

Vorlesung 1

Programmierung I

Java: Sprachkonzepte

Dozent: Prof. Dr. Peter Kelb
Raumnummer: Z4030
e-mail: peter.kelb@gmx.de
Sprechzeiten: nach Vereinbarung
Sourcen: <http://www1.hs-bremerhaven.de/kelb/>
Passwort: RDNPVUCJQQ

Organisatorisches

- Vorlesung (4 Stunden pro Woche)
- Übungen / Programmierlabor (2 (+ 2)??? Stunden pro Woche)
- zu Hause (> 8 Stunden pro Woche)
 - die Übungsaufgaben bearbeiten Sie zu Ende
 - Probleme versuchen Sie anhand der Vorlesung zu klären
 - offene Probleme klären Sie dann in den Übungen mit den Tutoren
- Semesterende: 2 Stunden Klausur
 - benotet
 - geht mit 7 ($=10 \cdot 0,7$) CPs (Creditpoints) in die Bachelornote ein

Wie lerne ich „Programmieren“?

- alleine !!!
- mit viel Zeit (10 CPs = 1/3 des kompletten ersten Semesters)
- mit einem Buch
 - erste Seite aufschlagen, durchlesen
 - erstes Programm **abtippen** (nicht runterladen, nicht von CD kopieren, **abtippen !!!**)
 - kompilieren und ausführen
 - erstes Programm verändern, kompilieren und ausführen
 - (Dauer: ca. einen ganzen Tag !!!)

Wie lerne ich nicht „Programmieren“?

- ich mache im Semester nichts für das Fach
- ich setze mich vier Wochen vor der Klausur hin und arbeite alte Klausuren durch
- ich bilde eine Arbeitsgruppe
- ich schaue jemanden, der es kann, beim Programmieren zu
- ich höre jetzt gerade nicht zu, weil ich mich auch durch die Schule schon durchgewurstelt habe

Aber, Sie sind nicht
mehr in der Schule

Was brauche ich zum Programmieren?

Buch

- irgendein nicht zu altes Java Buch (ab Java Version 1.5) ist ok
- nicht unbedingt exotische Bücher nehmen („Mit Java programmieren lernen für Dummies“)
- Online Buch:
Java-Programmierung - Das Handbuch zu Java 8, Guido Krüger und Heiko Hansen, O'Reilly Verlag, ISBN-13: 978-3955615147,
<http://www.javabuch.de/>

Tools:

- <http://www.oracle.com/technetwork/java/index.html> (Java SE)
- einfache IDE: <http://www.javaeditor.org/>, <http://drjava.sourceforge.net/>,
<https://bluej.org/>
- komplexe IDE: <http://www.eclipse.org/> (Eclipse),
<https://www.jetbrains.com/idea/> (IntelliJ), <https://netbeans.org/> (NetBeans)

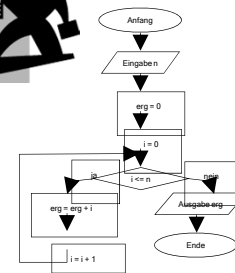
Los geht's ...

Am Anfang steht immer die Aufgabe: „Programmieren Sie ...“

1. Zunächst entwickeln Sie ein Idee



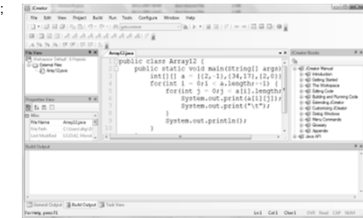
2. Aus der Idee wird ein *Algorithmus*



3. Aus dem Algorithmus wird ein Programm

```
public class Array12 {  
    public static void main(String[] args) {  
        int[] a = {{2,-1},{34,17},{2,0}};  
        for(int i = 0; i < a.length; ++i) {  
            for(int j = 0; j < a[i].length; ++j) {  
                System.out.print(a[i][j]);  
                System.out.print("\t");  
            }  
            System.out.println();  
        }  
    }  
}
```

4. Das Programm wird in den Rechner eingegeben



5. Ein *Compiler* übersetzt das Programm auf dem Rechner für den Rechner in ein ausführbares Programm

6. Das ausführbare Programm wird auf dem Rechner getestet

Was ist ein Algorithmus?

Ein Algorithmus ist eine

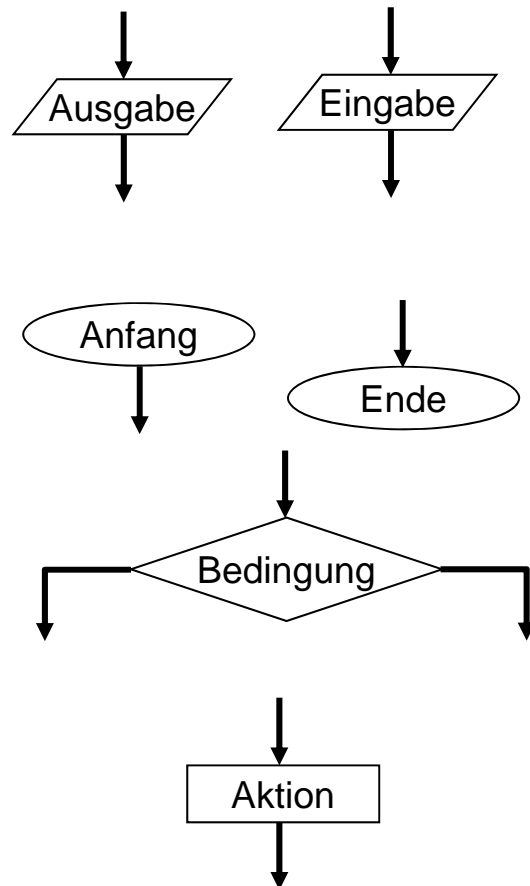
- Arbeitsanleitung zum Lösen eines Problems/Aufgabe,
- die so präzise formuliert ist, dass sie von einem Computer ausgeführt werden kann

Charakteristika:

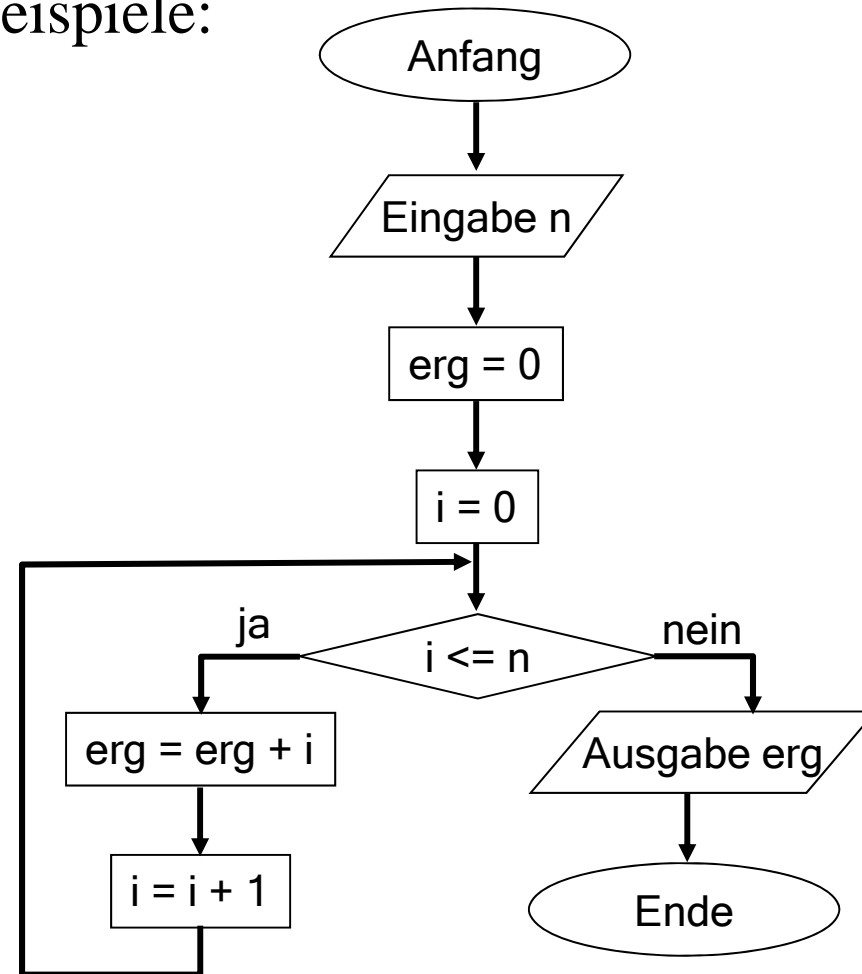
- Anweisungssequenzen
- bedingte Anweisungen
- Anweisungsschleifen
- Zutaten / Voraussetzungen

Programmablaufpläne / Flußdiagramme

Elemente:

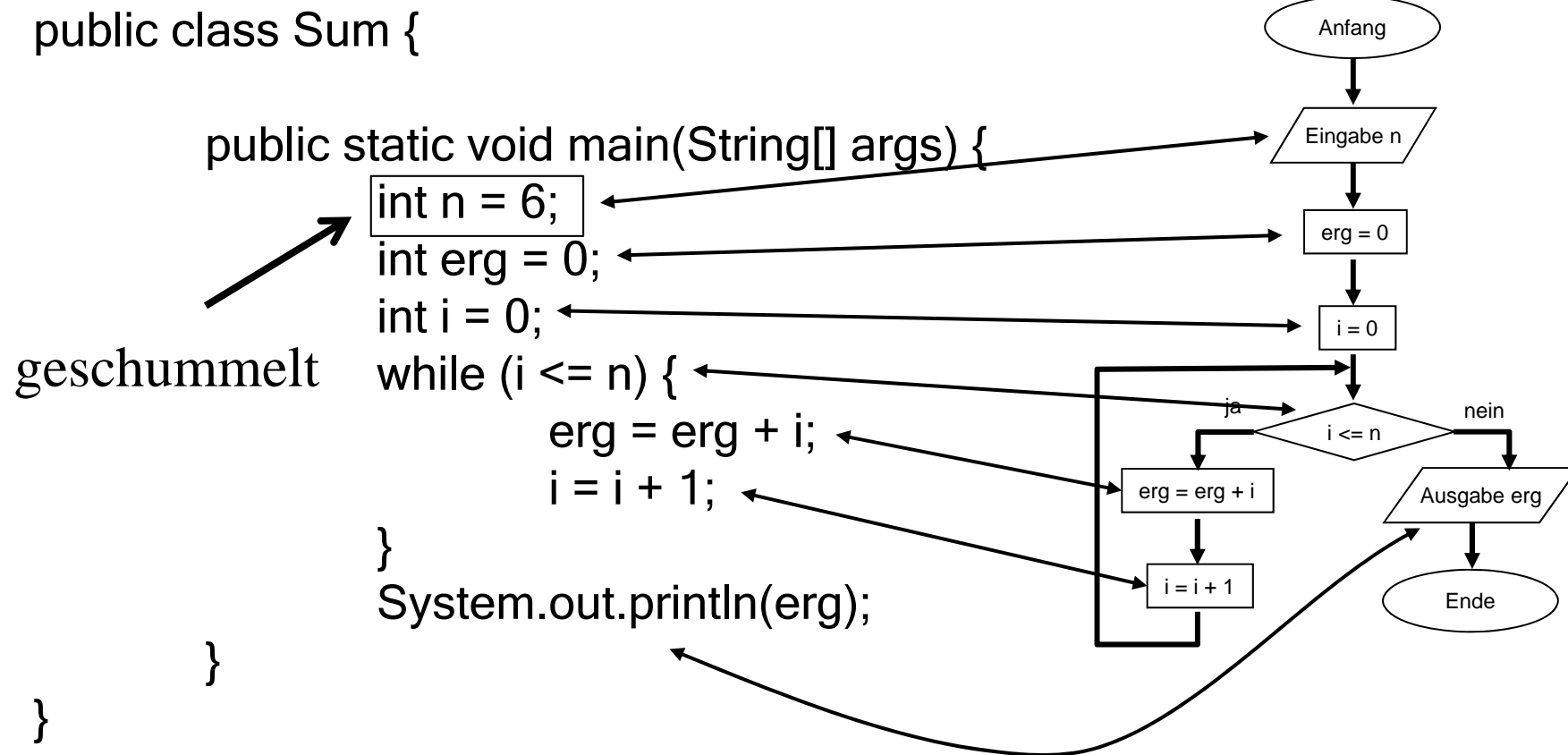


Beispiele:



Go!

Vom Flussdiagramm zum Programm



Syntax

Problem:

Wie kann die Syntax einer Programmiersprache beschrieben werden?

Lösung:

1. umgangssprachlich
2. mathematisch: mit einer eigenen Sprache, die mathematisch exakt definiert ist

Syntax (Fort.)

1. umgangssprachlich: sehr umfangreiche Beschreibung für moderne Programmiersprache; i.d.R. nicht exakt \Rightarrow

- der Benutzer weiß nicht, was genau er/sie hinschreiben darf
- der Compilerbauer weiß nicht, was der Compiler alles akzeptieren und übersetzen muss

2. mathematisch: exakte Definition der Syntax; der Compilerbauer weiß den exakten Sprachumfang; der Benutzer kann im Zweifelsfall nachschauen; i.A. für den Benutzer zu kompliziert

Fazit: beide Beschreibungen benutzen

Syntax (Fort.)

Definition von Tokens: mittels regulärer Ausdrücke (siehe Theorie)

Definition von Sätzen: mittels der Extended Backus Naur Form (EBNF) (siehe auch Theorie)

EBNF: Regelsatz, der erklärt, wie korrekte Sätze einer Programmiersprache gebildet werden. Besteht aus:

1. Terminalsymbole: Zeichen und Wörter, die der Programmierer verwendet
2. Nichtterminalsymbole: Meta-Zeichen, die die EBNF braucht, um die Regeln zu definieren. Sie werden durch Regeln beschrieben.

EBNF

Beispiel:

Hans ::= "a" "b" ("c" | "d")

- eine Regel namens „Hans“
- die Regel beschreibt, wie Namen gebildet werden
- die Zeichen, die in "" stehen, können beim Programmieren verwendet werden
- das Wort wird von links nach rechts gebildet
- (,) und | sind Metazeichen für die Regelbildung
- | bedeutet: rechte *oder* linke Seite
- in diesem Beispiel: 2 Wörter sind möglich: "abc" oder "abd"

EBNF (Fort.)

- in einer Regel können auch wieder die Regelsymbole vorkommen:

Otto ::= "a" Otto "b" | "c"

- kommt in einer Regel wieder ein Regelsymbole vor, so muss man an dieser Stelle die entsprechend zugehörige Regel wieder anwenden
- mit Hilfe von Metasymbolen werden die Regeln komplexer und mächtiger in ihrer Ausdrucksform

EBNF (Fort.)

- es gibt die folgenden Metasymbole:
() { } [] |
- die Klammern ‚(‘ und ‚)‘ werden für die Eindeutigkeit verwendet (wie in der Mathematik)
- ‚{‘ und ‚}‘ beschreiben die Möglichkeit zu beliebig vielen Wiederholungen (oder auch der Auslassung)

Bsp.: $A ::= \text{"a" "b" \{ "c" "d" \}}$
erzeugt: "ab", "abcd", "abcdcd", "abcdcdcd", ...

EBNF (Fort.)

- ,[' und ,]' beschreibt die Möglichkeit zu einer möglichen Auslassung

Bsp.: A ::= "a" "b" ["c" "d"]
erzeugt: "ab", "abcd"

- ,|' beschreibt die Wahl zwischen 2 Möglichkeiten

Bsp.: A ::= "a" "b" | "c" "d"
erzeugt: "ab", "cd"

EBNF (Fort.)

- zusammen mit den Prioritätenklammerung ergeben sich viele Möglichkeiten:

Bsp.: $A ::= "a" ("b" \mid "c") "d"$
erzeugt: "abd", "acd"


- Regelsymbole in einer Regel werden durch Substitution aufgelöst:

Bsp.: $A ::= "a" B "d"$
 $B ::= \{ "x" \}$
erzeugt: "ad", "axd", "axxd", "axxxd", ...

EBNF (Fort.)

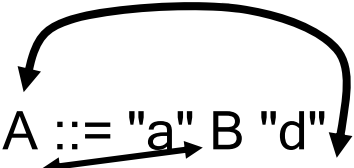
- eine Regel kann sein eigenes Regelsymbol enthalten (Rekursion):

Bsp.: $A ::= "a" A "d" \mid "x"$
erzeugt: "x", "axd", "aaxdd", "aaaxddd", ...



- es kann sein eigenes Symbol direkt (wie oben) oder indirekt enthalten:

Bsp.: $A ::= "a" B "d"$
 $B ::= "x" \mid "e" A "f"$
erzeugt: "axd", "aeaxdfd", "aeaeaxdfdfd", ...



EBNF (Fort.)

- Vorsicht vor "komischen" Regeln:

Bsp.: $A ::= "a" A "d"$
erzeugt: ????

- für das restliche Informatikerleben:

Rekursion ist ein *sehr* mächtiges Instrument. Es birgt aber die Gefahr von *Unendlichkeit* (unendliche Berechnung).

Syntaxdiagramme

- EBNF-Regeln kann man auch graphisch darstellen durch sogenannte *Syntaxdiagramme*
- Terminalsymbole werden durch Ovale dargestellt:

 statt "a"

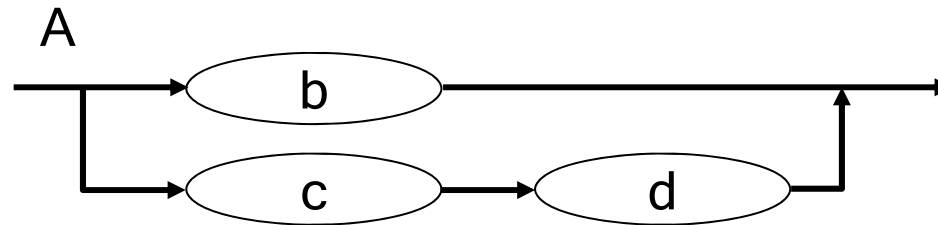
- Nichtterminalsymbole werden durch Rechtecke dargestellt:

 statt A

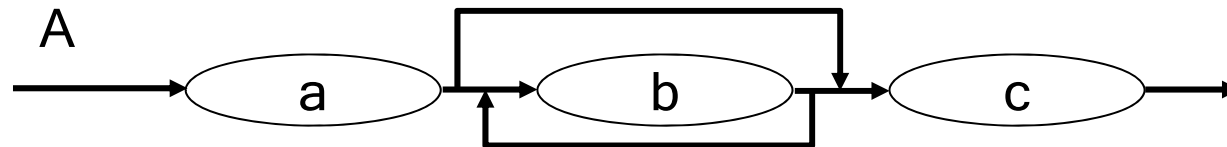
- Metasybole werden durch "Schleifen" von Pfeilen dargestellt:

Syntaxdiagramme (Fort.)

- Auswahl: $A ::= "b" \mid "c" "d"$

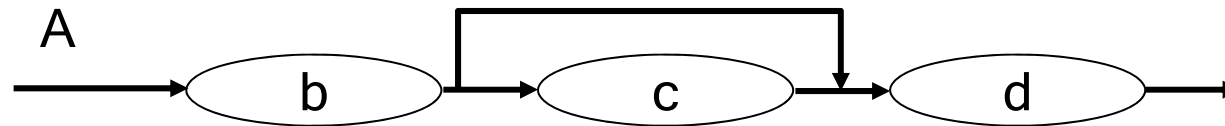


- Wiederholung: $A ::= "a" \{ "b" \} "c"$



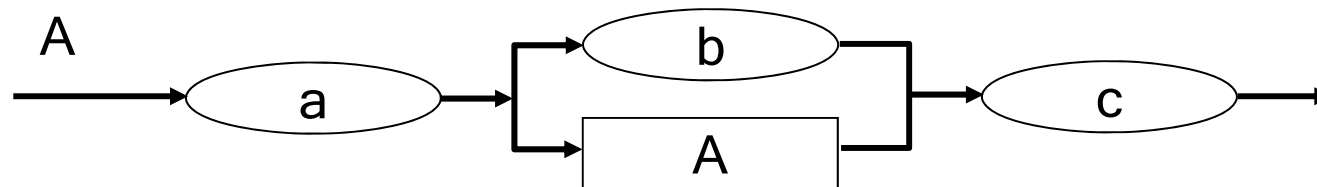
Syntaxdiagramme (Fort.)

- Auslassung: $A ::= \text{"b"} [\text{"c"}] \text{"d"}$



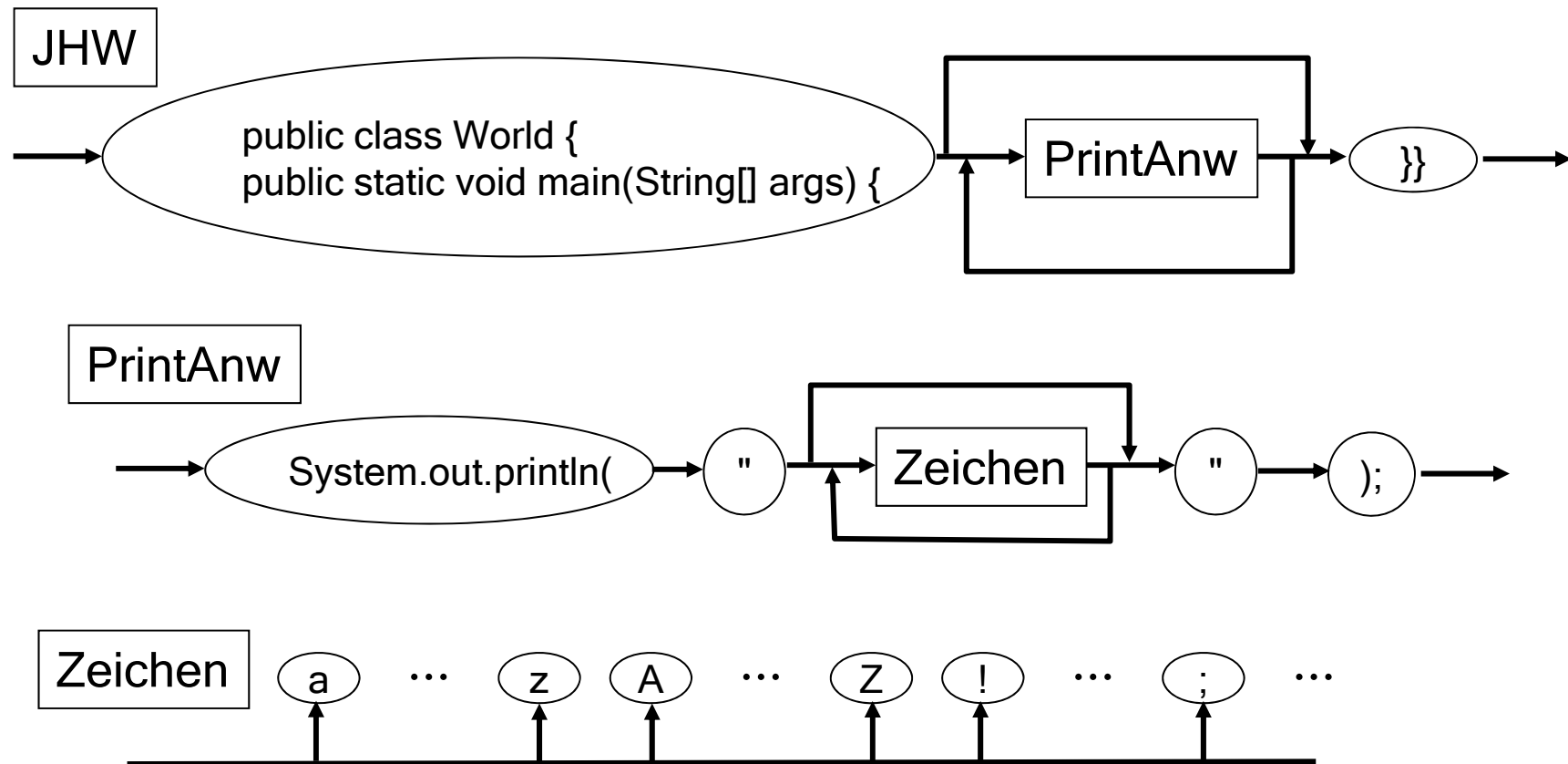
- einfache Klammerung: durch Unterdiagramme

$A ::= \text{"a"} (\text{"b"} \mid A) \text{"c"}$



Beispiel

- Syntaxdiagramm für einfache "Java-Hello-World"-Programme:



Beispiel (Fort.)

- EBNF für einfache "Java-Hello-World"-Programme:

JHW ::= "public class World {
 public static void main(String[] args) {"
 { PrintAnw } "}"

PrintAnw ::= "System.out.println(" " " { Zeichen } " " " ");"

Zeichen ::= "a" | "b" | ... | "z" | "A" | "B" | ... | "Z" | "!" | ... | ";" | ...

Terminal-
symbole

- Manche Regeln lassen sich besser durch Syntaxdiagramme darstellen, manche besser durch EBNF
- Problem macht das "-Zeichen; dient als Metasymbol und als Terminalsymbol

Beispiel (Fort.)

Beispiel für einfache "Java-Hello-World"-Programme:

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

```
public class World {  
    public static void main(String[] args) {  
        System.out.println("Hello World");  
        System.out.println("Dies ist die 2. Println-Anweisung");  
    }  
}
```

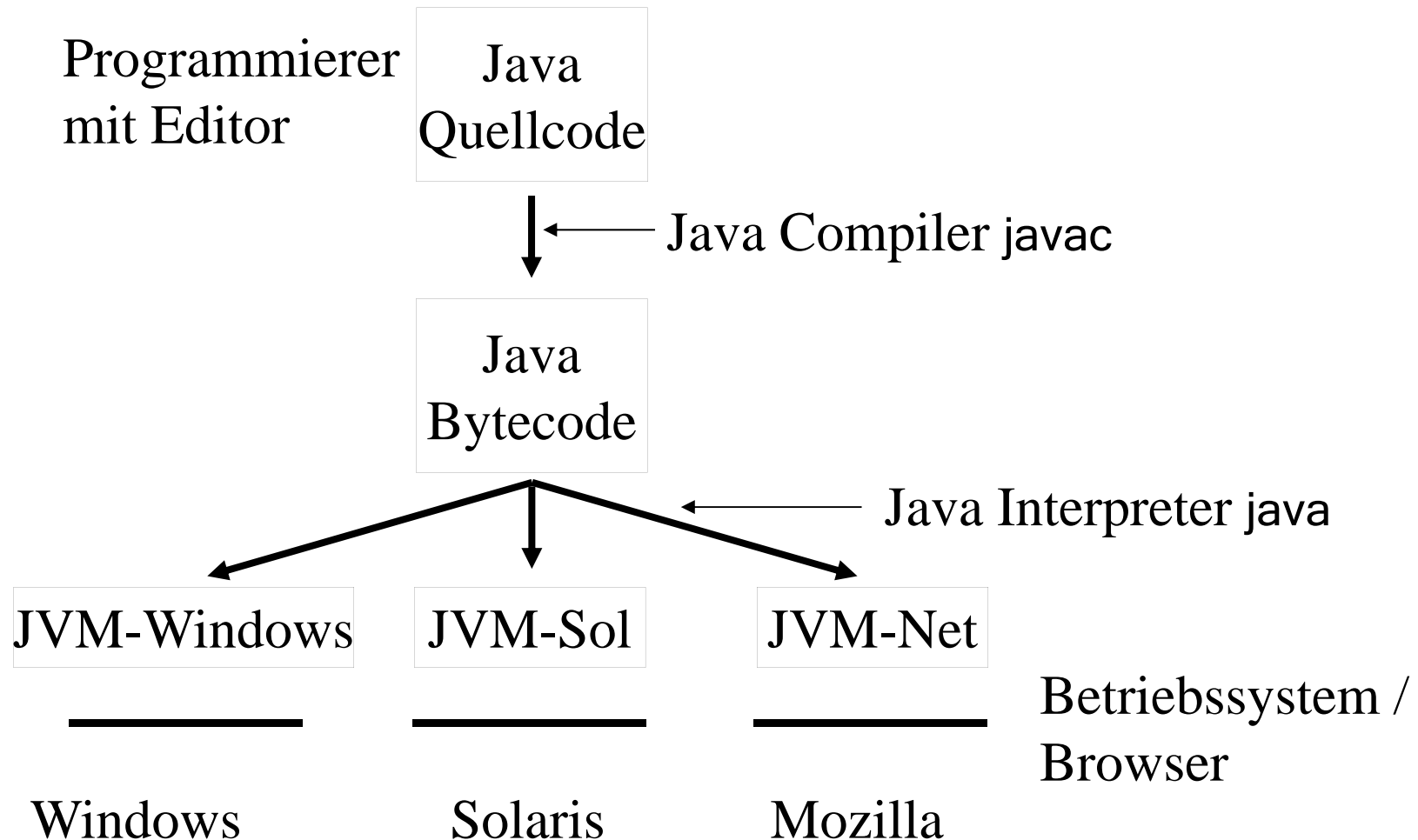
```
public class World {  
    public static void main(String[] args) {  
    }  
}
```

das
einfachste
Java-Hello-
World
Programm

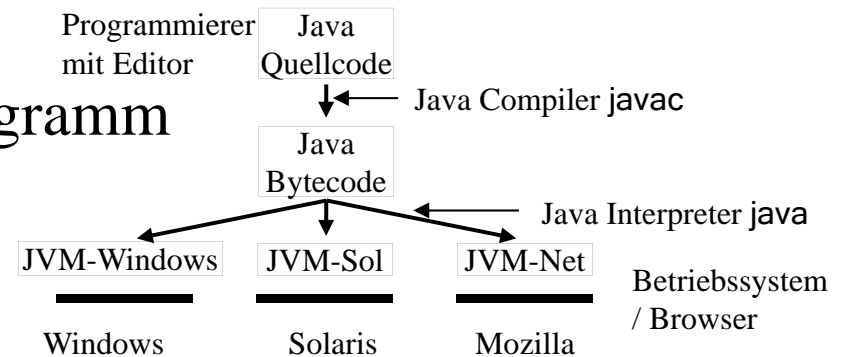


Vorlesung 2/1

Arbeitsweise von Java



Ein erstes Programm



```
public class World {  
    public static void main(String[] args) {  
        System.out.println("Hello world!");  
    }  
}
```

Java Quellcode:
abspeichern in
World.java

`javac World.java`

kompilieren: Ergebnis: die
Datei World.class

`java World`

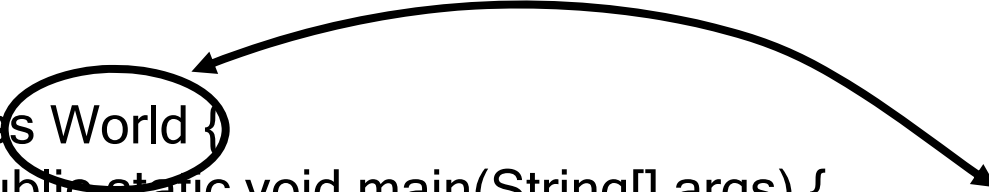
Ausführen des Programms

Ein erstes Programm (Fort.)

Wichtig:

```
public class World {  
    public static void main(String[] args) {  
        System.out.println("Hello world!");  
    }  
}
```

Java Quellcode:
abspeichern in
World.java



`javac World.java`

kompilieren: Ergebnis: die
Datei **World.class**

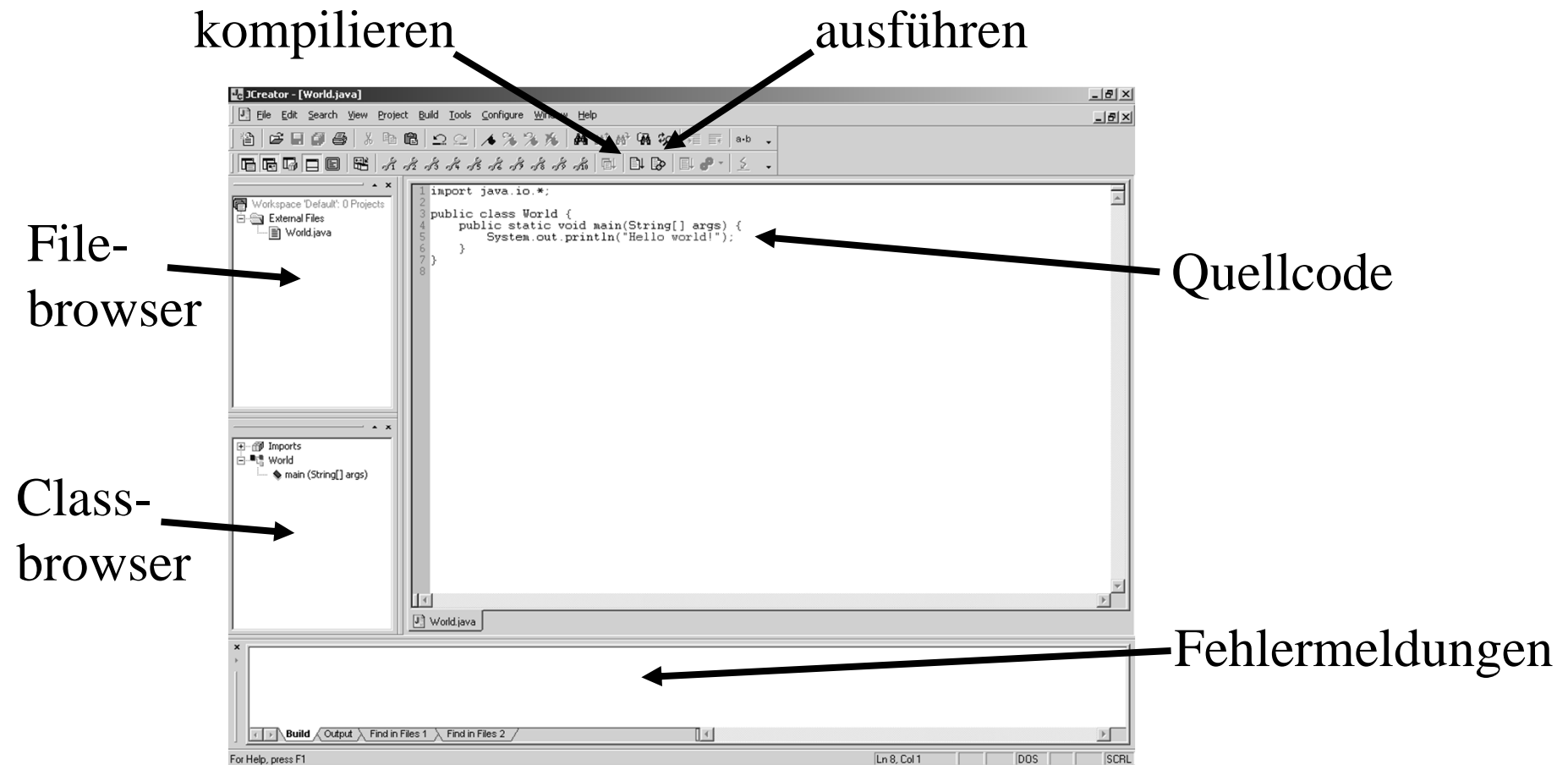
`java World`

Ausführen des Programms

Go!

Ein erstes Programm (Fort.)

In der Praxis: Arbeiten mit Entwicklungsumgebungen



Aussagenlogik (Boolesche Logik)

Dient dazu, einfache Aussagen, die **wahr** oder **falsch** sind, miteinander zu verknüpfen, um komplexere Aussagen zu erhalten, die aber wiederum **wahr** oder **falsch** sind.

wichtige Begriffe:

- einfache (atomare) Aussagen = boolesche Aussagen
- können **wahr** oder **falsch** sein = **true** oder **false** = T oder F
- werden miteinander verknüpft, das Ergebnis sind boolesche Ausdrücke

Beispiel für einfache Aussagen:

- es regnet draußen
- heute ist Sonntag
- Rot ist eine Farbe

Aussagenlogik (Fort.)

einfache Aussagen können durch boolesche Operatoren zu komplexen booleschen Ausdrücken verknüpft werden:

- Negation: \neg es regnet heute bedeutet: es regnet heute *nicht*
- Konjunktion: \wedge es regnet heute && heute ist Sonntag bedeutet: es regnet heute *und* es ist Sonntag
- Disjunktion: \vee es regnet heute || die Sonne scheint heute bedeutet: heute regnet es *oder* es scheint die Sonne
- Implikation: \Rightarrow es regnet heute \Rightarrow die Straße ist nass bedeutet: *wenn* es heute regnet *dann* ist die Straße nass
- Äquivalenz: \Leftrightarrow es regnet heute \Leftrightarrow die Straße ist nass bedeutet: genau dann wenn es heute regnet ist die Straße nass

Aussagenlogik (Fort.)

Definition der booleschen Operatoren mittels Wahrheitstabellen.
Im folgenden sind **P** und **Q** boolesche Aussagen.

Negation

P	!P
T	F
F	T

Konjunktion

P	Q	P && Q
T	T	T
T	F	F
F	T	F
F	F	F

Disjunktion

P	Q	P Q
T	T	T
T	F	T
F	T	T
F	F	F

Implikation

P	Q	P \Rightarrow Q
T	T	T
T	F	F
F	T	T
F	F	T

Äquivalenz

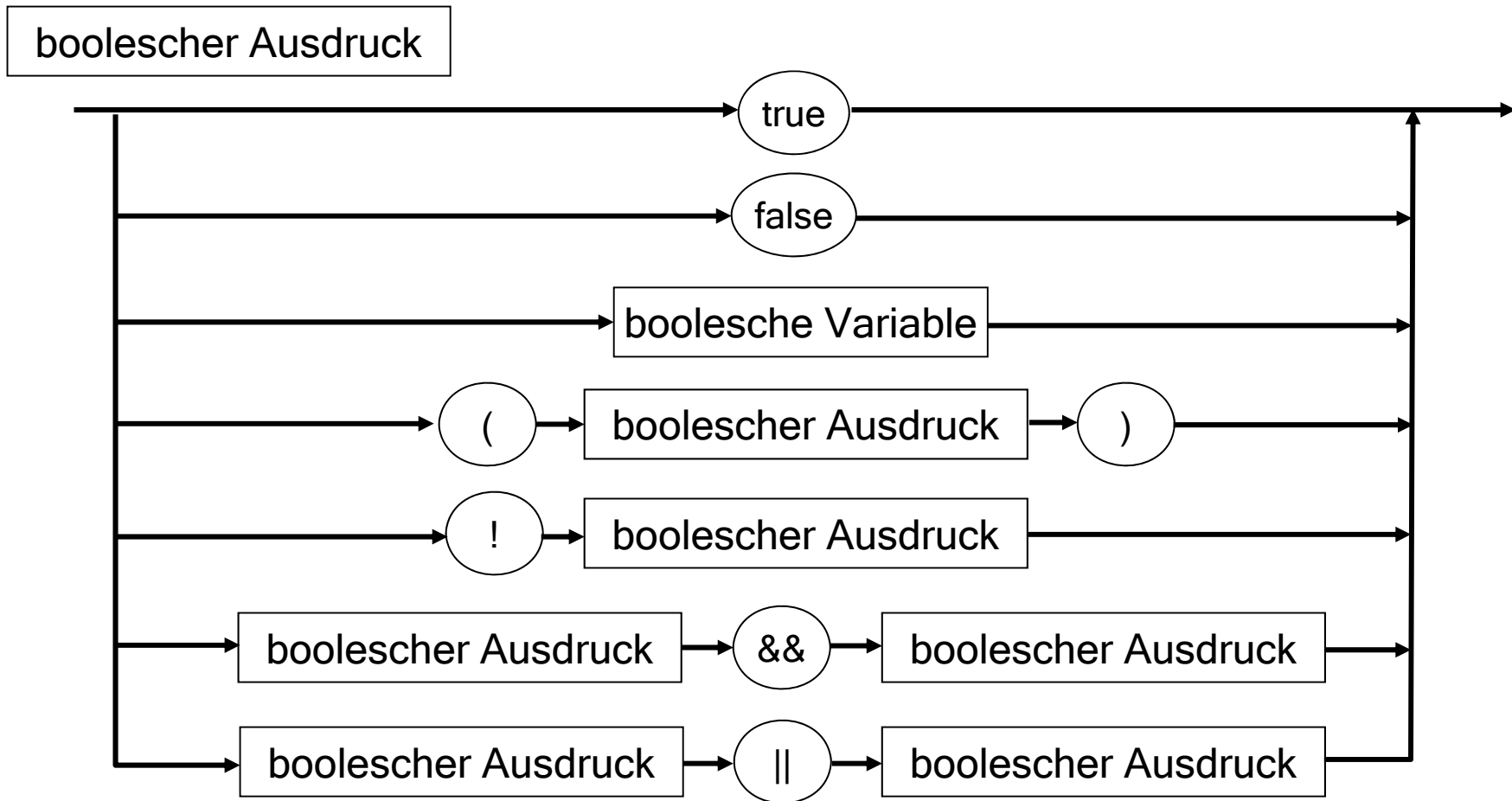
P	Q	P \Leftrightarrow Q
T	T	T
T	F	F
F	T	F
F	F	T

Aussagenlogik in Java

- in Java gibt es ebenfalls Aussagenlogik
- die beiden booleschen konstanten Wahrheitswerte heißen
 - **true**
 - **false**
- Variablen, die boolesche Werte enthalten (**true** oder **false**), müssen vom Typ **boolean** sein (später mehr zu Typen)
- die Negation wird mittels des Operators **!** gebildet
- die Konjunktion lautet **&&**
- die Disjunktion lautet **||**
- es gibt keine Implikation (ist ja auch nicht notwendig, weil $a \Rightarrow b$ das gleiche ist wie $!a \ || \ b$)
- es gibt keine Äquivalenz (dito.)

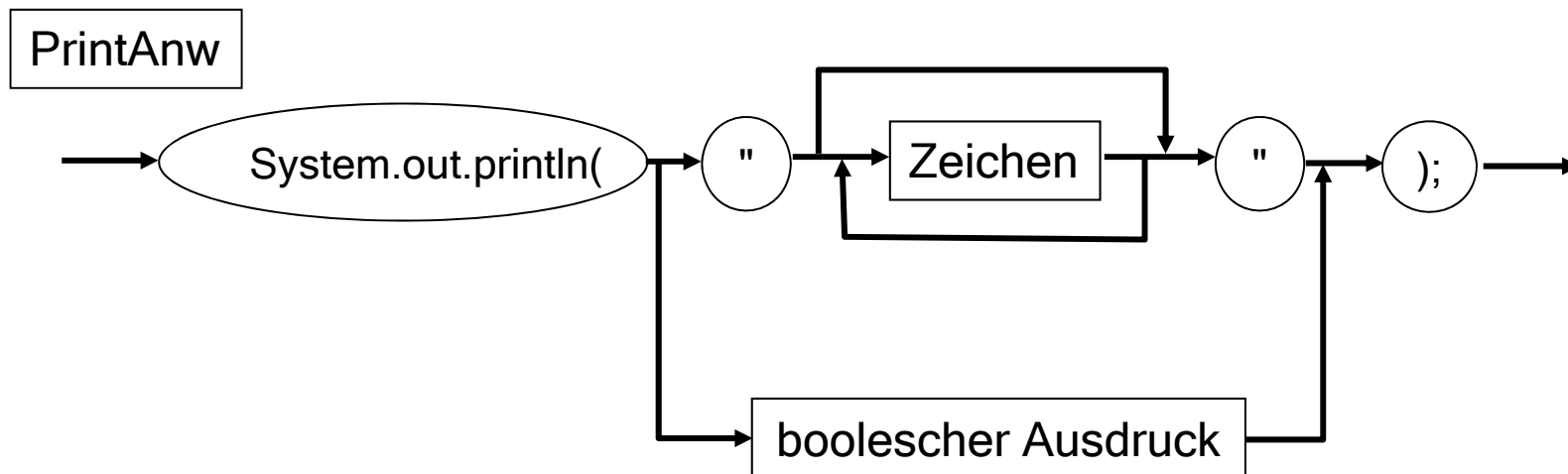
Aussagenlogik in Java (Fort.)

EBNF für boolesche Ausdrücke in Java



Aussagenlogik in Java (Fort.)

- boolesche Ausdrücke können in Java mittels der print-Anweisung ausgegeben werden
- statt einer Zeichenkette wird einfach ein boolescher Ausdruck der println-Funktion (Methode) übergeben
- wichtig: die Anführungsstriche *dürfen nicht* vorne und hinten stehen



Go!

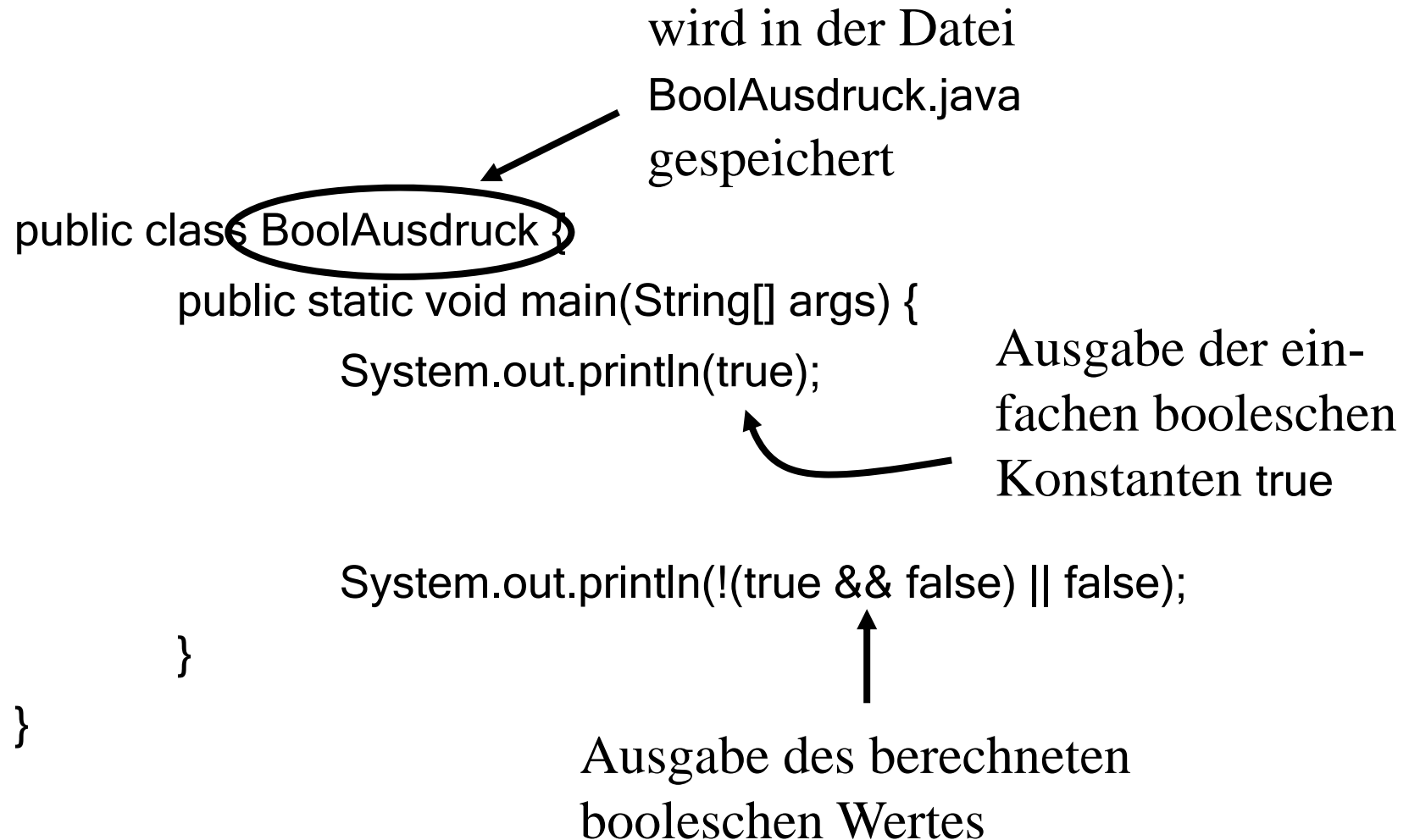
Aussagenlogik in Java: Beispiel

```
public class BoolAusdruck {  
    public static void main(String[] args) {  
        System.out.println(true);  
  
        System.out.println(!(true && false) || false);  
    }  
}
```

wird in der Datei
BoolAusdruck.java
gespeichert

Ausgabe der ein-
fachen booleschen
Konstanten true

Ausgabe des berechneten
booleschen Wertes



Variablen

Aufgabe: folgender boolescher Ausdruck soll in Java berechnet werden

$(\text{true} \ \&\& \ \text{false} \ || \ \text{!(true} \ \&\& \ \text{false)}) \Leftrightarrow ((\text{false} \ || \ \text{true}) \ \&\& \ \text{!false})$

da $P \Leftrightarrow Q$ das gleiche ist wie $P \Rightarrow Q \ \&\& \ Q \Rightarrow P$

und $P \Rightarrow Q$ das gleiche ist wie $\text{!}P \ || \ Q$,

ist $P \Leftrightarrow Q$ das gleiche wie $(\text{!}P \ || \ Q) \ \&\& \ (\text{!}Q \ || \ P)$

somit gilt für den obigen Ausdruck:

$(\text{!(true} \ \&\& \ \text{false} \ || \ \text{!(true} \ \&\& \ \text{false)}) \ || \ ((\text{false} \ || \ \text{true}) \ \&\& \ \text{!false})) \ \&\& \$
 $(\text{!((false} \ || \ \text{true}) \ \&\& \ \text{!false)} \ || \ (\text{true} \ \&\& \ \text{false} \ || \ \text{!(true} \ \&\& \ \text{false)}))$

Frage: wer sieht sofort das Ergebnis?

Variablen (Fort.)

Beobachtung: es kommen immer wieder dieselben Teilausdrücke in dem Gesamtausdruck vor

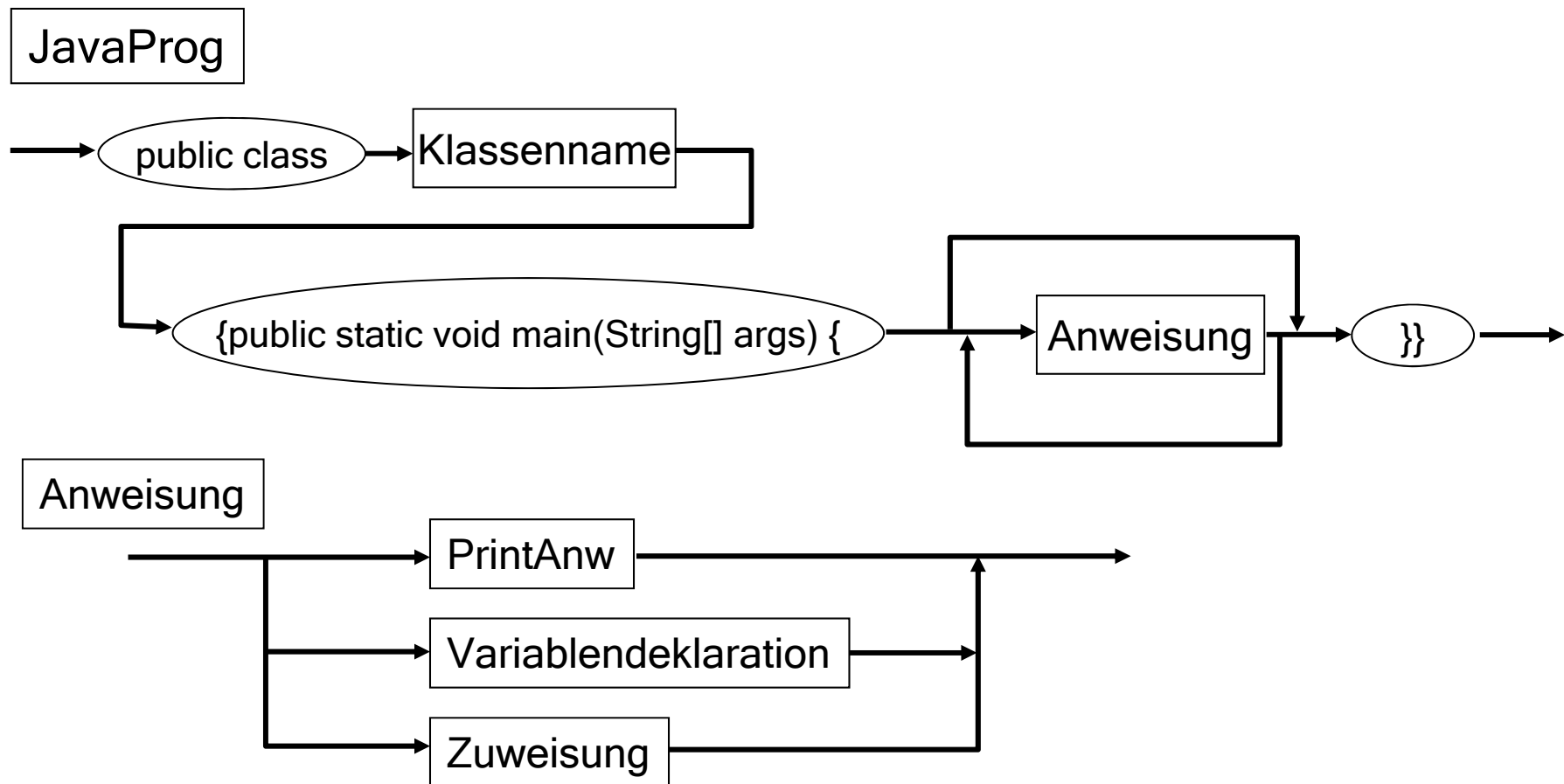
Idee: diese Teilausdrücke nur jeweils einmal berechnen und das Ergebnis im Speicher ablegen, um beim erneuten Bedarf direkt auf das Ergebnis zugreifen zu können

Lösung: eine Variable anlegen, die einen booleschen Wert speichern kann

Java: `boolean p;`
legt eine boolesche Variable mit dem Namen `p` an;
unter `p` kann ein boolescher Wert gespeichert werden

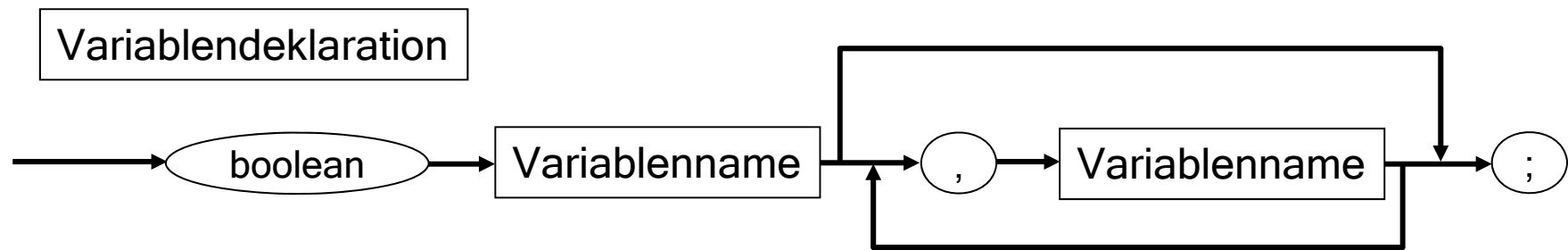
Variablen (Fort.)

Die Vereinbarung von Variablen (**Variablendeklaration**) ist eine Anweisung



Variablendeklaration

In einer Variablendeklaration können mehrere Variablen gleichzeitig deklariert werden



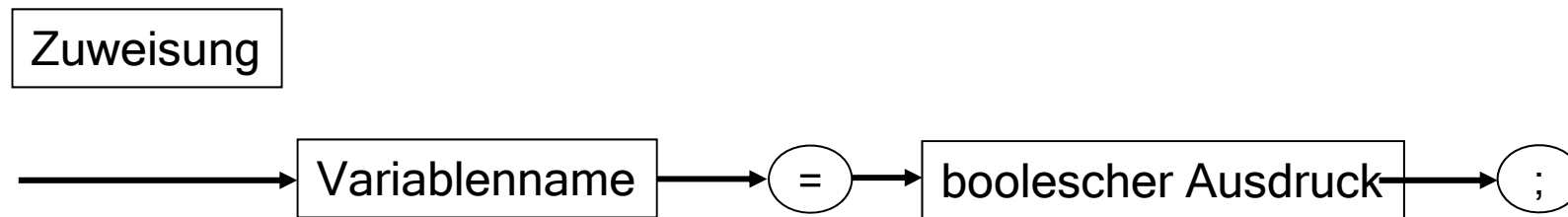
Beispiel:

```
boolean a;           // deklariert die boolesche Variable a
boolean b,c,juhu;    // deklariert die booleschen
                     // Variablen b, c und juhu
```

Zuweisung

Variablen bekommen in Form von Zuweisungen einen Wert zugewiesen, den sie sich solange merken, bis sie

- einen neuen Wert zugewiesen bekommen haben
- die Variable aus dem Speicher "verschwindet"



Beispiel:

`a = true;` // speichert den Wert true unter a ab

`juhu = !true || false;` // speichert den Wert false unter
// juhu ab

Zuweisung (Fort.)

das ursprüngliche Problem war, den Ausdruck

`(true && false || !(true && false)) \Leftrightarrow ((false || true) && !false)`

in Java zu berechnen, dazu

- 2 boolesche Variablen anlegen, z.B. `p` und `q`
- unter `p` den Ausdruck `(true && false || !(true && false))` abspeichern
- unter `q` den Ausdruck `((false || true) && !false)` abspeichern
- `p \Leftrightarrow q` berechnen als `(!p || q) && (!q || p)`

Go!

Zuweisung (Fort.)

```
public class BoolAusdruck2 {
```

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

```
        boolean p,q; Deklaration zweier boolescher Variablen
```

noch eine
boolesche
Variable



```
        boolean res;
```

```
        p = (true && false || !(true && false));
```

```
        q = ((false || true) && !false);
```

die beiden Teil-
ausdrücke
werden berechnet

das Ergebnis
wird berechnet

```
        res = (!p || q) && (!q || p);
```

```
        System.out.println("Das Ergebnis lautet:");
```

```
        System.out.println(res);
```

das Ergebnis wird
ausgegeben

```
    }
```

```
}
```

Zuweisung (Fort.)

- Es *müssen nicht* erst alle Variablen deklariert werden
- Nach Zuweisungen und Print-Anweisungen können auch wieder Variablendeklarationen erfolgen

```
public class BoolAusdruck2 {  
    public static void main(String[] args) {  
        boolean p,q;   
        p = (true && false || !(true && false));  
        System.out.println("Das Ergebnis lautet:");  
        q = ((false || true) && !false);  
        boolean res;  
        res = (!p || q) && (!q || p);  
        System.out.println(res);  
    }  
}
```

Deklaration (points to `boolean p,q;` and `boolean res;`)

Zuweisungen (points to `p = ...`, `q = ...`, and `res = ...`)

Print-Anweisungen (points to `System.out.println(...)`)


Zuweisung (Fort.)

- Wichtig: eine Variable ***muss vor*** ihrer Benutzung deklariert werden, d.h.
- bevor ihr ein Wert zugewiesen (beschrieben) wird,
- bevor sie in einem Ausdruck verwendet (gelesen) wird.

```
public class BoolAusdruck3 {  
    public static void main(String[] args) {  
        boolean p,q;  
        p = (true && false || !(true && false));  
        q = ((false || true) && !false);  
        res = (!p || q) && (!q || p);  
        boolean res;  
        System.out.println("Das Ergebnis lautet:");  
        System.out.println(res);  
    }  
}
```

res wird hier
beschrieben
... →

... aber erst hier
deklariert ←



Go!

Zuweisung (Fort.)

```
res = (!p || q) && (!q || p);  
boolean res;
```

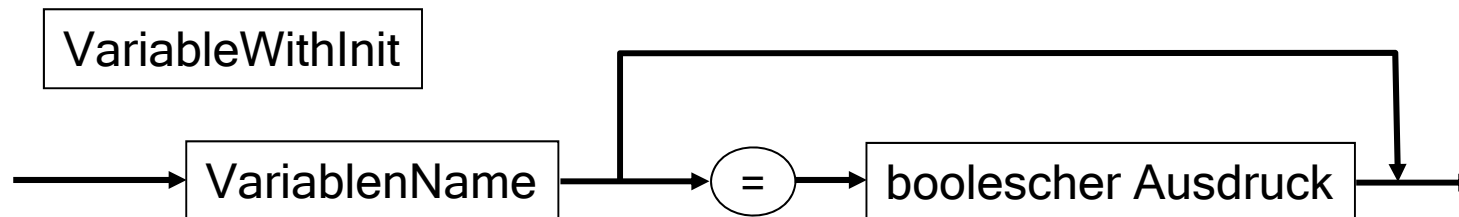
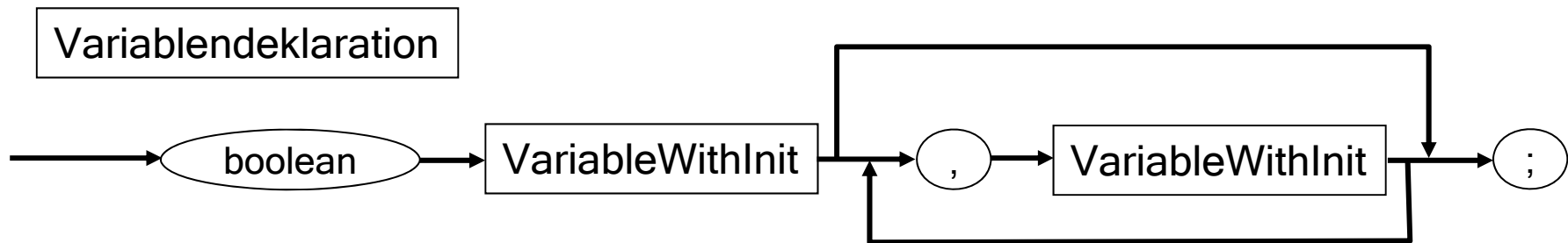
führt zu folgender Fehlermeldung:

```
...\BoolAusdruck3.java:6: cannot resolve symbol  
symbol  : variable res  
location: class BoolAusdruck3  
    res = (!p || q) && (!q || p);  
    ^
```

- das Programm ist syntaktisch korrekt
- das Programm hat einen semantischen Fehler
- dieser Fehler wird in der Phase des statischen Semantikchecks erkannt

Variablendeklaration mit Initialteil

- eine *Variablendeklaration* ist eine Anweisung
- eine *Variablenzuweisung* ist eine Anweisung
- beide Anweisungen kann man in eine Anweisung zusammenfassen



Go!

Beispiel: Variablendeklaration mit Initialteil

```
public class VarInit {  
    public static void main(String[] args) {  
        boolean p = true, q = false;  
        System.out.println("Die Variablen haben die Werte:");  
        {  
            System.out.println(p);  
            System.out.println(q);  
        }  
    }  
}
```

die booleschen Variablen **p**
und **q** werden deklariert
und gleichzeitig initialisiert

Guter Programmierstil: bei der Deklaration immer
gleich die Variablen initialisieren. *Warum?*

Vorlesung 2/2

Anweisungen

bisher:

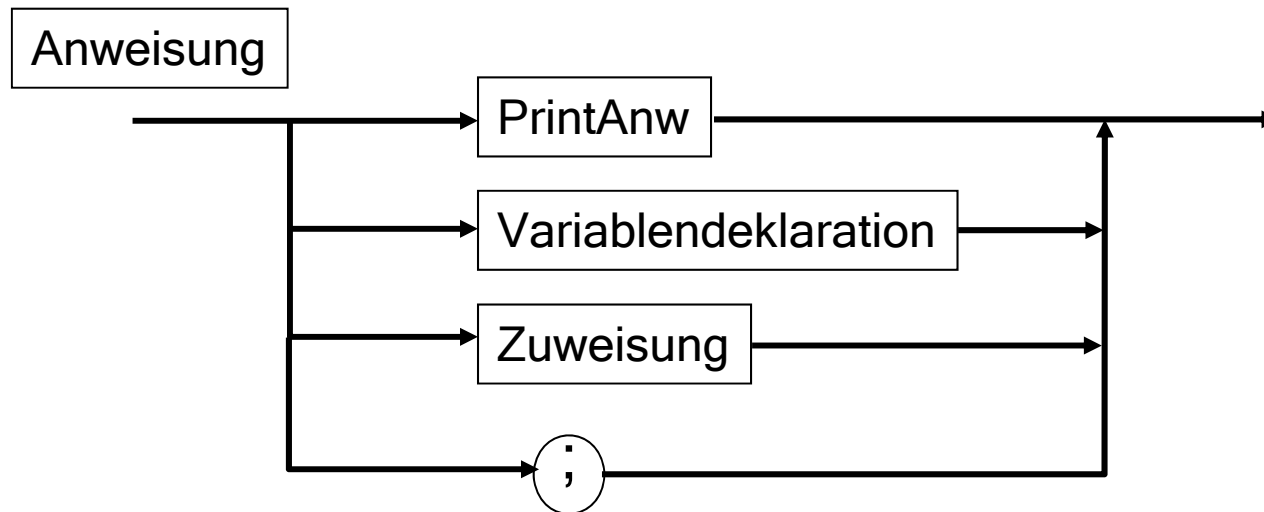
- print-Anweisungen
- Zuweisungen (boolescher Ausdrücke an boolesche Variablen)
- Variablendeklarationen

im folgenden:

- weitere Anweisungen, um den Programmfluss zu steuern

Elementare Anweisungen

- die einfachste Anweisung ist die, die gar nichts macht, die sogenannte *leere Anweisung*
- in Java wird sie durch ein einfaches `;` dargestellt



Die leere Anweisung

Beispiel für die leere Anweisung:

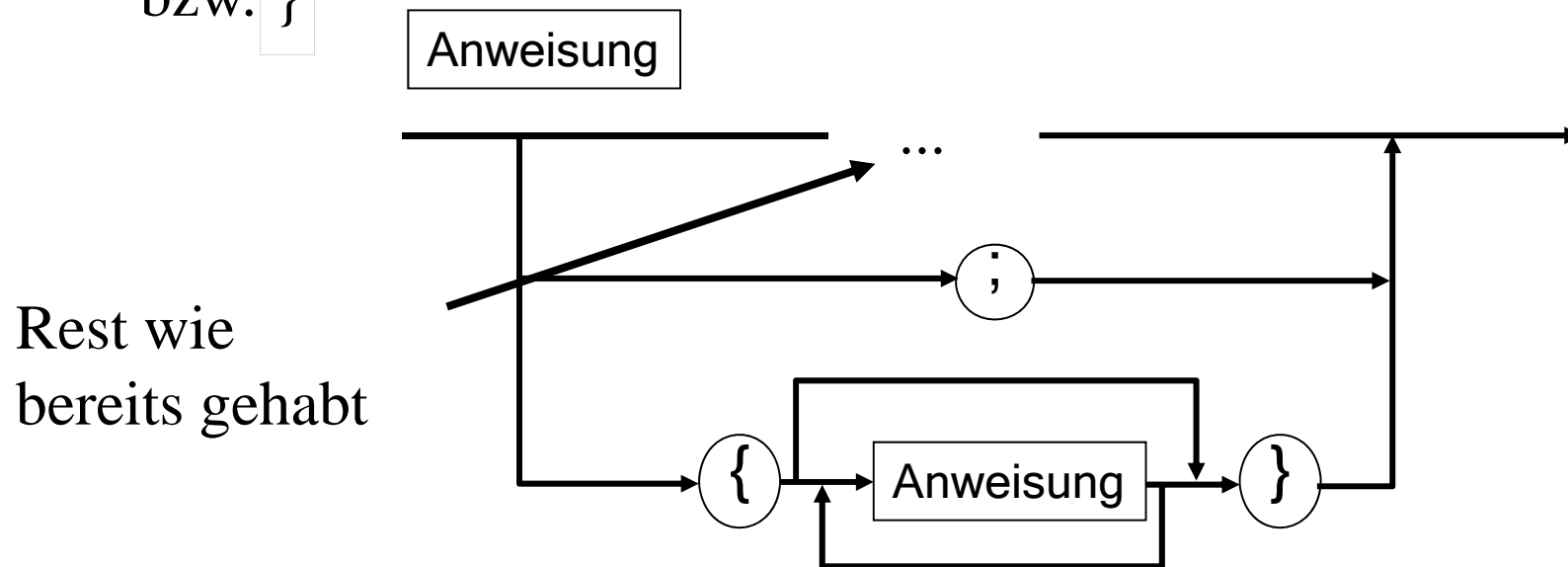
```
public class Nichts {  
    public static void main(String[] args) {  
        ;  
        System.out.println("Ich habe 4 Anweisungen");  
        ;  
        ;  
    }  
}
```

leere Anweisungen kommen in der
Praxis i.d.R. durch Tippfehler zustande

eine leere Anweisung ist ein leerer String (ein nichts)
gefolgt von einem Semikolon

Der Block

- man möchte mehrere Anweisungen zusammenfassen
- diese Zusammenfassung verhält sich dann wie eine Anweisung
- passiert in der Praxis sehr oft
- die Zusammenfassung erfolgt die Klammerung von { bzw. }



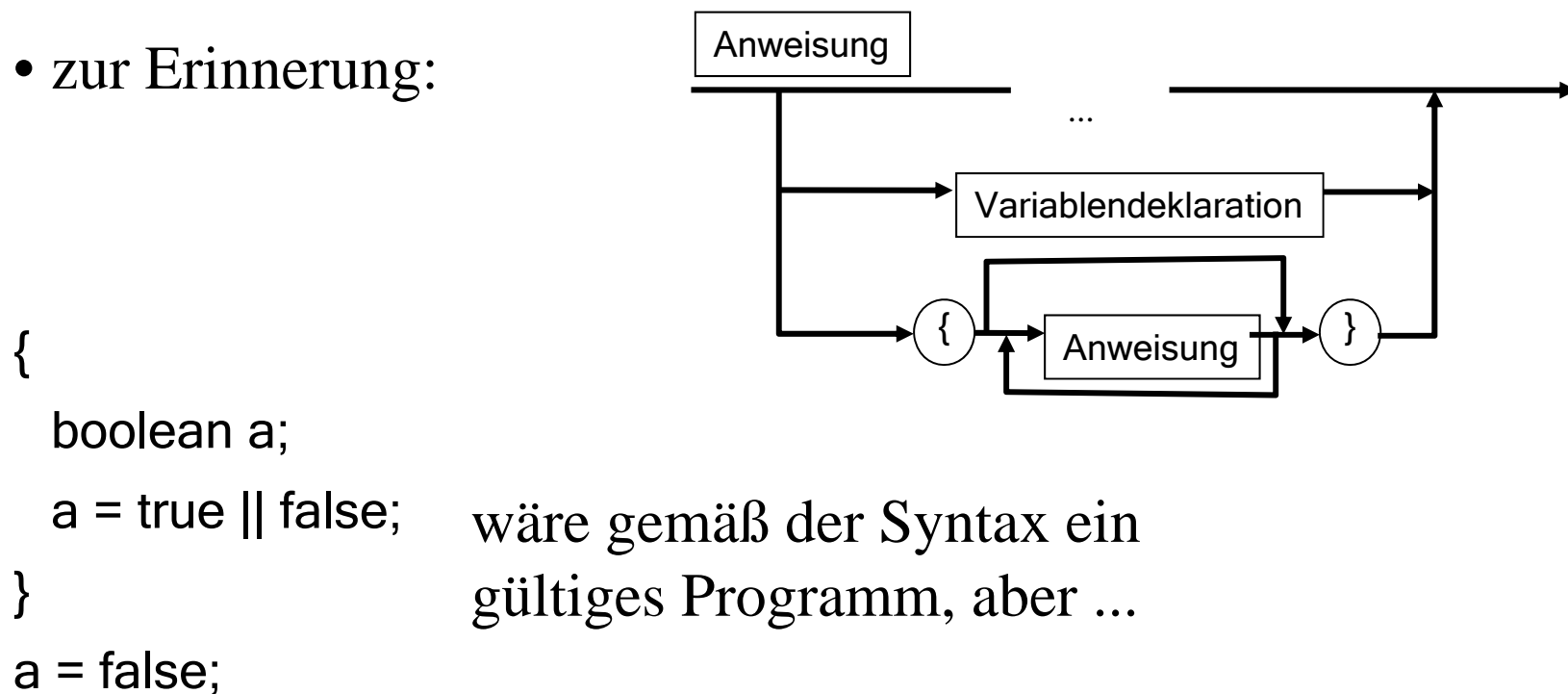
Der Block (Fort.)

```
public class EinBlock {  
    public static void main(String[] args) {  
        System.out.println("Ich habe 2 Anweisungen");  
        {  
            System.out.println("ich habe auch 2 Anweisungen");  
            System.out.println("die stehen aber in einem Block");  
        }  
    }  
}
```

- **main** hat 2 Anweisungen: **println** und einen **Block**
- **Block** enthält ebenfalls 2 Anweisungen

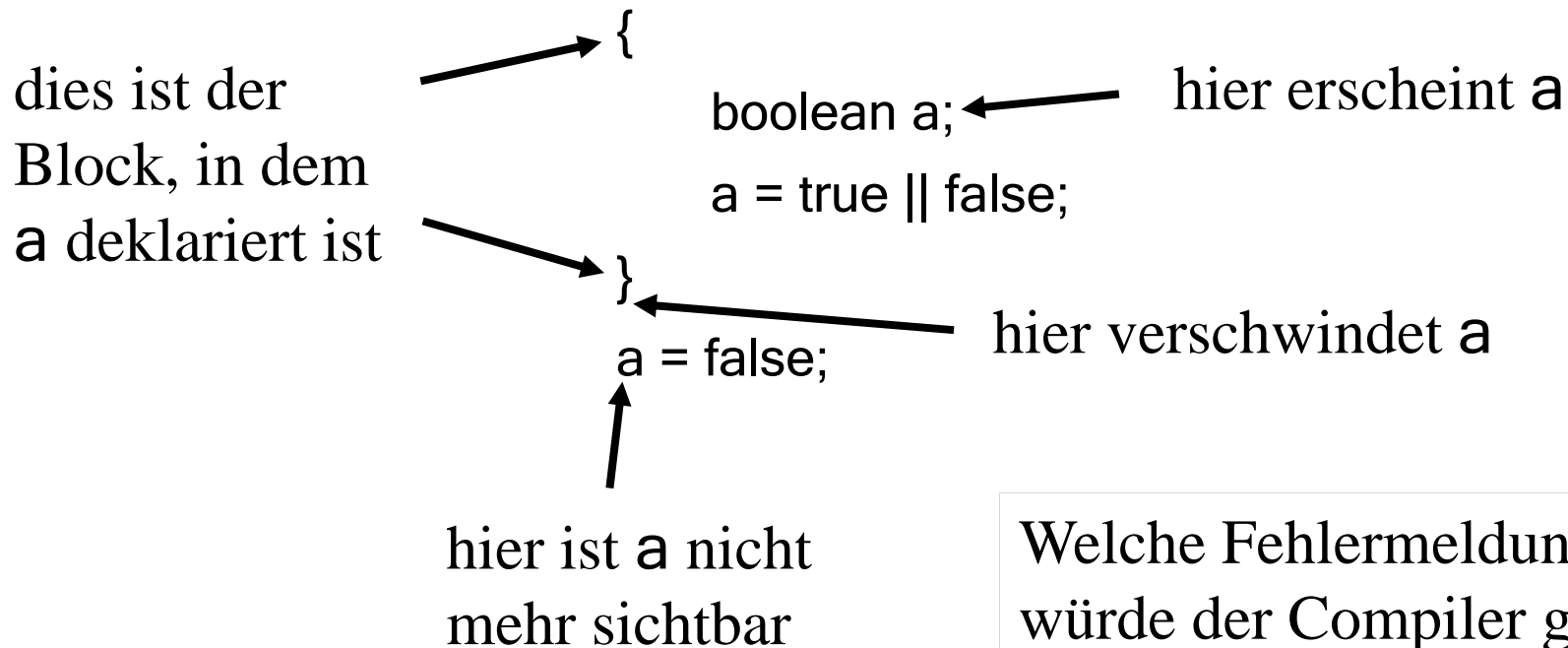
Der Block (Fort.)

- Blöcke werden oft in Verbindung mit anderen Anweisungen verwendet (später diese Vorlesung)
- Blöcke haben eine Wechselwirkung mit Variablendeklarationen
- zur Erinnerung:



Der Block (Fort.)

- Variablen erscheinen mit ihrer Deklaration
- Variablen verschwinden am Ende des Blocks, in dem sie deklariert werden



Sichtbarkeit

- wann Variablen erscheinen und wann sie verschwinden regelt die sogenannte *Sichtbarkeit*
- Problem des Überschreibens:

```
{  
    boolean a;  
    {  
        boolean a;  
        a = true || false;  
    }  
    a = false;  
}
```

Welches a ist hier gemeint?


In Java dürfen so einfache Variablen *nicht überdefiniert* werden (im Gegensatz zu C/C++)

Sichtbarkeit (Fort.)

1. Variablen erscheinen mit ihrer Deklaration
2. Variablen verschwinden am Ende des Blocks, in dem sie deklariert werden
3. dazwischen sind die überall (auch in Subblöcken) sichtbar

```
{  
    boolean a;  
    {  
        a = false;  
    }  
    a = true || false;  
}
```

hier ist **a** sichtbar, das
Programmstück ist korrekt



Verzweigungen

bisher:

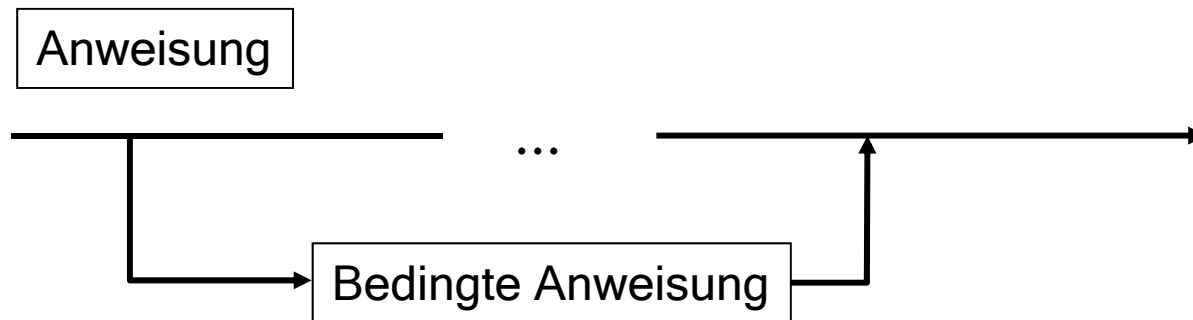
- boolesche Variablen deklarieren
- Werte zuweisen
- bei der Deklaration schon Werte zuweisen
- mit den Werten neue Werte berechnen
- Werte mittels println Anweisung auf dem Bildschirm ausgeben

nicht möglich bisher:

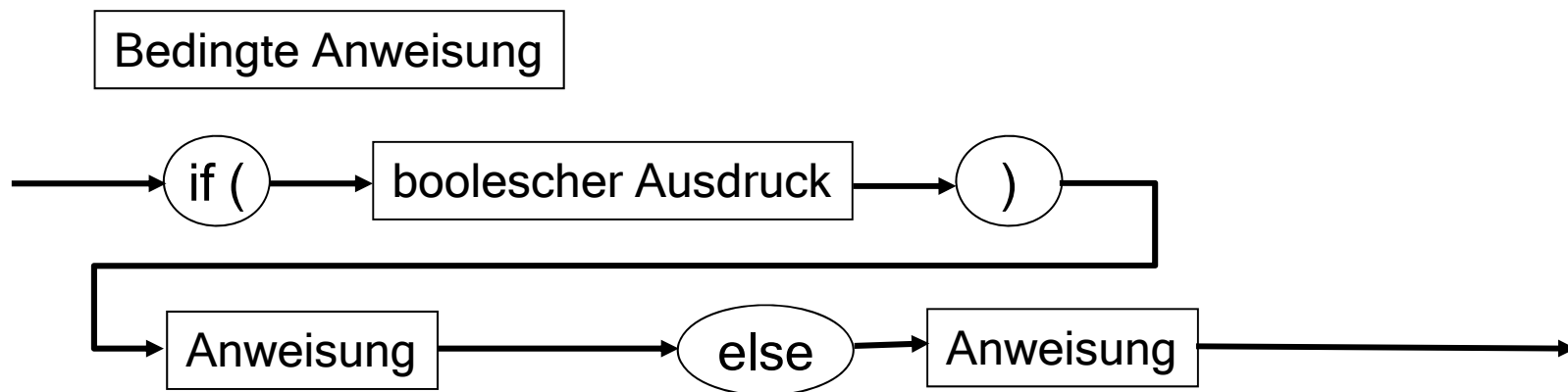
- in Abhängigkeit von einem Wert den Programmablauf steuern

Verzweigungen (Fort.)

- eine Anweisung kann eine bedingte Anweisung sein



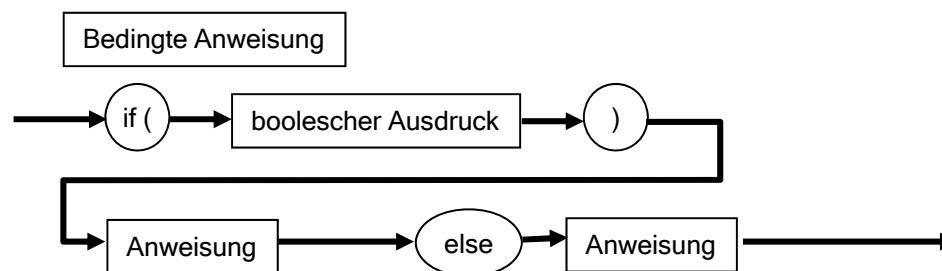
- eine bedingte Anweisung besteht aus 3 Teilen: einem booleschen Ausdruck und 2 Anweisungen (dem **then**- und **else** Branch)



Verzweigungen (Fort.)

Semantik:

- der boolesche Ausdruck wird ausgewertet
- ist er *wahr*, so wird die **1. Anweisung** ausgeführt
- ist er *falsch*, so wird die **2. Anweisung** ausgeführt
- nachdem *entweder* die 1. *oder* die 2. Anweisung ausgeführt worden ist, wird *nach* der bedingten Anweisung *weitergearbeitet*



Go!

Beispiel

```
public class BedingteAnweisung1 {  
    public static void main(String[] args) {  
        boolean a = true || false , b = !a;  
        if (!a || b)  
            System.out.println("der Ausdruck ist wahr");  
        else  
            System.out.println("der Ausdruck ist falsch");  
        !a || b wird ausgerechnet und eine der beiden  
        println Anweisungen wird ausgeführt  
        System.out.println("und jetzt ist Schluss");  
    }  
}
```

die booleschen Variablen **a** und **b** werden deklariert und initialisiert

danach geht es hier weiter

Go!

Verzweigungen (Fort.)

Was gibt folgendes Programm aus?

```
public class BedingteAnweisung2 {  
    public static void main(String[] args) {  
        if (true)  
            System.out.println("dies ist der True-Fall");  
        else  
            System.out.println("dies ist");  
            System.out.println(" der False-Fall");  
        }  
    }
```

Go!

Verzweigungen (Fort.)

Ist das folgende Programm korrekt?

```
public class BedingteAnweisung3 {  
    public static void main(String[] args) {  
        if (true)  
            System.out.println("dies ist ");  
            System.out.println("der True-Fall");  
        else  
            System.out.println("dies ist der False-Fall");  
    }  
}
```

Go!

Verzweigungen (Fort.)

- in dem **then-** bzw. **else-**Branch muss *genau eine* Anweisung stehen
- sollen mehrere Anweisungen unter Kontrolle einer bedingten Anweisung stehen, müssen sie zu einem **Block** zusammengefasst werden, da ...
- ... ein **Block** wie eine *einzelne Anweisung* behandelt wird

```
public class BedingteAnweisung4 {  
    public static void main(String[] args) {  
        if (true)  
        {  
            System.out.println("dies ist ");  
            System.out.println("der True-Fall");  
        }  
        else  
            System.out.println("dies ist der False-Fall");  
    }  
}
```

Verzweigungen (Fort.)

- bei Blöcken in bedingten Anweisungen unterscheidet man zwischen 2 üblichen Schreibweisen

```
if (...)
{
    ...;
    ...;
}
else
{
    ...;
    ...;
}
```

```
if (...) {
    ...;
    ...;
} else {
    ...;
    ...;
}
```

- Gewöhnen Sie sich eine an und bleiben Sie dabei !

Verzweigungen (Fort.)

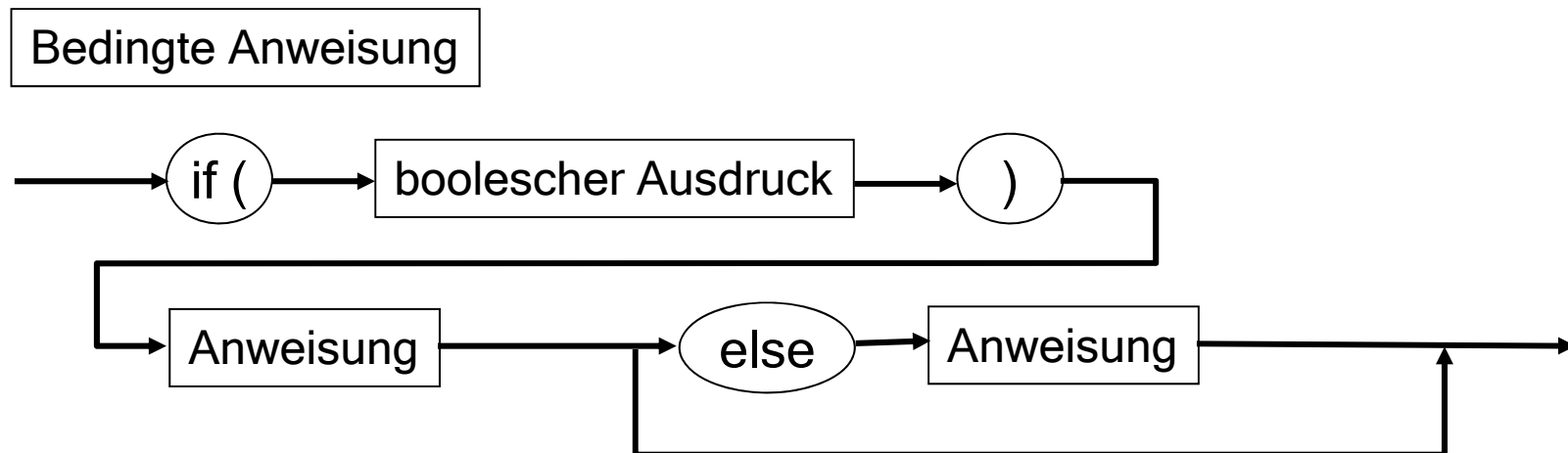
- Was ist zu tun, wenn nur im **then-Branch** Anweisungen stehen sollen, im **else-Branch** aber nichts getan werden soll?
- Lösung: leere Anweisung
 - entweder mit ;
 - oder mit einem leeren Block {}
- Beispiel:

```
if (a || b)
    System.out.println("fein");
else
    ;
```

```
if (a || b)
    System.out.println("fein");
else {
}
```

Verzweigungen (Fort.)

- Lösungen sind unbefriedigend, da
 - Situation kommt sehr häufig vor
 - ist nicht sehr lesbar
 - es werden 2 Zeilen gebraucht, die nichts machen
- richtige Lösung: der **else**-Branch kann weggelassen werden



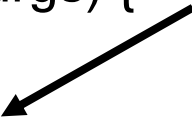
Go!

Verzweigungen (Fort.)

Beispiel:

```
public class BedingteAnweisung5 {  
    public static void main(String[] args) {  
        if (false)  
            System.out.println("dies ist ein True-Fall ohne False");  
        System.out.println("und Schluss");  
    }  
}
```

diese print-Anweisung
wird ***nicht*** ausgeführt




Go!

Verzweigungen (Fort.)

Vorsichtig: auch hier gilt, nur die *eine nachfolgende* Anweisung steht unter Kontrolle der Bedingung

```
public class BedingteAnweisung6 {  
    public static void main(String[] args) {  
        if (false)  
            System.out.println("dies ist ein ");  
            System.out.println("True-Fall ohne False");  
        System.out.println("und Schluss");  
    }  
}
```

diese print-Anweisung
steht *nicht* unter
Kontrolle der Bedingung



Hier muss zu einem Block geklammert werden

Go!

Verzweigungen (Fort.)

Vorsichtig vor geschachtelten bedingten Anweisungen
(*dangling-else Problem*)

```
public class BedingteAnweisung7 {  
    public static void main(String[] args) {  
        if (false)  
            if (true)  
                System.out.println("Fall 1");  
        else  
            System.out.println("Fall 2");  
    }  
}
```

Was gibt dieses Programm aus?

Go!

Verzweigungen (Fort.)

- eine **else**-Anweisung gehört immer zur letzten if-Anweisung
- es richtet sich ***nie*** an der Einrückung
- für (fast) alle Programmiersprachen spielt die ***Einrückung keine Rolle***

```
public class BedingteAnweisung8 {  
    public static void main(String[] args) {  
        if (false) {  
            if (true)  
                System.out.println("Fall 1");  
        } else  
            System.out.println("Fall 2");  
    }  
}
```

else gehört jetzt zu

Vorlesung 3/1

Kommentare

- in einem Quellcode möchte man u.U. auch umgangssprachliche Kommentare hinschreiben
- in Java kann dies erfolgen, indem man die Zeichen `//` verwendet
- alles, was zwischen `//` und dem Zeilenende steht ist ein Kommentar und wird vom Compiler überlesen
- Kommentare können auch in der Form `/*` und `*/` vorkommen
- alles, was zwischen diesen beiden Formen steht, ist Kommentar, auch wenn es über mehrere Zeilen geht
- aber Vorsicht, Kommentare der Form `/* */` können nicht geschachtelt werden

Kommentare (Fort.)

Beispiel (insgesamt 3 Kommentare)

```
public class World {  
    public static void main(String[] args) {  
        System.out.println("Hello world!");  
    }  
}
```

// dies ist ein Kommentar
// dies ist ein neuer
// Kommentar

Beispiel (insgesamt 1 Kommentar; Anweisung ist ausgeklammert)

```
public class World {  
    public static void main(String[] args) {  
        System.out.println("Hello world!");  
    }  
}
```

/* dies ist ein Kommentar
der bis hier geht */

Kommentare (Fort.)

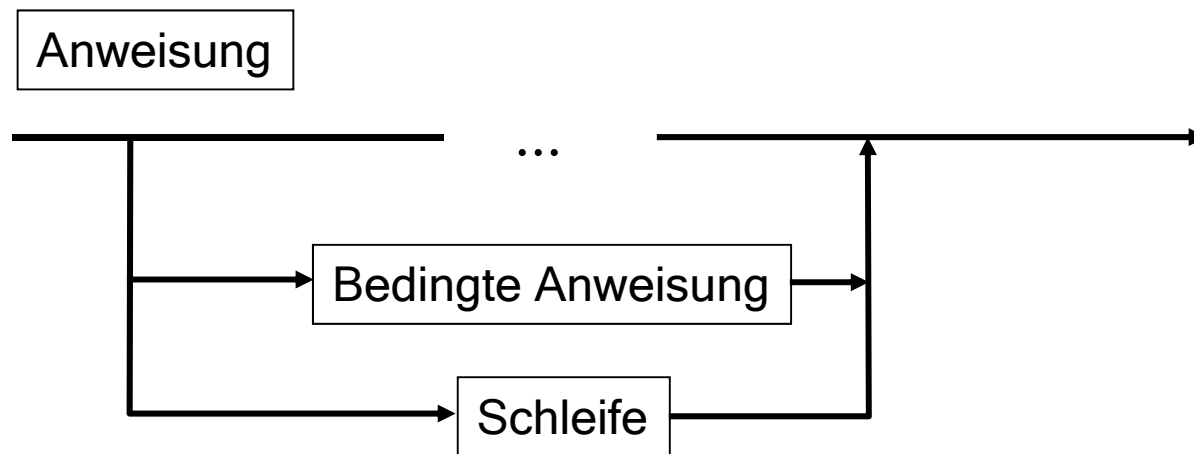
Beispiel (falscher Kommentar)

```
public class World {  
    public static void main(String[] args) {  
        System.out.println("Hello world!");  
    }  
}
```

```
/* dies ist ein Kommentar  
/* noch einer */  
der bis hier geht */
```

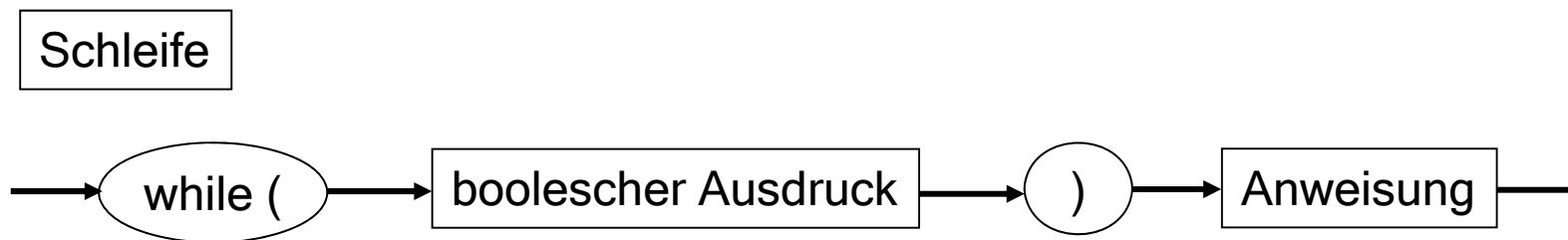
Schleifen

- Schleifen dienen dazu, Anweisungen immer wieder auszuführen solange eine bestimmte Bedingung gilt
- ist die Bedingung einmal nicht mehr erfüllt, so wird im Programmablauf nach der Schleife weitergearbeitet
- eine Schleife ist wiederum eine einzelne Anweisung



Schleifen (Fort.)

