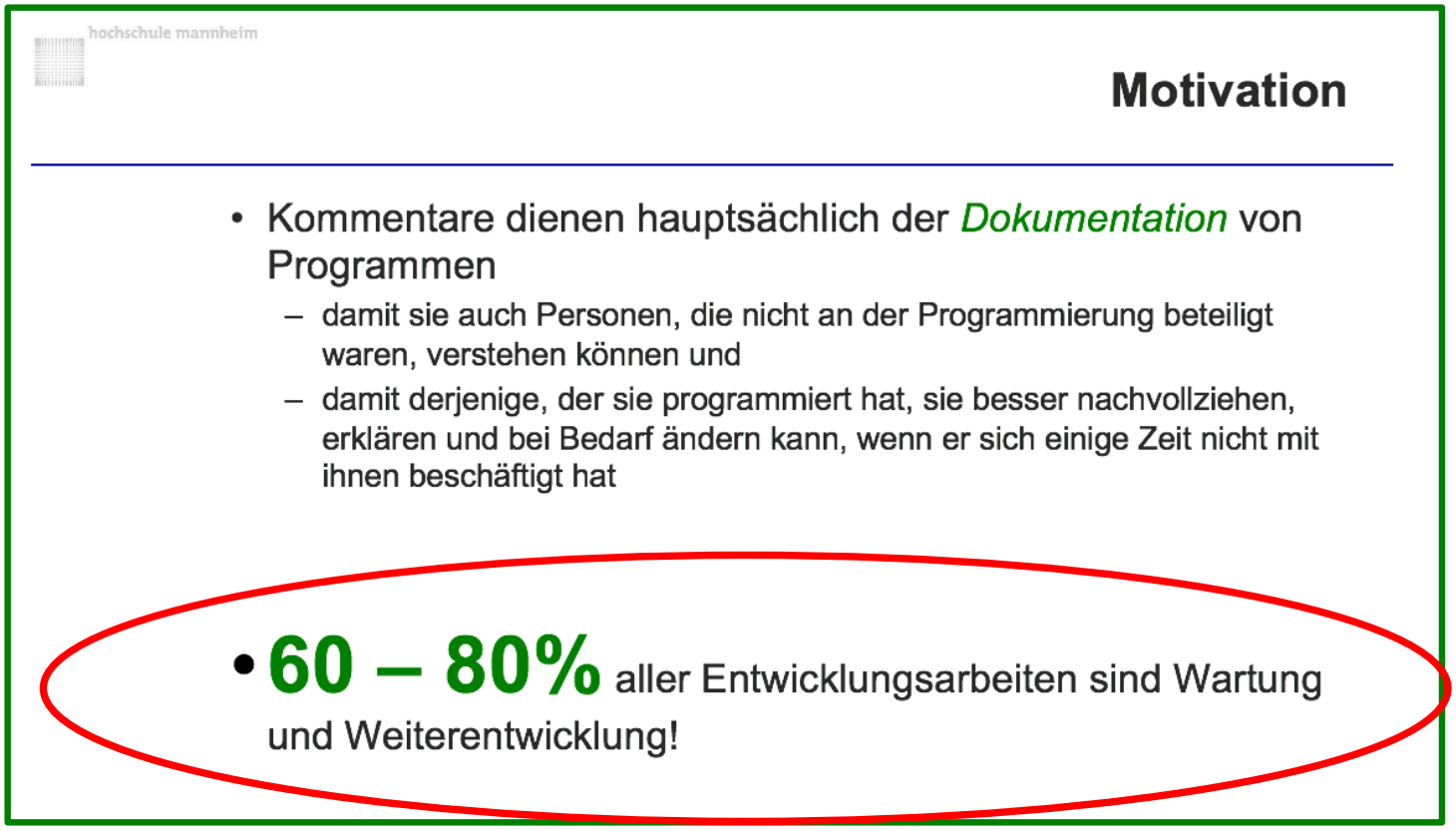
Lose Enden

- Namens- und andere Konventionen
- Ganze Zahlen
- Gleitkommazahlen und Casts
- Weitere Operatoren für Zuweisungen
- Boole‘sche Ausdrücke
- Operator-Vorrang
- Einzelne Zeichen (char)
- Strings und String-Methoden
- Konstanten
- Variablen (Formalisierung)

Konventionen: Motivation



-> Programme werden viel öfter gelesen als geschrieben

-> Man sollte etwas tun, um das zu erleichtern

Was sind Namenskonventionen und warum sollte man sich daran halten?

Namenskonventionen sind Nomenklaturen, also (*freiwillige*) *Vereinbarungen* von Programmierern, Datenbankentwicklern etc., die ein bestimmtes System für die Vergabe von Bezeichnern für Objekte, Variablen und Konstanten einführen, damit anhand der Namen sofort *Rückschlüsse auf den Verwendungszweck* im Programm / in der Datenbank gezogen werden können.

[Wikipedia “Namenskonvention (Datenverarbeitung)”]

Vereinbarungen in Java

Upper Camel Case mit Binnenmajuskeln für...

- Klassennamen; Beispiele:
  String, LinearSearch, ExcelReader

Lower Camel Case mit Binnenmajuskeln für...

