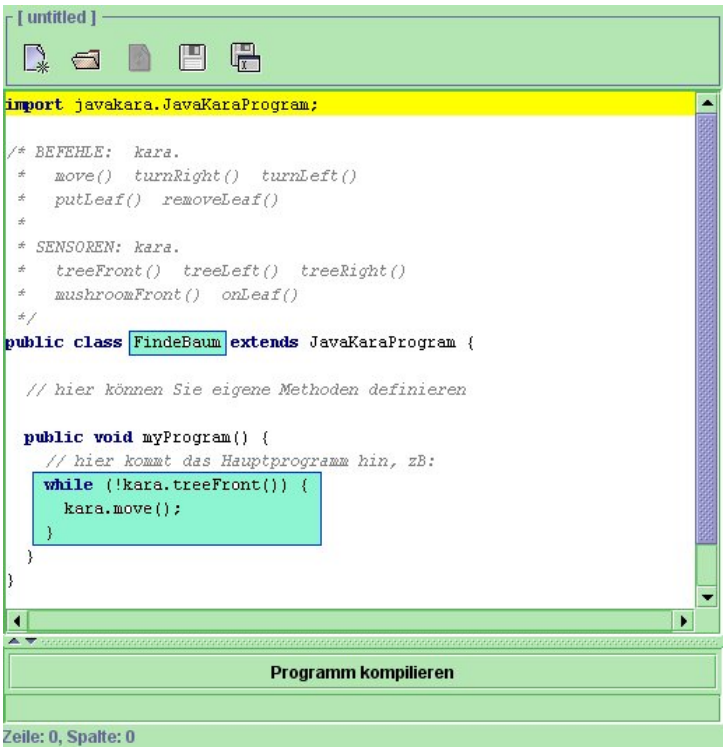
Einführung in JAVAKARA

Gerhard Bitsch

28. Juni 2008

1 Die JAVAKARA -Umgebung

Die Programmierumgebung JAVAKARA erlaubt es, Kara mit der Programmiersprache JAVA zu steuern. Dazu stellt JAVAKARA einen kleinen Programmeditor zur Verfügung, der beim Aufruf schon eine Programmschablone enthält.
Um Kara zu programmieren, muss der vorgegebene Name `FindeBaum` durch einen geeigneten Namen für das zu schreibende Programm ersetzt werden. Unter diesem Namen muss das Programm auch gespeichert werden.
Ebenso muss der im Bild markierte Teil von `myProgram()` durch eigenen Programmcode ersetzt werden.



Das im Bild gezeigte Programm `FindeBaum` zeigt schon einige Besonderheiten der Programmierung mit JAVA im Gegensatz zum Automatenmodell.

In der ersten Zeile wird eine Programmumgebung für KARA „importiert“, die dem Benutzer die Einzelheiten der Programmierung des Graphik-Interfaces (GUI) abnimmt. Mit ihr werden die für den Anfang benötigten Methoden bereitgestellt.

Diese Methoden sind in dem **Kommentar** unter dieser `import`-Anweisung aufgeführt. Der Kommentar beginnt mit `/*` und endet mit `*/`. Er kann sich über mehrere Zeilen erstrecken. Alternativ kann mit `//` der Rest einer Zeile als Kommentar markiert werden. Kommentare werden vom Programm während des Ablaufs ignoriert. Sie sollen lediglich die Verständlichkeit von Code verbessern.

Das Programm wird mit der Zeile

```
public class FindeBaum extends JavaKaraProgram {
```

eingeleitet. Die nähere Bedeutung der Schlüsselwörter `public, class` und `extends` wird später geklärt werden. Das mit dieser Deklaration begonnene Programm muss mit einer schließenden geschweiften Klammer } abgeschlossen werden.

Nach dieser Deklaration können die für das Programm benötigten Methoden definiert werden. Außer der Methode `myProgram` wird im Beispiel keine weitere Methode definiert. Die Methode `myProgram` **muss immer** definiert werden. Sie ist diejenige Methode, die beim Start des Programmes ausgeführt wird.

Das im Beispiel angezeigte Programm lasst KARA bis zum nächsten Baum laufen und dort stehen bleiben.

Das Bild zeigt ein Automatenprogramm mit der gleichen Wirkung. Die Anweisung, dass KARA , wenn er vor keinem Baum steht, einen Schritt machen soll und dann im Zustand `FindeBaum` bleiben soll, wird mit der `while`-Anweisung übertragen (while $\simeq$ solange). Der Schrittbefehl ist `kara.move()`.

Mit `kara.treeFront()` wird der Sensor abgefragt. Diese Abfrage liefert `true`, falls KARA vor einem Baum steht und `false` andernfalls. Das Ausrufezeichen ! steht für die Negation, d.h. es vertauscht die Werte `true` und `false`.

2 Einfache Anweisungen in JAVA

In diesem Abschnitt werden einfache JAVA -Anweisungen zur Strukturierung von Programmen vorgestellt.

Bedingungen: Beim Automatenmodell wird ein unterschiedliches Verhalten von KARA durch die Zustände der Sensoren gesteuert. In JAVA kann man diese Sensoren abfragen. Sie liefern dabei einen der Werte **true** oder **false** . Ausdrücke mit der Eigenschaft, einen dieser Werte zu liefern, nennt man **Bedingungen**.

Bedingungen können mit aussagenlogischen Operatoren verknüpft werden. Die folgende Tabelle gibt das Ergebnis einer solchen Verknüpfung für zwei Bedingungen a und b wieder:

a	b	(! a) nicht	(a && b) und	(a || b) oder
true	**true**	**false**	**true**	**true**
true	**false**	**false**	**false**	**true**
false	**true**	**true**	**false**	**true**
false	**false**	**true**	**false**	**false**

Beispiele:

`(! kara.onLeaf())` **true** , falls KARA auf einem leeren Feld steht.

`(kara.treeLeft() && kara.treeRight())` **true** , falls links und rechts von KARA ein Baum steht.

`(kara.treeFront() || kara.onLeaf())` **true** , falls *mindestens eine* der beiden Bedingungen zutrifft.

Blöcke: Man kann mehrere Anweisungen zu einer Anweisung (einem *Block*) zusammenfassen, indem man vor die erste Anweisung eine geöffnete geschweifte Klammer und nach der letzten Anweisung eine geschlossene geschweifte Klammer setzt.

while: Die `while`-Schleife, die schon im vorigen Abschnitt angesprochen wurde, hat folgende Gestalt:

`while (`*Bedingung*`) {`

Anweisung$_1$`;`

⋮ *Diese Anweisungen bilden einen Block*

Anweisung$_n$`;`

`}`

Bei der Ausführung einer `while`-Anweisung wird zunächst die (*immer in runden Klammern eingeschlossene*) Bedingung überprüft. Liefert sie den Wert **true** , so werden die (*immer durch ein Paar geschweifte Klammern eingeschlossenen*) Anweisungen (*jeweils durch einen Strichpunkt abgeschlossen*) nacheinander ausgeführt. Anschließend beginnt der Vorgang mit der Überprüfung der Bedingung neu. Dies wird so lange wiederholt, bis die Bedingung den Wert **false** liefert.

Verzweigung: Verzweigungen entstehen, wenn je nach dem Wert einer Bedingung unterschiedliche Anweisungen ausgeführt werden. Die allgemeine Form ist:

`if (`*Bedingung*`) {`

Anweisungen falls ***true***

`}`

`else {`

Anweisungen falls ***false***

`}`

Beispiel:

```
1  if (kara.onLeaf()) {    // falls Kara auf einem Blatt steht
2    kara.removeLeaf();    // Blatt entfernen
3  }
4  else {                  // andernfalls
5    kara.putLeaf();       // Blatt ablegen
6  }
```

Soll keine Anweisung ausgeführt werden, wenn die Bedingung nicht erfüllt ist, so kann man den `else`-Teil der Anweisung weglassen.

