

# Spiele-Programmierung in Java

Version 1.2



## Zum Inhalt:

Vorwort.....	3
Das Spielfeld.....	4
So sieht's aus .....	6
Initialisierungen .....	7
Die Spielschleife .....	8
Zeitmessung .....	9
So sieht's aus: .....	12
Weitere Methoden .....	13
Die Klasse Sprite .....	14
Interface Movable .....	14
Interface Drawable .....	14
Die eigentliche Klasse .....	15
Laden der Bilddaten .....	19
Das erste bewegte Objekt .....	21
Anpassung des GameLoop .....	22
Und so sieht's aus: .....	25
Die Steuerung .....	26
Die Methode checkKeys() .....	29
Und so sieht's aus: .....	30
Feinschliff .....	31
Es wird abstrakt .....	32
Die neue Klasse Heli .....	34
Änderung des Programm-Starts .....	35
Und so sieht's aus: .....	39
Weitere Objekte .....	40
Die Klasse Cloud .....	40
Und so sieht's aus: .....	43
Noch mehr Objekte .....	44
Anpassen der Klasse Sprite .....	45
Die Klasse Rocket .....	47
Und so sieht's aus .....	55
Feinschliff 2. Teil .....	56
Objekte wieder löschen .....	56
Hintergrundbild .....	60
Und so sieht's aus: .....	62
Kollisionen.....	64
So sieht's aus: .....	69
Spiel-Ende signalisieren .....	69
Und so sieht's aus: .....	75
Pixelgenaue Kollisionsermittlung .....	76
Und so sieht's aus: .....	78
Sound.....	79
Die Klasse SoundLib .....	80
Fertig – vorerst.....	85

## Vorwort

Dieses Tutorial wurde geschrieben, um eine Möglichkeit aufzuzeigen, wie man Spiele bzw. Animationen in Java programmiert. Es handelt sich dabei aber nur um (m)eine Herangehensweise – andere Vorgehensweisen sind jederzeit denkbar und vielleicht sogar, je nach Vorhaben, besser geeignet.

Das Tutorial ist für Einsteiger gedacht, daher wird auf einige Feinheiten vorerst verzichtet. Grundkenntnisse der Java-Programmierung werden aber vorausgesetzt. Darunter fällt vor allem, dass bekannt ist, wie man die API liest!!! ☺

Auf das Verwenden von Packages habe ich im ersten Beispiel bewusst verzichtet.

Das Tutorial wurde auf Basis Java 1.6 erstellt. Unbekannte oder nicht komplett dargestellte Methoden bitte ich in der entsprechenden API nachzulesen. Die verwendete IDE ist Eclipse, OS ist Windows XP.

An dieser Stelle auch einen Dank an alle Mitglieder von [www.java-forum.org](http://www.java-forum.org), die mitgeholfen haben, dass dieses Tutorial möglichst wenig Fehler beinhaltet und auch immer wieder einmal Fragen dazu beantworten.

Danke auch an Marco13, der so lange "genervt" hat, bis ich eine neue Version gebastelt habe. ☺☺☺

## Änderungen

Den nachträglichen Quatsch zu ConcurrentModificationException habe ich jetzt rausgeworfen und das mal gleich richtig programmiert, so dass der Fehler nicht mehr auftreten sollte. ☺ Dabei auch gleich noch etliche Tippfehler korrigiert und auch sonst ein paar Ungereimtheiten gerade gezogen, u. a. ein paar Bezeichnungen korrigiert, die ich gerne durcheinander schmeiße.

# Das Spielfeld

Los geht's....

Das Spielfeld ist die Anzeigekomponente, hier wird "die ganze Action ablaufen". Gleichzeitig ist es unsere Hauptklasse, welche die main-Methode und wichtige Methoden zur Initialisierung enthält.

Das Spielfeld erbt von JPanel, um eine vorhandene doppelt gepufferte Komponente zu verwenden und unnötigen Programmieraufwand zu sparen. Für komplexere Spiele wäre es denkbar, durch aktives Rendern die Animationen flüssiger ablaufen zu lassen, aber für kleinere Anwendungen ist die hier gezeigte Vorgehensweise ausreichend.

Im Konstruktor sollen lediglich Breite und Höhe des Spielfeldes übergeben werden.

```
1 import javax.swing.JPanel;
2
3 public class GamePanel extends JPanel{
4
5     private static final long serialVersionUID = 1L;
6
7
8     public GamePanel(int w, int h){
9
10    }
11
12
13 }
14
```

Als nächstes fügen wir eine main-Methode ein und hinterlegen im Konstruktor den Code um ein Fenster zu erzeugen, in das wir unser JPanel reinpacken. Auch das Fenster ist aus dem Swing-Paket. Hier bitte keinen Frame verwenden, sondern einen JFrame, um das unnötige Mischen von leicht- und schwergewichtigen Komponenten zu vermeiden.

```
4 public class GamePanel extends JPanel {
5
6     private static final long serialVersionUID = 1L;
7     JFrame frame;
8
9     public static void main(String[] args) {
10         new GamePanel(800, 600);
11     }
12
13     public GamePanel(int w, int h) {
14         this.setPreferredSize(new Dimension(w, h));
15         frame = new JFrame("GameDemo");
16         frame.setLocation(100, 100);
17         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18         frame.add(this);
19         frame.pack();
20         frame.setVisible(true);
21     }
22
23 }
```

Die markierten Zeilen enthalten den neu eingefügten Code.

#### **Zeile 7:**

Unseren JFrame definieren wir als Instanzvariable, da wir später noch darauf zugreifen wollen.

#### **Zeile 9 – 11:**

die main-Methode als Einsprungpunkt und zur Instanziierung unseres GamePanel. Hierbei werden für das GamePanel eine Breite von 800 Pixeln und eine Höhe von 600 Pixeln übergeben.

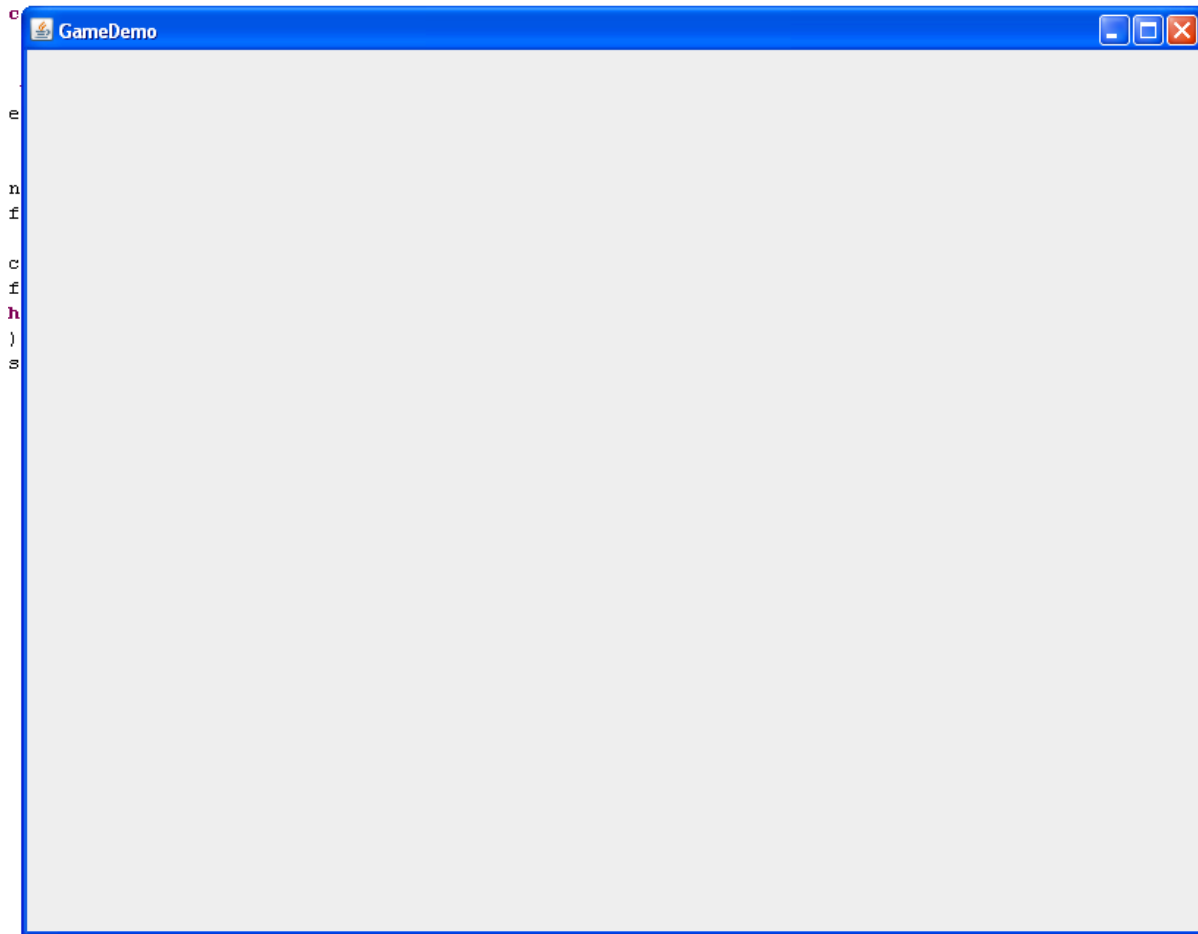
#### **Zeile 14 – 20:**

Mit der von JPanel geerbten Methode `setPreferredSize(...)` übergeben wir die gewünschte Größe an unser GamePanel. Anschließend wird ein Fenster erzeugt und das GamePanel eingebunden.

Über die `pack()`-Methode wird das Fenster an die gewünschte Größe des GamePanels angepasst.

## So sieht's aus

Das Ergebnis unserer Mühen sieht bis jetzt so aus:



Noch nicht wirklich toll, aber das ändert sich in Kürze. 😊

## Initialisierungen

Als nächstes basteln wir uns eine Methode für "einmalige Angelegenheiten", wie das Laden von Images, etc.. . Diese nennen wir entsprechend `doInitializations()`. Im weiteren Verlauf werden wir diese schrittweise weiter ausbauen:

```
4 public class GamePanel extends JPanel {
5
6     private static final long serialVersionUID = 1L;
7     JFrame frame;
8
9     public static void main(String[] args){
10         new GamePanel(800,600);
11     }
12
13     public GamePanel(int w, int h){
14         this.setPreferredSize(new Dimension(w,h));
15         frame = new JFrame("GameDemo");
16         frame.setLocation(100,100);
17         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18         frame.add(this);
19         frame.pack();
20         frame.setVisible(true);
21
22         doInitializations();
23     }
24
25     private void doInitializations() {
26
27
28     }
```

### Zeile 22:

`doInitializations()` werden wir sofort aufrufen, wenn das Fenster erzeugt ist. Da diese Methode später auch das Laden von Grafikdateien enthalten soll, haben wir dann zum Start des Programms z. B. alle Bilder geladen und müssen nicht mit Performance-Einbußen rechnen, wenn wir das während des Spiels machen wollten.

Außerdem werden wir `doInitializations()` auch nach einem Neustart eines Spiels durch Tastatureingabe aufrufen – dazu später mehr.

## Die Spielschleife

Kommen wir zur sog. Spielschleife oder neudeutsch: GameLoop. Innerhalb dieser Schleife werden ständig alle notwendigen Prüfungen und Berechnungen vorgenommen und das Spiel wird jedes Mal neu gezeichnet oder richtiger: das Neuzeichnen wird angestoßen.

Um zu gewährleisten, dass unser Spiel „rund“ läuft, packen wir diesen Teil in einen eigenen Thread – mit Hilfe des Interface Runnable:

```
4 public class GamePanel extends JPanel implements Runnable{
5
6     private static final long serialVersionUID = 1L;
7     JFrame frame;
8
9     public static void main(String[] args){
10         new GamePanel(800,600);
11     }
12
13     public GamePanel(int w, int h){
14         this.setPreferredSize(new Dimension(w,h));
15         frame = new JFrame("GameDemo");
16         frame.setLocation(100,100);
17         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18         frame.add(this);
19         frame.pack();
20         frame.setVisible(true);
21
22         doInitializations();
23
24         Thread th = new Thread(this);
25         th.start();
26     }
27
28     private void doInitializations() {
29
30     }
31
32
33     @Override
34     public void run() {
35
36         while(frame.isVisible()){
37
38
39
40             try {
41                 Thread.sleep(10);
42             } catch (InterruptedException e) {}
43
44         }
45
46     }
47
48 }
49
```



#### Zeile 4:

Hier implementieren wir das Interface Runnable in unser GamePanel, damit wir unsere Spielschleife in einem eigenen Thread laufen lassen können.

#### Zeile 24 – 25:

Hier wird unsere Thread erzeugt und gestartet. In der run-Methode des Threads werden wir dann weiter unten unseren GameLoop laufen lassen.

#### Zeile 33 – 44:

Dies ist die Ausprägung der Methode run() aus dem Interface Runnable. Sie enthält unsere Spielschleife. Innerhalb der dargestellten while-Schleife werden wir periodisch die notwendigen Methoden aufrufen, die unser Spiel am Leben erhalten. Damit dieser Thread nicht "unendlich" weiterläuft, fragen wir ab, ob unser Fenster noch sichtbar ist. Wird das Fenster geschlossen, greift die Bedingung der while-Schleife nicht mehr und die run-Methode wird ganz normal beendet.

Damit auch andere Prozesse nicht zu kurz kommen, wird der Thread jeweils für 10 Millisekunden schlafen gelegt.

## Zeitmessung

Als nächstes werden wir eine Zeitmessung implementieren und ausgeben. Die Durchlaufzeit für die einzelnen Schleifendurchläufe werden wir später noch verwenden, z. B. um unsere Sprites in Abhängigkeit des letzten Schleifendurchlaufs zu bewegen und damit eine gleichmäßige Bewegung zu gewährleisten.

Aus Gründen der Übersichtlichkeit werde ich jetzt nur noch Code-Ausschnitte einfügen.

```
4 public class GamePanel extends JPanel implements Runnable{
5
6     private static final long serialVersionUID = 1L;
7     JFrame frame;
8
9     long delta = 0;
10    long last = 0;
11    long fps = 0;
12
13    public static void main(String[] args){
14        new GamePanel(800,600);
15    }
```

Folgende Instanzvariablen werden implementiert:

**delta** : zur Errechnung der Zeit, die für den letzten Durchlauf benötigt wurde

**last** : Speicherung der letzten Systemzeit

**fps** : Für die Errechnung der Bildrate (fps = frames per second)

Die Variable `last` setzen wir in `doInitializations()` - unserer Methode für alle notwendigen Voreinstellungen. Da diese zur Berechnung der Schleifendurchläufe herangezogen wird, setzen wir hier den ersten Zeitwert. Später mehr dazu.

```
32 private void doInitializations() {  
33  
34     last = System.nanoTime();  
35  
36 }
```

Danach erweitern wir die Spielschleife um 2 Methodenaufrufe:

```
39 @Override  
40 public void run() {  
41  
42     while(frame.isVisible()){  
43  
44         computeDelta();  
45  
46  
47         repaint();  
48  
49         try {  
50             Thread.sleep(10);  
51         } catch (InterruptedException e) {}  
52     }  
53 }  
54  
55 }
```

#### Zeile 44:

`computeDelta()`. Hier werden wir die Zeit für den jeweils vorhergehenden Schleifendurchlauf errechnen.

#### Zeile 47:

`repaint()`. Dies ist die von `Component` geerbte Methode, die ein Neuzeichnen anstößt – dieses wird aber nicht notwendigerweise sofort ausgeführt (näheres dazu: siehe API)

Die Methode `computeDelta()` erhält folgenden Code:

```
57 private void computeDelta() {  
58  
59     delta = System.nanoTime() - last;  
60     last = System.nanoTime();  
61     fps = ((long) 1e9)/delta;  
62  
63 }
```

#### Zeile 59:

Errechnen der Zeit für den Schleifendurchlauf in Nanosekunden

#### Zeile 60:

Speicherung der aktuellen Systemzeit

## Zeile 61:

### Errechnung der Framerate

Weiterhin überschreiben wir jetzt noch die `paintComponent`-Methode unseres Panels um individuelle Zeichenoperationen vornehmen zu können. Vorerst begnügen wir uns aber damit die Framerate an den oberen Rand zu malen.

```
66 @Override
67 public void paintComponent(Graphics g) {
68     super.paintComponent(g);
69
70     g.setColor(Color.red);
71     g.drawString("FPS: " + Long.toString(fps), 20, 10);
72
73
74
75 }
76
77 }
```

Man kann darüber streiten, ob es notwendig ist, die Framerate anzuzeigen. Beim fertigen Spiel wird man sie sicherlich nicht verwenden. Im Verlauf der Spielentwicklung kann sie bei komplexen Berechnungen aber durchaus notwendig sein, um zu prüfen, ob die Animation weiterhin einigermaßen flüssig ausgeführt wird.

## Wichtig:

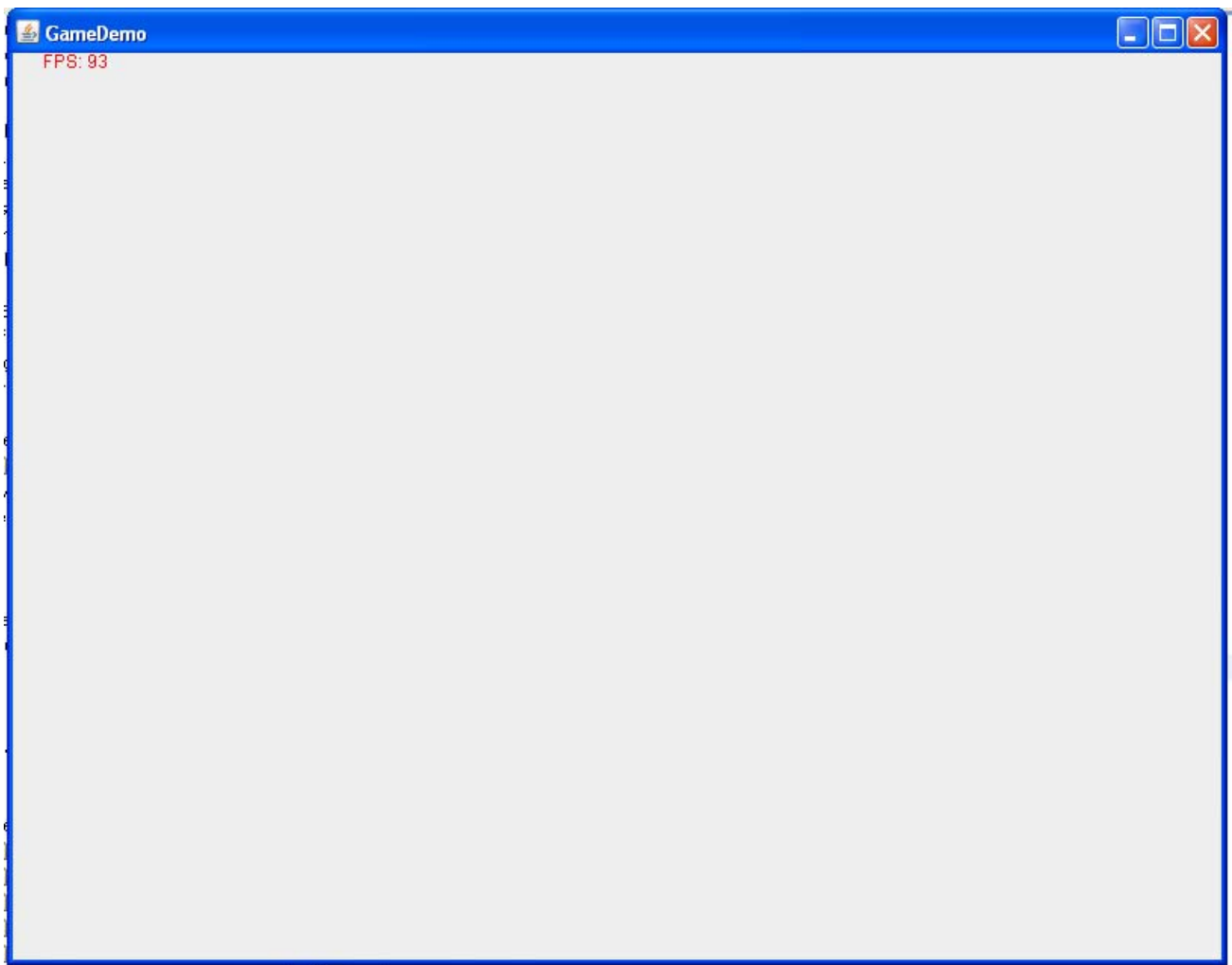
Es ist wichtig, dass die `paintComponent`-Methode der Superklasse am Anfang aufgerufen wird (Zeile 68). So vermeidet man, dass zusätzliche Komponenten, wie z. b. Menüs u.U. nicht gezeichnet werden.

