



Informatik I

Prof. Dr. Christian Pape

Kapitel 4

Einführung Programmieren mit
Java



Inhalt

- Klassen, main-Methode
- Lokale Variablen, Zuweisungen
- Primitive Datentypen
- Arithmetische Ausdrücke
- Boolesche Ausdrücke, if-else
- Methoden



Inhalt

- Ziel
 - Alle für Objekt-Orientierung (OO) Programmierung notwendigen Grundelemente von Java kennen lernen
 - Später:
 - OO Modellierung
 - OO Programmierung mit Java



Java

- 1995, Patrick Naughton, Mike Sheridan und **James Gosling**
- Erste Version: Oak (1992)
 - Eiche vor Fenster des Entwicklungsteam
- Plattformunabhängig
 - Virtuelle Maschine (virtuelles Betriebssystem)
 - Definierte Datentypen
- Interpretiert „Byte Code“ (virtueller Mikroprozessor)
- Für vernetzte Computer
 - Sicherheitskonzept
 - Netzwerkanbindung
- Verbreitung durch Unterstützung Netscape Browser
 - Applet



[Bild aus deutscher Wikipedia]

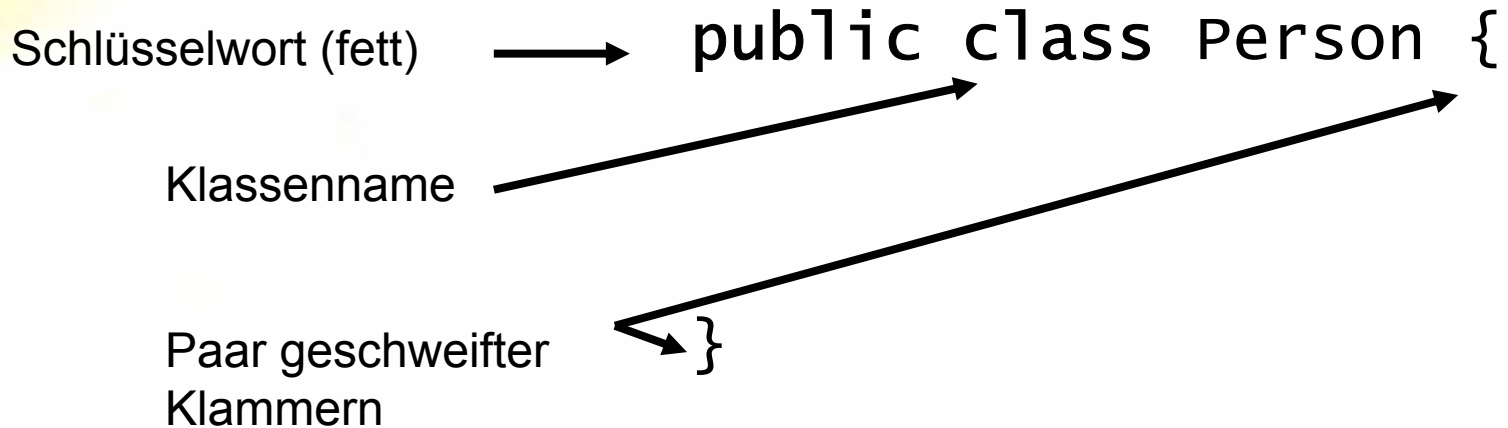


Java / Klassen

- Programmiersprache:
 - Syntaxdefinition
 - Semantik
- Syntax (Lehre vom Satzbau)
 - Beschreibt den Aufbau aller erlaubten Programm aus eine endlichen Anzahl von Symbolen
 - Im Folgenden: Umgangssprachlich + Beispiele
 - Formale Regelmechanismen für Syntax von Programmiersprachen: Theoretische Information
- Semantik (Bedeutung)
 - Was bedeuten einzelne Teile des Programms
 - Umgangssprachliche Erklärungen



Java / Klassen



- Klasse muss in Datei Person.java enthalten sein
- Vom Compiler erstellte Datei ist Person.class
- Namen für Klassen, Methoden, usw. heissen allgemein *Bezeichner* (engl. *identifier*)
- Schlüsselwörter sind als Bezeichner nicht erlaubt



Java / Bezeichner

- Bezeichner
 - Ist eine Folge (internationaler) alphanumerische Zeichen (alle Unicode Alphabete erlaubt)
 - a-z, A-Z, 0-9, ä, é, ب (Arabisch), β (Griechisch)
 - Unterstrich „_“ erlaubt
 - Kein Schlüsselwort erlaubt
 - Keine speziellen Literale wie `true`, `false`, `null` erlaubt
 - Keine Leerzeichen, Tabulatoren, Zeilentrenner, Steuerzeichen erlaubt
 - Kein Dollarzeichen „\$“ verwenden (soll nur in automatisch erzeugten Quelltext verwendet werden)
 - Gross-/Kleinschreibung verschieden (case-sensitive) („Abc“ ist ein anderer Bezeichner als „abc“)
- „aβθer_17“ ist ein gültiger Bezeichner
- Zwei Bezeichner sind identisch, wenn deren Unicode-Codierung identisch ist
 - b ist ungleich β
- Beliebig viele Leerzeichen vor und nach Bezeichner



Java / Bezeichner

gültig	ungültig
String	S t r i n g
i17	i&17
Αρετη	15465
MAX_VALUE	max-value
Class	class
\$DoNotUseDollar	



Java / Bezeichner

- Quelltext nicht immer übertragbar
- Arabisch wird rechts nach links gelesen (kuriöse Effekte im Editor)
- Zusatzwerkzeuge (Codegeneratoren) verstehen z.B. kein Griechisch

Konventionen Bezeichner

Verwende ausschliesslich a-z, A-Z, 0-9 und _ für Bezeichner

Verwende _ nur einmal in Folge und nie alleine

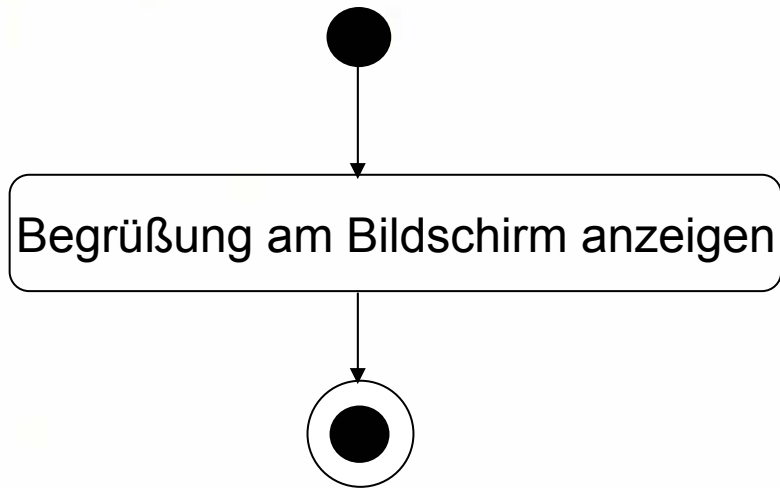


Java / Bezeichner Konventionen

Konvention beachtet	Konventionsverstoß
String	Αρετη
i17	—
MAX_VALUE	MAX__VALUE
Class	\$DoNotUseDollar



Ein sehr einfacher Verarbeitungsprozess



Java:

Prozess wird mit einer speziellen Methode gestartet (main-Methode)

Text mit Ausgabeanweisung anzeigen (Methode `System.out.println`)

Prozess wird nach sequentieller Ausführung aller Anweisungen beendet

Java / Programm-Anweisung

```
public class kleinstesAusfuehrbaresJavaProgramm {  
    public static void main(String [] args) {  
        System.out.println("Hallo");  
    }  
}
```

↑
Einzelne Programmanweisung mit abschliessendem Semikolon

- Ausführung dieses **Programms**:
 - Interpreter „java“ startet main() Methode
 - Anweisung wird ausgeführt
 - Danach: Programm beendet



Klassen

- **Klassennamen**
 - Baum
 - Temperatur
 - JavaProgramm

Konvention Klassennamen

Konvention wie Bezeichner

Möglichst keine Abkürzungen
(Teilwörter ausschreiben)

Möglichst Substantive als
Namen verwenden

Aussagekräftige Namen
verwenden

Erste Buchstabe gross



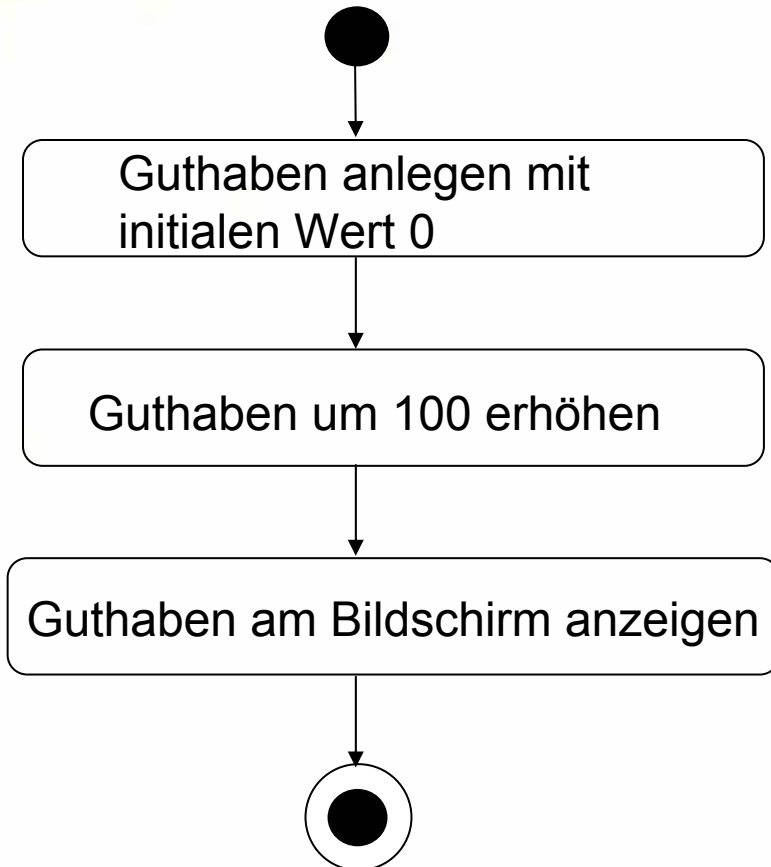
Inhalt

- Klassen, main-Methode
- Lokale Variablen, Zuweisungen
- Arithmetische Ausdrücke
- Primitive Datentypen
- Boolesche Ausdrücke, if-else
- Methoden



