

---

## ▶ Vorlesung Programmieren

Das sind die Webseiten zur Vorlesung "Programmieren mit Java" (Prof. Dr. G. Schiedermeier).

---

### ▶ 1 Einführung

---

#### ▶ 1.1 Erstes Beispiel

---

##### ▶ 1.1.1 Problemstellung

- Berechnung der **Zahlensumme**  $1 + 2 + 3 + \dots + n = ?$
  - $n$  ganzzahlig, positiv, fest **vorgegeben**
  - Geschlossene Formel für Lösung bekannt  $\S \Rightarrow$  Ergebnis **überprüfbar**
- 

##### ▶ 1.1.2 Lösungsidee

- Zuerst  **$n$  festlegen**
  - Ergebnis vorläufig mit **0 vorgeben**
  - Ergebnis durch fortgesetztes **Aufaddieren** akkumulieren
  - Addierschritte mit **Zähler** kontrollieren, der die Werte von 1 bis  $n$  durchläuft
  - Am Ende Ergebnis **ausgeben**
- 

##### ▶ 1.1.3 Algorithmus

- Beispielsweise für  $n = 4$ :
    - 1.) Gib  $n$  den Wert 4
    - 2.) Gib dem Ergebnis den Wert 0
    - 3.) Gib dem Zähler den Wert 1
    - 4.) Wiederhole solange der Zähler kleiner oder gleich  $n$  ist...
      - a.) Erhöhe das Ergebnis um den Wert des Zählers
      - b.) Erhöhe den Zähler um 1
    - 5.) Drucke den Wert des Ergebnisses aus
- 

##### ▶ 1.1.4 Programm

###### Java-Programm

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

```

int n = 4;           // 1.
int ergebnis = 0;  // 2.
int zaehler = 1;    // 3.
while(zaehler <= n)
{
    ergebnis = ergebnis + zaehler; // 4a.
    zaehler = zaehler + 1;         // 4b.
}
System.out.println(ergebnis); // 5.
}

```

## ▶ 1.1.5 Übersetzen

- **Programmtext** in der obigen Form (= "Quelltext") *unverdaulich* für Computer
- Umwandeln in *ausführbare Form* (= "Bytecode")
- Quelltext und Bytecode sind *äquivalent* (= exakt dieselbe Bedeutung)
- Umwandlung automatisch mit "**Compiler**"
- Compiler "javac" *liest Quelltext* aus einer Datei, *schreibt Bytecode* in andere Datei

```
$ javac Zahlensumme.java
```

Bytecode erscheint "unbemerkt" in der *Datei "Zahlensumme.class"* §

## ▶ 1.1.6 Ausführen

- Bytecode kann ausgeführt werden mit "**Virtueller Maschine**"
- Beim Ablauf wird die Berechnung lt. Quelltext *abgewickelt*

```
$ java Zahlensumme
10
```

- *Überprüfung* mit geschlossener Formel zeigt: Ergebnis korrekt

## ▶ 1.2 Programmiersprachen

### ▶ 1.2.1 Merkmale

- Programmiersprachen dienen zum *Formulieren von Programmen*
- Kommunikationsmittel von *Mensch an Maschine*
- Maschine ist äußerst pedantisch, präzise, spröde, ohne jede Phantasie ⇒ Programm muß *absolut exakt* formuliert sein

### ▶ 1.2.2 Generationen

- Erste Programmiersprachen etwa *50 Jahre alt*
  - Frühe Sprachen ausgerichtet auf Eigenschaften *einzelner Computer* (maschinenorientierte Sprachen: Assembler)
  - Später) treten *menschliche Denkmuster* in den Vordergrund (prozedurale Sprachen: Pascal, C)
  - Heute großes Gewicht auf Organisation *großer Programme* (objektorientierte Sprache: C++, Java)
  - In dieser Vorlesung *Java*, Sprache der neuesten Generation
- 

### ▶ 1.2.3 Eigenschaften

- Merkmale von Programmiersprachen auf verschiedenen Ebenen
  - **Syntax** (= Rechtschreibung): regelt korrekte Abfolge von Textzeichen
  - **Semantik** (= Bedeutung): regelt die Auswirkungen einzelner Sprachmittel
  - **Pragmatik**: beschreibt sinnvolle Sprachmuster
- 

### ▶ 1.2.4 Syntax

- Leicht zu erlernen
  - Compiler findet zuverlässig *alle* Syntaxfehler
  - Syntax läßt sich formal, eindeutig definieren
  - Ausdrucksmittel: EBNF–Grammatik (Text), Syntaxdiagramme
  - Java–Grammtik in EBNF: einige Buchseiten
- 

### ▶ 1.2.5 Semantik

- Erlernen durch Vorlesung, eigene Praxis, Fehler
  - Hauptgegenstand dieser Vorlesung
  - Formal nicht sinnvoll beschreibbar §
  - Java–Semantik: viele Bücher, z.T. ganze Serien
  - Elektronische Sprachbeschreibung von Sun verfügbar
- 

### ▶ 1.2.6 Pragmatik

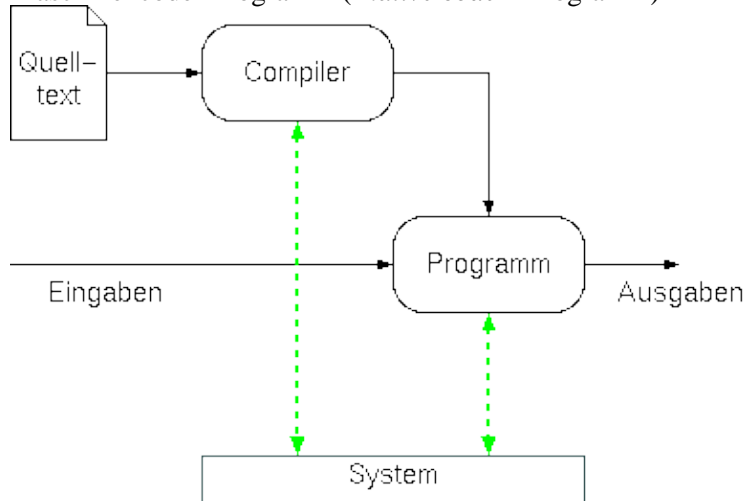
- Sehr schwer zu "erlernen"
- Nur durch langjährige Praxis
- Weitgehend persönlicher Erfahrungsschatz, macht "professionellen" Entwickler aus

- Formale Beschreibung steckt in den Kinderschuhen, aktives Forschungsgebiet §

## ▶ 1.3 Verarbeitungsmodell

### ▶ 1.3.1 Compiler und Maschinencode

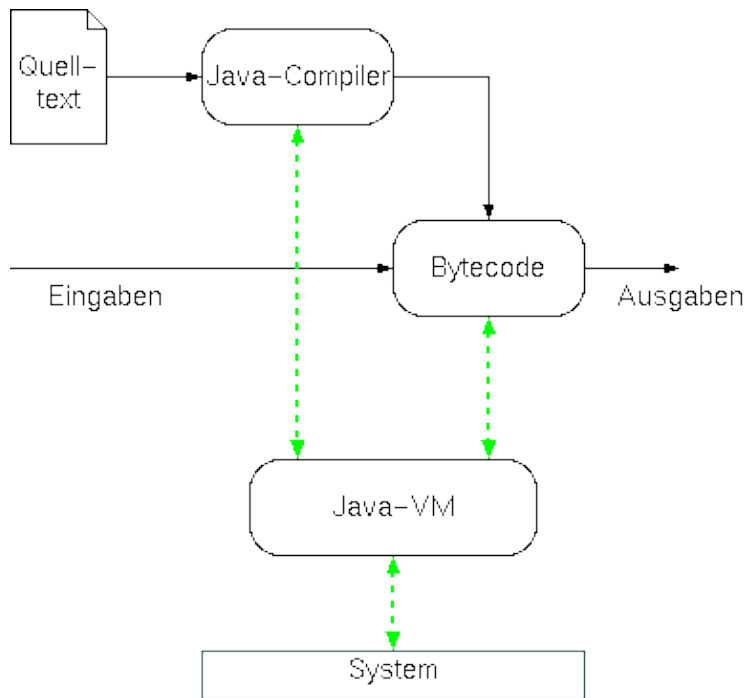
- Klassischer Compiler (C, Pascal, C++, *nicht* Java) § "übersetzt" Programmiersprache in Maschinencode-Programm ("*native code*"-Programm)



- Nachteile: nicht für Menschen lesbar, spezifisch für bestimmtes Rechnersystem §
- Vorteile: kompakt, effizient

### ▶ 1.3.2 Java-Compiler und VM

- Java vermeidet Abhängigkeit des Programms vom Rechnersystem:
- Java-Compiler erzeugt Code ("*Bytecode*") für ein hypothetisches Rechnersystem,
- Hypothetisches Rechnersystem = "*virtuelle Maschine*" = VM oder Java-VM = JVM
- JVM wird auf konkretem Rechner ausgeführt



- VM starten bspweise mit

```
$ java Zahlensumme
10
```

- Vorteil: Bytecode ist *portabel* ⇒ ausführbar auf jedem System, auf dem eine VM läuft
- Aber: Portiabilität des Bytecode wird mit neuen Nachteilen bezahlt

### ▶ 1.3.3 Ressourcenbedarf

- Resource = Betriebsmittel eines Computers: Rechenleistung, Speicher, Plattenkapazität, Peripheriegeräte, ...
- VM braucht Ressourcen
- Javaprogramm läuft langsamer als Native Code-Programm (heute 1.5–5× langsamer)
- Javaprogramm braucht mehr Speicherplatz als Native Code-Programm

### ▶ 1.3.4 Laufzeitbibliotheken

- Viele Algorithmen werden oft gebraucht
- Mit Compiler und VM geliefert
- Sammlung aller vorgegebenen Hilfsmittel: Laufzeit-Bibliothek ("*Java-API*")
- API weitaus komplexer als Compiler + VM
- Professioneller Entwickler: kennt Möglichkeiten (und Grenzen) des API

## 1.4 Java–Quelltext

---

### 1.4.1 Forderung an Quelltext

- Quelltext muß lesbar sein für
    - ◆ Mensch *und*
    - ◆ Maschine (Compiler)
  - Spätere Korrekturen: Autor muß *eigenen* Quelltext wieder lesen (verstehen) können
  - Erweiterungen: Anderer Entwickler muß Quelltext lesen können
  - Teamarbeit: Entwickler müssen Quelltext der Partner lesen können
  - Fazit: **Lesbarkeit** von Quelltext ebenso wichtig wie Korrektheit
- 

### 1.4.2 Dateinamen

- Jede Java–Quelltextdatei definiert **eine Klasse**. §
- Dateiname und Klassenname **müssen übereinstimmen** (einschließlich Groß– und Kleinschreibung)
- Die "*Extension*" (= Endung des Dateinamens nach dem Punkt) **muß ".java"** lauten, bspweise:

```
Zahlensumme.java
```

- Der Java–Compiler produziert aus jeder Klassendefinition im Quelltext eine Bytecodedatei. Bytecodedateien haben den gleichen Namen wie die Klassen und **die Extension ".class"**, also z.B.

```
Zahlensumme.class
```

- Die Java–VM erwartet den Namen einer Bytecodedatei (*ohne Extension!*) und führt diese aus, also z.B.

```
$ java Zahlensumme
```

---

### 1.4.3 Layout

- Leerzeilen, Einrückung, Zwischenraum = "*Layout*"
- Für Compiler irrelevant
- Konventionen:
  - ◆ nur *eine* Anweisung/Zeile,
  - ◆ Einrückung ~ Schachtelung,
  - ◆ Zwischenraum vor & nach Namen,

- ◆ Leerzeilen zwischen Klassen, Methoden, größeren Anweisungsblöcken,
  - ◆ usw...
  - Weitere Konventionen später
  - Beispiel: Programm Zahlensumme §
- 

## ▶ 1.4.4 Kommentare

Siehe auch Sprachdefinition)

- Freitext zur Erläuterung
- Wird vom Compiler ignoriert
- Zeilenendkommentar:

```
// blahblahblah... bis zum Ende der Zeile
```

- Blockkommentar:

```
/* Längere Textpassage,  
auch mehrzeilig,  
mehrseitig, ...  
*/
```

- Zusätzlich in Java: Doc-Kommentare der Form

```
/** Beschreibung einer Definition nach festem Schema  
...  
*/
```

- Kommentare großzügig verwenden
  - Trivialkommentare vermeiden §
- 

## ▶ 1.4.5 Identifier

- Viele Konstrukte in einem Programm sind benannt
- Name = Bezeichner = *Identifier*
- Syntax: Besteht aus Buchstaben und Ziffern
- Konvention: englisch, keine Abkürzungen, "kleinGross"-Schreibweise

- Beispiele: counter  
colorDepth  
iso9660  
runningTotal

```

aber nicht: 1stTry
            Herz Dame
            muenchen-erding
            dies

```

## ▶ 1.4.6 Programmrahmen

- Einfache Javaprogramme haben einheitlichen Rahmen (Vorspann, Abspann)

```

public class NameIhresProgramms
{
    public static void main(String[] args)
    {
        Anweisungen
        ...
    }
}

```

- Dazu einheitlicher Programmkopf

```

/*-----
 * Organisation
 * Autor
 * Projekt, Auftrag
 *
 * Sinn und Zweck dieses Programms
 *
 * Getestet mit
 *      Betriebssystem, Compilerversion
 */
public class ...weiter wie oben

```

- Siehe etwa [Beispielprogramm Zahlensumme](#)

## ▶ 1.4.7 Java–Versionen

- Java von **Sun** Microsystems, Inc., entwickelt
- Nach mehreren Vorläufern **1996 erstmals** veröffentlicht
- Seither Versionen 1.0.X, 1.1.X, 1.2.X, heute **1.3.0**
- **Sprache** Java seit 1.2 fixiert als "**Java 2**"

## ▶ 1.4.8 Software

- Sun Microsystems, Inc., bietet komplette Java–Entwicklungsumgebung ("**JDK**" = Java Development Kit) zur kostenlosen Nutzung
- JDK enthält (u.a.) Compiler (javac), VM (java), Laufzeitbibliothek, Dokumentation
- JDK verfügbar für alle wesentlichen Betriebssysteme

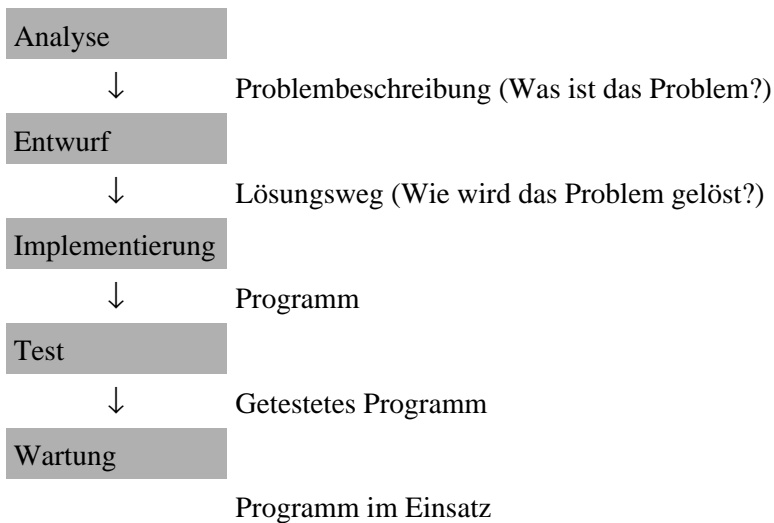
- Weitere Compiler: IBM Research "Jikes", GNU "Guavac" und viele kommerzielle Produkte
  - Weitere VMs: "Kaffe", "TYA", und viele kommerzielle Produkte
- 

## ▶ 1.5 Entwicklungsprozess

---

### ▶ 1.5.1 Software–Lifecycle

- Software = äußerst komplexe Strukturen
- Entwicklung planmäßig, systematisch
- Mehrere Verfahrensschritte



### ▶ 1.5.2 Problemanalyse

- Ziel: exakte, eindeutige, vollständige Beschreibung des zu lösenden Problems
  - Kaum formale Hilfsmittel, umgangssprachlich, in Skizzen, Szenarios, etc.
  - Schwierig bei Problemen von außerhalb der Informatik
  - "*Was* soll erledigt werden?"
- 

### ▶ 1.5.3 Entwurf

- Ziel: Blaupause des Programms
  - Ausdrucksmittel: Struktogramme, UML, Pseudocode, Umgangssprache
  - Erfordert bspweise Kenntnis oder Entwicklung von Algorithmen
  - Unabhängig von Programmiersprache
  - "*Wie* soll das Problem gelöst werden?"
-

## ▶ 1.5.4 Implementierung

- Ziel: funktionsfähiges Programm
  - Ausdrucksmittel: Programmiersprache
  - Hauptgegenstand des WS, zweitrangig in der Praxis
- 

## ▶ 1.5.5 Test

- Ziel: korrektes, robustes Programm
  - Implementierung sichert nur Grundfunktionen
  - Systematisches Testen ist schwierig, erfordert Spezialisten
  - Autor eines Programms denkbar ungeeignet zum Testen
- 

## ▶ 1.5.6 Wartung

- Reibungslosen Betrieb eines Programms im Dauereinsatz
  - Software lebt viel länger als Hardware
  - Dokumentation ist lebensnotwendig
- 

## ▶ 1.5.7 Kosten

- Fehler kommen umso teurer, je später sie auftauchen
  - Größte Sorgfalt in den frühen Phasen des Lifecycle
  - Heute kostet Wartung den Löwenanteil, gefolgt von Test
  - Schwierigster Schritt ist der Entwurf, am einfachsten die Implementierung
  - Psychologisches Problem: Projektfortschritt in der Implementierung am deutlichsten wahrnehmbar
- 

## ▶ 2 Grundbausteine

---

### ▶ 2.1 Numerische Ausdrücke

---

#### ▶ 2.1.1 Numerale

- "*Numeral*" = Zahlenkonstante, fester Zahlenwert
- Vorzeichen "+" oder "-"
- Vorzeichen ist *optional* (= darf weggelassen werden)

- Fehlendes Vorzeichen bedeutet "+"
- Beispiele:

```
0
23
+23
-4000
+230859160
```

- Syntax von Numeralen: §

```
numeral ::= [sign] digit+.
sign ::= "+" | "-".
digit ::= "0" | "1" | "2" | ... | "9".
```

## 2.1.2 Grundrechenarten

- Angelehnt an mathematische Ausdrücke
- Bestandteile: Numerale (Zahlenkonstanten), Operatoren
- Grundrechenarten +, -, \*, /
- Sequentielle Darstellung:  $\frac{3}{4}$  schreiben als

```
3/4
```

- Explizite Multiplikation: Mathematisches "3x" schreiben als

```
3*x
```

- Syntax eines einfachen Ausdrucks: §

```
expression ::= numeral operator numeral.
operator ::= "+" | "-" | "*" | "/".
```

(numeral wie oben definiert)

## 2.1.3 Ganzzahlige Division

- Arithmetik ist ganzzahlig
- Vorsicht insbesondere bei Division, Nachkommaanteil des Quotienten wird ignoriert  
Beispiel:  $11/4 \rightarrow 2$  (statt 2.75)
- Modulus-Operator "%" = Divisionsrest  
Beispiel:  $11\%4 \rightarrow 3$
- Vorsicht mit negativen Operanden! Ergebnis hat das Vorzeichen des Dividenden, also

$-11\%4 \rightarrow -3$ , denn  $-11 = (-2)*4 - 3$

$$11 \% -4 \rightarrow 3, \quad \text{denn } 11 = (-2) * (-4) + 3$$

$$-11 \% -4 \rightarrow -3, \quad \text{denn } -11 = (-2) * 4 - 3$$

- Allgemein gilt:

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

## 2.1.4 Geschachtelte Ausdrücke

- **Induktive Definition:**

1. Ein Numeral ist ein Ausdruck ("elementarer Ausdruck")
2. Zwei Ausdrücke mit einem Operator dazwischen sind auch ein Ausdruck ("zusammengesetzter Ausdruck")

- **Syntax** eines (allgemeinen) Ausdrucks: §

```
expression ::= numeral.
expression ::= expression operator expression.
operator ::= "+" | "-" | "*" | "/" | "%".
```

- Beispiele:

```
3 + 2*4
2*3 + 4*5
100 - 10 - 20
```

## 2.2 Operatoren

### 2.2.1 Priorität

- Priorität = Operatorenvorrang = Bindungsstärke
- Regelt die **Reihenfolge** der Anwendung bei *verschiedenen konkurrierenden* Operatoren

- Beispiel ("Punkt-vor-Strich"):

```
3*4 + 2
→ 12 + 2
→ 14
```

- **Klammern** zur expliziten Vorgabe der Auswertung

- Beispiel:

```
3*(4 + 2)
→ 3*6
→ 18
```

### 2.2.2 Unäre Operatoren

- Unäre Operatoren (Vorzeichenoperatoren) "+" und "-" §

- "Unär" = *ein* Operand (im Gegensatz zu *binären* Operatoren mit zwei Operanden)
- Priorität höher als binäre Operatoren
- Beispiel:  
 $3 * -4$   
 $\rightarrow -12$
- Beispiel:  
 $-3 + -4$   
 $\rightarrow -7$

### 2.2.3 Assoziativität

- Assoziativität = Bindungsrichtung
- Regelt die **Reihenfolge** der Anwendung bei *mehreren gleichen* § Operatoren (im Gegensatz zur Priorität für *verschiedene* Operatoren)
- Irrelevant für assoziative Operatoren § (" $+$ " und " $*$ ")
- Beispiel ("links nach rechts"):
 
$$7 - 3 - 2$$

$$\rightarrow 4 - 2$$

$$\rightarrow 2$$
- **Klammern** zur expliziten Vorgabe der Auswertung
- Beispiel:
 
$$7 - (3 - 2)$$

$$\rightarrow 7 - 1$$

$$\rightarrow 6$$

### 2.2.4 Syntax und Operatorentabelle

- Syntax kann Prioritäten und Assoziativitäten erfassen:

```

expression ::= expression addop term | term.
addop ::= "+" | "-".
term ::= term multop factor | factor.
multop ::= "*" | "/" | "%".
factor ::= unop factor | numeral | "(" expression ")".
unop ::= "+" | "-".
  
```

- Übersichtlicher: Operatorentabelle

<b>Priorität</b>	<b>Operator</b>	<b>Stelligkeit</b>	<b>Assoziativität</b>	<b>Bemerkung</b>
1	+	1	R→L	positives Vorzeichen
	-	1	R→L	negatives Vorzeichen
2	*	2	L→R	Multiplikation
	/	2	L→R	Division

	%	2	L→R	Modulus = Divisionsrest
3	+	2	L→R	Addition
	-	2	L→R	Subtraktion

## Komplette Operatortabelle

- Bisher nur ein kleiner Teil der Java-Operatoren erschlossen
- Hier vollständige Operatortabelle §

<i>Pri.</i>	<i>Operator</i>	<i>Stelligkeit</i>	<i>Operanden</i>	<i>Ass.</i>	<i>Bemerkung</i>
1	+	1	numerisch	R→L	positives Vorzeichen
	-	1	numerisch	R→L	negatives Vorzeichen
	++	1	numerisch	R→L	Prä- oder Postfix-Inkrement
	--	1	numerisch	R→L	Prä- oder Postfix-Dekrement
	~	1	ganzzahlig	R→L	Bit-Komplement
	!	1	boolean	R→L	logisches Not
	( <i>type</i> )	1	beliebig	R→L	Explizite Typkonversion
2	*	2	numerisch	L→R	Multiplikation
	/	2	numerisch	L→R	Division
	%	2	numerisch	L→R	Modulus = Divisionsrest
3	+	2	numerisch	L→R	Addition
	-	2	numerisch	L→R	Subtraktion
	+	2	String	L→R	String-Konkatenation
4	<<	2	ganzzahlig	L→R	Bitshift nach links
	>>	2	ganzzahlig	L→R	Bitshift nach rechts mit Vorzeichen-Einzug
	>>>	2	ganzzahlig	L→R	Bitshift nach rechts mit 0-Einzug
5	<	2	numerisch	L→R	Vergleich echt kleiner
	<=	2	numerisch	L→R	Vergleich kleiner oder gleich
	>	2	numerisch	L→R	Vergleich echt größer
	>=	2	numerisch	L→R	Vergleich größer oder gleich
	instanceof	2	Objekt, Typ	L→R	Typvergleich
6	==	2	primitiv	L→R	Gleichheit
	!=	2	primitiv	L→R	Nicht Gleichheit
	==	2	Objekt	L→R	Identität
	!=	2	Objekt	L→R	Nicht Identität

7	&	2	ganzzahlig	L→R	Bitweises Und
	&	2	boolean	L→R	Logisches Und (vollständig)
8	^	2	ganzzahlig	L→R	Bitweises exclusives Oder
	^	2	boolean	L→R	Logisches exclusives Oder (vollständig)
9		2	ganzzahlig	L→R	Bitweises Oder
		2	boolean	L→R	Logisches Oder (vollständig)
10	&&	2	boolean	L→R	Logisches Und (teilweise)
11		2	boolean	L→R	logisches Oder (teilweise)
12	?:	3	boolean, beliebig, beliebig	R→L	bedingte Auswertung
13	=	2	Variable, beliebig	R→L	Zuweisung
	*=	2	Variable, beliebig	R→L	Zuweisung mit Operator
	/=	2	Variable, beliebig	R→L	Zuweisung mit Operator
	%=	2	Variable, beliebig	R→L	Zuweisung mit Operator
	+=	2	Variable, beliebig	R→L	Zuweisung mit Operator
	--=	2	Variable, beliebig	R→L	Zuweisung mit Operator
	<<=	2	Variable, beliebig	R→L	Zuweisung mit Operator
	>>=	2	Variable, beliebig	R→L	Zuweisung mit Operator
	>>>=	2	Variable, beliebig	R→L	Zuweisung mit Operator
	&=	2	Variable, beliebig	R→L	Zuweisung mit Operator
	^=	2	Variable, beliebig	R→L	Zuweisung mit Operator
	=	2	Variable, beliebig	R→L	Zuweisung mit Operator

## 2.3 Variablen und Wertzuweisung

### 2.3.1 Variablen

- "Variable" = benannter Behälter für einen Wert
- Benennung von Variablen: beliebiger *Name* (= "Identifizier")
- Syntax: §

```

identifizier ::= letter (letter | digit)*.
letter ::= "a" | "b" | "c" | ... "z" |
          "A" | "B" | "C" | ... "Z".
digit ::= "0" | "1" | "2" | ... "9".

```

- Vorsicht: große Buchstaben *verschieden* von kleinen Buchstaben

## 2.3.2 Definition

- Variablen müssen *definiert* werden:

```
int n;
int zaehler;
int ergebnis;
```

- Definition = Anweisung in einem Javaprogramm, um Variable zur weiteren Verwendung bekannt zu geben

- Syntax:

```
definition ::= type identifier ";"
type ::= "int".
```

- Ein Name darf *nur 1x definiert* werden
- Bestimmte Namen sind reserviert ("*Schlüsselwörter*")  $\Rightarrow$  dürfen *nicht* als Variablennamen benutzt werden (wie z.B. "int", "main", "class")

## 2.3.3 Wertzuweisung

- Wertzuweisung ("*Assignment*") = Anweisung in einem Javaprogramm
- Kopiert einen Wert in eine Variable
- Der *vorhergehende* Wert der Variablen wird ersatzlos überschrieben!
- Syntax:

```
assignment ::= identifier "=" expression ";"
(identifier und expression wie weiter oben definiert)
```

- Beispiele:

```
absoluteZero = -273;
daysOfYear = 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 30 + 31;
theAnswer = 179%18*5/2;
```

- Java-Variablen nicht verwechseln mit mathematischen Variablen: Java-Variablen können immer wieder einen *neuen Wert erhalten*. In einem Javaprogramm macht es bspweise Sinn zu schreiben

```
counter = counter + 1;
```

(Der Wert von counter wird um 1 erhöht.) Für mathematische Variablen ist das Unfug.

## 2.3.4 Konstanten

- Variablen können gegen Wert-Änderung geschützt werden ("konstante Variablen") §

- In der Definition `final` vorsetzen
- Syntax

```
definition ::= ["final"] type identifier ";"
type ::= "int".
```

- Zweck: Schutz vor unabsichtlichen Änderungen (z.B. in einem komplexen Programm)
- Beispiel:

```
final int speedOfLight;
speedOfLight = 299793218; // Meter/sek
```

- Nur *exakt eine Wertzuweisung* erlaubt (aber zu einem beliebigen Zeitpunkt)
- Jeder Versuch einer weiteren Wertzuweisung bricht das Programm ab
- Allgemein: Möglichst viele Variablen `final` definieren, wenn aufgrund der Programmlogik keine Wertänderung sinnvoll

### 2.3.5 L- und R-Values

- Wertzuweisung hat zwei Seiten: links und rechts des Zuweisungszeichens
  - Linke Seite*  
*wohin* = Ort an dem ein Wert abgelegt werden soll ("*L-value*")
  - Rechte Seite*  
*was* = Wert der gespeichert werden soll ("*R-value*")
- *L-value* und *R-value* sind fundamentale Konzepte, die sich durch die ganze Programmierung ziehen
- Variable kann links und rechts vorkommen, spielt unterschiedliche Rolle
  - L-value*  
Schreibender Zugriff auf die Variable
  - R-value*  
Lesender Zugriff auf die Variable

- Beispiel:

```
fahrenheit = 80;
celsius = 4*fahrenheit/7 - 32;
```

- Anders gesehen:
  - L-value*  
hat bestimmte Größe, Lage im Hauptspeicher ("Adresse")
  - R-value*  
existiert nur vorübergehend, für kurze Zeit oder gefangen in einer Variablen ("Zwischenergebnis")

### 2.3.6 Anweisungsarten

- Ein Javaprogramm besteht aus einer Folge von Anweisungen §

- Bisher zwei Arten von Anweisungen:
  - ◆ Definition,
  - ◆ Wertzuweisungen  
(Ausdrücke sind Komponenten von Anweisungen, aber selbst keine Anweisungen) §
- Definition und Wertzuweisungen können gemischt werden...
- ...*aber*: Jede Variable muß *vor der ersten Verwendung* definiert werden. Folgendes Fragment ist korrekt:

```
int a;
a = 0;
int b;
b = 2*a;
```

Dagegen wäre fehlerhaft:

```
int a;
a = 0;
b = 2*a;      // undefined Variable: b
int b;
```

- Variablen können sofort bei der Definition mit einem ersten Wert versorgt werden: **Initialisierung** §
- Syntax:

```
definition ::= type identifier ["=" expression] ";".
type ::= "int".
```

- Beispiel:

```
int a = 0;
int b = 2*a;
```

## ▶ 2.4 Numerische Typen

### ▶ 2.4.1 Gleitkommawerte

- Bisher in Definitionen: "int" = Typ für ganzzahlige Werte
- int ungeeignet für reelle Werte wie 3.14 oder  $6 \cdot 10^{23}$
- Neuer **numerischer Typ** "double" § für **Gleitkommawerte** = "Floating point"-Werte ("Fp")
- double umfaßt int zuzüglich
  - ◆ Nachkommaanteil
  - ◆ Zehner-Exponenten  
oder beides

### ▶ 2.4.2 Notation

- double-Literale: Dezimalpunkt zwischen Vorkommaanteil und Nachkommaanteil

```
0.5
3.141592
1.0
```

- 10er-Exponent mit "E" (oder "e") anhängen:  
 $2E4 = 2 \cdot 10^4 = 20000.0$
- Exponenten immer ganzzahlig, nicht gebrochen:  $2E0.5$  für die Wurzel aus 2 ist nicht zulässig
- Negative Exponenten für kleine Werte:

$$1E-3 = 1 \cdot 10^{-3} = \frac{1}{1000}$$

$$-4.17E-4 = -4.17 \cdot 10^{-4} = -0.000417$$

- Verschiedene Darstellungen derselben Zahl:  
-20.0  
-0.2E2  
-200000E-4
- *Vorsicht*: Floatingpoint- und Ganzzahlkonstanten sind unterschiedliche Numerale: 20 ist verschieden von 20.0

### 2.4.3 Polymorphismus von Operatoren

- Arithmetische Operatoren funktionieren mit `int` und `double`:  
 $20 / 8 \rightarrow 2$   
 $20.0 / 8.0 \rightarrow 2.5$
- Beidesmal `/`, aber unterschiedliche Ergebnisse
- Allgemeines Phänomen "*Polymorphismus*"
  - ◆ *dieselbe Bezeichnung* (hier `/`), aber
  - ◆ *verschiedene Funktionsweise* je nach Kontext (hier ganzzahlige resp. floatingpoint-Division)

### 2.4.4 Genauigkeit vs. Geschwindigkeit

- Frage: Wozu `int`, wenn `double` alles genauer rechnet?
- **Laufzeit!** (= "*Performance*"): `double`-Arithmetik ist langsamer
- **Platzbedarf!** `double`-Werte brauchen mehr Platz §
- **Rechenfehler!**  $(1/x) * x$  liefert beispielsweise nicht unbedingt 1 §
- Fazit: **int wenn möglich, double wenn nötig**

## 2.4.5 Implizite Typkonversion

- Operanden *verschiedener Typen* in einem Ausdruck:

1 + 2 → 3

1.0 + 2.0 → 3.0

1 + 2.0 → ?

- Regel: Ergebnis hat Typ des *genaueren Operanden* (im dritten Beispiel `double`, d.h. 3.0)
- Zuweisung von `int` an `double`

```
double d;
d = 4;           // Ok: links double, rechts int
Ok, kein Informationsverlust
```

- **Implizite Typkonversion** `int` → `double` automatisch, stillschweigend
- Zuweisung von `double` an `int` unzulässig

```
int i;
i = 3.14;       #fehler: links int, rechts double
```

## 2.4.6 Explizite Typkonversion (*Typecast*)

- **Keine (automatische) Typkonversion** `double` → `int`, weil (potentieller) Informationsverlust!
- Typkonversion `double` → `int` kann erzwungen werden mit *Typecast* = **explizite Typkonversion**
- Syntax allgemein:

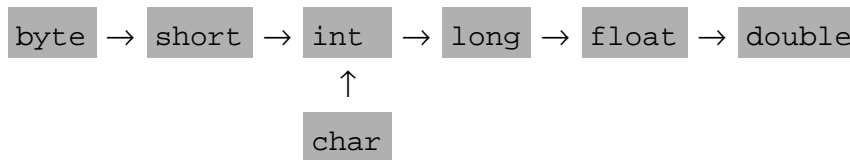
```
(type)expression
```

- Typecast entspricht einstelligem Operator mit hoher Priorität
- Beispiel

```
int i;
i = (int)3.14;  #kein Fehler: explizite Typkonversion
```

## 2.4.7 Konversion numerischer Typen

- Konversion von "kleinen" zu "großen" Typen implizit, weil *ohne* Informationsverlust
- Konversion in der umgekehrten Richtung nur mit Typecast
- Benutzer übernimmt die Verantwortung für möglichen Informationsverlust
- Konversionswege der numerischen Typen:



[siehe Guido Krüger: "Go To Java 2"]

## 2.4.8 Wertebereiche

- Für Werte wird Speicherplatz gebraucht
- Menge des reservierten Speicherplatzes charakteristisch für den Typ.
- Mit begrenztem Speicherplatz lassen sich nur endlich viele Werte codieren → es gibt eine kleinste und eine größte darstellbare `int`-Zahl (entsprechend für `double`)

Typ	Speicherplatz	minimaler Wert	maximaler Wert
<code>int</code>	4 Byte = 32 Bit	$-2^{31} = -2147483648$	$2^{31}-1 = +2147483647$
<code>double</code>	8 Byte	$\pm 4.94065 \cdot 10^{-324}$	$1.79769 \cdot 10^{308}$

- In Java sind die Wertebereiche und die Bitcodierung § fest definiert und gelten für alle Implementierungen §

## 2.4.9 Bereichsüberschreitung

- Numerische Operationen können überlaufen oder undefinierte Ergebnisse produzieren

- Die Behandlung unterscheidet sich bei `int` und `double`

*int-Überlauf*

Wird nicht abgefangen, das Programm **rechnet kommentarlos falsch**:

`2147483647 + 2147483647 → -2`

*Undefiniertes int-Ergebnis*

Nur bei `/` und `%` mit rechtem Operanden Null, **bricht das Programm ab**: §

`23/0 → "ArithmeticException: / by zero"`

*double-Überlauf*

Liefert den **Fluchtwert "Infinity"**:

`1e308 + 1e308 → Infinity`

*Undefiniertes double-Ergebnis*

Liefert den **Fluchtwert "NaN"** (*Not a Number*):

`Math.sqrt(-1) → NaN`

## 2.5 Bibliotheksfunktionen

### 2.5.1 Vordefinierte Algorithmen

- Viele Aufgaben müssen oft gelöst werden (Beispiel: Wurzel ziehen)
- Algorithmen sind vordefiniert, in Bibliothek gesammelt, mit Compiler geliefert
- Inhalt der Bibliothek ist festgeschrieben → Standard-Bibliothek von allen Herstellern gleich

- Beispiele für Bibliotheksfunktionen: Mathematische transzendente Funktionen `sin`, `cos`, `sqrt`, `exp`, `pow`, ...

## 2.5.2 Notation

- Alle (mathematischen) Bibliotheksfunktionen in gleicher Schreibweise. Syntax

```
mathfun ::= "Math." identifier "(" arglist ")".
arglist ::= expression ("," expression)*.
```

- Beispiel: Quadratwurzel aus 19

```
double x = Math.sqrt(19);
```

- [Siehe online-Referenz zur Klasse Math](#)

## 2.5.3 Argument und Ergebnis

- Jede Funktion schluckt ein (oder mehrere) *Argumente* = *Argumentliste*
- Argumente in Klammern, mit Kommata getrennt
- Beispiel: Berechnung von  $3^7$

```
double p = Math.pow(3, 7);
```

- Jeder Funktionsaufruf liefert *1 Ergebnis*
- Ein Funktionsaufruf ist ein *Ausdruck*, kann als Baustein in größeren Ausdrücken verwenden werden:

```
double d = 2*Math.pow(3, 16%9);
double e = Math.pow(Math.sin(0.7), Math.exp(2.25));
```

## 2.5.4 Signaturen

- Für Benutzer interessant:  
*Welche Argumente akzeptiert eine Bibliotheksfunktion, welchen Ergebnistyp liefert sie?*
- Typen der Argumente und des Ergebnisses = "*Signatur*" ("Abdruck")
- Schreibweise allgemein:  
*Name: Argumenttyp × Argumenttyp × ... → Ergebnistyp*
- Beispiele:

```
Math.sin : double → double
```

```
Math.abs : int → int
```

```
Math.pow : double × double → double
```

- Auch Operatoren lassen sich Signaturen zusprechen:

```
% : int × int → int
```

- Polymorphe Operatoren haben mehrere Signaturen: §

```
+ : double × double → double
```

```
+ : int × double → double
```

```
+ : double × int → double
```

```
+ : int × int → int
```

## 2.5.5 Laufzeiten

- Transzendente Funktionen sind *langsam*
- Nur dann benutzen, wenn unvermeidlich
- Beispiel: §

```
double x = Math.pow(4, 3); // langsam
double y = 4*4*4; // viel schneller
```

oder:

```
double x = Math.pow(4, 0.5); // langsam
double y = Math.sqrt(4) // etwas schneller
```

## 2.5.6 Dokumentation

- Sun Microsystems' JDK ist online dokumentiert
- Laufzeitbibliothek ist in "Packages" und "Klassen" gegliedert, die mathematischen Bibliotheksfunktionen stecken in der Klasse "java.lang.Math"
- Die Online-Dokumentation von "java.lang.Math" weist alle mathematischen Bibliotheksfunktionen aus

## 2.6 Fehler

### 2.6.1 Fehlerquellen

- Fehler sind unvermeidlich
- Peinlich berührtes Ignorieren hilft nicht weiter
- Fehler treten auf unterschiedlichen Ebenen auf: Syntax, statische Semantik, dynamische Semantik, Logik

### 2.6.2 Syntaxfehler

- Verstöße gegen die "Rechtschreibung", z.B.:

```
double 3.141592 = pi;
```

- Leicht zu finden: Compiler übersetzt nicht, bricht mit **Fehlermeldung** ab
- Fehlermeldungen verschiedener Compiler fallen verschieden aus. § Sun Microsystems' JDK-Compiler "javac" zählt zu den besten. §
- Problem: **Folgefehler** ("*Eine Fehlermeldung kommt selten alleine*")
- Pragmatische Lösung: Den **ersten gemeldeten** Fehler beheben, kurz darauf folgende Fehler (Zeilennummern!) zunächst ignorieren

---

## 2.6.3 Statische Semantik

- "**Statische Semantik**" = Teil der Semantik, der anhand einer (evtl. gründlichen) Untersuchung des Quelltextes überprüfbar ist.
- Beispiel:

```
final pi = 3.141592;  
pi = 2.3859; // pi ist final, nur 1 Zuweisung erlaubt!
```

- Die meisten Verstöße gegen die statische Semantik findet ein (guter) Compiler §

---

## 2.6.4 Dynamische Semantik

- "**Dynamische Semantik**" = Teil der Semantik, der erst beim Ablauf des Programms überprüfbar ist.
- Beispiel: Die Semantik von Java sagt:

*"Du sollst nicht durch Null teilen!"*

```
int x;  
/*  
...hier gibt den Benutzer den Wert von x an der Tastatur ein...  
*/  
int z = 1/(x - 17); // wenn "zufällig" 17 eingegeben wurde?
```

- Fehler in Syntax und statischer Semantik (vergleichsweise) harmlos: Compiler entdeckt sie ziemlich zuverlässig
- Fehler der dynamischen Semantik sind heikel

---

## Exkurs: Programmparameter

Viele Programme werden mit Parametern gesteuert. Die Parameter können auf verschiedenen Wegen an ein Programm übermittelt werden.

---

## 1 Konstanten im Quelltext

Die Parameter werden im Quelltext an bestimmte Variablen zugewiesen. Um das Programm mit anderen

Parameterwerten auszuprobieren, muß der Quelltext geändert und das Programm neu übersetzt werden. Das ist ziemlich umständlich und nur für die einfachsten Beispiele tragbar.

### Das Beispielprogramm

```
class Multiply
{
    public static void main(String[] args)
    {
        int m1 = 3;
        int m2 = 4;
        System.out.println(m1 + "x" + m2 + "=" + m1*m2);
    }
}
```

multipliziert zwei Zahlen (3 und 4) und gibt das Produkt aus.

```
$ java Multiply
3x4=12
```

## 2 Kommandozeilen–Argumente

Die Parameter können beim Programmstart auf der Kommandozeile angegeben werden, gleich hinter dem Programmnamen. Das Programm holt sich die Werte in den ersten Anweisungen und weist sie an passende Variablen zu.

In Java sind die Kommandozeilen–Argumente durchnummeriert. Das erste Argument (hinter dem Programmnamen) hat die Nummer ("*Index*") 0, das zweite den Index 1 usw. Um eine Zahl zu lesen, muß die Ziffernfolge in einen `int`–Wert umgewandelt werden. Das läßt sich erreichen mit einer Anweisung der Form

```
variable = Integer.parseInt(args[index]);
```

wobei *index* die Argumentnummer ist und *variable* irgendeine `int`–Variable.

### Das Beispielprogramm

```
class Multiply
{
    public static void main(String[] args)
    {
        int m1 = Integer.parseInt(args[0]);
        int m2 = Integer.parseInt(args[1]);
        System.out.println(m1 + "x" + m2 + "=" + m1*m2);
    }
}
```

akzeptiert zwei Zahlen von der Kommandozeile, multipliziert sie und gibt das Produkt aus. Es kann aufgerufen werden wie folgt:

```
$ java Multiply 3 4
3x4=12
```

## 3 Tastatureingabe

Das Programm kann an der Stelle, an der ein Parameterwert gebraucht wird, anhalten und auf eine Eingabe des Benutzers warten. Dieser tippt auf der Tastatur eine Ziffernfolge, die in einen `int`–Wert umgewandelt werden muß.

Das läßt sich in Java nicht mit einer einzigen Anweisung erledigen. Verwenden Sie statt dessen das Hilfsprogramm "Stdin". Übersetzen Sie das Programm mit

```
javac Stdin.java
```

In Ihren eigenen Programmen können Sie Stdin nun verwenden, um Tastatureingaben zu lesen:

```
variable = Stdin.readInt();
```

Ihr Programm hält an dieser Stelle an und wartet, bis der Benutzer eine Zahl eingetippt hat.

Leider kann weder Ihr Programm noch Stdin sicherstellen, daß niemand "Haha" eintippt anstelle einer Zahl. In diesem Fall kann das Programm nicht mehr sinnvoll fortgesetzt werden. Ergänzen Sie für diesen Fall die bunten Anweisungen

```
import java.io.*;

class Programmname
{
    public static void main(String[] args) throws IOException
    {
        ...
    }
}
```

Das Beispielprogramm

```
import java.io.*;

class Multiply
{
    public static void main(String[] args) throws IOException
    {
        System.out.println("Bitte ersten Faktor eingeben:");
        int m1 = Stdin.readInt();
        System.out.println("Bitte zweiten Faktor eingeben:");
        int m2 = Stdin.readInt();
        System.out.println(m1 + "x" + m2 + "=" + m1*m2);
    }
}
```

liest zwei Zahlen als Benutzereingaben, multipliziert sie und gibt das Produkt aus. Es kann aufgerufen werden wie folgt:

```
$ java Multiply
Bitte ersten Faktor eingeben:
3
Bitte zweiten Faktor eingeben:
4
3x4=12
```

Die Zahlen 3 und 4 gibt der Benutzer ein.

Wenn ein unkundiger Anwender eine unzulässige Eingabe liefert, bricht das Programm ab wie beabsichtigt:

```
$ java Multiply
Bitte ersten Faktor eingeben:
3
Bitte zweiten Faktor eingeben:
Haha
java.io.IOException
  at Stdin.nextNumber(Stdin.java:227)
  at Stdin.readInt(Stdin.java:142)
  at Multiply.main(Multiply.java:10)
```

## 3 Kontrollstrukturen

## 3.1 Sequenzen

- "**Kontrollstruktur**" allgemein = Sprachmittel zum Steuern der Ausführungs-Reihenfolge von Anweisungen §
- "**Sequenz**" = Einfachste Kontrollstruktur, nacheinander ausführen
- Syntax: Anweisungen hintereinander schreiben

```
sequence ::= statement*.
statement ::= assignment | definition.
```

## 3.2 Alternativen (if/else)

### 3.2.1 Idee

- Auch "Entscheidung", "Abzweigung", "*if*-Anweisung"
- Anweisung nur dann ausführen, wenn eine bestimmte Voraussetzung (= Bedingung) erfüllt ist, sonst auslassen
- Syntax schematisch

```
if(condition)
statement
```

- Semantik: Falls die Bedingung `condition` zutrifft, wird die Anweisung `statement` ausgeführt. Andernfalls wird sie ignoriert.
- Beispiel: Prozentangabe ausrechnen und auf den Bereich 0...100 begrenzen

```
int p;
// Wert von p einlesen

if(p > 100)
    p = 100;
if(p < 0)
    p = 0;
```

### 3.2.2 Bedingung

- Die "Bedingung" einer *if*-Anweisung ist eine **neue Sorte Ausdruck**
- Der Ausdruck "trifft zu" oder "trifft nicht zu"
- Der Wert ist kein Zahlenwert, sondern ein **Wahrheitswert**
- Ausdrücke mit Wahrheitswert heißen "**boole'sche Ausdrücke**" §
- Mehr zu boole'schen Ausdrücken später

### 3.2.3 Vergleichsoperatoren

- Auch "relationale Operatoren"
- In Java

<	echt kleiner
>	echt größer
<=	kleiner oder gleich
>=	größer oder gleich
==	gleich
!=	nicht gleich

- Weitere Sorte von Operatoren neben arithmetischen Operatoren
- Arithmetische Operatoren liefern Zahlen als Ergebnis, relationale Operatoren liefern Wahrheitswerte
- Relationale Operatoren sind polymorph, die sie können mit `int`- oder `double`-Werten arbeiten. Unterschiedliche Typen werden ggf. durch implizite Typkonversion angeglichen.

### 3.2.4 Beispiele für implizite Typkonversion

- `10*10 == 100`

```

                                10*10 == 100
int-Multiplikation 100 == 100
int-Vergleich      true

```

- `3 == 3.0`

```

                                3 == 3.0
int→double  3.0 == 3.0
double-Vergleich true

```

- `10/3 == 3.33333333333333`

```

                                10/3 == 3.33333333333333
int-Division  3 == 3.33333333333333
int→double  3.0 == 3.33333333333333
double-Vergleich false

```

- `10.0/3 == 3 + 1/3.0`

```

10.0/3 == 3 + 1/3.0

```

```

int→double    10.0/3.0      == 3 + 1/3.0
double-Division 3.333333333333 == 3 + 1/3.0
int→double    3.333333333333 == 3 + 1.0/3.0
double-Division 3.333333333333 == 3 + 0.333333333333
int→double    3.333333333333 == 3.0 + 0.333333333333
double-Addition 3.333333333333 == 3.333333333333
double-Vergleich true

```

### 3.2.5 Beispiele für if-Anweisungen

- Problem: Drei Werte gegeben, den größten auswählen

```

int a;
int b;
int c;
// Werte fuer a, b, c bestimmen

int max;

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

```

### 3.2.6 Zweiseitige Alternativen

- Erweiterung der einfachen ("einseitigen") if-Anweisung
- Syntax schematisch

```

if(condition)
    statement1
else
    statement2

```

- Semantik: Falls die Bedingung *condition* zutrifft, wie die Anweisung *statement1* ausgeführt und *statement2* ignoriert. Andernfalls wird *statement2* ausgeführt und *statement1* ignoriert. §
- Immer **genau eine** von beiden Anweisungen wird ausgeführt, aber
  - ◆ niemals beide,
  - ◆ niemals keine von beiden!
- Beispiel: Absolutwert (= Betrag) einer Zahl bestimmen

```

double x = ...;
double a;

```

```

if(x > 0)
    a = x; // x nicht-negativ
else
    a = -x; // x negativ

```

### 3.2.7 Geschachtelte Alternativen

- Eine if-Anweisung ist selbst eine Anweisung
- Unterscheidung...
  - "einfache Anweisung"
    - elementar, keine untergeordneten Anweisungen (Definition, Wertzuweisung)
  - "zusammengesetzte Anweisung"
    - enthält Unter-Anweisungen als Bestandteile (if-Anweisung)
- Folge: Eine if-Anweisung kann auch *untergeordnete* if-Anweisungen enthalten
- Beispiel: Signum-Funktion (positive Werte → 1, negative Werte → -1, Null → 0)

```

double x = ...;
int s;

if(x > 0)
    s = 1;
else
    if(x == 0)
        s = 0;
    else
        s = -1;

```

- Version *ohne* Schachtelung

```

double x = ...;
int s;

if(x > 0)
    s = 1;
if(x == 0)
    s = 0;
if(x < 0)
    s = -1;

```

Nachteil: prüft mehr als notwendig

### 3.2.8 Dangling Else

- Mischung aus ein- und zweiseitigen Alternativen erlaubt undurchsichtige Konstruktion:

```

if(condition1) if(condition2) statement1 else statement2

```

- Diese Situation heißt "dangling else".
- Zwei unterschiedliche Interpretationen: §

```

if(condition1)
    condition2
statement1
    else
statement2

```

oder...

```

if(condition1)
    condition2
statement1
else
statement2

```

- Java (und andere Programmiersprachen) ordnen ein `else` dem *letzten offenen if* zu (d.h. dem letzten `if` ohne `else`) ⇒ die erste Interpretation ist korrekt.
- Ausweg: Zweige in Alternativen immer als Block klammern.

### 3.2.9 if-Kaskade

- "if-Kaskade" = viele in Stufen geschachtelte if-Anweisungen
- Beispiel: Anzahl Tage eines gegebenen Monats (Januar = 1, Februar = 2, ..., Dezember = 12):

```

int monat;
int tage;
if(monat == 1)
    tage = 31;
else
    if(monat == 2)
        tage = 28;        // ignoriert Schaltjahre
    else
        if(monat == 3)
            tage = 31;
        else
            ...
            else
                if(monat == 11)
                    tage = 30;
                else
                    tage = 31;

```

- Einrückung anders arrangieren: besser lesbar

```

int monat;
int tage;
if(monat == 1)
    tage = 31;
else if(monat == 2)
    tage = 28;        // ignoriert Schaltjahre
else if(monat == 3)
    tage = 31;
...
else if(monat == 11)
    tage = 30;
else
    tage = 31;

```

(Es gibt auch elegantere Sprachmittel für derartig regelmäßig strukturierte if-Kaskaden.)