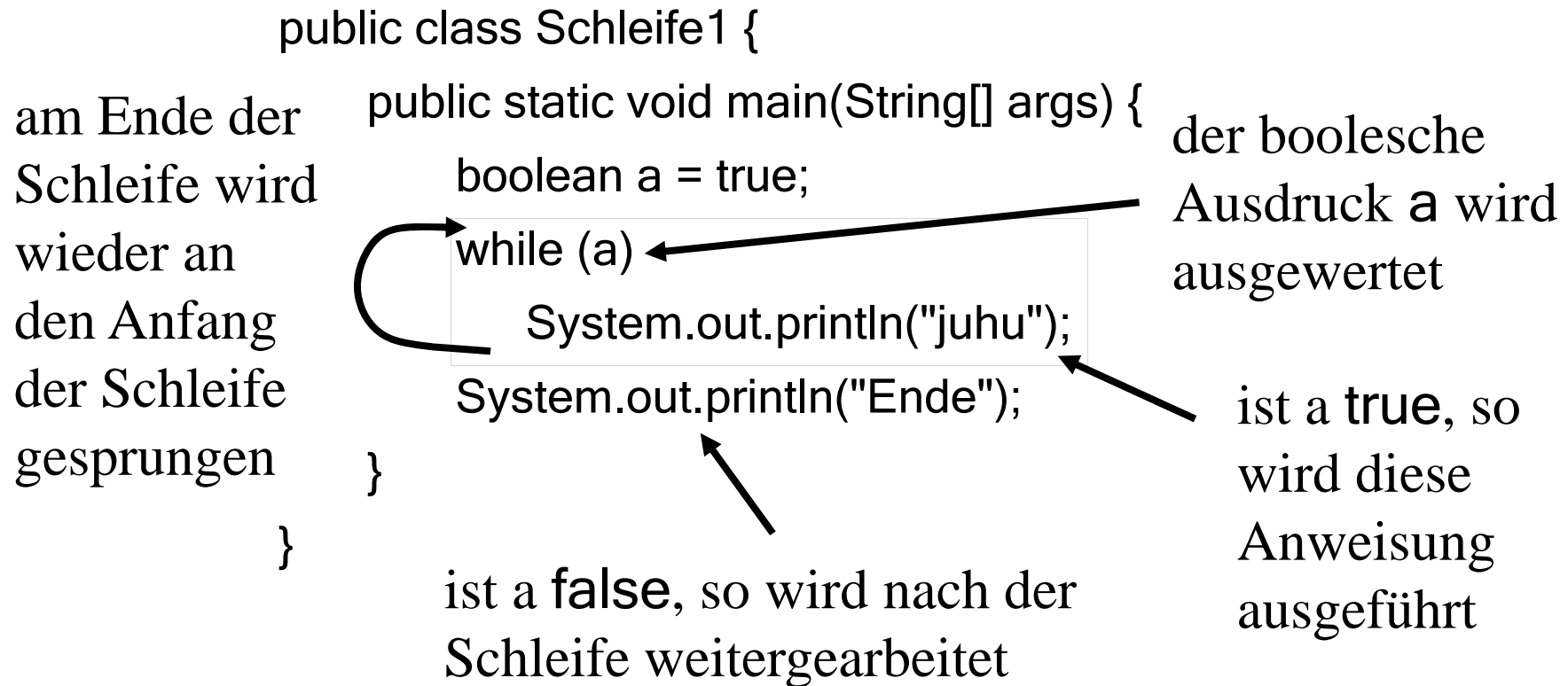
- es gibt in Java 3 verschiedene Schleifen
- die einfachste Schleife ist die while-Schleife



- der boolesche Ausdruck wird ausgewertet
- ist er **true**, so wird die Anweisung ausgeführt und die Schleife beginnt von neuem
- ist er **false**, so wird nach der Anweisung weitergearbeitet

Go!

Beispiel: while-Schleife



Was macht das Programm?

Go!

Schleifen (Fort.)

Vorsicht bei Schleifen:

- die Schleifenbedingung sollte sich im Schleifenrumpf irgendwann ändern
- ansonsten hat man eine *Endlosschleife*
- Endlosschleifen werden nie mehr verlassen, wenn sie einmal betreten worden sind
- nächster Versuch:
- ist er besser?

```
public class Schleife1b {  
    public static void main(String[] args) {  
        boolean a = true;  
        while (a)  
            System.out.println("juhu");  
        a = false;  
        System.out.println("Ende");  
    }  
}
```

Go!

Schleifen (Fort.)

Bei Schleifen gilt das Gleiche wie bei **then-** und **else-**Branches:

- zum Schleifenrumpf gehört nur genau eine Anweisung
- sollen *mehrere Anweisungen* hintereinander *im Schleifenrumpf* ausgeführt werden, müssen sie mittels eines *Blocks* zu einer Anweisung *zusammengefasst werden*

```
public class Schleife1c {  
    public static void main(String[] args) {  
        boolean a = true;  
        while (a) {  
            System.out.println("juhu");  
            a = false;  
        }  
        System.out.println("Ende");  
    }  
}
```

Typen

Unter *Typen* versteht man die *Menge der möglichen Werte*, die eine Variable annehmen kann.

bisher:

- einen Typen: **boolean**, die Menge der booleschen Werte (wahr/falsch bzw. true/false)
- reicht aus für Aussagenlogik
- Was ist mit dem Rechnen? Wie sieht es mit Buchstaben (Zeichen) und Buchstabenfolgen aus?

jetzt:

- neue Typen für Buchstaben und Folgen von Buchstaben
- neue Typen für Zahlen
- Konvertierung von einem Typ in einen anderen

Typen (Fort.)

Es gibt in Java 8 elementare Typen:

- **boolean** der Wahrheitswert, wahr oder falsch
- **char** das Zeichen, z.B. 'c' oder 'a' oder '?' oder '2' ...
- **byte** die ganzen Zahlen von -128 bis 127
- **short** die ganzen Zahlen von -32768 bis 32767
- **int** die ganzen Zahlen von -2^{31} bis $2^{31}-1$
- **long** die ganzen Zahlen von -2^{63} bis $2^{63}-1$
- **float** die Zahlen $\pm 3.40282347 \cdot 10^{38}$
- **double** die Zahlen $\pm 1.79769313486231570 \cdot 10^{308}$

Der boolesche Typ

- der boolesche Typ dient zur Darstellung der Wahrheitswerte wahr und falsch
- es gibt 2 Konstanten: **true** und **false**
- boolesche Ausdrücke werden über boolesche Variablen, den beiden Konstanten und über die booleschen Operatoren **&&** und **||** und **!** gebildet
- zusätzlich gibt es noch die Operatoren **^** und **&** und **|**
- boolesche Ausdrücke können auch durch Vergleiche der anderen Typen untereinander entstehen (siehe weitere Folien)

Die booleschen Operatoren \wedge und $\&$ und $|$

- der boolesche Operator \wedge verknüpft 2 boolesche Ausdrücke mittels des exklusiven Oders

Disjunktion

P	Q	$P \vee Q$
T	T	T
T	F	T
F	T	T
F	F	F

exklusives Oder

P	Q	$P \wedge Q$
T	T	F
T	F	T
F	T	T
F	F	F

- das exklusive Oder unterscheidet sich von dem normalen Oder, dass beide Werte unterschiedlich sein müssen
- damit ist es die Negation von der Äquivalenz

Die booleschen Operatoren ^ und & und | (Fort.)

- die beiden Operatoren & und | stellen die Konjunktion bzw. Disjunktion da
- sie unterscheiden sich jedoch von den beiden Operatoren && bzw. || dadurch, dass in jedem Fall beiden Teilausdrücke berechnet werden

```
boolean q;  
q = aus1 && aus2;
```

=

```
boolean q;  
if (aus1)  
    q = aus2;  
else  
    q = false;
```

```
boolean q;  
q = aus1 & aus2;
```

=

```
boolean q;  
if (aus1)  
    q = aus2;  
else {  
    boolean dummy = aus2;  
    q = false;  
}
```


Die booleschen Operatoren ^ und & und | (Fort.)

```
boolean q;  
q = aus1 || aus2;
```

=

```
boolean q;  
if (aus1)  
    q = true;  
else  
    q = aus2;
```

```
boolean q;  
q = aus1 | aus2;
```

=

```
boolean q;  
if (aus1) {  
    boolean dummy = aus2;  
    q = true;  
} else  
    q = aus2;
```

diese Unterscheidung macht sich nur dann bemerkbar, wenn **aus2** ein boolescher Ausdruck mit Seiteneffekten ist

Der Zeichentyp

- der Zeichentyp in Java besteht aus 2 Byte (16 Bit)
- er kann somit den Unicode darstellen
- dies ist deutlich mehr als der Ascii Zeichensatz (1 Byte = 8 Bit)
- hierin unterscheidet sich Java von den meisten anderen Programmiersprachen
- die normalen Zeichen können direkt eingegeben werden
- Sonderzeichen, die außerhalb des Ascii Zeichensatzes liegen, können durch `\uXXXX` eingegeben werden, wobei XXXX eine Hexadezimalzahl ist

Der Zeichentyp (Fort.)

- neben den durch Hexadezimalzahlen kodierten Zeichen gibt es noch die folgenden Zeichen:

`\b` Rückschritt (Backspace)

`\t` horizontaler Tabulator

`\n` Zeilenschaltung (Newline)

`\f` Seitenumbruch (Formfeed)

`\r` Wagenrücklauf (Carriage return)

`\"` doppeltes Anführungszeichen

`\'` einfaches Anführungszeichen

`\\` Backslash

Go!


Beispiel

```
public class Zeichen {  
    public static void main(String[] args) {  
        System.out.println("Hello world\u00f6");  
        System.out.println("dies \" ist ein doppeltes Anführungszeichen");  
        System.out.println("dies ' ist ein einfaches Anführungszeichen");  
        System.out.println("dies \b ist ein Rückschritt");  
        System.out.println("dies \t ist ein Tabulator");  
        System.out.println("dies \n ist ein neue Zeile");  
        System.out.println("dies \f ist ein Seitenumbruch");  
        System.out.println("dies \r ist ein Zeilenrücklauf");  
    }  
}
```

Go!

Beispiel

```
public class Zeichen2 {  
    public static void main(String[] args) {  
        while (true) {  
            System.out.print("\r|");  
            System.out.print("\r/");  
            System.out.print("\r-");  
            System.out.print("\r\\");  
        }  
    }  
}
```

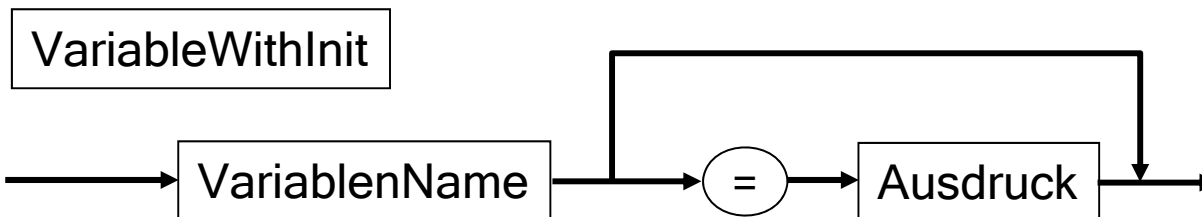
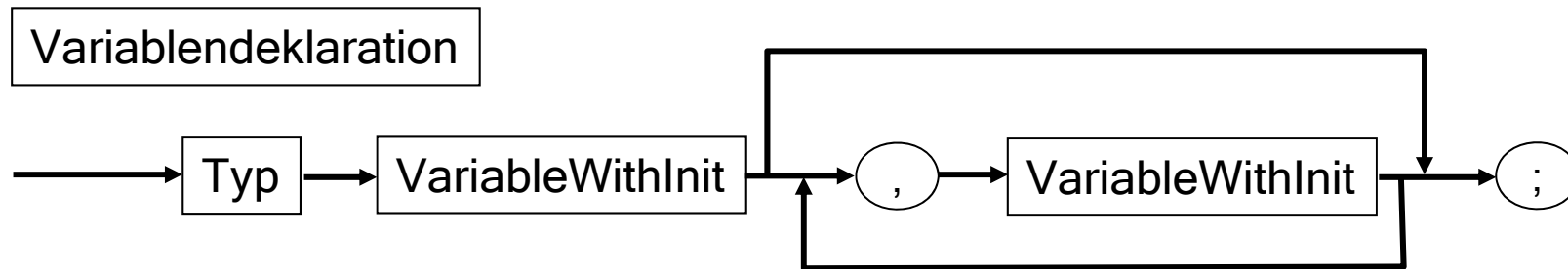


print druckt den String
ohne Zeilenumbruch

Was macht dieses Programm?

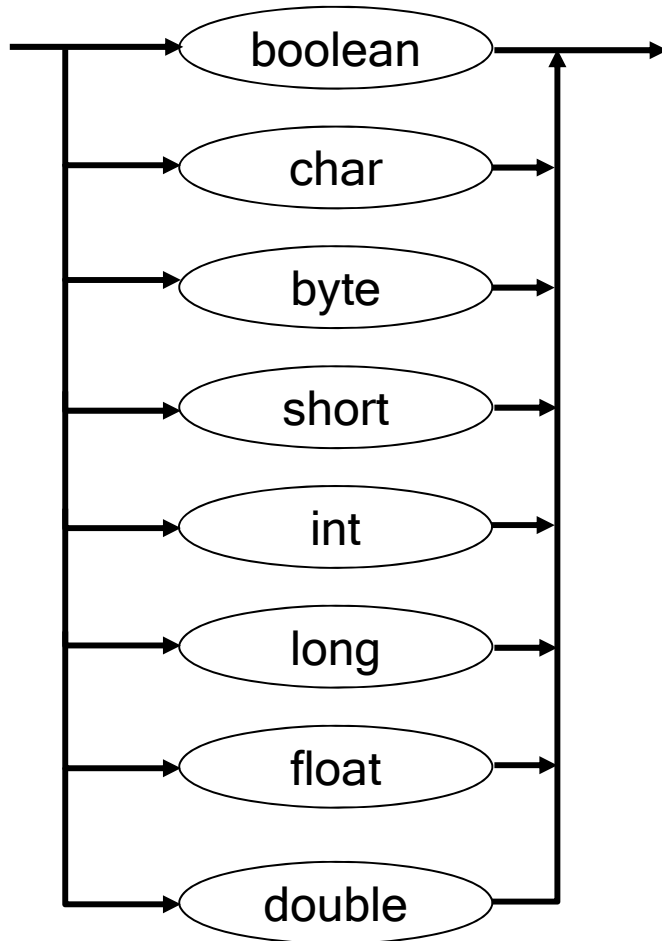
Typen (Fort.)

- analog zu booleschen Variablen gibt es auch Variablen, die vom Typ `char` sind und ein Zeichen abspeichern können
- die Deklaration solcher Variablen erfolgt analog zu den der booleschen Variablen



Typen (Fort.)

Typ



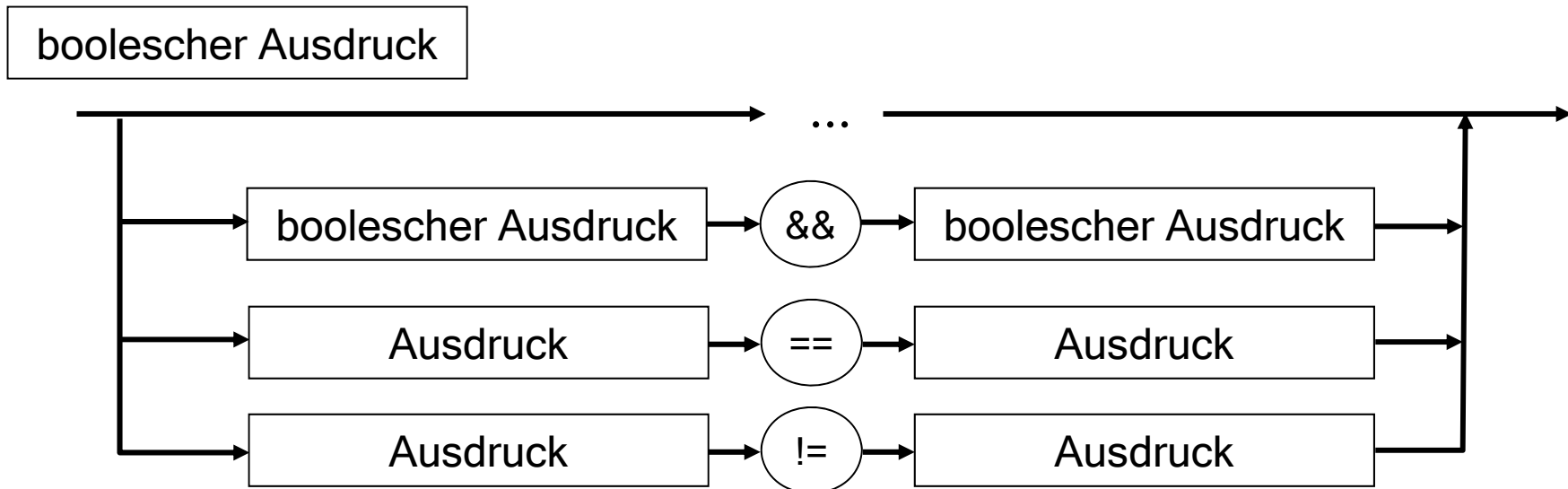
- eine Variable kann von einem beliebigen der 8 elementaren Datentypen sein
- wird die Variable bei der Deklaration initialisiert, so muss der Ausdruck vom Typ der Variablen sein, z.B.
 - `boolean a = true;`
 - `char c = '?';`
 - `int i = 43, j = 2*i;`

Der Zeichentyp (Fort.)

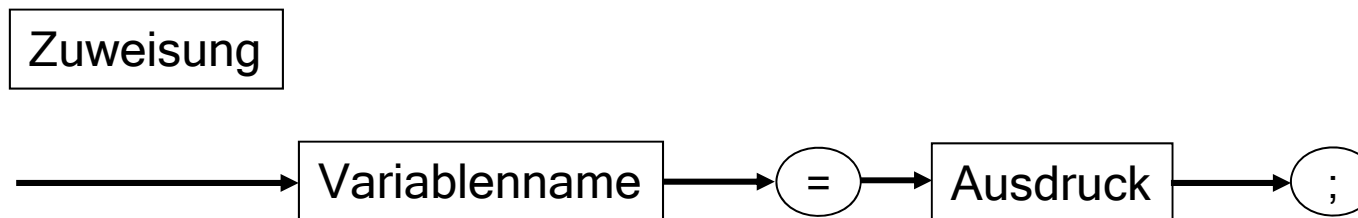
- einer Variablen vom Typ `char` können Ausdrücke vom Typ `char` zugewiesen werden
- dies sind aber nur die Zeichen selber
- Zeichen können aber durch die beiden Operatoren `==` und `!=` miteinander verglichen werden
- das Ergebnis ist ein boolescher Wert
- `==` ist wahr, wenn die beiden Zeichen rechts und links identisch sind
- `!=` ist wahr, wenn die beiden Zeichen rechts und links unterschiedlich sind

Der Zeichentyp (Fort.)

- ein boolescher Ausdruck ist auch ein Vergleich



- bei einer Zuweisung müssen Variable und Ausdruck vom gleichen Typ sein



Go!

Beispiel

```
public class Zeichen3 {  
    public static void main(String[] args) {  
        char c = '|';  
        while (true) {  
            System.out.print("\r");  
            System.out.print(c);  
            if (c == '|')  
                c = '/';  
            else if (c == '/')  
                c = '-';  
            else if (c == '-')  
                c = '\\';  
            else  
                c = '|';  
        }  
    }  
}
```

eine Variable vom Typ char
wird deklariert und initialisiert

Wagenrücklauf und Ausgabe des
Zeichens, dass in c gespeichert ist

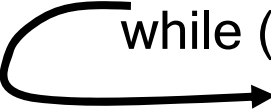
geschachtelte if-Anweisungen,
die das Zeichen in c umsetzen

Was macht das Programm?

Go!

Beispiel

```
public class Zeichen4 {  
    public static void main(String[] args) {  
        while (true) {  
            char c = '|';  
            System.out.print("\r");  
            System.out.print(c);  
            if (c == '|')  
                c = '/';  
            else if (c == '/')  
                c = '-';  
            else if (c == '-')  
                c = '\\';  
            else  
                c = '|';  
        }  
    }  
}
```



Deklaration von **c** in
die Schleife verlegt

Ist dies immer noch ein
gültiges Programm?

Wenn ja, was macht es?

Vorlesung 3/2

Nochmal: Sichtbarkeit

```
public class Zeichen4 {  
    public static void main(String[] args) {  
        while (true) {  
            char c = '|';  
            System.out.print("\r");  
            System.out.print(c);  
            ...  
        }  
    }  
}
```

In jedem Schleifendurchlauf wird ein neues **c** angelegt, somit werden die *Änderungen* **nicht** wahrgenommen

```
public class Zeichen3 {  
    public static void main(String[] args) {  
        char c = '|';  
        while (true) {  
            System.out.print("\r");  
            System.out.print(c);  
            ...  
        }  
    }  
}
```

Es gibt nur ein **c**, dass die ganze Laufzeit existiert; somit werden die *Änderungen* in der Schleife im nächsten Durchlauf wahrgenommen

Die Sichtbarkeit der Variablen hat einen starken Einfluss auf die Bedeutung des Programms

Die integralen Typen

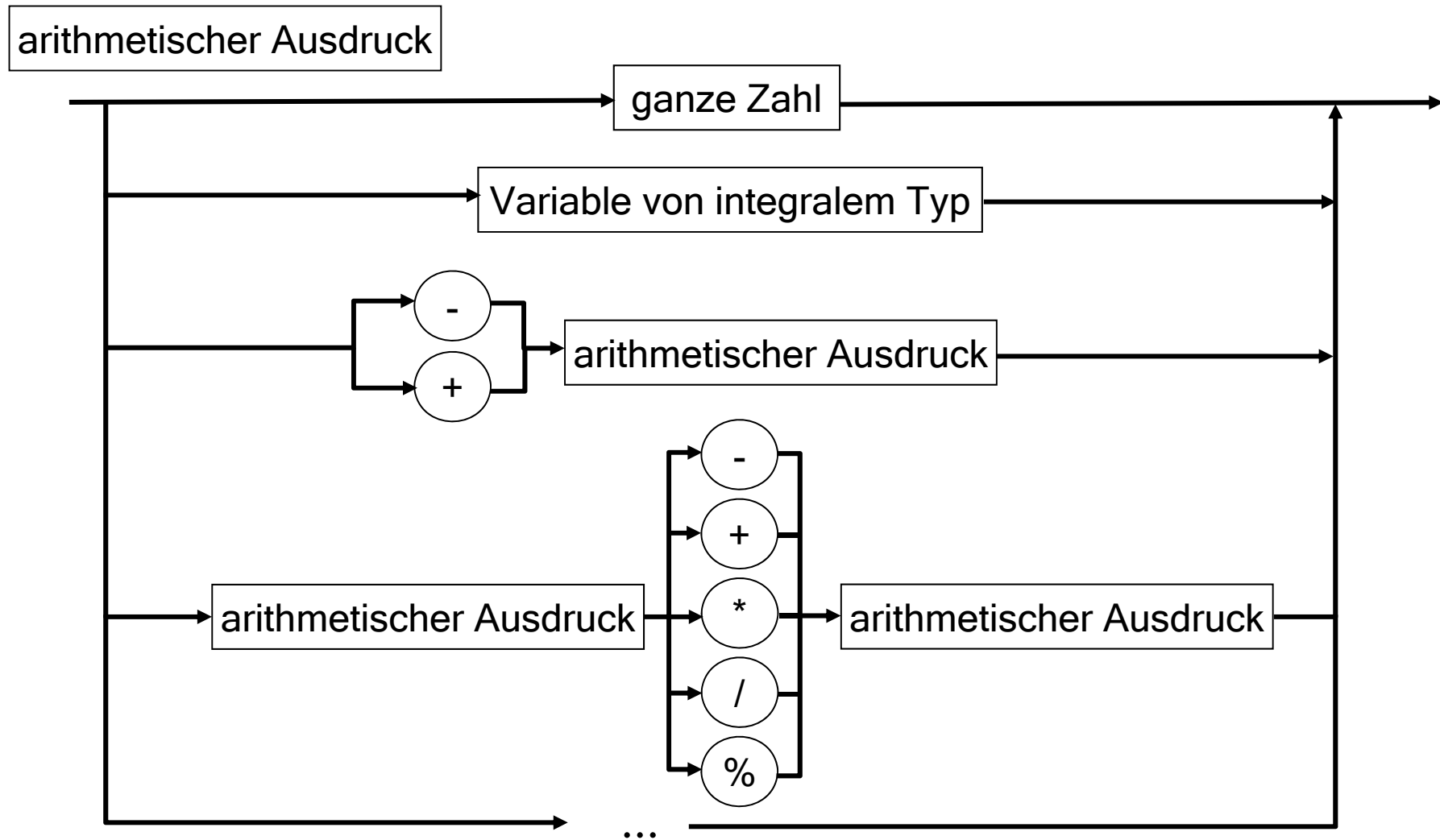
- die elementaren Typen `byte`, `short`, `int` und `long` dienen dazu, *ganze Zahlen* zu behandeln
- Variablen von diesen Typen können ganze Zahlen (*positive* und *negative*) *unterschiedlicher Größe* abspeichern
- `byte` wird in einem Byte abgespeichert \Rightarrow kann die Zahlen zwischen -128 und 127 abspeichern
- `short` wird in 2 Byte abgespeichert \Rightarrow kann die Zahlen zwischen -2^{15} und $2^{15}-1$ abspeichern
- `int` wird in 4 Byte abgespeichert \Rightarrow kann die Zahlen zwischen -2^{31} und $2^{31}-1$ abspeichern
- `long` wird in 8 Byte abgespeichert \Rightarrow kann die Zahlen zwischen -2^{63} und $2^{63}-1$ abspeichern

Die integralen Typen (Fort.)

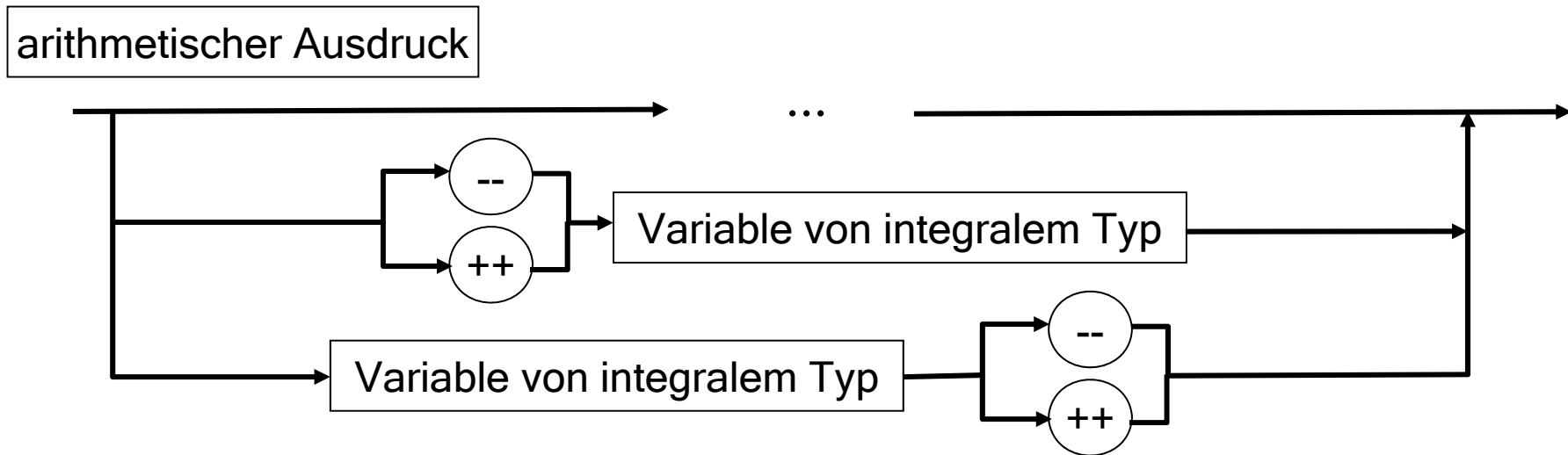
Ausdrücke über integrale Typen werden über konstante Werte (ganze Zahlen) und den folgenden Operatoren aufgebaut:

+	Vorzeichen oder Addition zweier ganzer Zahlen
-	Vorzeichen oder Subtraktion zweier ganzer Zahlen
*	Multiplikation zweier ganzer Zahlen
/	ganzzahliger Quotient zweier ganzer Zahlen
%	Restwert einer Ganzzahldivision (Modulo Operator)
++	Pre- bzw. Postinkrement (später mehr)
--	Pre- bzw. Postdekrement (später mehr)

Die integralen Typen (Fort.)



Die integralen Typen (Fort.)



- die oberen Operatoren sind die Präoperatoren; sie erhöhen bzw. erniedrigen den Wert der Variablen um 1; zusätzlich liefern sie den um 1 erhöhten/erniedrigten Wert zurück
- die unteren Operatoren sind die Postoperatoren; sie erhöhen bzw. erniedrigen den Wert der Variablen um 1; zusätzlich liefern sie den originalen Wert zurück

Die integralen Typen: Präoperatoren

- `++i` erhöht `i` um 1 liefert `i+1` zurück
- `i++` erhöht `i` um 1 liefert `i` zurück
- `--i` erniedrigt `i` um 1 liefert `i-1` zurück
- `i--` erniedrigt `i` um 1 liefert `i` zurück

Go!

Beispiel

```
public class PraeOp {  
    public static void main(String[] args) {  
        int i = 0;  
        System.out.println(i++);  
        System.out.println(i);  
        System.out.println(++i);  
        System.out.println(i);  
        i = 10;  
        System.out.println(--i);  
        System.out.println(i);  
        System.out.println(i--);  
        System.out.println(i);  
    }  
}
```

kleiner, aber feiner
Unterschied

kleiner, aber feiner
Unterschied, auch
bei --

Was ist die Ausgabe des Programms?

Go!

Die integralen Typen: Präoperatoren (Fort.)

- das Beispiel zeigt, dass die Post- und Preinkrement bzw. –dekrement Operatoren sogenannte *Seiteneffekte* haben
- d.h. die Variablen werden *nicht nur gelesen* sondern sie werden beim Lesen *auch verändert*
- dies kann zu kaum zu verstehenden Programmen führen

```
int i = 0;  
boolean a = false && ++i == 1;  
System.out.println(i);
```

```
int i = 0;  
boolean a = false & ++i == 1;  
System.out.println(i);
```

Worin besteht der Unterschied? Was wird ausgegeben?

Go!

Beispiel

```
public class Integer1 {  
    public static void main(String[] args) {  
        byte i = 0;    i wird deklariert und mit 0 initialisiert  
  
        while (i != -10) {    solange i ungleich -10 ist  
            System.out.println(i);  
            ++i;    i wird um 1 erhöht  
        }  
    }  
}
```

Was macht das Programm?

Überlauf


- das Programm hat gezeigt, dass die ganzen Zahlen in einem Ring angeordnet sind
- ist der maximale Wert einer Zahl erreicht, so ergibt die Erhöhung um 1 den minimalen Wert
- gleiches gilt in umgekehrter Reihenfolge
- ist der minimale Wert einer Zahl erreicht, so ergibt die Verringerung um 1 den maximalen Wert

Go!

Beispiel

```
public class Integer2 {  
    public static void main(String[] args) {  
        byte b = 127;  
        short s = 32767;  
        int i = 2147483647;  
        long l = 9223372036854775807L;  
        System.out.println(b); ++b;  
        System.out.println(b); --b; System.out.println(b);  
        System.out.println(s); ++s;  
        System.out.println(s); --s; System.out.println(s);  
        System.out.println(i); ++i;  
        System.out.println(i); --i; System.out.println(i);  
        System.out.println(l); ++l;  
        System.out.println(l); --l; System.out.println(l);  
    }  
}
```

long Zahlen müssen
durch ein L kenntlich
gemacht werden










Die Fließkommazahlen

- Neben den integralen Zahlen gibt es zwei Typen für Fließkommazahlen:
 - float benötigt 4 Byte
 - double benötigt 8 Byte
- Beide können positiv und negativ sein.
- Beide sind nur Approximationen der reellen Zahlen
- Es kann ein Suffix d oder f angestellt werden, um zu zeigen, dass es sich um double bzw. float Zahlen handelt
- Bsp.:

3.14	4.234d	0.123e43f
0.34e-34	3243.2434e45	

Go!

Beispiel

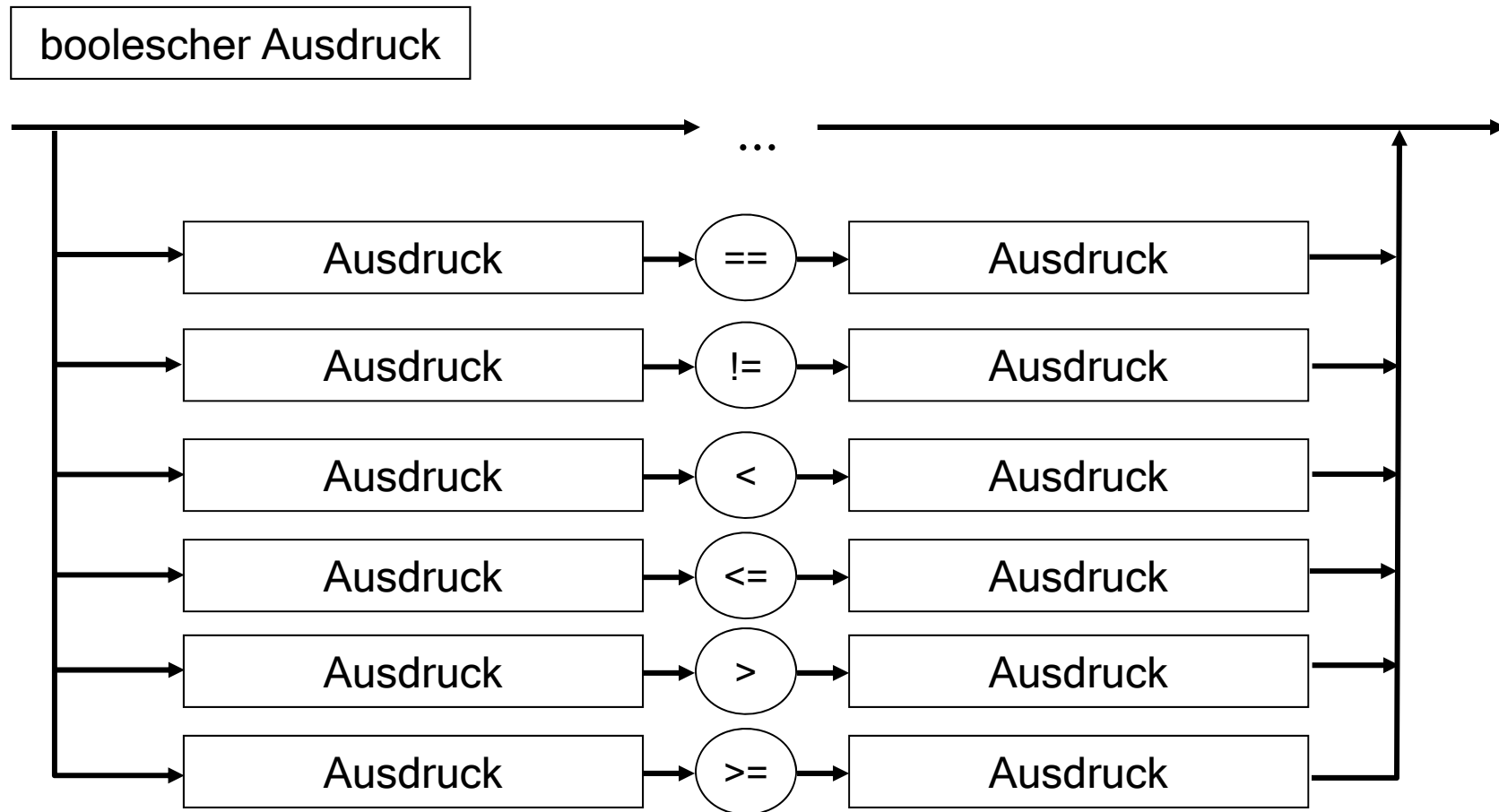
```
public class FliessKommaZahlen {  
    public static void main(String[] args) {  
        float f = 234234.342342e23f;  große Float Zahl  
        double d = 234234.342342e23d;  große Double Zahl  
        System.out.println(f);  
        System.out.println(d);  
        System.out.println(f * f);  gibt das Quadrat aus  
        System.out.println(d * d);   
        System.out.println(f + 1000.0);  addiert 1000 zu den  
        System.out.println(d + 1000.0);  großen Zahlen  
        float f2 = 3424.234234e-30f;  ganz kleine Float Zahl  
        System.out.println(f2);  
        System.out.println(f2 + 1000.0);  
    }  
}
```

Was gibt das Programm aus?

Relationale Operatoren

- alle elementaren Typen können untereinander verglichen werden
- der Vergleich kann auf Gleichheit oder Ungleichheit erfolgen
- es kann getestet werden, ob ein Ausdruck kleiner (oder gleich) oder größer (oder gleich) einem anderen Ausdruck ist
- das Ergebnis eines solchen Vergleichs ist ein boolescher Ausdruck

Relationale Operatoren (Fort.)



Go!

Beispiel

```
public class Rel1 {  
    public static void main(String[] args) {  
        int i = 0;  
        while(i < 8) {  
            if ((i % 2) == 0) {  
                System.out.println(i);  
                ++i;  
            }  
        }  
    }  
}
```

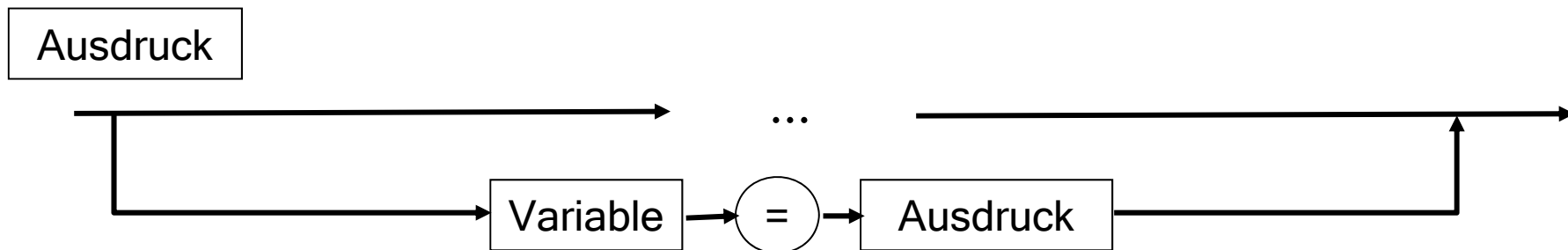
solange i kleiner als 8 ist, mache ...

wenn der Rest von i geteilt durch 2 gleich 0 ist, mache ...

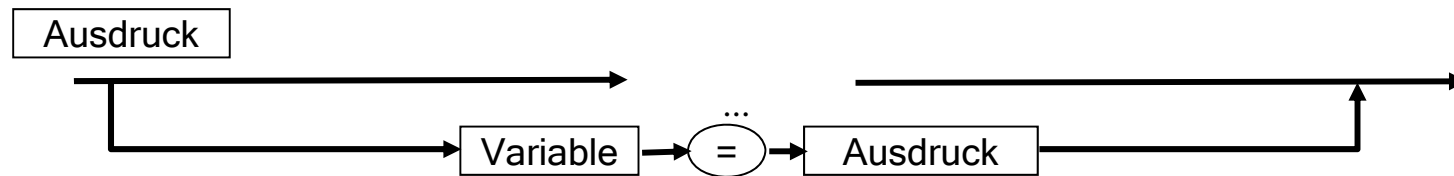
Was gibt dieses Programm aus?

Zuweisungen: der = Operator

- bisher ist der = Operator nur als Zuweisung eingesetzt worden
- Bsp.: `a = 34;`
- der Variablen `a` wird der Wert 34 zugewiesen
- jedoch macht der =-Operator mehr: es liefert den Wert des Ausdrucks auf der rechten Seite auch zurück
- somit kann der =-Operator auch in Ausdrücken verwendet werden



Zuweisungen: der = Operator (Fort.)



- der Typ dieses Zuweisungsausdrucks ist der Typ des Ausdrucks auf der rechten Seite
- der Wert dieses Zuweisungsausdrucks ist der Wert des Ausdrucks auf der rechten Seite
- dieser Wert wird zusätzlich in der Variablen abgespeichert
- neben den Prä- und Postinkrement- bzw. –dekrementoperatoren hat dieser Operator ebenfalls Seiteneffekte (er verändert die Variable)

Go!

Beispiel

```
public class Assign {  
    public static void main(String[] args) {  
        int a = 34;  
        int b;  
        b = (a = 12) - 3;  
        System.out.println(a);  
        System.out.println(b);  
        if (b > 1023 & (a = 23) == 45)  
            System.out.println("juhu");  
        System.out.println(a);  
    }  
}
```

Was gibt dieses Programm aus?

Zuweisungen: der = Operator (Fort.)

- häufig werden Variablen gelesen, modifiziert und der neue Wert wird wieder in die Variable zugeschrieben

- Bsp.:

```
int x;  
...  
x = x + 6;
```

- die Integer Variable x wird um 6 erhöht
- hierfür gibt es eine abkürzende Schreibweise:

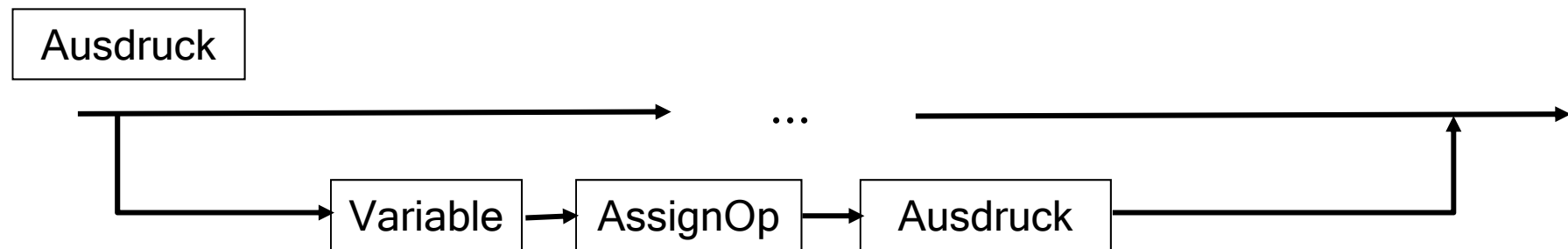
`x += 6;`

- dies gibt es für die alle Operatoren, die für Zahlen definiert sind:

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

Zuweisungen: der = Operator (Fort.)

- bei diesen Operatoren wird die Variable gemäß des Operators verändert
- der neue Wert wird in der Variablen abgespeichert
- der neue Wert wird auch zurückgeliefert, so dass er innerhalb eines Ausdrucks als Subausdruck verwendet werden kann



AssignOp ::= "=" | "+=" | "-=" | "*=" | "/=" | "%=" | "&=" |
"|" | "^=" | "<<=" | ">>=" | ">>>="

Go!

Beispiel

```
public class Assign2 {  
    public static void main(String[] args) {  
        int a = 34;  
        int b;  
        b = (a += 12) - 3;  
        System.out.println(a);  
        System.out.println(b);  
        if (b > 1023 & (a %= 23) == 45)  
            System.out.println("juhu");  
        System.out.println(a);  
    }  
}
```

Was gibt dieses Programm aus?

Der bedingte Ausdruck

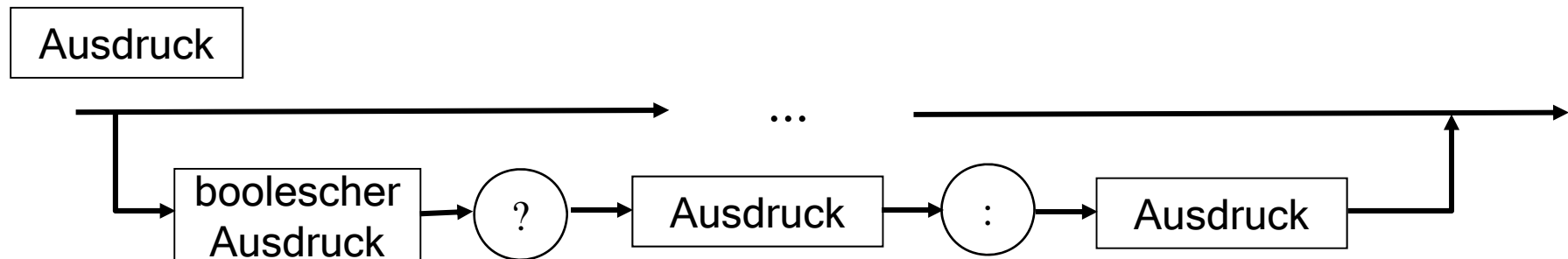
- gegeben sei folgendes Programm
- für y soll ein Wert berechnet werden, der von x und z abhängt
- der Wert von x wird unmittelbar vorher berechnet und hängt von dem booleschen Ausdruck $a \ \&\& \ b$ ab
- später wird der Wert von x nie wieder verwendet
- dafür kann man auch folgendes Programm schreiben

```
boolean a,b;  
...  
if (a && b)  
    x = 34;  
else  
    x = -17;  
y = x + 34 * z;
```

```
boolean a,b;  
...  
y = (a && b ? 34 : -17) + 34 * z;
```

Der bedingte Ausdruck (Fort.)

- der bedingte Ausdruck wird aus 2 Ausdrücken gebildet, die beide vom gleichen Typen sein müssen
- zwischen diesen beiden Ausdrücken wird mit einem booleschen Ausdruck ausgewählt
- die Schreibweise sieht wie folgt aus:
$$\langle \text{boolescher Ausdruck} \rangle ? \langle \text{Ausdruck} \rangle : \langle \text{Ausdruck} \rangle$$
- ist der boolesche Ausdruck wahr, so ist das Ergebnis des Ausdrucks der 1. Ausdruck, ist er falsch, so ist das Ergebnis des Ausdrucks der 2. Ausdruck



Go!

Beispiel

```
public class CondExpr {  
    public static void main(String[] args) {  
        int i = 0;  
        while (i < 15) {  
            int a = ((i % 2) != 0) ? i : -i;  
            System.out.println(a);  
            ++i;  
        }  
    }  
}
```

hier ist der
?: Operator

Was gibt dieses Programm aus?

Vorlesung 4/1

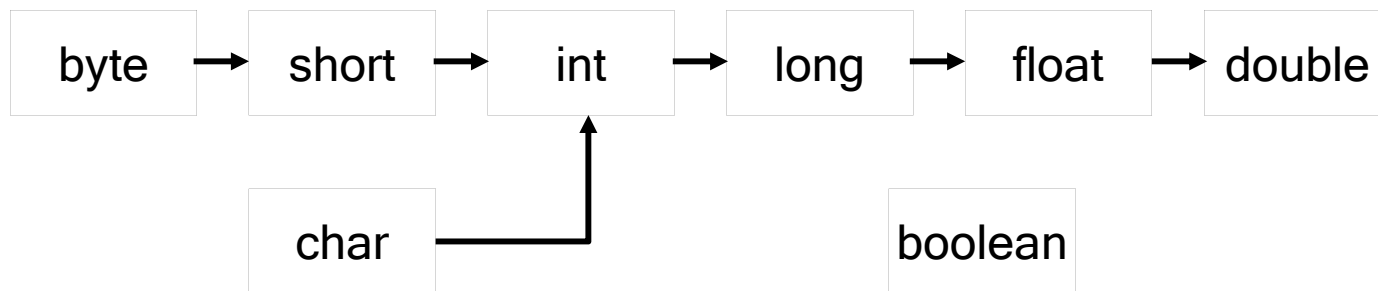
Typkonvertierung

machen die folgenden Variablendeklarationen und –
zuweisungen Sinn?

```
boolean a = 'c';  
short s = 34;  
int i = s*s;  
char c = i;  
float f = s*i + 34;  
double d = f*c;
```

Typkonvertierung (Fort.)

in Java können die 8 elementaren Datentypen ineinander konvertiert werden wie die folgende Graphik zeigt:



- d.h. die Zahlen können immer in den nächstgrößeren Typen konvertiert werden
- ein Zeichen kann immer in einen `int` konvertiert werden
- ein `boolean` wird nicht konvertiert

Go!

Beispiel

```
public class Convert {  
    public static void main(String[] args) {  
        byte b = -115;  
        short s = b;  
        int i = s;  
        long l = i;  
        float f = l;  
        double d = f;  
        System.out.println(b);System.out.println(s);System.out.println(i);  
        System.out.println(l);System.out.println(f);System.out.println(d);  
        char c = '?';  
        i = c;  
        l = c;  
        f = c;  
        d = c;  
        System.out.println(i);System.out.println(l);  
        System.out.println(f);System.out.println(d);  
    }  
}
```

implizite Typkonvertierung von einem
Typen in den nächsthöheren Typen


implizite Typkonvertierung von
einem char in die höheren Typen

Was gibt das Programm aus?

Typkonvertierung (Fort.)

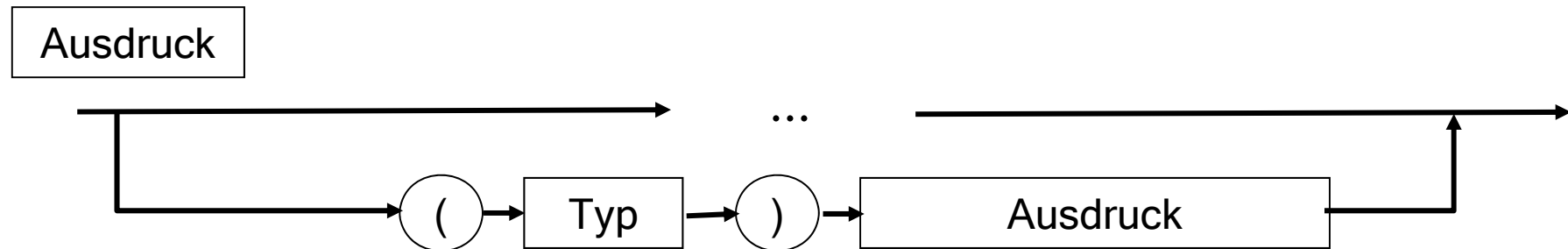
- manchmal soll aber auch ein Typ in einen niedrigeren Typen konvertiert werden
- Bsp.:

```
int i = 42;  
byte b = i;
```



- hier sollte kein Fehler entstehen, da der Programmierer weiß, dass diese Konvertierung sicher ist
- hier kann explizit der Typ konvertiert werden, indem der Zieltyp in Klammern vor dem Ausdruck geschrieben wird
- Bsp.: `byte b = (byte) i;`

Typkonvertierung (Fort.)



- ein Ausdruck eines Typs kann **explizit** zu einem anderen Typen konvertiert werden, indem dieser Typ in Klammern vor dem Ausdruck geschrieben wird
- ist der Zieltyp nicht so mächtig (hat einen kleineren Wertebereich), so **kann** durch eine explizite Typkonvertierung **Information verloren gehen**

Go!

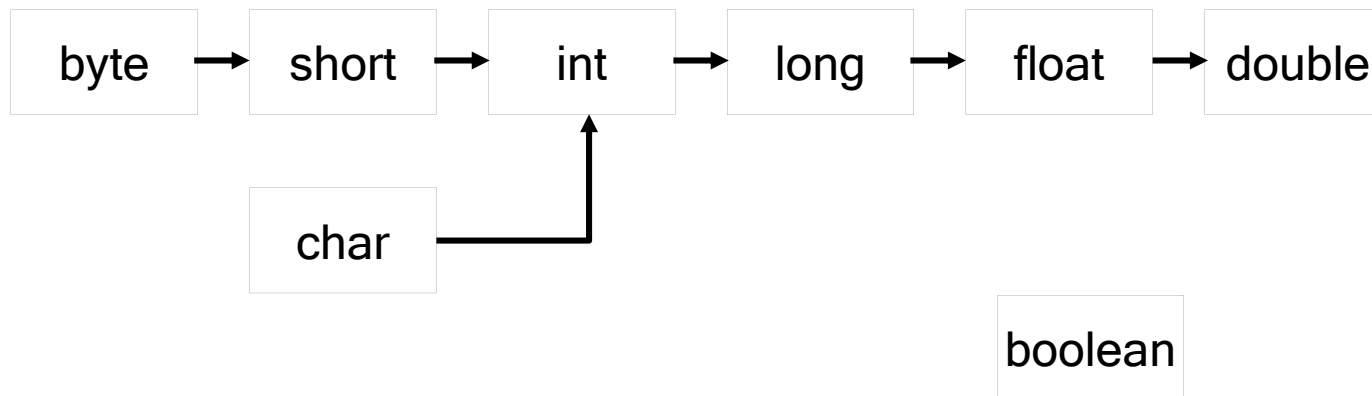
Beispiel

```
public class Convert2 {  
    public static void main(String[] args) {  
        int i = 42;  
        byte b = (byte)i; ← explizite Typkonvertierung  
        System.out.println(b);    von int in ein byte ohne  
        i = 257;                  Informationsverlust  
        b = (byte)i; ← explizite Typkonvertierung  
        System.out.println(b);    von int in ein byte mit  
    }                             Informationsverlust  
}
```

Was gibt das Programm aus?

Typkonvertierung (Fort.)

- eine explizite Typkonvertierung muss sich ebenfalls an das unten aufgeführte Schema halten
- so *können* die Zahlen in Zahlen kleineren Typs verwandelt werden
- es *kann aber kein* boolean in eine Zahl konvertiert werden



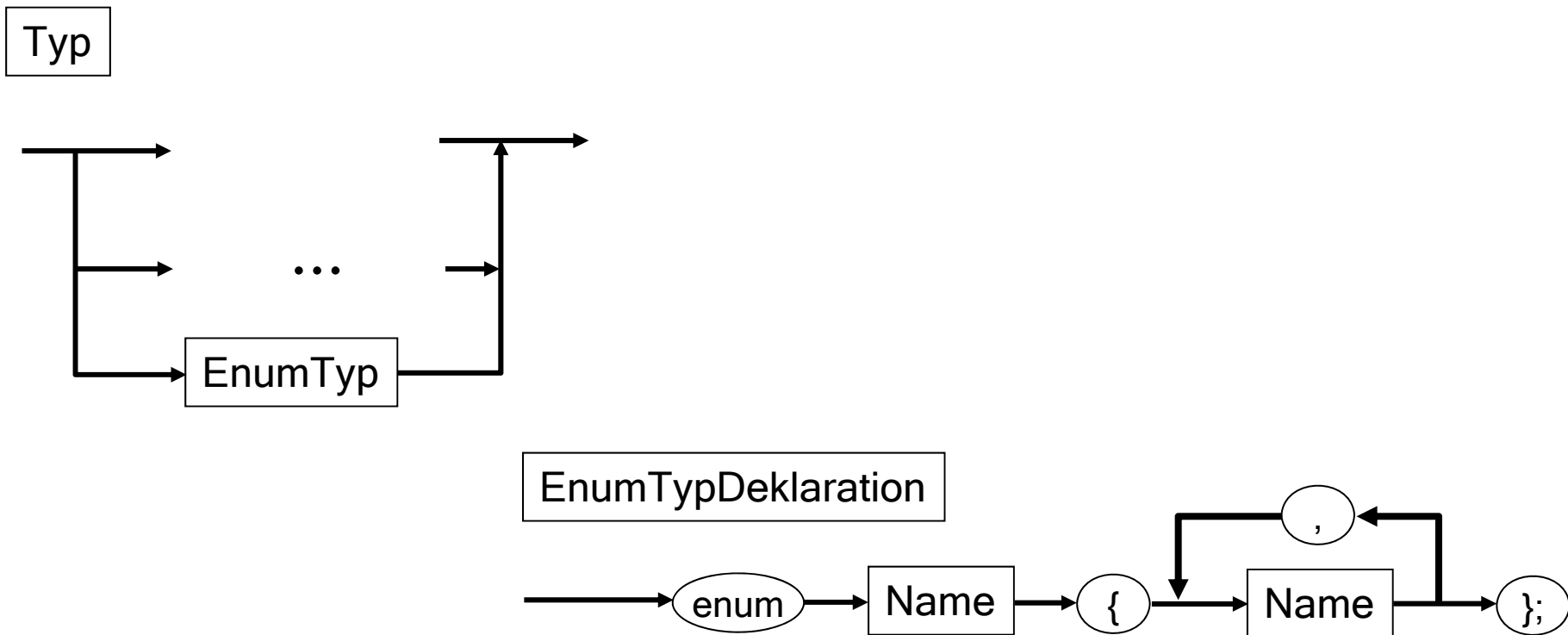
Aufzählungstypen

- in der Praxis werden oft Typen mit einer kleinen Menge von selbstgewählten festen Werten verwendet
- Beispiel:
 - Wochentage (Montag, Dienstag, Mittwoch, Donnerstag, Freitag, Samstag, Sonntag)
 - Jahreszeiten (Frühling, Sommer, Herbst, Winter)

Wie können solche Informationen mit den bisherigen Mitteln gespeichert und verarbeitet werden?

Aufzählungstypen (Forts.)

- es gibt ein eigenes Sprachkonstrukt für diese sogenannten Aufzählungstypen: enum
- Beispiel: enum Farbe {ROT, BLAU, GRUEN, GELB};



Go!

Beispiel

```
public class EnumBeispiel {
```

Typdeklaration (muss
in der Klasse erfolgen)

```
    enum Farbe {ROT, GRUEN, GELB, BLAU};  
    enum Wochentag {Montag, Dienstag, Mittwoch, Donnerstag,  
                    Freitag, Samstag, Sonntag};
```

```
    public static void main(String[] args) {  
        Farbe f = Farbe.ROT;  
        Wochentag t = Wochentag.Mittwoch;  
        System.out.println(f + " " + t);  
        if (f == Farbe.ROT)  
            f = Farbe.GRUEN;  
        else if (f == Farbe.GRUEN)  
            f = Farbe.GELB;  
        System.out.println(f + " " + t);  
    }  
}
```

Typname muss den
Aufzählungselementen
vorangestellt werden

Arrays

- alle elementaren Datentypen hatten bisher gemeinsam
 - pro Variable konnte genau 1 Wert gespeichert werden
- sollten mehrere Werte gespeichert werden, so benötigte man mehrere Variablen
- immer pro Wert eine Variable
- Bsp.: Rätsel über boolesche Aussagen: pro Aussage eine boolesche Variable
- kommt eine Aussage hinzu, braucht man eine neue boolesche Variable
- gesucht: eine Variable, die mehrere Werte speichern kann

Arrays (Fort.)

- Feld oder Array ist ein Datentyp, der mehrere Werte eines Typs in sequentieller Folge speichern kann
- Bsp.: Array von int

2	-45	23	0	-124	7	23
---	-----	----	---	------	---	----

- dies ist ein Array von 7 int Werten (gelesen wird von links nach rechts)
- das erste Element des Felds (des Arrays) enthält den Wert 2
- das zweite Element des Felds (des Arrays) enthält den Wert -45

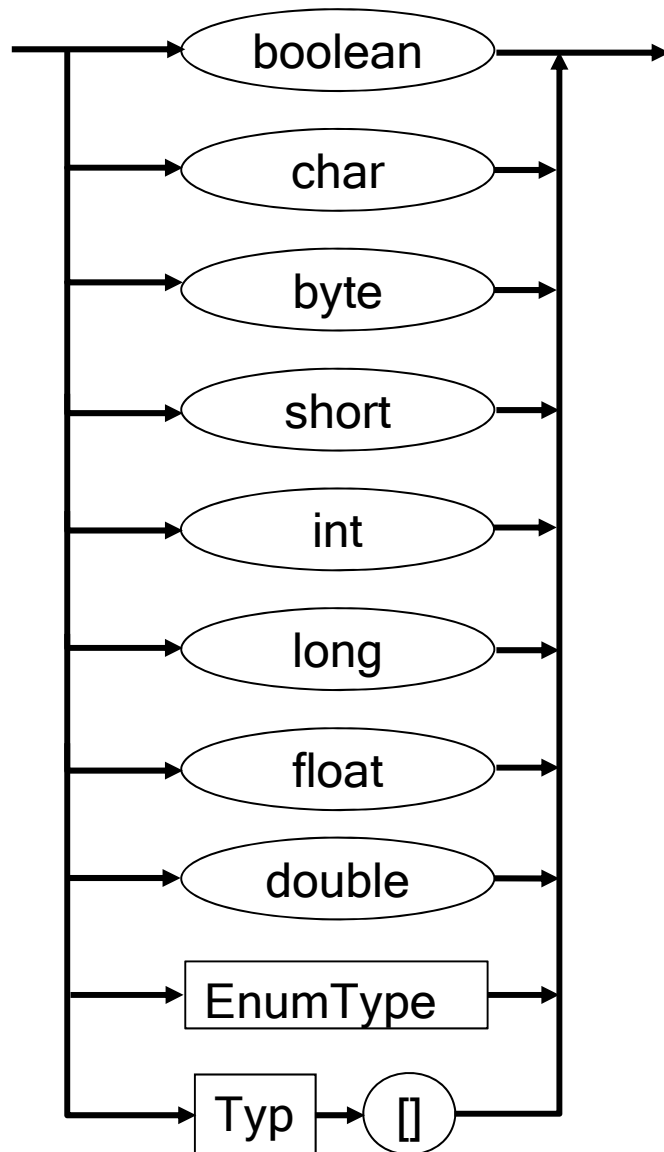
Arrays (Fort.)

- ein Arrays wird deklariert, indem hinter der Basistyp eckige Klammern geschrieben werden
- Bsp.:

<code>int[]</code>	ist ein Array von <code>int</code> Werten
<code>char[]</code>	ist ein Array von <code>char</code> Werten
<code>boolean[]</code>	ist ein Array von Wahrheitswerten
<code>float[]</code>	ist ein Array von Fließkommazahlen
...	

Typ

Arrays (Fort.)



- hat man einen Typen, wird ein Array Typ durch Hinzufügung von [] am Ende des Basistyps erzeugt

Go!

Arrays (Fort.)

- gegeben die Deklaration:

```
int [ ] i;
```

- d.h. die Variable i ist vom Typ Array von int Werten
- Frage: wieviele Werte kann in dem Array i abgespeichert werden?

```
public class Array1 {  
    public static void main(String[] args) {  
        int[] i = null;  
        System.out.println(i);  
    }  
}
```

Arrays (Fort.)

- Antwort: *gar keine*
- Begründung:
 - das Array existiert noch gar nicht
 - es muss noch erst erschaffen werden
 - die Variable `i` kann sich nur merken, wo so ein noch zu erschaffendes Array liegt
- warum legt die Deklaration
`int [] i;`
nicht gleich das Array an
- weil noch nicht feststeht, wie groß das Array ist (wieviele Einträge es hat)

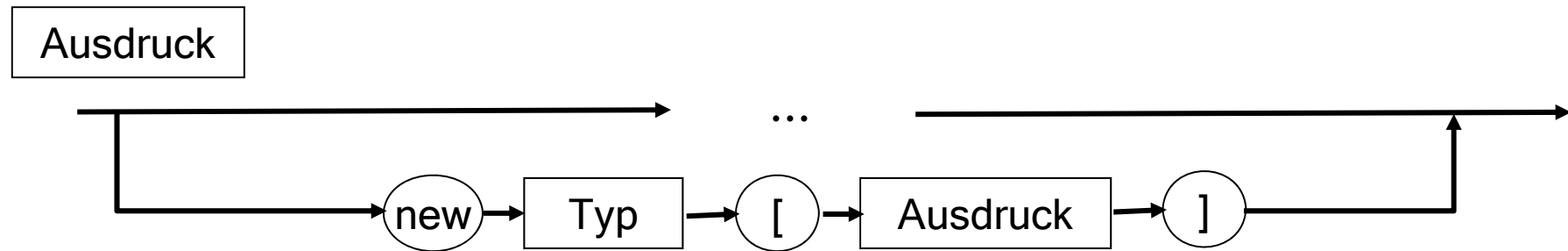
Go!

Arrays (Fort.)

- Erschaffung von einem `int` Array mit 5 Einträgen:
`new int[5]`
- dies ist ein Ausdruck und kann überall dort verwendet werden, wo Ausdrücke erlaubt sind

```
public class Array2 {  
    public static void main(String[] args) {  
        System.out.println(new int[5]);  
    }  
}
```

Arrays (Fort.)



- ein Array wird erzeugt mittels des Schlüsselwortes `new`
- danach folgt der Typ
- danach die Größe des Arrays in Klammern `[]`
- die Größe ist ein Ausdruck, der sich zu einer positiven ganzen Zahl auswerten lassen muss

Arrays (Fort.)

- der Typ des Ausdrucks `new int[5]` ist `Array` von `int`
- er kann überall dort verwendet werden, wo `Array` von `int` erwartet wird
- z.B. bei der Initialisierung einer Variable vom Typ `int-Array`

```
int [ ] i = new int [5];
```

legt ein `int-Array` an und merkt sich dieses Array in `i`

Go!

Arrays (Fort.)

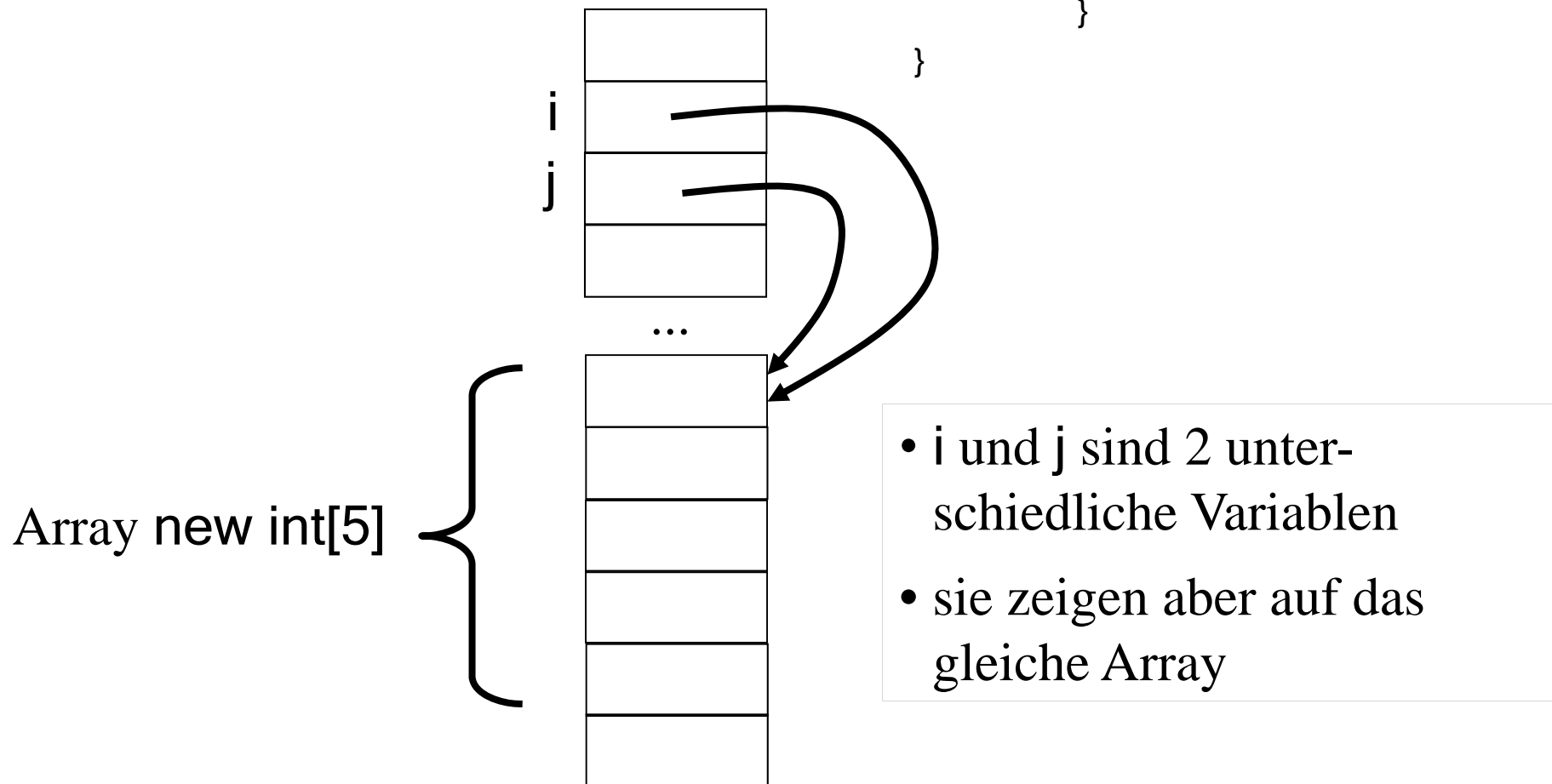
Beispiel:

```
public class Array3 {  
    public static void main(String[] args) {  
        int[] i = new int[5];           i merkt sich ein int-Array  
        System.out.println(i);  
        int[] j = i;                   j merkt sich das  
        System.out.println(j);         gleiche int-Array  
    }  
}
```

Arrays (Fort.)

```
public class Array3 {  
    public static void main(String[] args) {  
        int[] i = new int[5];  
        System.out.println(i);  
        int[] j = i;  
        System.out.println(j);  
    }  
}
```

Speicherbild von vorherigem
Programm:



Vorlesung 4/2

Arrays (Fort.)

Wie wird auf die einzelnen Elemente eines Arrays zugegriffen?

- auf die einzelnen Elemente wird mittels eines Index zugegriffen
- einem Ausdruck, der vom Typ *Array von irgendwas* ist, kann z.B. ein [3] nachgestellt werden
- dies selektiert das 4. Element des Arrays
- das Element ist vom Typ *irgendwas*
- **Vorsicht: gestartet wird bei 0 nicht bei 1**

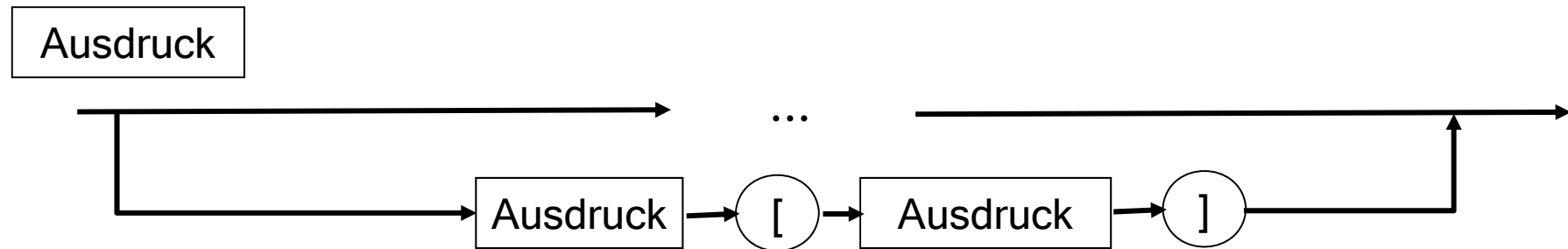
Arrays (Fort.)

32	-1	16	56765
0	1	2	3

ein int-Array mit 4 Elementen

- der 1. Eintrag hat den Index 0
- der 2. Eintrag hat den Index 1
- der 3. Eintrag hat den Index 2
- der 4. Eintrag hat den Index 3
- usw.

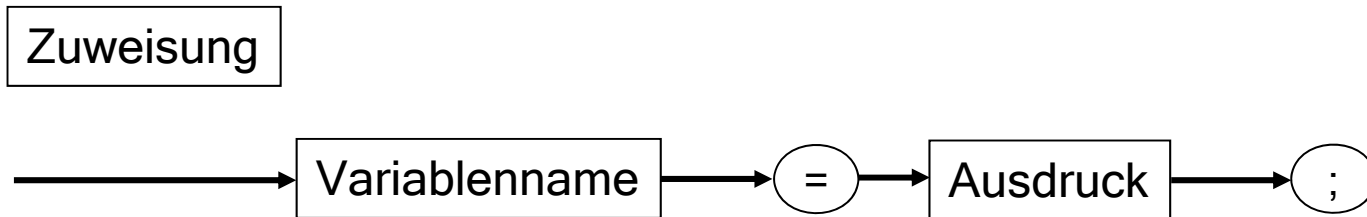
Arrays (Fort.)



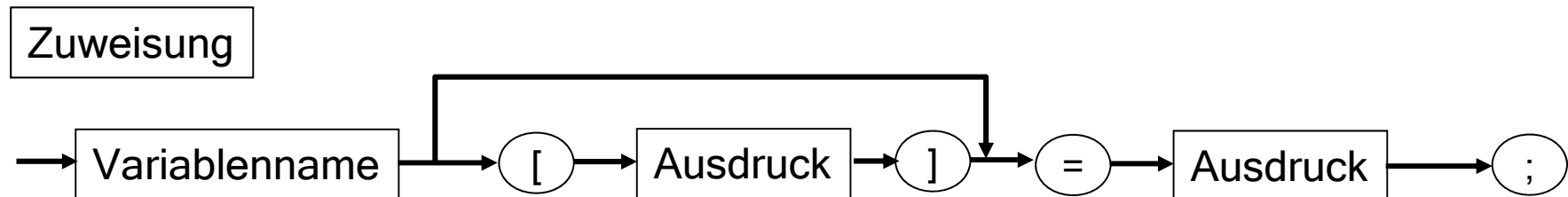
- ein neuer Ausdruck entsteht durch Selektion eines Arrays durch eine positive ganze Zahl
- der erste Ausdruck muss vom Typ Array sein, z.B. int-Array
- der zweite Ausdruck muss von integralem Typ sein
- der Typ des Ergebnisses ist der Basistyp des Arrays, z.B. int

Arrays (Fort.)

früher:



jetzt:



- wenn die Variable vom Typ Array ist, kann durch Selektion ein Element ausgewählt werden
- in einer Zuweisung wird diesem Element ein Wert zugewiesen

Go!

Beispiel

```
public class Array4 {  
    public static void main(String[] args) {  
        int[] i = new int[4];  
        i[0] = 32;  
        i[1] = -1;  
        i[2] = 16;  
        i[3] = 56765;  
        System.out.println(i[2]);  
    }  
}
```

4 Elemente
bekommen
einen Wert

legt ein int-Array
mit 4 Elementen an

der Wert des 3.
Elements wird
ausgegeben

Go!

Beispiel

```
public class Array5 {  
    public static void main(String[] args) {  
        int[] i = new int[4];  
        int[] j = i;  
        i[0] = 32;  
        i[1] = -1;  
        i[2] = 16;  
        i[3] = 56765;  
        System.out.println(i[2]);  
        j[2] = 13;  
        System.out.println(i[2]);  
    }  
}
```

legt ein int-Array
mit 4 Elementen an

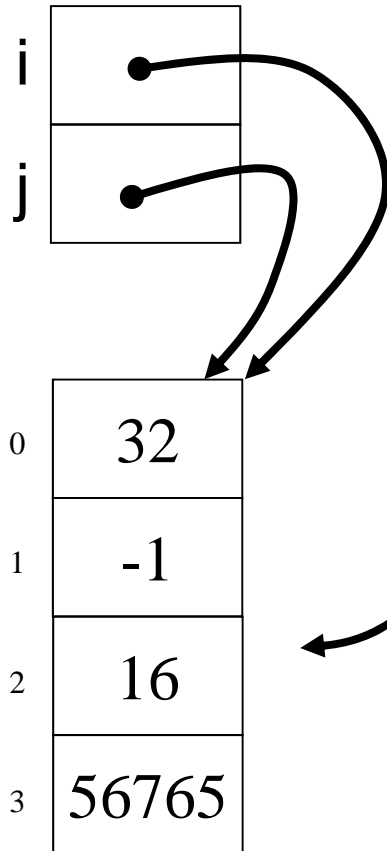
merkt sich das
Array auch in j

der Wert des 3.
Elements wird
ausgegeben

4 Elemente
bekommen
einen Wert

Was gibt dieses Programm aus?

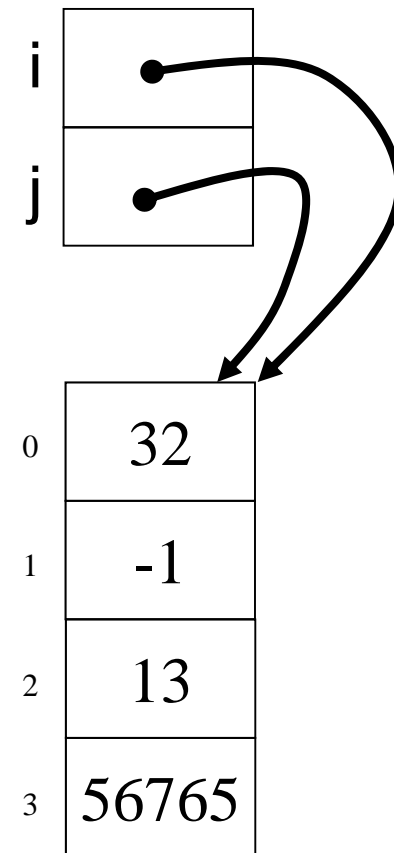
Beispiel: Erklärung



Situation vor der
ersten Print-
Anweisung

```
public class Array5 {  
    public static void main(String[] args) {  
        int[] i = new int[4];  
        int[] j = i;  
        i[0] = 32;  
        i[1] = -1;  
        i[2] = 16;  
        i[3] = 56765;  
        System.out.println(i[2]);  
        j[2] = 13;  
        System.out.println(i[2]);  
    }  
}
```

Situation nach
`j[2] = 13`



Arrays (Fort.)

- die Indizes müssen nicht immer konstante Ausdrücke sein
- sie können auch beliebige Ausdrücke sein, die zu ganzen Zahlen ausgewertet werden können

- Bsp.:

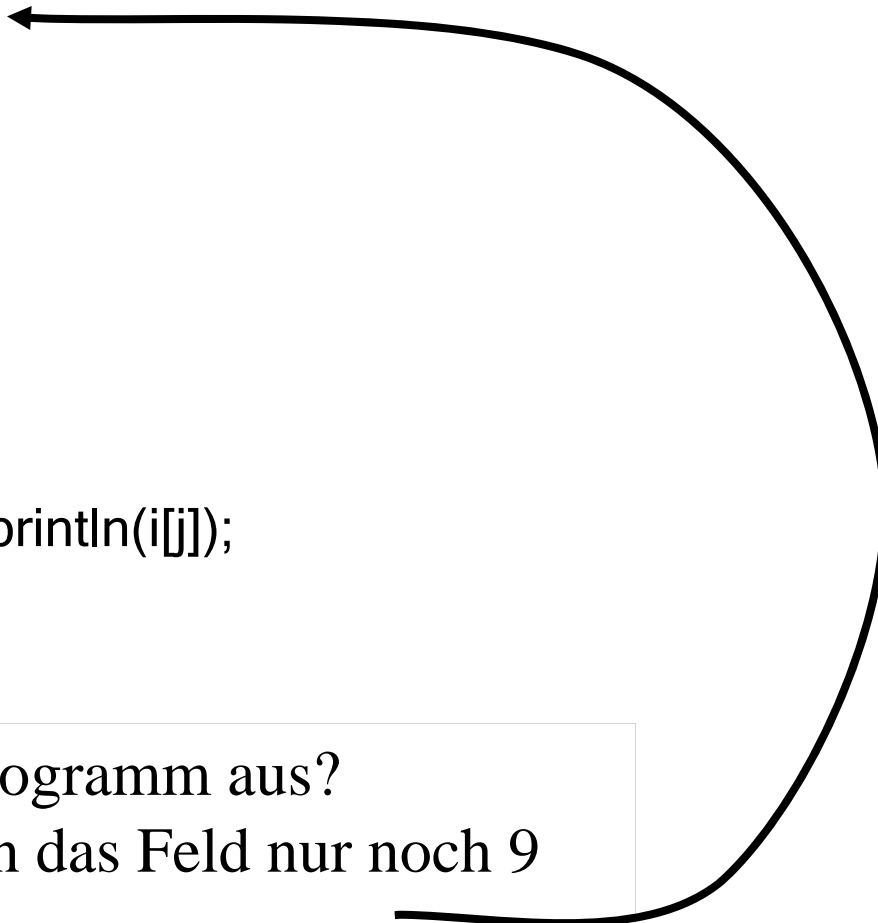
```
int[] i = new int[10];  
int j = 0;  
while(j < 10) {  
    i[j] = j*j;  
    ++j;  
}
```

- berechnet ein Feld mit 10 Einträgen mit dem jeweiligen Quadrat des Index

Go!

Beispiel

```
public class Array6 {  
    public static void main(String[] args) {  
        int[] i = new int[10];  
        int j = 0;  
        while(j < 10) {  
            i[j] = j*j;  
            ++j;  
        }  
        j = 0;  
        while(j < 10) {  
            System.out.println(i[j]);  
            ++j;  
        }  
    }  
}
```

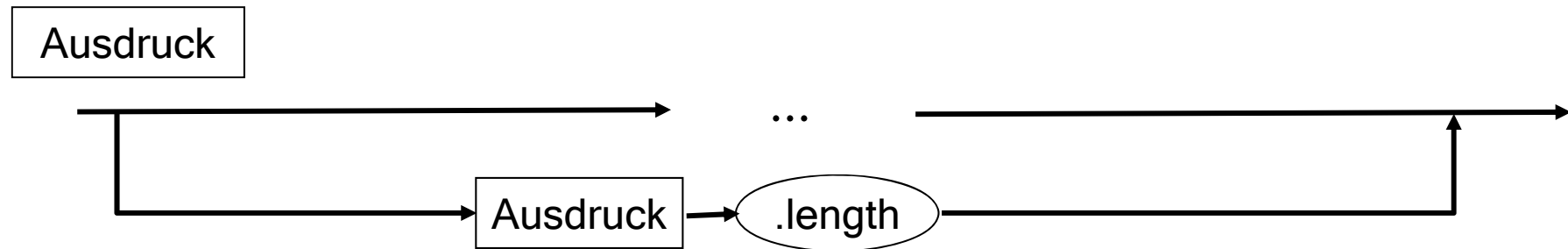


- Was gibt dieses Programm aus?
- Was passiert, wenn das Feld nur noch 9 Elemente hat?

Arrays (Fort.)

- wird auf ein Element eines Arrays zugegriffen, das außerhalb der Grenzen liegt, wird ein Fehler erzeugt
- außerhalb der Grenzen ist ein Zugriff, wenn
 - der Index negativ ist (< 0)
 - der Index größer oder gleich der Anzahl der Elemente ist (\geq Anzahl der Elemente)
- um immer sicher zu sein, wieviele Elemente in einem Array sind, kann diese Information abgefragt werden
- wenn z.B. `a` ein `int`-Array ist, so liefert `a.length` die Anzahl der Elemente

Arrays (Fort.)



- der erste Ausdruck muss vom Typ Array sein
- der Suffix `.length` macht hieraus einen Ausdruck, der von integralem Typ ist, also eine ganze Zahl
- diese Zahl ist i.A. größer 0
- sie gibt die Länge des Arrays aus der vorangestellten Ausdruck an

Go!

Beispiel

```
public class Array7 {  
    public static void main(String[] args) {  
        System.out.println(new int[8].length);  
        int[] i = new int[34];  
        System.out.println(i.length);  
    }  
}
```

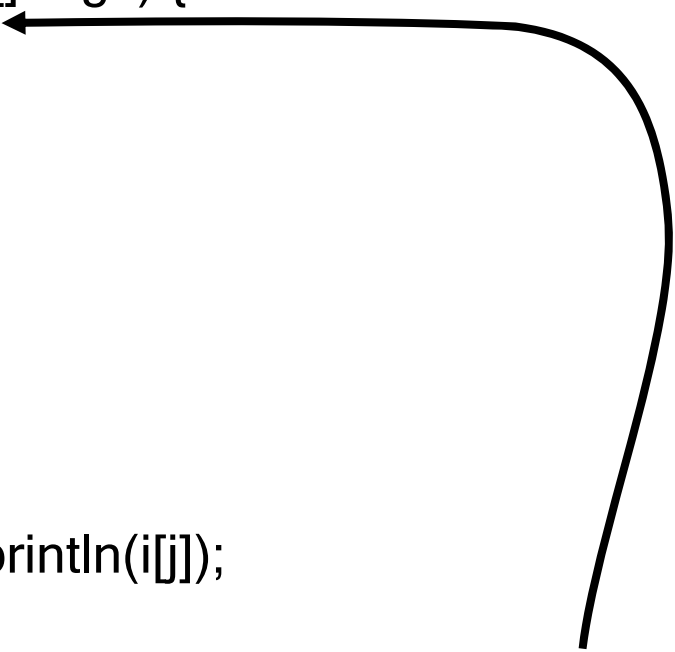
Was gibt dieses Programm aus?

Go!

Beispiel

Immer wenn die Anzahl der Elemente eines Arrays benötigt werden, den `.length` Operator verwenden

```
public class Array8 {  
    public static void main(String[] args) {  
        int[] i = new int[10];  
        int j = 0;  
        while(j < i.length) {  
            i[j] = j*j;  
            ++j;  
        }  
        j = 0;  
        while(j < i.length) {  
            System.out.println(i[j]);  
            ++j;  
        }  
    }  
}
```



Jetzt ist es kein Problem, die Größe des Arrays bei der Erzeugung zu verändern.

Go!

Arrays (Fort.)

- nach der Erzeugung eines Feldes:

```
int[] i;  
i = new int[5];
```

- Frage: welche Werte stehen in den Elementen des Feldes?

```
public class Array9 {  
    boolean[] b = new boolean[10];  
    int j = 0;  
    while(j < b.length) {  
        System.out.println(b[j]);  
        ++j;  
    }  
    int[] i = new int[5];  
    j = 0;  
    while(j < i.length) {  
        System.out.println(i[j]);  
        ++j;  
    }  
}
```

Arrays (Fort.)

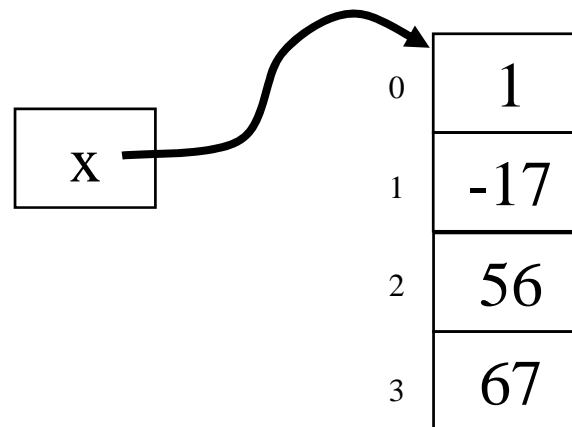
- bei der Erzeugung eines Feldes wird jedes Element für sich initialisiert
- sind die Elemente von elementaren Standardtyp, so werden die Standardwerte zugewiesen
- im Zweifelsfall sollte ein Feld durchlaufen werden und jedem Element ein Wert zugewiesen werden

Arrays (Fort.)

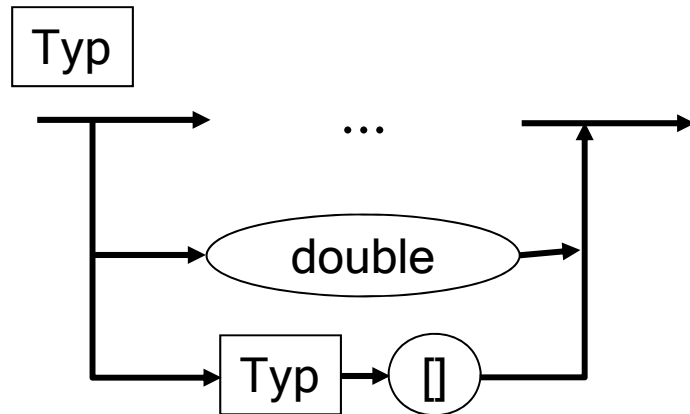
- Arrays können direkt bei ihrer Erzeugung initialisiert werden
- dazu werden in { }-Klammern die Werte des Arrays direkt angegeben
- Bsp.:

```
int[ ] x = {1,-17,56,67};
```

- legt ein int-Array mit 4 Elementen an und speichert dieses unter der Variablen x



Arrays (Fort.)



- aus einem beliebigen Typen kann ein Array-Typ erzeugt werden durch Anfügung von []
- dies ist dann ein neuer Typ
- somit kann auch diesem neuen Typ wieder ein [] angefügt werden, um einen neuen Array-Typ zu erzeugen

Arrays (Fort.)

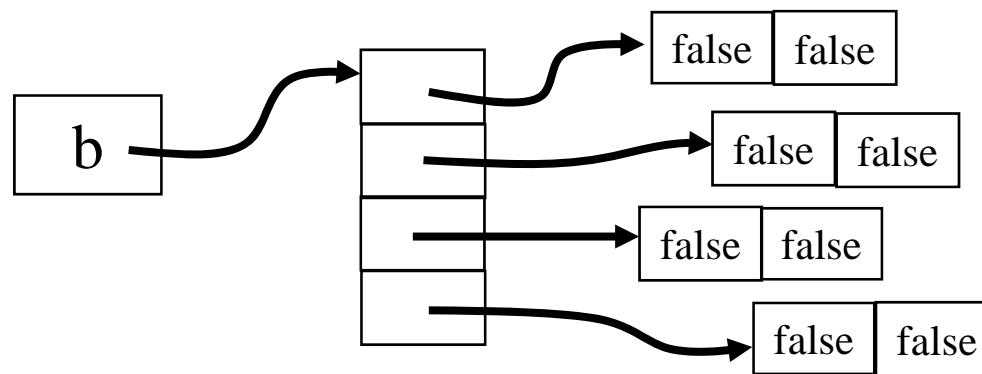
- Bsp.:

```
int [][] x;
```

- dies ist ein Array, bei dem jedes Element wieder ein int-Array ist
- dadurch erhält man ein „2-dimensionales“ Array
- Bsp.:

```
boolean [][] b = new boolean[4][2];
```

legt ein Array mit 4 Einträgen an; jeder Eintrag ist ein boolean Array mit 2 Einträgen



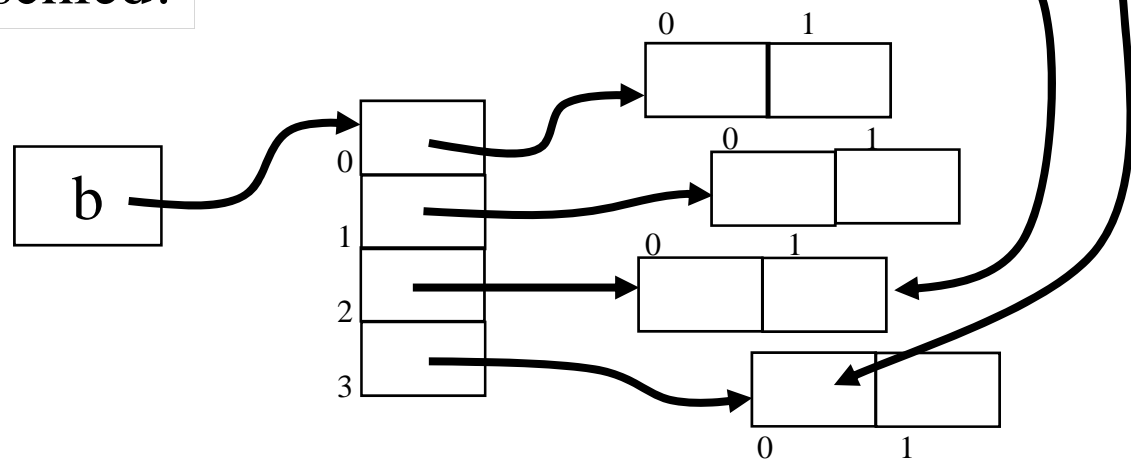
Arrays (Fort.)

- der Zugriff auf mehr-dimensionale Arrays erfolgt nacheinander durch die einzelnen Dimensionen
- bei diesem 2-dimensionalen booleschen Arrays

```
boolean [][] b = new boolean[4][2];
```

- selektiert `b[3][0]` diese *boolesche Variable*
- selektiert `b[2]` dieses *boolesche Array*

Wichtiger Unterschied!



Go!

Beispiel

```
public class Array11 {  
    public static void main(String[] args) {  
        boolean[][] b = new boolean[4][2];  
        System.out.println(b.length);  
        System.out.println(b[2].length);  
        System.out.println(b[0].length);  
    }  
}
```

Was gibt dieses Programm aus?

Arrays (Fort.)

- mehrdimensionale Felder können bei der Deklaration auch direkt initialisiert werden
- dazu werden in { }-Klammern die Werte angegeben
- Bsp.:

```
int[][] a = {{2,-1},{34,17},{2,0}};
```

legt ein 2-dimensionales integer Array an; die 1. Dimension enthält 3 Arrays, von dem jedes wiederum 2 Integer Einträge enthält

Go!

Beispiel

```
public class Array12 {  
    public static void main(String[] args) {  
        int[][] a = {{2,-1},{34,17},{2,0}};  
        int i = 0;  
        while(i < a.length) {  
            int j = 0;  
            while(j < a[i].length) {  
                System.out.print(a[i][j]);  
                System.out.print("\t");  
                ++j;  
            }  
            System.out.println();  
            ++i;  
        }  
    }  
}
```

Was gibt dieses Programm aus?

Arrays (Fort.)

- bei mehrdimensionalen Arrays müssen die Unterarrays nicht alle gleichgroß sein
- Bsp.:

```
int[][] a = {{2,-1}, {34,17,56,102}, {2,0,5}, {2} };
```

- ist eine gültige Deklaration
- es wird ein Array mit 4 Einträgen angelegt
- der 1. Eintrag ist ein int-Array mit 2 Einträgen
- der 2. Eintrag ist ein int-Array mit 4 Einträgen
- der 3. Eintrag ist ein int-Array mit 3 Einträgen
- der 4. Eintrag ist ein int-Array mit 1 Eintrag

Go!

Beispiel

```
public class Array13 {  
    public static void main(String[] args) {  
        int[][] a = {{2,-1}, {34,17,56,102}, {2,0,5}, {2} };  
        System.out.println(a.length);  
        System.out.println(a[0].length);  
        System.out.println(a[1].length);  
        System.out.println(a[2].length);  
        System.out.println(a[3].length);  
        int i = 0;  
        while(i < a.length) {  
            int j = 0;  
            while(j < a[i].length) {  
                System.out.print(a[i][j]);  
                System.out.print("\t");  
                ++j;  
            }  
            System.out.println();  
            ++i;  
        }  
    }  
}
```

Was gibt dieses Programm aus?

Arrays (Fort.)

- mehrdimensionale Arrays können auch nacheinander erschaffen werden
- Bsp.: `int[][] a = new int[4][];`
 - erschafft erst einmal ein Array mit 4 Einträgen, bei dem jeder Eintrag wieder ein int-Array ist
 - diese int-Arrays existieren aber noch nicht
 - sie müssen noch erst erschaffen werden
 - Bsp.: `a[2] = new int[34];`

Go!

Beispiel

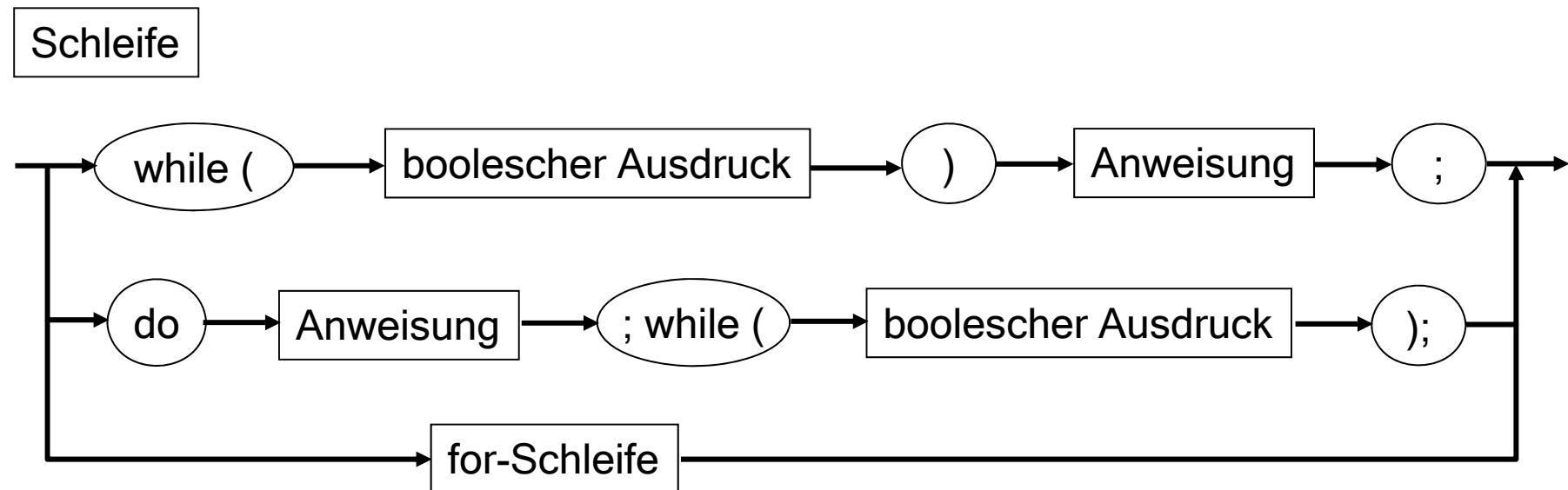
```
public class Array14 {  
    public static void main(String[] args) {  
        int[][] a = new int[4][]; ← noch nicht alle  
        int i = 0;                               Subarrays  
        while(i < a.length) {  
            a[i] = new int[i*i+1]; ← Subarrays werden  
            int j = 0;                               erschaffen ...  
            while(j < a[i].length) {  
                a[i][j] = i * j; ←  
                ++j;                               ... und initialisiert  
            }  
            ++i;  
        }  
        i = 0;  
        while(i < a.length) {  
            int j = 0;  
            while(j < a[i].length) {  
                System.out.print(a[i][j]);  
                System.out.print("\t");  
                ++j;  
            }  
            System.out.println();  
            ++i;  
        }  
    }  
}
```

Was gibt dieses Programm aus?