- Methodennamen; Beispiele:
  searchName, readInt, saveFile, getLineNumber
- Variablennamen; Beispiele:
  numberOfRecords, firstName, lastName
- Paketnamen; Beispiele:
  pr1.uebung03, java.awt.event, org.apache.poi.hssf.usermodel

Namen von Konstanten werden vollständig in Großbuchstaben geschrieben und durch Unterstriche abgesetzt; Beispiele:

PDF_FILE_NAME, MAX_ROW_NUMBER

```
public class DividersArray {

    public static void main(String[] args) {
        print("Geben Sie eine ganze Zahl ein: ");
        int input = readInt();

        if (input > 0) {
            int[] dividers = new int[35];
            int divider = 1;
            int index = 0;
            while (divider <= input) {
                if (input % divider == 0) {
                    dividers[index] = divider;
                    index = index + 1;
                }
                divider = divider + 1;
            }

            index = 0;
            while (index < 35) {
                print(dividers[index] + " ");
                index = index + 1;
            }
            println();
        } else {
            println("ungültige Eingabe");
        }
    }

}
```

Konventionen: Einrücken

Der Rumpf von

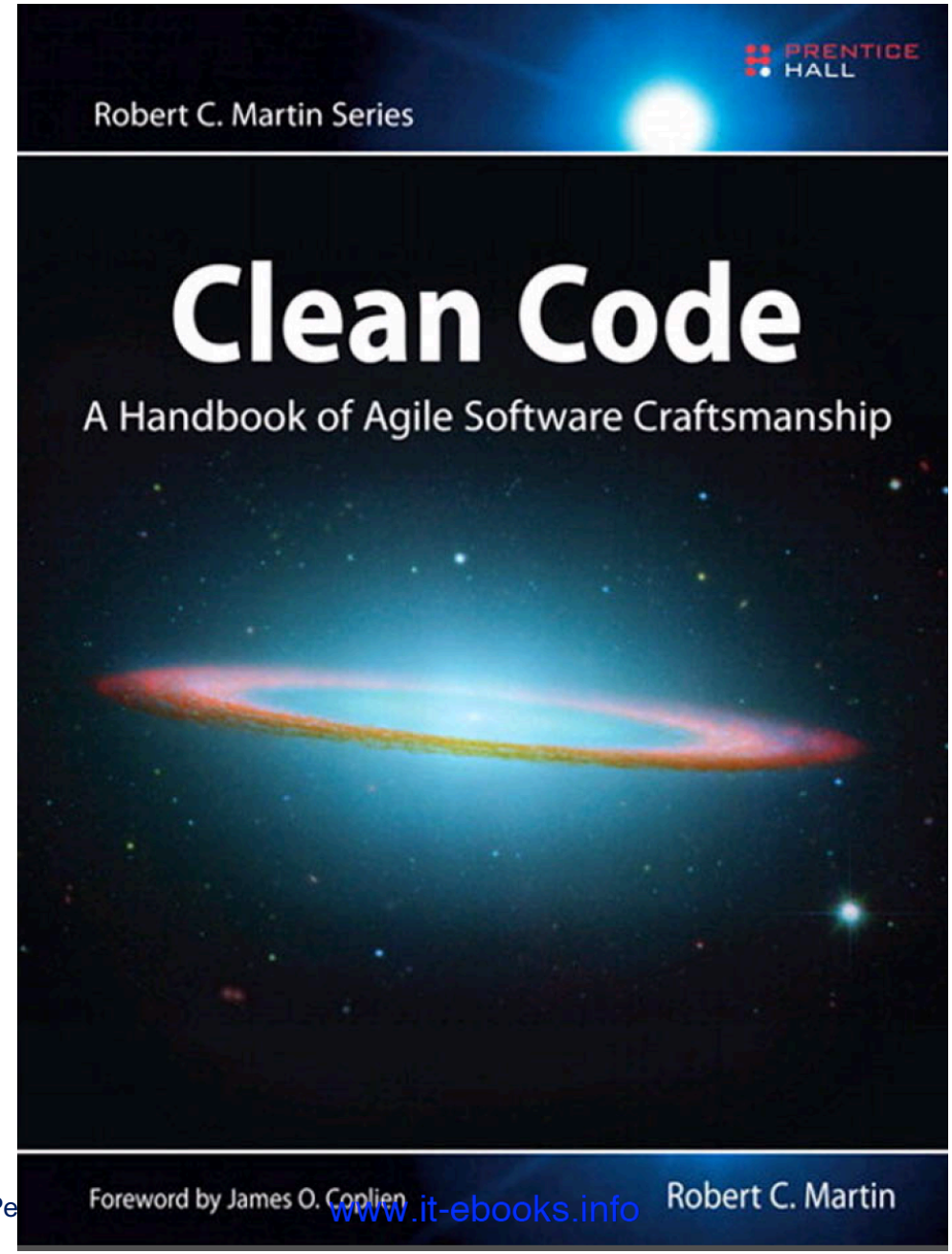
- Klassen,
- Methoden,
- Bedingungen und
- Schleifen

wird um jeweils einen Tabulator eingerückt

Tipp

- Eclipse unterstützt diese Konventionen!
- Tastaturkürzel (shortcuts)
  1. Cursor in das Fenster mit dem Programm setzen
  2. Windows: <strg>-a <strg>-i
     MacOs: <cmd>-a <cmd>-i
     oder
     Windows: <strg>-a <strg><shift>-f
     MacOs: <cmd>-a <cmd><shift>-f
- Erläuterung:
  - „a“ erinnert an „all“ == alles markieren
  - „i“ erinnert an „indentation“ == korrekt einrücken
  - „f“ erinnert an „format“ == alles geeignet formatieren

Empfehlenswert



Das Buch hat 462 Seiten...