Beispiel:

```
1  if (kara.treeFront()) {  // falls Kara vor einem Baum steht
2    kara.turnLeft();       // nach links drehen
3  }                        // ansonsten nichts tun,
4                           // Programm fortsetzen
```

Methoden: Außer `myProgram()` kann man auch eigenen Methoden definieren, die dann wie die in der Umgebung vorhandenen Methoden verwendet werden können. Es gibt dafür mehrere Möglichkeiten, die einfachste führen wir jetzt ein.

```
private void Methodenname() {
    Anweisungen
}
```

Das Schlüsselwort `private` besagt, dass diese Methode nur innerhalb des gegenwärtigen Programms zur Verfügung stehen soll. `void` gibt an, dass die Methode kein Ergebnis zurück gibt (sie verändert die Welt oder den Zustand KARAS).

Beispiel: KARA soll zum nächsten Baum gehen, sich dort umdrehen und weiter laufen und dies endlos wiederholen.

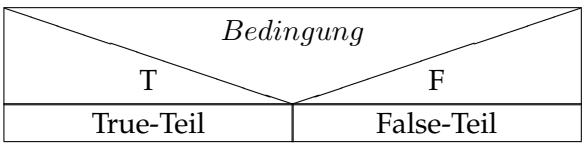
```
1  public class laufen extends JavaKaraProgram {
2
3    private void drehen() {          // Methode zur Drehung um 180°
4      kara.turnLeft();
5      kara.turnLeft();
6    }
7
8    public void myProgram() {
9      while (true) {                 // das erzeugt eine Endlosschleife
10       if (kara.treeFront()) {      // Wenn Kara vor einem Baum ist
11         drehen();                  // umdrehen
12       }
13       kara.move();                 // weiterlaufen
14     }
15   }
16 }
```

3 Struktogramme

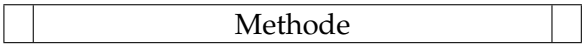
Struktogramme ermöglichen es, den Ablauf eines Programms *graphisch* zu entwerfen. Das ganze Programm wird in einem Rechteck eingetragen. Schleifen, Fallunterscheidungen und Methodenaufrufe (für selbstgeschriebene Methoden) werden wie folgt dargestellt:

- **while-Schleifen**

- **Fallunterscheidungen**



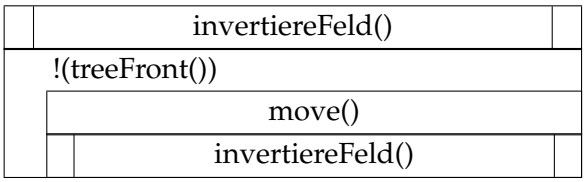
- **Methodenaufrufe**



Beispiel:

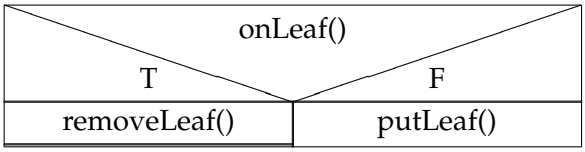
Hier ist ein einfaches Beispiel für ein **Struktogramm** zum Entwurf eines JAVAKARA -Programms. KARA soll zum nächsten Baum laufen und unterwegs alle Felder *invertieren*, d.h. auf Feldern mit Blatt das Blatt wegnehmen und auf Feldern ohne Blatt ein Blatt ablegen.

ZumBaum



In diesem Struktogramm wird eine Anweisung verwendet, die KARA nicht kennt: *invertiere-Feld()*. Diese Anweisung wird als Methode implementiert. Das zugehörige Struktogramm ist:

invertiereFeld()



Diese Struktogramme übersetzen sich in folgendes JAVAKARA -Programm:

```
1  public class zumBaum extends JavaKaraProgram {
2
3    private void invertiereFeld(){
4      if (kara.onLeaf()){
5        kara.removeLeaf();
6      }
7      else {
8        kara.putLeaf();
9      }
10   }
11
12   public void myProgram() {
13     invertiereFeld();
14     while (!kara.treeFront()){
15       kara.move();
16       invertiereFeld();
17     }
18   }
19 }
```

4 Variablen und Typen

Aufgabe: KARA soll zu einem Baum laufen, sich umdrehen und dann weiter laufen. Das soll er fünf mal wiederholen.

Im Automatenmodell wurden für diese Aufgabe fünf Zustände benötigt. Ändert man die Anzahl der Wiederholungen, so ändert sich die Anzahl der benötigten Zustände entsprechend. Es gibt im Automatenmodell keine einfache Möglichkeit, beliebig weit zu zählen.

In JAVAKARA kann man zählen, indem man **Variablen** verwendet. Variablen bieten eine Möglichkeit, Daten im Speicher des Rechners abzulegen und zu verändern. Da verschiedene Daten unterschiedlich viel Platz zur Speicherung beanspruchen, muss man (bei vielen Programmiersprachen, auch bei JAVA) im Programm vorher angeben, welchen *Typ* die zu speichernden Daten haben. Außerdem muss man sich den Speicherort merken, um auf die gespeicherten Daten zugreifen zu können. Dazu werden Variablen *deklariert*. In JAVA erfolgt das durch die Angabe des Datentyps, eines Namens und eventuell eines Anfangswertes:

```
int zähler = 0;
```

Diese Anweisung deklariert eine Variable `zähler`, in der ganze Zahlen (`int` $\simeq$ integer $\simeq$ ganze Zahl) gespeichert werden können und speichert 0 als Wert dieser Variablen.

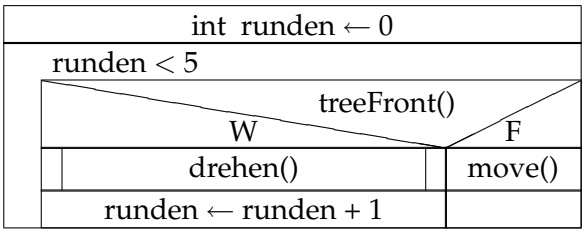
```
boolean gefunden;
```

Damit wird die Variable `gefunden` als eine Variable deklariert, die einen der Werte **true** oder **false** aufnehmen kann. Die Variable erhält keinen Anfangswert. (Der Typ `boolean` ist nach George Boole (1815 - 1864) benannt.)

In JAVA gibt es eine Vielzahl weiterer Datentypen (man kann auch eigene Typen erzeugen). Wenn man sich ein *Gedankenmodell* für Variable machen will, kann man sich folgendes vorstellen: mit einer Deklaration einer Variablen wird eine *Schachtel* (zum Typ) passender Größe angefertigt und mit einem Namensschild versehen. Ist ein Anfangswert angegeben, so wird dieser auf einen Zettel geschrieben und der Zettel in die Schachtel gelegt. Die Schachtel kommt in ein Lager, wo sie über den aufgeklebten Namen wieder gefunden werden kann.

Nun soll mit Hilfe einer Variablen das anfangs gestellte Problem gelöst werden. Zunächst ein Struktogramm:

FünfRunden



Die **Zuweisung** eines Wertes zu einer deklarierten Variablen symbolisieren wir im Struktogramm durch einen linksgerichteten Pfeil ←. Besonders interessant ist dabei die Zuweisung in der Fallunterscheidung. Sie zeigt an, dass der Variable `runden` ihr um 1 erhöhter alter Wert zugewiesen werden soll. Da dies jedesmal geschieht, wenn KARA vor einem Baum steht, zählt diese Variable die von KARA zurückgelegten Runden. Nach 5 Runden hat die Variable den Wert 5, die Schleifenbedingung ist also nicht mehr erfüllt und KARA bleibt stehen.

