

Ein endlicher Automat ist ein theoretisches Modell für eine informationsverarbeitende Maschine.

Gegeben ist er als 6-Tupel (Z, S, E, I, O, f) .

$Z = \{\text{Menge der Zustände}\}$, $S = \{\text{Startzustand}\}$, $E = \{\text{Endzustände}\}$, $I = \{\text{Eingabealphabet}\}$,

$O = \{\text{Ausgabealphabet}\}$, $f = \{\text{Überföhrungsfunktion } z_1 \times i \rightarrow z_2 \text{ (} z_i \text{ aus } Z, i \text{ aus } I)\}$.

Die Überföhrungsfunktion kann als Menge von 4-Tupeln (z_1, i, z_2, o) geschrieben werden, wobei o das Zeichen ist, das bei Anwendung der Regel ausgegeben wird.

Gestartet wird der Automat mit einem Eingabestring, während der Zustandswechsel werden Zeichen ausgegeben und der Eingabestring jeweils um das erste Zeichen verkürzt und schliesslich befindet der Automat sich in einem Endzustand oder nicht. Das erste Zeichen der Eingabe wird jeweils verarbeitet.

Beispiel:

$A = (\{z_1, z_2, z_3\}, z_1, \{z_2, z_3\}, \{a, b\}, \{a, b\}, f)$;

$f = \{(z_1, a, z_2, a), (z_1, b, z_3, b), (z_2, a, z_3, a), (z_2, b, z_2, b), (z_3, b, z_3, b), (z_3, a, z_2, a)\}$;

Dieser Automat befindet sich nach Ablauf im Endzustand z_2 , wenn das letzte Zeichen der Eingabe ein a war und im Zustand z_3 , wenn das letzte Zeichen ein b war.

Die Berechnungskraft eines solchen endlichen Automaten ist natürlich sehr eingeschränkt. Das Programm Automat in dieser Einführung ermöglicht die Konstruktion und das Testen von endlichen Automaten. Bei nichtdefinierten Eingaben gerät das Programm in einen nichtdefinierten Zustand (Programmabsturz). Mit einer komfortablen Oberfläche können Zustände und zugehörige Regeln kreierte werden. Die Programme zu diesem Text gibt es unter <http://www.muenster.de/~dambergj/qlr.zip> in digitaler Form.

Moderne Rechner sind angenäherte Universalrechner, die durch Programme gesteuert werden und aus Steuerwerk, Rechenwerk, Speicher und Ein/Ausgabegeräten bestehen. Sie stellen sich dem Programmierer mit ihrer Makroarchitektur dar. Diese umfasst im Wesentlichen Befehlssatz und Registerstruktur. Register sind extrem schnelle Speicherplätze, die nur einen Wert fassen können und die als Quelle und Ziel von Befehlen verwendet werden, wobei die Adressierung indirekt sein kann. Dann enthält das Register eine Speicheradresse, auf die zugegriffen wird.

Zur Erläuterung dieses Konzepts gehört ein einfacher Assembler für eine virtuelle Maschine inklusive Interpreter zu diesem Text. Die virtuelle Maschine ist ein Rechner mit vier Registern (A, B, C, D) und 4096×32 Bit Hauptspeicher, der ganze Zahlen der Länge 32 Bit addieren, subtrahieren, multiplizieren und dividieren sowie bedingte Sprünge im Programm durchführen kann. Wichtig: Beim Programmieren des Assemblers wird erzwungen, dass nur ein Befehl pro Zeile verwendet wird und dieser mit einem Semikolon endet.

Befehle:

LOAD R1 constant;

Dem Register R1 wird ein konstanter Wert zugewiesen.

Beispiel:

LOAD A 120;

Dieser Befehl weist dem Register A den Wert 120 zu.

LOAD R1 R2; (R1 und R2 Register)

Mit diesem Befehl wird der Wert des Registers R2 dem Register R1 zugewiesen.

Beispiel:

LOAD A B;

=>A=:B

ADD R1 R2;

Die Werte in R1 und R2 werden addiert und in R1 zurückgeschrieben.

Beispiel:
ADD A D;
=>A=:A+B

SUB R1 R2;
R2 wird von R1 abgezogen und nach R1 geschrieben.

MULT R1 R2;
R1 und R2 werden multipliziert und in R1 gespeichert.

DIV R1 R2;
R1 wird durch R2 dividiert und das Ergebnis nach R1 kopiert.

ADD R1 constant;
SUB R1 constant;
MULT R1 constant;
DIV R1 constant;
(sind auch gültige Befehle)

LOAD R1 (R2);
Die Hauptspeicherzelle mit Index R2 wird nach R1 kopiert.

Beispiel:
LOAD A (B);
=>A=:mem[B]

STORE R1 (R2);
Der Wert in R1 wird in die Hauptspeicherzelle mit Index R2 geschrieben.

Beispiel:
STORE A (B);
=>mem[B]=:A;

Das zweite Argument eines ADD/SUB/MULT/DIV Befehls kann durch eine indizierte Hauptspeicherzelle ersetzt werden.

Beispiel:
ADD A (B);
=>A=:A+mem[B]

Im Programm können textuelle Sprungmarken untergebracht werden. Diese werden mit einem Doppelpunkt voran markiert. Es gibt drei verschiedene Sprungbefehle: JMP, JMPA, JE. JMP springt einfach so zum folgenden Label (zur gleichnamigen Sprungmarke), JMPA verwendet A als Sprungziel und JE springt nur, wenn das Register A den Wert 0 hat.

Beispiele:
:label1;
:label2;
JMP label1;
JMPA;
JE label2;

Jedes Programm muss mit dem Befehl END beendet werden, sonst gerät es in einen undefinierten Zustand.

END;

Diese Makroarchitektur entspricht einer Teilmenge der Befehle, die heutzutage auf Rechnern verfügbar sind, die allerwichtigsten Befehle sind aber implementiert. Ein wesentlicher Unterschied ist, dass Programm- und Datenspeicher voneinander getrennt sind. Dies liegt daran, dass so die Eingabe des Programms im Programm "Assembler" vereinfacht wurde. Sonst hätten in der Oberfläche noch Operationen zur Veränderung des Hauptspeichers Platz finden müssen, die aber zum Verständnis der Zusammenhänge überflüssig gewesen wären. Sobald Betriebssysteme einen Rechner steuern ist ein getrennter Programm und Datenspeicher überhaupt nicht mehr sinnvoll. Zur Implementierung des Assemblers wurde übrigens intern ein endlicher Automat mit mehr als 40 Zuständen benutzt. Nach Ablauf des Programms wird jeweils der Wert des Registers A ausgegeben. Der Wertebereich der Konstanten ist ≥ 0 .

Dieses Programm berechnet 5! (5 kann durch andere Werte ersetzt werden).

A wird solange dekrementiert und mit B multipliziert, bis A=0 ist. Dann springt das Programm zu go und der Wert von B wird nach A kopiert, damit er nach END; ausgegeben wird.

```
LOAD A 5;
LOAD B 1;
:start;
JE go;
MULT B A;
SUB A 1;
JMP start;
:go;
LOAD A B;
END;
```

Dieses Programm berechnet die Summe der Zahlen von 1 bis 5.

```
LOAD A 5;
LOAD B 0;
:start;
JE go;
ADD B A;
SUB A 1;
JMP start;
:go;
LOAD A B;
END;
```

Dieses Programm berechnet $5\%2$:

A wird solange um zwei reduziert, bis A=0 oder A=1 ist. A=1 stellt man fest, indem man A in C speichert und A dann um eins reduziert und, wenn A=0 gilt, bedingt springt.

```
LOAD A 5;
:start;
JE ok;
LOAD B A;
SUB B 1;
LOAD C A;
LOAD A B;
JE ok2;
LOAD A C;
```

```

SUB A 2;
JMP start;
:ok2;
LOAD A C;
:ok;
END;

```

Dieses Programm berechnet $3*3*3*3$, weil eine Schleife vier Mal durchlaufen wird und dann das Ergebnis von C nach A kopiert wird.

```

LOAD A 4;
LOAD B 3;
LOAD C 1;
:start;
JE weiter;
MULT C B;
SUB A 1;
JMP start;
:weiter;
LOAD A C;
END;

```

Dieses Programm speichert den Wert $4*12=48$ in der Hauptspeicherzelle 48 und lädt den Wert dieser Zelle in das Register A.

```

LOAD B 12;
MULT B 4;
STORE B (B);
LOAD A (B);
END;

```

Ein anderer Ansatz der Programmierung ist der der funktionalen Programmierung. Zu dieser Einführung gehört die funktionale Programmiersprache Funktional. Das Programm wird nicht sequentiell durchlaufen. Der Quelltext entsteht aus mathematischen Definitionen von Funktionen, die sich gegenseitig aufgrund von Bedingungen aufrufen. Gestartet wird eine Berechnung durch einen Funktionsnamen und eine passende Anzahl an Argumenten. Zentral sind die Schlüsselworte FUNCTION, INTERVAL, DIRECT, CALL und SUB. Ein Funktional-Programm setzt sich aus einer Folge von Funktionen zusammen jeweils gefolgt von Intervallen. Zu jedem Intervall gehört ein DIRECT oder ein CALL in der Zeile danach. Mit DIRECT kann man einen Wert zurückgeben (Ausdruck bestehend aus Variablen, Konstanten, +, -, *, /) und mit CALL ruft man eine andere Funktion auf oder die Funktion selbst rekursiv mit anderen Werten. Hinter ein CALL kann ein SUB gestellt werden, dieses Instrument substituiert in eine beliebige Variable des CALL eine andere Funktion.

Syntax:

```

FUNCTION fname x1 x2 ... xn;
fname Funktionsname; x1, ..., xn Variablen
INTERVAL x1 U..U x2 U..12 x3 23..25 ... xn -12..U;
x1, x2, ..., xn Variablen; U unendlich (+/-), 23..25 Intervall zwischen 23 und 25
Die Teilintervalle xi ai..bi sind allesamt und-verknüpft.
DIRECT ausdruck;
CALL fname ausdruck1 ... ausdruckn
SUB variable fname ausdruck1 ... ausdruckn

```

ausdruck:

Variablen sind Ausdrücke: a, b, c, x, y, z, ...

Konstanten: -100, 0, 234, ...

Sind a und b Ausdrücke, so sind

(a)+(b), (a)-(b), (a)*(b), (a)/(b) Ausdrücke.

Wenn mit "fname 12 23 34" ein Programm gestartet wird, wird zunächst gesucht, welche Funktion fname heisst und drei Argumente hat. Dann werden die Variablen dieser Funktion mit 12, 23 und 34 belegt. Die Intervalle werden in der Reihenfolge des Quelltextes durchsucht und wenn die drei Variablen in das Intervall passen, wird der zugehörige CALL oder DIRECT gestartet. Im Fall, dass DIRECT durchgeführt wird, werden die Variablen des Ausdrucks genauso wie die Argumente belegt und der Ausdruck ausgerechnet und zurückgegeben. Ansonsten werden die Argumente der aufzurufenden Funktion als Ausdrücke berechnet und der aufgerufenen Funktion übergeben. Eventuell gehört ein SUB zum CALL. Die Argumente der substituierten Funktion werden kalkuliert, der Wert der substituierten Funktion rekursiv geholt und in eine Variable des aktuellen Kontext geschrieben.

Ein paar Beispiele:

G zählt X auf 0 herunter und gibt Y zurück.

```
FUNCTION G X Y;  
INTERVAL X 0..0 Y U..U;  
DIRECT (X)+(Y);  
INTERVAL X U..U Y U..U;  
CALL G (X)-(1) (Y);
```

F X Y berechnet $Y*Y*Y*Y$.

```
FUNCTION F X Y;  
INTERVAL X 0..0 Y U..U;  
CALL G Y;  
SUB Y G Y;  
INTERVAL X U..U Y U..U;  
CALL F (X)-(1) Y;  
FUNCTION G Y;  
INTERVAL Y U..U;  
DIRECT (Y)*(Y);
```

SUM X berechnet die Summe der ganzen Zahlen von 1 bis X plus X.

```
FUNCTION SUM X;  
INTERVAL X U..U;  
CALL G X X;  
FUNCTION G X Y;  
INTERVAL X 0..0 Y U..U;  
DIRECT (Y);  
INTERVAL X U..U Y U..U;  
CALL G (X)-(1) (Y)+(X);
```

Sowohl funktionale Programmiersprachen als auch Assemblercode werden in der modernen Programmierung nur in Ausnahmefällen verwendet. Den Markt dominieren objektorientierte Hochsprachen wie JAVA und C++. Als nächstes soll eine Einführung in die Grundlagen der Programmkonstruktion mit JAVA gegeben werden.

Ein Vorteil von JAVA ist, dass das JAVA Development Kit von Sun Microsystems kostenlos ist. Man kann es unter <http://www.javasoft.com> herunterladen. Nach der Installation befinden sich auf der Festplatte der Compiler javac.exe und der Interpreter java.exe. Mit "javac Program.java" compiliert man das Programm in der Datei Program.java, wobei in der Datei die Klasse Program enthalten sein muss (später dazu mehr). Der Compiler compiliert den JAVA-Code für eine virtuelle Maschine. Diese kann als realer Prozessor gegeben sein (selten) oder als Interpreter. Mit "java Program" ruft man das compilierte Programm auf und die JAVA Virtual Machine führt es aus.

Der Grundaufbau einer JAVA Klasse ist:

```
class Program{  
}
```

Eine Klasse enthält optional static Methoden, die eine Liste von Argumenten übergeben bekommen und einen Wert zurückgeben oder nicht ("public (typ|void) methodName(Typ t1,...,Typ tn){Anweisungen}"). Diese static Methoden können nur ihre Argumente, Konstanten und spezielle Klassenvariablen bei ihren Berechnungen berücksichtigen.

Beim Starten des Programms mit java.exe wird automatisch die Methode "public static void main(String[]args){}" aufgerufen, die exakt so heissen muss.

```
class Program{  
    public static void main(String[]args){  
        System.out.println("hallo");  
    }  
}
```

Mit der Zeile "System.out.println("hallo");" wird der Text "hallo" auf den Bildschirm ausgegeben. Genauer: Die Methode println(String string) im Objekt out der JAVA-internen Klasse System wird aufgerufen und liefert keinen Wert zurück. Übrigens: JAVA unterscheidet auf allen Ebenen Gross- und Kleinschreibung.

Folgende Typen sind in JAVA fest integriert:

```
boolean (true,false)  
byte (0...255)  
short (16 Bit Ganzzahl)  
int (32 Bit Ganzzahl)  
long (64 Bit Ganzzahl)  
float (32 Bit Gleitkommazahl)  
double (64 Bit Gleitkommazahl)  
String ("hallo! dies ist ein String")
```

Das Umwandeln von Typen geschieht in JAVA mit einer Casting-Anweisung. Beispiel:

```
int integer=198;  
byte by=(byte)integer;
```

Ein Beispiel für Klassenmethoden (static Methoden):

(Klassenmethoden und Klassenvariablen existieren nur ein Mal für jede Klasse.)

```
class Program{  
    //ein Kommentar wie dieser wird beim Compilervorgang nicht berücksichtigt.  
    static int konstant=122;
```

Zu Programmstart hat die int-Klassenvariable den Wert 122

```
    public int statiemethod(){  
        return konstant+10;
```

Die Klassenvariable wird beim Aufruf dieser Methode um 10 inkrementiert zurückgegeben.

```
    }  
    public static void setKonstant(int k){
```

```
konstant=k;
```

void bedeutet, dass die Methode keinen Wert zurückgibt. Der Konstante wird der übergebene Wert k zugewiesen.

```
}  
public static void main(String[]args){  
    System.out.println("staticmethod()="+staticmethod());  
    setKonstant(12);  
    System.out.println("staticmethod()="+staticmethod());  
}
```

Diese Methode wird vom JAVA-Interpreter als erstes aufgerufen. Die Konstante wird (inkrementiert) abgefragt, der neue Wert 12 gleich gesetzt und es wird schließlich noch einmal abgefragt.

```
}  
}
```

```
class SumMethod{  
    public static void main(String[]args){  
        int sum=new Integer(args[0]).intValue();
```

Diese Methode wird beim Start aufgerufen ("java SumMethod args[0]") und der Kommandozeilenparameter in eine Integer-Zahl konvertiert.

```
        System.out.println("Summe: "+getSum(sum));
```

Die erste Ausgabe ist die Berechnung der Summe aller Zahlen bis sum auf rekursive Art.

```
        int sum2=0;  
        for(int i=1;i<=sum;i=i+1){  
            sum2+=i;  
        }
```

for-Schleifen haben folgenden Aufbau:

```
for(int i=constant;boolscher Ausdruck;Anweisung){  
}
```

Einer int-Variablen wird ein Wert zugewiesen, dann gerät das Programm in eine Schleife. Vor dem Durchlauf wird ein boolscher Ausdruck geprüft (zB. $a < b$, $a \leq b$, $a > b$, $a \geq b$, $a == b$) und nach der Schleife eine Anweisung ausgeführt (zB. $i++$, $i=i+2$, $i--$, $i=i-10$) und zwar so lange, bis die Bedingung nicht mehr erfüllt ist.

```
        System.out.println("Summe2: "+sum2);
```

Der Wert der Variablen sum2, zu der alle Werte von 1 bis sum addiert wurden, wird auf den Bildschirm geworfen.

```
    }  
    public static int getSum(int x){
```

int ist der Rückgabebetyp, x ist das übergebene Argument, das wie eine lokale Variable behandelt wird.

```
        if(x==0)return 0;  
        return x+getSum(x-1);
```

Diese rekursive Definition bewirkt das gleiche wie die for-Schleife. Beispiel:

```
getSum(4)=4+getSum(3)=4+3+getSum(2)=4+3+2+getSum(1)=4+3+2+1+getSum(0)=4+3+2+1
```

```
    }  
}
```

In JAVA wird die Gleichheit zweier Ausdrücke mit == geprüft (a==b ist ein boolescher Ausdruck), mit = werden Werte zugewiesen (zB. a=10, int a=120, a=((b*x)-c)+(b*c). Die Anweisung i++ ist eine Kurzform der Anweisung i=i+1. Tauchen neue Variablen in Methoden auf, so müssen sie mit ihrem Typnamen als lokale Variablen initialisiert werden ("int x=129;"). Die static Variablen, die nicht zu Methoden gehören, müssen direkt hinter der Klassendeklaration am Anfang der Datei deklariert werden.

Standardtypenvariablen können dort auch schon initialisiert werden ("static byte b=255;"). Boolesche Ausdrücke sind zum Beispiel a<b, a<=b, a>b, a>=b, a==b aber auch a2&&b2 (a2 und b2) und a2||b2 (a2 oder b2) wobei a2 und b2 boolesche Ausdrücke sind.

```
class Sorter {
```

"Straight Selection" ist ein langsamer Sortieralgorithmus. Der Methode sort wird ein Integer-array übergeben und dieses wird von ihr sortiert. Die Methode liefert keinen Wert zurück (void) sondern sortiert direkt im Array.

Von beliebigen Typen kann man Arrays erzeugen, dies sind Listen einer festen Länge, die n verschiedene Werte eines bestimmten Typs enthalten.

Beispiel: int n=5;int[]array=new int[n];array[2]=12; (Deklaration, Instanzierung, Initialisierung an Listenstelle 2, mit array[0] wird auf das erste Element der Liste zugegriffen)

Mehrdimensionale Arrays sind erlaubt (zB. int[][] md=new int[2][2];md[1][2]=3;).

