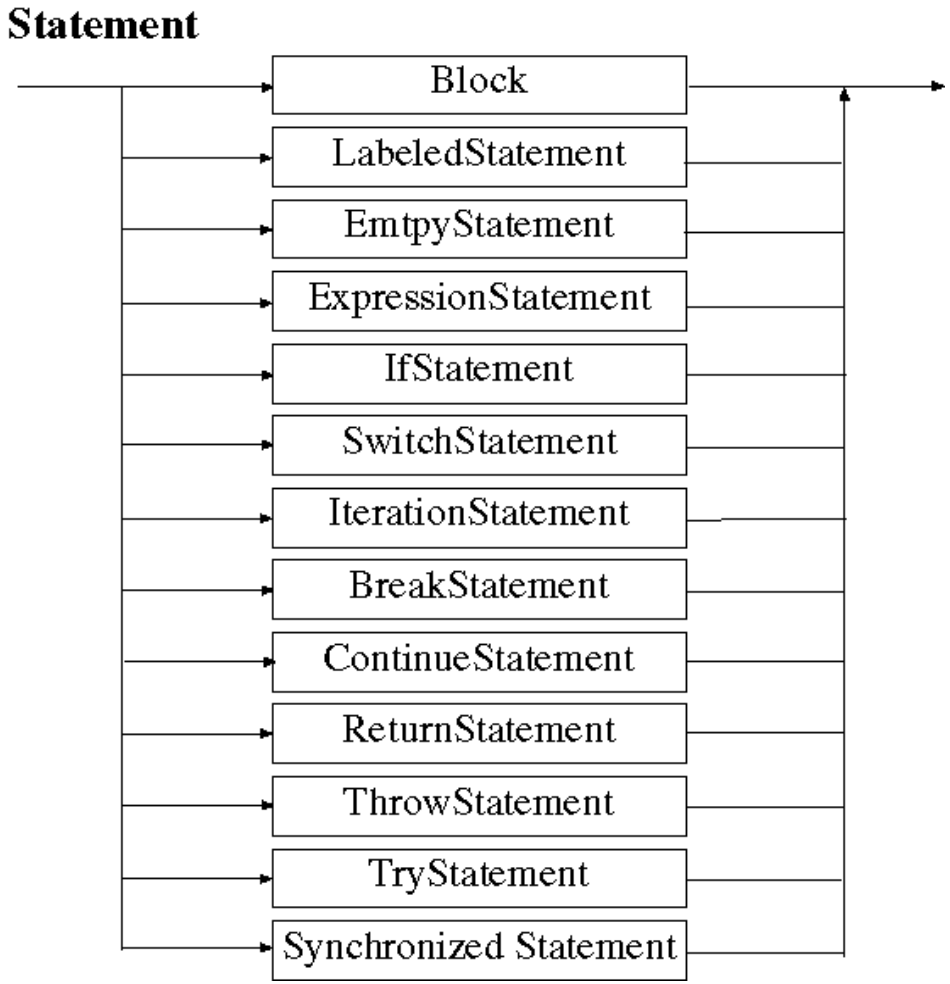
2.7 Anweisungen

Anweisungen (Statements) sind in Java die elementaren Einheiten des Kontrollflusses. Eine Anweisung stellt eine Aktivität dar, sie bewirkt etwas.

Ausdrücke sind in Java keine Anweisungen (anders als in C oder C++).

```
x+2; // Java-Fehler, korrekt in C oder C++
```

Statement

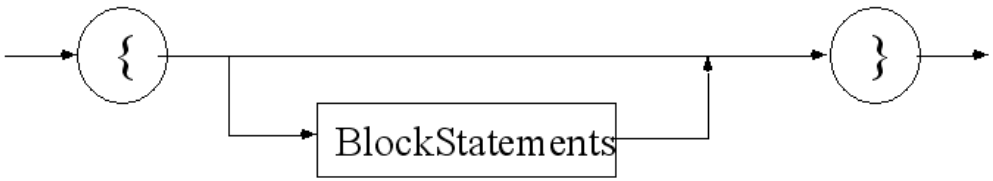

Statement
Block
LabeledStatement
EmtpyStatement
ExpressionStatement
IfStatement
SwitchStatement
IterationStatement
BreakStatement
ContinueStatement
ReturnStatement
ThrowStatement
TryStatement
Synchronized Statement