Das `Graphics`-Objekt, welches hier vom System übergeben wird, wird später per Methodenaufruf an alle Objekte weitergegeben. Es ist sozusagen die Referenz auf die Zeichenfläche unseres `JPanel`s, sozusagen unsere Leinwand.

Das `Graphics`-Objekt sollte immer mit übergeben werden und nie, nie, nie mit `getGraphics()` abholt werden (Ausnahmen bestätigen diese Regel ☺) – für Anfänger gilt: nie, nie, nie `getGraphics` verwenden. Dies hilft unnötige Fehler zu vermeiden. Genauer bitte in diversen Foren nachlesen. (Empfehlung: [www.java-forum.org](http://www.java-forum.org) ☺)

Der Nullpunkt für alle Zeichenoperation befindet sich in der linken, oberen Ecke! Daher wird durch die Koordinaten-Angabe 20/10 der `fps`-Wert dorthin gezeichnet, wo er zu sehen ist.

***So sieht's aus:***



## Weitere Methoden

Als nächstes sollen die für den Spielablauf notwendigen Methoden angelegt werden.

```
40 @Override
41 public void run() {
42
43     while(frame.isVisible()){
44
45         computeDelta();
46
47         checkKeys();
48         doLogic();
49         moveObjects();
50
51         repaint();
52
53         try {
54             Thread.sleep(10);
55         } catch (InterruptedException e) {}
56
57     }
58
59 }
```

Dazu wurden die folgenden Methoden angelegt:

### Zeile 47:

checkKeys(): Wird später zur Abfrage von Tastatureingaben verwendet

### Zeile 48:

doLogic(): Wird später zur Ausführung von Logik-Operation herangezogen

### Zeile 49:

moveObjects(): Verwenden wir später um unsere Objekte zu bewegen

Im Verlauf des Tutorials wird auf diese Methoden noch näher eingegangen. Wir werden diese dann nach und nach mit Leben füllen. Jetzt legen wir diese erst einmal leer an (oder lassen Sie von Eclipse generieren, um die (je nach Entwicklungsumgebung) die nervigen Fehlermeldungen weg zu bekommen).

```
61 private void moveObjects() {
62
63 }
64
65 private void doLogic() {
66
67 }
68
69 private void checkKeys() {
70
71 }
72
```

## Die Klasse Sprite

Kommen wir nun zu den bewegten Objekten. Hier wird zunächst einiges an Vorarbeit notwendig sein, bis wir unser erstes Objekt im Fenster anzeigen können – dafür haben wir es später etwas leichter.

Um für die Klasse Sprite einheitliche Methoden zur Verfügung zu stellen, definieren wir zunächst 2 Interfaces. Auch dies ist zunächst einmal zusätzlicher Aufwand (zugegebenermaßen relativ gering) wird uns aber später das Leben erleichtern.

### Interface Movable

```
2 public interface Movable {  
3  
4     public void doLogic(long delta);  
5  
6     public void move(long delta);  
7  
8 }
```

Das Interface enthält 2 Methoden zum Bewegen unserer Objekte: Die Methode `move(long delta)` für die eigentliche Bewegung und die Methode `doLogic(long delta)` für Logikoperationen, wie z. B. Kollisionserkennung, etc.

Die Namensgebung wurde so gewählt, dass eindeutig ist, aus welcher Methode unserer Spielschleife diese Methoden aufgerufen werden, wenn wir sie im Sprite implementiert haben.

### Interface Drawable

```
1 import java.awt.Graphics;  
2  
3 public interface Drawable {  
4  
5     public void drawObjects(Graphics g);  
6  
7 }
```

Das Interface Drawable stellt analog zum Interface Movable eine einheitliche Methode zum Zeichnen unseres Objektes zur Verfügung.

## Die eigentliche Klasse

Jetzt endlich erstellen wir die Klasse Sprite. Sie soll die beiden Interfaces implementieren und uns darüber hinaus die Möglichkeit zur Verfügung stellen, Animation aus einem Image heraus zu erzeugen.

```
5 public class Sprite extends Rectangle2D.Double implements Drawable, Movable{
6
7     private static final long serialVersionUID = 1L;
8     long delay;
9     GamePanel parent;
10    BufferedImage[] pics;
11    int currentpic = 0;
12
13    public Sprite(BufferedImage[] i, double x, double y, long delay, GamePanel p ){
14        pics = i;
15        this.x = x;
16        this.y = y;
17        this.delay = delay;
18        this.width = pics[0].getWidth();
19        this.height = pics[0].getHeight();
20        parent = p;
21    }
22
23    public void drawObjects(Graphics g) {
24    }
25
26    public void doLogic(long delta) {
27    }
28
29    public void move(long delta) {
30    }
31 }
```

Die Klasse Sprite erbt von Rectangle2D.Double, dadurch haben wir die Möglichkeit die aktuellen x- und y-Parameter genau zu errechnen und haben auch gleich noch einige andere grundlegenden Parameter und Methoden zur Verfügung, wie z. B. die Breite und die Höhe unseres Objektes – zumindest ungefähr. Wenn es um Kollisionen geht, wollen wir uns noch näher damit beschäftigen.

### Zeile 8:

Instanzvariable delay um das umschalten zwischen den einzelnen Bildern unseres Image-Arrays zu steuern. Die Variable soll einen Wert im Millisekundenbereich erhalten, der besagt, wie lange ein Bild unseres Arrays jeweils angezeigt wird.

### Zeile 9:

Die Referenz auf unser GamePanel

### Zeile 10:

Ein BufferedImage-Array zum Speichern der Animation in Einzelbildern.

### Zeile 11:

Zähler für das aktuell anzuzeigende Bild

### Zeile 13-21:

Der Konstruktor mit der Übergabe des ImageArrays für die Animation, den Positionswerten, der Verzögerung für die Animation und der Referenz auf unser GamePanel. Außerdem holen wir uns aus dem ersten Bild unseres Array die Höhe und die Breite des Bildes (in der Annahme, dass diese durchgehend gleich bleiben).

### Zeilen 23 - 30:

Die Methoden die wir für unseren Interfaces implementiert haben (je nach Entwicklungsumgebung automatisch), füllen wir später mit Leben.

## Die Methode drawObject(Graphics g)

Jetzt wollen wir die aus den Interfaces implementierten Methoden mit Leben füllen:

Die Methode drawObject(Graphics g) bekommt später das Graphics-Objekt unseres GamePanel übergeben (wir erinnern uns: nie, nie, nie getGraphics() ..) um damit bzw. darauf zeichnen zu können.

Innerhalb der Methode wird jetzt das aktuelle Bild gezeichnet.

```
23 public void drawObjects(Graphics g) {  
24     g.drawImage(pics[currentpic], (int) x, (int) y, null);  
25 }
```

Der x- und y-Parameter wird hier auf ganze Zahlen herunter gebrochen bzw. nach int gecastet (bzw. umgewandelt (um Denglisch zu vermeiden)).

Das ist notwendig, weil Graphics ganze Zahlen verlangt (es gibt ja auch keine halben Pixel). Um die Positionen unsere Objekte genau berechnen zu können macht es aber Sinn, diese als Double-Werte vorzuhalten (daher u.a. Rectangle2D.Double).

Wichtig ist an dieser Stelle, sich das Graphics-Object nicht mit getGraphics() zu holen, da dies – kurz gesagt – unnötige Probleme verursachen kann (kann man gar nicht oft genug wiederholen ☺). Mit der Übergabe des Graphics-Objektes ist man auf jeden Fall auf der sicheren Seite.

## Die Methode doLogic(long delta)

Innerhalb der Methode doLogic hinterlegen wir vorerst nur den Code für die Animation des Objektes. Später werden wir hier zusätzlichen Code einbauen. Da wir hier die Zeit kumulieren müssen, legen wir uns eine weitere Instanzvariable namens animation an.

```
5 public class Sprite extends Rectangle2D.Double implements Drawable, Movable{  
6  
7     private static final long serialVersionUID = 1L;  
8     long delay;  
9     long animation = 0;  
10    GamePanel parent;  
11    BufferedImage[] pics;  
12    int currentpic = 0;  
13  
14    protected double dx;  
15    protected double dy;  
16  
17    public Sprite(BufferedImage[] i, double x, double y, long delay, GamePanel p){  
18        pics = i;  
19        this.x = x;  
20        this.y = y;
```



Die Methode doLogic(long delta) prägen wir wie folgt aus:

```
38 public void doLogic(long delta) {  
39  
40     animation += (delta/10000000);  
41     if (animation > delay) {  
42         animation = 0;  
43         computeAnimation();  
44     }  
45  
46 }
```

Die Methode bekommt die Dauer des letzten Schleifen-Durchlaufs übergeben (vgl. Klasse GamePanel). Diesen kumulieren wir in der Variable animation.

Delta wird hier durch 10000000 dividiert, weil wir hier einen Wert übergeben bekommen, der in Nanosekunden berechnet wurde. Für die bequemere Einstellung der Animationsgeschwindigkeit sollen aber Millisekunden verwendet werden – daher die Umrechnung.

Ist der Wert der Variable animation größer als der voreingestellte Animationswert, setzen wir animation wieder auf Null und rufen die Methode computeAnimation() über die das nächste anzuzeigende Bild ermittelt wird.

Die Methode doAnimation() ist momentan sehr sparsam ausgelegt, kann aber bei der Entwicklung komplexerer Spiele viel umfangreicher werden, beispielsweise könnte man entsprechenden Code hinterlegen, um die Animation rückwärts laufen zu lassen oder ähnliches. Auch wir werden diese Methode später noch etwas modifizieren.

```
38 private void computeAnimation(){  
39  
40     currentpic++;  
41  
42     if(currentpic>pics.length){  
43         currentpic = 0;  
44     }  
45  
46 }
```

Momentan wird unsere Variable currentpic bei jedem Aufruf um eins erhöht. Wenn Sie die Anzahl der vorhandenen Bilder überschreitet (Arrays beginnen mit dem Zähler 0!), wird sie auf Null gesetzt, d. h. die Animation beginnt von vorne.

## Die Methode move(long delta)

Um unser Objekt bewegen zu können, müssen wir ihm noch mitteilen, wie schnell diese Veränderung stattfinden soll. Daher werden zwei weitere Instanzvariablen angelegt:

```
12  int currentpic = 0;
13
14  protected double dx;
15  protected double dy;
16
17  public Sprite(BufferedImage[] i, double
18      pics = i;
```

dx wird den Wert für die horizontale Veränderung enthalten (x-Wert)

dy wird den Wert für die vertikale Veränderung enthalten (y-Wert)

Für diese Bewegungs-Variablen legen wir auch gleich noch die entsprechenden get- und set-Methoden an, benennen diese aber etwas sprechender:

```
56  public double getHorizontalSpeed() {
57      return dx;
58  }
59
60  public void setHorizontalSpeed(double dx) {
61      this.dx = dx;
62  }
63
64  public double getVerticalSpeed() {
65      return dy;
66  }
67
68  public void setVerticalSpeed(double dy) {
69      this.dy = dy;
70  }
71
```

Jetzt aber die eigentliche Methode move(long delta):

```
52  public void move(long delta) {
53
54      if(dx!=0) {
55          x += dx*(delta/1e9);
56      }
57
58      if(dy!=0) {
59          y += dy*(delta/1e9);
60      }
61
62  }
```

Wenn unsere Delta-Werte nicht Null sind, verändern wir die Position des Objekts in die entsprechende Richtung – abhängig von der Zeit, die der letzte Durchlauf unseres GameLoop benötigt hat. Durch die Berücksichtigung der Durchlaufzeit wird eine gleichförmige Bewegung des Objektes gewährleistet, selbst wenn einmal umfangreichere Berechnungen während eines Schleifendurchlaufs anfallen sollten.

Jetzt sind wir fast fertig, das einzige was noch fehlt, damit wir endlich ein Sprite anzeigen können, sind die Bilddaten, die wir laden müssen.

## Laden der Bilddaten

Für die Bestückung des Image-Arrays wollen wir folgende kleine Bildsequenz verwenden. Es handelt sich um ein GIF, dass die Vorderansicht eines „Hubschraubers“ enthält. Jedes Einzelbild ist 30 Pixel breit und hoch.



Das GIF speichern wir in einem eigenen Ordner „pics“ im gleichen Verzeichnis, in dem auch die class-Dateien erzeugt werden:

Dateiname	Größe	Typ	Geändert ▲
pics		File Folder	22.10.2007 11:03
GamePanel.class	3 KB	CLASS-Datei	18.10.2007 15:46
Drawable.class	1 KB	CLASS-Datei	18.10.2007 15:50
Movable.class	1 KB	CLASS-Datei	18.10.2007 15:50
Sprite.class	2 KB	CLASS-Datei	22.10.2007 10:54

Nun gilt es, dieses GIF zu laden. Dies erledigen wir in unserer Hauptklasse GamePanel. Hierzu erstellen wir eine Methode `loadPics(String path, int pics)`, die uns eine `BufferedImage`-Array zurück liefert und den Speicherort und die Anzahl der Einzelbilder übergeben bekommt:

```
97 private BufferedImage[] loadPics(String path, int pics){
98
99     BufferedImage[] anim = new BufferedImage[pics];
100     BufferedImage source = null;
101
102     URL pic_url = getClass().getClassLoader().getResource(path);
103
104     try {
105         source = ImageIO.read(pic_url);
106     } catch (IOException e) {}
107
108     for(int x=0;x<pics;x++){
109         anim[x] = source.getSubimage(x*source.getWidth()/pics, 0,
110             source.getWidth()/pics, source.getHeight());
111     }
112
113     return anim;
114 }
115
116 }
```

**Zeile 97:**

Die Methode bekommt Speicherort und Anzahl der Einzelbilder übergeben

**Zeile 99:**

Wir erzeugen ein BufferedImage-Array in der Größe der Einzelbilder

**Zeile 100:**

Ein BufferedImage zum Laden des ganzen Bildes, das später dann aufgeteilt wird.

**Zeile 102:**

Ermitteln der URL des Speicherortes, den wir als Pfadangabe übergeben.

**Zeile 105:**

Laden des Quellbildes über ImageIO

**Zeile 108-110:**

Mit Hilfe der Methode `getSubimage(...)` aus `BufferedImage`, wird das Quellbild in die Anzahl der angegebenen Einzelbilder zerlegt.

**Wichtig:** Diese Methode kann nur Bildsequenzen laden, die hintereinander gezeichnet sind! Außerdem muss für jedes Objekt eine eigene Datei geladen werden. Für andere Ansätze müssen hier Anpassungen vorgenommen werden.

Das eigentliche Laden führen wir vor dem Beginn des Spieles durch, damit uns die dafür benötigte Zeit nicht den Spielfluß stört. Auf keinen Fall sollten Bilddaten erst in `paintComponent` geladen werden! Hierfür haben wir schon früher die Methode `doInitializations()` in unserem `GamePanel` angelegt.

```
37 private void doInitializations() {  
38  
39     last = System.nanoTime();  
40  
41     BufferedImage[] heli = loadPics("pics/heli.gif", 4);  
42  
43 }
```

**Wichtig dabei ist, dass die Pfadangabe wie oben dargestellt übergeben wird!**

## Das erste bewegte Objekt

Jetzt soll es aber endlich losgehen, damit wir das erste bewegte Objekt anzeigen können. Zunächst benötigen wir zwei neue Instanzvariablen in unserem GamePanel: Eine Collection (ich bevorzuge Vektoren (rein subjektiv)) und ein Objekt der Klasse Sprite – nennen wir es copter ☺

```
13  JFrame frame;  
14  
15  long delta = 0;  
16  long last = 0;  
17  long fps = 0;  
18  
19  Sprite copter;  
20  Vector<Sprite> actors;  
21  
22  public static void main(String[] args){  
23      new GamePanel(800,600);  
24  }
```

Diese beiden Instanzvariablen initialisieren wir ebenfalls in unserer Methode doInitializations():

```
41  private void doInitializations() {  
42  
43      last = System.nanoTime();  
44  
45      BufferedImage[] heli = loadPics("pics/heli.gif", 4);  
46  
47      actors = new Vector<Sprite>();  
48      copter = new Sprite(heli,400,300,100,this);  
49      actors.add(copter);  
50  }
```

### Zeile 47:

Instanziiieren des Vectors actors.

### Zeile 48:

Instanziiieren des Sprites an der Position 400/300 mit einer Bildwechselrate von 100 ms.

### Zeile 49:

Sprite in den Vector packen, der alle unsere Objekte beinhalten soll.

## Anpassung des GameLoop

Jetzt müssen wir noch die GameLoop-Methoden mit Leben füllen und dann kann es los gehen.

Der Zugriff auf den Vector sollte ausschließlich über einen Iterator erfolgen. Das verhindert einige Probleme, die in der ersten Version des Tutorials auftraten, wenn Objekte geändert wurden, während mit foreach über den Vector iteriert wurde. Wer sich hierzu tiefer einlesen möchte sollte mal nach ConcurrentModificationException suchen... ☹ (Ausführliche werde ich hier erst mal nicht – ist ja ein Anfängertutorial ☺).

Auch das Entfernen und Hinzufügen von Objekten sollte ausschließlich über einen Iterator erfolgen.

### doLogic():

```
84 private void doLogic() {  
85  
86     for(ListIterator<Sprite> it = actors.listIterator(); it.hasNext();){  
87         Sprite r = it.next();  
88         r.doLogic(delta);  
89     }  
90  
91 }
```

### moveObjects():

```
75 private void moveObjects() {  
76  
77     for(ListIterator<Sprite> it = actors.listIterator(); it.hasNext();){  
78         Sprite r = it.next();  
79         r.move(delta);  
80     }  
81  
82 }  
83
```

### paintComponent(Graphics g):

Das Zeichnen der Objekte ist ein klein wenig problematisch: Einerseits haben wir 2 Threads laufen – unseren GameLoop und den "Ursprungs-Thread", in dem wir unser Fenster erzeugt haben. Während unser GameLoop innerhalb des von uns erzeugten Threads läuft, gehört die paintComponent-Methode in der wir zeichnen nicht zu diesem Thread.