Ganze Zahlen in Java

- Wir benutzen bis jetzt nur ganze Zahlen vom Typ *int*
- Alle Zahl-*Literale*, die wir in Java-Programmen schreiben, werden intern als *int* dargestellt
- Es gibt aber noch andere ganzzahlige Typen…

Ganzzahlige Datentypen

- byte:
  - von -128 bis 127
  - Das entspricht von -2^7 bis 2^7-1
  - Das entspricht den Werten Byte.MIN_VALUE bis Byte.MAX_VALUE
- short:
  - von –32.768 bis 32.767
  - Das entspricht von -2^{15} bis $2^{15}-1$
  - Das entspricht den Werten Short.MIN_VALUE bis Short.MAX_VALUE
- int:
  - von –2.147.483.648 bis 2.147.483.647
  - Das entspricht von -2^{31} bis $2^{31}-1$
  - Das entspricht den Werten Integer.MIN_VALUE bis Integer.MAX_VALUE
- long:
  - von –9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807, also von ca. $-9*10^{18}$ bis $9*10^{18}$
  - Das entspricht von -2^{63} bis $2^{63}-1$
  - Das entspricht den Werten Long.MIN_VALUE bis Long.MAX_VALUE

Die angegebenen Grenzen sind im jeweiligen Wertebereich enthalten

long-Zahlen

- Ganze Zahlen, die intern als *long* dargestellt werden *sollen*, werden durch ein nachfolgendes großes "L" oder kleines "l" gekennzeichnet
  (Achtung: kein Leerzeichen dazwischen!)
- Beispiele
  - 12L
  - 123456789L
  - 1234l Achtung: das "l" kann man mit "1" oder mit "|" verwechseln, → "L" ist zu bevorzugen
- Ganze Zahlen, die außerhalb der *int*-Grenzen liegen, *müssen* mit "L" (oder "l") gekennzeichnet werden

Rechnen mit ganzzahligen Typen

- Im Fall von ungleichen Typen der beteiligten Operanden
  1. Zunächst wird die Zahl des kleineren Typs in die äquivalente Zahl des größeren Typs konvertiert
  2. Dann erst wird gerechnet
  3. Das Ergebnis ist eine Zahl des größeren Typs

- Im Fall, dass das Ergebnis einer Operation nicht in die Zahl des größeren Typs passt (Überlauf)
  1. Die Operation wird durchgeführt
  2. Die oberen (höchstwertigen) Bits des Ergebnisses werden abgeschnitten
  3. Nur diejenige Anzahl von Bits, die entsprechend des Ausdruckstyps dargestellt werden können, wird als Ergebnis geliefert

→ Achtung:
  – Ein eventueller *Integer-Überlauf* muss *vom Programmierer* vermieden werden (die Sprache/die VM sorgen *nicht* dafür)
  – Verlorengehen kann sonst ein Teil des Ergebniswertes und/oder sein Vorzeichen

Integer-Überlauf 1/2

- Es kann sein, dass das Ergebnis einer Operation *nicht* in einen „int“-Wert (= 32 Bit) *passt*

  In diesem Fall gilt:

  – Die Operation wird durchgeführt
  – Die oberen (höchstwertigen) Bits des Ergebnisses werden *abgeschnitten*
  – Nur diejenige Anzahl von Bits, die entsprechend des Ausdruckstyps dargestellt werden können, wird als Ergebnis geliefert

- Beispiel
  – In Dezimaldarstellung:

```
    2 100 000 000
+     100 000 000
= -2 094 967 296
```

  – In Binärdarstellung:

```
  01 11110 10010 10110 11101 01000 00000
+ 00 00010 11111 01011 11000 01000 00000
= 10 00001 10010 00010 10101 10000 00000
```

  ^ das erste Bit wird immer als Vorzeichen gewertet!

Zwei Möglichkeiten, mit Java-Bordmitteln ganze Zahlen einzulesen

```
Scanner in = new Scanner(System.in);
int number = in.nextInt();
in.close();
System.out.println(number);
```

```
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
int number = Integer.parseInt(in.readLine());
System.out.println(number);
```

Die Alternative zu diesen etwas „geschwätzigen“ Programmteilen heißt *readInt()*

Gleitkomma-Zahlen in Java

- Java unterscheidet zwei Arten von Gleitkomma-Zahlen, die sich durch ihre *Genauigkeit* und ihren *Wertebereich* unterscheiden
- *float*
  - Positive Zahlen sind im Bereich von $1.40239846*10^{-45}$ bis $3.40282347*10^{+38}$ darstellbar, negative entsprechend
  - Das entspricht den Konstanten Float.MIN_VALUE bis Float.MAX_VALUE
  - Jede *float* wird intern mit 32 Bit dargestellt (Mantisse inkl. VZ: 24 Bit)
- *double*
  - Positive Zahlen sind im Bereich von $4.94065645841246544*10^{-324}$ bis $1.79769313486231570*10^{+308}$ darstellbar, negative entsprechend
  - Das entspricht den Konstanten Double.MIN_VALUE bis Double.MAX_VALUE
  - Jede *double* wird intern mit 64 Bit dargestellt (Mantisse inkl. VZ: 53 Bit)

Abgekürzte Syntax-Regeln für Gleitkomma-Zahlen in EBNF

FloatingPointLiteral ::=

```
    { Digit }+   . { Digit }    [ e | E  [ + | - ]  { Digit }+ ]   [ f | F | d | D ]
  |              . { Digit }+   [ e | E  [ + | - ]  { Digit }+ ]   [ f | F | d | D ]

  |  { Digit }+                 ( e | E  [ + | - ]  { Digit }+ )   [ f | F | d | D ]
  |  { Digit }+                 [ e | E  [ + | - ]  { Digit }+ ]   ( f | F | d | D ).
```

- Gleitkomma-Zahlen (im Unterschied zu ganzen Zahlen) ...
  - enthalten entweder einen Punkt (Dezimalkomma)
  - oder sie besitzen einen Exponenten
  - oder sie sind als *float* oder *double* gekennzeichnet
  - oder sie enthalten eine Kombination daraus
- Beispiele...?

Interne Darstellung von Gleitkomma-Zahlen

- Gleitkomma-Zahlen im Programm werden (sofern nicht anders gekennzeichnet) intern *immer* als Wert vom Typ *double* dargestellt (das ist wie bei ganzen Zahlen und *int*)
- Das gilt auch, wenn ihr Wert in einen *float*-Typ passen würde

- Gleitkomma-Zahlen, die intern als *float* dargestellt werden sollen, *müssen* durch ein nachfolgendes "F" oder "f" gekennzeichnet werden (Achtung: kein Leerzeichen dazwischen!)
- Zur Unterscheidung können Gleitkomma-Zahlen, die intern als *double* dargestellt werden *sollen*, durch ein nachfolgendes "D" oder "d" gekennzeichnet werden (das ist aber unnötig;
  Achtung: kein Leerzeichen dazwischen!)