Im zugehörige JAVAKARA -Programm wird die Zuweisung von Werten an die Variable `runden` mit dem Gleichheitszeichen = geschrieben. Diese Verwendung entspricht natürlich nicht dem

Gebrauch von $=$ in der Mathematik, ist aber in vielen Programmiersprachen üblich. Für Vergleiche von Zahlen kann man mit `<`, `>`, `==` Bedingungen formulieren (das doppelte Gleichheitszeichen überprüft die Gleichheit zweier Zahlen).

```
1  public class FünfRunden extends JavaKaraProgram {
2
3    private void drehen() {
4      kara.turnLeft();
5      kara.turnLeft();
6    }
7
8    public void myProgram() {
9      int runden=0;
10     while (runden < 5) {
11       if (kara.treeFront()) {
12         drehen();
13         runden = runden + 1;      // keine Gleichung,
14       }                           // sondern eine Zuweisung
15       else {
16         kara.move();
17       }
18     }
19   }
20 }
```

Aufgaben

Fertigen Sie für die folgenden Aufgaben jeweils ein Struktogramm an, ehe Sie versuchen das Programm zu erstellen. verwenden Sie wo möglich eigene Methoden.

1. Bearbeiten Sie die Aufgabe Tunnelsucher II und die drei Aufgaben zur Kleeblattsuche im Wald.
2. Bearbeiten Sie die Aufgabe Pacman.
3. KARA soll ein 4×5-Rechteck mit Blättern belegen.
4. KARA soll ein Schachbrett (8×8) auslegen (weisse Felder leer, schwarze Felder mit Blatt).

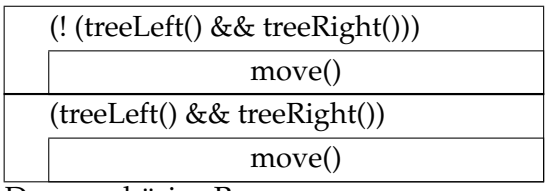
Lösungen

Aufgabe 1:

Tunnelsucher:

Den zwei benötigten Zuständen für dieses Problem im Automatenmodell entsprechen hier zwei `while`-Schleifen.

TunnelEnde



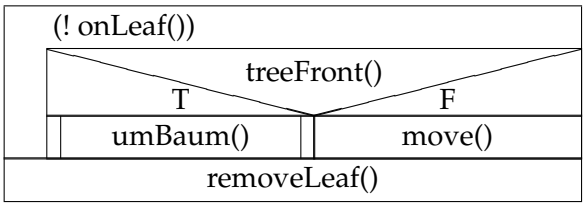
Das zugehörige Programm:

```
1  public class TunnelEnde extends JavaKaraProgram {
2      public void myProgram() {
3        while (!(kara.treeLeft() && kara.treeRight())) {
4            kara.move();            // zum Tunnelanfang
5        }
6        while (kara.treeLeft() && kara.treeRight()) {
7            kara.move();            // zum Tunnelende
8        }
9      }
10 }
```

Wald I

Das Struktogramm verwendet die Methode `umBaum()`, die noch definiert werden muss.

Wald I



Das zugehörige Programm:

```
1  public class Wald1 extends JavaKaraProgram {
2
3    private void umBaum() {
4      kara.turnLeft();
5      kara.move();
6      kara.turnRight();
7      kara.move();
8      kara.move();
9      kara.turnRight();
10     kara.move();
11     kara.turnLeft();
12   }
13
14   public void myProgram() {
15     while (!kara.onLeaf()) {
16      if (kara.treeFront()){
17        umBaum();
18       }
19      else {
20        kara.move();
21      }
22     }
23     kara.removeLeaf();
24   }
25 }
```

Wald II

Hier muss lediglich die Methode `umBaum()` angepasst werden. Der mittlere Teil mit den zwei `move()`-Befehlen muss ersetzt werden durch eine Schleife, um beliebig viele Bäume zu umgehen:

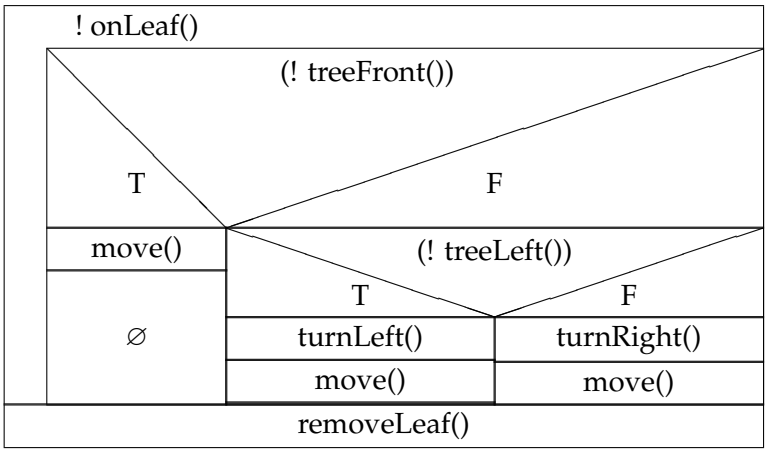
```
1   private void umBaum() {
2     kara.turnLeft();
3     kara.move();
4     kara.turnRight();
5     while (kara.treeRight()) {
6       kara.move();
7     }
8     kara.turnRight();
9     kara.move();
10    kara.turnLeft();
11  }
```

Wald III

Hier wird eine Unterscheidung von drei verschiedenen Fällen benötigt: Baum vorn, Baum links und Baum rechts. Man erledigt das mit einer *geschachtelten* Fallunterscheidung. Man prüft zunächst, ob vorn ein Baum steht. Ist dies nicht der Fall, prüft man, ob rechts oder links ein Baum steht.

Man erhält folgendes Struktogramm:

Wald3



Dies führt zu folgendem Programm. Die geschachtelte Fallunterscheidung wird aus dem Struktogramm so übertragen, dass innerhalb des `else`-Teils der ersten Fallunterscheidung eine zweite eingefügt wird. Dies lässt sich beliebig oft wiederholen.

```
1  public class Wald3 extends JavaKaraProgram {
2
3    public void myProgram() {
4      while (!kara.onLeaf()) {
5        if (!kara.treeFront()) {
6          kara.move();
7        }
8        else {
9          if (!kara.treeLeft()) {
```

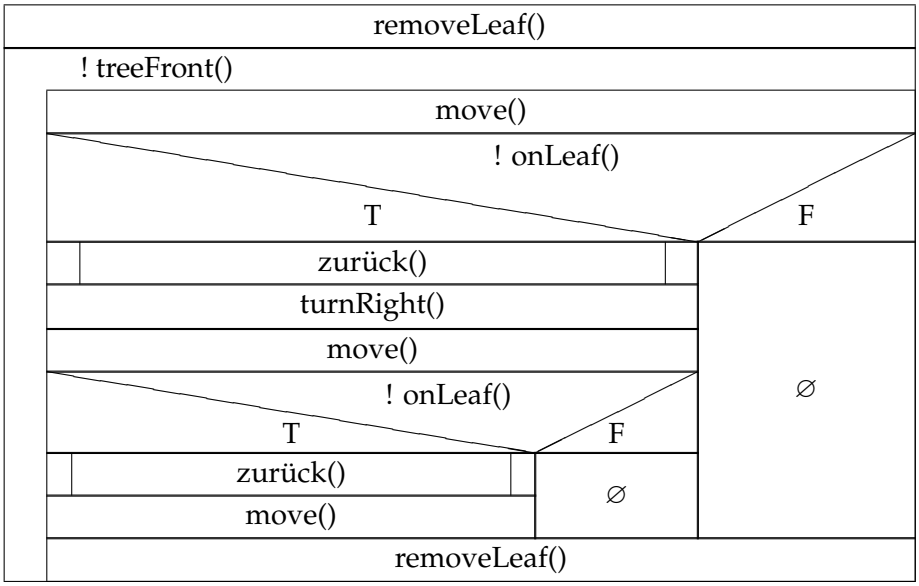
```
10              kara.turnLeft();
11              kara.move();
12            }
13            else {
14              kara.turnRight();
15              kara.move();
16            }
17          }
18        }
19        kara.removeLeaf();
20    }
21  }
```

Aufgabe 2:

Pacman

Dieses Programm benötigt eine Schleife und innerhalb der Schleife eine geschachtelte Fallunterscheidung. Im ersten Fall ist nichts zu tun, falls KARA auf einem Blatt ist, im anderen Fall muss KARA das nächste Blatt erst lokalisieren und sich auf das Feld mit dem Blatt begeben. Die Schleife hat also als Invariante die Bedingung, dass KARA auf einem Blatt stehen muss. Dieses wird am Ende jedes Schleifendurchgangs entfernt.

