Einführung in die Programmierung mit Java

Teil 2: Fundamentale Datentypen

Martin Hofmann   Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

19. Oktober 2017

Teil 2: Fundamentale Datentypen

1. Fundamentale Datentypen
   - Die Datentypen int, double
   - Variablen und Zuweisung
   - Zeichenketten
2. Fallunterscheidungen
   - Basisdatentyp boolean
3. Zusammenfassung

Münzwerte

```
public class Muenzen1
{
    public static void main(String[] args) {
        int zehnerl   = 8; // Anzahl 10 ct Muenzen
        int zwanzgerl = 4; // Anzahl 20 ct Muenzen
        int fuchzgerl = 3; // Anzahl 50 ct Muenzen

        double gesamt = zehnerl * 0.10 +
            zwanzgerl * 0.20 + fuchzgerl * 0.50;

        System.out.print("Gesamtwert = ");
        System.out.println(gesamt);
    }
}
```

Erinnerung: Alles hinter `//` bis zum Zeilenende wird ignoriert.
Alternative zu Kommentaren mit `/* ... */`

Die Typen `int` und `double`

Ganze Zahlen bilden den Typ `int`;
Fließkommazahlen den Typ `double`.

In Java wird `int` automatisch in `double` konvertiert.

Variablen

Das Statement

```
int zehnerl = 8;
```

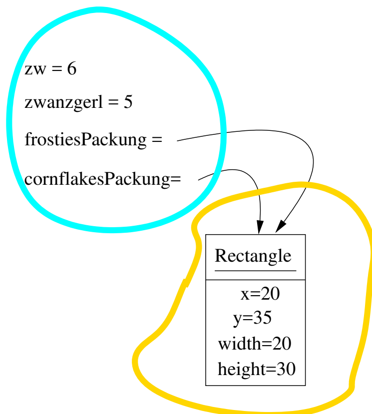
deklariert eine ganzzahlige **Variable** des Typs `int` mit dem Wert 8.
Man kann in Java einer schon deklarierten Variablen neue Werte zuweisen:

```
zwanzgerl = 5;
int zw = zwanzgerl;
zw = 6;
System.out.print("Wert von \"zwanzgerl\": ");
System.out.println(zwanzgerl);
System.out.print("Wert von \"zw\": ");
System.out.println(zw);
```

Was wird gedruckt?

```
Wert von "zwanzgerl": 5
Wert von "zw": 6
```

Der Grund ist, dass Integer- und Double-Variablen keine Verweise sind (wie Objektvariablen) sondern den jeweiligen Wert *direkt* enthalten.
Mit anderen Worten: eine Integer-Variable enthält einen Integer-Wert, eine Objekt-Variable enthält eine Speicheradresse (unter der sich ein Objekt befindet).



Fiktiver Speicherzustand bestehend aus Stack und Heap.

Stack und Heap

- Der von Java verwendete Speicherbereich ist immer in zwei Bereiche aufgeteilt:
- Den Stack (Stapel, Keller) und den Heap (Halde).
- Im Stack befinden sich die Werte der Programmvariablen, also Integerwerte und Speicheradressen von Objekten
- Typen von Variablen deren Werte direkt abgespeichert werden beginnen in Java meist mit Kleinbuchstaben.
- Im Heap befinden sich die Objekte samt den in ihnen enthalten Daten, wie z.B. Breite (width).
- Grafisch repräsentieren wir den Stack als untereinandergeschriebene Liste von Wertbindungen. (`zwanzgerl = 5`).

Stack und Heap

- Speicheradressen werden hierbei nicht explizit als Bitmuster sondern durch Pfeile in den Heap dargestellt.
- Die Objekte im Heap werden als Rechtecke dargestellt, die im oberen Teil den Klassennamen enthalten.
  NB Auch eine Klasse `Circle` würde durch ein Rechteck dargestellt.
- Die Daten in einem Objekt, wie `width` werden ähnlich wie im Stack dargestellt.
- Später lernen wir auch Objekte kennen, die in ihren Daten selbst wieder Objektverweise enthalten. Diese werden dann auch als Pfeile in den Heap dargestellt.

Initialisierung

Man muss Variablen nicht initialisieren:

```
int a;
int b = 4;
a = b + 2;
```

Sie müssen aber vor der ersten Verwendung einen Wert bekommen:

```
int a;
int b = 4;
System.out.println(a);
```

ist ein Programmierfehler, den der Java Compiler erkennt.