Rechnen mit Gleitkomma-Zahlen

Im Fall von ungleichen Typen der beteiligten Operanden

1. Zunächst wird die kleinere *float*-Zahl in die äquivalente größere *double*-Zahl konvertiert; das Ergebnis ist eine *double*-Zahl
   Ganze Zahlen (alle Typen) werden in den nächstliegenden Gleitkomma-Wert konvertiert
2. Dann wird exakt (!) gerechnet
3. Ist die Größenordnung des Ergebnisses darstellbar, wird der exakte Werte hin zur nächsten darstellbaren *double*-Zahl gerundet
4. Sonst ist ein Überlauf erfolgt, dann ist der Ergebniswert laut ANSI/IEEE 754 als *positiv oder negativ unendlich* definiert! →

Unendlich klein ... Unendlich groß

- Darstellung von positiv bzw. negativ Unendlich
  - Eine positiv unendliche *float*-Zahl wird durch *Float.POSITIVE_INFINITY* dargestellt
  - Eine negativ unendliche *float*-Zahl wird durch *Float.NEGATIVE_INFINITY* dargestellt
  - Eine positiv unendliche *double*-Zahl wird durch *Double.POSITIVE_INFINITY* dargestellt
  - Eine negativ unendliche *double*-Zahl wird durch *Double.NEGATIVE_INFINITY* dargestellt

- Mit den Unendlich-Werten kann weitergerechnet werden, sie können untereinander und mit anderen Zahlen addiert, verglichen werden etc.

Eine Zahl und doch keine Zahl 1/2

- Es gibt weitere Rechnungen, die zu einer besonderen Situation führen, z.B.:

  Die Division von 0 durch 0, z.B.

  0f / 0f Ergebnistyp: float

  0d / 0d Ergebnistyp: double

- In diesem Fall ist der Ergebniswert als *Not-a-Number* ("keine Zahl") definiert (wie in anderen Situationen, in denen ein Ergebnis mathematisch undefiniert ist)

→ Es tritt ***kein*** *Rechenfehler* auf!

- Java repräsentiert den Wert Not-a-Number als

  Float.NaN oder

  Double.NaN

→ Bei Operationen auf Gleitkomma-Zahlen tritt *nie eine Exception* auf

Eine Zahl und doch keine Zahl 2/2

- Der Wert „NaN“ ist eine Zahl, aber ungleich
  - zu allen anderen Zahlen und
  - zu sich selbst.
- Es existieren Testmethoden

  Float.isNaN(*Ausdruck*) und
  Double.isNaN(*Ausdruck*)

- Beispiele:

  Float.isNaN(0.0f / 0.0f)

  // ⇒

  0.0f / 0.0f == 0.0f / 0.0f

  // ⇒

  Float.NEGATIVE_INFINITY + Float.POSITIVE_INFINITY

  // ⇒

Typ-Konvertierung: Beispiele 1/2

- Werte numerischer Typen können explizit umgewandelt werden
- Sogenannte *Type Casts* können jeden numerischen Typ in jeden anderen numerischen Typ umwandeln
- Schreibweise:
  (*zieltyp*) *ausdruck*

- Beispiele:

```
int i = (int) 3.14 + 2;

i = (int) 3.14 + (int) 1.9f;

i = (int) (3.14 + 1.9f);

byte b = (byte) 140;
```

Typ-Konvertierung: Regeln

- Ist der Zieltyp „größer“ als der Typ des Wertes, dann ist der Cast *unnötig*, Java macht den Cast implizit
- Ist der Zieltyp „kleiner“ als der Typ des Wertes dann ist der Cast unbedingt *notwendig*, sonst meldet der Compiler einen Typfehler
  - Überzählige Bit-Stellen werden *abgeschnitten*
- Es gilt:

  byte < short < int < long < float < double

- Kleine ganzzahlige *Literale* werden *automatisch* angeglichen
- Bei Zuweisung ganze Zahl → Gleitkommazahl gilt
  - Die Größenordnung bleibt erhalten
  - Überzählige Bits der Mantisse werden abgeschnitten
- Bei Wandlungen Gleitkommazahl → ganze Zahl mit *Cast* gilt
  - Nachkommastellen werden abgeschnitten (Runden gegen 0)

Weitere Rechenoperationen: der Zuweisungs*operator*

- Zuweisungen schreiben sich

  variable = *ausdruck*; zum Beispiel
  name = "Schmidt";

- Die Wert-Zuweisung in Java ist *keine Anweisung* im üblichen Sinne, sondern ein Ausdruck (*AssignmentExpression*)

- Der Zuweisungsoperator in Java ist (wie in fast allen Programmiersprachen) *rechts*assoziativ (sinnvollerweise):
  - Der Ausdruck auf der rechten Seite des Operators wird zuerst ausgewertet
  - Das Ergebnis der Auswertung wird an die Variable auf der linken Seite des Operators zugewiesen
  - Bemerkung:
    Die Zuweisung hat einen Wert (das ist der zugewiesene Wert); mit diesem Wert kann weitergearbeitet werden

Abkürzende Schreibweisen 1/3

- Für häufig gebrauchte Operationen gibt es abkürzende Schreibweisen bei der Zuweisung
- Beispiel

  a = a + 10;

  kann geschrieben werden als

  a += 10;

- Das erlaubt die Syntaxregel

  AssignmentOperator: one of

  = *= /= %= += -= <<= >>= >>>= &= ^= |=

  Diese Operatoren behandeln wir nicht in PR1

- Der Wert einer solchen verkürzten Zuweisung entspricht dem Wert des ursprünglichen Ausdrucks