Ein einfacher Verarbeitungsprozess

Ablauf bei einem Bankkonto:



Java:

Prozess wird mit main() Methode gestartet

Lokale Variable als Datenspeicher

Datenspeicher ändern mit **Zuweisung**

Daten mit Ausgabeanweisung anzeigen

Lokale Variablen

```

public class Konto {
    public static void main(String [] args) {
        double guthaben = 0.0;
        guthaben = guthaben + 100.0;
        System.out.print("Guthaben = ");
        System.out.println(guthaben);
    }
}

```

Bezeichner

opt. initialer Wert

Typ double: 64 Bit Gleitkommazahl

■ Lokale Variable (Deklarationsanweisung)

- **Zwischenspeicher bzw. Behälter** für durch **Datentyp** definierte Werte
- Gültig nur innerhalb Methode, in der Variable deklariert wurde
- Initialer Wert kann bei Deklaration zugewiesen werden (bei double wissenschaftliches Format 123.456+e17)

■ Verwendung nur nach Deklaration

- in Zuweisungen (gleich), Ausdrücken, als Parameter in Methodenaufrufen (später)

Zuweisung

```
public class Konto {  
    public static void main(String[] args) {  
  
        double guthaben = 0.0;  
        guthaben = guthaben + 100.0;  
        System.out.println ("Guthaben      = " + guthaben);  
  
    }  
}
```

■ Zuweisungsanweisung

- Linke Seite: Name der Variable
- Rechte Seite: **Ausdruck** mit Werten und/oder Variablennamen
- Dazwischen: Zuweisungsoperator „=“
- Rechte Seite wird ausgewertet, berechneter Wert wird in Speicherplatz der Variablen geschrieben
- Vorheriger Wert Variable durch neuen Wert ersetzt
- Name einer Variablen steht symbolisch für dessen Wert (Ausnahme Variable ist auf linker Seite einer Zuweisung, dann bezeichnet Name den Speicherplatz für den Wert)



Zuweisung



```
double guthaben = 0.0;
guthaben = guthaben + 100.0;
```

Vor Ausführung Deklaration
existiert noch keine lokale Variable
guthaben im Hauptspeicher



```
double guthaben = 0.0;
guthaben = guthaben + 100.0;
```

guthaben

0.0

Nach Deklaration ist der Speicher abgelegt
Und die lokale Variable mit 0.0 initialisiert



```
double guthaben = 0.0;
guthaben = guthaben + 100.0;
```

guthaben

100.0

Nach der Zuweisung ist der neue berechnete Wert
(guthaben + 100.0) der Speicherzelle zugewiesen

Lokale Variablen

- Lokale Variablen
 - besitzen symbolischen Namen (Bezeichner)
 - haben Platz für einen Wert eines bestimmten Datentyps
 - Müssen in einer Methode *deklariert* werden:
Typ Bezeichner [= Wert];
`double guthaben = 0.0;`
 - müssen vor Benutzung (bei Zuweisung, in Ausdrücken) deklariert werden
 - sind nur innerhalb Methode, in der sie deklariert sind gültig
 - müssen innerhalb Methode eindeutigen Namen besitzen
 - Können an – fast – beliebiger Stellen innerhalb Methode deklariert werden (nicht nur am Anfang der Methode)
 - Können bei Deklaration mit Wert (ein Ausdruck) initialisiert werden
 - Ohne Initialisierung mit = wird Variable mit Wert 0 initialisiert
- Zuweisungen und Deklarationen sind Anweisungen
 - Semikolon zum Abschluss der Anweisung nötig
`guthaben = guthaben + 100.0 ;`



Lokale Variablen

Konvention beachtet	Konventionsverstoß
zahl	ζαηλ
kraftfahrzeug	Kfz
pi	iNPIUmrechnen
donauDampfschif fartsGesellscha ft	z64

Konvention Variablennamen

Konvention wie Bezeichner

Alle Buchstaben klein, bis Anfangsbuchstaben von Teilwörtern (ausser erstes)

Möglichst keine Abkürzungen (Teilwörter ausschreiben)

Möglichst Substantive als Namen verwenden

Aussagekräftige Namen verwenden



Inhalt

- Klassen, main-Methode
- Lokale Variablen, Zuweisungen
- **Primitive Datentypen**
- Arithmetische Ausdrücke
- Boolesche Ausdrücke, if-else
- Methoden



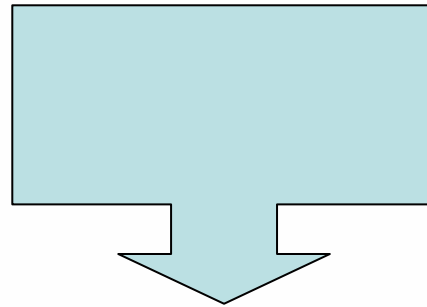
Primitive Datentypen

- 8 Datentypen (Daten: unteilbare Information)

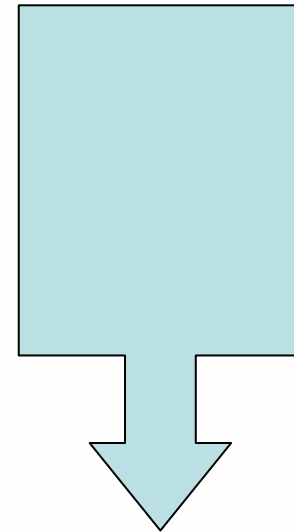
byte | **short** | **int** | **long** | **float** | **double** | **boolean** | **char**



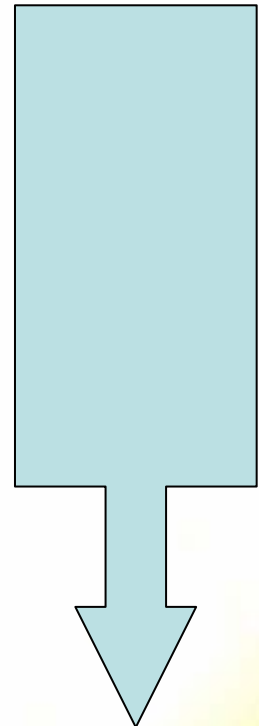
Typen für *ganzzahlige* Werte
 8 16 32 64
 Bit 2er Komplement



Typen für *Gleitkommawerte*
 32 64
 Bit IEEE Format



Typen für *Boolesche Werte*
 true false 8 Bit



Typen für *Unicodezeichen*
 16 Bit



Primitive Datentypen: int

- **Syntax**
 - Folge von Ziffern, beliebig viele führende Nullen
- **Semantik:**
 - 32 bit grosse ganze Zahl mit Vorzeichen
 - Wertebereich
-2 147 483 648 bis 2 147 483 647 (inklusive)

```
int temperatur = 0000;
```

```
int richtgeschwindigkeit = 130;
```

```
int minusEins = -1;
```



Primitive Datentypen: long

- Syntax
 - Folge von Ziffern, beliebig viele führende Nullen mit **nachfolgendem l oder L**
- Semantik:
 - 64 bit grosse ganze Zahl mit Vorzeichen
 - Wertebereich
-9 223 372 036 854 775 808 bis 9 223 372 036 854 775 807
(inklusive)

```
long temperatur = 00001;
```

```
long richtgeschwindigkeit = 130L;
```

```
long minusEins = -1l;
```



Primitive Datentypen: byte, short

- Syntax
 - Folge von Ziffern, beliebig viele führende Nullen
- Semantik:
 - 8 bzw. 16 bit grosse ganze Zahl mit Vorzeichen
 - Wertebereich
 - 128 bis 127 (byte)
 - 32 768 bis 32767 (short)

```
byte temperatur = 0000;
```

```
short richtgeschwindigkeit = 130;
```

```
byte minusEins = -1;
```



Primitive Datentypen

- Vorsicht beim Rechnen mit ganzen Zahlen in Java!

```
public static void main(String argv[]) {  
    int maximumInt = 2147483647;  
  
    maximumInt = maximumInt + 1;  
  
    System.out.println(maximumInt);  
}
```

Ausgabe ist - 2147483648

- „The *integral types* are byte, short, int, and long, whose values are 8-bit, 16-bit, 32-bit and 64-bit **signed two’s-complement integers**“ [Java Language Specification, 2.0, Seite 33]
- „The built-in integer operators do not indicate overflow or underflow in any way.“ [Seite 34]



Primitive Datentypen

- 8 Datentypen (Daten: unteilbare Information)

