Föreläsning 4

- Trådar och Timers
- Aktiva och passiva komponenter
- Tråd genom arv
- Tråd genom Runnable
- Starta och stoppa aktiva komponenter

JNP: s 53-70

Trådar

I ett program vill man ofta låta flera aktiviteter pågå parallellt. Detta löser man med hjälp av trådar.

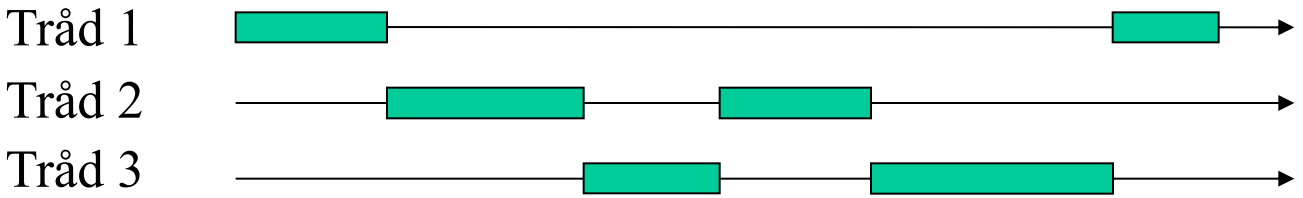
Ett par fall som lämpar sig för att utföras i separat tråd är:

- animeringar eller andra processorkrävande / tidskrävande processer. För att undvika att låsa programmet låter man dessa utföras i en separat tråd. Om man låser upp händelsehanteringen blir programmen riktigt trista.
- aktiviteter som går långsamt varvid processorn inte utnyttjas fullt ut, t.ex. inläsning av bilder över nätet eller inmatning från användare.
- Nätverkskommunikation. Många operationer blockerar den exekverande tråden under en viss tid.

Parallella processer

Ett system med många processorer:

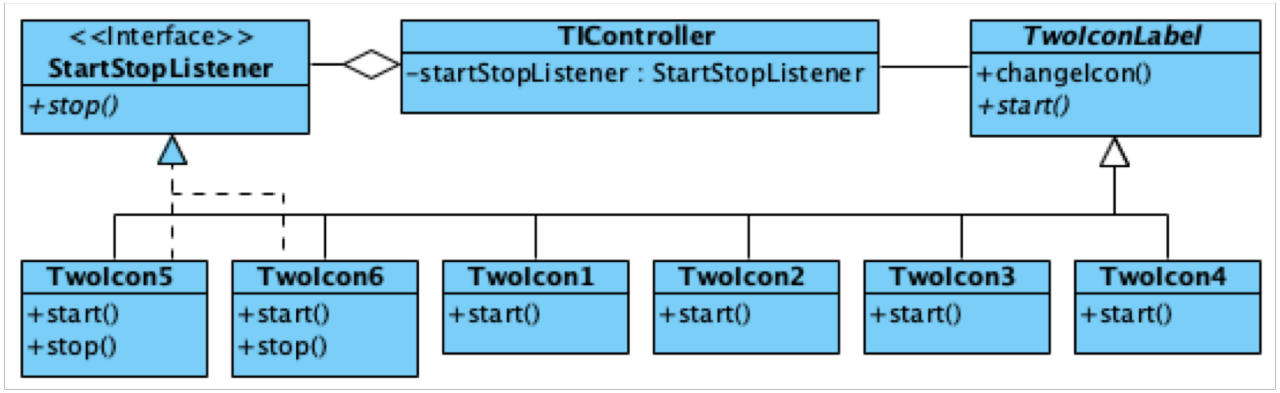
Ett system med en processor:



Aktiva och passiva komponenter

En passiv komponent gör ingenting utan att bli tillfrågad, dvs att en metod i komponenten anropas från ett annat objekt.

En aktiv komponent agerar själv. Den aktiva komponenten innehåller en eller flera trådar.



Timer-klasser

I java finns det ett par Timer-klasser, dvs. objekt som exekverar kod med en separat tråd vid speciella tidpunkter.

Gemensamt för dessa timer-klasser är att den kod som exekveras ska exekveras snabbt. Det kan vara många sekvenser kod som väntar på att exekveras. Långsam exekvering kan medföra att någon eller några av dessa kodsekvenser inte exekveras på avsedd tid.

