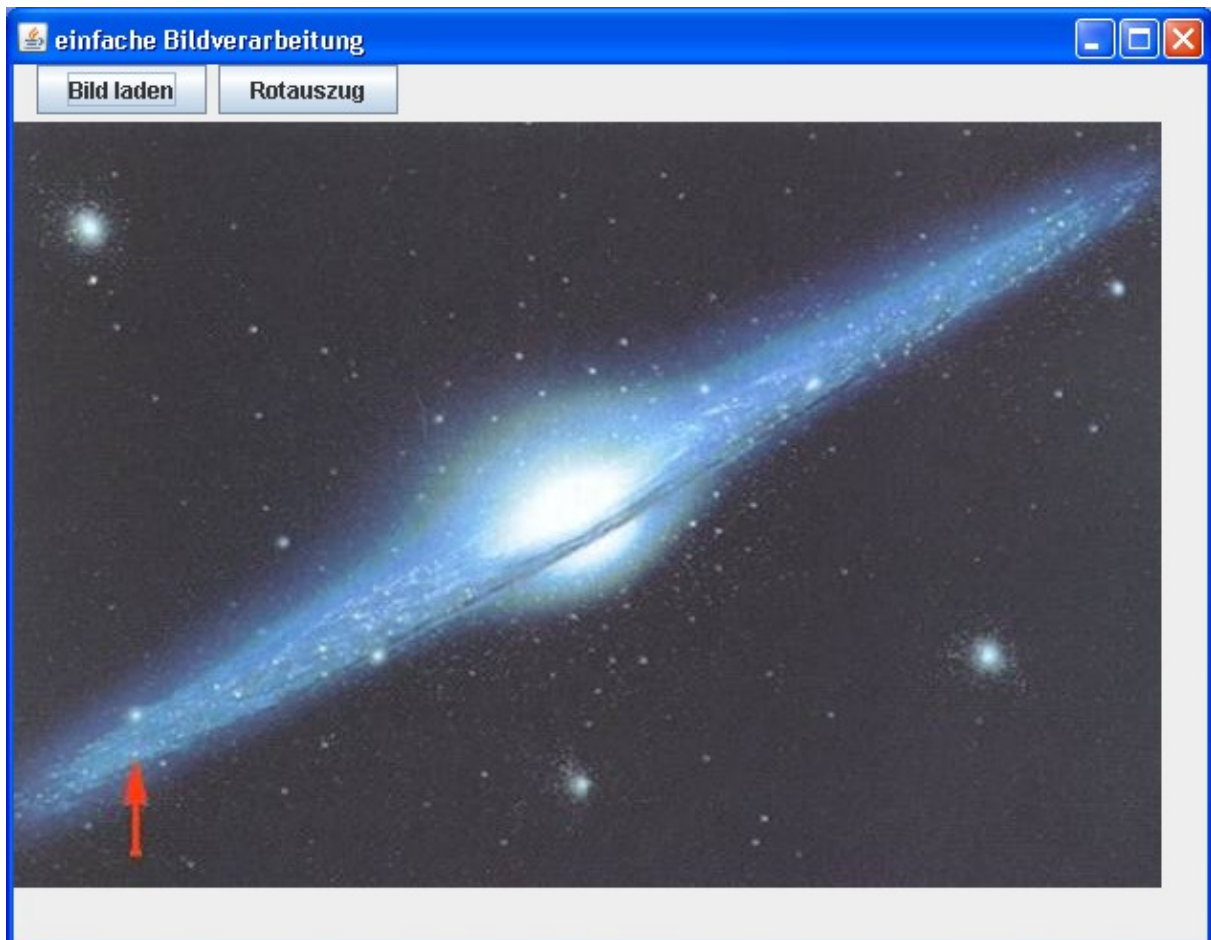


Einführung in die Informatik - Teil 5 -

Reihungen



Inhalt:

1. Bezug zum Unterricht: Anwendungen
2. Das Standardbeispiel: Sortieren
3. Aufgaben
4. Bildverarbeitung
5. Aufgaben

Literaturhinweise:

- Küchlin/Weber: Einführung in die Informatik, Objektorientiert mit Java, Springer 1998
- Krüger, Guido: Handbuch der Java-Programmierung, <http://www.javabuch.de> oder Addison Wesley 2002

