Einführung in die Informatik II

SS 2012

4 Java Grundlagen

Zu meiner Person - Alexander Holland

- Studium der Informatik an der Universität Dortmund
- Promotion an der Universität Siegen 2008
- Seit 2005 wiss. Mitarbeiter für Wissensbasierte Systeme und Wissensmanagement an der Universität Siegen
- Forschung: Computational Intelligence, Graphische Modellbildung, Entscheidungstheorie, Entscheidungsunterstützungs-Systeme, Verteilte Systeme, Soft Computing Anwendungen, Relationship Discovery im Wissensmanagement
- E-mail: alex@informatik.uni-siegen.de
- Web: http://www.uni-siegen.de/fb12/ws/mitarbeiter/
- Tel.: 0271/740-2276 Büro: H-A 8413
- Sprechstunde: Montag, 16:15 - 17:15 Uhr

4 Java Grundlagen …

Lernziele

- Programme in Java erstellen, übersetzen und ausführen können
- Objekte in Java erzeugen und nutzen können
- Sprachkonstrukte von Java kennen und beherrschen

Literatur

- [Ba99], Kap. 2
- [BK03], Kap. 3, 4, 8-11
- sowie weitere Java-Bücher ...

4 Java Grundlagen …

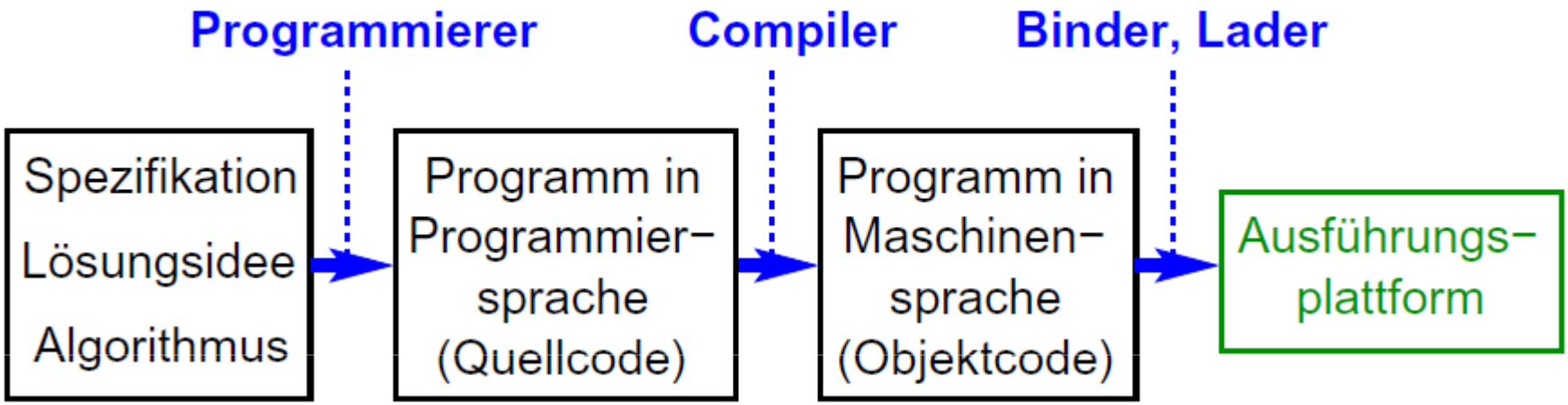
Inhalt

- Erstellen und Ausführen von Java-Programmen
- Beschreibung von Programmiersprachen
- Syntaktische Grundelemente in Java
- Anweisungen
- Typen und Variablen
- Objekte und Methoden
- Arrays und Strings
- Ausnahmebehandlung

} Für Erstsemester bzw. zur Wiederholung

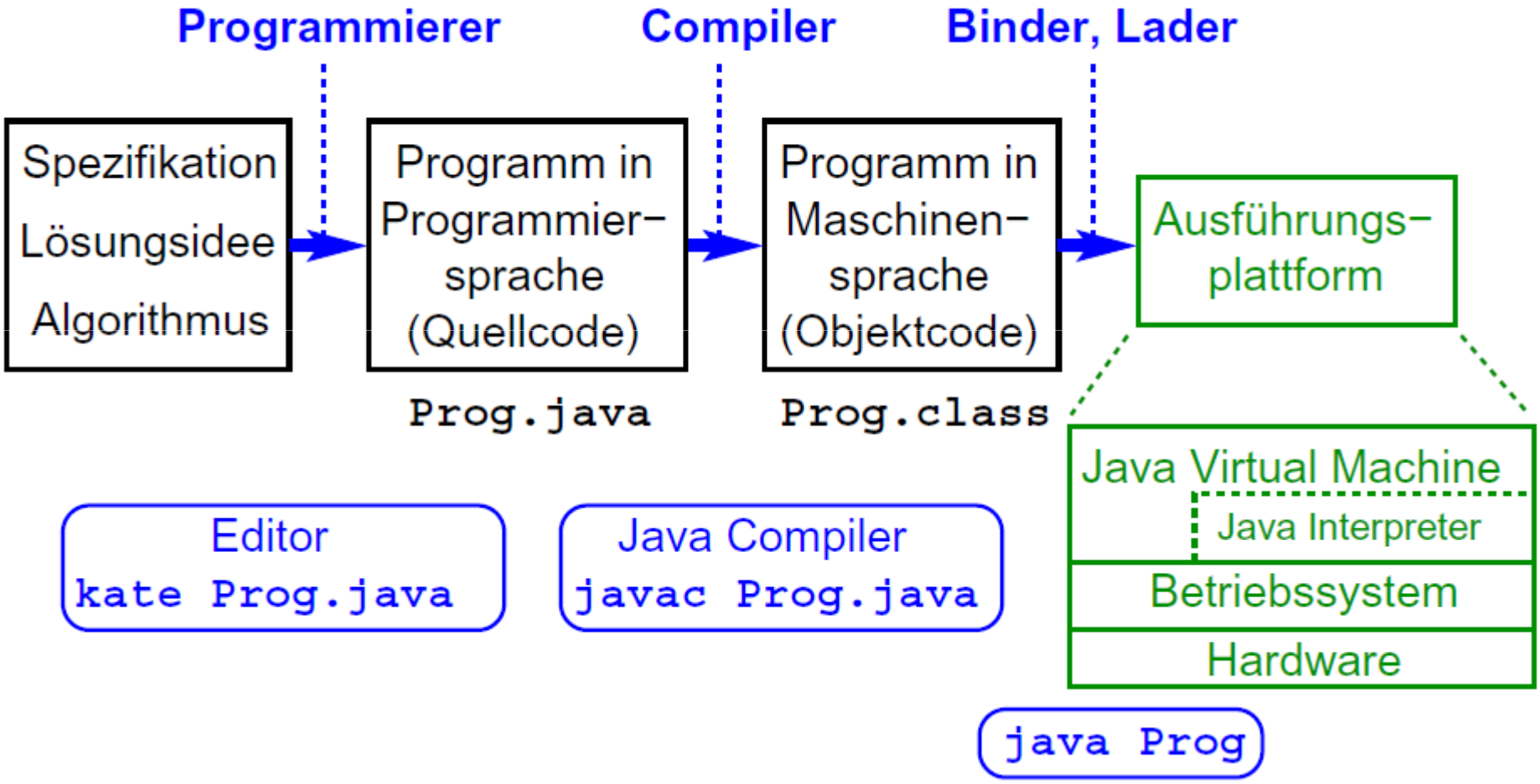
4.1 Erstellen und Ausführen von Java-Programmen

Entstehungsschritte eines Programms:



4.1 Erstellen und Ausführen von Java-Programmen

Entstehungsschritte eines Programms:



4.1 Erstellen und Ausführen von Java-Programmen

Programmstruktur

- Jedes Java-Programm besteht aus mindestens einer Klasse
- Der Programmcode einer öffentlichen Java-Klasse steht in einer eigenen Quelldatei, die den Namen der Klasse trägt
  - private Klassen können in beliebigen Dateien stehen
- Eine Java Quelldatei hat die Endung `.java`

Anmeldung.java

```
public
class Anmeldung
{
   ...
}
```

Student.java

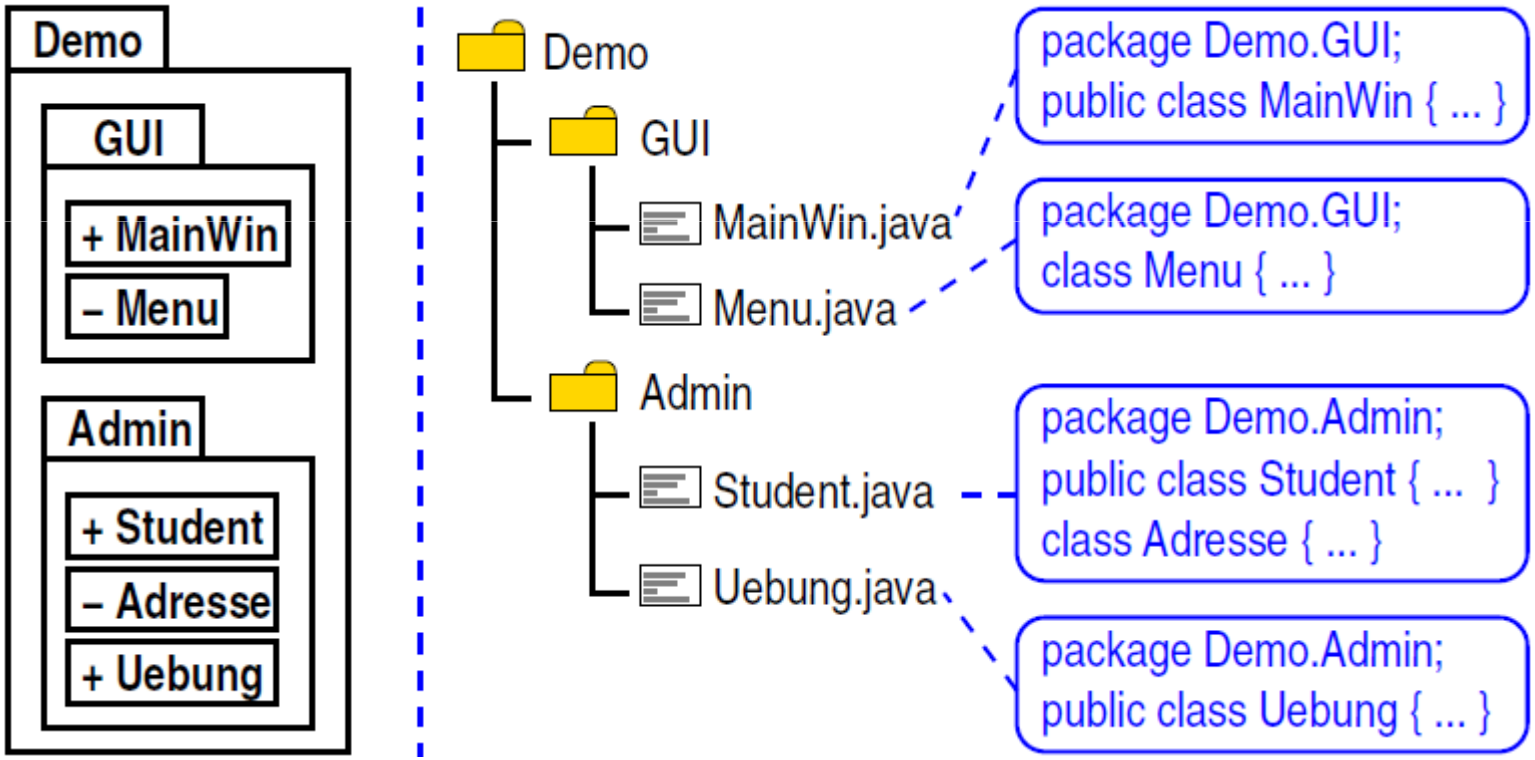
```
public
class Student
{
   ...
}
```

Hoersaal.java

```
public
class Hoersaal
{
   ...
}
```

Programmstruktur und Pakete

- Die Verzeichnisstruktur von Java-Quelldateien sollte der Paketstruktur entsprechen:



4.1 Erstellen und Ausführen von Java-Programmen

Übersetzung

- Aufruf des Java-Compilers: `javac <name>.java`
  - Compiler versucht auch, alle weiteren benötigten Quelldateien zu übersetzen
  - ggf. alle Dateinamen angeben, z.B.: `javac *.java`
- Bei Verwendung von Paketen:
  - Java-Compiler im Wurzel-Verzeichnis starten
    - z.B. `javac Demo/GUI/MainWin.java`
  - oder: Option `-classpath` nutzen, um Wurzelverzeichnis anzugeben
    - z.B. `javac -classpath /home/meier/code Uebung.java`
- Compiler erzeugt für jede Klasse eine `.class`-Datei

Ausführung von Java-Programmen

- Interpretation durch eine virtuellen Maschine
  - JVM: *Java Virtual Machine*
  - sie liest bei Bedarf `.class`-Dateien ein und arbeitet bei Operations-Aufrufen den Programmcode der Operation ab

- Java-Programme werden also nicht direkt vom Prozessor des Rechners ausgeführt
  - Vorteile: Portabilität und Sicherheit
    - Java-Code läuft auf jedem Rechner, unabhängig von Prozessortyp und Betriebssystem
    - die JVM kann Zugriffe des Programms auf Ressourcen des Rechners (z.B. Dateien) einschränken
  - Nachteil: geringere Ausführungsgeschwindigkeit

4.1 Erstellen und Ausführen von Java-Programmen

Starten eines Java-Programms

- Das Kommando `java <Klassenname>` startet eine JVM
  - die JVM lädt zunächst die angegebene Klasse
  - anschließend versucht sie, die Methode
    `public static void main(String[] args)`
    auszuführen
  - existiert diese Methode nicht, erfolgt eine Fehlermeldung
  - falls die Klasse in einem Paket liegt, muß der vollständige Name angegeben werden, z.B. `java MyPacket.MyClass`
- Bei Bedarf lädt die JVM während der Programmausführung weitere Klassen nach
  - damit diese gefunden werden, muß ggf. ein Klassen-Pfad (*Classpath*) definiert werden

4.1 Erstellen und Ausführen von Java-Programmen

Java-Klassenpfad

- Wird vom Compiler und der JVM benutzt, um den Code von benötigten Klassen zu finden
- Besteht aus einem/mehreren Verzeichnissen oder Java-Archiven
  - Trennzeichen ':' (Linux) bzw. ';' (Windows)
  - z.B.: `java -classpath /lib/classes:. MyPkt.MyClass`
    - sucht in `/lib/classes` und im aktuellen Verzeichnis
    - die Datei `MyClass.class` muß sich in einem Unterverzeichnis `MyPkt` befinden!
  - z.B.: `java -classpath /lib/myCode.jar MyClass`
    - sucht nur im Java-Archiv `/lib/myCode.jar`
    - ein Java-Archiv enthält einen kompletten Dateibaum (ggf. mit mehreren Paketen / Klassen)

Java-Klassenpfad ...

- Der Klassenpfad kann auch über eine Umgebungsvariable gesetzt werden:
  - Beispiel:
    - `export CLASSPATH=/lib/classes:.` (Linux)
    - `set CLASSPATH=D:\lib\classes;.` (Windows)
- Der so definierte Klassenpfad gilt sowohl für den Compiler als auch die JVM

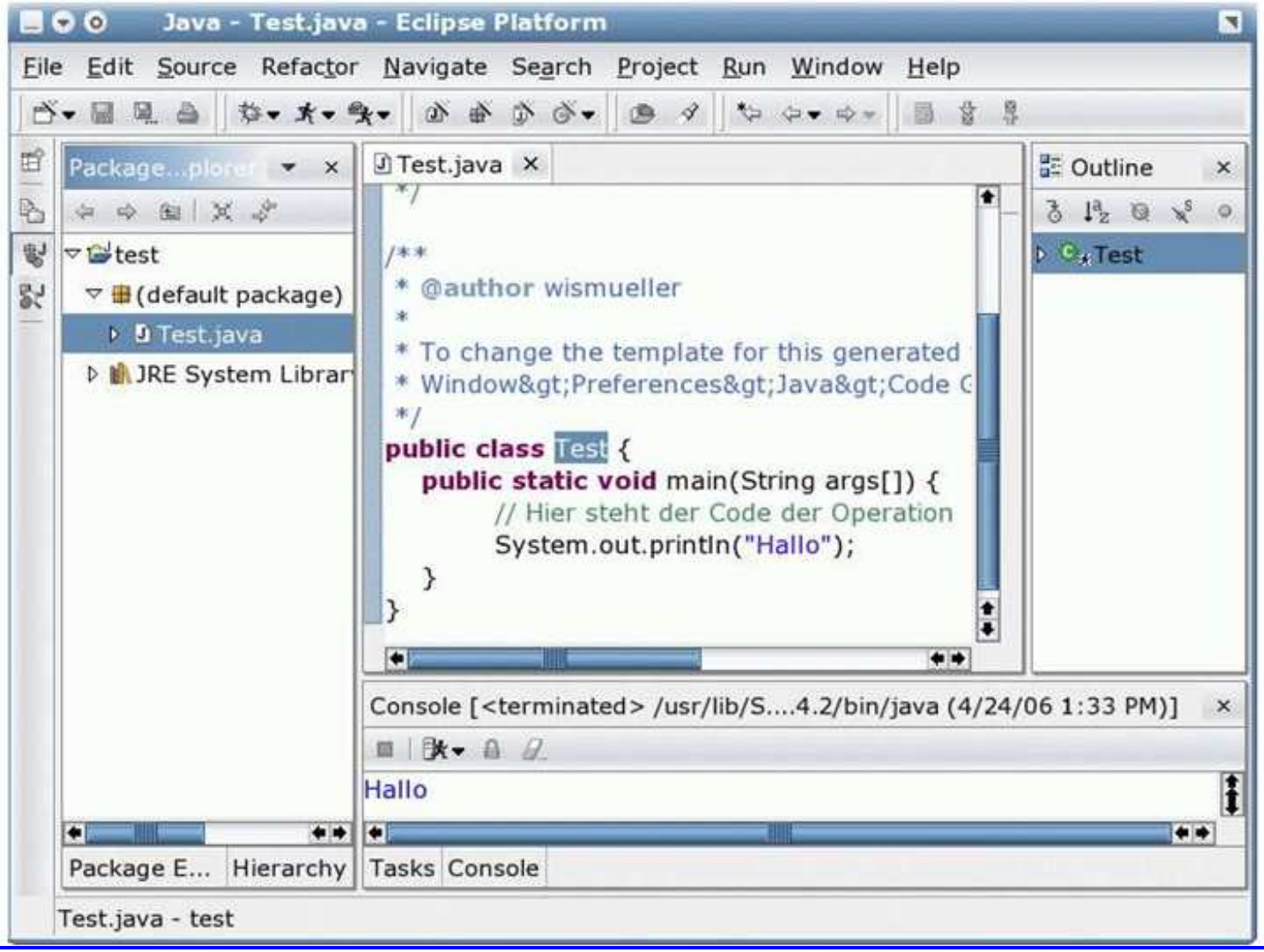
4.1 Erstellen und Ausführen von Java-Programmen