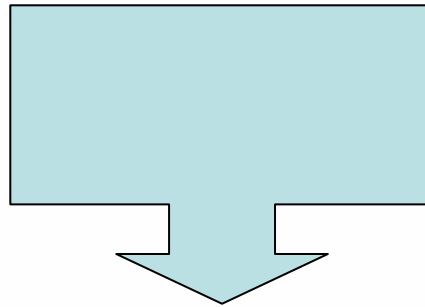
byte | **short** | **int** | **long** | **float** | **double** | **boolean** | **char**



Typen für *ganzzahlige* Werte

8 16 32 64

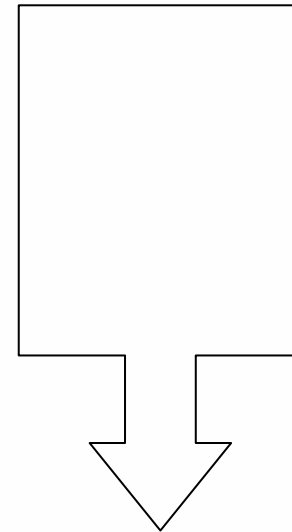
Bit 2er Komplement



Typen für *Gleitkommawerte*

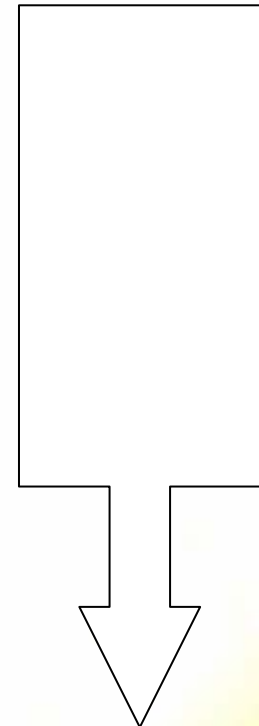
32 64

Bit IEEE Format



Typen für *Boolesche Werte*

true false



Typen für *Unicodezeichen*
16 Bit



Primitive Datentypen

- Syntax Gleitkommazahlen
 - Gleitkommazahlen *immer* mit Dezimalpunkt
0,123 als 0.123
-165,032 als -165.032
 - Führende 0 ist optional
0,123 als .123
 - Angabe in wissenschaftlicher Notation
0,123 als 1.23e-1, 0.123E-2, 0.0123e-3, ...
1 000 000 als e+6 oder 10e+5
- Bei float: nachgestelltes f oder F

```
double temperatur = 0.0;
```

```
float richtgeschwindigkeit = 130f;
```

```
double minusEins = -1;
```



Primitive Datentypen

- Semantik: „reelle Zahlen“
- Typ double
 - Arithmetic, ANSI/IEEE Standard 754-1985
 - Gleitkommazahl, 64 Bit
 - 1 Bit Vorzeichen V , 52 Bit Mantisse M , 11 Bit Exponent E (-1022 bis 1024)
$$(-1)^V \cdot M \cdot 2^E$$
- Typ float
 - 32 Bit Gleitkommazahl
- Details Gleitkommazahlen siehe Technische Informatik



Umrechnung von Geschwindigkeiten

■ Problemstellung

□ Gegeben:

Geschwindigkeit in Kilometer pro Stunde

□ Gesucht:

Geschwindigkeit in Meilen pro Stunde und

Geschwindigkeit in Meter pro Sekunde

■ $1 \text{ km} = 0,621 \text{ Meilen}$

■ $1 \text{ m/s} = 3,6 \text{ km/h}$

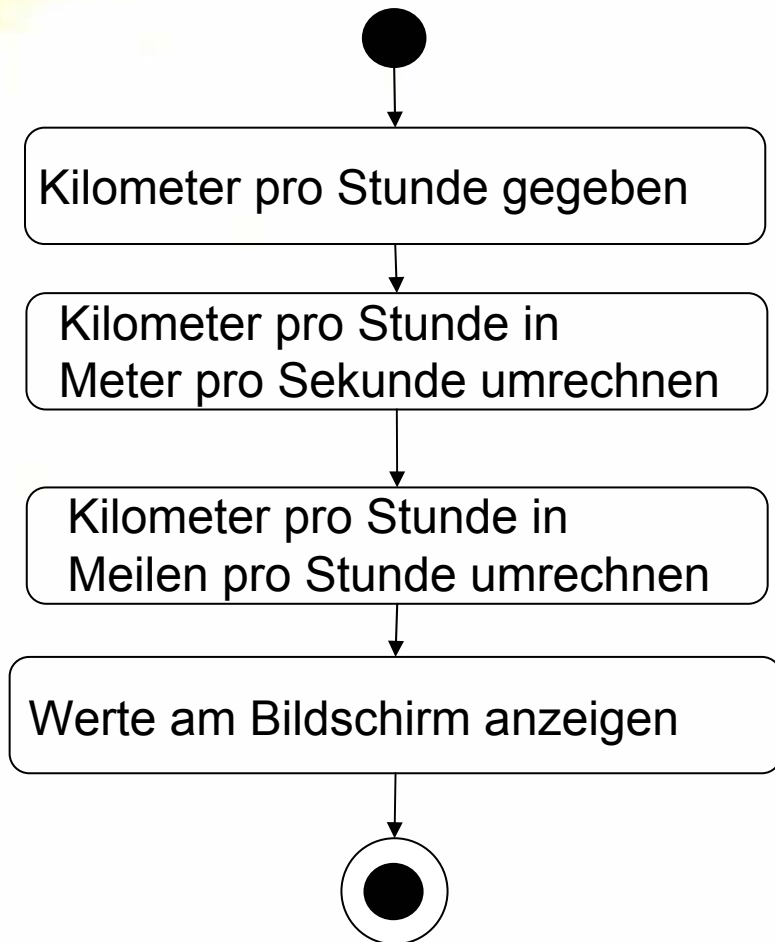
$$1 \text{ m} / 1 \text{ s} = 1000 \text{ m} / 1000 \text{ s} = 3,6 \text{ km} / 3600 \text{ s} = 3,6 \text{ km} / \text{h}$$

$$\text{also } 1 \text{ km/h} = 1 / 3,6 \text{ m/s}$$



Primitive Datentypen

■ Umrechnung von Geschwindigkeiten



Java:

Gegeben als lokale Variable

Weitere lokale Variable initialisiert mit umgerechneten Wert in m/s

Weitere lokale Variable initialisiert mit umgerechneten Wert Meilen/h

Ausgabeanweisungen für die drei lokalen Variablen



Primitive Datentypen

```
public class GeschwindigkeitsUmrechnung {  
  
    public static void main(String [] argv) {  
        double kilometerProStunde = 120.0;  
        double meterProSekunde = kilometerProStunde / 3.6;  
        double meilenProStunde = kilometerProStunde * 0.621;  
  
        System.out.println(kilometerProStunde + " km/h sind:");  
        System.out.println(meterProSekunde + " m/s und ");  
        System.out.println(meilenProStunde + " Meilen/h.");  
    }  
}
```

Ausgabe: 120.0 km/h sind:
33.33333334 m/s und
74.52 Meilen/h.



Primitive Datentypen

Typ	Bits	Initialer Wert	Min	Max
byte	8	0	-128	127
short	16	0	-32 768	32 767
int	32	0	-2 147 483 648	2 147 483 647
long	64	0l	-9 223 372 036 854 775 808 $< -9 \cdot 10^{18}$	9 223 372 036 854 775 807 $> 9 \cdot 10^{18}$



Primitive Datentypen

Typ	Bits	Initialer Wert	Min positiver Wert	Max
float	32	0.0f	1.40239846e-45f	3.40282347e+38f
double	64	0.0	4.9406564584124654 4e-324	1.7976931348623157 0e+308

- Kleinste Zahl entsprechend (lediglich negatives Vorzeichen)
- Grösste negative Zahl gleich Negation kleinster positiven Zahl

Primitive Datentypen (char)

- Intern 16bit Unicode
- **Angabe einzelnes Zeichen in einfachen Hochkommata**
- **Direkte Angabe Unicode in Hexadezimalform (genau 4 Ziffern)**
- Keine Operatoren für char (Zeichen sind keine Zahlen)
- Zeichenketten (String) in Java sind Folge von char Werten (“dies ist eine Zeichenkette“)
- Default ist `'\u0000'` (ASCII Steuerzeichen NUL, dies markiert in C,C++ , aber nicht in Java, das Ende einer Zeichenkette)

```
char kleinesA = 'a';
```

```
char oe = '\u00D8'; // Ø
```



Primitive Datentypen (boolean)

- Intern 1 Byte
- Nur zwei Werte möglich: true false (**Literale**)
- Literal: Vordefinierte Bezeichner, der einen Wert darstellt
- Default Wert ist false
- Bilden Boolescher Ausdrücke mit Booleschen Operatoren
- Keine Umwandlung in Zahlen möglich (auch nicht mit cast)

```
boolean wahr = true;
```

```
boolean falsch; // Default ist false
```



Primitive Datentypen

Konvention primitive Datentypen

- Verwende möglichst **int** (oft am schnellsten, kein L/I notwendig) bei ganzen Zahlen
- Verwende **double** statt **float**
- Vermeide wissenschaftliche Notation bei **float/double**



Inhalt

- Klassen, main-Methode
- Lokale Variablen, Zuweisungen
- Primitive Datentypen
- **Arithmetische Ausdrücke**
- Boolesche Ausdrücke, if-else
- Methoden



Arithmetische Ausdrücke

- Komplexere Berechnungen mit arithmetischen Ausdrücken
 - Grundrechenarten uvm.
 - Regeln, nach denen Ausdrücke aufgebaut sind (**Syntax**)
 - Auswertungsreihenfolge (**Semantik**) der Ausdrücke
- Im Prinzip wie mit einem wissenschaftlichen Taschenrechner