1. Bezug zum Unterricht: Anwendungen

Die im Unterricht benutzten Elemente einer Programmiersprache sollten sich nicht aus Vollständigkeits- oder anderen fachimmanenten Überlegungen ergeben, sondern möglichst aus einer Problemstellung abgeleitet werden, die Bezug zu den Möglichkeiten und Folgen der *Anwendung von Informatiksystemen* hat. Für Reihungen (*Felder, Arrays*) ergeben sich viele Anwendungsmöglichkeiten, da schon im Anfangsunterricht größere Datenmengen verarbeitet werden können, ohne eine echte Datei- oder Datenbankorganisation zu benötigen:

- Reine Datenverarbeitung (*Daten verwalten, sortieren, in Daten suchen, ...*)
- Bildbearbeitung (*Bilder erzeugen, verändern, konvertieren, darstellen, ...*)
- mit Zufallszahlen arbeiten (*Bereiche festlegen, ordnen, Verteilung, doppelte Z., ...*)
- Spiele (, ...)
- in Koordinatensystemen arbeiten (*zelluläre Automaten, Simulationen, ...*)

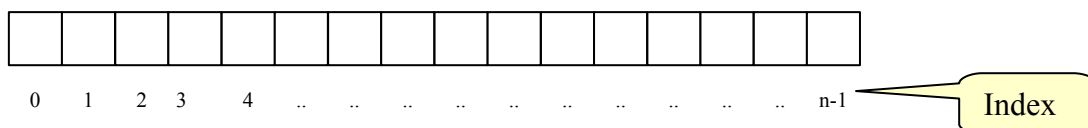
Da die Daten oft mühsam eingegeben werden müssen, sollte eine Hilfsklasse bereitgestellt werden, mit deren Hilfe Felddaten gespeichert und wieder geladen werden können.

2. Das Standardbeispiel: Sortieren

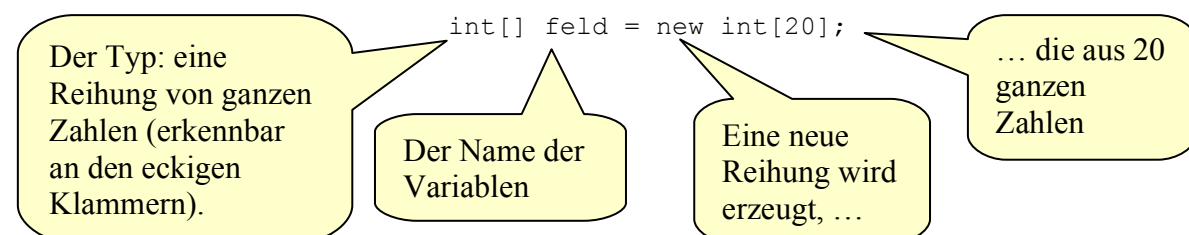
Eigentlich kennen wir Reihungen schon: Zeichenketten sind eine besondere Art von Reihungen – weil sie so oft vorkommen, haben sie zusätzliche Eigenschaften. Wir wissen aber schon, dass Java die Zählung bei Null beginnt und dass man z. B. die Länge einer Reihung (eines Strings) feststellen kann.

Um den technischen Umgang mit Reihungen möglichst knapp darzustellen, soll hier das übliche Verfahren angewandt werden: wir sortieren Zahlen¹.

Bei Reihungen handelt es sich um lineare Anordnungen gleicher Elemente. Dabei ist es erstmal gleichgültig, um welche Art von Elementen es sich handelt. Wir benutzen hier nur deshalb Zahlen, weil sie sich einfach (durch Zufallszahlen) erzeugen und dann darstellen lassen. Enthält eine Reihung n Elemente, dann können wir uns das Gebilde etwa so vorstellen:



Wichtig ist, dass wir mit dem Namen der Reihung die Gesamtheit der Elemente bezeichnen. Heißt eine Reihung `feld` und enthält 20 ganze Zahlen, dann vereinbaren diese Reihungen als:



¹ Im Unterricht halte ich von diesem Vorgehen überhaupt nichts. Man sollte lieber ein Problem wählen, aus dem sich der Bedarf nach Sortiermöglichkeiten ergibt.