Andererseits wurde bereits erwähnt, dass repaint() nicht sofort neu zeichnet, sondern nur vorschlägt, das Ganze doch bei Gelegenheit mal neu zu zeichnen.

Nun könnte es sein, dass während im einen Thread über den Vector iteriert wird, um diesen neu zu zeichnen, im anderen Thread ebenfalls auf den Vector zugegriffen wird und Objekte hinzugefügt oder gelöscht werden. Als Ergebnis würde uns eine ConcurrentModification um die Ohren fliegen (vgl. Google, API, Forum, etc.).

Daher werden wir zum Zeichnen eine Kopie unseres Vectors verwenden.  
Dazu benötigen wir zunächst eine weitere Instanz-Variable, wir benennen sie sprechend:

```
20  Sprite copter;  
21  Vector<Sprite> actors;  
22  Vector<Sprite> painter;  
23
```

Auch diesen Vector initialisieren wir bei Beginn:

```
43  private void doInitializations() {  
44  
45      last = System.nanoTime();  
46  
47      BufferedImage[] heli = loadPics("pics/heli.gif", 4);  
48  
49      actors = new Vector<Sprite>();  
50      painter = new Vector<Sprite>();  
51      copter = new Sprite(heli, 400, 300, 100, this);  
52      actors.add(copter);  
53  }
```

Nun müssen wir das Kopieren des Vectors nur noch in den GameLoop einfügen:

```
56  @Override  
57  public void run() {  
58  
59      while(frame.isVisible()){  
60  
61          computeDelta();  
62  
63          checkKeys();  
64          doLogic();  
65          moveObjects();  
66          cloneVectors();  
67  
68          repaint();  
69  
70          try {  
71              Thread.sleep(10);  
72          } catch (InterruptedException e) {}  
73  
74      }  
75  }  
76  
77  
78  @SuppressWarnings("unchecked")  
79  private void cloneVectors(){  
80      painter = (Vector<Sprite>) actors.clone();  
81  }
```

Ich habe mich hier dafür entschieden eine eigene Methode anzulegen, damit im GameLoop (fast) nur Methodenaufrufe stehen. Einfach, weil es mir so besser gefällt. Selbstverständlich könnte man das auch einfach vor dem repaint reinbasteln.

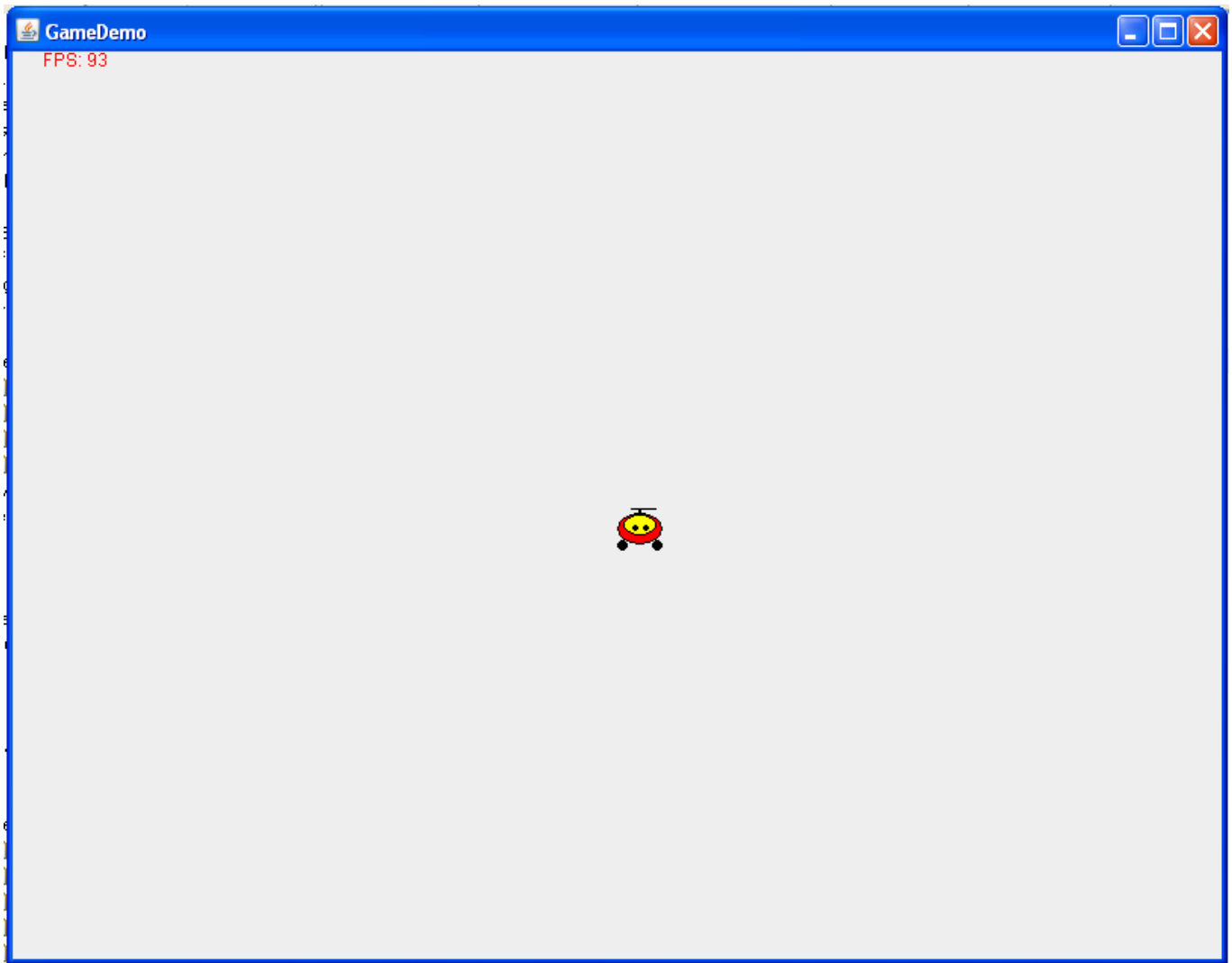
So, jetzt noch in der überschriebenen paintComponent-Methode die Objekte des Vectors zeichnen:

```
114 @Override
115 public void paintComponent(Graphics g) {
116     super.paintComponent(g);
117
118     g.setColor(Color.red);
119     g.drawString("FPS: " + Long.toString(fps), 20, 10);
120
121     if (painter != null) {
122         for (ListIterator<Sprite> it = painter.listIterator(); it.hasNext(); ) {
123             Sprite r = it.next();
124             r.drawObjects(g);
125         }
126     }
127
128 }
129
```

Da das Fenster erzeugt wird, bevor painter in doInitializations erzeugt wird, würde hier anfangs eine NullPointerException geworfen. Wir umgehen dies vorerst, indem wir prüfen, ob painter schon instanziiert wurde. Da mir diese Lösung aber nicht gefällt, bessern wir hier später noch nach.



***Und so sieht's aus:***



Dass der Hubschrauber jetzt schon animiert ist, sieht man hier natürlich nicht 😊

## Die Steuerung

Den Hubschrauber nur anzuzeigen ist aber nur der halbe Spaß. Jetzt wollen wir diesen auch noch über die Cursor-Tasten steuern können. Dafür benötigen wir 5 weitere Instanzvariablen:

```
21 Vector<Sprite> actors;  
22 Vector<Sprite> painter;  
23  
24 boolean up;  
25 boolean down;  
26 boolean left;  
27 boolean right;  
28 int speed = 50;  
29  
30 public static void main(  
31     // ... GamePanel 1/2000 600%
```

4 Booleans für die 4 Richtungen und eine Variable für die Geschwindigkeit der Bewegung. Für komplexere Spiele mag dies nicht die richtige Vorgehensweise sein, für unser kleines Spiel hier ist das aber ausreichend.

### Implementierung des KeyListeners

Um die o. g. Steuervariablen beeinflussen zu können, implementieren wir das KeyListener-Interface in unser GamePanel:

```
12  
13 public class GamePanel extends JPanel implements Runnable, KeyListener{  
14  
15     private static final long serialVersionUID = 1L;  
16     JFrame frame;  
17  
18     long delta = 0;
```

Außerdem müssen die entsprechenden Methoden angelegt werden, je nach IDE wird man dabei unterstützt oder muss dies manuell durchführen.

```
158 @Override  
159 public void keyPressed(KeyEvent e) {  
160  
161 }  
162  
163 @Override  
164 public void keyReleased(KeyEvent e) {  
165  
166 }  
167  
168 @Override  
169 public void keyTyped(KeyEvent e) {  
170
```

Jetzt noch im Konstruktor unseres GamePanels den KeyListener zu unserem Fenster hinzufügen und dann sind wir schon fast fertig:

```
36 public GamePanel(int w, int h) {
37     this.setPreferredSize(new Dimension(w,h));
38     frame = new JFrame("GameDemo");
39     frame.setLocation(100,100);
40     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
41     frame.add(this);
42     frame.addKeyListener(this);
43     frame.pack();
44     frame.setVisible(true);
45
46     doInitializations();
47
48     Thread th = new Thread(this);
49     th.start();
50 }
```

In der Methode keyPressed(KeyEvent e) setzen wir beim entsprechenden Tastsendruck die jeweilige Steuervariable auf true.

```
159 @Override
160 public void keyPressed(KeyEvent e) {
161
162     if(e.getKeyCode() == KeyEvent.VK_UP) {
163         up = true;
164     }
165
166     if(e.getKeyCode() == KeyEvent.VK_DOWN) {
167         down = true;
168     }
169
170     if(e.getKeyCode() == KeyEvent.VK_LEFT) {
171         left = true;
172     }
173
174     if(e.getKeyCode() == KeyEvent.VK_RIGHT) {
175         right = true;
176     }
177 }
```

Und machen dies bei `keyReleased(KeyEvent e)` entsprechend wieder rückgängig:

```
180 @Override
181 public void keyReleased(KeyEvent e) {
182
183     if (e.getKeyCode() == KeyEvent.VK_UP) {
184         up = false;
185     }
186
187     if (e.getKeyCode() == KeyEvent.VK_DOWN) {
188         down = false;
189     }
190
191     if (e.getKeyCode() == KeyEvent.VK_LEFT) {
192         left = false;
193     }
194
195     if (e.getKeyCode() == KeyEvent.VK_RIGHT) {
196         right = false;
197     }
```

Diese Vorgehensweise stellt sicher, dass unser Spiel auch richtig reagiert, wenn mehr als eine Taste gedrückt wird. Würden wir den Steuercode direkt in den `KeyListener`-Methoden hinterlegen, würde mit Sicherheit nicht jeder Tastendruck erkannt werden. Außerdem ist so sicher gestellt, dass etwas passiert solange eine Taste gedrückt wird.

## Die Methode checkKeys()

Innerhalb des GameLoop prüfen wir jetzt in der Methode checkKeys() die boolean-Werte ab und verändern die Geschwindigkeitsparameter unseres Sprites:

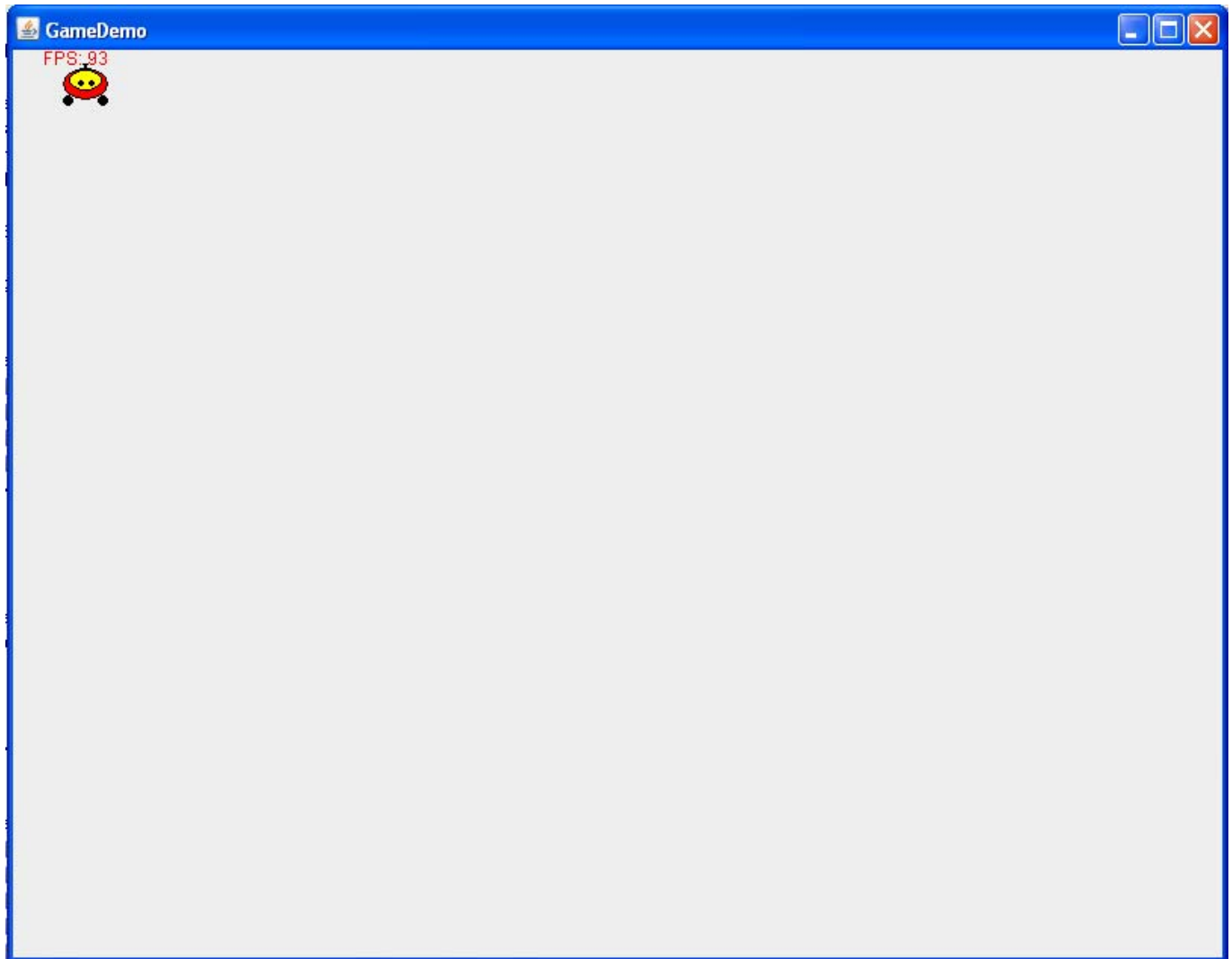
```
111 private void checkKeys() {  
112  
113     if(up) {  
114         copter.setVerticalSpeed(-speed);  
115     }  
116  
117     if(down) {  
118         copter.setVerticalSpeed(speed);  
119     }  
120  
121     if(right) {  
122         copter.setHorizontalSpeed(speed);  
123     }  
124  
125     if(left) {  
126         copter.setHorizontalSpeed(-speed);  
127     }  
128  
129     if(!up&&!down) {  
130         copter.setVerticalSpeed(0);  
131     }  
132  
133     if(!left&&!right) {  
134         copter.setHorizontalSpeed(0);  
135     }  
136  
137 }
```

Wichtig ist, auch zu prüfen ob die beiden gegenüberliegenden Werte beide false sind, um unser Objekt ggf. wieder anzuhalten. Es wäre hier auch denkbar in unserer Helikopter-Klasse komplexere Methoden bereit zu stellen, um z. B. den Hubschrauber nach und nach zu beschleunigen.

Für unser Anfänger-Spiel soll die normale Geschwindigkeitsänderung aber erst einmal ausreichen.

### ***Und so sieht's aus:***

Jetzt fliegt unser Heli nicht nur, wir können ihn auch beliebig auf dem GamePanel herum bewegen 😊



## Feinschliff

Als erstes wollen wir diese Abfrage in unserer paintComponent-Methode ändern, die aktuell prüft ob der Vektor painter nicht null ist. Dazu definieren wir uns eine Variabel vom Typ boolean:

```
26  boolean up;  
27  boolean down;  
28  boolean left;  
29  boolean right;  
30  boolean started;  
31  int speed = 50;  
32  
33  public static void main(String[] args) {
```

Diese setzen wir in unserem Universal-Initialisierer auf true. ☺

```
53  private void doInitializations() {  
54  
55      last = System.nanoTime();  
56  
57      BufferedImage[] heli = loadPics("pics/heli.gif", 4);  
58  
59      actors = new Vector<Sprite>();  
60      painter = new Vector<Sprite>();  
61      copter = new Sprite(heli, 400, 300, 100, this);  
62      actors.add(copter);  
63  
64      started = true;  
65  }
```

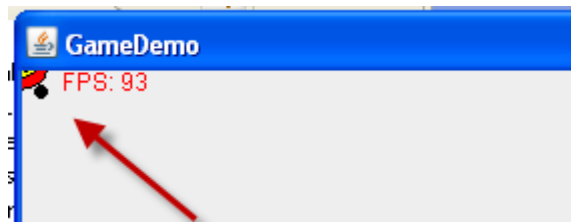
Wir entfernen die if-Bedingung um unseren List-Iterator und fragen stattdessen unsere neue Variable ab:

```
152  @Override  
153  public void paintComponent(Graphics g) {  
154      super.paintComponent(g);  
155  
156      g.setColor(Color.red);  
157      g.drawString("FPS: " + Long.toString(fps), 20, 10);  
158  
159  
160      if(!started){  
161          return;  
162      }  
163  
164      for (ListIterator<Sprite> it = painter.listIterator(); it.hasNext();) {  
165          Sprite r = it.next();  
166          r.drawObjects(g);  
167      }  
168  
169  }
```

So sieht das Ganze etwas "runder" aus. Zumindest gefällt mir das besser, als die Abfrage ob painter nicht null ist.

## Es wird abstrakt

Momentan können wir unseren Hubschrauber problemlos aus dem Bild heraus fliegen.



Das soll sich nun ändern. Da diese Logik aber ausschließlich für den Hubschrauber gelten soll und nicht für spätere Objekte, wollen wir die Klasse `Sprite` ab jetzt nur noch als Basis-Klasse verwenden, die erweitert bzw. vererbt, aber nicht mehr instanziiert wird.

Dies erreichen wir ganz einfach, indem wir die Klasse `Sprite` `abstract` setzen.

```
1 import java.awt.Graphics;
4
5 public abstract class Sprite extends Rectangle2D.Double implements Drawable, Movable{
6
7     private static final long serialVersionUID = 1L;
8     long delay;
9     long animation = 0;
10    GamePanel parent;
11
```

Je nach verwendeter IDE, erhalten wir sofort oder spätestens beim Kompilieren unserer Klassen eine Fehlermeldung, weil für diese Klasse jetzt kein Objekt mehr erzeugt werden darf. Daher legen wir uns eine Klasse `Heli` an, die alle Eigenschaften von `Sprite` erbt!

```
3
4 public class Heli extends Sprite {
5
6     private static final long serialVersionUID = 1L;
7
8     public Heli(BufferedImage[] i, double x, double y, long delay, GamePanel p) {
9         super(i, x, y, delay, p);
10    }
11
12
13 }
14
```

Wichtig ist dabei, den Konstruktor der Elternklasse aufzurufen (Zeile 9), damit alle Werte schön gesetzt werden.



