

Lektion 8 – Trådar (threads)

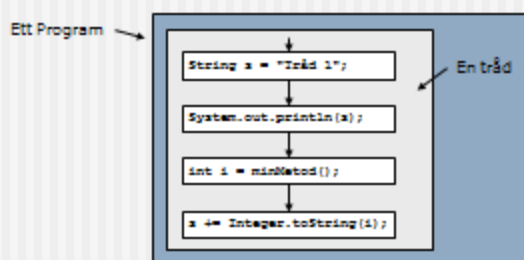
Java för C++-programmerare

Syfte: Att förstå begreppet tråda, deras livscykel och hur de kan användas.
Att kunna implementera komponenter som utnyttjar Javas trådmodell.
Att kunna synkronisera trådar.

Att läsa: Skansholm kapitel 13
<http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>
<http://docs.oracle.com/javase/tutorial/uiswing/concurrency/index.html>

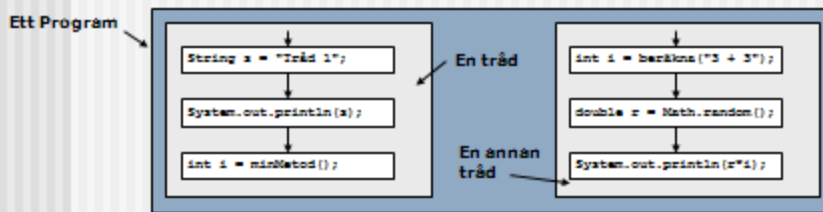
Vad är en tråd (thread)?

- Normalt är våra program en-trådiga
 - Startar i main-metoden.
 - Allt sker i viss ordning (sekventiellt).
 - Två olika saker kan inte ske samtidigt.



Vad är en tråd? (2)

- Med trådar kan vi utföra flera saker samtidigt i ett program.
- Trådarna exekveras parallellt:
 - Simuleras i ett enprocessor-system
 - Äkta parallell exekvering i flerprocessor





Att skapa trådar



- Finns olika sätt att skapa trådar:
 - Ärva klassen `java.lang.Thread` och överlagra metoden `run`.
 - Implementera gränssnittet `java.lang.Runnable` och definiera metoden `run`.
 - Skapa ett objekt av klassen `javax.swing.Timer`.

`java.lang.Thread`



- I Java beskrivs trådar med `Thread`
- Vi skapar en ny aktivitet/tråd genom att skapa ett `Thread`-objekt.

```
Thread nyAktivitet = new Thread();
```

- För att starta tråden anropas `start`

```
nyAktivitet.start();
```

- I `Thread` finns metoden `run` som körs när `start` anropas



Skapa egna trådar

- Derivera en ny klass från klassen `Thread`

```
public class MinKlass extends Thread
```

- Omdefiniera `run`

```
public void run()
{
    // Kod som den nya aktiviteten ska utföra.
    // Sker vanligtvis i en loop.
}
```

- Instansiera klassen och anropa metoden `start`

```
MinKlass klass = new MinKlass();
klass.start();
```

```
public class Djur extends Thread {
    private String typ, tal;

    public Djur(String typ, String tal) {
        this.typ = typ; this.tal = tal;
    }

    public void prata() {
        start();
    }

    public void sov() {
        int millisekunder = (int) (Math.random() * 1000);
        try {
            sleep(millisekunder * 4); // sover max 4 sek
        }
        catch (InterruptedException avbryten) {
            System.out.println("Djuret blev väckt");
        }
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(tal + " " + (i+1) + " från " + typ);
            sov();
        }
    }
}
```



Detta exempel använder sig av trådar för att få flera olika djur att prata samtidigt. Vi skapar en klass Djur vilken ärver sina egenskaper från klassen Thread (eftersom varje djur ska agera självständigt i en egen tråd).

För att få ett djur att börja prata (starta denna aktivitet (tråd)) anropar vi metoden `prata`. Denna anropar i sin tur metoden `start` som kör igång tråden.

I `run`-metoden har vi en `for`-loop som snurrar 10 varv. Vid varje var skrivs djurets läte ut tillsammans med ett ordningsnummer samt typ av djur. Därefter anropas metoden `sov`. I `sov` slumpas det fram hur många millisekunder djuret ska sova innan nästa prat kommer.

Så här fortsätter det till dess att djuret har pratat 10 gånger. Då avslutas `run`-metoden och tråden/aktiviteten är klar.

Se exemplet **Djur.java**.