Vorlesung 5/1

Schleifen

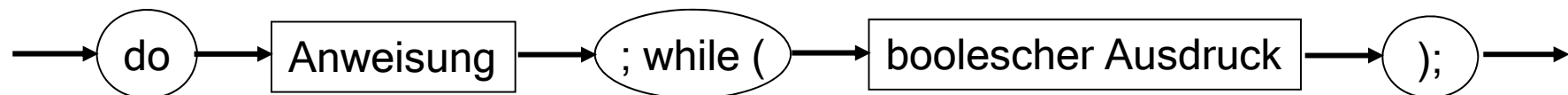
- neben der **while**-Schleife gibt es noch 2 andere Schleifen
- die **do**-Schleife
- die **for**-Schleife



do-Schleife

- die Anweisung wird in jedem Fall ausgeführt
- danach wird der boolesche Ausdruck ausgewertet
- ist das Ergebnis **true**, so wird die Anweisung nochmal ausgeführt, solange bis der boolesche Ausdruck **false** ergibt
- ist das Ergebnis **false**, so wird mit der Anweisung nach der do-Schleife weitergemacht

...



do-Schleife (Fort.)

...

- die do-Schleife ist im Grunde genommen wie die while-Schleife, jedoch mit dem Unterschied, dass der Schleifenrumpf mindestens 1-mal ausgeführt wird
- bei der while-Schleife wird der Schleifenrumpf u.U. gar nicht ausgeführt (wenn die Bedingung von Anfang an false ist)
- eine do-Schleife kann durch eine while-Schleife wie folgt ausgedrückt werden (die Anweisung wird verdoppelt):

```
do  
    anweisung;  
while (bedingung);
```

=

```
anweisung;  
while (bedingung)  
    anweisung;
```

do-Schleife (Fort.)

...

- aber auch die while-Schleife kann durch die do-Schleife ausgedrückt werden:

```
while (bedingung)  
    anweisung;
```

=

```
do  
    if (bedingung)  
        anweisung;  
while (bedingung);
```

- Beobachtung: das gegenseitige Ausdrücken ist *nie elegant*
- daher: genau überlegen, welche Fall vorliegt
- in der Praxis: do-Schleifen kommen selten vor

Schleifen (Fort.)

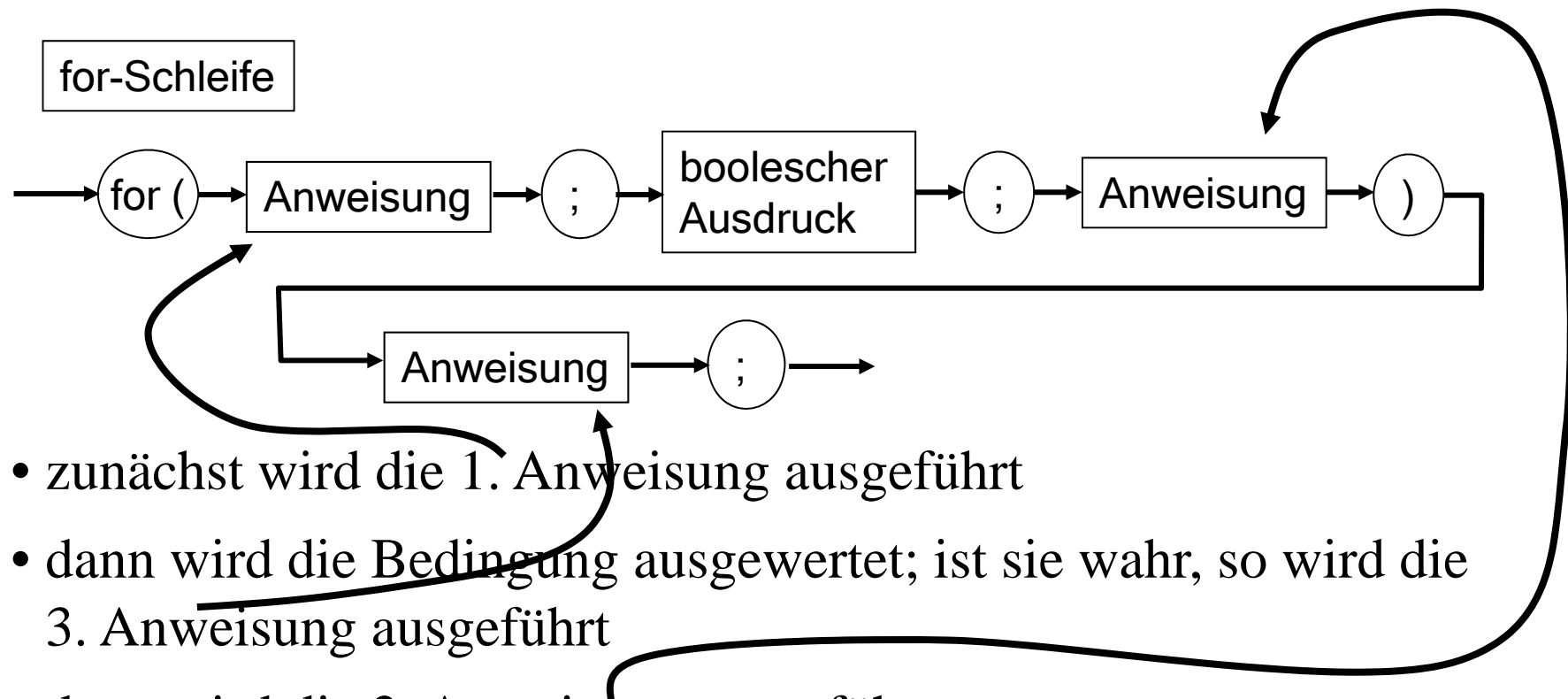
- die folgende Situation kommt sehr häufig vor:

```
anweisung; // Initialisierung
while (bedingung) {
    anweisung;    // allgemeiner Schleifenrumpf
    anweisung;    // verändert die Bedingung
}
```

- zunächst werden Variablen initialisiert
- dann wird die Bedingung überprüft
- ist die Bedingung wahr, wird der Schleifenrumpf ausgeführt
- am Ende des Schleifenrumpfes werden die Variablen der Bedingung (teilweise) geändert
- das wiederholt sich solange, bis die Bedingung nicht mehr gilt

for-Schleife

- diese Situation trägt die for-Schleife Rechnung



- zunächst wird die 1. Anweisung ausgeführt
- dann wird die Bedingung ausgewertet; ist sie wahr, so wird die 3. Anweisung ausgeführt
- dann wird die 2. Anweisung ausgeführt
- die letzten beiden Schritte werden solange wiederholt, bis die Bedingung falsch ist

Go!

Beispiel

```
public class Schleife4 {  
    public static void main(String[] args) {  
        int i = 1;  
        int j;  
        for(j = 2; j < 10; ++j) {  
            i = i*j;  
        }  
        System.out.println(i);  
    }  
}
```

der Variablen-
update

die Abbruch-
bedingung

die Initialisierung

for-Schleife (Fort.)

Das Programm `Integer1` (siehe unten) zeigt eine typische Struktur:

1. ein Zähler wird deklariert und initialisiert
2. eine Schleife wird durchlaufen, bis der Zähler einen bestimmten Wert erreicht hat
3. am Ende der Schleife wird der Zähler verändert

Solche Schleifen werden mit der `for`-Schleife modelliert

```
public class Integer1 {  
    public static void main(String[] args) {  
        byte i;  
        for(i = 0; i != -10; ++i) {  
            System.out.println(i);  
        }  
    }  
}
```

for-Schleife (Fort.)

- wird der Zähler nach der Schleife nicht mehr benötigt, so wird die Deklaration oft in die Schleife verschoben
- dies hat den Vorteil, dass nach der Schleife der Zähler nicht aus Versehen wieder benutzt wird

```
public class Integer1 {  
    public static void main(String[] args) {  
        for(byte i = 0; i != -10; ++i) {  
            System.out.println(i);  
        }  
        // hier ist i nicht mehr sichtbar  
    }  
}
```


Schleifen und Arrays

Die folgende Programme ist typisch für Schleifen und Arrays:

- for-Schleifen iterierten über eine Dimension von Arrays
- die Iteration erfolgte von 0 bis length-1
- es wurde eine neue Laufvariable angelegt, die nach der Schleife nicht mehr benötigt wird
- es wird *nur lesend* auf die Elemente zugegriffen

```
public class Schleife5 {  
    public static void main(String[] args) {  
        char[] a = {'b','?','4',':','X'};  
        for(int i = 0; i < a.length; ++i)    lesende Iteration  
            System.out.print(a[i] + " ");    über a  
        System.out.println();  
    }  
}
```

Go!

Schleifen und Arrays (Forts.)

- Diese Situation kann mit der erweiterten for-Schleife kürzer geschrieben werden
- statt `for(int i = 0; i < a.length; ++i) ... a[i]` wird geschrieben
- `for(char c : a) ... c`
- hierbei bezeichnet `c` schon das Element `a[i]`
- daher ist ein schreibender Zugriff nicht möglich

```
public class Schleife6 {  
    public static void main(String[] args) {  
        char[] a = {'b','?','4',':','X'};  
        for(char xyz : a)  
            System.out.print(xyz + " ");  
        System.out.println();  
    }  
}
```

lesende Iteration über `a`, `c` ist
jeweils ein Element aus `a`

Nochmal: Verzweigungen

- manchmal sollen in Abhängigkeit des Wertes einer Integervariable verschiedene Anweisungen ausgeführt werden
- dies kann man durch eine geschaltete **if-then-else** Struktur realisieren

```
public class Integer3 {  
    public static void main(String[] args) {  
        int i = 10;  
        if (i == 0) {  
            System.out.println("i ist 0");  
        } else if (i == 2) {  
            System.out.println("i ist 2");  
        } else if (i == 5) {  
            System.out.println("i ist 5");  
        } else {  
            System.out.println("i ist irgendwas anderes");  
        }  
    }  
}
```

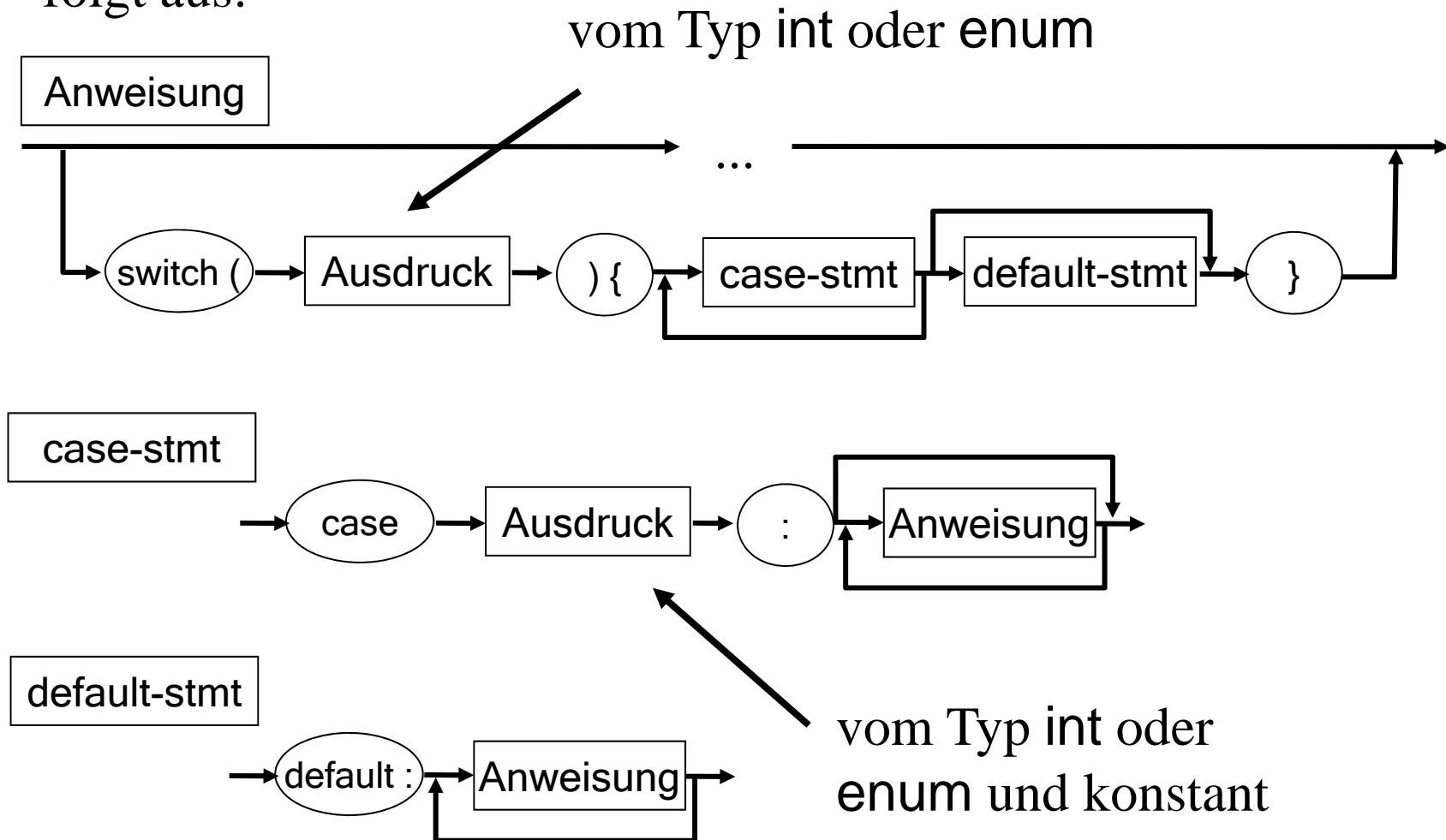
Nochmal: Verzweigungen (Fort.)

- solche geschachtelten **if-then-else** Anweisungen kann man auch durch eine sogenannte **switch**-Anweisung realisieren.

```
public class Integer3 {  
    public static void main(String[] args) {  
        int i = 10;  
        switch (i) {  
            case 0: System.out.println("i ist 0");  
                break;  
            case 2: System.out.println("i ist 2");  
                break;  
            case 5: System.out.println("i ist 5");  
                break;  
            default:  
                System.out.println("i ist irgendwas anderes");  
        }  
    }  
}
```

Nochmal: Verzweigungen (Fort.)

- die allgemeine Struktur von **switch** Anweisungen sieht wie folgt aus:



Nochmal: Verzweigungen (Fort.)

- das Verhalten einer **switch**-Anweisung ist wie folgt:
- der **switch**-Ausdruck wird ausgewertet und von oben nach unten mit den **case**-Ausdrücken verglichen
- stimmt einer überein, so werden dann *alle* folgenden Anweisungen ausgeführt
- dies kann nur durch eine **break** Anweisung unterbrochen werden



Go!

Nochmal: Verzweigungen (Fort.) Beispiel

```
public class Integer3 {  
    public static void main(String[] args) {  
        int i = 10;  
        switch (i - 6) {  
            case 0: System.out.println("i ist 0");  
                break;  
            case 2 + 2: System.out.println("i ist 2");  
                // break;  
            case 5: System.out.println("i ist 5");  
                break;  
            default:  
                System.out.println("i ist irgendwas anderes");  
        }  
    }  
}
```

beliebige int
oder enum
Ausdrücke (im
case zusätzlich
konstant)

hier fehlt
das break

Bitweise Operatoren

- Es besteht manchmal der Wunsch, auf die einzelnen Bits einer integralen Zahl direkt zugreifen zu können (lesend und schreibend)

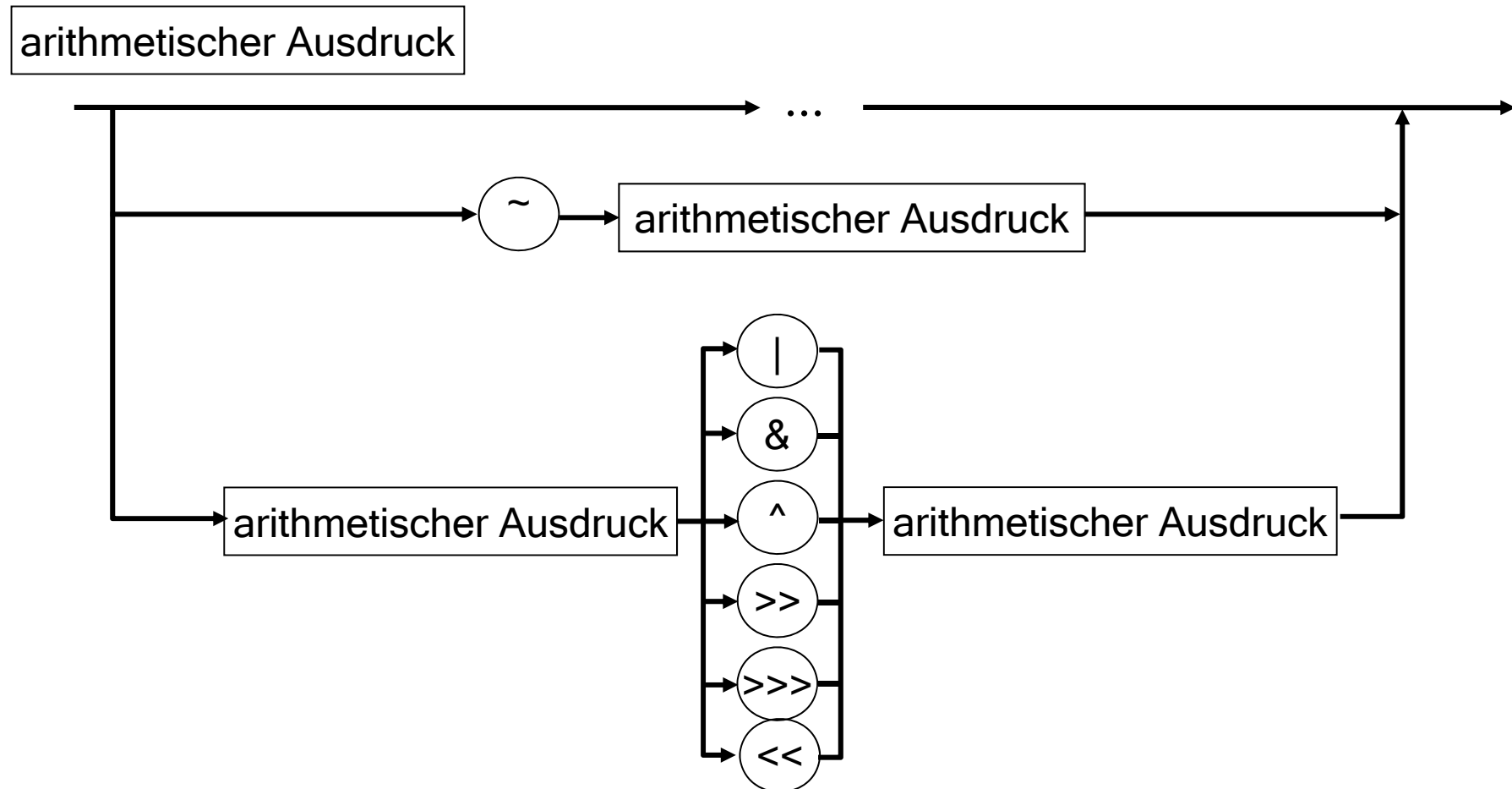
- Beispiel: int bestehend aus 32 Bits

0 ... 1	1 ... 0	0 ... 0	1 ... 0
31	24	16	8
			0

- dazu gibt es die Operatoren:

~	Einerkomplement, alle Bits werden invertiert
	bitweises Oder beider Operanden
&	bitweises Und beider Operanden
^	bitweises exklusives Oder beider Operanden
>>	Rechtssshift mit Vorzeichen
>>>	Rechtssshift ohne Vorzeichen
<<	Linkssshift

Bitweise Operatoren (Fort.)



Bitweise Operatoren (Fort.)

- Rechts- und Linksshift dienen zur schnellen Multiplikation und Division mit bzw. durch Zweierpotenzen
- Linksshift (immer ohne Vorzeichen): alle Bits werden um die angegebene Anzahl an Bits nach links verschoben. Von rechts wird immer eine 0 reingeschoben.

$01011_2 \ll 1$ wird zu 10110_2 (d.h. $11_{10} \ll 1 = 22_{10} \equiv 11_{10} * 2 = 22_{10}$)

$00101_2 \ll 2$ wird zu 10100_2 (d.h. $5_{10} \ll 2 = 20_{10} \equiv 5_{10} * 4 = 20_{10}$)

...

Bitweise Operatoren (Fort.)

- Rechtsshift ohne Vorzeichen: alle Bits werden um die angegebene Anzahl an Bits nach rechts verschoben. Von links wird immer eine 0 reingeschoben.

$01011_2 \ggg 2$ wird zu 00010_2 (d.h. $11_{10} \ggg 2 = 2_{10} \equiv 11_{10}/4_{10}=2_{10}$)

$10001_2 \ggg 2$ wird zu 00100_2 (d.h. $17_{10} \ggg 2 = 4_{10} \equiv 17_{10}/4_{10}=4_{10}$)

- Rechtsshift mit Vorzeichen: alle Bits werden um die angegebene Anzahl an Bits nach rechts verschoben. Ist das höchstwertige Bit 1 so wird von links immer 1 reingeschoben, sonst 0

$01011_2 \gg 2$ wird zu 00010_2 (d.h. $11_{10} \gg 2 = 2_{10} \equiv 11_{10}/4_{10}=2_{10}$)

$10001_2 \gg 2$ wird zu 11100_2 (d.h. $-15_{10} \gg 2 = -4_{10} \equiv -15_{10}/4_{10}=-4_{10}$)

Bitweise Operatoren (Fort.)

- Einerkomplement, bitweises Oder, Und, Exklusives Oder dienen für Mengenoperationen
- eine int Zahl m kann als Teilmenge $M \subseteq \{0,1,\dots,31\}$ verstanden werden, mit i -tes Bit ist in m gesetzt $\Leftrightarrow i \in M$ gilt
- Beispiel: `int m = 14; // m = 0...01110` entspricht $M = \{1,2,3\}$
31 4 3 2 1 0
- Einerkomplement: alle Bits werden invertiert
 $\sim m$: $\sim 0...01110$ wird $1...10001$ entspricht $\sim M = \{0,4,5,\dots,31\}$
d.h $\sim M \equiv \{0,\dots,31\} \setminus M$
- ...

Bitweise Operatoren (Fort.)

- ...
- Bitweises Oder: jeweiligen Bits der beiden Zahlen werden mittels Oder verknüpft
 $01011_2 \mid 10001_2$ wird 11011_2
dies entspricht der Mengenvereinigung
- Bitweises Und: jeweiligen Bits der beiden Zahlen werden mittels Und verknüpft
 $01011_2 \& 10001_2$ wird 00001_2
dies entspricht dem Mengendurchschnitt
- Bitweises exklusiv Oder: jeweiligen Bits der beiden Zahlen werden mittels exklusiven Oder verknüpft
 $01011_2 \wedge 10001_2$ wird 11010_2
dies entspricht $m1 \wedge m2 \equiv M1 \cup M2 \setminus M1 \cap M2$

Bitweise Operatoren (Fort.)

diese Operatoren können dazu verwendet werden, bestimmte Bitpositionen zu testen:

- $1 \ll 0$ ist die 0te Bitposition
- $1 \ll 1$ ist die 1te Bitposition
- $1 \ll 2$ ist die 2te Bitposition
- $m \& (1 \ll 2)$ ist m , wobei alle bis auf die 2te Bitposition ausgeblendet sind
- $(m \& (1 \ll 2)) \neq 0$ ist genau dann wahr, wenn die 2te Bitposition von m 1 ist

Vorlesung 5/2

Go!

Beispiel

```
public class Array10 {  
    public static void main(String[] args) {  
        boolean[] b = new boolean[4];  
        for(int i = 0; i < b.length; ++i)  
            b[i] = false;  
        int N = 1 << b.length;  
        for(int i = 0; i < N; ++i) {  
            for(int j = 0; j < b.length; ++j) {  
                b[j] = (i & (1 << j)) != 0;  
            }  
            for(int j = 0; j < b.length; ++j) {  
                System.out.print(b[b.length-j-1]+"\\t");  
            }  
            System.out.println();  
        }  
    }  
}
```

Initialisierung

$N = 2^{b.length}$

ist die j-te Position
von i ungleich 0?

Ausgabe

Was gibt dieses Programm aus?

Go!

Beispiel

```
public class AllBoolValue4 {  
    public static void main(String[] args) {  
        int m1 = 1<<1 | 1<<11 | 1<<17;    // m1 = {1,11,17}  
        int m2 = 1<<1 | 1<<2 | 1<<11;      // m2 = {1,2,11}  
        int m3 = m1 | m2;                   // m3 = m1 vereinigt m2  
        int m4 = m1 & m2;                   // m4 = m1 geschnitten m2  
        if ((m3 & (1<<17)) != 0)            // ist 17 Element von m3  
            System.out.println("17 ist Element von m3");  
        else  
            System.out.println("17 ist kein Element von m3");  
        if ((m4 & (1<<17)) != 0)            // ist 17 Element von m4  
            System.out.println("17 ist Element von m4");  
        else  
            System.out.println("17 ist kein Element von m4");  
    }  
}
```

Bitweise Operatoren (Fort.)

diese Operatoren können auch dazu verwendet werden, maximale Zahlen zu erzeugen:

- ~ 0 ist die Zahl, bei der alle Bits 1 sind
- diese Zahl ist die -1
- $-1 \ggg 1$ erzeugt eine Zahl, bei der alle Bits bis auf das ganz linke 1 ist
- diese Zahl ist die größte Zahl

Go!

Beispiel

```
public class MaxValue {  
    public static void main(String[] args) {  
        int i = ~0; ← erzeugt die -1  
        System.out.println(i);  
        i = i >>> 1; ← erzeugt die maximale Zahl  
        System.out.println(i);  
        i = i+1; ← maximale Zahl +1 ergibt  
                  die kleinste Zahl  
        System.out.println(i);  
        System.out.println(1 << 31); die kleinste Zahl kann auch  
                                     so erzeugt werden  
        System.out.println(1 << 63);  
        System.out.println(1L << 63);  
    }  
}
```

Wo ist hier der
Unterschied?

Methoden

Aufgabe:

- das folgende Rechteck soll 3-mal auf dem Bildschirm gedruckt werden

```
+-----+  
|       |  
|       |  
|       |  
|       |  
+-----+
```

Lösung 1:

- die println Anweisungen für ein Rechteck zusammenstellen
- alle Anweisungen kopieren

Lösung 2:

- die println Anweisungen für ein Rechteck in einer Schleife stellen

Lösungen

```

public class Rechteck1 {
    public static void main(String[] args) {
        System.out.println("+-----+");
        System.out.println("|           |");
        System.out.println("|           |");
        System.out.println("|           |");
        System.out.println("|           |");
        System.out.println("+-----+");
        System.out.println("+-----+");
        System.out.println("|           |");
        System.out.println("|           |");
        System.out.println("|           |");
        System.out.println("|           |");
        System.out.println("+-----+");
        System.out.println("+-----+");
        System.out.println("|           |");
        System.out.println("|           |");
        System.out.println("|           |");
        System.out.println("|           |");
        System.out.println("+-----+");
    }
}

```

```

public class Rechteck2 {
    public static void main(String[] args) {
        for(int i = 0; i < 3; ++i) {
            System.out.println("+-----+");
            System.out.println("|           |");
            System.out.println("|           |");
            System.out.println("|           |");
            System.out.println("|           |");
            System.out.println("+-----+");
        }
    }
}

```

Methoden (Fort.)

- diese Lösungen haben beide Nachteile:
- 1. Lösung: soll sich das Rechteck ändern, so ist diese Änderung in allen 3 Rechtecken nachzuvollziehen
 - ⇒ viel Arbeit
 - ⇒ fehleranfällig
 - ⇒ irgendwo wird eine Änderung vergessen
- 2. Lösung: soll zwischen den Rechtecken noch etwas anderes ausgegeben werden, muss es mit in die Schleife genommen werden
 - ⇒ es wird in jedem Schleifenschritt ausgegeben
 - ⇒ individuelle Ausgaben sind nur mit sehr viel Aufwand möglich

Methoden (Fort.)

gewünschte Lösung:

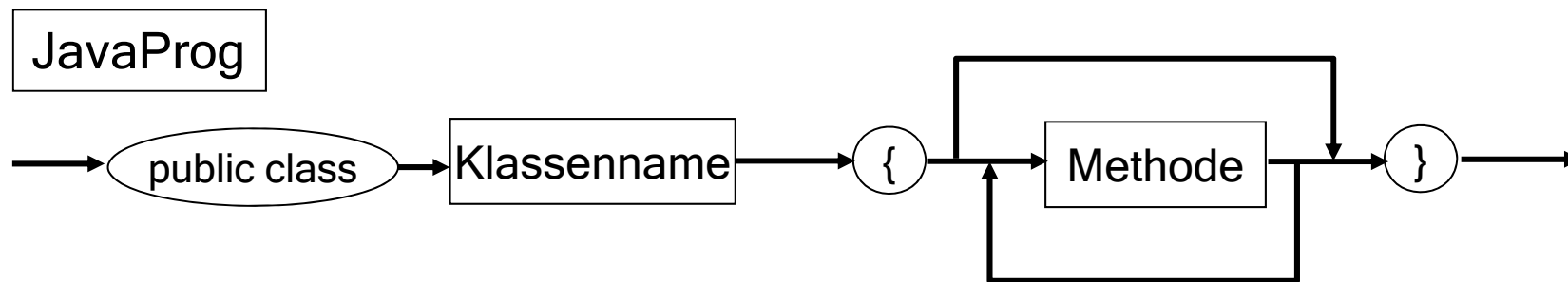
- eine Kapselung der Funktionalität "drucke Rechteck"
- diese Funktionalität kann dann immer wieder ausgeführt werden, wenn der Wunsch existiert

Konzept in der Programmierung:

- Funktionen und Prozeduren (klassische imperative Programmiersprachen)
- Methoden (objektorientierten Programmiersprachen)

Methoden (Fort.)

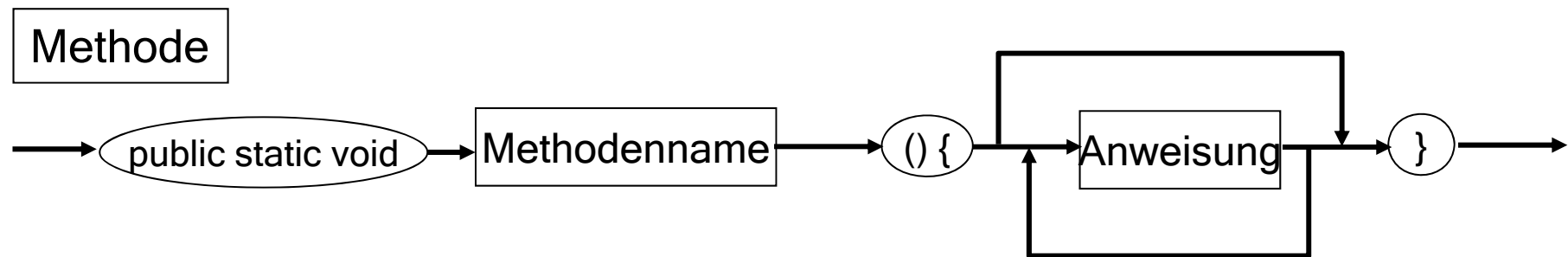
- Methoden müssen deklariert werden (wie sehen sie aus, was machen sie?)
- Methoden können dann aufgerufen werden (sie werden abgearbeitet)



- ein Java Programm besteht (zunächst) aus einer Klasse, in der mehrere Methoden deklariert sind (eine muss davon main heißen)

Methoden (Fort.)

- Methoden Deklaration:



- eine Methode fängt (zunächst) mit den Worten **public static void** an
- danach folgt ein selbstgewählter Methodenname
- dann kommen (zunächst) runde Klammern **()**
- in geschweiften Klammern **{ }** kommt dann eine Sequenz von Anweisungen

Go!

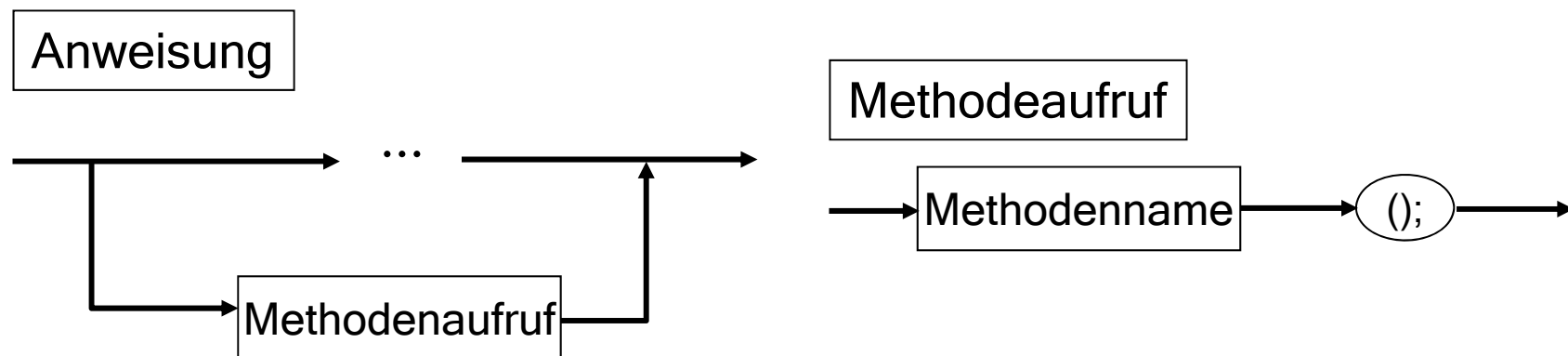
Beispiel

```
public class Rechteck3 {  
    public static void print_rechteck() {  
        System.out.println("+-----+");  
        System.out.println("|           |");  
        System.out.println("|           |");  
        System.out.println("|           |");  
        System.out.println("|           |");  
        System.out.println("+-----+");  
    }  
  
    public static void main(String[] args) {  
    }  
}
```

Was gibt dieses Programm aus?

Methodenaufruf

- das vorherige Programm hat nichts ausgegeben, weil:
 - ein Programm in der (einzigen) main-Methode startet
 - die Anweisungen in einer Methode nur dann ausgeführt werden, wenn diese Methode aufgerufen wird
 - in der main-Methode des vorherigen Programms nicht die Methode `print_rechteck` aufgerufen wurde



Go!

Beispiel

```
public class Rechteck4 {  
    public static void print_rechteck() {  
        System.out.println("+-----+");  
        System.out.println("|           |");  
        System.out.println("|           |");  
        System.out.println("|           |");  
        System.out.println("|           |");  
        System.out.println("+-----+");  
    }  
  
    public static void main(String[] args) {  
        print_rechteck();  
        print_rechteck();    die Methode print_rechteck  
        print_rechteck();    wird 3-mal aufgerufen  
    }  
}
```

Was gibt dieses Programm aus?

Methodenaufruf (Fort.)

- jetzt kann zum einen das Rechteck an genau einer Stelle geändert werden
in der Methode `print_rechteck`
- soll zwischen der Ausgabe der einzelnen Rechtecke noch eine weitere individuelle Ausgabe erfolgen, kann man das zwischen den Methodenaufrufen in der `main-Methode` machen

Go!

Beispiel

```
public class Rechteck5 {  
    public static void print_rechteck() {  
        System.out.println("+-----+");  
        System.out.println("|           |");  
        System.out.println("|    **    |");  
        System.out.println("|    **    |");  
        System.out.println("|           |");  
        System.out.println("+-----+");  
    }  
}
```

Änderungen am
Rechteck müssen
nur einmal gemacht
werden

```
public static void main(String[] args) {  
    print_rechteck();  
    System.out.println("dies war das 1. Rechteck");  
    print_rechteck();  
    System.out.println("gleich kommt das Letzte");  
    print_rechteck();  
}  
}
```

zwischen den
Methodenaufrufen
können andere
Anweisungen
erfolgen

Was gibt dieses Programm aus?

Methoden und Variablen

- eine Methode ist ein in sich abgeschlossener Block
- somit gelten die Sichtbarkeitsregeln für Variablen wie in Blöcken
- d.h. Variablen, die innerhalb einer Methode deklariert sind,
 - entstehen mit dem Methodenaufruf
 - verschwinden, nachdem die Methode abgearbeitet ist
- Variablen, die in einer Methode deklariert sind, sind in einer anderen Methode nicht sichtbar

Methoden und Variablen (Fort.)

Negativbeispiel:

```
public static void print_rechteck() {  
    boolean a = true;  
    System.out.println(a);  
}
```

... verschwindet hier

a erscheint hier ...

... kann hier verwendet werden ...

```
public static void main(String[] args) {  
    print_rechteck();  
    a = false;  
}
```

das **a** ist hier nicht definiert,
obwohl `print_rechteck`
zuvor ausgeführt wurde

Go!

Methoden und Variablen (Fort.)

- nach einer Methode verschwinden die Variable, die in einer Methode deklariert werden
- somit ist auch der Wert dieser Variablen verschwunden
- wird die Methode erneut aufgerufen, so stellt sich der alte Wert der Variablen der Methode *nicht* wieder ein

```
public class MethodCall1 {  
    public static void doit() {  
        int i = 0;  
        System.out.println(i);  
        i = 42;  
    }  
    public static void main(String[] args) {  
        doit();  
        doit();  
    }  
}
```

Was gibt dieses
Programm aus?

Go!

Methoden und Variablen (Fort.)

- in unterschiedlichen Methoden können Variablen gleichen Namens vorkommen
- diese Variablen haben nichts miteinander zu tun

```
public class MethodCall2 {  
  
    public static void doit() {  
        int i = 42;  
        System.out.println(i);  
    }  
  
    public static void main(String[] args) {  
        int i = 13;  
        doit();  
        System.out.println(i);  
    }  
}
```

Was gibt dieses
Programm aus?

Vorlesung 6/1

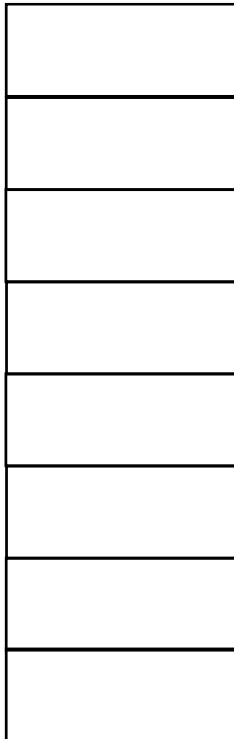
Methodenabarbeitung

- Methoden werden *stackartig* abgearbeitet
- wird innerhalb einer Methode **a** eine neue Methode **b** aufgerufen,
 - so merkt sich **a**, wo sie gerade ist
 - es wird (Speicher-)Platz für **b** geschaffen
 - **b** wird abgearbeitet
 - ist **b** fertig, wird der Speicherplatz wieder freigegeben und
 - **a** macht an der alten Stelle weiter
- wird innerhalb von **b** eine Methode **c** aufgerufen, so wird **b** auf dem Speicher abgelegt, **c** abgearbeitet, **b** danach wieder aktiviert und dann erst **a**

Methodenabarbeitung (Fort.)

Speicher (Stack)

main



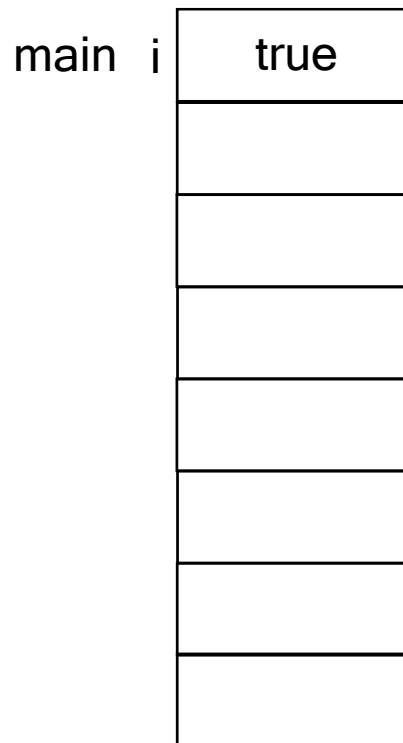
Hauptmethode main
wird aufgerufen

```
public class MethodCall3 {  
  
    public static void doit() {  
        int i = 42;  
        System.out.println(i);  
    }  
  
    public static void ichMacheWas() {  
        int i = 13;  
        System.out.println(i);  
        doit();  
    }  
  
    public static void main(String[] args) {  
        boolean i = true;  
        ichMacheWas();  
        System.out.println(i);  
    }  
}
```

hier fängt das
Programm an

Methodenabarbeitung (Fort.)

Speicher (Stack)

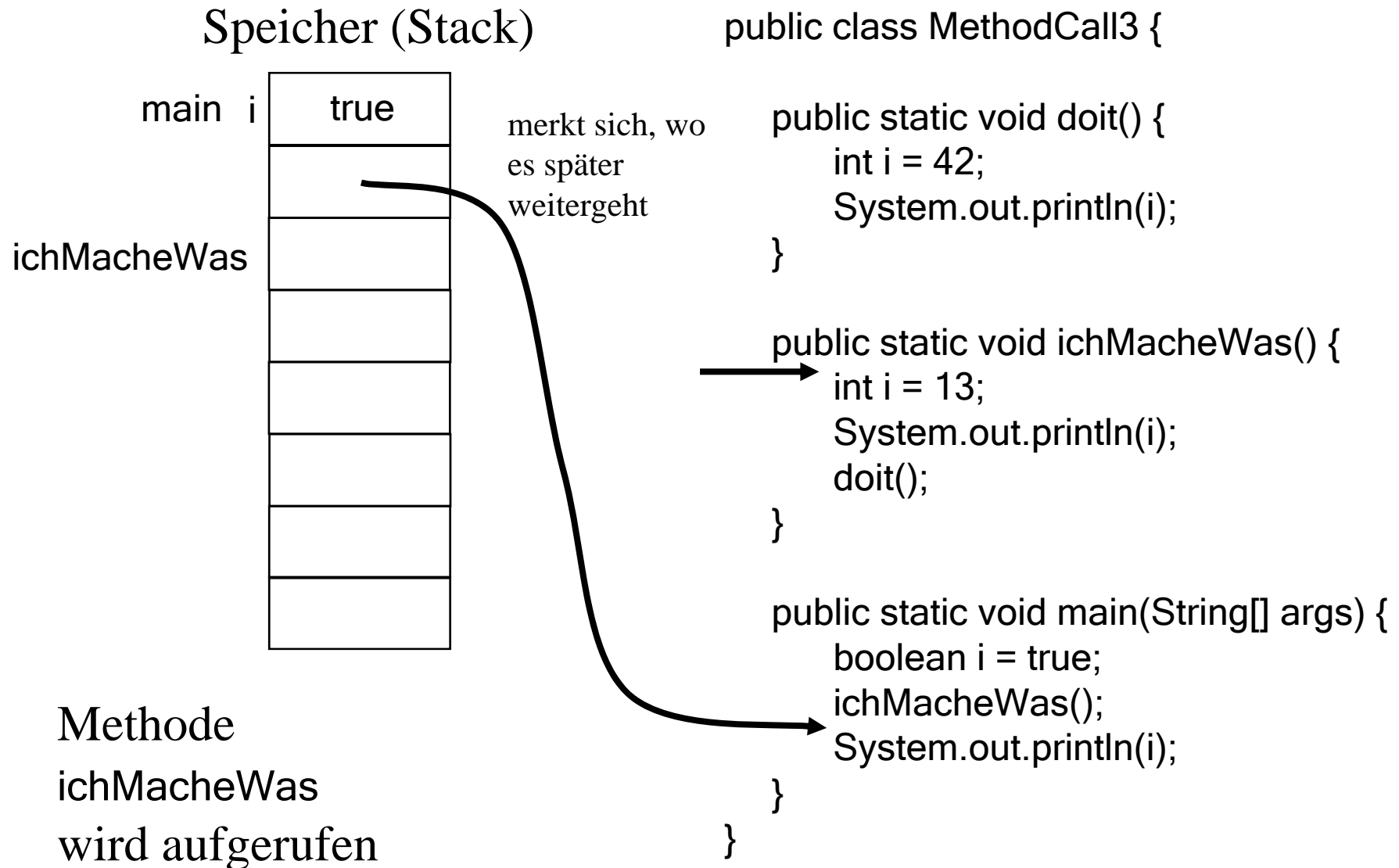


Variable i wird
angelegt

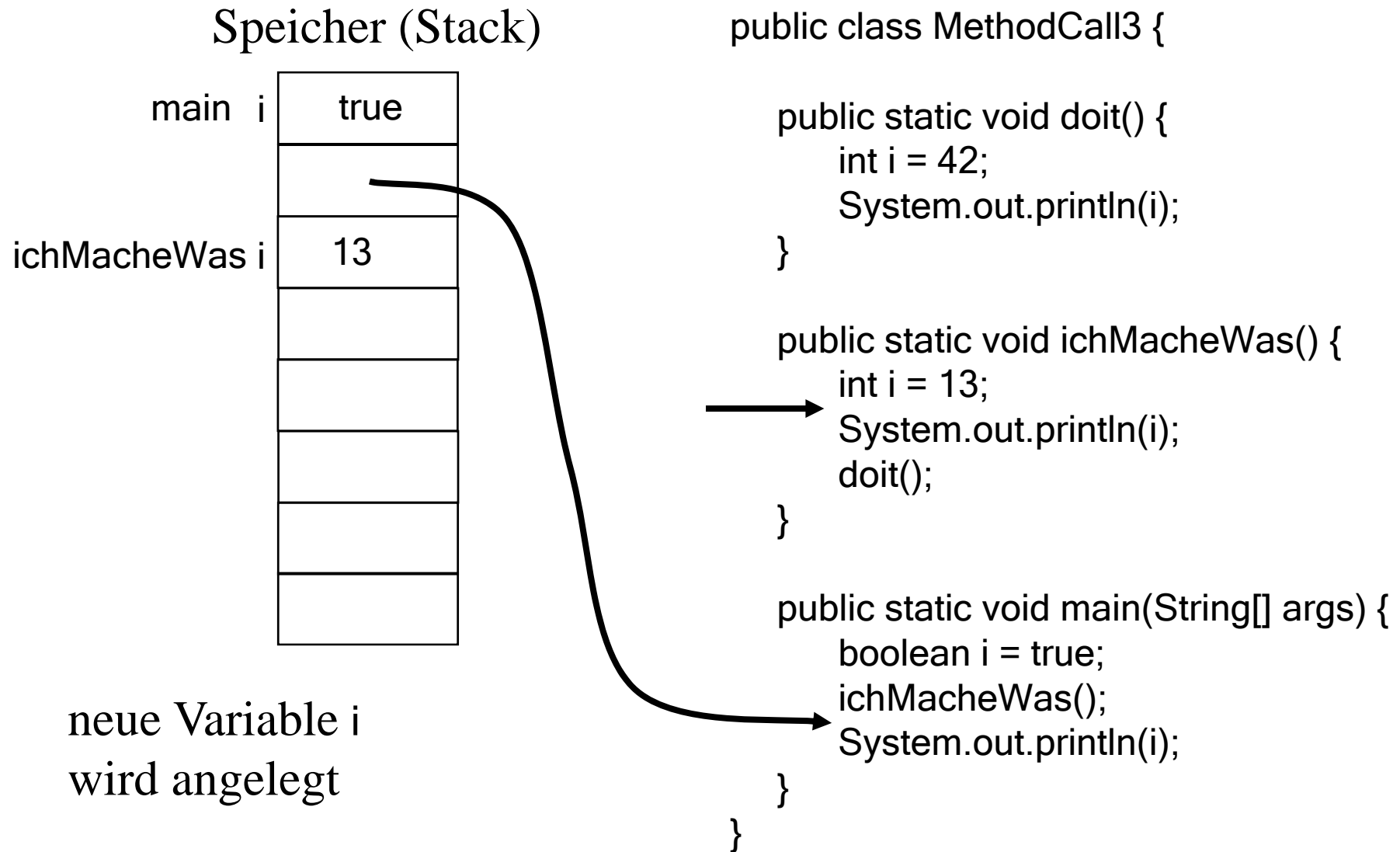
```
public class MethodCall3 {  
  
    public static void doit() {  
        int i = 42;  
        System.out.println(i);  
    }  
  
    public static void ichMacheWas() {  
        int i = 13;  
        System.out.println(i);  
        doit();  
    }  
  
    public static void main(String[] args) {  
        boolean i = true;  
        ichMacheWas();  
        System.out.println(i);  
    }  
}
```

→

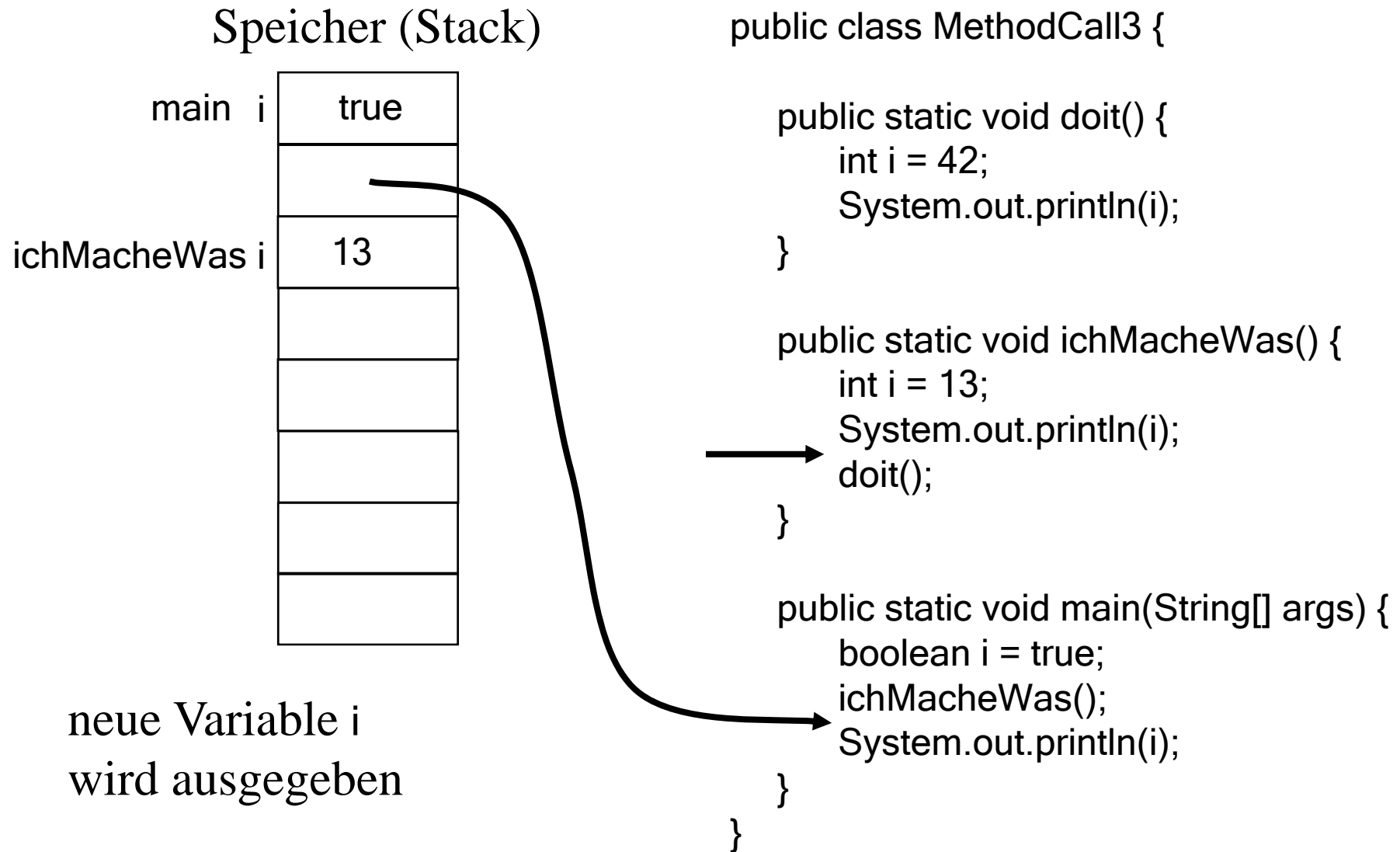
Methodenabarbeitung (Fort.)



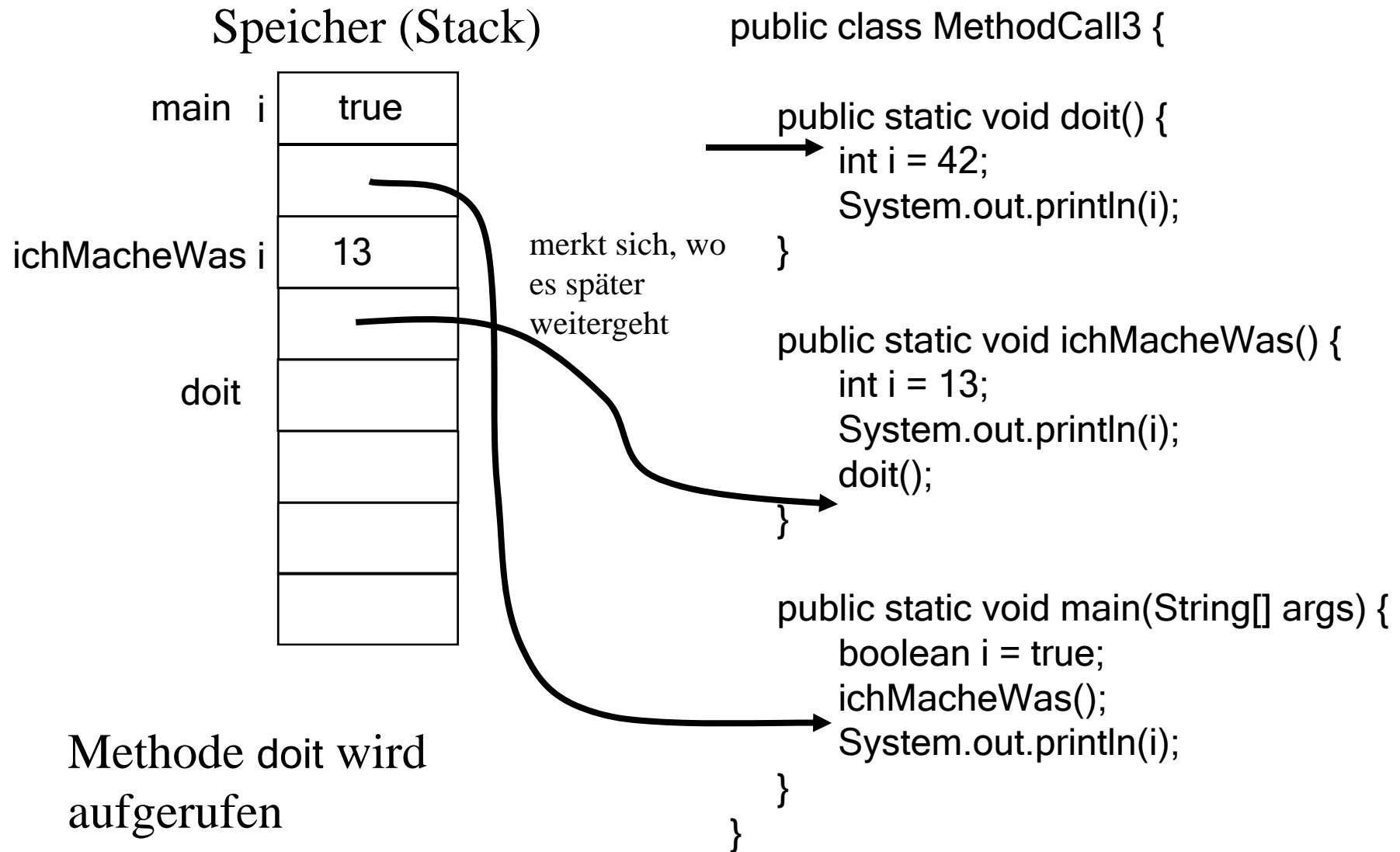
Methodenabarbeitung (Fort.)



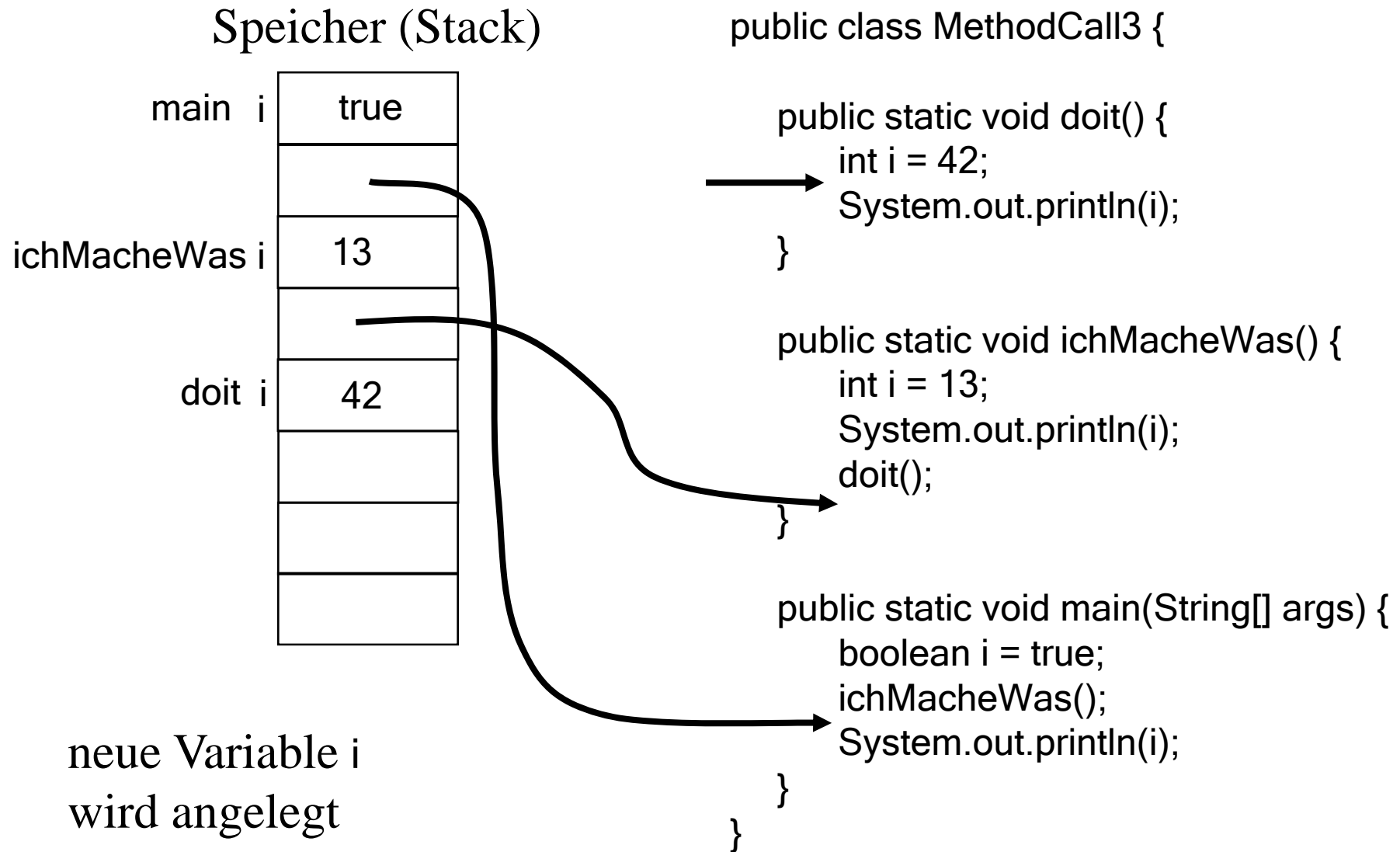
Methodenabarbeitung (Fort.)



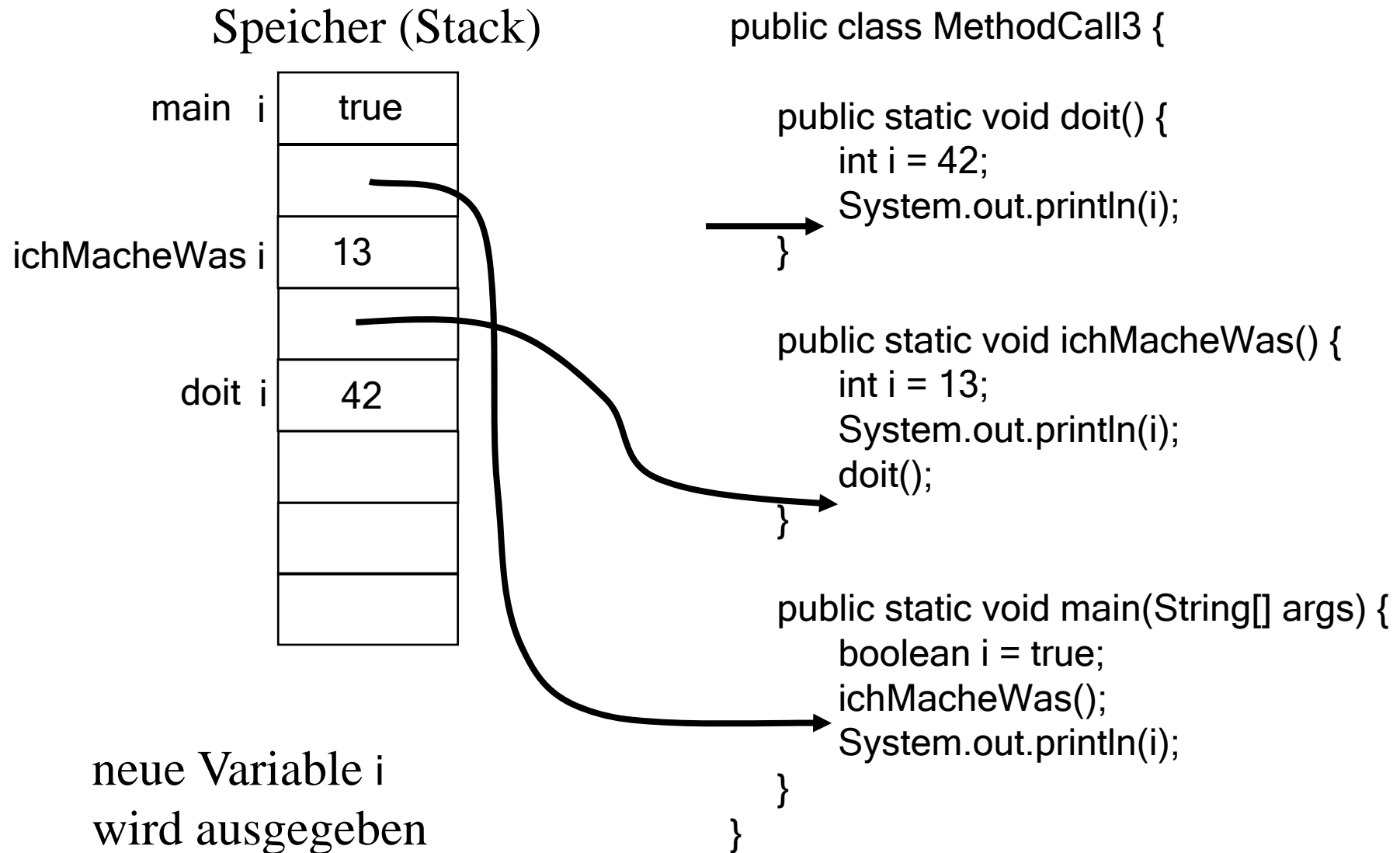
Methodenabarbeitung (Fort.)



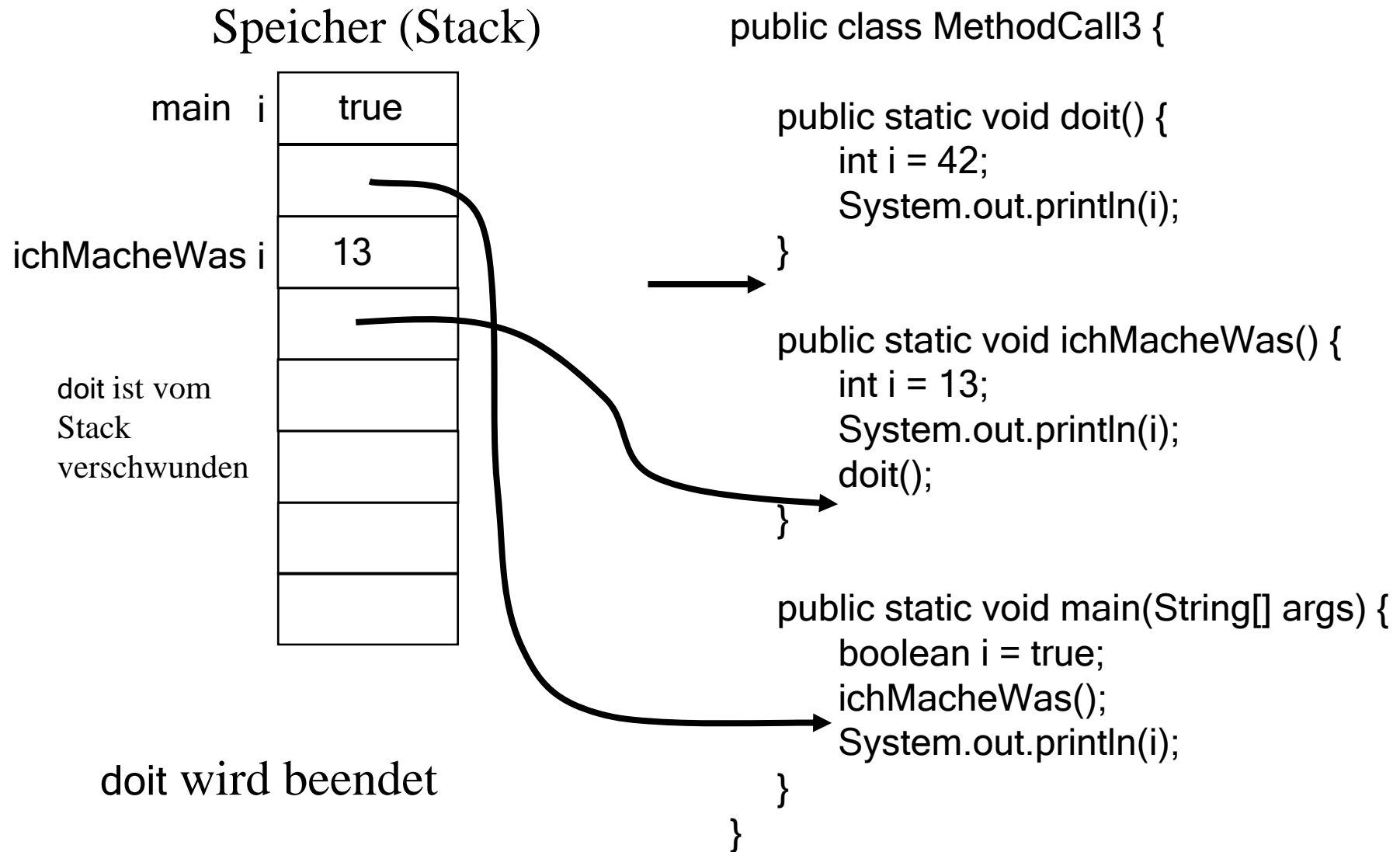
Methodenabarbeitung (Fort.)



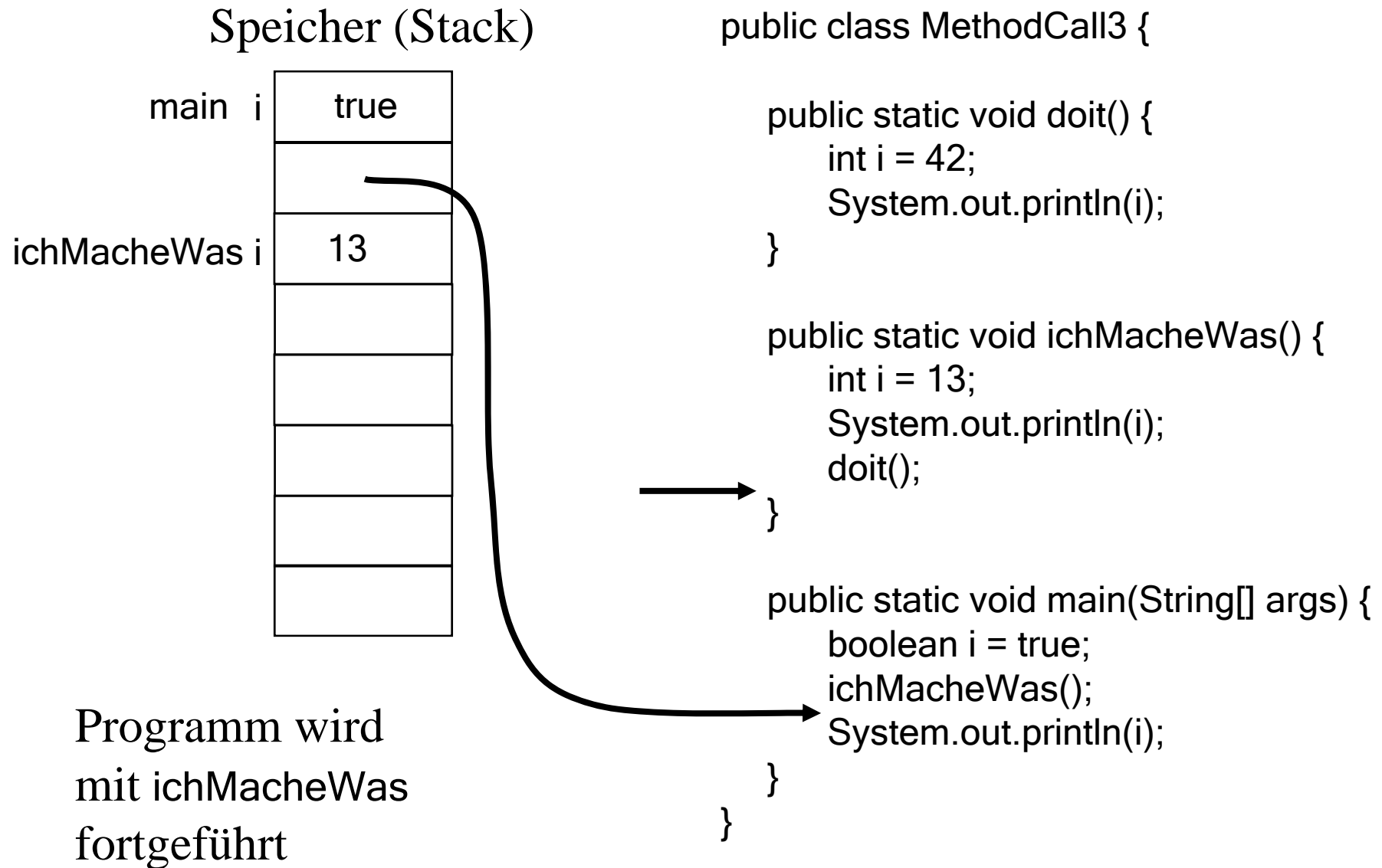
Methodenabarbeitung (Fort.)



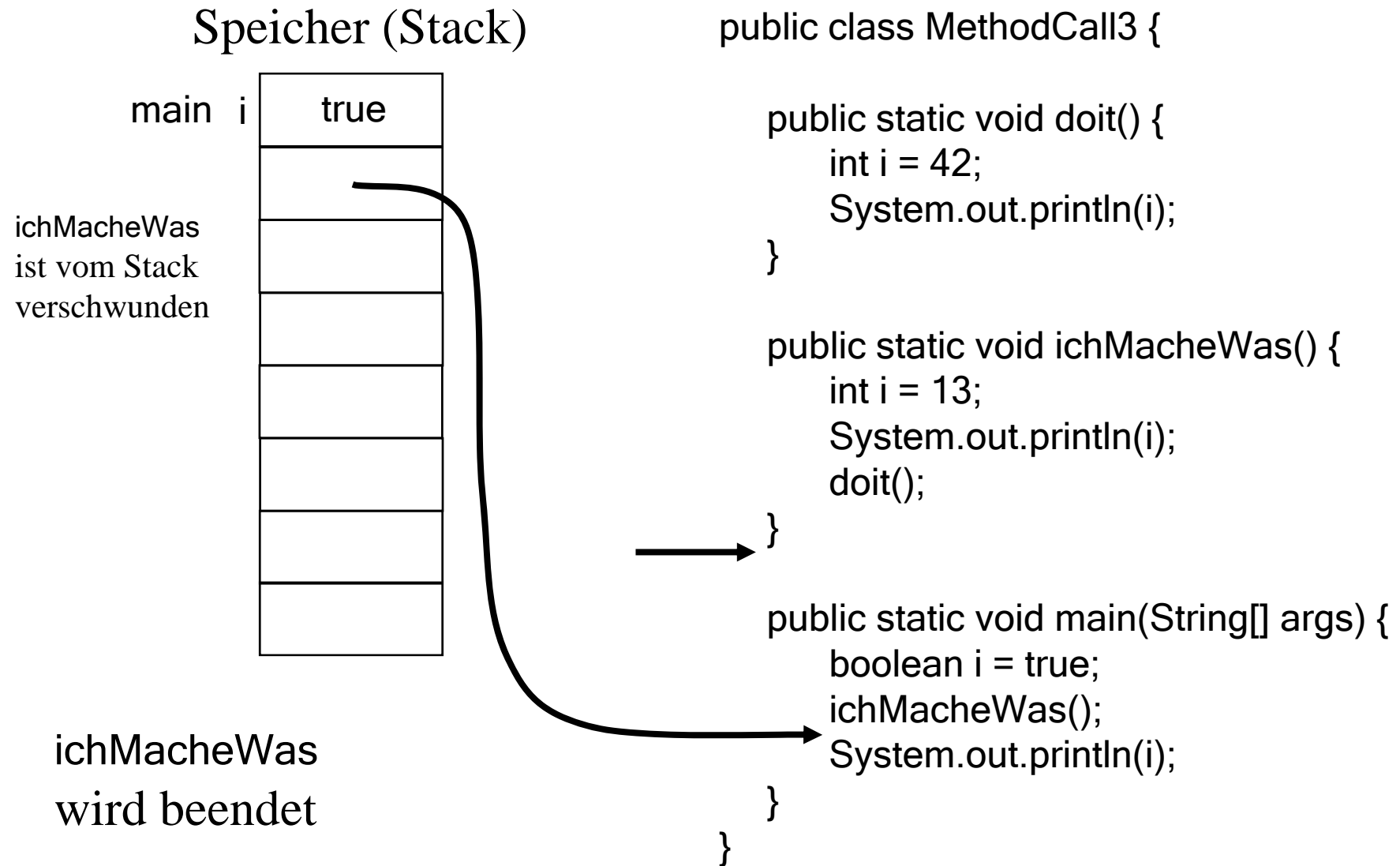
Methodenabarbeitung (Fort.)



Methodenabarbeitung (Fort.)

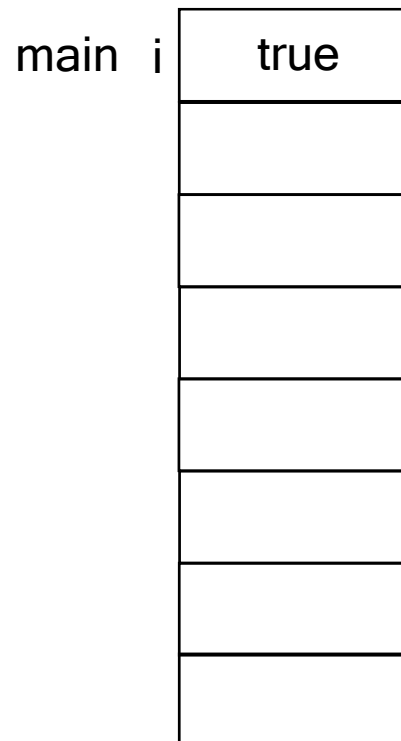


Methodenabarbeitung (Fort.)



Methodenabarbeitung (Fort.)

Speicher (Stack)



Programm wird mit
main fortgeführt

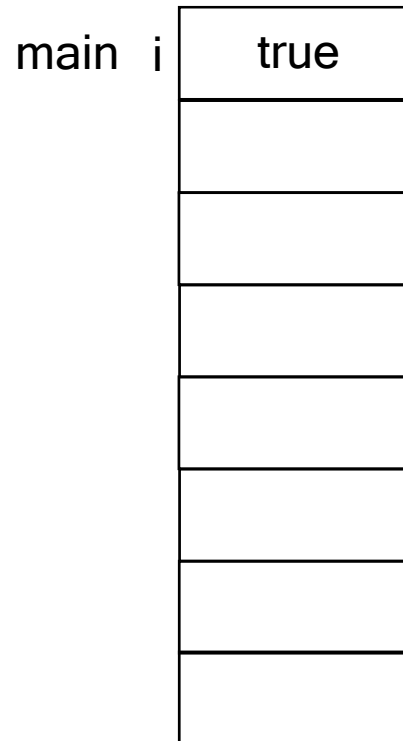
```
public class MethodCall3 {  
  
    public static void doit() {  
        int i = 42;  
        System.out.println(i);  
    }  
  
    public static void ichMacheWas() {  
        int i = 13;  
        System.out.println(i);  
        doit();  
    }  
  
    public static void main(String[] args) {  
        boolean i = true;  
        ichMacheWas();  
        System.out.println(i);  
    }  
}
```

→

Go!

Methodenabarbeitung (Fort.)

Speicher (Stack)



Variable i wird ausgegeben;
Programm wird beendet

```
public class MethodCall3 {  
  
    public static void doit() {  
        int i = 42;  
        System.out.println(i);  
    }  
  
    public static void ichMacheWas() {  
        int i = 13;  
        System.out.println(i);  
        doit();  
    }  
  
    public static void main(String[] args) {  
        boolean i = true;  
        ichMacheWas();  
        System.out.println(i);  
    }  
}
```

→

Methoden: Beispiel

Aufgabe:

- schreibe ein Programm, dass 3 Rechtecke auf dem Bildschirm ausdruckt
- diese 3 Rechtecke sollen sich in der Breite unterscheiden
- das 1. Rechteck soll 10 Zeichen breit sein
- das 2. Rechteck soll 15 Zeichen breit sein
- das 3. Rechteck soll 20 Zeichen breit sein

Methoden: Beispiel (Fort.)

Lösung 1 (ohne Methoden):

- 3-mal das Rechteck in Println-Anweisungen ausgeben
- hat die Nachteile, dass ein 4. Rechteck nicht so einfach ausgegeben werden kann
- die Breite der anderen Rechtecke zu ändern ist viel Änderungsaufwand

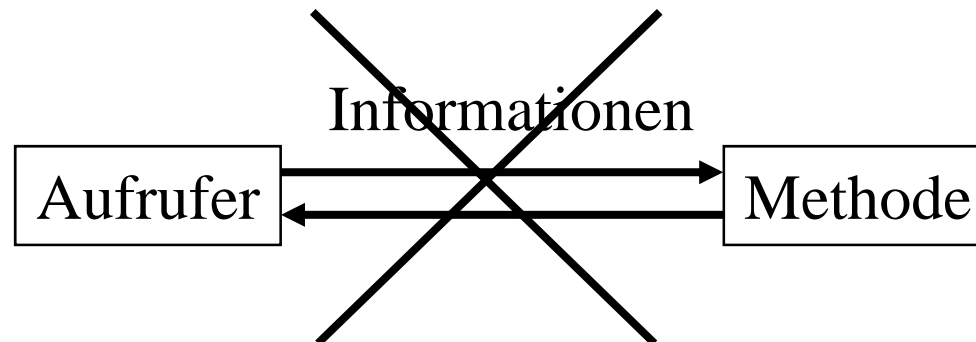
Lösung 2 (mit Methoden):

- Wie weiß die Methode, wie breit das Rechteck zu zeichnen ist?
- die Methode kann sich nichts merken
- die Methode erhält keine Information von ihrem Aufrufer

Methoden

Problem bei bisherigen Methoden:

- Methoden konnten aufgerufen werden, aber
- man konnte ihnen keine Informationen beim Aufruf mitgeben
- die Methoden konnten keine Informationen an den Aufrufer zurückgeben



Methoden mit Aufrufparameter

- um Methoden Informationen beim Aufruf mitzugeben, werden Methoden mit *Aufrufparametern* versehen
- diese Aufrufparameter werden beim *Aufruf mit Werten* versehen
- *verschiedene Aufrufe* können diese *Werte unterschiedlich* setzen
- somit wird i.d.R. die Methode ein *unterschiedliches Verhalten* zeigen, wenn sie mit unterschiedlichen Parametern aufgerufen wird

Methoden mit Aufrufparameter (Fort.)

Bsp.:

```
public static void print_rechteck(int iBreite) {  
    ...  
}
```

- die Methode `print_rechteck` hat einen Parameter bekommen
- dieser Parameter hat den Namen `iBreite`
- der Parameter hat den Typ `int`
- er verhält sich wie eine *lokale Variable* in der Methode `print_rechteck`, d.h.
 - er kann gelesen und beschrieben werden
 - nach dem Methodenaufruf verschwindet er wieder
 - es gelten die Sichtbarkeitsregeln

Methoden mit Aufrufparameter (Fort.)

- bei dem Aufruf einer Methode, die Parameter erwartet, müssen diese Parameter mit Werten gefüllt werden
- dazu gibt man beim Aufruf für jeden Parameter einen Ausdruck an

Bsp.:

```
public static void print_rechteck(int iBreite) {  
    ...  
}  
...  
print_rechteck(5*2);
```

Aufruf einer Methoden

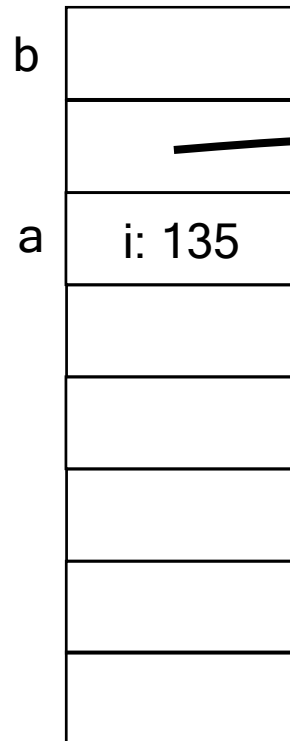
beim Aufruf einer Methode passieren folgende Dinge:

1. auf dem Stack merkt sich die aktuelle Methode (hier **b**), wo sie nach dem Aufruf von **a** weitermachen muss (hier die **print-Anweisung**)
2. auf dem Stack wird Platz für die neue Methode **a** geschaffen
3. in diesem Platz wird eine Variable **i** für den Parameter angelegt
4. der Ausdruck für den Parameter (hier **45*3**) wird ausgewertet
5. der Wert des Ausdrucks wird in die Variable **i** geschrieben
6. das Programm arbeitet die erste Anweisung von **a** ab

```
public static void a(int i) {  
    ...  
}  
  
public static void b() {  
    ──────────> a(45*3);  
                System.out.print(...);  
}
```


Aufruf einer Methoden (Fort.)

Speicher (Stack)



```
public static void a(int i) {  
    ...  
}
```

```
public static void b() {  
    a(45*3);  
    System.out.print(...);  
}
```


Situation:

- die Methode **b** hat **a** aufgerufen
- **a** liegt jetzt auf dem Stack
- die nächste abzuarbeitende Anweisung ist die 1. Anweisung von **a**
- der Parameter **i** von **a** liegt auf dem Stack
- der Parameter **i** von **a** hat den Wert des Aufrufausdrucks $45*3$

Beendigung einer Methoden

beim Beenden einer Methode passieren folgende Dinge:

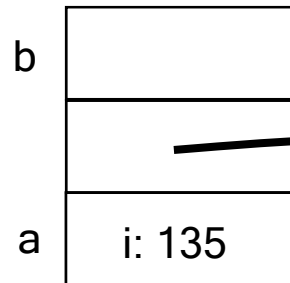
1. auf dem Stack wird die aktuelle Methode (hier **a**), entfernt
 2. die vorherige Methode (hier **b**) wird jetzt wieder die aktuelle Methode
 3. das Programm macht an der Stelle weiter, die sich vor dem Methodenaufruf auf dem Stack gemerkt wurde (hier die **print-Anweisung**)
- **WICHTIG:** es werden *keine Werte zurückgegeben*



```
public static void a(int i) {  
    ...  
}  
  
public static void b() {  
    a(45*3);  
    System.out.print(...);  
}
```

Beendigung einer Methoden (Fort.)

Speicher (Stack)

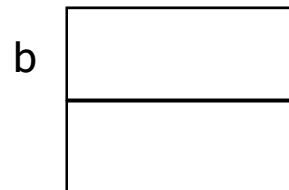


```
public static void a(int i) {  
    ...  
}  
  
public static void b() {  
    a(45*3);  
    System.out.print(...);  
}
```

Situation:

- die Methode **a** ist fertig
- **a** liegt jetzt nicht mehr auf dem Stack
- die Methode **b** arbeitet die nächste Anweisung nach dem Methodenaufruf ab

Speicher (Stack)



```
public static void a(int i) {  
    ...  
}  
  
public static void b() {  
    → a(45*3);  
    System.out.print(...);  
}
```

Go!

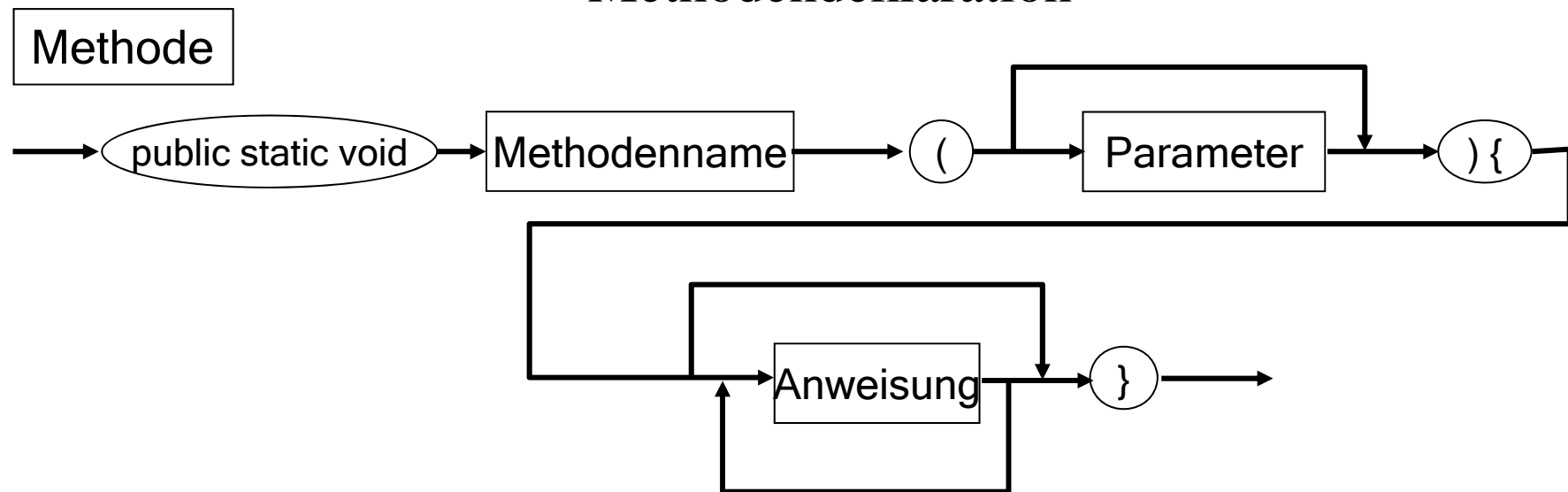
Beispiel

```
public class MethodCall6 {  
    public static void a(int i) {  
        System.out.println("a: i = " + i);  
    }  
  
    public static void b() {  
        a(45*3);  
        System.out.println("in b");  
    }  
  
    public static void main(String[] args) {  
        int i = 0;  
        System.out.println("vor b");  
        b();  
        System.out.println("nach b");  
        System.out.println("main: i = " + i);  
    }  
}
```

zwei ganz verschiedene i's; die
haben nichts miteinander zu tun

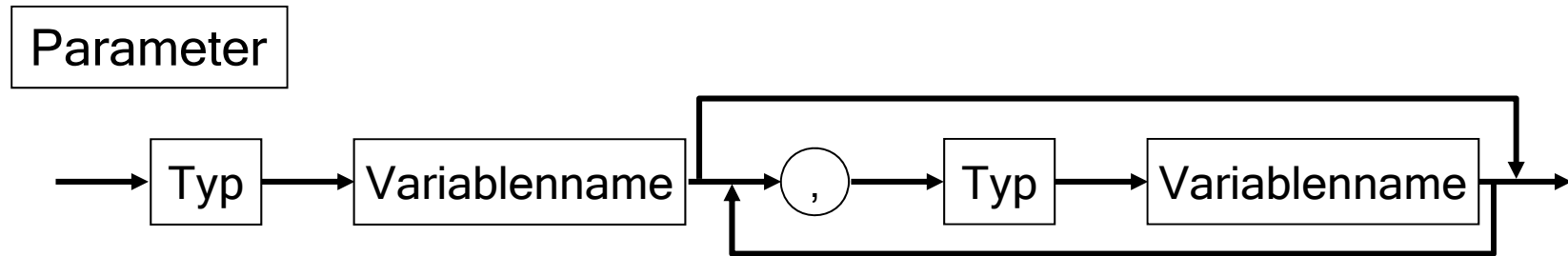
Vorlesung 6/2

Methodendeklaration



- eine Methode enthält zwischen den Klammern die optionale Deklaration ihrer Parameter, d.h.
- eine Methode muss keine Parameter enthalten

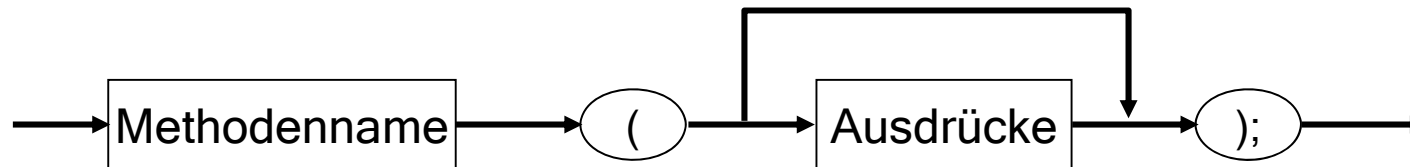
Methodendeklaration (Fort.)



- Parameter sind eine Liste von Variablennamen (Parameternamen) mit ihren vorangestellten Typen
- nach einem Parameter können noch beliebig weitere folgen
- mehrere Parameter werden durch Komma voneinander getrennt

Methodenaufruf

Methodenaufruf



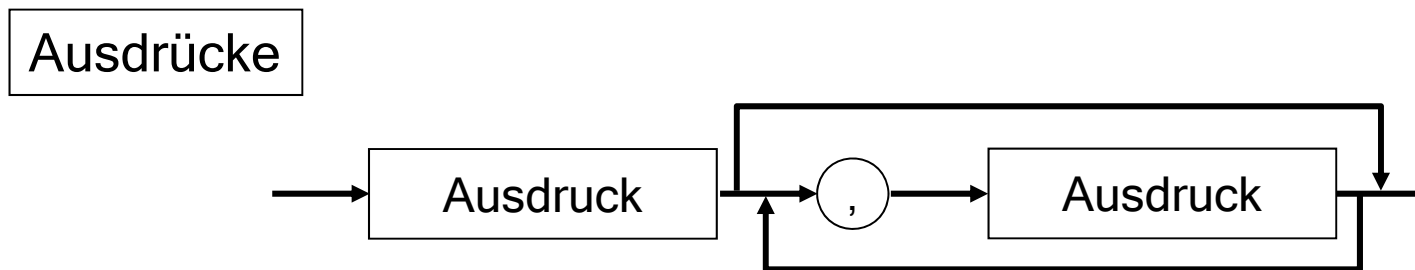
- um eine Methode, die Parameter hat, aufzurufen müssen beim Aufruf den Parameter Werte übergeben werden
- diese Werte sind allgemeine Ausdrücke
- die Werte werden den Parametern von links nach rechts übergeben, d.h. der linke Ausdruck wird dem linken Parameter übergeben usw.

⇒ die Anzahl der Parameter muss gleich der Anzahl der übergebenen Ausdrücke sein

⇒ die Typen der jeweiligen Ausdrücke müssen mit den Typen der korrespondierenden Parameter übereinstimmen

Methodenaufruf (Fort.)

- Ausdrücke sind eine Liste von einzelnen Ausdrücken , die durch Komma getrennt sind



zurück zur Aufgabe:

- schreibe ein Programm, dass 3 Rechtecke auf dem Bildschirm ausdruckt

Lösung:

- eine Methode, die die Breite als int-Wert übergeben bekommt

Begriffe

- die Parameter einer Methodendeklaration nennt man *formale Parameter*
- die Ausdrücke in einem Methodenaufruf nennt man *aktuelle Parameter*
- vor einem Methodenaufruf werden die aktuellen Parameter ausgewertet und an die *formalen Parameter gebunden oder übergeben*

Beispiel

```
public class Rechteck6 {  
    public static void print_rechteck(int iBreite) {  
        System.out.print("+");  
        for(int i = 0; i < iBreite-2; ++i) {  
            System.out.print("-");  
        }  
        System.out.println("+");  
        for(int j = 0; j < 4; ++j) {  
            System.out.print("|");  
            for(int i = 0; i < iBreite-2; ++i) {  
                System.out.print(" ");  
            }  
            System.out.println("|");  
        }  
        System.out.print("+");  
        for(int i = 0; i < iBreite-2; ++i) {  
            System.out.print("-");  
        }  
        System.out.println("+");  
    }  
}
```

drucke die Kopfzeile

drucke 4-mal ...

... die mittlere Zeile

drucke die Fußzeile

...

Go!

Beispiel (Fort.)

