Java-Tutorial

– Programmieren lernen –

Karsten Brodmann

www.punkt-akademie.de

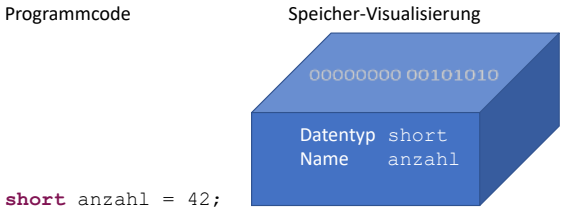
(Folge 8 - 28.12.2022)

Variablen und primitive Datentypen

Variablen sind benannte Speicherstellen, deren Inhalte gemäß Ihrer vereinbarten Datentypen interpretiert werden. Java unterstützt folgende integrierte Datentypen, die zur Sprachdefinition von Java gehören. Man nennt sie auch *primitive Datentypen*.

Typ	**Wertebereich**	**Art**	**Bits**
`boolean`	`true` oder `false`	Wahrheitswert	1
`char`	Unicode-Zeichen	Zeichen	16
`byte`	$-128..127$	Ganzzahl	8
`short`	$-32768..32767$	Ganzzahl	16
`int`	$-2147483648..2147483647$	Ganzzahl	32
`long`	$-9223372036854775808..9223372036854775807$	Ganzzahl	64
`float`	$\pm 1,4E{-}45..\pm 3,4E{+}38$	Gleitkommazahl	32
`double`	$\pm 4,9E{-}324..\pm 1,7E{+}308$	Gleitkommazahl	64

Visuell können Sie sich die Vereinbarung einer Variablen eines primitiven Datentyps wie eine Schachtel vorstellen:



Entsprechend des vereinbarten Datentypen wird ein Speicherbereich, eine Schachtel, rerserviert Auf der steht, welche Variable von welchem Typ darin abgelegt ist. Aufgrund des

Datentyps weiß Java, wie das abgelegt Bitmuster zu interpretieren ist. Adressiert wird der Speicherbereich über den Namen der vergebenen Variablen.

Ganze Zahlen

Alle ganzen Zahlen sind in Java vorzeichenbehaftet. Wird der Wertebereich überschritten, sind die Rechenergebnisse mit ganzen Zahlen falsch – zumindest vordergründig betrachet. Sofern Sie sich mit Restwertarithmetik auskennen, erkennen Sie jedoch eine Gesetzmäßigkeit. Würden Sie in dieser Arithmetik rechnen wollen, stellten Sie fest, dass Java genau diese anwendet. In Bezug auf die übliche Schularithmetik ist der Wertebereich ganzer Zahlen unbedingt einzuhalten, wollen sie keine falschen Rechenergebnisse riskieren.

Die Division durch 0 führt zu einer Ausnahme. Wird diese nicht behandelt, wird ein Programm bei Division durch 0 abgebrochen.

Literale werden üblicherweise in der gewohnten Dezimalschreibweise notiert. Alternativ können sie auch als Oktal-, Hexadezimal- oder Binärzahlen geschrieben werden.

Zahlensystem	**Präfix**	**gültige Zeichen**
oktal	`0`	0...7
hexadezimal	`0x` oder `0X`	0...9, a...f bzw. A...F
binär	`0b` oder `0B`	0, 1

Üblicherweise interpretiert Java ganzzahlige Zahlenliterale als `int`. Insbesondere liefert die ganzzahlige Arithmetik Werte vom Datentyp `int`. Ist etwas anderen gewollt, so muss der Wert entweder konvertiert oder entsprechend gekennzeichnet werden. Für den Datentyp `long` ist ein `L` oder `l` anzuhängen. Beispiel: `42L`. Für `short` und `byte` existiert keine Kennzeichnung. Hier muss(te) entsprechend konvertiert werden. Das geschieht durch das Voranstellen des Zieldatentyps, der in Klammern eingefasst wird. Beispiel: `(byte)(42+1)`. In aktuellen Java-Versionen kann dieser explizite Typecast weitestgehend entfallen.

Operator	**Bedeutung**	**Operator**	**Bedeutung**
`+`	Addition	`<<`	Linksshift
`-`	Subtraktion	`>>`	Rechtsshift
`*`	Multiplikation	`>>>`	Rechtsshift ohne Vorzeichen
`/`	ganzzahlige Division	`~`	bitweise Negation
`%`	Divisionsrest (Modulo)	`-`	Vorzeichenumkehr
`&`	bitweises AND	`++`	Pre-/Post-Inkrement
`|`	bitweises OR	`--`	Pre-/Post-Dekrement
`^`	bitweises XOR		

Tabelle 1: Operatoren für ganze Zahlen

Java beachtet die üblichen mathematischen Regeln bezüglich der Auswertungsreihenfolge der Operatoren. Nötigenfalls, wenn Sie diese ändern wollen, verwenden Sie Klammern.

