

**Inhalt**

## Rekursionen Teil 1

Einführung.....	1
Türme von Hanoi.....	2
Planung des Programms.....	4
Rekursionsablauf und Stack.....	6
Javaapplet für Türme von Hanoi .....	8
Verbesserung des Programms.....	10
Programmlisting 2 .....	12
Verzögerungen / Animation.....	15
Alternativen.....	20

## Rekursionen Teil 2

Fakultätsberechnungen.....	21
Sequenzdiagramm.....	21
Programmablauf (schrittweise) .....	22
Javaapplet .....	23
Aufgabe .....	24
primitiv-rekursive Funktionen .....	25
Übungsaufgaben.....	25
Ackermannfunktion .....	26
Übungsaufgaben.....	28
Übungen - Ulamfunktion .....	29
Hofstadterfunktion.....	30
Rekursion und Grafik .....	30
einfache Muster.....	31
Sierpinski dreieck .....	33
Übungen.....	34
Turtlegrafik .....	34
Turtlegrafik Java.....	35
Übungsaufgaben.....	39
Turtlerekursionen .....	40
Ein binärer Baum .....	41

Javaapplet zum binären Baum.....	42
Ternärer Baum.....	44
Übungsaufgaben.....	45
L-Systeme.....	46
Initiator und Generator .....	47
Kochkurven, Cesaro und Hilbert.....	48
Aufgabe: PentaPlexity ( Penrose) .....	51

### Rekursionen Teil 3

Backtracking .....	52
Einführung.....	52
Das Springerproblem .....	52
Planung der Lösung .....	53
Erläuterung.....	54
Planung eines Javaprogramms .....	57
Lösungsschritte .....	59
Teile des Listings.....	60
Weitere Informationmen.....	62
Das Acht-Damen-Problem.....	63
Lösungshilfen.....	64
Teile des Programms .....	66
Lösungsbeispiele und Aufgaben .....	69
Rekursive Klassen und Abschluss .....	70
Aufgabe:Permutationen.....	72

## Rekursionen

Rekursionen spielen in den Programmiersprachen eine große Rolle. Sie gelten mit Recht als ele-



gant und sind eine Zeichen wahrer Programmierkunst. Deswegen wollen wir das Schülerinnen und Schülern und uns selbst nicht vorenthalten. Rekursionen sind scheinbar leicht zu durchschauen und machen dennoch große Schwierigkeiten, wenn sie in ein Programm umgesetzt werden sollen oder wenn rekursive Programme analysiert werden. Eine sehr knappe und präzise Definition stammt von N.Wirth ( das war der Pascalerfinder )<sup>1</sup> :

*"Ein Objekt heißt rekursiv, wenn es sich selbst als Teil enthält oder mithilfe von sich selbst definiert ist."*

In der Mathematik kommen Rekursionen vielfach vor und wir werden entsprechende Beispiele behandeln.

Als Einstieg für den Unterricht kann man möglichst einfache übersichtliche Beispiele aussuchen, um das Wesen rekursiver Methoden zu erläutern oder gleich mit einem komplexen Problem beginnen, das dann schrittweise zerlegt wird. Beides ist möglich und sinnvoll. Hier wird der zwei-

<sup>1</sup>

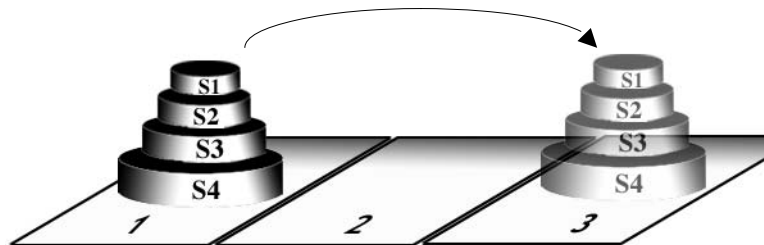
N. Wirth, Algorithmen und Datenstrukturen, Teubner Stuttgart, Seite 149

te Weg gewählt, daher sollte man sich nicht wundern, wenn es gleich ( aus Schülerinnen - und Schülersicht ) recht schwierig losgeht und danach sehr viel einfacher wird.

Ein Klassiker unter den Rekursionen sind die "TÜRME VON HANOI".

In allen Computersprachen, die Rekursionen unterstützen, ist dieses Problem schon behandelt worden. Kaum ein Informatikbuch, in dem nicht mindestens eine Lösung zu finden wäre. Oftmals jedoch aus der Sicht von Personen geschrieben, die die Verfahren sicher durchschaut haben. Das ist bei Schülerinnen und Schülern aber nicht der Fall und daher nun noch ein weiterer Versuch, dieses Problem für den Unterricht und in der Programmiersprache Java aufzuarbeiten.

Das Problem:



Ein Turm aus 4 ( und später beliebig vielen ) Scheiben soll von einem Platz "1" auf einen Platz "3" verschoben werden. Dabei gelten folgende Regeln:

- es darf immer nur eine Scheibe bewegt werden.
- man darf immer nur eine kleiner Scheibe auf eine größere legen, nie eine große auf eine kleine

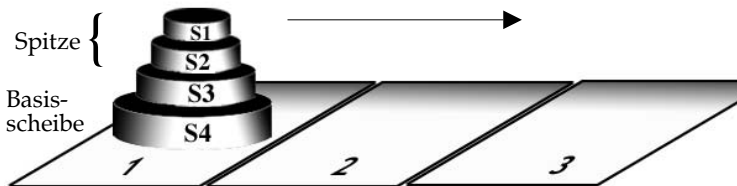
Für Kinder gibt es diese Türme von Hanoi aus bunten Holzscheiben oder auch Plastik und oft mit mehr als 4 Scheiben. Kinder halten sich aber kaum an die Spielregeln, aber unsere Schülerinnen und Schüler schon. Wie ist die Verschiebung mit möglichst wenigen Zügen zu realisieren ?

Hier kurz die Lösung:

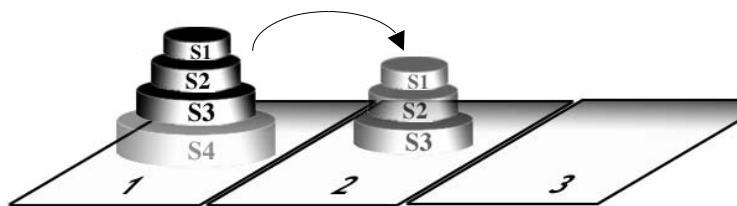
Zug	Scheibe	von	nach
1	S1	1	2
2	S2	1	3
3	S1	2	3
4	S3	1	2
5	S1	3	1
6	S2	3	2
7	S1	1	2
8	S4	1	3

Zug	Scheibe	von	nach
9	S1	2	3
10	S2	2	1
11	S1	3	1
12	S3	2	3
13	S1	1	2
14	S2	1	3
15	S1	2	3

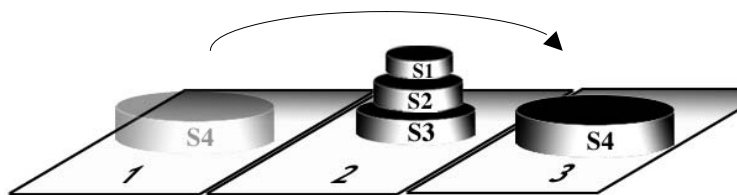
Und ? Schon eine Rekursion erkannt ? Das ist in diesem Beispiel nicht ohne Weiteres zu sehen. Schülerinnen und Schüler brauchen erfahrungsgemäß recht lange, um ihr Vorgehen so sortiert aufzuschreiben, dass eine "Regelmäßigkeit" erkennbar wird.



