Tråder 2

▼ Viktige begreper og metoder

Begreper

- **Monitor**: objekt som håndterer felles data (felles verdier, og med locks). Monitor er ikke et ord i Java, men betyr et objekt som er delt mellom flere tråder
- **Condition**: er et objekt, og som gitt en betingelse (typisk i en while-loop) ber trådene om å vente eller fortsette
- **Barriere**: et samlingspunkt i koden, der alle stopper og venter på at alle andre også har kommet frem.
- **Vranglås**: flere tråder som venter på hverandre i en syklus

Metoder

Condition:

- `.wait()`: Stopper midlertidig, gjør at tråden må vente på en betingelse er oppfyllt fram til tråden får signal til å fortsette. Bruk .wait() alltid i en while-loop
- `.signal()`: Signaliserer til en tråd som venter på denne Contition om å fortsette (tråden som har ventet lengst)
- `.signalAll()`: Signaliserer til alle trådene som venter på denne Contition om å fortsette

---

Monitor

→ et objekt som håndterer felles data (felles verdier, og med locks). Monitor er ikke et ord i Java, men betyr et objekt som er delt mellom flere tråder

En monitor er et objekt som innkapsler den delte dataen, og definerer synkroniserte metoder for å jobbe med dataen. En god strategi for å beskytte delt data er å lage en

monitor. . Vi skal i IN1010 *alltid* benytte monitorer, da dette er en god, objektorientert måte å gjøre synkronisering på.

```java
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());
    }
}
```

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.

Condtion brukes til på be tråder vente på å stoppe eller starte opp

Condition er et grensesnitt som for oss har tre relevante metoder: `await()`, `signal()`, `signalAll()`

Java-biblioteket java.concurrent

Interface Lock:

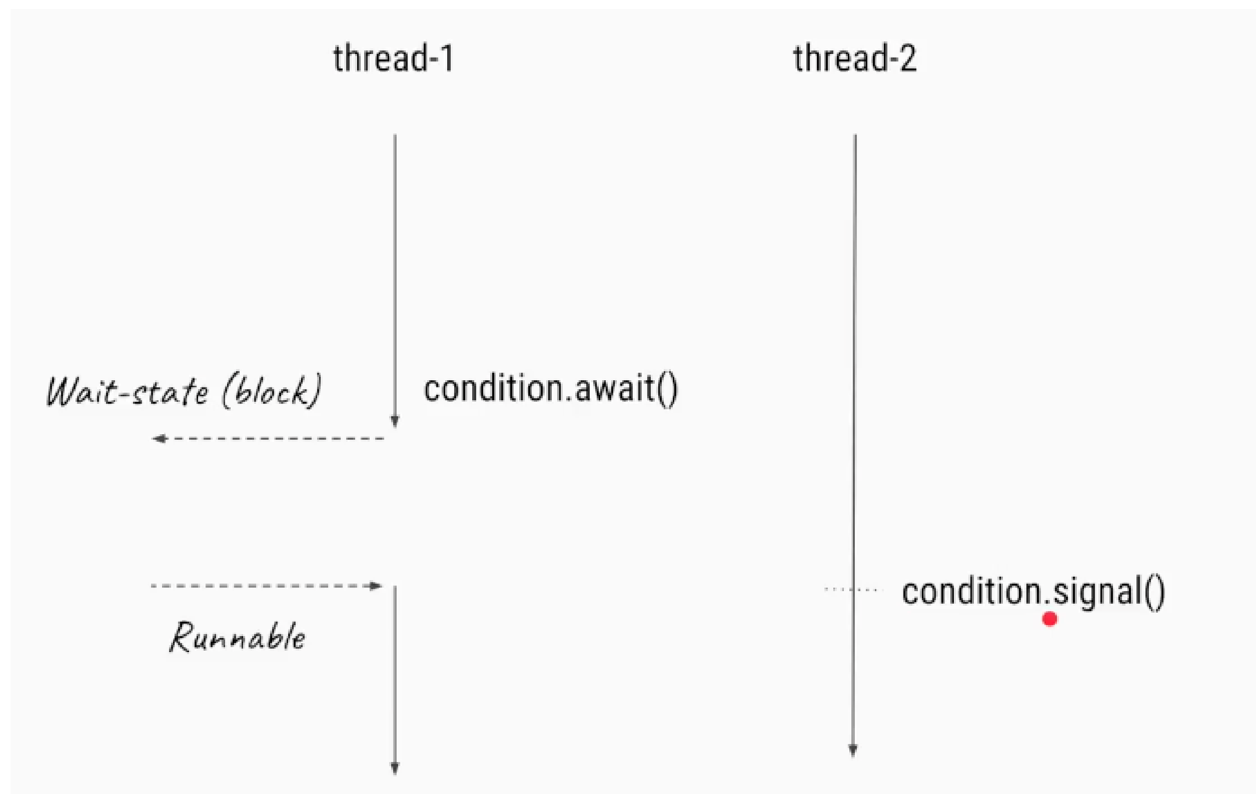
import java.concurrent.locks.*

- void **lock**() Acquires the lock
- void **unlock**() Releases the lock.
- Condition **newCondition**() Returns a new **Condition** instance that is bound to this Lock instance.

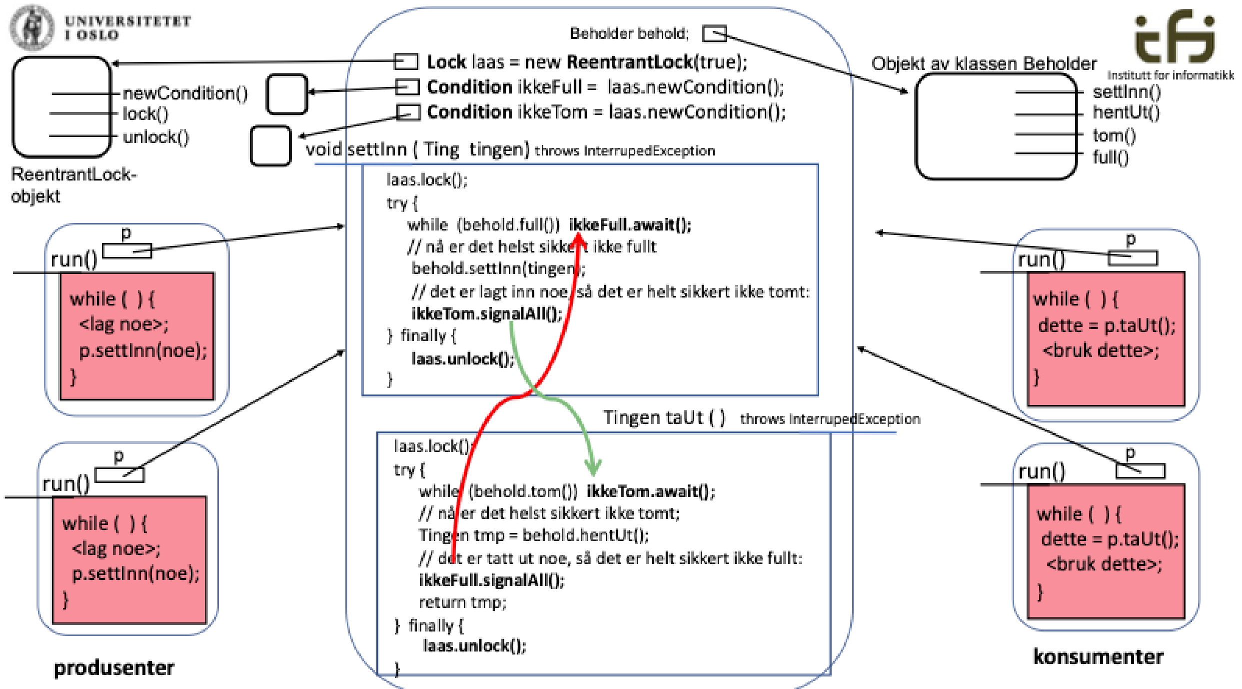
Interface Condition:

- void **await**() Causes the current thread to wait on this Condition until it is signalled (or interrupted)
- void **signal**() Wakes up one waiting thread (on this Condition).
- void **signalAll**() Wakes up all the waiting threads (on this Condition)

- class ReentrantLock implements Lock




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

    void signalAll();
}
```



Eksempel produsent/konsumenter (leggTil/taUt)

```java
private Lock lock = new ReentrantLock();
private Condition added = lock.newCondition();
private Condition removed = lock.newCondition();
```

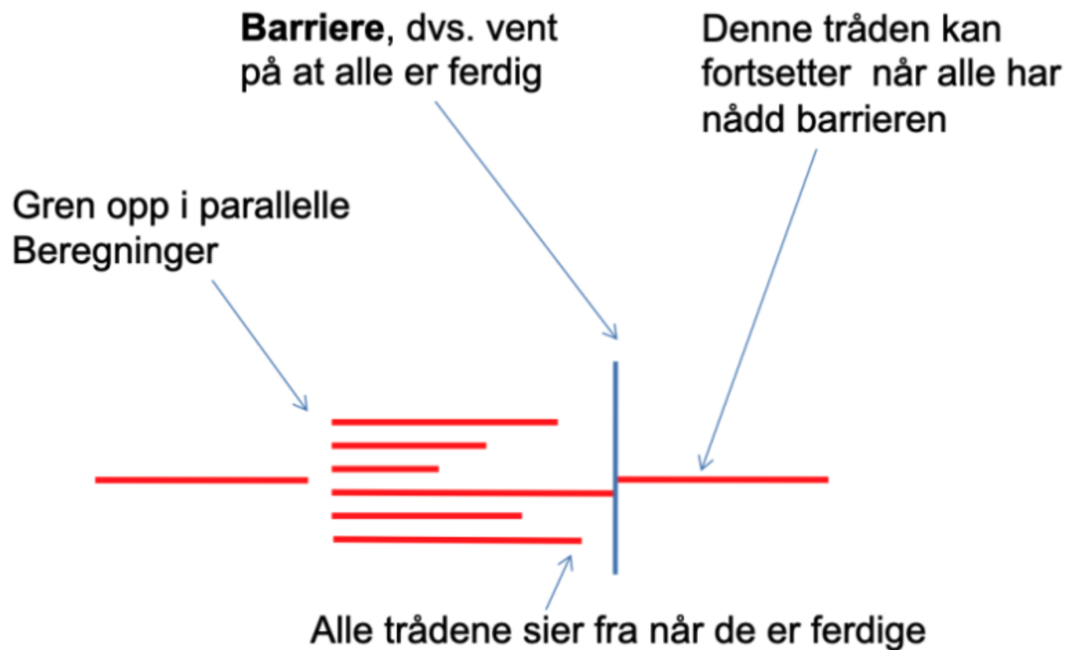
```java
public void produce() throws Interrup
    lock.lock();
    try {
        while (count == MAX_COUNT)
            removed.await();

        addData();
        added.signal();

    } finally {
        lock.unlock();
    }
}
```

```java
public String consume() throws Inte
    lock.lock();
    try {
        while (count == 0)
            added.await();

        String data = getData();
        removed.signal();

        return data;
    } finally {
        lock.unlock();
    }
}
```

Barrierer

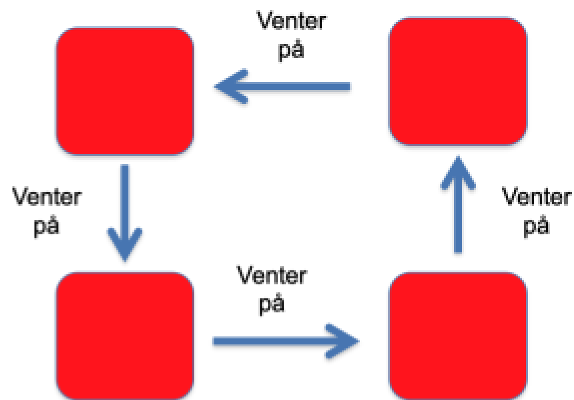
En barriere tenkes på som et samlingspunkt i koden, der alle stopper og venter på at alle andre også har kommet frem. Deretter kan vi la alle trådene fortsette, *samtidig*.

- `CountDownLatch(int count)` - Alle slipper gjennom barrierer når `.countDown()` har blitt kalt count ganger. Vi kan *velge* å vente ved barrieren med .`await()`
- `CyclicBarrier(int count)` - Når count tråder har kalt på `.await()`, slipper alle gjennom. Barrieren kan så *brukes på nytt* på samme måte.



Vranglås (deadlock)

→ Flere tråder som venter på hverandre



Vranglås kan oppstå når flere tråder kjemper om felles ressurser

- En eller flere felles ressurser ønskes av mer enn en tråd
- Hvis en tråd først tar en ressurs og deretter en annen . . .

Unngå vranglås

1. Ta bare én ressurs
2. Ta alle på en gang, eller ingen
3. Alle tråder tar alle ressurser i samme rekkefølge
   • Hvis vranglås har oppstått:
   – Frigi én og én ressurs til det ikke lenger er vranglås