Föreläsning 11 – Aktiva objekt och trådar, strömmar, kommunikation

DAT043, 2017-02-20

Aktiva objekt och trådar

Multitasking, parallella program

- Vanliga datorer har kunna köra flera program skenbart samtidigt sedan 80-talet, då grafiska gränssnitt växte fram.
- Detta kallas *multitasking*.
- Detta görs genom att processorn snabbt växlar fram och tillbaka mellan att exekvera olika *processer*.
- Nuförtiden finns flerkärniga processorer och då kan flera program faktiskt exekveras samtidigt.
- Eftersom olika program arbetar med varsitt avskilt minnesutrymme så tar det en del tid att växla mellan processer. Processer hanteras på operativsystemnivå.
- Man kan också skriva *parallella* (*concurrent*) program som exekverar olika delar samtidigt. Då använder man *trådar*, en lättviktig variant av processer som kan användas inom ett program. De olika trådarna delar på samma heap-utrymme.
- En anledning att använda trådar kan vara att få programmet att gå fortare på flerkärniga processorer.
- En annan anledning är att det är lämpligt att organisera programmet så att olika trådar sköter olika uppgifter.
- Trådar hanteras av runtime-systemet.

Aktiva objekt

- Objekt som kan utföra något utan att bli anropad kallas för *aktiva objekt*, till skillnad från *passiva objekt*.
- Sådana objekt har en tråd (eller flera).

Trådar i Java

- Klassen `Thread` representerar en exekveringstråd i Java.
- När man anropar en tråds instansmetod `start` så startas tråden och körs parallellt med tråden som gjorde anropet.
- Den metod som körs är metoden `run`. Man kan skapa en klass som ärver `Thread` och överskugga `run` i denna.
- Vanligare är att men implementerar interfacet `Runnable` och metoden `run` i denna. Man skapar då ett trådobjekt genom att ange den implementerande klassen som argument till konstrueraren.
- När ett program startas så körs main-metoden i en huvudtråd som skapas automatiskt. Operationer som är relaterade till trådar kan utföras för huvudtråden genom anrop till klassmetoderna i `Thread`.
- Ett Java-program avslutas när alla trådar avslutats, inte när huvudtråden avslutats.
- Trådar kan ges olika prioritet med metoden `setPriority`.

Stoppa trådar, pausa, vänta på trådar

- Man kan begära att trådar ska stoppas (innan slutet av `run`-metoden nås). Det gör man med metoden `interrupt`.
- Det är dock upp till tråden själv att avsluta när det är lämpligt. Tråden kan avgöra om den begärts avslutad genom klassmetoden `interrupted`.
- Exekveringen av en tråd pausas med klassmetoden `sleep`.
- Denna metod kastar `InterruptedException` om tråden begärs avbruten.
- Man kan invänta att en tråd avslutas med instansmetoden `join`.

Exempel, trådar

Synkronisering

- Eftersom olika trådar tillhör samma program och delar på heap-minnet så kan de komma samma objekt.
- När objekt ändras så kan de tillfälligt befinna sig i ett tillstånd som inte är giltigt.
- Om flera trådar ändrar på samma objekt samtidigt eller en tråd ändrar på ett objekt samtidigt som andra läser från det så kan det bli fel.
- För att undvika detta finns en mekanism för synkronisering.
- Om en klass ska användas i parallella program så kan man ange att vissa metoder ska exekveras färdigt innan någon annan tråd får anropa en metod för objektet. På så sätt hinner alla steg för att åter skapa ett giltigt tillstånd utföras innan något annat sker.
- Sådana metoder ges modifieraren `synchronized`.
- Klasser och kod som är säkra att använda med parallella trådar kallas trådsäkra (thread safe).