Arithmetische Ausdrücke (Syntax)

- Eine Zahl ist ein Ausdruck
 - Führende Nullen erlaubt
- Eine Variable ist ein Ausdruck
- **Binäre Arithmetische Operatoren**
 - Addition +, Subtraktion –
 - Multiplikation *, Division /
 - Linke und rechte Seite sind jeweils Ausdrücke: ergibt wieder ein Ausdruck
- **Unären Arithmetischer Operator**
 - - (Negation)
- Klammerung nach Belieben um einen Ausdruck herum

Arithmetische Ausdrücke (Syntax)

- Eine Zahl (int) ist ein Ausdruck
 - Führende Nullen erlaubt
- Eine Variable ist ein Ausdruck
- **Binäre Arithmetische Operatoren**
 - Addition +, Subtraktion –
 - Multiplikation *, Division /
 - Linke und rechte Seite sind jeweils Ausdrücke: ergibt wieder ein Ausdruck
- **Unären Arithmetischer Operator**
 - - (Negation)
- Klammerung nach Belieben um einen Ausdruck herum

7 63

007

guthaben

7 + 63

guthaben - 7

7 * 63

63 / 7

7 + 63 - 63 / 7

- 732

-732

((guthaben - 7) * 8 + ((9)))

Arithmetische Ausdrücke (Semantik)

- $+$, $-$, $*$, $/$ fast „normale“ Semantik
- Ganze Zahlen
 - $1 + 4$ ergibt 5
 - $127 + 1$ ergibt -128 bei byte
- Gleitkommazahlen
 - $16.5 - 0.5$ ergibt 16,0
 - $10 * 0.1$ ergibt **nicht** 1,0 sondern fast 1 (0,99999999999998),
Rechenungenauigkeiten bei Konvertierung in binäre Darstellung,
siehe TI 1
- Division $/$, Division mit Rest $\%$ (bei ganzen Zahlen)
 - $1 / 2$ ergibt 0 (0 Rest 1)
 - $1 \% 2$ ergibt 1 (0 Rest 1)
 - $13 / 5$ ergibt 2 (2 Rest 3)
 - $13 \% 5$ ergibt 3

Arithmetische Ausdrücke (Semantik)

- Lösungen (Überlauf bei 2er Komplement)
 1. Vor Operation Werte auf $a + b < 128$ prüfen **ohne** $a+b$ zu berechnen
 2. Nach short casten und $a + b < 128$ prüfen (bei long auf float casten)
- In der Regel keines von beiden machen

1.
????

2.
byte c;
byte a = 100;
byte b = 110;
if ((short) a + (short) b < 128) {
 c = a + b;
}

Arithmetische Ausdrücke (Semantik)

- Gleitkommazahlen
 - $10 * 0.1$ ergibt **nicht** 1,0 sondern fast 1 (0,99999999998)
 - Insbesondere ist $10 * 0.1$ **nicht gleich** 1.0 in Java (oder C, C++, C#, ...)
 - Vorsicht: Compiler optimiert konstante Ausdrücke wie $10 * 0.1$ zu 1.0
 - Statt Gleichheit immer prüfen, ob Betrag der Differenz nahe an 0.0 ist

Gleitkommazahl a ist „gleich“ b
genau dann, wenn $|a - b| < 0.000001$

In Java: `Math.abs(a-b) < 0.000001`

Arithmetische Ausdrücke (Semantik)

- Auswertung durch Java Interpreter **sequentiell** von **links** nach **rechts** (Linksassoziativität) bei Operatoren mit gleichem Vorrang
 1. 1 + 2 - 5 + 8 ergibt zur Laufzeit 3 - 5 + 8
 2. 3 - 5 + 8 ergibt -2 + 8
 3. -2 + 8 ergibt 6
- Auswertungsreihenfolge spielt nur eine Rolle, wenn im Ausdruck selbst Variablenwerte geändert werden (im obigen Beispiel nicht der Fall)
 - Nur Fall, wenn Zuweisung in Ausdrücken verwendet wird (Wert einer Zuweisung ist Ergebnis der rechten Seite der Zuweisung)
 - Bei den speziellen Inkrement / Dekrementoperatoren
 - Bei Funktionen in Term, die Variablenwerte ändern

Arithmetische Ausdrücke (Semantik)

- Auswertung von Ausdrücken links nach rechts (Linksassoziativität) **gemäß Vorrang der Operatoren und Klammerung**
 - Falls rechter Ausdruck geklammert ist oder der Operator im rechten Ausdruck eine höhere Priorität hat, dann wird erst der rechte Ausdruck ausgewertet
 - „Punkt vor Strichrechnung“
- $1 + 2 * (3 - 2)$
 - * hat Vorrang vor +: $2 * (3 - 2)$ wird ausgewertet
 - Geklammerter Ausdruck rechts * hat Vorrang: $(3 - 2)$ wird ausgewertet
 - 1. $1 + 2 * \underline{(3 - 2)}$ ergibt $1 + 2 * 1$
 - 2. $1 + \underline{2 * 1}$ ergibt $1 + 2$
 - 3. $\underline{1 + 2}$ ergibt 3
- Vorrang der Operatoren
 - Operatoren +, - haben niedrigere Priorität als /, * und %
 - +, - untereinander gleiche Priorität, *, /, % ebenso
 - Im Zweifelsfall Klammern () setzen



Arithmetische Ausdrücke

- Division mit Rest etwas genauer
 - % bei ganzen Zahlen definiert durch
$$a \% b = a - (a / b) * b$$
 - $5 \% (-3)$ ist 2,
da $5 - (5 / -3) * (-3) = 5 - 3 = 2$
 - $-5 \% 3$ ist -2
 - $-5 \% -3$ ist -2
 - Falls $b = 0$ gilt, dann wird eine `ArithmeticException` geworfen (auch bei /)
 - Ergebnis $a \% b$ ist immer kleiner als Divisor b



Arithmetische Ausdrücke

- Division mit Rest `%` existiert auch für Gleitkommazahlen
- $15.7 \% 4.3 = \text{????}$
- Wie ist die Semantik?
 1. 3 (abgerundet: $15 \% 4$)
 2. 0 (Werte gerundet: $16 \% 4$)
 3. $3,6511627\dots$ (wie $15.7 / 4.3$)
 4. Was ganz anderes



Arithmetische Ausdrücke

- Division mit Rest `%` existiert auch für Gleitkommazahlen
- $15.7 \% 4.3 = 3$
- Antwort 1
- Nachkommaanteil wird abgeschnitten
- Ergebnis wie bei ganzen Zahlen
- Aber
 - Bei Division durch 0 keine `ArithmeticException`
 - Ergebnis ist dann `NaN`



Arithmetische Ausdrücke

- Bei Zahlen: **Implizite** Konvertierungsregeln
 - Werte / Zahlen geringer Genauigkeit werden zu Werten höherer Genauigkeit konvertiert (widening conversion)
 - Keine Konvertierung bei Umkehrung, da Präzisionsverlust der Werte (narrowing conversion)
 - float nach double sowie byte nach short, short nach int, int nach long
 - Ganzzahlige Typen nach Gleitkommatypen
 - Bei arithmetischen Operatoren: Typ Gesamtausdruck ist Typ des genauesten Teilausdrucks
- Konvention
 - Mischen verschiedener Zahltypen vermeiden

```
double guthaben = 0.0f; // float-wert nach double  
long guthaben = 170; // int-wert nach long
```

```
float guthaben = 0.0; // Compilerfehler  
int guthaben = 170l; // Compiler
```

```
(1l + 10) * 0.5f; // Ergebnistyp float
```

long

Arithmetische Ausdrücke

- Bei Zahlen: **Explizite** Konvertierungsregeln (**casting**)
 - Bei narrowing conversion
 - Präzisionsverlust
 - Abschneiden Nachkommaanteil bei Konvertierung einer Gleitkommazahl zu einer ganzen Zahl (keine Rundung)
 - Etwas Genauer
 - Von double, float nach long, int, short, byte wird immer erst nach long konvertiert
 - Falls double, float Wert zu groß / zu klein (negativ) war, dann wird größte/kleinste long reps. int Wert genommen (dito. bei double -> float)
 - Höherwertige Bits des Ergebnis werden bei byte, short verworfen
 - Angabe des gewünschten Datentyps in Klammern vor Wert/Ausdruck

```
float guthaben = (float) 0.0; // Ergebnis ist 0.0f
int guthaben = (int) 170.50; // Ergebnis ist 170
(int) ((11 + 10) * 0.5f * 2); // Ergebnis ist 11
```

```
byte b = (byte) -128.50; // Ergebnis ist -128
byte b = (byte) -129.50; // Ergebnis ist 127
int i = (int) 1000000000000000000.0; // 2147483647
long i = (long) -1.0e20; // -9223372036854775808
```



Arithmetische Ausdrücke

- Kein direkte Umrechnung von char in Zahlen möglich
 - narrowing conversion, cast nötig
 - Die höherwertigen Bits werden einfach verworfen

```
char oe = '\u00D8'; // Ø
```

```
int positiveZahl = (int) 'Ø';
```

```
byte negativeZahl = (byte) 'Ø';
```