Pacman



Um KARA auf das nächste Blatt zu bewegen wird die Methode `zurück` verwendet. Das benötigte Struktogramm für die Methode `zurück` ist:

zurück

turnLeft()
turnLeft()
move()

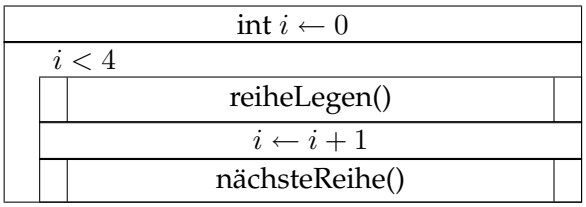
Daraus erhält man das folgende Programm:

```
1  public class Pacman extends JavaKaraProgram {
2
3    private void zurück() {
4      kara.turnLeft();
5      kara.turnLeft();
6      kara.move();
7    }
8
9    public void myProgram() {
10     kara.removeLeaf();              // Vorbedingung: Kara ist auf Kleeblatt
11     while (!kara.treeFront()) {     // solange nicht vor Baum,
12                                     // nächstes Kleeblatt suchen
13       kara.move();                  // Feld vorn
14       if (!kara.onLeaf()) {         // kein Kleeblatt vorn?
15         zurück();                   // dann links untersuchen
16         kara.turnRight();
17         kara.move();
18         if (!kara.onLeaf()) {       // auch kein Blatt links?
19           zurück();                 // dann muss es rechts sein!
20           kara.move();
21         }
22       }
23       kara.removeLeaf();            // Invariante: Kara ist auf Kleeblatt
24     }
25   }
26 }
```

Aufgabe 3:

Für das Rechteck erhält man zunächst eine Programmstruktur mit zwei ineinander geschachtelten Schleifen. Eine äußere Schleife ist für die aufeinanderfolgende Anordnung der Reihen des Rechtecks zuständig, in einer inneren Schleife wird jeweils eine einzelne Reihe ausgegeben.

Rechteck



Das Legen einer Reihe wird als Methode formuliert und mit einer Schleife realisiert:

reiheLegen()

int $j \leftarrow 0$
$j < 5$
putLeaf()
move()
$j \leftarrow j + 1$

nächsteReihe()

turnLeft()
turnLeft()
int $j \leftarrow 0$
$j < 5$
move()
$j \leftarrow j + 1$
turnLeft()
move()
turnLeft()

Der Wechsel zur nächsten Reihe ist ebenfalls als Methode realisiert. KARA geht zum Anfang der vorigen Reihe und wechselt dann in die zu legende neue Reihe. Das scheint umständlich, aber für abwechselnde Links- und Rechtsdurchgänge benötigt man jetzt noch nicht besprochene Hilfsmittel. Das folgende Programm zeichnet das gewünschte Rechteck. Dabei wird eine in JAVA erlaubte Abkürzung für den Befehl `i = i + 1` verwendet, nämlich `i++`.

```
1  public class Rechteck extends JavaKaraProgram {
2    private void reiheLegen(){
3      int j=0;
4      while (j<5) {
5        kara.putLeaf();
6        kara.move();
7        j++;
8      }
9    }
10   private void nächsteReihe() {
11     kara.turnLeft();
12     kara.turnLeft();
13     int j=0;
14     while (j<5) {
15       kara.move();
16       j++;
17     }
18     kara.turnLeft();
19     kara.move();
20     kara.turnLeft();
21   }
22   public void myProgram() {
23     int i=0;
24     while (i<4) {
25       reiheLegen();
26       i++;
27       nächsteReihe();
28     }
29   }
30 }
```

Aufgabe 4

Diese Aufgabe ist eine Variante des Rechteck-Problems. Allerdings müssen zwei aufeinander folgende Reihen um eins gegeneinander versetzt gelegt werden. Die folgenden Struktogramme beschreiben eine mögliche Lösung.

Schachbrett

int $i \leftarrow 0$	
$i < 4$	
	reiheLegen1()
	nächsteReihe()
	reiheLegen2()
	nächsteReihe()
	$i \leftarrow i + 1$

Die Methoden `reiheLegen1()` und `reiheLegen2` unterscheiden sich nur geringfügig.

reiheLegen1

int $j \leftarrow 0$
$(j < 4)$
putLeaf()
move()
move()
$j \leftarrow j + 1$

reiheLegen2

int $j \leftarrow 0$
$(j < 4)$
move()
putLeaf()
move()
$j \leftarrow j + 1$

Die Methode `nächsteReihe()` wird durch folgendes Struktogramm beschrieben:

nächsteReihe()

turnLeft()
turnLeft()
int $j \leftarrow 0$
$j < 8$
move()
$j \leftarrow j + 1$
turnLeft()
move()
turnLeft()

Das fertige Programm sieht dann so aus:

```
1  public class Schachbrett extends JavaKaraProgram {
2    private void reiheLegen1(){
3      int j=0;
4      while (j<4) {
5        kara.putLeaf();
6        kara.move();
7        kara.move();
8        j++;
9      }
10   }
```

```
11    private void reiheLegen2(){
12      int j=0;
13      while (j<4) {
14        kara.move();
15        kara.putLeaf();
16        kara.move();
17        j++;
18      }
19    }
20    private void nächsteReihe() {
21       kara.turnLeft();
22       kara.turnLeft();
23       int j=0;
24       while (j<8) {
25         kara.move();
26         j++;
27       }
28       kara.turnLeft();
29       kara.move();
30       kara.turnLeft();
31    }
32    public void myProgram() {
33      int i=0;
34      while (i<4) {
35        reiheLegen1();
36        nächsteReihe();
37        reiheLegen2();
38        nächsteReihe();
39        i++;
40      }
41    }
42  }
```

5 Methoden mit Parametern und Rückgabewerten, Eingaben und Ausgaben

Häufig will man einer Methode beim Aufruf Informationen übergeben, die den Ablauf der Methode beeinflussen können. Dazu muss die Methode entsprechend definiert werden.

Als erstes Beispiel soll eine Methode `move(int n)` vorgestellt werden, die KARA nicht nur ein Feld, sondern n Felder vorwärts bewegt.

```
1  private void move(int anzahl) {
2    int i=0;
3    while (i<anzahl) {
4      kara.move();
5      i++;
6    }
7  }
```

Im Unterschied zu den bisherigen Definitionen von Methoden steht hier in der Klammer hinter dem Methodennamen eine **Paramaterdeklaration**. Solche Deklarationen bestehen aus zwei Teilen:

- einem Typnamen (hier `int`), der angibt was für eine Art von Werten die Methode erwartet.
- einem Namen für den Parameter (hier `anzahl`). Unter diesem Namen kann auf den übergebenen Wert innerhalb der Methode zugegriffen werden.

Man kann in die Klammer auch mehrere Parameterdeklarationen schreiben. Diese werden dann durch Kommata voneinander getrennt. Beim Aufruf einer Methode mit Parametern müssen in die Klammern hinter dem Parameternamen Ausdrücke für die Parameter eingetragen werden (durch Kommata getrennt, falls es mehrere Parameter gibt). Im Beispiel würde etwa `move(5)` KARA veranlassen, fünf Felder nach vorn zu gehen. Den gleichen Effekt kann man auch mit `move(3+2)` erreichen. Rechenausdrücke sind bei der Parameterübergabe erlaubt.

Aufgaben:

1. Schreiben Sie eine Methode `turnLeft(int anzahl)` und eine Methode `turnRight(anzahl)`, die KARA veranlasst, sich so oft nach links bzw. rechts zu drehen, wie `anzahl` angibt. Programmieren Sie ein passendes Verhalten für negative Werte von `anzahl`.
2. Verändern Sie das Schachbrettprogramm (Seite 7) so, dass nur eine Methode zum Legen der Reihen benötigt wird und die Größe des Schachbretts variabel gehalten wird.

