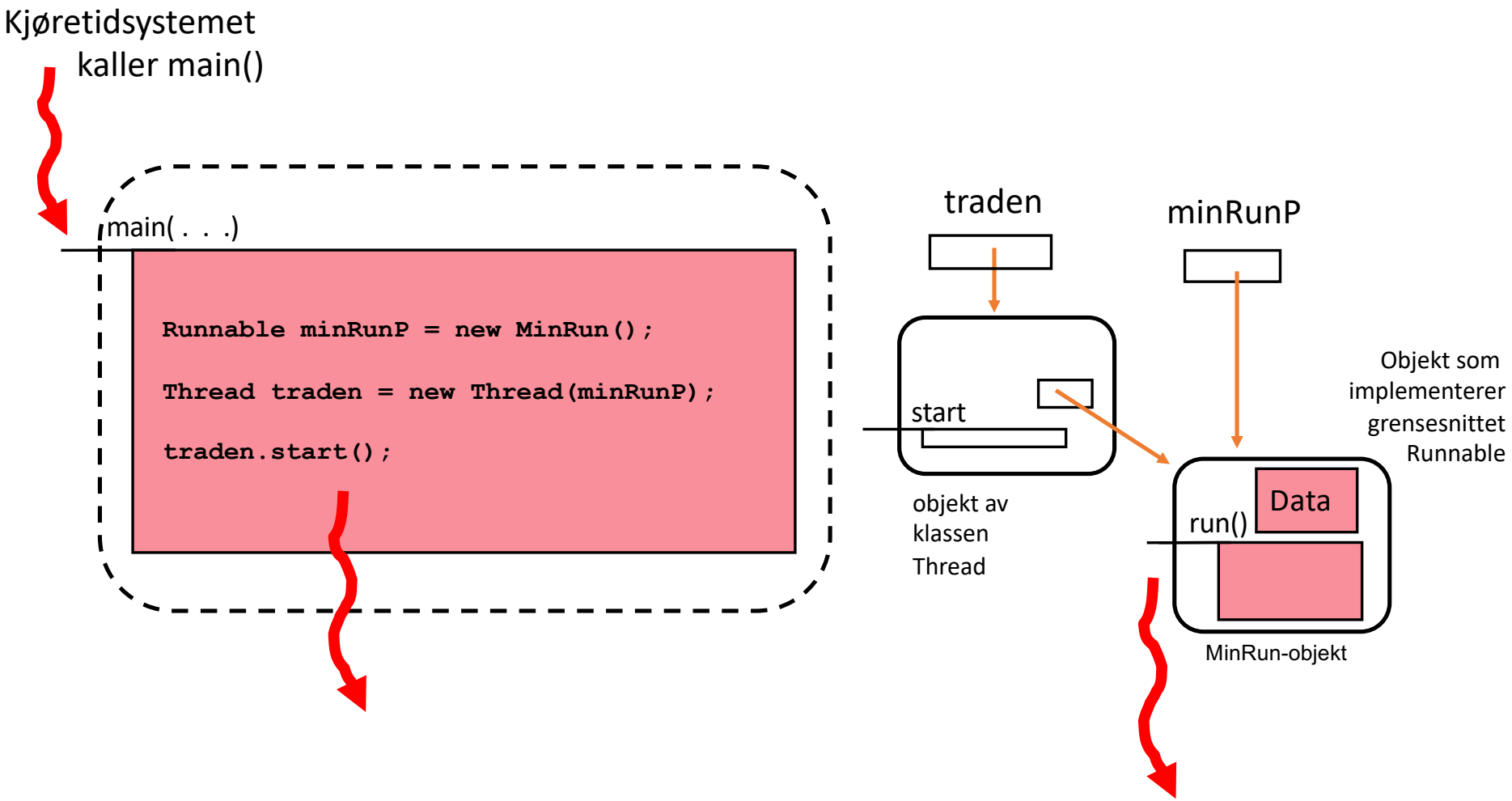
En tråd lages og startes opp slik (f.eks. i main()):

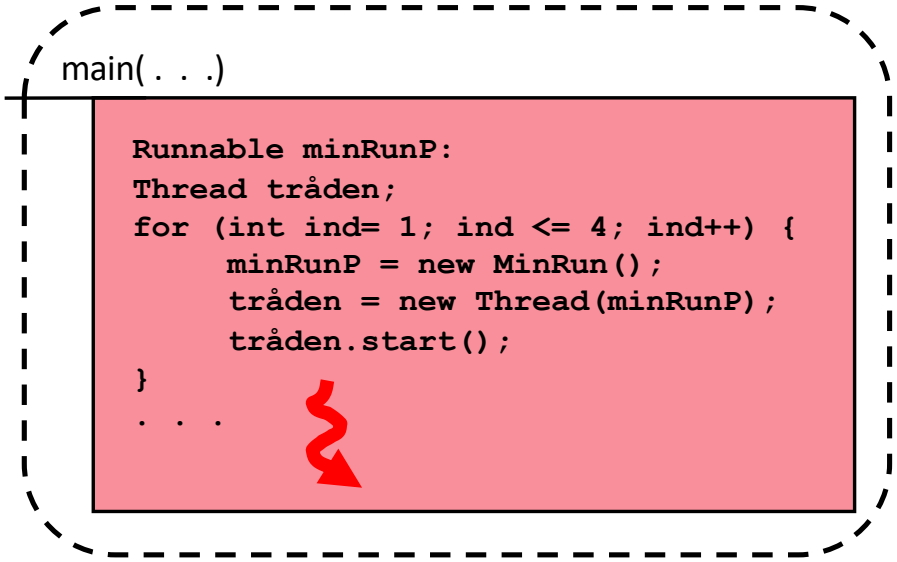
```
Runnable minRunP = new MinRun();
Thread traden = new Thread(minRunP);
traden.start( );
```



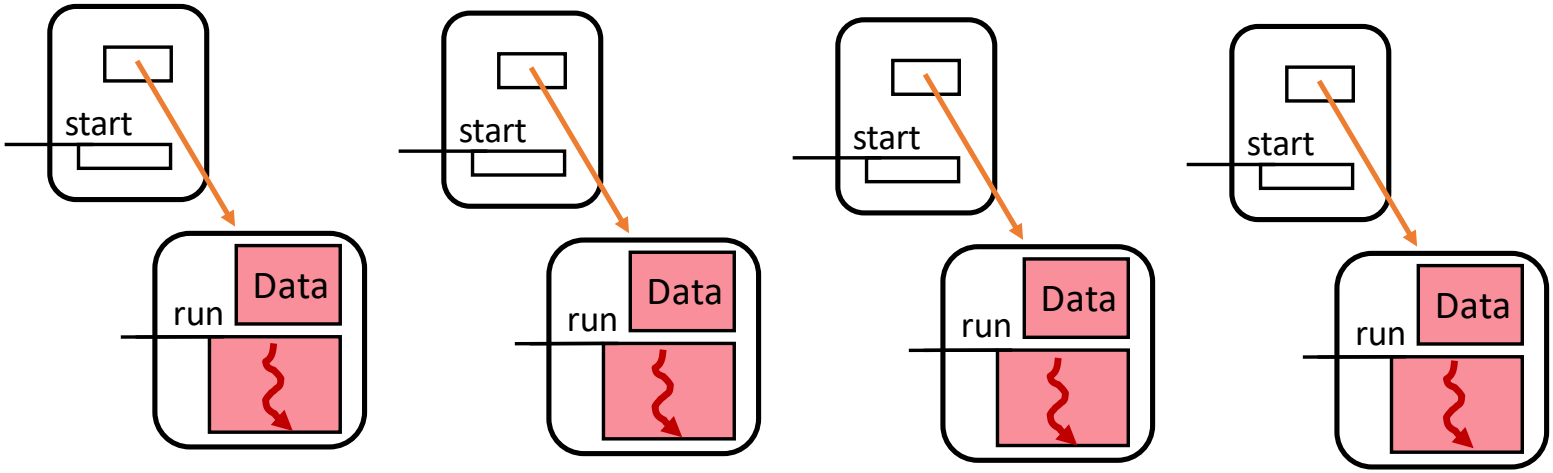
start( ) er en metode i Thread som må kalles opp for å få startet tråden. start-metoden vil igjen kalle metoden run (som vi selv programmerer).