Mit `Rectangle` geht dies ganz analog:

```
Rectangle r1 = new Rectangle(10,20,30,40);
```

ist identisch zu

```
Rectangle r1;
r1 = new Rectangle(10,20,30,40);
```

Nochmal Wertzuweisung

Man kann auch schreiben:

```
a = a + 1;
```

Dadurch wird der Wert von `a` um eins erhöht.
Manche Programmierer verwenden dafür die Kurzform

```
a++;
```

Daher auch der Name C++ für „Nachfolger von C“.

Rechengenauigkeit

Es gibt es nur 2^{32} verschiedene int-Werte. Ebenso hat `double` begrenzte Genauigkeit.

Es gibt den Datentyp `long`, der insgesamt 2^{64} verschiedene Werte aufweist. Long-Konstanten schreibt man so: `1340000000000000L`.

Schließlich gibt es die Klassen `BigInteger` und `BigDecimal` die praktisch unlimitiert sind. Sie repräsentieren Zahlen als Listen von Ziffern.

Nachteil: Langsam und umständlich zu benutzen.

```
import java.math.BigInteger;

...

BigInteger b = new BigInteger("100000000");
b = b.multiply(b) ; b = b.multiply(b) ; b = b.multiply(b);
b = b.subtract(new BigInteger("1"));
System.out.println(b);
```

Druckt: 9999999999999999999999999999999999999999999999999999999999999999

Typkonversion

```
int euros = 2;
double gesamt = euros; // ok

double euros = 2.0;
int anzahlEuros = euros; // geht nicht
```

Im ersten Beispiel wird der Integer-Wert *automatisch* in Double konvertiert.
Im zweiten Beispiel geht das nicht.

Typkonversion

Man kann aber schreiben:

```
double euros = 2.50;
int anzahlEuros = (int)euros;
System.out.println(anzahlEuros);
```

Das ist eine explizite Typkonversion (**typecast**).
Hier werden einfach alle Dezimalstellen abgeschnitten.
Will man **runden**, so verwende man

```
double a = 3.759;
System.out.println((int)Math.round(a));
```

Die (statische) Methode `Math.round` berechnet den nächstgelegenen **ganzzahligen** Double-Wert.

Rundungsfehler

```
double f = 4.35;
int n = (int)(100 * f);
System.out.print(n);
```

Druckt 434.
Grund: In Binärdarstellung ist 4,35 ein *echt periodischer* Bruch.

```
(int)Math.round(100 * f);
```

hat Wert 435.

Konstanten

```
int flaschen = 3;
int dosen = 5;
double mengeFanta = flaschen * 0.5 + dosen * 0.33;
```

ist unschön, da `0.5` und `0.33` einfach so dastehen.
Besser:

```
final double FLASCHEN_INHALT = 0.5;
final double DOSEN_INHALT = 0.33;
double mengeFanta = flaschen*FLASCHEN_INHALT+dosen*DOSEN_INHALT;
```

Werte, die mit `final` deklariert werden, können nur einmal initialisiert und danach nicht mehr verändert werden.
Vorteil gegenüber Variablen: Effizienz + Dokumentation.
Numerische Konstanten wie `0.5` *mitten im Programm* sind *schlechter Stil*.
Vordefinierte Double-Konstanten: `Math.PI` und `Math.E`.

Arithmetik

Plus + und Mal * hatten wir schon.
Division wird als / notiert.
Vorsicht: Sind beide Operanden von / Integers, so wird **abgerundet**.

```
int s1 = 5;
int s2 = 6;
int s3 = 3;
double mittelwert = (s1 + s2 + s3) / 3;
```

`mittelwert` hat den Wert `4` (statt `4.666666...`)
Richtig:

```
double mittelwert = (s1 + s2 + s3) / 3.0;
```

Grund: In Java (und C, C++) ist "/" bei `int` die **ganzzahlige Division** ("div").

WAS GIBT'S SONST NOCH

- Die „Punkt vor Strich“ Regel gilt.
- Mathematische Funktionen sind in der Klasse `Math` definiert.
- Automatische Typkonversionen erfolgen von innen nach außen.