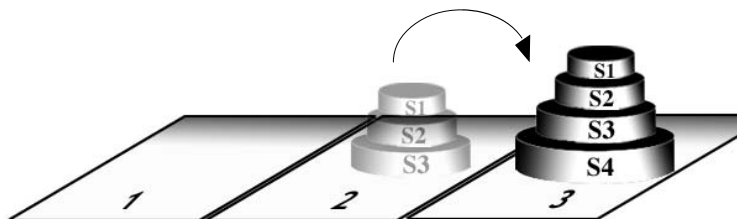
Den Turm von Platz 1 auf Platz 3 zu bewegen, bedeutet....



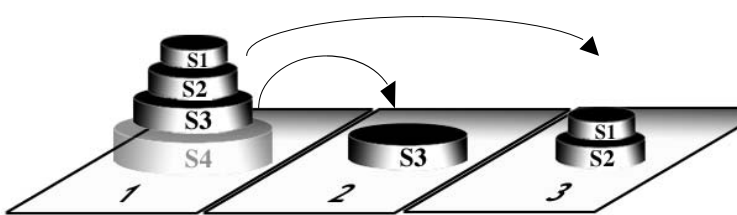
die Spitze auf das Zwischenlager ( Platz 2 ) abzulegen.....



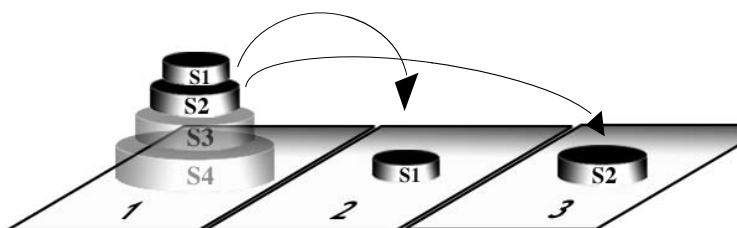
dann die Basisscheibe (S4) auf das Ziel ( Platz 3 ) zu legen und.....



die Spitze vom Zwischenlager ebenfalls auf den Zielplatz zu packen.



Das entspricht aber noch nicht ganz den Regeln, da wir mehr als eine Scheibe bewegt haben.



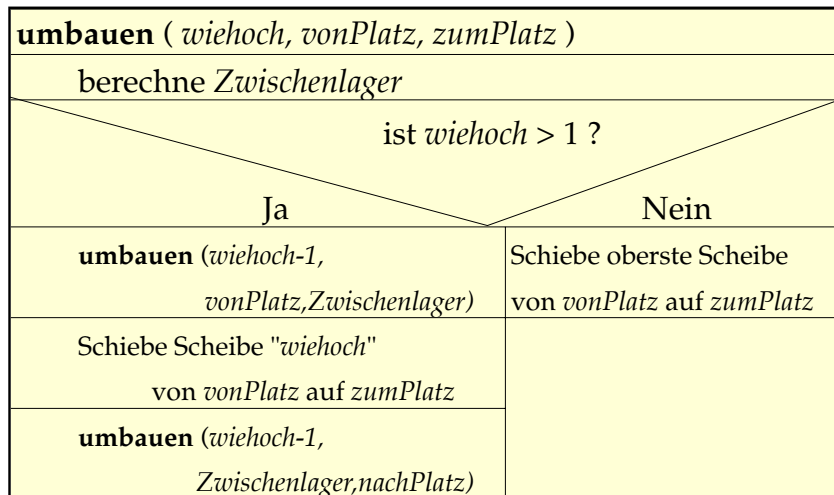
Der Transport der Spitze (S1 bis S3 ) auf das Zwischenlager ( Platz 2 ) geht aber nach dem gleichen Verfahren. Jetzt bildet die Scheibe S3 die Basisscheibe und die beiden oberen Scheiben sind die Spitze.

Nun wird also die Spitze (S1 und S2 ) auf das Zwischenlager ( jetzt Platz 3 )

gepackt, dann die Basis auf Platz 2 . Auch das Verschieben der Spitze ( S1 und S2 ) auf Platz 3 geht wieder nach dem bekannten Schema: Spitze ( S1 ) auf Zwischenlager (Platz 2), Basisscheibe

( S2 ) auf Zielplatz ( Platz 3 ) und dann S1 aus dem Zwischenlager auch auf das Ziel.

Bleibt nun die Frage, wie man soetwas übersichtlich formuliert. Wir beschreiben die Methode wie folgt:



Dieses Struktogramm wird für die weiteren Erläuterungen etwas modifiziert, um den Ablauf besser darstellen zu können.

Programm- schritt	Aufrufebene 1
	<b>umbauen</b> ( <i>wiehoch</i> , <i>vonPlatz</i> , <i>zumPlatz</i> )
1	<b><i>zwischenlager</i> = 6 - <i>vonPlatz</i> - <i>zumPlatz</i></b>
2	wenn <b><i>wiehoch</i> &gt; 1</b> dann
3	<b>umbauen</b> ( <i>wiehoch-1</i> , <i>vonPlatz</i> , <i>zwischenlager</i> )
4	Scheibe <b><i>wiehoch</i></b> von <b><i>vonPlatz</i></b> auf <b><i>zumPlatz</i></b>
5	<b>umbauen</b> ( <i>wiehoch-1</i> , <b><i>zwischenlager</i></b> , <i>zumPlatz</i> )
	sonst
6	Scheibe 1 von <b><i>vonPlatz</i></b> auf <b><i>zumPlatz</i></b>
7	Ende

Es lohnt sich, zum Verständnis rekursiver Verfahren, diese Methode nun genauer zu betrachten. Ein kleines Problem ist die Berechnung der Variablen *zwischenlager*, die angibt, wo die Spitze während des Umbaus abgelegt werden soll, damit man die Basisscheibe vom Start ( Variable *vonPlatz* ) auf das Ziel ( Variable *zumPlatz* ) legen kann.