Eksempel: main()-tråden starter én ny tråd





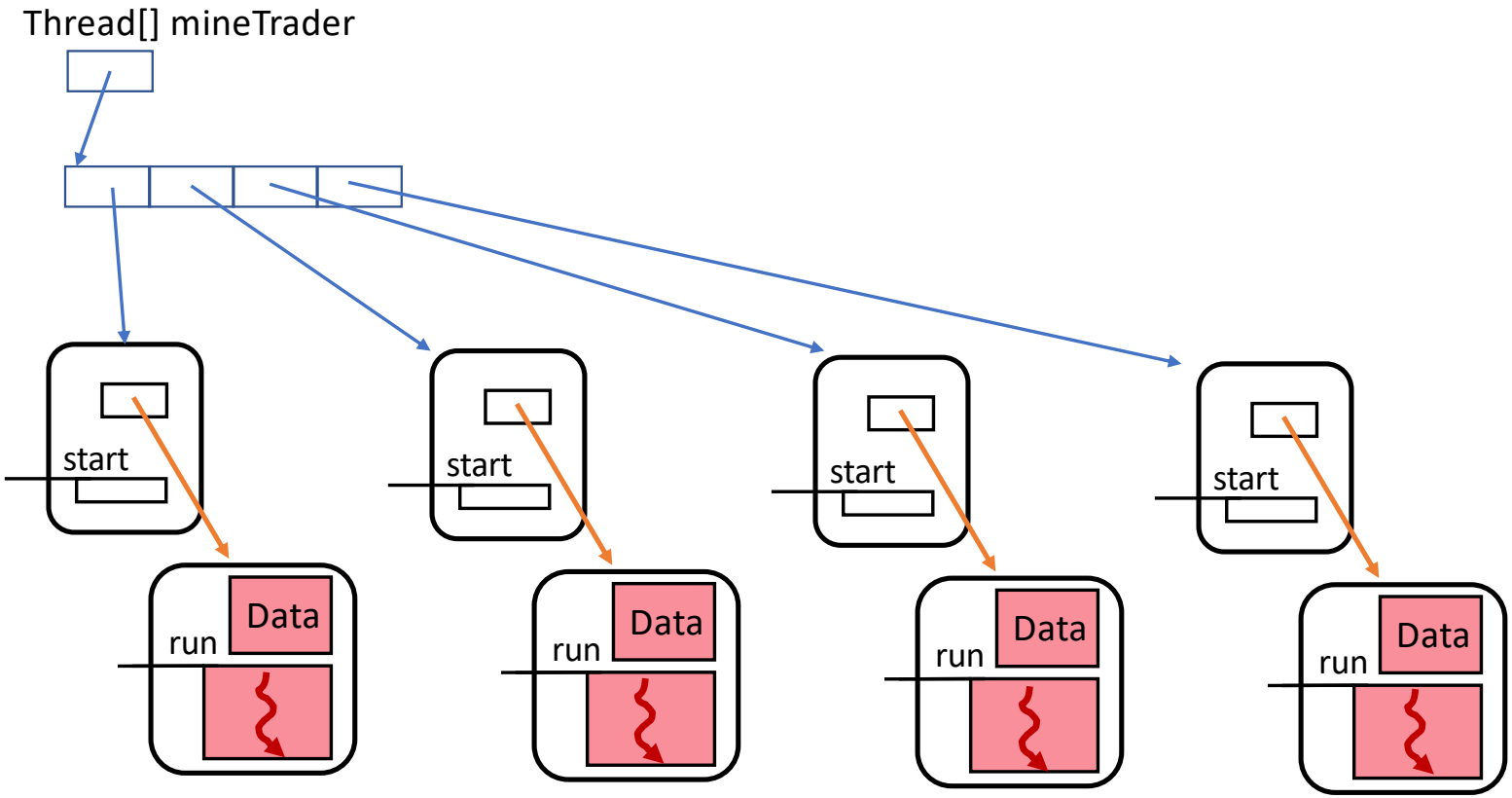
Eksempel:
main()-tråden starter 4 tråder



Trådeksempel

- Vi skal skrive et program som starter to tråder
- Hver tråd skriver en melding på skjermen i en løkke
- Programmet avslutter når vi trykker CTRL-C

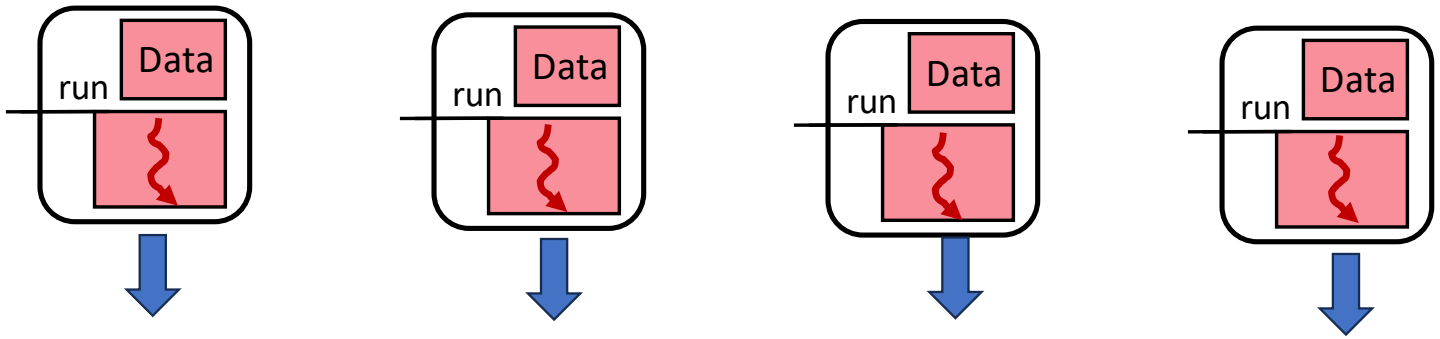
Noen ganger er det greit å ha referanser til alle trådene



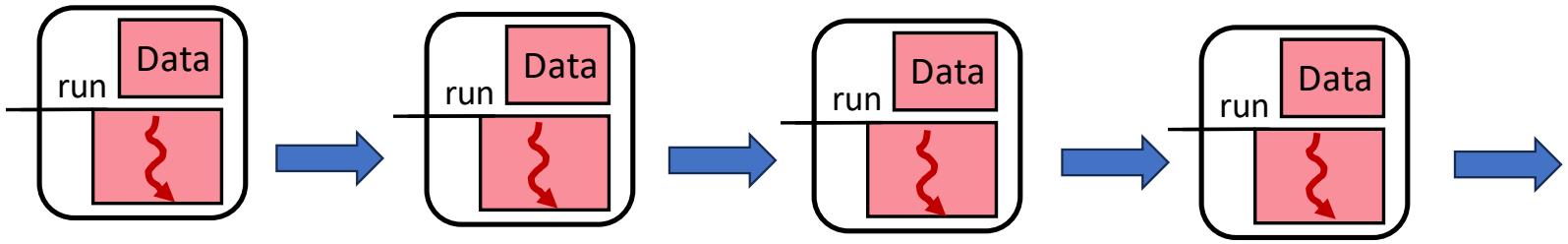
To hovedtyper parallellitet

- 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.”

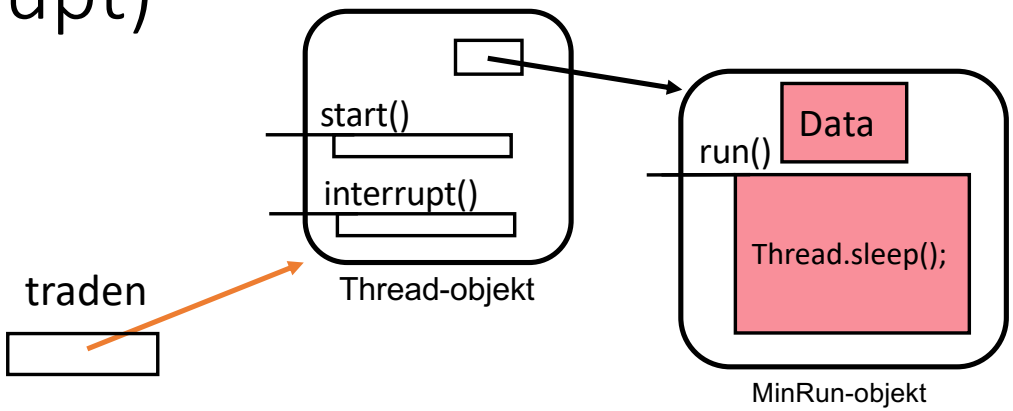
- Hvis noen avbryter denne tråden mens den sover skjer et unntak

```
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

- Vi skal nå skrive en stoppeklokke som teller sekunder
- Klokken skal starte på 0 sekunder og telle oppover
- Hvert sekund skrives verdien ut på skjermen
- Programmet avslutter når vi trykker return

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++;
        }
    }
}
```