Entwicklungsumgebungen für Java-Programme

- Entwicklungsumgebungen integrieren Editor, Java-Compiler und weitere Werkzeuge, u.a.
  - UML-Diagramme inkl. Erzeugung von Code-Rahmen
  - Erzeugung von Programmdokumentation
  - Debugger zur Fehlersuche (siehe später)
- Vorteil: durchgängige, einheitliche Software-Umgebung
- Beispiele (siehe auch WWW-Seite):
  - **BlueJ**: speziell für die Ausbildung
    - sehr einfache UML-Unterstützung, Objekterzeugung
  - **Eclipse**: professionelle Umgebung
    - UML-Unterstützung durch Plugin möglich

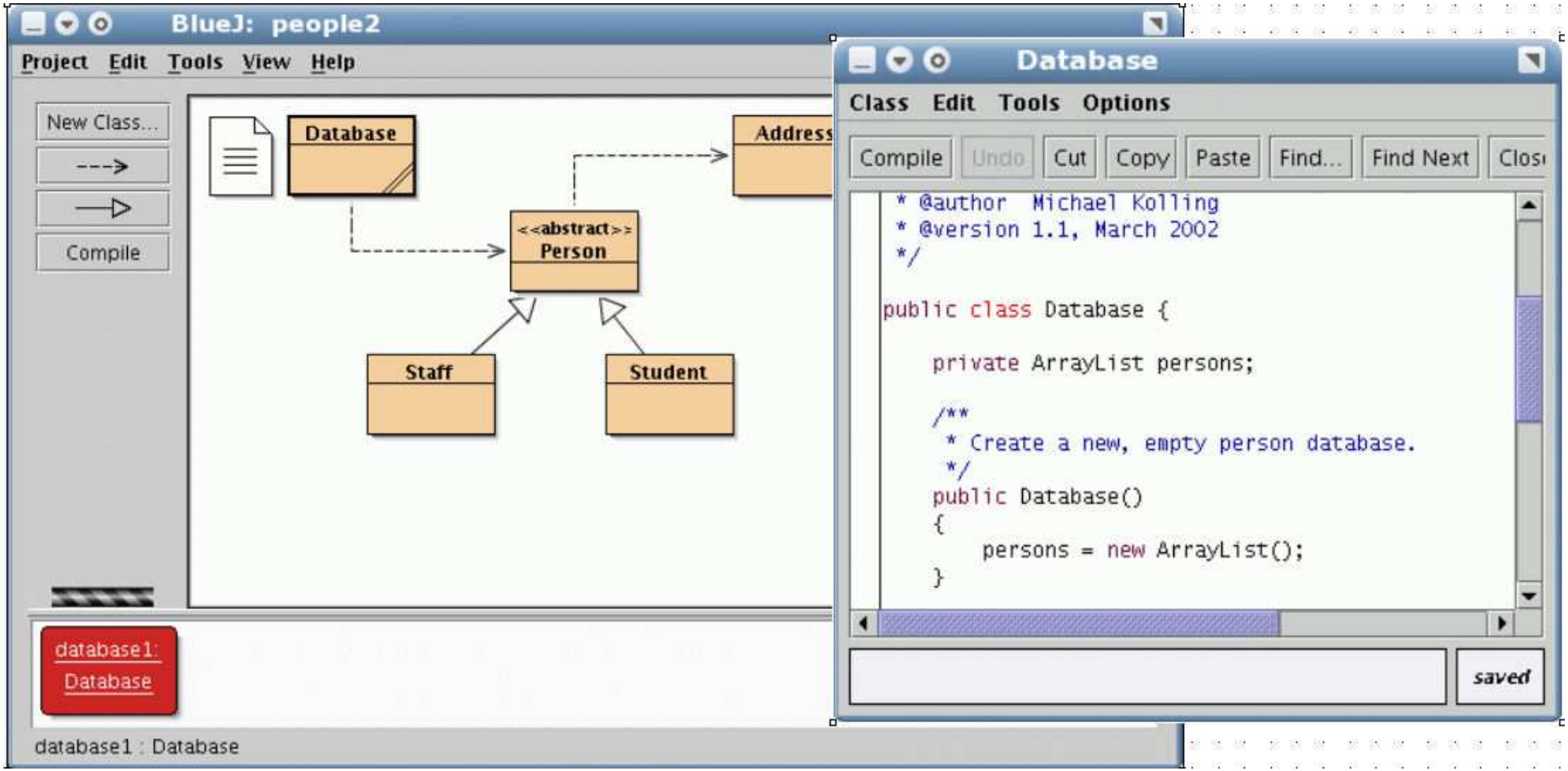
4.1 Erstellen und Ausführen von Java-Programmen

Eclipse



4.1 Erstellen und Ausführen von Java-Programmen

BlueJ



Beginn des Abschnitts für Erstsemester bzw. zur Wiederholung

Da die Syntax und Semantik der nicht-objektorientierten Konstrukte von Java weitestgehend mit C/C++ identisch ist, ist dieser Abschnitt für Studenten mit C/C++ Wissen nicht obligatorisch.

Ab Folie **333** geht es weiter mit einer Zusammenfassung der wichtigsten Unterschiede zwische Java und C/C++.

4.2 Beschreibung von Programmiersprachen

- Syntax
  - Form, Schreibweise, Grammatik der Sprachkonstrukte
  - Immer formal beschrieben
- Semantik
  - Bedeutung der Sprachkonstrukte
  - Fast immer informell beschrieben
- Pragmatik
  - Verwendung der Sprachkonstrukte
  - Meist überhaupt nicht beschrieben
- Im folgenden: Syntax und Semantik
  - Pragmatik: in den Übungen ...

4.2 Beschreibung von Programmiersprachen

Beschreibung von Syntax: Backus-Naur-Form (BNF)

- Einführung von Namen (Nichtterminale) für Programmteile
- Beschreibung des Aufbaus dieser Programmteile durch Regeln:

	Schreibweise:	Beispiele für <A>:
Alternativen	<A> ::= x | y | <B>; <B> ::= A | B	x A; B;
Optionale Teile	<A> ::= foo([bar])	foo() foo(bar)
Wiederholte Teile	<A> ::= x(P {,P})	x(P) x(P,P,P)

- Anmerkung: Farbe nur in der Vorlesung zur Verdeutlichung
  <Nichtterminal>, Terminal, Metasymbol

Beschreibung von Syntax

Aufgabe EBNF:

Setzen Sie die BNF für "**zahl**" voraus. Entwickeln Sie nun eine BNF für "infix_ausdruck", die alle zulässigen arithmetischen Ausdrücke ohne Klammern für beliebige Kombinationen der vier Grundrechenarten und Zahlen beschreibt, also beispielsweise 5+12-6-1-33*16/8. Eine einzelne Zahl ist auch ein zulässiger Ausdruck.

ziffer = 0 | ... | 9
zahl = < ziffer >
operator = + | - | * | /
infix_ausdruck = zahl { operator infix_ausdruck }

Beschreibung von Syntax: Backus-Naur-Form (BNF)

Aufgabe EBNF:

Zählen Sie alle Worte der folgenden Sprache auf. Um ein Wort der Sprache zu bilden, wird mit dem Nichtterminalsymbol **A** begonnen.

A ::= “a“ [B] | C
B ::= “b“ “c“ (“b“ | C)
C ::= “d“ ([“e“] | “f“)

1. **a**
2. **abcb**
3. **abcd**
4. **abcde**
5. **abcdf**
6. **d**
7. **de**
8. **df**

4.3 Syntaktische Grundelemente von Java

Reservierte Schlüsselworte:

abstract	const	float	int	protected	throw
boolean	continue	for	interface	public	throws
break	default	future	long	rest	transient
byte	do	generic	native	return	true
byvalue	double	goto	new	short	try
case	else	if	null	static	var
cast	extends	implements	operator	super	void
catch	false	import	outer	switch	volatile
char	final	inner	package	synchronized	while
class	finally	instanceof	private	this	

Ablaufkontrolle	Konstante	andere (Deklarationen)
Datentypen	Objekt−Orientierung	reserviert für Erweiterungen

4.3 Syntaktische Grundelemente von Java ...

Namen (Identifikatoren):

- Für Klassen, Attribute, Operationen, Parameter, Variablen,...
- Können beliebige Länge haben
- Dürfen nur aus Buchstaben, Ziffern, ’ ’ und ’$’ bestehen
- Müssen mit einem Buchstaben, ’ ’ oder ’$’ beginnen
- Dürfen kein Schlüsselwort sein
- Beispiele:

korrekt:	falsch:	Grund:
`Summe`	`get Name`	Leerzeichen verboten
`getName`	`Tuer-1`	’-’ verboten
`$all4you`	`2hoch4`	Ziffer am Anfang
`_1_2`	`while`	reserviertes Wort

4.3 Syntaktische Grundelemente von Java ...

Konstanten:

- `2002` `-2L` `0xFABEL` `2.1` `0.1E-23` `.1e+19` `'a'` `'\n'`
  `true` `false` `null` `"Hallo"` ...

Operatoren:

- `+` `-` `*` `/` `&` `&&` `=` `==` `>=` `*=` `>` `>>` `>>>` ...

Klammern:

- `(` `)` `[` `]` `{` `}`

Trennzeichen:

- `,` `;` `.` sowie Leerräume, Tabstops und Zeilenwechsel

4.3 Syntaktische Grundelemente von Java ...

Kommentare

- ```
  // bis zum Ende der Zeile
  ```
- ```
  /* über mehrere Zeilen
     hinweg bis zu */
  ```
- ```
  /** Dokumentation
      (automatisch extrahiert durch javadoc)
  */
  ```
- Falsch: `/* Kommentar darf */ nicht enthalten */`
  ```

4.4 Anweisungen

Vorbemerkung: Variablen

- Eine Variable bezeichnet einen Speicherbereich, der Daten eines gegebenen Typs speichern kann
- In Java gibt es drei Arten von Variablen:
  - Datenfelder (in Objekten)
    - speichern die Attributwerte von Objekten
  - formale Parameter (in Methoden)
    - speichern die Werte der beim Aufruf der Methode übergebenen Werte (aktuelle Parameter)
    - existieren nur, während die Methode ausgeführt wird
  - lokale Variable (in Methoden, siehe 4.5.3)
    - existieren nur, während die Methode ausgeführt wird
    - dienen der temporären Speicherung von Daten

4.4 Anweisungen

Vorbemerkung: Variablen …

- Eigenschaften von Variablen (in Java):
  - jede Variable muß vor ihrer Benutzung deklariert werden (siehe 4.5.3)
  - der Wert einer Variablen kann durch eine Zuweisung (jederzeit) geändert werden
  - eine Variable kann überall stellvertretend für ihren Wert verwendet werden
  - eine Variable hat einen genau definierten Gültigkeitsbereich, in dem sie verwendet werden darf (siehe 4.5.3)
  - jede Variable hat eine Lebensdauer, während der sie existiert (siehe vorige Folie)

4.4 Anweisungen

4.4.1 Ausdrücke und Zuweisungen

- Ein Ausdruck berechnet einen Wert
- Ein Ausdruck kann (u.a.) folgende Formen annehmen:

```
<Ausdruck> ::= <Operand>
             | <UnOp> <Operand> | <Operand> <PostOp>
             | <Operand> <BinOp> <Operand>

<Operand> ::= <Variable> | <Konstante>
            | <Ausdruck> | ( <Ausdruck> )

<UnOp> ::= + | - | ++ | -- | ...      (Unärer Präfix-Operator)

<PostOp> ::= ++ | --                  (Unärer Postfix-Operator)

<BinOp> ::= + | - | * | / | % | ...   (Binärer Operator)
```

- Beispiele: `4 + a * (b - c)` `((((-a) - b) - c) - d)`

4.4.1 Ausdrücke und Zuweisungen ...

Erlaubte Operatoren:

Arithmetische Operatoren:	+	-	*	/	%	++	--
Bitoperatoren:	&	|	^	~	<<	>>	>>>
Vergleichsoperatoren:	>	>=	==	!=	<=	<	? :
Logische Operatoren:	&	&&	|	||	!		
Zuweisungsoperatoren:	=	+=	-=	*=	/=	%=	
	&=	|=	^=	<<=	>>=	>>>=	

Unärer Operator — Unärer und binärer Operator

Binärer Operator — Ternärer Operator

4.4.1 Ausdrücke und Zuweisungen ...

Zuweisungen

- Eine Zuweisung ist eine Anweisung, die einer Variablen einen Wert zuweist, d.h. sie speichert den Wert im Speicherbereich der Variablen ab
- Syntax: `<Variable> = <Ausdruck> ;`
- Was passiert?
  - 1. bestimme Speicherbereich der Variablen
  - 2. berechne Wert des Ausdrucks
  - 3. speichere Wert im Speicherbereich ab
- Kurzform für Zuweisungen:

```
    <Variable> <BinOp>= <Ausdruck> ;
≡   <Variable> = <Variable> <BinOp> ( <Ausdruck> ) ;
```

4.4.1 Ausdrücke und Zuweisungen ...

Beispiele: arithmetische Operatoren

```
int x, y, breite, laenge, produkt, rest, quotient;

breite   = x + y;              // Addition
laenge   = laenge - 1;         // Subtraktion
produkt  = x * y;              // Multiplikation
quotient = x / y;              // Division
rest     = x % y;              // Modulo (Restoperation)
x++;                           // Inkrement: x = x + 1
x--;                           // Dekrement: x = x – 1

laenge   = 4 * (x + y);        // Klammerung

laenge = breite = 1;           // Eine Zuweisung ist wieder
                               // ein Ausdruck!

produkt = (x = 2) * (y = 5);   // Schlechter Stil!
```

4.4.1 Ausdrücke und Zuweisungen ...

Beispiele: Bitoperatoren

```
int x, y, z;

x = y & z;              // UND
x = y | z;              // ODER
x = y ^ z;              // Exklusives ODER
x = ~ y;                // Komplement
x = y << 1;             // Linksschieben (x = y * 2)
x = y >> z;             // Rechtsschieben
                        // (schiebt Vorzeichen nach)
x = y >>> 4;            // Rechtsschieben
                        // (schiebt 0 nach)

x = x | 010;            // Setzt Bit 3 in x (010 == 8)
y = (x & 0xF0) >> 4;    // y = Bits 7...4 von x (0xF0 = 240)
```

4.4.1 Ausdrücke und Zuweisungen ...

Beispiele: Vergleichsoperatoren

```
int x, y, summe, zaehler, minimum;

if (x > y) ...                    // größer
if (x >= y) ...                   // größer oder gleich
if (summe == x + y) ...           // gleich
if (x != y) ...                   // ungleich
if (zaehler < 100) ...            // kleiner
if (x + 1 <= 1 - y) ...           // kleiner oder gleich

minimum = (x < y) ? x : y;        // Operator mit 3 Operanden!
                                  // Ergebnis:
                                  //    Falls (x < y), dann x
                                  //                   sonst y
```

4.4.1 Ausdrücke und Zuweisungen ...

Beispiele: logische Operatoren

```
int x, y, zaehler;
boolean leseFlag, schreibFlag;

if (leseFlag || schreibFlag) ...          // logisches ODER
   // schreibFlag nicht ausgewertet, wenn leseFlag == true

if (leseFlag | schreibFlag) ...           // logisches ODER
   // immer beide Operanden ausgewertet

if ((x != 0) && (y / x < 5)) ...          // logisches UND
   // y / x nicht ausgewertet, wenn x == 0

if (leseFlag & (zaehler++ < 10)) ...      // logisches UND
   // immer beide Operanden ausgewertet

if (!(x < 0)) ...                         // logisches NICHT
   // entspricht if (x >= 0) ...
```

4.4.1 Ausdrücke und Zuweisungen ...

Beispiele: Zuweisungsoperatoren

```
int x, y, z, summe, laenge, produkt, rest, quotient;

summe     = x + y;          // Einfache Zuweisung
x        += -5;             // x = x + (-5);
laenge   -= 1;              // laenge = laenge - 1;
produkt  *= x - y;          // produkt = produkt * (x - y);
quotient /= 2;              // quotient = quotient / 2;
rest     %= 4;              // rest = rest % 4;
x        &= 0xFFFFFFFE;     // x = x & 0xFFFFFFFE;
x        |= 1;              // x = x | 1;
x        ^= 1;              // x = x ^ 1;
x       <<= 1;              // x = x << 1;
x       >>= z + y;          // x = x >> (z + y);
x      >>>= 4;              // x = x >>> 4;
```

4.4.1 Ausdrücke und Zuweisungen ...

Ausdrücke ...

- In welcher Reihenfolge werden die Operatoren bei Ausdrücken der Form

  `<Operand> <BinOp> <Operand> <BinOp> <Operand> ...`

  ausgewertet?
- Reihenfolge richtet sich
  - nach der Priorität der Operatoren
    - Operatoren höherer Priorität werden vor jenen niedrigerer Priorität ausgewertet
  - bei Operatoren gleicher Priorität nach der Assoziativität
    - von links nach rechts / von rechts nach links
- Gute Praxis: Im Zweifelsfall Klammern verwenden!

4.4.1 Ausdrücke und Zuweisungen ...

Prio.	Operatoren	Assoziativität
14	() , [] , postfix ++ , postfix --	von links
13	unäres + , unäres - , präfix ++ , präfix -- , ~ , !	von rechts
12	(<Typ>) , new	von links
11	* , / , %	von links
10	+ , -	von links
9	<< , >> , >>>	von links
8	< , <= , > , >= , instanceof	von links
7	== , !=	von links
6	&	von links
5	^	von links
4	|	von links
3	&&	von links
2	||	von links
1	?:	von links
0	=, +=, -=, *=, /=, %=, <<=, >>=, >>>=, &=, |=, ^=	von rechts

4.4.1 Ausdrücke und Zuweisungen ...

Beispiele

- Priorität und Assoziativität

```
a << b * c + d    ≡    a << ((b * c) + d)
 a / b * c % d    ≡    (((a / b) * c) % d)
          -~a     ≡    -(~a)
a += b = c * d    ≡    a += (b = c * d)
        a+++b     ≡    a++ + b
```

- Präfix und Postfix Inkrement /Dekrement

```
a = 10;                  |  a = 10;
b = ++a;                 |  b = a++;
// a = 11, b = 11        |  // a = 11, b = 10
```

4.4.2 Anweisungsfolge / Block

- In einer Anweisungsfolge werden mehrere Anweisungen hintereinander ausgeführt
  - jede Anweisung wird mit einem Strichpunkt (`;`) beendet
- Anweisungsfolgen treten z.B. im Rumpf von Methoden auf
- Eine Anweisungsfolge kann mit `{ }` zu einem Block geklammert werden, z.B.:

```
{
        länge = 10; breite = 15;
        fläche = länge * breite;
}
```

4.4.2 Anweisungsfolge / Block

- Ein Block ist überall da erlaubt, wo eine Anweisung stehen kann
  - der abschließende Strichpunkt entfällt dann

4.4 Anweisungen ...

4.4.3 Auswahl-Anweisungen

- Eine Auswahlanweisung dient dazu, die Ausführung von Anweisungen von einer Bedingung abhängig zu machen
- Bedingte Anweisung:

```
if ( <Bedingung> )          // Falls Bedingung erfüllt ist,
    <Anweisung> ;           //    führe Anweisung aus
```

- Alternativauswahl:

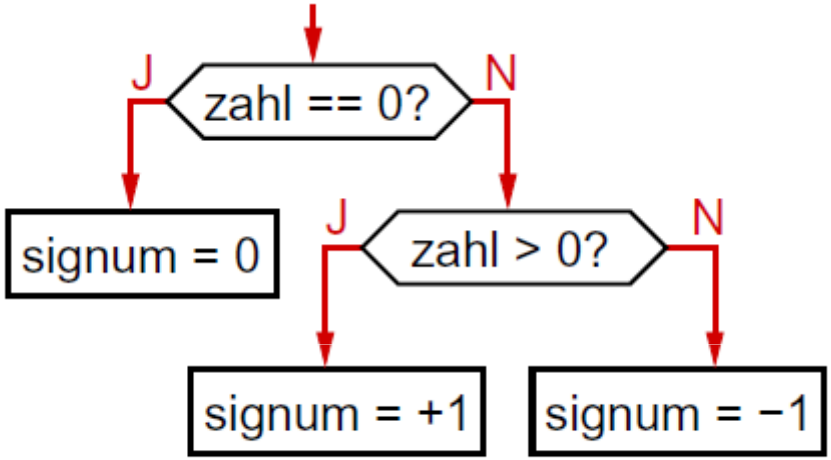
```
if ( <Bedingung> )          // Falls Bedingung erfüllt ist,
    <Anweisung_1> ;         //    führe Anweisung_1 aus,
else                        // sonst
    <Anweisung_2> ;         //    führe Anweisung_2 aus.
```

4.4.3 Auswahl-Anweisungen

Beispiel: Berechne das Signum einer Zahl

- Mit if-Anweisung:

```
if (zahl == 0)
        signum = 0;
else
if (zahl > 0)
        signum = +1;
else
signum = -1;
```



- Mit Auswahloperator:

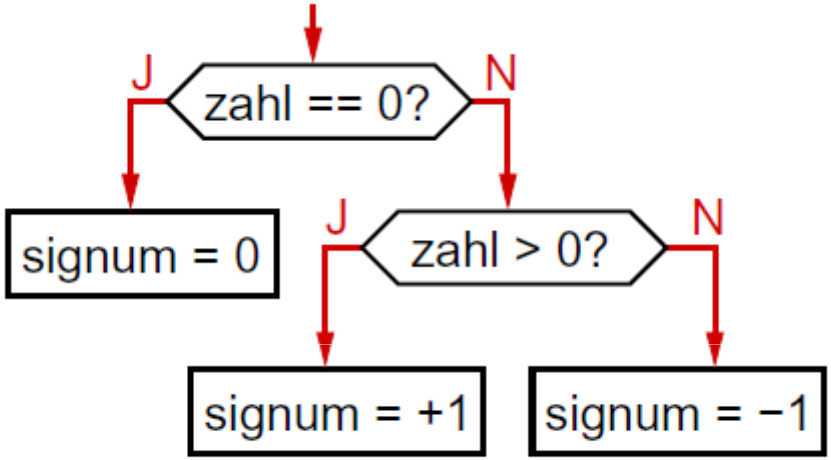
```
signum =  zahl == 0  ?  0  :  ( zahl > 0  ?  +1  :  -1 ) ;
```

4.4.3 Auswahl-Anweisungen

Beispiel: Berechne das Signum einer Zahl

- Mit if-Anweisung:

```
if (zahl == 0)
        signum = 0;
else
if (zahl > 0)
        signum = +1;
else
signum = -1;
```



- Mit Auswahloperator:

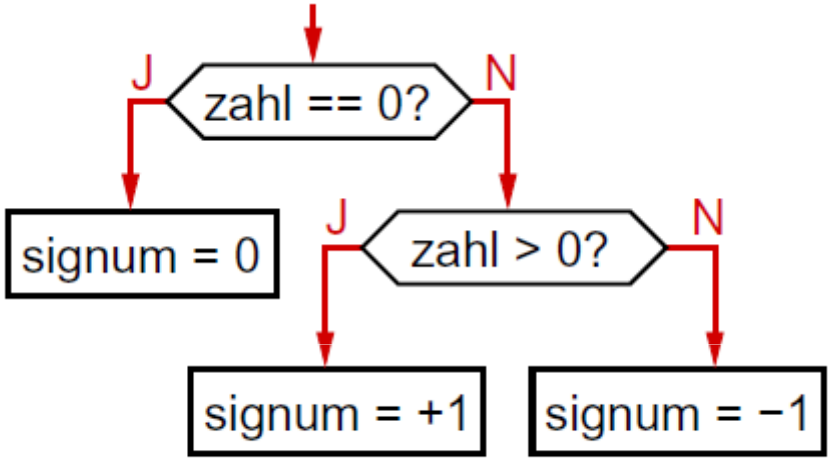
```
signum =  zahl == 0 ? 0 : ( zahl > 0 ? +1 : -1 ) ;
```

4.4.3 Auswahl-Anweisungen

Beispiel: Berechne das Signum einer Zahl

- Mit if-Anweisung:

```
if (zahl == 0)
        signum = 0;
else
if (zahl > 0)
        signum = +1;
else
signum = -1;
```



- Mit Auswahloperator:

```
signum = zahl == 0 ? 0 : ( zahl > 0 ? +1 : -1 ) ;
```

4.4.3 Auswahl-Anweisungen

Verschachtelte if-Anweisungen

```
if ( <Bedingung1> )
    if ( <Bedingung2> )
        <Anweisung1> ;
else
    <Anweisung2> ;
```

≠

```
if ( <Bedingung1> ) {
    if ( <Bedingung2> )
        <Anweisung1> ;
}
else
    <Anweisung2> ;
```

```
if ( <Bedingung1> )
    if ( <Bedingung2> )
        <Anweisung1> ;
else
    <Anweisung2> ;
```

=

```
if ( <Bedingung1> ) {
    if ( <Bedingung2> )
        <Anweisung1> ;
    else
        <Anweisung2> ;
}
```

4.4.3 Auswahl-Anweisungen

Viele Alternativen

- Beispiel: Menuauswahl

```
if (menuItem == 1) {
        ...
}
else if (menuItem == 2) {
        ...
}
else if ((menuItem == 3) || (menuItem == 4)) {
        ...
}
else {
        ...
}
```

- Besser: switch-Anweisung

4.4.3 Auswahl-Anweisungen

Die `switch`-Anweisung

- Die switch-Anweisung wählt abhängig vom Wert eines Ausdrucks eine von mehreren Alternativen aus

```
switch ( <Ausdruck> )    // Ausdruck muß ganzzahlig sein
{
  case <Wert_1> :         // Falls Ausdruck == Wert_1 ist:
    <Anweisungen_1> ;     //   Anweisungen_1 ausführen.
    break;                //   verlasse den switch-Block.
  case <Wert_2> :         // Falls Ausdruck == Wert_2 ist:
    <Anweisungen_2> ;     //   Anweisungen_2 ausführen.
    break;                //   verlasse den switch-Block.
  ...
  default:                // Falls keiner dieser Fälle:
    <Anweisungen_3> ;     //   Anweisungen_3 ausführen.
    break;                //   verlasse den switch-Block.
}
```

4.4.3 Auswahl-Anweisungen

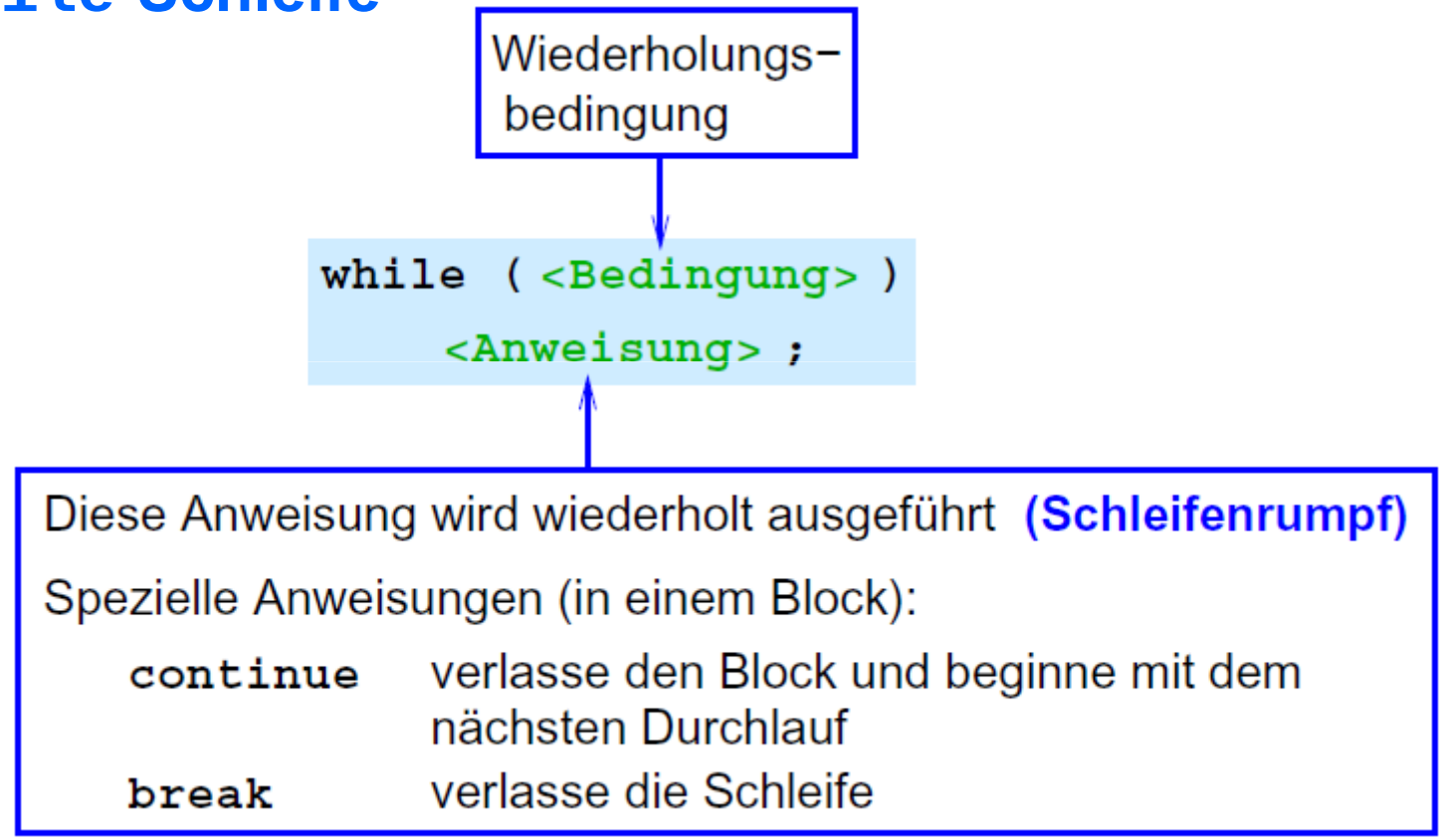
Beispiel: Menuauswahl

```
switch (menuItem) {
    case 1:          // menuItem == 1
        ...
        break;
    case 2:          // menuItem == 2
        ...
        break;
    case 3:          // (menuItem == 3)
    case 4:          // || (menuItem == 4)
        ...
        break;
    default:         // (menuItem < 1) || (menuItem > 4)
        ...
        break;
}
```

- Ohne `break` geht die Ausführung einfach weiter!

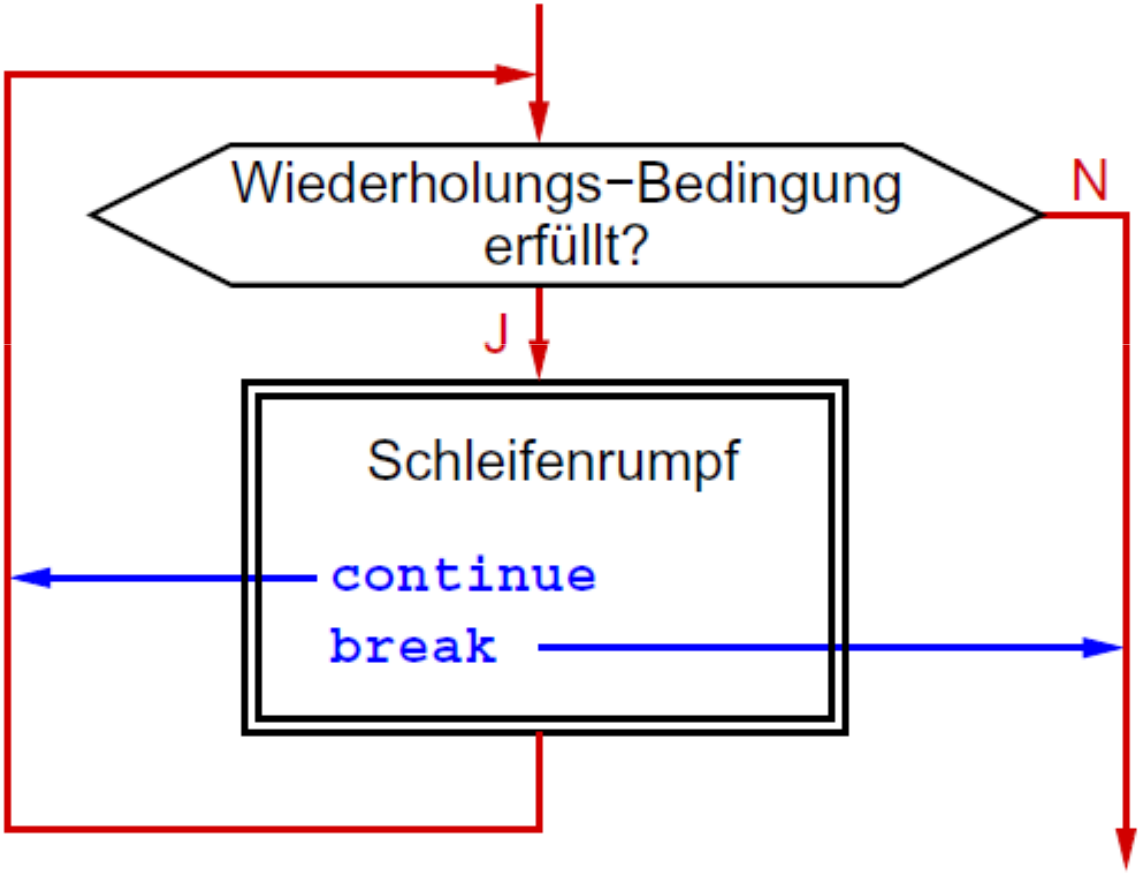
4.4.4 Wiederholungs-Anweisungen

Die `while`-Schleife



4.4.4 Wiederholungs-Anweisungen

Ablauf bei der Ausführung einer `while` -Schleife



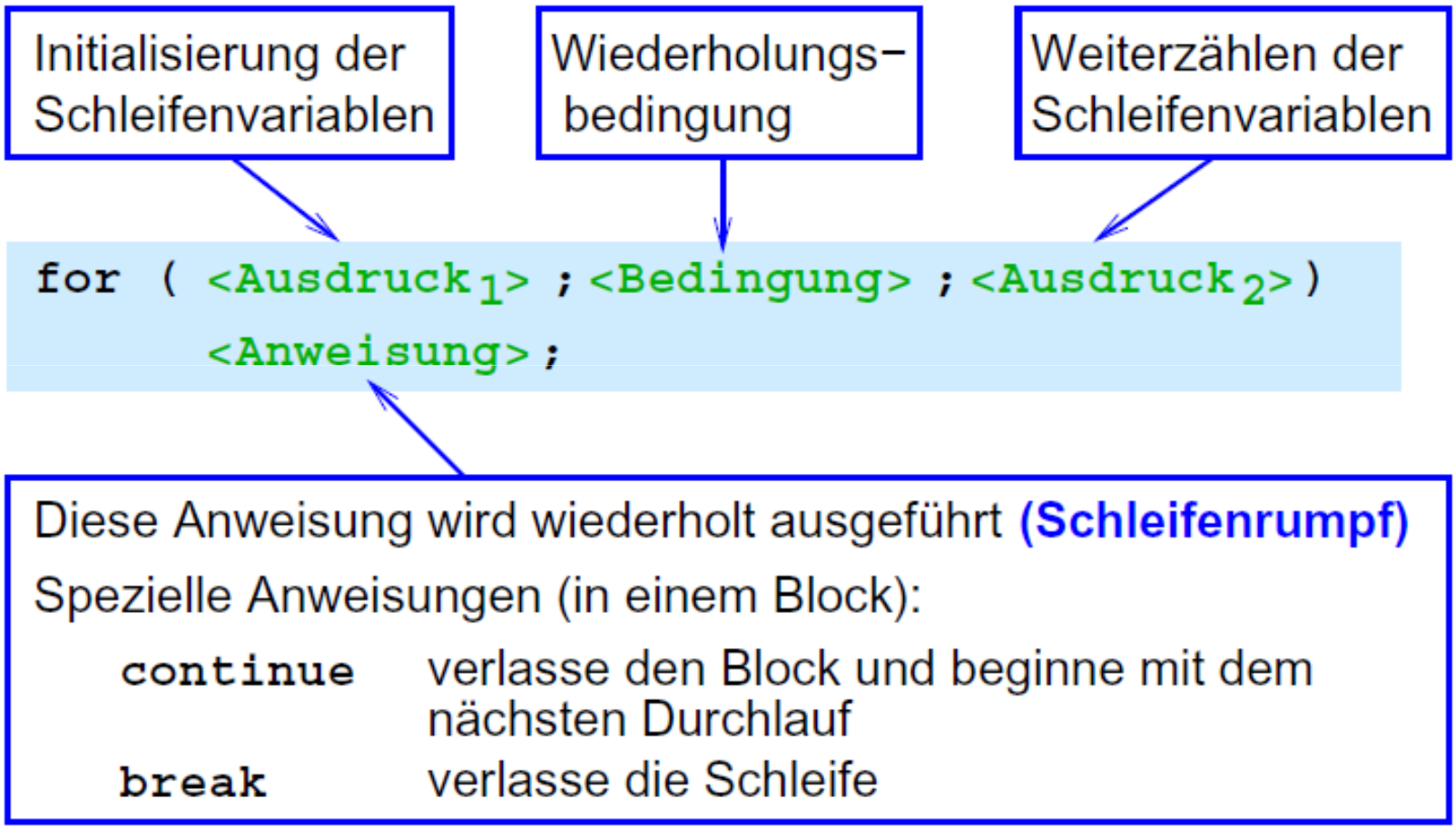
Beispiel: Aufsummieren der Zahlen 1 ... 10