Zusätzlich müssen wir natürlich noch die Klasse GamePanel anpassen und alle bisher verwendeten Objekte vom Typ Sprite in Objekte vom Typ Heli umbenennen:

Einmal hier:

```
13 public class GamePanel extends JPanel implements Runnable, KeyListener{
14
15     private static final long serialVersionUID = 1L;
16     JFrame frame;
17
18     long delta = 0;
19     long last = 0;
20     long fps = 0;
21
22     Heli copter;
23     Vector<Sprite> actors;
24     Vector<Sprite> painter;
25 }
```

Und einmal hier:

```
53 private void doInitializations() {
54
55     last = System.nanoTime();
56
57     BufferedImage[] heli = loadPics("pics/heli.gif", 4);
58
59     actors = new Vector<Sprite>();
60     painter = new Vector<Sprite>();
61     copter = new Heli(heli, 400, 300, 100, this);
62     actors.add(copter);
63
64     started = true;
65 }
```

Das war's. Jetzt ist es problemlos möglich, individuelle Logik für unseren Hubschrauber zu hinterlegen!

## Die neue Klasse Heli

An dieser neuen Klasse Heli zeigt sich nun, dass es durch den Aufwand, den wir bis jetzt mit der Klasse Sprite getrieben haben, relativ einfach ist, neue Objekte in unser Spiel einzufügen: Einfach von Sprite erben, den Konstruktor anpassen und gut ist's. ☺ Klingt einfach? Ist es auch!

Der Vorteil ist, dass wir jetzt durch Überschreiben von Methoden individuell Logik hinterlegen können. Das wollen wir für unseren Heli gleich mal tun und verhindern, dass er aus dem Fenster fliegt (das auf dem Bildschirm).

```
4 public class Heli extends Sprite {
5
6     private static final long serialVersionUID = 1L;
7
8     public Heli(BufferedImage[] i, double x, double y, long delay, GamePanel p) {
9         super(i, x, y, delay, p);
10    }
11
12    @Override
13    public void doLogic(long delta) {
14        super.doLogic(delta);
15
16        if(getX() < 0) {
17            setHorizontalSpeed(0);
18            x = 0;
19        }
20
21        if(getY() < 0) {
22            setHorizontalSpeed(0);
23            y = 0;
24        }
25
26        if(getX() + getWidth() > parent.getWidth()) {
27            setVerticalSpeed(0);
28            x = parent.getWidth() - getWidth();
29        }
30
31        if(getY() + getHeight() > parent.getHeight()) {
32            setVerticalSpeed(0);
33            y = parent.getHeight() - getHeight();
34        }
35    }
36 }
37
38 }
```

So sieht die überschriebene doLogic-Methode in unserer Klasse Heli aus. Wichtig ist, dass der Aufruf der Methode für die Elternklasse nicht vergessen wird (Zeile 14), damit die Logik, die wir in der Klasse Sprite für die Animation hinterlegt haben, auch weiter ausgeführt wird.

Diese Prüfung wird uns dadurch erleichtert, dass unsere Objekte von Rectangle2D erben, so dass wir die maximalen Dimensionen unserer Objekte leicht abprüfen können.

**Zeile 16 – 18:**

Ist die x-Position unseres Objekts kleiner Null verlässt der Heli die Zeichenfläche nach links. Das verhindern wir indem wir die Geschwindigkeit auf Null setzen und x (geerbt von Rectangle2D.Double) ebenfalls Null setzen. Damit hält der Heli direkt am linken Rand an.

**Zeile 21 – 23:**

Das Gleiche nur in vertikaler Richtung.

**Zeile 26 ff:**

Hier das Gleiche in die andere Richtung. Um bei späteren Änderungen keine Fehler zu provozieren, sollte nicht auf Festwerte geprüft werden, sondern vorhandene (soll heißen geerbte) Methoden benutzt werden.

Damit der Heli exakt anhält, muss man die Breite des Sprites berücksichtigen. Da wir von Rectangle2D erben, können wir diese mit getWidth() abfragen.

## ***Änderung des Programm-Starts***

Sehr unschön ist momentan auch noch, dass unser Spiel sofort losläuft, wenn wir das Programm starten. Wir wollen das Ganze nun so ändern, dass das Spiel mit der Enter-Taste gestartet wird und mit der Escape-Taste abgebrochen bzw. komplett beendet werden kann. Dazu bedienen wir uns der booleanschen Variable started, die wir bereits eingebaut haben.

Für komplexere Spiele (mit Intro, Ladebildschirm, etc.) wird man hier eine andere Lösung bevorzugen, für unser kleines Tutorial ist ein einzelner Boolean aber ausreichend.

Für diese Variable definieren wir uns auch noch die get- und set-Methoden in unserem GamePanel:

```
191 public boolean isStarted() {  
192     return started;  
193 }  
194  
195 public void setStarted(boolean started) {  
196     this.started = started;  
197 }
```

Außerdem müssen wir den bisherigen Code noch etwas modifizieren:

```
68 @Override
69 public void run() {
70
71     while (frame.isVisible()) {
72
73         computeDelta();
74
75         if (isStarted()) {
76             checkKeys();
77             doLogic();
78             moveObjects();
79             cloneVectors();
80         }
81
82         repaint();
```

## Die Methoden

checkKeys()  
doLogic()  
moveObjects() und  
cloneVectors()

packen wir in eine if-Bedingung, die unsere neue Variable via get-Methode abfragt.

Das computeDelta() lassen wir außen vor. Dies ist aber eher eine subjektive Vorliebe, ähnlich wie die Anzeige der fps, die durch diesen Methodenaufruf ermöglicht wird. ☺  
Wichtig ist, dass die 3 oben genannten Methoden erst abgearbeitet werden, wenn unser Spiel gestartet ist, andernfalls wäre es möglich, dass Objekte schon vor dem Spielstart beeinflusst werden.

Jetzt müssen wir nur noch dafür sorgen, dass die Spielstatus-Variable auch auf Tastendruck verändert werden kann.

Zunächst einmal werden wir den doInitializations-Aufruf aus dem Konstruktor entfernen:

```
37 public GamePanel(int w, int h) {
38     this.setPreferredSize(new Dimension(w, h));
39     frame = new JFrame("GameDemo");
40     frame.setLocation(100, 100);
41     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
42     frame.add(this);
43     frame.addKeyListener(this);
44     frame.pack();
45     frame.setVisible(true);
46
47     doInitializations();
48
49     Thread th = new Thread(this);
50     th.start();
51 }
```

Anschließend müssen wir in unserem ActionListener noch Code einfügen – es empfiehlt sich die Methode `keyReleased(..)` da diese eindeutig ist.

```
223 public void keyReleased(KeyEvent e) {
224
225     if(e.getKeyCode() == KeyEvent.VK_UP) {
226         up = false;
227     }
228
229     if(e.getKeyCode() == KeyEvent.VK_DOWN) {
230         down = false;
231     }
232
233     if(e.getKeyCode() == KeyEvent.VK_LEFT) {
234         left = false;
235     }
236
237     if(e.getKeyCode() == KeyEvent.VK_RIGHT) {
238         right = false;
239     }
240
241     if(e.getKeyCode() == KeyEvent.VK_ENTER) {
242         if(!isStarted()) {
243             doInitializations();
244             setStarted(true);
245         }
246     }
247
248     if(e.getKeyCode() == KeyEvent.VK_ESCAPE) {
249         if(isStarted()) {
250             setStarted(false);
251         } else {
252             frame.dispose();
253         }
254     }
255 }
```

#### **Zeile 241 ff:**

Ab hier wird der Code für das Drücken (bzw. Richtigerweise: das Loslassen) der Enter-Taste hinterlegt. Wenn unser Spiel noch nicht gestartet ist, wird unsere Initialisierungs-Methode aufgerufen und der boolean-Wert auf true gesetzt. Wichtig: Erst initialisieren, dann start auf true setzen, andernfalls kann es im GameLoop zu NullPointerExceptions kommen, weil die Collections ggf. noch nicht instanziiert wurden.

#### **Zeile 248 ff:**

Hier wird der Code für die Escape-Taste hinterlegt. Läuft das Spiel, wird es gestoppt. Ist das Spiel schon gestoppt, schließen wir da Fenster, damit alles sauber beendet wird.

#### **Anmerkung:**

Mit dieser Vorgehensweise vernachlässigen wir, dass bei jedem Aufruf von `doInitializations()` unsere Bild-Dateien unnötigerweise neu geladen werden.

Dies lassen wir vorerst unberücksichtigt, da aufgrund des geringen Umfangs der Dateien, dies nicht sonderlich ins Gewicht fallen wird. Bei komplexeren/fortgeschritteneren Spielen wird man die Grafiken dann sowieso anders verwalten (i.d.R. mit einer eigenen Klasse, die Grafiken lädt und speichert bzw. verwaltet)

Und noch eine letzte Änderung für den Feinschliff:

```
36
37 public GamePanel(int w, int h){
38     this.setPreferredSize(new Dimension(w,h));
39     this.setBackground(Color.BLUE);
40     frame = new JFrame("GameDemo");
41     frame.setLocation(100,100);
42     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
43     frame.add(this);
44     frame.addKeyListener(this);
45 }
```

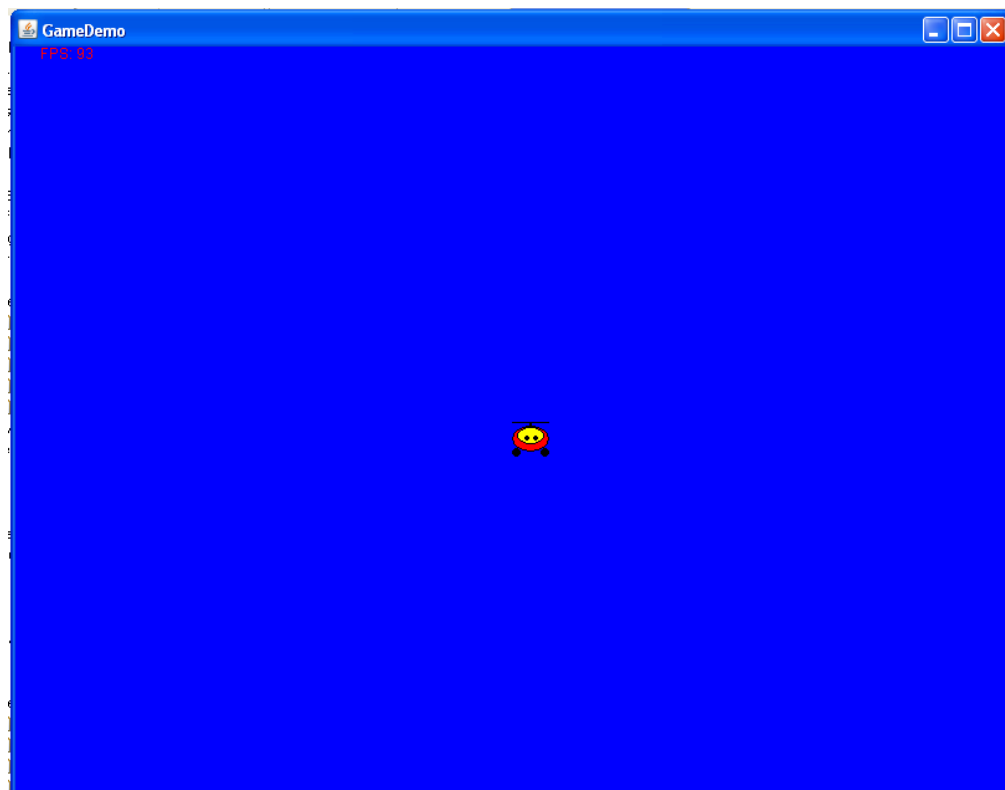
Schließlich fliegen wir am Himmel. ☺

## ***Und so sieht's aus:***

Vor dem Drücken der Enter-Taste:



Und nach dem Drücken der Enter-Taste:



## Weitere Objekte

Nun wollen wir das Ganze mal etwas mit Leben füllen. Wir wollen ein paar Wolken einfügen, die über den Bildschirm wandern, um das Ganze etwas aufzuhübschen☺. Wenn Sie auf der einen Seite verschwunden sind, sollen Sie zur anderen Seite wieder rein kommen. Dazu erstellen wir eine neue Klasse, die von Sprite erbt. Innerhalb dieser Klasse überschreiben wir lediglich die Methode doLogic() um die Logik für das Verlassen des Bildschirms zu hinterlegen.

### Die Klasse Cloud

```
4 public class Cloud extends Sprite{
5
6     private static final long serialVersionUID = 1L;
7     final int SPEED = 20;
8
9     public Cloud(BufferedImage[] i, double x, double y, long delay, GamePanel p) {
10         super(i, x, y, delay, p);
11     }
12
13
14     @Override
15     public void doLogic(long delta) {
16         super.doLogic(delta);
17     }
18
19
20
21
22 }
```

Dazu erzeugen wir die Klasse Cloud, die von Sprite erbt.

Zu dieser Kind-Klasse von Sprite fügen wir jetzt noch folgenden Code hinzu:

Als erstes etwas Logik im Konstruktor zur Bestimmung der Richtung in die sich die Wolke bewegt (das überlassen wir dem Zufall).

```
4 public class Cloud extends Sprite{
5
6     private static final long serialVersionUID = 1L;
7     final int SPEED = 20;
8
9     public Cloud(BufferedImage[] i, double x, double y, long delay, GamePanel p) {
10         super(i, x, y, delay, p);
11
12         if((int)(Math.random()*2)<1){
13             setHorizontalSpeed(-SPEED);
14         }else{
15             setHorizontalSpeed(SPEED);
16         }
17
18     }
```

#### Zeile 7:

Alle Wolken sind gleich schnell: 20 Pixel pro Sekunde.



**Zeile 12 ff:**

Abhängig vom Zufall bewegt sich die Wolke nach links oder rechts.

Weiterhin benötigen wird die Logik, mit der wir prüfen, ob sich die Wolke komplett außerhalb des sichtbaren Bereichs befindet. Ist das der Fall, versetzen wir sie auf die andere Seite – auch außerhalb des sichtbaren Bereichs. Dazu überschreiben wir, wie bei der Klasse Heli, die doLogic-Methode (super-Aufruf nicht vergessen!) und passen diese an:

```
20 @Override
21 public void doLogic(long delta) {
22     super.doLogic(delta);
23
24     if(getHorizontalSpeed()>0 && getX()>parent.getWidth()){
25         x = -getWidth();
26     }
27
28     if(getHorizontalSpeed()<0 && (getX()+getWidth()<0)){
29         x = parent.getWidth()+getWidth();
30     }
31
32 }
```

**Zeile 24 ff:**

Bewegt sich die Wolke nach rechts (Horizontalgeschwindigkeit > 0) und ist der x-Wert der Wolke größer als die Breite unseres GamePanels (d. h. die Wolke hat sich aus dem Bildschirm bewegt), setzen wir x so, dass der rechte Rand der Wolke noch außerhalb des Bildschirms liegt. Damit kann sie dann von links wieder rein wandern.

**Zeile 28 ff:**

Das Ganze für die Gegenrichtung...

Jetzt fehlen noch 2 Dinge:

1. die Wolke
2. ein bisschen Code, mit dem wir die Wolken-Objekte in unseren Vektor im GamePanel werfen

Das wär's. ☺

Die Wolke pinseln wir mal schnell mit der Sprühdose aus einem beliebigen Grafikprogramm – hier mit MS Paint:



Jetzt noch der restliche Code in unserem GamePanel:

```
53 private void doInitializations() {  
54  
55     last = System.nanoTime();  
56  
57     BufferedImage[] heli = loadPics("pics/heli.gif", 4);  
58  
59     actors = new Vector<Sprite>();  
60     painter = new Vector<Sprite>();  
61     copter = new Heli(heli, 400, 300, 100, this);  
62     actors.add(copter);  
63  
64     createClouds();  
65  
66     started = false;  
67 }  
68  
69 private void createClouds() {  
70  
71     BufferedImage[] bi = loadPics("pics/cloud.gif", 1);  
72  
73     for(int y=10; y<getHeight(); y+=50) {  
74         int x = (int) (Math.random() * getWidth());  
75         Cloud cloud = new Cloud(bi, x, y, 1000, this);  
76         actors.add(cloud);  
77     }  
78  
79 }  
80
```

#### Zeile 64:

In der Methode doInitializations() fügen wir einen weiteren Methodenaufruf ein: createClouds(). Hier werden wir die Wolken für das Spiel erzeugen. Nach dem erweitern von doInitializations() müssen wir noch die Methode erstellen.

#### Zeile 69:

Hier wird das GIF geladen – genau so, wie wir es für den Hubschrauber schon umgesetzt haben

#### Zeile 73:

Beginnend bei 10 setzen wir in 50er-Schritten Wolken über die gesamte Höhe des Spielfeldes.

**Zeile 74:**

Die x-Position lassen wir über einen Zufallswert erzeugen, damit diese schön horizontal verteilt werden.