Gleitkommazahlen

Gleitkommaliterale werden in Java in der amerikanischen Schreibweise notiert. Statt eines Dezimalkommas wird ein Punkt geschrieben. Parallel hierzu ist die wissenschaftliche Noation möglich, wie beispielsweise `1.0E2.0`. Das `E` darf auch als kleines `e` geschrieben werden. Es ist zu lesen als „*mal 10 hoch*“.

Java interpretiert Gleitkommaliterale als `double`. Sollen sie explizit als `float` interpretiert werden, so ist der Zahl ein `F` oder `f` anzufügen. Beispiel: `42.0f`.

Auf den Gleitkommazahlen sind die folgenden Operationen mit den gezeigten Operatoren definiert.

Operator	**Bedeutung**
+	Addition
−	Subtraktion
⋆	Multiplikation
/	Gleitkommadivision
++	Inkrement
−−	Dekrement

Gleitkommazahlen in Java entsprechen dem IEEE-754-Standard. Daraus ergeben sich einige Eigenschaften, die mit den Gewohnheiten und dem Wissen, welches Sie im Mathematikunterricht in der Schule erworben haben, kollidieren.

Es ist zu beachten, dass sowohl Darstellung als auch Berechnungen mit Gleitkommazahlen nur approximativ genau sind. Für die übliche Arithmetik ist die Genauigkeit hinreichend.

Es gibt zwei ausgezeichnete Werte: `NaN` (Not a Number) und `Infinity` (unendlich).

Bei den Gleitkommazahlen ist die Division durch 0 erlaubt, im Gegensatz zu den ganzen Zahlen. Wird eine Gleitkommazahl, die ungleich 0 ist, durch 0 geteilt, ist das Ergebnis `Infinity`. Das entspricht folgender Grenzwertbetrachtung:

$$\lim_{x \to 0} \frac{y}{x} = \pm\infty \text{ für } x, y \in \mathbb{R} : y \neq 0$$

0 geteilt durch 0 ergibt jedoch `NaN`.

Ebenso gewöhnungsbedürftig ist es, dass bei Gleitkommazahlen kein Überlauf eintritt, wenn deren Wertebereich überschritten wird. Sie erhalten dann den Wert `Infinity`. Das gilt im Positiven wie auch im Negativen.

Hat eine Variable den Wert `NaN` oder `Infinity` kann dieser durch Addieren oder einer anderen arithmetischen Operation auch nicht geändert werden. Etwas von unendlich abzuziehen oder ähnliches bleibt unendlich. Sie können der Variablen aber wieder einen anderen konkreten Wert zuweisen.

Gemischte Zahlenausdrücke

Bei Audrücken, die unterschiedliche Zahlendatentypen beinhalten, werden die Operanden vor Durchführung der Operationen automatisch in den jeweils größeren Datentypen konvertiert. Das Ergebnis entspricht dann dem größten beteiligten Datentypen.

```
int x = 1;
double y = 41.0;
double z;

StdOut.print(x+y);  // auch ohne Zuweisung: Ergebnis ist double
z = x + y;          // hier ist es klar...: Ergebnis ist double
```

Bei `x+y` wird das `x` implizit zu `1.0` gewandelt. Erst danach erfolgt die Addition.

Wahrheitswerte

Wahrheitswerte werden durch den Datentyp `boolean` repräsentiert und zum Speichern logischer Werte verwendet. Das sind die ausschließlichen Werte `true` (wahr) und `false` (falsch).

Vergleiche haben immer ein boolesches Ergebnis. Vergleichsoperatoren in Java sind: <, <=, ==, >, >= und !=.

Operator	Bedeutung
`&&`	logisches AND mit verkürzter Auswertung
`||`	logisches OR mit verkürzter Auswertung
`&`	logisches AND mit vollstündiger Auswertung
`|`	logisches OR mit vollstündiger Auswertung
`^`	XOR (exklusives OR, eigentlich Bit-Operation)
`==`	Gleichheit
`!=`	Ungleichheit
`!`	Negation

Tabelle 2: Boolesche Operatoren

Die Auswertungsreihenfolge entspricht der, die Sie aus dem Mathematikunterricht kennen. Wollen Sie sie modifizieren, verwenden Sie Klammern.

Mihilfe logischer Verknüpfungen können Sie Bedingungsausdrücke aus mehreren Teilbedingungen aufbauen. Die Verknüpfung logischer Werte und deren Ergebnis lässt sich am einfachsten in Form von Wahrheitstabellen darstellen. Seien `A` und `B` logische Ausdrücke, dann gilt:

A	B	A && B	A || B	A ^ B	!A
false	false	false	false	false	true
false	true	false	true	true	true
true	false	false	true	true	false
true	true	true	true	false	false

Tabelle 3: Wahrheitstabelle

Die verkürzte Auswertung erfolgt von links nach rechts und bricht frühestmöglich ab, sobald das Resultat klar ist. Beispiel:

```
while ((t > 0) && (n%t != b))
    t -= 1;
```

Wenn `t <= 0` gilt, wird der Rest der Bedingung, bei der verkürzten Auswertung, nicht mehr ausgewertet.

