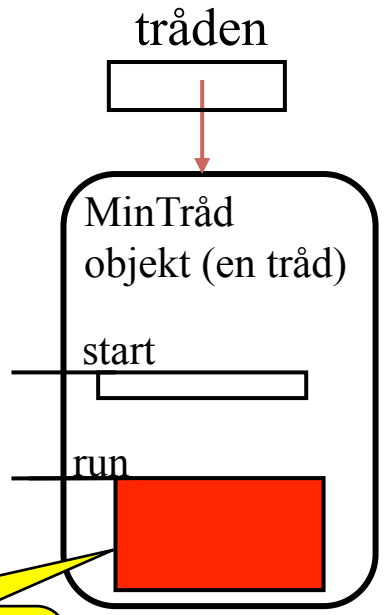
Tråder – del 2
Våren 2013

Stein Gjessing
Institutt for informatikk
Universitetet i Oslo

Repetisjon: # Tråder i Java:

```
class MinTråd extends Thread {
   public void run( ) {
      while (<mer å gjøre>) {
            <gjør noe>;
            try {sleep(<et tall, dvs. en stund>);}
            catch (InterruptedException e) { }
      }  // end while
   }  // end run
} //end class MinTråd
```



run inneholder vanligvis en løkke som utføres til oppgaven er ferdig.

En tråd lages og startes opp slik:

```
MinTråd tråden;
tråden = new MinTråd( );
tråden.start( );
```

Her går den nye og den gamle tråden (dette programmet), vider hver for seg

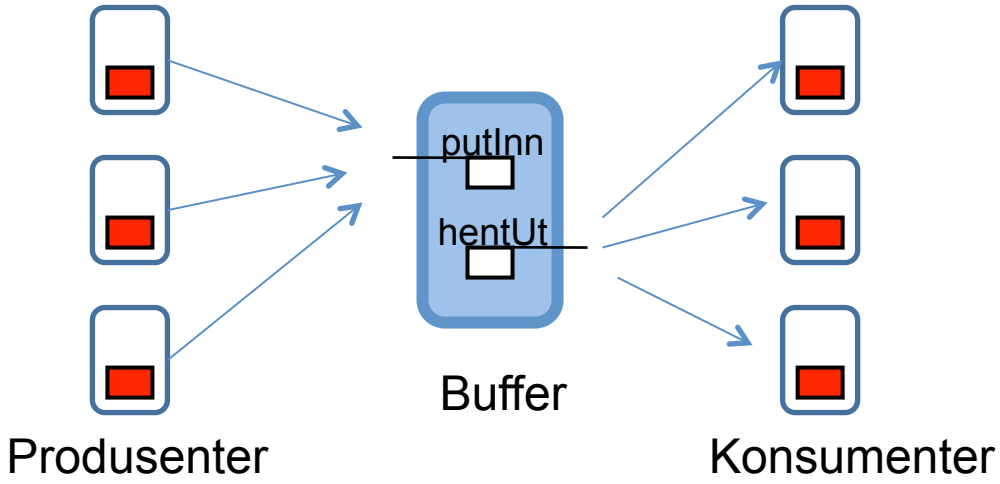
start( ) er en metode i Thread som må kalles opp for å få startet tråden.
start-metoden vil igjen kalle metoden run (som vi selv programmerer).

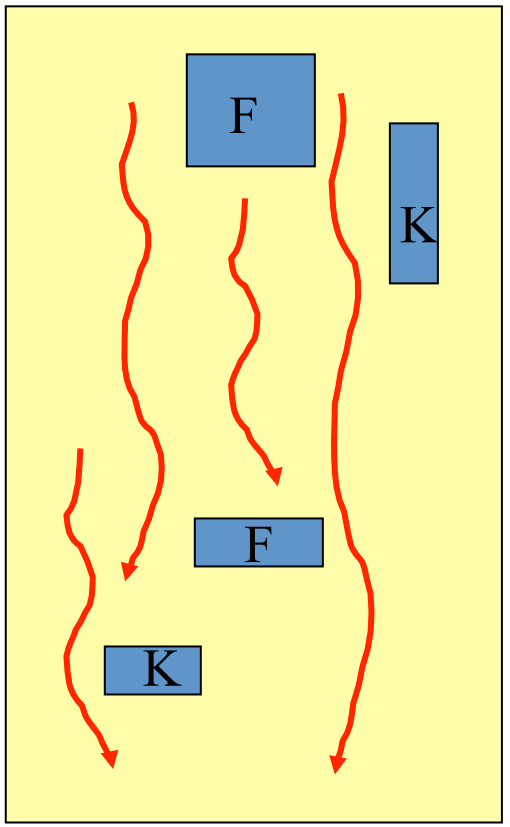
Kommunikasjon mellom tråder: Felles data

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 har modifikatoren **synchronized**

f.eks.

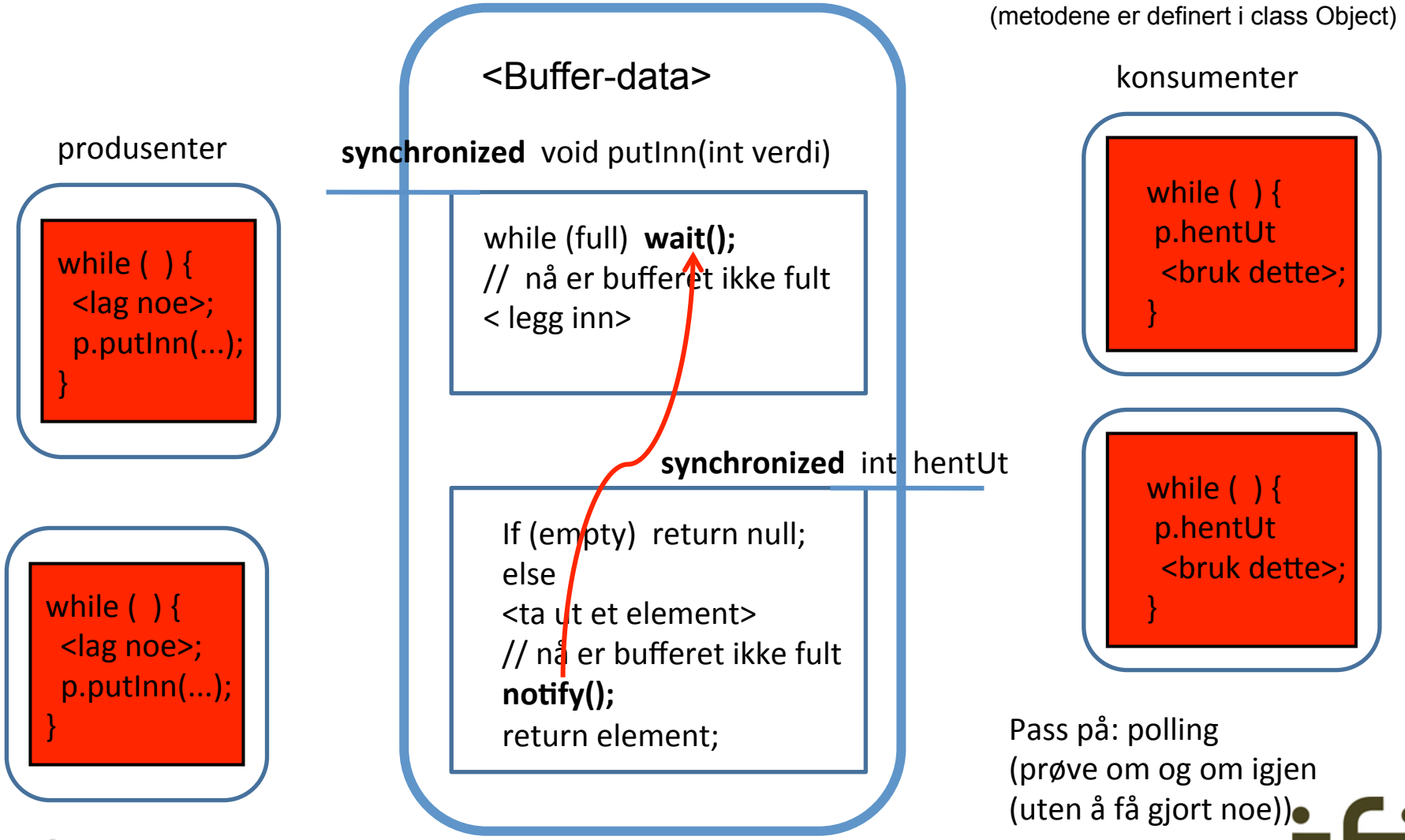




K: konstante data (immutable)

Basale Java-verktøy: synchronized, wait(), notify() og notifyAll()

