

WS 2017/18

# Einführung in die Informatik 1

Felix Brandt, Harald Räcke

Fakultät für Informatik  
TU München

<http://www14.in.tum.de/lehre/2017WS/info1/>

Wintersemester 2017/18

# Plagiatsstatistik 2016/17

Bachelor Informatik	482	17	3.5%
Bachelor Games Engineering	172	15	8.7%
Bachelor Wirtschaftsinformatik	186	23	12.4%
Master Wirtschaft mit Technologie	86	9	10.5%
Bachelor Technologie und Management	180	19	10.6%
Bachelor Maschinenwesen	2	1	50.0%
Master Maschinenwesen	1	1	100.0%
Bachelor Mathematik	88	3	3.4%
Summe	1197	88	7.4%

Diese Vorlesungsfolien basieren zum Grossteil auf der Einführungsvorlesung von Prof. Helmut Seidl (WS 2012/13).

# 1 Vom Problem zum Programm

Ein **Problem** besteht darin, aus einer Menge von Informationen eine weitere (unbekannte) Information zu bestimmen.

## mathematisch:

Ein Problem beschreibt eine Funktion  $f : E \rightarrow A$ , mit  $E =$  zulässige Eingaben und  $A =$  mögliche Ausgaben.

## Beispiele:

- ▶ Addition:  $f : \mathbb{Q} \times \mathbb{Q} \rightarrow \mathbb{Q}$
- ▶ Primzahltest:  $f : \mathbb{N} \rightarrow \{\text{yes, no}\}$
- ▶ Schach:  $f : \mathcal{P} \rightarrow \mathcal{Z}$ , wobei  $\mathcal{P}$  die Menge aller Schachpositionen ist, und  $f(P)$ , der beste Zug in Position  $P$ .

# Algorithmus

Ein **Algorithmus** ist ein **exaktes Verfahren** zur Lösung eines Problems, d.h. zur Bestimmung der gewünschten Resultate.

Man sagt auch ein Algorithmus **berechnet** eine Funktion  $f$ .



Ausschnitt aus Briefmarke, Soviet Union 1983  
Public Domain [↗](#)

Abu Abdallah  
Muhamed ibn Musa  
al-Chwarizmi, ca.  
780–835

## Beobachtung:

Nicht jedes Problem läßt sich durch einen Algorithmus lösen (↑**Berechenbarkeitstheorie**).

## Beweisidee: (↑**Diskrete Strukturen**)

- ▶ es gibt **überabzählbar unendlich** viele Probleme
- ▶ es gibt **abzählbar unendlich** viele Algorithmen

# Algorithmus

Das **exakte Verfahren** besteht i.a. darin, eine Abfolge von **elementaren Einzelschritten** der Verarbeitung festzulegen.

**Beispiel:** Alltagsalgorithmen

<i>Resultat</i>	<i>Algorithmus</i>	<i>Einzelschritte</i>
Pullover	Strickmuster	eine links, eine rechts, eine fallen lassen
Kuchen	Rezept	nimm 3 Eier ...
Konzert	Partitur	Noten

# Beispiel: Euklidischer Algorithmus

**Problem:** geg.  $a, b \in \mathbb{N}, a, b \neq 0$ . Bestimme  $\text{ggT}(a, b)$ .

**Algorithmus:**

1. Falls  $a = b$ , brich Berechnung ab. Es gilt  $\text{ggT}(a, b) = a$ .  
Ansonsten gehe zu Schritt 2.
2. Falls  $a > b$ , ersetze  $a$  durch  $a - b$  und setze Berechnung in Schritt 1 fort. Ansonsten gehe zu Schritt 3.
3. Es gilt  $a < b$ . Ersetze  $b$  durch  $b - a$  und setze Berechnung in Schritt 1 fort.

# Beispiel: Euklidischer Algorithmus

Hier sind  $q_a, q_b, q'_{a-b}, q'_b \in \mathbb{Z}$ .

## Warum geht das?

Wir zeigen, für  $a > b$ :  $\text{ggT}(a, b) = \text{ggT}(a - b, b)$ .

Seien  $g = \text{ggT}(a, b)$ ,  $g' = \text{ggT}(a - b, b)$ .

Dann gilt:

$$\begin{array}{l} a = q_a \cdot g \\ b = q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{l} a - b = q'_{a-b} \cdot g' \\ b = q'_b \cdot g' \end{array}$$

$$\begin{array}{l} a - b = (q_a - q_b) \cdot g \\ b = q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{l} a = (q'_{a-b} + q'_b) \cdot g' \\ b = q'_b \cdot g' \end{array}$$

Das heißt  $g$  ist Teiler von  $a - b, b$  und  $g'$  ist Teiler von  $a, b$ .

Daraus folgt  $g \leq g'$  und  $g' \leq g$ , also  $g = g'$ .

# Eigenschaften

Ein klassischer Algorithmus erfüllt alle Eigenschaften.  
Häufig spricht man aber auch von Algorithmen wenn einige dieser Eigenschaften verletzt sind.

**(statische) Finitheit.** Die Beschreibung des Algorithmus besitzt endliche Länge. (↑**nichtuniforme Algorithmen**)

**(dynamische) Finitheit.** Die bei Abarbeitung entstehenden Zwischenergebnisse sind endlich.

**Terminiertheit.** Algorithmen, die nach endlich vielen Schritten ein Resultat liefern, heißen **terminierend**. (↑**Betriebssysteme**, ↑**reaktive Systeme**)

**Determiniertheit.** Bei gleichen Eingabedaten gibt ein Algorithmus das gleiche Ergebnis aus. (↑**randomisierte Algorithmen**, ↑**nicht-deterministische Algorithmen**)

**Determinismus.** Der nächste anzuwendende Schritt im Verfahren ist stets eindeutig definiert. (↑**randomisierte Algorithmen**, ↑**nicht-deterministische Algorithmen**)

# Programm

Ein **Programm** ist die **Formulierung** eines Algorithmus in einer **Programmiersprache**.

Die Formulierung gestattet (hoffentlich) eine maschinelle Ausführung.

- ▶ Ein **Programmsystem** berechnet i.a. nicht nur eine Funktion, sondern **immer wieder** Funktionen in Interaktion mit Benutzerinnen und/oder der Umgebung.
- ▶ Es gibt viele Programmiersprachen: **Java**, **C**, **Prolog**, **Fortran**, **TeX**, **PostScript**, . . .

Eine Programmiersprache ist **gut**, wenn

- ▶ **die Programmiererin** in ihr algorithmische Ideen **natürlich** beschreiben kann, insbesondere später noch versteht was das Programm tut (oder nicht tut);
- ▶ **ein Computer** das Programm leicht verstehen und **effizient** ausführen kann.

## 2 Eine einfache Programmiersprache

Eine Programmiersprache soll

- ▶ Datenstrukturen anbieten
- ▶ Operationen auf Daten erlauben
- ▶ **Kontrollstrukturen** zur Ablaufsteuerung bereitstellen

Als Beispiel betrachten wir **Minijava**.

Variablen dienen zur Speicherung von Daten.

Um Variablen in **MiniJava** zu nutzen müssen sie zunächst eingeführt, d.h. **deklariert** werden.

# Variablen

## Beispiel:

```
int x, result;
```

Diese Deklaration führt die beiden Variablen mit den **Namen** `x` und `result` ein.

- ▶ Das Schlüsselwort `int` besagt, dass diese Variablen ganze Zahlen („Integers“) speichern sollen.  
`int` heißt auch **Typ** der Variablen `x` und `result`.
- ▶ Variablen können dann benutzt werden, um anzugeben, auf welche Daten Operationen angewendet werden sollen.
- ▶ Die Variablen in der Aufzählung sind durch Kommas „`,`“ getrennt.
- ▶ Am Ende steht ein Semikolon „`;`“.

# Operationen – Zuweisung

Operationen gestatten es, Werte von Variablen zu ändern. Die wichtigste Operation ist die **Zuweisung**.

## Beispiele:

▶ `x = 7;`

Die Variable `x` erhält den Wert `7`.

▶ `result = x;`

Der Wert der Variablen `x` wird ermittelt und der Variablen `result` zugewiesen.

▶ `result = x + 19;`

Der Wert der Variablen `x` wird ermittelt, `19` dazu gezählt und dann das Ergebnis der Variablen `result` zugewiesen.

## Achtung:

- ▶ **Java** bezeichnet die Zuweisung mit „`=`“ anstatt „`:=`“ (Erbschaft von **C**...)
- ▶ Eine Zuweisung wird mit „`;`“ beendet.
- ▶ In der Zuweisung `x = x + 1;` greift das `x` auf der rechten Seite auf den Wert **vor** der Zuweisung zu.

# Operationen – Input/Output

**Minijava** enthält Operationen um Daten (Zahlen) einlesen bzw. ausgeben zu können.

## Beispiele:

- ▶ `x = read();`  
Liest eine Folge von Zeichen ein und interpretiert sie als ganze Zahl, deren Wert sie der Variablen `x` als Wert zuweist.
- ▶ `write(42);`  
Schreibt `42` auf die Ausgabe.
- ▶ `write(result);`  
Bestimmt den Wert der Variablen `result` und schreibt dann diesen auf die Ausgabe.
- ▶ `write(x-14);`  
Bestimmt den Wert der Variablen `x`, subtrahiert `14` und schreibt das Ergebnis auf die Ausgabe.

# Operationen – Input/Output

## Achtung:

- ▶ Das argument der `write`-Operation in den Beispielen ist ein `int`.
- ▶ Um es ausgeben zu können muss es erst in eine `Zeichenfolge` umgewandelt werden, d.h. einen `String`

In `Minijava` können auch direkt Strings ausgegeben werden:

## Beispiel:

- ▶ `write("Hello World!!!");`  
Schreibt `Hello World!!!` auf die Ausgabe.

# Kontrollstrukturen – Sequenz

## Sequenz:

```
1 int x, y, result;  
2 x = read();  
3 y = read();  
4 result = x + y;  
5 write(result);
```

- ▶ Zu jedem Zeitpunkt wird nur eine Operation ausgeführt.
- ▶ Jede Operation wird genau einmal ausgeführt.
- ▶ Die Reihenfolge, in der die Operationen ausgeführt werden, ist die gleiche, in der sie im Programm stehen.
- ▶ Mit Beendigung der letzten Operation endet die Programm-Ausführung.

**Sequenz** alleine erlaubt nur sehr einfache Programme.

# Kontrollstrukturen – Selektion

## Selektion (bedingte Auswahl):

```
1 int x, y, result;
2 x = read();
3 y = read();
4 if (x > y)
5     result = x - y;
6 else
7     result = y - x;
8 write(result);
```

- ▶ Zuerst wird die Bedingung ausgewertet
- ▶ Ist sie erfüllt, wird die nächste Operation ausgeführt.
- ▶ Ist sie nicht erfüllt, wird die nächste Operation nach dem `else`-Zweig ausgeführt.

# Kontrollstrukturen – Selektion

## Beispiel:

- ▶ Statt einer einzelnen Operation können die Alternativen auch aus **Statements** bestehen:

```
1 int x;  
2 x = read();  
3 if (x == 0)  
4     write(0);  
5 else if (x < 0)  
6     write(-1);  
7 else  
8     write(+1);
```

# Kontrollstrukturen – Selektion

## Beispiel:

- ▶ ... oder aus (geklammerten) Folgen von Operationen und Statements:

```
1 int x, y;  
2 x = read();  
3 if (x != 0) {  
4     y = read();  
5     if (x > y)  
6         write(x);  
7     else  
8         write(y);  
9 } else  
10    write(0);
```

# Kontrollstrukturen – Selektion

## Beispiel:

- ▶ ...eventuell fehlt auch der `else`-Teil:

```
1 int x, y;  
2 x = read();  
3 if (x != 0) {  
4     y = read();  
5     if (x > y)  
6         write(x);  
7     else  
8         write(y);  
9 }
```

Auch mit Sequenz und Selektion kann noch nicht viel berechnet werden...

# Kontrollstrukturen – Iteration

## Iteration (wiederholte Ausführung)

```
1 int x, y;  
2 x = read(); y = read();  
3 while (x != y) {  
4     if (x < y)  
5         y = y - x;  
6     else  
7         x = x - y;  
8 }  
9 write(x);
```

Das Programm erfüllt die Spezifikation, dass es für  $x, y \in \mathbb{N} \setminus \{0\}$ , den GGT berechnet. Bei falscher Eingabe terminiert es eventuell nicht.

Im allgemeinen sollte man Eingaben vom Benutzer immer auf Zulässigkeit prüfen.

- ▶ Zuerst wird die Bedingung ausgewertet.
- ▶ Ist sie erfüllt, wird der **Rumpf** des **while**-statements ausgeführt.
- ▶ Nach Ausführung des Rumpfs wird das gesamte **while**-statement erneut ausgeführt.
- ▶ Ist die Bedingung nicht erfüllt fährt die Programmausführung hinter dem **while**-statement fort.

## 2 Eine einfache Programmiersprache

Eine Sprache mit dieser Eigenschaft nennt man auch **turingvollständig**.

### Theorem (↑**Berechenbarkeitstheorie**)

Jede (partielle) Funktion auf ganzen Zahlen, die überhaupt berechenbar ist, läßt sich mit Selektion, Sequenz, und Iteration, d.h., mithilfe eines MiniJava-Programms berechnen.

### Beweisidee

- ▶ Was heißt berechenbar?

Eine Funktion heißt berechenbar wenn man sie mithilfe einer Turingmaschine berechnen kann.

- ▶ Schreibe ein **MiniJava**-Programm, das eine Turingmaschine simuliert.

Für dieses Theorem ist es wichtig, dass die **int**-Variablen von **MiniJava** beliebige ganze Zahlen speichern können. Sobald man dies einschränkt (z.B. 32 Bit) ist die entstehende Sprache immer noch sehr mächtig aber streng formal nicht mehr **turingvollständig**.

## 2 Eine einfache Programmiersprache

Minijava-Programme sind ausführbares **Java**. Man muss sie nur geeignet **dekoriieren**.

**Beispiel:** das GGT-Programm.

```
1 int x, y;
2 x = read();
3 y = read();
4 while (x != y) {
5     if (x < y)
6         y = y - x;
7     else
8         x = x - y;
9 }
10 write(x);
```

# Ein Java-Programm

```
1 public class GGT extends MiniJava {
2     public static void main (String[] args) {
3         int x, y;
4         x = read();
5         y = read();
6         while (x != y) {
7             if (x < y)
8                 y = y - x;
9             else
10                x = x - y;
11        }
12        write(x);
13    } // Ende der Definition von main()
14 } // Ende der Definition der Klasse GGT
```

Datei "GGT.java"

# Ein Java-Programm

## Erläuterungen:

- ▶ Jedes Programm hat einen **Namen** (hier **GGT**)
- ▶ Der Name steht hinter dem Schlüsselwort **class** (was eine Klasse ist, was **public** ist lernen wir später)
- ▶ Der Dateiname muss zum Programmnamen „passen“, d.h. in diesem Fall **GGT.java** heißen.
- ▶ Das **MiniJava**-Programm ist der Rumpf des **Hauptprogramms**, d.h. der Funktion **main()**.
- ▶ Die Programmausführung eines **Java**-Programms startet stets mit einem Aufruf dieser Funktion **main()**.
- ▶ Die Operationen **write()** und **read()** werden in der Klasse **MiniJava** definiert.
- ▶ Durch **GGT extends MiniJava** machen wir diese Operationen innerhalb des **GGT**-Programms verfügbar.

```
1 import javax.swing.JOptionPane;
2 import javax.swing.JFrame;
3 public class MiniJava {
4     public static int read() {
5         JFrame f = new JFrame();
6         String s = JOptionPane.showInputDialog(f, "Eingabe:");
7         int x = 0; f.dispose();
8         if (s == null) System.exit(0);
9         try { x = Integer.parseInt(s.trim());
10        } catch (NumberFormatException e) { x = read(); }
11        return x;
12    }
13    public static void write(String x) {
14        JFrame f = new JFrame();
15        JOptionPane.showMessageDialog (f, x, "Ausgabe",
16        JOptionPane.PLAIN_MESSAGE);
17        f.dispose();
18    }
19    public static void write(int x) { write(""+x); }
20 }
```

Datei: "MiniJava.java"

## Weitere Erläuterungen:

- ▶ Jedes Programm sollte Kommentare enthalten, damit man sich selbst später noch darin zurecht findet!
- ▶ Ein Kommentar in **Java** hat etwa die Form:  
`// Das ist ein Kommentar!!!`
- ▶ Wenn er sich über mehrere Zeilen erstrecken soll dann  
`/* Dieser Kommentar ist verdammt  
Taaaaaaaaaaaang  
*/`

## 2 Eine einfache Programmiersprache

Das Programm GGT kann nun übersetzt und dann ausgeführt werden:

```
raecke> javac GGT.java  
raecke> java GGT
```

- ▶ Der Compiler `javac` liest das Programm aus den Dateien `GGT.java` und `MiniJava.java` ein und erzeugt für sie JVM-Code, den er in den Dateien `GGT.class` und `MiniJava.class` ablegt.
- ▶ Das Laufzeitsystem `java` liest die Dateien `GGT.class` und `MiniJava.class` ein und führt sie aus.

## Wichtige Erweiterung – Arrays

Arrays enthalten eine Gruppe von Variablen auf die über einen **index** zugegriffen wird:

- ▶ Deklariere Variablen `a[0], ..., a[99]` und `b[0], ..., b[4]`:

```
int[] a = new int[100], b = new int[5];
```

- ▶ Greife auf Element `a[5]` zu:

```
int i;  
int[] a = new int[100];  
a[5] = 1; // a[5] ist jetzt 1  
i = 5;  
a[i] = 7; // a[5] ist jetzt 7  
i = 7;  
a[i-2] = 8; // a[5] ist jetzt 8
```

Minijava ist sehr primitiv

Die Programmiersprache Java bietet noch eine Fülle von Hilfsmitteln an, die das Programmieren erleichtern sollen.

Insbesondere gibt es

- ▶ viele weitere Datentypen (nicht nur `int`) und
- ▶ viele weitere Kontrollstrukturen

... kommt später in der Vorlesung!

# 3 Syntax von Programmiersprachen

## Syntax („Lehre vom Satzbau“)

- ▶ formale Beschreibung des Aufbaus der „Worte“ und „Sätze“, die zu einer Sprache gehören;
- ▶ im Falle einer **Programmiersprache** Festlegung, wie Programme aussehen müssen.

## Hilfsmittel bei natürlicher Sprache

- ▶ Wörterbücher;
- ▶ Rechtschreibregeln, Trennungsregeln, Grammatikregeln;
- ▶ Ausnahmelisten;
- ▶ Sprachgefühl.

# Hilfsmittel

## Hilfsmittel bei Programmiersprachen

- ▶ Listen von **Schlüsselworten** wie `if`, `int`, `else`, `while`...
- ▶ Regeln, wie einzelne Worte (**Tokens**) z.B. **Namen** gebildet werden.  
Frage: Ist `x10` ein zulässiger Name für eine Variable (oder `_ab` oder `A#B` oder `0A?B`)?...
- ▶ Grammatikregeln, die angeben, wie größere Komponenten aus kleineren aufgebaut werden.  
Frage: Ist ein `while`-Statement im `else`-Teil erlaubt?
- ▶ Kontextbedingungen.  
**Beispiel:** Eine Variable muss erst deklariert sein, bevor sie verwendet wird.

Programmiersprachen sind

- ▶ formalisierter als natürliche Sprache
- ▶ besser für maschinelle Verarbeitung geeignet.

## Semantik („Lehre von der Bedeutung“)

- ▶ Ein Satz einer (natürlichen) Sprache verfügt zusätzlich über eine **Bedeutung**, d.h. teilt einem Hörer/Leser einen Sachverhalt mit (↑**Information**)
- ▶ Ein Satz einer Programmiersprache, d.h. ein Programm verfügt ebenfalls über eine **Bedeutung**...

# Syntax vs. Semantik

Die Bedeutung eines Programms ist

- ▶ alle möglichen **Ausführungen** der beschriebenen Berechnung (↑**operationelle Semantik**); oder
- ▶ die definierte **Abbildung** der Eingaben auf die Ausgaben (↑**denotationelle Semantik**).

## Achtung!

Ist ein Programm **syntaktisch korrekt**, heißt das noch lange nicht, dass es auch das „richtige“ tut, d.h. **semantisch korrekt** ist!!!

## 3.1 Reservierte Wörter

- ▶ `int`  
⇒ Bezeichner für Basistypen;
- ▶ `if, else, then, while...`  
⇒ Schlüsselwörter für Programmkonstrukte;
- ▶ `(, ), ", ', {, }, ,, ;`  
⇒ Sonderzeichen;

## 3.2 Was ist ein erlaubter Name?

### Schritt 1:

Festlegung erlaubter Zeichen:

```
letter ::= $ | _ | a | ... | z | A | ... | Z
digit  ::= 0 | ... | 9
```

- ▶ **letter** und **digit** bezeichnen **Zeichenklassen**, d.h. Mengen von Zeichen, die gleich behandelt werden.
- ▶ Das Symbol „|“ trennt zulässige Alternativen.
- ▶ Das Symbol „...“ repräsentiert die Faulheit, alle Alternativen wirklich aufzuzählen.

## 3.2 Was ist ein erlaubter Name?

Wir definieren hier **MiniJava**. Eigentliches **Java** erlaubt mehr Namen (z.B. sind UTF8-Symbole erlaubt).

### Schritt 2:

Festlegung der Zeichenanordnung:

`name ::= letter ( letter | digit )*`

- ▶ Erst kommt ein Zeichen der Klasse **letter**, dann eine (eventuell auch leere) Folge von Zeichen entweder aus **letter** oder aus **digit**.
- ▶ Der Operator „\*“ bedeutet „beliebig oft wiederholen“ („weglassen“ ist 0-malige Wiederholung).
- ▶ Der Operator „\*“ ist ein **Postfix**-Operator. Das heißt, er steht hinter seinem Argument.

# Beispiele

\_178

Das\_ist\_kein\_Name

x

—

\$Password\$

...sind legale Namen.

5ABC

!Hallo!

x'

a=b

-178

...sind keine legalen Namen.

## Achtung

Reservierte Wörter sind als Namen verboten.

## 3.3 Ganze Zahlen

Werte, die direkt im Programm stehen, heißen **Konstanten**.  
Ganze nichtnegative Zahlen bestehen aus einer nichtleeren Folge von Ziffern:

`number ::= digit digit*`

- ▶ Wie sähe die Regel aus, wenn wir führende Nullen verbieten wollen?

# Beispiele

17

12490

42

0

00070

...sind `int`-Konstanten

"Hello World!"

0.5e+128

...sind keine `int`-Konstanten

# Reguläre Ausdrücke

Die Alternative hat eine geringere Bindungsstärke als die Konkatenation. D.h.  $ab|c$  steht für die Wörter  $ab$  oder  $c$  und nicht für  $ab$  oder  $ac$ .

Ausdrücke, die aus Zeichen(-klassen) mithilfe von

- | (Alternative)
- \* (Iteration)
- (Konkatenation) sowie
- ? (Option)

...aufgebaut sind, heißen **reguläre Ausdrücke** (↑Automatentheorie).

Der Postfix-Operator „?“ besagt, dass das Argument eventuell auch fehlen darf, d.h. einmal oder keinmal vorkommt.

Gelegentlich sind auch  $\epsilon$ , d.h. das „leere Wort“ sowie  $\emptyset$ , d.h. die leere Menge zugelassen.

# Beispiele

Reguläre Ausdrücke reichen zur Beschreibung **einfacher** Mengen von Worten aus.

- ▶  $(\text{letter letter})^*$   
⇒ alle Wörter gerader Länge (über  $\$, \_, a, \dots, z, A, \dots, Z$ );
- ▶  $\text{letter}^* \text{test letter}^*$   
⇒ alle Wörter, die das Teilwort **test** enthalten;
- ▶  $\_ \text{digit}^* 17$   
⇒ alle Wörter, die mit  $\_$  anfangen, dann eine beliebige Folge von Ziffern aufweisen, die mit 17 aufhört;
- ▶  $\text{exp} ::= (\text{e|E})(+|-)? \text{digit digit}^*$   
 $\text{float} ::= \text{digit digit}^* \text{exp} \mid$   
 $\text{digit}^* (\text{digit} \cdot \mid \cdot \text{digit}) \text{digit}^* \text{exp}?$   
⇒ alle Gleitkommazahlen...

# Programmverarbeitung

## 1. Phase (↑Scanner)

Identifizierung von

- ▶ reservierten Wörtern,
- ▶ Namen,
- ▶ Konstanten

Ignorierung von

- ▶ Whitespace,
- ▶ Kommentaren

⇒ Input wird mit regulären Ausdrücken verglichen und dabei in Wörter („Tokens“) zerlegt.

## 2. Phase (↑Parser)

Analyse der **Struktur** des Programms.

## 3.4 Struktur von Programmen

Programme sind **hierarchisch** aus Komponenten aufgebaut. Für jede Komponente geben wir Regeln an, wie sie aus anderen Komponenten zusammengesetzt sein können.

```
program    ::=  decl* stmt*
decl       ::=  type name ( , name )* ;
type       ::=  int
```

- ▶ Ein Programm besteht aus einer Folge von Deklarationen, gefolgt von einer Folge von Statements.
- ▶ Eine Deklaration gibt den Typ an, hier: **int**, gefolgt von einer Komma-separierten Liste von Variablennamen.

# Anweisungen

```
stmt ::= ; | { stmt* } |  
      name = expr; | name = read(); |  
      write( expr ); |  
      if ( cond ) stmt |  
      if ( cond ) stmt else stmt |  
      while ( cond ) stmt
```

- ▶ Ein Statement ist entweder „leer“ (d.h. gleich ; ) oder eine geklammerte Folge von Statements;
- ▶ oder eine Zuweisung, eine Lese- oder Schreiboperation;
- ▶ eine (einseitige oder zweiseitige) bedingte Verzweigung;
- ▶ oder eine Schleife.

# Ausdrücke

`expr` ::= `number` | `name` | `( expr )` |  
`unop expr` | `expr binop expr`

`unop` ::= `-`

`binop` ::= `-` | `+` | `*` | `/` | `%`

- ▶ Ein Ausdruck ist eine Konstante, eine Variable oder ein geklammerter Ausdruck
- ▶ oder ein unärer Operator, angewandt auf einen Ausdruck,
- ▶ oder ein binärer Operator, angewandt auf zwei Argumentausdrücke.
- ▶ Einziger unärer Operator ist (bisher) die Negation.
- ▶ Mögliche binäre Operatoren sind Addition, Subtraktion, Multiplikation, (ganzzahlige) Division und Modulo.

# Bedingungen

```
cond ::= true | false | ( cond ) |  
      expr comp expr |  
      bunop ( cond ) | cond bbinop cond  
comp ::= == | != | <= | < | >= | >  
bunop ::= !  
bbinop ::= && | ||
```

- ▶ Bedingungen unterscheiden sich dadurch von Ausdrücken, dass ihr Wert nicht vom Typ `int` ist sondern `true` oder `false` (ein **Wahrheitswert** – vom Typ `boolean`).
- ▶ Bedingungen sind darum Konstanten, Vergleiche
- ▶ oder logische Verknüpfungen anderer Bedingungen.

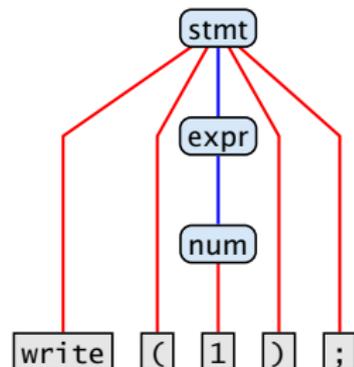
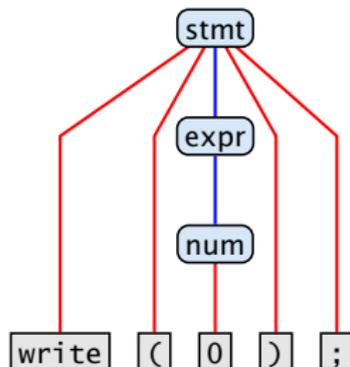
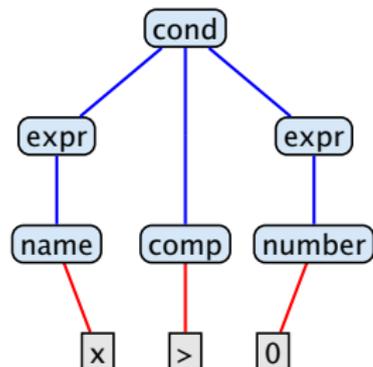
# Beispiel

```
int x;  
x = read();  
if (x > 0)  
    write(1);  
else  
    write(0);
```

Die hierarchische Untergliederung von Programm-Bestandteilen veranschaulichen wir durch **Syntax-Bäume**.

# Syntaxbäume

Syntaxbäume für  $x > 0$  sowie `write(0);` und `write(1);`

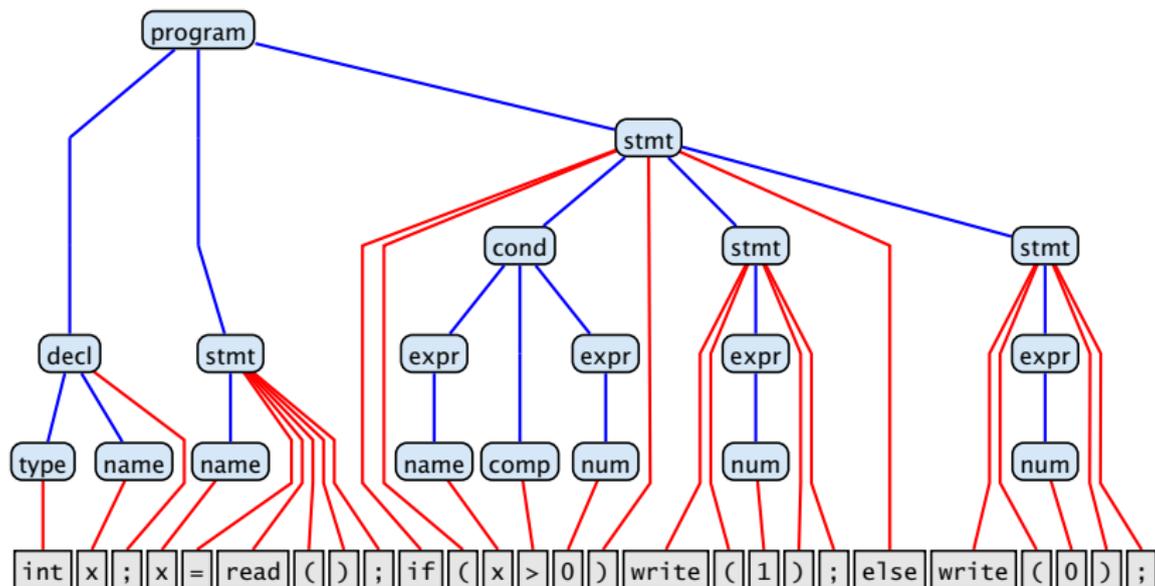


**Blätter:** Wörter/Tokens

**innere Knoten:** Namen von Programmbestandteilen

# Beispiel

Der komplette Syntaxbaum unseres Beispiels:

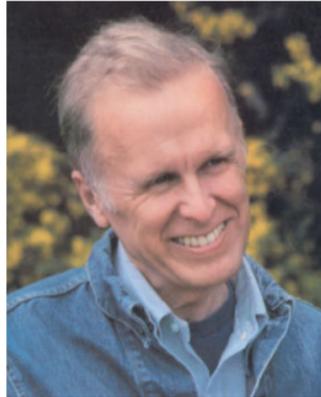


# Bemerkungen

- ▶ Die vorgestellte Methode der Beschreibung von Syntax heißt **EBNF-Notation** (**E**xtended **B**ackus **N**aur **F**orm Notation).
- ▶ Ein anderer Name dafür ist **erweiterte kontextfreie Grammatik** (↑**Linguistik**, ↑**Automatentheorie**).
- ▶ Linke Seiten von Regeln heißen auch **Nichtterminale**.
- ▶ Tokens heißen auch **Terminale**.



Noam Chomsky,  
MIT



John Backus, IBM  
Turing Award  
(Erfinder von Fortran)



Peter Naur,  
Turing Award  
(Erfinder von Algol60)

# Kontextfreie Grammatiken

## Achtung:

- ▶ Die regulären Ausdrücke auf den rechten Regelseiten können sowohl Terminale wie Nichtterminale enthalten.
- ▶ Deshalb sind kontextfreie Grammatiken **mächtiger** als reguläre Ausdrücke.

## Beispiel:

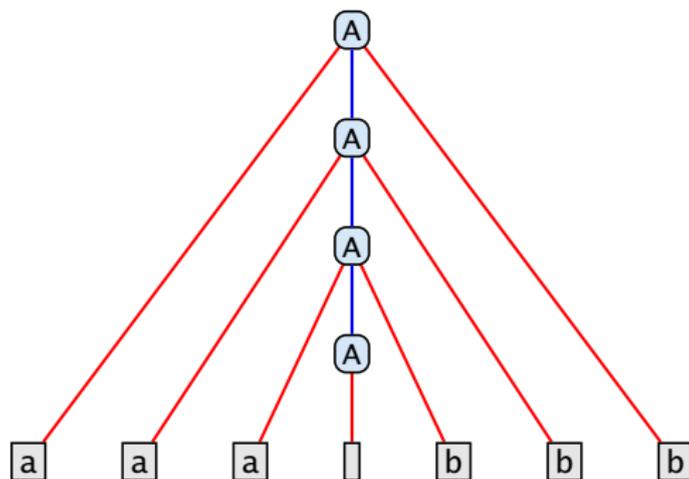
$$\mathcal{L} = \{\epsilon, ab, aabb, aaabbb, \dots\}$$

lässt sich mithilfe einer Grammatik beschreiben:

$$A ::= (a A b)?$$

# Kontextfreie Grammatiken

Syntaxbaum für das Wort **aaabbb**:



Für  $\mathcal{L}$  gibt es aber keinen regulären Ausdruck  
(↑Automatentheorie).

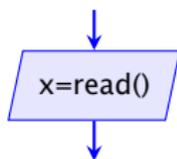
## 4 Kontrollflussdiagramme

In welcher Weise Programmteile nacheinander ausgeführt werden kann anschaulich durch **Kontrollflussdiagramme** dargestellt werden.

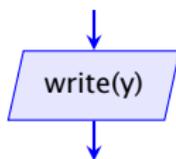
**Zutaten:**



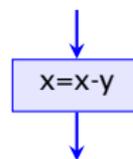
## 4 Kontrollflussdiagramme



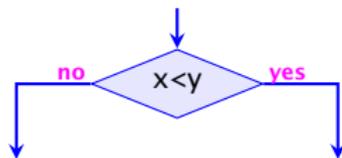
Eingabe



Ausgabe



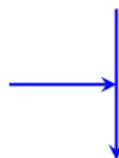
Zuweisung



bedingte  
Verzweigung



Kante



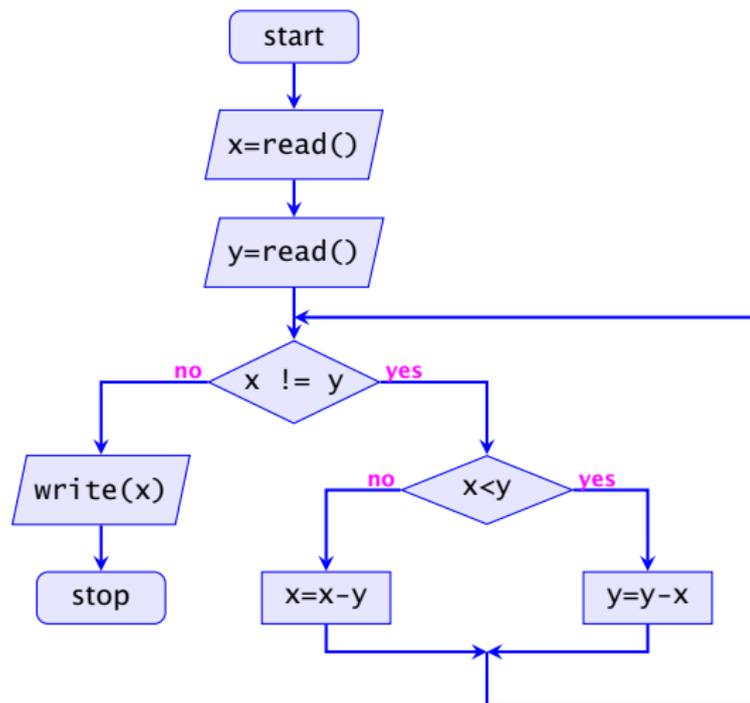
Zusammenlauf

# 4 Kontrollflussdiagramme

## Beispiel:

```
int x, y;  
x = read();  
y = read();  
while (x != y) {  
    if (x < y)  
        y = y - x;  
    else  
        x = x - y;  
}  
write(x);
```

GGT

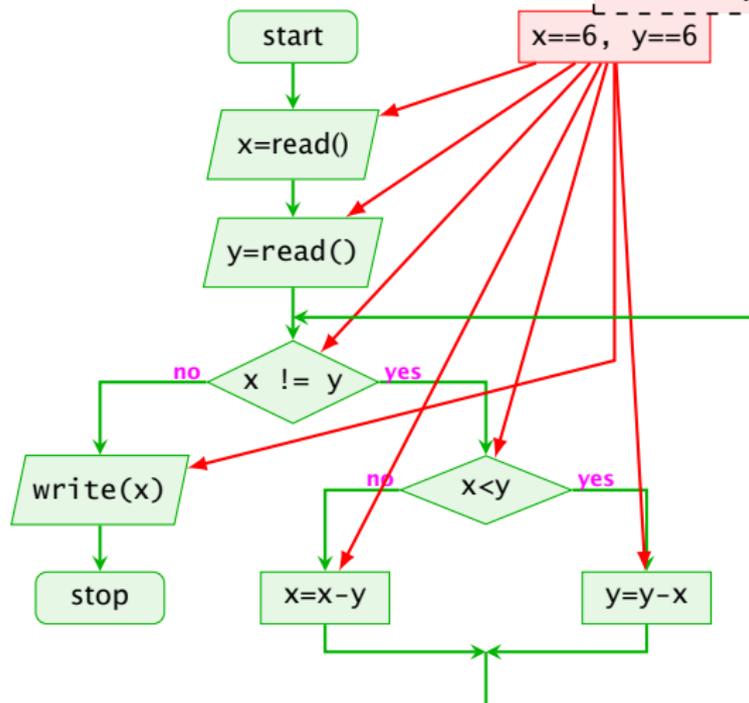


## 4 Kontrollflussdiagramme

- ▶ Die Ausführung des Programms entspricht einem **Pfad** durch das Kontrollflussdiagramm vom Startknoten zum Endknoten.
- ▶ Die Deklaration von Variablen muss man sich am Startknoten vorstellen.
- ▶ Die auf dem Pfad liegenden Knoten (außer Start- und Endknoten) sind Operationen bzw. auszuwertende Bedingungen.
- ▶ Um den Nachfolger an einem Verzweigungsknoten zu bestimmen, muss die Bedingung mit den aktuellen Werten der Variablen ausgewertet werden. (↑**operationelle Semantik**)

## 4 Kontrollflussdiagramme

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



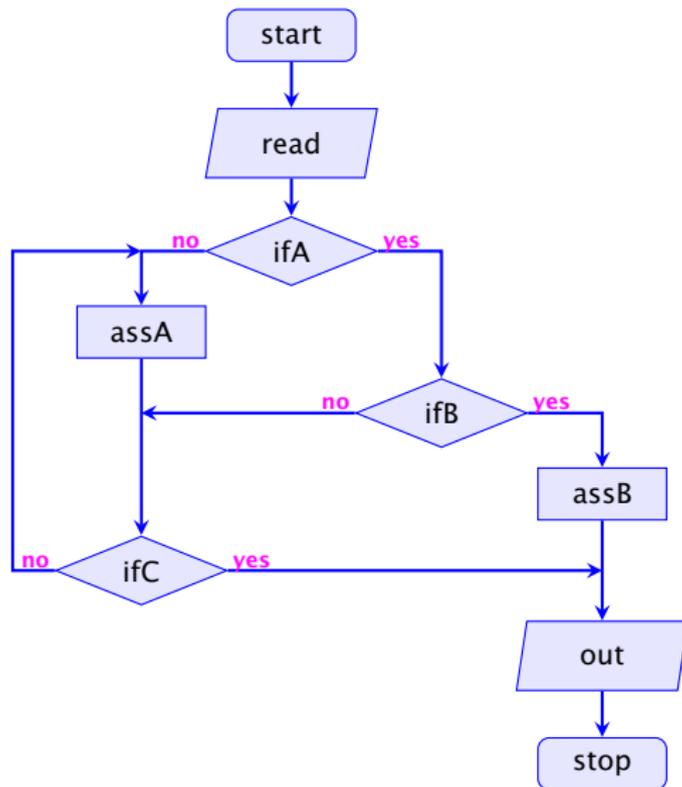
## 4 Kontrollflussdiagramme

- ▶ zu jedem **MiniJava**-Programm lässt sich ein Kontrollflussdiagramm konstruieren;
- ▶ die Umkehrung gilt auch, liegt aber nicht sofort auf der Hand

Die Umkehrung wäre sehr einfach zu bewerkstelligen, wenn wir in einem MiniJava-Programm **goto**-Befehle benutzen dürften, d.h. wenn wir von jedem Punkt zu jedem anderen innerhalb des Programms springen könnten.

Die obige Aussage bedeutet im Prinzip, dass man **goto**-Befehle immer durch geeignete Schleifen ersetzen kann.

## 4 Kontrollflussdiagramme



## 5 Mehr Java

Java ist **statisch typisiert**, d.h., **Variablen**, **Ergebnisse von Ausdrücken**, etc. haben einen **Datentyp**, der schon bei der Kompilierung festgelegt wird.

Java unterscheidet zwei Arten von Typen:

- ▶ Basistypen / Primitive Datentypen  
`byte`, `char`, `short`, `int`, `long`, `float`, `double`, `boolean`
- ▶ Referenzdatentypen  
kann man auch selber definieren

# Beispiel – Statische Typisierung

```
a = 5
a = a + 1
a = "Hello World." # a is now a string
a = a + 1          # runtime error
```

Python

```
int a;
a = 5;
a = "Hello World." // will not compile
```

Java

# 5.1 Basistypen

## Primitive Datentypen

- ▶ Zu jedem Basistypen gibt es eine Menge möglicher **Werte**.
- ▶ Jeder Wert eines Basistyps benötigt den gleichen **Platz**, um ihn im Rechner zu repräsentieren.
- ▶ Der Platz wird in **Bit** gemessen.

Wie viele Werte kann man mit  $n$  Bit darstellen?

# Primitive Datentypen – Ganze Zahlen

Es gibt **vier** Sorten ganzer Zahlen:

<i>Typ</i>	<i>Platz</i>	<i>kleinster Wert</i>	<i>größter Wert</i>
byte	8	-128	127
short	16	-32 768	32 767
int	32	-2 147 483 648	2 147 483 647
long	64	-9 223 372 036 854 775 808	9 223 372 036 854 775 807

Die Benutzung kleinerer Typen wie **byte** oder **short** spart Platz.

# Primitive Datentypen – Ganze Zahlen

Verwenden Sie niemals `l` als `long`-Suffix, da dieses leicht mit `1` verwechselt werden kann.

`_` darf nur **zwischen** Ziffern stehen, d.h. weder am Anfang noch am Ende.

Übung:

Geben Sie eine reguläre Grammatik an, die diese Regeln abbildet

## Literale:

- ▶ dezimale Notation
- ▶ hexadezimale Notation (Präfix `0x` oder `0X`)
- ▶ oktale Notation (Präfix `0`)
- ▶ binäre Notation (Präfix `0b` oder `0B`)
- ▶ Suffix `l` ☹ oder `L` für `long`
- ▶ `'_'` um Ziffern zu gruppieren

## Beispiele

- ▶ `192`, `0b11000000`, `0xC0`, `0300` sind alle gleich
- ▶ `20_000L`, `0xABFF_0078L`
- ▶ `09`, `0xFF` sind ungültig

# Primitive Datentypen – Ganze Zahlen

**Achtung:** `Java` warnt nicht vor Überlauf/Unterlauf!!!

**Beispiel:**

```
1 int x = 2147483647; // groesstes int
2 x = x + 1;
3 write(x);
```

liefert: `-2147483648`

- In realem `Java` kann man bei der Deklaration einer Variablen ihr direkt einen ersten Wert zuweisen (**Initialisierung**).
- Man kann sie sogar (statt am Anfang des Programms) erst an der Stelle deklarieren, an der man sie braucht!

# Primitive Datentypen – Gleitkommazahlen

Es gibt **zwei** Sorten von Gleitkommazahlen:

Typ	Platz	kleinster Wert	größter Wert	signifikante Stellen
float	32	ca. $-3.4 \cdot 10^{38}$	ca. $3.4 \cdot 10^{38}$	ca. 7
double	64	ca. $-1.7 \cdot 10^{308}$	ca. $1.7 \cdot 10^{308}$	ca. 15

$$x = s \cdot m \cdot 2^e \quad \text{mit } 1 \leq m < 2$$

- ▶ Vorzeichen  $s$ : 1 bit
- ▶ reduzierte Mantisse  $m - 1$ : 23 bit (float), 52 bit (double)
- ▶ Exponent  $e$ : 8 bit (float), 11 bit (double)

# Primitive Datentypen – Gleitkommazahlen

## Literale:

- ▶ dezimale Notation.
- ▶ dezimale Exponentialschreibweise (e, E für Exponent) Mantisse und Exponent sind dezimal; Basis für Exponent ist 10;
- ▶ hexadezimale Exponentialschreibweise. (Präfix 0x oder 0X, p oder P für Exponent) Mantisse ist hexadezimal; Exponent ist dezimal und muß vorhanden sein; Basis für Exponent ist 2;
- ▶ Suffix f oder F für float, Suffix d oder D für double (default is double) In der hexadezimalen Notation, gibt der Exponent die Anzahl der Bitpositionen an, um die das Komma verschoben wird.

## Beispiele

- ▶ 640.5F == 0x50.1p3f
- ▶ 3.1415 == 314.15E-2
- ▶ 0x1e3\_dp0, 1e3d 0x1e3d ist ein int und keine Gleitkommazahl
- ▶ 0x1e3d, 1e3\_d, 0x50.1 1e3\_d ist ungültig, da '\_' nicht zwischen 2 Ziffern steht (d ist keine Ziffer sondern das double-Suffix)

# Primitive Datentypen – Gleitkommazahlen

- ▶ Überlauf/Unterlauf bei Berechnungen liefert **Infinity**, bzw. **-Infinity**
- ▶ Division Null durch Null, Wurzel aus einer negativen Zahl etc. liefert **NaN**

## Weitere Basistypen

Typ	Platz	Werte
boolean	1	true, false
char	16	alle(?) Unicode-Zeichen

Unicode ist ein Zeichensatz, der alle irgendwo auf der Welt gängigen Alphabete umfasst, also zum Beispiel:

- ▶ die Zeichen unserer Tastatur (inklusive Umlaute);
- ▶ die chinesischen Schriftzeichen;
- ▶ die ägyptischen Hieroglyphen ...

### Literale:

- ▶ `char`-Literale schreibt man in Hochkomma: `'A'`, `'\u00ED'`, `';`, `'\n'`.
- ▶ `boolean`-Literale sind `true` und `false`.

Die ursprüngliche Idee war, dass `char` alle Unicodezeichen enthält. Nach der Einführung von `Java`, hat sich der Unicodestandard geändert. Deshalb kann ein `char` nur Zeichen der sogenannten **Basic Multilingual Plane** speichern. Andere Unicodezeichen werden über Strings codiert.

## 5.2 Strings

Der Datentyp `String` für Wörter ist ein Referenzdatentyp (genauer eine `Klasse` (dazu kommen wir später)).

Hier nur drei Eigenschaften:

- ▶ Literale vom Typ `String` haben die Form `"Hello World!"`;
- ▶ Man kann Wörter in Variablen vom Typ `String` abspeichern;
- ▶ Man kann Wörter mithilfe des Operators `'+'` konkatenieren.

# Beispiel

```
String s0 = "";  
String s1 = "Hel";  
String s2 = "lo Wo";  
String s3 = "rld!";  
  
write(s0 + s1 + s2 + s3);
```

...liefert: Hello World!

## 5.3 Auswertung von Ausdrücken

**Funktionen** in **Java** bekommen **Parameter**/Argumente als Input, und liefern als Output den Wert eines vorbestimmten Typs. Zum Beispiel könnte man eine Funktion

```
int min(int a, int b)
```

implementieren, die das Minimum ihrer Argumente zurückliefert.

**Operatoren** sind spezielle vordefinierte Funktionen, die in **Infix**-Notation geschrieben werden (wenn sie binär sind):

```
a + b = +(a, b)
```

Funktionen, werden hier nur eingeführt, weil wir sie bei der Ausdrucksauswertung benutzen möchten. Eine detaillierte Einführung erfolgt später.

## 5.3 Auswertung von Ausdrücken

Ein **Ausdruck** ist eine Kombination von Literalen, Operatoren, Funktionen, Variablen und Klammern, die verwendet wird, um einen Wert zu berechnen.

**Beispiele:** (x z.B. vom Typ `int`)

- ▶ `7 + 4`
- ▶ `3 / 5 + 3`
- ▶ `min(3,x) + 20`
- ▶ `x = 7`
- ▶ `x *= 2`

Unäre +/--Operatoren konvertieren `byte`, `short`, `char` zuerst nach `int`.

Man kann keinen legalen Ausdruck bilden, bei der die Assoziativität der Postfix-Operatoren (Gruppe Priorität 2) eine Rolle spielen würde.

## Unäre Operatoren:

<i>symbol</i>	<i>name</i>	<i>types</i>	<i>L/R</i>	<i>level</i>
<code>++</code>	Post-inkrement	(var) Zahl, char	keine	2
<code>--</code>	Post-dekrement	(var) Zahl, char	keine	2
<code>++</code>	Pre-inkrement	(var) Zahl, char	rechts	3
<code>--</code>	Pre-dekrement	(var) Zahl, char	rechts	3
<code>+</code>	unäres Plus	Zahl, char	rechts	3
<code>-</code>	unäres Minus	Zahl, char	rechts	3
<code>!</code>	Negation	boolean	rechts	3

Die Spalte „L/R“ beschreibt die **Assoziativität** des Operators.

Die Spalte „level“ die Priorität.