---

## ▶ 3.3 Wahrheitswerte

---

### ▶ 3.3.1 Datentyp `boolean`

- Die "Bedingung" einer `if`-Anweisung ist ein *Ausdruck* mit Wahrheitswert
  - Für Wahrheitswerte gibt es den Java-Typ "**`boolean`**" (gleichrangig neben `int` und `double`)
  - Es gibt nur *zwei Wahrheitswerte* ("trifft zu" und "trifft nicht zu")
  - Die entsprechenden `boolean`-Literele heißen  
`true` für "trifft zu" und  
`false` für "trifft nicht zu"
  - Allgemein: "*Literale*" sind wörtlich genannte, konstante Werte. "Numerale" sind demnach "Literale" für Zahlenwerte.
- 

### ▶ 3.3.2 Relationale Operatoren

- Jetzt ist klar: die weiter oben eingeführten relationalen Operatoren liefern als Ergebnisse `boolean`-Werte
- Beispiel: Es gilt  $10 < 20$ , deshalb  
`10 < 20 → true`
- "<" akzeptiert zwei `int`-Operanden, liefert `boolean`-Ergebnis.
- Signaturen von <:

```
double × double → boolean
int    × double → boolean
double × int    → boolean
int    × int    → boolean
```

Entsprechend die anderen relationalen Operatoren

---

### ▶ 3.3.3 Logische Operatoren

- Verknüpfen Wahrheitswerte, liefern neue Wahrheitswerte als Ergebnis
- Wichtigste logische Operatoren

```
&& logisches "Und"
||  logisches "Oder"
!   logisches "Nicht"
```

- Beispiel: Intervall prüfen

```
(x > 0) & (x < 10)
```

(lies: "(x ist größer als 0) und (x ist kleiner als 10)")

- Beispiel: Tage in einem Monat feststellen

```
if((m == 4) || (m == 6) || (m == 9) || (m == 11))
    tage = 30;
else if(m == 2)
    tage = 28;
else
    tage = 31;
```

- Beispiel: x ist kleiner als 10, aber nicht 7

```
if((x < 10) & !(x == 7))
    ...
```

- Signaturen logischer Operatoren:

```
&& boolean × boolean → boolean
|| boolean × boolean → boolean
! boolean → boolean
```

### 3.3.4 Wahrheitstabellen

- Kombinationsmöglichkeiten für Operanden der logischen Operatoren sehr begrenzt
- Beispiel "not" (unär):
  - !true → false
  - !false → true
- Für binäre logische Operatoren ("and", "or"): 2 Werte für den linken Operanden, 2 für den rechten Operanden ⇒  $2^2 = 4$  verschiedene mögliche Ergebnisse
- Logische Operatoren lassen sich bequem mit Tabellen definieren:

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

- Damit stellt sich die Frage, wie viele verschiedene binäre logische Operatoren es überhaupt geben kann. §

### 3.3.5 Blöcke als Anweisungsgruppe

- `if` kontrolliert nur **eine einzige** Anweisung
- Mehrere Anweisungen gruppieren mit `{...}` zu **einer Anweisung**
- Anweisungsfolge in `{...}` heißt "**Block**"
- Beispiel

```
int zaehler = ...;
int nenner = ...;

if(nenner != 0)
{
    quotient = zaehler/nenner;
    rest = zaehler%nenner;
}
```

- Sonderfall der **leeren Anweisung**: Semikolon alleine

```
;
```

- Sonderfall eines **leeren Blocks**: leere geschweifte Klammern

```
{}
```

- Die folgenden Fragmente sind gleichwertig:

```
if(...Bedingung...)
    ...Anweisung...

if(...Bedingung...)
    ...Anweisung...
else
    ;

if(...Bedingung...)
    ...Anweisung...
else
    {}

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

- Blöcke grenzen auch Gültigkeitsbereiche ab, siehe unten

### 3.3.6 Teilweise und vollständige Auswertung

- Oft ist der zweite von zwei aufeinanderfolgenden Tests nur dann sinnvoll, wenn der erste zutrifft.
- Beispiel: Division, *falls* der Nenner verschieden von Null:

```
if((d != 0) & (x/d > 238))
    ...
```

Fehler, wenn "wie ein arithmetischer Operator funktionieren würde (Division durch Null)!"

- Binäre logische Operatoren werden *anders abgewickelt*:
  1. Linken Operanden auswerten
  2. Falls das Ergebnis nicht feststeht: Rechten Operanden auswerten
 M.a.W.: der rechte Operand wird *nicht in jedem Fall* ausgewertet!
- Wann steht das Ergebnis fest?
 

*Erster Operand false*  $\Rightarrow$  *Ergebnis kann nur noch false lauten, unabhängig vom zweiten Operanden*

//

Erster Operand true  $\Rightarrow$  Ergebnis ist in jedem Fall true, unabhängig vom zweiten Operanden
- Binäre logische Operatoren "*werten teilweise aus*", arithmetische (und andere binäre) Operatoren "*werten vollständig aus*"

### 3.3.7 Operatortabelle

Die früher gezeigte Operatortabelle kann jetzt mit relationalen und logischen Operatoren vervollständigt werden.

Priorität	Operator	Wertigkeit	Assoziativität	Bemerkung
1	+	1	R→L	positives Vorzeichen
	-	1	R→L	negatives Vorzeichen
	!	1	R→L	logisches Not
2	*	2	L→R	Multiplikation
	/	2	L→R	Division
	%	2	L→R	Modulus = Divisionsrest
3	+	2	L→R	Addition
	-	2	L→R	Subtraktion
4	<	2	L→R	Vergleich echt kleiner
	<=	2	L→R	Vergleich kleiner oder gleich
	>	2	L→R	Vergleich echt größer
	>=	2	L→R	Vergleich größer oder gleich
5	==	2	L→R	Gleichheit
	!=	2	L→R	Nicht Gleichheit
6		2	L→R	logisches And (teilweise)
7		2	L→R	logisches Or (teilweise)
8	=	2	R→L	Wertzuweisung

### 3.3.8 Boole'sche Variablen

- boolean ist ein Typ wie int, double  $\Rightarrow$  Variablen definieren
- Beispiel:

```
boolean ok;
ok = true;
```

oder kuerzer

```
boolean ok = true;
```

- Boole'sche Ausdrücke nicht nur in Bedingungen, sondern als beliebige R-values:

```
boolean trash = x > 100;
if(trash)
    ...
```

- Oft aus Gewohnheit: unnötige Abfragen wie in:

```
boolean trash = x > 100;
if(trash == true)
    ...
```

(trash ist ein Wahrheitswert; der Vergleich mit true ist vollkommen überflüssig)

## 3.4 Schleifen (while)

### 3.4.1 while-Schleife

- "Schleife" allgemein: Konstrukt zur *wiederholten Ausführung* einer Anweisung
- Einfache Schleife in Java: "while"-Schleife §
- Syntax schematisch

```
while(condition)
    statement
```

*condition*: boole'scher Ausdruck

*statement*: beliebige Anweisung

- Semantik:
  1. Die Bedingung *condition* wird berechnet,
  2. Falls sie zutrifft (*true*), wird die Anweisung ausgeführt und zu Schritt 1 zurückgekehrt.
  3. Falls sie nicht zutrifft (*false*), ist die Schleife beendet und das Programm wird hinter der Schleife fortgesetzt.
- Die Anweisung heißt auch "*Schleifenrumpf*", der Vorsatz "*Schleifenkopf*"

### 3.4.2 Beispiel: Wertebereich abzählen

- Häufigste Aufgabe für Schleifen: Wertebereich schrittweise durchlaufen, dabei eine Anweisung wiederholen.
- Beispiel:

```
int loop = 0;
while(loop <10)
```

```

    loop = loop + 1;
// hier hat loop den Wert 10

```

- Beispiel erweitert um eine Ausgabe:

```

int loop = 0;
while(loop <10)
    loop = loop + 1;
    System.out.println(loop);

```

Das Programm druckt die Meldung *nur einmal aus*, weil die Ausgabe nicht im Schleifenrumpf steht.

- Korrigierte Fassung:

```

int loop = 0;
while(loop <10)
{
    loop = loop + 1;
    System.out.println(loop);
}

```

liefert die Ausgabe

```

1
2
3
...
9
10

```

### 3.4.3 Beispiel: Zahlensumme

- Das Beispiel vom Beginn aus der ersten Vorlesung sollte jetzt Sinn geben!

### 3.4.4 Beispiel: Collatzfolge

- Eine *Collatzfolge* ist eine Folge ganzer Zahlen  $n_0, n_1, n_2, \dots$
- Die Zahlenfolge ist induktiv definiert:
  1. Der Startwert  $n_0$  ist beliebig, aber fest vorgegeben
  2. Für das Element  $n_{i+1}$  ( $i > 0$ ) gilt

$$n_{i+1} = \begin{array}{ll} \frac{1}{2} \cdot n_i & \text{für gerade } n_i \\ 3 \cdot n_i + 1 & \text{für ungerade } n_i \end{array}$$

§

- Zur Startzahl  $n_0 = 6$  ergibt sich beispielsweise die Folge 6, 3, 10, 5, 16, 8, 4, 2, 1, 4, ...
- 4, 2, 1 ist offenbar ein Zyklus, der sich endlos wiederholt.
- Die Folge hat interessante Eigenschaften:
  1. Es sieht so aus, als würde *jede Folge* über kurz oder lang in einen Zyklus führen. M.a.W. es gibt keine endlos wachsende Folge.
  2. Es sieht so aus, als wäre 4, 2, 1 der *einzigste Zyklus*.
  3. Die Länge einer Folge ist i. allg. nur durch Ausprobieren zu ermitteln.
  4. Das Maximum einer Folge ist i. allg. nur durch Ausprobieren zu ermitteln.

- Die Collatzfolgen haben *chaotische* Eigenschaften: Minimale Variationen in den Ausgangsbedingungen haben beliebig weitreichende Auswirkungen auf das Ergebnis §
  - Beispielprogramm zum Berechnen der Collatzfolge.
  - Beispielprogramm zum Berechnen der Länge einer Collatzfolge.
  - Beispielprogramm zum Berechnen von Länge und Maximalwert einer Collatzfolge.
- 

### ▶ 3.4.5 Operatorzuweisungen

- Binärer Ausdruck

```
x = x op y
```

kürzer als

```
x op= y
```

- `op=` heißt "Operatorzuweisung"
  - Formal zweistellige Operatoren mit Priorität, Assoziativität wie Wertzuweisung
  - Semantisch gleichwertig, aber oft besser zu lesen
  - Beispielprogramm
- 

### ▶ 3.4.6 Inkrement und Dekrement

- Häufig gebrauchtes Hochzählen (Herunterzählen) um 1 mit Operatorzuweisung als

```
x += 1;  
y -= 1;
```

- Weitere Abkürzung mit Postfix-Operatoren "++" und "--": §

```
x++;  
y--;
```

- Die folgenden drei Anweisungen sind gleichwertig:

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

- "++" und "--" heißen "Inkrement-Operator" und "Dekrement-Operator".
  - Beispielprogramm
- 

### ▶ 3.4.7 Bedingter Operator

- Dreistelliger "bedingter Operator" mit zwei Operatorenymbolen:

```
condition ? expression1 : expression2
```

- Abwicklung etwa vergleichbar mit der zweiseitigen Alternative:

```
var = condition ? expression1 : expression2;
```

ist äquivalent zu

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

- *condition* ist ein boole'scher Ausdruck, *expression<sub>1</sub>* und *expression<sub>2</sub>* sind Ausdrücke beliebigen, aber gleichen Typs.
- Der bedingte Operator wertet *teilweise* aus (wie die logischen Operatoren und `||`):  
*condition = true:*  
*expression<sub>1</sub>* auswerten und Ergebnis zurückliefern, *expression<sub>2</sub>* ignorieren  
*condition = false:*  
 umgekehrt.
- Einer der beiden Ausdrücke wird in jedem Fall *ignoriert*, der andere in jedem Fall *ausgewertet*
- Beispielprogramm

### ▶ 3.4.8 Endlosschleifen

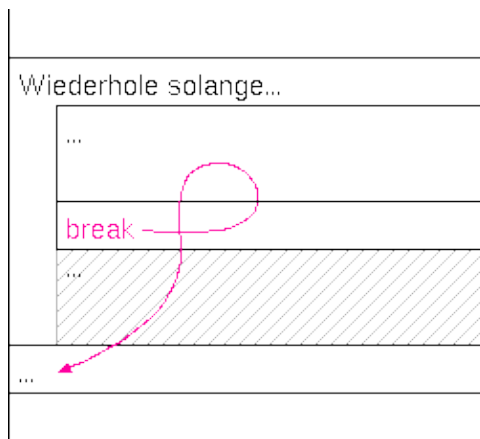
- Schleifen können endlos laufen ("Endlosschleife")
- Simplex Beispiel: §

```
while(true)
    ;
```

- Es ist wichtig, die Terminierung von Schleifen sicherzustellen. Das kann auf formalen Wege geschehen § oder auf informellem Wege.
- Oft ausreichend: Ausdruck im Schleifenrumpf finden, der sich mit jedem Durchlauf streng monoton (und nachweislich) zum Abbruchkriterium hin verändert.

### ▶ 3.4.9 Schleifenabbruch mit `break`

- Anweisung `break` (wie in einem Switch) im Schleifenrumpf beendet die Schleife sofort:

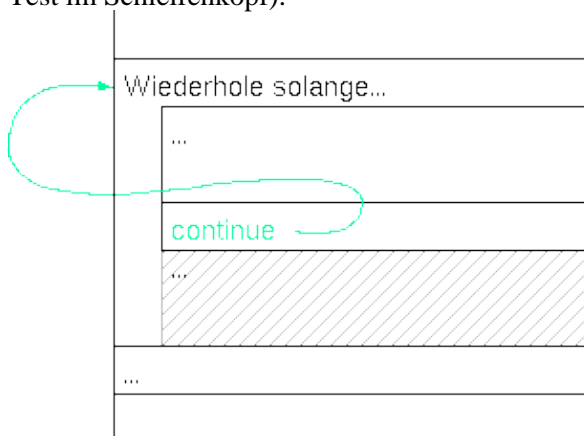


- Beispiel: Berechnung des GGT mit Euklids Algorithmus:

```
while(true)      // Schleife wird mit break beendet
{
    int r = a%b;
    if(r == 0)
        break;
    a = b;
    b = r;
}
```

### 3.4.10 Schleifenkurzschluß mit continue

- Anweisung `continue` im Schleifenrumpf startet sofort den nächsten Schleifendurchgang (mit dem Test im Schleifenkopf):



- Beispiel: Zahlenfolge einlesen und
  - ◆ positive Werte aufaddieren,
  - ◆ negative Werte ignorieren,
  - ◆ abbrechen bei Eingabe der Null.

```
import java.io.*;

class SumInput
{
    public static void main(String[] args) throws IOException
```

```

    {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        int sum = 0;
        while(true)
        {
            int n = Integer.parseInt(br.readLine());
            if(n == 0)
                break;
            if(n
<0)            if(n
                continue;
                sum += n;
            }
            System.out.println("Summe = " + sum);
        }
    }

```

### 3.4.11 Geschachtelte Schleifen

- While-Schleife = zusammengesetzte Anweisung
- Schleifen als Bausteine anderer Kontrollstrukturen, geschachtelte Schleifen
- Beispiel: Großes 1x1 ausgeben

```

int x = 1;
while(x <= 10)
{
    int y = 10;
    while(y
    {
        System.out.println(x + "*" + y + " = " + x*y);
        y++;
    }
    System.out.println();
    x++;
}

```

(Wie oft wird der innere Schleifenrumpf durchlaufen?)

- Vorsicht: Rumpf einer inneren Schleife wird evtl. *sehr oft ausgeführt* ("Hot spot") ⇒ marginale Performanceunterschiede eskalieren!
- Volksmund:  
*90% der Laufzeit werden in 10% des Quelltextes verbracht.*
- Beispielprogramm um Collatzfolge mit größter Länge und mit größtem Maximum zu suchen.

## 3.5 Blöcke

### 3.5.1 Gültigkeitsbereiche

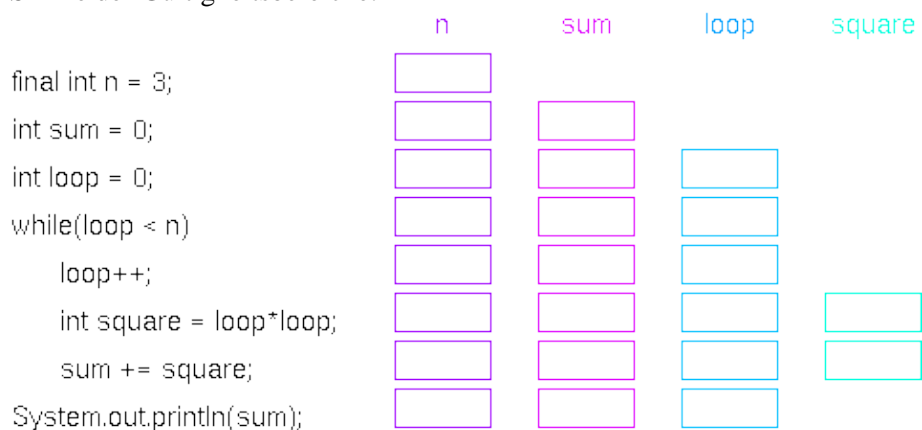
- Anweisungen in {...} gruppieren ⇒ "Block"
- In einem Block sind *alle Arten von Anweisungen* erlaubt, auch Definitionen
- Beispiel:

```
while(...)
{
    int temp = ...;
    ...
}
```

- Eine Variable gilt ab der Definition bis zum Ende des Blocks, aber *nicht mehr außerhalb*
- "**Gültigkeitsbereich**" einer Variablen = Quelltextbereich zwischen Definition und Block-Ende
- Beispiel:

```
final int n = ...;
int sum = 0;
int loop = 0;
while(loop < n)
{
    loop++;
    int square = loop*loop;
    sum += square;
}
System.out.println(sum);
```

- Skizze der Gültigkeitsbereiche:



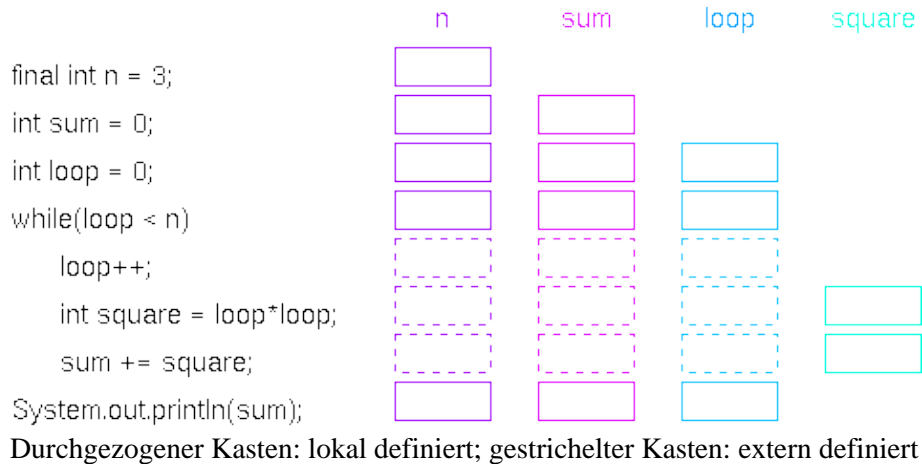
## 3.5.2 Lokale und externe Namen

- Eine Variable heißt "*lokal*" im Block ihrer Definition.
- Eine Variable gilt auch in allen Blöcken, die im Definitionsbereich geschachtelt sind.
- In diesen geschachtelten Blöcken heißt die Variable "*extern*".
- Beispiel:

```
final int n = ...;
int sum = 0;
int loop = 0;
while(loop < n)
{
    loop++;
    int square = loop*loop;
    sum += square;
}
```

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

- Skizze der Gültigkeitsbereiche:



### 3.5.3 Namenskonflikte

- Eine externe Variable darf nicht (mit gleichen) Namen lokal definiert werden. §
- An jedem Punkt im Quelltext *gilt genau eine Definition*.
- Beispiel: Nicht zulässig ist...

```
int tmp = ...;
while(...)
{
    int tmp = ...; // Fehler!
    ...
}
```

Dagegen wäre erlaubt:

```
while(...)
{
    int tmp = ...; // Ok
    ...
}
while(...)
{
    int tmp = ...; // Ok
    ...
}
```

Die beiden tmp-Variablen haben den gleichen Namen, sind aber verschiedene Objekte!

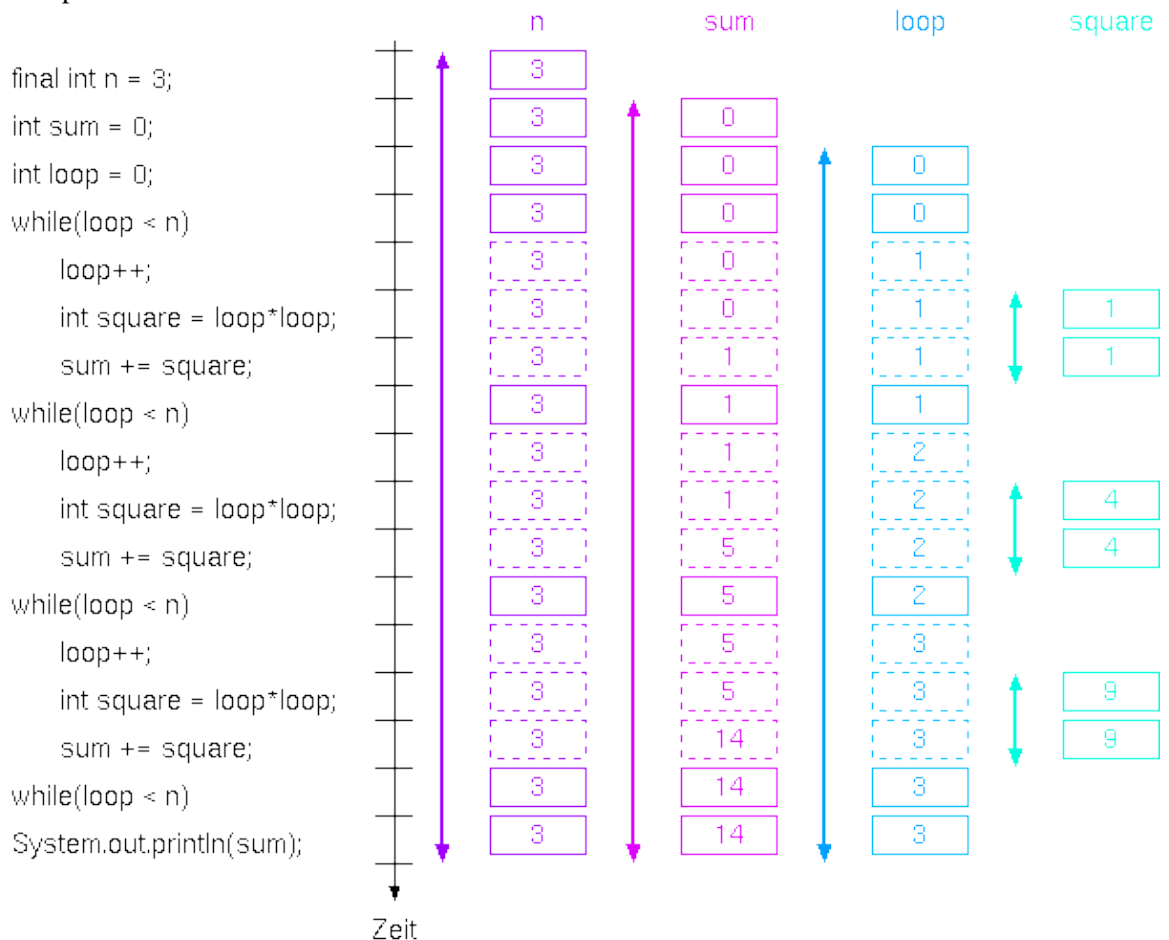
### 3.5.4 Existenzbereiche

- Der "*Existenzbereich*" einer Variablen ist das Zeitintervall, in dem die Variable existiert.
- Begriffe unterscheiden:
  - Gültigkeitsbereich*  
statisch, auf den Quelltext bezogen
  - Existenzbereich*  
dynamisch, auf die Laufzeit bezogen

- Eine Variable kann (zur Laufzeit) wiederholt geschaffen und zerstört werden:

```
final int n = ...;
int sum = 0;
int loop = 0;
while(loop < n)
{
    loop++;
    int square = loop*loop;
    sum += square;
}
System.out.println(sum);
```

- Beispiel:



(durchgezogene Kästen: lokal definierte Variablen; gestrichelte Kästen: extern definierte Variablen)

- Die aufeinanderfolgenden "Inkarnationen" der Variablen haben *nichts miteinander zu tun* §

## 3.6 Zählschleifen (for)

### 3.6.1 Syntax und Semantik

- Kurzform zum Durchzählen eines Wertebereichs mit einer Zählvariablen
- Syntax schematisch:

```
for(start; condition; next)
```

```
statement
```

- Semantik
  1. Beim Eintritt wird 1× die Anweisung *start* ausgeführt.
  2. Die Bedingung *condition* wird ausgewertet...
  3. Falls *condition* == true...
    - a. Die Anweisung *statement* wird ausgeführt und
    - b. Die Anweisung *next* wird ausgeführt
    - c. Zurück zu 2.)
  4. Falls *condition* == false ist die Schleife beendet.
- *start*, *condition* und *next* sind optional:
  - start fehlt*:  
Schleife beginnt sofort mit dem Test von *condition*
  - condition fehlt*:  
Wird behandelt wie true
  - next fehlt*:  
Am Ende des Schleifenrumpfes wird sofort wieder *condition* geprüft

### 3.6.2 Beispiele

- Zahlensumme (siehe Einführungsbeispiel)

```
int n = 4;
int ergebnis = 0;
for(int zähler = 1; zähler <= n; zähler++)
    ergebnis += zähler;
System.out.println(ergebnis);
```

- Berechnung der Collatzfolge

```
for(int n = Integer.parseInt(args[0]); n != 1; n = (n%2 == 0)? n/2: 3*n + 1)
    System.out.println(n);
```

- Variation des Euklidischen GGT-Algorithmus:

```
for(int r = a%b; r != 0; r = a%b)
{
    a = b;
    b = r;
}
```

### 3.6.3 Gegenüberstellung mit while-Schleifen

- Die for-Schleife

```
for(start; condition; next)
    statement
```

ist (aus Sicht der Abwicklung) zu folgender while-Schleife äquivalent:

```
start;
while(condition)
{
    statement
```

```
next
}
```

- Es gibt aber einige Unterschiede, insbesondere der Gültigkeitsbereich einer Definition in *start*:  
*for*

Innerhalb des Rumpfes

*while*

Außerhalb des Rumpfes, wie es die obige Gegenüberstellung nahelegt

- Praktische Konsequenz: Die beiden folgenden Schleifen sind korrekt, obwohl **dieselbe Zählvariable** verwendet wird:

```
int fakt = 1;
for(int loop = 1; loop <n/2; loop++)
    fakt *= loop;
for(int loop = n/2; loop <= n; loop++)
    fakt *= loop;
```

§

## 3.7 Verteiler (switch)

### 3.7.1 Ersatz für if-Kaskade

- Kompakte Form einer if-Kaskade
- Beispiel: Bestimmen der Tage eines Monats (frühere Fassung) formuliert als Verteiler:

```
switch(m)
{
    case 1:
        t = 31;
        break;

    case 2:
        t = 28;
        break;

    case 3:
        ...

    case 12:
        t = 31;
        break;
}
```

### 3.7.2 Syntax und Semantik

- Syntax schematisch

```
switch(expression)
{
    case constant1:
        statement11
        statement12...
        break;
```

```

    case constant2:
statement21
statement22...
        break;
    case
    ...
    default:
statementx1
statementx2...
        break;
}

```

- Ein Zweig mit `case` und nachfolgenden Statements heißt "*Case-Klausel*"
- Semantik:
  1. Der Ausdruck `expression` wird ausgewertet und sein Ergebnis in der Liste der `case`-Werte gesucht.
  2. Falls gefunden werden die Statements der `Case-Klausel` ausgeführt bis zum nächsten `break`.
  3. Ansonsten wird der `default`-Zweig gesucht und dessen Statements ausgeführt.

### ▶ 3.7.3 Reihenfolge der `case`-Klauseln

- Allgemein: Die Reihenfolge der `Case-Klauseln` *beliebig*
- Ein Compiler kann sie z.T. neu ordnen
- Mehrere `Case`-Werte pro Klausel zulässig (Sonderfall: einige Klauseln *ohne* Statements)
- Beispiel:

```

switch(m)
{
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
        t = 31;
        break;

    case 2:
        t = 28;
        break;

    case 4:
    case 6:
    case 9:
    case 11:
        t = 30;
        break;
}

```

### ▶ 3.7.4 Duplikate und Lücken

- *Mehrere gleiche Case-Werte* verboten – Fehler

- Auswahl der Case–Werte ansonsten beliebig, auch mit Lücken:

```
switch(m)
{
    case 1:
    case 4:
    case 7:
        System.out.println("gerade!");
        break;
}
```

- Falls der Switch–Ausdruck zu **keinem Case–Wert paßt**: (und keine Default–Klausel vorliegt): Switch ist wirkungslos §

### 3.7.5 Defaultfall

- Spezieller Casewert "default" – paßt auf jeden vorher nicht genannten Wert
- Muß die letztgenannte Case–Klausel sein
- Beispiel:

```
switch(note)
{
    case 1:
    ...
    default:
        System.out.println("System bored-- Please replace user");
        break;
}
```

- Entspricht dem schließenden "else" einer if–Kaskade

### 3.7.6 Fall through

- Anweisung "break;" (üblicherweise) am Ende jeder Klausel
- Falls fehlt: Programm wird mit den Anweisungen der *nachfolgenden Case–Klausel* fortgesetzt! ("Fall through") §
- Selten sinnvoll ⇒ break in der Praxis immer angeben

### 3.7.7 Einschränkungen

- Switch–Ausdruck und Case–Werte müssen Typ `int` haben § (insbesondere sind `double`, `boolean` unzulässig)
- Case–Ausdrücke müssen **vom Compiler berechenbar** sein ("konstante Ausdrücke")
- Beispiel:

```
case 23*7%17: // ok
    ...
case a*b/c: // Fehler!
```

...

---

### ▶ 3.7.8 Switch als Anweisung

- Ein kompletter Switch ist eine (zusammengesetzte) Anweisung
- Switches lassen sich bspweise schachteln oder als Bausteine in andere zusammengesetzte Anweisungen einbauen
- Vorsicht mit Definitionen *innerhalb* von Switches. Grundsätzlich möglich, aber von merkwürdigen Regeln beherrscht ⇒ vermeiden!

---

## ▶ 3.8 Algorithmen

### ▶ 3.8.1 Merkmale

- Zweck: Kommunikation einer Idee an andere Menschen
- Entwurfsmittel
- Vorgaben und Ergebnis exakt definiert
- Einzelschritte effizient, eindeutig

---

### ▶ 3.8.2 Darstellungsformen

- Darstellung frei, oft informell, **unabhängig** von konkreten Programmiersprachen
- Keine formale Syntax
- Beispiele:
  - ◆ Umgangssprachlicher Fließtext
  - ◆ Nummerierte Punkteliste
  - ◆ Flußdiagramme
  - ◆ Struktogramme

---

### ▶ 3.8.3 Nummerierte Punkteliste

- Folgen numerierter Schritte, legen Reihenfolge fest
  - Alltagsbeispiel: Kochrezept
  - Gruppieren durch Einrücken, neues Nummernschema
  - Einfache Steuerungsanweisungen
-

### 3.8.4 Beispiele

- Beispiel Zahlensumme
- Berechnen des "größten gemeinsamen Teilers" zweier positiver, ganzer Zahlen  $a$  und  $b$
- Erfunden im Altertum, bekannt als "Euklid'scher ggT-Algorithmus"
- Beschreibung
  - 1.) Lege  $a$  und  $b$  fest
  - 2.) Teile  $a$  durch  $b$ , der Rest sei  $r$
  - 3.) Falls  $r = 0$ , dann weiter mit Schritt 7
  - 4.) Gib  $a$  den Wert von  $b$
  - 5.) Gib  $b$  den Wert von  $r$
  - 6.) Fahre fort mit Schritt 2
  - 7.) Die Antwort ist  $b$
- Ablaufbeispiel:

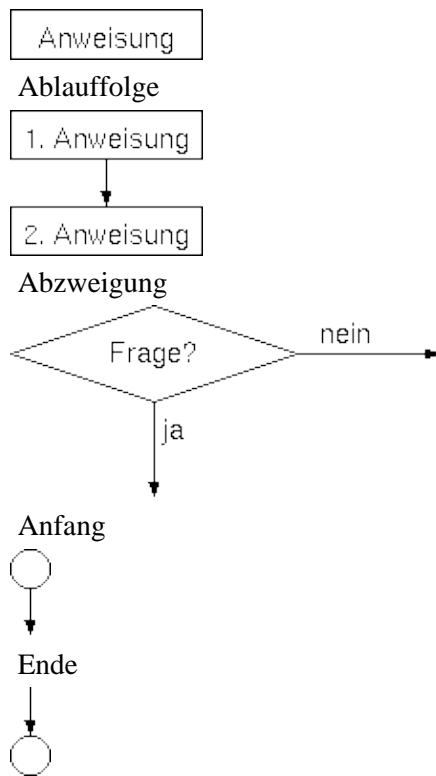
<i>Schritt</i>	<i>a</i>	<i>b</i>	<i>r</i>	
1	72	40	?	
2	72	40	32	
3	72	40	32	
4	40	40	32	
5	40	32	32	
6	40	32	32	
2	40	32	8	
3	40	32	8	
4	32	32	8	
5	32	8	8	
6	32	8	8	
2	32	8	0	
3	32	8	0	
7	32	8	0	Die Antwort ist 8

---

### 3.8.5 Flußdiagramme

- Darstellungselemente

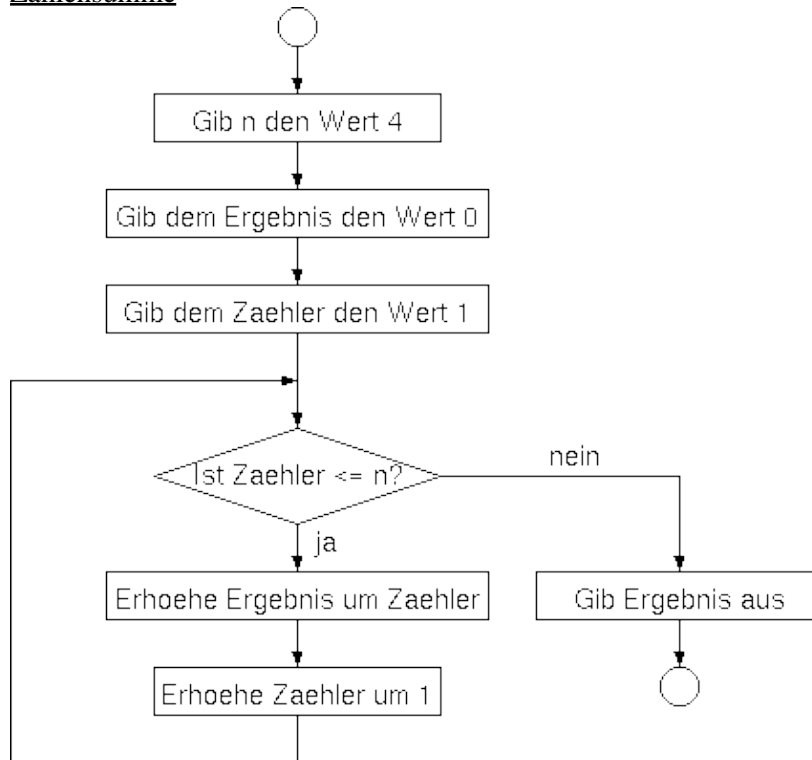
Elementare Anweisung
----------------------



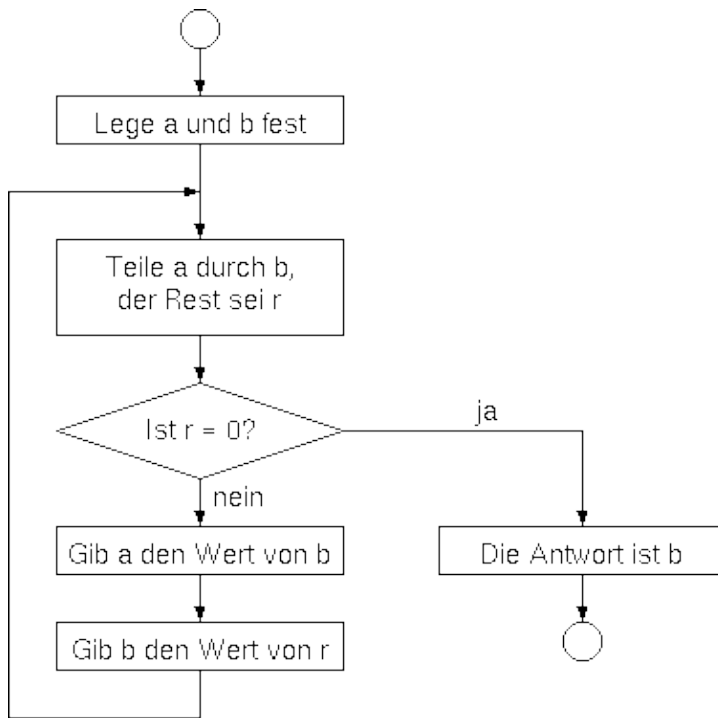
- Vorteil: viele Freiheiten bzgl. Kontrollfluß
- Nachteil: keine Ebenen, *zuviel* Freiheiten bzgl. Kontrollfluß

### 3.8.6 Beispiele Flußdiagramme

- Zahlensumme

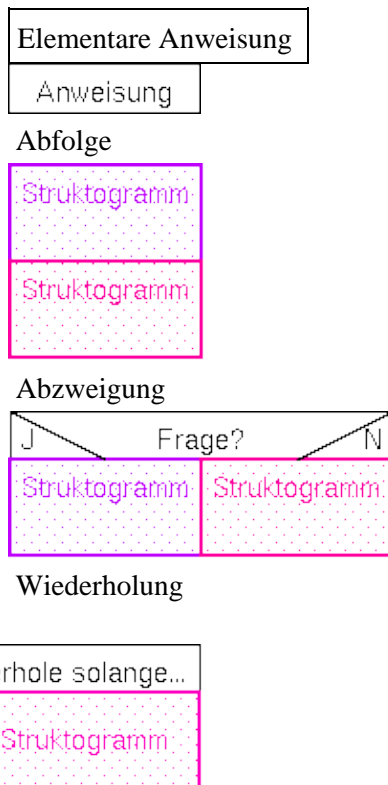


- Euklid's GGT-Algorithmus



### 3.8.7 Struktogramme

- Darstellungselemente
- Vorteil: Läßt sich leicht zerlegen, zusammenbauen



### 3.8.8 Beispiele Struktogramme

- **Beispiel: Zahlensumme**

Gib n den Wert 4
Gib Zaehler den Wert 1
Gib Summe den Wert 0
Wiederhole solange Zaehler <= n...
Erhoehe Summe um Zaehler
Erhoehe Zaehler um 1
Gib Ergebnis aus

- **Euklid's GGT-Algorithmus**

Lege a und b fest
Teile a durch b, der Rest sei r
Wiederhole bis r = 0...
Gib a den Wert von b
Gib b den Wert von r
Teile a durch b, der Rest sei r
Die Antwort ist b

## ▶ 4 Textzeichen und Strings

### ▶ 4.1 Textzeichen

#### ▶ 4.1.1 Javatypp char

- "char" = Javatypp für einzelne **Textzeichen** (z.B. wie auf einer Tastatur)
- **Primitiver, vordefinierter Typ**, gleichrangig mit int, double, boolean
- Char-Literale: wörtlich genannte **Zeichen in Quotes**, wie z.B.

'a' kleiner Buchstabe a

'5' Ziffer 5

'%' Prozent-Zeichen

' ' Leerzeichen ("Blank")

#### ▶ 4.1.2 Operationen mit char

- Variablendefinitionen:

```
char c;
char letter = 'A';
```

- Zuweisung

```
char x = letter;
x = '!';
```

- Prüfung auf Gleichheit, Ungleichheit

```
char five = '5';
if(five == 'V')
    ...
if(five != 'X')
    ...
```

- Größenvergleich bspweise für Buchstaben alphabetisch, für beliebige andere Zeichen gemäß Zeichencode

```
char letter = 'A';
if(letter < 'P')
    ...
```

### 4.1.3 Zeichencodes

- Mit char–Werten kann *gerechnet* werden, wie mit int–Werten.
- Ein Zeichen wird dabei *durch seinen Code* vertreten.
- Zeichencodes sind allgemein vereinbart, z.B.:

Zeichen	Code
'a'	97
'5'	53
'%'	37
' '	32
...	...

- Ein char kann als int benutzt werden (implizite Typkonversion). Das Zeichen wird dabei durch seinen Code ersetzt.
- Beispiel:

```
int i;
i = 'a';           // i hat den Wert 97
i = 2*'b' + 1;    // i hat den Wert 2*98 + 1 = 197
```

- Ein int kann *nicht* direkt als char benutzt werden. Dazu Typecast erforderlich (explizite Typkonversion)
- Beispiel:

```
char c;
c = (char)97;      // c hat den Wert 'a'
c = (char>('a' + 1)); // c hat den Wert 'b'
```

## 4.1.4 Zeichensätze

- Eine komplette Sammlung von Zeichen samt Codes heißt **Zeichensatz**.
- Früher gab es mehrere verschiedene Zeichensätze. Heute hat sich "ISO-8859-1" (auch: "ISO-Latin-1") durchgesetzt:

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00																
10																
20		!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
70	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
80																
90																
A0		ı	ç	£	¤	¥	¦	§	¨	©	<sup>a</sup>	«	¬	-	®	¯
	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B0	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C0	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D0	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E0	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F0	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ
	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

- Java benutzt "Unicode".
- Unicode umfaßt  $2^{16} = 65536$  Zeichen. Die ersten 256 Zeichen im Unicode decken sich mit ISO-Latin-1.

## 4.1.5 Hexadezimalschreibweise

- Nicht alle Zeichen des Unicodes sind auf jedem System wörtlich darstellbar. §
- Um **nicht darstellbare Zeichen** zu benennen, wird ihr Code in hexadezimaler Schreibweise

angegeben:

```
\uXXXX
```

wobei *XXXX* der hexadezimale Code ist.

- Beispiel: Die folgenden drei Wertzuweisungen sind gleichwertig:

```
a = 97;
a = 'a';
a = '\u0061';
```

## 4.1.6 Ersatzdarstellungen

- "**Kontrollzeichen**" = nicht druckbare Zeichen mit traditioneller Bedeutung
- Für Kontrollzeichen gibt es kompakte **Ersatzdarstellungen**

Code	Ersatzdarstellung	Unicode	Bezeichnung	Bedeutung
8	<code>\b</code>	<code>\u0008</code>	backspace (BS)	vorhergehendes Zeichen wegnehmen
9	<code>\t</code>	<code>\u0009</code>	horizontal tab (HT)	Blanks bis zur nächsten Tabulatorposition
10	<code>\n</code>	<code>\u000a</code>	linefeed (LF)	neue Zeile anfangen
12	<code>\f</code>	<code>\u000c</code>	form feed (FF)	neue Seite anfangen
13	<code>\r</code>	<code>\u000d</code>	carriage return (CR)	zurück zum Zeilenanfang
34	<code>\"</code>	<code>\u0022</code>	double quote	Gänsefüßchen
39	<code>\'</code>	<code>\u0027</code>	single quote	Hochkomma
92	<code>\\</code>	<code>\u005c</code>	backslash	Rückwärts-Schrägstrich

- Die letzten drei Ersatzdarstellungen lösen das syntaktische Problem der expliziten Angabe von Javabegrenzern. Z.B. ist `'\''` das Zeichen "Hochkomma".

## 4.1.7 Bibliotheksmethoden

- Häufig gebrauchte Character-Methoden:
  - `boolean isLetter(char ch)`  
`boolean isDigit(char ch)`  
Gibt Auskunft ob das Zeichen *ch* ein Buchstabe (eine Ziffer) ist.
  - `boolean isWhitespace(char ch)`  
Gibt Auskunft ob das Zeichen *ch* ein Zwischenraumzeichen ist.
  - `boolean isLowerCase(char ch)`  
`boolean isUpperCase(char ch)`  
Gibt Auskunft ob das Zeichen *ch* ein kleiner (großer) Buchstabe ist.
  - `char toLowerCase(char ch)`  
`char toUpperCase(char ch)`  
Liefert den entsprechenden kleinen (großen) Buchstaben zu *ch*, oder *ch* selbst wenn es kein Buchstabe ist.
- Zeichenklassifizierung orientiert sich am Unicode. Nationale Umlaute, wie das deutsche ä, ö, ü, ..., werden korrekt als Buchstaben erkannt (`isLetter`, ...) und behandelt (`toLowerCase('Ä')` →

'ä', ...)

- Weitere Bibliotheksmethoden zum Umgang mit Textzeichen.

## Exkurs: Text-I/O

### 1 Ein- und Ausgabe

- **"Konsole"** = Tastatur (Eingabe) + Bildschirm (Ausgabe)
- **Ausgabe auf Konsole:** Methode `System.out.println`, wie z.B.:

```
System.out.println(23*2);
System.out.println(Math.sqrt(19));
System.out.println('X');
```

- Zeichenweise **Eingabe von der Konsole** über `Mittlerobjekt`:

```
Reader r = new InputStreamReader(System.in);
r.read()...
```

- `read()` liefert den Code des nächsten Zeichens oder `-1`, wenn nichts mehr kommt.
- **Eingabe von einer Textdatei** über anderes `Mittlerobjekt`:

```
Reader r = new FileReader("dateiname");
r.read()...
```

### 2 Beispiel: Lesen eines Textes

- Liest Text von der Konsole und absorbiert ihn.

```
import java.io.*;

public class TextReader
{
    public static void main(String[] args) throws IOException
    {
        Reader r = new InputStreamReader(System.in);
        for(int code = r.read(); code >= 0; code = r.read())
            ;
    }
}
```

- Aufrufbeispiel

```
$ java TextReader
Hallo, world!
Anybody out there?
^d
$
```

^d steht für Control-d, unter Unix das Signal für das Ende einer Texteingabe von der Konsole. In einer DOS-Box von MS-Windows erfüllt ^z den gleichen Zweck.

### 3 Beispiel: Text kopieren

- Liest Text von der Konsole und gibt ihn unverändert wieder aus

```
import java.io.*;
public class TextCopy
{
    public static void main(String[] args) throws IOException
    {
        Reader r = new InputStreamReader(System.in);
        for(int code = r.read(); code >= 0; code = r.read())
        {
            char ch = (char)code;
            System.out.print(code);
        }
    }
}
```

- Aufrufbeispiel:

```
$ java TextCopy
Hallo, world!
Anybody out there?
^d
Hallo, world!
Anybody out there?
$
```

### 4 Beispiel: Zeichen und Zeilen zählen

- Liest Text von der Konsole, zählt die Zeichen und Zeilen, gibt die Summen aus

```
import java.io.*;
public class TextCounter
{
    public static void main(String[] args) throws IOException
    {
        Reader r = new InputStreamReader(System.in);
        int lines = 0;
        int chars = 0;
        for(int code = r.read(); code >= 0; code = r.read())
        {
            char ch = (char)code;
            chars++;
            if(ch == '\n')
                lines++;
        }
        System.out.println(lines);
        System.out.println(chars);
    }
}
```

- Aufrufbeispiel:

```
$ java TextCounter
```

```
Hallo, world!
Anybody out there?
^d
2
33
$
```

## 5 Beispiel: Zwischenraum löschen

- Liest Text von der Konsole, löscht alle Zwischenraumzeichen und gibt den Rest wieder aus

```
import java.io.*;
public class SpaceX
{
    public static void main(String[] args) throws IOException
    {
        Reader r = new InputStreamReader(System.in);
        for(int code = r.read(); code >= 0; code = r.read())
        {
            char ch = (char)code;
            if(!Character.isWhitespace(ch))
                System.out.print(ch);
        }
    }
}
```

- Aufrufbeispiel:

```
$ java SpaceX
Hallo, world!
Anybody out there?
^d
Hallo,world!Anybodyouthere?$
```

§

## 6 Beispiel: Textdatei lesen

- Liest Text von einer Datei und gibt ihn unverändert auf der Konsole aus.

```
import java.io.*;
public class FileRead
{
    public static void main(String[] args) throws IOException
    {
        Reader r = new FileReader(args[0]);
        for(int code = r.read(); code >= 0; code = r.read())
        {
            char ch = (char)code;
            System.out.print(ch);
        }
    }
}
```

- Aufrufbeispiel

```
$ java FileRead FileRead.java
```

```
import java.io.*;
public class FileRead
{
    public static void main(String[] args) throws IOException
    {
        Reader r = new FileReader(args[0]);
        for(int code = r.read(); code >= 0; code = r.read())
        {
            char ch = (char)code;
            System.out.print(ch);
        }
    }
}
$
```

## 7 Beispiel: Textdatei schreiben

- Liest Text von der Konsole und schreibt ihn unverändert in eine Textdatei.

```
import java.io.*;
public class FileWrite
{
    public static void main(String[] args) throws IOException
    {
        Writer w = new FileWriter(args[0]);
        Reader r = new InputStreamReader(System.in);
        for(int code = r.read(); code >= 0; code = r.read())
        {
            char ch = (char)code;
            w.write(ch);
        }
        w.close();
    }
}
$
```

- Aufrufbeispiel (in der Datei `textfile` stehen anschließend die beiden eingetippten Zeilen)

```
$ java FileWrite textfile
Hallo, world!
Anybody out there?
^d
$
```

## 4.2 Zeichenketten

### 4.2.1 Motivation

- **"String"** = Textstück = lineare Folge von Textzeichen
- In Java: Datentyp "String" (vordefiniert wie `int`, `double`, `boolean`, `char`)
- String ist anders als `int`, `double`, `boolean`, `char`: **Zusammengesetzter Typ** = "Containertyp"
- String = **Behälter** für darin enthaltene **char-Elemente**
- Beispiel: String "Java"

'J'	'a'	'v'	'a'
-----	-----	-----	-----

## 4.2.2 Literale

- String-Literale = **Klartext zwischen Begrenzern** (= Gänsefüßchen). Beispiele:

```
"Java"
"Sun Microsystems, Inc."
"T\u00E4t\u00E4"
"zwei-\nzeilig"
" "
" "
"\ \"
```

- Zwischen Begrenzern: Beliebige **Folge von char-Literalen in allen Schreibweisen** (wörtlich, hexadezimaler Unicode, Ersatzschreibweise)
- Jeder String enthält eine bestimmte **Anzahl Zeichen** als Elemente (obige Beispiele: 4, 22, 4, 12, 1, 0, 1)

## 4.2.3 Javatyp String

- Variablen vom Typ String definieren

```
String text;
```

- Wert an Variable zuweisen:

```
text = "blahblah";
```

- Ein Stringwert kann **nicht verändert werden**. D.h. daß man bspweise kein Zeichen in einem String auswechseln kann. §
- Einer Stringvariablen kann aber **ein anderer String** als Wert zugewiesen werden:

```
String text = "blahblah";
text = "fasel";
```

- Zwei Strings können mit dem Operator + (ein Fall von Polymorphismus) **zu einem neuen Strings verkettet** ("konkateniert") werden. Die Originalstrings bleiben dabei unverändert.

```
String b = "blah";
String f = "fasel";
String bf = b + f;
System.out.println(b);
System.out.println(f);
System.out.println(bf);
```

b =

'b'	'l'	'a'	'h'
-----	-----	-----	-----

f =

'f'	'a'	's'	'e'	'l'
-----	-----	-----	-----	-----

bf =

```
'b' 'l' 'a' 'h' 'f' 'a' 's' 'e' 'l'
```

## 4.2.4 Elementzugriff

- Zeichen aus einem String einzeln herauskopieren mit `charAt(int index)`:

```
String b = "blah";
char c = b.charAt(1); // kleines "l"
```

- Die im Augenblick merkwürdige Syntax "b.charAt(1)" wird nachher genauer erklärt.
- Zeichen innerhalb eines Strings immer **ab 0 durchnummeriert**. Zeichennummer = "Index"

```
'b' 'l' 'a' 'h'
```

```
Index = 0 1 2 3
```

- Zugriffsversuch auf **nicht existierendes Zeichen** führt zu einem Programmfehler <sup>§</sup>

```
String b = "blah";
char c = b.charAt(0); // ok. liefert 'b'
c = b.charAt(35); // Fehler
c = b.charAt(-1); // Fehler
```

- **Anzahl Zeichen eines Strings** (String-Länge) abfragen mit `length`:

```
String b = "blah";
System.out.println(b.length()); // druckt "4"
```

*Vorsicht:* Der Index des letzten Zeichens ist um 1 niedriger als die Länge!

## 4.2.5 Methodenaufrufe

- `charAt` und `length` sind Beispiele von Methoden, wie die früher benutzten mathematischen Bibliotheksfunktionen.
- Methoden richten sich an ein bestimmtes **Zielobjekt**, hier ein String. <sup>§</sup>
- Methoden werden mit **Parametern** aufgerufen, die spezifisch für eine bestimmte Methode sind. Beispielsweise erfordert der Aufruf von `charAt` als Parameter einen Zeichenindex.
- Methoden können einen **Ergebniswert** zurückliefern. Beispielsweise liefert `charAt` ein Zeichen zurück.
- Methodenaufrufe mit Ergebnissen sind aus syntaktischer Sicht **Ausdrücke** vom Typ des Ergebniswertes. Beispielsweise ist ein kompletter `charAt`-Aufruf ein Ausdruck vom Typ `char`.

## 4.2.6 Bibliotheksmethoden

- Häufig gebrauchte String-Methoden:

```
char charAt(int index)
```

Kopiert das Zeichen an Position `index` heraus und liefert es zurück.

```
"blah".charAt(0) → 'b'
```

```
"blah".charAt(2) → 'a'
```

```
String trim()
```

Liefert einen neuen String zurück, der gleich dem Zielobjekt ist, aber ohne führende und schließende Leerzeichen.

```
" blah fasel ".trim() → "blah fasel"
```

```
" blah fasel ".trim().trim() → "blah fasel"
```

```
int length()
```

Liefert die Länge (= Anzahl Zeichen) zurück.

```
"blah".length() → 4
```

```
int indexOf(char ch)
```

Liefert den Index des ersten Vorkommens des Zeichens `ch` zurück oder `-1`, wenn `ch` nicht im String vorkommt.

```
"blah".indexOf('h') → 3
```

```
"blah".indexOf('x') → -1
```

```
int indexOf(String other)
```

Liefert den Index des Anfangs des ersten Vorkommens des Substrings `other` zurück oder `-1`, wenn `other` nicht vorkommt.

```
"blah".indexOf("la") → 1
```

```
"blah".indexOf("al") → -1
```

```
"blah".indexOf("b") → 0
```

- Die Laufzeitbibliothek bietet zahlreiche Methoden zum Umgang mit Strings.

## 4.2.7 Vergleiche

- Vergleich von zwei Strings führt zu komplexerem Problem, siehe weiter unten §
- **Relationale Operatoren** `"=="` und `"!="` technisch anwendbar, aber selten sinnvoll
- Meist gebraucht: **Inhaltlicher Vergleich** mit Bibliotheksmethode `equals`
- Beispiele:

```
"blah".equals("blah") → true
```

```
"blah".equals("Blah") → false
```

```
"blah".equals("blah ") → false (der zweite String endet mit einem Leerzeichen)
```

- Vergleich der lexikographischen Ordnung mit Bibliotheksmethode `compareTo`
- Beispiele:

```
"blah".compareTo("Blah") → 1
```

```
"blah".compareTo("fasel") → -1
```

```
"blah".compareTo("blah") → 0
```

## 5 Arrays

Siehe auch: [\[The Java Language Specification\]](#)

### 5.1 Idee

#### 5.1.1 Container

- Eine Variable primitiven Typs speichert *einen* Wert
- Oft gebraucht: **Komplette Folge** von Werten in einer einzigen Variablen
- **Array** = Container, Behälter für viele Elemente
- Vergleichbar mit Strings, aber allgemeiner §

#### 5.1.2 Eigenschaften

- Alle Elemente haben **den gleichen Typ** §
- Die Elemente sind in einer **festen Reihenfolge** nacheinander angeordnet
- Elemente können weder angefügt, noch weggenommen werden ⇒ die Anzahl Elemente ist **unveränderlich**
- Skizze eines Arrays

71
-4
7220
0
238

#### 5.1.3 Typangabe

- Der Elementtyp mit leeren eckigen Klammern bezeichnet den korrespondierenden **Arraytyp**, z.B. "Array von ints":

```
int[]
```

- "int[]" ist ein **eigener Javatyp**, unabhängig von "int"
- "int" und "[]" bilden eine syntaktische Einheit
- **Wichtig**: Der Arraytyp **legt keine Elementanzahl fest!**

## 5.1.4 Arraytypen

- Zu *jedem Javatype* korrespondiert ein entsprechender *Arraytyp* §
- Arrays sind eine Typfamilie
- Beispiele

Elementtyp	Arraytyp
int	int[]
double	double[]
boolean	boolean[]
char	char[] (das ist <i>nicht</i> String!)
String	String[]

## 5.2 Referenzsemantik

### 5.2.1 Definition Arrayvariablen

- Definition einer Variablen eines Arraytyps wie üblich mit Typ + Name:

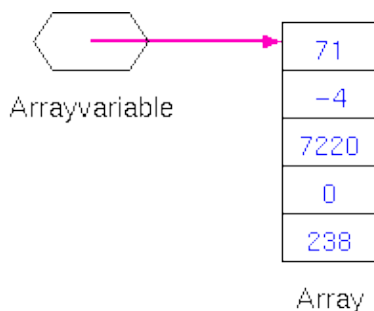
```
int[] table;
double[] samples;
String[] words;
```

- "Variable eines Arraytyps" auch kurz "Arrayvariablen"
- Die *Elementanzahl* bleibt offen, der *Elementtyp* liegt fest

### 5.2.2 Arrayvariable und Array

- Numerische Typen und boolean: Variable = Speicherplatz für den Wert
- Nicht so bei Arraytypen, hier gilt:

**Eine Arrayvariable existiert getrennt und unabhängig vom Array selbst!**



- Die Definition einer Arrayvariablen liefert noch kein Array.
- Allgemein: Typen, bei denen Variable und Objekt getrennt existieren, heißen **Referenztypen** §
- Gegensatz zur Referenztypen: **primitive Typen**
- Beispiele:

<i>Primitive Typen</i>	<i>Referenztypen</i>
int	int[ ]
double	double[ ]
boolean	boolean[ ]
char	char[ ]
...	String
	...

### 5.2.3 Allokieren

- Array selbst erzeugen mit Operator "new"
- Syntax schematisch

```
new type[expression]
type = Elementtyp
expression = Anzahl der Elemente (int)
```

- Beispiele

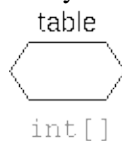
```
new int[4]
new double[1000 + 64/2]
new String[4*256]
```

- new ist ein **unärer Operator**, "new type[size]" ist ein **Ausdruck**
- Der Wert des Ausdrucks ist das neu geschaffene ("allokierte") Array

### 5.2.4 Referenzen

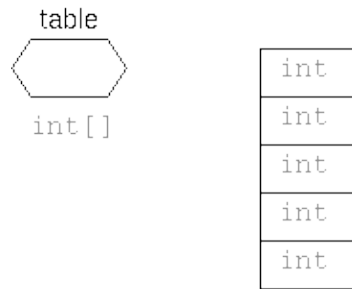
- Zeitlich unabhängige Schritte, nacheinander abgewickelt:

1. Arrayvariable definieren



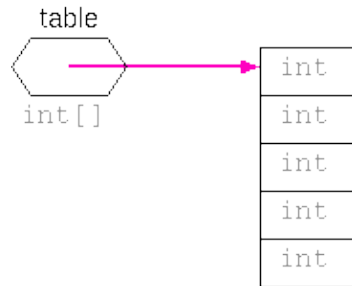
```
int[] table;
```

2. Array allokiieren



```
new int[5]
```

### 3. Array an Arrayvariable zuweisen



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

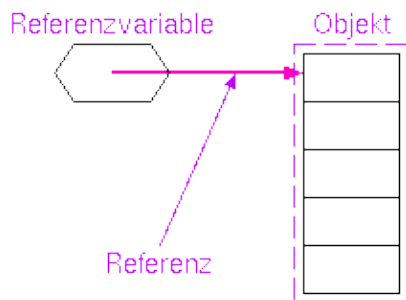
- Syntaktisch in einem Schritt:

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

## 5.2.5 Referenz als Wert

- Der eigentliche "Wert" einer Arrayvariablen ist nicht das Array, sondern die *Referenz*.

- Anschaulich:

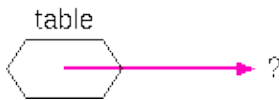


- Das Array selbst ist ein "*Objekt*". Es hat keinen Namen.
- In Java werden Referenzen ausschließlich von der VM erzeugt. Es gibt keinen Weg, neue Referenzen ohne die VM zu erhalten oder vorhandene Referenzen ohne die VM zu manipulieren. §

## 5.2.6 Nicht initialisierte Arrayvariablen

- Eine neu definierte Arrayvariable hat keinen definierten Wert:

```
int[] table;
```



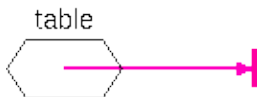
- Arrayvariablen verhalten sich genauso wie primitive Variablen
- Vor der ersten Verwendung muß ein Array zugewiesen werden

## 5.2.7 null-Referenz

- Der Fluchtwert "null" zeigt ein *abwesendes Objekt* an
- null ist eine wohldefinierte Referenz  $\Rightarrow$  kein undefinierter Wert
- Beispiel

```
int[] table;
table = null;
```

- null wird benutzt, wenn ausdrücklich *kein Array* zugewiesen werden soll
- Als Skizze

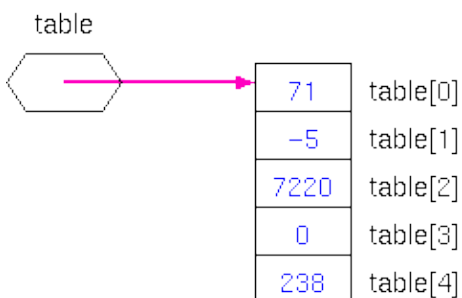


## 5.3 Elementzugriff

### 5.3.1 Indizes

- Die Elemente innerhalb eines Arrays sind *linear angeordnet*.
- Ein einzelnes Element läßt sich mit der *laufenden Nummer* (= "Index") ansprechen
- Indexwerte *beginnen immer mit 0* = erstes Element
- Index des letzten Elementes = (Anzahl - 1)

- Skizze



Array mit 5 Elementen  $\Rightarrow$  Indexwerte 0, 1, 2, 3, 4

### 5.3.2 Indexzugriff

- Syntax des Elementzugriffs

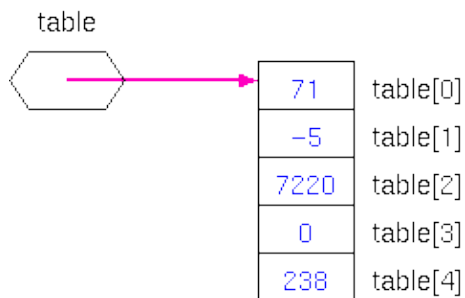
```
array[expression]
```

`array` = Arrayvariable

`expression` = Index des angesprochenen Elementes als integraler Ausdruck

- "`array[expression]`" ist ein L-value
- Der Typ von "`array[expression]`" ist der Elementtyp des Arrays
- Beispiele

```
int[] table = new int[5];
table[1] = -4;
table[3] = 0;
table[152%3] = -table[1]*1805;
table[1]--;
table[table[3]] = 71;
table[9 + table[1]] = 238;
```



### 5.3.3 Indexfehler

- Jedes Array umfaßt eine fixe Anzahl Elemente.
- Elementzugriffe außerhalb des Arrays (zu kleine oder zu große Indexwerte) führen zum **Programmabbruch** § §
- Indexfehler sind eine der **häufigsten Fehlerquellen**.

### 5.3.4 Initialisierung

- Elemente eines neu allokierten `int`-Arrays **automatisch mit 0 initialisiert**. § (Im Gegensatz zu lokalen Variablen: diese bleiben un-initialisiert!)
- "0-Werte" für Elementtypen

Typ	Wert
byte short int long	0
boolean	false

char	'\u0000' (Unicode-Zeichen mit Code 0)
float, double	0.0
alle anderen	null

### 5.3.5 Array-Literale

- Ein Array kann mit einer vorgegebenen Liste von Werten *explizit initialisiert* werden:

```
new type[] {element0, element1, ...}
```

- Bezeichnung "*Array-Literal*"

- Beispiel

```
new int[] {71, -4, 7220, 0, 238}
```

- Eckige Klammern leer: Länge der Liste ⇒ Anzahl Elemente
- Elemente der Liste = *beliebige Ausdrücke* (des Elementtyps) §
- Array-Literal = *Ausdruck*, wie "2+5", aber *keine Anweisung*
- Einsatz: Initialisieren einer Arrayvariablen: §

```
int[] a = new int[] {71, -4, 7220, 0, 238};
```

Etwa gleichwertig mit:

```
int[] a = new int[5];
a[0] = 71;
a[1] = -4;
a[2] = 7220;
a[3] = 0; // eigentlich unnoetig
a[4] = 238;
```

### 5.3.6 Eckige Klammern

- Syntaktische Konstrukte im Zusammenhang mit Arrays unübersichtlich
- Grund: Eckige Klammern in drei unterschiedlichen Situationen
- Am Beispiel eines int-Arrays:

*int[]*

Typangabe bei der Definition von Arrayvariablen.

*new int[n]*

Allokieren eines neuen Arrays mit *n* Elementen = Ausdruck vom Typ *int[]*

*a[i]*

Zugriff auf Element mit Index *i* im Array *a* = Ausdruck vom Typ *int*

## 5.4 Operationen

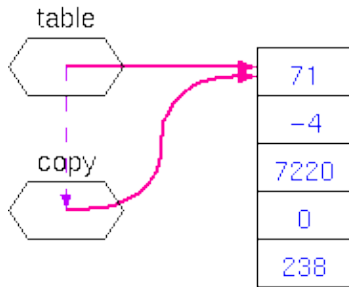
### 5.4.1 Zuweisung

- Zuweisung zwischen zwei Arrays verläuft auf der Ebene der Referenzen:

```
int[] table = new int[] {71, -4, 7220, 0, 238};
int[] copy = table;
```

Die zweite Wertzuweisung **kopiert die Referenz**, nicht das Array

- Aus Sicht der Speicherstrukturen ergibt sich:



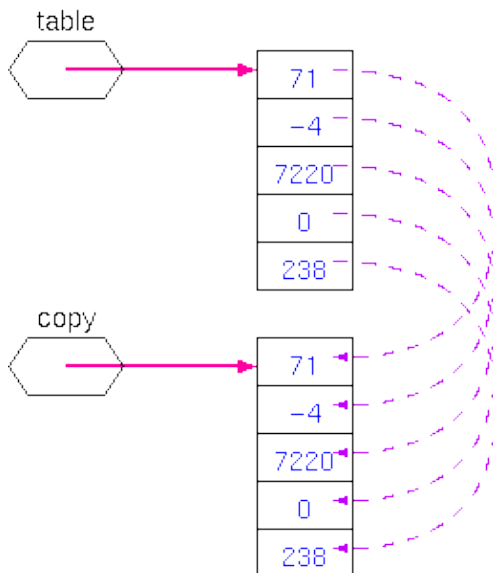
- Das Arrayobjekt selbst ist davon **unberührt**

### 5.4.2 Kopieren

- Um eine komplette Kopie eines Arrays zu produzieren, muß
  1. ein neues Array allokiert werden und
  2. das Original elementweise übertragen werden
- Codefragment:

```
int[] table = new int[] {71, -4, 7220, 0, 238};
int[] copy = new int[5];
for(int i = 0; <5; i++)
    copy[i] = table[i];
```

- Aus Sicht der Speicherstrukturen ergibt sich:



### 5.4.3 Vergleichen

- Zwei Arrays können auf verschiedenen Ebenen gegenübergestellt werden:

*Identität*

Beide Arrays sind ein und dasselbe Objekt (physischer Vergleich)

*Gleichheit*

Beide Arrays enthalten Elemente mit paarweise gleichen Werten (logischer Vergleich)

- Prüfung auf Identität entspricht Vergleich der Referenzen:

```
table == copy...
```

- Prüfung auf Gleichheit entspricht paarweisem Vergleich der Elemente

```
for(int i = 0; i <5; i++)
    table[i] == copy[i]...
```

### 5.4.4 Abfrage Anzahl Elemente

- Jedes Array hat eine fixe Länge = Anzahl Elemente
- Mit dem Suffix ".length" kann die Länge eines Arrays ermittelt werden §
- Syntax allgemein: §

```
array.length
```

Der Wert dieses Ausdrucks hat den Typ int

- Beispiel: Array inhaltlich kopieren:

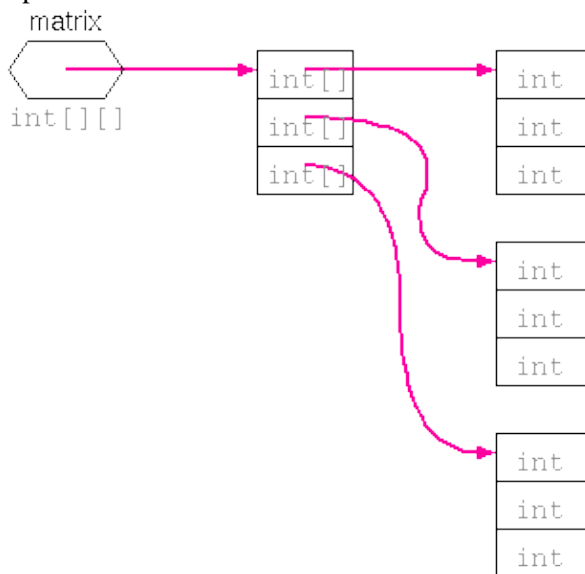
```
int[] table = new int[] {71, -4, 7220, 0, 238};
int[] copy = new int[table.length];
for(int i = 0; <table.length; i++)
```

```
copy[i] = table[i];
```

## 5.5 Geschachtelte Arrays

### 5.5.1 Arrays als Elemente

- **Jeder beliebige Javotyp** kann als Elementtyp verwendet werden
- Arrays sind selbst Javatypes  $\Rightarrow$  Arrays können andere Arrays als Elemente enthalten  $\Rightarrow$  "**geschachtelte Arrays**"
- Speicherstruktur:



- Definitionssyntax

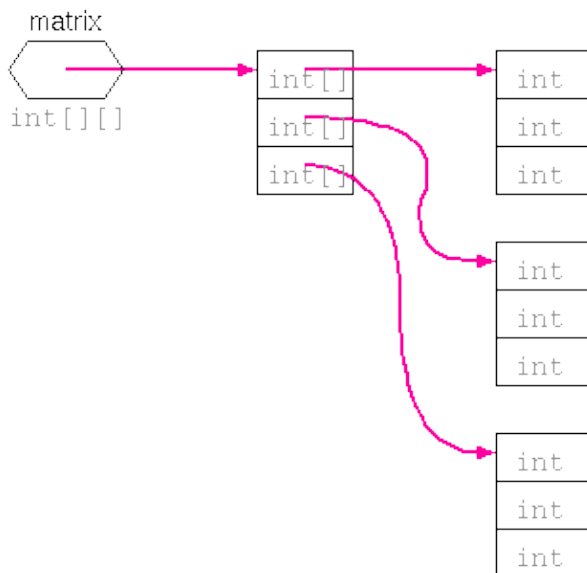
```
int[][] matrix;
```

### 5.5.2 Allokieren

- new akzeptiert **mehrere Elementangaben**:

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

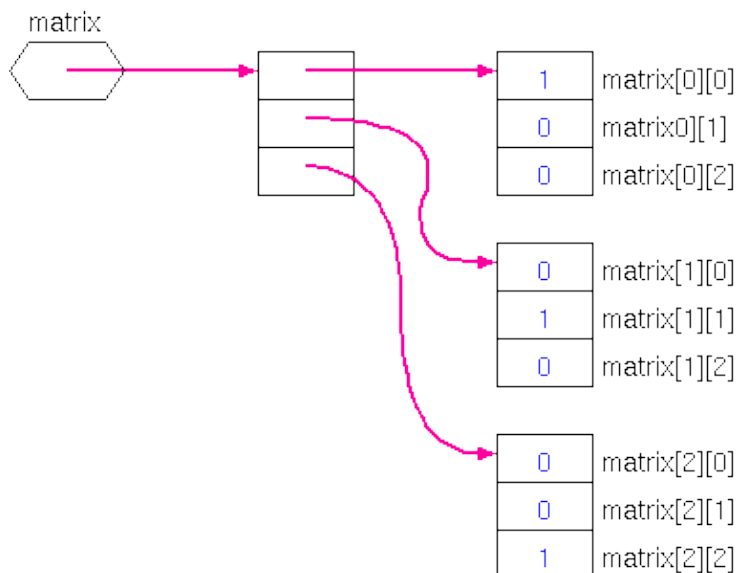
- Allokiert **alle Arrays und Sub-Arrays** in einem Zug



### 5.5.3 Elementzugriff

- Zwei aufeinanderfolgende Indexwerte nacheinander nennen
- Erster Index für das "übergeordnete" Array, zweiter Index für ein "untergeordnetes" Array.
- Beide Indexwerte werden getrennt geprüft
- Beispiel:

```
int[][] matrix = new int[3][3];
for(int i = 0; i < matrix.length; i++)
    matrix[i][i] = 1;
```



### 5.5.4 Initialisierung

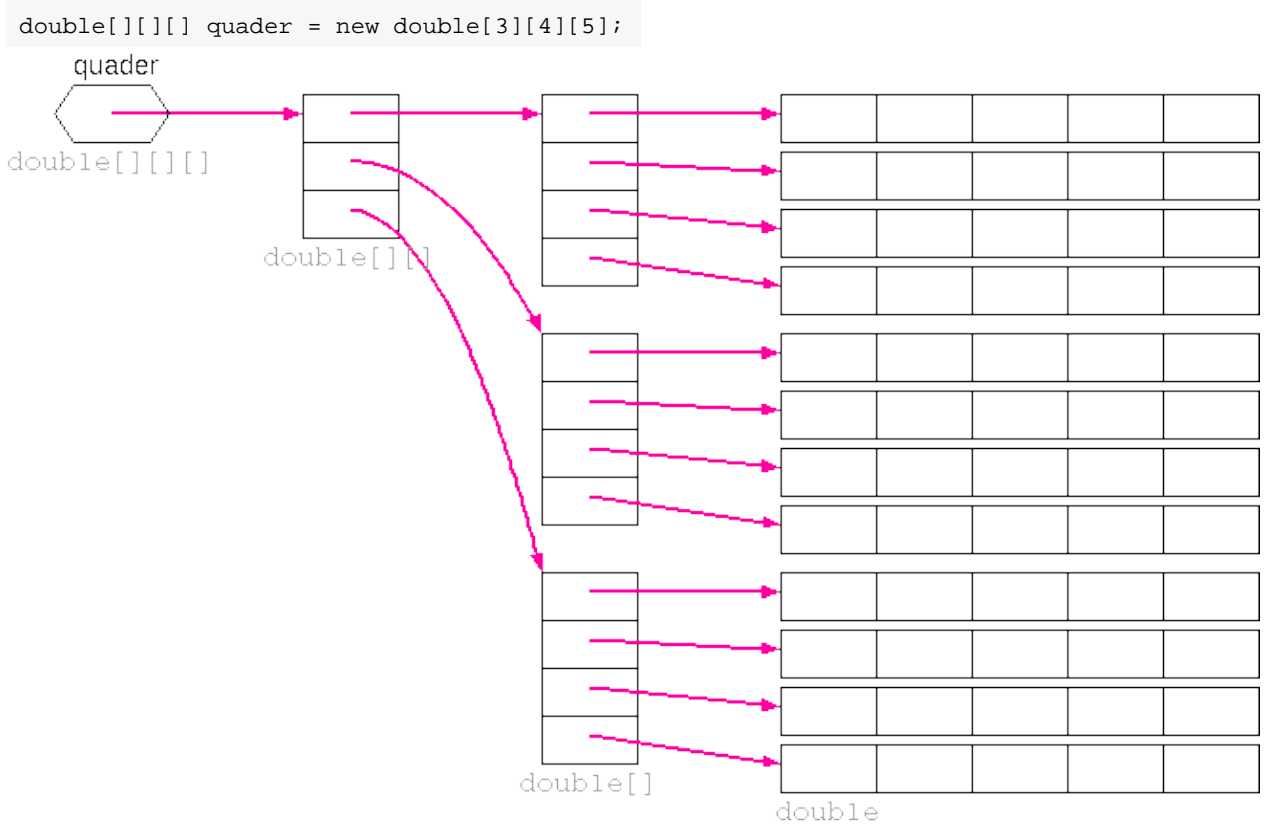
- Geschachtelte Arrays können explizit initialisiert werden:

```
int[][] matrix = new int[][]
{
    {1, 0, 0},
    {0, 1, 0},
    {0, 0, 1}
};
```

- Die Arraygröße ergibt sich aus der Länge der Elementlisten.

## 5.5.5 Mehrdimensionales Array

- Anzahl Dimensionen unbegrenzt, praktisch selten sinnvoll über 3
- Beispiel: Definition eines "quader-förmigen" Arrays



- Speicherplatzbedarf mindestens (Annahme: 1 Referenz beansprucht 4 Byte)

3*4 = 12 Arrays mit je 5 double-Elementen	12*5*8 = 480 Byte
+ 3 Arrays mit je 4 Referenzen	3*4*4 = 48 Byte
+ 1 Array mit 3 Referenzen	3*4 = 12 Byte
+ 1 Variable quader	1*4 = 4 Byte
	Zusammen = 544 Byte

- Füllen der Arrayelement mit dem Abstand vom Ursprung = Element an Position (0, 0, 0):

```
for(int x = 0; x < 3; x++)
    for(int y = 0; y < 4; y++)
```

```

for(5int z++)0; z
    quader[x][y][z] = Math.sqrt(x*x + y*y + z*z);
§

```

- Vorsicht mit vieldimensionalen Arrays: Würfel mit 256 double-Elementen Kantenlänge braucht 128+ MB Platz!

## 5.6 Anwendungen

### 5.6.1 Lineare Suche

- Gegeben: Array mit numerischen Elementen in beliebiger Anordnung.
- Gesucht: Index des ersten Elementes mit gegebenem Wert x.
- x nicht im Array  $\Rightarrow$  Mißerfolg signalisieren
- **Algorithmus "Lineare Suche"**
  1. Stelle den gesuchten Wert x fest
  2. Setze den Suchindex auf 0
  3. Setze den Trefferindex auf -1
  4. Wiederhole solange (Suchindex < Anzahl Elemente) *und* (Trefferindex < 0)...
    - a. Element[Suchindex] ist x?  
Setze Trefferindex := Suchindex
    - b. Sonst:  
Erhöhe den Suchindex um 1
  5. Gib den Trefferindex aus
- Das Ergebnis -1 zeigt Mißerfolg an, ein anderes Ergebnis gibt den Index von x an.
- Beispielprogramm `LinearSearch.java`

### 5.6.2 Aufwand für lineare Suche

- Trefferwahrscheinlichkeit bei vollkommen ungeordneten Elementen  $\frac{1}{n}$  für jeden Index gleich groß
- Im Mittel (bei häufigem Suchen) sind bei einem Array mit n Elementen  $\frac{1}{2}n$  Vergleiche zu erwarten
- Beispiel: Bei  $10^6$  Elementen ist mit etwa 500000 Vergleichen zu rechnen

### 5.6.3 Sortieralgorithmen

- Lineare Suche: Keine Anforderung an die Anordnung der Elemente
- Aber: Lineare Suche ist verhältnismäßig langsam
- Die viel schnellere binäre Suche setzt ein sortiertes Array voraus
- Es gibt viele verschiedene Sortieralgorithmen; einer der eingängigsten heißt "**Bubble Sort**"

### 5.6.4 Voraussetzungen

- Allen Sortieralgorithmen ist gemeinsam:
    - ◆ Eingabe ist ein Array mit Elementen in beliebiger Reihenfolge
    - ◆ Ergebnis ist dasselbe Array mit den Elementen nach aufsteigender  $\leq$  Größe sortiert
  - Es dürfen weder Elemente verschwinden noch dazukommen  
Sichergestellt durch ausschließlich paarweises Vertauschen
  - Außerdem: Elemente müssen vergleichbar sein, d.h. man kann für jedes Paar von Elementen  $a$  und  $b$  entscheiden " $a$  kleiner  $b$ ", " $a$  gleich  $b$ " oder " $a$  größer  $b$ ". ("Totalordnung")  $\leq$
  - **Elementare Operationen** von Sortieralgorithmen:
    1. Zwei Elemente **vertauschen**
    2. Zwei Elemente **vergleichen**
- 

### ▶ 5.6.5 Idee von *Bubble Sort*

- Ein Paar von zwei aufeinanderfolgenden Elementen  $(e_1, e_2)$  steht "richtig", wenn  $e_1 \leq e_2$
  - Das Paar steht "falsch", wenn  $e_1 > e_2 \Rightarrow$  "Fehlpaar"
  - Der Algorithmus durchsucht das ganze Array nach einem Fehlpaar und vertauscht dessen Elemente  $\Rightarrow$  das Fehlpaar ist beseitigt
  - Das wird so oft wiederholt, bis kein Fehlpaar mehr existiert
- 

### ▶ 5.6.6 Algorithmus *Bubble Sort*

- Elementpaare sequentiell von vorne nach hinten prüfen (und Fehlpaare tauschen)  
 $\Rightarrow$  das größte Element "blubbert" an's Ende des Arrays  
 $\Rightarrow$  Bezeichnung "Bubble Sort"
  - Dieses Element hat seine Endposition erreicht und muß nicht mehr berücksichtigt werden
  - Durchlauf wiederholen, jedesmal um 1 Element kürzer als vorher
  - *Bubble Sort* endet, wenn nur noch 1 Element zu prüfen  $\Rightarrow$  nichts mehr zu sortieren
  - **Algorithmus "Bubble Sort":**
    1. Setze Limit := Anzahl Elemente
    2. Wiederhole solange Limit  $> 1$ ...
      - a. Setze Paarpos := 0
      - b. Wiederhole solange Paarpos + 1  $<$  Limit...
        - i. Ist (Paarpos, Paarpos + 1) ein Fehlpaar?  
Vertausche die Elemente
        - ii. Erhöhe Paarpos um 1
      - c. Reduziere Limit um 1
- 

### ▶ 5.6.7 Beispiel *Bubble Sort*

- Ablaufbeispiel

- [Beispielprogramm BubbleSort.java](#) §
- 

## 5.6.8 Binäre Suche

- Ziel wie bei [der Linearen Suche](#): Index oder Abwesenheit eines Elementes  $x$  feststellen.
  - Zusätzlich Voraussetzung: Elemente im Array sortiert
  - Idee: Element  $m$  in der Mitte des Arrays betrachten...
    - $m$
    - $x$  kann, wenn überhaupt, nur in der rechten (oberen) Hälfte zu finden sein
    - $m > x$
    - dto. in der linken (unteren) Hälfte
    - $m == x$
    - Treffer, Ende.
  - Binäre Suche mit der verbleibenden Hälfte fortsetzen, das mittlere Element des Restes betrachten... usw.
  - Ende, wenn Element  $x$  gefunden oder nach fortgesetztem Halbieren kein Element mehr übrig (Mißerfolg)
- 

## 5.6.9 Beispiel binäre Suche

- Ablaufbeispiel
  - [Beispielprogramm BinSearch.java](#)
- 

## 5.6.10 Aufwand für binäre Suche

- Ein Array von  $n$  Elementen läßt sich maximal  $\log_2(n)$ -mal halbieren, bis nur noch 1 Element übrig bleibt
  - Spätestens dann ist die Suche beendet §
  - Beispiel: Bei  $10^6$  Elementen ist mit etwa  $\log_2(10^6) = 20$  Vergleichen zu rechnen (gegenüber etwa 500000 Vergleichen bei der linearen Suche!)
  - Fazit: Wenn ein Array selten verändert, aber oft durchsucht wird, § amortisiert sich der Sortieraufwand durch die wesentlich schnellere binäre Suche in kurzer Zeit.
- 

## 5.7 Bibliotheksmethoden

---

### 5.7.1 Klasse Arrays

- [Klasse Arrays](#) Teil der Laufzeitbibliothek (vergleichbar mit Math und Character)
- Nicht automatisch verfügbar, im Programmkopf erforderlich:

```
import java.util.Arrays;
```

- Sammlung einfacher, häufig gebrauchter Methoden

`void fill(a, x)`

· Kopiert den Wert `x` in alle Elemente des Arrays `a`.

· `x` muß den Elementtyp von `a` haben.

`void sort(a)`

Sortiert das `a` nach aufsteigenden Werten.

`boolean equals(a1, a2)`

· Vergleicht die beiden Arrays `a1` und `a2` elementweise mit `equals`.

· Beide Arrays müssen den gleichen Elementtyp haben.

· Das Ergebnis `true`, wenn die Arrays gleich lang sind und alle Elemente gleich sind.  
Ansonsten ist das Ergebnis `false`.

`int binarySearch(a, x)`

· Sucht `x` im Array `a`.

· `x` muß den Elementtyp von `a` haben.

· `a` muß sortiert sein.

· Das Ergebnis ist der Index des ersten `x` im Array oder ein negativer Wert, wenn `x` nicht gefunden wurde.

- Beispielprogramm
- 

## ▶ 5.7.2 Methode `arraycopy`

- Methode `arraycopy` verhältnismäßig "alt", syntaktisch etwas sonderbar
- Kopiert eine Reihe von Elementen, ersetzt eine Folge von Wertzuweisungen
- Fünf Parameter:

`src`                   Quelle = Array aus dem Elemente gelesen werden

`src_position`       Index des ersten gelesenen Elementes im Array `src`

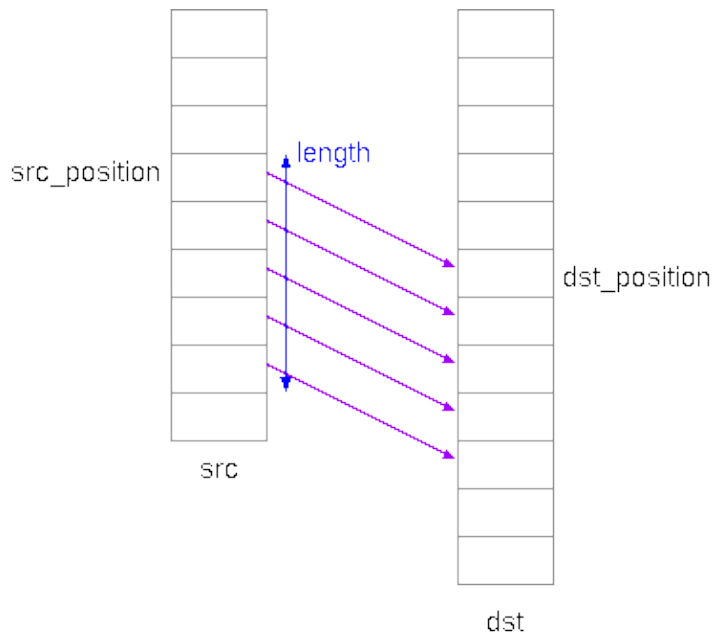
`dst`                   Ziel = Array in das Elemente geschrieben werden

`dst_position`       Index im Array `dst`, ab dem die Elemente nacheinander geschrieben werden

`length`               Anzahl kopierter Elemente

- Skizze

## Single page



- Effizient beim Kopieren großer Arrays (ab einigen Tausend Elementen)
- Funktioniert korrekt beim Kopieren innerhalb eines Arrays,
- Kommt insbesondere mit überlappenden Bereichen zurecht

---

## 6 Klassen

---

### 6.1 Klassendefinitionen und Objekte

#### 6.1.1 Ziel

- Alle bisher benutzten Typen (int, double, boolean, String, Arraytypen) waren vordefiniert
- Ziel: *neue Typen definieren*
- Neue Typen = Klassen §

---

#### 6.1.2 Klassendefinition

- Definition einer Klasse am Beispiel "Bruch"
- Java kennt keinen vordefinierten Typ für Brüche ⇒ Ein entsprechender Typ muß neu definiert werden
- Der Typ wird benannt, z.B. als "Rational"
- Beispiel:

```
class Rational
{
    Einzelheiten
}
```

- Ein Konstrukt dieser Art heißt **Klassendefinition**

### 6.1.3 Namen von Klassen

- Jede Klasse muß einen **eindeutigen Namen** tragen
- Der Name kann ein **beliebiger Java-Identifizier** sein (wie üblich abgesehen von Schlüsselwörtern)
- Jede Klassendefinition steht in einer **eigenen Quelltextdatei**
- Der **Namensrumpf** der Quelltextdatei ist **gleich dem Klassennamen** §
- Die **Extension liegt fest** als ". java"
- Beispiel: Die Definition

```
class Rational
{
    // Einzelheiten
}
```

muß in der Datei "Rational.java" gespeichert sein.

- Konvention: **Klassennamen mit Großbuchstaben beginnen**, mit kleinen Buchstaben fortsetzen; Neue Teilwörter in zusammengesetzten Namen wieder mit großen Buchstaben anfangen, wie z.B. "BufferedInputStream"

### 6.1.4 Datenelemente

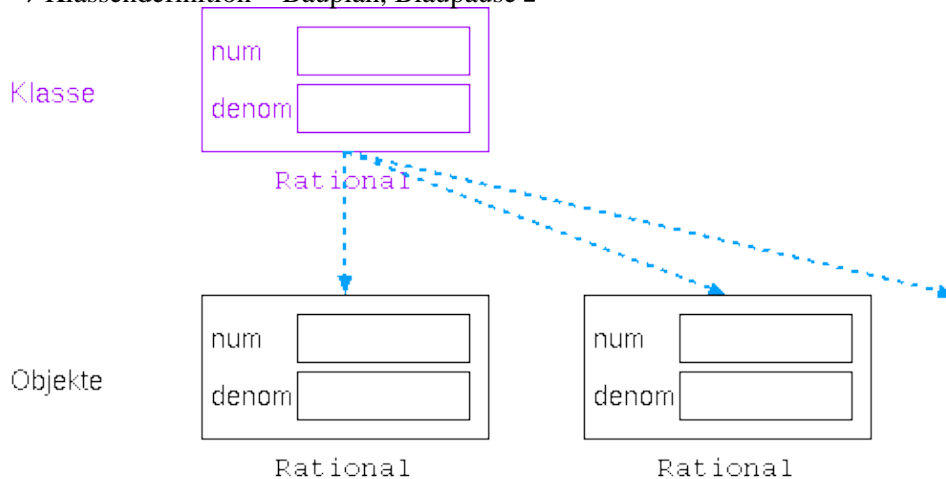
- Bestandteile eines Bruches:
  - ◆ Zähler (für sich gesehen vom Typ int)
  - ◆ Nenner (für sich gesehen vom Typ int)
- Die Klassendefinition nennt diese **Bestandteile**:

```
class Rational
{
    int num;           // Zähler = Numerator
    int denom;        // Nenner = Denominator
}
```

- Ein Bestandteil heißt **Datenelement** der Klasse.
- Rational hat zwei Datenelemente mit den Namen "num" und "denom"
- Datenelemente innerhalb der Klassendefinition **eindeutig benannt**
- Namen der Klasse und der Datenelemente unabhängig
- Konvention: Namen von Datenelementen beginnen **mit kleinen Buchstaben**

### 6.1.5 Objekte = Instanzen

- Eine **Klassendefinition** beschreibt den Aufbau aller Objekte der Klasse
- **Objekt = Instanz** der Klasse = 1 Exemplar, aufgebaut gemäß der Klassendefinition
- Es kann **beliebig viele Objekte** einer Klasse geben, eventuell auch überhaupt keine
- Die Klassendefinition schafft noch keine Objekte; Sie regelt nur, wie Objekte aufgebaut wären, falls es welche geben sollte  
⇒ Klassendefinition ≈ Bauplan, Blaupause §



## 6.1.6 Operator new

- Der Javacompiler nimmt eine Klassendefinition zur Kenntnis, erzeugt daraus aber nicht sofort ein ausführbares Programm §
- Java produziert neue Objekte nur nach **ausdrücklicher Aufforderung**, aber nicht automatisch und unbemerkt
- Die Aufforderung zum Anlegen eines neuen Objektes bringt der Operator "new" § zum Ausdruck
- new akzeptiert einen Klassennamen als Argument und liefert ein **neues Objekt** dieser Klasse als Ergebnis
- Beispiel: Der Ausdruck

```
new Rational()
```

legt ein neues Rational-Objekt an und liefert dieses zurück. (Die leeren runden Klammern werden später erklärt und vorläufig blind gesetzt.)

- Für jedes einzelne Rational-Objekt ist ein neuer Aufruf von new notwendig
- Das "Anlegen" eines neuen Objektes heißt auch **Erzeugen, Instanzieren** oder **Allokieren**
- "new" wurde bei Arrays zum gleichen Zweck benutzt

## 6.2 Variablen

### 6.2.1 Definition

- Benutzerdefinierte Klassen sind gleichberechtigte Typen neben den vordefinierten Typen
- Variablen von Klassen werden syntaktisch gleichartig definiert
- Beispiel:

```
Rational r; // Variable vom Typ Rational
(Eine Klasse Rational muß natürlich definiert worden sein!)
```

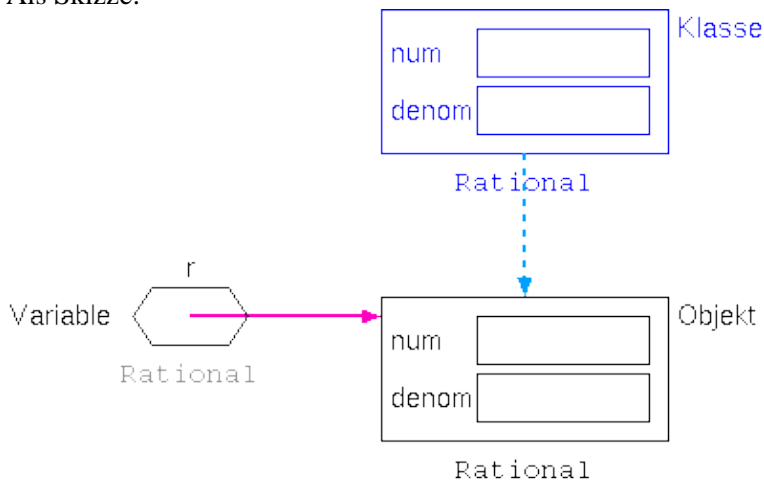
- Wie bei primitiven Typen: Frisch definierte Variablen sind nicht initialisiert  $\Rightarrow$  haben keinen Wert

## 6.2.2 Referenzsemantik

- Eine Variable eines Klassentyps *ist kein Objekt!*
- Objekte müssen nacheinander
  1. (mit new) allokiert und
  2. zugewiesen werden
- Klasse zeigen **Referenzsemantik**
- Beispiel:

```
Rational r; // Variable
r = new Rational(); // Objekt allokiert und zuweisen
```

- Als Skizze:



- Klassen sind mit Arraytypen verwandt, nicht mit primitiven Typen

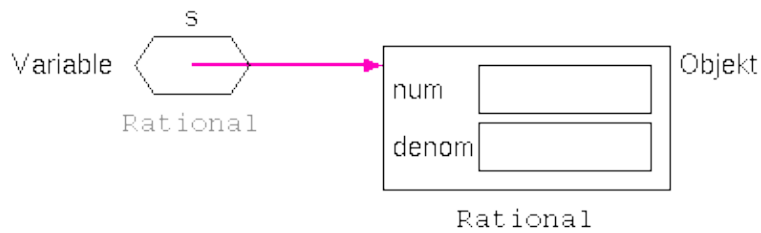
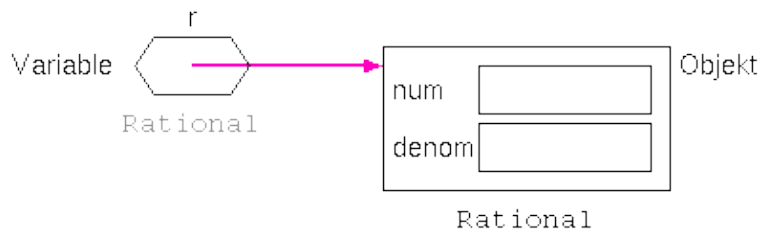
## 6.2.3 Identität vs. Gleichheit

- Referenzsemantik bedeutet, daß *Identität* und (*logische*) *Gleichheit* getrennt zu betrachten sind
- Zwei Variablen können verschiedene Objekte oder dasselbe Objekt referenzieren:

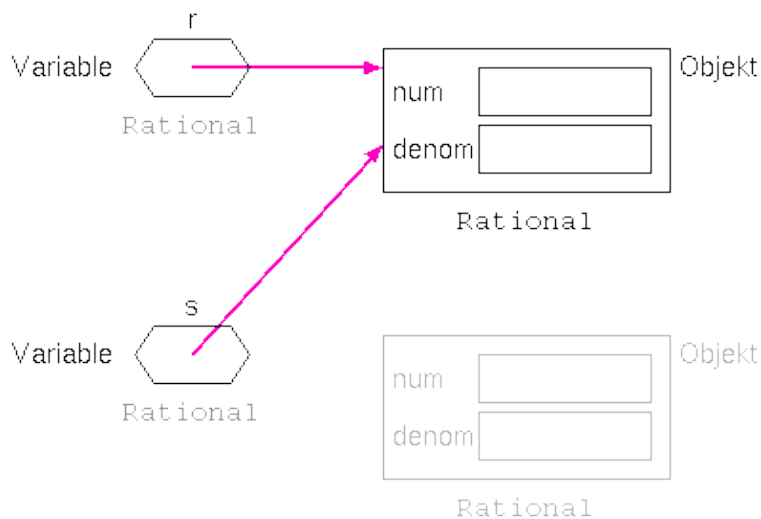
```
Rational r = new Rational();
Rational s = new Rational();
// 1.) r und s referenzieren zwei verschiedene Objekte
```

```
s = r;
// 2.) r und s referenzieren dasselbe Objekt
```

- Schematisch an Punkt 1.)



- An Punkt 2.)



- Zwei verschiedene Objekte können *logisch gleich* sein

## 6.2.4 null-Referenz

- Eine Variable muß kein Objekt referenzieren
- Die null-Referenz zeigt ein abwesendes Objekt an
- Beispiel:

```
Rational r = null;      // kein Objekt
                        // später...
r = new Rational();    // neu allokiertes Objekt
                        // noch später...
r = null;              // kein Objekt mehr
```

## 6.3 Datenelemente

### 6.3.1 Zugriff

- Jedes `Rational`-Objekt enthält die in der Klassendefinition festgelegten Datenelemente
- Datenelemente lassen sich über das Objekt (= *Zielobjekt*) ansprechen = Elementzugriff
- Syntax für Elementzugriff §

```
objekt.element
```

- Beispiel: Bruch als  $\frac{3}{4}$  festlegen:

```
Rational r = new Rational();
           // Objekt noch nicht initialisiert
r.num = 3;
r.denom = 4;
           // Objekt initialisiert mit 3/4
```

### 6.3.2 Datenelemente als *L-values*

- Elementzugriff ist ein *L-Value* (= Variable, Ort, Speicherplatz)
- Ein Datenelement kann in jeder Beziehung wie eine Variable eines primitiven Typs benutzt werden §
- Beispiele

#### *Zuweisung an ein Datenelement*

```
r.num = 21/7 - 1;
```

#### *Datenelement als Teilausdruck*

```
r.denom = 5 - r.num*3;
```

#### *Datenelement als Argument einer Methode*

```
r.num = (int)Math.sqrt(r.denom);
```

#### *Vergleich vom Datenelementen*

```
if(r.num >= r.denom + 1)
    ...
```

#### *Ausgabe von Datenelementen*

```
System.out.println("Rational r = "
    + r.num
    + "/"
    + r.denom);
```

### 6.3.3 Datenelemente unterschiedlicher Objekte

- Jeder Aufruf von `new` produziert ein *neues, selbständiges Objekt*
- Beispiel: Drei verschiedene Brüche

```
Rational r = new Rational();
Rational s = new Rational();
Rational t = new Rational();
```

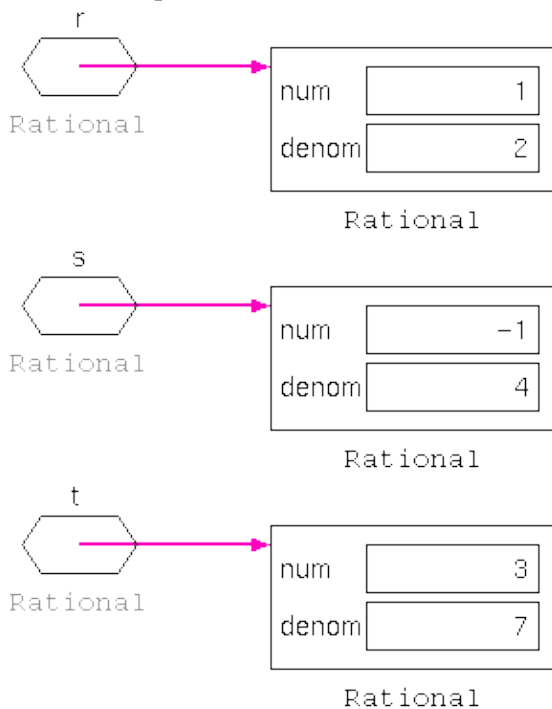
- Jedes Objekt enthält *seine eigenen Datenelemente*. Datenelemente in verschiedenen Objekten sind unabhängig.
- Beispiel: Initialisierungen (Definitionen wie oben)

```
r.num = 1;
r.denom = 2;
    // r ist initialisiert mit 1/2

s.num = -1;
s.denom = 4;
    // s ist initialisiert mit -1/4

t.num = 3*r.num;
t.denom = r.denom + 1 + s.denom;
    // t ist initialisiert mit 3/7
```

- Skizze der Speicherstrukturen:



## 6.3.4 Namensräume

- Elementnamen gelten *innerhalb* (= "lokal in") *einer Klassendefinition*
- Keine Kollision mit gleichlautenden Namen außerhalb oder in anderen Klassendefinitionen
- Beispiel

```
boolean denom;
Rational r = ...;
if(denom &(r.denom > 0)) // ok
    ...
```

- Der Javacompiler kann Namen anhand des Kontextes zuordnen §
  - Elementnamen können verhältnismäßig frei gewählt werden §
  - Eine Klassendefinition ist ein "*Namensraum*". Definitionen in unterschiedlichen Namensräumen sind unabhängig
- 

## 6.4 Methoden

---

### 6.4.1 Funktionalität

- "*Methoden*" werden in Klassen definiert, wie Datenelemente
  - Im Gegensatz zu Datenelementen legen Methoden *Aktivitäten* fest
  - Methoden bestimmen das *Verhalten* von Objekten der Klasse
  - Datenelemente bestimmen den *Zustand* von Objekten der Klasse.
- 

### 6.4.2 Definition einer Methode

- Methoden werden benannt, wie Datenelemente.
- Beispiel: Bruch ausgeben

```
class Rational
{
    int num;
    int denom;

    // Methode zur Ausgabe eines Bruches
    void output()
    {
        System.out.println(num + "/" + denom);
    }
}
```

- Der Name der Methode ist hier "output".
  - In den geschweiften Klammern ist festgelegt, was die Methode tut.
  - Vorteil: jeder Bruch wird in der gleichen Art ausgegeben
  - Vorteil: Anwender vom `Rational` unbelastet von Einzelheiten der Ausgabe
- 

### 6.4.3 Methoden und Datenelemente

- *Mehrere Methoden* in einer Klasse ok, nacheinander auflisten
- Die *Reihenfolge* von Methodendefinitionen ist *belanglos*
- Methoden und Datenelemente können *gemischt* werden

- Üblicherweise stehen **Methoden vorne, Datenelemente hinten**
- Schematisch:

```
class X
{
    Methode1
    Methode2
    ...

    Datenelement1
    Datenelement2
    ...
}
```

## 6.4.4 Kopf und Rumpf

- Eine Methodendefinition besteht aus "**Kopf**" und "**Rumpf**".
- Kopf: **Was** tut die Methode?
- Rumpf: **Wie** stellt die Methode das an?
- Der Rumpf ist syntaktisch ein Block
- Schema einer Methodendefinition (vorläufig)

```
class X
{
    ...
    void methodenname() // Kopf
    {
        // Rumpf
        Anweisung1
        Anweisung2
        ...
    }
    ...
}
```

## 6.4.5 Aufruf

- Ein **Methodenaufruf** ist an ein Objekt (= **Zielobjekt**) gerichtet
- Syntax: Ähnlich Datenelement-Zugriff

```
objekt.methodenname()
```

(Die runden Klammern unterscheiden einen Methodenaufruf von einem Datenelementzugriff.)

- Beispiel:

```
Rational r = new Rational();
r.num = 2;
r.denom = 3;
r.output(); // Ausgabe von "2/3"
```

## 6.4.6 Ablauf eines Aufrufs

- Drei Schritte:
  1. Aufrufer unterbrechen
  2. Methodenrumpf ausführen
  3. Aufrufer fortsetzen
- Wiederholter Aufruf: Rumpf wird jedesmal neu ausgeführt

```
Rational r = new Rational();

r.num = 2;
r.denom = 3;
r.output();    // Ausgabe von "2/3"

r.num = 5;
r.output();    // Ausgabe von "5/3"
```

## 6.4.7 Rückkehr

- Am Ende des Rumpfes "kehrt die Methode zurück"
- Nach der Rückkehr: Fortsetzung mit der *ersten Anweisung hinter dem Aufruf*
- Das Java-Laufzeitsystem stellt den korrekten Ablauf sicher

## 6.4.8 Rumpf als Block

- Methodenrumpf = Block, enthält Anweisungen + Definitionen
- Alle Anweisungsarten im Rumpf zulässig: Kontrollstrukturen, geschachtelte Blöcke, ...
- Im Rumpf sind Variablen *des Aufrufers* nicht erreichbar
- Datenelemente und Methoden der Klassendefinition im Methodenrumpf *ohne Zielobjekt* ansprechbar  
⇒ Zielobjekt = im Aufruf angegebenes Objekt
- Beispiel: Bruch kürzen (Zum Berechnen des GGT siehe beispielsweise Euklid's Algorithmus)

```
class Rational
{
    ...
    void reduce()
    {
        int ggt = ...; // GGT von num und denom berechnen
        num /= ggt;
        denom /= ggt;
    }
    ...
}
```

- Anwendungsbeispiel:

```
Rational r = new Rational();
r.num = 12;
r.denom = 9;
r.output()      // 12/9
r.reduce();
r.output();     // 4/3
```

## 6.5 Argumente und Parameter

### 6.5.1 Argumente und Parameter

- **Parameterübergabe** liefert Information vom Aufrufer an die Methode
- **Parameterliste** im Kopf: Variablen für erwartete Werte
- Syntax:

```
method ::= head body.
head := "void" methodname "(" [paramlist] ")".
paramlist ::= parameter ("," parameter)*.
parameter ::= type name.
```

- Jedem **Parameter** im Kopf der Methodendefinition steht ein **Argument** beim Aufruf gegenüber §
- Beispiel: Erweitern eines Bruches

```
class Rational
{
    void extend(int s)
    {
        num *= s;
        denom *= s;
    }
    ...
}
...
r.extend(2);
```

### 6.5.2 Übergabe

- Argumente und Parameter werden beim Aufruf 1:1 abgeglichen
- Jeder Parameter wird mit dem Wert des entsprechenden Argumentes initialisiert
- Im Rumpf: Parameter  $\approx$  lokale Variablen
- Aufrufe mit zu vielen, zu wenigen oder (im Typ) falschen Argumenten werden nicht übersetzt
- Argumente sind (beliebig komplexe) Ausdrücke §
- Beispiel:

```
class Rational
```

```

{
    void setNum(int n)
    {
        num = n;
    }

    void setDenom(int d)
    {
        denom = d;
    }
    ...
}

```

- Aufrufe der Methode:

```

Rational r = new Rational();
r.setNum(2);           // Argument = 2
r.setDenom(8 - 5);    // Argument = 3
r.output();           // gibt 2/3 aus

```

### 6.5.3 Call Sequence

- Einzelschritte bei einem Methodenaufruf:
  1. Werte der Argumente berechnen,
  2. Parameter erzeugen,
  3. Argumentwerte an die Parameter zuweisen,
  4. Rückkehradresse des Aufrufers merken,
  5. Methodenrumpf ausführen,
  6. Parameter löschen,
  7. Aufrufer an der gemerkten Rückkehradresse fortsetzen.
- Dieser Ablauf heißt "Call sequence".
- Am Beispiel des Aufrufs

```

r.setNum(8 + 3);
r.output();
...

```

1. Argumentwert ausrechnen:  
 $8 + 3 \rightarrow 11$
2. Parameter "int n" erzeugen
3. Parameter initialisieren:  
 $n = 11$
4. "r.output();" als Rückkehradresse des Aufrufers merken,
5. Methodenrumpf ausführen:  
 $num = n;$
6. Parameter "n" löschen,
7. Aufrufer nach "r.output();" fortsetzen.

### 6.5.4 Primitive Typen als Argumente

- Bei der Parameterübergabe: versteckte Wertzuweisung
- Für primitive Parametertypen: Argumentwert kopieren

- Der Methodenrumpf arbeitet mit *Kopien der Argumentwerte*
- Bei Bedarf: *implizite Typkonversion*, evtl. *explizite Typkonversion* (Typecast)
- Beispiel: char-Argument für int-Parameter

```
r.setNum('A');
r.setDenom((int)Math.PI);
r.output();           // gibt 65/3 aus
```

## 6.5.5 Mehrere Parameter

- Im Kopf: *Liste* beliebig vieler (auch 0) Parameter
- Beispiel:

```
class Rational
{
    void set(int n, int d)
    {
        num = n;
        denom = d;
    }

    void setZero()
    {
        num = 0;
        denom = 1;
    }
    ...
}
```

- Aufruf mit passender Anzahl Argumente:

```
r.set(2, 3);
r.output();    // gibt 2/3 aus
r.setZero();
r.output();    // gibt 0/1 aus
```

- Argumente werden von links nach rechts berechnet. Das kann wichtig sein:

```
int i = 1;
r.set(i++, i++);
r.output();    // gibt 1/2 aus
```

## 6.5.6 Referenztypen als Parameter

- Parameter von Referenztypen folgen der *Referenzsemantik*: Bei der Übergabe werden Referenzen kopiert, *aber keine Objekte!*
- Beispiel:

```
class Rational
{
    void mult(Rational x)
```

```

    {
        num *= x.num;
        denom *= x.denom;
    }
    ...
}

```

- Aufruf:

```

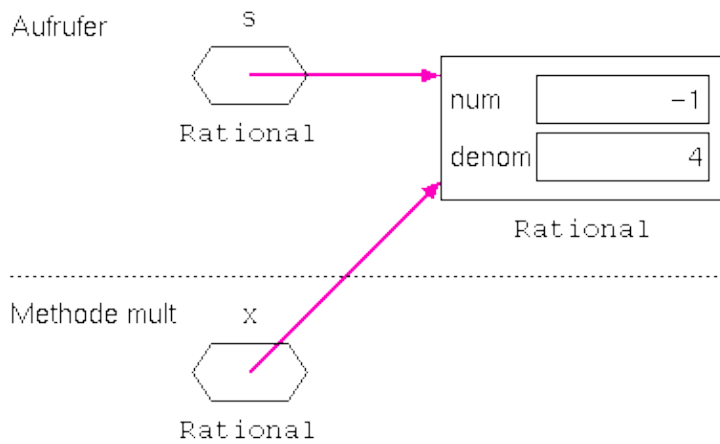
Rational r = new Rational();
r.set(2, 3);
Rational s = new Rational();
s.set(-1, 4);

r.output();    // gibt 2/3 aus
s.output();    // gibt -1/4 aus

r.mult(s);     // multipliziert r mit s

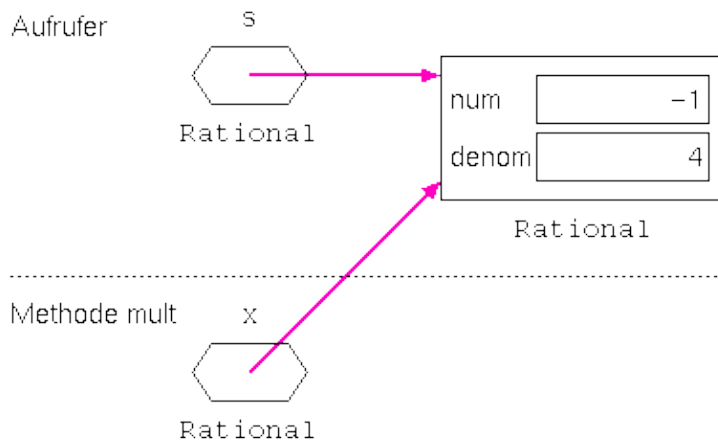
r.output();    // gibt -2/12 aus, modifiziert
s.output();    // gibt -1/4 aus, unverändert

```



## 6.5.7 Problematik: Seiteneffekte

- Im Rumpf von mult: Parameter und Variable des Aufrufers referenzieren *dasselbe Objekt*



- Manipulation im Rumpf wirkt sich beim Aufrufer aus!

- "Bösartige" Implementierung von mult:

```
class Rational
{
    void mult(Rational x)
    {
        num *= x.num;
        denom *= x.denom;
        x.denom = 0;    // BÖSE!
    }
    ...
}
```

- Aufruf (wie oben):

```
Rational r = new Rational();
r.set(2, 3)
Rational s = new Rational();
s.set(-1, 4)

r.output();    // gibt 2/3 aus
s.output();    // gibt -1/4 aus

r.mult(s);     // multipliziert r mit s, modifiziert s!

r.output();    // gibt -2/12 aus
s.output();    // gibt -1/0 aus!
```

- Allgemein: "**Seiteneffekt**" = bleibende Wirkung nach Ende eines Ablaufes
- Manchmal erwünscht (bspweise in reduce), oft ein Fehler oder wenigstens eine Schwäche
- Später diskutierte Konstruktion der Multiplikation vermeidet Seiteneffekte.

## 6.6 Konstruktoren

### 6.6.1 Motivation

- Datenelemente in neuen Objekten initialisieren
- Beispiel: Zähler = 0 und Nenner = 1 für neues Rational-Objekt
- Bisher: Passende Methode aufzurufen

```
Rational r = new Rational();
r.set(0, 1);
```

- Lösung technisch ok, aber **unzuverlässig**: Niemand *stellt sicher*, daß passende Methode zur Initialisierung immer aufgerufen wird
- Besser **Konstruktor**: spezielle Art von Methode, wird **automatisch mit new** aufgerufen §

### 6.6.2 Syntax

- Konstruktoren (Informatik-Slang "*Ctor*") wird definiert wie andere Methoden

- Name eines Konstruktors = Name der Klasse
- Definition eines Konstruktors nennen *keinen* Ergebnistyp (auch nicht void!)  $\Rightarrow$  das Resultat eines Konstruktors ist *immer* ein neues Objekt.
- Konstruktoren können Parameterliste haben, wie "gewöhnliche" Methoden (Custom-Konstruktoren)
- Konstruktor mit *leerer* Parameterliste = **Default-Konstruktor** ("def ctor").

```
class Rational
{
    Rational()        // Default-Konstruktor
    {
        num = 0;
        denom = 1;
    }
    ...
}
```

### 6.6.3 Aufruf

- Bei *jedem* Allokieren eines neuen Objektes **Konstruktor-Aufruf**
- Aufruf *automatisch*  $\Rightarrow$  "Vergessen" oder "Übersehen" unmöglich
- Zuverlässige Kontrolle über *alle Objekte* einer Klasse
- Beispiel

```
Rational r = new Rational();
r.output();           // gibt 0/1 aus
```

### 6.6.4 Konstruktoren mit Parametern

- Default-Konstruktor nicht immer passend
- Konstruktoren mit Parameterliste = **Custom-Konstruktor**
- Am Beispiel der Brüche:

```
class Rational
{
    Rational(int n, int d)
    {
        num = n;
        denom = d;
    }
    ...
}
```

- new-Aufruf mit passenden Argumenten:

```
Rational r = new Rational(2, 3);
r.output();           // gibt 2/3 aus
```

- Wahl passender Parameter im Einzelfall von der konkreten Klasse abhängig

## 6.6.5 Automatisch definierter Konstruktor

- *Kein* Konstruktor definiert  $\Rightarrow$  Java definiert *automatisch einen Default-Konstruktor*
- *Irgendein* Konstruktor explizit definiert  $\Rightarrow$  automatisch definierter Konstruktor *fällt weg*
- Folge: Jede Klasse hat immer (wenigstens) einen Konstruktor (entweder explizit oder automatisch definiert)  $\Rightarrow$  es gibt *keine Klasse ohne Konstruktor*
- Automatisch definierter Default-Konstruktor belegt alle Datenelemente mit Default-Werten vor (siehe Array-Initialisierung)

Typ	Wert
byte short int long	0
boolean	false
char	'\u0000' (Unicode-Zeichen mit Code 0)
float, double	0.0
alle anderen	null

- Eine Klasse muß keinen Default-Konstruktor haben (dann aber Custom-Konstruktoren)
- Beispiel:

```
class Rational
{
    Rational(int n, int d)
    {
        num = n;
        denom = d;
    }
    ...
}
...
new Rational(2, 3);    // ok
new Rational();       // Fehler
```

## 6.6.6 Überladen

- *Mehrere Konstruktoren* in einer Klasse zulässig
- Alle tragen denselben Namen, muß mit dem Klassennamen identisch sein
- Alle müssen *unterschiedliche Parameterlisten* haben
- Beispiel:

```

class Rational
{
    Rational()                // def-ctor
    {
        num = 0;
        denom = 1;
    }

    Rational(int n)          // custom-ctor
    {
        num = n;
        denom = 1;
    }

    Rational(int n, int d)   // custom-ctor
    {
        num = n;
        denom = d;
    }
    ...
}

```

- **Überladene Methoden** = Methoden mit gleichem Namen, verschiedenen Parameterlisten
- Alle Methoden können überladen werden, nicht nur Konstruktoren

## 6.6.7 Aufruf überladener Methoden

- Argumentliste beim Aufruf selektiert konkrete Methode (der Name alleine reicht nicht aus)
- Beispiel:

```

Rational r = new Rational(2, 3);    // ok, custom-ctor
Rational s = new Rational();        // ok, def-ctor
Rational t = new Rational(4);      // ok, custom-ctor

```

- Erscheinungsform des **Polymorphismus**

## 6.6.8 Auflösen von Konflikten

- "**Overload resolution**" = Mechanismus zur **Auswahl der passenden überladenen Methode** für gegebenen Aufruf
- **Implizite Typumwandlung** macht Overload Resolution problematisch
- Beispiel:

```

class Rational
{
    Rational(int n, int d)...
    Rational(char n, char d)...
    ...
}

```

- Aufrufe

```

new Rational(2, 3)      // Rational(int n, int d)...
new Rational('2', '3') // Rational(char n, char d)...
new Rational('2', 3)   // ?

```

- Stufen zur Auswahl
  1. Exakt passende Methode suchen
  2. Passende Methode nach impliziten Typkonversionen suchen
- Lösung beim letzten new:
  - Konversion char → int für erstes Argument
  - ⇒ Aufruf von Rational(int n, int d)
- Umgekehrter Weg: Konversion int → char für zweites Argument
  - ⇒ nicht möglich, weil *keine implizite Konversion* int → char

## 6.6.9 Mehrdeutige Aufrufe

- Manchmal keine klare Entscheidung möglich
- Beispiel (nur zur Illustration, in der Praxis kaum sinnvoll)

```

class Rational
{
    Rational(int n, char d)...
    Rational(char n, int d)...
    ...
}

```

- Aufrufe:

```

new Rational('2', 3)
new Rational(2, 3)
new Rational('2', '3')

```

- Untersuchung der Ausdrücke:
  - `new Rational('2', 3)`  
Ok: Konstruktor mit exakt passenden Parametertypen existiert ⇒ Aufruf von Rational(char, int)
  - `new Rational(2, 3)`  
Fehler: Keine implizite Konversion int → char ⇒ kein passender Konstruktor gefunden.
  - `new Rational('2', '3')`  
Fehler: Zwei mögliche implizite Konversionen char, → int zu *verschiedenen* Aufrufen ⇒ **mehrdeutiger Aufruf**
- Der Java-Compiler weist mehrdeutige Aufrufe ab ("*Constructor is ambiguous*")
- Vorsicht: Mehrdeutig ist der *Aufruf*, nicht die *Definition* der (überladenen) Methoden

## 6.6.10 Kopier-Konstruktor

- **Kopier-Konstruktor** (*copy-ctor*) produziert ein inhaltliches Duplikat ("Kopie", "Clone") eines gegebenen Objektes ("Original")

- Parameter eines Kopier-Konstruktors: Objekt derselben Klasse
- Beispiel:

```
class Rational
{
    Rational(Rational r)    // copy-ctor
    {
        num = r.num;
        denom = r.denom;
    }
    ...
}
```

- Aufruf:

```
Rational r = new Rational(2, 3);
Rational s = new Rational(r);    // auch 2/3, aber neues Objekt
```

- Ansatz der *Wertesemantik* für Referenztypen (Initialisierung)

## 6.6.11 Gegenseitiger Aufruf

- Oft definiert: Ein komplexer Konstruktor mit allen Parametern  
+ weitere einfachere Konstruktoren mit weniger Parametern
- Statt Code in jedem Konstruktor zu wiederholen: Anderen Konstruktor aufrufen
- Syntax

```
this(argument, ...);
```

- Schlüsselwort "this" hat noch andere Rollen
- Beispiel:

```
class Rational
{
    Rational()
    {
        this(0, 1);    // ruft Rational(int, int) auf
    }

    Rational(int n)
    {
        this(n, 1);    // ruft Rational(int, int) auf
    }

    Rational(int n, int d)
    {
        if(d == 0)
            ...
        num = n;
        denom = d;
    }
    ...
}
```

---

## 6.7 Ergebnisrückgabe

---

### 6.7.1 Rückgabe eines Wertes

- Methoden können einen Wert zurückliefern: Informationsfluss Methode → Aufrufer

- Syntax:

1. Ergebnistyp im Kopf (= "Methodentyp")
2. `return` + Ausdruck im Rumpf

- Schematisch

```
type name(...)  
{  
    ...  
    return expression;  
}
```

- Typ von *expression* = Typ im Methodenkopf = Methodentyp

- Beispiele

```
class Rational  
{  
    int getNum()  
    {  
        return num;  
    }  
  
    int getDenom()  
    {  
        return denom;  
    }  
  
    double getReal()  
    {  
        return (double)num/denom;  
    }  
    ...  
}
```

---

### 6.7.2 Mehrfache Rückkehrpunkte

- Mehrere `return`s: Rückkehr beim Erreichen des ersten `return`
- Textuelle Anordnung der `return`s belanglos, Kontrollfluß entscheidet
- Fehlendes `return` (Ausnahme: Ergebnistyp `void`): Fehler §
- Beispiel:

```
class Rational  
{  
    int siggetNum()  
}
```

```

    {
        if(num*denom > 0)
            return 1;
        if(num == 0)
            return 0;
        return -1;
    }
    ...
}

```

### 6.7.3 Rückgabe einer Referenz

- Primitiver Methodentyp: Rückgabe eines *Wertes* (Kopie)
- Referenz–Methodentyp: Rückgabe einer *Referenz* auf ein Objekt der Methode
- Beispiel: Kehrwert

```

class Rational
{
    Rational invert()
    {
        return new Rational(denom, num);
    }
    ...
}

```

- Methodenaufruf liefert Referenz = mögliches Zielobjekt für weiteren Methodenaufruf

```

Rational r = new Rational(3);
r.invert().output();           // gibt 1/3 aus
r.output();                    // gibt 3/1 aus

```

- Mehrere Methodenaufrufe links → rechts:

```

Rational r = new Rational(3);
r.invert().invert().invert().invert().output(); // gibt 3/1 aus

```

### 6.7.4 Neues Objekt als Ergebnis

- Methode liefert (Referenz auf) neues Objekt: Kein Seiteneffekt
- Beispiel: Multiplikation von Brüchen (vgl. die frühere Version mit Seiteneffekt auf das Zielobjekt)

```

class Rational
{
    Rational mult(Rational other)
    {
        return new Rational(num * other.getNum(),
                             denom * other.getDenom());
    }
    ...
}

```

- Damit bspweise Kettenmultiplikation möglich:

```
Rational r = new Rational(2, 3);
r.mult(r).mult(r).output(); // gibt r3 = 8/27 aus
```

- Kettenaufrufe werden links → rechts abgearbeitet
- Klammerung erzwingt ggf. eine andere Auswertungsreihenfolge
- Beispiel (unter der Annahme, daß `div` entsprechend zu `mult` die Rational–Division implementiert):

```
r.div(r).div(r) // (r/r)/r
(r.div(r)).div(r) // wie erstes Beispiel
r.div(r.div(r)) // r/(r/r)
```

## 6.7.5 Ergebnislose Rückkehr

- Methoden *ohne Ergebnis*: nur Seiteneffekte
- Pseudotyp `void`
- Im Rumpf: `return` ohne Ausdruck, legt nur Rückkehrpunkt fest
- Beispiel:

```
class Rational
{
    void reduce()
    {
        int ggt = ...;
        num /= ggt;
        denom /= ggt;
        return;
    }
    ...
}
```

- Rumpf *ohne return*: implizites `return` nach der letzten Anweisung
- Beispiel

```
class Rational
{
    void reduce()
    {
        int ggt = ...;
        num /= ggt;
        denom /= ggt;
    }
    ...
}
```

- Achtung: Konstruktoren haben *überhaupt keinen Ergebnistyp* (auch nicht `void`)

## 6.7.6 Werte– und Referenzsemantik

- Beispiel Multiplikation von Brüchen:

1. Produkt im *Zielobjekt*