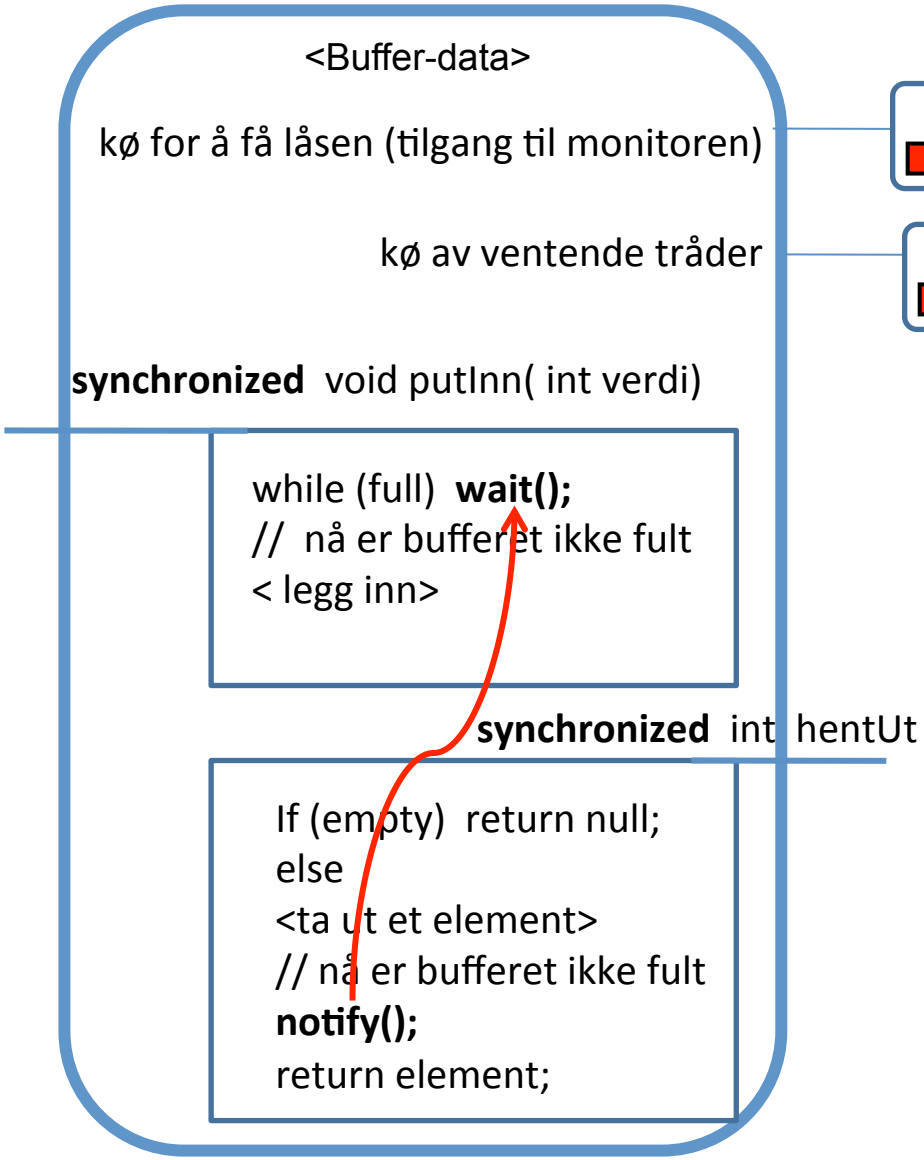
(metodene er definert i class Object)



En monitor (et objekt)

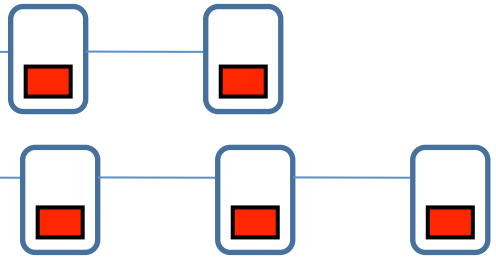
Pass på: polling
(prøve om og om igjen
(uten å få gjort noe))

To køer i en basal Java monitor:



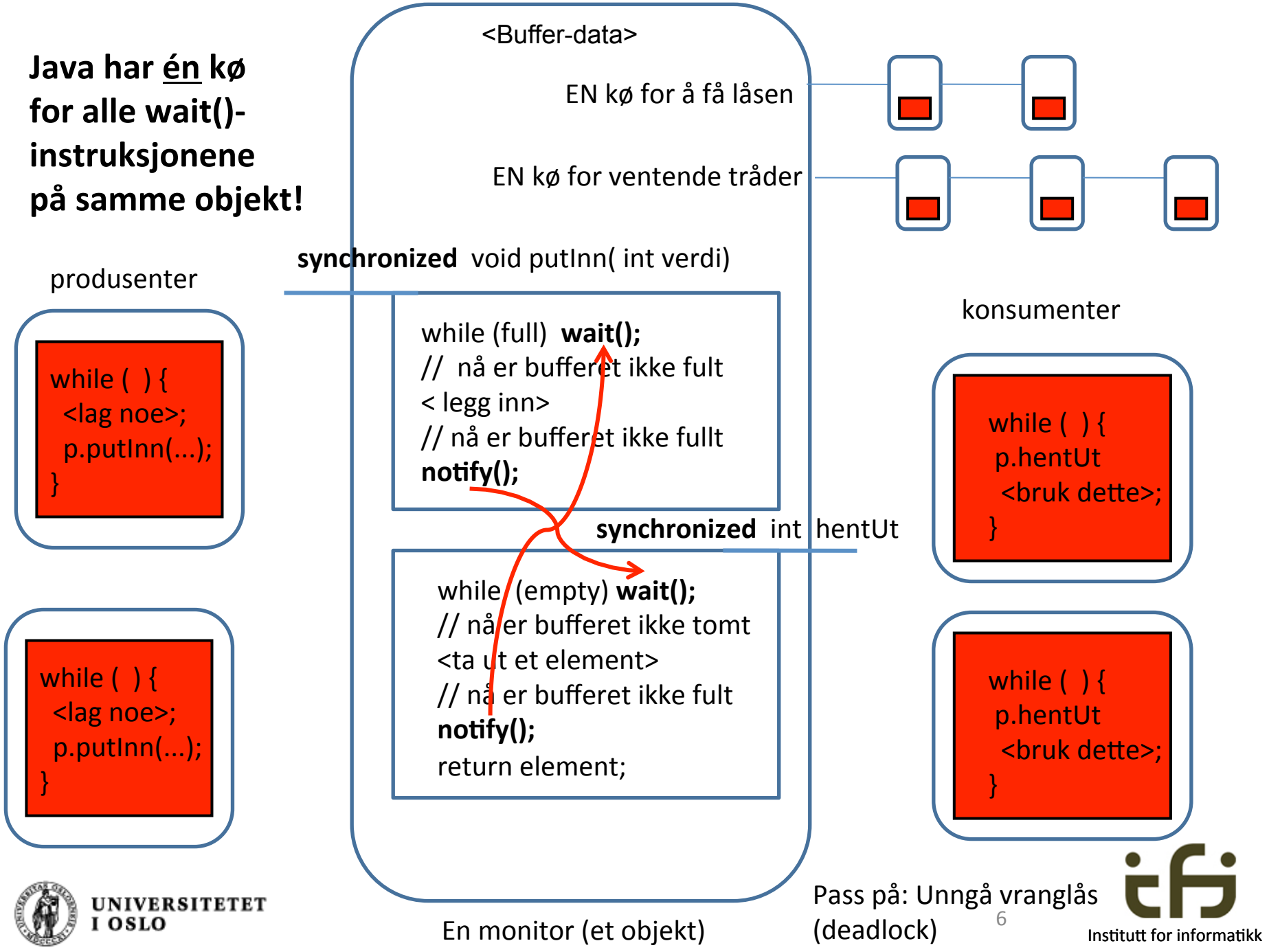
En monitor (et objekt)

En kø av ventende tråder
på hele monitoren



En kø av ventende tråder på
”wait”-instruksjoner
(wait-set).
Startes av notify () og/eller
notifyAll()
Legges da i den andre køen
(først ? (Nei, ingen garanti))
Derfor er det nødvendig med
”while …”

Java har <u>én</u> kø for alle wait()-instruksjonene på samme objekt!



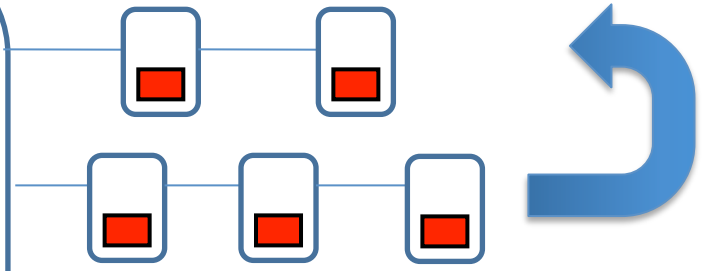
En monitor (et objekt)

Pass på: Unngå vranglås (deadlock)

<Buffer-data>

kø for å få låsen

ventende tråder



Flytt en: notify ();

Flytt alle: notifyAll ();

```
int ledig = totRes;
//invariant: 0 <= ledig <= totRes

synchronized  void reserver(int antall)
    while (ledig < antall)  wait();
    //  nå er det nok ledig
    ledig = ledig – antall;

synchronized  void  frigi(int antall)
    ledig = ledig + antall;
    // nå er mange flere ledig
    notifyAll();
```

En monitor (et objekt) for å reservere og frigi et antall like resurser

Hvis du ikke er HELT sikker på at enhver ventende tråd kan ordne skikkelig opp må du si:

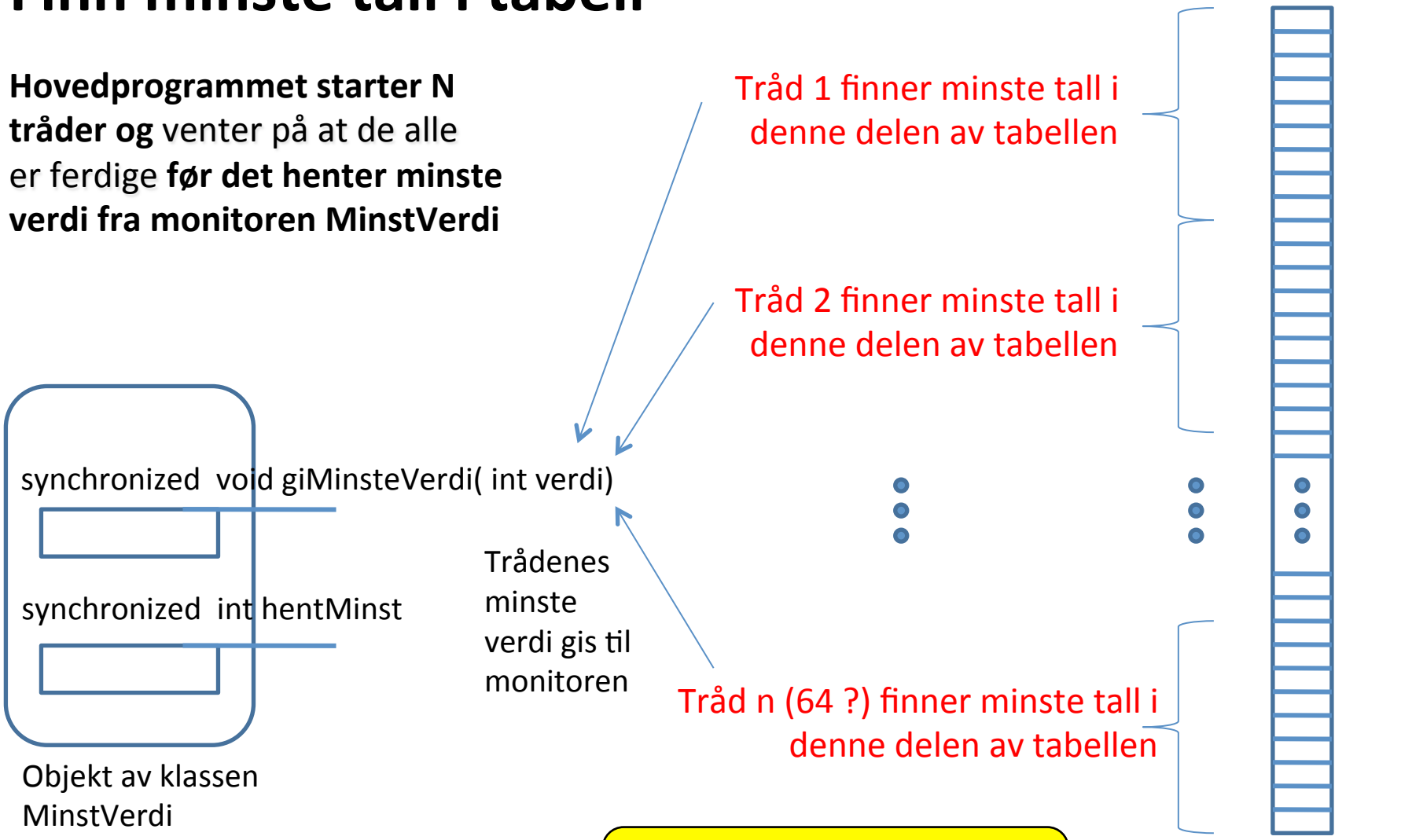
notifyAll();