In JAVA können mehrere Befehle in einer Zeile stehen, durch ein Semikolon getrennt, jedoch gilt dies als schlechter Programmierstil.

```
    public static void sort(int[] array){  
        int n=array.length;
```

Mit dieser Zeile speichert man die Länge des Arrays in der Variablen n.

```
        for(int i=0;i<n;i++){  
            int smallest=array[i];  
            int sindex=i;
```

Das i-te Element ist vor dem Vergleich mit den Elementen mit höherem Index das kleinste Element.

```
            for(int j=i+1;j<n;j++){  
                if(array[j]<smallest){  
                    sindex=j;  
                    smallest=array[j];
```

Ist das Element an der Stelle j kleiner, so werden j und der Wert an der Stelle j gespeichert.

```
            }  
        }  
        int temp=array[i];  
        array[i]=smallest;  
        array[sindex]=temp;
```

Das Element an der Stelle i wird mit dem kleinsten Element mit höherem Index (sindex) vertauscht, dabei muss das Element an der Stelle i zwischengespeichert werden.

```
    }  
  }  
}
```

```
class Prim {  
    public static void primBis(int n) {  
        for(int i=2; i<=n; i++) {
```

Die Zahlen von 2 bis n werden getestet, ob sie Primzahlen sind.

```
            boolean prim=true;  
            for(int j=2; j<=(int)Math.sqrt((double)i); j++) {  
                if(i%j==0) prim=false;
```

Hat die Zahl einen Teiler, so ist sie keine Primzahl. $i\%j$ berechnet den Rest der Division von i durch j. Ist der Rest 0, so ist i durch j teilbar. Die Zahlen bis $(int)\text{Math.sqrt}((double)i)$ sind die Zahlen bis zur Wurzel von i (abgerundet), weil die Systemmethode $\text{Math.sqrt}(\text{double } d)$ die Wurzel einer Doublezahl angenähert berechnet.

```
            }  
            if(prim) System.out.println("Primzahl: "+i);
```

Hat die for-Schleife keinen Teiler gefunden, so ist die Zahl eine Primzahl und prim immer noch true. (prim) ist eine Abkürzung für (prim==true).

```
        }  
    }  
    public static void main(String[] args) {  
        primBis(new Integer(args[0]).intValue());
```

Das erste Kommandozeilenargument wird als Ende der Primzahlenliste (bis) interpretiert.

```
    }  
}
```

```
class Converter {  
    public static String getDez(String bin) {  
        int gesamt=0;  
        int len=bin.length();
```

len ist die Länge des umzuwandelnden Binärstrings aus Nullen und Einsen.

```
        for(int i=0; i<len; i++) {  
            String current=bin.substring(len-i-1, len-i);
```

```

        if(current.equals("1")){
            gesamt=gesamt+hoch(2,i);
        }

```

Der String wird von hinten nach vorne durchgegangen (substring(len-i-1,len-i) ermittelt das len-i-te Element des Strings) und zur Gesamtzahl 2 hoch i addiert, wenn der Teilstring 1 ist.

```

    }
    return ""+gesamt;
}
public static int hoch(int base,int exp){
    if(exp==0)return 1;
    else if(exp==1)return base;
    else return base*hoch(base,exp-1);
}

```

Mit dieser Methode kann die Zahl base hoch exp berechnet werden. Die Definition dieser Methode ist rekursiv. Beispiel: hoch (2,4)=2*hoch(2,3)=2*2*hoch(2,2)=2*2*2*hoch(2,1)=2*2*2*2=16

```

}
public static String getBin(String dez){
    String ret="";
    int dezi=new Integer(dez).intValue();
}

```

Hier wird der Dezimalstring in einen int-Wert umgerechnet via JAVA-System-Klasse.

```

int i=0;
while(hoch(2,i)<=dezi){
    i++;
}
if(i==0)return dez;
int gesamt=0;

```

Die erste Stelle im zu ermittelnden Binärstring, die ungleich 0 ist, ist die Stelle i, so dass 2 hoch (i+1) grösser als der Dezimalwert ist. Wenn i==0 ist, dann muss der Binärwert auch 0 sein.

```

for(int j=i-1;j>=0;j--){

```

Sämtliche Stellen von der i-ten Stelle bis zur 0-ten Stellen können entweder 0 oder 1 sein.

```

    int hochj=hoch(2,j);
    if(gesamt+hochj<=dezi){
        ret=ret+"1";
        gesamt=gesamt+hochj;
    }
    else ret=ret+"0";
}

```

Die j-te Stelle wird auf 0 oder 1 geprüft. Wenn der Gesamtwert plus der Wert der j-ten Stelle (gesamt) grösser als der Dezimalwert ist, ist die Stelle 0 und sonst 1. In gesamt wird jeweils der Gesamtwert von der i-ten bis zur j-ten Stelle gespeichert.

```

}
return ret;
}
public static void main(String[]args){

```

```
System.out.println(getBin(args[0]));
```

Beim Starten des Programms wird das Kommandozeilenargument in eine Binärzahl umgewandelt und auf dem Bildschirm ausgegeben.

```
}  
}
```

Die Zugriffsrechte auf Variablen, Klassen und Methoden können in JAVA eingeschränkt werden mit den Schlüsselworten `private` (nur für die Klasse sichtbar) und `protected` (nur für Unterklassen sichtbar). Ausserdem existiert ein Unterschied zwischen `public`-Methoden und Methoden ohne Zugriffsparameter, die nur aus dem gleichen Paket Zugriffe bearbeiten können. In dieser Einführung ist auf die Zugriffshierarchie kein Wert gelegt worden, der Leser kann sich selbst überlegen, wo eine Zugriffseinschränkung sinnvoll wäre.

Neben `for`-Schleifen gibt es in JAVA auch `while`-Schleifen, die so lange weiterlaufen, bis ein boolescher Ausdruck `false` wird. Beispiel:

```
int a=0;  
int b=234;  
while(a<b){  
    a=a+10;  
}
```

Mit `while`-Schleifen lassen sich `for`-Schleifen simulieren, sie sind allgemeiner als `for`-Schleifen in ihrer ursprünglichen Form, die eine feste Durchlaufzahl haben. JAVA-`for`-Schleifen haben jedoch die Möglichkeit, die Indexvariable zu verändern und damit von der festen Durchlaufzahl abzuweichen. Die Programmiersprache JAVA ist objektorientiert: bisher haben wir nur statische Methoden kennengelernt, deren Funktionalität in ähnlicher Form von jeder höheren Programmiersprache unterstützt wird. Klassen können auch als Objektvorlage fungieren, von der Instanzen gebildet werden, die dann als Datenhalter von Mengen von Objekten gleicher Art gebraucht werden können. Eine Objektklasse besteht aus Attributen und Methoden und einem Konstruktor. Die Attribute können zum Beispiel `int`-, `short`- und `byte`-Werte sein oder auch Verweise auf andere Objekte. Werden Methoden über das Objekt angesprochen, so beziehen sich die Referenzen auf Variablen entweder auf Klassenvariablen, Argumente der Methode oder auf Variablen der aktuellen Instanz der Objektklasse. Der Konstruktor ist ein Instrument, dem Objekt bei der Erschaffung (Instanzierung) diverse Grundwerte mitzugeben, die verarbeitet werden oder direkt Attributen zugewiesen werden. Der Konstruktor wird so benannt wie die Klasse und sieht im Quellcode aus wie eine Methode ohne Returntyp.

```
class List{  
    ListElem start;  
    ListElem end;
```

Diese eine Liste repräsentierende Klasse hat zwei Attribute: `start` und `end` sind Objektverweise auf den Anfang und das Ende der Liste.

```
    public List(){  
    }
```

Der Konstruktor ist leer, weil keine Attribute initialisiert werden.

```
class ListElem{  
    ListElem next;
```

Object content;

ListElem ist eine interne Klasse von List zur Repräsentation eines Elementes der Liste zusammengesetzt aus enthaltenem Objekt und einem Verweis auf das nächste Element.

```
public ListElem(Object content){
    this.content=content;
}
```

Beim Instanzieren des Objekts (Konstruktor) wird dem Objektverweis ein Wert zugewiesen.

```
}
public void insert(Object obj){
    if(start==null){
        start=new ListElem(obj);
        end=start;
    }
```

Wenn noch kein Objekt in der Liste ist, wird mit dem Zeiger start ein neues Objekt erzeugt und der gewünschte Objektverweis weitergegeben. Ende ist gleich Anfang der Liste.

```
}
else{
    end.next=new ListElem(obj);
}
```

Der Objektverweis am Ende der Liste wird benutzt, um ein weiteres Objekt anzuhängen.

```
end=end.next;
```

Das Ende wird auf das neu erschaffene Element verschoben. Mit dem Punkt (end.next) greift man auf ein Attribut einer Instanz einer Objektklasse zu

```
}
}
public boolean contains(Object obj){
    ListElem cursor=start;
```

cursor ist eine Kopie des Zeigers start, deren Modifikation start nicht verändert.

```
while(cursor!=null){
    if(cursor.content==obj)return true;
    cursor=cursor.next;
```

Wenn das übergebene Objekt mit dem im aktuellen Listenelement enthaltenen Objekt übereinstimmt, wird der Wert true zurückgegeben (Rückgabebetyp ist boolean). Ansonsten wird die Liste Element für Element durchlaufen.

```
}
return false;
```

Wenn keine Übereinstimmung gefunden wurde, gibt die Methode false zurück.

```
}
public Object getObjectAt(int i){
    ListElem cursor=start;
```

```

for(int k=0;k<i;k++){
    if(cursor==null)return null;
    else cursor=cursor.next;
}

```

Die Methode geht vom Startelement aus und geht i mal ein Listenelement weiter. Das Objekt des Elements, wo der cursor verharrt, ist das Rückgabeobjekt.

```

return cursor.content;
}
public static void main(String[]args){
    List l=new List();

```

Eine neue Liste wird instanziiert, sie enthält ein leeres Start- und Endelement.

```

Integer s=new Integer("1");
l.insert(new Integer("0"));
l.insert(s);
l.insert(new Integer("2"));

```

Drei Zahlen werden in die Liste eingefügt, ein Zeiger auf s wird zum Test der contains-Methode behalten. Integer ist eine Objekt-Variante vom Typ int. Mit l.insert(Object o) wird auf die Instanz l der Objektklasse List zugegriffen und die innere Struktur der Instanz (nicht der Klasse) verändert.

```

System.out.println("enthalten s: "+l.contains(s));
System.out.println("enthalten 3: "+l.contains(new Integer("3")));
System.out.println("enthalten 2: "+l.contains(new Integer("2")));
System.out.println(""+l.getObjectAt(1));

```

Hier werden die Methoden contains und getObjectAt(int index) getestet, wobei auffällt, dass zwei Objekte nur dann gleich sind, wenn die Zeiger auf das selbe Objekt zeigen (s) und nicht, wenn sie denselben Wert repräsentieren.

```

}
}

```

```

class Matrix {
    double[][] entry;
    Matrix E;
    int n;

```

Eine Matrix ist eine Kollektion von $n*n$ Werten, die nach n Spalten und n Zeilen geordnet ist. Die Zeilen und Spalten werden durch das Array `double[][]entry` repräsentiert.

```

Matrix(int n){
    entry=new double[n][n];
    this.n=n;

```

Der Konstruktor der Klasse Matrix benötigt einen int-Wert n als Argument, um ausreichend Platz für die $n*n$ Werte der Matrix bereitzustellen. Mit `this.n=n` wird die Zahl n für das gesamte Objekt verfügbar, weil

dem Konstruktor übergebene Variablen zunächst lokale Variablen sind und this.n zur Unterscheidung des Attributs n der Klasse von der lokalen Variable benötigt wird.

```
}  
public void createMat(){  
    E=new Matrix(n);  
    for(int i=0;i<n;i++){  
        E.entry[i][i]=1;  
    }  
}
```

Mit der Methode wird der aktuellen Matrix eine passende Einheitsmatrix als Attribut instanziiert und mit sinnvollen Werten initialisiert.

```
}  
public Matrix minus(Matrix a,Matrix b)throws MatrixException {  
    int n=a.n;  
    int n2=b.n;  
    if(n!=n2)throw(new MatrixException());  
    Matrix retm=new Matrix(n);
```

Die Methode minus liefert eine Matrix als Rückgabe und erwartet zwei Matrizen als Argumente, deren Formate in n und n2 gespeichert werden. Sind die Formate verschieden, wird ein Fehler ausgelöst (throw), der von der aufrufenden Methode abgefangen werden muss (try). Methoden, die Fehler auslösen können, müssen in der Bezeichnung nach der Argumentliste den Ausdruck "throws FehlerArtException" integrieren. Mit throw(new Exception e()) löst man einen Fehler aus und mit try{callBeispielMethod();} catch(Exception e){BeispielAuffangCode;} fängt man den Fehler ab.

```
    for(int i=0;i<n;i++){  
        for(int j=0;j<n;j++){  
            retm.entry[i][j]=a.entry[i][j]-b.entry[i][j];
```

Die Werte an der Stelle entry[i][j] der Matrix retm ergeben sich aus der Differenz der Werte an den entsprechenden Stellen der übergebenen Matrizen. Mit matrixname.entry[i][j] greift man auf das Attribut entry der Matrix matrixname zu.

```
    }  
    }  
    return retm;  
}  
public Matrix plus(Matrix a,Matrix b)throws MatrixException {  
    int n=a.n;  
    int n2=b.n;  
    if(n!=n2)throw(new MatrixException());  
    Matrix retm=new Matrix(n);  
    for(int i=0;i<n;i++){  
        for(int j=0;j<n;j++){  
            retm.entry[i][j]=a.entry[i][j]+b.entry[i][j];
```

Analog zur Minusoperation. Wichtig ist die Zugriffsart instanzname.attribut, in diesem Fall auf ein int-Array der Dimension 2 und der Länge n. Man könnte auch die Zugriffe auf die Attribute indirekt durch Methoden durchführen, was aber bei Matrizen aufgrund der intuitiven Vorstellung von Matrizen ungünstig ist. Ansonsten ist der Zugriff über Methoden vorzuziehen: wenn die Methode später geändert wird, müssen nicht alle Programmstellen mit Zugriff auf die Attribute geändert werden.

```

    }
    }
    return retm;
}
public Matrix mal(Matrix a,double s){
    int n=a.n;
    Matrix retm=new Matrix(n);
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            retm.entry[i][j]=a.entry[i][j]*s;
        }
    }
}

```

Eine um den Faktor s gestreckte Matrix wird als neue Instanz zurückgegeben. Als Argument ist nur eine Matrix und ein Faktor des Typs double (64 bit Gleitkomma) übergeben.

```

    }
    }
    return retm;
}
public Matrix mult(Matrix a,Matrix b)throws MatrixException {
    int n=a.n;
    int n2=b.n;
    if(n!=n2)throw(new MatrixException());
}

```

Wenn die Formate nicht gleich sind, wird eine Exception ausgelöst.

```
Matrix retm=new Matrix(n);
```

Die Rückgabematrix hat das gleiche Format wie a und b.

```

for(int i=0;i<n;i++){
    for(int j=0;j<n;j++){
        for(int k=0;k<n;k++){
            retm.entry[i][j]+=(a.entry[i][k])*(b.entry[k][j]);
        }
    }
}

```

Die dritte for-Schleife ergibt sich aus der Definition der Matrizenmultiplikation (Hintergründe siehe diverse Lehrbücher/Skripte diverser Autoren über "Lineare Algebra").

```

    }
    }
    return retm;
}
public void printMatrix(){
}

```

Diese Methode wird als Methode zugehörig zu einer konkreten Instanz von Matrix aufgerufen (in einer anderen Implementation könnten die bisherigen Methoden auch static sein) und kann deshalb auf das zweidimensionale int[][]-Array entry ohne Punkt zugreifen.

```

System.out.println("new Matrix");
for(int i=0;i<n;i++){
    for(int j=0;j<n;j++){
        System.out.print(" "+entry[i][j]+" ");
    }
}

```

Der Matrixeintrag in der i-ten Zeile und j-ten Spalte wird von Leerzeichen eingerahmt auf den Bildschirm ausgegeben.

```
    }  
    System.out.println("");
```

Nach jeder Zeile beginnt eine neue Bildschirmzeile (Unterschied zwischen print und println: println fängt eine neue Zeile an).

```
    }  
    }  
    public double getDet(Matrix D){
```

Diese Methode berechnet die Determinante einer Matrix.

$\det(A) = \sum_{i=1}^n (-1)^{i+1} \text{entry}[0][i-1] \det(A_{1i})$

A_{1i} ist die Matrix, die aus A durch Weglassen der i-ten Spalte und 1-ten Zeile von A entsteht.

```
    double det=0;  
    if(D.entry.length==1){  
        return D.entry[0][0];  
    }
```

Die Determinante einer Matrix mit einem Element ist als das Element selbst definiert.

```
    for(int i=0;i<D.entry.length;i++){  
        int koff=koeff(i+2);  
        det=det+koff*D.entry[0][i]*getDet(D.getSubMatrix(0,i));  
    }
```

Diese for-Schleife addiert die Elemente der Summe aus der Definition und ruft sich selbst mit einer Submatrix als Argument auf. Die Argumente werden zum Teil indexverschoben verwendet.

```
    return det;  
    }  
    static public int koeff(int n){  
        int ret=1;  
        for(int k=0;k<n;k++){  
            ret=ret*(-1);  
        }  
        return ret;  
    }
```

Diese Methode gibt den Wert 1 zurück, wenn n gerade ist und -1 sonst.

```
    }  
    public Matrix getSubMatrix(int k,int l){  
        int n=entry.length;  
        Matrix ret=new Matrix(n-1);
```

Die Submatrix hat einen um eins reduzierten Durchmesser.

```
        for(int i=0;i<k;i++){  
            for(int j=0;j<l;j++){  
                ret.entry[i][j]=entry[i][j];  
            }  
        }
```

```
}
```

In die Rückgabematrix werden die Elemente `entry[i][j]` aufgenommen, für die $i < k$, $j < l$ gilt.

```
for(int i=k+1;i<n;i++){
    for(int j=0;j<l;j++){
        ret.entry[i-1][j]=entry[i][j];
    }
}
```

In die Rückgabematrix werden die Elemente `entry[i][j]` aufgenommen, für die $i > k$, $j < l$ gilt. Die i -Werte sind in der Rückgabematrix um 1 reduziert.

```
for(int i=0;i<k;i++){
    for(int j=l+1;j<n;j++){
        ret.entry[i][j-1]=entry[i][j];
    }
}
```

In die Rückgabematrix werden die Elemente `entry[i][j]` aufgenommen, für die $i < k$, $j > l$ gilt. Die j -Werte sind in der Rückgabematrix um 1 reduziert.

```
for(int i=k+1;i<n;i++){
    for(int j=l+1;j<n;j++){
        ret.entry[i-1][j-1]=entry[i][j];
    }
}
```

In die Rückgabematrix werden die Elemente `entry[i][j]` aufgenommen, für die $i > k$, $j > l$ gilt. Die i -Werte und j -Werte sind in der Rückgabematrix um 1 reduziert.

```
    return ret;
}
}
class MatrixException extends Exception{
}
```

Mit dem Wort `extends` wird JAVA mitgeteilt, dass alle Eigenschaften der JAVA-System-Klasse `Exception` übernommen werden (Vererbung). Die `MatrixException` kann also als spezieller Fehler ausgelöst werden. Geerbte Methoden können von einer Subklasse überschrieben werden.

```
class Chatter{
    String name;
    boolean explored=false;
    Chatter[] kennt;
    int klen;
```

Die Klasse `Chatter` repräsentiert einen Nutzer eines Instant-Messaging Systems als Objekt. Zur Identifikation hat er einen Namen und als Information ist eine Verweis-Liste auf andere `Chatter` integriert. Der boolesche Wert `explored` ist zu internen Zwecken notwendig. Die Variable `klen` zeigt die Anzahl der Chatpartner an.

```

public Chatter(String name){
    this.name=name;
    kennt=new Chatter[0];

```

Wird ein neuer Chatter der Graphenstruktur der Chatter und deren Beziehungen hinzugefügt, so wird bei der Konstruktion des Objekts der Name des neuen Chatters übergeben. kennt wird als neues Array der Länge 0 instanziiert.

```

}
public boolean kenntIndirekt(String name2){
    boolean ret=false;
    explored=true;

```

Der aktuelle Chatter wird bei der Suche nach name2 in der zusammenhängenden Community als bereits durchsucht gekennzeichnet (explored).

```

    if(name2.equals(name))ret=true;

```

Ist der aktuelle Chatter der gesuchte Chatter, so kann ein positives Suchergebnis gemeldet werden.

```

    for(int i=0;i<klen;i++){
        if(!kennt[i].explored&&kennt[i].kenntIndirekt(name2)){
            ret=true;
        }