Abkürzende Schreibweisen 2/3

- Für noch häufiger gebrauchte Operationen gibt es weitere abkürzende Schreibweisen bei der Zuweisung
- Beispiel
  a = a + 1; oder auch a += 1;
  kann noch kürzer geschrieben werden als
  a++;
  oder
  ++a;
- Genauso gibt es auch den "--" -Operator

- Diese Operatoren werden *Inkrement* und *Dekrement* genannt
- Da man sie *vor* (lat. prae) und *nach* (lat. post) der Variablen schreiben kann, spricht man von *Präfix-* bzw. *Postfix-Inkrement-* und *Dekrement-Operatoren*

Abkürzende Schreibweisen 3/3

- Der Unterschied zwischen den Präfix- und den Postfix-Operatoren liegt im Zeitpunkt der Auswertung
- Tritt ein *Präfix*-Operator in einem umgebenden Ausdruck auf, so wird das Inkrement/Dekrement ausgeführt, *bevor* mit dem neuen Wert der Variablen *weitergerechnet* wird
- Tritt ein *Postfix*-Operator in einem umgebenden Ausdruck auf, so wird das Inkrement/Dekrement ausgeführt, *nachdem* mit dem alten Wert der Variablen *gerechnet* wurde

- Beispiel

```
int a, b = 1;
a = b++;          → b == ?,  a == ?     // weiter im gleichen Programm:
a = ++b;          → b == ?,  a == ?
```

Bemerkungen

- Die häufige *abwechselnde* Verwendung von Präfix-/Postfix-Inkrement und -Dekrement zu Verständnisschwierigkeiten beim Lesen von Programmen führen

  → *Vorsichtig einsetzen*!

- Die beiden Varianten werden hauptsächlich bei Schleifen und der Indizierung von Arrays benutzt

Boole‘sche Ausdrücke

- Der einfache Datentyp *boolean* umfasst (nur) die beiden Werte *true* und *false*
- Folgende Operatoren sind auf Boole‘sche Ausdrücke anwendbar; das Ergebnis ist immer vom Typ boolean

  ==
  Ergibt *true* genau dann, wenn beide Operanden *gleich* sind, sonst *false*
  !=, ^
  Ergibt *true* genau dann, wenn beide Operanden *ungleich* sind, sonst *false*
  &&
  Ergibt *true* genau dann, wenn beide Operanden *true* sind, sonst *false*
  ||
  Ergibt *true* genau dann, wenn mindestens ein Operand *true* ist, sonst *false*
  !
  Ergibt *true* genau dann, wenn der (einzige) Operand *false* ist, sonst *false*

Besonderheiten: *verkürzte Auswertung* von && und ||

- Der Operator **&&** liefert genau dann *true*, wenn beide Operatoren *true* sind, aber:
  Der rechte Operand wird *nur ausgewertet*, wenn der linke Operand zu *true* ausgewertet wurde
- Der Operator **||** liefert genau dann *true*. Wenn mindestens ein Operand *true* ist, aber:
  Der rechte Operand wird *nur ausgewertet*, wenn der linke Operand zu *false* ausgewertet wurde

→In beiden Fällen wird "unnötige Arbeit" vermieden!

Anwendungsbeispiel:

```
String[] names = new String[100];
// names[0] bis names[99] mit Werten belegen

String nameToSearch = readString();

int i = 0;
while (i < names.length && !names[i].equals(nameToSearch))
    i++;
```

Operator-Vorrang-Regelung: Motivation

- In Java (wie in jeder Programmiersprache *muss* festgelegt werden (wie auch in der Mathematik), welche Operatoren wie "wichtig" sind

- Beispiele:

  3 + 4 * 5 = ?

  ```
  int a;
  a = 3 + 4;
  ```

Auswertungsreihenfolge von Ausdrücken

- Ausdrücke in Java werden nach folgenden Regeln ausgewertet:
  - Der linke Operand wird komplett ausgewertet, bevor die Auswertung des rechten Operanden beginnt
    Bei den Zuweisungsoperatoren =, +=, -= etc. wird der rechte Operand komplett ausgewertet, bevor die Auswertung des linken Operanden beginnt
  - Alle Operanden eines Ausdrucks werden komplett ausgewertet, bevor der Operator auf sie angewendet wird
    Diese Regel gilt nicht für die Operatoren &&, ||
  - Ausdrücke mit mehreren Operatoren werden anhand ihrer Klammerung ausgewertet, ansonsten anhand der Operator-Vorrang-Regelung
    (s. nächste Folie)
- Zitat aus der Sprachbeschreibung von Java:
  "It is recommended that code not rely crucially on this specification."

**Das Setzen von Klammern hilft manchmal wesentlich,
Ausdrücke für Menschen eindeutig verständlich zu machen!**

Operator-Vorrang

Der Operator-Vorrang ist in Java wie folgt geregelt:

Operator-Typ	Operator
Einstellig	+ - ! ++ -- ~
Arithmetisch (und shift)	* / % + - << >> >>>
Relational	> < >= <= == !=
Logisch (und bitweise)	&& || & | ^
Konditional (dreistellig)	? :
Zuweisung	= (und ihre Kurzschreibweisen)

Das Setzen von Klammern hilft manchmal wesentlich,
Ausdrücke für Menschen eindeutig verständlich zu machen!

