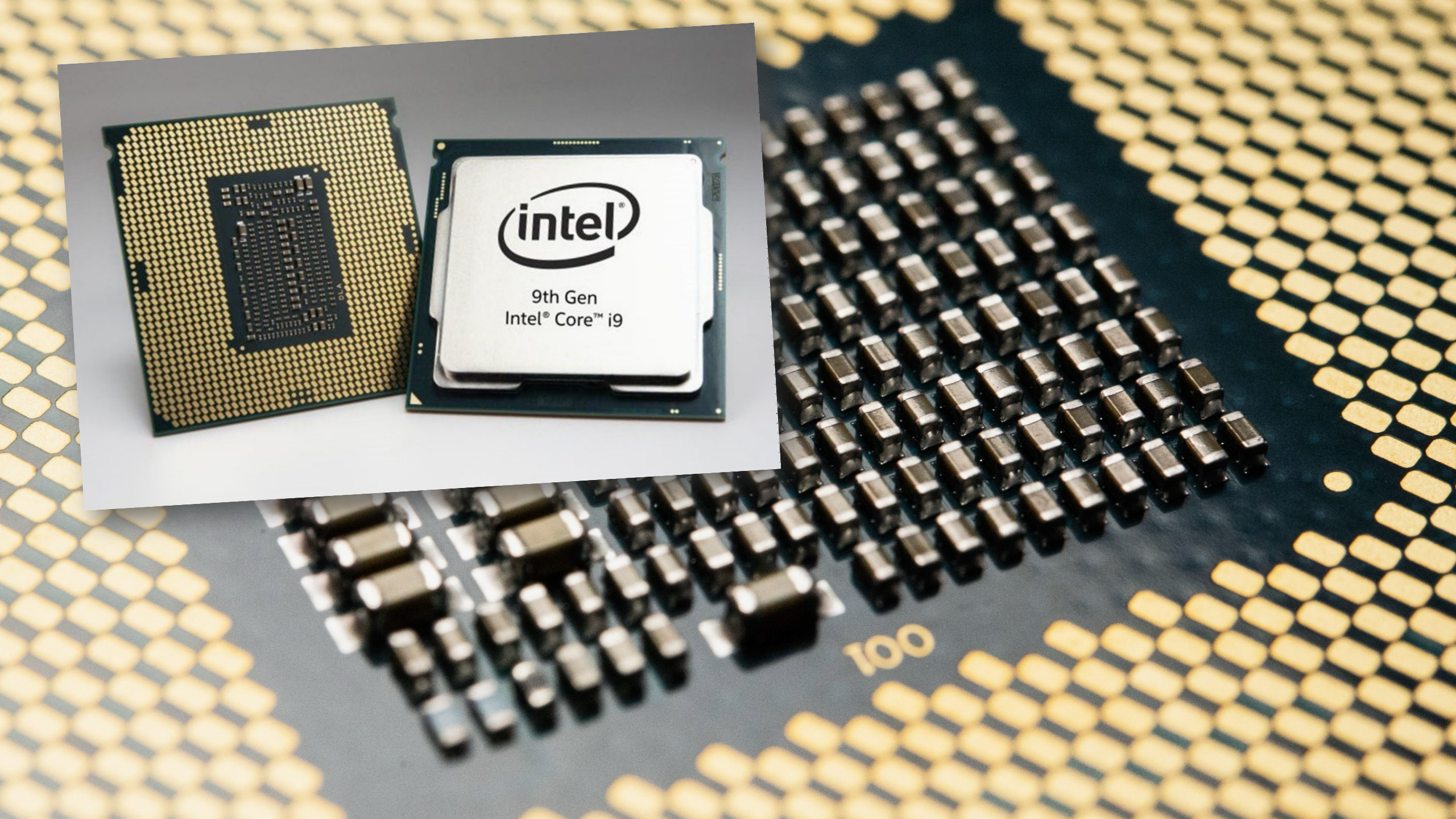
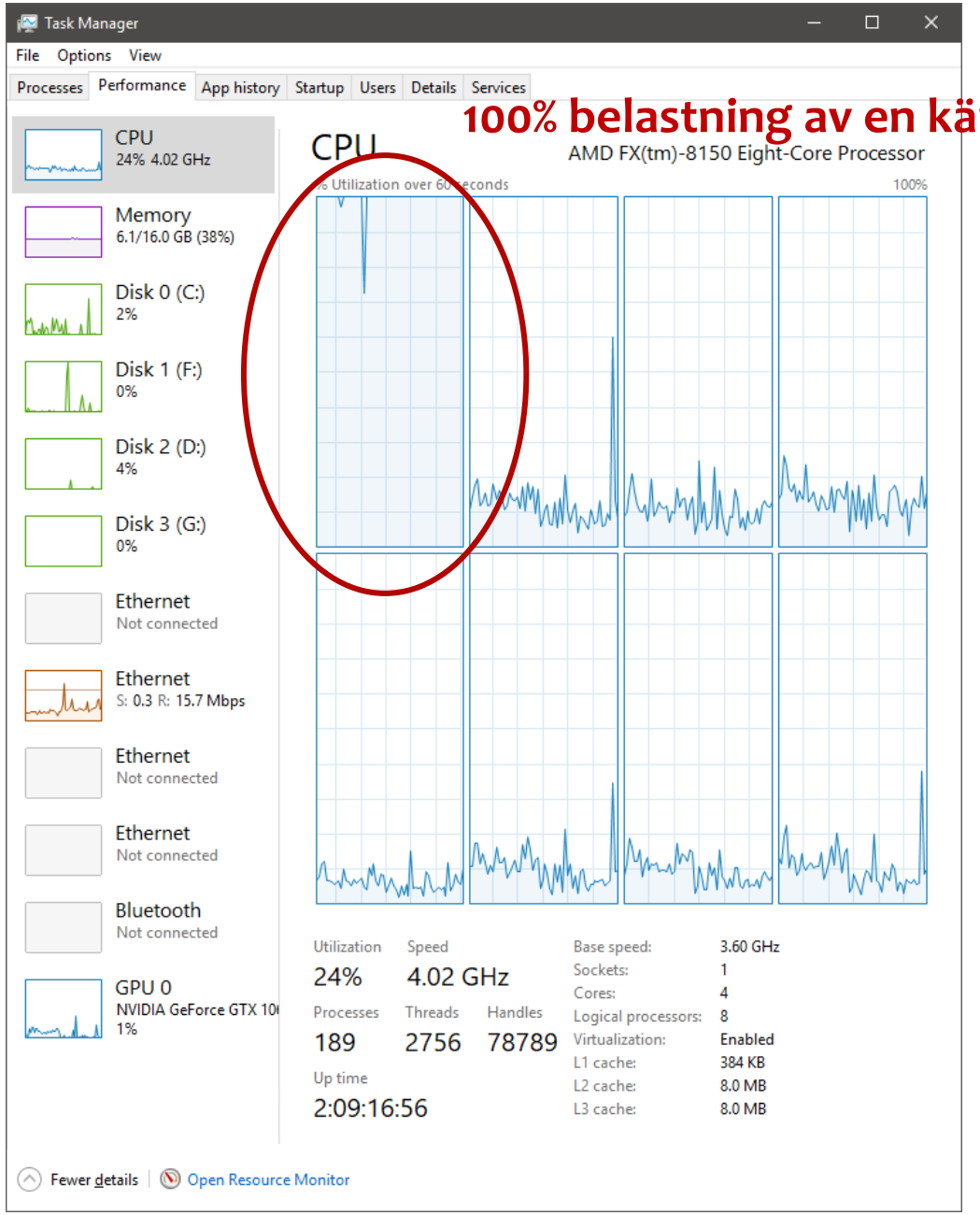
- DA343A -
Objektorienterad programutveckling, trådar och datakommunikation