```

In der Schleife werden alle Verweise auf Bekannte aufgerufen, nach name2 zu suchen und ihr Suchergebnis mitzuteilen. Die bereits explorierten Nutzer werden nicht berücksichtigt.

```

    }
    return ret;

```

Wenn der aktuelle Name nicht mit name2 übereinstimmt und die Suche bei allen Verweisen nicht erfolgreich war, ist ret immer noch false und wird returned.

```

}
public void printCommunity(){
    System.out.println("Person: "+name);

```

Hier wird der Name der aktuellen Chatter-Instanz ausgegeben.

```

    explored=true;
    for(int i=0;i<klen;i++){
        if(!kennt[i].explored)kennt[i].printCommunity();
    }

```

Alle Verweise werden nach weiteren Namen verfolgt, wenn sie noch nicht durchsucht wurden.

```

}
public void resetExplored(){
    explored=false;
    for(int i=0;i<klen;i++){
        if(kennt[i].explored)kennt[i].resetExplored();
    }

```

Mit dieser kleinen Methode werden sämtliche explored-Werte eines zusammenhängenden Gebiets auf false zurückgesetzt. Dies ist für die nächste Suche unbedingte Voraussetzung.

```
}
public void addChatter(Chatter c){
    klen++;
    Chatter[] c2=new Chatter[klen];
    for(int i=0;i<klen-1;i++){
        c2[i]=kennt[i];
    }
    c2[klen-1]=c;
    kennt=c2;
}
```

Dies ist eine Standardmethode, mit der Arrays verlängert werden. Hier wird ein neuer Chatter hinzugefügt. Die Länge des Arrays wird um 1 erhöht und ein zweites Chatter-Array-Objekt c2 dieser Länge erzeugt. Der Inhalt des alten Objekts wird mit einer for-Schleife in das neue kopiert und der letzte Index auf ein Argument gesetzt (Chatter c). Schliesslich wird der alte Zeiger gelöscht und auf die neue Liste gerichtet. Man beachte, dass Arrays der Länge n Indizes von 0 bis n-1 haben, auf die zugegriffen wird und höhere Indizes zu Fehlern führen.

```
}
}
```

```
class Person {
    Person mutter;
    Person vater;
    Person[] toechter;
    Person[] soehne;
    int t;
    int s;
    String name;
}
```

Diese Datenhaltungsklasse repräsentiert Personen. Neben dem Namen und Verweisen auf Mutter und Vater finden t Toechter und s Soehne als Verweise in einer Instanz Platz.

```
public Person(String name){
    this.name=name;
    toechter=new Person[0];
    soehne=new Person[0];
}
```

Bei der Konstruktion des Objekts muss bereits ein Name mitgegeben werden, weitere Verweise werden durch spezielle Methoden angefügt.

```
}
public void addMutter(Person p){
    mutter=p;
}
public void addVater(Person p){
    vater=p;
}
```

Weil es nur einen Vater und eine Mutter geben kann, sind diese beiden Methoden sehr einfach und verursachen nur das Setzen eines Verweises.

```
public void addTochter(Person p){
    t++;
    Person[] t2=new Person[t];
    for(int i=0;i<t-1;i++){
        t2[i]=toechter[i];
    }
    t2[t-1]=p;
    toechter=t2;
}
public void addSohn(Person p){
    s++;
    Person[] s2=new Person[s];
    for(int i=0;i<s-1;i++){
        s2[i]=soehne[i];
    }
    s2[s-1]=p;
    soehne=s2;
}
```

Diese beiden Methoden funktionieren haargenau so wie die addChatter(Chatter c) in der Klasse Chatter nur die Variablen sind anders benannt.

```
    }
    public void druckeVorfahren(){
        System.out.println("Person: "+name);
        if(mutter!=null){
            System.out.println("mutter: "+mutter.name);
        }
        if(vater!=null){
            System.out.println("vater: "+vater.name);
        }
    }
}
```

Der eigene Name und die Namen der Eltern werden auf den Bildschirm gedruckt. Die Abfrage (mutter!=null) ist wichtig, denn im Fall mutter==null (es liegen keine Informationen vor) träte sonst eine Nullpointerexception auf. Nicht initialisierte Objekte haben den Wert null und jeder Zugriff auf Attribute des Nullzeigers führt zu Fehlern.

```
        if(mutter!=null){
            System.out.println("Vorfahren mütterlicherseits");
            mutter.druckeVorfahren();
        }
        if(vater!=null){
            System.out.println("Vorfahren väterlicherseits");
            vater.druckeVorfahren();
        }
    }
}
```

Die Methode ruft sich selbst auf, aber mit anderen Objekten als Kontext (mutter, vater).

```
    }
    public void druckeNachkommen(){
        System.out.println("Person: "+name);
    }
}
```

```

System.out.println("Töchter: ");
for(int i=0;i<t;i++){
    System.out.println(toechter[i].name);
}
System.out.println("Söhne: ");
for(int i=0;i<s;i++){
    System.out.println(soehne[i].name);
}

```

Die Töchter und Söhne werden in zwei for-Schleifen durchlaufen und deren Attribut name ausgegeben.

```

for(int i=0;i<t;i++){
    toechter[i].druckeNachkommen();
}
for(int i=0;i<s;i++){
    soehne[i].druckeNachkommen();
}

```

Mit dem jeweiligen Kind als Kontext wird die Methode neu rekursiv aufgerufen.

```

}
public static void main(String[] args){
    Person p=new Person("person");
    p.addVater(new Person("vater"));
    p.addMutter(new Person("mutter"));
    p.mutter.addVater(new Person("Grossvater m"));
    p.mutter.addMutter(new Person("Grossmutter m"));
    p.addVater(new Person("vater"));
    p.vater.addVater(new Person("Grossvater v"));
    p.vater.addMutter(new Person("Grossmutter v"));
    p.addTochter(new Person("Tochter1"));
    p.addTochter(new Person("Tochter2"));
    p.addSohn(new Person("Sohn"));
    p.toechter[1].addTochter(new Person("Enkelin"));
    p.druckeVorfahren();
    p.druckeNachkommen();
}

```

Es wird ein partieller Verwandtschaftsbaum von Grossmutter über Person bis zur Enkelin erstellt und dann werden die Vorfahren und Nachfahren der Person ausgedruckt. Man geht von der Person aus und erzeugt mit Hilfe der Attributzeiger neue Objekte. Für einen korrekten Baum müssten für jede Beziehung zwei Verweise eingetragen werden, deshalb ist dieser Baum nur partiell.

```

}
}

```

```

class Autobahn{
    String name;
    String anfang;
    String ende;
    int laenge;
}

```

Diese Klasse modelliert eine Autobahn, wobei die ersten vier Attribute allgemeiner Art sind.

```
String[]kname;
Autobahn[]kreuz;
int[]entf;
int kanzahl;
```

Die Attribute kname, kreuz und entf sind Listen der Länge kanzahl und stellen die Menge der Autobahnkreuze dar. Wenn $i < \text{kanzahl}$, dann ist kname[i] der Name des Autobahnkreuzes und entf[i] die Position des Kreuzes in Kilometern relativ zum Anfang der Autobahn und kreuz[i] ein Verweis auf ein Objekt vom Typ Autobahn, welches die andere Autobahn des Kreuzes symbolisiert.

```
String[]abfahrt;
int[]abfahrtz;
int azahl;
boolean explored=false;
```

Die Arrays abfahrt und abfahrtz haben die Länge azahl, wobei Abfahrten und nicht Autobahnkreuze berücksichtigt werden und abfahrt die Namen und abfahrtz die relativen Positionen enthält. Wie in der Klasse Chatter ist explored eine Hilfsvariable zur Exploration der Datenstruktur.

```
public Autobahn(String name,String anfang,String ende,int laenge){
    this.name=name;
    this.anfang=anfang;
    this.ende=ende;
    this.laenge=laenge;
    kname=new String[0];
    kreuz=new Autobahn[0];
    entf=new int[0];
    abfahrt=new String[0];
    abfahrtz=new int[0];
```

Bei der Erschaffung eines neuen Autobahn-Objekts müssen die vier Grundcharakteristiken den Attributen des Objekts (this.) zugewiesen und aus den lokalen Variablen gelesen werden. Sämtliche Arrays haben zu Anfang die Länge 0.

```
    }
    public void addAbfahrt(String name,int len){
        azahl++;
        String[]abfahrt2=new String[azahl];
        int[]abfahrtz2=new int[azahl];
        for(int i=0;i<azahl-1;i++){
            abfahrt2[i]=abfahrt[i];
            abfahrtz2[i]=abfahrtz[i];
        }
        abfahrt2[azahl-1]=name;
        abfahrtz2[azahl-1]=len;
        abfahrt=abfahrt2;
        abfahrtz=abfahrtz2;
```

Diese Methode ist wiederum eine Standardmethode vergleichbar mit addChatter(Chatter c) wobei zwei Arrays (abfahrt, abfahrtz) verlängert werden.

```
    }
    public void addKreuz(String name,Autobahn a,int entf3){
```

```

kanzahl++;
String[]kname2=new String[kanzahl];
Autobahn[]kreuz2=new Autobahn[kanzahl];
int[]entf2=new int[kanzahl];
for(int i=0;i<kanzahl-1;i++){
    kname2[i]=kname[i];
    kreuz2[i]=kreuz[i];
    entf2[i]=entf[i];
}
kname2[kanzahl-1]=name;
kreuz2[kanzahl-1]=a;
entf2[kanzahl-1]=entf3;
kname=kname2;
kreuz=kreuz2;
entf=entf2;

```

Standardmethode: kname, kreuz und entf werden um name, a und entf verlängert.

```

}
public int abs(int a,int b){
    if(a>b)return a-b;
    else return b-a;
}

```

Diese Methode liefert den Betrag der Differenz zwischen den int-Werten a und b zurück.

```

}
public void resetExplored(){
    if(!explored)return;
    else{
        explored=false;
        for(int i=0;i<kanzahl;i++){
            kreuz[i].resetExplored();
        }
    }
}

```

Jedes Autobahnobjekt, das exploriert wurde, wird rekursiv wieder als nicht exploriert gekennzeichnet.

```

}
public int getEntf(String a,String b,int kilo){

```

Diese Methode erwartet einen Ausgangspunkt a (Abfahrt oder Kreuz, Name) auf der Autobahn der aktuellen Instanz und eine Zielabfahrt (b) sowie den int-Wert kilo (Länge der explorierten Strecke) und liefert die kürzeste Entfernung in Kilometern von a und b zurück. Die Rückgabe -1 bedeutet, dass diese Verzweigung nicht zum Ziel führt oder bereits exploriert wurde.

```

    if(explored)return -1;
    explored=true;
    int currentpos=0;
    for(int i=0;i<kanzahl;i++){
        if(kname[i].equals(a))currentpos=entf[i];
    }

```

Wenn a ein Autobahnkreuz ist, wird in der lokalen Variable currentpos die relative Entfernung zum Autobahnanfang dieses Kreuzes gespeichert.

```

for(int i=0;i<azahl;i++){
    if(abfahrt[i].equals(a))currentpos=abfahrztz[i];
}

```

Falls a eine Abfahrt ist, wird in currentpos die relative Position gesichert.

```

int targetpos=-1;
for(int i=0;i<azahl;i++){
    if(abfahrt[i].equals(b))targetpos=abfahrztz[i];
}

```

Falls das Ziel als Abfahrt auf der Autobahn ist, wird dessen relative Position festgehalten (targetpos).

```

int[] k=new int[kanzahl];
for(int i=0;i<kanzahl;i++){
    k[i]=kreuz[i].getEntf(kname[i],b,abs(currentpos,entf[i]));
}

```

Die Methode getEntf wird bezüglich jeder anderen kreuzenden Autobahn aufgerufen, um einen Weg zum Ziel b zu finden. Der neue Anfang wird als das jeweilige Kreuz festgelegt und als zusätzliche Kilometerentfernung die Entfernung von currentpos bis zum Kreuz kname[i] mit relativer Position entf[i] übergeben.

```

boolean directshorter=true;
if(targetpos>-1){
    int direntf=abs(currentpos,targetpos);
    for(int i=0;i<kanzahl;i++){
        if(k[i]!=-1&&k[i]<direntf)directshorter=false;
    }
    if(directshorter)return direntf+kilo;
}

```

Wenn das Ziel auf der aktuellen Autobahn ist (targetpos>-1), wird die Entfernung zum Ziel mit allen anderen Entfernungen in k verglichen, die über andere Autobahnen führen, und die Entfernung zurückgegeben, wenn sie kürzer ist. Es tritt das Problem auf, dass eine Strecke über eine andere Autobahn, die wieder auf die erste Autobahn führt, kürzer sein kann als der direkte Weg über die erste Autobahn. Dieses Problem kann in dieser Version nur dann umgegangen werden, wenn man diese Problemfälle im Objektgraphen erkennt und dementsprechend diverse Autobahnen mit denselben Attributen doppelt instanziiert.

```

}
int min=-1;
for(int i=0;i<kanzahl;i++){
    if(k[i]!=-1&&min==-1)min=k[i];
    if(k[i]!=-1&&k[i]<min)min=k[i];
}

```

Wenn die direkte Entfernung kürzer ist oder die Abfahrt nicht auf der Autobahn liegt, wird das Minimum der anderen Strecken zur Abfahrt gewählt. Der Wert -1 in k bedeutet, dass dieser Weg nicht zum Ziel führt.

```

}
if(min==-1)return -1;

```

(kein Weg führte zum Ziel)

```
return min+kilo;
```

Die Variable kilo wurde übergeben und ist der Offset von der vorigen Autobahn.

```
}
```

Eine idealere Version würde sich von Abfahrt zu Abfahrt vorkämpfen und immer nur die nächstnächste Abfahrt explorieren.

```
public static void main(String[] args) {
    Autobahn ab1=new Autobahn("A1", "", "", 200);
    Autobahn ab2=new Autobahn("A2", "", "", 200);
    ab1.addKreuz("K1", ab2, 75);
    ab2.addKreuz("K1", ab1, 50);
    ab1.addAbfahrt("Abfahrt 1", 10);
    ab2.addAbfahrt("Abfahrt 2", 15);
    ab1.addAbfahrt("Abfahrt 3", 180);
    ab2.addAbfahrt("Abfahrt 4", 190);
}
```

In diesem Beispiel sind zwei Autobahnen A1 und A2 mit dem Kreuz K1 und vier Abfahrten. Das Kreuz muss mit dem selben Namen beiden Autobahnen zugefügt werden.

```
System.out.println("Abfahrt 1-Abfahrt 2: "+ab1.getEntf("Abfahrt 1","Abfahrt 2",0));
ab1.resetExplored();
System.out.println("Abfahrt 1-Abfahrt 3: "+ab1.getEntf("Abfahrt 1","Abfahrt 3",0));
ab1.resetExplored();
System.out.println("Abfahrt 2-Abfahrt 4: "+ab2.getEntf("Abfahrt 2","Abfahrt 4",0));
ab2.resetExplored();
```

Hier werden drei mögliche Abstände berechnet, man beachte, dass resetExplored unbedingt notwendig ist.

```
}
}
```

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

Zur Verwendung der Klasse Color muss das awt mit JAVA-GUI-Klassen importiert werden.

```
interface GObject {
    public void shift(double x, double y);
    public void setColor(Color col);
    public Point getCenter();
    public void setCenterTo(double x, double y);
}
```

Ein Interface ist eine Auflistung von Methoden mit deren Rückgabewerten, die von anderen Klassen implementiert werden können. Über die übergeordnete Datenstruktur des Interface können dann verschiedene implementierende Objekte über eine gemeinsame Schnittstelle angesprochen werden.

```
class Point implements Gobject{
```

Das Schlüsselwort implements bedeutet, dass sämtliche Methoden in der Liste Interface Gobject implementiert werden müssen.

```
    double x;  
    double y;  
    int xi;  
    int yi;  
    Color col;
```

Ein Punkt hat die Attribute x, y und die Farbe col. Die int-Werte xi und yi sind gerundete Werte von x und y zur Ausgabe auf den Bildschirm.

```
    public Point(double x,double y){  
        this.x=x;  
        this.y=y;  
        xi=(int)x;  
        yi=(int)y;  
        col=Color.black;
```

Der Kontruktor verlangt nur die zwei Koordinaten als double.

```
    }  
    public void shift(double x,double y){  
        this.x+=x;  
        this.y+=y;  
        xi=(int)this.x;  
        yi=(int)this.y;
```

In dieser Methode wird eine Koordinatentransformation durchgeführt.

```
    }  
    public void setColor(Color col){  
        this.col=col;
```

Die Farbe des Punktes wird gesetzt.

```
    }  
    public Point getCenter(){  
        Point ret=new Point(x,y);  
        ret.setColor(col);  
        return ret;
```

Das Zentrum der Figur eines Punktes ist der Punkt selbst.

```
    }  
    public void setCenterTo(double x,double y){  
        Point p=getCenter();  
        shift(-p.x,-p.y);  
        shift(x,y);
```

Dem Punkt werden die Koordinaten x und y zugewiesen.

```

    }
}
class Line implements Gobject{
    Point a;
    Point b;

```

Im Fall der Linie werden die Methodenaufrufe auf die beiden Teilpunkte weitergegeben.

```

    public Line(Point a,Point b){
        this.a=a;
        this.b=b;
    }
    public void shift(double x,double y){
        a.shift(x,y);
        b.shift(x,y);
    }
    public void setColor(Color col){
        a.setColor(col);
        b.setColor(col);
    }
    public Point getCenter(){
        return new Point((a.x+b.x)/2,(a.y+b.y)/2);
    }

```

Das Zentrum der Linie ist der Mittelwert der Koordinaten.

```

    }
    public void setCenterTo(double x,double y){
        Point p=getCenter();
        shift(-p.x,-p.y);
        shift(x,y);
    }
}
class Triangle implements Gobject{

```

Diese Klasse ist analog zur Linie, jedoch müssen drei Punkte statt zwei Punkte verwaltet werden.

```

    Point a;
    Point b;
    Point c;
    public Triangle(Point a,Point b,Point c){
        this.a=a;
        this.b=b;
        this.c=c;
    }
    public void shift(double x,double y){
        a.shift(x,y);
        b.shift(x,y);
        c.shift(x,y);
    }
    public void setColor(Color col){
        a.setColor(col);
        b.setColor(col);
        c.setColor(col);
    }
}

```

```

public Point getCenter(){
    return new Point((a.x+b.x+c.x)/3,(a.y+b.y+c.y)/3);
}
public void setCenterTo(double x,double y){
    Point p=getCenter();
    shift(-p.x,-p.y);
    shift(x,y);
}
}
class Circle implements GObject{
    double x;
    double y;
    int radius;
    int xi;
    int yi;
    Color col;
}

```

Ein Kreis verhält sich wie ein Punkt, hat jedoch das zusätzliche Attribut radius.

```

public Circle(double x,double y,int radius){
    this.x=x;
    this.y=y;
    xi=(int)x;
    yi=(int)y;
    this.radius=radius;
    col=Color.black;
}
public void shift(double x,double y){
    this.x+=x;
    this.y+=y;
    xi=(int)this.x;
    yi=(int)this.y;
}
public void setColor(Color col){
    this.col=col;
}
public Point getCenter(){
    Point ret=new Point(x,y);
    ret.setColor(col);
    return ret;
}
public void setCenterTo(double x,double y){
    Point p=getCenter();
    shift(-p.x,-p.y);
    shift(x,y);
}
}
class Rectangle implements GObject{
    double x;
    double y;
    int w;
    int h;
    int xi;
    int yi;
}

```

```
Color col;
```

...wie Point nur mit Breite w und Höhe h.

```
public Rectangle(double x,double y,int w,int h){
    this.x=x;
    this.y=y;
    this.w=w;
    this.h=h;
    xi=(int)x;
    yi=(int)y;
    col=Color.black;
}
public void shift(double x,double y){
    this.x+=x;
    this.y+=y;
    xi=(int)this.x;
    yi=(int)this.y;
}
public void setColor(Color col){
    this.col=col;
}
public Point getCenter(){
    Point ret=new Point(x+w/2,y+h/2);
    ret.setColor(col);
    return ret;
}
```

Bei der Berechnung des Zentrums des Rechtecks müssen Breite und Höhe berücksichtigt werden.

```
    }
    public void setCenterTo(double x,double y){
        Point p=getCenter();
        shift(-p.x,-p.y);
        shift(x,y);
    }
}
class Manager implements Gobject{
    Gobject[]g;
```

Der Figur-Manager verwaltet eine Menge von Figuren unterschiedlicher Arten.

```
public Manager(Point[]p,Line[]l,Triangle[]t,Circle[]c,Rectangle[]r){
    int n1=p.length;
    int n2=n1+l.length;
    int n3=n2+t.length;
    int n4=n3+c.length;
    int n5=n4+r.length;
```

In der neuen Liste sind bestimmte Bereiche für die einzelnen Arraytypen reserviert.

```
g=new Gobject[n5];
```

Der int-Wert n5 ist die Summe aller Arraylängen. So lang ist g.

```

for(int i=0;i<n1;i++){
    g[i]=(Gobject)p[i];
}

```

Durch die Castinganweisung (Gobject), wobei Gobject ein Interface ist, werden die Punkte von ihrer ursprünglichen Objektart abstrahiert und als Interface betrachtet. Sämtliche Methoden in der Liste von "Interface Gobject" sind auf den Interface-Objekttyp "(Gobject)p[i]" anwendbar.

```

}
for(int i=n1;i<n2;i++){
    g[i]=(Gobject)l[i-n1];
}
for(int i=n2;i<n3;i++){
    g[i]=(Gobject)t[i-n2];
}
for(int i=n3;i<n4;i++){
    g[i]=(Gobject)c[i-n3];
}
for(int i=n4;i<n5;i++){
    g[i]=(Gobject)r[i-n4];
}
}

```

Der Konstruktor erwartet Listen von Figurtypen und schreibt sie in das gemeinsame Array g.

```

}
public void shift(double x,double y){
    for(int i=0;i<g.length;i++){
        g[i].shift(x,y);
    }
}

```

Hier wird die Methode shift auf alle Elemente g[i] (i=0,...,i=n5-1) des Managers angewandt.

```

}
public void setColor(Color col){
    for(int i=0;i<g.length;i++){
        g[i].setColor(col);
    }
}

```

Auch die Methode setColor(Color col) kann auf das gesamte Array angewandt werden. Spezielle Attribute wie der Radius eines Kreises oder Breite und Höhe eines Rechtecks sind für den Objekttyp nicht sichtbar, weil sie nicht für alle Objekte der Liste existieren. Das Interface erzwingt, dass bestimmte Methoden existieren.

```

}
public Point getCenter(){
    double x=0;
    double y=0;
    for(int i=0;i<g.length;i++){
        Point cen=g[i].getCenter();
    }
}

```

Für beliebige Gobjects h liefert h.getCenter() das Zentrum des Objekts h vom Typ Point zurück.

```

x+=cen.x;
y+=cen.y;
}

```

Die Koordinaten der Zentren aller Objekte werden addiert

```
x/=g.length;  
y/=g.length;
```

und der Mittelwert gebildet

```
return new Point(x,y);
```

und als einzelner Punkt zurückgegeben.

```
    }  
    public void setCenterTo(double x,double y){  
        Point p=getCenter();  
        shift(-p.x,-p.y);  
        shift(x,y);  
    }  
}
```

```
class Rakete{  
    String bezeichnung;  
    double hoehe;  
    double schubkraft;  
    double fehlstartquote;  
    String startort;  
    double preisprokilo;
```

Dieses Objekt repräsentiert eine Rakete mit diversen Eigenschaften. Anhand dieses Beispiels soll die Vererbung in JAVA erklärt werden.

```
    }  
class ESARakete extends Rakete{  
    double esafehlstarts;  
    public void contactESA(){};
```

Dem JAVA-Compiler wird mit Hilfe des Schlüsselworts extends mitgeteilt, dass alle Attribute und Methoden der Klasse Rakete auch für die Klasse ESARakete verfügbar sind. Optional können Methoden überschrieben werden, sie müssen dann den gleichen Namen, die gleichen Argumente und den gleichen Returntype haben. Im Fall der ESARakete wurden das Attribut esafehlstarts und die Methode contactESA() hinzugefügt. Diese sind für Objekte der Klasse Rakete nicht sichtbar. Mit Castings (Beispiel: "ESARakete esar=new ESARakete();Rakete r=(Rakete)esar;") kann man Objekte in Mutterobjekte konvertieren, wobei die zusätzlichen Methoden im Subobjekt wegfallen.

```
    }  
class NASARakete extends Rakete{  
    double nasafehlstarts;  
    public void contactNASA(){};
```

Es ist sinnvoll, in dieser spezialisierten Klasse Methoden einzubauen, die für die Oberklasse unsinnvoll sind. Gründe für Vererbungsstrukturen sind die Wiederverwendung von Code und die bessere logische Gliederung.

```
}  
class Ariane5 extends ESARakete{  
    public void flugBuchen(){};
```

Die Klasse Ariane5 ist eine Subklasse von ESARakete. Sie erbt die Attribute von Rakete und ESARakete (also von allen Vorfahren-Objekten) und die Methode contactESA(). Die Methode flugBuchen() ist nur für die Spezialisierung Ariane5 sinnvoll. Natürlich sind unbemannte Raketen-Flugbuchungen für Satteliten viel zu teuer, aber man darf ja träumen...

```
}  
class Ariane5b extends ESARakete{  
    public void flugBuchen(){};
```

Eine Klasse kann nur eine Mutterklasse haben, die wiederum eine Mutterklasse haben kann, aber eine Klasse kann beliebig viele Tochterklassen haben. Ausserdem dürfen beliebig viele Interfaces durch Komma getrennt implementiert werden (Beispiel: "class Hallo2 extends Hallo1 implements Interface1,Interface2,Interface3").

```
}  
class Atlas extends NASARakete{  
    public void flugBuchen(){};
```

```
}  
class Titan extends NASARakete{  
    public void flugBuchen(){};
```

```
}
```

```
class ThreadDemo{  
    public static void main(String[]args){  
        for(int i=0;i<8;i++){  
            new SpecialThread("T"+i).start();  
        }  
    }  
}
```

Wenn die Klasse ThreadDemo kompiliert und ausgeführt wird, wird zunächst diese main-Methode aufgerufen, die acht Threads erzeugt und startet, so dass deren run-Methoden gestartet werden.

```
}  
}  
class SpecialThread extends Thread{
```

Thread ist eine JAVA-Systemklasse, die die Eigenschaft vererbt, dass die Tochterklasse ein eigener Prozess ist, der scheinbar parallel zu anderen Prozessen abläuft. Die erbende Klasse muss die Methode public void run() implementieren, die mit .start() von anderen Prozessen aus gestartet wird.

```
String name;  
public SpecialThread(String name){  
    this.name=name;
```

```

}
public void run(){
    System.out.println("Thread "+name+" started");
    for(int i=0;i<20;i++){
        System.out.println("Thread "+name+" print: "+i);

```

Jeder SpecialThread schreibt nach dem Start zwanzig Mal eine Zahl auf den Bildschirm

```

    try{Thread.sleep((int)(Math.random()*2000));}catch(Exception e){};

```

und wartet dazwischen eine zufällige Zeit. Mit Math.random() berechnet man mit Hilfe der Math-Systemklasse eine double-Zahl zwischen 0 und 1. Nach der Multiplikation mit 2000 und der Umwandlung in einen int-Value erhält man eine int-Zahl zwischen 0 und 2000. Die Methode Thread.sleep(int i) hält den aktuellen Thread für i Millisekunden an. Dabei kann ein Fehler ausgelöst werden. Deshalb wird mit try{Code;}catch(Exception e){Auffangcode;}; der Fehler abgefangen, der Auffangcode wird hier einfach ignoriert.

```

    }
}
}

```

```

import java.awt.*;

```

Die Systemklassen des AWT (Abstract Window Toolkit), einer graphischen Userinterface-Bibliothek, werden importiert.

```

public class Function extends Frame{

```

Die Klasse Function soll eine Funktion in einem Fenster anzeigen, also wird die Klasse Frame (Fenster) als Oberklasse gewählt und deren Eigenschaften übernommen.

```

    public Function(){
        super("Function");

```

Mit super(String s) ruft man den Konstruktor der Oberklasse auf, in diesem Fall wird dem Fenster der Titel "Function" zugewiesen.

```

        resize(400,400);

```

Die Größe des Fensters soll 400*400 sein. Man beachte, dass resize eine Methode der Oberklasse ist, die einfach so benutzt werden kann, weil Function Unterklasse von Frame ist.

```

        show();

```

Mit diesem Methodenaufruf wird das Fenster auf dem Bildschirm mit Eintrag in die Taskleiste angezeigt.

```

    }
    public void paint(Graphics g){
        g.setColor(Color.black);
        g.drawLine(0,200,400,200);

```

```
g.drawLine(200,0,200,440);
```

"public void paint(Graphics g)" wird nach dem Öffnen des Fensters automatisch gestartet. g repräsentiert den Grafikkontext des Fensters und mit g.setColor(Color col) kann die aktuelle Malfarbe gewählt werden, wobei für die häufigsten Farben Konstanten in der Klasse Color reserviert sind (Color.black, Color.white, Color.blue, Color.red, Color.yellow, ...). Die Methode g.drawLine(int x,int y,int x2,int y2) malt eine Linie von (x,y) nach (x2,y2).

```
g.setColor(Color.blue);
int oldx=-200;
int oldy=(int)(75*f((double)-200/75));
for(int i=0;i<100;i++){
    int x=i*4-200;
    int y=(int)(75*f((double)x/75));
    g.drawLine(200+oldx,200-oldy,200+x,200-y);
}
```

Es werden zwei Punkte des Graphen der Funktion f berechnet, die nicht weit voneinander entfernt sind, und verbunden. Insgesamt werden 100 Punkte behandelt. Die x-Werte liegen in gleichmässigen Abständen zwischen -8/3 und +8/3. Dazu werden Werte zwischen 0 und 99 geeignet verschoben. Der y-Wert zu einem Wert x wird mit der Methode "double f(double x)" berechnet und mit 75 multipliziert, damit x-Achse und y-Achse gleich skaliert sind. Das Ergebnis ist die Annäherung eines Funktionsgraphen.

```
    oldx=x;
    oldy=y;
}
}
public double f(double x){
    return x*x*x*x-x*x*x;
}
```

Hier wird x hoch vier plus x hoch zwei berechnet, diese Funktion kann natürlich vor dem Compilieren variiert werden. Der Graphenausschnitt ist jedoch nicht in einer Zeile zu ändern.

```
    }
    public static void main(String[]args){
        new Function();
    }
```

Es wird ein neues Objekt vom Typ Function erzeugt, der Konstruktor "public Function()" wird gerufen und dieser zeigt das Fenster mit show() an.

```
    }
    public boolean handleEvent(Event evt){
        if(evt.id==Event.WINDOW_DESTROY)System.exit(0);
        return false;
    }
```

Die Methode "public boolean handleEvent(Event evt)" kann von Fenster-Subklassen überschrieben werden, sie wird bei Aktionen (Events) des User Interface aufgerufen und übergibt das Ereignis in der Variablen evt. Im Attribut evt.id von evt kann man die Event-Art feststellen (weitere Attribute sind evt.target [auslösendes Objekt], evt.arg, evt.x, evt.y, evt.key). Mittlerweile gibt es ein Listener-Eventmodell in JAVA, aber zur Zeit der Formulierung dieses Textes war das alte Modell immer noch vom AWT unterstützt und hat den Vorteil, dass es für kleine Projekte leichter zu programmieren ist.

```
    }
}
```

```
import java.awt.*;
public class Ellipse extends Frame{
```

Dieses Programm ist sehr ähnlich zum Programm Function und dient dazu, eine Ellipse gegebener Gleichung graphisch darzustellen.

```
    public Ellipse(){
        super("Function");
        resize(400,400);
        show();
    }
    public void paint(Graphics g){
        g.setColor(Color.black);
        g.drawLine(0,200,400,200);
        g.drawLine(200,0,200,440);
        g.setColor(Color.blue);
        for(int x=0;x<100;x++){
            for(int y=0;y<100;y++){
                double xd=((double)(x-50))/10;
                double yd=((double)(y-50))/10;
```

Die for-Schleife durchläuft alle Paare von x und y, wobei x und y zwischen 0 und 99 sind. Transformiert werden sie in double xd und double yd als Punktnetz im Intervall [-5,5]X[-5,5].

```
                if(isElem(xd,yd)){
```

Mit isElem wird geprüft, ob der Punkt (xd,yd) der Ellipsengleichung genügt. Weil die genauen Elemente des Ellipsenrands nur per Zufall getroffen werden, wird ein Toleranzintervall der Länge 0.4 eingebaut, so dass der Ellipsenrand breiter erscheint und deshalb nicht verschwindet.

```
                    int xdi=(x-50)*4;
                    int ydi=(y-50)*4;
                    g.fillRect(xdi+200,ydi+200,1,1);
```

Mit g.fillRect(int x,int y,int w,int h) wird ein gefülltes Rechteck in das Fenster gemalt an der Stelle (x,y) mit Breite w und Höhe h. Durch die Zuweisungen zu xdi und ydi wird das Intervall in die Mitte des Koordinatensystems verschoben und über das gesamte Fenster gestreckt. Das Malen geschieht dann im Zentrum des 400X400 Fensters. Die Koordinaten innerhalb eines Fensters haben ihren Ursprung in der linken oberen Ecke.

```
                }
            }
        }
    }
    public boolean isElem(double x,double y){
        if((x*x+y*y+0.5*x*y-x+2*y-1)<0.2&&(x*x+y*y+0.5*x*y-x+2*y-1>-0.2))return true;
```

Diese Ellipsengleichung in Abhängigkeit von x und y wird auf "fast-Nullstellen" geprüft.

```

    return false;
}
public static void main(String[] args){
    new Ellipse();

```

Ein neues Objekt vom Typ Ellipse (Oberklasse Frame) wird erzeugt und am Ende des Konstruktors stellt das Fenster sich selbst dar.

```

}
public boolean handleEvent(Event evt){
    if(evt.id==Event.WINDOW_DESTROY)System.exit(0);

```

Programmende wird beim Schließen den Fensters durchgeführt.

```

    return false;
}
}

```

```

import java.awt.*;
public class Sin extends Frame implements Runnable{
    double time;
    Thread t;
    public Sin(){
        super("Function");
        resize(400,400);
        show();
        t=new Thread(this);
        t.start();

```

Klassen, die das Interface Runnable interpretieren, müssen die Methode "public void run()" implementieren. Diese wird ausgelöst, wenn auf eine Instanz des Objekts die Methode start() angewandt wird. In diesem Fall wird der Klasse ein Attribut t des Typs Thread mitgegeben, der mit "t=new Thread(this)" instanziiert wird und mit "t.start()" gestartet wird. Damit beginnt die Methode run() als selbstständiger Thread zu laufen.

```

}
public void paint(Graphics g){
    g.setColor(Color.black);
    g.drawLine(0,200,400,200);
    g.drawLine(200,0,200,440);
    g.setColor(Color.blue);
    int oldx=-200;
    int oldy=(int)(75*f((double)-200/75));
    for(int i=0;i<100;i++){
        int x=i*4-200;
        int y=(int)(75*f((double)x/75));
        g.drawLine(200+oldx,200-oldy,200+x,200-y);
        oldx=x;
        oldy=y;
    }
}

```

Diese Methode ist identisch mit der Methode in der Klasse Function.

```
}  
public double f(double x){  
    return Math.sin(x+time);  
}
```

Mit "Math.sin(double d)" erhält man den Sinus des Winkels d (Radiant). Das Argument des Sinus wird um time verschoben, die Variable time ist Attribut der Instanz und wird von der run() Methode verändert, so dass auf dem Bildschirm eine Sinusschwingung in Aktion zu sehen ist.

```
}  
public void run(){  
    while(true){
```

Die Sinusschwingung bewegt sich immer weiter, also wird eine endlose while-Schleife gebraucht.

```
    try{  
        Thread.sleep(100);  
        time+=Math.PI/45;  
        repaint();
```

Alle hundert Millisekunden (0.1 Sekunden) wird die Sinusschwingung verschoben und der Bildbereich neu gezeichnet (repaint() ruft in Frame-Objekten die paint-Methode mit dem aktuellen Grafikkontext auf). Die try-Klausel fängt eventuelle Fehler bei der Warte-Aktion Thread.sleep(100) ab.

```
    }  
    catch(Exception e){}  
    }  
}  
public static void main(String[]args){  
    new Sin();  
}  
public boolean handleEvent(Event evt){  
    if(evt.id==Event.WINDOW_DESTROY)System.exit(0);  
    return false;  
}  
}
```

```
import java.awt.*;  
public class Fraktal extends Frame{  
    double[] doublex;  
    double[] doubley;
```

Dieses Programm beginnt mit der Annäherung eines Kreises und ergänzt die Eckpunkte mit Punkten zwischen den Eckpunkten, die um einen zufälligen Wert von dem Mittelpunkt der direkten Verbindung abweichen. Es entsteht eine etwas unförmige kreisartige Figur.

```
public Fraktal(){  
    resize(400,400);  
    show();
```

```

}
public void paint(Graphics g){
    fill();
}

```

Es werden mit fill() die Koordinaten der Fraktalpunkte in die int-Arrays doublex und doubley eingelesen.

```

int oldx=(int)doublex[0];
int oldy=(int)doubley[0];
g.setColor(Color.black);
int n=doublex.length;
for(int i=0;i<n;i++){
    int newx=(int)doublex[(i+1)%n];
    int newy=(int)doubley[(i+1)%n];
    g.drawLine(200+newx,200+newy,200+oldx,200+oldy);
    oldx=newx;
    oldy=newy;
}

```

Das Zeichnen auf dem Bildschirm verläuft so wie bei den vorhergehenden paint-Methoden, nur sind die Koordinaten in doublex und doubley schon gegeben. (i+1)%n ist der Rest der Division von i+1 durch n und erfüllt den Zweck, dass der Fall i=n-1 nicht gesondert betrachtet werden muss, um den Kreis zu schließen.

```

}
}
public void fill(){
    doublex=new double[20];
    doubley=new double[20];
    for(int i=0;i<20;i++){
        doublex[i]=125*Math.sin(Math.PI*2*i/20);
        doubley[i]=125*Math.cos(Math.PI*2*i/20);
    }
}

```

Ein angenäherter Kreis mit 20 Eckpunkten wird gespeichert. Math.sin(double d) und Math.cos(double d) sind JAVA-Systemmethoden, die nicht zusätzlich importiert werden müssen.

```

for(int j=0;j<4;j++){
    System.out.println("Stage: "+j);
    int len=doublex.length;
    double[] doublex2=new double[len*2];
    double[] doubley2=new double[len*2];
    for(int i=0;i<len;i++){
        double xdiff=doublex[(i+1)%len]-doublex[i];
        double ydiff=doubley[(i+1)%len]-doubley[i];
        double basepointx=doublex[i]+xdiff/2;
        double basepointy=doubley[i]+ydiff/2;
    }
}

```

Zunächst wird die Differenz der beiden Punkte berechnet und dann der Mittelpunkt (basepointx,basepointy).

```

double tangent=-xdiff/ydiff;

```

In tangent wird die Steigung der Senkrechten der Verbindungslinie festgehalten.

```

doublex2[i*2]=doublex[i];

```

```

doubley2[i*2]=doubley[i];
double grade=Math.random()-0.5;
doublex2[i*2+1]=basepointx+xdiff*grade;
doubley2[i*2+1]=basepointy+ydiff*grade*tangent;

```

Vom Mittelpunkt aus wird der neue Punkt in etwa entlang der Senkrechten zufällig ausgewählt.

```

}

```

Vier Mal wird das Array verdoppelt, in dem man zwischen zwei Punkten einen Zwischenpunkt in einer Region zufällig auswählt.

```

        doublex=doublex2;
        doubley=doubley2;
    }
}
public static void main(String[]args){
    new Fraktal();
}
public boolean handleEvent(Event evt){
    if(evt.id==Event.WINDOW_DESTROY)System.exit(0);
    return false;
}
}
}

```

```

import java.awt.*;
class Breakout extends Frame implements Runnable{
    Thread t;
    int[][] stones;
    double x;
    double y;
    double xdrift;
    double ydrift;
    int xbrett;
    Image img;
    Graphics h;
    boolean gameover=false;
}

```

Das Breakout-Spiel wird in einem Fenster gespielt und deshalb ist die zu erzeugende Instanz des Spiels eine Instanz einer Subklasse von Frame. Das stones-Array verwaltet, welche Steine noch zu treffen sind, x und y sind die Position des Balls, xdrift und ydrift der Geschwindigkeitsvektor und xbrett der x-Wert des Schlagpanels. Der boolesche Wert gameover teilt der run-Methode (implements Runnable) mit, ob das Spielbild geupdated werden muss, oder ob der Spieler verloren hat. Die Position und der Geschwindigkeitsvektor sind deshalb im Format double, weil dann der Flug des Balls sauberer scrollt.

```

public Breakout(){
    super("Breakout");
    resize(400,440);
    setLayout(null);
    stones=new int[20][5];
    t=new Thread(this);
}

```

Das Array verwaltet 20*5 Steine in der oberen Bildschirmhälfte.

```
img=new ImageFrame().img;
h=img.getGraphics();
```

Das Image wird als Bildschirmbuffer verwendet und h ist der zugehörige Grafikkontext. Das Bild muss in einer eigenen Klasse ImageFrame erzeugt werden, sonst gibt es eine Nullpointerexception (warum?).

```
show();
x=200;
y=200;
xdrift=1.5;
ydrift=-4;
xbrett=200;
t.start();
```

Die Startposition des Balls ist (200,200) und der Geschwindigkeitsvektor (1.5,-4). Die run() Methode dieser Klasse ("t=new Thread(this)") wird mit "t.start()" als eigener Prozess gestartet.

```
    }
    public void run(){
        while(true){
            try{
                Thread.sleep(100);
                if(!gameover)repaint();
            }
        }
    }
}
```

(wie in Klasse Sin)

```
    }
    catch(Exception e){
    }
}
}
}
public void update(Graphics g){
    h.setColor(Color.white);
    h.fillRect(0,0,400,440);
}
```

Der Inhalt des Grafikkontext h wird gelöscht.

```
h.setColor(Color.black);
for(int i=0;i<20;i++){
    for(int j=0;j<5;j++){
        if(stones[i][j]==0){
            h.fillRect(i*20+1,j*20+41,18,18);
        }
    }
}
```

Alle nicht abgetroffenen Steine (stones[i][j]==0) werden gemalt.

```
x=x+xdrift;
y=y+ydrift;
h.setColor(Color.blue);
```

```
h.fillOval((int)x-2,(int)y-2,4,4);
```

Der Ball wird verschoben und an seiner neuen Position blau gemalt.

```
h.fillRect(xbrett,380,30,20);
```

Das Schlagpanel wird an der x-Mausposition auf der Höhe 380 gemalt mit Breite 30 und Höhe 20.

```
if(y<40)ydrift=-ydrift;
else if(y>400){
    gameover=true;
}
```

Wenn der Ball das Schlagpanel hinter sich gelassen hat, ist das Spiel verloren.

```
if(x<0)xdrift=-xdrift;
else if(x>400)xdrift=-xdrift;
```

Die linke und rechte Seite des Spielfelds verursachen das Abprallen des Balls.

```
if(x>xbrett&& x<xbrett+30){
    if(y>380&& y<400){
        ydrift=-ydrift;
        xdrift=(x-((double)xbrett+15))/3;
    }
}
```

Trifft der Ball das Schlagpanel, wird seine neue Flugrichtung so berechnet, dass der Ball links von der Mitte nach links fliegt und rechts von der Mitte nach rechts. Die y-Richtung dreht sich einfach um.

```
for(int i=0;i<20;i++){
    for(int j=0;j<5;j++){
        if(stones[i][j]==0&& x>=i*20+1&& x<=i*20+19&& y>=j*20+41&& y<=j*20+59){
            stones[i][j]=1;
            ydrift=-ydrift;
        }
    }
}
```

Ist der Ball innerhalb des Gebietes eines Steins und der Stein nicht abgetroffen, so dreht sich die y-Bewegung des Balls um und es wird `stones[i][j]=1` gesetzt (abgetroffen). Das Gebiet eines Steins (i,j) ist das Intervall $[i*20+1, i*20+19] \times [j*20+41, j*20+59]$.

```
    }
}
}
g.drawImage(img,0,0,this);
paint(g);
```

Erst jetzt wird der gepufferte Bildschirminhalt vollständig auf den Bildschirm geschrieben. Dies vermeidet das Flackern des Fensterinhalts.

```
    }
public void paint(Graphics g){
}
public static void main(String[]args){
    new Breakout();
}
```

```

}
public boolean handleEvent(Event evt){

```

Diese Methode wird bei User-Interface Ereignissen eines Fensters automatisch aufgerufen (zB. Mouseclick, Mouse bewegt, Butten gedrückt, Menü gewählt, Taste gedrückt, Fenster geschlossen).

```

    if(evt.id==Event.WINDOW_DESTROY){
        System.exit(0);
    }
    else if(evt.id==Event.MOUSE_MOVE){
        xbrett=evt.x;
    }

```

Wird die Maus bewegt (in evt.x wird ihre Position übermittelt), so wird ihre x-Position in xbrett gespeichert. Auf xbrett kann zugegriffen werden, weil es ein Attribut der aktuellen Instanz ist.

```

    else if(evt.id==Event.KEY_PRESS){
        stones=new int[20][5];
        x=200;
        y=200;
        xdrift=1.5;
        ydrift=-4;
        xbrett=200;
        gameover=false;

```

Beim Drücken einer Taste werden alle Variablen, die für das Spiel wichtig sind, neu initialisiert (Positionsvektor des Balls, Geschwindigkeitsvektor, Steine, gameover Variable).

```

    }
    return false;
}
}
class ImageFrame extends Frame{
    Image img;
    public ImageFrame(){
        pack();
        img=createImage(600,600);
    }

```

Diese Klasse wird von Breakout zur Erzeugung des Bildes und Grafikkontext benötigt.

```

}

```

```

import java.io.*;

```

Mit dieser import-Anweisung werden JAVA-Systemklassen zur Verwendung von Dateien mitimportiert.

```

class WriteAndCount{
    public static void writeZufall(String fname,int length){
        try{

```

```
FileOutputStream fos=new FileOutputStream(fname);
```

Eine Datei mit dem Namen fname wird erzeugt. fos ist das Objekt, mit dem Daten in die Datei geschrieben werden.

```
for(int i=0;i<length;i++){
    fos.write((int)(Math.random()*256));
```

length zufällige Bytes (0...255) fügt die Methode fos.write (int i) der Datei fname an.

```
}
fos.close();
```

Der Zugriff auf die Datei sollte mit fos.close() beendet werden.

```
}
catch(Exception e){
    System.out.println("Exception while writing file");
```

Bei der Behandlung von Dateien muss man prinzipiell immer diverse Ausnahmefehler abfangen.

```
}
}
public static int countmuster(String fname,byte[]by){
    int ret=0;
    try{
        File f=new File(fname);
        int len=(int)f.length();
```

Mit File(fname) erzeugt man ein eine Datei repräsentierendes Objekt und mit "int len=(int)f.length()" bekommt man die Länge der Datei.

```
FileInputStream fis=new FileInputStream(fname);
int[] filearray=new int[len];
for(int i=0;i<len;i++){
    filearray[i]=fis.read();
}
```

Alle Bytes der Datei befinden sich nach der Ausführung dieses Codestücks im Array filearray der Länge len. Auch hier müssen Lesefehler abgefangen werden (try). "int i=fis.read()" liest das jeweils nächste Byte des ankommenden Datenstroms der Datei.

```
for(int i=0;i<len-by.length;i++){
    boolean matches=true;
    for(int j=0;j<by.length;j++){
        if(by[j]!=(byte)filearray[i+j]){
            matches=false;
        }
    }
    if(matches)ret++;
```

Es soll getestet werden, wie oft das Muster by (byte Array der Länge by.length) in der Datei vorkommt, also im Array filearray. Dazu werden zwei for-Schleifen benutzt. Die erste zählt alle möglichen Beginn-Indizes des vorkommenden Musters durch und in der zweiten for-Schleifen werden die folgenden Indizes

auf Gleichheit überprüft. Vor Beginn der inneren Schleife ist der boolesche Wert matches true, stimmt irgendein Zeichen nicht überein, wird er false. Ist matches nach Ende der Schleife noch true, wird der Zähler des Vorkommens inkrementiert.

```
    }
    fis.close();
}
catch(Exception e){};
return ret;
```

Die Datei wird geschlossen und der Zähler zurückgegeben.

```
    }
    public static void main(String[] args){
        writeZufall("zout.txt",1000);
        System.out.println(countmuster("zout.txt",new byte[1]));
    }
```

Dieses Beispiel schreibt eine 1000 Bytes lange zufällige Datei und zählt die 0 (new byte[100] erzeugt ein byte-Array der Länge 100 und initialisiert die Variablen der Liste mit 0).

```
    }
}
```

```
import java.io.*;
class Appender{
    public static void addLine(String fname,String line){
        try{
            FileOutputStream fos=new FileOutputStream(fname,true);
            PrintStream ps=new PrintStream(fos);
            ps.println(line);
            ps.close();
            fos.close();
        }
```

Diese Methode schreibt die Zeile line in die Datei fname. Wenn man ein PrintStream Objekt aus dem FileOutputStream Objekt macht ("PrintStream ps=new PrintStream(fos)") kann man mit ps.println(String line) ganze Textstrings als Zeile in die Datei schreiben.

```
    }
    catch(Exception e){
    }
}
```

try-catch Klauseln sind bei Dateioperationen obligatorisch (I/O Fehler wie "File Not Found").

```
    }
    public static void readFile(String fname){
        try{
            String inline=null;
            FileInputStream fis=new FileInputStream(fname);
            DataInputStream dis=new DataInputStream(fis);
            while((fname=dis.readLine())!=null){
                System.out.println(fname);
            }
        }
```

```
}
```

Man kann auch ganze Textstrings aus einer Datei lesen ("String s=dis.readLine()"), wenn man aus einem FileInputStream Objekt einen DataInputStream macht ("DataInputStream dis=new DataInputStream(fis)"). Die Datei ist zuende, wenn readLine() den Nullstring liefert, so lange läuft auch die while-Schleife. readFile(String fname) gibt also den Inhalt einer Datei mit Textzeilen auf dem Bildschirm aus. Einen Nullstring stellt man übrigens mit "s==null" fest und nicht mit "s=="null"".

```
        dis.close();
        fis.close();
    }
    catch(Exception e){
    }
}
public static void main(String[]args){
    addLine(args[0],args[1]);
    readFile(args[0]);
}
```

Das Beispiel erwartet zwei Argumente der Kommandozeile. Das zweite Argument wird in eine Datei geschrieben, das erste wird als Dateiname interpretiert. Mit dem zweiten Methodenaufruf wird die Datei ausgegeben. Man kann aus mit "java fname.txt hallo" der Datei fname.txt den String hallo anfügen.

```
}
}
```

```
import java.awt.*;
class GUIDemo extends Frame{
    TextArea ta;
    TextField tf;
    Button b;
    Choice ch;
    Checkbox cb;
    Menu m;
```

In diesem Beispiel werden einige Elemente des graphical user inferface demonstriert. In diesem Fenster finden eine TextArea, ein TextField, ein Button, eine Choice, eine Checkbox und ein Menu Platz. Die Namen sind bezüglich ihres Aussehens selbsterklärend.

```
public GUIDemo(){
    super("J2SE GUI Demo");
    resize(400,400);
    setLayout(null);
    ta=new TextArea();
    tf=new TextField();
    b=new Button("action");
    ch=new Choice();
    cb=new Checkbox("CB ");
    m=new Menu("Menu");
}
```

Manche Objekte verlangen einen String als Konstruktionsargument. Die Fenstergröße ist 400X400.

```
m.add(new MenuItem("Item 1"));
```

```
m.add(new MenuItem("Item 2"));
m.add(new MenuItem("Item 3"));
```

Das Menü m hat drei Einträge.

```
MenuBar mb=new MenuBar();
mb.add(m);
setMenuBar(mb);
```

Die Menüleiste ist ein eigenes Objekt, dem die Menüs angefügt werden ("mb.add(Menu m)"), mit "setMenuBar(MenuBar mb)" bekommt der Frame die Menüleiste.

```
add(ta);
add(tf);
add(b);
add(ch);
add(cb);
```

Der Frame erhält noch die anderen Objekte als Bedienelemente.

```
ta.reshape(10,60,200,190);
tf.reshape(10,260,100,27);
b.reshape(120,260,100,27);
```

Mit item.reshape(int x,int y,int w,int h) wird einem Bedienelement eine Position (x,y) zugewiesen, auch seine Ausbreitung wird durch (w,h) bestimmt.

```
ch.addItem("CH Item 1");
ch.addItem("CH Item 2");
```

Die Choice hat zwei Elemente zum Auswählen.

```
ch.reshape(10,300,100,27);
cb.reshape(10,330,100,27);
show();
```

Das show() sorgt dafür, dass der Frame mit dem Aufruf des Konstruktors angezeigt wird.

```
}
public boolean handleEvent(Event evt){
    if(evt.id==Event.WINDOW_DESTROY)System.exit(0);
    else if(evt.id==Event.ACTION_EVENT){
        ta.appendText(""+evt.target+"/"+evt.arg+"\n");
    }
    else if(evt.id==Event.MOUSE_DOWN){
        ta.appendText("MouseDown "+evt.x+"/"+evt.y+"\n");
    }
    else if(evt.id==Event.KEY_ACTION||evt.id==Event.KEY_ACTION){
        ta.appendText("Key "+evt.key+"\n");
    }
}
```

Tritt ein Ereignis auf, wird je nach Aktion eine Ausgabe über die Art des Ereignisses durchgeführt und zwar in der TextArea (ta.appendText("hallo\n(neue Zeile)"). Beim Klick eines Button wird "ACTION_EVENT" mit dem geklickten Button als evt.target ausgelöst.

```

    return false;
}
public static void main(String[] args){
    new GUIDemo();
}

```

Das Programm kann mit "java GUIDemo" gestartet werden.

```

}
}

```

```

import java.io.*;
import java.net.*;

```

I/O- und Internetbibliotheken werden importiert.

```

class TCPIM implements Runnable{

```

TCPIM integriert die Funktion eines Instant Messengers, wobei TCPS auf eingehende Daten auf Port 790 des lokalen Computers lauscht und TCPIM Eingabedaten an eine Internetadresse sendet. Beide Komponenten müssen unabhängig voneinander ständig laufen und sind deshalb Runnables (Threads).

```

    TCPS tcps;
    String connecto;
    Thread t;
    public TCPIM(String connecto){
        this.connecto=connecto;
    }

```

String connecto ist diejenige IP-Adresse, mit der gechattet werden soll.

```

        t=new Thread(this);
        t.start();
        tcps=new TCPS();
    }

```

Wird TCPIM gestartet, läuft immer der Port-790-Lauscher TCPS mit.

```

    }
    public void run(){
        try{
            System.in.read();
            Socket so=new Socket(connecto,790);
            OutputStream out=so.getOutputStream();
        }
    }

```

Nach dem Drücken einer Taste wird eine Internet-Verbindung (Socket) zur Adresse connecto (Port 790) aufgebaut und eine Schreibverbindung als OutputStream geholt ("so.getOutputStream()").

```

        while(true){
            int esc=System.in.read();
            if(esc==27)System.exit(0);
            out.write(esc);
        }
    }
}

```

Die Endlosschleife wird mit der Escape-Taste (27) beendet. Mit "int esc=System.in.read();" wird auf das Drücken einer Taste gewartet und mit out.write(esc) das Zeichen per Stream ins Netz geschickt an die Adresse connecto.

```
    }  
  }  
  catch(Exception e){};
```

Exceptions treten natürlich bei Internet-Datenströmen gehäuft (zB. "URL not found") auf, deshalb müssen Exceptions bei Netzzugriffen mit try abgefangen werden.

```
  }  
  public static void main(String[]args){  
    new TCPIM(args[0]);
```

Startet man TCPIM, wird die Zieladresse als Kommandozeilenargument übergeben. Mit "java TCPIM 127.0.0.1" kann man das Programm testen und mit sich selbst chatten. Denn die Adresse 127.0.0.1 ist der Local Host, die eigene Adresse.

```
  }  
}  
class TCPS implements Runnable{  
  ServerSocket seso;  
  Thread t;  
  public TCPS(){  
    try{  
      seso=new ServerSocket(790);  
      System.out.println("your ip (port 790): ");  
      System.out.println(seso.getInetAddress().getLocalHost().getHostAddress());
```

Ein ServerSocket ist das Gegenstück zu einem Socket. Es lauscht unter der eigenen IP-Adresse auf einem speziellen Port (in diesem Fall 790) auf eingehende Verbindungen. Die System.out.println-Aufrufe geben die eigene IP-Adresse aus, wenn man im Netz ist. Unter dieser Adresse können andere Nutzer Chat-Verbindungen zu einem selbst aufbauen (String connecto).

```
  }  
  catch(Exception e){};  
  t=new Thread(this);  
  t.start();
```

Das Lauschen ist ein zum Warten auf Eingabe unabhängiger Prozess, deshalb wird es als eigener Thread implementiert.

```
  }  
  public void run(){  
    while(true){  
      try{  
        Socket inps=seso.accept();  
        new GetData(inps);
```

In einer Endlosschleife nimmt das ServerSocket eingehende Verbindungen ("Socket inps=seso.accept();") an und erzeugt bearbeitende Objekte ("new GetData(inps)").

```

    }
    catch(Exception e){};
}
}
}
class GetData implements Runnable{
    Socket inps;
    Thread t;
    public GetData(Socket inps){
        this.inps=inps;
        t=new Thread(this);
        t.start();
    }
}

```

Jedes GetData Objekt liest die Daten einer eingehenden Verbindung. Dies ist wiederum ein eigenständiger Prozess für den ein Thread reserviert wird.

```

}
public void run(){
    try{
        InputStream is=inps.getInputStream();
    }
}

```

Die eingehende Verbindung wird zum Lesen geöffnet.

```

while(true){
    byte[]by=new byte[1];
    by[0]=(byte)is.read();
    System.out.print(new String(by));
}

```

Der JAVA-Interpreter wartet auf Zeichen, die in der Verbindung ankommen ("is.read()") und gibt sie als String aus, nachdem sie in ein byte-Array kopiert wurden. Man kann aus Bytes in einem Array einen String erzeugen mit Hilfe des Konstruktors "String s=new String(byte[]bytes)".

```

}
}
catch(Exception e){
}
}
}
}

```

```

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

```

In diesem kurzen Intermezzo beschäftigen wir uns mit der J2ME-Programmierung, also mit der JAVA-Programmierung für Handys. Es gibt diverse Einschränkungen, aber die Typen boolean und alle Ganzzahl-Typen sowie die Standardschleifen gibt es auch in J2ME und die Notation von Attributen, Methoden, Konstruktoren, Klassen, Interfaces und Vererbungen ist genauso. Die Hauptunterschiede liegen im GUI und im Fehlen der Unterstützung von Gleitkommazahlen und der Umfang der Systembibliotheken ist viel kleiner. Das J2ME ist wie das J2SE kostenlos unter <http://www.javasoft.com> verfügbar.

```
public class Fibo extends MIDlet{
```

Ausführbare Programme auf JAVA-Handys sind MIDlets Sie müssen die folgenden Methoden enthalten:

```
    Display d;
    public void destroyApp(boolean unconditional){
    }
    public void pauseApp(){
    }
    public void startApp(){
        Inputer inp=new Inputer();
        d=Display.getDisplay(this);
        d.setCurrent((Displayable)inp);
```

