Plan for timen (fra fjorårets opplegg)

Tema:

- Enum
- Tråder

Enum:
Når vi ønsker at en variabel skal være av en forhåndsbestemt type konstant. Ved hjelp av et enum så bestemmer vi at variablen må være av de forhåndsbestemte typene. Feks: Deklareres som en klasse eller interface, men nå med ordet “enum”.

```
public enum Dag {
       MANDAG, TIRSDAG, ONSDAG, TORSDAG, FREDAG, LORDAG, SONDAG
}

public class EnumTest {
    Dag dag;

    public EnumTest(Dag dag) {
        this.day = day;
    }

    public void tellItLikeItIs() {

            if (day == Dag.MANDAG){
                System.out.println("Mondays are bad.");
             }

            else if (dag == Dag.FREDAG){
                System.out.println("Fridays are better.");
            }

            else if (dag == Dag.LORDAG || dag == Dag.SONDAG){
                System.out.println("Weekends are best.");
            }

            else {:
                System.out.println("Midweek days are so-so.");
            }
        }
    }

    public static void main(String[] args) {
        EnumTest firstDay = new EnumTest(Dag.MANDAG);
        firstDay.tellItLikeItIs();
        EnumTest thirdDay = new EnumTest(Dag.ONSDAG);
        thirdDay.tellItLikeItIs();
        EnumTest fifthDay = new EnumTest(Dag.FREDAG);;
        fifthDay.tellItLikeItIs();
```

```
        EnumTest sixthDay = new EnumTest(Dag.LORDAG);
        sixthDay.tellItLikeItIs();
        EnumTest seventhDay = new EnumTest(Dag.SONDAG);
        seventhDay.tellItLikeItIs();
    }
}
```

Tråder:

- Basics (~15 min)
    - Kort om hva tråder er
    - Hvordan vi lager og bruker tråder i Java.
- Synkronisering (~30 min)
    - Problemet med samtidig aksess av delt data
    - Løsningen
        - Locks
        - Monitors
        - Conditions
- Live-koding: Brusautomat/SodaMachine (~45 min). Livekoding eller deler av programmet som oppgave!

Basics

Hva er en tråd?

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 INF1010 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`.

`Runnable`

Vi beskriver oppgaver som kan utføres ved objekter av grensesnittet `Runnable`:

```
interface Runnable {
    void run();
}
```

Eksempel:

```
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.

Eksempel:

```
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();
    }
}
```

Output (eksempel, ikke alltid samme utskrift):

```
Hello for the 1-th time!
Hello for the 1-th time!
Hello for the 2-th time!
Hello for the 3-th time!
```

```
Hello for the 2-th time!
Hello for the 4-th time!
Hello for the 3-th time!
Hello for the 4-th time!
```

Alternativ måte å lage tråder på:

Vi kan også lage tråder ved å lage en subklasse av `Thread`, hvor vi lager `run` inne i trådobjektet:

```
class MyThread extends Thread {
    @Override
    public void run() {
        ...
    }
}
class Main {
    public static void main(String[] args) {
        Thread worker = new MyThread();
        worker.start();
    }
}
```

Selv om dette er mulig, skal vi (stort sett) holde oss til å skille mellom tråder og oppgaver (`Runnable`). Dette er fordi vi ønsker å kunne gjenbruke *samme* tråd på *forskjellige* oppgaver, eller samme oppgaver utført av flere tråder. Lager vi tråder på måten vist her ender vi opp med *spesiallist-tråder*, som bare kan gjøre én ting. Det er også mer *objektorientert* å skille mellom oppgaver og arbeidere.

Synkronisering

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

Eksempel: Problemet

```
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();
        worker2.start();
    }
}

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 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`.

Eksempelet: Problem løst

```
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 kun en og en tråd oppdatere telleren, så 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).

(Akkurat i dette eksempelet er det ikke mye som kan gå galt, og det ville trolig gått fint uten å bruke `try/finally`. Likevel lager vi oss gode vaner og gjør det *hver* gang vi bruker en lås. På den måten slipper vi uheldige feil senere.)

Monitors

En god strategi for å beskytte delt data er å lage en *monitor*. En monitor er et objekt som innkapsler den delte dataen, og definerer synkroniserte metoder for å jobbe med dataen. Vi skal i INF1010 *alltid* benytte monitorer, da dette er en god, objektorientert måte å gjøre synkronisering på.

Eksempel: Problem løst, med monitor

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

public class Main {
    public static void main(String[] args) {
        CountMonitor monitor = new CountMonitor();
        Runnable task = new MyTask(monitor);
        Thread worker1 = new Thread(task);
        Thread worker2 = new Thread(task);
        worker1.start();
        worker2.start();
    }
```

```
}
class CountMonitor {
    private final Lock lock = new ReentrantLock();
    private int sharedCounter = 0; // The protected data.

    public void increment() {
        lock.lock();
        try {
            sharedCounter = sharedCounter + 1;
        } finally {
            lock.unlock();
        }
    }
    public int getCounter() { return sharedCounter; }
}
class MyTask implements Runnable {
    private final int MAX_COUNT = 10000;
    private final CountMonitor monitor;

    public MyTask(CountMonitor monitor) { this.monitor = monitor; }

    public void run() {
        for (int i = 0; i < MAX_COUNT; i++) {
            monitor.increment();
        }
        System.out.println("Done! Shared counter = " + monitor.getCounter());
    }
}
```

Her ender vi opp med samme output. Gleden med å bruke en monitor vil bli større når problemene blir mer komplekse.

Conditions

Mange ganger må tråder vente på at en betingelse er oppfylt før den kan gjøre noe fornuftig. Dette kan vi oppnå ved å bruke`java.util.concurrent.locks.Condition`. En `Condition` kan vi tenke på som en ventekø tilknyttet en monitor. Vi kan plassere tråder i denne køen, i påvente av at en betingelse skal bli sann. Når dette skjer kan vi *signalisere* tråden, slik at den kan fortsette.

`Condition` er et grensesnitt som for oss har tre relevante metoder:

```
interface Condition {
    void await();
    void signal();
    void signalAll();
}
```

Eksempel: Brusautomat

Vi tenker oss en brusautomat (vår monitor) som inneholder et gitt antall brusbokser. Videre har vi kunder (tråder) som forsøker å ta brusbokser ut av automaten. Vi har også én person (tråd) som er ansvarlig for å fylle opp automaten når den er tom.

Live-koding!