Multithreading und Animationen in Java

Manuel Rädle

Was ist eine Animation

- Schnell aufeinanderfolgende Anzeige einer Sequenz von Einzelbildern, Texten oder Objekten
- Die Bildfolge erscheint als zusammenhängende Bewegung (aufgrund der Trägheit des Auges)

Anwendungsbereiche von Animationen

- Präsentationen
- Visualisierungen (z.B. von math. Zusammenhängen, Algorithmen, etc.)
- Simulationen
- Filmische Darstellungen

Was sollte man bei Animationen beachten

- Für die Anzeige der Einzelbilder muss das richtige **Timing** gewählt werden.

  Zu wenig Bilder je Zeiteinheit: Ergebnis erscheint ruckelig.

  Zu viele Bilder je Zeiteinheit: paint Events gehen verloren
- Wird der paint-/repaint-Mechanismus von Java verwendet, wird die Darstellung durch ein **starkes Flackern** gestört. Dies muss verhindert werden. -> Double Buffering, Clipping
- Die Darstellung der Einzelbilder darf nicht die **Interaktivität** des Programms beeinträchtigen (Die Nachrichtenwarteschlange darf nicht blockiert werden). -> Multithreading

Einfache Animation Teil 1

Das Grundprinzip einer Animation besteht darin, in einer Schleife die Methode **repaint** wiederholt aufzurufen und in paint jedesmal etwas anderes zeichnen.

```
public void start()
{
   int i = 0;
   while(true)
   {
        repaint();  //diese ruft paint() auf
   }
}
public void paint(Graphics g)
{
   ++i;
   g.setColor(Color.blue);
   g.drawString(„Counter= " + i, 20, 45);
}
```

Probleme: Timing, Starkes Flackern und Interaktivität des Programms nicht gewährleistet (bei Applets wird nichts angezeigt, aufgrund der zu schnell aufeinanderfolgenden Iterationen.)

Einfache Animation Teil 2

- Schleife muss für eine gewisse Zeit angehalten werden, so dass in dieser Zeit die Grafikausgabe sichtbar werden kann
- Lösung: Statische Methode `Thread.sleep(long millis)`

```
public void start()
{
   while(true)
   {
        repaint();
        try
        {
            Thread.sleep(1000);
        }
        catch (InterruptedException e) {}
   }
}
```

- Nachteil: Die Methode `sleep()` hält den ganzen Thread (Applet) an. Interaktivität des Programms dadurch nicht gewährleistet (bei Applets wird wiederum nichts angezeigt, da der sleep Aufruf das ganze Applet anhält).

Einfache Animation Teil 3

- Lösung:

  Erzeugung eines separaten Threads zur Entkopplung der Animation vom Hauptprogramm

- Multithreading

Warum Multithreading?

- Multithreading bietet die Möglichkeit, zwei oder mehr Vorgänge gleichzeitig auszuführen (Kontrollstrukturen innerhalb eines Programms entkoppeln)
- Erleichert dadurch die Implementierung grafischer Anwendungen
  - Simulationen komplexer Abläufe sind oft nebenläufig
  - Bedienbarkeit von Dialoganwendungen werden verbessert, indem rechenintensive Anwendungen bzw. Berechnungen im Hintergrund laufen
  - Druckausgaben können in einem separaten Thread ausgeführt werden

Threads erzeugen

- Eigene Klasse von Thread ableiten (java.lang Package).

  Man erbt dadurch eine Reihe von Methoden zur Steuerung der Threadausführung.
- run() Methode überschreiben (Diese bestimmt die Lebensdauer des Threads).
- Die Methode start() aufrufen, um den Thread zu starten.
- Nach Aufruf von start() wird die Ausführung an die Methode run() übertragen und start() wird beendet, so dass der Aufrufer im Programm fortfahren kann

Threads erzeugen

Zu Beachten ist:

- run() nie direkt aufrufen, dies bewirkt ein normaler Methodenaufruf und kehrt erst nach Beendigung dieser Methode zum Aufrufer zurück
- Programme die Threads erzeugen, werden erst nach Beendigung des letzten Threads beendet und nicht nach Beendigung der main() Methode

Beispiel

```
class MyThread extends Thread
{
  public void run()
  {
    int i = 0;
    while(i<5000) {System.out.println(i++);}
  }
}
public class ThreadDemo
{
  public static void main(String[] szArgs)
  {
    MyThread t = new MyThread();
    t.start();
  }
}
```

Abbrechen eines Threads

- Endet mit dem Ende seiner run() Methode
- Thread von außen beenden:

  stop() Methode aufrufen

  Vorsicht !