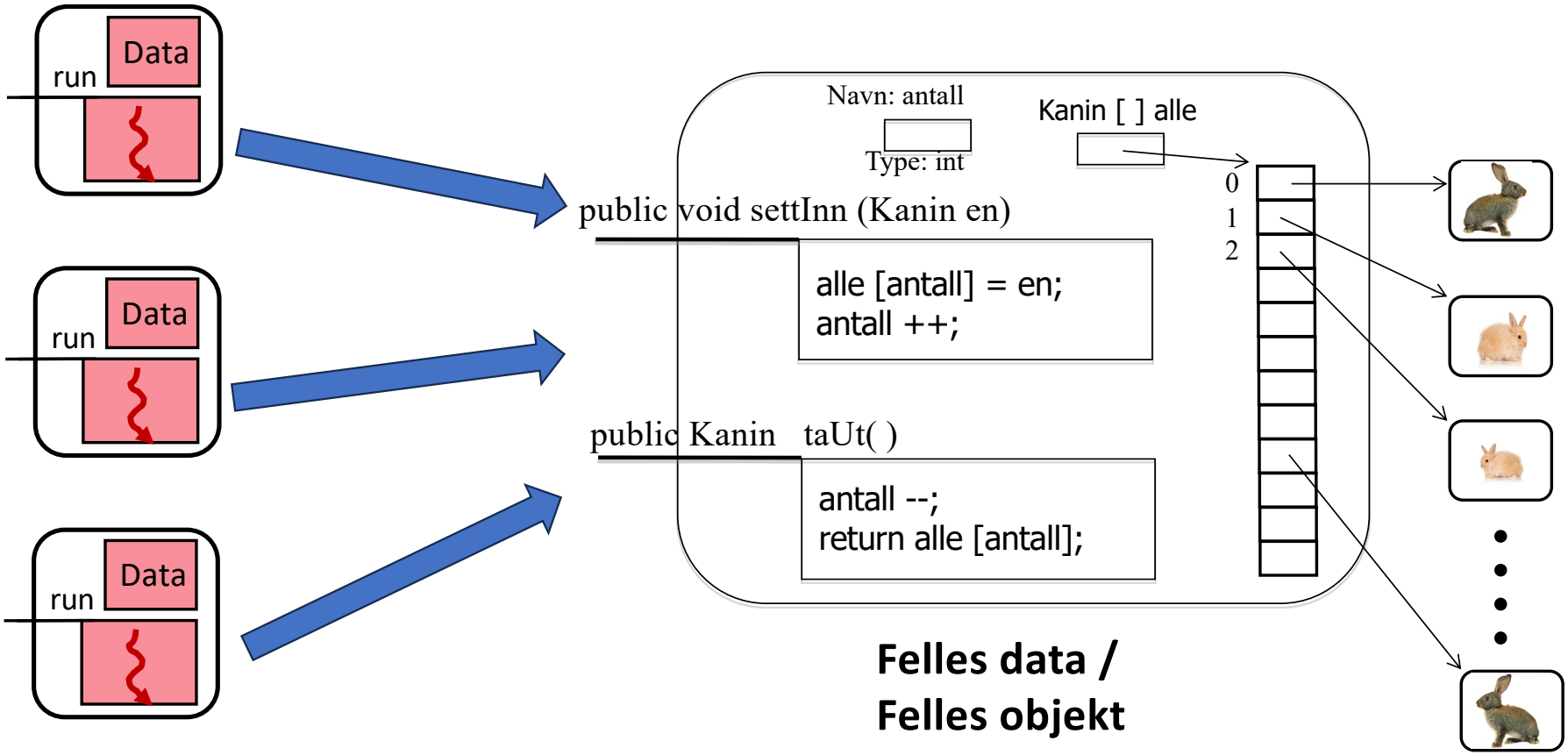
Kompiler og kjør denne koden selv. Gjør modifikasjoner.
Koden ligger på IN1010-siden
Her ligger også andre klokke-programmer

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?

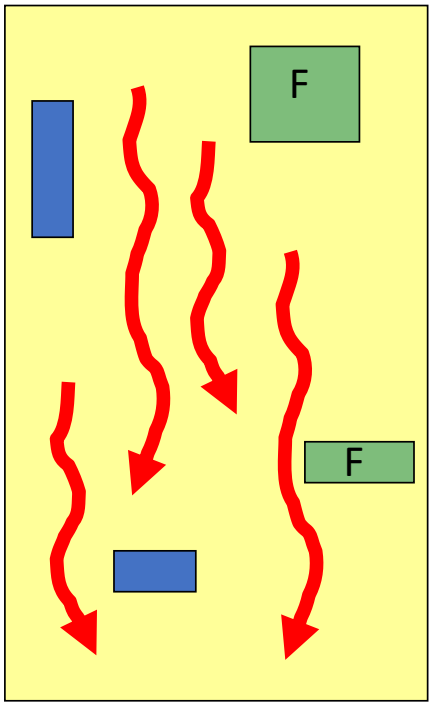


Felles data

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

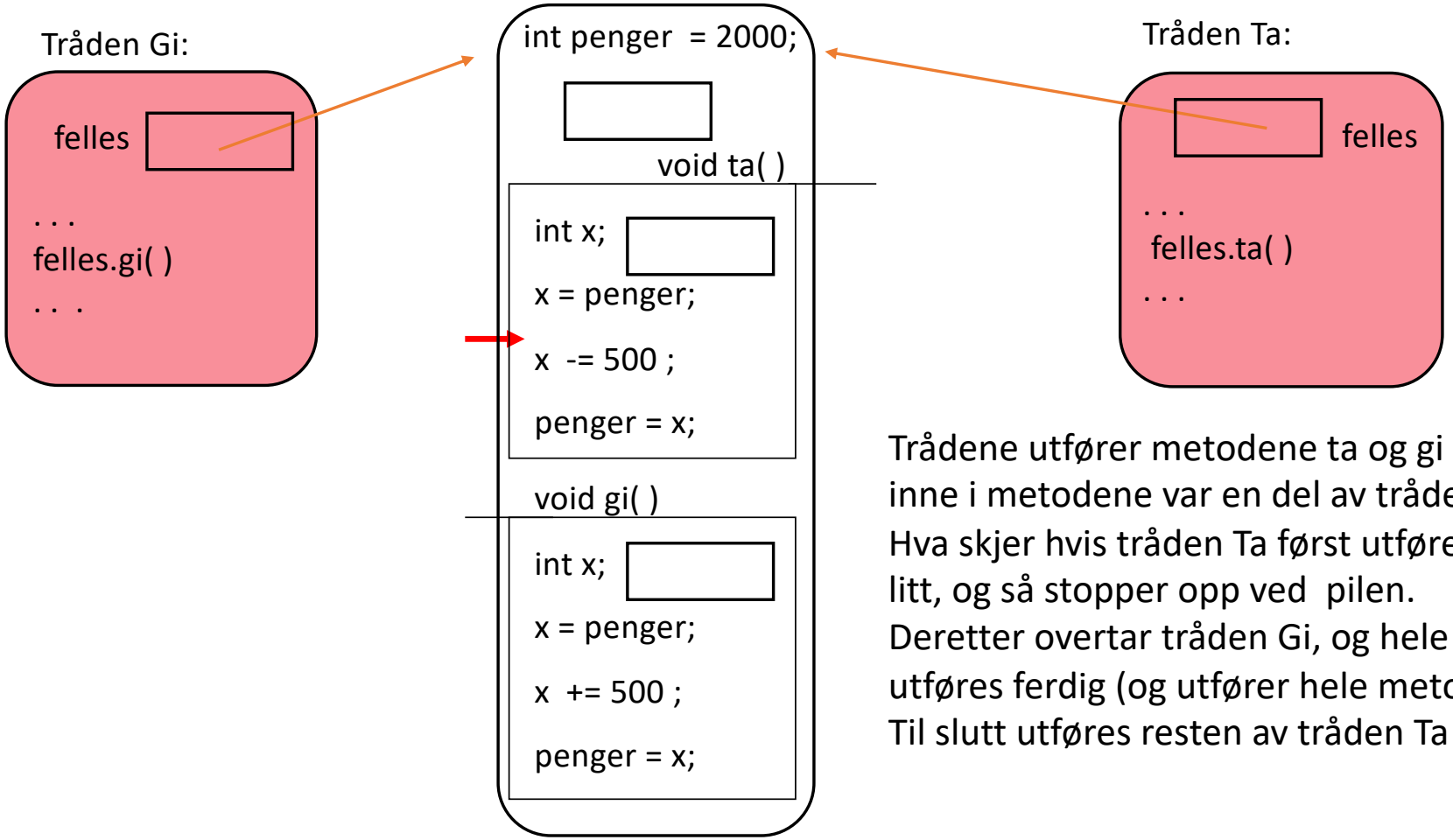
På figuren er det to områder vi har problemer med (dvs. at to eller flere tråder kan risikere å manipulere data i disse områdene samtidig). Disse to områdene er markert med F (for Felles). Kode som aksesserer felles data må skrives inne i en *kritisk region* (forklares snart).

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-resonnementer 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).