Da blir ALLE ventende tråder flyttet opp til køen for å få låsen

Eksempel på parallellisering: Finn minste tall i tabell

(samme teknikk for å finne sum / gjennomsnitt)

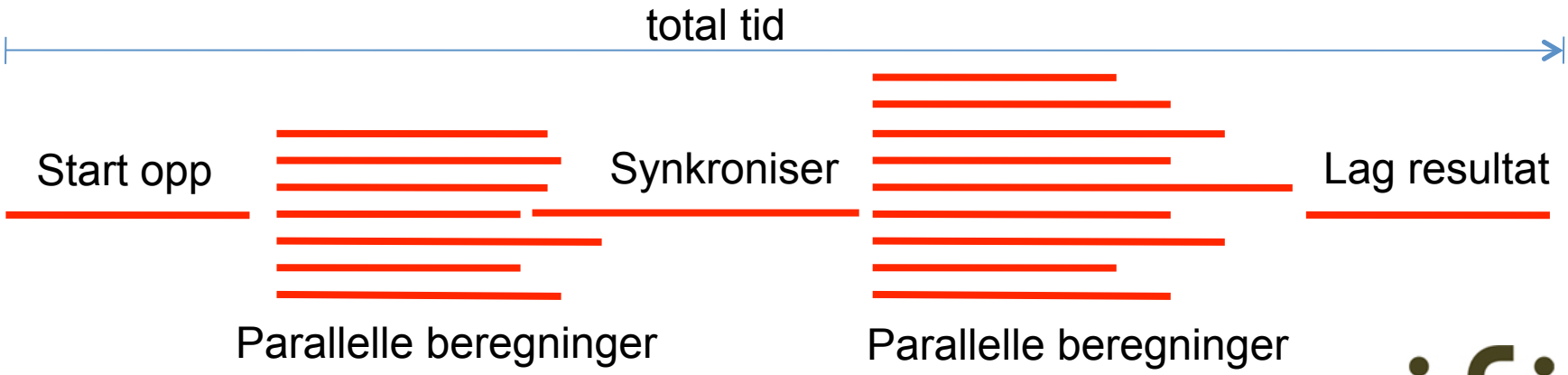
Hovedprogrammet starter N tråder og venter på at de alle er ferdige **før det henter minste verdi fra monitoren MinstVerdi**



BARE FANTASIEN SETTER GRENSER

Amdahls lov

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



Amdahls lov og "finn minste tall"

- Totaltid:
  - Tiden det tar å lage og sette i gang 64 tråder
    - (kan det gjøres i parallell ?) ($\log_2 64 = 6$) *
  - Tiden det tar å finne et minste tall i min del av tabellen
  - Til slutt i monitoren: En og en tråd må teste sitt resultat
    - (Kan det gjøres i parallell ?)

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

2, 4, 8, 16, 32, 64

Hovedprogrammet versjon 1

```
public class MinstR {
    final int maxVerdiInt = Integer.MAX_VALUE;
    int [ ] tabell;
    MinstMonitor monitor;

    public static void main(String[ ] args) {
        new MinstR();
    }

    public MinstR ( ) {
         tabell = new int[640000];
         for (int in = 0; in< 640000; in++)
                  tabell[in] = (int)Math.round(Math.random()* maxVerdiInt);
         monitor = new MinstMonitorR();
         for (int i = 0; i< 64; i++)
             // Lag og start 64 tråder
             new MinstTradR(tabell,i*10000,((i+1)*10000)-1,monitor).start();
         monitor.vent();
         System.out.println("Minste verdi var: " + monitor.hentMinste());
    }
}
```

Monitor for resultater og synkronisering

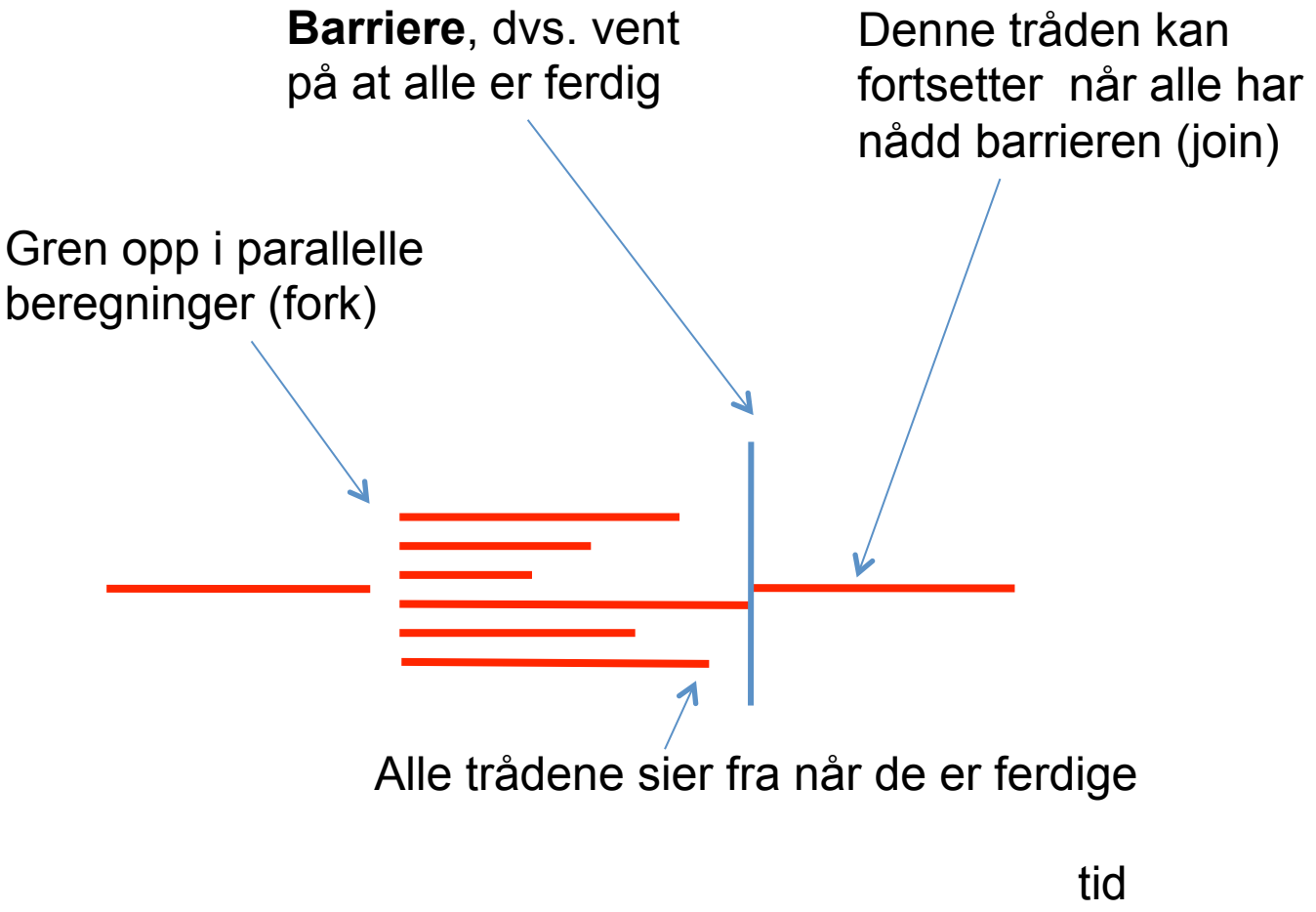
```
interface MinstMonitor {
  void vent();
  void giMinsteVerdi (int minVerdi);
  int hentMinste ();
}
```

```
class MinstMonitorR  implements MinstMonitor {
   private int minstTilNa = Integer.MAX_VALUE;
   private int antallFerdigeSubtrader = 0;
   synchronized void vent() {
       while (antallFerdigeSubtrader != 64) {
              try {wait();
                  System.out.println(antallFerdigeSubtrader + "
                                                 ferdige subtråder ");
              }
              catch (InterruptedException e) {
                     System.out.println(" Uventet avbrudd ");
                     System.exit(1);
              }
       // antall ferdige subtråder er nå 64
       }
   }
   synchronized void giMinsteVerdi (int minVerdi) {
      antallFerdigeSubtrader ++;
      if(minstTilNa > minVerdi) minstTilNa = minVerdi;
      if(antallFerdigeSubtrader == 64) notify();
      // eller hver gang, (men stort sett unødvendig): notify();
   }
   synchronized int hentMinste () {
       return minstTilNa;
   }
}
```