  Ab JDK 1.2 ist stop() deprecated d.h. sie sollte nicht mehr verwendet werden, da nicht vorhersehbar ist, an welcher Stelle der Thread unterbrochen wird (Vorsicht bei wichtigen Transaktionen, kritische Bereichen)

Abbrechen eines Threads

Falsches Beispiel:

```
class MyThread extends Thread
{
  public void run()
  {
     int i = 0;
     while(i<5000) {System.out.println(i++);}
  }
}
public class ThreadDemo
{
  public static void main(String[] szArgs)
  {
     MyThread t = new MyThread();
     t.start();
     try(Thread.sleep(2000)}catch(InterruptedException e){}
     t.stop();
  }
}
```

Abbrechen eines Threads

Richtige Vorgehensweisen:

- Der Thread reagiert selbst auf Unterbrechungsanforderungen (Membervariable canceled einführen)
- Methoden von Thread benutzen
  - public void interrupt()
    Aufruf signalisiert Unterbrechungs-Anforderung
  - public void isInterrupted()
    Thread kann feststellen, ob abgebrochen werden soll

Abbrechen eines Threads

```
class MyThread extends Thread
{
  public void run()
  {
     int i = 0;
     while(true)
     {
       if(isInterrupted() break;
       System.out.println(i++);
     }
  }
}
public class ThreadDemo
{
  public static void main(String[] szArgs)
  {
     MyThread t = new MyThread();
     t.start();
     try(Thread.sleep(2000)}catch(InterruptedException e){}
     t.interrupt();
```

Das Interface Runnable

- Wird benötigt, um eine Klasse threadfähig zu machen, die bereits von einer anderen Klasse als von Thread abgeleitet ist (Keine Mehrfachvererbung in Java!).
- Klasse muss das Interface Runnable implementieren
- run() Methode implementieren

Beispiel

```
class MyClass extends Class2 implements Runnable
{
  public void run()
  {
     int i = 0;
     while(true)
     {
             if(isInterrupted() break;
             System.out.println(i++);
     }
  }
}
public class ThreadDemo
{
  public static void main(String[] szArgs)
  {
     MyClass Test = new MyClass();
     Thread t = new Thread(Test);
     t.start();
     try(Thread.sleep(2000)}catch(InterruptedException e){}
     t.interrupt();
```

Synchronisation

- Die Kommunikation zweier Threads erfolgt in Java auf der Basis gemeinsamer Variablen.
- Zur Synchronisation stellt Java das Konzept des Monitors zur Verfügung. Kritische Abschnitte werden gekapselt um so den Zugriff auf gemeinsam benutzte Datenstrukturen zu koordinieren.

Beispiel ohne Synchronisation

```
class Synchro1 extends Thread
{
  static int count = 0;
  public void run()
  {
     while(count < 500)
     {  int i = count; i++; count = i;
        System.out.println(i++);
     }
  }
  public static void main(String[] szArgs)
  {
      Thread t1 = new Synchro1();
      Thread t2 = new Synchro1();
      t1.start();
      t2.start();
      while(t1.isAlive() && t2.isAlive()) {}
      System.out.println("Threads sind beendet");
  }
```

Die Ausgabe des Programms

0 1 2 3 4 5 6 **7** 8 9 **7** 10 11 12 13 14 15

16 17 18 **19** **21** 22 23 24 25 26 27 **20**

Beispiel mit Synchronisation

```
class Synchro1 extends Thread
{
    static int count = 0;
    public void run()
    {
          while(count < 500)
          {
            synchronized(getClass())
            {
                    int i = count;
                    i++;
                    count = i;
                    System.out.println(i++);
            }
          }
    }
    public static void main(String[] szArgs)
    {
           Thread t1 = new Synchro1();
           Thread t2 = new Synchro1();
           t1.start();
           t2.start();
           while(t1.isAlive() && t2.isAlive()) {}
           System.out.println("Threads sind beendet");
    }
```

Synchronisation von Methoden

```
public void run()
{
   while(count < 500)
   {
        tuWas();
   }
}

public static synchronized void tuWas()
{
   System.out.println(i);
}
```

Synchronisation mit wait() und notify()

- Zusätzlich zum Monitorkonzept eingesetzt
- Objekte besitzen neben der Sperre auch noch eine Warteliste (0 – X Threads/Prozesse)
- wait() fügt ein Prozess der Warteliste hinzu
- notify() gibt ein Prozess wieder frei
- wait() und notify() Aufrufe nur innerhalb eines synchronized Blockes zulässig
- Sind für elementare Synchronisations-aufgaben geeignet