Mit etwas Geschick kann man am oben genau dargestellten Beispiel erkennen:

$$zwischenlager = 6 - vomPlatz - zumPlatz$$



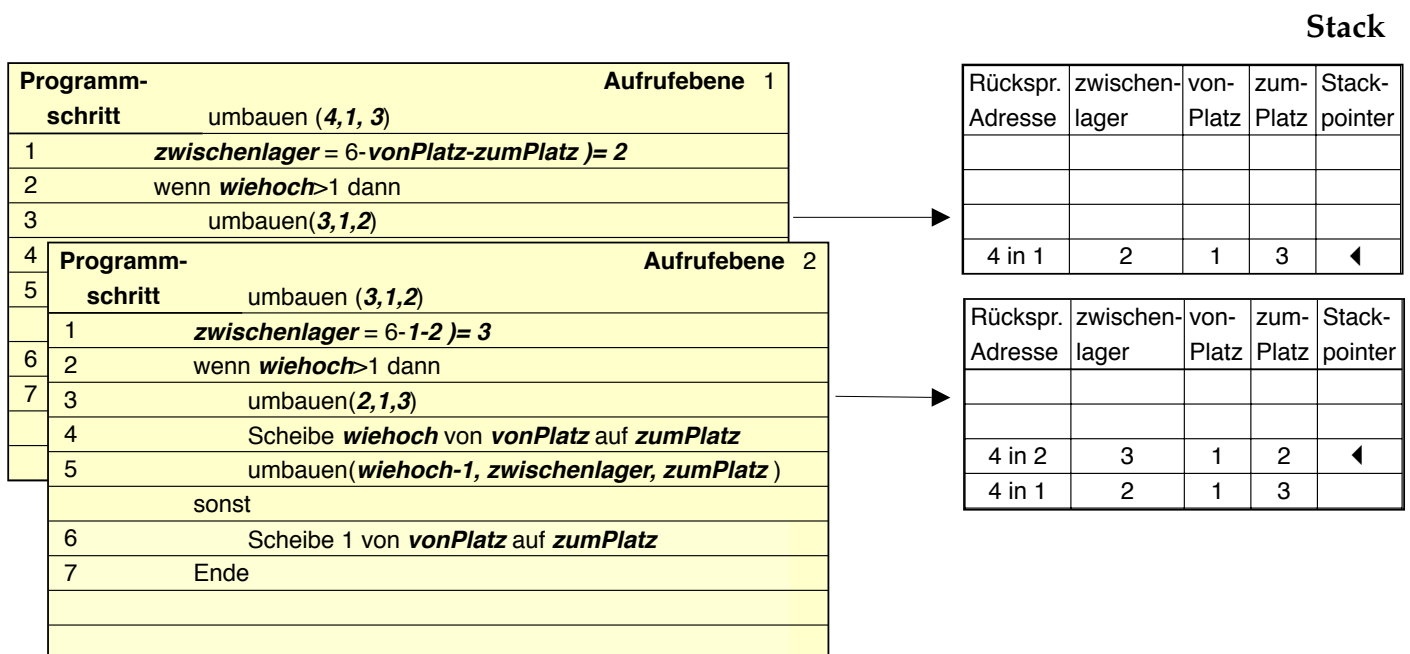
Damit ist das letzte verbleibende Problem gelöst ( könnte man glauben ), und wir schauen uns das obige Beispiel noch einmal an. Diesmal werden wir der Methode folgen und die notwendigen Variablenwerte im Auge behalten. Wenn alles klappt, muss es mit den Bilddarstellungen der vorangegangenen Seiten zusammenpassen. Die Zwischenwerte wollen wir auf einem Stapel (Stack) ablegen, der von unten nach oben wächst.

Aufrufebene 1, **umbauen** (4,1,3) ,Programmschritt 1:  $zwischenlager = 4-1-3 = 2$

Aufrufebene 1,Programmschritt 2: *wiehoch* ist  $> 1$  daher ....

Aufrufebene 1, Programmschritt 3: **umbauen**(3,1,2)

Nun wird die Methode **umbauen** erneut aufgerufen. Der Rechner merkt sich auf dem Stack, den



nächsten Programmschritt ( die sog. Rücksprungadresse) und die aktuellen Werte aller Variablen .

Aufrufebene 2, **umbauen** (3,1,2) ,Programmschritt 1:  $zwischenlager = 6-1-2 = 3$

Aufrufebene 2 ,Programmschritt 2: *wiehoch* ist  $> 1$  daher ....

Aufrufebene 2, Programmschritt 3: **umbauen**(2,1,3)

Wieder muss die Methode **umbauen** aufgerufen werden. Wieder merkt sich das Programm auf dem Stack die Rücksprungadresse ( hier Befehl 4 in Aufrufebene 2, und die Werte der Variablen) . Es geht nun in Aufrufebene 3 weiter.

Aufrufebene 3, **umbauen** (2,1,3) ,Programmschritt 1:  $zwischenlager = 6-1-3 = 2$

Aufrufebene 3,Programmschritt 2: *wiehoch* ist  $> 1$  daher ....