Exempel med Thread (2)

■ Test av Djur

```
public class TestAvDjur
{
    public static void main(String[] args)
    {
        System.out.println("TestAvDjur.main() start");

        Djur ko = new Djur("Ko", "muuu");
        Djur katt = new Djur("Katt", "mjau");

        ko.prata();
        katt.prata();

        System.out.println("TestAvDjur.main() slut\n");
    }
}
```

Output from console window:

```
TestAvDjur.main() start
TestAvDjur.main() slut
muuu 1 från Ko
mjau 1 från Katt
mjau 2 från Katt
muuu 2 från Ko
mjau 3 från Katt
muuu 3 från Ko
mjau 4 från Katt
mjau 5 från Katt
mjau 6 från Katt
mjau 7 från Katt
muuu 4 från Ko
muuu 5 från Ko
mjau 8 från Katt
mjau 9 från Katt
muuu 6 från Ko
mjau 10 från Katt
muuu 7 från Ko
muuu 8 från Ko
muuu 9 från Ko
muuu 10 från Ko
Tryck på en valfri tangent
```

TestAvDjur: i `main`-metoden skapar vi två **Djur**-objekt. På dessa objekt anropar vi metoden `prata` för att trådarna ska köras starta.

Sist i `main` görs en utskrift att `main` är slut.

Som du ser av bilden till höger kommer den sista utskriften i `main` att skrivas ut långt före att djuren har pratat klar. Även om `main` är slut kommer inte programmet att avslutas eftersom ett program inte är slut förrän alla trådar i programmet är klar.

Se exemplet **TestAvDjur.java**.



Användbara metoder

■ Några användbara metoder i Thread

```
Thread t = new Thread(); // Skapar en ny tråd t

t.start();                // startar tråden t

t.sleep(ms);              // låter tråden sova i ms millisekunder, kastar ett
                          // InterruptedException om tråden avbryts

t.interrupt();            // ber tråden avbryta sin aktivitet
t.isInterrupted();        // returnerar true om tråden blivit avbruten

t.join();                 // väntar till att t har kört klart
t.join(ms);               // väntar som längst ms till att t har kört klart

t.setPriority(p);         // ändrar trådens prioritet till p
t.getState();             // returnerar trådens nuvarande tillstånd
```



Avbryta en tråd

- För att avbryta en tråd kan man anropa `interrupt()`
- Tråden behöver inte avbrytas direkt
 - Sover tråden genereras ett `InterruptedException`
 - I annat fall sätts en avbrottsflagga
- I `run` måste vi kontrollera om tråden ska avbrytas

För att visa på hur en tråd kan avbrytas, genom att metoden `interrupt` anropas, utgår vi från en klass som slumpar fram punkter på en JPanel. Vi skapar en klass med namnet `SlumpTråd` och låter den ärvä klassen `Thread`.



```
import java.awt.Graphics;
import javax.swing.JPanel;

public class SlumpTråd extends Thread {
    private JPanel panel; // Panel att rita på

    // Konstruktorn
    public SlumpTråd(JPanel panel) {
        this.panel = panel;
    }

    public void avbryt() { // Anropas när vi vill avbryta tråden
        System.out.println("SlumpPanel.avbryt()");
        this.interrupt();
    }

    public void run() {
        System.out.println("SlumpTråd.run() - startar");
        Graphics g = panel.getGraphics(); // Panelens storlek

        int w = panel.getWidth();
        int h = panel.getHeight();

        while(!interrupted()) { // Loopa tills avbrott
            int x = (int)(Math.random() * w); // Slumpa en punkt
            int y = (int)(Math.random() * h);
            g.fillOval(x, y, 4, 4); // Rita punkten
            try {
                this.sleep(10); // Sover en stund
            } catch (InterruptedException ie) {
                // Dags att avbryta, gör ett break.
                break;
            }
        }
        System.out.println("SlumpTråd.run() - slutar");
    }
}
```

När en ny `SlumpTråd` skapas måste skaparen skicka med en referens till ett `JPanel`-objekt. Denna panel används för att rita ut punkterna på. Det intressant i denna klass är `run`-metoden och `while`-satsen i denna. Vi kontrollerar om tråden har ombetts att avbrytas genom att anropa metoden `interrupted()`. Denna metod returnerar `false` så länge `interrupt` inte har anropats. Sist i `run`-metoden sover vi en stund. Observera att när vi anropar `sleep` så kan ett `InterruptedException` kastas om metoden `interrupt` anropas under "sömnen". Eftersom vi vill att tråden ska avbrytas skriver vi `break` för att hoppa ur `while`-satsen. Ta en titt på exemplen `SlumpaTråd.java` och `TestAvSlumpTråd.java`.