Nur der Vollständigkeit halber: die komplette Vorrang-Tabelle

1	++ -- ~ ! *cast*
2	+ - (einstellig)
3	* / %
4	+ - (zweistellig)
5	<< >> >>>
6	< > <= >= *instanceof*
7	== !=
8	&
9	^
10	|
11	&&
12	||
13	? :
14	= *= /= %= += -= <<= >>= >>>= &= ^= !=

(Einzelne) Zeichen

- In Java (wie in den meisten Programmiersprachen) gibt es nicht nur *Strings*, d.h. Zeichenketten, sondern auch einzelne Zeichen vom Datentyp *char* (engl. character)
- Einzelne Zeichen werden in einfache Hochkommata (= Apostroph; nicht: Akzent-Zeichen!) eingeschlossen
- Beispiele:

  'a'

  '$'

  '='

  'Ö'

  usw.

Zugriff auf einzelne Zeichen im String

- Die Zeichen innerhalb eines Strings können mit der Methode

  *String.*charAt(*int*)

  abgefragt werden; der Parameter gibt die Position des Zeichens im String an; die Positionsnummern beginnen bei 0
- Diese Methode liefert das Zeichen an der angegebenen Position in dem String, an den sich die Anfrage richtet
- Beispiele
  - String s = "abcde";
    s.charAt(3)

    →
  - "a".charAt(0)

    →
  - "".charAt(3)

    →
  - ("abc" + "123").charAt(4)

    →

Escape-Darstellungen 1

- Problem:
  Der Apostroph kann nicht „normal“ dargestellt werden:

```
char ch = 'a';

ch = '''';
```

- Die Lösung in Java (andere Sprachen, andere Lösungen):
  Um den Apostroph in einem Zeichen darzustellen, wird ihm das Zeichen \ (Backslash) vorangestellt:

```
char ch = 'a';

ch = '\'';
```

- Das ist die Schreibweise für *ein einzelnes Zeichen*!

Escape-Darstellungen 2

- Problem Nummer 2:
  Der Backslash kann jetzt nicht mehr „normal“ dargestellt werden:

```
char ch = 'a';

ch = '\';
```

- Die Lösung:
  Um den Backslash in einem Zeichen darzustellen, wird auch ihm das Zeichen \ (Backslash) vorangestellt:

```
char ch = 'a';

ch = '\\';
```

- Das ist wiederum die Schreibweise für *ein einzelnes Zeichen*!

Escape-Darstellungen 3

- Weitere spezielle, nicht-schreibbare *char*s werden ebenfalls mittels Escape-Sequenz dargestellt; bekannte Beispiele:

  \t horizontal tab HT, Tabulator

  \n linefeed LF, Zeilenumbruch und -vorschub

  \r carriage return CR, Zeilenumbruch

- Verwendung

```
println("Hallo");

// Entsprechung in Linux/MacOS:
print("Hallo\n");

// Entsprechung in Windows:
print("Hallo\r\n");
```

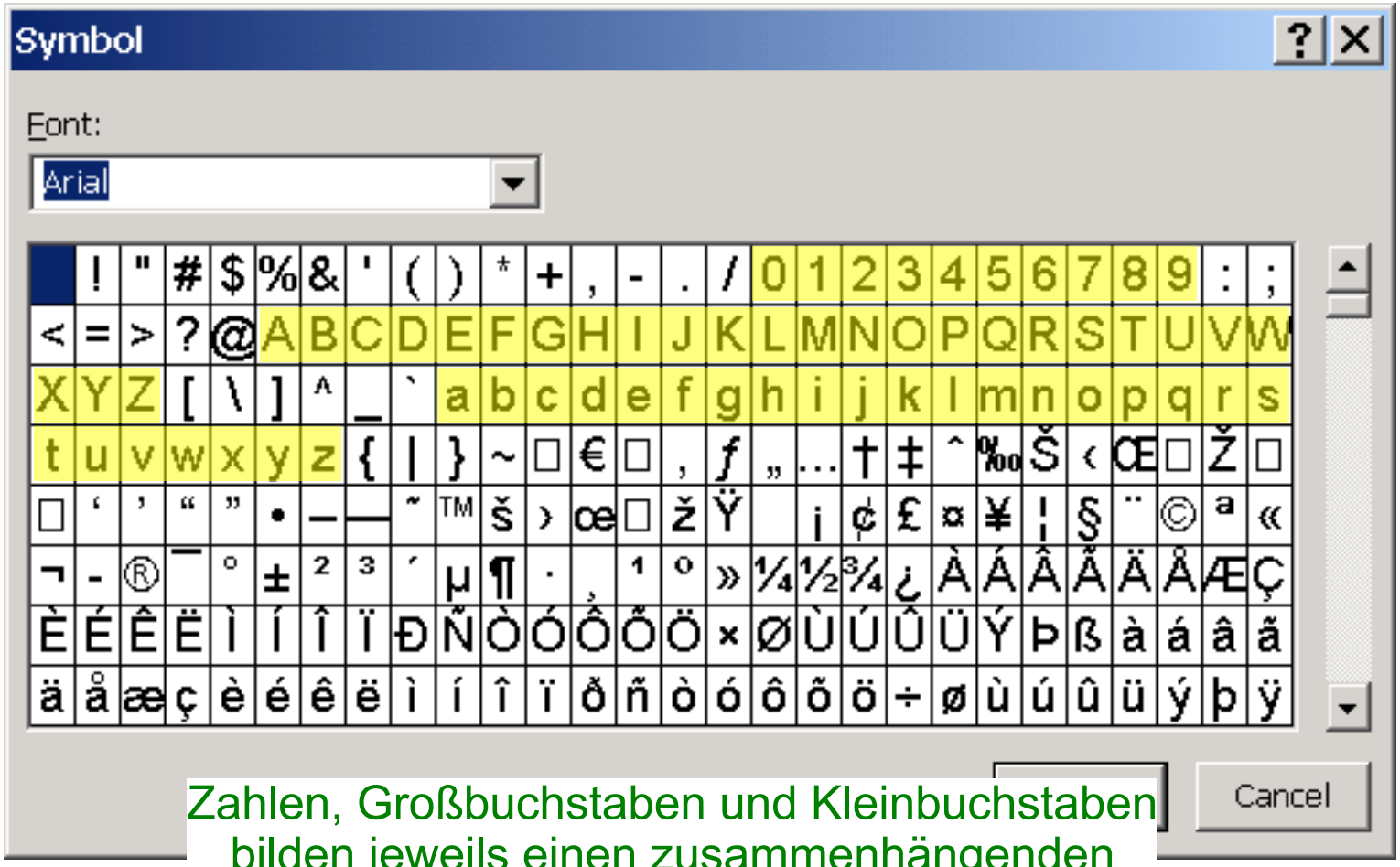
Der Unterschied wird oft (nur) in Textdateien bemerkt