Bank med oppdateringsproblemer

```java
class GiRun implements Runnable {
    Konto konto;

    GiRun(Konto konto) {
        this.konto = konto;
    }

    public void run() {
        for (int i = 0; i < 1000; i++) {
            this.konto.gi();
        }
    }
}
```

```java
class Konto {
    int penger;

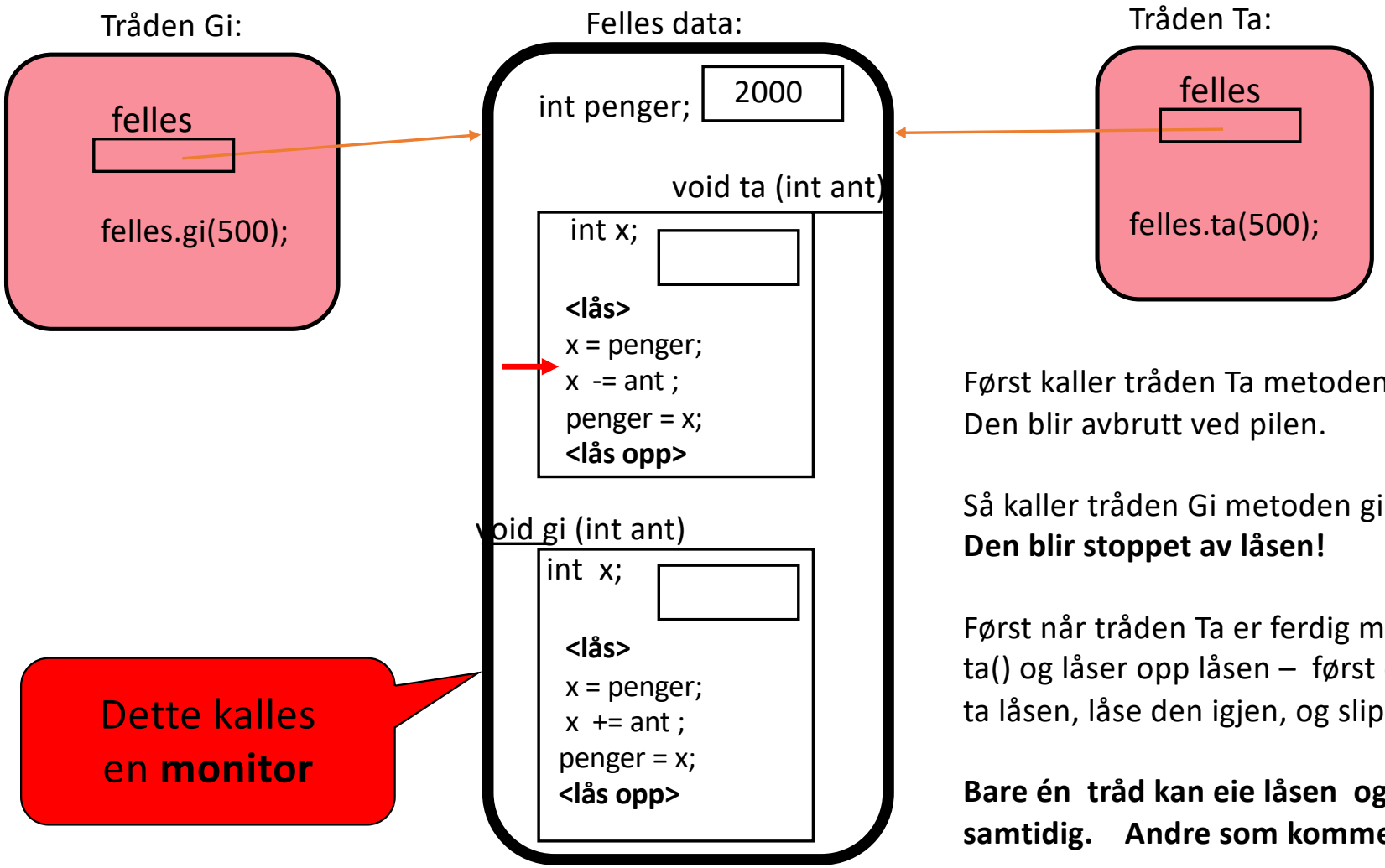
    Konto(int penger) {
        this.penger = penger;
    }

    public void ta() {
        int x;
        x = this.penger;
        x = x - 500;
        this.penger = x;
    }

    public void gi() {
        int x;
        x = this.penger;
        x = x + 500;
        this.penger = x;
    }
}
```

Koden ligger også på IN1010-siden.
Her ligger også flere bank-programmer som viser (og løser) problemet.

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().

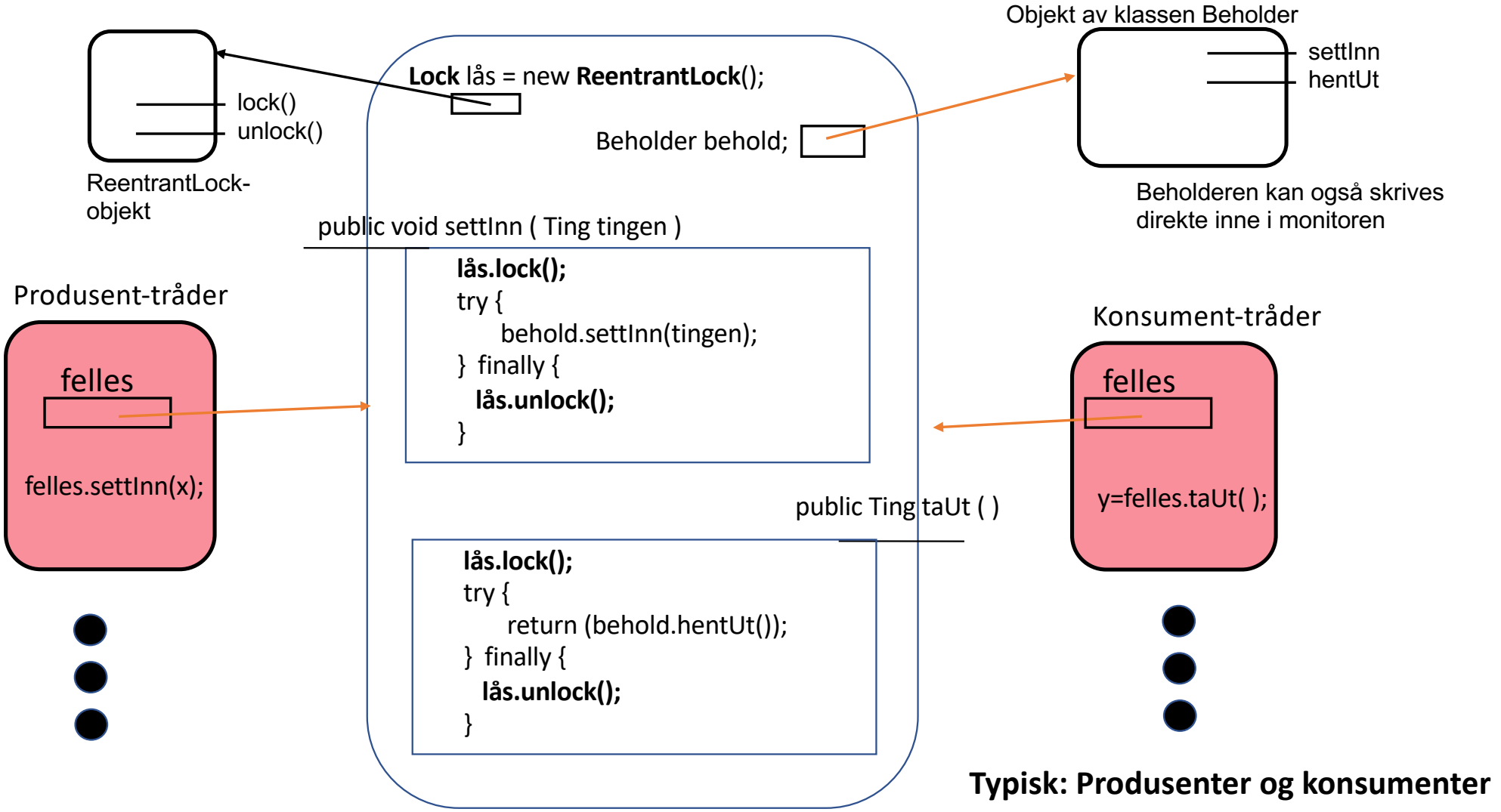
Bare én tråd kan eie låsen og de felles dataene samtidig. Andre som kommer må vente.

