Tråder 1

▼ Viktige begreper og metoder

Begreper

- **Felles ressurs:** noe som flere tråder er interessert i å bruke.
- **Kritisk region**: Kode som *endrer* en felles ressurs (metoden settInn i vårt tilfelle), kalles en **kritisk region**. For å unngå feil må kun én tråd utføre den kritiske regionen av gangen. Kritiske regioner kan beskyttes med en lås
- **Kappløp**: Ved manglende sikring av kritiske regioner kan det oppstå **kappløp** («race condition»). Flere tråder prøver å håndtere samme verdi samtidig

Metoder

- `.start()`: kaller på *.run()* til objekt som er sendt med tråden (klasse som implementerer *Runnable*)
- `.join()`: main venter på at denne tråden skal bli ferdig før main fortsetter
- `.interupt()`: avbryter kjøringen til tråden
- `Thread.sleep('millisekunder')`: “sove”, vente i en tid. 1000 gir 1 sek
- `.isAlive()`: true/false om .run() kjører

En tråd er en sekvens av instruksjoner i et program ("thread of execution").

Hittil har vi skrevet vanlig, ikke-parallelle programmer. Disse har hatt én sekvens av instruksjoner, altså én tråd.

Hvis vi ønsker at to eller flere oppgaver (sett med instrukser) skal kjøre samtidig, kan vi lage flere tråder. Disse oppgavene vil da kunne utføres i parallell.

Ulike tråder kan dele på objekter (minne), og "snakke" med hverandre. I IN1010 skal trådene ikke kommunisere direkte, kun bruke delte objekter.

Thread = worker, Runnable = task

Vi kan tenke på en tråd som en arbeider - noe som kan utføre et sett med instrukser, en oppgave. Arbeiderne er objekter av klassen Thread.

Oppgavene som trådene kan utføre er objekter av grensesnittet Runnable.

```
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);
    }
}
```

Thread

Vi lager nye arbeidere ved å lage et objekt av klassen Thread. Vi kan så gi tråden en oppgave den kan utføre, og så be den starte.

```
public Main {
    public static void main(String[] args) {
        Runnable task = new MyTask();
        Thread worker1 = new Thread(task);
        Thread worker2 = new Thread(task);
        worker1.start();  // Will make a call to task.run()
        worker2.start();
    }
}
```

- *Trådene worker1 og worker2 utfører samme oppgave parallelt.*

Vi skiller mellom tråder og oppgaver fordi vi ønsker å kunne gjenbruke *samme* tråder på *forskjellige* oppgaver, eller samme oppgaver utført av flere tråder.

Synkronisering

Dersom to tråder har tilgang på delt data, kan vi få uheldige resultater dersom vi ikke passer på.

```java
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);
    }
}
```

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

```
Starting! Shared counter = 0
Starting! Shared counter = 0
Done! Shared counter = 10297
Done! Shared counter = 11953
```


```java
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);
    }
}
```

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

```
Starting! Shared counter = 0
Starting! Shared counter = 0
Done! Shared counter = 10297
Done! Shared counter = 11953
```

Problemet er at begge trådene forsøker å oppdatere samme verdi *samtidig*. Dette kalles en *race condition*, og er (kanskje) den største kilden til feil i parallelle programmer.

→ Løsningen:

Vi løser problemet ved å sørge for at kun en tråd kan aksessere et delt objekt til en gitt tid. Vi lager en såkalt *mutex*, en kritisk region hvor kun en tråd er om gangen. Andre tråder må stoppe og vente på sin tur.

Locks

```
// Vi trenger en lås fra Java-biblioteket:
import java.util.concurrent.locks.*;

// Vi må opprette en lås:
Lock laas = new ReentrantLock(true);

// Vi kan nå låse den kritiske aksessen:
laas.lock();

// Etterpå må vi låse opp igjen:
laas.unlock();

// Try-finally sikrer at låsen ALLTID blir låst opp
```

Vi lager en mutex med *låser*. Tenk at du har dører på hver side av koden du vil beskytte. Hvis trådene låser døren når de går inn, må de andre trådene vente til tråden som låste låser opp igjen.

Vi bruker java.util.concurrent.locks.Lock, et grensesnitt som spesifiserer (blant annet) to metoder: lock() og unlock(). Den faktiske klassen vi lager låsobjekter av heter java.util.concurrent.locks.ReentrantLock.

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

// Main uendret
public class Main {
    public static void main(String[] args) {
        Runnable task = new MyTask();
        Thread worker1 = new Thread(task);
        Thread worker2 = new Thread(task);
        worker1.start();  // Will make a call to task.run()
        worker2.start();
    }
}

class MyTask implements Runnable {
    private final Lock lock = new ReentrantLock();
    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++) {
            lock.lock();
            try {
                sharedCounter = sharedCounter + 1;
            } finally {
                lock.unlock();
            }
        }
        System.out.println("Done! Shared counter = " + sharedCounter);
    }
}
```

Output:

```
Starting! Shared counter = 0
Starting! Shared counter = 0
Done! Shared counter = 16792
Done! Shared counter = 20000
```

→ Nå kan en og en tråd bytte på å oppdatere telleren, slik at vi får ønsket oppførsel.

Legg merke til bruk av try{...} finally{lock.unlock();}. Dette sørger for at
låsen blir låst opp, uansett hva som måtte gå galt mens låsen var låst.
Dette er en sikkerhetsmekanisme for å unngå *deadlocks* (alle venter på låsen, ingen kan låse opp). Ikke alltid nøvdvendig, men slik slipper vi feil.