Anweisungsblock

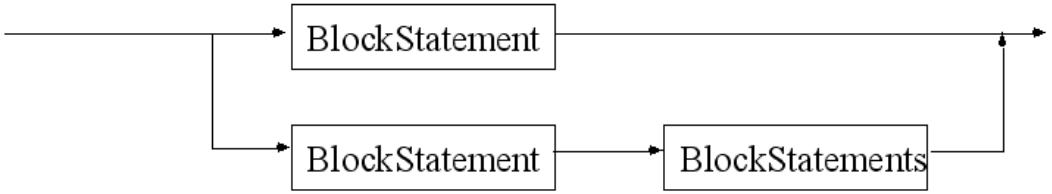
Einzelne Anweisungen können in Java zu Blöcken zusammengefasst werden. Dazu dienen die geschweiften Klammern { und }. Jeder Methodenrumpf besteht aus einem Block.

Block

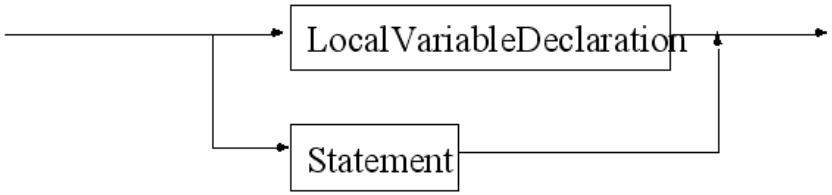
Block



BlockStatements



BlockStatement



Lokale Variablen

In einem Block können lokale Variablen deklariert werden. Die Deklaration besteht aus dem optionalen `final`-Spezifizierer und einer Typangabe, gefolgt von einem oder mehreren Bezeichnern und jeweils optionalen Initialisierern. Jeder Bezeichner deklariert eine lokale Variable, die den Bezeichner als Namen erhält. Zum Beispiel ist

```
int i,j = 1, k;
```

äquivalent zu:

```
int i;
int j = 1; // Deklaration mit Initialisierer
int k;
```

Der Geltungsbereich von lokalen Variablen

Der Geltungsbereich einer lokalen Variablen – das ist der Java-Code, in dem man ihren Namen verwenden kann – ist der Rest des Blocks, der auf die Deklaration folgt. Im Unterschied zu anderen Sprachen ist es in Java nicht zulässig, Variablen in geschachtelten Blöcken zu redeklarieren. Folgendes wird also nicht übersetzt:

```
{
    int 1 = 5;
    {
        double i = 5.7; //Fehler Redeklaration von i
    }
}
```

Deklarationen mit Initialisierungen sind ausführbare Anweisungen!

Sie werden von links nach rechts ausgeführt. Beispielsweise erhält `j` hier den Wert 3.

```
int i = 2, j = i + 1;
```

`Mit final` spezifizierte lokale Variablen

Wenn eine lokale Variable `final` spezifiziert ist, wird sie zur symbolischen Konstanten. Jede Zuweisung an eine solche lokale Variable, außer der, mit der mit der sie ihren konstanten Wert erhält, ist ein Fehler.

Typischweise wird man diesen Wert mit einem Initialisierer bereitstellen:

```
final int MAX_ANZAHL = 300;
```

Wertzuweisung an eine `final` - Variable (1)

Es ist auch möglich, eine final lokale Variable nicht bei ihrer Deklaration zu initialisieren; sie ist dann *unspezifiziert* und **`final`**. Einer unspezifizierten **`final`**-Variablen muss an genau einer Stelle des Programms – vor dem ersten Zugriff auf ihren Wert – ein Wert zugewiesen werden. Dies wird vom Java-Compiler geprüft.

```
// BlankFinal.java

import java.util.*;

class BlankFinal {

    public static void main(String[] args) {

        GregorianCalendar c = new GregorianCalendar();

        final int JAHR =   c.get(Calendar.YEAR),
                           ANZAHL_TAGE;

        if (c.isLeapYear(JAHR))

            ANZAHL_TAGE = 366;
```

Wertzuweisung an eine `final` - Variable (2)

```
        else
            ANZAHL_TAGE = 365;
        System.out.println("Das Jahr " + JAHR + " hat "
            + ANZAHL_TAGE + " Tage.");
    }
}
```

Die `if`-Anweisung

Die Syntaxregel für die `if`-Anweisung ist

`if` (Ausdruck) Anweisung;

`if` (Ausdruck) Anweisung `else` Anweisung;

Der Ausdruck muss jeweils vom Typ `boolean` sein. Bei der Ausführung der Anweisung wird als Erstes dieser Ausdruck ausgewertet. Ergibt sich `true`, wird die Anweisung nach `if` ausgeführt. Anderenfalls wird – falls vorhanden – die Anweisung nach `else` ausgeführt. Wenn das Resultat `false` ist und kein `else` vorhanden ist, wird die `if`-Anweisung ohne weitere Aktivität beendet.