Beispiel: Die „Lösungsformel“:

```
double a;
double b;
double c;
/* Wertzuweisung */
x1 = (-b + Math.sqrt(b * b - 4 * a * c)) / (2 * a);
x2 = (-b - Math.sqrt(b * b - 4 * a * c)) / (2 * a);
```

Ebenso: `a * a + b * b - 2 * a * b * Math.sin(phi);`

Zeichenketten

Der Datentyp `String` besteht aus **Zeichenketten**, d.h. Folgen von Buchstaben und Sonderzeichen, eingeschlossen durch `"`

```
String name = "Matthias";
       name = "Johanna";
System.out.println(name);
```

Gibt aus: `Johanna`.

```
int n = name.length();
```

Die Variable `n` hat den Wert 7.

Teile einer Zeichenkette

Ausdruck `s.substring(anfang, endePlusEins)` bezeichnet die Teilzeichenkette von `s` angefangen vom Zeichen an Position `anfang` bis (ausschließlich) zum Zeichen an Position `endePlusEins`.

- Positionen beginnen immer bei Null.
- Länge der Teilzeichenkette = `endePlusEins - anfang`

```
String s = "Hello, World!";
String sub1 = s.substring(0,5);
String sub2 = s.substring(4,8);
```

Was sind die Werte von `sub1` und `sub2`?
Welcher `substring`-Ausdruck hat den Wert `World` ?

Teile einer Zeichenkette

Ausdruck `s.substring(anfang, endePlusEins)` bezeichnet die Teilzeichenkette von `s` angefangen vom Zeichen an Position `anfang` bis (ausschließlich) zum Zeichen an Position `endePlusEins`.

- Positionen beginnen immer bei Null.
- Länge der Teilzeichenkette = `endePlusEins - anfang`

```
String s = "Hello, World!";
String sub1 = s.substring(0,5);
String sub2 = s.substring(4,8);
```

Was sind die Werte von `sub1` und `sub2`?

Welcher `substring`-Ausdruck hat den Wert `World` ?

Antworten:

`sub1` den Wert `"Hello"`

`sub2` den Wert `"o, W"`

Der Ausdruck `s.substring(7,12)` hat den Wert `"World"`

Teile einer Zeichenkette

Will man alle Zeichen von `anfang` bis zum Ende der Zeichenkette, dann kann man

```
s.substring(anfang, s.length())
```

schreiben. Das letzte Zeichen hat nämlich die Position `s.length()` − 1.

Eine erlaubt Kurzform dafür ist auch `s.substring(anfang)`:

```
String s = "01234567";
System.out.println(s.substring(3));
// gibt "34567" aus
```

Fehlerbehandlung

Ruft man `s.substring` mit unpassenden Argumenten auf, so gibt es einen Fehler. Z.B.: `s.substring(4,30)` führt zu:

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:
String index out of range: 20
        at java.lang.String.substring(String.java:1473)
        at Namen.main(Namen.java:5)
```

Man sagt:
"der Ausdruck wirft eine Ausnahme" (**throws an exception**)

Es ist möglich, so eine Ausnahme im Programm "aufzufangen" und benutzerdefinierte Befehle auszuführen, z.B. eine ordentliche Fehlermeldung.

Noch besser ist es, das Auftreten solcher Ausnahmen von vornherein zu vermeiden. später mehr dazu

Verkettung

Zeichenketten kann man konkatenieren.
In Java verwendet man dafür das Pluszeichen.
Der Ausdruck `"Matthias" + "Johanna"` hat den Wert `"MatthiasJohanna"`.

Der Ausdruck

```
"Euro" + "s".substring(0,n)
```

hat den Wert `"Euro"` oder `"Euros"`, je nachdem, ob `n` gleich 0 oder 1 ist. Alle anderen Werte von `n` sind aber nicht erlaubt!

Was "sind" Zeichenketten?

- Eine Zeichenkette ist ein **Objekt**. ⇒ im Heap (seit JDK7)
- Es versteht u.a. die Methoden `length` und `substring`.
  Die Methode `length` liefert die Länge zurück.
  Die Methode `substring` ein *neues* String-Objekt, das den jeweiligen Teilstring enthält.
- Man kann eine Zeichenkette *nie verändern*!
  im Gegensatz zu veränderlichen Objekten wie `Rectangle`.
  Im Computer ist ein String eine Speicheradresse.
  Unter dieser Adresse befindet sich die Länge, z.B. n. In den n darauffolgenden Speicherstellen befinden sich die Zeichen.
  In Java gibt es keine Möglichkeit, diese Zeichen oder die Länge zu verändern, obwohl die Maschinensprache das im Prinzip zuließe. `substring` liefert neues Objekt der Klasse `String`.

Verkettung mit Zahlwerten

