IN1010 Repetisjonstime

Tema: Tråder

Parallellitet

- Hittil har vi kun skrevet sekvensielle programmer
  - Koden blir utført fra topp til bunn, altså er det ingenting som skjer samtidig, men i en bestemt rekkefølge
- Vi kan få til *parallellitet* i programmene ved å lage oppgaver som kan utføres samtidig (i parallell)
- Det kan være naturlig å ta en oppgave, og stykke den inn i flere deloppgaver som kan løses samtidig
  - Divide and conquer (ish)
- Hvis en gjeng lager taco, så er det naturlig at én kutter grønnsaker, én steker kjøttdeig også videre, og at dette skjer samtidig, ikke én etter én

Tråder

- Hvordan oppnå parallellitet i Java? TRÅDER!!!
- En tråd er en sekvens med instruksjoner i et program
  - Noe kjent: main-metoden er en tråd som kjøres
- Vi kan kjøre flere tråder samtidig for å oppnå parallellitet
- Tråder kan vente, sove og avbrytes
- IN1010: Trådene skal ikke kommunisere med hverandre (men det er en mulighet)

Tråder i Java

- Tråder utfører *oppgaver*, og vi kan definere hva disse oppgavene som skal være
  - Eget interface: *Runnable*
  - Runnable tilbyr én metode, *run()*
- Vi kan lage egne klasser som implementerer runnable, og definere hva trådene skal gjøre i run() metoden
- For å lage en tråd bruker vi klassen *Thread*, og gir objektet en instans av oppgaveklassen vi lagde
- Når vi kaller på metoden *start()* på tråden, så vil den kjøre run() metoden

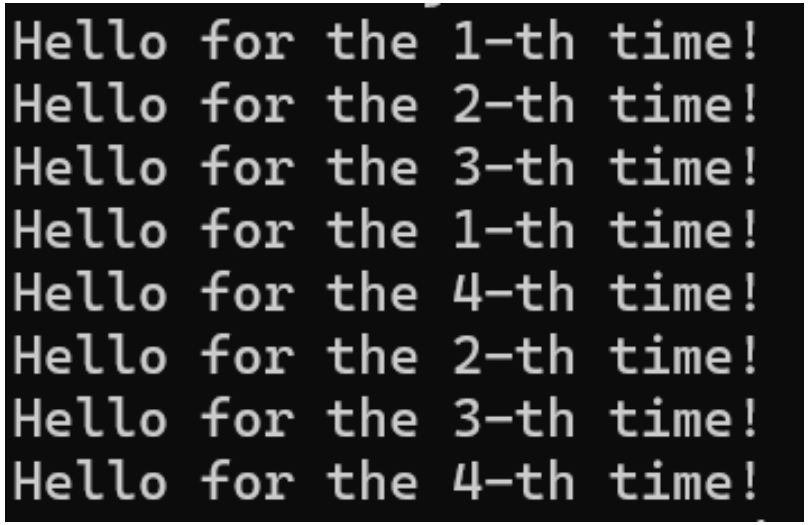
```java
// Eksempel med Runnable

public class MyTask implements Runnable {
    /* Say hello, but think for a while between each time. */
    public void run() {
        for (int i = 1; i < 5000; i++)
            if (i % 1000 == 0)
                System.out.printf("Hello for the %d-th time!\n", i / 1000);
    }
}

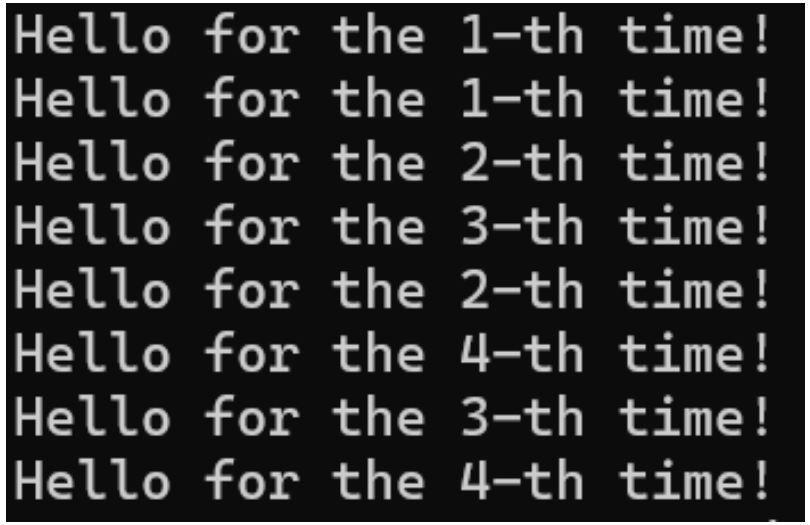
// Eksempel med thread og start fra main-tråden

public Main {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        Runnable task = new MyTask();
        Thread thread1 = new Thread(task);
        Thread thread2 = new Thread(task);
        thread1.start(); thread2.start(); // Will make a call to task.run()
    }
}
```

