Objektorienterad Programmering DAT043

Föreläsning 11
19/2 -18
Moa Johansson
(delvis baserat på Fredrik Lindblads material)

Multitasking och parallella program

- *Multitasking:* växla mellan olika processer.
  - Med flerkärniga processorer kan program exekveras samtidigt i moderna datorer.
  - Tar tid att växla.
  - Operativsystemet hanterar.

- *Parallella program* - trådar (“lättviktiga” processer).
  - Program exekverar olika delar samtidigt.
  - Kan gå snabbare på flerkärninga processorer.
  - Olika trådar sköter olika uppgifter, e.g. GUI, beräkningar.
  - Runtimesystemet hanterar.

•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.

Trådar i Java

- Klassen `Thread` representerar en exekveringstråd.
  - Programmeraren måste skriva koden som ska köras i tråden.
- Som subklass av `Thread` (överskugga metoden `run()`).
- Implementera interface `Runnable` - metoden `run()`. Detta är vanligare.

```
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
         // Skapa nytt Thread-objekt.
         // Som argument till konstruerare i Thread anges en Runnable.
        Thread t1 = new Thread(new HelloRunnable()));
        t1.start();
    }
}
```

•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 Thread-objekt kan konstrueras genom att man ger något av typ Runnable som argument (här ett objekt av vår klass HelloRunnable, som ju implementerar Runnable interfaces).
* Metoden start i Thread startar den nya tråden.
•Trådar kan ges olika prioritet med metoden setPriority.

Trådar i Java

- Klassen `Thread` representerar en exekveringstråd.
  - Programmeraren måste skriva koden som ska köras i tråden.
- Som subklass av `Thread` (överskugga metoden `run()`).
- Implementera interface `Runnable` - metoden `run()`.
  Detta är vanligare.

```
public static void main(String args[]) {
   ...
   // Skapa ett Thread objekt med lambda-uttryck för run-metoden.
   Thread t2 = new Thread(() ->
                          System.out.println("Hello from another thread!"));
   t2.start();
}
```

Kom ihåg: Ett interface som bara innehåller en abstrakt metod kallas funktions-interface. Runnable är ett sådant, eftersom det bara innehåller metoden run().
Vi kan alltså använda ett lambda-uttryck för att definiera en enkel run()-metod.

DEMO: HelloThread.java.

Notera att ordningen i vilken trådarna skriver ut sina meddelanden är olika från gång till gång när programmet körs!

Trådar: stoppa, pausa, vänta

- Be att stoppa en tråd innan run-metoden är klar: metoden `interrupt()`.
- Metoden `Thread.interrupted()`: tråden kan kontrollera om någon begärt att den avslutas.

```
t1.start();
...
t1.interrupt(); //Be t1 avsluta
```

```
// I trådens run-metod
if(Thread.interrupted()){
  //Kod för att avsluta
}
```

•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.
•

Trådar: stoppa, pausa, vänta

- En tråd kan pausas med metoden sleep():

```
try {
// Pausa i ett antal millisekunder.
    Thread.sleep(interval * 1000);
// Om interrupt medans tråden sover.
} catch (InterruptedException e) {
    ... }
```

- Invänta att en annan tråd ska bli klar: metoden `join()`.

```
t1.start();
...
t1.join(); //Vänta tills t1 blir klar
... // Fortsätt med programmet.
```

•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.
•

Synkronisering

- Trådar i samma program kan komma åt samma objekt.
- Om flera trådar ändrar på samma objekt samtidigt kan det bli fel.
- Detta kallas *thread interference.*

```
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }
    public void decrement() {
        c--;
    }
    public int value() {
        return c;
    }
}
```

1.Hämta värde på c
2. Öka/minska värde med 1.
3.Skriv nytt värde på c.

Anta att vi har ett objekt `Counter count`

Tråd A: `count.increment();`
Tråd B: `count.decrement();`

```
Tråd A : hämta c --> 0
Tråd A : beräkna 0++ --> 1
Tråd B : hämta c --> 0
Tråd A : skriv c = 1
Tråd B : beräkna 0-- --> -1
Tråd B : skriv c = -1 //FEL!
```

•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

- Metoder kan göras trådsäkra med nyckelordet `syncronized`.
- Om en tråd anropat en synkroniserad metod måste andra trådar vänta tills denna är klar.

```
class Counter {
    private int c = 0;

    public void syncronized increment() {
        c++;
    }
    public void syncronized decrement() {
        c--;
    }
    public int syncronized value() {
        return c;
    }
}
```

```
Tråd A : hämta c --> 0 // Lås
Tråd A : beräkna 0++ --> 1
Tråd A : skriv c = 1
// Nu kan nästa tråd anropa
// synkroniserade metoder
Tråd B : hämta c --> 1
Tråd B : beräkna 1-- --> 0
Tråd B : skriv c = 0
```

•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 (statiska metoder) 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) {
  ...
}
```

Notera att synkronisering kan vara mycket svårt att få till på rätt sätt i stora system. En risk med synkronisering är att man kan får sämre prestanda om trådar ofta blockeras och måste vänta. Det kan även uppstå s.k. deadlocks när flera trådar “fastnar” och väntar på varandra. Därför kan asynkron parallellism vara att föredra, om det går att undvika att flera trådar manipulerar samma objekt.

Aktiva Objekt

- Aktiva objekt kan starta (en eller flera) trådar.

```
public static class Timer implements Runnable {
    Thread thread = new Thread(this);
   final String name;
   final int interval;
   int timeLeft;

   public Timer(String name, int intervalSeconds, int startTimeSeconds) {
           this.name = name;
           interval = intervalSeconds;
           timeLeft = startTimeSeconds;
       }
   @Override
    public void run() {
           while (timeLeft > 0) {
      .....
    }
}
```

•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).

DEMO: ActiveObjectnThreadsDemo.java

Ett alternativ för att skapa Aktiva Objekt i modernare versioner av Java är att använda det s.k. ForkJoin-framework. Här kan vi istället för att behöva instrumentera klassen Timer med en explicit tråd, istället skicka dess metoder till en ForkJoinPool. Den tar emot “tasks” och utför dem i en lämplig tråd. Det returneras istället en sk joinForkTask, som är typ en lättviktig motsvarighet till en tråd. Kan t.ex. joinas med huvudtråden för att få den att vänta.

ForkJoin

- Alternativ till att använda explicita trådar.

```
public static class AktivTimer {
   final ForkJoinPool fj = new ForkJoinPool();
   final String name;
   final int interval;
   int timeLeft;

   public ForkJoinTask startTimer() {
        return fj.submit (() -> {....});
   }
```

```
public static void main(String[] args) {
       AktivTimer t1 = new AktivTimer("A", 2, 10);
       AktivTimer t2 = new AktivTimer("B", 3, 12);
       ForkJoinTask taskA = t1.startTimer();
       ForkJoinTask taskB = t2.startTimer();
       taskA.join();
       taskB.join();

       System.out.println("both are done");
    }
```

Se demo-kod
Ett alternativ för att skapa Aktiva Objekt i modernare versioner av Java är att använda det s.k. ForkJoin-framework. Här kan vi istället för att behöva instrumentera klassen Timer med en explicit tråd, istället skicka dess metoder till en ForkJoinPool. Den tar emot “tasks” och utför dem i en lämplig tråd.
Klassen ForkJoinPool har bl.a. en metod submit som skickar en metod att utföra till poolen (kan utföras i FIFO ordning).

När vi skickar något till Poolen får vi tillbaka ett objekt av typ joinForkTask, som är typ en lättviktig motsvarighet till en tråd. Kan t.ex. joinas med huvudtråden för att få den att vänta.

DEMO: ForkJoin.java

Kommunikation mellan trådar

- Gemensamt trådsäkert objekt delas mellan trådar.
- E.g. `Interface` BlockingQueue i Javas API.
  - `ArrayBlockingQueue`, `LinkedBlockingQueue` m.fl.

```
public interface BlockingQueue<E> extends Queue<E> {
   ...
   void put (E e);     //Lägg e i kön om det finns plats, annars vänta.
   E take ();          //Ta element från kön, vänta om den är tom.

   //Lägg e i kö om det finns plats, vänta max angiven tid.
   boolean offer (E e, long timeout, TimeUnit unit);
   //Ta element från kö, vänta max angiven tid om tom.
   E poll (long timeout, TimeUnit unit);
   ...
}
```

•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

- Swing är inte trådsäkert.
- Event Dispatching Thread: sätter upp och hanterar GUI.
  - Skapas automatiskt av Javas runtime system.
  - Kallar t.ex. lyssnarmetoderna (detta gör vi aldrig själv).
- Tyngre beräkningar bör utföras i en separat tråd.

```
public abstract class SwingWorker<T,V>{
   ...
   protected abstract T doInBackground();
   ...
}
```

- SwingWorker<T,V>: T - resultattyp, V - typ för delresultat.
- Överskugga metoden doInBackground

•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.

Demo: Trådar i Swing

```
public static class MyWorker extends SwingWorker<Boolean, Object> {
   ...
   @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");
     }
}
```

•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.

DEMO: Modifierad SwingHelloWorld.java.
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.

Interfacet `Stream`, piplines och parallel exekvering

Detta känner ni igen från Haskell och funktionell programmering, och har ganska nyligen även introducerats i Java.
Vi ska repetera lite om Java Collection Framework och sedan titta på en utökning som gjordes i Java 8 för att tillåta lat evaluering och s.k. pipelines. Vi ska även titta på hur detta kan användas vid parallell exekvering.

Rep: Lambda-uttryck och funktions-interface

- Ett interface med bara en enda abstrakt metod kallas funktions-interface.
- Kan skapa objekt av dessa typer enbart genom att ange lambda-uttryck motsvarande den abstrakta metoden.

```
@FunctionalInterface
public interface Predicate<T> {
   boolean test (T t);                    // Evaluerar predikatet på t
   ...
}
```

```
//Lambda uttrycken specificerar metoden test i Interfacet Predicate
Predicate<Integer> isEven = x -> x%2==0;
Predicate<Integer> isOdd = x -> x%2!=0;
```

- Funktions-interface har bara en abstrakt metod (men kan även ha flera default-metoder).
- Dessa går att använda med lambda-uttryck som specificerar vad den enda abstrakta metoden ska göra. Vi slipper på så vis skapa en inre (anonym) klass, utan det räcker med att specificera metoden.
- Vi har sett flera exempel som ActionListener i Swing, Runnable för trådar och Predicate.

Rep: Java Collections Framework, defaultimplementationer

- Java Collection Framework:
  - Interfaces: `Collection<E>`, utökas av t.ex. `List<E>`, `Set<E>`
  - Klasser, t.ex: `LinkedList<E>, ArrayList<E>, TreeSet<E>, HashSet<E>`
- Default-metoder i interface: innehåller implementationer.

```
Interface Collection<E>{
   ...
   default Stream<E> stream (){...}

}
```

Repetition: Java Collections Framework - en samling interfaces och klasser för datasamlingar. Huvudinterfacet heter Collection<T>, och utökas av många andra interface, t.ex. List<E>, Set<E> osv. Vi har även sett flera klasser som implementerar dessa interface, exempelvis LinkedList<E>, ArrayLi st<E>, TreeSet<E>, HashSet<E>, men det finns även många fler (se API).

I senare versioner av Java utökades funktionaliteten för Interfaces till att även tillåta s.k. “default-metoder”. “Vanliga” interface innehåller bara metodsignaturer (dvs metodens namn, retur- och argumenttyper). Default-metoder innehåller även implementation, så klasser som implementerar interfaces behöver inte definiera dessa default-metoder.

En av orsakerna till att man introducerade default-metoder var att man ville lägga till metoder till interface i Java Collection Framework, vilket skulle ha orsakat mycket jobb för programmerare som underhåller den stora mängd klasser som implementerar dessa interface. Med default-implementationer i interfacet fick dessa klasser “automatiskt” nya metoder.

En sån defaultmetod som lades till i Collection är stream().

Metoden stream() och pipelines

- Interface Stream

```
List<Integer> list = new LinkedList<Integer>();
...
list
  .stream()                 // skapa en stream från list.
  .filter(e -> e%2==0)      // gör en eller flera stream-operationer
  .forEach(e -> System.out.println e); //Avslutande operation
```

- Ger samma resultat som:

```
for(Integer e : list){
  if(e%2 ==0)
    System.out.println e;
}
```

Metoden stream() returnerar ett objekt av typ Stream<E>. Vi kan tänka på en Stream som en lat lista (som i Haskell).
(En stream kan även skapas från andra källor än en Collection, t.ex. klassen Random eller BufferedReader som läser från filer. Finns även hjälp-metoder som Stream.builder för att skapa strömmar från andra typer - se exempel i 17.8 i kursboken)

Stream-objekt kan kopplas ihop i s.k. pipelines. I interfacet Stream definieras flera s.k. stream opertations (aka aggregate operations). Dessa operationer returnerar en ny Stream, så kan därför kopplas ihop. Det som kommer ut som resultat från en operation går in som argument till nästa operation.

Pipelinen avslutas med en s.k. terminal operation, som producerar ett resultat (ej en stream) eller utför någon sido-effekt (som här, skriver ut alla jämna tal).

Många stream-operationer, som t.ex. filter känns igen från funktionell programmering. Notera att många tar ett argument av en typ som är ett funktionellt interface (dvs endast en abstrakt metod). Som vi minns från förra veckan kan vi då ange ett lambda-uttryck som argument. Här tar t.ex. metoden filter ett argument av typen Predicate<T>. Vi behöver inte skapa någon anonym klass, utan kan helt enkelt bara ange ett lambda-uttryck som motsvarar metoden test i interfacet Predicate.

Parallel exekvering av pipelines

- Vi kan enkelt utnyttja parallel exekvering

```
List<Long> list = new LinkedList<Long>();
...
list
  .parallelStream()      // skapa en stream som tillåter parallellisering.
  .filter(e -> isPrime(e))    // gör en eller flera stream-operationer
  .sum();                     //Avslutande operation
```

- Notera: restriktioner på stream-operationer:
  - Ej modifiera källan till strömmen (`list`).
  - *Stateless* (får ej bero på tillstånd som förändras).

Stream-operationer tillåts göra många typer av optimeringar. Notera att stream-operationer använder sig av s.k. lat evaluering, vilket betyder att de inte räknar ut något resultat förrän (om) en avslutande operation kräver det för ett konkret resultat.

Man kan även tillåta att operationerna utförs parallellt, genom metoden parallelStream() (istället för stream()). Vi tillåter då att våra operationer på vår stream delas upp i delar, som exekveras parallellt.

För att allt detta ska fungera får inte stream-operationer göra vad som helst. De får t.ex. inte modifiera själva källan till strömmen (här listan list) och får inte bero på något tillstånd som kan ändras under exekvering av pipelinen (de måste vara stateless).

(I/O)Strömmar, filer och Nätverkskommunikation

Notera: Det här är andra strömmar än de som vi nyss såg. Olyckligtvis heter båda stream i Java...

Strömmar för att läsa och skriva data

- Superklasser: `InputStream` och `OutputStream`
  - Läser/skriver rå data (byte-sekvenser)
  - `FileInputStream` och `FileOutputStream` är subklasser för att läsa/skriva till/från filer.

- Superklasser: `InputStreamReader` och `OutputStreamWriter`
  - Läser/skriver tecken (characters).
  - `FileReader` och `FileWriter` är subklasser för att läsa/skriva tecken till/från filer.

•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

- Buffra läsning/skrivning:
  - `BufferedInputStream`, `BufferedOutputStream`
  - `BufferedReader`, `BufferedWriter`

```
BufferedReader in = new BufferedReader(new FileReader(“foo.in"));
...
String nextLine = in.readLine();
...
Stream<String> = in.lines(); // För att kunna använda pipelines

in.close();
```

•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.
* Kan vara mer effektivt att använda en buffrad läsare när man läser från fil.
•Den fil som hör till en filström öppnas när objektet skapas och stängs när man anropar close.
* Från en BufferedReader går det att skapa en Stream<String> (interfacet för pipelines talar vi om här) med metoden lines().

Filegenskaper

- Klassen File representerar filer och deras egenskaper.

```
File myfile = new File (“lectures/kod/Demo.java“);
...
boolean b = myfile.exists();
boolean b1 = myfile.isDirectory();
long mod = myfile.lastModified();
....
```

•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

- Klassen URL representerar en resurs på internet.

```
URL url = new URL(”http://www.cse.chalmers.se/edu/course/DAT043/”);

URLConnection urlconn = url.openConnection();
InputStream in = urlconn.getInputStream();
// Alt.
InputStram in = url.getInputStream();
```

•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.

DEMO: URLDemo.java

Portar och Sockets

- *Sockets*: för nätverkskommunikation på lägre nivå.
  - Representerar virtuell kanal (via en port) mellan två datorer.
- *Osynkroniserat*: Datagram (inga garantier, kan bli i oordning)
- *Synkroniserat*: Klient-Server

•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

- Datagram: paket med data som skickas över nätverk.
  - Innehåller avsändare+mottagares adress, portnummer.
  - Datainnehåll som array av bytes.
- Klassen InetAdress representerar nätverks adresser.

```
InetAddress adr = InetAddress.getByName(“www.xyz.pqr.se");
String msg = “Hej där!”;
byte[] data = msg.getBytes();   // Konvertera till byte-array.

// Skapa paketet.
DatagramPacket packet = new DatagramPacket(data, data.length, adr, port);
DatagramSocket sock = new DatagramSocket();
socket.send(packet);      // Slutligen: skicka iväg!
```

•För att identifiera datorer används InetAddress. Inga konstruerare, man kan skapa ett objekt från en sträng med klassmetoden getByName(String address). Adressen ska vara antingen symbolisk eller en IP-adress.
•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.
* Inga garantier för att Datagram kommer fram, eller att de kommer fram i samma ordning som de skickades i.

Klient - Server

- Klient skapar socket med serverns adress och port nummer.

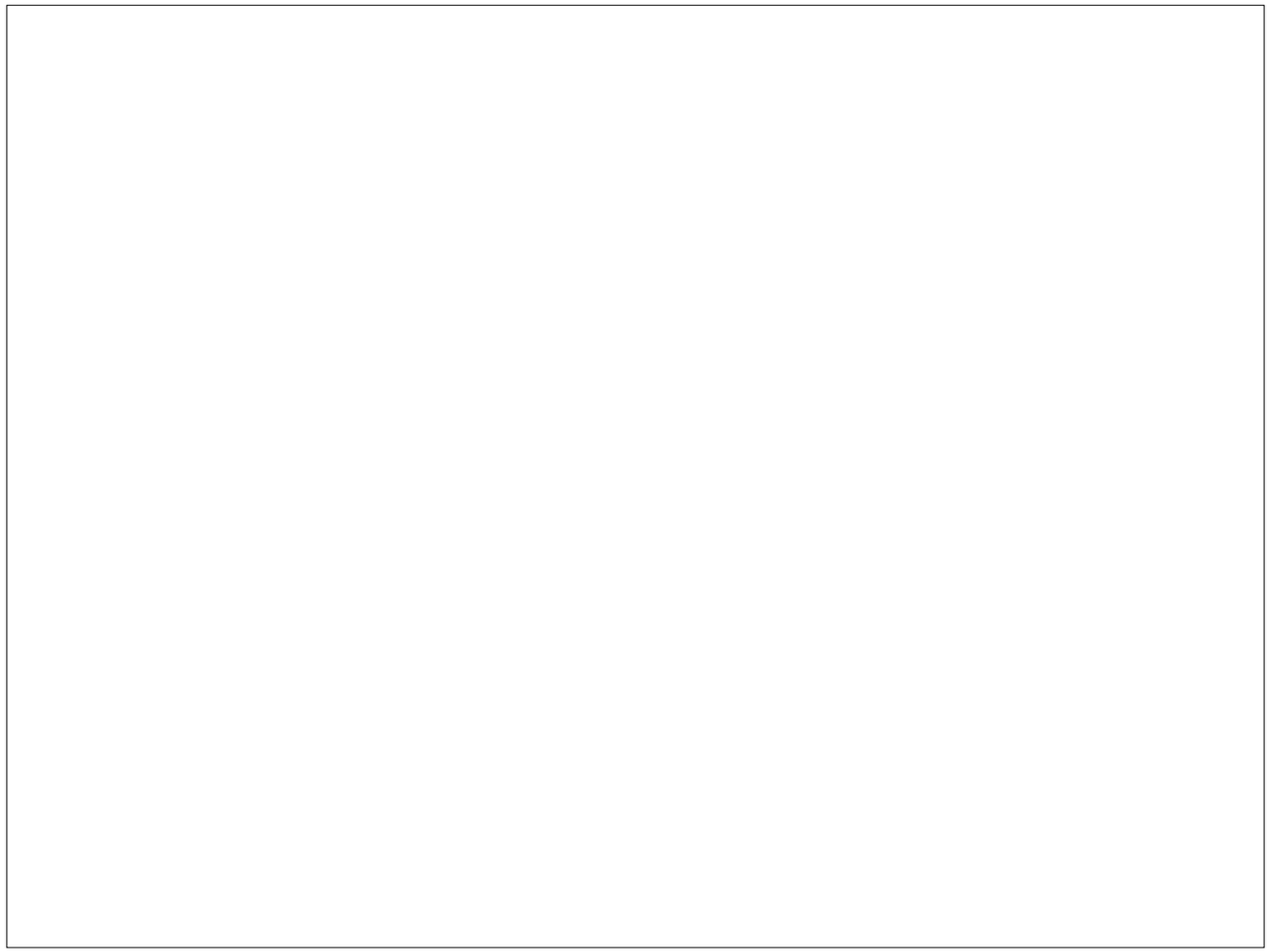
```
Socket sock = new Socket("www.xyz.server.se", 13781);
DataInputStream in = new DataInputStream(sock.getInputStream());
DataOutputStream out = new DataOutputStream(sock.getOutStream());
```

- Server skapar socket för att lyssna efter förbindelser.

```
ServerSocket servSock = new ServerSocket(13781); //lyssna på port 13781

while(true){
  Socket clientSock = servSock.accept();
  // Starta ny tråd (möjligen som aktivt objekt) för clientSock
  // Fortsätt lyssna efter andra klienter.
}
```

•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.



Repetition: Java Collections Framework - en samling interfaces och klasser för datasamlingar. Huvudinterfacet heter Collection<T>, och utökas av många andra interface, t.ex. List<E>, Set<E> osv. Vi har även sett flera klasser som implementerar dessa interface, exempelvis LinkedList<E>