Tråder som finner minste tall i del av tabellen

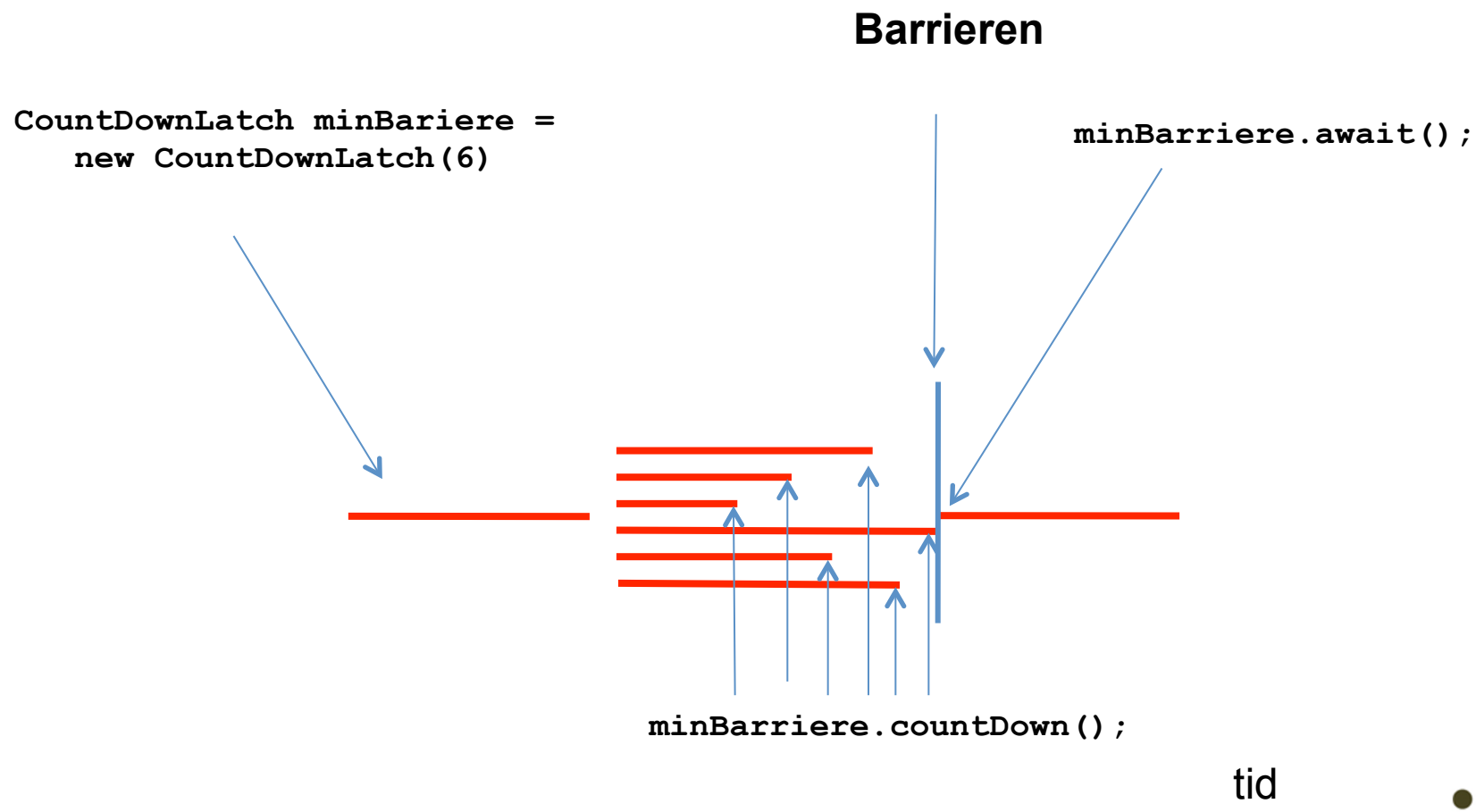
```
class MinstTradR extends Thread {

    int [ ] tab; int startInd, endInd;
    MinstMonitor mon;

    MinstTradR(int [ ] tb, int st, int en, MinstMonitor m) {
      tab = tb; startInd = st; endInd = en;
      mon = m;
    }

    public void run(){
      int minVerdi = Integer.MAX_VALUE;
      for ( int ind = startInd; ind <= endInd; ind++)
                  if(tab[ind] < minVerdi) minVerdi = tab[ind];
       // denne tråden er ferdig med jobben:
       mon.giMinsteVerdi(minVerdi);
     } // slutt run

} // slutt MinstTradR
```

Barrierer i parallellprogrammering

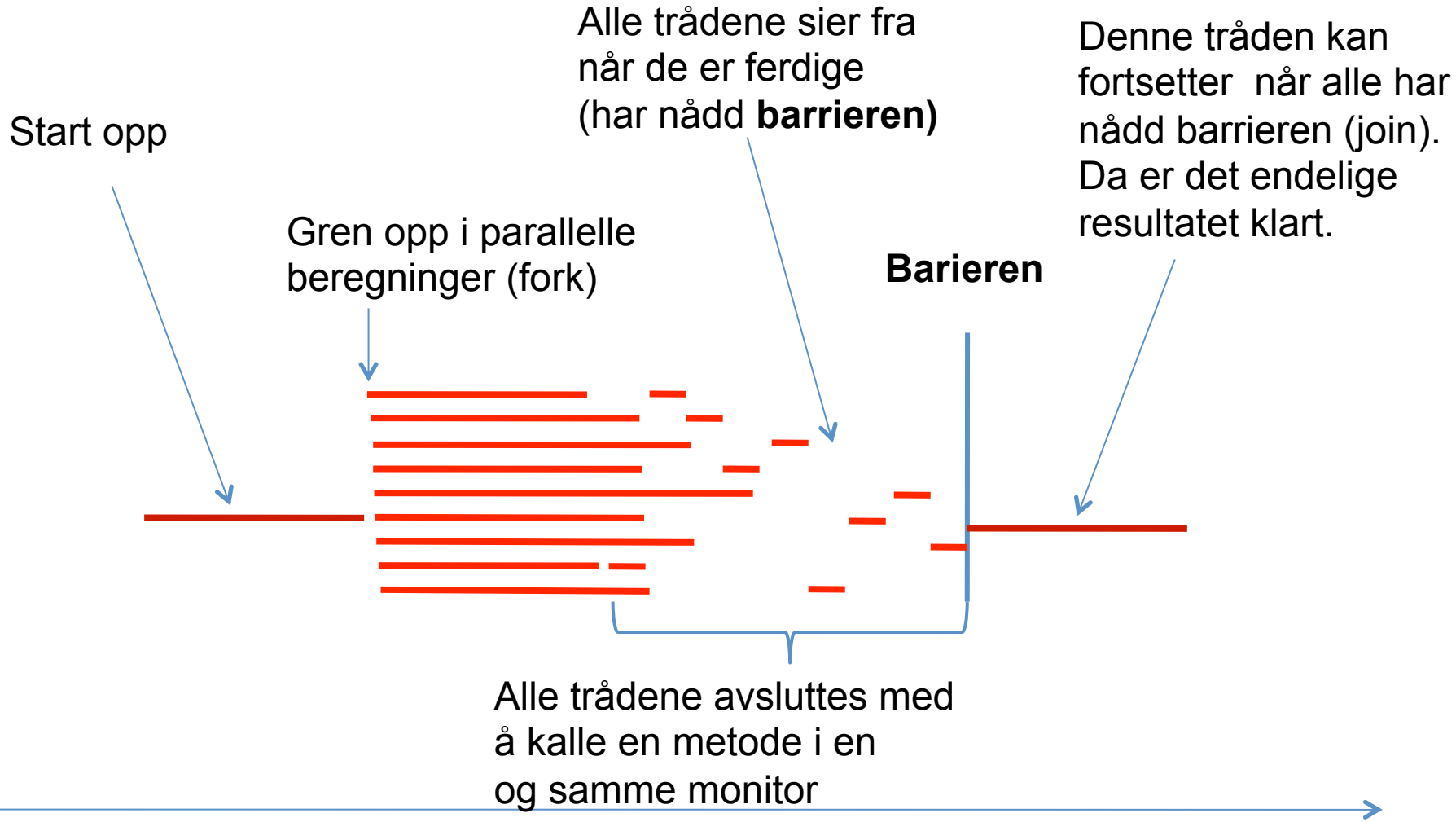


Barrierer i parallellprogrammering

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



Barrieren i dette eksemplet



Kallet `monitor.vent();` i minstR-programmet er en hjemmelaget barriere

Hovedprogrammet versjon med barriere

```
import java.util.concurrent.*;
public class MinstB {

    final int maxVerdiInt = Integer.MAX_VALUE;
    int [ ] tabell;

    final int antallTrader = 64;

    MinstMonitor monitor = new MinstMonitorB();;
    CountDownLatch minBarriere = new CountDownLatch(antallTrader);

    public static void main(String[ ] args) {
        new MinstB();
    }

    public MinstB ( ) {
         tabell = new int[640000];
         for (int in = 0; in< 640000; in++)
                 tabell[in] = (int)Math.round(Math.random()* maxVerdiInt);
         for (int i = 0; i< antallTrader; i++)
              // Lag og start antallTrader tråder
              new MinstTradB(tabell,i*10000,((i+1)*10000)-1,monitor,minBarriere).start();
          //vent på at alle trådene er ferdig:
          try {
               minBarriere.await();  // vent på at alle "Minst-trådene" har nådd barrieren
               System.out.println("Minste verdi var: " + monitor.hentMinste() );
          }
          catch (InterruptedException ex){ }
    }
}
```

