

VO Einführung in die Programmierung Teil I

WS 2017/18

H.Hagenauer
FB Computerwissenschaften

Übersicht zur LV

Ziel Erstellung einfacher Programme und
Erlernung von algorithmischem Denken
mittels Programmiersprache Java

dazu wird benötigt

- ein Compiler
- ein Editor (kein Textverarbeitungssystem!)
- oder alternativ
- eine einfache Entwicklungsumgebung

Compiler, Editor

Compiler

Java Development Kit – JDK von Oracle (früher Sun Microsystems)

Editor

im Prinzip jeder schon am Rechner vorhandene Editor möglich -
jedoch etwas komfortabler z.B. *Notepad++* (für Windows, besitzt
u.A. Syntaxhighlighting)

Entwicklungsumgebung

z.B. *jGRASP*

Eclipse, NetBeans, ... (Bsp. für integrierte Entwicklungs-
umgebungen – IDE) nicht empfehlenswert für Einstieg in die
Programmierung.

Literaturhinweise und Webseiten

Bücher

- W.Savitch: *Java – An Introduction to Problem Solving & Programming*, 7th edition, Pearson, 2014
- H.Mössenböck: *Sprechen Sie Java?*, 5.Auflage, dpunkt.verlag, 2014
- ... weitere umfangreiche Auswahl

Webseiten

- <http://www.oracle.com/technetwork/java/index.html>
viele zu Java vom Softwarehersteller Oracle (u.a. JDK)
- <http://docs.oracle.com/javase/8/docs/api/index.html>
API-Spezifikation (Application Programming Interface)
- <http://docs.oracle.com/javase/tutorial/>
Tutorien zu verschiedenen Themen rund um Java
- ... große Auswahl mit verschiedenen Schwerpunkten (Tutorien, Tipps, Regelwerk, Fragen, ...)

Kapitel 1

Erstes Programm

Erstes Programm

Berechnung von
Umfang und
Fläche eines
Kreises

Eingabe: Radius

```
public class Kreis{  
  
    public static void main(String[] args){  
  
        double radius, umfang, flaeche;  
        final double PI = 3.14159;  
  
        System.out.println();  
        System.out.println("Radius bitte: ");  
        radius = SavitchIn.readLineDouble();  
  
        umfang = 2*PI*radius;  
        flaeche = PI*radius*radius;  
  
        System.out.println("Umfang  = " + umfang);  
        System.out.println("Flaeche = " + flaeche);  
        System.out.println();  
    }  
}
```

Schritte zum lauffähigen Programm

1. (Programm)Code mittels Editor eingeben
2. Programm speichern als Datei mit Namen `Kreis.java` (z.B. im Verzeichnis `einfprog`)

```
C:\> cd einfprog
```

3. ins Verzeichnis `einfprog` wechseln

4. Datei `Kreis.class` erzeugen
(Hinweis: Datei `SavitchIn.java` oder `SavitchIn.class` muss im selben Verzeichnis vorhanden sein!)

```
C:\einfprog> javac Kreis.java
```

5. Programm ausführen („laufen lassen“)

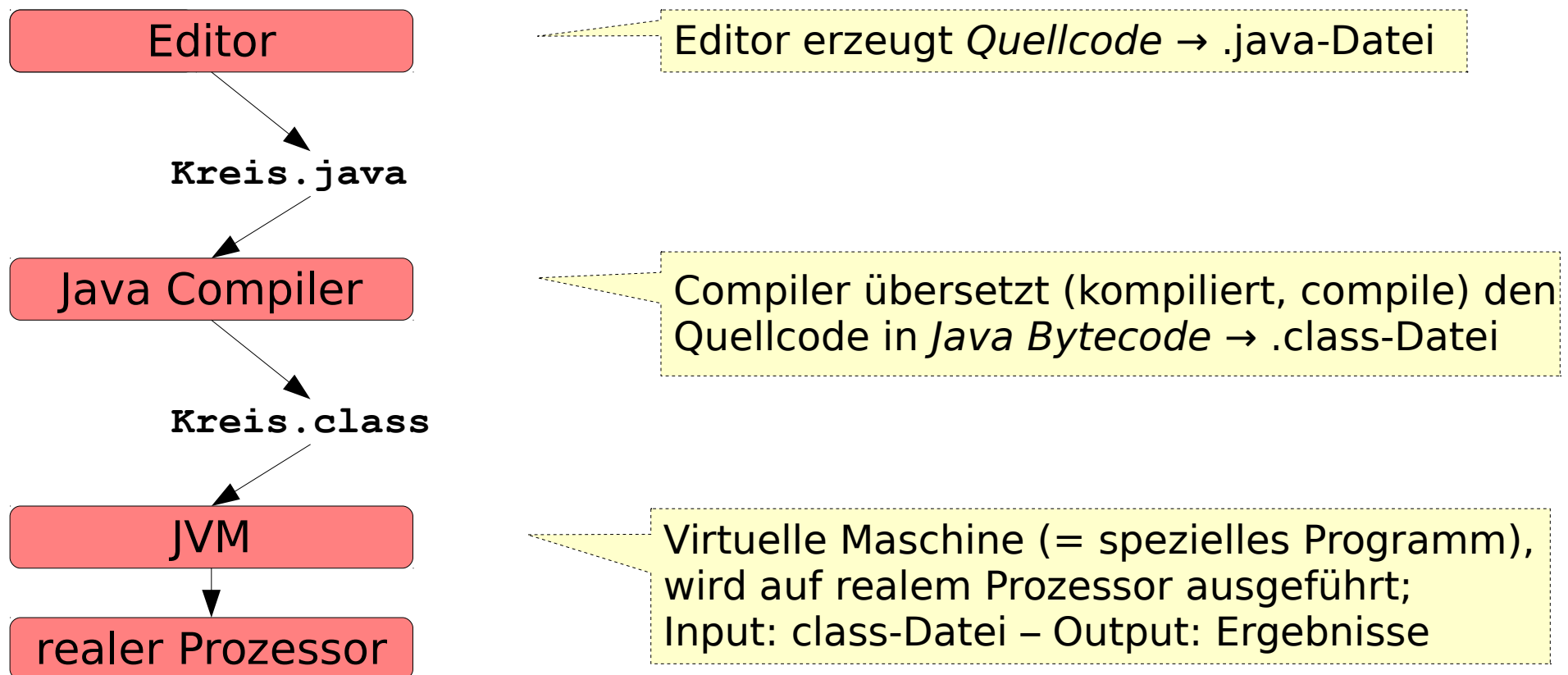
```
C:\einfprog> java Kreis
```

(Hinweis: Eingaben mit return-Taste abschließen)

Compiler – Java Bytecode

Compiler: spezielles Programm zur Übersetzung von Quellcode in Bytecode

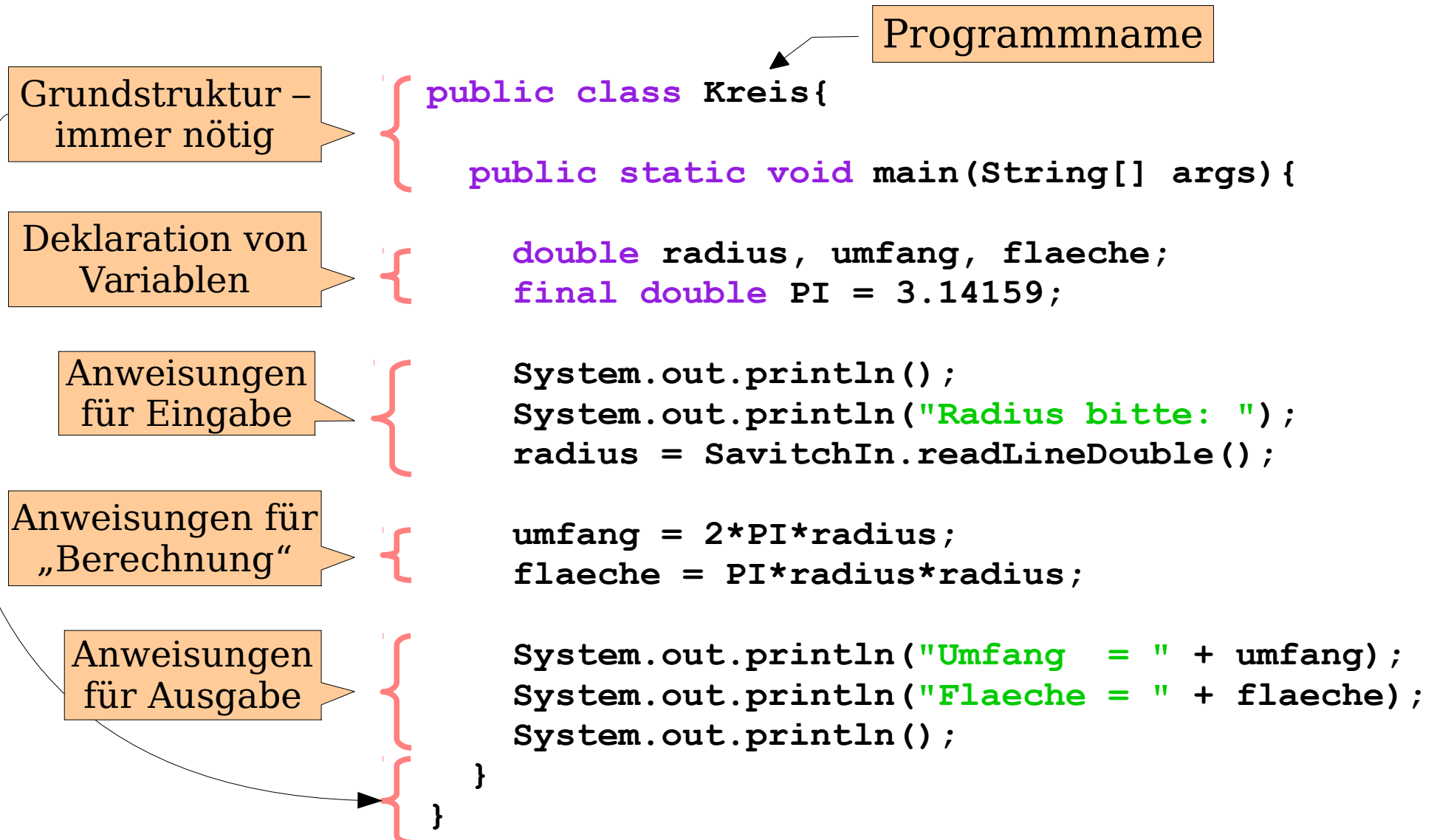
Bytecode: *Maschinencode* (d.h. vom Prozessor lesbarer Code) für einen „virtuellen“ Prozessor (Maschine) – *Java Virtual Maschine (JVM)*



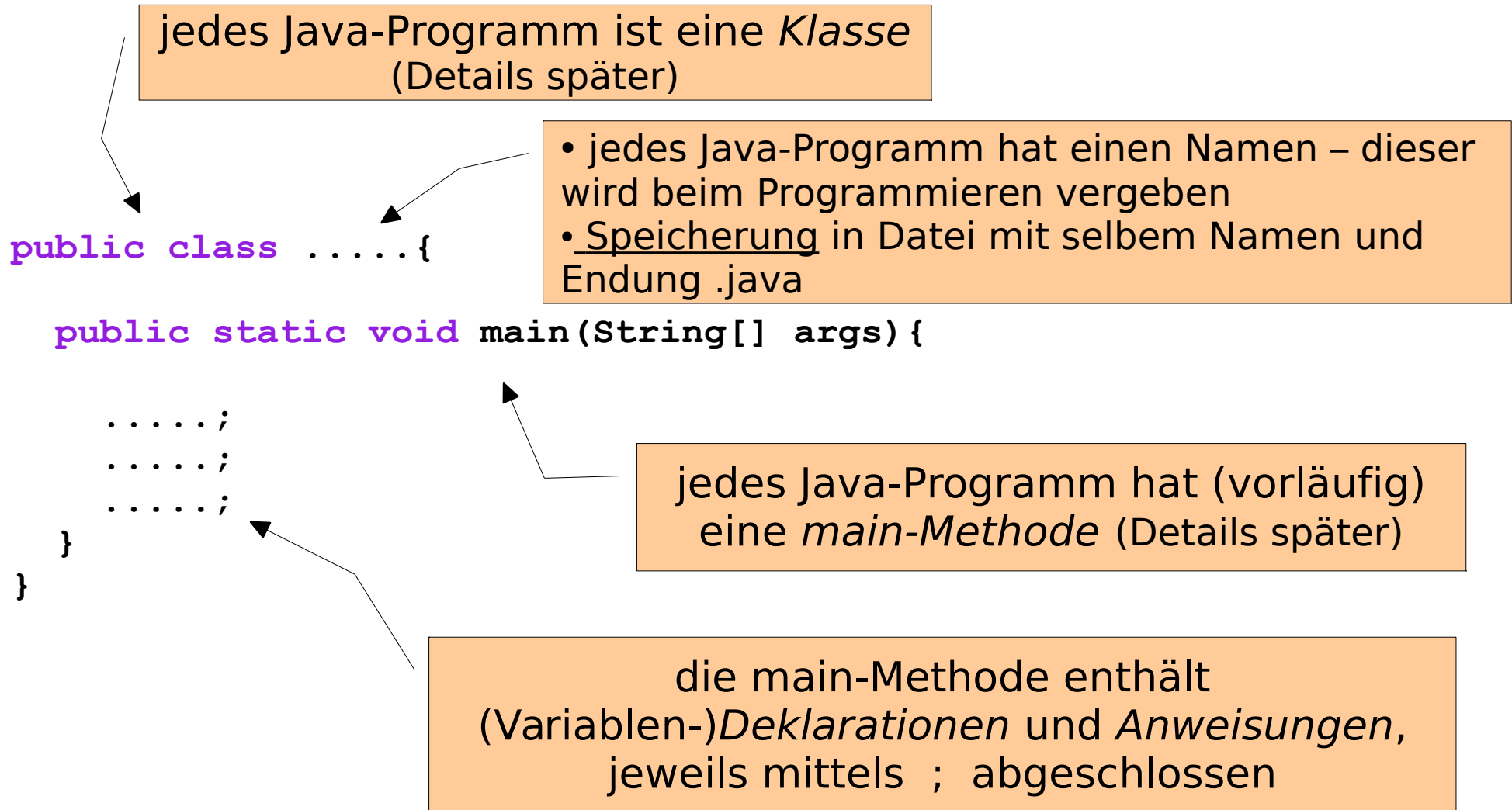
Programmiersprache Java

- Entwicklung ab 1991 bei Sun Microsystems (unter James Gosling) für Waschmaschinen, TV-Geräte, ...
- etwa 1994: Java als Programmiersprache für Webbrowser
- Java ist eine *general purpose* Programmiersprache – d.h. für allgemeine Anwendungen geeignet
- Java ist eine *high-level language* - d.h. für Menschen (relativ) leicht verständlich und schreibbar;
weitere Bsp.: C, C++, Visual Basic, Ada, Pascal, ...;
dazu im Gegensatz: Maschinensprachen – einfache Befehle, werden direkt vom Prozessor ausgeführt, für Menschen kaum lesbar
- Java ist eine *objektorientierte* Programmiersprache (*object oriented programming, OO-Programmierung*) – d.h. Software wird in verschiedene Teile mit spezifischen Aufgaben gegliedert (Details dazu später)

Erstes Programm – genauer betrachtet



Grundstruktur eines Java-Programms



- alle-Teile sind beim Programmieren entsprechend einzusetzen.
- Blöcke werden in Java mittels {...} gekennzeichnet.

Verwendete Symbole

Java-Programme bestehen aus

Buchstaben, Ziffern, Satzzeichen, mathematischen Symbolen, Leerzeichen (blanks), Sonderzeichen

Daraus werden gebildet

Namen (Bezeichner, identifier)

- bezeichnen z.B. Programmname, Variablen, Konstanten, Typen, Methoden, ...
- bestehen aus: Buchstaben, Ziffern, `_`, `$`;
erstes Zeichen darf keine Ziffer sein;
Groß- und Kleinbuchstaben werden unterschieden!

z.B.:

`Kreis` `radius` `x` `meinProg` `min` `Min` `m2n` `all_Numbers`

nicht erlaubt z.B.:

`5ter` `my prog` `sbg-wien` `private`

Verwendete Symbole (Forts.)

Schlüsselwörter – reservierte Wörter

- sind spezielle Bezeichner, die Programmteile kennzeichnen, in Java fix definiert;
dürfen nicht als eigene Namen verwendet werden!

z.B.: `public class static void if while private`
(Hinweis: komplette Liste siehe Literatur)

Zahlen (dezimal)

- ganze Zahlen, z.B.: `5 815 1000000`
- Gleitkommazahlen, z.B. `3.14154 0.35 2.5`
- Hexadezimal-, Binärzahlen (siehe Literatur)

Verwendete Symbole (Forts.)

Kommentare

- fügen Erläuterungen im Programmcode hinzu
- werden vom Compiler ignoriert → keine Auswirkungen auf Programmablauf
- 2 Arten von Kommentaren (in Java)

Zeilenkommentar

- beginnt mit //
- reicht bis zum Ende der Zeile

```
double sum; //alle Ausgaben
```

Blockkommentar

- begrenzt durch `/* ... */`
- kann sich über mehrere Zeilen erstrecken

```
/*  
Programm berechnet Umfang,  
Flaeche eines Kreises.  
Eingabe: Radius  
*/
```

Kapitel 2

Einfache Programme: Variablen, Grundtypen, Algorithmus

Variablen und deren Deklaration

- Variablen dienen zur Speicherung von Daten (z.B. Eingabewerte, (Zwischen-)Ergebnisse, ...)
- Variablen sind „Behälter“ mit Namen (Bezeichner) und sie besitzen einen bestimmten *Typ*
- Variablen müssen vor ihrer ersten Verwendung *deklariert* werden, um
 - Typ und Name festzulegen
 - Speicherplatz zu reservieren
- Variable können immer nur *einen* Wert des vereinbarten Typs speichern
- Werte von Variablen sind änderbar (mittels *Zuweisungen*)

Variablen und deren Deklaration (Forts.)

- der Typ einer Variablen bestimmt welche Werte gespeichert werden können (keine Addition von Äpfel und Birnen!)
 - Compiler prüft dies – findet solche Fehler
 - erforderlicher Speicherplatz wird bereitgestellt, Codierung festgelegt

```
double radius;      // Variable radius vom Typ double
int k, n;           // Variablen k und n vom Typ int
long grosseZahl;    // Variable grosseZahl vom Typ long
```

Variablendeklaration (vorerst):

allgemeine Form

type variable_1, variable_2, ...;

beliebiger
primitiver Typ

beliebige Anzahl von
Variablenbezeichnern

Primitive Typen von Java (vorläufig)

Allgemein bestimmt ein (*Daten-*)*Typ* welche Werte eine Variable speichern kann – Wertebereich wird festgelegt

Java unterscheidet 2 Arten von Typen

- *primitive Typen*: einfache Werte wie Zahlen, Zeichen, ...
- *Klassentypen* (siehe später)

Folgende *primitive Typen* (*Grundtypen, primitive types*) werden vorerst verwendet (weitere später):

Typ	Speicher- bedarf	Wertebereich
int	4 Byte	ganze Zahlen im Bereich $-2^{31} \dots 2^{31}-1$
long	8 Byte	ganze Zahlen im Bereich $-2^{63} \dots 2^{63}-1$
double	8 Byte	Gleitkommazahlen

Zahltypen, Numerale

Unveränderliche Zahlwerte werden mittels *Numeralen* direkt im Programm angegeben.

