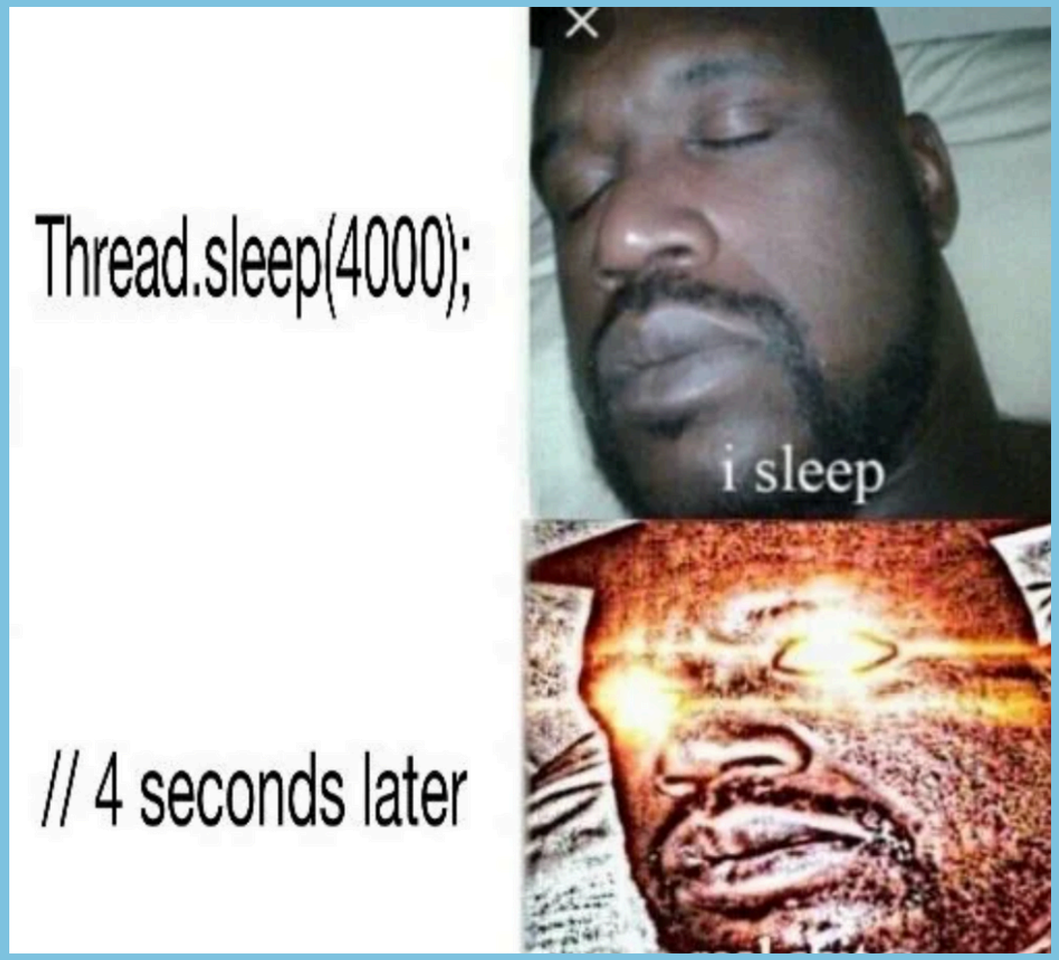
Uke 10 - Tråder 2

gruppe 5 - jonasbny@ifi.uio.no

Agenda



Litt repetisjon

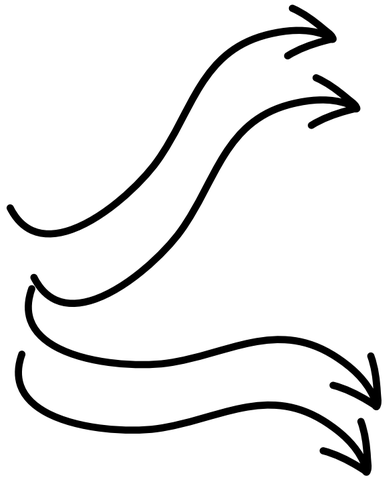
barrierer

join

CountDownLatch

Tråder

Hittil har vi skrevet sekvensielle, ikke-parallelle programmer. Programmer utført av en tråd, *main tråden*