  (geeignet für die Steuerung der zeitlichen Abläufe und nicht auf die Kommunikation)

Aufruf von wait()

- Bereits gewährte Sperre wird temporär zurückgenommen
- Prozess kommt in die Warteschleife des Objektes
- Dadurch wird er unterbrochen und im Scheuduler als wartend markiert

Aufruf von notify()

- Ein beliebiger Prozess wird aus der Warteliste des Objektes entfernt
- Stellt die temporär aufgehobene Sperre wieder her
- Führt den Prozess aus der Warteliste wieder dem normalen Scheuduling zu

Beispiel

```
class Producer extends Thread
{
    private Vector v;
    public Producer(Vector v)
    { this.v = v;  }

    public void run()
    {
        String s;
        while (true)
        {
            synchronized (v)
            {
                s = "Wert "+Math.random();
                v.addElement(s);
                System.out.println("Produzent erzeugte "+s);
                v.notify();
            }
            try
            {
                Thread.sleep((int)(100*Math.random()));
            }
            catch (InterruptedException e)
            {
                //nichts
            }
```

```
class Consumer extends Thread
{
    private Vector v;
    public Consumer(Vector v)
    { this.v = v; }

    public void run()
    {
        while (true)
        {
            synchronized (v)
            {
                if (v.size() < 1)
                {
                    try
                    {
                        v.wait();
                    } catch (InterruptedException e)
                    {
                        //nichts
                    }
                }
                System.out.print("Konsument fand "+(String)v.elementAt(0));
                v.removeElementAt(0);
                System.out.println(" (verbleiben: "+v.size()+")");
            }
            try {
                  Thread.sleep((int)(100*Math.random())); }
```

```
public class Synchronisation
{
    public static void main(String[] args)
    {
        Vector v = new Vector();
        Producer p = new Producer(v);
        Consumer c = new Consumer(v);
        p.start();
        c.start();
    }
}
```

```
Produzent erzeugte Wert 0.09100924684649958
Konsument fand Wert 0.09100924684649958 (verbleiben: 0)
Produzent erzeugte Wert 0.5429652807455857
Konsument fand Wert 0.5429652807455857 (verbleiben: 0)
Produzent erzeugte Wert 0.6277057416210464
Konsument fand Wert 0.6277057416210464 (verbleiben: 0)
Produzent erzeugte Wert 0.6965546173953919
Produzent erzeugte Wert 0.6990053250441516
Produzent erzeugte Wert 0.9874467815778902
Produzent erzeugte Wert 0.12110075531692543
Produzent erzeugte Wert 0.5957795111549329
Konsument fand Wert 0.6965546173953919 (verbleiben: 4)
Produzent erzeugte Wert 0.019655027417308846
Konsument fand Wert 0.6990053250441516 (verbleiben: 4)
Konsument fand Wert 0.9874467815778902 (verbleiben: 3)
Konsument fand Wert 0.12110075531692543 (verbleiben: 2)
```

Weitere Thread Methoden

- currentThread()
  Liefert eine Referenz auf den Thread, der zur Zeit vom Prozessor abgearbeitet wird.
- sleep()
  Methode, um einen Thread schlafen zu legen
- isAlive()
- join()
- stop(), resume(), suspend() (Diese Methoden sind deprecated, da sie sich als absturzfördernd erweisen oder Deadlocks verursachen)

Verwaltung von Threads

Threads können nicht nur ausgeführt und synchronisiert werden, sie besitzen auch noch zusätzliche administrative Eigenschaften

- setName() (sonst Name = "Thread-"+n)
- getName()
- setPriority() (für Scheuduler wichtig)
- getPriority()
- Weitere Methoden um ThreadGruppen erzeugen und verwalten

Zurück zu den Animationen

- 2 Aufgaben sollen gleichzeitig erledigt werden
  1. Die Anzeige im Appletbereich.
  2. Abfrage der aktuellen Zeit und Formatierung in einen String.
- Aufgaben werden in 2 Threads aufgeteilt.
  1. Hauptthread übernimmt Aufgabe 1
  2. Zusätzlicher Thread für Aufgabe 2

Animation mit Threads

```
public class DigitalClock extends Applet
{
   ZeitThread ZeitThread = null;
   String szZeit = null;
   public void init()
   {
        ZeitThread = new ZeitThread();
        ZeitThread.start();
   }
   public void stop()
   {
        if(this.ZeitThread != null)
        {
            ZeitThread.interrupt();
            ZeitThread = null;
        }
   }
   public void paint(Graphics g)
   {
        g.setColor(Color.blue);
        g.drawString("Zeit: " + this.szZeit, 20, 45);
   }
}
```

Animation mit Threads

```
class ZeitThread extends Thread
{
   public void run()
   {
        while(isInterrupted() == false)
        {
          szZeit = new Date(System.currentTimeMillis());
          repaint();
          try
          {
               Thread.sleep(1000);
          }
          catch (InterruptedException e)
          {
               return;
          }
        }
   }
}
```

Reduzieren des Flimmerns in Animationen

- Flimmer ist ein Nebeneffekt der Art, wie der Ausgabebereich in einem Java-Programm aktualisiert wird.
- Je später ein Bildanteil innerhalb eines Animationsschrittes angezeigt wird, desto stärker ist das Flimmern.
- Das Ausmaß ist abhängig von
  1. Der Qualität der Java-Laufzeitumgebung
  2. Der Prozessorgeschwindigkeit

Ursprung des Flackerns

- repaint() -> paint()
- repaint() -> update() -> paint()
- Die update() Methode löscht den kompletten Bildschirminhalt mit der aktuellen Hintergrundfarbe des Applets.

  Update() ist der Übeltäter in Bezug auf das Flimmerproblem
- Appletfenster springt immer beim Aufruf von repaint() vom Zustand „gelöscht" und „neugezeichnet" hin und her (flimmern).

3 Hauptmethoden um das Flimmern zu vermeiden

- update() Methode überschreiben
  1. Bildschirm überhaupt nicht löschen
  2. Nur die Bereiche löschen, die sich geändert haben („Clipping“)
- „Dubble Buffering“

Wann soll welche Methode angewandt werden

Hintergrund nicht löschen:

- Bei nicht bewegten Animationen

Dubble Buffering oder teilweise Löschung des Hintergrundes („Clipping“):

- Bei bewegten Animationen

update() Methode überschreiben

- Standard update() Methode

```
public void update(Graphics g)
{
  g.setColor(getBackgroundColor());
  g.fillRect(0,0,size().width(),size()height);
  g.setColor(getForegroundColor());
  paint(g);
}
```

Bildschirm nicht löschen

- Bildschirm gar nicht löschen

```
public void update(Graphics g)
{
    paint(g);
}
```

Dubble Buffering

- Alle Zeichenaktivitäten werden in einem Puffer (zweite Zeichenfläche) abseits des Bildschirms vorgenommen.

  Anschließend wird der Puffer in einem Schritt am Bildschirm angezeigt.

  Dadurch wird verhindert, dass Zwischen-schritte innerhalb eines Zeichenvorgangs aus Versehen erscheinen und die Animation stören.

Vorgehensweise DoubleBuffering

- Buffer Objekt aus java.awt.Image anlegen
- Buffer Objekt mit Component.createImage() instanzieren

```
java.lang.Object
|
+--java.awt.Component
       |
       +--java.awt.Container
                  |
                  +--java.awt.Panel
                             |
                             +--java.applet.Applet
```

- Graphics g = „Buffer Objekt".getGraphics();
- update() Methode überschreiben

```
Graphics BufferGraphics;
Image Buffer;
Size Size;

public void init()
{
   Size           = this.size();
   buffer         = this.createImage(size.width, size.height);
   BufferGraphics = buffer.getGraphics();
}
public void update(Graphics g)
{
   paint(g);
}
public void paint(Graphics g)
{
   BufferGraphics.setColor(getBackground());
   BufferGraphics.fillRect(0, 0,this.getSize().width,
        this.getSize().height);
   ...
   g.drawImage(Buffer,0,0,this);
}
```

Vor-/Nachteile von Double Buffering

- Vorteile:
  - entfernt alle Flimmereffekte
  - Bilder werden komplett aufgebaut, bevor sie angezeigt werden
- Nachteile:
  - mehr Speicherplatz
  - weniger effizient als der reguläre Puffer
  - doppeltes Schreiben der Bilddaten

Clipping

- Nur die Bereiche vom Bildschirm löschen, die auch tatsächlich neu gezeichnet werden müssen
- Vorteil:

  Schnelles Zeichnen auf den Bildschirm
- Nachteil:

  Man muss sich alte und neue Positionen merken

Vorgehensweise Clipping

2 Möglichkeiten:

- Am Graphics Objekt eine Clipping Region einstellen, somit wird dem System mitgeteilt, dass es Zeichenoperationen nur innerhalb dieser Region ausführen muss

```
paint(Graphics g)
{
    g.setClip(int x, int y, int width, int
          heigth);
}
```

- Oder repaint() wie folgt aufrufen:

```
repaint(int x, int y, int width, int heigth);
```