Weil man im Umgang mit Reihungen natürlich logische Fehler macht (z. B. falsch sortiert), test man seine Programme tunlichst mit wenigen Elementen. Erst danach vergrößert man die Zahl der Elemente auf den wirklich benötigten Wert – und macht damit neue Fehler, weil man an einigen Stellen vergisst, die Größe zu ändern. Dieses Problem umgeht man, indem man eine *Konstante* benutzt, die die aktuelle Größe der Reihung enthält. Wenn man dann alle Anweisungen auf diese Konstante bezieht, braucht man zuletzt nur noch deren Wert an einer Stelle zu verändern – ohne neue Fehler zu produzieren.

```
final int n = 20;           // die Konstante
int[] feld = new int[n];   // hier wird die Konstante benutzt
```

Auf die Einzelelemente einer Reihung greifen wir mithilfe des *Index* zu. Enthält die Reihung *n* Elemente, dann kann der Index Werte zwischen 0 und *n*-1 annehmen (s.o.). Das dritte Element (mit dem Index 2!) sprechen wir durch das folgende Konstrukt an:

```
feld[2]
```

Statt also explizit das erste Element mit dem zweiten und dann mit dem dritten usw. zu vergleichen, kann man formulieren, dass ein Element z. B. mit allen folgenden verglichen werden soll. Der Index ist über eine Variable angebar!

```
feld[i]
```

Die typische Kontrollstruktur für Reihungen ist die Zählschleife. Da man oft (fast) alle Elemente einer Reihung gleichartig bearbeiten will und man die Anzahl der Elemente mithilfe von `length` (ohne Klammern!²) ermitteln kann, sprechen wir über die Zählvariable alle Elemente der Reihung an:

```
for(int i=0;i<feld.length;i++)
{
    //tu was mit den Feldelementen
}
```

Jetzt geht es los! Wir ermitteln Lottozahlen.

Lottozahlen erhalten wir mithilfe des Zufallsgenerators:

```
(int)Math.floor(Math.random()*48+1)
```

Eine Reihung mit Platz für sechs Lottozahlen füllen wir mit der Anweisungsfolge

```
for(int i=0;i<feld.length;i++)
{
    feld[i] = (int)Math.floor(Math.random()*48+1);
}
```

Das verpacken wir eine Appletmethode `zahlenZiehen`. Das gesamte Applet sieht dann so aus:

```
public class AppletLotto extends java.applet.Applet
{
    final int n = 6;
    int[] feld = new int[n];

    public void zahlenZiehen()
    {
        for(int i=0;i<feld.length;i++)
        {
            feld[i] = (int)Math.floor(Math.random()*48+1);
        }
    }
}
```

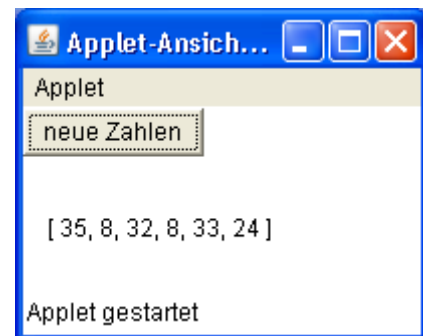
² Im Gegensatz zu den Zeichenketten. Wer darauf wohl gekommen ist?

Natürlich sollen die Zahlen auch angezeigt werden. Wir ergänzen also unser Applet um eine Methode `toString`, der wir die Reihung übergeben. Die Methode bastelt daraus eine Zeichenkette zusammen, in der die Zahlen schön mit Kommas getrennt sind und mit Klammern drum herum. Die Reihung benutzen wir hier noch als *globale Variable*, damit sie in der Methode bekannt ist.

```
public String toString()
{
    String h="[ ";
    for(int i=0;i<feld.length-1;i++)
        h = h + feld[i]+", ";
    h = h+feld[feld.length-1]+" ]";
    return h;
}
```

Ins Applet fügen wir mithilfe des Designers einen Button `bNeueZahlen` und eine Label-Komponente `lAnzeige` ein, um Lottozahlen zu erzeugen und anzuzeigen.

```
private void bNeueZahlenActionPerformed
    (java.awt.event.ActionEvent evt)
{
    zahlenZiehen();
    lAnzeige.setText(toString());
}
```

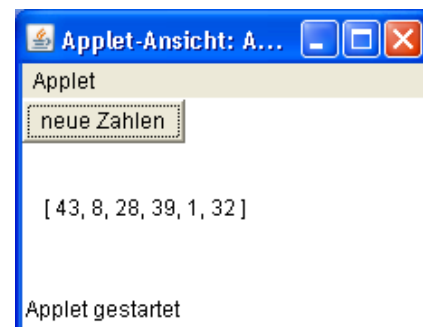


Und schon geht es schief! Die 8 ist doppelt.