Anmerkung: Die meisten Programmiersprachen verwenden als Codierng für Wahrheitswerte die ganzzahligen Werte 0 und 1, um die beiden möglichen Zustände zu implementieren. In Java sind Wahrheitswerte ein eigentständiger Datentyp, der im Gebrauch vollkommen losgelöst von ganzen Zahlen ist. Dadurch können versehentliche Zuweisungen numerischer Datentypen in Bedingungen direkt als Fehler erkannt werden.

```
int x = 1;
...
if (x = 10) {
    ...
}
```

Der Programmcode wird in Java als fehlerhaft erkannt. In der `if`-Bedingung hätte `x == 10` formuliert werden müssen.

Zeichen

Java verwendet intern 16 Bit große Unicode-Zeichen. Jedes Zeichen wird also durch zwei Byte kodiert. Die niederwertigen 7 Bits stimmen mit dem ASCII-Zeichensatz überein.

Zeichenliterale werden, jeweils einzeln, paarweise in einfache Anführungszeichen eingeschlossen. Verschiedene Sonderzeichen können durch Escape-Seqenzen ausgedrückt werden.

Escape-Sequenz	Bedeutung
\a	bell
\b	backspace
\t	horizontal tab
\v	vertical tab
\n	newline
\r	carriage return
\f	form feed
\101	Buchstabe A in Oktalnotation (andere analog)
\u0041	Buchstabe A in Hexadezimalnotation (andere analog)
\"	doppelte Anführungszeichen
\'	einfaches Anführungszeichen
\\	backslash

Tabelle 4: Escape-Seqenzen

Zeichen lassen sich mit den üblichen Vergleichsoperatoren vergleichen: <, <=, ==, >, >=, !=. Das Ergebnis eiens Vergleichs ist ein Wahrheitswert.

Die Anordnung/Reihenfolge der Zeichen für Vergleiche ergibt sich aus deren Kodierung. In allen Zeichensätzen der Welt gilt:

```
...< 0 < 1 <...< 9 <...< A < B <...< Z <...< a < b <...< z <...
```

Hier ist eine ASCII-Tabelle der darstellbaren Zeichen und deren jeweiliger Kodierung, die mit dem Programm `ASCII.java` (siehe Folgeseite) erstellt wurde.

```
 32:     33: !   34: "   35: #   36: $   37: %   38: &   39: '
 40: (   41: )   42: *   43: +   44: ,   45: -   46: .   47: /
 48: 0   49: 1   50: 2   51: 3   52: 4   53: 5   54: 6   55: 7
 56: 8   57: 9   58: :   59: ;   60: <   61: =   62: >   63: ?
 64: @   65: A   66: B   67: C   68: D   69: E   70: F   71: G
 72: H   73: I   74: J   75: K   76: L   77: M   78: N   79: O
 80: P   81: Q   82: R   83: S   84: T   85: U   86: V   87: W
 88: X   89: Y   90: Z   91: [   92: \   93: ]   94: ^   95: _
 96: `   97: a   98: b   99: c  100: d  101: e  102: f  103: g
104: h  105: i  106: j  107: k  108: l  109: m  110: n  111: o
112: p  113: q  114: r  115: s  116: t  117: u  118: v  119: w
120: x  121: y  122: z  123: {  124: |  125: }  126: ~  127:
```

Das Zeichen mit der Kodierung 32 ist das Leerzeichen. Hinter der Kodierung 127 verbirgt sich DELETE. Dieses Zeichen ist auf dem Bildschirm nicht sichtbar.

Beachten Sie, wie Zeichen in ganze Zahlen umgewandelt werden können und umgekehrt. Im gezeigten Programm besorgt das eine Formatierungsangabe. Hier wurde der Formatspezifizier `%c` in `printf` genutzt. Er akzeptiert ein Zeichen oder eine ganze Zahl. Letztere