Arithmetische Ausdrücke

- Formatierung von Ausdrücken
 - Immer ein Leerzeichen vor und nach jedem zweistelligen Operator
 - Kein Leerzeichen bei Klammern
- Umbrechen überlanger Ausdrücke (bei 80-100 Zeichen)
 - **vor** Operator mit **geringster Bindung** (Klammerung berücksichtigen)
 - **ca. 8 Zeichen** bzw. bis zugehöriger linken Teilformel einrücken
 - Trennlinie im Editor einstellen!
- Regeln dienen der Lesbarkeit des Quelltextes

$$1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11$$


$$1 + 2 + 3 + 4 + 5 + 6 + 7 + 8$$

$$+ 9 + 10 + 11$$

$$a * a * a + 3 * a * a * b + 3 * a * b * b + b * b * b$$


$$a * a * a + 3 * a * a * b$$

$$+ 3 * a * b * b$$

$$+ b * b * b$$

Arithmetische Ausdrücke

- Umgebrochene Teilformel einrücken bis zur zugehörigen anderen Teilformel (falls möglich)

$$x1 * 3 + (a + b + 6) * ((x2 + 8) + x3 / 11)$$


$$x1 * 3 + (a + b + 6) \\ * ((x2 + 8) \\ + x3 / 11)$$

- Automatische Formatierung von Quelltext in Eclipse
 - Quelltext markieren
 - Source -> Format auswählen
 - Ergebnis überprüfen, es wird nach dem eingestellten Codestil formatiert



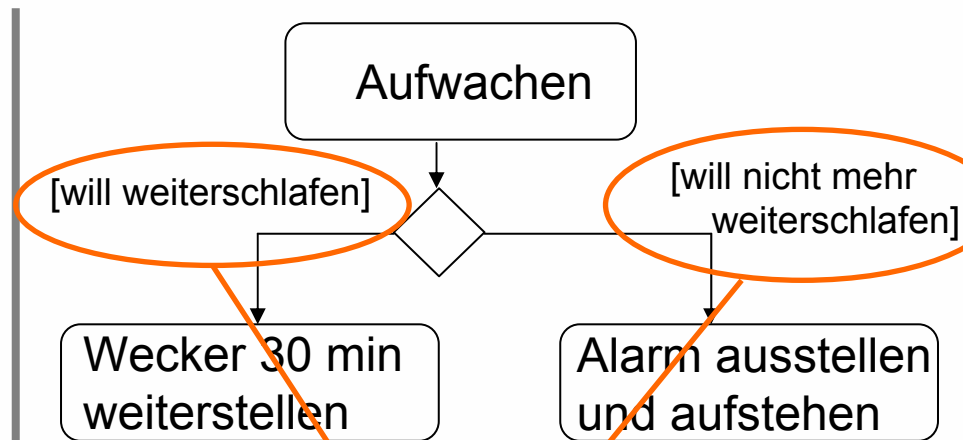
Inhalt

- Klassen, main-Methode
- Lokale Variablen, Zuweisungen
- Arithmetische Ausdrücke
- Primitive Datentypen
- **Aussagenlogik**
- Kontrollanweisung: if-else
- Boolesche Ausdrücke
- Methoden



Aussagenlogik

- **Zweiwertige Logik:**
 - Zwei Wahrheitswerte
True, false / Ja, Nein
- **Einfache Aussagen**
 - Will weiterschlafen
 - Morgen regnet es
- **Verneinte Aussagen**
 - Will **nicht** weiterschlafen
 - Morgen regnet es **nicht**
- **Logische Verknüpfungen**
 - Will weiterschlafen **und** morgen regnet es nicht (*Konjunktion*)
 - Morgen regnet es **oder** morgen regnet es nicht (*Disjunktion*)
 - **Entweder** morgen regnet es **oder** morgen regnet es nicht (*Antivalenz*)
 - **Wenn** es morgen regnet, **dann** will ich weiterschlafen (*Implikation*)



- **Bedingungen**, die **Aussagen** über *reale* Ereignisse machen
- Software implementiert *reale* Prozesse oder Abläufe
- Aussagen kann entweder **wahr** oder **falsch** sein

Drei (oder mehr) -wertige Logiken

Möchtest Du mit ins Kino gehen?

Ja, Nein, Vielleicht



Aussagenlogik

Java

<ul style="list-style-type: none"> ■ Zweiwertige Logik: <ul style="list-style-type: none"> □ Zwei Wahrheitswerte True, false 	<p style="text-align: center;">true false</p>
<ul style="list-style-type: none"> ■ Einfache Aussagen <ul style="list-style-type: none"> □ Will weiterschlafen □ Morgen regnet es 	<p style="text-align: right;">Boolesche Variablen</p> <pre>boolean willWeiterschlafen; boolean morgenRegnetEs = true;</pre>
<ul style="list-style-type: none"> ■ Verneinte Aussagen <ul style="list-style-type: none"> □ Will nicht weiterschlafen □ Morgen regnet es nicht 	
<ul style="list-style-type: none"> ■ Logische Verknüpfungen <ul style="list-style-type: none"> □ Will weiterschlafen und morgen regnet es nicht □ Morgen regnet es oder morgen regnet es nicht □ Entweder morgen regnet es oder morgen regnet es nicht □ Wenn es morgen regnet, dann will ich weiterschlafen 	



Aussagenlogik

Java

<ul style="list-style-type: none"> ■ Zweiwertige Logik: <ul style="list-style-type: none"> □ Zwei Wahrheitswerte True, false 	<p style="text-align: center;">true false</p>
<ul style="list-style-type: none"> ■ Einfache Aussagen <ul style="list-style-type: none"> □ Will weiterschlafen □ Morgen regnet es 	<p style="text-align: right;">Boolesche Variablen</p> <pre>boolean willWeiterschlafen; boolean morgenRegnetEs = true;</pre>
<ul style="list-style-type: none"> ■ Verneinte Aussagen <ul style="list-style-type: none"> □ Will nicht weiterschlafen □ Morgen regnet es nicht 	<p style="text-align: right;">Negation</p> <pre>! willWeiterschlafen ! morgenRegnetEs</pre>
<ul style="list-style-type: none"> ■ Logische Verknüpfungen <ul style="list-style-type: none"> □ Will weiterschlafen und morgen regnet es nicht □ Morgen regnet es oder morgen regnet es nicht □ Entweder morgen regnet es oder morgen regnet es nicht □ Wenn es morgen regnet, dann will ich weiterschlafen 	



Aussagenlogik

Java

<ul style="list-style-type: none"> ■ Zweiwertige Logik: <ul style="list-style-type: none"> □ Zwei Wahrheitswerte True, false 	<p style="text-align: center;">true false</p>
<ul style="list-style-type: none"> ■ Einfache Aussagen <ul style="list-style-type: none"> □ Will weiterschlafen □ Morgen regnet es 	<p style="text-align: right;">Boolesche Variablen</p> <pre>boolean willWeiterschlafen; boolean morgenRegnetEs = true;</pre>
<ul style="list-style-type: none"> ■ Verneinte Aussagen <ul style="list-style-type: none"> □ Will nicht weiterschlafen □ Morgen regnet es nicht 	<p style="text-align: right;">Negation</p> <pre>! willWeiterschlafen ! morgenRegnetEs</pre>
<ul style="list-style-type: none"> ■ Logische Verknüpfungen <ul style="list-style-type: none"> □ Will weiterschlafen und morgen regnet es nicht □ Morgen regnet es oder morgen regnet es nicht □ Entweder morgen regnet es oder morgen regnet es nicht □ Wenn es morgen regnet, dann will ich weiterschlafen 	<pre>willWeiterschlafen & morgenRegnetEs willWeiterschlafen morgenRegnetEs willWeiterschlafen ^ morgenRegnetEs</pre> <p style="text-align: center;">Gibt es in Java nicht (direkt)</p>



Aussagenlogik

Java

true

false

Boolesche Variablen

```
boolean willweilerschlafen;
boolean morgenRegnetEs = true;
```

Negation

true	←	! false
false	←	! true

! willweilerschlafen

! morgenRegnetEs

willweilerschlafen & morgenRegnetEs

willweilerschlafen | morgenRegnetEs

willweilerschlafen ^ morgenRegnetEs



Aussagenlogik

Java

true

false

Boolesche Variablen

```
boolean willweilerschlafen;
boolean morgenRegnetEs = true;
```

Negation

true ← ! false
false ← ! true

```
! willweilerschlafen
! morgenRegnetEs
```

false false & true
true false | true
false false ^ true

```
willweilerschlafen & morgenRegnetEs
willweilerschlafen | morgenRegnetEs
willweilerschlafen ^ morgenRegnetEs
```



Aussagenlogik

& (und)	true	false
true	true	false
false	false	false

 (oder)	true	false
true	true	true
false	true	false

^ (ex. oder)	true	false
true	false	true
false	true	false

! (nicht)	
true	false
false	true



Aussagenlogik

- **Boolesche Algebra**
 - 0 (false) und 1 (true)
- **Gebräuchliche Operatoren Schreibweisen**

	Konjunktion	Disjunktion	Antivalenz	Implikation	Negation
Hardwareentwurf	•	+	\oplus		\neg
Logik (USA)	\cup	\cap		\supset	
Logik (Europa)	\wedge	\vee	\Leftrightarrow	\Rightarrow	\neg



Aussagenlogik

■ Regeln (Überprüfung mit Wahrheitstafeln)