Mit diesen drei Statements wird eine neue Instanz der Unterklasse Inputer der Klasse Form (Dialogformular) erzeugt und auf das Handy-Display geworfen.

```
    }
}
class Inputer extends Form implements CommandListener {
    Command co;
    Command co2;
    TextField tf;
```

Commands sind Belegungen für die OK- und die Zurücktaste des Handys. Wird eine Taste mehrfach belegt öffnet sich ein Menü zur Auswahl. Das TextField ist genauso wie in J2SE.

```
    public Inputer(){
        super("Fibonacci");
        co=new Command("rekursiv",Command.BACK,1);
        co2=new Command("iterativ",Command.OK,1);
```

Der Titel des Formulars (Form) ist "Fibonacci" und die Commands co und co2 bekommen die Beschriftungen "rekursiv" und "iterativ" und werden ihren Handy-Tasten zugeordnet.

```
        addCommand(co);
        addCommand(co2);
```

Mit addCommand werden die Commands dem aktuellen Formular zugeordnet.

```
        setCommandListener(this);
```

Das Interface CommandListener wird von der Klasse selbst implementiert, deshalb wird die Methode commandAction(Command c,Displayable d2) bei der Auslösung eines Commands vom System aufgerufen. c ist ein Verweis auf den ausgelösten Command.

```
        tf=new TextField("Number ", "",64,TextField.ANY);
        append(tf);
```

Der Konstruktor des TextFields im J2ME unterscheidet sich deutlich von dem im J2SE. "Number " ist eine externe Beschriftung, "" der standardmäßige Inhalt, 64 die maximale Länge und TextField.ANY teilt dem GUI mit, dass jedes Zeichen eingegeben werden darf. Mit dem Statement append(Item i) fügt man dem Formular neue Items hinzu.

```

}
public void commandAction(Command c, Displayable d2){
    if(c==co){
        append(""+getFib(Integer.parseInt(tf.getString()))+"\n");

```

Man konvertiert den Inhalt des TextFields in einen int und berechnet die n-te Fibonaccizahl rekursiv.

```

}
else if(c==co2){
    int num=Integer.parseInt(tf.getString());

```

Mit Integer.parseInt werden Zahlenstrings in int-Zahlen konvertiert.

```

int[] fibs=new int[num+1];
fibs[0]=0;
fibs[1]=1;
for(int i=2;i<num+1;i++){
    fibs[i]=fibs[i-2]+fibs[i-1];
}

```

Die iterative Berechnung (wie Sum) ist natürlich schneller.

```

append(""+fibs[num)+"\n");

```

Mit append(String s) gibt man am Ende des Formulars auch Strings aus.

```

}
}
public int getFib(int i){
    if(i<2)return i;
    else return getFib(i-2)+getFib(i-1);

```

(einfache rekursive Berechnung der i-ten Fibonaccizahl.)

```

}
}

```

```

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

```

J2ME GUI

```

import javax.microedition.rms.*;

```

Systemklassen zur Datenspeicherung

```

public class Editor extends MIDlet{

```

Wie in Fibo ist das MIDlet ausführbare Programmklasse.

```

    Display d;

```

```

    public void destroyApp(boolean unconditional){
    }
    public void pauseApp(){
    }
    public void startApp(){
        Loader loa=new Loader(this);
        d=Display.getDisplay(this);
        d.setCurrent((Displayable)loa);
    }
}
class Loader extends Form implements CommandListener{
    ChoiceGroup l;
    Command co;
    Command co2;
    MIDlet m;
    public Loader(MIDlet m){
        super("Load Form");
        co=new Command("Load",Command.BACK,1);
        addCommand(co);
        setCommandListener(this);
        this.m=m;
    }
}

```

Das MIDlet wird übergeben, um die Anzeige des Displays später zu ändern.

```

    l=new ChoiceGroup("Load file: ",ChoiceGroup.EXCLUSIVE);

```

Eine ChoiceGroup ist eine Auswahlliste, in diesem Fall mit der Überschrift "Load file" und dem Single-Select-Modus (ChoiceGroup.EXCLUSIVE).

```

    loadFiles();
    append(l);

```

ChoiceGroups sind Items und können deshalb mit append angehängt werden.

```

    }
    public void loadFiles(){
        String[] stores=RecordStore.listRecordStores();
    }

```

Alle dem MIDlet zugeordneten Dateien werden in String[]stores gespeichert.

```

    l.append("newfile.txt",null);

```

Ausserdem soll es die Möglichkeit geben, eine neue Datei zu eröffnen.

```

    if(stores!=null){
        for(int i=0;i<stores.length;i++){
            l.append(stores[i],null);
        }
    }

```

Die Dateinamen werden als Element der ChoiceGroup zur Anwahl bereitgestellt.

```

    }
    }
    public void commandAction(Command c,Displayable d2){

```

```

if(c==co){
    String name=l.getString(l.getSelectedIndex());
    Editor2 edi=new Editor2(m,name);
    Display d=Display.getDisplay(m);
    d.setCurrent((Displayable)edi);
}
}
}
}

```

Nach der Betätigung des Load-Button übergibt man einer neuen Instanz des Editors wieder das MIDlet und den zu öffnenden Dateinamen. Mit "l.getString(l.getSelectedIndex())" erhält man den selektierten Eintrag der ChoiceGroup.

```

}
}
}
}
}
class Editor2 extends TextBox implements CommandListener {

```

Es gibt im J2ME eine eigene Klasse für einen Editor, deren Eigenschaften man für eigene Editoren vererben kann. Die Subklasse zeigt man mit "d.setCurrent(TextBox b)" an, sie muss nicht Element eines Formulars sein. Man beachte, dass Variablen nur einmal mit Typ deklariert werden und dann mit Werten initialisiert werden. Die Referenzierung erfolgt dann ausschließlich über den Variablennamen. Bei näheren Methodenbeschreibungen im erläuternden Text wird der Typ als allgemeine Aussage angegeben.

```

MIDlet m;
Command co;
Command co2;
String name;
public Editor2(MIDlet m,String name){
    super("Editor","",4096,TextField.ANY);

```

Der Konstruktor der Oberklasse verlangt ein Label, einen Anfangstext, eine maximale Textlänge und eine Eingabeinformation wie bei TextFields.

```

co=new Command("Save",Command.BACK,1);
co2=new Command("Exit",Command.BACK,1);
addCommand(co);
addCommand(co2);
setCommandListener(this);
this.m=m;
this.name=name;
try{
    RecordStore rs=RecordStore.openRecordStore(name,false);
    byte[] by=rs.getRecord(1);
    String sets=new String(by);
    setString(sets);

```

Die dem Konstruktor bekannte Datei (name) wird geöffnet und der erste Record gelesen ("rs.getRecord(1)"). Ein Record ist eine Ansammlung von Bytes, die mit "new String(byte[]by)" in einen String konvertiert und mit "setString(sets);" in die TextBox geschrieben wird.

```

}
catch(Exception e){System.out.println(name+e);};

```

Hier kann eine Exception auftreten (RecordStore leer oder falsche RecordID). Theoretisch können die Daten in beliebigen ersten n Records stehen, aber für diesen Editor ist es Konvention, die Daten im ersten Record zu behalten.

```

}
public void commandAction(Command c,Displayable d2){
    if(c==co){
        byte[] by=getString().getBytes();

```

Der Inhalt der TextBox ist im byte-Array by. "getString()" liest den Inhalt der TextBox in einen String.

```

        Saver sav=new Saver(m,by,name,this);

```

Das Speicherformular bekommt die Bytes in der TextBox und den Dateinamen zur Verarbeitung.

```

        Display d=Display.getDisplay(m);
        d.setCurrent((Displayable)sav);
    }
    if(c==co2){
        Loader loa=new Loader(m);
        Display d=Display.getDisplay(m);
        d.setCurrent((Displayable)loa);

```

Mit dem zweiten Command kehrt man zum LadeFormular zurück.

```

    }
}
}
class Saver extends Form implements CommandListener {
    TextField tf;
    Command co;
    Command co2;
    MIDlet m;
    byte[]rec;
    String name;
    Editor2 edt;
    public Saver(MIDlet m,byte[]rec,String name,Editor2 edt){
        super("Save Form");
        tf=new TextField("Filename ",name,30,TextField.ANY);

```

In dieses TextField kann man den zu speichernden Dateinamen eingeben.

```

        this.m=m;
        this.rec=rec;
        this.edt=edt;
        this.name=name;
        co=new Command("Save",Command.OK,1);
        co2=new Command("Back",Command.BACK,1);
        addCommand(co);
        addCommand(co2);
        setCommandListener(this);
        append(tf);

```

Dieses Formular besteht aus zwei Commands und einem TextField.

```

}
public void commandAction(Command c,Displayable d2){
    if(c==co){

```

```

try{
    name=tf.getString();
    RecordStore rs=RecordStore.openRecordStore(name,true);

```

(true: create if necessary)

```

try{
    rs.setRecord(1,rec,0,rec.length);
}
catch(Exception e){
    rs.addRecord(rec,0,rec.length);
}

```

Wenn die Datei existiert und einen Record enthält überschreibt man diesen ("setRecord(int id,byte[]rec,int anfang,int ende)"), sonst (Ausnahmefehler) wird ein Record zur leeren Datei hinzugefügt ("addRecord(byte[]rec,int anfang,int ende)").

```

Loader loader=new Loader(m);
Display d=Display.getDisplay(m);
d.setCurrent((Displayable)loader);

```

Nach erfolgreicher Speicherung lädt man die nächste Datei.

```

}
catch(Exception e){System.out.println(e);};
}
if(c==co2){
    Display d=Display.getDisplay(m);
    d.setCurrent((Displayable)edt);
}

```

Hier springt der Interpreter zur ursprünglichen TextBox zurück. Der Konstruktor speicherte die aufrufende TextBox in der Variablen edt.

```

}
}
}

```

```

import java.awt.*;
class Automat extends Frame{
    List zustaende;
    List regeln;
    Choice startz;
    Choice endz;
    TextField neuzustand;
    Button addzustand;
    TextField input;
    TextField output;
    TextField folgezustand;
    Button addregel;
    TextField automatin;
    TextField automatout;
}

```

```
TextField result;  
Button start;
```

Die GUI-Elemente sind selbsterklärend und sind zusammen eine Oberfläche, mit der man Zustände und Überführungsregeln eingeben kann.

```
Zustand[] zust;  
int zlen;  
Zustand currentzust;
```

Zentrale Objekte sind Zustände, denen als Unterobjekte Regeln zugeordnet sind.

```
public Automat() {  
    super("Automat");  
    resize(400,390);  
    setLayout(null);
```

Diese Anweisung ist unbedingt erforderlich, um Grafikfehler zu vermeiden.

```
    zust=new Zustand[0];  
    zustaende=new List();  
    regeln=new List();  
    startz=new Choice();  
    endz=new Choice();  
    startz.addItem("Startzustand");  
    startz.addItem("kein Startzustand");  
    endz.addItem("Endzustand");  
    endz.addItem("kein Endzustand");  
    neuzustand=new TextField("neuer zustand");  
    addzustand=new Button("new zustand");  
    input=new TextField("eingabe");  
    output=new TextField("ausgabe");  
    folgezustand=new TextField("folgezustand");  
    addregel=new Button("add regel");  
    automatin=new TextField("input");  
    automatout=new TextField("output");  
    start=new Button("start");  
    result=new TextField("result");  
    zustaende.reshape(10,40,180,150);  
    regeln.reshape(210,40,180,150);  
    startz.reshape(10,200,80,27);  
    endz.reshape(110,200,80,27);  
    neuzustand.reshape(10,240,80,27);  
    addzustand.reshape(110,240,80,27);  
    input.reshape(210,240,80,27);  
    output.reshape(310,240,80,27);  
    folgezustand.reshape(210,270,80,27);  
    addregel.reshape(310,270,80,27);  
    automatin.reshape(10,280,80,27);  
    automatout.reshape(110,280,80,27);  
    start.reshape(10,320,80,27);  
    result.reshape(10,350,80,27);  
    add(zustaende);  
    add(regeln);
```

```

    add(startz);
    add(endz);
    add(neuzustand);
    add(addzustand);
    add(input);
    add(output);
    add(folgezustand);
    add(addrregel);
    add(automatin);
    add(automatout);
    add(start);
    add(result);
    show();
}
public static void main(String[] args){
    new Automat();
}
public void runProg(){
    String input=automatin.getText();

```

Aus dem TextField automatin liest man den String, auf den der Automat angesetzt wird. Die Methode runProg ist die zentrale Ablaufmethode.

```

    automatout.setText("");

```

Ein neuer Durchlauf verlangt ein leeres Ausgabefeld.

```

    int n=input.length();

```

Die Länge des Inputs gibt an, wieviele Überführungsregeln der Interpreter anwendet.

```

    Zustand cur=null;
    for(int i=0;i<zlen;i++){
        if(zust[i].startzustand){
            cur=zust[i];
        }
    }

```

Unter den Zuständen wird der Startzustand gesucht und in cur festgehalten.

```

    }
    for(int i=0;i<n;i++){
        String chara=input.substring(i,i+1);

```

chara ist das aktuelle Zeichen, zu dem man eine passende Regel sucht.

```

    Regel[] rl=cur.regeln;
    int n2=rl.length;
    Regel r=null;
    for(int j=0;j<n2;j++){
        if(rl[j].input.equals(chara))r=rl[j];

```

Unter den Regeln zum Zustand sucht man eine, bei der Eingabezeichen und das aktuelle Zeichen übereinstimmen. r ist ein Regelobjekt, das das Ergebnis der Suche repräsentiert.

```

    }
    automatout.setText(automatout.getText()+r.output);

```

Eine Regel kann bei ihrer Ausführung dem Nutzer eine Ausgabe mitteilen.

```

for(int j=0;j<zlen;j++){
    if(zust[j].name.equals(r.folgezustand))cur=zust[j];

```

Der Folgezustand wird als String festgehalten, deshalb sucht die for-Schleife einen neuen Zustand, dessen Name gleich ist.

```

    }
}
result.setText(cur.name+" endzustand: "+cur.endzustand);

```

Im Ausgabefeld steht der letzte Zustand und die Information, ob er Endzustand ist.

```

}
public boolean handleEvent(Event evt){
    if(evt.id==Event.WINDOW_DESTROY)System.exit(0);
    if(evt.id==Event.LIST_SELECT){
        if(evt.target==zustaende){

```

Der User selektierte einen neuen Zustand aus der Liste.

```

        currentzust=zust[zustaende.getSelectedIndex()];

```

currentzust ist ein Zeiger auf den Zustand, dessen Regeln in der zweiten Liste angezeigt werden.

```

        regeln.removeAll();

```

Diese Zeile löscht alle Regeln der zweiten Liste.

```

        for(int i=0;i<currentzust.regeln.length;i++){

```

```

            regeln.addItem(currentzust.regeln[i].input+";"+currentzust.regeln[i].output+";"+currentzust.regeln[i].folg
            ezustand+";");
        }

```

Die Regeln des Zustands stehen nach dieser for-Schleife in der zweiten Liste als Tripel "input;output;folgezustand;".

```

    }
}
if(evt.id==Event.ACTION_EVENT){
    if(evt.target==addzustand){
        String zname=neuzustand.getText();
        int s=startz.getSelectedIndex();
        int e=endz.getSelectedIndex();

```

Der Name des neuen Zustands steht im TextField neuzustand und die Choice-Objekte startz und endz geben an, ob der Zustand Start- oder Endzustand ist.

```

        boolean sb=false;

```

```

boolean eb=false;
if(s==0)sb=true;
if(e==0)eb=true;
Zustand nz=new Zustand(zname,sb,eb);
addZust(nz);

```

Man verlängert die Liste um den neuen Zustand mit seinen drei Parametern.

```

currentzust=nz;
regeln.removeAll();
zustaende.addItem(zname);
}
else if(evt.target==addregel){
int select=zustaende.getSelectedIndex();
if(select!=-1)currentzust=zust[select];
Regel r=new Regel(input.getText(),output.getText(),folgezustand.getText());
currentzust.addRegel(r);

```

Die Klasse Zustand besitzt eine Methode zum Erweitern der Regelmenge.

```

regeln.addItem(r.input+" "+r.output+" "+r.folgezustand+"");

```

Regeln sind eindeutig durch (inputzeichen, output, folgezustand) definiert. Diese Methode besorgt das Einfügen in die Datenstruktur.

```

}
else if(evt.target==start){
runProg();

```

Ein Klick auf den Startbutton ruft den Interpreter auf, der den Automateninput verarbeitet.

```

}
}
return false;
}
public void addZust(Zustand z){
zlen++;
Zustand[]z2=new Zustand[zlen];
for(int i=0;i<zlen-1;i++){
z2[i]=zust[i];
}
z2[zlen-1]=z;
zust=z2;
}
}

```

Diese Art von Methoden tauchte bereits in vielen Beispielen auf.

```

}
class Zustand{
String name;
Regel[]regeln;
int rlen;
boolean endzustand;
boolean startzustand;

```

```

public Zustand(String name,boolean start,boolean end){
    this.name=name;
    endzustand=end;
    startzustand=start;
    regeln=new Regel[0];

```

Ein Zustand hat einen Namen und zwei Flags, die angeben, ob der Zustand Start- oder Endzustand ist. Ausserdem ist Speicherplatz für Verweise auf Regeln reserviert.

```

    }
    public void addRegel(Regel r){
        rlen++;
        Regel[]regeln2=new Regel[rlen];
        for(int i=0;i<rlen-1;i++){
            regeln2[i]=regeln[i];
        }
        regeln2[rlen-1]=r;
        regeln=regeln2;
    }
}
class Regel{
    String input;
    String output;
    String folgezustand;

```

Dies ist eine Datenhaltungsklasse für die drei Attribute einer Regel.

```

    public Regel(String i,String o,String f){
        input=i;
        output=o;
        folgezustand=f;
    }
}

```

Das folgende Programm ist eine Implementation des Assemblers, für den am Beginn der Einführung einige Beispielprogramme aufgelistet wurden. Die Strategie ist, den Quellcode in eine Reihe von Token zu zerlegen (Code einer virtuellen Maschine) und diese dann auszuführen. Pro Zeile verwaltet dieser Assembler 10 Token, auf Seite des Interpreters sind aber höchstens drei Token (ein Befehl) pro Zeile erlaubt. Wir vereinbaren folgende Token:

| | |
|----|-------|
| 0 | ADD |
| 1 | SUB |
| 2 | MULT |
| 3 | DIV |
| 4 | LOAD |
| 5 | STORE |
| 6 | A |
| 7 | B |
| 8 | C |
| 9 | D |
| 10 | (A) |
| 11 | (B) |

```

12    (C)
13    (D)
14    constant (zB. 12, 34, 1023)
15    label (zB. :hallo1, :hallo2, :test;)
16    JMP
17    JMPA
18    END
19    JE

```

```

import java.awt.*;
import java.awt.event.*;
class Assembler extends Frame{
    TextArea program;
    TextArea output;
    Button button;

```

In die TextArea program gibt man das Programm ein, den Wert des Registers A gibt das Programm in die TextArea output aus, der Button startet das Programm.

```

Table t;
int[] register=new int[4];
int[] ram=new int[4096];

```

Die virtuelle Maschine hat vier Register und einen Hauptspeicher von 4096 int-Worten.

```

public Assembler(){
    super("Assembler-Interpreter");
    resize(400,440);
    setLayout(null);
    t=new Table();
    program=new TextArea();
    button=new Button("start");
    output=new TextArea();
    add(program);
    add(button);
    add(output);
    program.reshape(10,40,380,200);
    button.reshape(10,250,100,27);
    output.reshape(10,290,300,100);
    button.addActionListener(new ActionListener(this));

```

Diesmal wird der Start-Button mit dem neuen Event-Modell verwaltet. ActionListener implementiert einen ActionListener, der auf den Kontext des Fensters zugreifen kann.

```

    show();
}
public boolean handleEvent(Event evt){
    if(evt.id==Event.WINDOW_DESTROY)System.exit(0);