Vänta på en tråd

- För att i en tråd vänta på att en annan tråd ska köra klart anropas `join()`

```
public class TestAvDjurJoin {  
  
    public static void main(String[] args) {  
        System.out.println("Start");  
        Djur ko = new Djur("Ko", "muuu");  
        Djur katt = new Djur("Katt", "mjau");  
        ko.prata();  
        katt.prata();  
  
        try {  
            ko.join();  
            katt.join();  
        } catch (InterruptedException ie) {}  
  
        System.out.println("Slut");  
    }  
}
```

Om vi vill att main avslutas medan en eller fler trådar är aktiva kan vi förhindra detta genom att anropa metoden `join` på en tråd. Detta förhindrar då den aktuella tråden (som gör anropet) att fortsätta sin aktivitet till dess att den andra (anropade) tråden är avslutad.

I bilden ovan kommer nu båda djuren att hinna prata klart innan main-metoden fortsätter med sin sista utskrift. Observera att anropet till `join` måste läggas i en try-catch-sats eftersom ett `InterruptedException` kastas om tråden avbryts (någon anropar `interrupt()`).

Se exemplen **Djur.java** och **TestAvDjurJoin.java**.

En tråds prioritet

- När flera trådar körs samtidigt kan trådarna ges olika prioritet
 - Viktiga trådar ges hög prioritet
 - Mindre viktiga ges en lägre prioritet
- Prioritet ges som ett heltal mellan
 - `Thread.MIN_PRIORITY` --> 0
 - `Thread.NORM_PRIORITY` --> 5
 - `Thread.MAX_PRIORITY` --> 10


```
public class TestAvDjurPrioritet
{
    public static void main(String[] args)
    {
        System.out.println("TestAvDjurPrioritet.main() start");

        Djur2 ko = new Djur2("Ko", "muuu");
        Djur2 katt = new Djur2("Katt", "mjau");
        Djur2 hund = new Djur2("Hund", "voff");

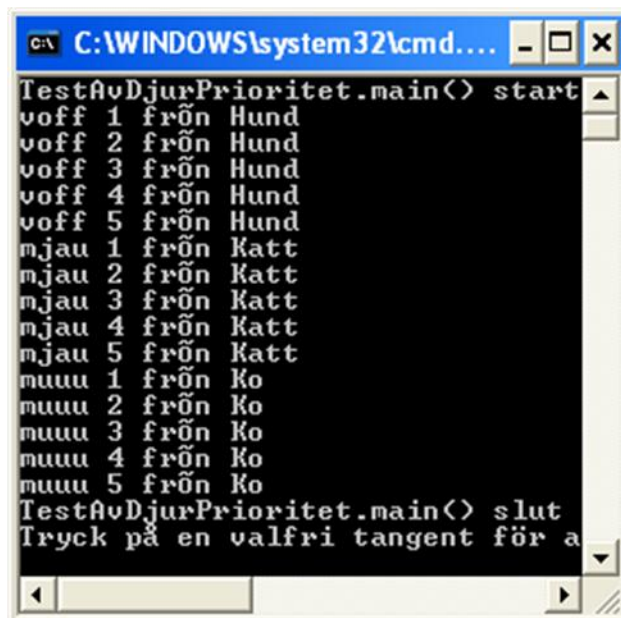
        ko.setPriority(Thread.MIN_PRIORITY);
        hund.setPriority(Thread.MAX_PRIORITY);

        ko.prata();
        katt.prata();
        hund.prata();

        try {
            ko.join();
            katt.join();
            hund.join();
        }
        catch (InterruptedException ie) {}

        System.out.println("TestAvDjurPrioritet.main() slut");
    }
}
```

Genom att anropa metoden `setPriority` för en tråd kan vi ge tråden en viss prioritet. I detta exempel skapas tre olika djur (trådar) där alla ges olika prioriteter. Resultatet visar att hunden pratar klart först eftersom den har getts den högsta prioriteten. Kon kommer att pratas sist



```
C:\WINDOWS\system32\cmd...
TestAvDjurPrioritet.main() start
voff 1 frön Hund
voff 2 frön Hund
voff 3 frön Hund
voff 4 frön Hund
voff 5 frön Hund
mjau 1 frön Katt
mjau 2 frön Katt
mjau 3 frön Katt
mjau 4 frön Katt
mjau 5 frön Katt
muuu 1 frön Ko
muuu 2 frön Ko
muuu 3 frön Ko
muuu 4 frön Ko
muuu 5 frön Ko
TestAvDjurPrioritet.main() slut
Tryck på en valfri tangent för a
```



eftersom den har getts den lägsta möjliga prioriteten.

