1.5 Variablen, Datentypen und Operatoren

Zu Beginn des letzten Jahrhunderts gehörte das Rechnen mit Zahlen zu denjenigen Fähigkeiten, die ausschließlich intelligenten Wesen zugeschrieben wurden. Heute weiß jedes Kind, dass ein Computer viel schneller und verlässlicher rechnen kann als ein Mensch. Als „intelligent“ werden heute gerade diejenigen Tätigkeiten angesehen, die früher eher als trivial galten: Gesichter und Muster erkennen, Assoziationen bilden, sozial denken. Zukünftig werden Computer sicherlich auch diese Fähigkeiten erlernen, zumindest teilweise. Wir werden uns aber in diesem Skript auf die grundlegenden Operationen beschränken, die früher als intelligent galten und heute als Basis für alles Weitere dienen. Fangen wir mit den Grundrechenarten an.

Wie jede Programmiersprache besitzt auch Java die Grundrechenoperationen. Sie bestehen wie die meisten Operationen aus einem „Operator“ und zwei „Operanden“. Z.B. besteht die Operation `2+3` aus dem Operator `+` und den Operanden `2` und `3`. Allgemein kann man eine Operation wie folgt definieren.

Definition 1.1. Eine Operation ist gegeben durch einen Operator und mindestens einen Operanden. Hierbei ist ein *Operator* ein Zeichen, das die Operation symbolisiert, meist eine zweistellige Verknüpfung, und die Argumente dieser Verknüpfung heißen *Operanden*. □

Diese Definition einer Operation erscheint trivial, sie hat es jedoch in sich: Wendet man denselben Operator auf verschiedene Datentypen an, so kommen möglicherweise verschiedene Ergebnisse heraus. Wir werden das in der nächsten Applikation sehen.

```
1  import javax.swing.JOptionPane;
2  /**
3   * Führt verschiedene arithmetische Operationen durch
4   */
5  public class Arithmetik {
6     public static void main( String[] args ) {
7        int m, n, k;    // ganze Zahlen
8        double x, y, z; // 'reelle' Zahlen
9        String ausgabe = "";
10       // diverse arithmetische Operationen:
11       k = - 2 + 6 * 4;
12       x = 14 / 4;  y = 14 / 4.0;  z = (double) 14 / 4;
13       n = 14 / 4;  m = 14 % 4;
14       // Ausgabestring bestimmen:
15       ausgabe = "k = "+ k;
16       ausgabe = ausgabe + "\nx = "+ x + ", y = "+ y + ", z = "+ z;
17       ausgabe = ausgabe + "\nn = "+ n + ", m = "+ m;
18       JOptionPane.showMessageDialog(
19          null, ausgabe, "Ergebnis", JOptionPane.PLAIN_MESSAGE
20       );
```

```
21     }
22 }
```

Das Programm führt einige arithmetische Operationen aus. Auf den ersten Blick etwas überraschend ist die Ausgabe, siehe Abb. 1.6: Je nach Datentyp der Operanden ergibt der (scheinbar!)

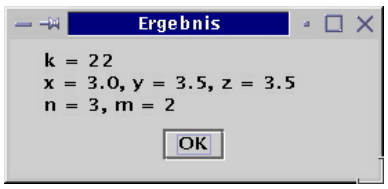


Abbildung 1.6. Die Applikation Arithmetik.

gleiche Rechenoperator verschiedene Ergebnisse.

1.5.1 Variablen, Datentypen und Deklarationen

Die Zeile 7 der obigen Applikation,

```
int m, n, k;
```

ist eine *Deklaration*. Der Bezeichner `k` ist der Name einer *Variablen*. Eine Variable ist ein Platzhalter, der eine bestimmte Stelle im Arbeitsspeicher (RAM) des Computers während der Laufzeit des Programms reserviert, siehe Abb. 1.7. Dort kann dann ein *Wert* gespeichert werden.