Ett par Timer-klasser:

- javax.swing.Timer
- java.util.Timer

javax.swing.Timer

I paketet ***javax.swing*** finns en ***Timer***-klass. Den är lämplig att använda om koden som ska exekveras ändrar i det grafiska användargränssnittet. Klassen använder nämligen händelse-tråden för att exekvera koden. Och det är endast denna tråd som ska användas vid arbete med GUI:t (även om vi i många fall skapat GUI:et med main-tråden – se GUILaboration för korrekt kod).

Användandet av *Timer*-klassen liknar användandet av GUI-komponenter.

1. Man skapar ett *Timer*-objekt.
   Timer timer = new Timer(int ms, ActionListener list);
   Det andra argumentet anger den klass som implementerar *ActionListener* och som ska anropas av timern.
2. Man låter en klass implementera *ActionListener* och skriver metoden *actionPerformed*. Koden i *actionPerformed* exekveras med det intervall *ms* i konstruktorn anger.

```
public void actionPerformed( ActionEvent e ) {
    // kod som ska exekveras
}
```

javax.swing.Timer

I paketet ***javax.swing*** finns en ***Timer***-klass. Den är lämplig att använda om koden som ska exekveras ändrar i det grafiska användargränssnittet. Klassen använder nämligen händelse-tråden för att exekvera koden. Och det är endast denna tråd som ska användas vid arbete med GUI:t (även om vi i många fall skapat GUI:et med main-tråden – se GUILaboration för korrekt kod).

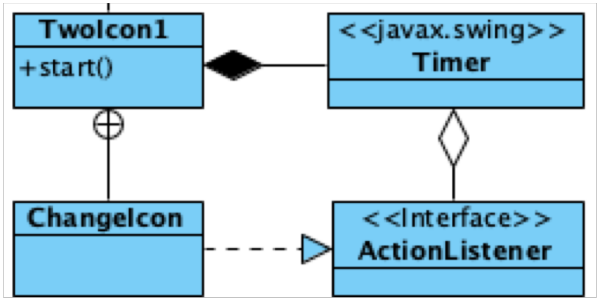
Användandet av *Timer*-klassen liknar användandet av GUI-komponenter.

3. Man startar timern
   timer.start();
   varvid actionPerformed-metoden anropas upprepat med ms millisekunder mellan anropen.
4. Man avslutar timern med
   timer.stop();
   Därefter startas timern med *start* på nytt.

TwoIconLabel.java

TwoIcon1.java

TIController.java



java.util.Timer

I paketet ***java.util*** finns en ***Timer***-klass. Den använder en separat tråd för att exekvera TimerTask-objekt, dvs run-metoden i klasser som ärver klassen TimerTask.

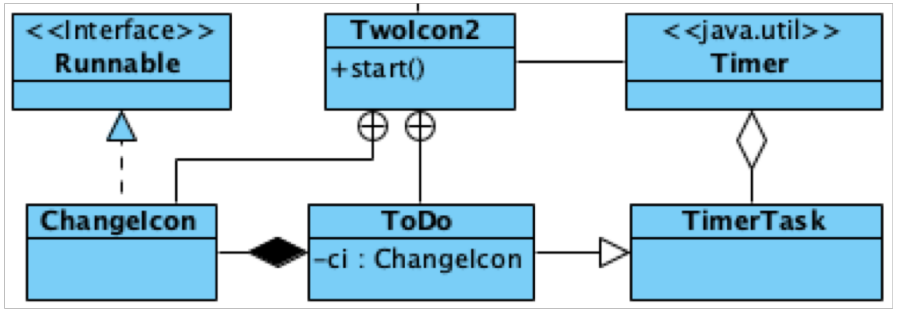
För att använda *Timer*-klassen gör man så här:

1. Man skapar ett *Timer*-objekt.
   ```
   Timer timer = new Timer();
   ```
2. Man skriver en klass som ärver TimerTask och överskuggar run-metoden:
   ```
   public class ToDo extends TimerTask {
       public void run(  ) {
           // kod som ska exekveras
       }
   }
   ```
3. Man registrerar att koden i run ska exekveras periodiskt:
   ```
   timer.schedule( new ToDo(), 2000, 8000 );   // Koden i run anropas efter 2 sekunder
                                               // och sedan var 8:e sekund.
   ```
   eller en gång:
   ```
   timer.schedule( new ToDo(), 2000 );         // Koden i run anropas efter 2 sek
   ```
   Det går utmärkt att schemalägga många olika aktiviter i samma Timer.
4. Man avslutar timern med:
   ```
   timer.cancel();
   ```

TwoIcon2 | TIController

java.util.Timer

Skillnaderna mot förra systemet är val av Timer-klass, implementation av Timertask och att endast händelsetråden får anropa UI-metoder.



