KAPITEL 4

Einfache Datenstrukturen – Zahlen und Bool'sche Werte

Ziele:

- Verstehen der Grunddatentypen von Java
- Verstehen von Typkonversion in Java
- Lernen lokale Variablen und Konstanten zu initialisieren
- Verstehen der Speicherorganisation von lokalen Variablen
- Lernen Fallunterscheidungen zu bilden und iterative Programme zu schreiben

Die Binärzahlen, also das Zahlensystem mit dem Computer rechnen, wurde zum ersten Mal von W. Leibnitz in 'Ëxplication de l'Arithmétique Binaire„ beschrieben. Wir wollen im weiteren jedoch nicht auf diese (bitweisen) Operationen eingehen, sondern gleich zur Beschreibung von Zahlen in Programmiersprachen übergehen: Die meisten Programmiersprachen stellen sogenannte Datentypen und deren charakteristische Operationen zur Verfügung. In Java sind dies Datentypen für

- Ganze Zahlen
- Gleitpunktzahlen
- Zeichen
- Boole'sche Werte
- und Felder (später)

Typ	Länge	Wertebereich		
byte	1 Byte	−128	bis	127
short	2 Byte	−32768	bis	32767
int	4 Byte	−2 147 483 648	bis	2 147 483 647
long	8 Byte	−9 223 372 036 854 775 808	bis	9 223 372 036 854 775 807

Tabelle 4.1: Typen ganzer Zahlen in Java

Typ	Länge	Wertebereich
float	4 Byte	bis $\approx 10^{37}$
double	8 Byte	bis $\approx 10^{308}$

Tabelle 4.2: Typen von Gleitpunktzahlen in Java

4.1 Zahlen

Im Rechner werden nur endliche Ausschnitte der Zahlen realisiert.

Ganze Zahlen

Java hat vier Typen ganzer Zahlen, die jeweils 1, 2, 4 und 8 Byte (1 Byte sind 8 Bit) repräsentieren, siehe hierzu Tabelle 4.1. Üblicherweise benützen wir den Typ `int`.

Gleitpunktzahlen

Nach dem IEEE-754–Standard (1985) gibt es zwei Typen von Gleitpunktzahlen, siehe Tabelle 4.2.

Gleitpunktzahlen schreibt man in Java in der Form

```
double:  6.22,   622E-2,   62.2e-1
 float:  6.22F,  622E-2f,  62.2e-1F
```

Der Exponent wird durch „`e`“ oder „`E`“ von der Mantisse getrennt. Eine **float**-Zahl wird durch das Postfix „`f`“ oder „`F`“ identifiziert. Üblicherweise ist die Mantisse normalisiert, d.h. die erste Ziffer vor dem Komma ist 1. Beispiel: $-2.5 \mathrel{\widehat{=}} -1 * 1.01 * 2^1$

Arithmetische Operationen

Die wichtigsten arithmetischen Operationen und Vergleichsoperationen sind in Tabelle 4.3 zu finden. Jeder Kasten enthält Operatoren gleicher Präzedenz. Die Operatoren in den oberen Kästen haben höhere Präzedenz als die in den unteren.

Beispiele: (für Modulo)

Operator	Bedeutung
*	Multiplikation
/	Division
%	Modulo (Rest)
+	Addition
-	Subtraktion
>	größer
>=	größer oder gleich
<	kleiner
<=	kleiner oder gleich
==	gleich
!=	nicht gleich
=	Zuweisung

Tabelle 4.3: Wichtige Operatoren in Java

$$5\%3 = 2$$
$$5.2\%3 = 2.2 \quad \text{bzw. } 5.2\%2.6 = 0.0$$

während für die Division gilt:

$$5/3 = 1$$
$$5.2/3 = 1.733\ldots \quad \text{bzw. } 5.2/2.6 = 2.0$$
$$5.2/3.0 = 1.733\ldots$$

Typkonversion

Java bringt die zahlartigen Datentypen in folgende „Kleiner-Beziehungen“:

byte < **short** < **int** < **long** < **float** < **double**

In Ausdrücken werden Elemente kleinerer Datentypen automatisch in den größeren Typ konvertiert, falls dies nötig ist.