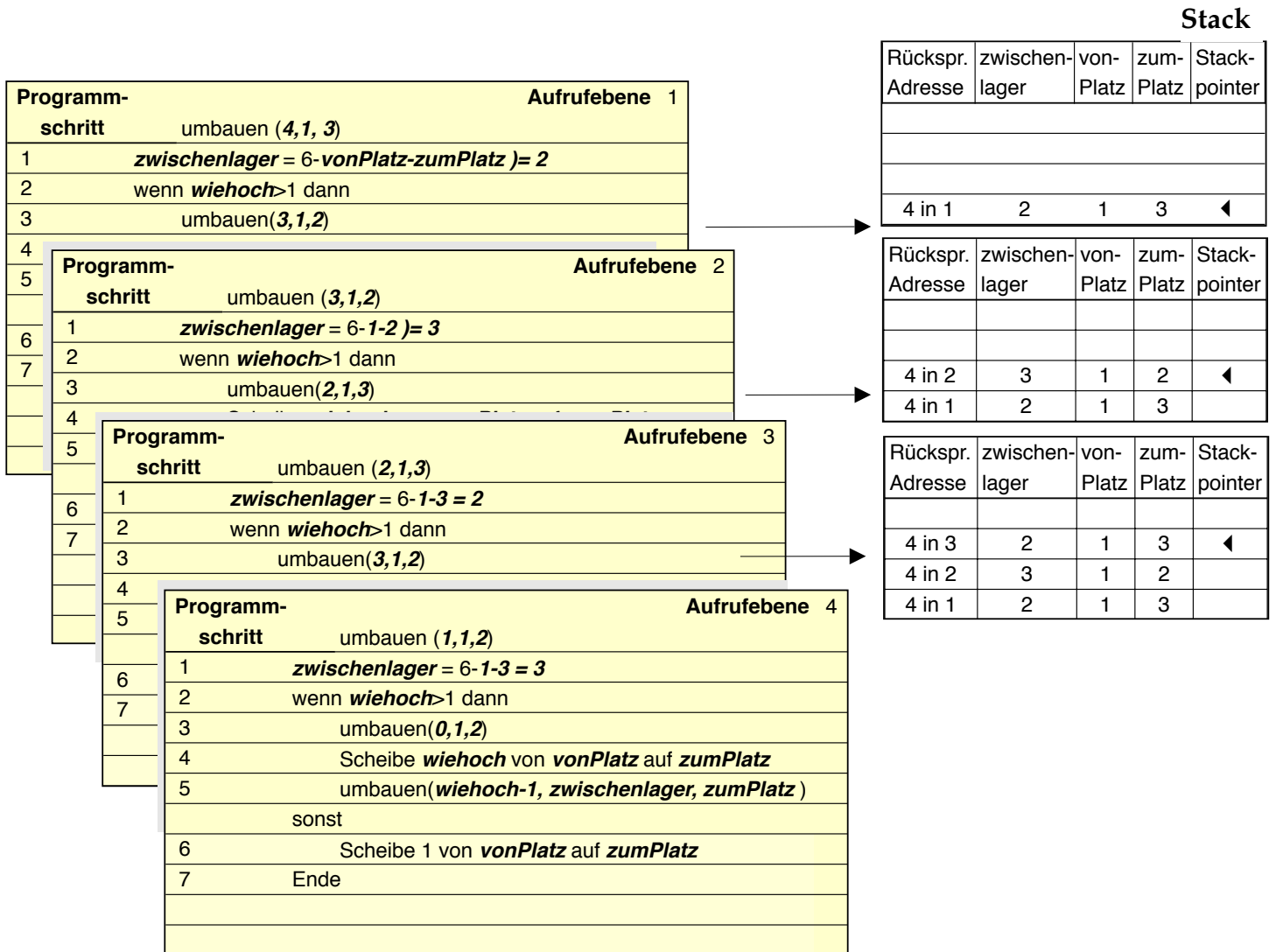


Aufrufebene 3, Programmschritt 3: **umbauen**(1,1,2)

Wieder muss die Methode **umbauen** aufgerufen werden. Wieder merkt sich das Programm auf dem Stack die Rücksprungadresse (hier Befehl 4 in Aufrufebene 3, und die Werte der Variablen) . Es geht nun in Aufrufebene 4 weiter.

Aufrufebene 4, **umbauen** (3,1,2) ,Programmschritt 1:  $zwischenlager = 6-1-2 = 3$

Aufrufebene 4,Programmschritt 2: *wiehoch* ist = 1 daher ....



Aufrufebene 4, Programmschritt 6: **Scheibe 1 von Platz 1 auf Platz 2**

**Aufrufebene 4, Programmschritt 7: Ende**

Nach dem letzten Programmschritt in Aufrufebene 4, wird an der Stelle fortgesetzt, auf die der Stackpointer zeigt. Es ist die Rücksprungadresse 4 in Aufrufebene 3 , mit den Variablenwerten, die ebenfalls auf dem Stack liegen. Die oberste Lage im Stack wird gelöscht und der Stackpointer

um eine Stelle nach unten gesetzt.

Nun geht es also in Aufrufebene 3 weiter:

Aufrufebene 3, Programmschritt 4, Scheibe S2 von 1 auf 3

Aufrufebene 3, Programmschritt 5, **umbauen**(3,2,3).

Nun wieder **umbauen** aufrufen und die Werte auf Stack lagern.

### Stack

Rückspr. Adresse	zwischen-lager	von-Platz	zum-Platz	Stack-pointer
4 in 2	3	1	2	◀
4 in 1	2	1	3	

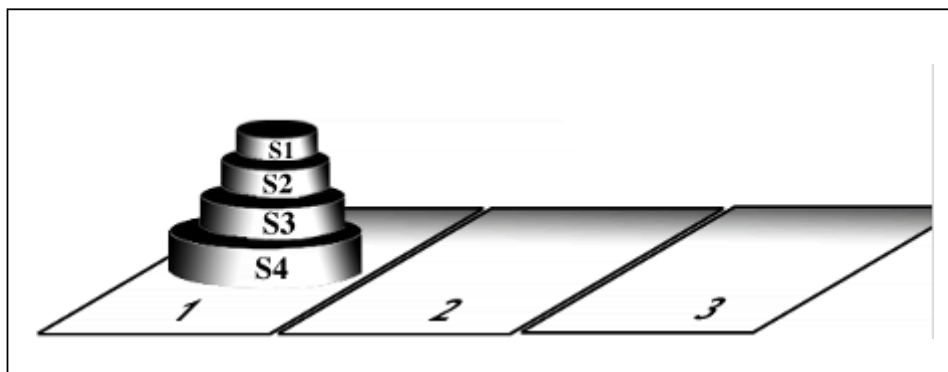
Da bei einer Rückkehr aus der nächsten Aufrufebene der Schritt 6 nicht abgearbeitet wird, wird die Rücksprungadresse mit dem Ende der Methode übereinstimmen (hier Befehl 7).

Dies scheint im ersten Moment nicht zu stimmen. Wenn man aber das ursprüngliche Struktogramm betrachtet, dann sieht man, dass es eine *if ... then ... else* -Verzweigung war, was bedeutet, dass **entweder** die Befehle 3, 4 und 5 **oder** die Befehle 6 und 7 ausgeführt werden. Ganz bestimmt werden also die Befehle 6 bis 7 nicht *nach* den Befehlen 3 bis 5 ausgeführt. Die sequentielle Darstellung des Struktogramms kann hier also zu Missverständnissen führen.

In dem beschriebenen Sinne geht es nun mit der Abarbeitung weiter, wobei der Stack immer wieder auf - und abgebaut wird.

Notieren sie sich selbst, wie die Werte auf dem Stack aussehen und spielen sie das Programm bis zum Ende durch.

Man sieht an diesem Beispiel, dass Rekursionen nur möglich sind, wenn die Programmiersprache Stacks unterstützt und verwaltet. Das war z.B. bei frühen Varianten von Basic nicht der Fall. Weiter kann man sehen, dass eine Rekursion eine Abbruchbedingung braucht, die dafür sorgt, dass die "Selbstaufrierei" zu einem Ende kommt. Dieses Problem wird uns in den späteren Beispielen noch beschäftigen.