I `Djur2.java` är metoden `sov` borttagen så att djuret pratar så mycket de får. Då blir det alltså operativsystemet som avgör vilken tråd som får exekvera med hänsyn tagen till de tilldelade prioriteterna.

Om vi i tråden anropar `sleep` kommer operativsystemet automatiskt låta andra trådar exekvera medan vår tråden sover. Se exemplen **`Djur2.java`** och **`TestAvDjurPrioritet.java`**.

Synkronisering av trådar



- Metoder i objekt som flera trådar har tillgång till bör synkroniseras
- När en tråd anropar en metod som är synkroniserad sätts ett lås
- Låset tvingar andra metoder att vänta till första tråden är klar och låset släpps
- Flera metoder i samma klass kan vara synkroniserade

Synkronisering av trådar (2)



■ Exempel

```
public class TelePrinter {  
  
    public void print(String s) {  
        for (int i = 0; i < s.length(); i++)  
        {  
            System.out.print(s.charAt(i));  
  
            try {  
                Thread.sleep(100);  
            }  
            catch (InterruptedException ie)  
            { }  
        }  
        System.out.println();  
    }  
}
```

```
public class PrintThread extends Thread  
{  
    private String text;  
    private TelePrinter printer;  
  
    public  
        PrintThread(TelePrinter p, String t)  
    {  
        this.printer = p;  
        this.text = t;  
    }  
  
    public void run()  
    {  
        printer.print(text);  
    }  
}
```



Exemplet visar en situation där synkronisering av metoder är nödvändig. Programmet skriver ut textsträngar, ett tecken i taget, med en viss fördröjning. Det kan liknas vid en gammal teleprinter. Vi skapar en klass med namnet `TelePrinter` vilken innehåller endast en metod `print`.

Metoden `print` tar en textsträng som argument och loopar igenom strängen och skriver ut strängens tecken en efter en. Vi gör en fördröjning mellan varje bokstav genom att anropa `Thread.sleep()`.

För att utnyttja denna teleprinter skapar vi en tråd med namnet `PrintThread`. Denna innehåller två instansvariabler: den text som ska skrivas ut och den teleprinter som ska skriva ut texten. I trådens `run`-metod anropar vi metoden `print` för att starta utskriften av texten.

Synkronisering av trådar (3)

■ Exempel

```
public class TestAvTelePrinter
{
    public static void main(String[] args)
    {
        TelePrinter printer = new TelePrinter();

        PrintThread pt1 = new PrintThread(printer, "abcdefghijklmnopqrstuvwxyz");
        PrintThread pt2 = new PrintThread(printer, "0123456789012345678901234567");

        pt1.start();
        pt2.start();

        printer.print("Detta är main som skriver");
    }
}
```

C:\WINDOWS\system32\cmd.exe

Da8ehitc2td3ae4 f56g6rh7 i8nj9ak8ilinn2 n3so4op5mq6 r7ss8kt9ra8iuiuv2ex3re4
45667

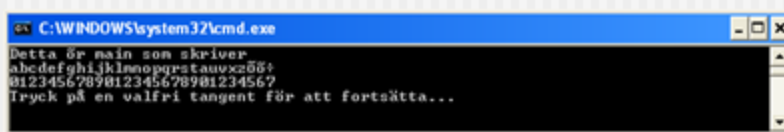
Tryck på en valfri tangent för att fortsätta...

Vi skapar ett objekt av klassen `TelePrinter` samt två objekt av `PrintThread`. Vi använder samma teleprinter när vi skapar de två trådarna (de två `PrintThread`-objekten). För att starta de två trådarna anropar vi metoden `start`. Sist i `main` använder vi samma teleprinter för att skriva ut ytterligare en text. Vi har därför tre olika trådar som använder samma metod i klassen `TelePrinter`. Som du ser i bilden ovan är utskriften inte speciellt ordnad utan bokstäver från de tre olika texter som ska skrivas ut blandas huller om buller. Se exemplen **`TelePrinter.java`**, **`PrintThread.java`** och **`TestAvTelePrinter.java`**.