Im Folgenden sind (für binäre Operatoren) beide Operanden jeweils vom gleichen Typ.

„Zahl“ steht hier für einen der Zahltypen `byte`, `short`, `int`, `long`, `float` oder `double`.

Diese Beschreibung der Vorrangregeln in Form von Prioritäten für Operatoren findet sich nicht im Java Reference Manual. Dort wird nur die formale kontextfreie Grammatik von Java beschrieben. Die Vorrangregeln leiten sich daraus ab und erleichtern den Umgang mit Ausdrücken, da man nicht in die formale Grammatik schauen muß um einen Ausdruck zu verstehen.

Es gibt im Internet zahlreiche teils widersprüchliche Tabellen, die die Vorrangregeln von Java-Operatoren beschreiben :( Die gesamte Komplexität der Ausdruckssprache von Java läßt sich wahrscheinlich nicht in dieses vereinfachte Schema pressen.

# Prefix- und Postfixoperator

- ▶ Die Operatoranwendungen `++x` und `x++` inkrementieren beide den Wert der Variablen `x` (als **Seiteneffekt**).
- ▶ `++x` tut das, **bevor** der Wert des Ausdrucks ermittelt wird (**Pre-Inkrement**).
- ▶ `x++` tut das, **nachdem** der Wert ermittelt wurde (**Post-Inkrement**).
- ▶ `b = x++;` entspricht:

```
b = x;  
x = x + 1;
```

- ▶ `b = ++x;` entspricht:

```
x = x + 1;  
b = x;
```

Die Entsprechung gilt z.B. für `ints`. Für `shorts` müßte es heißen:

```
b = x;  
x = (short) (x + 1);
```

da `x = x + 1` nicht kompiliert wenn `x` ein `short` ist.

`(short)` ist hier ein **Typecast-Operator**, den wir später kennenlernen.

# Operatoren

## Binäre arithmetische Operatoren:

byte, short, char werden nach int konvertiert

<i>symbol</i>	<i>name</i>	<i>types</i>	<i>L/R</i>	<i>level</i>
*	Multiplikation	Zahl, char	links	4
/	Division	Zahl, char	links	4
%	Modulo	Zahl, char	links	4
+	Addition	Zahl, char	links	5
-	Subtraktion	Zahl, char	links	5

## Konkatenation

<i>symbol</i>	<i>name</i>	<i>types</i>	<i>L/R</i>	<i>level</i>
+	Konkatenation	String	links	5

Für Referenzdatentypen (kommt später)  
vergleichen die Operatoren `==` und `!=`  
nur die Referenzen.

## Vergleichsoperatoren:

<i>symbol</i>	<i>name</i>	<i>types</i>	<i>L/R</i>	<i>level</i>
>	größer	Zahl, char	keine	7
>=	größergleich	Zahl, char	keine	7
<	kleiner	Zahl, char	keine	7
<=	kleinergleich	Zahl, char	keine	7
==	gleich	alle	links	8
!=	ungleich	alle	links	8

## Boolsche Operatoren:

<i>symbol</i>	<i>name</i>	<i>types</i>	<i>L/R</i>	<i>level</i>
&&	Und-Bedingung	boolean	links	12
	Oder-Bedingung	boolean	links	13

## Zuweisungsoperatoren:

<i>symbol</i>	<i>name</i>	<i>types</i>	<i>L/R</i>	<i>level</i>
=	Zuweisung	(links var) alle	rechts	15
*=, /=, %= +=, -=	Zuweisung	(links var) alle	rechts	15

Für die letzte Form gilt:

$$v \Leftarrow a \iff v = (\text{type}(v)) (v \circ a)$$

# Operatoren

Ein Seiteneffekt sind Änderungen von Zuständen/Variablen, die durch die Auswertung des Ausdrucks entstehen.

## Warnung:

- ▶ Eine Zuweisung  $x = y$ ; ist in Wahrheit ein **Ausdruck**.
- ▶ Der Wert ist der Wert der rechten Seite.
- ▶ Die Modifizierung der Variablen  $x$  erfolgt als **Seiteneffekt**.
- ▶ Das Semikolon ';' hinter einem Ausdruck wirft nur den Wert weg.

## Fatal für Fehler in Bedingungen:

```
boolean x = false;  
if (x = true)  
    write("Sorry! This must be an error ...");
```

In C ist diese Art des Fehlers noch wesentlich häufiger, da auch z.B.  $x = 1$  (für `int x`) in der Bedingung vorkommen kann. Das Ergebnis des Ausdrucks (1) wird in den booleschen Wert `true` konvertiert. Letzteres ist in **Java** nicht möglich.

In **Java** kann man durch das ';' aus den meisten Ausdrücken eine Anweisung machen, die nur den Seiteneffekt des Ausdrucks durchführt.

## 5.3 Auswertung von Ausdrücken

### Assoziativität

- ▶ Die Assoziativität entscheidet über die Reihenfolge bei Operatoren gleicher Priorität. (links = der linkeste Operator wird zuerst ausgeführt)
- ▶ Alle Operatoren einer Prioritätsgruppe haben dieselbe Assoziativität.
- ▶ Bis auf Zuweisungsoperatoren (=, +=, etc.) sind alle binären Operatoren linksassoziativ.
- ▶ unäre Operatoren, die ihr Argument rechts erwarten sind rechtsassoziativ
- ▶ unäre Operatoren, die ihr Argument links erwarten (postfix-Operatoren ++, --) sind linksassoziativ
- ▶ Der ternäre Bedingungsoperator (später) ist rechtsassoziativ

## 5.3 Auswertung von Ausdrücken

Die Auswertung eines Ausdrucks liefert

- ▶ eine Variable (**var**),
- ▶ einen reinen Wert (**val**) oder
- ▶ void (**void**)

In den ersten beiden Fällen hat der Ausdruck dann einen

- ▶ Typ, z.B.: **int**, und einen
- ▶ Wert, z.B.: **42**

Für z.B. Zuweisungen muss die Auswertung des Ausdrucks auf der linken Seite eine Variable ergeben!!!

## 5.3 Auswertung von Ausdrücken

In **Java** werden Unterausdrücke von links nach rechts ausgewertet. D.h. um den Wert einer Operation zu berechnen:

- ▶ werte (rekursiv) alle Operanden von links nach rechts aus
- ▶ führe die Operation auf den Resultaten aus

**Ausnahmen:** `||`, `&&`, und der ternäre Bedingungsoperator `?:`, werten nicht alle Operanden aus (**Kurzschlussauswertung**).

**Man sollte nie Ausdrücke formulieren, deren Ergebnis von der Auswertungsreihenfolge abhängt!!!**

Eine Kurzschlussauswertung ist natürlich ok. Dafür gibt es sehr nützliche Anwendungen.

In **C/C++**, ist die Auswertungsreihenfolge nicht definiert, d.h., sie ist compilerabhängig.

Den Bedingungsoperator lernen wir später kennen.

## 5.3 Auswertung von Ausdrücken

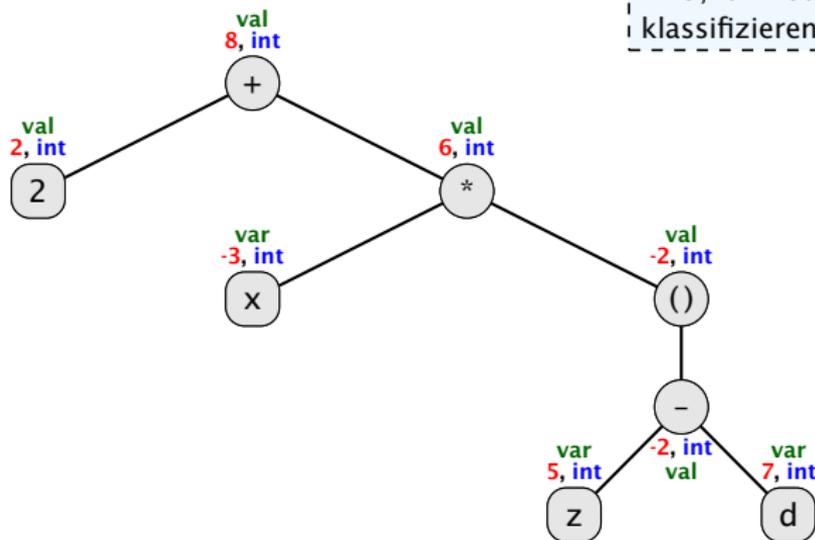
Im Folgenden betrachten wir Klammern als einen Operator der nichts tut:

<i>symbol</i>	<i>name</i>	<i>types</i>	<i>L/R</i>	<i>level</i>
()	Klammerung	alle	links	0

# Beispiel: $2 + x * (z - d)$

Punkt geht vor Strichrechnung.

Ganzahliliterale sind vom Typ `int`, wenn nicht z.B. ein `L` angehängt wird, um das Literal als `long` zu klassifizieren.



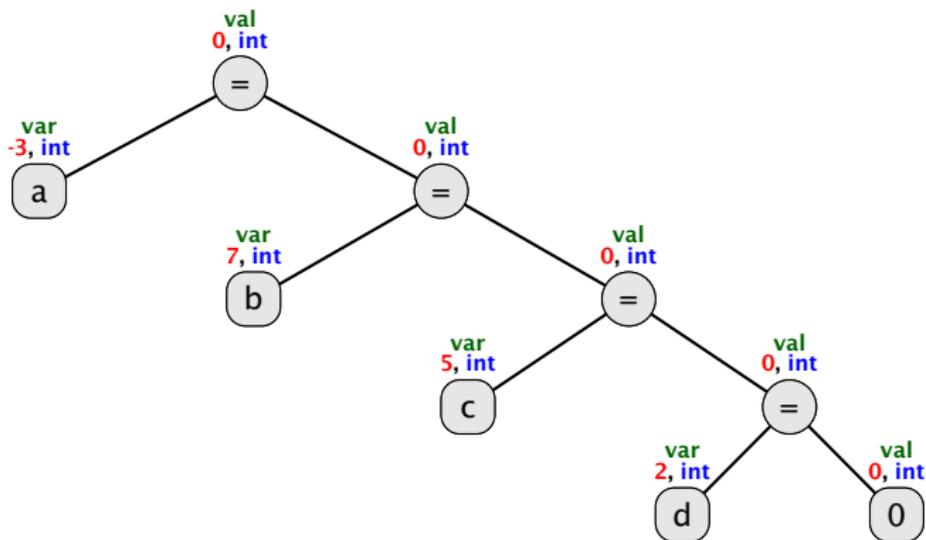
x

d

z

# Beispiel: $a = b = c = d = 0$

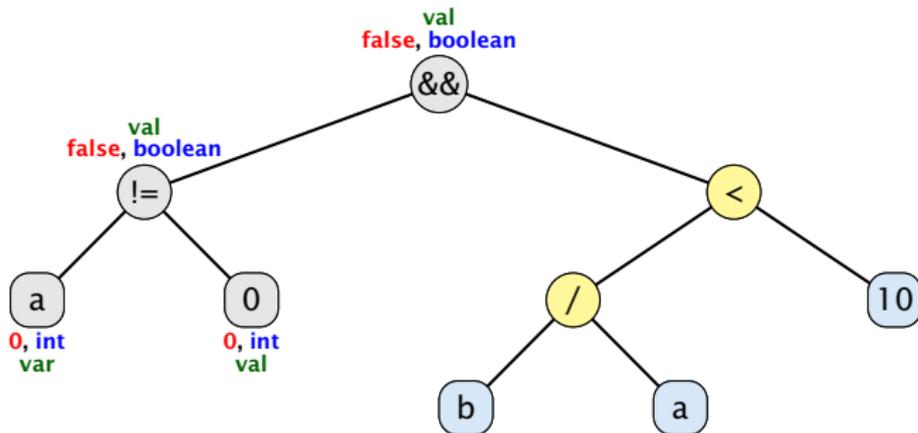
Das funktioniert nur, da der Zuweisungsoperator rechtsassoziativ ist.



a     b     c     d

# Beispiel: $a \neq 0 \ \&\& \ b/a < 10$

Die vollständige Auswertung der Operanden würde hier zu einem Laufzeitfehler führen (Division durch Null).  
Mit Kurzschlussauswertung ist alles ok.



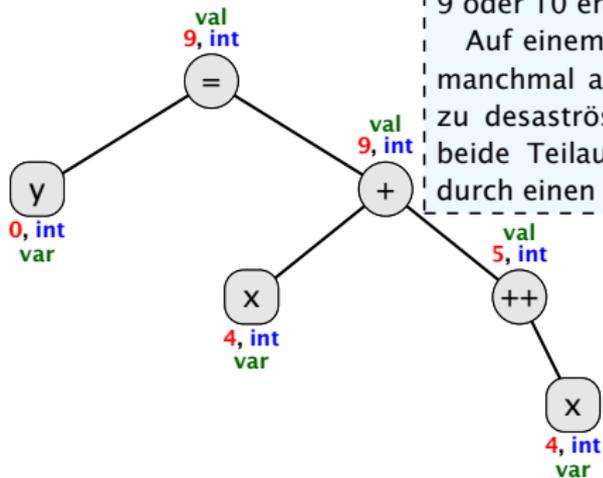
a 0

b 4

## Beispiel: $y = x + ++x$

In C ist die Reihenfolge der Auswertung von Unterausdrücken nicht definiert. Auf einem sequentiellen Rechner hängt die Reihenfolge vom Compiler ab und in diesem Beispiel könnte dies das Resultat 9 oder 10 ergeben.

Auf einem Parallelrechner können Teilausdrücke manchmal auch parallel ausgewertet werden, was zu desaströsen Konsequenzen führen kann, falls beide Teilausdrücke eine Variable enthalten, die durch einen Seiteneffekt verändert wird.



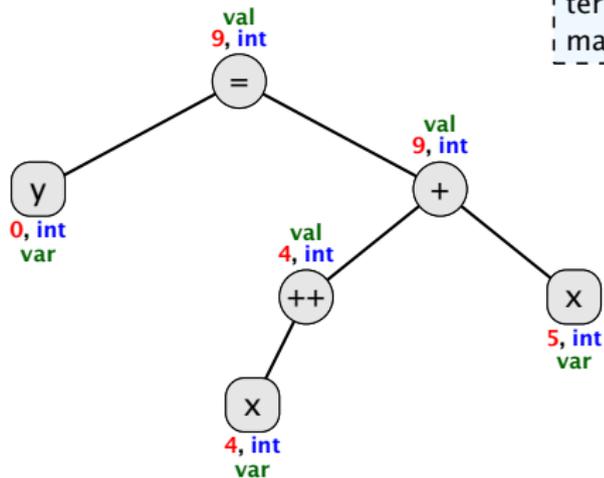
x

y

# Beispiel: $y = x++ + x$

Der Postfix-Operator ändert die Variable nach dem der Wert des **Teilausdrucks** bestimmt wurde.

Wenn die Variable im Ausdruck später nochmal ausgewertet wird, bekommt man den neuen Wert.



x 5

y 9

# Impliziter Typecast

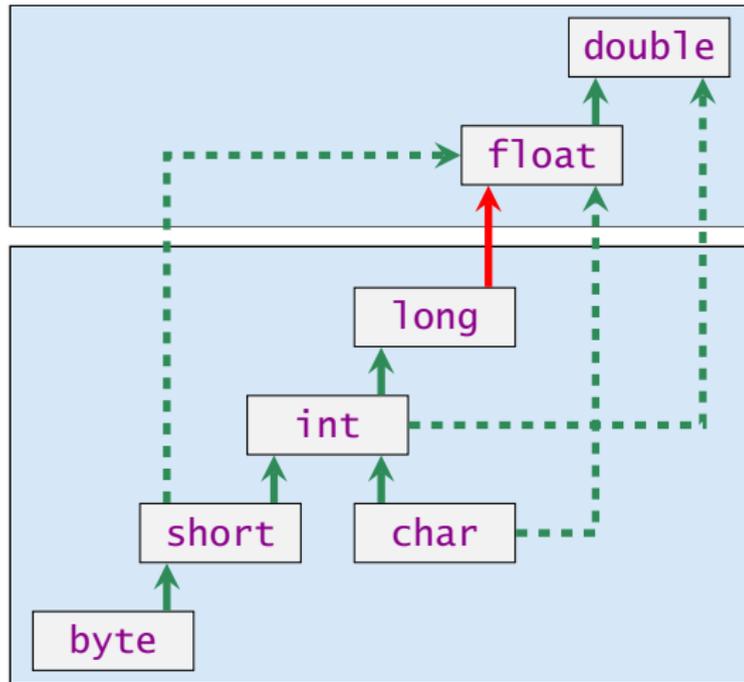
Wenn ein Ausdruck vom **TypA** an einer Stelle verwendet wird, wo ein Ausdruck vom **TypB** erforderlich ist, wird

- ▶ entweder der Ausdruck vom **TypA** in einen Ausdruck vom **TypB** **gecastet** (**impliziter Typecast**),
- ▶ oder ein Compilerfehler erzeugt, falls dieser Cast nicht (automatisch) erlaubt ist.

## Beispiel: Zuweisung

```
long x = 5;  
int y = 3;  
x = y; // impliziter Cast von int nach long
```

# Erlaubte Implizite Typecasts – Numerische Typen



Gleitkommazahlen

Man nennt diese Art der Casts, **widening conversions**, da der Wertebereich im Allgemeinen erweitert wird.

ganze Zahlen, char

Keine Typumwandlung zwischen `boolean` und Zahltypen (weder implizit noch explizit).

Konvertierung von `long` nach `double` oder von `int` nach `float` kann Information verlieren wird aber **automatisch** durchgeführt.

## Welcher Typ wird benötigt?

Operatoren sind üblicherweise **überladen**, d.h. ein Symbol (+, -, ...) steht in Abhängigkeit der Parameter (Argumente) für unterschiedliche Funktionen.

+ : int → int

+ : long → long

+ : float → float

+ : double → double

+ : int × int → int

+ : long × long → long

+ : float × float → float

+ : double × double → double

+ : String × String → String

Es gibt keinen +-Operator für **short**, **byte**, **char**.

Der +-Operator für Strings macht Konkatination.

Der Compiler muss in der Lage sein **während der Compilierung** die richtige Funktion zu bestimmen.

# Impliziter Typecast

Der Compiler wertet nur die Typen des Ausdrucksbaums aus.

- ▶ Für jeden inneren Knoten wählt er dann die geeignete Funktion (z.B.  $+ : \text{long} \times \text{long} \rightarrow \text{long}$  falls ein  $+$ -Knoten zwei  $\text{long}$ -Argumente erhält).
- ▶ Falls keine passende Funktion gefunden wird, versucht der Compiler durch **implizite Typecasts** die Operanden an eine Funktion anzupassen.
- ▶ Dies geschieht auch für selbstgeschriebene Funktionen (z.B.  $\text{min}(\text{int } a, \text{int } b)$  und  $\text{min}(\text{long } a, \text{long } b)$ ).
- ▶ Der Compiler nimmt die Funktion mit der speziellsten **Signatur**.

# Speziellste Signatur

1. Der Compiler bestimmt zunächst alle Funktionen, die passen könnten (d.h. die vorliegenden Typen können durch **widening conversions** in die Argumenttypen der Funktion umgewandelt werden).
2. Eine Funktion  $f_1$  ist spezifischer als eine andere  $f_2$ , wenn die Argumenttypen von  $f_1$  auch für einen Aufruf von  $f_2$  benutzbar sind (z.B. `min(int, long)` spezifischer als `min(long, long)` aber nicht spezifischer als `min(long, int)`).  
Dieses definiert eine partielle Ordnung auf der Menge der Funktionen.
3. Unter den möglichen Funktionen (aus Schritt 1) wird ein kleinste Element bzgl. dieser partiellen Ordnung gesucht. Falls genau ein kleinstes Element existiert, ist dies die gesuchte Funktion. Andernfalls ist der Aufruf ungültig. (Beachte: Rückgabetypp spielt für Funktionsauswahl keine Rolle).

# Ordnungsrelationen

Relation  $\preceq$ :  $\text{TypA} \preceq \text{TypB}$  falls  $\text{TypA}$  nach  $\text{TypB}$  (implizit) gecasted werden kann:

- ▶ **reflexiv:**  $T \preceq T$
- ▶ **transitiv:**  $T_1 \preceq T_2 \wedge T_2 \preceq T_3 \Rightarrow T_1 \preceq T_3$
- ▶ **antisymmetrisch:**  $T_1 \preceq T_2 \wedge T_2 \preceq T_1 \Rightarrow T_1 = T_2$

d.h.,  $\preceq$  definiert **Halbordnung auf der Menge der Typen**.

Relation  $\preceq_k$ :  $(T_1, \dots, T_k) \preceq_k (T'_1, \dots, T'_k)$  falls  $T_i \preceq T'_i$  für alle  $i \in \{1, \dots, k\}$ :

- ▶ **reflexiv:**  $\mathcal{T} \preceq_k \mathcal{T}$
- ▶ **transitiv:**  $\mathcal{T}_1 \preceq_k \mathcal{T}_2 \wedge \mathcal{T}_2 \preceq_k \mathcal{T}_3 \Rightarrow \mathcal{T}_1 \preceq_k \mathcal{T}_3$
- ▶ **antisymmetrisch:**  $\mathcal{T}_1 \preceq_k \mathcal{T}_2 \wedge \mathcal{T}_2 \preceq_k \mathcal{T}_1 \Rightarrow \mathcal{T}_1 = \mathcal{T}_2$

d.h.,  $\preceq_k$  definiert **Halbordnung auf Menge der  $k$ -Tupel von Typen**

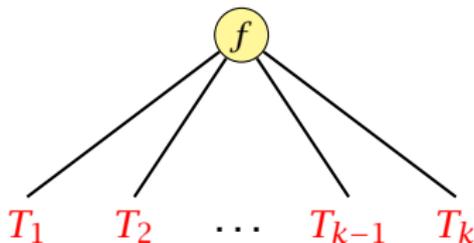
Wir betrachten Relation auf der Menge von Parametertupeln für die  $f$  implementiert ist. Aus Antisymmetrie folgt, dass keine zwei Funktionen das gleiche  $k$ -Tupel an Parametern erwarten.

$R_1 \quad f(\mathcal{T}_1)$

$R_2 \quad f(\mathcal{T}_2)$

$\vdots$

$R_\ell \quad f(\mathcal{T}_\ell)$



$\mathcal{T}_1, \dots, \mathcal{T}_\ell$  sind  $k$ -Tupel von Typen für die eine Definition von  $f$  existiert.

$\mathcal{T} = (T_1, \dots, T_k)$  ist das  $k$ -tupel von Typen mit dem  $f$  aufgerufen wird.

Menge aller möglichen Funktionen/Tupel:

$$M := \{\mathcal{T}_i \mid \mathcal{T} \preceq_k \mathcal{T}_i\} .$$

Wähle **kleinstes** Element aus  $M$  falls  $M$  ein eindeutig kleinstes Element besitzt (sonst Compilerfehler).

# Impliziter Typecast – Numerische Typen

Angenommen wir haben Funktionen

```
int min(int a, int b)
```

```
float min(float a, float b)
```

```
double min(double a, double b)
```

definiert.

```
1 long a = 7, b = 3;  
2 double d = min(a, b);
```

würde die Funktion `float min(float a, float b)` aufrufen.

# Impliziter Typecast

Bei Ausdrücken mit Seiteneffekten (Zuweisungen, ++ , --) gelten andere Regeln:

## Beispiel: Zuweisungen

= : byte\* × byte → byte  
= : char\* × char → char  
= : short\* × short → short  
= : int\* × int → int  
= : long\* × long → long  
= : float\* × float → float  
= : double\* × double → double

Es wird nur der Parameter konvertiert, der nicht dem Seiteneffekt unterliegt.

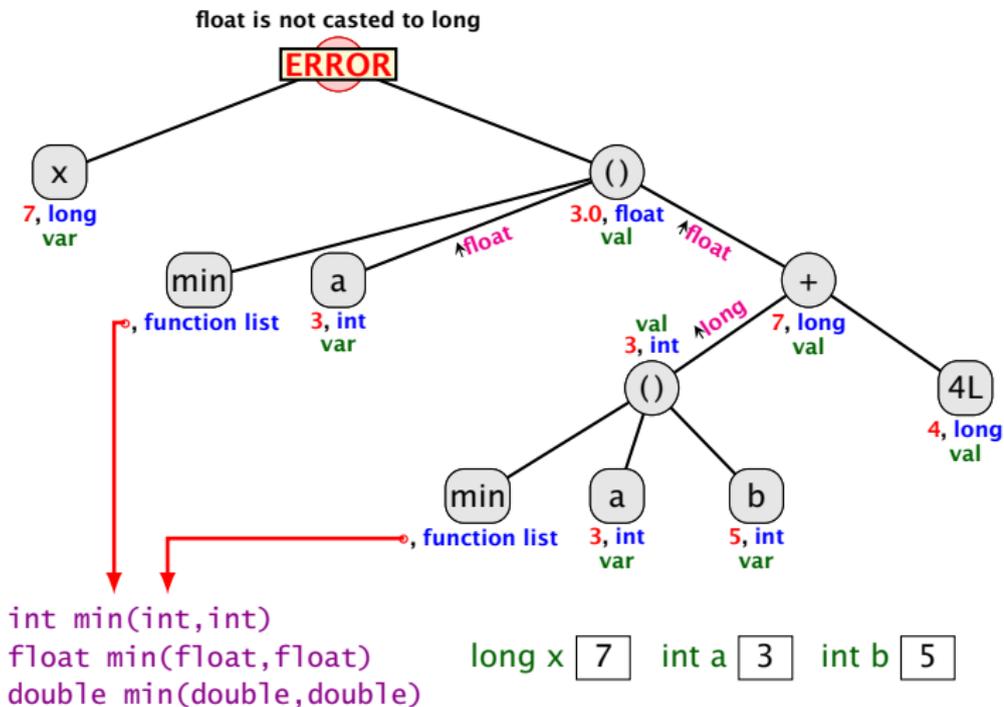
## 5.3 Auswertung von Ausdrücken

### Der Funktionsaufrufoperator:

<i>symbol</i>	<i>name</i>	<i>types</i>	<i>L/R</i>	<i>level</i>
()	Funktionsaufruf	Funktionsname, *	links	1

Wir modellieren den Funktionsaufrufoperator hier als einen Operator, der beliebig viele Argumente entgegennimmt. Das erste Argument ist der Funktionsname, und die folgenden Argumente sind die Parameter der Funktion. Üblicherweise hat der Funktionsaufrufoperator nur zwei Operanden: den Funktionsnamen, und eine Argumentliste.

# Beispiel: $x = \min(a, \min(a,b) + 4L)$



**Achtung:** Dieses ist eine sehr vereinfachte und teilweise inkorrekte Darstellung. Der eigentliche Prozess, der vom Funktionsnamen zu eigentlichen Funktion führt ist sehr kompliziert. **function list** ist auch kein Typ in **Java**.

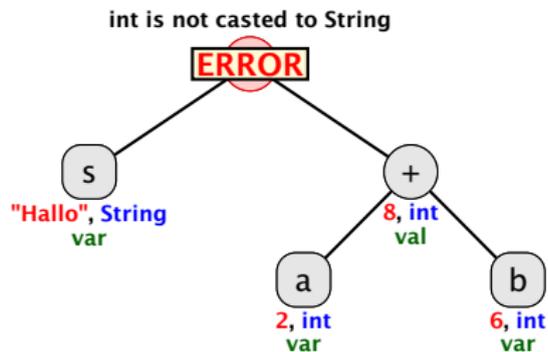
# Impliziter Typecast – Strings

## Spezialfall

- ▶ Falls beim Operator `+` ein Typ vom Typ `String` ist, wird der andere auch in einen `String` umgewandelt.  
⇒ Stringkonkatenation.
- ▶ Jeder Typ in `Java` besitzt eine Stringrepräsentation.

**Funktioniert nicht bei selbstgeschriebenen Funktionen.**

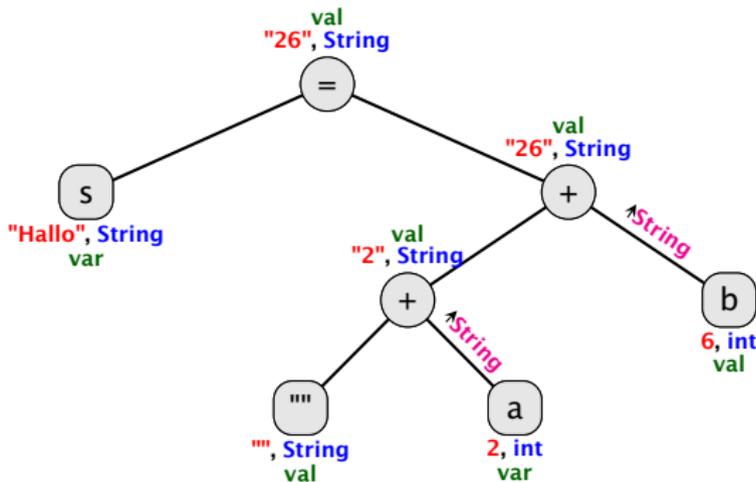
# Beispiel: $s = a + b$



String s  → "Hallo"      a       b

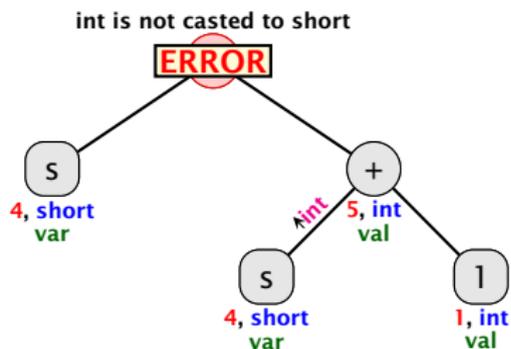
Beispiel: `s = "" + a + b`

Strings are immutable! Falls eine weitere Referenz auf "Hallo" verweist, hat sich für diese nichts geändert.



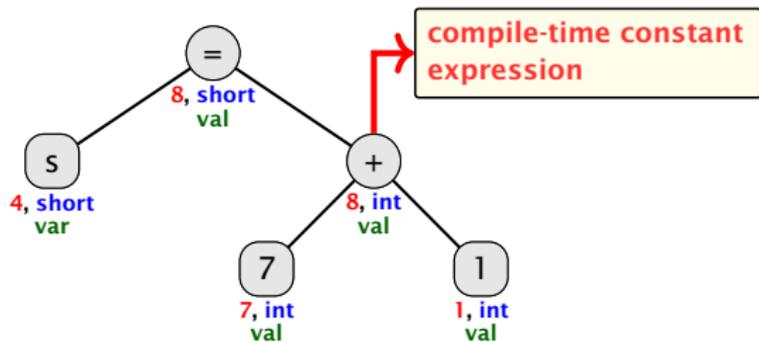
**Achtung: vereinfachte Darstellung!!!**  
Eigentlich arbeitet Java mit Objekten vom Typ `StringBuffer` um den `+` Operator zu realisieren...

# Beispiel: $s = s + 1$



short s 4

# Beispiel: $s = 7 + 1$



short s 8

Wenn der `int`-Ausdruck, der zugewiesen werden soll, zu Compilerzeit bekannt ist, und er in einen `short` „passt“, wird der Cast von `int` nach `short` durchgeführt.

Funktioniert nicht für `long`-Ausdrücke, d.h., `byte b = 4L`; erzeugt einen Compilerfehler.

# Expliziter Typecast

<i>symbol</i>	<i>name</i>	<i>type</i>	<i>L/R</i>	<i>level</i>
(type)	typecast	Zahl, char	rechts	3

## Beispiele mit Datenverlust

- ▶ `short s = (short) 23343445;`

Die obersten bits werden einfach weggeworfen...

- ▶ `double d = 1.5;`  
`short s = (short) d;`  
`s` hat danach den Wert `1`.

## ...ohne Datenverlust:

- ▶ `int x = 5;`  
`short s = (short) x;`

Man kann einen cast zwischen Zahltypen erzwingen (evtl. mit Datenverlust). Typecasts zwischen Referenzdatentypen kommen später.

## 5.4 Arrays

Oft müssen viele Werte gleichen Typs gespeichert werden.

### Idee:

- ▶ Lege sie konsekutiv ab!
- ▶ Greife auf einzelne Werte über ihren Index zu!

Feld:	17	3	-2	9	0	1
Index:	0	1	2	3	4	5

# Beispiel

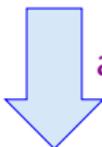
```
1 int[] a; // Deklaration
2 int n = read();
3
4 a = new int[n]; // Anlegen des Felds
5 int i = 0;
6 while (i < n) {
7     a[i] = read();
8     i = i + 1;
9 }
```

Einlesen eines Feldes

# Beispiel

- ▶ `type[] name;` deklariert eine Variable für ein Feld (**array**), dessen Elemente vom Typ `type` sind.
- ▶ Alternative Schreibweise:  
`type name[];`
- ▶ Das Kommando `new` legt ein Feld einer gegebenen Größe an und liefert einen **Verweis** darauf zurück:

a 



`a = new int[6];`

a 



Alles was mit `new` angelegt wird, ist vorinitialisiert. Referenztypen zu `null`, Zahltypen zu `0`, boolean zu `false`.

Man kann auch leere Felder anlegen: `new int[0]`.

# Was ist eine Referenz?

Eine Referenzvariable speichert eine Adresse; an dieser Adresse liegt der eigentliche Inhalt der Variablen.



Wir können die Referenz nicht direkt manipulieren (nur über den **new**-Operator, oder indem wir eine andere Referenz zuweisen).

Eine Referenz zeigt dadurch nie auf einen beliebigen Ort im Speicher; sie zeigt immer auf ein gültiges Objekt oder auf das **null**-Objekt.

Wir geben üblicherweise nie den Wert einer Referenzvariablen an, sondern symbolisieren diesen Wert durch einen Pfeil auf die entsprechenden Daten.

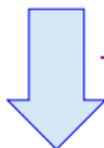
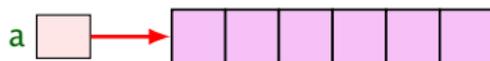
a:

Adresse	Inhalt
⋮	⋮
0000 0127	
0000 0128	0000 012C
0000 0129	
0000 012A	
0000 012B	
0000 012C	0000 0004
0000 012D	0000 0003
0000 012E	0000 0000
0000 012F	0000 0009
0000 0130	
⋮	⋮

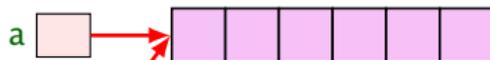
A red arrow originates from the 'a:' label and points to the memory address '0000 012C' in the 'Adresse' column of the table. Another red arrow points from the '0000 012C' cell in the 'Inhalt' column back to the 'a:' label, forming a loop.

## 5.4 Arrays

- ▶ Der Wert einer Feld-Variable ist also ein Verweis!!!
- ▶ `int[] b = a;` kopiert den Verweis der Variablen `a` in die Variable `b`:



`int[] b = a;`



Insbesondere beim Kopieren von Feldern (und anderen Referenztypen) muss man sich dessen immer bewusst sein.

- ▶ **Alle nichtprimitive Datentypen sind Referenztypen, d.h., die zugehörige Variable speichert einen Verweis!!!**

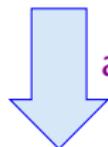
## 5.4 Arrays

- ▶ Die Elemente eines Feldes sind von 0 an durchnummeriert.
- ▶ Die Anzahl der Elemente des Feldes `name` ist `name.length`.
- ▶ Auf das  $i$ -te Element greift man mit `name[i]` zu.
- ▶ Bei jedem Zugriff wird überprüft, ob der Index erlaubt ist, d.h. im Intervall  $\{0, \dots, \text{name.length}-1\}$  liegt.
- ▶ Liegt der Index außerhalb des Intervalls, wird eine `ArrayIndexOutOfBoundsException` ausgelöst (↑`Exceptions`).

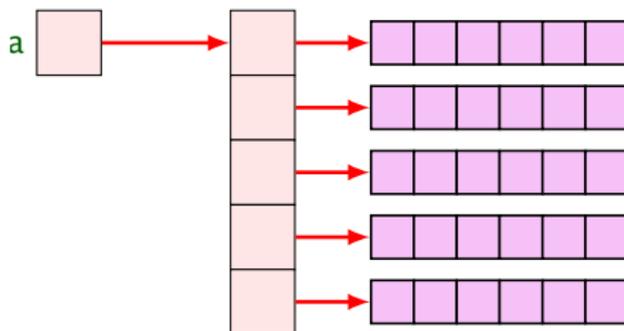
Sie sollten beim Programmieren möglichst nie diese Exception bekommen. In anderen Sprachen (z.B. C/C++) gibt es aus Effizienzgründen keine Überprüfung. Wenn Sie in einer solchen Sprache einen solchen Fehler verursachen, ist der sehr schwierig zu finden.

# Mehrdimensionale Felder

- ▶ **Java** unterstützt direkt nur eindimensionale Felder.
- ▶ ein zweidimensionales Feld ist ein Feld von Feldern...



`a = new int[5][6];`



# Der new-Operator

So etwas wie `new int[3][][4]` macht keinen Sinn, da die Größe dieses Typs nicht vom Compiler bestimmt werden kann.

<i>symbol</i>	<i>name</i>	<i>types</i>	<i>L/R</i>	<i>level</i>
<code>new</code>	new	Typ, Konstruktor	links	1

Erzeugt ein Objekt/Array und liefert eine Referenz darauf zurück.

1. Version: Erzeugung eines Arrays (Typ ist Arraytyp)

- ▶ `new int[3][7];` oder auch
- ▶ `new int[3][];` (ein Array, das 3 Verweise auf `int` enthält)
- ▶ `new String[10];`
- ▶ `new int[]{1,2,3};` (ein Array mit den `ints` 1, 2, 3)

2. Version: Erzeugung eines Objekts durch Aufruf eines Konstruktors

- ▶ `String s = new String("Hello World!");`

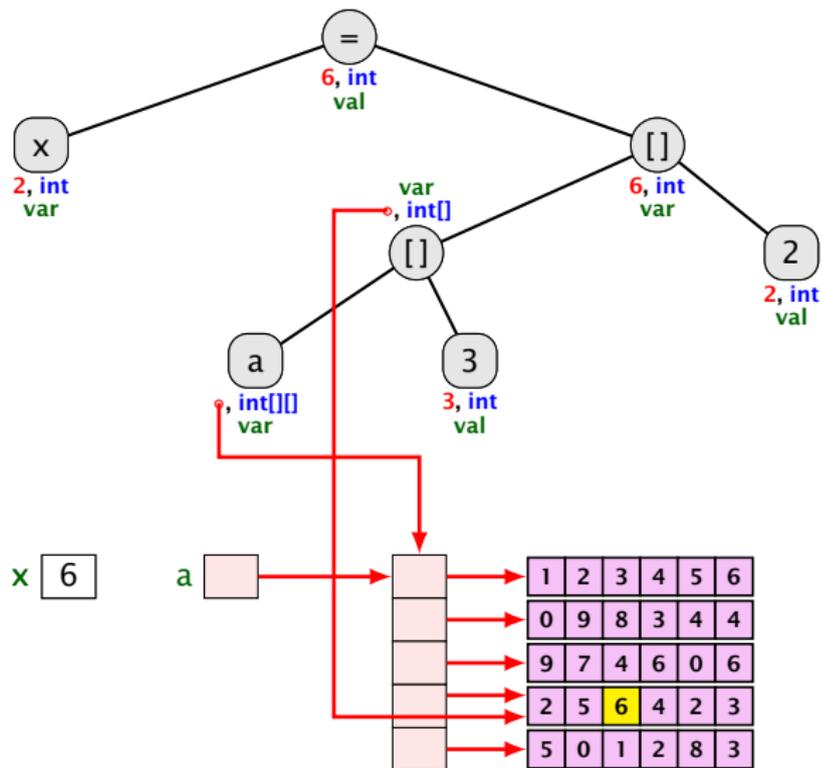
Was genau ein Konstruktor ist kommt später.

# Der Index-Operator

<i>symbol</i>	<i>name</i>	<i>types</i>	<i>L/R</i>	<i>level</i>
<code>[]</code>	index	array, int	links	1

Zugriff auf ein Arrayelement.

# Beispiel: $x = a[3][2]$



# Der .-Operator

<i>symbol</i>	<i>name</i>	<i>types</i>	<i>L/R</i>	<i>level</i>
.	member access	Array/Objekt/Class, Member	links	1

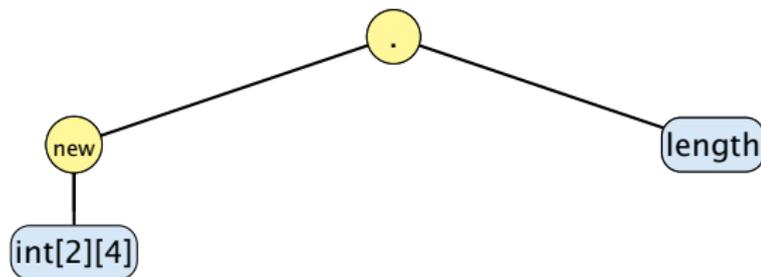
Zugriff auf Member.

## Beispiel:

- ▶ `x = new int[2][4].length`  
x hat dann den Wert 2.

## Beispiel: `new int[2][4].length`

Das Parsing für den `new`-Operator passt nicht in das Schema:



Beachte den Unterschied zwischen `new int[2][3]` und `(new int[2])[3]`. Bei letzterem ist das zweite Klammerpaar ein Index-Operator während es beim ersten Ausdruck zum Typ gehört.

# Arrayinitialisierung

1. `int[] a = new int[3];`  
`a[0] = 1; a[1] = 2; a[2] = 3;`
2. `int[] a = new int[]{ 1, 2, 3};`
3. `int[] a = new int[3]{ 1, 2, 3};`
4. `int[] a = { 1, 2, 3};`
5. `char[][] b = { { 'a', 'b' }, new char[3], {} };`
6. `char[][] b;`  
`b = new char[][]{ { 'a', 'b' }, new char[3], {} };`
7. `char[][] b;`  
`b = { { 'a', 'b' }, new char[3], {} };`

## 5.5 Mehr Kontrollstrukturen

Typische Form der Iteration über Felder:

- ▶ Initialisierung des Laufindex;
- ▶ `while`-Schleife mit Eintrittsbedingung für den Rumpf;
- ▶ Modifizierung des Laufindex am Ende des Rumpfs.

# Beispiel

```
1 int result = a[0];
2 int i = 1;      // Initialisierung
3 while (i < a.length) {
4     if (a[i] < result)
5         result = a[i];
6     i = i + 1;  // Modifizierung
7 }
8 write(result);
```

Bestimmung des Minimums

# Das For-Statement

```
1 int result = a[0];  
2 for (int i = 1; i < a.length; ++i)  
3     if (a[i] < result)  
4         result = a[i];  
5 write(result);
```

Bestimmung des Minimums

# Das For-Statement

```
for (init; cond; modify) stmt
```

entspricht:

```
{ init; while (cond) { stmt modify; } }
```

## Erläuterungen:

- ▶ `++i;` ist äquivalent zu `i = i + 1;`
- ▶ die `while`-Schleife steht innerhalb eines **Blocks** (`{...}`)

die Variable `i` ist außerhalb dieses Blocks nicht sichtbar/zugreifbar

## 5.6 Funktionen und Prozeduren

Oft möchte man:

- ▶ Teilprobleme **separat** lösen; und dann
- ▶ die Lösung **mehrfach** verwenden.

# Beispiel

Funktionsname

Typ des Rückgabewertes

Liste der formalen Parameter

```
public static int[] readArray(int number) {  
    // number = Anzahl zu lesender Elemente  
    int[] result = new int[number]; // Feld anlegen  
    for (int i = 0; i < number; ++i) {  
        result[i] = read();  
    }  
    return result;  
}
```

Einlesen eines Feldes

Rückgabe des Ergebnisses + Beenden der Funktion

Funktionsrumpf

## 5.6 Funktionen und Prozeduren

### Erläuterungen:

- ▶ Die erste Zeile ist der **Header** der Funktion.
- ▶ **public** und **static** kommen später
- ▶ **int[]** gibt den Typ des Rückgabe-Werts an.
- ▶ **readArray** ist der Name, mit dem die Funktion aufgerufen wird.
- ▶ Dann folgt (in runden Klammern und komma-separiert) die Liste der **formalen Parameter**, hier: **(int number)**.
- ▶ Der Rumpf der Funktion steht in geschweiften Klammern.
- ▶ **return expr;** beendet die Ausführung der Funktion und liefert den Wert von **expr** zurück.

## 5.6 Funktionen und Prozeduren

### Erläuterungen:

- ▶ Die Variablen, die innerhalb eines Blocks angelegt werden, d.h. innerhalb von '{' und '}', sind nur innerhalb dieses Blocks **sichtbar** d.h. benutzbar.
- ▶ Der Rumpf einer Funktion ist ein Block. Dort deklarierte Variablen nennt man **lokale Variablen**.
- ▶ Die formalen Parameter können auch als lokale Variablen aufgefasst werden.
- ▶ Bei dem Aufruf `readArray(7)` erhält der formale Parameter `number` den Wert `7` (**aktueller Parameter**).

# Beispiel

```
public static int min(int[] b) {  
    int result = b[0];  
    for (int i = 1; i < b.length; ++i) {  
        if (b[i] < result)  
            result = b[i];  
    }  
    return result;  
}
```

Bestimmung des Minimums

# Beispiel

```
public class Min extends MiniJava {
    public static int[] readArray(int number) { ... }
    public static int min(int[] b) { ... }
    // Jetzt kommt das Hauptprogramm
    public static void main(String[] args) {
        int n = read();
        int[] a = readArray(n);
        int result = min(a);
        write(result);
    } // end of main()
} // end of class Min
```

Programm zur Minimumsberechnung

# Beispiel

## Erläuterungen:

- ▶ Manche Funktionen, deren Ergebnistyp `void` ist, geben gar keine Werte zurück – im Beispiel: `write()` und `main()`. Diese Funktionen heißen **Prozeduren**.
- ▶ Das Hauptprogramm hat immer als Parameter ein Feld `args` von `String`-Elementen.
- ▶ In diesem Argument-Feld werden dem Programm Kommandozeilen-Argumente verfügbar gemacht.

```
public class Test extends MiniJava {  
    public static void main (String[] args) {  
        write(args[0]+args[1]);  
    }  
} // end of class Test
```

# Beispiel

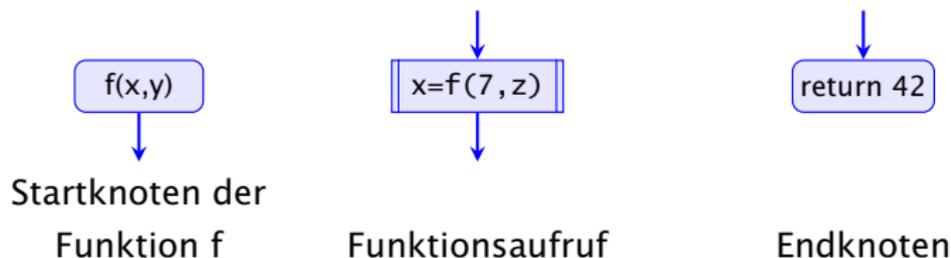
Der Aufruf

```
java Test "He1" "1o Wor1d!"
```

liefert: He1lo Wor1d!

## 5.6 Funktionen und Prozeduren

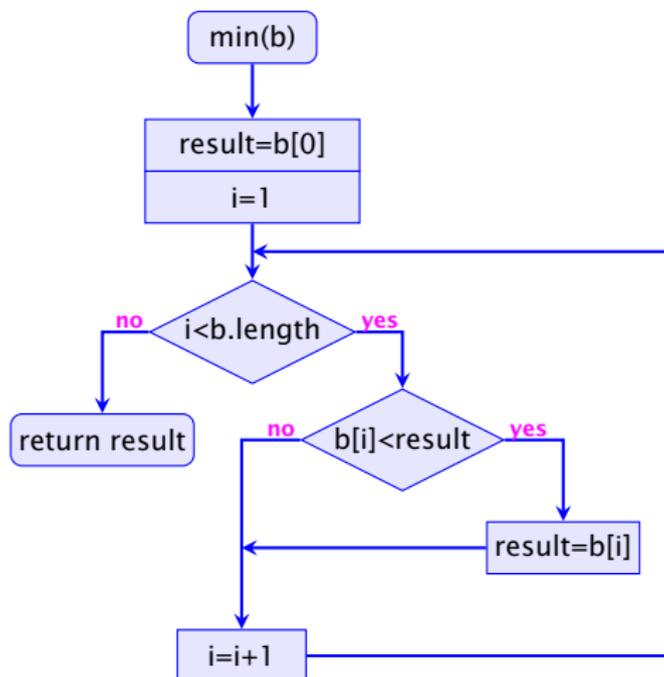
Um die Arbeitsweise von Funktionen zu veranschaulichen erweitern/modifizieren wir die Kontrollflussdiagramme



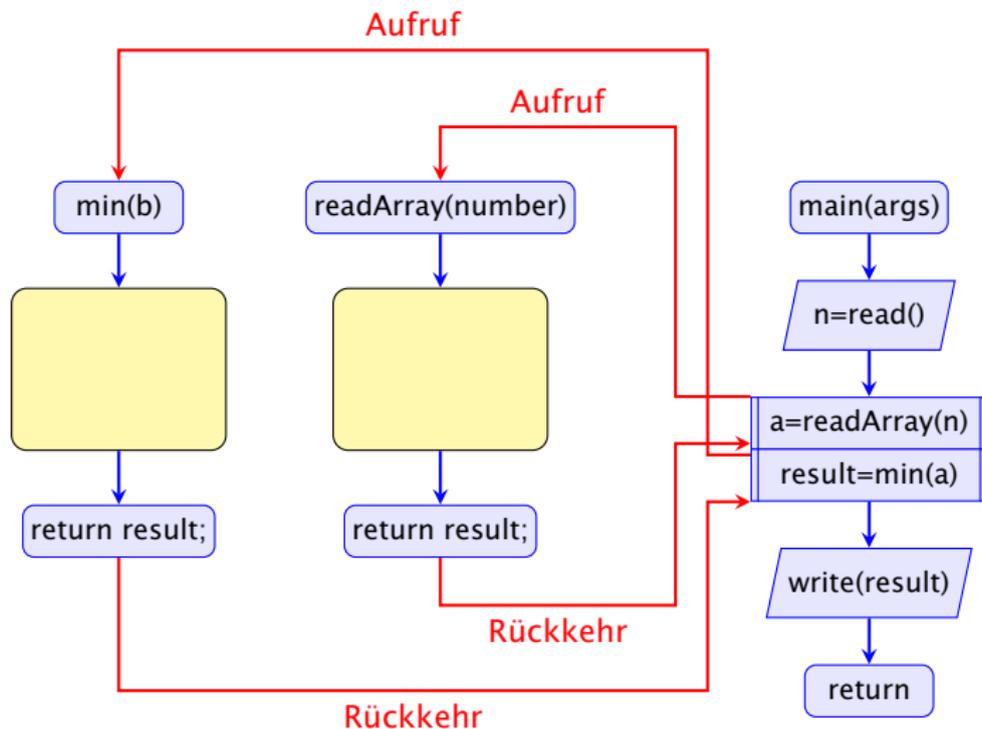
- ▶ Für jede Funktion wird ein eigenes Teildiagramm erstellt.
- ▶ Ein Aufrufknoten repräsentiert eine Teilberechnung der aufgerufenen Funktion.