```
double betrag = 34.99;
int nummerMahnung = 2;
String anweisung = nummerMahnung + ". Mahnung: " +
        "Bitte zahlen Sie " + betrag + " EUR.";

System.out.println(anweisung);
```

Gibt aus:

```
2. Mahnung: Bitte zahlen Sie 34.99 EUR.
```

Verkettung mit Zahlwerten

Ist ein Operand von + eine Zeichenkette, so wird der andere automatisch in eine Zeichenkette umgewandelt. Das ist *keine* Typkonversion:

```
String betrag = 34.99 * 2;
```

löst aus:

```
incompatible types
found    : double
required: java.lang.String
        String betrag = 34.99 * 2;
```

Vielmehr hat das +-Zeichen je nach Typ der Operanden leicht unterschiedliche Bedeutung. Man spricht von **overloading** (Überladung).

später mehr dazu

Verkettung mit Zahlwerten

Man kann schreiben `""+x` um `x` in eine Zeichenkette umzuwandeln.
Vorsicht:

```
String anweisung = nummerMahnung + ". Mahnung: " +
    "Bitte zahlen Sie " + betrag/3 + " EUR.";
System.out.println(anweisung);
```

Ergebnis:

```
2. Mahnung: Bitte zahlen Sie 11.663333333333334 EUR.
```

Abhilfe: Formatierte Ausgabe

```
import java.text.NumberFormat;
⋮
NumberFormat formatierer = NumberFormat.getNumberInstance();

formatierer.setMaximumFractionDigits(2);
formatierer.setMinimumFractionDigits(2);

double betrag = 34.99;
int nummerMahnung = 2;
String anweisung = nummerMahnung + ". Mahnung: " +
   "Bitte zahlen Sie " + formatierer.format(betrag/3) + " EUR.";
```

Ergebnis:

```
2. Mahnung: Bitte zahlen Sie 11,66 EUR.
```

Sogar Tausenderpunkte werden eingesetzt, z.B.: `1.192.279,33 EUR`.

Beispiel: Benutzernamen

Wir möchten aus dem ersten und letzten Buchstaben des Namens und einer laufenden Nummer einen Benutzernamen erzeugen:

```
String name = "Johanna";
int lfdNo = 1728;
```

Der Benutzername sollte `Ja1728` sein.

Beispiel: Benutzernamen

Wir möchten aus dem ersten und letzten Buchstaben des Namens und einer laufenden Nummer einen Benutzernamen erzeugen:

```
String name = "Johanna";
int lfdNo = 1728;
```

Der Benutzername sollte `Ja1728` sein.

Kein Problem:

```
String benutzerName;
benutzerName = name.substring(0,1) +
     name.substring(name.length() - 1) + lfdNo;
```

Parsing von Zeichenketten

Wie erhalten wir aus einem Benutzernamen die laufende Nummer?

```
benutzerName.substring(2)
```

enthält zwar die Ziffern der lfd. Nr. ist aber immer noch ein String.

Lösung:

```
int nummer = Integer.parseInt(benutzerName.substring(2));
```

`parseInt` ist eine **statische Methode** der Klasse `Integer` und dient dazu, eine Zeichenkette in einen integer umzuwandeln.

$\Rightarrow$ Statische Methoden werden nicht an ein Objekt geschickt, sondern können "einfach so" mit dem Klassennamen ausgeführt werden. später mehr dazu

Parsing von Doubles

... geht analog mit `Double.parseDouble`, z.B.:

```
double c = Double.parseDouble("2.97E9"); /* oder so */
```

Vorsicht: eine “deutsche” Zahl, wie 1.234,59 kann `parseDouble` nicht verarbeiten.
Wie das geht, siehe Java Reference Manual

Fallunterscheidungen

Oft will man ein Statement nur dann ausführen, wenn eine bestimmte Bedingung gilt. Das geht mit dem `if`-Statement:

```
if (zinsSatz > 100.0) {
     System.out.println("Fehler.");
} else
     rate = restschuld * zinsSatz/100.0 / 12.0 + tilgung;
```

Bemerkung:

Ausgaben an `System.out` sind dilettantisch.
Hier sollte eine Ausnahme geworfen werden oder eine Fehlerbehandlung stattfinden.

Professionellere Lösung