```
int i, summe;

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

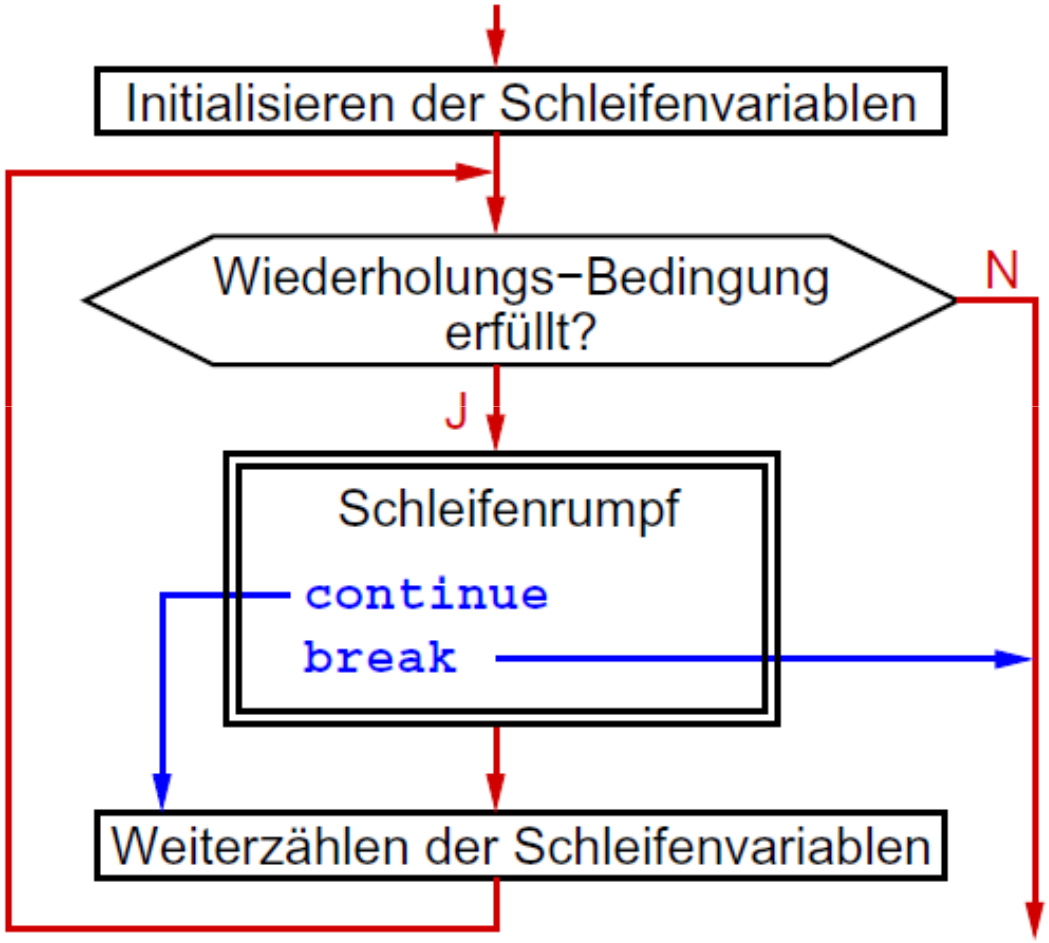
4.4.4 Wiederholungs-Anweisungen

Die `for`-Schleife



4.4.4 Wiederholungs-Anweisungen

Ablauf bei der Ausführung einer `for`-Schleife



4.4.4 Wiederholungs-Anweisungen

Beispiel: Aufsummieren der Zahlen 1 ... 10

- Übliche Realisierung:

```
int i, summe;
summe = 0;
for (i = 1; i <= 10; i++)
          summe += i;
```

- Alternative (schlechter Stil, nur zur Demonstration):

```
int i, summe;
for (i = 1, summe = 0; i <= 10; summe += i, i++) ;
```

  - mehrere, mit Komma getrennte Ausdrücke sind möglich
  - ein leerer Rumpf ist möglich

4.4.4 Wiederholungs-Anweisungen

Die `do while`-Schleife

Diese Anweisung wird wiederholt (aber **mindestens einmal**) ausgeführt **(Schleifenrumpf)**

Spezielle Anweisungen (in einem Block):

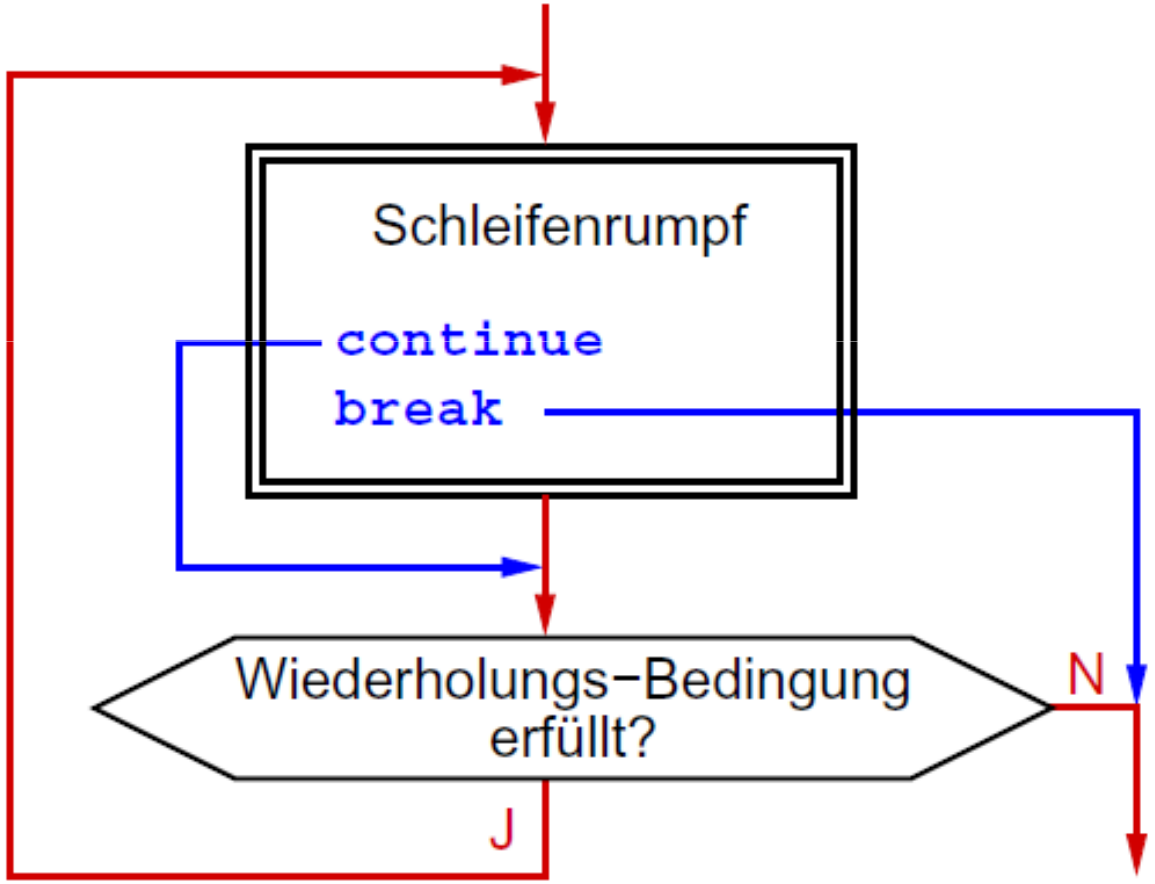
`continue`	verlasse den Block und beginne mit dem nächsten Durchlauf
`break`	verlasse die Schleife

```
do
    <Anweisung>;
while (<Bedingung> );
```

Wiederholungs-bedingung

4.4.4 Wiederholungs-Anweisungen

Ablauf bei der Ausführung einer `do while`-Schleife



Beispiel: Aufsummieren der Zahlen 1 ... 10

```
int i, summe;

summe = 0;
i = 1;
do {
    summe += i;
    i++;
} while(i <= 10);
```

4.4.4 Wiederholungs-Anweisungen

Pragmatik: wann welche Schleife?

- `for`-Schleife:
  - für Zählschleifen (z.B. ” für alle *`i = 1 ...N`* “)
  - wenn natürlicherweise eine Schleifenvariable vorhanden ist / benötigt wird
- `while`-Schleife:
  - wenn die (maximale) Zahl der Wiederholungen nicht im Voraus bekannt ist
  - bei komplexen Wiederholungsbedingungen
- `do while`-Schleife:
  - wenn der Schleifenrumpf mindestens einmal ausgeführt werden soll

4.4.4 Wiederholungs-Anweisungen

Programmiertechnische Hinweise

- Beim Entwurf einer Schleife ist zu beachten:
  - Initialisierung
  - Abbruchbedingung
  - Terminierung: der Schleifenrumpf muß irgendwann die Abbruchbedingung herstellen
  - Unnütze Wiederholung von Berechnungen vermeiden
- Häufige Fehler:
  - Falsche / fehlende Initialisierung
  - Falsche Abbruchbedingung (Schleife terminiert nicht)
  - ”Off by one“
  - `while` statt `do while` und umgekehrt

4.4 Anweisungen …

4.4.5 Beispiel: Primzahlen

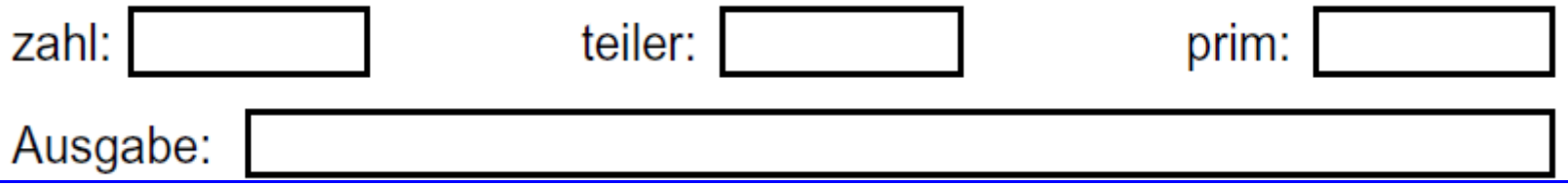
```
// Datum 12.05.10
// Beschreibung: Drucke die Liste der Primzahlen, die
// kleiner als 100 sind. (Naiver Algorithmus;
// es gibt wesentlich bessere!)
public class PrimZahlen
{
    public static void main(String[] args)
   {
        int zahl, teiler;
        boolean prim;
        System.out.println(2);            // Ausgabe einer 2
```

4.4.5 Beispiel: Primzahlen ...

```
        for (zahl = 3; zahl < 100; zahl += 2) {
                prim = true;
                for (teiler = 3; teiler < zahl; teiler += 2) {
                        if (zahl % teiler == 0) {
                                prim = false;
                                break;
                        }
                }
                if (!prim)
                        continue;              Schlechter Stil !!
                System.out.println(zahl);  // Ausgabe von 'zahl'
        }
    }
}
```

4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```



4.4.5 Beispiel: Primzahlen ...

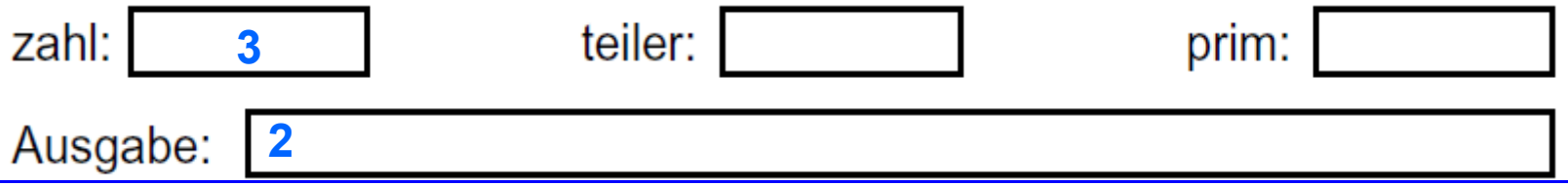
```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl: teiler: prim:

Ausgabe: 2

4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```



4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl: 3 teiler: prim:

Ausgabe: 2

4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl: 3 teiler: prim: true

Ausgabe: 2

4.4.5 Beispiel: Primzahlen ...

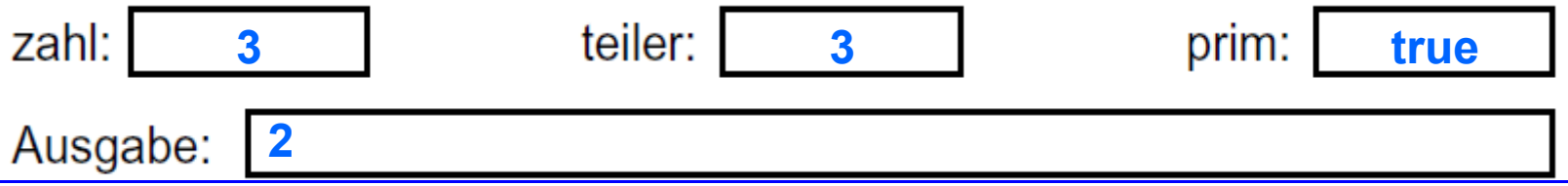
```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl: 3 teiler: 3 prim: true

Ausgabe: 2

4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```



4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl: 3 teiler: 3 prim: true

Ausgabe: 2

4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl: 3 teiler: 3 prim: true

Ausgabe: 2 3

4.4.5 Beispiel: Primzahlen ...

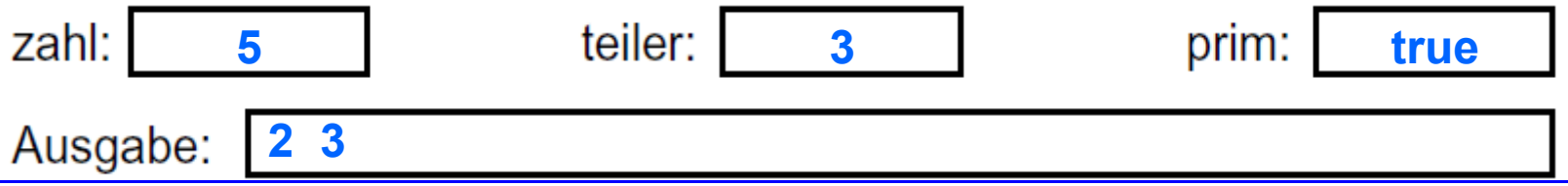
```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl: 5 teiler: 3 prim: true

Ausgabe: 2 3

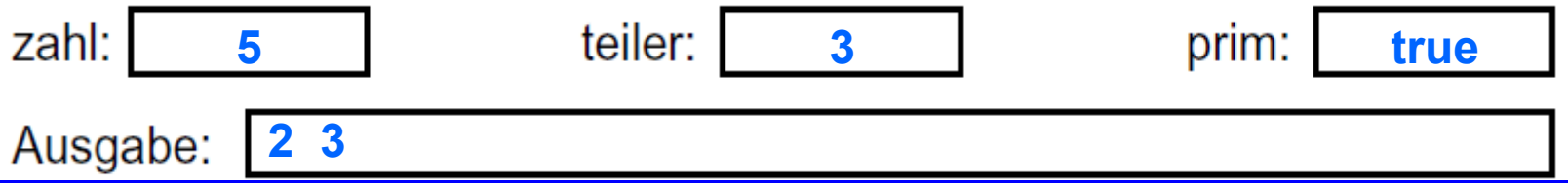
4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```



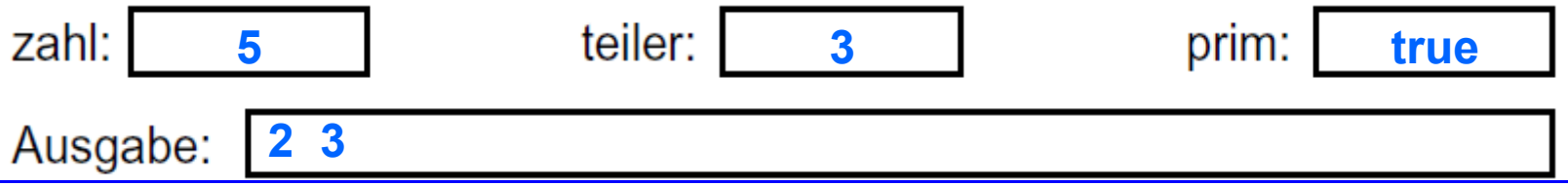
4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```



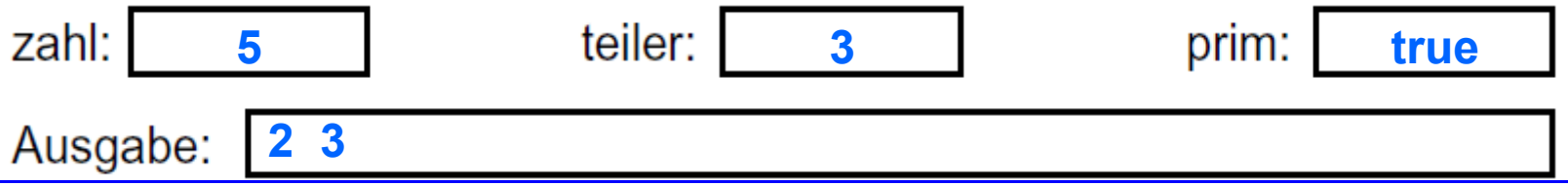
4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```



4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```



4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
        if (zahl % teiler == 0) {
            prim = false;
            break;
        }
   }
   if (!prim)
        continue;
   System.out.println(zahl);
}
```

zahl: 5 teiler: 3 prim: true

Ausgabe: 2 3

4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl:	5	teiler:	5	prim:	true

Ausgabe: 2 3

4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl:	5	teiler:	5	prim:	true

Ausgabe: 2 3

4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl: 5 teiler: 5 prim: true

Ausgabe: 2 3

4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl: 5 teiler: 5 prim: true

Ausgabe: 2 3 5

4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl: 7 teiler: 5 prim: true

Ausgabe: 2 3 5

4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl: 7 teiler: 5 prim: true

Ausgabe: 2 3 5

4.4.5 Beispiel: Primzahlen ...

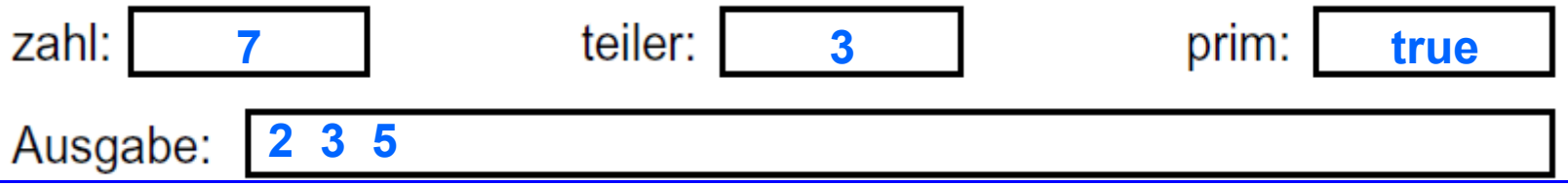
```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl: 7 teiler: 5 prim: true

Ausgabe: 2 3 5

4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```



4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl: 7 teiler: 3 prim: true

Ausgabe: 2 3 5

4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl: 7 teiler: 3 prim: true

Ausgabe: 2 3 5

4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl: 7 teiler: 5 prim: true

Ausgabe: 2 3 5

4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl: 7 teiler: 5 prim: true

Ausgabe: 2 3 5

4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl: 7 teiler: 5 prim: true

Ausgabe: 2 3 5

4.4.5 Beispiel: Primzahlen ...

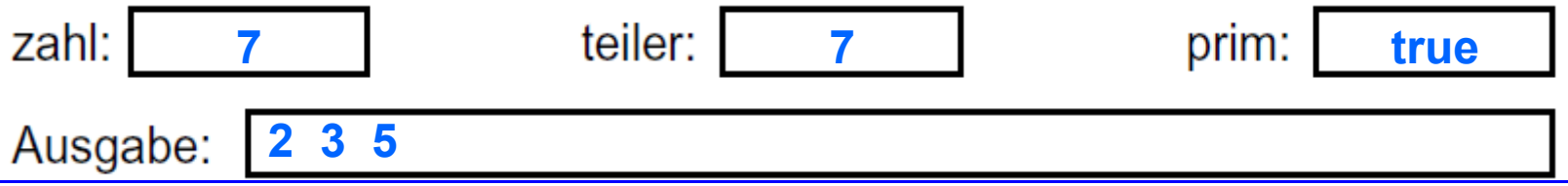
```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl: 7 teiler: 7 prim: true