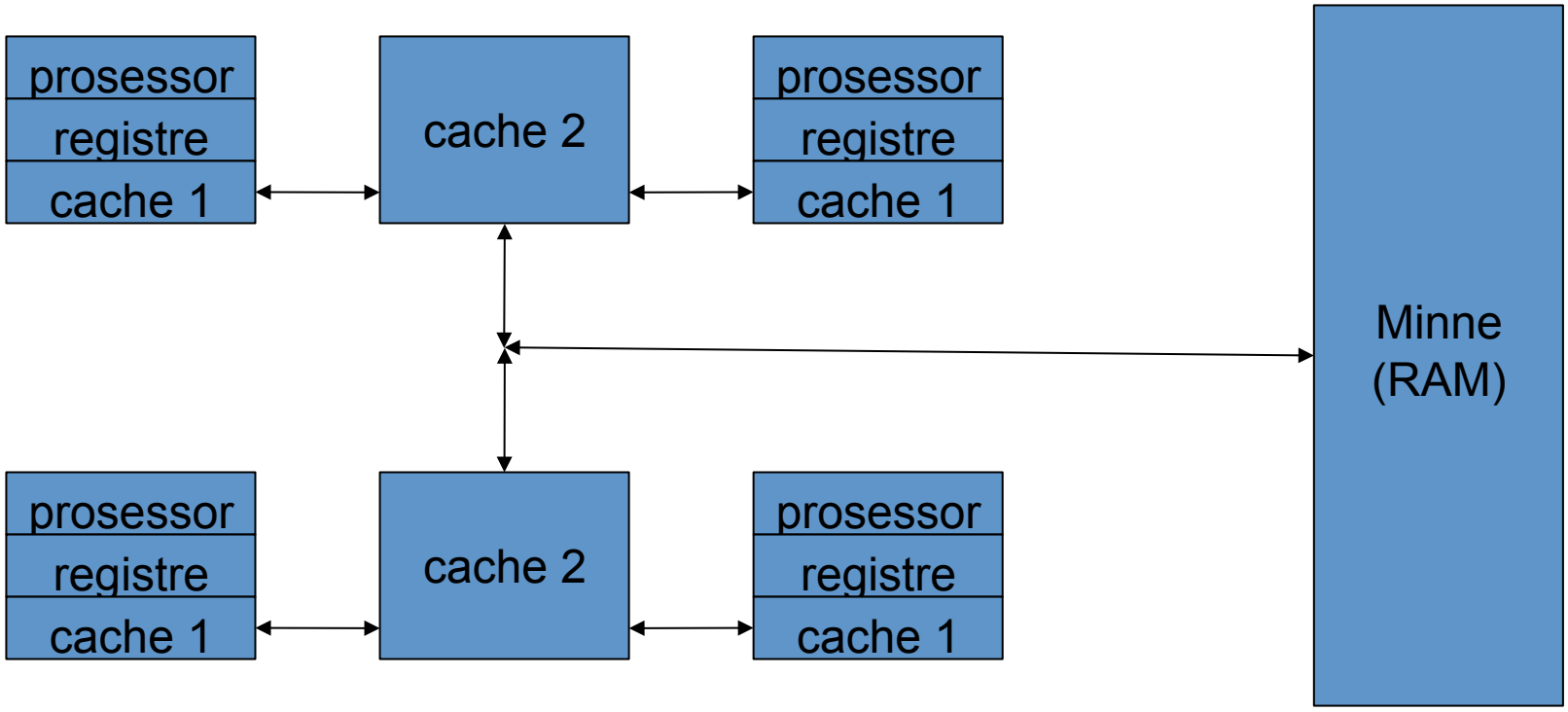
Monitor for resultater

```
class MinstMonitorB implements MinstMonitor {
      private int minstTilNa = Integer.MAX_VALUE;

  void vent(){ // brukes ikke. Bruker barriere isteden.
  }

  synchronized void giMinsteVerdi (int minVerdi) {
      if(minstTilNa > minVerdi) minstTilNa = minVerdi;
  }

  synchronized int hentMinste () {return minstTilNa;}

}
```

Tråder som finner minste tall i del av tabellen

```
class MinstTradB extends Thread {
    int [ ] tab; int startInd, endInd;
    CountDownLatch barriere;
    MinstMonitor  mon;
    MinstTradB(int [ ] tb, int st, int en, MinstMonitor mon,
                                        CountDownLatch barriere) {
        tab = tb; startInd = st; endInd = en;
        this.mon = mon;  this.barriere = barriere;
  }


  public void run(){
     int minVerdi = Integer.MAX_VALUE;
     for ( int ind = startInd; ind <= endInd; ind++)
                  if(tab[ind] < minVerdi) minVerdi = tab[ind];
        // gi minste resultat til monitoren:
      mon.giMinsteVerdi(minVerdi);
        // signaler at denne tråden er ferdig med jobben:
      barriere.countDown(); // sier fra at jeg har nådd barrieren
  } // slutt run

} // slutt MinstTradB
```

Maskinarkitektur, f.eks. 4 prosessorer (repetisjon)



Javas minnemodell (memory model)

- En minnemodel defineres ved å se på lovlige rekkefølger av lese og skriveoperasjoner på variable.
- Innad i en tråd skjer ting i rekkefølgen gitt av programmet, bortsett fra at uavhengige operasjoner kan ”re-ordnes” (for optimalisering).
- Mellom tråder skjer ting ikke i rekkefølge unntatt
  - Ved bruk av volatile variable
  - Alle operasjoner før en ”unlock” skjer i rekkefølge før alle operasjoner etter en etterfølgende ”lock” på den samme monitoren.

Volatile variable

- en variabel som er deklarert volatile caches ikke (og oppbevares ikke i registre (lenger enn helt nødvending)).
  - En volatil variable skrives helt tilbake i primærlageret før neste instruksjon utføres.
- Rekkefølgen av skriv/les-operasjoner på volatile variable og låsing/opplåsing av låser definerer en **sekvensielt konsistent rekkefølge** av alle operasjoner som er en del av semantikken til programmeringsspråket Java.

Synkronisering og felles variable

- Synchronization ensures that memory writes by a thread before or during a synchronized block are made visible in a predictable manner to other threads which synchronize on the same monitor. After we exit a synchronized block, we **release** the monitor, which has the effect of flushing the cache to main memory, so that writes made by this thread can be visible to other threads. Before we can enter a synchronized block, we **acquire** the monitor, which has the effect of invalidating the local processor cache so that variables will be reloaded from main memory. We will then be able to see all of the writes made visible by the previous release.

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

Synkronisering og felles variable

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

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

Og fra definisjonen av Java: Versjon 3 side 579:

17.7 Non-atomic Treatment of double and long

Some implementations may find it convenient to divide a single write action on a 64-bit long or double value into two write actions on adjacent 32 bit values. For efficiency's sake, this behavior is implementation specific; Java virtual machines are free to perform writes to long and double values atomically or in two parts.

For the purposes of the Java programming language memory model, a single write to a non-volatile long or double value is treated as two separate writes: one to each 32-bit half. This can result in a situation where a thread sees the first 32 bits of a 64 bit value from one write, and the second 32 bits from another write.

Writes and reads of **volatile** long and double values are always atomic. Writes to and reads of **references** are always atomic, regardless of whether they are implemented as 32 or 64 bit values.

VM implementors are encouraged to avoid splitting their 64-bit values where possible. **Programmers are encouraged to declare shared 64-bit values as volatile** or synchronize their programs correctly to avoid possible complications.

Mer fra definisjonen av Java: Versjon 3 side 579:

For example, in the following (broken) code fragment, assume that this.done is a non-volatile boolean field:

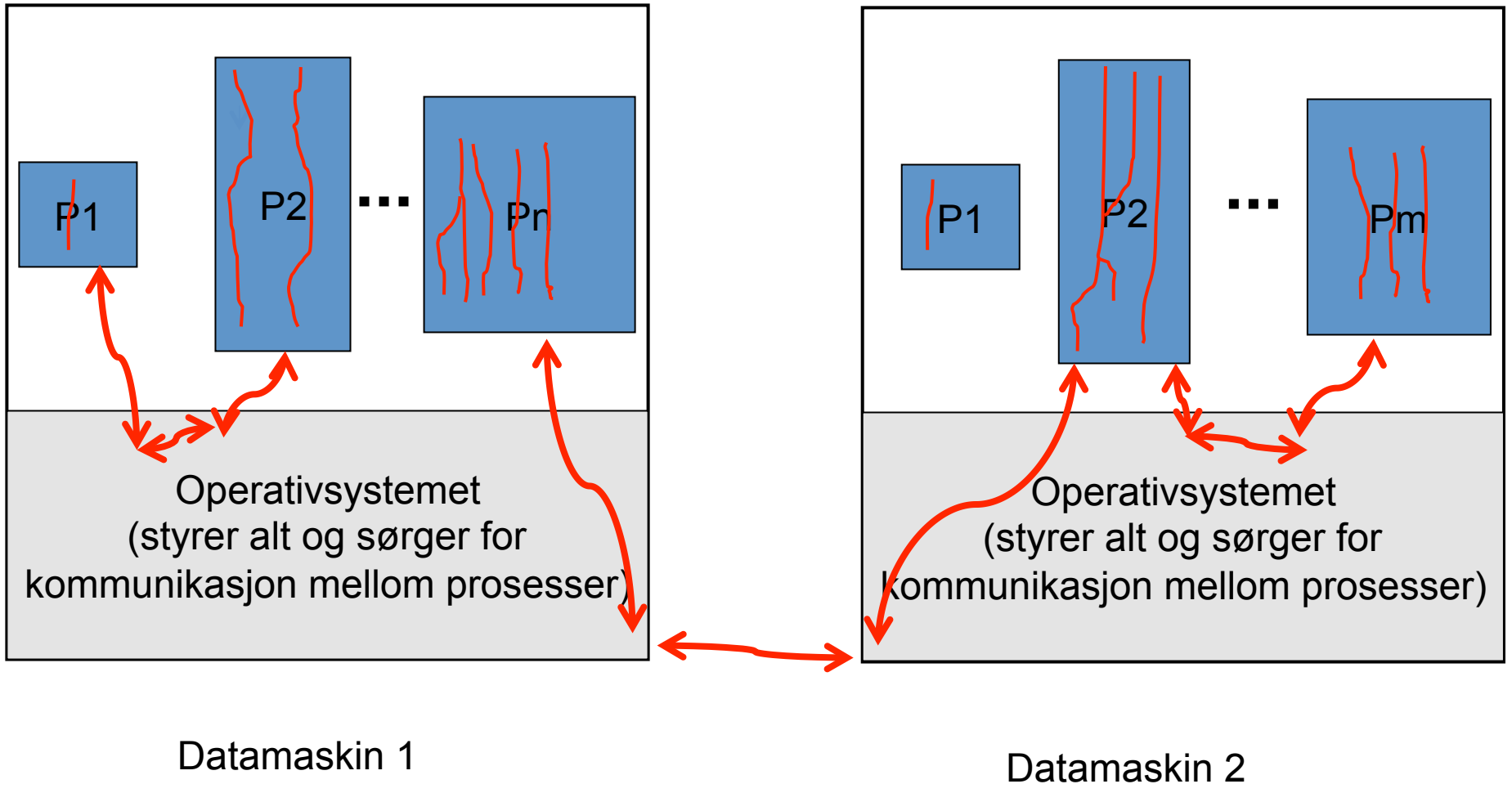
```
while (!this.done)
    Thread.sleep(1000);
```

The compiler is free to read the field this.done just once, and reuse the cached value in each execution of the loop. This would mean that the loop would never terminate, even if another thread changed the value of this.done.

