

16 Aktiva objekt och trådar

- ◆ Parallella program, processer och trådar
- ◆ Aktiva Swing-komponenter
- ◆ Trådar i Java
- ◆ Synkronisering
- ◆ Trådar och Swing

Parallella program och processer

- ◆ De flesta operativsystem tillåter samtidig exekvering av flera program i **parallella processer**
 - Processbyte i OS är en resurskrävande operation
- ◆ Ett **parallellt program** har flera exekveringspunkter
- ◆ Ett **realtidssystem** är ett (parallellt) program med tidskrav
 - Ex. program för övervakning och styrning

Trådar

- ◆ En **tråd** är en "lättnviktsprocess" där parallellismen administreras internt av programspråkets eget exekveringsmaskineri
 - till en lägre kostnad än hantering av OS-processer
- ◆ Trådar tilldelas exekveringstid i turordning (*time-slicing*)
- ◆ Trådar kan exekveras med olika **prioritet**
- ◆ Vissa programmeringsspråk, t.ex. Ada, har inbyggda språkkonstruktioner för parallellprogrammering
- ◆ I Java utförs parallellprogrammering med aktiva objekt

Korrekthetsproblem i parallella program

När flera processer exekverar samtidigt i olika inbördes hastigheter finns risk för komplikationer

- ◆ **Låsning:** Flera processer väntar i evighet på respons från varandra eller på tillgång till en gemensam resurs
- ◆ **Svält:** En process hindras varaktigt från att göra framsteg
- ◆ **Inkonsistens:** När flera processers operationer på en gemensam resurs överlappar i tiden kan resultatet bli felaktigt även om de enskilda operationerna utförs korrekt var för sig

Exempel: Felaktig banktransaktion

```
public class Konto {
    private long saldo;

    public Konto(long startBelopp) {
        saldo = startBelopp;
    }

    public void trans(long x) {
        long nyttSaldo = saldo + x;
        if (nyttSaldo >= 0)
            saldo = nyttSaldo;
    }
}

...
Konto k(1000);
```

Möjligt scenario

variabel	exekveringssteg				
	t1	t2	t3	t4	t5
k.saldo	1000	1000	1000	3000	300
process 1 exekverar k.trans(-700);	x ₁	-700	-700	-700	-700
nyttSaldo ₁	?	300	300	300	300
process 2 exekverar k.trans(2000);	x ₂	2000	2000	2000	2000
nyttSaldo ₂	?	?	3000	3000	3000
		process 1 läser	process 2 läser	process 2 skriver	process 1 skriver

Fakt: 2000 borde det alltså ha blivit. Båda bank!

Passiva och aktiva objekt

- ♦ Ett **passivt objekt** utför inget på eget initiativ utan reagerar enbart på metदानrop från omgivningen
- ♦ Ett **aktivt objekt** exekverar **autonomt** sin egen algoritm.
 - men kan även manipuleras med metoder
- ♦ Ett **parallellt** objektorienterat program har flera aktiva objekt

Aktiva objekt och trådar i Java

- ♦ Aktiva objekt exekveras i trådar
 - En tråd per aktivt objekt
- ♦ Enkla aktiva objekt kan styras med hjälp av en timer
 - utan extra trådar
 - En Timer
 - ▼ exekveras i en egen bakgrundstråd.
 - ▼ genererar händelser av typen `ActionEvent`.

Några standardklasser i Java för parallellprogrammering

Klasser

- ♦ `javax.swing.Timer`
- ♦ `java.util.Timer`
- ♦ `java.util.TimerTask`
- ♦ `java.lang.Thread`

Användning

- objekt
- objekt
- arv
- objekt, arv

Abstrakta gränssnitt

- ♦ `java.lang.Runnable` arv (implement)

Ex 1. Klocka med `javax.swing.Timer`

```
import javax.swing.*;
import java.awt.event.*;
/**
 * A clock based on javax.swing.Timer
 */
public class Clock implements ActionListener
{
    private long seconds = 0;
    private final int period = 1000;
    javax.swing.Timer timer = new javax.swing.Timer(period, this);

    public void start() {
        timer.start();
    }

    public void stop() {
        timer.stop();
    }

    public void actionPerformed(ActionEvent e) {
        System.out.println(++seconds);
    }
}
```

Anropas varje period ms

Ex 2. Klocka med `java.util.Timer`

```
import java.util.*;
/**
 * run is called periodically by java.util.Timer
 */
public class ClockTick extends TimerTask
{
    private long seconds = 0;

    public void run() {
        System.out.println(++seconds);
    }
}
```

forts. Klocka med `java.util.Timer`

```
import java.util.*;
/**
 * A clock based on java.util.Timer
 */
public class Clock2
{
    private final int delay = 1000;
    private final int period = 1000;
    java.util.Timer timer = new java.util.Timer();
    ClockTick tick = new ClockTick();

    public Clock2() {
        timer.schedule(tick, delay, period);
    }

    public void stop() {
        timer.cancel();
        tick.cancel();
    }
}
```