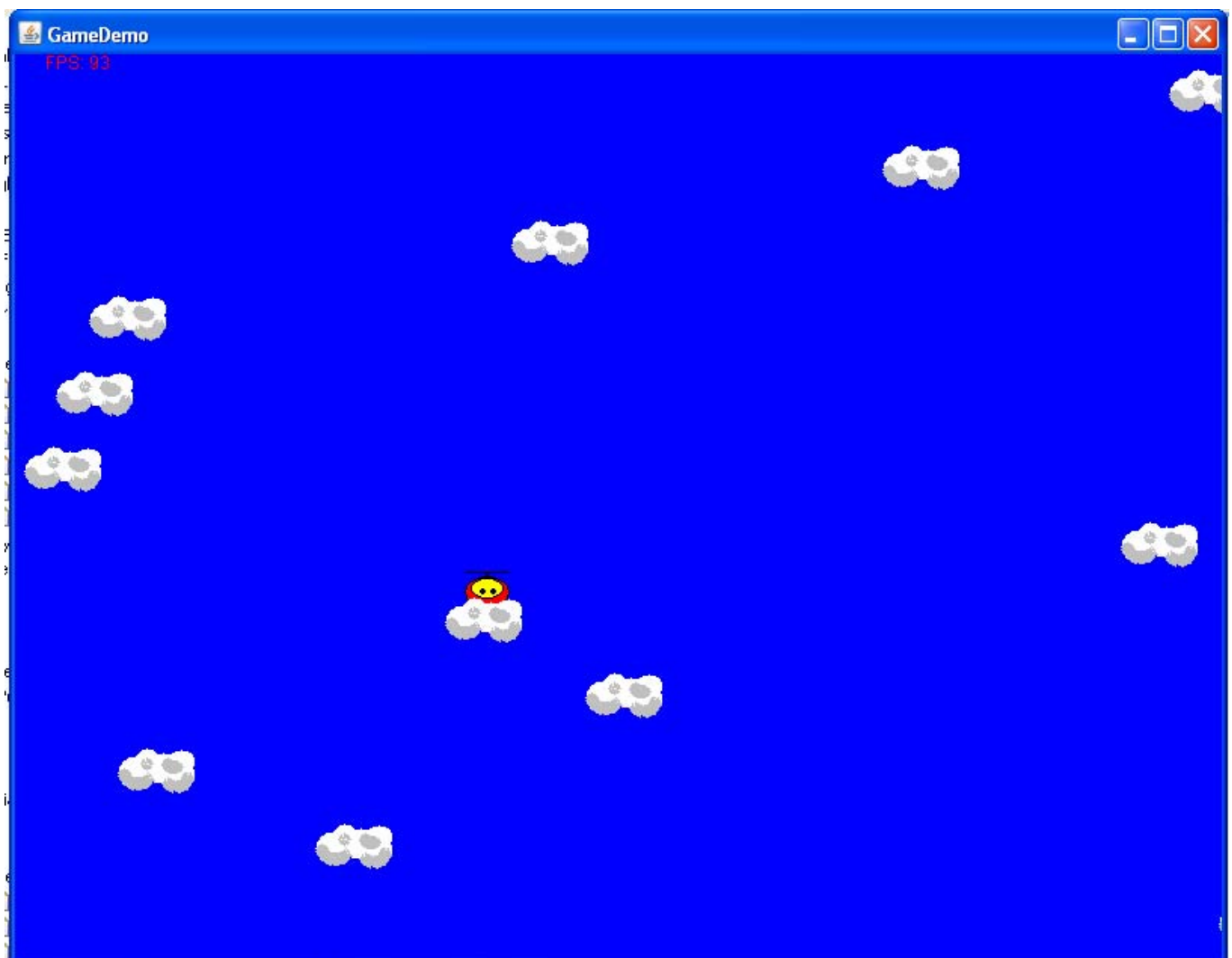
**Zeile 75:**

Hier wird unser Cloud-Objekt instanziiert – analog zu unserem Hubschrauber.

**Zeile 76:**

Dann das Objekt in die Collection packen – fertig. An dieser Stelle arbeiten wir noch ohne Iterator, da noch niemand parallel auf die Collections zugreifen kann. Später werden wir für diese Operation einen Iterator verwenden.

Alles andere (Logik-Aufrufe, Bewegungen, Zeichnen) haben wir schon lange in unserem GameLoop implementiert. An dieser Stelle sparen wir uns durch den Aufwand mit den Interfaces, der abstrakten Klasse, etc. einiges an Arbeit, da nach einem Programmstart unsere Wolken jetzt problemlos eingefügt wurden:

***Und so sieht's aus:***

Anmerkung:

Es ist an dieser Stelle beabsichtigt, dass der Hubschrauber hinter den Wolken verschwindet. Wem dies nicht gefällt, der muss sicherstellen, dass alle Objekte vor dem Hubschrauber in die Collection eingefügt werden, damit der Hubschrauber als letztes gezeichnet wird!

## Noch mehr Objekte

So langsam wird es Zeit, das Ganze etwas interessanter zu machen. Daher wollen wir nun ein paar Feinde in unser Spiel einbinden. Im Prinzip funktioniert das nicht anders, als wir es bisher mit allen Objekten gemacht haben:

1. Objekt von Sprite erben lassen
2. zusätzliche Logik einbauen
3. Instanzieren
4. In die Collection packen, die unsere Spielobjekte enthält

Klingt einfach, oder? Ist es auch ☺

Das nächste Objekt soll nun eine Rakete sein, die unseren Hubschrauber zerstören könnte. Die entsprechend Grafik für die Animation sieht so aus:



Dabei fällt auf, dass die Grafikdatei 2 Animationsreihen enthält, um nicht für jede Bewegungsrichtung ein GIF ablegen zu müssen (man denke an den Aufwand für komplexe/flüssigere Animationen und/oder detailliertere Richtungsänderungen).

Für unser erstes Spiel halten wir es einfach:

- 4 Bilder für die Bewegung nach links und
- 4 Bilder für die Bewegung nach rechts.

## Anpassen der Klasse Sprite

Dafür wird es jetzt aber notwendig, die Animationsroutine zu ändern, so dass wir unseren Objekten mitteilen können, nur über einen bestimmten Bereich des Bildarrays zu „loopen“.

Da diese Funktion für alle Objekte sinnvoll sein kann, realisieren wir sie in der Klasse Sprite, so dass sie für alle Objekte zur Verfügung steht.

```
5 public abstract class Sprite extends Rectangle2D.Double implements Drawable, Movable{
6
7     private static final long serialVersionUID = 1L;
8     long delay;
9     long animation = 0;
10    GamePanel parent;
11    BufferedImage[] pics;
12    int currentpic = 0;
13
14    protected double dx;
15    protected double dy;
16
17    int loop_from;
18    int loop_to;
19
20
21    public Sprite(BufferedImage[] i, double x, double y, long delay, GamePanel p ){
22        pics = i;
23        this.x = x;
24        this.y = y;
25        this.delay = delay;
26        this.width = pics[0].getWidth();
27        this.height = pics[0].getHeight();
28        parent = p;
29        loop_from = 0;
30        loop_to = pics.length-1;
31    }
```

### Zeile 17 + 18:

Definition der Variablen, die Beginn und Ende des zu animierenden Intervalls angeben.

### Zeile 29 + 30:

Im Konstruktor werden diese Variablen standardisiert belegt. Da ein Array mit 0 beginnt, erhält loop\_from den Wert Null. Entsprechend erhält bei dieser Vorgehensweise die Variable loop\_to einen Wert, der der Länge des Arrays -1 entspricht (weil wir ja bei Null zu zählen beginnen und nicht bei 1).

Nun müssen wir noch die Animations-Routine anpassen und schon können wir aus einer Datei unterschiedliche Animationssequenzen verwenden.

```
37 public void doLogic(long delta) {
38
39     animation += (delta/1000000);
40     if (animation > delay) {
41         animation = 0;
42         computeAnimation();
43     }
44 }
45
46 private void computeAnimation(){
47
48     currentpic++;
49
50     if(currentpic>loop_to){
51         currentpic = loop_from;
52     }
53
54 }
55
56
57
58 public void setLoop(int from, int to){
59     loop_from = from;
60     loop_to = to;
61     currentpic = from;
62 }
63
```

#### **Zeile 51-53:**

Hier haben wir in der bereits vorhandenen Methode computeAnimation() die Bedingung so angepasst, dass loop\_from und loop\_to berücksichtigt werden und ggf. nicht mehr das gesamte Bild-Array.

#### **Zeile 68-72:**

Zusätzlich wurde eine neue Methode implementiert, mit der das Animations-Intervall verändert werden kann. Damit es keine logischen „Kollisionen“ gibt, wird hier jedes Mal die Variable currentpic auf den Beginn des Intervalls gesetzt.

## Die Klasse Rocket

Jetzt wollen wir unsere Rakete ins Leben rufen, damit es für den Heli etwas gefährlicher wird. Die Rakete soll folgende Eigenschaften haben:

1. Sie soll zufällig von links oder rechts kommen
2. Abhängig von der Startposition soll sie sinken oder steigen
3. Wenn Sie den Hubschrauber erfasst hat, soll sie diesen „verfolgen“

Auch diese Klasse erbt von Sprite, somit müssen wir hauptsächlich wieder die doLogic-Methode überschreiben um das Verhalten der Rakete zu beeinflussen.

Zusätzlich möchte ich hier die Bewegung des Objektes erst nach dessen Erzeugung festlegen – einfach um mal eine unterschiedliche Vorgehensweise im Vergleich zur Klasse Cloud zu zeigen. Daher werden wir noch die Methoden zur Festlegung der Geschwindigkeit überschreiben, damit dort dann entschieden werden kann, welche Bilder des Arrays zu verwenden sind.

Der Rumpf der neuen Klasse sieht dann so aus:

```
4 public class Rocket extends Sprite{
5
6     private static final long serialVersionUID = 1L;
7
8     public Rocket(BufferedImage[] i, double x, double y, long delay, GamePanel p) {
9         super(i, x, y, delay, p);
10    }
11
12
13    @Override
14    public void doLogic(long delta) {
15        super.doLogic(delta);
16    }
17
18
19    public void setHorizontalSpeed(double d) {
20        super.setHorizontalSpeed(d);
21    }
22
23 }
```

Nun beginnen wir, das Verhalten der Rakete zu programmieren.

```
5 public class Rocket extends Sprite{
6
7     private static final long serialVersionUID = 1L;
8
9     int verticalspeed = 70;
10    Rectangle2D.Double target;
11    boolean locked = false;
12
13    public Rocket(BufferedImage[] i, double x, double y, long delay, GamePanel p) {
14        super(i, x, y, delay, p);
15
16        if (getY() < parent.getHeight() / 2) {
17            setVerticalSpeed(verticalspeed);
18        } else {
19            setVerticalSpeed(-verticalspeed);
20        }
21    }
22 }
```

**Zeile 9:**

Variable für die Sinkgeschwindigkeit der Rakete.

**Zeile 10:**

Instanzvariable für die Zielverfolgung, dazu später mehr

**Zeile 11:**

Instanzvariable, ob die Rakete ein Ziel erfasst hat, auch dazu später mehr

**Zeile 16 – 19:**

Hier entscheiden wir, ob sich die Rakete nach oben oder unten bewegen soll.

Ich hatte mir hier überlegt einmal die Geschwindigkeit bewusst nicht als Variable zu hinterlegen, so dass jeder der hier noch ein bisschen Tuning vornehmen möchte, einmal die Unterschiede erkennt (eine Variable ändern vs. alle Zahlen ändern). Aber da dies eine unschöne Vorgehensweise ist und nicht jeder das Tutorial detailliert liest, habe ich es gelassen.



## Die Logic-Methode:

```
23 @Override
24 public void doLogic(long delta) {
25     super.doLogic(delta);
26
27     if(getHorizontalSpeed()>0){
28         target = new Rectangle2D.Double(getX()+getWidth(),getY(),
29             parent.getWidth()-getX(),getHeight());
30     }else{
31         target = new Rectangle2D.Double(0,getY(),getX(),getHeight());
32     }
33
34     if(!locked&&parent.copter.intersects(target)){
35         setVerticalSpeed(0);
36         locked = true;
37     }
38
39     if(locked){
40         if(getY()<parent.copter.getY()){
41             setVerticalSpeed(40);
42         }
43         if(getY()>parent.copter.getY()+parent.copter.getHeight()){
44             setVerticalSpeed(-40);
45         }
46     }
```

### Zeile 27 - 31:

Abhängig von der Bewegungsrichtung erzeugen wir hier ein Rechteck, welches den Raum von der Vorderseite der Rakete bis zum Bildschirm-Rand beinhaltet.

### Zeile 34 – 36:

Wenn die Rakete noch kein Ziel erfasst hat (locked = false), sich unser Helikopter aber im gerade berechneten Rechteck befindet, beenden wir die vertikale Bewegung und setzen das Flag "locked". Hier zeigt sich jetzt wieder der Vorteil aus der Verwendung der Klasse Rectangle2D als Basisklasse. Für diese einfache „Kollisionserkennung“ müssen wir keinen weiteren Code implementieren, sondern können auf vorhandene Method zugreifen – hier: intersects(Rectangle r).

### Zeile 39 – 45:

Wenn die Rakete ein Ziel „erfasst“ hat, passt sie Ihre Höhe an den Hubschrauber an. Die vertikale Veränderung ist hier geringer gewählt, damit unser Hubschrauber noch eine Chance hat. (Sollte dieser Code von Rüstungskonzernen verwendet werden, sind hier Anpassungen nötig ☺).

Zu guter Letzt müssen wir noch die Methode `setHorizontalSpeed(double d)` überschreiben, damit das richtige Bildintervall angezeigt wird.

```
51 public void setHorizontalSpeed(double d) {  
52     super.setHorizontalSpeed(d);  
53  
54     if (getHorizontalSpeed() > 0) {  
55         setLoop(4, 7);  
56     } else {  
57         setLoop(0, 3);  
58     }  
59 }
```

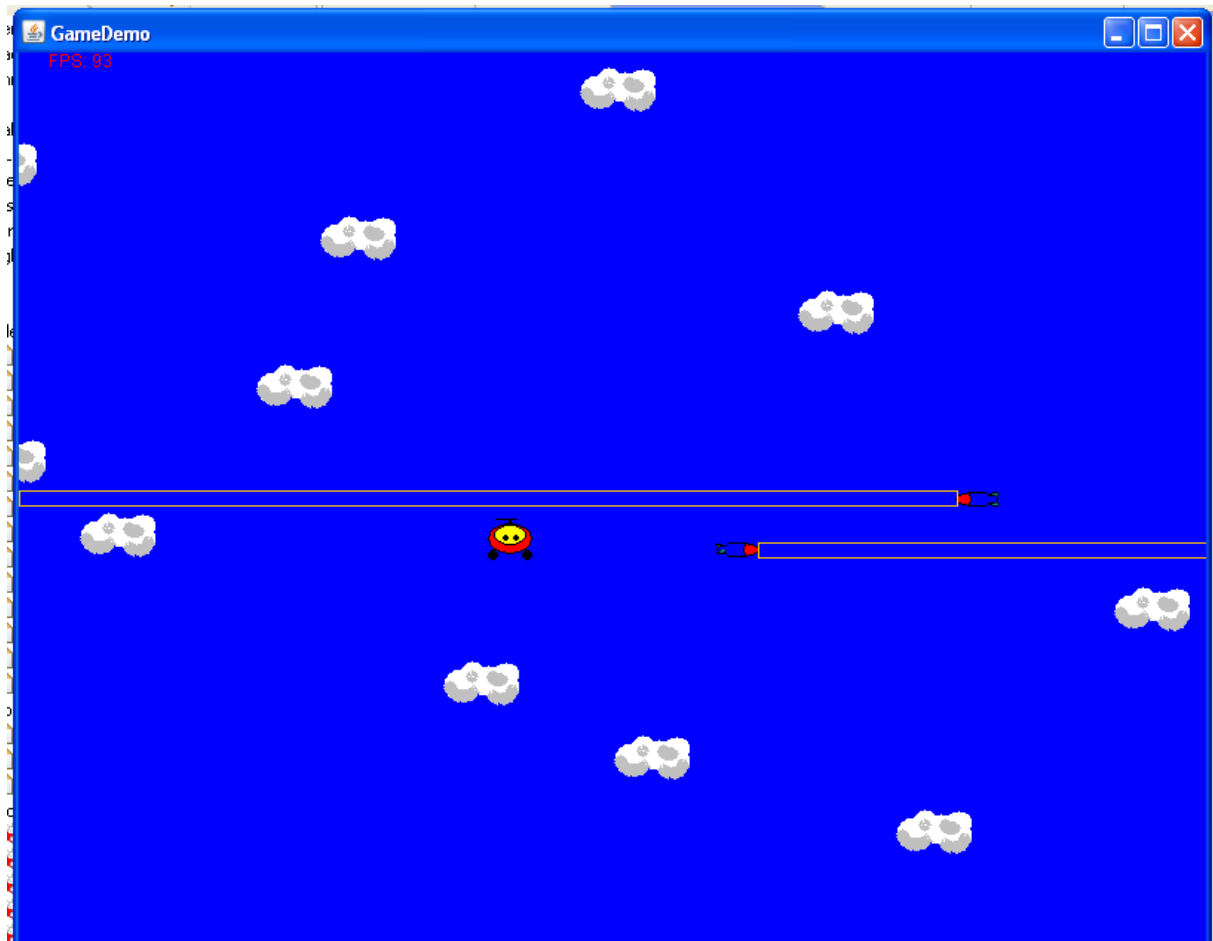
#### **Zeile 54 – 57:**

Abhängig von der Bewegungsrichtung, wird ein anderes Bildintervall gesetzt. Dabei ist zu beachten, dass Arrays bei Null mit dem zählen beginnen!

Für den Test des „Zielrechtecks“ könnte man noch die `paint`-Methode überschreiben und diese anzeigen lassen. Dies ist dann sinnvoll, wenn der Code nicht wie gewünscht funktioniert.

```
54 @Override  
55 public void drawObjects(Graphics g) {  
56     super.drawObjects(g);  
57     g.setColor(Color.ORANGE);  
58     g.drawRect((int) target.x, (int) target.y, (int) target.width, (int) target.height);  
59 }
```

Das Ergebnis (testhalber) sähe dann so aus:



## Modifikationen der Klasse GamePanel:

Über einen Timer soll alle 3 Sekunden eine Rakete erzeugt werden. Dazu verwenden wir den Timer aus dem Paket javax.swing. Es muss dabei darauf geachtet werden, dass das richtige Package importiert wird, da es in der API mehrere Klassen namens Timer gibt.

Da der Timer einen ActionEvent erzeugt, müssen wir noch das entsprechende Interface importieren. Zudem werden der Timer und ein Array vom Typ BufferedImage als Klassenvariable definiert. Dies ist notwendig damit der Timer aus anderen Methoden heraus gestoppt werden kann und damit wir die Bilddaten für die Rakete zentral ablegen können.

Würden wir eine eigene Klasse zur Verwaltung der Bilddaten verwenden könnten wir uns dies sparen. Da wir aber bei jeder Erzeugung eines neuen Rocket-Objekts auf die Bilddaten zugreifen müssen, würde es uns einiges an Performance kosten, wenn wir diese jedes Mal laden müssten.

```
15 public class GamePanel extends JPanel implements Runnable, ActionListener{
16
17     private static final long serialVersionUID = 1L;
18     JFrame frame;
19
20     long delta = 0;
21     long last = 0;
22     long fps = 0;
23
24     Heli copter;
25     Vector<Sprite> actors;
26     Vector<Sprite> painter;
27
28     boolean up;
29     boolean down;
30     boolean left;
31     boolean right;
32     boolean started;
33     int speed = 50;
34
35     Timer timer;
36     BufferedImage[] rocket;
```

### Zeile 15:

Implementierung des Interfaces ActionListener

### Zeile 35:

Timer-Klasse als Instanzvariable:

### Zeile 36:

BufferedImage-Array als Instanzvariable

Anschließend noch die „nackte“ Methode für den ActionListener

```
281 @Override
282 public void actionPerformed(ActionEvent e) {
283
284 }
285
```

## Modifikation in doInitializations():

```
58 private void doInitializations() {  
59  
60     last = System.nanoTime();  
61  
62     BufferedImage[] heli = loadPics("pics/heli.gif", 4);  
63     rocket = loadPics("pics/rocket.gif", 8);  
64  
65     actors = new Vector<Sprite>();  
66     painter = new Vector<Sprite>();  
67     copter = new Heli(heli, 400, 300, 100, this);  
68     actors.add(copter);  
69  
70     createClouds();  
71  
72     timer = new Timer(3000, this);  
73     timer.start();  
74  
75     started = false;  
76 }
```

In der Methode doInitializations füllen wir die zusätzlichen Objekte mit Leben:

### Zeile 63:

Hier laden wir die Bilddaten, mit unserer selbst erstellten Methode. Dies wird dann allerdings bei jedem Spielstart durchgeführt und damit spätestens beim 2. Mal unnötigerweise. An dieser Stelle schmerzt das allerdings nicht so sehr, so dass wir dies für unser erstes Spiel vernachlässigen. Wie bereits öfter bemerkt sollte man bei größeren Spielen eine eigene „Sprite-Lib“ programmieren um das unnötige Laden von Bilddaten zu vermeiden.