## 5.6 Funktionen und Prozeduren

### Teildiagramm der Funktion `min()`:



## 5.6 Funktionen und Prozeduren

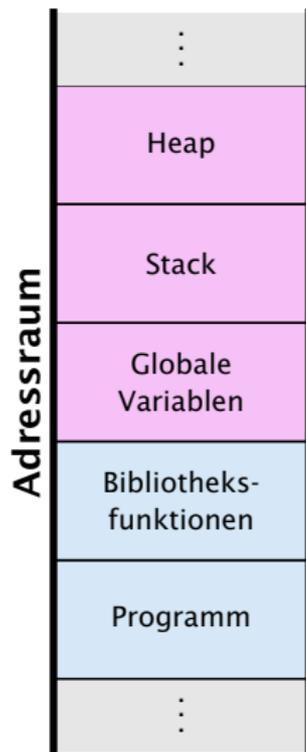


## 6 Speicherorganisation

Der Speicher des Programms ist in verschiedene Speicherbereiche untergliedert

- ▶ Speicherbereiche, die den eigentlichen Programmcode und den Code der Laufzeitbibliothek enthalten;
- ▶ einen Speicherbereich für **globale/statische Variablen**;
- ▶ einen Speicherbereich **Heap**, und
- ▶ einen Speicherbereich **Stack**.

Variablen werden üblicherweise auf dem Heap oder dem Stack gespeichert.



# Heap vs. Stack vs. statisch

## Heap

Auf dem Heap können zur Laufzeit zusammenhängende Speicherbereiche angefordert werden, und in beliebiger Reihenfolge wieder freigegeben werden.

## Stack

Der Stack ist ein Speicherbereich, auf dem neue Elemente oben gespeichert werden, und Freigaben in umgekehrter Reihenfolge (d.h. oben zuerst) erfolgen müssen (**LIFO** = **L**ast **I**n **F**irst **O**ut).

## Statische Variablen

Statische Variablen werden zu Beginn des Programms angelegt, und zum Ende des Programms wieder gelöscht.

In **Java** müssen Elemente auf dem Heap nicht explizit wieder freigegeben werden. Diese Freigabe übernimmt der **Garbage Collector**.

# Statische Variablen

**Statische Variablen** (auch Klassenvariablen) werden im Klassenrumpf **ausserhalb** einer Funktion mit dem zusätzlichen Schlüsselwort **static** definiert.

Jede Funktion der Klasse kann dann diese Variablen benutzen; deshalb werden sie manchmal auch globale Variablen genannt.

# Beispiel – Statische Variablen

```
1 public class GGT extends MiniJava {
2     static int x, y;
3     public static void readInput() {
4         x = read();
5         y = read();
6     }
7     public static void main (String[] args) {
8         readInput();
9         while (x != y) {
10            if (x < y)
11                y = y - x;
12            else
13                x = x - y;
14        }
15        write(x);
16    }
17 }
```

# Verwendung des Heaps

Speicherallokation mit dem Operator `new`:

```
int[][] arr;  
arr = new int[10][]; // array mit int-Verweisen
```

Immer wenn etwas mit `new` angelegt wird, landet es auf dem Heap.

Wenn keine Referenz mehr auf den angeforderten Speicher existiert **kann** der **Garbage Collector** den Speicher freigeben:

```
int[][] arr;  
arr = new int[10][]; // array mit int-Verweisen  
arr = null; // jetzt koennte GC freigeben
```

# Verwendung des Heaps

## Beispiel:

```
1 public static int[] readArray(int number) {
2     // number = Anzahl zu lesender Elemente
3     int[] result = new int[number];
4     for (int i = 0; i < number; ++i) {
5         result[i] = read();
6     }
7     return result;
8 }
9 public static void main(String[] args) {
10     readArray(6);
11 }
```

Da die von `readArray` zurückgegebene Referenz nicht benutzt wird, kann der GC freigeben.

# Verwendung des Heaps

## Beispiel:

```
1 public static void main(String[] args) {  
2     int[] b = readArray(6);  
3     int[] c = b;  
4     b = null;  
5 }
```

Da `c` immer noch eine Referenz auf das array enthält erfolgt keine Freigabe.

# Verwendung des Stacks

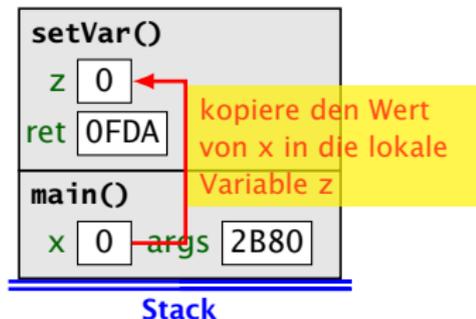
- ▶ Bei Aufruf einer Funktion (auch `main()`) werden lokale Variablen (d.h. auch Werte von aktuellen Parametern) und die Rücksprungadresse als **Frames** auf dem Stack gespeichert.
- ▶ Während der Programmausführung sind nur die Variablen im obersten Frame zugreifbar.
- ▶ Bei der Beendigung einer Funktion wird der zugehörige Stackframe gelöscht.

# Parameterübergabe – Call-by-Value

Die Variable, die wir bei dem Aufruf übergeben, verändert ihren Wert nicht.

```
public static void setVar(int z) {  
    z = 1;  
}
```

```
public static void main(String[] args) {  
    int x;  
    x = 0;  
    setVar(x);  
    write("x == " + x);  
}
```



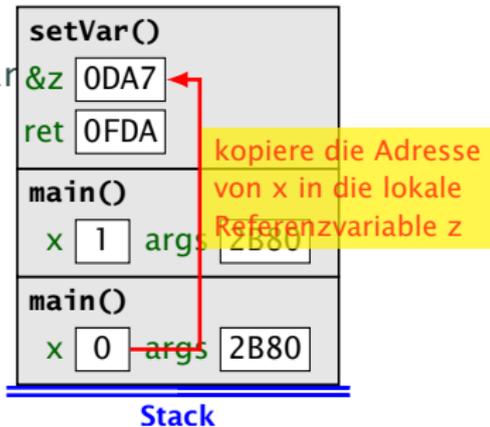
Das ist die einzige Form der Parameterübergabe, die **Java** unterstützt.

# Parameterübergabe – Call-by-Referenz

Animation ist nur in der Vorlesungsversion der Folien vorhanden.

```
public static void setVar(int &z) {  
    z = 1;  
}
```

```
public static void main(String[] args)  
{  
    int x;  
    x = 0;  
    setVar(x);  
    write("x == " + x);  
}
```

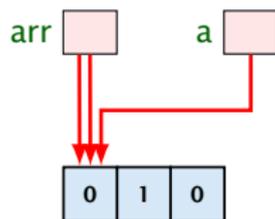


Diese Form der Parameterübergabe ist in **Java** nicht möglich, aber z.B. in **C++**.

## Parameterübergabe – Referenzvariablen

Auch Referenzvariablen werden per call-by-value übergeben. Man kann den **Inhalt** des zugehörigen Objekts/Arrays aber verändern.

```
public static void setVar(int[] a) {  
    a[1] = 1;  
}  
  
public static void main(String[] args) {  
    // initialize array elements to 0  
    int[] arr = new int[3];  
  
    setVar(arr);  
    write("arr[1] == " + arr[1]);  
}
```



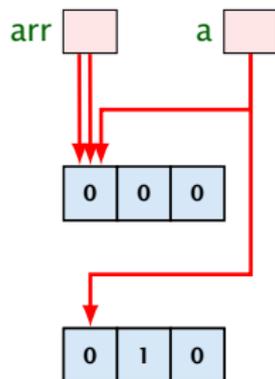
Ausgabe: **arr[1] == 1**

## Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {  
    a = new int[3];  
    a[1] = 1;  
}
```

```
public static void main(String[] args) {  
    // initialize array elements to 0  
    int[] arr = new int[3];  
    setVar(arr);  
    write("arr[1] == " + arr[1]);  
}
```



Ausgabe: `arr[1] == 0`

# Rekursive Funktionen

Animation ist nur in der Vorlesungsversion der Folien vorhanden.

**Rekursive Funktionen** sind Funktionen, die sich selbst aufrufen (eventuell über Umwege).

**Beispiel:** Fakultätsberechnung

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

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

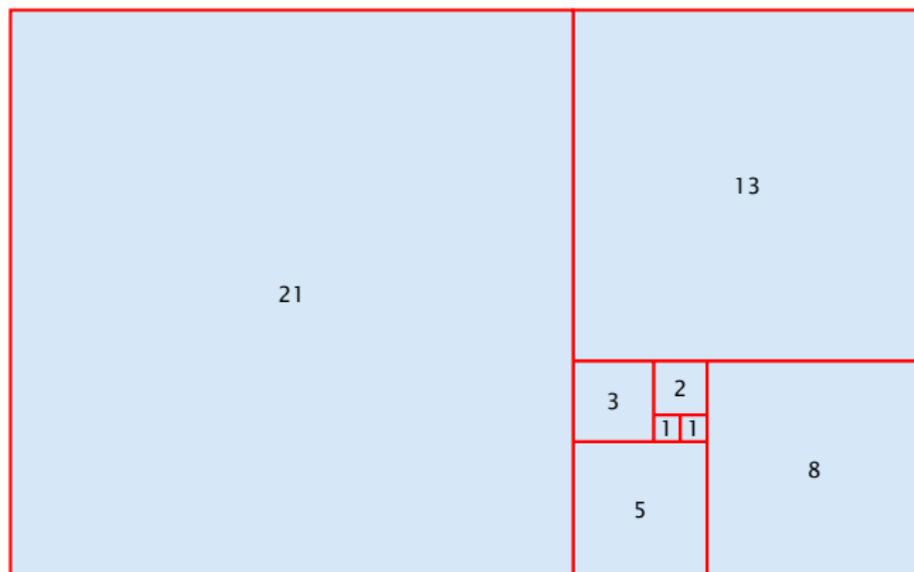


# Vollständiger Code

```
1 public class Fakulttaet {
2     public static long fak(int n) {
3         if (n > 0)
4             return n * fak(n-1);
5         else
6             return 1;
7     }
8     public static void main(String args[]) {
9         System.out.println(fak(20));
10    }
11 }
```

# Fibonaccizahlen

$$F_n = \begin{cases} n & 0 \leq n \leq 1 \\ F_{n-1} + F_{n-2} & n \geq 2 \end{cases}$$



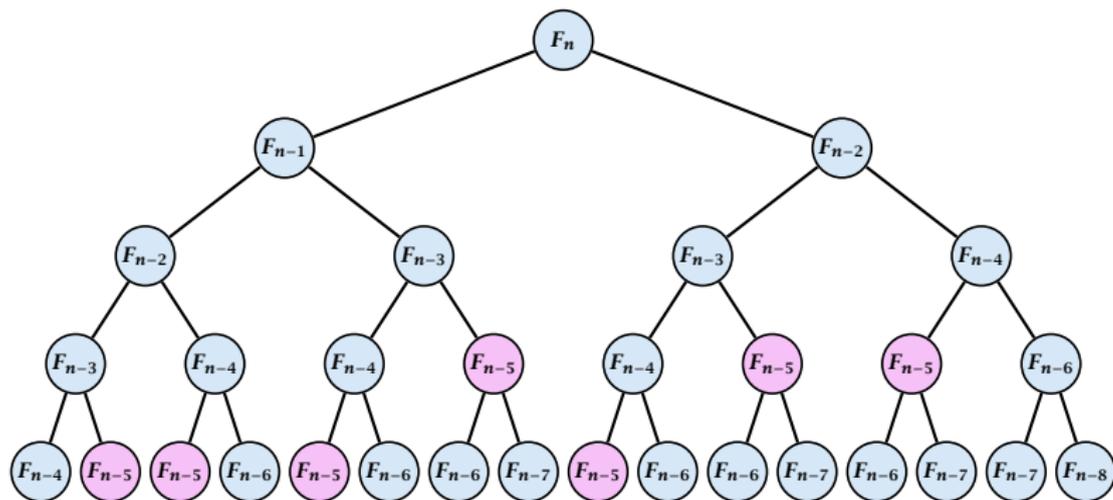
# Vollständiger Code

```
1 public class Fibonacci {
2     public static long fib(int n) {
3         if (n > 1)
4             return fib(n-1)+fib(n-2);
5         else
6             return n;
7     }
8
9     public static void main(String args[]) {
10        System.out.println(fib(50));
11    }
12 }
```

# Fibonaccizahlen

Programmmlauf benötigt mehr als 1 min.

Warum ist das so langsam?



Wir erzeugen viele rekursive Aufrufe für die gleichen Teilprobleme!

## Lösung

- ▶ Speichere die Lösung für ein Teilproblem in einer **globalen Variable**.
- ▶ Wenn das Teilproblem das nächste mal gelöst werden soll braucht man nur nachzuschauen...

# Vollständiger Code

```
1 public class FibonacciImproved {
2     // F93 does not fit into a long
3     static long[] lookup = new long[93];
4
5     public static long fib(int n) {
6         if (lookup[n] > 0) return lookup[n];
7
8         if (n > 1) {
9             lookup[n] = fib(n-1)+fib(n-2);
10            return lookup[n];
11        } else
12            return n;
13    }
14    public static void main(String args[]) {
15        System.out.println(fib(50));
16    }
17 }
```

Hier nutzen wir die Tatsache, dass der `new`-Operator `long`-Variablen mit dem Wert Null initialisiert.

**Achtung:** lokale Variablen werden nicht initialisiert.

## 7 Anwendung: Sortieren

**Gegeben:** eine Folge von ganzen Zahlen.

**Gesucht:** die zugehörige aufsteigend sortierte Folge.

**Idee:**

- ▶ speichere die Folge in einem Feld ab;
- ▶ lege ein weiteres Feld an;
- ▶ füge der Reihe nach jedes Element des ersten Felds an der richtigen Stelle in das zweite Feld ein!

⇒ Sortieren durch Einfügen (↑InsertionSort)

## 7 Anwendung: Sortieren

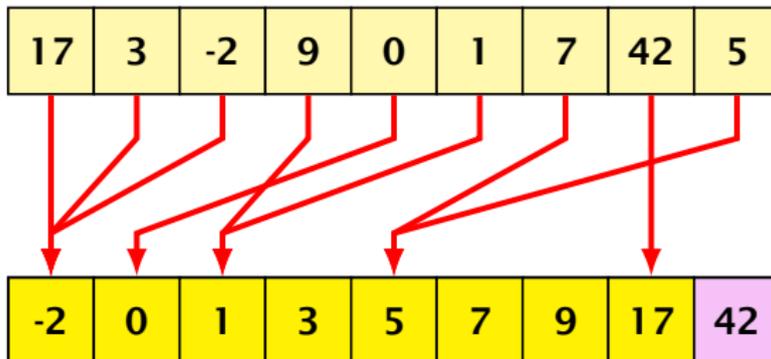
```
1 public static int[] sort(int[] a) {
2     int n = a.length;
3     int[] b = new int[n];
4     for (int i = 0; i < n; ++i)
5         insert(b, a[i], i);
6         // b    = Feld, in das eingefuegt wird
7         // a[i] = einzufuegendes Element
8         // i    = Anzahl von Elementen in b
9     return b;
10 } // end of sort ()
```

### Sortieren durch Einfügen

Teilproblem: wie fügt man ein?

# Beispiel

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



## 7 Anwendung: Sortieren

```
1 public static void insert(int[] b, int x, int i) {
2     // finde Einfuegestelle j fuer x in b
3     int j = locate(b,x,i);
4     // verschiebe in b Elemente b[j],...,b[i-1]
5     // nach rechts
6     shift(b,j,i);
7     b[j] = x;
8 }
```

### Einfügen

- ▶ Wie findet man Einfügestelle?
- ▶ Wie verschiebt man nach rechts?

## 7 Anwendung: Sortieren

```
public static int locate(int[] b, int x, int i) {
    int j = 0;
    while (j < i && x > b[j]) ++j;
    return j;
}

public static void shift(int[] b, int j, int i) {
    for (int k = i-1; k >= j; --k)
        b[k+1] = b[k];
}
```

- ▶ Warum läuft Iteration in `shift()` von `i-1` **abwärts** nach `j`?

## 7 Anwendung: Sortieren

### Erläuterungen

- ▶ Das Feld `b` ist (ursprünglich) lokale Variable von `sort()`.
- ▶ Lokale Variablen sind nur im eigenen Funktionsrumpf sichtbar, nicht in den aufgerufenen Funktionen.
- ▶ Damit die aufgerufenen Hilfsfunktionen auf `b` zugreifen können, muss `b` explizit als Parameter übergeben werden!

### Achtung:

Das Feld wird nicht kopiert. Das Argument ist der Wert der Variablen `b`, also nur eine Referenz!

- ▶ Deshalb benötigen weder `insert()`, noch `shift()` einen separaten Rückgabewert. . .
- ▶ Weil das Problem so klein ist, würde eine erfahrene Programmiererin hier keine Unterprogramme benutzen...

## 7 Anwendung: Sortieren

```
1 public static int[] sort(int[] a) {
2     int[] b = new int[a.length];
3     for (int i = 0; i < a.length; ++i) {
4         // begin of insert
5         int j = 0;
6         while (j < i && a[i] > b[j]) ++j;
7         // end of locate
8         for (int k = i-1; k >= j; --k)
9             b[k+1] = b[k];
10        // end of shift
11        b[j] = a[i];
12        // end of insert
13    }
14    return b;
15 } // end of sort
```

# 7 Anwendung: Sortieren

## Diskussion

- ▶ Die Anzahl der ausgeführten Operationen wächst quadratisch in der Größe des Felds  $a$ .
- ▶ Glücklicherweise gibt es Sortierverfahren, die eine bessere Laufzeit haben (↑[Algorithmen und Datenstrukturen](#)).

## 8 Anwendung: Suchen

**Gegeben:** Folge  $a$  ganzer Zahlen; Element  $x$

**Gesucht:** Wo kommt  $x$  in  $a$  vor?

**Naives Vorgehen:**

- ▶ Vergleiche  $x$  der Reihe nach mit  $a[0]$ ,  $a[1]$ , usw.
- ▶ Finden wir  $i$  mit  $a[i] == x$ , geben wir  $i$  aus.
- ▶ Andernfalls geben wir  $-1$  aus: „Element nicht gefunden“!

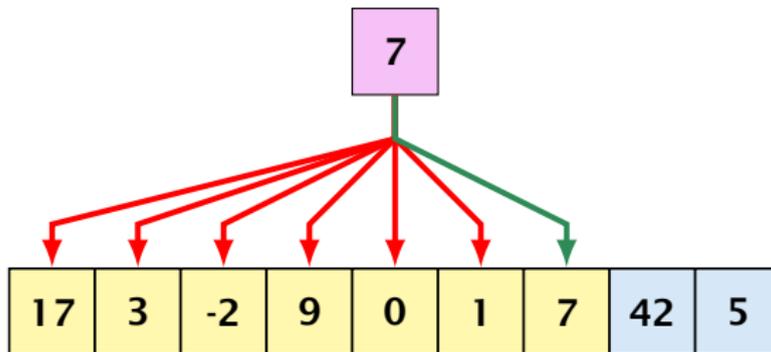
# Naives Suchen

```
1 public static int find(int[] a, int x) {  
2     int i = 0;  
3     while (i < a.length && a[i] != x)  
4         ++i;  
5     if (i == a.length)  
6         return -1;  
7     else  
8         return i;  
9 }
```

## Naives Suchen

# Beispiel

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



yes

# Naives Suchen

- ▶ Im Beispiel benötigen wir 7 Vergleiche
- ▶ Im schlimmsten Fall (**worst case**) benötigen wir bei einem Feld der Länge  $n$  sogar  $n$  Vergleiche.
- ▶ Kommt  $x$  tatsächlich im Feld vor, benötigen wir selbst im Durchschnitt  $(n + 1)/2$  Vergleiche.

**...geht das nicht besser?**

# Binäre Suche

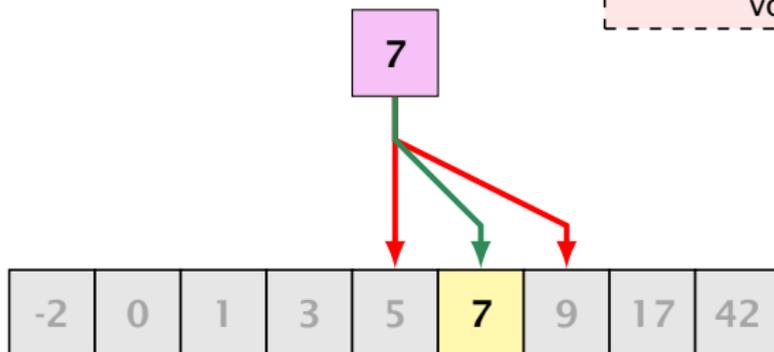
## Idee:

- ▶ Sortiere das Feld.
- ▶ Vergleiche  $x$  mit dem Wert, der in der Mitte steht.
- ▶ Liegt Gleichheit vor, sind wir fertig.
- ▶ Ist  $x$  kleiner, brauchen wir nur noch links weitersuchen.
- ▶ Ist  $x$  größer, brauchen wir nur noch rechts weiter suchen.

⇒ binäre Suche

# Beispiel

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



- ▶ wir benötigen nur **drei** Vergleiche
- ▶ hat das Feld  $2^n - 1$  Elemente, benötigen wir maximal  $n$  Vergleiche

# Implementierung

## Idee:

Führe Hilfsfunktion

```
public static int find0(int[] a,int x,int n1,int n2)
```

ein, die im Interval  $[n1, n2]$  sucht.

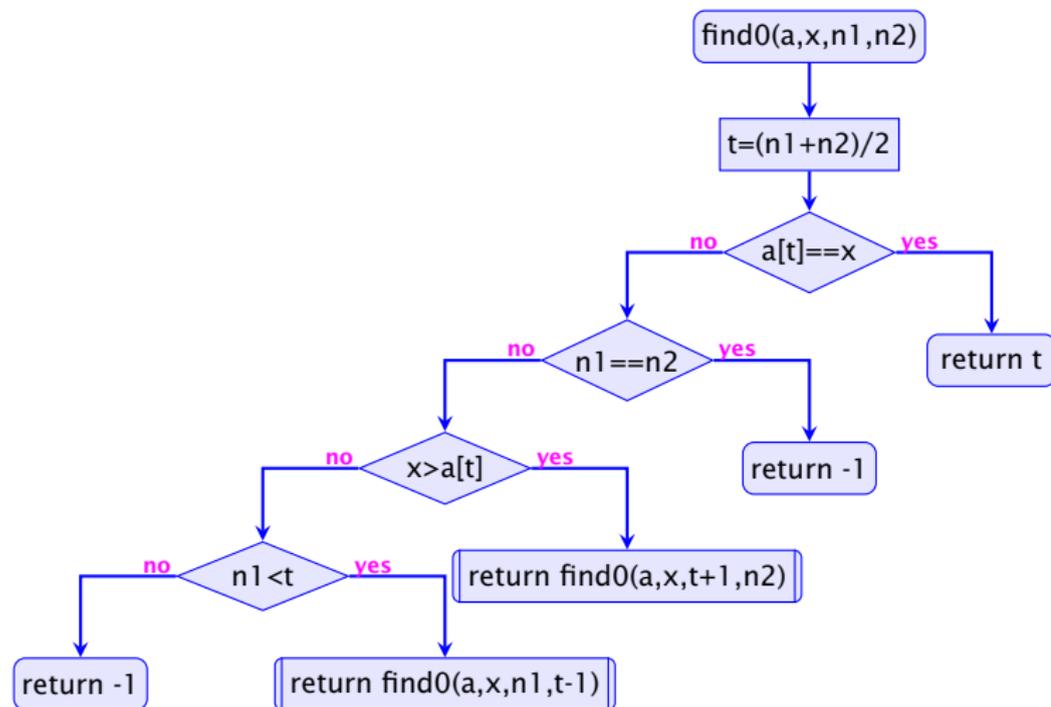
Damit:

```
public static int find(int[] a, int x) {  
    return find0(a,x,0,a.length-1);  
}
```

# Implementierung

```
1 public static int find0(int[] a, int x, int n1, int n2) {
2     int t = (n1 + n2) / 2;
3     if (a[t] == x)
4         return t;
5     else if (n1 == n2)
6         return -1;
7     else if (x > a[t])
8         return find0(a, x, t+1, n2);
9     else if (n1 < t)
10        return find0(a, x, n1, t-1);
11    else return -1;
12 }
```

# Kontrollflussdiagramm für find0

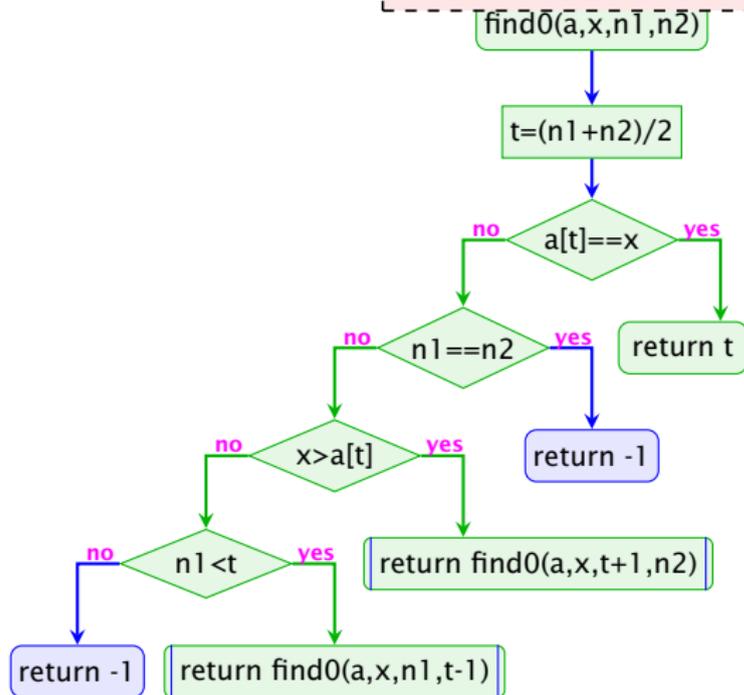


## Erläuterungen:

- ▶ zwei der `return`-Statements enthalten einen Funktionsaufruf – deshalb die Markierungen an den entsprechenden Knoten.
- ▶ (Wir hätten stattdessen auch zwei Knoten und eine Hilfsvariable `result` einführen können)
- ▶ `find0()` ruft sich selbst auf, ist also **rekursiv**.

# Ausführung

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



Stack

# Terminierung

Um zu **beweisen**, dass `find0()` terminiert, beobachten wir:

1. Wird `find0()` für ein einelementiges Intervall  $[n, n]$  aufgerufen, dann terminiert der Funktionsaufruf direkt.
2. wird `find0()` für ein Intervall  $[n1, n2]$  aufgerufen mit mehr als einem Element, dann terminiert der Aufruf entweder direkt (weil `x` gefunden wurde), oder `find0()` wird mit einem Intervall aufgerufen, das **echt** in  $[n1, n2]$  enthalten ist, genauer: sogar maximal die Hälfte der Elemente von  $[n1, n2]$  enthält.

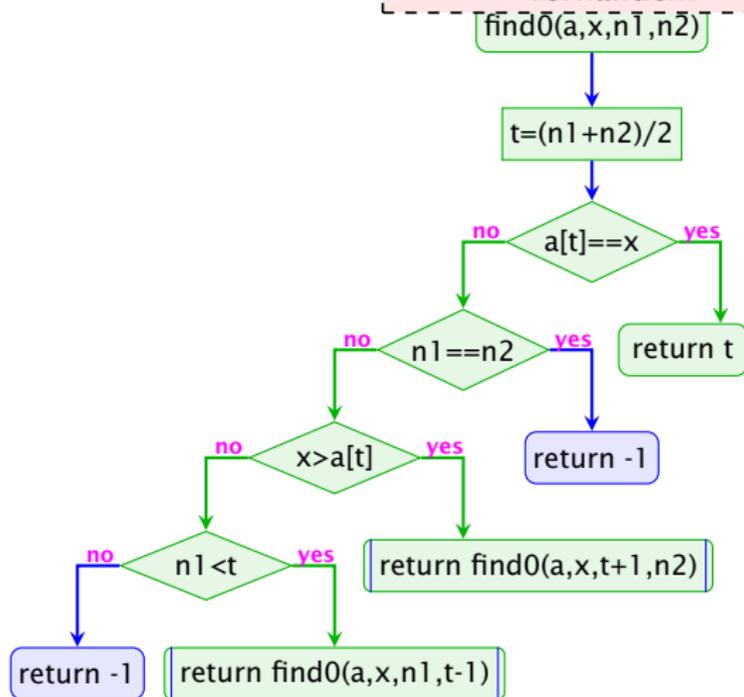
Ähnliche Beweistechnik wird auch für andere rekursive Funktionen verwendet.

# Beobachtung

- ▶ Das Ergebnis eines Aufrufs von `find0()` liefert **direkt** das Ergebnis auch für die aufrufende Funktion!
- ▶ Solche Rekursion heißt **End-** oder **Tail-Rekursion**.
- ▶ End-Rekursion kann auch ohne Aufrufkeller implementiert werden. . .
- ▶ **Idee:** lege den neuen Aufruf von `find0()` nicht oben auf den Stapel drauf, sondern **ersetze** den bereits dort liegenden Aufruf!

# Verbesserte Ausführung

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



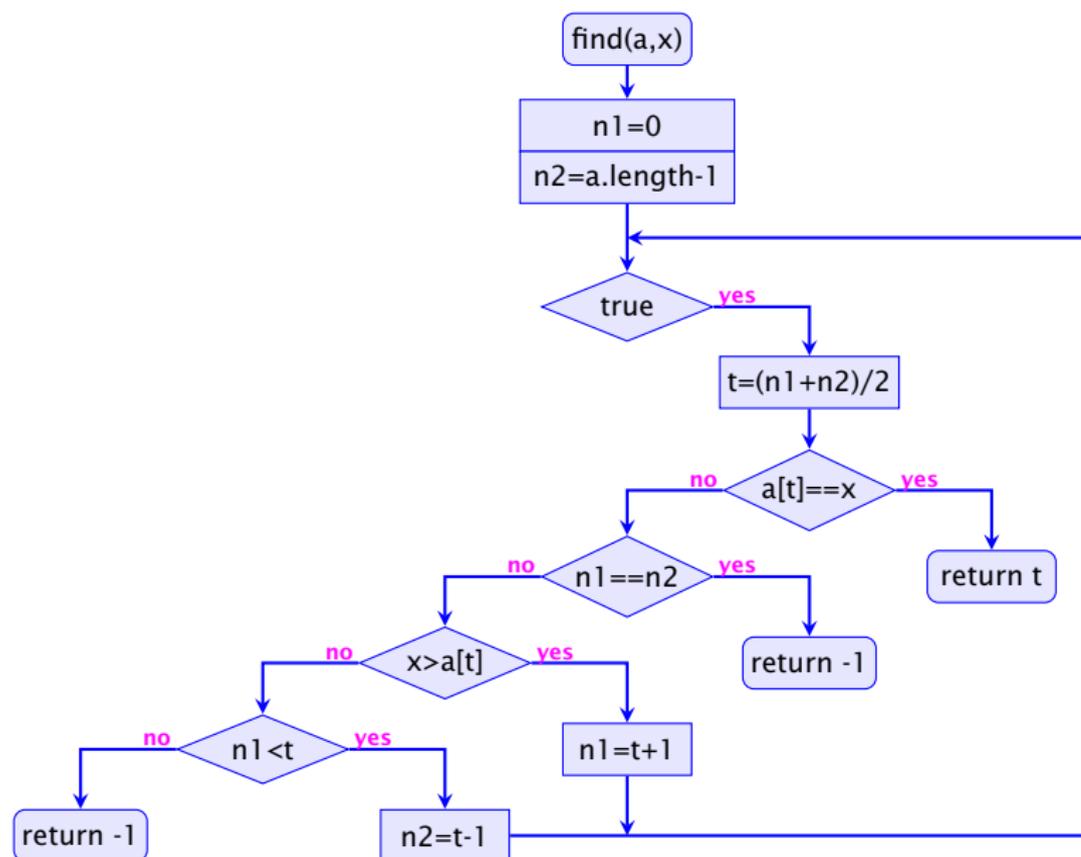
Stack

# Endrekursion

Endrekursion kann durch **Iteration** ersetzt werden...

```
1 public static int find(int[] a, int x) {
2     int n1 = 0;
3     int n2 = a.length-1;
4     while (true) {
5         int t = (n2 + n1) / 2;
6         if (x == a[t]) return t;
7         else if (n1 == n2) return -1;
8         else if (x > a[t]) n1 = t+1;
9         else if (n1 < t) n2 = t-1;
10        else return -1;
11    } // end of while
12 } // end of find
```

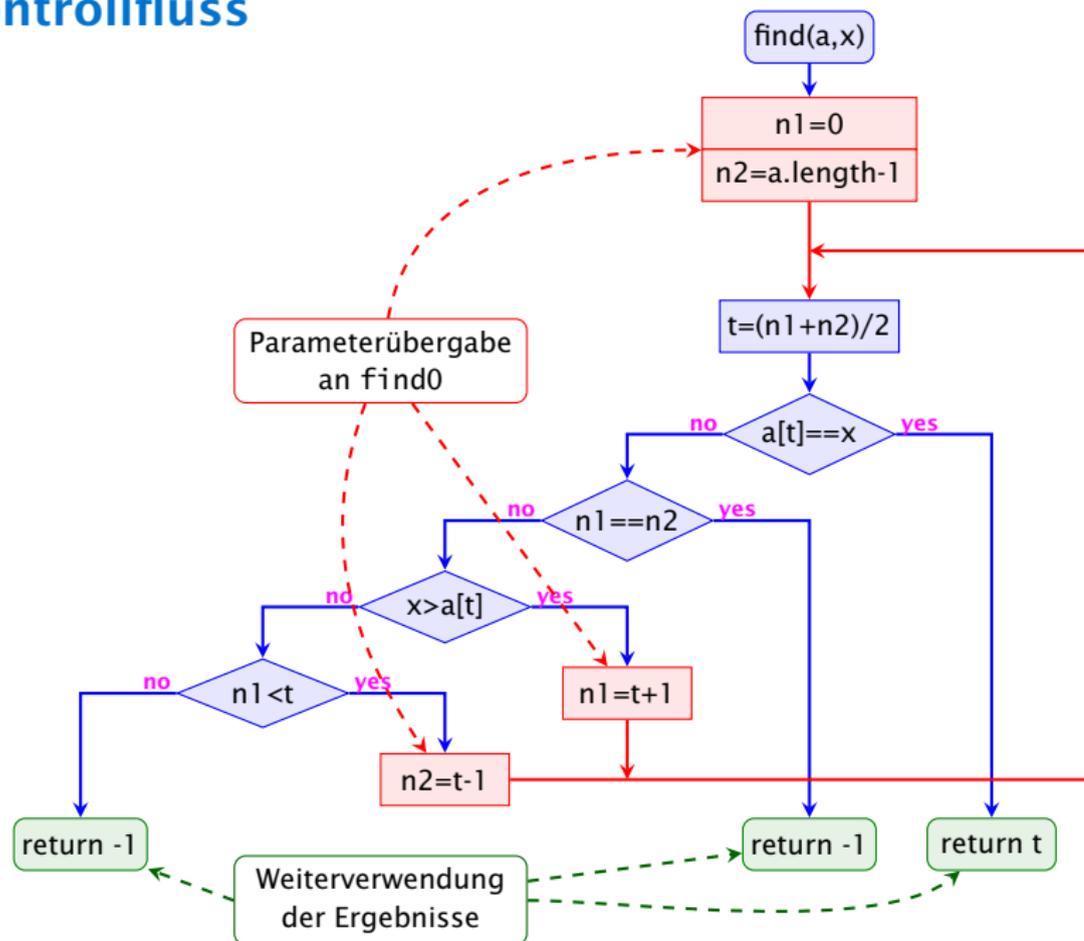
# Kontrollfluss



# Verlassen von Schleifen

- ▶ Die Schleife wird hier alleine durch die **return**-Anweisungen verlassen.
- ▶ Offenbar machen Schleifen mit **mehreren** Ausgängen Sinn.
- ▶ Um eine Schleife zu verlassen, ohne gleich ans Ende der Funktion zu springen, kann man das **break**-Statement benutzen.
- ▶ Der Aufruf der endrekursiven Funktion wird ersetzt durch:
  1. Code zur Parameter-Übergabe;
  2. einen **Sprung** an den Anfang des Rumpfs.

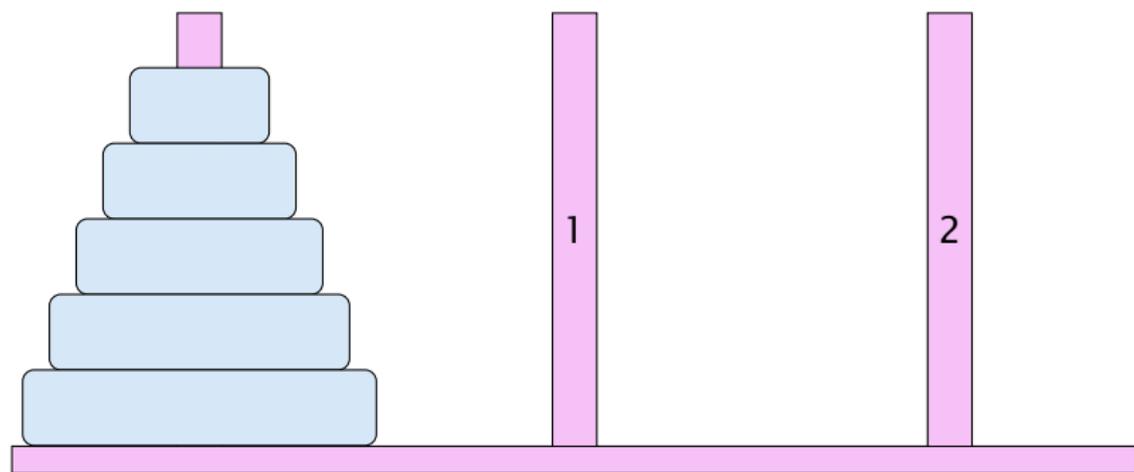
# Kontrollfluss



## Bemerkung

- ▶ Jede Rekursion läßt sich beseitigen, indem man den Aufruf-Keller **explizit** verwaltet.
- ▶ Nur im Falle von Endrekursion kann man auf den Keller verzichten.
- ▶ Rekursion ist trotzdem nützlich, weil rekursive Programme oft **leichter zu verstehen** sind als äquivalente Programme ohne Rekursion. . .

## 9 Türme von Hanoi



- ▶ Bewege Stapel von links nach rechts.
- ▶ In jedem Zug darf genau ein Ring bewegt werden.
- ▶ Es darf nie ein größerer auf einen kleineren Ring gelegt werden.

# 9 Türme von Hanoi

## Idee

- ▶ Für Turm der Höhe  $h = 0$  ist das Problem trivial.
- ▶ Falls  $h > 0$  zerlegen wir das Problem in drei Teilprobleme:
  1. Versetze oberen  $h - 1$  Ringe auf freien Platz
  2. Bewege die unterste Scheibe zum Ziel
  3. Versetze die zwischengelagerten Ringe zum Ziel
- ▶ Versetzen eines Turms der Höhe  $h > 0$  erfordert also zweimaliges Versetzen eines Turms der Höhe  $h - 1$ .

**Es gibt keine andere Möglichkeit!!!**

# Implementierung

```
1 public static void move(int h, byte a, byte b) {
2     if (h > 0) {
3         byte c = free(a,b);
4         move(h-1,a,c);
5         write("move "+a+" to "+b+"\n");
6         move(h-1,c,b);
7     }
8 }
```

... bleibt die Ermittlung des freien Rings

# Beobachtung

Offenbar hängt das Ergebnis nur von der Summe der beiden Argumente ab...

	0	1	2
0		2	1
1	2		0
2	1	0	

$\text{free}(x,y)$

	0	1	2
0		1	2
1	1		3
2	2	3	

$\text{sum}(x,y)$

# Implementierung

Um solche Tabellen **leicht** implementieren zu können stellt **Java** das **switch**-statement zur Verfügung:

```
1 public static byte free(byte a, byte b) {  
2     switch (a + b) {  
3         case 1: return 2;  
4         case 2: return 1;  
5         case 3: return 0;  
6         default: return -1;  
7     }  
8 }
```

# Allgemeines Switch-Statement

```
switch (expr) {  
    case const0: (ss0)? (break;)?  
    case const1: (ss1)? (break;)?  
        ...  
    case constk-1: (ssk-1)? (break;)?  
    (default: ssk)?  
}
```

- ▶ **expr** sollte eine ganze Zahl/**char** oder ein **String** sein.
- ▶ Die **const<sub>i</sub>** sind Konstanten des gleichen Typs.
- ▶ Die **ss<sub>i</sub>** sind alternative Statement-Folgen.
- ▶ **default** ist für den Fall, dass keine der Konstanten zutrifft
- ▶ Fehlt ein **break**-Statement, wird mit den Statement-Folgen der nächsten Alternative fortgefahren!

## Beispiel

Dies dient nur als Beispiel für die `switch`-Anweisung. Im vorliegenden Fall wäre es übersichtlicher für jeden Monat einen eigenen `case` einzuführen (d.h., kein `default`), und den „fall-through“ in den jeweils nächsten `case` zu vermeiden.

```
1 int numOfDays;  
2 boolean schaltjahr = true;  
3 switch (monat) {  
4     case "April":  
5     case "Juni":  
6     case "September":  
7     case "November":  
8         numOfDays = 30;  
9         break;  
10    case "Februar":  
11        if (schaltjahr)  
12            numOfDays = 29;  
13        else  
14            numOfDays = 28;  
15        break;  
16    default:  
17        numOfDays = 31;  
18 }
```

`monat` darf nicht `null` sein;  
man kann nicht mithilfe eines `switch`-statements gegen `null` testen.

# Der Bedingungsoperator

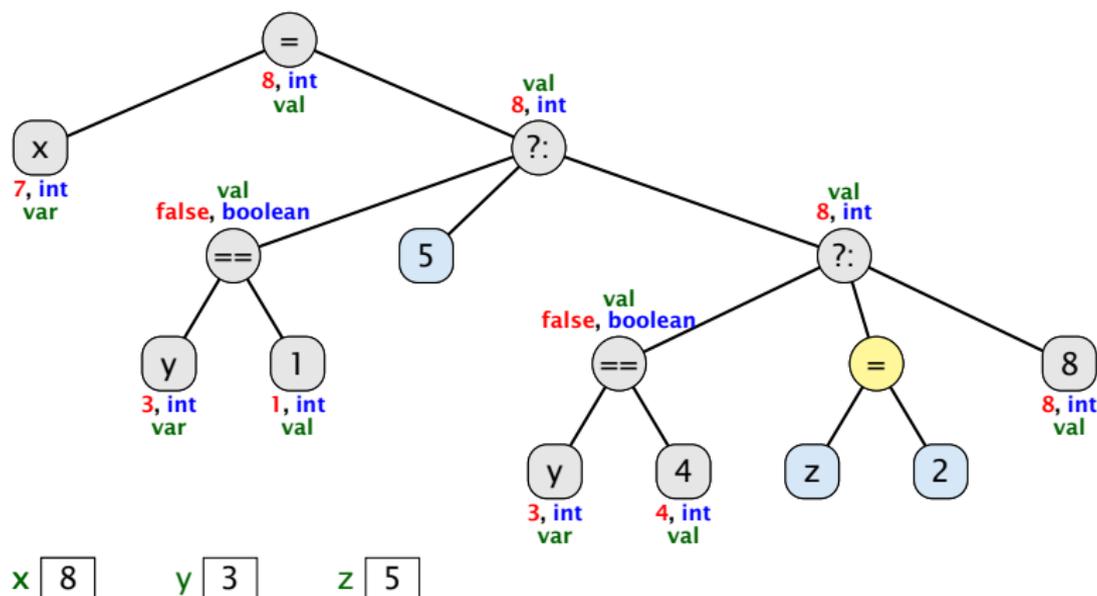
Eine Alternative zu einem `switch` ist der **Bedingungsoperator**:

`condition ? expr1 : expr2`

Der Bedingungsoperator

<i>symbol</i>	<i>name</i>	<i>types</i>	<i>L/R</i>	<i>level</i>
<code>? :</code>	Bedingungsoperator	boolean, 2*Typ	rechts	14

Beispiel:  $x = y == 1 ? 5 : y == 4 ? z = 2 : 8$



## Beispiel

`String` ist ein Referenzdatentyp. Ein Vergleich `monat == "Januar"` vergleicht nur die Referenzen, der `Strings` `monat` und `"Januar"`. Die sind im allgemeinen unterschiedlich, auch wenn `monat` und `"Januar"` den gleichen `Inhalt` haben.