Beispiele:

`1 + 1.7`	ist vom Typ **double**
`1.0f + 1`	ist vom Typ **float**
`1.0f + 1.0`	ist vom Typ **double**

Umgekehrt ist es möglich durch Voranstellen von `(type)` einen Typ vom Typ `type` zu erzwingen, sofern „`type`“ semantisch korrekt ist. Man nennt dies explizite Konversion (engl. TypeCasting)

Beispiele:

`(byte) 3`	ist vom Typ **byte**
`(int)(2.0 + 5.0)`	ist vom Typ **int**
`(float) 1.3e-7`	ist vom Typ **float**

Typcasting kann auch mit der automatischen (Aufwärts-)Konversion verbunden sein:

`(int) 2.0 + 5.0` ist vom Typ **double**,

da Typcasting stärker bindet als binäre Operatoren.

Bei der Typkonversion kann Information verloren gehen. So können z.B. nach der automatischen Konversion von **int** nach **double** Rundungsfehler auftreten. Bei der expliziten Konversion von Gleitpunktzahlen in ganze Zahlen gehen die Dezimalstellen verloren:

```
(int) 5.2 == 5
(int) -5.2 == -5
```

4.2 Zeichen

Der Typ **char** (für character) bezeichnet die Menge der Zeichen aus dem Unicode-Zeichensatz (mit 16 Bit, d.h. 0–65535 Zeichen, siehe `http://www.unicode.org`). Dieser umfasst den sogenannten ASCII-Zeichensatz mit kleinen und großen Buchstaben, Zahlen und verschiedenen Sonderzeichen. Zeichen werden zur Darstellung von Apostrophen umrahmt. Zeichen können in Integer-Werte und Integer-Werte in die entsprechenden Zeichen des Unicode konvertiert werden.

Beispiele:

`(char) 3 == '♡'`	`(int) '7' == 55`
`(char) 48 == '0'`	`(int) 'd' == 100`
`(char) 100 == 'd'`	`(int) '!' == 33`

Bemerkung 1: Obwohl die Elemente von **char** keine Zahlen sind, können sie automatisch in **int** konvertiert werden. Das ist aber schlechter Programmierstil!

Beispiel 4.1 Schlechter Programmierstil

```
int three = 3;
char one = '1';

char four = (char) (three + one);
```

Die Variable `four` in Beispiel 4.1 hat den Typ **char**, obwohl `three + one` den Typ **int** besitzt, der aber durch explizite Konversion in **char** konvertiert wurde.

Bemerkung 2: Der Typ `String` ist kein primitiver Datentyp, sondern ein Objekt. Zeichenketten vom Typ `String` werden in doppelten Anführungszeichen eingeschlossen. Die Konkatenation auf Strings wird mit + notiert. Fast alles wird nach `String` konvertiert:

```
3 + "17" == "317"
```

<table>
<tr><th>Operator</th><th>Bedeutung</th></tr>
<tr><td>!</td><td>strikte Negation</td></tr>
<tr><td>&</td><td>strikte Konjunktion („und“, auch bitweise Addition)</td></tr>
<tr><td>^</td><td>strikte Disjunktion („entweder – oder“)</td></tr>
<tr><td>|</td><td>strikte Adjunktion („oder“)</td></tr>
<tr><td>&&</td><td>sequentielle Konjunktion (≙ *)andalso in SML)</td></tr>
<tr><td>||</td><td>sequentielle Adjunktion (≙ *)orelse in SML)</td></tr>
</table>

Tabelle 4.4: Boole'sche Operationen in Java

^	true	false
true	false	true
false	true	false

(a) entweder – oder

<table>
<tr><th>|, ||</th><th>true</th><th>false</th></tr>
<tr><td>true</td><td>true</td><td>true</td></tr>
<tr><td>false</td><td>true</td><td>false</td></tr>
</table>

(b) oder

Abbildung 4.1: Unterschied „entweder – oder“ und „oder“

Konversionen wie (**int**) "7" sind allerdings nicht möglich!