Mehrdeutigkeit beim `else`

Wenn mehrere **`if`**-Anweisungen geschachtelt sind, gehört ein **`else`** immer zum unmittelbar voranstehenden **`if`**. Das heißt:

```
if ( i >= 0 )
    if ( j > 0 )
        System.out.println(j + "ist positiv" );
else
    System.out.println(j + "ist negativ" );
```

ist missverständlich formatiert; wenn `i` beispielsweise den Wert -5 hat, wird nichts ausgegeben.

Die `switch`-Anweisung

Mit der **`switch`**-Anweisung kann man im Unterschied zur **`if`**-Anweisung zu einer aus beliebig vielen Anweisungen verzweigen.

`switch` (Ausdruck) **`switch`**-Block;

Ein **`switch`**-Block ist ein Block, dessen Anweisungen mit einer oder mehreren **`switch`**-Marken markiert sein können und der mit einer beliebigen Anzahl solcher Marken enden kann. Der Wert des Ausdrucks nach **`switch`** entscheidet darüber, an welcher **`switch`**-Marke die VM ihre Arbeit fortsetzt.

switch-Marke:

`case` Konstanter-Ausdruck:

`default`:

Formale Anforderung an die `switch`-Anweisung

- Der Typ des Ausdrucks nach **`switch`** muss **`byte`**, **`short`**, **`int`** oder **`char`** sein.
- Jeder konstante Ausdruck in einer **`switch`**–Marke muss zuweisungs-kompatibel zu diesem Typ sein.
- Die konstanten in den **`switch`**-Marken eines **`switch`**-Blocks müssen paarweise verschiedene Werte haben.
- Innerhalb eines **`switch`**-Blocks darf höchstens einmal **`default`** als Marke vorkommen.

Ausführung der `switch`-Anweisung (1)

Bei der Ausführung einer **`switch`**-Anweisung wird als erstes der Ausdruck nach dem **`switch`** ausgewertet. Sein Wert wird dann mit den konstanten Ausdrücken aller **`switch`**-Marken verglichen. Stimmt er mit einer dieser Konstanten überein, werden alle Anweisungen hinter der entsprechenden Marke aufgeführt. Anderenfalls werden, falls eine **`default`**-Marke vorhanden ist, alle Anweisungen hinter dieser ausgeführt. Anderenfalls, d.h., wenn weder eine der Konstanten „passt“ noch eine **`default`**-Marke vorhan-den ist, wird die **`switch`**-Anweisung ohne weitere Aktivität beendet.

Zum Beispiel wird hier, wenn `i` den Wert 5 hat, viele ausgegeben, ein `i`-Wert von 2 führt zu den Ausgaben zwei und viele:

```
switch (i) {
    case 1: System.out.println("eins");
    case 2: System.out.println("zwei");
    default: System.out.println("viele");
}
```

Ausführung der `switch`-Anweisung (2)

In der Regel wird das im Beispiel auftretende „Durchfallen“ zu den Anweisungen hinter den nachfolgenden **`switch`**-Marken unerwünscht sein. Man kann es durch **`break`**-Anweisungen unterbinden.

```
case 1: System.out.println("eins"); break;

case 2: System.out.println("zwei"); break;
```

Die `while`-Anweisung

Die **`while`**-Anweisung führt die Auswertung eines Ausdrucks und eine Anweisung solange wiederholt aus, bis der Wert des Ausdrucks **`false`** ist. Sie hat die Syntax:

`while` (Ausdruck) Anweisung;

Der Ausdruck muss vom Typ **`boolean`** sein. Bei der Ausführung der **`while`-Anweisung** wird als Erstes der Wert des Ausdrucks ermittelt. Ist er **`true`**, wird die Anweisung nach **`while`** ausgeführt. Dann wir mit der Ausführung der gesamten **`while`**-Anweisung von vorne begonnen – beginnend mit der Ermittlung des Werts des Ausdrucks. Wenn anderenfalls der Wert des Ausdrucks **`false`** ist, wird die **`while`**-Anweisung ohne weiter Aktivität be-endet.

