KAPITTEL 18

Tråder

Når et program kjøres, utføres programsetningene i en bestemt rekkefølge. En tråd er det samme som denne programflyten, og i dette kapitlet skal vi se på hvordan vi starter og stopper en tråd, og hvordan vi kan ha flere tråder som kjører samtidig.

La oss begynne med en repetisjon av hva som skjer når du kjører Java-programmet nedenfor.

```
class Hei {
  public static void main(String[] args) {
    System.out.println("Hei  1");
    System.out.println("Hei  2");
  }
}
```

Etter kompilering starter du programmet ved å skrive `java Hei` på kommandolinja. Bak kulissene starter kjøresystemet til Java opp en tråd (eller en programflyt, om du vil) som kalles main-tråden. Main-tråden hopper til starten av main-metoden, som den forventer å finne i klassen Hei. Der følger den instruksjonene, en etter en i den rekkefølge de står. Først får vi skrevet ut `Hei 1` og så `Hei 2`. Eksekveringen av tråden fortsetter så fra main-metoden hvis den inneholder kall på metoder, vi gjentar visse instruksjoner dersom de er inne i løkker, osv. Hele tiden holder programmet på med å utføre en og bare en instruksjon når programmet bare har en tråd. At det er en slik tråd, kan du oppdage hvis du prøver å starte en klasse uten main-metoden – eller skriver feil klassenavn. Skriver vi for eksempel `java hei` (liten «h»), får vi følgende feilmelding før programmet avslutter:

```
Exception in thread "main"
   java.lang.NoClassDefFoundError:  hei
```

Som du ser, har vi altså en tråd (eng. thread), som heter main. Unntaket blir kastet i denne tråden (se kapittel 17).

18.1 Stoppe og starte en tråd

Det første vi nå skal se på, er hvordan vi kan få main-tråden til å ta en pause i eksekveringen. Vi benytter den statiske metoden `Thread.sleep(long millisek)`. Metoden kan kaste et unntak, InterruptedException, som vi må fange opp dersom det skjer en feil. Programmet nedenfor skriver først ut en linje med teksten «skal sove... ») og venter i minst to sekunder[11] før neste linje, som er «Nå er jeg våken», skrives ut.

```
class SoveProgram {

  public static void main(String[] args) {
    long sek = 2 * 1000; // to sekunder
    System.out.println("Skal  sove...");
    try {

      Thread.sleep(sek);

    } catch (InterruptedException  e) {return;}
    System.out.println("Nå  er jeg våken");
  }
}
```

18.2 Lage en ny tråd

La oss forsøke å lage en stoppeklokke. For hvert sekund som er passert, skal programmet skrive ut antall sekunder. Utskriften hvert sekund kan gjøres i følgende while-løkke:

[11] Det kan ta litt lengre tid hvis prosessoren er opptatt med noe annet enn vårt program i det øyeblikket vår prosess har sovet i to sekunder. Som oftest vil forsinkelsen være så liten at vi kan se bort fra den.

```
boolean stop = false;
int tid = 0;

while (!stop) {
  System.out.println(tid++);
  try {
    Thread.sleep(1 * 1000); // ett sekund
  } catch (InterruptedException e) {return;}
}
```

Problemet ligger i hvordan vi får stoppet og startet klokken. Løsningen er å la while-løkka kjøre i en egen tråd, mens vi i main-tråden styrer start og stopp av klokken.

Vi begynner med å lage klassen og metoden while-løkka skal ligge i. Vi lar klassen være en subklasse av Thread-klassen. Thread-klassen inneholder en metode som automatisk kalles når man starter en ny tråd, nemlig run-metoden. I Thread-klassen gjør ikke run-metoden noe som helst, men vi redefinerer den (det er en virtuell metode) og får dermed følgende (foreløpige) klasse:

```
class Stoppeklokke extends Thread {
  private volatile boolean stop = false;

  public void run() {
    int tid = 0;
    while (!stop) {
      System.out.println(tid++);
      try {
        Thread.sleep(1 * 1000); // ett sekund
      } catch (InterruptedException e) {return;}
    }
  }
}
```

Vi skal ikke gjøre kallet på `run()` direkte, i stedet skal vi kalle på en metode som heter `public void start()`, arvet fra Thread-klassen. Ved kall på `start()` skjer to ting: Systemet lager en ny tråd som starter opp run-metoden vår, og den gamle tråden, som utførte kallet på `start`, fortsetter umiddelbart etter kallstedet.

Til slutt i Stoppeklokke-klassen vår trenger vi en metode for å stoppe

klokken. Ideelt sett burde metoden hete public void stop(), men det er det en utdatert metode i Thread-klassen som heter allerede, så vi må finne på et annet navn – for eksempel kan vi kalle den `public void end()`. Denne metoden setter variabelen `stop` til verdien `true`, noe som vil føre til at while-løkka i run-metoden avsluttes. Siden run-metoden dermed er ferdig, dør stoppeklokketråden. Denne metoden kan vi dermed kalle fra main-tråden for å stoppe stoppeklokken.