4.3 Boole'sche Werte

Der Typ **boolean** hat genau zwei Werte, **true** und **false**. Die Boole'schen Operationen mit Präzedenzen in absteigender Reihenfolge sind in Tabelle 4.4 aufgeführt. Beispiel 4.2 und Beispiel 4.3 sollen den Unterschied in der Arbeisweise von sequentiellen und strikten boole'schen Operatoren verdeutlichen.

Beispiel 4.2 Beispiel für die strikte/sequentielle Konjunktion

```
int teiler = 0;
(teiler != 0) && (100/teiler > 1) == false; // Ok
(teiler != 0) & (100/teiler > 1) == false;  // Laufzeitfehler
```

4.4 Deklaration lokaler Variablen und Konstanten

Eine einfache Deklaration lokaler Variablen hat die Form

1. <Type> <VarName>;   bzw.
2. <Type> <VarName> = <Expression>;

Beispiel 4.3 Beispiel für die strikte/sequentielle Adjunktion

```
true || (1/0 == 1) == true; // Ok
true | (1/0 == 1)           // Laufzeitfehler
```

Gleitpunktzahlen	**float**, **double**
unäres Minus	-
Division	/
Modulo (Rest)	%
Konversion nach ganze Zahl	(**int**)
Ganze Zahlen	**int**
Ganzzahldiv.	/
Modulo (Rest)	%
Boole'sche Werte	**boolean**
strikte Konj.	&
sequ. Konj.	&&
strikte Adj.	|
sequ. Adj.	||
strikte Disj.	^
Worte	String
Konkatenation	+

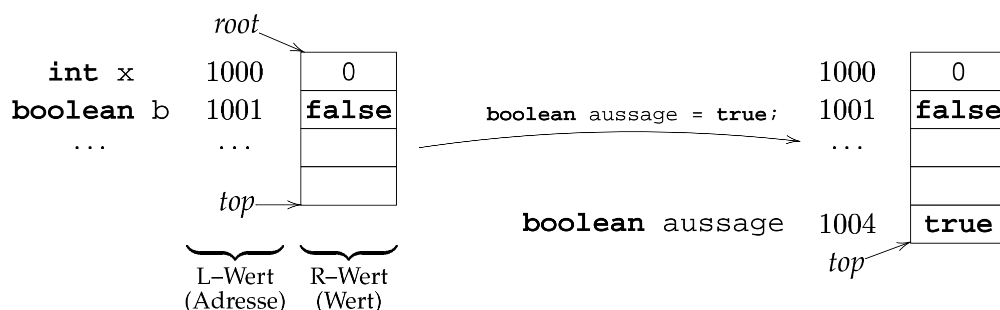
Tabelle 4.5: Wichtige Operationen

Der Ausdruck `<Expression>` gibt den Initialwert der Variable `<VarName>` an. Im Fall 2 auf Seite 21 wird `<VarName>` der Standardwert des Typs `<Type>` als Initialwert zugewiesen. Initialwerte sind z.B.

int, **byte**, **short**	0
float, **double**	0.0f, 0.0

Allerdings dürfen *nur initialisierte* Variablen in Ausdrücken benutzt werden. Deshalb ist es sinnvoll, lokale Variablen sofort zu initialisieren.

Lokale Variablen werden im „Keller" (Stack) gespeichert. Durch die Deklaration wird eine neue Speicherzelle für die lokale Variable reserviert und mit ihrem Initialwert belegt. Das folgende Bild zeigt, wie der Speicher bei Deklaration einer lokalen Variable `aussage` um eine Zelle wächst.



Mehrere Deklarationen lokaler Variablen gleichen Typs können zusammengefaßt werden:

```
TypName VarName1 = Ausdruck1, VarName2,
        VarName3 = Ausdruck3;
```

ist eine Abkürzung für

```
TypName VarName1 = Ausdruck1;
TypName VarName2;
TypName VarName3 = Ausdruck3;
```

Beispiel 4.4 Variablendeklaration

```
int total = 17, max = 100, i, j;
```

ist eine Abkürzung für

```
int total = 17;
int max = 100;
int i;
int j;
```

Insgesamt ergibt sich folgende Syntax:

LocalVarDeclaration :=
Type VarName ["=" Expression] { "," VarName ["=" Expression]} ";"

In der Java-Spezifikation werden Deklarationen lokaler Variablen folgendermaßen ausgedrückt (vereinfacht):

VariableDeclarator :=
VarName |
VarName "=" Expression

VariableDeclarators :=
VariableDeclarator |
VariableDeclarator "," VariableDeclarators

LocalVariableDeclaration :=
Type VariableDeclarators ";"

Typnamen sind Spezialfälle von „*Type*“. Zum Beispiel ist `String[]` ein *Type*, aber kein Typname.