Beispiel für die `while`-Anweisung

```
// Berechne die Quersumme
// While.java
class While {
    public static void main(String[] args) {
        int z = 123456, x = z, querSumme = 0;
        while (x != 0) {
            querSumme += x%10;
            x /= 10;
        }
        System.out.println(z + " hat als Quersumme " +
                querSumme);
    }
}
```

Die do-Anweisung

Die **do**-Anweisung führt eine Anweisung und die Auswertung eines Ausdrucks solange wiederholt aus, bis der Wert des Ausdrucks **false** ist. Sie hat die Syntax:

`do` Anweisung `while` (Ausdruck);

Der Ausdruck muss vom Typ **boolean** sein. Bei der Ausführung der **do**-Anweisung wird als erstes die Anweisung nach **do** ausgeführt. Anschließend wir der Wert des Ausdrucks ermittelt. Ist er **true**, wird die gesamte **do**-Anweisung erneut ausgeführt. Andernfalls, d. h., wenn der Wert des Ausdrucks **false** ist, wird die **do**-Anweisung ohne weiter Aktivität beendet.

Beispiel für die `do-while`-Anweisung

Im Unterschied zur **`while`**-Anweisung wird die Anweisung in der **`do`**-Anweisung mindestens einmal ausgeführt. Zum Beispiel berechnet

```
final double EPS = 1e-15;

 double x = 2.0, links = 0.0, rechts = ((x >= 0) ? x :
       1.0), quad;

do {
    quad = 0.5 * (links + rechts);
    if (quad * quad > x)
        rechts = quad;
    else
        links = quad;
} while (rechts - links > EPS);
```

die Quadratwurzel (`quad`) einer positiven Zahl `x` mittels Intervallhalbierung. `EPS` ist die gewünschte Genauigkeit.

Die `for`-Anweisung

Die **`for`**-Anweisung nimmt zuerst eine Initialisierung vor, führt dann die Auswertung eines Ausdrucks, eine Anweisung und einige Updates solange wiederholt aus, bis der Wert des Ausdrucks **`false`** ist. Sie wird typischerweise zur Iteration über die Komponenten von Feldern benutzt. Die Syntaxnotation ist:

`for` (For-Init$_{opt}$; Ausdruck$_{opt}$; For-Update$_{opt}$;) Anweisung;

Der Ausdruck muss vom Typ **`boolean`** sein, er dient wieder als Bedingung, die den Abbruch oder die Fortsetzung der Wiederholungen kontrolliert.

Die Elemente der `for`-Anweisung

For-Init ist eine Deklaration lokaler Variablen oder einer Liste von Ausdrücken. *For-Update* ist eine Liste von Ausdrücken, mit denen man in der Regel die Zählervariablen fortschreibt. In beiden Fällen sind nur die in Ausdrucksanweisungen möglichen Ausdrücke, also Zuweisungs-, Inkrement-, Dekrement-, oder **`new`**-Ausdrücke sowie Methodenaufrufe zulässig.

Die Berechnung der **`for`**-Anweisung werden wie folgt vorgenommen: Der Ausdruck wird ausgewertet (falls vorhanden). Ist sein Wert **`true`**, so wird die Anweisung nach **`for`** ausgeführt, die Update-Ausdrücke werden (falls vorhanden) von links nach rechts ausgewertet, und eine neue Iteration wird begonnen. Ist der Wert des Ausdrucks **`false`**, wird die **`for`**-Anweisung ohne weitere Aktivität beendet.

Beispiel für die `for`-Anweisung

Mit den folgenden Anweisungen berechnen wir den Mittelwert (xQuer) von n ganzen Zahlen:

```
double xQuer = 0.0;
    for (int i = 1; i <= n; i++) {
        System.out.print("x" + i + "? ");
        xQuer += Integer.parseInt(in.readLine());
    }
    xQuer /= n;
```

Markierte Anweisung

Anweisungen können in der Form

Bezeichner : Anweisung

markiert werden. Die Markierung einer Anweisung wirkt sich auf deren Ausführung nicht aus. Sie hat jedoch eine Bedeutung im Zusammenhang mit den **`break`**- oder **`continue`**-Anweisungen, die wir in den beiden folgenden Abschnitten besprechen.