Tråd genom arv av Thread

I java skapar man nya trådar med klassen **Thread**. Man kan använda ett av följande sätt:

1.Genom att **ärva klassen Thread**. Exekvering av tråden sker med anrop till metoden **start()** och metoden **run()** ska överskuggas.

```
public class ClassT extends Thread {
   :
   public void run() {
      // kod som utgör själva tråden
   }
}
-------------------------------------------------------
// Kod som startar tråden
ClassT cls = new ClassT();                    // Klassen ClassT är en tråd
cls.start();                                  // run-metoden kommer att börja
exekveras
```

Counter1 | TestCounter

Tråd genom implementering av Runnable

2. En klass som implementera gränssnittet **Runnable**. Och ett objekt av typen Thread. Vilken använder en instans av Runnable-implementeringen.

```
public interface Runnable {
   public abstract void run();
}

public class ClassR implements Runnable {
   :
   public void run() {
      // kod som utgör själva tråden
   }
}

// Kod som startar tråden
Thread thread = new Thread(new ClassR());
thread.start();
```

En Runnable-implementering är argument vid instansiering av Thread.

```
Thread thread = new Thread(new ClassR());
```

Tråden startas med anrop till **start()** (**thread.start()** ovan) och metoden **run()** ska finnas i klassen som implementerar Runnable.

Counter2 | TestCounter

En klass har en tråd – inre klass

En klass som ärver en annan klass och dessutom ska innehålla en tråd kan använda någon av följande tekniker:

1. Ärva Thread i en **inre klass**. Man har en instansvariabel till den inre klassen i klassen (som är en tråd).

```
public class ClassIT extends JFrame {
  private InnerClass thread = new InnerClass();

  public void start() {  // en start-metod kan vara bra
     thread.start();
  }

  private class InnerClass extends Thread {
    public void run() {
      // kod som utgör själva tråden
    }
  }
}
```

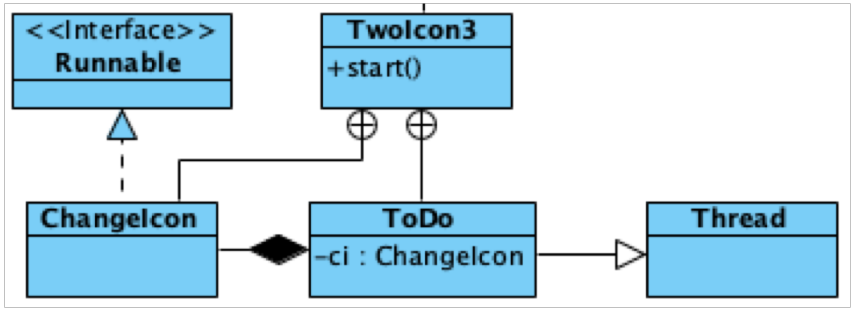
Tråden startas med anrop till **start()** (**thread.start()** ovan) och metoden **run()** ska finnas i den inre klassen.
Med denna teknik kan en klass innehålla flera trådar med olika uppgifter (flera inre klasser).

Counter3 | TestCounter

En klass har en tråd – inre klass

I detta system har **TwoIcon3** två inre klasser:

- **ChangeIcon** vilken implementerar **Runnable**. Vid anrop av *run*-metoden anropas i sin tur *changeIcon*-metoden i **TwoIcon3**-instansen.
- **ToDo** vilken ärver **Thread**. **ToDo** har en referens av typen **ChangeIcon** och placerar en instans av **ChangeIcon** i händelsetrådens buffert med jämna mellanrum.