Eine Konstante wird durch Angabe des „Modifiers“ **`final`** deklariert. Zum Beispiel bezeichnet

```
final int TOTAL = 100;
```

eine Konstante mit Wert 100. Konstanten werden i.a. mit Großbuchstaben geschrieben und sollten (wie auch Variablen) „sprechende“ Namen besitzen. Außerdem sollten in einem Programm Konstanten immer auch als Konstanten deklariert werden und nicht als reine Zahlenwerte gegeben sein. Verwenden Sie *nie* „Magic Numbers“, d.h. Zahlen im Programm, die eine spezielle Bedeutung haben! Beispiele:

- Anstelle von 365 im Programm für „Anzahl der Tage im Jahr“ verwende man besser `final int TAGE_PRO_JAHR = 365;`
- Für die mathematischen Größen π und e verwende man anstelle von 3.14159 und 2.7182 besser `Math.Pi` bzw. `Math.E`

Syntax

ConstantDeclaration ::=
"`final`" LocalVariableDeclaration

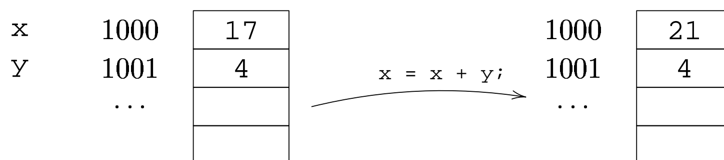
4.5 Zuweisung, sequentielle Komposition und Block

Eine Zuweisung wird in Java mit „`=`“ geschrieben (im Gegensatz zu Pascal, Modula etc., wo „`:=`“ verwendet wird).

```
<VarName> = <Ausdruck>;
```

bedeutet, daß der Variablen mit Name `<VarName>` der Wert von `<Ausdruck>` als R–Wert zugewiesen wird.

Beispiel:



Bemerkung: Später werden wir sehen, daß auf der rechten Seite der Zuweisung allgemeinere Ausdrücke stehen können.

Für manche spezielle Zuweisungen gibt es Abkürzungen, die von der Programmiersprache C übernommen wurden. Sei `x` eine Variable von numerischem Typ und `op` eine binäre Operation. Dann gibt es folgende Abkürzungen:

`x++;`	steht für	`x = x + 1;`
`x--;`	steht für	`x = x - 1;`
`x op= <Ausdruck>;`	steht für	`x = x op <Ausdruck>;`

Beispiele:

`x += y;`	steht für	`x = x + y;`
`x &&= c;`	steht für	`b = b && c;`
`x += 3*y;`	steht für	`x = x + 3*y;`

Syntax

Assignment :=
Name "=" *Expression* ";"

Sequentielle Komposition in Java wird durch Hintereinanderschreiben ausgedrückt:

Statements :=
Statement | *Statement Statements*

wobei bisher

Statement :=
LocalVariableDeclaration | *ConstantDeclaration* | *Assignment*

Beispiel 4.5 Sequentielle Komposition in Java

```
int total = 100;
total = total + 100;
```

Doppeldeklarationen von Variablen sind *nicht* erlaubt!

Mehrere Anweisungen können durch geschweifte Klammern zu einem *Block* zusammengefügt werden.

BlockStatement :=
"{"Statements "}"

wobei ein Block wieder selbst als Anweisung betrachtet wird:

Statement := BlockStatement

Der Gültigkeitsbereich einer lokalen Variablen oder Konstante ist der die Deklaration umfaßende Block. Außerhalb dieses Blocks existiert die Variable *nicht*! Geschachtelte Doppeldeklarationen sind (im Gegensatz zu Modulen) verboten.

Beispiel 4.6 Gültigkeitsbereiche

```
{
    int wert = 0;
    wert = wert + 17;
    {
        int total = 100;
        wert = wert - total;
    }
    wert = 2 * wert;
}
```

Der Gültigkeitsbereich von `wert` in Beispiel 4.6 erstreckt sich von Zeile 2 bis Zeile 9, derjenige von `total` von Zeile 5 bis 7.

4.6 Fallunterscheidung

Die Fallunterscheidung in Java hat die Form

`if` (Boolescher Ausdruck) *Statement*

bzw.

`if` (Boolescher Ausdruck) *Statement* **`else`** *Statement*

Falls mehrere Anweisungen in der Fallunterscheidung vorkommen sollten, faßt man diese in einem Block zusammen. Die Blockklammern in Beispiel 4.7 sind sehr wichtig! Will man z.B. auch im **`else`**–Zweig mehrere Anweisungen verwenden und vergißt die Blockklammer, so erhält man meist falsche Ergebnisse, wie dies in Beispiel 4.8 auf der nächsten Seite zu sehen ist. Hier wird die letzte Zeile ausgeführt, auch wenn der Kontostand den abgehobenen Betrag überschreitet! Deshalb muß geklammert werden, die nötige Korrektur ist in Beispiel 4.9 auf der nächsten Seite zu finden.

Bemerkung: Falls das Programm später geändert wird, ist empfohlen, die Blockklammern immer zu verwenden.

Mehrere Fallunterscheidungen hat Beispiel 4.10 auf Seite 28. Hier ist es wichtig,

Beispiel 4.7 mehrere Anweisungen in einer Fallunterscheidung

```
if (kontoStand >= betrag)
{
    double neuerStand = kontoStand - betrag;
    kontoStand = neuerStand;
}
else
{
    kontoStand = kontoStand - betrag - UEBERZIEH_GEBUEHR;
}
```

Beispiel 4.8 mehrere Anweisungen II

```
if (kontoStand >= betrag)
{
    double neuerStand = kontoStand - betrag;
    kontoStand = neuerStand;
}
else
    kontoStand = kontoStand - betrag - UEBERZIEH_GEBUEHR;
    gebuehren += UEBERZIEH_GEBUEHR;
```

Beispiel 4.9 mehrere Anweisungen III

```
if (kontoStand >= betrag)
{
    double neuerStand = kontoStand - betrag;
    kontoStand = neuerStand;
}
else
{
    kontoStand = kontoStand - betrag - UEBERZIEH_GEBUEHR;
    gebuehren += UEBERZIEH_GEBUEHR;
}
```

Beispiel 4.10 Prozessorleistung

```
if (leistung < 200)
    System.out.println(leistung + " MHz ist zu langsam");
else if (leistung < 350)
    System.out.println(leistung + " MHz schnell genug");
else if (leistung <= 550)
    System.out.println(leistung + " MHz ist sehr schnell");
else
    System.out.println("Hat Ihr System wirklich die Leistung " +
        leistung + " MHz?");
```

„**else**“ zu verwenden, da z.B. für `leistung == 400` (MHz) alle drei ersten Alternativen zutreffen!

Bei Java wird immer das **else** zum nächsten **if** gebunden, d.h.

```
if (c1) if (c2) S1 else S2
```

und

```
if (c1) { if (c2) S1 else S2 }
```

sind äquivalent. Innerhalb einer Fallunterscheidung ist kein „kurzes **if**“ (**if** ohne **else**–Zweig) als Anweisung erlaubt!

Syntax

ShortIfStatement :=
"`if`" "`(`" *Expression* "`)`" *Statement*[1]
IfStatement :=
ShortIfStatement "`else`" *Statement*[1]

wobei Expression vom Typ **boolean** ist. Dadurch wird *Statement* erweitert durch

Statement :=
ShortIfStatement | *IfStatement*

4.7 Iteration

In Java (wie in den meisten imperativen Systemen) gibt es drei Konstrukte zur Iteration: **while**-, **for**- und **do**-Schleifen.