Schemalägg tick.run() varje period ms

Trådklass med arv från Thread

```
public class MyThreadClass extends Thread
{
    // ...

    public void run() {
        while ( ! interrupted() ) {
            try {
                sleep(some time interval);
            }
            catch (InterruptedException e) {
                break;
            }
            // Do some useful work here ...
        }
    }
}
```

Trådklass med internt Thread-objekt

```
public class MyThreadClass implements Runnable
{
    private Thread activity = new Thread(this);
    // ...

    public void run() {
        while ( ! Thread.interrupted() ) {
            try {
                Thread.sleep(some time interval);
            }
            catch (InterruptedException e) {
                break;
            }
            // Do some useful work here ...
        }
    }
}
```

Exempel: skrivare

```
public class Writer extends Thread
{
    String text;
    private long interval; // ms

    public Writer(String text, long interval)
    {
        this.text = text;
        this.interval = interval;
    }

    public void run() {
        while ( ! interrupted() ) {
            try {
                sleep(interval);
            }
            catch (InterruptedException e) {
                break;
            }
            System.out.println(text); // the actual work
        }
    }
}
```

Exempel: 2 skrivartrådar kör parallellt

```
public class TwoWriterThreads
{
    private Writer s1 = new Writer("C++ is best!", 1122);
    private Writer s2 = new Writer("Java is better...", 738);

    public TwoWriterThreads()
    {
        s1.start();
        s2.start();
    }

    public void stop() {
        s1.interrupt();
        s2.interrupt();
    }
}
```

Exempel: skrivare med java.util.Timer

```
import java.util.*;
public class Writer2 extends TimerTask
{
    String text;
    int count = 0;

    public Writer2(String text)
    {
        this.text = text;
    }

    public void run() {
        System.out.println(text);
        count++;
    }

    public int getCount() {
        return count;
    }
}
```

forts. skrivare med java.util.Timer

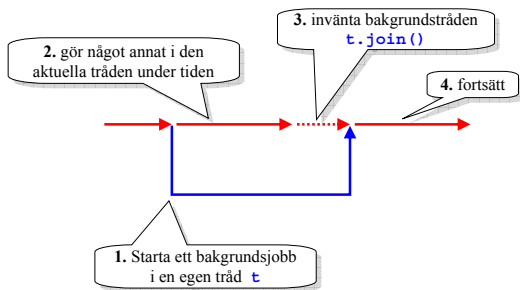
```
public class TwoWriters
{
    private Timer t = new Timer();
    private Writer2 s1 = new Writer2("C++ is best!"),
        s2 = new Writer2("Java is better...");

    public TwoWriters() throws InterruptedException
    {
        // schemalägg de två objekten
        t.schedule(s1, 0, 738);
        t.schedule(s2, 0, 1122);
    }

    public void stop() {
        s1.cancel();
        s2.cancel();
        t.cancel();
        System.out.println("s1: " + s1.getCount() + ", "
            + s2: " + s2.getCount() );
        System.exit(0);
    }
}
```

Metoden `join()` i Thread

`t.join()` Aktuell tråd väntar tills `t` har avslutats



Exempel: bakgrundsjobb med `join()`

```
public class BackgroundJob extends Thread
{
    private int maxSteps = 0;
    private long sum = 0; // Calculate sum = 1+2+3+...
    private long interval;

    public BackgroundJob(long interval, int maxSteps)
    {
        this.maxSteps = maxSteps; this.interval = interval;
    }

    public void run() {
        int i = 1;
        while ( i <= maxSteps && ! Thread.interrupted() ) {
            try { Thread.sleep(interval); }
            catch (InterruptedException e) { break; }
            System.out.println("Background work... " + i++);
            sum += i;
        }
        System.out.println("Background work finished");
    }

    public long getResult() {
        return sum;
    }
}
```

forts. bakgrundsjobb med `join()`

```
public class Main {
    private BackgroundJob backgroundJob = new BackgroundJob(1000,10);

    public Main() {
        backgroundJob.start(); // Start background thread
        foregroundJob(); // Do foreground job in parallel
    }

    public void foregroundJob() {
        int i = 0;
        while ( i <= 20 && ! Thread.interrupted() ) {
            try { Thread.sleep(200); }
            catch (InterruptedException e) { break; }
            System.out.println("Foreground: " + i++); // work, work, work, ...
        }

        // Now wait for background job to finish before continuing foreground job
        try { backgroundJob.join(); }
        catch (InterruptedException e) { }

        System.out.println("Background result: " + backgroundJob.getResult() );
        System.out.println("Foreground work continues...");
        // ...
    }
}
```

Exempel: Job shop

- ◆ Producenter och konsumenter
- ◆ Buffring krävs eftersom trådarna kan arbeta i olika takt
- ◆ Exempel: Simulering av help desk



Synkronisering

- ◆ När flera trådar delar en gemensam resurs krävs **ömsesidig uteslutning** vid access till resursen.