## Ein Javaapplet für die Türme von Hanoi

Man ist leicht verführt, seine ganze Programmierkunst sofort in nette Animation oder grafische Darstellungen zu investieren. Da es aber in erster Linie um Rekursion geht, wollen wir zu Beginn ein sehr simples Programm schreiben, das das Verschieben der Scheiben nur als Textnachricht ausgibt.

```
// Hanoi1.java
// Ein erstes rekursives Programm. Hier ist die einfachste Art der
// Lösung gewählt- eine schlichte Textausgabe
// VLIN 30 H.-G.Beckmann, 8/02

import java.awt.*;
import java.applet.*;

public class Hanoi1 extends Applet
{
    TextArea    myArea=new TextArea("",20,50,TextArea.SCROLLBARS_VERTICAL_ONLY);
                // 20 Zeilen, 50 Zeichen

    public void init()
    {
        add(myArea);    // Textarea am Bildschirm
        umbauen(4,1,3); // Hier Aufruf der rekursiven Methode
    }
}
```

Man sieht hier, dass nicht mehr als eine TextArea gebraucht wird. Da unser Turm zuerst die feste Höhe von 4 Scheiben haben wird, die von Platz 1 auf Platz 3 verschoben werden , ist also der Aufruf `umbauen(4,1,3)`

Das Verschieben der Scheiben wird nun einfach in der TextArea ausgegeben.

```
/**
// Die rekursive Methode mit simpler Textausgabe
**/

public void umbauen (int wiehoch, int vonPlatz, int zumPlatz)
{
    int zwischenlager= 6-vonPlatz-zumPlatz;    // Berechnung des Zwischenlagers
    if (wiehoch > 1)
    {
        umbauen(wiehoch-1,vonPlatz,zwischenlager);
        myArea.append("Schiebe die Scheibe "+ wiehoch+" von Platz "+vonPlatz+" zum Platz
"+zumPlatz+"\n");}
}
```

```

umbauen(wiehoch-1,zwischenlager,zumPlatz);
}
else
{
myArea.append("Schiebe Scheibe 1 von Platz "+vonPlatz+" zum Platz "+zumPlatz+"\n");
} // Ende von else
} // Ende von umbauen
} // Ende von Applet

```

Das Programm ist recht kurz und hoffentlich übersichtlich. Das Ergebnis ist wie erwartet.

Nun kann man das mit 5, 6 oder mehr Scheiben laufen lassen, damit das Textfeld auch mal was zu scrollen hat.

Es bleibt aber natürlich eine grafische Darstellung das Ziel unserer Übung. Bei der Gelegenheit lassen sich viele bekannte Dinge wiederholen.



Wünschenswert ist eine Darstellung, bei der man einfach die Anzahl der Scheiben angeben kann, dann nur noch einen Startknopf mit der Maus klickt und dann eine nette Animation - also "fliegende Scheiben" - sieht.

Ordentlich geplant, muss zuerst überlegt werden, welche Klassen benötigt werden.

Von einer Scheibe, die in einem Turm liegt, müssen wir folgende Dinge wissen:

- Welche Nummer hat sie ?
- Auf welchem Platz liegt die gerade ?
- In welcher Höhe liegt sie ( in welcher Etage )?
- Wie breit ist sie ?
- Wie dick ist sie ?
- Welche Farbe hat sie ?

Weiterhin muss man mit der Scheibe etwas tun können:

- Man muss sie (am Bildschirm) malen können.
- Man muss sie (am Bildschirm) löschen können.
- Man muss sie verschieben können.
- Man muss sie auf einem Zielplatz die oberste freie Etage finden lassen können.

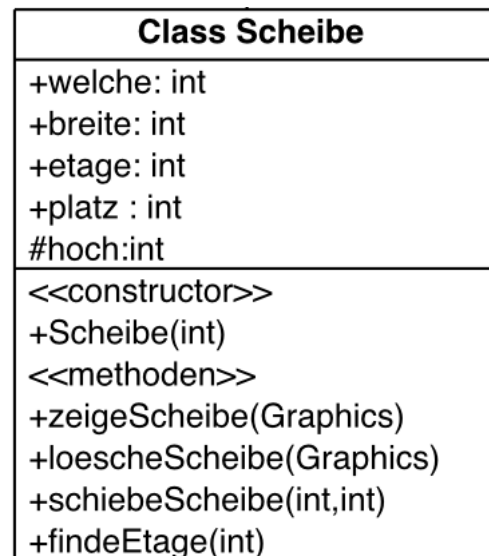
Damit könnte eine UML-Darstellung der *Scheibe* wie nebenstehend aussehen.

Das Attribut *hoch* muss nicht public sein.

Das Attribut *breite* wird sich aus der Scheibenummer ergeben.

Auf die Farbe kann man verzichten.

Es könnte auch beschlossen werden, dass die Methoden *schiebeScheibe* und *findeEtage* nicht zur Klasse gehören sollen, da es etwas ist, was nicht innere Eigenschaft des Objekts ist, sondern etwas, was mit dem Objekt *Scheibe* von außen gemacht wird.



Entsprechend wäre dann die Klasse anders zu definieren.

Weiter kann man ein Objekt / eine Klasse *Turm* definieren, die *Scheiben* enthält.

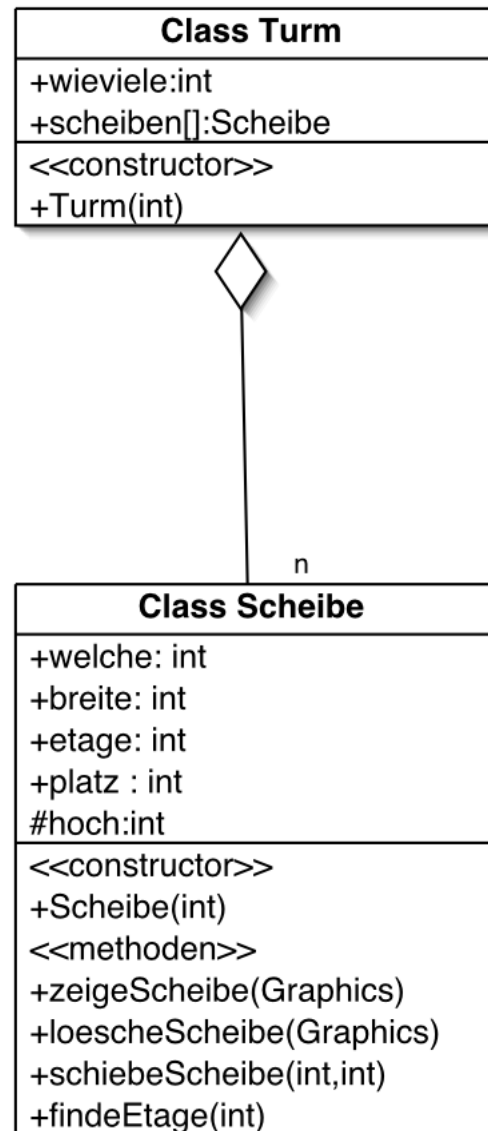
Das entsprechende UML-Diagramm sieht wie auf der nächsten Seite dargestellt aus:

Ein *Turm* besteht dann aus *n* *Scheiben*. Die Definition der Klasse *Turm* ist aber nicht nötig, da man gleich das ganze Applet nehmen kann , das neben vielen anderen Methoden auch eine enthalten kann, die den *Turm* aus Instanzen der Klasse *Scheibe* aufbaut.