Unicode

- Java stellt intern jedes Zeichen in *Unicode* dar, wofür 1 bis 4 Bytes verwendet werden
- In vielen *anderen* Programmiersprachen wird die *ASCII-Darstellung* verwendet, die mit einem Byte, also 8 Bit, auskommt (auskommen muss)
  (Nur) Die ersten 128 Zeichen in ASCII und Unicode entsprechen einander
- Vorteil von Unicode:
  Es können viel mehr Zeichen, z.B. Umlaute oder andere nationale Sonderzeichen, auch deutsche, französische, russische, asiatische etc. dargestellt werden

- Mehr Informationen zu Unicode: http://www.unicode.org

Zeichen-Anordnung



Zahlen, Großbuchstaben und Kleinbuchstaben bilden jeweils einen zusammenhängenden geordneten Bereich

Rechnen mit Zeichen

- Java behandelt auch Werte vom Typ *char* als ganze Zahlen, daher kann mit *char*s ähnlich wie mit Zahlen gerechnet werden
- Wertebereich von *char*:
  - Von 0 bis 65.535
- Typische Anwendung:

```
ch = readString().charAt(0);

while (ch < '0' || ch > '9')
    ch = readString().charAt(0);

int digit = ch - '0';

println(digit * 10);
```

Zugehörige Ein-/Ausgabe:

```
1
10
```

Rechnen mit Zeichen

- Nur zur Illustration:
  Der Unterschied zwischen Zeichen und Ziffern zeigt sich beim Rechnen

- Typische Anwendung:

```
ch = readString().charAt(0);

while (ch < '0' || ch > '9')
    ch = readString().charAt(0);

int digit = ch - '0';

println(digit * 10);

println(ch * 10);
```

Zugehörige Ein-/Ausgabe:

```
1
10
490
```

Merk-Regel:

Es ist ganz schlechter Stil,

Zeichen durch ihren

Dezimalcode (z.B. 97 statt 'a') darzustellen

(Punktabzug in den Übungen!)

Zeichen und ihr Dezimalcode

```
print("Haben Sie heute Geburtstag (j/n)? ");
ch = readString().charAt(0);

if (ch == 106)
    println("Herzlichen Glückwunsch!");
```

schlechtes **Beispiel**

Zeichen und ihr Dezimalcode

```
print("Haben Sie heute Geburtstag (j/n)? ");
ch = readString().charAt(0);

if (ch == 106)
    println("Herzlichen Glückwunsch!");


if (ch == 'j')
    println("Herzlichen Glückwunsch!");
```

schlechtes **Beispiel**

richtiges **Beispiel**

Erhoffter Lernerfolg:

Es kommt *nicht nur* darauf an,

***dass* ein Programm irgendwie tut,**

was es soll, *sondern auch,*

***wie* es das tut/wie es aufgebaut ist**

Sie wollen ja Informatiker werden, nicht Bastler!

Strings und String-Methoden

- Strings werden in Java durch doppelte Hochkommata begrenzt
- Problem:
  Die doppelte Hochkommata können in einem String nicht „normal“ dargestellt werden:

```
String s = "Ein String mit " Hochkommata";
```

- Die Lösung:
  Es wird das Zeichen \ (Backslash) vorangestellt:

```
String s1 = "Ein String mit \" Hochkommata";

String s2 = "Ein String mit \\ Backslash";
```

- Das ist die Schreibweise für *ein einzelnes Zeichen* im String!

Längenbestimmung von Strings

- Wir kennen bereits die Methode

  *String.*length()

  um die Länge eines Strings zu erfragen
- Die Methode liefert die Anzahl der Zeichen in dem String, an den sich die Anfrage richtet, als *int*-Wert
- Beispiele
  - "a".length() →
  - ("abc" + "123").length() →
  - "".length() →
  - "ab\"c\'d\"ef".length() →

Korrektes Vergleichen von Strings

- Wir kennen bereits die Methode

  *String.*equals(*String*)

  um den Inhalt zweier Strings zu vergleichen
- Die Methode liefert dann und nur dann *true*, wenn die beiden Strings gleich lang und Zeichen für Zeichen gleich sind, sonst liefert sie *false*

Strings vs. Array of Character

- Anders als in anderen Sprachen (C, Pascal etc.) sind in Java Array of Character, d.h. char[ ], *nicht* gleichbedeutend mit Strings
- Beispiel:
  Ausgehend von folgenden Vereinbarungen

```
char[ ] ca = new char[3];                    // Array
ca[0] = 'a';
ca[1] = 'b';
ca[2] = 'c';
String s = "abc";                            // String
```

  sind folgende Zuweisungen *nicht* zulässig

```
ca = s;                                      // Array = String
s = ca;                                      // String = Array
```

Umwandlung String → char[ ]

- Strings können einfach in Character-Arrays überführt werden
- Die Umwandlung erfolgt mittels einer *Methode* der *Klasse String*

```
String s = "text";
char[ ] ca;

ca = s.toCharArray();
ca = "dies ist auch ein String".toCharArray();
```