- Doppelte Negation

$$\neg \neg A = A$$

- Tertium Non Datur

$$\neg A \wedge A = \text{false}$$

$$\neg A \vee A = \text{true}$$

- Idempotenz

$$A \wedge A = A$$

$$A \vee A = A$$

- Kommutativität

$$A \wedge B = B \wedge A$$

$$A \vee B = B \vee A$$

- Assoziativität

$$(A \wedge B) \wedge C = A \wedge (B \wedge C)$$

$$(A \vee B) \vee C = A \vee (B \vee C)$$

- Absorption

$$A \wedge (A \vee B) = A$$

$$A \vee (A \wedge B) = A$$

- Distributivität

$$A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$$

$$A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$$

Aussagenlogik

- Wahrheitstafel für $A \wedge (A \vee B) = A$

A	B	$(A \vee B)$	$A \wedge (A \vee B)$	A
t	t	t	t	t
t	f	t	t	t
f	t	t	f	f
f	f	f	f	f



Wahrheitstafel

- $(A \wedge B) \wedge C = A \wedge (B \wedge C)$

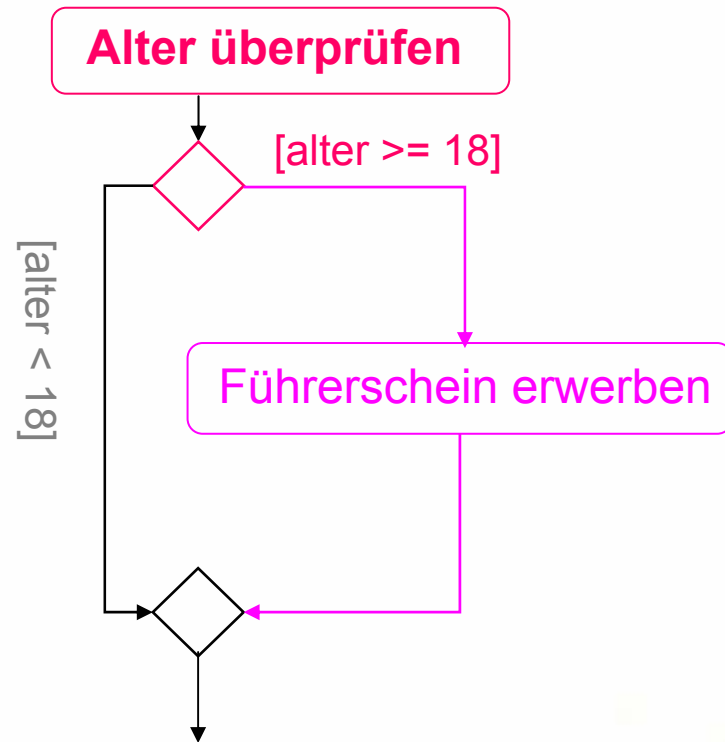


Inhalt

- Klassen, main-Methode
- Lokale Variablen, Zuweisungen
- Arithmetische Ausdrücke
- Primitive Datentypen
- Aussagenlogik
- **Kontrollanweisung: if-else**
- Boolesche Ausdrücke
- Methoden

if-else

- Bisher nur sequentielle Programmausführung
- Realität: Fallunterscheidungen
 - Ein Person gilt als volljährig, wenn sie mindestens 18 Jahre alt ist.
 - Wenn eine Person volljährig ist, darf sie einen Führerschein erwerben; andernfalls darf sie kein Kraftfahrzeug erwerben
- Kontrollanweisungen
 - Steuern Programmablauf (Verzweigungen, Wiederholungen)



```

public static void main(String argv[]) {
    int alter = 16;
    if (alter >= 18)
        System.out.println("Person ist volljährig");
}
  
```

if-else

```
if (alter >= 18)
    System.out.println("Führerschein erwerben");
```

■ Syntax

- Schlüsselwort **if** gefolgt von einem **Booleschen Ausdruck** (gleich) in runden Klammer
- Ein einzelne Anweisung oder ein Block mit mehreren Anweisungen
- **Block**: Mehrere Anweisungen die mit { .. } zusammengruppiert werden

```
if (alter >= 18) {
    System.out.println("Person ist volljährig und");
    System.out.println("darf einen Führerschein erwerben");
}
```

■ Konvention

- Anweisung(en) 2 bis 4 Zeichen nach links einrücken
- **Immer** geschweifte Klammern verwenden

```
if (alter >= 18) {
    System.out.println("Führerschein erwerben");
}
```

if-else

```
if (alter >= 18) {  
    System.out.println("Person ist volljährig und");  
    System.out.println("darf einen Führerschein erwerben");  
}  
System.out.println("Dies wird immer ausgegeben");
```

■ Semantik

- Wenn bei Ausführung der if-Anweisung der **Ausdruck** wahr ist, dann wird **der zum if gehörige Anweisungsblock** ausgeführt
- Danach wird das Programm sequentiell fortgeführt (**Anweisungen nach dem if**)
- Ausgabe „Person ist volljährig“
 - Boolesche Ausdruck `alter >= 18` ist wahr
 - Anweisung nach if wird ausgeführt

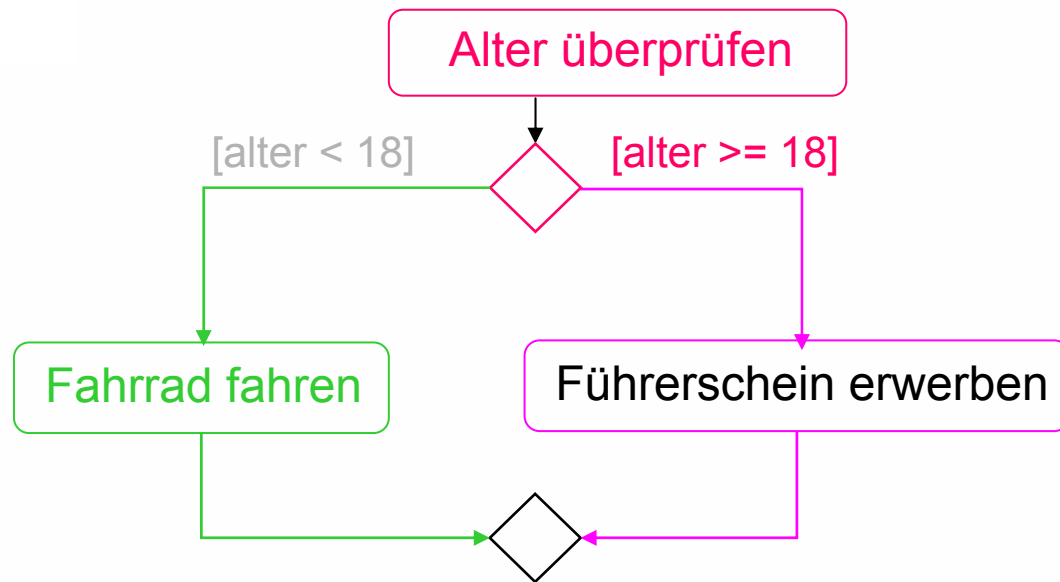
■ Ausgabe dieses Programmfragments:

Dies wird immer ausgegeben

- da `alter` nicht größer, gleich 18 ist

if-else

- Fallunterscheidung mit **alternativen** Ausführung
- Zusätzliches Schlüsselwort **else**
- Keine zusätzliche Bedingung für die Alternative



```
if (alter >= 18) {  
    System.out.println("Führerschein erwerben");  
} else {  
    System.out.println("Fahrrad fahren");  
}
```

if-else

■ Semantik

- Bedingung wird ausgewertet
- Falls sie wahr ist, dann werden zu if-gehörige Anweisungen ausgeführt
- Falls sie falsch ist, dann werden zu else-gehörige Anweisungen ausgeführt

```
if (alter >= 18) {  
    System.out.println("Führerschein erwerben");  
} else {  
    System.out.println("Fahrrad fahren");  
}
```



Inhalt

- Klassen, main-Methode
- Lokale Variablen, Zuweisungen
- Arithmetische Ausdrücke
- Primitive Datentypen
- Aussagenlogik
- Kontrollanweisung: if-else
- **Boolesche Ausdrücke**
- Methoden



Boolesche Ausdrücke

- Boolescher Ausdruck
 - Ausdruck, der zwei Werte `true` oder `false` annimmt
 - Datentyp `boolean`
- Binäre Vergleichsoperatoren: `==`, `!=`, `>`, `<`, `>=`, `<=`
- Führen zu elementaren Vergleichsaussagen

```
alter >= 18 // true, wenn alter Wert grösser, gleich 18 hat
```

```
alter == 18 // true, wenn alter Wert gleich 18 ist
```

```
alter != 18 // true, wenn alter Wert ungleich 18 ist
```

```
(a + b) * (a + b) == a * a + 2 * a * b + b * b // immer true
```



Boolesche Ausdrücke

Beispiel	Ergebnis	Kommentar
$1 \neq 1$	false	
$42.0 \neq 29345.2934$	true	Beide Zahlen haben (im Speicher) unterschiedliche Repräsentation
$1 + 7 \neq 7$	true	8 ist ungleich 7
$1 < 1$	false	
$18.0 \geq 18.000001$ $18.0 \geq 18.0$	false true	
$1 + 7 > 7$	true	8 ist größer 7



Boolesche Ausdrücke

■ Boolesche Operatoren

- Und (Konjunktion) : &
- Oder (Disjunktion): |
- Exklusives Oder: ^
- Negation: !

```
boolean b = true;
```

```
b & false // immer false
```

```
b | false // true, wenn b true ist
```

```
alter <= 16 & alkoholisiert // sollte besser nie true werden
```

Java / Boolesche Bedingungen

Beispiel	Ergebnis	Kommentar
<code>!(1 == 1)</code>	false	1 == 1 ist true, Negation (!) von true ist false
<code>int alter = 21;</code> <code>boolean fahrpruefungBestanden = true;</code> <code>(alter >= 18) & fahrpruefungBestanden</code>	true	alter > 18 ist true true & true ergibt true
<code>int a = ...</code> <code>(a > 0) & (a < 0)</code>	false	a kann nie zugleich negativ und positiv sein
<code>int a = ...</code> <code>(a == 0) (a > 0) (a < 0)</code>	true	a muss entweder 0, negativ oder positiv sein