In vielen Fällen möchte man schon für `myProgram` zur Laufzeit Daten eingeben können. Dazu stellt JAVAKARA unter anderen (Seite 21) die Methode `intInput(String Titel)` zur Verfügung. Diese Methode wird nun verwendet, um die Rechteck-Klasse zu verallgemeinern. `intInput` benötigt als Parameter eine Eingabeaufforderung in Form eines `Strings` (Zeichenkette). Strings werden eingegeben, indem man den gewünschten Text in Anführungszeichen einschließt. Außerdem hat die Methode einen Rückgabewert vom Typ `int`. Man kann deshalb Anweisungen wie

```
int anzahl = tools.intInput("Geben Sie eine Anzahl an!")
```

schreiben. Dabei wird beim Lauf des Programms ein Fenster geöffnet und die dort eingegebene Zahl der Variablen `anzahl` zugewiesen.

```
1  public class Rechteck2 extends JavaKaraProgram {
2    private void reiheLegen(int breite){
3      int j=0;
4      while (j<breite) {
5        kara.putLeaf();
6        kara.move();
7        j++;
8      }
9    }
10   private void nächsteReihe(int breite) {
11     kara.turnLeft();
12     kara.turnLeft();
```

```
13        int j=0;
14        while (j<breite) {
15          kara.move();
16          j++;
17        }
18        kara.turnLeft();
19        kara.move();
20        kara.turnLeft();
21    }
22    public void myProgram() {
23      int hoehe = tools.intInput("Höhe des Rechtecks? ");
24      int breite = tools.intInput("Breite des Rechtecks? ");
25      int i=0;
26      while (i<hoehe) {
27        reiheLegen(breite);
28        i++;
29        nächsteReihe(breite);
30      }
31    }
32  }
```

KARA soll nun zu einem Baum laufen, die Anzahl der Schritte zählen, zu seinem Ausgangspunkt zurückkehren und die Anzahl der Schritte bis zum Baum ausgeben.

Für die Ausgabe verwenden wir die Methode (Seite 21) `tools.showMessage(String Text)`. Die Schritte werden in einer Variablen mitgezählt. Dies führt zu folgender Klasse.

```
1   public class Schritte extends JavaKaraProgram {
2     public void myProgram() {
3       int zaehler = 0;
4       while (!kara.treeFront()) {
5         kara.move();
6         zaehler++;
7       }
8       kara.turnLeft();
9       kara.turnLeft();
10      int j=0;
11      while (j<zaehler) {
12        kara.move();
13        j++;
14      }
15      kara.turnLeft();
16      kara.turnLeft();
17      tools.showMessage("Zum Baum sind es "+zaehler+" Schritte!");
18    }
19  }
```

Besondere Beachtung verdient die Art, wie hier der Text der Meldung zusammengestellt wird. In JAVA wird Text (Datentyp: *String*) durch Anführungszeichen begrenzt. Verschiedene Strings werden mit + zusammengesetzt. Werte von Zahlvariablen können ebenso in Text eingefügt werden. JAVA wandelt in diesem Fall Zahlen in ihre Textdarstellung um.

Will man Methoden schreiben, die Werte als Ergebnis liefern, so muss man den Typ des gelieferten Wertes in der Deklaration der Methode angeben. In der Methode muss dann mit `return` *Wert* ein Wert passenden Types zurückgegeben werden. Dazu ein einfaches Beispiel:

```
1 private boolean treeBack() {
2   kara.turnLeft();
3   boolean baum = kara.treeLeft();
4   kara.turnRight();
5   return baum;
6 }
```

Aufgaben

3. Schreiben Sie eine Methode `boolean mushroomLeft()` und eine Methode `boolean mushroomRight()` und testen Sie diese in geeigneter Umgebung.

6 Schleifen

Bisher sind Schleifen mit der `while`-Anweisung konstruiert worden:

```
1 while (Bedingung) {
2    Anweisung(en);
3 }
```

Der Anweisungsblock wird dabei so lange wiederholt ausgeführt, wie die Bedingung den Wert **true** liefert. Es kann also auch vorkommen, dass der Block gar nicht ausgeführt wird, weil die Bedingung schon zu Beginn nicht erfüllt ist. Man nennt solche Schleifen deshalb auch **abweisende** Schleifen oder Schleifen mit **Vorbedingung**.

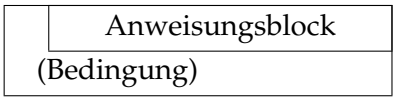
Mit der Anweisung:

```
1 do {
2   Anweisung(en);
3 } while (Bedingung);
```

definiert man eine Schleife, deren Anweisungsblock immer einmal ausgeführt wird. Danach wird erst die Bedingung für eine weitere Wiederholung geprüft. Die Anweisungen solcher Schleifen werden also mindestens einmal ausgeführt. Diese Schleife ist nicht abweisend, es ist eine Schleife mit **Nachbedingung**.

Im Struktogramm hat die `do-while`-Schleife folgende Form:

do-while-Schleife



Beispiele:

```
private void zumBaum () {
  do {
    kara.move();
  } while (!kara.treeFront());
}
```

KARA bewegt sich zum nächsten Baum in seiner Richtung und bleibt davor stehen. Man erhält eine Fehlermeldung, wenn sich KARA beim Methodenstart direkt vor einem Baum befindet.

```
public class DoBeispiel extends JavaKaraProgram {
  public void myProgram() {
   int eingabe;
   do {
     eingabe = tools.intInput("Eine Zahl zwischen 0 und 100?");
    } while(!((eingabe>0)&&(eingabe<100)));
   tools.showMessage("Ihre Zahl: "+eingabe);
  }
}
```

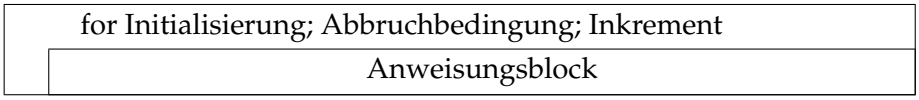
In diesem Fall macht die Verwendung einer `do-while`-Schleife Sinn, der Schleifenblock muss mindestens einmal durchlaufen werden, da der Benutzer mindestens eine Eingabe tätigen muss. Bei Fehleingaben wird die Eingabeaufforderung wiederholt.

Sowohl bei der `while`- als auch bei der `do-while`-Schleife ist die Anzahl der Wiederholungen nicht im voraus festgelegt. In vielen Fällen weiß man aber von vornherein, wie oft ein Anweisungsblock ausgeführt werden soll. Für solche Fälle gibt es die `for`-Schleife.

```
for (Initialisierung; Abbruchbedingung; Inkrement) {
    Anweisung(en);
}
```

Das zugehörige Struktogramm-Element hat folgende Form:

for-Schleife



Beispiele:

```
private void umdrehen() {
  for (int i=0; i<2; i++){
    kara.turnLeft();        //Kara dreht sich zweimal links
  }
}
```

```
private void move(int schritte){
  for (int i=0; i<schritte; i++){
    kara.move();
  }
}
```

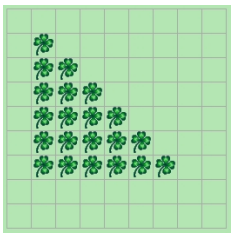
Das zweite Beispiel ist natürlich nicht sicher, wenn Bäume oder Pilze existieren, weil KARA dann möglicherweise nicht die angegebene Zahl von Schritten machen kann.