Wie lösen wir das Problem?

Wie müssen feststellen können, ob eine Zahl schon gezogen worden ist. Dazu vergleichen wir jede neue Zahl mit den schon gezogenen.

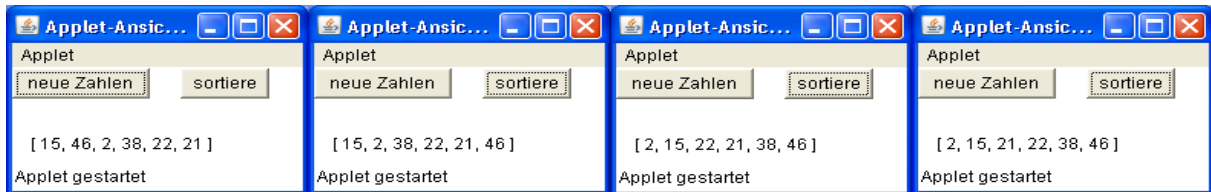
```
public void zahlenZiehen()
{
    for(int i=0;i<feld.length;i++)
    {
        boolean vorhanden; //gibt an, ob die Zahl schon vorhanden ist
        do
            //beliebig oft wiederholen
        {
            vorhanden = false; // das nehmen wir erstmal an
            feld[i] = (int)Math.floor(Math.random()*48+1);
            for(int j=0;j<i;j++) // alle vorhergehenden Zahlen vergleichen
                if(feld[j] == feld[i]) vorhanden = true; // Doppelte gefunden
        }
        while(vorhanden);
    }
}
```



Schon besser. Aber für die Ausgabe sollten Zahlen jetzt noch sortiert werden (na also!). Dafür gibt es viele Verfahren. Ein einfaches ist, jeweils benachbarte Zahlen zu vergleichen und zu vertauschen, falls sie in falscher Reihenfolge stehen. Bei jedem Durchlauf rutscht dann die größte Zahl nach ganz rechts. Bei den Vertauschungen benötigen wir eine Hilfsgröße (hier: `h`), die Zwischenwerte aufnehmen kann. Wir tauschen die Zahlen also in einer Art Ringtausch. Der Anschaulichkeit halber wollen wir diesen Vorgang auf Mausklick nur jeweils einmal ablaufen lassen. Das Feld wird dann durch sechs solcher Durchläufe sortiert.

```
private void bSortiereActionPerformed(java.awt.event.ActionEvent evt)
{
    for(int i=0;i<feld.length-1;i++)
        if(feld[i]>feld[i+1]) //Da mit dem Nachfolger verglichen wird,
        { //läuft der Index nur bis zur vorletzten Zahl
            int h = feld[i]; //Hilfsgröße für Ringtausch
            feld[i] = feld[i+1];
            feld[i+1] = h;
        }
    lAnzeige.setText(toString());
}
```

Das probieren wir jetzt mal aus!



Natürlich können wir mithilfe einer zweiten Schleife den Sortiervorgang auch vollständig ausführen. Wir vergleichen diesmal, von links beginnend, jede Zahl mit allen nachfolgenden und vertausche bei Bedarf. Damit sorgen wir dafür, dass die kleinste Zahl nach links wandert. Das Verfahren ist eine Variante von *Bubblesort*.

```
private void bSortiereActionPerformed(java.awt.event.ActionEvent evt)
{
    for(int i=0;i<feld.length-1;i++)
        for(int j=i+1;j<feld.length;j++)
            if(feld[i]>feld[j])
            {
                int h = feld[i];
                feld[i] = feld[j];
                feld[j] = h;
            }
    lAnzeige.setText(toString());
}
```

Weitere Details zu Reihungen finden Sie im Abschnitt „Referenztypen“.