```
numOfDays =
    "Januar".equals(monat) ? 31 :
    "Februar".equals(monat) ? (schaltjahr ? 29 : 28) :
    "Maerz".equals(monat) ? 31 :
    "April".equals(monat) ? 30 :
    "Mai".equals(monat) ? 31 :
    "Juni".equals(monat) ? 30 :
    "Juli".equals(monat) ? 31 :
    "August".equals(monat) ? 31 :
    "September".equals(monat) ? 30 :
    "Oktober".equals(monat) ? 31 :
    "November".equals(monat) ? 30 :
    "Dezember".equals(monat) ? 31 :
    -1 ;
```

# Implementierung

Für unseren Fall geht das viel einfacher:

```
1 public static byte free(byte a, byte b) {  
2     return (byte) (3-(a+b));  
3 }
```

## 9 Türme von Hanoi

### Bemerkungen:

- ▶ `move()` ist rekursiv, aber nicht end-rekursiv.
- ▶ Sei  $N(h)$  die Anzahl der ausgegebenen Moves für einen Turm der Höhe  $h \geq 0$ . Dann ist

$$N(h) = \begin{cases} 0 & \text{für } h = 0 \\ 1 + 2N(h-1) & \text{andernfalls} \end{cases}$$

- ▶ Folglich ist  $N(h) = 2^h - 1$ .
- ▶ Bei genauerer Analyse des Problems lässt sich auch ein nicht ganz so einfacher nicht-rekursiver Algorithmus finden.

**Hinweis:** Offenbar rückt die kleinste Scheibe in jedem zweiten Schritt eine Position weiter. . .

# Objektorientierte Programmierung

Bis jetzt kennen wir (fast) nur primitive Datentypen.

Diese entsprechen weitestgehend der Hardware des Rechners (z.B. besitzt ein Rechner Hardware um zwei `floats` zu addieren).

Wir möchten Dinge der realen Welt modellieren, dafür benötigen wir komplexere Datentypen.

**Lösung:** selbstdefinierte Datentypen

# Objektorientierte Programmierung

Angenommen wir möchten eine Adressverwaltung schreiben.  
Dazu müßten wir zunächst eine Adresse **modellieren**:

Harald Räcke Boltzmannstraße 3 85748 Garching
---

Adresse	
+ Name	: String
+ Strasse	: String
+ Hausnummer	: int
+ Postleitzahl	: int
+ Stadt	: String

Zumindest für diesen Fall ist die Modellierung sehr einfach.

Datentyp ist hier nur eine **Komposition** (Zusammensetzung) von anderen einfacheren Grundtypen.

Wir visualisieren den Datentyp hier über ein UML-Diagramm. Dies ist eine grafische Modellierungssprache um Software zu spezifizieren. UML ist nicht speziell für **Java** entwickelt worden; deshalb unterscheidet sich die Syntax leicht.

# Objektorientierte Programmierung

Wir werden diesem Grundprinzip, dass man Objekte **nur** über Methoden ändern sollte, nicht immer folgen...

Wie benutzt man den Datentyp?

Geht aus der Ansammlung der Grundtypen nicht hervor. Wenn der Datentyp sehr komplex ist (Atomreaktor), kann man leicht Fehler machen, und einen ungültigen Zustand erzeugen.

## **Grundidee:**

Ändere Variablen des Datentyps nur über Funktionen/Methoden.

Falls diese korrekt implementiert sind, kann man keinen ungültigen Zustand erzeugen.

Daten und Methoden  
gehören zusammen  
(abstrakter Datentyp)

# Objektorientierte Programmierung

Ein (abstrakter) Datentyp besteht aus Daten und einer Menge von Methoden (Schnittstelle) um diese Daten zu manipulieren.

## Datenkapselung / Information Hiding

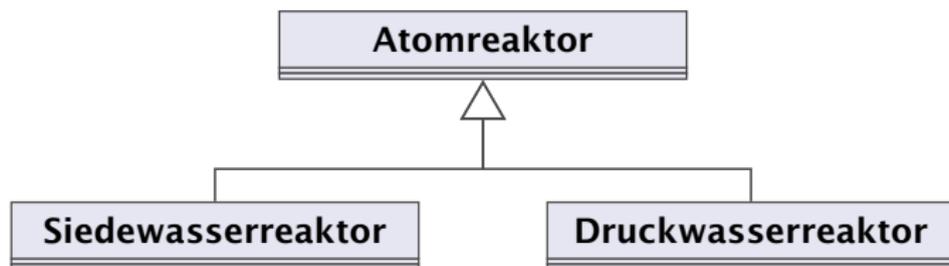
Die Implementierung des Datentyps wird vor dem Benutzer versteckt.

- ▶ minimiert Fehler durch unsachgemäßen Zugriff
- ▶ **Entkopplung** von Teilproblemen
  - ▶ gut für Implementierung, aber auch
  - ▶ Fehlersuche und Wartung
- ▶ erlaubt es die Implementierung später anzupassen (↑**rapid prototyping**)
- ▶ erzwingt in der Designphase über das **was** und nicht über das **wie** nachzudenken....

# Objektorientierte Programmierung

## Generalisierung + Vererbung

Identifiziere Ähnlichkeiten zwischen Datentypen und lagere gemeinsame Teile in einen anderen Datentyp aus.



- ▶ vermeidet Copy&Paste...
- ▶ verringert den Wartungsaufwand...

# Objektorientierte Programmierung

**Klasse** = Implementierung eines abstrakten Datentyps

**Objekt** = Instanz/Variable einer Klasse

## Beispiel: Rationale Zahlen

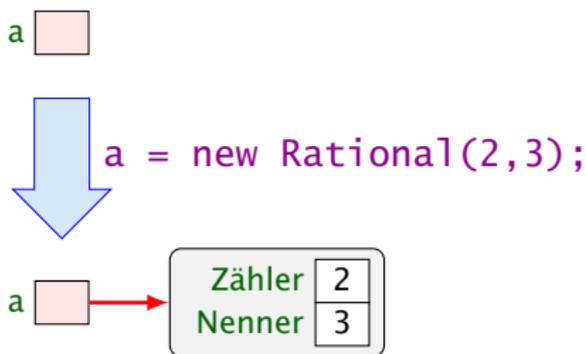
- ▶ Eine rationale Zahl  $q \in \mathbb{Q}$  hat die Form  $\frac{x}{y}$ , wobei  $x, y \in \mathbb{Z}$ .
- ▶  $x$  und  $y$  heißen Zähler und Nenner von  $q$ .
- ▶ Ein Objekt vom Typ `Rational` sollte deshalb als Komponenten `int`-Variablen `zaehler` und `nenner` erhalten:



- ▶ Die Daten eines Objektes heißen **Instanz-Variablen** oder **Attribute**.

## Beispiel: Rationale Zahlen

- ▶ `Rational name`; deklariert eine Variable für Objekte der Klasse `Rational`.
- ▶ Das Kommando `new Rational(...)` legt das Objekt an, ruft einen **Konstruktor** für dieses Objekt auf, und liefert einen **Verweis** auf das neue Objekt zurück.

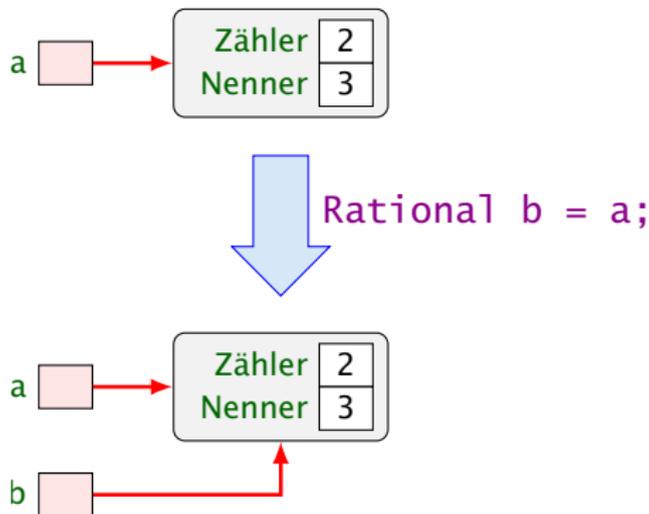


- ▶ Der Konstruktor ist eine Prozedur, die die Attribute des neuen Objektes initialisieren kann.

## Erinnerung: Referenzen

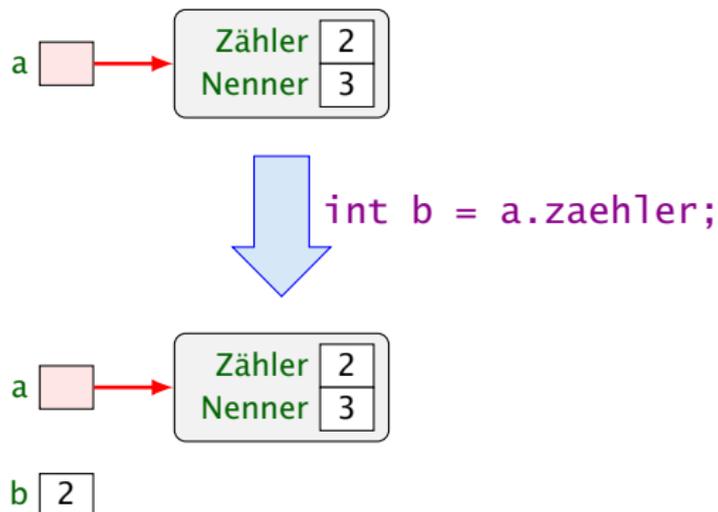
Der Wert der `Rational`-Variablen ist eine **Referenz/Verweis** auf einen Speicherbereich.

`Rational b = a;` kopiert den Verweis aus `a` in die Variable `b`:



# Beispiel: Rationale Zahlen

`a.zaehler` liefert den Wert des Attributs `zaehler` des Objektes auf das `a` verweist:

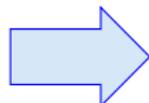


## Beispiel: Rationale Zahlen

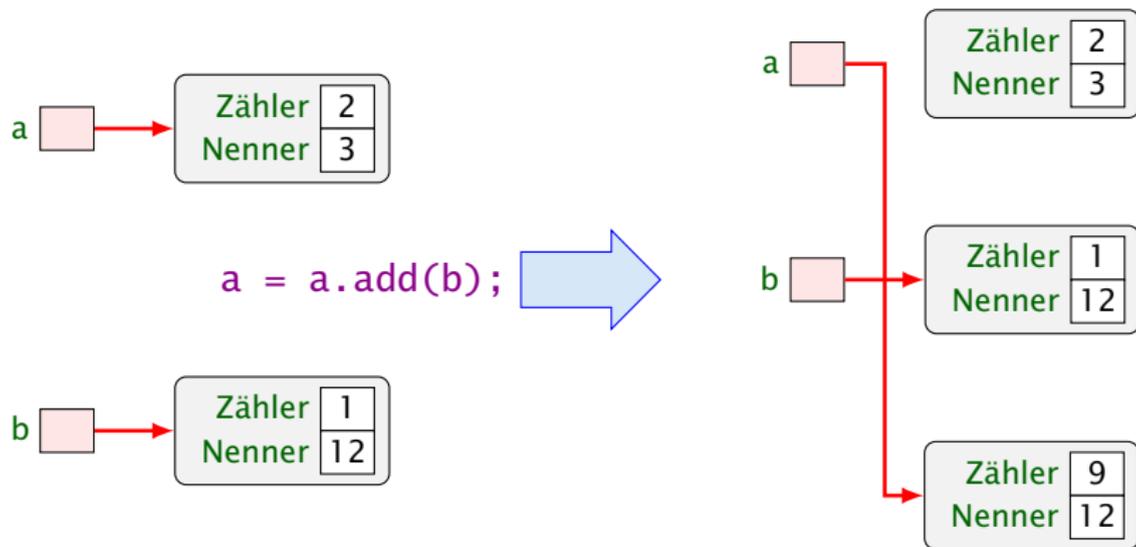
`a.add(b)` ruft die Operation `add` für `a` mit dem zusätzlichen aktuellen Parameter `b` auf:



`Rational c = a.add(b);`



# Beispiel: Rationale Zahlen



Die Operationen auf Objekten einer Klasse heißen auch **Methoden**, genauer: **Objekt-Methoden**.

# Zusammenfassung

Eine Klassendeklaration besteht folglich aus:

- ▶ **Attributen** für die verschiedenen Wertkombinationen der Objekte;
- ▶ **Konstruktoren** zur Initialisierung der Objekte;
- ▶ **Methoden**, d.h. Operationen auf Objekten.

Diese Elemente heißen auch **Members** der Klasse.

# Implementierung

```
1 public class Rational {
2     // Attribute:
3     private int zaehler, nenner;
4     // Konstruktoren:
5     public Rational(int x, int y) {
6         zaehler = x;
7         nenner = y;
8     }
9     public Rational(int x) {
10        zaehler = x;
11        nenner = 1;
12    }
```

# Implementierung

```
13 // Objekt-Methoden:
14 public Rational add(Rational r) {
15     int x = zaehler * r.nenner + r.zaehler * nenner;
16     int y = nenner * r.nenner;
17     return new Rational(x,y);
18 }
19 public boolean isEqual(Rational r) {
20     return zaehler * r.nenner == r.zaehler * nenner;
21 }
22 public String toString() {
23     if (nenner == 1) return "" + zaehler;
24     if (nenner > 0) return zaehler + "/" + nenner;
25     return (-zaehler) + "/" + (-nenner);
26 }
27 public static Rational[] intToRationalArray(int[] a) {
28     Rational[] b = new Rational[a.length];
29     for(int i=0; i < a.length; ++i)
30         b[i] = new Rational(a[i]);
31     return b;
32 }
```

# Implementierung

```
33 // Jetzt kommt das Hauptprogramm
34 public static void main(String[] args) {
35     Rational a = new Rational(1,2);
36     Rational b = new Rational(3,4);
37     Rational c = a.add(b);
38
39     System.out.println(c.toString());
40 } // end of main()
41 } // end of class Rational
```

## Bemerkungen:

- ▶ Jede Klasse **sollte** in einer separaten Datei des entsprechenden Namens stehen.
- ▶ Die Schlüsselworte **public** bzw. **private** klassifizieren, für wen die entsprechenden Members sichtbar, d.h. zugänglich sind.
- ▶ **private** heißt: nur für Members der gleichen Klasse sichtbar.
- ▶ **public** heißt: innerhalb des gesamten Programms sichtbar.
- ▶ Nicht klassifizierte Members sind nur innerhalb des aktuellen **Package** sichtbar.

Falls kein Konstruktor definiert wird, stellt **Java** einen **Default-Konstruktor** zur Verfügung, welcher keine Argumente entgegennimmt.

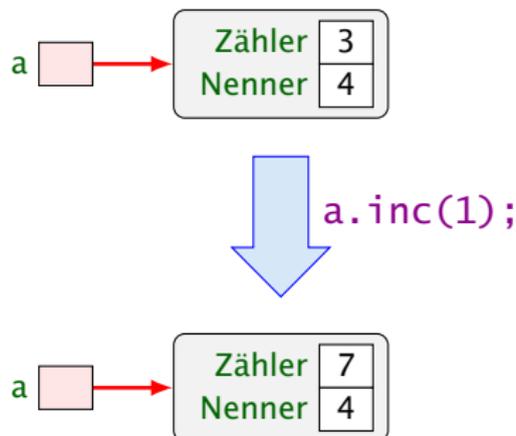
## Bemerkungen:

- ▶ Konstruktoren haben den gleichen Namen wie die Klasse.
- ▶ Es kann mehrere geben, sofern sie sich im Typ ihrer Argumente unterscheiden.
- ▶ Konstruktoren haben **keine** Rückgabewerte und darum auch keinen Rückgabetyt.
- ▶ Methoden haben dagegen **stets** einen Rückgabe-Typ, evt. **void**.

```
1 public void inc(int b) {  
2     zaehler = zaehler + b * nenner;  
3 }
```

# Implementierung

Die Objekt-Methode `inc()` modifiziert das Objekt, für das sie aufgerufen wird.



Eine Klasse, deren Objekte nach der Initialisierung nicht verändert werden können, ist **immutable**. Mit dem Hinzufügen der Operation `inc` wird die Klasse `Rational` **mutable**. Es ist eine sehr wichtige Designentscheidung ob man eine Klasse als mutable oder immutable implementiert.

# Implementierung

- ▶ Die Objektmethode `isEqual()` ist nötig, da der Operator `==` bei Objekten die **Identität** der Objekte testet, d.h. die Gleichheit der Referenz!!!
- ▶ Die Objektmethode `toString()` liefert eine **String**-Darstellung des Objekts.
- ▶ Sie wird implizit aufgerufen, wenn das Objekt als Argument für die Konkatenation `+` auftaucht.
- ▶ Innerhalb einer Objektmethode/eines Konstruktors kann auf die Attribute des Objektes **direkt** zugegriffen werden.
- ▶ **private**-Klassifizierung bezieht sich auf die Klasse nicht das Objekt: die Attribute **aller Rational**-Objekte sind für **add** sichtbar!!!

`isEqual` ist auch nötig, da Brüche mit unterschiedlichen Werten für Zähler und Nenner trotzdem gleich sind. Normalerweise sollte man für den Gleichheitstest eine Methode `equals` definieren, da diese Methode von verschiedenen Java-Klassen vorausgesetzt wird. Für eine vernünftige Implementierung dieser Methode benötigen wir aber weitere Konzepte...

# UML-Diagramm

Eine graphische Visualisierung der Klasse `Rational`, die nur die wesentliche Funktionalität berücksichtigt, könnte so aussehen:

<b>Rational</b>	
-	zaehler : int
-	nenner : int
+	add (y : Rational) : Rational
+	isEqual (y : Rational) : boolean
+	toString () : String

# Diskussion und Ausblick

- ▶ Solche Diagramme werden von der **UML**, d.h. der **Unified Modelling Language**, bereitgestellt, um Software-Systeme zu entwerfen (↑**Software Engineering**)
- ▶ Für einzelne Klassen lohnt sich ein solches Diagramm nicht wirklich.
- ▶ Besteht ein System aber aus **sehr vielen** Klassen, kann man damit die **Beziehungen** zwischen den Klassen verdeutlichen.

## Achtung:

**UML** wurde nicht speziell für **Java** entwickelt. Darum werden Typen abweichend notiert. Auch lassen sich manche Ideen nicht oder nur schlecht modellieren.

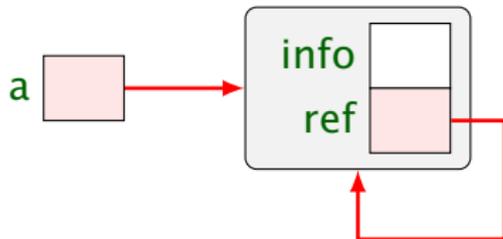
## 10.1 Selbstreferenzen

```
1 public class Cyclic {
2     private int info;
3     private Cyclic ref;
4     // Konstruktor
5     public Cyclic() {
6         info = 17;
7         ref = this;
8     }
9     ...
10 } // end of class Cyclic
```

**Innerhalb** eines Members kann man mit Hilfe von **this** auf das aktuelle Objekt selbst zugreifen!

## 10.1 Selbstreferenzen

Für `Cyclic a = new Cyclic();` ergibt das



# Modellierung einer Selbstreferenz



Die Rautenverbindung heißt auch **Aggregation**

Das Klassendiagramm vermerkt, dass jedes Objekt der Klasse **Cyclic** **einen** Verweis mit dem Namen **ref** auf ein weiteres Objekt der Klasse **Cyclic** enthält.

Ausserdem, dass jedes **Cyclic**-Objekt in genau **einem** anderen **Cyclic**-Objekt die Rolle **ref** übernimmt.

# Die this-Referenz

Woher kommt die Referenz `this`?

- ▶ Einem Aufruf einer Objektmethode (z.B. `a.inc()`) oder eines Konstruktors wird implizit ein versteckter Parameter übergeben, der auf das Objekt (hier `a`) zeigt.

- ▶ Die Signatur von `inc(int x)` ist eigentlich:

```
void inc(Rational this, int x);
```

- ▶ Zugriffe auf Objektattribute innerhalb einer Objektmethode werden mithilfe dieser Referenz aufgelöst, d.h.:

```
zaehler = zaehler + b * nenner;
```

in der Methode `inc()` ist eigentlich

```
this.zaehler = this.zaehler + b * this.nenner;
```

## 10.2 Klassenattribute

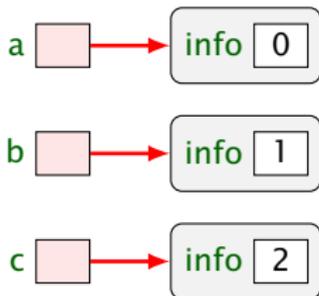
- ▶ Objektattribute werden für jedes Objekt neu angelegt,
- ▶ **Klassenattribute** einmal für die gesamte Klasse,
- ▶ Klassenattribute erhalten die Qualifizierung **static**

```
1 public class Count {
2     private static int count = 0;
3     private int info;
4     // Konstruktor
5     public Count() {
6         info = count++;
7     }
8     ...
9 } // end of class Count
```

## 10.2 Klassenattribute

count 3

Count b = new Count();



## 10.2 Klassenattribute

- ▶ Das Klassenattribut `count` zählt hier die Anzahl der bereits erzeugten Objekte.
- ▶ Das Objektattribut `info` enthält für jedes Objekt eine eindeutige Nummer.
- ▶ Außerhalb der Klasse `Class` kann man auf die öffentliche Klassenvariable `name` mit Hilfe von `Class.name` zugreifen.
- ▶ Funktionen und Prozeduren der Klasse `ohne` das implizite `this`-Argument heißen `Klassenmethoden` und werden auch durch das Schlüsselwort `static` kenntlich gemacht.

Man kann auf Klassenattribute und Klassenmethoden zugreifen ohne überhaupt je ein Objekt der Klasse zu instantiieren.

# Beispiel

In `Rational` definieren wir:

```
public static Rational[] intToRationalArray(int[] a) {  
    Rational[] b = new Rational[a.length];  
    for(int i=0; i < a.length; ++i)  
        b[i] = new Rational(a[i]);  
    return b;  
}
```

- ▶ Die Funktion erzeugt für ein Feld von `int`'s ein entsprechendes Feld von `Rational`-Objekten.
- ▶ Außerhalb der Klasse `Class` kann die öffentliche Klassenmethode `meth()` mit Hilfe von `Class.meth(...)` aufgerufen werden.

# 11 Abstrakte Datentypen

## Erinnerung:

- ▶ Abstrakter Datentyp spezifiziert nur die Operationen
- ▶ Implementierung und andere Details sind verborgen

Dieses ist ein sehr puristischer Ansatz. Im folgenden werden wir häufig nicht ganz so streng sein, und manchmal Zugriff auf die Datenstruktur auch über direkte Manipulation von Attributen gestatten.

# 11.1 Listen

**Nachteil** von Feldern:

- ▶ feste Größe
- ▶ Einfügen neuer Elemente nicht möglich
- ▶ Streichen ebenfalls nicht

**Idee: Listen**



# Listen – Version A

- `info` : Datenelement der Liste;
- `next` : Verweis auf nächstes Element;
- `null` : leeres Objekt.

Eigentlich ist die Liste, die wir hier implementieren, kein abstrakter Datentyp. Die Operationen `insert` und `delete` benötigen eine Referenz auf ein Listenelement hinter dem eingefügt bzw. gelöscht wird. Dies erlaubt keine gute **Datenkapselung**.

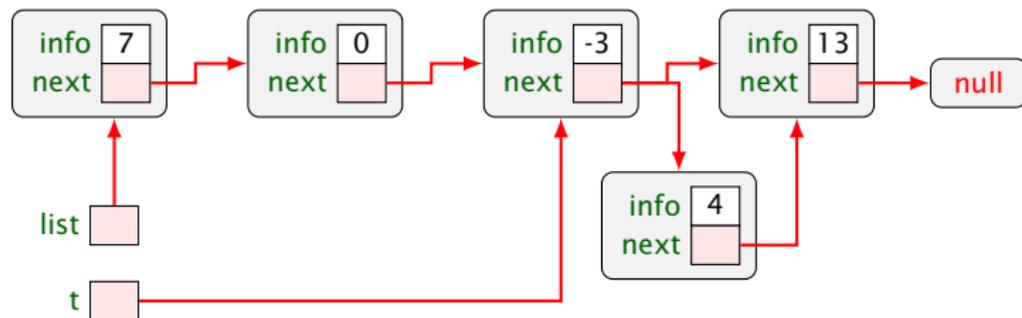
## Operationen:

- `void insert(int x)` : fügt neues `x` hinter dem aktuellen (ersten) Element ein;
- `void delete()` : entfernt Knoten hinter dem aktuellen (ersten) Element;
- `String toString()` : liefert eine `String`-Darstellung.

Modellierung als UML-Diagramm:

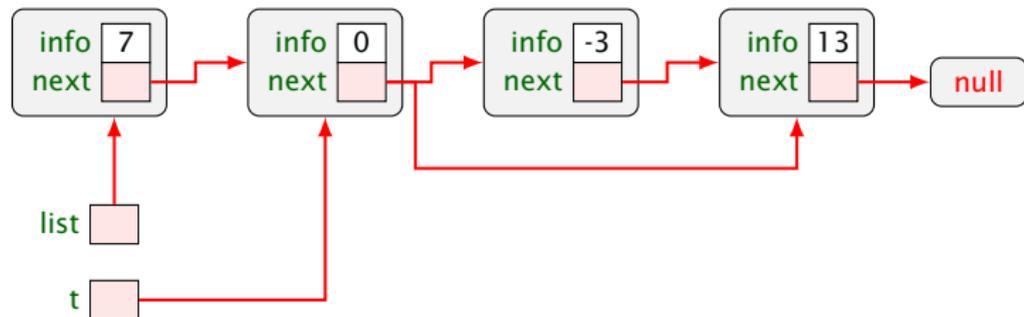


# Listen - Insert



`t.insert(4)`

# Listen - Delete



`t.delete()`

# 11.1 Listen

## Weitere Operationen:

- ▶ Liste auf Leerheit testen
- ▶ Neue Listen erzeugen ( $\Rightarrow$  Konstruktoren)
  - ▶ z.B. eine einelementige Liste
  - ▶ eine bestehende Liste verlängern
- ▶ Umwandlung zwischen Listen und Feldern...

Das `null`-Objekt versteht keinerlei Objektmethoden; da wir `null` als leere Liste interpretieren, müssen wir uns etwas einfallen lassen...

# Listen - Implementierung

```
1 public class List {
2     public int info;
3     public List next;
4
5     // Konstruktoren:
6     public List (int x, List l) {
7         info = x;
8         next = l;
9     }
10    public List (int x) {
11        info = x;
12        next = null;
13    }
14 // continued...
```

# Listen - Implementierung

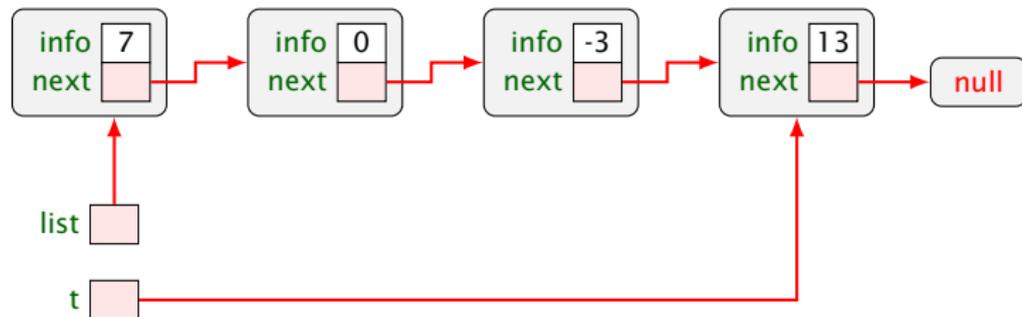
```
15 // Objekt-Methoden:
16 public void insert(int x) {
17     next = new List(x,next);
18 }
19 public void delete() {
20     if (next != null)
21         next = next.next;
22 }
23 public String toString() {
24     String result = "[" + info;
25     for(List t = next; t != null; t = t.next)
26         result = result + ", " + t.info;
27     return result + "]";
28 }
29 // continued...
```

- ▶ Die Attribute sind `public` und daher beliebig einsehbar und modifizierbar; sehr fehleranfällig.
- ▶ `insert()` legt einen neuen Listenknoten an, und fügt ihn hinter dem aktuellen Knoten ein.
- ▶ `delete()` setzt den aktuellen `next`-Verweis auf das übernächste Element um.

## Achtung:

Wenn `delete()` mit dem letzten Listenelement aufgerufen wird, zeigt `next` auf `null`; wir tun dann nichts...

# Listen - Delete



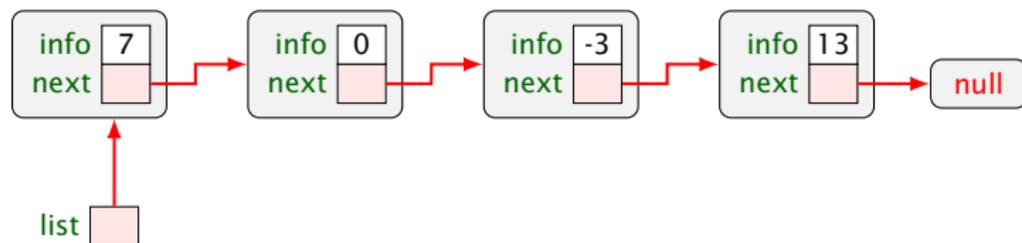
`t.delete()`

Weil Objektmethoden nur für von `null` verschiedene Objekte aufgerufen werden können, kann die leere Liste nicht mittels `toString()` als `String` dargestellt werden.

Der Konkatenations-Operator `+` ist so schlau, vor Aufruf von `toString()` zu überprüfen, ob ein `null`-Objekt vorliegt. Ist das der Fall, wird "null" ausgegeben.

Für eine andere Darstellung benötigen wir eine Klassenmethode `toString(List l)`;

# Listen - toString()



```
write("" + list);
```

liefert: „[7, 0, -3, 13]“

# Listen - toString()



```
write("" + list);
```

liefert: „null“

# Listen - Implementierung

```
30 // Klassen-Methoden:
31 public static boolean isEmpty(List l) {
32     return (l == null);
33 }
34 public static String toString(List l) {
35     if (l == null)
36         return "[]";
37     else
38         return l.toString();
39 }
40 // continued...
```

Der Aufruf erfolgt dann über `List.isEmpty(a)` bzw. `List.toString(a)` für eine Liste `a`. Leider funktioniert letzteres nicht zusammen mit dem Konkatenationsoperator. Über diesen wird weiterhin „null“ ausgegeben.

# Listen - Implementierung

```
41     public static List arrayToList(int[] a) {
42         List result = null;
43         for(int i = a.length-1; i >= 0; --i)
44             result = new List(a[i], result);
45         return result;
46     }
47     public int[] listToArray() {
48         List t = this;
49         int n = length();
50         int[] a = new int[n];
51         for(int i = 0; i < n; ++i) {
52             a[i] = t.info;
53             t = t.next;
54         }
55         return a;
56     }
57     // continued...
```

## Listen - Implementierung

- ▶ Damit das erste Element der Ergebnisliste `a[0]` enthält, beginnt die Iteration in `arrayToList()` beim **letzten** Element.
- ▶ `listToArray()` ist als Objekt-Methode realisiert und funktioniert darum nur für **nicht-leere** Listen.
- ▶ Wir benötigen die Länge einer Liste:

```
58     private int length() {
59         int result = 1;
60         for(List t = next; t != null; t = t.next)
61             result++;
62         return result;
63     }
64 } // end of class List
```

# Listen – Implementierung

- ▶ Weil `length()` als `private` deklariert ist, kann es nur von den Methoden der Klasse `List` benutzt werden.
- ▶ Damit `length()` auch für `null` funktioniert, hätten wir analog zu `toString()` auch noch eine Klassen-Methode `int length(List l)` definieren können.
- ▶ Diese Klassen-Methode würde uns ermöglichen, auch eine Klassen-Methode `static int[] listToArray (List l)` zu definieren, die auch für leere Listen definiert ist.

Es ist eine generelle Stilfrage ob man eine leere Liste als `null` implementieren sollte. Die meisten `Java`-Bibliotheken nutzen ein spezielles Objekt, das eine leere Liste/Collection etc. repräsentiert. Dann kann man z.B. immer `a.toString()` aufrufen anstatt `List.toString(a)` etc.

# Mergesort – Sortieren durch Mischen

Mergesort ist ein schneller Sortieralgorithmus der auf auf der Mischoperation beruht.



John von Neumann (1945)

# Mergesort – Sortieren durch Mischen

## Die Mischoperation

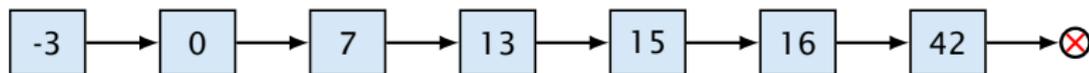
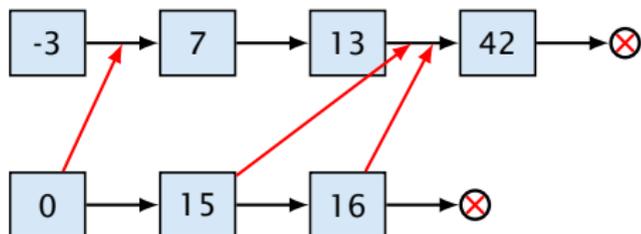
Input: zwei sortierte Listen

Output: eine gemeinsame sortierte Liste

Später bauen wir damit einen Sortieralgorithmus...

## Beispiel – Mischen

Hier benutzen wir das Symbol  $\otimes$  für das `null`-Objekt.



# Mergesort – Sortieren durch Mischen

## Idee:

- ▶ Konstruiere sukzessive die Ausgabeliste aus den Argumentlisten.
- ▶ Um das nächste Element für die Ausgabe zu finden, vergleichen wir die beiden kleinsten Elemente der noch verbliebenen Inputlisten.
- ▶ Falls  $m$  und  $n$  die Längen der Argumentlisten sind, sind offenbar maximal nur  $m + n - 1$  Vergleiche nötig.

# Beispiel – Mischen

Animation ist nur in der  
Vorlesungsversion der Folien  
vorhanden.

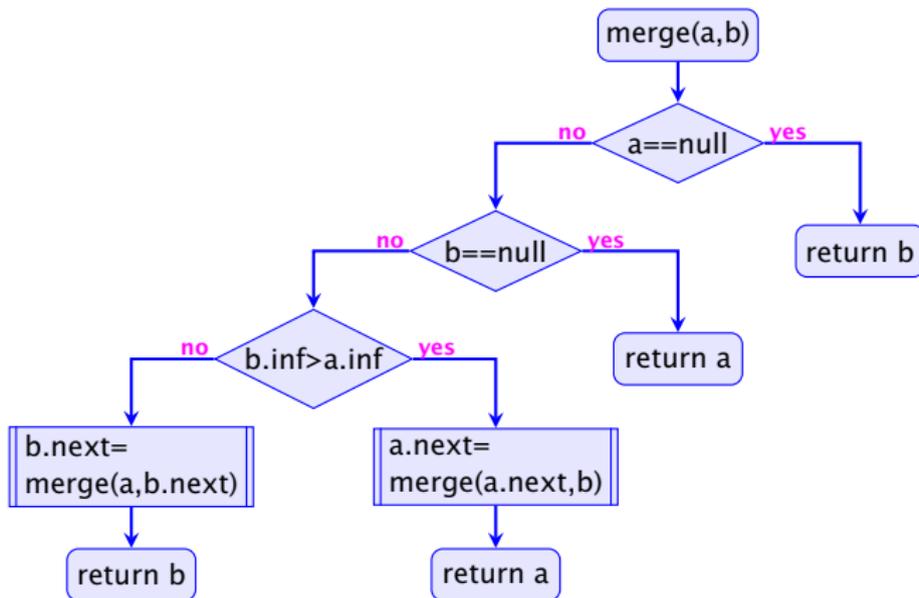
## Rekursive Implementierung

- ▶ Falls eine der beiden Listen **a** und **b** leer ist, geben wir die andere aus.
- ▶ Andernfalls gibt es in jeder der beiden Listen ein erstes (kleinstes) Element.
- ▶ Von diesen beiden Elementen nehmen wir ein kleinstes.
- ▶ Dahinter hängen wir die Liste, die wir durch Mischen der verbleibenden Elemente erhalten. . .

# Mergesort – Implementierung

```
1 public static List merge(List a, List b) {
2     if (b == null)
3         return a;
4     if (a == null)
5         return b;
6     if (b.info > a.info) {
7         a.next = merge(a.next, b);
8         return a;
9     } else {
10        b.next = merge(a, b.next);
11        return b;
12    }
13 }
```

# Kontrollfluss

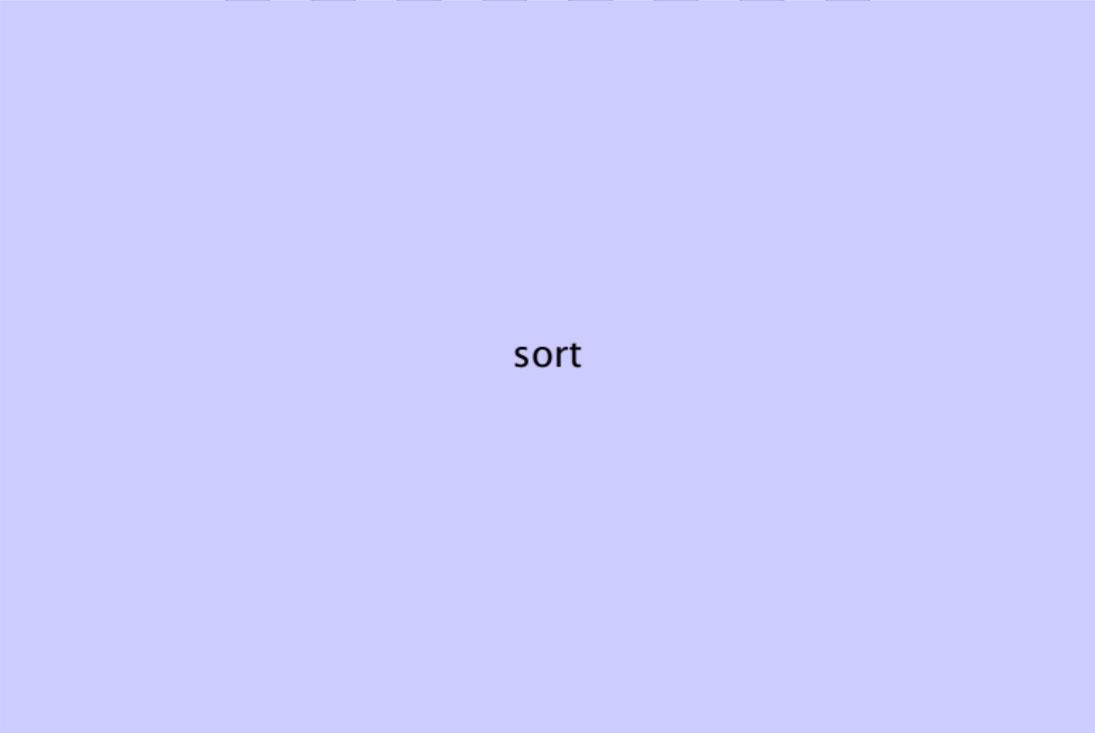


## Sortieren durch Mischen:

1. Teile zu sortierende Liste in zwei Teillisten;
2. sortiere jede Hälfte für sich;
3. mische die Ergebnisse!

# Mergesort

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



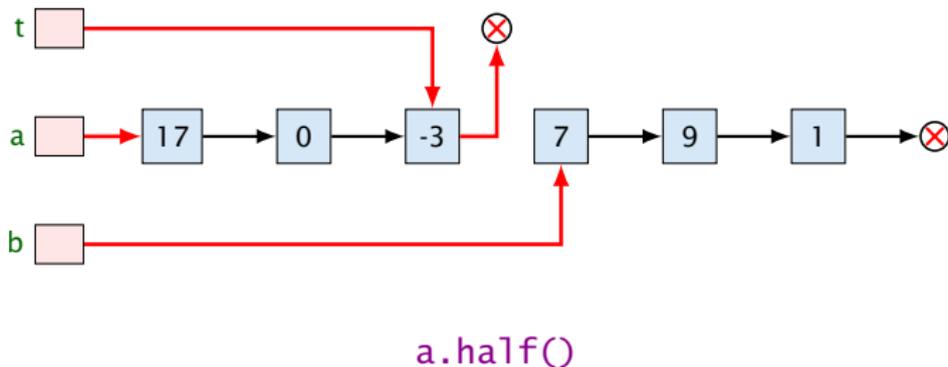
# Mergesort – Implementierung

```
1 public static List sort(List a) {
2     if (a == null || a.next == null)
3         return a;
4     List b = a.half(); // Halbiere!
5     a = sort(a);
6     b = sort(b);
7     return merge(a,b);
8 }
```

# Mergesort – Implementierung

```
1 public List half() {
2     int n = length();
3     List t = this;
4     for (int i = 0; i < n/2-1; i++)
5         t = t.next;
6     List result = t.next;
7     t.next = null;
8     return result;
9 }
```

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



# Mergesort – Analyse

- ▶ Sei  $V(n)$  die Anzahl der Vergleiche, die Mergesort maximal zum Sortieren einer Liste der Länge  $n$  benötigt.  
Dann gilt:

$$V(1) = 0$$

$$V(2n) \leq 2 \cdot V(n) + 2 \cdot n$$

- ▶ Für  $n = 2^k$ , sind das dann nur  $k \cdot n = n \log_2 n$  Vergleiche!!!

Dies ist wesentlich effizienter als die Methode „Sortieren durch Einfügen“, die wir vorher kennengelernt haben.

## Achtung:

- ▶ Unsere Funktion `sort()` zerstört ihr Argument!
- ▶ Alle Listenknoten der Eingabe werden weiterverwendet.
- ▶ Die Idee des Sortierens durch Mischen könnte auch mithilfe von Feldern realisiert werden (wie?)
- ▶ Sowohl das Mischen wie das Sortieren könnte man statt rekursiv auch iterativ implementieren (wie?)

## 11.2 Keller (Stacks)

### Operationen:

- `boolean isEmpty()` : testet auf Leerheit;
- `int pop()` : liefert oberstes Element;
- `void push(int x)` : legt `x` oben auf dem Keller ab;
- `String toString()` : liefert eine String-Darstellung

Weiterhin müssen wir einen leeren Keller anlegen können.

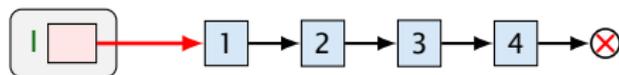
# Modellierung Stack

Stack	
+ Stack	()
+ isEmpty	() : boolean
+ push	(x : int) : void
+ pop	() : int
+ toString	() : String

# Stack via List

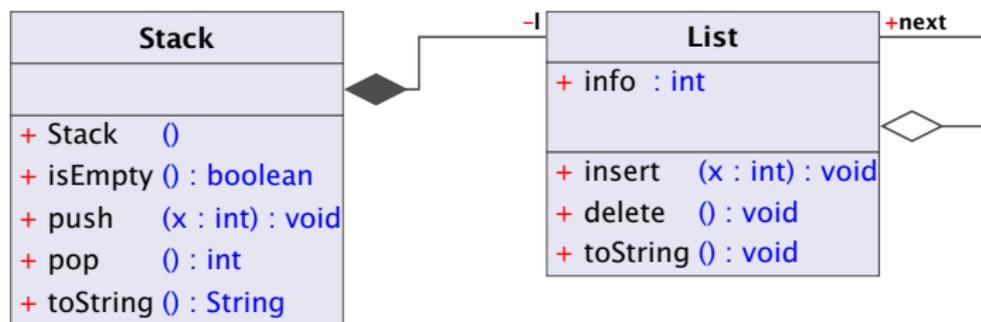
## Idee

- ▶ Realisiere **Stack** mithilfe einer Liste:



- ▶ Das Attribut **l** zeigt auf das oberste Kellerelement.

# Modellierung Stack via List



Die **gefüllte Raute** bezeichnet eine **Komposition**. Die Liste existiert nur solange wie der Stack (d.h. wird üblicherweise durch diesen erzeugt und zerstört). Außerdem kann die Liste nur Teil eines Stacks sein.

# Stack – Implementierung

```
1 public class Stack {
2     private List l;
3     // Konstruktor :
4     public Stack() {
5         l = null;
6     }
7     // Objektmethoden :
8     public boolean isEmpty() {
9         return l == null;
10    }
11    // continued...
```

# Stack - Implementierung

```
12     public int pop() {
13         int result = l.info;
14         l = l.next;
15         return result;
16     }
17     public void push(int a) {
18         l = new List(a,l);
19     }
20     public String toString() {
21         return List.toString(l);
22     }
23 } // end of class Stack
```

# Bemerkungen

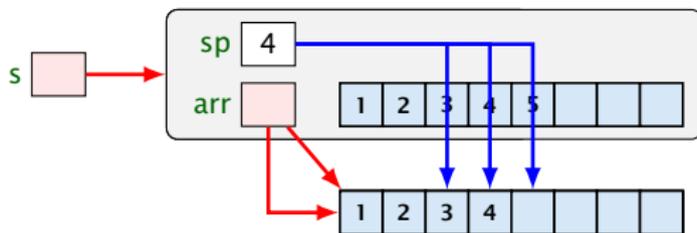
- ▶ Implementierung ist sehr einfach;
- ▶ nutzt gar nicht alle Features von `List`;
- ▶ **Nachteil:** Die Listenelemente sind evt. über den gesamten Speicher verstreut:  
⇒ schlechtes Cache-Verhalten!

## Zweite Idee:

- ▶ Realisiere Keller mithilfe eines Feldes und eines Stackpointers, der auf das oberste Element zeigt.
- ▶ Läuft das Feld über, ersetzen wir es durch ein größeres.

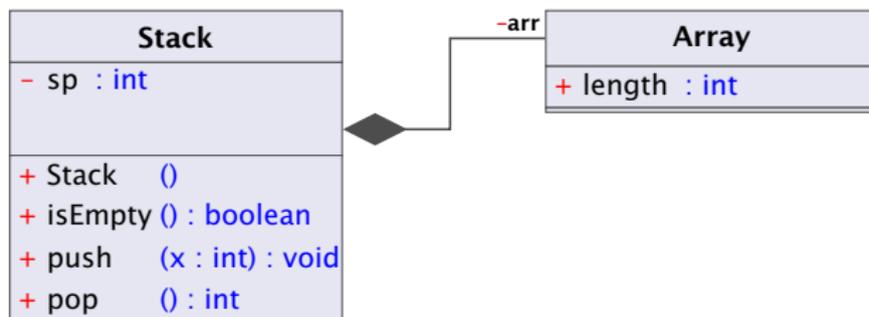
# Stack via Array

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



`s.push(5)`

# Modellierung Stack



# Implementierung

```
1 public class Stack {
2     private int sp;
3     private int[] arr;
4     // Konstruktoren:
5     public Stack() {
6         sp = -1;
7         arr = new int[4];
8     }
9     // Objekt-Methoden:
10    public boolean isEmpty() {
11        return sp < 0;
12    }
13    // continued...
```

# Implementierung

```
14     public int pop() {
15         return arr[sp--];
16     }
17     public void push(int x) {
18         ++sp;
19         if (sp == arr.length) {
20             int[] b = new int[2*sp];
21             for (int i = 0; i < sp; ++i) b[i] = arr[i];
22             arr = b;
23         }
24         arr[sp] = x;
25     }
26     public String toString() {...}
27 } // end of class Stack
```

## 11.2 Keller (Stacks)

### Nachteil:

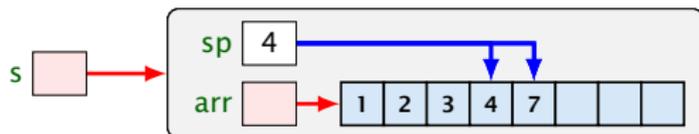
- ▶ Es wird zwar neuer Platz allokiert, aber nie welcher freigegeben.

### Erste Idee:

- ▶ Sinkt der Pegel wieder auf die Hälfte, geben wir diese frei. . .

# Stack via Array

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



ssppøø(ø)

## 11.2 Keller (Stacks)

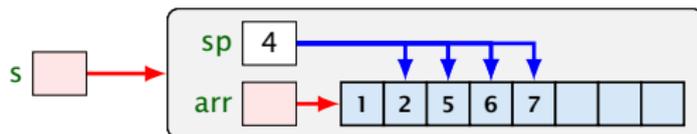
- ▶ Im schlimmsten Fall müssen bei **jeder** Operation sämtliche Elemente kopiert werden.

### Zweite Idee:

- ▶ Wir geben erst frei, wenn der Pegel auf **ein Viertel** fällt – und dann auch nur die Hälfte!

# Stack via Array

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



ssppøø(ø)

## 11.2 Keller (Stacks)

### Beobachtung:

- ▶ Vor jedem Kopieren werden mindestens halb so viele Operationen ausgeführt wie Elemente kopiert werden.
- ▶ Gemittelt über die gesamte Folge der Operationen werden pro Operation maximal zwei Zahlen kopiert (↑amortisierte Aufwandsanalyse)

# Implementierung

```
1 public int pop() {
2     int result = arr[sp];
3     if (sp == arr.length/4 && sp >= 2) {
4         int[] b = new int[2*sp];
5         for(int i = 0; i < sp; ++i)
6             b[i] = arr[i];
7         arr = b;
8     }
9     sp--;
10    return result;
11 }
```

## 11.3 Schlangen (Queues)

(Warte-) Schlangen verwalten ihre Elemente nach dem **FIFO**-Prinzip (**F**irst-**I**n-**F**irst-**O**ut).

### Operationen:

`boolean isEmpty()` : testet auf Leerheit;  
`int dequeue()` : liefert erstes Element;  
`void enqueue(int x)` : reiht `x` in die Schlange ein;  
`String toString()` : liefert eine String-Darstellung.

Weiterhin müssen wir eine leere Schlange anlegen können.

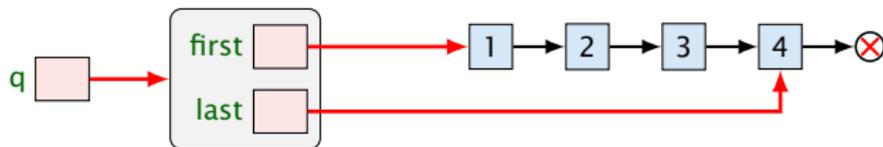
# Modellierung Queue

Queue	
+	Queue ()
+	isEmpty () : boolean
+	enqueue (x : int) : void
+	dequeue () : int
+	toString () : String

# Queue via List

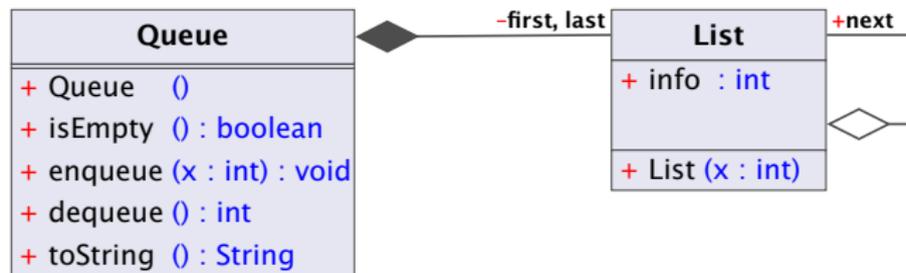
## Erste Idee:

- ▶ Realisiere Schlange mithilfe einer Liste:



- ▶ `first` zeigt auf das nächste zu entnehmende Element;
- ▶ `last` zeigt auf das Element hinter dem eingefügt wird;

# Modellierung: Queue via List



Objekte der Klasse **Queue** enthalten zwei Verweise auf Objekte der Klasse **List**.

# Queue – Implementierung

```
1 public class Queue {
2     private List first, last;
3     // Konstruktor:
4     public Queue() {
5         first = last = null;
6     }
7     // Objekt-Methoden:
8     public boolean isEmpty() {
9         return List.isEmpty(first);
10    }
11    // continued...
```

## Queue – Implementierung

```
12     public int dequeue() {
13         int result = first.info;
14         if (last == first) last = null;
15         first = first.next;
16         return result;
17     }
18     public void enqueue(int x) {
19         if (first == null)
20             first = last = new List(x);
21         else {
22             last.insert(x);
23             last = last.next;
24         }
25     }
26     public String toString() {
27         return List.toString(first);
28     }
29 } // end of class Queue
```

# Bemerkungen

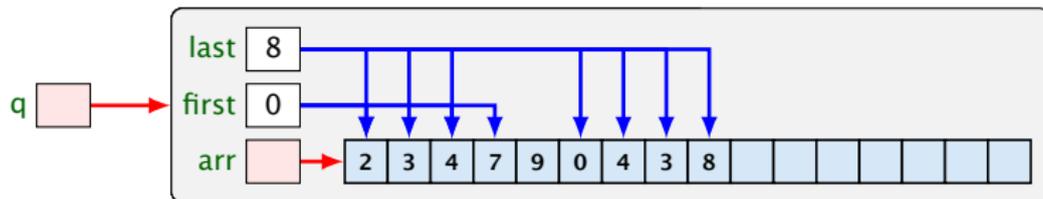
- ▶ Implementierung ist wieder sehr einfach;
- ▶ nutzt mehr Features von `List`;
- ▶ **Nachteil:** Die Listenelemente sind evt. über den gesamten Speicher verstreut:  
⇒ schlechtes Cache-Verhalten!

## Zweite Idee:

- ▶ Realisiere Schlange mithilfe eines Feldes und zweier Pointer, die auf erstes bzw. letztes Element zeigen.
- ▶ Läuft das Feld über, ersetzen wir es durch ein größeres.

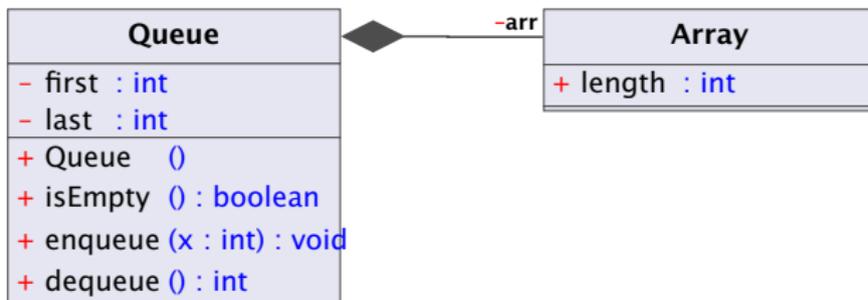
# Queue via Array

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



`q.dequeue()`

# Modellierung: Queue via Array



# Implementierung

```
1 public class Queue {
2     private int first, last;
3     private int[] arr;
4     // Konstruktor:
5     public Queue() {
6         first = last = -1;
7         arr = new int[4];
8     }
9     // Objekt-Methoden:
10    public boolean isEmpty() { return first == -1; }
11    public String toString() {...}
12    //continued...
```

# Implementierung von enqueue()

- ▶ Falls die Schlange leer war, muss `first` und `last` auf 0 gesetzt werden.
- ▶ Andernfalls ist das Feld `a` genau dann voll, wenn das Element `x` an der Stelle `first` eingetragen werden sollte.
- ▶ In diesem Fall legen wir ein Feld doppelter Größe an.

Die Elemente `a[first], ..., a[a.length-1]`, `a[0], a[1], ..., a[first-1]` kopieren wir nach `b[0], ..., b[a.length-1]`.

- ▶ Dann setzen wir `first = 0; last = a.length; a = b;`
- ▶ Nun kann `x` an der Stelle `a[last]` abgelegt werden.

# Implementierung