```

Dieser Teil zeigt, dass das alte und das neue Eventmodell gut miteinander zusammen arbeiten.

```

    return false;
}

```

```

public static void main(String[] args){
    new Assembler();
}
public void runProg(){
    CodeObject[] cob=compile();

```

Die methode compile() liefert ein Array vom Typ CodeObject, welches die zusammengehörigen Token enthält.

```

int pc=0;

```

Dies ist der "Program Counter, PC", der die Adresse des aktuell bearbeiteten Befehl (Codezeile) verwaltet.

```

boolean end=false;
int token2=0;
int token3=0;
int valu=0;
while(!end){
    CodeObject current=cob[pc];

```

Der aktuelle Behl wird geholt.

```

switch (current.tokens[0].number){

```

Je nachdem, welches Token das erste des Befehls ist, springt das Programm zu einer der Case-Fallbearbeitungen. Die switch-Anweisung ersetzt und beschleunigt in JAVA else- und else if-Anweisungen, weil das Sprungziel zur Compilezeit feststeht (Form einer switch-Anweisung: switch(int a){case 0: anweisung; break; case 1; anweisung; break}).

```

case 0:

```

Dies ist der Fall current.tokens[0].number==0.

```

token2=current.tokens[1].number;
token3=current.tokens[2].number;

```

Mit dem Befehlstoken 0 wird addiert. token2 und token3 sind die Operanden.

```

valu=current.tokens[2].value;

```

Falls das zweite Token eine Konstante ist, erhält valu deren Wert.

```

if(token3==14){
    register[token2-6]+=valu;

```

Die Konstante wird dann dem ersten Openrandenregister zugefügt.

```

}
else if(token2>=6&&token3>=6&&token2<=9&&token3<=9){
    register[token2-6]=register[token2-6]+register[token3-6];

```

Sind beide Operanden Register, schreibt der Interpreter die Summe in das erste.

```

}

```

```

else if(token2>=6&&token3>=10&&token2<=9&&token3<=13){
    register[token2-6]=register[token2-6]+ram[register[token3-10]];

```

Ist das zweite Register indirekt adressiert, findet ein Hauptspeicherzugriff statt und der Inhalt der Zelle wird zum Inhalt des ersten Registers summiert.

```

}
pc++;

```

Nach dem Addieren geht der Interpreter zum nächsten Befehl über.

```

break;
case 1:

```

Die Fälle der Token 1 bis 3 (SUB, MULT, DIV) sind völlig analog.

```

token2=current.tokens[1].number;
token3=current.tokens[2].number;
valu=current.tokens[2].value;
if(token3==14){
    register[token2-6]-=valu;
}
else if(token2>=6&&token3>=6&&token2<=9&&token3<=9){
    register[token2-6]=register[token2-6]-register[token3-6];
}
else if(token2>=6&&token3>=10&&token2<=9&&token3<=13){
    register[token2-6]=register[token2-6]-ram[register[token3-10]];
}
pc++;
break;
case 2:
token2=current.tokens[1].number;
token3=current.tokens[2].number;
valu=current.tokens[2].value;
if(token3==14){
    register[token2-6]*=valu;
}
else if(token2>=6&&token3>=6&&token2<=9&&token3<=9){
    register[token2-6]=register[token2-6]*register[token3-6];
}
else if(token2>=6&&token3>=10&&token2<=9&&token3<=13){
    register[token2-6]=register[token2-6]*ram[register[token3-10]];
}
pc++;
break;
case 3:
token2=current.tokens[1].number;
token3=current.tokens[2].number;
valu=current.tokens[2].value;
if(token3==14){
    register[token2-6]/=valu;
}
else if(token2>=6&&token3>=6&&token2<=9&&token3<=9){
    register[token2-6]=register[token2-6]/register[token3-6];
}

```

```

}
else if(token2>=6&&token3>=10&&token2<=9&&token3<=13){
    register[token2-6]=register[token2-6]/ram[register[token3-10]];
}
pc++;
break;
case 4:
token2=current.tokens[1].number;
token3=current.tokens[2].number;
valu=current.tokens[2].value;

```

Der Befehl LOAD hat zwei Operanden, wobei der zweite eine Konstante sein kann.

```

if(token3==14){
    register[token2-6]=valu;
}

```

Dann wird dem ersten Operanden, einem Register, der Wert der Konstanten zugewiesen.

```

}
else if(token2>=6&&token3>=6&&token2<=9&&token3<=9){
    register[token2-6]=register[token3-6];
}

```

Wenn beide Operanden Register sind, kopiert das Programm den Inhalt des zweiten in den ersten.

```

}
else if(token2>=6&&token3>=10&&token2<=9&&token3<=13){
    register[token2-6]=ram[register[token3-10]];
}

```

Der wichtigste Fall ist, wenn der Interpreter den Inhalt einer Hauptspeicherzelle, die durch den zweiten Operanden (ein Register) indirekt adressiert ist, in das erste übergebene Register schreibt.

```

}
pc++;
break;
case 5:
token3=current.tokens[1].number;
token2=current.tokens[2].number;
valu=current.tokens[2].value;

```

Der erste Operand ist entweder eine Konstante oder eine Registerbezeichnung. Der zweite Operand ist ein Register, in dem eine Hauptspeicheradresse steht.

```

if(token3==14){
    ram[register[token2-10]]=valu;
}

```

Man speichert hier die Konstante in einer Hauptspeicherzelle.

```

}
else if(token2>=10&&token3>=6&&token3<=10&&token2<=13){
    ram[register[token2-10]]=register[token3-6];
}

```

Der Inhalt der indirekt adressierten Adresse wird mit dem Wert des Registers überschrieben.

```

}

```

```
pc++;
break;
case 15:
pc++;
```

Label werden im Bytecode einfach übergangen.

```
break;
case 16:
pc=t.getLine(current.tokens[0].values);
```

Findet ein unbedingter Sprung statt, wird die Adresse des Sprungziels, von dem zunächst nur der Name bekannt ist, in der Tabelle t nachgeguckt und dem Programcounter zugewiesen. Die Tabelle verwaltet Bezeichnungen und Adressen von Sprungmarken.

```
break;
case 17:
pc=register[0];
break;
```

Mit JMPA springt das Programm zu der Adresse mit dem Wert des Registers A.

```
case 18:
end=true;
break;
case 19:
if(register[0]==0)pc=t.getLine(current.tokens[0].values);
else pc++;
```

Wenn das Register A=0 ist, springt das Programm zur Sprungmarke im nächsten Token, sonst läuft das Programm einfach weiter.

```
break;
}
}
output.appendText("\n"+register[0]);
```

Nach Ende des Programmablaufs gibt der Interpreter den Wert des Registers A im Outputfenster aus.

```
}
public CodeObject[] compile(){
String[]proglines=getLines();
```

Im Compilervorgang wird das Programm in durch Semikolon getrennte Befehlszeilen bestehend aus Befehl und Argumenten zerlegt.

```
int n=proglines.length;
```

n ist die Programmlänge in Befehlen.

```
CodeObject[] cob=new CodeObject[n];
for(int i=0;i<n;i++){
cob[i]=getCodes(proglines[i],i);
```

Jede Zeile wird in eine Folge von Token zerlegt, die in der CodeObject Datenstruktur überwacht werden.

```
    }
    for(int i=0;i<n;i++){
        if(cob[i].tokens[0].number==15){
            cob[i].tokens[0].addToTable();
        }
    }
```

Der Compiler erzeugt einen Eintrag in der Sprungmarkentabelle, die zum Übersetzen von Sprungmarken in absolute Programmadressen dient.

```
    }
    for(int i=0;i<n;i++){
        if(cob[i].tokens[0].number==16){
            cob[i].tokens[0].value=t.getLine(cob[i].tokens[0].values);
        }
    }
```

Die Werte der Sprungtabelle trägt der Compiler in einer zusätzlichen Schleife an den notwendigen Stellen ein.

```
    }
    }
    return cob;
}
public String[] getLines(){
    String[]ret=null;
    String text=program.getText();
    int l=text.length();
    int arraylen=0;
    for(int i=0;i<l;i++){
        if(text.substring(i,i+1).equals(";"))arraylen++;
    }
}
```

Die Anzahl der Semikolonen entspricht der Anzahl der Befehle im Programm-Eingabetextfeld.

```
ret=new String[arraylen];
int pos=0;
for(int i=0;i<arraylen;i++){
    ret[i]="";
    while(!text.substring(pos,pos+1).equals(";")){
        ret[i]=ret[i]+text.substring(pos,pos+1);
        pos++;
    }
}
```

Die einzelnen Befehle werden voneinander getrennt und im Array ret gespeichert.

```
    }
    ret[i]=ret[i]+" ";
    pos++;
    if(text.substring(pos,pos+1).equals("\n"))pos++;
}
```

Der Scanner übergeht Return-Zeichen einfach.

```
    }
    return ret;
}
```

```

public CodeObject getCodes(String codeline,int line){
    CodeObject co=new CodeObject(codeline,line,t);
    co.transform();

```

Die Transformation von Zeilen in Tokenfolgen führt die Methode transform() im Objekttyp CodeObject durch. Sie ist als endlicher Automat mit vielen Zuständen organisiert.

```

        return co;
    }
}
class CodeObject{
    String codeline;
    int pos;
    int state;
    Token[] tokens=new Token[10];

```

Jede Zeile kann maximal 10 Token fassen, in dieser Version sind jedoch lediglich 3 erlaubt.

```

    int tokenpos;
    int line;
    Table t;
    public CodeObject(String codeline,int line,Table t){
        this.codeline=codeline;
        this.line=line;
        this.t=t;

```

Wichtige Objekte zur Verarbeitung sind der zu verarbeitende String, die zugehörige Zeilennummer und die Sprungmarkentabelle.

```

    }
    public void transform(){
        int constant=0;
        String lab="";
        while(pos<codeline.length()){

```

Zeichen für Zeichen scannt der Automat die Zeile und erzeugt aus Endzuständen Token für die Liste.

```

        String chara=codeline.substring(pos,pos+1);
        switch(state){
            case 0:
//Startzustand
//Es wird der Weg zum Token von ADD kommentiert, die anderen Fälle sind analog.
                if(chara.equals("A")){
                    state=1;
                }
                if(chara.equals("S")){
                    state=4;
                }
                if(chara.equals("M")){
                    state=7;
                }
                if(chara.equals("D")){
                    state=11;
                }

```

```

    if(chara.equals("L")){
        state=14;
    }
    if(chara.equals("Q")){
        state=18;
    }
    if(chara.equals("B")){
        state=23;
    }
    if(chara.equals("C")){
        state=24;
    }
    if(chara.equals("D")){
        state=11;
    }
    if(chara.equals("J")){
        state=37;
    }
    if(chara.equals("(")){
        state=26;
    }
}

```

```

if(chara.equals("0")||chara.equals("1")||chara.equals("2")||chara.equals("3")||chara.equals("4")||chara.equals
("5")||chara.equals("6")||chara.equals("7")||chara.equals("8")||chara.equals("9")){
    constant=new Integer(chara).intValue();
    state=35;
}
if(chara.equals(":")){
    state=36;
}
if(chara.equals("E")){
    state=42;
}
break;
case 1:
//Zeichen A eingegeben
if(chara.equals("D")){
    state=2;
}
if(chara.equals(" ")){
//kein weiteres Zeichen
    state=0;
    tokens[tokenpos]=new Token(6,0,null,null);
    tokenpos++;
//Token ist nur Registerbezeichnung für A
}
break;
case 2:
//AD eingegeben
if(chara.equals("D")){
    state=3;
}
break;
case 3:

```

```

//ADD eingegeben, Zustand verharrt.
    if(chara.equals(" ")) {
//"ADD " eingegeben, neues Token in Liste
        state=0;
        tokens[tokenpos]=new Token(0,0,null,null);
        tokenpos++;
    }
    break;
case 4:
    if(chara.equals("U")) {
        state=5;
    }
    if(chara.equals("T")) {
        state=19;
    }
    break;
case 5:
    if(chara.equals("B")) {
        state=6;
    }
    break;
case 6:
    if(chara.equals(" ")) {
        state=0;
        tokens[tokenpos]=new Token(1,0,null,null);
        tokenpos++;
    }
    break;
case 7:
    if(chara.equals("U")) {
        state=8;
    }
    break;
case 8:
    if(chara.equals("L")) {
        state=9;
    }
    break;
case 9:
    if(chara.equals("T")) {
        state=10;
    }
    break;
case 10:
    if(chara.equals(" ")) {
        state=0;
        tokens[tokenpos]=new Token(2,0,null,null);
        tokenpos++;
    }
    break;
case 11:
    if(chara.equals("I")) {
        state=12;
    }
}

```

```
if(chara.equals(" ")){\n    state=0;\n    tokens[tokenpos]=new Token(9,0,null,null);\n    tokenpos++;\n}\nbreak;\ncase 12:\nif(chara.equals("V")){\n    state=13;\n}\nbreak;\ncase 13:\nif(chara.equals(" ")){\n    state=0;\n    tokens[tokenpos]=new Token(3,0,null,null);\n    tokenpos++;\n}\nbreak;\ncase 14:\nif(chara.equals("O")){\n    state=15;\n}\nbreak;\ncase 15:\nif(chara.equals("A")){\n    state=16;\n}\nbreak;\ncase 16:\nif(chara.equals("D")){\n    state=17;\n}\nbreak;\ncase 17:\nif(chara.equals(" ")){\n    state=0;\n    tokens[tokenpos]=new Token(4,0,null,null);\n    tokenpos++;\n}\nbreak;\ncase 18:\nif(chara.equals(" ")){\n    state=0;\n    tokens[tokenpos]=new Token(18,0,null,null);\n    tokenpos++;\n}\nbreak;\ncase 19:\nif(chara.equals("O")){\n    state=20;\n}\nbreak;\ncase 20:\nif(chara.equals("R")){\n
```

```
    state=21;
}
break;
case 21:
if(chara.equals("E")){
    state=22;
}
break;
case 22:
if(chara.equals(" ")) {
    state=0;
    tokens[tokenpos]=new Token(5,0,null,null);
    tokenpos++;
}
break;
case 23:
if(chara.equals(" ")) {
    state=0;
    tokens[tokenpos]=new Token(7,0,null,null);
    tokenpos++;
}
break;
case 24:
if(chara.equals(" ")) {
    state=0;
    tokens[tokenpos]=new Token(8,0,null,null);
    tokenpos++;
}
break;
case 25:
if(chara.equals(" ")) {
    state=45;
}
break;
case 26:
if(chara.equals("A")) {
    state=27;
}
if(chara.equals("B")) {
    state=28;
}
if(chara.equals("C")) {
    state=29;
}
if(chara.equals("D")) {
    state=30;
}
break;
case 27:
if(chara.equals("")) {
    state=31;
}
break;
case 28:
```

```

    if(chara.equals("")){
        state=32;
    }
    break;
case 29:
    if(chara.equals("")){
        state=33;
    }
    break;
case 30:
    if(chara.equals("")){
        state=34;
    }
    break;
case 31:
    if(chara.equals(" ")){
        state=0;
        tokens[tokenpos]=new Token(10,0,null,null);
        tokenpos++;
    }
    break;
case 32:
    if(chara.equals(" ")){
        state=0;
        tokens[tokenpos]=new Token(11,0,null,null);
        tokenpos++;
    }
    break;
case 33:
    if(chara.equals(" ")){
        state=0;
        tokens[tokenpos]=new Token(12,0,null,null);
        tokenpos++;
    }
    break;
case 34:
    if(chara.equals(" ")){
        state=0;
        tokens[tokenpos]=new Token(13,0,null,null);
        tokenpos++;
    }
    break;
case 35:
    if(chara.equals(" ")){
        state=0;
        tokens[tokenpos]=new Token(14,constant,null,null);
        tokenpos++;
    }
}

```

```

if(chara.equals("0")||chara.equals("1")||chara.equals("2")||chara.equals("3")||chara.equals("4")||chara.equals
("5")||chara.equals("6")||chara.equals("7")||chara.equals("8")||chara.equals("9")){
    state=35;
    constant=constant*10;
    constant=constant+(new Integer(chara).intValue());
}

```

```
}
break;
case 36:
if(chara.equals(" ")) {
    state=0;
    tokens[tokenpos]=new Token(15,line,lab,t);
    tokenpos++;
}
else {
    state=36;
    lab=lab+chara;
}
break;
case 37:
if(chara.equals("M")) {
    state=38;
}
if(chara.equals("E")) {
    state=25;
}
break;
case 38:
if(chara.equals("P")) {
    state=39;
}
break;
case 39:
if(chara.equals(" ")) {
    state=40;
}
if(chara.equals("A")) {
    state=41;
}
break;
case 40:
if(chara.equals(" ")) {
    state=0;
    tokens[tokenpos]=new Token(16,line,lab,t);
    tokenpos++;
}
else lab=lab+chara;
break;
case 41:
if(chara.equals(" ")) {
    state=0;
    tokens[tokenpos]=new Token(17,0,null,null);
    tokenpos++;
}
break;
case 42:
if(chara.equals("N")) {
    state=43;
}
break;
```

```

        case 43:
        if(chara.equals("D")){
            state=44;
        }
        break;
        case 44:
        if(chara.equals(" ")){
            state=0;
            tokens[tokenpos]=new Token(18,0,null,null);
            tokenpos++;
        }
        break;
        case 45:
        if(chara.equals(" ")){
            state=0;
            tokens[tokenpos]=new Token(19,line,lab,t);
            tokenpos++;
        }
        else lab=lab+chara;
    }
    pos++;
}
}
}
}
class Token {
    int number;
    int value;
    String values;
    Table t;
    public Token(int number,int value,String values,Table t){
        this.number=number;
        this.value=value;
        this.values=values;
        this.t=t;
    }
}

```

Die Nummer dient zur Unterscheidung der Art, value und values sind Attribute zur Speicherung von Konstanten und Sprungmarkenbezeichnungen und solche müssen natürlich in einer gemeinsamen übergebenen Tabelle festgehalten werden.

```

    }
    public void addToTable(){
        t.add(values,value);
    }
}
class Table {
    int[]line=new int[0];
    String[]label=new String[0];
    int len=0;
    public void add(String label2,int line2){
        len++;
        int[]line3=new int[len];
        String[]label3=new String[len];
        for(int i=0;i<len-1;i++){
            line3[i]=line[i];
        }
    }
}

```

```

        label3[i]=label[i];
    }
    line3[len-1]=line2;
    label3[len-1]=label2;
    line=line3;
    label=label3;

```

Hier wurde wieder eine Standard-Arrayverlängerung implementiert.

```

    }
    public int getLine(String s){
        for(int i=0;i<len;i++){
            if(label[i].equals(s)){
                return line[i];
            }
        }
    }

```

Einen mit s übereinstimmenden Eintrag in der Liste der Sprungmarken sucht der Compiler und gibt die zugehörige Zeile zurück.

```

    }
    }
    return -1;
}
}
class ACListener implements ActionListener{
    Assembler ass;
    public ACListener(Assembler ass){
        this.ass=ass;
    }
    public void actionPerformed(ActionEvent e){
        ass.runProg();
    }
}

```

Man kann das neue EventModell erkennen, wo in speziellen Klassen ein Interface implementiert wird, das die Methode actionPerformed überschreibt. Mit dem Start-Button wird das Programm gescannt, kompiliert und interpretiert (runProg).

```

    }
}

```

```

import java.awt.*;
class Funktional extends Frame{
    TextArea program;
    Button button;
    TextArea output;
    TextField argm;
    VTable t;
}

```

Dies ist die Implementierung einer funktionalen Programmiersprache, die als der algorithmische Höhepunkt dieser Einführung gelten kann, weil sie an der tiefsten Stelle eine Verschachtelung von fünf for-Schleifen und einer while-Schleife aufweist. Der Quelltext wird in die TextArea geschrieben und mit dem Button ausgeführt, wobei die Funktion und ihre Argumente (Beispiel: F 12 34) im TextField argm

stehen müssen und das Ergebnis in der TextArea output ausgegeben wird. Einige Programmbeispiele gab es am Anfang dieser Einführung.

```
public Funktional(){
    super("Funktional");
    resize(400,440);
    setLayout(null);
    program=new TextArea();
    button=new Button("Start");
    output=new TextArea();
    argm=new TextField(30);
    add(program);
    add(button);
    add(output);
    add(argm);
    program.reshape(10,40,380,200);
    button.reshape(10,250,100,27);
    argm.reshape(120,250,200,27);
    output.reshape(10,290,360,100);
    t=new VTable();
    show();
}
public int runProg(String func,VTable t){
    Function[] progcode=compile();
```