Numerale für ganze Zahlen

- Folge von Ziffern ohne Blanks dazwischen
- optional kann ein Vorzeichen vorangestellt werden: - oder + (bei keinem Vorzeichen wird + angenommen)

```
1  
4711  
0  
-25343  
2879345  
+55
```

Zahltypen, Numerale (Forts.)

Numerale für Gleitkommazahlen

- Folge von Ziffern mit Dezimalpunkt (verpflichtend!), welcher den Nachkommaanteil kennzeichnet; Vorzeichen analog zu ganzen Zahlen
- alternative wissenschaftliche Notation (scientific notation): E oder e zur Kennzeichnung eines Exponenten zur Basis 10

$2e3$ bedeutet $2 \cdot 10^3$

Exponenten sind ganze Zahlen,
optional mit Vorzeichen

Beachte: 18 und 18.0 sind mathematisch gleiche Werte – jedoch verschiedene Typen!

```
1.0 //Dezimalpkt. nötig!  
0.001  
-25.47  
25000.0
```

```
1E0  
1e-3  
-2.547e1  
2.5E4
```

Arithmetische Ausdrücke und Operatoren

Arithmetische Ausdrücke (expressions) berechnen mit Hilfe von Operatoren einen numerischen Wert aus Variablen und Konstanten

- binäre (zweistellige) Operatoren für Grundrechnungsarten:
+ - * / (Addition, Subtraktion, Multiplikation, Division)
 - es gelten die bekannten Vorrangregeln der Mathematik (d.h. Punktrechnung (*, /) vor Strichrechnung (+, -) bzw. anders ausgedrückt: *, / haben höhere Priorität als +, -)
 - Klammern (nur runde ()) heben die Vorrangregeln auf
 - gleichrangige Operatoren werden von links nach rechts ausgewertet (*links-assoziativ*)

```
3 + 5 * 2           // Erg: 13
(3 + 5) * 2         // Erg: 16
3 + (5 * 2)         // Erg: 13
3 - 5 + 2 - 4       // Erg: -4
3 - (5 + (2 - 4))   // Erg: 0
```

siehe Programm:
ArithmAusdruck

Arithmetische Ausdrücke und Operatoren (Forts.)

- unäre (einstellige) Operatoren für Vorzeichen: $+$ $-$
 - beziehen sich auf nur einen Operanden
 - binden stärker (höhere Priorität) als obige binäre Operatoren
- ganzzahlige Division
 - gilt dann, wenn beide Operanden integer-Typen sind
 - Ergebnis ist ganzzahlig (ganzzahliger Anteil des Quotienten)
 - Nachkommaanteil wird abgeschnitten – kein runden!

```
5 + -2    // Erg: 3
-5 + 2    // Erg: -3
-(5 + 2)  // Erg: -7
```

```
5 / 3      // Erg: 1
23 / 5     // Erg: 4
5 / 10     // Erg: 0
-7 / 3     // Erg: -2
```

Arithmetische Ausdrücke und Operatoren (Forts.)

- Modulus- oder %-Operator: %
 - berechnet den Rest bei Division
 - binärer Operator, vor allem für integer-Typen
 - selbe Priorität wie Multiplikation und Division
 - Zusammenhang mit ganzzahliger Division:

$$a \% b = a - (a/b) * b$$

```
5 % 3          // Erg: 2
12 % 2         // Erg: 0
-5 % 3         // Erg: -2
7 - 15 % 4     // Erg: 4
```

Zuweisungen (assignments)

Zuweisungen speichern einen (berechneten) Wert in einer Variablen

allgemeine Form

variable = expression;

zu berechnender
Ausdruck

Zuweisungsoperator

Vorgangsweise:

1. Ausdruck auf rechter Seite wird ausgewertet
2. Wert wird der Variablen auf der linken Seite zugewiesen

Dabei ist zu beachten

- bisheriger Wert der Variablen wird überschrieben
- dies ist *keine* mathematische Gleichung!

```
umfang = 2 * PI * radius;  
int n; double x;  
n = 11;  
n = n + 1;  
x = 15.1;  
n = n*n + 5;  
    //n: 149  
x = x * 1.8 + 32.0  
    //x: 59.18
```


Zuweisungen - Zuweisungskompatibilität

Zuweisungen sind nur dann möglich, wenn die Typen auf beiden Seiten einer Zuweisung die *Zuweisungskompatibilität* (*assignment compatibility*) erfüllen:

- Typ der linken und rechten Seite ist gleich oder
- (bei primitiven Typen) Typ der rechten Seite kann einer Variablen zugewiesen werden, dessen Typ in folgenden Liste weiter rechts steht

byte → short → int → long → float → double

```
int n; double x;  
n = 11;  
n = 4.27;           //nicht erlaubt!!  
x = 2 * n + 1;  
n = x - 0.5;        //nicht erlaubt!!
```

siehe Programm:
WertTausch,
Zuweisung

Initialisierung von Variablen

Initialisierung von Variablen

- Variablen können (und sollen!) bei der Deklaration *initialisiert* werden – d.h. sie erhalten einen Anfangswert
- Variablen können erst verwendet (gelesen) werden, nachdem sie einen Wert bekommen haben!
- Kombination von Deklaration mit Zuweisung → Initialisierung – gehört zu gutem Programmierstil!

```
int sum = 0;  
int k = 3, n = 5 + k;  
double x = 2.75;  
double y = x * x - 2 * x + 3;  
int eins = 2, zwei; //nur eins initialisiert
```

Konstantendeklaration

Um einem bestimmten Wert einen Namen geben zu können, erlaubt Java die Deklaration von *benannten Konstanten* (*named constants*):

- Kennzeichnung durch reservierten Bezeichner `final`
- entspricht initialisierter Variablen, deren Wert nicht mehr verändert werden kann
- Wert wird an einer Stelle festgelegt und kann beliebig oft verwendet werden → guter Programmstil, Programm einfacher änderbar
- Bezeichner aus Großbuchstaben (lt. Konvention)

allgemeine Form

`final type variable = expression;`

beliebiger Typ

```
final double PI = 3.14154;  
final double STEUERSATZ = 20.0;  
final int MAX_VERSUCHE = 3;
```

Syntax, Semantik

Ähnlich einer natürlichen Sprache benötigt auch eine Programmiersprache Regeln und Beschreibungen um korrekte Programme erstellen und erkennen zu können.

Syntax

- Regelwerk für den korrekten Aufbau von Programmen oder Programmteilen
- die Menge aller Syntaxregeln einer Programmiersprache wird *Grammatik* genannt
- z.B. eine Zuweisung besteht aus einer Variablen, einem Zuweisungsoperator und einem Ausdruck, hat also die Form
variable = expression;

Semantik

- gibt die Bedeutung von syntaktisch richtigen Programmteilen an, d.h. sie beschreibt die Wirkung bei der Ausführung
- z.B. lautet die Semantik einer Zuweisung: berechne den Ausdruck auf der rechten Seite des Zuweisungsoperators und weise das Ergebnis der Variablen zu

Einfache Ein-/Ausgabe

Ausgabe

- Verwendung der Standardausgabe von Java, z.B.
 - inkl. Zeilenumbruch: `System.out.println(argument);`
 - ohne Zeilenumbruch: `System.out.print(argument);`
- *argument* besteht (vorläufig) aus Zeichenketten (Strings) und Variablen (Details später):
 - Strings werden gekennzeichnet durch `"..."`
 - Wert einer Variablen wird in einen String umgewandelt
 - Konkatination (Verknüpfung) von Strings und Variablenwerte mittels dem speziellen Operator `+`

```
System.out.println("Bitte Radius eingeben:");  
System.out.println("Umfang = " + umfang);  
System.out.println("x: " + x + " n: " + n);  
System.out.print("Ergebnis: " + x); //kein Zeilenumbruch  
System.out.println(); //nur Zeilenumbruch
```

Einfache Ein-/Ausgabe (Forts.)

Eingabe mittels `SavitchIn`

- benötigt wird die Datei `SavitchIn.class` (oder `.java`) im selben Verzeichnis
- enthält verschiedene Methoden für die Eingabe spezifischer Typen, jeweils ein Wert pro Zeile:
 - `SavitchIn.readLineInt()` → int-Wert einlesen
 - `SavitchIn.readLineDouble()` → double-Wert einlesen
 - `SavitchIn.readLine()` → String einlesen
- eingegebener Wert muss einer Variablen mit passendem Typ zugewiesen werden

Eingabe mittels der Klasse `Scanner`

- `Scanner` ist im Java-System enthalten
- etwas komplizierter für einfache Programme
- Verwendung siehe Programm `KreisMitScanner`

Problem - Algorithmus - Programm

Programme geben Schritt für Schritt ein Lösungsverfahren an – sie beruhen auf einem oder mehreren *Algorithmen (algorithms)*

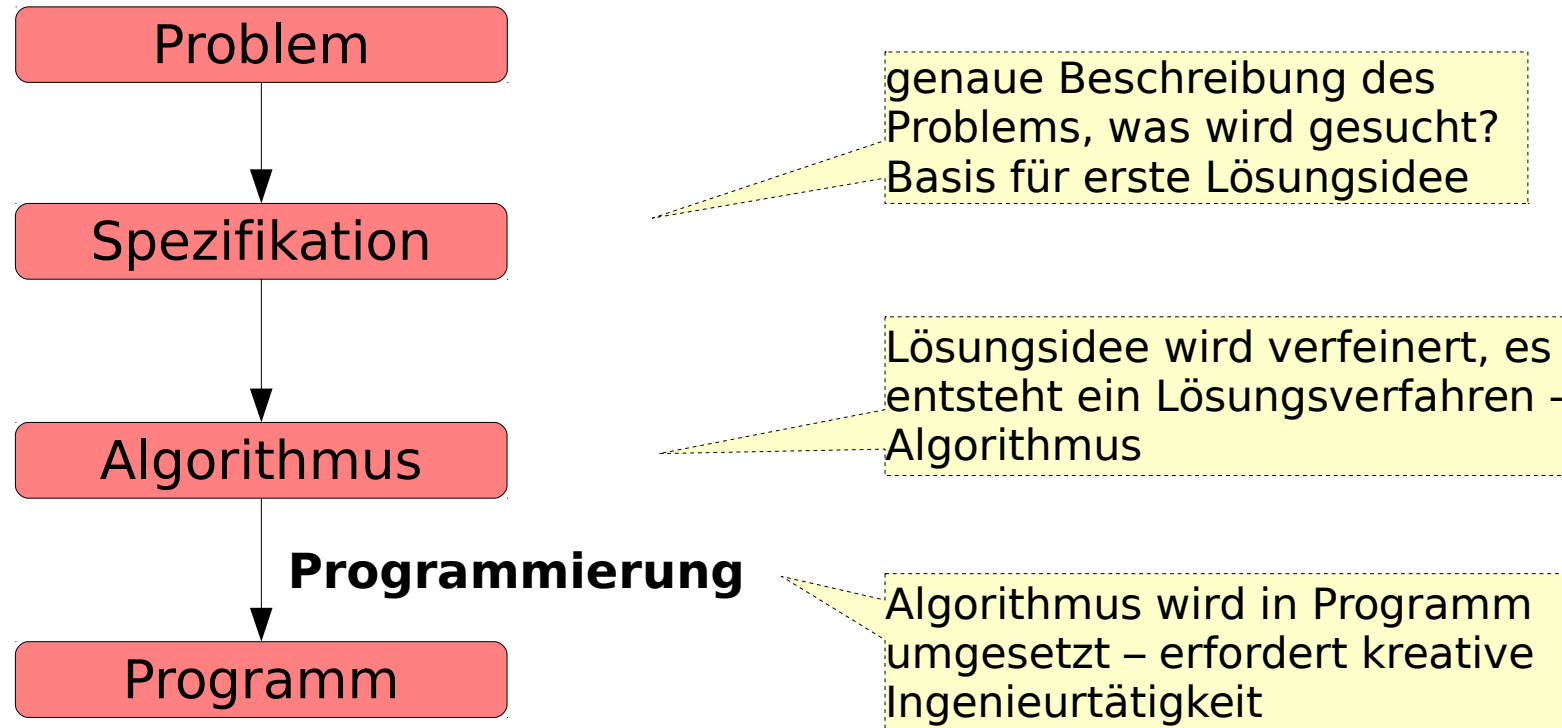
Eine kompakte Definition für *Algorithmus* lautet:

schrittweises, präzises Verfahren zur Lösung eines Problems.

D.h. ein Algorithmus legt eine eindeutige und vollständige Folge von (elementaren) Operationen zur Lösung eines Problems fest.

- Algorithmen existieren in verschiedenen Formen, z.B. natürliche Sprache (Kochrezept, Wegbeschreibung), graphisch (Anleitung um Möbel selbst zusammen zu bauen), Pseudocode, Mischformen, ...
- Informatik: präzise Algorithmen entstehen durch schrittweise Verfeinerung (d.h. immer mehr Details werden beachtet)
- Programmierung heißt: Algorithmus in Programmiersprache umsetzen;
dazu nötig: algorithmisches Denken (mittels Übung zu erlernen!)

Problem - Algorithmus - Programm (Forts.)



Fallbeispiel Wechselgeld

Problem: verschiedene Automaten (für Getränke, Fahrkarten, ...) geben Wechselgeld zurück. Es ist zu einem bestimmten Betrag die erforderliche Menge an Münzen zu bestimmen.

Spezifikation:

Das Programm berechnet die Art und Anzahl der benötigten Euro- und Centmünzen. Die Gesamtanzahl soll möglichst gering sein.

Eingabe: Betrag des Wechselgelds in Euro in üblicher Form (z.B. 3.75).

Ausgabe: ursprünglicher Betrag gemeinsam mit Art der Münzen und deren jeweiliger Anzahl um diesen Betrag darzustellen.

Lösungsidee (erste Variante):

- Variable um Betrag zu speichern
- Variablen um Anzahl der einzelnen Münzen zu speichern
- Berechnung der Anzahl einer Münzart: $\text{Betrag} / \text{Münzwert}$

Fallbeispiel Wechselgeld (Forts.)

Algorithmus (Pseudocode, Version 1):

- * eingegebenen Betrag in Variable `betrag` speichern
- * Anzahl der 2-Euro Münzen berechnen und in `zweiE` speichern
- * restlichen Betrag in `betrag` speichern
- * Anzahl der 1-Euro Münzen für restlichen Betrag berechnen und in `einE` speichern
- * restlichen Betrag in `betrag` speichern
- * obige 2 Schritte für jede fehlende Münzart durchführen, wobei immer die größte verbleibende herangezogen wird (→ Münzanzahl minimieren!)
- * Originalbetrag und Anzahl der Münzarten ausgeben

Zu bedenken:

- Variable `betrag` : Wert in Euro oder Cent?
 - Wert in Cent, ganzzahlig (exakte Ergebnisse, meist schneller)
- Originalbetrag ist am Ende wieder auszugeben, wird jedoch laufend verändert!
 - zwei Variablen: `betragEuro` für eingegebenen Eurowert, `betragCent` für sich ändernden Centwert

Fallbeispiel Wechselgeld (Forts.)

Algorithmus (Pseudocode, Version 2):

- * eingegebenen Betrag in Variable `betragEuro` speichern
- * Betrag in Cent umwandeln
 - `int betragCent = betragEuro * 100;`
- * Anzahl der 2-Euro Münzen berechnen und in `zweiE` speichern
- * restlichen Betrag in `betragCent` speichern
- * Anzahl der 1-Euro Münzen für restlichen Betrag berechnen und in `einE` speichern
- * restlichen Betrag in `betragCent` speichern
- * obige 2 Schritte für jede fehlende Münzart durchführen, wobei immer die größte verbleibende herangezogen wird (→ Münzanzahl minimieren!)
- * Originalbetrag und Anzahl der Münzarten ausgeben

Kapitel 3

if-Anweisung, while-Schleife, Methoden - erste Motivation

if-Anweisung, Verzweigung (vorläufig)

Bisherige Programme: lineare *Anweisungsfolgen (flow of control)*, d.h. Anweisungen wurden nacheinander ausgeführt.

Jedoch ist es oft nötig, Anweisungen nur auszuführen, wenn bestimmte Bedingungen erfüllt sind.

Bsp.: was ist der maximale Wert von 2 Variablen?

Lösung (Algorithmus): Zahlen a , b und Maximum max ;

wenn a größer oder gleich als b dann $\text{max} = a$;

sonst $\text{max} = b$;

```
int a, b, max;
```

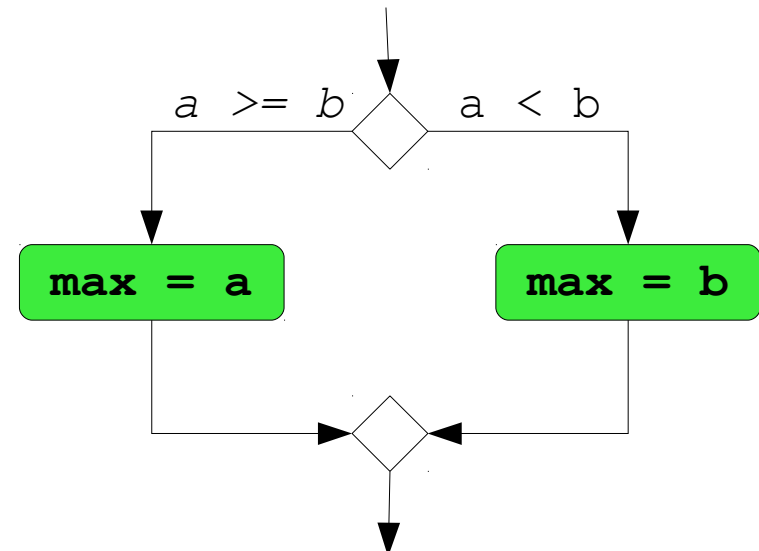
```
if (a >= b)
```

```
    max = a;
```

```
else
```

```
    max = b;
```

Flussdia-
gramm



if-Anweisung, Verzweigung (Forts.)

Ein *if-Anweisung* (*Verzweigung, bedingte Anweisung, if-statement*) prüft eine Bedingung. Abhängig davon ob die Bedingung wahr (true) oder falsch (false) ist, werden verschiedene Anweisungen ausgeführt.

Boolescher Ausdruck: wahr oder falsch
(bzw. trifft zu oder trifft nicht zu)

res. Wort: if

res. Wort: else

allgemeine Form
if (*boolean_expression*)
 statement_1
else
 statement_2

Semantik einer if-Anweisung: ergibt die Auswertung des Booleschen Ausdrucks wahr (true), wird *statement_1* ausgeführt sonst alternativ *statement_2*.

Bsp. siehe Programme `Maximum` und `Muenzwurf`.

if-Anweisung, Verzweigung (Forts.)

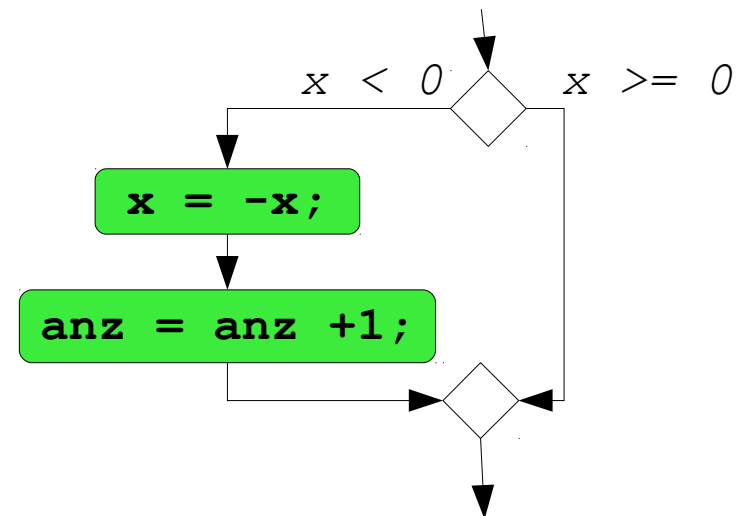
Anstelle einer einzelnen Anweisung in einer Alternative kann ein *Anweisungsblock* verwendet werden – Kennzeichnung mittels `{...}` .