3. Aufgaben

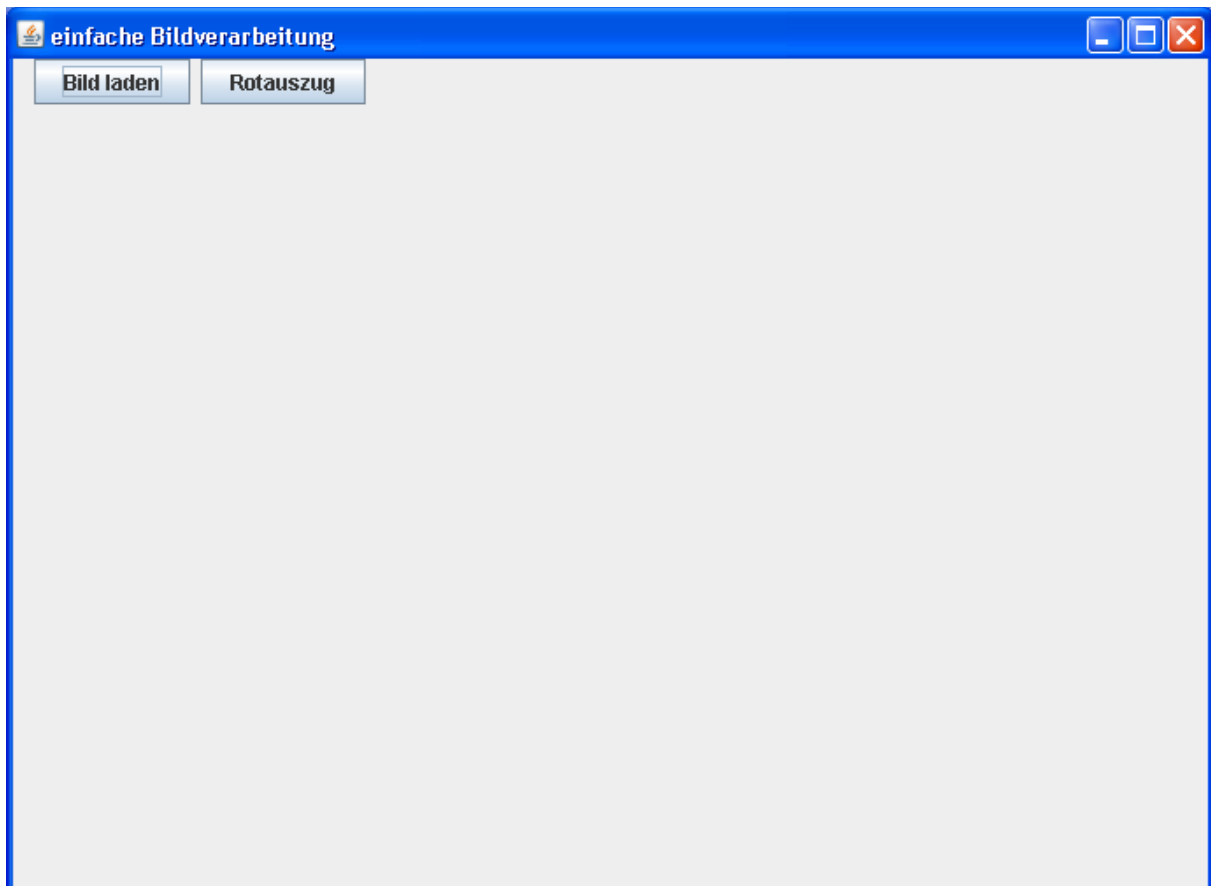
1. a: Schreiben Sie eine Methode *kleinste()*, die die kleinste Zahl in der Reihung findet.
- b: Ermöglichen Sie es, die Zahlen absteigend zu sortieren.
- c: Implementieren Sie eine bessere Sortiermethode.
- d: Implementieren Sie eine Methode *richtige()*, die den Inhalt der Reihung mit einer zweiten Reihung vergleicht, in der sich die gezogenen Zahlen befinden..
2. a: Informieren Sie sich im Netz über verschiedene Sortierverfahren.
- b: Schreiben Sie Applets, die die Arbeitsweise jeweils eines Verfahrens grafisch ansprechend darstellen.

3. a: Gegeben seien zwei Felder mit Zufallszahlen. Diese sollen so aneinander gehängt werden, dass ein einziges Feld entsteht.
- b: Sortieren Sie das ganze Feld unter der Annahme, dass die beiden Felder schon sortiert waren.
- c: Sortieren Sie in das Feld einzelne weitere Zahlen ein.