TwoIcon3

En klass har en tråd – inre klass

2. Implementera Runnable i en inre klassen. Man har en instansvariabel som är en tråd och som använder Runnable-implementeringen.

```
public class ClassIR extends JFrame {
  private Thread thread = new Thread(new Activity);

  public void start() {  // en start-metod kan vara bra
     thread.start();
  }

  private class Activity implements Runnable {
    public void run() {
      // kod som utgör själva tråden
    }
  }
}
```

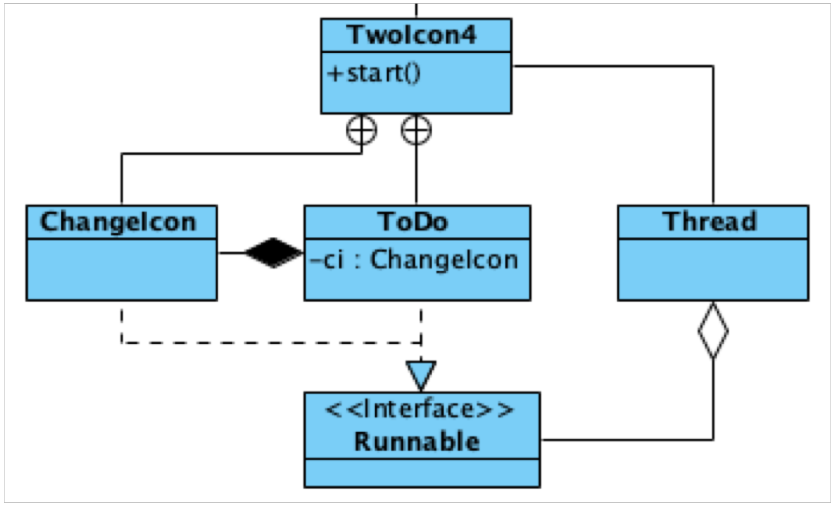
Tråden startas med anrop till **start()** (**thread.start()** ovan) och metoden **run()** ska finnas i den inre klassen.

Counter4 | TestCounter

En klass har en tråd – inre klass

I detta system har **TwoIcon4** två inre klasser:

- **ChangeIcon** vilken implementerar **Runnable**. Vid anrop av *run*-metoden anropas i sin tur *changeIcon*-metoden i **TwoIcon4**-instansen.
- **ToDo** vilken implementerar **Runnable**. *run*-metoden innehåller koden som exekveras av Thread-instansen. **ToDo** har en referens av typen **ChangeIcon** och i *run*-metoden placeras en instans av **ChangeIcon** i händelsetrådens buffert med jämna mellanrum.



TwoIcon4

Klassen Thread - metoder

start()	Startar exekveringen av run
interrupt()	Avbryter exekveringen av run
setPriority(int prioritet)	Ändrar prioriteten för tråden
join()	Programmet väntar tills tråden har avslutats
yield()	Tråden pausar och andra trådar (med samma prioritet) kan exekveras.
run()	Metod som ska överskuggas

Klassmetoder

interrupted()	Returnerar true om tråden avbrutits
sleep(int ms)	Tråden väntar i ms millisekunder

Starta/Stoppa, javax.swing.Timer

Klassen **TwoIcon5** implementerar interfacet **StartStopListener** och innehåller därmed metoderna ***start()*** respektive ***stop()***.

TIController kontrollerar om **TwoIconLabel**-subklassen implementerar **StartStopListener**. Om så är fallet aktiveras Start- respektive Stop-knappen.

start startar bildväxling mellan Icon-implementeringarna. ***stop*** avslutar bildväxlingen.

Med ***javax.swing.Timer*** är det enkelt att implementera denna funktionalitet. En sådan timer kan startas/stoppas flera gånger genom anrop till klassens *start*-/*stop*-metod. Och det går bra att starta en redan startad timer.

Skillnaden mellan klasserna **TwoIcon1** och **TwoIcon5** är att **TwoIcon5** implementerar **StartStopListener** (och att timern inte startas i konstruktorn).