Sowohl einzelne Anweisungen als auch Blöcke sind einzurücken → bessere Lesbarkeit

Der else-Zweig kann auch weggelassen werden.

```
if (x < 0) {  
    x = -x;  
    anz = anz + 1;  
}
```

```
if (a >= b) {  
    max = a;  
    System.out.println("Max. in a");  
}  
else {  
    max = b;  
    System.out.println("Max. in b");  
}
```



Vergleichsoperatoren und Boolesche Werte

- Ergebnis eines Vergleichs ist `true` oder `false` – einfache Form eines *Booleschen Ausdrucks* (*boolean expression*)
- für Vergleiche werden *Vergleichsoperatoren* (*comparison operators*) verwendet
- Vergleichsoperatoren binden schwächer (geringere Priorität) als arithmetische Operatoren

Operator	Bedeutung
<code>==</code>	gleich
<code>!=</code>	ungleich
<code>></code>	größer
<code>>=</code>	größer oder gleich
<code><</code>	kleiner
<code><=</code>	kleiner oder gleich

```
int a = 3, b = 5;

a == 3      //Erg: true
b != a + 1  //Erg: true
a > b       //Erg: false
a >= b - 2  //Erg: true
b < 5       //Erg: false
b <= 2 * a  //Erg: true
```

Achtung: `=` bedeutet Zuweisung und `==` bedeutet Vergleich!

```
if (a = 0) a = a + 1; // Compilerfehler!
```


Schleifen allgemein

Fast alle Programme müssen bestimmte (Gruppen von) Anweisungen wiederholen – z.B. Notendurchschnitt einer Prüfung berechnen, Suchen von Einträgen in Datenbanken, ...

Es wird so lange wiederholt, bis eine bestimmte Bedingung zutrifft (bzw. nicht mehr gültig ist) – dafür nötig **Schleifen (loops)**

Beispiel:

Berechnung von $1+2+3+\dots+n = \sum_{i=1}^n i$

Lösungsidee:

- Variable für Summe (und Teilsummen): `sum`
- Variable für aktuell zu addierenden Wert: `i`
- addiere zum bisherigen Teilergebnis den aktuellen Wert hinzu
- erhöhe den aktuellen Wert um 1
- wiederhole bis Grenze `n` erreicht ist

while-Schleife

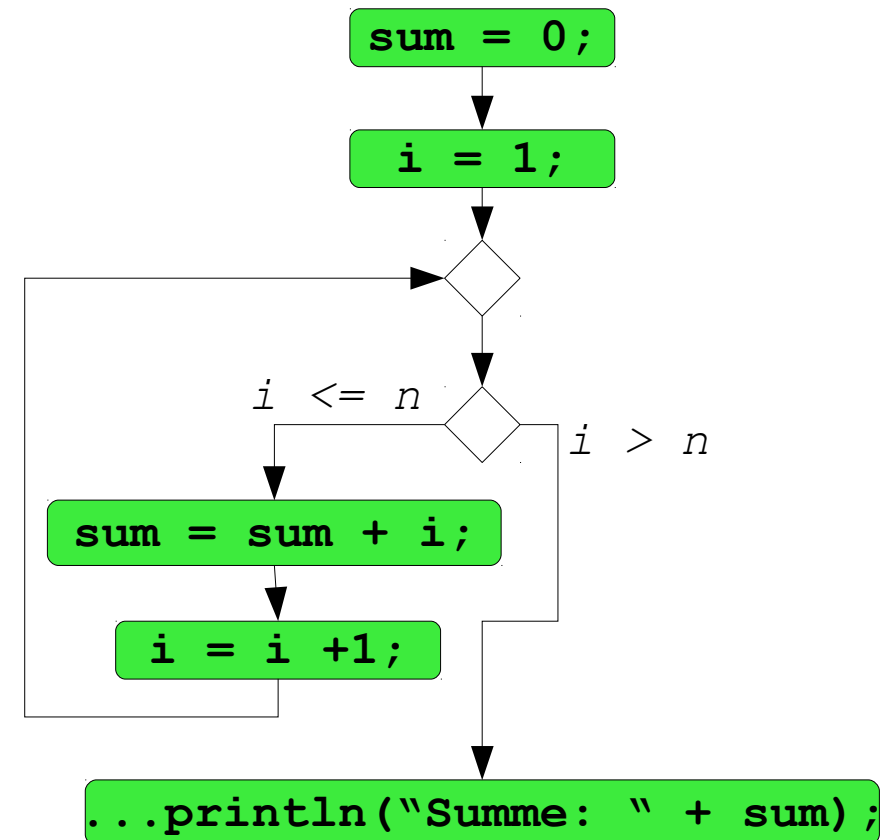
Eine Möglichkeit sind sogenannte *while-Schleifen* (*while-Anweisungen*, *while-statements*)

- Anweisungen (genannt **Schleifenrumpf**, *loop body*) werden wiederholt bis
- eine **Schleifenbedingung** (*controlling boolean expression*) nicht mehr wahr ist

```
sum = 0;  
i = 1;  
  
while (i <= n) {  
    sum = sum + i;  
    i = i + 1;  
}  
...println("Summe: " + sum);
```

Schleifenbedingung

Schleifenrumpf



while-Schleife (Forts.)

res. Wort: while

allgemeine Form

```
while (boolean_expression)  
    body
```

Schleifenbedingung

body / Schleifenrumpf: wenn aus mehreren Anweisungen bestehend muss geklammert werden {...}

Semantik: solange die Schleifenbedingung `true` ist, wird der Schleifenrumpf wiederholt ausgeführt.

- jede Wiederholung des Schleifenrumpfs wird *Iteration* genannt
- Schleifenrumpf kann 0, 1, 2, ... -mal durchlaufen werden (da 0-mal möglich auch *abweisende Schleife* genannt)
- nach Beendigung der Schleife: Schleifenbedingung gilt nicht mehr!

Beispiel: Berechnung des größten gemeinsamen Teilers (ggT) von 2 Zahlen mittels Differenzalgorithmus.

Algorithmus: ziehe die kleinere von der größeren Zahl ab bis zwei gleiche Werte übrig bleiben – dies ist der ggT.

(Siehe Prog. `GGTDifferenz`)

Methoden: erste Motivation

Programme sind i.A. sehr umfangreich, daher

- Aufteilung in kleinere Anweisungsfolgen
- eine Anweisungsfolge erfüllt eine logisch zusammengehörende Teilaufgabe
- so eine Anweisungsfolge erhält einen Namen
- mit diesem Namen kann die Anweisungsfolge beliebig oft aufgerufen werden

Solche Anweisungsfolgen werden **Methoden** genannt!

Methoden sind ein wichtiges Konzept bei der Strukturierung von Programmen.

Methoden: Definition

- eine Methode besteht aus Methodenkopf (header) und -rumpf (body)
- Methodenkopf enthält (vorläufig)
 - reservierte Wörter `static` (s. später) und `void` (kein Rückgabewert)
 - den Namen der Methode
 - Parameter oder leere Klammern ()
- Methodenrumpf besteht aus Deklarationen und Anweisungen, begrenzt mittels {...}

(Siehe Prog. GGT3Zahlen_m1)

Methodenkopf

Name der
Methode

```
static void printHeader() {  
    System.out.println();  
    printSeparator();  
}
```

Methodenrumpf

Methoden: Aufruf (vorläufig)

- Methoden werden außerhalb der main-Methode deklariert
- eine Methoden kann mittels ihrem Namen beliebig oft aufgerufen werden (in main- oder anderen Methoden)
- eine Klasse kann beliebig viele Methoden enthalten

```
public class ... {  
    public static void main (String[] args) {  
        ...  
        printHeader();  
        ...  
        printHeader();  
    }  
  
    static void printHeader() {  
        System.out.println();  
        printSeparator();  
    }  
  
    ... //weitere Methoden  
}
```

Methodenaufruf

Methodenaufruf

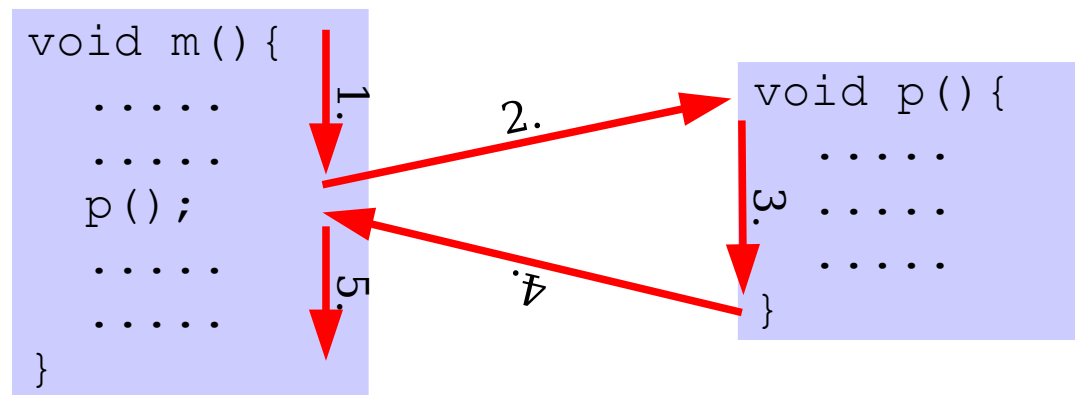
Methodendeklaration

Methoden: Wirkung eines Aufrufs (vorläufig)

schematische Darstellung mittels der Methoden $m()$, $p()$

1. Abarbeitung von m erreicht Aufruf von p
2. Verzweigung zur ersten Anweisung von p
3. Abarbeitung von p
4. nach Beendigung von p erfolgt Rückkehr nach m - und zwar an jene Stelle wo p aufgerufen wurde
5. Fortsetzung von m mit der unmittelbar nächsten Anweisung nach dem Aufruf von p .

Eine aufgerufene Methode kann weitere (andere) Methoden aufrufen!



Methoden: Parameter

Parameter dienen der Übergabe von Werten an die Methode

- Parameter werden im Methodenkopf deklariert
- Parameterdeklarationen haben die selbe Form wie Variablendeklarationen, mehrere Parameter getrennt durch ,
- Aufruf der Methode erfordert die „Übergabe“ passend vieler Parameter
 - der jeweilige Wert wird in den entsprechenden Parameter kopiert - Reihenfolge wichtig!
 - übergebener Wert und Parameter müssen zuweisungskompatibel sein

```
static void printSeparator(int anz) {  
    int i = 1;  
    System.out.print(">");  
    while (i <= anz) {  
        System.out.print("-");  
        i = i + 1;  
    }  
    System.out.println("<");  
}
```

ein int-Parameter

```
int z = ...;  
printSeparator(25);  
printSeparator(z);  
printSeparator(2*z+5);  
  
double d = ...;  
printSeparator(d);  
//Error!
```

(Siehe Prog. GGT3Zahlen_m2)

Methoden mit Rückgabewert

Methoden können auch einen Ergebniswert an den aufrufenden Programmteil zurückgeben.

- Methodenkopf enthält statt `void` den Typ des Rückgabewerts
- Methodenrumpf muss mindestens eine `return`-Anweisung enthalten

Typ des Rückgabewerts

```
static int ggt(int a, int b){  
    while (a != b)  
        if (a > b)  
            a = a - b;  
        else  
            b = b - a;  
    return a;  
}
```

return-Anweisung

somit 2 Arten von Methodenaufrufen

- ohne Rückgabewert: Aufruf als Anweisung, abgeschlossen mit ;
- mit Rückgabewert: überall einsetzbar, wo ein Wert des Typs des Rückgabewerts verwendet werden darf

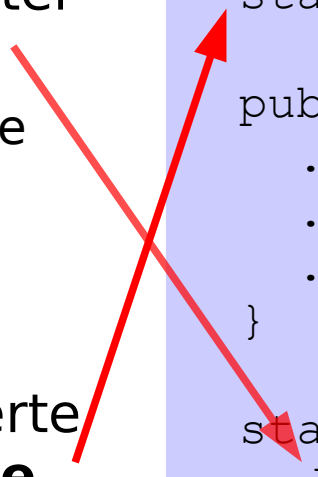
(Siehe Prog. GGT3Zahlen_m3)

```
int erg, x, y;  
x = ...; y = ...;  
printSeparator(25);  
  
erg = ggt(x, y);  
erg = (2 * ggt(x, 12)) % 3;  
System.out.println(  
    "Ergebnis: " + ggt(x, y));
```

Methoden: lokale und globale Variablen (vorläufig)

Eine Methode kann auch Deklarationen von Variablen enthalten.

- *innerhalb* einer Methode deklarierte Variable werden **lokale Variable** genannt (gilt auch für die Parameter einer Methode)
 - können nur innerhalb dieser Methode verwendet werden
 - Existenz endet mit Beendigung der Methode
- *außerhalb* einer Methode deklarierte Variable werden **globale Variable** genannt
 - werden vorläufig mit `static` definiert
 - können in allen Methoden der Klasse verwendet werden
 - existieren solange das Programm läuft



```
public class ... {  
    static int a;  
  
    public ... main (...){  
        ...  
        ...println(a);  
        ...  
    }  
  
    static void demo(int p){  
        double d;  
        ...  
        d = a * d;  
        ...  
    }  
}
```

Kapitel 4

**Typen, if-Anweisung (Erg.),
Boolean, char, String, switch, ...**

Programme mit Fehlern

Fehler beim Programmieren (*bugs, errors*) sind „ganz normal“.

Syntaxfehler (*syntax error*)

- Verstoß gegen die Sprachregeln
- Compiler erkennt dies und gibt eine Fehlermeldung aus (*compile-time error*)
 - Fehlermeldung gibt auch die Art des Fehlers an – dies muss nicht immer stimmen, der Compiler gibt eine Vermutung an!
 - immer versuchen, den ersten Fehler zu beheben und neu kompilieren (meist sind die weiteren Fehler vom ersten abhängig)
- z.B. fehlender “;” am Anweisungsende, fehlende schließende Klammer, falscher Typ bei Zuweisung, ...

Programme mit Fehlern (Forts.)

Laufzeitfehler (Run-time error)

- tritt während des Programmlaufs auf
- Fehlermeldung (oft schwer verständlich) wird erstellt und Programmlauf abgebrochen
- z.B. Speicher erschöpft, Eingabe nicht richtig interpretierbar, benötigte Datei nicht zu finden, ...

Logischer Fehler (logical error)

- Fehler im zugrunde liegenden Algorithmus
- inkorrekte Umsetzung in der Programmiersprache – jedoch den Syntaxregeln entsprechend
- keine Fehlermeldungen vom Compiler oder zur Laufzeit – jedoch liefert das Programm *falsche* Ergebnisse!
- sind schwer zu finden – erfordern logische Analyse und Programmtests
- z.B. falschen Operator verwendet (“/“ statt “%“), falsche Variable für Ausgabe verwendet, Denkfehler im Algorithmus, ...

Programmierstil

```
public class Unlesbar{public static void  
main(String[]args){double radius,umfang,flaeche;final  
doublePI=3.14159;System.out.println();System.out.println  
("Bitte Radius  
eingeben:");radius=SavitchIn.readLineDouble();umfang=2*P  
I*radius;flaeche=PI*radius*radius;System.out.println("Um  
fang  = "+umfang);System.out.println("Flaeche  =  
"+flaeche);System.out.println();}}
```

Ist dies ein Java-Programm?

→ ja, wird vom Compiler ohne Fehlermeldung übersetzt und kann ausgeführt werden

Was berechnet dieses Programm?

→ berechnet Umfang und Fläche eines Kreises – ist jedoch schwer zu erkennen

Daher wichtig: *guter Programmierstil*, um Lesbarkeit und damit auch Wartbarkeit von Programmen zu erleichtern!

Programmierstil (Forts.)

Für Java existieren Richtlinien zur Gestaltung von gut lesbaren Programmen.

Namen für Bezeichner

- Variablen: beginnen mit Kleinbuchstaben
z.B: `radius`, `betragEuro`, `timeAverage`
- Konstante: bestehen aus Großbuchstaben
z.B: `PI`, `MIN_TIME`, `MAX`
- Klassen: beginnen mit Großbuchstaben
z.B. `Kreis`, `WechselGeld`, `SavitchIn`
- Methoden: beginnen mit Kleinbuchstaben
z.B. `readLineDouble()`, `printHeader()`, `ggt(int a, int b)`

Zusammengesetzte Wörter werden mittels Großbuchstaben gekennzeichnet.

Auch möglich: Unterstreichungszeichen (underliner).

Als Sprache soll Englisch (meist kürzere Namen) oder Deutsch verwendet werden – jedoch einheitlich!

Programmierstil (Forts.)

Einrücken

- dienen der leichten Erkennbarkeit der Programmstruktur
- Einrücktiefe 2 bis 4 Leerzeichen – auf jeden Fall einheitlich!
- i.A. eine Anweisung pro Zeile
 - Ausnahme: kurze zusammengehörende Anweisungen können auch in einer Zeile stehen

Kommentare

- sollen wichtige Informationen ergänzen, die nicht aus dem Code erkennbar sind
- Verwendung als Trennung / Einleitung von Anweisungsfolgen
z.B. `//Muenzanzahlen berechnen`
- Richtlinie: Kommentare sparsam, aber gezielt einsetzen

Primitive Typen von Java

Allgemein bestimmt ein (*Daten-*)*Typ* welche Werte eine Variable speichern kann – Wertebereich wird festgelegt

Java unterscheidet 2 Arten von Typen

- *primitive Typen*: einfache Werte wie Zahlen, Zeichen, Wahrheitswerte
- *Klassentypen* (siehe später)

Folgende *primitive Typen* (*Grundtypen*, *primitive types*) sieht Java vor:

Typen für ganze Zahlen (integers)		
Typ	Speicherbedarf	Wertebereich
byte	1 Byte	$-2^7 \dots 2^7-1$ (-128 ... +127)
short	2 Byte	$-2^{15} \dots 2^{15}-1$ (-32768 ... +32767)
int	4 Byte	$-2^{31} \dots 2^{31}-1$ (-2147483648 ... 2147483647)
long	8 Byte	$-2^{63} \dots 2^{63}-1$ ($\sim \pm 9 \cdot 10^{18}$)

Primitive Typen von Java (Forts.)

Typen für Gleitkommazahlen (floating point numbers)

Typ	Speicherbedarf	Wertebereich
float	4 Byte	$\pm 3.40282347 \cdot 10^{38} \dots$ $\pm 1.40239846 \cdot 10^{-45}$
double	8 Byte	$\pm 1.79769313486231570 \cdot 10^{308} \dots$ $\pm 4.94065645841246544 \cdot 10^{-324}$

Typ für einzelne Zeichen (characters)

Typ	Speicherbedarf	Wertebereich
char	2 Byte	alle Unicode-Zeichen

Typ für Wahrheitswerte (Boole'sche Werte, boolean values)

Typ	Speicherbedarf	Wertebereich
boolean	1 Bit	true, false

Typkonversion (type cast)

I.A. sollen in einem Ausdruck alle Operanden den selben Typ besitzen (Erinnerung: „keine Addition von Äpfel und Birnen“).