4. Bildverarbeitung

Das Standardbeispiel für zweidimensionale Reihungen sind Bilder. Dabei verfügt Java einerseits über sehr schlagkräftige Methoden zur Bildverarbeitung, andererseits verlangt es, dass man sich ziemlich in die Java-Bibliotheken hineinwühlt.

Als Beispiel wollen wir ein Bild mit vorgegebenem Namen, das sich in unserem Projektverzeichnis befindet, anzeigen. Anschließend erzeugen wir einen Rotauszug des Bildes.



Wir gehen das Problem schrittweise an.

Zuerst vereinbaren wir die erforderlichen Variablen. Das ist nur ein `Image`, also ein Objekt, das ein Bild enthalten kann. Weiterhin können sich `Image`-Objekte ihr Bild aus einer Datei laden – das brauchen wir ja gerade. Damit das Programm übersetzt werden kann, geben wir ganz oben die benutzten Bibliotheken an (als `import`-Anweisungen):

```
import java.awt.*;  
import java.awt.image.*;  
  
public class FrameBildverarbeitung extends javax.swing.JFrame  
{  
    Image bild = null;  
    ...  
}
```

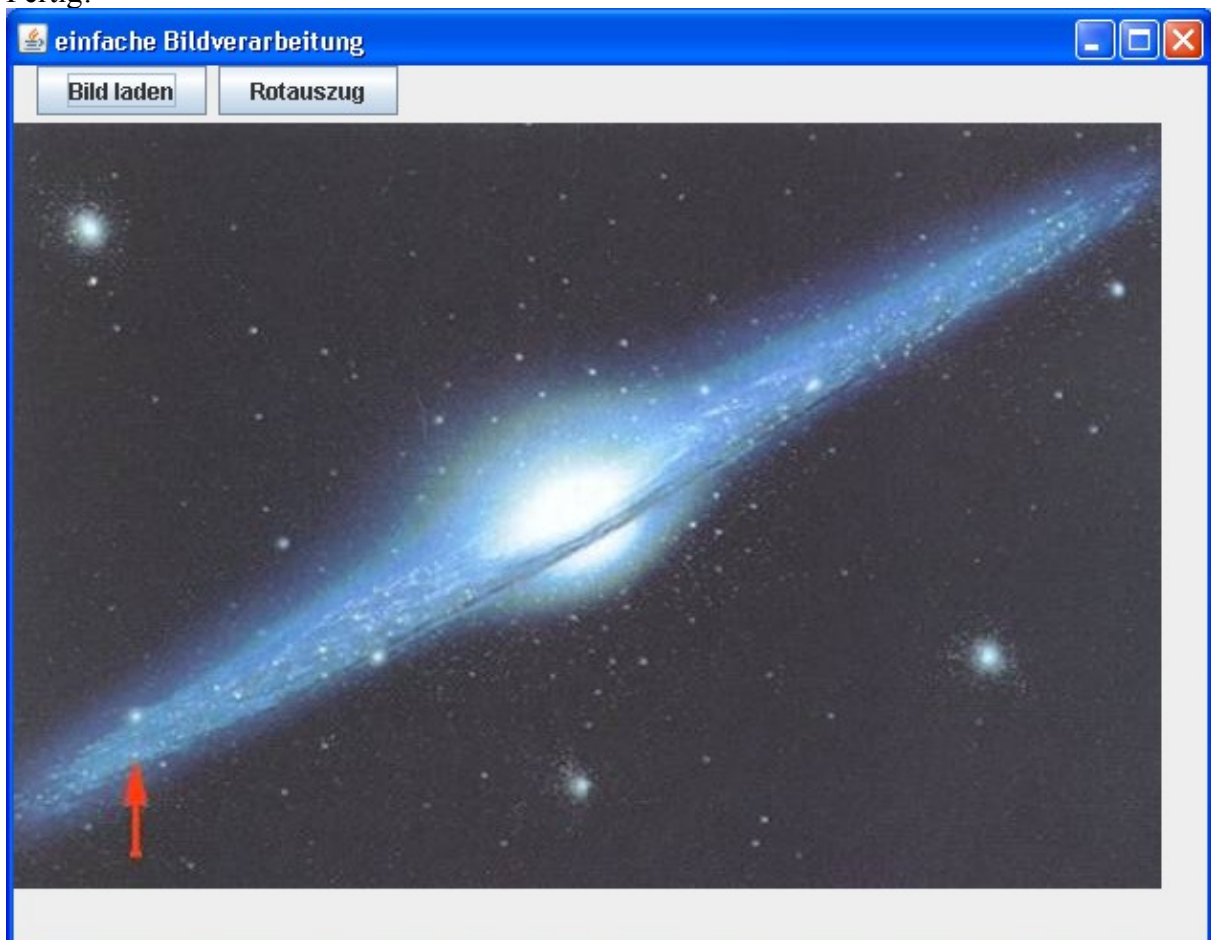
Im Eventhandler des `Lade`-Knopfes erfolgt die Anweisung, das Bild zu laden (hier: ohne Erfolgskontrolle). Danach wird das Fenster neu gezeichnet.

```
private void bLadenActionPerformed(java.awt.event.ActionEvent evt)  
{  
    bild = getToolkit().getImage("galaxie.jpg");  
    repaint();  
}
```