Hvis vi vil at to eller flere oppgaver skal kjøre *samtidig* kan vi lage flere tråder

Vi snakker om *naturlig parallellitet* når det er naturlig for oss å stykke opp en oppgave i flere deloppgaver og løse de samtidig for å komme frem til ønsket resultat

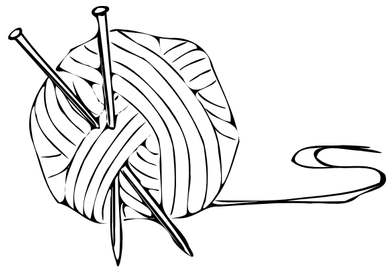
Skal vi lage taco sammen, er det naturlig at en kutter grønnsaker og en annen steker kjøttdeigen

(workerClass må ha implementert Runnable)

Tråder

```
Thread traad = new Thread(new workerClass());
```

Thread er et innebygd objekt i Java som kan starte en ny **tråd**



Som parameter trenger en tråd et objekt som implementerer interfacet **Runnable**

Deretter kan man kalle på metoden start() på tråd objektet for å sette i gang en ny separat tråd

Det er koden i arbeider objektet som utføres i den nye tråden

Oppgave

Hva er riktig måte å lage en arbeiderklasse?

```
public class workerClass implements Runnable{
```

eller

```
public class workerClass extends Thread{
```

Oppgave

Man kan gjøre begge deler!

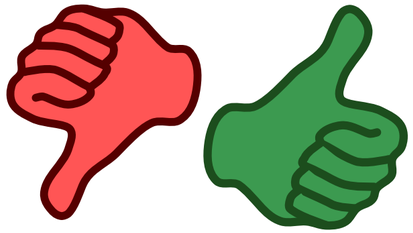
```
public class workerClass implements Runnable{
```

eller

```
public class workerClass extends Thread{
```

Oppgave

Fordeler eller ulemper med de?



```
public class workerClass implements Runnable{
```

eller

```
public class workerClass extends Thread{
```

Arbeiderklasse



Arbeider-klassen kan hete akkurat hva vi vil, men gjerne noe beskrivende

Klassen må implementere **Runnable**, som krever at vi skriver metoden **run()**

Det er i **run()** metoden vi skriver koden vi vil at arbeideren skal utføre

```
@Override
public void run(){
    // ...
}
```

Synkroniseringsproblemer

Dersom to eller flere tråder deler data, kan vi få uheldige resultater om vi ikke lar bare en tråd utføre endringer på dataene av gangen og de prøver å oppdatere samme verdi *samtidig*

Dette kalles *race condition* og er kanskje den største kilden til feil i parallelle programmer

Løsningen får vi ved å bruke låser

(Noen ganger klarer de det, noen ganger ikke:)

```
Expected result: 20000
starting thread, shared counter: 0
starting thread, shared counter: 0
done!, shared counter: 11070
done!, shared counter: 15668
```

```
Expected result: 20000
starting thread, shared counter: 0
starting thread, shared counter: 0
done!, shared counter: 10000
done!, shared counter: 20000
```

Låser og Monitorer

For å løse synkroniseringsproblemer sørger vi for at kun en tråd kan aksessere et delt objekt til en gitt tid

Vi lager en kritisk region hvor kun en tråd kan gjøre endringer av gangen, andre tråder må stoppe og vente på sin tur

Man kan se for seg at man har dører på hver side av koden eller dataen man vil beskytte

Hvis en tråd låser døren når de går inn og skal gjøre endringer, må de andre trådene vente på at tråden som låste låser opp døren igjen



Vente på tråder

Den vanligste måten å vente på tråder på er å bruke **.join()**

Da står vi (main tråden) og venter helt til en gitt tråd er ferdig

Men kanskje vi vi samle alle trådene mer enn en gang, eller kanskje vi vil vente på main tråden



Da kan vi bruke **barrierer**

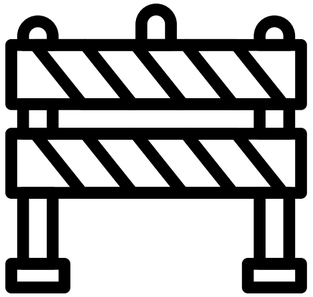
I IN1010 er de mest relevante **CountDownLatch** og **Join()**

(For ekstra interesserte kan man også se på **CyclicBarrier**)

Barrierer

Barrierer i programmering er synkroniseringsmekanismer som sørger for at tråder møtes ved et bestemt punkt i programmet.