```
class Rational
{
    void mult(Rational other)
    {
        num *= other.num;
        denom *= other.denom;
    }
    ...
}
```

2. Produkt in einem *neuen Objekt*

```
class Rational
{
    Rational mult(Rational other)
    {
        return new Rational(num * other.getNum(),
                             denom * other.getDenom());
    }
    ...
}
```

## • Vergleich:

	1. Produkt im Zielobjekt	2. Produkt in neuem Objekt
<i>Seiteneffekte</i>	das Zielobjekt wird verändert	keine Seiteneffekte
<i>Semantik</i>	Referenzsemantik	Wertesemantik (aus der Sicht des Benutzers)
<i>Effizienz</i>	schnell	langsam: Allokieren eines neues Objektes mit new
<i>Sicherheit</i>	unsicher: der Benutzer muß sich der Referenzsemantik bewußt sein	sicher: wegen fehlender Seiteneffekte keine irrtümliche Manipulation möglich

- Faustregel: Wertesemantik wenn möglich, Referenzsemantik wenn nötig

## 6.8 Datenkapselung

### 6.8.1 Ziel: Reduktion von Abhängigkeiten

- OO-Programmieren soll Konstruktion *großer Programme* erleichtern
- Großes Programm zerlegen in kleinere Moduln
- Kleinere Moduln möglichst unabhängig voneinander halten
- Abhängigkeiten zwischen Moduln minimieren  $\Rightarrow$  mehr Freiheiten bei Änderungen, Korrekturen, Erweiterungen, Upgrades, ...

### 6.8.2 Aufbau und Operationen

- Klassen definieren...

*Datenelemente* → **Aufbau** eines Objektes aus Bestandteilen (statische Eigenschaften)  
*Methoden* → **Operationen** = Dienste, Services, Funktionalität eines Objektes (dynamische Eigenschaften)

- Objekte werden von anderen Klassen benutzt; diese sind nur an den Operationen interessiert, aber *nicht am Aufbau!* §
- Logische Konsequenz: Operationen und Aufbau **trennen**

### 6.8.3 Schnittstelle

- **Schnittstelle** einer Klasse = alle Informationen, die für Anwender interessant sind
- Dazu zählen
  - ◆ öffentlich zugängliche Datenelemente
  - ◆ Methodenköpfe
- Rest = **Implementierung** der Klasse

### 6.8.4 Zugriffsschutz

- Jedes Element (Datenelement, Methode) einzeln
  - ◆ verbergen oder
  - ◆ sichtbar machen
- Definition mit Qualifier...
  - public*  
Element frei zugänglich
  - private*  
Element nur innerhalb der Klasse selbst erreichbar
- §
- Allgemeine Faustregel:

**Methoden public**  
**Datenelemente private**

### 6.8.5 Kunde, Lieferant und Vertrag

- Beziehung zwischen Anwender und Klasse ähnelt einer Geschäftsbeziehung

Klasse	Lieferant
Nutzer der Klasse	Kunde
public-Elemente der Klasse	Zugesicherte Leistungen
private-Elemente der Klasse	Interne Maßnahmen und Hilfsmittel

- Wenn ein Lieferant zugesicherte Leistungen streicht oder ändert, sind alle Kunden betroffen
- Interne Maßnahmen können ohne Auswirkungen modifiziert werden
- **Fazit:** Nach außen nur die notwendigen Verbindlichkeiten eingehen, nicht mehr!
- Idee läßt sich mit Assertions technisch umsetzen

---

## 6.9 Assertions

Siehe [\[Java\(TM\) 2 SDK, Standard Edition, version 1.4, Summary of New Features and Enhancements\]](#)

---

### 6.9.1 Motivation

- **Assertion** = Zusicherung: Anweisung mit boole'schem Ausdruck, der immer `true` sein muss
- Beispiel: Nenner in einem `Rational`-Objekt ist niemals 0
- JVM wertet Assertions aus, liefert Fehler falls nicht zutreffend (boole'scher Ausdruck `false`)
- Entwickler fixiert Wissen über Programm im Code  $\Rightarrow$  reduziert Fehlermöglichkeiten
- Funktionsweise eines Programm unabhängig von Assertions

---

### 6.9.2 Syntax

- Gebunden an Schlüsselwort `assert`
- Anweisung im Kontrollfluss

```
assert expression;
```

- *expression* muss vom Typ `boolean` sein
- Beispiel:

```
class Rational
{
    ...
    void reduce()
    {
        int ggt = ...;
        assert ggt != 0;
        num /= ggt;
        denom /= ggt;
        assert denom != 0;
    }
}
```

---

### 6.9.3 Wirkungsweise

- Ausdruck wird ausgewertet und Ergebnis geprüft:  
`true`  
Programm läuft weiter, als wäre nichts geschehen

*false*Programm stoppt sofort mit `AssertionError`

- Nach fehlgeschlagener Assertion keine Möglichkeit zur Programmfortsetzung
- Art des Programmabbruchs wird später genauer beleuchtet
- Beispiel:

```
class TestAssert
{
    public static void main(String[] args)
    {
        System.out.println("starting up...");
        assert args.length == 0;
        System.out.println("winding down...");
    }
}
```

(Die Assertion wird in diesem Beispiel eigentlich mißbraucht. Es ist hier der Einfachheit wegen gezeigt.)

- Aufruf ohne Argument: `$`

```
$ java TestAssert
starting up...
winding down...
$
```

Aufruf mit Argument:

```
$ java TestAssert x
starting up...
Exception in thread "main" java.lang.AssertionError
    at Assert.main(Assert.java:1)
$
```

## 6.9.4 Compilerversionen

- Assertions ab Java 1.4
- `assert` in "altem" Quelltext kein Schlüsselwort, sondern normaler Bezeichner
- Compiler umschaltbar zwischen "altem" und "neuem" Quelltext.
  - ◆ Voreinstellung: Alter Quelltext (`assert` normaler Bezeichner):

```
$ javac TestAssert.java
```

- ◆ Neuer Quelltext, mit Assertions (Schlüsselwort `assert`):

```
$ javac -source 1.4 TestAssert.java
```

- Unveränderte Funktionsweise von `javac` für allen bisher geschriebenen Quelltext vorläufig sichergestellt
- Voreinstellung "alter Quelltext" nur vorübergehend

## 6.9.5 Aktivieren und stilllegen

- Assertions kosten Rechenzeit (wenn auch minimal)
- Lassen sich zur Laufzeit **aktivieren** oder **stilllegen**
- Aktivieren mit Schalter `-ea` (*enable assertions*):

```
$ java -ea TestAssert x
starting up...
Exception in thread "main" java.lang.AssertionError
    at Assert.main(Assert.java:1)
```

- Voreinstellung ohne Schalter = stillgelegt

```
$ java TestAssert x
starting up...
winding down...
```

- Explizites Stilllegen mit Schalter `-da` (*disable assertions*)
- Assertions gezielt in einzelnen Klassen aktivieren oder stilllegen:

```
$ java -ea:Rational TestAssert
```

- Mehrere aufeinanderfolgende Schalter `-ea` und `-da` ok, von links nach rechts angewendet
- Später zusätzlich: Assertions schalten auf Ebene kompletter Packages

## 6.9.6 Was zusichern (und was nicht)?

- Angemessener Einsatz von Assertions schwieriger als Syntax und Semantik:

### **Zusichern:**

Bedingungen, die eine Implementierung *kraft eigener Logik erzwingt*

### **Nicht zusichern:**

Bedingungen, die von äußeren Einflüssen abhängen und außerhalb der Kontrolle der Programmlogik liegen

- Beispiel: **Korrekte Parameterwerte** öffentlicher Methoden **nicht** zu garantieren <sup>§</sup>

```
// Darf nicht mit Nenner 0 aufgerufen werden!
public Rational(int n, int d)
{
    assert d != 0;
}
```

- Beispiel: Offensichtlichkeiten oder **Trivialitäten nicht** zusichern

```
int a = 1;
assert a == 1;
```

- Positive Beispiele: siehe unten

- Faustregeln:
    - ◆ Eine *Assertion scheitert* nur dann, wenn das *Programm falsch* ist.
    - ◆ *Fehlerhafte Anwendung* eines Programms oder einer Klasse bringt *keine Assertion* zum Scheitern.
  - Einsatz von Assertions dokumentiert:
 

**"Dieses Programm ist fehlerfrei."**
  - Benutzen Sie Assertions! §
- 

## 6.9.7 Kontrollfluss

- Beispiel Wertebereiche:

```

if(x > 0)
    ...
else if(x == 0)
    ...
else
{
    assert x
    ...
}

```

- Weiteres Beispiel mit eigener Methode wuerfeln():

```

int w = wuerfeln();
assert w >= 1 & w <= 6;

```

- Beispiel Invarianten:

```

int n = ...positive Zahl...;
while(n != 1)
{
    if(n%2 == 0)
        n /= 2;
    else
        n = 3*n + 1;
    assert n > 0;
}

```

- Beispiel unerreichbarer Code:

```

int m = muenzeWerfen();
if(m == KOPF)
    ...
    return;
if(m == ZAHL)
    ...
    return;
assert false;

```

- Zusichern: Zusammenhänge zwischen *entfernten* (bspweise Methode und Aufrufer) oder in *längeren* Codeabschnitten (Schleifenrumpfe etc.)
- 

## ▶ 6.9.8 Klasseninvarianten

- Datenelemente einer Klasse erfüllen oft bestimmte *Integritätsbedingungen* ("Klasseninvarianten")
  - Bedingungen zusichern, üblicherweise am *Ende von Methoden*
  - Insbesondere wichtig für *ändernde Methoden*, lesende Methoden weniger interessant
  - Evtl. auslagern in private Hilfsmethode mit `boolean`-Ergebnis  
⇒ Assertions rufen nur noch Hilfsmethode auf
- 

## ▶ 6.9.9 Pre- und Postconditions

- Weiter oben eingeführt: Klassendefinition als *Vertrag* zwischen Kunden (Aufrufer) und Lieferant (Objekt)
  - Beide Vertragspartner sichern Leistungen zu:
    - ◆ Kunde beim Aufruf von Methoden (vor dem Ablauf ⇒ *preconditions*)
    - ◆ Lieferant bei der Rückkehr aus Methoden (nach dem Ablauf ⇒ *postconditions*)
  - *Preconditions* sind Sache des Kunden:  
lassen sich nicht alleine kraft Klassendefinition sicherstellen ⇒ *ungeeignet* für Assertions
  - *Postconditions* sind Verbindlichkeiten des Lieferanten:  
ergeben sich aus Klassendefinition ⇒ *Assertions!*
  - Fazit: Was immer eine Methode verspricht zu liefern, *am Ende der Methode* mit Assertions *fixieren*
- 

## ▶ 6.10 Statische Datenelemente

---

### ▶ 6.10.1 Idee

- *Statische Elemente* = besondere Verwaltung §
- Statisch können sein: Datenelemente (zuerst betrachtet) und Methoden (weiter hinten)
- Merkmal: Es gibt nur *ein einziges Exemplar* des Datenelementes
- Alle Objekte der Klasse teilen sich das einzige Exemplar
- Anders ausgedrückt: Ein statisches Datenelement ist *der Klasse als Ganzes* zugeordnet
- Folge: Es existiert *unabhängig von Objekten* der Klasse
- Weitere Folge: Ein statisches Datenelement existiert auch *ohne Objekte*

- Syntax: Wie normales Datenelement, aber mit Vorsatz "static"

```
static int d;
§
```

- In der UML werden statische Elemente unterstrichen
- 

## 6.10.2 Zugriff

- Statisches Datenelement existiert *ohne Bezug zu einem bestimmten Objekt*
- Folge: Die "normale" Zugriffssyntax

```
objekt.element
kann nicht verwendet werden
```

- Statt dessen: Zugriff über den Klassennamen:

```
klasse.element
§
```

- Bekanntes Beispiel: Statisches Datenelement PI der Klasse Math: `Math.PI`
- 

## 6.10.3 Beispiel: Objekte zählen

- Frage: Wie viele Objekte einer Klasse wurden erzeugt? §
- Jedes Objekt entsteht mit einem Konstruktoraufruf
- In den Konstruktoren: statischen Zähler hochzählen
- Beispiel

```
class Rational
{
    Rational()
    {
        total++;
        ...
    }
    ...
    static int total = 0;
}
```

- Abruf der Objektanzahl durch Auslesen des statischen Datenelementes:

```
if(Rational.total > 100)
    System.out.println("Insert coin for more Rational numbers");
```

§

---

## 6.10.4 Beispiel: Objekte nummerieren

- Objekte mit eindeutigen *Seriennummern* stempeln (etwa wie Fahrgestellnummern in Autos)
- Statisches Datenelement für die jeweils nächste freie Seriennummer
- In jedem Konstruktor: Seriennummer zugeteilt und dann (für den nächsten Konstruktoraufwurf) hochzählen
- Beispiel

```
class Rational
{
    Rational()
    {
        serial = nextSerial;
        nextSerial++;
        ...
    }

    int id()
    {
        return serial;
    }

    ...
    private static int nextSerial = 0;
    private int serial;
}
```

- Auslesen der Seriennummer über eine passende Methode:

```
Rational r = new Rational();
...
if(r.id() == 17)
    System.out.println("Sorry-- Rational #17 is out of service");
```

### 6.10.5 Beispiel: Einzelobjekte ("*Singletons*")

- Objekte repräsentieren manchmal externe Ressourcen, wie z.B. Geräte (Maus, Bildschirm, Drucker)
- Existenz dieser Objekte direkt an das betreffende Gerät gebunden
- Nur 1 Tastatur angeschlossen § ⇒ nur 1 Objekt Keyboard als Repräsentant
- Derartige Einzelobjekte heißen "*Singletons*"
- Modellierung mit statischen Datenelementen §

### 6.10.6 Beispiel Singleton: `System.out`

- `System.out` = statisches Datenelement "out" der Klasse "System" § repräsentiert den Bildschirm: `System.out`
- Selbst ein Objekt der Klasse `PrintStream` §
- `PrintStream` definiert allerlei Methoden, so z.B.

```
println(String)
print(String)
```

- Ein Ausdruck der Art

```
System.out.println("Hi")
```

liest sich also folgendermaßen:

1. Die Klasse `java.lang.System` wird angesprochen
2. In `java.lang.System` wird das statische Datenelement `out` benutzt
3. Die Methode `println()` der Klasse von `System.out`, d.h. `PrintStream`, wird gesucht
4. Die für `String`-Parameter überladene Methode `PrintStream.println` wird aufgerufen

## 6.11 Statische Methoden

### 6.11.1 Idee

- Statische Methode = "Klassenmethode"
- Methode richtet sich an die **ganze Klasse**, nicht an ein einzelnes Objekt
- Entsprechend zu statischen Datenelementen §

### 6.11.2 Definition und Aufruf

- Definition: analog zu statischen Datenelementen mit Qualifier "static":

```
static void foo(...)
{...}
```

- Aufruf: analog zum Zugriff auf statische Datenelemente mit dem **Klassennamen**:

```
klasse.methode(argumente, ...)
```

- Statische Methoden können aufgerufen werden, **ohne daß ein Objekt existiert**

### 6.11.3 Beispiel: Hauptprogramm

- Die Methode "`main()`" ist eine statische Methode
- `main()` wird beim Programmstart aufgerufen. Zu diesem Zeitpunkt existiert noch kein Objekt, `main` muß deshalb *statisch* definiert sein!

### 6.11.4 Beispiel: Bibliotheksfunktionen

- Primitive Typen sind keine Klassen  $\Rightarrow$  keine Methoden
- Methoden trotzdem oft wünschenswert, etwa Sinus-Funktion für `double`-Werte

- Ausweg: Künstliche *Hilfsklasse* mit ausschließlich statischen Methoden
- Beispiele: algebraische und transzendente Funktionen (Quadratwurzel, Sinus, Cosinus, Logarithmus etc.) als statische Methode in `Math`

### 6.11.5 Beispiel: Objekte zählen

- Statische Datenelemente ebenso kritisch wie "normale" Datenelemente  $\Rightarrow$  Zugriffsschutz anwenden
- Im o.g. Beispiel Auslesen des Objektzählers *nicht durch direkten Zugriff*, sondern über *Methode*

```
class Rational
{
    Rational()
    {
        total++;
        ...
    }

    static int objects()
    {
        return total;
    }

    ...
    private static int total = 0;
}
```

- Abruf der Objektanzahl durch Aufruf der statische Methode:

```
if(Rational.objects() > 100)
    System.out.println("Insert coin for more Rational numbers");
```

### 6.11.6 Beispiel: GGT–Methode

- GGT–Algorithmus zum Kürzen von Brüchen unabhängig vom einem Zielobjekt
- Läßt sich als statische Methode definieren
- Klassendefinition

```
class Rational
{
    static int ggT(int a, int b)
    {...}

    void reduce()
    {
        int g = ggT(num, denom);
        num /= g;
        denom /= g;
    }
    ...
}
```

### 6.11.7 Einschränkungen

- Aufruf einer statischen Methoden in ganz anderem Kontext als normale (nicht–statische) Methode
  - Einschränkungen einer statischen Methode...
    1. ...kann keine nicht–statischen Datenelemente benutzen
    2. ...kann keine nicht–statischen Methoden aufrufen
    3. ...kann this nicht benutzen
    4. ...wird nicht dynamisch gebunden
- 

## 6.12 Unified Modelling Language *UML*

---

### 6.12.1 Ziel

- Quelltext einer Klassendefinition kann umfangreich werden
    - ⇒ unübersichtlich
    - ⇒ beladen mit syntaktischen Details
    - ⇒ enthält Informationen, die den Anwender nicht interessieren
  - Gesucht: *Neutrale Darstellung* für Anwender einer Klasse
  - *UML* = "*Unified Modelling Language*"
  - Einige Punkte in diesem Abschnitt beziehen sich auf das später behandelte Thema "Vererbung"
- 

### 6.12.2 Merkmale

- UML ist programmiersprachen–unabhängig §
  - UML umfaßt viele verschiedene Notationsformen für unterschiedliche Zwecke
  - Baut auf vorangegangenen Ansätzen auf
  - UML heute allgemein verstanden und akzeptiert
  - Weitere Informationen im Web
- 

### 6.12.3 Statische Klassendiagramme

- Von den verschiedenen Elementen der UML interessant: *Statische Klassendiagramme* und *Aufrufdiagramme*
  - Statische Klassendiagramme beschreiben...
    1. die Struktur von Klassen
    2. Beziehungen zwischen Klassen
    3. erläuternde Anmerkungen
- 

### 6.12.4 Struktur einer Klasse

- Ziel: Übersicht über Bestandteile einer Klasse, in erster Linie Anwendersicht
- Box mit...

- ◆ *Name* der Klasse
- ◆ optional: Liste der *Datenelemente*
- ◆ Liste der *Methoden–Signaturen*

- Signaturen in der Form

*Methodenname(Parameter, Parameter, ...): Ergebnistyp*

- Private Elemente mit Vorsatz "-", protected-Elemente mit Vorsatz "#"
- Statische Elemente unterstrichen
- Beispiel:

<b><i>Rational</i></b>
-int num -int denom
Rational() Rational(int, int) Rational(Rational) mult(Rational): Rational invert() <u>ggT(int, int): int</u>

## 6.12.5 Beziehungen zwischen Klassen

- Verschiedene *logische Beziehungen*:

*Assoziation*

Klasse A nutzt Dienste der Klasse B

*Aggregation*

Klasse A verwaltet Objekte der Klasse B

*Komposition*

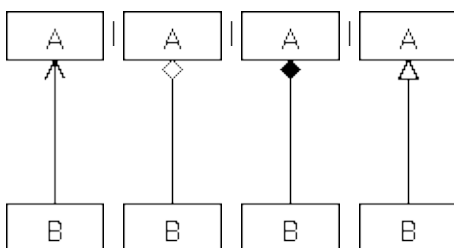
Klasse A besteht aus Objekten der Klasse B

*Vererbung*

A ist Basisklasse der abgeleiteten Klasse B

- Darstellung mit Kanten zwischen Klassen:

<i>Assoziation</i>	<i>Aggregation</i>	<i>Komposition</i>	<i>Vererbung</i>
--------------------	--------------------	--------------------	------------------



## 6.12.6 Rollen und Kardinalitäten

- **Klartext** an Kanten erläutert Aufgaben
- Zahlen am Ende von Kanten nennen die Anzahl der Beteiligten
- Beispiele:

ohne Angabe	1 Objekt
*	beliebig viele Exemplare (auch überhaupt keine)
+	mehrere Exemplare (aber wenigstens eines)
$n$	genau $n$ Exemplare
$n-m$	zwischen $n$ und $m$ Exemplaren

## 6.12.7 Java-Interpretation

- UML ist programmiersprachen-neutral
- UML-Elemente bilden sich auf typische Javakonstrukte ab:
  - Assoziation*  
A-Methoden rufen Methoden der Klasse B auf.
  - Aggregation*  
Ein A-Objekt enthält eines oder mehrere B-Objekte. Die Lebenszeiten von A- und B-Objekt sind unabhängig.
  - Komposition*  
Ein A-Objekt enthält von Anfang an und für die ganze Existenzdauer ein B-Objekt als Baustein. Die Lebenszeiten von A- und B-Objekt sind gekoppelt.
  - Vererbung*  
Direkt als Ableitung.

## 6.12.8 Beispiel

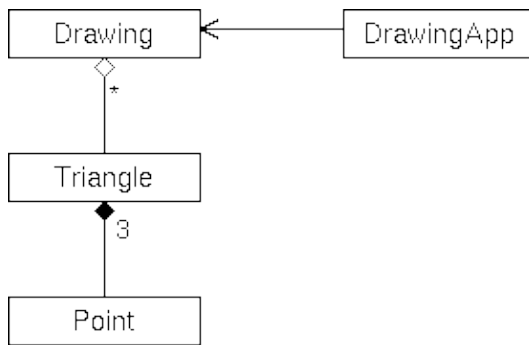
- **Klassen**

DrawingApp	Zeichenprogramm
Drawing	Zeichnung
Triangle	Dreieck
Point	Punkt in der kartesischen Koordinatenebene

- Typische **Beziehungen**:

<i>Assoziation</i>	Die DrawingApp manipuliert ein Objekt Drawing
<i>Aggregation</i>	Ein Zeichnung der Klasse Drawing enthält mehr oder weniger Triangles
<i>Komposition</i>	Triangle enthält 3 Objekte der Klasse Point

- Als **UML**:



- Später werden weitere UML–Elemente eingeführt.

## 6.13 Wie wird eine Klasse definiert?

### 6.13.1 Rezept

<i>Was soll ein Objekt der Klasse repräsentieren?</i>	Geben Sie der Klasse einen Namen (meist ein Substantiv).
<i>Welche <b>Eigenschaften</b> beschreiben den <b>Zustand</b> eines Objektes?</i>	Legen Sie Datenelemente mit Namen und Typen fest.
<i>Welche <b>unveränderlichen Bedingungen</b> gelten für die Eigenschaften (Datenelemente)?</i>	Schreiben Sie das nieder.
<i>Wie wird ein Objekt der Klasse geschaffen?</i>	Legen Sie die Köpfe von Konstruktoren fest.
<i>Welche Eigenschaften eines Objektes sollen <b>öffentlich bekannt</b> sein?</i>	Legen Sie die Köpfe von Auskunftsmethoden fest (Namen meist Adjektive oder Adverbien, Ergebnistyp gemäß Datenelement, aber keine Parameter).
<i>Welche Eigenschaften des Objektes sollen <b>von außen manipulierbar</b> sein?</i>	Legen Sie die Köpfe von Manipulationsmethoden fest (Namen meist Adjektive oder Adverbien, Ergebnistyp "void", ein Parameter gemäß Datenelement).
<i>Welche weiteren <b>Dienste</b> bietet ein Objekt der Klasse an?</i>	Legen Sie Methodenköpfe fest (Namen meist Verben, Parameter und Ergebnistypen).
<i>Welche <b>festen, unveränderlichen Grenzwerte</b> u.ä., gelten für alle Objekte der Klasse?</i>	Legen Sie statische, öffentliche und unveränderliche Datenelemente fest.
<i>Skizzieren Sie ein <b>UML–Klassendiagramm</b></i>	
<i>Welche Methoden haben <b>nicht–triviale Abläufe</b>?</i>	Stellen Sie Struktogramme für diese Methoden auf.
<i><b>Implementieren</b> Sie die Klasse.</i>	

### 6.13.2 Beispiel

*Was soll ein Objekt der Klasse repräsentieren?*

Eine komplexe Zahl:

Complex

Welche **Eigenschaften** beschreiben den **Zustand** eines Objektes?

Realteil und Imaginärteil:

```
double real;
double imag;
```

Welche **unveränderlichen Bedingungen** gelten für die Eigenschaften (Datenelemente)?

Keine.

Wie wird ein Objekt der Klasse **geschaffen**?

Ohne Vorgaben als 0:

```
Complex()
```

Mit Vorgabe einer Floatingpoint-Zahl, Imaginärteil 0:

```
Complex(double r)
```

Mit Vorgabe von Real- und Imaginärteil:

```
Complex(double r, double i)
```

Als Kopie einer anderen komplexen Zahl:

```
Complex(Complex x)
```

Welche **Eigenschaften** eines Objektes sollen **öffentlich bekannt** sein?

Realteil und Imaginärteil, Betrag und Phase:

```
double getReal()
double getImag()
double getValue()
double getPhase()
```

Welche **Eigenschaften** des Objektes sollen **von außen manipulierbar** sein?

Keine wegen Wertesemantik.

Welche weiteren **Dienste** bietet ein Objekt der Klasse an?

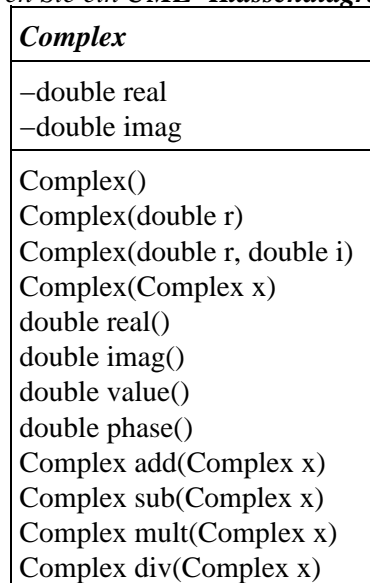
Arithmetik mit Wertesemantik:

```
Complex add(Complex x)
Complex sub(Complex x)
Complex mult(Complex x)
Complex div(Complex x)
```

Welche **festen, unveränderlichen Grenzwerte** u.ä., gelten für alle Objekte der Klasse?

Keine.

Skizzieren Sie ein **UML-Klassendiagramm**



Welche **Methoden** haben **nicht-triviale Abläufe**?

Alle geklärt (trivial oder Formelsammlung)

**Implementieren** Sie die Klasse.

:–)

---

## 7 Vererbung

---

### 7.1 Interfaces

---

#### 7.1.1 Schnittstelle vs. Implementierung

- "Was" bietet eine Klasse? Festgelegt durch...
  1. ...Köpfe der öffentlichen Methoden und...
  2. ...öffentliche Datenelemente
- "Wie" funktioniert eine Klasse? Festgelegt in...
  1. ...Methodenrümpfen und...
  2. ...privaten Elementen
- Anwender braucht nur "Was", aber nicht "Wie"
- Maßnahme: Trennen von "Was" und "Wie"
- Bezeichnungen:

"Was" = *Schnittstelle*

"Wie" = *Implementierung*

---

#### 7.1.2 Interfaces in Java

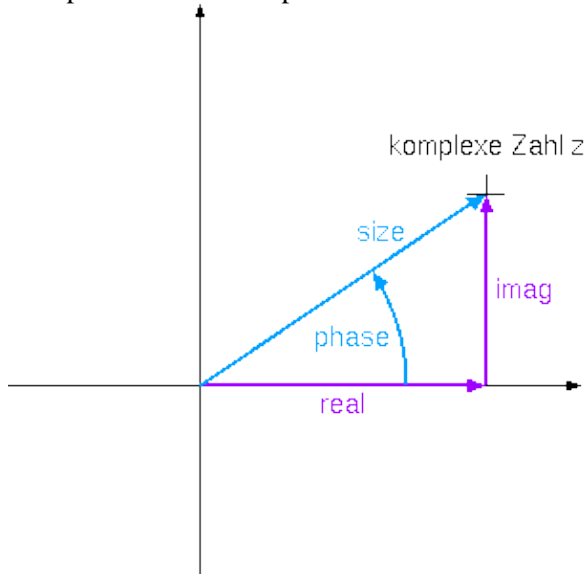
- **Interface** = Java-Sprachmittel zum Definieren einer Schnittstelle
- Ein Interface *definiert*...
  - ◆ Name, Parameterliste, Ergebnistyp der öffentlichen Methoden
  - ◆ Name, Typ öffentlicher Datenelemente
- Ein Interface *definiert nicht*...
  - ◆ Konstruktoren
  - ◆ Rümpfe der öffentlichen Methoden
  - ◆ alle privaten Elemente (Daten, Methoden)
- Syntaktisch

```
interface SomeInterface
{
    ...
}
```

- Äußerlich Interface ≈ Klassendefinition
-

### 7.1.3 Beispiel: Komplexe Zahlen

- Komplexe Zahlen: entsprechen Punkten in der Gauß'schen Zahlenebene:



- Interface für komplexe Zahlen:

```
interface Complex
{
    // Zugriff
    double getReal();
    double getImag();
    double getSize();
    double getPhase();

    // Operationen
    Complex add(Complex c);
    Complex sub(Complex c);
    Complex mult(Complex c);
    Complex div(Complex c);

    // Ausgabe
    String toString();
}
```

- Keine Methodenrumpfe (statt dessen ";" )
- Keine Konstruktoren
- Keine (nicht-öffentlichen) Datenelemente

### 7.1.4 Interface vs. Klasse

- Ein Interface ist *keine Klasse*
- Ein Interface legt fest, was (noch zu definierende) Klassen anbieten werden
- Interfaces können nicht instanziiert werden  $\Rightarrow$  keine Konstruktoren  $\Rightarrow$  Aufruf von `new` = *Fehler*
- Wie bei Klassen: 1 Interface pro Quelltextdatei, Namen korrespondierend

- Bytecode eines übersetzten Interface enthält keinen ausführbaren Code
- Methoden in Interfaces automatisch "public"
- Datenelemente in Interfaces automatisch "public static final"

## 7.2 Implementierung

### 7.2.1 Konkrete Klasse

- Klasse passend zu Interface definieren
- Syntax:

```
class SomeClass implements SomeInterface
{
    ...
}
```

- Klassenelemente: wie bisher
- Klassen- und Interfacenamen eindeutig (= unterschiedlich)
- Beispiel: Klasse (bisher) Complex umbenennen in Cartesian §

```
class Cartesian implements Complex
{
    // Kartesische Koordinaten in der Gauss'schen Zahlenebene
    private double real;
    private double imag;
    ...
}
```

- Klasse **muß** Methoden mit den Köpfen lt. Interface definieren
- Weitere Methoden **dürfen** definiert werden

```
class Cartesian implements Complex
{
    Cartesian()
    {}

    Cartesian(double r, double i)
    {
        real = r;
        imag = i;
    }

    double getReal()
    {
        return real;
    }

    double getImag()
    {...}

    double getSize()
}
```

```

    {
        return Math.sqrt(real*real + imag*imag);
    }

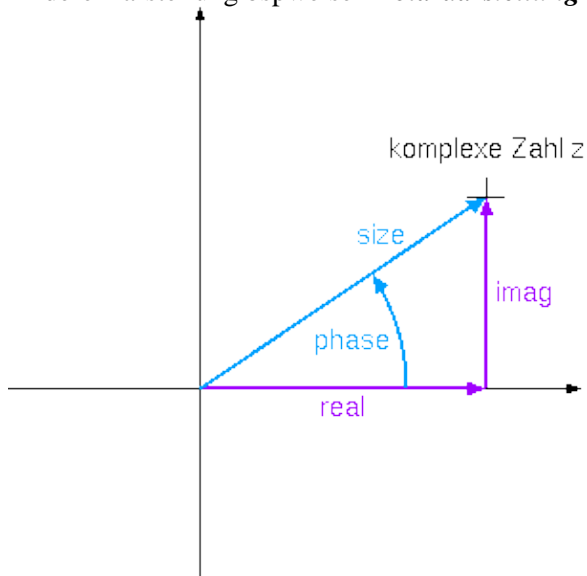
    // alle weiteren Methoden gemäß Interface Complex
    ...

    private double real;
    private double imag;
}

```

## 7.2.2 Alternative Implementierungen

- Kartesische Darstellung komplexer Zahlen: nur *eine* Alternative neben anderen
- Andere Darstellung bspweise "*Polardarstellung*" (Phase und Betrag):



- Interface Complex ist offen, legt *keine bestimmte Darstellung* fest
- Auch Polar folgt dem Interface Complex

```

class Polar implements Complex
{
    Polar()
    {}

    Polar(double s, double ph)
    {
        size = s;
        phase = ph;
    }

    double getReal()
    {
        return size*Math.cos(ph);
    }

    double getImag()
    {...}

    double getSize()
    {

```

```

        return size;
    }

    // alle weiteren Methoden gemäß Interface
    ...

    private double size;
    private double phase;
}

```

- Methodenköpfe in Cartesian und Polar gleich
- *Aber*: Rümpfe unterschiedlich
- Polar und Cartesian sind *gleichberechtigte Implementierungen* desselben Interface Complex

### 7.2.3 Variablen- und Objekttyp

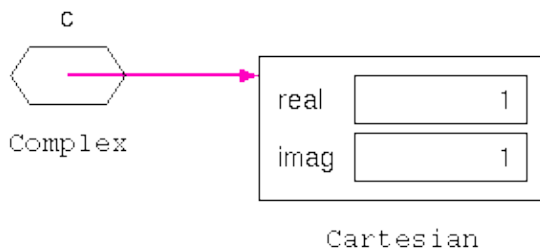
- Polar und Cartesian sind gleich verwendbar, austauschbar
- Beispiel

```

Complex c = new Cartesian(1, 1);
System.out.println(c.getReal() + "/" + c.getImag());

```

- `c` = **Variable** vom Typ Complex
- An `c` wird ein **Objekt** vom Typ Cartesian zugewiesen
- Variable und Objekt haben **unterschiedlichen Typ** §



- Dieser Unterschied ist zulässig wenn gilt:

***Der Objekttyp implementiert den Variablentyp.***

§

### 7.2.4 Methodenauswahl

- Im Beispiel

```

Complex c = new Cartesian(1, 1);

```

Cartesian ersetzen durch Polar:

```

Complex c = new Polar(1.4142136, 0.78539816);

```

Sonst keine Änderung am Programm

- Frage: *Welche Methode* wird aufgerufen in:

```
System.out.println(c.getReal() + "/" + c.getImag());
```

Könnte sein

- ◆ `Cartesian.getReal()` oder
- ◆ `Polar.getReal()`  
(Entsprechend `getImag()`)

- Antwort: Abhängig vom vorangegangenen Konstruktoraufruf *korrekte Auswahl automatisch*

## 7.2.5 Dynamisches Binden

- *Beobachtung*: Der Compiler kann nicht feststellen, ob `c` in der Ausgabeanweisung `Cartesian` oder `Polar` ist:

```
Complex c;
if(...Bedingung...)
    c = new Cartesian(1, 1);
else
    c = new Polar(1.4142136, 0.78539816);
System.out.println(c.getReal() + "/" + c.getImag());
```

- "Bedingung" kann i. allg. erst zur Laufzeit berechnet werden ⇒ Auswahl der "korrekten" Methode zur Laufzeit, d.h. "dynamisch"
- Zentraler Begriff der objektorientierten Programmierung:

**"Dynamisches Binden" =  
Zuordnen eines Rumpfes zum Aufruf zur Laufzeit**

## 7.2.6 Parameterübergabe

### Interfacetypen als Parameter

- Analog zur Wertzuweisung: Parameterübergabe an Methode
- *Parameter* einer Methode = Interfacetyp,  
*Argument* beim Aufruf = beliebige konkrete Klasse (die das Interface implementiert)
- Methodenrumpf benutzt Parameterobjekt, kennt den *konkreten Typ nicht*.

### Beispiel: Kopierkonstruktor

- Beispiel: Kopierkonstruktor für `Cartesian`-Klasse

```
class Cartesian implements Complex
{
    // Kopier-Konstruktor
    Cartesian(Complex c)
```

```

    {
        real = c.getReal();
        imag = c.getImag();
    }
    ...
}

```

- Parameter = Complex (nicht Cartesian)
- Direkter Zugriff auf Cartesian-Datenelemente (wie früher benutzt) unzulässig
- Statt dessen: *Zugriffsmethoden* des Interface
- Dynamisches Binden: passende Methoden `getReal()` und `getImag()` werden aufgerufen
- Aufruf:

```

Complex p = new Polar(2, 1);
Complex c = new Cartesian(p);

```

## ▶ Beispiel: Logischer Vergleich von Objekten

- Methode `same` zum inhaltlichen (logischen) Vergleich komplexer Zahlen §
- Interface definiert Methodenschnittstelle:

```

interface Complex
{
    boolean same(Complex c);
    ...
}

```

- Parametertyp = Complex (weder Cartesian noch Polar)
- Implementierung:

```

class Cartesian implements Complex
{
    boolean same(Complex c)...
}

class Polar implements Complex
{
    boolean same(Complex c)...
}

```

- Implementierung: Interface-Methoden benutzen

```

class Cartesian implements Complex
{
    boolean same(Complex c)
    {
        return (real == c.getReal())
            & (imag == c.getImag());
    }
    ...
}

```

```

}

class Polar implements Complex
{
    boolean same(Complex c)
    {
        // Vorsicht: Versatz um Vollkreis = 2 Pi
        return (size == c.getSize()
            & (phase == c.getPhase()));
    }
    ...
}

```

- Aufruf mit beliebigen Complex-Objekten

```

Complex p = new Polar(2, 1);
Complex c = new Cartesian(p);

if(p.same(c)) // Polar mit Cartesian vergleichen
    ...
if(c.same(p)) // Cartesian mit Polar vergleichen
    ...

```

## 7.2.7 Beispiel: Rüsseltiere

- Ziel: Zoo-Verwaltung, Zoo hält Rüsseltiere
- Gemeinsamen Eigenschaften im Interface:

```

interface Rüsseltier
{
    boolean tröten();
    void füttern();
}

```

- Im Zoo gibt es Elefanten und Mammuts:

```

class Elephant implements Rüsseltier
{
    Elephant(String name)...

    boolean tröten()...

    void füttern()...
}

class Mammut implements Rüsseltier
{
    Mammut(String name)...

    boolean tröten()...

    void füttern()...
}

```

- Für alle Rüsseltiere gilt: Sie müssen gefüttert werden, wenn sie tröten
- Die Pflege der Rüsseltiere wird in einer Methode des Hauptprogramms erledigt:

```
class Zoo
{
    void pflege(Rüsseltier r)
    {
        if(r.tröten())
            r.füttern();
    }
    ...
}
```

- `pflege()` nennt *keine konkrete Rüsseltierklasse*, sondern kommt mit beliebigen Rüsseltieren zurecht
- Im Zoo gibt es 2 Elefanten und 1 Mammut:

```
Rüsseltier[] tiere = new Rüsseltier[...];
tiere[0] = new Elephant("Emil");
tiere[1] = new Elephant("Lotte");
tiere[2] = new Mammut("Zacharias");
```

- Alle werden gepflegt:

```
class Zoo
{
    void allePflegen()
    {
        ...
        for(int i=0; i<tiere.length; i++)
            pflege(tiere[i]);
    }
    ...
}
```

- Der Vorteil wird deutlich, wenn eine neue Sorte Rüsseltiere in's Spiel kommt...

```
class BlauerAmeisenbär implements Rüsseltier
{
    ...
}
```

... und in den Zoo aufgenommen wird:

```
Rüsseltier tiere[3] = new BlauerAmeisenbär("Elise");
...
```

- Die Pflege der Rüsseltiere *bleibt unverändert!*

## 7.2.8 Ergebnisrückgabe

- Ziel: Komplexe Arithmetik für alle konkreten Klassen
- Methoden mit *Wertesemantik* §
- Ergebnistyp = Complex (aber nicht Cartesian oder Polar)
- Jede einzelne Methode liefert ein konkretes Objekt zurück

- Beispiel: Cartesian-Multiplikation (analog zur früher definierten Addition):

```
class Cartesian implements Complex
{
    Complex mult(Complex c)
    {
        return new Cartesian(real*c.getReal() - imag*c.getImag(),
                             real*c.getImag() + imag*c.getReal());
    }
    ...
}
```

- Multiplikation der Polar-Klasse:

```
class Polar implements Complex
{
    Complex mult(Complex c)
    {
        return new Polar(size*c.getSize(),
                         phase + c.getPhase());
    }
    ...
}
```

- Beispiel für den Aufruf:

```
Complex c = new ...; // Cartesian oder Polar
Complex d = new ...; // Cartesian oder Polar
Complex p = c.mult(d);
```

Der konkrete Typ von p bleibt offen

## 7.3 Konkrete Basisklassen

### 7.3.1 Ausklammern der Grundfunktionalität

- Vorgegeben: Klasse Counter für Zähler

```
class Counter
{
    Counter()
    {
        count = 0;
    }

    void step()
    {
        count++;
    }

    int read()
    {
        return count;
    }

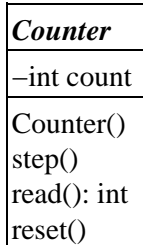
    void reset()
    {
        count = 0;
    }
}
```

```

    }
    private int count;
}

```

- Als UML:



- Anwendungsbeispiel

```

Counter c = new Counter();
while(...irgendeine Bedingung...)
    c.step();
System.out.println("Zählerstand = " + c.read());

```

- Counter dient als **Basisklasse**

## 7.3.2 Erweitern der Funktionalität

- Ziel: **Zusätzlicher Eigenschaft**

*Merken einer Zählerposition und direkter Rücksprung zu dieser gemerkten Position §*

- Ausbau der Basisklasse Counter zur neuen Klasse MemCounter um zwei neue Methoden

```

class MemCounter
{
    // merkt sich den aktuellen Zählerstand
    void mark()...

    // springt zum zuletzt gemerkten Zählerstand
    void recall()...

    ...
}

```

- Anwendungsbeispiel

```

MemCounter mc = new MemCounter();

while(...irgendeine Bedingung...)
    mc.step();
mc.mark();

// irgendwas mit mc machen

mc.recall(); // zurück zur gemerkten Position.

```

## 7.3.3 Lösungswege

### Neue Klasse

MemCounter als eigenständige Klasse zusätzlich zu Counter definieren.

**Nachteile:**

1. Code von Counter in MemCounter dupliziert
2. kein erkennbarer Bezug zwischen Counter und MemCounter
3. Wechsel von Counter zu MemCounter: alle Typangaben zu ändern

### Interface

Gemeinsames Interface, implementiert von Counter und MemCounter.

**Nachteile:** 1. wie oben, nicht aber 2. und 3.

### Ableiten

vermeidet alle drei Nachteile

## 7.3.4 Vererbung als Erweiterung

- Idee:

```

    Basisklasse
+ Erweiterungen
-----
= Abgeleitete Klasse

```

- Am Beispiel

*Basisklasse*

Counter

*Erweiterungen*

Methoden mark und recall

*Abgeleitete Klasse*

MemCounter

- Nomenklatur

Deutsch	Englisch
Basisklasse	base class, superclass
Ableitung	derivation
Vererbung	inheritance
abgeleitete Klasse	derived class

- "Ableitung" und "Vererbung" sind Synonyme

## 7.3.5 Syntax

- Schlüsselwort extends (syntaktisch an derselben Position wie implements):

```

class MemCounter extends Counter
{
    ...
}

```

}

- MemCounter "erbt" automatisch alle Elemente der Basisklasse Counter
- MemCounter definiert nur *zusätzliche Elemente neu*:

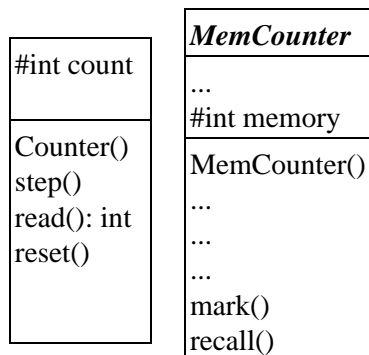
```
class MemCounter extends Counter
{
    MemCounter()
    {
        memory = 0;
    }

    void mark()
    {
        memory = count;
    }

    void recall()
    {
        count = memory;
    }

    private int memory;
}
```

- Elemente der Basisklasse verwendbar, als wären sie lokal definiert (bspweise Datenelement count) §
- Als Skizze:



("..." zeigt ererbte Elemente an)

### 7.3.6 Zugriffsschutz protected

- private erlaubt Zugriff nur aus der eigenen Klasse
- Ausgeschlossen sind alle anderen Klassen, auch abgeleitete Klassen §
- Problem: Abgeleitete Klasse soll zugreifen, aber niemand sonst
- Alternative public: zu großzügig
- Java-Sprachmittel: Zugriffsschutz protected erlaubt Zugriff für

- ◆ die Klasse selbst
- ◆ die (direkt oder indirekt) abgeleiteten Klassen
- ◆ die Klassen im gleichen Package (das Konzept der Packages wird später noch eingeführt)

### 7.3.7 Beispiel für protected

- Basisklasse ändern:

```
class Counter
{
    Counter()
    ...

    protected int count;    // vorher: private int count;
}
```

- Zugriff gewährt für alle unmittelbar und mittelbar abgeleiteten Klassen, insbesondere auch MemCounter.
- protected hat noch weitergehende Folgen, die später erläutert werden.

### 7.3.8 Methodenaufruf

#### Methoden von Basisklassen

- Basisklassenmethoden verfügbar für abgeleitete Objekte:

```
MemCounter c = new MemCounter();
c.step();           // Counter.step
c.mark();           // MemCounter.mark
c.recall();         // MemCounter.recall
c.reset();          // Counter.reset
```

- Für den Anwender kein Unterschied erkennbar
- Paßende Methoden existiert zwangsläufig
- Java-Laufzeitsystem "sucht" schrittweise nach einer Methode...
  1. in der Objektklasse selbst,
  2. in der Basisklasse,
  3. in der Basisklasse der Basisklasse usw.

#### Methoden abgeleiteter Klassen

- Beispiel:

```
Counter c = new MemCounter();
// an c ist ein MemCounter-Objekt zugewiesen

c.step();           // ok, Aufruf von Counter.step
c.mark();           // Fehler!
```

```
c.recall();    // Fehler!
c.reset();    // ok, Aufruf von Counter.reset
```

- Compiler meldet Fehler: Kann nicht sicherstellen, daß die Methoden *in jedem Fall* verfügbar sind §
- Beispiel:

```
Counter c;
if(...irgendeine Bedingung...)
    c = new MemCounter();
else
    c = new Counter();
...
c.mark();    // ok oder nicht?
```

Methode mark() ist verfügbar (falls Bedingung true) oder nicht (Bedingung false) ⇒ kein brauchbares Programm ⇒ wird nicht übersetzt §

### 7.3.9 Modifikation einer Basisklasse

- Beispiel MemCounter: Abgeleitete Klasse *erweitert* die Basisklasse Counter, ererbte Elemente *unverändert*
- Weitere Möglichkeit: Methoden der Basisklasse *redefinieren* (= ändern, ersetzen, überschreiben)
- Kopf muß bei Redefinition *unverändert übernommen bleiben*, Rumpf wird ersetzt
- Beispiel: Klasse LtdCounter für "begrenzten Zähler" (der Zähler bleibt am Anschlag einfach stehen)

```
class LtdCounter extends Counter
{
    LtdCounter(int l)
    {
        limit = l;
    }

    void step()
    {
        <limit>if(count
                    count++;
    }

    protected int limit;
}
```

*Eerbte Elemente*

read(), reset(), count

*Zusätzliche Elemente*

limit, LtdCounter(int)

*Redefinierte Elemente*

step()

- Skizze:

<i>Counter</i>	<i>LtdCounter</i>
#int count	... #int limit
Counter() step() read(): int reset()	LtdCounter(int) <b>step()</b> ... ...

("..." zeigt ererbte Elemente an, redefinierte Elemente sind bunt gedruckt)

- Redefinition: nur Methoden, aber *keine Datenelemente!*

### 7.3.10 Dynamisches Binden redefinierter Methoden

- Aufruf einer redefinierten Methode: Auswahl der "richtigen" Methode zur Laufzeit ⇒ Dynamisches Binden
- Tatsächlicher Objekttyp legt tatsächlich ausgeführte Methode fest
- Beispiel

```

Counter c;
if(...Bedingung...)
    c = new LtdCounter(5);
else
    c = new Counter();

while(true)
{
    c.step();
    System.out.println(c.read());
}

```

Je nach *Bedingung*: Zähler läuft immer weiter oder bleibt bei 5 stehen

### 7.3.11 Einschränken einer Basisklasse

- Abgeleitete Klassen können
  - ◆ Elemente *neu definieren*, d.h. die Basisklasse *erweitern*
  - ◆ Methoden (nicht Datenelemente) *redefinieren*, d.h. die Basisklasse *modifizieren*  
Aber sie können **nicht**
  - ◆ Elemente *wegnehmen*, d.h. die Basisklasse *einschränken*
- Grund: Ein abgeleitetes Objekt § kann ein Basisobjekt *in jedem Kontext* ersetzen.  
⇒ Ein abgeleitetes Objekte muß *mindestens dieselben Möglichkeiten* bieten!

### 7.3.12 Statisches Binden

- Java: Voreinstellung dynamisches Binden (im Ggs. zu C++: Voreinstellung statisches Binden).

- Ausnahmen in Java:
  - static-Methoden*  
Bezugspunkt kein Objekt, sondern eine ganze Klasse. §
  - final-Methoden*  
Können ausdrücklich *nicht* redefiniert werden ⇒ dynamisches Binden entfällt mangels Alternativen.
  - private-Methoden*  
Nur innerhalb der Klasse aufrufbar ⇒ nur 1 Implementierung verfügbar.

### 7.3.13 Konstruktoren

#### Dynamisches Binden

- Konstruktoren werden *nicht dynamisch gebunden*
- Dynamisches Binden orientiert sich am tatsächlichen Objekttyp
- Beim Aufruf eines Konstruktors existiert noch kein Objekt — der Konstruktor soll ja eines produzieren!

#### Automatischer Aufruf Basisklassen–Default–Konstruktors

- Konstruktor einer abgeleiteten Klasse: ruft automatisch den *Default–Konstruktor der Basisklasse*
- Zeitpunkt: Beim Start des Rumpfes
- Beispiel:

```
class LtdCounter extends Counter
{
    LtdCounter(int l)
    {
        // hier wird automatisch Counter() aufgerufen
        limit = l;
    }
    ...
}
```

- Folge: Die Elemente der Basisklasse sind im abgeleiteten Konstruktor bereits initialisiert

#### Custom–Konstruktor einer Basisklasse

- Problem: Abgeleiteter Konstruktor soll einen *Custom–Konstruktor der Basisklasse* aufrufen (und nicht den Default–Konstruktor)
- Syntaktisches Mittel: Ansprechen der Basisklasse über Pseudonamen `super`:

```
Derived()
{
    // ersetzt Aufruf des Basisklassen-Defaultkonstruktors
    super(...);
    ...
}
```

- "super(...)" darf...
  - ◆ nur *I*x und dann
  - ◆ nur als *I. Anweisung* im Konstruktor benutzt werden
- Der automatische Aufruf des Basisklassen-Defaultkonstruktors entspricht

```
Derived()
{
    super();
    ...
}
```

## ▶ Beispiel

- Ziel: Weitere Counter-Klasse LoopCounter: Anschlag wie LtdCounter, aber Rücksetzen auf 0 beim Überlauf
- Definition durch Ableiten von LtdCounter:

```
class LoopCounter extends LtdCounter
{
    LoopCounter(int l)
    {...}

    void step()
    {
        if(count == limit)
            count = 0;    // Überlauf, Rücksetzen auf 0
        else
            count++;
    }
}
```

- Konstruktor "LoopCounter(int l)" muß **Custom-Konstruktor** "LtdCounter(int l)" aufrufen  
⇒ explizit ansprechen mit super(...):

```
class LoopCounter extends LtdCounter
{
    LoopCounter(int l)
    {
        super(l);    // ruft auf: LtdCounter(int l)
    }
    ...
}
```

- Skizze:

<i>Counter</i>	<i>LtdCounter</i>	<i>LoopCounter</i>
#int count	... #int limit	... ...

Counter() step() read(): int reset()	LtdCounter(int) <b>step()</b> ... ...	LoopCounter(int) <b>step()</b> ... ...
---	--	---

("..." zeigt ererbte Elemente an, redefinierte Elemente sind bunt gedruckt)

### 7.3.14 Bezug auf Basisklasse mit super

- Mit "super" (Java-Schlüsselwort) läßt sich allgemein die (direkte) Basisklasse ansprechen
- Beispiel:

```
class LoopCounter extends LtdCounter
{
    LoopCounter(int l)
    {...}

    void step()
    {
        if(super.read() == limit)
            super.reset();
        else
            super.step(); // Vorsicht: Falle!
    }
}
```

- Bei "super.step()": "super" notwendig, sonst *Selbstaufruf!* §
- Bei "super.reset()" und "super.read()": "super" überflüssig

### 7.3.15 Selbstreferenz mit this

- Bezug auf Zielobjekt mit der Pseudo-Variablen "this" (Java-Schlüsselwort)
- "this" wird nicht explizit definiert ⇒ steht automatisch in jeder Methode zur Verfügung §
- Typ von this = immer eigene Klasse §

### 7.3.16 Beispiel für Einsatz von this

- Modifizierte Version der step-Methode für Counter: Methode liefert Zielobjekt (= "sich selbst") als *Ergebniswert* zurück
- Frühere Version:

```
class Counter
{
    ...
    void step()
    {
        count++;
    }
}
```

- Modifizierte Version:

```
class Counter
{
    ...
    Counter step()
    {
        count++;
        return this;
    }
}
```

- Vorteil der modifizierten Version: Erlaubt verkettete Aufrufe

```
Counter c = ...;
c.step().step().step(); // zählt 3x hoch
```

Dieser Aufruf wäre ohne Rückgabe von `this` unzulässig

- Faustregel: Statt `void` evtl. `this` zurückliefern (mehr Flexibilität für wenig Aufwand)

### 7.3.17 Ableiten als Konstruktionsprinzip am Beispiel `Frame`

#### Vordefinierte Basisklasse `java.awt.Frame`

- Konkretes *Beispiel* für die Nutzung der *Vererbung als Konstruktionsmittel* in objektorientierten Systemen
- Die *Java-Laufzeitbibliothek* (Java-API) definiert `java.awt.Frame`  $\cong$  *Bildschirmfenster*
- Fenster ist *leer*  $\Rightarrow$  für sich gesehen ziemlich *nutzlos*
- `Frame` soll zu "*brauchbaren*" Klasse *abgeleitet* werden
- `Frame` definiert Methoden für *Grundfunktionalität* aller Fenster, bspweise:
  - `Frame.setVisible(boolean t)`  
Fenster *sichtbar* (`true`) oder *verborgen* (`false`)
  - `Frame.setSize(int w, int h)`  
*Fenstergröße* von `w` Pixeln Breite und `h` Pixeln Höhe festlegen
- Beispielprogramm

```
import java.awt.*; // Package java.awt zugänglich machen

class MainFrame
{
    public static void main(String[] args)
    {
        Frame f = new Frame();
        f.setVisible(true);
        f.setSize(100, 100);
    }
}
```

produziert ein Fenster mit den Außenmaßen 100×100 Pixel: §



## Redefinition von `Frame.paint`

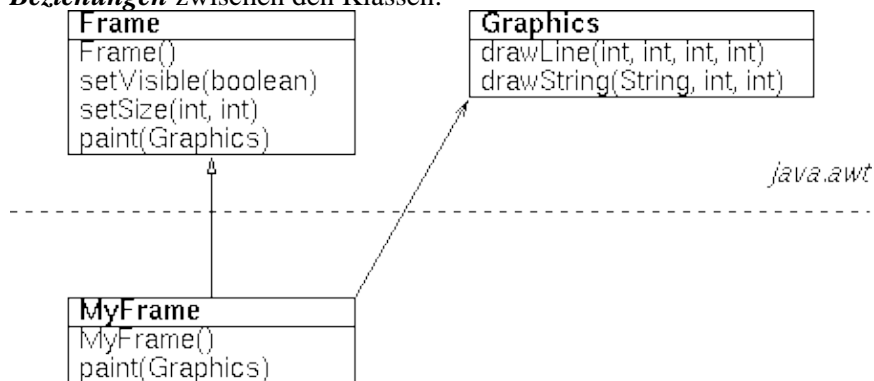
- Weitere Methode

`Frame.paint(Graphics g)`  
sorgt für den **Inhalt** des Fensters

- Basisklassenmethode `Frame.paint` ist leer  $\Rightarrow$  keine Aktion  $\Rightarrow$  **Fenster leer**
- `paint()` in abgeleiteten Klassen **redefinieren**,  $\Rightarrow$  sinnvolle Inhalte einzeichnen

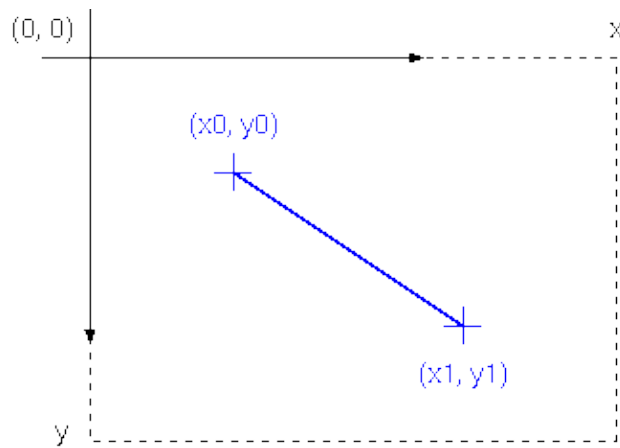
```
class MyFrame extends Frame
{
    public void paint(Graphics g)
    {
        // sinnvollen Fensterinhalt aufbauen
    }
}
```

- **Beziehungen** zwischen den Klassen:



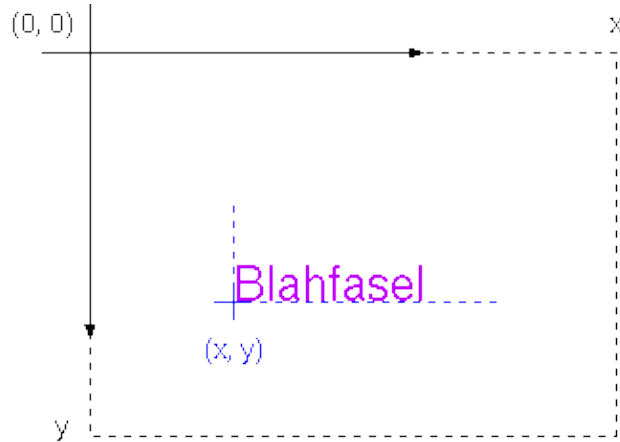
## Klasse `java.awt.Graphics`

- **`paint()`-Parameter** Typ `java.awt.Graphics`
- **`Graphics`** ist in der Laufzeitbibliothek **vordefiniert**, konkreter Aufbau nicht interessant
- `Graphics`-Objekt  $\cong$  **Zeichenfläche**
- `Graphics`-Methoden implementieren **primitive Zeichenoperationen**, wie z.B.:  
`drawLine(int x0, int y0, int x1, int y1)`  
Zieht eine **Linie** vom Punkt  $(x_0, y_0)$  zum Punkt  $(x_1, y_1)$ . Beide Punkte werden in Pixelkoordinaten (von links nach rechts, von oben nach unten, Ursprung in der linken, oberen Fensterecke) definiert:



`drawString(String s, int x, int y)`

Schreibt den **Text** *s* an die Pixelposition  $(x, y)$ . Die Position gibt die linke, untere Ecke des ersten Buchstaben vor.



`setColor(Color c)`

Stellt (bis zum nächsten Aufruf) die gegebene Farbe als **Zeichenfarbe** ein.

*u.v.m.*

Einzelheiten siehe API-Referenz.

## ▶ Ableiten von `java.awt.Frame`

- Leere Basisklassenmethode `Frame.paint(Graphics)` in abgeleiteter Klasse mit Inhalt **redefinieren**
- Im redefinierten `paint`: Mit **Zeichenoperationen** das Fenster füllen
- Beispiel