Unter bestimmten Bedingungen ist die Umwandlung eines Wertes in einen anderen Typ nötig. Dies kann automatisch (implizit) oder im Programm erzwungen werden (explizit).

Die arithmetischen Operatoren `+`, `-`, `*`, `/` können sowohl mit ganzen Zahlen als auch mit Gleitkommazahlen verwendet werden (gilt auch für `%`, jedoch selten benötigt bei Gleitkommazahlen)!

Welchen Typ besitzt nun ein Ausdruck, dessen Operanden verschiedene Typen aufweisen?

```
int i = 3;
double d = 2.5;

//Typ, Wert, erlaubt??
d = 2*i + 0.5;
i = d - 0.5 + 1;
```

Typkonversion (type cast) (Forts.)

Folgende Regeln gelten in Java für arithmetische Operatoren um den Typ eines Ausdrucks festzulegen:

- Typ eines Numerals für ganzzahlige Werte: `int`
- Typ eines Numerals für Gleitkommazahlen: `double`
- bei einem Operator mit verschiedenen Operandentypen wird der gemäß Zuweisungskompatibilität „weiter links“ stehende Typ in den anderen beteiligten Typ konvertiert. Der Ausdruck erhält dann den gleichen Typ wie seine Operanden, „mindestens“ aber den Typ `int`.

(Erinnerung: `byte` → `short` → `int` → `long` → `float` → `double`)

→ *implizite Typkonversion*, wird automatisch vorgenommen

```
byte b; int i; float f; double d;  
4 + 3      //int  
2 + 1.0    //double  
f + i      //float  
b + b      //int - nicht byte  
d / 7      //double mit Nachkommastellen  
i / 7      //int, ganzzahlig
```

Typkonversion (type cast) (Forts.)

Eine Typkonversion kann auch im Programm festgelegt werden
→ *explizite Typkonversion*

allgemeine Form
(*type*) *expression*

dabei ist zu beachten

- Ausdruck wird berechnet, dann in Zieltyp umgewandelt
- selbe Priorität wie Vorzeichenoperatoren
- abschneiden der Nachkommastellen z.B. bei `double` nach `int`
- allgemein: Zieltyp muss umzuwandelnden Wert darstellen können – sonst falscher Wert!

Vorsicht: type cast ist Fehlerquelle
– nur wenn nötig verwenden

siehe Programm:
TypeCast

```
byte b = 127; int i = 4; double d = 4.6;
d = i / 10;           //d: 0.0
d = (double)i / 10;   //d: 0.4
i = d * 4;            //nicht erlaubt
i = (int)(d) * 4;      //i: 0
i = (int)(d * 4);      //i: 1
b = (byte)(i * 128)    //b: -128 !!!Ueberlauf
```

[Wh.] if-Anweisung, Verzweigung

Bisherige Programme: linearen *Anweisungsfolgen* (*flow of control*), d.h. Anweisungen wurden nacheinander ausgeführt.

Jedoch ist es oft nötig, Anweisungen nur auszuführen, wenn bestimmte Bedingungen erfüllt sind.

Bsp.: was ist der maximale Wert von 2 Variablen?

Lösung (Algorithmus): Zahlen a , b und Maximum max ;

wenn a größer oder gleich als b dann $\text{max} = a$;

sonst $\text{max} = b$;

```
int a, b, max;
```

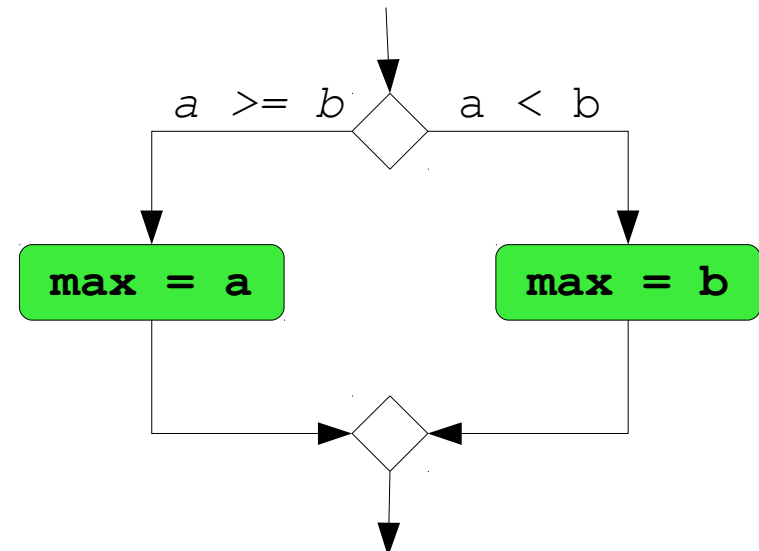
```
if (a >= b)
```

```
    max = a;
```

```
else
```

```
    max = b;
```

Flussdiagramm



Geschachtelte if-Anweisungen

Eine if-Anweisung ist eine *zusammengesetzte* Anweisung - d.h. ihre Teile sind auch wieder Anweisungen.

Wenn diese Teile selbst wieder if-Anweisungen sind, wird von *geschachtelten* if-Anweisungen gesprochen.

Z.B.: Maximum von 3 Zahlen

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

oder anders formatiert

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

if-Anweisung: Dangling else

Welchen Wert besitzt die Variable `erg` nach Ausführung des folgenden Programmstücks für `a=1` und `b=2` ?

```
if (a >= b)
    if (a > 0)
        erg = a;
else
    erg = b;
```

Ist dies das selbe?

```
if (a >= b)
    if (a > 0)
        erg = a;
else
    erg = b;
```

Dangling else: zu welchem `if` gehört das `else`?

Mehrdeutigkeit ist nicht erlaubt – daher gilt folgende Regel: `else` bezieht sich immer auf das textuell letzte freie `if` im selben Block.

→ beide Varianten für den Compiler gleich, die rechte ist richtig formatiert.

Soll das `else` zum ersten `if` gehören, dann sind `{ }` zu verwenden.

```
if (a >= b) {
    if (a > 0)
        erg = a;
}
else
    erg = b;
```

siehe Programm: `DanglingElse`

Mehrfachverzweigungen, if-Kaskaden

if-Anweisungen sind beliebig tief schachtelbar – führt zu Mehrfachverzweigungen bzw. if-Kaskaden.

```
if (punkte > 21)
    note = 1;
else
    if (punkte > 18)
        note = 2;
    else
        if (punkte > 15)
            note = 3;
        else
            if (punkte >= 12)
                note = 4;
            else
                note = 5;
```

gleichbedeutend mit

```
if (punkte > 21)
    note = 1;
else if (punkte > 18)
    note = 2;
else if (punkte > 15)
    note = 3;
else if (punkte >= 12)
    note = 4;
else
    note = 5;
```

Beide Varianten sind für den Compiler gleich – rechte ist besser lesbar!

Wirkung: gehe zur ersten erfüllten Bedingung und führe zugehörige Anweisung aus.

Datentyp boolean, Boolesche Werte

Für *Wahrheitswerte* (*Boolesche Werte*, *logical values*) steht der primitive Typ `boolean` zur Verfügung.

(Benannt nach George Boole, englischer Mathematiker, 19.Jhd.)

- `boolean` besitzt 2 Werte: `true` für wahr, `false` für falsch
- `true` und `false` sind vordefinierte Literale
- `boolean`-Variable können Boolesche Werte speichern
- `boolean` ist nicht kompatibel zu numerischen Typen
- Vergleiche liefern als Ergebnis einen Wert vom Typ `boolean` (z.B. die Bedingungen bei `if`-Anweisungen)

```
boolean p, q;  
int a = -3;  
p = false;  
q = a < 0;    //q: true  
p = p && q;    //p: false
```

[Wh.] Vergleichsoperatoren und Boolesche Werte

- Ergebnis eines Vergleichs ist `true` oder `false` – einfache Form eines *Booleschen Ausdrucks* (*boolean expression*)
- für Vergleiche werden *Vergleichsoperatoren* (*comparison operators*) verwendet
- Vergleichsoperatoren binden schwächer (geringere Priorität) als arithmetische Operatoren

Operator	Bedeutung
<code>==</code>	gleich
<code>!=</code>	ungleich
<code>></code>	größer
<code>>=</code>	größer oder gleich
<code><</code>	kleiner
<code><=</code>	kleiner oder gleich

```
int a = 3, b = 5;

a == 3      //Erg: true
b != a + 1  //Erg: true
a > b       //Erg: false
a >= b - 2  //Erg: true
b < 5       //Erg: false
b <= 2 * a  //Erg: true
```

Achtung: `=` ist Zuweisung und `==` ist Vergleich!

```
if (a = 0) a = a + 1; // Compilerfehler!
```

Boolesche Werte und logische Operatoren

- Boolesche Werte (und damit auch Vergleiche) können mit *logischen Operatoren* verknüpft werden – *Boolesche Ausdrücke* (*boolean expressions*)
- Ergebnistyp eines Booleschen Ausdrucks: `boolean`
- Wahrheitstafeln definieren logische Operatoren
`&&` logisches Und (and) `||` logisches Oder (or) `!` Negation (not)
- Vorrangregeln: `!` bindet stärker als `&&`
`&&` bindet stärker als `||`

`&&` logisches Und

x	y	x && y
true	true	true
true	false	false
false	true	false
false	false	false

`||` logisches Oder

x	y	x y
true	true	true
true	false	true
false	true	true
false	false	false

`!` Negation

x	!x
true	false
false	true

Boolesche Werte und logische Operatoren (Forts.)

- logische Operatoren für Bereichstest nötig:
Temperatur von 20 bis 24 Grad
- `boolean`-Variable anstelle von Vergleichen bei `if`-Anweisungen verwendbar
- `!` wenn möglich vermeiden
- Klammern heben Vorrangregeln auf
- `!` hat selbe Priorität wie Vorzeichen `+`, `-`

```
double temp; boolean klimaEin;  
// Bereich testen: 20 < temp < 25  
if (temp > 20 && temp < 25)  
    System.out.println("angenehm");
```

```
// außerhalb des Bereichs  
if (temp <= 20 || temp >= 25)  
    System.out.println("nicht angenehm");
```

```
klimaEin = temp <= 20 || temp >= 25;  
if (klimaEin)  
    System.out.println("eingeschaltet");
```

```
if (!(temp > 19))  
    klimaEin = true;  
  
// besser:  
if (temp <= 19)  
    klimaEin = true;
```

Logische Operatoren und teilweise Auswertung

Bei einer Verknüpfung mit den logischen Operatoren `&&` und `||` wird abgebrochen, wenn das Ergebnis feststeht –
Kurzschlussauswertung (teilweise Auswertung, short-circuit evaluation, lazy evaluation)

Klassischer Anwendungsfall: Vermeidung einer Division durch 0

für `n==0` Division durch 0 → Laufzeitfehler

```
if ((n != 0) && (sum / n > 70)) erg = 1;
```

wenn false → gesamter Ausdruck false und zweiter Operand von `&&` wird nicht mehr ausgewertet

äquivalent mit

```
if (n != 0)
    if (sum / n > 70)
        erg = 1;
```

```
if ((n == 0) || (sum / n < 50)) erg = 0;
```

wenn true → gesamter Ausdruck true und zweiter Operand von `||` wird nicht mehr ausgewertet

äquivalent mit

```
if (n == 0)
    erg = 0;
else if (sum / n < 50)
    erg = 0;
```

Welches Programm ist besser?

```
//ProgA
if (a >= b)
    if (a >= c)
        max = a;
    else
        max = c;
else
    if (b >= c)
        max = b;
    else
        max = c;
```

```
//ProgB
max = a;
if (b > max)
    max = b;
if (c > max)
    max = c;
```

Was ist mit „besser“ gemeint?

- Kürze: ProgB ist kürzer
- Effizienz:
 - ProgA benötigt 2 Vergleiche und 1 Zuweisung
 - ProgB benötigt 2 Vergleiche und im Schnitt 2 Zuweisungen
- Lesbarkeit?
- Erweiterbarkeit?

Datentyp char

Zur Repäsentation und Darstellung von einzelnen *Zeichen* (*characters*) existiert der primitive Datentyp `char` .

- `char`-Literele gekennzeichnet durch einfache Hochkommas `'...'`
- Variablendeklarationen wie bei anderen primitiven Typen
- Zeichen werden durch Zahlen codiert, es existieren verschiedene Standards
 - ASCII (American Standard Code for Information Interchange): 1 Zeichen – 1 Byte (128 Zeichen darstellbar); z.B. keine Umlaute; siehe z.B. <http://www.asciitable.com/>
 - Unicode: 1 Zeichen – 2 Bytes (65536 Zeichen darstellbar); Umlaute, griechische Zeichen, ..., ASCII ist Teilmenge davon
siehe z.B. <http://www.unicode.org/>

```
char c = 'p';
```


Spezielle Zeichen und int-Kompatibilität des Typs char

Für häufig vorkommende Steuerzeichen und Darstellung spezieller Zeichen stellt Java sogenannte *escape-Sequenzen* (*Ersatzdarstellungen*) zur Verfügung

- beginnen mit Backslash \
- können in Zeichen- und Zeichenkettenkonstanten (String-Konstanten) verwendet werden

- die wichtigsten:

\n ... Zeilenvorschub (newline)

\\ ... Backslash als Zeichen

\\" ... doppeltes Hochkomma als Zeichen

\' ... Hochkomma als Zeichen

```
//char " speichern  
char c = '\\"';
```

Zuweisung von char-Werte an int-Variablen ist möglich

- char-Werte sind wie short-Werte Teilmenge von int (siehe Hierarchie primitiver Typen)
- int-Wert entspricht jenem des char-Werts in der Unicodetabelle!
- Umkehrung ist nicht möglich!

```
char c = '8'; int a;  
a = c; //a: 56
```

Operationen mit char-Werten

- Vergleiche mit den üblichen Vergleichsoperatoren
 - geordnet nach Wert in der Unicode-Tabelle
 - Ziffern < Großbuchstaben < Kleinbuchstaben
 - innerhalb einer Gruppe direkt aufeinander folgend
 - innerhalb einer Gruppe die übliche Reihenfolge
- arithmetische Operationen mit den üblichen Operatoren
 - Achtung: Ergebnistyp ist i.A. int !

```
char c; boolean istZiffer;  
if ('0' <= c && c <= '9')  
    istZiffer = true;  
else  
    istZiffer = false;  
  
//besser  
istZiffer = ('0' <= c) && (c <= '9');
```

```
char c = '8'; int a;  
a = c + c;    //a: 112  
c = c + c;    //Compilerfehler!
```

Zeichenketten - Strings

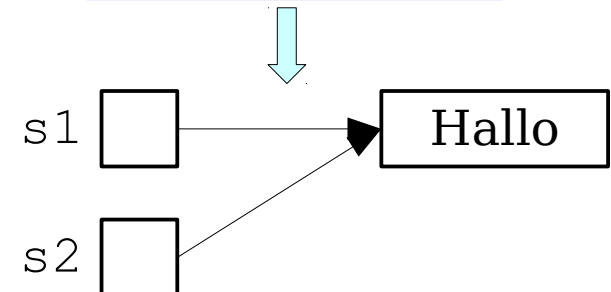
Zur Darstellung und Verwaltung von Zeichenketten wird in Java der vordefinierte Typ (bzw. Klasse) *String* verwendet.

- Bibliothekstyp für Zeichenketten (-reihen) bestehend aus `char`-Elementen
- Stringlitterale gekennzeichnet durch doppelte Hochkomma "..."
- Unterschied leerer String "" und String bestehend aus einem Leerzeichen " "
- Stringvariablen sind Zeiger (Referenzen) auf Stringobjekte – d.h. Stringvariablen werden Zeigerwerte und nicht eigentliche Werte zugewiesen (im Gegensatz zu primitiven Datentypen!)

```
String s1, s2;  
s1 = "Hallo";  
s2 = "Guten Tag!";
```

```
//leerer String  
s1 = "";  
//Blank als String  
s2 = " ";
```

```
s1 = "Hallo";  
s2 = s1;
```



Zeichenketten – Strings (Forts.)

- Stringobjekte sind unveränderlich – d.h. bei Manipulation wird neues Stringobjekt erstellt
- Strings werden mit dem Operator `+` verkettet (*Konkatenation*);
Ergebnis: neuer String mit Inhalte der Operanden hintereinander
- Konkatenation von Strings mit primitiven Typen: implizite Typkonversion um Wert als String darzustellen
(z.B. bei Ausgaben schon verwendet)
- escape-Sequenzen in Strings verwendbar und oft nützlich

```
String s1, s2;  
s1 = "Guten ";  
s2 = s1 + "Tag";  
    //Erg: "Guten Tag"  
s2 = "4" + "2";    //Erg: "42"
```

```
int a = 7; String s1;  
System.out.println("a: " + a);  
  
s1 = a + "5";    //Erg: "75"  
s1 = 1 + s1;     //Erg: "175"  
s1 = (3 == 3) + "wahr";  
    //Erg: "truewahr"
```

```
int a = 7; String s1;  
System.out.println("a: " + a + "\n a+1: " + (a+1));  
  
s1 = "Ein \" im String";  
System.out.println (s1 + "\nneue Zeile");
```

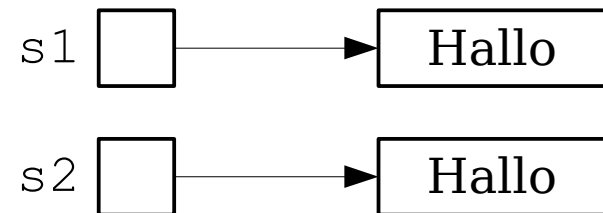
Vergleiche von Strings

Vorsicht bei Vergleichen von Strings!

- Stringvariable enthalten Zeiger → mit `==` Vergleich von Zeigern, nicht von Werten!
- für Vergleich von String-Werten → **immer** Methode `equals` verwenden:

`s1.equals(s2)` vergleicht die Inhalte von `s1` und `s2`

```
String s1 = "Hallo", s2;  
s2 = SavitchIn.readLine();  
    //z.B. String "Hallo" eingeben  
  
if (s1 == s2) ...  
    //Zeigervergleich: false  
if (s1.equals(s2) ...  
    //Wertevergleich: true
```



Achtung: String-Literale gleichen Inhalts werden als nur ein Stringobjekt gespeichert – könnte Verwirrung schaffen!

```
String s1 = "Hallo";  
if (s1 == "Hallo") ...  
    //Zeigervergleich: true
```

Stringoperationen

Der Typ (Klasse) String bietet einige Operationen an, z.B.

- `length`-Methode: liefert Anzahl der Zeichen des Strings; Ergebnistyp: `int`
- `charAt`-Methode: nennt Zeichen an angegebener Position; Zählung beginnt mit 0! Ergebnistyp: `char`
- `compareTo`-Methode: lexikographischer Vergleich von 2 Strings -
`s1.compareTo(s2)` hat Ergebnistyp `int` !
 `s1` kleiner `s2` → Ergebnis `< 0`
 `s1` gleich `s2` → Ergebnis `= 0`
 `s1` größer `s2` → Ergebnis `> 0`
- weitere
 siehe
 Literatur

```
String s1 = "Jetzt neu";  
int anz = s1.length();  
    //Erg: 9
```

```
char c = s1.charAt(3);  
    //Erg: z
```

```
int vergl;  
vergl = s1.compareTo("Prog");   //Erg: -6  
  
if (s1.compareTo("Jetzt neu") == 0) //Bedingung: true  
    ...
```

switch-Anweisung

Eine Mehrwegverzweigung wird durch eine *switch-Anweisung* ermöglicht (vergleiche: if-Anweisung ist Zweiwegverzweigung).

```
int note = ...;
```

switch-Ausdruck

case-Label

break-
Anweisung

default-Label

```
switch(note) {  
    case 1:  
        System.out.println("mit Erfolg teilgenommen");  
        break;  
    case 0:  
        System.out.println("ohne Erfolg teilgenommen");  
        break;  
    default:  
        System.out.println("ungueultig");  
        break;  
}
```

Semantik:

1. switch-Ausdruck berechnen
2. Sprung zum passenden case-Label
 - wenn keines passt, springe zu default
 - wenn kein default vorhanden, springe ans Ende der switch-Anweisung

switch-Anweisung (Forts.)

- break-Anweisung
 - springt ans Ende der switch-Anweisung
 - fehlt ein break, wird mit den Anweisungen des nächsten case-Labels weiter gemacht (oft Fehlerursache – *fall through*)
- weitere Bedingungen
 - switch-Ausdruck muss `int` oder `char` sein (auch `String` erlaubt)
 - case-Labels müssen Konstante sein
 - Typ der case Labels muss zum Typ des switch-Ausdrucks passen
 - case-Labels müssen alle verschieden sein
 - default-Label kann höchstens einmal vorkommen und muss dann am Ende stehen
 - mehrere case-Labels mit einer gemeinsamen Anweisungsfolge sind möglich (siehe Programm `TageProMonat`)

Kapitel 5

Schleifen (Loops)

[Wh.] Schleifen allgemein

Fast alle Programme müssen bestimmte (Gruppen von) Anweisungen wiederholen – z.B. Notendurchschnitt einer Prüfung berechnen, Suchen von Einträgen in Datenbanken, ...

Es wird so lange wiederholt, bis eine bestimmte Bedingung zutrifft (bzw. nicht mehr gültig ist) – dafür nötig **Schleifen (loops)**

Beispiel:

Berechnung von $1+2+3+\dots+n = \sum_{i=1}^n i$

Lösungsidee:

- Variable für Summe (und Teilsummen): `sum`
- Variable für aktuell zu addierenden Wert: `i`
- addiere zum bisherigen Teilergebnis den aktuellen Wert hinzu
- erhöhe den aktuellen Wert um 1
- wiederhole bis Grenze `n` erreicht ist

[Wh.] while-Schleife

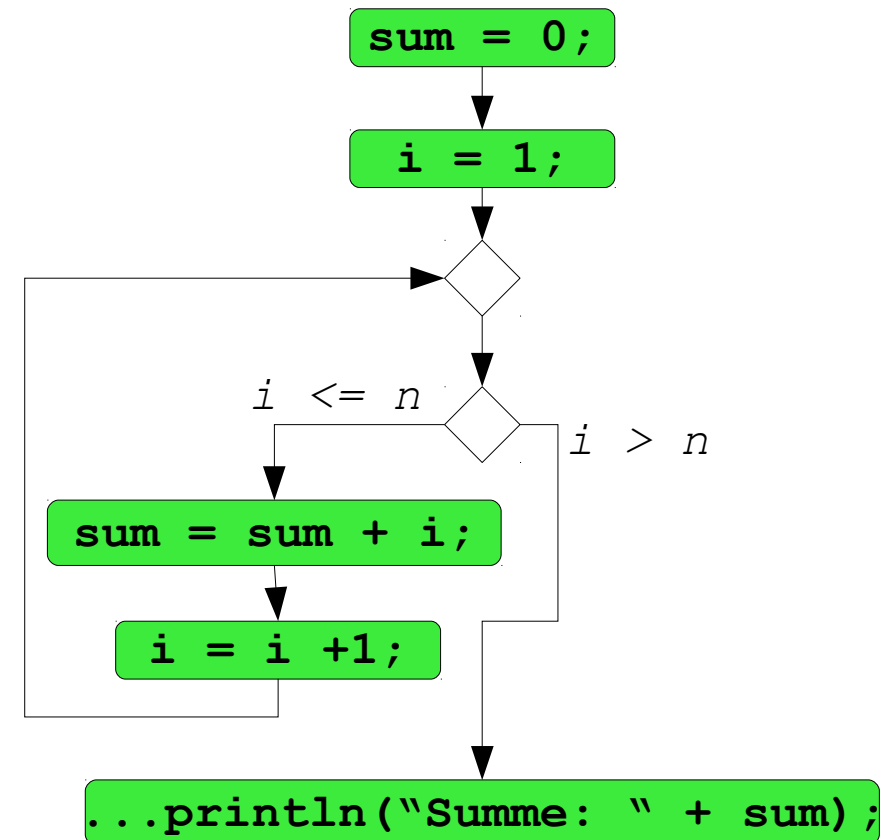
Eine Möglichkeit sind sogenannte *while-Schleifen* (*while-Anweisungen*, *while-statements*)

- Anweisungen (genannt **Schleifenrumpf**, *loop body*) werden wiederholt bis
- eine **Schleifenbedingung** (*controlling boolean expression*) nicht mehr wahr ist

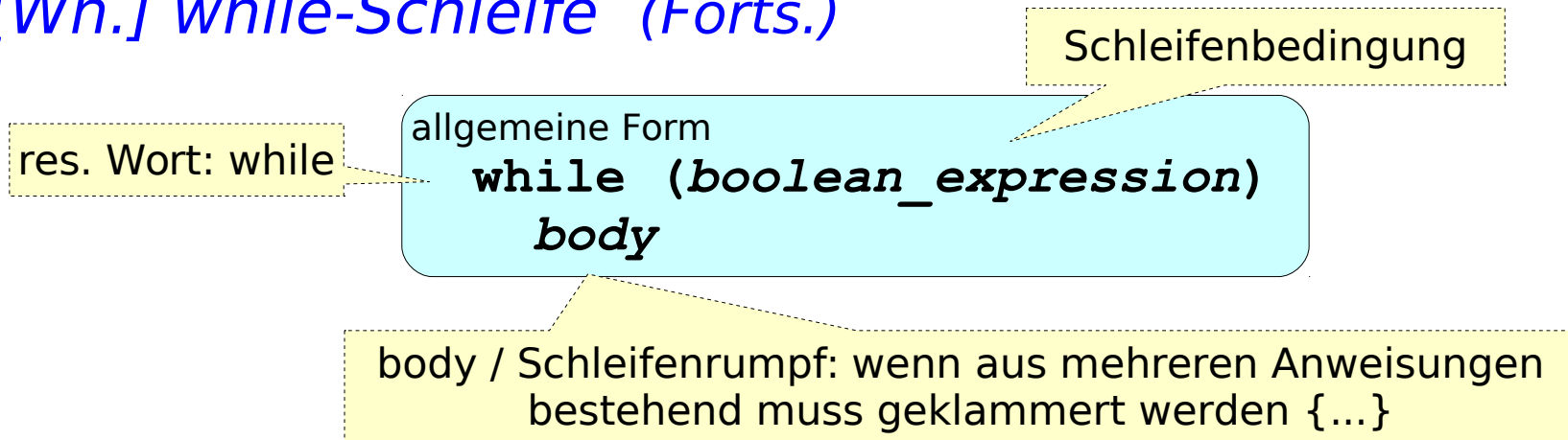
```
sum = 0;  
i = 1;  
  
while (i <= n) {  
    sum = sum + i;  
    i = i + 1;  
}  
...println("Summe: " + sum);
```

Schleifenbedingung

Schleifenrumpf



[Wh.] while-Schleife (Forts.)



Semantik: solange die Schleifenbedingung `true` ist, wird der Schleifenrumpf wiederholt ausgeführt.

- jede Wiederholung des Schleifenrumpfs wird *Iteration* genannt
- Schleifenrumpf kann 0, 1, 2, ... -mal durchlaufen werden (da 0-mal möglich auch *abweisende Schleife* genannt)
- nach Beendigung der Schleife: Schleifenbedingung gilt nicht mehr!

Beispiel: Berechnung des größten gemeinsamen Teilers (ggT) von 2 Zahlen mittels Differenzalgorithmus.

Algorithmus: ziehe die kleinere von der größeren Zahl ab bis zwei gleiche Werte übrig bleiben – dies ist der ggT.

(Siehe Prog. `GGTDifferenz`)

do-while-Schleife

Ähnlich einer while-Schleife – jedoch wird die Schleifenbedingung am Ende jedes Schleifendurchlaufs überprüft

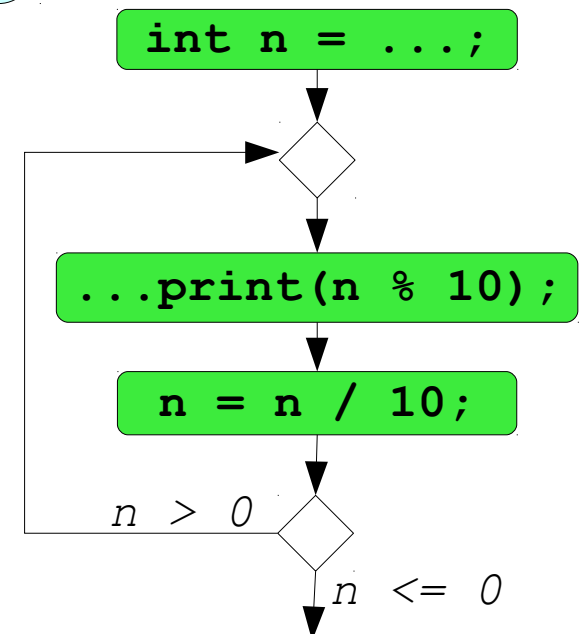
→ Schleifenrumpf wird mindestens 1-mal durchlaufen
(daher auch *Durchlaufschleife* genannt)



(Siehe Prog.
ZiffernSturz)

```
//Ziffern in umgekehrter
// Reihenfolge
int n = ...;

do {
    ...print(n % 10);
    n = n / 10;
} while (n > 0);
```



Inkrement-Operator

Oft wird die Erhöhung eines Variablenwerts um 1 benötigt.

→ Java stellt speziellen *Inkrement-Operator* (*increment operator*) zur Verfügung: `++`

- als eigenständige Anweisung: `n++;` oder `++n;`
beide gleichbedeutend mit `n = n + 1;`

```
n = 3;  
n++;    //n: 4
```

- nur auf Variablen anwendbar (nicht auf Ausdrücke)
- für Variablen mit numerischen Typ – meistens jedoch für ganzzahligen Typ sinnvoll
- auch in Ausdrücken verwendbar – dann jedoch kompliziertere Semantik (von Verwendung wird abgeraten!):
 - `n++` zuerst Wert von `n` lesen, dann Wert von `n` um 1 erhöhen;
 - `++n` zuerst Wert von `n` um 1 erhöhen, dann Wert von `n` lesen;

```
n = 2;  
erg = n++ * 3; //erg: 6, n: 3
```

```
n = 2;  
erg = ++n * 3; //erg: 9, n: 3
```

Dekrement-Operator, Operator mit Zuweisung

Dekrement-Operator

Analog zum Inkrement-Operator existiert auch ein *Dekrement-Operator* (*decrement operator*) zur Verminderung um 1 : --

Operatoren kombiniert mit Zuweisungen

- Programme enthalten oft Zuweisungen der Form: $x = x + y$;
- dafür existiert eine kürzere Schreibweise
- ist nur Kurzform – kein schnelleres Programm, keine Erweiterungen

Kurzform	gleichwertig mit
$x \ += \ y;$	$x \ = \ x \ + \ y;$
$x \ -= \ y;$	$x \ = \ x \ - \ y;$
$x \ *= \ y;$	$x \ = \ x \ * \ y;$
$x \ /= \ y;$	$x \ = \ x \ / \ y;$
$x \ \% = \ y;$	$x \ = \ x \ \% \ y;$

Endlosschleifen

Nicht terminierende Schleifen werden *Endlosschleifen (infinite loops)* genannt.

Dafür gibt es 2 wesentliche Gründe

- das Programm soll immer laufen (z.B. Geldautomat, Alarmanlage, Steuerung einer Klimaanlage, ...)
- Fehler im Programm
 - Bedingung falsch, Rumpf nicht korrekt
 - Terminierung kann meist auch formal bewiesen werden (siehe später)

(Siehe auch Prog. Endlos)

```
while (true){  
    ...  
}
```

```
//terminiert nicht  
// fuer ungerade n  
while (n > 0){  
    n = n % 2;  
}
```

```
//terminiert immer  
while (n > 0){  
    n = n / 2;  
}
```


for-Schleife

Eine weitere Schleifenart ist die *for-Schleife* (*for-Anweisung*, *Zählschleife*, *Laufanweisung*, *for-statement*):

- nützlich, wenn die Wiederholung durch einen Zähler (*Laufvariable*) gesteuert wird
- Anzahl der Durchläufe steht meist im Vorhinein fest
- besteht aus: Initialisierungsteil, Schleifenbedingung, Inkrementierungsteil, Schleifenrumpf

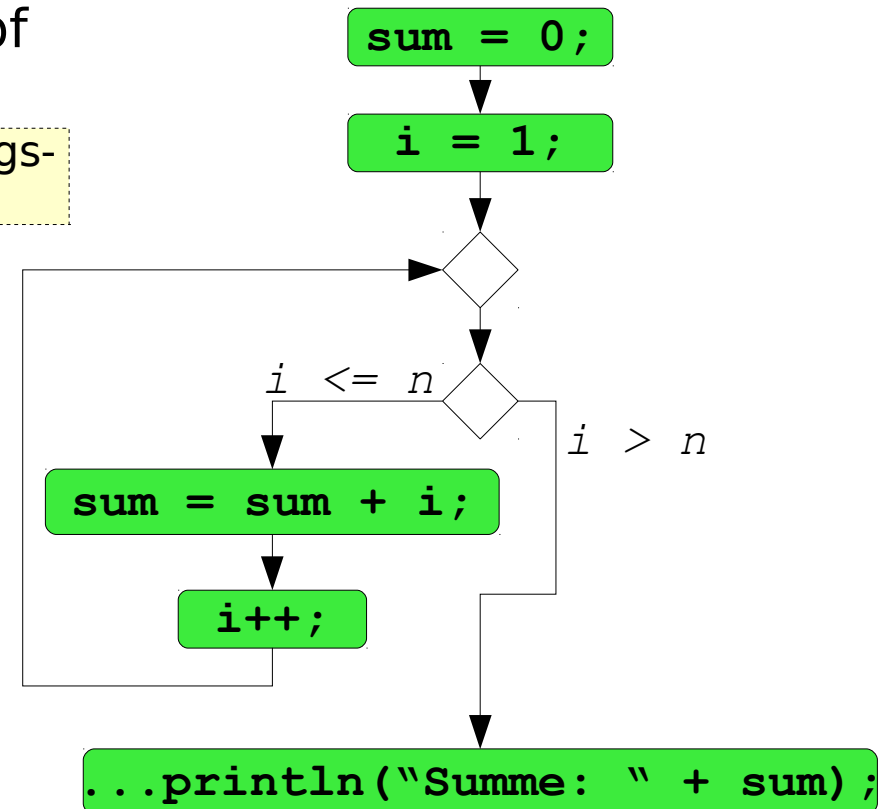
```
sum = 0;
for (i = 1; i <= n; i++) {
    sum = sum + i;
}
...println("Summe: " + sum);
```

Initialisierungsteil

Schleifenbedingung

Inkrementierungsteil

Schleifenrumpf



for-Schleife (Forts.)

allgemeine Form

```
for (initializing_action; boolean_expression, update_action)  
  body
```

res. Wort: for

Semantik:

- Initialisierungsteil (*initializing_action*): wird vor dem Betreten der Schleife ausgeführt - Laufvariable erhält einen Wert
- Schleifenbedingung (*boolean_expression*): wird jedesmal vor einem neuen Schleifendurchlauf geprüft
- Inkrementierungsteil (*update_action*): wird am Ende jedes Schleifendurchlaufs ausgeführt (z.B. Laufvariable erhöhen)
- Schleifenrumpf (*body*): wenn aus mehreren Anweisungen bestehend, muss geklammert werden {...}

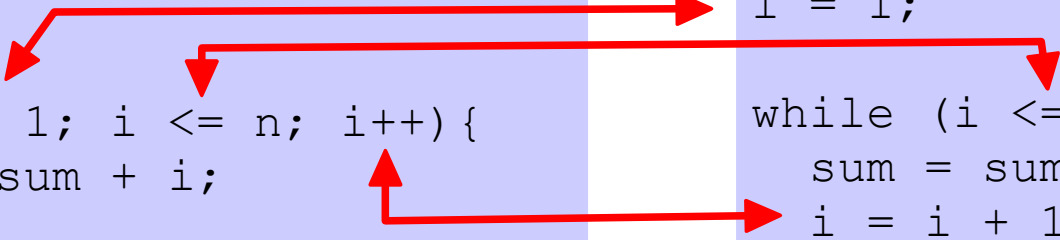
(Siehe Prog. `SummeNFor`)

for-Schleife (Forts.)

for-Schleife ist eigentlich eine kompakte Form einer while-Schleife

```
sum = 0;
for (i = 1; i <= n; i++) {
    sum = sum + i;
}
...println("Summe: " + sum);
```

```
sum = 0;
i = 1;
while (i <= n) {
    sum = sum + i;
    i = i + 1;
}
...println("Summe: " + sum);
```



- Initialisierungs- und Inkrementierungsteil können auch mehrere Anweisungen, getrennt durch Komma, enthalten (soll i.A. vermieden werden)
- Initialisierungsteil kann auch Variablendeklarationen enthalten

```
for (int i = 0, int j = 1; i < n && j > m; i++, j += 2) {
    ...
}
```

for-Schleife (Forts.)

- enthält der Initialisierungsteil eine Variablendeklaration, so ist diese Variable nur innerhalb des Schleifenrumpfs gültig!
 - ist hauptsächlich sinnvoll, wenn eine Variable nur in der Schleife verwendet wird (z.B. als Zähler)
- Vorsicht bei folgenden Schleifenvarianten:
 - `for (...);` → Schleife ohne Anweisungen im Rumpf!
 - `for (;;) -` alle Teile sind „leer“ (d.h. keine Bedingungen, kein Update) → Endlosschleife!

```
int n = 3;

for (int i = 1; i <= n; i++){
    ...println(i);
}
...println("danach - i: " + i);
//Compilerfehler!
```

```
int k, sum = 0;
for (k = 1; k < n; k++);
    sum = sum + k;
//sum: 3
```

```
for (;;)
    ...println("innerhalb");
```

Geschachtelte Schleifen

Beispiel: Zeichnen einer symbolischen Treppe als Programmausgabe mittels einem bestimmten Zeichen – dh. in der 1. Zeile ein Zeichen, 2. Zeile zwei Zeichen, ...; Die Höhe der Treppe (Anzahl der Zeilen) und das zu verwendende Zeichen ist einzugeben.

Algorithmus: zeichne in der i-ten Zeile i Zeichen, solange bis die Höhe erreicht ist.

Dies erfordert

- eine Schleife für die Anzahl der Zeilen (entspricht der Höhe)
- eine Schleife pro Zeile für die Anzahl der Zeichen

→ dafür nötig: *geschachtelte Schleifen (nested loops)*

- der Schleifenrumpf enthält wieder eine Schleife (Schleifenrumpf kann alle Arten von Anweisungen enthalten)
- innere Schleife muss komplett innerhalb des Schleifenrumpfs der äußeren liegen
- für alle Schleifenarten möglich (while, do-while, for)

(Siehe Prog. `Zeichentreppe`)

Abbruchanweisung: *break*

In manchen Fällen soll eine Schleife mitten im Schleifenrumpf abgebrochen werden – es tritt eine Bedingung ein, welche eine weitere Ausführung der Schleife zwecklos macht.