Eine goto-Anweisung gibt es in Java nicht!

Die `break`-Anweisung

Die **`break`**-Anweisung hat die Form:

`break` Bezeichner$_{opt}$;

Der optimale Bezeichner in einer **`break`**-Anweisung muß Marke einer markierten Anweisung sein.

Eine **`break`**-Anweisung ohne Marke darf nur in einer **`switch`**-Anweisung oder einer Wiederholungsanweisung (**`do`**, **`for`**, **`while`**) vorkommen. In diesem Fall beendet die **`break`**-Anweisung die Ausführung der innersten, das **`break`** umschließenden Schleife bzw. **`switch`**-Anweisung. Die Ausführung wird mit der Anweisung, die auf die abgebrochene Anweisung folgt, fortgesetzt.

Beispiel für die `break`-Anweisung (1)

```
// Break.java
import java.util.*;
class Break {
    public static void main(String[] args) {
        final int MONAT = new GregorianCalendar().get(Calendar.MONTH);
        switch (MONAT) {
            default: System.out.println("vorlesungsfrei"); break;
            case 9: case 10: case 11: case 0: case 1:
                System.out.println("Wintersemester"); break;
            case 3: case 4: case 5: case 6:
                System.out.println("Sommersemester"); break;
        }
    }
}
```

Beispiel für die `break`-Anweisung (2)

Ohne die **`break`**-Anweisung würden hier wegen des Durchfallens zu nachstehenden **`switch`**-Marken zum Teil falsche Ausgaben erfolgen.

Die `continue`-Anweisung

Eine **`continue`**-Anweisung darf nur innerhalb von Wiederholungsanweisungen (**`while`**, **`do`**, **`for`**) stehen. Sie hat die Form:

`continue` Bezeichner$_{opt}$;

Der optionale Bezeichner in einer **`continue`**-Anweisung muss Marke einer markierten Weiderholungsanweisung sein, die das **`continue`** enthält.

Die Ausführung einer **`continue`**-Anweisung ohne Marke bewirkt einen Sprung an das Ende der innersten, das **`continue`** umschließenden Schleife und beginnt gegebenenfalls eine erneute Iteration. Das heißt, bei einer **`while`**- oder **`do`**-Schleife wird die Abbruchbedingung (der boolesche Ausdruck) ausgewertet, bei einer **`for`**-Schleife werden noch die Update-Ausdrücke ausgewertet.

Beispiel für die `continue`-Anweisung (1)

```
// Continue.java
class Continue {
    static int wurf() {
        return (int)(6*Math.random()) + 1;
    }
    public static void main(String[] args) {
        int w1, w2;
        while (true) {
            w1 = wurf();
            if (w1 != 6) {
                System.out.println(w1);
```

Beispiel für die `continue`-Anweisung (2)

```
                continue;
            }
            w2 = wurf();
            System.out.println(w1 + "    " + w2);
            if (w2 == 6)
                break;
        }
    }
}
```

Es wird mit einer Methode **`wurf`** solange mit zwei Würfeln „gewürfelt“, bis zwei Sechsen gefallen sind.

2.8. Felder (Arrays)

In Java sind Felder **Objekte**! Sie werden (wie alle Objekte) dynamisch erzeugt.

Die Variablen in einem Feld, die *Feldkomponenten*, haben keinen eigenen Namen. Man greift auf sie über den Feldnamen und einen in `[` und `]` geklammerten ***Index*** zu. Wenn ein Feld n Komponenten hat, ist n die ***Länge*** des Felds. Die Feldkomponenten werden dann mit `0,1,…, n-1` indiziert.

Alle Komponenten eines Felds haben denselben Typ. Wenn dieser Typ `T` ist, kann für den Typ des Felds kurz `T[]` geschrieben werden.

Java unterstützt keine mehrdimensionalen Felder.