Synkronisering av trådar (4)

- Deklarera metoden `print` i klassen `TelePrinter` som `synchronized`

```
public synchronized void print(String s)
```



- Nu kan endast en tråd i taget exekvera i metoden `print`

Ändra källkoden för klassen **TelePrinter.java** så att metoden `print()` deklareras som `synchronized`!

Ett alternativ till att derivera en klass från `Thread` är att implementera interfacet `Runnable`.

`java.lang.Runnable`

- Vad göra om en klass redan ärver?
- I stället för att ärva `Thread` kan man implementera `Runnable`

```
public class MinKlass extends AnnanKlass implements Runnable {  
    private Thread aktivitet = new Thread(this);
```

- Definiera därefter `run` som vanligt

```
public void run {  
    while (!aktivitet.isInterrupted())  
    {  
        // Gör något  
        try {aktivitet.sleep(1000);}  
        catch (InterruptedException e) {}  
    }  
}
```

java.lang.Runnable (2)

- Kan nu inte starta/avbryta tråden som när Thread ärvs

```
MinKlass mk = new MinKlass();  
mk.start(); // FEL, eftersom ingen start() finns i MinKlass
```

- Tillhandahåll egna metoder för att starta/pausa/stoppa tråden

```
public void start() {  
    if (aktivitet == null) {  
        aktivitet = new Thread(this);  
        aktivitet.start();  
    }  
}
```

```
public void stop() {  
    if (aktivitet != null) {  
        aktivitet.interrupt();  
        aktivitet = null;  
    }  
}
```

java.lang.Runnable (3)

- Exempel linjer som studsar

```
public class StudsLinje extends JPanel implements Runnable {  
    private Thread aktivitet;  
    private boolean ritaMedLinje;  
    private final int antalPunkter;  
    private int x[], y[], xfart[], yfart[];  
  
    public StudsLinje(int antalPunkter, boolean ritaMedLinje) {  
        this.antalPunkter = antalPunkter;  
        this.ritaMedLinje = ritaMedLinje;  
        // Skapar arrayer för koordinater och fart  
        x = new int[antalPunkter];  
        y = new int[antalPunkter];  
        xfart = new int[antalPunkter];  
        yfart = new int[antalPunkter];  
    }  
  
    public void start() {  
        if (aktivitet == null) {  
            slumpaPunkter();  
            aktivitet = new Thread(this);  
            aktivitet.start();  
        }  
    }  
  
    public void stop() {  
        if (aktivitet != null) {  
            aktivitet.interrupt();  
            aktivitet = null;  
        }  
    }  
}
```

Vi måste använda oss av interfacet Runnable är när vi redan utnyttjat arvsmekanismen en gång, t.ex. om skapar en komponent som ärver JPanel, och därför inte kan ärva klassen Thread.




Det komponenten ska göra är att visa ett antal punkter som studsar runt i panelen. Antalet punkter bestäms när komponenten skapas (konstruktorn). I konstruktorn skapas även de arrayer som används för punkternas koordinater och fart.

Eftersom klassen inte ärver sina egenskaper från Thread kan vi inte anropa metoderna `start()` och `interrupt()` som tidigare för att starta och avbryta tråden. Vi måste därför i klassen tillhandahålla egna metoder för detta. För enkelhetens skull kallar vi dem `start` och `stop`.

I `start` måste vi göra en kontroll om tråden aktivitet redan är igång eller inte. Detta görs genom att kontrollera om aktivitet refererar till `null`. Om så är fallet finns ingen tråd och vi kan skapa en ny tråd av det aktuella objektet och starta tråden med `aktivitet.start()`. I denna metod anropas även `slumpaPunkter` som slumpar fram koordinater och fart för de olika punkterna. I metoden `stop` kontrollerar vi först om tråden aktivitet refererar till `null` eller inte. För att avbryta tråden med `interrupt` får aktivitet inte referera till `null` eftersom ett `NullPointerException` då kastas. Refererar tråden inte till `null` är det säkert att avbryta tråden med `aktivitet.interrupt()`. Vi sätter därefter aktivitet att referera till `null` så att vi kan starta den senare igen med `start()`.