→ `break`; beendet die Schleife mitten im Rumpf - d.h. die Programmausführung setzt nach der entsprechenden Schleife fort.

- `break` bei allen Schleifenarten einsetzbar
- soll möglichst vermieden werden, da schwerer zu verifizieren: wenn mehrere Abbruchbedingungen existieren, ist Zustand am Ende nicht bekannt!

(Siehe Prog. `SchleifenAbbruch`)

```
sum = 0;
wert = ...readLineInt();
while (wert >= 0){
    sum = sum + wert;
    if (sum > MAX){
        System.out.print("Max.!");
        break;
    }
    wert = ...readLineInt();
}
```

Abbruchanweisung: *break* (Forts.)

- `break` ist ersetzbar – meist mittels `while`-Schleife, jedoch unter Umständen etwas kompliziert

```
sum = 0;
wert = ...readLineInt();
while (wert >= 0 && sum <= MAX) {
    sum = sum + wert;
    if (sum <= MAX) {
        wert = ...readLineInt();
    }
}
if (sum > MAX)
    ...print("Max.!");
```

Hinweis: es existiert in Java auch die Anweisung `continue`

→ beendet die aktuelle Iteration und springt zum Beginn des nächsten Schleifendurchlaufs
(soll möglichst nicht verwendet werden)

Kapitel 6

Arrays

Arrays: allgemeine Idee

Problem: Für jeden Tag einer Woche soll die Temperatur zu Mittag notiert werden und der Durchschnitt davon berechnet werden. Weiters ist anzugeben, wieviele Einzelwerte über bzw. unter dem durchschnittlichen Wert liegen.

Lösungsidee:

- (1) 7 Werte einlesen, Summe bilden, Summe durch 7 teilen ergibt Durchschnitt
- (2) jeden der 7 Werte mit Durchschnitt vergleichen

→ alle 7 Werte sind zu speichern (einlesen und summieren allein reicht nicht aus) – 7 verschiedene Variablen nötig!

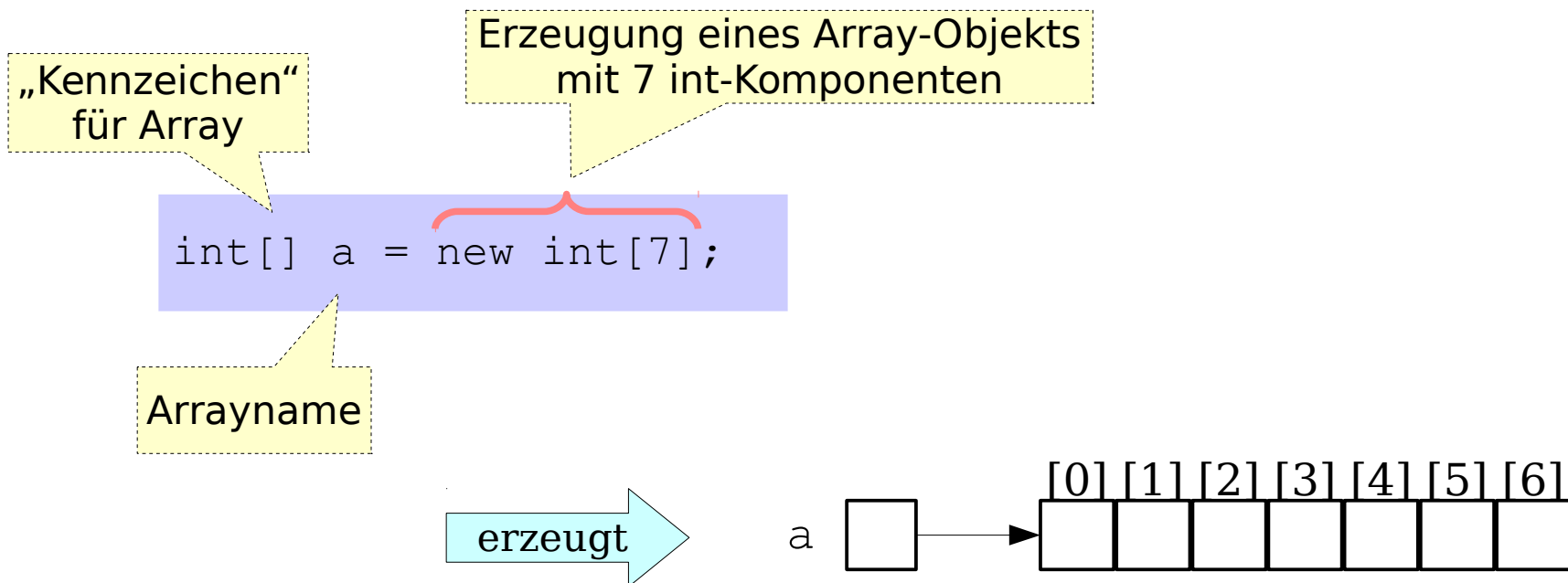
Praktisch nicht durchführbar, insbesondere bei größerer Anzahl von Werten (z.B. 365 für ein Jahr, > 12000 für alle Studenten der Universität Salzburg, ...).

Arrays: allgemeine Idee (Forts.)

Sehr viele Lösungen erfordern das Arbeiten mit vielen Werten (Listen oder Tabellen von Werten) – dafür können *Arrays* (*Reihungen, Felder*) verwendet werden.

Ein **Array** ist eine „Tabelle“ von Elementen eines bestimmten Typs, wobei einzelne Elemente mittels ganzzahliger Indizes angesprochen werden.

Deklaration und Erzeugung eines Arrays:



Arrays: Deklaration, Erzeugung

res. Wort: new

allgemeine Form

```
base_type[] array_name = new base_type[length];
```

- Deklaration (auch ohne Erzeugung möglich)
 - deklariert Array mit Namen `a`
 - Elemente sind vom typ `int`
 - Länge (Anzahl der Elemente) ist noch nicht festgelegt
- Erzeugung
 - erzeugt ein neues `int`-Array mit 7 Elementen
 - weist dessen Adresse der Variablen `a` zu
 - Elemente ansprechbar über Indizes, z.B. `a[3]`
 - Indizes beginnen immer bei 0 - d.h. Indizes der Elemente von `a` sind 0, 1, 2, 3, 4, 5, 6
 - Erzeugung irgendwann nach Deklaration möglich, jedoch nicht vorher!

```
int[] a;
```

```
a = new int[7];
```

Arrays: Zugriff auf Elemente

Für Java gilt:

- Arrays sind spezielle Objekte!
- Array-Variable enthalten Zeiger (Referenzen) auf Array-Objekte
- Länge eines Arrayobjekts kann nicht verändert werden

Zugriff auf Array-Elemente

- einzelne Elemente werden wie Variablen behandelt (lesend und schreibend darauf zugreifbar)
- Index muss ein ganzzahliger Ausdruck sein
- Laufzeitfehler, wenn Index < 0 oder Index \geq Arraylänge ist!
- Länge eines Arrays mit dem Attribut `length` abfragbar: `a.length` liefert die Anzahl der Elemente von `a`

```
int i = 2;  
a[5] = 815;  
a[i+1] = a[i] * 2;  
a[2*i-1] = 3 / a[i];
```

```
i = 6;  
a[i] = a[i+1] / 2;  
//Laufzeitfehler!
```

```
int anz = a.length;  
//anz: 7
```

Arrays: erste Beispiele

- alle Arrayelemente einlesen
- alle Arrayelemente aufsummieren
- Arraylänge einlesen, Array erzeugen, alle Elemente mit einem Wert vorbelegen

```
for(int i = 0; i < a.length; i++)  
    a[i] = ...readInt();
```

```
int sum = 0;  
for(int i = 0; i < a.length; i++)  
    sum = sum + a[i];
```

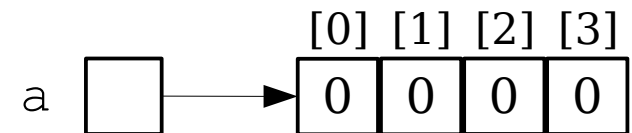
```
double[] d;  
double init;  int n;  
  
n = ...readLineInt();  
init = ...readLineDouble();  
  
d = new double[n];  
for(int i = 0; i < d.length; i++)  
    d[i] = init;
```

(siehe Prog. Temperatur)

Array: verschiedene Zuweisungen

- Arrayelemente von numerischen Typen werden in Java mit 0 vorbelegt
- es wird empfohlen, immer nur auf jene Elemente zuzugreifen, die im Programm auch Werte erhalten haben

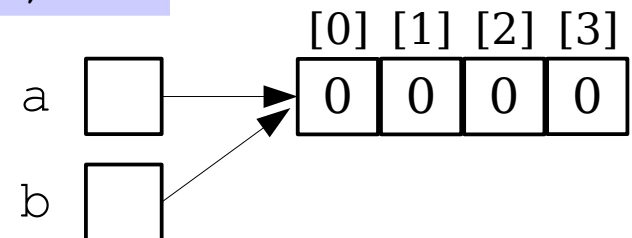
```
int[] a = new int[4];
```



- Array-Zuweisung in Java: es werden Zeigerwerte zugewiesen!

→ `b` „zeigt“ auf selbes Arrayobjekt wie `a`

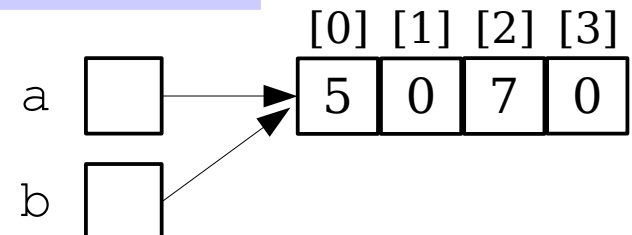
```
int[] b;  
b = a;
```



- Elemente können mittels `a` und `b` verändert werden

- `b[2]` liefert 7 - wie `a[2]`
`a[0]` liefert 5 - wie `b[0]`

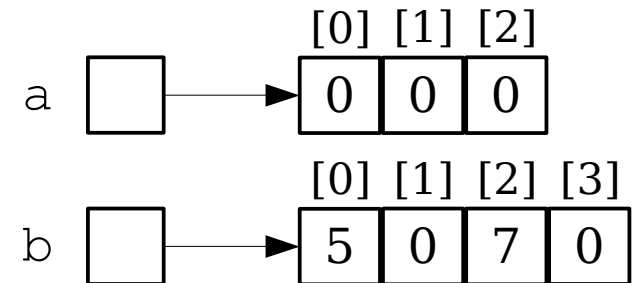
```
a[2] = 7;  
b[0] = 5;
```



Array: verschiedene Zuweisungen (Forts.)

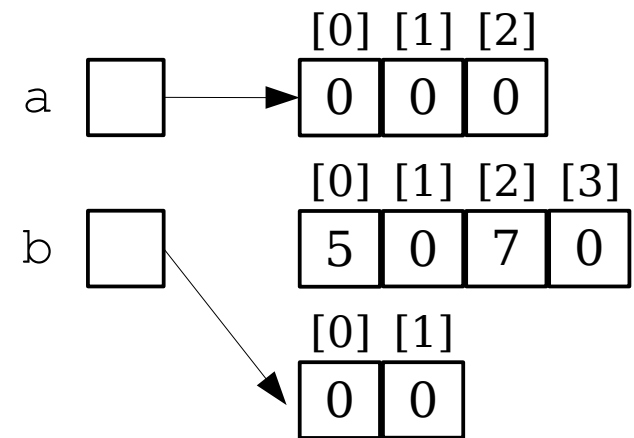
```
a = new int[3];
```

- neues Arrayobjekt wird erzeugt
- a zeigt auf dieses neue Arrayobjekt



- neues Arrayobjekt wird erzeugt, `b` zeigt darauf
- „altes“ Arrayobjekt nicht mehr greifbar!

```
b = new int[2];
```



Garbage Collection (automatische Speicherbereinigung) in Java: nicht mehr referenzierte Objekte werden automatisch freigegeben, ihr Speicherplatz steht für neue Objekte zur Verfügung.

Arrays: Initialisierung

Folgende Möglichkeiten für direkte Initialisierung von Arrays sind in Java vorhanden:

- Deklaration einer Arrayvariablen und Initialisierung
 - Bsp. erzeugt neues int-Array mit 5 Elementen, diese werden mit den angegebenen Werten initialisiert
- Erzeugung eines Arrayobjekts und Initialisierung
 - Bsp. erzeugt neues int-Arrayobjekt mit 4 Elementen, diese werden mit den angegebenen Werten initialisiert

```
int[] g = {2, 4, 6, 8, 10};
```

```
int[] u;  
u = new int[]{1, 3, 5, 7};
```


Arrays: Beispiele

Arrays sind viel genutzte Datenstrukturen, welche für viele Probleme verwendbar sind.

- Tage pro Monat mit Array (effizienter als mit switch-Anweisung)
 - Tabelle (als Array) für alle 12 Monate mit jeweiliger Anzahl der Tage (siehe Prog. `TageProMonatArray`)

```
int[] tageProMonat = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30,  
                     31, 30, 31};
```

- Array kopieren, d.h. ein weiteres Arrayobjekt mit den selben Werten
 - der Inhalt von jedem Element ist zu kopieren

Hinweis: auch mittels
`System.arraycopy(...)`
(siehe Literatur)

```
int[] a = {2, 3, 5, 7, 11};  
int[] kopie = new int[a.length];  
for(int i = 0; i < a.length; i++)  
    kopie[i] = a[i];
```

Arrays: Beispiele (Forts.)

- maximalen Wert in einem Array bestimmen
 - alle Elemente müssen geprüft werden
 - als erstes Maximum beliebiger Wert des Arrays möglich – praktisch sinnvoll ist jener mit Index 0

```
int[] a = ...;  
int max = a[0];  
for(int i = 1; i < a.length; i++)  
    if (a[i] > max) max = a[i];
```

- bestimmten Wert in einem Array suchen
 - mit letzter Position beginnen
 - wenn Wert gefunden, Suche abbrechen und Position merken
 - wenn Position < 0 erreicht, dann Wert nicht gefunden!

```
int gesucht = ...;  
int pos = a.length - 1;  
while (pos >= 0 && a[pos] != gesucht)  
    pos--;  
//nach Schleife: pos == -1 => nicht gefunden  
//                sonst gefunden an Position pos
```

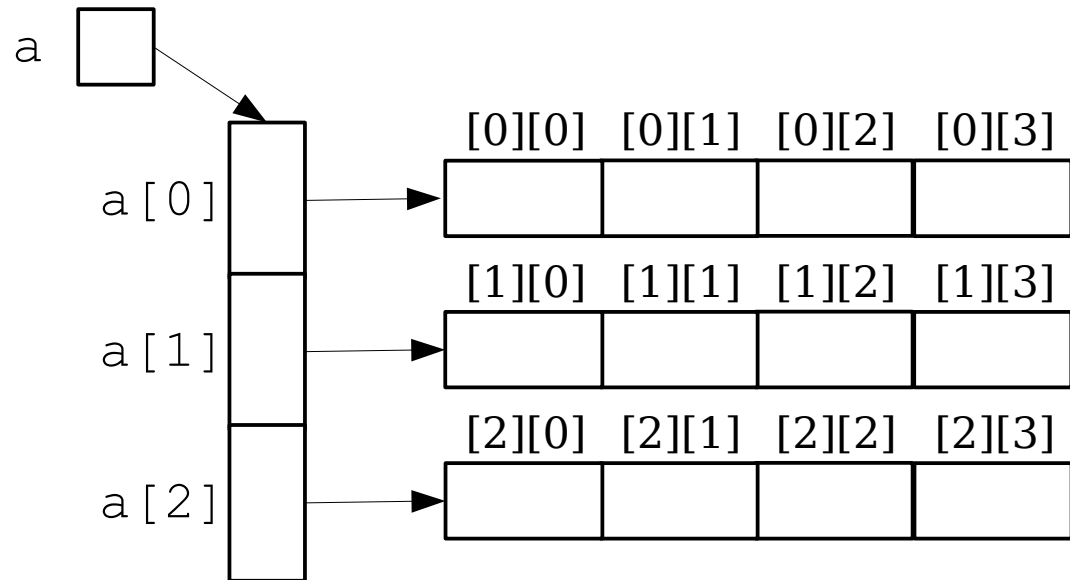
(siehe Prog. TypischArray)

Mehrdimensionale Arrays

Die Arrayelemente können wieder Arrays sein, d.h. ein Array von Arrays (oder zweidimensionale Tabelle bzw. Matrix) – ein *zweidimensionales Array*.

Analog zu einer Matrix kann auch von Zeilen und Spalten gesprochen werden: 2-dimensionales Array ist ein Array aus Zeilen, jede Zeile hat Spaltenelemente.

```
//Deklaration  
int[][] a;  
//Erzeugung  
a = new int[3][4];  
  
//Elementzugriff  
m = a[2][1];  
a[i][k] = k + i;
```



Mehrdimensionale Arrays (Forts.)

- zum Durchlauf werden geschachtelte Schleifen benötigt
 - z.B. jedem Element die Summe seiner Indizes zuweisen

```
for (int zeile = 0; zeile < a.length; zeile++)  
    for (int spalte = 0; spalte < a[zeile].length; spalte++)  
        a[zeile][spalte] = zeile + spalte;
```

(Siehe Prog. `ZweiDimArray`)

- Zeilen können unterschiedliche Längen aufweisen
- auch mehr als 2-dimensionale Arrays möglich: Deklaration, Erzeugung, Zugriff analog

```
int[][] v = new int[2][];  
v[0] = new int[5];  
v[1] = new int[3];
```

```
int[][][] h = new int[3][2][5];
```

for-each-Schleife

Oft ist das gesamte Array zu durchlaufen, um alle Werte für eine weitere Verarbeitung zu lesen. Java bietet dazu eine kompakte Form der for-Schleife, die sogenannte *for-each-Schleife (enhanced for)* an:

- erlaubt sequenziellen Durchlauf aller Elemente, beginnend bei erstem
- Zugriff auf Elemente nur lesend – d.h. Array wird nicht verändert!
- kann nur ein Array durchlaufen (nicht zwei Arrays parallel!)

- bei jedem Schleifendurchlauf wird der Variablen `e1` der Wert des nächsten Arrayelements zugewiesen
- Typ der `e1` Variablen muss dem Elementtyp des Arrays entsprechen

(Siehe Prog. `ForEach-Demo`)

```
int[] a = ...;
for(int e1: a)
    System.out.print(e1 + " ");
```

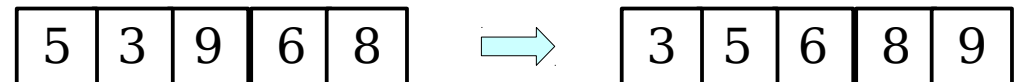
Beispiel: Sortieren

Problem: eine Menge von Werten ist nach einem bestimmten Kriterium zu sortieren – klassische Aufgabe der Informatik!

Beschränkung hier: `int`-Werte aufsteigend sortieren.