Damit können wir nun das Javaprogramm beginnen.

```
//
// Hanoi3.java
//
// Die zweite Version der Türme.
// Diesmal mit Grafik
// Interessant sind die selbstdefinierten
// Klassen , die hier zum Einsatz kommen
//
// August 2004
// Hans-Georg Beckmann

import java.awt.*;
import java.applet.*;
import java.awt.event.*; // Für Button
```



Es wird also ein Array aus Scheiben gebaut. Nun muss man bedenken, dass die Indizes in Arrays mit 0 beginnen. Hat man 5 Scheiben, dann hat man die Nummern 0 bis 4 , um die Scheiben 1 bis 5 im Array zu finden. Da nicht gleich feststeht, wieviele Scheiben man haben will, wird das in der Definition offen gelassen. Das ganze Spiel besteht aus drei Plätzen, auf denen maximal alle *n*-Scheiben liegen können (das ist ganz am Anfang der Fall und ganz am Ende ). Während des Umbaus sind nicht alle möglichen Etagen auf den jeweiligen Plätzen belegt. Die Plätze benehmen sich also wie Stacks mit dem LIFO-Prinzip ( Last In First Out ). Immer die Scheibe, die oben liegt kann als erste wieder bewegt werden. Es muss also im Programm festgehalten werden, welche Etage auf welchem Platz belegt ist. Das geschieht in einem 2-dimensionalen Array.

```
public class Hanoi3 extends Applet
{
    Scheibe[] turm;           // Platz für einen Array aus Scheiben
    boolean[][] etageInPlatzfrei; // 2-Dim-Array für das ganze Spiel
    Button startbutton      = new Button("Start");           // der Startknopf
    Choice auswahl = new Choice();           // Auswahl für die Scheibenzahl
    Label anzahlLabel      = new Label("Scheibenanzahl"); // Beschriftung
    int wieviele;           // die Anzahl der Scheiben
    Lauscher meinLauscher=new Lauscher(); // ein ActionListener für den Button
}
```

Die Auswahl der Anzahl der Scheiben soll in einem PopUp-Menü erfolgen. Man braucht also eine Exemplar von *choice*, eine Beschriftung *label* und den Startknopf, der eine Instanz der Klasse *Button* ist. Diese Elemente werden initialisiert und im Layout plaziert. Am Ende wird die Methode *turmbauen* aufgerufen, die den Startturm erzeugt und darstellt.

```
public void init()
{
    setLayout (null);
    startbutton.setBounds(300,450,100,20); // Position und Größe des Buttons
    add(startbutton); // rein ins Layout
    startbutton.addActionListener(meinLauscher); // der Knopf bekommt einen Listener
    // Nun die Werte für das Pop-Up-Menü einbauen
    for (int i=5; i<15;i++)
        {auswahl.add(Integer.toString(i));}
    // es werden Strings gebraucht, daher int---> string
    auswahl.setBounds(150,450,60,20); // Größe und Position
    add(auswahl); // rein ins Layout
    anzahlLabel.setBounds(60,450,100,20); // und noch eine Beschriftung
    add(anzahlLabel); // rein ins Layout
    wieviele=5; // Startwert , ändert sich
    turmbauen(); // siehe unten
} // Ende von init
```

Damit sind alle wichtigen Initialisierungen erledigt.










```

//*****
//  PAINT malt alle Scheiben
//*****

public void paint (Graphics g)
{
  g.clearRect(20,20,400,400);      // Bildschirm säubern
  for(int j=0;j<wieviele;j++)      // alle Scheiben malen
  {
    turm[j].zeigeScheibe(g);      // Stellt alle Scheiben dar. Methode
                                   // zeigeScheibe findet sich in der
                                   // Klasse Scheibe -- siehe unten
  }
} // Ende von paint

```

Nun kommt das Turmbauen. Um im Verlauf des Turmbaus feststellen zu können, in welcher Höhe eine Scheibe auf einem Platz abgelegt werden kann, muss man sich die drei möglichen Stapel anschauen können, um festzustellen, welche Etage schon belegt sind und welche nicht. Die Etagen seien von unten nach oben durchgezählt. Die Scheiben haben im Beispiel die Nummern 1 bis 4 (im Array *turm* die Indexnummern 0 bis 3 ). Die Plätze mit den Ziffern 1, 2 und 3 haben im Array die Indexnummern 0,1 und 2. Will man nun z.B. die Scheibe 1 auf den Platz 1 legen, dann muss man feststellen können, in welcher Etage sie zu liegen kommt. Dann muss im Platz 1 die Etage 2 nun belegt werden und die Etage 2 im Platz 3 kann freigegeben werden. Das sieht komplizierter aus , als es dann sein wird. Man muss nur beachten, wie herum in welchem Array gezählt wird. In unserem Beispiel benutzen wir für die Verwaltung ein zweidimensionales Feld mit boolschen Werten, um festzuhalten, welche Plätze belegt sind.

	Platz 1 mit der Indexnummer 0	Platz 2 mit der Indexnummer 1	Platz 3 mit der Indexnummer 1	
Etage 4				Index 3
Etage 3				Index 2
Etage 2				Index 1
Etage 1				Index 0



Zuerst wird aus der Auswahl die Scheibenzahl geholt. Das macht *getSelectedIndex*, wobei die Indexnummer in der Auswahl mit 0 beginnt. Da unsere erste Eintragung aber 5 ist, muss hier "+5" stehen.

```
/**
 * *****
 * // Turmbauen am Anfang
 * *****
 */
public void turmbauen()
{
    wieviele= auswahl.getSelectedIndex()+5; // hole Scheibenzahl aus choice
    turm=new Scheibe[wieviele];           // Platz für Array mit neuen Scheiben
    etageInPlatzfrei = new boolean[3][wieviele]; // 2-Dim-Array für das ganze Spiel
    for(int i=0;i<wieviele;i++) // im Array Nummern 0,..., wieviele-1
    {
        turm[i]=new Scheibe(i); // der Scheibe wird eine Nummer übergeben
        turm[i].platz=0; // alle Scheiben auf Platz 1 mit Index 0
        turm[i].etage=wieviele-1-i; // Etage im Turm von oben nach unten
    }
    for(int j=0;j<wieviele;j++)
    {
        etageInPlatzfrei[0][j]=false; // auf Platz 1 mit Index 0 ist alles belegt
        etageInPlatzfrei[1][j]=true; // auf Platz 2 mit Index 1 alles frei
        etageInPlatzfrei[2][j]=true; // auf Platz 3 mit Index 2 alles frei
    }
    repaint();
} // Ende von Turmbauen
```

Nun kommt eine kleine Methode, die mit der Rekursion garnichts zu tun hat.