- Das Array mit den *char*s wird in der passenden Länge angelegt

Umwandlung char[ ] → String

- Character-Arrays können mittels *Konstruktor-Aufruf* mit dem Array als Parameter in einen String überführt werden:

```
char[ ] ca = new char[4];
ca[0] = 'T';
ca[1] = 'e';
ca[2] = 'x';
ca[3] = 't';

String s = new String( ca );

println( s );                    → "Text"
```

Unveränderliche Strings

- Strings in Java haben einen konstanten Wert, sie können *nicht verändert* werden, also ihre Zeichen können *nicht* einzeln modifiziert werden
  Es gibt also keine Methode setCharAt(...) oder ähnliche

- Strings können nicht über ein Zeilenende fortgesetzt werden, lange Strings, die schlecht in einer Zeile notiert werden können, können durch die String-Konkatenation zusammengebaut werden

Konstanten: Motivation

- Konstanten ermöglichen es, verschiedene voneinander abhängige Werte zentral zu ändern
- Beispiel

```
String[] nachnamen = new String[100];
String[] vornamen = new String[100];
int[] matrikelNummern = new int[100];
int[] punktzahl = new int[100];
```

- Was passiert, wenn 100 Einträge nicht mehr ausreichen?
  - Textsuche nach „100“ im gesamten Programm
  - Ersetzen derjenigen „100“er, die erhöht werden müssen
- Das ist mühselig und fehlerabhängig

Konstanten 2

- Bessere Variante

```
public class StudentenArrays {

    static final int STUDENTEN_ANZAHL = 100;

    public static void main(String[] args) {

        String[] nachnamen = new String[STUDENTEN_ANZAHL];
        String[] vornamen = new String[STUDENTEN_ANZAHL];
        int[] matrikelNummern = new int[STUDENTEN_ANZAHL];
        int[] punktzahl = new int[STUDENTEN_ANZAHL];
```

- Die Konstante „STUDENTEN_ANZAHL“ erlaubt es, an einer zentralen Stelle alle betroffenen Arraylängen zu steuern
- Zur Erinnerung:
  Namen von Konstanten werden vollständig in Großbuchstaben geschrieben und durch Unterstriche abgesetzt

Konstanten 3: Lesbarkeit; Vermeiden von „magic numbers“

```
public class Foo {
    public void setPassword(String password) {
        // don't do this
        if (password.length() > 7) {
            throw new InvalidArgumentException("password");
        }
    }
}
```

```
public class Foo {
    public static final int MAX_PASSWORD_SIZE = 7;

    public void setPassword(String password) {
        if (password.length() > MAX_PASSWORD_SIZE) {
            throw new InvalidArgumentException("password");
        }
    }
}
```

Konstanten 4: Lesbarkeit; Vermeiden von „magic numbers“

hochschule mannheim

Alle ganzzahligen Datentypen

- byte:
  - von -128 bis 127
  - Das entspricht von -2^7 bis 2^7-1
  - Das entspricht den Werten Byte.MIN_VALUE bis Byte.MAX_VALUE
- short:
  - von –32.768 bis 32.767
  - Das entspricht von -2^{15} bis $2^{15}-1$
  - Das entspricht den Werten Short.MIN_VALUE bis Short.MAX_VALUE
- char
  - von 0 bis 65.535
  - Das entspricht den Werten Character.MIN_VALUE bis Character.MAX_VALUE
- int:
  - von –2.147.483.648 bis 2.147.483.647
  - Das entspricht von -2^{31} bis $2^{31}-1$
  - Das entspricht den Werten Integer.MIN_VALUE bis Integer.MAX_VALUE
- long:
  - von –9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807, also von ca. $-9*10^{18}$ bis $9*10^{18}$
  - Das entspricht von -2^{63} bis $2^{63}-1$
  - Das entspricht den Werten Long.MIN_VALUE bis Long.MAX_VALUE

Die angegebenen Grenzen sind im jeweiligen Wertebereich enthalten

Variablen-Vereinbarungen

- Eine *Variablen-Deklaration* vereinbart einen *Namen für einen bisher ungenutzten Speicherbereich*
- Die Bits in diesem Speicherbereich repräsentieren dann den augenblicklichen Wert der Variablen mit diesem Namen
- Damit immer definiert ist, welchen Wert eine Variable (in einem Ausdruck) hat, wenn sie verwendet wird, *muss* jede Variable mit einem (definierten) Wert belegt werden, *bevor* sie verwendet werden darf

Gültiger Programmcode:

```
int zähler1, zähler2, summe;
zähler1 = 12;
zähler2 = 13;
summe = zähler1 + zähler2;

String nachname, vorname, name;
nachname = "Schmidt";
name = "Hans " + nachname;
```

Ungültiger Programmcode

```
int zähler1, zähler2, summe;
zähler1 = 12;
summe = zähler1 + zähler2;

String nachname, vorname, name;
nachname = "Schmidt";
name = vorname + " " +
nachname;
vorname = "Hans";
```

Variablen-Vereinbarungen

- Variablen werden bei ihrer Deklaration mit einem *Typ* versehen;
  dieser wird auch *statischer Typ* genannt
- Der Typ definiert
  - welche Werte die entsprechende Variable annehmen kann,
  - welche Operationen auf dieser Variablen erlaubt sind und
  - die Größe des zu reservierenden Speicherbereichs
- Vorausschau:
  Es gibt auch einen *dynamischen* Typ von Variablen…

Belegung von Variablen mit Werten

- Eine Variable in Java kann auf zwei Arten mit einem Wert belegt werden:
  - durch eine *Wertzuweisung* oder
  - durch eine *Initialisierung*

- Der Java-Compiler überprüft, dass jede Variable auf (zumindest) eine dieser Arten (mindestens einmal) einen Wert bekommen hat, bevor die Variable verwendet wird
- In einem Programm können einer Variable beliebig oft neue Werte zugewiesen werden:
  der Wert der Variablen ist *variabel*, daher der Name…
  im Unterschied zu *Konstanten*...