Spezifikation etwas genauer: ein `int`-Array `werte` mit beliebigen Werten ist aufsteigend zu sortieren, d.h. die Elemente sind so umzustellen, dass

für alle `i` mit $0 \leq i < \text{werte.length} - 1$ gilt
 $\text{werte}[i] \leq \text{werte}[i+1]$

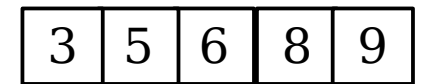
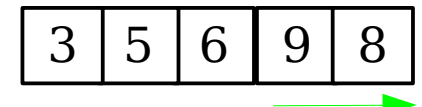
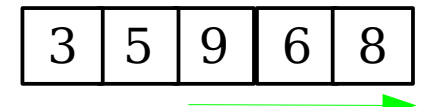
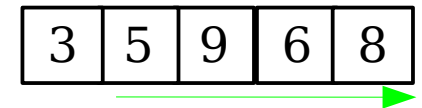
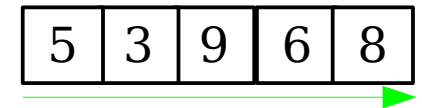


- Mehrfachvorkommen von Werten ist erlaubt
- Ausgangsarray muss nicht erhalten werden – d.h. Sortierung wird durch Vertauschung von Elementen erreicht (kein zusätzliches Array nötig)

Beispiel: Sortieren (Forts.)

Idee für Algorithmus:

1. suche im gesamten Array das Minimum und tausche es an Position 0
2. suche Minimum ab Position 1 und tausche es an Position 1
3. wiederhole dies mit allen weiteren Positionen bis Arrayende erreicht ist



Daraus folgt Programmidee (in Pseudocode):

```
for(int i = 0; i < werte.length - 1; i++){  
    finde Minimum von werte[i] ... werte[length-1];  
    tausche Minimum nach Position i;  
}
```

(siehe Prog. MinSort)

Weitere Überlegung: wieviele Vergleiche sind nötig und wie steigt deren Anzahl im Vergleich zur Arraylänge?

Exkurs: Parameter in der Kommandozeile

Wie bekannt enthält die main-Methode in Javaprogrammen einen Parameter in der Form `String[] args`

- ist ein Array aus String-Elemente mit Namen `args`
- wird implizit erzeugt
- erhält jene Werte, die bei Programmstart in der Kommandozeile angegeben werden (nach dem Programmnamen)

z.B. `java KommandoZeile Wir 10`

ergibt `args[0]: "Wir"`

`args[1]: "10"`

- primitive Form der Eingabe – keine Information oder Aufforderung an die Benützer
- sinnvoll für „technische“ Zwecke (z.B. Name einer Datei, die vom Programm bearbeitet werden soll)

(siehe Prog. `KommandoZeile`)

```
public static void main (String[] args){  
    String name = args[0];  
    int n = Integer.parseInt(args[1]);  
}
```


Beispiel: Binäre Suche

Problem: bei großen Datenmengen wird lineare Suche (siehe „bestimmten Wert in einem Array suchen“) sehr aufwändig! Ein Ergebnis wird rasch erreicht, wenn die zu durchsuchenden Daten geordnet sind → Suchbereich kann schnell verkleinert werden (z.B. Lexikon, Telefonbuch)!

Spezifikation etwas genauer: suche bestimmtes Element im Array `werte` (aufsteigend sortiert, Mehrfachvorkommen erlaubt). Bei Erfolg Position angeben, sonst Meldung über Misserfolg.

Idee für Bereichsverkleinerung:

- markiere aktuellen Suchbereich mit Position `links` und `rechts`
 - berechne davon die mittlere Position
$$\text{mitte} = (\text{links} + \text{rechts}) / 2$$
 - vergleiche `gesucht` mit `werte[mitte]` und verkleinere Suchbereich
 - wenn `gesucht < werte[mitte]` suche in linker Hälfte weiter
 - wenn `gesucht > werte[mitte]` suche in rechter Hälfte weiter
- solange bis gefunden oder Suchbereich erschöpft

Beispiel: Binäre Suche (Forts.)

Programmidee (in Pseudocode):

```
links = 0; rechts = werte.length - 1;
while(links <= rechts
    && "nicht gefunden"){
    mitte = (links + rechts) / 2;
    if (gesucht < werte[mitte])
        rechts = mitte - 1;
    else if (gesucht > werte[mitte])
        links = mitte + 1;
    else
        "gefunden"
}
```

Zu beachten:

- initialer Suchbereich ist gesamtes Array
- nach Verlassen der Schleife
 - gefunden auf Position `mitte` oder
 - Suchbereich erschöpft → gesuchtes Element nicht gefunden

(siehe Prog. `BinaereSuche`)

gesucht wird: 11

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
3	4	5	5	8	11	14	15	15	18



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
3	4	5	5	8	11	14	15	15	18



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
3	4	5	5	8	11	14	15	15	18



Ergänzung: typische Fehler bei Schleifen

- Endlosschleife für bestimmte Werte (siehe auch „Endlosschleifen“)
 - terminiert meist für viele bzw. typische Werte
 - keine Terminierung bei „seltenen“ oder untypischen Werten

```
anz = 0;
while(guthaben <= 0){
    guthaben = guthaben - gebuehr;
    guthaben = guthaben + rate;
    anz++;
}
...println(anz + "Monate bis positiv");
```

Endlosschleife bei
zu geringer `rate`

besser: dies ausschließen

```
if(rate <= gebuehr)
    ...println("Zahlung zu gering!");
else{
    anz = 0;
    while(guthaben <= 0){
        ....
    }
    ...println(anz + "Monate bis positiv");
}
```

Ergänzung: typische Fehler bei Schleifen (Forts.)

- ein Schleifendurchlauf zu viel oder zu wenig (off-by-one error)
 - meist wenn Schleifenbedingung schlampig formuliert (z.B. $<$ statt \leq)
 - Test auf Gleichheit $==$ (bzw. Ungleichheit $!=$): geeignet für ganzzahlige-Typen (und `char`) – jedoch nicht für floating-point Typen (hier besser: Test auf Bereich)

```
//binaere Suche
links = 0;
rechts = werte.length - 1;
        //richtig: <=
while(links < rechts
        && "nicht gefunden"){
    .....
}
```

Fehlersuche mittels Protokollierung von Variablenwerten (Tracing)

Ausgabe von passenden Werten an „Schlüsselstellen“ (z.B. Werte von Variablen der Schleifenbedingung) für jeden Schleifendurchlauf und danach.

Auch nach erfolgreicher Fehlersuche – testen nötig!

```
anz = 0;
while(guthaben <= 0){
    guthaben = guthaben - gebuehr;
    guthaben = guthaben + rate;
    anz++;
    ...println("..." + guthaben);
}
.....
```

Exkurs: Wertebereiche von Zahltypen

Wie bereits in verschiedenen Beispielen gesehen, zeigt Java bei *Überschreitung von Wertebereichen (Overflow)* für numerische Typen folgendes Verhalten:

- ganzzahlige Typen (`int`, `long`, ...): Fortsetzung am „anderen“ Ende des Wertebereichs - „wrap around“
- Gleitkomma-Typen: der Wert `Infinity` (oder `-Infinity`) wird erzeugt

Weiters zu beachten ist Division mit 0

- Gleitkomma-Typen: `NaN` (Not-a-Number) als spezieller Wert wird verwendet
- ganzzahlige Typen: Laufzeitfehler (`ArithmeticException`) wird ausgelöst

(Bsp. siehe Prog. `Overflow` , Details siehe Literatur)

Kapitel 7

Klassen, Objekte, Methoden

Motivation

Arrays gruppieren Datenelemente vom selben Typ zu einer größeren Einheit. Ist dies auch für konzeptionell zusammengehörige Datenelemente (ev. mit verschiedenen Typen) möglich?

Z.B. Datum

```
int day;  
String month;  
int year;
```

Bruch

```
int num;  
int den;
```

Konto

```
double balance;  
String name;
```

Problem dabei: wenn etwa n Datumsexemplare benötigt werden, sind $n*3$ Variablen zu deklarieren (alternativ 3 Arrays mit jeweils n Komponenten!).

- viel Schreibaufwand, viele Bezeichner nötig, unpraktisch bis nicht praktikabel

Klasse – erste Deklaration

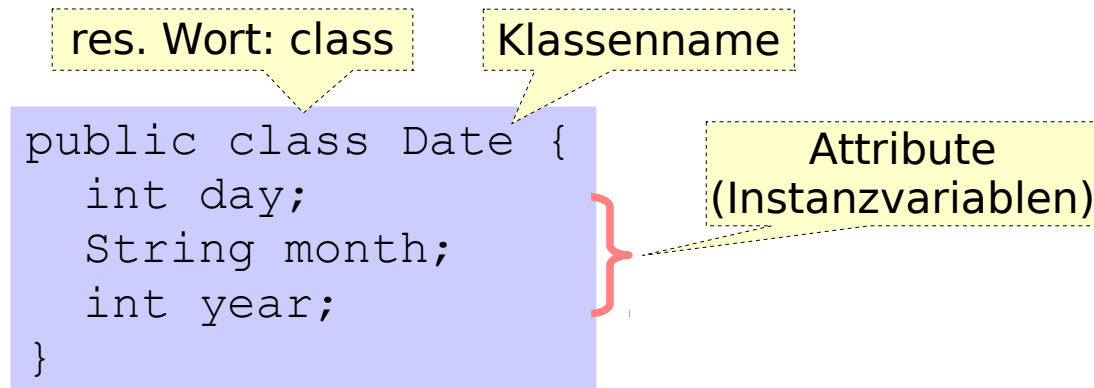
Klassen (classes) können mehrere Variablen (auch von verschiedenen Typen) zu einem neuen Typ zusammenfassen.

Dann können (beliebig viele) Variablen und Arrays dieses „neuen“ Typs deklariert werden.

res. Wort: class Klassenname

```
public class Date {  
    int day;  
    String month;  
    int year;  
}
```

Attribute
(Instanzvariablen)



```
public class Fraction {  
    int num;    //Zaehler  
    int den;    //Nenner  
}
```

```
public class Account {  
    double balance;  
    String name;  
}
```

- reserviertes Wort `class` gefolgt vom Namen der Klasse
- gesamte Definition der Klasse in `{...}`
- Variablen der Klasse heißen **Attribute** oder **Instanzvariablen** (*instance variables, fields*)

Verwendung von Klassen und Objekten

- Klassendeklaration in Java
 - in eigenen Dateien – z.B. `Date.java`, `Account.java`, `Fraction.java` (Klassenname mit Erweiterung `.java`)
 - (wird in dieser LV meistens angewendet)
 - Kompilierung nicht vergessen!
 - gemeinsam mit anderen Klassen in einer Datei

- Variablen von Klassentypen werden wie bisher deklariert

- Datei mit entsprechender Klasse muss im selben Ordner (Verzeichnis) liegen

```
Date today, birthday;  
Account myAccount;  
Fraction x, y;
```

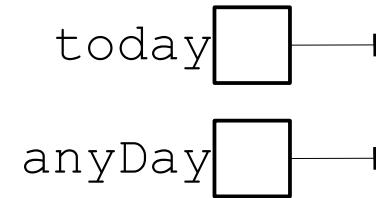
- Variablen von Klassentypen sind Zeiger (Referenzen, Pointer) – eine Deklaration erzeugt noch keine „Werte“ (genauer: keine Objekte)!
- Objekte einer Klasse müssen explizit mittels `new` erzeugt werden
 - Objekte werden als **Instanzen** einer Klasse bezeichnet
 - eine Klasse ist wie eine Schablone - beliebig viele Objekte sind erzeugbar

```
today = new Date();  
myAccount = new Account();  
x = new Fraction();  
y = new Fraction();
```

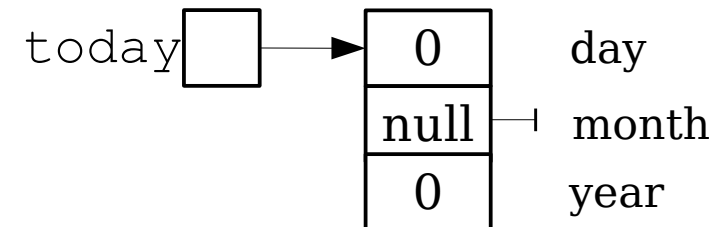
Erzeugung von Objekten genauer

- Variablendeklaration
 - Objektvariablen werden angelegt
 - zeigen auf kein Objekt – Wert `null` („Zeiger ins Leere“)
- Objekterzeugung
 - mittels `new`, Name der Klasse und `()`
 - `new` erzeugt ein Objekt und liefert Zeiger auf dieses
 - Zeiger auf das Objekt an Variable zuweisen (sonst nicht auffindbar!)
 - kombinierbar mit Variablendeklaration
- Zugriff auf Instanzvariable mittels „Dot-Notation“
 - Variable.Instanzvariable
 - lesend und schreibend

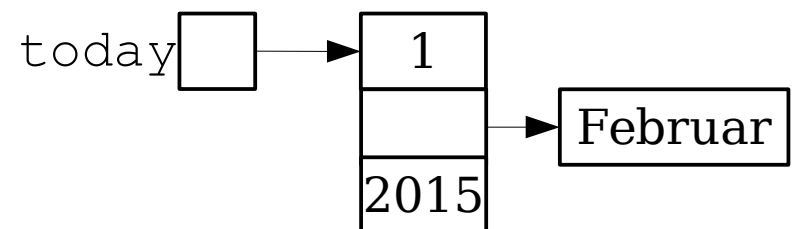
```
Date today, anyDay;
```



```
today = new Date();
```



```
today.day = 1;  
today.month = "Februar";  
today.year = 2015;
```

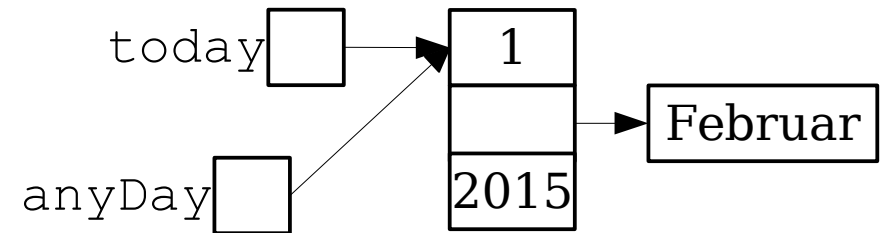


Erzeugung von Objekten genauer (Forts.)

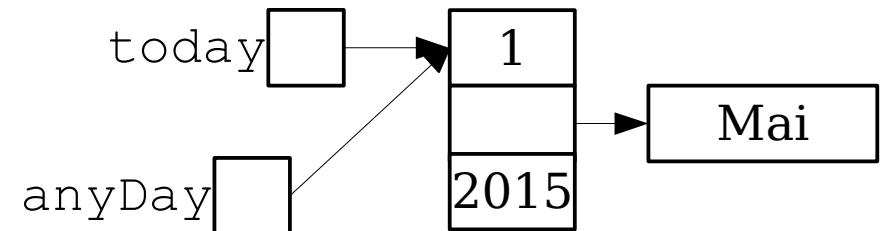
- Zuweisung zwischen Objektvariablen
 - möglich, wenn zur selben Klasse (Typ) gehörend (vorerst)
 - Zeigerwert wird zugewiesen!
- Achtung: Änderung von Werten der Instanzvariablen über alle Objektvariablen, die auf das selbe Objekt zeigen, möglich
 - `today.month` liefert "Mai" und `anyDay.month` liefert "Mai"

(siehe Prog. DateDemo)

```
anyDay = today;
```



```
anyDay.month = "Mai";
```



Methoden: Motivation

Beispiel Konto: etwa ein Girokonto bei einer Bank sollte in einfacher Version folgende Daten aufweisen:

- aktueller Kontostand
- Bezeichnung bzw. Name

```
public class Account {  
    double balance;  
    String name;  
}
```

→ Klasse `Account` mit entsprechenden Instanzvariablen

Jedoch: der Kontostand eines bestimmten Kontos wird sich laufend ändern (einzahlen, abheben)

- d.h. Daten haben meist auch Operationen, die auf sie zugreifen und Änderungen vornehmen!

Daten und „ihre“ Operationen gehören zusammen – sie sollen in einem gemeinsamen Sprachkonstrukt definiert werden

→ **Klassen (classes)**

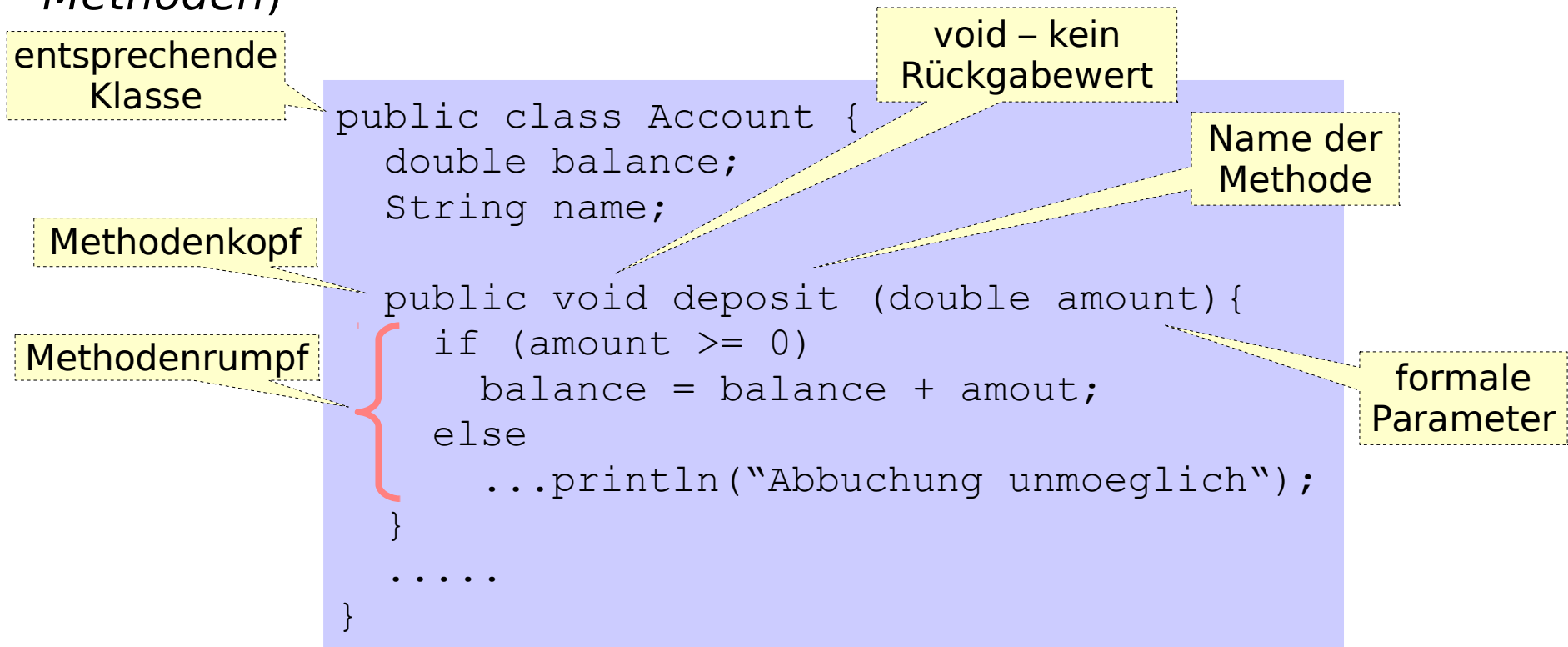
Klassen sind die Bausteine *objektorientierter Programmierung*, einer Standardtechnik der Software-Entwicklung.