Da aktuelle Rechner viel zu schnell sind, sieht man vom Scheibenschieben nichts und es sieht am Bildschirm so aus, als wenn sich nur der ganze Turm ( ruck-zuck) auf seinen neuen Platz bewegt. Wie macht man das Programm langsam ? Man könnte in mehrfach geschachtelten Wiederholschleifen den Logarithmus einer Sinusfunktion eines Zahlenwertes berechnen lassen. Eleganter ist aber die folgenden Methode, die so tut, als hätten wir es mit Threads zu tun. Benutzen sie die Methode einfach, ohne lange darüber nachzudenken.<sup>2</sup>

<sup>2</sup>

Siehe Lehrbuch Grundlagen der Informatik von Helmut Blazert, Spektrum Verlag S. 590 und das zugehörige Programm auf zugehöriger CD



```
/**
// auch wenn gar kein Tread benutzt wird, funktioniert diese Methode
//
public void delay()
{
    try
    {
        Thread.sleep(100);                // 100 Millisekunden Pause
    }
    catch (InterruptedException e)
    {
    }
}
```

Nun kommt die Kernmethode des Programms. Im Vergleich zu der Textversion ist hier eine kleine Änderung zu sehen: *zwischenlager= 3-vonPlatz-zumPlatz;*

Das liegt daran, dass die Plätze nun nicht mehr die Nummern 1, 2 und 3 tragen , sondern 0, 1 und 2. Genauso ist es jetzt in der if-Abfrage *if (wiehoch > 0)*, weil die Nummerierung der Scheiben von 0 bis wieviele - 1 geht.

Um zu verstehen, wie die Methoden *schiebeScheibe* funktioniert, muss man in der Klasse *Scheibe* nachschauen, die weiter unten zu finden ist.

```
/**
// Die rekursive Methode
//
public void umbauen (int wiehoch, int vonPlatz, int zumPlatz)
{
    Graphics g;                // Wird zum Malen gebraucht
    g=getGraphics();          // Hole dir den aktuellen Grafikkontext
    int zwischenlager= 3-vonPlatz-zumPlatz; // Berechnung des Zwischenlagers
    if (wiehoch > 0)
    {
        umbauen(wiehoch-1,vonPlatz,zwischenlager);
        delay();                // Pause einlegen
        turm[wiehoch].loescheScheibe(g);
        turm[wiehoch].schiebeScheibe(vonPlatz,zumPlatz);
    }
}
```



```
    turm[wiehoch].zeigeScheibe(g);
    paint(getGraphics());    // paint aufrufen mit dem Grafikkontext, den
    delay();                 // getGraphics() liefert
    umbauen(wiehoch-1,zwischenlager,zumPlatz);
}
else
{
    turm[0].loescheScheibe(g);
    turm[0].schiebeScheibe(vonPlatz,zumPlatz);    // Scheibe 1 mit Index 0
    turm[0].zeigeScheibe(g);                    // bewegen
} // Ende von else
} // Ende von umbauen

//*****
// Nun der ActionListener für Button
//*****

class Lauscher implements ActionListener
{
    public void actionPerformed (ActionEvent myEvent)
        // Achtung kein einfacher Event, sondern ein ActionEvent
    {
        Object myObjekt =myEvent.getSource();    // welches Objekt meldet sich ?
        if(myObjekt==startbutton)                // ist es etwa der Startbutton ?
        {
            repaint();                            // ... es geht los
            update(getGraphics());
            wieviele= auswahl.getSelectedIndex()+5; // Scheibenzahl
            turmbauen();
            umbauen(wieviele-1,0,2);              // die rekursive Methode
            paint(getGraphics());
        }
    } // Ende von actionPerformed
} // Ende von meinLauscher
```

Nun kommt am Ende die Klasse *Scheibe* mit den Methoden, die schon in *umbauen* benutzt wurden. Die Variable *hoch* gibt die Scheibendicke an , kann nicht verändert werden und soll auch gelten, wenn gar keine Scheibe existiert. Sie ist daher *final static* definiert.

```

//*****
// Die Klasse Scheibe
//*****

class Scheibe
{
final static int hoch=20;           // 20 Punkte dicke Scheiben
public int breite;                  // Wird aus der Nummer berechnet
public int welche;                  // Nummer der Scheibe
public int etage;                   // für die Darstellung; welcher Platz, welche Lage
public int platz;                   // drei Plätze gibt es Nummern 1,2,3 Index im Array 0,1,2

//*****
// Der Konstruktor
//*****

public Scheibe(int welche) // erzeugt eine Scheibe mit der passenden Nummer
{
this.welche=welche;               // Wert von außen wird hier nach innen übernommen
breite=(welche+1)*16;             // breite ergibt sich aus der Scheibenummer
}

```

In den nachfolgenden Malmethoden werden Bildschirmkoordinaten aus Platznummer und Etagennummer für jede Scheibe berechnet. Man hat z.B. 40 Punkte vom rechten Rand Abstand. Die drei Plätze sind dann 0\*150, 1\*150 oder 2\*150 Punkte weiter in positive x-Richtung angesetzt. Je nach Scheibenummer wird um Vielfaches von 8 Punkten eingerückt u.s.w. Man kann diese Werte nach Belieben ändern, solange noch alle Scheiben ins Appletfenster passen.

Die eine Methode zeichnet die jeweilige Scheibe neu, die andere Methode löscht die entsprechende Scheibe durch Übermalen mit Weiß.