...

```
public static void main(String[] args) {  
    print_rechteck(10);  
    System.out.println("dies war das 1. Rechteck");  
    print_rechteck(15);  
    System.out.println("gleich kommt das Letzte");  
    print_rechteck(20);  
}  
}
```

die Methode print_rechteck
wird 3-mal mit
verschiedenen Werten für
iBreite aufgerufen

Diskussion des Beispiels

bei der Methode `print_rechteck` fällt auf:

- 3-mal werden die gleichen bzw. ähnliche Dinge getan
- dies sollte man wieder als Methode auslagern
- Idee dieser Methode:
 - drucke das erste Zeichen
 - drucke dann `iBreite`-mal die Zwischenzeichen
 - drucke dann das letzte Zeichen mit einem Zeilenumbruch
- dazu muss dieser Methode `iBreite` und 2 Zeichen übergeben werden

```
System.out.print("+");
for(int i = 0; i < iBreite-2; ++i) {
    System.out.print("-");
}
System.out.println("+");
```

```
System.out.print("|");
for(int i = 0; i < iBreite-2; ++i) {
    System.out.print(" ");
}
System.out.println("|");
```

```
System.out.print("+");
for(int i = 0; i < iBreite-2; ++i) {
    System.out.print("-");
}
System.out.println("+");
```

Diskussion des Beispiels (Fort.)

<pre>public static void print_line(int iBreite,char cS,char cB) {</pre>	
<pre> System.out.print(cS);</pre>	drucke das 1. Zeichen
<pre> for(int i = 0;i < iBreite-2;++i) {</pre>	
<pre> System.out.print(cB);</pre>	drucke die Zeichen dazwischen
<pre> }</pre>	
<pre> System.out.println(cS);</pre>	drucke das letzte Zeichen
<pre>}</pre>	

- eine Methode `print_line`, die eine Zeile drucken soll
- die Zeilen unterscheiden sich voneinander in
 - ihrer Länge
 - ihrem ersten und letzten Zeichen
 - ihren Zeichen dazwischen
- daher wird neben der Länge auch die anderen beiden Zeichen übergeben

Beispiel

```
public class Rechteck7 {  
    public static void print_line(int iBreite, char cS, char cB) {  
        System.out.print(cS);  
        for(int i = 0; i < iBreite-2; ++i) {  
            System.out.print(cB);  
        }  
        System.out.println(cS);  
    }  
}
```

```
public static void print_rechteck(int iBreite) {  
    print_line(iBreite, '+', '-');  
    for(int j = 0; j < 4; ++j) {  
        print_line(iBreite, '|', ' ');  
    }  
    print_line(iBreite, '+', '-');  
}
```

...

die Methode print_line wird 6-mal mit der gleichen Breite iBreite aber unterschiedlichen Zeichen aufgerufen

Go!

Beispiel (Fort.)

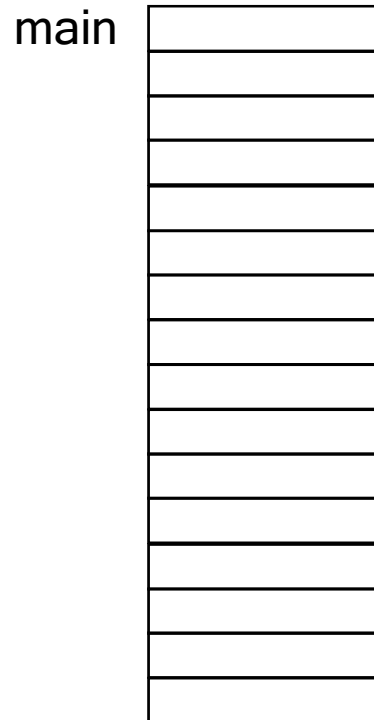
...

```
public static void main(String[] args) {  
    print_rechteck(10);  
    System.out.println("dies war das 1. Rechteck");  
    print_rechteck(15);  
    System.out.println("gleich kommt das Letzte");  
    print_rechteck(20);  
}  
}
```

die Methode print_rechteck
wird 3-mal mit
verschiedenen Werten für
iBreite aufgerufen

Beispiel (Fort.)

Speicher (Stack)



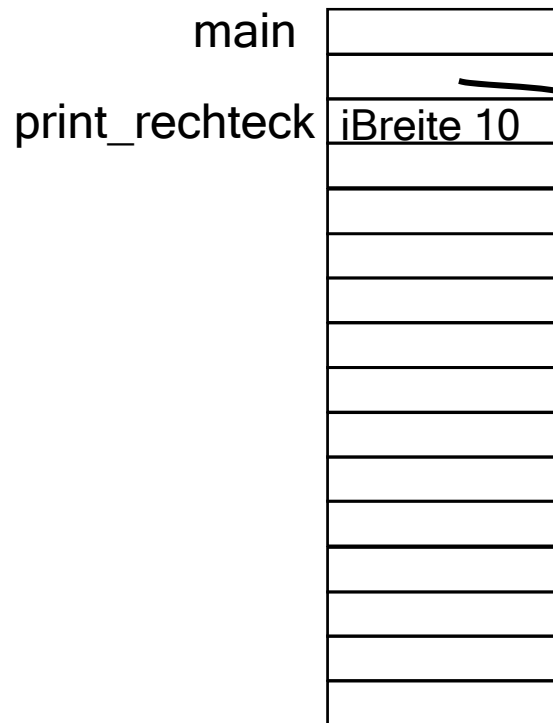
Hauptmethode main
wird aufgerufen

```
public class Rechteck7 {  
    public static void print_line( int iBreite, char cS, char cB) {  
        System.out.print(cS);  
        for(int i = 0; i < iBreite-2; ++i) {  
            System.out.print(cB);  
        }  
        System.out.println(cS);  
    }  
  
    public static void print_rechteck(int iBreite) {  
        print_line(iBreite, '+', '-');  
        for(int j = 0; j < 4; ++j) {  
            print_line(iBreite, '|', ' ');  
        }  
        print_line(iBreite, '+', '-');  
    }  
  
    public static void main(String[] args) {  
        print_rechteck(10);  
        System.out.println("dies war das 1. Rechteck");  
        print_rechteck(15);  
        System.out.println("gleich kommt das Letzte");  
        print_rechteck(20);  
    }  
}
```

hier fängt das
Programm an

Beispiel (Fort.)

Speicher (Stack)

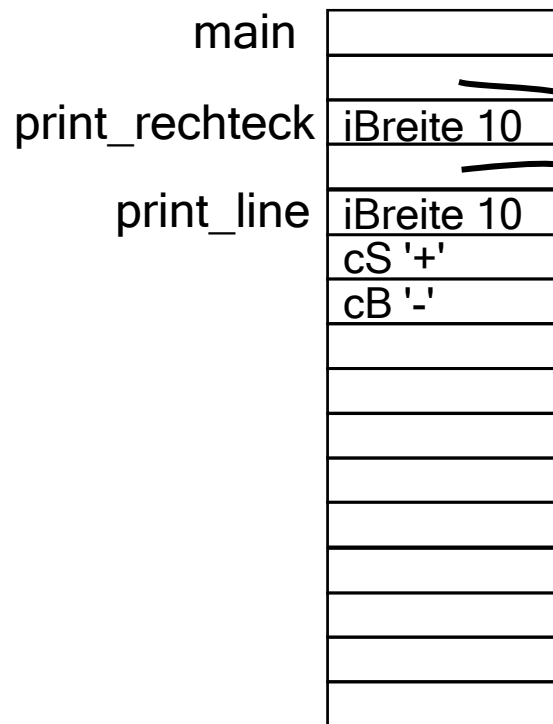


Methode print_rechteck ist
aufgerufen; Parameter
iBreite ist auf 10 gesetzt

```
public class Rechteck7 {  
    public static void print_line( int iBreite, char cS, char cB) {  
        System.out.print(cS);  
        for(int i = 0; i < iBreite-2; ++i) {  
            System.out.print(cB);  
        }  
        System.out.println(cS);  
    }  
  
    public static void print_rechteck(int iBreite) {  
        print_line(iBreite, '+', '-');  
        for(int j = 0; j < 4; ++j) {  
            print_line(iBreite, '|', ' ');  
        }  
        print_line(iBreite, '+', '-');  
    }  
  
    public static void main(String[] args) {  
        print_rechteck(10);  
        System.out.println("dies war das 1. Rechteck");  
        print_rechteck(15);  
        System.out.println("gleich kommt das Letzte");  
        print_rechteck(20);  
    }  
}
```

Beispiel (Fort.)

Speicher (Stack)

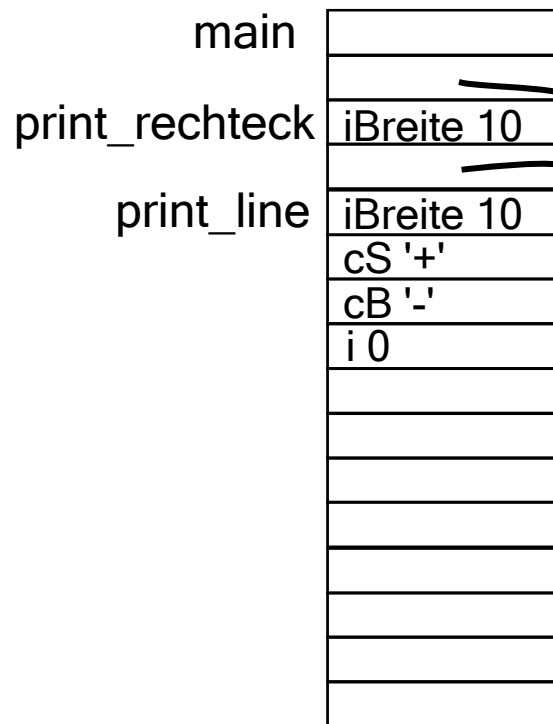


Methode `print_line`
ist aufgerufen

```
public class Rechteck7 {  
    public static void print_line( int iBreite, char cS, char cB) {  
        System.out.print(cS);  
        for(int i = 0; i < iBreite-2; ++i) {  
            System.out.print(cB);  
        }  
        System.out.println(cS);  
    }  
  
    public static void print_rechteck(int iBreite) {  
        print_line(iBreite, '+', '-');  
        for(int j = 0; j < 4; ++j) {  
            print_line(iBreite, '|', ' ');  
        }  
        print_line(iBreite, '+', '-');  
    }  
  
    public static void main(String[] args) {  
        print_rechteck(10);  
        System.out.println("dies war das 1. Rechteck");  
        print_rechteck(15);  
        System.out.println("gleich kommt das Letzte");  
        print_rechteck(20);  
    }  
}
```

Beispiel (Fort.)

Speicher (Stack)



```
public class Rechteck7 {  
    public static void print_line( int iBreite, char cS, char cB) {  
        System.out.print(cS);  
        for(int i = 0; i < iBreite-2; ++i) {  
            System.out.print(cB);  
        }  
        System.out.println(cS);  
    }  
}
```

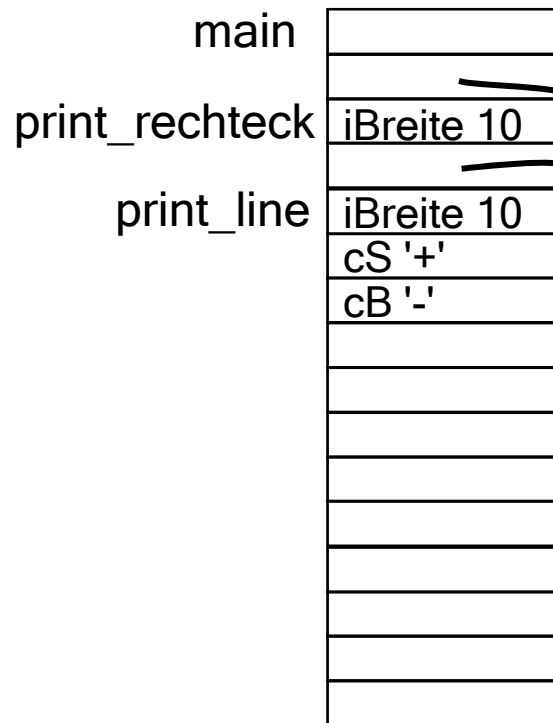
```
public static void print_rechteck(int iBreite) {  
    print_line(iBreite, '+', '-');  
    for(int j = 0; j < 4; ++j) {  
        print_line(iBreite, '|', ' ');  
    }  
    print_line(iBreite, '+', '-');  
}
```

```
public static void main(String[] args) {  
    print_rechteck(10);  
    System.out.println("dies war das 1. Rechteck");  
    print_rechteck(15);  
    System.out.println("gleich kommt das Letzte");  
    print_rechteck(20);  
}
```

Methode `print_line` ist in der
Schleife; lokale Variable `i`
liegt auf dem Stack

Beispiel (Fort.)

Speicher (Stack)

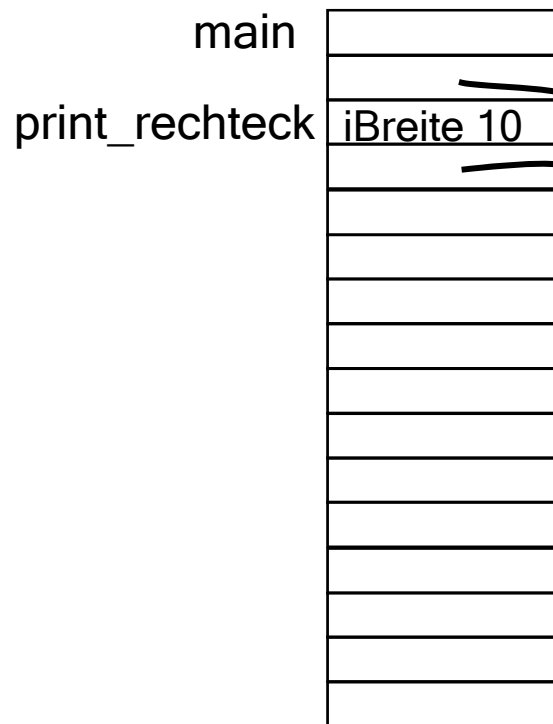


Methode `print_line` hat die Schleife verlassen; lokale Variable `i` wird gelöscht

```
public class Rechteck7 {  
    public static void print_line( int iBreite, char cS, char cB) {  
        System.out.print(cS);  
        for(int i = 0; i < iBreite-2; ++i) {  
            System.out.print(cB);  
        }  
        System.out.println(cS);  
    }  
  
    public static void print_rechteck(int iBreite) {  
        print_line(iBreite, '+', '-');  
        for(int j = 0; j < 4; ++j) {  
            print_line(iBreite, '|', ' ');  
        }  
        print_line(iBreite, '+', '-');  
    }  
  
    public static void main(String[] args) {  
        print_rechteck(10);  
        System.out.println("dies war das 1. Rechteck");  
        print_rechteck(15);  
        System.out.println("gleich kommt das Letzte");  
        print_rechteck(20);  
    }  
}
```

Beispiel (Fort.)

Speicher (Stack)

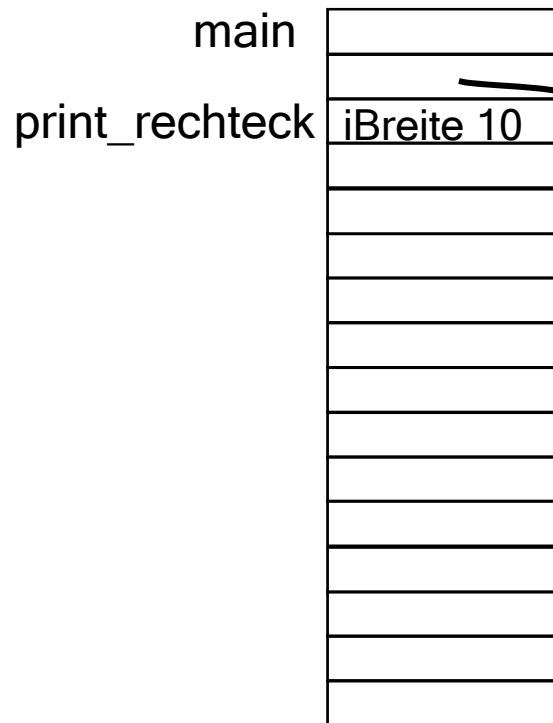


Methode print_line ist
beendet und wird vom
Stack genommen

```
public class Rechteck7 {  
    public static void print_line( int iBreite, char cS, char cB) {  
        System.out.print(cS);  
        for(int i = 0; i < iBreite-2; ++i) {  
            System.out.print(cB);  
        }  
        System.out.println(cS);  
    }  
  
    public static void print_rechteck(int iBreite) {  
        print_line(iBreite, '+', '-');  
        for(int j = 0; j < 4; ++j) {  
            print_line(iBreite, '|', ' ');  
        }  
        print_line(iBreite, '+', '-');  
    }  
  
    public static void main(String[] args) {  
        print_rechteck(10);  
        System.out.println("dies war das 1. Rechteck");  
        print_rechteck(15);  
        System.out.println("gleich kommt das Letzte");  
        print_rechteck(20);  
    }  
}
```

Beispiel (Fort.)

Speicher (Stack)

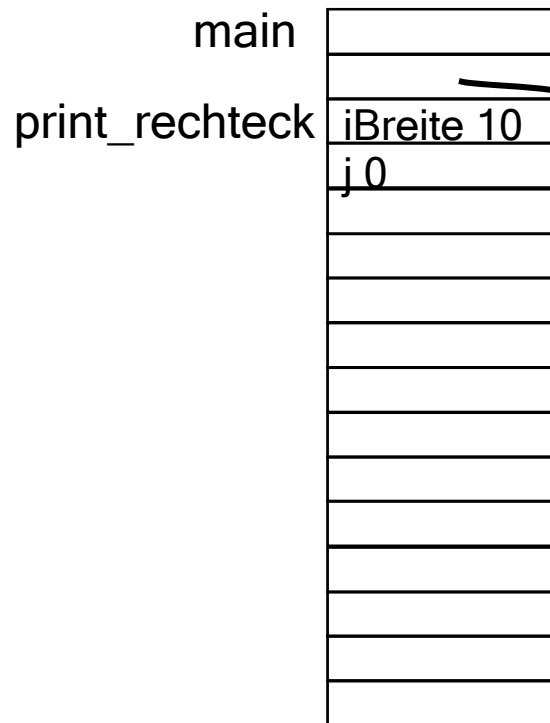


Methode print_rechteck
wird weiter abgearbeitet

```
public class Rechteck7 {  
    public static void print_line( int iBreite, char cS, char cB) {  
        System.out.print(cS);  
        for(int i = 0; i < iBreite-2; ++i) {  
            System.out.print(cB);  
        }  
        System.out.println(cS);  
    }  
  
    public static void print_rechteck(int iBreite) {  
        print_line(iBreite, '+', '-');  
        for(int j = 0; j < 4; ++j) {  
            print_line(iBreite, '|', ' ');  
        }  
        print_line(iBreite, '+', '-');  
    }  
  
    public static void main(String[] args) {  
        print_rechteck(10);  
        System.out.println("dies war das 1. Rechteck");  
        print_rechteck(15);  
        System.out.println("gleich kommt das Letzte");  
        print_rechteck(20);  
    }  
}
```

Beispiel (Fort.)

Speicher (Stack)

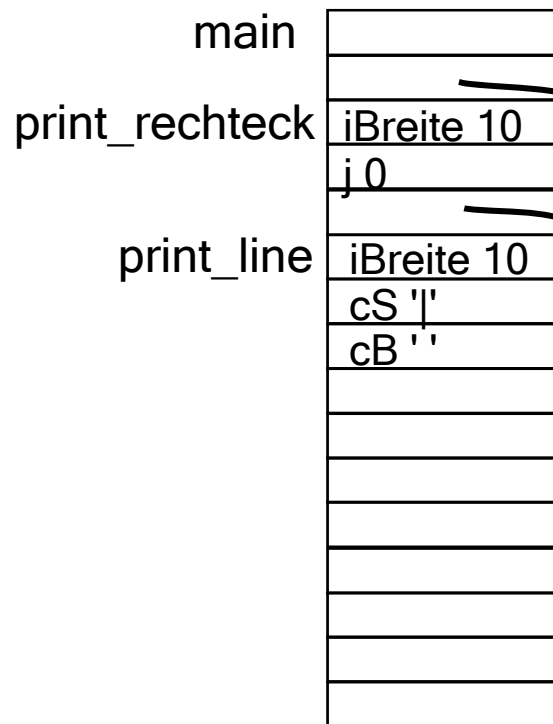


Methode print_rechteck wird
weiter abgearbeitet; lokale
Variable j wird angelegt

```
public class Rechteck7 {  
    public static void print_line( int iBreite, char cS, char cB) {  
        System.out.print(cS);  
        for(int i = 0; i < iBreite-2; ++i) {  
            System.out.print(cB);  
        }  
        System.out.println(cS);  
    }  
  
    public static void print_rechteck(int iBreite) {  
        print_line(iBreite, '+', '-');  
        for(int j = 0; j < 4; ++j) {  
            print_line(iBreite, '|', ' ');  
        }  
        print_line(iBreite, '+', '-');  
    }  
  
    public static void main(String[] args) {  
        print_rechteck(10);  
        System.out.println("dies war das 1. Rechteck");  
        print_rechteck(15);  
        System.out.println("gleich kommt das Letzte");  
        print_rechteck(20);  
    }  
}
```


Beispiel (Fort.)

Speicher (Stack)

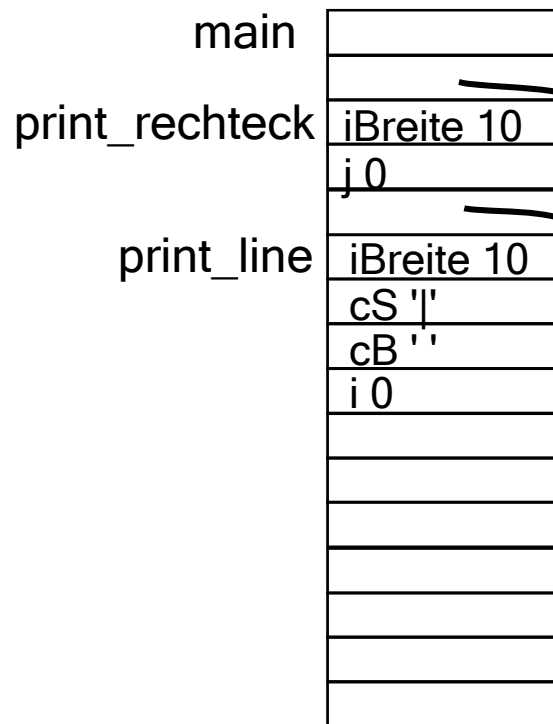


Methode print_line
wird aufgerufen

```
public class Rechteck7 {  
    public static void print_line( int iBreite, char cS, char cB) {  
        System.out.print(cS);  
        for(int i = 0; i < iBreite-2; ++i) {  
            System.out.print(cB);  
        }  
        System.out.println(cS);  
    }  
  
    public static void print_rechteck(int iBreite) {  
        print_line(iBreite, '+', '-');  
        for(int j = 0; j < 4; ++j) {  
            print_line(iBreite, '|', ' ');  
        }  
        print_line(iBreite, '+', '-');  
    }  
  
    public static void main(String[] args) {  
        print_rechteck(10);  
        System.out.println("dies war das 1. Rechteck");  
        print_rechteck(15);  
        System.out.println("gleich kommt das Letzte");  
        print_rechteck(20);  
    }  
}
```

Beispiel (Fort.)

Speicher (Stack)

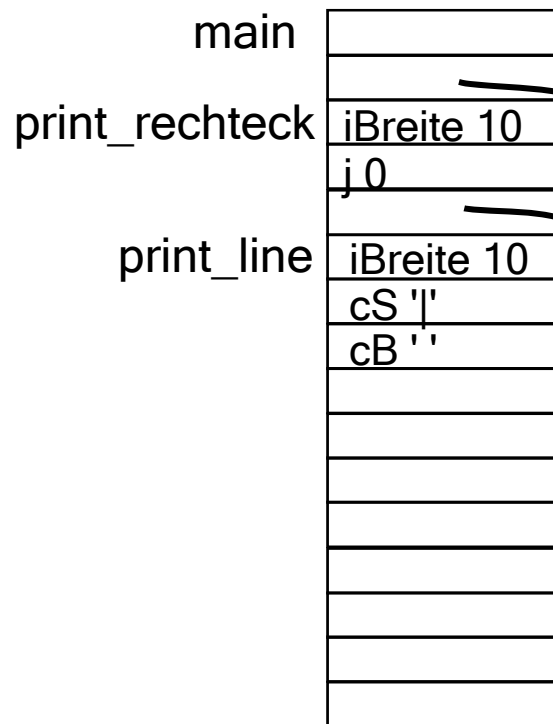


Methode `print_line` ist in der Schleife; lokale Variable `i` liegt auf dem Stack

```
public class Rechteck7 {  
    public static void print_line( int iBreite, char cS, char cB) {  
        System.out.print(cS);  
        for(int i = 0; i < iBreite-2; ++i) {  
            System.out.print(cB);  
        }  
        System.out.println(cS);  
    }  
  
    public static void print_rechteck(int iBreite) {  
        print_line(iBreite, '+', '-');  
        for(int j = 0; j < 4; ++j) {  
            print_line(iBreite, '|', ' ');  
        }  
        print_line(iBreite, '+', '-');  
    }  
  
    public static void main(String[] args) {  
        print_rechteck(10);  
        System.out.println("dies war das 1. Rechteck");  
        print_rechteck(15);  
        System.out.println("gleich kommt das Letzte");  
        print_rechteck(20);  
    }  
}
```

Beispiel (Fort.)

Speicher (Stack)

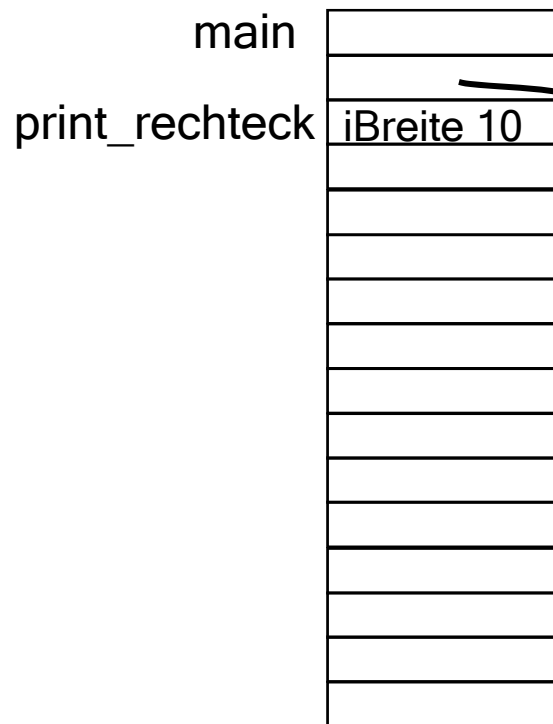


am Ende der Methode
print_line ist die Variable i
vom Stack verschwunden

```
public class Rechteck7 {  
    public static void print_line( int iBreite, char cS, char cB) {  
        System.out.print(cS);  
        for(int i = 0; i < iBreite-2; ++i) {  
            System.out.print(cB);  
        }  
        System.out.println(cS);  
    }  
  
    public static void print_rechteck(int iBreite) {  
        print_line(iBreite, '+', '-');  
        for(int j = 0; j < 4; ++j) {  
            print_line(iBreite, '|', ' ');  
        }  
        print_line(iBreite, '+', '-');  
    }  
  
    public static void main(String[] args) {  
        print_rechteck(10);  
        System.out.println("dies war das 1. Rechteck");  
        print_rechteck(15);  
        System.out.println("gleich kommt das Letzte");  
        print_rechteck(20);  
    }  
}
```

Beispiel (Fort.)

Speicher (Stack)

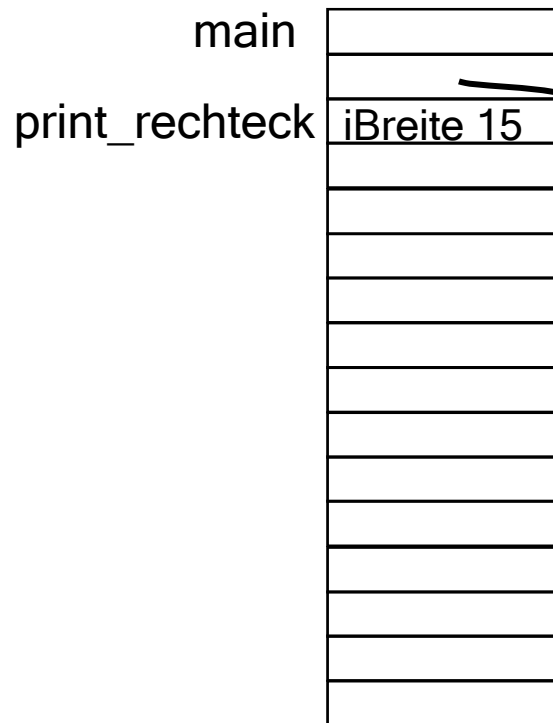


nach der Methode `print_line`
und nach der Schleife von
`print_rechteck` ist auch `j` vom
Stack verschwunden

```
public class Rechteck7 {  
    public static void print_line( int iBreite, char cS, char cB) {  
        System.out.print(cS);  
        for(int i = 0; i < iBreite-2; ++i) {  
            System.out.print(cB);  
        }  
        System.out.println(cS);  
    }  
  
    public static void print_rechteck(int iBreite) {  
        print_line(iBreite, '+', '-');  
        for(int j = 0; j < 4; ++j) {  
            print_line(iBreite, '|', ' ');  
        }  
        print_line(iBreite, '+', '-');  
    }  
  
    public static void main(String[] args) {  
        print_rechteck(10);  
        System.out.println("dies war das 1. Rechteck");  
        print_rechteck(15);  
        System.out.println("gleich kommt das Letzte");  
        print_rechteck(20);  
    }  
}
```

Beispiel (Fort.)

Speicher (Stack)



nach 2 weiteren
Methodenaufrufen ist
wieder print_rechteck aktiv

```
public class Rechteck7 {  
    public static void print_line( int iBreite, char cS, char cB) {  
        System.out.print(cS);  
        for(int i = 0; i < iBreite-2; ++i) {  
            System.out.print(cB);  
        }  
        System.out.println(cS);  
    }  
  
    public static void print_rechteck(int iBreite) {  
        print_line(iBreite, '+', '-');  
        for(int j = 0; j < 4; ++j) {  
            print_line(iBreite, '|', ' ');  
        }  
        print_line(iBreite, '+', '-');  
    }  
  
    public static void main(String[] args) {  
        print_rechteck(10);  
        System.out.println("dies war das 1. Rechteck");  
        print_rechteck(15);  
        System.out.println("gleich kommt das Letzte");  
        print_rechteck(20);  
    }  
}
```

Vorlesung 7/1

Parameter Übergabe

- es gibt unterschiedliche Arten der Parameterübergabe für imperative Programmiersprachen
- man unterscheidet
 - call-by-value (z.B. Ada, Algol 60, Java, C, C++)
 - call-by-reference (z.B. Fortran, C, C++)
 - call-by-name (z.B. Algol 60, C, C++)
 - call-by-result (z.B. Ada, Verilog)
 - call-by-value-result (z.B. Ada, Verilog)
- für nicht-imperative Programmiersprachen gibt es noch
 - lazy-evaluation (Haskell)
 - unification (Prolog)
 - constraints (Verallgemeinerung der Unification)

Call-by-Value

```
public static void doit(int i) {  
    i = 43;  
}  
public static void main(String[] args) {  
    int x = 13;  
    doit(x);  
}
```

- vor dem Methodenaufruf:
 - Wert des aktuellen Parameters ,x‘ wird ausgerechnet (evaluiert)
 - in den formalen Parameter ,i‘ kopiert
- während des Methodenaufrufs:
 - es wird nur mit ,i‘ gearbeitet
- nach dem Methodenaufruf
 - der formale Parameter ,i‘ verschwindet

Call-by-Reference

```
public static void doit(int i) {  
    i = 43;  
}  
  
public static void main(String[] args) {  
    int x = 13;  
    doit(x);  
}
```

- vor dem Methodenaufruf:
 - der aktuelle Parameter ,x‘ wird mit dem formalen Parameter ,i‘ verbunden
 - ,i‘ ist nur noch ein anderer Name für ,x‘
- während des Methodenaufrufs:
 - wenn immer mit ,i‘ gearbeitet wird, wird in Wirklichkeit mit ,x‘ gearbeitet (gelesen, beschrieben)
- nach dem Methodenaufruf
 - die Verbindung zwischen ,x‘ und ,i‘ wird aufgehoben
 - der formale Parameter ,i‘ verschwindet

Call-by-Name

```
public static void doit(int i) {  
    i = 43;  
}  
public static void main(String[] args) {  
    int x = 13;  
    doit(x);  
}
```

- vor dem Methodenaufruf:
 - der aktuelle Parameter ,x‘ ersetzt textuell den formalen Parameter ,i‘
 - während des Methodenaufrufs:
 - der aktuelle Parameter ,x‘ wird jedes Mal ausgewertet, wenn ,i‘ verwendet wird
 - ist ,x‘ ein komplexer Ausdruck, wird dieser Ausdruck immer wieder ausgerechnet
 - nach dem Methodenaufruf
 - es passiert nichts
- Ist der aktuelle Parameter eine einfache Variable (wie in diesem Fall), ist Call-by-Name identisch zu Call-by-Reference

Call-by-Result

```
public static void doit(int i) {  
    i = 43;  
}  
public static void main(String[] args) {  
    int x = 13;  
    doit(x);  
}
```

- vor dem Methodenaufruf:
 - der Speicherort des aktuellen Parameters ‚x‘ wird ausgerechnet und gemerkt
 - für den formalen Parameter ‚i‘ wird eine neue Variable angelegt
- während des Methodenaufrufs:
 - es wird nur mit dem formalen Parameter ‚i‘ gearbeitet
- nach dem Methodenaufruf
 - der Wert des formalen Parameters ‚i‘ wird in den aktuellen Parameter ‚x‘ kopiert

Call-by-Result ist ein reiner
Rückgabemechanismus

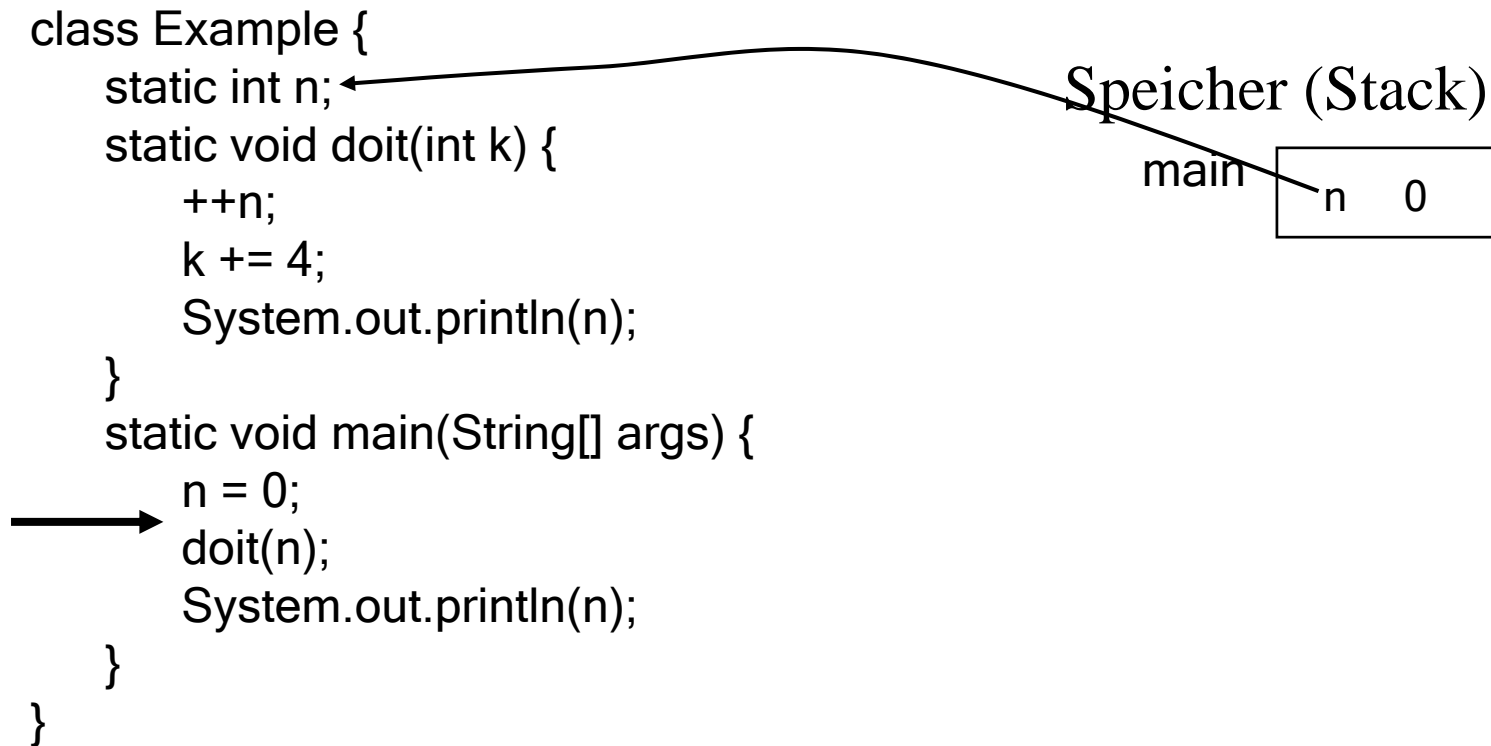
Call-by-Value-Result

```
public static void doit(int i) {  
    i = 43;  
}  
  
public static void main(String[] args) {  
    int x = 13;  
    doit(x);  
}
```

- vor dem Methodenaufruf:
 - der Speicherort des aktuellen Parameters ,x‘ wird ausgerechnet und gemerkt
 - für den formalen Parameter ,i‘ wird eine neue Variable angelegt
 - der Wert des aktuellen Parameters ,x‘ wird in die neue Variable ,i‘ kopiert
- während des Methodenaufrufs:
 - es wird nur mit dem formalen Parameter ,i‘ gearbeitet
- nach dem Methodenaufruf
 - der Wert des formalen Parameters ,i‘ wird in den aktuellen Parameter ,x‘ kopiert

Call-by-Value-Result ist ein Ein-Ausgabe-Mechanismus. Er kombiniert
P Call-by-Value und Call-by-Result.

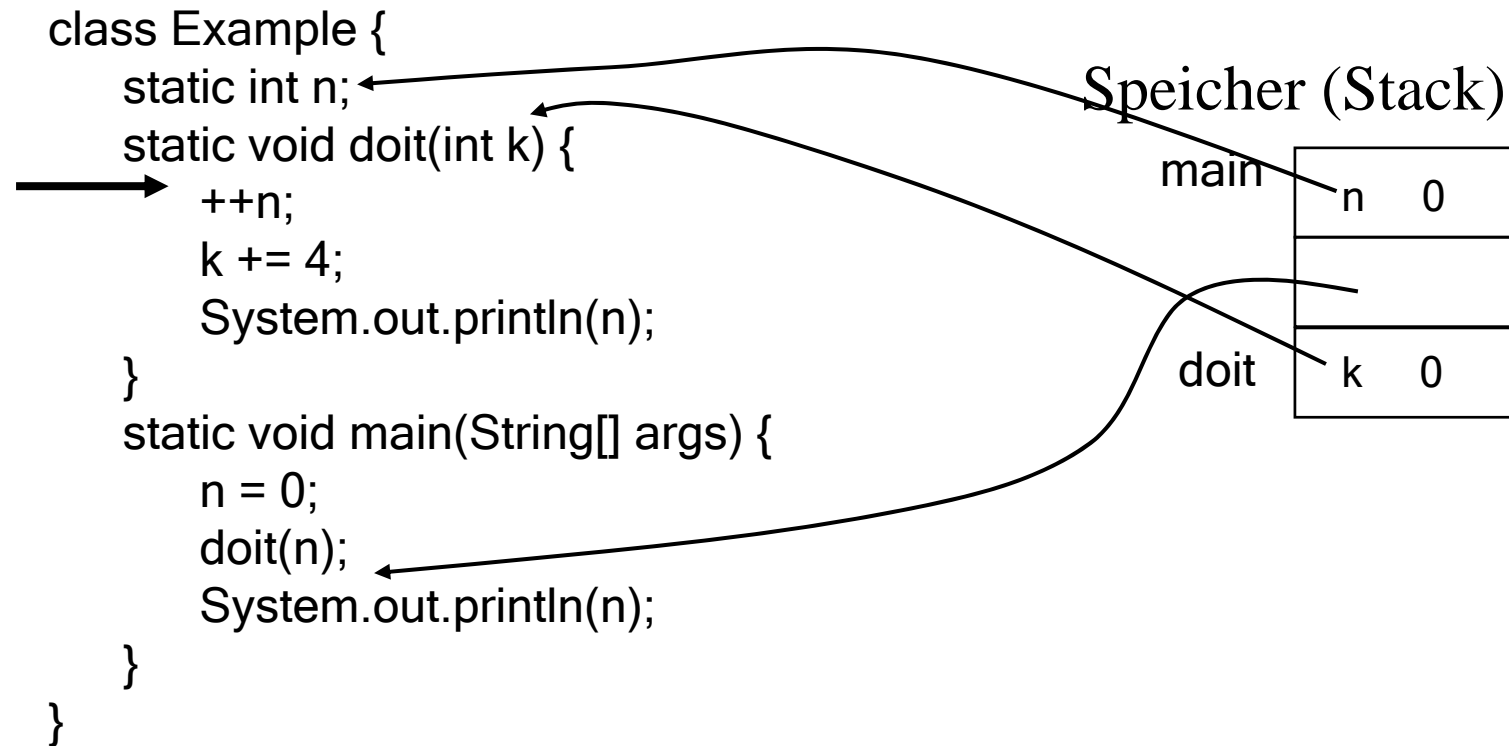
Unterschiede: Call-by-Value, -Value-Result, -Reference



- Call-by-Value:

- der Wert ,0‘ aus ,n‘ wird in die neue Variable ,k‘ kopiert
- ,k‘ verschwindet am Ende von ,doit‘

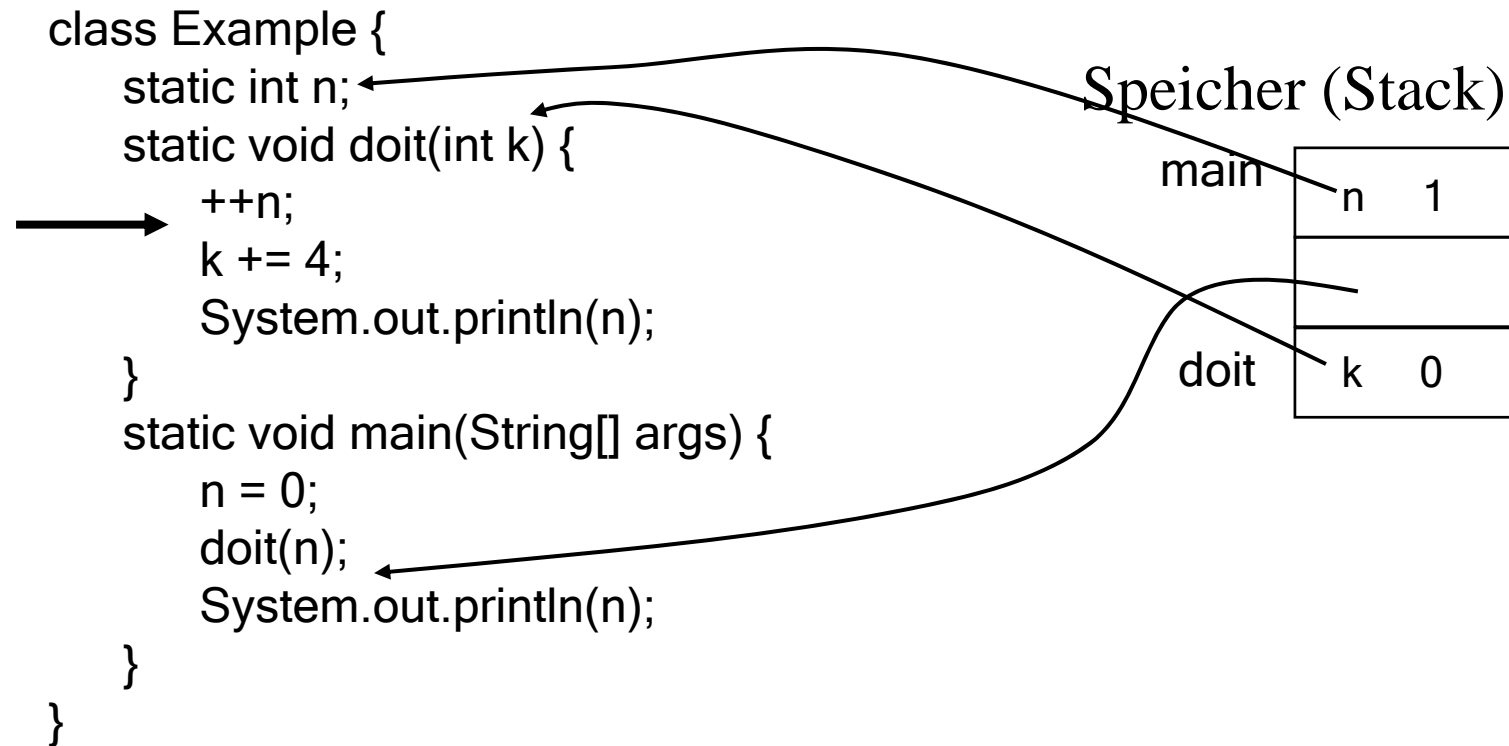
Unterschiede: Call-by-Value, -Value-Result, -Reference



- Call-by-Value:

- der Wert ,0‘ aus ,n‘ wird in die neue Variable ,k‘ kopiert
- ,k‘ verschwindet am Ende von ,doit‘

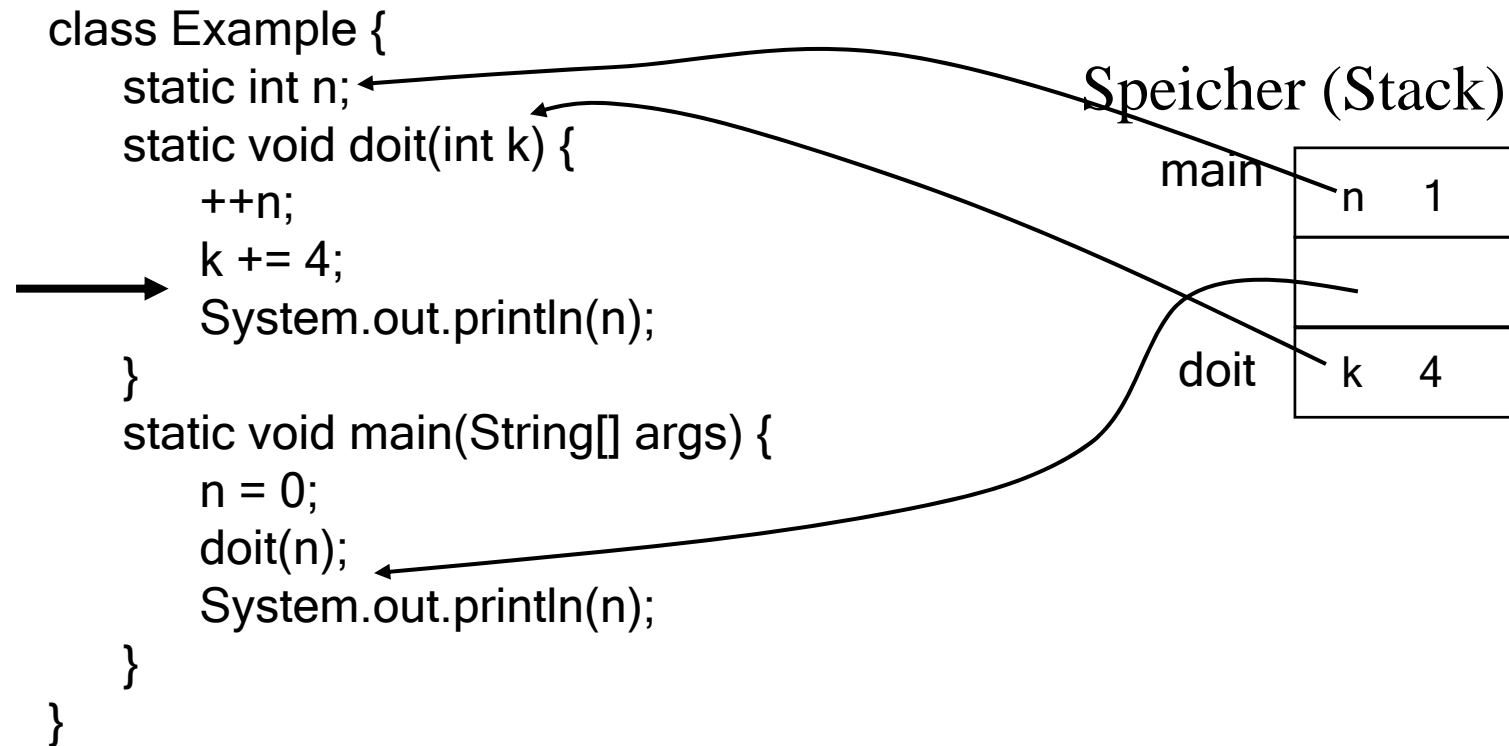
Unterschiede: Call-by-Value, -Value-Result, -Reference



- Call-by-Value:

- der Wert ,0‘ aus ,n‘ wird in die neue Variable ,k‘ kopiert
- ,k‘ verschwindet am Ende von ,doit‘

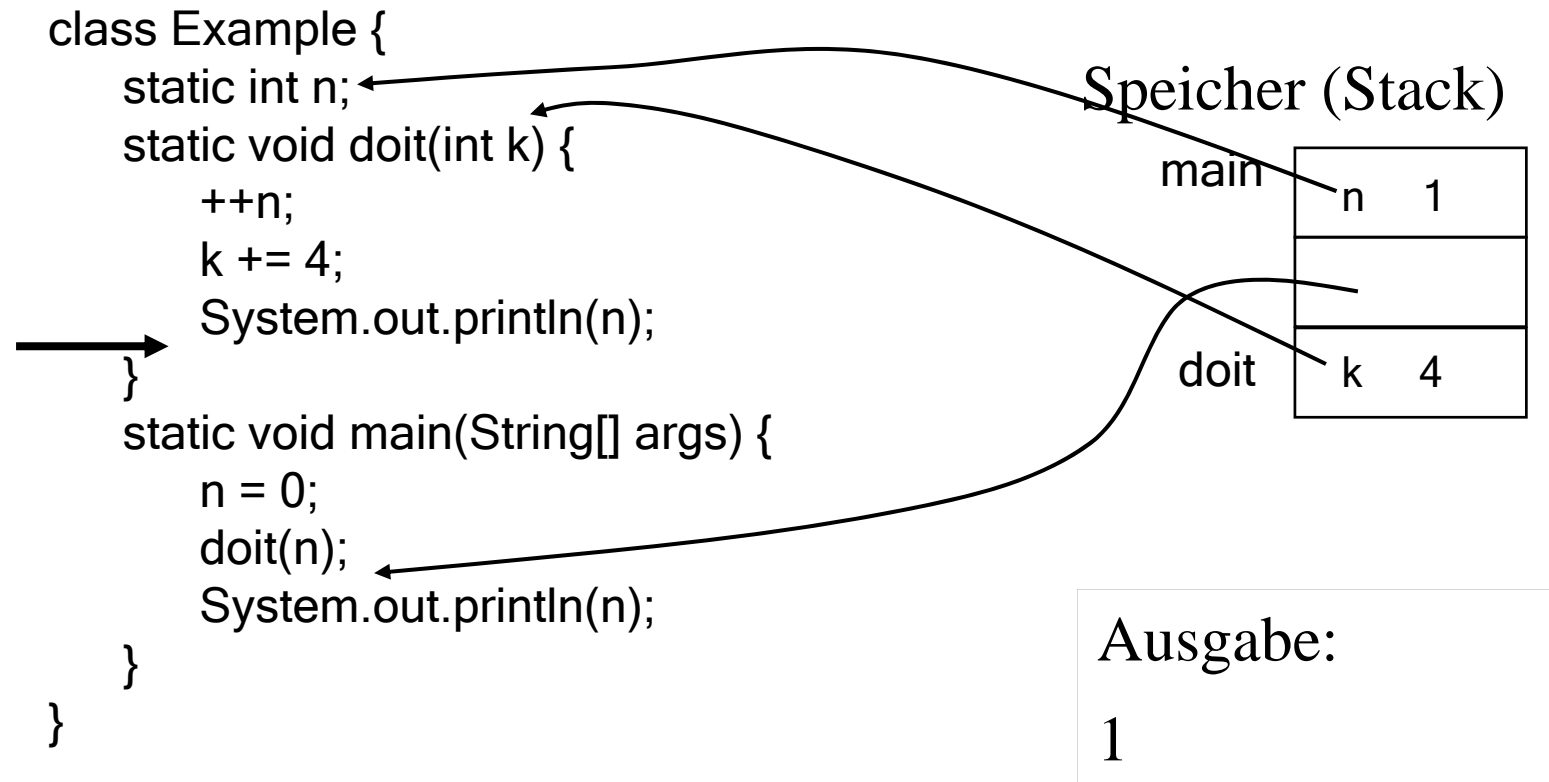
Unterschiede: Call-by-Value, -Value-Result, -Reference



- Call-by-Value:

- der Wert ,0‘ aus ,n‘ wird in die neue Variable ,k‘ kopiert
- ,k‘ verschwindet am Ende von ,doit‘

Unterschiede: Call-by-Value, -Value-Result, -Reference



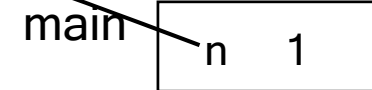
- **Call-by-Value:**

- der Wert ,0‘ aus ,n‘ wird in die neue Variable ,k‘ kopiert
- ,k‘ verschwindet am Ende von ,doit‘

Unterschiede: Call-by-Value, -Value-Result, -Reference

```
class Example {  
    static int n;  
    static void doit(int k) {  
        ++n;  
        k += 4;  
        System.out.println(n);  
    }  
    static void main(String[] args) {  
        n = 0;  
        doit(n);  
        System.out.println(n);  
    }  
}
```

Speicher (Stack)



Ausgabe:

1

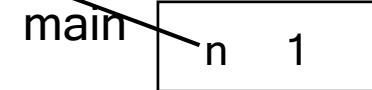
- Call-by-Value:

- der Wert ,0‘ aus ,n‘ wird in die neue Variable ,k‘ kopiert
- ,k‘ verschwindet am Ende von ,doit‘

Unterschiede: Call-by-Value, -Value-Result, -Reference

```
class Example {  
    static int n;  
    static void doit(int k) {  
        ++n;  
        k += 4;  
        System.out.println(n);  
    }  
    static void main(String[] args) {  
        n = 0;  
        doit(n);  
        System.out.println(n);  
    }  
}
```

Speicher (Stack)



Ausgabe:

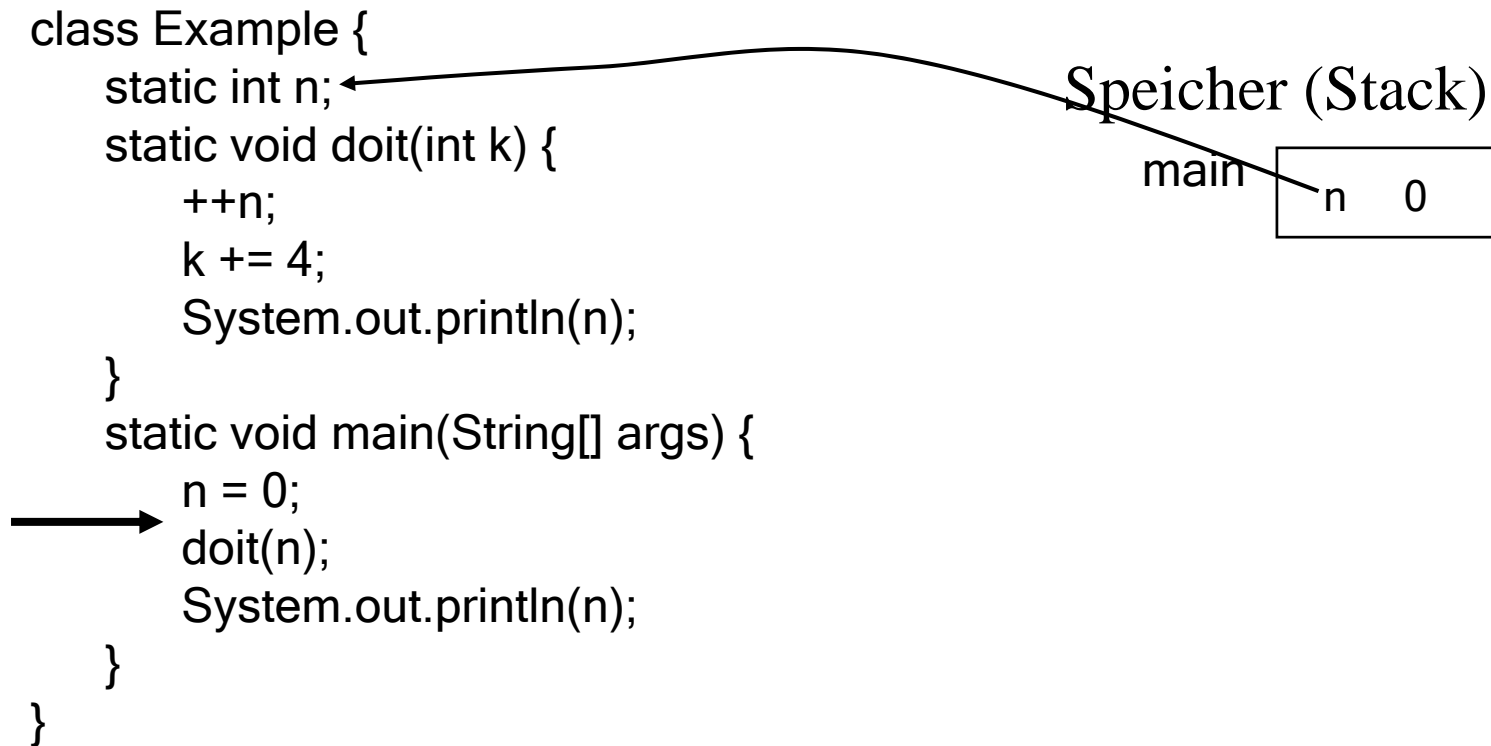
1

1

- Call-by-Value:

- der Wert ,0‘ aus ,n‘ wird in die neue Variable ,k‘ kopiert
- ,k‘ verschwindet am Ende von ,doit‘

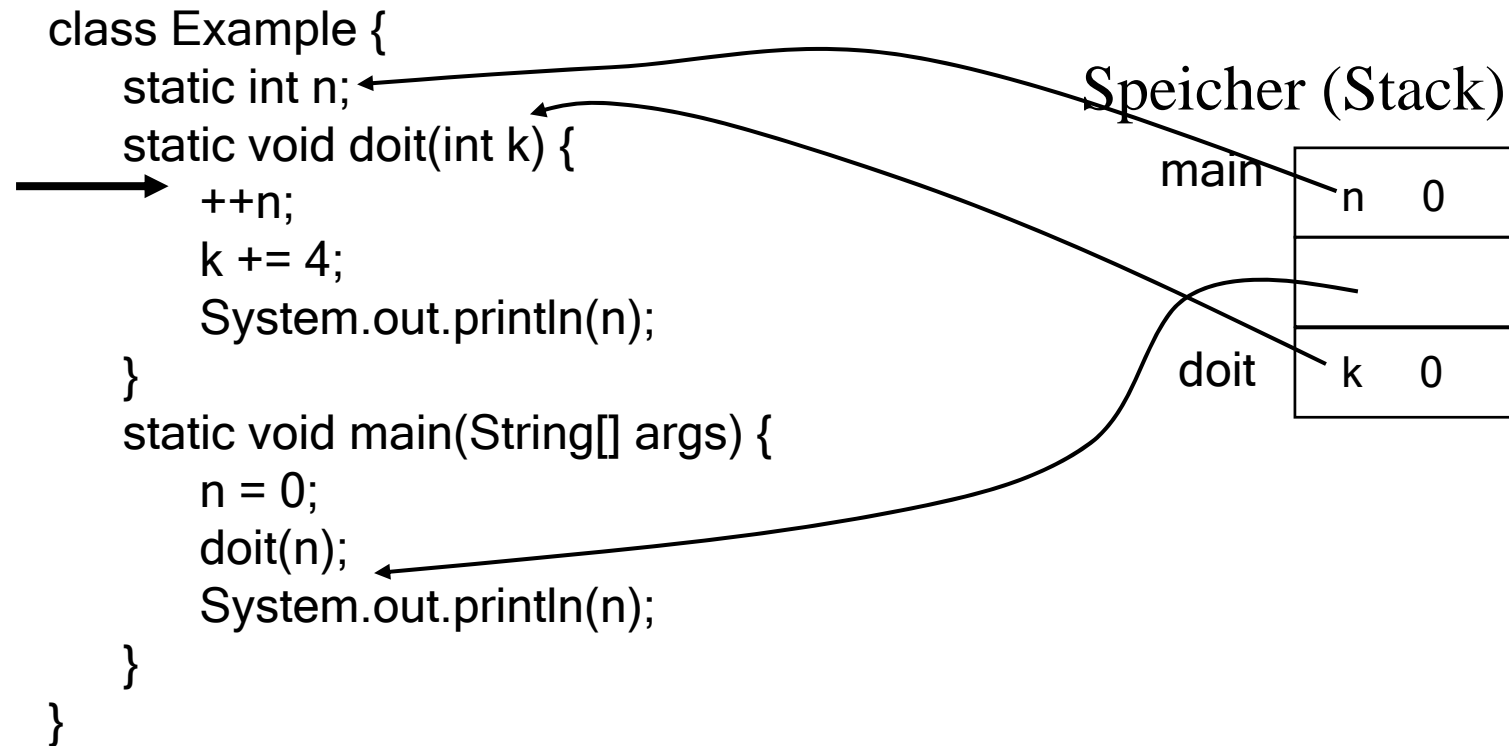
Unterschiede: Call-by-Value, -Value-Result, -Reference



- Call-by-Value-Result:

- der Wert ,0‘ aus ,n‘ wird in die neue Variable ,k‘ kopiert
- der Wert von ,k‘ wird am Ende von ,doit‘ wieder in ,n‘ kopiert

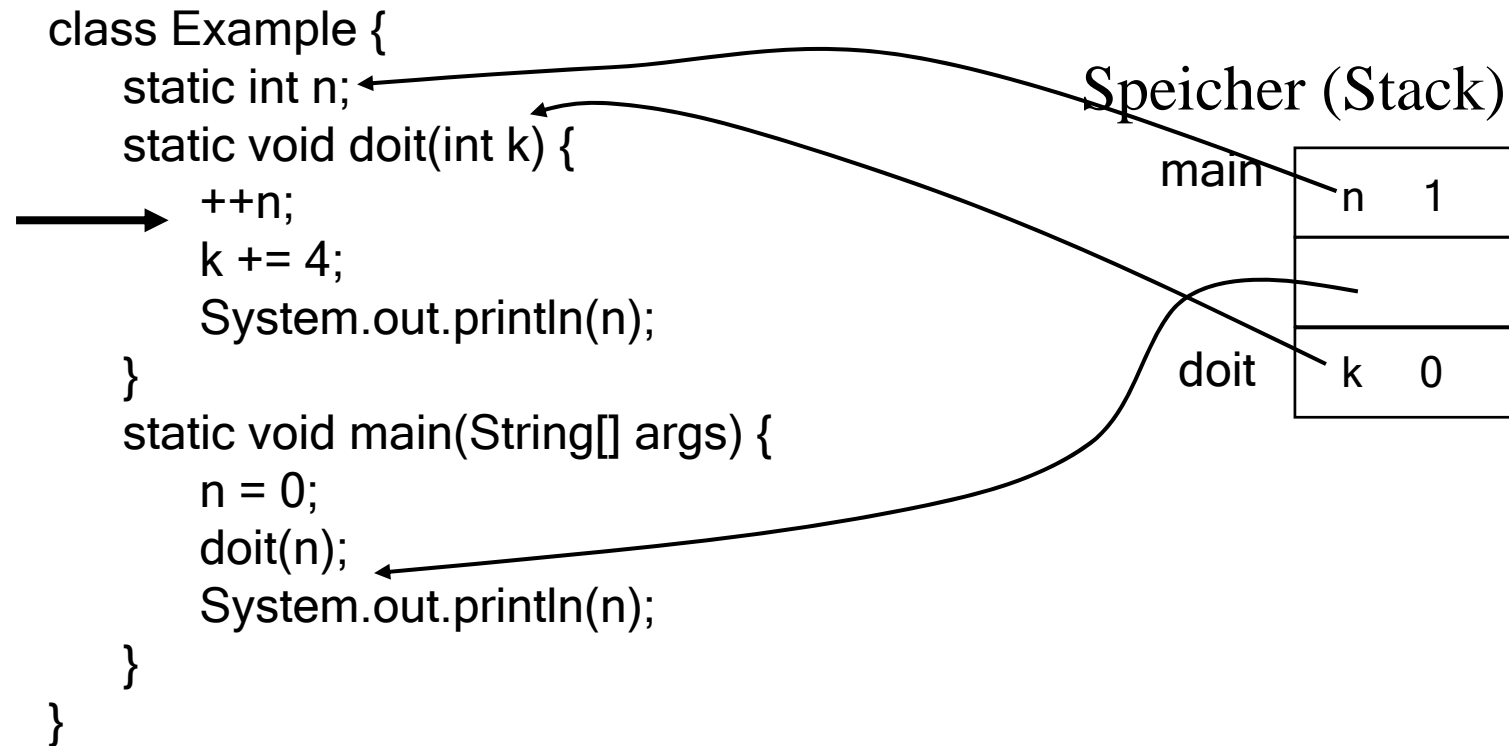
Unterschiede: Call-by-Value, -Value-Result, -Reference



- Call-by-Value-Result:

- der Wert ‚0‘ aus ‚n‘ wird in die neue Variable ‚k‘ kopiert
- der Wert von ‚k‘ wird am Ende von ‚doit‘ wieder in ‚n‘ kopiert

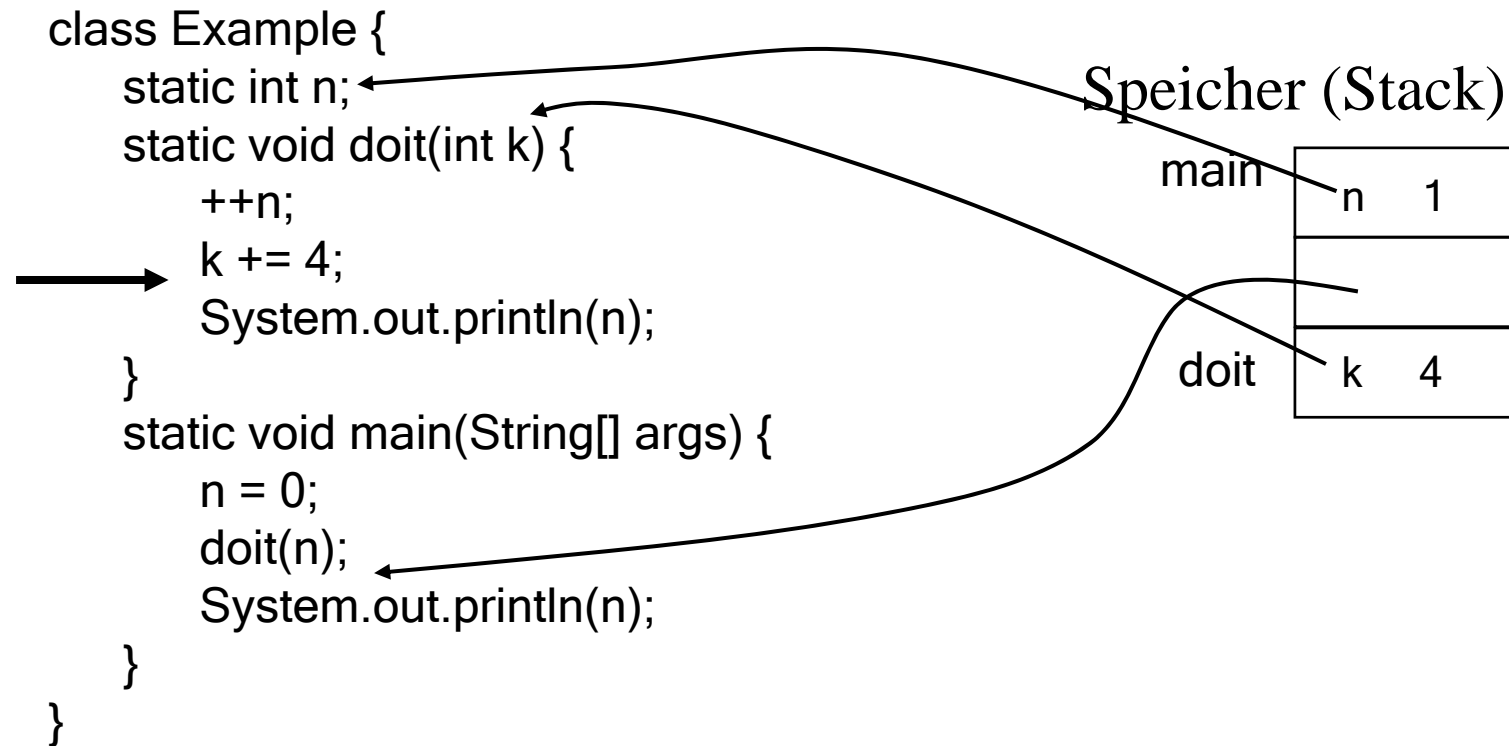
Unterschiede: Call-by-Value, -Value-Result, -Reference



- Call-by-Value-Result:

- der Wert ,0‘ aus ,n‘ wird in die neue Variable ,k‘ kopiert
- der Wert von ,k‘ wird am Ende von ,doit‘ wieder in ,n‘ kopiert

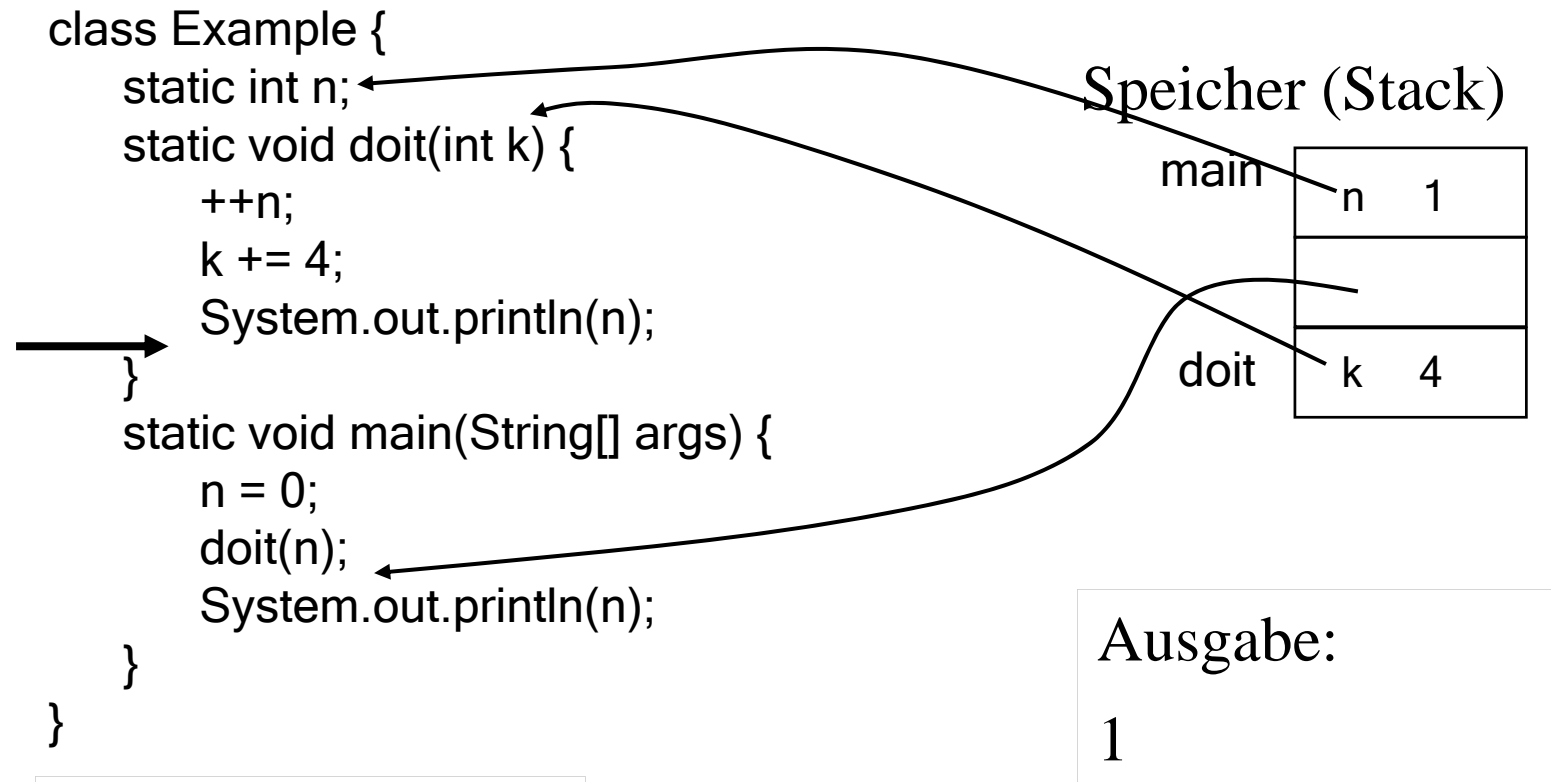
Unterschiede: Call-by-Value, -Value-Result, -Reference



- Call-by-Value-Result:

- der Wert ‚0‘ aus ‚n‘ wird in die neue Variable ‚k‘ kopiert
- der Wert von ‚k‘ wird am Ende von ‚doit‘ wieder in ‚n‘ kopiert

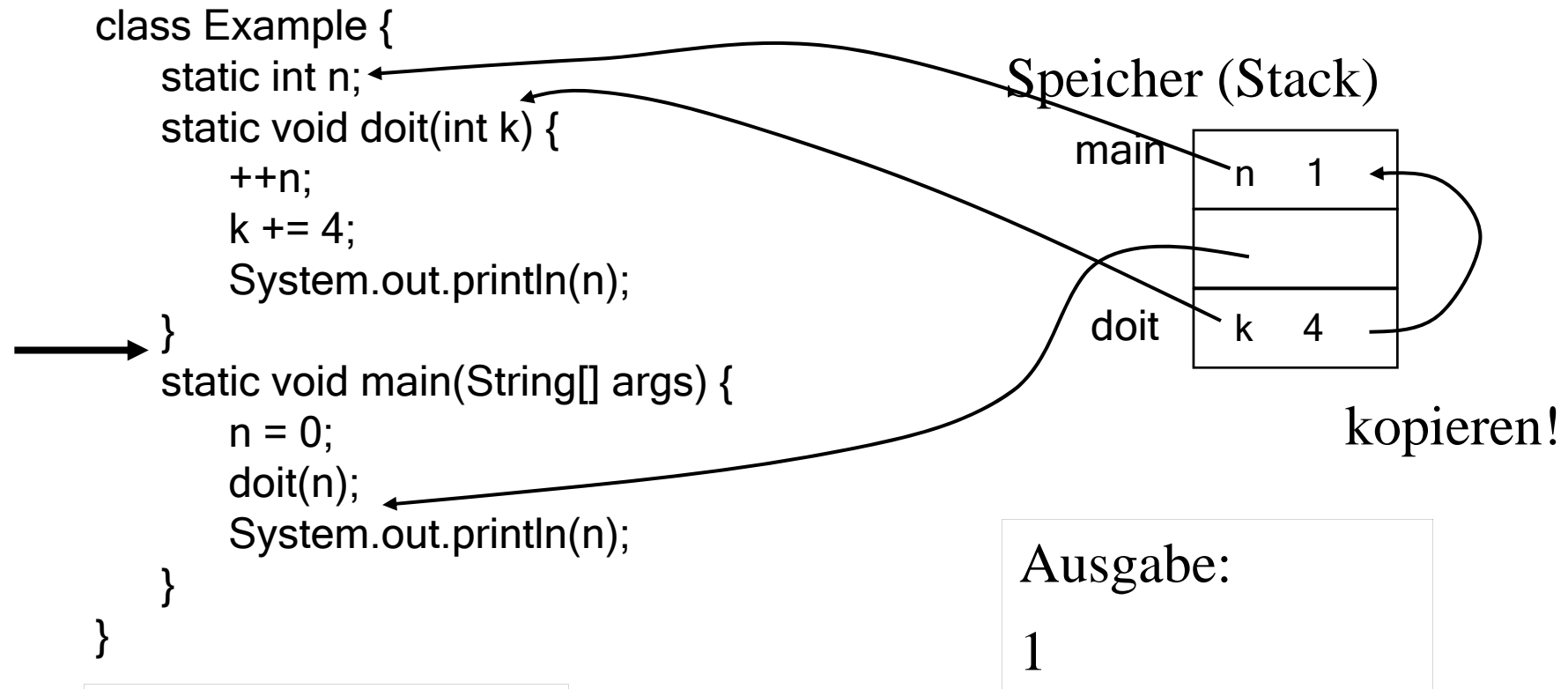
Unterschiede: Call-by-Value, -Value-Result, -Reference



- Call-by-Value-Result:

- der Wert ,0‘ aus ,n‘ wird in die neue Variable ,k‘ kopiert
- der Wert von ,k‘ wird am Ende von ,doit‘ wieder in ,n‘ kopiert

Unterschiede: Call-by-Value, -Value-Result, -Reference



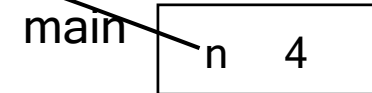
- Call-by-Value-Result:

- der Wert ,0‘ aus ,n‘ wird in die neue Variable ,k‘ kopiert
- der Wert von ,k‘ wird am Ende von ,doit‘ wieder in ,n‘ kopiert

Unterschiede: Call-by-Value, -Value-Result, -Reference

```
class Example {  
    static int n;  
    static void doit(int k) {  
        ++n;  
        k += 4;  
        System.out.println(n);  
    }  
    static void main(String[] args) {  
        n = 0;  
        doit(n);  
        System.out.println(n);  
    }  
}
```

Speicher (Stack)



,n' hat sich
geändert!

Ausgabe:

1

- Call-by-Value-Result:

- der Wert ,0' aus ,n' wird in die neue Variable ,k' kopiert
- der Wert von ,k' wird am Ende von ,doit' wieder in ,n' kopiert

Unterschiede: Call-by-Value, -Value-Result, -Reference

```
class Example {  
    static int n;  
    static void doit(int k) {  
        ++n;  
        k += 4;  
        System.out.println(n);  
    }  
    static void main(String[] args) {  
        n = 0;  
        doit(n);  
        System.out.println(n);  
    }  
}
```

Speicher (Stack)

main

n	4
---	---

Ausgabe:

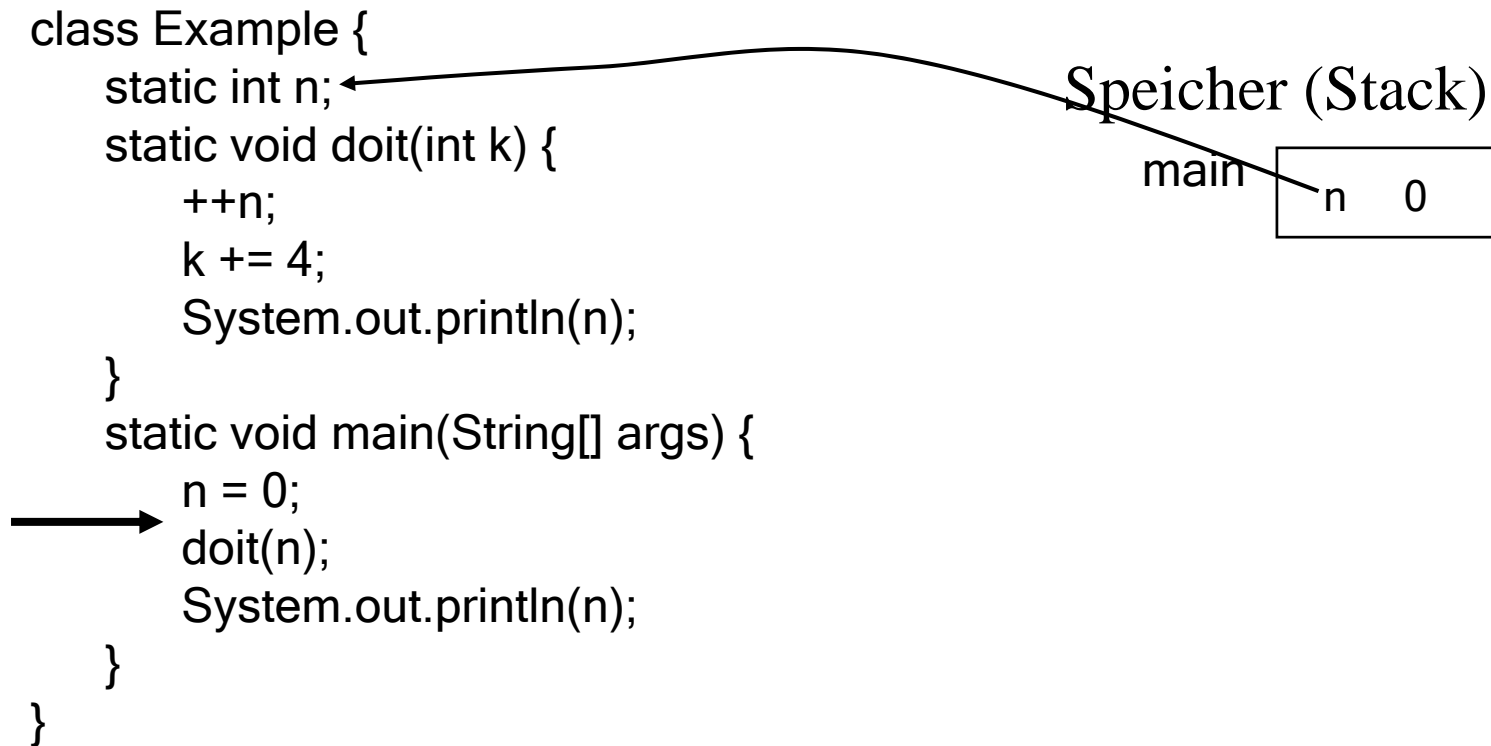
1

4

- Call-by-Value-Result:

- der Wert ,0‘ aus ,n‘ wird in die neue Variable ,k‘ kopiert
- der Wert von ,k‘ wird am Ende von ,doit‘ wieder in ,n‘ kopiert

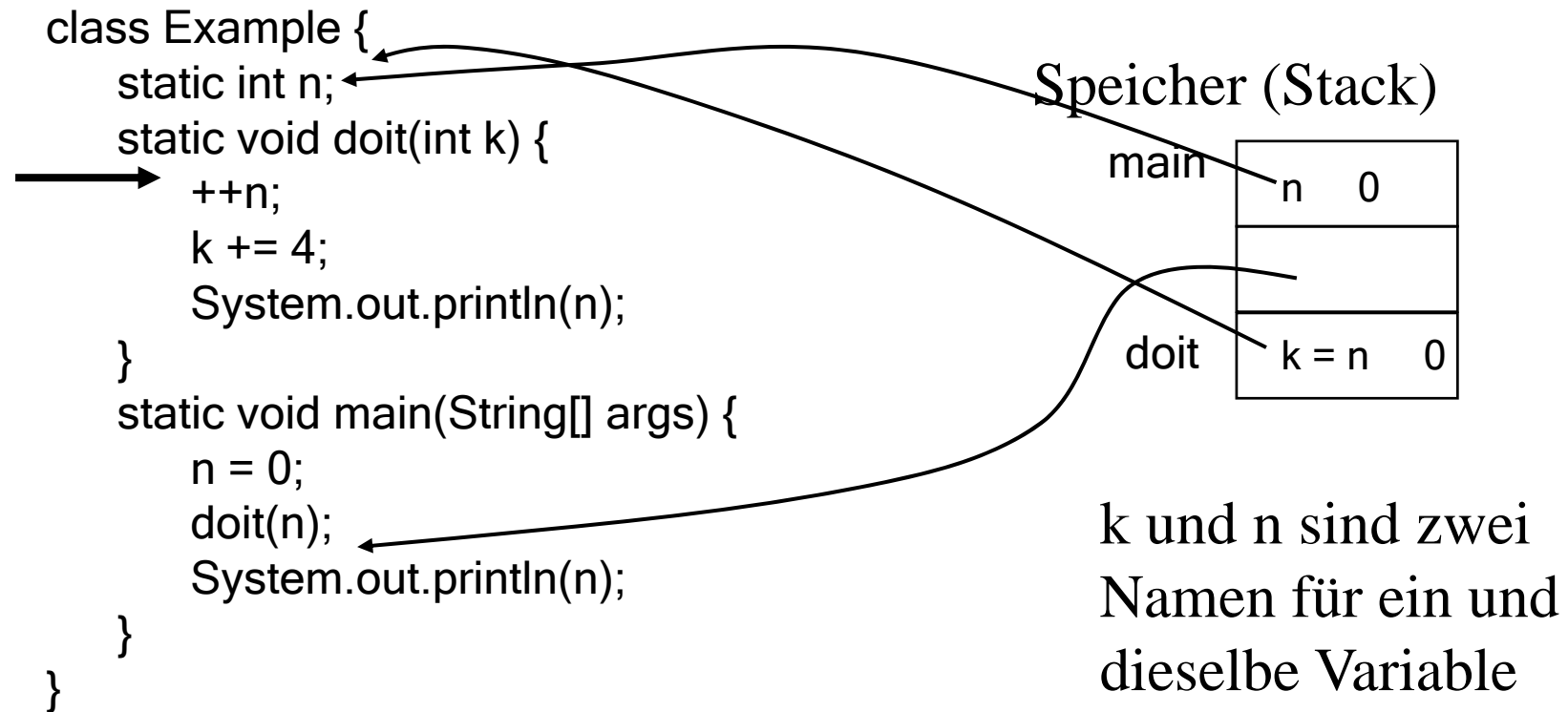
Unterschiede: Call-by-Value, -Value-Result, -Reference



- Call-by-Reference:

- ‚k‘ wird nur ein anderer Name für ‚n‘
- ‚k+=4‘ wird zu ‚n+=4‘

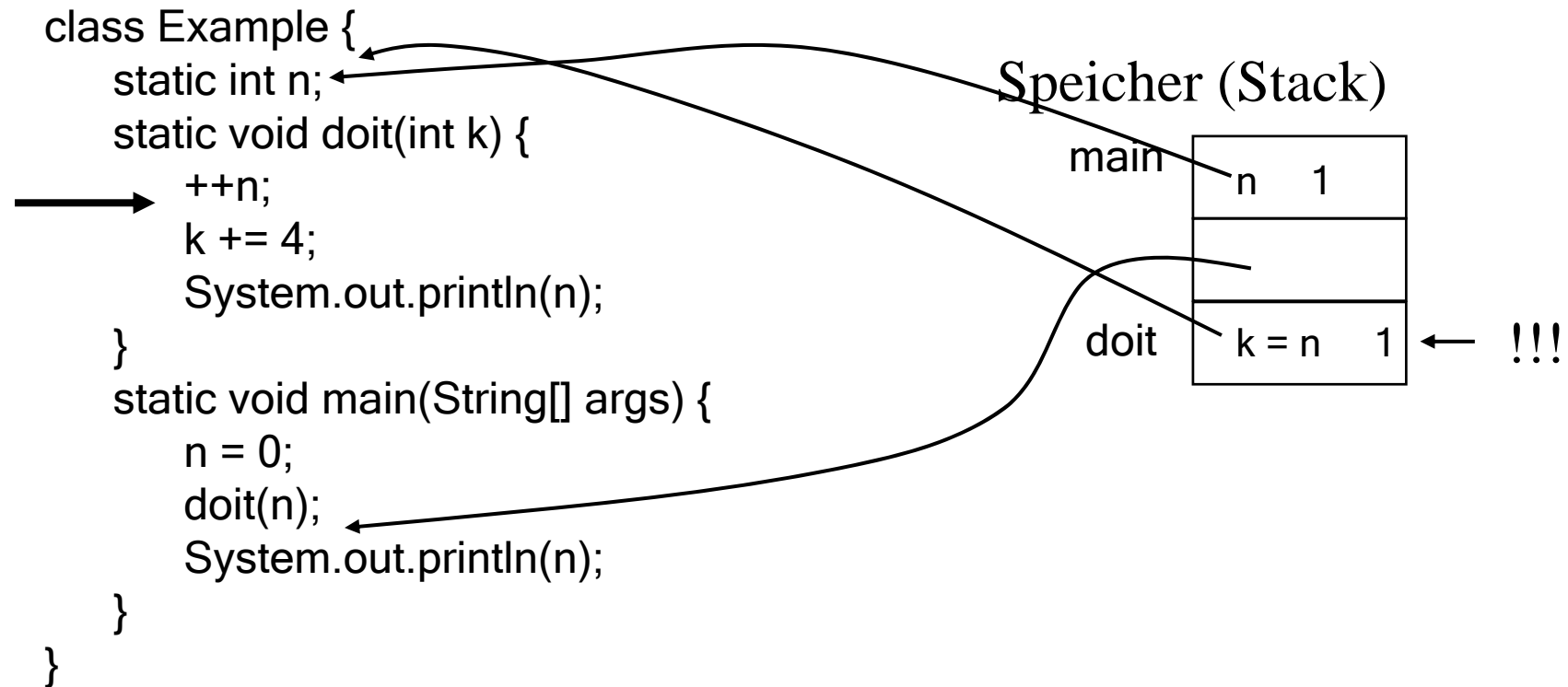
Unterschiede: Call-by-Value, -Value-Result, -Reference



- **Call-by-Reference:**

- ‚k‘ wird nur ein anderer Name für ‚n‘
- ‚k+=4‘ wird zu ‚n+=4‘

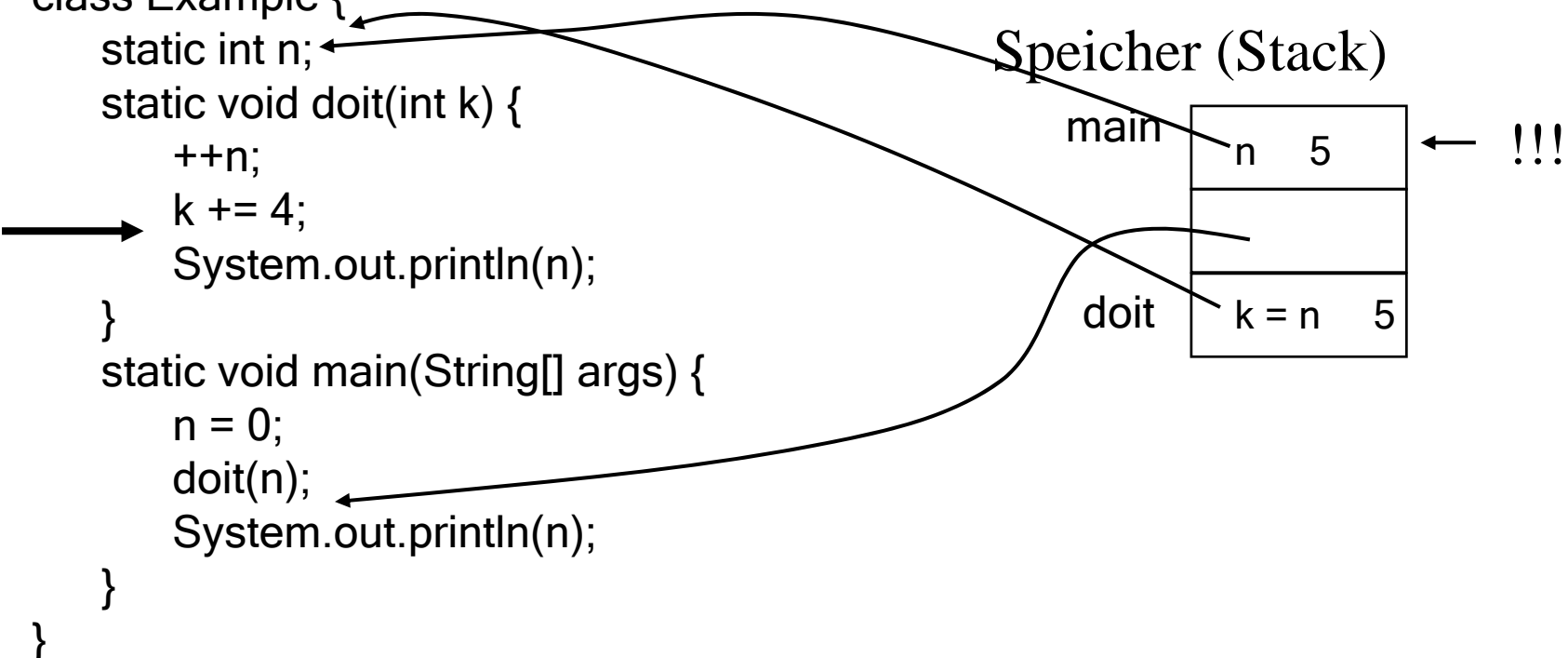
Unterschiede: Call-by-Value, -Value-Result, -Reference



- Call-by-Reference:

- ‚k‘ wird nur ein anderer Name für ‚n‘
- ‚k+=4‘ wird zu ‚n+=4‘

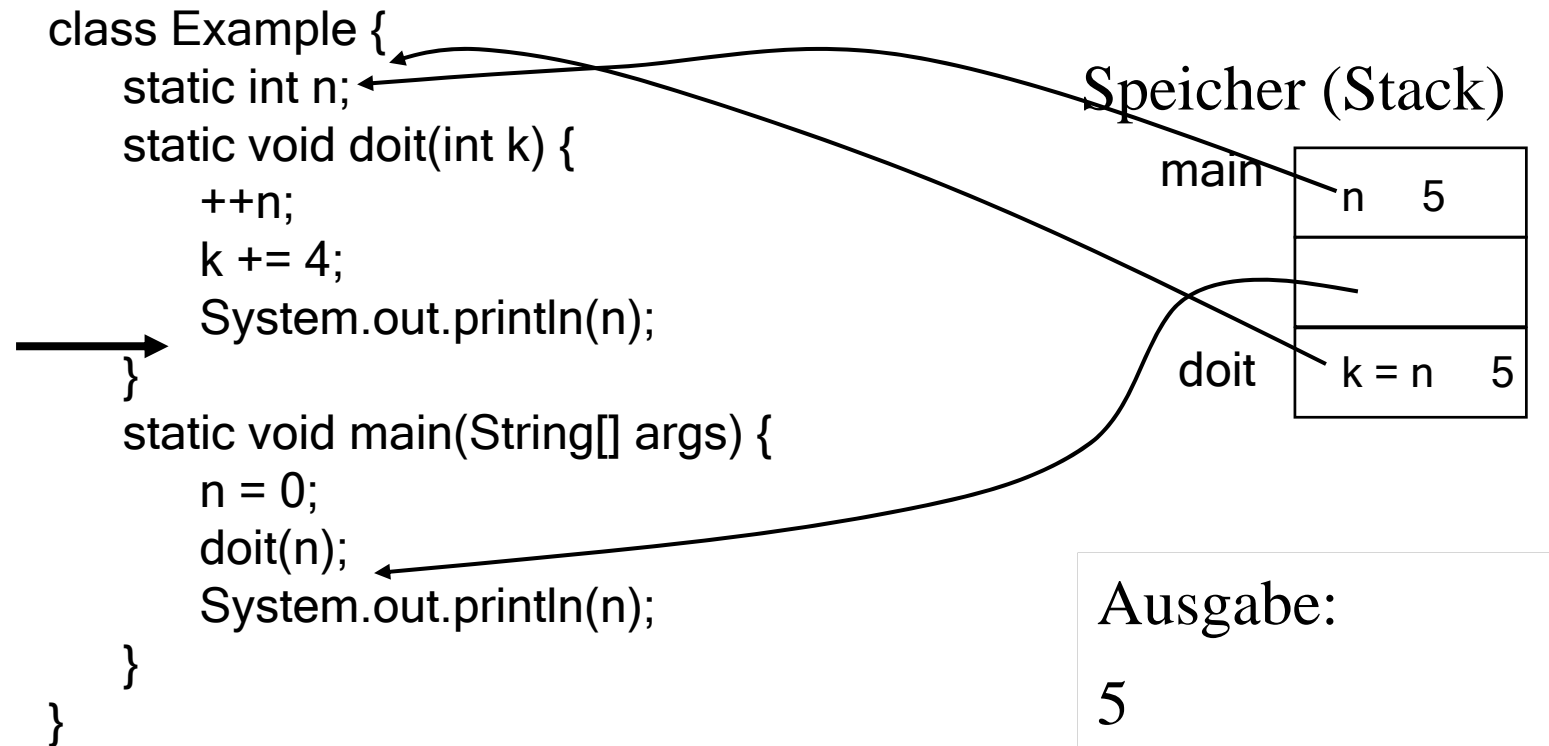
-Reference



Call-by-Reference:

- ‚k‘ wird nur ein anderer Name für ‚n‘
- ‚k+=4‘ wird zu ‚n+=4‘

Unterschiede: Call-by-Value, -Value-Result, -Reference



- Call-by-Reference:

- ‚k‘ wird nur ein anderer Name für ‚n‘
- ‚k+=4‘ wird zu ‚n+=4‘

Unterschiede: Call-by-Value, -Value-Result, -Reference

```
class Example {  
    static int n;  
    static void doit(int k) {  
        ++n;  
        k += 4;  
        System.out.println(n);  
    }  
    static void main(String[] args) {  
        n = 0;  
        doit(n);  
        System.out.println(n);  
    }  
}
```

Speicher (Stack)

main

n	5
---	---

Ausgabe:

5

- Call-by-Reference:

- ‚k‘ wird nur ein anderer Name für ‚n‘
- ‚k+=4‘ wird zu ‚n+=4‘

Unterschiede: Call-by-Value, -Value-Result, -Reference

```
class Example {  
    static int n;  
    static void doit(int k) {  
        ++n;  
        k += 4;  
        System.out.println(n);  
    }  
    static void main(String[] args) {  
        n = 0;  
        doit(n);  
        System.out.println(n);  
    }  
}
```

Speicher (Stack)

main

n	5
---	---

Ausgabe:

5

5

- Call-by-Reference:

- ‚k‘ wird nur ein anderer Name für ‚n‘
- ‚k+=4‘ wird zu ‚n+=4‘

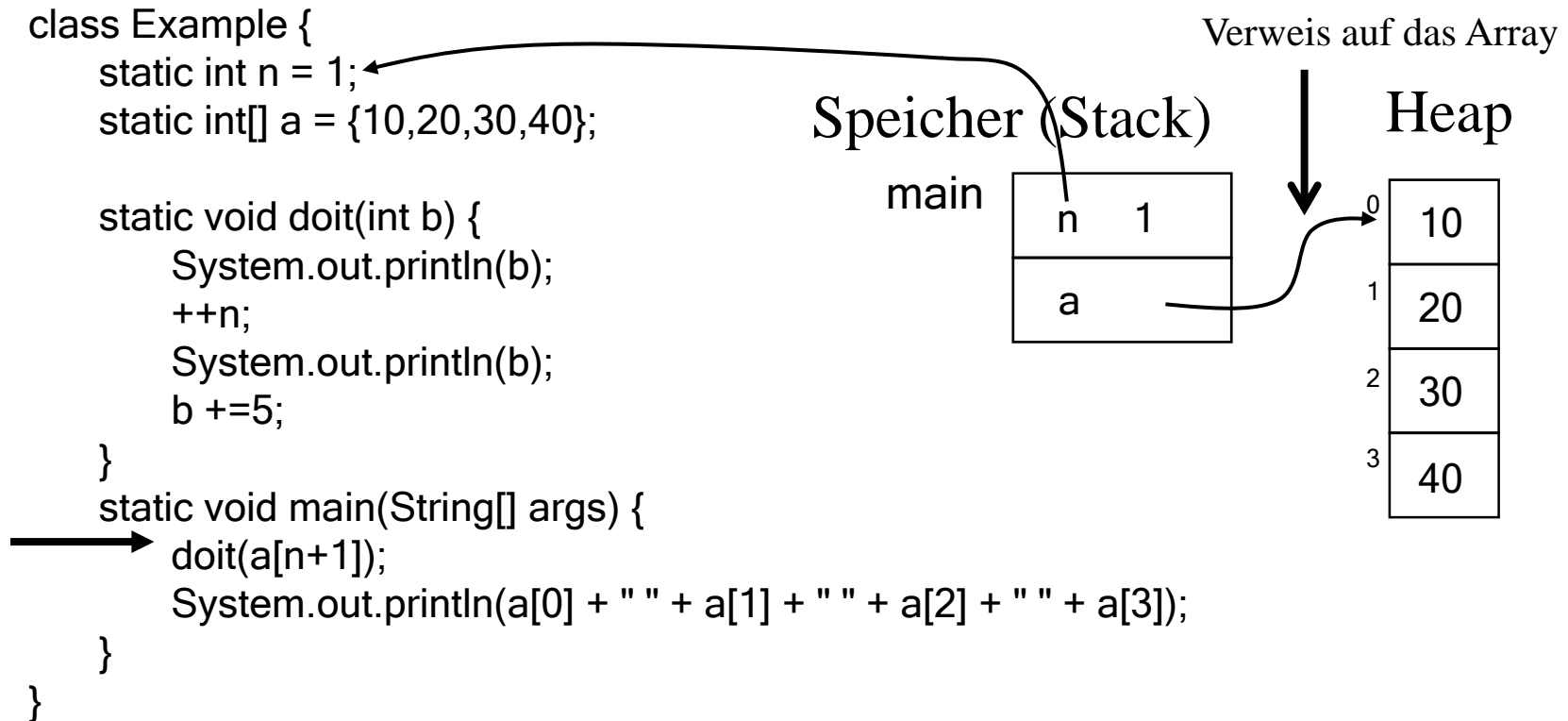
Vorlesung 7/2

Unterschiede: Call-by-Value, -Value-Result, -Reference

```
class Example {  
    static int n;  
    static void doit(int k) {  
        ++n;  
        k += 4;  
        System.out.println(n);  
    }  
    static void main(String[] args) {  
        n = 0;  
        doit(n);  
        System.out.println(n);  
    }  
}
```

Ausgabe		
Call-by-Value	Call-by-Value-Result	Call-by-Reference
1	1	5
1	4	5

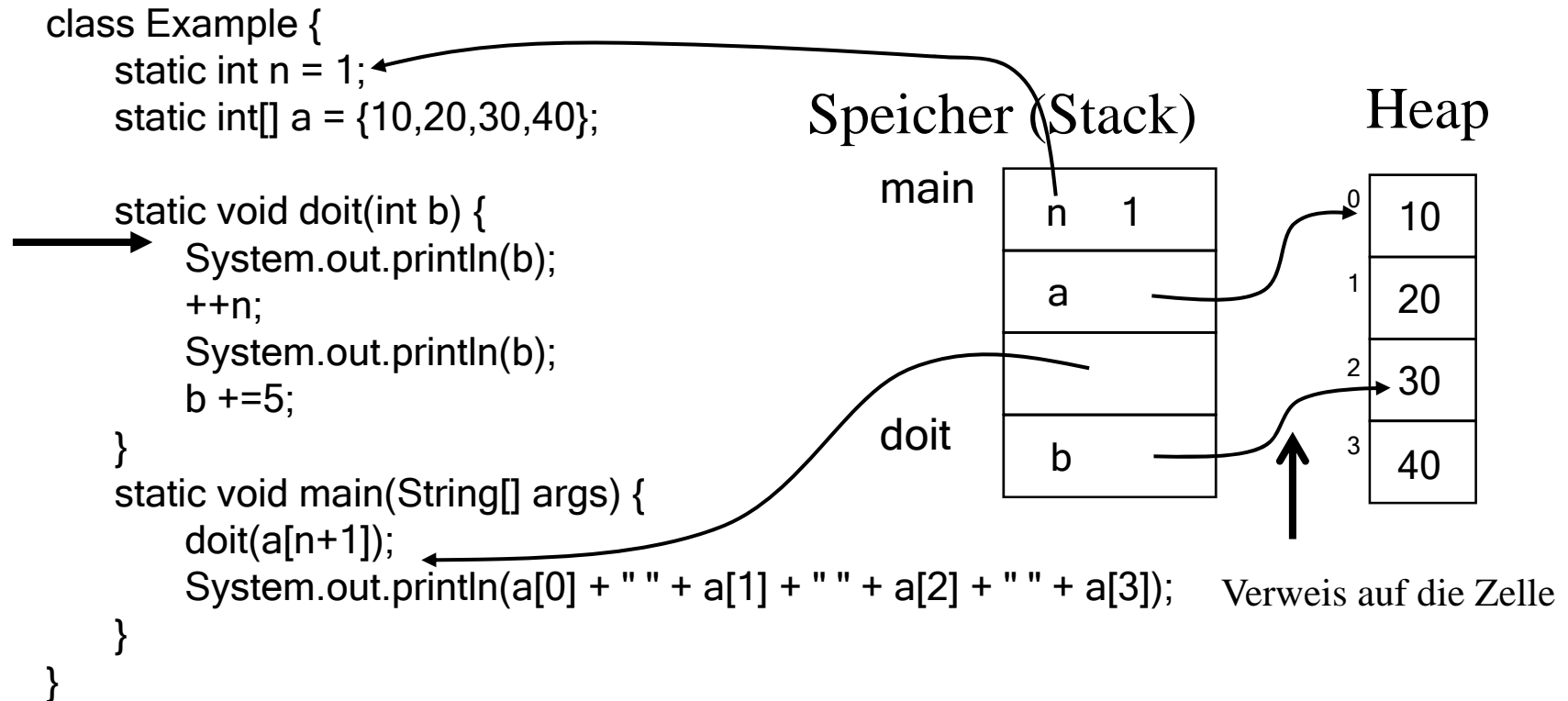
Unterschiede: Call-by-Reference und -Name



- Call-by-Reference:

- ‚a[n+1]‘ wird zu ‚a[2]‘ ausgewertet
- ‚b‘ wird ein anderer Name von a[2]
- ‚b‘ verschwindet am Ende von ‚doit‘

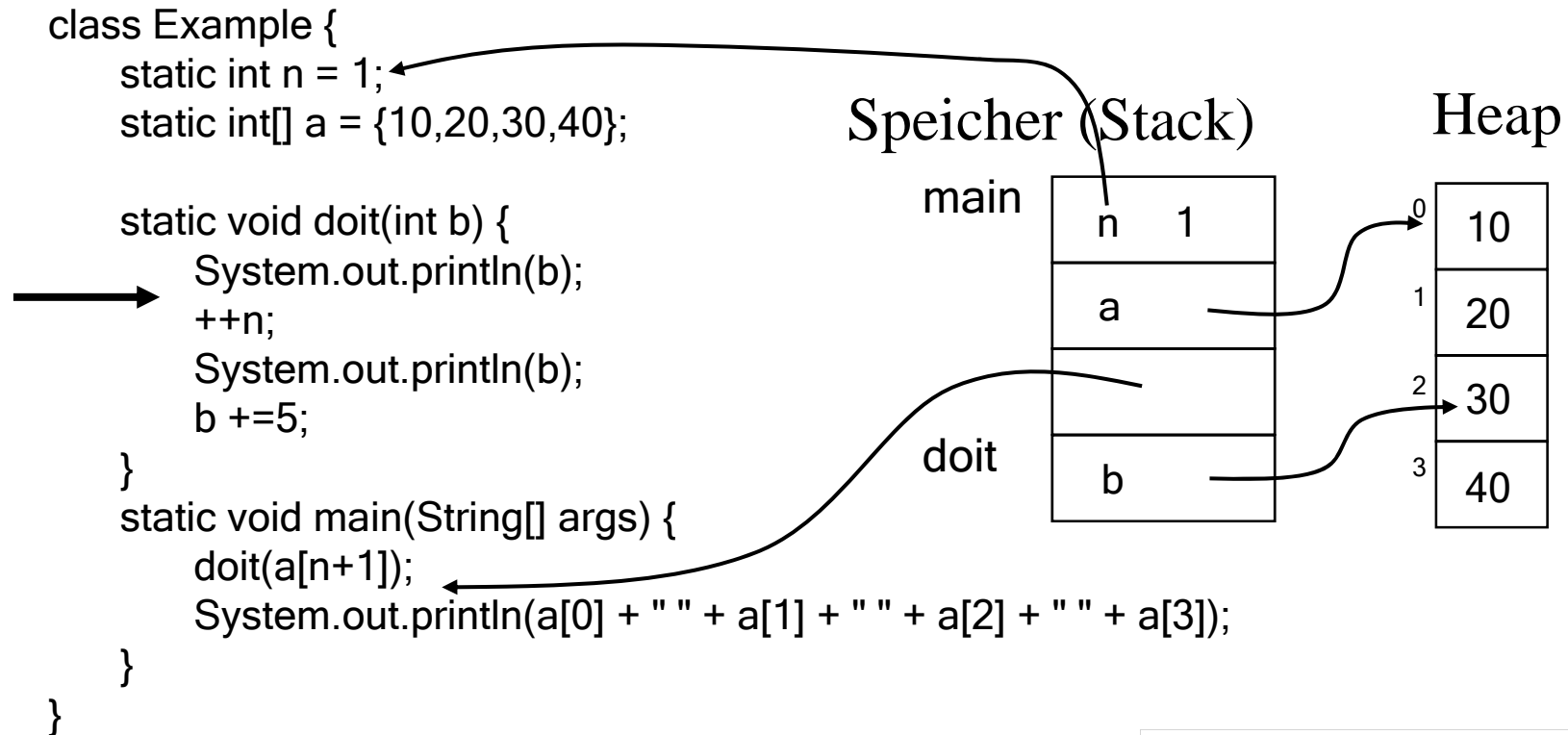
Unterschiede: Call-by-Reference und -Name



- Call-by-Reference:

- ‚a[n+1]‘ wird zu ‚a[2]‘ ausgewertet
- ‚b‘ wird ein anderer Name von a[2]
- ‚b‘ verschwindet am Ende von ‚doit‘

Unterschiede: Call-by-Reference und -Name



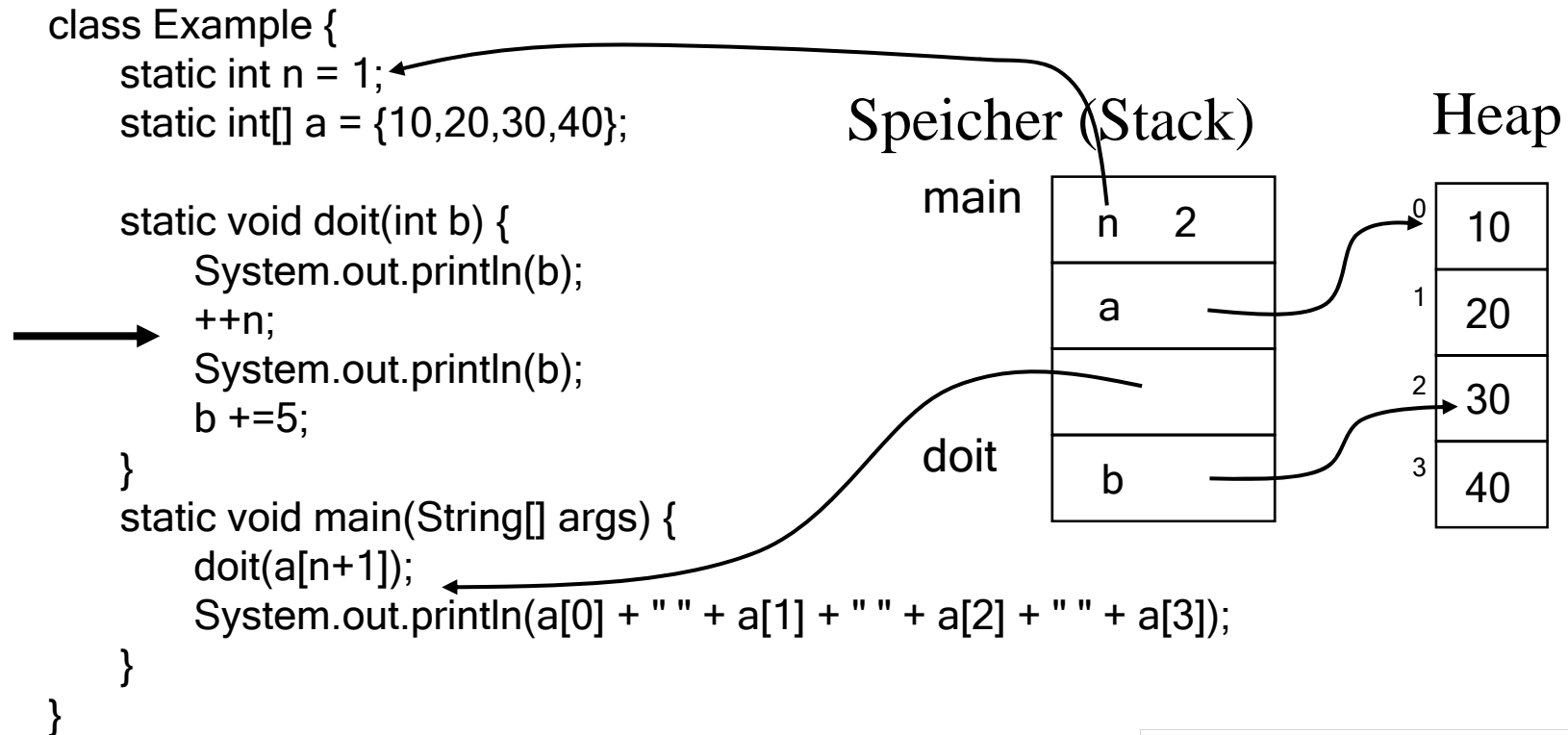
- **Call-by-Reference:**

- ‚a[n+1]‘ wird zu ‚a[2]‘ ausgewertet
- ‚b‘ wird ein anderer Name von a[2]
- ‚b‘ verschwindet am Ende von ‚doit‘

Ausgabe:

30

Unterschiede: Call-by-Reference und -Name



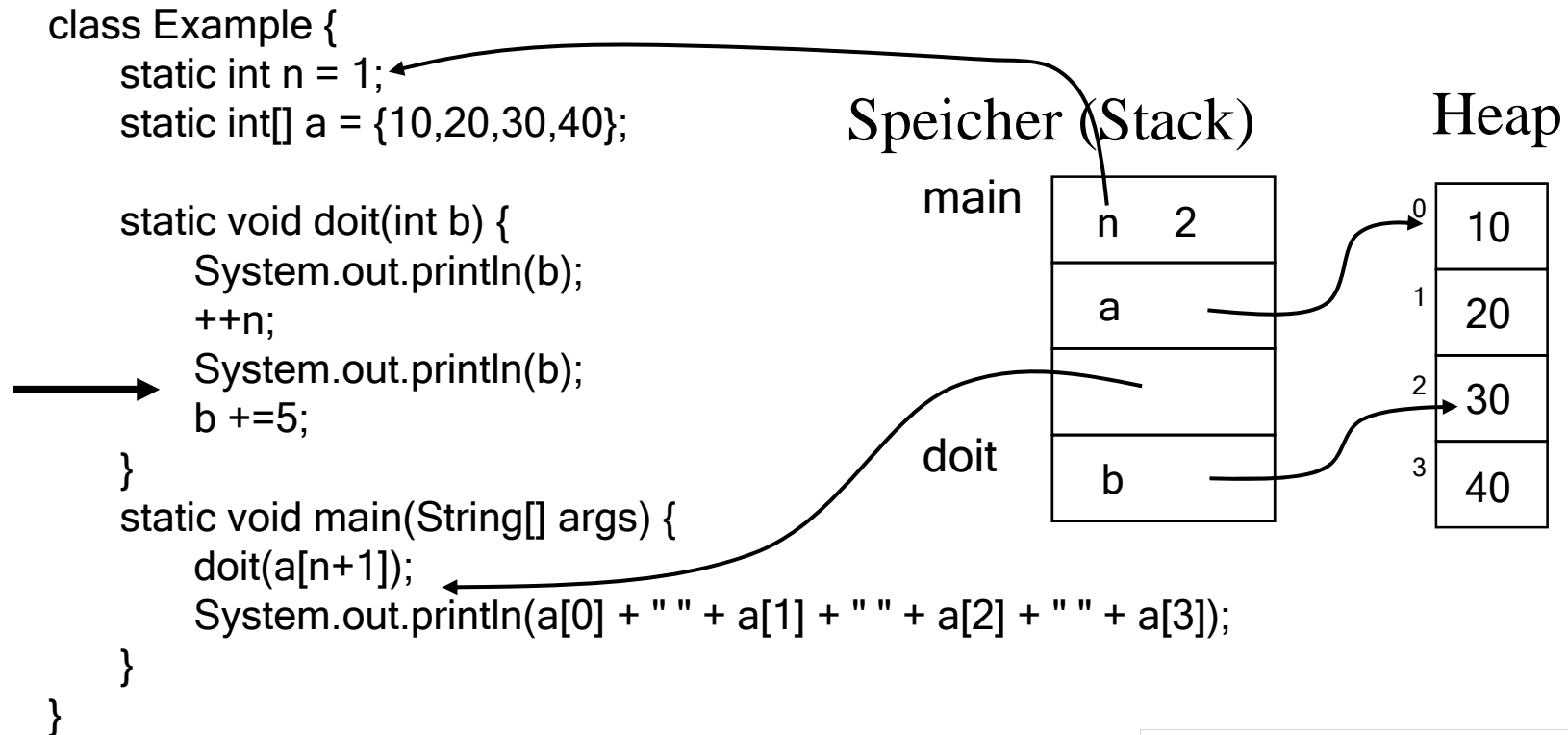
- **Call-by-Reference:**

- ‚a[n+1]‘ wird zu ‚a[2]‘ ausgewertet
- ‚b‘ wird ein anderer Name von a[2]
- ‚b‘ verschwindet am Ende von ‚doit‘

Ausgabe:

30

Unterschiede: Call-by-Reference und -Name



- Call-by-Reference:

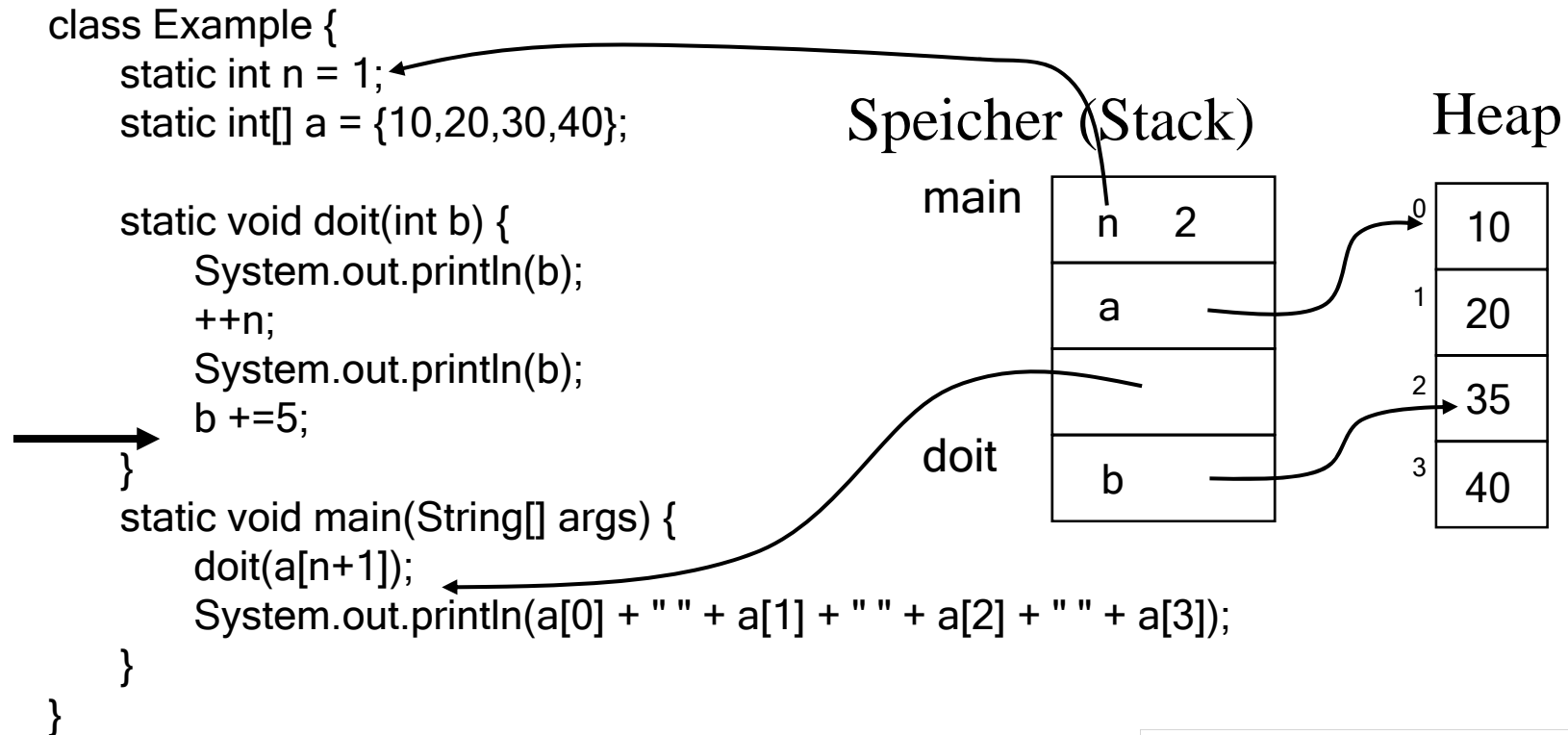
- ‚a[n+1]‘ wird zu ‚a[2]‘ ausgewertet
- ‚b‘ wird ein anderer Name von a[2]
- ‚b‘ verschwindet am Ende von ‚doit‘

Ausgabe:

30

30

Unterschiede: Call-by-Reference und -Name



- **Call-by-Reference:**

- ‚a[n+1]‘ wird zu ‚a[2]‘ ausgewertet
- ‚b‘ wird ein anderer Name von a[2]
- ‚b‘ verschwindet am Ende von ‚doit‘

Ausgabe:

30

30

Unterschiede: Call-by-Reference und -Name

```
class Example {  
    static int n = 1;  
    static int[] a = {10,20,30,40};
```

```
    static void doit(int b) {  
        System.out.println(b);  
        ++n;  
        System.out.println(b);  
        b +=5;
```

```
    }
```

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

```
        doit(a[n+1]);
```

```
        System.out.println(a[0] + " " + a[1] + " " + a[2] + " " + a[3]);
```

```
    }
```

```
}
```

Speicher (Stack)

main

n	2
a	

Heap

0	10
1	20
2	35
3	40

- Call-by-Reference:

- ‚a[n+1]‘ wird zu ‚a[2]‘ ausgewertet
- ‚b‘ wird ein anderer Name von a[2]
- ‚b‘ verschwindet am Ende von ‚doit‘

Ausgabe:

30

30

Unterschiede: Call-by-Reference und -Name

```
class Example {  
    static int n = 1;  
    static int[] a = {10,20,30,40};
```

```
    static void doit(int b) {  
        System.out.println(b);  
        ++n;  
        System.out.println(b);  
        b +=5;
```

```
    }  
    static void main(String[] args) {  
        doit(a[n+1]);  
        System.out.println(a[0] + " " + a[1] + " " + a[2] + " " + a[3]);  
    }  
}
```

Speicher (Stack)

main

n	2
a	

Heap

0	10
1	20
2	35
3	40

- Call-by-Reference:

- ‚a[n+1]‘ wird zu ‚a[2]‘ ausgewertet
- ‚b‘ wird ein anderer Name von a[2]
- ‚b‘ verschwindet am Ende von ‚doit‘

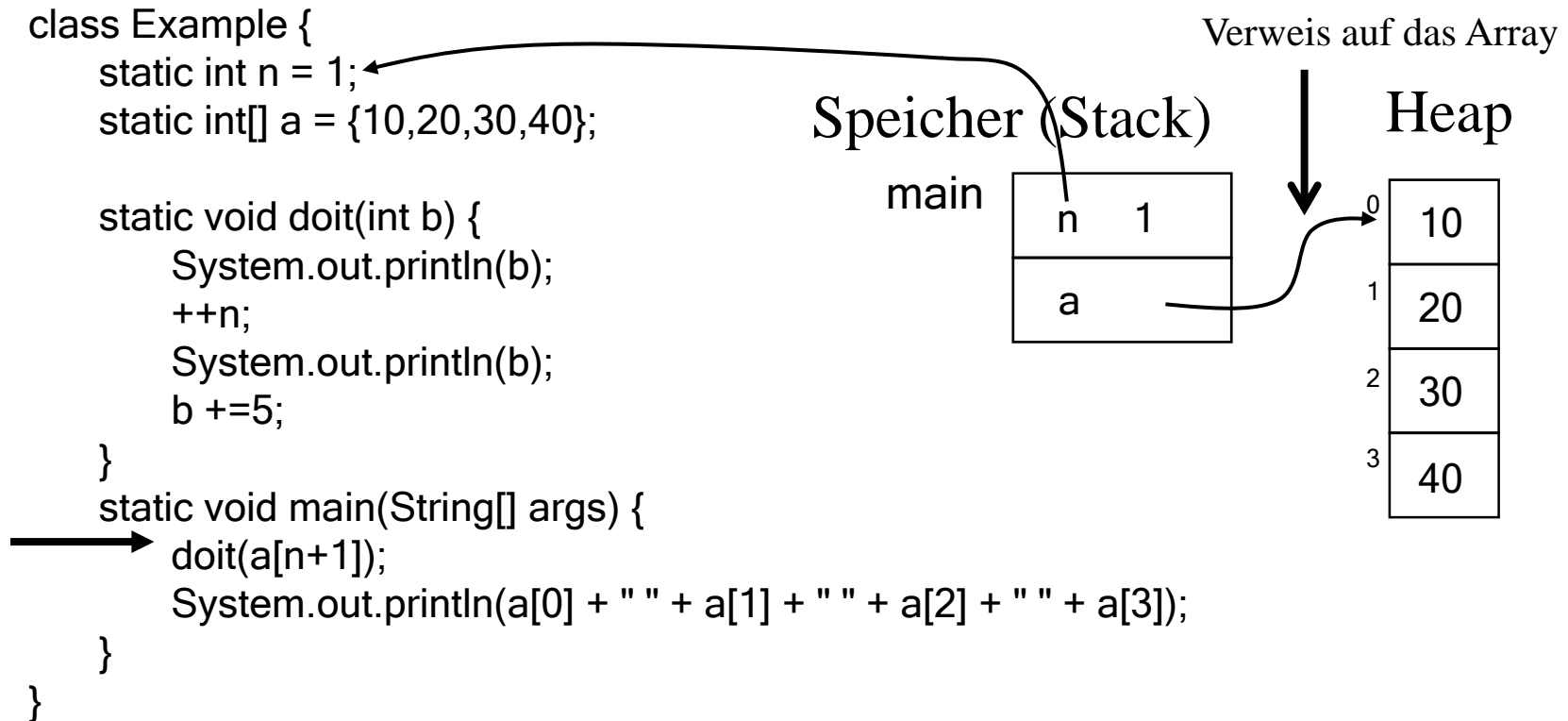
Ausgabe:

30

30

10 20 35 40

Unterschiede: Call-by-Reference und -Name



- Call-by-Name:

- der formale Parameter ,b‘ in ,doit‘ wird textuell durch ,a[n+1]‘ (nicht ,a[2]‘ !!!) ersetzt

Unterschiede: Call-by-Reference und -Name

```
class Example {  
    static int n = 1;  
    static int[] a = {10,20,30,40};
```

```
static void doit(int b) {  
    System.out.println(b);  
    ++n;  
    System.out.println(b);  
    b += 5;  
}
```

```
static void main(String[] args) {  
    doit(a[n+1]);  
    System.out.println(a[n+1]);  
    ++n;  
    System.out.println(a[n+1]);  
    a[n+1] += 5;  
    System.out.println(a[0] + " " + a[1] + " " + a[2] + " " + a[3]);  
}
```

Speicher (Stack)

main

n	1
a	

Heap

0	10
1	20
2	30
3	40

kopieren und ,b'
durch ,a[n+1]'
ersetzen

- Call-by-Name:

- der formale Parameter ,b' in ,doit' wird textuell durch ,a[n+1] (nicht ,a[2] !!!) ersetzt

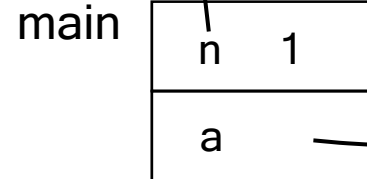
Unterschiede: Call-by-Reference und -Name

```
class Example {  
    static int n = 1;  
    static int[] a = {10,20,30,40};
```

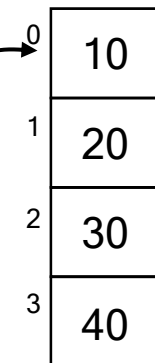
```
static void doit(int b) {  
    System.out.println(b);  
    ++n;  
    System.out.println(b);  
    b += 5;  
}
```

```
static void main(String[] args) {  
    doit(a[n+1]);  
    System.out.println(a[n+1]); ,a[n+1]‘ ist ,a[2]‘  
    ++n;  
    System.out.println(a[n+1]);  
    a[n+1] += 5;  
    System.out.println(a[0] + " " + a[1] + " " + a[2] + " " + a[3]);  
}
```

Speicher (Stack)



Heap



Ausgabe:

30

- Call-by-Name:

- der formale Parameter ,b‘ in ,doit‘ wird textuell durch ,a[n+1]‘ (nicht ,a[2]‘ !!!) ersetzt

Unterschiede: Call-by-Reference und -Name

```
class Example {  
    static int n = 1;  
    static int[] a = {10,20,30,40};
```

```
static void doit(int b) {  
    System.out.println(b);  
    ++n;  
    System.out.println(b);  
    b += 5;  
}
```

```
static void main(String[] args) {  
    doit(a[n+1]);  
    System.out.println(a[n+1]);  
    ++n;  
    System.out.println(a[n+1]);  
    a[n+1] += 5;  
    System.out.println(a[0] + " " + a[1] + " " + a[2] + " " + a[3]);  
}
```

Speicher (Stack)

main

n	2
a	

Heap

0	10
1	20
2	30
3	40

Ausgabe:

30

- Call-by-Name:

- der formale Parameter ,b‘ in ,doit‘ wird textuell durch ,a[n+1]‘ (nicht ,a[2]‘ !!!) ersetzt

Unterschiede: Call-by-Reference und -Name

```
class Example {  
    static int n = 1;  
    static int[] a = {10,20,30,40};
```

```
static void doit(int b) {  
    System.out.println(b);  
    ++n;  
    System.out.println(b);  
    b += 5;  
}
```

```
static void main(String[] args) {  
    doit(a[n+1]);  
    System.out.println(a[n+1]);  
    ++n;  
    System.out.println(a[n+1]); ,a[n+1] ist ,a[3]‘  
    a[n+1] += 5;  
    System.out.println(a[0] + " " + a[1] + " " + a[2] + " " + a[3]);  
}  
}
```

Speicher (Stack)

main

n	2
a	

Heap

0	10
1	20
2	30
3	40

Ausgabe:

30

40

- Call-by-Name:

- der formale Parameter ,b‘ in ,doit‘ wird textuell durch ,a[n+1]‘ (nicht ,a[2]‘ !!!) ersetzt

Unterschiede: Call-by-Reference und -Name

```
class Example {  
    static int n = 1;  
    static int[] a = {10,20,30,40};
```

```
static void doit(int b) {  
    System.out.println(b);  
    ++n;  
    System.out.println(b);  
    b += 5;  
}
```

```
static void main(String[] args) {  
    doit(a[n+1]);  
    System.out.println(a[n+1]);  
    ++n;  
    System.out.println(a[n+1]); ,a[n+1] ist ,a[3]‘  
    a[n+1] += 5;  
    System.out.println(a[0] + " " + a[1] + " " + a[2] + " " + a[3]);  
}  
}
```

Speicher (Stack)

main	n	2
	a	

Heap

0	10
1	20
2	30
3	45

Ausgabe:

30

40

- Call-by-Name:

- der formale Parameter ,b‘ in ,doit‘ wird textuell durch ,a[n+1]‘ (nicht ,a[2]‘ !!!) ersetzt

Unterschiede: Call-by-Reference und -Name

```
class Example {  
    static int n = 1;  
    static int[] a = {10,20,30,40};
```

```
static void doit(int b) {  
    System.out.println(b);  
    ++n;  
    System.out.println(b);  
    b += 5;  
}
```

```
static void main(String[] args) {  
    doit(a[n+1]);  
    System.out.println(a[n+1]);  
    ++n;  
    System.out.println(a[n+1]);  
    a[n+1] += 5;  
    System.out.println(a[0] + " " + a[1] + " " + a[2] + " " + a[3]);  
}
```

- Call-by-Name:

- der formale Parameter ,b‘ in ,doit‘ wird textuell durch ,a[n+1]‘ (nicht ,a[2]‘ !!!) ersetzt

Speicher (Stack)

main

n	2
a	

Heap

0	10
1	20
2	30
3	45

Ausgabe:

30

40

10 20 30 45

Unterschiede: Call-by-Reference und -Name

```
class Example {  
    static int n = 1;  
    static int[] a = {10,20,30,40};  
  
    static void doit(int b) {  
        System.out.println(b);  
        ++n;  
        System.out.println(b);  
        b +=5;  
    }  
    static void main(String[] args) {  
        doit(a[n+1]);  
        System.out.println(a[0] + " " + a[1] + " " + a[2] + " " + a[3]);  
    }  
}
```

Ausgabe	
Call-by-Reference	Call-by-Name
30	30
33	40
10 20 35 40	10 20 30 45

Java und Parameterübergabe

- Java kennt nur Call-by-Value
- C/C++ kennt Call-by-Value, -Reference, -Name
- manchmal findet man Aussagen der Form: „Call-by-Value ist bei imperativen Programmiersprachen vollkommen ausreichend. Call-by-Reference und Call-by-Name braucht man nicht.“
- Meine Meinung: dies ist Blödsinn !
- Call-by-Name: sehr schwer zu überblicken, sollte nicht eingesetzt werden
- Call-by-Reference: ist in jedem Fall sinnvoll, ohne dem sind Algorithmen manchmal schwer zu implementieren (siehe Bäume in Java und C++)

Vorlesung 8/1

Parameter

- bei einem Methodenaufruf werden die Werte der Ausdrücke an die Parameter übergeben
- der Wert einer int-Variablen ist die Zahl, die unter dieser Variablen im Speicher liegt

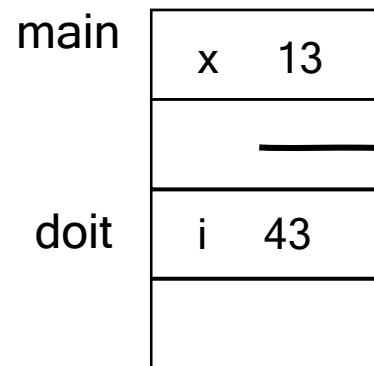
```
public static void doit(int i) {  
    i = 43;  
}  
  
public static void main(String[] args) {  
    int x = 13;  
    doit(x);  
    System.out.println(x);  
}
```

Was gibt dieses
Programm aus?
13 oder 43?

Parameter (Fort.)

die Ausgabe ist 13, weil ...

Speicher (Stack)



```
public static void doit(int i) {  
    i = 43;  
}
```

```
public static void main(String[] args) {  
    int x = 13;  
    doit(x);  
    System.out.println(x);  
}
```

... während des Methodenaufrufs es eine neue Variable *i* gibt die verändert wird, d.h. *x* von *main* wird nicht verändert

Parameter (Fort.)

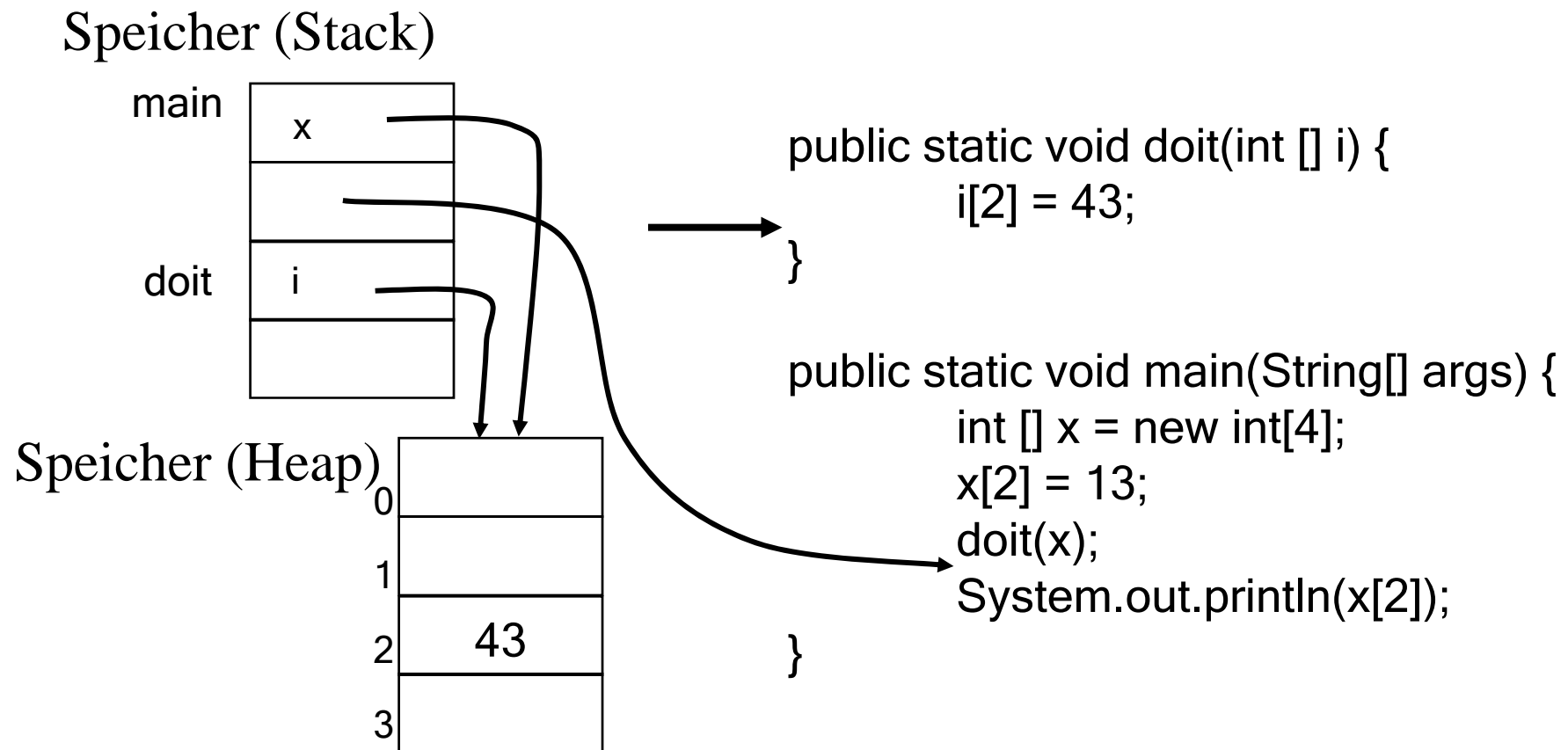
- bei einem Methodenaufruf werden die Werte der Ausdrücke an die Parameter übergeben
- der Wert eines Arrays ist der Verweis, wo das Array im Speicher liegt

```
public static void doit(int [] i) {  
    i[2] = 43;  
}  
  
public static void main(String[] args) {  
    int [] x = new int[4];  
    x[2] = 13;  
    doit(x);  
    System.out.println(x[2]);  
}
```

Was gibt dieses
Programm aus?
13 oder 43?

Parameter (Fort.)

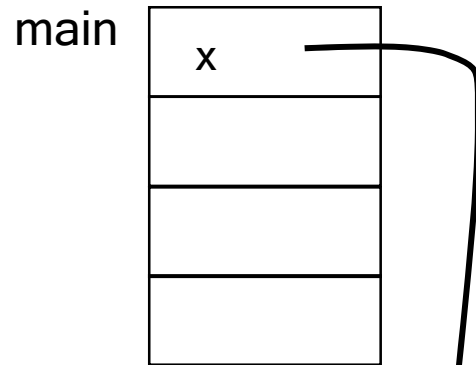
die Situation während des Methodenaufrufs: beide Variablen x und i zeigen auf den gleichen Speicherbereich



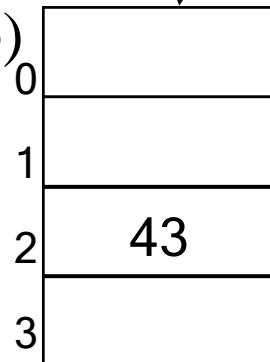
Parameter (Fort.)

die Ausgabe ist 43, weil nach dem Methodenaufruf ...

Speicher (Stack)



Speicher (Heap)



```
public static void doit(int [] i) {  
    i[2] = 43;  
}
```

```
public static void main(String[] args) {  
    int [] x = new int[4];  
    x[2] = 13;  
    doit(x);  
    System.out.println(x[2]);  
}
```

... i immer noch auf das im Methodenaufruf modifizierte Array zeigt

Parameter (Fort.)

- bei einem Methodenaufruf werden die Werte der Ausdrücke an die Parameter übergeben
- der Wert eines Arrays ist der Verweis, wo das Array im Speicher liegt

```
public static void doit(int [] j) {  
    j = new int[3];  
    j[2] = 43;  
}
```

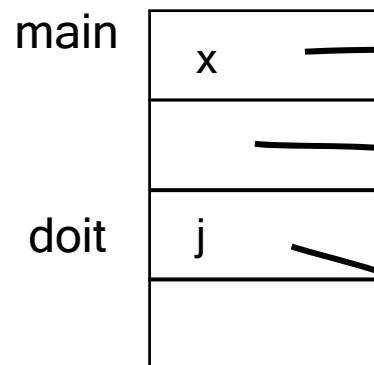
```
public static void main(String[] args) {  
    int [] x = new int[4];  
    x[2] = 13;  
    doit(x);  
    System.out.println(x[2]);  
}
```

Was gibt dieses
Programm aus?
13 oder 43?

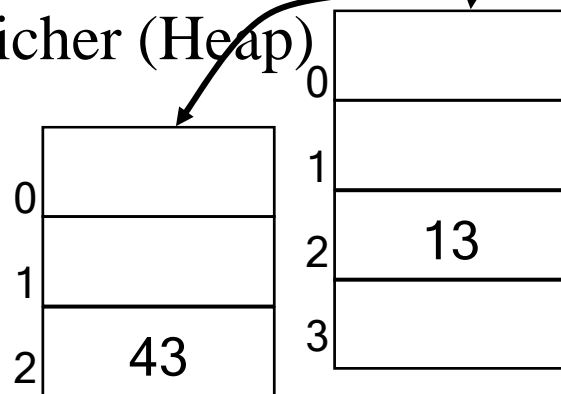
Parameter (Fort.)

die Situation während des Methodenaufrufs: j von doit zeigt auf das neuangelegte Array, daher bekommt x von main die Änderung nicht mit

Speicher (Stack)



Speicher (Heap)



```
public static void doit(int [] j) {  
    j = new int[3];  
    j[2] = 43;  
}
```

```
public static void main(String[] args) {  
    int [] x = new int[4];  
    x[2] = 13;  
    doit(x);  
    System.out.println(x[2]);  
}
```

Parameter (Fort.)

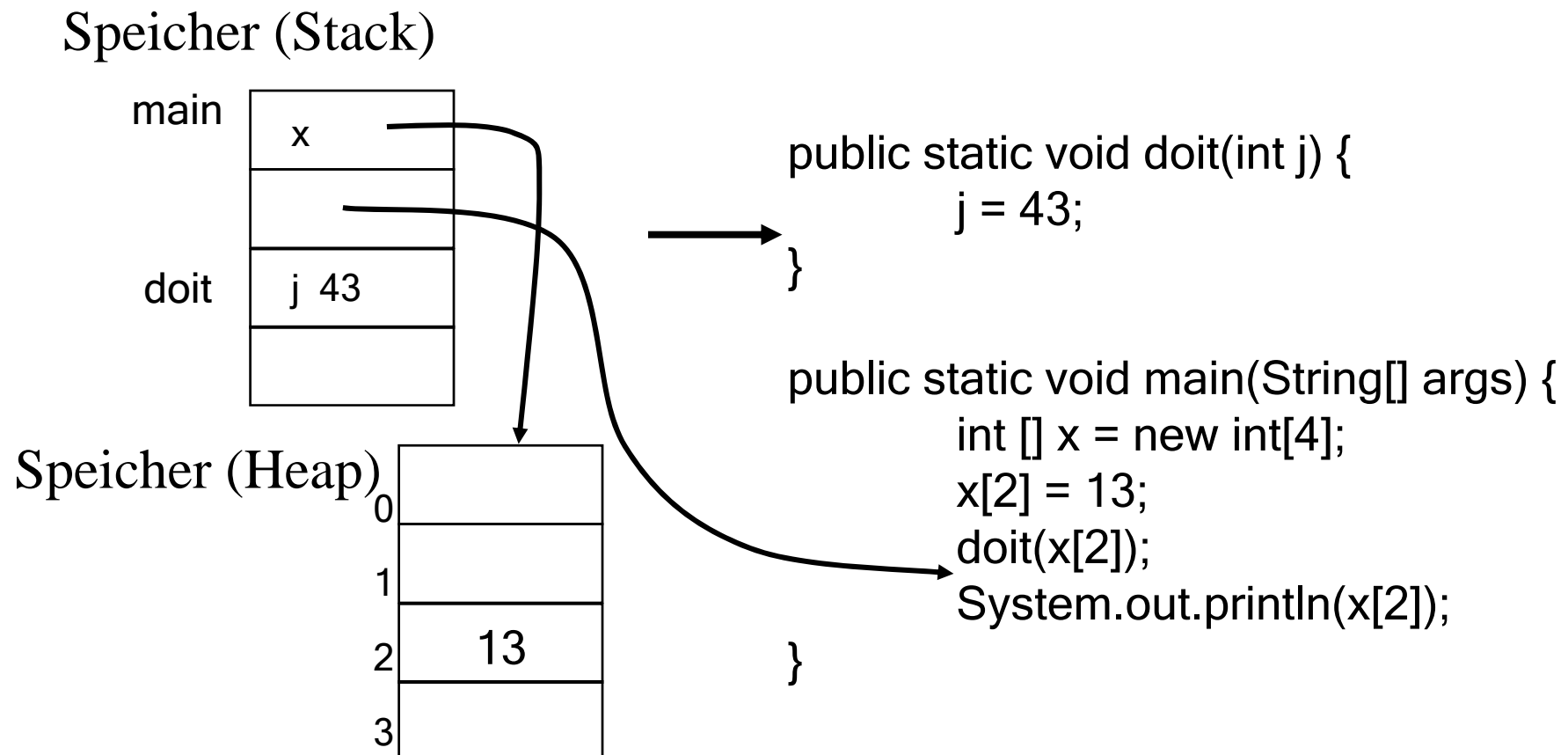
- bei einem Methodenaufruf werden die Werte der Ausdrücke an die Parameter übergeben
- der Wert einer int-Variablen ist die Zahl, die unter dieser Variablen im Speicher liegt

```
public static void doit(int j) {  
    j = 43;  
}  
  
public static void main(String[] args) {  
    int [] x = new int[4];  
    x[2] = 13;  
    doit(x[2]);  
    System.out.println(x[2]);  
}
```

Was gibt dieses
Programm aus?
13 oder 43?

Parameter (Fort.)

die Situation während des Methodenaufrufs: x von main zeigt auf das Array, aber j von doit liegt auf dem Stack und hat seinen eigenen Wert



Go!

Variable Parameterliste

- betrachtet man sich das folgende Beispiel, so wirkt die Parameterübergabe sehr gekünstelt
- da der letzte Parameter vom Typ Array ist, kann die sogenannte *variable Parameterliste* verwendet werden

```
public class MethodCall4 {
```

```
    public static void doit(int i, int j, char[] k) {  
        System.out.println(i + " " + j);  
        for(char c : k)  
            System.out.print(c);  
        System.out.println();  
    }
```

```
    public static void main(String[] args) {  
        doit(13,14,new char[]{'a','b','c','?'});  
    }
```

```
}
```


Go!

Variable Parameterliste (Fort.)

- statt `char[]` kann in der Parameterliste `char ...` stehen
- dies ist nur für den letzten Parameter möglich (warum?)
- macht nur für lesenden Zugriff Sinn (warum?)

```
public class MethodCall5 {
```

```
    public static void doit(int i, int j, char... k) {  
        System.out.println(i + " " + j);  
        for(char c : k)  
            System.out.print(c);  
        System.out.println();  
    }
```

nach 2 int können
beliebig viele char's
übergeben werden

```
    public static void main(String[] args) {  
        doit(13,14,'a','b','c','?');  
        doit(-10,-20);  
    }
```

Anwendung sieht
eleganter aus

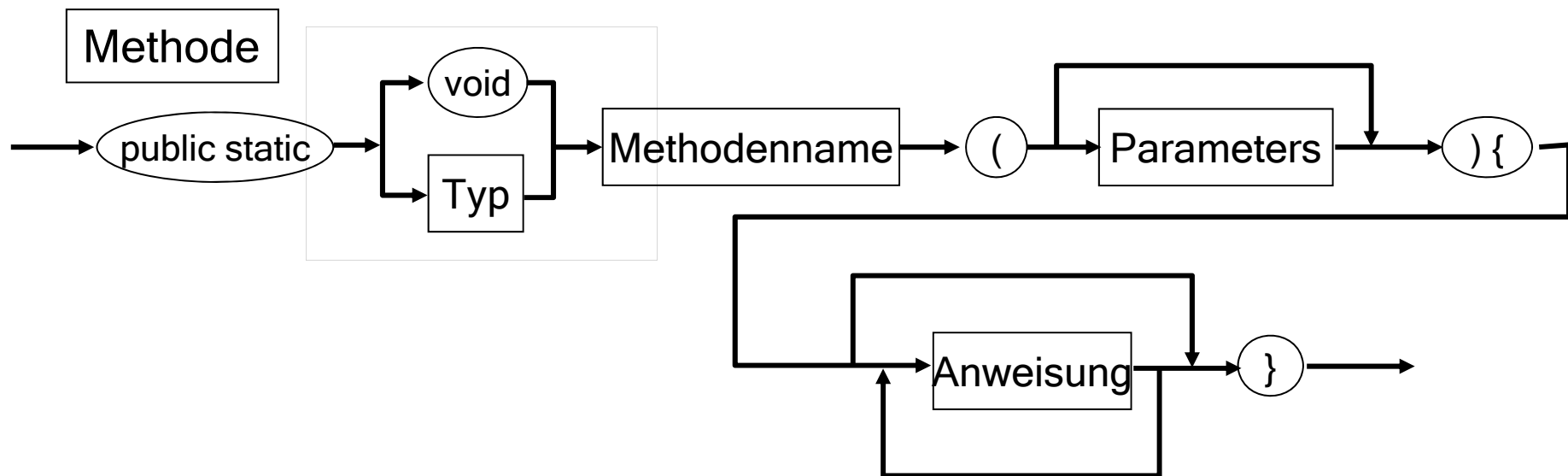
Informationsfluss



- die bisherigen Methoden lassen einen Informationsfluss vom Aufrufer zur aufgerufenen Methode zu (via Parameter)
- werden Arrays als aktuelle Parameter übergeben, so können diese Arrays in der Methode modifiziert werden
- diese Modifikation ist danach beim Aufrufer sichtbar
- somit ist eine eingeschränkte Informationsrückgabe möglich
- diese Art der Informationsrückgabe nennt man aber *Seiteneffekt* der Methode

Informationsrückgabe

- soll eine Methode einen Wert an ihren Aufrufer zurückgeben, so muss der Typ dieser möglichen Werte deklariert werden
- der Typ wird vor dem Methodennamen geschrieben
- der spezielle Typ void gibt an, dass kein Wert zurückgegeben wird



Informationsrückgabe (Fort.)

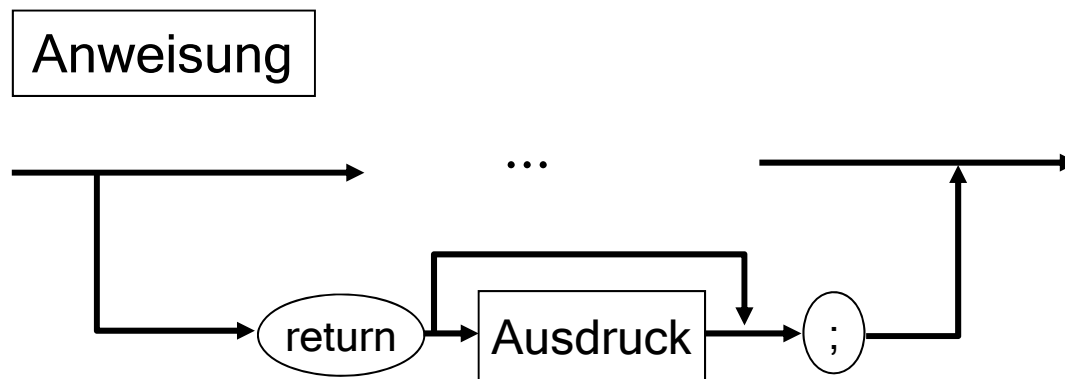
Bsp.: die folgende Methode

```
public static int doit(float f) { ... }
```

- erhält als Argument einen float-Wert
- liefert als Ergebnis einen int-Wert zurück
- man sagt, dass die Methode doit einen Rückgabewert vom Typ int hat

Return-Anweisung

- Um in der Methode zu sagen, wann welcher Wert zurückgeliefert werden soll, gibt es die return Anweisung.



- Einer return Anweisung kann ein Ausdruck folgen.
- Der Ausdruck muss von gleichem Typ sein wie die Methode, in der die return Anweisung vorkommt.
- Ist die Methode vom Typ void, so darf **kein** Ausdruck angegeben werden.

Return-Anweisung (Fort.)

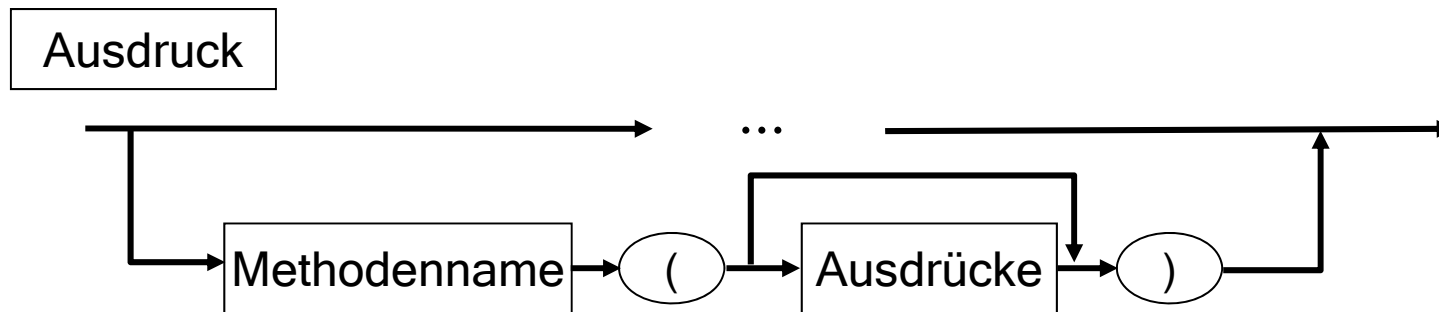
- eine return-Anweisung beendet die aktuelle Methode sofort
- der Programmablauf kehrt zu der Aufrufer zurück
- das Programm wird nach dem Methodenaufruf fortgesetzt

```
public static void doit(boolean b) {  
    if (b) {  
        System.out.println("toll");  
        return;  
    } else {  
        System.out.println("super");  
    }  
    System.out.println("juhu");  
}  
  
public static void main(String[] args) {  
    doit(3 > 4);  
    doit(true);  
}
```

Frage: Was ist
die Ausgabe?

Return-Anweisung (Fort.)

- wenn eine Methode einen Wert zuliefert, so kann dieser Wert in einem Ausdruck verwendet werden
- somit kann ein Methodenaufruf in einem Ausdruck verwendet werden, wenn die Methode *nicht* vom Typ void ist



- der Wert eines solchen Methodenaufrufs ist der, der nach der ausgeführten return-Anweisung steht

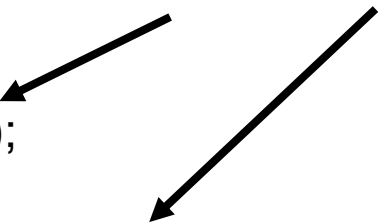
Go!

Beispiel

```
public class Funktion1 {  
    public static int quadrat_mit_vorzeichen(int i) {  
        if (i > 0)  
            return i*i;  
        else  
            return -i*i;  
    }  
}
```

2-mal wird die Methode
quadrat_mit_vorzeichen
aufgerufen

```
public static void main(String[] args) {  
    int j = quadrat_mit_vorzeichen(-4);  
    System.out.println(j);  
    System.out.println(quadrat_mit_vorzeichen(-3));  
}  
}
```



Was gibt dieses Programm aus?

Return-Anweisung (Fort.)

- Vorsicht mit der Rückgabe von Arrays
- es wird nicht das Array, sondern der Verweis auf ein Array zurückgegeben
- dadurch kann es zum gemeinsamen Verweisen auf die gleichen Arrays kommen

Go!

Beispiel

```
public class Funktion2 {  
    public static int[] strange(int[] i) {  
        if (i.length > 3)  
            return i;  
        else  
            return new int[4];  
    }
```

liefert das übergebene Array
oder ein Neues zurück

```
    public static void store(int[] j) {  
        j[2] = 13;  
    }
```

speichert in dem über-
gebenen Array einen Wert

```
    public static void main(String[] args) {  
        int[] i1 = new int[4];  
        int[] i2 = new int[3];  
        i1[2] = 42;  
        i2[2] = 42;  
        store(strange(i1));  
        store(strange(i2));  
        System.out.println(i1[2]);  
        System.out.println(i2[2]);  
    }
```

legt 2 Arrays an und beschreibt
sie an der gleichen Stelle mit
dem gleichen Wert ...

Was gibt dieses Programm aus?

Vorlesung 8/2

Sortieren (Der Anfang)

Beim Sortieren geht es darum, mehrere Elemente, die untereinander vergleichbar sind, derart anzuordnen, dass jeder Nachfolger eines Elementes größer oder gleich seinem Vorgänger ist.

Voraussetzung:

- auf den Elementen muss eine totale Ordnung R existieren
- zu zwei gegebenen Elementen x und y muss einfach berechenbar sein, ob $(x,y) \in R$?

Sortieren: Grundlagen

Es werden 2 Situationen unterschieden:

1. die zu sortierenden Daten passen in den Hauptspeicher
(*interne Sortierung*)
2. die zu sortieren Daten sind derart umfangreich, dass sie nur auf der Platte oder Magnetbändern Platz finden
(*externe Sortierung*)

Unterschied:

interne Sortierung \Rightarrow *direkter Zugriff* auf jedes Element

externe Sortierung \Rightarrow *sequentieller Zugriff* auf ein Element über seinen Vorgänger

Sortieren: Grundlagen (Fort.)

Geeignete Datenstrukturen für:

internes Sortieren: Arrays, Vektoren

externes Sortieren: Listen, Files

Sortieralgorithmen werden unterschieden:

- wie viel Speicherplatz sie *zusätzlich* benötigen
- *wie lange* sie benötigen, um n Elemente zu sortieren
- ob sie *stabil* sind, d.h. ob zwei gleiche Elemente nach dem Sortieren immer noch in der gleichen Reihenfolge stehen wie vor dem Sortieren

1. Algorithmus: Selection Sort

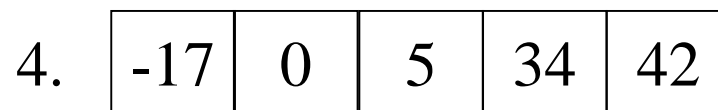
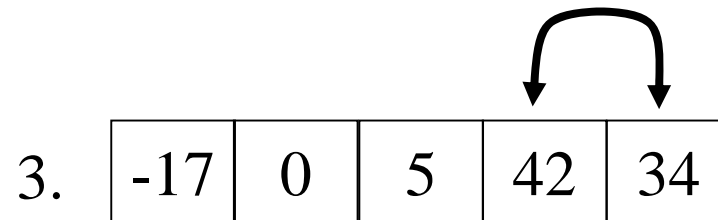
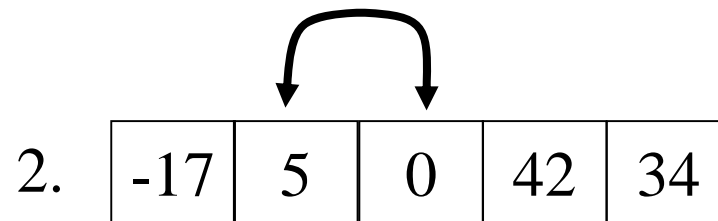
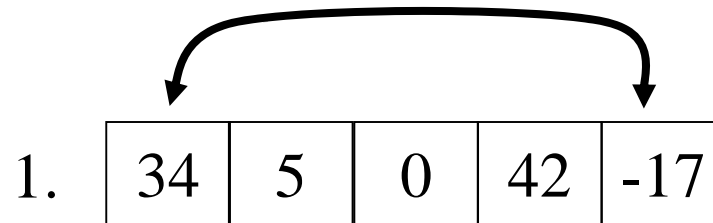
Einfachstes Sortiervverfahren, dass jeder intuitiv anwendet:

1. suche das kleinste Element
2. lege es am Anfang ab
3. suche das zweitkleinste Element
4. lege es ein Element nach dem Anfang ab
5. usw.

1. Algorithmus: Selection Sort (Fort.)

Ausgangssituation:

34	5	0	42	-17
----	---	---	----	-----



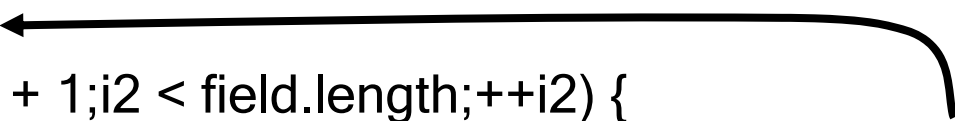
Selection Sort: Implementierung

```
static void sort(int[] field) {  
    for(int i1 = 0; i1 < field.length - 1; ++i1) {  
        int min = i1;  
        for(int i2 = i1 + 1; i2 < field.length; ++i2) {  
            if (field[i2] < field[min])  
                min = i2;  
        }  
        swap(field, min, i1);  
    }  
}
```

tausche die Elemente aus

```
static void swap(int[] field, int iPos1, int iPos2) {  
    int tmp = field[iPos1];  
    field[iPos1] = field[iPos2];  
    field[iPos2] = tmp;  
}
```

vertauscht die beiden
Elemente an den Positionen
iPos1 und iPos2



min merkt sich immer
die Position des
kleinsten Elements

Selection Sort: Analyse

Laufzeit:

- 1. Durchlauf: $n-1$ Schritte
- 2. Durchlauf: $n-2$ Schritte
- 3. Durchlauf: $n-3$ Schritte
- ...

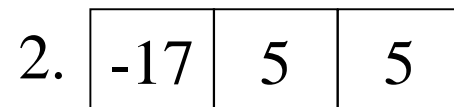
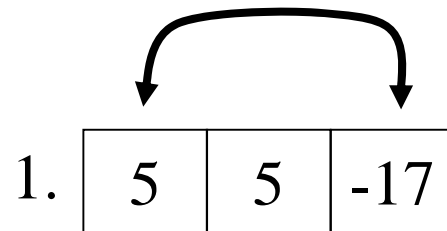
insgesamt: $\frac{n^2-n}{2}$

$O(n^2)$ (zwei ineinander geschachtelte for-Schleifen)

zusätzlich benötigter Speicherplatz: keiner

Stabilität des Selection Sorts

In der einfachen Implementierung ist der Selection Sort *nicht* stabil:

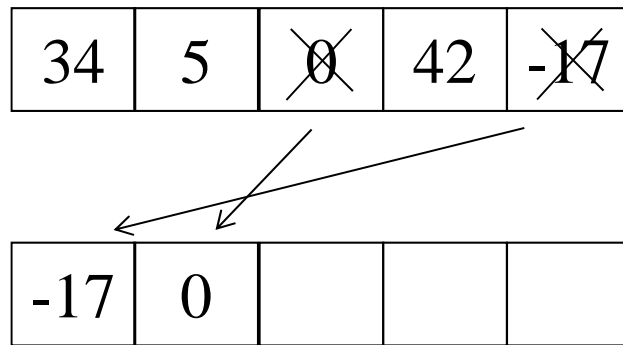


Die beiden Fünfen haben ihre Reihenfolge getauscht, obwohl sie gemäß der Ordnung identisch sind

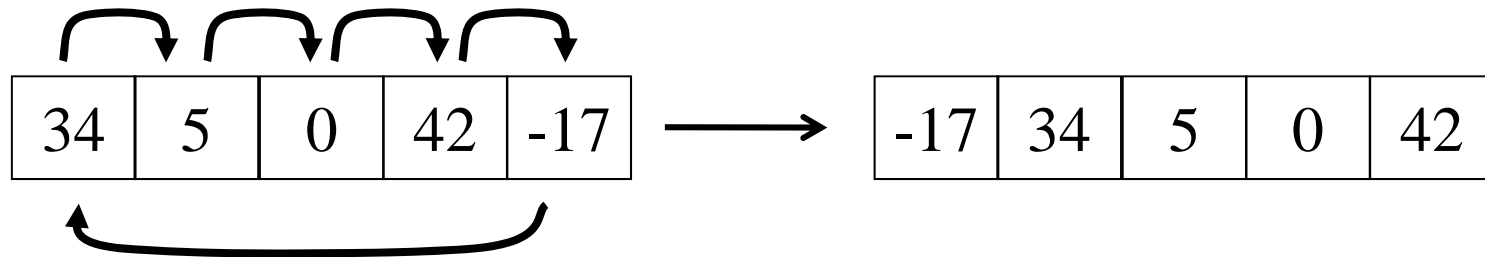
Stabiles Selection Sort

es gibt zwei Möglichkeiten, dass der Selection Sort stabil sortiert

1. das zu sortierende Array kopieren, oder



2. das gefundene kleinste Element nicht mit dem aktuellen vertauschen, sondern alle Elemente aufschieben



Insertion Sort: Motivation

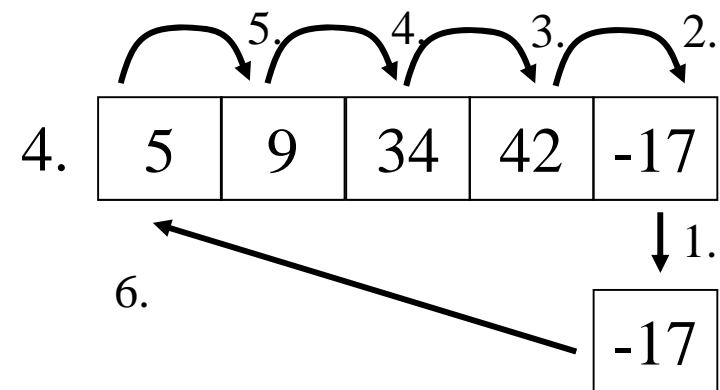
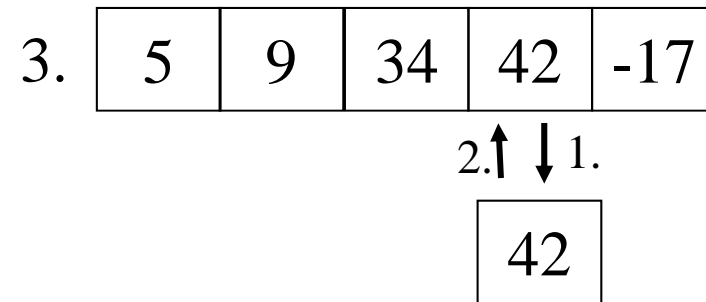
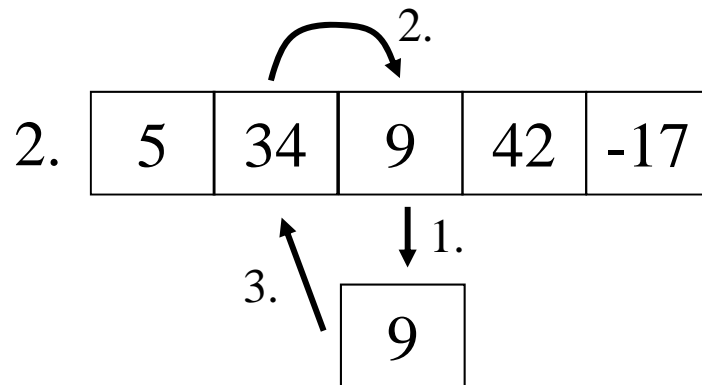
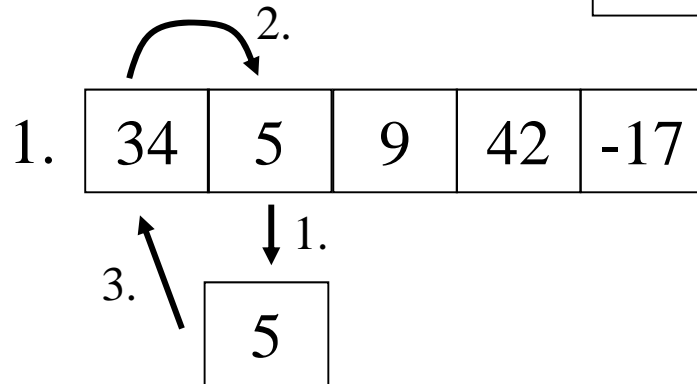
wie beim Sortieren von Spielkarten:

- nehme eine neue Karte und füge sie an der richtigen Stelle in den bereits sortierten Teil ein
- dazu muss eventuell Platz geschaffen werden und die anderen Karten nach hinten verschoben werden

Insertion Sort: Beispiel

Ausgangssituation:

34	5	9	42	-17
----	---	---	----	-----



Insertion Sort: Implementierung

```
static void insertion_sort(int[] field) {  
    for(int i1 = 1; i1 < field.length; ++i1) {  
        final int IVAL = field[i1];  
        int i2 = i1;  
        while (i2 > 0 && field[i2 - 1] > IVAL) {  
            field[i2] = field[i2 - 1];  
            --i2;  
        }  
        field[i2] = IVAL;  
    }  
}
```

IVAL ist das
Element, das
eingefügt
werden soll

verschiebe die bereits
sortierten Elemente, bis
IVAL richtig platziert ist

speichere IVAL an dem
neu geschafften Platz ab

Insertion Sort: Analyse

Laufzeit: 1. Durchlauf: maximal 1 Schritt
 2. Durchlauf: maximal 2 Schritte
 3. Durchlauf: maximal 3 Schritte
 ...
 insgesamt: $\frac{n^2-n}{2}$

$O(n^2)$ (zwei ineinander geschachtelte Schleifen (for und while))

zusätzlich benötigter Speicherplatz: keiner

Stabilität: ja

Insertion Sort: Analyse (Fort.)

Was passiert bei Insertion Sort für dieses Array?

5	9	34	42	102
---	---	----	----	-----

Laufzeit: 1. Durchlauf: 1 Schritt
 2. Durchlauf: 1 Schritt
 3. Durchlauf: 1 Schritt
 ...
 insgesamt:

$O(n)$ (Abarbeitung der äußeren for-Schleife)

d.h.: Laufzeit hängt stark von der Vorsortierung ab!

Vergleich: Insertion Sort – Selection Sort

- Insertion Sort und Selection Sort sind interne Sortiervverfahren; Insertion Sort kann auch für externe Sortierung verwendet werden
- Selection Sort benötigen den direkten Zugriff auf den Speicher
 - `swap(field, min, i1);`
- Insertion Sort benötigen den sequentiellen Zugriff auf den Speicher
 - `field[i2] = IVAL;`
- Selection Sort ist somit *nicht geeignet*, eine *Datei zu sortieren*

Bubble Sort: Idee

- vergleiche 2 benachbarte Elemente
 - sind sie in der richtigen Reihenfolge, mache mit dem nächsten Element weiter
 - sind sie nicht in der richtigen Reihenfolge, vertausche sie und mache mit dem nächsten Element weiter
- man lässt die maximalen Elemente wie Luftblasen in dem Array aufsteigen
- oben sammelt sich erst das maximale Element, dann das zweitgrößte, usw.

Bubble Sort: Beispiel

Ausgangssituation:

34	5	9	42	-17
----	---	---	----	-----

1.

34	5	9	42	-17
----	---	---	----	-----

2.

5	34	9	42	-17
---	----	---	----	-----

3.

5	9	34	42	-17
---	---	----	----	-----

4.

5	9	34	-17	42
---	---	----	-----	----

5.

5	9	-17	34	42
---	---	-----	----	----

6.

5	-17	9	34	42
---	-----	---	----	----

7.

-17	5	9	34	42
-----	---	---	----	----

Bubble Sort: Implementierung

```
static void bubble_sort(int[] field) {  
    for(int i1 = 1; i1 < field.length; ++i1) {  
        for(int i2 = 0; i2 < field.length-i1; ++i2) {  
            if (field[i2] > field[i2 + 1])  
                swap(field, i2, i2+1);  
        }  
    }  
}
```

sind 2 aufeinanderfolgende
Elemente nicht sortiert,
werden sie vertauscht

geeignet für externes
Sortieren, da nur sequentiell
auf die Elemente zugegriffen
wird (nach i2 kommt i2+1)

i2 läuft immer
über das Array,
lässt dabei
immer ein
Element mehr
aus

Bubble Sort: Analyse

Laufzeit:

1. Durchlauf: $n-1$ Schritte
2. Durchlauf: $n-2$ Schritte
3. Durchlauf: $n-3$ Schritte
- ...

insgesamt: $\frac{n^2-n}{2}$

$O(n^2)$ (zwei ineinander geschachtelte for-Schleifen)

zusätzlich benötigter Speicherplatz: keiner

Stabilität: ja

Bubble Sort: Analyse (Fort.)

Was passiert bei Bubble Sort für dieses Array?

5	9	34	42	102
---	---	----	----	-----

Laufzeit:

1. Durchlauf: $n-1$ Schritte, aber kein **swap**
2. Durchlauf: $n-2$ Schritte, aber kein **swap**
3. Durchlauf: $n-3$ Schritte, aber kein **swap**

...

insgesamt: $\frac{n^2-n}{2}$

also: $O(n^2)$

Frage: wenn in einem Durchlauf kein **swap** ausgeführt wurde, ist dann alles sortiert?

Bubble Sort: Optimierung

```
static void bubble_sort_opt(int[] field) {  
    for(int i1 = 1; i1 < field.length; ++i1) {  
        boolean bAtLeastOneSwap = false;  
        for(int i2 = 0; i2 < field.length-i1; ++i2) {  
            if (field[i2] > field[i2 + 1]) {  
                swap(field, i2, i2+1);  
                bAtLeastOneSwap = true;  
            }  
        }  
        if (!bAtLeastOneSwap)  
            return;  
    }  
}
```

merkt sich, ob
wenigstens ein **swap**
ausgeführt wurde

ja, es ist ein **swap** ausgeführt
worden, das Array ist noch
nicht sortiert

wenn im letzten Durchlauf
kein **swap** ausgeführt
wurde, sind wir fertig

Bubble Sort: Analyse (Fort.)

Was passiert beim *optimierten* Bubble Sort für dieses Array?

5	9	34	42	102
---	---	----	----	-----

Laufzeit: 1. Durchlauf: n Schritte, kein **swap**, Abbruch

also: $O(n)$

Distribution Counting

besondere Situation:

- es sollen n natürliche oder ganze Zahlen sortiert werden
- die Zahlen liegen in einem Bereich zwischen 0 und m
- n kann eine sehr große Zahl sein
- m ist eine relativ kleine Zahl

Idee:

- lege ein Feld mit m Einträgen an
- merke in der Stelle i , wie viele Zahlen i in den n Zahlen vorkommen

Distribution Counting: Beispiel

2	3	1	7	5	6	2	7	3	1
---	---	---	---	---	---	---	---	---	---

besondere Situation:

- es sollen 10 Zahlen sortiert werden
- die Zahlen liegen in einem Bereich zwischen 0 und 8, also $[0,8)$

0	2	2	2	0	1	1	2
0	1	2	3	4	5	6	7

Ergebnis:

- ein Feld, das sich merkt, wie oft Index als Zahl in der zu sortierenden Folge vorkommt

Distribution Counting: Beispiel (Fort.)

0	2	2	2	0	1	1	2
0	1	2	3	4	5	6	7

nach der "Sortierung": Ausgabe

- 2-mal die 1
- 2-mal die 2
- 2-mal die 3
- 1-mal die 5
- 1-mal die 6
- 2-mal die 7

1	1	2	2	3	3	5	6	7	7
---	---	---	---	---	---	---	---	---	---

Distribution Counting: Implementierung

```
static void distribution_counting(int[] field, int m) {  
    int[] count = new int[m];  
    for(int i = 0; i < field.length; ++i) {  
        ++count[field[i]];  
    }  
    for(int i1 = 0, i2 = 0; i1 < count.length; ++i1) {  
        for(int i3 = 0; i3 < count[i1]; ++i3) {  
            field[i2++] = i1;  
        }  
    }  
}
```

lege zusätzliches Feld an

zähle die Einträge

speichere die Einträge
aus count gezielt
wieder in field ab

Distribution Counting: Analyse

```
static void distribution_counting(int[] field, int m) {  
    int[] count = new int[m];  
    for(int i = 0; i < field.length; ++i) {  
        ++count[field[i]];           O(n)  
    }  
    for(int i1 = 0, i2 = 0; i1 < count.length; ++i1) {  
        for(int i3 = 0; i3 < count[i1]; ++i3) {  
            field[i2++] = i1;        ???  
        }  
    }  
}
```

Distribution Counting: Analyse (Fort.)

```
...  
    for(int i1 = 0, i2 = 0; i1 < count.length; ++i1) {  
        for(int i3 = 0; i3 < count[i1]; ++i3) {  
            field[i2++] = i1;  
        }  
    }
```

Überlegung:

- die äußere Schleife wird `count.length`-mal durchlaufen (also m)
- die innere Schleife wird sooft durchlaufen, so viele `count[i1]`-Zahlen es in der zu sortierenden Folge gibt
- alle `count[i1]`-Zahlen für alle Einträge können aber nicht mehr als die zu sortierenden Zahlen sein, d.h. $\sum_{i1=0}^m \text{count}[i1] = n$
- d.h., die Komplexität und damit Gesamtkomplexität ist $O(n)$
- Frage: ist das Verfahren stabil?

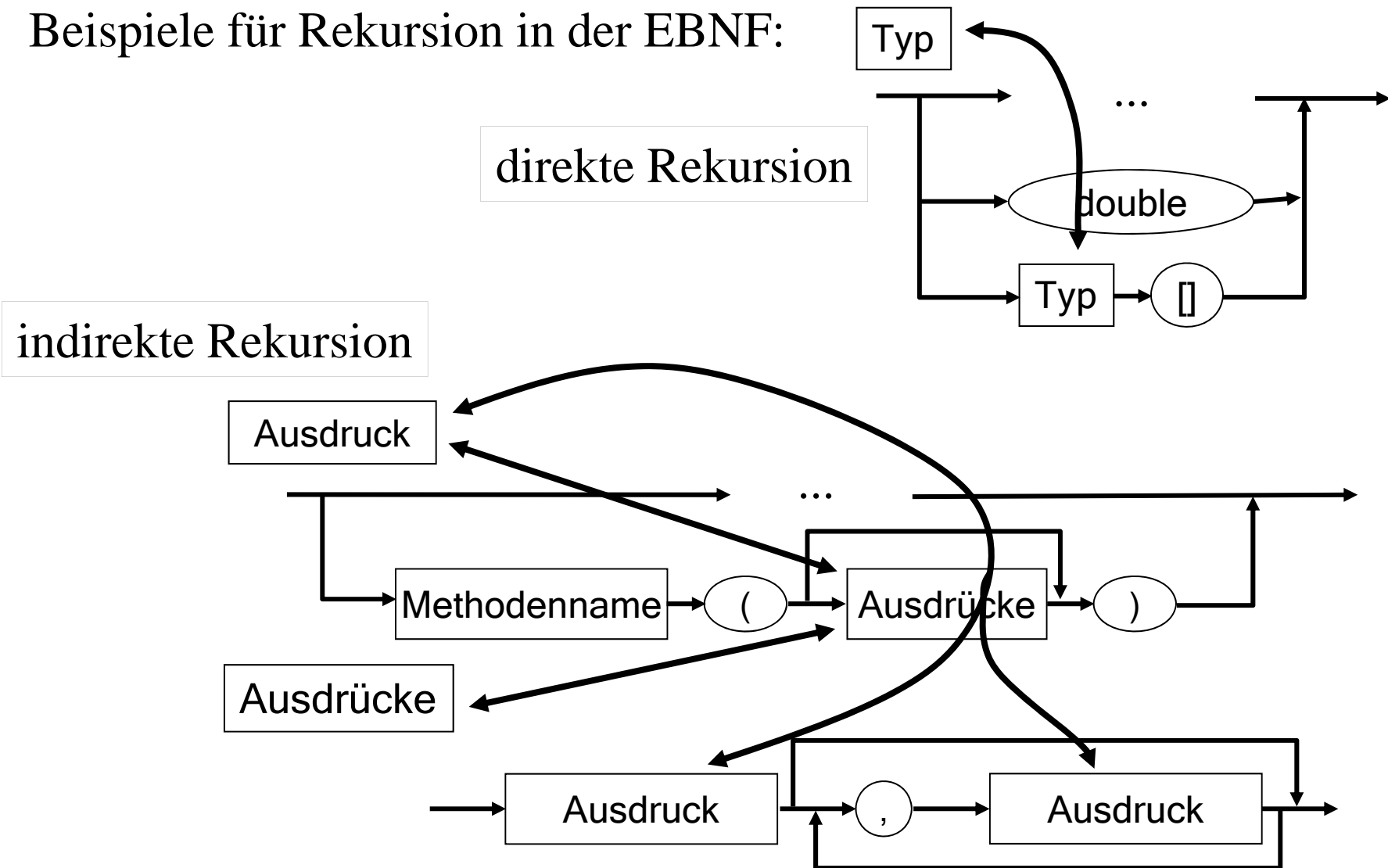
Vorlesung 9/1

Rekursion

- Wenn sich Strukturen für ihre Definition auf sich selbst berufen/abstützen, so nennt man dies Rekursion.
- Rekursion kann direkt oder indirekt sein, d.h.
 - direkt: die Definition stützt sich sofort auf sich selber ab
 - indirekt: die Definition stützt sich auf eine oder mehrere andere Strukturen ab, die sich dann wieder auf die ursprüngliche abstützen.

Rekursion (Fort.)

Beispiele für Rekursion in der EBNF:

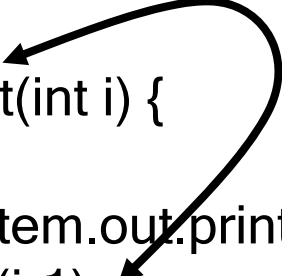


Go!

Rekursion in der Programmierung

- Rekursion kann auch in der Programmierung vorkommen
- eine Methode kann andere Methoden aufrufen
- eine Methode kann aber auch sich selber aufrufen

```
public class Rekursion1 {  
    public static void doit(int i) {  
        if (i > 0) {  
            System.out.println(i);  
            doit(i-1);  
        }  
    }  
    public static void main(String[] args) {  
        doit(10);  
    }  
}
```



Rekursiver Aufruf

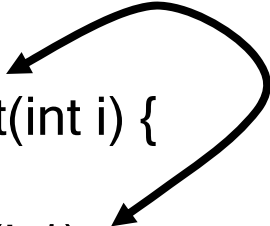
Was gibt dieses
Programm aus?

Go!

Rekursion in der Programmierung (Fort.)

- es ist nicht egal, an welcher Stelle die Methode rekursiv aufgerufen wird
- durch Verschieben der Aufrufstelle wird sich das Programm i.d.R. deutlich ändern

```
public class Rekursion2 {  
    public static void doit(int i) {  
        if (i > 0) {  
            doit(i-1);  
            System.out.println(i);  
        }  
    }  
    public static void main(String[] args) {  
        doit(10);  
    }  
}
```



Rekursiver Aufruf,
diesmal vor
System.out.println

Was gibt dieses
Programm aus?


Go!

Rekursion in der Programmierung (Fort.)

- Rekursion kann manchmal durch einfache Schleifen ersetzt werden

```
public class Rekursion3 {  
    public static int sum_rec(int i) {  
        return i<=0 ? 0 : sum_rec(i-1) + i;  
    }  
}
```

rekursive
Definition



```
public static int sum_iter(int i) {  
    int res = 0;  
    for(int j = 0; j <= i; ++j)  
        res += j;  
    return res;  
}
```

iterative
Definition

Was gibt dieses
Programm aus?

```
public static void main(String[] args) {  
    System.out.println(sum_rec(10));  
    System.out.println(sum_iter(10));  
}
```

Go!

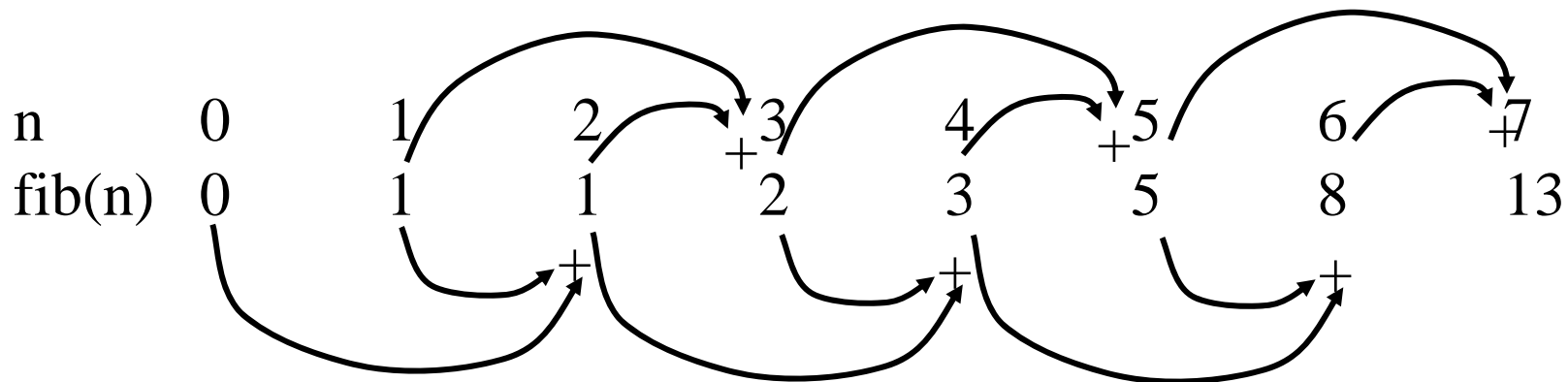
Rekursion in der Programmierung (Fort.)

- Rekursion kann auch mehrfach in einem Ausdruck vorkommen

Wie sieht die
iterative
Lösung aus?

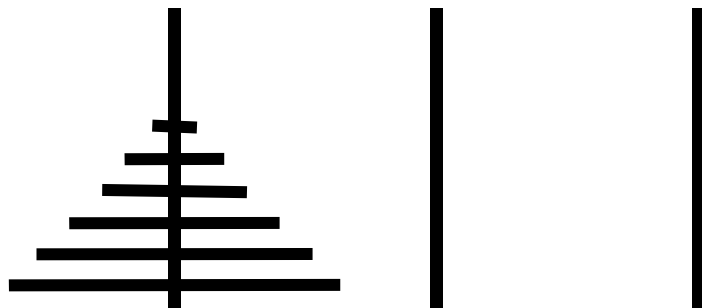
```
public class Rekursion4 {  
    public static int fib(int n) {  
        return n > 1 ? fib(n-2)+fib(n-1) : n;  
    }  
    public static void main(String[] args) {  
        System.out.println(fib(10));  
    }  
}
```

rekursive
Definition



Rekursion in der Programmierung (Fort.)

- Rekursion ist ein sehr mächtiges Instrument zur Lösung von Problemen
- Manche Probleme lassen sich lösen, indem man sie auf sich selbst reduziert, jedoch von kleinerer Größe
- Diese kleine Lösung lässt sich dann zur Gesamtlösung zusammenfassen
- Beispiel: Die Türme von Hanoi



Aufgabe:

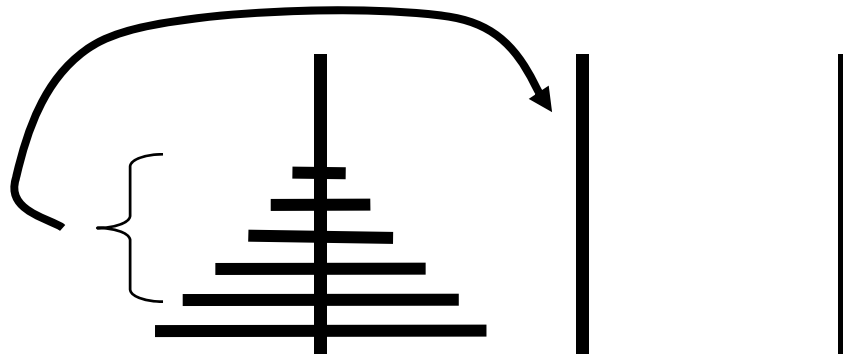
- bewege den Turm von links nach rechts
- es darf immer nur eine Scheibe bewegt werden
- es darf niemals eine größere Scheibe auf einer kleineren liegen

Die Türme von Hanoi

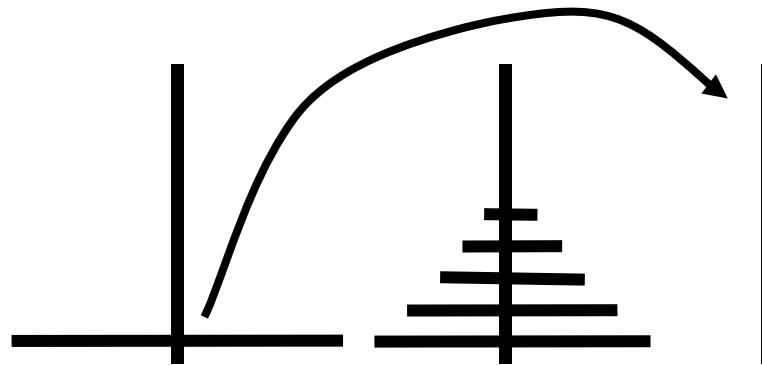
- Iterative Lösung? Ist direkt nicht einsichtig!
- Rekursive Lösung?
 - einfach, wenn es nur eine Scheibe wäre: schiebe einfach die Scheibe rüber
 - wenn es N viele Scheiben sind, dann
 1. verschiebe $N-1$ viele Scheiben von links in die Mitte
 2. verschiebe die größte Scheibe von links nach rechts
 3. verschiebe $N-1$ viele Scheiben von der Mitte nach rechts
- rekursive Lösung reduziert das Problem auf sich selber, aber mit weniger Scheiben

Die Türme von Hanoi (Fort.)

Schritt 1: rekursiver Aufruf, wobei das Problem um 1 verringert ist

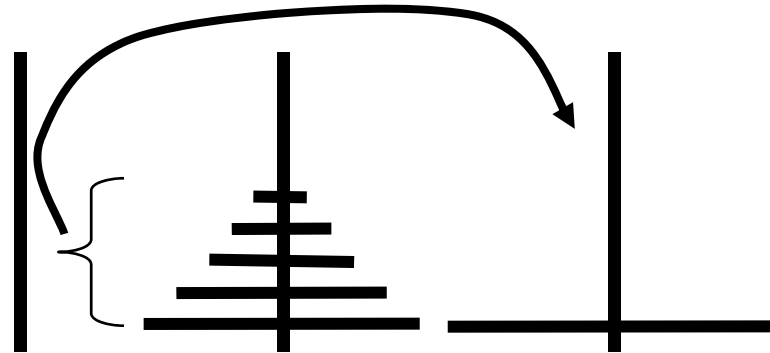


Schritt 2: Lösung kann direkt angegeben werden

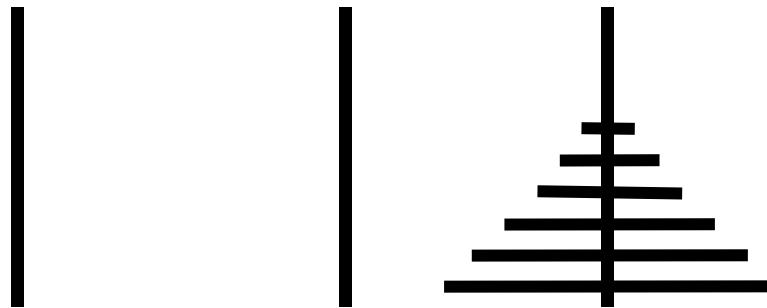


Die Türme von Hanoi (Fort.)

Schritt 3: rekursiver Aufruf, wobei das Problem um 1 verringert ist



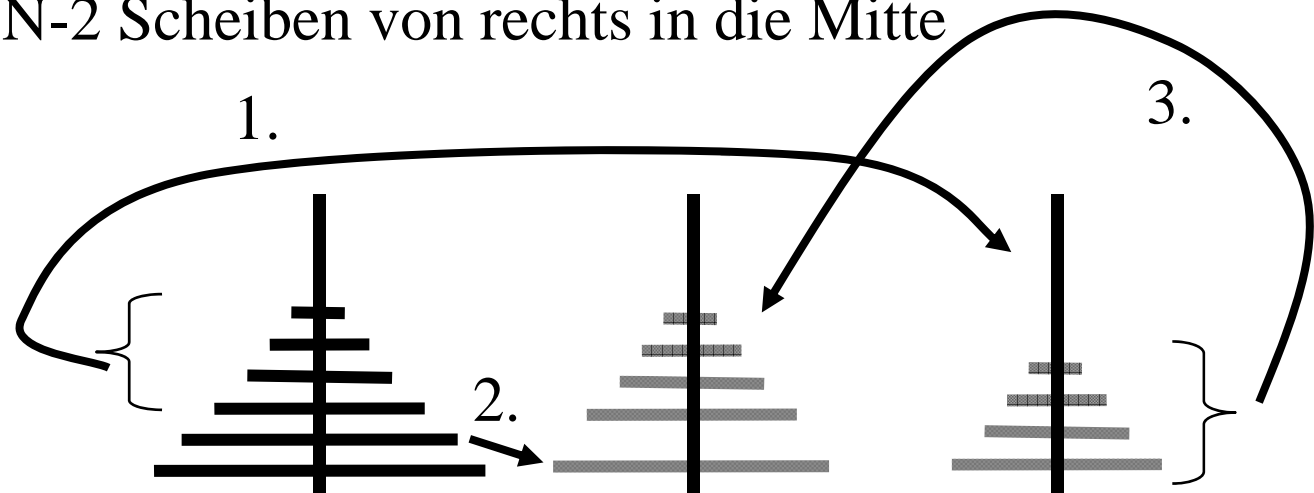
Ergebnis: Die Türme wurden gemäß der Regel verschoben



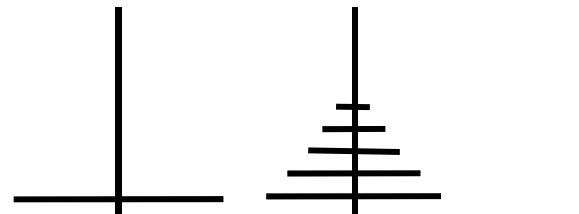
Die Türme von Hanoi (Fort.)

Wie werden aber nun $N-1$ Scheiben von links in die Mitte geschoben?

1. verschiebe $N-2$ Scheiben von links nach rechts
2. verschiebe die zweitgrößte Scheibe von links in die Mitte
3. verschiebe $N-2$ Scheiben von rechts in die Mitte

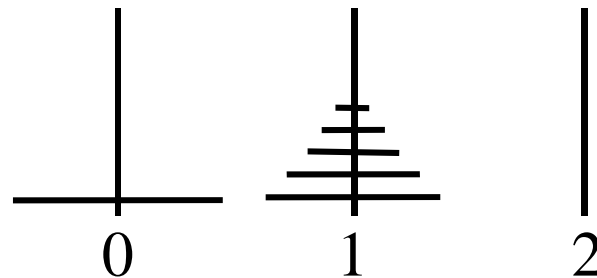


Ergebnis:



Die Türme von Hanoi: Implementierung

- benötigt wird eine Methode, die sich dann 2-mal rekursiv aufruft
- diese Methode braucht:
 - die Anzahl der zu bewegendenden Steine
 - die Position von welcher Stange zu welcher Stange verschoben werden soll
 - die Position, welche Stange als Zwischenablage dienen kann
- die Stangen werden hier von 0 bis 2 durchnummeriert



Die Türme von Hanoi: Implementierung (Fort.)

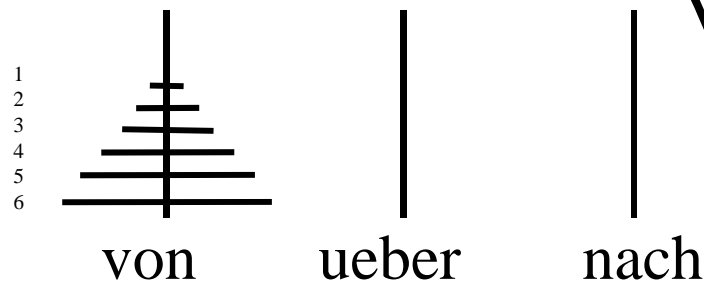
wie viele Steine

woher

wohin

welches ist die
Hilfsstange

```
public static void move(int N, int von, int nach, int ueber) {  
    if (N > 0) {  
        move(N-1, von, ueber, nach);  
        System.out.println("Bewege Stein von " + von + " nach " + nach);  
        move(N-1, ueber, nach, von);  
    }  
}
```



Beispiel:

- 6 Steine von "von" nach "nach" über "ueber" verschieben.
- Dazu 5 Steine von "von" nach "ueber" über "nach" verschieben
- Großen Stein von "von" nach "nach" verschieben
- 5 Steine von "ueber" nach "nach" über "von" verschieben

Go!

Die Türme von Hanoi: Implementierung (Fort.)

```
public class Hanoi1 {  
  
    public static void move(int N,int von,int nach,int ueber) {  
        if (N > 0) {  
            move(N-1,von,ueber,nach);  
            System.out.println("Bewege Stein von " + von + " nach " + nach);  
            move(N-1,ueber,nach,von);  
        }  
    }  
}
```

```
    public static void main(String[] args) {  
        move(3, 0, 2, 1);  
    }  
}
```

Bewege 3
Steine ...

... von
Turm 0 ...

... zu
Turm 2 ...

... unter Zuhilfe-
nahme von Turm 1

Frage: Wie oft werden
Steine verschoben, bis
ein Turm mit 10 Steinen
verlegt worden ist?

Die Türme von Hanoi: Animation

- gewünscht ist ein Programm, das
 - nicht nur die Spielzüge für die Türme von Hanoi ausgibt, sondern
 - die Türme auch auf dem Bildschirm zeichnet und so
 - die verschiedenen Spielsituationen anzeigt
- dazu muss sich das Programm die Türme merken
- für jeden einzelnen Turm muss sich das Programm merken, welche Steine auf ihm liegen
- dafür muss sich das Programm die Steine merken

Die Türme von Hanoi: Animation (Fort.)

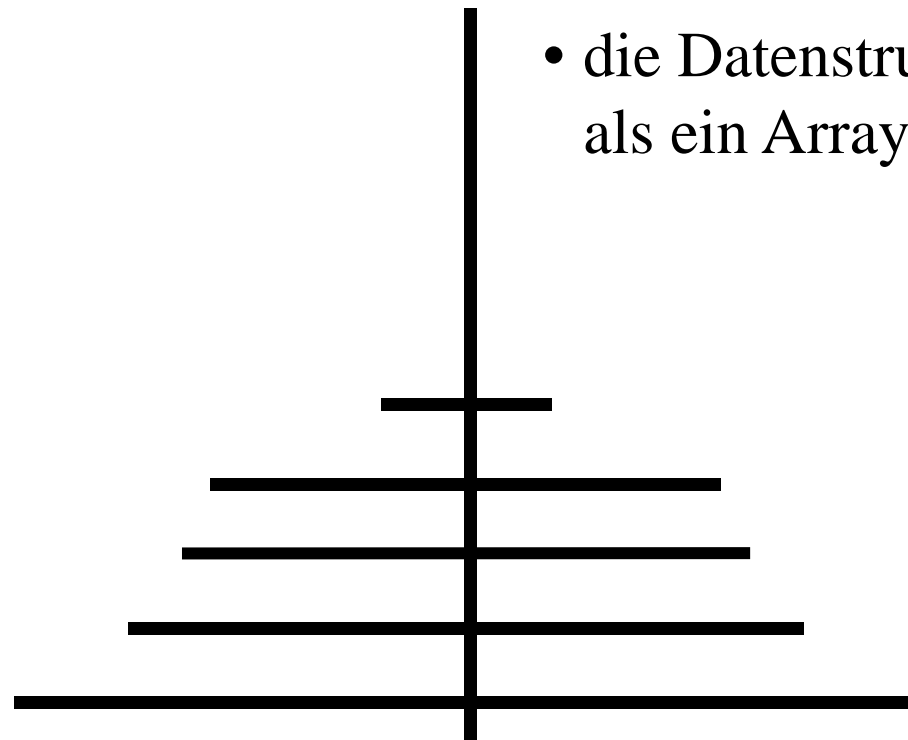
- ein Stein wird durch eine einfache Zahl kodiert
- die Zahl gibt die Breite der Steins an
- ein Turm ist dann ein int-Array
- der Turm wächst nach unten, d.h. der Index 0 ist der oberste Eintrag, der Index `length-1` ist unterste Eintrag
- ein Wert von 0 besagt, dass kein Stein an der Stelle liegt
- wenn also für einen Index `i` gilt, dass der Wert 0 ist, warum ist der Wert für alle anderen Indizes $< i$ auch 0

Wenn für einen Index `i` gilt, dass der Wert 0 ist, warum ist der Wert für alle anderen Indizes $< i$ auch 0?

Die Türme von Hanoi: Animation (Fort.)

- ein Turm mit maximal 10 Einträgen sähe z.B. so aus:

0	0
1	0
2	0
3	0
4	0
5	3
6	7
7	9
8	13
9	19



- die Datenstruktur für 3 Türme ist als ein Array mit 3 int-Arrays

Die Türme von Hanoi: Animation (Fort.)

Das Hauptprogramm ist einfach

```
public class Hanoi2 {
```

```
...
```

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

```
        int [][] towers = new int[3][5];
```

```
        init(towers);
```

```
        printTowers(towers);
```

```
        move(towers,towers[0].length,0,2,1);
```

```
    }
```

```
}
```

Die 3 Türme werden in einem
2-dim. int-Array gespeichert ...

... initialisiert, ...

... ausgedruckt ...

... und dann beginnt das eigentliche
Spiel, dass jetzt die Türme braucht.

Die Türme von Hanoi: Animation (Fort.)

Die Initialisierung

für alle 3 Türme ...

```
public static void init(int[][] towers) {  
    for(int i = 0; i < towers.length; ++i) {  
        for(int j = 0; j < towers[i].length; ++j) {  
            towers[i][j] = 0;  
        }  
    }  
    setInitTower(towers);  
}
```

... wird jeder Stein ...

... zunächst auf 0
gesetzt.

```
public static void setInitTower(int[][] towers) {  
    for(int i = 0; i < towers[0].length; ++i) {  
        towers[0][i] = 2*i+1;  
    }  
}
```

Anschließend wird der
erste Turm mit Steinen
der Breite 1, 3, 5, 7, ...
besetzt.

Die Türme von Hanoi: Animation (Fort.)

Das Drucken der 3 Türme ...

... erfolgt
zeilenweise (!!!).

```
public static void printTowers(int[][] towers) {  
    for(int j = 0; j < towers[0].length; ++j) {  
        for(int i = 0; i < towers.length; ++i) {  
            printStone(towers[i][j]);  
        }  
        System.out.println();  
    }  
  
    System.out.print("-----");  
    System.out.println("-----\n\n");  
}
```

Für jeden Turm i ...

... wird der Stein der aktuellen Zeile j gedruckt.

Nach einer Zeile wird ein Zeilenumbruch gedruckt.

Am Ende wird eine lange Zeile mit – Zeichen gedruckt

Die Türme von Hanoi: Animation (Fort.)

Das Drucken eines Steins soll immer genau 21 Zeichen breit sein.

```
public static void printStone(int iWidth) {  
    if (iWidth == 0) {  
        for(int i = 0; i < 21; ++i)  
            System.out.print(" ");  
    } else {  
        for(int i = 0; i < (21-iWidth) / 2; ++i)  
            System.out.print(" ");  
        for(int i = 0; i < iWidth; ++i)  
            System.out.print("+");  
        for(int i = 0; i < (21-iWidth) / 2; ++i)  
            System.out.print(" ");  
    }  
}
```

Wenn kein Stein (=0)
gedruckt werden soll ...

... werden 21 Blanks
" " gedruckt.

Ansonsten werden
erst Blanks ...

... dann + Zeichen ...

... und zum Schluss wieder Blanks
gedruckt, so dass die
Gesamtbreite 21 Zeichen umfasst.

Die Türme von Hanoi: Animation (Fort.)

Das eigentliche Spiel sah ursprünglich so aus:

```
public static void move(int N, int von, int nach, int ueber) {  
    if (N > 0) {  
        move(N-1,von,ueber,nach);  
        System.out.println("Bewege Stein von " + von + " nach " + nach);  
        move(N-1,ueber,nach,von);  
    }  
}
```

Im neuen Spiel wird der
Stein bewegt

Die Türme müssen
übergeben werden.

```
public static void move(int[][] towers,int N,int von,int nach,int ueber) {  
    if (N > 0) {  
        move(towers,N-1,von,ueber,nach);  
        moveStone(towers,von,nach);  
        printTowers(towers);  
        move(towers,N-1,ueber,nach,von);  
    }  
}
```

bewege den obersten Stein
von "von" nach "nach" ...

... und drucke das
Spielfeld aus.

Go!

Die Türme von Hanoi: Animation (Fort.)

Zum Bewegen eines Steins müssen die Türme übergeben werden.

```
public static void moveStone(int[][] towers,int iFrom,int iTo) {  
    for(int i = 0;i < towers[iFrom].length;++i) {  
        if (towers[iFrom][i] != 0) {  
            int iStone = towers[iFrom][i];  
            towers[iFrom][i] = 0;  
            for(int j = 0;j < towers[iTo].length;++j) {  
                if (towers[iTo][j] != 0) {  
                    towers[iTo][j-1] = iStone;  
                    return;  
                }  
            }  
            towers[iTo][towers[iTo].length-1] = iStone;  
            return;  
        }  
    }  
}
```

Suche von oben

... bis der 1. Stein gefunden ist.

Merke in den Stein und lösche ihn.

Suche im Zielturm den 1. Stein ...

... und speicher darüber den Stein ab. Fertig!

Vielleicht gibt es noch gar kein Stein im Zielturm. Dann speichere den Stein ganz unten ab. Fertig!

Die Türme von Hanoi: Animation (Fort.)

Diskussion:

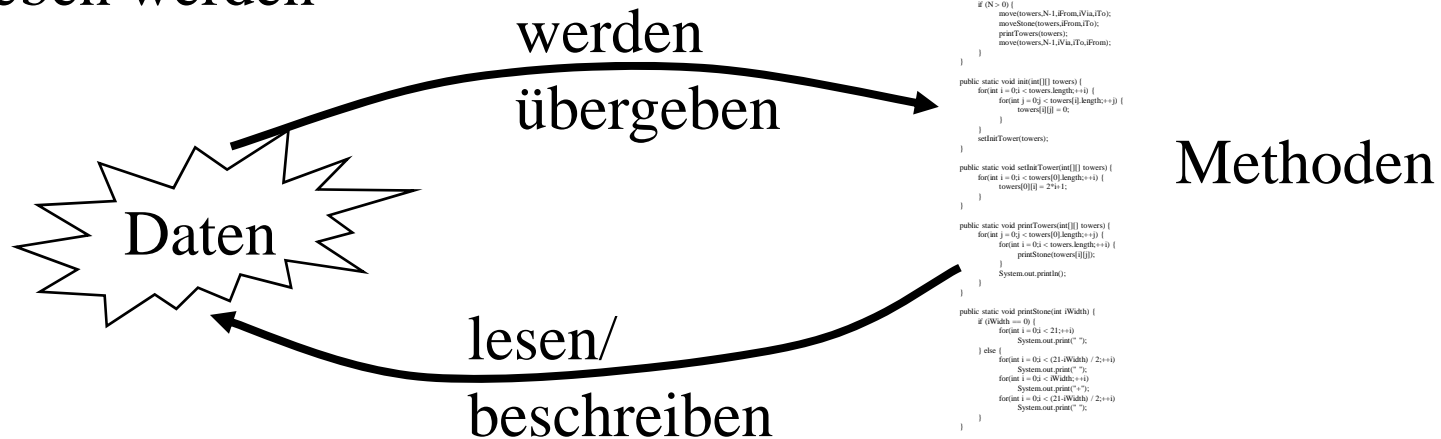
- aus einem einfachen Programm mit 14 Zeilen ist ein sehr komplexes Programm mit 74 Zeilen geworden
- die Datenstruktur für die Verwaltung der Türme (2-dim. int-Array) ist sehr zentral
- das 2-dim. int-Array wird fast allen Methoden übergeben

Vorlesung 9/2

Diskussion

Klassische imperative Programmierung:

- Trennung von Daten und ihrer Manipulation
- hier:
 - 2-dim. int-Array auf der einen Seite
 - Methoden, die das Array lesen/beschreiben auf der anderen Seite
- die Kommunikation erfolgt, indem die Daten den Methoden übergeben werden



```
public class Hanoi2 {  
    public static void move(int[] towers, int N, int fFrom, int fTo, int fVia) {  
        if (N > 0) {  
            moveTowers(N-1, fFrom, fVia, fTo);  
            moveSomeTowers(fFrom, fTo);  
            printTowers(towers);  
            moveTowers(N-1, fVia, fTo, fFrom);  
        }  
    }  
  
    public static void init(int[] towers) {  
        for(int i = 0; i < towers.length; i++) {  
            towers[i] = 0; // towers[i][j] = 0;  
        }  
        setInitTower(towers);  
    }  
  
    public static void setInitTower(int[] towers) {  
        for(int i = 0; i < towers[0].length; i++) {  
            towers[0][i] = 2*i+1;  
        }  
    }  
  
    public static void printTowers(int[] towers) {  
        for(int j = 0; j < towers[0].length; j++) {  
            for(int i = 0; i < towers.length; i++) {  
                printSomeTowers(towers[i][j]);  
            }  
            System.out.println();  
        }  
    }  
  
    public static void printSome(int WWidth) {  
        if (WWidth == 0) {  
            for(int i = 0; i < 21; i++)  
                System.out.print(" ");  
        } else {  
            for(int i = 0; i < (21-WWidth) / 2; i++)  
                System.out.print(" ");  
            for(int i = 0; i < WWidth; i++)  
                System.out.print(" ");  
            for(int i = 0; i < (21-WWidth) / 2; i++)  
                System.out.print(" ");  
        }  
    }  
}
```

Diskussion (Fort.)

Nachteil der Trennung von Daten und Methoden:

- eine Methode muss die Bedeutung der Daten kennen, um sinnvoll mit ihnen arbeiten zu können
- keiner kann garantieren, dass diesen Methoden nicht andere Daten von gleichem Typ mit einer anderen Bedeutung übergeben werden

• Bsp.:

```
public static void setInitTower(int[][] towers) {  
    for(int i = 0; i < towers[0].length; ++i) {  
        towers[0][i] = 2*i+1;  
    }  
}
```

setInitTower geht davon aus, dass es mindestens einen Eintrag im Array towers gibt. Ist dem nicht so, wird es einen Laufzeitfehler geben.

Diskussion (Fort.)

moveStone hat folgende Annahmen an die Daten, die übergeben werden:

1. iFrom und iTo sind gültige Indizes der 1. Dimension von towers
2. in dem Zielturm iTo gibt es noch mindestens einen freien Platz, d.h. es muss immer gelten:
`towers[iTo][0] == 0`

```
public static void moveStone(int[][] towers,int iFrom,int iTo) {  
    for(int i = 0;i < towers[iFrom].length;++i) {  
        if (towers[iFrom][i] != 0) {  
            int iStone = towers[iFrom][i];  
            towers[iFrom][i] = 0;  
            for(int j = 0;j < towers[iTo].length;++j) {  
                if (towers[iTo][j] != 0) {  
                    towers[iTo][j-1] = iStone;  
                    return;  
                }  
            }  
            towers[iTo][towers[iTo].length-1] = iStone;  
            return;  
        }  
    }  
}
```

Klassen und Objekte

Idee bei der objektorientierten Programmierung:

- Methoden werden immer für bestimmte Daten entwickelt
- Daten werden so konzipiert, dass sie nur von bestimmten Methoden verarbeitet werden soll
- Daten sollen nicht mit Methoden zu tun haben, die nicht für sie entwickelt worden sind (und umgekehrt)

d.h.

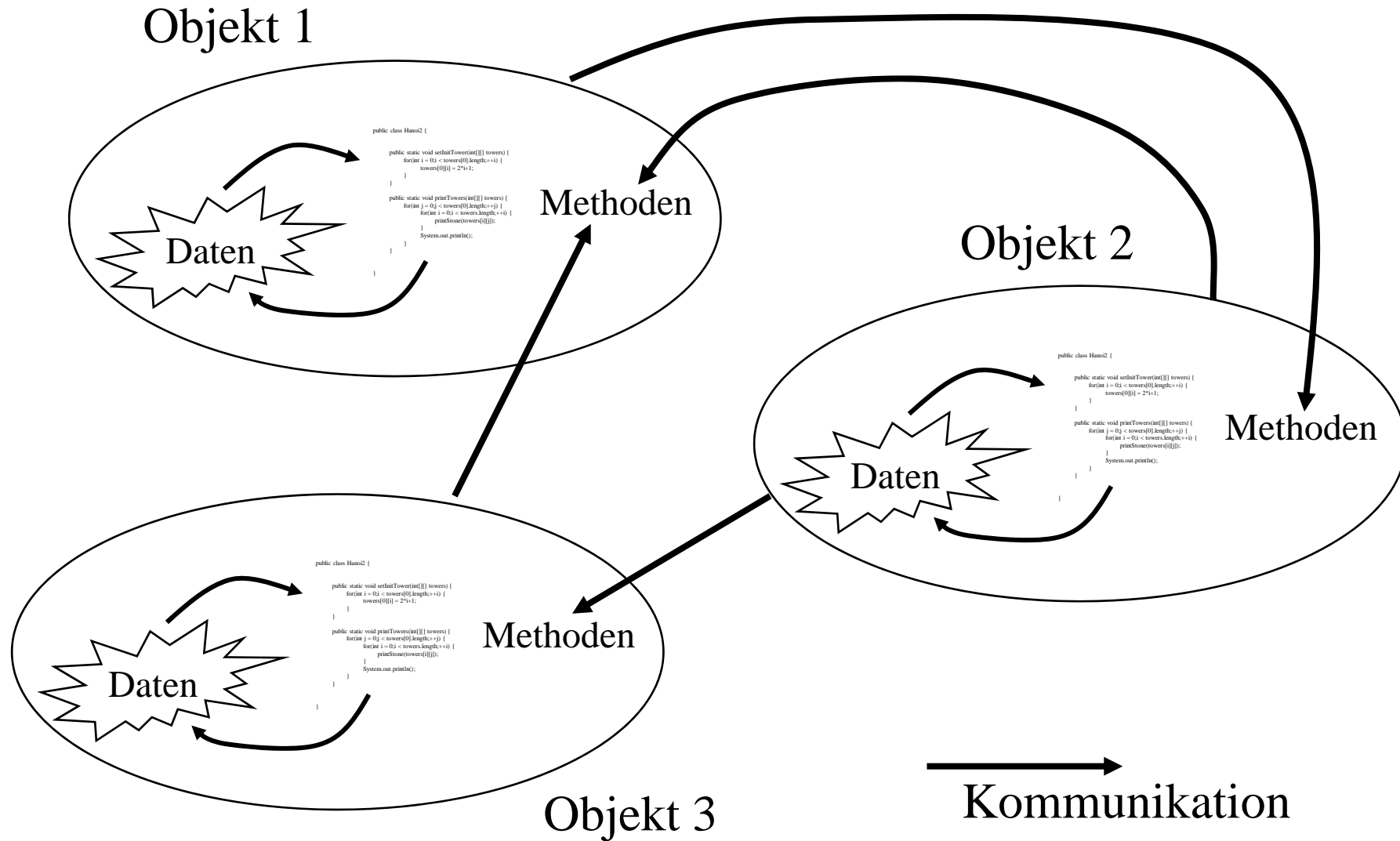
- Daten und ihre Methoden sollen wesentlich enger zusammenrücken

Klassen und Objekte (Fort.)

Idee bei der objektorientierten Programmierung:

- Methoden und ihre *zugehörigen* Daten bilden sogenannte *Objekte*
- während des Programmablaufs kann es viele Objekte geben
- es können *neue* Objekte *entstehen*
- es können Objekte *sterben*
- Objekte können miteinander kommunizieren, indem ein Objekt die Methoden eines anderen Objekts aufruft

Klassen und Objekte (Fort.)



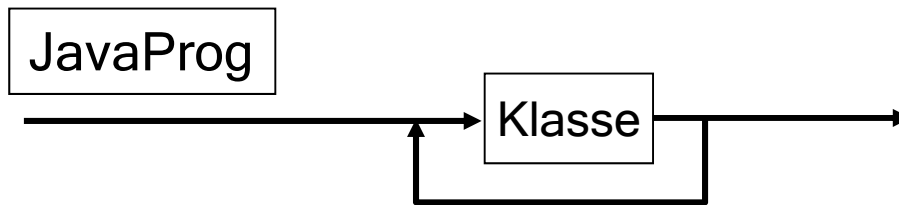
Klassen und Objekte (Fort.)

- Objekte sind die Einheiten, die während des Programmablaufs existieren
- ihre Eigenschaften (welche Daten werden gespeichert, welche Methoden gibt es) werden zum Programmierzeitpunkt festgelegt
- ihre Eigenschaften sind unabhängig vom Programmablauf
- die Festlegung der Eigenschaften erfolgt in einer sogenannten Klasse
- Analogien:

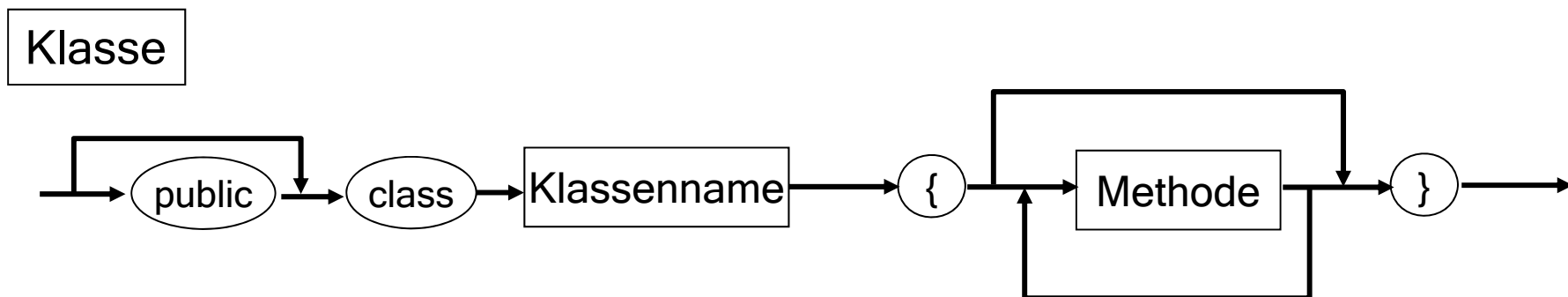
Typ	--	Wert
Klasse	--	Objekt
- in der OOP ist eine Klasse ein Typ
- ein Objekt ist ein Wert einer Variablen, die vom Typ der entsprechenden Klasse ist

Klassen und Objekte (Fort.)

- ein Java Programm besteht auf einer Menge von Klassen



- eine Klasse kann mit dem Wort *public* anfangen



- in jeder .java Datei darf nur genau eine Klasse mit public beginnen, der Klassenname muss gleich dem Dateinamen sein

Go!

Klassen und Objekte (Fort.)

- Beispiel für ein Java-Programm mit 2 Klassen
- Das Beispiel ist abzuspeichern in der Datei "Top.java"

```
class Juhu {  
    public static void doit() {  
        System.out.println("ich bin die Methode von juhu");  
    }  
}  
  
public class Top {  
    public static void main(String[] args) {  
        System.out.println("ich bin die Hauptmethode");  
    }  
}
```

Klassen und Objekte (Fort.)

- das vorangegangene Programm führt nur die Hauptmethode aus, weil
- es zwar eine Klasse namens Juhu gibt,
- es ist jedoch kein Objekt von dieser Klasse erzeugt worden
- Objekte werden analog zu Arrays mittels new erzeugt



Go!

Klassen und Objekte (Fort.)

- analog zu Arrays gibt die print-Anweisung zu einem Objekt nur die Adresse aus, unter der das Objekt im Speicher liegt

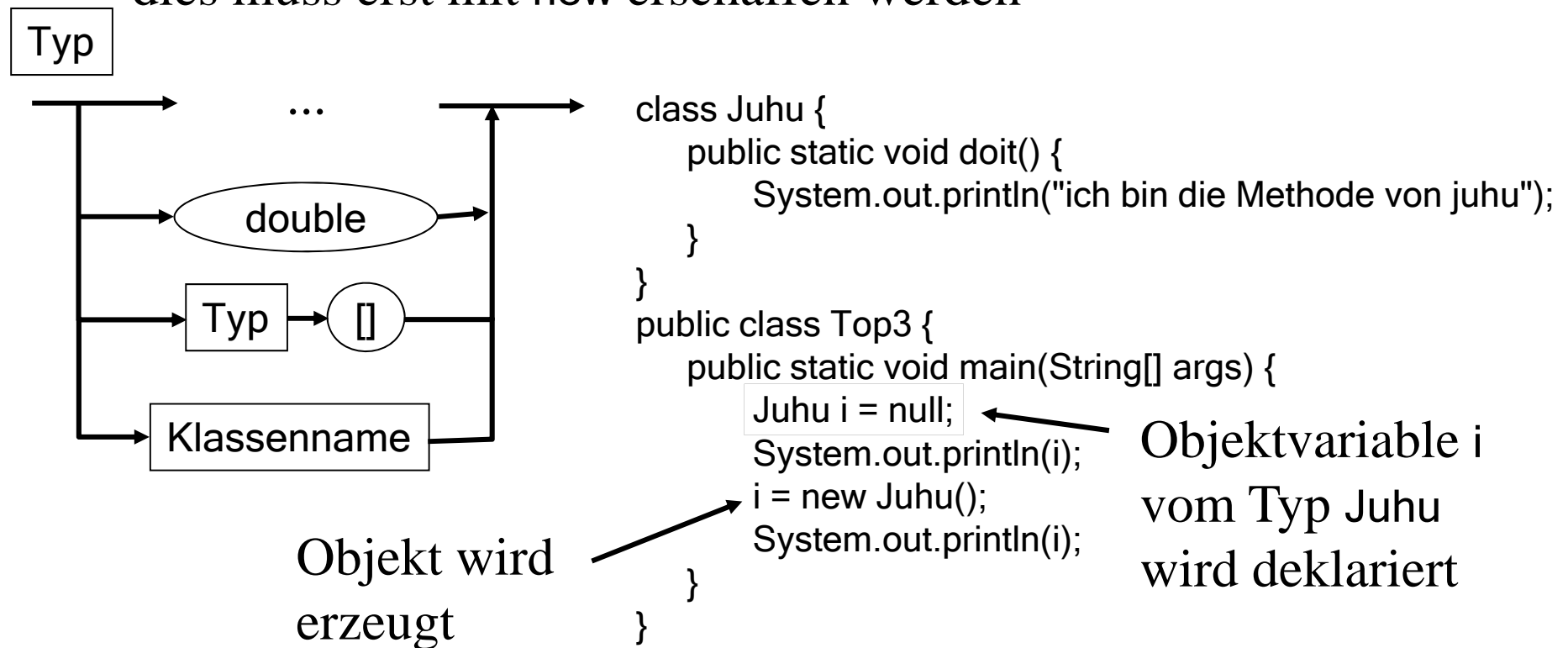
```
class Juhu {  
    public static void doit() {  
        System.out.println("ich bin die Methode von juhu");  
    }  
}
```

```
public class Top2 {  
    public static void main(String[] args) {  
        System.out.println(new Juhu());  
    }  
}
```

Go!

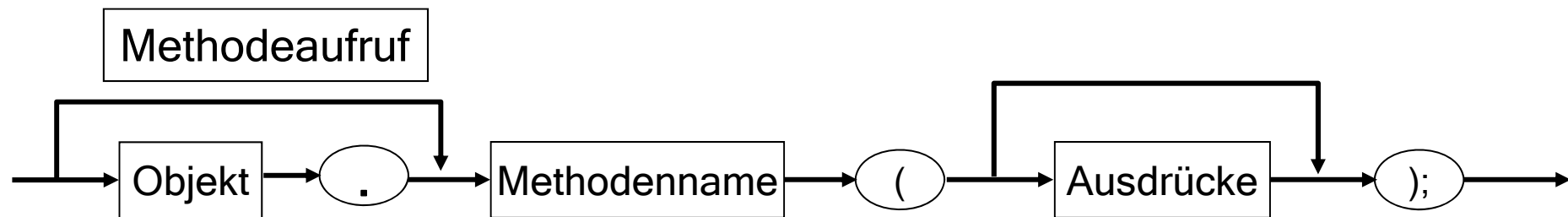
Klassen und Objekte (Fort.)

- Objekte können in Variablen gespeichert werden
- dazu gilt der Klassenname als Typ für die Variable
- analog zu Arrays enthalten solche Variablen noch kein Objekt
- dies muss erst mit new erschaffen werden

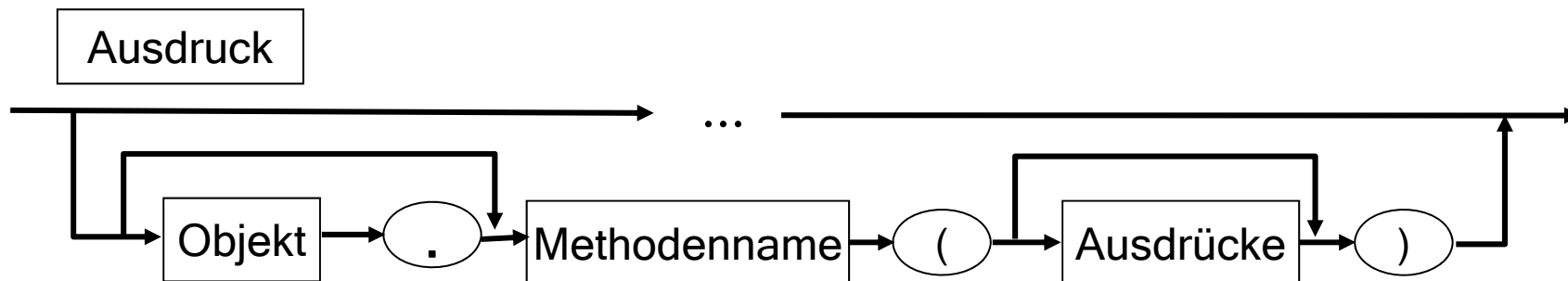


Klassen und Objekte (Fort.)

- obwohl ein Objekt der Klasse Juhu erzeugt worden ist, wird immer noch nicht die Methode doit abgearbeitet
- um eine Methode eines Objektes aufzurufen, muss dem Objekt ein Punkt folgen und dann der Methodename



- analoges gibt beim Methodenaufruf mit Rückgabewert



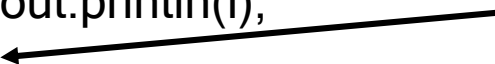
Go!

Klassen und Objekte (Fort.)

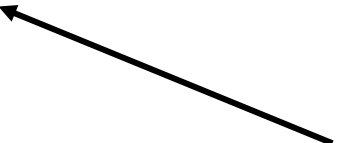
```
class Juhu {  
    public static void doit() {  
        System.out.println("ich bin die Methode von juhu");  
    }  
}
```

```
public class Top4 {  
    public static void main(String[] args) {  
        Juhu i = null;  
        System.out.println(i);  
        i = new Juhu();  
        System.out.println(i);  
        i.doit();  
        (new Juhu()).doit();  
    }  
}
```

doit Methode
vom Objekt i
wird aufgerufen



doit Methode eines neuen,
anonymen Objekts wird
aufgerufen

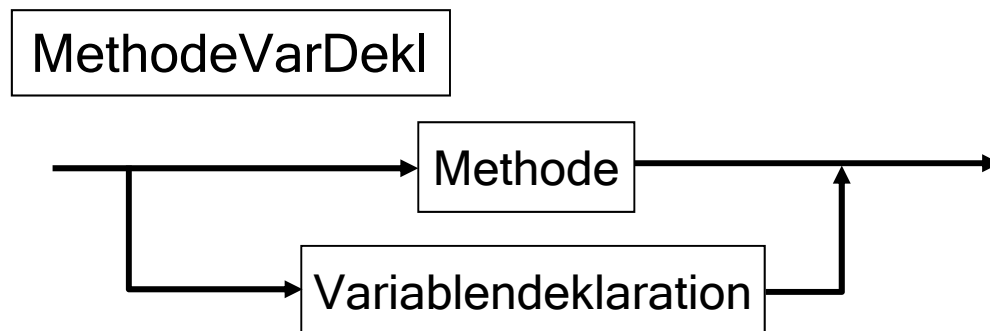
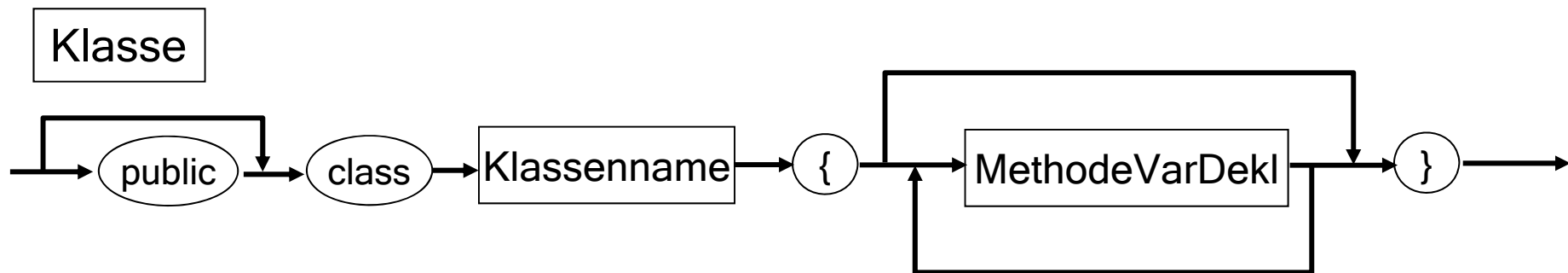


Klassen und Objekte (Fort.)

- die Top-Programme *haben* bisher noch *keine Daten* mit den Objekten assoziiert
- damit ein Objekt Daten speichern kann, muss *in der Klasse* sogenannte *Objektvariablen* deklariert werden
- dies sind Variablen, die *nicht zu einer Methode gehören* und mit ihr wieder verschwinden, sondern
- Objektvariablen *entstehen* zusammen *mit dem Objekt*
- sie *behalten ihren Wert* während der gesamten Objektlebenszeit
- sie *verschwinden* mit dem *Sterben des Objekts*
- verschiedene *Objekte* einer Klasse haben ihre *eigenen Objektvariablen*

Klassen und Objekte (Fort.)

- eine Klasse kann neben Methoden auch Deklarationen von Variablen enthalten



Klassen und Objekte (Fort.)

Beispiel:

das nebenstehende Programm
erzeugt den folgenden Fehler:

```
Top5.java:3: non-static variable  
i cannot be referenced from a  
static context
```

```
System.out.println(++i);
```

Grund:

static vor doit zeigt an, dass doit
keine Objektmethode ist

```
1 class Juhu {  
2     public static void doit() {  
3         System.out.println(++i);  
4     }  
5  
6     int i;  
7 }  
8  
9 public class Top5 {  
10    public static void main(String[] args) {  
11        Juhu i = new Juhu();  
12        Juhu k = new Juhu();  
13        i.doit();  
14        i.doit();  
15        k.doit();  
16    }  
17 }
```

Objekt- und Klassenmethoden (Fort.)

- um in der Methode `doit` auf die Objektvariable `i` zugreifen zu können, ***muss es ein Objekt geben***
- das Schlüsselwort `static` besagt, dass diese ***Methode auch aufgerufen werden kann***, wenn es ***kein Objekt*** gibt
- wenn es aber kein Objekt gibt, welches `i` soll dann gelesen werden?
- da es hierauf keine vernünftige Antwort gibt, liefert der Compiler einen Fehler
- wird das Schlüsselwort `static` weggelassen, so kann die ***Methode nur aufgerufen werden***, wenn es auch ***ein Objekt gibt***

```
public static void doit() {  
    System.out.println(++i);  
}
```

Objekt- und Klassenmethoden (Fort.)

- Methoden können also mit static oder ohne definiert werden
- mit static:
 - *sogenannte Klassenmethoden*,
 - können auch *ohne Objekte aufgerufen* werden
 - können *nicht* auf die *Objektvariablen zugreifen*
- ohne static:
 - *sogenannte Objektmethoden*
 - können *nur mit Objekten aufgerufen* werden
 - können auf die *Objektvariablen zugreifen*

Go!

Beispiel

```
class Juhu {  
    public void doit() {  
        System.out.println(++i);  
    }  
    int i;  
}
```

Objektmethode: gehört zum Objekt, kann auf die Objektvariable i zugreifen

Objektvariable: gehört zum Objekt, nicht zu irgendeiner Methode

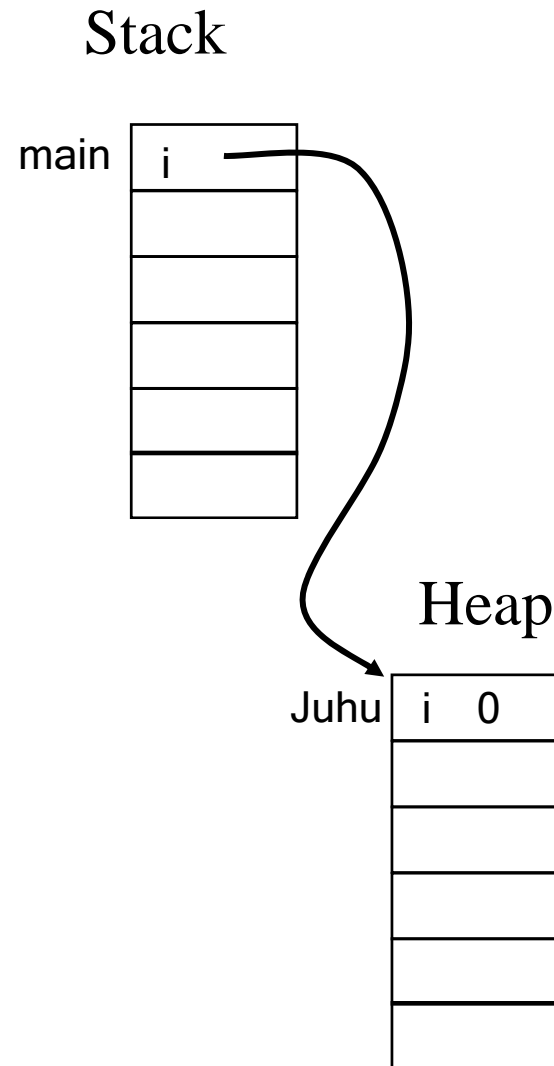
```
public class Top5 {  
    public static void main(String[] args) {  
        Juhu i = new Juhu();  
        Juhu k = new Juhu();  
        i.doit();  
        i.doit();  
        k.doit();  
    }  
}
```

2 Objekte werden angelegt

ruft 2-mal doit von Objekt i auf

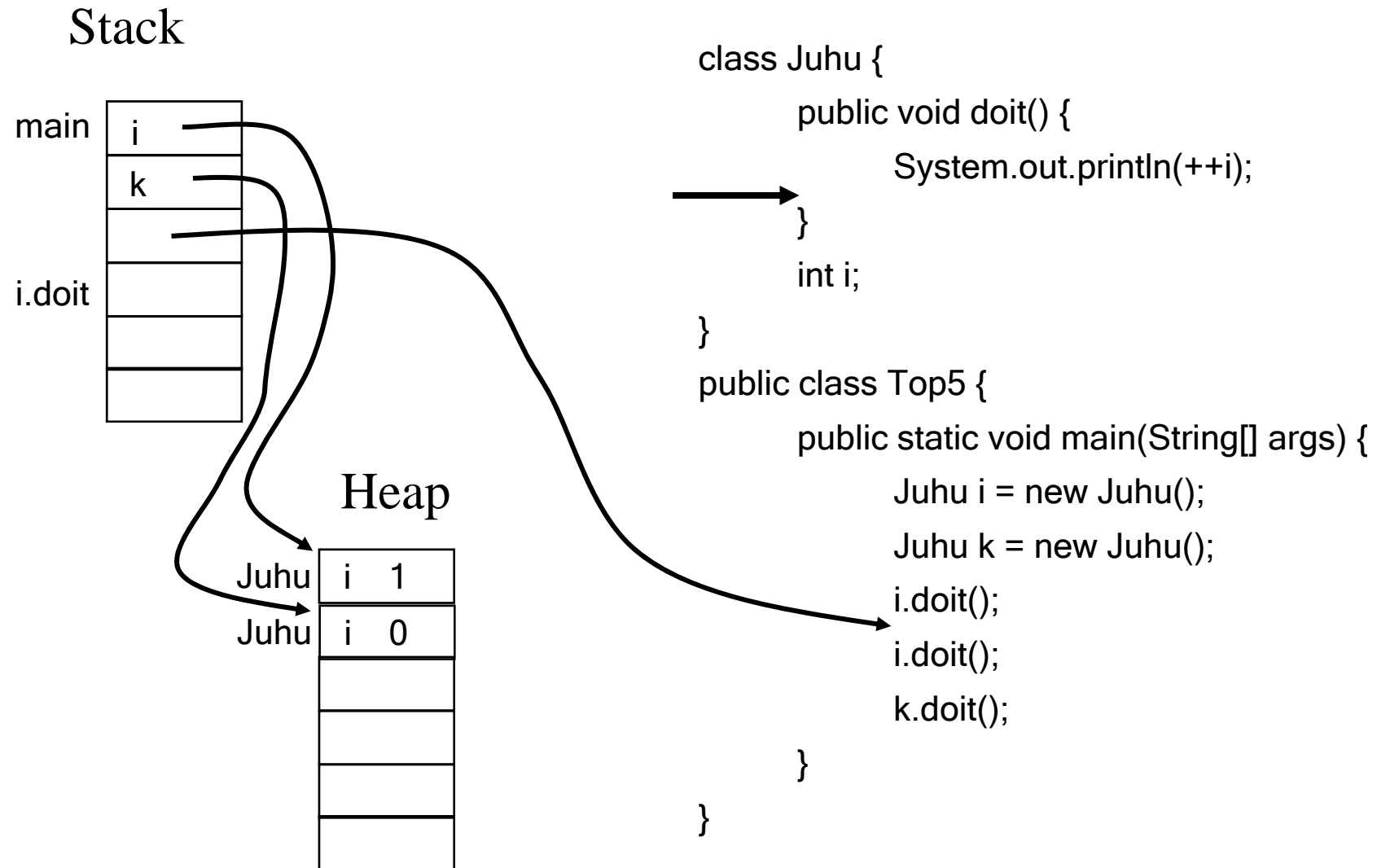
ruft 1-mal doit von Objekt k auf

Beispiel (Fort.)

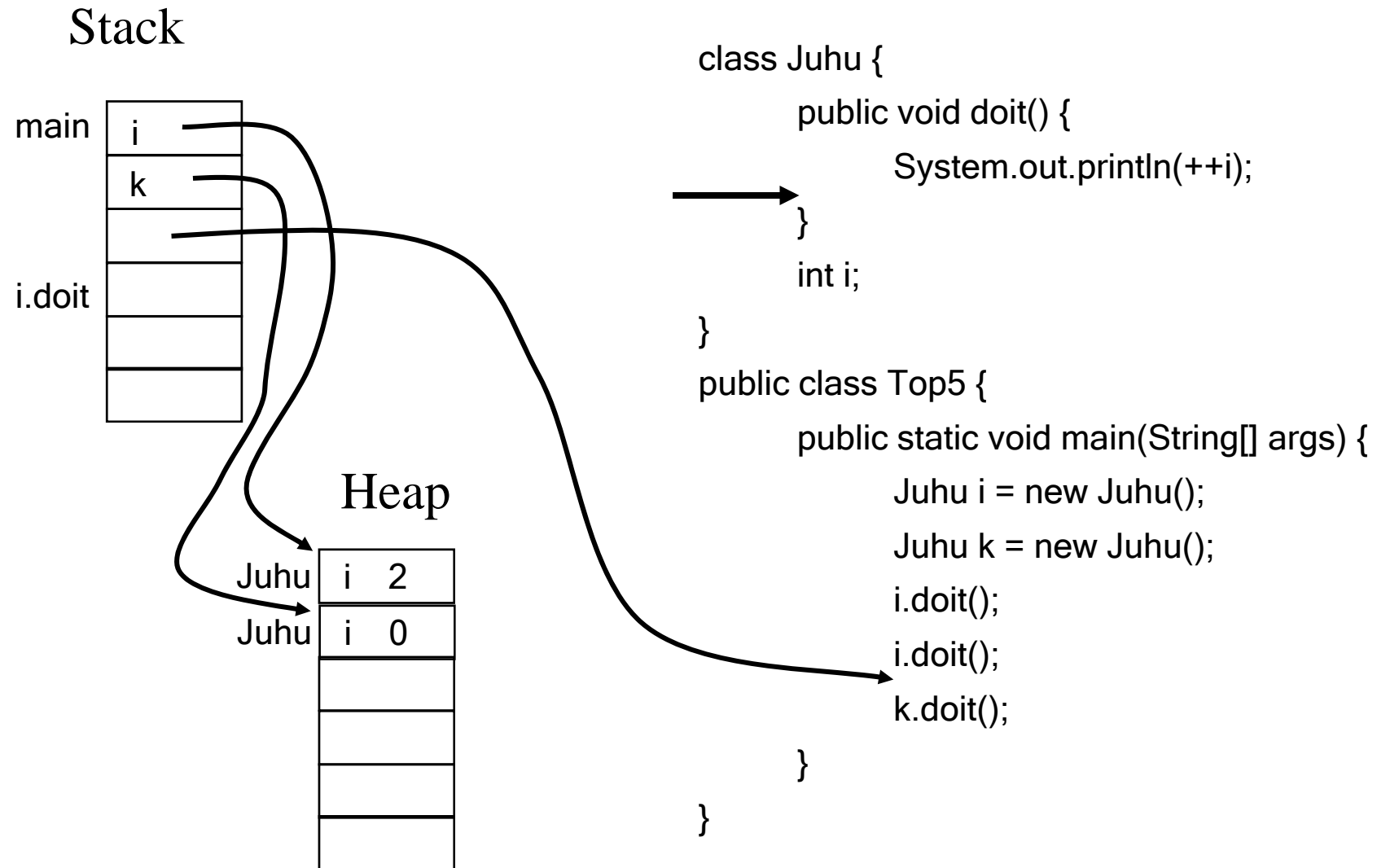


```
class Juhu {  
    public void doit() {  
        System.out.println(++i);  
    }  
    int i;  
}  
  
public class Top5 {  
    public static void main(String[] args) {  
        Juhu i = new Juhu();  
        Juhu k = new Juhu();  
        i.doit();  
        i.doit();  
        k.doit();  
    }  
}
```

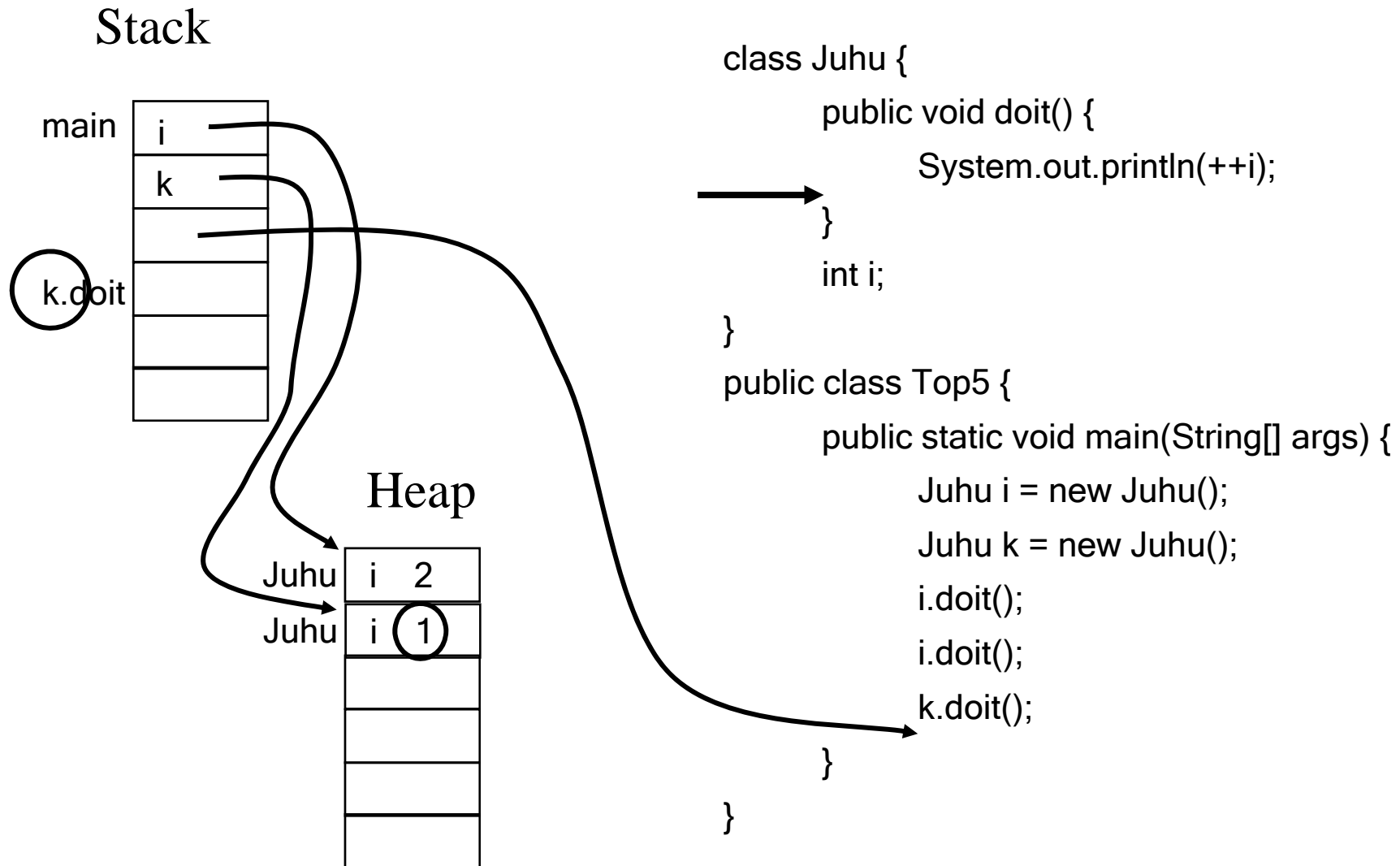
Beispiel (Fort.)



Beispiel (Fort.)



Beispiel (Fort.)

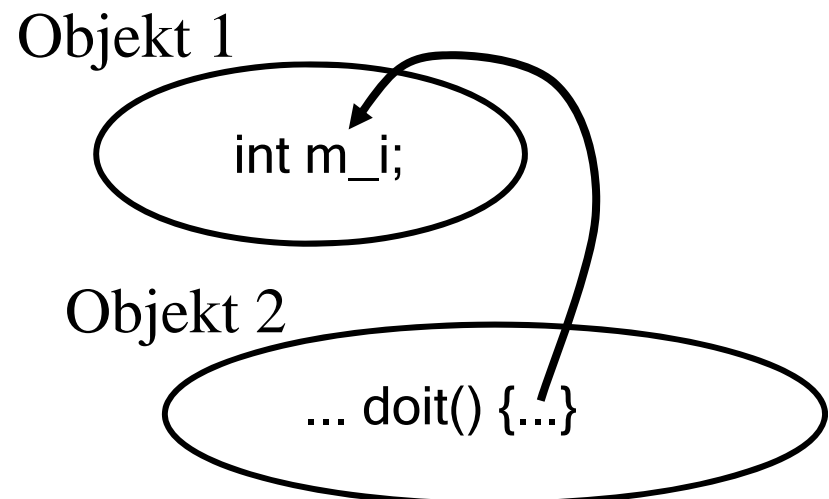
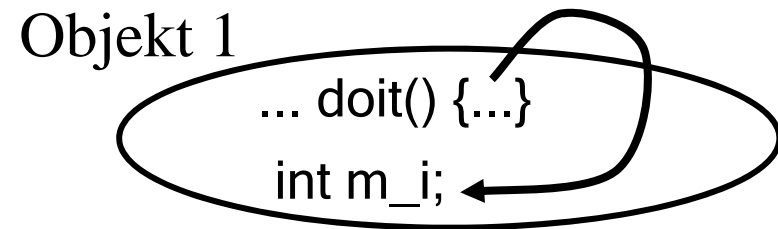


Vorlesung 10/1

Zugriff auf Objektvariablen

Möchte man auf *Objektvariablen zugreifen* (lesend oder schreibend), so müssen 2 Situationen unterschieden werden:

- man befindet sich in einer Methode eines Objekts und möchte auf die *eigenen Objektvariablen zugreifen*
- man hat ein Objekt gegeben und möchte auf *dessen Objektvariablen zugreifen*, d.h. man möchte auf die Variablen eines anderen Objekts zugreifen



Zugriff auf Objektvariablen (Fort.)

- auf seine eigenen Variablen kann durch den einfachen Namen der Variablen zugegriffen werden

```
class A {  
    int m_i;  
    public void doit() {    greift auf sein  
        m_i = 42;          eigenes m_i zu  
    }  
}
```

- auf die Variablen eines anderen Objekts greift man zu, indem das Objekt mit einem Punkt "." vorangestellt wird (analog zum Methodenaufruf)

```
class B {  
    public void doit() {  
        A juhu = new A();    greift auf m_i vom  
        juhu.m_i = 42;       Objekt juhu zu  
    }  
}
```

Go!

Beispiel

```
class A {
```

```
    public void doit() {  
        System.out.println(m_j++);  
    }
```

doit greift auf die eigene
Objektvariable m_j zu

```
    int m_j;  
}
```

```
public class ObjVar {
```

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

```
        A juhu = new A();
```

```
        juhu.doit();
```

```
        juhu.m_j = 13;
```

```
        juhu.doit();
```

```
        juhu.doit();
```

```
        System.out.println(juhu.m_j);
```

```
    }
```

```
}
```

Was gibt dieses
Programm aus?

hier wird auf die Objekt-
variable m_j des Objekts
juhu zugegriffen

Objekterzeugung

folgende Situation:

- eine Klasse hat ein int-Array als Objektvariable
- eine Methode initialisiert diese Objektvariable
- eine andere Methode greift auf dieses int-Array zu

Objekterzeugung (Fort.)

```
class A {  
    int[] m_arField;  
    public void init() {  
        m_arField = new int[10];  
    }  
    public int get(int i) {  
        return m_arField[i];  
    }  
}
```

das Array ist bei der Objekterzeugung noch nicht angelegt

dies wird von der init-Methode erledigt

```
public class B {  
    public static void main(String[] args) {
```

A a = new A(); Objekt der Klasse A wird erzeugt

a.init(); die init-Methode des Objekts wird ausgeführt

System.out.println(a.get(5)); das Ergebnis der get-Methode des Objekts wird ausgegeben

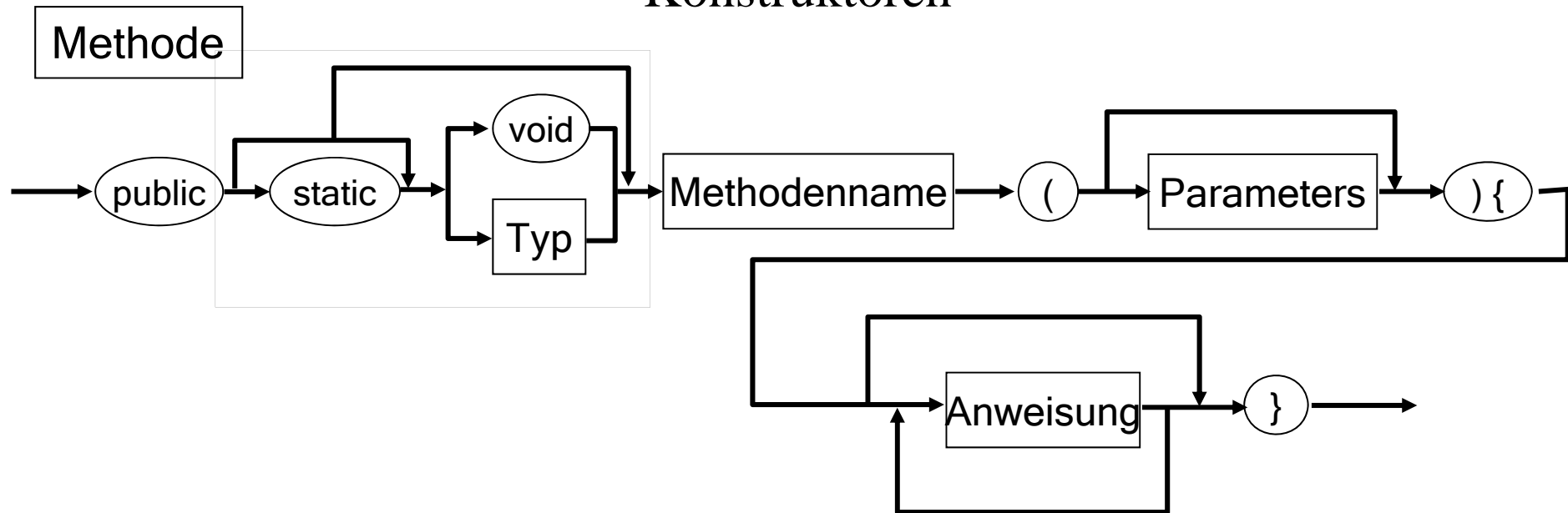
```
    }  
}
```

Objekterzeugung (Fort.)

Problem:

- wenn die init-Methode *nicht* aufgerufen wird, erzeugt die get-Methode einen Fehler (NullPointerException)
- d.h. *nach* jeder *Erzeugung* eines A Objekts sollte *zuerst* die *init-Methode* aufgerufen werden bevor eine andere Methode aufgerufen wird
- d.h. die *Objekterzeugung* und die anschließende *init-Methode* bilden eine *Einheit*
- diese Einheit wird in OOP durch sogenannte *Konstruktoren* realisiert
- Konstruktoren enthalten Code, der unmittelbar nach Erzeugung eines Objekts ausgeführt wird

Konstruktoren



- ***Konstruktoren*** werden ***wie Methoden*** gebildet
- sie dürfen ***nicht static*** sein
- sie haben weder einen Ergebnistypen noch sind sie vom Typ void
- der ***Name eines Konstruktors*** muss identisch zum ***Klassenname*** sein

Konstrukturen (Fort.)

```
class A {  
    int[] m_arField;
```

```
    public A() {  
        m_arField = new int[10];  
    }
```

dies ist der Konstruktor: kein
static, kein void, kein Typ

```
    public int get(int i) {  
        return m_arField[i];  
    }  
}
```

```
public class B {  
    public static void main(String[] args) {  
        A a = new A();  
        System.out.println(a.get(5));  
    }  
}
```

bei der Erzeugung von A wird
der Konstruktor ausgeführt

es muss kein init mehr aufgerufen
werden; gibt es auch nicht mehr

Konstruktoren (Fort.)

Vorteil von Konstruktoren:

- die *Initialisierung* des Objekts kann nicht mehr vergessen werden, da sie unmittelbar mit der *Objekterzeugung* verbunden ist
- damit gibt es *keinen Zeitpunkt* mehr, an dem das *Objekt inkonsistent* ist, d.h. einen illegalen Zustand besitzt
- die *Verantwortung* für ein korrektes Objekt ist vom Erzeuger des Objekts zu dem *Objekt selber* verschoben worden

Konstruktoren (Fort.)

- Konstruktoren können auch Parameter bei der Erzeugung des Objekts übergeben werden
- diese Parameter können dann während der Initialisierung ausgewertet werden

```
class A {  
    int[] m_arField;  
    public A(int iSize) {  
        m_arField = new int[iSize];  
    }  
    public int get(int i) {  
        return m_arField[i];  
    }  
}  
...
```

Konstruktor hat einen Parameter iSize

aktueller Parameter wird bei Objekterzeugung angegeben

```
...  
public class B {  
    public static void main(String[] args) {  
        A a = new A(10);  
        System.out.println(a.get(5));  
    }  
}
```

Go!

Beispiel für Konstruktoren

```
class A {  
    public A(int iDepth) {  
        System.out.println("cool, it's a " + iDepth);  
    }  
}
```

Klasse A mit
Konstruktor

```
class B {  
    public B() {  
        System.out.println("ich bin der Konstruktor von B");  
    }  
    public void doit() {  
        System.out.println("ich bin doit von B");  
    }  
}
```

Klasse B mit
Konstruktor
und Methode

```
public class Konstruktor {  
    public static void main(String[] args) {  
        A a1 = new A(5);  
        A a2 = new A(20);  
        new B().doit();  
        new A(17);  
    }  
}
```

3 A Objekte und 1 B
Objekt werden erzeugt

Go!

Beispiel für Konstruktoren

```
class A {
```

```
    public A(int i) {
```

```
        new Konstruktor2(i+1);
```

```
        System.out.println(i);
```

```
    }
```

```
}
```

erzeuge ein Objekt der Klasse
Konstruktor2 und drucke
anschließend die Zahl i

```
public class Konstruktor2 {
```

```
    public Konstruktor2(int i) {
```

```
        System.out.println(i);
```

```
        new A(i+1);
```

```
    }
```

drucke die Zahl i und erzeuge
anschließend ein A Objekt

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

```
        new A(0);
```

```
    }
```

```
}
```

erzeuge ein A Objekt

Was gibt dieses
Programm aus?

Go!

Beispiel für Konstruktoren

```
class A {  
    public A(int i) {  
        m_j = new Konstruktor3(i).m_i;  
    }  
    int m_j;  
}
```

erzeuge 1 Konstruktor3 Objekt und
speichere dessen Objektvariable
(m_i) in der eigenen (m_j) ab

```
public class Konstruktor3 {  
    public Konstruktor3(int i) {  
        if (i != 1)  
            m_i = new A(i % 2 == 0 ? i / 2 : i * 3 + 1).m_j;  
        else  
            m_i = i;  
    }  
    public static void main(String[] args) {  
        System.out.println(new A(21).m_j);  
    }  
    int m_i;  
}
```

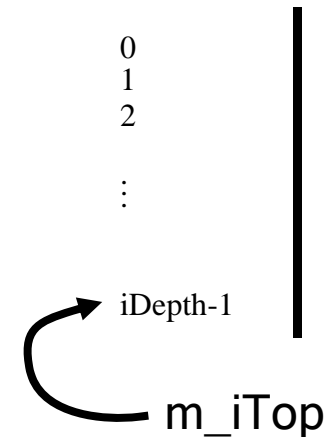
erzeuge ein A Objekt und
speichere dessen Objektvariable
(m_j) in der eigenen (m_i) ab

erzeuge ein A Objekt und
drucke dessen Objektvariable

Was gibt dieses
Programm aus?

Türme von Hanoi: objektorientiert

- 2 Klassen sind offensichtlich
 - eine Klasse modelliert einen Turm: class Tower
 - eine Klasse enthält 3 Türme und modelliert das eigentliche Spiel: class Game



```
class Tower {
```

```
    public Tower(int iDepth) {  
        m_arStones = new int[iDepth];  
        m_iTop = iDepth-1;  
    }
```

bei der Erzeugung wird gesagt,
wieviele Steine maximal
abgelegt werden können

```
...
```

```
    int[] m_arStones;  
    int m_iTop;  
}
```

zunächst liegt kein Stein auf dem Turm

Turm merkt sich Steine und Position,
an der der nächste Stein abgelegt wird

Türme von Hanoi: objektorientiert (Fort.)

- ein Stein wird auf einem Turm abgelegt indem
 - an der Stelle `m_iTop` der Stein gespeichert wird
 - `m_iTop` um 1 erniedrigt wird

der abzulegende Stein
wird übergeben

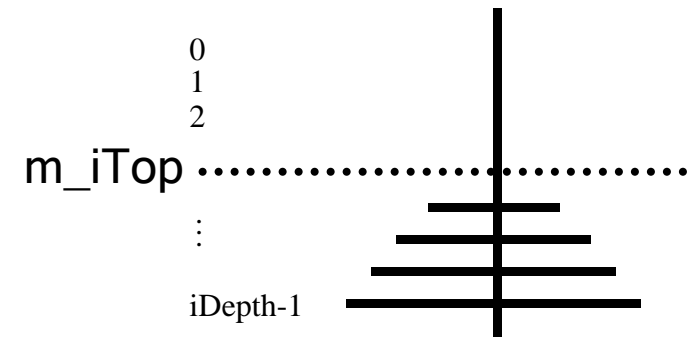
...

```
public void addStone(int iStone) {  
    m_arStones[m_iTop--] = iStone;  
}
```

...

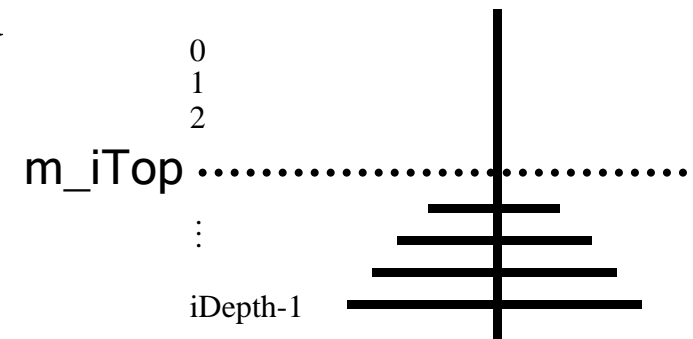
... und erniedrige
danach `m_iTop`

speichere den Stein ab ...



Türme von Hanoi: objektorientiert (Fort.)

- ein Stein wird vom Turm entfernt, indem
 - zuerst `m_iTop` um 1 erhöht wird
 - der Stein, der bei `m_iTop` liegt, gemerkt wird
 - durch Überschreiben mit 0 gelöscht wird
 - der gemerkte Stein zurückgegeben wird



liefere den gelöschten
Stein zurück

...

```
public int removeStone() {  
    int res = m_arStones[++m_iTop];  
  
    m_arStones[m_iTop] = 0;  
    return res;  
}
```

erhöhe zuerst `m_iTop` und
speichere den Wert ...

... und lösche den Stein

...

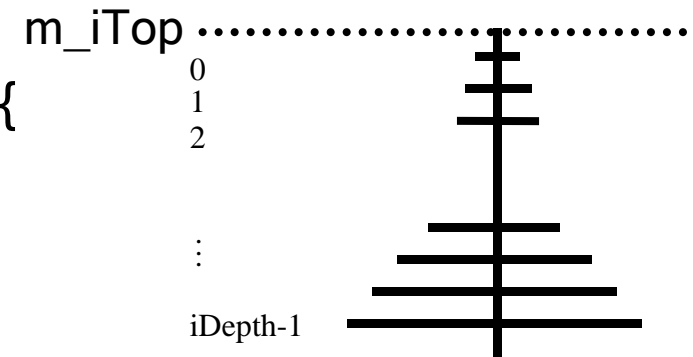
Türme von Hanoi: objektorientiert (Fort.)

- ein Turm wird mit Scheiben initialisiert, indem
 - auf allen Positionen Steine abgelegt werden
 - `m_iTop` auf `-1` gesetzt wird, weil `m_iTop` immer auf den nächsten freien Platz zeigt, es jetzt aber keinen freien Platz mehr gibt
- der Turm kann *nicht im Konstruktor initialisiert* werden, da *nur 1 und nicht alle 3 Türme* zum Anfang Scheiben haben

...

```
public void init() {  
    for(int i = 0; i < m_arStones.length; ++i) {  
        m_arStones[i] = i*2+1;  
    }  
    m_iTop = -1;  
}
```

...



Türme von Hanoi: objektorientiert (Fort.)

- das Drucken eines Steines eines Turms erfolgt analog zu der ursprünglichen Implementierung
- hier wird jedoch die zu druckende Breite als Parameter übergeben

...

```
public void print(int iWidth,int iPos) {  
    if (m_arStones[iPos] == 0) {  
        for(int i = 0;i < iWidth;++i)    wenn kein Stein dort  
            System.out.print(" ");      liegt, drucke Blanks  
    } else {  
        for(int i = 0;i < (iWidth-m_arStones[iPos]) / 2;++i)  
            System.out.print(" ");  
        for(int i = 0;i < m_arStones[iPos];++i)    drucke Blanks vor  
            System.out.print("+");                und nach dem Stein  
        for(int i = 0;i < (iWidth-m_arStones[iPos]) / 2;++i)  
            System.out.print(" ");  
    }  
}
```

Türme von Hanoi: objektorientiert (Fort.)

- ein Spiel besteht aus 3 Türmen
- da die move-Methode durch einen Index auf die Türme zugreifen möchte, werden die drei Türme in einem Array verwaltet (Array von Objekten!!!)

```
class Game {  
    public Game(int iDepth) {  
...  
    }
```

ein Spiel bekommt als Parameter die Anzahl der Scheiben übergeben

```
Tower[] m_arTowers = new Tower[3];  
int m_iDepth;  
}  
    merkt sich 3 Türme und  
    die Anzahl der Scheiben
```

Objektvariablen können bei der Erzeugung initialisiert werden; diese Initialisierung erfolgt noch vor dem Konstruktor

Türme von Hanoi: objektorientiert (Fort.)

- ein Spiel merkt sich die Anzahl der Scheiben
- ein Spiel muss zunächst 3 Türme erzeugen
- der 1. Turm muss mit Scheiben gefüllt werden

...

```
public Game(int iDepth) {
```

```
    m_iDepth = iDepth;
```

```
    m_arTowers[0] = new Tower(m_iDepth);
```

```
    m_arTowers[1] = new Tower(m_iDepth);
```

```
    m_arTowers[2] = new Tower(m_iDepth);
```

```
    m_arTowers[0].init();
```

```
}
```

```
    lege Scheiben
```

...

```
    auf den 1. Turm
```

erzeuge 3 Türme



das Array m_arTowers existiert bereits, da die Objektvariable bei der Erzeugung initialisiert wurde



Türme von Hanoi: objektorientiert (Fort.)

- das Drucken erfolgt analog zu der 1. Implementierung

```
...  
public void print() {  
    for(int i = 0; i < m_iDepth; ++i) {  
        for(int j = 0; j < m_arTowers.length; ++j) {  
            m_arTowers[j].print(m_iDepth * 2 + 3, i);  
        }  
        System.out.println();  
    }  
    for(int i = 0; i < m_iDepth*6+9; ++i)  
        System.out.print("-");  
    System.out.println("\n\n");  
}  
...
```

für jede Ebene ...

... für jeden Turm ...

... drucke den Stein

drucke zum Schluss
noch eine Linie

Türme von Hanoi: objektorientiert (Fort.)

- das Spiel beginnt damit, dass die Türme zunächst ausgedruckt werden
- danach wird das eigentliche Spiel gestartet

```
...  
public void play() {  
    print();  
    playInternal(m_iDepth,0,2,1);  
}  
...
```

schiebe vom 1. Turm ...

... zum 3. Turm ...

... über den 2. Turm

wieviele Scheiben sollen verschoben werden

Türme von Hanoi: objektorientiert (Fort.)

- das eigentliche Spiel entfernt einen Stein vom Turm iFrom und
- legt ihn auf den Turm iTo
- anschließend wird das Spiel erneut ausgedruckt

...

```
public void playInternal(int N,int iFrom,int iTo,int iVia) {  
    if (N > 0) {  
        playInternal(N-1,iFrom,iVia,iTo);  
        int iStone = m_arTowers[iFrom].removeStone();  
        m_arTowers[iTo].addStone(iStone);  
        print();  
        playInternal(N-1,iVia,iTo,iFrom);  
    }  
}
```

entferne den Stein und ...

... lege ihn wieder ab

...

Go!

Türme von Hanoi: objektorientiert (Fort.)

- um das Spiel zu starten, muss zunächst ein Objekt der Klasse Game erzeugt werden
- dem Konstruktor muss mitgeteilt werden, wieviele Steine auf dem 1. Turm abgelegt werden
- danach wird einfach die play-Methode des Spiels aufgerufen

```
public class Hanoi3 {
```

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

```
        Game g = new Game(5);
```

```
        g.play();
```

```
    }
```

```
}
```

Erzeuge ein Spiel mit 5 Steinen



Starte das Spiel



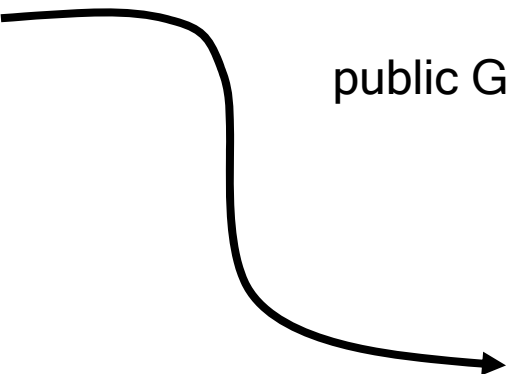
Vorlesung 10/2

Diskussion

- durch die Verwendung von Klassen ist das Programm nicht kürzer geworden
- jedoch ist das Programm übersichtlicher geworden
- es gibt viele Methoden, aber jede einzelne ist recht kurz gehalten
- die Klasse Game muss sich nicht darum kümmern, wie ein Turm seine Steine verwaltet
- der Klasse Tower wird von Game lediglich die Nachricht geschickt, dass ein Stein entfernt werden soll bzw. dass ein Stein abgelegt wird
- wie dieses im einzelnen erfolgt, interessiert an dieser Stelle in der Game Klasse nicht

Diskussion (Fort.)

- jedoch muss die Klasse Game an einer Stelle wissen, dass ein einfacher Turm zunächst keine Steine hat
- im Konstruktor wird daher vom 1. Turm die init-Methode aufgerufen



```
public Game(int iDepth) {  
    m_iDepth = iDepth;  
    m_arTowers[0] = new Tower(m_iDepth);  
    m_arTowers[1] = new Tower(m_iDepth);  
    m_arTowers[2] = new Tower(m_iDepth);  
    m_arTowers[0].init();  
}
```

- schön wäre es, wenn dem Turm bei der Konstruktion mitgeteilt wird, ob Steine auf ihm abzulegen sind oder nicht

Diskussion (Fort.)

mögliche Lösung:

- dem Konstruktor von Tower noch ein Flag mitgeben, dass anzeigt, ob die init-Methode aufgerufen werden soll oder nicht

```
public Tower(int iDepth, boolean blnit) {  
    m_arStones = new int[iDepth];  
    m_iTop = iDepth-1;  
    if (blnit)  
        init(); ← ruft seine eigene  
                init-Methode auf  
}
```

```
public Game(int iDepth) {  
    m_iDepth = iDepth;  
    m_arTowers[0] = new Tower(m_iDepth, true);  
    m_arTowers[1] = new Tower(m_iDepth, false);  
    m_arTowers[2] = new Tower(m_iDepth, false);  
}
```

Diskussion (Fort.)

andere Lösung:

- eine neue Klasse TowerWithStones definieren
- kann alles was Tower auch kann, aber zusätzlich werden bei der Konstruktion Steine auf dem Turm gelegt

```
class Tower {  
    ...  
}  
  
class TowerWithStones {  
    ...  
    kopiere alles aus Tower  
  
    public TowerWithStones(int iDepth) {  
        ...  
    }  
    }  
    zusätzliche Initialisierung  
    ...
```

Diskussion (Fort.)

Problem:

- sehr viel Schreibaufwand
- fehleranfällig, da Änderungen in Tower auch in TowerWithStones nachgeführt werden muss und umgekehrt
- *es funktioniert nicht !!!, weil ...*

```
class Game {  
  
    public Game(int iDepth) {  
        m_iDepth = iDepth;  
        m_arTowers[0] = new TowerWithStones(m_iDepth);  
        m_arTowers[1] = new Tower(m_iDepth);  
        m_arTowers[2] = new Tower(m_iDepth);  
    }  
    ...  
    Tower[] m_arTowers = new Tower[3];  
}
```

Typfehler: in m_arTowers können nur Objekte von Tower aber nicht von TowerWithStone abgespeichert werden

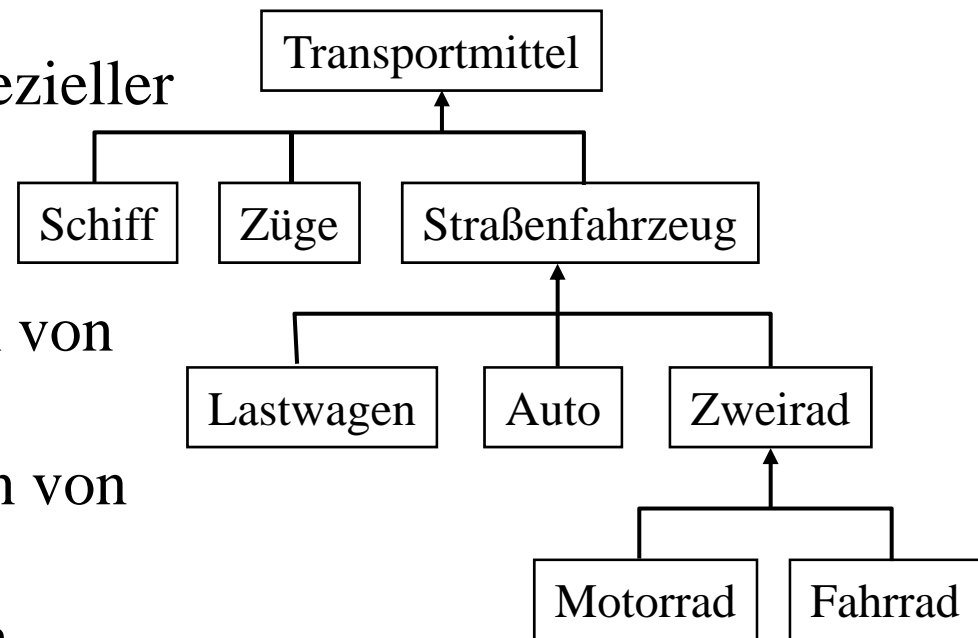
Lösung: Vererbung

Vererbung

- in der Natur treten häufig Beziehung gemäß einer Generalisierung bzw. einer Spezialisierung auf
- eine Spezialisierung findet statt, wenn Klassen einer (oder mehreren) Klassen alle Eigenschaften erben und denen u.U. noch weitere Eigenschaften hinzufügen
- Beispiel: ein Laubbaum ist eine Spezialisierung von einem Baum, während ein Baum auch eine Spezialisierung der noch allgemeineren Klasse Pflanze ist
- die Generalisierung findet in umgekehrter Richtung statt

Vererbung (Fort.)

- diese Beziehungen werden durch Bäume dargestellt, die von oben nach unten wachsen
- nach unten wird es immer spezieller
- Fahrrad erbt die Eigenschaften von Zweirad
- Zweirad erbt die Eigenschaften von Straßenfahrzeug
- Straßenfahrzeug erbt die Eigenschaften von Transportmittel



Ein *Fahrrad* ist ein *Zweirad*, ist ein *Straßenfahrzeug*, ist ein *Transportmittel*

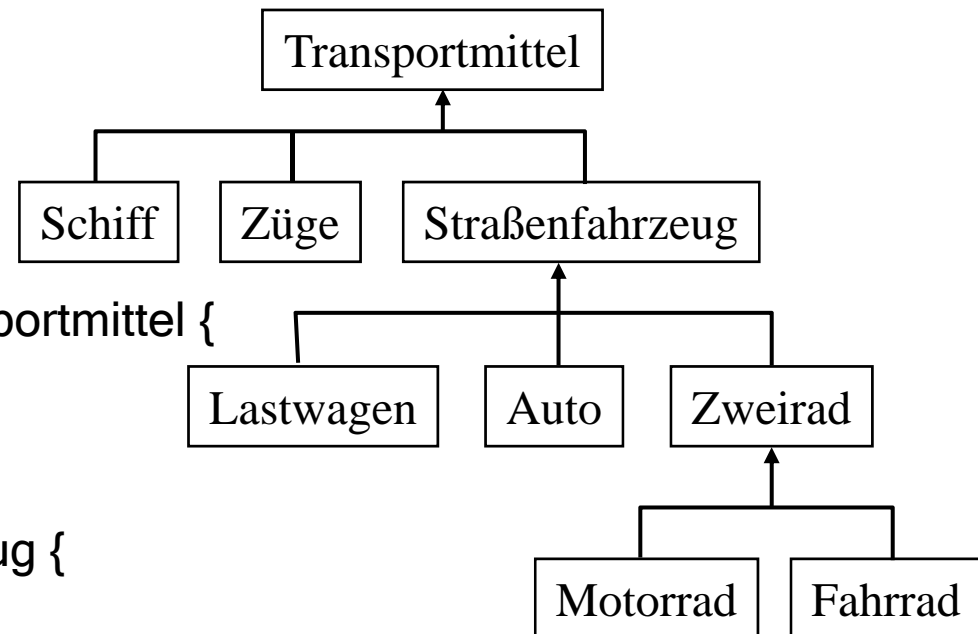
Vererbung (Fort.)

- werden die Kästen durch Klassen modelliert, so soll die Vererbung ebenfalls in Java ausgedrückt werden
- in Java kann eine Klasse immer von maximal einer Klasse erben

```
class Transportmittel {  
    ...  
}
```

```
class Strassenfahrzeug extends Transportmittel {  
    ...  
}
```

```
class Zweirad extends Strassenfahrzeug {  
    ...  
}
```



Go!

Vererbung (Fort.)

Begriffe: `class B extends A { ... }`

- die Klasse B wird von *A abgeleitet*
- A ist die *Basisklasse* von B
- B *erbt* alle Objektmethoden und Objektvariablen von A

```
class A {  
    public void doit() {System.out.println(m_j++);}  
    int m_j;  
}
```

```
class B extends A {  
}
```

B wird von A abgeleitet,
fügt A aber nichts hinzu

```
public class Derive1 {  
    public static void main(String[] args) {  
        B juhu = new B();  
        juhu.doit();  
        juhu.doit();  
    }  
}
```

von einem B Objekt wird 2-mal
die doit-Methode aufgerufen

Go!

Vererbung (Fort.)

```
class A {  
    public void doit() {  
        System.out.println(m_j++);  
    }  
    int m_j;  
}
```

```
class B extends A {  
    public void makeMyDay() {  
        System.out.println(m_j);  
    }  
}
```

```
public class Derive2 {  
    public static void main(String[] args) {  
        B juhu = new B();  
        juhu.makeMyDay();  
        juhu.doit();  
    }  
}
```

Welches m_j ist das hier?

B wird von A abgeleitet, und
fügt eine Methode hinzu

von einem B Objekt werden
beide Methoden aufgerufen

Vererbung (Fort.)

Was passiert bei der Konstruktion eines Objektes einer abgeleiteten Klasse?

1. es wird Platz für das Objekt auf dem Heap geschaffen
(der Platz reicht für die Objektvariablen der abgeleiteten Klasse ***und*** der Basisklasse)
2. es wird der Konstruktor der Basisklasse ausgeführt
3. es wird der Konstruktor der abgeleiteten Klasse ausgeführt

Go!

Vererbung (Fort.)

```
class A {  
    public A() {  
        System.out.println("A+");  
    }  
}
```

Basisklasse hat
einen Konstruktor

```
class B extends A {  
    public B() {  
        System.out.println("B+");  
    }  
}
```

abgeleitete Klasse
hat auch einen
Konstruktor

```
public class Derive3 {  
    public static void main(String[] args) {  
        new B();  
    }  
}
```

1 B Objekt wird erzeugt

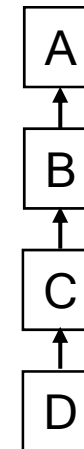
Was wird ausgegeben?

Go!

Vererbung (Fort.)

Die Abarbeitungsreihenfolge ändert sich auch nicht, wenn eine der Klassen keine Konstruktor hat!

```
class A {  
    public A() {System.out.println("A+");}  
}  
class B extends A {    B hat keine Konstruktor  
}  
class C extends B {  
    public C() {System.out.println("C+");}  
}  
class D extends C {    D hat keine Konstruktor  
}  
public class Derive4 {  
    public static void main(String[] args) {  
        new D();  
    }  
}
```



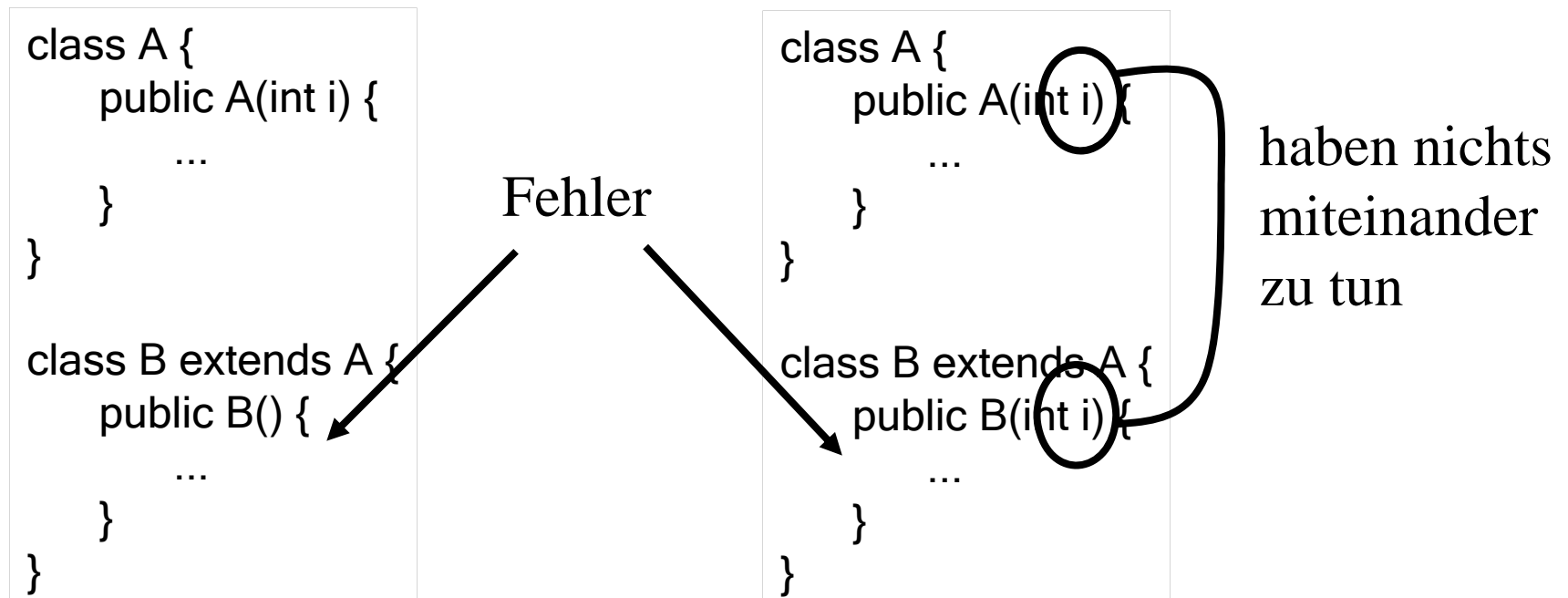
Vererbung (Fort.)

Frage:

Wie kann aber der Konstruktor einer Basisklasse ausgeführt werden, wenn er einen oder mehrere Parameter erwartet?

Antwort:

gar nicht! Der Parameter muss irgendwie angegeben werden



Vererbung (Fort.)

- um dem Konstruktor der Basisklasse Parameter zu übergeben, wird der Konstruktor explizit aufgerufen (wie eine Methode)
- dies erfolgt mittels des Schlüsselworts
`super`
- da immer der Konstruktor der Basisklasse vor dem eigene Konstruktor ausgeführt wird, muss das `super` der erste Befehl sein

Go!