Fordelen med konstanter (immutable objects)

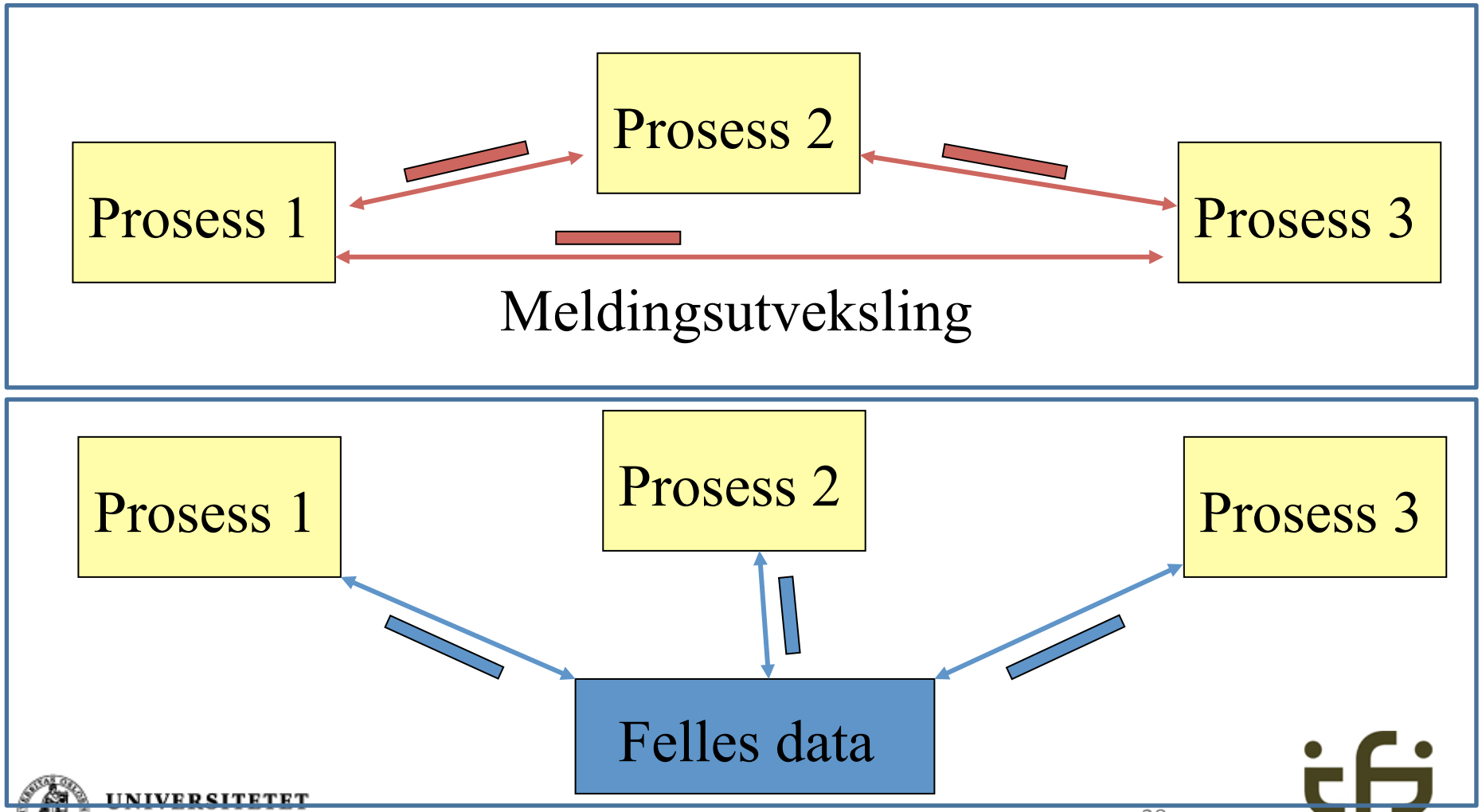
- Konstanter kan det aldri skrives til
- Minnemodellen for konstanter blir derfor veldig enkel.
- Prøv å ha mest mulig konstanter når data skal deles mellom tråder.
- Eksempel: Geografiske data, observasjoner
- Hvis du ønsker å gjøre en forandring:
  - Kast det gamle objektet og lag et nytt.

Prosesser og tråder



Kommunikasjon mellom prosesser

Samarbeidende prosesser sender meldinger til hverandre (brune piler, mest vanlig)
eller leser og skriver i felles lager (blå piler). (Repetisjon)



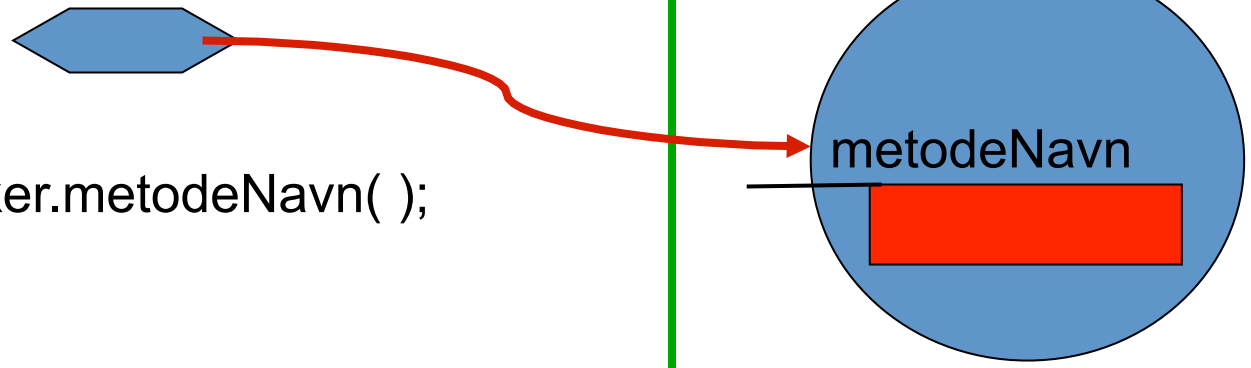
Kommunikasjon mellom Java-**prosesser**: RMI: Remote Method Invocation

(Norsk: Fjern-metode-kall)

Dette ønsker vi å oppnå:

Et Java program
(på en maskin)

```
InterfaceNavn fjernPeker = …….;

. . .
fjernPeker.metodeNavn( );
. . .
```



Et annet Java program
(på en annen maskin)