```
13     public void enqueue(int x) {
14         if (first == -1) {
15             first = last = 0;
16         } else {
17             int n = arr.length;
18             last = (last + 1) % n;
19             if (last == first) {
20                 int[] b = new int[2*n];
21                 for (int i = 0; i < n; ++i)
22                     b[i] = arr[(first + i) % n];
23                 first = 0;
24                 last = n;
25                 arr = b;
26             }
27         } // end if and else
28         arr[last] = x;
29     }
```

# Implementierung von dequeue()

- ▶ Falls nach Entfernen von  $a[\text{first}]$  die Schlange leer ist, werden  $\text{first}$  und  $\text{last}$  auf  $-1$  gesetzt.
- ▶ Andernfalls wird  $\text{first}$  um  $1$  (modulo der Länge von  $\text{arr}$ ) inkrementiert.

Falls danach höchstens noch  $n/4$  Elemente da sind, werden diese an die Positionen  $b[0], \dots, b[\text{num}-1]$  in einem neuen Array kopiert.

# Implementierung

```
30 public int dequeue() {
31     int result = arr[first];
32     if (last == first) {
33         first = last = -1;
34         return result;
35     }
36     int n = arr.length;
37     first = (first+1) % n;
38     // Anzahl der Elemente ist (last-first) mod n + 1
39     // aber % in Java ist keine modulo-Operation
40     int num = (last-first+n) % n + 1;
41     // continued...
```

# Implementierung

```
42     if (num > 1 && num <= n/4) {
43         int[] b = new int[n/2];
44         for (int i = 0; i < num; ++i)
45             b[i] = arr[(first + i) % n];
46         first = 0;
47         last = num - 1;
48         arr = b;
49     }
50     return result;
51 }
```

# 11 Abstrakte Datentypen

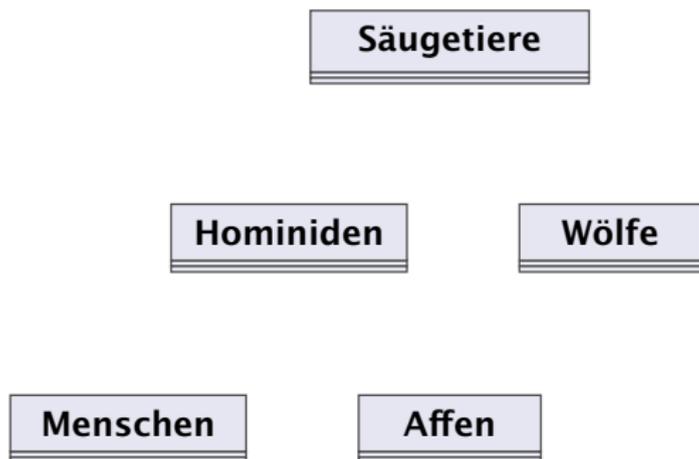
## Zusammenfassung

- ▶ Der Datentyp `List` ist nicht sehr **abstrakt**, dafür extrem flexibel (gut für **rapid prototyping**)
- ▶ Für die **nützlichen** (eher) abstrakten Datentypen `Stack` und `Queue` lieferten wir zwei Implementierungen. Eine sehr einfache, und eine cache-effiziente.
- ▶ **Achtung:** oft werden bei diesen Datentypen noch weitere Operationen zur Verfügung gestellt.

# 12 Vererbung

## Beobachtung

Oft werden mehrere Klassen von Objekten benötigt, die zwar ähnlich, aber doch verschieden sind.



# 12 Vererbung

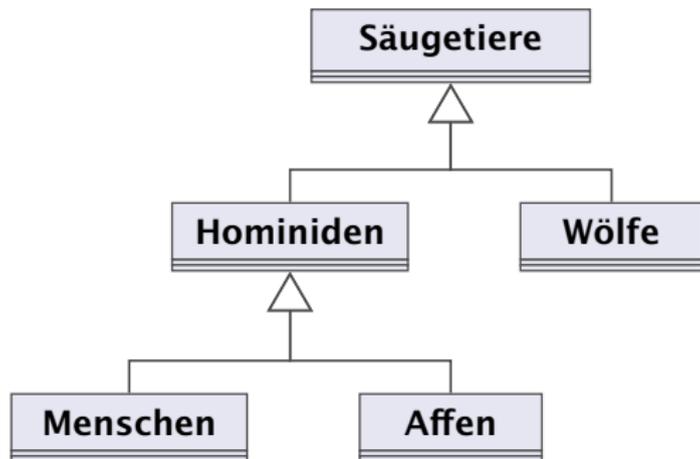
## Idee:

- ▶ Finde Gemeinsamkeiten heraus!
- ▶ Organisiere in einer Hierarchie!
- ▶ Implementiere zuerst was allen gemeinsam ist!
- ▶ Implementiere dann nur noch den Unterschied!

⇒ inkrementelles Programmieren

⇒ Software Reuse

# 12 Vererbung

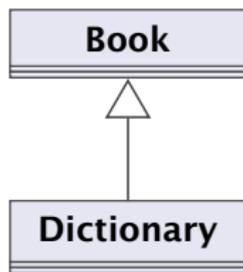


# 12 Vererbung

## Prinzip

- ▶ Die Unterklasse verfügt über all Members der Oberklasse und eventuell noch über weitere.
- ▶ Das Übernehmen von Members der Oberklasse in die Unterklasse nennt man **Vererbung** (oder **inheritance**).

## Beispiel



# Implementierung

```
1 public class Book {
2     protected int pages;
3     public Book() {
4         pages = 150;
5     }
6     public void page_message() {
7         System.out.println("Number of pages: "+pages);
8     }
9 } // end of class Book
10 // continued...
```

# Implementierung

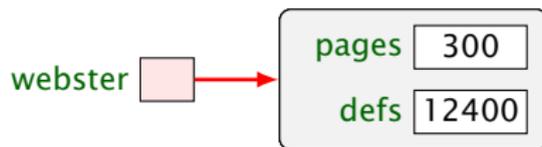
```
1 public class Dictionary extends Book {
2     private int defs;
3     public Dictionary(int x) {
4         pages = 2*pages;
5         defs = x;
6     }
7     public void defs_message() {
8         System.out.println("Number of defs: "+defs);
9         System.out.println("Defs per page: "+defs/pages);
10    }
11 } // end of class Dictionary
```

# Erläuterungen

- ▶ `class A extends B { ... }` deklariert die Klasse `A` als Unterklasse der Klasse `B`.
- ▶ Alle Members von `B` stehen damit automatisch auch der Klasse `A` zur Verfügung.
- ▶ Als `protected` klassifizierte Members sind auch in der Unterklasse `sichtbar`.
- ▶ Als `private` deklarierte Members können dagegen in der Unterklasse `nicht` direkt aufgerufen werden, da sie dort nicht sichtbar sind.
- ▶ Wenn ein Konstruktor der Unterklasse `A` aufgerufen wird, wird `implizit` zuerst der Konstruktor `B()` der Oberklasse aufgerufen.

# Beispiel

Dictionary webster = new Dictionary(12400);  
liefert



# Methodenaufruf

```
1 public class Words {
2     public static void main(String[] args) {
3         Dictionary webster = new Dictionary(12400);
4         webster.page_message();
5         webster.defs_message();
6     } // end of main
7 } // end of class Words
```

- ▶ Das neue Objekt `webster` enthält die Attribute `pages` und `defs`, sowie die Objekt-Methoden `page_message()` und `defs_message()`.
- ▶ Kommen in der Unterklasse nur weitere Members hinzu, spricht man von einer `is_a`-Beziehung. (Oft müssen aber Objekt-Methoden der Oberklasse in der Unterklasse undefiniert werden.)

# Methodenaufruf

Die Programmausführung liefert:

Number of pages: 300

Number of defs: 12400

Defs per page: 41

## 12.1 Das Schlüsselwort `super`

- ▶ Manchmal ist es erforderlich, in der Unterklasse **explizit** die Konstruktoren oder Objekt-Methoden der Oberklasse aufzurufen. Das ist der Fall, wenn
  - ▶ Konstruktoren der Oberklasse aufgerufen werden sollen, die Parameter besitzen;
  - ▶ Objekt-Methoden oder Attribute der Oberklasse und Unterklasse gleiche Namen haben.
- ▶ Zur Unterscheidung der aktuellen Klasse von der Oberklasse dient das Schlüsselwort **super**.

# Beispiel

```
1 public class Book {
2     protected int pages;
3     public Book(int x) {
4         pages = x;
5     }
6     public void message() {
7         System.out.println("Number of pages: "+pages);
8     }
9 } // end of class Book
```

# Beispiel

```
1 public class Dictionary extends Book {
2     private int defs;
3     public Dictionary(int p, int d) {
4         super(p);
5         defs = d;
6     }
7     public void message() {
8         super.message();
9         System.out.println("Number of defs: "+defs);
10        System.out.println("Defs per page: "+defs/pages);
11    }
12 } // end of class Dictionary
```

## „super“ als Konstruktoraufruf

- ▶ `super(...)`; ruft den entsprechenden Konstruktor der Oberklasse auf.
- ▶ Analog gestattet `this(...)`; den entsprechenden Konstruktor der eigenen Klasse aufzurufen.
- ▶ Ein solcher expliziter Aufruf muss stets ganz am Anfang eines Konstruktors stehen.

## „super.“ zum Zugriff auf members der Oberklasse

Deklariert eine Klasse **A** einen Member **memb** gleichen Namens wie in einer Oberklasse, so ist nur noch der Member **memb** aus **A** sichtbar.

- ▶ Methoden mit unterschiedlichen Argument-Typen werden als verschieden angesehen.
- ▶ **super.memb** greift für das aktuelle Objekt **this** auf Attribute oder Objekt-Methoden **memb** der Oberklasse zu.
- ▶ Eine andere Verwendung von **super.** ist **nicht gestattet**.

Insbesondere Zugriffe wie **super.super.memb** sind nicht möglich.

# Verschattung von Variablen

## Falls `memb` eine Methode ist:

- ▶ Wenn `memb` eine Methode mit den gleichen Argumenttypen (in der gleichen Reihenfolge), und dem gleichen Rückgabetyper ist, dann ist zunächst nur `memb` aus `A` sichtbar (**Überschreiben**).  
(seit Java 5 darf der Rückgabetyper auch ein Untertyp des ursprünglichen Rückgabetyper sein)
- ▶ Wenn `memb` eine Methode mit unterschiedlichen Argumenttypen ist, dann sind sowohl `memb` aus `A` als auch die Methode der Oberklasse sichtbar (**Überladen**).
- ▶ Wenn die Argumenttypen übereinstimmen, aber der Rückgabetyper kein Untertyp ist, erhält man einen Compilerfehler.

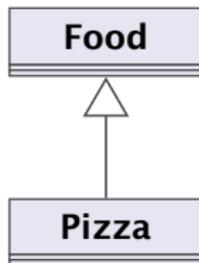
Der kovariante Rückgabetyper ist nur für Referenzdatentypen möglich.

# Verschattung von Variablen

Falls **memb** eine Variable ist:

- ▶ Direkt (d.h. ohne **super.**) ist nur **memb** aus **A** sichtbar. **memb** kann einen anderen Typ als in der Oberklasse haben.

## 12.2 Private Variablen und Methoden



Das Programm **Eating** soll die Anzahl der **Kalorien pro Mahlzeit** ausgeben.

# Implementierung

```
1 public class Eating {
2     public static void main (String[] args) {
3         Pizza special = new Pizza(275);
4         System.out.print("Calories per serving: " +
5             special.caloriesPerServing());
6     } // end of main
7 } // end of class Eating
```

# Implementierung

```
7 public class Food {
8     private int CALORIES_PER_GRAM = 9;
9     private int fat, servings;
10    public Food (int numFatGrams, int numServings) {
11        fat = numFatGrams;
12        servings = numServings;
13    }
14    private int calories() {
15        return fat * CALORIES_PER_GRAM;
16    }
17    public int caloriesPerServing() {
18        return calories() / servings;
19    }
20 } // end of class Food
```

## Implementierung + Erläuterungen

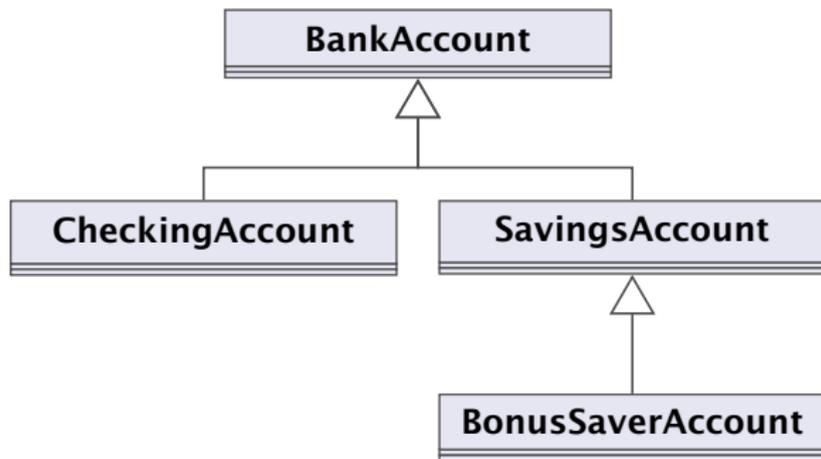
```
21 public class Pizza extends Food {
22     public Pizza (int amountFat) {
23         super(amountFat,8);
24     }
25 } // end of class Pizza
```

- ▶ Die Unterklasse `Pizza` verfügt über alle Members der Oberklasse `Food` — nicht alle `direkt` zugänglich.
- ▶ Die Attribute und die Objekt-Methode `calories()` der Klasse `Food` sind privat, und damit für Objekte der Klasse `Pizza` verborgen.
- ▶ Trotzdem können sie von der `public` Objekt-Methode `caloriesPerServing` benutzt werden.

Ausgabe des Programms:

Calories per serving: 309

## 12.3 Überschreiben von Methoden



# Aufgabe

- ▶ Implementierung von einander abgeleiteter Formen von Bankkonten.
- ▶ Jedes Konto kann eingerichtet werden, erlaubt Einzahlungen und Auszahlungen.
- ▶ Verschiedene Konten verhalten sich unterschiedlich in Bezug auf Zinsen und Kosten von Kontobewegungen.

# Einige Konten

```
1 public class Bank {
2     public static void main(String[] args) {
3         SavingsAccount savings =
4             new SavingsAccount(4321, 5028.45, 0.02);
5         BonusSaverAccount bigSavings =
6             new BonusSaverAccount (6543, 1475.85, 0.02);
7         CheckingAccount checking =
8             new CheckingAccount (9876,269.93, savings);
9         savings.deposit(148.04);    System.out.println();
10        bigSavings.deposit(41.52);  System.out.println();
11        savings.withdraw(725.55);   System.out.println();
12        bigSavings.withdraw(120.38); System.out.println();
13        checking.withdraw(320.18);  System.out.println();
14    } // end of main
15 } // end of class Bank
```

"Bank.java"

Die Parameter beschreiben id, initialer Kontostand, und Zinssatz für die ersten beiden Typen. Für den `CheckingAccount` gibt es keine Zinsen. Der dritte Parameter ist ein zugehöriger `SavingsAccount`.

# Implementierung

```
1 public class BankAccount {
2     // Attribute aller Konten-Klassen:
3     protected int account;
4     protected double balance;
5     // Konstruktor:
6     public BankAccount(int id, double initial) {
7         account = id; balance = initial;
8     }
9     // Objekt-Methoden:
10    public void deposit(double amount) {
11        balance = balance + amount;
12        System.out.println(
13            "Deposit into account " + account + "\n"
14            + "Amount:\t\t" + amount + "\n"
15            + "New balance:\t" + balance);
16    }
```

"BankAccount.java"

- ▶ Anlegen eines Kontos `BankAccount` speichert eine (hoffentlich neue) Kontonummer sowie eine Anfangseinlage.
- ▶ Die zugehörigen Attribute sind `protected`, d.h. können nur von Objekt-Methoden der Klasse bzw. ihrer Unterklassen modifiziert werden.
- ▶ die Objekt-Methode `deposit` legt Geld aufs Konto, d.h. modifiziert den Wert von `balance` und teilt die Kontobewegung mit.

# Implementierung

```
17     public boolean withdraw(double amount) {
18         System.out.println(
19             "Withdrawal from account " + account + "\n"
20             + "Amount:\t\t" + amount);
21         if (amount > balance) {
22             System.out.println(
23                 "Sorry, insufficient funds...");
24             return false;
25         }
26         balance = balance - amount;
27         System.out.println(
28             "New balance:\t" + balance);
29         return true;
30     }
31 } // end of class BankAccount
```

"BankAccount.java"

- ▶ Die Objekt-Methode `withdraw()` nimmt eine Auszahlung vor.
- ▶ Falls die Auszahlung scheitert, wird eine Mitteilung gemacht.
- ▶ Ob die Auszahlung erfolgreich war, teilt der Rückgabewert mit.
- ▶ Ein `CheckingAccount` verbessert ein normales Konto, indem im Zweifelsfall auf die Rücklage eines Sparkontos zurückgegriffen wird.

# Ein Girokonto

```
1 public class CheckingAccount extends BankAccount {
2     private SavingsAccount overdraft;
3     // Konstruktor:
4     public CheckingAccount(int id, double initial,
5         SavingsAccount savings) {
6         super(id, initial);
7         overdraft = savings;
8     }
9 }
```

"CheckingAccount.java"

## Modifiziertes withdraw()

```
8 // modifiziertes withdraw():
9 public boolean withdraw(double amount) {
10     if (!super.withdraw(amount)) {
11         System.out.println("Using overdraft...");
12         if (!overdraft.withdraw(amount-balance)) {
13             System.out.println(
14                 "Overdraft source insufficient.");
15             return false;
16         } else {
17             balance = 0;
18             System.out.println(
19                 "New balance on account "
20                 + account + ": 0");
21         }
22     }
23     return true;
24 }
25 } // end of class CheckingAccount
```

"CheckingAccount.java"

# Erläuterungen

- ▶ Die Objekt-Methode `withdraw` wird neu definiert, die Objekt-Methode `deposit` wird übernommen.
- ▶ Der Normalfall des Abhebens erfolgt (als Seiteneffekt) beim Testen der ersten `if`-Bedingung.
- ▶ Dazu wird die `withdraw`-Methode der Oberklasse aufgerufen.
- ▶ Scheitert das Abheben mangels Geldes, wird der Fehlbetrag vom Rücklagen-Konto abgehoben.
- ▶ Scheitert auch das, erfolgt keine Konto-Bewegung, dafür eine Fehlermeldung.
- ▶ Andernfalls sinkt der aktuelle Kontostand auf `0` und die Rücklage wird verringert.

# Ein Sparbuch

```
1 public class SavingsAccount extends BankAccount {
2     protected double interestRate;
3     // Konstruktor:
4     public SavingsAccount(int id,double init,double rate){
5         super(id, init);
6         interestRate = rate;
7     }
8     // zusaetzliche Objekt-Methode:
9     public void addInterest() {
10        balance = balance * (1 + interestRate);
11        System.out.println(
12            "Interest added to account: "+ account
13            + "\nNew balance:\t" + balance);
14    }
15 } // end of class SavingsAccount
```

"SavingsAccount.java"

- ▶ Die Klasse `SavingsAccount` erweitert die Klasse `BankAccount` um das zusätzliche Attribut `double interestRate` (Zinssatz) und eine Objekt-Methode, die die Zinsen gutschreibt.
- ▶ Alle sonstigen Attribute und Objekt-Methoden werden von der Oberklasse geerbt.
- ▶ Die Klasse `BonusSaverAccount` erhöht zusätzlich den Zinssatz, führt aber Strafkosten fürs Abheben ein.

## Ein Bonus-Sparbuch

```
1 public class BonusSaverAccount extends SavingsAccount {
2     private int penalty;
3     private double bonus;
4     // Konstruktor:
5     public BonusSaverAccount(int id, double init,
6                               double rate) {
7         super(id, init, rate);
8         penalty = 25;
9         bonus = 0.03;
10    }
11    // Modifizierung der Objekt-Methoden:
12    public boolean withdraw(double amount) {
13        boolean res;
14        if (res = super.withdraw(amount + penalty))
15            System.out.println(
16                "Penalty incurred:\t"+ penalty);
17        return res;
18    }
```

"BonusSaverAccount.java"

# Ein Bonus-Sparbuch

```
19     public void addInterest() {
20         balance = balance * (1 + interestRate + bonus);
21         System.out.println(
22             "Interest added to account: " + account
23             + "\nNew balance:\t" + balance);
24     }
25 } // end of class BonusSaverAccount
```

"BonusSaverAccount.java"

# Programmausgabe

Deposit into account 4321

Amount: 148.04

New balance: 5176.49

Deposit into account 6543

Amount: 41.52

New balance: 1517.37

Withdrawal from account 4321

Amount: 725.55

New balance: 4450.94

Withdrawal from account 6543

Amount: 145.38

New balance: 1371.989999999998

Penalty incurred: 25

Withdrawal from account 9876

Amount: 320.18

Sorry, insufficient funds...

Using overdraft...

Withdrawal from account 4321

Amount: 50.25

New balance: 4400.69

New balance on account 9876: 0

# 13 Polymorphie

## Problem:

- ▶ Unsere Datenstrukturen `List`, `Stack` und `Queue` können einzig und allein `int`-Werte aufnehmen.
- ▶ Wollen wir `String`-Objekte oder andere Arten von Zahlen ablegen, müssen wir die jeweilige Datenstruktur nochmal definieren.

## 13.1 Unterklassen-Polymorphie

Eine Funktion, die für mehrere Argumenttypen definiert ist, heißt auch **polymorph**.

### Idee:

Überall wo ein Objekt vom Typ `ClassA` verwendet wird, können wir auch ein Objekt einer Unterklasse von `ClassA` nutzen.

- ▶ Zuweisungen:

```
ClassA a;  
ClassB b = new ClassB();  
a = b; // weise Objekt einer Unterklasse zu
```

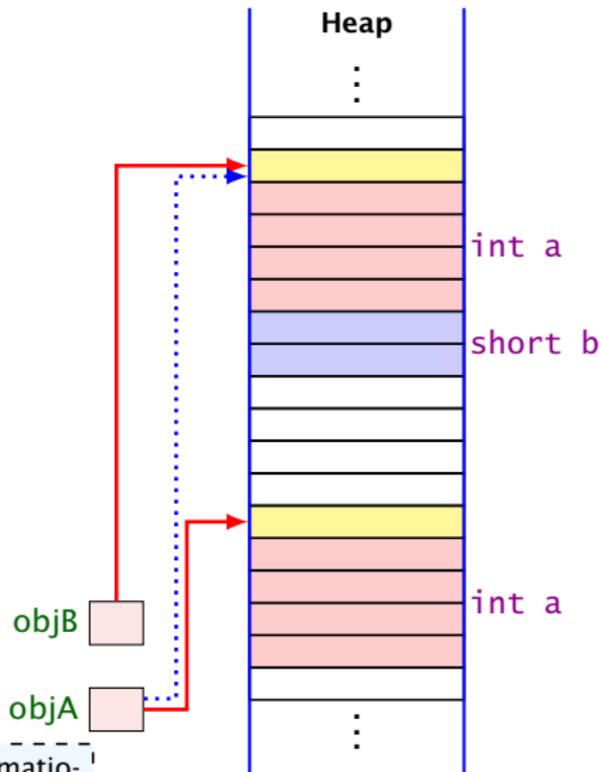
- ▶ Methodenaufrufe:

```
void meth(ClassA a) {};  
ClassB b = new ClassB();  
void bla() {  
    meth(b); // rufe meth mit Objekt von  
            // Unterklasse auf  
}
```

# Was passiert hier eigentlich?

```
public ClassA {  
    int a;  
}  
public ClassB extends ClassA {  
    short b;  
}  
public class Test {  
    public static void main() {  
        ClassA objA = new ClassA();  
        ClassB objB = new ClassB();  
        objA = objB;  
    }  
}
```

Ein Objekt vom Typ **ClassB** ist auch ein Objekt vom Typ **ClassA**.



Die gelben Felder enthalten Verweise auf Typinformationen, die aber für diesen Fall der Zuweisung nicht benötigt werden.

## Unrealistisches Beispiel

Der Finanzminister möchte jedem Konto 10€ gutschreiben, um die Wirtschaft anzukurbeln:

Das Array `arr` enthält Objekte unterschiedlicher Typen.

```
void boostEconomy(BankAccount[] arr) {  
    for (int i=0; i<arr.length; i++) {  
        arr[i].deposit(10);  
    }  
}
```

- ▶ Die Methode bekommt ein Array mit allen Konten übergeben.
- ▶ Die einzelnen Elemente des Arrays können `BankAccount`, `CheckingAccount`, `SavingsAccount`, oder `BonusSaverAccount` sein.
- ▶ Es wird jeweils die Methode `deposit` aufgerufen, die in der Klasse `BankAccount` implementiert ist.

Beachte, dass die Unterklassen von `BankAccount` die Methode `deposit` nicht überschreiben.

## Realistisches Beispiel

Die Mafia ist durch einen Hack in den Besitz einer großen Menge von Bankdaten gekommen. Diese gilt es auszunutzen:

```
void exploitHack(BankAccount[] arr) {  
    for (int i=0; i<arr.length; i++) {  
        arr[i].withdraw(10);  
    }  
}
```

- ▶ Hier wird die (spezielle) `withdraw`-Methode des jeweiligen Account-Typs aufgerufen.
- ▶ **Die kann der Compiler aber nicht kennen!!!**
- ▶ **Dynamische Methodenbindung!!!**

## Statischer vs. dynamischer Typ

Der **statische Typ** eines Ausdrucks ist der Typ, der sich gemäß den Regeln zur Auswertung von Ausdrücken ergibt.

Der **dynamische Typ** eines Referenzausdrucks **e** ist der Typ des wirklichen Objekts auf das **e zur Laufzeit** zeigt.

### Beispiel:

```
SavingsAccount s = new SavingsAccount(89,10,0.2);  
BankAccount b = s;
```

```
s //statischer Typ SavingsAccount  
b //statischer Typ BankAccount
```

```
s //dynamischer Typ SavingsAccount  
b //dynamischer Typ SavingsAccount
```

# Ermittlung der aufgerufenen Methode

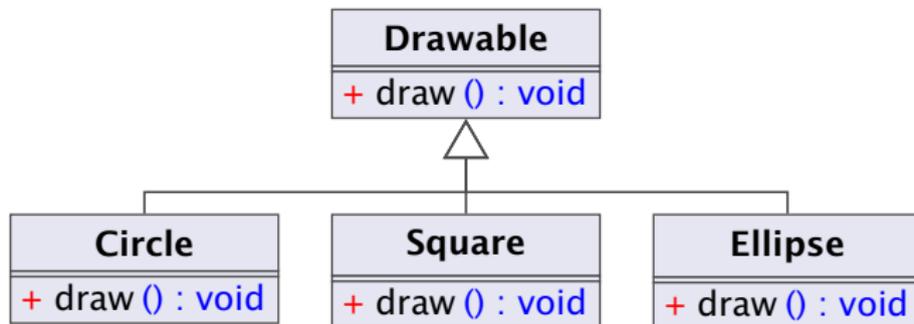
Das Verfahren beschreibt den Vorgang für **Objektmethoden**. Bei statischen Aufrufen, würde die in Schritt 2 gefundene Methode gewählt.

Betrachte einen Aufruf  $e_0.f(e_1, \dots, e_k)$

1. Bestimme die **statischen** Typen  $T_0, \dots, T_k$  der Ausdrücke  $e_0, \dots, e_k$ .
2. Suche in einer **Oberklasse** von  $T_0$  nach einer Methode mit Namen  $f$ , deren Liste von Argumenttypen bestmöglich zu der Liste  $T_1, \dots, T_k$  passt.  
Sei  $S$  **Signatur** dieser rein statisch gefundenen Methode  $f$ .
3. Der **dynamische** Typ  $D$  des Objekts, zu dem sich  $e_0$  auswertet, gehört zu einer Unterklasse von  $T_0$ .
4. Es wird die Methode  $f$  aufgerufen, die Signatur  $S$  hat, und die in der nächsten Oberklasse von  $D$  implementiert wird.

Das Ermitteln der Methode, die am besten **passt**, wurde schon im Kapitel über die Auswertung von Ausdrücken behandelt. Es kommen nur zusätzliche implizite Typecasts hinzu: Ein Cast von einer Unterklasse in eine zugehörige Oberklasse ist immer möglich, und wird vom Compiler als **impliziter Typecast** durchgeführt.

## Weiteres Beispiel



```
1 public class Figure {
2     Drawable[] arr; // contains basic shapes of figure
3     Figure(/* some parameters */) {
4         /* constructor initializes arr */
5     }
6     void draw() {
7         for (int i=0; i<arr.length; ++i) {
8             arr[i].draw();
9         }
10 } }
```

# Die Klasse `Object`

- ▶ Die Klasse `Object` ist eine gemeinsame Oberklasse für **alle** Klassen.
- ▶ Eine Klasse ohne angegebene Oberklasse ist eine direkte Unterklasse von `Object`.

# Die Klasse `Object`

Einige nützliche Methoden der Klasse `Object`:

- ▶ `String toString()` liefert Darstellung als `String`;
- ▶ `boolean equals(Object obj)` testet auf **Objekt-Identität** oder Referenz-Gleichheit:

```
1 public boolean equals(Object obj) {  
2     return this == obj;  
3 }
```

- ▶ `int hashCode()` liefert Nummer für das Objekt.
- ▶ ...viele weitere **geheimnisvolle Methoden**, die u.a. mit **paralleler Programmausführung** zu tun haben.

**Achtung:** `Object`-Methoden können aber in Unterklassen durch geeignete Methoden überschrieben werden.

# Beispiel

```
1 class PolyA {
2     public String toString() { return "Hello"; }
3 }
4 public class PolyTestA {
5     public static String addWorld(Object x) {
6         return x.toString() + " World!";
7     }
8     public static void main(String[] args) {
9         Object x = new PolyA();
10        System.out.println(addWorld(x));
11    }
12 }
```

liefert: "Hello World!"

- ▶ Die Klassen-Methode `addWorld()` kann auf jedes Objekt angewendet werden.
- ▶ Die Klasse `PolyA` ist eine Unterklasse von `Object`.
- ▶ Einer Variable der Klasse `ClassA` kann ein Objekt **jeder Unterklasse** von `ClassA` zugewiesen werden.
- ▶ Darum kann `x` das neue `PolyA`-Objekt aufnehmen.

Die Klasse `PolyA` enthält keinen explizit definierten Konstruktor.

Eine Klasse `ClassA`, die keinen anderen Konstruktor besitzt, erhält einen **Default-Konstruktor** `public ClassA()`.

# Beispiel

```
1 class PolyB {
2     public String greet() { return "Hello"; }
3 }
4 public class PolyTestB {
5     public static void main(String[] args) {
6         Object x = new PolyB();
7         System.out.println(x.greet()+" World!");
8     }
9 }
```

liefert: **Compilerfehler**

Method greet() not found in class java.lang.Object.  
System.out.print(x.greet() + " World!\n");

^

1 error

# Erklärung

- ▶ Die Variable `x` ist als `Object` deklariert.
- ▶ Der Compiler weiß nicht, ob der aktuelle Wert von `x` ein Objekt aus einer Unterklasse ist, in welcher die Objektmethode `greet()` definiert ist.
- ▶ Darum lehnt er dieses Programm ab.

# Methodenaufruf

Der Aufruf einer **statischen** Methode:

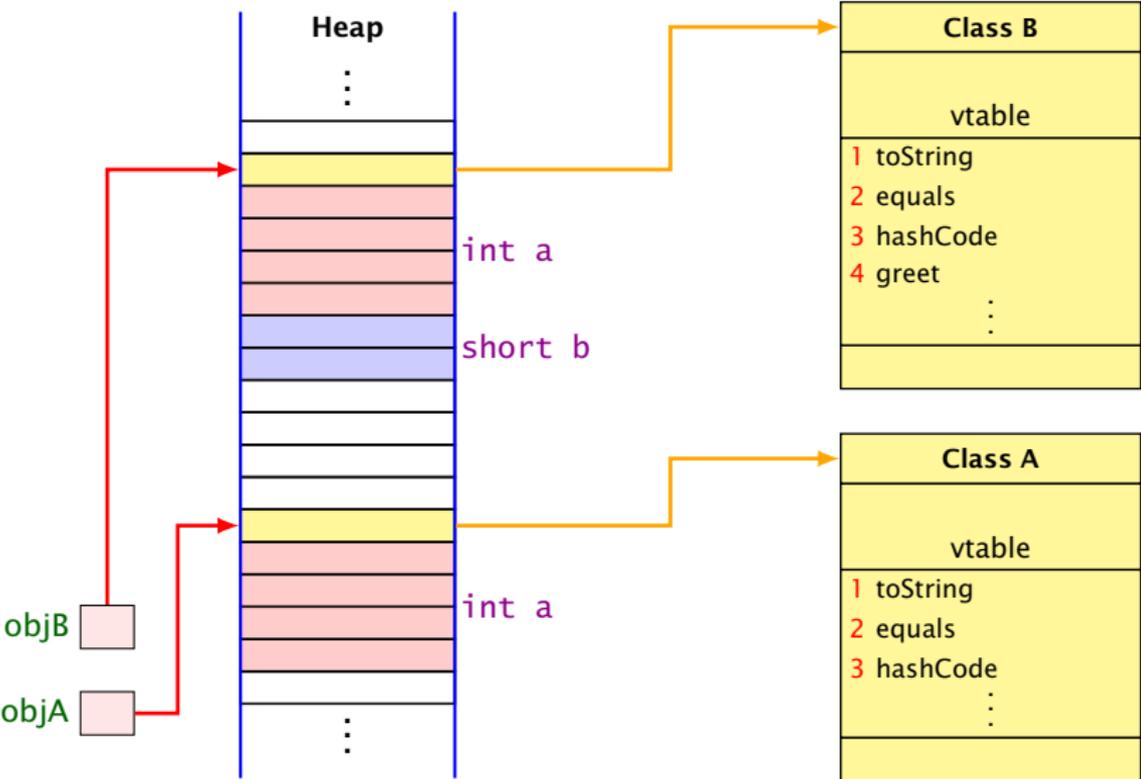
1. Aktuelle Parameter und Rücksprungadresse auf den Stack legen.
2. Zum Code der Funktion springen.

Aufruf einer Objektmethode:

1. Aktuelle Parameter (auch **this**) und Rücksprungadresse auf den Stack legen.
2. **Problem:** Die aufgerufene Funktion ist zur Compilezeit noch nicht bekannt; existiert vielleicht nicht einmal.

Die Funktion `addWorld()` im vorherigen Beispiel könnte schon existieren, bevor die Klasse `Poly` geschrieben wird. Wie kann dann `addWorld()` die richtige `toString()`-Funktion aufrufen?

# Methodenaufruf



## Methodenaufruf

Ein Aufruf `obj.equals()` wird wie folgt verarbeitet.

- Suche die `vtable` des Objekts auf das `obj` zeigt.
- Suche in dieser `vtable` nach dem Index von `equals()`.
- Springe an die dort gespeicherte Sprungadresse.

- ▶ Jede Klasse hat eine Tabelle (`vtable`) mit Methoden, die zu dieser Klasse gehören. Darin wird die Adresse des zugehörigen Codes gespeichert.
- ▶ Ein Aufruf einer Objektmethode (z.B. `equals`) sucht in dieser Tabelle nach der Sprungadresse.
- ▶ Beim **Überschreiben** einer Methode in einer Unterklasse wird dieser Eintrag auf die Sprungadresse der neuen Funktion geändert.
- ▶ **Dynamische Methodenbindung**

### Wichtig

Der Index der Funktionen innerhalb der (`vtable`) ist in jeder abgeleiteten Klasse gleich.

Dies ist nur eine (sehr naheliegende) Möglichkeit dynamische Methodenbindung zu realisieren. D.h. nicht, dass die **JVM** dies genau so umsetzt.

# Beispiel

```
1 class PolyB {
2     public String greet() { return "Hello"; }
3 }
4 public class PolyTestB {
5     public static void main(String[] args) {
6         Object x = new PolyB();
7         System.out.println(x.greet()+" World!");
8     }
9 }
```

liefert: **Compilerfehler**

Method greet() not found in class java.lang.Object.  
System.out.print(x.greet() + " World!\n");

^

1 error

Benutze einen expliziten **cast** in die entsprechende Unterklasse!

```
class PolyC {
    public String greet() { return "Hello "; }
}
public class PolyTestC {
    public void main(String[] args) {
        Object x = new PolyC();
        if (x instanceof PolyC)
            System.out.println(((PolyC) x).greet()+"World!");
        else
            System.out.print("Cast not possible!\n");
    }
}
```

Java vergisst die Zugehörigkeit zu **B** nicht vollständig. Bei einem Aufruf von Objektmethoden, werden evt. von **B** überschriebene Methoden aufgerufen.

- ▶ Eine Variable **x** einer Klasse **A** kann Objekte **b** aus sämtlichen Unterklassen **B** von **A** aufnehmen.
- ▶ Durch diese Zuweisung vergisst **Java** die Zugehörigkeit zu **B**, da **Java** alle Werte von **x** als Objekte der Klasse **A** behandelt.
- ▶ Mit dem Ausdruck **x instanceof B** können wir zur **Laufzeit** die Klassenzugehörigkeit von **x** testen;
- ▶ Sind wir uns sicher, dass **x** aus der Klasse **B** ist, können wir in diesen Typ **casten**.
- ▶ Ist der aktuelle Wert der Variablen **x** bei dem versuchten Cast tatsächlich ein Objekt (einer Unterklasse) der Klasse **B**, liefert der Ausdruck genau dieses Objekt zurück. Andernfalls wird eine **Exception** ausgelöst.

Umwandlungen zwischen primitiven Datentypen (widening/narrowing conversions) erzeugen keine Laufzeitfehler.



# 13 Polymorphie

## Problem:

- ▶ Unsere Datenstrukturen `List`, `Stack` und `Queue` können einzig und allein `int`-Werte aufnehmen.
- ▶ Wollen wir `String`-Objekte oder andere Arten von Zahlen ablegen, müssen wir die jeweilige Datenstruktur nochmal definieren.

## Beispiel — Listen

Wir definieren Liste für `Object` anstatt jeweils eine für `Rational`, `BankAccount`, etc.

```
1 public class List {
2     public Object info;
3     public List next;
4     public List(Object x, List l) {
5         info = x;
6         next = l;
7     }
8     public void insert(Object x) {
9         next = new List(x, next);
10    }
11    public void delete() {
12        if (next != null) next = next.next;
13    }
14 // continued...
```

## Beispiel — Listen

```
14     public String toString() {
15         String result = "[" + info;
16         for (List t = next; t != null; t = t.next)
17             result = result + ", " + t.info;
18         return result + "]";
19     }
20     ...
21 } // end of class List
```

- ▶ Die Implementierung funktioniert ganz analog zur Implementierung für `int`.
- ▶ Die `toString()`-Methode ruft implizit die (stets vorhandene) `toString()`-Methode der Listenelemente auf.

# Beispiel — Listen

## Achtung:

```
1 //...
2 Poly x = new Poly();
3 List list = new List(x);
4 x = list.info;
5 System.out.println(x);
6 //...
```

liefert einen **Compilerfehler**. Der Variablen `x` dürfen nur Unterklassen von `Poly` zugewiesen werden.

# Beispiel — Listen

## Stattdessen:

```
1 //...
2 Poly x = new Poly();
3 List list = new List(x);
4 x = (Poly) list.info;
5 System.out.println(x);
6 //...
```

Das ist hässlich!!! Geht das nicht besser???

## 13.2 Generische Klassen

### Idee:

- ▶ Java verfügt über generische Klassen...
- ▶ Anstatt das Attribut `info` als `Object` zu deklarieren, geben wir der Klasse einen `Typ-Parameter T` für `info` mit!
- ▶ Bei Anlegen eines Objekts der Klasse `List` bestimmen wir, welchen `Typ T` und damit `info` haben soll...

## Beispiel — Listen

```
1 public class List<T> {
2     public T info;
3     public List<T> next;
4     public List (T x, List<T> l) {
5         info = x;
6         next = l;
7     }
8     public void insert(T x) {
9         next = new List<T> (x, next);
10    }
11    public void delete() {
12        if (next != null) next = next.next;
13    }
14    //continued...
```

## Beispiel — Listen

```
15     public static void main (String[] args) {
16         List<Poly> list
17             = new List<Poly> (new Poly(), null);
18         System.out.println(list.info.greet());
19     }
20 } // end of class List
```

- ▶ Die Implementierung funktioniert ganz analog zur Implementierung für `Object`.
- ▶ Der Compiler weiß aber nun in `main`, dass `list` vom Typ `List` ist mit Typparameter `T = Poly`.
- ▶ Deshalb ist `list.info` vom Typ `Poly`.
- ▶ Folglich ruft `list.info.greet()` die entsprechende Methode der Klasse `Poly` auf.

# Bemerkungen

```
class List<T> {  
    public static T info;  
}
```

funktioniert nicht, da `info` kein Objekt-Attribut ist.

- ▶ Die Typ-Parameter der Klasse dürfen nur in den Typen von Objekt-Attributen und Objekt-Methoden verwendet werden!!!
- ▶ Jede Unterklasse einer parametrisierten Klasse muss mindestens die gleichen Parameter besitzen:

`A<S,T> extends B<T>` ist erlaubt.

`A<S> extends B<S,T>` ist **verboten**.

- ▶ `Poly` ist eine Unterklasse von `Object`; aber `List<Poly>` ist **keine** Unterklasse von `List<Object>`!!!

```
1 List<Poly> l;  
2 // some other code...  
3 Poly p = l.info;
```

## Hauptvorteil von Generics:

Der Compiler garantiert, dass in Zeile 3 `l.info` immer ein `Poly`-Objekt ist (**statische Typsicherheit**).

Dieses kann nicht garantiert werden, wenn für Unterklasse `ClassB` von Elternklasse `ClassA` gelten würde, dass `List<ClassB>` Unterklasse von `List<ClassA>` ist.

# Fallstricke

```
1 List<Animal> a;  
2 List<Dog> d = new List<Dog>(...);  
3  
4 a = d; // Compilerfehler  
5 a.insert(new Cat());  
6 myDog = d.info;
```

Der Code ist eindeutig fehlerhaft. Deshalb erlaubt der Compiler die Zuweisung in Zeile 4 nicht.

`List<Dog>` ist keine Unterklasse von `List<Animal>`.

# Fallstricke

Ähnliche Probleme ergeben sich bei der Verwendung von Arrays. Deshalb kann man keine Arrays von generischen Typen erstellen.

```
List<Dog>[] arrOfLists = new List<Dog>[100];
```

liefert: **Compilerfehler**

```
TestGenericArray.java:11: error: generic array creation
    List<Dog>[] arrOfLists = new List<Dog>[100];
                                ^
```

1 error

Wir werden später einen Workaround dafür kennenlernen.

# Generics/Type Erasure/Arrays

Parametrisierte Datentypen sind in Java über Type-Erasure implementiert.

Das heißt, dass für die JVM `List<Poly>` und `List<String>` gleich aussehen und es nur einen Typ `List<Object>` gibt.

Nur der Compiler unterscheidet zwischen `List<Poly>` und `List<String>` und stellt z.B. sicher, dass

```
1 List<Poly> l;  
2 // some other code...  
3 Poly p = l.info;
```

in Zeile 3 immer einen `Poly` zurückliefert.

## Beispiel – Arrays

Die Zuweisung in Zeile 4 ist erlaubt, da Arrays in **Java** kovariant sind.

Der Basistyp des Arrays ist (normalerweise) zur Laufzeit bekannt. Deshalb kann in Zeile 5 eine Typprüfung (zur Laufzeit) stattfinden.

```

1 Animal[] a;
2 Dog[] d = new Dog[100];
3
4 a = d;
5 a[0] = new Cat(); // Laufzeitfehler
6 myDog = d[0];

```

Die Tatsache, dass Zeile 5 zur Laufzeitfehlern führen kann, ist eine Schwäche des Typsystems in **Java**.

### Kovarianz/Kontravarianz/Invarianz

Seien **A** und **B** Typen und  $f$  eine Typrtransformation.

- ▶ Falls  $A \leq B \Rightarrow f(A) \leq f(B)$  heißt  $f$  **kovariant**.
- ▶ Falls  $A \leq B \Rightarrow f(A) \geq f(B)$  heißt  $f$  **kontravariant**.
- ▶ Falls  $A \leq B$  keine Beziehung zwischen  $f(A)$  und  $f(B)$  impliziert heißt  $f$  **invariant**.

Beispiele:

- ▶  $A \leq B \Rightarrow A[] \leq B[]$  (Arrays sind in **Java** kovariant)
- ▶  $A \leq B$  impliziert keine Beziehung zwischen  $List<A>$  und  $List<B>$
- ▶  $A \leq B \Rightarrow A.meth() \leq B.meth()$  (wenn man eine Methode überschreibt muß der Rückgabebetyp kovariant sein)

# Beispiel – Generische Listen

```
1 List<Animal> a;  
2 List<Dog> d = new List<Dog>(...);  
3  
4 a = d; // Compilerfehler  
5 a.insert(new Cat());  
6 myDog = d.info;
```

Zeile 4 erzeugt einen Compilerfehler, da generische Datentypen in **Java invariant** sind.

Zeile 5: hier kann **Java** aufgrund von **Type-Erasure** keinen Laufzeitfehler erzeugen. Für die JVM sehen **List<Cat>** und **List<Dog>** gleich aus, also kann man diese Zuweisung nicht verbieten; außerdem ist die Idee hinter generischen Listen, dass **statische** Typsicherheit garantiert wird, d.h. man möchte keinen Laufzeitfehler erzeugen.

Zeile 6: Eine Hauptvorteil von Generics in Java ist, dass diese Zuweisung nicht zu einem Laufzeitfehler führt; d.h., der Compiler stellt zur Compilezeit sicher, dass dieses funktioniert. Deshalb möchte man hier keinen Laufzeitfehler erzeugen.

# Beispiel – Wildcards

```
1 List<?> a;  
2 List<Dog> d = new List<Dog>(...);  
3  
4 a = d;  
5 a.insert(new Cat()); // Compilerfehler  
6 myDog = d.info;
```

`List<?>` steht für eine Liste mit einem beliebigen (aber unbekanntem) Datentyp. Man kann solch einer Liste beliebige Listen der Form `List<T>` zuweisen (für konkrete Werte von `T`). Demnach ist Zeile 4 erlaubt.

Zeile 5: Da der eigentliche Typ der Liste (bei Zugriff über `a`) nicht bekannt ist, können keine Methoden aufgerufen werden, die den parametrisierten Datentyp benutzen (aber z.B. `a.length()` könnte man aufrufen).

# Beispiel – Raw Types

Raw Types sollte man so weit wie möglich vermeiden!

```
1 List a;  
2 List<Dog> d = new List<Dog>(...);  
3  
4 a = d;  
5 a.insert(new Cat()); // Compilerwarnung  
6 myDog = d.info; // Laufzeitfehler
```

`List` ist ein **Raw Type**. Bei der Verwendung dieses Typs wird die Typprüfung des Compilers umgangen. Deshalb ist Zeile 4 erlaubt.

Zeile 5: Da `a` eine Liste mit einem unbekanntem Basistyp ist, ist diese Zuweisung gefährlich und kann nicht vom Compiler auf Typsicherheit geprüft werden.

In Zeile 6 bekommt man den Laufzeitfehler, der durch die Umgehung der Typprüfung entsteht. Wenn man keine Raw-Typen verwendet (und alles kompiliert) ist eine Zuweisung wie `myDog = d.info` **statisch** geprüft.

## Beispiel – Arrays

Die JVM kann in Zeile 6 (unten) keinen Laufzeitfehler erzeugen, da durch Type Erasure auf beiden Seiten der Zuweisung einfach nur eine Liste steht. Stattdessen gibt es einen Compilerfehler in Zeile 2.

```
1 Animal[] a;  
2 Dog[] d = new Dog[100];  
3  
4 a = d;  
5 a[0] = new Cat(); // Laufzeitfehler  
6 myDog = d[0];
```

```
1 Object[] a;  
2 List<Dog>[] d =  
3     new List<Dog>[100]; // Compilerfehler!  
4  
5 a = d;  
6 a[0] = myCatList; // Laufzeitfehler???  
7 myDog = d[0].info;
```

Man sollte bei gleichzeitiger Verwendung von Generics und Arrays in Java aufpassen; die Kovarianz der Arrays und die Type-Erasure sorgen dafür, dass die statische Typprüfung der Generics nicht mehr funktioniert.

# Bemerkungen

- ▶ Für einen Typ-Parameter `T` kann man auch eine Oberklasse (oder ein Interface) angeben, das `T` auf jeden Fall erfüllen soll...