Slik ser det ferdige programmet ut:

```
import easyIO.*;

class Klokke {
  public static void main(String[] args) {

    In stdin = new In();
    System.out.println("Stopp  og start med [ENTER]");
    stdin.inLine();
    // Her lages stoppeklokke-objektet:
    Stoppeklokke stoppeklokke = new Stoppeklokke();
    // og her settes den nye tråden i gang.
    stoppeklokke.start();

    stdin.inLine();
    stoppeklokke.end();
  }
}


class Stoppeklokke extends Thread {
  boolean stop = false;

  // blir kalt opp av superklassens start-metode.
  public void run() {
    int tid = 0;
    while (!stop) {
      System.out.println(tid++);
      try {
        Thread.sleep(1 * 1000); // ett sekund
      } catch (InterruptedException e) {return;}
    }
  }
```

```
  public void end() {
    stop = true;
  }
}
```

En alternativ måte å lage nye tråder på er å la en klasse implementere interfacet Runnable, som inneholder en abstrakt metode kalt `run`. En ny tråd lages da ved å benytte en konstruktør i Thread-klassen som tar et Runnable-objekt som parameter, og den startes ved å kalle på Thread-objektets start-metode. Vi kunne altså ha laget programmet over som følger:

```
import easyIO.*;

class AlternativKlokke {
  public static void main(String[] args) {
    In stdin = new In();
    System.out.println("Start  og stopp med [ENTER]");
    stdin.inLine();

    // Lag stoppeklokke-objektet:
    AlternativStoppeklokke stoppeklokke =
       new AlternativStoppeklokke();

    // Lag tråden som skal kjøre det nye objektet:
    Thread stoppeklokketråd = new Thread(stoppeklokke);

    // Her settes den nye tråden i gang:
    stoppeklokketråd.start();

    stdin.inLine();
    stoppeklokke.end();
  }
}

class AlternativStoppeklokke implements Runnable {
  boolean stop = false;

  // Blir kalt opp av superklassens start-metode:
  public void run() {
    int tid = 0;
```

```
    while (!stop) {
      System.out.println(tid++);
      try {
        Thread.sleep(1 * 1000); // ett sekund
      } catch (InterruptedException e) {return;}
    }
  }

  public void end() {
    stop = true;
  }
}
```

En kanskje mer elegant metode er å la den nye tråden starte og stoppe seg selv. I klassen som implementerer Runnable legger vi til to metoder; `start()` og `stop()`:

```
class AltStoppeklokke2 implements Runnable {
  Thread klokketråd = null;
  public void start() {
    if (klokketråd = null) {
      klokketråd = new Thread(this);
    }
    klokketråd.start();
  }

  public void stop() {
    klokketråd = null;
  }

  public void run() {
    Thread denneTråden = Thread.currentThread();
    while (denneTråden == klokketråd) {
      ... som før
    }
  }
}
```

For å starte og stoppe stoppeklokken over kan vi ganske enkelt kalle på de to

metodene start() og stop(). I start-metoden lages den nye tråden, hvis start-metode (fra Thread) kalles.

Metoden `Thread.currentThread()` returnerer en peker til Thread-objektet som kalte på metoden – det vil si denne tråden. Vi benytter dette i run-metoden, i betingelsen til while-løkka:

```
Thread denneTråden = Thread.currentThread();
while (denneTråden == klokketråd) {
  ...
}
```

Siden de to pekerne i utgangspunktet peker på det samme Thread-objektet når run-metoden starter, utføres innholdet i while-løkka.

```
public void stop() {
  klokketråd = null;
}
```

Når denne metoden blir kalt (gjerne fra en annen tråd), settes klokketråden til å peke på `null`. Effekten er at betingelsen i run-metodens while-løkke blir usann, og run-metoden avsluttes.

Det er altså to metoder for å lage en ny tråd: enten ved å subklasse Thread, skrive over run-metoden og kalle på start-metoden:

```
class Traad1 extends Thread {
  public void run() {
    ...
  }
}

class Starter {
  new Traad1().start();
}
```

eller å implementere Runnable-interfacet. I så fall benyttes en annen konstruktør i Thread-klassen til å få laget selve tråd-objektet:

```
class Traad2 implements Runnable {
  public void run() {
    ...
  }
}

class Starter {
  new Thread(new Traad2()).start();
}
```

Hvorvidt du velger den ene eller den andre løsningen, blir opp til deg selv.