Der Compiler liefert eine interpretierbare Datenstruktur als ein Function-Array.

```
FKAutomat fk=new FKAutomat(func);
```

Die Argumente werden im FKAutomat extrahiert. Dieser Automat wandelt im Prinzip durch Leerzeichen getrennte Wortfolgen in String-Arrays um.

```
fk.extract();
String[]anf=fk.getResult();
String cfunc=anf[0];
```

Erstes Element der Argumentfolge ist die Funktion selbst.

```
boolean end=false;
int n=progcode.length;
```

Die Anzahl der Funktionen im Programm ist mit n festgelegt.

```
int retv=0;
while(!end){
```

Auch dieses Programm läuft im Fetch-Decode-Execute-Zyklus in einer while-Schleife ab.

```
for(int i=0;i<n;i++){
    if(progcode[i].id.equals(cfunc)){
```

Die erste for-Schleife sucht nach einer identisch lautenden Funktion, wobei cfunc immer die Bezeichnung der momentan aufgerufenen Funktion ist.

```

Interval[] ivs=progrcode[i].ivs;
int ilen=ivs.length;
String[]args=progrcode[i].args;
boolean in=false;

```

Ist die Funktion herausgesucht, werden die zugehörigen Intervalle (Fallunterscheidungen) im Array ivs zwischengespeichert so wie die Argumentbezeichnungen der Funktion in args.

```

for(int j=0;j<ilen&&!in;j++){
    boolean isin=true;
    for(int k=1;k<args.length;k++){
        if(!ivs[j].isIn(new Integer(anf[k]).intValue(),args[k]))isin=false;
    }
    if(isin){

```

Die Argumente liegen genau dann im Intervall, wenn jedes Argument im passenden Teilintervall ist (siehe Intervall-Notation am Anfang der Einführung). Dies stellt die for-Schleife sicher.

```

        in=true;
        if(ivs[j].call!=null){

```

Wenn das Intervall einen CALL beherbergt, ruft der Interpreter eine andere Funktion auf.

```

            int slen=ivs[j].call.slen;
            Sub[]subs=ivs[j].call.subs;
            for(int k=0;k<slen;k++){

```

Diese for-Schleife geht alle in die Argumente substituierten Funktionen (subs) durch.

```

                String[] subk=subs[k].args;
                int sk=subk.length;
                String funcst=subk[1];
                Direct[]dire=new Direct[sk-2];
                String subcall=subk[1];
                for(int l=2;l<sk;l++){
                    dire[l-2]=new Direct(subk[l]);

```

Die Argumente der substituierten Funktionen dürfen wieder Ausdrücke sein und werden in dieser Schleife extrahiert und als direkte Ausdrücke behandelt.

```

                    for(int m=1;m<anf.length;m++){
                        dire[l-2].setVal(args[m],new Integer(anf[m]).intValue());

```

Die tiefste Schleife rechnet diese Ausdrücke aus und speichert die Argumente.

```

                    }
                    subcall=subcall+" "+dire[l-2].getValue();
                }
                int subval=runProg(subcall+" ",t.clone());

```

Der Wert der substituierten Funktion kann durch einen Selbstaufruf ermittelt werden.

```

                t.setValue(subk[0],subval);
                for(int l=0;l<args.length;l++){

```



```
if(evt.id==Event.ACTION_EVENT&&evt.target==button)output.appendText("\n"+runProg(argm.getText()+" ",t));
```

Mit einem Button startet man das Programm, zum Extrahieren wird dem Aufrufstring ein Leerzeichen angehängt und die erste Variablen-tabelle übergeben.

```
    return false;
}
public String[] getLines(){
    String[]ret=null;
    String text=program.getText();
    int l=text.length();
    int arraylen=0;
    for(int i=0;i<l;i++){
        if(text.substring(i,i+1).equals(";"))arraylen++;
    }
    ret=new String[arraylen];
    int pos=0;
    for(int i=0;i<arraylen;i++){
        ret[i]="";
        while(!text.substring(pos,pos+1).equals(";")){
            ret[i]=ret[i]+text.substring(pos,pos+1);
            pos++;
        }
        ret[i]=ret[i]+" ";
        pos++;
        if(text.substring(pos,pos+1).equals("\n"))pos++;
    }
    return ret;
}
```

Diese Methode ist aus dem Assembler übernommen (Befehlstrennung).

```
    }
    public String[] getIVs(String ivs){
        IVAutomat iva=new IVAutomat(ivs);
        iva.extract();
        return iva.getResult();
    }
    public Function[] compile(){
        String[] prgcode=getLines();
        Function[]ret=new Function[0];
        int fulen=0;
        Function current=null;
        Interval currentiv=null;
        Caller ccall=null;
        int n=prgcode.length;
        for(int i=0;i<n;i++){
            String line=prgcode[i];
            if(line.length()>=8){
                if(line.substring(0,8).equals("FUNCTION")){
                    String fline=line.substring(9);
                    fulen++;
                    current=new Function(t,fline);
                }
            }
        }
    }
}
```

```

Function[]temp=new Function[fulen];
for(int j=0;j<fulen-1;j++){
    temp[j]=ret[j];
}
temp[fulen-1]=current;
ret=temp;
}

```

Ein neues Funktionsobjekt findet seinen Platz in der Liste und erhält auch Variablen-tabelle und Argumentliste als String fline. Das Schlüsselwort ist FUNCTION am Beginn der Zeile.

```

}
if(line.length()>=8){
    if(line.substring(0,8).equals("INTERVAL")){
        String[] ivs=getIVs(line.substring(9));
        Interval iv=new Interval();
        for(int j=0;j<(ivs.length/5);j++){
            String vn=ivs[j*5];
            boolean l=false;
            boolean r=false;
            int ls=new Integer(ivs[j*5+1]).intValue();
            int rs=new Integer(ivs[j*5+2]).intValue();
            if(ivs[j*5+3].equals("true"))l=true;
            if(ivs[j*5+4].equals("true"))r=true;
            iv.addInterval(ls,rs,vn,l,r);
        }
        currentiv=iv;
        current.addInt(currentiv);
    }
}

```

Der aktuellen Funktion wird ein Fallunterscheidungstabelleneintrag angefügt. Die Trennung der Teilintervalle führt der IVAutomat durch ("getIVs(")).

```

}
}
if(line.length()>=4){
    if(line.substring(0,4).equals("CALL")){
        Caller call=new Caller(line.substring(5));
        call.extract();
        currentiv.addCall(call);
        ccall=call;
    }
}

```

Die Definition des CALL wird von der Klasse Caller verwaltet und dann einem Intervall zugeordnet.

```

}
}
if(line.length()>=6){
    if(line.substring(0,6).equals("DIRECT")){
        Direct direct=new Direct(line.substring(7));
        currentiv.addDirect(direct);
    }
}

```

Die Klasse Direct ist Informationshalter für Ausdrücke mit direkter Rückgabe.

```

}

```

```

    if(line.length()>=3){
        if(line.substring(0,3).equals("SUB")){
            Sub sub=new Sub(line.substring(4));
            sub.extract();
            ccall.addSub(sub);
        }
    }
}
return ret;
}
}

```

SUBs sind vergleichbar mit einem zweiten verschachtelten CALL (Behandlung ähnlich).

```

}
}
}
return ret;
}
}
class Function {
    Interval[] ivs;
    int ilen;
    VTable t;
    String id;
    String[] args;
    String arg;
    public Function(VTable t,String arg){
        this.t=t;
        this.arg=arg;
        FKAutomat fk=new FKAutomat(arg);
    }
}

```

Standardinstrument von Funktional ist der FKAutomat, der Strings in Teilstrings zerlegt.

```

        fk.extract();
        args=fk.getResult();
        id=args[0];
    }
    public void addInt(Interval ni){
        ilen++;
        Interval[] ivs2=new Interval[ilen];
        for(int i=0;i<ilen-1;i++){
            ivs2[i]=ivs[i];
        }
        ivs2[ilen-1]=ni;
        ivs=ivs2;
    }
    public void addVar(Variable v){
        t.addVariable(v);
    }
}
class Interval {
    int[]a;
    int[]b;
    String[]v;
    int len=0;
    Caller call;
    Direct direct;
    boolean[] leftinf;
    boolean[] rightinf;
}

```

```

public void addCall(Caller call){
    this.call=call;
}
public void addDirect(Direct direct){
    this.direct=direct;
}

```

Zu einem Intervall gibt es nur einen CALL oder DIRECT aber so viele Teilintervalle wie Argumente.

```

public void addInterval(int x,int y,String z,boolean l,boolean r){
    len++;
    int[]a2=new int[len];
    int[]b2=new int[len];
    boolean[]l2=new boolean[len];
    boolean[]r2=new boolean[len];
    String[]v2=new String[len];
    for(int i=0;i<len-1;i++){
        a2[i]=a[i];
        b2[i]=b[i];
        v2[i]=v[i];
        l2[i]=leftinf[i];
        r2[i]=rightinf[i];
    }
    a2[len-1]=x;
    b2[len-1]=y;
    v2[len-1]=z;
    l2[len-1]=l;
    r2[len-1]=r;
    a=a2;
    b=b2;
    v=v2;
    leftinf=l2;
    rightinf=r2;
}
public boolean isIn(int x,String y){
    for(int i=0;i<len;i++){
        if(v[i].equals(y)&&a[i]<=x&&b[i]>=x)return true;
        if(v[i].equals(y)&&rightinf[i]&&x>=a[i])return true;
        if(v[i].equals(y)&&leftinf[i]&&x<=b[i])return true;
    }
}

```

rightinf und leftinf bedeuten, dass das Teilintervall nach links oder rechts bis unendlich geht.

```

    }
    return false;
}
}
class Caller {
    String ausdruck;
    String[] args;
    Sub[] subs;
    int slen;
    public Caller(String ausdruck){
        this.ausdruck=ausdruck;
        subs=new Sub[0];
    }
}

```

```

    }
    public void addSub(Sub sub){
        slen++;
        Sub[] subs2=new Sub[slen];
        for(int i=0;i<slen-1;i++){
            subs2[i]=subs[i];
        }
        subs2[slen-1]=sub;
        subs=subs2;
    }
    public void extract(){
        FKAutomat fk=new FKAutomat(ausdruck);
        fk.extract();
        args=fk.getResult();
    }
}
class Sub{
    String ausdruck;
    String[] args;
    public Sub(String ausdruck){
        this.ausdruck=ausdruck;
    }
    public void extract(){
        FKAutomat fk=new FKAutomat(ausdruck);
        fk.extract();
        args=fk.getResult();
    }
}
class Direct{
    String start;
    Ausdruck aus;
    public Direct(String start){
        this.start=start;
        aus=getAusdruck(start);
    }
    public int getValue(){
        return aus.getValue();
    }
    public void setVal(String s,int val){
        aus.setVal(s,val);
    }
    public Ausdruck getAusdruck(String s){

```

Diese Methode wandelt geklammerte Ausdrücke in eine passende Datenstruktur um.

```

    int n=s.length();
    int klammern=0;
    for(int i=0;i<n;i++){
        if(s.substring(i,i+1).equals("("))klammern++;
        if(s.substring(i,i+1).equals(")")klammern--;
    }
    if(klammern>=4){
        int pos=1;
        int count=1;

```

```

int closed=0;
boolean isclosed=false;
for(int cursor=pos;cursor<n;cursor++){
    if(s.substring(cursor,cursor+1).equals("("))count++;
    if(s.substring(cursor,cursor+1).equals(")"))count--;
    if(!isclosed&&count==0){
        isclosed=true;
        closed=cursor;
    }
}
String half=s.substring(1,closed);

```

half ist der String zwischen der ersten Klammer und deren Schließung.

```

String conjunction=s.substring(closed+1,closed+2);
String zweit=s.substring(closed+2);
String first=zweit.substring(0,1);
if(first.equals("(")){
    int w=0;
    int leng=zweit.length();
    if(zweit.substring(leng-1,leng).equals(" ")w=1;
    zweit=zweit.substring(1,zweit.length()-1-w);
}

```

zweit ist der String zwischen der letzten Klammer und deren Öffnung.

```

if(conjunction.equals("+")){
    return new Sum(getAusdruck(half),getAusdruck(zweit));
}
else if(conjunction.equals("-")){
    return new Diff(getAusdruck(half),getAusdruck(zweit));
}
else if(conjunction.equals("*")){
    return new Mult(getAusdruck(half),getAusdruck(zweit));
}
else if(conjunction.equals("/")){
    return new Div(getAusdruck(half),getAusdruck(zweit));
}

```

Die beiden Teilausdrücke werden konjugiert und rekursiv berechnet.

```

}
else{

```

Wenn keine oder nur zwei Klammern im Ausdruck sind, dann ist der Ausdruck Konstante oder Variable. Syntaktisch ist "(x)+12" nicht erlaubt sondern nur "(x)+(12)", auf eine strikte Klammerung bei der Funktional-Programmierung ist zu achten.

```

if(klammern==0&& s.substring(n-1,n).equals(" "))s=s.substring(0,n-1);
if(s.substring(0,1).equals("(")){
    int bonus=0;
    if(s.substring(n-1,n).equals(" "))
        bonus=1;
}

```



```

    }
}
}
public int getValue(String vn){
    for(int i=0;i<len;i++){
        if(var[i].name.equals(vn))return var[i].value;
    }
    return 0;
}
}
interface Ausdruck{
    int getValue();
    void setVal(String s,int val);
    String getString();
}
class Sum implements Ausdruck{

```

Die Datenhaltungsklassen für Ausdrücke sind selbsterklärend und implementieren ein Interface zur rekursiven Verschachtelung.

```

    Ausdruck a;
    Ausdruck b;
    public Sum(Ausdruck a,Ausdruck b){
        this.a=a;
        this.b=b;
    }
    public int getValue(){
        return a.getValue()+b.getValue();
    }
    public String getString(){
        return a.getString()+" "+b.getString();
    }
    public void setVal(String s,int val){
        a.setVal(s,val);
        b.setVal(s,val);
    }
}
class Diff implements Ausdruck{
    Ausdruck a;
    Ausdruck b;
    public Diff(Ausdruck a,Ausdruck b){
        this.a=a;
        this.b=b;
    }
    public int getValue(){
        return a.getValue()-b.getValue();
    }
    public String getString(){
        return a.getString()+"-"+b.getString();
    }
    public void setVal(String s,int val){
        a.setVal(s,val);
        b.setVal(s,val);
    }
}

```

```

}
class Mult implements Ausdruck {
    Ausdruck a;
    Ausdruck b;
    public Mult(Ausdruck a,Ausdruck b){
        this.a=a;
        this.b=b;
    }
    public int getValue(){
        return a.getValue()*b.getValue();
    }
    public String getString(){
        return a.getString()+"*"+b.getString();
    }
    public void setVal(String s,int val){
        a.setVal(s,val);
        b.setVal(s,val);
    }
}
class Div implements Ausdruck {
    Ausdruck a;
    Ausdruck b;
    public Div(Ausdruck a,Ausdruck b){
        this.a=a;
        this.b=b;
    }
    public int getValue(){
        return a.getValue()/b.getValue();
    }
    public String getString(){
        return a.getString()+"/"+b.getString();
    }
    public void setVal(String s,int val){
        a.setVal(s,val);
        b.setVal(s,val);
    }
}
class Constant implements Ausdruck {
    int value=0;
    public Constant(int value){
        this.value=value;
    }
    public int getValue(){
        return value;
    }
    public String getString(){
        return ""+value;
    }
    public void setVal(String s,int val){
    }
}
class Variable implements Ausdruck {
    String name="";
    int value=0;
}

```

```

public Variable(String name,int value){
    this.name=name;
    this.value=value;
}
public Variable cloneo(){
    return new Variable(new String(name),value);
}
public String getString(){
    return name;
}
public int getValue(){
    return value;
}
public void setVal(String s,int val){
    if(name.equals(s))value=val;
}
}
class IVAutomat{
    String[]ret;
    String s;
    int pos;
    public IVAutomat(String s){

```

Der IVAutomat erkennt Intervallstrings und liest sie in eine String-Liste ein. Je fünf Indizes des String-Arrays stellen ein Teilintervall dar (Variable, linkes Ende, rechtes Ende, links-unendlich, rechts-unendlich).

```

        this.s=s;
        ret=new String[0];
    }
    public String[] getResult(){
        return ret;
    }
    public void extract(){
        int state=0;
        int n=s.length();
        String vname="";
        int a=0;
        int b=0;
        boolean l=false;
        boolean r=false;
        boolean lneg=false;
        boolean rneg=false;
        for(int i=0;i<n;i++){
            String chara=s.substring(i,i+1);
            switch (state){
                case 0:
                    if(!chara.equals(" ")){
                        vname=vname+chara;

```

Ein Nicht-Leerzeichen im Zustand 0 wird dem Variablenstring angehängt.

```

        }
        else state=1;

```

```

break;
case 1:
if(chara.equals("U")){
    l=true;
}
else if(chara.equals("-")){
    lneg=true;
}
else if(!chara.equals(".")){
    a=a*10;
    a=a+(new Integer(chara).intValue());
}
else state=2;

```

Das linke Intervallende ist eine Zahl (a), eine negative Zahl (a) oder minus unendlich.

```

break;
case 2:
if(chara.equals(".")){
    state=3;
}

```

Der Zwischenstring "." steht zwischen zwei Intervallgrenzen.

```

break;
case 3:
if(chara.equals("U")){
    r=true;
}
else if(chara.equals("-")){
    rneg=true;
}
else if(!chara.equals(" "))){
    b=b*10;
    b=b+(new Integer(chara).intValue());
}

```

Das rechte Intervallende ist unendlich oder eine ganze Zahl (b).

```

else{
    state=0;
    if(lneg)a=a*(-1);
    if(rneg)b=b*(-1);
    pos=pos+5;
    String[] ret2=new String[pos];
    for(int j=0;j<pos-5;j++){
        ret2[j]=ret[j];
    }
    ret2[pos-5]=new String(vname);
    ret2[pos-4]=""+a;
    ret2[pos-3]=""+b;
    ret2[pos-2]=""+l;
    ret2[pos-1]=""+r;
    l=false;
}

```

```

r=false;
ret=ret2;
vname="";
a=0;
b=0;

```

Sämtliche Informationen über das Intervall werden in das String-Array ret eingelesen und zu alten Informationen hinzugefügt. Bei unendlichen Grenzen ist eine Grenzzahl beliebig.

```

    }
  }
}
}
}
class FKAutomat{
  String[]ret;
  String s;
  int pos;
  public FKAutomat(String s){
    this.s=s;
    ret=new String[0];
  }
  public String[] getResult(){
    return ret;
  }
  public void extract(){
    int state=0;
    int n=s.length();
    String vname="";
    String arg="";
    int a=0;
    int b=0;
    for(int i=0;i<n;i++){

```

Der einfache Automat kennt zwei Zustände, in denen Strings verlängert und in ein Array gelesen werden, wobei dann der temporäre String neu mit der Länge 0 initialisiert wird.

```

String chara=s.substring(i,i+1);
switch (state){
  case 0:
    if(!chara.equals(" ")){
      vname=vname+chara;
    }
  else{
    pos=1;
    ret=new String[pos];
    ret[0]=vname;
    state=1;
  }
  break;
  case 1:
    if(!chara.equals(" ")){
      arg=arg+chara;
    }
}

```

```
else{
    state=1;
    pos++;
    String[] ret2=new String[pos];
    for(int j=0;j<pos-1;j++){
        ret2[j]=ret[j];
    }
    ret2[pos-1]=new String(arg);
    ret=ret2;
    arg="";
}
}
}
}
```