```
class MyFrame extends Frame
{
    public void paint(Graphics g)
    {
        g.setColor(Color.green);
        g.drawString("Blahfasel", 10, 50);
        g.setColor(Color.yellow);
        g.drawLine(10, 55, 70, 55);
    }
}
```

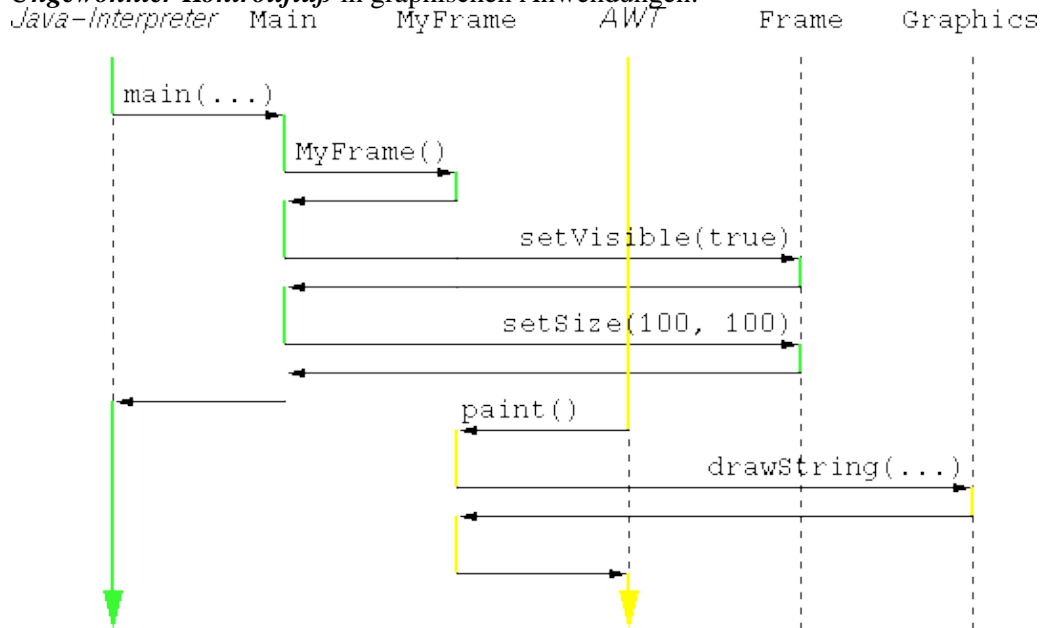
produziert das Fenster



- Passend modifiziertes Hauptprogramm notwendig

## Kontrollfluß

- **Ungewohnter Kontrollfluß** in graphischen Anwendungen:



(siehe früheres Beispiel)

- Senkrechte *Spalten* = *Objekte* und andere Akteure
- = benutzer-definierte Objekte; andere Spalten automatisch, im Verborgenen
- Pfeile quer = Methodenaufrufe und Rückkehr
- **Zwei parallele Abläufe:**
  - ◆ Hauptprogramm = "Java-Interpreter"
  - ◆ Fenstersystem AWT
- Main ruft `paint()` (und dessen Unteraufrufe) **nicht direkt auf**, sondern das **AWT zu unbestimmten Zeitpunkten**
- Programm **läuft zweimal gleichzeitig:**
  - ◆ Ausführen des Inhalts von `Main`
  - ◆ Aufrufe von `paint()`
- Derartige "Aktivitäten" = "**Thread's**" (weiterführende Diskussion später)

## 7.4 Abstrakte Basisklassen

### 7.4.1 Idee

- Ableiten bisher von...

*Interface*

nur Schnittstellen, keine Implementierung

*konkreter Basisklasse*

Methoden mit allen Implementierungen

- Mittelweg: Klasse definieren mit...

◆ *einigen* Methoden

◆ *einigen* Schnittstellen

⇒ **Abstrakte Basisklasse** ("Abstract Base Class" = ABC)

- Beispiel Counter:

```
abstract class Counter
{
    // normale Methoden und Schnittstellen
    ...
}
```

- Elemente in ABCs

*konkrete (= normale) Methoden*

Kopf + Rumpf

*abstrakte Methoden*

Schlüsselwort "abstract" + Kopf, kein Rumpf

- Beispiel:

```
abstract class Counter
{
    // konkrete Methode
    void reset()
    {
        count = 0;
    }

    // abstrakte Methode
    abstract void step();

    // Datenelement
    protected int count;
}
```

### 7.4.2 Eigenschaften

	Interface	ABC	Konkrete Klasse
Muß abgeleitet (implementiert) werden?	ja	ja	nein
Kann (nicht-statische) Datenelemente enthalten?	nein	ja	ja

Kann instanziiert werden ( <code>new</code> )?	nein	nein	ja
Kann Konstruktoren definieren?	nein	ja	ja

ABCs sind Zwischenformen von Interfaces und konkreten Klassen

---

### 7.4.3 Beispiel: Zähler

- Gemeinsam für alle Counter-Klassen

- ◆ `void reset()`
- ◆ `int read()`
- ◆ `boolean same()`
- ◆ `int count`

- In jeder konkreten Zählerklasse unterschiedlich, deshalb in der Basisklasse Counter abstrakt:

- ◆ `void step()`

- Insgesamt:

```
abstract class Counter
{
    protected Counter()
    {
        count = 0;
    }

    int read()
    {
        return count;
    }

    boolean same(Counter c)
    {
        return count == c.read();
    }

    abstract void step();

    protected int count;
}
```

- Abgeleitete konkrete Klasse offener Zähler (SimpleCounter)

```
class SimpleCounter extends Counter
{
    void step()
    {
        count++;
    }
}
```

- Abgeleitete konkrete Klasse Zähler mit Anschlag (LtdCounter)

```

class LtdCounter extends Counter
{
    LtdCounter(int limit)
    {
        this.limit = limit;
    }

    void step()
    {
        <limit)if(count
                count++;
    }

    protected int limit;
}

```

- Abgeleitete konkrete Klasse Zähler mit Rücksetzen (LoopCounter)

```

class LoopCounter extends LtdCounter
{
    LoopCounter(int limit)
    {
        super(limit);
    }

    void step()
    {
        <limit)if(count
                count++;
                else
                count = 0;
    }
}

```

- Der Einsatz von Vererbung als Konstruktionsmittel wird durch abstrakte Basisklasse wesentlich vereinfacht, weil man *pro Methode* (statt pro Klasse) eine passende Vererbungsbeziehung wählen kann §

#### 7.4.4 Rein abstrakte Basisklassen

- Basisklasse mit *ausschließlich abstrakten Methoden* = "**rein abstrakte Basisklasse**" (pure ABC)
- Aber: **Rein abstrakte Basisklasse ≠ Interface!**
- Unterschiede ABC/Interface:
  - ◆ ABCs haben **Konstruktoren**, Interfaces nicht
  - ◆ ABCs können **Datenelemente** enthalten, Interfaces nicht §
  - ◆ Abgeleitete Klassen können nur von **einer einzigen Basisklasse** (konkret, abstrakt oder rein abstrakt) erben, aber **beliebig viele Interfaces** implementieren §
  - ◆ ABC-Methoden haben beliebigen **Zugriffsschutz**, Interface-Methoden sind immer `public`

#### 7.4.5 Mehrfache Vererbung

- Java bietet *einfache Vererbung* ⇒ eine Klasse kann von *nur 1 Basisklasse* erben

```
class A
{...}

class B
{...}

class X extends A, B    // FEHLER
{...}
```

- C++ (und andere objektorientierte Sprachen) bieten *mehrfache Vererbung* ⇒ eine Klasse kann von *beliebig vielen Basisklassen* erben
- Erben *widersprechender Definitionen* aus unterschiedlichen Basisklassen
  - ◆ Einfache Vererbung: nicht möglich ⇒ kein Thema
  - ◆ Mehrfache Vererbung: komplexer Auflösungsmechanismus, schwer beherrschbar §

## 7.4.6 Mehrfache Interfaces

- Einfache Vererbung in Java bezieht sich auf *Basisklassen*
- Implementieren mehrerer Interfaces zulässig:

```
class X extends B implements I1, I2, ...
```

- *Gleiche Methode* aus *unterschiedlichen Interfaces*: ok, 1× zu implementieren §

```
interface A
{
    public void foo();
}

interface B
{
    public void foo();
}

class X implements A, B
{
    public void foo()
    {...}
}
```

- Entscheidend ist die Signatur (= Name + Parametertypen, ohne Ergebnistyp) §

```
interface A
{
    public void foo();
}

interface B
{
    public void foo(int x);
}
```

```
class X implements A, B
{
    public void foo()
    {...}

    public void foo(int x)
    {...}
}
```

## 7.5 Wurzelklasse Object

- Alle Klassen in Java erben *automatisch* von `java.lang.Object` §
- Object ist vordefiniert
- Einige Methoden werden in Object definiert und an *alle Javaklassen* vererbt, wie z.B.
  - `public String toString()`  
liefert eine lesbare Textdarstellung.
  - `public boolean equals(Object x)`  
prüft zwei Objekte auf (logische) Gleichheit. Die Methode wird weiter unten genauer diskutiert.
  - `public int hashCode()`  
berechnet eine eindeutige Kennnummer des Objektes (logisch gleiche Objekte sollen gleiche Kennnummern haben, logisch unterschiedliche Objekte verschiedene Kennnummern).

## 7.6 Dynamische Typinformation

### 7.6.1 Statischer und dynamischer Typ

- **Statischer Typ** = Typ laut Definition im Quelltext. § Beispiel:

```
Counter c;
c hat statischen Typ Counter
```

- **Dynamischer Typ** = Typ des tatsächlichen Objektes. § Beispiel:

```
Counter c;
c = new LtdCounter(23);
c hat dynamischen Typ LtdCounter
```

- Dynamisches Binden benutzt den dynamischen Typ (daher die Bezeichnung)
- Statischer und dynamischer Typ können *nur bei Vererbung* abweichen.

### 7.6.2 Abgeleitete Klassen mit zusätzlichen Methoden

- Bei redefinierten Methoden löst dynamisches Binden das Zuordnungsproblem
- Bei zusätzlichen Methoden (in abgeleiteten Klassen gegenüber der Basisklasse) tritt ein Problem auf: Zum Aufruf muß der *dynamische Typ* sichergestellt sein.

- Beispiel:

```
class Counter
{
    ...
}

class MemCounter extends Counter
{
    void recall() // neu in der Klasse MemCounter, fehlt in Counter
    {...}
}

Counter c = ...;
...
c.recall(); // ok, oder nicht?
```

- Der Compiler lehnt den Aufruf von `c.recall()` ab, weil `c` möglicherweise den falschen (dynamischen) Typ hat
- Problem: Wie kann `recall()` trotzdem benutzt werden?
- Anders formuliert: Wie kann man bzgl. des dynamischen Typs sichergehen §

### 7.6.3 Lösung: Fette Basisklassen

- Dummy-Versionen zusätzlicher Methoden abgeleiteter Klassen in die Basisklasse aufnehmen
- Beispiel:

```
class Counter
{
    void recall() // Dummy-Methode
    {}
}

class MemCounter extends Counter
{
    void recall() // "echte" Methode
    {...}
}

Counter c = ...;
c.recall(); // ok!
```

- Nachteile:
  - ◆ Passende Implementierung der Dummymethode in der Basisklasse unbestimmt (keine Reaktion, Fehlermeldung, Programmabbruch, ...?)
  - ◆ Basisklasse *verfettet* mit Dummymethoden aus *allen* abgeleiteten Klassen (deshalb: "*fette Basisklasse*")
  - ◆ Erweiterungen in abgeleiteten Klassen erfordern Erweiterungen der Basisklasse ⇒ widerspricht dem Hauptziel der Reduktion von Abhängigkeiten
- Fazit: Lösung i. allg. unbrauchbar

## 7.6.4 Lösung: Dynamische Typinformation mit `instanceof`

- Idee: Objekt nach dem tatsächlichen (dynamischen) Typ *fragen*: Abruf der **Laufzeit-Typinformation**
- Je nach Antwort:
  - ◆ Methode der abgeleiteten Klasse aufrufen oder
  - ◆ Aufruf übergehen
- Schematisch:

```
class Counter...

class MemCounter extends Counter...

Counter c = ...;
...
if(...Ist c ein MemCounter?... )
    c.recall(); // ok!
```

- Abfrage in Java mit Operator "instanceof":

```
Objekt instanceof Klasse
liefert den boolean-Wert...
true
    wenn Objekt den Typ Klasse hat und
false
    ansonsten
```

- Am Beispiel

```
Counter c = ...;
if(c instanceof MemCounter)
    c.recall();
```

(dieses Codefragment funktioniert immer noch nicht, siehe nächster Punkt.)

## 7.6.5 *Typecast*

- Technisches Problem: Trotz Prüfung bleibt der statische Typ von `c` unverändert  $\Rightarrow$  der Compiler übersetzt den Aufruf nicht
- Ausweg: **Neues Objekt** vom dynamischen Typ aus der "alten" Variablen produzieren
- Umwandeln in einen neuen Typ heißt "*Type Cast*" (als Verb im Informatik-Slang "*casten*")
- Syntaktisch:

```
(Zieltyp)Ausdruck
```

- Am Beispiel

```

if(c instanceof MemCounter)
{
    MemCounter mc = (MemCounter)c;           // Type cast
    mc.recall();
}

```

oder kürzer

```

if(c instanceof MemCounter)
    ((MemCounter)c).recall();

```

§

## 7.6.6 Beispiel: equals

- equals soll das Zielobjekt mit einem anderen Objekt other vergleichen
- Von Object ererbte Fassung prüft nur Identität von Objekten  $\Rightarrow$  redefinieren zur Prüfung logischer Gleichheit
- other kann *irgendein Objekt* sein  $\Rightarrow$  Parametertyp Object §
- Redefiniertes equals muß zuerst die Typgleichheit sicherstellen:

```

class X
{
    public boolean equals(Object other)
    {
        if(other == null)
            // Zielobjekt existiert, other nicht
            return false;
        else if(other instanceof X)
            // weitere Pruefungen
        else
            // Zielobjekt und other haben unterschiedliche Typen
            return false;
    }
    ...
}

```

- Weitere Probleme bleiben offen. Weiterführende Diskussion siehe etwa: Mark Davis: "Liberte, equalite, fraternite", Java Report 1/2000, pg. 46.

## 7.6.7 Typfehler

- Im vorhergehenden Beispiel: Abfrage stellt sicher, daß der Typecast funktioniert
- Welche Folgen hat ein Typecast auf einen "falschen" Typ?
- Beispiel

```

...(LtdCounter)c... // ohne Pruefung mit instanceof!

```

- Laufzeitfehler (Exception), ähnlich wie bei Division durch Null
- Fazit: instanceof und Typecast meist gekoppelt

## 8 Exception Handling

Siehe auch: [\[The Java Language Specification\]](#)

---

### 8.1 Problematik

- Laufzeitfehler unvermeidbar
  - Beispiele:
    - ◆ Division durch Null
    - ◆ Zugriff auf Array-Element außerhalb der Grenzen
    - ◆ Fehlerhafte Benutzereingabe
  - Ziel: kontrollierte Reaktion
  - "Zufällige" Auswirkungen eines Fehlers inakzeptabel (Sicherheitsaspekte)
- 

### 8.2 Alternativen

---

#### 8.2.1 Globale Fehlervariable

- Mittel: überall erreichbare Variable (statische Fehlervariable)
  - Fehler: Wert an Fehlervariable zuweisen
  - Nachteile:
    - ◆ Test notwendig nach *jeder* Anweisung
    - ◆ Mehrere Fehler: nur der letzte feststellbar
    - ◆ Programmfortsetzung im Fehlerfall möglich
    - ◆ Korrekte Interpretation der Werte per Vereinbarung (unzuverlässig!)
    - ◆ "Normaler" Programmtext aufgebläht mit Prüfungen der Fehlervariablen
- 

#### 8.2.2 Beispiel für Fehlervariable

- Aufgabe: Tage eines Monats bestimmen
- Methode `daysOfMonth` liefert die Tage des Monats mit der gegebenen Nummer zurück. Ignoriert Schaltjahre, d.h. jeder Februar hat 28 Tage.
- Parameter: `m` = Nummer des Monats ab 1 = Januar bis 12 = Dezember.
- Ergebnis: Tage im Monat.

```

int daysOfMonth(int m)
{
    int result = 31;    paßt für die meisten Monate

    errorCode = 0; auf Verdacht    //

    switch(m)
    {
        case 4:        // April
        case 6:        // Juni
        case 9:        // September
        case 11:       // November
            result = 30;
            break;
        case 2:        // Februar
            result = 28;
            break;
        default:
            if(m < 1)
                errorCode = 1;    /1 = Monat-Nr zu klein
            if(m > 12)
                errorCode = 2;    /2 = Monat-Nr zu groß
            break;
    }

    return result;
}

```

- Aufruf der Methode

```

...daysOfMonth(m)...
if(errorCode != 0)
    Fehler/behandeln
else
    Normaler Programmablauf

```

Abfrage notwendig nach *jedem* Aufruf

### 8.2.3 Fluchtwerte

- Für Methoden: "unsinniger" Ergebniswert = Fehler
- Nachteile:
  - ◆ Wert muß gezielt geprüft werden
  - ◆ Programmfortsetzung im Fehlerfall möglich
  - ◆ Fluchtwert nicht immer zu finden §
  - ◆ Korrekte Interpretation der Werte per Vereinbarung (unzuverlässig!)
  - ◆ "Normaler" Programmtext aufgebläht mit Tests

### 8.2.4 Beispiel: Fluchtwert

- Beispiel: Tage eines Monats bestimmen, wie oben.

- Ergebnis: Tage im Monat  
*oder* 0 = Monatsangabe war unzulässig.

```
int daysOfMonth(int m)
{
    int result = 31;    paßt für die meisten Monate

    switch(m)
    {
        case 4:        // April
        case 6:        // Juni
        case 9:        // September
        case 11:       // November
            result = 30;
            break;
        case 2:        // Februar
            result = 28;
            break;
        default:
            if(m < 1)
                result = 0;    Flüchtwert
            if(m > 12)
                result = 0;    Flüchtwert
            break;
    }

    return result;
}
```

- Aufruf der Methode

```
if(daysOfMonth(m) == 0)
    Fehler/behandeln
else
    Normaler Programmablauf
```

Abfrage notwendig nach *jedem* Aufruf

---

## 8.2.5 Exceptions

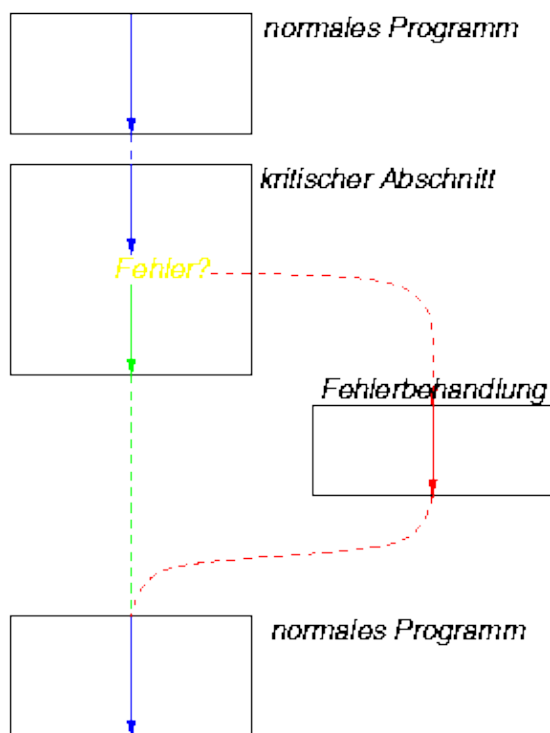
- Idee: Direkter Sprung vom Fehler zur Behandlung
- Vorteile:
  - ◆ Fehler kann nicht irrtümlich als "normales" Verhalten mißverstanden werden
  - ◆ Programm läuft nicht normal weiter
  - ◆ Weder Fluchtwert noch allgemeine Fehlervariable notwendig
  - ◆ Kann nicht ignoriert werden
  - ◆ Normaler Programmtext weitgehend unberührt
- Sprung transportiert ein Objekt,
  - ◆ ...das Einzelheiten zur Fehlerursache enthalten kann

◆ ...dessen Typ die Art des Fehlers klassifiziert

## 8.3 Java–Sprachmittel

### 8.3.1 Ablauf

- Kritische Abschnitte im Code abgrenzen (potentiell Laufzeitfehler)
- Fehlerbehandlung ausgliedern
- **Im Fehlerfall:** Exception auslösen ⇒ Normaler Programmablauf abgebrochen, direkter Fortsetzung mit Fehlerbehandlung
- **Im Normalfall:** Keine Exception ⇒ Normaler Programmablauf, Fehlerbehandlung wird ignoriert
- Skizze:



### 8.3.2 Exception auslösen: throw

- Exception auslösen am Fehlerort: Anweisung "throw"
- throw wirft ein Objekt = Exceptionobjekt
- Syntax allgemein

```
throw Exceptionobjekt
```

- Exceptionobjekt muß Typ `java.lang.Throwable` (oder eine abgeleitete Klasse) haben

### 8.3.3 Beispiel: Exception auslösen

- Exceptionobjekt vom Typ "IllegalArgumentException"
- Beispiel: Tage eines Monats bestimmen, wie oben.
- Parameter: m = Nummer des Monats ab 1 = Januar bis 12 = Dezember.
- Ergebnis: Tage im Monat.

```
int daysOfMonth(int m)
{
    int result = 31;    paßt für die meisten Monate

    switch(m)
    {
        case 4:        // April
        case 6:        // Juni
        case 9:        // September
        case 11:       // November
            result = 30;
            break;
        case 2:        // Februar
            result = 28;
            break;
        default:
            <1) if(m
throw new IllegalArgumentException();
            if(m > 12)
throw new IllegalArgumentException();
            break;
    }

    return result;
}
```

### 8.3.4 Fehlerbehandlung catch-Klausel

- Exception auffangen: "catch-Klausel"
- Syntax §

```
catch(type id)
{
    Code zur Behandlung des Fehlers
}
```

- "Exceptiontyp" *type* passend zum Exceptionobjekt
- Mehrere catch-Klauseln mit *unterschiedlichen* Exceptiontypen
- Erste passende catch-Klausel gilt, Rest wird ignoriert §
- Äußerlich: catch-Klausel ~ Methodenkopf
- Beispiel

```
catch(IllegalArgumentException x)
{
    System.out.println("Unzulässige Monatsangabe");
}
```

### 8.3.5 try-Block

- Abgrenzen eines kritischen Codeabschnittes mit try-Block
- Syntax §

```
try
{
    Anweisungen, die eine Exception auslösen könnten
}
```

- try-Block und catch-Klauseln *unmittelbar nacheinander*
- Beispiel

```
int daysOfMonth(int m)
{
    int result = 31;    paßt für die meisten Monate

    try
    {
        switch(m)
        {
            case 4:    // April
            case 6:    // Juni
            case 9:    // September
            case 11:   // November
                result = 30;
                break;
            case 2:    // Februar
                result = 28;
                break;
            default:
                if(m
<1)
                throw new IllegalArgumentException();
                if(m > 12)
                throw new IllegalArgumentException();
                break;
        }
    }
    catch(IllegalArgumentException x)
    {
        System.out.println("Unzulässige Monatsangabe");
    }

    return result;
}
```

- Keine Exception im try-Block: catch-Klauseln ignoriert

### 8.3.6 finally-Klausel

- Anweisungen für Aufräumarbeiten: "finally"-Klausel

- Syntax

```
finally
{
    abschließende Anweisungen
}
```

- Wie `catch`-Klauseln nach `try`-Block
- *Immer* nach Verlassen des `try`-Blocks
- Es spielt *keine Rolle*, wie der `try`-Block verlassen wird (Exception, `return` oder normaler Ablauf)
- `finally` auch sinnvoll ohne Exceptions

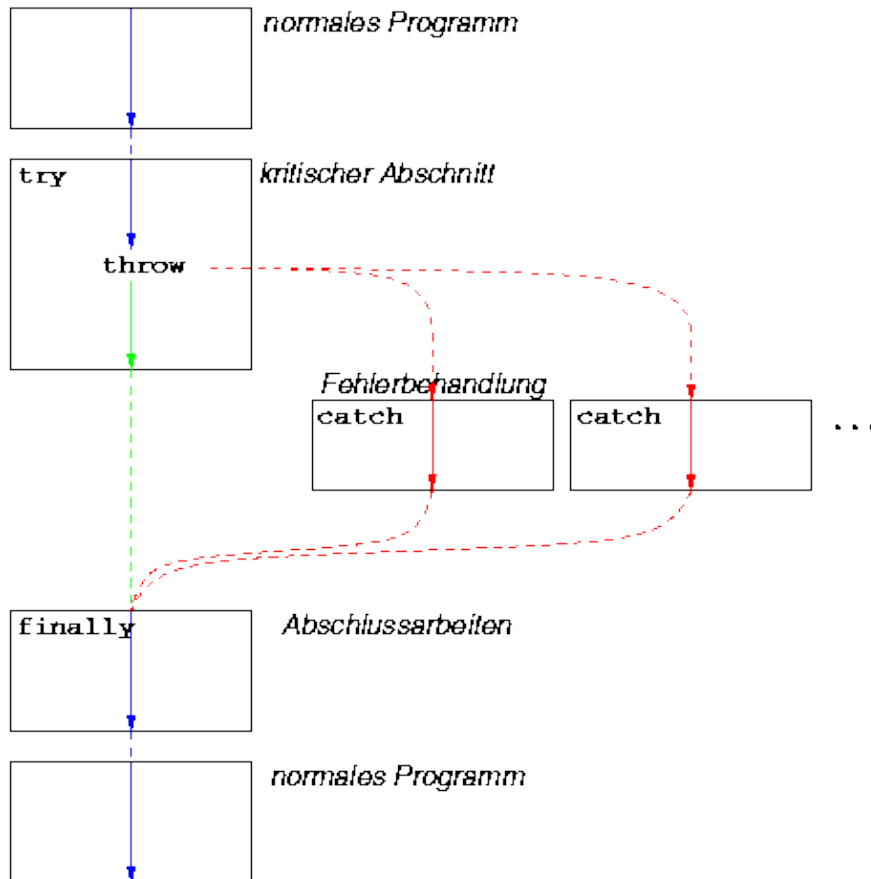
---

### 8.3.7 Schema für Exceptionhandling

- Gleichbleibendes Schema:

```
try
{
    Anweisungen, die eine Exception auslösen könnten
    throw Exceptionobjekt;
}
catch(type1 id)
{
    Code zur Behandlung einer Fehlerart
}
catch(type2 id)
{
    Code zur Behandlung einer anderen Fehlerart
}
...weitere catch-Klauseln...
finally
{
    abschließende Anweisungen
}
```

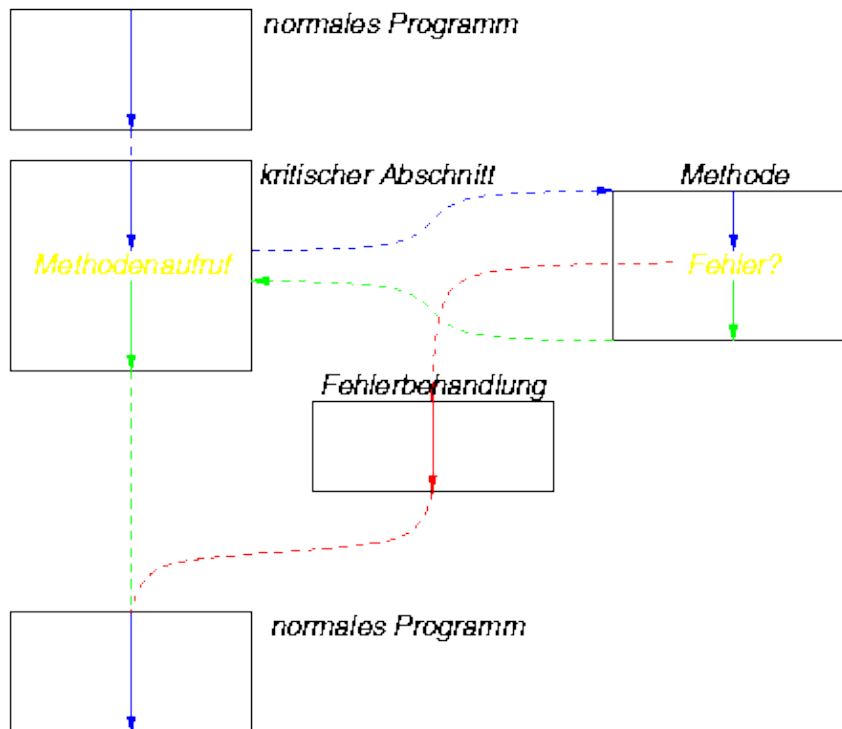
- Als Skizze:



## 8.4 Schachtelung

### 8.4.1 Dynamische Schachtelung

- Aufruf einer Methode, die Exception wirft, aber nicht auffängt: Weiterleiten der Exception an den Aufrufer
- Skizze



- Exception durchsucht Aufrufhierarchie

## 8.4.2 Beispiel

- Methode mit Exceptions:

```
int daysOfMonth(int m)
{
    int result = 31;    paßt für die meisten Monate

    hier kein try/catch!
    switch(m)
    {
        case 4:    // April
        case 6:    // Juni
        case 9:    // September
        case 11:   // November
            result = 30;
            break;
        case 2:    // Februar
            result = 28;
            break;
        default:
            <1)    if(m
throw new IllegalArgumentException();
                if(m > 12)
throw new IllegalArgumentException();
                break;
            }
        return result;
    }
}
```

- Bei Aufrufer:

```

try
{
    int BenutzerEingabe...
    System.out.println(daysOfMonth(m));
}
catch(IllegalArgumentException x)
{
    System.out.println("Unzulässige Monatsangabe");
}

```

### 8.4.3 Geschachtelte try-Blöcke

- Mehrere *geschachtelte* try-Blöcke
- Exception sucht von *innen nach außen*
- *Erste passende* catch-Klausel (Exceptiontyp) wird gewählt
- Weitere passende catch-Klauseln weiter außen *werden ignoriert*
- Beispiel: Methode wirft (immer) Exception

```

void bombOut()
{
    throw new IllegalArgumentException();
}

```

Aufrufe in geschachtelten try-Blöcken:

```

try
{
    try
        {
            bombOut() // innerer Catch/
        }
    catch(IllegalArgumentException x)
        {
            System.out.println("inneres Catch");
        }
    bombOut() // zum äußeren Catch
}
catch(IllegalArgumentException x)
{
    System.out.println("äußeres Catch");
}

```

### 8.4.4 Programmabbruch

- Exception *ohne catch* (fehlt ganz oder kein passender Typ) ⇒ **Programmabbruch**
- Beispiel:

```

void bombOut()
{
    throw new IllegalArgumentException();
}

```

Aufruf ohne try/catch:

```

public static void main(String[] args)

```

```
{
    bombOut();
}
```

- Programm starten:

```
java.lang.IllegalArgumentException
    at Bomb.bombOut(Bomb.java:4)
    at Bomb.main(Bomb.java:9)
```

Ausgaben mit Angabe von *Exception*typ, *Methode*, *Datei*, *Zeilennummer* §

---

## 8.5 Exceptionsignaturen

- Zwei Alternativen zum *Umgang* mit Exceptions:
  1. *Behandeln* (try/catch)
  2. *Weitergeben* an Aufrufer
- Zweiter Fall: Exceptions *deklarieren* = "*Exceptionsignatur*"
- Syntax

```
Methodenname(Parameterliste) throws Exceptiontyp1, Exceptiontyp2, ...
```

- Exceptionsignatur *gleichrangig* zu Ergebnistyp und Parameterliste §
  - *Java-Compiler prüft* Exceptionsignaturen (Anwesenheit und Korrektheit)
  - Fehlende oder fehlerhafte Exceptionsignaturen: Programm wird *nicht übersetzt*
- 

## 8.6 Exceptionklassen

---

### 8.6.1 Throwables

- Throwable = *vordefinierte Basisklasse* für alle Exceptionobjekte
- Von Throwable sind direkt zwei Klassen abgeleitet
  - Error*
    - Interne Fehler* des Java-Interpreters, die nicht behandelt werden können. M.a.W. es gibt keine sinnvolle Gegenmaßnahme.
    - Benutzercode sollte keine Error-Objekte werfen.

*Exception*

Gemeinsame Basisklasse für alle "*normalen*" *Fehlersituationen* (sofern eine Fehlersituation normal sein kann)

- Von Exception sind wiederum mehrere Klassen abgeleitet.
- Eine besondere Rolle spielen dabei RuntimeExceptions: Sie können praktisch überall auftreten.
- Für Error und RuntimeException brauchen *keine Exceptionsignaturen* angegeben werden, weil sie praktisch in jedem Codefragment auftreten könnten §

- `Error` und `RuntimeException` können (wie alle `Exceptions`) mit `catch`-Klauseln behandelt werden

## 8.6.2 Elemente der vordefinierten Exceptionklassen

- Exceptionobjekte enthalten **Klartextmeldung** über die Fehlerursache
- `Throwable` definiert einen passenden **Konstruktor**

```
Throwable(String)
```

- Abruf der Klartextmeldung mit

```
String Throwable.getMessage()
```

abgefragt werden

- Weitere nützliche Methoden sind

```
void printStackTrace()
void printStackTrace(PrintStream s)
```

Drucken **Aufrufhierarchie** zum Zeitpunkt des `throw` aus

## 8.6.3 Benutzerdefinierte Exceptionklassen

- Für eigene `Exceptions`
  - ◆ Passende **vordefinierte Klasse** suchen (nicht gerade eine `RuntimeException`) oder...
  - ◆ **Neue Exceptionklasse** definieren
- Beispiel für neue `Exception`klasse:

```
class ObscureFailure extends Exception
{
    public ObscureFailure()
    {}

    public ObscureFailure(String s)
    {
        super(s);
    }
}
```

Im Anwendungsprogramm:

```
...
try
{
    ...throw new ObscureFailure();...
}
catch(ObscureFailure x)
{
    System.out.println("Seltsame Dinge gehen vor...");
}
```

- Oft *weder Datenelemente noch Methoden* (erbt von `Exception`)

## 9 Packages

Siehe auch: [\[The Java Language Specification\]](#)

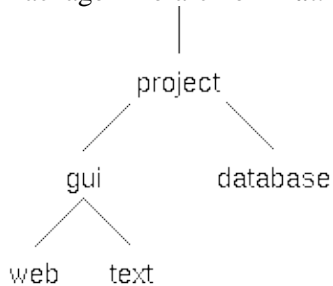
### 9.1 Arbeitsweise

#### 9.1.1 Motivation

- *Bytecodedatei* = Bausteine von Javaprogrammen
- 1 Bytecodedatei = 1 Klassendefinition
- Zweckmäßig auch: 1 Sourcecodedatei = 1 Klassendefinition §
- Problem: *Orientierung* (zu viele Klassen)
- Problem: *Namenskonflikte* (verschiedene Klassen mit gleichen Namen)
- Vergleichbar mit Dateien im Filesystem
- Fazit: Klassen müssen *organisiert* werden

#### 9.1.2 Packages

- *Package* = Menge logisch zusammengehörender Klassen
- Klassen mit *gleichen Namen* in *unterschiedlichen Packages* kollidieren *nicht*.
- Packages benannt mit *Java-Identifiern*
- Packages lassen sich schachteln, d.h. ein Package kann *Subpackages* enthalten, dieses wieder Subpackages usw.
- Package-Hierarchie  $\approx$  *Baumstruktur*. Beispiel:



Packagenamen:  
 project  
 project.gui  
 project.gui.web  
 project.gui.text  
 project.database

- Innerhalb eines Packages: Alle Namen *eindeutig* (Klassen und Subpackages)

- Jede Klasse ist Element *genau eines Packages*.
  - Geschachtelte Packages nur dem Namen nach verwandt, technisch alle Packages gleichrangig. <sup>§</sup>
- 

### ▶ 9.1.3 Dynamisches Laden von Bytecode

- JVM lädt Bytecode *nach Bedarf* (*on-the-fly*)
  - Forderung: Bytecode *schnell und gezielt* lokalisierbar <sup>§</sup>
  - Lösung: *Package* einer Klasse fixiert *Pfad im Filesystem* <sup>§</sup> ⇒ JVM findet Bytecode sofort <sup>§</sup>
- 

### ▶ 9.1.4 Abbildung ins Filesystem

- Idee: Packages 1:1 abbilden auf Directories
- Konsequenz: geschachtelte Packages entsprechen geschachtelten Directories
- Frage: Wo liegt die *Wurzel* des Baumes?
- Umgebungsvariable "CLASSPATH" legt Startpunkt fest <sup>§</sup>
- Komplette Pfadangabe = CLASSPATH + Packagepfad + Klassenname
- Beispiel:

```
CLASSPATH = /home/developer
Package =    project.gui.web
Klassenname =  IndexPage
Zusammen:    /home/developer/project/gui/web/IndexPage.class
```

- CLASSPATH kann *mehrere alternative Startpunkte* nennen. <sup>§</sup> Werden nacheinander durchsucht bis zum ersten Treffer.
- 

## ▶ 9.2 Umgang mit Packages

---

### ▶ 9.2.1 Qualifizierte Namen

- *Qualifizierte Namen* bezeichnen Elemente aus fremden Packages (Klassen, Methoden, Datenelemente, ...)
- Qualifizierter Name = *Packagepräfix* + *Java-Identifizier*
- *Punkt* (.) als *Trennzeichen* zwischen Namensbestandteilen Beispiel:

```
project.gui.web.IndexPage
```

- Qualifizierte Namen *gleichwertig* zu anderen Identifiern. Beispiel:

```
project.gui.web.IndexPage p = new project.gui.web.IndexPage();
```

## 9.2.2 import-Klausel

- Häufiger Einsatz qualifizierter Namen mühsam, unleserlich, fehlerträchtig
- Abhilfe: Qualifizierten Namen *1× importieren*, dann *ohne Packagepräfix verwenden*
- Syntax **import-Klausel**

```
import package1.package2...packagen.class;
```

- Beispiel:

```
project.gui.web.IndexPage p = new project.gui.web.IndexPage();
```

mit import-Klausel:

```
import project.gui.web.IndexPage;
...
IndexPage p = new IndexPage();
```

- import-Klausel muß am *Anfang des Quelltexts* stehen, vor Klassendefinitionen
- **Mehrere import-Klauseln** nacheinander ok
- **Alle Namen** gemeinsam importieren §

```
import package1.package2...packagen.*;
```

- Eine import-Klausel betrifft nur den Compiler, nicht die JVM (deshalb "Klausel" und nicht "Anweisung")

## 9.2.3 package-Klausel

- import-Klausel regelt Zugriff auf fremde Namen
- Gegenstück: **package-Klausel** regelt die eigene **Package-Zugehörigkeit** einer Klasse
- Syntax:

```
package package1.package2...packagen;
```

- package-Klausel **nur 1× pro Datei**
- package-Klausel **als erstes im Quelltext** (noch vor den import-Klauseln).
- Eine Java-Quelltextdatei ist schematisch wie folgt aufgebaut:

```
Package-Klausel
Import-Klauseln...
```

## 9.2.4 Default–Package

- Frage: Wo liegen Klassen *ohne Packageangabe*?
- Kein Packagepräfix: anonymes Package = lokales Package = *Default–Package*
- Abbildung ins Dateisystem: CLASSPATH ohne Fortsetzung
- *Import* aus dem Default–Package *nicht möglich*

## 9.2.5 Namenskollisionen

- Problem: Nutzung gleicher Namen aus verschiedenen Packages
- Import noch ok, aber *Benutzung* liefert *Fehler*
  1. Fall: Lokales/fremdes Package.  
Beispiel: Datei other/Thing.java:

```
package other;

public class Thing{}
```

Datei Thing.java:

```
import other.*;

class Thing{

    ..new Thing()...           // ok, lokales Thing
    ..new other.Thing()...    // ok, importiertes Thing
```

Name ohne Qualifier gebunden an lokale Definition

2. Fall: zwei fremde Packages.

Beispiel: Datei other/Thing.java:

```
package other;

public class Thing{}
```

Datei strange/Thing.java:

```
package strange;

public class Thing{}
```

Irgendeine andere Datei:

```
import other.*;
import strange.*;

..new Thing()...           // Fehler!
..new other.Thing()....    // ok
..new strange.Thing()....  // ok
```

Mehrdeutigkeit ⇒ Fehler

## 9.2.6 Assertions

- Assertions lassen sich aktivieren und stilllegen

- Bisher: Alle Assertions

```
$ java -ea Program
```

oder in einzelnen Klassen

```
$ java -ea:MyClass Program
```

- Zusätzlich: für alle Klassen eines Package samt Sub-Packages

```
$ java -ea:SomePackage... Program
```

Die drei Punkte unterscheiden Klassen von Packages

- Anonymes Default-Package ansprechen mit . . . alleine:

```
$ java -ea:... Program
```

- Ebenso Assertions stilllegen mit `-da`, auch Klassen und Packages gemischt

## 9.3 Zugriffsrechte

- Bisher Zugriffsrechte public, private, protected
- Jetzt vierte Stufe: Keine Angabe § = **Zugriffsrecht "package"** §
- "package" bedeutet:
  - ◆ frei verfügbar für alle Klassen *im gleichen Package*, aber
  - ◆ gesperrt für *andere Packages*.
- "package" zwischen `public` und `private`.
- Nachtrag: **protected** *schließt "package"* mit ein
- Vollständige Liste der Zugriffsrechte:
  - public*  
Keine Einschränkung.
  - protected*  
Innerhalb des eigenen Package und  
in allen abgeleiteten Klassen, unabhängig vom Package.
  - package*  
Innerhalb des eigenen Package.
  - private*  
Innerhalb der Klasse.

## 9.4 Archivdateien

### 9.4.1 Zweck

- Archivdateien *alternative Organisationsform* für Bytecode, gegenüber Directorybaum.
- Formate: "*Zip*" und "*Jar*", weitgehend gleich § §

- Extensions *per Konvention* .zip und .jar
- *Vorteile* beispielsweise:
  - ◆ Leicht zusammen zu halten
  - ◆ Effizient übertragbar, besonders über langsame Medien (Internet)
  - ◆ Digital signierbar
- *Packagehierarchie* bleibt innerhalb einer Archivdatei erhalten
- JVM findet Bytecode auch in *Archivdateien*
- CLASSPATH kann komplette Archivdatei nennen, beispielsweise:

```
CLASSPATH=/usr/java/classes/rt.jar
```

## 9.4.2 Jar-Archive und das jar-Tool

- Java SDK enthält das Werkzeug "jar"
- Kommandozeilenwerkzeug ohne graphische Oberfläche. §

- Wichtige Aufrufe:

### *Erzeugen*

Die Datei *jarfile* wird neu erstellt und enthält die nachfolgend genannten Bytecodedateien. In den meisten Betriebssystemen können Jokerzeichen benutzt werden.

```
jar cf jarfile bytecode1 bytecode2 bytecode3 ...
```

### *Inhaltsverzeichnis*

Die erste Form liefert nur Pfadnamen. In der zweiten Form wird zusätzlich Information über Größe und Zeitmarken mit ausgegeben.

```
jar tf jarfile
jar tvf jarfile
```

### *Auspacken*

In der ersten Form werden alle Bytecodedateien aus *jarfile* extrahiert und samt der ursprünglichen Directoryhierarchie wieder restauriert. In der zweiten Form wird nur die angegebene Bytecodedatei herausgeholt. Das *jarfile* wird nicht verändert.

```
jar xf jarfile
jar xf jarfile bytecodex
```

### *Bedienungsanleitung*

```
jar
```

- Beispiel für Aufrufe von jar sind:

```
jar cf project.jar project/gui/*.class # c = create
jar tf project.jar # t = table
jar xf project.jar # x = extract
```

## 10 Programmierstil

Siehe auch Java Code Conventions

---

## ▶ 10.1 Wozu Programmierrichtlinien?

- Nach allgemein akzeptierten Schätzungen fließen etwa **80%** der Gesamtkosten im Softwarebereich die **Wartung**
  - Einheitliche **Programmierrichtlinien** helfen, diesem Ungleichgewicht entgegenzuwirken.
  - Programmierrichtlinien sollen:
    - ◆ die **Lesbarkeit** von Quelltext verbessern, insbesondere für andere Verantwortliche als den ursprünglichen Autor,
    - ◆ den **Austausch** von Quellen erleichtern, z.B. zwischen verschiedenen Mitgliedern eines Entwicklerteams,
    - ◆ die Qualität von Software steigern durch Förderung von systematischem und **planmäßigem Programmieren**.
  - Das Hauptproblem liegt in der vom Einzelnen geforderten **Selbstdisziplin**. Es gibt kein Werkzeug, das Verstöße gegen Programmierrichtlinien erkennt und abweist (wie etwa der Compiler Syntaxfehler abweist)
- 

## ▶ 10.2 Programmstruktur

- Definieren Sie nur **eine einzige Klasse** (oder Interface) pro Quelltextdatei, und sei sie noch so kurz und unscheinbar.
  - Ordnen Sie Elemente innerhalb einer Klasse nach einer **festen Abfolge**, z.B.:
    1. öffentlich Konstanten (alle "public final static")
    2. Konstruktoren
    3. Auskunftsmethoden ("Reader")
    4. Änderungsmethoden ("Writer")
    5. private Hilfsmethoden
    6. Datenelemente (alle "private" und zusätzlich möglichst "final")
  - Definieren Sie in jeder Klasse die Methode "toString()"
- 

## ▶ 10.3 Layout

- Schreiben Sie Quelltext überwiegend bis zu 80 Zeichen breit (= 1 Bildschirmbreite), brechen Sie längere Zeilen um.
  - Verwenden Sie **keine Umlaute** und anderen nationalen Sonderzeichen in Bezeichnern.
  - Rücken Sie konsistent ein (z.B. immer 4 oder immer 8 Spalten pro Block).
  - Verwenden Sie freizügig **Zwischenraum und Leerzeilen**.
- 

## ▶ 10.4 Namen

- Benutzen Sie **englische Bezeichner** und keine deutschen.

- Schreiben Sie Wörter aus, kürzen Sie möglichst wenig ab.
- Überladen Sie Variablenamen nicht mit Typkürzeln und anderen technischen Einzelheiten.
- Verwenden Sie Namen mit einem Buchstaben nur in begrenztem Kontext, wie z.B. als Zähler in einer kurzen Schleife:

```
for(int i = 0; ...)
...kurzer Rumpf...
```

- Wählen Sie Namen für Methoden und Datenelemente immer so, daß ihre Bedeutung **bezüglich der Klasse eindeutig** ist. Wenn z.B. die Länge eines Vektors ausgelesen werden soll, dann reicht `getLength` oder nur `length` anstelle von `getLengthOfVector`

## ▶ 10.5 Groß- und Kleinschreibung von Namen

JavaSoft-Konventionen für Groß- und Kleinschreibung:

### *Klassen und Interfaces*

Hauptwörter mit großem Anfangsbuchstaben und kleiner Fortsetzung. Zusammengesetzte Wörter mit jeweils großem Anfangsbuchstaben im Wort. Beispiele:

```
Image
FileInputStream
PolarComplex
```

Benennen Sie, was Objekte der Klasse repräsentieren.

### *Methoden*

Verben mit kleinen Anfangsbuchstaben und passenden Zusätzen. Beispiele:

```
loadHeader
normalize
expandToMaxWidth
```

Benennen Sie, was die Methode abwickelt oder bewirkt.

### *primitive Zugriffsmethoden*

(Reader und Writer) Überladen Sie den Attributnamen ohne weitere Zusätze. Beispiele:

```
double real()
void real(final double r)
```

Diese Vorgabe deckt sich nicht mit den JavaSoft-Richtlinien, die Vorsätze "get..." und "set..." empfehlen, also in diesem Beispiel:

```
double getReal()
void setReal(final double r)
```

### *Datenelemente*

Substantive mit kleinem Anfangsbuchstaben. Beispiele:

```
real
imag
```

### *Öffentliche Konstanten*

Komplett in großen Buchstaben, Wortteile getrennt mit Underscore. Beispiele:

```
PI
MAX_WIDTH
```

## ▶ 10.6 Variablen

- Plazieren Sie nur eine Definition pro Zeile, nur eine Variable pro Definition. Also besser

```
int first;
int[] rest;
double average;
```

statt

```
int first, rest[]; double average;
```

- Definieren Sie Namen im kleinsten möglichen Block, aber dort am Anfang und nicht später zwischen anderen Anweisungen. §
- Initialisieren Sie Variablen möglichst schon bei der Definition.
- Erklären Sie Datenelemente in Klassen als "final" wo immer möglich.

## 10.7 Methoden

- Benutzen Sie keine Numerale, sondern ersetzen Sie diese immer durch **benannte Konstanten**.
- **Klammern Sie Teilausdrücke** in komplexen Gesamtausdrücken, auch wenn das formal unnötig wäre.
- Verwenden Sie die **angemessenen primitiven Typen** im passenden Kontext. Benutzen Sie z.B. nicht 0 für logisches "Falsch" und 1 für "Wahr".
- Beschränken Sie Methodenrumpfe auf **höchstens 20–30 Anweisungen**. Spalten Sie eine Methoden lieber in mehrere kürzere Methoden auf. Davon ausgenommen sind Methoden mit sehr regelmäßigem Aufbau.
- Schachteln Sie Blöcke nicht tiefer als **4 oder 5 Ebenen**. Spalten Sie besser private Hilfsmethoden ab, die Teilaufgaben erledigen.
- Behandeln Sie Fehler ausschließlich mit **Exceptions** und planen Sie das von Anfang an mit ein.
- Nutzen Sie das Angebot der **Laufzeitbibliothek**, und erfinden Sie das Rad nicht jedesmal neu.
- Analysieren Sie **fremden Code** und passen Sie ihn an Ihre Anforderungen an. (Vermeiden des "not-invented-here"-Syndroms, oder anders gesagt: Andere Leute sind auch nicht dumm.)

## 10.8 Klassen

- Definieren Sie Datenelemente grundsätzlich **"private"**.
- Erklären Sie Methoden überwiegend als **package-lokal** (= ohne Zugriffsangabe), geben Sie nur das Minimum als "public" frei.
- Klassendefinitionen sind keine große Sache. Definieren Sie auch für scheinbar triviale Strukturen **neue Klassen**.
- Reduzieren Sie **static-Methoden auf das Minimum**.
- Überlegen Sie sich vorher eine detaillierte **Packagestruktur**

## 10.9 Kommentare

- Hier geht es um "normale" Kommentare, nicht um Doc-Kommentare.
- Vertun Sie keine Zeit mit Schmuckbordüren (diese sind zu umständlich zu pflegen)
- Erklären Sie Ihre Programmlogik nicht im Kommentar (programmieren Sie lieber klarer).
- Kommentieren Sie *keine Trivialitäten*, wie z.B.:

```
int a = 1;      // hier wird a deklariert und mit 1 initialisiert
```

- Beginnen Sie jede Datei mit einem einheitlichen **Kopf**, der nennt:
  - ◆ Organisation
  - ◆ Autor
  - ◆ Projekt gesamt
  - ◆ Dateiname
  - ◆ Compiler, Plattform

---

## ▶ 11 Dokumentation

Siehe auch: [\[The Java Language Specification\]](#)

---

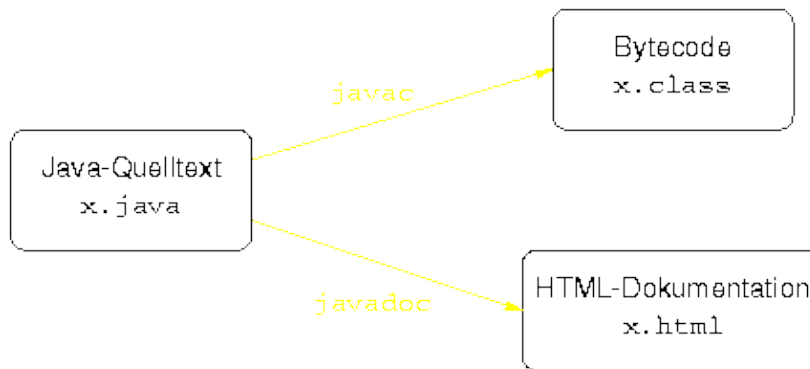
### ▶ 11.1 Doc-Kommentare

- In der Praxis erweist sich *konsistente Dokumentation* (Dokumentation die zum aktuellen Stand des Quelltextes paßt) als problematisch. Eine der Hauptursachen sind die getrennte Verwaltung von Dokumentation und Quellen.
- In Java ist der Weg vom Quelltext zu konsistenter Dokumentation bequem. Dokumentation wird nicht separat erstellt, sondern direkt *aus Quelltext erzeugt*. Der Entwickler muß nur *einen einzigen Text* bearbeiten.
- Der Schlüssel sind "*Doc-Kommentare*", die in Java fest integriert sind.
- Doc-Kommentare erweitern die Syntax normaler Blockkommentare:

```
/**  
:  
*/
```

Entscheidend ist der zweite Stern.

- Das **Werkzeug javadoc** extrahiert Doc-Kommentare aus Java-Quelltext.
- javadoc generiert aus Doc-Kommentaren automatisch Dokumentation in Form von **Webseiten** (HTML-Dateien).



- Für den Javacompiler `javac` sind Doc-Kommentare nicht von Blockkommentaren zu unterscheiden.
- Doc-Kommentare werden verfaßt zu
  - ◆ Klassen und Interfaces
  - ◆ Methoden
  - ◆ Datenelementen
- Alle Doc-Kommentare haben eine *einheitliche Gliederung*:

```

/** 1. Kurze Erklärungen im Klartext mit Punkt am Ende.
 * 2. Weitere Einzelheiten im zweiten und den folgenden Sätzen.
 *    Nur der erste Satz wird in die Zusammenfassung übernommen.
 * 3. Liste beliebig vieler Tags (auch keine)
 */
  
```

- Weitere Einzelheiten auf der [Javadoc-Homepage](#) von Javasoft

## 11.2 Tags

- **Tags** markieren Bestandteile mit fester Bedeutung innerhalb von Doc-Kommentaren.
- Alle Tags beginnen mit einem **@-Zeichen** und einem **Schlüsselwort**, gefolgt von zusätzlichen Angaben. Das @-Zeichen jedes Tags muß (abgesehen von Sternen) als erstes Zeichen in einer eigenen Zeile stehen.
- Für jede Art von Definition gibt es bestimmte Tags:
  - Klassen und Interfaces*
    - @author
    - @version
  - Methoden*
    - @param
    - @return
    - @exception
  - Datenelemente*
    - keine Tags
- Die zusätzlichen Informationen hängen von der Art des Tags ab.
  - @author

(mehrfach) Name des Autors, evtl. als HTML-Link mit mailto-URL. Beispiel:

```
@author <a href=no7@grossefreiheit.de>Hans Albers</a>
```

*@version*

(nur einmal) Datum (im europäischen Standardformat YYYY-MM-DD) und Versionsnummer. Beispiel:

```
@version 1998-03-01 1.3
```

*@param*

(mehrfach) Name und Beschreibung, kein Typ. Beispiel:

```
@param maxWidth Obergrenze fuer die Bildbreite.
```

*@return*

(nur einmal) Beschreibung ohne Typangabe. Beispiel:

```
@return Laenge der Bilddaten in Byte.
```

*@exception*

(mehrfach) Exceptiontyp und Beschreibung der Ursache. Beispiel:

```
@exception InvalidAction Aktion war im aktuellen Spielstand nicht erlaubt.
```

## ▶ 11.3 Generator javadoc

- Das Werkzeug javadoc ist Teil des JDK. Es wird von der Kommandozeile aufgerufen:

```
javadoc {schalter} java-Dateien...
```

- Schalter sind

*-author*

überträgt auch das @author-Tag in die generierten Webseiten (Voreinstellung: wird ausgelassen).

*-version*

überträgt auch das @version-Tag in die generierten Webseiten (Voreinstellung: wird ausgelassen).

*-d pfad*

Zielpfad für generierte Dateien (Voreinstellung: Arbeitsdirectory)

*-package*

überträgt auch package-lokale Elemente in die generierten Webseiten (Voreinstellung: public- und protected-Elemente).

*u.a.*

- Die generierten Webseiten sind mit jedem Webbrowser lesbar §

## ▶ 12 I/O (Ein- und Ausgabe)

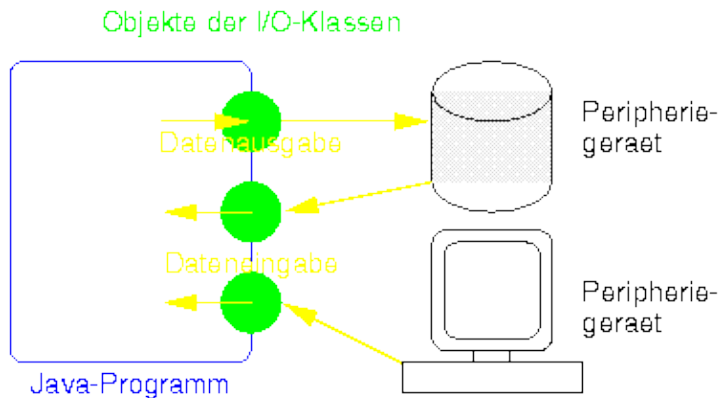
Siehe auch: [\[The Java Language Specification\]](#)

### ▶ 12.1 Standard-I/O

#### ▶ 12.1.1 Problematik

- I/O (= Input/Output): Mechanismen zum Datenaustausch zwischen Programm und Umgebung
- "Umgebung" vielgestaltig: Tastatur, Bildschirm, Datei, Netzwerk, anderes Programm, ...

- Hauptproblem: Kaum verbindliche Annahmen über die Umgebung möglich §
- Bestimmte Java-Objekte als Vermittler "nach außen":



- Vorteil dieser Trennung: Systemabhängigkeiten isoliert in wenigen Klassen (eben diesen I/O-Klassen), ⇒ der größte Teil des Programms unabhängig von speziellen Systemeigenschaften

## ▶ 12.1.2 Standard-I/O

- Einfachster I/O-Mechanismus: Text-Austausch via Tastatur und Bildschirm
- Bezeichnungen: "Standard-I/O" oder "Konsol-I/O"
- Minimale Systemanforderungen ⇒ auf praktisch jedem System verfügbar. §
- Standard-I/O über fest vordefinierte Objekte: §

<i>Objekt</i>	<i>Bezeichnung</i>	<i>Bindung</i>
<code>System.in</code>	Standard-Eingabe	Tastatur
<code>System.out</code>	Standard-Ausgabe	Bildschirm
<code>System.err</code>	Fehlerausgabe	Bildschirm

- Die voreingestellte Zuordnung (Tastatur resp. Bildschirm) läßt sich aufheben via "I/O-Redirection" (= Ein-/Ausgabe-Umlenkung). § Diese Umlenkung läuft außerhalb von Java ab und ist für ein Javaprogramm nicht erkennbar.
- `System.in` usw. = statische, öffentliche (`public`) Datenelemente der Klasse `System`.

## ▶ 12.2 Byteströme

### ▶ 12.2.1 Abstrakte Basisklassen

- **Bytestrom** = lineare Bytefolge = Sequenz isolierter Bytes
- Basis aller weiteren I/O-Mechanismen, oft nicht wahrnehmbar
- Alle Bytes eines Bytestroms (aus der Sicht des Bytestroms) gleichwertig ⇒ keine Interpretation,

strukturlos

- Byteströme auf praktisch allen Systemen verfügbar ⇒ Datenaustausch zwischen verschiedenen Systemen und Programmen §
- Java-Klassen für Byteströme im Package "java.io":

<i>Klasse</i>	<i>Zweck</i>
<u>InputStream</u>	Eingabestrom, liefert Bytes
<u>OutputStream</u>	Ausgabestrom, akzeptiert Bytes

- Abstrakte Basisklassen ⇒ isoliert nutzlos

## ▶ 12.2.2 Methoden zur Ein- und Ausgabe

- **Elementare Methoden:**

<i>Klasse</i>	<i>Methode</i>	<i>Zweck</i>
InputStream	<u>int read()</u>	liefert das nächste Byte
OutputStream	<u>void write(int b)</u>	gibt das Byte b aus

- read liefert -1, wenn die Eingabe erschöpft ist. §
- read blockiert, bis Daten zur Verfügung stehen
- Methode zum Testen, wieviele Bytes gelesen werden *könnten*:

<i>Klasse</i>	<i>Methode</i>	<i>Zweck</i>
InputStream	<u>int available()</u>	liefert die Anzahl verfügbarer Bytes

- available *liest nichts*, prüft nur
- Methode close() schließt einen Stream ⇒ vor allem notwendig bei OutputStreams

## ▶ 12.2.3 Beispielprogramm: Standard-I/O kopieren

- InputStream und OutputStream abstrakt ⇒ abzuleiten in konkrete Klassen
- Beispiele für Objekte abgeleiteter Klassen: `System.in` und `System.out`
- Kopieren über primitive Methoden `read` und `write`
- Programm CopyByteStream.java

## ▶ 12.3 I/O-Medien

## 12.3.1 Medien

- Konkrete Klassen, paarweise von `InputStream` resp. `OutputStream` abgeleitet
- Jeweils bestimmte Datenquelle bzw. bestimmtes Datenziel
- Beispiele:

<i>Klassen</i>	<i>Quelle bzw. Ziel</i>
<code>FileInputStream</code> <code>FileOutputStream</code>	Datei im Filesystem
<code>ByteArrayInputStream</code> <code>ByteArrayOutputStream</code>	Array von Bytes
<code>PipedInputStream</code> <code>PipedOutputStream</code>	Anderer Thread

- Gemeinsame (ererbte) Eigenschaften
  - ◆ Transportieren Bytes
  - ◆ Bieten Methoden `read`, `write`, `available` §

## 12.3.2 File-I/O

- Beispiel für Klassen mit einem *konkreten Ein-/Ausgabe-Ziel*: **externe Dateien** ("extern" aus der Sicht des Javaprogramms)
- "File-I/O" = Sammelbegriff für jede Art von Ein- und Ausgabe vom/zum Filesystem.
- Java-Klassen für File-I/O

<i>Klasse</i>	<i>Zweck</i>
<code>FileInputStream</code>	Liest Bytes von einer Binärdatei
<code>FileOutputStream</code>	Schreibt Bytes auf eine Binärdatei

- Objekte dieser Klassen repräsentieren Dateien
- Dateinamen im Konstruktor:

```
FileOutputStream fos = new FileOutputStream("results.data");
```

## 12.3.3 Beispielprogramm: Binärdatei kopieren

- Quelltext `CopyFile.java`
- Gegenüber dem Kopieren von Standard-I/O: Nur Konstruktoraufrufe verändert
- Beobachtung: Die Performance des Beispielprogramms ist verhältnismäßig schlecht §

### 12.3.4 Block-I/O

- Methoden `int read()` und `write(int)` transportieren *einzelne Bytes* ⇒ ineffizient
- Basisklassen bietet zweiten Satz I/O-Methoden für *Byte-Arrays*

Klasse	Methode	Zweck
<code>InputStream</code>	<code>int read(byte[] b)</code>	liest Bytes und speichert sie in b
<code>OutputStream</code>	<code>void write(byte[] b)</code>	gibt die Bytes in b aus
<code>OutputStream</code>	<code>void write(byte[] b, int from, int count)</code>	gibt count Bytes aus b aus ab Index from

- Ergebnis: `read` liefert die Anzahl tatsächlich gelesener Bytes oder `-1`, wenn das Ende der Eingabe erreicht ist
- Fehler: Beide Methoden werfen `IOExceptions`, wenn die Operation scheitert

### 12.3.5 Beispielprogramm: Binärdatei blockweise kopieren

- Array-I/O: Anzahl I/O-Operationen sinkt
- Jede einzelne I/O-Operation transportiert mehr Daten
- Quelltext [CopyFileBlocked.java](#)
- Wahl einer günstigen Arraygröße systemabhängig
- Zwei Effekte mit gegensätzlicher Wirkung:
  1. Kosten pro Byte: Sinkt bei steigender Puffergröße
  2. Kosten zum Allokieren des Puffer-Arrays: Steigt mit Puffergröße
 Typisch: Minimum bei 2–8 kByte

### 12.3.6 Adaptive Blockgrößen

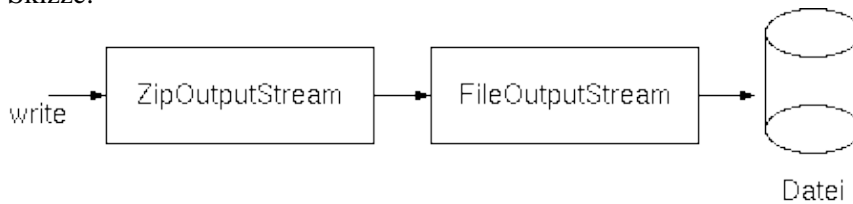
- Blockgröße offen lassen, zur Laufzeit festlegen
- Auskunftsmethode `available()` steuert Blockgröße
- Quelltext [CopyFileAvailableBlocked.java](#)
- Reduziert `read/write`-Aufrufe auf das Minimum

## 12.4 Filter

### 12.4.1 Idee

- *Filter* = Objekte zur *Manipulation* durchfließender Daten

- *Keine* konkrete Datenquelle oder konkretes Datenziel (wie oben)
- Als "Vorbrenner" (oder "Nachbrenner") an bereits existierenden Stream gekoppelt
- Skizze:



- Aus Objektsicht: *Zwei* gekoppelte Objekte, die nacheinander arbeiten

## 12.4.2 Einordnung in Klassenhierarchie

- Klassen FilterInputStream und FilterOutputStream: Gemeinsame **Basisklassen** für alle Arten von Filtern
- Selbst von Input/OutputStream abgeleitet
- FilterInput/OutputStream *enthalten* einen anderen Stream als Datenelement, der gefiltert werden soll
- Vorsicht *Verwirrungsgefahr*:
  1. FilterStreams sind von den ABCs Input/OutputStream *abgeleitet*  
⇒ gleiche Methoden, einheitliche Verwendung, zur Compilezeit relevant und
  2. FilterStreams *enthalten* jeweils ein Input/OutputStream-Objekt  
⇒ Abnehmer bzw. Lieferant für (ungefilterte) Daten, zur Laufzeit relevant

## 12.4.3 Beispiele

- Einige vordefinierte Filterstreams in der Laufzeitbibliothek

<i>Klasse</i>	<i>Zweck</i>
BufferedInputStream	liest Eingaben in Blöcken, Effizienzgewinn
PushbackInputStream	kann bereits gelesene Eingaben zurückschieben, können noch einmal gelesen werden
ZipInputStream	expandiert komprimierte Eingaben

- Weitere Filterstreams neu zu definieren

## 12.4.4 Gepufferte Ströme

- BufferedInputStream und BufferedOutputStream: implementieren blockweises Lesen und Schreiben
- Ziele wie bei I/O von Arrays (statt Bytes): Effizienzgewinn
- Vorteil: Blockung unsichtbar, Benutzercode entlastet

- Nachteil: Weniger wirksam als explizite Blockung: `read/write`-Aufrufe unverändert pro Byte
- Quelltext [CopyFileBuffered.java](#)

## 12.4.5 Komprimierte Ströme

- Vordefinierte Kompressionsarten "Zip", "GZip" (GNU-Zip), "Jar"
- Enthalten im Package `java.util.zip`.
- `InflaterInputStream` und `DeflaterOutputStream`: Gemeinsame **Basisklassen** für komprimierende/expandierende Streams
- Von `FilterInput/OutputStream` abgeleitet
- Konkrete Klassen:

<i>Klasse</i>	<i>Zweck</i>
<code>GZIPInputStream</code>	expandiert GZip-komprimierte Eingabe
<code>GZIPOutStream</code>	komprimiert Ausgabe mit GZip
<code>ZipInputStream</code>	expandiert Zip-komprimierte Eingabe
<code>ZipOutputStream</code>	komprimiert Ausgabe mit Zip
<code>JarInputStream</code>	expandiert Jar-komprimierte Eingabe
<code>JarOutputStream</code>	komprimiert Ausgabe mit Jar

- Kompression/Expansion kosten Zeit

## 12.4.6 Beispielprogramm: Komprimierte Datei lesen und erzeugen

- Expandieren einer Gzip-komprimierten Eingabedatei:
  - Quelltext [GZIPExpand.java](#)
- Erzeugen einer Zipdatei:
  - Quelltext [ZipCompress.java](#)
- Die Ausgabedatei wird von Zip-Utilities akzeptiert

## 12.4.7 Performance

- Meßdaten eines beliebigen Systems, Dateigröße 5 MB §

<i>Art des Datentransportes</i>	<i>relative Laufzeit</i>
Kopieren byteweise	178
Kopieren über gepufferte Streams	18.2

Kopieren blockweise, Blockgröße 4k	0.3
Komprimieren blockweise, Blockgröße 4k	3.1
Expandieren blockweise, Blockgröße 4k	0.5
Komprimieren blockweise, Blockgröße 4k, Eingabe nicht komprimierbar	5.9

- Gzip und Zip haben gleiche Kompressionsraten und Laufzeiten §
- Zeit für Start der JVM subtrahiert

## 12.5 Text-I/O

### 12.5.1 Interne vs. externe Darstellung

- Interne Darstellung = "**Binärdarstellung**": innerhalb der JVM benutzt
- Externe Darstellung = "**Textdarstellung**": zur Kommunikation mit der Außenwelt, für menschliche Leser
- Beispiel: Planck'sche Konstante als `double`-Wert

*Binärdarstellung*

in 8 Bytes (hexadezimale Notation):

```
F2 08 82 C0 00 00 08 80
```

*Textdarstellung*

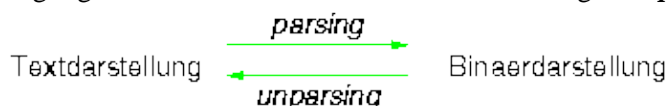
aus Dezimalziffern und Sonderzeichen

```
6.625E-34
```

- Beide Darstellungen **gleichwertig**, für unterschiedliche Zwecke geeignet
- Vorteile

<i>Binärdarstellung</i>	<i>Textdarstellung</i>
kompakt	lesbar
effizient zu verarbeiten	systemneutral
eindeutig	flexibel zu verarbeiten (Texteditor)

- Umwandlung externe → interne Darstellung = **Parsing**
- weniger gebräuchlich interne → externe Darstellung = **Unparsing**



### 12.5.2 Klassen für Text-I/O

- Ein-/Ausgabe von Text wichtig ⇒ von **eigenen Javaklassen** unterstützt.

<i>Klasse</i>	<i>Zweck</i>
Reader	Eingabestrom, liefert Textzeichen
Writer	Ausgabestrom, akzeptiert Textzeichen

- Reader/Writer transportieren lineare Sequenzen von **Textzeichen** (Unicode-char), nicht Bytes.
- Repräsentieren **keine Byteströme**, nicht von `InputStream` resp. `OutputStream` abgeleitet.

### ▶ 12.5.3 Methoden für Text-I/O

- Methoden zur Ein- und Ausgabe **einzelner Textzeichen**:

<i>Klasse</i>	<i>Methode</i>	<i>Zweck</i>
Reader	<code>int read()</code>	liefert das nächste Textzeichen
Writer	<code>void write(char c)</code>	gibt das Textzeichen c aus

- **Gepufferte Klassen** zur effizienteren Ein- und Ausgabe:

<i>Klasse</i>	<i>Zweck</i>
BufferedReader	Eingabestrom, liefert Textzeilen
PrintWriter	Ausgabestrom, schreibt Textzeilen

- Pufferung orientiert sich an **Zeilen**

<i>Klasse</i>	<i>Methode</i>	<i>Zweck</i>
BufferedReader	<code>String readLine()</code>	liefert die nächste Textzeile
PrintWriter	<code>void println(String s)</code>	gibt s als Textzeile aus

- `println` ist überladen zur **Ausgabe primitiver Werte** in Textdarstellung:

<code>println(int)</code>
<code>println(boolean)</code>
<code>println(String)</code>
usw.
<code>println()</code> (nur Zeilenvorschub)

- Ausgabe benutzerdefinierter Klassen über Strings
- Entsprechend zu `println` jeweils **print** ohne nachgesetzten Zeilenvorschub
- `PrintWriter` implementiert Unparsing.

### ▶ 12.5.4 Standardmethode `toString`

- Textdarstellung von Objekten beliebiger Klassen über die Methode `String toString()` §
- Wird *automatisch aufgerufen*, wenn ein Objekt...
  - ◆ mit `print` (`println`) ausgegeben wird
  - ◆ ein Objekt mit Strings verkettet wird
- Entspricht einer impliziten Typumwandlung in Strings

## ▶ 12.5.5 Parsermethoden und Wrapperklassen

- Nicht direkt ein Thema für I/O
- `print/println` leisten Unparsing für primitive und Klassentypen
- *Parsing* über Methoden von *Wrapperklassen*.
- *Parsermethoden* benannt mit Präfix "parse...":

<i>Primitiver Typ</i>	<i>Wrapperklasse</i>	<i>Parsermethode</i>
boolean	Boolean	
byte	Byte	<code>parseByte(String s)</code>
char	Character	
int	Integer	<code>parseInt(String s)</code>
short	Short	<code>parseShort(String s)</code>
long	Long	<code>parseLong(String s)</code>
float	Float	<code>parseFloat(String s)</code>
double	Double	<code>parseDouble(String s)</code>

Alle Wrapperklassen definiert im Package `java.lang` ⇒ keine import-Klausel notwendig

## ▶ 12.5.6 Brückenklassen

- Ausgabe von Text auf Dateien (oder andere Medien): `Writer` an `OutputStream` koppeln
- Verbindung mit "*Brückenklassen*"

<i>Klasse</i>	<i>Zweck</i>
<code>InputStreamReader</code>	Liefert Textzeichen aus einem Bytestrom
<code>OutputStreamWriter</code>	Schreibt Textzeichen auf einen Bytestrom

- Konstruktoren akzeptieren beliebigen Bytestrom
- Objekte der Brückenklassen codieren *Unicode-Zeichen in Bytes* (bzw. decodieren Bytes in Zeichen). Die Abbildung ist systemspezifisch. §

- Quelltext [CopyTextFile.java](#)

## 12.6 Package java.io

- I/O-Klassen in vier Gruppen

Richtung	Art	
	Bytes (binär)	Zeichen (Text)
Ausgabe	<a href="#">OutputStream</a>	Writer
Eingabe	<a href="#">InputStream</a>	Reader

- Jede Gruppe ähnlich strukturiert. Beispiel [OutputStream](#)

OutputStream (ABC)	<a href="#">FileOutputStream</a>			
	<a href="#">ByteArrayOutputStream</a>			
	<a href="#">PipedOutputStream</a>			
	<a href="#">FilterOutputStream</a> (Nachverarbeitung)	BufferedOutputStream		
		DeflaterOutputStream (Expansion)	ZipOutputStream	
			GZIPOutputStream	
			JarOutputStream	

## 12.7 Definition neuer I/O-Klassen

- Vordefinierte Klassen als Basisklassen
- Beispiel `UppercaseInputStream`: abgeleitet von [FilterInputStream](#), wandelt alle kleinen Buchstaben in große um
- Quelltext [UppercaseInputStream.java](#)

- Redefinition der betroffenen Methoden:

<code>int read()</code>
<code>int read(byte[])</code>
<code>int read(byte[], int, int)</code>

- Abgeleitete Klasse
  - ◆ holt Rohdaten (unverarbeitet) vom Datenelement in der Basisklasse [FilterInputStream](#)
  - ◆ untersucht unternimmt Prüfungen bzw. Umwandlung
  - ◆ gibt modifizierte Zeichen zurück an Aufrufer
- Anwendung der neuen Klasse wie vordefinierte Filterklassen

- Quelltext `CopyFileUppercase.java`

## ▶ 12.8 Serialisierung

### ▶ 12.8.1 Ziel

- Eigene Implementierung für Speichern und Laden benutzerdefinierter Klassen aufwendig und fehlerträchtig
- Universallösung für beliebige Klassen
- Einsatz wo immer Persistenz (Daten überdauern Programmläufe) gefordert

### ▶ 12.8.2 Interface `Serializable`

- **Tagging-Interface** = ohne Methoden, nur zur Markierung
- Kann von jeder Klasse implementiert werden
- Weitere Maßnahmen (als `implements`-Angabe) nur in seltenen Sonderfällen

### ▶ 12.8.3 Objektströme

- Speichern (Laden) auf (von) ***ObjectStreams***
- Vordefinierte Klassen:

<i>Klasse</i>	<i>Methode</i>	<i>Zweck</i>
<code>ObjectInputStream</code>	<code>Object readObject()</code>	liefert das nächste Objekt
<code>ObjectOutputStream</code>	<code>void writeObject(Object x)</code>	schreibt Objekt x

- Ergebnis von `readObject` vom Typ `Object`. Entweder  
*Typ bekannt*  
 ⇒ Typumwandlung ohne weitere Prüfung, oder  
*Typ unbekannt oder begrenzte Auswahl*  
 ⇒ Laufzeit-Typprüfung mit `instanceof`, dann Typumwandlung

### ▶ 12.8.4 Probleme

- Objekte enthalten ***weitere Objekte*** als Datenelemente: Objekt-I/O setzt sich rekursiv fort
- Zwei (oder mehr) Objekte enthalten Referenzen auf das jeweils andere Objekt = ***Referenzzyklen***:  
wird erkannt und korrekt behandelt
- Datenelemente mit ***flüchtigen Werten*** (Mausposition, Netzwerkverbindung, ...):  
Mit Qualifier "`transient`" definieren ⇒ bei Objekt-I/O ignoriert
- Bytecode abgespeicherter Objekte beim Laden nicht verfügbar ⇒ `ClassNotFoundException`

- Bytecode abgespeicherter Objekte beim Laden modifiziert (bspweise neu übersetzt): ⇒ `InvalidClassException`

## ▶ 12.8.5 Beispiel

- Objekt der Klasse `Box` speichern und laden: implementiert Interface `Serializable`

```
class Box implements Serializable
{
    Box(String n, int s)
    {
        name = n;
        size = s;
    }

    String name;
    int size;
}
```

- Hauptprogramm speichert `Box`-Objekt (Programmargument "store") oder liest eines ("load").
- Quelltext `ObjectIO.java`

## ▶ 13 Suchen und Sortieren

### ▶ 13.1 Suchen

#### ▶ 13.1.1 Problemstellung

- Siehe früher bei der Diskussion von Arrays
- Allgemeine Formulierung  
*Gegeben:*  
**Folge** von Elementen vergleichbarer Typen und  
 bestimmtes **Element  $x$**   
*Gesucht:*  
**Position von  $x$** , definiert durch seinen Wert oder andere Eigenschaften oder ein Signal für die  
**Abwesenheit von  $x$**
- Allgemeine Bezeichnung **Suchalgorithmen**

#### ▶ 13.1.2 Lineare Suche

- Geradlinige Idee: alle Elemente **absuchen** bis...  
 Treffer ( $x$  gefunden) oder  
 Ende der Folge.
- Algorithmus siehe früher
- Reihenfolge der Suche: **linear** von vorne nach hinten.
- Ende der Folge erreicht ⇒ Mißerfolg, Element nicht gefunden

### 13.1.3 Komplexität

- "**Komplexität**" eines Algorithmus = **Größenordnung** der Laufzeit bei gegebener Elementzahl  $n$  (Problemgröße) §
- Beispiel lineare Suche: im Mittel  $\frac{1}{2}n$  Schritte bis zu einem Treffer §
- Angabe **ohne Proportionalitätsfaktoren**  $\Rightarrow$  abstrahiert von allen äußeren Einflußfaktoren (Prozessorleistung, Systembelastung, Codequalität, Programmiersprache, Compileroptimierung, ...) §
- Bei Summentermen: nur der **am schnellsten wachsende Summand** §
- Formal: **Funktion  $O$**  (groß- $O$ ) der Anzahl Elemente
- Beispiel lineare Suche: Komplexität =  $O(n)$  §
- Ziel:
  - ◆ *keine konkrete* Laufzeitvorhersage in irgendeinem Zeitmaß,
  - ◆ nur **Bezug zwischen Problemgröße und Rechenaufwand**
- Trotzdem wichtige Aussagen: Für  $O(n)$  bspweise: Elementzahl verdoppeln  $\Rightarrow$  Suchdauer wird sich verdoppeln §

### 13.1.4 Einschätzung der Komplexität

- Beispiele für Komplexitäten:
  - $O(1)$  konstante Komplexität
  - $O(n)$  lineare Komplexität  
lineare Suche in einer unsortierten Folge
  - $O(\log(n))$  logarithmische Komplexität  
binäre Suche in einer sortierten Folge.
  - $O(n^2)$  quadratische Komplexität  
Bubblesort
- **Auswirkungen** unterschiedlicher Komplexitäten: Angenommen, ein bestimmter Algorithmus verarbeitet eine Folge mit  $n$  Elementen in einer Zeit  $t$ . Die Laufzeit ändert sich mit steigendem  $n$  wie folgt:

Komplexität		Anzahl Elemente				
		n	2n	4n	8n	16n
konstant	$O(1)$	t	t	t	t	t
linear	$O(n)$	t	2t	4t	8t	16t
logarithmisch	$O(\log(n))$	t	t+1	t+2	t+3	t+4
quadratisch	$O(n^2)$	t	4t	16t	64t	256t

exponentiell	$O(e^n)$	t	$t^2$	$t^4$	$t^8$	$t^{16}$
--------------	----------	---	-------	-------	-------	----------

- Allgemein:

Logarithmische Komplexität = "gut"

Exponentielle Komplexität = "schlecht"

- **Ursache der Komplexität:** Eigenschaft eines *Algorithmus* oder eines *Problems*?
- Unterschiedliche Sortieralgorithmen  $\Rightarrow$  Komplexität liegt im Algorithmus
- Oft: Komplexität eines Problems = Komplexität des besten (bekanntesten) Lösungsalgorithmus

### ▶ 13.1.5 Binäre Suche

- Algorithmus früher vorgestellt
- Binäre Suche verkürzt die Suchzeit auf Kosten der Vorverarbeitung (Sortieren).
- Sortieren lohnt sich, wenn die Folge *selten verändert*, aber *oft durchsucht* wird.
- Komplexität der binären Suche =  $O(\log(n))$   $\stackrel{\S}{\Rightarrow}$  **wesentlich besser** als lineare Suche mit  $O(n)$

## ▶ 13.2 Sortieren

### ▶ 13.2.1 Motivation

- Ziel: **Anordnung der Elemente**, bspweise in steigender Größe  $\stackrel{\S}{\Rightarrow}$
- Elemente *in place* (= an Ort und Stelle) sortieren, d.h. im Array tauschen, ohne weitere Speichermöglichkeit
- Überwiegend **algorithmisches Problem**  $\Rightarrow$  Objektorientierung bietet keinen entscheidenden Vorteil
- Sehr **häufiges Problem**
- Sortieralgorithmen heute **gut erforscht**  $\Rightarrow$  Eigenschaften recht genau bekannt.

### ▶ 13.2.2 Voraussetzung: Vergleichen

- Totalordnung = beliebige Elemente **paarweise vergleichbar**
- Trivial für die meisten primitiven Typen, wie z.B. Zahlen, Zeichen, Strings...
- Weniger naheliegend für Objekte beliebiger Klassen, bspweise "Person" (Körpergröße, Schuhgröße, Name im Alphabet, ...?)
- Ggf. passende **Vergleichsmethoden** explizit definieren, wie z.B.

```
int compareTo(x)
```

mit drei möglichen Ergebnissen: §

- 0 wenn das Objekt und x gleich sind,
- 1 wenn das Objekt kleiner als x ist und
- +1 wenn das Objekt größer als x ist.

### ▶ 13.2.3 Voraussetzung: Vertauschen

- **Vertauschungen** von Elementen an beliebigen Positionen zulässig  $\Rightarrow$  "random-access"-Container, wie bspweise ein Array
- "**Einfacher**" *Sortieralgorithmus*: Platzbedarf konstant, unabhängig von der Länge der Folge

### ▶ 13.2.4 Vergleich der Algorithmen

- Sortieralgorithmen mit ganz *unterschiedliche Eigenschaften*
- Interessante Kriterien:
  - ◆ Anzahl *Vertauschungen*
  - ◆ Anzahl *Vergleiche*
- Absolute Anzahlen weniger interessant, vielmehr die **Komplexität**, getrennt für Vertauschungen und Vergleiche
- Vertauschungen und Vergleiche unterschiedlich "teuer", je nach Art der Elemente und der Folge
- Zu prüfen: Verhalten des Algorithmus im...
  - worst case*  
= schlechtesten möglichen Fall = ungünstigste Startbedingungen
  - best case*  
= bestmöglichen Fall = beste Startbedingungen
  - average case*  
= durchschnittlichen Fall
- Durchschnittlicher Fall irgendwo zwischen dem besten und dem schlechtesten Fall, aber kaum genau in der Mitte!

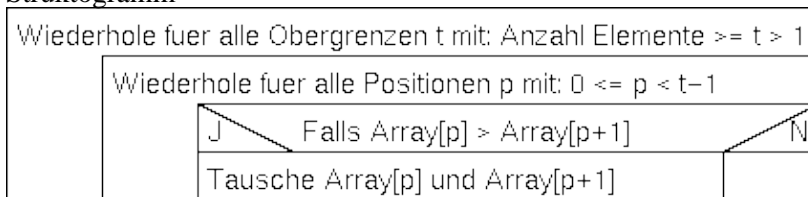
## ▶ 13.3 Einfache Sortieralgorithmen

Hier einige populäre, einfache Sortieralgorithmen.

### ▶ 13.3.1 Bubble Sort

- **Bubblesort** folgt einer simplen Idee: Man sucht *Fehlpaare*, d.h. benachbarte Elemente in "falscher" Anordnung, und vertauscht diese.
- Dieser Elementarschritt wird wiederholt, bis *alle Fehlpaare verschwunden* sind. Die Folge ist sortiert.

- Die Art der Suche nach Fehlpaaren spielt keine Rolle. Im Folgenden wird linear von vorne nach hinten gesucht.
- Der Algorithmus lässt sich mit einer naheliegenden Beobachtung vereinfachen: Nach dem ersten Durchgang steht das **größte Element am Ende** der Folge (es ist "hochgeblubbert" wie eine Luftblase in einer Flüssigkeit ⇒ "Bubblesort"). Dieses Element steht an der richtigen Stelle und kann in Zukunft ignoriert werden.
- Nach dem n. Durchlauf sind die hinteren n Elemente richtig einsortiert und können ignoriert werden.
- Der Algorithmus stoppt, wenn nur noch 1 Element zu "sortieren" ist.
- Struktogramm



- Ablaufbeispiel (verglichenes Paar fett, × = Tauschen, – = ok, sortierte Elemente farbig)

Durchlauf	Position	Array-Inhalt	Bemerkung
1	0	<b>23</b> ×08 59 16 03 56	Fehlpaar bunt markiert
1	1	08 <b>23</b> –59 16 03 56	kein Fehlpaar, Elemente stehen richtig
1	2	08 23 <b>59</b> ×16 03 56	Fehlpaar, tauschen
1	3	08 23 16 <b>59</b> ×03 56	Fehlpaar, tauschen
1	4	08 23 16 03 <b>59</b> ×56	Fehlpaar, tauschen
2	0	<b>08</b> –23 16 03 56 <b>59</b>	kein Fehlpaar, Elemente stehen richtig
2	1	08 <b>23</b> ×16 03 56 <b>59</b>	Fehlpaar, tauschen
2	2	08 16 <b>23</b> ×03 56 <b>59</b>	Fehlpaar, tauschen
2	3	08 16 03 <b>23</b> –56 <b>59</b>	kein Fehlpaar, Elemente stehen richtig
3	0	<b>08</b> –16 03 23 <b>56</b> <b>59</b>	kein Fehlpaar, Elemente stehen richtig
3	1	08 <b>16</b> ×03 23 <b>56</b> <b>59</b>	Fehlpaar, tauschen
3	2	08 03 <b>16</b> –23 <b>56</b> <b>59</b>	kein Fehlpaar, Elemente stehen richtig
4	0	<b>08</b> ×03 16 <b>23</b> <b>56</b> <b>59</b>	Fehlpaar, tauschen
4	1	03 <b>08</b> –16 <b>23</b> <b>56</b> <b>59</b>	kein Fehlpaar, Elemente stehen richtig
5	0	<b>03</b> –08 <b>16</b> <b>23</b> <b>56</b> <b>59</b>	kein Fehlpaar, Elemente stehen richtig
6	–	03 <b>08</b> <b>16</b> <b>23</b> <b>56</b> <b>59</b>	Sortierbereich 1 Element, Algorithmus endet

- Die Anzahl der Vergleiche liegt fest als

$$(n-1) + (n-2) + \dots + 1 = \frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n$$

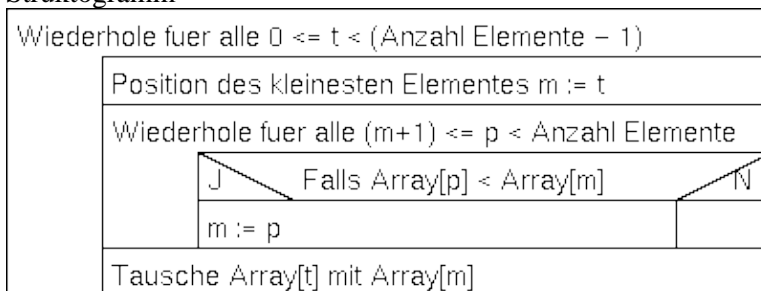
Bzgl. der *Vergleiche* hat Bubblesort *quadratische Komplexität*.

- Wenn man annimmt, daß im Mittel jedes zweite verglichene Elementepaar ein Fehlpaar ist, dann liegt die Anzahl der Vertauschungen etwa bei der Hälfte der Vergleiche, d.h. etwa  $\frac{1}{4}n^2$ . Bzgl. der *Vertauschungen* hat Bubblesort ebenfalls **quadratische Komplexität**.
- Im mittleren Fall sind  $\frac{1}{2}n^2$  Vergleiche und  $\frac{1}{4}n^2$  Vertauschungen zu erwarten. (Verifizieren Sie diese Angaben mit den Implementierungen)

### 13.3.2 Selection Sort

- Selectionsort trennt das Array in zwei Teile, einen sortierten (vorne) und einen unsortierten Teil (hinten).
- Der sortierte Teil ist zuerst 0 Elemente groß und wächst dann elementweise, bis er das ganze Array umfaßt.
- Beim Start ist das ganze Array unsortiert.
- Dann wird wiederholt:
  1. das kleinste Element *min* im unsortierten Teil suchen,
  2. *min* mit erstem Element im unsortierten Teil tauschen,
  3. den sortierten Teil um *min* verlängern.
 ...bis der unsortierte Teil auf 1 Element geschrumpft ist.

- **Struktogramm**



- Ablaufbeispiel (kleinstes Element fett, sortierter Bereich farbig)

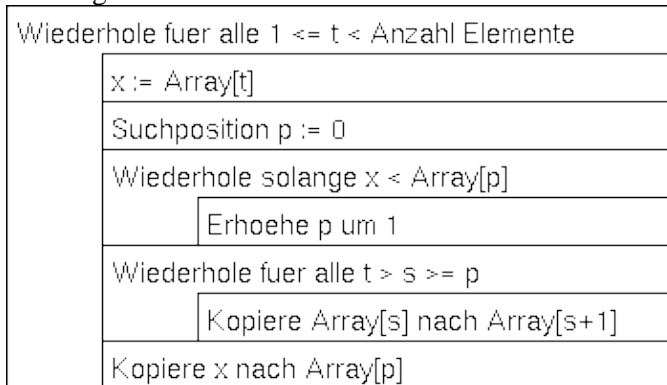
Schritt	Array-Inhalt	Bemerkung
1	23 08 59 16 <b>03</b> 56	unsortierter Bereich = gesamtes Array
2	<b>03</b> 08 59 16 23 56	kleinstes Element steht schon vorne im unsortierten Teil, nichts zu tauschen
3	<b>03</b> <b>08</b> 59 <b>16</b> 23 56	
4	<b>03</b> <b>08</b> <b>16</b> 59 <b>23</b> 56	
5	<b>03</b> <b>08</b> <b>16</b> <b>23</b> 59 <b>56</b>	
6	<b>03</b> <b>08</b> <b>16</b> <b>23</b> <b>56</b> 59	Ende, unsortierter Teil nur noch 1 Element

- Im mittleren Fall braucht Selectionsort  $\frac{1}{2}n^2$  Vergleiche und  $n$  Vertauschungen, d.h. die Komplexität bzgl. *Vergleichen* ist **quadratisch**, die bzgl. *Vertauschungen* **linear**.

### 13.3.3 Insertion Sort

- Die Folge wird in einen sortierten Teil (vorne) und einen unsortierten Teil (hinten) aufgeteilt, wie bei Selectionsort.
- Beim Start umfaßt der sortierte Teil das erste Element.
- Dann wird wiederholt:
  1. das erste Element *next* im unsortierten Teil herausnehmen,
  2. die passende Position im sortierten Teil suchen (binäre Suche),
  3. *next* dort einschieben.
 ...bis der unsortierte Teil verschwunden (= auf 0 Elemente geschrumpft) ist.

• **Struktogramm**



(Hier wird die Einfügeposition mit linearer statt binärer Suche ermittelt, um die Skizze übersichtlich zu halten.)

- Wenn man einem Kind einen Kartenstapel zum Sortieren gibt, benutzt es Insertionsort. §
- Ablaufbeispiel (betrachtetes Element fett, sortierter Bereich farbig, Einschublücke mit > markiert)

Schritt	Array-Inhalt	Bemerkung
1	>23 08 59 16 03 56	sortierer Bereich = 1. Element
2	08 23>59 16 03 56	Element 59 steht richtig, Tauschen nicht nötig
3	08>23 59 16 03 56	
4	>08 16 23 59 03 56	
5	03 08 16 23>59 56	
6	03 08 16 23 56 59	

- Implementierung siehe unten
- Im mittleren Fall braucht Selectionsort  $\frac{1}{4}n^2$  Vergleiche und  $\frac{1}{4}n^2$  Vertauschungen. **Beide Komplexitäten** sind **quadratisch**.
- Bei binärer Suche im sortierten Teil fällt die Anzahl der Vergleiche auf  $n \cdot \log(n)$  statt vorher  $\frac{1}{4}n^2$
- Ungeschickt: Verschieben der Elemente um 1 Position durch fortgesetztes, paarweises Tauschen.  
 Besser:
  - Erstes Element herausnehmen,
  - alle anderen um 1 Platz nach vorne kopieren,
  - letztes Element wieder einfügen
 ⇒ Anzahl Arrayzugriffe fällt auf die Hälfte

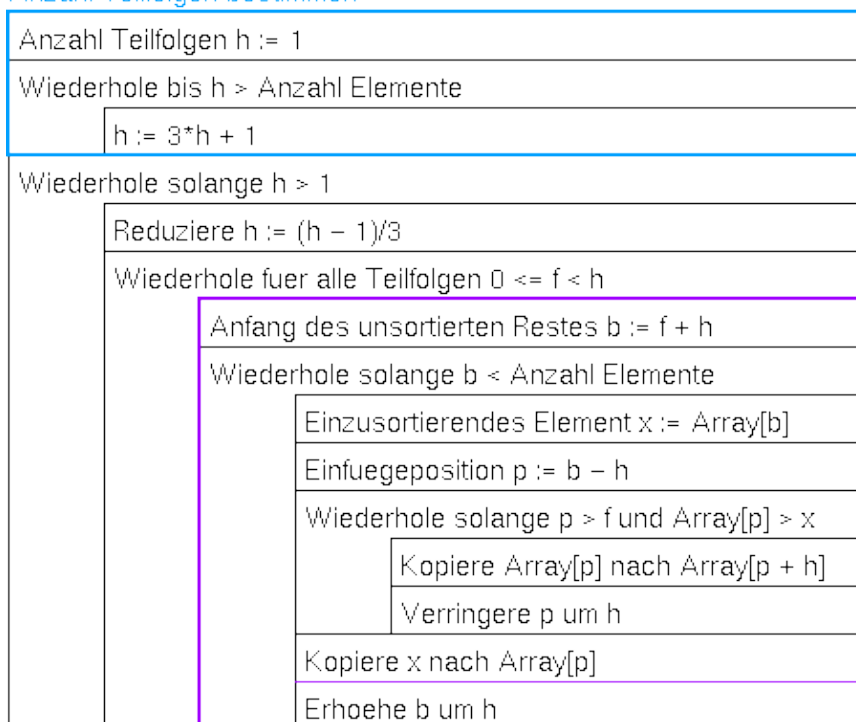
⇒ Komplexität bzgl. **Vertauschungen**  $n^2/8$  (statt vorher  $n^2/4$ ).

Aber formal: Operation zum Vertauschen von je 2 Elementen nicht geeignet

### ▶ 13.3.4 Shellsort

- Erfunden von *D. L. Shell*, 1959
- Shells Ziel war, Elemente "schnell" (= mit wenigen Vertauschungen) in die "richtige Gegend" zu transportieren.
- Die ganze Folge wird in  $n$  disjunkte Teilfolgen aufgeteilt, deren Elemente versetzt liegen. Ein Beispiel für  $n = 4$  Teilfolgen (je eine Farbe): **0 1 2 3 4 5 6 7 8 9 10 11** :
- Die Teilfolgen werden nacheinander isoliert sortiert, dazu wird irgendein anderer Sortieralgorithmus benutzt.
- Nachdem alle Teilfolgen für sich sortiert worden sind, wird die gesamte Folge in neue Teilfolgen zerlegt, diesmal aber weniger. Der Algorithmus endet, sobald die ganze Folge in einem Stück sortiert worden ist, m.a.W. sobald die Anzahl der Teilfolgen auf 1 reduziert worden ist.
- **Struktogramm**

Anzahl Teilfolgen bestimmen



Insertion sort

- Ein zunächst undurchsichtiges Problem liegt in der Wahl der richtigen Anzahl Teilfolgen.
- Experimente zeigen schnell, daß fallende Zweiterpotenzen keine guten Ergebnisse liefern (z.B. nacheinander 32, 16, 8, 4, 2, 1 Teilfolgen). Als günstig erweisen sich teilerfremde Anzahlen.
- Gute Ergebnisse erzielt man mit Aufteilungen gemäß

$$n_{i+1} = 3n_i + 1$$

also z.B. 40, 13, 4, 1 Teilfolgen

- Ablaufbeispiel (Elemente der jeweils behandelten Teilfolge farbig gedruckt)

Schritt	Array-Inhalt	Bemerkung
1	23 08 59 16 03 56	Anzahl der Teilfolgen = 4, erste Teilfolge wird sortiert
2	03 08 59 16 23 56	zweite Teilfolge wird sortiert; die weiteren Teilfolgen enthalten nur 1 Element, Sortieren fällt aus.
3	03 08 59 16 23 56	Anzahl der Teilfolgen = 1, ab hier normales Insertionsort
...		

- Im mittleren Fall braucht Shellsort  $n^{1.2}$  Vergleiche und ebenso viele Vertauschungen. Diese Angaben sind nur schwer theoretisch zu verifizieren.

### 13.3.5 Gegenüberstellung der Ergebnisse

- Eine Zusammenfassung der oben erwähnten Einzelergebnisse:

Algorithmus	Vergleiche	Vertauschungen
Bubblesort	$\frac{1}{2}n^2$	$\frac{1}{4}n^2$
Selectionsort	$\frac{1}{2}n^2$	n
Insertionsort (lineare Suche)	$\frac{1}{4}n^2$	$\frac{1}{4}n^2$
Insertionsort (binäre Suche)	$< n \cdot \log(n)$	
Shellsort	$n^{1.2}$	$n^{1.2}$

- Fazit: Bubblesort ist der einfachste und langsamste Algorithmus, Shellsort der komplizierteste und schnellste

### 13.3.6 Quicksort

- Quicksort ist *kein "einfacher"* Sortieralgorithmus, weil der Speicherplatzbedarf nicht konstant ist und mit steigender Länge der Folge ebenfalls steigt.
- Quicksort wird hier der Vollständigkeit halber erwähnt, die Diskussion zurückgestellt bis zur Rekursion
- Quicksort ist effizienter als alle hier betrachteten Sortieralgorithmen.

### 13.4 Beispielprogramme

- Für Such- und Sortieralgorithmen wird jeweils eine abstrakte Basisklasse definiert, die allgemeine Datenelemente und Methoden enthält, den eigentlichen Algorithmus aber offen läßt (abstrakte Methode).

Searcher.java

ABC für verschiedene Implementierungen von Suchern

Sorter.java

## ABC für verschiedene Implementierungen von Sortierern

- Die Basisklassen bieten Methoden zum Vergleichen und Austauschen von Elementen an. Diese Methoden sind in den abgeleiteten Klassen anstelle der primitiven Operatoren (== und >) zu verwenden. Sie liefern dieselben Ergebnisse, zählen aber die Anzahl der Aufrufe mit.
- Von der `ABC Searcher` werden konkrete Klassen abgeleitet, die lineare bzw. binäre Suche implementieren:
  - [LinearSearcher.java](#)  
Implementierung eines linearen Suchers
  - [BinarySearcher.java](#)  
Implementierung eines binären Suchers
- Von der `ABC Sorter` werden konkrete Klassen abgeleitet, die verschiedene Sortieralgorithmen implementieren:
  - [BubbleSorter.java](#)  
Implementierung von Bubblesort
  - [InsertionSorter.java](#)  
Implementierung von Insertionsort
  - [BinaryInsertionSorter.java](#)  
Dito. mit binärer Suche im sortieren Bereich
  - [SelectionSorter.java](#)  
Implementierung von Selectionsort
  - [ShellSorter.java](#)  
Implementierung von Shellsort
- Einfache Hauptprogramme erzeugen ein Objekt für den gewünschten Algorithmus und rufen dann die entsprechende Methode auf.
  - [SearchMain.java](#)  
Hauptprogramm für Suchalgorithmen
  - [SortMain.java](#)  
Hauptprogramm für Sortieralgorithmen

---

## ▶ 14 Collection–Framework

---

### ▶ 14.1 ArrayLists

---

#### ▶ 14.1.1 Problem mit Arrays

- Problem mit Arrays: Größe (= Anzahl Elemente) unveränderlich, mit "new" fixiert
- Praxis: Vorhersage schwierig oder sogar unmöglich §
- Ausweg: Klasse [ArrayList](#).

---

#### ▶ 14.1.2 ArrayLists vs. Arrays

- Übereinstimmende Eigenschaften Arrays/[ArrayList](#):
  - ◆ Lineare Anordnung von Elementen,
  - ◆ Elementzugriff *random-access*, d.h. auf alle Elemente gleich schnell,

- ◆ Elementauswahl über `int`-Index,
- ◆ Erstes Element hat Index 0.
- ArrayList ähnlich wie Arrays verwendbar
- **Unterschied:** Größe (Anzahl Elemente) von ArrayLists veränderlich  $\Rightarrow$  zur Laufzeit Elemente einschieben, löschen

### ▶ 14.1.3 ArrayListen erzeugen, Element anfügen

- Default-Konstruktor produziert leere ArrayList (null Elemente):

```
ArrayList names = new ArrayList(); // leer, 0 Elemente
```

- Methode `add` fügt am Ende ein weiteres Element an:

```
names.add("Lara");
```

- Beispiel: 100× Element anfügen

```
ArrayList names = new ArrayList();
for(int i = 0; <100; i++)
    names.add("Name#" + i);
```

## ▶ 14.2 Elementzugriff

### ▶ 14.2.1 Elementtyp Arrays

- Arrays eindeutig dem Elementtyp zugeordnet
- Zu jedem Javatypt existiert korrespondierender Arraytyp:

Elementtyp	Arraytyp
<code>int</code>	<code>int[]</code>
<code>float</code>	<code>float[]</code>
<code>boolean</code>	<code>boolean[]</code>
...	
<code>Complex</code>	<code>Complex[]</code>
...	
<code>int[]</code>	<code>int[][]</code>

- Irrtum fast  $\S$  ausgeschlossen:  $\S$

```
A[] a = new A[1];
a[0] = new B(); // Compiler meldet Fehler
```

## 14.2.2 Elementtyp ArrayList

- Nur *eine einzige* ArrayList-Klasse
- Typ aller ArrayList-Elemente: "java.lang.Object"
- Folge: In einer ArrayList Objekte beliebiger (Referenz-)Typen
- Beispiel:

```
ArrayList mixed = new ArrayList();

mixed.add("Me Willie");
mixed.add(new Complex(2, 3));
mixed.add(new int[] {1, 2, 4, 19});
mixed.add(new ArrayList());
```

- Fragwürdige Freiheit: Compiler kann Elementtyp nicht überwachen

## 14.2.3 Wrapperklassen

- Primitive Typen *nicht* von Object abgeleitet § ⇒ ungeeignet als ArrayList-Elemente
- Praxis: Unhaltbar!
- Ausweg: Hüllenklassen ("Wrapper") zum Einpacken primitiver Werte in Objekte
- Wrapper-Objekte enthalten primitiven Wert, aber als Objekte verwendbar
- Beispiel: ArrayList mit int-Werten

```
ArrayList table = new ArrayList();

table.add(new Integer(4));
table.add(new Integer(-227));
```

- Passende Methoden (z.B. Integer.intValue()) packen primitiven Wert wieder aus §

## 14.2.4 Umgang mit ArrayLists

### Zugriffsmethode ArrayList.get

- ArrayList-Elemente über Index ansprechen:

```
Object get(int index)
```

- Ergebnis: Element an Position "index", gezählt ab 0 = erstes Element
- Ergebnis = Object, wahrer Typ unbekannt
- Zurück zum "wahren" Elementtyp via Laufzeit-Typinformation

- Erst Elementtyp prüfen, dann Typumwandlung
- Falls Irrtum ausgeschlossen: Typumwandlung ohne Prüfung

## ▶ Prüfen des Elementtyps

- Operator `instanceof` liefert Laufzeittyp eines Objektes
- Beispiel:

```
// ArrayList names enthält Strings
for(int i = 0; ...; i++)
{
    if(names.get(i) instanceof String)
        // ok, weitermachen:
    else
        // Fehler: Element ist kein String!
}
```

## ▶ Typumwandlung ("*type cast*")

- Mit `Typecast` Typumwandlung erzwingen:

```
// ArrayList names enthält Strings
for(int i = 0; ...; i++)
{
    String s = (String)names.get(i);
    //...
}
```

- Falls Typumwandlung scheitert: "`ClassCastException`"
- Exception sofort behandeln (selten):

```
// ArrayList names enthält Strings
for(int i = 0; ...; i++)
{
    try
    {
        String s = (String)names.get(i);
        //...
    }
    catch(ClassCastException x)
    {
        // Fehler: Element ist kein String!
    }
}
```

- Meist Exception weiter außen behandelt

## ▶ 14.2.5 ArrayList–Methoden

- Nützliche `ArrayList`–Methoden

Element hinten *anfügen*

boolean add(Object o)

Element *auslesen*

Object get(int index)

*Anzahl Elemente* abfragen

int size()

Element *einschieben*

void add(int index, Object element)

Element *löschen* löschen

Object remove(int index)

Element *ersetzen*

Object set(int index, Object element)

Erstes bzw. letztes Vorkommen eines gegebenen Elementes *suchen* und Position zurückliefern (-1 bei Fehlschlag)

int indexOf(Object elem)

## ▶ 14.3 Organisation des Collection-Frameworks

### ▶ 14.3.1 Idee

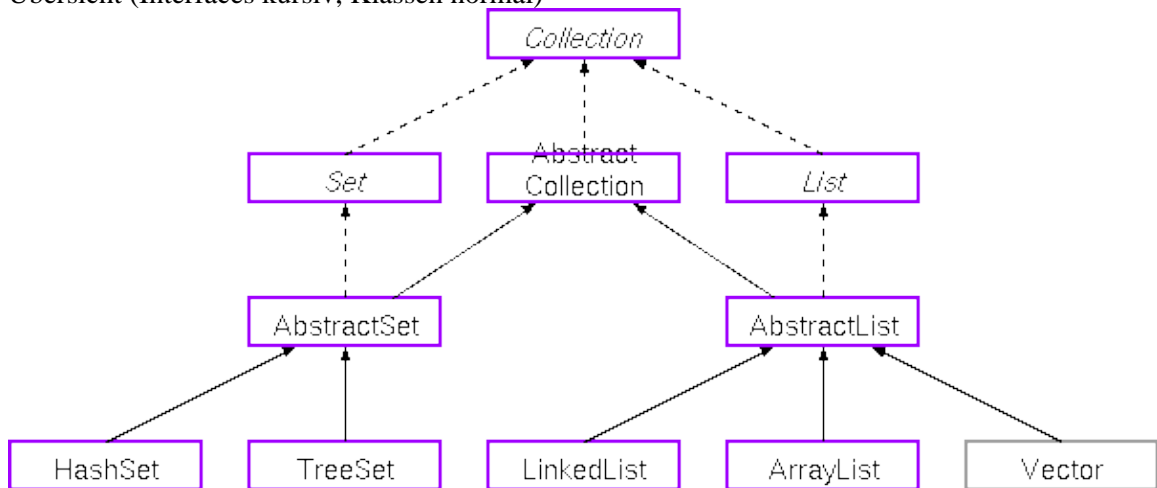
- ArrayList nur ein Vertreter einer *größeren Gruppe*
- Insgesamt "*Collection-Framework*" im Package `java.util`
- Ziel: *Einheitliche Behandlung* von Containern mit unterschiedlichen Eigenschaften
- *Beispiele* für Container mit unterschiedlichen Eigenschaften:
  - Tabelle*  
Lineare Anordnung, wahlfreier Zugriff einfach, Größe starr
  - Liste*  
Lineare Anordnung, sequentieller Durchlauf einfach, leichtes Wachsen und Schrumpfen
  - Menge*  
Sammlung ohne Duplikate
- Container oft gebraucht ⇒ *vordefiniert* in einer Bibliothek

### ▶ 14.3.2 Klassen und Interfaces

- Framework geteilt in *zwei Gruppen*:
  - Collections*  
speichern *Einzelemente*
  - Maps*  
speichern *Elementpaare* (später diskutiert)
- Entsprechend *zwei Interfaces* Collection und Map
- Viele weitere abgeleitete Interfaces und Klassen, gruppiert nach gemeinsamen Eigenschaften

### ▶ 14.3.3 Collection-Hierarchie

- Übersicht (Interfaces kursiv, Klassen normal)



- Ältere Klasse `Vector` grau

### 14.3.4 Allgemeine Methoden

- Jede `Collection` unterstützt das gleiche *Mindestsortiment* an Methoden
- **Erweitern** einer `Collection` um ein neues Element mit

```
boolean add(Object o)
```

- Ergebnis von `add...`
  - `true` Collection hat sich beim Erweitern vergrößert
  - `false` Keine Änderung

- Abruf eines *Iterators* mit

```
Iterator iterator()
```

### 14.3.5 Neue Collections

- Neue `Collection` implementieren gemäß Interface `Collection`: **15 Methoden** ⇒ Aufwand
- **Vereinfachung** über ABC `AbstractCollection`: Definiert alle `Collection`-Methoden außer `size()`, `iterator()`
- Neue `Collection` ableiten von `AbstractCollection`: Nur **2 Methoden** zu implementieren
- Zusätzlich zu definieren: *Iterator* mit **3 Methoden**
- **Beispiel**: `Collection` der 26 großen Buchstaben mit Hauptprogramm

### 14.3.6 Ältere Klassen

- Collections *seit Java 1.2*

- In *allen Javaversionen* Vector (Einzelemente) und Hashtable (Elementpaare)
- Vorteil: Collectionklassen *schneller*.
- Nachteil: Collectionklassen *nicht sicher* beim gleichzeitigen Zugriff durch mehrere parallel laufende Threads. §

## ▶ 14.4 Iteratoren

### ▶ 14.4.1 Ziel

- *Einheitlicher Zugriffsmechanismus* auf Elemente von Collections
- (Fast) *gleich* bei *verschiedenen* Collection-Klassen
- Zugriff über *Iterator*-Objekte
- Gemeinsames Interface Iterator

### ▶ 14.4.2 Anwendung

- Iterator definiert 3 Methoden:  
*boolean hasNext()*  
     Auskunft ob weitere Elemente verfügbar sind  
*Object next()*  
     Liefert nächstes Element  
*void remove()*  
     Löscht das zuletzt überquerte Element (siehe unten)
- Schleife über alle Elemente einer Collection:

```
Collection collection = ...;
Iterator i = collection.iterator();
while(i.hasNext())
{
    ... i.next() ...
}
```

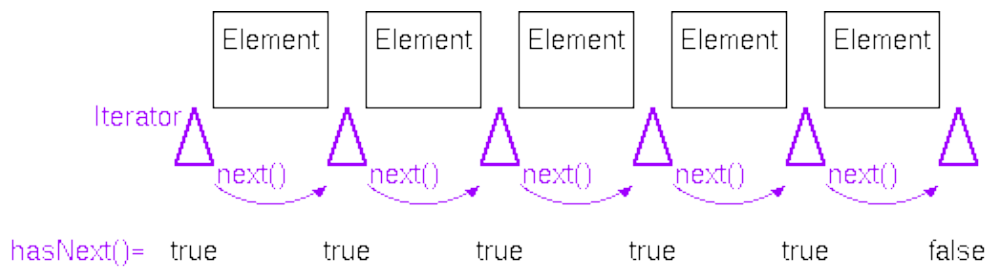
- Schleife über alle Elemente eines Arrays:

```
int[] array = ...;
int i = 0;
while(i < array.length)
{
    ... array[i++] ...
}
```

### ▶ 14.4.3 Grenzen

- Iteratoren stehen konzeptionell *zwischen* zwei Elementen §

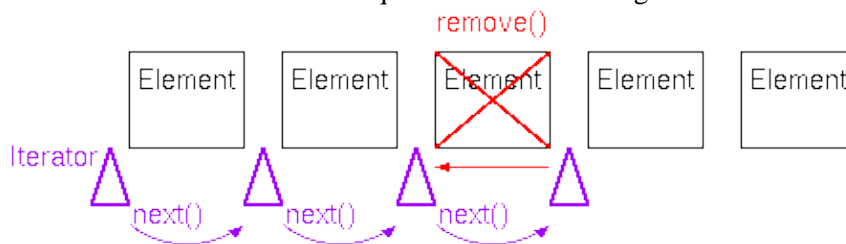
## Single page



- Kein *Random-access*: Elemente nur sequentiell
- Kein Rücksetzen: Iterator durchlaufen  $\Rightarrow$  verbraucht, nicht wiederverwendbar

### 14.4.4 Löschen von Elementen

- Zuletzt von einem Iterator überquertes Element kann gelöscht werden



- Vorsicht! Nach `remove()` zunächst kein "zuletzt überquertes" Element (das wurde ja gerade gelöscht)  $\Rightarrow$  `remove()` darf nicht sofort noch einmal aufgerufen werden, sonst `UnsupportedOperationException`
- Motivation: Löschen eines Elementes aus (großer) Collection an **bekannter Position effizienter**
- Iterator definiert nicht: Einfügen oder Ersetzen von Elementen  $\Rightarrow$  Aufgabe konkreter Klassen

### 14.4.5 ListIterator

- Iterator läuft nur in einer Richtung (vorwärts = von vorne nach hinten)
- Abgeleitetes Interface ListIterator: **vorwärts und rückwärts**. Methoden:
  - `hasPrevious()`  
Auskunft ob Vorgängerelement existiert (entsprechend zu `hasNext()`)
  - `previous()`  
Liefert das Vorgängerelement (entsprechend zu `next()`)
- ListIterator definiert für AbstractList und abgeleitete Klassen
- Außer Löschen auch Methoden
  - `add(Object x)`  
**Einfügen** eines neuen Elementes
  - `set(Object x)`  
**Ersetzen** des zuletzt überquerten Elementes

## 14.5 Maps

### 14.5.1 Assoziatives Array

- Elemente in Collections und Arrays verschiedene Typen, **Indexwerte immer *int***
- Gegensatz: "**Assoziatives Array**" = Container mit unterschiedlichen (insbes. nicht-numerischen) Indextypen
- Beispiel: Telefonbuch

Index    %.% Name    Indextyp = 'String'  
 Element    → Telefonnummer    Elementtyp = long  
 .Passende Container = "Maps", speichern **Elementpaare** = "Schlüssel" + "Wert"

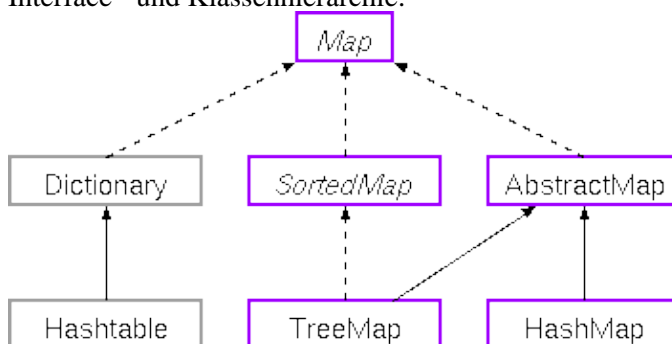
## ▶ 14.5.2 Schlüssel und Werte

- Indexwerte = "**Schlüssel**" (*keys*)
- Elemente = "**Werte**" (*values*)
- Im Beispiel:n

Name    %.% Schlüssel  
 Telefonnummer    → Wert  
 .Maps enthalten **Einträge** = Schlüssel/Wert-Paare. §. Grundsätzlich: Schlüssel **eindeutig**, §  
 Werte **beliebig** (insbesondere auch mehrfach)

## ▶ 14.5.3 Maps vs. Collections

- In Java: Maps getrennt von Collections §
- Gemeinsames Interface Map
- Interface- und Klassenhierarchie:



- Interfaces kursiv, Klassen normal, ältere Klassen grau

## ▶ 14.5.4 Grundlegende Methoden

- Definiert im Interface Map

Object put(Object k, Object x)	Neuen <b>Eintrag aufnehmen</b> (Schlüssel = k, Wert = x)
Object get(Object k)	Eintrag <b>suchen</b> (Schlüssel = k)
Object remove(Object k)	Eintrag <b>löschen</b> (Schlüssel = k)
int size()	Abfrage der <b>Anzahl Einträge</b>

- Einige weitere Methoden

## ▶ 14.6 HashMap

### ▶ 14.6.1 Einordnung

- Implementiert Interface Map
- Erlaubt null als Schlüssel und als Wert §
- Keine bestimmte Ordnung unter den Elementen

### ▶ 14.6.2 Beispiel: Telefonbuch

- Telefonbuch als Beispiel für assoziatives Array
- Schlüssel-Typ String, Werte-Typ long §
- Implementierung mit HashMap:

```
Map phonebook = new HashMap();

phonebook.put("Lara", new Long(1805225100));
phonebook.put("Schiedermeier", new Long(22222222));
phonebook.put("Sekretariat", new Long(11111111));
...
```

- Bei Elementzugriff Prüfen des dynamischen Typs + Type Cast (siehe ArrayList)

```
(Long)phonebook.get("Sekretariat")...
```

### ▶ 14.6.3 Bestandteile einer Map

- Schlüssel einer Map i.allg. nicht aufzählbar
- Drei verschiedene Abbildungen von Maps auf Collections:
  1. Alle Schlüssel als Set
  2. Alle Werte als Collection
  3. Alle Einträge als Set
- Methoden:

Set keySet()	Liefert die Menge aller Schlüssel
Collection values()	Liefert eine Collection aller Werte

Set entrySet() Liefert die Menge aller Einträge

- Weitere Verarbeitung der Collections wie oben gezeigt, bspweise mit Iteratoren
- 

## ▶ 14.6.4 Gleichheit von Schlüsseln

- Schlüssel beliebige Objekte, aber eindeutig in einer Map
- Frage: Wie ist *Objekt-Gleichheit* definiert?
- Ohne weitere Maßnahmen: Map stützt sich auf *Objekt-Identität*
- Oft unbrauchbar ⇒ benutzer-gesteuerter Mechanismus wünschenswert
- Eingriff über Standardmethode `int Object.hashCode()`
- In eigener Klasse redefinieren, so daß für *logisch gleiche* Objekte *derselbe Hashcode* zurückgegeben wird
- Beispiel: Studentenverwaltung, Klasse Student:

```
public int hashCode()
{
    return matrikelnummer;
}
```

- Student-Objekte als Schlüssel einer Map ⇒ maximal *ein* Eintrag pro Matrikelnummer
- 

## ▶ 14.6.5 Einsatz von HashMaps

- Gründe für den Einsatz von HashMaps:
    - ◆ Schlüssel nicht fortlaufend aufzählbar
    - ◆ Schneller Zugriff in großes Datenvolumen
    - ◆ Elemente oft gesucht, aber selten entfernt
    - ◆ Elemente mit weit verstreuten Ordnungsnummern
  - Geeignet als "Mini-Datenbank"
- 

## ▶ 14.7 Algorithmen

---

### ▶ 14.7.1 Idee

- Wiederkehrende Algorithmen mit Collections und Maps einmal implementieren ⇒ Routinearbeit reduzieren
- Beispiele: Suchen, sortieren

- Erfolgreich praktiziert in der C++-STL, übertragen auf das Collection-Framework
- Angebot im Collection-Framework (noch) nicht so weitreichend wie STL

## ▶ 14.7.2 Organisation

- Algorithmen als *statische Methoden der Klasse Collections*
- Klasse Collections (Plural) nicht zu verwechseln mit Interface Collection (Singular)
- Collections enthält *nur* statische Methoden
- Alle Collections-Algorithmen arbeiten mit *minimalen Interfaces*  $\Rightarrow$  funktionieren mit *allen* abgeleiteten Klassen
- Algorithmen an keine bestimmte Implementierung gebunden
- Eigenschaften zugesichert, beispielsweise Sortieren in Zeitkomplexität  $O(n \cdot \log(n))$

## ▶ 14.7.3 Beispiele

- Nützliche Methoden

<code>int binarySearch(List l, Object k)</code>	Suche Objekt <code>k</code> und liefert den Index
<code>Object max(Collection c)</code>	Sucht das größte Element (entsprechend <code>min</code> )
<code>void shuffle(List l)</code>	Mischt die Elemente zufällig
<code>void sort(List l)</code>	Sortiert die Elemente aufsteigend

## ▶ 14.7.4 Größenvergleich

- Algorithmen und sortierte Container (`TreeSet`, `TreeMap`) erfordern Ordnung unter den Elementen einer `Collection`
- Für beliebige Objekte nicht ohne weiteres definiert §
- Methode `int compareTo(Object x)` liefert Aussage über Ordnung von Zielobjekt und `x`
- Definiert im Interface Comparable
- Benutzerdefinierte Klasse implementiert Comparable, definiert `compare`  $\Rightarrow$  Objekte geeignet für Algorithmen und sortierte Container
- Weiteres Interface Comparator für frei definierbare Vergleichs-Objekte  $\Rightarrow$  gebraucht für Elemente, die Comparable nicht implementieren (können) §

## ▶ 14.7.5 Beispiele

- Beispielprogramm füllt Liste mit zufällig gewählten Brüchen

- Ohne weitere Maßnahmen: Collections–Sortieralgorithmus nicht einsetzbar (Übersetzung ok, Exception zur Laufzeit)
- Alternative 1: Klasse der Listenelemente (Rational) implementiert Interface Comparable und definiert Methode `Rational.compareTo()` (siehe Beispielprogramm)

```
public int compareTo(Object x)
{
    if(x instanceof Rational)
    {
        Rational r = (Rational)x;
        // this und r vergleichen...
        // Ergebnis -1, 0 oder 1
    }
    throw new ClassCastException();
}
```

- Erfordert Änderung der Listenelementklasse
- Alternative 2: Getrennte Vergleicherklasse `RationalComparator` für `Rationals` definieren, implementiert Interface Comparator (siehe Beispielprogramm)

```
class RationalComparator implements Comparator
{
    public int compare(Object x1, Object x2)
    {
        if(x1 instanceof Rational & x2 instanceof Rational)
        {
            Rational r1 = (Rational)x1;
            Rational r2 = (Rational)x2;
            // r1 und r2 vergleichen...
            // Ergebnis -1, 0 oder 1
        }
        throw new ClassCastException();
    }
}
```

- Keine Änderung der Listenelementklasse nötig, aber Nachteil: neue Klasse

## ▶ 14.8 Gegenüberstellung

### ▶ 14.8.1 Konkrete Containerklassen

- Konkrete Containerklassen des Collection–Framework

<i>Klasse</i>	<i>Zugriff</i>	<i>Geordnet</i>	<i>Sortiert</i>	<i>Duplikate</i>	<i>Einfügen</i>
ArrayList	Index	ja	nein	ja	langsam
LinkedList	sequentiell	ja	nein	ja	schnell
HashSet	Objekt	nein	nein	nein	schnell
TreeSet	Objekt	ja	ja	nein	schnell
HashMap	Schlüssel	nein	nein	ja (Werte)	schnell
TreeMap	Schlüssel	ja	ja	ja (Werte)	schnell

- In allen Java-Versionen:

- ◆ Vector (~ArrayList)
- ◆ Hashtable (~HashMap)
- ◆ Stack (Vector mit zusätzlichen, stack-typischen Methoden)
- ◆ BitSet (~Vector mit boolean-Elementen, verhältnismäßig effizient)

## ▶ 14.8.2 Klassen und Interfaces im Collection-Framework

<u>AbstractCollection</u>	ABC mit Default-Implementierungen für die meisten Collection-Methoden
<u>AbstractList</u>	ABC mit Default-Implementierungen für die meisten List-Methoden
<u>AbstractMap</u>	ABC mit Default-Implementierungen für die meisten Map-Methoden
<u>AbstractSet</u>	ABC mit Default-Implementierungen für die meisten Set-Methoden
<u>ArrayList</u>	Konkreter Container mit Array-ähnlichen Eigenschaften
<u>Collections</u>	Klasse mit statischen Methoden für generische Algorithmen
<u>Collection</u>	Basis-Interface für alle Collections (speichern Einzelemente)
<u>Comparable</u>	Interface für größen-vergleichbare Objekte
<u>Comparator</u>	Interface für Vergleicher, die beliebige Objekte vergleichen können
<u>HashMap</u>	Ungeordnete Menge von Schlüssel/Wert-Paaren
<u>HashSet</u>	Ungeordnete Objektmenge
<u>Hashtable</u>	Ungeordnete Menge von Schlüssel/Wert-Paaren, in allen Javaversionen, ähnlich HashMap
<u>Iterator</u>	Minimaler Collection-Durchläufer
<u>LinkedList</u>	Konkreter Container für sequentiellen Zugriff
<u>ListIterator</u>	Durchläufer für geordnete Collections
<u>List</u>	Interface für alle Collections mit geordneten Objekten
<u>Map</u>	Basis-Interface für alle Maps (speichern Elementpaare)
<u>Set</u>	Interface für alle Collections mit ungeordneten Objekten
<u>SortedMap</u>	Interface für alle geordneten Mengen von Schlüssel/Wert-Paaren
<u>TreeMap</u>	Geordnete Menge von Schlüssel/Wert-Paaren
<u>TreeSet</u>	Geordnete Objektmenge
<u>Vector</u>	Konkreter Container mit Array-ähnlichen Eigenschaften, in allen Javaversionen, ähnlich ArrayList

## ▶ 15 Verkettete Listen

### ▶ 15.1 Einfach verkettete Listen

#### ▶ 15.1.1 Motivation

- Listen = Container, wie Arrays und Collection-Klassen
- In Java 1 Listen nicht vordefiniert, in Java 2 im Collection Framework als java.util.LinkedList
- Inhalt (= Elemente = "Nutzlast") beliebig vom Typ `Object`
- Im diesem Kapitel `int` als Typ der Elemente  $\Rightarrow$  Vereinfachung §
- Listen häufig gebraucht  $\Rightarrow$  grundlegendes Konzept in der Programmierung

## 15.1.2 Aufbau

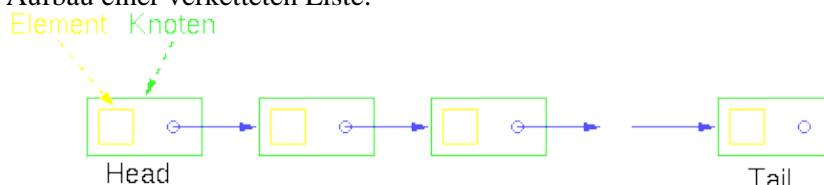
- Jedes Element in einem eigenen Objekt verpackt
- Element + Verpackung = "**Knoten**" der Liste.
- Knoten als lineare Folge arrangiert
- Jeder Knoten kennt...
  - ◆ ...seinen Nachfolger,
  - ◆ ...sonst keine anderen Knoten der Liste
- Knoten  $\approx$  Perlen auf einer Kette  $\Rightarrow$  "**Verkettete Liste**" (*linked list*)

## 15.1.3 Bezeichnungen

- Für die weitere Diskussion vereinbarte **Benennungen**

Knoten	<i>node</i>	Einzelkomponente der Liste
Element	<i>element, data</i>	Nutzlast in einem Knoten
Kopf	<i>head</i>	Erster Knoten einer Liste
Ende	<i>tail</i>	Letzter Knoten einer Liste
Nachfolger	<i>next</i>	Nächster Knoten in der Liste (existiert evtl. nicht)
Vorgänger	<i>previous</i>	Vorhergehender Knoten in der Liste (existiert evtl. nicht)
Länge		Anzahl Knoten
Leere Liste		Liste ohne Knoten, mit 0 Knoten

- Aufbau einer verketteten Liste:



## 15.1.4 Eigenschaften

- **Vorteile** von Listen (gegenüber etwa Arrays, Vektoren, Hashtables)
  - ◆ Für jedes Element 1 Knoten  $\Rightarrow$  kein Knoten zuviel, kein Verschnitt
  - ◆ Listenstruktur (= Abfolge der Knoten) billig zu ändern  $\Rightarrow$  kein Verschieben und Kopieren
  - ◆ Leicht zu teilen, zusammenzufügen
- **Nachteile**
  - ◆ *Random-access* (= Elementzugriff via Index) teuer
  - ◆ Für jedes Element zusätzlicher Speicherplatzverbrauch (Knoten)
  - ◆ Anzahl Knoten nur umständlich feststellbar

## ▶ 15.2 Implementierung, 1. Ansatz

### ▶ 15.2.1 Knotenklasse

- Ansatz Knotenklasse:

```
class Node
{
    private int element;    // Nutzlast
    private Node next;     // Nachfolgerknoten
}
```

- Im Tail-Knoten: `next = null`.

### ▶ 15.2.2 Primitive Zugriffsmethoden

- Umgang mit Knoten über einfache Methoden
- **Konstruktor** für neuen, isolierten Knoten mit gegebenem Element `e`:

```
Node(int e)
{
    element = e;
    next = null;    // vorläufig
}
```

`next` erst später, beim Aufbau einer Liste festlegen

- **Abfrage des Inhalts:** §

```
int element()
{
    return element;
}
```

- **Abfrage des Nachfolgerknotens** (`null` im Endknoten):

```
Node next()
{
    return next;
}
```

- **Ersetzen des Nachfolgers** durch einen gegebenen Knoten other:

```
void next(Node other)
{
    next = other;
}
```

- UML

<b>Node</b>
-next: Node -element: int
Node(int) element(): int next(): Node next(Node):

- Implementierung [Beispielprogramm Node.java](#)

### 15.2.3 Probleme

- Definition technisch ok, aber...
- Verkettung der Knoten vom Benutzer konsistent zu halten ⇒ brüchig!
- Freier Zugriff auf die Verkettung ⇒ keinerlei Schutz vor unzulässigen Konstruktionen (ein Knoten mit zwei Vorgängern usw.)
- [Beispielprogramm](#) zum Umgang mit simpler Knotenklasse

## 15.3 Implementierung, 2. Ansatz

### 15.3.1 Listenoperationen

- Direkten Zugriff auf Listenstruktur unterbinden, alte Methode "void next(Node)" entfällt
- Ersetzen durch Methoden für listen-spezifische Operationen: Einfügen, Löschen, Suchen von Knoten
- Inkonsistente Strukturen ausgeschlossen
- Methodenköpfe für Listenoperationen:

insert(Node n, Node a)	Der neue Knoten n wird in die Liste vor dem Ankerknoten a eingefügt.
remove(Node n)	Der Knoten n wird aus der Liste entfernt.

<code>find(int e)</code>	Der erste Knoten mit dem Inhalt <code>e</code> wird gesucht.
--------------------------	--

- Nach außen: Head = Repräsentant der ganzen Liste
- UML

<b><i>Node</i></b>
-next: Node -element: int
Node(int) element(): int next(): Node insert(Node, Node): Node remove(Node): Node find(int): Node -previous(Node): Node

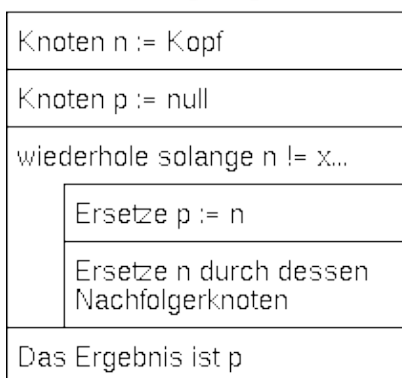
- Implementierung Beispielprogramm Node.java

## ▶ 15.3.2 Hilfsmethode Vorgängerknoten

- Die Semantik der Methoden `insert` und `remove` muß sorgfältig geplant werden.
- Nützliche private Hilfsmethode: Vorgänger eines gegebenen Knotens `x`
- Definiert als

```
private Node previous(Node x)
{...}
```

- **Struktogramm:**  
*Suche den Vorgänger des Knotens x*



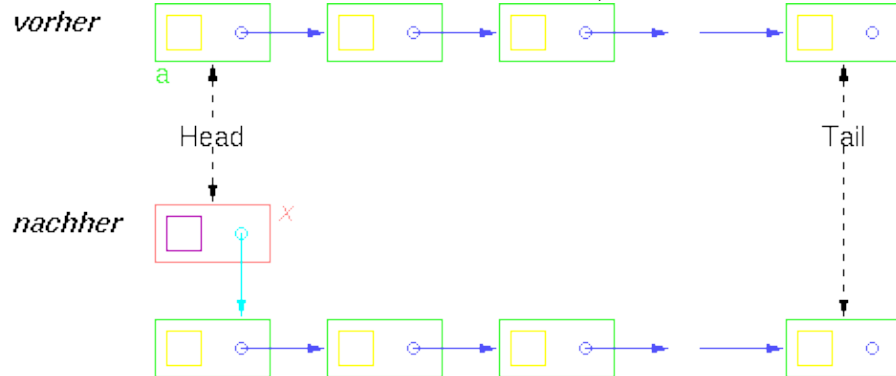
- Grenzfälle:
  1. Vorgänger von Head → null
  2. Vorgänger von null → Tail
- Grenzfälle nachher nutzbar

## ▶ 15.3.3 Einfügen

- Fälle zu unterscheiden:

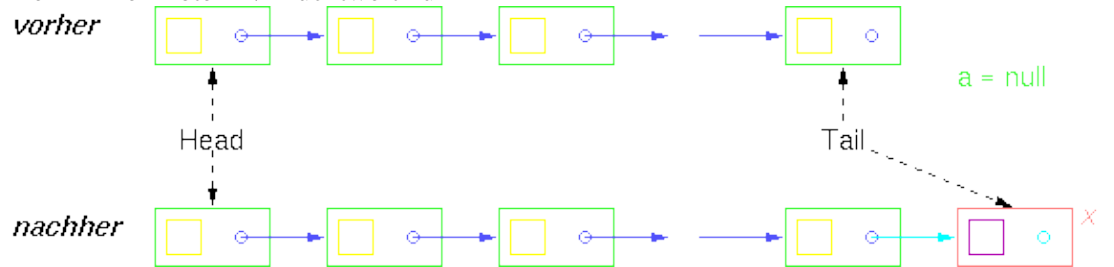
1. Einfügen vor Head

Ankerknoten = Head  $\Rightarrow$  Neuer Knoten wird Head, insert liefert neuen Head zurück:



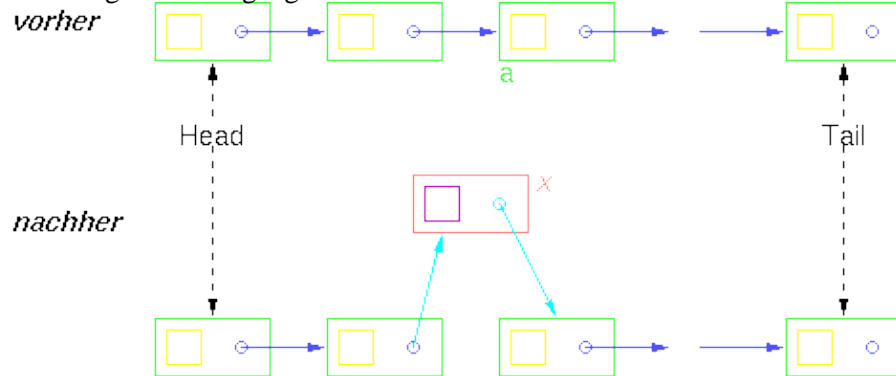
2. Einfügen nach Tail

Kein Ankerknoten  $\Rightarrow$  Fluchtwert null



3. Einfügen in der Mitte

Ankerknoten wird Nachfolger des neuen Knotens,  
Nachfolger des Vorgängers des Ankerknotens = neuer Knoten



Reihenfolge der Zuweisungen heikel!

- Struktogramm

*Fuege Knoten x vor Knoten a ein*

Ist der Ankerknoten a == Kopf?		
Nachfolger von x := Kopf	Ist der Ankerknoten == null?	
Kopf := x	Nachfolger von x := null	Nachfolger von x := a
	Nachfolger des Endknotens := x	Nachfolger des Vorgaengers von a := x
Ergebnis ist der Kopf		

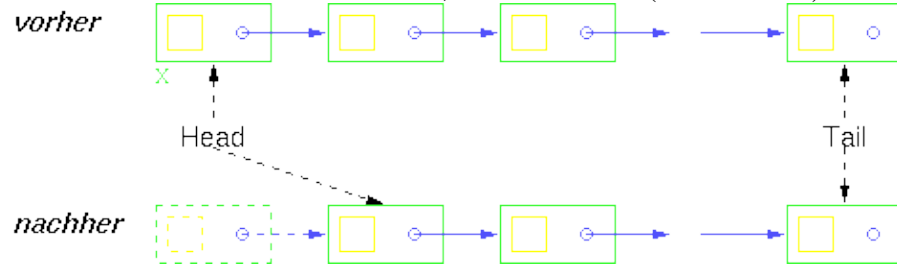
### 15.3.4 Löschen

- previous(Node) auch hier hilfreich

• Fälle (entsprechend "insert"):

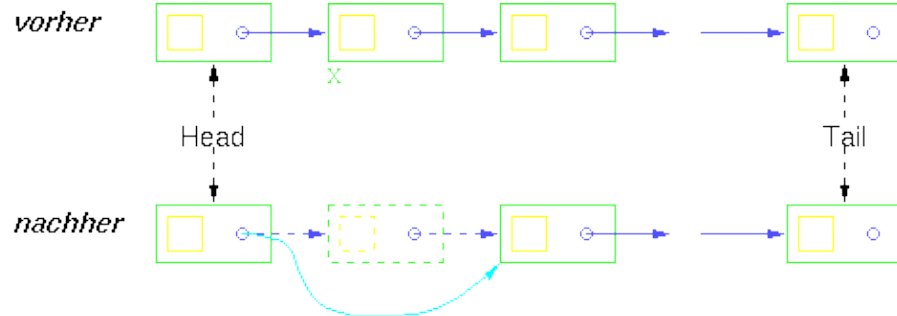
1. Löschen von Head

Bisher zweiter Knoten = neuer Head, remove liefert (wie insert) neuen Head zurück



2. Löschen in der Mitte

Knoten x kurzschließen:



3. Löschen von Tail

Keine Sonderbehandlung, entsprechend Fall 2

• Struktogramm:

Loesche Knoten x

Ist x == Kopf?	
Kopf := Nachfolger des Kopfes	Knoten p := Vorgaenger von x Ersetze des Nachfolger von p durch den Nachfolger von x
Ergebnis ist der Kopf	

### 15.3.5 Probleme

- Listenoperationen bequemer, sicherer als 1. Ansatz
- Gelöste Probleme
  - ◆ Einfügen von null als neuem Knoten,
  - ◆ Einfügen vor einem Ankerknoten, der überhaupt nicht in der Liste vorkommt,
  - ◆ Löschen eines Knotens, der nicht in der Liste vorkommt.
- Offene Probleme:
  - ◆ Knoten kann in zwei Listen eingefügt werden
  - ◆ Repräsentation einer leeren Liste §
  - ◆ Semantische Fragen (siehe unten)

- Beispielprogramm zum Umgang mit Listen

## ▶ 15.4 Implementierung, 3. Ansatz

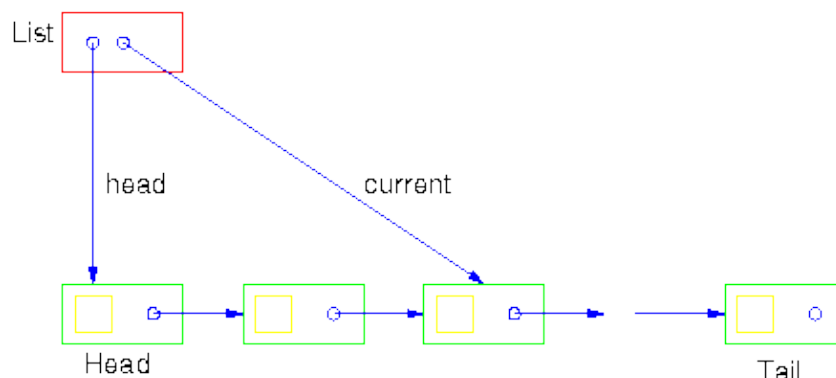
### ▶ 15.4.1 Listen- und Knotenklasse

- Idee: Knoten in Listenobjekt verwalten
- Benutzer sieht nur Listenobjekt, aber kein Knotenobjekt mehr
- Listenoperationen von List übernommen:

```
class List
{
    List();           // neue Liste ohne Knoten
    void insert(int e); // neuen Knoten mit Element e einfügen
    void remove();    // Knoten löschen
    int element();    // Auskunft über den Inhalt
    int length();     // Anzahl Knoten
}
```

### ▶ 15.4.2 Aktueller Knoten

- Die meisten Listenoperationen (insert, remove usw.) brauchen einen Bezugsknoten
- Aber: List-Klasse verbirgt Knoten von Benutzer
- Lösung: List verwaltet intern einen "aktuellen Knoten" (Datenelement "current")
- Listenoperationen beziehen sich immer auf jeweils aktuellen Knoten
- Ebenso: Head im List-Objekt festhalten (Datenelement "head")



- Definition

```
class List
{
    // ...wie oben
    private Node head; // Kopf
    private Node current; // aktueller Knoten
}
```

- Zusätzlich erforderlich: Methoden zum Ändern des aktuellen Knotens:

```
class List
{
    // ...wie oben
    void head();           // aktuellen Knoten an den Kopf zurück
    void next();          // aktuellen Knoten um 1 Knoten nach hinten
    boolean ok();         // aktueller Knoten noch in der Liste?
}
```

### 15.4.3 Innere Klassen

- Java erlaubt die Definition einer Klasse innerhalb einer anderen Klasse
- Geschachtelte Klasse = "innere Klasse" (= "lokale Klasse" = *inner class*).
- Innere Klassen hier sinnvoll anzuwenden: `Node` wird lokal in `List` definieren
- Zugriff auf Innere Klasse über Zugriffsrecht steuern (wie Methoden, Datenelemente)
- Beispiel `List`:

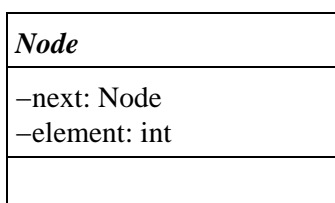
```
public class List
{
    private class Node
    {
        public Node()
        //...
    }

    //...
    private Node head;
}
```

- Eigenschaften
  - ◆ `Node` als `private` definiert  $\Rightarrow$  für Anwender von `List` nicht erreichbar
  - ◆ Konstruktor `Node()` ist `public` definiert  $\Rightarrow$  für `List` uneingeschränkt verfügbar

### 15.4.4 Probleme mit aktuellen Knoten

- Nur *ein* aktueller Knoten pro Liste
- Zwei oder mehr aktuelle Positionen nicht gleichzeitig verwaltbar §
- Lösung nachher mit Iteratoren
- UML



Node(int) element(): int next(): Node insert(Node, Node): Node remove(Node): Node find(int): Node -previous(Node): Node
---

<b>List</b>
-current: Node -head: Node
List() insert(int) remove() find(int): boolean ok(): boolean head() next() length(): int

- Implementierung [Beispielprogramm List.java](#)
- [Beispielprogramm](#) zum Umgang mit Listen

## ▶ 15.5 Iteratoren

### ▶ 15.5.1 Idee

- Listenpositionen vom List-Objekt trennen
- List-Objekt kennt *überhaupt keine* Position mehr
- Listenpositionen in eigene Klasse "Iterator" § packen
- Iteratoren an bestimmte Liste gebunden, nicht auf andere Liste übertragbar
- Beliebig viele Iteratoren auf derselben Liste
- Jeder Iterator kennt "seine" Liste, Liste kennt Iteratoren nicht

### ▶ 15.5.2 Konsistenzfragen

- Trennung von Listen und Positionen schafft neue Probleme §
- Lösungsidee: List-Objekt verwaltet "seine" Iteratoren.
- Operation wird verweigert, wenn sie einen inkonsistenten Zustand hinterlassen würde
- Idee fragwürdig, widerspricht dem ursprünglichen Ziel der Trennung von Zuständigkeiten

- Fazit: Bisher keine allgemeingültige Lösung gefunden

## 15.6 Fragen der Semantik

### 15.6.1 Ebenen

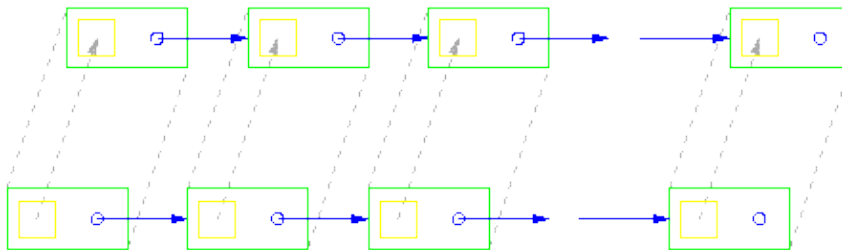
- Elementare Operationen für Klassentypen auf zwei verschiedenen Ebenen

Ebene	Operation	
	Kopieren	Vergleichen
<i>Referenzen</i>	Zuweisungsoperator =	Gleichheitsoperator == (prüft Identität)
<i>Logische Ebene</i>	Methode <code>clone</code>	Methode <code>equals</code> (prüft Gleichheit)

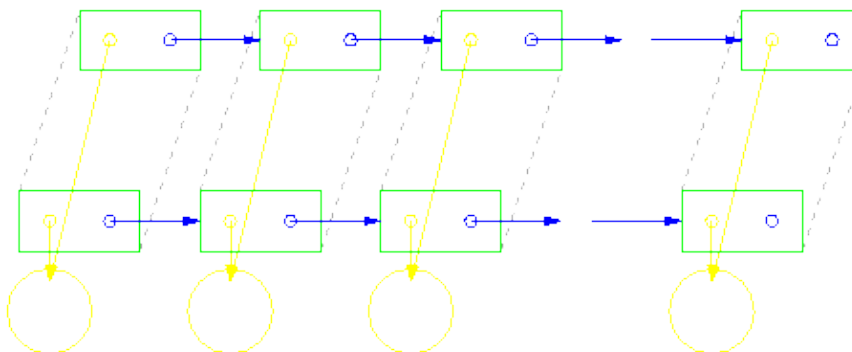
- Bezeichnungen für Kopieren auf "*shallow copy*" (Referenzen) und "*deep copy*" (logische Kopie)
- Für primitive Typen fallen Referenz- und logische Ebene zusammen
- Zu klären für Listen

### 15.6.2 Kopieren und Vergleichen

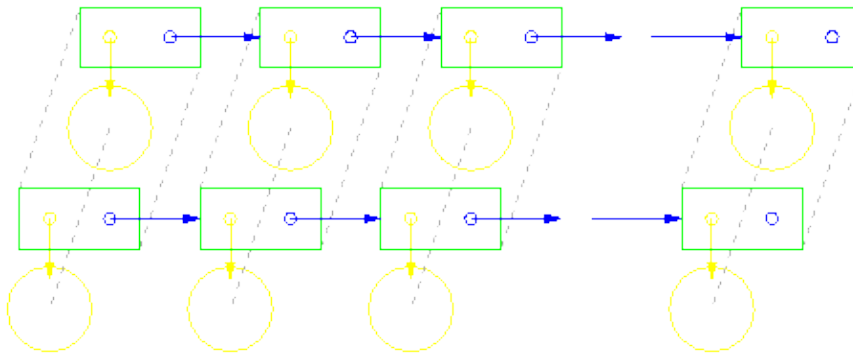
- Logisches Kopieren einer Liste: alle Knoten duplizieren, Duplikate zu neuer Liste verketteten
- Typ der Elemente i. allg. unbekannt § ⇒ wie werden Elemente kopiert?
- "*Shallow copy*" benutzt Zuweisungsoperator: ok für primitive Typen



- Problematisch für Referenztypen: §



- "*Deep copy*" kopiert Elemente logisch



### 15.6.3 Cloning

- Frage: Wie soll Objekt eines unbekanntes Typs (logisch) kopiert werden?
- Einheitliches Verfahren:
  1. **Interface Cloneable** implementieren und
  2. **Methode clone()** definieren
- Klasse `Object` liefert **Vordefinition** der Methode `clone()`, von allen Klassen erbt §
- `Object.clone()` kopiert alle Datenelemente eines Objektes per **Zuweisungsoperator** ⇒ implementiert **kein deep copy**
- `clone` **für deep copy neu definieren**, d.h. `Object.clone()` redefinieren
- Frage: Warum soviel Aufwand?
  1. **Kopierkonstruktor** unbrauchbar:  
wird nicht dynamisch gebunden ⇒ Aufruf erfordert Kenntnis des konkreten Objekttyps
  2. Nicht alle Klassen können (sollen) Objekte duplizieren  
⇒ Implementieren von `Cloneable` optional, nicht automatisch
  3. Logisches Kopierens ist klassen-spezifisch  
⇒ Benutzerdefinierter Code = Redefinition von `clone()` erforderlich

### 15.6.4 Schema für `clone()`

- Jede Klasse erbt `Object.clone()` ⇒ Compiler übersetzt (zunächst) Aufruf von `clone` für jedes Objekt klaglos
- Falls Interface `Cloneable` nicht implementiert ist: Laufzeitfehler `CloneNotSupportedException`
- Abgeleitetes Objekt muß Basisklassenobjekt ebenfalls kopieren ⇒ `clone` der Basisklasse aufrufen via super
- Primitive Datenelemente werden von `Object.clone` kopiert ⇒ müssen nicht explizit kopiert werden
- Nicht-primitive Datenelemente mit weiteren `clone`-Aufrufen kopieren
- Schema für einfache `clone`-Methode einer Klasse `C`

```

public Object clone() throws CloneNotSupportedException
{
    C copy = (C)super.clone();

    // Fuer alle nicht-primitiven Datenelemente vom Typ T...
    copy.element1 = (T)element1.clone();
    ...

    // Kopie zurueckgeben
    return copy;
}

```

- Beispielimplementierung der Klasse List und Hauptprogramm, das Listen kopiert.
- Klasse `List.Iterator` zeigt interessante Variante: Implementiert nur Interface `Cloneable`, *nicht* `clone`:  
Ererbtes `Object.clone()` paßt genau  $\Rightarrow$  Redefinition unnötig

## 15.6.5 Operationen mit kompletten Listen

- Neben den knoten-bezogenen Listenmethoden weitere Operationen sinnvoll, die sich auf komplette Listen beziehen:

### *Liste trennen*

Eine Liste wird an einer bestimmten Stelle in zwei Teillisten gespalten.

### *Liste umdrehen*

Die Reihenfolge der Knoten einer Liste wird umgekehrt (der Kopf wird zum Ende und umgekehrt).

### *Listen zusammenfassen*

Zwei Listen werden hintereinander zu einer Liste verschmolzen.

### *Teillisten ausschneiden und einfügen*

Statt einzelner Knoten werden ganze Teillisten eingeschoben oder ausgesägt.

### *Filtern*

Aus einer Liste werden Knoten herauskopiert und in einer neuen Liste gesammelt, die bestimmte Eigenschaften aufweisen.

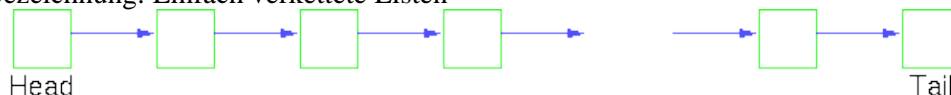
- Die korrekte Verwaltung von Iteratoren wird bei diesen Operationen schwierig.

## 15.7 Weitere verkettete Strukturen

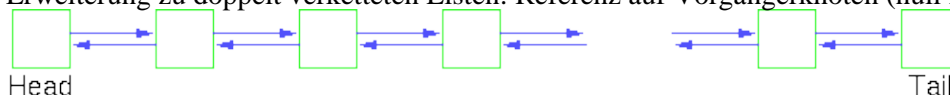
### 15.7.1 Doppelt verkettete Listen

- Bisher gezeigte Listen nur in einer Richtung verkettet  $\Rightarrow$  effizient von vorne nach hinten zu durchlaufen, aber nur langsam umgekehrt

- Bezeichnung: Einfach verkettete Listen



- Erweiterung zu doppelt verketteten Listen: Referenz auf Vorgängerknoten (null für Head)

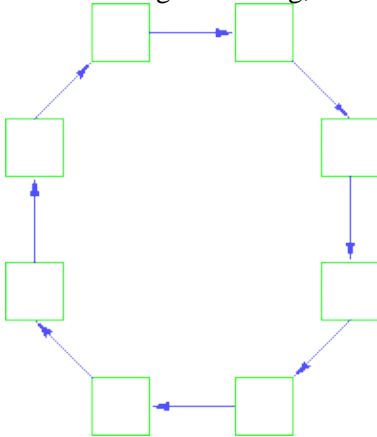


- Durchlauf in beiden Richtungen gleich schnell

- Aber: Listenoperationen komplizierter: müssen zwei Referenzketten intakt halten
- 

## ▶ 15.7.2 Ringe

- Ringe ähnlich wie einfach verkettete Listen
- Zusätzlich: Referenz von Tail auf Head
- Alle Knoten gleichwertig, Head und Tail bedeutungslos



- Doppelt verkettete Ringe: Referenzen auf beide Nachbarknoten
  - Aktueller Knoten oder Iterator frei im Ring beweglich
- 

## ▶ 16 Rekursion

### ▶ 16.1 Grundlagen

---

#### ▶ 16.1.1 Rekursion und Iteration

- Rekursion = elementares Konstruktionsprinzip aus der Mathematik (Induktion), übertragen in die Informatik
  - Alternative zu Schleifen (Iteration)
  - Im ersten Moment verwirrend, aber insgesamt leichter und flexibler als Iteration
  - Formal: Iteration und Rekursion äquivalent
- 

#### ▶ 16.1.2 Beispiel: Zahlensumme

- Ziel: Berechnung von  $\text{sum}(n) = 0 + 1 + 2 + 3 + \dots + n$  für gegebenes  $n$  ( $n$  ganzzahlig, nicht-negativ)
- Beispiel:  $\text{sum}(4) = 0 + 1 + 2 + 3 + 4 = 10$
- Induktiv definiert als

$$\text{sum}(0) := 0$$

$\text{sum}(n) := n + \text{sum}(n - 1)$  für  $n > 0$

- "rekursive" Definition: nimmt Bezug auf sich selbst
- Wichtige Beobachtung: Die Definition nennt
  1. *Abbruchkriterium* für einfachen Fall (" $\text{sum}(0)$ ") und
  2. *Selbstbezug* für einen reduzierten Fall (" $\text{sum}(n - 1)$ ")

### ▶ 16.1.3 Beispiel: Fakultätsfunktion

- "Fakultät"  $\text{fakt}(n) = n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$  für gegebenes  $n$  ( $n$  ganzzahlig, positiv)
- Praxis: Anzahl Möglichkeiten,  $n$  verschiedene Gegenstände in einer Reihe anzuordnen.
- Beispiel:  $\text{fakt}(6) = 6! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 = 720$
- Induktiv definiert als

$\text{fakt}(1) := 1$

$\text{fakt}(n) := n \cdot \text{fakt}(n - 1)$  für  $n > 1$

- gleiches Schema wie Zahlensumme, dieselben Merkmale: Abbruchkriterium und Selbstbezug mit reduziertem Argument.

### ▶ 16.1.4 Beispiel: Fibonacci-Zahlen

- In Worten: Fibonacci-Zahl  $\text{fib}(n)$  ist Summe der beiden vorhergehenden Fibonacci-Zahlen
- Per definitionem: §

$\text{fib}(1) := 1$

$\text{fib}(2) := 1$

- Beispiel:

$\text{fib}(1) = 1$

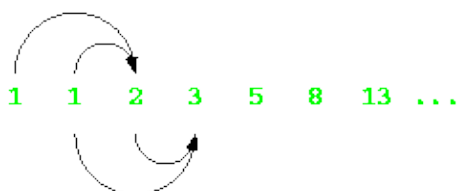
$\text{fib}(2) = 1$

$\text{fib}(3) = \text{fib}(1) + \text{fib}(2) = 1 + 1 = 2$

$\text{fib}(4) = \text{fib}(2) + \text{fib}(3) = 1 + 2 = 3$

$\text{fib}(5) = \text{fib}(3) + \text{fib}(4) = 2 + 3 = 5$

$\text{fib}(6) = \text{fib}(4) + \text{fib}(5) = 3 + 5 = 8$



- Definition formal:

$\text{fib}(1) := 1$

$\text{fib}(2) := 1$

$\text{fib}(n) := \text{fib}(n - 1) + \text{fib}(n - 2)$  für  $n > 2$

- Zeigt obige Merkmale: Abbruchkriterium + Selbstbezug für reduzierten Fall.

## 16.2 Programmiersprachliche Umsetzung

### 16.2.1 Selbstaufruf

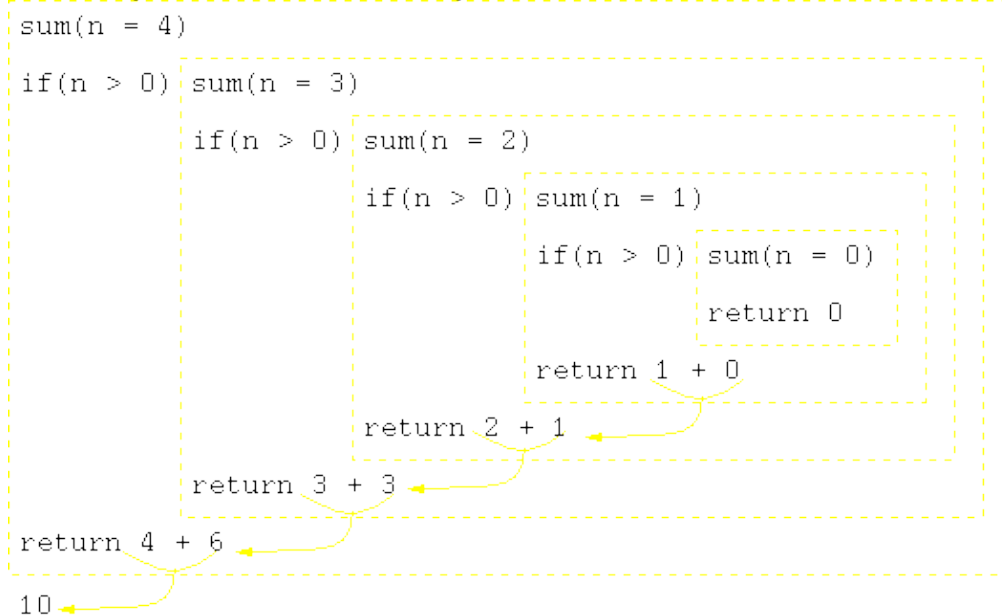
- Formale Definitionen direkt umsetzen in Methode, die *sich selbst* aufruft:

```
int sum(int n)
{
    if(n > 0)
        return sum(n-1) + n;
    else
        return 0;
}
```

- Java (und andere Programmiersprachen) stapeln offene (noch nicht zurückgekehrte) Funktionsaufrufe auf einem dafür reservierten Stack ("Laufzeitstack").
- Jeder Aufruf merkt sich "seinen" Satz lokaler Variablen (einschließlich Funktionsparametern).
- [Beispielprogramm Sum.java](#)

### 16.2.2 Aufrufverschachtelung

- Darstellung der Aufrufverschachtelung



- Im "untersten" rekursiven Aufruf: fünf offene rekursive Aufrufe gestapelt

- Offene Aufrufe bei der Rückkehr aus der Rekursion wieder abgebaut
- Rekursion sichtbar im Programm SumTraced.java:

```
main() called
sum(6) called
sum(5) called
sum(4) called
sum(3) called
sum(2) called
sum(1) called
sum(0) called
sum(0) returns 0
sum(1) returns 1
sum(2) returns 3
sum(3) returns 6
sum(4) returns 10
sum(5) returns 15
sum(6) returns 21
main() returns
```

## ▶ 16.2.3 Komplexe Aufrufverschachtelung

- Implementierung der Fibonacci-Funktion geradlinig entsprechend der Definition:

```
int fib(int n)
{
    if(n == 1)
        return 1;
    else if(n == 2)
        return 1;
    else
        fib(n - 1) + fib(n - 2);
}
```

- Beispielprogramm Fib.java liefert korrekt  $\text{Fib}(8) = 21$
- Modifizierte Version mit Protokollausgabe zeigt komplexen Ablauf

## ▶ 16.3 Äquivalenz zwischen Rekursion und Iteration

### ▶ 16.3.1 Endrekursion (*tail recursion*)

- Wie weit sind Rekursion und Iteration gleichwertig?
- Für bestimmte, einfache Fälle bekannt: Systematisches Transformationsverfahren zwischen iterativer und rekursiver Lösung
- Direkte rekursive Implementierung der Summenfunktion addiert bei der *Rückkehr* aus der Rekursion: Schematisch:

$$\begin{aligned} & \text{sum}(4) \\ &= 4 + \text{sum}(3) \\ &= 4 + 3 + \text{sum}(2) \end{aligned}$$

$$\begin{aligned}
&= 4 + 3 + 2 + \text{sum}(1) \\
&= 4 + 3 + 2 + 1 + \text{sum}(0) \\
&= 4 + 3 + 2 + \underline{1+0} \\
&= 4 + 3 + \underline{2+1} \\
&= 4 + \underline{3+3} \\
&= \underline{4+6} \\
&= 10
\end{aligned}$$

- Alternative: Ergebnis beim *Abstieg* in Rekursion akkumulieren (Hilfsmittel: zweiter Parameter)

$$\begin{aligned}
&\text{sum}(4, 0) \\
&= \text{sum}(3, \underline{0+4}) \\
&= \text{sum}(2, \underline{4+3}) \\
&= \text{sum}(1, \underline{7+2}) \\
&= \text{sum}(0, \underline{9+1}) \\
&= 10
\end{aligned}$$

- Java-Definition

```

int sum(int n, int s)
{
    if(s == 0)
        return s;
    else
        return sum(n - 1, n + s);
}

```

- Ergebnis steht am tiefsten Punkt der Rekursion fest, bei der Rückkehr *nichts mehr* zu berechnen
- Ergebnis des rekursiven Aufrufs wird unverändert zurückgegeben
- Bezeichnung "**Endrekursion**" (= *tail recursion*).

## 16.3.2 Schematische Gegenüberstellung

- Allgemeines Schema einer endrekursiven Funktion R:

```

R(zähler, ergebnis)
{
    ende)if(
        ergebnis;return
    else
        neuer_zähler, neues_ergebnis);
}

```

- Schema für äquivalente iterative Funktionen I:

```

I(zähler, ergebnis)

```

```

{
    while(!
        {
    ergebnis = neues_ergebnis;
    zähler = neuer_zähler;
        }
    ergebnis;
}

```

- Gegenüberstellung zeigt: Endrekursion und Iteration äquivalent

### ▶ 16.3.3 Anwendung

- "Tail recursion elimination" = Umformung Endrekursion → Iteration
- Endrekursive Fassung der Fibonaccifunktion

```

/*
n = Kontrollvariable
last = letzte Fibonaccizahl
butlast = vorletzte Fibonaccizahl
*/
int fib(int n, int last, int butlast)
{
    if(n < 2)
        return last;
    else
        return fib(n - 1, last + butlast, last);
}

```

- Iterative Fassung braucht temporäre Variable, um die *gleichzeitige* Änderung der beiden Ergebnisvariablen nachzubilden. §

```

int fib(int n, int last, int butlast)
{
    while(!(n < 2))
    {
        int tmp = last;
        last = last + butlast;
        butlast = tmp;
        n = n - 1;
    }
    return last;
}

```

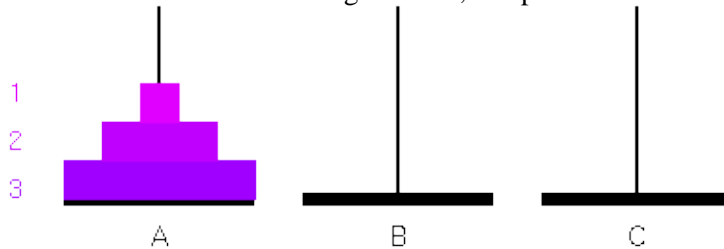
### ▶ 16.3.4 Effizienzfrage

- Methodenaufrufe haben fixen Kosten (Laufzeit, Speicherplatz)
- Iterative Lösung effizienter als rekursive Lösung
- Aber: Ersparte Kosten wiegen selten klaren Entwurf auf ⇒ rekursive Lösung i.allg. vorzuziehen
- Rekursion ist nicht "eleganter", sie nutzt nur einen ohnehin vorhandenen Automatismus (Laufzeitstack) wirksam aus

## 16.4 Beispiel: Türme von Hanoi

### 16.4.1 Problemstellung

- Gegeben: Drei **Stäbe** auf denen  $n$  unterschiedlich große **Scheiben** stecken.
- **Stäbe** und **Scheiben** eindeutig benannt, beispielsweise Stäbe A, B, C und für  $n = 3$  Scheiben 1, 2, 3



- **Ziel:** Kompletten Turm von einem **Startstab** (hier A) auf einen anderen **Zielstab** versetzen, bspweise B
- **Regeln**
  1. Es darf immer nur die oberste Scheibe von einem Stab auf einen anderen versetzt werden
  2. Eine größere Scheibe darf niemals auf einer kleineren Scheibe liegen

### 16.4.2 Lösungsbeispiel

### 16.4.3 Rekursive Lösung

- Methode `hanoi` soll  $n$  Scheiben von einem Stab auf einen anderen transportieren
- Parameter der Methode:

```
void hanoi(int n, int from, int to, int aux)
{
    ...noch zu entwickeln...
}
```

`n`

Anzahl  $n$  der Scheiben, die versetzt werden sollen

`from`

Stab von dem der Stapel abtransportiert wird

`to`

Stab zu dem der Stapel transportiert wird

`aux`

Stab der als Zwischenlager benutzt werden kann

- Idee:
  1. Die oberen  $n-1$  Scheiben des Stapels von `from` auf den Hilfsstab `aux` transportieren
  2. Die unterste (= größte) Scheibe  $n$  des Stapels von `from` auf `to` legen
  3. Die oberen  $n-1$  Scheiben vom Hilfsstab `aux` nach `to` transportieren
- Schritte 1 und 3 durch **rekursive** Aufrufe erledigen
- **Abbruchkriterium:** Für  $n = 1$  entfallen die Schritte 1 und 3  $\Rightarrow$  die (einzige) Scheibe kann sofort transportiert werden

## ▶ 16.4.4 Implementierung

- Oben gezeigte Idee läßt sich direkt in Java umsetzen:

```
void hanoi(int n, int from, int to, int aux)
{
    if(n == 1)
        System.out.println(n + ": " + from + "-->" + to);
    else
    {
        hanoi(n - 1, from, aux, to);
        System.out.println(n + ": " + from + "-->" + to);
        hanoi(n - 1, aux, to, from);
    }
}
```

## ▶ 16.4.5 Ergebnisse

- Implementierung liefert Lösungen für beliebige  $n$
- Beispiel für  $n = 3$ :

```
$ java Hanoi
1: A-->B
2: A-->C
1: B-->C
3: A-->B
1: C-->A
2: C-->B
1: A-->B
```

- Iterative Lösung aufwendiger
- In den "halb angewickelten" rekursiven Aufrufen stecken viele Informationen, die eine iterative Lösung explizit verwalten muß §

## ▶ 16.5 Arten der Rekursion

### ▶ 16.5.1 Anzahl Aufrufe

- Unterschiedliche Arten der Rekursion
- Endrekursion nur *eine* (sehr einfache) Art
- Merkmal: Erwartete Anzahl rekursiver Aufruf (in Abhängigkeit vom Startwert der Kontrollvariablen)
- Andere Arten der Rekursion nur mühsam in Iteration umzusetzen §

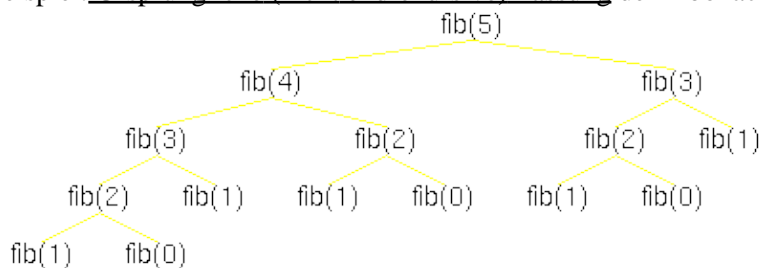
### ▶ 16.5.2 Lineare Rekursion

- Merkmal: Jeder Aufruf löst (maximal) *einen weiteren* Aufruf aus.

- Anzahl rekursiver Aufrufe linear abhängig von Kontrollvariablen
- Beispiele: Fakultätsfunktion, Zahlensumme
- Lineare Rekursion oft als Endrekursion formulierbar  $\Rightarrow$  direkt in Iteration umformbar

### 16.5.3 *Fat recursion*

- Dt. auch "verzweigte Rekursion"
- Merkmal: Jeder Aufruf löst *mehrere weitere* Aufrufe aus
- Aufrufhierarchie zeigt Baumstruktur
- Beispiel: Ursprüngliche (nicht endrekursive) Fassung der Fibonaccifunktion



- Laufzeiten und Anzahl Aufrufe:

Fibonaccizahl	Laufzeit (ms)	Anzahl rekursiver Aufrufe
fib(1)	0	1
fib(2)	0	3
fib(3)	0	5
fib(4)	0	9
fib(5)	0	15
fib(6)	0	25
fib(7)	0	41
fib(8)	0	67
fib(9)	0	109
fib(10)	0	177
fib(11)	1	287
fib(12)	1	465
fib(13)	1	753
fib(14)	1	1219
fib(15)	2	1973
fib(16)	3	3193
fib(17)	6	5167

fib(18)	9	8361
fib(19)	14	13529
fib(20)	63	21891
fib(21)	38	35421
fib(22)	62	57313
fib(23)	100	92735
fib(24)	160	150049
fib(25)	263	242785
fib(26)	414	392835
fib(27)	668	635621
fib(28)	1077	1028457
fib(29)	1778	1664079
fib(30)	2829	2692537
fib(31)	4592	4356617

- Laufzeiten, Anzahl Aufrufe: in etwa Fibonaccifolgen.
- Quotient  $\text{fib}(n+1)/\text{fib}(n) \approx 1.62$  für große  $n$ , also exponentielles Wachstum
- Beispiel: fib(80) auf diesem Weg, mit heutiger Hardware, nicht zu berechnen §
- Endrekursive Fassung liefert fib(80) = 23416728348467685 in Sekundenbruchteilen

## 16.5.4 Compound Recursion

- Dt. auch "verschachtelte Rekursion"
- Merkmal: Ein Argument des rekursiven Aufruf = selbst rekursiver Aufruf
- Beispiel: *Ackermann-Funktion* nach der Definition:

$$\text{ack}(x, 0) := x + 1$$

$$\text{ack}(0, y) := \text{ack}(1, y - 1) \quad \text{für } y > 0$$

$$\text{ack}(x, y) := \text{ack}(\text{ack}(x - 1, y), y - 1) \quad \text{für } x > 0 \text{ und } y > 0$$

- Für kleine Argumentwerte viele Aufrufe (jeweils Funktionsergebnis/Anzahl Aufrufe zur Berechnung):

	x=0	x=1	x=2	x=3	x=4	x=5
y=0	1/1	2/1	3/1	4/1	5/1	6/1
y=1	2/2	3/4	4/6	5/8	6/10	7/12
y=2	3/5	5/14	7/27	9/44	11/65	13/90

y=3	5/15	13/106	29/541	61/2432	125/10307	253/42438
y=4	13/107	65533/?	...	...	...	...

- Kein praktischer Nutzen, interessant für theoretische Informatik
- Anderes Beispiel *Takeuchi-Funktion*

$\text{tak}(x, y, z) := z$  für  $x \leq y$   
 $\text{tak}(x, y, z) := \text{tak}(\text{tak}(x - 1, y, z), \text{tak}(y - 1, z, x), \text{tak}(z - 1, x, y))$  sonst

- Maximiert Anzahl rekursiver Aufrufe bei geringer Rekursionstiefe
- $\text{tak}(18, 12, 6) \rightarrow 63609$  nach 47706 rekursiven Aufrufen, dabei maximale Rekursionstiefe = 18.

## ▶ 16.6 Rekursive Listenoperationen

### ▶ 16.6.1 Strukturelle Rekursion

- Verkettete Listen: "*struktureller*" *Rekursion* (Jeder Knoten referenziert den Nachfolger)
- *Listenoperationen* elegant realisieren mit rekursiven Funktionen, statt Schleifen
- Ausgangspunkt: Knotendefinition mit Listenoperationen
- Instanzdaten in Knoten:

```

class Node
{
    private int element;    // Payload
    private Node next;     // Nachfolger
}
  
```

### ▶ 16.6.2 Liste ausgeben

- Ziel: (Lesende) Operation der Reihe nach mit jedem Knoten
- Beispiel: Liste ausdrucken
- Rekursive Implementierung
  1. Datenelement ausgeben
  2. Falls Nachfolger existiert: Rekursive Anwendung auf Nachfolger
- Konkrete Umsetzung:

```

void print()
{
    System.out.println(element);

    if(next != null)
        next.print();
}
  
```

- `print` produziert Seiteneffekt, verwendet weder Parameter, noch Ergebnis

### ▶ 16.6.3 Ergebnisrückgabe

- Rekursive Funktion mit Ergebnis
- Beispiel: Länge (= Anzahl Elemente) einer Liste bestimmen
- Rekursive Implementierung
  - Falls kein Nachfolger existiert:*  
Länge = 1
  - Ansonsten:*  
Länge = 1 + Länge der Restliste ab Nachfolger

- Konkrete Umsetzung:

```
int length()
{
    if(next == null)
        return 1;
    else
        return 1 + next.length();
}
```

### ▶ 16.6.4 Strukturelle Modifikation

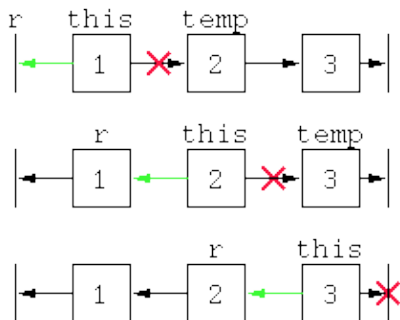
- Rekursive Operation mit Änderung der Knotenfolge
- Beispiele: Einfügen, Löschen
- Löschen eines Knotens `x`: Fälle zu unterscheiden
  - `x` gefunden*  
Nachfolger zurückliefern
  - Liste endet*  
Letzten Knoten zurückliefern
  - ...ansonsten*
    1. Rekursiver Aufruf mit dem Nachfolgerknoten,
    2. Ergebnis als neuen Nachfolgerknoten einsetzen, <sup>§</sup>
    3. aktuellen Knoten zurückliefern
- Konkrete Umsetzung:

```
Node remove(Node x)
{
    if(element == x.element)
        return next;
    else if(next == null)
        return this;
    else
    {
        next = next.remove(x);
        return this;
    }
}
```

## 16.6.5 Neue Liste als Ergebnis

- Ergebnis eines rekursiven Aufrufs: Neue Liste
- Bsp: Liste umdrehen = Referenzen zwischen Knoten invertieren
- Idee: Drei Knoten beteiligt:
  1. Aktueller Knoten
  2. Vorgänger = Kopf der bereits umgedrehten Teilliste
  3. Nachfolger = Kopf der noch unverarbeiteten Restliste
- Elementarschritt:
  1. Referenz zum Nachfolger auf Vorgänger ändern,
  2. Rekursiver Aufruf mit  
Nachfolger → aktueller Knoten und  
Aktueller Knoten → Vorgänger

- Als Skizze:



- Konkrete Umsetzung:

```
Node reverse(Node r)
{
    if(next == null)
    {
        next = r;
        return this;
    }
    else
    {
        Node temp = next;
        next = r;
        return temp.reverse(this);
    }
}
```

- Start mit Aufruf `reverse(null)`

---

## 16.6.6 Implementierung

Siehe [Knotenklasse](#) und [Anwendung \(Hauptprogramm\)](#)

---

## 16.7 Divide-&-Conquer: Quicksort

### 16.7.1 Begriff

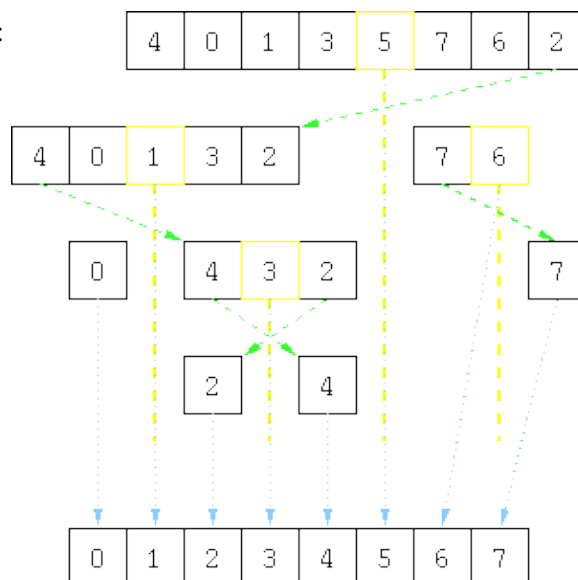
- "*Divide-&-Conquer*" ("Teile und herrsche"): Ein größeres Problem wird in kleinere Teilprobleme aufgebrochen, die (rekursiv) isoliert behandelt werden, bis hinab zum trivialen Fall.
- I.allg. Merkmale verzweigter Rekursion  $\Rightarrow$  nur mühsam iterativ zu realisieren

## 16.7.2 Beispiel: Quicksort

- Ziel: Sortieralgorithmus
- Typische Anwendung des "*Divide-&-Conquer*"  $\Rightarrow$  am besten rekursiv zu realisieren
- Gegeben: Sequenz fester Länge von ungeordneten Elementen
- Voraussetzung: Elemente lassen sich vergleichen, austauschen

## 16.7.3 Ablauf von Quicksort

- Sequenz teilen in zwei Bereiche
- Mittleres Element an mittlerer Position ("Median") auswählen
- Sequenz am Median trennen in zwei Bereiche (willkürlich "vorne", "hinten")
- Alle Elemente  $>$  Median vom vorderen in den hinteren Bereich verschieben
- Entsprechend Elemente  $<$  Median vom hinteren in den vorderen Bereich
- Prozeß mit beiden Bereichen getrennt wiederholen
- Ende wenn nur noch 1 Element im Bereich
- Beispiel:



## 16.7.4 Realisierung

- Elemente seien gespeichert in Sequenz `array`

- Bereiche abgrenzen mit Indexwerten low (exklusive), high (inklusive)
- Schritte zum Bearbeiten eines Bereichs:
  1. Abbruchkriterium prüfen
  2. Median ermitteln (trivial)
  3. In zwei Bereiche trennen
  4. Rekursive Fortsetzung mit beiden Bereichen
- Konkrete Realisierung:

```
void quicksort(int low, int high)
{
    // 1. Abbruchkriterium prüfen
    if(low + 1 >= high)
        return;

    // 2. Median ermitteln
    int m = array[(low + high)/2];

    // 3. In zwei Bereiche trennen
    int p = partition(low, high, m);

    // 4. Rekursive Fortsetzung mit beiden Bereichen
    quicksort(low, p);
    quicksort(p + 1, high);
}
```

## 16.7.5 Hilfsfunktion

- Bereichen trennen mit rekursiver Methode `partition`: Akzeptiert Originalbereich und Median, liefert Grenze zwischen beiden Teilbereichen zurück
- Nachbedingung: Am zurückgegebenen Grenz-Index steht der Median
- Tabelle mit allen möglichen Fällen

	UG<m	UG==m	UG>m
OG<m	1a. <m...<m	2a. m...<m	3a. >m...<m
OG==m	1b. <m...m	2b. m...m	3b. >m...m
OG>m	1c. <m...>m	2c. m...>m	3c. >m...>m

mit

UG = array[low] und

OG = array[high - 1]

- `partition` prüft, ob...
  1. der Bereich nur noch 1 Element groß ist  $\Rightarrow$  Abbruch der Rekursion
  2. das Element am unteren Rand  $<$  Median ist (Fälle 1a, 1b, 1c)  $\Rightarrow$  unteres Element steht richtig, Bereich von unten her verkleinern
  3. das Element am oberen Rand  $>$  Median ist (2c, 3c)  $\Rightarrow$  oberes Element steht richtig, Bereich von oben her verkleinern

Andernfalls werden die Elemente an den Bereichsgrenzen vertauscht. Damit werden die Fälle wie folgt abgebildet:

2a  $\rightarrow$  1b

2b → 2b (unverändert)

3a → 1c

3b → 2c

partition prüft nocheinmal, ob...

4. das Element am unteren Rand  $<$  Median ist (Fälle 1b, 1c)  $\Rightarrow$  unteres Element steht richtig, Bereich von unten her verkleinern

5. Andernfalls bleiben nur die Fälle 2b, 2c  $\Rightarrow$  oberes Element steht richtig, Bereich von oben her verkleinern

- Konkrete Realisierung:

```
int partition(int low, int high, int m)
{
    if(low + 1 >= high)                // 1.
        return low;

    if(compareLess(array[low], m))     // 2.
        return partition(low + 1, high, m);

    if(compareLess(m, array[high - 1])) // 3.
        return partition(low, high - 1, m);

    swap(low, high - 1);

    if(compareLess(array[low], m))     // 4.
        return partition(low + 1, high, m);

    return partition(low, high - 1, m); // 5.
}
```

## ▶ 16.7.6 Verhalten

- Performance: Viel besser als bei einfachen Sortieralgorithmen

- Aber: hängt von einer guten Wahl von  $m$  ab:

*bester Fall:*

es wird tatsächlich immer der mittlere Wert getroffen,  
beide Teile sind immer gleich groß,  
 $\frac{n}{6} \log_2(n)$  Elementpaare werden getauscht (Mittelwert, ohne Beweis).  
dieser Fall ist aber eher unwahrscheinlich.

*schlechtester Fall:*

es wird immer nur 1 Element aussortiert,  
 $n!$  Elementpaare werden getauscht.

*mittlerer Fall:*

bei zufälliger Wahl der Grenze: nur etwa  $1.4\times$  schlechter als im besten Fall.

- Quicksort ist *kein einfacher* Sortieralgorithmus  $\Rightarrow$  Platzbedarf abhängig von der Anzahl Elemente

- Implementierung QuickSorter

- Die abgeleitete Klasse TracedQuickSorter, protokolliert entscheidenden Methodenaufrufe.

## ▶ 16.8 Backtracking

## 16.8.1 n–Damen–Problem

- Gegeben: Schachbrett mit  $n \times n$  Feldern (z.B.  $n = 8$ ) und  $n$  Damen.
- Problem: alle Damen so platzieren, daß sie sich gegenseitig nicht schlagen können. §
- Lösung für  $n = 4$ :

	●		
			●
●			
		●	

## 16.8.2 Lösungsidee

- Beobachtung:
  1. *Exakt* 1 Dame pro Zeile und pro Spalte
  2. *Maximal* 1 Dame pro Hauptdiagonale und pro Nebendiagonale
- Ansatz: Schachbrett zeilenweise von oben nach unten füllen, in jede Zeile 1 Dame platzieren
- In jeder Zeile alle Positionen durchprobieren, ob neue Dame mit den bereits weiter oben platzierten Damen kollidieren würde...
  - Falls ja:*
    - Weiter mit nächster Spalte
  - Ansonsten:*
    - Dame platzieren und prüfen, ob in den restlichen Zeilen Damen platziert werden könnten...
    - Falls ja:*
      - Zurück mit **Erfolg**
    - Ansonsten:*
      - Vorher platzierte Dame zurücknehmen und weiter mit nächster Spalte
- Falls alle Spalten ohne Erfolg: zurück mit **Fehlschlag**
- Typischer "Backtracking"-Algorithmus
- Keine "Intelligenz", probiert systematisch
- Wenn nicht zu viele Möglichkeiten: Lösung in brauchbarer Zeit.

## 16.8.3 Zustand der Berechnung

- Programm muß bereits platzierte Damen aufzeichnen
- "Gedächtnis" als zweidimensionales Array board mit boolean-Elementen (true = Feld mit Dame, false = leeres Feld)

```
boolean[][] board = new boolean[n][n];
```

- Voreinstellung: Spielbrett leer = alle Elemente false

- Array für alle Methoden erreichbar  $\Rightarrow$  Übergabe als Parameter unnötig

## 16.8.4 Implementierung

- Zwei rekursive Methoden:

*queens*

versucht in einer Zeile *z* und allen darunter liegenden Zeilen Damen zu platzieren

*columns*

versucht in einer Zeile *z* ab Spalte *p* eine Dame zu platzieren

- Beide Methoden liefern ein boolean-Ergebnis:

*true*

Dame(n) konnten platziert werden

*false*

Dame(n) konnten nicht platziert werden

- *queens* prüft, ob...

1. die letzte Zeile erreicht ist  $\Rightarrow$  Ergebnis *true*

2. ansonsten in der aktuellen Zeile eine Dame platziert werden kann (Aufruf von *columns*)  $\Rightarrow$  Ergebnis zurückgeben.

- Implementierung:

```
boolean queens(int z)
{
    <n>if(z
        return columns(z, 0);
    return true;
}
```

## 16.8.5 Hilfsmethode

- *columns* unterscheidet die folgenden Fälle:

1. Spaltenposition *p* außerhalb des Spielfeldes  $\Rightarrow$  Fehlschlag, Ergebnis *false*.

2. Ist die aktuelle Spalte besetzt (darüber liegende Felder prüfen)?  $\Rightarrow$  nächste Spalte versuchen, rekursiver Aufruf.

3. Ist die aktuelle Hauptdiagonale besetzt (links oben liegende Felder prüfen)?  $\Rightarrow$  nächste Spalte versuchen, rekursiver Aufruf.

4. Ist die aktuelle Nebendiagonale besetzt (rechts oben liegende Felder prüfen)?  $\Rightarrow$  nächste Spalte versuchen, rekursiver Aufruf.

5. Position ist möglicherweise frei...

a. Versuchsweise Dame aufstellen

b. Können in den weiteren Zeilen ebenfalls Damen platziert werden (rekursiver Aufruf von *queens*)?  $\Rightarrow$  Ergebnis *true*!

c. Versuchsweise aufgestellte Dame wieder zurücknehmen

6. Nächste Spalte versuchen, rekursiver Aufruf.

- Implementierung

```
boolean columns(int z, int p)
{
    if(p >= n) // 1.
        return false;
```

```

for(int i = 0; z - i >= 0; i++) // 2.
    if(board[z - i][p])
        return column(z, p + 1);

for(int i = 0; z - i >= 0 & p - i >= 0; i++) // 3.
    if(board[z - i][p - i])
        return column(z, p + 1);

for(int i = 0; z - i >= 0 & p + i < n; i++) // 4.
    if(board[z - i][p + i])
        return column(z, p + 1);

board[z][p] = true; // 5a.
if(queens(z + 1)) // 5b.
    return true;
board[z][p] = false; // 5c.

return column(z, p + 1); // 6.
}

```

## 16.8.6 Ergebnis

- Implementierung liefert für  $n = 4$  die oben gezeigte Lösung:

```

$ java Queens 4
. . x .
x . . .
. . . x
. x . .

```

- Aufruf für  $n = 8$

```

. . . x . . . .
. x . . . . .
. . . . . x .
. . x . . . .
. . . . . x .
. . . . . . x
. . . . x . . .
x . . . . . .

```

- Läßt sich schnell als korrekt erkennen

## 16.8.7 Alle Lösungen

- Erste Implementierung liefert nur *eine* Lösung, bricht dann ab
- Modifikationen um alle Lösungen zu finden: `columns` fährt fort, falls Lösung gefunden

```

boolean columns(int z, int p)
{
    ...
    if(queens(z + 1))
        /* return true */;
    ...
}

```

- Beispielprogramm liefert:

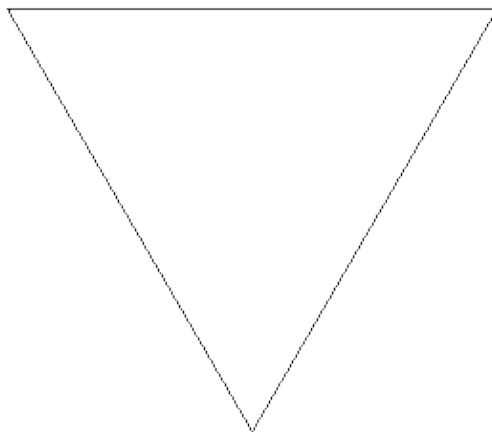
$n$	Anzahl Lösungen	Rechenzeit [sek]
1	0	<1
2	0	<1
3	0	<1
4	2	<1
5	10	<1
6	4	<1
7	40	1.1
8	92	1.1
9	352	1.2
10	724	1.8
11	2680	4.6
12	14200	19.7

- Aufwendiger: rotations- und spiegelsymmetrische Lösungen zusammenfassen, nur 1× zählen

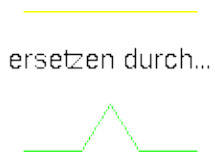
## ▶ Exkurs: Schneeflockenkurve

### ▶ 1 Idee

- Schneeflockenkurve: Folge von Geradenstücken
- Startfigur: gleichseitiges Dreieck



- Konstruktionsregel: Jedes Geradenstück ersetzen durch vier neue Geradenstücke mit je einem Drittel der Originallänge in folgender Anordnung:



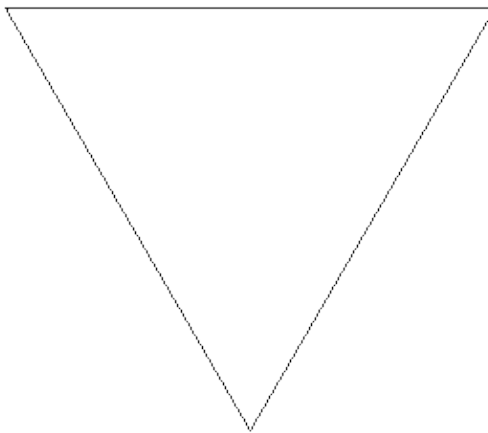
- Ersetzen fortgesetzt (rekursiv) wiederholen

## 2 Ergebnis

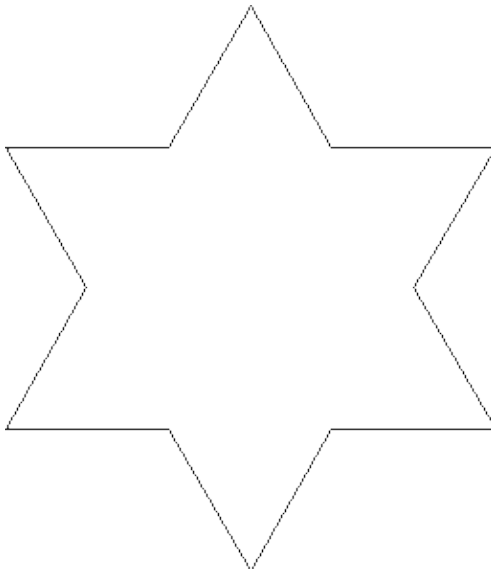
- Ergebnisse

Anzahl Ersetzungsschritte

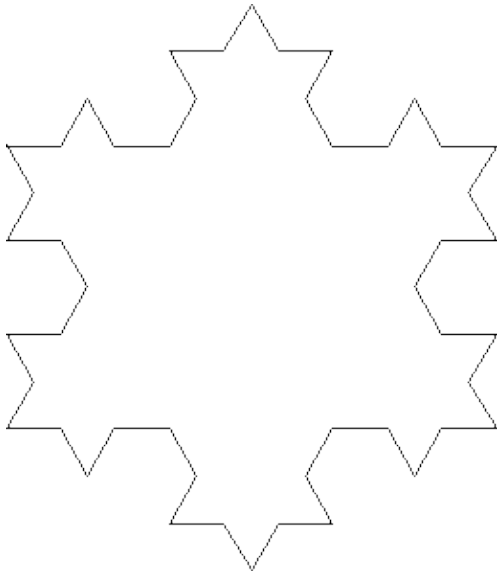
0 (Startfigur)



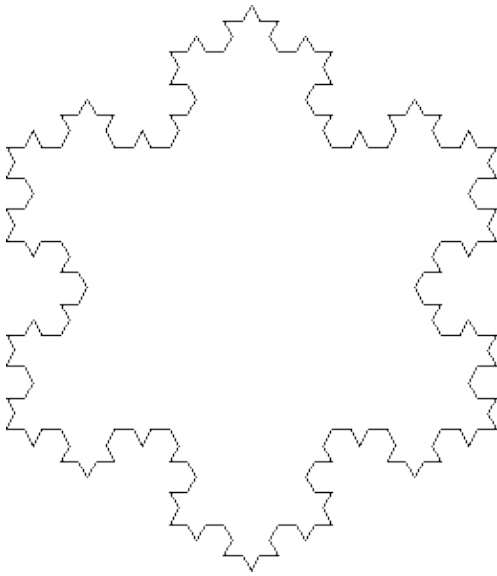
1



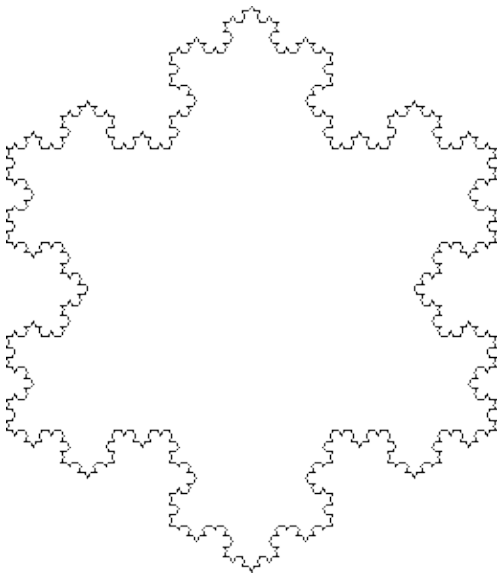
2



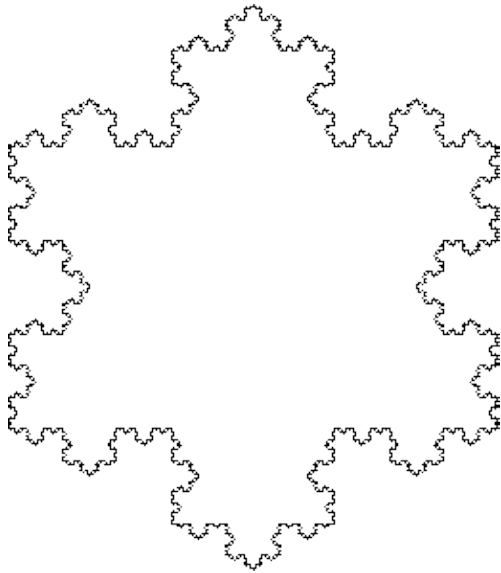
3



4



5



usw...

- Die Kurve zeigt interessante Eigenschaften:
  1. Die umschlossene Fläche ist endlich, weil z.B. ohne weiteres ein umschreibendes Quadrat angegeben werden kann, das eine obere Schranke für die Fläche liefert
  2. Die Länge der Kurve wächst für steigende Rekursionstiefen über alle Schranken
- Bei unbegrenzter Rekursionstiefe ergibt sich ein Kurve unendlicher Länge, die ein endliches Gebiet umschließt.
- Das sind typische Merkmale eines *Fraktals*.

### 3 Bezeichnungen

- Startfigur = *Initiator*
- Konstruktionsregel = *Generator*
- Ändern von Initiator bzw. Generator: Ergebnis ändert sich drastisch
- Schlüsselmerkmale der rekursiven Definition:
  - Abbruchkriterium*  
Maximale Anzahl Ersetzungsschritte
  - Reduktion des Arguments*  
Geradenstücke werden durch *immer kürzere* Geradenstücke ersetzt, niemals durch gleich lange oder sogar längere

### 4 Turtle–Graphik

- Darstellung durch Einsatz einfacher graphischer Routinen ("*Turtle–Graphik*") §
  - `move(double l)`  
fährt um die Distanz `l` weiter in Fahrtrichtung und hinterläßt dabei einen Strich auf dem Papier.
  - `fly(double l)`  
bewegt sich um die Distanz `l` weiter in Fahrtrichtung und hinterläßt dabei *keinen* Strich auf dem Papier.
  - `turn(int d)`

dreht sich auf der Stelle um  $d^\circ$ .

- Implementierung der Schneeflockenkurve mit Hilfe der Turtlegraphik.

## 5 Konfiguration

- Initiator und Generator kompakt definieren mit Strings
- Einzelzeichen in Strings bedeuten

G (*go*) = um Strecke 1 vorwärts

F (*fly*) = um Strecke 1 vorwärts ohne Zeichnen

T (*turn*) = um Winkel  $d$  drehen

- Beispiele:

$n$	$d$	Initiator	Generator	
3	$60^\circ$	GTTTTGTTTTG	GTGTTTTGTG	Schneeflockenkurve)
4	$90^\circ$	GTGTGTG	GTGTTTGTTTGGTGTGTTT	
6	$90^\circ$	GTGTGTG	GGTTTFGTGGTGTGGTTTFTTTGGGG	

- Beispielprogramm Snowflake2.java akzeptiert derartige Strings

## 17 Threads

Siehe auch: [\[The Java Language Specification\]](#)

### 17.1 Idee

#### 17.1.1 Konzept

- **"Thread"** = Kontrollfluß in einem Programm
- Bisher: **Ein einziger** Thread im Programm §
- Jetzt: **Mehrere Threads gleichzeitig** im selben Programm
- Objekte werden geteilt, gemeinsam für alle Threads
- Variablen, primitive Werte, Referenzen für jeden Thread alleine

#### 17.1.2 Golfplätze

- Programm = Golfplatz
- Threads = Spieler
- Variablen, primitive Werte, Referenzen = Schläger, Bälle, ...

- Objekte = Greens
- Synchronisation notwendig!

### ▶ 17.1.3 Anwendungen

- Server, die mehrere Clients bedienen
- Programme mit automatisierten, wiederkehrenden Aufgaben, (Indexerstellung einer Datenbasis, Bereinigung und Optimierung von Datenstrukturen, ...)
- Kommunikation mit langsamen Peripheriegeräten (Drucker, Netzwerkverbindungen)
- Programme mit graphischen Oberflächen
- Fazit: Zunehmend mehr Programme  $\Rightarrow$  Threading mittlerweile unverzichtbar <sup>§</sup>

### ▶ 17.1.4 Threads vs. Prozesse

- Verwandte Konzepte

<i>Prozeß</i>	<i>Thread</i>
Verwaltet im Betriebssystem	Verwaltet im Laufzeitsystem eines Prozesses
Daten getrennt	Gemeinsame Daten
Start aufwendig	Starten schnell
"Heavy weight"	"Light weight"

- Betriebssysteme kennen...
  - ◆ Prozesse und Threads (Sun Solaris)
  - ◆ nur Threads (neuere MS Windows)
  - ◆ nur Prozesse (Linux)
- Java-Threads können...
  - ◆ auf Betriebssystem-Threads abgebildet werden
  - ◆ auf Betriebssystem-Prozesse abgebildet werden
  - ◆ lokal in der JVM verwaltet werden
- Thread-Konzept in Java abstrahiert von Implementierungsabhängigkeiten

## ▶ 17.2 Threadklassen

### ▶ 17.2.1 Klasse Thread

- Vordefinierte Klasse `java.lang.Thread`
- Implementiert "leeren" Thread ohne Funktion
- Eigene Threadklassen ableiten von Thread
- Beispiel

```
class Ticker extends Thread
{...}
```

## ▶ 17.2.2 Erzeugen von Threadobjekten

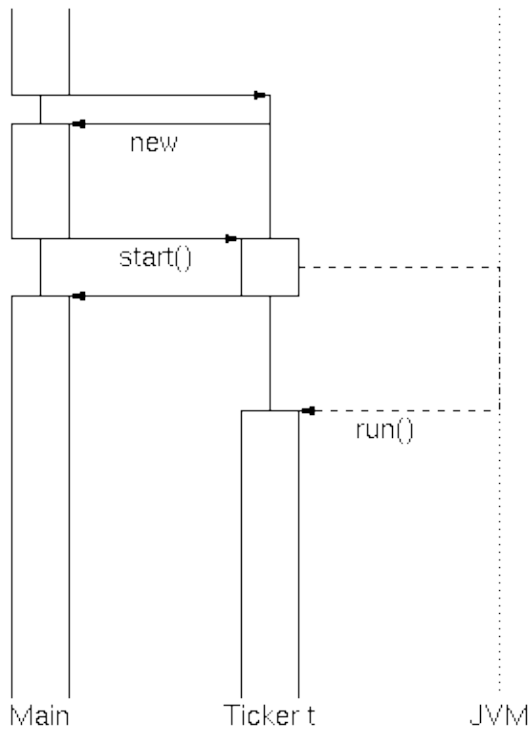
- Threadobjekte mit `new` erzeugen
- Neues Threadobjekt zunächst passiv, wartet auf Startschuß
- Beispiel

```
class Ticker extends Thread
{...}

class Main
{
    public static void main(String[] args)
    {
        new Ticker()t =
    }
}
```

## ▶ 17.2.3 Start von Threads

- Methode `run()` implementiert eigentliche Thread-Aktivität
- `run()` nicht direkt aufzurufen, sondern mittelbar über Methode `start()`
- `run()` = Callback-Methode  $\Rightarrow$  von der VM aufgerufen
- `start()` führt zum Aufruf von `run()`
- `start()` darf nur 1× aufgerufen werden, kehrt *sofort zurück*
- Beispiel
- Als Skizze:



## 17.2.4 Beenden von Threads

- Thread endet mit Rückkehr aus `run()` aus "eigener Kraft"
- Beenden eines Threads durch anderen Thread über "Interrupt" §
- Interrupt senden mit `interrupt()`
- Interrupt bricht Empfänger-Thread *nicht sofort* ab
- Empfänger-Thread kann § Interrupt abfragen, ggf. freiwillig enden
- Test auf Interrupt...
  - ...des laufenden Threads  
mit statischer Methode `interrupted()`
  - ...eines anderen Threads  
mit `isInterrupted()`
- Beispiel
- Beendeter Thread kann nicht mehr fortgesetzt werden

## 17.2.5 Interface Runnable

- Java kennt nur *einfache Vererbung*
- Einschränkung bei Threads: Von Thread abgeleitete Klasse kann von *keiner anderen Klasse* mehr erben
- Ausweg: Implementieren des Interface `Runnable` ⇒ Klasse frei für beliebige andere Basisklasse

- Interface `Runnable` erfordert nur Methode `run()`

## ▶ 17.2.6 Start eines Thread mit `Runnable`-Objekt

- Neues Thread-Objekt erzeugen mit existierendem `Runnable`-Objekt
- Schematisch

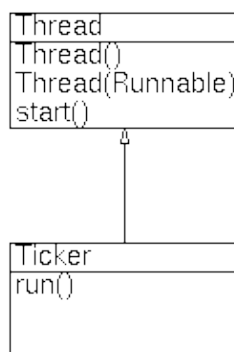
```
class Ticker implements Runnable
{
    public void run()
    {...}
}

public class Main
{
    public static void main(String[] args)
    {
        /*
        Gleichwertig aber kuerzer:
        new Thread(new Ticker()).start();
        */
        Ticker t = new Ticker();
        Thread th = new Thread(t);
        th.start();
    }
}
```

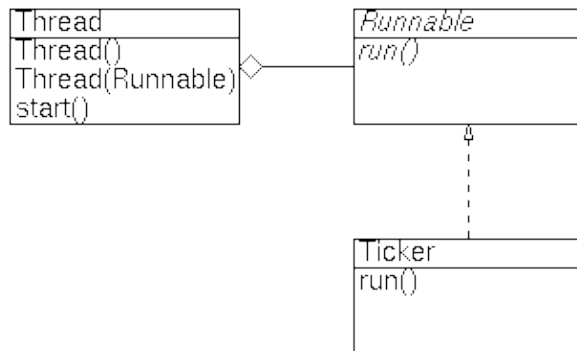
- Unterschied nur im Erzeugen des Thread-Objektes, weiteres Verhalten identisch
- Beispiel

## ▶ 17.2.7 Thread vs. Runnable

- Konstruktionen mit ähnlichem Ziel
- Ableiten von Thread-Klasse nutzt Vererbung:



- Implementieren des `Runnable`-Interface baut auf Delegation:



- Praxis: Lösung über Runnable-Interface flexibler  $\Rightarrow$  häufiger anzutreffen

## ▶ 17.3 Scheduling

### ▶ 17.3.1 Problem

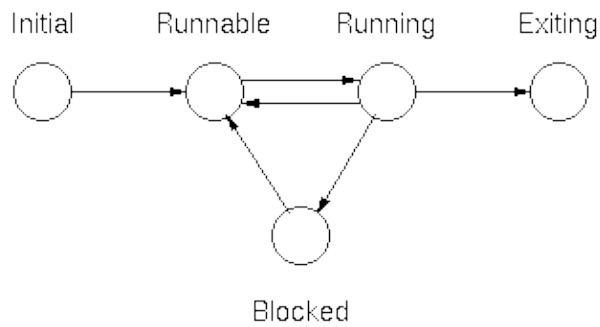
- Maximal *ein* Thread zu einem Zeitpunkt aktiv  $\S$  (= belegt die CPU), aber mehrere Threads existieren
- Nicht jeder existierende Thread kann jederzeit rechnen (Beispiel: wartet auf Tastatureingabe)
- Lösung: Scheduling = Verwaltung von Prozessen, Zuteilung der CPU

### ▶ 17.3.2 Thread-Zustände

- Thread ist zu jedem Zeitpunkt in einem der folgenden Zustände
  - Initial*  
Nach dem Erzeugen, vor `start()`
  - Runnable*  
Bereit zum Rechnen, aber CPU nicht zugeteilt
  - Running*  
Rechnet, belegt CPU
  - Blocked*  
Kann nicht rechnen, wartet auf ein äußeres Ereignis
  - Exiting*  
`run()` beendet, Fortsetzung nicht mehr möglich
- Zustandsmodell übernommen aus Betriebssystemen, dort längst etabliert

### ▶ 17.3.3 Zustandswechsel

- Mögliche Zustandswechsel



- Übergänge
  - Initial* → *Runnable*  
Aufruf von `start()`
  - Runnable* → *Running*  
JVM hat Thread ausgewählt und CPU zugewiesen
  - Running* → *Runnable*  
JVM hat Thread unterbrochen und von der CPU verdrängt
  - Running* → *Blocked*  
Thread hat Methode aufgerufen, die erst nach einem externen Ereignis zurückkehrt
  - Blocked* → *Runnable*  
Externes Ereignis ist eingetreten, Thread könnte jetzt wieder rechnen
  - Running* → *Exiting*  
Methode `run()` ist beendet

### ▶ 17.3.4 Time–Slicing

- Auslöser für Wechsel *Running/Runnable*
  1. Konkurrierender Thread höherer Priorität erreicht Zustand *Runnable*
  2. Zeitquantum (*Time slice*) abgelaufen
- Implementierung von 2. optional, für "korrekte" Implementierung von Java *nicht erforderlich!*
- Umkehrschluß: Programmlogik muß *mit und ohne* Time–slicing funktionieren
- Beispiel: Mehrere Threads § starten mit unterscheidbaren Ausgaben
- Mögliche Ergebnisse:
  - JVM ohne Time–Slicing*  
Nur 1, weder 2 noch 3 §
  - JVM mit Time–Slicing*  
Mischungen aus 1, 2, 3 §

### ▶ 17.3.5 Prioritäten

- Jeder Thread hat eine spezifische Priorität
- Mehrere Threads im Zustand *Runnable*: Kandidat mit höchster Priorität gewinnt
- Thread wechselt Priorität nicht selbst
- Priorität = ganzzahliger Wert im Bereich  
*MIN PRIORITY*

niederste Priorität  
NORM PRIORITY  
normale Priorität  
MAX PRIORITY  
höchste Priorität

- Methoden `setPriority(int)`, `getPriority()` manipulieren Priorität §
- Ohne weitere Maßnahmen: Neuer Thread übernimmt Priorität seines Starters

### ▶ 17.3.6 Selbststeuerung

- Statische Methoden der Klasse Thread sprechen "laufenden Thread" an
- Methode `yield()` löst Übergang *Running* → *Runnable* aus
- Methode `sleep(long)` blockiert Thread für gegebenes Zeitintervall (in Millisekunden) ⇒ löst Übergang *Running* → *Blocked* aus §
- Beispiel: Eine Sekunde warten, dann weiter

```
Thread.sleep(1000);
```

- Beispielprogramm

## ▶ 17.4 Synchronisation

### ▶ 17.4.1 Konkurrierender Zugriff

- Lokale Variablen in Methoden: Getrennt für jeden Thread
- Objekte: Geteilt von allen Threads
- Konkurrierender Zugriff möglich, konkreter Ablauf abhängig vom Scheduling-Algorithmus
- Ein Beispielprogramm läuft binnen kurzer Zeit auf Fehler

### ▶ 17.4.2 Problem

- Frage: Warum scheitert obiges Beispiel?
- Threadwechsel zu *unbestimmten Zeitpunkten*
- Kritischer Fall: Threadwechsel zwischen *Prüfen und Ändern* eines Zählers:

```
if(mycounter.read())
```

- Ablaufskizze:

Zeitpunkt	Worker(1)	Worker(2)
-----------	-----------	-----------

1	<code>if(mycounter.read())</code>	
2		<code>if(mycounter.read())</code>
3		<code>mycounter.up();</code>
4	<code>mycounter.up();</code>	

- Ablauf im einzelnen mit für `mycounter.read() = 4`
  - 1.) `if(mycounter.read())`  
Worker(1) prüft Zähler, stellt fest: `mycounter.read() = 4` ⇒ kann hochgezählt werden
  - 2.) `if(mycounter.read())`  
Worker(2) prüft *den selben* Zähler, stellt fest: `mycounter.read() = 4` ⇒ kann hochgezählt werden
  - 3.) `mycounter.up();`  
Worker(2) zählt hoch auf 5 ⇒ Zähler am Anschlag
  - 4.) `mycounter.up();`  
Worker(1) zählt *nocheinmal* hoch auf 6 ⇒ Überlauf!

### ▶ 17.4.3 Lösungsidee

- Für konkretes Problem: Zwischen **Lesen und Ändern** darf kein anderer Thread das Objekt benutzen
- Allgemein: Objekt für einen bestimmten Thread reservieren, für alle anderen Threads **sperren**
- Java-Konstrukt: "**Lock**" = Objektsperre
- Java-Locks = spezieller Fall allgemeiner **Monitore** §

### ▶ 17.4.4 Objekt-Locks

- Jedes Java-Objekt § hat eigenes Lock
- Lock wird von einem Thread angefordert, zugewiesen, freigegeben
- JVM prüft beim Anfordern: Lock...  
*frei?*  
Thread erhält das Lock, fährt fort, gibt Lock später frei  
*vergeben?*  
Thread wird angehalten, wartet bis Lock freigegeben wird
- Verwaltung der Locks **transparent** für Benutzerprogramm, Abwicklung automatisch innerhalb der JVM

### ▶ 17.4.5 Synchronized-Methoden und -Blöcke

- Schlüsselwort "synchronized": Lock anfordern
- Verlassen des zugeordneten Blocks: Lock freigeben §
- Anwendungsmöglichkeiten §  
*Synchronized-Methode*  
Jeder Aufruf der Methode sperrt das Zielobjekt

```
synchronized Methodenkopf
{
  ...Rumpf...
}
```

### Synchronized-Block

Codeblock sperrt das genannte Objekt

```
synchronized(Object)
{
  ...
}
```

- Beispielprogramm für synchronized-Block
- Wahl der passenden Alternative anhängig vom Kontext, Mechanismus in beiden Fällen derselbe
- Beispiel für synchronized-Methode

## ▶ 17.4.6 Konsequenzen

- Prüfen von Locks kostet Laufzeit ⇒ Wohlüberlegt einsetzen, nicht pauschal verstreuen
- Potentielle Gefahr von **Deadlocks**:  
2 Locks, 2 Threads: Jeder Thread hält ein Lock, fordert das andere an  
⇒ Programm bleibt stecken
- Beispiel für Deadlock
- Problem von Deadlocks sehr vielschichtig ⇒ keine "Patentlösung", Gegenstand aktiver Forschung

## ▶ 17.5 Kommunikation

### ▶ 17.5.1 Idee

- **Pipedstreams** = Alternative zu gemeinsamen Objekten
- Gerichteter Kommunikationskanal
- Zwei gekoppelte Enden (Lesen/Schreiben) in zwei verschiedenen Threads
- Weniger anfällig für Synchronisationsprobleme, `synchronized` nicht notwendig
- Aufwendiger bei Übermittlung komplexer Datenstrukturen §

### ▶ 17.5.2 Pipedstreams

- Von `InputStream` und `OutputStream` abgeleitete Klassen  
`PipedInputStream`  
Lesen von anderen Ende  
`PipedOutputStream`  
Schreiben auf anderes Ende
- Umgang mit Pipedstreams wie mit allen I/O-Streams (read, write, Filter vorsetzen, etc.)

- Pipedstreams nur paarweise benutzbar (`PipedInputStream` und `PipedOutputStream`)
- Einzelner Pipedstream nur Zwischenprodukt, isoliert unbrauchbar zur Kommunikation

### 17.5.3 Aufbau von Pipedstreams

- Zwei Konstruktorpaare zum Aufbau eines Kommunikationskanals
- Defaultkonstruktoren
  - `PipedInputStream()`  
Lesendes Ende eines **neuen** Pipedstreams im laufenden Thread
  - `PipedOutputStream()`  
Entsprechend schreibendes Ende
- Custom-Konstruktoren
  - `PipedInputStream(PipedOutputStream)`  
Lesendes Ende zu einem **bereits existierenden** schreibenden Ende
  - `PipedOutputStream(PipedInputStream)`  
Entsprechend schreibendes Ende zu einem **bereits existierenden** lesenden Ende
- Konstruktionschema allgemein:

```
class Foo extends Thread
{
    Foo(PipedOutputStream output)
    {
        new PipedInputStream(output);
    }

    public void run()
    {
        ...Lesen von input...
    }

    private PipedInputStream input;
}

class Bar
{
    Bar()
    {
        PipedOutputStream new PipedOutputStream();
        new Foo(output).start();
        ...Schreiben auf output...
    }
}
```

- Mehrere verschiedene Pipedstreams zwischen zwei Threads ok (bspweise in jede Richtung einen)

### 17.5.4 Beispiel

- Programm wiederholt fortlaufend...
  1. Zahl von Standardeingabe lesen
  2. Zahl an Worker-Thread schicken (via Pipedstream)
- Workerthread wiederholt fortlaufend...
  1. Zahl vom Pipedstream lesen

2. Langwierige Berechnung durchführen (Zahl verdoppeln)
3. Ergebnis ausgeben

- Vorteil: Programm immer bereit zur Eingabe, eigentliche Arbeit läuft "im Hintergrund" (= in einem anderen Thread) ab

- Programm

## 18 Networking

### 18.1 Netzwerkgrundlagen

#### 18.1.1 ISO/OSI–Referenzmodell

- Veröffentlicht von ISO  $\S$  = *International Standards Organisation*,  
OSI = *Open Systems Interconnection*
- Populäres Schichtenmodell für Netzwerkverbindungen

Schicht	Bezeichnung (Layers)	Aufgaben
7	<i>Application</i>	Spezifisch für einzelne Applikationen, wie Email, Name Services, Remote Procedure Call, ...
6	<i>Presentation</i>	Regelt die Codierung von Daten, wie z.B. Zeichensätze, Verschlüsselung, Kompression, ...
5	<i>Session</i>	Verwaltet Sitzungen, regelt Identifikation, Synchronisation, Rücksetzen bei Dialogfehlern, ...
4	<i>Transport</i>	Stellt zuverlässige Verbindung sicher, d.h. behandelt fehlende, fehlgeleitete, falsch sortierte Datenpakete
3	<i>Network</i>	Sorgt für eine Kommunikationsmöglichkeit zwischen beliebigen Rechnern im Netz, d.h. organisieren Routen usw.
2	<i>Data</i>	Sichert eine Verbindung zwischen zwei gekoppelten Rechnern, behebt Bit–Übertragungsfehler
1	<i>Physical</i>	Netzwerkhardware: Kabel, Adapter, Hubs usw.

- Schicht = Protokoll
- Jede Schicht...
  1. stellt Dienste für darüberliegende Schicht bereit,
  2. nutzt Dienste der darunterliegenden Schicht.
- Klare Abgrenzung erlaubt unabhängige Implementierung einzelner Schichten
- Schichten von unten nach oben zunehmend abstrakter, bequemer (aber auch: komplexer, langsamer)
- Schichten 5–7 manchmal zusammengefaßt, nicht immer konsequent getrennt

## 18.1.2 TCP/IP

- **IP** (Internet Protocol) = eines von verschiedenen Network-Protokollen (Layer 3)
- Alternative **ICMP** für Statusabfragen, z.B. ping
- physikalische Ebene für IP-Verbindung transparent
- Simplex Protokoll, macht keine Zusagen über Zustellung (*send and forget*)
- **TCP** (Transmission Control Protocol) = auf IP basierte, zuverlässige, bidirektionale Verbindung zwischen zwei Hosts. Von den meisten Internet-Anwendungen benutzt
- Alternative **UDP** unzuverlässige Verbindung für Massendaten, z.B. Videostreaming

## 18.1.3 IP-Adressen

- IP-Hosts identifiziert über **Internet-Adressen**
- Internet-Adresse = 4 Byte, eindeutig im Internet §
- Notation: 4 Dezimalwerte der Bytes, wie z.B. 129.187.208.133
- 32 Bits einer IP-Adresse zweigeteilt in **Network-ID** und **Host-ID**
- Die höchstwertigen Bits regeln die Interpretation der restlichen Bits (Network gelb, Host grün):

MSB				LSB	
0	7 Bit Net-ID		24 Bit Host-ID		
1	0	14 Bit Net-ID		16 Bit Host-ID	
1	1	0	21 Bit Net-ID		8 Bit Host-ID
1	1	1	0	reserviert	

- Bezeichnungen

0... = Class-A-Network

10... = Class-B-Network

110... = Class-C-Network

1110... = Class-D

- Bits der Host-ID oft weiter geteilt in Subnet-ID + Host-ID
- **Netmask** = Bitmaske für Net-ID
- Beispiel: IP-Adresse des Gateway der FH-Landshut = 193.175.141.9

dezimal:     193       .175       .141       .9

hexadezimal: C1       .AF       .8D       .9

binär:           11000001 .10101111 .10001101 .00001001  
⇒ Class-C-Network-Adresse  
Netmask = 255.255.255 .0  
Net-ID = 193.175.141  
Host-ID =                   9

---

## ▶ 18.1.4 Domain-Namen

- Zuordnung von Namen für IP-Adressen
  - Beispiel: cel6.fh-landshut.de ⇒ 193.175.215.146
  - Die Auflösung des Namens erfolgt beim Verbindungsaufbau in mehreren Stufen, in der Regel erst über lokale Datei `hosts` und dann über DNS (Domain Name Server)
- 

## ▶ 18.2 Sockets

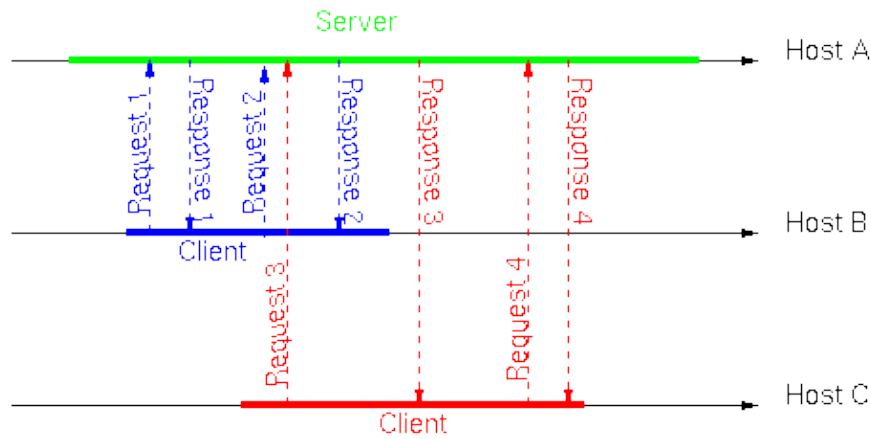
---

### ▶ 18.2.1 Ports

- Hardware-Sicht: gesamter Netzwerkverkehr über Netzwerkschnittstellen (bspweise Ethernet-Adapter, ISDN-Adapter, Modem-Schnittstelle)
  - Logische Sicht (Layer 3): Verkehr aufgeteilt nach *Services* (= Dienste ~ Anwendungen)
  - Jedem Service ein *Port* zugeordnet, nummeriert 0...65535 (2-Byte-Integers)
  - Ports sind reine *Software-Strukturen* (kein Gegenstück in Hardware)
  - Portnummern §
    - 0...255      weltweit verbindlich zugeordnet (*well-known services*)
    - 256...1023   ursprünglich nur für Unix verbindlich zugeordnet, heute allgemein
    - 1024+       frei verfügbar
  - Portnummern sind protokoll-spezifisch §
- 

### ▶ 18.2.2 Client/Server-Modell

- Mehrzahl der Services ist Client-/Server-basiert
- Auf einem Host läuft der Serverprozess (passiv, lauscht auf "Requests")
- Auf anderem Host läuft ein Client (schickt aktiv "Request", erwartet "Response")



- Ein Server bedient viele Clients gleichzeitig, auf einem Host evtl. mehrere Server für unterschiedliche Services
- Client muß Internetadresse + Port des Servers kennen

## 18.2.3 Netzwerk-Dienste

### 18.2.3.1 Echo-Service

- Beispiel für einen sehr einfachen Service: **Echo** § = Requests zurückschicken
- Port = 7, TCP und UDP, lt. /etc/services:

```
...
echo      7/tcp
echo      7/udp
...
```

- Kein dedizierter Serverprozeß, normalerweise von Netzwerk-Subsystem direkt behandelt
- Beispiel für Verbindung (Eingaben unterstrichen) (Telnet wird weiter unten erklärt)

```
$ telnet lucy 7
Trying 192.168.0.2...
Connected to lucy.
Escape character is '^]'.
hi
hi
anybody out there?
anybody out there?
^_

telnet> quit
Connection closed.
$
```

- Vor allem für Testzwecke

### 18.2.3.2 Telnet-Client

- **Telnet-Client:** Universell einsetzbar, erlaubt Dialog mit beliebigem Port auf fernem Host
- Aufruf mit Parametern *Host* und *Port-#* (siehe Beispiel Echo-Service)
- Tastatureingaben an den fernen Port, Rückgaben auf den Bildschirm
- Dialog beenden mit Tastenkombination Control-]
- Für Testzwecke
- Verfügbar auf praktisch jedem System (nicht nur Unix)

### ▶ 18.2.3.3 Telnet-Server

- Akzeptiert Login über Netzwerk §
- Wickelt normale Authentisierung ab (Username, Password)
- Well-known Port = 23 (Voreinstellung für Telnet-Client)
- Erlaubt normalen (textorientierten) Dialog mit fernem Rechner
- Beispiel:

```

$ telnet lupo
Trying 192.168.0.7...
Connected to lupo.localnet.
Escape character is '^]'.

Linux 2.0.35 (lupo.localnet) (ttyp2)

lupo login: gschied
Password:
Last login: Thu Feb 25 10:29:08 from lucy.localnet
$ exit
logout
Connection closed by foreign host.

$

```

Der farbige Dialogabschnitt läuft auf dem fernen Rechner ab.

### ▶ 18.2.3.4 FTP

- "Ftp" = *File Transfer Protocol*
- Service zum **Übertragen von Dateien** zwischen zwei Hosts
- Well-known Port = 21
- Server akzeptiert begrenzte **Ftp-Kommandos**, u.a.:

help    Komplette Kommandoliste

`get file` Datei vom Server zum Client kopieren  
`put file` Datei vom Client zum Server kopieren  
`bin` Ab jetzt Binärmodus (Textmodus) beim Kopieren  
`asc`  
`quit` Dialog beenden

- Ftp wickelt **Authentisierung** ab
- Dialogbeispiel

```

$ ftp lupo
Connected to lupo.localnet.
220 lupo FTP server ready.
Name (lupo:schieder): gschied
331 Password required for gschied.
Password:
230 User gschied logged in.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> get testfile
200 PORT command successful.
150 Opening BINARY mode data connection for testfile (3873 bytes).
226 Transfer complete.
3873 bytes received in 0.0407 secs (93 Kbytes/sec)
ftp> quit
221 Goodbye.
$
  
```

### ▶ 18.2.3.5 Anonymous Ftp

- Öffentliche Ftp-Server akzeptieren Logins von beliebigen Hosts  $\Rightarrow$  keine Authentisierung
- Bezeichnung "Anonymous Ftp" oder "public ftp"
- Konventionen:
  - Login-Name*  
"ftp" oder "anonymous"
  - Passwort*  
eigene Emailadresse oder nur "user@"
- Zugriffe auf public-Ftp-Server protokolliert
- Beispiel: Software-Archiv der TU München via Host `ftp.leo.org`

### ▶ 18.2.3.6 NFS

- NFS = *Network File System*
- Ursprünglich von Sun Microsystems, Inc., inzwischen allgemein verfügbar
- Basiert auf UDP (nicht TCP)
- **NFS-Server** stellt lokale Filesysteme zum Export zur Verfügung (definiert in `/etc/exports`)

- **NFS-Client** importiert Filesysteme vom Server via Mounting, präsentiert sie wie lokale Filesysteme des Clients
  - Resultat: Clients "sehen" Directorybaum des Servers
  - Potentielle Probleme:
    - ◆ Performance (Netzwerklast)
    - ◆ Authentisierung (UIDs auf Server und Clients evtl. nicht kompatibel)
    - ◆ Sicherheit (Network sniffer, ipgrab, &c.)
    - ◆ Zuverlässigkeit (Netzwerkkomponenten, Routen, ...)
  - Trotzdem: Heute weit verbreitet, über Unix hinaus
  - Alternativen: Samba (Windows/Unix-Kopplung), Coda (frei verfügbar, experimentell), AFS (kommerziell)
- 

### ▶ 18.2.3.7 Ping

- Eigentlich kein "Service", kein Server, nur Client-Programm "ping"
- Nutzt ICMP-Pakete, kein Transport-Protokolle
- Sichert Erreichbarkeit eines Hosts, meldet Paketlaufzeiten, vorwiegend zu Testzwecken
- Auf jedem System mit IP-Networking verfügbar → zuverlässig
- Beispiel:

```
$ ping lupu
PING lupu.localnet (192.168.0.7): 56 data bytes
64 bytes from 192.168.0.7: icmp_seq=0 ttl=64 time=0.2 ms
64 bytes from 192.168.0.7: icmp_seq=1 ttl=64 time=0.1 ms
64 bytes from 192.168.0.7: icmp_seq=2 ttl=64 time=0.1 ms
64 bytes from 192.168.0.7: icmp_seq=3 ttl=64 time=0.1 ms

--- lupu.localnet ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.1/0.1/0.2 ms
```

### ▶ 18.2.4 Sockets

- Socket ~ Steckdose in der Wand (eines Prozesses)
  - Sockets auf verschiedenen Hosts **verbunden** → bidirektionaler Datenverkehr
  - In fast allen Programmiersprachen verfügbar (C/C++, Perl, Java, ...)
  - Aus Java Sicht wird ein Socket wie ein Bytestream behandelt.
-

## ▶ 18.2.5 Umgang mit Sockets

- Öffnen von Sockets unterschiedlich für Server und Client
  - Handhabung offener Sockets identisch: primitive Operationen `write`, `read`, `close`
- 

## ▶ 18.2.6 Client–Socket

- Aufbau einer Socketverbindung zum Server
- Schaffen eines neuen Objekts vom Typ `Socket`
- Parameter für Socketkonstruktor
  1. Hostname des Servers als `String`
  2. Portnummer als `int`

- Beispiel:

```
Socket s = new Socket("cel6.fh-landshut.de", 8080)
```

- Schreiben auf Socket: Socketobjekt nach `OutputStream` fragen

- Beispiel:

```
OutputStream os = s.getOutputStream();
```

- Lesen vom Socket: Socketobjekt nach `InputStream` fragen

- Beispiel:

```
InputStream is = s.getInputStream();
```

- Pufferung und Schließen wie bei allen Streams
- 

## ▶ 18.2.7 Server–Socket

- Akzeptieren einer Socketverbindung vom Client
- Schaffen eines neuen Objekts vom Typ `ServerSocket`
- Parameter für Socketkonstruktor
  1. Portnummer als `int`
- Codeskizze für simplen Server

```
ServerSocket ss = new ServerSocket(8080);
while(true)
{
    Socket s = ss.accept();
    ...
    s.close();
}
```

- Schritte im einzelnen:
  1. Serversocket definieren
  2. Warten auf Request von einem Client mit Aufruf von `accept`, blockiert bis Client-Request
  3. Client Request bearbeiten: `accept` kehrt zurück, liefert neuen Socket `s`. `s` ist gekoppelt mit Socket des Clients.
  4. Verbindung zum Client schließen mit `close()`

## ▶ 18.2.8 Mehrfache Requests

- Erster Serverentwurf zeigt gravierendes Problem: Server kann nur *einen* Request auf einmal bearbeiten ⇒ weitere Requests werden nicht angenommen
- In der Praxis nicht haltbar ⇒ § Server muß viele Requests *gleichzeitig* bearbeiten
- Lösung: Clientrequest in eigenem Thread bearbeiten
  - Serverthread*  
startet sofort nächsten Schleifendurchlauf, kehrt sofort zum `accept` zurück, wartet auf neuen Request (Endlosschleife)
  - Clientthread*  
Wickelt Kommunikation mit Client ab, schließt Socket und endet dann
- Arbeitsweise `accept` genau passend konstruiert
- Skizze für brauchbaren Server:

```
ServerSocket ss = new ServerSocket(8080);
while(true)
{
    Socket s = ss.accept();
    new ResponseThread(s).start();
}
```

- Schema für Clientthread-Klasse

```
class ResponseThread extends Thread {
    ResponseThread(Socket s){
        client = s;
    }
    public void run() {
        ...
        client.close();
    }
    private Socket client;
}
```

## ▶ 18.2.9 Beispielprogramme

- Ein minimaler Webbrowser holt sich Seiten von einem Webserver und listet den HTML-Quelltext auf der Konsole auf.
- Aufruf mit den Argumenten Server, Port, URL

```
java NanoBrowser www.sun.com 80 /index.html
```

- Ein ebenso minimaler Webserver liefert immer eine Seite an den Browser.
- Aufruf mit der Portnummer:

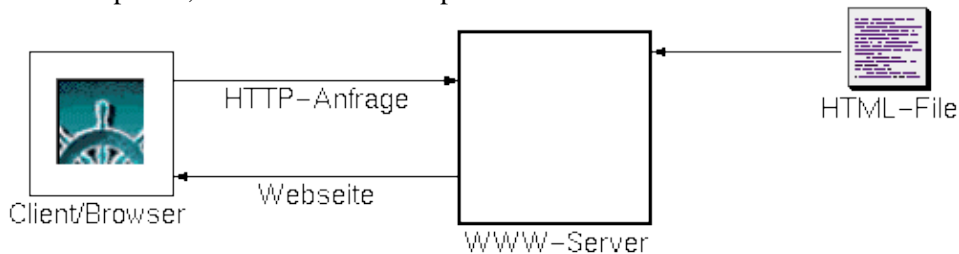
```
java PicoServer 16356
```

- Mit einem Webbrowser zugreifen unter der URL <http://localhost:16356/index.html>.

---

## ▶ 18.3 HTTP

- Hypertext Transfer Protocol für WWW-Anwendungen
- Webserver und Webbrowser ("Client") kommunizieren über HTTP
- Browser = aktiv, schickt Requests an Server
- Server = passiv, liefert Daten als Responses an Browser zurück



- häufige Request Methoden **GET** und **POST**, unterscheiden sich darin wie Daten über die Leitung gehen
- Webserver wartet auf HTTP-Request auf Port 80
- Angabe eines anderen Ports im Request, z.B.:

```
http://www.fh-landshut.de.cel6:8080/index.html
```

- Auszug aus einem Serverprotokoll

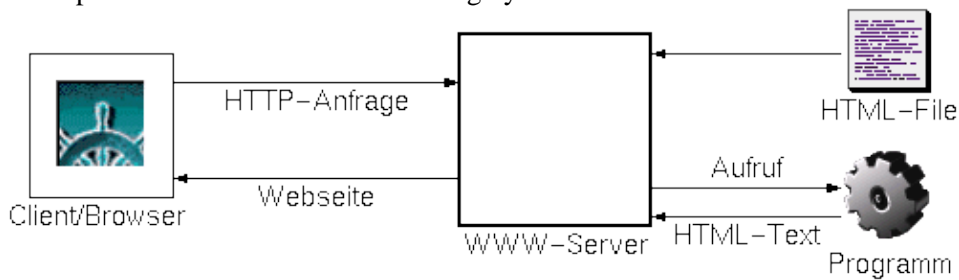
```
192.168.0.7 - - [27 Jul 1998 18:25:19 GMT] "GET /plainpage.html HTTP/1.0" 200 -  
192.168.0.7 - - [27 Jul 1998 18:25:19 GMT] "GET /smiley.gif HTTP/1.0" 200 -
```

---

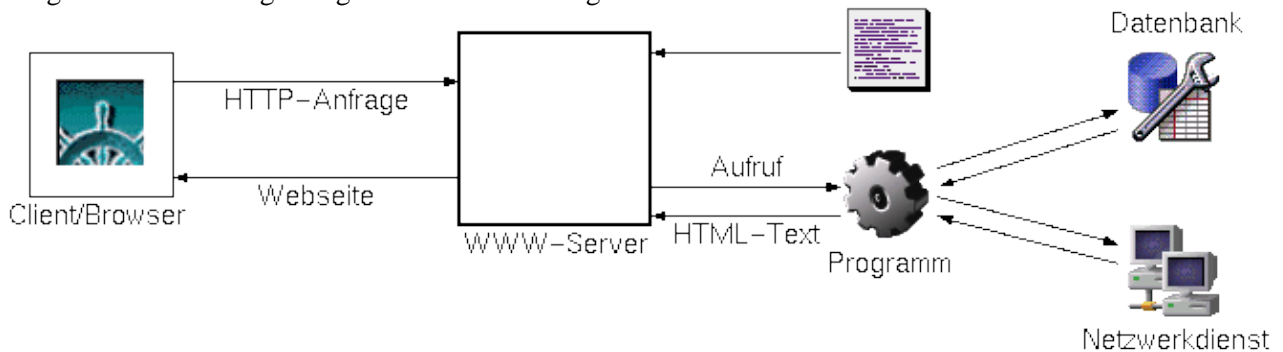
## ▶ 18.4 Statische/ dynamische Seiten

- statische HTML Seite enthält immer gleichen Inhalt, sieht immer gleich aus, in der Regel Kopie eines Files
- dynamische Seite, enthält veränderlichen Inhalt, der bei jedem Zugriff anders aussehen kann
- Webserver ruft externe Programme auf, die dynamische Seiten erzeugen und liefert diese Seiten an den Client zurück
- CGI (Common Gateway Interface) = eine populäre Schnittstelle zwischen Webserver und externen Programmen

- für Java stattdessen Servlets oder JSP
- Prinzipieller Ablauf bei der Generierung dynamischer Web-Seiten



- Programm hat beliebige Möglichkeiten Daten zu gewinnen



## 18.5 Servlets

### 18.5.1 Überblick

- HTML-Code mit Ausgabeanweisungen in Java Code erzeugen
- Request/Response orientiert
- