Grundlagen der Informatik für Ingenieure I

Threads, Images, Animation and Sound

11.1 Aktivitätsträger

- Grundsätzlich laufen in Rechensystemen immer zur Benutzerapplikation parallele (quasi parallele) Aktivitäten ab (Siehe B5).
- Inwieweit diese Möglichkeiten aber auch Benutzerapplikationen verfügbar gemacht werden, hängt von den Betriebssystemen ab, die den Rechner steuern und dem Benutzer eine abstrakte Sicht auf das Rechensystem bereitstellen. Was bei Unix-Systemen, die von jeher Multiprocessing/Multiuser-Systeme waren, kaum zu Problemen führt, kann bei PC-Systemen unter Microsoft-Betriebssystemen, die z. T. immer noch auf MS-DOS basieren, durchaus ein Problem sein.
  U. a. auch aus diesem Grunde stellt Java selbst einen “Parallelisierungsmechanismus” zur Verfügung, den man immer dann einführen sollte, wenn CPU-intensive Aktivitäten zur Bearbeitung anstehen, z. B. bei einer Animation.
- Der Hintergrund ist der, dass der Browser immer in der Lage sein soll, Aktivitäten wahrzunehmen, z. B. während wir mit dem Applet einen Ausgabestrom auslösen, sollte trotzdem der Scrollbar des Browserfensters veränderbar sein. Wenn Browser und Applet aber denselben Aktivitätsträger (Thread) benutzen, kann der Browser erst dann wieder aktiv werden, wenn das Applet "fertig" ist.

11.1 Aktivitätsträger

- Dem Thema *Threads* haben wir ein eigenes Teil-Kapitel gewidmet. Hier werden wir sie zunächst ohne weitere tiefergehende Erläuterungen zur **vereinfachten** Benutzung einführen:
  - Grundsätzlich gibt es zwei Möglichkeiten *Threads* zu erzeugen
    - Ableiten von *Thread*
    - Implementieren des *Runable-Interfaces*
  - Für Applets müssen wir das *Runable-Interface* verwenden(Warum?):

```
public class MeinProblem extends java.applet.Applet
implements Runnable {
...
{
```

  - Da wir jetzt einen “eigenen” Aktivitätsträger kontrollieren müssen, benötigen wir einen *“handle”* dies zu tun. Dafür vereinbaren wir eine Variable vom Typ *Thread*:

```
Thread animation;
```


11.1 Aktivitätsträger

- Wir verwenden in diesem Kapitel nur drei Methoden zur Threadkontrolle:
  - Die *run()-method,* definiert im *Runable-Interface*.
    Diese Methode enthält den Code, den der *Thread* abarbeiten soll:

```
public void run{
...Code des Threads...
}
```

  - Die *start()-method* und *stop()-method*, definiert in der Klasse *Thread,* sollten immer paarweise verwendet werden:
    - *start()-method:* Startet einen *Thread*:

```
public void start() {        // Start des Applets!
    if (animation == null) {
        animation = new Thread(this);
        animation.start();    // Start des Threads!
    }
}
```

11.1 Aktivitätsträger

- *stop()-method:* Stoppt einen Thread:

```
public void stop() {
    if (animation != null) {
        animation.stop();     // Stoppt den Thread!
        animation = null;
    }
}
```


11.1 Aktivitätsträger

- Ein vollständiges Beispiel: “Die Java-Digital-Clock”

```
import java.awt.Graphics;
import java.awt.Font;
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.TimeZone;

public class DigitalClock extends java.applet.Applet
    implements Runnable {

    Font theFont = new Font("TimesRoman",Font.BOLD,24);
    GregorianCalendar theDate;
    Thread runner;

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

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

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



```
    public void paint(Graphics g)

/*  Achtung: Funktioniert korrekt nur ab Java 1.2 und
    auch nur in dieser Form. Die TimeZone "CEST"
    ist z. Zt. fest auf Kuweit eingestellt!
    Pruefen!
*/
            TimeZone tz = TimeZone.getTimeZone("GMT");
            tz.setRawOffset(7200000);

        theDate = new GregorianCalendar();
        g.setFont(theFont);
        g.drawString("" + theDate.getTime(), 10, 50);
    }
}
```

11.2 MIME - Formate und Datei-*Extensions*

- HTML ist nicht das einzige Format, das im Internet übertragen wird. Damit Browser und andere Applikationen die verschiedenen Dokumenttypen erkennen und verarbeiten können, wurden sie standardisiert und klassifiziert(x- heißt: noch nicht standardisiert).
  MIME steht für: *Multipurpose Internet Mail Extension*
  - Hier eine Auswahl:

Dokumententyp	Format	Date-Extensions	MIME-Typ
Text	HTML	htm; html	text/html
Text	*"plain"* text	txt	text/plain
Anwendung	PostScript	ps; PS; ai; eps	application/postscript
Gemischtes Dokument	mehrere	offen	multipart/x-mixed-replace
Bild	GIF	gif	image/gif
Bild	JPEG	jpg; jpeg	image/jpeg
Ton	AU	au	audio/basic
Ton	WAV	wav	audio/x-wav
Ton	mp3	mp3	???
Video	MPEG	mpeg; mpg	video/mpeg
Virtuelle Welt	VRML	wrl	x-world/x-vrml (model/vrml)

11.3 Images

- Von *Images* spricht man, wenn man Bilder, Graphiken oder Texte punkt-(bzw. pixel-) weise darstellt. Diese Darstellungsart ist sehr speicherintensiv. Ein Standard Videobild z. B. mit 625 x 800 Punkten und 24 Bit Farbtiefe benötigt 1.5 MByte Speicherplatz.
  Aus diesem Grund wird man diese Bilder in der Regel kompremiert ablegen, was wiederum bedeutet, dass das *Image* - Laden mit einer Datendekompression einhergehen muss.
  Laden und Dekompremieren von *Images* ist zeitintensiv.

- *Images* liegen in seperaten Dateien und müssen aus diesen heraus geladen werden. Die Klasse *Applet* stellt dafür die *getImage()-method* bereit, die das *Image* lädt und uns zur Weiterverarbeitung ein *Image*-Objekt zur Verfügung stellt.
  Wie wir schon im Umgang mit HTML- und Applet-Dateien gelernt haben, gibt es verschiedenen Methoden separat angelegte Dateien im Java-Umfeld anzusprechen:
  - Die *Image*-Datei ist unter einer festen URL zu finden:

```
Image img = getImage(
new URL("http://www.cip/mybase/images/bild.gif"));
```

11.3 Images

- Die *Image*-Datei liegt in der gleichen *Directory* wie die HTML-Datei, die die Referenz auf das Java-Applet enthält:

```
IMAGE img = getImage(getDocumentBase(), “bild.gif”);
```

- Die *Image*-Datei liegt in der gleichen Directory wie das Java-Applet:

```
IMAGE img = getImage(getCodeBase(), “bild.gif”);
```

- Man kann natürlich auch relative Pfadnamen angeben, wie:

```
IMAGE img = getImage(getCodeBase(),
“gif_images/bild.gif”);
```

- *Image*-Formate
  Java unterstützt zur Zeit folgende *Image*-Formate:
  - *GIF - Graphical Interchange Format*
  - *JPEG - Joint Photographer Expert Group*

11.3.1 Mit *Images* arbeiten

- Wie bereits erwähnt, ist das Laden und Dekompremieren ein aufwendiger Vorgang, so dass für diese Aktivität ein besonderer *Thread* kreiert wird. Die Kontrolle geht “sofort” an den Aufrufer zurück, der weitere “Aktivitäten” ausführen kann. Die Parallelarbeit im Hintergrund wird vom *ImageObserver* überwacht, der z. B. entscheidet, wann ein *Image* wirklich auf den *Screen* ausgegeben wird.
- Für die Ausgabe eines *Images* gibt es zwei Methoden *drawImage()*:
  (Die Angabe des Parameter *‘this’* bedeutet, man verwendet den *default ImageObserver*)
  - In der einen Methode wird das *Image* in Originalform relativ zur Koordinate (0,0) plaziert:

```
public void paint() {
    g.draw.Image(img, 10, 10, this);
}
```

11.3.1 Mit *Images* arbeiten

- In der zweiten Methode wird die Größe der *Image-Bounding-Box* zusätzlich explizit mit zwei Parametern festgelegt:

```
public void paint() {
    g.draw.Image(img, 10, 10, breite, hoehe, this);
}
```

- mit folgenden Methoden erhält man die Orginalgrößen:

```
breite = myImage.getWidth(this);
hoehe = myImage.get.Height(this);
```

Man ist gut beraten, den Returnparameter zu kontrollieren, denn solange das *Image* nicht vollständig geladen ist, wird eine '-1' zurückgegeben!

11.3.2 Ein vollständiges *Image*-Beispiel

```
import java.awt.Graphics;
import java.awt.Image;

public class LadyBug extends java.applet.Applet {

    Image bugimg;

    public void init() {
        bugimg = getImage(getCodeBase(),
            "images/ladybug.gif");
    }

    public void paint(Graphics g) {
        int iwidth = bugimg.getWidth(this);
        int iheight = bugimg.getHeight(this);
        int xpos = 10;
        // 25 %

        g.drawImage(bugimg, xpos, 10,
            iwidth / 4, iheight / 4, this);
```

11.3.2 Ein vollständiges *Image*-Beispiel}

```
            // 50 %

            xpos += (iwidth / 4) + 10;
            g.drawImage(bugimg, xpos , 10,
                iwidth / 2, iheight / 2, this);
            // 100%
            xpos += (iwidth / 2) + 10;
            g.drawImage(bugimg, xpos, 10, this);

            // 150% x, 25% y
            g.drawImage(bugimg, 10, iheight + 30,
                (int)(iwidth * 1.5), iheight / 4, this);
        }
    }
```


11.4 Animation

- Animation - bewegte Bilder - sind letztendlich optische Täuschungen. Wie bei einem herkömmlichen Film, der uns in einer schnellen Folge leicht unterschiedliche Einzelbilder projeziert, wird auch in einem Rechner verfahren. Im Java-Umfeld bedeutet das, einzelne - leicht unterschiedliche - *Graphics-Objekte* mit der *draw()-method* auf den Screen auszugeben.
- Nur mit *draw()* immer wieder etwas verschiedene Bilder auszugeben wird langsam den *Screen* füllen. Notwendig ist, dass der *Screen* vor jedem *draw()* gelöscht wird. Dafür sorgt die Methode *repaint(), die wiederum die Methode update() aufruft*.
  - Die *update-method()* ist wie folgt implementiert:

```
public void update(Graphics g) {
    g.setColor(getBackground());
    g.fillRect(0,0,width, height);
    g.setColor(getForeground());
    paint(g);
}
```

11.4.1 Flackernde Animation

- Je langsamer ein Graphiksystem bei Wiederaufbau eines Bildes ist, desto störender wirken sich die Wechsel zwischem Löschen und Zeichnen als flackern aus.
- Um dieses zu vermeiden, gibt es folgende Verfahren:
  - Wo sich keine Konturen verändern, verzichtet man auf das Löschen.
  - Man löscht bei geringen Veränderungen nur die Teile im Bild, die verschieden vom vorhergehenden sind oder
  - man verwendet das sogenannte *Double Buffering*.
- *Double Buffering* ist ein Verfahren neben den Puffer für den realen Screen einen Puffer für einen virtuellen Screen zu verwenden. Der notwendige Neuaufbau des Bildes wird jeweils auf den *Off_Screen_Buffer* vollzogen; danach schaltet man den veränderten Puffer auf den Screen. Auf diese Methode kommen wir am Ende des Kapitels zurück.
- Overriding der *update()-method* derart, dass kein Löschen erfolgt, demonstriert an einem **geeigneten** Beispiel:

11.4.1 Flackernde Animation

Ein flackerndes Beispiel:

```
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Font;

public class ColorSwirl extends java.applet.Applet implements
Runnable {
  Font f = new Font("TimesRoman", Font.BOLD, 48);
  Color colors[] = new Color[50];
  Thread runner;
  public void start() {
    if (runner == null); {
      runner = new Thread(this);
      runner.start();
    }
  }
  public void stop() {
    if (runner != null) {
      runner.stop();
      runner = null;
    }
  }
```

11.4.1 Flackernde Animation

```
public void run() {

   float c = 0;
   for (int i = 0; i <colors.length; i++) {
    colors[i] = Color.getHSBColor(c, (float)1.0, (float)1.0);
   c += .02;
   }
   int i = 0;
   while (true) {
      setForeground(colors[i]);
      repaint();
   i++;
      try { Thread.sleep(200); }
      catch (InterruptedException e) { }
      if (i == colors.length) i = 0;
   }
 }

 public void paint(Graphics g) {
   g.setFont(f);
   g.drawString("Look to the Cookie!", 15, 50);
 }
```


11.4.1 Flackernde Animation

```
/* Um den Effekt des Flackerns deutlich zu machen, der bei unserer
   schnellen Graphik sonst nicht zur Geltung kommt, wurde die
   Standard - Implementation von update() nachimplementiert und eine
   Verzoegerung eingebaut.
*/

  public void update(Graphics g) {
     g.setColor(getBackground());
     g.fillRect(0,0,size().width,size().height);

     try { Thread.sleep(100); }
     catch (InterruptedException e) { }

     g.setColor(getForeground());
     paint(g);
  }
}
```

- Override der *update()-method* derart, dass sie nichts löscht:

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

11.4.2 Animation mit *Images*

- Zum Neko-Applet
  - Neko ist eine kleine Animation auf dem Macintosh von Kenji Gotoh (Künstlername?/1989); eine Katze (japanisch “Neko”) versucht den *Mouse*-Zeiger zu fangen. Neun dieser *Images* stehen uns zur Verfügung.
  - Wir Organisieren die *Images* als *array_of Images*
  - In der *init()-method* laden wir diese *Images*
  - Unsere Animation:
    - Neko läuft von links in den Screen,
    - hält in der Mitte an und gähnt,
    - kratzt sich 4 mal,
    - schläft,
    - wacht auf und läuft nach rechts aus dem Screen.

11.4.2 Animation mit *Images*

- Erläutert direkt am Beispiel:

```
import java.awt.Graphics;
import java.awt.Image;
import java.awt.Color;

public class Neko extends java.applet.Applet
    implements Runnable {

    Image nekopics[] = new Image[9];
    Image currentimg;
    Thread runner;
    int xpos;
    int ypos = 50;

    public void init() {
        String nekosrc[] = { "right1.gif", "right2.gif",
            "stop.gif", "yawn.gif", "scratch1.gif",
            "scratch2.gif","sleep1.gif", "sleep2.gif",
            "awake.gif" };
        for (int i=0; i < nekopics.length; i++) {
            nekopics[i] = getImage(getCodeBase(),
            "images/" + nekosrc[i]);
        }
    }
```

11.4.2 Animation mit *Images*

```
    public void start() {
        if (runner == null) {
            runner = new Thread(this);
            runner.start();
        }
    }
    public void stop() {
        if (runner != null) {
            runner.stop();
            runner = null;
        }
    }
    public void run() {
        setBackground(Color.white);

    // run from one side of the screen to the middle
        nekorun(0, size().width / 2);
    // stop and pause
        currentimg = nekopics[2];
        repaint();
        pause(1000);
  // yawn
        currentimg = nekopics[3];
        repaint();
        pause(1000);
```


11.4.2 Animation mit *Images*

```
            // scratch four times
            nekoscratch(4);
            // sleep for 5 "turns"
            nekosleep(5);
            // wake up and run off
            currentimg = nekopics[8];
            repaint();
            pause(500);
            nekorun(xpos, size().width + 10);
        }
        void nekorun(int start, int end) {
            for (int i = start; i < end; i += 10) {
                xpos = i;
                // swap images
                if (currentimg == nekopics[0])
                    currentimg = nekopics[1];
                else currentimg = nekopics[0];
                    repaint();
                pause(150);
            }
        }
```

11.4.2 Animation mit *Images*

```
    void nekoscratch(int numtimes) {
        for (int i = numtimes; i > 0; i--) {
            currentimg = nekopics[4];
            repaint();
           pause(150);
            currentimg = nekopics[5];
            repaint();
            pause(150);
        }
    }
    void nekosleep(int numtimes) {
        for (int i = numtimes; i > 0; i--) {
            currentimg = nekopics[6];
            repaint();
            pause(250);
            currentimg = nekopics[7];
            repaint();
            pause(250);
        }
    }
```


11.4.2 Animation mit *Images*

```
    void pause(int time) {
        try { Thread.sleep(time); }
        catch (InterruptedException e) { }
    }
    public void paint(Graphics g) {
        if (currentimg != null)
            g.drawImage(currentimg, xpos, ypos, this);
    }
}
```

11.5 Sound

- Analog zu *Images* kann man auch Audio-Dateien laden und ausgeben, natürlich nicht auf den *Screen*, aber auf das “Audio-System”, sofern am Rechner eines vorhanden ist.
- Java unterstützt zur Zeit nur das *AU*-Format.
- Analog zu den entsprechenden *Images*-Methoden existieren auch Audio-Methoden
  - Laden eines Audionclips

```
AudioClip myClip =
getAudioClip(getCodebase(),”miau.au”);
```

  - Abspielen; Immer wieder abspielen; Stoppen eines Clips

```
clip.play();clip.loop; clip.stop
```

  - Laden und Abspielen:

```
play(getCodeBase(), “miau.au”);
```


11.5 Sound

- Ein Beispiel - Audio-Thread läuft im Hintergrund!

```
import java.awt.Graphics;
import java.applet.AudioClip;
public class AudioLoop extends java.applet.Applet
    implements Runnable {
    AudioClip bgsound;
    AudioClip beep;
    Thread runner;

    public void start() {
         if (runner == null) {
             runner = new Thread(this);
             runner.start();
         }
    }
    public void stop() {
        if (runner != null) {
            if (bgsound != null) bgsound.stop();
            runner.stop();
            runner = null;
        }
    }
```

11.6 Sound

```
public void init() {
      bgsound = getAudioClip(getCodeBase(),"audio/loop.au");
       beep = getAudioClip(getCodeBase(), "audio/beep.au");
    }
    public void run() {
        if (bgsound != null) bgsound.loop();
        while (runner != null) {
            try { Thread.sleep(5000); }
            catch (InterruptedException e) { }
            if (beep != null) beep.play();
        }
    }

    public void paint(Graphics g) {
        g.drawString("Playing Sounds....", 10, 10);
    }
}
```


11.7 Double Buffering

- Nach dem wir nun wissen, wie man mit Images umgeht, können wir uns auch einen *Off_Screen_Image* schaffen, auf dem wir unsere Ausgaben vorbereiten bevor wir sie tatsächlich auf den Sreen ausgeben.
- *Double Buffering* kostet Leistung und Resourcen!
- Was ist zu tun?
  - Wir benötigen zwei Instanzvariablen, die den *Image*-Context und den Graphik-Context unseres *Off_Screen_Buffers* repräsentieren:

    ```
    Image offscreenImage;
    Graphics offscreenGraphics;
    ```

  - in der *init()-method* kreieren wir uns die entsprechenden Objekte:

    ```
    offscreenImage = createImage(size().Width,
    size().Height);
    offscreenGraphics = offscreenImage.getGraphics();
    ```

  - Sämtliche *draw()'s* machen wir auf den *Off_Screen_Buffer*; normalerweise innerhalb der *paint()-method:*

    ```
    offscreenGraphics.drawImage(img,10,10,this);
    ```

11.7 Double Buffering

- Sind alle Ausgaben auf den *Off_Sreen_Buffer* abgeschlossen überschreiben wir (mit dem letzten Statement in der *paint()-method*) den Screen mit

```
g.drawImage(offScreenImage, 0, 0, this)
```

- Nun müssen wir nur noch ein *override* der *update()-method* vornehmen:

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


11.7 Double Buffering

- Ein Beispiel:
Hier die “Off-Screen-Variante”. An der Ihnen bekannten Stelle finden Sie unter “Checkers” auch die Originalversion ohne *Double Buffering.*

```
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Image;

public class Checkers2 extends java.applet.Applet
implements Runnable {

  Thread runner;
  int xpos;
  Image offscreenImg;
  Graphics offscreenG;

  public void init() {
    offscreenImg = createImage(this.size().width,
                               this.size().height);
    offscreenG = offscreenImg.getGraphics();
  }
```

11.7 Double Buffering

```
public void start() {
  if (runner == null); {
    runner = new Thread(this);
    runner.start();
  }
}
public void stop() {
  if (runner != null) {
    runner.stop();
    runner = null;
  }
}
public void run() {
  while (true) {
    for (xpos = 5; xpos <= 105; xpos+=4) {
      repaint();
      try { Thread.sleep(100); }
      catch (InterruptedException e) { }
    }
   xpos = 5;
  }
}
```


11.7 Double Buffering

```
    public void update(Graphics g) {
       paint(g);
    }
  public void paint(Graphics g) {
      // Draw background
      offscreenG.setColor(Color.black);
      offscreenG.fillRect(0,0,100,100);
      offscreenG.setColor(Color.white);
      offscreenG.fillRect(100,0,100,100);

      // Draw checker
      offscreenG.setColor(Color.red);
      offscreenG.fillOval(xpos,5,90,90);
      g.drawImage(offscreenImg,0,0,this);
    }

    public void destroy() {
       offscreenG.dispose();
    }
  }
```

Notizen

Notizen