I klassen måste vi givetvis implementera metoden `run()` som definieras i `Runnable`. I `run`-metoden flyttar vi punkterna och gör kontroller om punkterna studsar mot någon "vägg". För att rita ut punkterna i panelen överlagrar vi metoden `paintComponent`. I denna ritar vi ut varje punkt som en fylld oval med `g.fillOval`. Om `ritaMedLinje` är satt till `true` ritar vi ut dessutom även en polygon med `g.drawPolygon`. Exemplet finns i **StudsLinje.java**.



java.lang.Runnable (4)

■ Exempel


```
import java.awt.*;
import javax.swing.*;
public class TestAvStudsLinje extends JFrame
    implements ActionListener
{
    private StudsLinje linje;

    public TestAvStudsLinje()
    {
        // Sätt storlek, titel, placering m.m
        // för fönstret

        linje = new StudsLinje(15, true);
        add(linje, BorderLayout.CENTER);

        setVisible(true);
        linje.start();
    }

    public static void main(String[] args) {
        TestAvStudsLinje test = new TestAvStudsLinje();
    }
}
```





För att testa skapar vi ett nytt applikationsfönster i vilken ett objekt av StudsLinje skapas. Vi anger att antalet punkter ska vara 15 samt att linjer ska ritas ut mellan punkterna. Vi lägger till objektet i fönstret och visar fönstret.

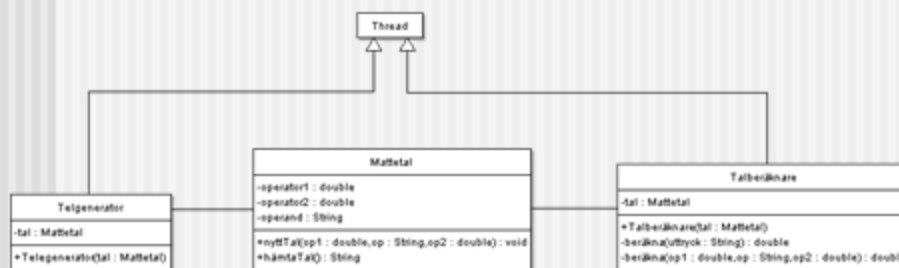
Ta en titt på exemplet **TestAvStudsLinjer.java**, som även innehåller knappar för att starta och stoppa animeringen av punkter.

wait och notify

- Ibland är det nödvändigt att flera trådar kommunicerar med varandra
- Kommunikationen kan ske via ett delat objekt (helst en buffer)
- Trådarna behöver meddela varandra när objektet förändras
 - wait – tråden väntar
 - notify – väntande tråd kan fortsätta
 - notifyAll – om flera trådar väntar

Exempel wait / notify

- Trådar som genererar slumpade mattetal och beräknar dessa tal






För att ge ett exempel på s.k. Producer/Consumer-problem utgår vi från ett program som innehåller två trådar.

- Ena tråden genererar slumpar mattetal av typen $1 + 2$, $3 / 2$, $5 - 1$ etc.
- Den andra tråden är tänkt att beräkna de mattetal första tråden genererar.

Eftersom en tråd genererar tal och en tråd beräknar tal måste de båda trådarna kommunicera med varandra så att den beräknande tråden får det tal den genererande tråden skapar.

Kommunikationen sker via klassen `Mattetal`. Denna klass innehåller tre instansvariabler och två metoder (båda är synkroniserade eftersom flera trådar kan manipulera data i klassen samt att de måste vara synkroniserade för att använda `wait` och `notify`). Instansvariablerna innehåller de två talen och vilken operator (+, -, /, *) som ska appliceras på de båda talen. Den ena metoden, `nyttTal`, används för att sätta ett nytt mattetal (ge instansvariablerna nya värden). Den andra metoden, `hämtaTal`, returnerar aktuellt mattetal som en sträng.



Exempel `wait / notify` (2)

■ `Mattetal`

```
public synchronized void nyttTal(double op1,
    String op, double op2) {
    while (operator != null) {
        try { wait(); }
        catch (InterruptedException ie) {}
    }

    operand1 = op1; operator = op; operand2 = op2;
    notify();
}

public synchronized String hämtaTal() {
    while (operator == null) {
        try { wait(); }
        catch (InterruptedException ie) {}
    }
    String uttryck = Double.toString(operand1) +
        " " + operator + " " +
        Double.toString(operand2);

    operator = null;
    notify();
    return uttryck;
}
```

I klassen `Mattetal` används instansvariabeln `operator` som en indikator på om ett mattetal är satt eller inte. Om `operator` refererar till `null` så innebär det att inget mattetal är satt. Om `operator` refererar till något annat än `null` innebär det att ett mattetal är satt. Detta används i de båda metoderna för att kontrollera om det är ok att sätta ett nytt mattetal och om det finns något mattetal att hämta.