Ausgabe: 2 3 5

4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```



4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl: 7 teiler: 7 prim: true

Ausgabe: 2 3 5

4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl: 7 teiler: 7 prim: true

Ausgabe: 2 3 5 7

4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl: 9 teiler: 7 prim: true

Ausgabe: 2 3 5 7

4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl: 9 teiler: 7 prim: true

Ausgabe: 2 3 5 7

4.4.5 Beispiel: Primzahlen ...

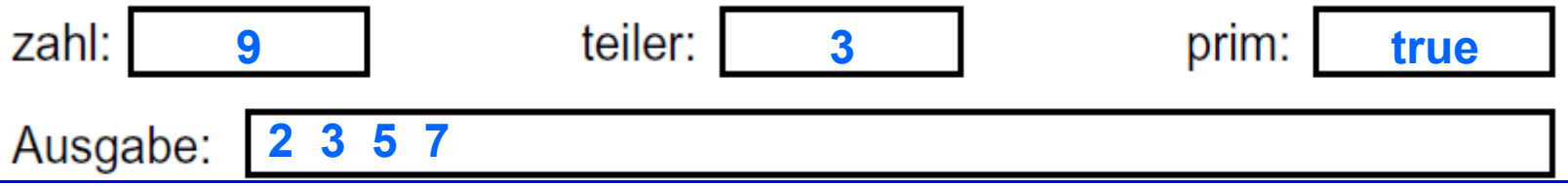
```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl: 9 teiler: 7 prim: true

Ausgabe: 2 3 5 7

4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```



4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl: 9 teiler: 3 prim: true

Ausgabe: 2 3 5 7

4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl: 9 teiler: 3 prim: true

Ausgabe: 2 3 5 7

4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl: 9 teiler: 3 prim: false

Ausgabe: 2 3 5 7

4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl: 9 teiler: 3 prim: false

Ausgabe: 2 3 5 7

4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl: 9 teiler: 3 prim: false

Ausgabe: 2 3 5 7

4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl: 9 teiler: 3 prim: false

Ausgabe: 2 3 5 7

4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl: 11 teiler: 3 prim: false

Ausgabe: 2 3 5 7

4.4.5 Beispiel: Primzahlen ...

```
System.out.println(2);
for (zahl = 3; zahl < 100; zahl += 2) {
   prim = true;
   for (teiler = 3; teiler < zahl; teiler += 2) {
       if (zahl % teiler == 0) {
           prim = false;
           break;
       }
   }
   if (!prim)
       continue;
   System.out.println(zahl);
}
```

zahl: 11 teiler: 3 prim: false

Ausgabe: 2 3 5 7

4.4 Anweisungen …

4.4.6 Programmier-Konventionen

- Verwenden Sie aussagekräftige Namen
- Verdeutlichen Sie die Programmstruktur durch Einrückungen
- Bei Unklarheit: auch einzelne Anweisungen mit `{ }` klammern
- Nach `if, switch, for, while` Leerraum lassen
- Verwenden Sie Leerzeilen und Kommentare um Programmabschnitte zu trennen

4.4 Anweisungen …

4.4.6 Programmier-Konventionen

- Verwenden Sie aussagekräftige Namen
- Verdeutlichen Sie die Programmstruktur durch Einrückungen
- Bei Unklarheit: auch einzelne Anweisungen mit `{ }` klammern
- Nach `if, switch, for, while` Leerraum lassen
- Verwenden Sie Leerzeilen und Kommentare um Programmabschnitte zu trennen

SO NICHT:

```
public class PrimZahlen{public static void main(String[]
   a){int z,t;for(System.out.println(2),z=3;z<100;z+=2){for
   (t=3;t<z&&z%t!=0;t+=2);if(t==z)System.out.println(z);}}}
```

Einführung in die Informatik II

SS 2012

4 Java Grundlagen – 14.05.2012

Zu meiner Person - Andreas Hoffmann

- Studium der Technischen Informatik an der Universität Siegen
- Promotion an der Universität Siegen 2010
- Seit 2005 wiss. Mitarbeiter für Betriebssysteme und verteilte Systeme an der Universität Siegen
- Forschung: Verteilte Systeme, Sicherheit und Datenschutz von verteilten Systemen, eAssessment – / eLearning Systeme, Entwicklung von mobile Anwendungen (Apps) speziell im Bildungsbereich
- E-mail: andreas.hoffmann@uni-siegen.de
- Web: http://www.uni-siegen.de/fb12/ws/mitarbeiter/
- Tel.: 0271/740-4047 Büro: H-B- 8405
- Sprechstunde: Mittwochs, 13:30 - 14:30 Uhr

4.5 Datentypen und Variablen

4.5.1 Datentypen

- Ein Typ definiert
  - eine Menge von **Werten**
  - die darauf anwendbaren **Operationen**
- Alle Variablen und auch alle Konstanten besitzen in Java zwingend einen Typ
  - bei Konstanten durch die Form (Syntax) festgelegt
  - bei Variablen durch deren Deklaration festgelegt
- Datentypen in Java:
  - `byte, short, int, long, float, double, char, boolean`
  - Arrays (Felder), Strings
  - plus: Klassen als benutzerdefinierte Typen

4.5.1 Datentypen ...

Ganze Zahlen (integer)

- Vier Typen mit unterschiedlichen Wertebereichen:

Typ	Bytes	Bits	Wertebereich
byte	1	8	-128 ... 127
short	2	16	-32 768 ... 32 767
int	4	32	-2 147 483 648 ... 2 147 483 647
long	8	64	$\sim -9 \cdot 10^{18}$ ... $9 \cdot 10^{18}$

- Operationen:
  - Arithmetische-, Vergleichs-, Bit-, Zuweisungs-Operationen
  - Typkonversion
- Fehlerbehandlung:
  - Division (/ und %) durch 0: Ausnahme (siehe später)
  - Bei Überlauf: keine Behandlung (falsches Ergebnis) !

4.5.1 Datentypen ...

Gleitkomma-Zahlen (floating point)

- Zwei Typen mit unterschiedlichen Wertebereichen:

Typ	Bytes	Bits	Wertebereich	Genauigkeit
`float`	4	32	-3.4e38 ... 3.4e38	7 Stellen
`double`	8	64	-1.7e308 ... 1.7e308	17 Stellen

- Operationen:
  - arithmetische (incl. %), Vergleichs-, Zuweisungs- Operationen
  - Typkonversion
- Fehlerbehandlung:
  - Bei Überlauf und Division durch 0: `[-] Infinity`
  - Bei `0.0 / 0.0` und ähnlichem: `Nan` (Not a Number)

4.5.1 Datentypen ...

Gleitkomma-Zahlen (floating point) …

- Reelle Zahlen können im Computer nicht exakt dargestellt werden
  - Java: IEEE 754 Standard (8 Byte, 17 Stellen Genauigkeit)
- Dadurch entstehen **Rundungsfehler**
- In der Gleitkomma-Arithmetik gelten Assoziativitäts- und Kommutativitätsgesetz nicht mehr!
- Beispiel:

```
double a, b, x, y;
a = 5.0e-12;        // a =         0.000 000 000 005
b = 1.0e+5;         // b = 100 000.000 000 000 00
x = a + a + b;      // x = 100 000.000 000 000 01
y = b + a + a;      // y = 100 000.000 000 000 00
```

4.5.1 Datentypen ...

Einzelzeichen (char)

Typ	Bytes	Bits	Wertebereich
char	2	16	Unicode (65536 Zeichen) ⊃ ASCII

- Operationen:
  - Vergleichs-, Zuweisungs-Operationen
  - Inkrement, Dekrement und <op>=
  - Typkonversion
- Fehlerbehandlung:
  - Bei Überlauf: Abschneiden der oberen Bits

4.5.1 Datentypen ...

Wahrheitswerte (boolean)

Typ	Wertebereich
`boolean`	`true` (wahr), `false` (falsch)

- Operationen:
  - Vergleich (nur == und `!=`), Zuweisungs-Operationen
  - Boole'sche Operatoren: `&&, ||, &, |, ^, !`
  - **keine** Typkonversion
- Vergleichsoperatoren erzeugen Ergebnis vom Typ boolean
  - Beispiel:

```
boolean ende;
ende = (zahl >= 100) || (eingabe == 0);
```

4.5.2 Konstanten

- Explizite Datenwerte im Programm
- Können bereits vom Compiler interpretiert werden
- Besitzen einen Typ, ersichtlich an ihrer Syntax:

Konstante	Typ
1, -9999, 0xFABE, 0125	int
1L, -9999l, 0xFABEL, 0125L	long
1.0, 2002.5d, 3.14159E8, 05e-9D	double
1.0f, 2002.5f, 3.14159E8f, 05e-9F	float
'A', '\u0041', '\'', '\\', '\n'	char
"String", "\"Hallo\", sagte er\n"	String (Klasse!)
true, false	boolean

4.5.3 Variablen

Deklaration

- Der Typ von Variablen wird durch ihre Deklaration festgelegt.
- Eine Deklaration ist eine Anweisung
- Beispiele:

```
int x, y;
int zähler = 0, produkt = 1;
final double PI = 3.14159265358979323846;
boolean istPrim;
```

- Allgemein:

```
[final] <Typ> <Name> [ = <Ausdruck> ]
                    { , <Name> [ = <Ausdruck> ] } ;
```

- Der Modifier `final` zeigt an, daß die (initialisierte!) Variable nicht veränderbar ist (benannte Konstante)

4.5.3 Variablen ...

Gültigkeitsbereich

- Eine in einer Methode deklarierte Variable (= lokale Variable) ist ab der Deklaration bis zum Ende des umgebenden Blocks gültig (Gültigkeitsbereich)

```
{
    int a = 3, b = 1;
    if (a > 0) {
        c = a + b;  // Fehler: c ungültig!
        int c = 0;
        c += a;     // OK
    }
    b = c;          // Fehler: c ungültig!
}
```

4.5.3 Variablen ...

Gültigkeitsbereich …

- Die Gültigkeitsbereiche von lokalen Variablen mit demselben Namen dürfen nicht überlappen!

```
{
    int a = 3, b = 1;
    if (a > 0) {
        int b; // Fehler: b bereits deklariert!
        int c;      // OK
        ...
    }
    {
        char c;     // OK
        ...
    }
    double a;   // Fehler: a bereits deklariert!
}
```

4.5.4 Typkonversionen

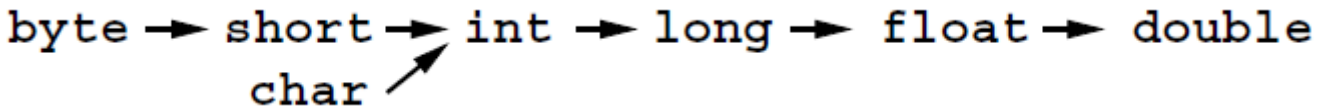
- Der Typ eines Ausdrucks kann (in Grenzen) angepaßt werden:
  - explizite Typkonversion
  - implizite (automatische) Typkonversion
- Explizite Typkonversion:
  - durch Voranstellen von (<Typ>), z.B.:

```
(double) 3 // == 3.0
(char) 65 // == 'a'
(int) x + y // == ((int) x) + y
(int) (x + y)
```

  - Achtung: Dabei kann Information verlorengehen (Programmfehler, z.B. Absturz der Ariane 5!)
  - keine Konversion von / nach `boolean`

4.5.4 Typkonversionen ...

- Implizite Typkonversion:
  - Immer nur von ”kleineren“ zu ”größeren“ Typen:



  - In Ausdrücken:

Mind. ein Op.	andere Operanden	konv. zu
double	double, float, long, int, short, byte, char	double
float	float, long, int, short, byte, char	float
long	long, int, short, byte, char	long
int, short, byte, char	int, short, byte, char	int

4.5.4 Typkonversionen ...

Beispiele

```
double x, y = 1.1;
int i, j = 5;
char c = 'a';
boolean b = true;

x = 3;          // OK: x == 3.0
x = 3/4;        // x == 0.0!
x = c + y * j; // OK, Konversion nach double
i = j + x;     // Fehler: (j + x) hat Typ double
i = (int)x;    // OK, i = ganzzahl. Anteil von x
i = (int)b;    // Fehler: keine Konv. von boolean
b = i;         // Fehler: keine Konv. zu boolean
b = (i != 0); // OK, Vergleich liefert boolean
c = c + 1;     // Fehler: c + 1 hat Typ int
c += 1;        // OK! c == 'b'
```

Ende des Abschnitts

für Erstsemester bzw. zur Wiederholung

Beginn des Abschnitts

Zusammenfassung für C/C++ Programmierer

4.6 Unterschiede zwischen Java und C/C++

- Das Hauptprogramm ist in Java eine statische Klassenmethode

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

- Identifikatoren können in Java auch ’$’ enthalten

- Java hat einige zusätzliche Operatoren:
  - ’>>>’: Rechtsschieben mit Nachschieben einer Null
  - ’&’ bzw. ’|’ für Boole’sche Operatoren
    - logisches Und / Oder, wertet immer beide Operanden aus

- Einige Datentypen sind in Java neu bzw. anders als in C++
  - Datentyp `byte` für 8-Bit Werte
  - Datentyp `char` für Unicode-Zeichen mit 16 Bit
  - Datentyp `boolean` für Boole’sche Werte
  - eingeschränkte Operationen für `char` und `boolean`

4.6 Unterschiede zwischen Java und C/C++

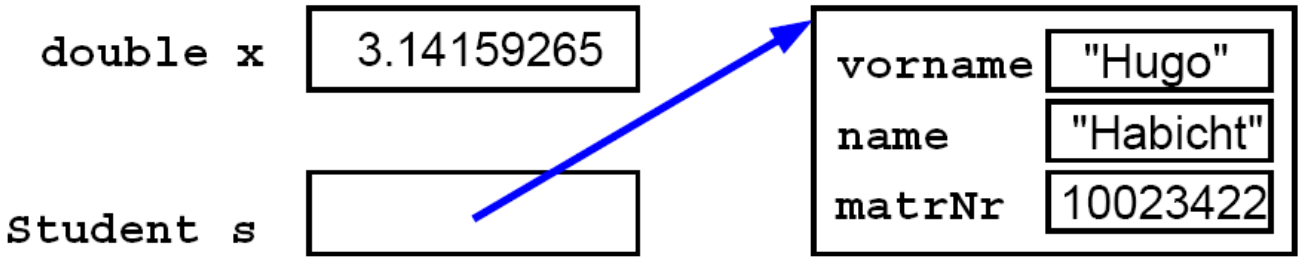
- Deklaration von Konstanten:
  - in Java mit Schlüsselwort final
- Gültigkeitsbereiche von Variablen dürfen in Java nicht überlappen
- Implizite Typkonversion erfolgt in Java nur von ”kleineren“ zu ”größeren“ Typen
- Weitere, teils größere Unterschiede bei Objekterzeugung, Arrays, Strings, Vererbung und Exceptions
  - siehe nachfolgende Abschnitte!

Ende des Abschnitts
Zusammenfassung für C/C++ Programmierer

4.7 Objekte und Methoden

4.7.1 Erzeugung von Objekten

- Erinnerung: Objekte werden in Java nicht direkt in Variablen gespeichert
  - gilt auch für Arrays (und Strings)
- In Variablen werden nur **Referenzen auf solche Speicherbereiche** gespeichert
  - spezieller Wert null verweist nirgendwohin.



- Der Speicherbereich für die Daten wird dynamisch durch den Operator **new** angelegt (**Erzeugung eines Objekts)**

4.7.1 Erzeugung von Objekten

Konstruktoren

- Eine Klasse kann spezielle Operationen definieren, die bei der Erzeugung eines Objekts ausgeführt werden: **Konstruktoren**
- Typische Aufgaben eines Konstruktors:
  - Initialisierung der Attributwerte des neuen Objekts
  - ggf. Erzeugung existenzabhängiger Teil-Objekte
- Ein Konstruktor hat immer denselben Namen wie die Klasse
  - er kann Parameter besitzen, hat aber keinen Ergebnistyp (nicht einmal void)
  - Konstruktoren können auch überladen werden
- Definiert eine Klasse keinen Konstruktor, wird automatisch ein parameterloser Konstruktor erzeugt
  - Attribute werden mit Standardwerten (0 bzw. null) initialisiert

4.7.1 Erzeugung von Objekten

Beispiel

```
class Kugel {
    // Klassenattribute:
    final static double PI = 3.14159265;
    static int anzahl = 0; // zählt erzeugte Kugeln

    // Attribute:
    double xMitte, yMitte, zMitte;
    double radius;

    // Parameterloser Konstruktor (Default−Konstruktor)
    Kugel() {
        radius = 1;                 // setze Radius des neuen Objekts,
                                    // andere Attribute sind mit 0 initialisiert
        anzahl++;                   // Klassenvariable erhöhen
}}
```

4.7.1 Erzeugung von Objekten

Beispiel ...

```
    Kugel(double x, double y, double z) {
        this();                  // ruft Konstruktor Kugel() auf.
        xMitte = x;              // Aufruf von this() muß die
        yMitte = y;              // allererste Anweisung sein!
        zMitte = z;
    }

    Kugel(double x, double y, double z, double r) {
        this(x, y, z);           // ruft Kugel(x,y,z) auf
        radius = r;              // ändert Wert des Radius auf r
    }
}
```

➢ `this( [<Parameterliste>] )` als erste Anweisung führt zum Aufruf eines anderen Konstruktors für dasselbe Objekt

Erzeugung von Objekten

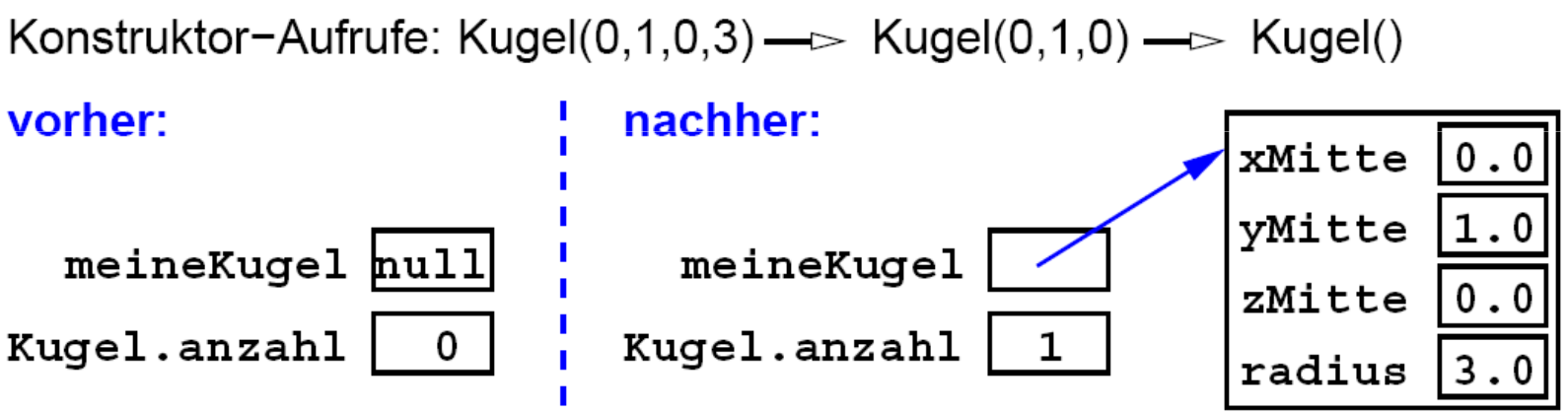
- Der Ausdruck new <Klassenname> ([<Parameterliste>]) erzeugt (instanziiert) ein neues Objekt der angegebenen Klasse
  - Wert des Ausdrucks ist eine Referenz auf das Objekt
- Beispiele: `new Kugel()`

  `new Kugel(1, 0, 0)`
- Ablauf bei der Instanziierung:
  - Anlegen des Objekts im Speicher
    - dabei Belegung der Attribute mit Standardwerten
  - Aufruf des Konstruktors für das neue Objekt
    - passend zu Anzahl / Typen der übergebenen Parameter
  - Rückgabe der Referenz auf das Objekt

4.7.1 Erzeugung von Objekten

Beispiel

```
Kugel meineKugel;
meineKugel = new Kugel(0, 1, 0, 3);
```



4.7.1 Erzeugung von Objekten

```
Kugel() {
        radius = 1;
        anzahl++;
}
...
Kugel(double x, double y, double z) {
        this();
        xMitte = x; yMitte = y; zMitte = z;
}
. . .
Kugel(double x, double y, double z, double r) {
        this(x, y, z);
        radius = r;
}
```

Kugel(0,1,0,3)

Kugel.anzahl=

xMitte=
yMitte=
zMitte=
radius=

4.7.1 Erzeugung von Objekten

Weitere Möglichkeiten zur Initialisierung von Attributen

- Gemeinsam mit der Deklaration

```
static int anzahl = 0;
double radius = 1.0; // Default-Radius: 1.0
```

- In einem eigenen, speziellen Block der Klassendefinition

```
class Kugel {
    ...
    static {                    // wird nur einmal durchlaufen,
        anzahl = 0;             // wenn die Klasse geladen wird
    }
    { // wird für jedes erzeugte Objekt
        radius = 2.0;           // ausgeführt (vor dem Konstruktor)
    }
}
```

4.7.1 Erzeugung von Objekten

Lebensdauer von Objekten

```
{
    Student s;
    {
        Student thomas = new Student("Thomas");
        Student tom = new Student("Tom");
        // 'Tom' u. 'Thomas' sind verschiedene Studenten
        tom = thomas;
        // Hier wurde nur die Referenz kopiert!
        // tom u. thomas verweisen jetzt auf dasselbe Objekt!
        // Das Objekt 'Tom' ist nicht mehr zugreifbar!
        s = thomas;
    }
    // Das Objekt 'Thomas' existiert noch!
    // (s ist eine Referenz auf 'Thomas')
}
```

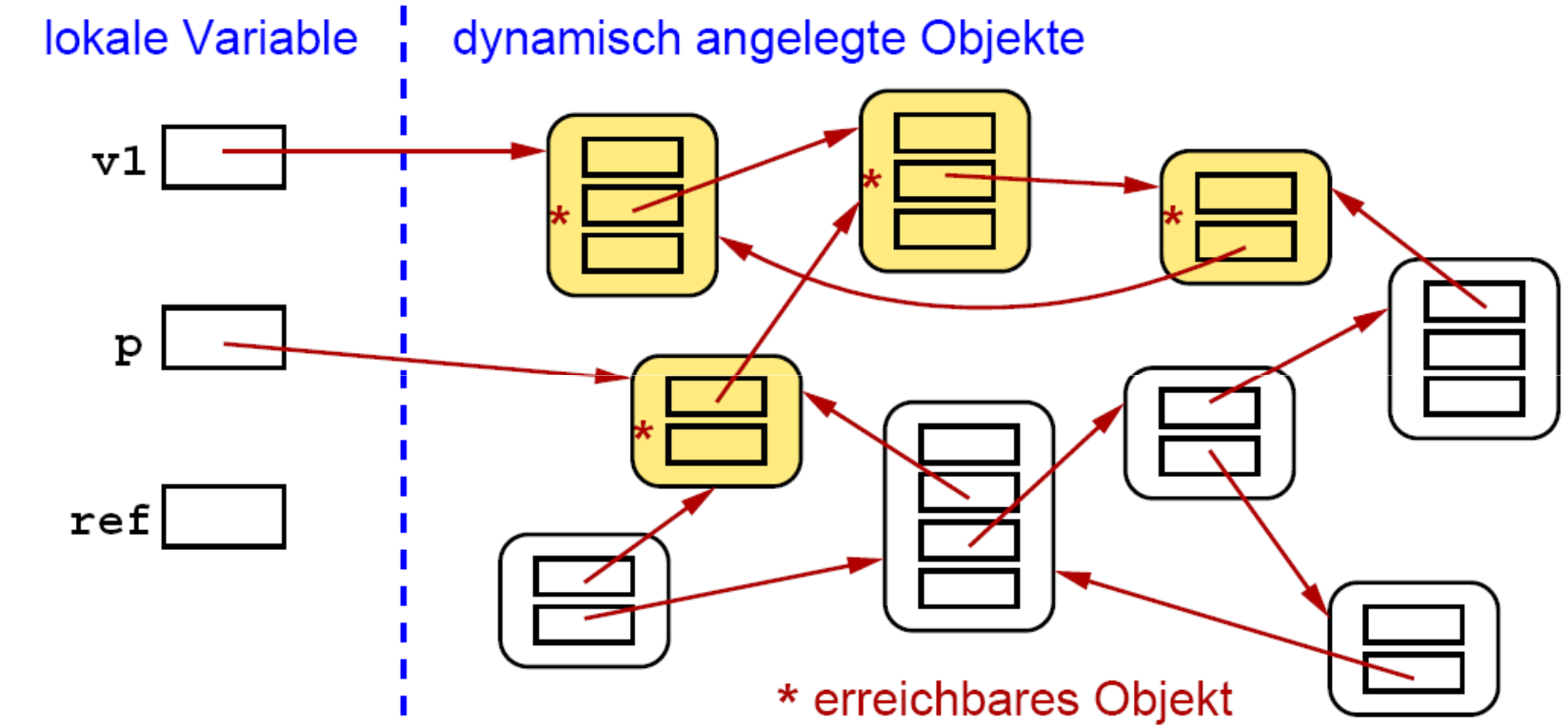
4.7.1 Erzeugung von Objekten

Freigabe von Speicherbereichen

- In Java existiert ein einmal erzeugtes Objekt weiter, solange es noch eine Möglichkeit gibt auf das Objekt zuzugreifen
  - d.h., solange es noch über eine Kette von Referenzen von einer Variable aus erreicht werden kann
- Ein mit new angelegter Speicherbereich (Objekt oder Array) wird von der JVM **automatisch** wieder freigegeben, wenn
  - Speicherplatz benötigt wird und
  - der Speicherbereich nicht mehr zugreifbar ist
- Die Suche nach solchen Speicherbereichen und deren Freigabe heißt ***Garbage Collection***

4.7.1 Erzeugung von Objekten

Beispiel zur *Garbage Collection*



Wenn `ref` ungültig (oder mit `null` belegt) wird, markiert der *Garbage Collector* alle erreichbaren Objekte. Die unmarkierten werden dann gelöscht.

4.7.2 Zugriff auf Attribute und Methoden

- Die Attribute und Methoden eines Objekts können über eine Objektreferenz angesprochen werden:

```
<Objektreferenz> . <Attributname>
<Objektreferenz> . <Methodenname> ( [<Parameterliste>] )
```

  - gilt auch für Klassenattribute / -methoden
  - Voraussetzung: Sichtbarkeit erlaubt den Zugriff
  - Beispiele:

```
meineKugel.radius                // Attribut
meineKugel.anzahl                // Klassenattribut
meineKugel.volumen()             // Methode
```

- Klassenattribute/-methoden können unabhängig von einer Objektreferenz auch über den Klassennamen angesprochen werden
  - Beispiel: `Kugel.anzahl`

4.7.2 Zugriff auf Attribute und Methoden ...

Namen in Methoden

- Ein **qualifizierter Name** ist ein Name mit expliziter Angabe des Objekts oder der Klasse
  - z.B.: `meineKugel.radius, Kugel.anzahl`
- Alle andern Namen (ohne Punkt) heißen **einfache Namen**
  - z.B.: `radius, i, anzahl`
- Eine Methode kann auf folgende Methoden über einfache Namen zugreifen:
  - die Klassenmethoden der eigenen Klasse
  - die Methoden des eigenen Objekts (nur, wenn die aufrufende Methode keine Klassenmethode ist)

4.7.2 Zugriff auf Attribute und Methoden ...

Namen in Methoden ...

- Eine Methode kann auf folgende Variablen über einfache Namen zugreifen:
  - die Klassenattribute der eigenen Klasse
  - die Attribute des eigenen Objekts (nur, wenn die Methode keine Klassenmethode ist)
- Eine Methode greift immer über einfache Namen zu auf:
  - ihre Parameter
  - ihre lokalen Variablen
- Beispiel (neue Methode der Klasse Kugel):

```
double volumen() {
   return 4.0/3.0 * PI * radius * radius * radius;
} // PI: Klassenattribut; radius: Attribut
```

4.7.2 Zugriff auf Attribute und Methoden ...

Die Referenzvariable `this`

- Existiert in jeder Methode (außer in Klassenmethoden)
- Muß / darf nicht deklariert werden
- Zeigt immer auf das eigene Objekt
  - Vorstellung: `this` ist ein Parameter, in dem der Methode eine Referenz auf das eigene Objekt übergeben wird
- Nützlich z.B. zur Unterscheidung von Attributen und gleichnamigen Parametern:

```
void neuerRadius(double radius)
{
   // radius: Parameter; this.radius: Attribut
   this.radius = radius;
}
```

4.7.3 Aufruf von Methoden

- Syntax: <Name> ([<Parameterliste>])
  <Parameterliste> ::= <Ausdruck> {, <Ausdruck>}
  - Name ist ggf. ein qualifizierter Name
- Beispiel:

```
class Kugel {
  ...
  private double cube(double x) {
    return x * x * x;
  }
  public double volume() {
    return 4.0/3.0 * PI * cube(radius);
  }
}
```

formaler Parameter

aktueller Parameter (Argument)

Methodenaufruf

4.7.3 Aufruf von Methoden ...

Ablauf eines Methodenaufrufs

1. Die Ausdrücke in der Parameterliste werden ausgewertet

2. Die **Werte** der aktuellen Parameter werden an die formalen Parameter der Methode zugewiesen (***call by value***)

3. Der Methodenrumpf wird ausgeführt
   - bis zum Ende (nur bei void-Methoden erlaubt)
   - oder bis zur Anweisung return [<Ausdruck>]
     - der Wert von <Ausdruck> bestimmt ggf. den Wert des Methodenaufrufs (der selbst ein Ausdruck ist)

4. Die Abarbeitung der aufrufenden Methode wird nach dem Methodenaufruf fortgesetzt

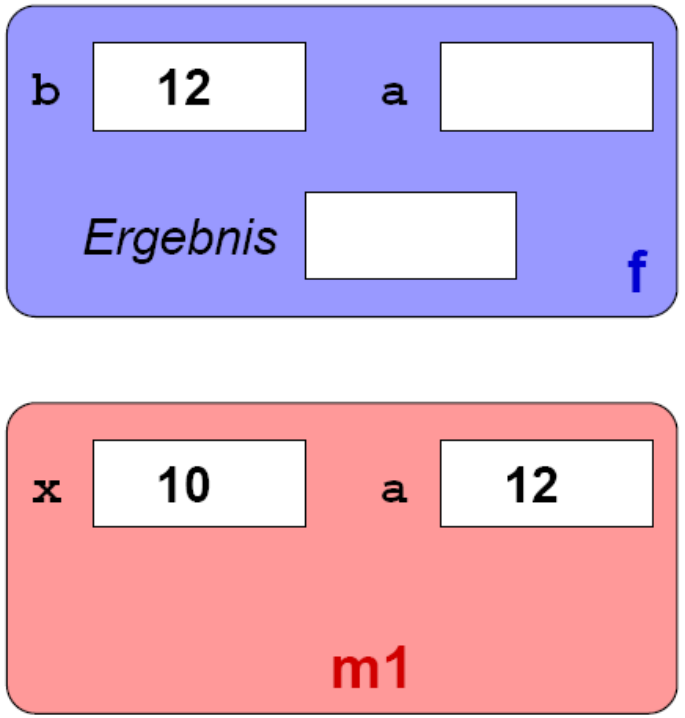
(Bei überladenen Methoden wird die zu Anzahl und Typen der aktuellen Parameter passende Methode bereits vom Compiler ausgewählt)

4.7.3 Aufruf von Methoden ...

Beispiel zum Ablauf eines Methodenaufrufs

```
int f(int b, int a)
{
    a = 2 * b + a * a;
    return a + 1;
}


void m1()
{
    int x = 10;
    int a = 12;
    a = f(a, ++x) - f(a, a+3);
    ...
}
```



Stapel (Keller, Stack) von Aktivierungsrahmen, die lokale Variable der Methoden speichern

4.7.3 Aufruf von Methoden ...

Referenzen als Parameter

- Auch Objekte können Parameter von Methoden sein
- übergeben werden dabei jedoch nicht die Objekte, sondern nur **Referenzen** auf diese Objekt (***call by reference***)
- Die aufgerufene Methode kann dabei die übergebenen Objekte verändern
  - so lassen sich auch Ein-/Ausgabe-Parameter realisieren
  - unerwartetes Verändern übergebener Objekte sollte aber vermieden werden!
- Analog können Objekt-Referenzen auch als Ergebnis einer Methode auftreten

4.7.3 Aufruf von Methoden ...

Beispiel: Vertauschen von Werten

```
class Bsp1 {
  void swap(int x, int y) {
    int tmp = x;
    x = y;
    y = tmp;
  }
  void m1() {
    int a, b;
    a = 1; b = 2;
    swap(a, b);
    // a = 1, b = 2 !
  }
}
```

```
class Pair {
  public int x;
  public int y;
}

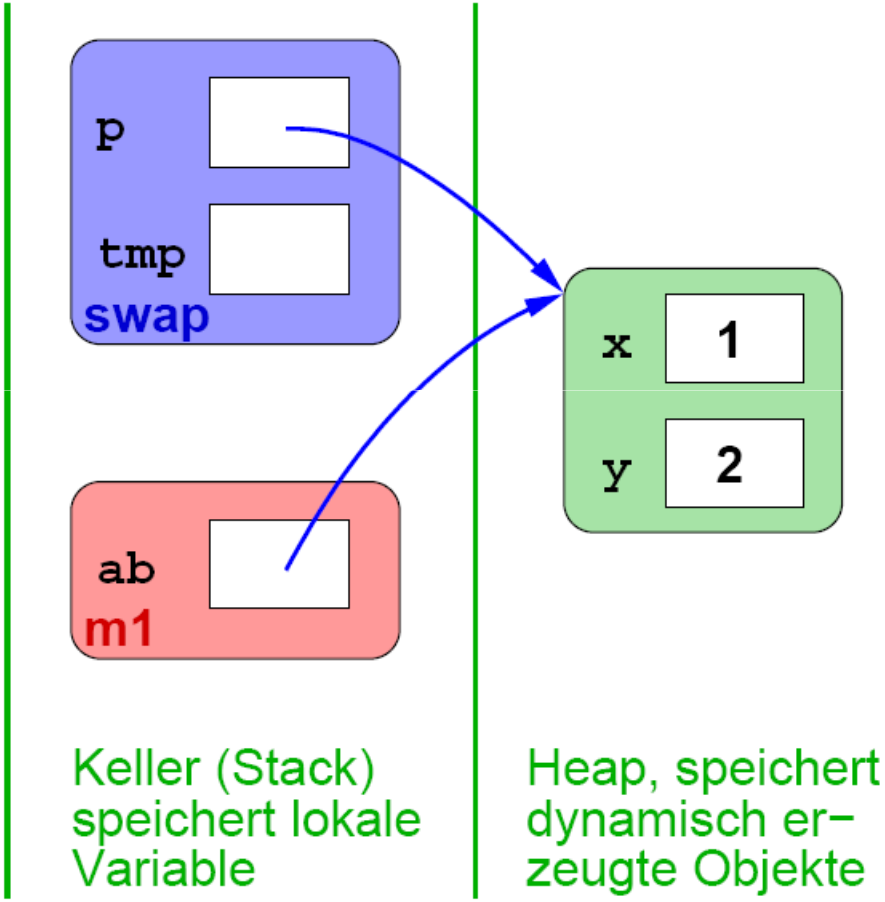
class Bsp2 {
  void swap(Pair p) {
    int tmp = p.x;
    p.x = p.y;
    p.y = tmp;
  }
  void m1() {
    Pair ab = new Pair();
    ab.x = 1; ab.y = 2;
    swap(ab);
    // ab.x = 2, ab.y = 1 !
  }
}
```

4.7.3 Aufruf von Methoden ...

Ablauf des Beispiels