Aufgaben

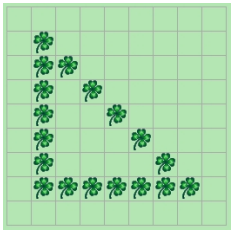
1. Schreiben Sie die Klasse `Rechteck2` (Seite 15) mit Hilfe von `for`-Schleifen um.
2. Fertigen Sie für das folgende Programm ein Struktogramm an und erläutern Sie seine Funktion.

```
1  public class WasIstDas extends JavaKaraProgram {
2    public void move(int i){
3      for (int j=0;j<i;j++){
4        kara.move();
5      }
6    }
7    private void reiheLegen(boolean rechts){
8      for (int j=0; j<4;j++){
9        kara.putLeaf();
10       move(2);
11     }
12     if (rechts){
13       kara.turnRight();
14       kara.move();
15       kara.turnRight();
16       kara.move();
17     }
18     else {
19       kara.turnLeft();
20       kara.move();
21       kara.turnLeft();
22       kara.move();
23     }
24    }
25   public void myProgram() {
26     for(int i=0;i<4;i++){
27       reiheLegen(true);
28       reiheLegen(false);
29     }
30   }
31 }
```

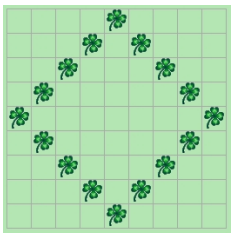
3. Schreiben Sie eine Klasse `Spirale`, die vom Benutzer die Anzahl der Strecken und die Anzahl der Blätter erfragt, die bei jeder Strecke im Vergleich zur vorigen mehr gelegt werden sollen und dann entsprechend Blätter legt.
4. Schreiben Sie eine Klasse `Dreieck` die vom Benutzer die Anzahl der gewünschten Reihen erfragt und dann ein entsprechendes (gefülltes) Dreieck aus Blättern legt.



5. Schreiben Sie eine Klasse `Dreieck1` die vom Benutzer die Anzahl der gewünschten Reihen erfragt und dann den Rand eines entsprechenden (gleichschenklig rechtwinkligen) Dreieck aus Blättern legt.



6. Schreiben Sie eine Klasse `Quadrat` die vom Benutzer die Anzahl der gewünschten Reihen erfragt und dann den Rand eines entsprechenden Quadrats aus Blättern legt.



7 Vordefinierte Methoden in JAVAKARA

In der Umgebung von JAVAKARA stehen unterschiedliche Methoden zur Verfügung. Man kann sich eine Kurzbeschreibung der Methoden mit der Hilfe-Schaltfläche des Welt-Fensters anzeigen lassen. Die Methoden sind in verschiedene Gruppen eingeteilt.

Zunächst sind einige der Methoden dazu da, KARA zu Aktionen zu veranlassen:

void move()	Ein Feld vorwärts
void turnLeft()	Vierteldrehung nach links
void turnRight()	Vierteldrehung nach rechts
void putLeaf()	Blatt ablegen
void removeLeaf()	Blatt aufnehmen

Die Sensoren werden über Methoden abgefragt, die ein Ergebnis vom Typ **boolean** – also **true** oder **false** – zurückgeben.

boolean treeFront()	Baum vorn?
boolean treeLeft()	Baum links?
boolean treeRight()	Baum rechts?
boolean onLeaf()	auf Blatt?
boolean mushroomFront()	vor Pilz?

Schließlich gibt es noch Methoden, KARA direkt auf einem bestimmten Feld zu positionieren und eine Methode, Karas Position abzufragen.

void setPosition (int x, int y)	setzt KARA auf das Feld mit den Koordinaten $(x; y)$
void setDirection(int d)	setzt KARAS Richtung: 0: Nord, 1: West, 2: Süd, 3: Ost
java.awt.Point getPosition()	liefert KARAS Position in Form eines **Point**-Objekts
double getX()	liefert die x-Komponente eines Point-Objekts
double getY()	liefert die y-Komponente eines Objekts

Alle diese Methoden (außer getX() und getY()) müssen mit dem Prefix `kara` aufgerufen werden, also z. B. nicht `move()`, sondern `kara.move()`. Leider gibt es keine Methode zum Ermitteln von KARAS Richtung.

Es gibt auch Methoden zur direkten Veränderung von KARAS Welt. Sie müssen mit dem Prefix `world` aufgerufen werden.

void clearAll()	Alles entfernen (auch KARA).
void setLeaf (int x, int y, boolean legen?)	legt oder entfernt ein Blatt bei (x;y)
void setTree (int x, int y, boolean legen?)	legt oder entfernt einen Baum bei (x;y)
void setMushroom (int x, int y, boolean legen?)	legt oder entfernt einen Pilz bei (x;y)
void setSize (int xMax, int yMax)	Neue Größe der Welt festlegen
int getSizeX()	horizontale Weltgröße zurückgeben
int getSizeY()	vertikale Weltgröße zurückgeben
boolean isEmpty(int x,int y)	**true** , falls Feld (x;y) leer ist
boolean isTree(int x,int y)	**true** , falls ein Baum auf Feld (x;y) ist
boolean isLeaf(int x,int y)	**true** , falls ein Blatt auf Feld (x;y) ist
boolean isMushroom(int x,int y)	**true** , falls ein Pilz auf Feld (x;y) ist

Die nun folgenden Methoden sind in der Klasse `tools` definiert, müssen also mit dem Prefix `tools` aufgerufen werden. Diese Methoden sind vor allem für Ein- und Ausgaben nützlich. Der Typ `string` ist für Zeichenketten (also Texte) bestimmt.

void showMessage(String text)	gibt `text` in einem Dialogfenster aus
String stringInput(String Titel)	fordert die Eingabe eines Strings in einem Dialogfenster. `Titel` liefert die Fensterüberschrift. Die Rückgabe ist die Konstante **null**, falls die Eingabe abgebrochen wird.
int intInput(String Titel)	Eingabe einer ganzen Zahl in einem Dialogfenster mit der Überschrift `Titel`. Das Ergebnis ist `Integer.MIN_VALUE`, wenn die Eingabe abgebrochen wird oder keine ganze Zahl eingegeben wird.
double doubleInput(String Titel)	Eingabe einer Dezimalzahl in einem Dialogfenster mit der Überschrift `Titel`. Das Ergebnis ist `Double.MIN_VALUE`, wenn die Eingabe abgebrochen wird oder keine Dezimalzahl eingeben wird.
int random(int Obergrenze)	liefert eine Zufallszahl aus dem Intervall $[0; Obergrenze]$
void sleep(int ms)	stoppt die Programmausführung für `ms` Millisekunden
void checkState()	überprüft die Bedienungsleiste.

Unter Verwendung dieser Methoden kann man viele interessante kleine Probleme behandeln.

Aufgaben

1. Erzeugen Sie eine 20×20-Welt, umranden Sie sie mit Bäumen und setzen Sie KARA auf das Feld $(10|10)$ mit Richtung Norden.

2. Schreiben Sie eine Klasse, die die Anzahl der Bäume einer Welt ermittelt und in einem Message-Fenster ausgibt.

   ANLEITUNG: Ermitteln Sie die Breite und Höhe der Welt und untersuchen Sie in einer doppelten `for`-Schleife die Felder der Welt darauf, ob auf ihnen ein Baum steht. KARA wird dazu nicht benötigt!

3. Schreiben Sie eine Klasse, die eine leere 15×15-Welt erzeugt und in ihr per Zufall 25 Bäume verteilt.

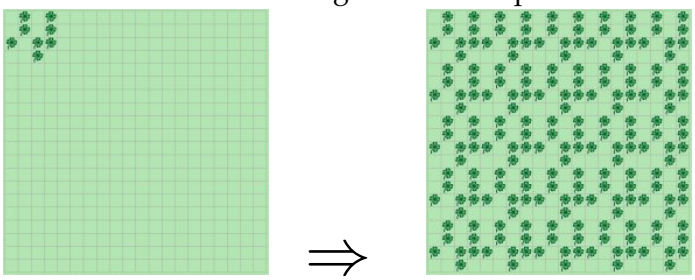
   HINWEIS: Verwenden sie die `world`- und `tools`-Funktionen. Erzeugen Sie zwei Zufallszahlen, um die Zeile und Spalte eines Feldes für einen Baum auszuwählen. Beachten Sie, dass kein Feld mehr als einmal ausgewählt werden darf.