objekt av en klasse som implementerer *InterfaceNavn*

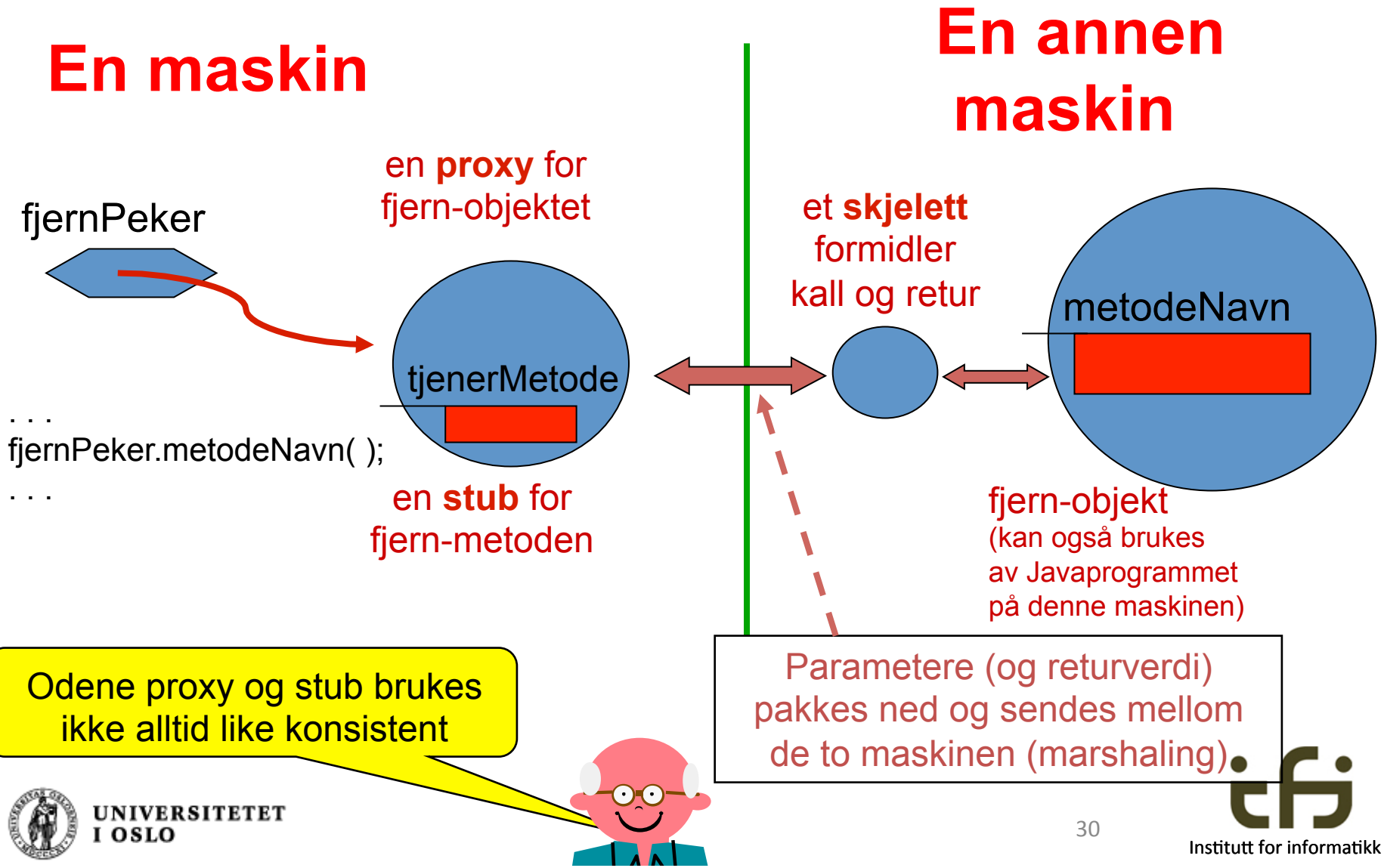
UNIVERSITETET I OSLO

Alternativt navn: RPC: Remote Procedure Call

Institutt for informatikk

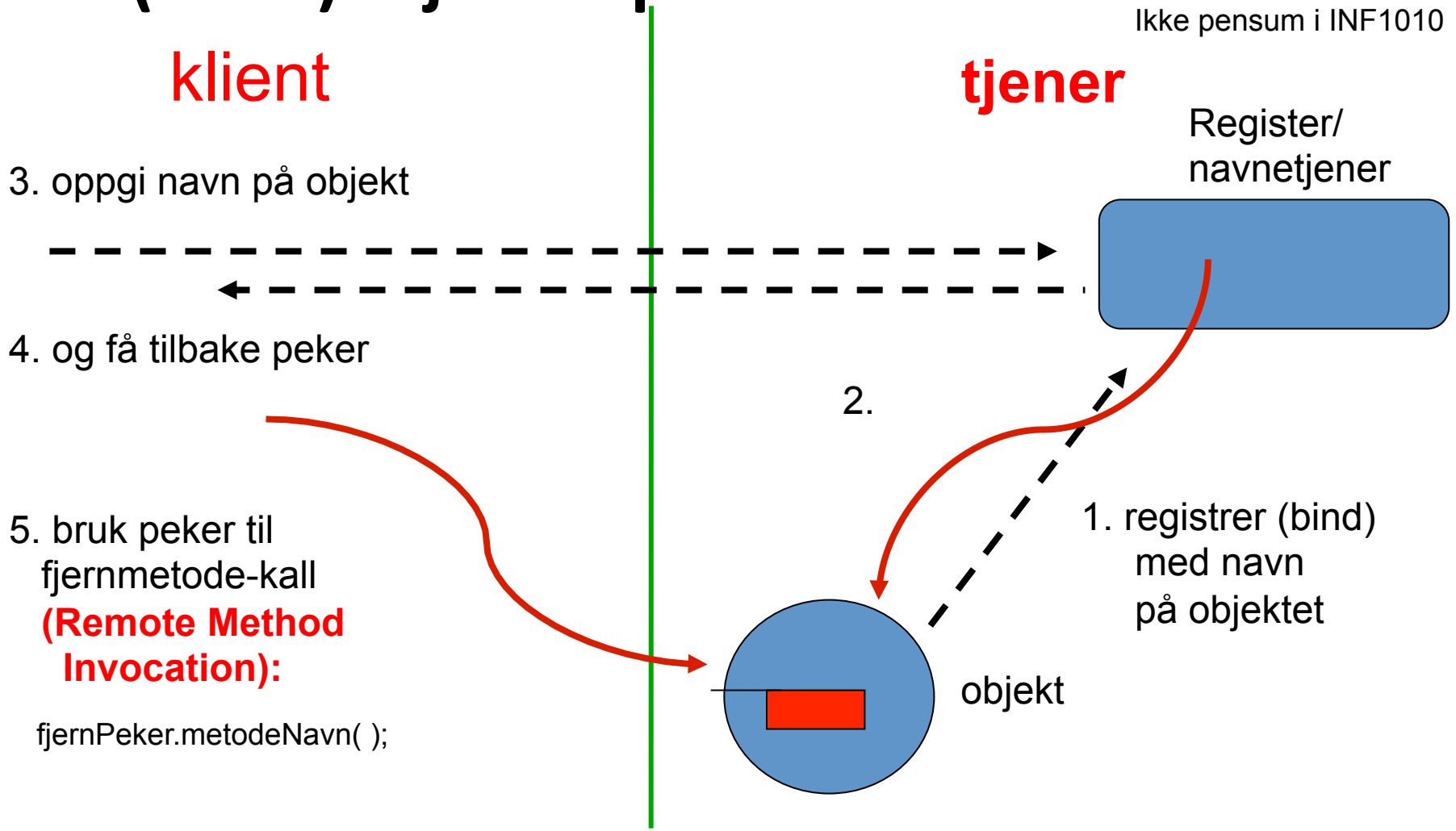
RMI: Implementasjon
(bak kulissene)

Ikke pensum i INF1010



RMI: Hvordan kalle metoder i (Java-)objekter på andre maskiner

Ikke pensum i INF1010

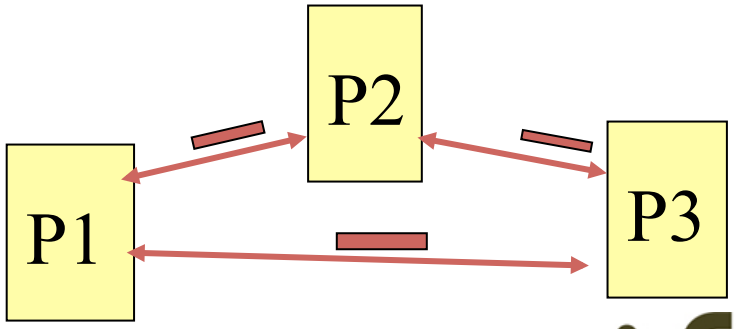


Og det eneste som er kjent på begge maskinene er Interfacet til objektet og navnet

CORBA, MPI

- CORBA (Common Object Request Broker Architecture) er (var) en språkuavhengig måte å definere kommunikasjon mellom fjern-objekter (a la side 30-32)
- Et IDL (Interface Definition Language) definerer grensesnittet til objektene (på samme måte som Interface i Java)
- En IDL-kompilator oversetter til ditt valgte språk (Java, C++, C#, Smalltalk, …)
- Dette gjør at prosesser skrevet i forskjellige objektorienterte språk og som kjører på forskjellige (eller samme) maskin kan kommunisere.

- Språk som ikke er objektorienterte:
  Send en melding
  (MPI – Message-Passing Interface)



Time out fra tråder
To typer invarianter / tilstandspåstander:

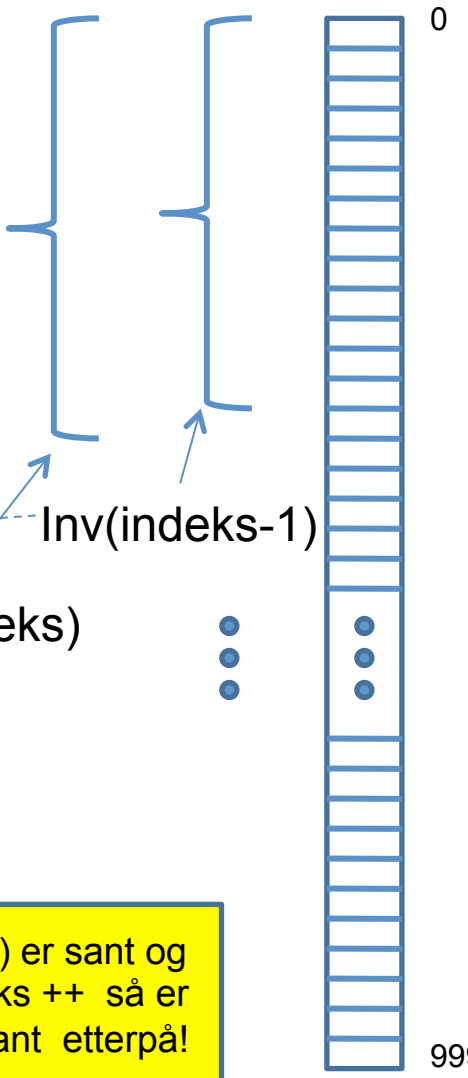
- Når du lager en løkke er det alltid en invariant som sier hvor langt arbeidet i løkka er kommet
- Data i et objekt er alltid styrt av en (eller flere) invarianter eller konsistensregler

- De to neste sidene innholder eksempler på dette

Invarianter på data i løkker

Eksempel: **Finne minste verdi i tabell**

```
// Vi vet ingenting annet enn at tabell [0] til og med
// tabell[999] inneholder tall. Vi skal finne det minste

int minstTilNaa = tabell[0];

  // minstTilNaa inneholder minste verdi i området
  //  fra og med tabell [0] til og med tabell[0]

for (indeks = 1; indeks < 1000; indeks ++) {

    // minstTilNaa inneholder minste verdi i området
    //  fra og med tabell [0] til og med tabell[indeks-1]

    if (minstTilNaa > tabell [indeks] ) minstTilNaa = tabell[indeks]

    // minst TilNaa inneholder minste verdi i området
    //  fra og med tabell [0] til og med tabell[indeks]
}

// Nå er indeks == 1000
 // minstTilNaa inneholder minste verdi i området
 //  fra og med tabell [0] til og med tabell[indeks-1]
// Da følger:
 // minstTilNaa inneholder minste verdi i området
 //  fra og med tabell [0] til og med tabell[999] !!!!!!!
```



Hvis Inv(indeks) er sant og vi utfører: indeks ++ så er Inv(indeks-1) sant etterpå!