```
void swap(Pair p) {
   int tmp = p.x;
   p.x = p.y;
   p.y = tmp;
}

void m1() {
   Pair ab = new Pair();
   ab.x = 1; ab.y = 2;
   swap(ab);
}
```



4.7.3 Aufruf von Methoden ...

Rekursion

- Eine Methode kann sich auch selbst aufrufen
- Beispiel: Fakultätsfunktion

Mathematische Definition:

$$n! = \begin{cases} 1 & \text{für } n = 0 \\ n \cdot (n-1)! & \text{für } n > 0 \end{cases}$$

Java-Code:

```
static long fak(long n) {
  if (n == 0)
    return 1L;
  else if (n > 0)
    return n * fak(n-1);
}
```

- Jede Aktivierung einer Methode bekommt einen eigenen Aktivierungsrahmen auf dem Stapel
  - damit hat jede Aktivierung ihre eigenen lokalen Variablen

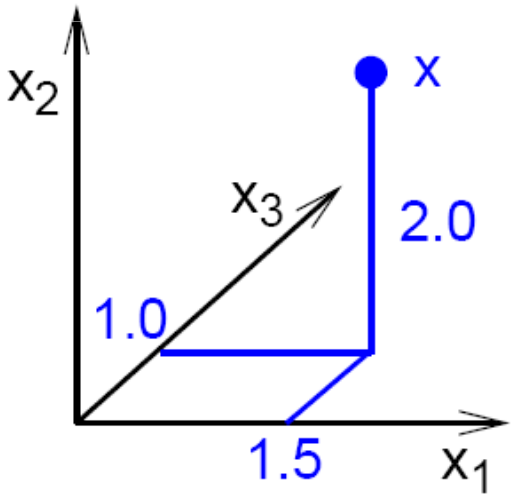
4.8 Arrays

- In einem Array kann man mehrere Variablen des gleichen Typs zusammenfassen
  - vgl. Attribute mit Multiplizität > 1 in UML
- Beispiel: Koordinaten eines Punkts im Raum:
  - Mathematisch: ~x = (x1, x2, x3) = (1.5, 2.0, 1.0)
  - In Java: Array mit Elementen vom Typ double

```
double[] x = new double[3];
x[0] = 1.5;
x[1] = 2.0;
x[2] = 1.0;
```

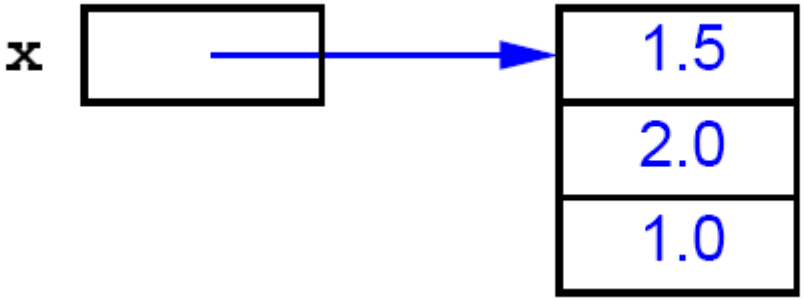
oder kürzer:

```
double[] x = { 1.5, 2.0, 1.0 };
```



4.8 Arrays …

- Ein Element eines Arrays wird über einen ganzzahligen **Index** ausgewählt
  - Z.B. `x[1]` oder `x[i+j]`
  - Das erste Element hat den Index 0 !
  - Bei Indexüberlauf: Ausnahme (`IndexOutOfBoundsException`)
- Arrays müssen wie Objekte dynamisch angelegt werden (`new`)
- Das eigentliche Array wird über eine Referenz angesprochen:



- Arrays besitzen ein ”Attribut“ length: Anzahl der Elemente

4.8 Arrays …

Beispiele

- Deklaration:

```
double[] x;                    // Bevorzugte Schreibweise
double y[];                    // Geht auch
```

- Erzeugung des (eigentlichen) Arrays

```
x = new double[3];             // Drei Elemente
                               // Initialwerte: 0
```

- Deklaration, Erzeugung und Initialisierung

```
int[] z = { 1, 2, 3, 4 };      // automatisches new
```

- Zugriff auf Array-Elemente:

```
for (int i = 0; i < x.length; i++)      // x.length = Länge
    x[i] = 3.0 - i;
x[0] = 1.5;                             // ersetzt alten Wert 3.0
```

Beispiele ...

- Zugriffe auf das ganze Array:

```
double[] y = x;                 // erzeugt neue Referenz-
                                // variable y, die auf dasselbe
                                // Array zeigt wie x
y[2] = 4.0;                     // ändert auch den Wert von x[2]!

// So wird z eine echte Kopie von x:

double[] z = new double[x.length];
for (int j = 0; j < x.length; j++)
   z[j] = x[j];
```

Arrays von Objekten

- Wie in einfachen Variablen werden in Arrays nur **Objektreferenzen** gespeichert
- Beispiel: Array mit 2 Studenten

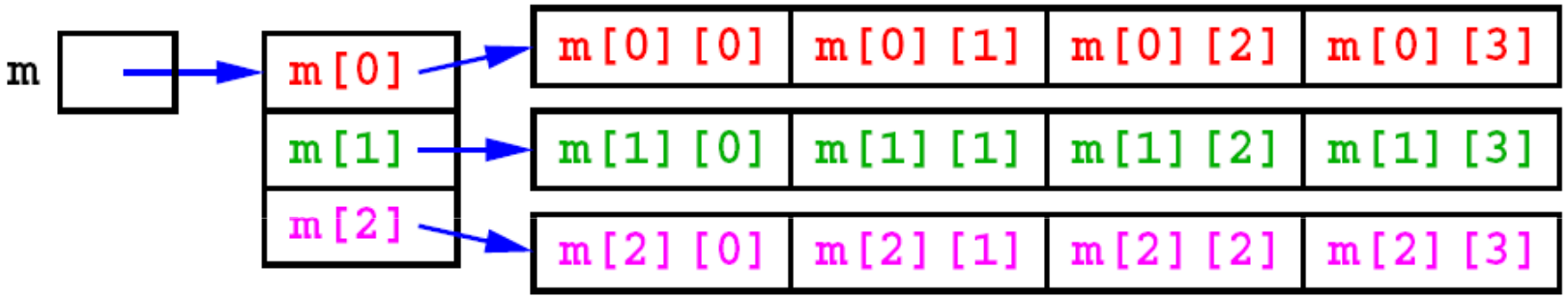
```
Student[] a1 = new Student[2]; // Initialisierung mit null!
a1[0] = new Student("Hans"); // Init. der Elemente
a1[1] = new Student("Fritz");
```

- Kürzer:

```
Student[] a2 = {                // Delaration mit
   new Student("Hans"),         // Initialisierung der
   new Student("Fritz")         // Elemente
};
```

Mehrdimensionale Arrays

- Die Elemente eines Arrays können auch Referenzen auf Arrays enthalten:



- Deklaration der Referenzvariable:

```
int[][] m;          // 2-dimensionales Array: Matrix
int[][][] mat3d;    // 3-dimensionale Matrix
int m3d[][][];      // 3-dimensionale Matrix, alternativ
```

4.8 Arrays …

- Erzeugung des Arrays:

```
m = new int[3][4];
```

// m =

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

```
m[2] = new int[2];
m[2][1] = 7;
```

// m =

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 7 & & \end{pmatrix}$$

- Anmerkungen:
  - `m[2]` ist eine Referenz auf ein Array von `int`-Zahlen
  - `m[2]` hat den Typ `int[]`
  - `m.length == 3`
  - Die Länge der Zeilen kann variieren:
    `m[0].length == 4`, aber `m[2].length == 2`

Mehrdimensionale Arrays ...

- `m = new int[3][4]`

  ist gleichbedeutend mit:

```
m = new int[3][];              // Lege Array für 3 Zeilen−
                               // verweise an
for (int i=0; i<3; i++)        // Initialisiere die
   m[i] = new int[4];          // Zeilenverweise
```

- Aber:

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

  führt zu Fehlermeldung, da

  - die Referenzen auf die Zeilen nirgends abgespeichert werden können
  - die Anzahl der zu erzeugenden Zeilen unbekannt ist

4.9 Strings

- In Java ist ein String (Zeichenkette) ein **Objekt** der Klasse String
- Beispiel:

```
String motto = "Wir lernen Java!";     // automatisches new!
```

- `motto` ist eine Referenzvariable
  - speichert nicht den String, sondern nur die Referenz darauf
- Ein String ist eine Folge von (Unicode-)Zeichen, jedes Zeichen hat eine Position (gezählt ab 0):

- Aber: Ein String ist **kein** Array!

4.9 Strings …

- Nach einer neuen Zuweisung an die Referenzvariable, z.B.

  ```
  motto = "Carpe Diem";
  ```

  ist der String `"Wir lernen Java!"` nicht mehr zugreifbar und wird vom *Garbage Collector gelöscht.*

- Die folgenden Zuweisungen sind unterschiedlich!

  ```
  motto = null;          // motto zeigt auf keinen String mehr
  motto = "";            // motto zeigt auf den leeren String
  ```

4.9 Strings …

Operationen auf Strings

- Zusammenfügen (Konkatenation)

```
String s = "Zahl 1" + "2";          // ”Zahl12”
s += " ist gleich 3";               // ”Zahl12 ist gleich 3”
s = "elf ist " + 1 + 1;             // ”elf ist 11”
s = 1 + 1 + " ist zwei";            // ”2 ist zwei”
s = 0.5 + 0.5 + " ist " + 1;        // ”1.0 ist 1”
```

- Der Operator + ist überladen:
  - `int + int`: Addition
  - `String + String`: Konkatenation
  - `String + int` und `int + String`: Umwandlung der Zahl in einen String und anschließende Konkatenation
  - analog für andere Datentypen

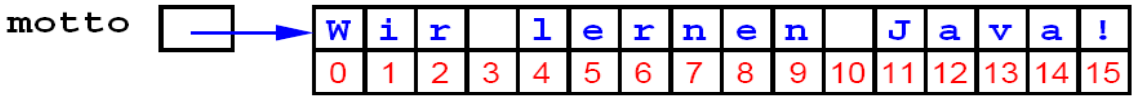
4.9 Strings …

Operationen auf Strings ...

- Vergleichsoperatoren (nur == und !=):
  - der Operator == liefert true, wenn beide Operanden auf **denselben** String verweisen (Objektidentität!)
- Vergleich:
  - `s1.equals(s2)`
    liefert true, wenn `s1` und s2 zeichenweise übereinstimmen
  - `s1.compareTo(s2)`
    < 0, wenn s1 alphabetisch vor s2
    = 0, wenn s1 und s2 zeichenweise übereinstimmen
    > 0, wenn s1 alphabetisch nach s2
- Länge: `motto.length()`

4.9 Strings …

Weitere Methoden der Klasse String



- Vergleich mit Anfang und Ende:

```
boolean a = motto.startsWith("Wir");          // true
boolean b = motto.endsWith(".");              // false
```

- Zugriff auf einzelne Zeichen:

```
char c = motto.charAt(5);                     // 'e'
```

- Suche nach Zeichen:

```
int i = motto.indexOf('e',0);                 // 5
int j = motto.lastIndexOf('e',15);            // 8
```

- Ausschneiden / Ersetzen:

```
String s = motto.substring(11,15);            // "Java!"
String t = motto.replace('a','A');            // "Wir lernen JAvA!"
```

4.9 Strings …

Strings und Arrays

- String in Array von char umwandeln:

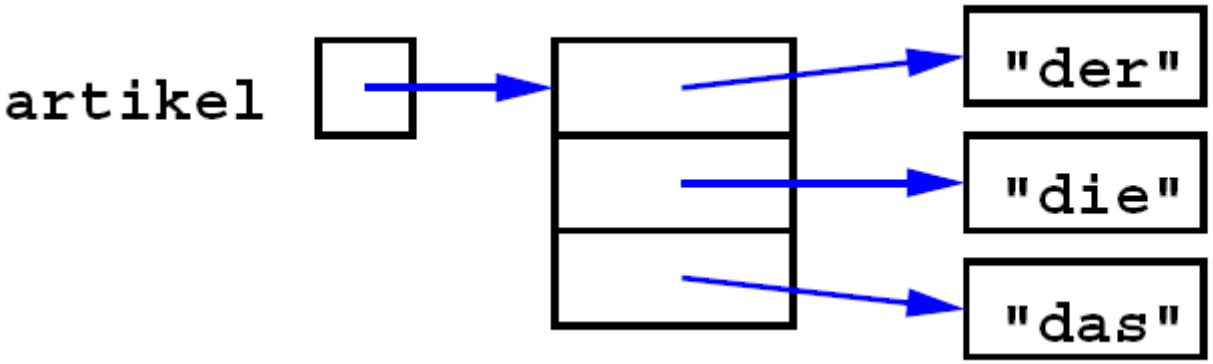
```
char[] textArray = motto.toCharArray();
```

- Array von char in String umwandeln:

```
String str = new String(textArray);
```

- Array von Strings:

```
String[] artikel = { "der", "die", "das" };
```



4.9 Strings …

Anmerkung

- Strings können **nicht** verändert werden (”*immutable“)*
  - die Operation + oder z.B. die Methode replace erzeugen jeweils einen **neuen** String
  - dadurch verhalten sich Strings wie einfache Datentypen
    - z.B. *call-by-value-Semantik bei Parameter übergabe*
- Es gibt auch eine Klasse `StringBuffer`, deren Objekte auch verändert werden können
  - Geschwindigkeitsvorteil, wenn viele Manipulationen an Strings vorgenommen werden
- Dokumentation der Klasse String im WWW unter

  `http://java.sun.com/javase/6/docs/api/java/lang/String.html`

4.9 Strings …

StringBuffer

Beispiel:

```
// Klasse StringBufferTest
      public class StringBufferTest { public static void main(String[] args) {
// StringBuffer mit Anfangsstring
      StringBuffer buf = new StringBuffer("Java ist schwer");
// Einfügen eines Strings
      buf.insert(9, "nicht ");
// Anhängen eines Strings
       buf.append("er als C++!");
// Modifizierter String String
      resultat = buf.toString();
// Ausgabe des modifizierten Strings
      System.out.println(resultat); } }
```

Java ist nicht schwerer als C++!

4.10 Vererbung und Polymorphie

Konstruktoren und Vererbung

- Die Konstruktoren einer Klasse werden **nicht** an die Unterklassen vererbt
- In einem Konstruktor kann mittels **super**([<Parameterliste>]) ein Konstruktor der Oberklasse aufgerufen werden:

```
class Shape {
   Shape(int color) { ... }
   ...
}
class Circle extends Shape {
   Circle(double[] center, double radius, int color) {
     super(color);                    // ruft Konstruktor Shape(int color),
     ...                              // muß erste Anweisung sein!
   }
   ...
}
```

Konstruktoren und Vererbung ...

- Vor der Ausführung eines Unterklassen-Konstruktors wird **immer** ein Konstruktor der Oberklasse ausgeführt
  - falls kein expliziter Aufruf erfolgt, wird der Default-Konstruktor der Oberklasse ausgeführt
- Beispiel:

```
class A {
  A() { ... A }
}
class B extends A {
  B(int i) { ... B }
}
class C extends B {
  C() { super(1); ... C }
}
```

implizites super()

Reihenfolge der Konstruktor-**Aufrufe** bei `new C()`:
C() → B(1) → A()

Reihenfolge der **Abarbeitung** der Konstruktor-Rümpfe:
A → B → C

Die ”Referenzvariable“ super

- In Instanzmethoden abgeleiteter Klassen gibt es eine spezielle ”Referenzvariable“ super
- Sie erlaubt u.a. den Zugriff auf überschriebene Methoden der
- Basisklasse:

```
class Ober {
    int op(int i) { ... }
}
class Unter extends Ober {
    int op(int i) { return super.op(i) / 2; }
}
```

Ruft `op(i)` in Klasse `Ober`

- Wie `this` muß auch super nicht deklariert werden
- Im Gegensatz zu this ist super aber keine Referenz auf ein reales Objekt!

4.10 Vererbung und Polymorphie …

Beispiel UEFA 2012

- Entwicklung eines Fußballmanagers
- Klassen **Person**, Spieler, Trainer, Torwart, Mannschaft

```
public class Person{
        //Eigenschaften einer Person
        private String name;
        private int alter;

      // Konstruktoren
        public Person(String n, int a){
           name  = n;
          alter = a;      }

      // Funktionen (get und set)
        public String getName(){
           return name;  }
         public void setName(String n){
           name = n;     }

        …
}
```

4.10 Vererbung und Polymorphie …

Beispiel UEFA 2012

- Entwicklung eines Fußballmanagers
- Klassen Person, Spieler, **Trainer**, Torwart, Mannschaft

```
public class Trainer extends Person{
        //Zusätzliche Eigenschaften eines Trainers
        private int erfahrung;   // int aus 1..10

      // Konstruktoren
        public Trainer(String n, int a, int e){
           super(n,a);
           erfahrung = e;      }

      // Funktionen (get und set) Erfahrung

        …
}
```

„Erzeuge Trainer“ / neues Objekt Trainer und Parameter setzen

```
Trainer trainer = new Trainer();
        trainer.setName(„Löw“);
        trainer.setAlter(45);
        trainer.setErfahrung(6);
```

4.10 Vererbung und Polymorphie …

Beispiel UEFA 2012

- Entwicklung eines Fußballmanagers
- Klassen Person, **Spieler**, Trainer, Torwart, Mannschaft

```
public class Spieler extends Person {
   // Zusätzliche Eigenschaften eines Spielers:
   private int staerke;
   private int torschuss;
   private int motivation;
   private int tore;

   // Konstruktor
   public Spieler(String n, int a, int s, int t, int m){
      super(n, a);
      staerke     = s;
      torschuss   = t;
      motivation  = m;
      tore        = 0;
   }
```

4.10 Vererbung und Polymorphie …

Beispiel UEFA 2012 Spieler


```
// Funktionen (get und set):

   public int getStaerke(){
      return staerke;     }
    public void setStaerke(int s){
      staerke = s; }

   public int getTorschuss(){
      return torschuss; }
   public void setTorschuss(int t){
      torschuss = t; }

   public int getMotivation(){
      return motivation;}
   public void setMotivation(int m){
      motivation = m; }

   public int getTore(){
      return tore;}
   public void addTor(){
      tore++;
           motivation = Math.min(10, motivation+1);}
```

4.10 Vererbung und Polymorphie …

Beispiel UEFA 2012 Torwart


```
import java.util.Random;
public class Torwart extends Spieler{
   // Zusätzliche Eigenschaften eines Torwarts:
   private int reaktion;

   // Konstruktor
   public Torwart(String n, int a, int s, int t, int m, int r){
      super(n, a, s, t, m); reaktion    = r;}

   // Funktionen (get und set):
   public int getReaktion(){ return reaktion; }
   public void setReaktion(int r){ reaktion = r; }

   // Torwartfunktionen:
   public boolean haeltDenSchuss(int schuss){Random r = new Random();
      // Reaktion kann abweichen [+1,0,-1]
      int ret = reaktion + r.nextInt(3)-1;
      if (ret >= schuss)
         return true;  // Ball gehalten
      else
         return false; // TOR!!! }
}
```

Beispiel UEFA 2012 Mannschaft


```
public class Mannschaft{
   // Eigenschaften einer Mannschaft
   private String name;
   private Trainer trainer;
   private Torwart torwart;
   private Spieler[] kader;

   // Konstruktoren
   public Mannschaft(String n, Trainer t, Torwart tw, Spieler[] s){
      name     = n;
      trainer  = t;
      torwart  = tw;
      kader    = s;
   }

   // Funktionen (get und set)

         ...
```

4.10 Vererbung und Polymorphie …

Polymorphie (griech.”Vielgestaltigkeit“)

- Eigenschaft eines Bezeichners (Operation, Funktion, Variable, ...), je nach Umgebung verschiedene Wirkung zu zeigen
- Verschiedene Arten der Polymorphie:
  - Überladen von Bezeichnern (z.B. ’+’ für `int, double, String`)
  - parametrisierbare Datentypen / Klassen (Typ als Parameter)
  - polymorphe Funktionen können Ergebnisse unterschiedlichen Typs liefern
  - polymorphe Variable können je nach Umgebung verschiedenartige Größen bezeichnen
- Referenzvariablen sind polymorphe Variablen
  - können auf Objekte unterschiedlicher Klassen verweisen

4.10 Vererbung und Polymorphie …

Beispiel zur Polymorphie

```
import java.util.Random;
abstract class Tier {
  protected String gattung;
  public Tier(String gattung) { this.gattung = gattung; }
  public void print() {
    System.out.println("Ich bin ein Tier der Gattung "
                       + gattung + ".");
  }
}
class Hund extends Tier {
  protected String name;
  protected String rasse;
  public Hund(String aName, String aRasse) {
    super("Hund"); name = aName; rasse = aRasse;
  }
  public void print() {
    System.out.println("Ich bin " + name + ", der "
                       + rasse + ".");
  }
}
```

System.out = Ausgabestrom für Konsole
println() = Ausgabeoperation

4.10 Vererbung und Polymorphie …

Beispiel zur Polymorphie ...

