

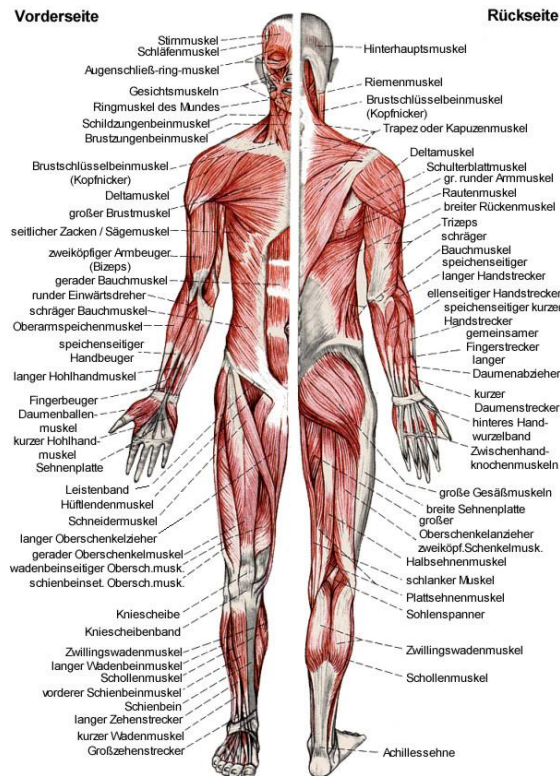
1. Der „imperative Teil“ von Java

Elementare Datentypen
Variablen, Referenzen, Zuweisungen
Ausdrücke, Anweisungen
Prozeduren
Programme
Datentypen, Module, Klassen

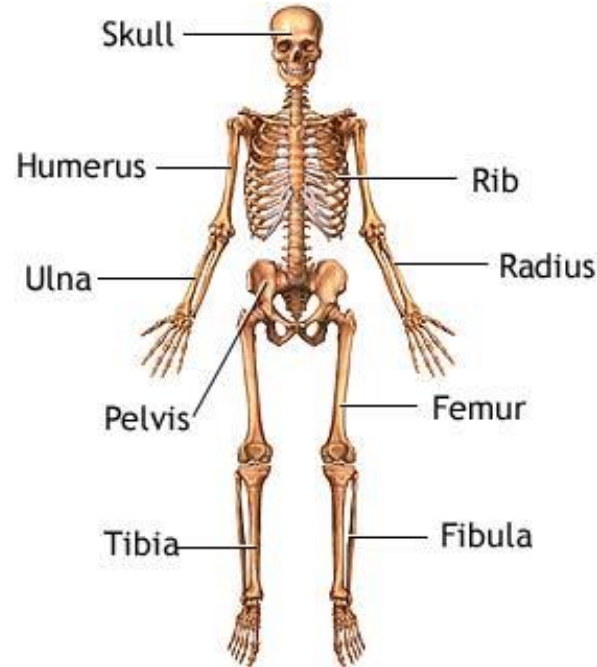
„Anatomie“ von Programmiersprachen

Anatomie des Menschen

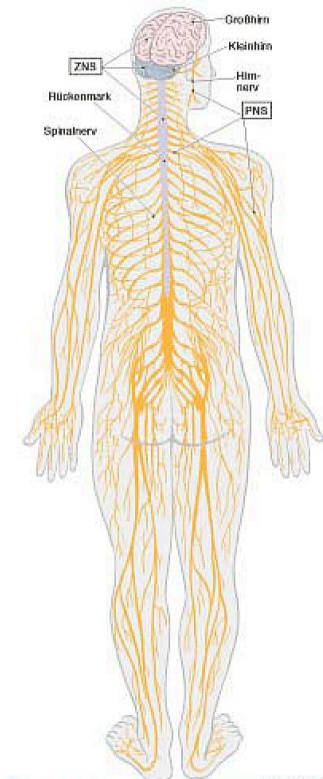
Muskulatur



Skelett



Nervensystem



Anatomie von Programmiersprachen

Deklarationen

Benutzerdefinierte

Programme
Typen
Funktionen
Prozeduren
Variablen

Vordefinierte

Programme
Typen
Funktionen
Prozeduren
Variablen

Ausdrücke

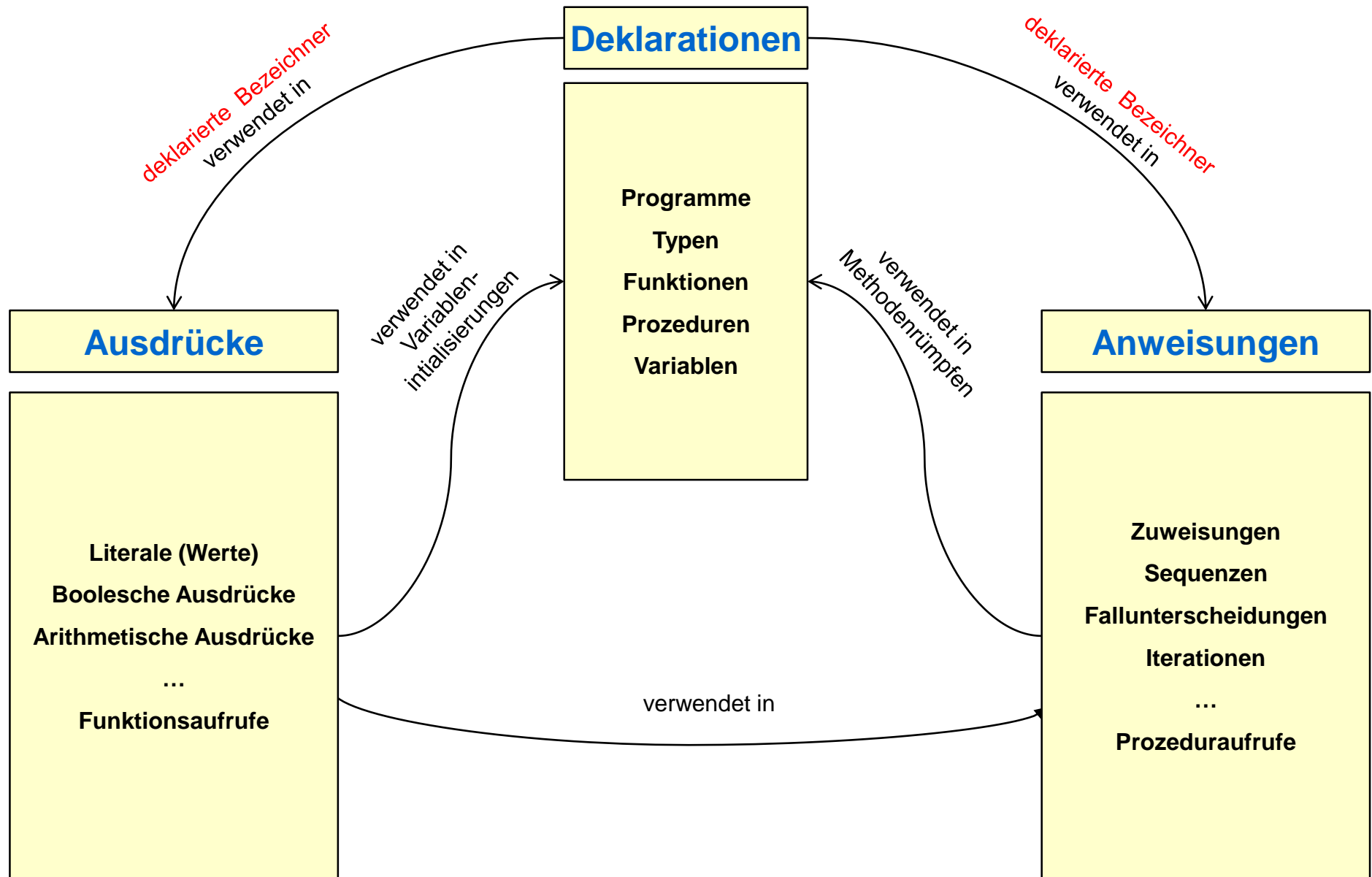
Literale (Werte)
Boolesche Ausdrücke
Arithmetische Ausdrücke
...
Funktionsaufrufe

Anweisungen

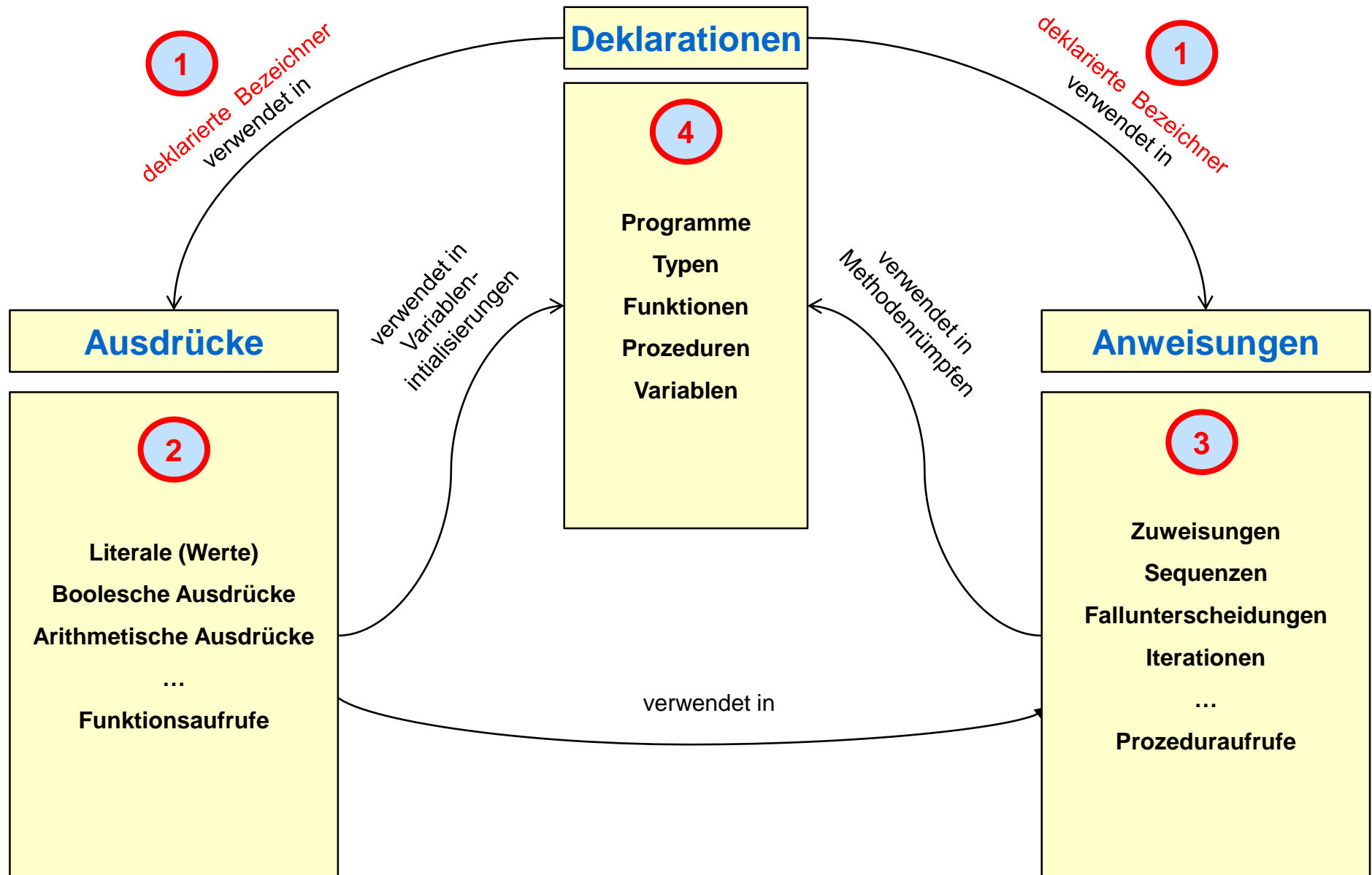
Zuweisungen
Sequenzen
Fallunterscheidungen
Iterationen
...
Prozeduraufrufe



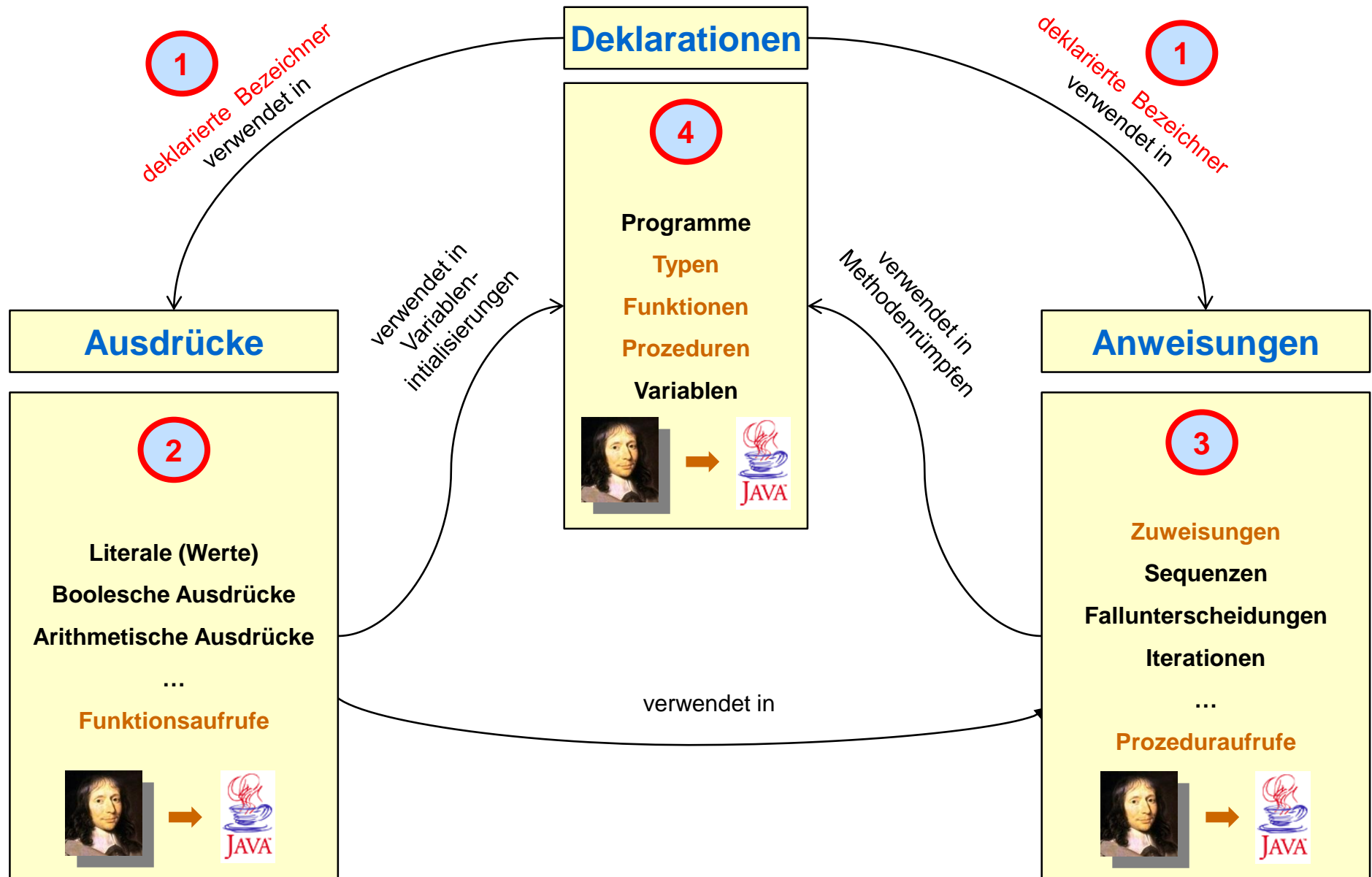
Physiologie: Zusammenspiel



Wie geht's weiter?



Wie geht's weiter?



Bezeichner

Bezeichner in Java

- Ein Bezeichner ist eine Folge von Buchstaben und Zahlen, die **mit einem Buchstaben beginnt**.
- Als Alphabet wird **UNICODE** zugrunde gelegt
 - ◆ „Günter“ ist ein legaler Bezeichner, trotz Umlaut!
 - ◆ Unicode beinhaltet auch die Richtung in der mit den jeweiligen Zeichen geschrieben wird!
 - ⇒ Legal: 123 = عَصِخْتَج; weil in der arabischen Schreibweise von rechts nach links geschrieben wird. Der Bezeichner steht daher rechts, die zugewiesene Zahl links!
 - ◆ Motivation: Internationalisierung
 - ⇒ Programme sollen auf beliebigen Betriebssystemen und in beliebigen Sprachen verfasst werden können
 - ⇒ Um es anderen zu erleichtern die eigenen Programme zu lesen, macht es trotzdem Sinn, sich englischer Bezeichnungen zu bedienen

Selbstdefinierte und reservierte Bezeichner

- Selbstdefinierte Bezeichner
 - ◆ Selbstdefinierte Bezeichner sind Namen für im Programm deklarierte syntaktische Einheiten
 - ◆ Beispiel: Namen von Typen, Variablen, Prozeduren, ...
 - ◆ Die meisten Bezeichner sind selbstdefiniert
- Reservierte Bezeichner („Schlüsselwörter“)
 - ◆ Reservierte Bezeichner sind fester Teil der Programmiersprache
 - ◆ Sie dürfen nicht als Namen eigener Deklarationen verwendet werden

Reservierte Bezeichner in Java

<code>abstract</code>	<code>double</code>	<code>interface</code>	<code>switch</code>
<code>assert</code>	<code>else</code>	<code>long</code>	<code>synchronized</code>
<code>boolean</code>	<code>extends</code>	<code>native</code>	<code>this</code>
<code>break</code>	<code>final</code>	<code>new</code>	<code>throw</code>
<code>byte</code>	<code>finally</code>	<code>package</code>	<code>throws</code>
<code>case</code>	<code>float</code>	<code>private</code>	<code>transient</code>
<code>catch</code>	<code>for</code>	<code>protected</code>	<code>try</code>
<code>char</code>	<code>goto</code>	<code>public</code>	<code>void</code>
<code>class</code>	<code>if</code>	<code>return</code>	<code>volatile</code>
<code>const</code>	<code>implements</code>	<code>short</code>	<code>while</code>
<code>continue</code>	<code>import</code>	<code>static</code>	
<code>default</code>	<code>instanceof</code>	<code>strictfp</code>	
<code>do</code>	<code>int</code>	<code>super</code>	

- Zur Zeit nicht verwendet: `goto` und `const`
- Erst in Java 2 eingeführt: `assert` und `strictfp`
- Zusätzlich reservierte Wörter: die Literale `true`, `false`, `null`

Kein „Operator Overloading“

- „Operator overloading“ (in C++)
 - ◆ Namen von Operatoren können für selbstdefinierte Funktionen wiederverwendet werden
 - ◆ Dadurch wird die Bedeutung der vordefinierten Operatoren geändert
- In Java ist es verboten!
 - ◆ Reservierte Bezeichner sind tatsächlich reserviert
 - ◆ Verwendung der reservierten Bezeichner für eigene Deklarationen wird vom Compiler als Fehler gekennzeichnet

Namenskonventionen

- Namenskonventionen sind Regeln die nicht vom Compiler überprüft werden, sich aber als „guter Sprachgebrauch“ eingebürgert haben
- Sie erleichtern das Lesen von Programmen
 - ◆ durch andere Programmierer
 - ◆ dem Autor selbst, wenn er sein Programm später wieder liest

Namenskonventionen in Java

- Namen für globale Konstrukte (Klassen, Schnittstellen) beginnen mit großem Buchstaben
 - ◆ Beispiel: **K**unde, **G**erät, **K**ontostand, ...
- Namen für Funktionen (Methoden) und Variablen beginnen mit kleinem Buchstaben
 - ◆ Variablenname: **z**, **a**ktueller**K**unde
 - ◆ Methodennamen: **w**ähle**K**anal oder **b**ewege**P**unkt**Zu****P**unkt
- Bei zusammengesetzten Namen schreibt man angefügte Teile jeweils wieder groß (s.o.)
 - ◆ „Kamelhöckerschreibweise“
 - ◆ engl. „camel case“



Foto: Gabriella Fabbri

Elementare Datentypen in Java

byte	8-bit-Zahl in Zweierkomplement-Darstellung
short	16-bit-Zahl in Zweierkomplement-Darstellung
int	32-bit-Zahl in Zweierkomplement-Darstellung
long	64-bit-Zahl in Zweierkomplement-Darstellung
float	32-bit IEEE 754-1985 Gleitkommazahl
double	64-bit IEEE 754-1985 Gleitkommazahl
char	16-bit Unicode 2.0 Zeichen
boolean	Wahrheitswert, true oder false

- N-Bit-Zahl in Zweierkomplement??
- N-Bit IEEE 754-1985 Gleitkommazahl???
- Unicode???

Siehe Vorlesung „Technische Informatik“ im ersten Semester!

Zahlenbereiche der Ganzzahltypen

Typ	Bits	Minimalwert	Maximalwert
byte	8	-128	127
short	16	-32.768	32.767
int	32	-2.147.483.648	2.147.483.647
long	64	-9.223.372.036.854.775.808	9.223.372.036.854.775.807
char	16	0 (' \u0000 ')	65.535 (' \uffff ')

Reihungs-Typen (Arrays)

- In Java sind **Reihungen** (arrays) Spezialfälle von Klassen
 - ◆ Wir führen sie daher erst später im Detail ein 😊
 - ◆ Da sie aber zu den Grundkonzepten des Programmierens gehören, geben wir hier bereits einen kurzen Überblick
- Ist **T** ein Typ, so ist **T[]** der Typ **Reihung von Elementen des Typs T**
 - ◆ Kurz **Reihung von T** oder **T-array**
 - ◆ Der Komponententyp T kann einfach oder wieder zusammengesetzt sein
- Beispiele
 - ◆ **int[]** ist eine Reihung von ganzen Zahlen
 - ◆ **String[]** eine Reihung von Strings (Zeichenketten)

Hüllklassen für elementare Datentypen

- Für alle elementare Datentypen gibt es "Hüllklassen"
 - ◆ Haben den Namen des jeweiligen elementare Datentyps mit Großbuchstaben am Anfang: `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`, `Boolean`
- Funktion von Hüllklassen
 - ◆ Bündelung von Konstanten und Methoden, die für den jeweiligen Typ nützlich sind, z.B. Konversion
 - ⇒ ... aus Strings: `Integer.parseInt("123");`
 - ⇒ ... in Strings: `Integer.toString();`
 - ⇒ ... von `float` nach `int` mit gleichem Bitmuster: `Integer.floatValue(123);`
 - ⇒ ... von `double` nach `long` mit gleichen Bitmuster: `Long.doubleValue(123.4);`
 - ◆ Verwendung von elementaren Datentypen als Objekte
 - ⇒ Siehe später

Ausdrücke

Anweisungen versus Ausdrücke in Java

- Anweisungen

- ◆ steuern den Kontrollfluss des Programms

- ⇒ Beispiel: `if (happy) danceWith(woolf); else kill(woolf);`

- Ausdrücke

- ◆ berechnen Werte

- ⇒ Beispiel: „3+1“ wird ausgewertet zu „4“

- ◆ können jedoch auch Seiteneffekte haben!

- ⇒ Beispiele: Funktionsaufrufe und Zuweisungen (gleich mehr dazu)

- ⇒ Ja, tatsächlich: **Zuweisungen sind in Java Ausdrücke mit Seiteneffekten, keine Anweisungen!**

- Mischformen

- ◆ Bedingter Ausdruck: `Bedingung ? Expr1 : Expr2`

- ⇒ Wenn `Bedingung` wahr ist, liefere als Ergebnis den Wert von `Expr1`, sonst den Wert von `Expr2`

- ◆ Sequentielle boolesche Operatoren: `Expr1 && Expr2, ...`

- ⇒ Werte nur `Expr1` aus wenn das reicht um das Endergebnis zu bestimmen

Syntax von Ausdrücken

Ausdrücke setzen sich zusammen aus

- Literalen
- Variablenbezeichnern
- Funktionsaufrufen
- Operatorsymbolen
- Klammern

Ausdrücke: Literale

Konstanten für Zahlen, Zeichen, Zeichenketten, Wahrheitswerte und Objekte

Literale

- Literale sind eine textuelle Darstellung von Werten.
- Es gibt in Java Literale für
 - ◆ elementare Datentypen
 - ◆ Zeichenketten („Strings“)
 - ◆ die auf nichts verweisende Referenz / das leere Objekt
- Array-Werte haben ebenfalls eine textuelle Darstellung (obwohl sie in Java formal nicht als Literale betrachtet werden)

Literale für ganze Zahlen

- **int**-Literale sind Ziffernfolgen (evtl. mit Vorzeichen)
 - ◆ Ziffernfolgen, die **nicht** mit **0** oder **0x** beginnen werden als **Dezimalzahlen** interpretiert
 - ⇒ Beispiele: 2 -34 53
 - ◆ Ziffernfolgen, die mit **0** beginnen werden als **Oktalzahlen** interpretiert
 - ⇒ Beispiel: 027
 - ◆ Ziffernfolgen, die mit **0x** beginnen sind **Hexadezimalzahlen**
 - ⇒ Dürfen auch die Zeichen **a, b, c, d, e, f** für 10, 11, 12, 13, 14, 15 enthalten
 - ⇒ Beispiel: 0x17 0x1b 0xffff
 - ◆ Achtung: 14 ≠ 014 ≠ 0x14 !!!
- **long**-Literale sind int-Literale mit nachgestelltem **l** oder **L**
 - ◆ Beispiel: 23l -3L +53L 027L

Literale für reelle Zahlen

- **double**-Literale enthalten
 - ◆ einen Dezimalpunkt
 - ◆ oder ein **e** bzw. **E** für Zehnerexponenten
 - ◆ oder ein nachgestelltes **d** (oder **D**)
 - ◆ Beispiele: 2. -3.4 53**e**4 0**d** 2**d** -3.4**d** 53**e**4**d**
- **float**-Literale sind double-Literale, die statt des **d** bzw. **D** ein nachgestelltes **f** (oder **F**) haben
 - ◆ 2**f** -3.4**f** 53**e**4**f**

Literale für alphabetische Zeichen (Character)

In Java stellt man **einzelne Zeichen** dar

- **entweder** als Zeichen in Hochkommata
 - ◆ Falls die Tastatur es erlaubt und es kein Sonderzeichen ist
 - ◆ Beispiel: `char c = 'A';`
- **oder** als Hexadezimalzahl für Bitmuster in Unicode
 - ◆ Beispiel: `char c = '\u0041';`

Escape-Sequenz	Unicode	Zeichen
<code>\b</code>	<code>\u0008</code>	<i>backspace</i> , BS
<code>\t</code>	<code>\u0009</code>	Tabulator, <i>horizontal tab</i> , HT
<code>\n</code>	<code>\u000a</code>	Zeilenvorschub, <i>linefeed</i> , <i>newline</i> , LF
<code>\f</code>	<code>\u000c</code>	Seitenvorschub, <i>form feed</i> , FF
<code>\r</code>	<code>\u000d</code>	Wagenrücklauf, <i>carriage return</i> , CR
<code>\"</code>	<code>\u0022</code>	Anführungszeichen, <i>double quote</i> , "
<code>\'</code>	<code>\u0027</code>	Hochkomma, <i>single quote</i> , '
<code>\\</code>	<code>\u005c</code>	<i>backslash</i> , \

Literale für Zeichenketten, Wahrheitswerte, Objekte

- **String**-Literale sind in doppelte Anführungszeichen eingeschlossene Zeichenketten
 - ◆ `"Welcome to the real world!"`
 - ◆ Auf Strings gehen wir später weiter ein, da sie in Java keine primitiven Werte, sondern Objekte sind.
- **boolean**-Literale sind `true` und `false`
- `null` ist das einzige **Objekt**-Literal
 - ◆ Es stellt die Referenz dar, die auf nichts zeigt.
 - ◆ Alternativ kann man es als das nicht vorhandene Objekt interpretieren.

Array-“Literale”

- Array-Werte haben ebenfalls eine textuelle Darstellung
 - ◆ `{ 100, 200, 300, -99.99 };` // Ein Array mit 4 reellen Zahlen
 - ⇒ Alle Elemente werden auf den allgemeinsten verwendeten Typ konvertiert
 - ◆ `{ {1, 2} , {3, 4, 5, 6} }` // Ein Array von int-Arrays
 - ⇒ Beachte, dass die Elementarrays unterschiedlich lang sein können!
 - ◆ `{ {1, 2} , {3, 4, 5, 6}, 7 }` // Illegal
 - ⇒ Die 7 ist kein int-Array!
 - ◆ `{ {1, 2} , {3, 4, 5, 6}, null }` // Legal
 - ⇒ `null` ist ein legaler Wert für ein Array-Objekt!
- In Java gibt es formal den Begriff des Array-Literals nicht. Obiges wird als „Array Initializer“ bezeichnet.

Ausdrücke: Variablenbezeichner

Variablen

- Variablen dienen der Speicherung von Werten
- Werte sind (in Java)
 - ◆ Elemente primitiver Datentypen *oder*
 - ◆ Referenzen auf Objekte
- Variablen, deren Wert eine Referenz auf ein Objekt ist, heißen Objektvariablen
 - ◆ In Java sind die Werte von Variablen nie Objekte sondern immer nur Referenzen auf Objekte!
 - ◆ In C++ und Eiffel gibt es beide Möglichkeiten (eine Variable kann ein komplettes Objekt beinhalten)
- Die Verwendung eines Variablennamens ist (neben Literalen) die einfachste Form eines Ausdrucks.

Deklaration von Variablen

Vorwarnname: Deklarationen kommen eigentlich erst später dran! Variablen sind aber einfach zu elementar um sie so lange zu verschweigen... 😊

- Variablendeklarationen haben *immer* die Form

```
[Modifikatoren] Typname Variablenname [= Initialisierung];
```

Eckige Klammern gehören hier nicht zur Java-Syntax sondern markieren optionale Teile

- ◆ Obiges gilt auch für nicht-elementare Typen (d.h. für die später eingeführten Klassen und Interfaces)

- Beispiele

```
int counter;                // Deklaration eines Zählers
int counter = 0;            // Deklaration mit Initialisierung
int start, end;             // Zwei integer-Variablen
double[] prices = { 100, 200, 300, -99.99 }; // Initialisierte Deklaration eines Double-Arrays
final double pi = 3.14159265; // Konstantendeklaration (,final')
```

- ◆ Wir werden später noch weitere **Modifikatoren** kennen lernen.

Ausdrücke: Operatoren

Arithmetische Operatoren

Zuweisung und Zuweisungsoperatoren

Boolesche Operatoren

Bitweise Operatoren

Ausdrücke: Operatoren

- Operatoren sind in die Programmiersprache eingebaute Funktionen, die dadurch gegebenenfalls komfortabler geschrieben werden können
- In Java verwendete Notationen
 - ◆ **Präfix**: vorangestellt, z.B. Preinkrement `++i`
 - ◆ **Postfix**: nachgestellt, z.B. Postinkrement `i++`
 - ◆ **Infix**: in die Mitte gestellt, z.B. `1+2`
 - ◆ **Roundfix**: drumherumgestellt, z.B. ein Klammernpaar als Operator aufgefaßt: `{1, 3, 4}` als Operator der ein Array konstruiert.

Präzedenzen von Operatoren

- Die besondere Schreibweise, die bei eingebauten Operatoren möglich ist, kann zu mehrdeutigen Ausdrücken führen
 - ◆ $1+2*3$ könnte syntaktisch (gemäß den Regeln 1 bis 3 für die Konstruktion von Ausdrücken) sowohl als Ausdruck $(1+2)*3$ als auch als $1+(2*3)$ gelesen werden
- Damit die Semantik eindeutig ist, ohne Ausdrücke mit Klammern pflastern zu müssen, gibt man jedem Operator eine bestimmte **Präzedenz**
 - ◆ Synonyme: Präzedenz, Bindungskraft, Vorrang
 - ◆ Wir sagen $*$ bindet stärker als $+$ und meinen damit, dass der Ausdruck als $1+(2*3)$ gelesen werden soll

Präzedenzen von Operatoren in Java

Postfix Operatoren

[] . (params) expr++ expr--

Unäre Operatoren

++expr --expr +expr -expr ~ !

Erzeugung und Anpassung

new (type) expr

Multiplikative Op.

* / %

Additive Op.

+ -

Schiebe-Op.

<< >> >>>

Relationale Op.

< > >= <= instanceof

Gleichheits-Op.

== !=

Bitweises und log. UND

&

Bitweises und log. XOR

^

Bitweises und log. ODER

|

Bedingtes logisches UND

&&

Bedingtes logisches ODER

||

Bedingung

?:

Zuweisungs-Operator

= += -= *= /= %= >>= <<= >>>=

&= ^= |=

Assoziativität: Motivation

- Problem: Präzedenz reicht nicht!
 - ◆ In $f(1) + g(2) + h(3)$ könnten die Funktionen f , g und h Seiteneffekte auslösen (z.B. Ausgaben am Bildschirm)
 - ◆ Daher muss selbst bei gleichen Operatoren (gleiche Präzedenz, mathematisch kommutativ) die Reihenfolge der Ausführung definiert sein!
- Idee
 - ◆ Die **Assoziativität** regelt die Reihenfolge der Ausführung von Operatoren gleicher Präzedenz
 - ◆ Ergibt die Anwendung der Präzedenz- und Assoziativitätsregeln nicht die gewünschte Ausführungsreihenfolge so müssen explizit Klammern gesetzt werden

Assoziativität: Definition

- **Rechts-Assoziativität:** Auswertung beginnt rechts
 - ◆ Zuweisungsoperatoren sind in Java rechts-assoziativ
 - ◆ $x=y=1$ steht somit für $x=(y=1)$
- **Links-Assoziativität:** Auswertung beginnt links
 - ◆ Alle binären Operatoren außer den Zuweisungsoperatoren sind links-assoziativ
 - ◆ $1+2+3$ steht somit für $(1+2)+3$

Stil: Expliziter ist besser!

- Was ist das Ergebnis des Ausdrucks `a & b == b | c`?
 - ◆ Da `==` stärker bindet als `&` und `|` entspricht Obiges
`a & (b == b) | c`
 - ◆ Da `&` stärker bindet als `|` entspricht es ferner
`(a & true) | c` und somit `a | c`
- Solche Fehler sind sehr schwer zu finden!
 - ◆ Das Programm ist syntaktisch korrekt, somit meldet der Compiler keine Fehler
 - ◆ Selbst überliest man den Fehler immer wieder
- Empfehlung: Eindeutigkeit durch Klammerung
 - ◆ Bei Verwendung von weniger gebräuchlichen Operatoren die intendierte Semantik durch Klammerung festlegen!

Ausdrücke: Zuweisung und Zuweisungsoperatoren

Assignment is undoubtedly the most characteristic feature of programming a digital computer, and one that most clearly distinguishes it from other branches of mathematics.

C. A. R. Hoare (1969)

Anweisungen versus Ausdrücke

● Anweisungen

◆ steuern den Kontrollfluss des Programms

⇒ Beispiel: `if (...) danceWith(woolf); else kill(woolf);`

● Ausdrücke

◆ berechnen Werte

⇒ Beispiel: „3+1“ wird ausgewertet zu „4“

◆ können in Java Seiteneffekte haben!

⇒ Beispiele: Funktionsaufrufe und Zuweisungen (gleich mehr dazu)

● Mischformen

◆ Bedingter Ausdruck: `Bedingung ? Expr1 : Expr2`

⇒ Wenn Bedingung wahr ist, liefere den Wert von Expr1, sonst den Wert von Expr2

◆ Sequentiellen boolesche Operatoren: `Expr1 && Expr2`,

⇒ Werte nur `Expr1` aus wenn es reicht um das Endergebnis zu bestimmen

Zuweisung: Pascal versus Java

Pascal

- Der Gleichheitstests ist
=
- Der Zuweisungsoperator ist
:=
- Die Zuweisung ist eine
Anweisung
 - ◆ Sie liefert keine Ergebnis!

Java, C, C++

- Der Gleichheitstests ist
==
- Der Zuweisungsoperator ist
=
- Die Zuweisung ist ein
Ausdruck!
 - ◆ Sie liefert ihren rechten Teil als Ergebnis!
 - ◆ Die eigentliche Zuweisung geschieht als Seiteneffekt des Ausdrucks!

Zuweisung: Beispiele

Pascal

- $x := y := 1$ ist illegal. Ersatz:
 - ◆ $y := 1;$
 - ◆ $x := y;$
 - ◆ Letztere Zeile könnte auch $x := 1;$ lauten.
- $x := (y := 1) + 5$ ist illegal. Ersatz:
 - ◆ $y := 1;$
 - ◆ $x := y + 5;$

Java, C, C++

- $x = y = 1$
 - ◆ bezeichnet $x = (y = 1)$
 - ◆ weist y den Wert 1 zu
 - ◆ der an x zugewiesene Wert von $y = 1$ ist 1
- $x = (y = 1) + 5$
 - ◆ hat den Wert 6
 - ◆ weist y den Wert 1 zu
 - ◆ weist x den Wert 6 zu

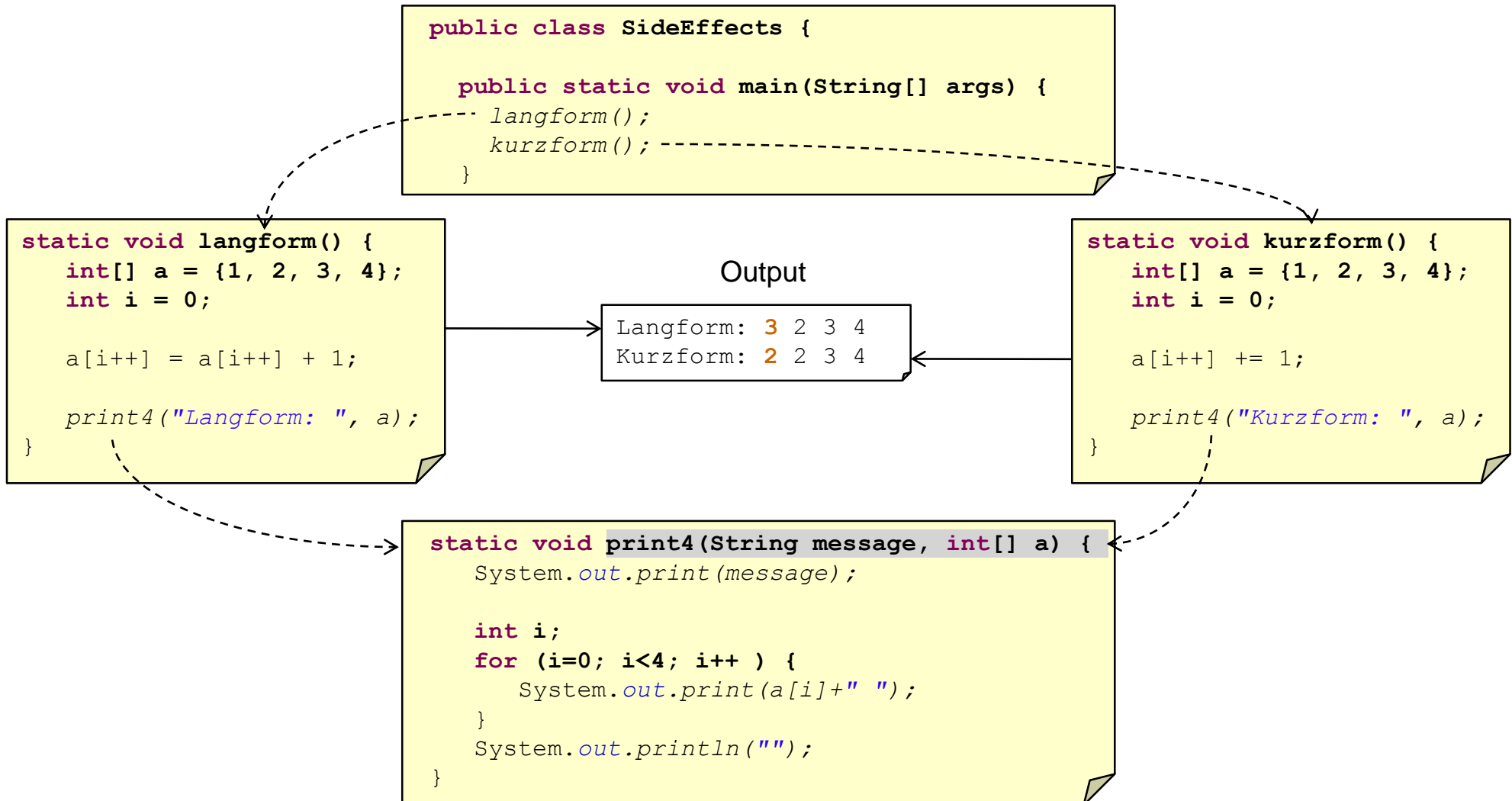
Kombinierte Zuweisungsoperatoren in Java

- Zuweisungsausdrücke der Form `v += expr ;` sind eine Schreibabkürzung für `v = v + expr ;`
 - ◆ Solche Zuweisungen kommen häufig vor
 - ◆ Hier ist Java wieder in der Tradition von `C` und `C++`
- Für **jeden binären Operator** `o` gibt es einen entsprechenden kombinierten Zuweisungsoperator `o=` in Java.
 - ◆ Beispiele: `-=` `*=` `/=` `%=` `>>=` usw.

Kombinierte Zuweisungsoperatoren versus Langform

- Ist Wert von $v = v \circ expr$ und $v \circ= expr$ immer gleich?
 - ◆ Dabei steht \circ für einen der binären Operatoren
- Falls v keine Seiteneffekte hat: „Ja!“
- Falls v Seiteneffekte hat: „Nein!“
 - ◆ Beispiel: Falls v ein Arrayzugriff mit Seiteneffekt ist:
 - ⇒ $a[i++] = a[i++] + 1;$
 - ⇒ $a[i++] += 1;$
 - ◆ Bei $v \circ= expr$ wird Wert von v nur ein einziges Mal ausgewertet, im Gegensatz zu $v = v \circ expr$
 - ◆ Eher pathologische Fälle
 - ⇒ Sollten bei „gutem Programmierstil“ nicht vorkommen

Beispiel zu Kurzform versus Langform



Beispiel zu Kurzform versus Langform – Erklärung des Ergebnisses

```
public class SideEffects {  
  
    public static void main(String[] args) {  
        langform();  
        kurzform();  
    }  
}
```

5. a[0] wird das Ergebnis von 2+1 zugewiesen

```
static void langform() {  
    int[] a = {1, 2, 3, 4};  
    int i = 0;  
  
    a[i++] = a[i++] + 1;  
    print4("Langform: ", a);  
}
```

2. a[0] wird als Ziel der Zuweisung bestimmt

1. i==0 wird gelesen, danach i auf 1 gesetzt

3. i==1 wird gelesen, danach i auf 2 gesetzt

4. a[1] == 2 wird gelesen

4. a[0] wird das Ergebnis von 1+1 zugewiesen

```
static void kurzform() {  
    int[] a = {1, 2, 3, 4};  
    int i = 0;  
  
    a[i++] += 1;  
    print4("Kurzform: ", a);  
}
```

2. a[0] wird als Ziel der Zuweisung bestimmt

1. i==0 wird gelesen, danach i auf 1 gesetzt

3. zu a[0] == 1 wird 1 addiert

Langform: 3 2 3 4
Kurzform: 2 2 3 4

Zur Erklärung der Reihenfolge der Operationen, siehe die Folien über Präzedenzregeln!

Ausdrücke: Weitere Operatoren

Boolesche Operatoren

Bitmuster-Operatoren

Arithmetische Operatoren

Boolesche Operatoren

- In Java wird der Typ `boolean` mit den Literalen `true` und `false` zur Verfügung gestellt
 - ◆ In C und in frühen Versionen von C++ benutzt man stattdessen 0 für false und Werte ungleich 0 für true (in Analogie zur Elektrotechnik, wo keine Spannung als false und Spannung als true gilt)
- **Boolesche Operatoren** sind

<code>&</code>	logisches UND (\wedge)
<code> </code>	logisches ODER (\vee)
<code>^</code>	logisches EXKLUSIVES ODER (XOR)
<code>!</code>	logische Negation (\neg)
<code>&&</code>	bedingtes logisches UND (\wedge)
<code> </code>	bedingtes logisches ODER (\vee)

Bedingte und symmetrische boolesche Operatoren

- Die **bedingten booleschen Operatoren** **&&** und **||** werten ihren rechten Operanden nur dann aus, falls das Ergebnis nicht schon aus dem Linken herleitbar ist
 - ◆ Beispiel: **((b = false) && (c = true))**
 - ⇒ Die Zuweisung **c = true** wird nicht ausgeführt, da der Wert der Zuweisung **b = false** wieder **false** ist und damit der Wert des Gesamtausdrucks schon als **false** feststeht
 - ◆ Synonyme: **Faule Auswertung, lazy evaluation**
- Die **symmetrischen binären Operatoren** **&**, **|** und **^** werten hingegen immer **beide** Operanden aus
 - ◆ Beachte, dass diese Operatorsymbole auch Operationen auf Bitmustern bezeichnen können

Vergleichsoperatoren

- In Java gibt es die üblichen Vergleichsoperatoren für die gängigen Datentypen

>	größer
>=	größer oder gleich
<	kleiner
<=	kleiner oder gleich
==	gleich
!=	ungleich

- Beispiele

- ◆ $(x \% 2 == 0)$ liefert true falls x eine gerade ganze Zahl ist
- ◆ $((a >= b) \parallel (a <= b)) \&\& (a != a)$ liefert false

Vergleichsoperatoren: Warnung!

- In Java, C und C++ führt die Verwechslung des Gleichheitsoperators `==` mit dem Zuweisungsoperator `=` häufig zu schwer zu findenden Fehlern
 - ◆ Beispiel: Obwohl `a == false` evaluiert `(a = true)` zu `true`, da dies der Wert der Zuweisung ist
- Es hat sich daher eingebürgert, keine Vergleiche auf boolesche Werte anzuwenden
 - ◆ Man schreibt `(a && !b)` statt `(a == true && b == false)`

Bitmuster und bitweise logische Operatoren

- Jeder Wert eines Ganzzahltyps kann auch als Bitmuster angesehen werden
- Die **bitweisen („bitwise“) logischen Operatoren** wenden die jeweilige Operation simultan auf jedes Bit der Operanden an und liefern ein neues Bitmuster als Wert

$a \ \& \ b$ bitweises UND
 $a \ | \ b$ bitweises ODER
 $a \ \wedge \ b$ bitweises EXKLUSIV-ODER
 $\sim a$ bitweises KOMPLEMENT

- Beispiel

x	y	$x \ \& \ y$	$x \ \ y$	$x \ \wedge \ y$	$\sim x$	$\sim y$
1010	1100	1000	1110	0110	0101	0011

Bitmuster: Schiebeoperatoren

- Die **Schiebe-Operatoren** („shift operators“) verschieben das Bitmuster innerhalb eines Ganzzahlwertes
 - ◆ $x \ll n$ schiebt die Bits in x um n Stellen nach links und füllt rechts Nullen auf
 - ◆ $x \gg n$ schiebt die Bits in x um n Stellen nach rechts und füllt links mit dem Vorzeichenbit auf (**arithmetic shift**)
 - ◆ $x \ggg n$ schiebt die Bits in x um n Stellen nach rechts und füllt links mit Nullen auf (**logical shift**)
- Die jeweils herausgeschobenen Bits werden einfach fallengelassen

Bitmuster: Schiebeoperatoren

- $x \ll n$ und $x \gg n$ entsprechen einer **Multiplikation** mit 2^n (bzw. **Division durch 2^n**)
 - ◆ Der arithmetische Shift dient zur Bewahrung des Vorzeichens bei Divisionen einer negativen Zahl
 - ⇒ Man beachte, dass $(-1 \gg 1) == -1$, wogegen $(-1/2) == 0$
- Der Ergebnistyp einer Schiebeoperation ist der Typ des linken (zu verschiebenden) Operanden
- Der rechte Operand, der ganzzahlig sein muss, ist der **Schiebezähler** (shift count)
 - ◆ Für den Schiebezähler ist nur ein nicht-negativer Wert sinnvoll, der kleiner ist als die Anzahl der Bits im Typ des zu verschiebenden Wertes
 - ⇒ Um dies sicherzustellen wird in Java der Wert des Schiebezählers automatisch entsprechend maskiert

Arithmetische Operatoren

- Für die elementaren Zahltypen sind jeweils die folgenden Operationen definiert
 - ◆ unäres **+** und **-**
 - ◆ binäres **+**, **-**, *****, **/** und **%** (Rest bei Division)
- Besonderheiten
 - ◆ Auf den **Ganzzahltypen** ist **/** die Ganzzahldivision
 - ◆ Auf **Floating-Point** Werten ist **/** die Division reeller Zahlen
 - ◆ Die *modulo* Operation **%** existiert im Gegensatz zu C/C++ auch auf den Gleitkommatypen
 - ⇒ Beispiel: **7.0 % 2.5** ergibt **2.0**

Inkrement und Dekrement-Operatoren

- Java kennt die unären Operatoren **++** und **--**
 - ◆ In Java können sie auch auf Werte vom Typ **float** und **double** angewendet werden
- Sie können in Postfix und Präfixform verwendet werden
 - ◆ **i--** (bzw. **i++**) liefert den Wert der Variablen **i**
 - ⇒ als **Seiteneffekt** wird der Wert von **i** um Eins erniedrigt (bzw. erhöht)
 - ◆ **--i** (bzw. **++i**) liefert den Wert von **i-1** (bzw. **i+1**)
 - ⇒ als **Seiteneffekt** wird der Wert von **i** um Eins erniedrigt (bzw. erhöht)
 - ◆ Der Seiteneffekt der **Präfixform** findet **vor** der Rückgabe des Variablenwertes statt, der der **Postfixform** **danach**!

Inkrement und Dekrement-Operatoren

- Die Operatoren wie **+=** und **++** wurden in **C** eingeführt, um spezielle Instruktionen von **CISC Prozessoren** ausnützen zu können
 - ◆ Und damit effizienteren Code erzeugen zu können
- Ob sie zu einem Geschwindigkeitsgewinn führen, hängt vom Prozessor und dem Übersetzer ab
 - ◆ Bei heutigen Übersetzern, die den erzeugten Code hoch optimieren können, wird es so gut wie nie zu einem Geschwindigkeitsvorteil kommen

Inkrement und Dekrement-Operatoren

- In jedem Fall ist der Gebrauch der Kurzformen inzwischen z. T. **idiomatisch** geworden
 - ◆ d. h. Ausdrücke wie **i++** sind Teil gewisser **Programmiersmuster** (oder **-Idiome**), die ein **C**, **C++** oder **Java Programmierer automatisch anwendet und versteht**
- Es hat sich z. B. **eingebürgert**, die Postfixform **i++** als **kanonische Inkrementform** zu benutzen; man schreibt also **immer i++**; statt der gleichwertigen Anweisungen **i += 1**; oder **i = i+1**;
- Wir werden weitere solche Idiome im Zusammenhang mit Schleifenkonstrukten und Reihungen kennen lernen

Ganzzahlarithmetik

- Ganzzahlarithmetik in Java ist **Zweierkomplement-Arithmetik modulo dem Darstellungsbereich**
 - ◆ Es werden keine Überläufe erzeugt sondern nicht darstellbare Bits einfach abgeschnitten
 - ◆ Dadurch wird der Darstellungsbereich zu einem „Ring“ geschlossen
 - ◆ Zählt man über das Ende des positiven Bereichs hinaus, kommt man zur kleinsten negativen Zahl
- Siehe Foliensatz „Exkurse“, Abschnitt „Zahlendarstellung“ bzw. Vorlesung „Technische Informatik“

Ganzzahlarithmetik: Ganzzahldivision

- Ganzzahldivision wird in Java mit `/` bezeichnet
- Der Rest bei Ganzzahldivision (modulo) mit `%`
- Java rundet immer zur Null hin, um Ganzzahlen zu erhalten
 - ◆ Die Ganzzahl-Division `5/2` ergibt `2`
 - ◆ Die Ganzzahl-Division `-5/2` ergibt `-2`
 - ◆ Der Rest `5%2` bei Ganzzahldivision ergibt `1`
 - ◆ Der Rest `(-5)%2` bei Ganzzahldivision ergibt `-1`
- Für Ganzzahlen gilt $(x/y)*y + x\%y == x$

Gleitkommaarithmetik nach IEEE 754-1985 Standard

- Arithmetik kann nach $+\infty$ und $-\infty$ überlaufen (**overflow**)
 - ◆ Der Zahlenbetrag wird zu groß für die gewählte Repräsentation
 - ◆ Es gibt Werte für **POSITIVE INFINITY** und **NEGATIVE INFINITY**
- Arithmetik kann nach $+0.0$ oder -0.0 unterlaufen (**underflow**)
 - ◆ Der Zahlenbetrag wird zu klein für die gewählte Repräsentation
 - ◆ Es gibt eine positive und eine negative Null, wobei $+0.0 == -0.0$

Gleitkommaarithmetik: NaN

- Java-Arithmetik ist ‚non-stop‘
 - ◆ Rechnung liefert immer einen Wert, evtl. NaN
- Der Wert NaN („not a number“)
 - ◆ Repräsentiert Rechnungen, denen kein sinnvoller Wert zugewiesen werden kann
 - ◆ Beispiele
 - ⇒ $0 * (+\infty)$
 - ⇒ $0/0$
 - ⇒ $+\infty + -\infty$
 - ◆ Beachte aber, dass etwa $+\infty + 5$ den Wert $+\infty$ hat
 - ◆ Ausdrücke, in denen NaN vorkommt, liefern immer NaN als Ergebnis

Ausdrücke: Typkonversion

Motivation

Ersetzbarkeitsprinzip / Subtypbegriff

Erweiternde Konversion (Widening)

Einschränkende Konversion (Narrowing)

Warum Typkonversion?

- Bisher sind wir davon ausgegangen, dass alle Ausdrücke **genau typkorrekt** aufgebaut sind
 - ◆ Hat ein Operator oder eine Funktion ein Argument **a** vom Typ **T** verlangt, haben wir für **a** nur Ausdrücke eingesetzt, die **genau den Typ T** haben.
- Allerdings ist diese Regel zu starr
 - ◆ Der Ausdruck **2+2L** ist in obigem Sinn nicht typkorrekt
 - ◆ Für **int x = 2; long y;** ist der Ausdruck **y = x** ebenfalls nicht genau typkorrekt
- Wunsch: Mehr Flexibilität
 - ◆ Man möchte zumindest dort eine **automatische Typkonversion** haben, wo dies „**problemlos**“ geschehen kann

Ersetzbarkeit

- Frage
 - ◆ Was soll auf der vorherigen Folie „problemlos“ bedeuten?
- Antwort: Ersetzbarkeitsprinzip
 - ◆ Konversion von Typ S zu Typ T ist problemlos, wenn Werte von Typ S überall eingesetzt werden können, wo Werte von Typ T erwartet werden.
 - ◆ In diesen Fall sagen wir „ S ist Subtyp von T “ und schreiben „ $S \subseteq T$ “

„Die Idee klingt einleuchtend, ist mir aber zu abstrakt. Was heißt das denn konkret für Zahlenwerte?“

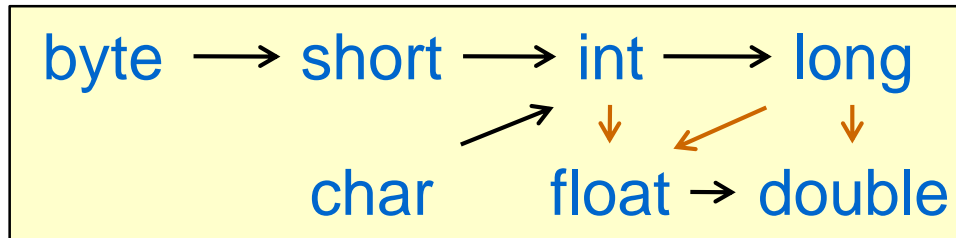
Subtypbeziehungen elementarer Zahlentypen

In Java gilt für elementare Zahlentypen:

- S ist Subtyp von T , $S \subseteq T$, falls der Wertebereich von T größer ist als der von S
 - ◆ Im Bereich der Gleitkommazahlen gilt $\text{float} \subseteq \text{double}$
 - ◆ Im Bereich der Ganzzahltypen gilt $\text{byte} \subseteq \text{short} \subseteq \text{int} \subseteq \text{long}$
- Außerdem erlaubt man die Benutzung eines char als Ganzzahl

„Erweiternde“ Konversionen elementarer Typen („widening“)

- Behalten die Größenordnung der Zahl bei
 - ◆ verlieren beim Übergang zur Gleitkommadarstellung aber an Genauigkeit (die kleinstwertigsten Nachkommastellen fallen weg):



Legende:

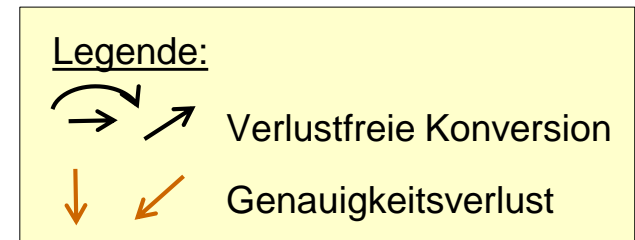
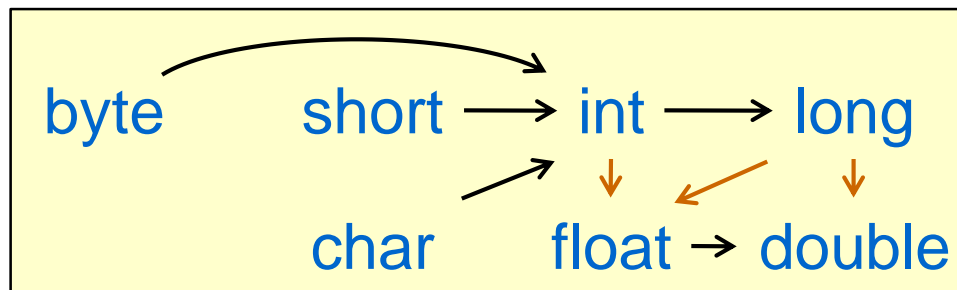
- ↗ Verlustfreie Konversion (Subtypen)
- ↓ ↘ Konversion mit Genauigkeitsverlust

- Werden implizit ausgeführt
 - ◆ bei Zuweisungen / Parameterübergaben
 - ◆ bei Anwendung unärer und binärer Operatoren

Erweiternde Typkonversionen – für binäre Operatoren

- Allgemein

- ◆ Operanden mit unterschiedlichen Darstellungen, werden zu derjenigen der beiden Darstellungen konvertiert, die nachfolgend weiter rechts oder weiter unten steht:



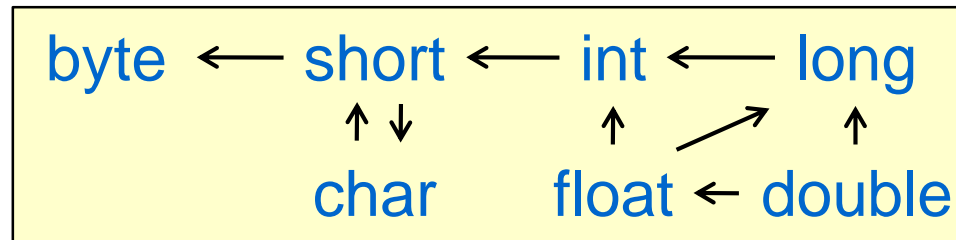
- ◆ Besonderheit: `byte` wird direkt in `int` umgewandelt!
- Danach bezeichnet das Operatorsymbol die Operation des erweiterten Typs. Dies ist auch der Typ des Ausdrucks.

Typkonversionen für Operatoren: Beispiele

- Der Typ von $1 + 1L$ ist `long`
- Der Typ von $1 + 1.0f$ ist `float`
- Der Typ von $1L + 1.0f$ ist `float`
- Der Typ von $1 + 1.0$ ist `double`
- Sei `byte a, b, c;` gegeben.
 - ◆ Der Typ von $a + b$ ist `int`
 - ◆ Der Typ von $-b$ ist `int`
 - ◆ Die Zuweisungen $a = -b;$ und $c = a + b;$ sind `ungültig`
 - ⇒ Ein `int` darf `nicht implizit` zu einem `byte` `verengt` werden
(um vor Fehlern zu schützen)
 - ⇒ Allgemein: `Einengende Konversionen` sind nur explizit erlaubt!
(Siehe nächste Folien)

„Einengende“ Konversionen elementarer Typen („narrowing“)

- Verringern möglicherweise sowohl die Größenordnung als auch die Genauigkeit und das Vorzeichen der Zahl
 - ◆ höchstwertige Bits werden gestrichen (incl. Vorzeichenbit)!
 - ◆ es wird zur Null hin gerundet
- Mögliche Übergänge



- Einengende Konversionen werden **nur** auf explizite Anweisung hin ausgeführt → **Explizite Typkonversion**

Explizite Typkonversionen

- Sind möglich mit Hilfe des Konversionsoperators
`(type) expr`
- Wandeln den Typ des Ausdrucks `expr` zu `type` um, sofern möglich
 - ◆ Nicht jede Typkonversion kann vom Programmierer erzwungen werden
 - ◆ Z.B. ist die Konversion von `boolean` nach `int` nicht legal
- Synonyme Bezeichnungen
 - ◆ deutsch: **explizite Typkonversion, Typerzwingung**
 - ◆ englisch: **type coercion, type cast**

Explizite Typkonversionen: Beispiele

- `int x = (int) 1L;`
 - ◆ Der Wert 1 vom Typ long wird zum Wert 1 vom Typ int umgewandelt
- `char z = (char) 127;`
 - ◆ Der Wert 127 vom Typ int wird zu einem Zeichenwert umgewandelt (der das DEL-Zeichen repräsentiert)
- `byte c = (byte) (a + b);`
 - ◆ Der Ausdruck ist korrekt, da der Typ des Teilausdrucks (a + b) explizit von int zu byte umgewandelt wird, wodurch der Wert modulo 128 reduziert wird

Explizite Typkonversionen: Beispiele (Forts.)

(byte) 128 ist -128.

(byte) 129 ist -127.

(byte) 127 ist 127.

(byte) -128 ist -128.

(byte) 256 ist 0.

(byte) 257 ist 1.

(byte) 255 ist -1.

(byte) -13.5 ist -13.

Typkonversionen: Rückblick

Erweiterung (widening)

Ist meist eine injektive, total definierte Funktion.

Kann implizit oder explizit erfolgen!

T
↑
↓
S

Einengung (narrowing)

Ist meist nicht injektiv und manchmal auch nur partiell definiert.

Kann nur explizit erfolgen!

Implizite Typkonversion

- führt immer nur eine **Erweiterung** durch
- Subtyp S → Obertyp T

Explizite Typkonversion

- ist der einzige Weg um eine **Einengung** zu erzwingen
- Obertyp T → Subtyp S

Anweisungen

Anweisungen in Java

- Eine **Anweisung** (**statement**) weist den Computer an
 - ◆ eine Berechnung vorzunehmen (**expression statement**)
 - ◆ eine Variable einzurichten und zu initialisieren (**declaration statement**)
 - ◆ einen Schritt im Programmfluss vorzunehmen (**control flow statement**)
- Eine Anweisung wird durch ein Semikolon **;** abgeschlossen
- Die **leere Anweisung** besteht nur aus dem abschließenden Semikolon **;** oder aus einem Paar geschweifter Klammern **{ }**

Ausdrucks-Anweisung (expression statement)

- Eine Ausdrucks-Anweisung
 - ◆ nimmt eine Berechnung vor
 - ◆ besteht aus einem **Ausdruck**, der durch ein Semikolon abgeschlossen ist
- Es sind aber nur bestimmte Ausdrücke zugelassen
 - ◆ **Zuweisungsausdrücke** (mit **=**, **+=** etc.),
 - ⇒ `int i=5; x += 1;`
 - ◆ Präfix- oder Postfixformen von **++** oder **--**
 - ⇒ `i++;`
 - ◆ **Methodenaufrufe**
 - ⇒ `System.out.print("hello"); Math.sin(Math.PI);`
 - ◆ **Objekterschaffungen** mit **new**
 - ⇒ `new int[5];`

Deklarations-Anweisung (declaration statement)

- Deklarations-Anweisungen sind mit ; abgeschlossene Variablendeklarationen

[Modifikatoren] Typname Variablenname [= Initialisierung]

- Beispiele

int counter;	// Deklaration eines Zählers
int counter = 0;	// Deklaration mit Initialisierung
int start, end;	// Zwei integer-Variablen
double[] prices = { 100, 200, 300, -99.99 };	// Initialisierte Deklaration eines Double-Arrays
final float π = 3.14159265;	// Konstantendeklaration (final)

Kontrollfluss-Anweisungen: Blöcke

Blöcke

Lokale Deklarationen / Lokale Variablen

Speicherverwaltung auf dem Stapel

Kontrollfluss-Anweisungen (control flow statements)

- Blöcke (blocks)
 - ◆ Blöcke fassen durch Klammerung `{ }` Anweisungssequenzen zu einer einzigen Anweisung zusammen
- Verzweigungen (branches)
 - ◆ Verzweigungen veranlassen Übergänge zu anderen Anweisungen im Kontrollfluss
 - ⇒ bedingte Übergänge: `if`, `switch`
 - ⇒ unbedingte Übergänge: `break`, `continue`
- Iterationen (loops)
 - ◆ Iterationen organisieren strukturierte Wiederholungen im Kontrollfluss: `while`, `do while`, `for`

Blöcke

- Ein **Block** (**block**) ist eine Anweisung, die aus einer Folge von Anweisungen besteht, die durch **Begrenzer** zusammengefasst werden
 - ◆ In **C**, **C++** und **Java** sind die Begrenzer geschweifte Klammern **{ ... }**
 - ◆ In der **ALGOL-Familie** (incl. **Pascal**) sind die Begrenzer ein Paar **begin ... end**
- Der Block **{ }** stellt die leere Anweisung dar
- **Geschachtelte Blöcke** (**nested blocks**) sind möglich, also z. B. **{...{...}...}**
 - ◆ Wir sprechen von **äußeren** und **inneren Blöcken** (**outer / inner blocks**) und von der **Schachtelungstiefe** eines Blocks (**nesting depth**)

Blöcke mit lokalen Deklarationen

- In einem Block enthaltene Deklarationen werden als **lokale Deklarationen** dieses Blocks bezeichnet
 - ◆ Jeder Block kann lokale Deklarationen (von **Variablen** oder **Typen**) enthalten
 - ◆ In einem Block deklarierte Variablen sind **lokale Variablen** des Blocks.
- In einem Block deklarierte Namen sind **nur** in diesem Block und seinen geschachtelten Unterblöcken sichtbar
 - ◆ Dies dient vor allem der **Kapselung von Information**
 - ⇒ Deklaration und Gebrauch von Namen sollen nahe beisammen liegen, und die Deklaration eines Namens soll nicht den ganzen globalen Namensraum belasten
 - ◆ Somit können **verschiedene Blöcke** jeweils ihre **eigene Variable i** oder **x** haben, ohne sich gegenseitig zu stören
 - ⇒ In Java gibt es eine Ausnahme (siehe später)
- In Java dürfen in einem geschachtelten Block Namen aus einem umgebenden Block nicht erneut deklariert werden
 - ◆ **Verdecken** umgebender Deklarationen ist verboten (anders als in C++)
 - ⇒ **außer** wenn man Felder mit lokalen Variablen verdeckt (später)!

Vorlesung „Objektorientierte Softwareentwicklung“

Kapitel 1: Der imperative Teil von Java

There exist means of expressing the conditions under which these various processes are required to be called into play. It is not even necessary that two courses only should be possible. Any number of courses may be possible at the same time; and the choice of each may depend on any number of conditions.

Charles Babbage (1864)

Kontrollfluss-Anweisungen: Verzweigungen (branches)

Bedingte Übergänge: `if`, `switch`
Unbedingte Übergänge: `break`, `continue`

Bedingte Anweisungen: if – else

- Die einfache if-Anweisung hat die Form

```
if (condition) statement1
```

- Die allgemeine if-else-Anweisung hat die Form

```
if (condition) statement1 else statement2
```

- Dabei ist `condition` ein Boolescher Ausdruck
 - ◆ Falls `condition` zu true evaluiert wird `statement1` ausgeführt
 - ◆ Im Fall der if-else Anweisung wird sonst `statement2` ausgeführt
- Jedes `statement` ist eine Anweisung
 - ◆ Also evtl. ein Block oder wieder eine if-Anweisung, oder die leere Anweisung, etc.

Bedingte Anweisungen: Beispiele

Beispiel 6.8.6. Die Anweisung

```
if( a >= 0 )  
    res = a;  
else  
    res = -a;
```

belegt die Variable `res` durch den Absolutwert der Variablen `a`.

Beispiel 6.8.7. Die Anweisung

```
if( a <= 0 ) {  
    res = a;  
}  
else  
    if( a+b < 0 ) {  
        res = a+b;  
    }  
    else {  
        res = b;  
    }  
}
```

belegt die Variable `res` durch:

`res = a`, falls `a` negativ oder gleich 0 ist,
`res = a+b`, falls `a` größer als Null und `a+b` negativ ist und
`res = b`, falls `a` größer als Null und `a+b` positiv oder gleich Null ist.

Bedingte Anweisungen – switch-case-default

- Eine weitere Möglichkeit einer bedingten Anweisung ist die **Fallunterscheidung** (**switch – case – default**)

```
switch (c)
{
    case konst_1: {anweisung_1; break;}
    case konst_2: {anweisung_2; break;}
    :
    case konst_n: {anweisung_n; break;}
    default :    {anweisung_d;}
}
```

anweisung_i wird ausgeführt, falls *c == konst_i* gilt.

- Nur **Konstanten** erlaubt
und zwar vom Typ
char, byte, short, int,
Character, Byte, Short, Integer
oder einem Aufzählungstyp

- Die **break**-Anweisung am Ende jeder **case**-Zeile bewirkt das **sofortige Verlassen** der **switch**-Anweisung.
 - ◆ Falls das **break** fehlt, wird der nachfolgende **case**-Fall mit ausgeführt, ohne Test ob seine Bedingung zutrifft!

Bedingte Anweisungen – switch-case-default

Java Language Specification

- Beispiel (aus [JLS, S. 379]): In der Methode howMany() setzt jeder **case** mit dem Nächsten fort, da jegliche **break**-Anweisungen fehlen

```
class Toomany {  
    public static void howMany(int k) {  
        switch (k) {  
            case 1: System.out.print("one ");  
            case 2: System.out.print("too ");  
            case 3: System.out.println("many");  
        }  
    }  
  
    public static void test() {  
        howMany(3);  
        howMany(2);  
        howMany(1);  
    }  
}
```

Dementsprechend liefert der Aufruf

```
Toomany.test();
```

die Ausgabe

```
many  
too many  
one too many
```

Bedingte Anweisungen – switch-case-default

Beispiel

Beispiel 6.8.8. Dieses Programmstück zählt in der Variable `res1` die Anzahl der in einem Text vorkommenden a's, in `res2` die vorkommenden o's, in `res3` die vorkommenden u's und in `res4` die restlichen Buchstaben. Der Text ist in `EingabeText` enthalten, der den System-Datentyp `String` besitzt.

```
int index=0;
int res1 = 0, res2 = 0, res3 = 0, res4 = 0;
char c;
while( index < EingabeText.length() ) {
    c=EingabeText.charAt(index);
    index++;
    switch(c) {
        case 'a': res1++; break; ← Zähle a's
        case 'o': res2++; break; ← Zähle o's
        case 'u': res3++; break; ← Zähle u's
        default:  res4++;        ← Zähle alles andere
    } }
```


Bedingte Anweisungen – switch-case-default

Beispiel mit weggelassenem ‚break‘

Beispiel 6.8.9. Durch das Weglassen von `{anweisung; break;}` können mehrere case-Fälle zusammengefaßt werden: Das Programmstück

```
switch(c) {  
    case 0: case 1: case 2: case 3: { res = 1; break; }  
    case 4: case 5: case 8: case 9: { res = 2; break; }  
    case 6: case 7: { res = 3; break; }  
    default: { res = 0; }  
}
```

liefert:

res = 1, falls c den Wert 0, 1, 2 oder 3 hat,
res = 2, falls c den Wert 4, 5, 8 oder 9 hat,
res = 3, falls c den Wert 6 oder 7 hat und
res = 0 in allen anderen Fällen.

Schleifen-Anweisungen – while und do-while

Das Konstrukt

while (*condition*) *statement*;
führt *statement* aus, solange der
Ausdruck in *condition* zu **true** evaluiert.

Oftmals wird es sich bei *statement* um
einen Block handeln.

Die Anweisung

do *statement* **while** (*condition*);
führt *statement* aus und prüft danach
anhand von *condition*, ob der Schleifen-
durchgang wiederholt werden soll.

Bei do-Anweisungen ist
statement fast immer ein Block.

Die Anweisung

do *statement* **while** (!*condition*);
entspricht dem Konstrukt
do *statement* **until** (*condition*);
in anderen Sprachen.

repeat ... until
in Pascal

while-Schleife: Beispiel

Beispiel 6.8.11. Die folgende `while`-Schleife wird so lange ausgeführt, bis die Bedingung $(i < 10)$ nicht mehr erfüllt ist. Dieses Programmstück summiert in `res` die Zahlen von 1 bis 9 auf. Eine passende Schleifeninvariante ist $res = \sum_{j=1}^{i-1} j$. Zu Beginn gilt $0 = \sum_{j=1}^0 j$ und zum Schluß gilt $res = \sum_{j=1}^{10-1} j$.

```
{ int i = 1;
  int res = 0;
  while ( i < 10 )
  {
    res = res + i;
    i++;
  }
}
```

D.h. nach dem letzten Schleifendurchlauf

D.h. vor dem ersten Schleifendurchlauf

Die Schleife kann natürlich verallgemeinert werden, indem man den festen Wert 10 durch eine Variable `x` ersetzt; dann gilt zum Schluß $res = \sum_{j=1}^{x-1} j$. ❖

for-Schleife (1)

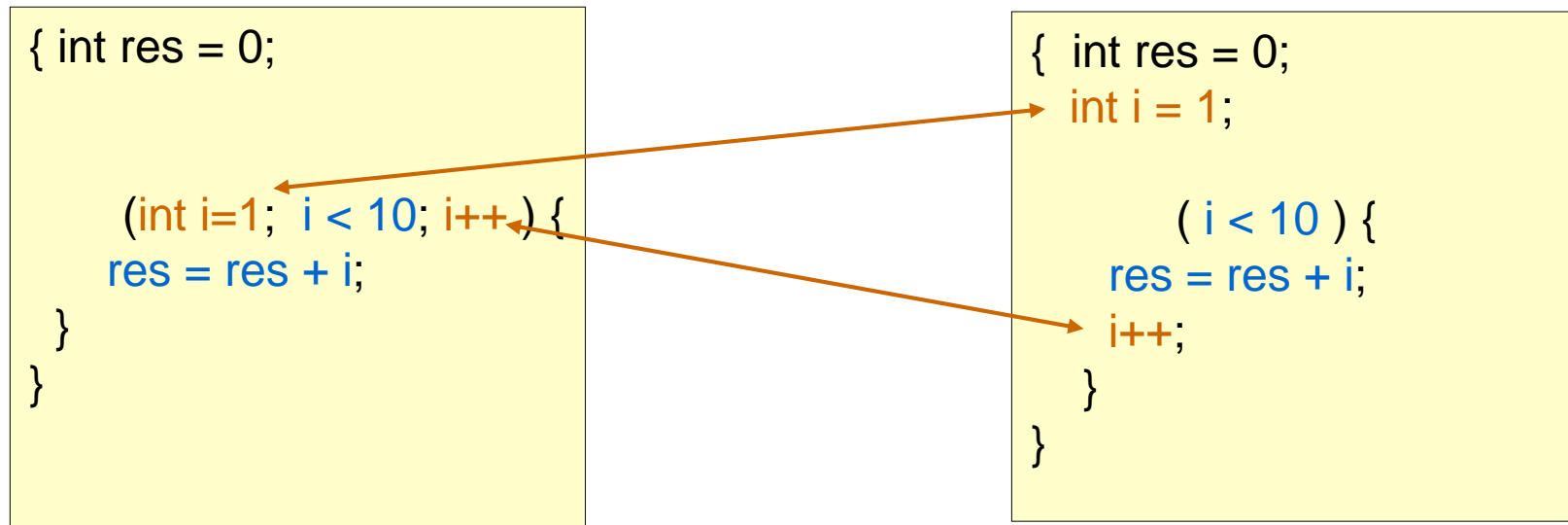
- Die Anweisung

```
(init; condition; increment) statement
```

entspricht

```
{ init;      (condition) {statement; increment ; } }
```

- Spezielle Syntax, da häufig vorkommend



for-Schleife (2)

- Der **init**-Teil kann die Deklaration einer Variablen enthalten
 - ◆ Diese Variable ist im Rumpf der for-Schleife **lokal**
- Der **increment**-Teil kann aus einer **durch Kommata getrennten Liste von Ausdrücken** bestehen, die von links nach rechts evaluiert werden
 - ◆ Dadurch besteht oft die Alternative, Code entweder im Schleifenrumpf oder im Kopf unterzubringen
 - ◆ Der Kopf sollte immer den Code enthalten, der die Kontrolle über die Schleifendurchgänge behält; der Rumpf sollte die eigentlichen Arbeitsanweisungen enthalten
- Jeder der drei Teile in einer for-Anweisung kann auch leer sein und zum Beispiel durch ein entsprechendes Konstrukt im Schleifenkörper ersetzt werden

for-Schleife (3)

- Beispiele für „guten“ und „schlechten“ Programmierstil

1. In der folgenden Schleife wird eindeutig klargemacht, daß `i` die Laufvariable ist und nur innerhalb der Schleife Bedeutung hat.

```
{ int res=0;
  for (int i=1; i<10; i++)
    res += i;
}
```

Guter Stil 😊

2. In der folgenden Schleife wird die „Arbeit“ `res+=i` im Kopf der Schleife verrichtet. Dies ist ebenso *schlechter Stil*, wie die Schleifenkontrolle im Rumpf zu erledigen.

```
{ int res=0;
  for (int i=1; i<10; res+=i,i++)
    ;
}
```

Syntaktisch korrekt
Schlechter Stil ☹️

for-Schleifen-Beispiel: Maximum eines Array

- Die folgende Schleife berechnet das Maximum der Werte eines Array **a** der Länge **len**

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

Angenommen **len** und **a** wurden passend initialisiert, z.B. als

```
int len=5;  
float[] a = {1.0, 3.7, 47.11, 0.815, 42};
```

- **Bemerkung**

- ◆ Mit **a.length** erhält man die Länge des Reihungsobjektes **a**.
- ◆ Die Schleifensteuerung der Art
for (int i=0; i < **a.length**; i++)
ist ein übliches **Idiom**. So braucht man keine separate Variable, in der die Länge des Arrays gespeichert ist.

Labels, break und continue

- Jedes Statement kann markiert werden:
 - ◆ *label: statement*
 - ◆ Sinnvoll i.d.R. nur bei Blöcken, Schleifen
- In Java
 - ◆ *kein* allgemeiner Sprung „goto label“
- In Java aber möglich
 - ◆ Sprung ans Ende eines Blocks
 - ◆ Sprung an Beginn oder ans Ende einer Schleife
 - ◆ Sprung aus einer Methode zurück an die aufrufende Methode

Labels und Break

- Unlabeled break statement: **break;**
 - ◆ beendet die unmittelbar umgebende **switch-**, **while-**, **do-** oder **for-Anweisung**
- Labeled break statement: **break label;**
 - ◆ beendet die umgebende **mit label gekennzeichnete** Anweisung
 - ◆ Beispiel: Teste, ob in einem Array von Arrays alle Werte positiv sind

```
{ double[][] a = ... // ein "2-dimensionales" Array
  boolean positive = true;
  search:
    for (int i=0; i<a.length; i++) {

        for (int j=0; j<a[i].length; j++)
            if (a[i][j] < 0.0) {
                positive=false;
                break search;
            }
    }
}
```

Es gibt in Java keine mehrdimensionalen Arrays, sondern nur Arrays von Arrays ... von Arrays!

a[i] ist ein eigenes Array, dessen Länge unabhängig von der anderer Elemente von a ist!

Labels und Continue

- Unlabeled continue statement: **continue;**
 - ◆ nur in **while-**, **do-**, **for-Anweisungen**
 - ◆ transferiert Kontrollfluss zur nachfolgenden Iteration **der gleichen Schleife**
- Labeled continue statement: **continue label;**
 - ◆ nur in **while-**, **do-**, **for-Anweisungen**
 - ◆ transferiert Kontrollfluss zur nachfolgenden Iteration **der mit „label“ gekennzeichneten Schleife**
- Markierte **break-** und **continue-statements** sind nützlich, wenn mehrere Schleifenebenen gleichzeitig übersprungen werden sollen

Methoden (Prozeduren und Funktionen)

[...] procedures are one of the most powerful features of a high level language, in that they both simplify the programming task and shorten the object code.

C. A. R. Hoare (1981)

Methoden

- **Methode** ist in objektorientierten Sprachen (incl. Java) der Sammelbegriff für **Prozedur** und **Funktion**
- In Java sind *syntaktisch* alle Methoden Funktionen
 - ◆ Sie haben immer einen Ergebnistyp
 - ◆ Allerdings bedeutet der vordefinierte Ergebnistyp **void**, dass **kein Ergebnis** zurückgeliefert wird
- In Java sind *konzeptionell* somit
 - ◆ **Prozeduren** → Methoden mit **void** als Ergebnistyp
 - ◆ **Funktionen** → allen anderen Methoden

Methoden: Prozeduren

- Methoden ohne Rückgabewert entsprechen Prozeduren
- Der fehlende Rückgabewert wird durch den Ergebnistyp **void** angezeigt

```
static void sayHelloTo(String name) {  
    System.out.println("Hello " + name);  
}
```

Konkatenation von Strings

Textausgabe auf Konsole

Methoden: Funktionen

- Erfordern die Deklaration eines Ergebnistyps
 - ◆ Java Methoden haben nur einen Rückgabewert
- Erfordern ein **return-Statement** am Ende eines jeden Kontrollflusspfades der die Methode „erfolgreich verlässt“
 - ◆ „Nicht erfolgreiches Verlassen“ einer Methode tritt im Fehlerfall auf
 - ⇒ Siehe später „Ausnahmen (Exceptions)“

```
public static long fakultät(int n)
{ if (n < 0) /* Fehlerbehandlung */ ;

  → if (n = 1) return 1;

  long result = 1;
  for (int i = 2; i <= n; i++) result = * i;
  → return ergebnis;
}
```

Parameterübergabe

- Die Parameterübergabe ist die Substitution der aktuellen Parameter eines Aufrufs für die formalen Parameter der aufgerufenen Operation
- Es gibt grundsätzlich mehrere Verfahren
 - ◆ Werteaufruf (call by value)
 - ◆ Referenzaufruf (call by reference)
 - ◆ Namensaufruf (call by name)
- Java unterstützt nur den Werteaufruf (call by value)
 - ◆ Jedoch sind ein grosser Teil der Werte Referenzen!
 - ◆ Namensaufruf gibt es aber definitiv nicht.

Keine Ergebnisparameter

- Java erlaubt **keine Ergebnisparameter**
 - ◆ Beispiel: Folgende Methode tut nichts sinnvolles, da die Parameteränderung an der Aufrufstelle nicht sichtbar wird.

```
static void bewege (int richtung, int position) {  
    switch (richtung) {  
        case LINKS:    position--; break;  
        case RECHTS:   position++;  
    }  
}
```

Passende Konstanten definiert, z.B:

```
static final int LINKS = 0;  
static final int RECHTS = 1;
```

- ◆ Test: Folgende if-Bedingung hat immer Erfolg, da **position** unverändert ist!

```
int position = 0;  
bewege (RECHTS, position);  
bewege (RECHTS, position);  
  
if (position == 0)  
    /* ... immer ausgeführt ...*/ ;
```


Ergebnisse nur per Return-Anweisung

- Ergebnisse sind ausschließlich Rückgabewerte die mittels einer **return-Anweisung** übergeben werden

- ◆ Beispiel: Sinnvolle Definition von bewege()

```
static int bewege (int richtung, int position) {  
    switch (richtung) {  
        case LINKS:    position--; break;  
        case RECHTS:   position++;  
    }  
    return position;  
}
```

Passende Konstanten definiert, z.B:

final int LINKS = 0;

final int RECHTS = 1;

- ◆ Test (vergleiche vorherige Seite):

```
int position = 0;  
position = bewege (RECHTS, position);  
position = bewege (RECHTS, position);  
  
if (position == 0)  
    /* ... nie ausgeführt ... */ ;
```

Methoden: Prozeduren und Funktionen

- Bisher

- ◆ Methoden als ungefähres Äquivalent von Prozeduren und Funktionen
- ◆ Java-spezifischen scheinbare Einschränkungen
 - ⇒ nur ein Ergebniswert, keine Ergebnisparameter

- Später (in Zusammenhang mit Objekten)

- ◆ Ausgleich obiger Einschränkungen
 - ⇒ Zusammenfassung mehrerer Werte in einem Objekt
- ◆ Methoden bieten mehr!
 - ⇒ **Dynamisches Binden** (**dynamic binding**)

Signatur einer Methode

- Die Signatur einer Methode (method signature)
 - ◆ besteht in Java aus deren **Name** und der Folge der **Parameter-Typen**
 - ◆ In anderen Sprachen gehört auch der **Ergebnistyp** dazu
- Nutzen der Signatur
 - ◆ Sie gibt die **Syntax des Aufrufs** wieder
 - ⇒ Sie heisst daher auch **Aufrufschnittstelle** (call interface)
 - ◆ Sie unterscheidet verschiedene gleich benannte Methoden
 - ⇒ „Überladen“ von Methodennamen (Method Overloading)

Überladen von Methodennamen

- Überladen von Methodennamen
 - ◆ Es kann im gleichen Gültigkeitsbereich mehrere Methoden gleichen Namens aber unterschiedlicher Signatur geben
- Folge aus Signaturdefinition ohne Resultattyp
 - ◆ Zwei Java-Methoden gleichen Namens können sich nicht lediglich um ihren Resultattyp unterscheiden

Überladen: Beispiele

Mathematische Notation

- Signatur
 - ◆ **plus: $\text{int} \times \text{int} \rightarrow \text{int}$**
- Legale weitere Signatur
 - ◆ **plus: $\text{float} \times \text{float} \rightarrow \text{float}$**
- **Illegale** weitere Signatur
 - ◆ **plus: $\text{int} \times \text{int} \rightarrow \text{float}$**



Nur Resultattyp
verschieden

Java-Syntax

```
public static int plus( int a, int b ) {  
    return a+b;  
}
```

```
public static float plus( float a, float b ) {  
    return a+b;  
}
```

```
public static float plus( int a, int b ) {  
    return (float) (a+b);  
}
```

Überladen: Nutzen

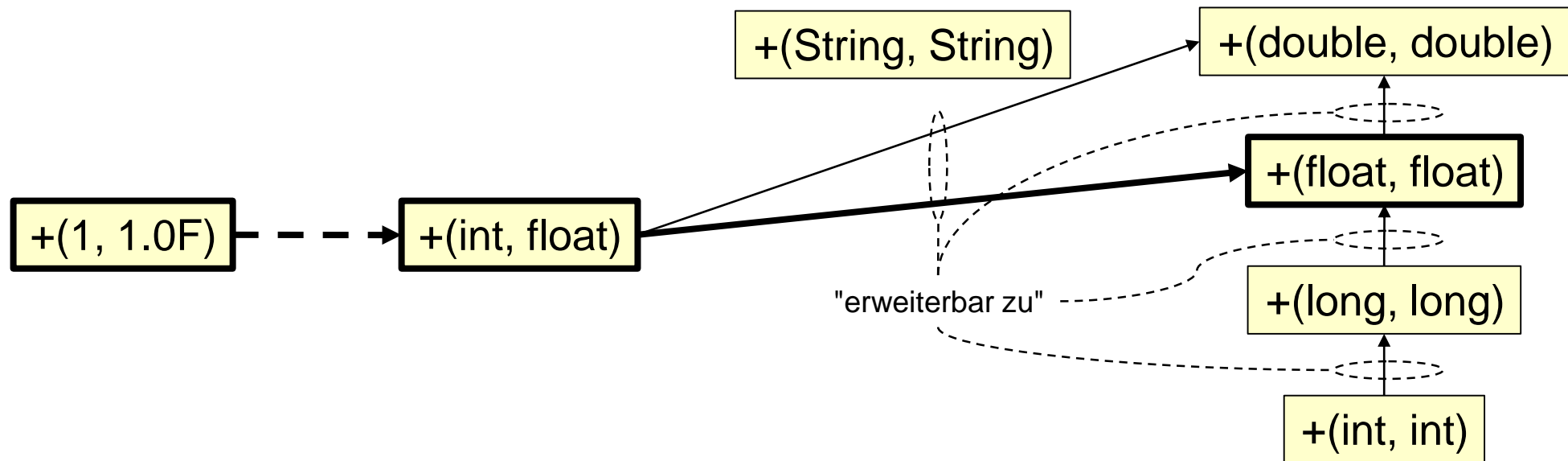
- Einfachere Notation, bessere Lesbarkeit
 - ◆ Kein Zwang neu Namen zu erfinden, wenn gleiche Operation auf verschiedene Parametertypen anwendbar ist
- Beispiel: Arithmetischen Operatoren in Java
 - ◆ Möglich
 - ⇒ `1 + 1`
 - ⇒ `1.0 + 1.0`
 - ⇒ `"abc" + "def"`
 - ◆ Anstatt
 - ⇒ `1+1`
 - ⇒ `addFloat(1.0, 1.0)`
 - ⇒ `concatStrings("abc", "def")`

Überladen: Bestimmung der aufzurufenden Methode

Aufruf

Aufrufsignatur

Hierarchie der Methodensignaturen



- Prinzip: Es wird die **spezifischste Methodensignatur** bestimmt, zu der sich die **Aufrufsignatur** erweitern lässt

Bestimmung der aufzurufenden Methode: Aufrufsignatur

- Die Signatur die man erhält, wenn man für einem Aufruf die **statisch** bekannten Parametertypen einsetzt

```
public class OverloadingTest {  
    public static void main(String args[]) {  
        int i=1;  
        float f=1.0f;  
  
        long l = myFunction(i+f);  
    }  
  
    static float myFunction(float f) {  
        return 1.0f;  
    }  
  
    static long myFunction(long l) {  
        return l;  
    }  
}
```

1. Die Aufrufsignatur von **i+f** ist **int+float**

2. Die spezifischste anwendbare Operation für **i+f** ist somit **float+float**

3. Der Ergebnistyp von **float+float** ist **float**

4. Die Aufrufsignatur von **myFunction(i+f)** ist somit **myFunction(float)**

5. Die spezifischste anwendbare Operation für **myFunction(i+f)** ist somit **myFunction(float)**

6. Der Ergebnistyp von **myFunction(float)** ist **float**.


7. Die Zuweisung **long = float** ist **illegal!**

Bestimmung der aufzurufenden Methode: Signaturerweiterung

- Informelle Definition

- ◆ Die Signatur **Sig** ist erweiterbar zur Signatur **SigE**, falls beide gleiche Namen haben und für jede Parameterposition gilt, dass der entsprechende Paramtertyp von **Sig** ein **Subyp** des Parametertyps von **SigE** ist

- Formale Definition

- ◆ $\text{name}(T_1, \dots, T_n) \subseteq \text{name}(T'_1, \dots, T'_n) :\Leftrightarrow \forall i=1..n: T_i \subseteq T'_i$


- Notation

- ◆ $\text{Sig} \subseteq \text{SigE}$ wird gelesen als **Sig ist erweiterbar zu SigE**
bzw. **Sig ist ein (Signatur-)Subtyp von SigE**

Signaturerweiterung: Beispiele

Vergleichbar anhand \subseteq

- $+(int,int) \subseteq +(long,long)$

Nicht vergleichbar anhand \subseteq

- $+(int,int) \not\subseteq +(String,String)$
- $+(String,String) \not\subseteq +(int,int)$
- $f(int,double) \not\subseteq f(double,int)$
- $f(double,int) \not\subseteq f(int,double)$

Bestimmung der aufzurufenden Methode: Spezifischste Signatur

- Spezifischster Typ
 - ◆ Subtyp aller damit verglichenen Typen
- Spezifischste Signatur
 - ◆ Signatur-Subtyp aller damit verglichenen Signaturen
- Problem: Mehrdeutigkeit
 - ◆ Manchmal gibt es unter den Methodensignaturen zu denen einer Aufrufsignatur erweiterbar ist keine Spezifischste
 - ◆ Beispiel
 - ⇒ Aufruf: *f(1, 1)*
 - ⇒ Aufrufsignatur: *f(int, int)*
 - ⇒ Definierte Methodensignaturen: *f(int, double)* und *f(double, int)*
 - ⇒ Die Aufrufsignatur ist auf beide Methodensignaturen erweiterbar, aber keine davon ist spezifischer als die andere

Überladen: Mögliche Fehler

- Der Compiler meldet für einen Operationsaufruf einen Fehler wenn
 - ◆ es **keine Methodensignatur** gibt, zu der man die Aufrufsignatur erweitern kann (**undefinierter Aufruf**) oder
 - ◆ es **keine spezifischste Methodensignatur** gibt (**mehrdeutiger Aufruf**)

Zusammenfassung

Besprochene Imperative Konzepte und ihre Ausprägung in Java

Bisher ausgesparte Themen

Ausblick

Java bietet die meisten Konzepte imperativer Sprachen – gegebenenfalls leicht angepasst

- Deklarationen

- ◆ Typen → bisher nur elementare Typen
- ◆ Variablen → Konstanten per Modifikator
- ◆ Funktionen, Prozeduren → Methoden
- ◆ Überladene Fkt. / Proz. → Überladene Methoden
- ◆ Programme → Klassen

Java bietet die meisten Konzepte imperativer Sprachen – gegebenenfalls leicht angepasst

● Ausdrücke

- ◆ Literale → viele elementare Datentypen
- ◆ Variablennamen → wie gewohnt
- ◆ Funktionsaufrufe → bisher nur "call by value"
- ◆ Operatoraufrufe → Vielzahl an Operatoren incl. vieler Zuweisungvarianten!
- ◆ Typkonversionen → Subtypbegriff, implizite und explizite Konversion
- ◆ Auflösung von Aufrufen → Aufrufsignatur, spezifischste Kandidatensignatur

Java bietet die meisten Konzepte imperativer Sprachen – gegebenenfalls leicht angepasst

- Anweisungen

- ◆ Blöcke → Deklarationen + Anweisungen
- ◆ Fallunterscheidungen → if, if-else, switch-case-default
- ◆ Schleifen → while, do-while, for
- ◆ Kontrollierte Sprünge → break, continue
- ◆ Prozeduraufrufe → void-Methoden

Was wir bisher ignoriert haben → Ausblick

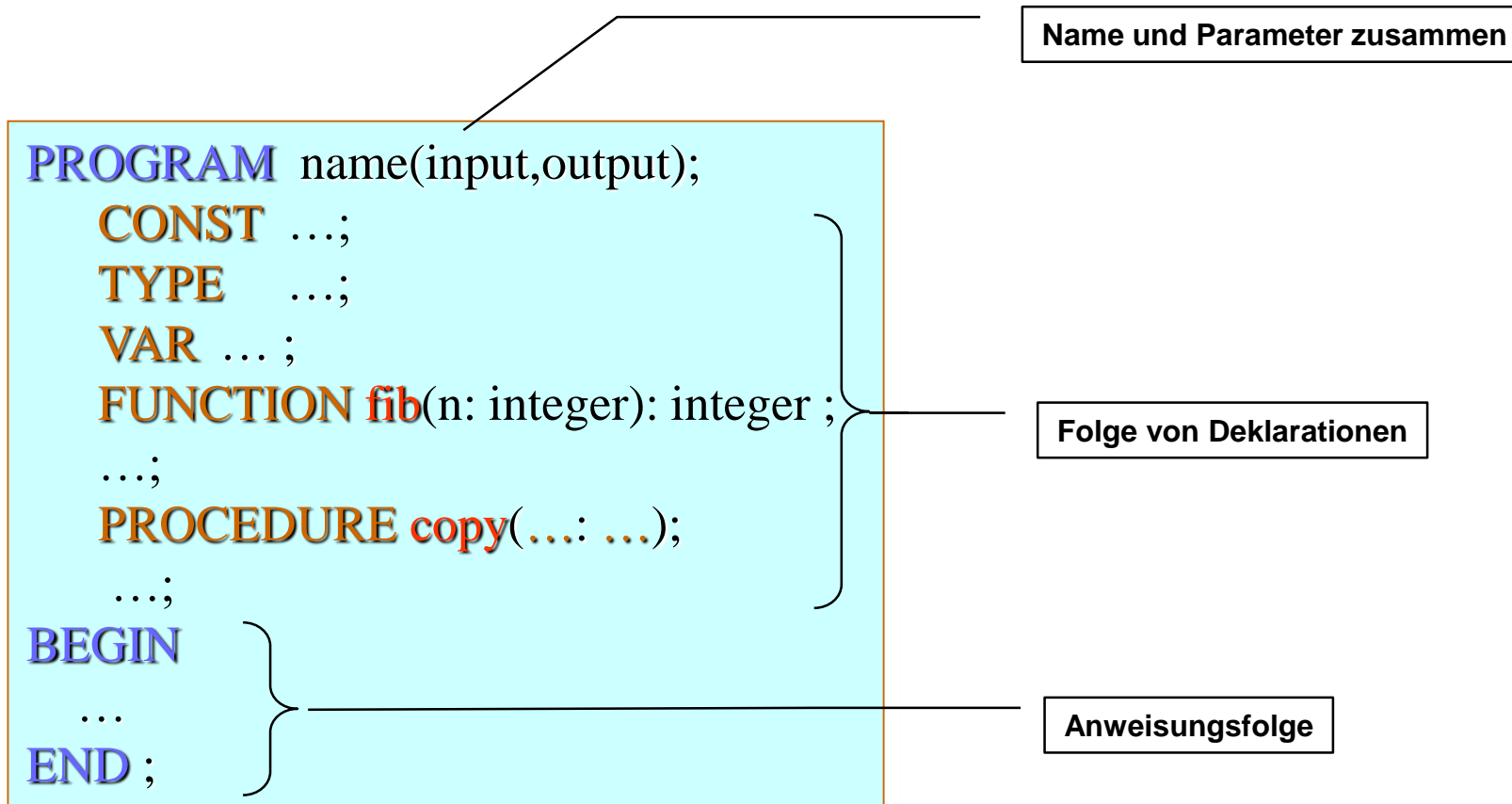
- Implementierungsaspekte imperativer Sprachen
 - ◆ Übersetzung / Interpretation von Programmen
 - ◆ Speicherverwaltung bei Eintritt in Blöcke (insbesondere bei Methodenaufrufen)
 - Essentielles holen wir nach, sofern die Zeit reicht
- Praktische Aspekte des Umgangs mit Java
 - ◆ Die JVM, das JDK, ...
 - Kommt noch
- Jegliche objektorientierten Konzepte
 - ◆ ... und ihre Auswirkungen auf die imperativen Anteile
 - Fokus der weiteren Foliensätze

Programme

Klassen

Die "main" Methode

Programme in Pascal



- Das "Highlander-Prinzip": "Es kann nur einen geben!"
 - ◆ Alle Deklarationen befinden sich im gleichen Gültigkeitsbereich
 - ◆ Probleme: Komplexität! Verständlichkeit! Änderungen! Wartbarkeit! Qualität!

Komplexität

- Systeme mit einigen zigtausend Codezeilen gelten als klein
- Moderne Systeme umfassen schnell einige hunderttausend oder Millionen Zeilen Code
 - ◆ Der Eclipse-Kern umfasst ca. 1 Mio. Codezeilen Java
 - ◆ Die Java-Entwicklungsumgebung in Eclipse ist sogar noch größer
- Von Anlagensteuerung, Bankanwendungen, Flugzeugsteuerung, ... wollen wir gar nicht erst sprechen

Verständlichkeit

- Wie behalte ich hier die Übersicht, wenn alle das in **einem** großen Gültigkeitsbereich liegt?
 - ◆ Welche Variablen / Typen gehören zusammen?
 - ◆ Welche Funktionen / Prozeduren gehören zusammen?
 - ◆ Welche Funktionen / Prozeduren arbeiten auf welchen Daten?
 - ◆ ...