```
1 class Drawable {
2     void draw() {}
3 }
4 public class DrawableList<E extends Drawable> {
5     E element;
6     DrawableList<E> next;
7     void drawAll() {
8         element.draw();
9         if (next == null) return;
10        else next.drawAll();
11    }
12 }
```

## 13.3 Wrapper-Klassen

### Problem

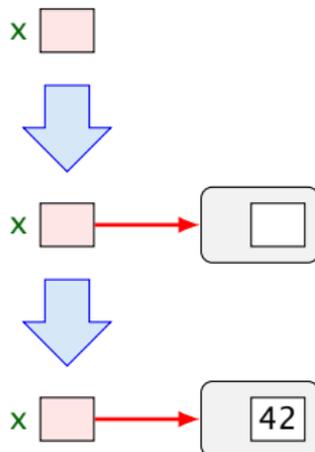
- ▶ Der Datentyp `String` ist eine Klasse;
- ▶ Felder sind Klassen; **aber**
- ▶ **Basistypen** wie `int`, `boolean`, `double` sind keine Klassen!  
(Eine Zahl ist eine Zahl und kein Verweis auf eine Zahl.)

### Ausweg

- ▶ Wickle die Werte eines Basis-Typs in ein Objekt ein!  
⇒ **Wrapper-Objekte** aus **Wrapper-Klassen**.

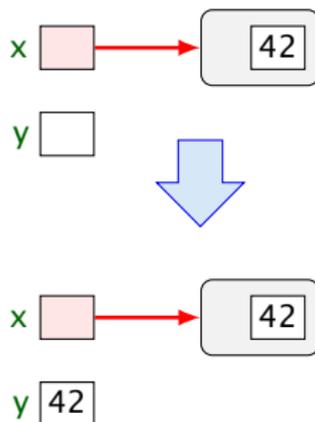
# Beispiel

Die Zuweisung `Integer x = new Integer(42);` bewirkt:



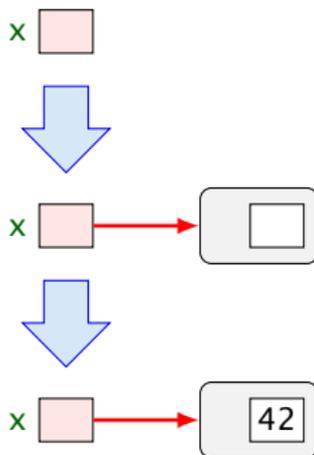
# Beispiel

Eingewickelte Werte können auch wieder ausgewickelt werden.  
Bei Zuweisung `int y = x;` erfolgt **automatische Konvertierung**:



## Beispiel

Umgekehrt wird bei Zuweisung eines `int`-Werts an eine `Integer`-Variable: `Integer x = 42;` automatisch der Konstruktor aufgerufen:



Man nennt diese Konvertierungen **boxing-** bzw. **unboxing-conversions**

# Nützliches

Gibt es erst einmal die Klasse `Integer`, lassen sich dort auch viele andere nützliche Dinge ablegen.

## Beispiele:

- ▶ `public static int MIN_VALUE = -2147483648;`  
liefert den kleinsten `int`-Wert;
- ▶ `public static int MAX_VALUE = 2147483647;` liefert den größten `int`-Wert;
- ▶ `public static int parseInt(String s) throws NumberFormatException;` berechnet aus dem `String`-Objekt `s` die dargestellte Zahl — sofern `s` einen `int`-Wert darstellt.

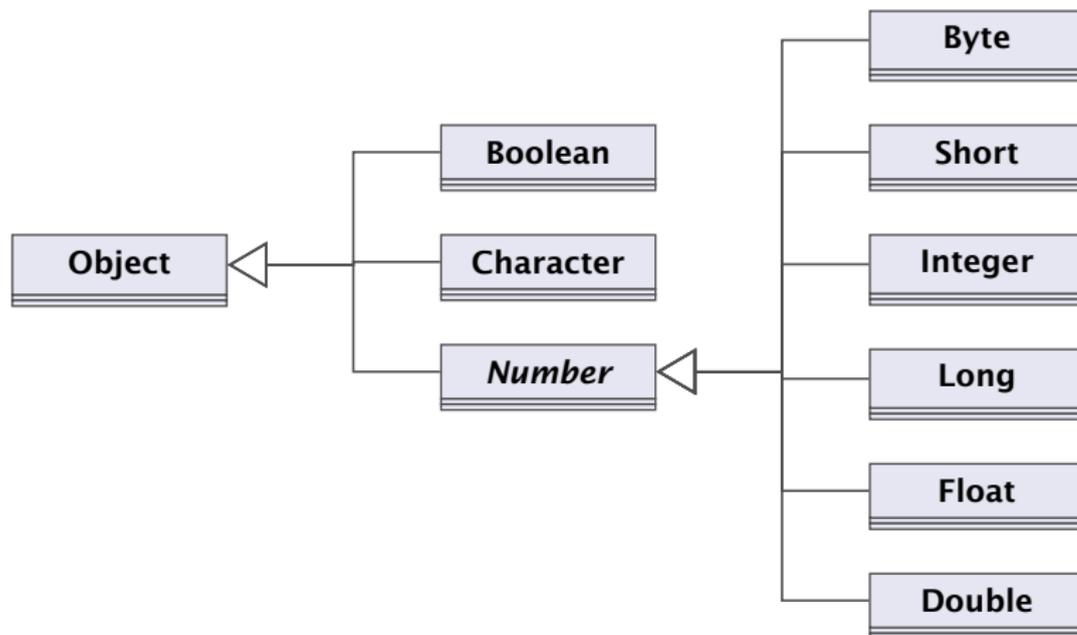
Andernfalls wird eine `Exception` geworfen.

# Bemerkungen

- ▶ Außer dem Konstruktor: `public Integer(int value);` gibt es u.a. `public Integer(String s) throws NumberFormatException;`
- ▶ Dieser Konstruktor liefert zu einem `String`-Objekt `s` ein `Integer`-Objekt, dessen Wert `s` darstellt.
- ▶ `public boolean equals(Object obj);` liefert `true` genau dann wenn `obj` den gleichen `int`-Wert enthält.

Ähnliche Wrapper-Klassen gibt es auch für die übrigen Basistypen...

# Wrapper-Klassen



Die Klasse **Number** ist hier in italics geschrieben, da es sich um eine **abstrakte Klasse** handelt.

## Bemerkungen

- ▶ Sämtliche Wrapper-Klassen für Typen `type` (außer `char`) verfügen über
  - ▶ Konstruktoren aus Basiswerten bzw. String-Objekten;
  - ▶ eine statische Methode `type parseType(String s)`;
  - ▶ eine Methode `boolean equals(Object obj)` die auf Gleichheit testet (auch `Character`).
- ▶ Bis auf `Boolean` verfügen alle über Konstanten `MIN_VALUE` und `MAX_VALUE`.
- ▶ `Character` enthält weitere Hilfsfunktionen, z.B. um Ziffern zu erkennen, Klein- in Großbuchstaben umzuwandeln. . .
- ▶ Die numerischen Wrapper-Klassen sind in der gemeinsamen Oberklasse `Number` zusammengefasst.
- ▶ Diese Klasse ist **↑abstrakt**, d.h. man kann keine `Number`-Objekte anlegen.

# Spezielles

- ▶ `Double` und `Float` enthalten zusätzlich die Konstanten

`NEGATIVE_INFINITY` = `-1.0/0`

`POSITIVE_INFINITY` = `+1.0/0`

`NaN` = `0.0/0`

- ▶ Zusätzlich gibt es die Tests

- ▶ `public static boolean isInfinite(double v);`

- ▶ `public static boolean isNaN(double v);`

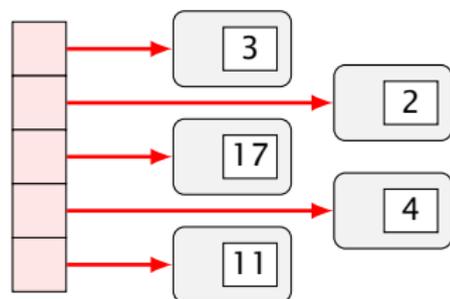
(analog für `float`)

- ▶ `public boolean isInfinite();`

- ▶ `public boolean isNaN();`

mittels derer man auf (Un)Endlichkeit der Werte testen kann.

# Integer vs. Int



Integer[]



int[]

- + **Integers** können in polymorphen Datenstrukturen hausen.
- Sie benötigen mehr als doppelt so viel Platz.
- Sie führen zu vielen kleinen (evt.) über den gesamten Speicher verteilten Objekten  
⇒ schlechteres Cache-Verhalten.

## 14 Abstrakte Klassen, finale Klassen, Interfaces

- ▶ Eine **abstrakte** Objekt-Methode ist eine Methode, für die keine Implementierung bereit gestellt wird.
- ▶ Eine Klasse, die abstrakte Objekt-Methoden enthält, heißt ebenfalls **abstrakt**.
- ▶ Für eine abstrakte Klasse können offenbar keine Objekte angelegt werden.
- ▶ Mit abstrakten Klassen können wir Unterklassen mit verschiedenen Implementierungen der gleichen Objekt-Methoden zusammenfassen.

## Auswertung von Ausdrücken

```
1 public abstract class Expression {
2     private int value;
3     private boolean evaluated = false;
4     public int getValue() {
5         if (!evaluated) {
6             value = evaluate();
7             evaluated = true;
8         }
9         return value;
10    }
11    abstract protected int evaluate();
12 } // end of class Expression
```

- ▶ Die Unterklassen von `Expression` repräsentieren die verschiedenen Arten von Ausdrücken.
- ▶ Allen Unterklassen gemeinsam ist eine Objekt-Methode `evaluate()` — immer mit einer anderen Implementierung.

# Abstrakte Methoden und Klassen

- ▶ Eine abstrakte Objekt-Methode wird durch das Schlüsselwort `abstract` gekennzeichnet.
- ▶ Eine Klasse, die eine abstrakte Methode enthält, muss selbst ebenfalls als `abstract` gekennzeichnet sein.
- ▶ Für die abstrakte Methode muss der vollständige Kopf angegeben werden — inklusive den Parametertypen und den (möglicherweise) geworfenen Exceptions.
- ▶ Eine abstrakte Klasse kann konkrete Methoden enthalten, hier: `int getValue()`.

# Beispiel

- ▶ Die Methode `evaluate()` soll den Ausdruck auswerten.
- ▶ Die Methode `getValue()` speichert das Ergebnis in dem Attribut `value` ab und vermerkt, dass der Ausdruck bereits ausgewertet wurde.

## Beispiel für einen Ausdruck:

```
1 public final class Const extends Expression {
2     private int n;
3     public Const(int x) { n = x; }
4     protected int evaluate() {
5         return n;
6     } // end of evaluate()
7 } // end of class Const
```

## Das Schlüsselwort `final`

- ▶ Der Ausdruck `Const` benötigt ein Argument. Dieses wird dem Konstruktor mitgegeben und in einer privaten Variable gespeichert.
- ▶ Die Klasse ist als `final` deklariert.
- ▶ Zu als `final` deklarierten Klassen dürfen keine Unterklassen deklariert werden!!!
- ▶ Aus Sicherheits- wie Effizienz-Gründen sollten so viele Klassen wie möglich als `final` deklariert werden. . .
- ▶ Statt ganzer Klassen können auch einzelne Variablen oder Methoden als `final` deklariert werden.
- ▶ Finale Members dürfen nicht in Unterklassen umdefiniert werden.
- ▶ Finale Variablen dürfen zusätzlich nur initialisiert, aber nicht modifiziert werden  $\Rightarrow$  **Konstanten**.

## Andere Ausdrücke

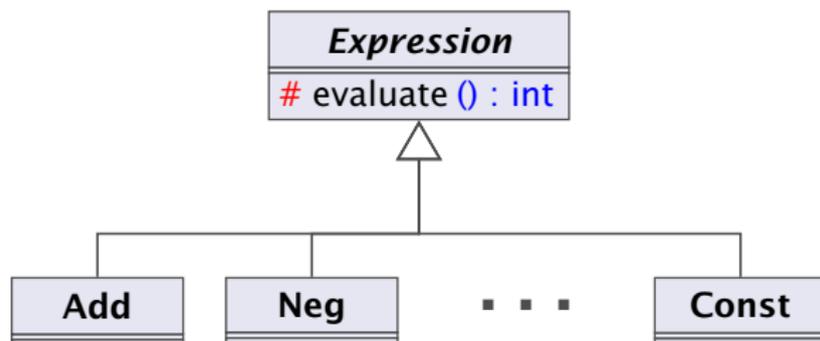
```
1 public final class Add extends Expression {
2     private Expression left, right;
3     public Add(Expression l, Expression r) {
4         left = l; right = r;
5     }
6     protected int evaluate() {
7         return left.getValue() + right.getValue();
8     } // end of evaluate()
9 } // end of class Add
10 public final class Neg extends Expression {
11     private Expression arg;
12     public Neg(Expression a) { arg = a; }
13     protected int evaluate() { return -arg.getValue(); }
14 } // end of class Neg
```

# main()

```
1 public static void main(String[] args) {
2     Expression e = new Add (
3         new Neg (new Const(8)),
4         new Const(16));
5     System.out.println(e.getValue());
6 }
```

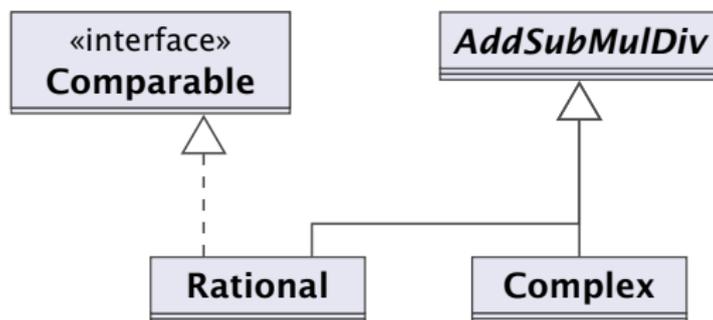
- ▶ Die Methode `getValue()` ruft eine Methode `evaluate()` sukzessive für jeden Teilausdruck von `e` auf.
- ▶ Welche konkrete Implementierung dieser Methode dabei jeweils gewählt wird, hängt von der konkreten Klasse des jeweiligen Teilausdrucks ab, d.h. entscheidet sich erst zur Laufzeit.
- ▶ Das nennt man auch **dynamische Bindung**.

# Klassenhierarchie



**Leider** (zum Glück?) lässt sich nicht die ganze Welt hierarchisch organisieren. . .

# Beispiel



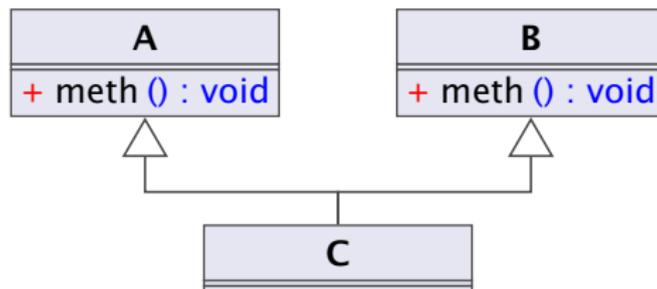
**AddSubMulDiv** = Objekte mit Operationen `add()`, `sub()`, `mul()`, und `div()`

**Comparable** = Objekte, die eine `compareTo()`-Operation besitzen.

# Mehrfachvererbung

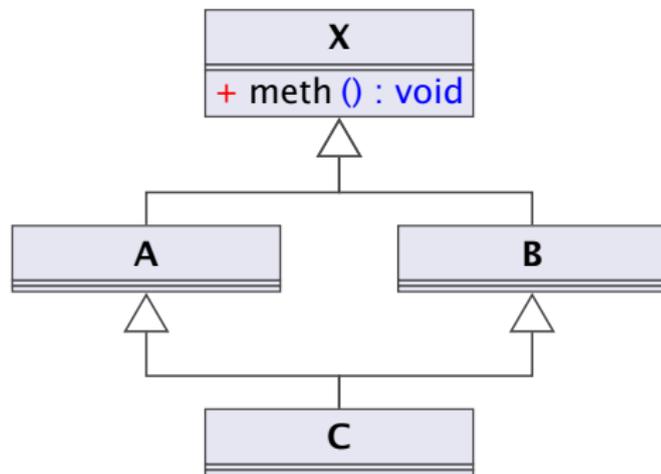
Mehrere direkte Oberklassen einer Klasse führen zu konzeptuellen Problemen:

- ▶ Auf welche Klasse bezieht sich **super**?
- ▶ Welche Objekt-Methode **meth()** ist gemeint, wenn mehrere Oberklassen **meth()** implementieren?



# Mehrfachvererbung

- ▶ Welche Objekt-Methode `meth()` ist gemeint, wenn mehrere Oberklassen `meth()` implementieren? Insbesondere



**deadly diamond of death**

# Interfaces

- ▶ Kein Problem entsteht, wenn die Objekt-Methode `meth()` in allen Oberklassen abstrakt ist,
- ▶ oder zumindest nur in maximal einer Oberklasse eine Implementierung besitzt.

Ein **Interface** kann aufgefasst werden als eine abstrakte Klasse, wobei:

1. alle Objekt-Methoden abstrakt sind;
2. es keine Klassen-Methoden gibt;
3. alle Variablen **Konstanten** sind.

Seit Java 8 kann ein Interface Default-Implementierungen enthalten, d.h. Punkt 1 gilt nicht mehr.

Seit Java 8 kann ein Interface statische Funktionen enthalten, d.h. Punkt 2 gilt nicht mehr.

# Beispiel

```
1 public interface Comparable {  
2     int compareTo(Object x);  
3 }
```

- ▶ **Object** ist die gemeinsame Oberklasse aller Klassen.
- ▶ Methoden in Interfaces sind automatisch Objektmethoden und **public**.
- ▶ Es muss eine **Obermenge** der in Implementierungen geworfenen Exceptions angegeben werden.
- ▶ Evt. vorkommende Konstanten sind automatisch **public static**.

# Beispiel

```
1 public class Rational extends AddSubMulDiv
2         implements Comparable {
3 private int zaehler, nenner;
4 public int compareTo(Object cmp) {
5     Rational fraction = (Rational) cmp;
6     long left = (long)zaehler * (long)fraction.nenner;
7     long right = (long)nenner * (long)fraction.zaehler;
8     return left == right ? 0:
9           left < right ? -1:
10          1;
11 } // end of compareTo
12 ...
13 } // end of class Rational
```

# Erläuterungen

- ▶ `class A extends B implements B1, B2, ..., Bk { ... }`  
gibt an, dass die Klasse `A` als Oberklasse `B` hat und zusätzlich die Interfaces `B1, B2, ..., Bk` unterstützt, d.h. passende Objektmethoden zur Verfügung stellt.
- ▶ `Java` gestattet maximal eine Oberklasse, aber beliebig viele implementierte Interfaces.
- ▶ Die Konstanten des Interface können in implementierenden Klassen **direkt** benutzt werden.
- ▶ Interfaces können als Typen für formale Parameter, Variablen oder Rückgabewerte benutzt werden.
- ▶ Darin abgelegte Objekte sind dann stets aus einer implementierenden Klasse.
- ▶ Expliziter Cast in eine solche Klasse ist möglich (und leider auch oft nötig).

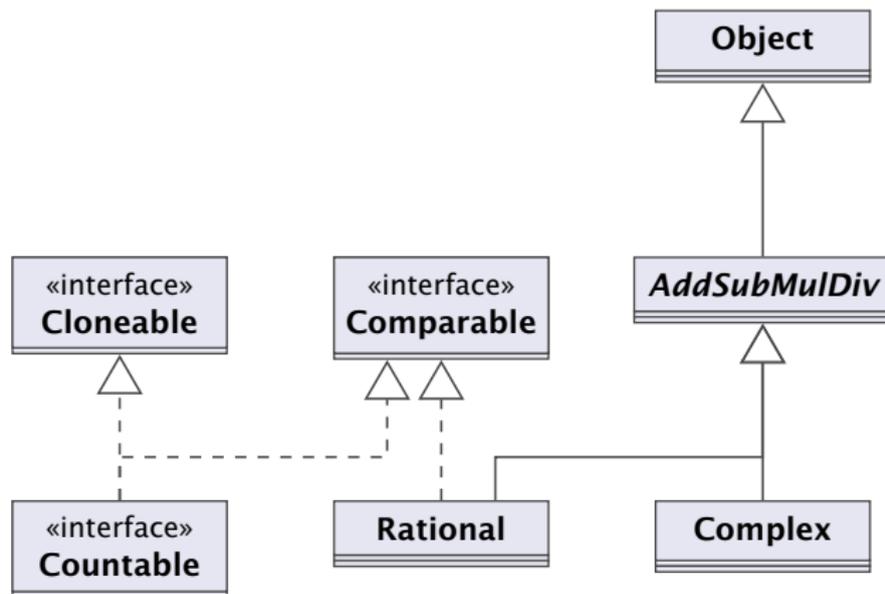
# Erläuterungen

- ▶ Interfaces können andere Interfaces erweitern oder gar mehrere andere Interfaces zusammenfassen.
- ▶ Erweiternde Interfaces können Konstanten umdefinieren...
- ▶ Kommt eine Konstante gleichen Namens `const` in verschiedenen implementierten Interfaces `A` und `B` vor, kann man sie durch `A.const` und `B.const` unterscheiden.

## Beispiel:

```
1 public interface Countable extends Comparable, Cloneable {  
2     Countable next();  
3     Countable prev();  
4     int number();  
5 }
```

- ▶ Das Interface `Countable` umfasst die (beide vordefinierten) Interfaces `Comparable` und `Cloneable`.
- ▶ Das vordefinierte Interface `Cloneable` verlangt eine Objektmethode `public Object clone()` die eine Kopie des Objekts anlegt.
- ▶ Eine Klasse, die `Countable` implementiert, muss über die Objektmethoden `compareTo()`, `clone()`, `next()`, `prev()` und `number()` verfügen.



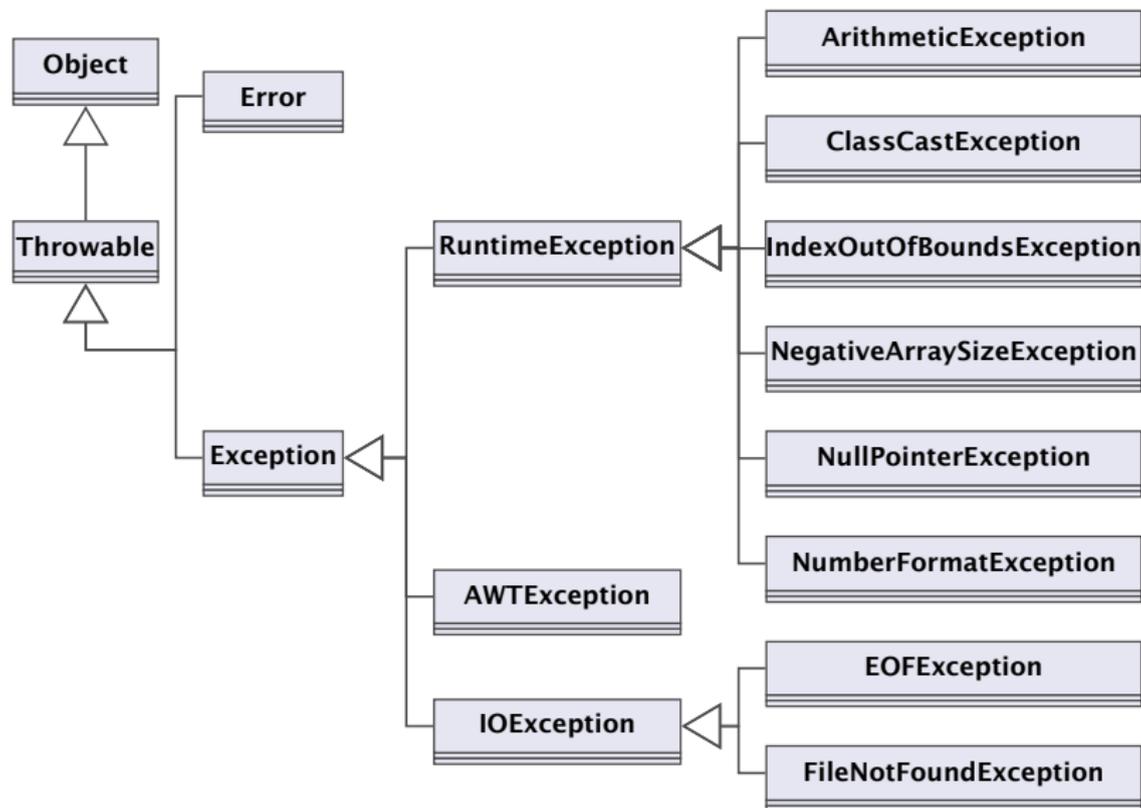
# 15 Fehlerobjekte: Werfen, Fangen, Behandeln

- ▶ Tritt während der Programmausführung ein Fehler auf, wird die normale Programmausführung abgebrochen und ein Fehlerobjekt erzeugt (**geworfen**).
- ▶ Die Klasse **Throwable** fasst alle Arten von Fehlern zusammen.
- ▶ Ein Fehlerobjekt kann **gefangen** und geeignet **behandelt** werden.

## Trennung von

- ▶ normalem Programm-Ablauf (der effizient und übersichtlich sein sollte); und
- ▶ Behandlung von Sonderfällen (wie illegalen Eingaben, falscher Benutzung, Sicherheitsattacken, . . .)

# Fehlerklassen



# Fehlerklassen

Die direkten Unterklassen von `Throwable` sind:

- ▶ `Error` — für fatale Fehler, die zur Beendigung des gesamten Programms führen, und
- ▶ `Exception` — für bewältigbare Fehler oder Ausnahmen.

## unchecked exception

Ausnahmen der Klasse `Error` und `RuntimeException` müssen nicht im Methodenkopf deklariert werden.

## checked exception

Die anderen Ausnahmen, die in einer Methode auftreten können und dort nicht selbst abgefangen werden, müssen **explizit** im Kopf der Methode aufgelistet werden!!!

# Fehlerklassen

- ▶ Die Unterklasse `RuntimeException` der Klasse `Exception` fasst die bei normaler Programmausführung evt. auftretenden Ausnahmen zusammen.
- ▶ Eine `RuntimeException` kann jederzeit auftreten. . .
- ▶ Sie kann, muss aber nicht abgefangen werden.

## Arten der Fehlerbehandlung:

- ▶ Ignorieren;
- ▶ Abfangen und Behandeln dort, wo sie entstehen;
- ▶ Abfangen und Behandeln an einer anderen Stelle.

# Fehlerbehandlung

Tritt ein Fehler auf und wird nicht behandelt, bricht die Programmausführung ab.

## Beispiel:

```
1 public class Zero {
2     public static void main(String[] args) {
3         int x = 10;
4         int y = 0;
5         System.out.println(x/y);
6     } // end of main()
7 } // end of class Zero
```

# Fehlermeldung

Das Programm bricht wegen Division durch `(int)0` ab und liefert die Fehler-Meldung:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero  
at Zero.main(Compiled Code)
```

Die Fehlermeldung besteht aus drei Teilen:

1. der `Thread`, in dem der Fehler auftrat;
2. `System.err.println(toString());`; d.h. dem **Namen** der Fehlerklasse, gefolgt von einer Fehlermeldung, die die Objekt-Methode `getMessage()` liefert, hier: „/ by zero“.
3. `printStackTrace(System.err);`; d.h. der **Funktion**, in der der Fehler auftrat, genauer: der Angabe sämtlicher Aufrufe im **Call-Stack**.

# Was passiert wenn man Exceptions nicht fängt?



Ariane 5, 4 Juli 1996

- ▶ Beim Jungfernflug der Ariane 5 trat im Programm für die Steuerkontrolle eine Exception bei der Konvertierung einer Gleitkommazahl in einen Integer auf.
- ▶ Da diese nicht behandelt/gefangen wurde, wurde das Programm abgebrochen.

# Fehlerbehandlung

Soll die Programm-Ausführung nicht beendet werden, muss der Fehler abgefangen werden.

## Beispiel: NumberFormatException

```
1 public class Adding extends MiniJava {
2     public static void main(String[] args) {
3         int x = getInt("1. Zahl:\t");
4         int y = getInt("2. Zahl:\t");
5         write("Summe:\t\t" + (x+y));
6     } // end of main()
7     public static int getInt(String str) {
8 //continued...
```

- ▶ Das Programm liest zwei `int`-Werte ein und addiert sie.
- ▶ Bei der Eingabe können möglicherweise Fehler auftreten:
  - ▶ ... weil die Eingabe keine syntaktisch korrekte Zahl ist;
  - ▶ ... weil sonstige unvorhersehbare Ereignisse eintreffen.
- ▶ Die **Behandlung** dieser Fehler ist in der Funktion `getInt()` verborgen...

# Fehlerbehandlung

```
1      String s;  
2      while (true) {  
3          try {  
4              s = readString(str);  
5              return Integer.parseInt(s);  
6          } catch (NumberFormatException e) {  
7              System.out.println(  
8                  "Falsche Eingabe! ...");  
9          } catch (NullPointerException e) {  
10             System.out.println(  
11                 "Eingabeproblem: Ende ...");  
12             System.exit(0);  
13         }  
14     } // end of while  
15 } // end of getInt()  
16 } // end of class Adding
```

# Beispielablauf

```
> java Adding  
1. Zahl: abc  
Falsche Eingabe! ...  
1. Zahl: 0.3  
Falsche Eingabe! ...  
1. Zahl: 17  
2. Zahl: 25  
Summe: 42
```

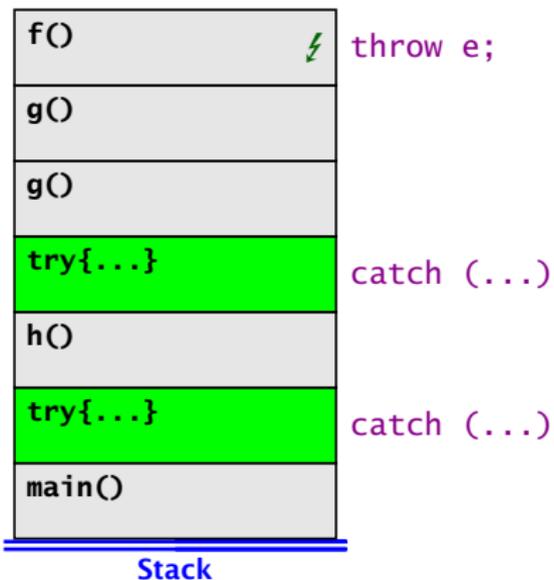
# Ausnahmebehandlung

- ▶ Ein **Exception-Handler** besteht aus einem **try**-Block `try{ss}`, in dem der Fehler möglicherweise auftritt; gefolgt von einer oder mehreren **catch**-Regeln.
- ▶ Wird bei der Ausführung der Statement-Folge `ss` kein Fehlerobjekt erzeugt, fährt die Programm-Ausführung direkt hinter dem Handler fort.
- ▶ Wird eine Exception ausgelöst, durchsucht der Handler mithilfe des geworfenen Fehler-Objekts sequentiell die **catch**-Regeln.

# Ausnahmebehandlung

- ▶ Jede `catch`-Regel ist von der Form: `catch(Exc e) {...}`  
wobei `Exc` eine Klasse von Fehlern angibt und `e` ein formaler Parameter ist, an den das Fehler-Objekt gebunden wird.
- ▶ Eine Regel ist **anwendbar**, sofern das Fehlerobjekt aus (einer Unterklasse) von `Exc` stammt.
- ▶ Die erste `catch`-Regel, die anwendbar ist, wird angewendet. Dann wird der Handler verlassen.
- ▶ Ist keine `catch`-Regel anwendbar, wird der Fehler propagiert.

# Was passiert auf dem Stack?



# Ausnahmebehandlung

- ▶ Auslösen eines Fehlers verlässt abrupt die aktuelle Berechnung.
- ▶ Damit das Programm trotz Auftretens des Fehlers in einem geordneten Zustand bleibt, ist oft Aufräumarbeit erforderlich – z.B. das Schließen von I/O-Strömen.
- ▶ Dazu dient `finally { ss }` nach einem `try`-Statement.

# Ausnahmebehandlung

- ▶ Die Folge `ss` von Statements wird **auf jeden Fall** ausgeführt.
- ▶ Wird kein Fehler im `try`-Block geworfen, wird sie im Anschluss an den `try`-Block ausgeführt.
- ▶ Wird ein Fehler geworfen und mit einer `catch`-Regel behandelt, wird sie nach dem Block der `catch`-Regel ausgeführt.
- ▶ Wird der Fehler von keiner `catch`-Regel behandelt, wird `ss` ausgeführt, und dann der Fehler weitergereicht.

# Beispiel NullPointerException

```
1 public class Kill {
2     public static void kill() {
3         Object x = null; x.hashCode ();
4     }
5     public static void main(String[] args) {
6         try { kill();
7         } catch (ClassCastException b) {
8             System.out.println("Falsche Klasse!!!");
9         } finally {
10            System.out.println("Das wars...");
11        }
12    } // end of main()
13 } // end of class Kill
```

## Resultat:

```
> java Kill
```

```
Leider nix gefangen ...
```

```
Exception in thread "main" java.lang.NullPointerException  
    at Kill.kill(Compiled Code)  
    at Kill.main(Compiled Code)
```

# Selbstdefinierte Fehler

Exceptions können auch

- ▶ selbst definiert und
- ▶ selbst geworfen werden.

## Beispiel:

```
1 class Killed extends Exception {
2     Killed() {}
3     Killed(String s) {super(s);}
4 } // end of class Killed
5 public class Kill2 {
6     public static void kill() throws Killed {
7         throw new Killed();
8     }
9 // continued...
```

# Beispiel

```
10 public static void main(String[] args) {
11     try {
12         kill();
13     } catch (RuntimeException r) {
14         System.out.println("RunTimeException "+ r);
15     } catch (Killed b) {
16         System.out.println("Killed It!");
17         System.out.println(b);
18         System.out.println(b.getMessage());
19     }
20 } // end of main
21 } // end of class Kill
```

## Selbstdefinierte Fehler

- ▶ Ein selbstdefinierter Fehler sollte als Unterklasse von `Exception` deklariert werden!
- ▶ Die Klasse `Exception` verfügt über die Konstruktoren  
`public Exception();`  
`public Exception(String str);`  
(`str` ist die evt. auszugebende Fehlermeldung).
- ▶ `throw exc` löst den Fehler `exc` aus — sofern sich der Ausdruck `exc` zu einem Objekt einer Unterklasse von `Throwable` auswertet.
- ▶ Weil `Killed` keine Unterklasse von `RuntimeException` ist, wird die geworfene Exception erst von der zweiten `catch`-Regel gefangen.
- ▶ **Ausgabe:**  
Killed It!  
Killed  
Null

- ▶ Fehler in **Java** sind Objekte und können vom Programm selbst behandelt werden.
- ▶ **try ... catch ... finally** gestattet, die Fehlerbehandlung deutlich von der normalen Programmausführung zu trennen.
- ▶ Werden spezielle neue Fehler/Ausnahmen benötigt, können diese in einer Vererbungshierarchie organisiert werden.

# Checked Exceptions

Checked Exceptions sind eine Spezialität von Java. In anderen Programmiersprachen wie C#, C++ findet man diese nicht.

## Vorteile:

- ▶ Man sieht sofort welche Exceptions eine Funktion werfen kann.
- ▶ Der Compiler „überprüft“ ob man die Exceptions behandelt.

## Mögliche Nachteile:

- ▶ Skalierbarkeit: Wenn man viele API-Funktionen aufruft muß man eventuell sehr viele Exceptions angeben, die man werfen könnte.
- ▶ Exceptions können Details der Implementierung verraten (e.g. `SQLException`). D.h. es könnte schwierig sein die Implementierung einer API-Funktion später zu ändern.

## Exceptions – Hinweise

- ▶ Exceptions fangen ist teuer. D.h. man sollte Exceptions nur für außergewöhnliche Situationen nutzen. **Nicht zur Ablaufkontrolle.**
- ▶ Fehler sollten dort behandelt werden, wo sie auftreten.
- ▶ Es ist besser **spezifischere** Fehler zu fangen als **allgemeine** — z.B. mit `catch(Exception e) {}`
- ▶ Was passiert, wenn **catch**- und **finally**-Regeln selbst wieder Fehler werfen?
- ▶ Eine API-Funktion sollte Exceptions werfen, die der Abstraktion der Funktion angemessen sind. (Also `getUser()` sollte z.B. `UserNotFoundException` werfen, anstatt `SQLException`.)
- ▶ Programmierfehler (z.B. falsche Verwendung einer Klasse) sollten durch `RuntimeException` signalisiert werden.
- ▶ Checked Exceptions sollte man nur verwenden, wenn der Aufrufer sie sinnvoll behandeln kann.

## Exception Safety

Es gibt verschiedene Garantien, die eine Objektmethode bzgl. Exceptions erfüllen kann:

### **Basic Garantie**

Nach einer Exception erfüllt das Objekt alle seine Invarianten.  
Keine Leaks.

Häufig nicht sehr hilfreich. Das Objekt kann sich beliebig verändert haben; das Gute ist, dass wir das Objekt noch löschen können.

### **Strong Garantie**

Im Fall einer Exception hat sich das Objekt nicht verändert.  
(Transaktionales Verhalten; entweder funktioniert alles, oder keine Änderung wird durchgeführt).

### **No-throw Garantie**

Die Funktion wirft keine Exceptions.

# Threads – Einführung

- ▶ Die Ausführung eines **Java**-Programms besteht in Wahrheit nicht aus einem, sondern **mehreren** parallel laufenden **Threads**.
- ▶ Ein Thread ist ein sequentieller Ausführungsstrang.
- ▶ Der Aufruf eines Programms startet einen Thread **main**, der die Methode **main()** des Programms ausführt.
- ▶ Ein weiterer Thread, den das Laufzeitsystem parallel startet, ist die **Garbage Collection**.
- ▶ Die Garbage Collection soll mittlerweile nicht mehr erreichbare Objekte beseitigen und den von ihnen belegten Speicherplatz der weiteren Programmausführung zur Verfügung stellen.

# Threads – Anwendungen

- ▶ Mehrere Threads sind auch nützlich, um
  - ▶ ...mehrere Eingabe-Quellen zu überwachen (z.B. Maus, Tastatur) ↑**Graphik**;
  - ▶ ...während der Blockierung einer Aufgabe etwas anderes Sinnvolles erledigen zu können;
  - ▶ ...die Rechenkraft mehrerer Prozessoren auszunutzen.
- ▶ Neue Threads können deshalb vom Programm selbst erzeugt und gestartet werden.
- ▶ Dazu stellt **Java** die Klasse **Thread** und das Interface **Runnable** bereit.

# Version A

```
1 public class MyThread extends Thread {
2     public void hello(String s) {
3         System.out.println(s);
4     }
5     public void run() {
6         hello("I'm running ...");
7     } // end of run()
8     public static void main(String[] args) {
9         MyThread t = new MyThread();
10        t.start();
11        System.out.println("Thread has been started ...");
12    } // end of main()
13 } // end of class MyThread
```

# Erläuterungen

- ▶ Neue Threads werden für Objekte aus (Unter-) Klassen der Klasse `Thread` angelegt.
- ▶ Jede Unterklasse von `Thread` sollte die Objekt-Methode `public void run();` implementieren.
- ▶ Ist `t` ein `Thread`-Objekt, dann bewirkt der Aufruf `t.start();` das folgende:
  1. ein neuer Thread wird initialisiert;
  2. die (parallele) Ausführung der Objektmethode `run()` für `t` wird angestoßen;
  3. die eigene Programmausführung wird hinter dem Aufruf fortgesetzt.

## Version B

```
1 public class MyRunnable implements Runnable {
2     public void hello(String s) {
3         System.out.println(s);
4     }
5     public void run() {
6         hello("I'm running ...");
7     } // end of run()
8     public static void main(String[] args) {
9         Thread t = new Thread(new MyRunnable());
10        t.start();
11        System.out.println("Thread has been started ...");
12    } // end of main()
13 } // end of class MyRunnable
```

# Erläuterungen

- ▶ Auch das Interface `Runnable` verlangt die Implementierung einer Objektmethode `public void run()`;
- ▶ `public Thread(Runnable obj)`; legt für ein `Runnable`-Objekt `obj` ein `Thread`-Objekt an.
- ▶ Ist `t` das `Thread`-Objekt für das `Runnable obj`, dann bewirkt der Aufruf `t.start()`; das folgende:
  1. ein neuer Thread wird initialisiert;
  2. die (parallele) Ausführung der Objekt-Methode `run()` für `obj` wird angestoßen;
  3. die eigene Programm-Ausführung wird hinter dem Aufruf fortgesetzt.

# Mögliche Ausführungen

## Entweder

```
Thread has been started ...  
I'm running ...
```

## oder

```
I'm running ...  
Thread has been started ...
```

# Scheduling

- ▶ Ein Thread kann nur eine Operation ausführen, wenn ihm ein Prozessor (CPU) zur Ausführung zugeteilt worden ist.
- ▶ Im Allgemeinen gibt es mehr Threads als CPUs.
- ▶ Der **Scheduler** verwaltet die verfügbaren CPUs und teilt sie den Threads zu.
- ▶ Bei verschiedenen Programmläufen kann diese Zuteilung verschieden aussehen!!!
- ▶ Es gibt verschiedene Strategien, nach denen sich Scheduler richten können (↑**Betriebssysteme**). Z.B.:
  - ▶ Zeitscheibenverfahren
  - ▶ Naives Verfahren

## Strategie

- ▶ Ein Thread erhält eine CPU nur für eine bestimmte Zeitspanne (**Time Slice**), in der er rechnen darf.
- ▶ Danach wird er unterbrochen. Dann darf ein anderer.

# Beispiel: Zeitscheibenverfahren

Thread-3: 

Thread-2: 

Thread-1: 

Schedule: 



# Erläuterungen – Zeitscheibenverfahren

- ▶ Ein Zeitscheiben-Scheduler versucht, jeden Thread **fair** zu behandeln, d.h. ab und zu Rechenzeit zuzuordnen — egal, welche Threads sonst noch Rechenzeit beanspruchen.
- ▶ Kein Thread hat jedoch Anspruch auf einen bestimmten Time-Slice.
- ▶ Für den Programmierer sieht es so aus, als ob sämtliche Threads „echt“ parallel ausgeführt werden, d.h. jeder über eine eigene CPU verfügt.

## Strategie

- ▶ Erhält ein Thread eine CPU, darf er laufen, so lange er will...
- ▶ Gibt er die CPU wieder frei, darf ein anderer Thread arbeiten...

# Beispiel – Naives Scheduling

Thread-3: 

Thread-2: 

Thread-1: 

Schedule: 

# Beispiel

```
1 public class Start extends Thread {
2     public void run() {
3         System.out.println("I'm running...");
4         while (true);
5     }
6     public static void main(String[] args) {
7         (new Start()).start();
8         (new Start()).start();
9         (new Start()).start();
10        System.out.println("main is running...");
11        while (true);
12    }
13 } // end of class Start
```

# Beispiel

**Ausgabe** (bei naivem Scheduling)

```
main is running...
```

Weil **main** nie fertig wird, erhalten die anderen Threads keine Chance, sie **verhungern**.

Faires Scheduling mit Zeitscheibenverfahren würde z.B. liefern:

```
I'm running...  
main is running...  
I'm running...  
I'm running...
```

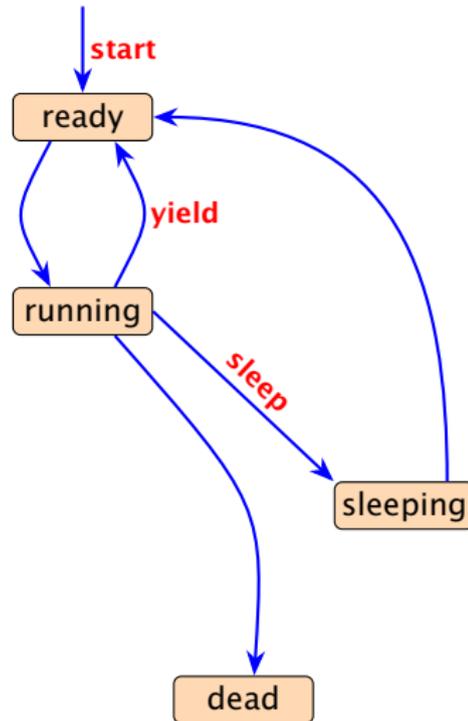
# Implementierung in Java

- ▶ **Java** legt nicht fest, wie intelligent der Scheduler ist.
- ▶ Die aktuelle Implementierung unterstützt **fares** Scheduling.
- ▶ Programme sollten aber für jeden Scheduler das **gleiche Verhalten** zeigen. Das heißt:
- ▶ ... Threads, die aktuell nichts sinnvolles zu tun haben, z.B. weil sie auf Verstreichen der Zeit oder besseres Wetter warten, sollten stets ihre CPU anderen Threads zur Verfügung stellen.
- ▶ ... Selbst wenn Threads etwas Vernünftiges tun, sollten sie ab und zu andere Threads laufen lassen.

**Achtung:** Threadwechsel ist teuer!!!

- ▶ Dazu verfügt jeder Thread über einen **Zustand**, der bei der Vergabe von Rechenzeit berücksichtigt wird.

# Threadzustände

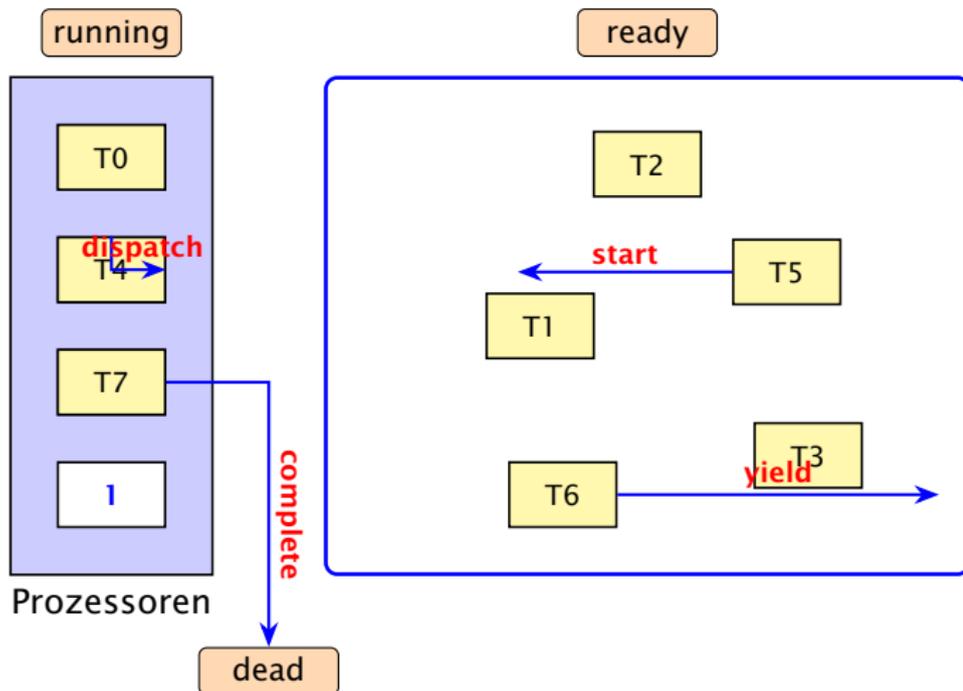


# Threadzustände

- ▶ `public void start();` legt einen neuen Thread an, setzt den Zustand auf `ready` und übergibt damit den Thread dem Scheduler zur Ausführung.
- ▶ Der Scheduler ordnet den Threads, die im Zustand `ready` sind, Prozessoren zu („dispatching“). Aktuell laufende Threads haben den Zustand `running`.
- ▶ `public static void yield();` setzt den aktuellen Zustand zurück auf `ready` und unterbricht damit die aktuelle Programmausführung. Andere ausführbare Threads erhalten die Gelegenheit zur Ausführung.
- ▶ `public static void sleep(int msec) throws InterruptedException;` legt den aktuellen Thread für `msec` Millisekunden schlafen, indem der Thread in den Zustand `sleeping` wechselt.

# Threadzustände

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



## 16.1 Futures

- ▶ Die Berechnung eines Zwischenergebnisses kann lange dauern.
- ▶ Während dieser Berechnung kann möglicherweise etwas anderes Sinnvolles berechnet werden.

### Idee:

- ▶ Berechne das Zwischenergebnis in einem eigenen Thread.
- ▶ Greife auf den Wert erst zu, wenn sich der Thread beendet hat.

## 16.1 Futures

Eine **Future** startet die Berechnung eines Werts, auf den später zugegriffen wird. Das generische Interface

```
public interface Callable<T> {  
    T call () throws Exception;  
}
```

aus `java.util.concurrent` beschreibt Klassen, für deren Objekte ein Wert vom Typ `T` berechnet werden kann.

```
1 public class Future<T> implements Runnable {  
2     private T value = null;  
3     private Exception exc = null;  
4     private Callable<T> work;  
5     private Thread task;  
6 // continued...
```

# Implementierung

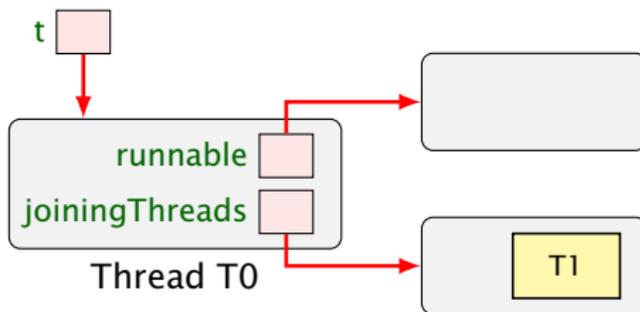
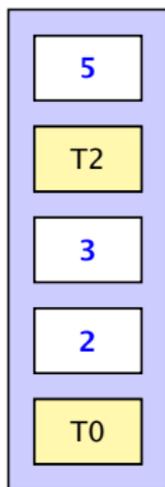
```
7     public Future(Callable<T> w) {
8         work = w;
9         task = new Thread (this);
10        task.start();
11    }
12    public void run() {
13        try {value = work.call();}
14        catch (Exception e) { exc = e;}
15    }
16    public T get() throws Exception {
17        task.join();
18        if (exc != null) throw exc;
19        return value;
20    }
```

# Erläuterungen

- ▶ Der Konstruktor erhält ein `Callable`-Objekt.
- ▶ Die Methode `run()` ruft für dieses Objekt die Methode `call()` auf und speichert deren Ergebnis in dem Attribut `value` — bzw. eine geworfene Exception in `exc` ab.
- ▶ Der Konstruktor legt ein Thread-Objekt für die Future an und startet diesen Thread, der dann `run()` ausführt.
- ▶ Die Methode `get()` wartet auf Beendigung des Threads. Dazu verwendet sie die Objekt-Methode `public final void join() throws InterruptedException` der Klasse `Thread`...
- ▶ Dann liefert `get()` den berechneten Wert zurück — falls keine Exception geworfen wurde. Andernfalls wird die Exception `exc` erneut geworfen.

# Die Join-Operation

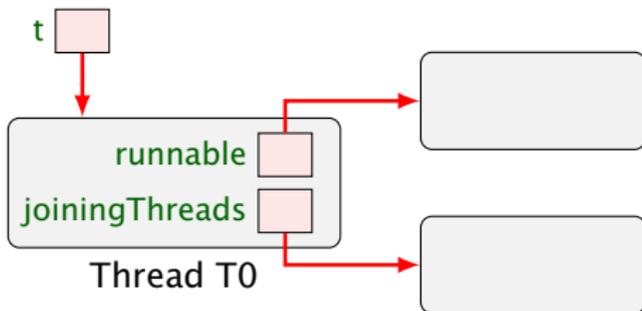
Animation ist nur in der Vorlesungsversion der Folien vorhanden.