wird durch ihn automatisch in ein Zeichen konvertiert. Sie können das aber auch manuell selbst tun. `(char)65` liefert Ihnen das Zeichen A. Umgekehrt wandelt `(int)'A'` das Zeichen A in den dezimalen Wert 65.

```
1  import de.pakad.tools.StdOut;
2
3  /**
4       * Ausgabe einer ASCII-Tabelle.
5       *
6       * <p>Die Zeichen mit den Codes 32 bis einschließlich 127
7       * werden in einer mehrspaltigen Tabelle paarweise (Code: Zeichen)
8       * ausgegeben.</p>
9       *
10      * @author Karsten Brodmann
11      */
12 public class ASCII {
13      /**
14           * @param args nicht verwendet
15           */
16      public static void main(String[] args) {
17           /* Der Zeichencode errechnet sich als:
18            *            (Zeilenindex*8) + Spaltenindex
19            *
20            * Das erste Zeichen ist das Zeichen mit dem Code 32, weil
21            * die Spaltenanzahl auf 8 festgesetzt und die Zeilen
22            * mit 4 beginnen.
23            */
24           short i = 4;          // Zeilenindex
25           while(i <= 15) {      // über alle Zeilen...
26                short j = 0;     // Spaltenindex
27                while(j <= 7) {  // über alle Spalten...
28                     StdOut.printf("%3d: %c   ", (i*8)+j, (i*8)+j);
29                     j++;
30                }
31                StdOut.println();
32                i++;
33           }
34      }
35 }
```

ASCII.java

Anmerkung: Die `while`-Schleife ist hier eigentlich ungeschickt. Man sollte besser eine `for`-Schleife verwenden. Die haben wir aber noch nicht behandelt.

Die in Zeile 26 gesetzten Klammern um `(i*8)+j` dürfen entfallen. Aufgrund desen Java „Punkt vor Strichrechnung“ berücksichtigt, sind sie redundant. Ich habe sie dennoch gesetzt, weil der Code, wie ich meine, auf diese Weise leichter lesbar und verständlich ist.

Zeichenketten

Der Datentyp für Zeichenketten heißt in Java `String`. Das ist kein primitiver Datentyp. Allerdings genießen die Zeichenketten in Java, weil sie so häufig genutzt werden, einen Sonderstatus. Sie lassen sich vielfach so einfach wie ein primitiver Datentyp nutzen.

Zeichenkettenliterale werden in doppelte Anführungszeichen eingeschlossen. Zum Verketten/Aneinanderfügen von Zeichenketten ist der +-Operator in Java überladen. Er dient also nicht ausschließlich dazu arithmetische Operationen auszuführen.

Zeichenketten sind Objekte. Sie haben daher Methoden. Das sind Routinen, die auf Zeichenketten operieren. So gibt es beispielsweise eine Methode `length`, welche die Länge einer Zeichenkette zurückliefert. Es gibt noch diverse weitere Methoden. Schauen Sie sich die Klasse `String` in der Java-Dokumentation an.

Ganz wichtig zu bemerken ist: **Zeichenketten sind unveränderlich (immutable)!**

Eine Zeichenkette kann leer sein, im Gegensatz zu einem Zeichen.

Standardwerte

Java ist bemüht, nicht initialisierte Variablen zu vermeiden. Der lesende Zugriff auf solche Variablen wäre/ist ein Fehler. Der Java-Compiler versucht deshalb nicht initialisierte Variablen mit Standardwerten zu belegen.

Typ	**Standardwert**
`boolean`	`false`
`char`	`'\u0000'`
`byte`	`0`
`short`	`0`
`int`	`0`
`long`	`0L`
`float`	`0.0f`
`double`	`0.0`
`String`	`""`

Konstanten

Gelegentlich benötigen wir konstante Werte in einem Programm. Sei dies als Beispiel der Mehrwertsteuersatz von 19%. Er soll als Faktor `0.19` im Programm verwendet werden.

Der erste Gedanke könnte sein, an allen Stellen, wo eine Mehrwertsteuer berechnet werden soll, diesen Faktor zu notieren. Was passiert aber, wenn sich der Steuersatz ändert? – An mehreren Stellen wäre `0.19` durch den neuen Wert zu ersetzen. Mit Suchen und Ersetzen

kann das leicht durchgeführt werden. Der Teufel steckt aber im Detail. Wird die gleiche Konstante in einem anderen Kontext verwendet, soll aber nicht geändert werden, dann funktioniert das einfache Suchen und Ersetzen nicht mehr.

Eine bessere Idee könnte es daher sein, eine Variable `double tax = 0.19` zu vereinbaren und an den erforderlichen Stellen die Variable einzusetzen. Die wäre allerdings im Programmcode (versehentlich) veränderbar. Um das zu unterbinden, können wir in Java *Konstanten* definieren. Das sind „finalisierte Variablen“. Üblicherweise schreibt man die Bezeichner dann groß.

```
final double TAX = 0.19;
```

Setzt man diese Konstante nun überall im Programmcode ein, wo der Steuersatz erforderlich ist, kann man ihn an zentraler Stelle einfach ändern. Ein versehentliches Zuweisen im Code wird unterbunden. Konstanten können nur einmalig, bei deren Inirialisierung, einen Wert zugewiesen bekommen. Zusätzlich verrät die Großschreibung, es hier mit einer Konstanten zu tun zu haben, was die semantische Bedeutung unterstreicht.