Synkronisering

- Om en tråd anropar en synkroniserad metod och en annan tråd anropar någon (kan vara en annan) synkroniserad metod för samma objekt så kommer denna exekvering inte start förrän den första är klar.
- I en klass som ska vara trådsäker bör ofta alla metoder som läser och ändrar variabler i objektet vara synkroniserade. Konstrueraren behöver inte vara det. Under skapandet är det bara en tråd som har en referens till objektet.
- Klassmetoder kan också vara synkroniserade. För dessa blockeras exekveringen av andra synkroniserade klassmetoder.
- En sats eller block av satser kan också vara synkroniserad. Man anger då vilket objekt det gäller.

```
synchronized (obj) {
   ...
}
```

Kommunikation mellan trådar

- Ofta måste trådar i parallella program kommunicera med varandra.
- Detta kan de göra genom att ha ett gemensamt trådsäkert objekt som de läser och skriver till.
- Ofta kommunicerar trådar via köer. En tråd skickar information i form av en sekvens av objekt som en annan tråd tar emot i samma ordning som de skickades.
- I Javas API finns gränssnittet `BlockingQueue` tillsammans med implementerande klasser för detta.
- Dessa köer är trådsäkra och har funktionalitet för att låta en tråd vänta på att något ska läggas i kön ifall den är tom.

Trådar i Swing

- AWT och Swing är inte implementerat på ett trådsäkert sätt. Därför sker allt som har med gränssnittet i en speciell tråd som skapas automatiskt. Denna kallas event dispatching thread.
- All kod som interagerar med det grafiska gränssnittet ska utföras i lyssnarmetoder. Lyssnarmetoder ska man heller aldrig själv anropa.
- Undantag är metoder som orsakar något indirekt, såsom `repaint`.
- Beräkningar som tar tid bör inte utföras i event dispatching thread, utan i en separat tråd.
- Klassen `SwingWorker<T,V>` kan användas för detta.
- Man överskuggar `doInBackground` som körs i en separat tråd när `execute` anropas. Resultatet av beräkningen som denna metod returnerar är av typen `T`.

Trådar i Swing

- När `doInBackground` är klar så anropas automatiskt `done` i event dispatching thread. Denna metod överskuggar man med koden som uppdaterar GUI:n enligt beräkningens resultat. I `done` kan man anropa get för att få resultatet av beräkningen, d.v.s det som `doInBackground` returnerade.
- Man kan begära att beräkningen i en `SwingWorker` avbryts med `cancel`. I `doInBackground` kontrollerar man om detta begärts med `isCancelled`.
- I `doInBackground` kan man skicka delresultat av typen `V` med metoden `publish`. Detta anropar automatiskt `process` som man kan överskugga och som körs i event dispatching thread.
- I `doInBackground` kan man också ange hur långt i procent beräkningen kommit med `setProgress`. Detta anropar lyssnare som lagts till med `addPropertyChangeListener`.

Exempel på beräkningstråd i Swing

Lägg till en textruta och en label till något GUI-exempel (t.ex. MVC-exempel från förel. 7):

```
JTextField tf = new JTextField(10);
frame.add(tf);

JLabel lb = new JLabel();
frame.add(lb);

tf.addActionListener(e -> {
  MyWorker worker = new MyWorker(lb, Long.parseLong(tf.getText()));
  worker.execute();
  lb.setText("wait...");
});
```

I MVC-exempel kan koden läggas in i metoden `run` före raden `frame.pack();`

Definiera `MyWorker` som avgör om ett tal är primtal (t.ex. inuti huvudklassen i MVC-exemplet):

```
public static class MyWorker extends SwingWorker<Boolean, Object> {
  final long number;
  final JLabel resultLabel;
  public MyWorker(JLabel resultLabel, long number) {
    this.number = number;
    this.resultLabel = resultLabel;
  }
  @Override
  protected Boolean doInBackground() throws Exception {
    for (long i = 2; i < number; i++) {
      if (number % i == 0) return false;
    }
    return true;
  }
  @Override
  protected void done() {
    Boolean isPrime = false;
    try {
      isPrime = get();
    } catch (Exception e) {
    }
    resultLabel.setText(isPrime ? "is prime" : "is not prime");
  }
}
```

Prova programmet med ett stort primtal, t.ex. 982451653. Medan beräkningen sker bör du kunna använda de övriga komponenterna och se att programmet är responsivt under tiden.

Strömmar

Strömmar

- Vi har tidigare använt `Scanner` och `PrintWriter`. Dessa klasser kan läsa och skriva tal i en ström av tecken.
- Strömmar är en generellt begrepp för in- och utdata från perifera resurser, t. ex. filer.
- De två superklasserna `InputStream` och `OutputStream` representerar in- och utströmmar.
- Med dessa kan man läsa och skriva rå data, d.v.s. sekvenser av bytes.
- `FileInputStream` och `FileOutputStream` är subklasser som läser och skriver filer.
- `InputStreamReader` och `OutputStreamWriter` är superklasser för in- och utströmmar av tecken (characters).
- `FileReader` och `FileWriter` är subklasser till dessa för filer.

Strömmar

- Ofta är det mer effektivt att läsa och skriva hela sjok data istället för ett tecken i taget. För att automatiskt buffra läsning och skrivning kan man använda `BufferedInputStream`, `BufferedOutputStream`, `BuffererReader` och `BufferedWriter`.
- `BufferedReader` innehåller metoden `readLine` som ofta kommer till nytta.
- Den fil som hör till en filström öppnas när objektet skapas och stängs när man anropar `close`.

Filegenskaper

- Klassen `File` kan förutom att representera referenser till filer vars innehåll man vill läsa eller skriva också användas för att ta reda på och ändra egenskaper om en fil.
- Exempel är `exists`, `canWrite`, `canRead`, `setReadOnly`, `isFile`, `isDirectory`, `lastModified`, `length`

Nätverkskommunikation

URL

- Precis som `File` motsvarar en referens till en fil i filsystemet så har Javas API klassen `URL` som representerar en resurs på internet.
- Man kan skapa ett `URL`-objekt med en sträng, t.ex.
  `new URL("http://www.cse.chalmers.se/edu/course/DAT043/")`
- Med `url.openConnection()` skapar man en ett objekt av typen `URLConnection`.
- Med `urlconn.getInputStream()` får man en `InputStream`.
- Denna kan man sedan läsa från som vanligt från strömmar.
- På slutet stänger man strömmen.
- `url.openStream()` är en genväg till strömmen.
- Med en `URLConnection` kan man få meta-information med t.ex. `getContentLength`, `getContentType`, `getDate`.

Exempel, URL, URLConnection

```
URL url = new URL("http://www.cse.chalmers.se/edu/course/DAT043/");
URLConnection conn = url.openConnection();
BufferedReader in = new BufferedReader(new InputStreamReader(
                                        conn.getInputStream()));
String inputLine;
while (true) {
  inputLine = in.readLine();
  if (inputLine == null) break;
  System.out.println(inputLine);
}
in.close();
```

Portar och Sockets

- Vill man kommunicera på nätverk/internet på lägre nivå så kan man använda sockets.
- En socket motsvarar en förbindelse till en annan dator via en virtuell kanal, en s.k. port. (Port 80 brukar användas för HTTP-kommunikation)
- Man kan kommunicera antingen osynkroniserat eller synkroniserat. Detta motsvarar ungefär skicka sms och ringa ett samtal med telefon.
- Vid osynkroniserad kommunikation skickar man datagram. Man vet inte när eller i vilken ordning dessa kommer fram.
- Vid synkroniserad kommunikation upprättar man en förbindelse där en dator är server och en eller flera är klienter.

Datagram

- För att identifiera datorer används `InetAddress`. Man kan skapa ett objekt från en sträng med klassmetoden `getByName`.
- Ett paket med data (ett datagram) representeras av klassen `DatagramPacket`.
- Man kan skapa ett med konstrueraren `DatagramPacket(data, data.length, inetAddr, port)` där data är en byte-array och port är porten på mottagardatorn som ska användas.
- För att skicka och ta emot datagram skapar man en `DatagramSocket` och metoderna `send` och `receive`.

Klient/Server

- För att upprätta en stadig förbindelse mellan två datorer skapar man sockets.
- Servern skapar en `ServerSocket` med konstrueraren `ServerSocket(port)`.
- Servern anropar sedan `accept` som väntar tills en klient har anslutit sig och returnerar då en `Socket` som motsvarar anslutningen till denna klient.
- En klient skapar en `Socket` med konstrueraren `Socket(inetAddr, port)`.
- Från en socket kan man få in- och utströmmar med `getInputStream` och `getOutputStream`. Dessa strömmar används som vanligt.