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

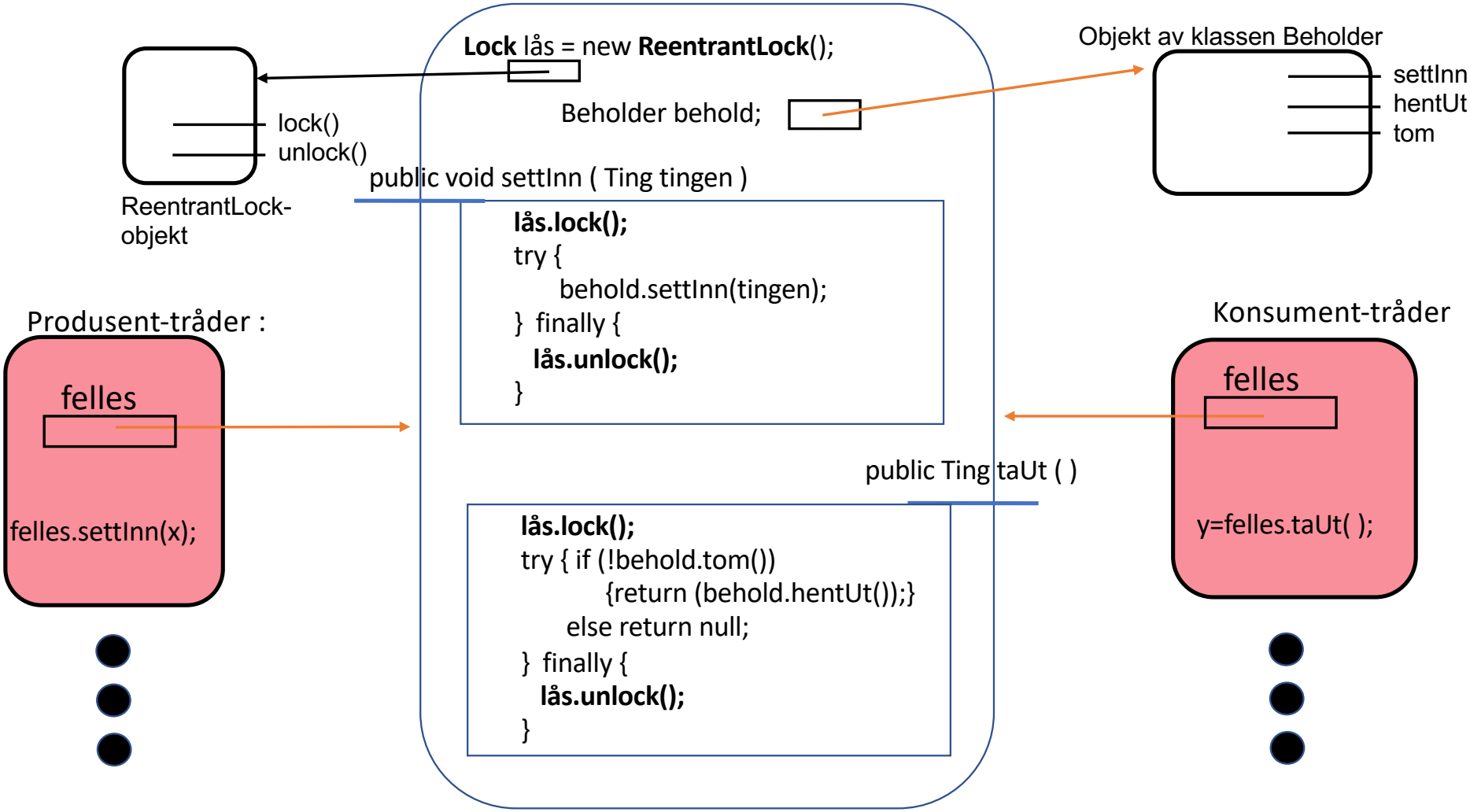
```
void  settInn (Ting tingen)  throws InterrupedException {
   lås.lock();
   try {
      :
      :
    }  finally {
     lås.unlock();
    }
}
```