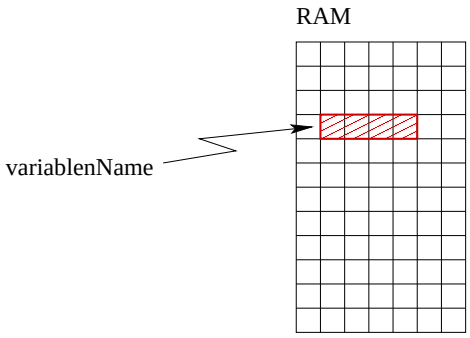


Abbildung 1.7. Die Deklaration der Variable `variablenName` bewirkt, dass im Arbeitsspeicher (RAM) eine bestimmte Anzahl von Speicherzellen reserviert wird. Die Größe des reservierten Speichers (= Anzahl der Speicherzellen) ist durch den Datentyp bestimmt. Vgl. Tabelle 1.3 (S. 20)

Es ist nicht ein beliebig großer Wert möglich, denn dafür wäre ein unbegrenzter Speicherplatz notwendig: Die Menge der möglichen Werte wird durch den *Datentyp* festgelegt. In unserem Beispiel ist dieser Datentyp **`int`**, und der Wertebereich erstreckt sich von -2^{31} bis $2^{31} - 1$.

Deklarationen werden mit einem Semikolon (`;`) beendet und können über mehrere Zeilen aufgesplittet werden. Man kann mehrere Variablen des gleichen Typs in einer einzigen Deklaration durch Kommas (`,`) getrennt deklarieren, oder jede einzeln. Man könnte also in unserem Beispiel genauso schreiben:

```
int m;
int n;
int k;
```

Ein Variablenname kann ein beliebiger gültiger Bezeichner sein, also eine beliebige Zeichenfolge, die nicht mit einer Ziffer beginnt und keine Leerzeichen enthält. Es ist üblich, Variablennamen (genau wie Methodennamen) mit einem kleinen Buchstaben zu beginnen.

Variablen müssen stets mit ihrem Namen und ihrem *Datentyp* deklariert werden. In Zeile 7 besagt die Deklaration, dass die Variablen vom Typ `int` sind.

1.5.2 Der Zuweisungsoperator =

Eine der grundlegendsten Anweisungen ist die Wertzuweisung. In Java wird die Zuweisung mit dem so genannten *Zuweisungsoperator (assign operator)* = bewirkt. In Zeile 11,

```
k = - 2 + 6 * 4;
```

bekommt die Variable `k` den Wert, der sich aus dem Term auf der rechten Seite ergibt, also 22. (Java rechnet Punkt vor Strichrechnung!) Der Zuweisungsoperator ist gemäß Def. 1.1 ein Operator mit zwei Operanden.

Merkregel 3. Der Zuweisungsoperator = weist der Variablen auf seiner linken Seite den Wert des Ausdrucks auf seiner rechten Seite zu. Der Zuweisungsoperator = ist ein *binärer Operator*, er hat zwei *Operanden* (die linke und die rechte Seite).

Alle Operationen, also insbesondere Rechenoperationen, müssen stets auf der rechten Seite des Zuweisungsoperators stehen. Ein Ausdruck der Form `k = 20 + 2` ist also sinnvoll, **aber eine Anweisung** `20 + 2 = k` **hat in Java *keinen Sinn*!**

Datentypen und Speicherkonzepte

Variablennamen wie `m` oder `n` in der Arithmetik-Applikation beziehen sich auf bestimmte Stellen im Arbeitsspeicher des Computers. Jede Variable hat einen *Namen*, einen *Typ*, eine *Größe* und einen *Wert*. Bei der Deklaration am Beginn einer Klassendeklaration werden bestimmte Speicherzellen für die jeweilige Variable reserviert. Der Name der Variable im Programmcode verweist während der Laufzeit auf die Adressen dieser Speicherzellen.

Woher weiß aber die CPU, wieviel Speicherplatz sie reservieren muss? Die acht sogenannten *primitiven Datentypen* bekommen in Java den in Tabelle 1.3 angegebenen Speichergrößen zugewiesen.

Datentyp	Größe	Werte	Bemerkungen
`boolean`	1 Byte	`true`, `false`	
`char`	2 Byte	`'\u0000'` bis `'\uFFFF'`	Unicode, 2^{16} Zeichen, stets in Apostrophs (!)
`byte`	1 Byte	−128 bis +127	$-2^7, -2^7+1, \ldots, 2^7-1$
`short`	2 Byte	−32 768 bis +32 767	$-2^{15}, -2^{15}+1, \ldots, 2^{15}-1$
`int`	4 Byte	−2 147 483 648 bis +2 147 483 647	$-2^{31}, -2^{31}+1, \ldots, 2^{31}-1$ Beispiele: `123`, `-23`, `0x1A`, `0b110`
`long`	8 Byte	−9 223 372 036 854 775 808 bis +9 223 372 036 854 775 807	$-2^{63}, -2^{63}+1, \ldots, 2^{63}-1$ Beispiele: `123L`, `-23L`, `0x1AL`, `0b110L`
`float`	4 Byte	±1.4E-45 bis ±3.4028235E+38	$[\approx \pm 2^{-149}, \approx \pm 2^{128}]$ Beispiele: `1.0f`, `1.F`, `.05F`, `3.14E-5F`
`double`	8 Byte	±4.9E-324 bis ±1.7976931348623157E+308	$[\pm 2^{-1074}, \approx \pm 2^{1024}]$ Beispiele: `1.0`, `1.`, `.05`, `3.14E-5`