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

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

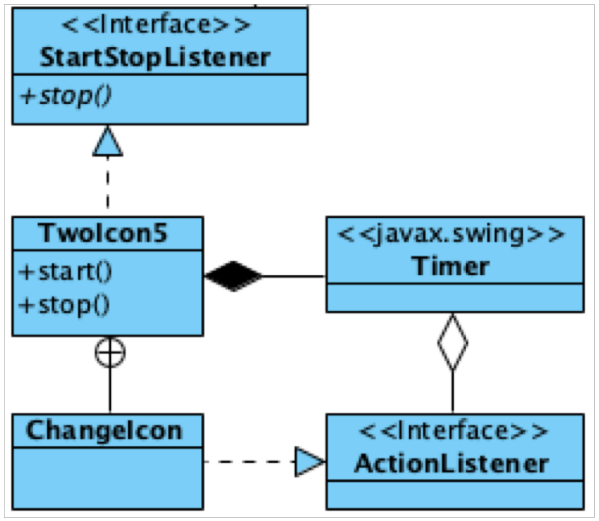
StartStopListener

TwoIcon5

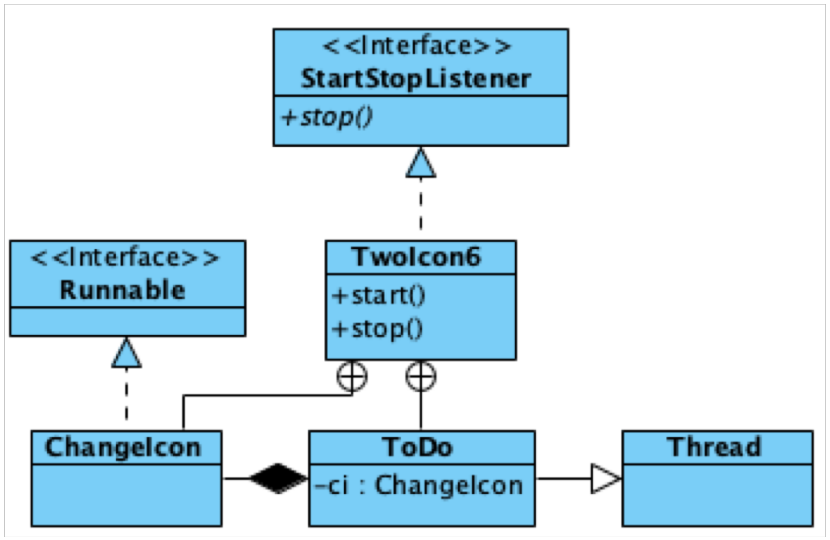
TIController

Starta/Stoppa, javax.swing.Timer

Designen över systemet med **TwoIcon5** är väldigt likt **TwoIcon1**-systemet.



Designen över systemet med **TwoIcon6** är väldigt likt **TwoIcon3**-systemet.



Starta/Stoppa, java.util.Timer

Med ***java.util.Timer*** är det något mer komplicerat att implementera start/stop-funktionalitet. En sådan Timer kan nämligen endast startas en gång.

Det innebär att man måste skapa en ny timer varje gång start anropas. Anropar användaren start flera gånger i rad så ska endast en timer vara verksam. På samma sätt ska det endast gå att stoppa en timer en gång.

```
public void start() {
   if(timer == null) {
      timer = new Timer();
      timer.schedule(new ToDo(),0,delay);;
   }
}

public void stop() {
   if(timer!=null) {
      timer.cancel();
      timer = null;
   }
}
```

Starta/Stoppa, Thread

Klassen **TwoIcon6** är identisk med **TwoIcon3** bortsett från att:

- **StartStopListener** implementeras
- Instansvariabeln *thread* är *null* från början och det startas ingen tråd i konstruktorn.

En avbruten tråd kan inte startas på nytt. Man måste alltså skapa en ny tråd i ***start***-metoden. Men flera trådar ska inte kunna startas:

```
public void start() {
   if(thread==null) { // finns redan aktiv tråd? Om inte så:
      thread = new ToDo();
      thread.start();
   }
}
```

TwoIcon6

TIController

Starta/Stoppa, Thread

stop-metoden måste utformas så att tråden avslutas (*run*-metoden avslutas). Och dessutom måste *thread* ges värdet *null* så att start-metoden startar en ny tråd.

Metoden *interrupt()* gör en av två saker:

1. Är tråden normalt exekverande så sätts en flagga till *true*. Denna flagga kan kontrolleras av klassmetoden *Thread.interrupted()*.
2. Är tråden avbruten (t.ex. av sleep) så kastas ett *InterruptedException*

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

private class ToDo extends Thread {
   private Runnable changeIcon = new ChangeIcon();

   public void run() {
      while( !Thread.interrupted() ) {     // 1. Thread.interrupted() är true om interrupt() anropats
         try {
            Thread.sleep(delay);           // 2. InterruptedException kastas vid anrop till interrupt()
         } catch(InterruptedException e) {
            break;
         }
         SwingUtilities.invokeLater( changeIcon );
      }
      thread = null;
   }
}
```

TwoIcon6

TIController