I metoden `nyttTal` kontrolleras om `operator` refererar till något annat än `null`. Om så är fallet finns redan ett mattetal satt och metoden `wait` anropas för att tråden ska vänta till att någon



hämtar talet. När det inte längre finns något mattetal satt (operator refererar till `null`) sätts nya värden på instansvariablerna och metoden `notify` anropas för att meddela eventuella trådar som väntar på att få hämta talet.

I metoden `hämtaTal` måste en tråd som vill hämta aktuellt mattetal vänta om operator refererar till `null` (eftersom det då inte finns något mattetal att hämta). När operator refererar till något annat än `null` skapas strängen som ska returnera talet. Instansvariabeln `operand` sätts även till `null` för att indikera att aktuellt tal har hämtats och att det är ok att sätta ett nytt tal. Metoden `notify` anropas för att meddela andra trådar detta.

Se exemplet **Matteklass.java**.

Exempel wait / notify (3)

■ Talgenerator

```
public class Talgenerator extends Thread {
    private Mattetal tal;

    public Talgenerator(Mattetal tal) {
        this.tal = tal;
    }

    public void run() {
        while (!isInterrupted()) {
            try {
                this.sleep((int)(Math.random() * 10000));
            }
            catch (InterruptedException ie) {
                break;
            }

            int op1 = (int)(Math.random() * 11);
            int op2 = (int)(Math.random() * 11);
            // Slumpar vilken operator som ska användas
            tal.nyttTal(op1, op, op2);
        }
    }
}
```

Klassen `Talgenerator` ärver sina egenskaper från `Thread`. Den enda instansvariabeln är ett objekt av klassen `Mattetal`. När ett objekt skapas av klassen måste man ange det `Mattetal`-objekt som ska användas när nya mattetal genereras. I `run`-metoden används en `while`-loop som körs så länge som någon inte bitt tråden att avbryta med `interrupt()`.

Vi börjar med att låta tråden sova en stund bara för att simulera att det kan ta lång tid att generera ett nytt mattetal. Därefter slumpas värden för talet fram samt vilken operator som ska används. Till sist anropas metoden `nyttTal` för att sätta det nya talet. Observera att detta anrop kan innebära att tråden får vänta (`wait`) om det aktuella mattetalet i `Mattetal (tal)` ännu inte har hämtats.

Precis som `Talgenerator` ärver `Talberäknare` sina egenskaper från `Thread` och har som instansvariabel ett objekt av `Matteklass` (vilken måste anges för att skapa ett objekt av klassen).



I `run`-metoden skapas ett `Scanner`-objekt för att läsa indata från användaren. I `while`-loopen börjar vi med att hämta ett tal från `Mattetal` (`tal`). Om inget tal finns att hämta (talgeneratorn har inte hunnit generera ett nytt tal) kommer denna tråd att få vänta (`wait`). När ett nytt tal har hämtats skriver vi ut talet på skärmen och frågar användaren efter svar. Talet beräknas och användarens svar kontrolleras om det är korrekt eller inte.

Se exemplen **Talberäknare.java** och **TestAvMattetal.java**.

Piped-strömmar



- För att låta två aktiva objekt kommunicera med varandra kan följande klasser användas:
 - `PipedReader`
 - `PipedInputStream`
 - `PipedWriter`
 - `PipedOutputStream`
- Klasserna för tecken- och byte-strömmar fungerar på liknande sätt

Piped-strömmar (2)



- Varje tråd ska ha en in- och en utström
- Inströmmen i första tråden är kopplad till utströmmen i andra tråden
- Inströmmen i andra tråden är kopplad till utströmmen i första tråden

```
PipedWriter utTråd1 = new PipedWriter();
PipedReader inTråd2 = new PipedReader(utTråd1);
PipedWriter utTråd2 = new PipedWriter();
PipedReader inTråd1 = new PipedReader(utTråd2);

Thread tråd1 = new Aktivitet(utTråd1, inTråd1);
Thread tråd2 = new Aktivitet(utTråd2, inTråd2);
```

- Skriver enbart strängar



Exempel Piped-strömmar

```
import java.io.*;

public class Talgenerator extends Thread {
    private PipedWriter out;
    private PipedReader in;

    public Talgenerator(PipedWriter out, PipedReader in) {
        this.out = out;
        this.in = in;
    }

    public void run() {
        while(!isInterrupted()) {
            // Samma kod som från Exemplet Talgenerator i Lektion 1 (Ex0122)

            PrintWriter pw = new PrintWriter(out);
            pw.println(op1 + " " + op + " " + op2);

            BufferedReader br = new BufferedReader(in);
            String svar = br.readLine();
        }
    }
}
```

För att visa på hur två aktiva objekt kan kommunicera med varandra med hjälp av PipedWriter och PipedReader utgår vi i från ett exemplet med Talgenerator och Talberäknare, som via ett objekt av klassen Mattetal kunde kommunicera med varandra.