Natürlich muss dann in der `paint`-Methode des Fensters auch dafür gesorgt werden, dass das Bild erscheint – wenn es denn vorhanden ist.

```
public void paint(Graphics g)  
{  
    if(bild != null) g.drawImage(bild,0,60,this);  
}
```

Fertig!



Jetzt kommt es etwas heftiger.

- Da Ladevorgänge etwas dauern können, beauftragen wir ein **MediaTracker**-Objekt, das Laden zu überwachen. Damit verhindern wir, dass Bildverarbeitungsoperationen ausgelöst werden, bevor das Bild überhaupt vollständig geladen wurde.
- Weiterhin vereinbaren wir ein **BufferedImage**-Objekt, das als Kopie des Originalbildes dient, mit der wir arbeiten. **BufferedImage**-Objekte können **Image**-Objekte ersetzen, lassen aber einfacher als diese Zugriffe auf einzelne Pixel zu. Die Variable **breite** und **hoehe** sollen die Maße des Bildes aufnehmen.
- In der **paint**-Methode zeichnen wir jetzt die Kopie statt des Originals, um eventuelle Änderungen sichtbar zu machen.

```
import java.awt.*;
import java.awt.image.*;

public class FrameBildverarbeitung extends javax.swing.JFrame
{
    Image bild          = null;
    BufferedImage kopie = null;
    int breite, hoehe;

    public void paint(Graphics g)
    {
        if(kopie != null) g.drawImage(kopie, 0, 0, this);
    }

    private void bLadenActionPerformed(java.awt.event.ActionEvent evt)
    {
        bild = getToolkit().getImage("galaxie.jpg");

        MediaTracker mt = new MediaTracker(this); //Erfolgskontrolle
        mt.addImage(bild, 1);
        try
        {
            mt.waitForID(1);
        }
        catch (InterruptedException ex)
        {}

        breite = bild.getWidth(this); //Bildmaße bestimmen
        hoehe = bild.getHeight(this);
        // In die Kopie das Originalbild "hineinzeichnen"
        kopie = new BufferedImage(breite, hoehe, BufferedImage.TYPE_INT_ARGB);
        Graphics g = kopie.getGraphics();
        g.drawImage(bild, 0, 0, null);

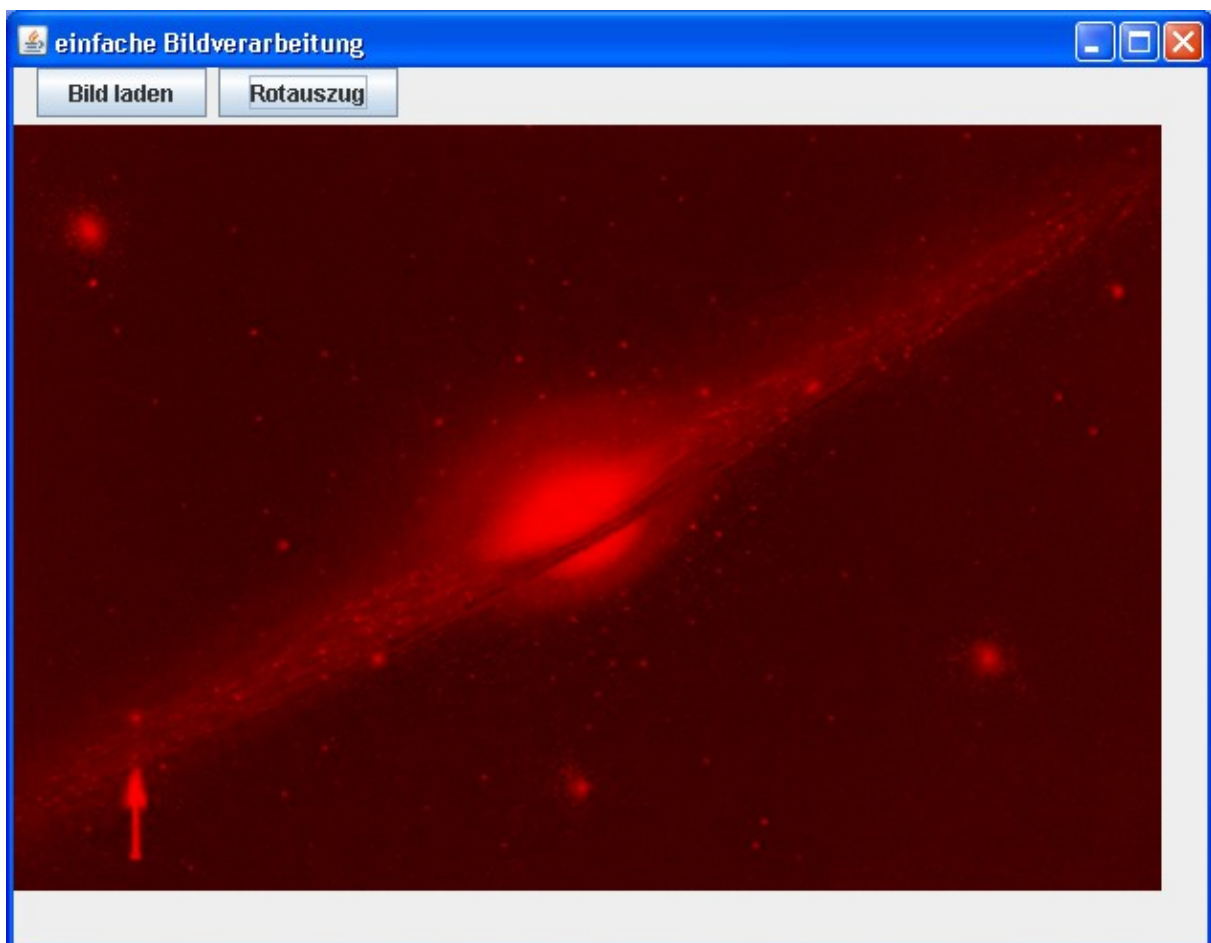
        repaint();
    }
}
```