Vererbung (Fort.)

```
class A {  
    public A(int i) {  
        System.out.println(i);  
        m_i = 2*i;  
    }  
    int m_i;  
}
```

Basisklasse A hat einen
Konstruktor mit Parametern

```
class B extends A {  
    public B(int j) {  
        super(j);  
        System.out.println(m_i);  
    }  
}
```

Konstruktor der abgeleiteten Klasse
ruft Basiskonstruktor direkt auf



```
public class Derive5 {
```

Was wird ausgegeben?

```
    public static void main(String[] args) {  
        new B(13);  
    }
```

1 B Objekt wird erzeugt

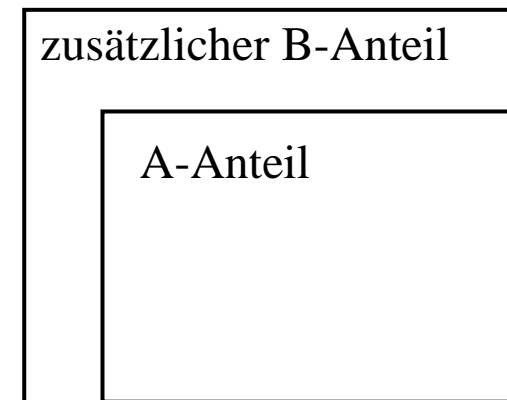
Vererbung (Fort.)

- Wie hängen die Typen von Basisklasse und abgeleiteter Klasse zusammen?

```
class B extends A { ... }
```

- Objekte der Klasse B "enthalten" auch als Teil ein Objekt der Klasse A
- *damit ist B auch vom Typ A*
- aber: ein Objekt der Klasse A ist *nicht auch* vom Typ B

B-Objekt



Go!

Vererbung (Fort.)

```
class A {  
    public A() {m_i = 13;}  
    public void doit() {System.out.println(++m_i);}  
  
    int m_i;  
}  
class B extends A {  
    public void makeMyDay() {System.out.println(m_i);}  
}  
public class Derive6 {  
    public static void main(String[] args) {  
        B b = new B();  
        A a1 = new B();  
        A a2 = b;  
        b.makeMyDay();  
        a1.doit();  
        a2.doit();  
    }  
}
```

Was wird ausgegeben?

2 B Objekt
werden erzeugt

kein Typkonflikt, kein Casting,
B ist auch vom Typ A

von a1, a2 kann nicht die
makeMyDay Methode
aufgerufen werden

Vererbung (Fort.)

- zurück zu den Türmen von Hanoi
- mittels Vererbung kann eine spezielle Klasse TowerWithStones von der Klasse Tower abgeleitet werden
- diese Klasse führt im Konstruktor die Initialisierung mit Steinen durch
- die Klasse Game erzeugt für den 1. Turm ein Objekt der Klasse TowerWithStones und für den 2. und 3. Turm Objekte der Klasse Tower

Go!

```
class Tower {  
    public Tower(int iDepth) {...}  
    public int removeStone() {...}  
    public void addStone(int iStone) {...}  
    public void print(int iWidth,int iPos) {...}  
}
```

Vererbung (Fort.)

Klasse Tower ohne
init-Methode

```
class TowerWithStones extends Tower {  
    public TowerWithStones(int iDepth) {  
        super(iDepth);  
        for(int i = 0;i < m_arStones.length;++i) {  
            m_arStones[i] = i*2+1;  
        }  
        m_iTop = -1;  
    }  
}
```

abgeleitete Klasse

Konstruktor der
Basisklasse aufrufen

Steine setzen (ehemalige init-Methode)

```
class Game {  
    public Game(int iDepth) {  
        m_iDepth = iDepth;  
        m_arTowers[0] = new TowerWithStones(m_iDepth);  
        m_arTowers[1] = new Tower(m_iDepth);  
        m_arTowers[2] = new Tower(m_iDepth);  
    } .....
```

nur 1 Turm hat Steine

Die Klasse Object

- jede Klasse erbt von einer anderen Klasse, entweder
 - explizit durch das Schlüsselwort **extends**, oder
 - implizit von der Klasse **Object**

```
class Object {  
    boolean equals(Object obj);  
    protected Object clone();  
    String toString();  
    int hashCode();  
}
```

wird von `println()`
aufgerufen

- dadurch sind alle Objekte vom Typ **Object**

Go!

Die Klasse Object (Fort.)

```
class A extends Object {  
}
```

Mit `extends Object` oder ...

```
class B {  
}
```

ohne, alle Klasse sind am
Ende von `Object` abgeleitet

```
public class AllObject {  
    public static void main(String[] args) {  
        Object o = new A();  
        System.out.println(o);  
        o = new B();  
        System.out.println(o);  
    }  
}
```

Sowohl A als auch B
sind beide auch vom
Typ `Object`

Go!

Die Klasse Object (Fort.)

- Vorteil, dass alle Klasse direkt oder indirekt von Object abgeleitet sind:
- alle Objekte können in Variablen vom Typ Object gespeichert werden

```
public class PrettyPrint {
```

```
    static void print(Object o) {  
        System.out.print(">>>>> ");  
        System.out.print(o);  
        System.out.println(" <<<<<");  
    }
```

jedes Objekt kann print
übergeben werden

```
    public static void main(String[] args) {  
        Object o = new A();  
        print(o);  
        print(new A());  
        print(new B());  
        print(new String("juhu"));  
    }  
}
```

Go!

Die Wrapperklassen

- Problem der vorherigen print Methode:
 - nur Objekte können übergeben werden
 - Werte oder Variablen der Basistypen *sollten nicht* verarbeitet werden können

```
public class PrettyPrint2 {  
    static void print(Object o) {  
        System.out.print(">>>>> ");  
        System.out.print(o);  
        System.out.println(" <<<<<");  
    }  
    public static void main(String[] args) {  
        print(3);  
        print(true);  
        print('j');  
    }  
}
```

funktioniert nicht mit
Java Versionen < 1.5

3 ist int, kein Object
true ist boolean, kein Object
'j' ist char, kein Object

Go!

Die Wrapperklassen (Forts.)

- für alle Basistypen gibt es entsprechende Wrapperklassen
- diese merken sich nur einen Wert des entsprechenden Basistyps
- für int gibt es die Klasse Integer
- für char gibt es die Klasse Character
- für boolean gibt es die Klasse Boolean
- ...

funktioniert jetzt auch
mit Java Versionen < 1.5

```
public class PrettyPrint3 {  
    ...  
    public static void main(String[] args) {  
        print(new Integer(3));  
        print(new Boolean(true));  
        print(new Character('j'));  
    }  
}
```

Wrapperklassen sind
auch vom Typ Object

Go!

Auto(un)boxing

- da die Verwendung der Wrapperklassen oft umständlich ist, gibt es **ab Version 1.5** das Autoboxing/Autounboxing
- Autoboxing: wird ein Objekt erwartet aber ein Wert eines Basistypes angegeben, wird automatisch die Wrapperklasse verwendet
- Autounboxing: wird Basistyp erwartet, aber es wird ein Objekt der Wrapperklasse verwendet, wird automatisch der Wert aus dem Wrapperobjekt gezogen

...

```
public static void main(String[] args) {  
    Integer i = new Integer(3);  
    int j = i; ← Autounboxing: Integer → int  
    print(j);  
} ← Autoboxing: int → Integer
```

Vorlesung 11/1

Klassenvariablen

- analog zu Objekt- und Klassenmethoden werden Variablen in *Objektvariablen* und *Klassenvariablen* unterschieden

- Objektvariablen gehören zu einem Objekt
- sie werden zusammen mit dem Objekt erschaffen
- sie existieren solange das Objekt existiert
- sie behalten ihren Wert über Aufrufe von Methoden

- Klassenvariablen gehören zu einer Klasse
- sie werden zum „Programmmanfang“ erschaffen
- sie existieren immer
- sie behalten ihren Wert über Aufrufe von Methoden
- für eine Klasse gibt es sie nur 1-mal, unabhängig von der Anzahl der Objekte

Klassenvariablen (Fort.)

- Klassenvariable werden durch das Schlüsselwort `static` gekennzeichnet
- analog zu Klassenmethoden stellt man es der Variablendeklaration voran

```
class A {  
    static int m_i;  
    boolean g_b;  
}
```

Dies ist eine Objektvariable

Dies ist eine Klassenvariable

Go!

Klassenvariablen: Beispiel

- auf Klassenvariablen kann direkt durch das Voranstellen des Klassennamens zugegriffen werden
- es muss kein Objekt existieren

```
class A {
```

```
    static int g_i = 13;
```

```
}
```

Klasse mit Klassenvariable



```
public class ClassVar {
```

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

```
        System.out.println(A.g_i);
```

```
    }
```

```
}
```

Was wird ausgegeben?

Referenziert die Klassen-
variable der Klasse A



Go!

Klassenvariablen: Beispiel

- Innerhalb einer Klasse kann auf die Klassenvariable direkt zugegriffen werden, d.h. ohne den Klassennamen voranzustellen
- Klassenmethoden werden außerhalb durch Voranstellen des Klassennamens aufgerufen

```
class A {  
    public static void doit() {  
        System.out.println(g_i);  
    }  
    static int g_i = 13;  
}  
public class ClassVar2 {  
    public static void main(String[] args) {  
        A.doit();  
    }  
}
```

Klasse mit Klassenmethode und ...

Klassenvariable wird direkt referenziert

... Klassenvariable

Referenziert die Klassenmethode der Klasse A

Was wird ausgegeben?

Go!

Klassenvariablen: Beispiel

- Klassenvariablen können auch aus Objektmethoden referenziert werden
- sie können auch durch ein Objekt referenziert werden

```
class A {  
    public void doit() {  
        System.out.println(g_i);  
    }  
    static int g_i = 13;  
}  
  
public class ClassVar3 {  
    public static void main(String[] args) {  
        A juhu = new A();  
        juhu.g_i = 42;  
        juhu.doit();  
    }  
}
```

Klasse mit Objektmethode und ...

Klassenvariable wird direkt referenziert

... Klassenvariable

Referenziert die Klassenvariable und Objektmethode der Klasse A

Was wird ausgegeben?

Go!

Klassenvariablen: Beispiel

- eine Klassenvariable lebt schon vor dem ersten Objekt der Klasse

```
class A {  
    public void doit() {  
        System.out.println(g_i);  
    }  
    static int g_i = 13;  
}
```

Objektmethode

← ... Klassenvariable

Was wird
ausgegeben?

```
public class ClassVar4 {  
    public static void main(String[] args) {  
        A.g_i = 42;  
        A juhu = new A();  
        juhu.doit();  
    }  
}
```

Referenziert die
Klassenvariable

Referenziert die
Objektmethode

Go!

Klassenvariablen: Beispiel

- eine Klassenvariable überlebt jedes Objekt

Was wird
ausgegeben?

```
class A {  
    static int g_i = 13; ← Klassenvariable  
}  
public class ClassVar5 {  
    public static void cool() {  
        A blabla = new A(); ← lokales Objekt ...  
        blabla.g_i = 34; ← ... verschwindet hier wieder  
    }  
    public static void main(String[] args) {  
        System.out.println(A.g_i);  
        cool();  
        System.out.println(A.g_i);  
    }  
}
```

Referenziert die
Klassenvariable

Go!

Klassenvariablen: Beispiel

- alle Objekte einer Klasse teilen sich die Klassenvariablen
- dies gilt auch für Vererbung

```
class A {  
    public A() {  
        System.out.println("ich bin das " + ++g_i + ". A");  
    }  
    static int g_i = 0;  
}  
class B extends A {  
}  
public class ClassVar6 {  
    public static void main(String[] args) {  
        A juhu = new A();  
        new B();  
        A blabla = new A();  
    }  
}
```

Was wird
ausgegeben?

im Konstruktor
wird die
Klassenvariable
verändert

2 A-Objekte und
dazwischen ein
B-Objekt

Klassenvariablen: Initialisierung

- Klassenvariablen werden genau wie Objektvariablen nur einmal initialisiert
- dies passiert, wenn die Klasse das 1. Mal geladen wird
- die Klassenvariablen werden gemäß ihrer Deklaration mit dem Standardwert initialisiert oder durch den Ausdruck, der bei der Deklaration angegeben ist

```
class A {  
    static int g_i = 34;  
    static int g_j = 3*g_i;  
}
```

Klassenvariablen g_i
und g_j werden mit 34
bzw. 102 initialisiert


Go!

Klassenvariablen: Initialisierung (Fort.)

- sollen Klassenvariablen komplexer initialisiert werden, so benötigt man etwas Ähnlich wie einen Konstruktor für Objekte
- gesucht ist ein Konstruktor für Klassen
- dies wird durch einen Block erreicht, der mit dem Schlüsselwort `static` beginnt und in der Klasse enthalten ist

```
class A {  
    static int g_i;  
    static int g_j;  
    static {  
        g_i = 34;  
        g_j = 3 * g_i;  
    }  
}  
  
public class ClassVar7 {  
    public static void main(String[] args) {  
        System.out.println(A.g_j);  
    }  
}
```

Klassenvariablen `g_i` und `g_j` werden mit 34 bzw. 102 im Klassenkonstruktor initialisiert



Was wird
ausgegeben?


Go!

Klassenvariablen: Initialisierung (Fort.)

- ein Klassenkonstruktor wird einmal ausgeführt
- dies passiert *unmittelbar* vor der 1. Benutzung der Klasse

```
class A {  
    static int g_i;  
    static int g_j;  
    static {  
        g_i = 34;  
        g_j = 3 * g_i;  
        System.out.println("Klassenkonstruktor A");  
    }  
}  
  
public class ClassVar8 {  
    public static void main(String[] args) {  
        System.out.println("juhu");  
        System.out.println(A.g_j);  
        System.out.println(A.g_i);  
    }  
}
```

Klassenkonstruktor wird ausgeführt, bevor die Klasse verwendet wird



Was wird ausgegeben?

Go!

Klassenvariablen: Initialisierung (Fort.)

- Benutzung einer Klasse kann auch das Erzeugen eines Objekts sein

```
class A {  
    static int g_i;  
    static int g_j;  
    static {  
        g_i = 34;  
        g_j = 3 * g_i;  
        System.out.println("Klassenkonstruktor A");  
    }  
}  
  
public class ClassVar9 {  
    public static void main(String[] args) {  
        new A();  
        System.out.println("juhu");  
        System.out.println(A.g_j);  
        System.out.println(A.g_i);  
    }  
}
```

Was wird
ausgegeben?

hier wird die Klasse
das 1. Mal verwendet

Go!

Klassenvariablen: Initialisierung (Fort.)

- wird die Klasse nicht benutzt, wird der Klassenkonstruktor auch nicht ausgeführt

```
class A {  
    static int g_i;  
    static int g_j;  
    static {  
        g_i = 34;  
        g_j = 3 * g_i;  
        System.out.println("Klassenkonstruktor A");  
    }  
}  
  
public class ClassVar10 {  
    public static void main(String[] args) {  
        System.out.println("juhu");  
    }  
}
```

Was wird
ausgegeben?

die Klasse A wird
nirgends verwendet

Go!

Klassenvariablen: Initialisierung (Fort.)

- abgeleitete Klassen verhalten sich bzgl. ihrer Objekt- und Klassenkonstruktoren analog
- erst wird der Klassenkonstruktor der Basisklasse, dann der abgeleiteten Klasse ausgeführt

```
class A {  
    static {System.out.println("Klassenkonstruktor A");}  
}  
class B extends A {  
    static {System.out.println("Klassenkonstruktor B");}  
}  
public class ClassVar11 {  
    public static void main(String[] args) {  
        System.out.println("juhu");  
        new B();  
    }  
}
```

Was wird
ausgegeben?

verwendet die Klasse
B das 1. Mal



Go!

Klassenvariablen: Initialisierung (Fort.)

- auch hier gilt: nur die Klassenkonstruktoren der verwendeten Klassen werden ausgeführt

```
class A {  
    static {System.out.println("Klassenkonstruktor A");}  
}  
class B extends A {  
    static {System.out.println("Klassenkonstruktor B");}  
}  
public class ClassVar12 {  
    public static void main(String[] args) {  
        System.out.println("juhu");  
        new A();  
        System.out.println("blabla");  
        new B();  
    }  
}
```

Was wird
ausgegeben?

verwendet die Klasse
A das 1. Mal

verwendet die Klasse
B das 1. Mal

Go!

Klassenvariablen: Initialisierung (Fort.)

- auch hier gilt: die Klassenkonstruktoren werden nur einmal ausgeführt

```
class A {  
    static {System.out.println("Klassenkonstruktor A");}  
}  
class B extends A {  
    static {System.out.println("Klassenkonstruktor B");}  
}  
public class ClassVar13 {  
    public static void main(String[] args) {  
        System.out.println("juhu");  
        new B();  
        System.out.println("blabla");  
        new A();  
    }  
}
```

Was wird
ausgegeben?

diesmal wird erst ein
B-Objekt und dann
ein A-Objekt erzeugt

Klassen- und Objektvariablen und –methoden: Zusammenfassung

<ul style="list-style-type: none">• Klassenmethoden & -variablen kennzeichnet man durch static	<ul style="list-style-type: none">• Objektmethoden & -variablen werden ohne static spezifiziert
<ul style="list-style-type: none">• sie gehören zur Klasse	<ul style="list-style-type: none">• sie gehören zum Objekt
<ul style="list-style-type: none">• sie können ohne Objekt verwendet werden	<ul style="list-style-type: none">• sie können nur mit einem Objekt verwendet werden
<ul style="list-style-type: none">• es gibt sie nur einmal, unabhängig von Anzahl der Objekte	<ul style="list-style-type: none">• jedes Objekt hat seine eigenen Objektmethoden & -variablen
<ul style="list-style-type: none">• sie leben immer	<ul style="list-style-type: none">• sie leben solange das zugehörige Objekt lebt
<ul style="list-style-type: none">• Initialisierung durch Klassenkonstruktoren	<ul style="list-style-type: none">• Initialisierung durch Objektkonstruktoren

Vorlesung 11/2

Modifier

```
class A {  
    public A() {  
        m_i = 2;  
    }  
    public int div(int i) {  
        return i / m_i;  
    }  
    int m_i;  
}
```

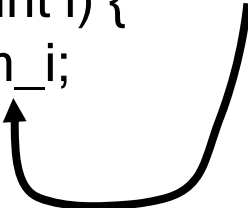
Was ist das Problem
an dem Programm?

```
public class B {  
    public static void main(String[] args) {  
        A juhu = new A();  
        juhu.m_i = 0;  
        System.out.println(juhu.div(4));  
    }  
}
```


Modifier (Fort.)

```
class A {  
    public A() {  
        m_i = 2;  
    }  
    public int div(int i) {  
        return i / m_i;  
    }  
    int m_i;  
}
```

für die Klasse ist es wichtig,
dass `m_i` niemals 0 wird



```
public class B {  
    public static void main(String[] args) {  
        A juhu = new A();  
        juhu.m_i = 0;  
        System.out.println(juhu.div(4));  
    }  
}
```



hier wird der Zustand des
Objekts juhu direkt
verändert; das Objekt ist
jetzt "kaputt", es hat einen
illegalen Zustand

Modifier (Fort.)

- Objektvariablen stellen den Zustand eines Objekts dar
- oft stellen nur bestimmte Variablenbelegungen einen gültigen Objektzustand dar
- werden Variablen von außerhalb des Objekts verändert, kann es schnell passieren, dass das Objekt einen illegalen Zustand annimmt
- um sicher von außerhalb den Objektzustand zu verändern, muss man die Klasse sehr gut kennen
- dies widerspricht der dem Konzept der Abstraktion
- man möchte den Inhalt einer Klasse/eines Objekts nicht kennen
- ...

Modifier (Fort.)

- ...
- daher sollte es verboten sein, die Variablen außerhalb eines Objekts verändern zu können
- manchmal möchte man aber doch die Variablen direkt lesen können (z.B. die `.length` Variable von Arrays)
- daher kann der Programmier spezifizieren, welche Variable außerhalb der Klasse sichtbar ist

...

```
A juhu = new A();  
juhu.m_i = 0;
```

...

Ziel: hier soll es einen
Compilerfehler geben!!

Modifier (Fort.)

- um diese Art der Zugriffsrechte zu regeln, gibt es sogenannte *Modifier*
- diese stehen vor jeder Objekt- und Klassenmethode und -variable und vor jeder Klasse
- sie regeln, ob auf Elemente zugegriffen werden darf, und wenn ja, ob es *nur lesend oder auch schreibend* ist

Modifier (Fort.)

die Zugriffsrechte werden geregelt durch die 4 Modifier

1. **public**: alle anderen dürfen auf dieses Element zugreifen
2. **private**: nur die eigene Klasse darf auf die Elemente zugreifen; die abgeleiteten Klassen und die Erzeuger können nicht darauf zugreifen
3. **protected**: die eigene und die abgeleiteten Klassen können auf die Elemente zugreifen; die Erzeuger können nicht darauf zugreifen; alle im gleichen Paket können darauf zugreifen
4. ***ohne Modifier*** (Standard): alle im gleichen Paket können darauf zugreifen; außerhalb des Pakets sind sie nicht zugreifbar; sie liegen somit zwischen **public** und **protected**

Modifier: Beispiel

```
class A {  
    A() {  
        m_i = 13;  
    }  
    protected void doit() {  
        System.out.println(m_i);  
    }  
    private int m_i;  
}  
class B extends A {  
    void makeMyDay() {  
        doit();  
        m_i = 42;  
    }  
}  
public class Modifier {  
    public static void main(String[] args) {  
        B juhu = new B();  
        juhu.makeMyDay();  
        juhu.m_i = 32;  
        juhu.doit();  
    }  
}
```

Konstruktor; sichtbar
in diesem Paket

Objektvariable; nur
sichtbar in dieser Klasse

die ererbte protected Methode
kann aufgerufen werden

die ererbte private
Elemente sind gesperrt

die anderen private
Elemente sind gesperrt

Go!

Modifier (Fort.)

- Auf Elemente, die als `private` deklariert sind, kann nur von der eigenen Klasse und *von anderen Objekten der gleichen Klasse* zugegriffen werden.

```
class A {  
    void doit(A juhu) {  
        juhu.m_i = 42;  
        System.out.println(m_i);  
    }  
    private int m_i = 13;  
}  
  
public class Modifier2 {  
    public static void main(String[] args) {  
        A a1 = new A();  
        A a2 = new A();  
        a1.doit(a2);  
        a2.doit(a1);  
    }  
}
```

Zugriff auf die private Variable von juhu

Was wird ausgegeben?

Modifier: final

- ein anderer wichtiger Modifier ist das final
- es kann vor Klassen, Methoden und Variablen geschrieben werden
- es besagt, dass das nachfolgende Element in gewissem Sinne konstant ist
- konstante Elemente können nur einmal einen Wert bekommen und ihn später nicht mehr ändern

Go!

Modifier: final (Fort.)

```
class A {  
    void doit() {  
        // m_i = 42;  
        System.out.println(m_i);  
    }  
  
    private final int m_i = 13;  
}
```

m_i kann hier nicht verändert werden

die Objektvariable m_i ist privat und konstant

```
public class Final1 {  
  
    public static void main(String[] args) {  
        A a1 = new A();  
        a1.doit();  
    }  
}
```

Go!

Modifizier: final (Fort.)

- final Variablen müssen nicht direkt in der Deklaration initialisiert werden

```
class A {  
    A(int i) {  
        m_i = i;  
    }  
    void doit() {  
        // m_i = 42;  
        System.out.println(m_i);  
    }  
    private final int m_i;  
}  
public class Final2 {  
    public static void main(String[] args) {  
        A a1 = new A(34);  
        a1.doit();  
    }  
}
```

m_i wird das 1. und einzige Mal beschrieben

noch kein Wert zuweisen

Go!

Modifier: final (Fort.)

- final kann auch für Parameter verwendet werden
- in diesem Fall kann dem Parameter in der Methode kein Wert zugewiesen werden

```
public class Final3 {  
    public static void doit(final int i) {  
        //i = 0;  
        System.out.println(i);  
    }  
  
    public static void main(String[] args) {  
        doit(78);  
    }  
}
```

i bekommt einen Wert
beim Aufruf zugewiesen

dieser kann nicht
verändert werden

Go!

Modifier: final (Fort.)

- final Variablen, die Objekte speichern, kann *kein neues Objekt zugewiesen* werden
- diese *Objekte dürfen* sich aber *ändern*

```
class A {  
    void doit() {  
        System.out.println(++m_i);  
    }  
    private int m_i = 13;  
}  
  
public class Final4 {  
    public static void main(String[] args) {  
        final A juhu = new A();  
        juhu.doit();  
        juhu.doit();  
        //juhu = new A();  
    }  
}
```

doit ändert das Objekt

juhu ist konstant

das Objekt von juhu kann sich aber ändern

juhu kann aber kein neues Objekt zugewiesen werden

Go!

Modifier: final (Fort.)

- analoges gilt für Arrays

```
public class Final5 {  
    public static void main(String[] args) {  
        final int[] juhu = new int[6];  
        juhu[3] = 23;  
        juhu[3] = 42;  
        //juhu = new int[4];  
    }  
}
```

juhu ist ein konstantes
int-Array

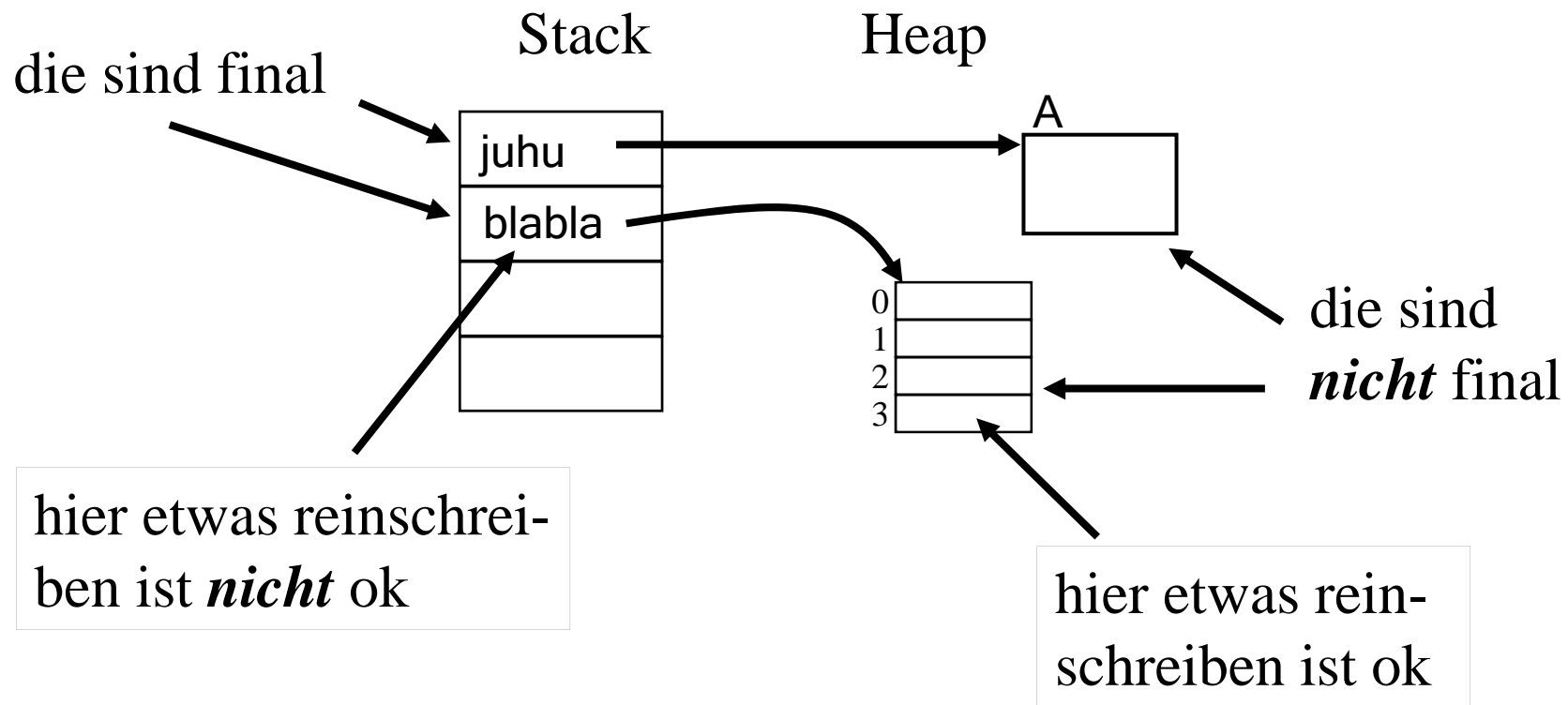
die einzelnen Elemente
sind nicht konstant

juhu kann aber kein
neues int-Array
zugewiesen werden

Modifizier: final (Fort.)

- Wie kann das final Verhalten von Arrays und Objekten erklärt werden?

```
final A juhu = new A();  
final int[] blabla = new int[4];
```



Modifier: final (Fort.)

- wir eine Klasse als final deklariert, so kann von ihr nicht weiter abgeleitet werden

```
class A {  
    ...  
}  
  
final class B extends A {  
    ....  
}  
  
class C extends B {  
    ...  
}
```

- Klassen als konstant zu deklarieren erlaubt es dem Java-Compiler, effizienteren Code zu erzeugen

Fehler: von B kann nicht weiter abgeleitet werden



Destruktoren

- wenn ein Objekt einer Klasse *erzeugt wird*, wird sein *Konstruktor* aufgerufen (wenn er existiert)
- *im Konstruktor* können wichtige *Datenstrukturen initialisiert* werden
- manchmal ist es sinnvoll, beim *Sterben eines Objekts* noch *Code auszuführen*
- dies ist für Sprachen wie C++ viel wichtiger als für Java
- hierzu kann man zu einer Klasse einen *sogenannten Destruktor* definieren

Go!

Destruktoren (Fort.)

- ein Destruktor hat immer die Form

`protected void finalize() {...}`

- da das Sterben eines Objekts nicht vom Benutzer gestartet wird, kann dem Destruktor keine Parameter übergeben werden

Was wird
ausgegeben?

```
class A {                                     Klasse mit Konstruktor ...
    A() {
        System.out.println("A+");
    }
    protected void finalize() {
        System.out.println("A-");
    }
}                                             ... und Destruktor

public class Ende1 {
    public static void main(String[] args) {
        new A();
    }
}
```

neues Objekt wird erzeugt und ?

Destruktoren (Fort.)

- Java verfügt über ein automatisches Speichermanagement
- Objekte sterben nicht sofort wenn ihre Sichtbarkeit verloren geht

```
class A { ... }  
public class Ende1 {  
    static void doit() {  
        A juhu = new A();  
    }  
    public static void main(String[] args) {  
        doit();  
    }  
}
```

hier entsteht ein neues A

hier sieht man es nicht mehr

- vielmehr wird regelmäßig vom Java-Interpreter ein Aufräumen gestartet
- es ist unsicher, ob es überhaupt gestartet wird, und wenn, wann und wie oft

Go!

Destruktoren (Fort.)

Was wird
ausgegeben?

```
class A {  
    public A() {  
        System.out.println("ich bin das " + ++g_i + ". A");  
    }  
    protected void finalize() {  
        System.out.println("Ich sterbe. Jetzt gibt es nur noch " + --g_i + " meiner Art.");  
    }  
    static int g_i = 0;  
}
```

in Klassenvariable wird die
Anzahl der Objekte gezählt

```
public class Ende2 {  
    public static void main(String[] args) {  
        for(int i = 0; i < 1200; ++i)  
            new A();  
    }  
}
```

1200 A-Objekte werden
erzeugt, deren Sichtbarkeit
gleich wieder erlöscht

Go!

Destruktoren (Fort.)

Was wird
ausgegeben?

- funktioniert auch bei abgeleiteten Klassen

```
class A {  
    public A() {  
        System.out.print("ich bin das " + ++g_i + ". A");  
    }  
    protected void finalize() {  
        System.out.println("Ich sterbe. Jetzt gibt es nur noch " + --g_i + " meiner Art.");  
    }  
    static int g_i = 0;  
}
```

```
class B extends A {  
    B() {System.out.println(": ein neues B");}  
}
```

B hat einen eigenen
Konstruktor

```
public class Ende5 {  
    public static void main(String[] args) {  
        for(int i = 0; i < 1200; ++i)  
            new B();  
    }  
}
```

1200 B-Objekte werden
erzeugt, deren Sichtbarkeit
gleich wieder erlöscht

Go!

Destruktoren (Fort.)

```
class A {  
    public A() {System.out.print("ich bin das " + ++g_i + ". A");}  
    protected void finalize() {  
        System.out.println("Ich sterbe. Jetzt gibt es nur noch " + --g_i + " meiner Art.");  
    }  
    static int g_i = 0;  
}  
class B extends A {  
    B() {  
        System.out.println(": ein neues B ");  
    }  
    protected void finalize() {  
        System.out.println("ein B weniger");  
    }  
}  
public class Ende3 {  
    public static void main(String[] args) {  
        for(int i = 0; i < 1200; ++i)  
            new B();  
    }  
}
```

A hat einen Destruktor

Was wird ausgegeben?

B hat ebenfalls einen Destruktor

1200 B-Objekte werden erzeugt, deren Sichtbarkeit gleich wieder erlöscht

Go!

Destruktoren (Fort.)

- im *Gegensatz zu den Konstruktoren* sind die *Destruktoren nicht verkettet*
- soll der *Destruktor der Basisklasse* ausgeführt werden, so muss er *explizit* mittels `super.finalize()` *aufgerufen* werden

```
class A { ... }  
class B extends A {  
    B() { ... }  
    protected void finalize() {  
        System.out.print("ein B weniger: ");  
        super.finalize();  
    }  
}  
public class Ende3 {  
    public static void main(String[] args) {  
        ...  
    }  
}
```

Was wird
ausgegeben?

explizit Destruktor
von A aufrufen

Überladen und Überlagern von Methoden

- Methoden können *überladen* und *überlagert* werden
- klingt ähnlich, meint aber zwei ganz unterschiedliche Konzepte der objektorientierten Programmierung

Überladen:	Überlagern:
<ul style="list-style-type: none">• 2 <i>unterschiedliche Methoden</i> in <i>einer Klasse</i>	<ul style="list-style-type: none">• 2 <i>unterschiedliche Methoden</i> in 2 <i>voneinander abgeleiteten Klassen</i>
<ul style="list-style-type: none">• beide haben den <i>gleichen Namen</i>	
<ul style="list-style-type: none">• sie <i>unterscheiden</i> sich in ihren <i>Parametern</i>	<ul style="list-style-type: none">• sie haben die <i>gleichen Parametern</i>

Überladen von Methoden

- Methoden werden überladen, wenn verschiedene Methoden mit unterschiedlichen Parametern eine ähnliche Funktionalität ausführen
- Beispiel:
 - eine Klasse, die einen mathematischen Vektor implementiert
 - diese braucht Multiplikationsmethoden
 - eine nimmt einen Vektor und multipliziert ihn mit dem aktuellen Vektor (Vektormultiplikation)
 - eine andere nimmt eine reelle Zahl und multipliziert diese mit dem aktuellen Vektor (Skalarmultiplikation)
 - beide Methoden sollten `mult` heißen

Überladen von Methoden (Fort.)

```
class Vector {
```

```
    public Vector mult(float f) {  
        // berechne eine Skalarmultiplikation  
        // mit dem aktuellen Objekt  
    }  
}
```

```
    public Vector mult(Vector v) {  
        // berechne eine Vektormultiplikation  
        // mit dem aktuellen Objekt  
    }  
}
```

2 verschiedene
Methoden mit gleichem
Namen in einer Klasse

sie unterscheiden
sich in den Typen
ihrer Parametern

Überladen von Methoden (Fort.)

- welche der Methoden ausgeführt werden, wird beim Aufruf anhand der Typen herausgefunden

```
class Vector {
```

```
    public Vector mult(float f) { ... }
```

```
    public Vector mult(Vector v) { ... }
```

```
}
```

```
...
```

```
Vector v1,v2,v3,v4;
```

```
...
```

```
v2 = v1.mult(v4);
```

```
v3 = v2.mult(3.5f);
```

ruft diese Methode
auf, weil ein Vector
übergeben wird

ruft diese Methode
auf, weil ein float
übergeben wird

Go!

Überladen von Methoden(Fort.)

- zur Compilezeit wird anhand des Typs festgestellt, welche der überladenen Methoden aufgerufen wird

```
class A {  
    void doit(int i) {  
        System.out.println("ich bekomme ein int");  
    }  
    void doit(double b) {  
        System.out.println("ich bekomme ein double");  
    }  
}
```

doit ist überladen

```
public class OverLoad {  
    public static void main(String[] args) {  
        A juhu = new A();  
        juhu.doit(23);  
        juhu.doit(23.0);  
    }  
}
```

Was wird
ausgegeben?

hier ist klar, welches
doit jeweils aufgerufen
werden muss

Go!

Überladen von Methoden(Fort.)

- Überladen funktioniert auch für Klassenmethoden
- kann anhand des Typs nicht eindeutig die Methode identifiziert werden, so gibt es einen Compilerfehler

```
public class OverLoad2 {  
    static void doit(short s) {  
        System.out.println("ich bekomme ein short");  
    }  
    static void doit(byte b) {  
        System.out.println("ich bekomme ein byte");  
    }  
    public static void main(String[] args) {  
        byte b = 23;  
        doit(b);  
        //doit(23);  
    }  
}
```

Was wird
ausgegeben?

hier ist klar, welches doit gemeint ist

hier ist es nicht klar, da 23 ein int
ist, also weder byte noch short

Go!

Überladen von Methoden(Fort.)

- überladene Methoden können sich auch in ihren Rückgabewert unterscheiden

```
public class OverLoad3 {  
    static int doit(short s) {  
        System.out.println("ich bekomme ein short");  
        return 42;  
    }  
    static void doit(byte b) {  
        System.out.println("ich bekomme ein byte");  
    }  
    public static void main(String[] args) {  
        byte b = 23;  
        short s = 34;  
        doit(b);  
        System.out.println(doit(s));  
    }  
}
```

Was wird
ausgegeben?

dieses doit liefert nichts zurück

dieses doit liefert einen int Wert zurück


Go!

Überladen von Methoden(Fort.)

- es reicht *nicht*, dass sich überladene Methoden *nur im Rückgabetyt unterscheiden*

```
public class OverLoad4 {  
    static int doit(int i) {  
        return 42;  
    }  
    static boolean doit(int i) {  
        return false;  
    }  
    public static void main(String[] args) {  
        int i = doit(23);  
        boolean b = doit(13);  
    }  
}
```

hier ist in beiden
Fällen unklar, welches
doit gemeint ist



Go!

Überladen von Methoden(Fort.)

- für überladene Methoden reicht auch eine unterschiedliche Anzahl von Parametern aus, um sich zu unterscheiden

```
public class OverLoad5 {  
    static int doit(int i,int j) {  
        return 42;  
    }  
    static boolean doit(int i) {  
        return false;  
    }  
    public static void main(String[] args) {  
        int i = doit(23,24);  
        boolean b = doit(13);  
        System.out.println(i + "\t" + b);  
    }  
}
```

beide doit unterscheiden
sich durch die Anzahl
der Parameter

hier ist klar, welches
doit gemeint ist

Was wird
ausgegeben?

Go!

Überladen von Methoden(Fort.)

- durch das Überladen von Methoden kann man den Parametern Standardwerte übergeben

```
public class OverLoad6 {  
    static void doit(int i,int j) {  
        System.out.println("i: " + i + "\tj: " + j);  
    }  
    static void doit(int i) {  
        doit(i,23);  
    }  
    static void doit() {  
        doit(42);  
    }  
    public static void main(String[] args) {  
        doit();  
        doit(3);  
        doit(56,107);  
    }  
}
```

die Methode doit mit 2,
1 oder 0 int-Werte an

Standardwert
für 2. Parameter

Standardwert
für 1. Parameter

Was wird
ausgegeben?

Vorlesung 12/1

Überladen und Überlagern von Methoden (Fort.)

Überladen:	Überlagern:
<ul style="list-style-type: none">• 2 <i>unterschiedliche Methoden</i> in <i>einer Klasse</i>	<ul style="list-style-type: none">• 2 <i>unterschiedliche Methoden</i> in 2 <i>voneinander abgeleiteten Klassen</i>
<ul style="list-style-type: none">• beide haben den <i>gleichen Namen</i>	
<ul style="list-style-type: none">• sie <i>unterscheiden</i> sich in ihren <i>Parametern</i>	<ul style="list-style-type: none">• sie haben die <i>gleichen Parametern</i>

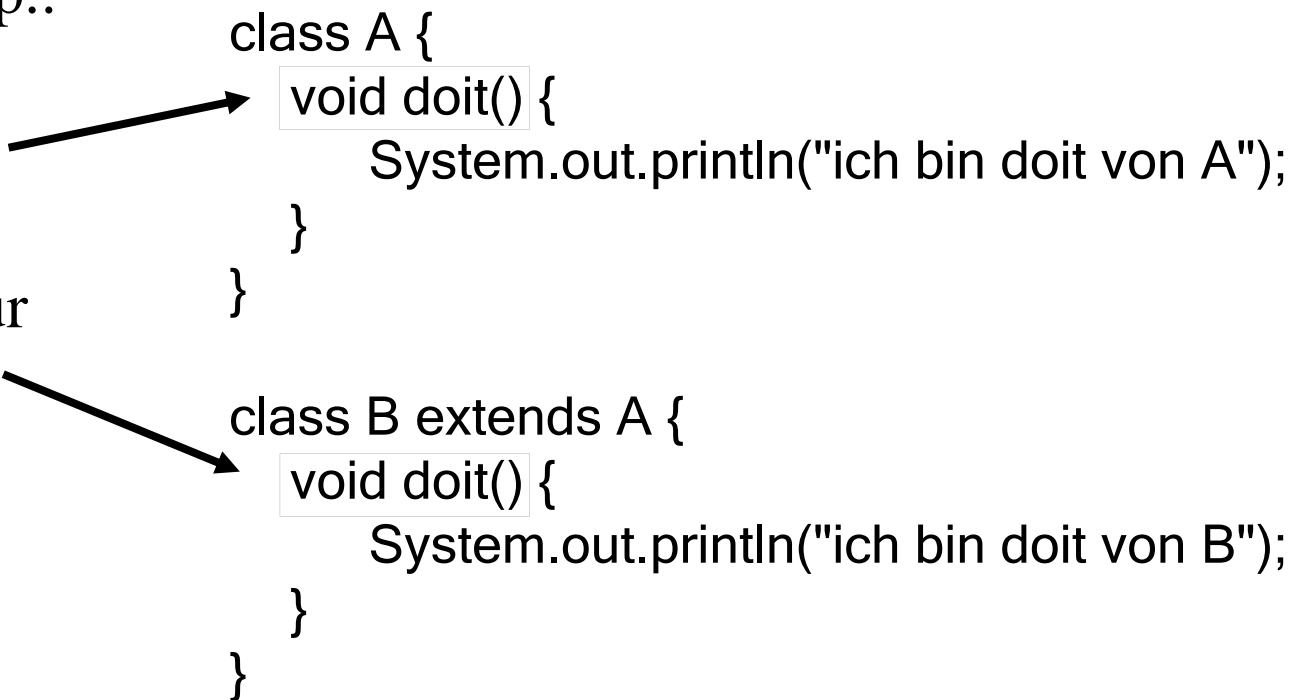
zusammen mit Vererbung und Konstruktoren ist die Überlagerung eines der wichtigsten Konzepte in der OOP

Überlagern von Methoden

- Methoden werden überlagert, indem eine Methode mit gleichem Namen und gleichen Parametern in einer abgeleiteten Klasse nochmals implementiert wird
- Frage: welche der beiden Methoden wird ausgeführt?
- Bsp.:

gleiche
Signatur

```
class A {  
    void doit() {  
        System.out.println("ich bin doit von A");  
    }  
}  
  
class B extends A {  
    void doit() {  
        System.out.println("ich bin doit von B");  
    }  
}
```

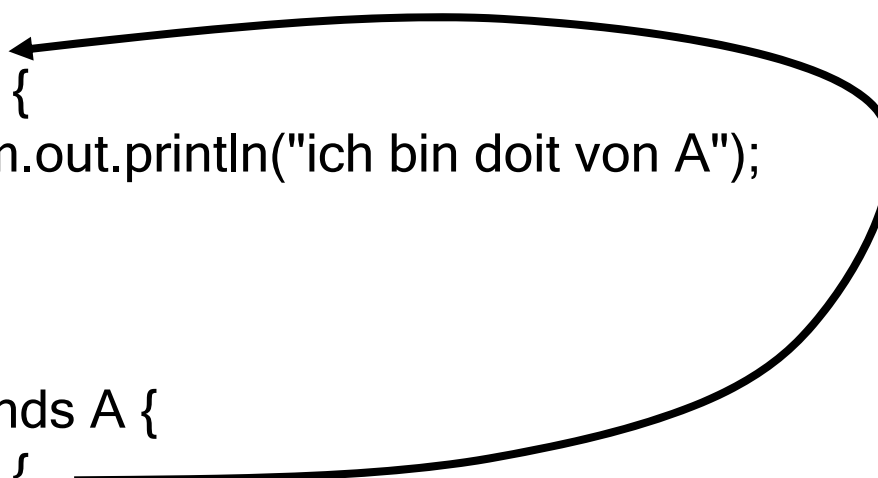


Überlagern von Methoden (Fort.)

- durch das Ableiten erbt die Klasse B von A die Methode doit
- durch die Neudefinition von doit in B wird die ererbte Methode überlagert
- nun hat die *neue Definition Vorrang* und *wird ausgeführt*

neue Methode hat
Vorrang vor
ererbten Methode

```
class A {  
    void doit() {  
        System.out.println("ich bin doit von A");  
    }  
}  
  
class B extends A {  
    → void doit() {  
        System.out.println("ich bin doit von B");  
    }  
}
```



Go!

Überlagern von Methoden (Fort.)

- die neue Definition wird natürlich nur ausgeführt, wenn es sie gibt, d.h. ein Objekt der Basisklasse hat diese Neudefinition nicht

```
class A {  
    void doit() {  
        System.out.println("ich bin doit von A");  
    }  
}  
class B extends A {  
    void doit() {  
        System.out.println("ich bin doit von B");  
    }  
}  
public class Virtual1 {  
    public static void main(String[] args) {  
        A x = new A();  
        B y = new B();  
        x.doit();  
        y.doit();  
    }  
}
```

überlagert doit

kennt das neue
doit *nicht*

Was wird
ausgegeben?

Go!

Überlagern von Methoden (Fort.)

- die Entscheidung, welche Methode auszuführen ist, kann erst zur Laufzeit stattfinden, weil ...
- in einer Variablen vom Typ A auch ein B Objekt abgespeichert sein kann

speichert ein
B in einer A-
Variable

```
class A {  
    void doit() {  
        System.out.println("ich bin doit von A");  
    }  
}  
class B extends A {  
    void doit() {  
        System.out.println("ich bin doit von B");  
    }  
}  
public class Virtual2 {  
    public static void main(String[] args) {  
        A x = new A();  
        A y = new B();  
        x.doit();  
        y.doit();  
    }  
}
```

Was wird
ausgegeben?

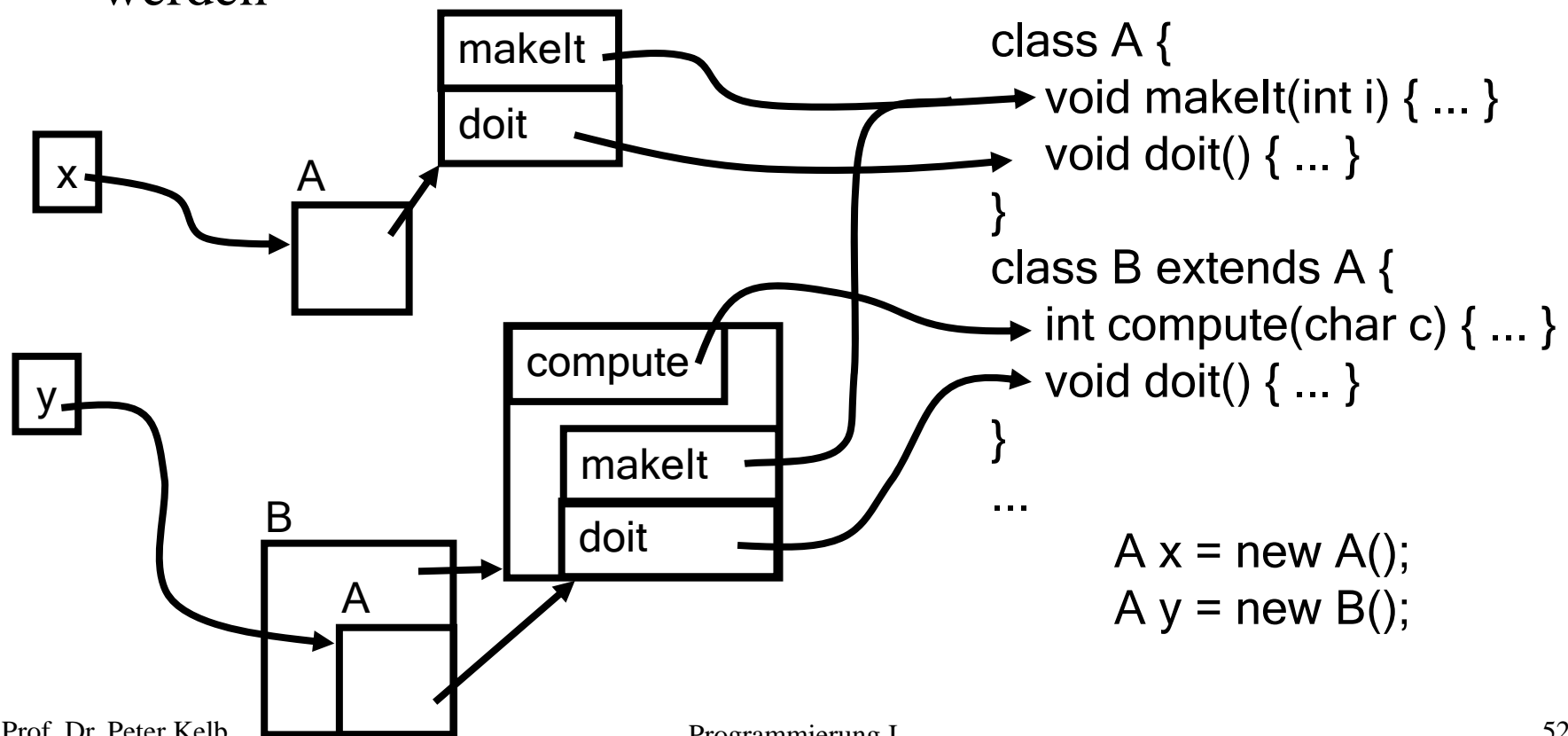
hier ist immer
noch ein B drin

Überlagern von Methoden (Fort.)

- jedes Objekt merkt sich in einer kleinen Tabelle zu einem Methodennamen, welche Methodenkörper zum Namen gehört
- beim Überlagern wird ein solcher Eintrag überlagert
- beim Aufruf einer Methode wird in dieser Tabelle nachgeschaut, welche Methode auszuführen ist

Überlagern von Methoden (Fort.)

- y merkt sich nur den A-Anteil des B-Objekts
- der A-Anteil merkt sich nur den Teil der Tabelle, der für die A-Klasse relevant ist
- durch die Überlagerung wird dennoch doit von B aufgerufen werden



Go!

Beispiel

```
class A {  
    void doit() {  
        System.out.println("ich bin doit von A");  
    }  
}
```

```
class B extends A {
```

```
    B(int i) {  
        m_i = i;
```

```
    }
```

```
    void doit() {
```

```
        System.out.println("ich bekam eine " + m_i);
```

```
    }
```

```
    int m_i;
```

```
}
```

```
public class Virtual3 {
```

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

```
        A[] juhu = new A[10];
```

```
        for(int i = 0; i < juhu.length; ++i)
```

```
            juhu[i] = Math.random() < 0.5 ? new A() : new B(i);
```

```
        for(int i = 0; i < juhu.length; ++i)
```

```
            juhu[i].doit();
```

```
    }
```

```
}
```

B merkt sich im
Gegensatz zu A
einen int-Wert

Was wird
ausgegeben?

juhu merkt sich
nur A-Objekte

zur Laufzeit wird entschieden,
welches doit gemeint ist

Überlagern von Methoden (Fort.)

- Überlagern von Methoden wird dann verwendet, wenn abgeleitete Klassen ihre eigene Interpretation einer Methode haben
- Bsp.: das Zeichnen geometrischer Figuren
 - die Figuren Dreieck, Rechteck und Quadrat sind alles geometrische Figuren
 - alle sollten sich zeichnen lassen
 - jede einzelne Klasse wird aber unterschiedlich gezeichnet

Überlagern von Methoden (Fort.)

```
class Figur {  
    void draw(char c) {}  
}
```

die Basisklasse, die nur
eine leere draw-Methode
enthält

...

```
public class Virtual4 {  
    public static void main(String[] args) {  
        Figur[] x = new Figur[10];  
        for(int i = 0; i < x.length; ++i) {  
            x[i] = ???  
        }  
        for(int i = 0; i < x.length; ++i) {  
            x[i].draw('#');  
            System.out.println();  
        }  
    }  
}
```

die Anwendung: 10
verschiedene Figuren
sollen erzeugt werden ...

... und anschließend
gezeichnet werden

Überlagern von Methoden (Fort.)

```
...  
class Dreieck extends Figur {  
    Dreieck(int iHoehe) {  
        m_Hoehe = iHoehe;  
    }  
}
```

ein Dreieck ist eine Figur ...

... und merkt sich die Höhe ...

```
void draw(char c) {  
    for(int i = 0; i < m_Hoehe; ++i) {  
        for(int j = 0; j <= i; ++j) {  
            System.out.print(c);  
        }  
        System.out.println();  
    }  
}
```

... und hat seine eigene
Implementierung für
das Zeichnen

```
private int m_Hoehe;  
}  
...
```

Überlagern von Methoden (Fort.)

```
...  
class Rechteck extends Figur {  
    Rechteck(int iBreite,int iHoehe) {  
        m_Breite = iBreite;  
        m_Hoehe = iHoehe;  
    }  
}
```

ein Rechteck ist eben-
falls eine Figur ...

... und merkt sich die
Höhe und die Breite ...

```
void draw(char c) {  
    for(int i = 0;i < m_Hoehe;++i) {  
        for(int j = 0;j < m_Breite;++j) {  
            System.out.print(c);  
        }  
        System.out.println();  
    }  
}
```

... und hat auch seine
eigene Implemen-
tierung für das Zeichnen

```
private int m_Hoehe;  
private int m_Breite;  
}  
...
```

Überlagern von Methoden (Fort.)

```
...  
class Quadrat extends Rechteck {  
    Quadrat(int iBreite) {  
        super(iBreite,iBreite);  
    }  
}  
...
```

ein Quadrat ist ebenfalls
eine Figur aber vor allem
ein Rechteck ...

... und muss sich daher
nichts merken ...

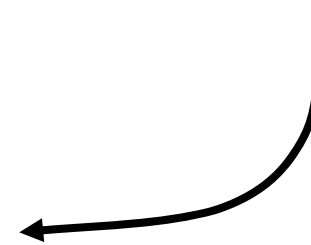
... und hat auch keine
eigene Implemen-
tierung für das Zeichnen

Go!

Überlagern von Methoden (Fort.)

```
public class Virtual4 {  
  
    public static void main(String[] args) {  
        Figur[] x = new Figur[10];  
        for(int i = 0; i < x.length; ++i) {  
            final double D = Math.random();  
            if (D < 0.33)  
                x[i] = new Dreieck(1+i);  
            else if (D < 0.66)  
                x[i] = new Quadrat(1+i);  
            else  
                x[i] = new Rechteck(1+i, 15);  
        }  
        for(int i = 0; i < x.length; ++i) {  
            x[i].draw('#');  
            System.out.println();  
        }  
    }  
}
```

jetzt können Objekte dieser
3 Klassen zufällig erzeugt
werden



Überlagern von Methoden (Fort.)


- Aufgabe: Quadrate sollen mit einem anderen Zeichen gezeichnet werden, als Rechtecke
- mögliche Lösung: die Klasse Quadrat erhält ihre eigene draw-Methode

Problem:

- m_Hoehe und m_Breite sind private Objektvariablen von Rechteck und somit nicht lesbar in Quadrat
- viel Schreibaufwand

```
class Quadrat extends Rechteck {  
    Quadrat(int iBreite) {  
        super(iBreite,iBreite);  
    }  
    void draw(char c) {  
        for(int i = 0;i < m_Hoehe;++i) {  
            for(int j = 0;j < m_Breite;++j) {  
                System.out.print('o');  
            }  
            System.out.println();  
        }  
    }  
}
```

Quadrat mit o's zeichnen



Go!

Überlagern von Methoden (Fort.)

- bessere Lösung: Implementierung von Rechteck direkt aufrufen
- dies erfolgt analog zu dem Konstruktoraufruf der Basisklasse mit dem super Befehl

```
class Quadrat extends Rechteck {  
    Quadrat(int iBreite) {  
        super(iBreite,iBreite);  
    }  
  
    void draw(char c) {  
        super.draw('o');  
    }  
}
```

ruft die draw-Methode
von Rechteck auf

Go!

Überlagern von Methoden (Fort.)

```
class A {  
    A() {  
        init();  
    }  
    void init() {  
        System.out.println("init von A");  
    }  
}
```

im Konstruktor
wird init aufgerufen

Was wird
ausgegeben?

```
}  
public class Virtual9 extends A {  
    void init() {  
        System.out.println(m_i);  
    }  
    public static void main(String[] args) {  
        new Virtual9();  
    }  
    private int m_i = 13;  
}
```

init wird hier überlagert
und greift auf die eigenen
Objektvariablen zu

Objektvariable m_i wird
direkt initialisiert

Überlagern von Methoden (Fort.)

- Vorsicht: keine überlagerten Methoden im Basiskonstruktor aufrufen
- werden Methoden im Konstruktor aufgerufen, sollte diese Methoden später nicht mehr überlagert werden können
- also: das Überlagern sollte erlaubt und verboten werden können

Überlagern von Methoden (Fort.)

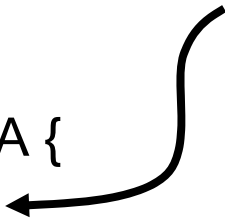
- die Identifikation, welche der überlagerten Methoden zur Laufzeit ausgeführt werden soll, nennt man *dynamisches Binden*
- diese Identifikation kostet Laufzeit und ist nicht billig
- manchmal muss sie auch verboten werden können
- daher muss in Sprachen wie C++ direkt gesagt werden, welche Methoden dynamisch gebunden werden soll (mittels des Schlüsselworts `virtual`)
- in Java kann man das dynamische Binden unterdrücken, indem
 - die Methode privat gemacht wird
 - die Methode `static` ist, oder
 - die Methode mittels des Modifiers `final` konstant gemacht wird

Überlagern von Methoden (Fort.)

- Verhinderung des dynamischen Bindens durch Verwendung von final

```
1 class A {  
2     final void nice() {}  
3 }  
4  
5 class B extends A {  
6     void nice() {}  
7 }
```

hier wird versucht, eine
konstante Methode zu
überlagern



führt zu folgendem
Compilerfehler

```
Virtual6.java:6: nice() in B cannot override nice() in A;  
overridden method is final
```

```
    void nice() {}  
        ^
```

```
1 error
```

Überlagern von Methoden (Fort.)

- Verhinderung des dynamischen Bindens durch die Verwendung von `private`
- hier wird *zur Compilezeit* bestimmt, welche Methode auszuführen ist
- die überlagernde Methode wird *nicht* in die Tabelle *eingetragen*

Go!

Überlagern von Methoden (Fort.)

Was wird
ausgegeben?

doit1 kann nicht
überlagert werden

```
class A {  
    private void doit1() {System.out.println("ich bin doit1 von A");}  
    void doit2() {System.out.println("ich bin doit2 von A");}  
    void makelt() {  
        doit1();  
        doit2();  
    }  
}
```

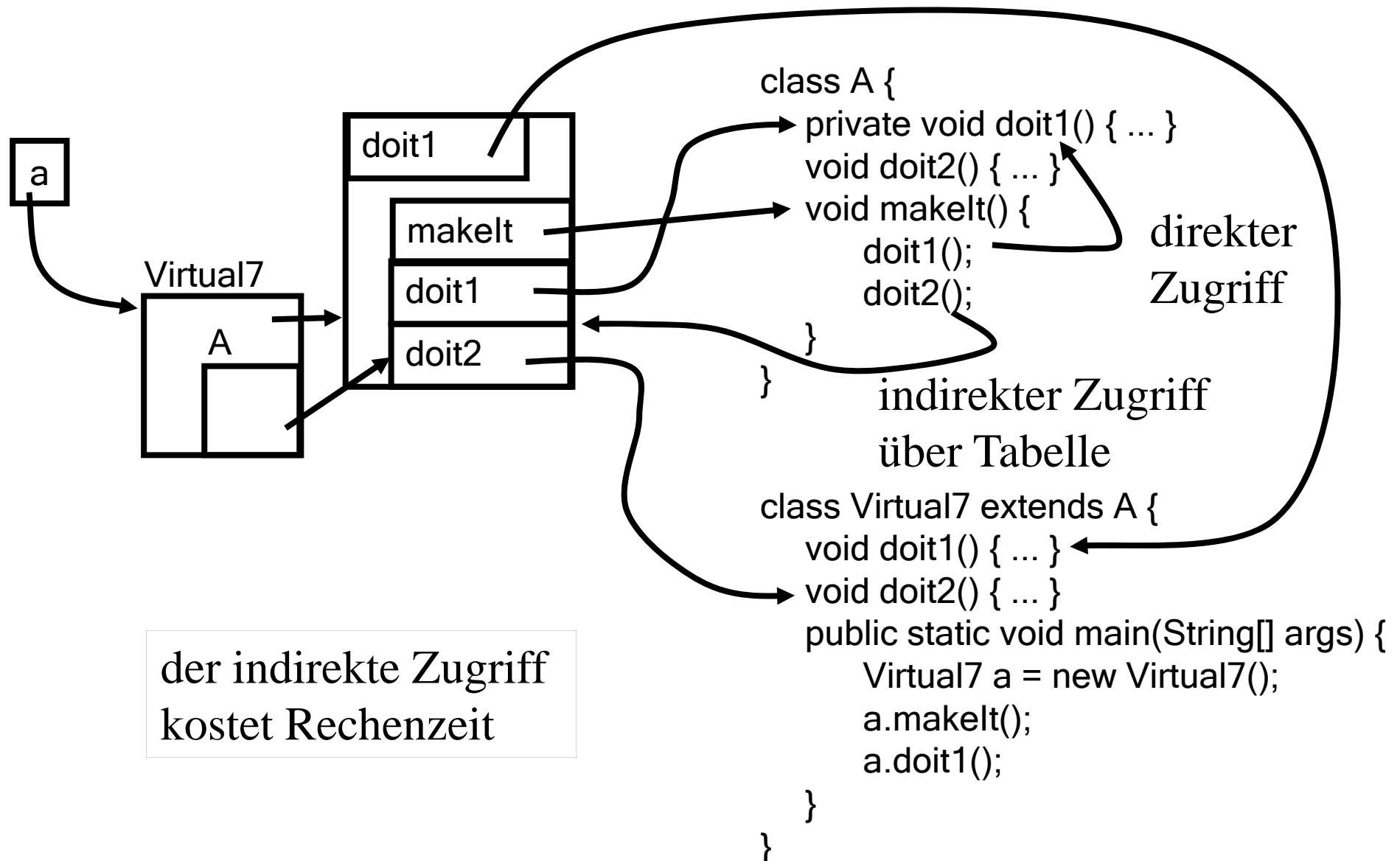
zur Compilezeit bestimmen

zur Laufzeit bestimmen

```
class Virtual7 extends A {  
    void doit1() {System.out.println("ich mache auch etwas");}  
    void doit2() {System.out.println("ich auch");}  
    public static void main(String[] args) {  
        Virtual7 a = new Virtual7();  
        a.makelt();  
        a.doit1();  
    }  
}
```

doit1 kann aber neu
definiert werden

Überlagern von Methoden (Fort.)



Überlagern von Methoden (Fort.)

- Verhinderung des dynamischen Bindens durch die Verwendung von static
- hier wird *zur Compilezeit* bestimmt, welche Methode auszuführen ist
- die Bestimmung findet ausschließlich über die Typinformation zur Compilezeit statt
- 3 Situationen: die statische Methode ...
 - ... wird über den Klassennamen aufgerufen
 - ... wird über einen Objektnamen aufgerufen
 - ... wird aus einem Kontext direkt aufgerufen

Überlagern von Methoden (Fort.)

- Klassenname: nehme die Methode der Klasse
- Objektname: schaue, von welcher Klasse die Variable ist und nehme die Methode dieser Klasse
- Kontext: nehme die Methode des aktuellen Kontextes

Go!

Überlagern von Methoden (Fort.)

```
class A {  
    static void doit() {  
        System.out.println("ich bin doit von A");  
    }  
}  
class Virtual8 extends A {  
    static void doit() {  
        System.out.println("ich nicht");  
    }  
    static void juhu(A a) {  
        a.doit();  
        doit();  
        A.doit();  
    }  
    public static void main(String[] args) {  
        Virtual8 v = new Virtual8();  
        v.doit();  
        juhu(v);  
    }  
}
```

statische Methoden können nicht überlagert werden

Was wird ausgegeben?

Auflösung über die Klasse des Parameters a

Auflösung über den Kontext

Auflösung über den Klassennamen

Auflösung über die Klasse der Variablen v

Vorlesung 12/2

Abstrakte Klassen

```
class Figur {  
    void draw(char c) {}  
}
```


- nochmals die Klasse Figur
- für die Klasse Figur gibt es *keine sinnvolle Implementierung* der draw-Methode
- alle *abgeleiteten Klassen* haben ihre *eigene Implementierung* der draw-Methode
- es wird *niemals ein Objekt der Klasse Figur* angelegt
- hierfür gibt es ein Sprachkonstrukt: *abstrakte Klassen*

```
abstract class Figur {  
    abstract void draw(char c);  
}
```

Abstrakte Klassen (Fort.)

- von abstrakten Klassen können keine Objekte erzeugt werden

```
1  abstract class A {  
2      abstract void doit();  
3  }  
4  
5  class Abstract1 {  
6      public static void main(String[] args) {  
7          new A();  
8      }  
9  }
```



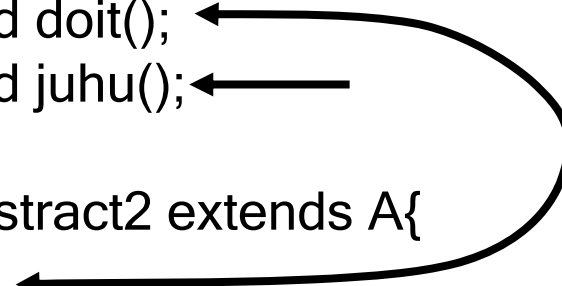
```
Abstract1.java:7: A is abstract; cannot be instantiated  
    new A();  
        ^
```

1 error

Abstrakte Klassen (Fort.)

- eine von einer abstrakten Klasse *abgeleiteten Klasse* ist *wiederum abstrakt*, wenn sie *nicht alle abstrakten Methoden implementiert*

```
1  abstract class A {  
2      abstract void doit();  
3      abstract void juhu();  
4  }  
5  public class Abstract2 extends A {  
6      void doit() {  
7          System.out.println("doit");  
8      }  
10     public static void main(String[] args) {  
11         new Abstract2();  
12     }  
13 }
```



```
Abstract2.java:5: Abstract2 should be declared abstract; it  
does not define juhu() in A  
public class Abstract2 extends A {  
    ^
```

1 error

Abstrakte Klassen (Fort.)

- abstrakte Methoden (und damit Klassen) machen besonders Sinn, wenn die Methode einen Wert zurückliefert
- Bsp.: die Klasse Figur soll eine Methode flaechenInhalt erhalten, die den Flächeninhalt der Figur zurückliefert
- für die Implementierung in Figur gäbe es keinen sinnvollen Wert, der zurückgeliefert werden könnte
- daher wird sie als abstrakt deklariert und erst in den abgeleiteten Klassen implementiert

Abstrakte Klassen (Fort.)

```
abstract class Figur {  
    abstract void draw(char c);  
    abstract int flaechenInhalt();  
}
```

← beide Methoden sind abstrakt

```
class Dreieck extends Figur {  
....  
    int flaechenInhalt() {  
        return m_Hoehe * m_Hoehe / 2;  
    }  
...  
}
```

← für flaechenInhalt hätte auch kein sinnvoller Standardwert zurückgegeben werden können

```
class Rechteck extends Figur {  
...  
    int flaechenInhalt() {  
        return m_Hoehe * m_Breite;  
    }  
...  
}
```

← individuelle Berechnungen der Flächeninhalte

Go!

Abstrakte Klassen (Fort.)

```
...  
public class Abstract3 {  
  
    public static void main(String[] args) {  
        Quadrat q = new Quadrat(9);  
        Dreieck d = new Dreieck(6);  
        System.out.println(q.flaechenInhalt());  
        System.out.println(d.flaechenInhalt());  
    }  
}
```

Objekte können angelegt
werden, da alle abstrakten
Methoden implementiert
sind

Was wird
ausgegeben?

Abstrakte Klassen (Fort.)

- auch wenn von abstrakten Klassen keine Objekte angelegt werden können, können sie Objektvariablen und –methoden enthalten
- auch Klassenvariablen und –methoden können in abstrakten Klassen definiert werden

Go!

Abstrakte Klassen (Fort.)

Was wird
ausgegeben?

```
abstract class A {  
    A(int i) {  
        m_i = i;  
    }  
    abstract void doit();  
    static void juhu() {  
        System.out.println("ich bin juhu");  
    }  
    int m_i;  
}  
  
public class Abstract4 extends A {  
    Abstract4() {  
        super(13);  
    }  
    void doit() {  
        System.out.println(m_i);  
    }  
    public static void main(String[] args) {  
        A.juhu();  
        Abstract4 a = new Abstract4();  
        a.doit();  
    }  
}
```

abstrakte Klassen
können auch Objekt-
variablen besitzen

Klassenmethoden von
abstrakten Klassen können
direkt aufgerufen werden

Go!

Abstrakte Klassen (Fort.)

- auch wenn von abstrakten Klassen keine Objekte angelegt werden können, kann es Variablen vom Typ dieser Klasse geben
- die Variable kann natürlich nur Objekte von abgeleiteten Klassen enthalten (warum?)

obwohl A abstrakt ist, kann man ein A Objekt übergeben

```
abstract class A {  
    abstract void doit();  
}  
class B extends A {  
    void doit() {  
        System.out.println("B");  
    }  
}  
class C extends A {  
    void doit() {  
        System.out.println("C");  
    }  
}  
...
```

doit wird implementiert

```
...  
class Abstract5 {  
    static void makelt(A a) {  
        a.doit();  
    }  
    public static void main(String[] args) {  
        makelt(new C());  
        makelt(new B());  
    }  
}
```

Was wird
ausgegeben?

Go!

Interfaces

- *abstrakte Klassen*, die
 - nur abstrakte Methoden und / oder Konstanten enthalten,
 - *keine Objektvariablen* enthalten (Ausnahme: Konstanten)
 - *keine nicht-abstrakten Objektmethoden* enthalten
 - *keine Klassenmethode* enthalten

können auch als sogenannten *Interfaces* realisiert werden

ein Interface fängt mit dem Schlüsselwort `interface` an

```
interface A {  
    public void doit();  
    public int m_i = 13;  
}
```

keine Variable sondern Konstante

man *erbt nicht* von Interfaces, sondern *implementiert* sie

```
class B implements A {  
    public void doit() {  
        System.out.println("B");  
    }  
}
```

Interfaces (Fort.)

- *implementiert* eine Klasse *nicht alle Methoden* aus einem Interface, so muss die Klasse als abstract deklariert werden
- von solch einer Klasse kann natürlich kein Objekt erzeugt werden (warum?)

```
interface A {
```

```
    public void doit();  
    public void juhu();
```

```
}
```

doit wird implementiert, juhu nicht

```
abstract class B implements A {
```

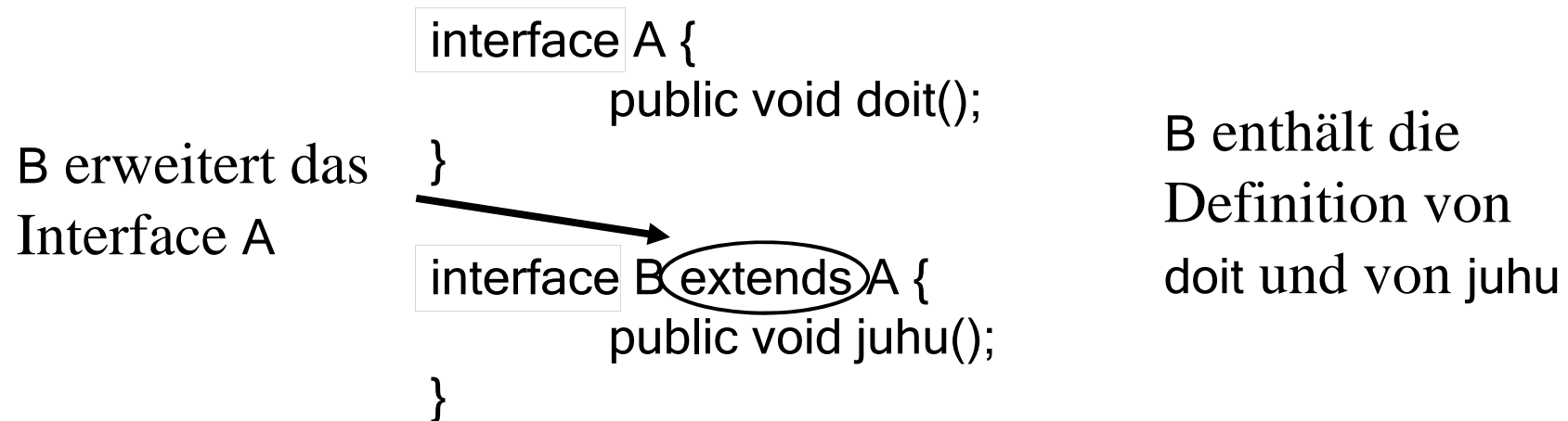
```
    public void doit() {  
        System.out.println("B");
```

```
    }
```

```
}
```


Interfaces (Fort.)

- analog zu Klassen können auch *Interfaces voneinander abgeleitet* werden
- das *Ergebnis* ist *nicht eine Klasse, sondern* wieder *ein Interface*, dass die ererbten Methoden und die neu definierten Methoden deklariert



Go!

Interfaces (Fort.)

- um ein abgeleitetes Interface zu implementieren, müssen alle Methoden implementiert werden

2 Interfaces

```
interface A {  
    public void doit();  
}  
interface B extends A {  
    public void juhu();  
}  
class C implements B {  
    public void doit() {  
        System.out.println("C");  
    }  
    public void juhu() {  
        System.out.println("auch C");  
    }  
}
```

...
}

implements
tiert alles

...

```
public class Interface3 {  
    static void makelt(A a) {  
        a.doit();  
    }  
    static void makelt(B b) {  
        b.juhu();  
    }  
    public static void main(String[] args) {  
        C c = new C();  
        A a = c;  
        makelt(a);  
        makelt(c);  
    }  
}
```

Überladung

2 Verweise
auf 1 Objekt

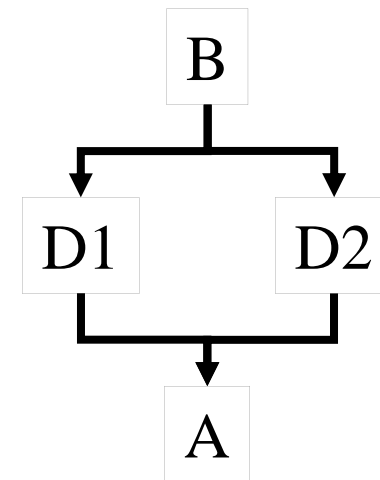
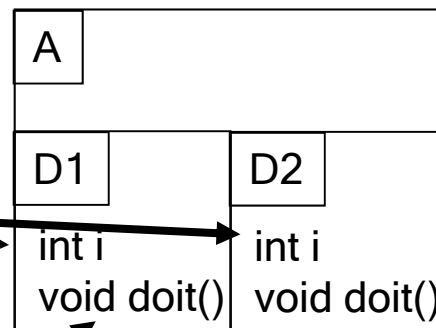
Interfaces (Fort.)

In Java gibt es keine Mehrfachvererbung (wie in C++).
Grund:

```
class B {  
    int i;  
    void doit() { ... };  
}
```

...

```
A a = new A();  
a.i = 42;  
a.doit();
```



Interfaces (Fort.)

Problem bei Mehrfachvererbung:

- welcher Member (Objektvariable) ist gemeint?
- welche Methode ist gemeint?

Frage:

was ist, wenn die mehrfachvererbte Klasse gar keine Members hat und alle Methoden abstrakt sind?

Interfaces (Fort.)

Vorteil von Interfaces gegenüber (abstrakten) Klassen:

- eine Klasse kann mehrere Interfaces implementieren,
- aber nur von einer (abstrakten) Klasse abgeleitet werden

```
interface A {  
    public void doit();  
}  
interface B {  
    public void juhu();  
}  
class C implements B, A {  
    public void doit() {  
        System.out.println("doit");  
    }  
    public void juhu() {  
        System.out.println("juhu");  
    }  
}
```

2 Interfaces, die nichts miteinander zu tun haben

C implementiert beide Interfaces

Go!

Interfaces (Fort.)

- durch gleichzeitige Implementierung mehrere Interfaces sind die Objekte von mehreren Typen

```
...
public class Interface4 {
    static void makelt(A a) {
        a.doit();
    }
    static void makelt(B b) {
        b.juhu();
    }
    public static void main(String[] args) {
        C c = new C();
        A a = c;
        B b = c;
        makelt(a);
        makelt(b);
    }
}
```

2 total unterschiedliche Typen

c hat 3 Typen:
C, A und B

3 Verweise
auf 1 Objekt


Go!

Interfaces: Java 8

- ab Version Java 8 gibt es default Implementierungen in Interfaces
- Grund: Interfaces könnten sonst im nachhinein nicht erweitert werden, ohne alle Implementierungen anzupassen

```
interface A {  
    public void doit();  
    default public int juhu(int i) {  
        return i*i;  
    }  
}  
  
public class Interface5 implements A {  
    public void doit() {  
        System.out.println(juhu(3));  
    }  
    public static void main(String[] args) {  
        Interface5 o = new Interface5();  
        o.doit();  
    }  
}
```

2. Methode des
Interfaces hat schon
eine Implementierung



Go!

Interfaces: Java 8 (Forts.)

- Standardimplementierungen können natürlich überlagert werden

```
interface A {  
    public void doit();  
    default public int juhu(int i) {  
        return i*i;  
    }  
}  
  
public class Interface6 implements A {  
    public int juhu(int i) {  
        return i*2;  
    }  
    public void doit() {  
        System.out.println(juhu(3));  
    }  
    public static void main(String[] args) {  
        Interface6 o = new Interface6();  
        o.doit();  
    }  
}
```

← 2. Methode des Interfaces
hat schon eine
Implementierung, die ...

← ... hier überlagert wird

Go!

Interfaces: Java 8 (Forts.)

- Mit Standardimplementierungen kommen fast alle Probleme der Mehrfachvererbung wieder auf

```
interface A {  
    public void doit();  
    default public int juhu(int i) {  
        return i*i;  
    }  
}  
  
interface B {  
    public void doit();  
    default public int juhu(int i) {  
        return i*2;  
    }  
}  
  
public class Interface7 implements A,B {  
    public void doit() {  
        System.out.println(juhu(3));  
    }  
    public static void main(String[] args) {  
        Interface6 o = new Interface6();  
        o.doit();  
    }  
}
```

kein Widerspruch

Problem: welches
,juhu' ist gemeint?

Go!

Interfaces: Java 8 (Forts.)

- Folgende drei Probleme können auftreten:
- 1. Situation: Konflikt zwischen Interface und Basisklasse
- Lösung: Basisklasse gewinnt!

```
interface A {  
    default public int juhu(int i) {  
        return i*i;  
    }  
}  
class B {  
    public int juhu(int i) {  
        return i*2;  
    }  
}  
public class Interface8 extends B implements A {  
    public static void main(String[] args) {  
        Interface6 o = new Interface6();  
        System.out.println(o.juhu(3));  
    }  
}
```

Problem: ‚juhu‘ von Interface
‚A‘ oder Basisklasse ‚B‘?

Lösung: Basisklasse
gewinnt

Go!

Interfaces: Java 8 (Forts.)

- 2. Situation: Konflikt zwischen Interfaces in Ableitungen
- Lösung: letzte Implementierung gewinnt!

```
interface A {  
    default public int juhu(int i) {  
        return i*i;  
    }  
}
```

```
interface B extends A {  
    default public int juhu(int i) {  
        return i*2;  
    }  
}
```

```
public class Interface9 implements B {  
    public static void main(String[] args) {  
        Interface9 o = new Interface9();  
        System.out.println(o.juhu(3));  
    }  
}
```

Problem: ‚juhu‘ von Interface
‚A‘ oder Interface ‚B‘?

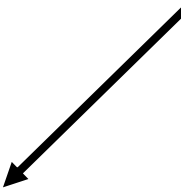
Lösung: letzte
Implementierung
(hier: ‚B‘) gewinnt

Interfaces: Java 8 (Forts.)

- 3. Situation: Konflikt zwischen Interfaces, die parallel implementiert werden
- Lösung: Fehler!

```
interface A {  
    default public int juhu(int i) {  
        return i*i;  
    }  
}  
interface B {  
    default public int juhu(int i) {  
        return i*2;  
    }  
}  
public class Interface10 implements A,B {  
  
    public static void main(String[] args) {  
        Interface10 o = new Interface10();  
        System.out.println(o.juhu(3));  
    }  
}
```

Interface10.java:13: error:
class Interface10 inherits
unrelated defaults for juhu(int)
from types A and B



This

- werden Objekte in Variablen gespeichert, so kann man die Objekte unter dem Namen der Variable ansprechen

`A juhu = new A();` // Objekt ist unter dem Namen juhu bekannt

- im Kontext eines Objektes möchte man manchmal auf sich selber verweisen
- im natürlichsprachlichem ist das "ich"
- im OOP ist dies das Schlüsselwort "this"

This (Fort.)

- this kann verwendet werden, um zwischen lokalen Variablen oder Parametern und Objektvariablen gleichen Namens zu unterscheiden
- durch das Voranstellen von this werden direkt die Objektvariablen referenziert

Go!

This (Fort.)

```
class A {  
    private int i;  
    private int j;  
  
    A(int i,int j) {  
        this.i = i;  
        this.j = j;  
    }  
    public void doit() {  
        System.out.println("i = "+ i + ", j = " +j);  
    }  
}
```

Objektvariablen und Parameter
haben identische Namen

ohne this ist der Parameter gemeint

mit this ist die Objekt-
variable gemeint

bei Eindeutigkeit muss this
nicht verwendet werden

```
public class This1 {  
    public static void main(String[] args) {  
        A a = new A(23,12);  
        a.doit();  
    }  
}
```

This (Fort.)

- this kann verwendet werden, um einen Konstruktor aufzurufen
- hat man mehrere Konstrukteure (durch Überladung), so kann einer durch den anderen implementiert werden, indem im Konstruktor durch this der andere aufgerufen wird
- dadurch bekommen Parameter Standardwerte

dies ist der
allgemeinere
Konstruktor

Klasse mit 2
(überladenen)
Konstrukturen

```
class A {  
    A(int i) { ... }  
    A() {  
        this(42);  
    }  
    ...  
}
```

Standard-
wert

der speziellere Konstruktor
ruft den Allgemeineren auf

```
A a1 = new A();  
A a2 = new A(13);
```


Go!

This (Fort.)

```
class A {  
    private int i; private int j;  
    A(int i, int j) {  
        this.i = i; this.j = j;  
    }  
    A(int i) {  
        this(i, 13);  
    }  
    A() {  
        this(43);  
    }  
    public void doit() {  
        System.out.println("i = " + i + ", j = " + j);  
    }  
}  
public class This2 {  
    public static void main(String[] args) {  
        A a1 = new A(23, 12); A a2 = new A(23); A a3 = new A();  
        a1.doit(); a2.doit(); a3.doit();  
    }  
}
```

definiert einen Standardwert
für den 2. Parameter

definiert einen Standardwert
für den 1. Parameter

A kann mit 2, 1 oder 0
Parameter aufgerufen werden

Go!

```
class B {  
    B(A a) { m_a = a; }  
    void doit(int i) {  
        m_a.juhu(i);  
    }  
    A m_a;  
}
```

B erwartet
ein A-Objekt

This (Fort.)

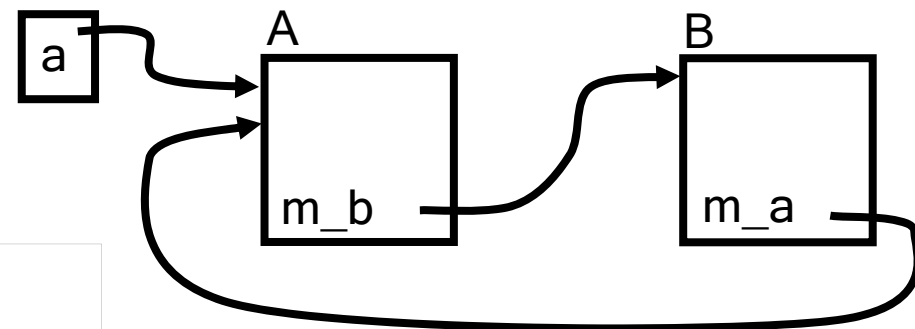
- this kann verwendet werden, um sich als Objekt zu übergeben

```
class A {  
    A() {m_b = new B(this);}  
    void juhu(int i) {  
        if (i < 10) {  
            System.out.println(i);  
            m_b.doit(i+1);  
        }  
    }  
    B m_b;  
}
```

bei der Erzeugung von A wird
ein B-Objekt erzeugt, dem
man sich selber übergibt

```
public class This3 {  
    public static void main(String[] args) {  
        A a = new A();  
        a.juhu(0);  
    }  
}
```

Was wird
ausgegeben?



A und B kennen
sich gegenseitig

Vorlesung 13/1


Go!

Fehlerbehandlung: Exceptions

- Exceptions dienen dazu, Laufzeitfehler im Programm anzuzeigen
- tritt ein Fehler auf, so wird eine sogenannte *Exception geworfen* oder *ausgelöst*
- der Programmierer hat die Möglichkeit, Exceptions abzufangen und zu behandeln

```
public class Exception1 {  
  
    public static void main(String[] args) {  
        int[] a = new int[4];  
        a[34] = 0;  
    }  
}
```

bei diesem Arrayzugriff
entsteht ein Fehler



Exceptions: Behandlung

- rechnet man in einem Programmstück damit, dass Exceptions auftreten können, so kann man diese Exceptions abfangen und behandeln
- dies erfolgt in einem sogenannten *try-catch Block*
- die Anweisung sieht wie folgt aus

```
try {  
    <Anweisung>;  
    <Anweisung>;  
    ...  
} catch (<Ausnahmetyp> x) {  
    <Anweisung>;  
    <Anweisung>;  
    ...  
}
```

hier steht der normale
Programmcode

tritt oben ein Fehler vom
Typ <Ausnahmetyp> auf, so
werden diese Anweisungen
ausgeführt

Go!

Exceptions: Beispiel

- die Exception `ArrayIndexOutOfBoundsException` kann vom Arrayzugriff erzeugt werden
- diese kann in einer catch-Anweisung abgefangen und behandelt werden

```
public class Exception2 {
```

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

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

```
        try {
```

```
            a[34] = 0;
```

```
        } catch (ArrayIndexOutOfBoundsException x) {
```


```
            System.out.println("das war wohl nix");
```

```
        }
```

```
    }
```

```
}
```

hier wird normal auf
das Array zugegriffen



wird außerhalb der Arraygrenzen zu-
gegriffen, wird dieser Code ausgeführt



Go!

Exceptions: Beispiel

- wird von einer Anweisung im try-Block eine Exception ausgelöst, werden die nachfolgenden Anweisungen nicht mehr ausgeführt

```
public class Exception19 {
```

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

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

```
        try {
```

```
            a[34] = 0;
```

```
            System.out.println("das dürfte nicht mehr ausgegeben werden");
```

```
        } catch (ArrayIndexOutOfBoundsException x) {
```

```
            System.out.println("das war wohl nix");
```

```
        }
```

```
    }
```

```
}
```

die Print Anweisung wird
nicht mehr ausgeführt



wird außerhalb der Arraygrenzen zu-
gegriffen, wird dieser Code ausgeführt



Go!

Exceptions (Fort.)

- **WICHTIG:** der catch-Block wird natürlich nur dann ausgeführt, wenn auch der Fehler wirklich auftritt
- tritt der Fehler nicht auf, so wird der catch-Block einfach übersprungen

```
public class Exception3 {
```

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

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

```
        try {
```

```
            a[3] = 0;
```

```
        } catch (ArrayIndexOutOfBoundsException x) {
```

```
            System.out.println("das war wohl nix");
```

```
        }
```

```
        System.out.println("diesmal sollte es gutgehen");
```

```
    }
```

```
}
```

hier wird normal auf
das Array zugegriffen

tritt kein Fehler
auf, sollte dies
nicht ausgeführt
werden

das wird in jedem Fall ausgeführt


Go!

Beispiel

```
public class Exception4 {
```

```
    static int magic(int i) {  
        return magic(i+1);  
    }
```

Endlosrekursion, die einen
Stackoverflow produziert



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

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

```
        try {
```

```
            a[42] = magic(3);
```

```
        } catch (ArrayIndexOutOfBoundsException x) {
```

```
            System.out.println("das war wohl nix");
```

```
        }
```

```
        System.out.println("diesmal sollte es gutgehen");
```

```
    }
```

```
}
```

hier stecken 2 Fehlerquellen



Was macht dieses Programm?

Exceptions (Fort.)

- können mehrere Fehler auftreten und sollen alle behandelt werden, so müssen mehrere catch-Blöcke nacheinander aufgeführt werden
- jeder catch-Block ist zur Behandlung eines bestimmten Fehlers zuständig

```
try {  
    <Anweisung>;  
    ...  
} catch (<Ausnahmetyp> x) {  
    <Anweisung>;  
    ...  
} catch (<Ausnahmetyp> y) {  
    <Anweisung>;  
    ...  
} catch (<Ausnahmetyp> z) {  
    ...  
}
```

Go!

Beispiel

```
public class Exception5 {  
  
    static int magic(int i) {  
        return magic(i+1);  
    }  
  
    public static void main(String[] args) {  
        int[] a = new int[4];  
        try {  
            a[42] = magic(3);  
        } catch (ArrayIndexOutOfBoundsException x) {  
            System.out.println("das war wohl nix");  
        } catch (StackOverflowError x) {  
            System.out.println("das war wohl zu aufwendig zu berechnen");  
        }  
        System.out.println("diesmal sollte es gutgehen");  
    }  
}
```

hier wird der Array-
zugriff abgefangen

hier wird der
Stackoverflow
abgefangen

Das Fehlerobjekt

- durch die Exception wird ein Fehlerobjekt geworfen
- die catch-Regeln versuchen, dieses Objekt abzufangen
- wie jedes Objekt hat auch dieses Fehlerobjekt einen Typen
- dies ist der Typ, den man in der catch-Regel angibt
- das Fehlerobjekt ist dann unter dem Variablennamen, der in der catch-Regel angegeben wird, ansprechbar
- ein Fehlerobjekt hat die 3 Methoden

<code>public String getMessage();</code>	konvertiert die Fehler- meldung in einen String
<code>public String toString();</code>	
<code>public void printStackTrace();</code>	druckt den Stackverlauf, der zum Fehler geführt hat

Go!

Beispiel

```
public class Exception6 {
```

```
    static void doit(int[] a) {
```

```
        try {
```

```
            a[42] = 0;
```

```
        } catch (ArrayIndexOutOfBoundsException x) {
```

```
            System.out.println(x.toString());
```

```
            System.out.println(x.getMessage());
```

```
            System.out.println("Der Fehler ist hier entstanden:");
```

```
            x.printStackTrace();
```

```
        }
```

```
    }
```

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

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

```
        doit(b);
```

```
    }
```

```
}
```

gebe die Fehler-
meldung 2-mal aus

drucke den Stack aus, der
zum Fehler geführt hat

Go!

Fehler in der Fehlerbehandlung

- tritt in der Fehlerbehandlung erneut ein Fehler auf, so wird dieser Fehler *nicht* von den catch-Blöcken behandelt
- auch nicht von den nachfolgenden

```
public class Exception7 {  
    static void doit() {  
        doit();  
    }  
    public static void main(String[] args) {  
        int[] a = new int[4];  
        try {  
            a[42] = 0;  
        } catch (ArrayIndexOutOfBoundsException x) {  
            doit();  
        } catch (StackOverflowError y) {  
            System.out.println("das war ein bisschen viel");  
        }  
    }  
}
```

Endlosrekursion: löst
StackOverflowError aus

Fehler, der hier behandelt wird

erneuter Fehler,
der nirgends
behandelt wird

Go!

Fehler in der Fehlerbehandlung (Fort.)

- jedoch können try-catch-Blöcke ineinander geschachtelt sein, um Fehler in der Fehlerbehandlung abzufangen

```
public class Exception8 {  
    static void doit() {doit();}  
    public static void main(String[] args) {  
        int[] a = new int[4];  
        try {  
            a[42] = 0;  
        } catch (ArrayIndexOutOfBoundsException x) {  
            try {  
                doit();  
            } catch (StackOverflowError z) {  
                System.out.println("jetzt habe ich es geschafft");  
            }  
        } catch (StackOverflowError y) {  
            System.out.println("das war ein bisschen viel");  
        }  
    }  
}
```

bezieht
sich
hierauf

geschachtelter
try-catch-Block

Die Finally Regel

- am Ende eines try-Blocks mit einer oder mehreren catch-Blöcken kann eine sogenannte finally-Regel angegeben werden
- sie enthält Code, der in jedem Fall am Ende ausgeführt wird
- wird der try-Block normal abgearbeitet, so wird danach die finally-Regel abgearbeitet
- tritt im try-Block ein Fehler auf, so wird erst die zugehörige catch-Regel ausgeführt und dann der finally-Block

```
try {  
    ...  
} catch (<Ausnahmetyp> x) {  
    ...  
} finally {  
    ...  
}
```


Go!

Beispiel

```
public class Exception9 {  
    static void doit(int[] a,int i) {  
        try {  
            a[i] = 0;  
        } catch (ArrayIndexOutOfBoundsException x) {  
            System.out.println("bisschen zu gross der Wert");  
        } finally {  
            System.out.println("ich werde immer ausgefuehrt");  
        }  
    }  
  
    public static void main(String[] args) {  
        int[] b = new int[4];  
        doit(b,42);  
        doit(b,3);  
    }  
}
```

ob eine Exception
geworfen wird oder
nicht hängt von i ab

sie sollte immer
ausgeführt werden

mal ein Fehler,
mal keiner

Go!

Die Finally Regel (Fort.)