```
if (zinsSatz > 100.0) {
     Fehlerbearbeitung.fehler(falscherZinsSatz);
} else
     rate = restschuld * zinsSatz/100.0 / 12.0 + tilgung;
```

- `Fehlerbearbeitung.fehler` ist eine **benutzerdefinierte** Methode, die **Fehlerobjekte** anzeigt und entsprechend verfährt.
- Ein (benutzerdefiniertes) **Fehlerobjekt** (hier `falscherZinsSatz`) beinhaltet den Anzeigetext (üblicherweise in verschiedenen Sprachen) und Instruktionen wie bei seinem Auftreten zu verfahren ist.

Bedingungen

…stehen immer in Klammern und sind von der Form x_1 *op* x_2 wobei

- *op* ist `<`, `>`, `<=`, `>=`, `==`
- Die Operanden sind Zahlen des gleichen Typs, evtl. wird implizite Typkonversion vorgenommen.

Später lernen wir noch andere Bedingungen kennen.

Vorsicht:

Testen von Doubles auf Gleichheit ist problematisch wegen möglicher Rundungsfehlern. Besser die Größe der Differenz testen:

```
double final EPSILON = 1E-10; // zum Beispiel
if ( Math.abs(x - y) <= EPSILON )
```

Blöcke & Zusammengesetzte Statements

Blöcke

Formal steht nach der Bedingung des `if` genau *ein* Statement.

Braucht man mehrere, so kann man eine Folge von Statements mit geschweiften Klammern `{ }` zu einem **Block** zusammenfassen.

Ein Block wird praktisch wie ein einzelnes Statement betrachtet.

zusammengesetzte Statements

Das ganze `if` - `else` Konstrukt wird als ein einziges Statement aufgefasst, es ist ein **zusammengesetztes Statement**.

Man darf den `else` Teil auch komplett weglassen.

Formale Syntax für Statements

$$\begin{array}{lcl}\langle \textit{statement} \rangle & ::= & \langle \textit{type_expression} \rangle\ \langle \textit{ident} \rangle\ [\texttt{=}\ \langle \textit{expression} \rangle]\texttt{;} \\ & | & \texttt{\{}\ (\langle \textit{statement} \rangle)^{+}\ \texttt{\}} \\ & | & \texttt{if (}\ \langle \textit{expression} \rangle \texttt{)}\ \langle \textit{statement} \rangle\ [\texttt{else}\ \langle \textit{statement} \rangle] \\ & | & \dots \end{array}$$

...

Durch diese sog. **Backus Naur Form** (kurz **BNF**) wird die Menge der Statements formal definiert.

- Jedes in ⟨...⟩ geschriebene Wort bezeichnete eine Menge von Ausdrücken/Sprachelementen.
- `Courier`-Text bezeichnet wörtliche Anteile
- | trennt mehrere Alternativen voneinanders
- [...] bedeutet “optional”
- $(\dots)^{+}$ bedeutet “mindestens ein oder mehrere”
- $(\dots)^{*}$ bedeutet “keins oder mehrere”

Was ist hier falsch?

```
                                                  // Fehler!
if (betrag <= kontostand)
     double neuerKontostand = kontostand - betrag;
     kontostand = neuerKontostand;
```

Was ist hier falsch?

```
                                            // Fehler!
if (betrag <= kontostand)
     double neuerKontostand = kontostand - betrag;
     kontostand = neuerKontostand;
```

Antwort: Die geschweiften Klammern fehlen!
Ohne geschweifte Klammern wird die dritte Zeile immer ausgeführt.

Glücklicherweise wird der Fehler hier vom Kompiler erkannt, da die Variable `neuerKontostand` ja nicht immer deklariert wäre.
Im Allgemeinen ist dies aber eine mögliche Fehlerquelle
$\Rightarrow$ Besser immer explizit Klammern!

Was ist hier falsch?