Metodsynkronisering i Java `synchronized`

- ◆ När en **synkroniserad metod** exekveras är **objektet låst** för anrop av alla synkroniserade metoder.
 - Synkroniserade metoder får dock anropas för andra objekt av klassen

Metoderna `notify()` och `wait()`

`notify()`

- ◆ signalerar till andra trådar att objektet ändrats

`wait()`

- ◆ suspenderar tråden och öppnar låset för objektet
- ◆ andra trådar får exekvera synkroniserade metoder
- ◆ tråden väcks när någon tråd anropar `notify` för objektet

SimpleQueue: En synkroniserad köklass

```
public class SimpleQueue<T> {
    private LinkedList<T> queue = new LinkedList<T>();

    public int size() {
        return queue.size();
    }

    public synchronized void put(T obj) {
        queue.add(obj);
        notify();
    }

    public synchronized T take() {
        ... nästa bild
    }
}
```

Metoden SimpleQueue.take()

```
public synchronized T take() {
    while (queue.isEmpty()) {
        try {
            wait();
        }
        catch (InterruptedException e) {
            return null;
        }
    }
    T obj = queue.getFirst();
    queue.removeFirst();
    return obj;
}
```

Exempel: Jobshop

```
public class JobShop
{
    private SimpleQueue helpDeskQueue = new SimpleQueue<String>();
    private JobGenerator jobGenerator = new JobGenerator();
    private ArrayList<Thread> threads = new ArrayList<Thread>();

    public JobShop() throws InterruptedException {
        createAndStartThreads();

        // sleep while the simulation is running
        Thread.sleep(20000);

        stopThreads();
        System.out.println("Jobs left in queue: " + helpDeskQueue.size());
        System.exit(0);
    }

    public void createAndStartThreads() { ... }

    public void stopThreads() { ... }
}
```

Jobshop-metoder

```
public void createAndStartThreads() {
    threads.add(
        new Producer("Lundin", 6237, helpDeskQueue, jobGenerator));
    threads.add(
        new Producer("von Hacht", 2846, helpDeskQueue, jobGenerator));
    threads.add(
        new Producer("Holmer", 1239, helpDeskQueue, jobGenerator));

    threads.add(new Consumer("Erik", 982, helpDeskQueue));
    threads.add(new Consumer("Mikael", 1654, helpDeskQueue));

    for (Thread t : threads)
        t.start();
}

public void stopThreads() {
    for (Thread t : threads)
        t.interrupt();
}
```

Class Producer

```
public class Producer extends Thread {
    private String name;
    private long interval;
    private SimpleQueue<String> queue;
    private JobGenerator jobGenerator;

    public Producer( ... ) { ... }

    public void run() {
        while (!Thread.interrupted()) {
            try {
                Thread.sleep(interval);
            }
            catch (InterruptedException e) {
                break;
            }
            queue.put(name + " : " + jobGenerator.pickRandomJob());
        }
    }
}
```

Class Consumer

```
public class Consumer extends Thread
{
    private String name;
    private long interval;
    private SimpleQueue<String> queue;

    public Consumer( ... ) { ... }

    public void run() {
        while (!Thread.interrupted()) {
            try {
                Thread.sleep(interval);
            }
            catch (InterruptedException e) {
                break;
            }
            System.out.println(name + " received job from "
                               + queue.take());
        }
    }
}
```

Trådar och Swing

- ◆ Program som använder Swing har en **händelsetråd** (*event-dispatching-thread*) som exekverar all kod som förändrar swing-komponenter, t.ex. anropas **actionPerformed** från händelsetråden.
- ◆ Operationer som förändrar en Swing-komponent **efter första uppritningen** skall exekveras av händelsetråden.
 - Sådan kod bör alltså placeras i lyssnarmetoder.
- ◆ Resurskrävande beräkningar som startas från Swing riskerar att "lagga" GUI:t
 - sådana beräkningar kan köras i *bakgrunden* med hjälp av standardklassen **SwingWorker**
- ◆ Var försiktig med att kombinera egna trådar med Swing!