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

Bank-eksempel

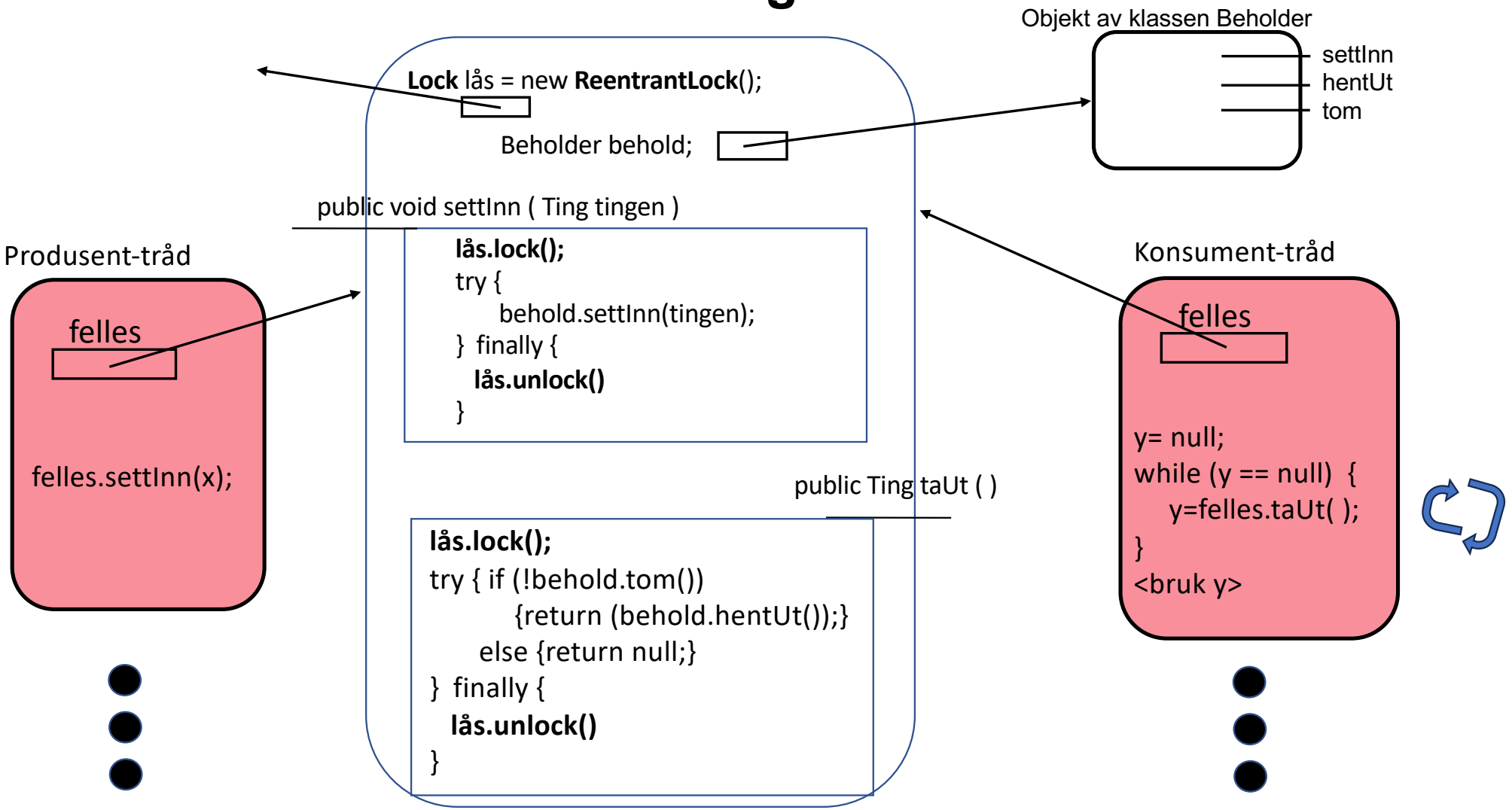
- Vi legger til låser og monitor

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

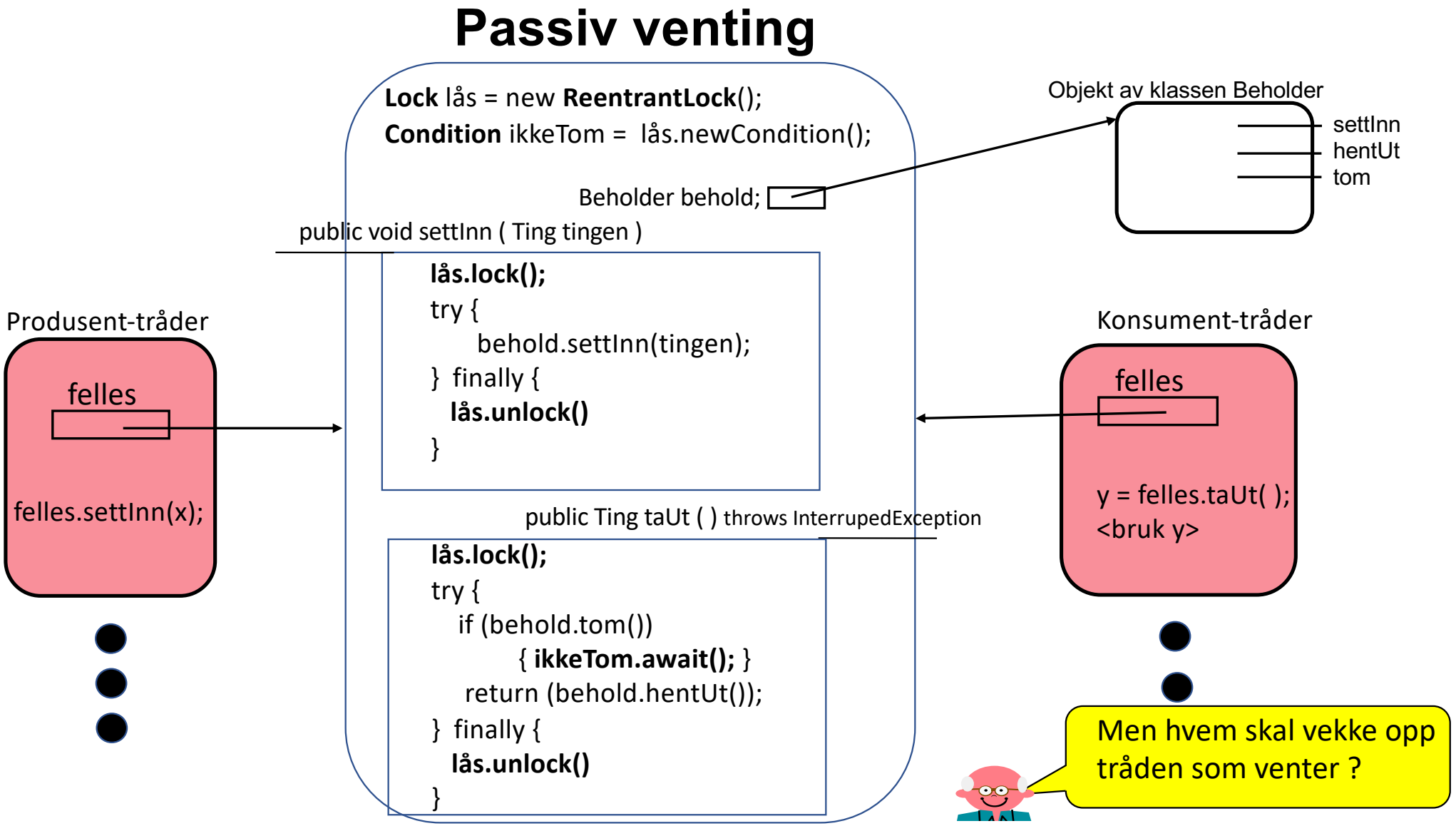
Aktiv venting



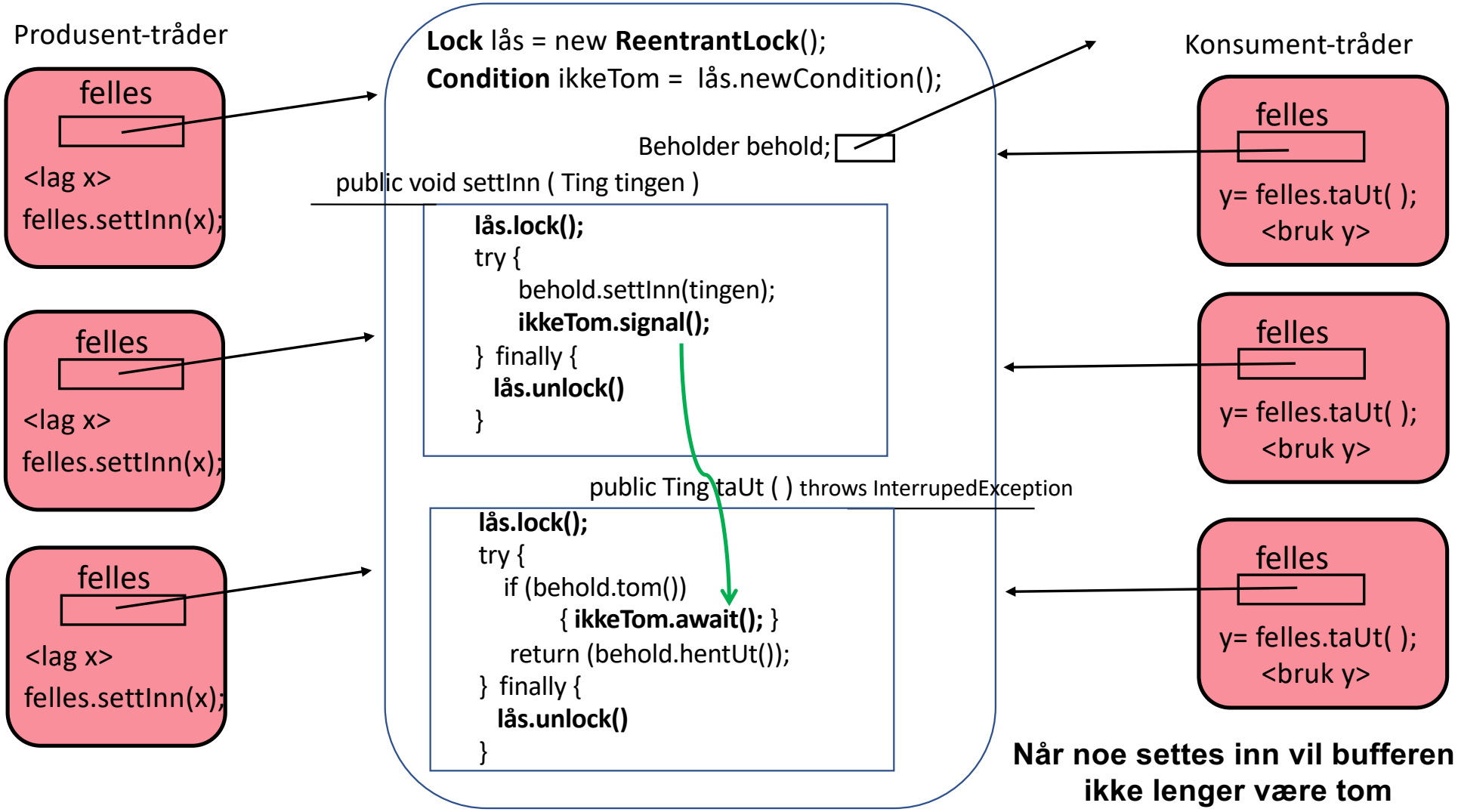
Passiv venting er bedre enn aktiv venting

- 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 konkurrerer og korte kritiske regioner
  - Ødelegger (blokkerer) ikke for andre og kan ikke bruke prosessorkapasiteten til noe annet