### Zeile 72 – 72:

Hier wird das Timer-Objekt erzeugt und gestartet. Intervall: 3 Sekunden.

## Das Erzeugen der Raketen:

Wir benötigen noch eine Methode, die wir zum Erzeugen der Raketen aufrufen:

```
90 private void createRocket(){
91
92     int x = 0;
93     int y = (int) (Math.random() * getHeight());
94     int hori = (int) (Math.random() * 2);
95
96     if(hori==0){
97         x = -30;
98     }else{
99         x = getWidth()+30;
100     }
101
102
103     Rocket rock = new Rocket(rocket, x, y, 100, this);
104     if(x<0){
105         rock.setHorizontalSpeed(100);
106     }else{
107         rock.setHorizontalSpeed(-100);
108     }
109
110     ListIterator<Sprite> it = actors.listIterator();
111     it.add(rock);
112 }
```

### Zeile 92:

Definition der Variable für die x-Position. Ist hier erstmal Null.

### Zeile 93:

Die Höhe generieren wir über eine Zufallsfunktion

### Zeile 94:

Zufallszahl (0 oder 1) um zu entscheiden, ob die Rakete von Links oder Rechts einfliegen soll.

### Zeile 96 – 99:

Je nachdem, von welcher Seite die Rakete einfliegt, soll sie deutlich außerhalb des Bildschirms starten. Hier passen wir jetzt die x-Variable an.

### Zeile 103:

Erzeugen eines Rocket-Objekts über den von Sprite geerbten Konstruktor.

### Zeile 104 – 107:

Hier setzen wir die Geschwindigkeit abhängig von der Startposition der Rakete. Es wäre ja doch unschön, wenn eine Rakete, die links außerhalb des Bildschirms startet, nach links wegfliegt, so dass wir sie nie zu sehen bekommen. ☺

### Zeile 110 - 111:

Die Rakete noch in unseren Objekt-Pool packen. Hier arbeiten wir jetzt mit einem Iterator, da wir jetzt während des Spiels den Vektor verändern.

**Wichtig:** Dies ist hier als Demonstration gedacht, wie zusätzlicher Code die Übersichtlichkeit vermindert. In diesem Fall würde es viel mehr Sinn machen, den Großteil des in der Methode implementierten Codes in den Konstruktor der Rakete zu packen!

## 2 letzte Modifikationen

In der Methode `keyReleased(KeyEvent e)`, die wir in unserem `GamePanel` implementiert haben, müssen wir noch dafür sorgen, dass der Timer beim Abbruch des Spiels gestoppt wird:

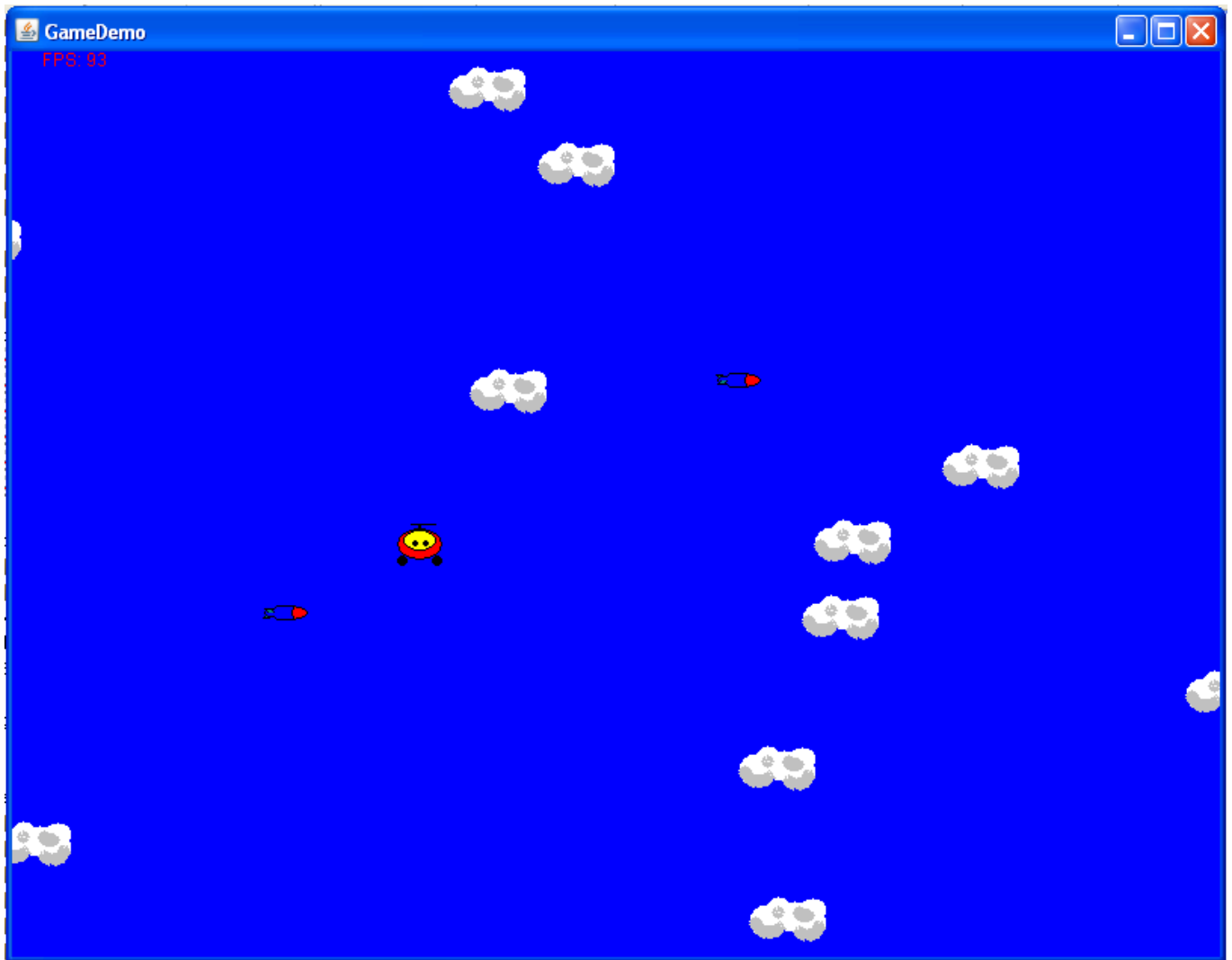
```
295     if (e.getKeyCode() == KeyEvent.VK_ESCAPE) {
296         if (isStarted()) {
297             setStarted(false);
298             timer.stop();
299         } else {
300             frame.dispose();
301         }
302     }
303 }
```

Zu guter Letzt müssen wir noch dafür sorgen, dass aus der `ActionPerformed`-Methode heraus ein Raketen-Objekt erzeugt wird:

```
310 @Override
311 public void actionPerformed(ActionEvent e) {
312     if (isStarted() && e.getSource().equals(timer)) {
313         createRocket();
314     }
315 }
```

Dabei fragen wir vorsichtshalber noch ab, ob das Spiel schon gestartet wurde. Damit wird verhindert, dass versehentlich ständig Raketen erzeugt werden.

*Und so sieht's aus*



## Feinschliff 2. Teil

Hier wollen wir jetzt noch 2 Punkte abhaken:

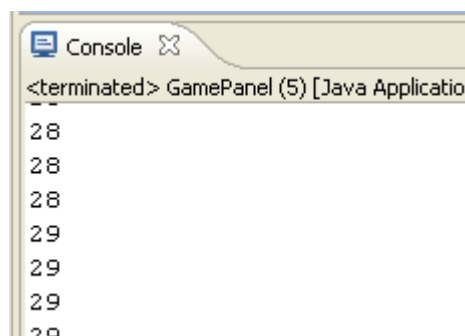
- Es gibt im Spiel noch eine potentielle Fehlerquelle, die bisher noch nicht auffällt.
- Außerdem wäre ein schöneres Hintergrundbild wünschenswert.

### Objekte wieder löschen

Die potentielle Fehlerquelle lässt sich sehr rasch verdeutlichen. Dazu fügen wir in der Methode `doLogic()` in unserem `GamePanel` vorübergehend eine Zeile Code ein:

```
154 private void doLogic() {  
155  
156     for(ListIterator<Sprite> it = actors.listIterator(); it.hasNext();){  
157         Sprite r = it.next();  
158         r.doLogic(delta);  
159     }  
160     System.out.println(actors.size());  
161  
162 }
```

In Zeile 160 geben wir die aktuelle Größe unseres Vectors in die Konsole aus. Wenn wir unser Programm dann eine Weile laufen lassen, sieht das Ergebnis ungefähr so aus:



Ursache ist, dass wir zwar fleißig Objekte in unserem Vector ablegen, diese aber niemals wieder entfernen – selbst wenn diese gar nicht mehr benötigt werden. Hiermit steigen der Speicherbedarf des Spiels und die Anzahl der abzuarbeitenden Objekte kontinuierlich an.

Um dies zu beheben wollen wir ab sofort Objekte, die nicht mehr benötigt werden, wieder entfernen. Ob ein Objekt noch benötigt wird, entscheiden wir in der jeweiligen Logik-Methode. Hierzu definieren wir uns ein entsprechendes Flag. Da wir dies für die meisten Objekte benötigen, packen wir es als Instanzvariable in unsere Klasse `Sprite`.



```

5 public abstract class Sprite extends Rectangle2D.Double implements Drawable, Movable{
6
7     private static final long serialVersionUID = 1L;
8     long delay;
9     long animation = 0;
10    GamePanel parent;
11    BufferedImage[] pics;
12    int currentpic = 0;
13
14    protected double dx;
15    protected double dy;
16
17    int loop_from;
18    int loop_to;
19
20    boolean remove;

```

Anschließend hinterlegen wir in der Logik-Methode der Klasse Rocket zusätzlichen Code, der unseren Boolean auf true setzt, wenn die Rakete nicht mehr sichtbar ist.

```

23 @Override
24 public void doLogic(long delta) {
25     super.doLogic(delta);
26
27     if(getHorizontalSpeed()>0){
28         target = new Rectangle2D.Double(getX()+getWidth(),getY(),
29             parent.getWidth()-getX(),getHeight());
30     }else{
31         target = new Rectangle2D.Double(0,getY(),getX(),getHeight());
32     }
33
34     if(!locked&&parent.copter.intersects(target)){
35         setVerticalSpeed(0);
36         locked = true;
37     }
38
39     if(locked){
40         if(getY()<parent.copter.getY()){
41             setVerticalSpeed(40);
42         }
43         if(getY()>parent.copter.getY()+parent.copter.getHeight()){
44             setVerticalSpeed(-40);
45         }
46     }
47
48     if(getHorizontalSpeed()>0 && getX()>parent.getWidth()){
49         remove = true;
50     }
51
52     if(getHorizontalSpeed()<0 && getX()+getWidth()<0){
53         remove = true;
54     }
55 }
56

```

#### Zeile 48 – 49:

Bewegt sich die Rakete nach rechts und ist der x-Wert größer als die Breite des Spielfeldes (d. h. die Rakete ist vollständig außerhalb des sichtbaren Bereichs), setzen wir unser Flag auf true

**Zeile 52 – 53:**

Andere Richtung, gleiche Bedingung. ☺

Jetzt benötigen wir in unserem GamePanel nur noch etwas Code, der uns die Objekte „aufräumt“. Den hinterlegen wir in doLogic() (wo sonst? ☺).


```
154 private void doLogic() {  
155  
156     for(ListIterator<Sprite> it = actors.listIterator(); it.hasNext();){  
157         Sprite r = it.next();  
158         r.doLogic(delta);  
159  
160         if(r.remove){  
161             it.remove();  
162         }  
163     }  
164     System.out.println(actors.size());  
165  
166 }  
167
```

**Zeile 160 - 161:**

Ist für ein Sprite (gleich welcher Art) das remove-Flag gesetzt, löschen wir es aus dem Vector. Auch diese wieder über den Iterator um ConcurrentModificationExceptions zu vermeiden.

Für die Überprüfung der Funktionsweise, lassen wir uns die Anzahl der gespeicherten Objekte in die Konsole ausgeben.

Ergebnis:

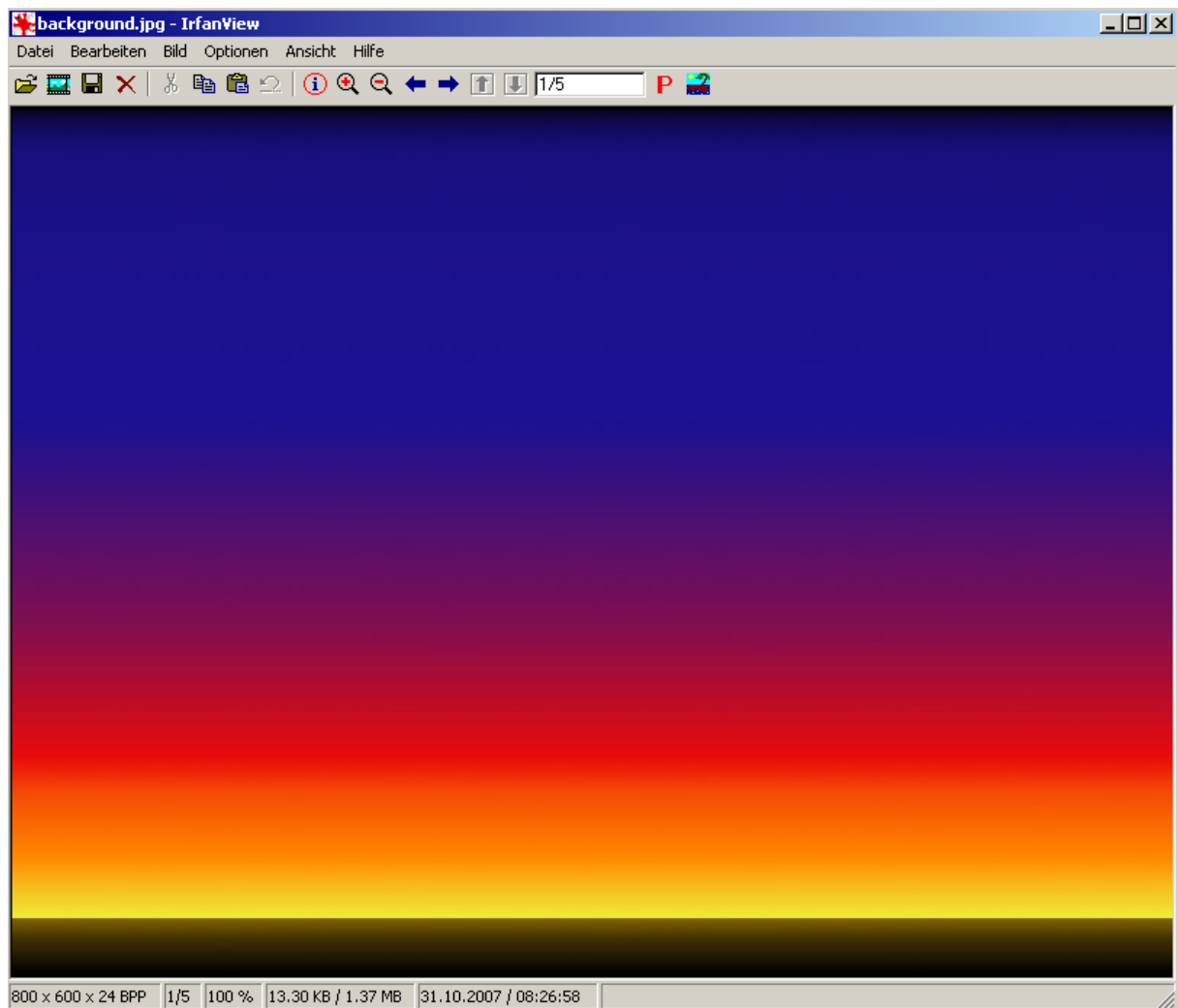


```
<terminated> GamePanel (5) [Java Application] C
15
15
15
15
15
16
16
16
```

Aktuell schwankt die Anzahl etwas, aber die Anzahl der gespeicherten Objekte übersteigt nicht die 16 (1 Hubschrauber, 12 Wolken und bis zu 3 Raketen). Der Code funktioniert also wie gewünscht. ☺

## Hintergrundbild

Nun wollen wir noch ein schöneres Hintergrundbild einbauen. Mit einem beliebigen Grafik-Programm erstellen wir ein Hintergrundbild (hier ein Farbverlauf aus GIMP) und speichern das Bild in dem Ordner in dem alle Grafik-Dateien abgelegt werden.



Da das Hintergrundbild ständig verfügbar sein muss, definieren wir uns ein Objekt als Klassenvariable und laden/füllen dieses jeweils in der Methode `doInitializations()` mit Leben.

```
15 public class GamePanel extends JPanel implements Runnable, KeyListener, ActionListener{
16
17     private static final long serialVersionUID = 1L;
18     JFrame frame;
19
20     long delta = 0;
21     long last = 0;
22     long fps = 0;
23
24     Heli copter;
25     Vector<Sprite> actors;
26     Vector<Sprite> painter;
27
28     boolean up;
29     boolean down;
30     boolean left;
31     boolean right;
32     boolean started;
33     int speed = 50;
34
35     Timer timer;
36     BufferedImage[] rocket;
37     BufferedImage background;
38 }
```

Zeile 32: `BufferedImage` zum Speichern des Hintergrundbildes

Ein Besonderheit gilt es in `doInitializations()` zu beachten:

```
59 private void doInitializations() {
60
61     last = System.nanoTime();
62
63     BufferedImage[] heli = loadPics("pics/heli.gif", 4);
64     rocket = loadPics("pics/rocket.gif", 8);
65     background = loadPics("pics/background.jpg", 1)[0];
66 }
```

Da unsere Methode `loadPics(..)` ein Array zurück liefert, wir aber nur ein Bild wollen, müssen wir aus dem zurück gegebenen Objekt das erste (und einzige) Bild abgreifen. Dies geschieht in Zeile 55 durch die Null in eckigen Klammern, die auf das erste Objekt des Arrays verweist. Jetzt müssen wir das Bild nur noch in unsere überschriebene `paintComponent(..)`-Methode packen:

```

207 @Override
208 public void paintComponent(Graphics g) {
209     super.paintComponent(g);
210
211     g.drawImage(background, 0, 0, this);
212
213     g.setColor(Color.red);
214     g.drawString("FPS: " + Long.toString(fps), 20, 10);
215
216
217     if(!started){
218         return;
219     }
220
221     for (ListIterator<Sprite> it = painter.listIterator(); it.hasNext();) {
222         Sprite r = it.next();
223         r.drawObjects(g);
224     }
225
226 }

```

Wichtig ist, dass das Bild als erstes nach dem super-Aufruf gezeichnet wird, damit es ganz hinten liegt. Alle folgenden Objekte werden dann darüber gezeichnet.