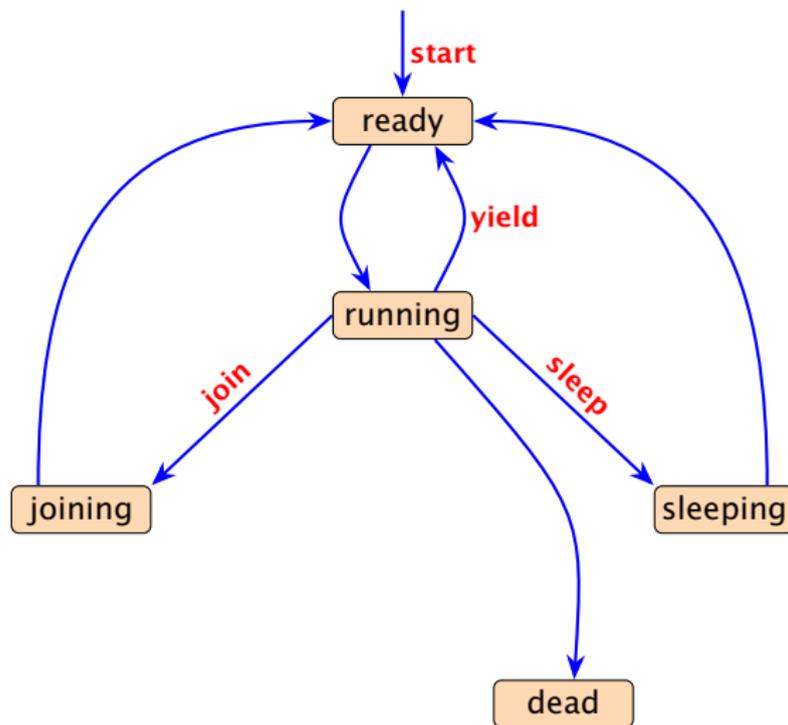
- ▶ Für jedes Threadobjekt `t` gibt es eine Schlange `ThreadQueue joiningThreads`.
- ▶ Threads, die auf Beendigung des Threads `t` warten, werden in diese Schlange eingefügt.
- ▶ Dabei gehen sie konzeptuell in einen Zustand `joining` über und werden aus der Menge der ausführbaren Threads entfernt.
- ▶ Beendet sich ein Thread, werden alle Threads, die auf ihn warteten, wieder aktiviert. . .

# Die Join-Operation

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



# Threadzustände



## Weiteres Beispiel

```
1 public class Join implements Runnable {
2     private static int count = 0;
3     private int n = count++;
4     private static Thread[] task = new Thread[3];
5     public void run() {
6         try {
7             if (n>0) {
8                 task[n-1].join();
9                 System.out.println("Thread-"+n+
10                    " joined Thread-"+(n-1));
11             }
12         } catch (InterruptedException e) {
13             System.err.println(e.toString());
14         }
15     } // continued...
```

## Weiteres Beispiel

```
16     public static void main(String[] args) {
17         for(int i=0; i<3; i++)
18             task[i] = new Thread(new Join());
19         for(int i=0; i<3; i++)
20             task[i].start();
21     }
22 } // end of class Join
```

liefert:

```
> java Join
Thread-1 joined Thread-0
Thread-2 joined Thread-1
```

# Variation

```
1 public class CW implements Runnable {
2     private static int count = 0;
3     private int n = count++;
4     private static Thread[] task = new Thread[3];
5     public void run() {
6         try { task[(n+1) % 3].join(); }
7         catch (InterruptedException e) {
8             System.err.println(e.toString());
9         }
10    }
11    public static void main(String[] args) {
12        for(int i=0; i<3; i++)
13            task[i] = new Thread(new CW());
14        for(int i=0; i<3; i++) task[i].start();
15    }
16 } // end of class CW
```

# Variation

- ▶ Das Programm terminiert möglicherweise nicht...
- ▶ `task[0]` wartet auf `task[1]`,  
`task[1]` wartet auf `task[2]`,  
`task[2]` wartet auf `task[0]`

ist möglich...

`t.join` angewendet auf einen nicht gestarteten Thread, hat keine Auswirkungen. Deshalb kann das Programm je nach Scheduling auch durchlaufen...

# Deadlock

- ▶ Jeder Thread geht in einen Wartezustand (hier: **joining**) über und wartet auf einen anderen Thread.
- ▶ Dieses Phänomen heißt auch **Circular Wait** oder **Deadlock** oder Verklemmung — eine unangenehme Situation, die man in seinen Programmen tunlichst vermeiden sollte.

Die Vermeidung von Deadlocks ist ein sehr schwieriges Problem.

## 16.2 Monitore

- ▶ Damit Threads sinnvoll miteinander kooperieren können, müssen sie miteinander Daten austauschen.
- ▶ Zugriff mehrerer Threads auf eine gemeinsame Variable ist problematisch, weil nicht feststeht, in welcher Reihenfolge die Threads auf die Variable zugreifen.
- ▶ Ein Hilfsmittel, um geordnete Zugriffe zu garantieren, sind **Monitore**.

## Beispiel — Erhöhen einer Variablen

```
1 public class Inc implements Runnable {
2     private static int x = 0;
3     private static void pause(int t) {
4         try {
5             Thread.sleep(((int) (Math.random()*t*1000)));
6         } catch (InterruptedException e) {
7             System.err.println(e.toString());
8         }
9     }
10    public void run() {
11        String s = Thread.currentThread().getName();
12        pause(3); int y = x;
13        System.out.println(s+ " read "+y);
14        pause(4); x = y+1;
15        System.out.println(s+ " wrote "+(y+1));
16    }
17 // continued...
```

## Beispiel

```
18     public static void main(String[] args) {
19         (new Thread(new Inc())).start();
20         pause(2);
21         (new Thread(new Inc())).start();
22         pause(2);
23         (new Thread(new Inc())).start();
24     }
25 } // end of class Inc
```

- ▶ `public static Thread currentThread();` liefert (eine Referenz auf) das ausführende Thread-Objekt.
- ▶ `public final String getName();` liefert den Namen des Thread-Objekts.
- ▶ Das Programm legt für 3 Objekte der Klasse `Inc` Threads an.
- ▶ Die Methode `run()` inkrementiert die Klassen-Variable `x`.

# Beispiel

## Mögliche Ausführung

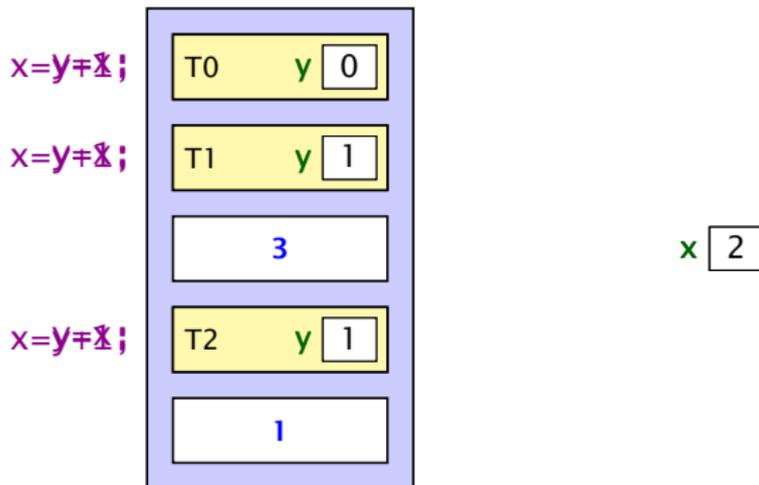
```
> java Inc  
Thread-0 read 0  
Thread-0 wrote 1  
Thread-1 read 1  
Thread-2 read 1  
Thread-1 wrote 2  
Thread-2 wrote 2
```

x wurde dreimal erhöht, hat aber am Ende den Wert 2!!!

Beachte, dass das gleiche auch passieren könnte, wenn wir in der Methode `run()`, direkt `x=x+1` schreiben würden.

# Erklärung

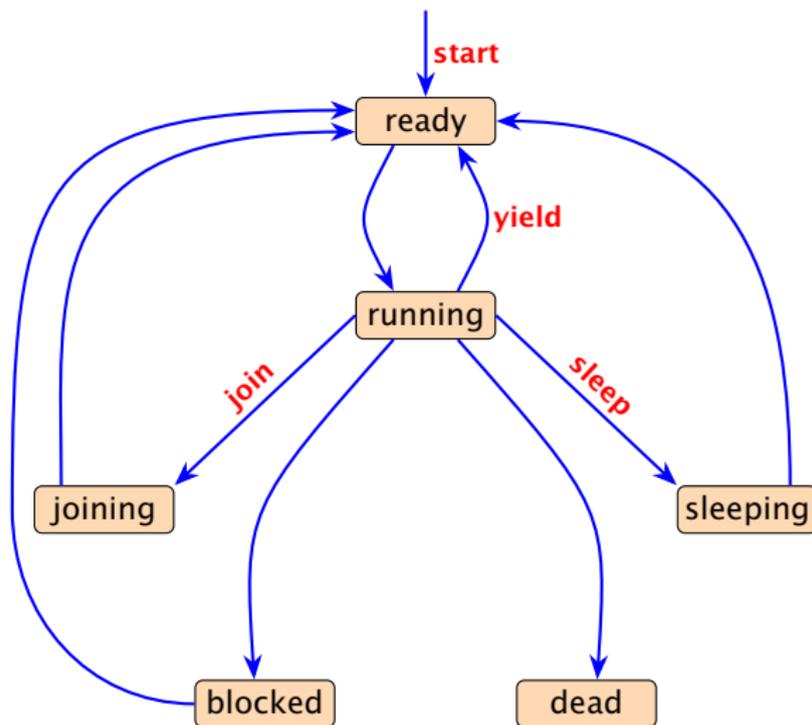
Animation ist nur in der Vorlesungsversion der Folien vorhanden.



## Monitore — Idee

- ▶ Inkrementieren der Variable `x` sollte ein **atomarer Schritt** sein, d.h. nicht von parallel laufenden Threads unterbrochen werden können.
- ▶ Mithilfe des Schlüsselworts **synchronized** kennzeichnen wir Objekt-Methoden einer Klasse `L` als ununterbrechbar.
- ▶ Für jedes Objekt `obj` der Klasse `L` kann zu jedem Zeitpunkt nur ein Aufruf `obj.synchMeth(...)` einer **synchronized**-Methode `synchMeth()` ausgeführt werden. Die Ausführung einer solchen Methode nennt man **kritischen Abschnitt** („critical section“) für die gemeinsame Resource `obj`.
- ▶ Wollen mehrere Threads gleichzeitig in ihren kritischen Abschnitt für das Objekt `obj` eintreten, werden alle bis auf einen **blockiert**.

# Threadzustände



# Locks

Dieses ist nur ein mentales Modell dafür was bei Eintritt/Austritt aus einer synchronized Methode passiert. Man kann diese Attribute nicht direkt zugreifen, und die Implementierung könnten in der Realität ganz anders aussehen.

- ▶ Ein Objekt `obj` mit `synchronized`-Methoden verfügt über:
  1. boolesches Flag `boolean locked`; sowie
  2. eine Warteschlange `ThreadQueue blockedThreads`.
- ▶ Vor Betreten seines kritischen Abschnitts führt ein Thread (**implizit**) die atomare Operation `obj.lock()` aus:

```
private void lock() {  
    if (!locked) locked = true; // betrete krit. Abschnitt  
    else { // Lock bereits vergeben  
        Thread t = Thread.currentThread();  
        blockedThreads.enqueue(t);  
        t.state = blocked; // blockiere  
    }  
} // end of lock()
```

# Locks

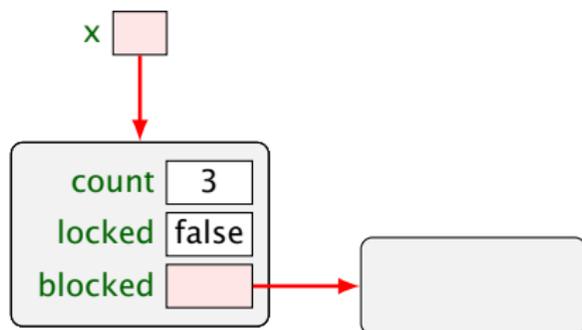
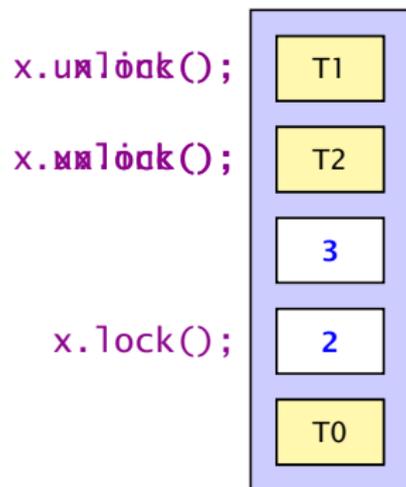
- ▶ Verlässt ein Thread seinen kritischen Abschnitt für `obj` (evt. auch mittels einer Exception), führt er (implizit) die atomare Operation `obj.unlock()` aus:

```
1 private void unlock() {
2     if (blockedThreads.empty())
3         locked = false; // Lock frei geben
4     else { // Lock weiterreichen
5         Thread t = blockedThreads.dequeue();
6         t.state = ready;
7     }
8 } // end of unlock()
```

- ▶ Dieses Konzept nennt man **Monitor**.

# Beispiel

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



# Implementierung

```
class Count {
    private int count = 0;
    public synchronized void inc() {
        String s = Thread.currentThread().getName();
        int y=count; System.out.println(s+" read "+y);
        count=y+1; System.out.println(s+" wrote "+count);
    }
} // end of class Count
public class IncSync implements Runnable {
    private static Count x = new Count();
    public void run() { x.inc(); }
    public static void main(String[] args) {
        (new Thread(new IncSync())).start();
        (new Thread(new IncSync())).start();
        (new Thread(new IncSync())).start();
    }
} // end of class IncSync
```

# Beispiel

**liefert:**

```
> java IncSync
Thread-0 read 0
Thread-0 wrote 1
Thread-1 read 1
Thread-1 wrote 2
Thread-2 read 2
Thread-2 wrote 3
```

## Achtung:

- ▶ Die Operation `lock()` erfolgt nur, wenn der Thread nicht bereits **vorher** das Lock des Objekts erworben hat.
- ▶ Ein Thread, der das Lock eines Objekts `obj` besitzt, kann **weitere** Methoden für `obj` aufrufen, ohne sich selbst zu blockieren.

## Locks — im Detail

Diese Art von Locks heißen auch **Reentrant Locks** oder **Rekursive Locks**. Sie haben teilweise einen schlechten Ruf.

- ▶ Um das zu garantieren, legt ein Thread für jedes Objekt `obj`, dessen Lock er nicht besitzt, aber erwerben will, einen neuen Zähler an:

```
int countLock[obj] = 0;
```

- ▶ Bei jedem Aufruf einer **synchronized**-Methode `m(...)` für `obj` wird der Zähler inkrementiert, für jedes Verlassen (auch mittels Exceptions) dekrementiert:

```
if (0 == countLock[obj]++) lock();  
obj.m(...)  
if (--countLock[obj] == 0) unlock();
```

- ▶ `lock()` und `unlock()` werden nur ausgeführt, wenn  
`(countLock[obj] == 0)`

## Beispiel: synchronized

```
1 public class StopThread extends Thread {
2     private static boolean stopRequested;
3
4     public void run() {
5         int i = 0;
6         while (!stopRequested)
7             i++;
8     }
9     public static void main(String[] args) {
10        Thread background = new StopThread();
11        background.start();
12        try {Thread.sleep(5);}
13        catch (InterruptedException e) {};
14        stopRequested = true;
15    }
16 }
```

## Beispiel: synchronized

Das Programm terminiert (eventuell) nicht.

Die Änderung der Variablen `stopRequested` wird dem anderen Prozess nie mitgeteilt.

Wenn man auf die Variable durch `synchronized`-Methoden zugreift, ist sichergestellt, dass die Kommunikation korrekt durchgeführt wird.

## Beispiel: synchronized

```
1 public class StopThreadCorrect extends Thread {
2     private static boolean stopRequested;
3     private static synchronized void setStop() {
4         stopRequested = true;
5     }
6     private static synchronized boolean getStop() {
7         return stopRequested;
8     }
9     public void run() {
10        int i = 0;
11        while (!getStop()) i++;
12    }
13    public static void main(String[] args) {
14        Thread background = new StopThreadCorrect();
15        background.start();
16        try {Thread.sleep(5);}
17        catch (InterruptedException e) {};
18        setStop();
19    } }
```

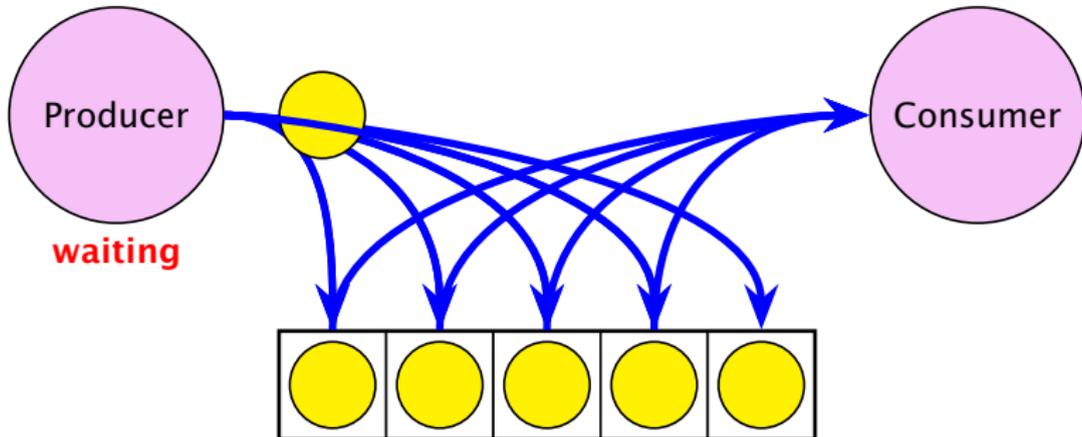
## 16.3 Semaphore

### Aufgabe:

- ▶ Zwei Threads möchten mehrere/viele Daten-Objekte austauschen.
- ▶ Der eine Thread erzeugt die Objekte einer Klasse **Data** (**Producer**).
- ▶ Der andere konsumiert sie (**Consumer**).
- ▶ Zur Übergabe dient ein Puffer, der eine feste Zahl **N** von **Data**-Objekten aufnehmen kann.

# Beispiel

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



# Consumer/Producer

## 1.Idee

- ▶ Wir definieren eine Klasse `Buffer`, die (im wesentlichen) aus einem Feld der richtigen Größe, sowie zwei Verweisen `int first`, `last` zum Einfügen und Entfernen verfügt:

```
1 class Data {}
2 public class Buffer {
3     private int cap, free, first, last;
4     private Data[] a;
5     public Buffer(int n) {
6         free = cap = n; first = last = 0;
7         a = new Data[n];
8     }
9 // continued...
```

- ▶ Einfügen und Entnehmen sollen synchrone Operationen sein...

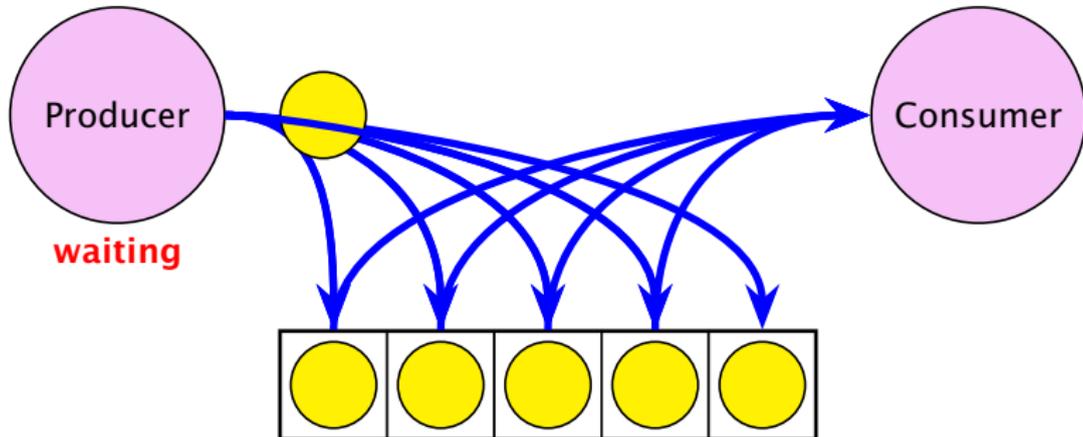
## Probleme

- ▶ Was macht der Consumer, wenn der Producer mit der Produktion nicht nachkommt, d.h. der Puffer leer ist?
- ▶ Was macht der Producer, wenn der Consumer mit der Weiterverarbeitung nicht nach kommt, d.h. der Puffer voll ist?

Lösungsvorschlag: **Warten...**

# Beispiel

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



# Umsetzung

- ▶ Jedes Objekt (mit `synchronized`-Methoden) verfügt über eine weitere Schlange `ThreadQueue` `waitingThreads` am Objekt wartender Threads sowie die Objekt-Methoden:

```
public final void wait() throws InterruptedException;  
public final void notify();  
public final void notifyAll();
```

- ▶ Diese Methoden dürfen nur für Objekte aufgerufen werden, über deren Lock der Thread verfügt!!!

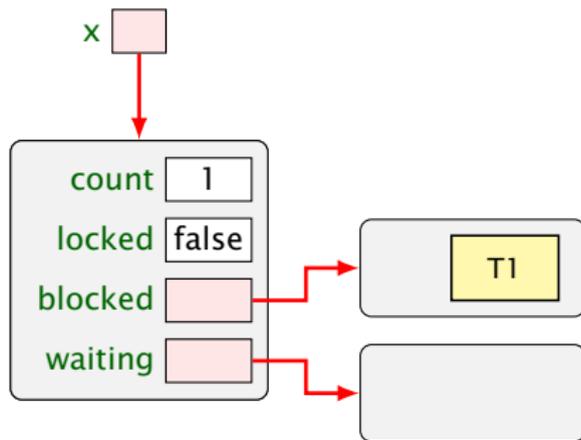
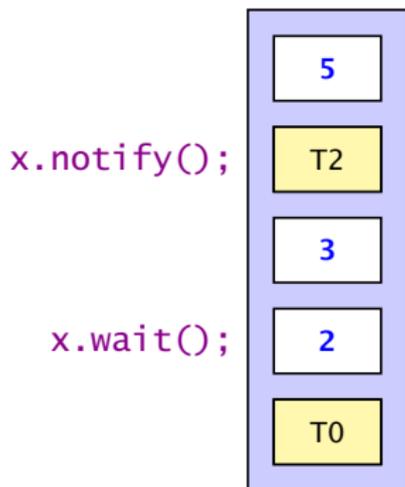
# Umsetzung

- ▶ Ausführen von `wait()`; setzt den Zustand des Threads auf `waiting`, reiht ihn in eine geeignete Warteschlange ein, und gibt das aktuelle Lock frei:

```
public void wait() throws InterruptedException {  
    Thread t = Thread.currentThread();  
    t.state = waiting;  
    waitingThreads.enqueue(t);  
    unlock();  
}
```

# Beispiel

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



## Umsetzung

- ▶ Ausführen von `notify()`; weckt den ersten Thread in der Warteschlange auf, d.h. versetzt ihn in den Zustand `blocked` und fügt ihn in `blockedThreads` ein:

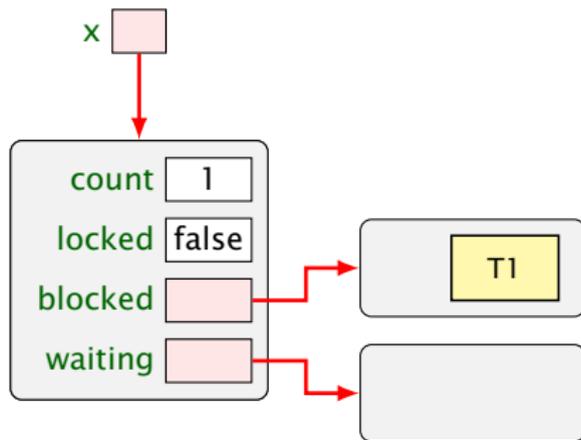
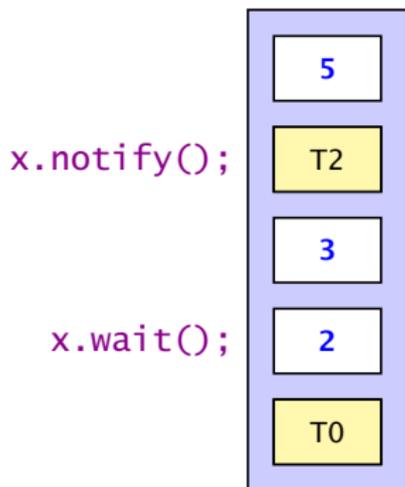
```
public void notify() {  
    if (!waitingThreads.isEmpty()) {  
        Thread t = waitingThreads.dequeue();  
        t.state = blocked;  
        blockedThreads.enqueue(t);  
    }  
}
```

- ▶ `notifyAll()`; weckt alle wartenden Threads auf d.h. fügt alle in `blockedThreads` ein:

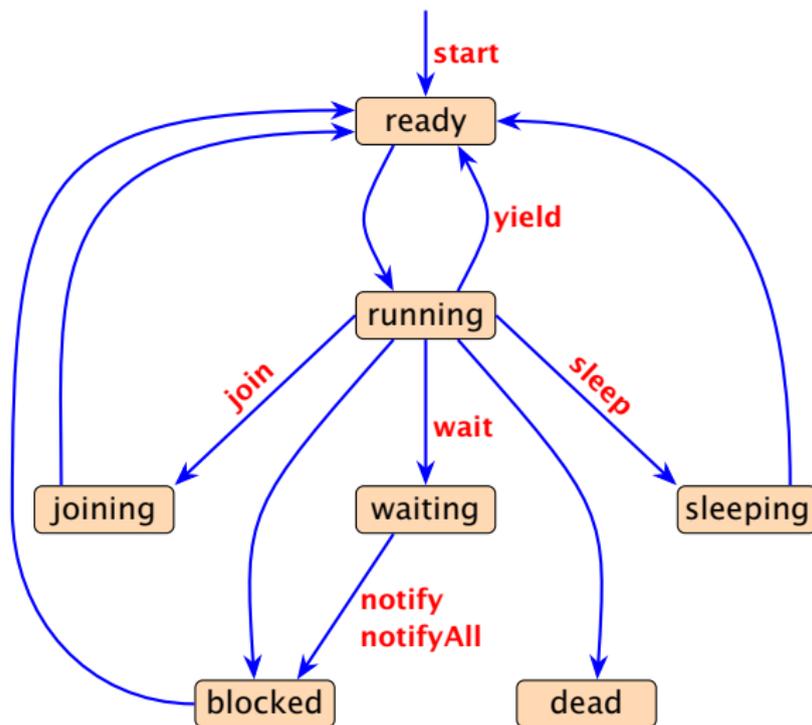
```
public void notifyAll() {  
    while (!waitingThreads.isEmpty()) notify();  
}
```

# Beispiel

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



# Threadzustände



# Implementierung

```
10     public synchronized void produce(Data d)
11         throws InterruptedException {
12         if (free==0) wait(); free--;
13         a[last] = d;
14         last = (last+1) % cap;
15         notify();
16     }
17     public synchronized Data consume()
18         throws InterruptedException {
19         if (free==cap) wait(); free++;
20         Data result = a[first];
21         first = (first+1) % cap;
22         notify();
23         return result;
24     }
25 } // end of class Buffer
```

# Erläuterungen

- ▶ Ist der Puffer voll, d.h. keine Zelle frei, legt sich der Producer schlafen.
- ▶ Ist der Puffer leer, d.h. alle Zellen frei, legt sich der Consumer schlafen.
- ▶ Gibt es für einen Puffer genau einen Producer und einen Consumer, weckt das `notify()` des Consumers stets den Producer (und umgekehrt).
- ▶ Was aber, wenn es **mehrere** Producers gibt? Oder **mehrere** Consumers?

Puffer der Größe 1, mit einem gespeichertem Element:

```
P1->lock(), P1->wait(), C1->lock(), P2->lock(), C1->storeData(), C1->notify(),  
C1->unlock(), P2->removeData(), P2->unlock(), P1->removeData()⚡
```

P1 versucht ein Element zu entfernen, welches nicht mehr vorhanden ist.

# Consumer/Producer

## 2. Idee: Wiederholung des Tests

- ▶ Teste nach dem Aufwecken erneut, ob Zellen frei sind.
- ▶ Wecke nicht einen, sondern alle wartenden Threads auf...

```
9 public synchronized void produce(Data d)
10     throws InterruptedException {
11     while (free==0) wait(); free--;
12     a[last] = d;
13     last = (last+1) % cap;
14     notifyAll();
15 }
```

Man sollte auch bei einem Producer/Consumer immer `while` benutzen und nie nur `if`. Es kann vorkommen, dass ein Thread aufgeweckt wird, ohne dass `notify` aufgerufen wurde (*spurious wakups*).

Problem ohne `notifyAll()` (wieder Puffer der Größe 1 mit einem Element):

```
P1->lock(), P1->wait(), P2->lock(), P2->wait(), C1->lock(), C2->lock(),
C1->removeData(), C1->notify(), C1->unlock(), C2->wait(), P1->storeData(),
P1->notify(), P1->unlock(), P2->wait()⚡
```

Ein Consumer wartet während der Puffer voll ist.

# Consumer/Producer

```
16 public synchronized Data consume()
17     throws InterruptedException {
18     while (free==cap) wait(); free++;
19     Data result = a[first];
20     first = (first+1) % cap;
21     notifyAll();
22     return result;
23 }
```

- ▶ Wenn ein Platz im Puffer frei wird, werden **sämtliche** Threads aufgeweckt — obwohl evt. nur einer der Producer bzw. nur einer der Consumer aktiv werden kann.

# Consumer/Producer

## 3. Idee: Semaphore

- ▶ Producers und Consumers warten in **verschiedenen** Schlangen.
- ▶ Die Producers warten darauf, dass **free > 0** ist.
- ▶ Die Consumers warten darauf, dass **cap-free > 0** ist.

```
1 public class Sema {
2     private int x;
3     public Sema(int n) { x = n; }
4     public synchronized void up() {
5         x++; if (x <= 0) this.notify();
6     }
7     public synchronized void down()
8         throws InterruptedException {
9         x--; if (x < 0) this.wait();
10    }
11 } // end of class Sema
```

Diese Implementierung ist problematisch, da es in Java zu **spurious wakups** kommen kann.

# Semaphore

- ▶ Ein **Semaphor** enthält eine private **int**-Objekt-Variable und bietet die **synchronized**-Methoden **up()** und **down()** an.
- ▶ **up()** erhöht die Variable, **down()** erniedrigt sie.
- ▶ Ist die Variable positiv, gibt sie die Anzahl der verfügbaren Ressourcen an.  
Ist sie negativ, zählt sie die Anzahl der wartenden Threads.
- ▶ Eine **up()**-Operation weckt genau einen wartenden Thread auf.

Machmal wird die **up()**-Routine mit **V()** bezeichnet, und die **down()**-Routine mit **P()**.  
**V** steht für *verhoog* (erhöhen) und **P** für *prolaag*.

# Semaphore – Alternative Implementierung

```
1 public class SemaCorrected {
2     private int waiting, permits;
3     public SemaCorrected(int n) { permits = n;
4         }
5     public synchronized void up() {
6         permits++; if (waiting > 0) notify();
7     }
8     public synchronized void down()
9         throws InterruptedException {
10        waiting++;
11        while (permits <= 0) wait();
12        waiting--;
13        permits--;
14    } // end of class SemaCorrected
```

# Anwendung - 1. Versuch

```
1 public class BufferFaulty {
2     private int cap, first, last;
3     private Sema free, occupied;
4     private Data[] a;
5     public BufferFaulty(int n) {
6         cap = n;
7         first = last = 0;
8         a = new Data[n];
9         free = new Sema(n);
10        occupied = new Sema(0);
11    }
12 // continued...
```

# Anwendung - 1. Versuch

```
13     public synchronized void produce(Data d)
14                                     throws InterruptedException {
15         free.down();
16         a[last] = d;
17         last = (last+1) % cap;
18         occupied.up();
19     }
20     public synchronized Data consume()
21                                     throws InterruptedException {
22         occupied.down();
23         Data result = a[first];
24         first = (first+1) % cap;
25         free.up();
26         return result;
27     }
28 } // end of class BufferFaulty
```

- ▶ Gut gemeint — aber leider fehlerhaft...
- ▶ Jeder Producer benötigt zwei Locks gleichzeitig, um zu produzieren:
  1. dasjenige für den Puffer;
  2. dasjenige für einen Semaphor.
- ▶ Muss er für den Semaphor ein `wait()` ausführen, gibt er das Lock für den Semaphor wieder zurück... nicht aber dasjenige für den Puffer!!!
- ▶ Die Folge ist, dass niemand mehr eine Puffer-Operation ausführen kann, insbesondere auch kein `up()` mehr für den Semaphor ⇒ **Deadlock**

## 2. Versuch – Entkopplung der Locks

```
12 // Methoden sind nicht synchronized
13 public void produce(Data d) throws
14         InterruptedException {
15     free.down();
16     synchronized (this) {
17         a[last] = d; last = (last+1) % cap;
18     }
19     occupied.up();
20 }
21 public Data consume() throws
22         InterruptedException {
23     Data result; occupied.down();
24     synchronized (this) {
25         result = a[first]; first = (first+1) % cap;
26     }
27     free.up(); return result;
28 }
29 } // end of corrected class Buffer
```

- ▶ Das Statement `synchronized (obj) { stmts }` definiert einen kritischen Bereich für das Objekt `obj`, in dem die Statement-Folge `stmts` ausgeführt werden soll.
- ▶ Threads, die die neuen Objekt-Methoden `void produce(Data d)` bzw. `Data consume()` ausführen, benötigen zu jedem Zeitpunkt nur genau ein Lock.

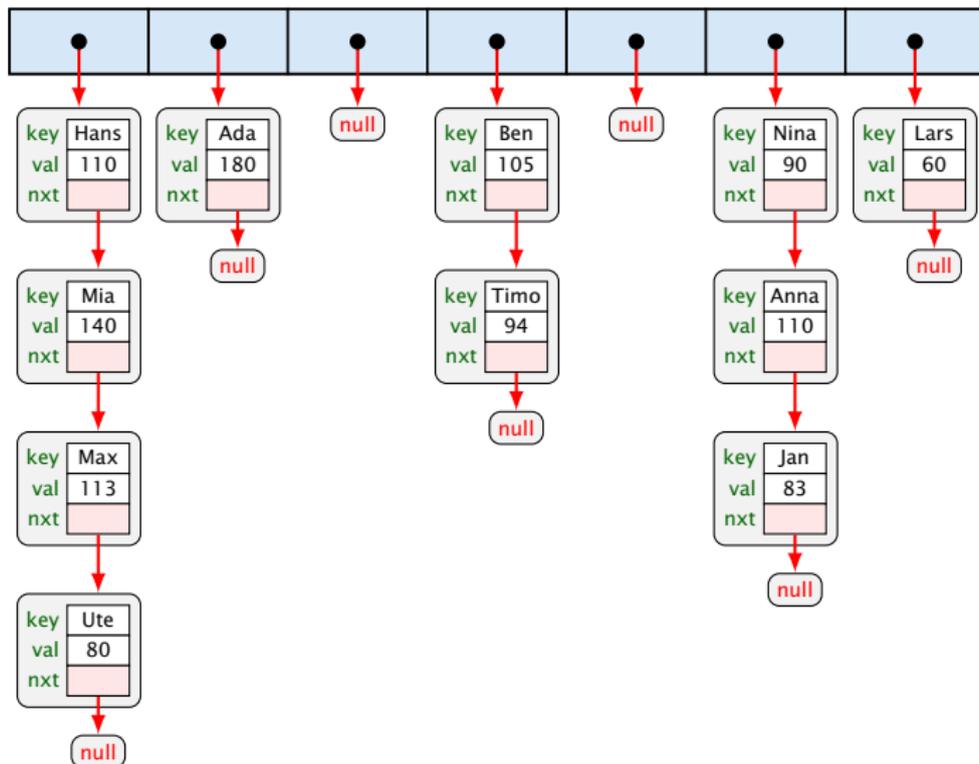
Man kann sich eine `synchronized`-Methode vorstellen, als hätte sie solch einen `synchronized (this)`-Block um den gesamten Funktionsrumpf.

## 16.4 RW-Locks

### Ziel:

- ▶ eine Datenstruktur soll gemeinsam von mehreren Threads benutzt werden.
- ▶ Jeder Thread soll (gefühl) **atomar** auf die Datenstruktur zugreifen.
- ▶ Lesende Zugriffe sollen die Datenstruktur nicht verändern, schreibende Zugriffe dagegen können die Datenstruktur modifizieren.

# Hashtabelle



## 1. Idee: Synchronisiere Methodenaufrufe

```
1 public class HashTable<K,V> {
2     private List2<K,V> [] a;
3     private int n;
4     public HashTable (int n) {
5         a = new List2[n]; // Compilerwarnung
6         this.n = n;
7     }
8     public synchronized V lookup (K k) {...}
9     public synchronized void update (K k, V v) {...}
10 }
```

# Arrays und Generics

Wir sagen `a = new List2[n]` anstatt `a = new List2<K,V>[n]`. Letzteres führt zu einem Compilerfehler.

Wir bekommen hier eine Compilerwarnung, da wir Arrays mit Generics mischen.

Weder der Compiler noch die Laufzeitumgebung kann sicherstellen, dass das Array wirklich nur Elemente vom Typ `List2<K,V>` enthält.

Der Programmierer muss das sicherstellen...

# Diskussion

- ▶ Zu jedem Zeitpunkt darf nur ein Thread auf die HashTable zugreifen.
- ▶ Für schreibende Threads ist das evt. sinnvoll.
- ▶ Threads, die nur lesen, stören sich gegenseitig aber überhaupt nicht!

⇒ **ReaderWriterLock**

- ▶ ist entweder im Lese-Modus, im Schreibmodus oder frei.
- ▶ Im Lese-Modus dürfen beliebig viele Leser eintreten, während sämtliche Schreiber warten müssen.
- ▶ Haben keine Leser mehr Interesse, ist das Lock wieder frei.
- ▶ Ist das Lock frei, darf ein Schreiber eintreten. Das RW-Lock wechselt nun in den Schreib-Modus.
- ▶ Im Schreib-Modus müssen sowohl Leser als auch weitere Schreiber warten.
- ▶ Ist ein Schreiber fertig, wird das Lock wieder frei gegeben. . .

# Implementierung

```
1 public class RW {
2     private int countReaders = 0;
3
4     public synchronized void startRead()
5         throws InterruptedException {
6         while (countReaders < 0) wait();
7         countReaders++;
8     }
9
10    public synchronized void endRead() {
11        countReaders--;
12        if (countReaders == 0) notify();
13    }
14    // continued...
```

# Implementierung

```
15     public synchronized void startWrite()
16         throws InterruptedException {
17         while (countReaders != 0) wait();
18         countReaders = -1;
19     }
20
21     public synchronized void endWrite() {
22         countReaders = 0;
23         notifyAll ();
24     }
25 }
```

# Diskussion

- ▶ Die Methoden `startRead()`, und `endRead()` sollen eine Leseoperation eröffnen bzw. beenden.
- ▶ Die Methoden `startWrite()`, und `endWrite()` sollen eine Schreiboperation eröffnen bzw. beenden.
- ▶ Die Methoden sind `synchronized`, damit sie selbst atomar ausgeführt werden.
- ▶ Die unterschiedlichen Modi eines RW-Locks sind mit Hilfe des Zählers `count` implementiert.
- ▶ Ein negativer Zählerstand entspricht dem Schreib-Modus, während ein positiver Zählerstand die Anzahl der aktiven Leser bezeichnet. . .

# Diskussion

- ▶ `startRead()` führt erst dann kein `wait()` aus, wenn das RW-Lock entweder frei oder im Lese-Modus ist. Dann wird der Zähler inkrementiert.
- ▶ `endRead()` dekrementiert den Zähler wieder. Ist danach das RW-Lock frei, wird ein(!) wartender Thread benachrichtigt; dieses ist automatisch ein schreibender Thread.
- ▶ `startWrite()` führt erst dann kein `wait()` aus, wenn das RW-Lock definitiv frei ist. Dann wird der Zähler auf -1 gesetzt.
- ▶ `endWrite()` setzt den Zähler wieder auf 0 zurück und benachrichtigt dann alle wartenden Threads.

# Die HashTable mit RW-Lock

```
1 public class HashTable<K,V> {
2     private RW rw;
3     private List2<K,V> [] a;
4     private int n;
5     public HashTable (int n) {
6         rw = new RW();
7         a = new List2[n];
8         this.n = n;
9     }
10 // continued...
```

## Die HashTable mit RW-Lock

```
11     public V lookup(K key) throws
12         InterruptedException {
13         rw.startRead();
14         int i = Math.abs(key.hashCode() % n);
15         V result = (a[i] == null) ? null :
16             a[i].lookup(key);
17         rw.endRead();
18         return result;
19     }
```

- ▶ Da `lookup()` nicht weiß, wie mit einem interrupt umzugehen ist, wird die Exception weiter geworfen.

## Die HashTable mit RW-Lock

```
20     public void update(K key, V value) throws
21                           InterruptedException {
22         rw.startWrite();
23         int i = Math.abs(key.hashCode() % n);
24         if (a[i] == null)
25             a[i] = new List2<K,V> (key,value,null);
26         else
27             a[i].update(key,value);
28         rw.endWrite();
29     }
```

- ▶ Die Methode `update(K key, V value)` der Klasse `List2<K,V>` sucht nach Eintrag für `key`. Wird dieser gefunden, wird dort das Wert-Attribut auf `value` gesetzt. Andernfalls wird ein neues Listenobjekt für das Paar `(key,value)` angefügt.

# Diskussion

- ▶ Die neue Implementierung unterstützt nebenläufige Lesezugriffe auf die HashTable.
- ▶ Ein einziger Lesezugriff blockiert aber Schreibzugriffe — selbst, wenn sie sich letztendlich auf **andere Teillisten** beziehen und damit unabhängig sind...
- ▶ Genauso blockiert ein einzelner Schreibzugriff sämtliche Lesezugriffe, selbst wenn sie sich auf **andere Teillisten** beziehen...

⇒ Eingrenzung der kritischen Abschnitte...

Erst suchen der richtigen Liste (**nicht synchronisiert**), dann Lesen/Schreiben dieser Liste (**synchronisiert**).

# Realisierung

```
1 class ListHead<K,V> {
2     private List2<K,V> list = null;
3     private RW rw = new RW();
4     public V lookup (K key) throws InterruptedException {
5         rw.startRead();
6         V result= (list==null) ? null : list.lookup(key);
7         rw.endRead();
8         return result;
9     }
10    public void update (K key, V value) throws
11        InterruptedException {
12        rw.startWrite();
13        if (list == null)
14            list = new List2<K,V>(key,value,null);
15        else list.update(key,value);
16        rw.endWrite();
17 } }
```

# Diskussion

- ▶ Jedes Objekt der Klasse `ListHead` enthält ein eigenes RW-Lock zusammen mit einer Liste (eventuell `null`).
- ▶ Die Methoden `lookup()` und `update` wählen erst (unsynchronisiert) die richtige Liste aus, um dann geordnet auf die ausgewählte Liste zuzugreifen...

```
// in der Klasse HashTable:  
public V lookup (K key) throws InterruptedException {  
    int i = Math.abs(key.hashCode() % n);  
    return a[i].lookup(key);  
}  
public void update (K key, V value)  
                throws InterruptedException {  
    int i = Math.abs(key.hashCode() % n);  
    a[i].update (key, value);  
}
```

# Lock Support in Java

- ▶ ReentrantLock

`lock()`, `unlock()`, `tryLock()`, `newCondition()`

`wait()`, `notify()` heißen `await()` und `signal()` und werden auf einem `Condition`-Objekt ausgeführt.

- ▶ Ein ReentrantLock kann in einer anderen Funktion/einem anderen Block (anders als `synchronized`) freigegeben werden.
- ▶ **Achtung:** Ein Reentrant-Lock kann nicht von einem anderen Thread freigegeben werden.

# Lock Support in Java

```
1 class Test {
2     synchronized void put() throws
3         InterruptedException {
4         ...
5         while (...) wait();
6         ...
7     }
8 }
```

# Lock Support in Java

```
1 class Test {
2     Lock lock = new ReentrantLock();
3     Condition cond = lock.newCondition();
4     void put() {
5         lock.lock();
6         try {
7             ...
8             while (...) cond.await()
9             ...
10        }
11        finally {
12            lock.unlock();
13        }
14    }
15 }
```

# Semaphore Support in Java

- ▶ Klasse `Semaphore`,
- ▶ Konstruktor `Semaphore(int permits)`
- ▶ Hauptmethoden: `acquire()`, `release()`

Kann von einem anderen Thread `release()`d werden als von dem, der `acquire()` aufgerufen hat.

# RW-Locks in Java

- ▶ Klasse `ReentrantReadWriteLock`
- ▶ `readLock()`, `writeLock` gibt das zugehörige `readLock`, bzw. `writeLock` zurück.

```
1 ...
2 ReentrantReadWriteLock rw;
3 rw.readLock().lock();
4 try {
5     ...
6 }
7 finally { rw.readLock().unlock(); }
8 ...
```

# Warnung

Threads sind nützlich, sollten aber nur mit Vorsicht eingesetzt werden. Es ist besser,

- ▶ ... **wenige** Threads zu erzeugen als mehr.
- ▶ ... **unabhängige** Threads zu erzeugen als sich wechselseitig beeinflussende.
- ▶ ... kritische Abschnitte zu **schützen**, als nicht synchronisierte Operationen zu erlauben.
- ▶ ... kritische Abschnitte zu **entkoppeln**, als sie zu schachteln.

# Warnung

Finden der Fehler bzw. Überprüfung der Korrektheit ist ungleich schwieriger als für sequentielle Programme:

- ▶ Fehlerhaftes Verhalten tritt eventuell nur gelegentlich auf. . .
- ▶ bzw. nur für bestimmte Scheduler.
- ▶ Die Anzahl möglicher Programm-Ausführungsfolgen mit potentiell unterschiedlichem Verhalten ist gigantisch.
- ▶ Heisenbugs.

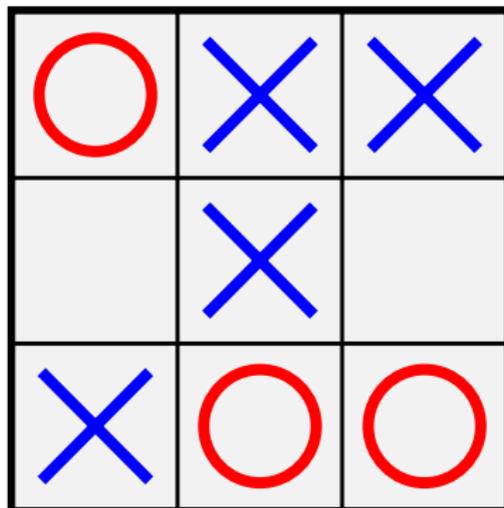
Ein Heisenbug ist ein Bug, der verschwindet wenn man das System beobachtet...

# 17 Tic-Tac-Toe

## Regeln:

- ▶ Zwei Personen setzen abwechselnd **Steine** auf ein  $(3 \times 3)$ -Spielfeld.
- ▶ Wer zuerst drei Steine in einer **Reihe** erreicht, gewinnt.
- ▶ Zeilen, Spalten und Haupt-Diagonalen sind Reihen.

# Beispiel



... offenbar hat die anziehende Partei gewonnen.

## Fragen

- ▶ Ist das immer so? D.h. kann die anziehende Partei immer gewinnen?
- ▶ Wie implementiert man ein **Tic-Tac-Toe**-Programm, das
  - ▶ ...möglichst oft gewinnt?
  - ▶ ...eine **ansprechende** Oberfläche bietet?

# Hintergrund — Zwei-Personen-Nullsummenspiele

Tic-Tac-Toe ist ein endliches **Zwei-Personen-Nullsummen-Spiel**, mit **perfekter Information**. Das heißt:

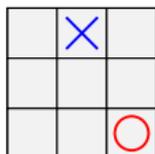
- ▶ Zwei Personen spielen gegeneinander.
- ▶ Was der eine gewinnt, verliert der andere.
- ▶ Es gibt eine endliche Menge von Spiel-**Konfigurationen**.
- ▶ Die Spieler ziehen abwechselnd. Ein **Zug** wechselt die Konfiguration, bis eine **Endkonfiguration** erreicht ist.
- ▶ Jede Endkonfiguration ist mit einem **Gewinn** aus  $\mathbb{R}$  bewertet.
- ▶ **Person 0** gewinnt, wenn Endkonfiguration mit negativem Gewinn erreicht wird; sonst gewinnt **Person 1**.

Man spricht häufig auch von einem **Minimum**-Spieler (Person 0) und einem **Maximum**-Spieler (Person 1).

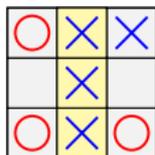
Perfekte Information bedeutet, dass die Spieler alle Informationen besitzen und demzufolge das Spiel (im Prinzip) vollständig berechenbar ist (wie z.B. Mühle, Dame, Schach, Go, etc). Ein Nullsummenspiel ohne vollständige Information ist z.B. Poker.

## ...im Beispiel

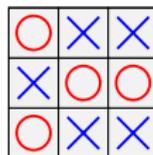
Konfiguration:



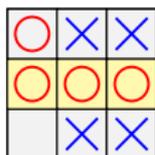
Endkonfigurationen:



Gewinn -1



Gewinn 0

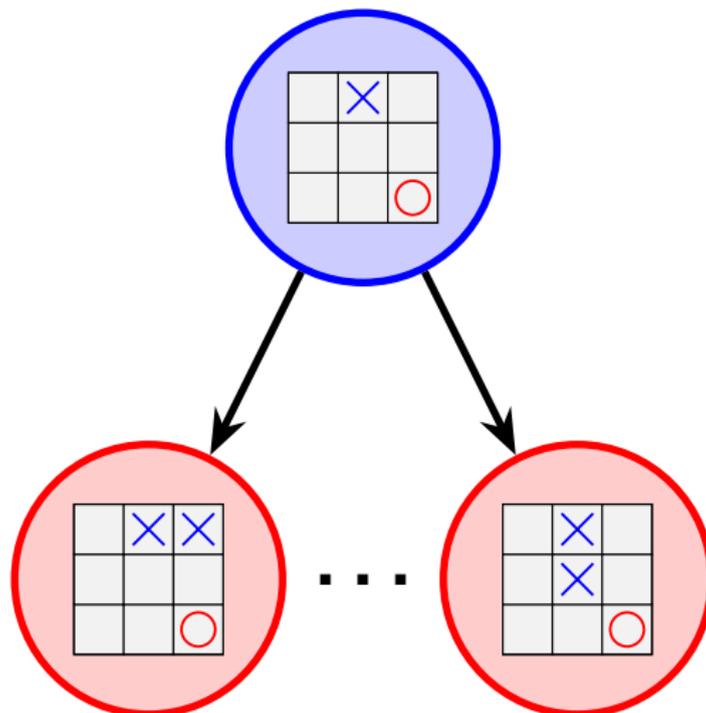


Gewinn +1

## ...im Beispiel

Der Minimum-Spieler (cross/blau), wählt eine der möglichen Nachfolgekonfigurationen.