4.7.1 while-Schleifen

Die **while**-Schleife hat die Form

while (Boolescher Ausdruck)
Statement

[1] hier darf *Statement kein ShortIfStatement* sein!

wobei *Statement* meist ein Block ist. Solange der Boolesche Ausdruck den Wert **`true`** hat, wird das Statement ausgeführt; Beispiele für **`while`** sind Beispiel 4.11 und Beispiel 4.12.

Beispiel 4.11 10mal „`tick`" drucken

```
int n = 1, end = 10;
while (n <= end)
{
    System.out.println("tick" + n);
    n++;
}
```

Beispiel 4.12 Quersumme von x

```
int qs = 0, x = 352;
while (x > 0)
{
    qs = qs + x % 10;
    x = x / 10;
}
```

Syntax

WhileStatement :=
"`while`" "`(`" *Expression* "`)`" *Statement*

wobei *Expression* vom Typ **`boolean`** ist. Innerhalb eines **`while`** darf Statement kein „kurzes **`if`**" sein (sonst besteht die Gefahr eines sogenannten Dangling **`else`**).

4.7.2 `for`-Schleifen

Die häufigste Form der **`while`**-Schleife ist

```
i = start;
while (i < end)
{
    ...
    i++
}
```

Dies kann durch eine **`for`**-Schleife abgekürzt werden:

```
for (i = start; i <= end; i++)
{
    ...
}
```

Beispiel 4.13 „`tick`“ mit einer **`for`**-Schleife

```
int end = 10;
for (int n = 1; n <= end; n++)
{
    System.out.println("tick" + n);
}
```

Allgemein hat eine **`for`**-Schleife die Gestalt

> **`for`** (Initialisierung ; Bedingung ; Zählerkorrektur ;)
> *Statement*

Dabei wird zunächst die Initialisierung ausgeführt. Dann wird *Statement* ausgeführt und der Zähler geändert, solange die Bedingung wahr ist. Guter Stil ist es, **`for`**-Schleifen nur folgendermaßen zu schreiben:

```
for (setze counter auf start;
     Test, ob counter bei end;
     aendere counter)
{
    ... // counter, start, end und increment werden
        // hier nicht geaendert!
}
```

Außerdem ist es sinnvoll, den Zähler `counter` in der Initialisierung zu deklarieren. Dann ist `counter` lokal für die **`for`**-Anweisung und außerhalb nicht definiert.

4.7.3 `do`-Schleifen

Die **`do`**-Schleife ist eine **`while`**-Schleife, bei der die Anweisung mindestens einmal aufgeführt wird. Die Bedingung wird erst nach Ausführung der Anweisung überprüft. Sie hat die Form

> **`do`**
> *Statement*
> **`while`** (Boolescher Ausdruck)

4.8 Zusammenfassung

1. Java besitzt 4 Grunddatentypen für ganze Zahlen (**`byte`**, **`short`**, **`int`**, **`long`**) und 2 Grunddatentypen für Gleitpunktzahlen (**`float`**, **`double`**). Dazu kommen noch **`boolean`** und **`char`**. `String` ist *kein* Grunddatentyp.
2. Java hat eine automatische Konversion in den „größeren“ Grunddatentyp. Konversion in einen „kleineren“ Datentyp geschieht explizit durch Typcasting.

3. Eine Fallunterscheidung erlaubt es, abhängig von einer Bedingung, verschiedene Anweisungen auszuführen.

4. Zur Vermeidung des Dangling-**`else`**-Problems dürfen „Ja"- und „Nein"-Zweig von **`if`** sowie die Anweisung von **`while`** kein „Short-**`if`**" sein.

5. Eine Iteration (Schleife) dient zur mehrfachen Ausführung eines Blocks von Anweisungen. Die Terminierungsbedinung kontrolliert, wie häufig der Block ausgeführt wird.

6. Es gibt 3 Arten von Iterationen: **`while`**-, **`for`**- und **`do`**-Schleifen. **`for`**-Schleifen sollten verwendet werden, wenn die Schleifenvariable von einem Anfangswert bis zu einem Endwert mit einem *konstanten* Inkrement oder Dekrement läuft; **`do`**-Schleifen sind passend, wenn der Schleifenrumpf mindestens einmal ausgeführt werden muß.