Passiv venting



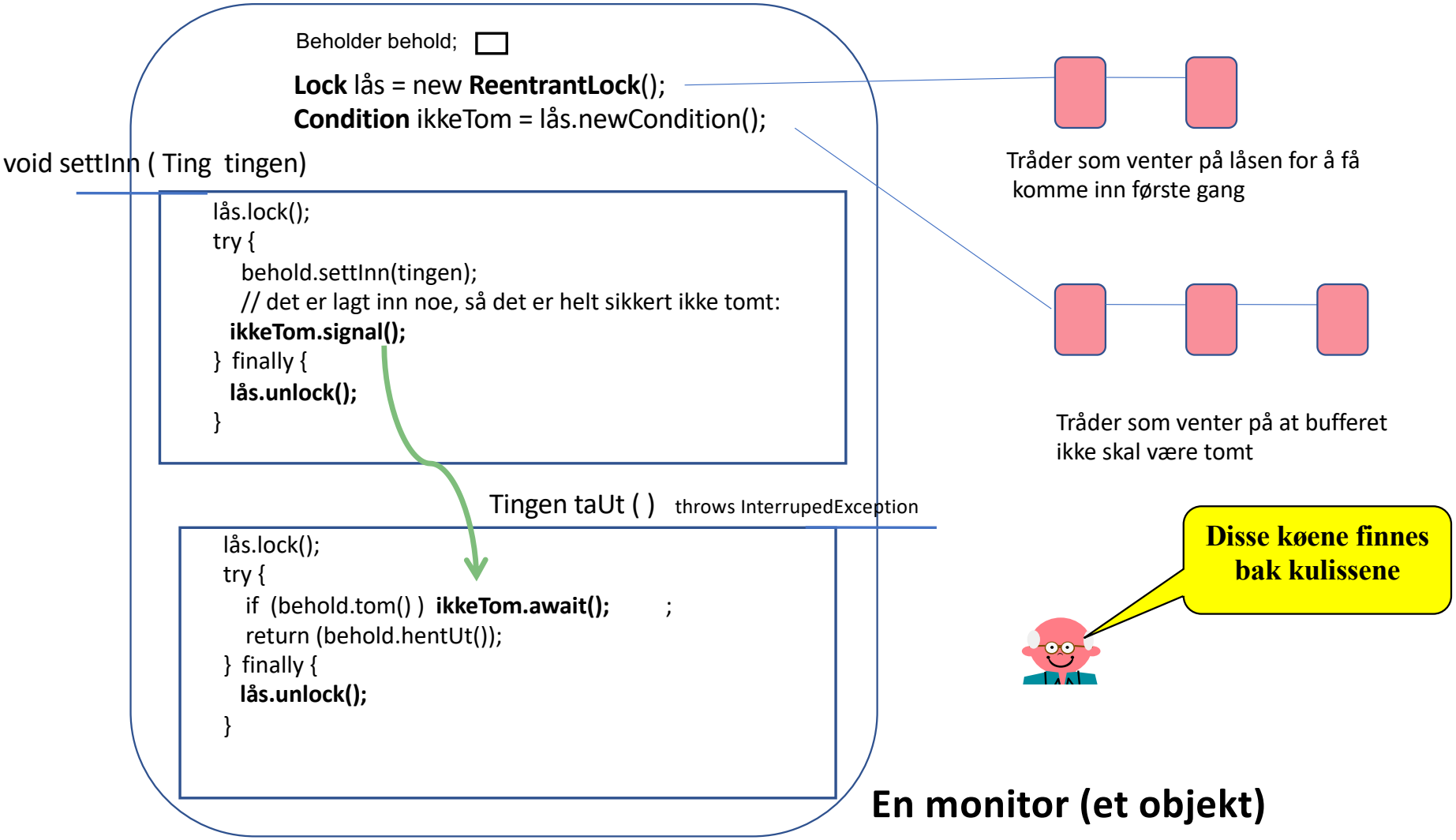
Passiv venting og signalering

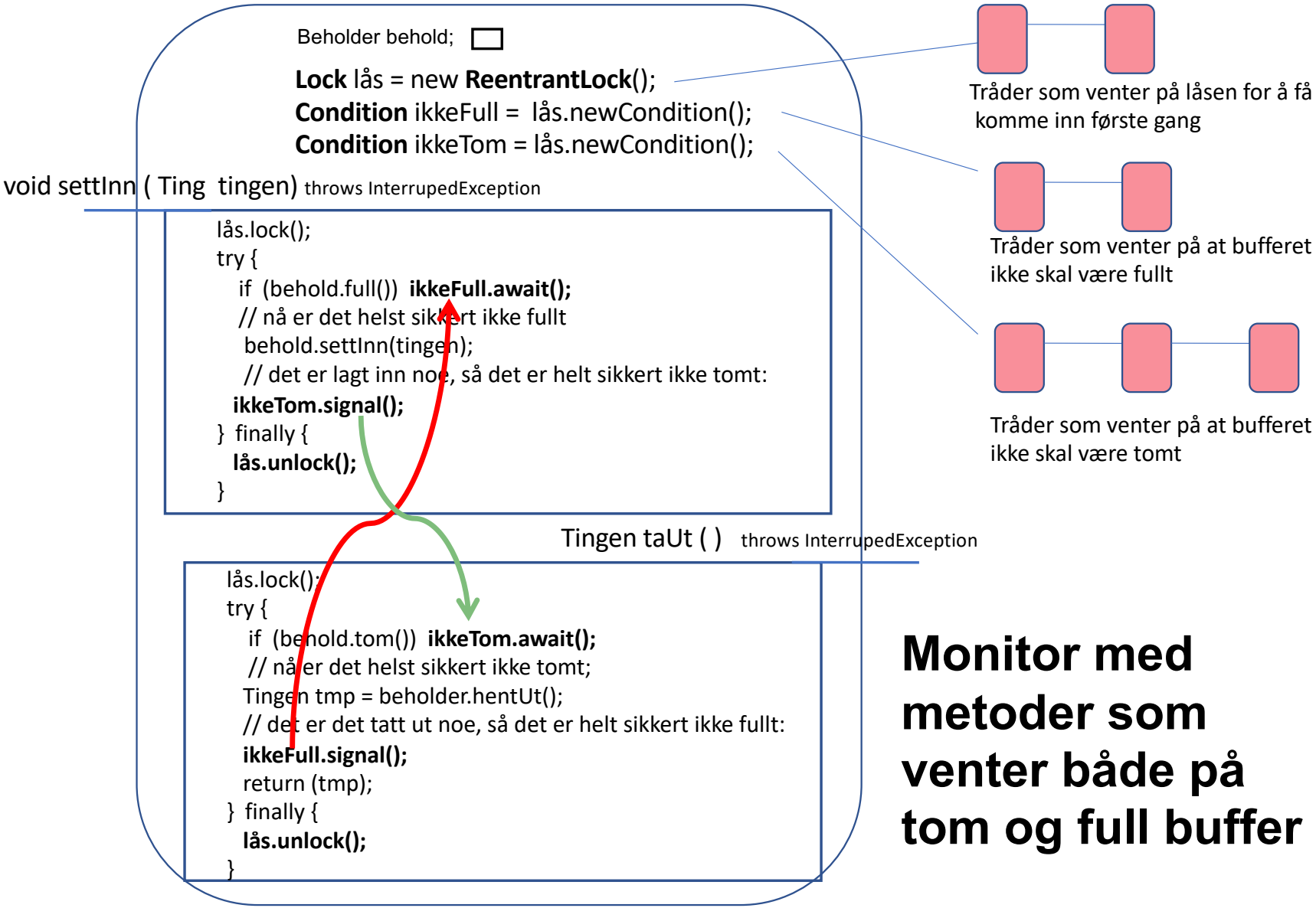


Bank-eksempel

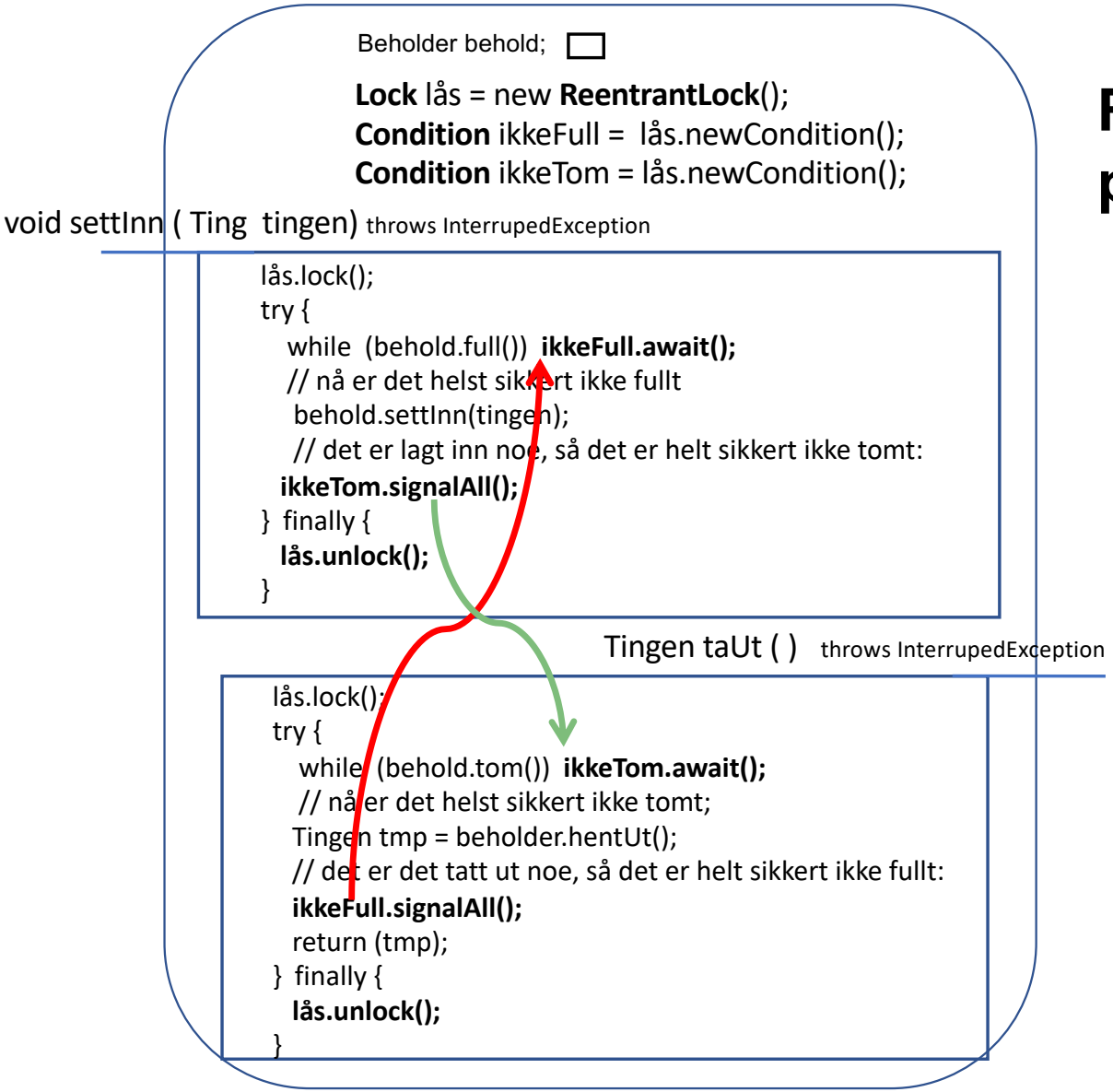
- Tidligere kunne vi kalle ta() også når det ikke var penger på kontoen
- Vi modifiserer nå ta() til å vente på at gi() skal kalles og sette inn penger først