18.3 Samarbeid mellom tråder

Alle tråder som er satt i gang av den samme virtuelle Java-maskinen, har felles adresserom. Det betyr at to tråder kan ha tilgang til de samme metoder og variabler – noe vi kan se av stoppeklokkeprogrammet. I run-metoden, som kjøres av stoppeklokketråden, sjekkes sannhetsverdien av variabelen `stop`, og main-tråden endrer variabelen til `false` ved hjelp av end-metoden.

Noen ganger har vi behov for å synkronisere to eller flere tråder. Det vil si at vi vil sikre oss at en tråd er ferdig med en oppgave før neste tråd begynner. Vi skal nå se på hvordan dette gjøres.

Vi tenker oss følgende program: En kokk som legger mat på en tallerken og setter den fram på disken, lager en ny tallerken og setter på disken, osv., og en servitør som tar en tallerken og serverer maten, henter en ny tallerken og serverer, osv. De to jobber i litt ulikt tempo. Kokken kan både være raskere, men også tregere til å sette fram maten enn servitøren er til å hente den. Dette er en variant av et klassisk trådproblem, kalt produsent-konsument-problemet. Kokken vår er produsenten, og servitøren er konsumenten. Produsent og konsument kjøres i hver sin tråd. I tillegg har restauranten en egen tråd.

Vi må hindre at to eller flere tråder samtidig skriver variabler de alle har adgang til. Hvis vi foran en metode skriver Java-ordet `synchronized`, vil alle slike synkroniserte metoder i den klassen være sperret, og alle tråder som seinere prøver å bruke en slik metode, må vente inntil den tråden som holder på med en av de synkroniserte metodene er ferdig.

For ikke å gjøre programmet altfor omfattende jukser vi litt og lar tallerkenene være representert ved to tellere i Restaurant-klassen, en for antall ferdige tallerkener fra kokken, og en teller for antall serverte tallerkener. I tillegg til de to tallerkentellerne inneholder klassen fire synkroniserte metoder:

– en for å ta imot en tallerken fra kokken (øker `antLaget`-telleren med en)
– en for å servere en tallerken (øker `antServert` med en)
– en for å teste om servitøren er ferdig
– en for å teste om kokken er ferdig

Det er nå to mulige feilsituasjoner: at servitøren prøver å servere en tallerken som ennå ikke er laget, og at kokken lager flere tallerkener enn det er behov for (vi tillater høyst tre tallerkener som ennå ikke er servert). Restaurant-klassen blir som følger:

```
class Restaurant {
  int antBestilt ;
  int antLaget= 0, antServert = 0; // tallerkenretter

  Restaurant(int ant) {
    antBestilt = ant;
  }

  public static void main(String[] args) {
    Restaurant rest;
    rest = new Restaurant(Integer.parseInt(args[0]));
    Kokk kokk = new Kokk(rest);
    kokk.start();
    Servitor servitør = new Servitor(rest);
    servitør.start();
  }

  synchronized boolean kokkFerdig () {
    return antLaget == antBestilt;
  }

  synchronized boolean servitørFerdig () {
    return antServert == antBestilt;
  }

  synchronized void putTallerken() {
    // Kokketråden blir eier av låsen.

    while (antLaget - antServert > 2) {
      /* så lenge det er minst 2 tallerkener
```

```
        * som ikke er servert, skal kokken vente. */
      try {
        wait(); /* Kokketråden gir fra seg
                 * låsen og sover til den
                 * blir vekket */

    } catch (InterruptedException  e) {return;}
    // Kokketråden blir igjen eier av låsen.
  }

    antLaget++;
    System.out.println("Kokken  laget nr " + antLaget);

    notify(); /* Si ifra til servitøren. */
  }

  synchronized void getTallerken() {
    // Servitørtråden blir eier av låsen.

    while (antLaget == antServert) {
      /* så lenge kokken ikke har plassert
       * en ny tallerken. Dermed skal
       * servitøren vente. */
      try {  wait();       /* Servitørtråden gir fra seg
                            * låsen og sover til den
                            * blir vekket
                            */
      } catch (InterruptedException  e) {return;}
      // Servitørtråden blir igjen eier av låsen.
    }
    antServert++;
    System.out.println("Servitør  serverer nr:" +
                       antServert);
    notify(); /* si ifra til kokken. */
  }
}
```

Kokken og servitøren skal henholdsvis produsere og konsumere det antall tallerkener vi oppgir på kommandolinja når vi starter programmet. Når kokken har produsert tallerken nr. 1, kan servitøren konsumere den. Så, når kokken

har produsert neste tallerken, kan servitøren konsumere tallerken nr. 2, osv. Vi lar også gjerne kokken lage ferdig inntil 3 tallerkener før vi stopper kokken. De to klassene ser dermed slik ut:

```
class Kokk extends Thread {
  Restaurant rest;
  Kokk(Restaurant rest) {
    this.rest = rest;
  }

  public void run() {
    while (!rest.kokkFerdig()) {
      rest.putTallerken();   // lever tallerken
      try { sleep((long) (1000 * Math.random()));
      } catch (InterruptedException e) {return;}
    }
    // Kokken er ferdig
  }
}

class Servitor extends Thread {
  Restaurant rest;
  Servitor(Restaurant rest) {
    this.rest = rest;
  }

  public void run() {
    while (!rest.servitørFerdig())   {
      rest.getTallerken();  /* hent tallerken og
                             * server */
      try { sleep((long) (1000 * Math.random()));
      } catch (InterruptedException e) {return;}
    }
    // Servitøren er ferdig
  }
}
```