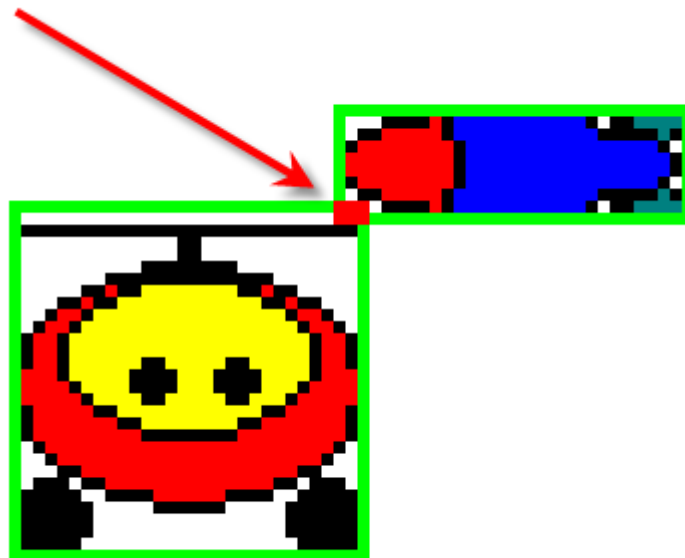
***Und so sieht's aus:***



# Kollisionen

Wenn wir das Spiel bis jetzt testen, dann bleibt das Aufeinandertreffen von Hubschrauber und Rakete folgenlos. Das wollen wir jetzt abstellen.

Wir wollen uns im ersten Schritt zunächst auf eine sehr einfache Kollision beschränken, in dem wir das Überschneiden der Rectangle-Objekte prüfen. Dies wird allerdings eine ungenaue Kollisions-Ermittlung, weil wir unsere Objekte nun mal nicht bis in die letzte Ecke gezeichnet haben. Dies wird schnell deutlich, wenn wir uns die Ränder der Bilder einmal einzeichnen:



Auch o. a. Überschneidung wird bei uns zunächst als Kollision gelten! Wir werden dies aber ändern, wenn unsere erste „Primitiv-Kollisionsprüfung“ funktioniert.

## Abstrakte Methoden

Da die Kollisionsüberprüfung für die meisten Objekte notwendig sein wird, wollen wir uns zwingen, diese in alle Objekte einzubauen, die von Sprite erben. Daher definieren wir zunächst nur einen abstrakten Methodenrumpf, den wir überall einbauen müssen:

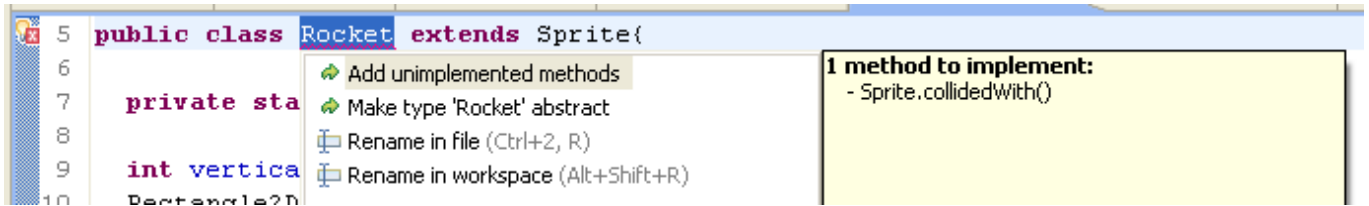
```
84+ public double getVerticalSpeed() {  
87  
88+ public void setVerticalSpeed(double dy) {  
91  
92     public abstract boolean collidedWith(Sprite s);  
93  
94 }
```

### Zeile 92:

Definition der abstrakten Methode in Klasse Sprite.



Je nach verwendetem Editor bzw. verwendeter IDE werden wir jetzt benötigt, diese Methode in allen Objekten, die von Sprite erben zu realisieren:



Entsprechend lassen wir uns in allen Objekten die benötigte Methode generieren bzw. fügen Sie von Hand ein (hier z. B. in Klasse Rocket):

```
68 @Override
69 public boolean collidedWith(Sprite s) {
70
71     return false;
72 }
```

Jetzt füllen wir diese Methode auch gleich mit Leben:

```
68 @Override
69 public boolean collidedWith(Sprite s) {
70
71     if(this.intersects(s)){
72         System.out.println("Rums!!!");
73         return true;
74     }
75
76
77     return false;
78 }
```

Der Code dazu ist denkbar kurz, da wir das Ereignis zunächst nur der Konsole bekannt geben (Zeile 72). Hier sparen wir uns erneut einige Mühe, da wir alle Objekte von Rectangle2D.Double erben lassen und uns somit schon die geeignete Methode zur Verfügung steht.

Die wiederholen wir auch für die Klasse Heli:

```
38 @Override
39 public boolean collidedWith(Sprite s) {
40
41     if(this.intersects(s)){
42         System.out.println("Aua!");
43     }
44
45     return false;
46 }
```

Nun fehlt uns nur noch der Code, um die Objekte gegenseitig zu überprüfen, dies packen wir in unsere Logik-Methode im GamePanel:

```
156 private void doLogic() {
157
158     for(ListIterator<Sprite> it = actors.listIterator(); it.hasNext();){
159         Sprite r = it.next();
160         r.doLogic(delta);
161
162         if(r.remove){
163             it.remove();
164         }
165     }
166
167     for(int i = 0; i < actors.size(); i++){
168         for(int n = i+1; n < actors.size(); n++){
169
170             Sprite s1 = actors.elementAt(i);
171             Sprite s2 = actors.elementAt(n);
172
173             s1.collidedWith(s2);
174
175         }
176     }
177
178
179
180
181 }
```

#### **Zeile 167:**

1. Schleife über alle Objekte des Vektors. Hier ohne Iterator. Hier ist die Logik etwas bequemer implementierbar, wenn man direkt auf den Vektor zugreift. Da wir nur lesen sollte diese Lösung unproblematisch sein.

#### **Zeile 168:**

2. Schleife – Ermittlung aller Objekte die auf das Objekt aus Zeile 168 noch folgen.

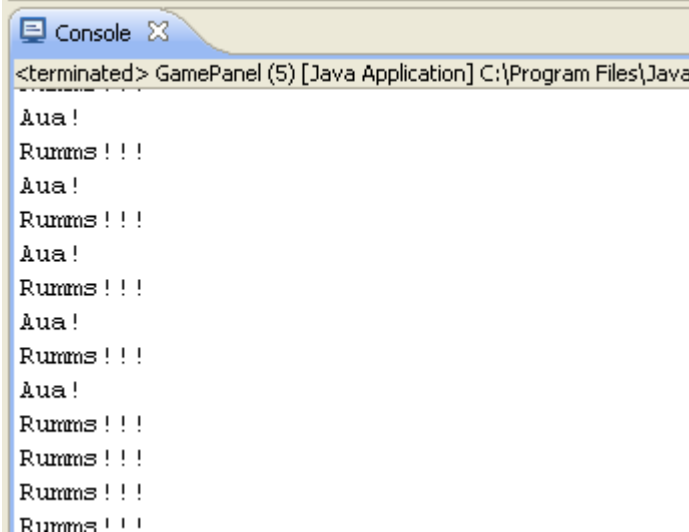
#### **Zeile 170 - 171:**

Die beiden zu prüfenden Sprites aus dem Vector holen

#### **Zeile 173:**

Kollisionsprüfung durchführen. Die einzelnen Objekte schreiben bei erfolgter Kollision in die Konsole.

Bei einem Test des Programms erhalten wir folgende Konsolenausgabe:



```
<terminated> GamePanel (5) [Java Application] C:\Program Files\Java
Aua!
Runns!!!
Aua!
Runns!!!
Aua!
Runns!!!
Aua!
Runns!!!
Aua!
Runns!!!
Runns!!!
Runns!!!
Runns!!!
```

Wem das im Moment etwas viel erscheint, der möge bedenken, dass jedes Objekt wiederholt Einträge erzeugt, solange es sich mit einem anderen überschneidet, da mit jeder Spielschleife die Prüfung wiederholt wird. Zudem werden für die Rakete und den Helikopter im Moment auch Kollisionen mit den Wolken registriert.

Daher wollen wir jetzt die Objekte nach erfolgter Kollision löschen. In die Klassen Heli und Rocket fügen wir in der Methode collidedWith(Sprite s) jeweils folgenden Code ein:

```
68 @Override
69 public boolean collidedWith(Sprite s) {
70
71     if(remove){
72         return false;
73     }
74
75     if(this.intersects(s)){
76
77         if(s instanceof Heli){
78             remove = true;
79             s.remove = true;
80             return true;
81         }
82
83         if(s instanceof Rocket){
84             remove = true;
85             s.remove = true;
86             return true;
87         }
88     }
89 }
90
```

**Zeile: 61 – 72:**

Ist das Objekt schon zum Entfernen vor gemerkt, führen wir keine weiteren Prüfungen durch.

**Zeile 77 – 80:**

Wenn das Objekt mit dem kollidiert wird eine Instanz der Klasse Heli ist, setzen wir das „Entfernen-Flag“ für beide Objekte. **Diese erste Bedingung können wir in der Klasse Heli natürlich weglassen, da wir keine 2 Heli-Objekte haben, die zusammen stoßen könnten.**

**Zeile 81 – 84:**

Kollidiert unserem aktuellen Objekt mit einem Objekt der Klasse Rocket, setzen wir bei beiden Objekten das „Entfernen-Flag“.

Das Entfernen-Flag wird hier etwas unschön mit direktem Zugriff auf die Variable gesetzt. Wer dieser schöner lösen möchte, darf hier auch gerne getter- und setter-Methoden verwenden. ☺

Wenn wir nun testen, werden alle kollidierten Objekte entfernt. Dummerweise kratzt dies unser Spiel aber momentan noch nicht, es läuft einfach ohne Hubschrauber weiter.

**So sieht's aus:**



Tja, hier sieht man eigentlich nichts. So soll es auch sein, da eine Rakete den Helikopter zerstört hat und beide entfernt wurden.

## ***Spiel-Ende signalisieren***

Damit das Spiel beendet wird, wenn unser Hubschrauber „stirbt“ müssen wir etwas Code hinterlegen.

Zunächst definieren wir uns eine Klassenvariable `gameover` vom Typ `long` in `GamePanel`:

```
15 public class GamePanel extends JPanel implements Runnable, KeyListener, ActionListener{
16
17     private static final long serialVersionUID = 1L;
18     JFrame frame;
19
20     long delta    = 0;
21     long last     = 0;
22     long fps      = 0;
23     long gameover = 0;
24
25     Heli heli;
```

Diese müssen wir auch bei jedem Aufruf von `doInitializations()` wieder auf 0 setzen:

```
60 private void doInitializations() {
61
62     last = System.nanoTime();
63     gameover = 0;
64
65     BufferedImage[] heli = loadPics("pics/heli.gif", 4);
66     rocket = loadPics("pics/rocket.gif", 8);
67     background = loadPics("pics/background.jpg", 1)[0];
68
69     actors = new Vector<Sprite>();
```

Wird unser Helikopter nun zerstört, übergeben wir dieser Variable die Systemzeit. Dies prüfen wir in unserer Logik-Methode. Anschließend hinterlegen wir noch etwas Code, um das Spiel anzuhalten.

```
158 private void doLogic() {
159
160     for(ListIterator<Sprite> it = actors.listIterator(); it.hasNext();){
161         Sprite r = it.next();
162         r.doLogic(delta);
163
164         if(r.remove){
165             it.remove();
166         }
167     }
168
169     for(int i = 0; i < actors.size(); i++){
170         for(int n = i+1; n < actors.size(); n++){
171
172             Sprite s1 = actors.elementAt(i);
173             Sprite s2 = actors.elementAt(n);
174
175             s1.collidedWith(s2);
176
177         }
178     }
179
180     if(copter.remove && gameover==0){
181         gameover = System.currentTimeMillis();
182     }
183
184     if(gameover>0){
185         if(System.currentTimeMillis()-gameover>3000){
186             stopGame();
187         }
188     }
189
190 }
191
192 private void stopGame(){
193     timer.stop();
194     setStarted(false);
195 }
```

**Zeile 180 - 181:**

Hier prüfen wir, ob unser Fluggerät zerstört wurde. Ist dies der Fall belegen wir die Variable `gameover` einmal mit der aktuellen Systemzeit in Millisekunden.

**Zeile 84-186:**

Wir prüfen, ob die Zeitdifferenz seit dem Setzen 3 Sekunden bzw. 3000 Millisekunden her ist. Dies ist hier einmal als Fixwert hinterlegt, da wir hier vermutlich weder groß herum experimentieren oder ändern werden. Ist diese Prüfung erfolgreich rufen wir die Methode `stopGame()` auf.

**Zeile 192 – 194:**

Die Methode `stopGame()` existiert noch nicht. Daher legen wir diese jetzt an: Sie enthält den gleichen Code, wie unser `KeyListener`, wenn wir während des laufenden Spiels die Escape-Taste drücken. Dementsprechend ändern wir unseren `KeyListener` hier auch noch etwas ab:

```

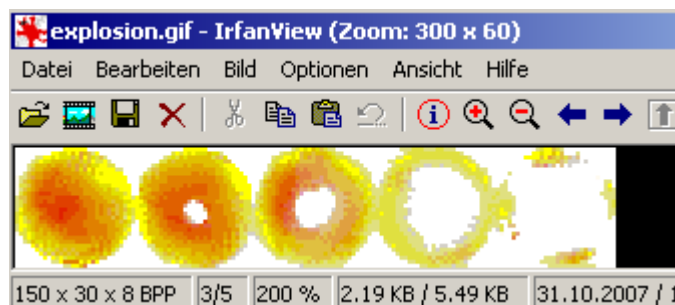
331     if (e.getKeyCode() == KeyEvent.VK_ESCAPE) {
332         if (isStarted()) {
333             stopGame();
334         } else {
335             frame.dispose();
336         }
337     }
338 }
```

Hier ersetzen wir die beiden vorhandenen Zeilen durch den Methodenaufruf.

Ab jetzt wird das aktuelle Spiel, 3 Sekunden nachdem der Hubschrauber zerstört wurde, beendet. Das Zeitintervall ist willkürlich gewählt. Es soll sicherstellen, dass die Explosionen, die wir gleich noch einbauen auch bis zum Ende animiert werden.

Explosionen

Für den Einbau der Explosionen benötigen wir nichts Neues mehr, daher werde ich dies etwas kürzer halten. Zunächst benötigen wir wieder eine Grafik-Datei mit der Animationsfolge, die wir am üblichen Ort speichern:



Außerdem definieren wir uns einen Klasse `Explosion`, die von `Sprite` erbt in der wir

- die Logik-Methode überschreiben und
- die abstrakte Methode `collidedWith(.)` implementieren.

Und so sieht sie aus:

```
4 public class Explosion extends Sprite{
5
6     private static final long serialVersionUID = 1L;
7     int old_pic = 0;
8
9     public Explosion(BufferedImage[] i, double x, double y, long delay, GamePanel p) {
10         super(i, x, y, delay, p);
11     }
12
13
14     @Override
15     public void doLogic(long delta) {
16         old_pic = currentpic;
17         super.doLogic(delta);
18         if(currentpic==0 && old_pic!=0){
19             remove = true;
20         }
21     }
22
23
24
25     @Override
26     public boolean collidedWith(Sprite s) {
27
28         return false;
29     }
30
31 }
```

Als Besonderheit ist hier anzumerken, dass in der Logik-Methode etwas mehr Code hinterlegt wird, um zu garantieren, dass nach dem ersten Durchlaufen der Animation, das remove-Flag gesetzt wird.

In diesem Fall wird geprüft, ob current\_pic das zweite Mal den Wert Null hat. Diese etwas umständliche Prüfung ist notwendig, da die Logik-Methode aus doLogic() unserer Spielschleife aufgerufen wird und dort auch direkt im Anschluss alle Objekte, die das Lösch-Flag gesetzt haben, gelöscht werden. Daher kann man nicht abfragen, ob die Animation beim letzten Bild angelangt ist, da das Objekt sonst gelöscht würde, ohne das letzte Bild je gezeichnet zu haben.

Die Methode collidedWith(Sprite s) lassen wir leer. Wollten wir die anderen Objekte beeinflussen, wenn sie durch die Explosionswolke fliegen, könnten wir hier Code hinterlegen.



In unserer Hauptklasse GamePanel definieren wir uns wieder eine Klassenvariable, um die Bilddaten immer vorrätig zu haben:

```
15 public class GamePanel extends JPanel implements Runnable, KeyListener, ActionListener{
16
17     private static final long serialVersionUID = 1L;
18     JFrame frame;
19
20     long delta = 0;
21     long last = 0;
22     long fps = 0;
23     long gameover = 0;
24
25     Heli copter;
26     Vector<Sprite> actors;
27     Vector<Sprite> painter;
28
29     boolean up;
30     boolean down;
31     boolean left;
32     boolean right;
33     boolean started;
34     int speed = 50;
35
36     Timer timer;
37     BufferedImage[] rocket;
38     BufferedImage[] explosion;
39     BufferedImage background;
40 }
```

Und füllen Sie in unserer Initialisierungs-Methode mit Leben:

```
61 private void doInitializations() {
62
63     last = System.nanoTime();
64     gameover = 0;
65
66     BufferedImage[] heli = loadPics("pics/heli.gif", 4);
67     rocket = loadPics("pics/rocket.gif", 8);
68     background = loadPics("pics/background.jpg", 1)[0];
69     explosion = loadPics("pics/explosion.gif", 5);
70
71     actors = new Vector<Sprite>();
72     painter = new Vector<Sprite>();
73     copter = new Heli(heli, 400, 300, 100, this);
74     actors.add(copter);
75
76     createClouds();
77 }
```

Zusätzlich fügen wir noch eine Methode ein, die es uns ermöglicht, eine Explosion an einer bestimmten Stelle zu erzeugen:

```
96 public void createExplosion(int x, int y){
97     ListIterator<Sprite> it = actors.listIterator();
98     it.add(new Explosion(explosion, x, y, 100, this));
99 }
```

Danach fügen wir in die Klassen Heli und Rocket noch den Code zum Erzeugen der Explosionen ein:

```
68 @Override
69 public boolean collidedWith(Sprite s) {
70
71     if(remove){
72         return false;
73     }
74
75     if(this.intersects(s)){
76
77         if(s instanceof Heli){
78             parent.createExplosion((int)getX(), (int)getY());
79             parent.createExplosion((int)s.getX(), (int)s.getY());
80             remove = true;
81             s.remove = true;
82             return true;
83         }
84
85         if(s instanceof Rocket){
86             parent.createExplosion((int)getX(), (int)getY());
87             parent.createExplosion((int)s.getX(), (int)s.getY());
88             remove = true;
89             s.remove = true;
90             return true;
91         }
92     }
93 }
```

Dies wird für jeweils beide Objekte durchgeführt, da bei diesen im Anschluss das Entfernen-Flag auf true gesetzt wird und das „Gegner-Objekt“ durch die Bedingung in Zeile 69 dann nicht mehr abgearbeitet würde, wenn es im Rahmen der Kollisionsprüfung an die Reihe käme (bedingt durch den Code in Zeile 69).

*Und so sieht's aus:*



# Pixelgenaue Kollisionsermittlung

Wie zu Beginn der Kollisionsermittlung beschrieben, ist es sehr ungenau, wenn die Kollision nur über die Methoden der Klasse Rectangle2D durchgeführt wird. Gerade wenn man Grafiken verwendet, die relativ viel transparenten Hintergrund enthalten kann so ein Spiel schnell frustrierend werden.