4. Lassen Sie KARA auf einem Feld beliebiger Größe eine Zufallswanderung durchführen. Bei jedem Schritt geht KARA zufallsgesteuert ein Feld nach vorn oder nach links oder nach rechts und legt ein Blatt ab, falls dort keines liegt. Zählen Sie die Anzahl der Schritte, bis die Welt gefüllt ist.

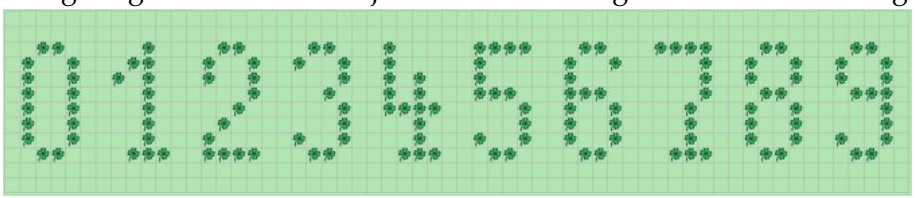
   HINWEIS: Verwenden Sie `int random(int n)`.

5. Ändern Sie das vorige Programm so ab, dass KARA bei belegten Feldern auf seinem Weg das Blatt aufnimmt. Lassen Sie KARA etwa so viele Schritte tun, wie in der vorigen Aufgabe zum Füllen der Welt benötigt wurden und prüfen Sie, wieviel der Welt dann mit Blättern bedeckt ist.

6. Man kann mit `world.setLeaf(x,y,true)` auf dem Feld $(x|y)$ ein Blatt ablegen. Man kann dies verwenden, um Muster zu erzeugen. Schreiben Sie eine Klasse, die eine 20×20-Welt erzeugt, im oberen linken Eck ein zufälliges 4×4-Muster aus Kleeblättern erzeugt und dieses Muster dann auf die ganze Welt kopiert.



7. Die angezeigten Ziffern sollen jeweils mit einer eigenen Methode erzeugt werden.



   Dabei gibt es folgende Vorgaben:

   - Jede Ziffer beansprucht ein Rechteck mit 6×8 Feldern.
   - In den seitlichen Rändern des Rechtecks liegen keine Blätter.
   - die Methode erhält als Eingabe die Koordinaten des linken unteren Eckfelds des Rechtecks.

   Die Eins kann man beispielsweise wie folgt erzeugen:

```
1  public void eins(int x,int y){
2    for (int i=2;i<5;i++){
3      world.setLeaf(x+i,y,true);
4    }
5    for (int i=1;i<8;i++){
6      world.setLeaf(x+3,y-i,true);
7    }
```

```
8         world.setLeaf(x+2,y-6,true);
9         world.setLeaf(x+1,y-5,true);
10    }
```

Analysieren Sie diese Methode und schreiben Sie Methoden für die restlichen Ziffern. Schreiben Sie damit eine Klasse, die das oben gezeigte Bild erzeugt.

HINWEIS: Es empfiehlt sich, eine allgemeine Methode

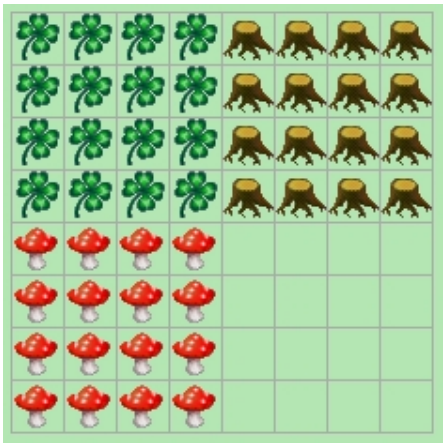
```
public void schreibe(int ziffer,int x,int y)
```

zu schreiben, die je nach Ziffer die passende Schreibmethode wählt. Dazu sollte man sich ansehen (Internet), wie man Fallunterscheidungen in JAVA mit `switch` realisiert.

8. Schreiben Sie unter Verwendung der vorigen Aufgabe eine Klasse, die eine natürliche Zahl vom Benutzer erfragt und diese Zahl dann in eine passende Welt schreibt.

   HINWEIS: Den Divisionsrest von a:b erhält man durch `a % b`. Im Bereich der ganzen Zahlen liefert JAVA immer den ganzzahligen Quotienten ohne Rest, wenn man `a/b` berechnet. anders ausgedrückt: Gilt $a = q \cdot b + r$ mit $r < b$, so ist $a/b = q$ und $a \,\%\, b = r$.

9. Von Manfred Eigen (`http://de.wikipedia.org/wiki/Manfred_Eigen`) stammt die folgende Simulation zum Thema Evolution.
   Man nehme ein 8×8-Feld und belege es in jedem der vier Teilquadrate mit einem Symbol, wie rechts gezeigt (das vierte „Symbol“ ist hier ganz einfach ein leeres Feld). Nun wird folgendes Verfahren wiederholt, bis nur noch eine Symbolsorte auf dem Feld ist: Man wählt per Zufall zwei verschiedene Felder des Quadrats. Das Symbol auf dem ersten Feld wird entfernt und durch ein Symbol von der Art des auf dem zweiten Feld befindlichen ersetzt.

   
   Startbelegung

   Schreiben Sie eine Klasse, die diese Simulation durchführt.

8 Arrays

Wir wollen den verwendeten Zufallszahlengenerator `tools.random(int n)` benutzen, um einen Würfel zu simulieren. Dazu sollen hundert Zufallsszahlen im Bereich von 1 bis 6 erzeugt werden. Wir wollen die Anzahlen festhalten, mit denen jede der Zahlen erzeugt wird.

Die Zufallszahlen werden mit folgender Methode erzeugt (`random(5)` erzeugt Zahlen von 0 bis 5):

```
1 private int würfel() {
2   return (1+tools.random(5));
3 }
```

Zum Mitzählen der einzelnen Ergebnisse könnte man jetzt sechs Variablen definieren und diese mit einer Fallunterscheidung bei jedem Ergebnis um eins erhöhen. Dies wäre äußerst umständlich. Da solche und ähnliche Probleme häufig auftauchen, sehen die meisten Programmiersprachen auch Datenstrukturen vor, die solche Aufgaben vereinfachen.

Will man mehrere Daten gleichen Typs gruppieren, verwendet man einen **Array** (deutsch: Feld). In einem Array werden eine feste Anzahl von Daten gleichen Typs gespeichert. Auf die einzelnen Daten kann man über einen *Index* zugreifen. In JAVA werden Arrays beispielsweise wie folgt vereinbart:

```
int[] zahlen={1,2,3,4,5,6};
```

Diese Anweisung deklariert eine Array-Variable `zahlen` für 6 Einträge vom Typ `int`, die mit den in den geschweiften Klammern stehenden Zahlen initialisiert werden. Auf das dritte Element des Arrays wird mit dem Ausdruck `zahlen[2]` zugegriffen (die Zählung der Array-Elemente beginnt mit 0).

Alternativ kann man ein Array auch ohne Initialisierung deklarieren und die Zuweisung von Werten später vornehmen. Mit der folgenden Methode wird die Anzahl der Elemente des Arrays festgelegt. Bei Grundtypen wie `int` wird dabei automatisch jedes Element mit einem Standardwert initialisiert. Für `int` ist dies 0.

```
int[] werte=new int[100];
```

erzeugt einen Array mit 100 Elementen vom Typ `int`, die alle mit 0 initialisiert werden. Das Programm könnte dann etwa so aussehen:

```
1 public void myProgram() {
2   int[] z=new int[7];
3   for (int j=0; j<100;j++) {
4     z[würfel()]++;
5   }
6 }
```

Der Array `z` wird mit 7 Elementen definiert, weil der Zufallszahlengenerator Werte von 1 bis 6 liefert, das erste Arrayelement aber immer den Index 0 hat. So kann man für jede Zahl i des Würfels ganz einfach das $i$te Element von `z` als Zähler verwenden. Jetzt fehlt allerdings noch eine Ausgabe des Ergebnisses. Diese führt man am besten in einer separaten Methode durch:

```
1 private void anzeige(int[] liste){
2   String erg="[";
3   for (int i=1;i<liste.length-1;i++){
4     erg =erg+liste[i]+",";
5   }
6   erg =erg+liste[liste.length-1]+"]";
7   tools.showMessage(erg);
8 }
```

Die Methode wird nach der `for`-Schleife in `myProgram` mit `anzeige[z]` aufgerufen. Sie liefert eine Ausgabe der Form `[23,13,15,16,16,17]`. Diese Methode ist nötig, weil JAVA den Array für eine Textausgabe nicht automatisch in einen String umwandelt, wie es das für einfache Zahlen tut.