Som du ser, lar vi kokken og servitøren vente mellom 0 og 1 sekund, tilfeldig valgt, før neste runde i while-løkka. En kjøring av programmet kan for eksempel resultere i følgende:

```
>java Restaurant 6
Kokken laget nr 1
Servitør serverer nr 1
Kokken laget nr 2
Kokken laget nr 3
Servitør serverer nr 2
Kokken laget nr 4
Kokken laget nr 5
Servitør serverer nr 3
Kokken laget nr 6
Servitør serverer nr 4
Servitør serverer nr 5
Servitør serverer nr 6
```

Fordi vi trekker tilfeldige ventetider, har en annen kjøring med samme antall tallerkener gitt:

```
>java Restaurant 6
Kokken laget nr 1
Servitør serverer nr 1
Kokken laget nr 2
Servitør serverer nr 2
Kokken laget nr 3
Servitør serverer nr 3
Kokken laget nr 4
Kokken laget nr 5
Servitør serverer nr 4
Kokken laget nr 6
Servitør serverer nr 5
Servitør serverer nr 6
```

En advarsel til slutt. Programmering med tråder er ikke lett. Å lage slike programmer tilhører vanligvis hovedfagspensum. Hovedproblemet er at vi ikke kan stole på at setningene blir utført en etter en, slik de blir det i et vanlig program; en del av programmet vårt kan nemlig stoppes tilfeldig ett eller annet sted, og en annen del kan få utført noen av sine instruksjoner. En viktig grunn til å nevne dette i en begynnerbok i Java-programmering er at mange av metodene i Java-biblioteket starter tråder uten at det går klart fram av dokumentasjonen. Hvis man for eksempel programmerer grafiske

grensesnitt med Swing, vil man få startet (minst) en ekstra tråd. Metoden paint() vil alltid kjøre på denne nye tråden. Hvis man vil unngå noen av problemene fra grafisk brukergrensesnitt-programmering, vil følgende regel være nyttig: Skriv ikke på (oppdater ikke) noen av variablene i programmet inne i paint(), men du kan godt lese alle typer av variabler i paint().

18.4 Oppsummering

- Et program består av en eller flere tråder (programflyter), og alle tråder i et program har felles adresserom (ser de samme klassene med deres variabler).
- Metoder som kan bli nyttet av mer enn en tråd, må få Java-ordet `synchronized` foran deklarasjonen. Da sperres andre tråder fra å bruke noen av de metodene i klassen som er synkronisert (de må vente), hvis en slik metode allerede utføres av en av trådene.
- Et vanlig program har bare én tråd, men ved enten å lage en subklasse av Thread, implementere grensesnittet `Runnable` eller kalle noen av klassene i Java-biblioteket (og det er ikke dokumentert hvilke), kan vi få to eller flere tråder i programmet.
- En tråd som utføres, også vanlige programmer med bare en tråd, kan vente et visst antall millisekunder ved å kalle på `Thread.sleep(millisek)`.
- Et brukergrensesnitt starter en egen tråd, og paint-metoden kjører på denne tråden.
- Program med flere tråder som ikke er riktig programmert, stopper enten helt opp (henger) eller gir gale resultater, eller begge deler.
- Det er vanskelig å programmere med tråder, og man må ha gode grunner til å gjøre det.

18.5 Oppgaver

Oppgave 1

Utvid stoppeklokkeeksemplet til å kjøre to klokker samtidig i hver sin tråd. En tråd skriver ut en tekst hvert minutt, og en annen tråd skriver ut et tall hvert sekund.

Oppgave 2

Utvid Restaurant-programmet til å ha to objekter av klassen Kokk. Dette gjør du ved å kopiere fra de to kodelinjene i `main` som lager en kokk. Kjør dette programmet flere ganger og se om du kan observere feil oppførsel. (Hint: Prøv å kjøre programmet flere ganger med at du bare skal lage 1 tallerken.) Greier du a) å forklare de feilene du kan observere, og b) å rette feilene?

Oppgave 3

Vanskelig: Utvid Restaurant-programmet slik at du samtidig har 5 kokker og
7 servitører (hver med et nummer som kommer med i utskriften). Hint: Du må bl.a. innføre flere tester ved kall på de metodene som allerede er definert for å få dette programmet riktig.