Java / Bedingungen, Ausdrücke

- Bedingungen und Ausdrücke werden von links nach rechts (Links-Assoziativ) unter Berücksichtigung des *Vorrangs* beteiligter binärer Operatoren ausgewertet
- Arithmetische Operatoren
 - *, /, % haben gleichen Vorrang
 - +, - haben gleichen Vorrang
 - *, /, % vor +, - (Punkt vor Strichrechnung)
- Boolesche Operatoren
 - & vor ^, ^ vor | (und natürlich & vor |)
- Vergleichsoperatoren
 - ==, != gleichen Vorrang (Vergleichsoperatoren)
 - >=, >, <=, < gleichen Vorrang (Relationaloperatoren)
 - Relationaloperatoren vor Vergleichsoperatoren



Java / Bedingungen, Ausdrücke

Vorrang Boolesche Ausdrücke

$a \& b \& c$ entspricht $(a \& b) \& c$

$a | b \wedge c$ entspricht $a | (b \wedge c)$

$a | b \& c$ entspricht $a | (b \& c)$

$a | b \& c \wedge d$ entspricht $a | ((b \& c) \wedge d)$



Java / Bedingungen, Ausdrücke

- Vorrang Vergleichsoperatoren

int a,b;

a == b == c entspricht (a == b) == c

macht normalerweise keine Sinn

a == b **hat Typ boolean**, c muss also auch boolean sein

- **int** a,b,c,d;

a == b == c == d ergibt Compilerfehler

a == b == (c == d) ist OK



Java / Bedingungen, Ausdrücke

- Boolesche Operatoren `&&`, `||`
 - Semantik wie `&`, `|`, aber
 - Auswertung von link nach rechts bricht ab, wenn linkes Teilergebnis Gesamtergebnis determiniert

`false && A`, A wird nicht mehr ausgewertet

`true && A`, A wird noch ausgewertet

`true || A`, A wird nicht mehr ausgewertet
- `^^` gibt es nicht
 - `A ^ B` ist true, wenn A false und B true oder A true und B false: rechte Seite muss immer ausgewertet werden



Java / Bedingungen, Ausdrücke

- Boolesche Operatoren &&, ||
 - Semantik wie &, |, aber
 - Auswertung von link nach rechts bricht ab, wenn linkes Teilergebnis Gesamtergebnis determiniert
 - false && A, A wird nicht mehr ausgewertet
 - true && A, A wird noch ausgewertet
 - true || A, A wird nicht mehr ausgewertet
- Zuweisung (a = 5) ist Ausdruck mit Wert der rechten Seite von „=“, also 5

Seiteneffekt: Änderung von Werten in Ausdrücken

```
int a = 7;
(a == 4) && ( (a=5)
== 5)
// a hat Wert 7
```

```
int a = 7;
(a == 4) &( (a=5) ==
5)
// a hat Wert 5
```



Java / Bedingungen, Ausdrücke

- Wann & verwenden und wann &&?
 - & (analog |):
 - wenn jeder Teilterm ausgewertet werden muss, z.B. bei gewollten Seiteneffekten
 - wenn Ausdruck rechts von & (oder |) Fehler haben könnten; bei && wird er beim Testen vielleicht nie ausgeführt
 - && (analog ||):
 - sonst, da „schneller“; immer dann, wenn Ausdruck Seiteneffekt-frei ist.
- Seiteneffekte vermeiden



Inhalt

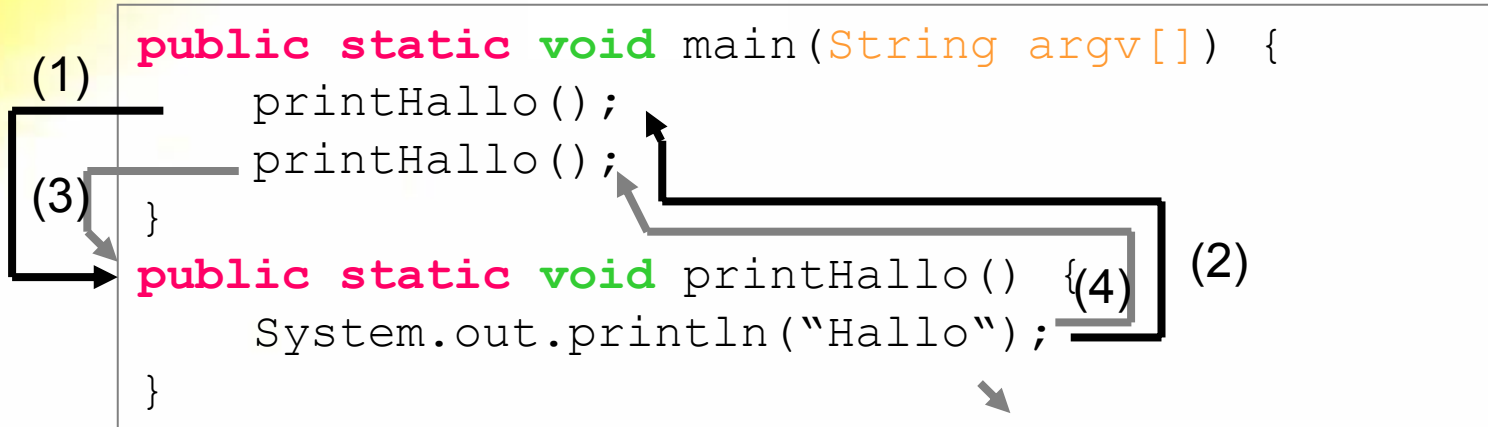
- Klassen, main-Methode
- Lokale Variablen, Zuweisungen
- Arithmetische Ausdrücke
- Primitive Datentypen
- Boolesche Ausdrücke, if-else
- **Methoden**



Methoden

- Bis jetzt
 - Programm besteht aus einer Klasse und einer (main) Methode
 - Einfache Berechnungen mit Zahlen
 - Einfacher Programmablauf mit Verzeigungen
- Berechnungen nur an einem Ort
 - Umwandlung km/h in Meilen/h nicht wiederholt an verschiedenen Stellen implementieren
 - Berechnungen zusammenfassen in einer Methode (Funktion, Prozedur)
- Analog Kochrezept
 - Rezept für Mürbeteig
 - Rezept für Obstkuchen wiederholt nicht Rezept für Teig, sondern verweist nur darauf

Methoden



- Methode beinhaltet (Teil)programm
- Teilprogramm kann andere Methoden *aufrufen*
 - *Programmablauf verzweigt zum Programm der Methode (1, 3)*
 - *Nach Beendigung Methode kehrt Programmablauf zurück zum ursprünglichen Methodenaufruf (2, 4) und führt folgende Anweisungen aus*

Methoden

```
public static void main(String argv[]) {
    printHallo();
    AndereKlasse.printHallo();
}
public static void printHallo() {
    System.out.println("Hallo");
}
```

- *printHallo()*
 - Methode „*printHallo()*“ in der **gleichen** Klasse wird aufgerufen
 - Methode kann *public*, *private*, *protected* sein (Bedeutung später)
 - Compilerfehler, falls keine derartige Methode existiert
- *MeineKlasse.printHallo()*
 - Methode *printHallo()* in der Klasse „*AndereKlasse*“ wird aufgerufen
 - Methode muss *static* und *public* sein (*protected* später in Info 2)
 - *private*: Methode kann nicht von anderer Klasse aufgerufen werden



Methoden

```
public class AndereKlasse {  
    public static void printHello() {  
        System.out.print("Hello");  
    }  
}
```

(1)

(2)

```
public class HelloWorld {  
    public static void main(String argv[ ]) {  
        AndereKlasse.printHello();  
        printWorld();  
    }  
}
```

(3)

```
public static void printWorld() {  
    System.out.println(" World");  
}
```

(4)



Methoden

```
public class AndereKlasse {  
    public static void printHello() {  
        System.out.print("Hello");  
    }  
}
```

```
public class HelloWorld {  
    public static void main(String argv[ ]) {  
        AndereKlasse.printHello();  
        printWorld();  
    }  
  
    public static void printWorld() {  
        System.out.println(" World");  
    }  
}
```



Methoden

```
public static void main(String argv[]) {  
    print("Hello");  
    print(" World");  
}  
  
public static void print(String str) {  
    System.out.println(str);  
}
```

- Methoden können bei Aufruf *Werte* als *Parameter* übergeben werden
- Zugriff auf Werte innerhalb Methode über *Parametername* (wie lokale Variable)
- Konvention Parametername wie lokale Variablen
- Parametername und lokale Variablen müssen für Methode eindeutige Namen haben
- Typ: primitiver Datentyp oder Klasse (später)



Methoden

```
public static void main(String argv[]) {
    print("Hello");
    print(" World");
}

public static void print(String str) {
    System.out.println(str);
}
```

- Bei Aufruf print("Hello")
 - Programm verzweigt zu Methode „print“
 - Ähnlich lokalen Variablen wird für jeden Parameter Speicherplatz reserviert
 - Wert "Hello" wird Parameter „str“ zugewiesen (d.h. in für „str“ reservierten Speicherplatz kopiert)
 - Anweisungen der Methoden im Rumpf werden ausgeführt
- Übergebene Wert muss „gleichen“ Typ wie in Deklaration des Parameter besitzen
- Übergebene Wert kann auch ein komplexer Ausdruck sein:
 - print("H" + "e" + "l" + "l" + "o");