Natürlich wird man das Verfahren mehrfach durchführen wollen. Dazu kann man in `myProgram` um die Würfel-Schleife eine weitere Schleife legen, die den Vorgang der 100 simulierten Würfe wiederholt. Dabei könnte man auch auf die Idee kommen, statt bei jedem Durchgang einen neuen Zähler `z` zu erzeugen, einen vordefinierten auf Null stehenden Zähler jeweils wieder an `z` zu übergeben. Die Klasse sieht dann so aus:

```
1  public class Zufallszahlen1 extends JavaKaraProgram {
2    private int[] z0={0,0,0,0,0,0,0};
3    // würfel und anzeige wie bisher...
4    public void myProgram() {
5      for (int i=0; i<5; i++){
6        int[] z=z0;
7        for (int j=0; j<100;j++) {
8          z[würfel()]++;
9        }
10       anzeige(z);
11     }
12   }
13 }
```

Damit wir die Simulation fünf mal wiederholt (Zeile 5). In Zeile 2 wird ein Zähler-Array `z0` definiert und auf Null gesetzt. Dieses Array wird dann bei jedem Durchgang `z` zugewiesen (Zeile 6). Der Effekt ist aber, dass jetzt bei jedem Versuch in der Anzeige *nicht die Ergebnisse des aktuellen Versuchs* erscheinen, sondern *die Summen aller bisherigen Versuche*.

Der Grund für dieses Verhalten liegt darin, dass durch die Anweisung in Zeile 6 nicht `z0` in einer Kopie an `z` übergeben wird, sondern dass `z` lediglich als weiterer Name des in Zeile 2 erzeugten Arrays etabliert wird. Sowohl `z` als auch `z0` greifen auf die gleichen Daten zu, die damit in der Würfelschleife verändert werden. Eine Rückstellung nach der Würfelschleife erfolgt nicht.

Generell gilt, dass bei Zuweisungen in JAVA *zusammengesetzte Daten nicht kopiert werden, sondern nur Verweise auf den Speicherort der Daten übergeben werden.*

Wir können nun die Klasse so erweitern, dass die Einzelergebnisse und die summierten Ergebnisse angezeigt werden:

```
1  public class Zufallszahlen1 extends JavaKaraProgram {
2    private int[] z0={0,0,0,0,0,0,0};
3    // würfel und anzeige wie bisher...
4    public void myProgram() {
5      for (int i=0; i<10; i++){
6        int[] z=new int[7];
7        for (int j=0; j<100;j++) {
8          int w=würfel();
9          z[w]++;
10         z0[w]++;
11       }
12       anzeige(z);
13       anzeige(z0);
14     }
15   }
16 }
```

Aufgaben

1. Schreiben Sie eine Klasse, welche die Bäume, Pilze und Blätter einer Welt mit einem Array zählt und das Ergebnis anzeigt.
2. Simulieren Sie ein Galtonbrett mit Kara als „Kugel“ und „Bäumen“ als Stifte.
   (Informationen dazu unter `http://de.wikipedia.org/wiki/Galtonbrett`)

9 Erweiterungen

Eine Reihe von Problemen wären einfacher lösbar, wenn KARA mehr Sensoren hätte. Man kann eigene Sensoren als Methoden leicht zusätzlich definieren.

Beispiel: Es soll ein Sensor definiert werden, der einen Pilz links von KARA entdeckt.

Dazu muss sich KARA nach links drehen, sich merken, ob jetzt vor ihm ein Pilz steht und sich dann zurück nach rechts drehen.

```
1  public boolean pilzLinks() {
2    kara.turnLeft();
3    boolean pilz=kara.mushroomFront();
4    kara.turnRight();
5    return pilz;
6  }
```

Nach diesem Verfahren kann man natürlich auch einen Sensor für rechts von KARA stehende Pilze definieren. Allerdings hat das Verfahren den Nachteil, dass KARA bewegt werden muss, was die Programmausführung verlangsamt. Will man entsprechend Sensoren definieren, die erkennen, ob KARA links von, rechts von oder vor einem Blatt steht, so wird der Aufwand noch höher, weil der vorhandene Sensor Blätter nur erkennt, wenn KARA auf ihnen steht. Um zu erkennen, ob KARA vor einem Blatt steht, muss KARA daher einen Schritt vorwärts machen. Das geht aber nicht immer. KARA könnte vor einem Baum stehen, dann kann er keinen Schritt vorwärts machen. Steht er vor einem Pilz, so kann er vielleicht einen Schritt vorwärts machen, verschiebt aber dabei den Pilz.

Wenn man die Methoden der Klasse `world` verwendet, kann man alternativ etwa folgenden Sensor erzeugen:

```
1  public boolean leafSouth() {
2    int y= (int)kara.getPosition().getY();
3    int x= (int)kara.getPosition().getX();
4    int ymax= world.getSizeY()-1;
5    if (y==ymax) {
6      return world.isLeaf(x,0);
7    }
8    else {
9      return world.isLeaf(x,y+1);
10   }
11 }
```

Hier wird zunächst KARAS Position abgefragt. Das in Klammern stehende `(int)` ist notwendig, weil die Abfrage die Position als `double` (Dezimalzahlformat) liefert. Mit `(int)` wird daraus (durch Abschneiden des hier ohnehin nicht vorhandenen) Nachkommateils eine ganze Zahl. Außerdem muss berücksichtigt werden, dass die Welt in form einer Torus vorliegt. Wenn KARA auf der untersten Zeile steht, so ist die oberste Zeile „südlich" von KARA . Dem wird mit der Fallunterscheidung Rechnung getragen.

Entsprechend kann man auch Methoden schreiben, die KARA in eine bestimmte Himmelsrichtung drehen. Natürlich kann das auch direkt mit `kara.setDirection` bewerkstelligt werden, aber eine passende Neudefinition ist einprägsamer:

```
1   public void turnEast(){
2     kara.setDirection(3);
3   }
```

Wenn man eine Reihe solcher Definitionen häufiger verwenden will, kann man sie in einer eigenen von JavaKaraProgram abgeleiteten Klasse definieren. Die ganzen Definitionen der Sensoren und die Drehungen sind beispielsweise in der Klasse KepiKara zusammengestellt. Die Methode `myProgram()` wird dabei leer gelassen. Speichert man diese Klasse im Verzeichnis der Datei javakara.jar, so kann man danach neue Klassen statt von JavaKaraProgram von KepiKara ableiten und hat dabei sowohl die zu JAVAKARA gehörenden als auch die neudefinierten Methoden zur Verfügung.

Die folgende Klasse verwendet diese Methode, um eine vereinfachte Fassung von `Pacman` zu erzeugen:

```
1  import javakara.JavaKaraProgram;
2  public class PacmanNeu extends KepiKara {
3    private void nextLeaf() {
4      if (leafEast()) {
5        turnEast();
6      }
7      else if (leafNorth()) {
8        turnNorth();
9      }
10     else if (leafWest()) {
11       turnWest();
12     }
13     else {
14       turnSouth();
15     }
16     kara.move();
17     kara.removeLeaf();
18   }
19
20   public void myProgram() {
21     kara.removeLeaf();
22     while (!kara.treeFront()) {
23       nextLeaf();
24     }
25   }
26 }
```