```
class Katze extends Tier {
  protected String name;
  public Katze(String n) { super("Katze"); name = n; }
  public void print() {
    System.out.println("Ich bin " + name + ".");
    super.print();
  }
}
public class Demo {
  public static void main(String[] args) {
    Tier[] meinZoo = new Tier[3];
    meinZoo[0] = new Hund("Waldi", "Dackel");
    meinZoo[1] = new Katze("Chanty");
    meinZoo[2] = new Hund("Hasso", "Boxer");
    Random rnd = new Random();
    for (int i=0; i<4; i++) {
      Tier t = meinZoo[rnd.nextInt(meinZoo.length)];
      t.print();
    }
  }
}
```

4.10 Vererbung und Polymorphie …

Beispiel zur Polymorphie ...

```
class Katze extends Tier {
  protected String r
  public Katze(Strir                                    n; }
  public void print() {
    System.out.println("Ich bin " + name + ".");
    super.print();
  }
}
public class Demo {
  public static void main(String[] args) {
    Tier[] meinZoo = new Tier[3];
    meinZoo[0] = new Hund("Waldi", "Dackel");
    meinZoo[1] = new Katze("Chanty");
    meinZoo[2] = new Hund("Hasso", "Boxer");
    Random rnd = new Random();
    for (int i=0; i<4; i++) {
      Tier t = meinZoo[rnd.nextInt(meinZoo.length)];
      t.print();
    }
  }
}
```

meinZoo[] enthält Verweise auf Objekte verschiedener Klassen (Hund, Katze)

t ist eine polymorphe Variable. Sie verweist auf einen Hund oder eine Katze.

Die aufgerufene Methode richtet sich nach dem konkreten Objekt, auf das t verweist!

4.10 Vererbung und Polymorphie …

Mögliche Ausgabe des Beispiels:

```
Ich bin Chanty.                          } print() von Katze
Ich bin ein Tier der Gattung Katze.      }
Ich bin Hasso, der Boxer.                  print() von Hund
Ich bin Waldi, der Dackel.                 print() von Hund
Ich bin Chanty.                          } print() von Katze
Ich bin ein Tier der Gattung Katze.      }
```

4.10 Vererbung und Polymorphie …

Binden von Bezeichnern

- Bezeichner können zu unterschiedlichen Zeiten an Objekte, Datentypen oder Datenstrukturen gebunden werden:
  - zur Übersetzungszeit (**statische Bindung, frühe Bindung**)
  - zur Laufzeit (**dynamische Bindung, späte Bindung**)
- Methoden werden in Java **dynamisch** gebunden
  - abhängig vom Objekt, für das die Methode aufgerufen wird (Polymorphie)
  - Merkregel: jedes Objekt kennt seine Methoden!
- Klassenmethoden und Attribute werden **statisch** gebunden
- abhängig vom Typ der Referenzvariable, die für den Zugriff benutzt wird

4.10 Vererbung und Polymorphie …

Beispiele:

```
Interface I {
  public void op(int i);
}
class OK implements I {
  public int attr = 1;
  public static int kattr = 10;
  public void op(int i) { ... }
  public static void kop() { ... }
}
class UK extends OK {
  public int attr = 2;
  public static int kattr = 20;
  public void op(int i) { ... }
  public void op(double d) { ... }
  public static void kop() { ... }
}
```

```
I[] i = { new OK(),
          new UK() };
i[0].op(1);  // op() aus OK
i[1].op(2);  // op() aus UK

OK o = new UK();
UK u = new UK();
o.op(1);     // op() aus UK
u.op(1);     // op() aus UK
o.op(1.5);   // FEHLER!!!
u.op(1.5);   // op() aus UK
z = o.attr;  // attr aus OK
z = u.attr;  // attr aus UK
o.kop();     // kop() aus OK
u.kop();     // kop() aus UK
z = o.kattr;// kattr aus OK
z = u.kattr;// kattr aus UK
```

Konversion von Objekttypen

- Eine neue Klasse definiert in Java auch einen neuen Typ
  - z.B.:
    ```
    Tier einTier;
    Hund fido = new Hund("Fido", "Spaniel");
    ```
    `einTier` und `fido` sind Variablen verschiedenen Typs
- Java ist **typsicher**
  - Zuweisungen und Operationen sind nur mit Variablen bzw. Ausdrücken des korrekten Typs erlaubt
- Warum können wir dann
  ```
  einTier = fido;
  ```
  schreiben?
- Antwort: Java macht eine **implizite Typkonversion** von einer Klasse zu einer Oberklasse (bzw. implementierten Schnittstelle)!

Konversion von Objekttypen ...

- Explizite Typkonversionen:

```
einTier = (Tier)fido;                    // korrekt, unnötig
fido = (Hund)einTier;                    // korrekt!
Student s = (Student)einTier;            // Compilerfehler!
Katze pucki = (Katze)einTier;            // Laufzeitfehler!
```

- Typkonversion bewegt sich in der Klassenhierarchie immer nur
  - aufwärts (implizite oder explizite Konversion), oder
  - abwärts (nur explizite Konversion)

- Test des Objekttyps über Operator instanceof möglich:

```
if (einTier instanceof Katze) {
Katze pucki = (Katze)einTier;            // OK!
```

4.11 Die universelle Klasse `Object`

- Java definiert eine Klasse `Object`, die an der Spitze **jeder** Klassenhierarchie steht
- Alle Klassen, die nicht explizit von einer anderen Klasse abgeleitet werden, sind (implizit) von Object abgeleitet:

```
public class Tier { ... }
```

heißt damit

```
public class Tier extends Object { ... }
```

- Die Klasse Object definiert 9 Methoden, die **alle** Klassen erben (direkt oder indirekt)
  - siehe Java Klassendokumentation: http://java.sun.com/ `javase/6/docs/api/java/lang/Object.html`
  - diese Methoden können wie alle anderen auch überschrieben werden!

4.11 Die universelle Klasse `Object` ...

Einige Methoden der Klasse Object:

- `public String toString()`
  - gibt eine textuelle Darstellung des Objekts zurück
  - Implementierung in Object: “Klassenname @ Adresse”
- `public boolean equals(Object obj)`
  - stellt fest, ob zwei Objekte gleich sind
  - Implementierung in `Object: return this == obj;`
- `protected Object clone()`
  - erzeugt eine Kopie des Objekts
  - Implementierung in Object kopiert alle Instanzvariablen
- `protected void finalize()`
  - aufgerufen, bevor Objekt vom *Garbage Collector gelöscht wird*
  - Implementierung in `Object` tut nichts

4.11 Die universelle Klasse `Object` ...

`Object, toString()` und `println()`

- System.out.println() kann beliebige Objekte ausdrucken, z.B.:

```
class Hund extends Tier {
   ...
   public String toString() {
     return "Ich bin " + name + ", der " + rasse + ".";
   }
}
...
   Hund waldi = new Hund ("Waldi", "Dackel");
   System.out.println(waldi);
```

- Wie geht das? So:

```
public void println(Object obj) {
   String str = obj.toString();        // Polymorphie!!
   println(str);                       // Aufruf der überladenen Methode
}
```

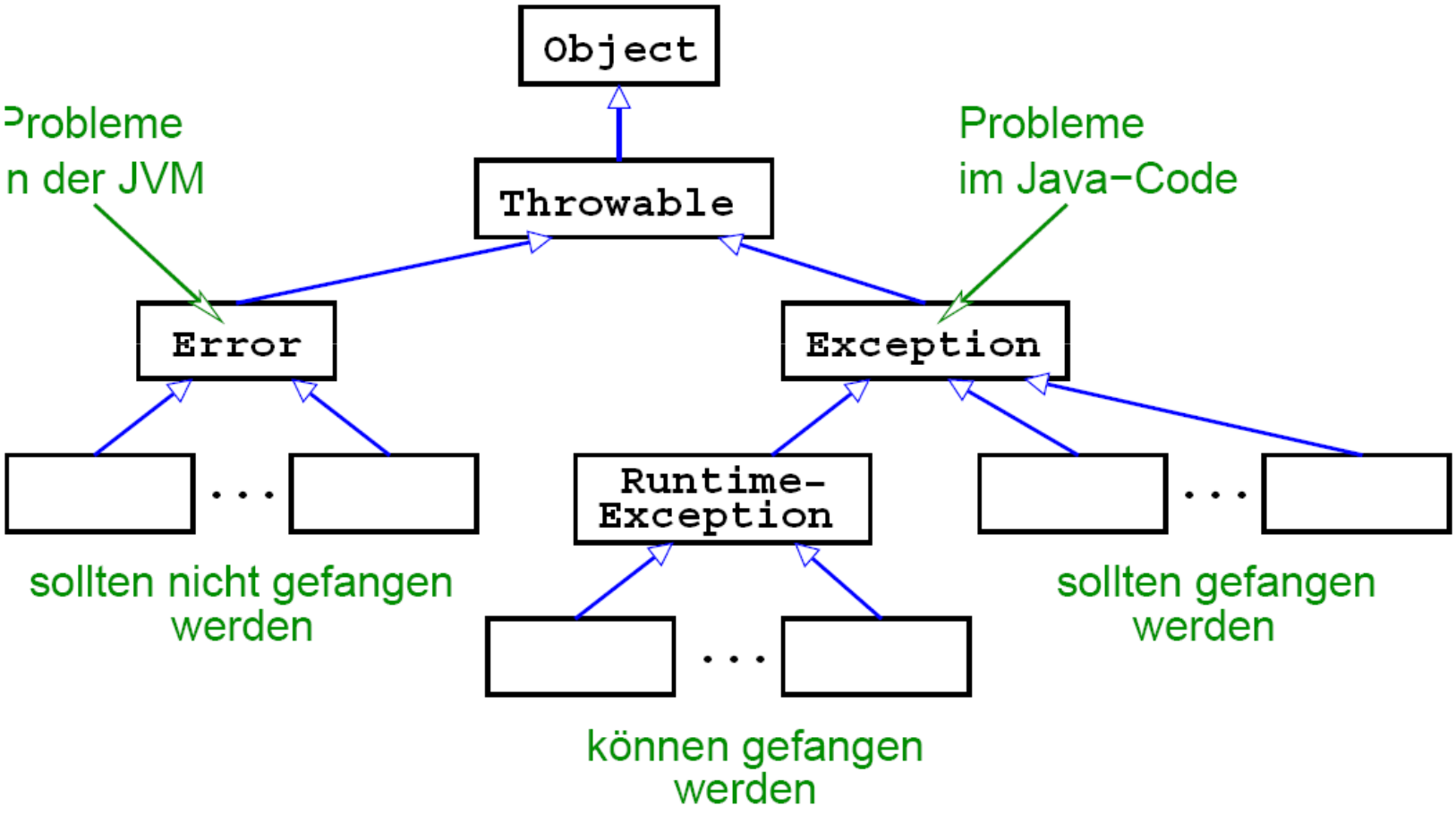
4.12 Exceptions

- **Exceptions (Ausnahmen)** signalisieren Fehler zur Laufzeit eines Programms
- Sie können
  - implizit durch Java-Anweisungen (z.B. x/0, a[-1])
  - explizit durch die Anweisung `throw`

  ausgelöst (”**geworfen**“) werden
- Exceptions sind Ausnahmesituationen – sie sollten außerhalb des normalen Programmcodes behandelt (”**gefangen**“) werden
  - höhere Übersichtlichkeit des Codes!
- In Java sind Exceptions **Objekte,** die in einer Fehlersituation dynamisch erzeugt werden
  - ihre Attribute beschreiben den Fehler genauer

4.12 Exceptions …

Exceptions: Objekthierarchie



4.12 Exceptions …

Unterklasse `RuntimeException`

- `RuntimeExceptions` werden meist durch Fehler im Programmcode verursacht
- Sie müssen daher nicht behandelt werden
- Beispiele (siehe auch Dokumentation zum Paket java.lang):

```
ArithmeticException: z.B. 1/0 (ganzzahlig!)
IndexOutOfBoundsException: z.B. array[-1]
NegativeArraySizeException: z.B. new double[-5]
NullPointerException:
   z.B. Hund meinHund = null; meinHund.gibLaut();
ClassCastException:
   z.B. Tier t = new Hund("fido"); Katze k = (Katze)t;
```

Behandlung von Exceptions

- Einfachster Fall: Exception wird nicht behandelt
  - Methode bricht beim Auftreten der Exception **sofort ab**
  - Exception wird an Aufrufer der Methode weitergegeben
  - Wenn die Exception nicht spätestens in der main-Methode gefangen wird: Abbruch des Programms
  - Jede Methode muß deklarieren, welche Exceptions sie werfen kann (ausgenommen Error und RuntimeException):
    - `void anmelden(...) throws AnmeldungsException`

    oder
    - `public static void main(...) throws AnmeldungsException, OtherException`

Behandlung von Exceptions ...

- try - catch - Block:

```
try {
  int z = zahlen[index];
  int kehrwert = 1 / z;
  ...
}
catch (IndexOutOfBoundsException e) {
  System.out.println("Unzulässiger Index");
}
catch (ArithmeticException e) {
  System.out.println("Fehler: " + e.getMessage());
}
catch (Exception e) {
  System.out.println(e);
}
```

Behandlung von Exceptions ...

- Wenn Exception im try-Block auftritt:
  - try-Block wird verlassen
  - **Erster**”passender“ catch-Block wird ausgeführt, Ausführung wird nach dem letzten catch-Block fortgesetzt
  - falls kein passender catch-Block vorhanden: Abbruch der Methode, Weitergeben der Exception an Aufrufer
- Wann ist der catch-Block
  - ”passend“? wenn das erzeugte Exception-Objekt an den Parameter des catch-Blocks zugewiesen werden kann
  - d.h. die erzeuge Exception ist identisch mit der spezifizierten Exception-Klasse oder ist eine Unterklasse davon

4.12 Exceptions …

finally-Block

- Nach dem letzten catch-Block kann noch ein finally-Block angefügt werden
  - auch try - finally (ohne catch) ist erlaubt
- Die Anweisungen dieses Blocks werden **immer** nach Verlassen des try-Blocks ausgeführt, egal ob
  - der try-Block normal beendet wird
  - der Block (und die Methode) durch return verlassen wird
  - eine Exception auftritt und durch catch gefangen wird
    - hier: finally nach catch ausgef ührt
  - eine Exception auftritt und an den Aufrufer weitergegeben wird
- Anwendung:”Aufräumarbeiten“
  - z.B. Löschen temporärer Dateien, Schließen von Fenstern, ...

4.12 Exceptions …

„Werfen“ von Exceptions

- Exceptions können im Programm explizit durch die Anweisung `throw` ausgelöst werden:

```
public static double invertiere(double x)
{
  if (x == 0.0) {
    throw new ArithmeticException("Division durch 0");
  }
  return 1.0 / x;
}
```

- `throw` kann auch in einem `catch`-Block verwendet werden:

```
catch (Exception e) {
  ...          // Lokale Fehlerbehandlung ...
  throw e;  // Exception an Aufrufer weitergeben
}
```

4.12 Exceptions …

Definition eigener Exceptions

```
public class Vektor
{
  double[] vektor;
  ...
  public void invertiere() throws MyOwnException
  {
    for (int i = 0; i < vektor.length; i++) {
      if (vektor[i] == 0.0) {
        // Erzeuge und wirf Exception
        // Zusatzinfo: Index des Vektorelements
        throw new MyOwnException("Division durch 0", i);
      }
      vektor[i] = 1.0 / vektor[i];
    }
  }
}
```

4.12 Exceptions …

Definition eigener Exceptions ...

```
public class MyOwnException extends Exception {
{
  private int elementNr = -1;
  // Die nächsten beiden Konstruktoren sollten für jede
  // Exception definiert werden!
  public MyOwnException() {}
  public MyOwnException(String s) { super(s); }
  // Konstruktor mit Zusatzinformation zum Fehler
  public MyOwnException(String s, int elNr)
  {
    super(s);
    elementNr = elNr;
  }
```

4.12 Exceptions …

Definition eigener Exceptions ...

```
    // Auslesen der Zusatzinformation
    public int getElementNr() {
        return elementNr;
    }
    // Zum Ausdrucken ...
    public String toString()
    {
      return "Eigener Fehler im Element " + elementNr
        + ": " + getMessage();
      // getMessage ist Methode der Oberklasse Throwable,
      // liefert den String zurück, der dem Konstruktor
      // übergeben wurde
    }
}
```