De sikrer at tråder ikke går videre før alle trådene har nådd en “barriere”



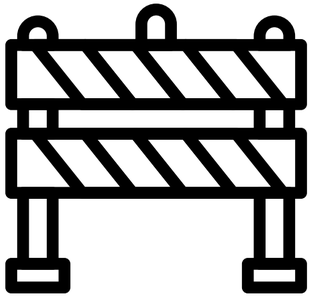
Så med barrierer kan man kontrollere rekkefølgen av operasjoner på en trygg måte i programmer, og de er derfor essensielle innen paralellprogrammering

eksempel på barriere

.Join()

Taco scenario

Vi har en rekke deloppgaver som må gjøres før vi kan si at vi har oppnådd ønsket resultat

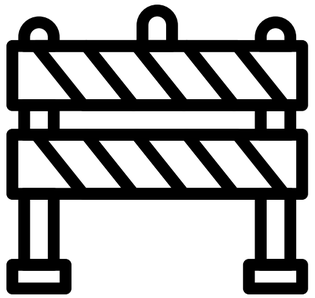


Selve serveringen av taco kan anses som en barriere alle oppgavene må nå før vi kan si vi har oppnådd resultatet vi vil ha

Vi kan kalle **join()** på alle trådene main tråden skal vente på

Barrieren sikrer at ingen elementer blir glemt, at vi forhindrer det som i programmering kalles race condition

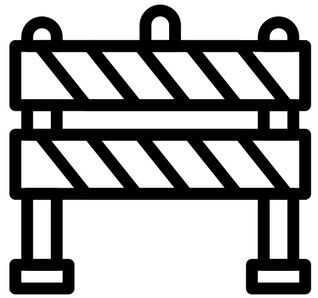
.Join()



join()

Waits for this thread to die.

.Join()



join()

Waits for this thread to die.

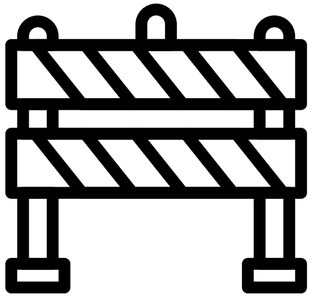
join(long millis)

Waits at most `millis` milliseconds for this thread to die.

CountDownLatch

En **CountDownLatch** er litt som en “veisperring” som krever at et gitt antall tråder står og venter før den åpner seg

Når man oppretter en **CountDownLatch** må man gi den et tall som representerer hvor mange ganger man må “telle ned” før bommen åpnes

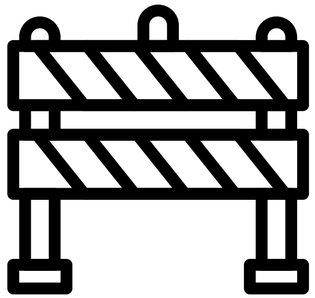


Dette kan brukes for eksempel for å gjøre at alle trådene må vente på at et “startsignal” fra main tråden før de setter i gang

Man kan også bruke det om man vil at main tråden skal vente til trådene er ferdige med *en del* av oppgavene sine

oppgave

Hvilke fordeler og ulemper har barrierer sammenlignet med monitorer?



Barrierer i kombinasjon med monitorering

Barrierer kan fint brukes i kombinasjon med monitorer

Ved å kombinere bruker vi dermed to former for synkronisering. En i form av barrierer og en i form av kritiske regioner med låsing i en monitor.

Det kan være nyttig i tilfeller der tråder deler data, samtidig som flere tråder må vente på andre tråder underveis

Ukas shoutout

La Cetta på bislett, god kebab as



Oppgaver