So, und jetzt erstellen wir einen Rotauszug des Bildes: wir zeigen nur den Rotanteil der Bildpunkte. Dazu

- Stellen wir zur Sicherheit eine neue Kopie des Originalbildes,
- durchlaufen wir die zweidimensionale Reihung der Bildpunkte,
- holen und die Farbe der einzelnen Bildpunkunkte,
- reduzieren diese auf den Rotanteil
- und schreiben das Ergebnis zurück.

```
private void bRotauszugActionPerformed(java.awt.event.ActionEvent evt)
{
    Graphics g = kopie.getGraphics(); //Originalbild wiederherstellen
    g.drawImage(bild, 0, 0, null);

    for(int x=0; x < breite; x++) //alle Bildpunkte durchlaufen
        for(int y=0; y<hoehe; y++)
        {
            int pixel = kopie.getRGB(x,y); //Bildpunkt holen
            Color c = new Color(pixel); //dessen Farbe ermitteln
            Color rot = new Color(c.getRed(),0,0); //Rotanteil bestimmen
            pixel = rot.getRGB(); //in Zahlenwert verwandeln
            kopie.setRGB(x,y,pixel); //diesen zurückschreiben
        }
    repaint();
}
```



5. Aufgaben

1. Führen Sie Knöpfe für Grün- und Blauauszüge ein.
2. Vertauschen Sie Rot- und Blauauszug. Sehen Sie sich das Ergebnis bei Gesichtern an.
3. Versuchen Sie eine Methode zu schreiben, die ein Muster im Bild finden kann, z. B. ein Kreuz.