Deklaration von Feldern

Eine Variable eines Feldtyps oder eine Feldvariable deklariert man durch die Angabe des Komponententyps, gefolgt von leeren Klammern `[]`, dem Variablennamen und einem optionalen Initialisierer.

```
int[] a;              // int-Feld

String [] c;          // String-Feld

short [][] b;         // Feld mit short-Variablen
                      //  als Komponenten
```

Die Erzeugung von Feldern (1)

Feldobjekte kann man auf drei verschiedene Wege erzeugen:

1. Durch explizite Objekterzeugung mittels **`new`**, z. B.

   ```
   a = new int[5];
   ```

   Hier wird ein Feldobjekt der Länge 5 erzeugt. In seinen Komponenten stehen zunächst die Standartwerte, also jeweils 0.

2. Durch Spezifikation einer Liste von Initialisierern bei der Variablen-deklaration. Beispielsweise:

   ```
   double[] d = {1.1, 1.21, 1.331};
   ```

   Hier stellt die VM die Feldlänge fest, erzeugt das Feld mittels **`new`** und trägt die Werte aus der Initialisiererliste in die Feldkomponenten ein.

Die Erzeugung von Feldern (2)

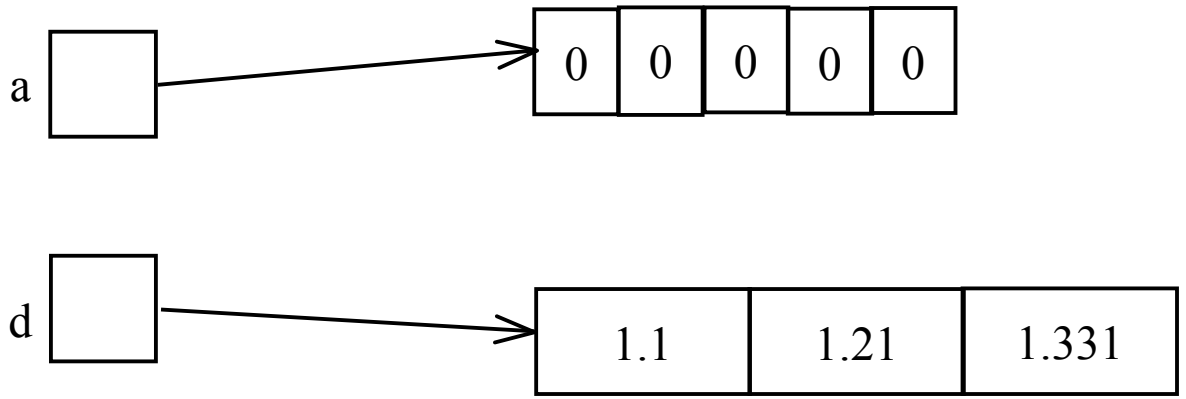
3. Durch Kombination von 1. und 2., z. B.

```
c = new String[]{"Die", "Erzeugung", "von",
      "Feldern"};
```

Hier darf in den Klammern `[]` keine Feldlänge – auch nicht die korrekte, also im Beispiel 4 – angegeben werden .

Die erzeugten Feldobjekte

Die erzeugten Feldobjekte sind in der folgenden Abbildung zusammen mit ihren Feldvariablen dargestellt. Es ist wichtig, sich den Unterschied zwischen der Feldvariablen, die einen Namen besitzt und das Feld referenziert, sowie dem Feld, das namenlos ist, klarzumachen.



Der Zugriff aus Feldkomponenten

Auf die Komponenten eines Feldes greift man mit einem in `[]` geklammerten Index in elementaren Ausdrücken zu. Dabei ist zu beachten, dass die erste Komponente immer den Index 0 hat.

Felder werden mit **`int`**-Werten indiziert. Auch **`byte`**-, **`short`**- oder **`char`**-**Werte** sind möglich, sie werden vor dem Zugriff implizit in int's umgewandelt.

Beispiel

```
int n = 1001;

int[] x = new int[n];

for(int i = 0; i < n; i++)

    x[i] = i;
```

Felder als Objekte

Wie schon erwähnt, sind Felder Objekte. Neben den Methoden **`clone`**, **`equals`** usw., die jedes Objekt von der Superklasse **`Object`** erbt, verfügen Felder über eine **`final`**-Instanzvariable **`length`**, die die Anzahl der Feldkomponenten enthält. Zum Zugriff auf die Feldlänge benutzt man die Feldvariable und einen "." Man schreibt bei einem mit `x` bezeichneten Feld somit `x.length`.