```
double neuerKontostand = 0;                // Noch schlimmer!
if (betrag <= kontostand)
     neuerKontostand = kontostand - betrag;
     kontostand = neuerKontostand;
```

Antwort: Die geschweiften Klammern fehlen!
Ohne geschweifte Klammern wird die dritte Zeile immer ausgeführt.

Glücklicherweise wird der Fehler hier vom Kompiler erkannt, da die Variable `neuerKontostand` ja nicht immer deklariert wäre.
Im Allgemeinen ist dies aber eine mögliche Fehlerquelle
$\Rightarrow$ Besser immer explizit Klammern!

Antwort

Merke: Einrückungen dienen der Lesbarkeit,
werden aber vom Java Compiler ignoriert.

Richtig:

```
if (betrag <= kontostand) {
    double neuerKontostand = kontostand - betrag;
    kontostand = neuerKontostand;
}
```

Oder auch so:

```
if (betrag <= kontostand)
{
    double neuerKontostand = kontostand - betrag;
    kontostand = neuerKontostand;
}
```

Mehrere if-Statements

Natürlich können in den Zweigen eines if-Statements wiederum solche stehen:

```
if ( richter >= 8.0 )
   s = "Grosse Verwuestung";
else if ( richter >= 7.0 )
   s = "Viele Gebaeude zerstoert";
else if ( richter >= 6.0 )
   s = "Viele Gebaeude beschaedigt";
else if ( richter >= 4.5 )
```

...

Dangling else

Typsicher Fehler:

```
if (betrag > 0)
  if (kontostand > betrag)
    kontostand = kontostand - betrag; // Abbuchung
else
  kontostand = kontostand - betrag;   // Gutschrift
```

Ein `else` bezieht sich immer auf das nächstgelegene `if`.
Ohne Klammern ist dies hier das aber innere! ↯
Erneut gilt: Einrückungen werden vom Java Compiler ignoriert!

Dangling else

Typsicher Fehler:

```
if (betrag > 0)
  if (kontostand > betrag)
    kontostand = kontostand - betrag; // Abbuchung
else
  kontostand = kontostand - betrag;   // Gutschrift
```

Ein `else` bezieht sich immer auf das nächstgelegene `if`.
Ohne Klammern ist dies hier das aber innere! ↯
Erneut gilt: Einrückungen werden vom Java Compiler ignoriert!

```
if (betrag > 0) {
  if (kontostand > betrag)
    kontostand = kontostand - betrag; // Abbuchung
} else {
  kontostand = kontostand - betrag;   // Gutschrift
}
```

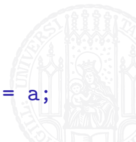
Beispiel: Maximum und Minimum

Wir wollen aus `int a, b, c` das größte und das kleinste berechnen.

Die können wir mit verschachtelten `if`-statements effizient erreichen.

Allerdings sollte man solche größere Code-Blöcke nicht direkt einbauen.

```
if (a >= b)
    if (a >= c) {
        max = a;
        if (c >= b)
            min = b;
        else
            min = c;
    } else {
        max = c; min = b;
    }
else
    if (b >= c) {
        max = b;
        if (c >= a)
            min = a;
        else
            min = c;
    } else {
        max = c; min = a;
    }
```



Beispiel: Maximum und Minimum

Wir wollen aus `int a, b, c` das größte und das kleinste berechnen.

Die können wir mit verschachtelten `if`-statements effizient erreichen.

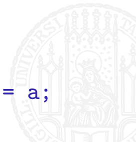
Allerdings sollte man solche größere Code-Blöcke nicht direkt einbauen.

Man kann (und sollte) auch einfach Bibliotheksfunktionen verwenden:

```
max = Math.max(a, Math.max(b,c));
min = Math.min(a, Math.min(b,c));
```

Deutlich lesbarer;
Korrektheit offensichtlich!