Tabelle 1.3. Die primitiven Datentypen in Java

Wird nun einer Variable ein Wert zugewiesen, z.B. durch die Anweisung

```
k = - 2 + 6 * 4;
```

der Variablen `k` der berechnete Wert auf der rechte Seite des Zuweisungsoperators, so wird er in den für `k` reservierten Speicherzellen gespeichert. Insbesondere wird ein Wert, der eventuell vorher dort gepeichert war, überschrieben. Umgekehrt wird der Wert der Variablen `k` in der Anweisung

```
ausgabe = "k = " + k;
```

der Zeile 15 nur aus dem Speicher gelesen, aber nicht verändert. Der Zuweisungsoperator zerstört nur den Wert einer Variablen auf seiner *linken* Seite und ersetzt ihn durch den Wert der Operationen auf seiner *rechten* Seite.

Eine Variable muss bei ihrem ersten Erscheinen in dem Ablauf eines Programms auf der linken Seite des Zuweisungsoperators stehen. Man sagt, sie muss *initialisiert* sein. Hat nämlich eine Variable keinen Wert und steht sie auf der rechten Seite des Zuweisungsoperators, so wird ein leerer Speicherplatz verwendet. In Java ist die Verarbeitung einer nichtinitialisierten Variablen ein Kompilierfehler. Würde man die Anweisung in Zeile 11 einfach auskommentieren, so würde die folgende Fehlermeldung beim Versuch der Kompilierung erscheinen:

```
    /Development/Arithmetik.java:11: variable k might not have been initialized
    ausgabe = "k = " + k;
                       ^
1 error
```

Sie bedeutet, dass in Zeile 11 des Programms dieser Fehler auftritt.

Merkregel 4. Eine Variable muss initialisiert werden. Zuweisungen von nichtinitialisierten Werten ergeben einen Kompilierfehler. Eine Variable wird initialisiert, indem sie ihre erste Anweisung eine Wertzuweisung ist, bei der sie auf der linken Seite steht.

In der Applikation `Arithmetik` wird beispielsweise die Variable `ausgabe` direkt bei ihrer Deklaration initialisiert, hier mit dem leeren String:

```
String ausgabe = "";
```

1.5.3 Datenkonvertierung mit dem *Cast*-Operator

Man kann in einem laufenden Programm einen Wert eines gegebenen Datentyps in einen anderen explizit konvertieren, und zwar mit dem *Cast-Operator* `()`: Durch

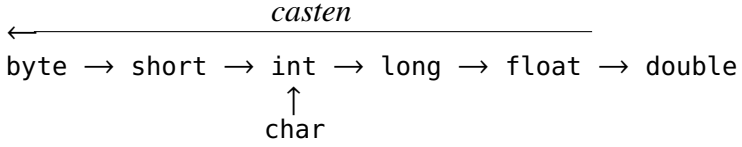
(⟨*Datentyp*⟩) *Ausdruck*

wird der Wert des Ausdrucks in den Typ umgewandelt, der in den Klammern davor eingeschlossen ist. Beispielsweise castet man die Division der beiden **`int`**-Werte `14` und `3` in eine Division von **`double`**-Werten, indem man schreibt `(double) 14 / 3`. In diesem Falle ist die Wirkung exakt wie `14.0 / 3`, man kann jedoch mit dem Cast-Operator auch Werte von Variablen konvertieren. So könnte man durch die beiden Zeilen

```
double x = 3.1415;
int n = (int) x;
```

erreichen, dass die Variable `n` den **`int`**-Wert `3` erhält. Der Cast-Operator kann nicht nur in primitive Datentypen konvertieren, sondern auch zwischen komplexen Datentypen („Objekten“, wie wir später noch kennen lernen werden).

Der Cast-Operator bewirkt eine sogenannte *explizite Typumwandlung*, sie ist zu unterscheiden von der *impliziten* Typumwandlung, die „automatisch“ geschieht.



Ein solcher impliziter Cast geschieht zum Beispiel bei den Anweisungen `double x = 1;` (int → double) oder `int z = 'z';` (char → int).

1.5.4 Operatoren, Operanden, Präzedenz

Tauchen in einer Anweisung mehrere Operatoren auf, so werden sie nach einer durch ihre *Präzedenz* (oder ihrer *Wertigkeit*) festgelegten Reihenfolge ausgeführt. Eine solche Präzedenz ist z.B. „Punkt vor Strich“, wie bei den Integer-Operationen in Zeile 14: Hier ist das Ergebnis wie erwartet `22`. In Tabelle 1.4 sind die arithmetischen Operatoren und ihre Präzedenz aufgelistet.

Operator	Operation	Präzedenz
`()`	Klammern	werden zuerst ausgewertet. Sind Klammern von Klammern umschlossen, werden sie von innen nach außen ausgewertet.
`*`, `/`, `%`	Multiplikation, Division, Modulus	werden als zweites ausgewertet
`+`, `-`	Addition, Subtraktion	werden zuletzt ausgewertet

Tabelle 1.4. Die Präzedenz der arithmetischen Operatoren in Java. Mehrere Operatoren gleicher Präzedenz werden stets von links nach rechts ausgewertet.

Arithmetische Ausdrücke in Java müssen in einzeiliger Form geschrieben werden. D.h., Ausdrücke wie „`a` geteilt durch `b`“ müssen in Java als `a / b` geschrieben werden, so dass alle Konstanten, Variablen und Operatoren in einer Zeile sind. Eine Schreibweise wie $\frac{a}{b}$ ist nicht möglich. Dafür werden Klammern genau wie in algebraischen Termen zur Änderung der Rechenreihenfolge benutzt, so wie in `a * (b + c)`.

Operatoren hängen ab von den Datentypen ihrer Operanden

In Anmerkung (3) sind die Werte für `y` und `z` auch wie erwartet, `3.5`. Aber `x = 3` — Was geht hier vor? Die Antwort wird angedeutet durch die Operationen in Anmerkung (3): Der Operator `/` ist mit zwei Integer-Zahlen als Operanden nämlich eine *Integer-Division* oder eine *ganzzahlige Division* und gibt als Wert eine Integer-Zahl zurück. Und zwar ist das Ergebnis genau die Anzahl, wie oft der Nenner ganzzahlig im Zähler aufgeht, also ergibt `14 / 4` genau `3`. Den Rest der ganzzahligen Division erhält man mit der modulo-Operation `%`. Da mathematisch

$$14/4 = 3 \text{ Rest } 2,$$

ist `14 % 4` eben `2`.

Um nun die uns geläufige Division reeller Zahlen zu erreichen, muss mindestens einer der Operanden eine **double**- oder **float**-Zahl sein. Die Zahl 14 wird zunächst stets als **int**-Zahl interpretiert. Um sie als **double**-Zahl zu markieren, gibt es mehrere Möglichkeiten:

```
double s = 14.0,  t = 14.,  u = 14d,  v = 1.4e1;
```
(1.2)

String als Datentyp und Stringaddition

Wie die dritte der Deklarationen in der Arithmetik-Applikation zeigt, ist ein String, neben den primitiven Datentypen aus Tab. 1.3, ein weiterer Datentyp. Dieser Datentyp ermöglicht die Darstellung eines allgemeinen Textes.

In der Anweisung in Anmerkung (6)

```
ausgabe = "k = " + k;
```
(1.3)

wird auf der rechten Seite eine neue (!) Operation „+“ definiert, die *Stringaddition* oder *Konkatenation* (Aneinanderreihung). Diese Pluszeichen kann nämlich gar nicht die übliche Addition von Zahlen sein, da der erste der beiden Operanden `"k = "` ein String ist! Die Konkatenation zweier Strings `"text1"` und `"text2"` bewirkt, dass ihre Werte einfach aneinander gehängt werden,

```
"text1" + "text2"   ↦   "text1text2"
```
(1.4)

Hier ist die Reihenfolge der Operanden natürlich entscheidend (im Gegensatz zur Addition von Zahlen). Die Stringaddition `"2" + "3"` ein anderes Ergebnis liefert als die Addition `2+3`:

```
"2" + "3" ↦ "23",     2 + 3 ↦ 5.
```
(1.5)

Auch hier, ähnlich wie bei der unterschiedlichen Division von **int** und **double**-Werten, haben zwei verschiedene Operatoren dasselbe Symbol. Das steckt hinter der Definition 1.1.

Eine Eigentümlichkeit ist aber noch nicht geklärt: Was wird denn für `k` eingesetzt? Das ist doch ein **int**-Wert und kein String. Die Antwort ist, dass der zur Laufzeit in dem Speicherbereich für `k` stehende Wert (also `22`) automatisch in einen String umgeformt wird,

```
"k = " + 22   ↦   "k = " + "22"   ↦   "k = 22"
```
(1.6)

Nach Ausführung der Anweisung (6) steht also in der Variablen `augabe` der String `"k = 22"`.

Mit unserem bisherigen Wissen über den Zuweisungsoperator gibt nun der mathematisch gelesen natürlich völlig unsinnige Ausdruck bei Anmerkung (6),

```
ausgabe = ausgabe + "\nx = " + x ... ;
```

einen Sinn: An den alten Wert von `ausgabe` wird der String `"..."` angehängt und als neuer Wert in den Speicherplatz für `ausgabe` gespeichert.

1.5.5 Kombinierte Operatoren

Es gibt eine ganze Reihe von Operatoren, die häufig verwendete Anweisungen verkürzt darstellen. Sie heißen *kombinierte Operatoren* und können Zuweisungsoperatoren oder Operatoren mit einem Operanden (*unäre Operatoren*) sein. Wir geben Sie hier tabellarisch an.

Zuweisungs-operator	Beispiel	Bedeutung	Wert für c, wenn **int** c=11, x=4
+=	`c += x;`	`c = c + x;`	15
-=	`c -= x;`	`c = c - x;`	7
*=	`c *= x;`	`c = c * x;`	44
/=	`c /= x;`	`c = c / x;`	2
%=	`c %= x;`	`c = c % x;`	3

So bewirkt als die Anweisung `c += 4;`, dass der beim Ausführen der Anweisung aktuelle Wert von `c`, beipielsweise 11, um 4 erhöht wird und den alten Wert überschreibt.

Daneben gibt es in Java den *Inkrementoperator* `++` und den *Dekrementoperator* `-`. Wird eine Variable `c` um 1 erhöht, so kann man den Inkrementoperator `c++` oder `++c` anstatt der Ausdrücke `c = c+1` oder `c += 1` verwenden, wie aus der folgenden Tabelle ersichtlich.

Operator	Bezeichnung	Beispiel	Bedeutung
`++`	präinkrement	`++c`	erhöht erst `c` um 1, und führt dann den Rest der Anweisung mit dem neuen Wert aus
`++`	postinkrement	`c++`	führt erst die gesamte Anweisung aus und erhöht *danach* den Wert von `c` um 1
`--`	prädekrement	`--c`	erniedrigt erst `c` um 1, und führt dann den Rest der Anweisung mit dem neuen Wert aus
`--`	postdekrement	`c--`	führt erst die gesamte Anweisung aus und erniedrigt erst dann den Wert von `c` um 1

Besteht insbesondere eine Anweisung nur aus einem dieser Operatoren, so haben Prä- und Postoperatoren dieselbe Wirkung, also: `c++;` $\Longleftrightarrow$ `++c;` und `c-;` $\Longleftrightarrow$ `-c;`. Verwenden Sie also diese Operatoren nur dann in einer längeren Anweisung, wenn Sie das folgende Programmbeispiel sicher verstanden haben!

Da die Operatoren `++` und `-` nur einen Operanden haben, heißen sie „unär“ *(unary)*.

Überlegen Sie, was in dem folgenden Beispielprogramm als `ausgabe` ausgegeben wird.

```
import javax.swing.JOptionPane;
/** Unterschied präinkrement - postinkrement. */
public class Inkrement {
   public static void main( String[] args ) {
     int c;
     String ausgabe = "";
     // postinkrement
     c = 5;
     ausgabe += c + ", ";
     ausgabe += c++ + ", ";
     ausgabe += c + ", ";
     // präinkrement
     c = 5;
     ausgabe += c + ", ";
     ausgabe += ++c + ", ";
     ausgabe += c;
     // Ausgabe des Ergebnisses:
     JOptionPane.showMessageDialog(
       null, ausgabe, "prä- und postinkrement", JOptionPane.PLAIN_MESSAGE
     );
   }
}
```

(Antwort: `"5, 5, 6, 5, 6, 6"`)