- der finally-Block wird *in jedem Fall* ausgeführt, egal was im try- oder den catch-Blöcken passiert

```
public class Exception10 {
```

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

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

```
        try {
```

```
            a[42] = 0;
```

```
        } catch (ArrayIndexOutOfBoundsException x) {
```

```
            a[42] = 0;
```

```
            System.out.println("bisschen zu gross der Wert");
```

```
        } finally {
```

```
            System.out.println("ich werde immer ausgefuehrt");
```

```
        }
```

```
    }
```

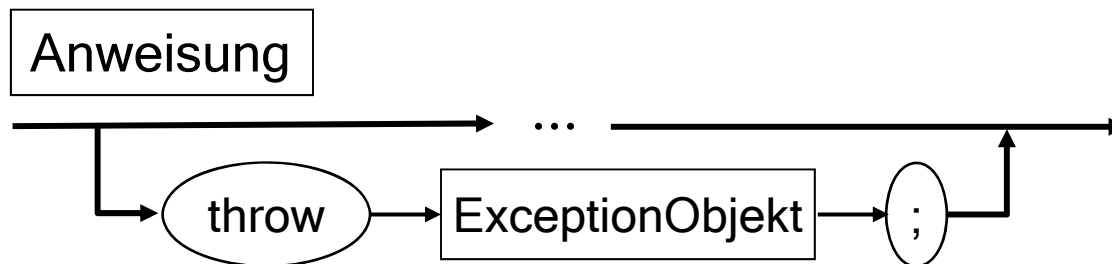
```
}
```

in einem catch-Block
tritt *erneut ein Fehler*
auf, der *nicht* behandelt
wird

wird immer ausgeführt

Fehler erzeugen

- eigene Exception löst man mittels der throw Anweisung aus
- nach dem throw muss das Ausnahmeobjekt folgen, dass dann später eventuell in einem übergeordnetem catch-Block gefangen wird



Go!

Beispiel

```
public class Exception11 {  
    static int div(int i) {  
        if (i != 0)  
            return 42 / i;  
        else  
            throw new ArrayIndexOutOfBoundsException();  
    }  
    static void doit(int i) {  
        try {  
            System.out.println(div(i));  
        } catch (ArrayIndexOutOfBoundsException x) {  
            System.out.println("bisschen zu 0 der Wert");  
        }  
    }  
    public static void main(String[] args) {  
        doit(3);  
        doit(0);  
    }  
}
```

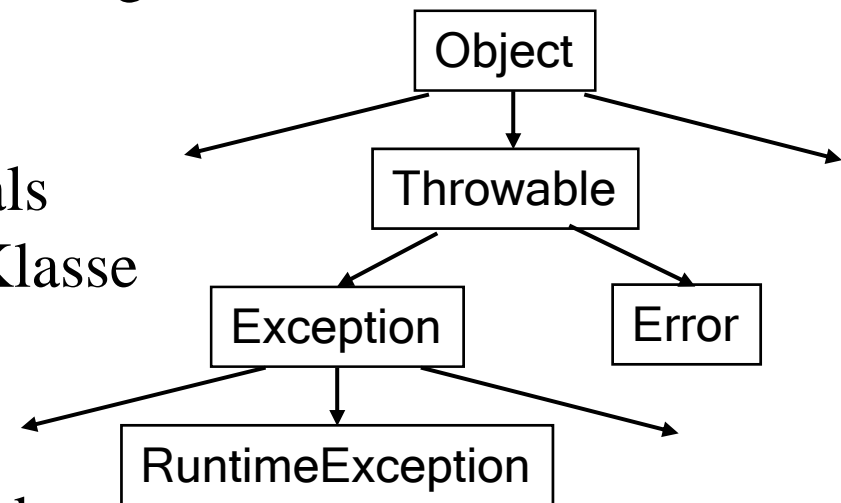
div wirft eine Exception,
wenn i gleich 0 ist

behandelt den Fehler, der
in div auftreten kann

mal korrekt,
mal ein Fehler

Eigene Fehler erzeugen

- Möchte man Objekte einer selbstgeschriebenen Fehlerklasse als Exception werfen, so muss diese Klasse von Throwable (oder einer der Subklassen) abgeleitet sein
- Methoden, die Exceptions direkt oder indirekt Exceptions auslösen können, müssen dies im Methodenkopf deklarieren



```
void doit(int i) throws MyException {  
    ...  
    throw new MyException();  
    ...  
}
```

schon im Methodenkopf sieht man, welche Exceptions geworfen werden können

Go!

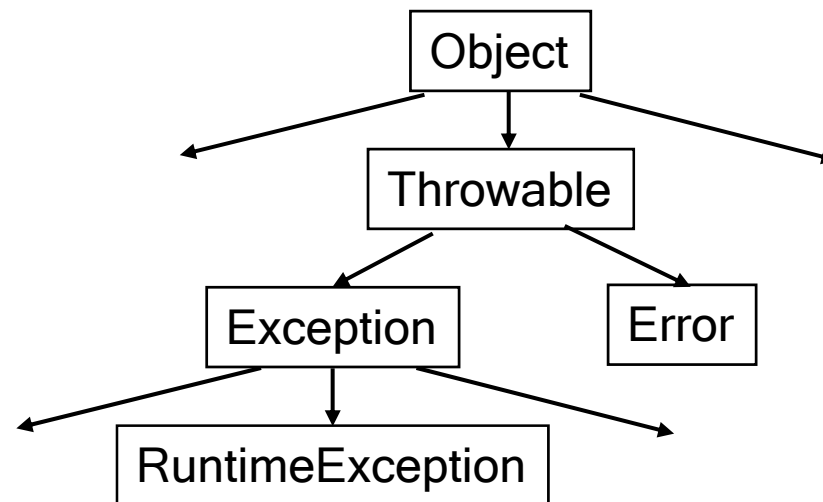
Eigene Fehler erzeugen: Beispiel

```
class MyException extends Throwable {  
}  
public class Exception14 {  
    static int div(int i) throws MyException {  
        if (i != 0)  
            return 42 / i;  
        else  
            throw new MyException();  
    }  
    public static void main(String[] args) {  
        try {  
            System.out.println(div(0));  
        } catch (MyException x) {  
            System.out.println("bisschen zu 0 der Wert");  
        }  
    }  
}
```

spezifiziert, dass
MyException geworfen
werden kann

Die RuntimeException Klasse

- Nachteil dieses Vorgehens: jede Methode, die Zahlen dividiert oder auf Arrays zugreift müsste berichten, dass sie Exceptions werfen könnte
- daher müssen alle Fehlerklassen, die von der Klasse RuntimeException abgeleitet sind, nicht berichtet werden
- Ableitungsbaum für Exceptionklassen

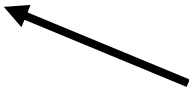


Go!

Eigene Fehler erzeugen (Fort.)

```
class MyException extends RuntimeException {  
}
```

MyException ist von
RuntimeException
abgeleitet worden



```
public class Exception15 {
```

```
    static int div(int i) {  
        if (i != 0)  
            return 42 / i;  
        else  
            throw new MyException();  
    }
```

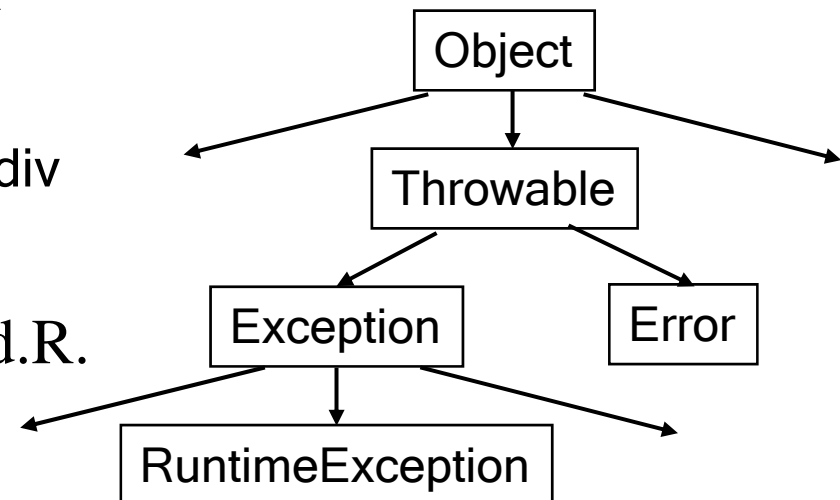
daher muss hier keine
throws Deklaration
vorkommen



```
    public static void main(String[] args) {  
        try {  
            System.out.println(div(0));  
        } catch (MyException x) {  
            System.out.println("bisschen zu 0 der Wert");  
        }  
    }  
}
```


Die RuntimeException Klasse (Fort.)

- in diesem Fall ist die Ableitung von RuntimeException ungeschickt, da es sich nicht um einen Laufzeitfehler handelt
- hier ist es ein Benutzerfehler: die div Methode wurde falsch benutzt
- eigene Fehlerklassen leitet man i.d.R. von Exception ab
- dann muss man sie natürlich auch berichten, d.h. throws Spezifikationen im Methodenkopf angeben



Eigene Fehler erzeugen (Fort.)

- in der Throwable Klasse sind die Methoden getMessage, toString und printStackTrace definiert
- diese werden von der eigenen Fehlerklasse geerbt
- Wie sieht ihre Anwendung aus?
- Was erbt man da?

Go!

Eigene Fehler erzeugen (Fort.)

```
class MyException extends Exception {  
}  
public class Exception16 {  
    static int div(int i) throws MyException {  
        if (i != 0)  
            return 42 / i;  
        else  
            throw new MyException();  
    }  
    public static void main(String[] args) {  
        try {  
            System.out.println(div(0));  
        } catch (MyException x) {  
            System.out.println("getMessage: " + x.getMessage());  
            System.out.println("toString: " + x.toString());  
            System.out.print("Stack: ");  
            x.printStackTrace();  
        }  
    }  
}
```

MyException ist von
Exception abgeleitet

daher muss hier eine
throws Deklaration geben

Informationen aus dem
Fehlerobjekt herausholen

Eigene Fehler erzeugen (Fort.)

- die Methode toString liefert den Klassennamen
- die Methode getMessage liefert nichts zurück
- die Methode printStackTrace funktioniert wie zuvor und druckt den Aufrufstack aus
- dem Fehlerobjekt in der Klasse Exception kann ein String im Konstruktor mitgegeben werden
- dieser String wird vom Fehlerobjekt gemerkt und bei getMessage zurückgeliefert

```
public class Exception extends Throwable {  
    public Exception(String msg);  
    public Exception();  
}
```

Go!

Eigene Fehler erzeugen (Fort.)

```
class MyException extends Exception {  
    MyException(String msg) {  
        super(msg);  
    }  
}
```

MyException erwartet einen String, der an Exception weitergegeben wird

```
public class Exception17 {  
    static int div(int i) throws MyException {  
        if (i != 0)  
            return 42 / i;  
        else  
            throw new MyException("durch 0 geteilt");  
    }  
}
```

MyException muss jetzt ein String mitgegeben werden

```
public static void main(String[] args) {  
    try {  
        System.out.println(div(0));  
    } catch (MyException x) {  
        System.out.println("getMessage: " + x.getMessage());  
        System.out.println("toString: " + x.toString());  
        System.out.print("Stack: ");  
        x.printStackTrace();  
    }  
}
```

jetzt gibt getMessage auch den übergebenen String zurück

Vorlesung 13/2

Streams

- für die sequentielle Verarbeitung von Zeichenfolgen gibt es 4 Klassenhierarchien
- es wird unterschieden zwischen:
 1. Zeichen versenden (Writer) und empfangen (Reader)
 2. Zeichen, die als Character oder als Byte kodiert sind

	Zeichen sind Character	Zeichen sind Byte
Zeichen lesen	class Reader FileReader (+8 Klassen)	class InputStream DataInputStream (+8 Klassen)
Zeichen schreiben	class Writer FileWriter PrintWriter (+6 Klassen)	class OutputStream DataOutputStream (+9 Klassen)

Go!

FileWriter: Beispiel

öffnet das
Java Paket io

```
import java.io.*;
```

die möglichen Fehler werden
nicht abgefangen sondern
einfach an den Java-
Interpreter weitergegeben

```
public class Stream1 {
```

```
    public static void main(String[] args) throws IOException {
```

```
        FileWriter f = new FileWriter("hello.txt");
```

```
        f.write("juhu\n");
```

```
        f.write("hello world");
```

```
        f.write("das sollte jetzt angehängt werden : ");
```

```
        f.write(64);
```

```
        f.close();
```

```
    }
```

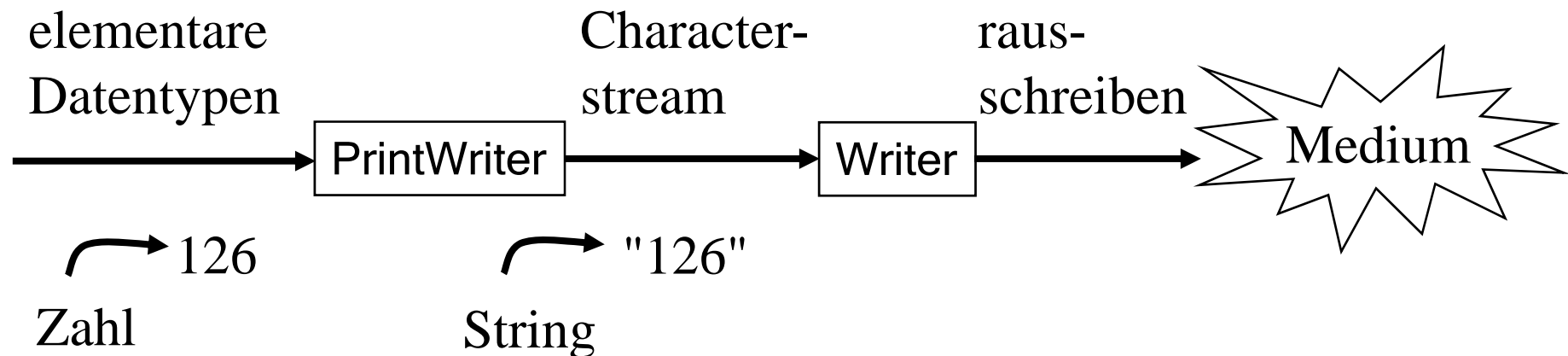
```
}
```

öffnet die Datei
hello.txt und löscht
den Inhalt

schreibt 3 Strings und einen
Integer Wert in die Datei und
schließt sie anschließend

PrintWriter

- ist ein *Medium*, dass die *elementaren Datentypen in Strings verwandelt* und diese dann weiterleitet
- da die Verwandlung zunächst *nichts mit Dateien* zu tun hat, ist der PrintWriter auch *unabhängig von FileWriter*
- der PrintWriter muss vielmehr an einen *anderen Writer gebunden* werden



PrintWriter: Beispiel

```
import java.io.*;
```

```
public class Stream3 {
```

```
    public static void main(String[] args) throws IOException {
```

```
        FileWriter f = new FileWriter("hello.txt");
```

```
        PrintWriter p = new PrintWriter(f);
```

```
        p.print("die erste Zeile\n");
```

```
        p.print(64);
```

```
        p.print(true);
```

```
        p.close();
```

```
    }
```

```
}
```

öffnet die
Datei hello.txt

erzeugt einen neuen
PrintWriter, der mit der
Datei hello.txt verknüpft ist

es wird nur noch
auf den PrintWriter
geschrieben

nicht das Schließen der Datei vergessen

Go!

Beispiel

```
import java.io.*;
```

```
public class Reader1 {
```

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

```
        try {
```

```
            FileReader f = new FileReader("Reader1.java");
```

```
            int c;
```

```
            while ((c = f.read()) != -1) {
```

```
                System.out.print((char) c);
```

```
            }
```

```
            f.close();
```

```
        } catch (FileNotFoundException x) {
```

```
            System.out.println("Datei nicht gefunden");
```

```
        } catch (IOException x) {
```

```
            System.out.println("Das Lesen hat nicht funktioniert");
```

```
        }
```

```
    }
```

```
}
```

öffnet die Datei
"Reader1.java"

liest die Datei
zeichenweise ein

kann vom
Konstruktor
geworfen werden

kann von read geworfen werden

DataOutputStream

- besonders interessant ist die Klasse `DataOutputStream`
- mit der zugehörigen Klasse `DataInputStream` können primitive Datentypen geschrieben und wieder gelesen werden
- dazu verfügt die Klasse über viele **write**-Methoden

```
class DataOutputStream extends OutputStream {  
    public DataOutputStream(OutputStream out);  
    public void writeInt(int i) throws IOException;  
    public void writeBoolean(boolean b) throws IOException;  
    public void writeFloat(float f) throws IOException;  
    public void writeChars(String str) throws IOException;  
    public void writeUTF(String str) throws IOException;  
    ...  
}
```

merkt sich das Ziel

existieren für
alle elementaren
Datentypen

schreibt jedes Zeichen von str
in 1, 2 oder 3 Bytes hinaus

Go!

Bytestreams

Beispiel

```
import java.io.*;
```

```
public class DataOutput1 {
```

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

```
        try {
```

```
            DataOutputStream out =
```

```
                new DataOutputStream(new FileOutputStream("hello.txt"));
```

```
            out.writeInt(-1);
```

```
            out.writeInt(1);
```

```
            out.writeDouble(0.23657423);
```

```
            out.writeUTF("Dies ist ein Test auch für Umlaute");
```

```
            out.close();
```

```
        } catch (IOException x) {
```

```
            System.out.println("irgendwas hat nicht funktioniert");
```

```
        }
```

```
    }
```

```
}
```

öffnet die Datei
"hello.txt" zum
Schreiben von Bytes

die Standardtypen
werden formatiert
ausgegeben

notwendig wegen
des Umlauts

DataInputStream

- die von der Klasse DataOutputStream geschriebenen Byte-Streams können mittels der DataInputStream Klasse gelesen werden
- dazu muss jedoch bekannt sein, welche Daten man lesen möchte, d.h. sind erst 2 Integer und dann 3 Floats geschrieben worden, müssen auch erst 2 Integer und dann 3 Floats gelesen werden
- dazu verfügt die Klasse über viele **read-Methoden**

```
class DataInputStream extends InputStream {  
    public DataInputStream(InputStream in);  
    public boolean readBoolean() throws IOException;  
    public int readInt() throws IOException;  
    public String readUTF() throws IOException;  
    ...  
}
```

merkt sich die
Herkunft der Bytes

für alle elementaren
Datentypen existieren die
Einlesemethoden

Go!

Bytestreams

Beispiel (Fort.)

```
import java.io.*;
```

```
public class DataInput1 {
```

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

```
        try {
```

```
            DataInputStream in =
```

```
                new DataInputStream(new FileInputStream("hello.txt"));
```

```
            System.out.println(in.readInt());
```

```
            System.out.println(in.readInt());
```

```
            System.out.println(in.readDouble());
```

```
            System.out.println(in.readUTF());
```

```
            in.close();
```

```
        } catch (IOException x) {
```

```
            System.out.println("irgendwas hat nicht funktioniert");
```

```
        }
```

```
    }
```

```
}
```

öffnet die Datei
"hello.txt" zum Lesen
von Bytes

die Standardtypen
werden formatiert
eingelesen

erst 2 Integer, dann ein
Double und dann ein UTF-
kodierten String einlesen

Beispiel (Fort.)

- hält man sich nicht an die Reihenfolge, die man beim Rausschreiben gewählt hatte, können
 - Werte falsch eingelesen werden
 - folgende Einleseversuche scheitern

```
System.out.println(in.readInt());  
System.out.println(in.readInt());  
System.out.println(in.readFloat());  
System.out.println(in.readUTF());  
in.close();  
} catch (IOException x) {  
    System.out.println("irgendwas hat nicht funktioniert");  
    System.out.println(x);  
}
```

statt einem Double wird
ein Float eingelesen

ursprünglich wurde der Double
0.23657423 geschrieben

Die Klasse System

- die Klasse **System** besitzt die 3 Klassenvariablen `err`, `out` und `in`
- `err` und `out` sind vom Typ `PrintStream`, d.h. Ausgabestreams für Zeichen, bei denen die elementaren Datentypen in textueller Form rausgeschrieben werden (i.A. mit der Konsole verbunden)
- `in` ist vom Typ `InputStream`, d.h. es können Zeichen in nichtformatierter Form eingelesen werden (i.A. mit der Tastatur verbunden)

```
public final class System extends Object {  
    public static final PrintStream err;  
    public static final PrintStream out;  
    public static InputStream in;  
    ...  
}
```

Go!

Einlesen von Zahlen (Fort.)

```
public class ReadInt {  
    public static int readInt() throws IOException, NumberFormatException {  
        BufferedReader s = new BufferedReader(new InputStreamReader(System.in));  
        return Integer.parseInt(s.readLine());  
    }  
    public static double readDouble() throws IOException, NumberFormatException {  
        BufferedReader s = new BufferedReader(new InputStreamReader(System.in));  
        return Double.parseDouble(s.readLine());  
    }  
    public static void main(String[] args) {  
        try {  
            System.out.println(readInt(System.in));  
            System.out.println(readDouble(System.in));  
        } catch (Exception x) {  
            System.out.println("irgendwas hat nicht funktioniert");  
            System.out.println(x);  
        }  
    }  
}
```

- Auf System.in einen
InputStreamReader bilden
- ...
- ...
- darauf einen BufferedReader
- hiervon eine Zeile einlesen

Weiteres zu File Handling

- alle bisherigen Klassen dienten zum sequentiellen Zugriff auf Dateien
- für beliebigen Zugriff gibt es die `RandomAccessFile` Klasse
- um Informationen über Dateien und Verzeichnisse zu erhalten wird die Klasse `File` verwendet
- die Klasse `File` hat 3 verschiedene Konstruktoren
 - `public File(String pathname);`
 - `public File(String parent, String child);`
 - `public File(File parent, String child);`
- `pathname` gibt den Datei- oder Verzeichnisnamen an
- `parent` und `child` trennt den Datei- von dem Verzeichnisnamen

Go!

Beispiel

```
import java.io.*;
```

```
public class File2 {
```

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

```
        File f1 = new File("File1.java");
```

```
        File f2 = new File(".");
```

```
        File f3 = new File(f2, "File2.java");
```

```
        File f4 = new File("juhu");
```

```
        System.out.println(f1.getName() + "\texists: " + f1.exists());
```

```
        System.out.println(f4.getName() + "\texists: " + f4.exists());
```

```
        System.out.println(f1.getName() + "\tcan be written: " + f1.canWrite());
```

```
        System.out.println(f1.getName() + "\tcan be read: " + f1.canRead());
```

```
        System.out.println(f1.getName() + "\tis a file: " + f1.isFile());
```

```
        System.out.println(f1.getName() + "\tis a directory: " + f1.isDirectory());
```

```
        System.out.println(f2.getName() + "\texists: " + f2.exists());
```

```
        System.out.println(f2.getName() + "\tis a file: " + f2.isFile());
```

```
        System.out.println(f2.getName() + "\tis a directory: " + f2.isDirectory());
```

```
    }
```

```
}
```

kann f1 gelesen
und beschrieben
werden

überprüft, ob die
Dateien existieren


File oder Directory?

Lesen von Verzeichnisse

- bezeichnet ein File Objekt ein Verzeichnis, kann man mit der Methode list() auf alle Einträge in dem Verzeichnis zugreifen
- zusätzlich gibt es die Klassenmethode listRoots(), die alle Wurzeln zurückliefert
- unter Unix ist das genau eine Wurzel "/"
- unter Windows sind es i.d.R. mehrere Wurzeln "a:\", "c:\", ...

```
public class File {  
    ...  
    public String[] list();  
    public static File[] listRoots();  
    ...  
}
```

liefert null zurück, wenn File
eine Datei ist oder nicht existiert



Go!

Beispiel

```
import java.io.*;
public class File3 {
    public static void printStrings(String[] list) {
        if (list != null) {
            for(int i = 0; i < list.length; ++i)
                System.out.println(list[i]);
        }
    }

    public static void main(String[] args) {
        File f1 = new File(".");
        File f2 = new File("File3.java");
        System.out.println("\t\tEintraege im aktuellen Verzeichnis:");
        printStrings(f1.list());
        System.out.println("\t\tEintraege in der Datei File3.java:");
        printStrings(f2.list());
        System.out.println("\t\tWurzeln:");
        File[] roots = File.listRoots();
        for(int i = 0; i < roots.length; ++i)
            System.out.println(roots[i].getAbsolutePath());
    }
}
```

druckt die übergebene
Liste von Strings

aktuelles Verzeichnis

Datei

die Wurzeln sind selber
wieder File Objekte

Strings

- Klasse zur Darstellung und Ver-/Bearbeitung von Zeichenfolgen
- Erzeugung: `String s = new String(„Juhu“)`
- Methode zur Längenermittlung: `s.length()`
- Zugriff: `s.charAt(3)`
- WICHTIG: Strings ändern **niemals** ihren Wert (sprich Zeichen)
- kleiner Auszug aus der Methodenvielfalt:
 - `public String substring(int begin, int end);`
 - `public int compareTo(String s); /* Vergleich */`
 - `public int indexOf(String s); /* s kann auch vom Typ char sein */`
 - `public int indexOf(String s, int fromIndex); /* dito */`
 - `public int lastIndexOf(String s); /* dito */`
- Konkatination: `s + „toll“ + „hello“ + s`

Vergleich von Strings (Fort.)

- die Methode `compareTo` führt einen *lexikalischen Vergleich* zwischen 2 Strings durch
- der Rückgabewert wird unterschieden zwischen
 - *negativer Wert* \Rightarrow der *aktuelle String ist kleiner* als der übergebene String
 - *positiver Wert* \Rightarrow der *aktuelle String ist größer* als der übergebene String
 - *0* \Rightarrow die beiden Strings sind *gleich*

Go!

Beispiel

```
public class String6 {  
  
    public static void main(String[] args) {  
        String s1 = new String("Mayer");  
        String s2 = new String("Maier");  
        String s4 = new String("Schmidt");  
        String s3 = new String("Anton der Erste");  
        if (s3.startsWith("Ant"))  
            System.out.println("In Anton steckt eine Ameise");  
        if (s3.endsWith("e"))  
            System.out.println("die mit 'e' aufhoert");  
        System.out.println(s1.compareTo(s2));  
        System.out.println(s1.compareTo(s3));  
        System.out.println(s1.compareTo(s4));  
    }  
}
```

Im Telefonbuch:

1. Anton der Erste
2. Maier
3. Mayer
4. Schmidt

Go!

Beispiel

```
public class String8 {
```

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

```
        String s = new String("Anton der Erste, der Puritaner");
```

```
        System.out.println(s.indexOf("der"));
```

```
        System.out.println(s.indexOf("der",12));
```

```
        System.out.println(s.indexOf("der",s.lastIndexOf("der") + 1));
```

```
        System.out.println(s.substring(s.lastIndexOf('d'),s.length()));
```

```
    }
```

```
}
```

sucht den Teilstring
"der" von Anfang ...

kann ja nicht
funktionieren

... bzw. ab der
Position 12

Die Klasse StringBuffer

- Objekte der String Klasse können nicht verändert werden
- einmal erzeugt ist ihr Wert konstant
- sollen Strings während ihrer Lebensdauer verändert werden, muss die Klasse StringBuffer verwendet werden

```
public class StringBuffer {  
    public StringBuffer();  
    public StringBuffer(String s);  
    public StringBuffer append(String s);  
    public StringBuffer insert(int offset, String s);  
    public StringBuffer deleteCharAt(int index);  
    public StringBuffer delete(int start, int end);  
    public StringBuffer replace(int start, int end, String str);  
    public void setCharAt(int index, char c)  
        throws StringIndexOutOfBoundsException;  
    public int length();  
    ...  
}
```

Konstruktoren

verändern alle das Objekt und liefern das veränderte Objekt zurück

Go!

Beispiel

```
public class String11 {
```

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

```
        StringBuffer s1 = new StringBuffer("Hello World");
```

```
        System.out.println(s1);
```

```
        s1.append(": juhu");
```

anhängen

```
        System.out.println(s1);
```

```
        s1.insert(5, " old");
```

einfügen

```
        System.out.println(s1);
```

```
        s1.deleteCharAt(8);
```

einzelnen Zeichen löschen

```
        System.out.println(s1);
```

```
        s1.delete(6,9);
```

Bereich löschen

```
        System.out.println(s1);
```

```
        s1.setCharAt(6, 'w');
```

Zeichen ersetzen

```
        System.out.println(s1);
```

```
        s1.replace(5,6, " dies alles fuer ein LeerZeichen");
```

Bereich ersetzen

```
        System.out.println(s1);
```

```
    }
```

```
}
```

Vorlesung 14/1

Go!

```
class Pair extends Object {  
    int i;  
    char c;  
    Pair(int i,char c) {  
        this.i = i;  
        this.c = c;  
    }  
    public String toString() {  
        return i + " " + c;  
    }  
}
```

Beispiel 18

Frage: Was macht das Programm?

```
class Beispiel18 {  
    int value;  
    public Beispiel18() {  
        value = 42;  
    }  
    public Pair getValue(char ch) {  
        return new Pair(value,ch);  
    }  
    public static void main(String[] args) {  
        Beispiel18 b = new Beispiel18();  
        System.out.println(b.getValue('?'));  
    }  
}
```

Frage: Muss die Klasse Pair an dieser Stelle deklariert sein?

```

class Pair extends Object {
    int i;
    char c;
    Pair(int i,char c) {
        this.i = i;
        this.c = c;
    }
    public String toString() {
        return i + " " + c;
    }
}
class Beispiel18 {
    int value;
    public Beispiel18() {
        value = 42;
    }
    public Pair getValue(char ch) {
        return new Pair(value,ch);
    }
    public static void main(String[] args) {
        Beispiel18 b = new Beispiel18();
        System.out.println(b.getValue('?'));
    }
}

```

Innere Klassen

Frage: Muss die Klasse Pair an dieser Stelle deklariert sein?

Antwort: Nein, sie kann auch innerhalb von der Klasse Beispiel18 deklariert sein.

Go!

```
class Beispiel18 {  
    int value;  
    class Pair extends Object {  
        int i;  
        char c;  
        Pair(int i,char c) {  
            this.i = i;  
            this.c = c;  
        }  
        public String toString() {  
            return i + " " + c;  
        }  
    }  
    public Beispiel18() {  
        value = 42;  
    }  
    public Pair getValue(char ch) {  
        return new Pair(value,ch);  
    }  
    public static void main(String[] args) {  
        Beispiel18 b = new Beispiel18();  
        System.out.println(b.getValue('?'));  
    }  
}
```

Innere Klassen (Fort.)

Innere Klassen sind Klassen, die **innerhalb** einer anderen Klasse oder eines Blocks deklariert werden.

Innere Klassen (Fort.)

Es werden vier verschiedene Anwendungen von inneren Klassen unterschieden:

1. geschachtelte Top-Level Klassen
2. Elementklassen
3. Lokale Klassen
4. Anonyme Klassen

Geschachtelte Top-Level Klassen

```
class A {  
    static class B {  
        ...  
    }  
    ...  
}
```

Die Klasse B ist in A geschachtelt, aber dennoch auf der Topebene zu verwenden, da sie static in A deklariert ist.

```
class Beispiel20 {  
    ...  
    public static void main(String [] args) {  
        A.B b = new A.B();  
    }  
}
```

Elementklassen

```
class A {  
    class B {  
        ...  
    }  
    ...  
}
```

Die Klasse B ist in A geschachtelt und kann auf der Topebene **nicht** verwendet werden, da sie nicht static deklariert ist.

```
class Beispiel20 {  
    ...  
    public static void main(String [] args) {  
        A.B b = new A.B();  
    }  
}
```



Elementklassen (Fort.)

```
class Beispiel19 {  
    int value;  
    class Dummy extends Object {  
  
        public String toString() {  
            return Integer.toString(value);  
        }  
    }  
    public Beispiel19(int i) {  
        value = i;  
    }  
    public Dummy getValue() {  
        return new Dummy();  
    }  
    public static void main(String[] args) {  
        Beispiel19 b1 = new Beispiel19(23);  
        Beispiel19 b2 = new Beispiel19(42);  
        System.out.println(b1.getValue());  
        System.out.println(b2.getValue());  
    }  
}
```

Objekte von Elementklassen können auf die Objektvariablen ihrer umschließenden Klassen zugreifen.

Objekte von Elementklassen merken sich die Objekte ihrer Oberklassen, aus denen sie erzeugt werden.

Instanziierung der Elementklasse erfolgt aus der umschließenden Klasse heraus.

Go!

Elementklassen (Fort.)

```
class Beispiel19 {  
    int value;  
    class Dummy extends Object {  
  
        public String toString() {  
            return Integer.toString(value);  
        }  
    }  
    public Beispiel19(int i) {  
        value = i;  
    }  
    public Dummy getValue() {  
        return new Dummy();  
    }  
    public static void main(String[] args) {  
        Beispiel19 b1 = new Beispiel19(23);  
        Beispiel19 b2 = new Beispiel19(42);  
        System.out.println(b1.getValue());  
        System.out.println(b2.getValue());  
    }  
}
```

Was macht das Programm?

Lokale Klassen

```
class A {  
    void doit() {  
        class B {  
            ...  
        }  
  
        B b = new B();  
        ...  
    }  
  
    public void test(String [] args) {  
        B b = new B();  
    }  
  
    ...  
}
```

Die Klasse B ist in der Methode doit geschachtelt und kann nur innerhalb der Methode verwendet werden.



Anonyme Klassen

```
class Beispiel19 {  
    int value;  
    class Dummy extends Object {  
  
        public String toString() {  
            return Integer.toString(value);  
        }  
    }  
    public Beispiel19(int i) {  
        value = i;  
    }  
    public Dummy getValue() {  
        return new Dummy();  
    }  
    public static void main(String[] args) {  
        Beispiel19 b1 = new Beispiel19(23);  
        Beispiel19 b2 = new Beispiel19(42);  
        System.out.println(b1.getValue());  
        System.out.println(b2.getValue());  
    }  
}
```

Frage:

- Braucht man die Klasse Dummy eigentlich?
- Braucht man den Namen "Dummy"?
- Wo braucht man den Namen "Dummy"?

Anonyme Klassen (Fort.)

```
class Beispiel19 {  
    int value;  
    class Dummy extends Object {  
  
        public String toString() {  
            return Integer.toString(value);  
        }  
    }  
    public Beispiel19(int i) {  
        value = i;  
    }  
    public Dummy getValue() {  
        return new Dummy();  
    }  
    public static void main(String[] args) {  
        Beispiel19 b1 = new Beispiel19(23);  
        Beispiel19 b2 = new Beispiel19(42);  
        System.out.println(b1.getValue());  
        System.out.println(b2.getValue());  
    }  
}
```

Idee:

Deklariere die Klasse dort, wo sie einmal gebraucht wird. Dann muss die Klasse keinen Namen haben

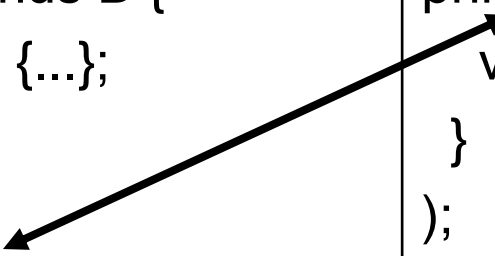
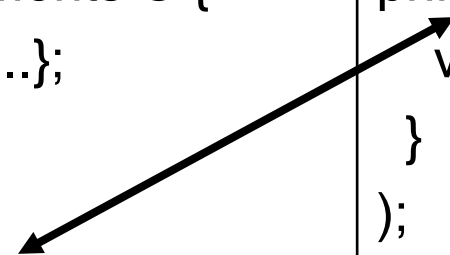
Go!

Anonyme Klassen (Fort.)

```
class Beispiel19 {  
    int value;  
    public Beispiel19(int i) {  
        value = i;  
    }  
    public Object getValue() {  
        return new Object() {  
            public String toString() {  
                return Integer.toString(value);  
            }  
        };  
    }  
    public static void main(String[] args) {  
        Beispiel19 b1 = new Beispiel19 (23);  
        Beispiel19 b2 = new Beispiel19 (42);  
        System.out.println(b1.getValue());  
        System.out.println(b2.getValue());  
    }  
}
```

- Anonyme Klassen können nur an genau einer Stelle instanziiert werden.

Anonyme Klassen und Interfaces

normale Klasse A	Anonyme Klasse
<pre>class A extends B { void doit() {...}; } ... print(new A());</pre>	<pre>print(new B() { void doit() {...}; }));</pre> 
<pre>class A implements C { void doit() {...}; } ... print(new A());</pre>	<pre>print(new C() { void doit() {...}; }));</pre> 

Innere Klassen: Zusammenfassung

- geschachtelte Top-Level Klassen

Klassen, die in einer anderen Klasse (mit **static**) deklariert werden, können auch woanders verwendet werden

- Elementklassen

Klassen, die in einer anderen Klasse (ohne **static**) deklariert werden, können nicht woanders verwendet werden, merken sich das umschließende Objekt

- Lokale Klassen (ist eine Elementklasse)

Klassen, die innerhalb eines Blocks definiert werden, können nur innerhalb des Blocks verwendet werden

- Anonyme Klassen (ist eine lokale Klasse, ist eine El.)

Lokale Klassen ohne Namen, können nur an der Stelle verwendet werden, an der sie deklariert werden

nochmal Sortieren

konzeptionelle Nachteile bisher vorgestellter Sortiervverfahren:

- Insertion-, Selection- und Bubblesort: langsam, sprich $O(n^2)$
- Distribution Counting: nur für eine spezielle Anwendung

Lösung: andere Algorithmen

implementationstechnische Nachteile der bisherigen Implementierungen:

- funktionieren nur für int-Arrays
- für Arrays anderen Typs müssen sie neu implementiert werden

Lösung: Generics und Interfaces

Go!

Generics: kurze Einführung

Generics: Parametrisierung von Klassen und Methoden in Typen

Aufgabe: Implementierung einer Klasse, die sich zwei beliebige Werte von beliebigen Typ (= Object) merkt

```
public class Pair1 {  
    private Object m_o1,m_o2;  
    public Pair1(Object o1,Object o2) {  
        m_o1 = o1;m_o2 = o2;  
    }  
    Object get1() {return m_o1;}  
    Object get2() {return m_o2;}  
  
    public static void main(String[] args) {  
        Pair1 p = new Pair1(2,'?');  
        System.out.println(p.get1());  
        char c      = (Character) p.get2();  
        double d    = (Double)   p.get1();  
        System.out.println(c + " " + d);  
    }  
}
```

Autoboxing: int → Integer
und char → Character

expliziter Typcast
mit Absturz

Go!

Generics: kurze Einführung (Forts.)

Klasse `Pair2` ist parametrisiert
in den Typen `T1` und `T2`

```
public class Pair2<T1,T2> {  
    private T1 m_o1;  
    private T2 m_o2;  
    public Pair2(T1 o1,T2 o2) {  
        m_o1 = o1;m_o2 = o2;  
    }  
    T1 get1() {return m_o1;}  
    T2 get2() {return m_o2;}  
}
```

- ab Version 1.5 verfügbar
- sehr schwache Imitation
von Templates (C++)

Members sind vom Typ `T1` bzw. `T2`

Konstruktor erwartet die
beiden Typen `T1` und `T2`

```
public static void main(String[] args) {  
    Pair2<Integer,Character> p = new Pair2<Integer,Character>(2,'?');  
    System.out.println(p.get1());  
    char c = p.get2();  
    double d = p.get1();  
    System.out.println(c + " " + d);  
}
```

Bei Instanziierung:
Festlegung der Typparameter

Autounboxing: `Character` → `char`,
`Integer` → `int` → `double`

Generics: kurze Einführung (Beispiel)

- eine Methode, die das Minimum zweier Werte beliebigen Typs ermittelt

```
static <K> K min(K a1, K a2) {  
    if (a1 < a2)  
        return a1;  
    else  
        return a2;  
}
```

zwei Werte a1 und a2
eines beliebigen Typs K

Rückgabe ist natürlich
auch vom Typ K

Problem: <-Operator ist auf einen
beliebigen Typen nicht definiert

Generics: kurze Einführung (Beispiel)

- Lösung: zusätzlich ein Interface mitgeben, das beschreibt, wie zwei Werte vom Typ K verglichen werden

```
interface Compare<T> {  
    boolean isLess(T a1,T a2);  
}
```

Interface mit einer Methode
isLess, die besagt, ob **a1** vom
beliebigen Typ **T** kleiner als **a2** ist

```
static <K> K min(K a1,K a2,Compare<K> c) {  
    if (c.isLess(a1,a2))  
        return a1;  
    else  
        return a2;  
}
```

die Methode **min** ist von
beliebigen Typ **K**. Das
Compare Interface soll
genau diesen Typ verwenden.

isLess Methode des Interfaces
Compare ersetzt den **<-Operator**

Go!

Generics: kurze Einführung (Beispiel)

- Für den Aufruf von min muss zunächst eine Klasse, die das Interface Compare implementiert, definiert werden

```
interface Compare<K> {  
    boolean isLess(K a1,K a2);  
}  
class MyCompare implements Compare<Integer> {  
    public boolean isLess(Integer a1,Integer a2) {  
        return a1<a2;  
    }  
}  
public class Sorted2 {  
    static <K> K min(K a1,K a2,Compare<K> c) {  
        if (c.isLess(a1,a2))  
            return a1;  
        else  
            return a2;  
    }  
    public static void main(String[] args) {  
        System.out.println(min(3,4,new MyCompare()));  
    }  
}
```

MyCompare implementiert
den Vergleich für **Integer**
Objekte (und nur für solche)

Beim Aufruf der min Methode
muss ein **MyCompare** Objekt
übergeben werden

Go!

Generics: kurze Einführung (Diskussion)

- Der Aufruf von min ist ok
- Die Definition der Klasse **MyCompare** wirkt fehl am Platz
- hier könnte man eine anonyme Klasse verwenden

```
interface Compare<K> { ... }
```

```
public class Sorted5 {  
    static <K> K min(K a1,K a2,Compare<K> c) { ... }
```

```
    public static void main(String[] args) {  
        System.out.println(min (3,4,
```

anonyme Klasse, die das
Compare<Integer>
Interface implementiert

```
new Compare<Integer>() {  
    public boolean isLess(Integer a1,Integer a2) {  
        return a1<a2;  
    }  
}
```

```
    );  
}
```

Generics: kurze Einführung (Diskussion)

- Viel Schreibaufwand, anonyme Klasse könnte auch vom Compiler generiert werden
- nur `a1 < a2` ist neu und kann nicht vom Compiler generiert werden
- Lösung hierzu: Lambda Ausdrücke

```
public static void main(String[] args) {  
    System.out.println(min (3,4,
```

könnte vom Compiler
erzeugt werden

```
new Compare<Integer>() {  
    public boolean isLess(Integer a1,Integer a2) {  
        return a1 < a2;  
    }  
}
```

muss explizit
programmiert werden

```
);  
}
```

Lamda Ausdrücke

Problem mit der Klasse **MyCompare** und anonymen Klasse:

- sehr viel Schreibarbeit
- die komplette Deklaration der Klasse könnte der Compiler selber schreiben
- die einzige Information, die der Compiler nicht kennt, ist die Anwendung des <-Operators, also der Methodenrumpf
- daher ab Java 1.8: Lamda Ausdrücke (Begriff aus der funktionalen Programmierung)
- nur noch der Inhalt der Funktion muss implementiert werden
- nicht mehr implementiert werden muss:
 - Deklaration der Klasse
 - Deklaration der überlagerten Methode

Lamda Ausdrücke (Forts.)

- Lambda Ausdrücke funktionieren nur bei Interfaces, die genau eine Methode enthalten
- sie funktionieren *nicht* bei
 - Interfaces mit mehreren Methoden
 - abstrakten Klassen
 - normale Klassen
- solche Interfaces heißen „funktionale Interfaces“ (sie spezifizieren im Wesentlichen eine Funktion)

Go!

Lamda Ausdrücke (Forts.)


- Lambda Ausdrücke: ein näherer Blick

```
interface Juhu {  
    public void doit();  
}
```

```
public class Lambda1 {
```

```
    public static void main(String[] args) {  
        Juhu j1 = new Juhu() {  
            public void doit() {  
                System.out.println("dies ist der alte Weg");  
            }  
        };  
        Juhu j2 = () -> System.out.println("mit Lambda Ausdruck");  
  
        j1.doit();  
        j2.doit();  
    }  
}
```

ohne Parameter müssen leere
Klammern gesetzt werden



Go!

Lamda Ausdrücke (Forts.)

- die Methoden können auch mehrere Parameter enthalten

```
interface Juhu {  
    public void doit(int i,float f);  
}
```

```
public class Lambda2 {  
  
    public static void main(String[] args) {  
        Juhu j1 = new Juhu() {  
            public void doit(int i,float f) {  
                System.out.println("old school: i = " + i + " f = " + f);  
            }  
        };  
        Juhu j2 = (i,f) -> System.out.println("Lambda: i = " + i + " f = " + f);  
  
        j1.doit(12,34.6f);  
        j2.doit(5,7.8f);  
    }  
}
```

mehrere Parameter müssen
auch in Klammern gesetzt
werden

Go!

Lamda Ausdrücke (Forts.)

- mehrere Statements müssen in einem Block zusammengefasst werden

```
interface Juhu {  
    public void doit(int i,int j);  
}
```

```
public class Lambda4 {
```

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

```
        Juhu j = (i1,i2) -> {
```

```
            System.out.println("jetzt kommt " + i1 + " mal die " + i2);
```

```
            for(int i = 0; i < i1; ++i)
```

```
                System.out.println(i2);
```

```
        };
```

```
        j.doit(10,13);
```

```
    }
```

```
}
```

Block von hier bis hier

Go!

Lamda Ausdrücke (Forts.)

- die Methoden können auch einen Rückgabewert haben

```
interface Juhu {  
    public int doit(int i,int j);  
}
```

```
public class Lambda3 {
```

```
    public static void main(String[] args) {  
        Juhu j1 = (i1,i2) -> {return i1*i2;};  
        Juhu j2 = (x,y) -> x / y;
```

```
        int a = j1.doit(12,10);  
        int b = j2.doit(12,5);  
    }  
}
```

obwohl nur ein Statement (**return**)
muss es dennoch im Block stehen

Spezialfall „Lambda Ausdrücke“:
das **return** kann weggelassen
werden, dann auch ohne Block

Go!

Lamda Ausdrücke (Forts.)


- das min-Beispiel mit Lamda Ausdrücken:

```
interface Compare<K> {  
    boolean isLess(K a1,K a2);  
}
```

Die Klasse MyCompare
fehlt komplett

```
public class Sorted3 {  
    static <K> K min(K a1,K a2,Compare<K> c) {  
        if (c.isLess(a1,a2))  
            return a1;  
        else  
            return a2;  
    }  
    public static void main(String[] args) {  
        System.out.println(min(3,4,(a1,a2) -> a1<a2));  
    }  
}
```

Der Lambda Ausdruck, der die
MyCompare Definition und
die Objekterzeugung ersetzt



Go!

Lamda Ausdrücke (Forts.)

- großer Vorteil: soll die min-Methode nicht mehr das Minimum sondern das Maximum berechnen, muss nur der <-Operator durch den >-Operator ersetzt werden mit Lamda Ausdrücken:

```
interface Compare<K> {  
    boolean isLess(K a1,K a2);  
}  
  
public class Sorted4 {  
    static <K> K min(K a1,K a2,Compare<K> c) {  
        if (c.isLess(a1,a2))  
            return a1;  
        else  
            return a2;  
    }  
    public static void main(String[] args) {  
        System.out.println(min(3,4,(a1,a2) -> a1<a2));  
        System.out.println(min(3,4,(a1,a2) -> a1>a2));  
    }  
}
```

Berechnet Minimum

Berechnet Maximum

Vorlesung 14/2

Sortieren (die Rückkehr)

Zur Erinnerung: Insertion Sort

```
static void insertion_sort(int []field) {  
    for(int i1 = 1; i1 < field.length; ++i1) {  
        final int IVAL = field[i1];  
        int i2 = i1;  
        while (i2 >= 1 && field[i2 - 1] > IVAL) {  
            field[i2] = field[i2 - 1];  
            i2 = i2 - 1;  
        }  
        field[i2] = IVAL;  
    }  
}
```

der Datentyp int muss an
diesen beiden Stellen
geändert werden

der Vergleichsoperator
muss an dieser Stelle
verändert werden

Insertion Sort mit Generics und Lambda

```
static <K> void insertion_sort(K[] field, Compare<K> c) {  
    for(int i1 = 1; i1 < field.length; ++i1) {  
        final K IVAL = field[i1];  
        int i2 = i1;  
        while (i2 >= 1 && c.isLess(IVAL, field[i2 - 1])) {  
            field[i2] = field[i2 - 1];  
            i2 = i2 - 1;  
        }  
        field[i2] = IVAL;  
    }  
}
```

...

```
Integer[] f = {5,3,4,-2,0,-17};
```

```
insertion_sort(f, (x,y) -> x<y);
```

```
insertion_sort(f, (x,y) -> x>y);
```

aufsteigend sortieren

absteigend sortieren

Insertion Sort: eine genauere Beobachtung

```
static <K> void insertion_sort(K[] field, Compare<K> c) {  
    for(int i1 = 1; i1 < field.length; ++i1) {  
        final K IVAL = field[i1];  
        int i2 = i1;  
        while (i2 >= 1 && c.isLess(IVAL, field[i2 - 1])) {  
            field[i2] = field[i2 - 1];  
            i2 = i2 - 1;  
        }  
        field[i2] = IVAL;  
    }  
}
```

← gehe alle bis auf das 1. Element durch

← füge sie vorne sortiert ein

Nachteil:

- ein Element kann immer nur 1 Schritt aufrücken
- dadurch dauert es sehr lange, bis kleine Elemente von hinten nach vorne kommen
- Ziel: das muss schneller gehen

Shellsort

Idee:

- basierend auf Insertion Sort
- vergleiche nicht unmittelbar benachbarte Elemente, sondern nehme welche mit großem Abstand und vergleiche diese
- wiederhole das Vorgehen mit kleinerem Abstand
- wenn der Abstand einmal 1 ist, ist es der normale Insertion Sort und damit ist die Folge *danach* sortiert

Beispiel

712	452	307	145	102	50	12	7	3	1
-----	-----	-----	-----	-----	----	----	---	---	---

schlimmster Fall für Insertion Sort:

- invertiert sortierte Liste

Idee bei Shellsort:

- betrachte Teillisten, bei der die Nachbarn nur jedes 4. Element sind und sortiere sie nach Insertion Sort, d.h.

712	452	307	145	102	50	12	7	3	1
-----	-----	-----	-----	-----	----	----	---	---	---

712	452	307	145	102	50	12	7	3	1
-----	-----	-----	-----	-----	----	----	---	---	---

712	452	307	145	102	50	12	7	3	1
-----	-----	-----	-----	-----	----	----	---	---	---

712	452	307	145	102	50	12	7	3	1
-----	-----	-----	-----	-----	----	----	---	---	---

Beispiel (Fort.)

712	452	307	145	102	50	12	7	3	1
-----	-----	-----	-----	-----	----	----	---	---	---

712	452	307	145	102	50	12	7	3	1
-----	-----	-----	-----	-----	----	----	---	---	---

712	452	307	145	102	50	12	7	3	1
-----	-----	-----	-----	-----	----	----	---	---	---

712	452	307	145	102	50	12	7	3	1
-----	-----	-----	-----	-----	----	----	---	---	---

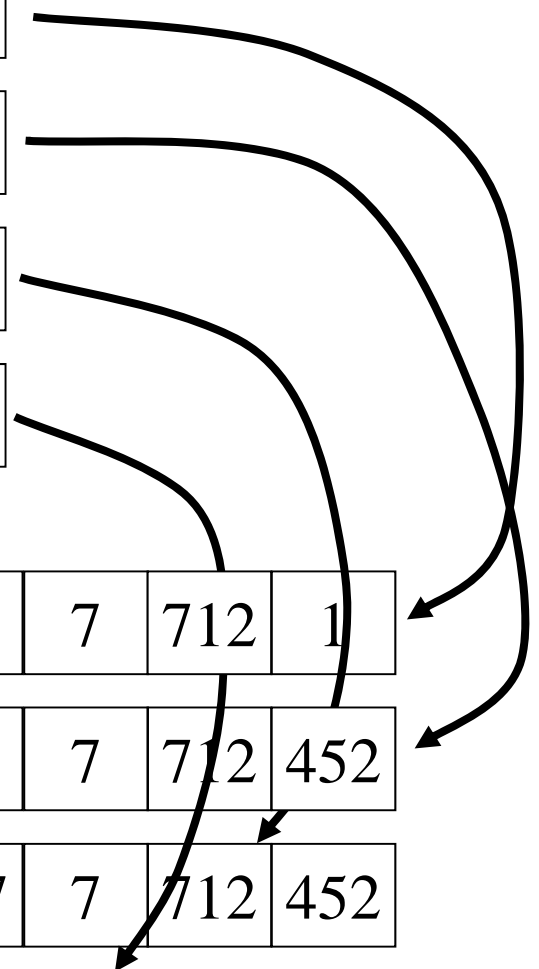
⇒

3	452	307	145	102	50	12	7	712	1
---	-----	-----	-----	-----	----	----	---	-----	---

3	1	307	145	102	50	12	7	712	452
---	---	-----	-----	-----	----	----	---	-----	-----

3	1	12	145	102	50	307	7	712	452
---	---	----	-----	-----	----	-----	---	-----	-----

3	1	12	7	102	50	307	145	712	452
---	---	----	---	-----	----	-----	-----	-----	-----



Beispiel (Fort.)

Das Ergebnis der 1. Durchgangs mit 4er Abstand:

3	1	12	7	102	50	307	145	712	452
---	---	----	---	-----	----	-----	-----	-----	-----

Beobachtung:

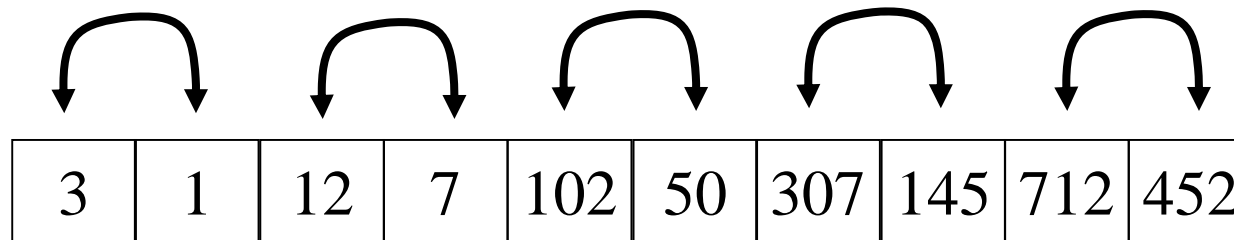
- diese Liste ist wesentlich sortierter, als die Anfangsliste
- die kleinen Elemente sind von rechts nach links gewandert
- die großen Elemente sind von links nach rechts gewandert
- es fanden nur wenige Austausche statt

Nächster Schritt:

- mit Abstand 1 wiederholen, d.h. normalen Insertion Sort

Beispiel (Fort.)

Normaler Insertion Sort:



Ergebnis nach einem Durchlauf:

1	3	7	12	50	102	145	307	452	712
---	---	---	----	----	-----	-----	-----	-----	-----

- die Liste ist fertig sortiert
- es musste nur jeweils 1 Element verschoben werden, d.h. hier direkter Tausch war möglich

Shell Sort: Abstände

- In diesem Beispiel wurden 2 Abstände gewählt: 4 und 1
- Welche Abstände sollte man im allgemeinen wählen?
 - Bsp.: ..., 1093, 364, 121, 40, 13, 4, 1
..., 64, 32, 16, 8, 4, 2, 1
- Welche dieser Folgen ist besser?

Ziel:

- eine gute Durchmischung der Vergleiche
- in verschiedenen Durchläufen sollen verschiedene Elemente verglichen werden

Shell Sort: Abstände (Fort.)

- die Wahl der richtigen Abstände ist ganz entscheidend für das Laufzeitverhalten
- es gibt *keine eindeutig richtige* Wahl für die Abstände
- es gibt *aber eindeutig falsche* Wahlen für Abstände, z.B. 64, 32, 16, 8, 4, 2, 1 da hier immer die gleichen Elemente miteinander verglichen werden
- also: die Folge sollte möglichst ungleichmäßig sein
- somit werden verschiedene Elemente in den verschiedenen Durchläufen miteinander verglichen

Shell Sort: Implementierung

- für den Abstand wird die Variable iDist für Distanz eingeführt
- statt des unmittelbaren Nachbarn wird der Nachbar genommen, der iDist entfernt liegt
- es wird nicht mit dem 1. Element angefangen, sondern mit dem iDist

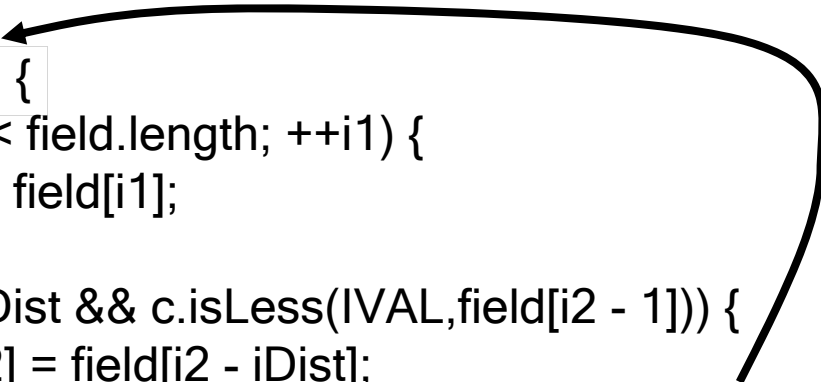
```
static <K> void insertion_sort(K[] field, Compare<K> c) {  
    for(int i1 = 1; i1 < field.length; ++i1) {  
        final K IVAL = field[i1];  
        int i2 = i1;  
        while (i2 >= 1 && c.isLess(IVAL, field[i2 - 1])) {  
            field[i2] = field[i2 - 1];  
            i2 = i2 - 1;  
        }  
        field[i2] = IVAL;  
    }  
}
```

diese Stellen müssen
geändert werden

Shell Sort: Implementierung (Fort.)

- bisher läuft der Algorithmus einmal über das Feld mit dem Abstand iDist
- iDist muss jetzt noch verringert werden, der Algorithmus muss erneut laufen

```
static <K> void shell_sort(K[] field, Compare<K> c) {  
...  
    for( ; iDist > 0; iDist /= 3) {  
        for(int i1 = iDist; i1 < field.length; ++i1) {  
            final K IVAL = field[i1];  
            int i2 = i1;  
            while (i2 >= iDist && c.isLess(IVAL, field[i2 - 1])) {  
                field[i2] = field[i2 - iDist];  
                i2 = i2 - iDist;  
            }  
            field[i2] = IVAL;  
        }  
    }  
}
```



der Abstand wird nach
jedem Durchlauf auf
ein Drittel reduziert

Shell Sort: Implementierung (Fort.)

- Frage: mit welchem Abstand wird angefangen?

```
static <K> void shell_sort(K[] field, Compare<K> c) {  
    int iDist = 1;  
    for( ; iDist <= field.length / 9; iDist = 3 * iDist + 1) {  
    }  
    for( ; iDist > 0; iDist /= 3) {  
        for(int i1 = iDist; i1 < field.length; ++i1) {  
            final K IVAL = field[i1];  
            int i2 = i1;  
            while (i2 >= iDist && c.isLess(IVAL, field[i2 - 1])) {  
                field[i2] = field[i2 - iDist];  
                i2 = i2 - iDist;  
            }  
            field[i2] = IVAL;  
        }  
    }  
}
```

Vorsicht:
leere Schleife

im 1. Durchlauf sollen
maximal 9 Elemente
miteinander verglichen
werden

Shell Sort: Analyse

- bisher ist unklar, wie schnell Shell Sort arbeitet
- die Geschwindigkeit hängt sehr stark von der Folge der Abstände ab
- die Güte der Abstände hängt aber wiederum von der Vorsortierung ab
- in der Praxis läuft dieser Algorithmus *sehr gut*
- er ist sehr einfach zu implementieren

Fragen:

1. Ist Shell Sort stabil?
2. Ist Shell Sort für externes Sortieren geeignet?

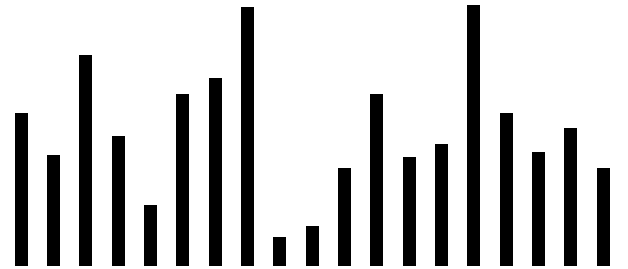
Quicksort

Idee:

- eine Folge mit nur einem Element ist immer (trivialerweise) sortiert
- hat man mehr Elemente, die zu sortieren sind, teilt man das Problem auf
 - in eine Gruppe kommen alle großen Elemente
 - in eine Gruppe kommen alle kleinen Elemente
 - sortiere die beiden Gruppen jede für sich
 - danach ist die gesamte Folge sortiert

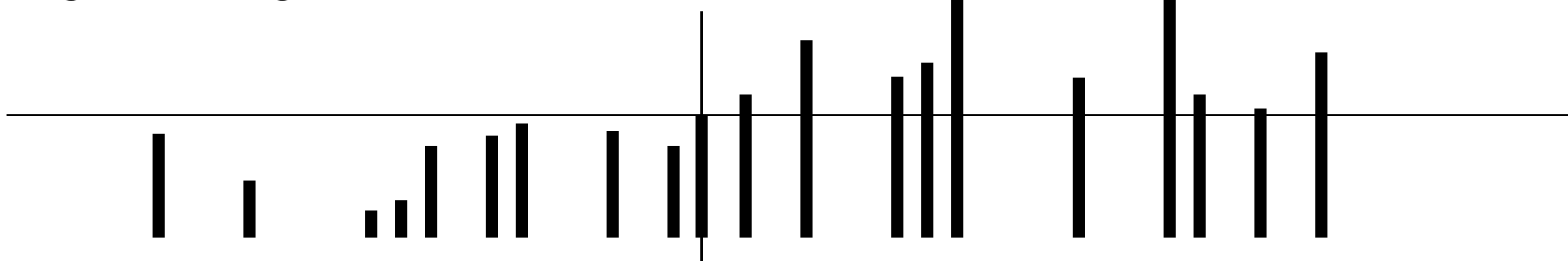
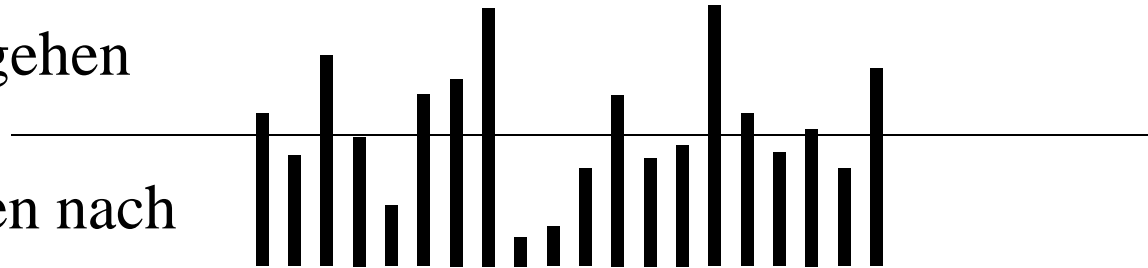
Quicksort: Illustration

Ausgangssituation:

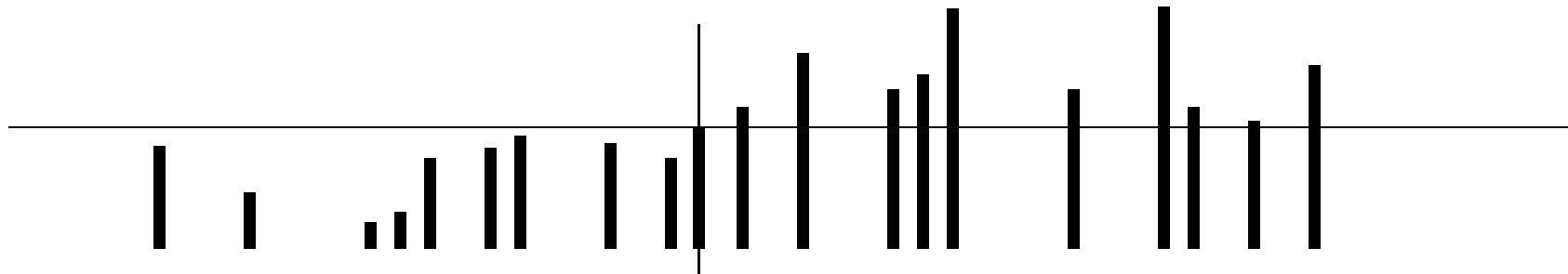


Zerlege gemäß der Line:

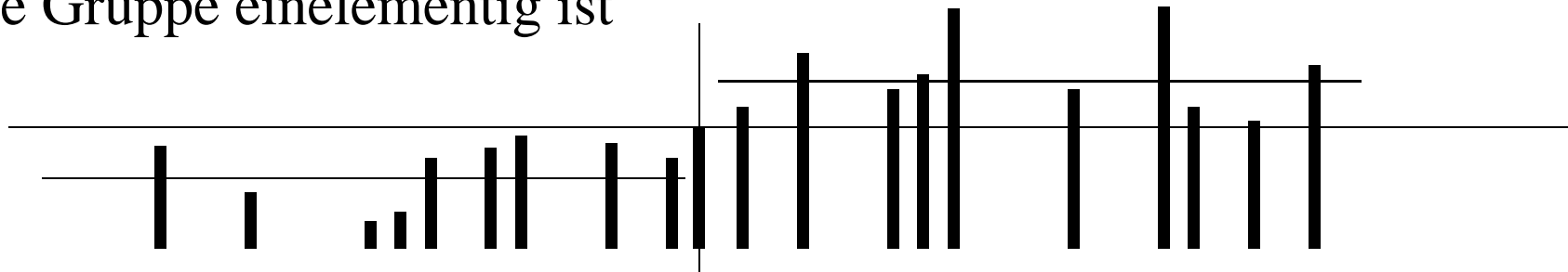
- alle, die größer sind gehen nach rechts,
- alle kleineren kommen nach links
- in der Mitte bleibe die, die genauso groß sind



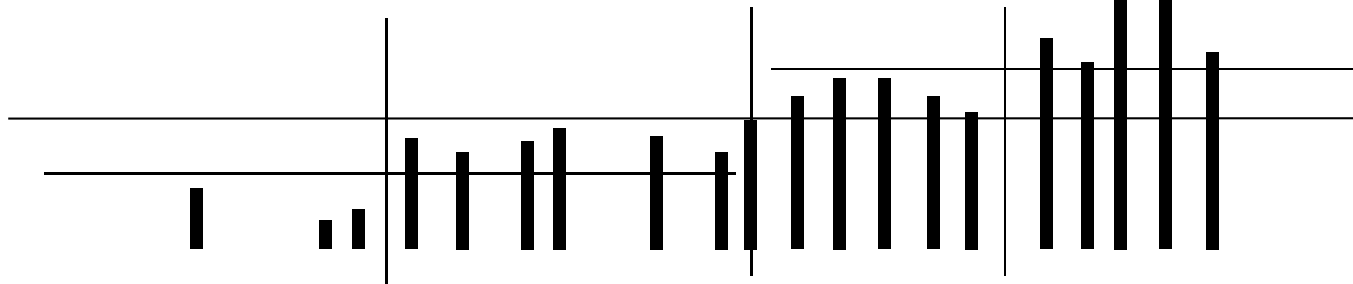
Quicksort: Illustration (Fort.)



Mache jetzt mit der linken und rechten Gruppe weiter, bis die Gruppe einelementig ist



Ergebnis:



Quicksort: Implementierung

ruft Hilfs-
funktion mit
maximalen
Grenzen auf

```
static <K> void quick_sort(K[] field, Compare<K> c) {  
    quick_sort_help(field, c, 0, field.length-1);  
}  
static <K> void quick_sort_help(K[] field, Compare<K> c, int iLeft, int iRight) {  
    final K MID = field[(iLeft + iRight) / 2];  
    int l = iLeft;  
    int r = iRight;  
  
    while(l < r) {  
        while(c.isLess(field[l], MID)) { ++l; }  
        while(c.isLess(MID, field[r])) { --r; }  
        if(l <= r)  
            swap(field, l++, r--);  
    }  
    if (iLeft < r)  
        quick_sort_help(field, c, iLeft, r);  
    if (iRight > l)  
        quick_sort_help(field, c, l, iRight);  
}
```

← nach MID müssen
sich alle richten

← suche Elemente, die
noch vertauscht
werden müssen

← sortiere rekursiv die
beiden restlichen Teile,
wenn notwendig

Quicksort: Analyse

Optimaler Fall:

- $\text{field}[(i\text{Left}+i\text{Right})/2]$ liegt in der Mitte, d.h. es gibt genauso viele kleinere wie größere Elemente
- d.h. nach einem Durchlauf wird das Problem der Größe N auf 2 Probleme jeweils der Größe $N/2$ reduziert
- d.h. $N + 2 * O(N/2) = N * \log(N)$

```
static <K> void quick_sort_help(K[] field,
                                Compare<K> c,
                                int iLeft,
                                int iRight) {
    final K MID = field[(iLeft + iRight) / 2];
    int l = iLeft;
    int r = iRight;

    while(l < r) {
        while(c.isLess(field[l], MID)) { ++l; }
        while(c.isLess(MID, field[r])) { --r; }
        if(l <= r)
            swap(field, l++, r--);
    }
    if (iLeft < r)
        quick_sort_help(field, c, iLeft, r);
    if (iRight > l)
        quick_sort_help(field, c, l, iRight);
}
```

Quicksort hat im Durchschnitt eine Komplexität von $O(N \log N)$

Quicksort: Analyse (Fort.)

Schlechter Fall:

- $\text{field}[(i\text{Left}+i\text{Right})/2]$ ist das kleinste oder größte Element
- d.h. nach einem Durchlauf wird das Problem der Größe N auf 2 Probleme der Größe $N-1$ und 1 reduziert
- d.h. $N + O(N-1) + O(1) = N^2$

```
static <K> void quick_sort_help(K[] field,
                                Compare<K> c,
                                int iLeft,
                                int iRight) {
    final K MID = field[(iLeft + iRight) / 2];
    int l = iLeft;
    int r = iRight;

    while(l < r) {
        while(c.isLess(field[l],MID)) { ++l; }
        while(c.isLess(MID,field[r])) { --r; }
        if(l <= r)
            swap(field, l++, r--);
    }
    if (iLeft < r)
        quick_sort_help(field,c, iLeft, r);
    if (iRight > l)
        quick_sort_help(field,c, l, iRight);
}
```

Quicksort hat im schlimmsten Fall eine Komplexität von $O(N^2)$

Vorlesung 15/1

Mergesort

- Idee:
 - wenn man zwei sortierte Listen hätte, dann könnte man eine neue sortierte Liste erzeugen, indem
 - man das kleinste Element der beiden Köpfe nimmt,
 - dieses entfernt
 - und mit dem Rest weitermacht

Mergesort: Beispiel

1.

200	155
102	40
45	30
23	-17

2.

200	155
102	40
45	30
23	

3.

200	155
102	40
45	30

4.

200	155
102	40
45	

5.

200	155
102	
45	

6.

200	155
102	

Ergebnis:

-17	23	30	40	45
-----	----	----	----	----

Mergesort: Implementierung

```
static <K> void merge_sort2(K[] field, Compare<K> c) {  
    merge_sort_help2(field, c, 0, field.length-1);  
}
```

```
static <K> void merge_sort_help2(K[] field, Compare<K> c, int iLeft, int iRight) {  
    if (iLeft < iRight) {
```

```
        final int MIDDLE = (iLeft + iRight) / 2;  
        merge_sort_help2(field, c, iLeft, MIDDLE);  
        merge_sort_help2(field, c, MIDDLE + 1, iRight);
```

die Mitte



sortiere links und
rechts der Mitte

```
        K[] tmp = (K[]) new Comparable[iRight - iLeft + 1];  
        for(int i = iLeft; i <= MIDDLE; ++i)
```

```
            tmp[i - iLeft] = field[i];  
        for(int i = MIDDLE+1; i <= iRight; ++i)  
            tmp[tmp.length-i+MIDDLE] = field[i];
```

lege eine Kopie an,
drehe dabei die 2. Hälfte
um

```
        int iL = 0;  
        int iR = tmp.length-1;  
        for(int i = iLeft; i <= iRight; ++i)  
            field[i] = c.isLess(tmp[iL], tmp[iR]) ? tmp[iL++] : tmp[iR--];
```

```
    }
```

```
}
```

mische aus der Kopie
in das Originalfeld

Mergesort: Analyse

- im Gegensatz zu Quicksort wird bei Mergesort das Feld immer genau in 2 gleichgroße Teile zerlegt
- beim Mischen wird $O(N)$ Zeit verbraucht
- somit ergibt sich eine Gesamtkomplexität von $O(N \log N)$
- der zusätzliche Speicheraufwand beträgt $O(N)$
- da beim Mischen immer nur auf den Kopf von 2 Läufern zugegriffen wird, eignet sich dieses Verfahren zum externen Sortieren
- im Durchschnitt ist das Verfahren langsamer als Quicksort

Der Mergesort hat garantiert ein $O(N \log N)$
Verhalten

Heapsort

Sei A eine Datenstruktur mit folgenden Eigenschaften:

- bei der Initialisierung sagt man, wieviele Elemente gespeichert werden sollen
- es gibt eine Methode insert(), der man das zu sortierende Element mitgibt
- es gibt eine Methode remove(), die das größte Element *zurückliefert und* dieses auch noch *entfernt*

Dann könnte man wie folgt sortieren:

Heapsort (Fort.)

field enthält N Elemente,
die zu sortieren sind

```
static <K> void heap_sort_1(K[] field, Compare<K> c) {  
    A<K> a = new A<K>(field.length);  
    for(int i = 0; i < field.length; ++i)    füge alle Elemente ein  
        a.insert(field[i], c);  
    for(int i = 0; i < field.length; ++i)  
        field[field.length - i - 1] = a.remove(c);  
}
```

lese alle Elemente
sortiert aus (mit dem
größten beginnend)

Gesucht ist eine solche Datenstruktur A

Heapsort (Fort.)

Möglichkeiten für eine solche Datenstruktur A:

- ein unsortiertes Array
 - insert erfolgt am Ende: Komplexität $O(1)$
 - remove durchläuft die Liste und sucht das Maximum: Komplexität $O(N)$
 - dies würde dem *Selection Sort* entsprechen: Komplexität $O(N^2)$
- ein sortiertes Array
 - insert erfolgt sortiert in das Array: Komplexität $O(N)$
 - remove entfernt das letzte Element: Komplexität $O(1)$
 - dies würde dem *Insertion Sort* entsprechen: Komplexität $O(N^2)$

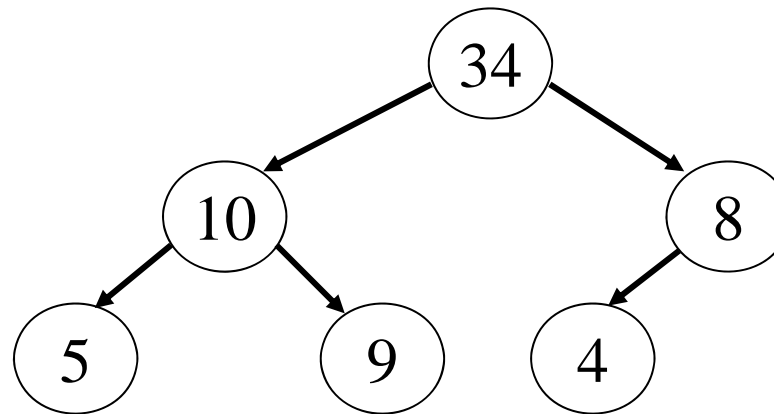
Heapsort (Fort.)

andere Möglichkeiten für eine solche Datenstruktur A:

- ein binärer Baum mit der folgenden Eigenschaft
- jeder Knoten enthält einen zu sortierenden Schlüssel
- der Schlüssel eines jeden Knoten ist größer oder gleich der Schlüssel seiner Söhne
- der Baum ist ausgeglichen, d.h. der Unterschied zwischen dem längsten und dem kürzesten Pfad von der Wurzel zu den Blättern beträgt maximal 1
- eine solche Datenstruktur nennt man ***Heap*** (siehe Graphentheorie)

Heapsort (Fort.)

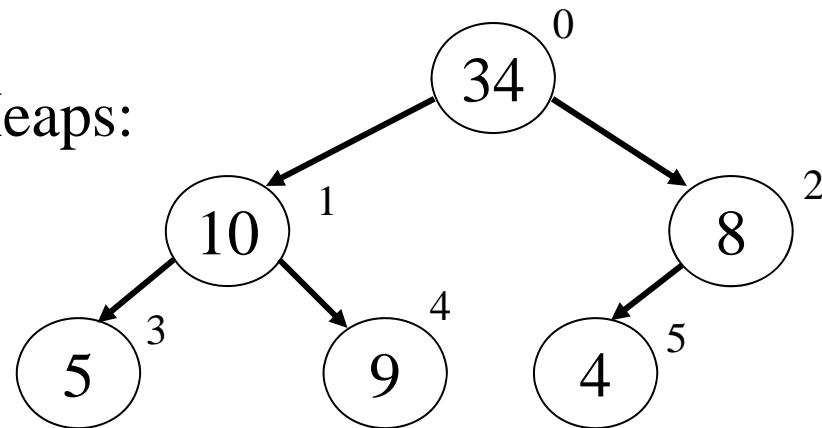
Beispiel für einen solchen Baum / Heap:



- jeder Knoten enthält einen Schlüssel, der größer als die seiner Söhne sind
- die Länge der Pfade zu den Blättern unterscheiden sich maximal um 1

Heapsort (Fort.)

Darstellung solcher Bäume / Heaps:



- wenn bekannt ist, wieviele Knoten maximal abgespeichert werden, können der Baum in einem Array abgespeichert werden

34	10	8	5	9	4
0	1	2	3	4	5

- von einem Knoten mit Index k wird auf die Söhne mittels $2*k+1$ und $2*k+2$ zugegriffen
- von einem Knoten mit Index k wird auf den Vater mittels $(k-1)/2$ zugegriffen

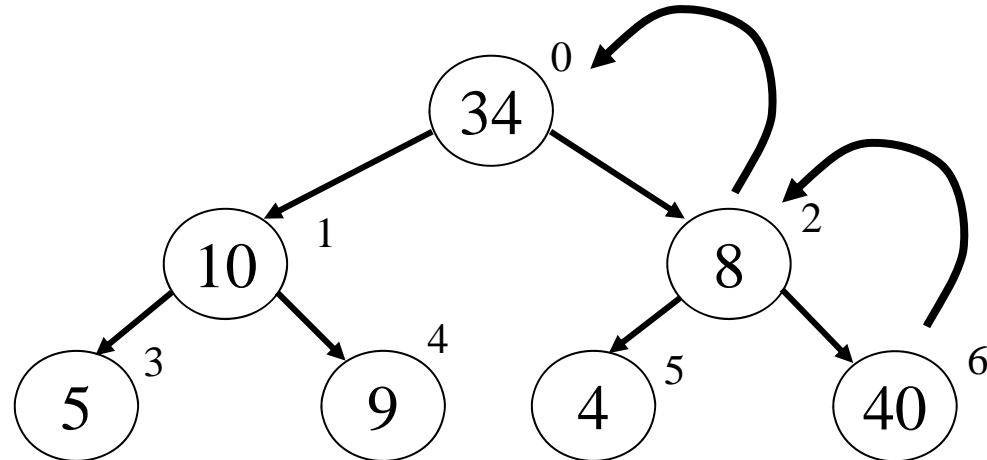
Heapsort (Fort.)

Implementierung eines Heaps für Comparable-Werte:

```
class Heap<K> {  
  
    public Heap(int iSize) {  
        m_iNext = 0;  
        m_Keys = (K[])new Object[iSize];  
    }  
  
    private int    m_iNext;        // der nächste freie Index  
    private K[]    m_Keys;        // die einzelnen Schlüssel  
}
```

Heapsort (Fort.)

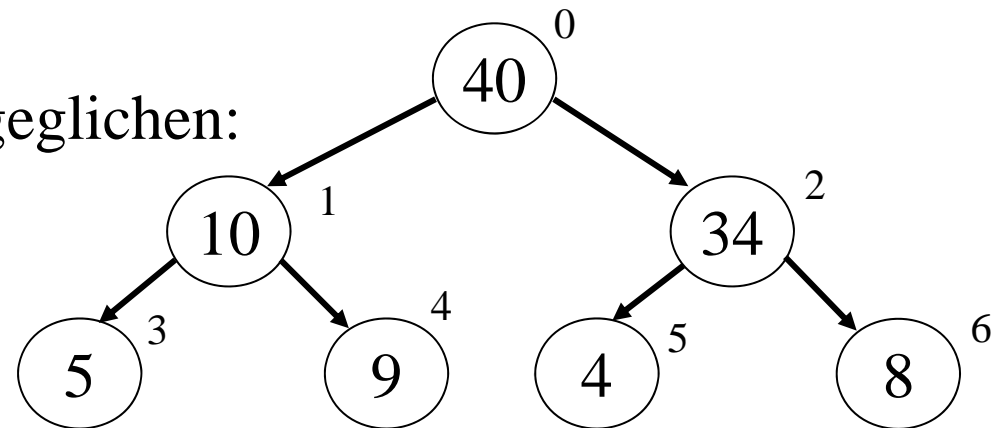
Einfügen eines Elements in einen solchen Baum:



- das neue Element wird am Ende des Arrays, sprich unten im Baum eingefügt
- dadurch verliert der Baum u.U. seine Eigenschaft, dass alle Knoten größere Schlüssel als ihre Söhne haben
- solche Schlüssel müssen dann nach oben wandern

Heapsort (Fort.)

dieser Baum ist wieder ausgeglichen:



- das nach-oben-wandern wird von der folgenden Methode upheap erledigt

```
private void upheap(int ilIndex, Compare<K> c) {  
    K k = m_Keys[ilIndex];  
    while (ilIndex != 0 && c.isLess(m_Keys[(ilIndex-1) / 2], k)) {  
        m_Keys[ilIndex] = m_Keys[(ilIndex-1) / 2];  
        ilIndex = (ilIndex - 1) / 2;  
    }  
    m_Keys[ilIndex] = k;  
}
```

Heapsort (Fort.)

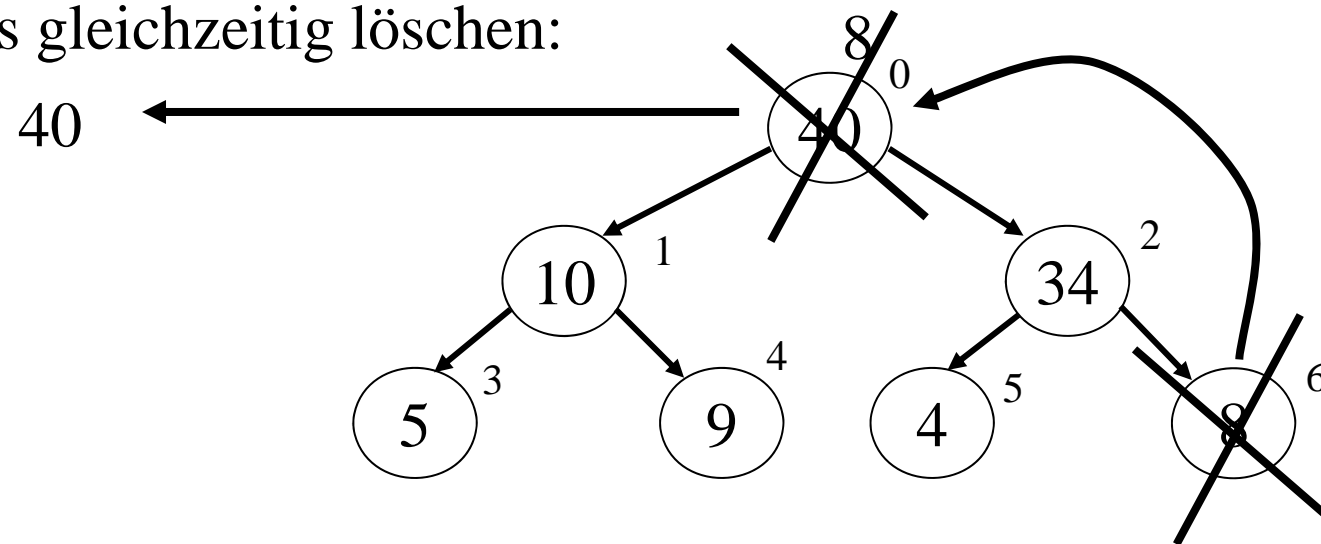
- basierend auf der upheap Methode kann die Insert Methode wie folgt implementiert werden:

```
public void insert(K key, Compare<K> c) {  
    m_Keys[m_iNext] = key;  
    upheap(m_iNext, c);  
    ++m_iNext;  
}
```

- zunächst wird das neue Element am Ende eingefügt
- dann wird die damit verbundene Unordnung wieder hergestellt
- am Ende wird der nächste freie Index um 1 erhöht

Heapsort (Fort.)

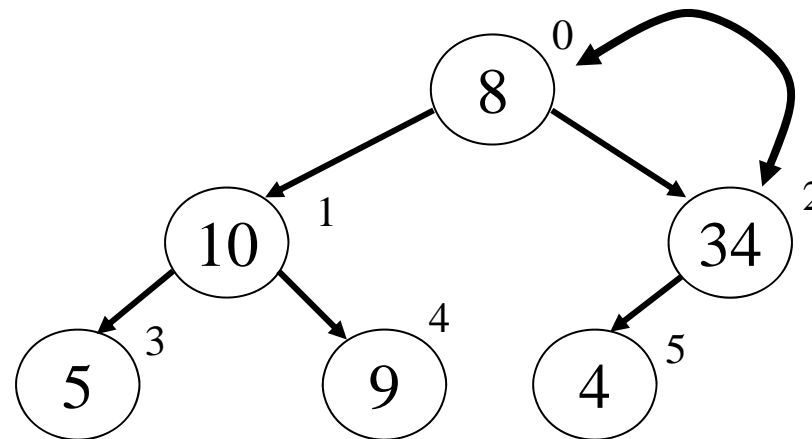
- die remove Methode soll das größte Element zurückliefern
- und es gleichzeitig löschen:



- das größte Element ist an der Spitze
- um es zu löschen, wird das letzte Element an dessen Stelle gesetzt

Heapsort (Fort.)

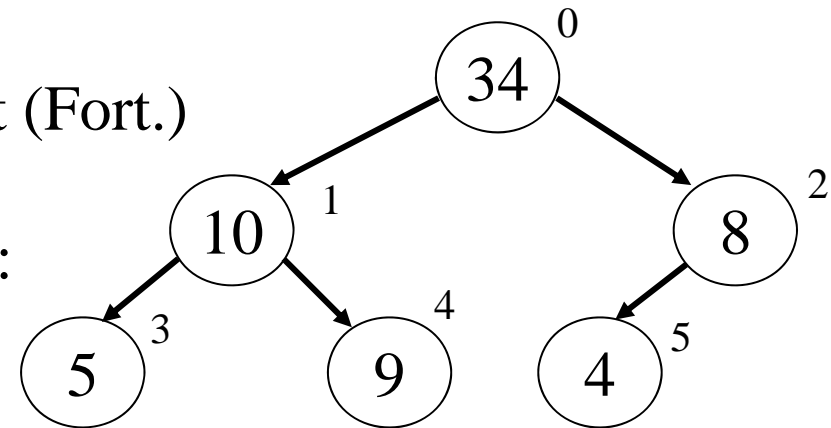
- der resultierende Baum ist nicht mehr korrekt
- die Spitze wird jetzt i.d.R. nicht mehr größer sein als die beiden Söhne



- solche Elemente müssen jetzt nach unten wandern
- ein Knoten wird dazu mit dem maximalen Sohn ausgetauscht

Heapsort (Fort.)

dieser Baum ist wieder ausgeglichen:



- das nach-unten-wandern wird von downheap erledigt

```

private void downheap(K ilIndex, Compare c) {
    int k = m_Keys[ilIndex];
    while (ilIndex < m_iNext / 2) {
        int iSon = 2 * ilIndex + 1;
        if (iSon < m_iNext-1 &&
            c.isLess(m_Keys[iSon], m_Keys[iSon + 1]))
            ++iSon;
        if (!c.isLess(k, m_Keys[iSon])) break;
        m_Keys[ilIndex] = m_Keys[iSon];
        ilIndex = iSon;
    }
    m_Keys[ilIndex] = k;
}
  
```

es gibt noch einen Sohn

1. Sohn

größerer der
beiden Söhne

Heapsort (Fort.)

- basierend auf der downheap Methode kann die Remove Methode wie folgt implementiert werden:

```
public K remove(Compare<K> c) {  
    K res = m_Keys[0];  
    m_Keys[0] = m_Keys[--m_iNext];  
    downheap(0,c);  
    return res;  
}
```

- zunächst wird das erste (größte) Element zwischengespeichert
- dann wird das letzte Element an die vorderste Front gestellt
- der inkonsistente Zustand wird durch das Hinunterwandern des 1. Elements wieder korrigiert

Heapsort (Fort.)

- mit den beiden Methoden insert und remove ist jetzt ein Sortierverfahren implementiert
- jede der beiden Operationen benötigt $O(\log N)$ Schritten, da der Binärbaum ausgeglichen ist
- somit ist die Gesamtlaufzeit $O(N \log N)$
- leider wird ein zusätzlicher Platz von $O(N)$ benötigt

Heapsort braucht
garantiert nur $O(N \log N)$
Zeit, ist im Durchschnitt
aber ein bisschen
langsamer als Quicksort

```
static <K> void heap_sort_1(K[] field, Compare<K> c) {  
    Heap<K> a = new Heap<K>(field.length);  
    for(int i = 0; i < field.length; ++i)  
        a.insert(field[i], c);  
    for(int i = 0; i < field.length; ++i)  
        field[field.length - i - 1] = a.remove(c);  
}
```

Heapsort (Fort.)

- Heapsort kann derart modifiziert werden, dass ohne zusätzlichen Speicherplatz sortiert werden kann
- Idee:
 - betrachte jeden Teilbaum von unten aufsteigend und mache ihn zum Heap, d.h. jeder Knoten muss einen größeren Schlüssel als seine Söhne haben (ist für die Blätter trivialerweise erfüllt)
 - jetzt steht das maximale Element am Anfang
 - vertausche das maximale Element mit dem letzten Element und stelle die Heapeigenschaft für das um 1 verkleinerte Array wieder her

Heapsort (Fort.)

- die Methode heapsort stützt sich auf eine Funktion downheap ab
- die Argumente
 - das Array, das den Heap enthält
 - die Anzahl der Elemente in dem Heap
 - den Index des Elements, dass jetzt in dem Heap runterwandern soll

```
static <K> void heap_sort(K[] field, Compare<K> c) {  
    for(int i = ((field.length-1)-1) / 2; i >= 0; --i)  
        Heap.downheap(field,c, field.length, i);  
    for(int i = field.length-1; i > 0; --i) {  
        swap(field, 0, i);  
        Heap.downheap(field,c, i, 0);  
    }  
}
```

Heapsort (Fort.)

- die Klassenmethode `downheap` der Klasse `Heap` für das Aufbauen des Heaps ohne neuen Speicher

```
public static <K> void downheap(K[] keys,  
                                Compare<K> c,  
                                int iEnd,  
                                int iIndex) {  
    K k = keys[iIndex];  
    while (iIndex < iEnd / 2) {  
        int iSon = 2 * iIndex + 1;  
        if (iSon < iEnd-1 && c.isLess(keys[iSon],keys[iSon + 1]))  
            ++iSon;  
        if (!c.isLess(k,keys[iSon]))  
            break;  
        keys[iIndex] = keys[iSon];  
        iIndex = iSon;  
    }  
    keys[iIndex] = k;  
}
```

Go!

Heapsort (Fort.)

- Vergleich der neuen Klassen- zur alten Objektmethode

```
public static <K> void downheap(K[] keys,  
                                Compare<K> c,  
                                int iEnd,  
                                int ilIndex) {  
    K k = keys[ilIndex];  
    while (ilIndex < iEnd / 2) {  
        int iSon = 2 * ilIndex + 1;  
        if (iSon < iEnd-1 &&  
            c.isLess( keys[iSon],  
                    keys[iSon + 1]))  
            ++iSon;  
        if (!c.isLess(k,keys[iSon]))  
            break;  
        keys[ilIndex] = keys[iSon];  
        ilIndex = iSon;  
    }  
    keys[ilIndex] = k;  
}
```

```
private void downheap(K ilIndex,  
                      Compare c) {  
    int k = m_Keys[ilIndex];  
    while (ilIndex < m_iNext / 2) {  
        int iSon = 2 * ilIndex + 1;  
        if (iSon < m_iNext-1 &&  
            c.isLess(m_Keys[iSon],  
                    m_Keys[iSon + 1]))  
            ++iSon;  
        if (!c.isLess(k ,m_Keys[iSon]))  
            break;  
        m_Keys[ilIndex] = m_Keys[iSon];  
        ilIndex = iSon;  
    }  
    m_Keys[ilIndex] = k;  
}
```


vordefinierte Sortiervverfahren in Java und C++

- in Java:

```
class Collection {  
    static void sort(List list);  
    static void sort(List list, Comparator c);  
}
```
- in C++: `sort`, `stable_sort`, `partial_sort`, `partial_sort_copy`

Sortiervverfahren: ein Vergleich

- Ein animierter Vergleich der vorgestellten Sortiervverfahren findet man hier:
- <http://www.sorting-algorithms.com/>

Sorting Algorithm Animations

SHARE

Problem Size: [20](#) · [30](#) · [40](#) · [50](#) Magnification: [1x](#) · [2x](#) · [3x](#)

Algorithm: [Insertion](#) · [Selection](#) · [Bubble](#) · [Shell](#) · [Merge](#) · [Heap](#) · [Quick](#) · [Quick3](#)

Initial Condition: [Random](#) · [Nearly Sorted](#) · [Reversed](#) · [Few Unique](#)

	 Insertion	 Selection	 Bubble	 Shell	 Merge	 Heap	 Quick	 Quick3
 Random								
 Nearly Sorted								
 Reversed								
 Few Unique								

Discussion

These pages show 8 different sorting algorithms on 4 different initial conditions. These visualizations are intended to:


- Show how each algorithm operates.
- Show that there is no best sorting algorithm.
- Show the advantages and disadvantages of each algorithm.
- Show that worst-case asymptotic behavior is not the deciding factor in choosing an algorithm.
- Show that the initial condition (input order and key distribution) affects performance as much as the algorithm choice.

The ideal sorting algorithm would have the following properties:

- Stable: Equal keys aren't reordered.
- Operates in place, requiring $O(1)$ extra space.
- Worst-case $O(n \lg(n))$ key comparisons.
- Worst-case $O(n)$ swaps.
- Adaptive: Speeds up to $O(n)$ when data is nearly sorted or when there are few unique keys.

There is no algorithm that has all of these properties, and so the choice of sorting algorithm depends on the application.

Directions

- Click on  above to restart the animations in a row, a column, or the entire table.
- Click directly on an animation image to start or restart it.
- Click on a problem size number to reset all animations.

Key

- Black values are sorted.
- Gray values are unsorted.
- A red triangle marks the algorithm position.
- Dark gray values denote the current interval (shell, merge, quick).
- A pair of red triangles marks the left and right pointers (quick).

References

Algorithms in Java, Parts 1-4, 3rd edition by Robert Sedgewick. Addison Wesley, 2003.

Programming Pearls by Jon Bentley. Addison Wesley, 1986.

Quicksort is Optimal by Robert Sedgewick and Jon Bentley, Knuthfest, Stanford University, January, 2002.