```
if (a >= b)
    if (a >= c) {
        max = a;
        if (c >= b)
            min = b;
        else
            min = c;
    } else {
        max = c; min = b;
    }
else
    if (b >= c) {
        max = b;
        if (c >= a)
            min = a;
        else
            min = c;
    } else {
        max = c; min = a;
    }
```



Der Datentyp boolean

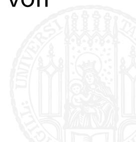
```
int x = 5;
System.out.println(x < 10);
```

Gibt aus: `true`

In Java ist `x < 10` ein **Ausdruck des Typs** `boolean`, genauso wie `x + 12` einer vom Typ `int` ist.

Werte des Typs `boolean` sind (nur) `true` und `false`.

Werte des Typs `boolean` liegen auf dem Stack oder innerhalb von Objekten eines anderen Typs, aber nicht alleine im Heap
Ausnahme: `Boolean`



Logische Verknüpfung

Auf dem Typ `boolean` sind die **zweistelligen** Verknüpfungen `&&` und `||` erklärt:

- e_1 `&&` e_2 bedeutet: "e_1 und e_2";
  bzw. "sowohl e_1, als auch e_2"; "e_1 und e_2 beide `true`".
- e_1 `||` e_2 bedeutet: "e_1 oder e_2";
  bzw. "mindestens eins von beiden, e_1 oder e_2, ist `true`".

Außerdem gibt es die **einstellige** Operation `!`:

- `!`e bedeutet: "nicht e", "das Gegenteil von e".

Beachte: bei e_1`||`e_2 wird zunächst e_1 ausgewertet. Ist das Ergebnis `true`, so wird e_2 gar nicht erst ausgewertet. Analog für `&&`.

Zu Berücksichtigen, wenn Seiteneffekte auftreten können.

Beispiel 1

Johannas Geburtstag ist der 21.12.
Angenommen, wir haben `int`-Variablen `day` und `month`.
Welcher Boole'sche Ausdruck ist `true` genau dann, wenn die Werte der beiden Variablen Johannas Geburtstag entsprechen?

Beispiel 1

Johannas Geburtstag ist der 21.12.
Angenommen, wir haben `int`-Variablen `day` und `month`.
Welcher Boole'sche Ausdruck ist `true` genau dann, wenn die Werte der beiden Variablen Johannas Geburtstag entsprechen?

Antwort

Der Ausdruck `day == 21 && month == 12`

So können wir ihn verwenden:

```
if (day == 21 && month == 12)
     System.out.println("Happy birthday, Johanna!");
```

Beispiel 2

Wie drücken wir aus, dass die `double` Variable `heat` zwischen `100.0` und `120.0` liegt?

Beispiel 2

Wie drücken wir aus, dass die `double` Variable `heat` zwischen `100.0` und `120.0` liegt?

Antwort

```
100.0 <= heat && heat <= 120.0
```

Äquivalent sind auch

```
!(heat < 100.0) && !(heat > 120.0)
!(heat < 100.0 || heat > 120.0)
```

De Morgan's Gesetz:

```
!(a && b)  =  !a || !b
!(a || b)  =  !a && !b
```

Boole'sche Variablen

Man kann auch Variablen des Typs `boolean` deklarieren:

```
boolean mitBedienung;
boolean tischGedeckt;
boolean draussen;

/* Initialisierung von draussen und tischGedeckt */

mitBedienung = !draussen || tischGedeckt;

if (mitBedienung)
   System.out.println("Hier keine Selbstbedienung!");
```

Andere Boole'sche Ausdrücke

Methoden können einen `boolean` Wert zurückliefern:
Die Klasse `String` enthält die Methode `equals`, die einen `String` Parameter hat und einen `boolean` zurückliefert.

```
String answer;

System.out.println("Koennen Sie mir helfen?");
/* Eingabe von answer */
if (answer.equals("ja") || answer.equals("Ja")) {
    System.out.println("Danke!");
} else {
    System.out.println("Schade.");
}
```

Stringvergleiche

Es gibt auch die Methode `equalsIgnoreCase`, die Groß/Kleinschreibung ignoriert.

```
if (antwort.equalsIgnoreCase("ja"))
   System.out.println("Danke!");
```

Zum Vergleich von Strings soll man *nicht* `==` verwenden.

```
String x = "a";
System.out.println("ja" == "ja");
System.out.println("ja" == "j" + "a");
System.out.println("ja" == "j" + x);
```

Gibt aus: `true true false`

Grund: `==` bezeichnet hier Identität, d.h. liegen die Strings an derselben Stelle im Speicher. ⇒ *gilt für alle Objekte!*

INTERNALISIERUNG