Eksempelutskrift 1



Eksempelutskrift 2



Merk at trådene kan oppføre seg litt annerledes hver gang

Hva skjer hvis to tråder prøver å oppdatere en verdi samtidig?

```java
public class MyTask implements Runnable {
    private final int MAX_COUNT = 10000;
    private int sharedCounter = 0;

    public void run() {
        System.out.println("Starting! Shared counter = " + sharedCounter);
        for (int i = 0; i < MAX_COUNT; i++) {
            sharedCounter = sharedCounter + 1;
        }
        System.out.println("Done! Shared counter = " + sharedCounter);
    }
}

public class Hoved {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        Runnable task = new MyTask();
        Thread thread1 = new Thread(task);
        Thread thread2 = new Thread(task);
        thread1.start(); thread2.start(); // Will make a call to task.run()
    }
}
```

Utskrift

```
Done! Shared counter = 10770
Done! Shared counter = 16738
```

Forventet utskrift var at tråden som er ferdig sist skriver ut 2 * MAX_COUNT

Problemet oppstår siden to tråder prøver å oppdatere samme verdi samtidig. Dette kalles en race condition og kan løses med låser.

Monitor

- En monitor kan ses på som en slags 'hub' for alle trådene, hvor felles ressurser blir lagret
- Disse felles ressursene (kritiske regioner) kan aksesseres av alle trådene.
  - Monitoren skal forhindre race conditions
- Vi gir monitoren en *ReentrantLock*, slik at vi kan låse monitoren når vi vil.
  - Vi ønsker å låse monitoren hver gang en kritisk region skal aksesseres.
  - Når monitoren låses, så kan kun én tråd bruke monitorens metoder om gangen.
- Slik kan vi holde litt orden over de kritiske regionene.

Merk: Monitoren trenger kun å låses når en tråd skal aksessere en kritisk region!

```java
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class CounterMonitor {
    private Lock lås = new ReentrantLock();
    private int sharedCounter = 0; // Kritisk region

    public void incrementCounter() {
        lås.lock();
        try {
            sharedCounter++;
        } finally {
            lås.unlock();
        }
    }

    public int getCounter() {
        lås.lock();
        try {
            return sharedCounter;
        } finally {
            lås.unlock();
        }
    }
}
```

```java
public class IncrementCounter implements Runnable {
    private CounterMonitor monitor;
    private final int MAX_COUNT = 10000;
    private static int nr = 0;
    private int threadNr;

    public IncrementCounter(CounterMonitor m) {
        monitor = m;
        nr++;
        threadNr = nr;
    }

    @Override
    public void run() {
        while (monitor.getCounter() < MAX_COUNT) {
            monitor.incrementCounter();
            System.out.println("Thread " + threadNr + ": " + monitor.getCounter());
        }
        System.out.println(threadNr + " Done!");
    }
}
```

```java
public class Hoved {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        CounterMonitor m = new CounterMonitor();
        Thread t1 = new Thread(new IncrementCounter(m));
        Thread t2 = new Thread(new IncrementCounter(m));
        Thread t3 = new Thread(new IncrementCounter(m));
        Thread t4 = new Thread(new IncrementCounter(m));
        Thread t5 = new Thread(new IncrementCounter(m));
        Thread t6 = new Thread(new IncrementCounter(m));
        t1.start();
        t2.start();
        t3.start();
        t4.start();
        t5.start();
        t6.start();
    }
}
```

```
Thread 2: 9988
Thread 2: 9989
Thread 2: 9990
Thread 2: 9991
Thread 2: 9992
Thread 2: 9993
Thread 2: 9994
Thread 2: 9995
Thread 2: 9996
Thread 2: 9997
Thread 2: 9998
Thread 2: 9999
Thread 2: 10000
Thread 3: 9729
Thread 4: 9986
Thread 1: 9979
Thread 6: 9963
Thread 5: 9901
2 Done!
4 Done!
1 Done!
6 Done!
5 Done!
3 Done!
```