Passiv venting, køer





Monitor med metoder som venter både på tom og full buffer



Regler for robust programmering

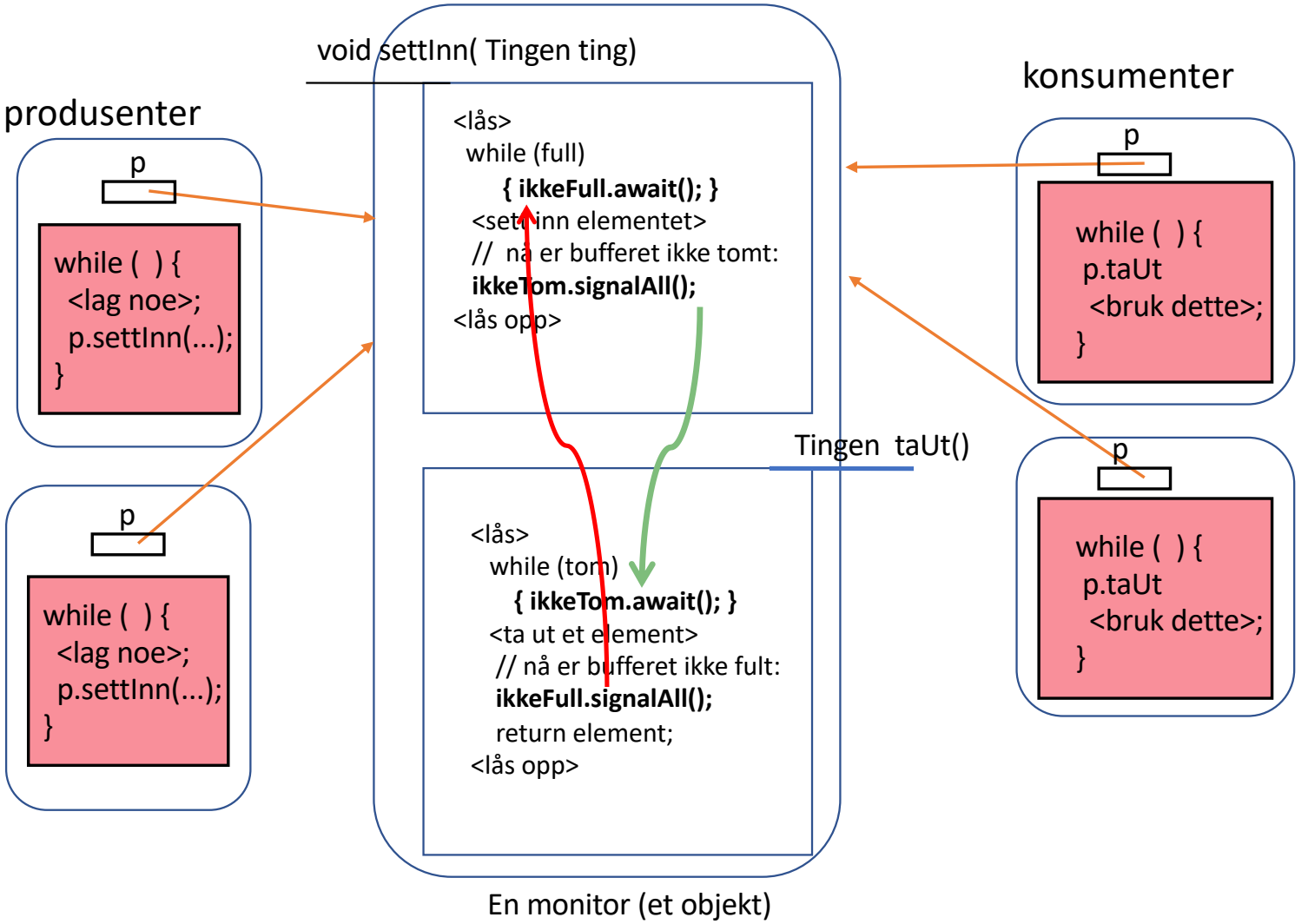
Her har vi skiftet ut if-testen i taUt() med en while-test.

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

Dette gjør programmet mer robust.

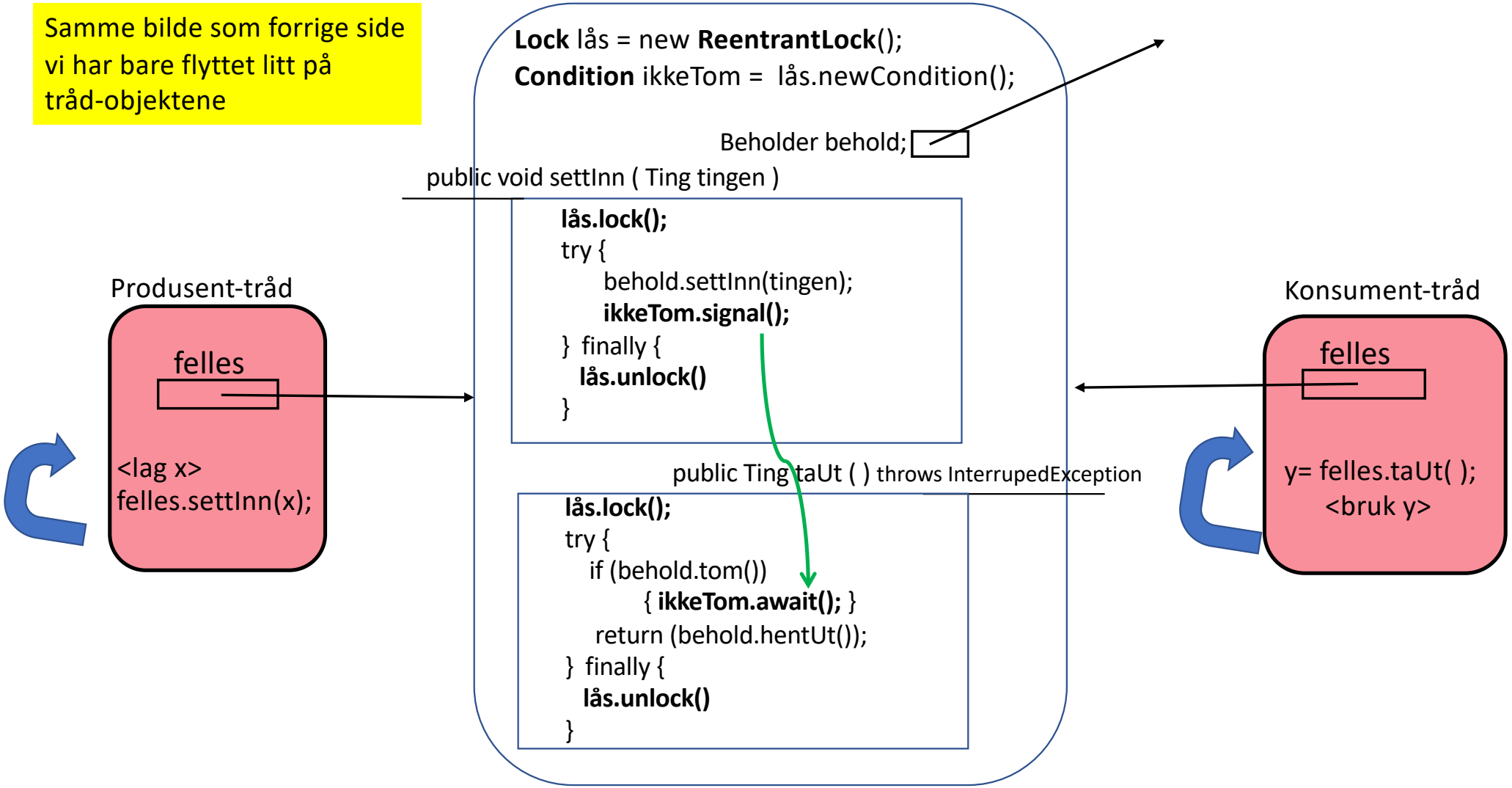
Se notatet om tråder.

Produsenter og konsumenter er et viktig eksempel



Produsent og konsument

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

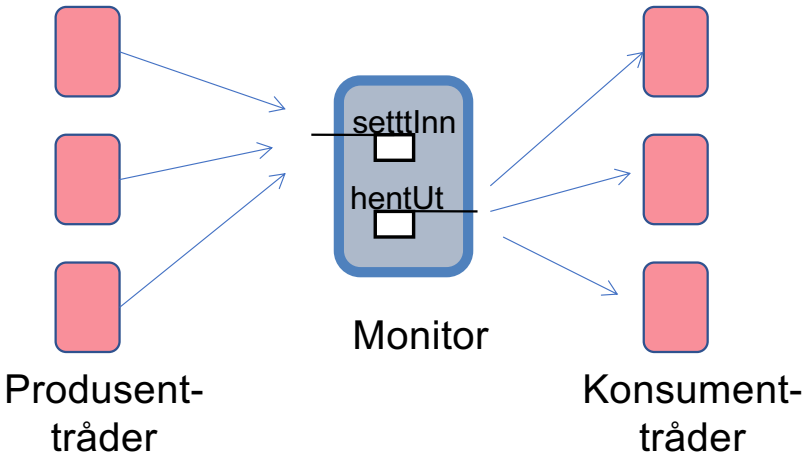


Hovedpoengene i dag

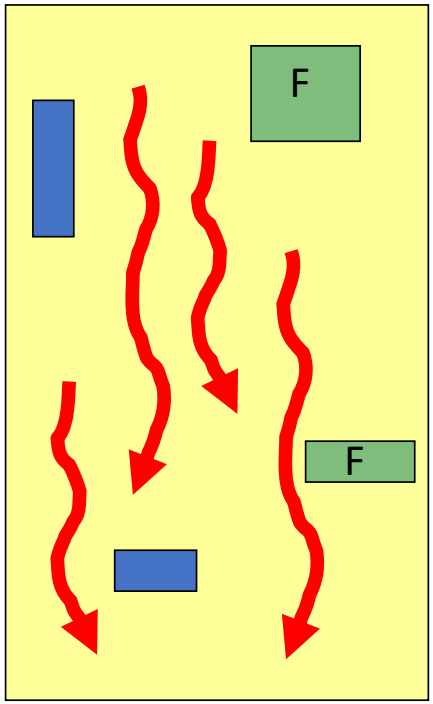
Du kan programmere parallelle aktiviteter i Java ved hjelp av tråder.

Tråder deler adresserom. Felles data (grønne 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.*







Innlevering 5

- Legges ut i dag – obligatorisk
  - Lurt å starte tidlig!
- Innlevering 20. april
- Mye av tråd-stoffet dekket i dag – vil snakke mer om tråder og innlevering 5 etter påske (7. april)

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.

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