Föreläsning 7

Johan Holmgren (Utvecklad av Fabian Lorig)


intel
9th Gen
Intel® Core™ i9



100% belastning av en kärna

Trådar – varför och när

Operativsystemet använder sig av **multitrådning** (*multithreading*) för att kunna utföra olika aktiviteter parallellt och för att kunna utnyttja alla resurser

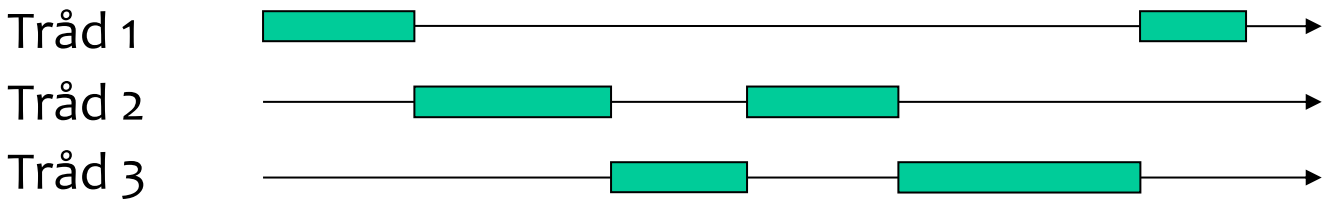
I ett program vill man ofta låta flera aktiviteter pågå parallellt. Detta löser man med hjälp av **trådar**.

Ett par fall som lämpar sig för att utföras i separata tråder är:

- **Animeringar** eller andra **processorkrävande / tidskrävande processer**.
- **Aktiviteter som går långsamt**
- **Nätverkskommunikation**

Parallella processer

Ett system med en processor:



Ett system med flera processorer:

TIMER

Timer-klasser

- Enkelt sätt att **planera** en aktivitet i bakgrunden
- I Java finns det några Timer-klasser, dvs. objekt som exekverar kod i en separat tråd vid given tidpunkt
- 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.

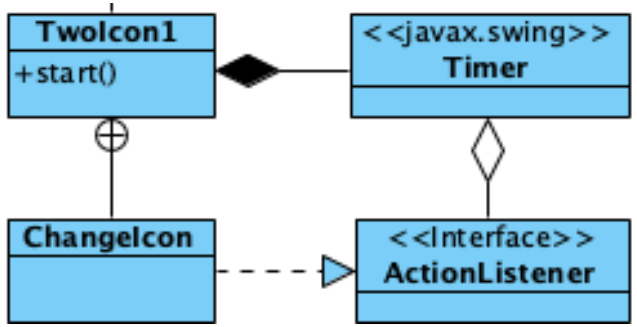
Exempel på Timer-klasser:

- *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 händelse-tråden för att exekvera koden och det är endast denna tråd som ska användas vid arbete med GUI:t

1. Skapa ett *Timer*-objekt.
2. Låt en klass implementera *ActionListener* och skriv metoden *actionPerformed*
3. Starta timern
4. Avsluta timern



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

javax.swing.Timer

TwoIconLabel.java

TIController.java

TwoIcon1.java

1. **Skapa ett *Timer*-objekt.**

```
Timer timer = new Timer( int ms, ActionListener list );
```

Det andra argumentet anger den klassen som implementerar *ActionListener* och som ska anropas av timern.

2. Låt en klass **implementera *ActionListener*** och skriv metoden ***actionPerformed***.

Koden i *actionPerformed* exekveras med det intervall i *ms* som angetts i konstruktorn.

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

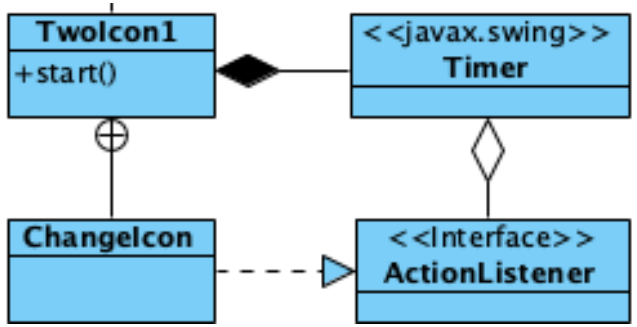
3. **Starta timern** varvid actionPerformed-metoden anropas upprepat med *ms* millisekunder mellan anropen.

```
timer.start();
```

4. **Avsluta timern**

```
timer.stop();
```

Därefter startas timern med *start* på nytt.



javax.swing.Timer

- Alla timer ”väntar” i samma tråd
  - Som skapas av den första timern
- Exekvering av ActionHandler i en annan tråd
- Aktion måste utföras snabbt så att GUI:t är responsivt
- Två fördelar med javax.swing.Timer
  - Liknar användandet av GUI-komponenter
  - Alla delar samma tråd (som bl.a. gör att markören blinkar)

```
int delay = 1000; //milliseconds
ActionListener taskPerformer = new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        //...Perform a task...
    }
};

new Timer(delay, taskPerformer).start();
```

java.util.Timer

TIController.java

TwoIcon2.java

I paketet ***java.util*** finns en ***Timer***-klass som 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. **Skapa ett *Timer*-objekt.**

```
Timer timer = new Timer();
```

2. Skriv en klass som **ärver TimerTask** och **överskuggar *run*-metoden**:

```
public class ToDo extends TimerTask {
    public void run(  ) {
            // kod som ska exekveras
    }
}
```

3. **Registrera** 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. **Avsluta timern** med:

```
timer.cancel();
```

java.util.Timer

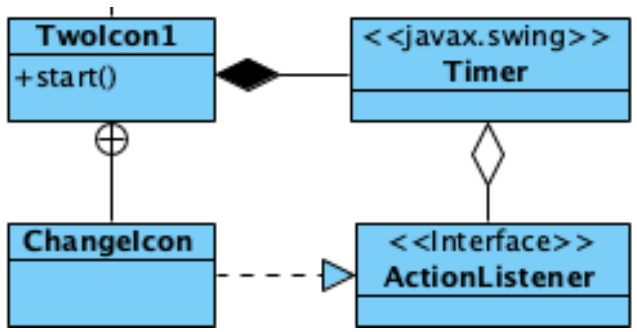
```
public static void main(String[] args) {
    Timer timer = new Timer();
    TimerTask task = new Helper();

    timer.schedule(task, delay: 2000, period: 5000);
}

static class Helper extends TimerTask
{
    public void run()
    {
        System.out.println("!");
    }
}
```

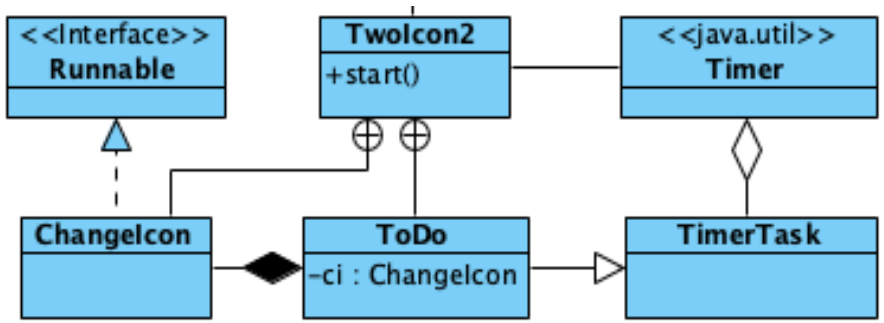
Timer

javax.swing.Timer

java.util.Timer

Skillnaderna mellan systemen är val av Timer-klass,
implementation av Timertask
och att endast händelsetråden får anropa UI-metoder.





TRÅDAR

Trådar – två sätt att skapa

Ärva (extends) Thread

Genom att ärva klassen Thread.

```
public class ClassT extends Thread {
}

Thread t = new Thread();
t.start();
```

Implementera (implements) Runnable

Genom att implementera gränssnittet Runnable.

```
public class ClassR implements Runnable {
}

Thread t = new Thread( new ClassR() )
t.start();
```

Counter1.java

TestCounter.java

Tråd genom arv av Thread

- I java skapas nya trådar t.ex. genom att **ärva klassen Thread**.
- Exekvering av tråden sker med anrop till metoden **start()** och metoden **run()** ska överskuggas
- ***Obs!* run()** ska inte anropas direkt!
- När run() metoden terminerar avslutar tråden.

```
public class ThreadClass extends Thread {

   public void run() {
      // kod som utgör själva tråden
   }
}


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

Counter2.java

TestCounter.java

Tråd genom implementering av Runnable

Kombination av en klass som implementerar gränssnittet **Runnable** och ett objekt av typen **Thread** som använder en instans av Runnable-implementeringen.

En Runnable-implementering är argument vid instansiering av Thread.

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

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

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

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

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

Thread vs. Runnable

- I Java kan en klass ärva **endast en** klass (*single inheritance*)
- Gränssnitt ger bättre separation mellan egen kod och implementeringen av Thread
- Flexibilitet: Runnable kan men måste inte exekveras som tråd
- En tråd kan inte startas om när den har terminerat
- Att ärva från en klass betyder att den ska förändras, modifieras eller förbättras. Om det inte behövs räcker det med Runnable

En klass har en tråd – inre klass

Counter3.java

TestCounter.java

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

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

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

Med denna teknik kan en klass innehålla flera trådar med olika uppgifter (flera inre klasser).

```
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
      }
   }
}
```

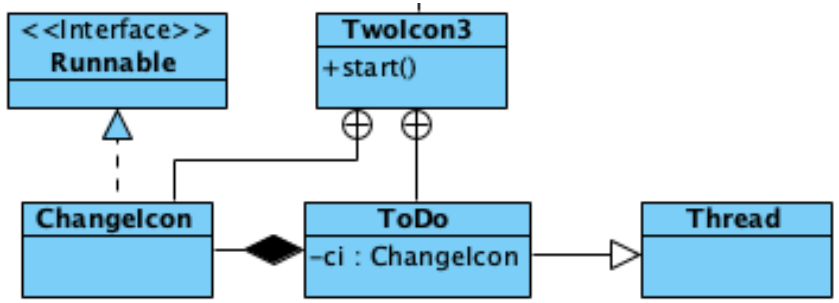
TwoIcon3.java

En klass har en tråd – inre klass

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

- ***ChangeIcon*** som 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.



MALMÖ UNIVERSITY

En klass har en tråd – inre klass

Counter4.java

TestCounter.java

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

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

```
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
      }
   }
}
```

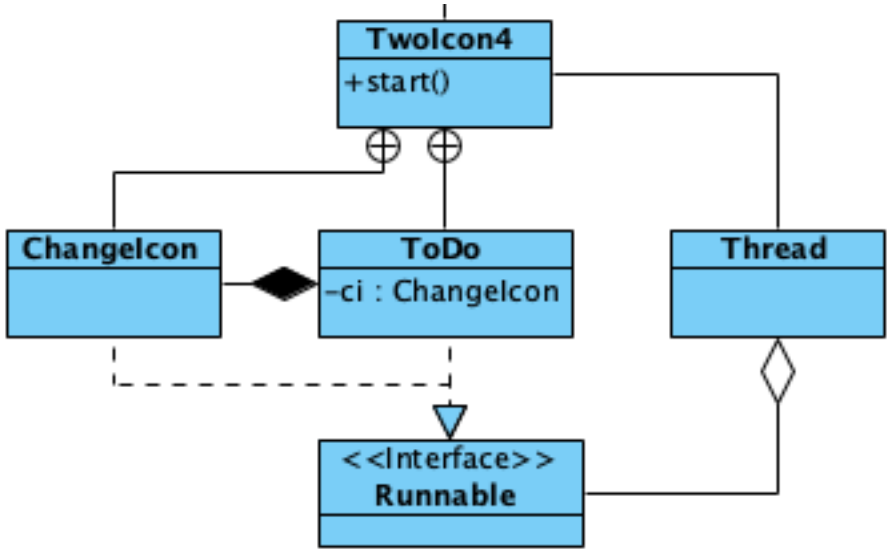
TwoIcon4.java

En klass har en tråd – inre klass

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

- **ChangeIcon** som 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.



MALMÖ UNIVERSITY

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 (ska inte anropas direkt!)

Klassmetoder

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

HANTERING AV TRÅDAR OCH TIMER

Starta och stoppa en javax.swing.Timer

StartStopListener.java

TwoIcon5.java

TIController.java

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 och ***stop*** avslutar bildväxlingen.

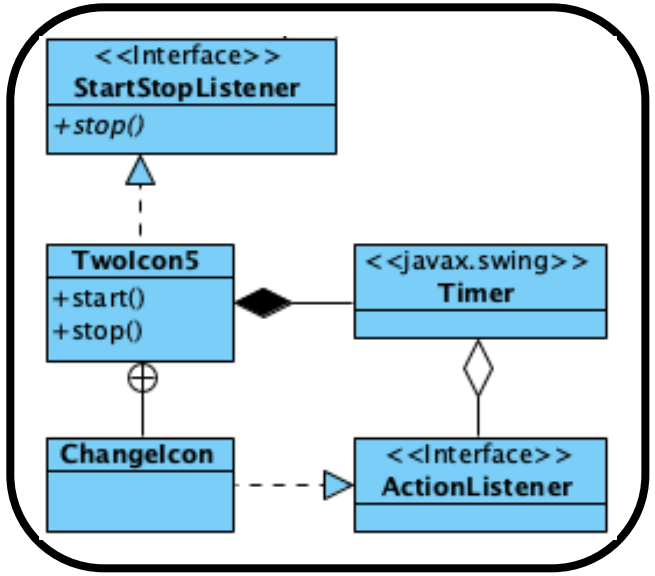
```
public void start(){
    timer.start();
}

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

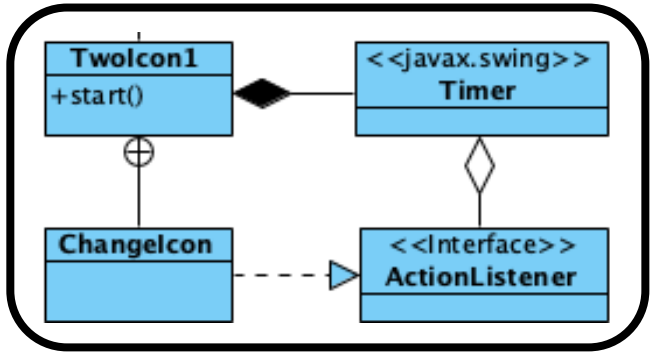
Med ***javax.swing.Timer*** är det enkelt att implementera denna funktionalitet.
En sådan timer kan startas och stoppas flera gånger genom anrop till klassens *start-*/*stop*-metod.

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

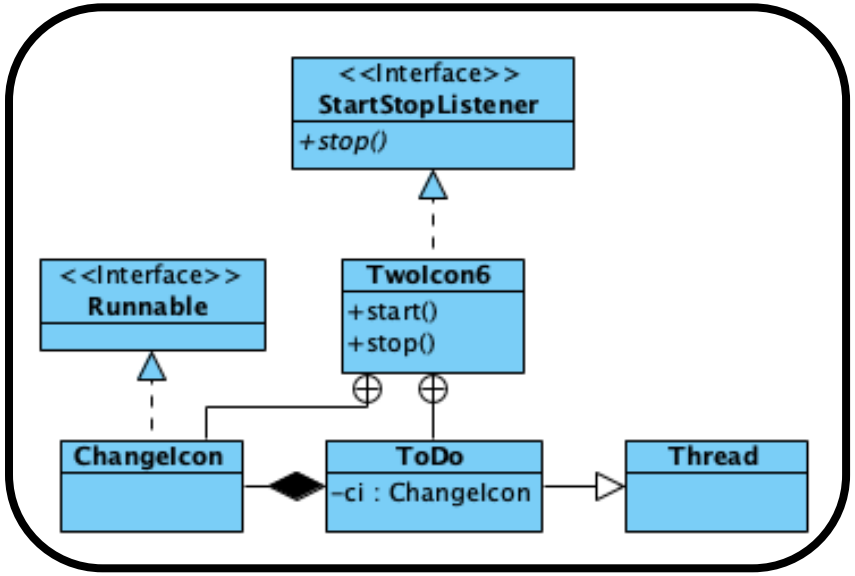
Starta och stoppa en javax.swing.Timer

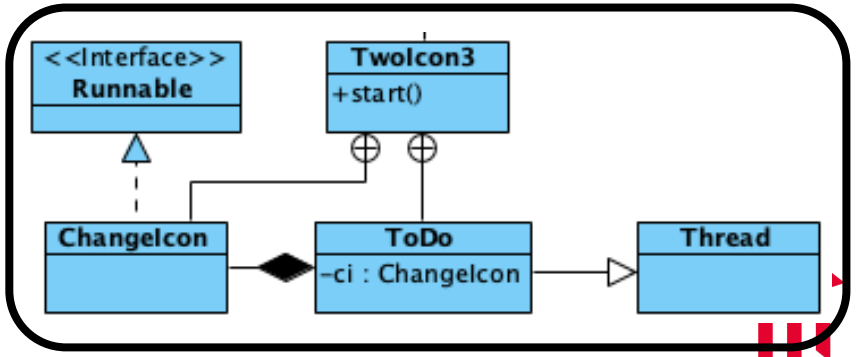
Designen över systemet med TwoIcon5 är väldigt likt TwoIcon1-systemet.





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





Starta och stoppa en 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 och stoppa en tråd

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 skapa en ny tråd i *start*-metoden. Men flera trådar ska inte kunna startas:

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

TwoIcon6.java

TIController.java

Starta och stoppa en tråd

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;
    }
}
```