Methoden

```
public static void main(String argv[]) {  
    double kmh = 120.0;  
(1) double mph = berechneMph(kmh);  
    System.out.println(kmh + " sind " + mph + " mph");  
}  
  
public static double berechneMph(double kmh) {  
    double mph = kmh * 0.621;  
    return mph;  
} (2)
```

- Methoden können einen berechnete Wert an aufrufende Stelle zurückgeben (2) (Funktionen)
 - Rückgabetyt des Wertes bei Methode deklarieren
 - Mit return-Anweisung Wert zurückgeben
 - Funktionen können in Ausdrücken verwendet werden (s.o. bei Initialisierung von mph)



Methoden

```
public static void main(String argv[]) {  
    double kmh = 120.0;  
    System.out.println(kmh + " sind " +  
                        (2) berechneMph(kmh) + " mph");  
    kmh = 324.12;  
    System.out.println(kmh + " sind " +  
                        berechneMph(kmh) + " mph");  
}  
  
public static double berechneMph(double kmh) {  
    return kmh * 0.621;(1)  
}
```

- Wert bei return-Anweisung kann komplexer Ausdruck sein (1)
- Funktionen (Methoden mit Rückgabewert) können in Ausdrücken verwendet werden (2)

Methoden

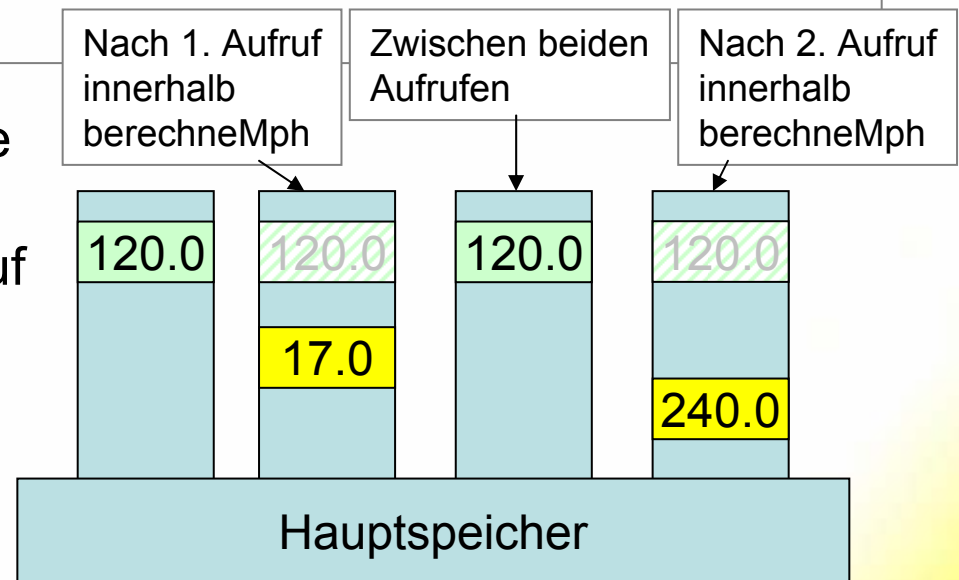
```

public static void main(String argv[]) {
    double kmh = 120.0;
    double mph1 = berechneMph(17.0); // 1. Aufruf
    // zwischen beiden Aufrufen
    double mph2 = berechneMph(kmh * 2.0); // 2. Aufruf
}

public static double berechneMph(double kmh) {
    return kmh * 0.621;
}

```

- Parameter verhalten sich wie lokale Variablen
- Initialer Wert ist der bei Aufruf übergebene Wert
- Lokale Variablen vorheriger Aufrufe nicht mehr sichtbar (lokal)



Methoden

```

public static void main(String argv[]) {
    double x = 10.0;
    double y = 5.5;
    System.out.println(x + " - " + y
                       + " = " + subtrahieren(x, y) );
}

public static double subtrahieren(double a, double b) {
    return a - b;
}

```

Interne Konvertierungsregeln zu einer Zeichenkette

- Methoden können mehrere Parameter besitzen
 - Reihenfolge Parameter bei Aufruf beachten
- Konventionen
 - Nicht zu viele Parameter definieren (höchstens sechs bis sieben)
 - Methodenname soll ausdrücken, was Methode tut (aber nicht wie es das tut)
 - Verben als Methodennamen, klein schreiben, erste Buchstabe Teilwörter groß
printHelloWorld(); kontoEroeffnen(); immatrikulieren();

Methoden

```
public static double subtrahieren(double a, double b) {  
    return a - b;  
}  
public static int subtrahieren(int a, int b) {  
    return a - b;  
}  
public static double subtrahieren(int a, int b) {  
    return a - b;  
}
```

- Methoden mit identischen Namen aber unterschiedlichen Parametern (Typen) sind verschieden (Polymorphie)
- Methoden mit unterschiedlichen Rückgabetypen und identischen Parametern (Typen) müssen unterschiedliche Namen besitzen
 - Eine der beiden letzten Methoden muss entfernt werden (Compilerfehler)
- Compiler sucht passende Methode bei aufrufender Stelle
 - double e1 = subtrahieren(10.0, 7.0); // erste, da double Parameter
 - double e2 = subtrahieren(10.0, 7); // Fehler, keine passende Methode
 - int e2 = subtrahieren(8, 9); // zweite, da int Parameter

Methoden in Klasse Math

static int	<u>max</u> (int a, int b) Returns the greater of two int values.	static double	<u>abs</u> (double a) Returns the absolute value of a double value.
static long	<u>max</u> (long a, long b) Returns the greater of two long values.	static float	<u>abs</u> (float a) Returns the absolute value of a float value.
static double	<u>min</u> (double a, double b) Returns the smaller of two double values.	static int	<u>abs</u> (int a) Returns the absolute value of an int value.
static float	<u>min</u> (float a, float b) Returns the smaller of two float values.	static long	<u>abs</u> (long a) Returns the absolute value of a long value.
static int	<u>min</u> (int a, int b) Returns the smaller of two int values.		
static long	<u>min</u> (long a, long b) Returns the smaller of two long values.		
static double	<u>pow</u> (double a, double b) Returns the value of the first argument raised to the power of the second argument.		
static double	<u>random</u> () Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.		
static long	<u>round</u> (double a) Returns the closest long to the argument.		
static int	<u>round</u> (float a) Returns the closest int to the argument.		

- cos, sin, log, exp, pow, tan, ...



Methoden

- Guthaben eines Konto manipulieren
 - Betrag abheben und neues Guthaben zurückgeben; falls Betrag negativ wird, dann soll altes Guthaben zurückgegeben werden
 - Betrag einzahlen
 - Keine Bildschirmausgaben
- Zweite Klasse TestKonto
 - Methoden aufrufen
 - Bildschirmausgaben



Klassenvariable

- Bisher
 - Lokale Variablen sind nur innerhalb einer Methode gültig und sichtbar
 - Vor und nach Aufruf existieren sie nicht
- Klassenvariablen
 - Existieren pro Klasse
 - Name pro Klasse eindeutig
 - Deklaration innerhalb Klasse analog lokale Variablen
 - Modifier wie Methoden (**static**, **public**)
 - Initialisierung genau einmal, wenn Klasse zum ersten mal verwendet wird (genauer: wenn sie geladen wird)



Klassenvariable

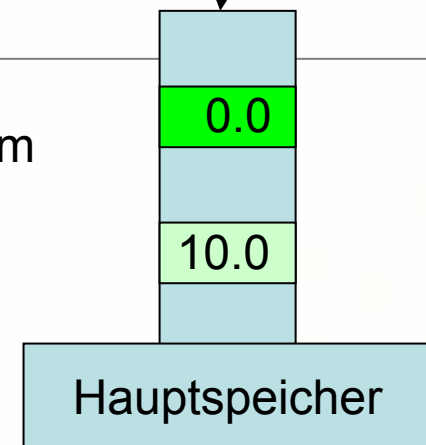
```
public class Konto {  
  
    public static double guthaben = 0.0;  
  
    public static void betragAbheben(double betrag) {  
        if (betrag <= Konto.guthaben) {  
            Konto.guthaben = Konto.guthaben - betrag;  
        }  
    }  
  
    public static void betragEinzahlen(double betrag) {  
        Konto.guthaben = Konto.guthaben + betrag;  
    }  
}
```

Klassenvariable

```

public class Konto {
    public static double guthaben = 0.0;
    public static void betragAbheben(double betrag) {
        double guthaben = 10.0;
        if (betrag <= Konto.guthaben) {
            Konto.guthaben = guthaben - betrag;
        }
    }
    public static void betragEinzahlen(double betrag) {
        Konto.guthaben = guthaben + betrag;
    }
}

```



- Lokale Variable gleichen Namens möglich: Zugriff auf Klassenvariablen mit vorangestelltem Klassennamen
- Lokale Variable **verdeckt** Klassenvariable gleichen Namens (falls diese nicht mit Klassenname eindeutig referenziert wird)
- Konvention: Klassennamen immer davor schreiben (wie bei Methoden)



Kommentare

- Einzeilige Kommentare
`// Dies ist ein Kommentar (bis zum Ende der Zeile)`
- Mehrzeilige Kommentare
`/* Dieser Kommentare
erstreckt sich über
vier Zeilen
*/`
- Dürfen an beliebiger Stelle im Quelltext vorkommen
- Werden vom Compiler ignoriert
- Dienen als zusätzliche Information für menschlichen Leser
- Mehr zum Kommentieren von Programmen später