Änderungen

- Programme sind selten von vornherein komplett spezifiziert, sondern Anforderungen ändern sich!
 - ◆ Neue Marktsituation
 - ◆ Neue Produkte
 - ◆ Neue Gesetze
 - ◆ Neue Ideen
 - ◆ Neue Technologiebasis
- Änderungen sind die einzige Konstante!
 - ◆ Änderungen schon während der Produktentwicklung
 - ◆ Änderungen im laufenden Betrieb
- Kosten für Softwarewartung macht 80% der Gesamtkosten aus!

Wartbarkeit

- Wie stelle ich sicher, dass ich nichts "kaputt mache" wenn ich etwas ändere?
 - ◆ Darf ich diese Variable umbenennen?
 - ◆ Darf ich ihren Typ ändern?
 - ◆ Darf ich sie durch eine Methode ersetzen, die den Wert berechnet?
 - ◆ Darf ich das Verhalten dieser Funktion geringfügig ändern?
 - ◆ Wie bringe ich neue Varianten von bestehendem Code ein ohne alles zu kopieren?
 - ◆ Wie stell ich sicher, dass Änderungen möglichst lokal begrenzt sind?
- Konsequenzen, wenn man obiges nicht kann
 - ◆ Hohe Kosten
 - ◆ Schlechte Qualität
 - ◆ ... oder auch beides!

Lösungsansatz: Datentypen und Module

- Konzept: Ein **Konkreter Datentyp** ist eine Einheit aus
 - ◆ Daten
 - ◆ Signaturen der darauf arbeitenden Prozeduren / Funktionen
 - ◆ Implementierung der Prozeduren / Funktionen
- Syntax
 - ◆ Ein **Modul** ist ein syntaktisch kenntlicher Gültigkeitsbereich von Deklarationen, der die Daten und Prozeduren eines konkreten Datentyps zusammenfasst
- Motivation
 - ◆ Bessere **Verständlichkeit** und **Wartbarkeit** durch Aufteilung eines Programs in Module

Module in Java

```
public class Auto {  
  
    static int anzahlRäder = 4;  
  
    static float tankinhalt() {  
        // ...  
    }  
  
    static void anlassen(/*...*/) {  
        // ...  
    }  
  
}
```

Name ohne Parameter!

Folge von Deklarationen

- Jede Klasse ist ein Modul!
- Klassenvariablen und Klassenmethoden
 - ◆ Der Modifikator „**static**“ gibt an, dass die damit markierten Variablen und Methoden zum jeweiligen Modul gehören.

Heißen auch
„**Klassenmitglieder**“
(class members)

Programme in Java

```
public class Fahrrad {
```

```
public class Auto {
```

```
    static int anzahlRäder = 4;
```

```
    static float tankinhalt() {  
        //...
```

```
    }
```

```
    static void anlassen(/*...*/) {  
        // ...
```

```
    }
```

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

```
}
```

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

Ein Java Programm kann
beliebig viele Klassen
enthalten!

Jede Klasse **kann** eine
main()-Methode enthalten!

Jede main()-Methode ist ein
eigenes Hauptprogramm!

Die main()-Methode ersetzt die
Anweisungsfolge und die
Parameter eines Pascal-
Programms

Zusammenfassung „Programme“

- Ein Programm ist eine Menge von Klassen
 - ◆ Evtl. muss man die Umgebungsvariable CLASSPATH setzen, damit der Übersetzer all zusammengehörigen Klassen findet!
- Der Einstiegspunkt in ein Programm ist eine Klasse, die eine main()-Methode enthält
 - ◆ Meist gibt es genau eine solche Klasse.
 - ◆ Weitere Main-Methoden werden manchmal als ad-hoc Tests der jeweiligen Klassen genutzt
- Die Signatur der Main-Methode ist genau festgelegt!

```
public static void main(String[] args)
```

Namensräume / Pakete

Paketdeklaration

Vollständig qualifizierte Namen

Import-Deklaration

Sonderfälle

Gültigkeitsbereiche

- **Klassen-Mitglieder** (Felder und Methoden) sind nur **lokal** innerhalb der Klasse bekannt
 - ◆ Keine Namenskollisionen mit Mitgliedern anderer Klassen 😊
- **Klassen** sind **global** bekannt
 - ◆ Namenskollisionen möglich, vor allem bei großen ☹️
- **Problem**: Wie kann man mehrere Versionen eines Datentyps (= Klasse) im gleichen Programm verwenden?
 - ◆ Mein Programm soll durch die Integration einer neuen Bibliothek nicht plötzlich (wegen Namenskollisionen) fehlerhaft sein!
- **Idee**: Namensräume!

Pakete und Namensräume

- Namensräume sind vor allem bei großen Softwaresystemen wichtig
 - ◆ In denen mehrere Entwickler zusammenarbeiten
 - ◆ „Fremder“ (oder „alter“) Code integriert werden soll
- Ohne Namensräume Gefahr groß, dass in einem anderen Programmteil der gleiche Name (für eine Klasse) schon gebraucht wurde
 - ◆ Konzept der Namensräume wurde daher inzwischen auch in C++ eingeführt

Namensräume / Pakete (Namespaces / Packages)

- Package = Namensraum

- ◆ Gültigkeitsbereich für Klassennamen

- Importmechanismus

- ◆ Namen aus anderen Paketen in aktueller Datei bekannt machen
- ◆ Implizit: `*` (alles importieren)
- ◆ Explizit: **Klassenname**
 - ⇒ empfohlen!

- Entsprechungen

- ◆ Klasse \approx Datei
 - ⇒ mehrere Klassen pro Datei möglich, aber nur eine "öffentliche" (public)
- ◆ Mit `.` abgetrennter Teil des Paketnamens \approx Ordner
 - ⇒ oft verwendet, aber nicht zwingend (Ausnahmen: Klassen in ZIP-Archiven, Datenbanken, ...)

```
package java.util;  
public class Hashtable {  
    .. package java.util;  
    public class Vector {  
        ...  
    }  
}
```

```
import java.util.Vector;  
class Test {  
    Vector v;  
}
```

Zugriff auf Klassen in anderem Paket

- Immer möglich durch „vollständige Qualifizierung“ des Klassennamens
 - ◆ Mit Paketnamen als Präfix
 - ◆ Z.B. `java.util.Vector`
- Nach Importieren des Klassennamens kann er ohne vollständige Qualifizierung verwendet werden
 - ◆ „Intern“ wird aber immer der vollständig qualifizierte Namen verwendet

```
class Test1 {  
    java.util.Vector v;  
}
```

```
import java.util.Vektor;  
class Test2 {  
    Vektor v;  
}
```


Sonderfall „java.lang“ und „default“

- „java.lang“ Paket

- ◆ Die Klassen in diesem Paket müssen nicht explizit importiert werden.
- ◆ Sie werden vom Übersetzer implizit importiert.
- ◆ Jeder Datei wird automatisch eine Anweisung der Form `import java.lang.*;` vorangestellt.

- „default“ Package

- ◆ Alle Klassen ohne Paketdeklaration liegen implizit in dem namenlosen „default“ package.
- ◆ Das ist schlechter Stil! Verwenden Sie immer sinnvolle Paketnamen!

Kapselung (Encapsulation)

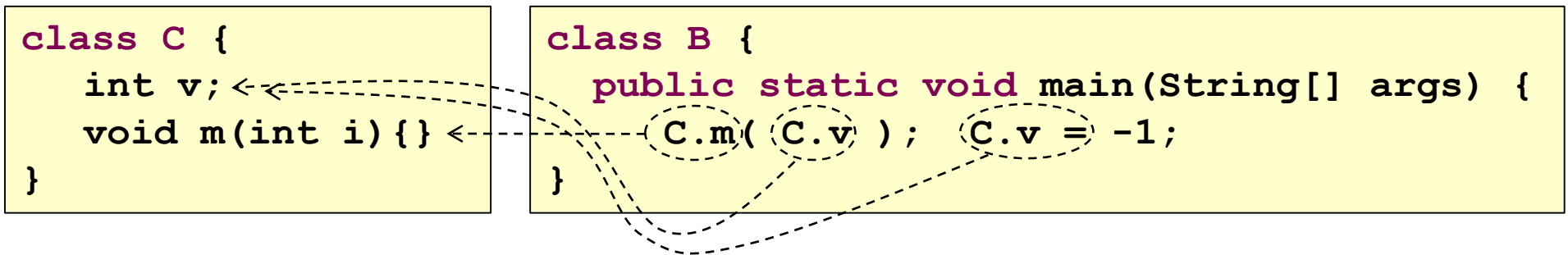
Problem: Abhängigkeiten → Schlechte Wartbarkeit

Idee: Information Hiding / Kapselung

Sichtbarkeiten: Private, Public, Package

Zugriff auf Klassen

- Zugriff auf Klassenvariable **v** in Klasse **C**: **C.v** bzw. **C.v = ...;**
- Zugriff auf Klassenmethode **m()** in Klasse **C**: **C.m()**



- **Problem:** Jede Klasse kann
 - ◆ die Daten einer jeden anderen unkontrolliert verändern
→ **Fehleranfälligkeit** ☹️
 - ◆ sich von internen Implementierungsdetails Anderer abhängig machen
→ **Schlechte Wartbarkeit** ☹️
- **Idee:** Sichtbarkeit einschränken!

Kapselung und Zugriffskontrolle

- Jeder Name eines Mitglieds (Attribut oder Methode) einer Klasse hat einen **Sichtbarkeitsbereich** (scope)
 - ◆ Ein Mitgliedsname kann nur dort verwendet werden, wo er sichtbar ist
- Zugriffe auf das Mitglied unterliegen damit einer **Zugriffskontrolle** (access control)

Kapselung und Zugriffskontrolle

- Der Sichtbarkeitsbereich eines Mitglieds wird durch einen der folgenden **Moderatoren** (access modifiers) angegeben:
 - ◆ **public** (öffentlich, global)
 - ◆ **private** (eigene Klasse)
 - ◆ **protected** (geschützt)
 - ⇒ In Java: **kein zusätzlicher Schutz** sondern Ausweitung des Zugriffs auf alle von dieser Klasse abgeleiteten Klassen
 - ◆ Ist nichts angegeben, gilt der **Standard-Sichtbarkeitsbereich** (default scope) „Paket“

Kapselung und Zugriffskontrolle: Beispiel

```
// the class named Date is public:

public class Date {
    private byte day;      // private field
    private byte month;    // private field
    private short year;    // private field

    public void m1() {}    // public method

    void m2() {}           // package method

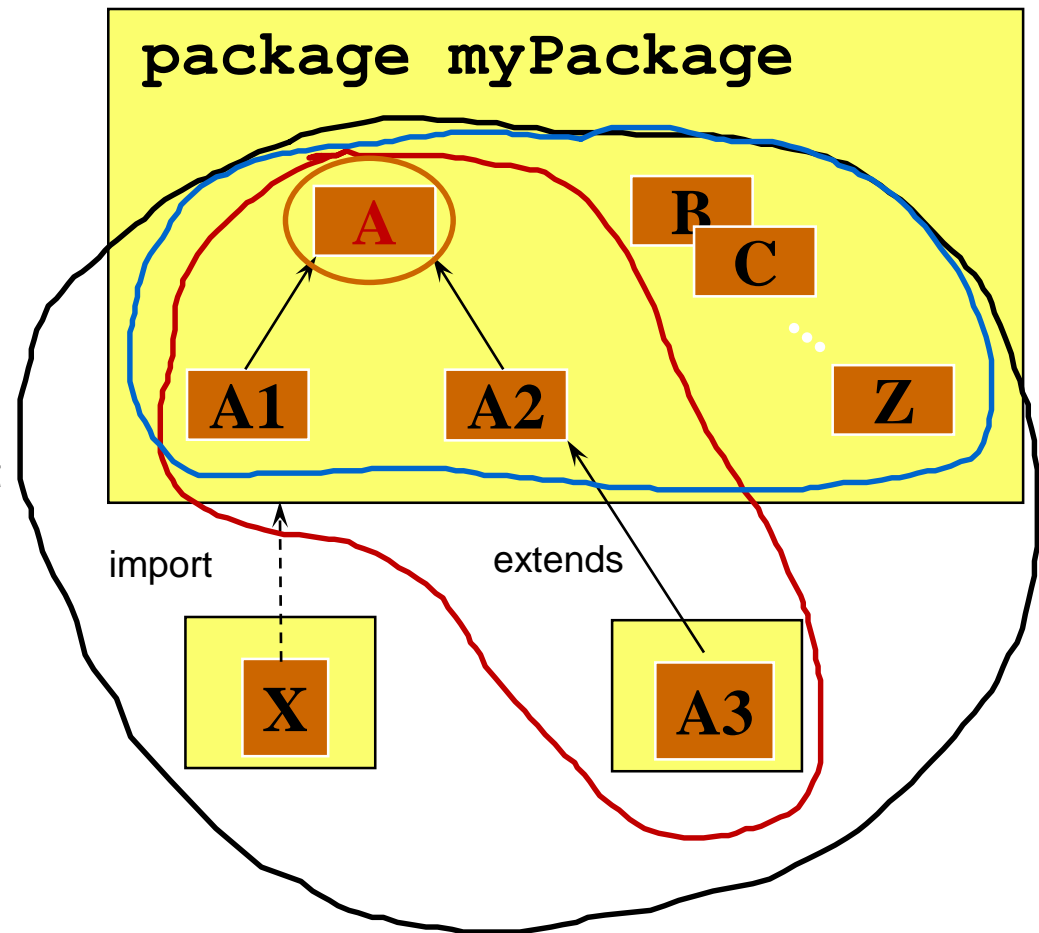
    private boolean m3()  // private method
    {}

}
```

OOP: Zugriffskontrolle (2)

Beispiel: Sichtbarkeitsbereich einer Methode aus Klasse **A**

- ➡ **public** - für alle
- ➡ **protected** - für Unterklassen
- ➡ **private** - nur für A
- ➡ - für alle Klassen im Paket
↕
(implizit, wenn nichts
anderes angegeben)



OOP: Sichtbarkeitskontrolle (1)

● Für Klassen

- ◆ Ist Klasse **C** außerhalb des Package **pkg** bekannt? Ist Zugriff **pkg.C** erlaubt?

- ⇒ Klasse ist „public“: ja
- ⇒ Keine Angabe: nein

```
import pkg.C;  
...
```

```
package pkg;  
public class C {...  
};
```

● Für Methoden + Variablen

- ◆ Ist Methode/Variable **mv** außerhalb der Klasse **C** bekannt? Ist der Zugriff **C.mv** erlaubt?

- ⇒ Methode Variable ist ...

```
class D {  
    ... C.mv ...  
};
```

```
class C {  
    public mv;  
};
```