Spielzug:



# Spielbaum

Ein **Spielbaum** wird folgendermassen (rekursiv) konstruiert:

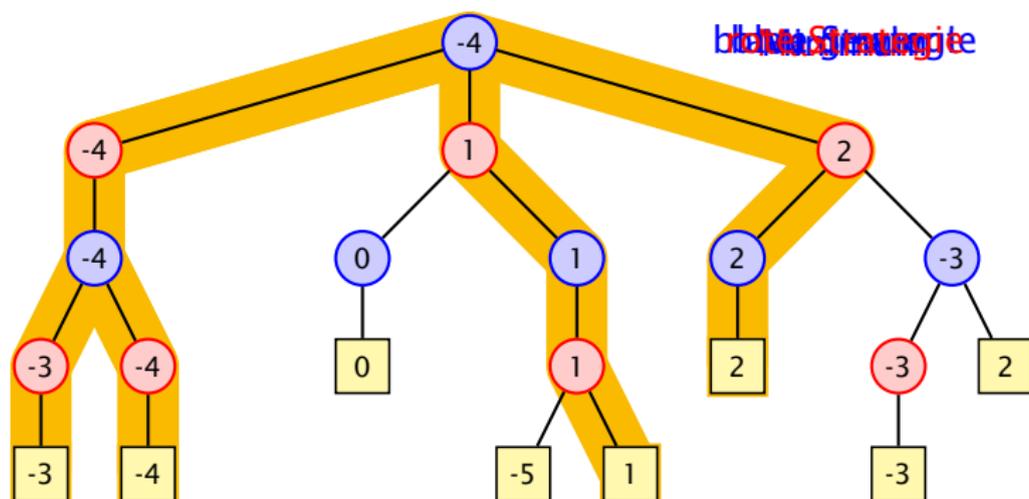
- ▶ gegeben ein Knoten  $v$ , der eine Spielkonfiguration repräsentiert
- ▶ für jede mögliche Nachfolgekongfiguration erzeugen wir einen Kindknoten, den wir mit  $v$  verbinden;
- ▶ dann starten wir den Prozess rekursiv für alle Kindknoten.

## Eigenschaften:

- ▶ jeder Knoten repräsentiert eine Konfiguration; allerdings kann dieselbe Konfiguration sehr oft vorkommen
- ▶ Blattknoten repräsentieren Endkonfigurationen
- ▶ Kanten repräsentieren Spielzüge
- ▶ jedes Spiel ist ein Pfad von der Wurzel zu einem Blatt

## Beispiel — Spielbaum

Dieser Spielbaum repräsentiert ein beliebiges Zwei-Personen-Nullsummenspiel. Deshalb sind die Bewertungen in den Blättern nicht nur  $\{-1, 0, 1\}$ .



## Fragen:

- ▶ Wie finden wir uns (z.B. als **blaue** Person) im Spielbaum zurecht?
- ▶ Was müssen wir tun, um **sicher** ein negatives Blatt zu erreichen?

Der Spielbaum wird üblicherweise so konstruiert, dass die Wurzel der aktuellen Stellung entspricht, in der wir am Zug sind.

## Idee:

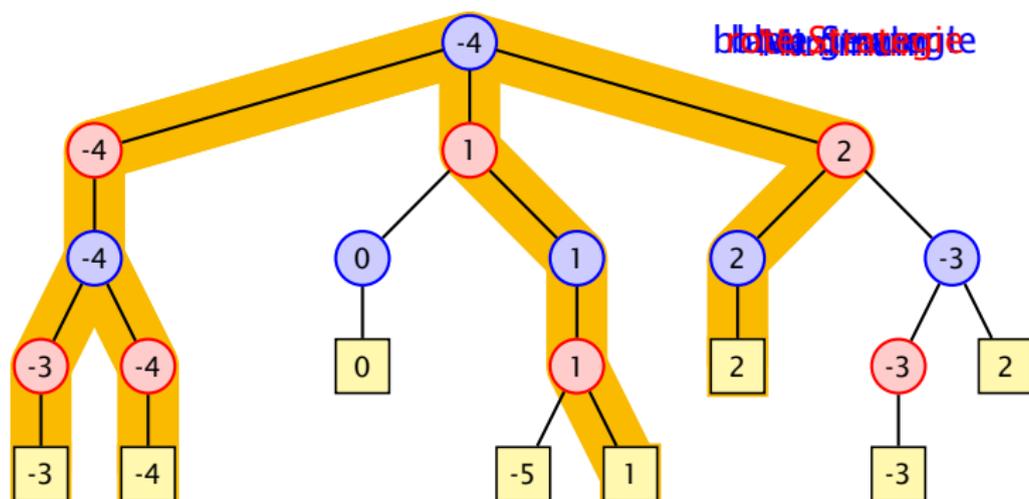
- ▶ Wir ermitteln für jede Konfiguration den jeweils **besten** zu erzielenden Gewinn.
- ▶ Seien die Gewinne für sämtliche Nachfolger einer Konfiguration bereits ermittelt.

**Fall 1** Die Konfiguration ist **blau**: wir sind am Zug. Wir können garantiert das **Minimum** der Gewinne der Söhne erzielen.

**Fall 2** Die Konfiguration ist **rot**: der Gegner ist am Zug. Er kann garantiert das **Maximum** der Gewinne der Söhne erzielen.

## Beispiel — Spielbaum

Dieser Spielbaum repräsentiert ein beliebiges Zwei-Personen-Nullsummenspiel. Deshalb sind die Bewertungen in den Blättern nicht nur  $\{-1, 0, 1\}$ .



# Strategien

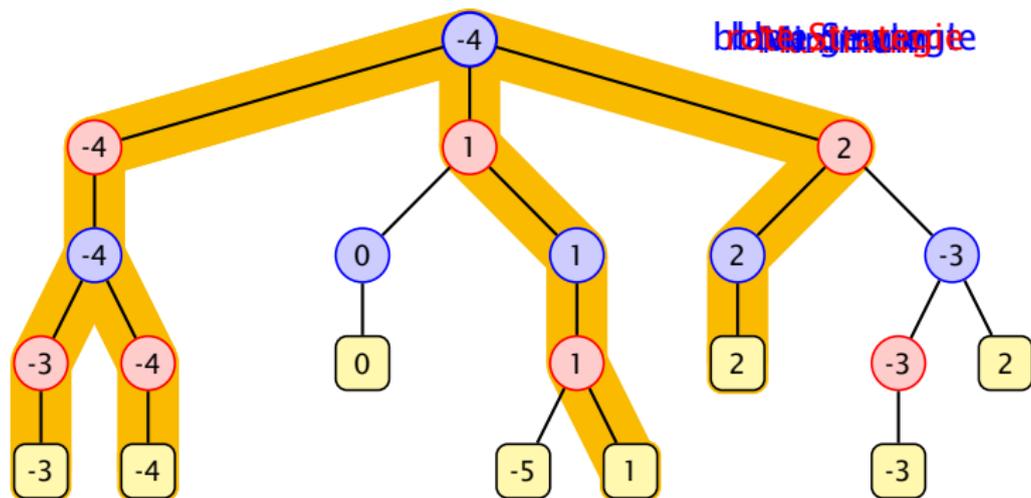
Eine **Strategie** ist eine Vorschrift, die uns in jeder (erreichbaren) Konfiguration mitteilt, welchen Nachfolger wir auswählen sollen.

Eine **optimale** Strategie ist eine, deren Anwendung garantiert zu einer Endkonfiguration führt, deren Wert mindestens so groß ist wie der berechnete garantierte Gewinn.

Eine **akzeptable** Strategie ist eine, deren Anwendung einen Verlust des Spiels verhindert, wann immer das möglich ist ...

## Beispiel — Spielbaum

Dieser Spielbaum repräsentiert ein beliebiges Zwei-Personen-Nullsummenspiel. Deshalb sind die Bewertungen in den Blättern nicht nur  $\{-1, 0, 1\}$ .



# Struktur

GameTreeNode	
- pos	: Position
- type	: int
- value	: int
- child	: GameTreeNode[]
+ GameTreeNode (Position p)	
+ getBestMove	() : int

Position	
- playerToMove	: int
- arena	: int[]
+ Position ()	
+ Position (Position p)	
+ won	() : int
+ getMoves	() : Iterator<Integer>
+ makeMove	(int m)
+ movePossible	(int m) : boolean
+ getPIToMv	() : int

Game	
- g	: GameTreeNode
- p	: Position
- view	: View
+ makeBestMove ()	
+ makePlayerMove ()	
+ movePossible	() : boolean
+ finished	() : boolean

Das Attribut **view** enthält eine Referenz auf ein Objekt, das für die Visualisierung sorgt. Darauf werden wir später ausführlich eingehen. Für ein Verständnis der Spielmodellierung kann man die Aufrufe von **view**-Methoden ignorieren.

Die hier aufgeführten Methoden gehören zum Interface **Model** (mehr dazu im Abschnitt über die GUI-Implementierung).

# Inner Class

```
1 public class OuterClass {
2     private int var;
3     public class InnerClass {
4         void methodA() {};
5     }
6     public void methodB() {};
7 }
```

- ▶ Instanz von `InnerClass` kann auf alle Member von `OuterClass` zugreifen.
- ▶ Wenn `InnerClass` `static` deklariert wird, kann man nur auf statische Member zugreifen.
- ▶ Statische innere Klassen sind im Prinzip normale Klassen mit zusätzlichen Zugriffsrechten.
- ▶ Nichtstatische innere Klassen sind immer an eine konkrete Instanz der äußeren Klasse gebunden.

## Beispiel – Zugriff von Außen

```
1 class OuterClass {
2     private int x = 1;
3     public class InnerClass {
4         void show() {
5             System.out.println("x = " + x);
6         }
7     }
8     public void showMeth() {
9         InnerClass b = new InnerClass();
10        b.show();
11    }
12    public class TestInner {
13        public static void main(String args[]) {
14            OuterClass a = new OuterClass();
15            OuterClass.InnerClass x = a.new InnerClass();
16            x.show();
17            a.showMeth();
18        }
19    }
20 }
```

Um von außen ein Objekt der inneren Klasse zu erzeugen, muss man erst ein Objekt der äußeren Klasse generieren.

Dann erzeugt man ein Objekt der Klasse z.B. durch `objOuter.new InnerClass()`, wobei wir hier annehmen, dass `InnerClass` einen Defaultkonstruktor hat.

"TestInner.java"

## Beispiel – Zugriff von Außen

Normalerweise erzeugt man keine Objekte einer inneren Klasse von außen. Stattdessen bietet häufig die äußere Klasse eine Funktion, die ein Objekt der inneren Klasse zur Verfügung stellt.

```
1 class OuterClass {
2     private static int x = 1;
3     public static class InnerClass {
4         void show() {
5             System.out.println("x = " + x);
6         } }
7     public void showMeth() {
8         InnerClass b = new InnerClass();
9         b.show();
10 } }
11 public class TestInnerStatic {
12     public static void main(String args[]) {
13         OuterClass a = new OuterClass();
14         OuterClass.InnerClass x =
15             new OuterClass.InnerClass();
16         x.show(); a.showMeth();
17 } }
```

Häufig (vgl. verkettete Liste mit innerer Klasse `ListElem`) werden innere Klassen aber auch nur zur Datenkapselung eingesetzt und sind dann privat.

"TestInnerStatic.java"

Eine Anwendung von öffentlichen, inneren Klassen sind z.B. Adapterklassen (vgl. Iterator).

# Local Inner Class

Eine **lokale, innere Klasse** wird innerhalb einer Methode deklariert:

```
1 public class OuterClass {
2     private int var;
3     public void methodA() {
4         class InnerClass {
5             void methodB() {};
6         }
7     }
8 }
```

- ▶ Kann zusätzlich auf die **finalen** Parameter und Variablen der Methode zugreifen.

Man kann nicht von außen auf die Klasse zugreifen. Deshalb machen modifier wie **private**, **public** keinen Sinn und sind nicht erlaubt.

## Beispiel – Iterator

```
1 interface Iterator<T> {  
2     boolean hasNext();  
3     T next();  
4     void remove(); // optional  
5 }
```

- ▶ Ein Iterator erlaubt es über die Elemente einer Kollektion zu iterieren.
- ▶ Abstrahiert von der Implementierung der Kollektion.
- ▶ `hasNext()` testet, ob noch ein Element verfügbar ist.
- ▶ `next()` liefert das nächste Element (falls keins verfügbar ist wird eine `NoSuchElementException` geworfen).
- ▶ `remove()` entfernt das zuletzt über `next()` zugegriffene Element aus der Kollektion.

Falls die Kollektion das Entfernen von Elementen nicht erlaubt, bleibt `remove()` unimplementiert und liefert bei Aufruf eine Exception.

## Beispiel – Iterator

`curr` zeigt auf das Element, das beim letzten Aufruf von `next()` zurückgegeben wurde.

```
1 public class TestIterator implements Iterable<Integer> {
2     Integer[] arr;
3     TestIterator(int n) {
4         arr = new Integer[n];
5     }
6     public Iterator<Integer> iterator() {
7         class MyIterator implements Iterator<Integer> {
8             int curr = arr.length;
9             public boolean hasNext() { return curr>0;}
10            public Integer next() {
11                if (curr == 0)
12                    throw new NoSuchElementException();
13                return arr[--curr];
14            }
15        }
16        return new MyIterator();
17    }
}
```

Das Interface `Iterable<Integer>` „verspricht“ die Implementierung der Funktion `Iterator<Integer> iterator()`.

"TestIterator.java"

# Beispiel – Iterator

## Anwendung des Iterators:

```
18     public static void main(String args[]) {
19         TestIterator t = new TestIterator(10);
20         for (Iterator<Integer> iter = t.iterator();
21             iter.hasNext();) {
22             Integer i = iter.next();
23             System.out.println(i);
24         }
25         for (Integer j : t) {
26             System.out.println(j);
27         }
28     }
29 }
```

Erweiterte Syntax der For-Schleife. Da `t` das Interface `Iterable<Integer>` implementiert, kann man mit dieser Syntax über die Elemente der Kollektion iterieren.

"TestIterator.java"

In diesem Fall wird nur 20 mal null ausgegeben...

# Implementierung - SpielbaumA

`nodeCount` gehört nicht zur Spiellogik sondern zählt nur die Anzahl der Knoten im Spielbaum.

```
1 import java.util.*;
2 public class GameTreeNode implements PlayConstants {
3     static public int nodeCount = 0;
4
5     private int value;
6     private int type;
7     private int bestMove = -1;
8     private Position pos;
9     private GameTreeNode[] child = new GameTreeNode[9];
10
11     public int getBestMove() {
12         return bestMove;
13     }
```

"GameTreeNodeA.java"

- ▶ das interface `PlayConstants` definiert die Konstanten `MIN = -1`, `NONE = 0`, `DRAW = 0`, `EMPTY = 0`, `MAX = 1`;

# Implementierung - SpielbaumA

```
14 public GameTreeNode(Position p) { nodeCount++;
15     pos = p; type = p.getPlToMv();
16     // hab ich schon verloren?
17     if (p.won() != NONE) { value = p.won(); return; }
18     // no more moves --> no winner
19     Iterator<Integer> moves = p.getMoves();
20     if (!moves.hasNext()) { value = DRAW; return; }
21     value = -2*type;
22     while (moves.hasNext()) {
23         int m = moves.next();
24         child[m] = new GameTreeNode(p.makeMove(m));
25         if (type == MIN && child[m].value < value ||
26             type == MAX && child[m].value > value) {
27             value = child[m].value;
28             bestMove = m;
29     } } }
```

## "GameTreeNodeA.java"

Für den Maximumspieler ( $type==1$ ) verhält sich  $-2*type$  wie  $-\infty$  so dass die nachfolgende `while`-Schleife ein Maximum berechnet. Analog für den Minimumspieler.

# Implementierung – SpielbaumA

Die einzigen TicTacToe-spezifischen Informationen in der Klasse `GameTreeNode` sind

- ▶ die Größe des Arrays `child`; wir wissen, dass wir höchstens 9 Züge machen können
- ▶ wir kennen die Gewinnwerte:

`MIN` gewinnt : `value = -1`

unentschieden : `value = 0`

`MAX` gewinnt : `value = +1`

deswegen könne wir z.B. `value` mit `-2*type` initialisieren.

Die anderen Regeln werden in die Klasse `Position` ausgelagert.

# Klasse Position - Kodierung

Das Array `arena` enthält die Spielstellung z.B.:  
`arena = {1,0,-1,0,-1,0,1,-1,1}` bedeutet:

0	1	2
3	4	5
6	7	8

Koordinaten

○		×
	×	
○	×	○

Konfiguration

1	0	-1
0	-1	0
1	-1	1

Kodierung

## Implementierung - Position

```
1 public class Position implements PlayConstants {
2     private int[] arena;
3     private int playerToMove = MIN;
4     public Position() { arena = new int[9]; }
5     public Position(Position p) {
6         arena = (int[]) p.arena.clone();
7         playerToMove = p.playerToMove;
8     }
9     public Position makeMove(int place) {
10        Position p = new Position(this);
11        p.arena[place] = playerToMove;
12        p.playerToMove = -playerToMove;
13        return p;
14    }
15    private boolean free(int place) {
16        return (arena[place] == EMPTY);
17    }
18    public boolean movePossible(int pl) {
19        return (getMoves().hasNext() && free(pl));
20    }
```

# Implementierung - Position

Die Methoden `String toString()` und `int won()` sind hier nicht gezeigt...

```
21 private class PossibleMoves implements Iterator<Integer> {
22     private int nxt = 0;
23     public boolean hasNext() {
24         if (won() != NONE) return false;
25         for (; nxt<9; nxt++)
26             if (free(nxt)) return true;
27         return false;
28     }
29     public Integer next() {
30         if (!hasNext())
31             throw new NoSuchElementException();
32         return nxt++;
33     } }
34 public Iterator<Integer> getMoves() {
35     return new PossibleMoves();
36 }
```

"Position.java"

# Klasse Game

Die Klasse `Game` sammelt notwendige Datenstrukturen und Methoden zur Durchführung des Spiels:

```
1 public class Game implements PlayConstants, Model {
2     private Position    p;
3     private GameTreeNode g;
4     private View view;
5
6     Game(View v) {
7         view = v;
8         p = new Position();
9     }
10    private void initTree() {
11        g.nodeCount = 0;
12        g = new GameTreeNode(p);
13        System.out.println("generate tree... (" +
14                            g.nodeCount + " nodes)");
15    }
```

"Game.java"

# Klasse Game

```
17     private void makeMove(int place) {
18         view.put(place,p.getPlToMv());
19         p = p.makeMove(place);
20         if (finished())
21             view.showWinner(p.won());
22     }
23     public void makeBestMove() {
24         initTree();
25         makeMove(g.getBestMove());
26     }
27     public void makePlayerMove(int place) {
28         makeMove(place);
29         if (!finished()) {
30             makeBestMove();
31         }
32     }
```

"Game.java"

# Klasse Game

```
33     public boolean movePossible(int place) {
34         return p.movePossible(place);
35     }
36     public boolean finished() {
37         return !p.getMoves().hasNext();
38     }
39     public static void main(String[] args) {
40         Game game = new Game(new DummyView());
41         while (!game.finished()) {
42             game.makeBestMove();
43             System.out.println(game.p);
44         }
45     } }
```

"Game.java"

# Output – Variante A

generate tree... (549946 nodes)

x..

...

...

generate tree... (59705 nodes)

x..

.o.

...

generate tree... (7332 nodes)

xx.

.o.

...

generate tree... (935 nodes)

xxo

.o.

...

generate tree... (198 nodes)

xxo

.o.

x..

generate tree... (47 nodes)

xxo

oo.

x..

generate tree... (14 nodes)

xxo

oox

x..

generate tree... (5 nodes)

xxo

oox

xo.

generate tree... (2 nodes)

xxo

oox

xox

Wenn in jedem Spiel genau 9 Züge gemacht würden, dann hätte der Baum  $9! \cdot \left(\frac{1}{1!} + \frac{1}{2!} + \dots + \frac{1}{9!}\right) = 623\,530$  Knoten. Da manche Spiele aber früher beendet sind, kommt man auf einen geringeren Wert.

Wie können wir das effizienter gestalten?

1. Den Spielbaum nur einmal berechnen, anstatt jedesmal neu.  
gewinnt nicht sehr viel...
2. Wenn wir z.B. als MaxPlayer schon einen Wert von 1 erreicht haben, brauchen wir nicht weiterzusuchen...

Spielbaum ist dann unvollständig; Wiederverwendung schwierig...

⇒ Baue keinen vollständigen Spielbaum; nur Wert und Zug an der Wurzel müssen korrekt sein.

# Implementierung - SpielbaumB

Nur Zeile 30 wurde eingefügt.

```
14 public GameTreeNode(Position p) { nodeCount++;
15     pos = p; type = p.getPlToMv();
16     // hab ich schon verloren?
17     if (p.won() != NONE) { value = p.won(); return; }
18     // no more moves --> no winner
19     Iterator<Integer> moves = p.getMoves();
20     if (!moves.hasNext()) { value = DRAW; return; }
21     value = -2*type;
22     while (moves.hasNext()) {
23         int m = moves.next();
24         child[m] = new GameTreeNode(p.makeMove(m));
25         if (type == MIN && child[m].value < value ||
26             type == MAX && child[m].value > value) {
27             value = child[m].value;
28             bestMove = m;
29             // we won; don't search further
30             if (value == type) return;
31     } } }
```

"GameTreeNodeB.java"

# Output – Variante B

generate tree... (94978 nodes)

X..

...

...

generate tree... (3763 nodes)

X..

.O.

...

generate tree... (1924 nodes)

XX.

.O.

...

generate tree... (61 nodes)

XXO

.O.

...

generate tree... (50 nodes)

XXO

.O.

X..

generate tree... (17 nodes)

XXO

OO.

X..

generate tree... (10 nodes)

XXO

OOX

X..

generate tree... (5 nodes)

XXO

OOX

XO.

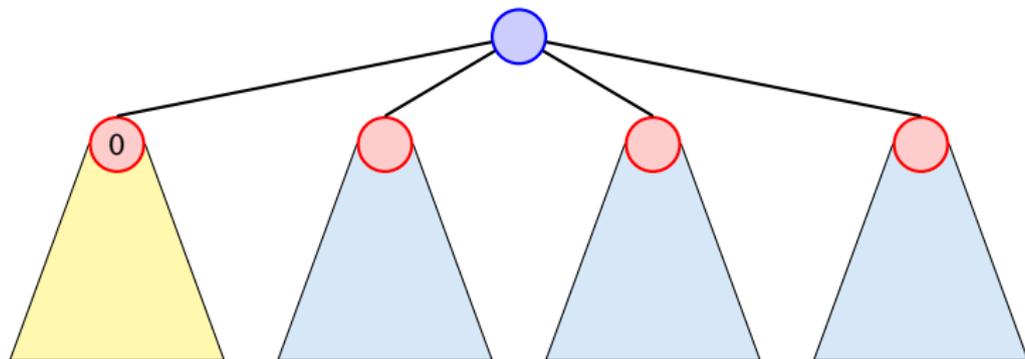
generate tree... (2 nodes)

XXO

OOX

XOX

## Effizienz - Alpha-Beta-Pruning



Ein Wert  $> 0$  innerhalb der blauen Teilbäume kann nicht zu Wurzel gelangen (Wurzel ist MIN-Knoten). Deshalb kann ein MAX-Knoten innerhalb dieser Bäume abbrechen, wenn er einen Wert  $\geq 0$  erzielt hat.

Analog für MIN.

Einige Werte im Spielbaum sind dann nicht korrekt; aber das wirkt sich nicht auf den Wert an der Wurzel aus. Man muss dafür sorgen, dass **bestMove** an der Wurzel nicht auf einen Knoten mit einem inkorrekten Wert zeigt. Wenn z.B. zwei Knoten mit Wert 0 existieren (ein echter, und einer dessen wirklicher Wert größer sein könnte, muss **bestMove** auf den echten zeigen).

# Implementierung - SpielbaumC

Änderungen am Konstruktor:

```
1 private GameTreeNode(Position p,  
2                       int goalMin, int goalMax) {  
3     nodeCount++;  
4     pos = p; type = p.getPlToMv();  
5     if (p.won() != NONE) { value = p.won(); return; }  
6     Iterator<Integer> moves = p.getMoves();  
7     if (!moves.hasNext()) { value = DRAW; return; }  
8  
9     value = -2*type;  
10    while (moves.hasNext()) {  
11      int m = moves.next();  
12      child[m] = new GameTreeNode(p.makeMove(m),  
13                                goalMin, goalMax);  
14    }  
14 // continued...
```

"GameTreeNodeC.java"

# Implementierung - SpielbaumC

Zeilen 24/25 können auch durch `if (goalMin >= goalMax) return;` ersetzt werden.

```
15     if (type == MIN && child[m].value < value ||
16         type == MAX && child[m].value > value) {
17         value = child[m].value;
18         bestMove = m;
19
20         // update goals
21         if (type == MIN && goalMax > value) goalMax=value;
22         if (type == MAX && goalMin < value) goalMin=value;
23         // leave if goal is reached
24         if (type == MIN && value <= goalMin) return;
25         if (type == MAX && value >= goalMax) return;
26     } } // if, while, Konstruktor
27     public GameTreeNode(Position p) {
28         this(p, MIN, MAX);
29     }
```

Wenn wir Zeilen 21/22 löschen, gilt immer `goalMin == MIN` und `goalMax == MAX`. Dann ist dies das Gleiche wie Variante B.

"GameTreeNodeC.java"

In Zeile 15, 16 ist es wichtig, dass wir `bestMove` nur ändern, wenn der neue Wert strikt besser als der alte ist. Es kann sein, dass der Wert von `child[m].value` nicht korrekt ist (z.B. zu klein, wenn wir minimieren). Dann wäre eine Auswahl dieses Nachfolgers als besten Zug schlecht.

# Output – Variante C

generate tree... (16811 nodes)

X..

...

...

generate tree... (1903 nodes)

X..

.O.

...

generate tree... (728 nodes)

XX.

.O.

...

generate tree... (61 nodes)

XXO

.O.

...

generate tree... (50 nodes)

XXO

.O.

X..

generate tree... (17 nodes)

XXO

OO.

X..

generate tree... (10 nodes)

XXO

OOX

X..

generate tree... (5 nodes)

XXO

OOX

XO.

generate tree... (2 nodes)

XXO

OOX

XOX

Bis jetzt haben wir bei den Effizienzsteigerungen das eigentliche Spiel ignoriert.

- ▶ Wenn wir einen Zug haben, der sofort gewinnt, kennen wir den Wert des Knotens und den besten Zug.
- ▶ Falls das nicht zutrifft, aber der Gegner am Zug einen sofortigen Gewinn hätte, dann ist der beste Zug dieses zu verhindern. D.h. wir kennen den besten Zug, aber noch nicht den Wert des Knotens.

`int forcedWin(int player)` in der Klasse `Position` überprüft, ob `player` einen Zug mit sofortigem Gewinn hat.

- ▶ falls ja, gibt es diesen Zug zurück
- ▶ sonst gibt es `-1` zurück

## Implementierung - SpielbaumD

```
1 private GameTreeNode(Position p,
2                       int goalMin, int goalMax) {
3     nodeCount++; pos = p; type = p.getPIToMv();
4     if (p.won() != NONE) { value = p.won(); return; }
5     Iterator<Integer> moves = p.getMoves();
6     if (!moves.hasNext()) { value = DRAW; return; }
7     int m;
8     if ((m=p.forcedWin(type)) != -1) {
9         bestMove = m;
10        value = type;
11        return;
12    }
13    if ((m=p.forcedWin(-type)) != -1) {
14        bestMove = m;
15        child[m] = new GameTreeNode(p.makeMove(m),
16                                   goalMin, goalMax);
17        value = child[m].value;
18        return;
19    }
```

"GameTreeNodeD.java"

## Implementierung - SpielbaumD

```
20 value = -2*type;
21 while (moves.hasNext()) {
22     m = moves.next();
23     child[m] = new GameTreeNode(p.makeMove(m),
24                                 goalMin,goalMax);
25
26     if (type == MIN && child[m].value < value ||
27         type == MAX && child[m].value > value) {
28         value = child[m].value;
29         bestMove = m;
30
31         // update goals
32         if (type == MIN && goalMax > value) goalMax=value;
33         if (type == MAX && goalMin < value) goalMin=value;
34         // leave if goal is reached
35         if (goalMin >= goalMax) return;
36
37 } } // if, while
```

"GameTreeNodeD.java"

# Output – Variante D

generate tree... (2738 nodes)

X..

...

...

generate tree... (271 nodes)

X..

.O.

...

generate tree... (106 nodes)

XX.

.O.

...

generate tree... (9 nodes)

XXO

.O.

...

generate tree... (8 nodes)

XXO

.O.

X..

generate tree... (7 nodes)

XXO

OO.

X..

generate tree... (6 nodes)

XXO

OOX

X..

generate tree... (5 nodes)

XXO

OOX

XO.

generate tree... (2 nodes)

XXO

OOX

XOX

## Was könnte man noch tun?

- ▶ Eröffnungen; für die initialen Konfigurationen den besten Antwortzug speichern.
- ▶ Ausnutzen von Zugumstellungen. Überprüfen ob man die aktuelle Stellung schon irgendwo im Spielbaum gesehen hat (Hashtabelle).
- ▶ Ausnutzen von Symmetrien.

Aber für Tic-Tac-Toe wäre dieses wohl overkill...

Für komplexe Spiele wie Schach oder Go ist eine exakte Auswertung des Spielbaums völlig illusorisch...

# 18 Tic-Tac-Toe: GUI

## Idee:

- ▶ Unbesetzte Felder werden durch **Button**-Komponenten repräsentiert.
- ▶ Die Knöpfe werden in einem  $(3 \times 3)$ -Gitter angeordnet.
- ▶ Wird ein Zug ausgeführt, wird der entsprechende Knopf durch eine (bemale) **Canvas**-Fläche ersetzt (**Kreis** oder **Kreuz**).

# GUI: Model – View – Controller

## Modell (Model):

Repräsentiert das Spiel, den aktuellen Spielzustand, und die Spiellogik.

## Ansicht (View)

Die externe graphische(?) Benutzeroberfläche mit der die Benutzerin interagiert.

## Steuerung (Controller)

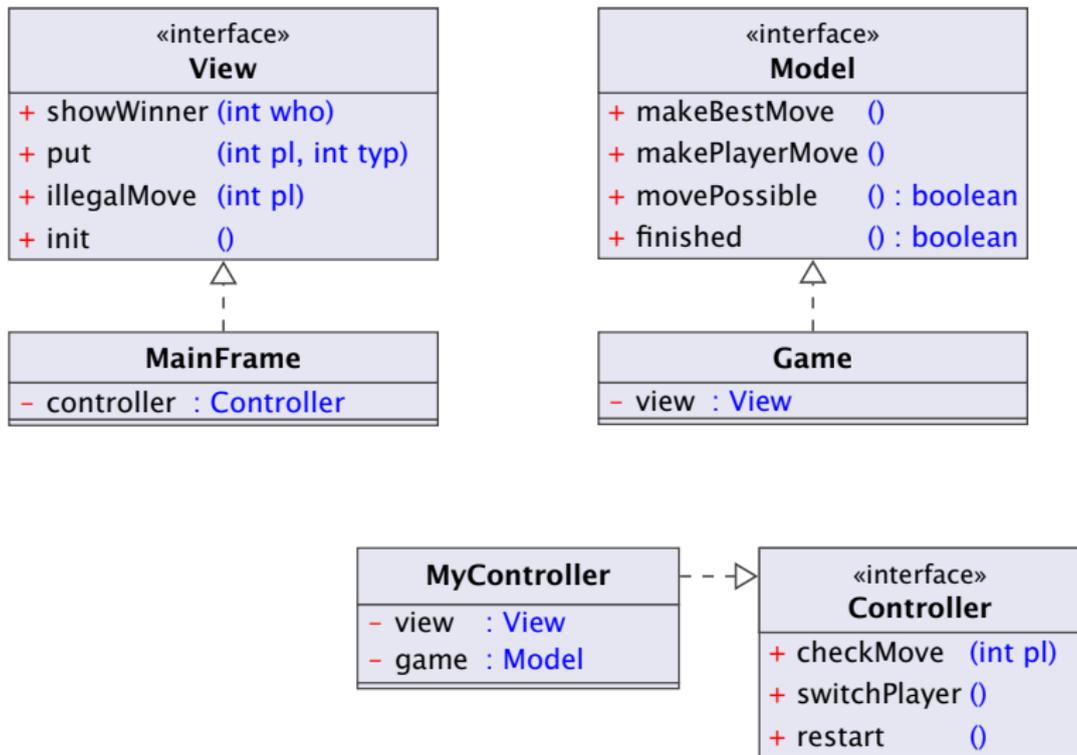
Kontrollschicht, die Aktionen der Nutzerin and die Spiellogik weiterleitet, und Reaktionen sichtbar macht.

Typisch für viele interaktive Systeme. Es gibt viele Varianten (Model-View-Presenter, Model-View-Adapter, etc.).

# GUI: Model – View – Controller

- ▶ Es gibt viele solcher Standardvorgehensweisen, für das Strukturieren, bzw. Schreiben von großen Programmen (**Design Patterns**, ↑**Softwaretechnik**).
- ▶ Es gibt auch **Anti Patterns**, d.h., Dinge, die man normalerweise nicht tun sollte (die aber trotzdem häufig vorkommen).

# TicTacToe - GUI



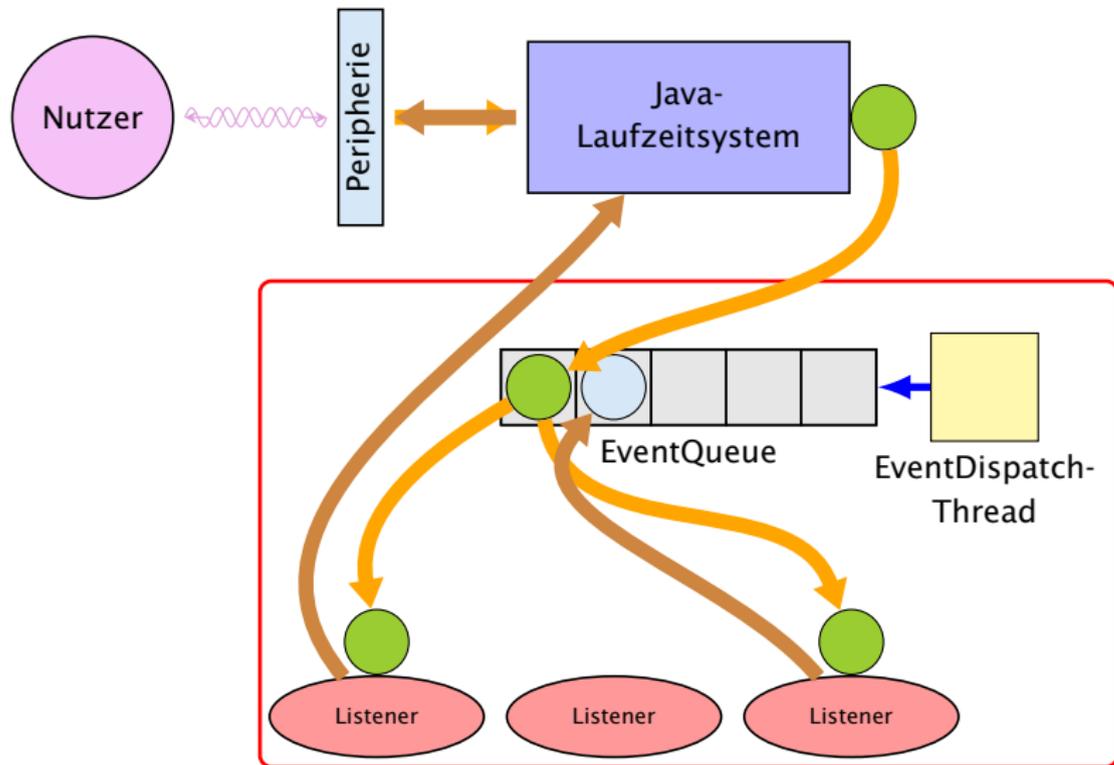
## 18.1 Hintergrund – GUIs

Eine graphische Benutzer-Oberfläche (**GUI**) ist i.A. aus mehreren Komponenten zusammen gesetzt, die einen (hoffentlich) **intuitiven Dialog** mit der Benutzerin ermöglichen sollen.

### Idee:

- ▶ Einzelne Komponenten bieten der Benutzerin Aktionen an.
- ▶ Ausführen der Aktionen erzeugt **Ereignisse**.
- ▶ Ereignisse werden an die dafür zuständigen Listener-Objekte weiter gereicht **Ereignis-basiertes Programmieren**.

# Überblick – Events



# Ereignisse

- ▶ Maus-Bewegungen und -Klicks, Tastatureingaben etc. werden von der Peripherie registriert und an das ↑Betriebssystem weitergeleitet.
- ▶ Das Java-Laufzeitsystem nimmt die Signale vom Betriebssystem entgegen und erzeugt dafür AWTEvent-Objekte.
- ▶ Diese Objekte werden in eine AWTEventQueue eingetragen Producer!
- ▶ Die Ereignisschlange verwaltet die Ereignisse in der Reihenfolge, in der sie entstanden sind, kann aber auch mehrere ähnliche Ereignisse zusammenfassen. . .
- ▶ Der AWTEvent-Dispatcher ist ein weiterer Thread, der die Ereignis-Schlange abarbeitet Consumer!

# Ereignisse

- ▶ Abarbeiten eines Ereignisses bedeutet:
  1. Weiterleiten des **AWTEvent**-Objekts an das Listener-Objekt, das vorher zur Bearbeitung solcher Ereignisse **angemeldet** wurde;
  2. Aufrufen einer speziellen Methode des Listener-Objekts.
- ▶ Die Objekt-Methode des Listener-Objekts hat für die Reaktion des GUI zu sorgen.

Abarbeitung von Events erfolgt sequentiell.

# Ein Button

```
1 import java.awt.*;
2 import java.awt.event.*;
3 import javax.swing.*;
4 import javax.swing.border.*;
5
6 public class MyButton extends JButton {
7     private int number;
8     public MyButton(int i) {
9         number = i;
10        setBackground(Color.WHITE);
11        setBorder(new LineBorder(Color.WHITE, 0));
12    }
13    public int getNumber() { return number; }
14 }
```

- ▶ `MyButton` erweitert die Klasse `JButton` um die Objektvariable `int number`
- ▶ `number` wird vom Konstruktor gesetzt und mit `int getNumber()`; abgefragt.

## Der View

```
8 public class MainFrame extends JFrame implements
9                               PlayConstants,View {
10     private Controller controller;
11     private JDialog dia;
12     private Container c;
13
14     public MainFrame(Controller con) {
15         controller = con;
16         c = getContentPane();
17         setVisible(true);
18         init();
19     }
```

- ▶ Wenn man ein Objekt dieser Klasse anlegt öffnet sich ein Fenster, da die Klasse von `JFrame` abgeleitet ist.
- ▶ `c` ist der Hauptcontainer für graphische Elemente.
- ▶ `init` initialisiert die graphischen Komponenten für ein (neues) Spiel.

## Der View

`pack()` computes the size of the frame, depending on its contents and the `LayoutManager`.

```
20     public void init() { invokeLater(()->{
21         c.removeAll();
22         c.setBackground(Color.BLACK);
23         c.setLayout(new GridLayout(3,3,3,3));
24         c.setPreferredSize(new Dimension(600,600));
25         for (int i=0; i<9;i++) {
26             MyButton b = new MyButton(i);
27             b.addActionListener( this::buttonAction );
28             c.add(b); }
29     pack(); });}
```

- ▶ Die Funktion `void invokeLater(Runnable r)` speichert ein Objekt `r` in der `EventQueue`, das später vom `EventDispatchThread` ausgeführt wird.
- ▶ Jede Operation, die die GUI verändert, muß vom `EventDispatchThread` ausgeführt werden.

- ▶ `removeAll()` entfernt alle graphischen Elemente des Containers;
- ▶ Wir wählen `GridLayout` als `Layout-Manager`.
- ▶ `GridLayout(int row, int col, int cs, int rs);` konstruiert ein `GridLayout`-Objekt mit `row` Zeilen, `col` Spalten sowie Abständen `cs` und `rs` zwischen Spalten bzw. Zeilen.
- ▶ Wir fügen 9 durchnummerierte `MyButton`-Elemente ein.

# Lambda-Ausdrücke

Ein **funktionales Interface** ist ein Interface, das **genau** eine Methode enthält.

```
interface Runnable {  
    void run();  
}
```

Ein **Lambda-Ausdruck** ist das Literal eines Objektes, das ein funktionales Interface implementiert. Z.B.:

## Syntax:

- ▶ allgemein

```
(%Parameterliste) -> {...}
```

- ▶ nur **return**-statement/eine Anweisung (bei **void**-Funktion)

```
(%Parameterliste) -> %Ausdruck
```

- ▶ nur **genau** ein Parameter

```
a -> {...}
```

Die formalen Parameter eines Lambda-Ausdrucks, müssen keine Typbezeichner enthalten. Der notwendige Typ wird durch den Kontext bei der Verwendung des Lambda-Ausdrucks ermittelt.

# Beispiele

Bei Namenskollisionen verhalten sich die beiden Codestücke leicht unterschiedlich. Insbesondere falls der Name `Foo` schon verwendet wird...

```
Runnable r = () -> {System.out.println("Hello!");};
```

ist (im Prinzip) äquivalent zu

```
class Foo implements Runnable {  
    void run() {  
        System.out.println("Hello!");  
    }  
}  
Runnable r = new Foo();
```

# Methodenreferenzen

An der Stelle, an der ein Lambda-Ausdruck möglich ist, kann man auch eine **Methodenreferenz** einer passenden Methode angeben.

## Beispiel:

- ▶ Klasse `ClassA` verfügt über statische Methode `boolean less(int a, int b)`.
- ▶ Das **Funktionsinterface** `Iface` verlangt die Implementierung einer Funktion, die zwei `ints` nach `boolean` abbildet.
- ▶ Außerdem existiert Funktion `sort(int[] a, Iface x)`.
- ▶ Dann sortiert der Aufruf:

```
int[] arr = {5,8,7,2,11};  
sort(arr, ClassA::less);
```

gemäß der durch `less` vorgegebenen Ordnung.

Wenn man eine Objektmethode benutzen möchte müsste man `x::less` schreiben, wobei `x` ein Objekt der Klasse `ClassA` ist.

Vorausgesetzt die Funktion `sort` ist so implementiert, dass sie tatsächlich sortiert...

## Button – Events

- ▶ Wenn ein Knopf gedrückt wird, wird ein `ActionEvent` ausgelöst, d.h., ein Objekt dieser Klasse in die Queue eingefügt.
- ▶ Ein Objekt, das auf solch ein Ereignis reagieren soll muss das Interface `ActionListener` implementieren.

```
interface ActionListener {  
    void actionPerformed(ActionEvent e);  
}
```

- ▶ Die Methode `actionPerformed()` erhält das Objekt zu dem auslösenden Ereignis als Parameter.
- ▶ Wir konstruieren das Listenerobjekt über eine Methodenreferenz.

## Button - Events

```
54     public void buttonAction(ActionEvent e) {  
55         MyButton button = (MyButton) e.getSource();  
56         int place = button.getNumber();  
57         controller.checkMove(place);  
58     }
```

- ▶ Falls ein Knopf gedrückt wird, überprüft der `controller` ob der entsprechende Zug möglich ist.
- ▶ Die Spiellogik (überprüfen des Zuges) ist von der Visualisierung bzw. der Userinteraktion getrennt.

## View – Interfacemethoden

```
30 public void put(int place, int type) {
31     invokeLater(()->{
32         JPanel canvas;
33         if (type == MIN) canvas = new Cross();
34         else canvas = new Circle();
35         c.remove(place);
36         c.add(canvas, place);
37         revalidate();
38         repaint();
39     });
40 }
41 public void illegalMove(int place) {
42     System.out.println("Illegal move: "+place);
43 }
```

"MainFrame.java"

# View – Interfacemethoden

`effectively final` bedeutet, dass man den Parameter als `final` deklarieren könnte, ohne dass es zu einem Compilerfehler führt.

```
44 public void showWinner(int who) {
45     String str = "";
46     switch(who) {
47         case -1: str = "Kreuz gewinnt!"; break;
48         case 0: str = "Unentschieden!"; break;
49         case 1: str = "Kreis gewinnt!"; break;
50     }
51     final String s = str;
52     invokeLater(()->dia = new MyDialog(this,s));
53 }
```

- ▶ Lokale, innere Klassen dürfen auf Parameter der umgebenden Funktion zugreifen, aber nur wenn diese `effectively final` sind.
- ▶ Durch den Lambdaausdruck wird implizit eine lokale, innere Klasse erzeugt.

# Ein Dialog

```
1 public class MyDialog extends JDialog {
2     JButton  repeat, kill;
3     public MyDialog(MainFrame frame, String str) {
4         setSize(300,70);
5
6         setLayout(new FlowLayout());
7         add(new JLabel(str));
8         repeat = new JButton("new");
9         repeat.addActionListener(frame::dialogAction);
10        add(repeat);
11        kill = new JButton("kill");
12        kill.addActionListener(frame::dialogAction);
13        add(kill);
14        setVisible(true);
15    }
16 }
```

Ein `JLabel`-Objekt wird genutzt um den Text anzuzeigen.

Der Konstruktor erhält einen anzuzeigenden Text (`str`) und eine Referenz auf das Objekt in dem die `ActionListener`-Methoden definiert sind.

Wir fügen zwei Knöpfe hinzu, die beide die gleiche `ActionListener`-Funktion aufrufen.

# ActionListener des Dialogs

```
59     public void dialogAction(ActionEvent e) {
60         JButton b = (JButton) e.getSource();
61         if (e.getActionCommand().equals("kill")) {
62             System.exit(0);
63         } else {
64             controller.restart();
65             dia.dispose();
66         }
67     }
```

"MainFrame.java"

- ▶ `getActionCommand()` gibt per default den Text des zugehörigen `JButtons` zurück.
- ▶ `dispose()` löscht das Dialogfenster.

# Controller - Attributes

```
1 import java.awt.*;
2 import java.awt.event.*;
3
4 public class MyController implements PlayConstants,
5                                     Controller {
6     private Model game;
7     private View view;
8     public void setup(Model m, View v) {
9         game = m; view = v;
10    }
```

"MyController.java"

# Controller – Methoden

```
11     public void checkMove(int place) {
12         if (game.movePossible(place)) {
13             game.makePlayerMove(place);
14         }
15         else view.illegalMove(place);
16     }
17     public void switchPlayer() {
18         if (game.finished()) return;
19         game.makeBestMove();
20     }
21     public void restart() {
22         view.init();
23         game = new Game(view);
24 } }
```

"MyController.java"

# Main

```
68     public static void main(String[] args) {
69         invokeLater()->{
70             Controller c = new MyController();
71             View v = new MainFrame(c);
72             Model m = new Game(v);
73             c.setup(m,v);
74         });
75     }
```

"MainFrame.java"

**Was ist hier falsch?**

**Was passiert wenn wir einen sehr grossen Spielbaum berechnen?**

Die eingefügten Befehle simulieren eine Baumberechnung die ziemlich lange dauert. Dies verdeutlicht das Problem.

```
16     private void initTree() {
17         try {Thread.sleep(4000);}
18         catch(InterruptedException e) {}
19         g.nodeCount = 0;
20         g = new GameTreeNode(p);
21         System.out.println("generate tree... (" +
22                             g.nodeCount + " nodes)");
23     }
```

"GameWithDelay.java"

Die GUI reagiert nicht mehr, da wir die gesamte Berechnung im **Event Dispatch Thread** ausführen.

# View - ActionListener

```
54     public void buttonAction(ActionEvent e) {
55         MyButton button = (MyButton) e.getSource();
56         int place = button.getNumber();
57         ctrl.exec(()->ctrl.checkMove(place));
58     }
59     public void dialogAction(ActionEvent e) {
60         JButton b = (JButton) e.getSource();
61         if (e.getActionCommand().equals("kill")) {
62             System.exit(0);
63         } else {
64             ctrl.exec(()->ctrl.restart());
65             dia.dispose();
66         }
67     }
```

"MainFrameNew.java"

Die neue Methode `exec` im Controller startet ein übergebenes `Runnable`-Objekt in einem speziellen Controllerthread. Falls dieser Thread gerade aktiv ist (z.B. mit einer Spielbaumberechnung) passiert gar nichts.

# Controller – Attributes

```
1 import java.awt.*;
2 import java.awt.event.*;
3 import java.util.concurrent.locks.*;
4
5 public class MyController
6     extends Thread
7     implements PlayConstants,
8         Controller {
9     private Model game;
10    private View view;
11    final Lock lock = new ReentrantLock();
12    final Condition cond = lock.newCondition();
```

"MyControllerNew.java"

Die geänderte Klasse `MyController` repräsentiert jetzt einen eigenen `Thread` und enthält Variablen für die Koordination.

## Controller – Methoden

`exec` versucht durch `lock.trylock()` zu erhalten. Falls dieses nicht gelingt blockiert diese Funktion nicht, sondern kehrt einfach zum Aufrufer zurück.

```
14     Runnable r = null;
15     public void exec(Runnable r) {
16         if (lock.tryLock()) {
17             if (r != null) this.r = r;
18             cond.signal();
19             lock.unlock();
20         } }
21     public void run() {
22         lock.lock(); try {
23             while (true) {
24                 while (r == null)
25                     cond.await();
26                 r.run();
27                 r = null;
28             }}
29     catch (InterruptedException e) {}
30     finally { lock.unlock(); }
31 }
```

Die `run()`-Methode des Threads läuft in einer Endlosschleife. Wenn dem `runnable`-Objekt etwas zugewiesen wird, wird die zugehörige Methode ausgeführt.

# Controller – Methoden

```
32     public void setup(Model m, View v) {  
33         game = m; view = v;  
34         start();  
35     }
```