Här visar vi hur samma trådar i stället använder PipedWriter och PipedReader för att direkt kommunicera med varandra.

Som instansvariabler använder vi en PipedWriter och en PipedReader (som är kopplad till den andra trådens PipedWriter). Trådens run-metod körs så länge inte någon brett tråden att avbryta. Inuti while-loopen används samma kod som förut att slumpa fram uttrycket. I stället för att använda sig av metoden write i PipedWriter, som skriver tecken för tecken, kopplar vi på en PrintWriter. Vi kan då enkelt använda oss av metoden println för att skriva en hel sträng. Observera att det är onödigt att skapa detta PrintWriter-objekt varje gång vi ska skicka något till den andra tråden. Som instansvariabel skulle vi i stället för en PipedWriter haft en PrintWriter. Konstruktorn tar fortfarande emot en PipedWriter, men skapar direkt en PrintWriter av denna (så som vi gör nu i run-metoden).

Efter att vi har skickat uttrycket till den andra tråden skapar vi en BufferedReader av PipedReader. Vi kan då enkelt använda metoden readLine för att läsa av svaret som den andra tråden skickar. Nu gör vi inget med svaret i denna tråd. Den andra tråden skulle kunna skicka sitt svar på uttrycket och att denna tråd får kontrollera om det är rätt eller fel.

I exemplet ovan har vi inte tagit med den felhantering som behövs. Vi har inte heller stängt strömmarna sist i run-metoden vilket vi bör göra.

Se exemplet **Talgenerator.java** i mappen PipedStreams.



Exempel Piped-strömmar (2)

```
public class Talberäknare extends Thread {
    private PipedWriter out; private PipedReader in;

    public Talberäknare(PipedWriter out, PipedReader in) {
        this.out = out; this.in = in;
    }

    public void run() {
        Scanner tgb = new Scanner(System.in);
        while (!isInterrupted()) {
            BufferedReader br = new BufferedReader(in);
            String uttryck = br.readLine();
            System.out.print(uttryck + " = ");
            double svar = tgb.nextDouble();

            PrintWriter pw = new PrintWriter(out);
            if (svar == beräkna(uttryck))
                pw.println("ok");
            else
                pw.println("fel");
        }
    }
}
```

Även i klassen Talberäknare använder vi som instansvariabler en PipedWriter och en PipedReader (som är kopplad till den första trådens PipedWriter). Trådens run-metod körs så länge inte någon brett tråden att avbryta. Inuti while-loopen börjar vi med att skapa en BufferedReader kopplad till PipedReader så att av kan ta emot strängen som skickas från första tråden. Därefter får användaren skriva in sitt svar.

Vi skapar en PrintWriter av vår PipedWriter av samma anledning som i Talgenerator. Beroende på om användaren skrivit rätt svar eller inte skickas strängen "ok" eller "fel" till den första tråden. Se exemplet **Talberäknare.java** i mappen PipedStreams



Exempel Piped-strömmar (3)

```
// Skapar de strömmar som behövs
PipedWriter utTråd1 = new PipedWriter();
PipedReader inTråd2 = new PipedReader(utTråd1);
PipedWriter utTråd2 = new PipedWriter();
PipedReader inTråd1 = new PipedReader(utTråd2);

// Skapar trådar
Talgenerator generator = new Talgenerator(utTråd1, inTråd1);
Talberäknare beräknare = new Talberäknare(utTråd2, inTråd2);

// Startar trådarna
generator.start();
beräknare.start();
```





För att låta de två trådarna kommunicera med varandra krävs det nu att vi skapar två `PipedReader` och två `PipedWriter` (en till vardera tråd) där ena trådens `PipedReader` är kopplad till den andra trådens `PipedWriter`. Därefter skapar vi trådarna där vi skickar med de skapade `Piped`-strömmarna.

Se exemplet **`TestAvMattetal.java`** i mappen `PipedStreams`.