Conditions

- Noen ganger har vi produsent- og konsumenttråder som kjører samtidig, altså tråder som legger til ting og tråder som fjerner ting
- Da kan det hende at en konsument prøver å fjerne noe før produsenten har lagt det til, og det vil vi ikke ha.
- En løsning er å la tråden terminere, men en bedre løsning er å la konsumenten *vente* på at verdien legges til (så lenge produsenttrådene fortsatt kjører)
- Dette kan gjøres ved å lage en *condition* som trådene venter på
  - signalAll for å signalisere trådene som venter

```java
public class CounterMonitor {
    private Lock lås = new ReentrantLock();
    private Condition ikkeNull = lås.newCondition();
    private int sharedCounter = 0; // Kritisk region

    public void incrementCounter() {
        lås.lock();
        try {
            sharedCounter++;
            ikkeNull.signalAll();
        } finally {
            lås.unlock();
        }
    }

    public void decrementCounter() {
        lås.lock();
        try {
            while (sharedCounter < 1) {
                ikkeNull.await();
            }
            sharedCounter--;
        } catch (InterruptedException e) {
            System.out.println("Womp");
        } finally {
            lås.unlock();
        }
    }
```

```java
public class IncrementCounter implements Runnable {
    private CounterMonitor monitor;
    private final int MAX_COUNT = 10000;
    private static int nr = 0;
    private int threadNr;

    public IncrementCounter(CounterMonitor m) {
        monitor = m;
        nr++;
        threadNr = nr;
    }

    @Override
    public void run() {
        for (int i = 0; i < MAX_COUNT; i++) {
            monitor.incrementCounter();
            System.out.println("I-Thread " + threadNr + ": " + monitor.getCounter());
        }
        System.out.println(threadNr + " Done!");
    }
}
```

```java
public class DecrementCounter implements Runnable {
    private CounterMonitor monitor;
    private final int MAX_COUNT = 10000;
    private static int nr = 0;
    private int threadNr;

    public DecrementCounter(CounterMonitor m) {
        monitor = m;
        nr++;
        threadNr = nr;
    }

    @Override
    public void run() {
        for (int i = 0; i < MAX_COUNT; i++) {
            monitor.decrementCounter();
            System.out.println("D-Thread " + threadNr + ": " + monitor.getCounter());
        }
        System.out.println(threadNr + " Done!");
    }
}
```

```java
public class Hoved {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        CounterMonitor m = new CounterMonitor();
        Thread t1 = new Thread(new IncrementCounter(m));
        Thread t2 = new Thread(new IncrementCounter(m));
        Thread t3 = new Thread(new DecrementCounter(m));
        Thread t4 = new Thread(new DecrementCounter(m));

        t1.start();
        t2.start();
        t3.start();
        t4.start();

    }
}
```

```
D-Thread 1: 1483
D-Thread 1: 1482
D-Thread 1: 1481
D-Thread 1: 1480
D-Thread 1: 1479
I-Thread 1: 1493
I-Thread 1: 1479
I-Thread 2: 1492
I-Thread 1: 1480
I-Thread 1: 1482
I-Thread 1: 1483
I-Thread 1: 1484
D-Thread 1: 1478
D-Thread 1: 1484
D-Thread 1: 1483
D-Thread 1: 1482
D-Thread 1: 1481
D-Thread 1: 1480
D-Thread 1: 1479
D-Thread 1: 1478
D-Thread 1: 1477
I-Thread 1: 1485
I-Thread 2: 1481
I-Thread 2: 1478
```

```
D-Thread 1: 26
D-Thread 1: 25
D-Thread 1: 24
D-Thread 1: 23
D-Thread 1: 22
D-Thread 1: 21
D-Thread 1: 20
D-Thread 1: 19
D-Thread 1: 18
D-Thread 1: 17
D-Thread 1: 16
D-Thread 1: 15
D-Thread 1: 14
D-Thread 1: 13
D-Thread 1: 12
D-Thread 1: 11
D-Thread 1: 10
D-Thread 1: 9
D-Thread 1: 8
D-Thread 1: 7
D-Thread 1: 6
D-Thread 1: 5
D-Thread 1: 4
D-Thread 1: 3
D-Thread 1: 2
D-Thread 1: 1
D-Thread 1: 0
1 Done!
```

Barrierer

- En barriere er en synkroniseringsmekanisme som gjør at tråder kan møtes på et bestemt punkt i programmet
- Hvis vi tråder kjører i parallell, så vil vi ofte gå tilbake til sekvensiell kjøring etter trådene er ferdige
- Barrierer kan sikre at én eller flere tråder ikke går videre før én eller flere andre tråder har nådd barrieren.
- De gjør at vi kan kontrollere rekkefølgen av operasjoner på en trygg måte.
- To typer barrierer (i IN1010):
  - .join()
  - CountDownLatch

.join()

- .join() er en funksjon som en tråd kan kalle på
- Funksjonen gjør at tråden som funksjonen blir kalt *i* vil vente på at tråden som blir kalt *på* er ferdig
  - For eksempel: Hvis vi skriver *traad.join()* i main, så vil main-tråden vente på at *traad* kjører ferdig før den fortsetter å kjøre
  - Uten en barriere så kunne main-metoden potensielt prøvd å gjøre operasjoner på en verdi produsert av *traad* før verdien er blitt produsert

```java
public class TacoPreparation {
    public static void main(String[] args) {
        Thread vegetableThread = new Thread(new ChopVegetablesTask());
        Thread meatThread = new Thread(new CookMeatTask());
        Thread tortillaThread = new Thread(new HeatTortillasTask());
        vegetableThread.start();
        meatThread.start();
        tortillaThread.start();

        try {
            vegetableThread.join();
            meatThread.join();
            tortillaThread.join();
            System.out.println("All tasks are completed! Serve the tacos!");
        } catch (InterruptedException e) {
            System.err.println("Main thread was interrupted. Dinner is cancelled!");
            System.exit(1);
        }
    }}
```

Livekode

Frukt-marked