Beim Zuweisen von Feldvariablen ist daran zu erinnern, dass – wie immer bei Referenzen auf Objekte – lediglich der Referenzwert, aber nicht das Feld kopiert wird. Im Beispiel

```
boolean[] a = {true, true, true, false};

boolean[] b = a;

a[a.length - 1] = true;
```

wird ein einziges Feld erzeugt, das von zwei Variablen `a` und `b` referenziert wird. Nach der letzten Zuweisung liefern sowohl `a[3]` als auch `b[3]` den Wert true.

Kopieren von Feldern mit der Methode `clone`

Zum Kopieren von Feldern kann man die Methode **`clone`** benutzen. Sie wird (wie bei **`length`**) zum Aufruf durch einen "." mit dem Feldnamen verknüpft. Wenn wir das obige Beispiel abändern zu

```
boolean[] a = {true, true, true, false};

boolean[] b = (boolean[])a.clone();

a[a.length - 1] = true;
```

legt Java ein zweites **`boolean`**-Feld an, das durch **`b`** referenziert wird. In **`a[3]`** bzw. **`b[3]`** erhalten wir jetzt den neuen Wert **`true`** bzw. den alten Wert **`false`**.

Der explizite Cast auf den Typ **`boolean`** ist hier notwendig, weil **`clone`** als Resultat immer eine Referenz des Typs **`Object`** liefert.

Kopieren von Feldern mir `arraycopy`

Neben dem Klonen bietet Java noch die Möglichkeit, mittels **`arraycopy`** einen bestimmten Bereich aus einem Feld in ein anderes Feld, das bereits existieren muss, zu kopieren. Es handelt sich dabei um eine Klassenmethode der Klasse **`System`**.

`System.arraycopy(von, vonInd, nach, nachInd, anzahl)` kopiert **`anzahl`** Komponenten aus dem Feld **`von`** in das Feld **`nach`**, jeweils beginnend ab dem Index **`vonInd`** bzw. **`nachInd`**. Wenn die Indizes dabei die Feldgrenzen verlassen, wird eine Ausnahme des Typs **`ArrayIndexOutOfBoundException`** ausgeworfen.

Beispiel

```
char[] a = {'a', 'b', 'c', 'd', 'e', 'f', 'g',
            'h', 'i', 'j'};
char[] b = new char[15];
for(int i = 0; i < b.length; i++)
     b[i] = 'x';
System.arraycopy(a, 1, b, 5, 6);
```

In dem Feld `b` sind nun die Zeichen xxxxxbcdefgxxxx gespeichert.

Felder und Zeichenketten

Im Unterschied zu C/C++ haben **`String`**-Objekte in Java nicht den Typ **`char[]`**. Auch sind weder **`String`** noch **`char`**-Felder mit '\u0000' terminiert. **`String`**-Objekte haben einen konstanten Wert, der beim Übersetzen fest-gelegt wird. In der Klasse **`String`** ist jedoch eine Instanzmethode **`toCharArray`** deklariert, die bei ihrem Aufruf ein **`char`**-Feld mit derselben Zeichenfolge wie im String erzeugt und als Resultat liefert. Wir können sie beispielsweise folgendermaßen aufrufen

```
String str = "StringChars";
char[] c = str.toCharArray();
for (int i = 0; i < c.length; i++)
    System.out.print(c[i]);
```

Bemerkungen zu Feldern

- In allen Beispielprogrammen war die Methode **`main`** der initialen Klasse so deklariert: **`public static void main(String args[]){…}`**

  Mit **`args`** ist hier ein **`String`**-Feld bezeichnet, in dem die VM die beim Interpretierungsaufruf spezifizierten Kommandozeilen-Argumente ablegt.

- Die Spezifizierung eines Feldes als **`final`** (symbolische Konstante) wirkt sich nicht auf die Feldkomponenten aus; diese sind dennoch veränderbar. Lediglich die Feldvariable ist konstant und kann kein anderes Feld referenzieren. Zum Beispiel:

```
final int[] a = { -1, -2, -3, -4, -5 };

a[0] *= -1;                          //korrekt

a  = new int[] {102, 104};           //Fehler
```

- Ein Feld wird – wie jedes andere Objekt -, wenn es nicht mehr referenziert wird, von Javas Garbage Collector gelöscht.