```
String x = "a";
System.out.println("ja" == "ja");
System.out.println("ja" == "j" + "a");
System.out.println("ja" == ("j" + x));
System.out.println("ja" == ("j" + x).intern());
```

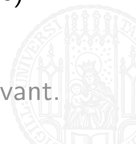
Gibt aus: `true true false true`

Grund: Methode `intern` schaut nach, ob ein identischer String schon vorhanden ist und gibt ggf. diesen zurück. Ansonsten werden String-Objekte verglichen.

Besser:

Strings vergleichen mit `equals` oder `compareTo` (nächste Folie).

Der Inhalt dieser Folie ist nicht prüfungsrelevant.



Stringvergleiche

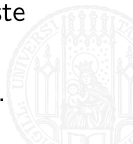
Die Methode `compareTo` vergleicht nach der lexikographischen Ordnung, liefert aber einen `int` zurück:

s_1`.compareTo(`s_2`)` ist

- < 0, wenn s_1 alphabetisch vor s_2 kommt
- $= 0$, wenn s_1 und s_2 gleich sind
- > 0, wenn s_1 alphabetisch nach s_2 kommt.

Beispiele:

- `"Huber".compareTo("Schmidt")` ist < 0
- `"huber".compareTo("Schmidt")` ist > 0 (Kleinbuchstaben kommen erst nach den Großbuchstaben)
- `"Vorlesung".compareTo("Vorlesen")` ist > 0 (Die erste Nicht-Übereinstimmung entscheidet)
- `"AAA".compareTo("A")` ist > 0 (Kurz kommt vor Lang).



Übungen

Was ist an den folgenden Statements falsch ?

```
if cents > 0 then System.out.println(cents + " Cents");

if (1 + x > Math.pow(x, Math.sqrt(2)) y = y +x;

if (x = 1) y++; else if (x = 2) y = y + 2;

if (x && y == 0) p = new Point(x,y);

if (1 <= x <= 10) {System.out.println("Danke.");}

if (!antwort.equalsIgnoreCase("Ja ") ||
            !antwort.equalsIgnoreCase("Nein"))
    System.out.println("Antworten Sie mit Ja oder Nein.");
```

Beispiel: Schaltjahre

Jedes vierte Jahr ist ein Schaltjahr, es sei denn, die Jahreszahl ist durch hundert teilbar. In diesem Fall liegt ein Schaltjahr nur vor, wenn die Jahreszahl durch 400 teilbar ist.

Beispiel: 2000 war ein Schaltjahr, 1900 war keins.

Man schreibe einen Boole'schen Ausdruck, der `true` ist, genau dann wenn `jahr` ein Schaltjahr ist.

Hilfe: `x % y` ist der Rest der ganzzahligen Division von `x` durch `y`.

Z.B.: `12%5`=2.

Beispiel: Schaltjahre

Jedes vierte Jahr ist ein Schaltjahr, es sei denn, die Jahreszahl ist durch hundert teilbar. In diesem Fall liegt ein Schaltjahr nur vor, wenn die Jahreszahl durch 400 teilbar ist.
Beispiel: 2000 war ein Schaltjahr, 1900 war keins.
Man schreibe einen Boole'schen Ausdruck, der `true` ist, genau dann wenn `jahr` ein Schaltjahr ist.
Hilfe: `x % y` ist der Rest der ganzzahligen Division von `x` durch `y`.
Z.B.: `12%5`=2.

Antwort

```
jahr % 4 == 0 && !(jahr % 100) == 0 || jahr % 400 == 0
```

Merke:

- `&&` bindet stärker als `||`
- `!` bindet stärker als `&&`

Beispiel: Schaltjahre

Jedes vierte Jahr ist ein Schaltjahr, es sei denn, die Jahreszahl ist durch hundert teilbar. In diesem Fall liegt ein Schaltjahr nur vor, wenn die Jahreszahl durch 400 teilbar ist.
Beispiel: 2000 war ein Schaltjahr, 1900 war keins.
Man schreibe einen Boole'schen Ausdruck, der `true` ist, genau dann wenn `jahr` ein Schaltjahr ist.
Hilfe: `x % y` ist der Rest der ganzzahligen Division von `x` durch `y`.
Z.B.: `12%5`=2.

Antwort

```
((jahr % 4 == 0) && !((jahr % 100) == 0)) || jahr % 400 == 0
```

Merke:

- `&&` bindet stärker als `||`
- `!` bindet stärker als `&&`

Zusammenfassung Kapitel 2

- Datentypen `int, double`.
  - Elemente und Grundoperationen
  - Implizite Konversion
  - Anwendungsbeispiele
- Fallunterscheidungen
  - Anwendungsbeispiele
  - Formale Syntax mit Backus-Naur-Form
    (wird hierdurch auch exemplarisch eingeführt)
  - Geschachtelte Fallunterscheidungen und deren Anwendung
- Der Datentyp `boolean`: Verwendung und Grundoperationen
- Der Datentyp `String`: Verwendung und Grundoperationen
- Aufteilung des Speichers in Stack (für Programmvariablen) und Heap (Objekte)