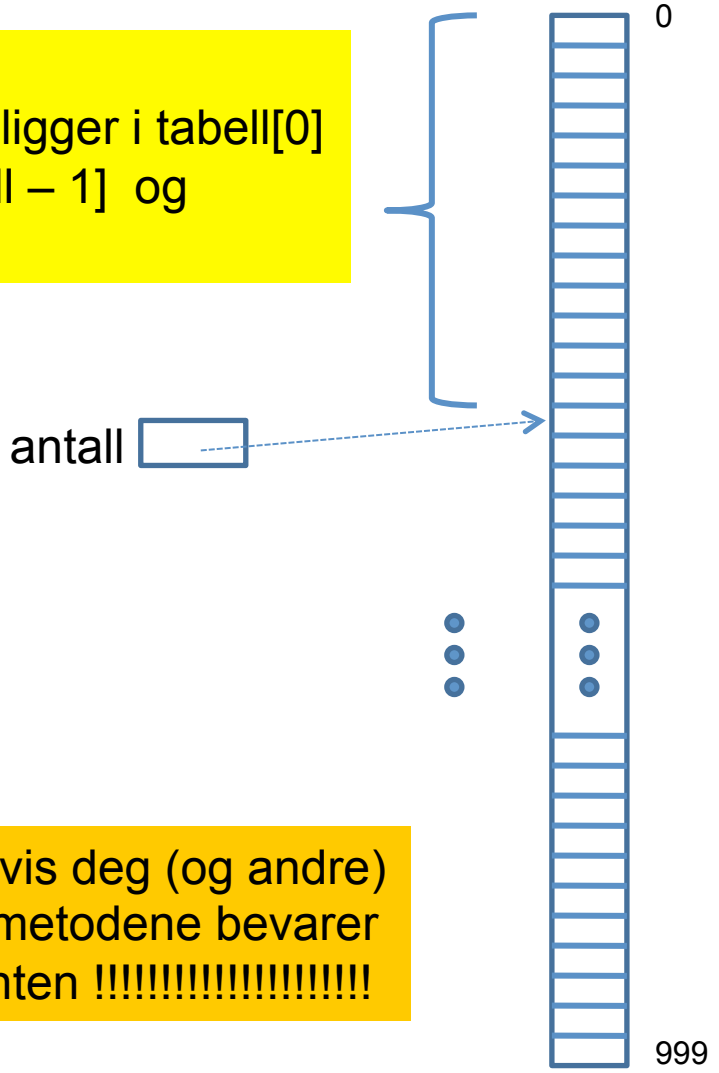
Induksjons-basis
Induksjons-skritt

Invarianter på data i objekter

Invariant:
Alle dataene vi lagrer ligger i tabell[0] til og med tabell [antall – 1] og
0 <= antall <= 1000

```
settInn(x) {
    if (antall == 1000) return ;
    antall ++;
    tabell[antall-1] = x;
}
```

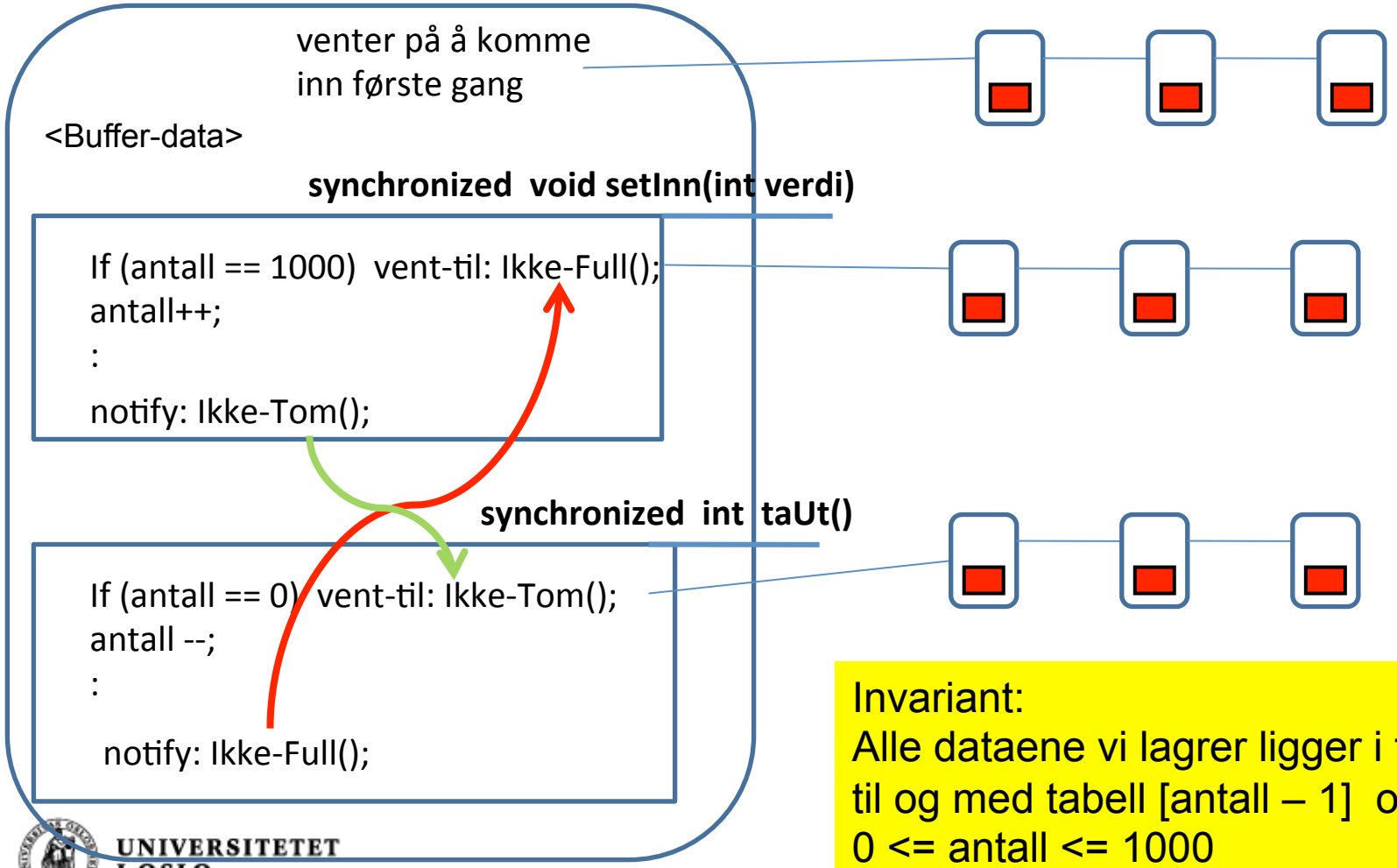
```
taUt ( ) {
    if (antall == 0) return null;
    antall --;
    return (tabell[antall]);
}
```



Overbevis deg (og andre) om at metodene bevarer Invarianten !!!!!!!!!!!!!!!!!!!!

Vi tar opp tråden (☺) fra sidene foran og fra side 7:

Hvordan bevare invarianter på data i objekter når vi ikke har ansvaret alene. Svar: *Vi venter ofte på at andre skal gjøre objektets (monitorens) tilsand hyggeligere.*



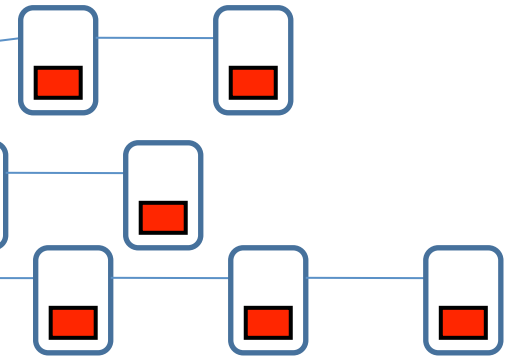
En monitor (et objekt)

Invariant:
Alle dataene vi lagrer ligger i tabell[0] til og med tabell [antall – 1] og
0 <= antall <= 1000

Biblioteket java.concurrent:

Flere køer av ventende tråder

```
Lock laas = new ReentrantLock();

Condition ikkeFull =  laas.newCondition();

Condition ikkeTom = laas.newCondition();


  void settInn ( int verdi)

     laas.lock();
     try {
        while  (full)  ikkeFull.await();
        // nå er det helst sikkert ikke fult
        :
         // det er lagt inn noe, så det er
         //  helt sikkert ikke tomt:
       ikkeTom.signal();
     }  finally {
       laas.unlock()
     }

int taUt ( )
```



En kø for selve låsen og en kø for hver condition-variabel

(import java.concurrent.locks.*)

Metodene er ikke lenger ”synchronized” og vi må låse (og låse opp !) monitoren selv :-(

Java´s API (java.concurrent)

Interface Condition

- void **await**()

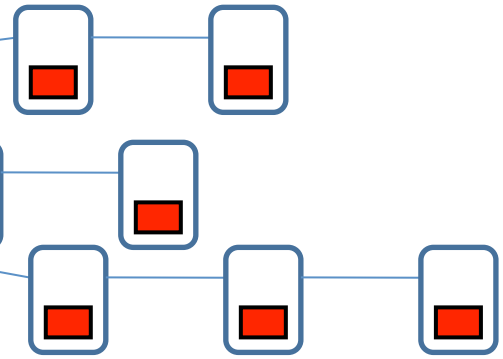
  Causes the current thread to wait
  until it is signalled (or interrupted)

- void **signal**()

  Wakes up one waiting thread.

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

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

void settInn ( int verdi) throws InterrupedException
    laas.lock();
    try {
       while  (full)  ikkeFull.await();          // OK med if ?
       // nå er det helt sikkert ikke fult
       :
        // det er lagt inn noe, så det er helt sikkert ikke tomt:
      ikkeTom.signal();
    }  finally {
      laas.unlock();
    }

int taUt ( )  throws InterrupedException
     laas.lock();
     try {
        while (tom)  ikkeTom.await();        // OK med if ?
        // nå er det helt sikkert ikke tomt;
        :
         // det er det tatt ut noe, så det er helt sikkert ikke fult:
       ikkeFull.signal();
     }  finally {
       laas.unlock();
     }
```



Nå har vi en kø per betingelse som skal oppfylles
(og vi kan til og med få rettferdighet!)

Bra!

Legg merke til bruken av finally

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

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