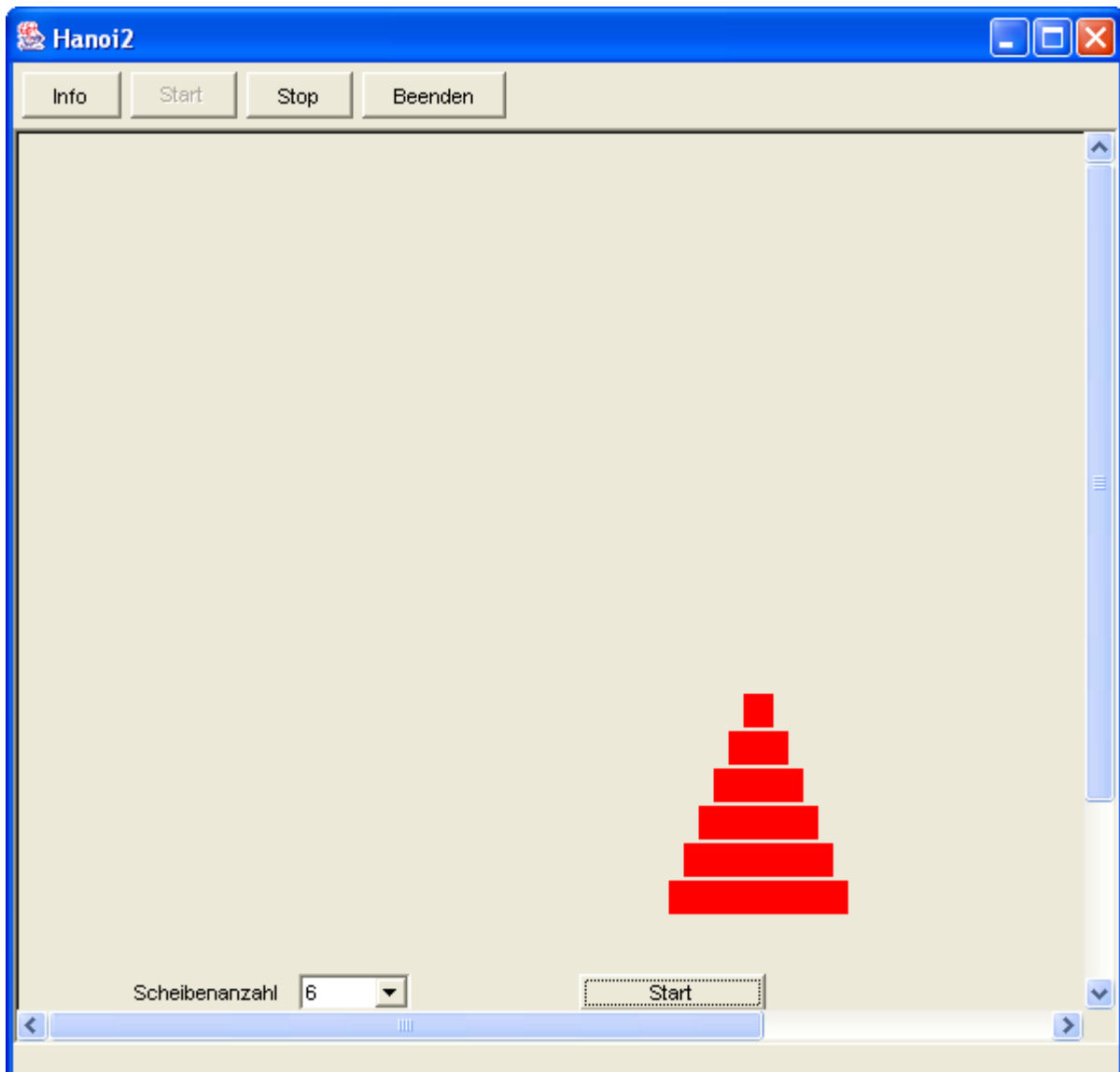
```
//*****  
//  zeige die Scheibe in der Grafik  
//*****  
public void zeigeScheibe(Graphics g)  
{  
    g.setColor(Color.red);           // unsere Scheiben sind rot  
    g.fillRect(40+150*platz+(wieviele-welche)*8,400-(etage)*hoch,breite,hoch-2);  
} // hoch - 2, damit zwischen den Scheiben ein kleiner Spalt bleibt  
//*****  
// lösche die Scheibe in der Grafik  
//*****  
public void loescheScheibe(Graphics g)  
{  
    g.setColor(Color.white);  
    g.fillRect(40+150*platz+(wieviele-welche)*8,400-(etage)*hoch,breite,hoch-2);  
}  
//*****  
// schiebe eine Scheibe bedeutet, dass die  
// entsprechenden Variablen umgesetzt werden  
//*****  
public void schiebeScheibe(int vonPlatz, int zumPlatz)  
{  
    int altetage;  
    altetage=etage;           // alter Wert der Variablen  
    platz=zumPlatz;         // neuer Platz für die Scheibe  
    etage=findeEtage(platz); // freie Etage auf dem Ziel finden =  
                           //neuer Etagenwert für Scheibe  
    etageInPlatzfrei[vonPlatz][altetage]=true; // Werte umsetzen  
    etageInPlatzfrei[zumPlatz][etage]=false;  
}  
//*****  
//      Man muss auf dem Zielplatz feststellen können,  
//      in welcher Etage der oberste freie Platz  
//      zu finden ist, weil dort die Scheibe hingeschoben wird  
//*****  
public int findeEtage(int Platz) // Platz kann 0,1 oder 2 sein  
{  
    int i;  
    int j=0;  
    for (i=wieviele-1;i>-1;i--)
```

```

{
  if (etageInPlatzfrei[Platz][i]==true) {j=i;}
} // von oben her abgearbeitet
return j;
} // Ende von findeetage
} // Ende von Klasse Scheibe
} // Ende vom Applet

```

Lassen sie das Programm mit verschiedenen Scheibenzahlen laufen. Ändern sie bei Bedarf den Verzögerungswert in der Methode *delay()*.



Eine "richtige" Animation ist das natürlich nicht, was da zu sehen ist. Eigentlich sollte man nun als fleißiger Programmierer eine butterweiche Animation programmieren, bei der die Scheiben sanft über den Bildschirm gleiten. Wenn sie ganz viel lange Weile haben, tun sie es !

### **Alternativen zum vorgestellten Programm**

Eine studierenswerte Lösung der Türme von Hanoi findet sich im Lehrbuch Grundlagen der Informatik von Helmut Blazert ( S. 588 ff). Der Autor macht sich dabei eine Klasse aus der Sammlung *java.util* zunutze. Die Klasse heißt **Stack** und ist genau das, was der Name vermuten läßt - ein Stack auf dem Objekte ( egal welche ) abgelegt werden können. Also kann man dort auch Instanzen einer selbstdefinierten Klasse *Scheibe* ablegen. Das Spiel besteht dann aus drei Stacks, für die alle das LIFO-Prinzip gilt. Die Klasse *Stack* stellt dabei Methoden zur Verfügung, die das Drauflegen eines Objektes auf einen Stapel ( *push* ), das Herunternehmen eines Objektes vom Stapel ( *pop* ) und das Erkennen des obersten Objektes ( *peek* ) möglich machen. Weiterhin gibt es die Abfrage *empty*, die feststellt, ob der Stapel leer ist.

( Genaueres dazu findet man auch in Guido Krüger: GotoJava 2 Kapitel 14.3 )

Mit der Klasse *Stack* lassen sich einige der oben benutzen Methoden ( *findeEtage ..* ) einsparen. Die Lösung ist dann etwas eleganter, als die hier vorgestellte.

Zum Schluss dieses Beispiels sein noch auf die Anzahl der Arbeitsschritte eingegangen, die die Rekursion braucht. Ist **n** die Scheibenzahl, dann braucht man zum Umbau  $2^{n-1}$  Schritte.

Bei 11 Scheiben sind das immerhin schon 2047 Schritte. Eine gute Übung für Schülerinnen und Schüler ist es, diese Formel durch vollständige Induktion zu beweisen.