Methoden: Definition

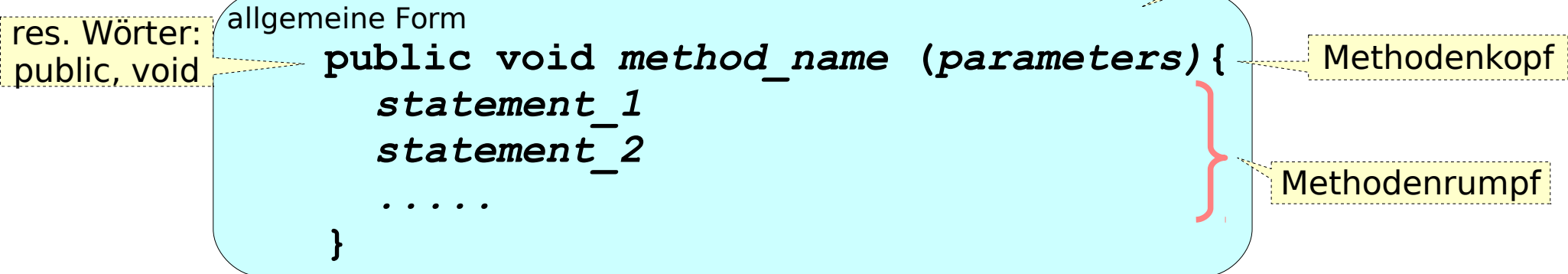
Operationen werden meist (auch in Java) **Methoden (methods)** genannt.

Eine **Klasse** definiert **Daten** und ihre dazugehörigen **Methoden**!

Definition von Methoden **ohne** Rückgabewert (*Prozeduren, void-Methoden*)



Methoden: Definition (Forts.)



Syntax für Methoden **ohne** Rückgabewert (*void-methods*)

- jede Methode gehört zu einer Klasse (Definition innerhalb der Klassendefinition)
- res. Wort `public`: Zugriff von außerhalb der Klasse erlaubt (andere Varianten später)
- res. Wort `void`: kein Rückgabewert
- Methodenname ist ein gültiger Java-Bezeichner (soll mit Kleinbuchstaben beginnen; erster Teil soll Verb sein)
- (*formale*) *Parameter* dienen der Übergabe von Werten an die Methode (genauer etwas später)
 - wenn kein Parameter vorgesehen ist, dann leere Klammern ()

Methoden: Definition (Forts.)

- *Methodenkopf (method header)* erstreckt sich von `public` bis inkl. Parameter
- *Methodenrumpf (method body)* besteht aus beliebig vielen Anweisungen, die die Funktion der Methode bestimmen

Definition von Methoden **mit** Rückgabewert (*Funktionen*): bis auf folgende Änderungen wie zuvor

```
public class Account {  
    .....  
  
    public double getBalance () {  
        return balance;  
    }  
    .....  
}
```

Typ des
Rückgabewerts

return-Anweisung

Methoden: Definition (Forts.)

Syntax für Methoden **mit** Rückgabewert

- Typ des Rückgabewerts anstatt `void` im Methodenkopf
- der Methodenrumpf muss mindestens eine `return`-Anweisung enthalten
- sonst wie Methoden ohne Rückgabewert

Für beide Methodenvarianten gilt, dass Anweisungen im Methodenrumpf

- auf Instanzvariablen der Klasse zugreifen können;
- andere Methoden der selben Klasse aufrufen können;
- die Parameter der Methode wie Variablen verwenden.

Eine Klasse kann beliebig viele Methodendefinitionen enthalten.

Aufruf von Methoden

Ein **Methodenaufruf** (**method invocation, method call**) erfolgt meist von einer anderen Klasse (als jener, in welcher die Methode definiert wird).

Um eine Methode einer Klasse aufrufen zu können

- muss ein Objekt dieser Klasse vorhanden sein (mittels `new` erzeugt) und
- auf dieses Objekt über eine Objektvariable zugegriffen werden können.
(Weitere Möglichkeiten später.)

Der Aufruf erfolgt mittels „Dot-Notation“:

`variable_name.method_name(...);`

Name der
Objektvariablen

```
public class AccountDemo {  
    Account myAccount = new Account();  
    double howmuch;  
    .....  
    myAccount.deposit(howmuch);  
    .....  
}
```

Methodenname

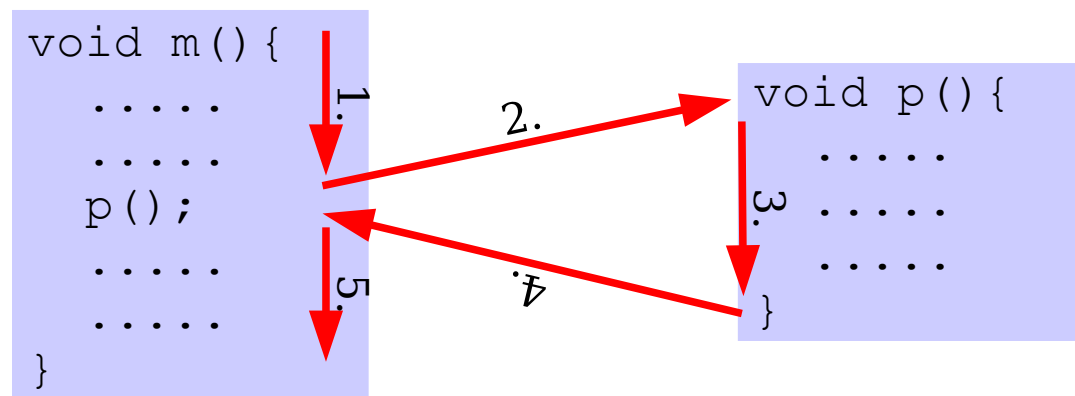
zu übergebender
Parameter (aktueller
Parameter)

Aufruf von Methoden: Wirkung

Allgemeine Semantik eines Methodenaufrufs (schematisch mittels der Methoden $m()$, $p()$)

1. Abarbeitung von m erreicht Aufruf von p
2. Verzweigung zur ersten Anweisung von p
3. Abarbeitung von p
4. nach Beendigung von p erfolgt Rückkehr nach m - und zwar an jene Stelle wo p aufgerufen wurde
5. Fortsetzung von m mit der unmittelbar nächsten Anweisung nach dem Aufruf von p .

Eine aufgerufene Methode kann weitere (andere) Methoden aufrufen!

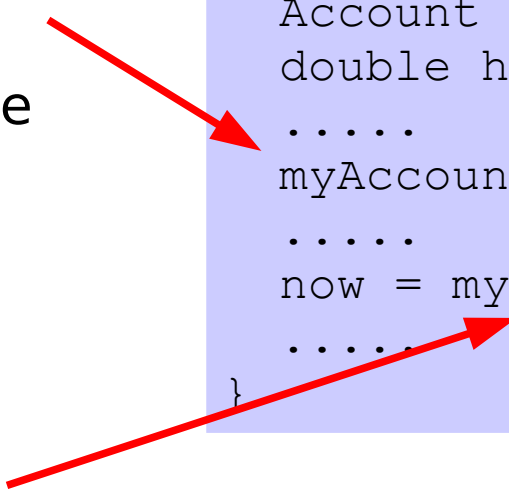


Aufruf von Methoden: 2 Arten

Methodenaufrufe (mittels Dot-Notation) können abhängig von der Art der Methode folgende Gestalt haben:

- Methoden ohne Rückgabewert: Methodenaufruf mit ; abgeschlossen ergibt eine Anweisung

```
public class AccountDemo {  
    Account myAccount = new Account();  
    double howmuch, now;  
    .....  
    myAccount.deposit(howmuch);  
    .....  
    now = myAccount.getBalance();  
    .....  
}
```



- Methoden mit Rückgabewert: überall einsetzbar, wo ein Wert des Typs des Rückgabewerts (return-Typ) verwendet werden darf (z.B. in arithmetischen Ausdrücken, Ausgabeanweisungen, ...)

return-Anweisung

- return-Anweisung in Methoden mit Rückgabewert
 - bestimmt den Rückgabewert der Methode und beendet den Methodenaufruf
 - jede Methoden mit Rückgabewert *muss* mit einer return-Anweisung der Form

`return expression;`

beendet werden

- reserviertes Wort `return`
- Ergebnistyp von *expression* muss zuweisungskompatibel zu return-Typ sein
- in einer Methode können mehrere return-Anweisungen auftreten

```
public class Account {  
    double balance = 0;  
    .....  
    public double getBalance () {  
        return balance;  
    }  
    .....  
}
```

```
public class Foo {  
    public int max (int x, int y) {  
        if (x > y)  
            return x;  
        else  
            return y;  
    }  
    .....  
}
```

return-Anweisung (Forts.)

- return-Anweisung in Methoden ohne Rückgabewert:
 - ist nicht zwingend erforderlich
 - beendet den Methodenaufruf (also vor Erreichen des Endes des Methodenrumpfs)
 - bestimmt keinen Rückgabewert und hat daher die Form
`return;`

Zusammengefasst also:


- Methoden mit Rückgabewert (Funktionen) müssen mit einer return-Anweisung beendet werden.
- Methoden ohne Rückgabewert (Prozeduren) können mit einer return-Anweisung beendet werden.

Parameter für Methoden

Mittels **Parametern (parameters)** werden einer Methode vom aufrufenden Programmteil Werte übergeben.

- **formale** Parameter

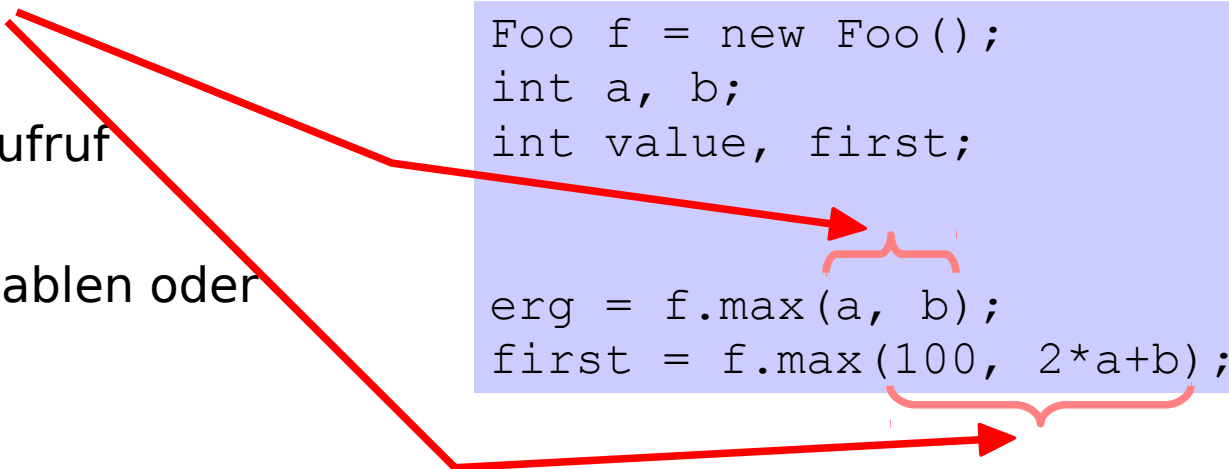
- werden im Methodenkopf definiert
- sind Variablen für die Methode



```
public class Foo {  
    public int max (int x, int y) {  
        if (x > y)  
            return x;  
        else  
            return y;  
    }  
}
```

- **aktuelle** Parameter
(Argumente)

- werden beim Methodenaufruf festgelegt
- sind spezielle Werte, Variablen oder Ausdrücke



```
Foo f = new Foo();  
int a, b;  
int value, first;  
  
erg = f.max(a, b);  
first = f.max(100, 2*a+b);
```

Parameterübergabe

Parameterübergabe bedeutet: aktuelle Parameter werden den formalen Parametern zugewiesen.

z.B: `erg = f.max (a, b);` bedeutet `x = a; y = b;`

`first = max(100, 2*a+b);` bedeutet `x = 100; y = 2*a+b;`

- Zuordnung von aktuellen zu formalen Parametern gemäß der Reihenfolge in der Methodendefinition
- aktuelle Parameter müssen zuweisungskompatibel mit den formalen Parametern sein
- formale Parameter enthalten Kopien der Werte der aktuellen Parameter (call-by-value)
 - d.h. Wert des aktuellen Parameters wird durch Methodenaufruf nicht verändert
 - Wert des formalen Parameters in der Methode kann im Methodenrumpf verändert werden

main-Methode

Java erlaubt Anweisungen nur im Rahmen von Methoden.

Ein Methodenaufruf ist jedoch selbst eine Anweisung.

- Wie soll also eine Programmausführung starten?

→ Java benutzt die Konvention:

jede Klasse, die eine main-Methode

```
public static void main (String[] args) {...}
```

enthält, kann vom Java-Laufzeitsystem wie ein Programm aufgerufen werden.

z.B.: `java AccountDemo`

Blöcke und lokale Variablen, Sichtbarkeitsbereiche

Blöcke (compound statements) sind für die Zusammenfassung von Anweisungen (z.B. if-Anweisung, Schleifen).

Blöcke können auch Variablendeklarationen enthalten – diese werden **lokale Variable (local variables)** genannt:

- sie sind nur innerhalb des Block verwendbar, in dem sie deklariert wurden – d.h. verwendbar
 - ab ihrer Deklaration und
 - bis zum Ende dieses Blocks
- **Sichtbarkeitsbereich** der lokalen Variablen
- sie existieren nur, bis die Ausführung des Blocks endet

Variablen deklariert in Methoden:

- sind lokal bezüglich der Methode
- der Block ist hier die umgebende Methode

Blöcke und lokale Variablen, Sichtbarkeitsbereiche (Forts.)

Instanzvariablen einer Klasse werden auch **globale Variablen** (**global variables**) genannt

- gehören zu einem Objekt
- existieren die gesamte Lebensdauer des Objekts
- können von Methoden des Objekts (der Klasse) verwendet werden

Mögliches Problem: durch Deklaration einer Variablen in einem inneren Block mit gleichem Namen findet eine *Verdeckung* (*Verschattung, Namenskonflikt*) statt!

→ in Java nur erlaubt: lokale Variable verdeckt Instanzvariable
(verschiedene Regeln in anderen Programmiersprachen)

(siehe Progr. LocalVars)

Selbstreferenz this; Initialisierung von Instanzvariablen

Die Variable `this` (reserviertes Wort in Java) enthält eine Selbstreferenz auf das eigene Objekt:

- `this` ist in jeder Methode automatisch definiert
- `this` hat verschiedene Anwendungen, z.B. Zugriff auf (verdeckte) Instanzvariablen
(siehe Prog. `Fraction`, `LocalVars`)

Initialisierung von Instanzvariablen

- bei Deklaration der Instanzvariablen möglich
- Startwert wird bei Erzeugung eines Objekts der entsprechenden Instanzvariablen zugewiesen

```
public class Account {  
    double balance = 0;  
    . . . . .  
}
```

Datenkapselung

Betrachten z.B. die Klasse `Account`: der Kontostand wird durch die `double`-Instanzvariable `balance` repräsentiert. Damit sind alle arithmetischen Operationen und Funktionen (z.B. `sin`, Wurzel, ...) möglich – jedoch nicht sinnvoll für Kontostände!

→ beliebigen Zugriff verhindern – Zugriff nur über Methoden erlauben!

Zugriffsrechte werden in Java mittels *Modifikatoren* (*modifiers*) gesteuert:

`private` : Zugriff nur innerhalb der Klassendefinition

`public` : freier Zugriff von beliebigen Klassen aus

keine Angabe: Zugriff möglich für Objekte von Klassen, die sich im selben Verzeichnis befinden

(Details dazu in späteren LV oder Literatur)

Datenkapselung (Forts.)

Damit ergibt sich folgende, übliche Vorgehensweise:

- Instanzvariablen als `private` deklarieren
- Zugriffsmöglichkeiten über `public` Methoden anbieten

```
public class Account {  
  
    private double balance = 0;  
    private String name;  
  
    public void deposit(double amount) {  
        .....  
    }  
    .....  
}
```

- Methoden schränken Zugriff ein (bzw. steuern ihn) – z.B. für Kontostand nur Addition und Subtraktion möglich, negativer Kontostand unmöglich
- es sollen sogenannte getter- und setter-Methoden verwendet werden, um Werte von Instanzvariablen setzen bzw. abfragen zu können

Dieses Prinzip heißt **Datenkapselung (data encapsulation)**.

Die `public`-Teile werden als **Schnittstelle (interface)** bezeichnet. Eine Schnittstelle dient als zentrale Verknüpfung von Programmteilen und soll möglichst unverändert bleiben (sonst hoher Bedarf an Überarbeitung)!

(siehe Prog. Account, Bank, BankDemo)

Konstrukturen

Konstrukturen (constructors) sind spezielle Methoden, die bei der Erzeugung von Objekten einer Klasse (mit `new`) aufgerufen werden:

- sie haben den selben Namen wie die Klasse
- sie haben keinen Rückgabetyt und auch nicht `void`
- sie werden automatisch bei der Erzeugung von Objekten mit `new` aufgerufen und ausgeführt

```
public class Square {  
    private int size;  
    private char printSymbol;  
  
    public Square(int dim, char sym) {  
        size = dim;  
        printSymbol = sym;  
    }  
    .....  
}
```

entsprechender Aufruf:

```
Square ourSquare = new Square(9, '=');  
ourSquare.display();
```

Konstrukturen (Forts.)

Wird in der Klasse kein Konstruktor definiert, setzt Java automatisch den *Default-Konstruktor* ein:

- besitzt keine Parameter
- führt im wesentlichen keine Anweisungen aus (neben Objekterzeugung)

Ein Default-Konstruktor kann auch in einer Klasse definiert werden (nun meist mit Aktionen) – sinnvoll dann, wenn

- bestimmte Aktionen nötig sind
- neben Konstruktoren mit Parametern auch einer ohne Parameter nützlich ist (Objekterzeugung mit `new()`)

→ mehrere Konstruktoren in einer Klasse möglich – Unterscheidung mittels

verschiedener Parametrisierung

– genannt *überladen (overloading)*.

```
public class Square {  
    .....  
    public Square() {  
        size = 5;  
        printSymbol = 'x';  
    }  
    .....  
}
```

static-Methoden und -Variablen

static-Methoden und **Variablen** gehören zu einer Klasse und sind *nicht* auf ein konkretes Objekt bezogen (auch als Klassenmethoden bzw. -variablen bezeichnet)

→ Verwendung ohne Instanziierung einer Klasse
(Widerspruch zu OO, jedoch teilweise praktisch)

- Definition mit reserviertem Wort `static`
- Methodenaufruf bzw. Variablenzugriff mit
Klassenname.Bezeichner

- bereits
bekannt:
`Math`,
`SavitchIn`

```
public class Formulas {  
    public static final double PI = 3.14159;  
    public static double circleArea(double radius){  
        return (radius * radius * PI);  
    }  
    .....  
}
```

```
double r = ...;  
double area;  
area = Formulas.circleArea(r);
```


Überladen von Methoden

Mehrere Methoden in einer Klasse mit dem selben Namen sind möglich – **überladen von Methoden (overloading)**.

Die Unterscheidung beruht auf der Parameterliste:

- Anzahl, Typ oder Reihenfolge der formalen Parameter müssen unterschiedlich sein
- keine Rolle spielen return-Typ und Namen der Parameter

```
public class Example {  
    public void calc(int x){...}  
    public void calc(double x){...}  
}
```

```
Example ex = new Example();  
ex.calc(8);      //Aufruf von int-Vers.  
ex.calc(7.5);    //Aufruf von double-Vers.
```