Um dies zu vermeiden, wollen wir den bisherigen Code so abändern, dass eine genauere Prüfung durchgeführt wird, sobald eine potentielle Kollision erkannt wurde. Hierfür ist jedoch einiges an zusätzlichem Code notwendig.

Da wir den Code für alle Objekte zur Verfügung haben wollen, fügen wir diesen in die Klasse Sprite ein:

## Die Methode checkOpaqueColorCollisions(Sprite s):

```
95 public boolean checkOpaqueColorCollisions(Sprite s){
96
97     Rectangle2D.Double cut = (Double) this.createIntersection(s);
98
99     if((cut.width<1)|| (cut.height<1)){
100         return false;
101     }
102
103     // Rechtecke in Bezug auf die jeweiligen Images
104     Rectangle2D.Double sub_me = getSubRec(this,cut);
105     Rectangle2D.Double sub_him = getSubRec(s,cut);
106
107     BufferedImage img_me = pics[currentpic].getSubimage((int)sub_me.x, (int)sub_me.y,
108                                                         (int)sub_me.width, (int)sub_me.height);
109     BufferedImage img_him = s.pics[s.currentpic].getSubimage((int)sub_him.x, (int)sub_him.y,
110                                                             (int)sub_him.width, (int)sub_him.height);
111
112     for(int i=0;i<img_me.getWidth();i++){
113         for(int n=0;n<img_him.getHeight();n++){
114
115             int rgb1 = img_me.getRGB(i,n);
116             int rgb2 = img_him.getRGB(i,n);
117
118
119             if(isOpaque(rgb1)&&isOpaque(rgb2)){
120                 return true;
121             }
122
123         }
124     }
125 }
```

Zur Ermittlung der pixelgenauen Kollision erzeugen wir die Methode oben, die das gegnerische Objekt übergeben bekommt.

### Zeile 97:

Hier ermitteln wir zunächst das Rechteck, welches die Schnittmenge der beiden kollidierenden Rechtecke darstellt

### Zeile 99:

Prüfung, dass in Zeile 103 auch wirklich eine verwertbare Schnittmenge erzeugt wurde.

**Zeile 104-105:**

Hier errechnen wir mit einer selbsterstellten Methode 2 Rechtecke in Bezug auf die aktuellen Sprites/Images bzw. deren aktuellen Bildschirmposition. Der Code dieser Methode wird weiter unten dargestellt

**Zeile 107-110:**

Aus dem aktuell angezeigten Bild der beiden Sprite wird die oben ermittelte Schnittmenge mit `getSubImage(..)` ausgeschnitten. Diese Methode ist Teil von `BufferedImage` und kann in der API nachgelesen werden.

**Zeile 112 – 123:**

Danach wird für jedes einzelne Pixel auf den beiden erzeugten Bildern geprüft, ob es eines gibt, dass auf beiden Bildern nicht transparent ist. Diese Prüfung wird mit der selbst erstellten Methode `isOpaque(..)` durchgeführt (Ausprägung weiter unten). Wurde ein Pixel gefunden, dass auf beiden Bildern nicht transparent ist, wird eine Kollision zurück gemeldet.

**Die Methode `getSubRectangle(..)`:**

```
129 protected Rectangle2D.Double getSubRec(Rectangle2D.Double source, Rectangle2D.Double part) {  
130  
131     //Rechtecke erzeugen  
132     Rectangle2D.Double sub = new Rectangle2D.Double();  
133  
134     //get X - compared to the Rectangle  
135     if(source.x>part.x){  
136         sub.x = 0;  
137     }else{  
138         sub.x = part.x - source.x;  
139     }  
140  
141     if(source.y>part.y){  
142         sub.y = 0;  
143     }else{  
144         sub.y = part.y - source.y;  
145     }  
146  
147     sub.width = part.width;  
148     sub.height = part.height;  
149  
150     return sub;  
151 }
```

Die Methode `getSubRectangle(..)` bekommt 2 `Rectangle2D.Double` übergeben: als erstes das Rechteck des Sprites aus dem die Bilddaten der Schnittmenge ermittelt werden sollen und als 2. eben diese Schnittmenge der beiden Rechtecke.

Die Methode errechnet dann die Koordinaten der Schnittmenge bezogen auf die Position des ersten Rechtecks bzw. des ersten Sprites. Damit ist es später dann möglich, die Schnittmenge der beiden Sprites in grafischer Form zu ermitteln.

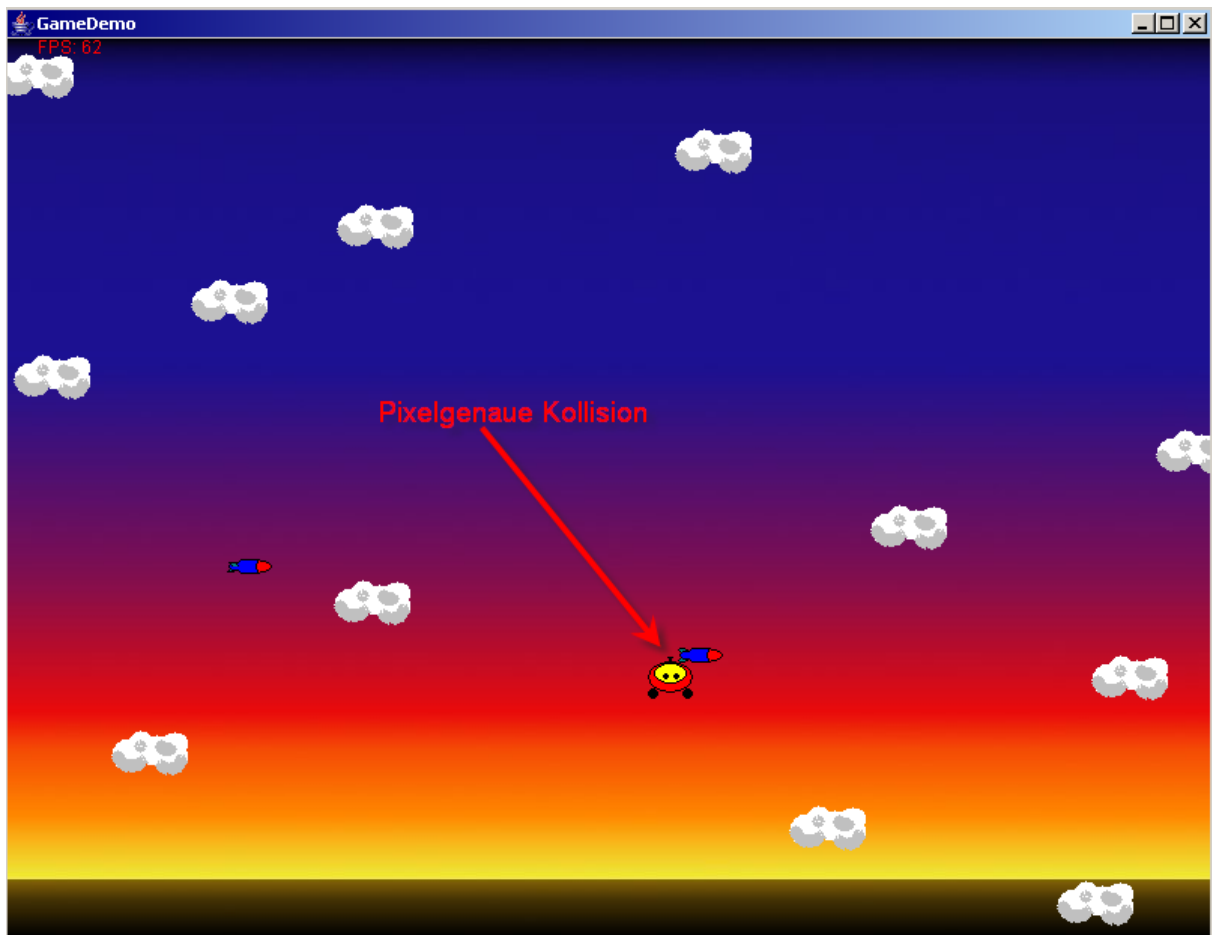
## Die Methode isOpaque(int rgb):

```
159 protected boolean isOpaque(int rgb) {  
160  
161     int alpha = (rgb >> 24) & 0xff;  
162     //red    = (rgb >> 16) & 0xff;  
163     //green  = (rgb >>  8) & 0xff;  
164     //blue   = (rgb ) & 0xff;  
165  
166     if(alpha==0){  
167         return false;  
168     }  
169  
170     return true;  
171  
172 }
```

Die Methode bekommt einen RGB-Wert übergeben und ermittelt den Alpha-Wert durch Bit-Verschiebung.

Farb-Informationen innerhalb des rgb-Wertes lt. API:  
(Bits 24-31 = alpha, 16-23 = rot 8-15 = grün, 0-7 = blau).

## Und so sieht's aus:



Bzw. auf diesem Bild eben keine Kollision wg. Pixelgenauer Prüfung ☺

# Sound

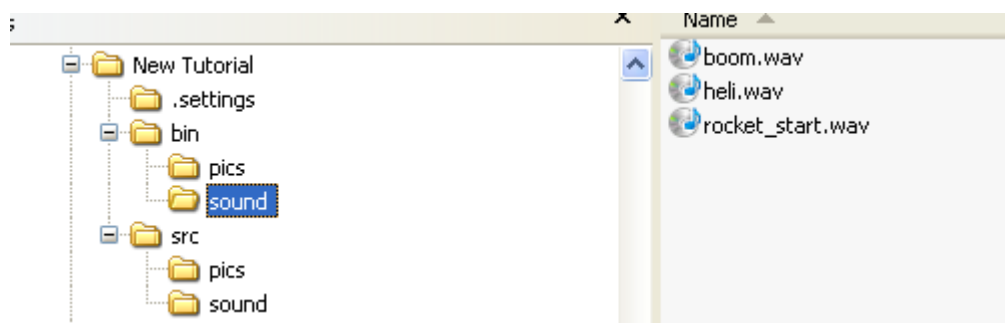
Zu guter Letzt fehlt uns noch der Sound. Ohne diesen ist so ein Spiel nur halb programmiert. In diesem Anfängertutorial wollen wir uns auf eine relative einfache Lösung beschränken. Gerade zum Thema Sound gibt es sehr viele, auch komplexe, Lösungsmöglichkeiten und entsprechende externe Bibliotheken.

Es gibt jedoch auch eine einfache Standard-Lösung die für kleinere Spiele durchaus ausreichend ist.

Diesmal wollen wir eine eigene Klasse zum Verwalten der Sounds verwenden, was letzten Endes viel komfortabler ist – auch unter dem Gesichtspunkt der Wiederverwendung des Codes in späteren Projekten. Zudem wird dann auch der Unterschied z. B. zu den Bilddaten deutlicher, wo wir auf eine eigene Klasse zur Verwaltung der Daten verzichtet haben.

Basis für unsere Sound-Ausgabe ist das Interface AudioClip. Instanzen dieses Interfaces können über die statische Methode `newAudioClip(URL u)` der Klasse `Applet` geladen werden (vgl. API). Es ist möglich, mehrere AudioClips gleichzeitig abzuspielen, sowie diese wiederholt ablaufen zu lassen (loop), was z. B. bei Fahrzeuggeräuschen sehr nützlich ist. (wiederum siehe API)

Für dieses Anfänger-Tutorial wollen wir uns auf 3 Sounds beschränken, die wir, analog zu den Bilddateien, in einem eigenen Verzeichnis ablegen.



## Die Klasse SoundLib

```
1 import java.applet.*;
2 import java.net.*;
3 import java.util.*;
4
5 public class SoundLib {
6
7     Hashtable<String, AudioClip> sounds;
8     Vector<AudioClip> loopingClips;
9
10    public SoundLib(){
11        sounds = new Hashtable<String, AudioClip>();
12        loopingClips = new Vector<AudioClip>();
13    }
14
15    public void loadSound(String name, String path){
16
17        if(sounds.containsKey(name)){
18            return;
19        }
20
21        URL sound_url = getClass().getClassLoader().getResource(path);
22        sounds.put(name, (AudioClip)Applet.newAudioClip(sound_url));
23    }
24
25    public void playSound(String name){
26        AudioClip audio = sounds.get(name);
27        audio.play();
28    }
29
30    public void loopSound(String name){
31        AudioClip audio = sounds.get(name);
32        loopingClips.add(audio);
33        audio.loop();
34    }
35
36    public void stopLoopingSound(){
37        for(AudioClip c:loopingClips){
38            c.stop();
39        }
40    }
41
42 }
```

### Zeile 7:

HashTable um die AudioClips abzuspeichern

### Zeile 8:

Vector, in dem die AudioClips gespeichert werden, die sich permanent wiederholen. Damit diese später beendet werden können.

### Zeile 10 – 13:

Konstruktor. Die Collections werden hier instanziiert.



**Zeile 15 – 23:**

Methode zum Laden der AudioClips. Die Methode bekommt einen Namen zum Speichern im HashTable und die Pfadangabe übergeben. Wenn der AudioClip schon im HashTable existiert wird abgebrochen, andernfalls wird der AudioClip geladen und im HashTable abgelegt.

**Zeile 25 – 28:**

Methode zum einmaligen Abspielen eines AudioClips. Der AudioClip wird anhand des übergebenen Namens im HashTable gesucht und mit play() gestartet.

**Zeile 30 – 34:**

Methode zum permanenten Abspielen eines AudioClips. Die entsprechenden AudioClips werden in einem Vector gespeichert.

**Zeile 36 – 40:**

Über diese Methode können alle AudioClips, die permanent abgespielt werden, beendet werden.

**Verwenden der Klasse SoundLib**

Zunächst definieren wir uns eine Instanzvariable:

```
15 public class GamePanel extends JPanel implements Runna
16
17     private static final long serialVersionUID = 1L;
18     JFrame frame;
19
20     long delta    = 0;
21     long last     = 0;
22     long fps      = 0;
23     long gameover = 0;
24
25     Heli copter;
26     Vector<Sprite> actors;
27     Vector<Sprite> painter;
28
29     boolean up;
30     boolean down;
31     boolean left;
32     boolean right;
33     boolean started;
34     int speed = 50;
35
36     Timer timer;
37     BufferedImage[] rocket;
38     BufferedImage[] explosion;
39     BufferedImage background;
40
41     SoundLib soundlib;
42
```

In unserer Initialisierungs-Methode instanziiieren wir die Klasse und versorgen Sie auch gleich mit den wav-Dateien:

```
63 private void doInitializations() {
64
65     last      = System.nanoTime();
66     gameover  = 0;
67
68     BufferedImage[] heli = loadPics("pics/heli.gif", 4);
69     rocket     = loadPics("pics/rocket.gif", 8);
70     background = loadPics("pics/background.jpg", 1)[0];
71     explosion  = loadPics("pics/explosion.gif", 5);
72
73     actors = new Vector<Sprite>();
74     painter = new Vector<Sprite>();
75     copter  = new Heli(heli, 400, 300, 100, this);
76     actors.add(copter);
77
78     soundlib = new SoundLib();
79     soundlib.loadSound("boomm", "sound/boom.wav");
80     soundlib.loadSound("rocket", "sound/rocket_start.wav");
81     soundlib.loadSound("heli", "sound/heli.wav");
82
83     createClouds();
84
85     timer = new Timer(3000, this);
86     timer.start();
87
88     started = false;
89 }
```

#### Zeile 78 – 81:

Instanziiieren des Objektes und laden der 3 Sounddateien

Da der Helikopter-Sound während des ganzen Spiels laufen soll und beim Ende des Spiels anhalten soll, erweitern wir um einen die Methode stopGame() und erzeugen uns analog eine Methode zum Spielstart:

```
206 private void startGame() {
207     doInitializations();
208     setStarted(true);
209     soundlib.loopSound("heli");
210 }
211
212 private void stopGame() {
213     timer.stop();
214     setStarted(false);
215     soundlib.stopLoopingSound();
216 }
```

#### Zeile: 206 – 209:

Das ist unsere neue Methode startGame(). Sie initialisiert alle notwendigen Variablen, setzt den Status auf gestartet und stößt die Sound-Schleife an.

### Zeile 215:

Hier fügen wir in `stopGame()` noch das Ende der Schleife ein, damit der Sound nicht weiter läuft, wenn wir das Spiel beenden.

Jetzt müssen wir noch kurz den `KeyListener` anpassen und die beiden vorhandenen Einzelbefehle durch den Methodenaufruf ersetzen.

```
327 @Override
328 public void keyReleased(KeyEvent e) {
329
330     if(e.getKeyCode() == KeyEvent.VK_UP) {
331         up = false;
332     }
333
334     if(e.getKeyCode() == KeyEvent.VK_DOWN) {
335         down = false;
336     }
337
338     if(e.getKeyCode() == KeyEvent.VK_LEFT) {
339         left = false;
340     }
341
342     if(e.getKeyCode() == KeyEvent.VK_RIGHT) {
343         right = false;
344     }
345
346     if(e.getKeyCode() == KeyEvent.VK_ENTER) {
347         if(!isStarted()) {
348             startGame();
349         }
350     }
351
352     if(e.getKeyCode() == KeyEvent.VK_ESCAPE) {
353         if(isStarted()) {
354             stopGame();
355         } else {
356             frame.dispose();
357         }
358     }
359 }
```

Nun müssen die Sounds noch den Ereignissen zugeordnet werden:

```
103 public void createExplosion(int x, int y) {
104     ListIterator<Sprite> it = actors.listIterator();
105     it.add(new Explosion(explosion, x, y, 100, this));
106     soundlib.playSound("bumm");
107 }
```

### Zeile 106:

Wenn eine Explosion erzeugt wird, wird der entsprechende Sound ausgelöst

Beim Erzeugen eines `Rocket`-Objekts lassen wir einen Warnton erzeugen (der im Beispiel zugegebenermaßen sehr nervig ist☹):

```

109 private void createRocket() {
110
111     int x = 0;
112     int y = (int) (Math.random() * getHeight());
113     int hori = (int) (Math.random() * 2);
114
115     if (hori == 0) {
116         x = -30;
117     } else {
118         x = getWidth() + 30;
119     }
120
121
122     Rocket rock = new Rocket(rocket, x, y, 100, this);
123     if (x < 0) {
124         rock.setHorizontalSpeed(100);
125     } else {
126         rock.setHorizontalSpeed(-100);
127     }
128
129     ListIterator<Sprite> it = actors.listIterator();
130     it.add(rock);
131     soundlib.playSound("rocket");
132
133 }

```

Und schließlich müssen wir noch beim Beenden des Spiels, die Soundschleife für den Helikopter beenden.

Das war's ab sofort haben wir ein „richtiges“ Spiel.

Hier gibt's jetzt kein "So sieht's aus"-Kapitel. Sound lässt sich halt leider nicht grafisch abbilden 😊. Aber das Ergebnis kann durch einen Programm-Test selbst begutachtet werden.

## Fertig – vorerst

So, das war's. Das erste Spiel ist fertig und läuft. Damit ist jetzt eine erste Grundlage in Richtung Spiele-Programmierung gelegt. Natürlich ist das ein komplexes und durchaus anspruchsvolles Thema und dieses Tutorial kratzt sozusagen nur an der Oberfläche. Ich würde mich aber freuen, wenn Ihr Lust auf mehr bekommen habt. 😊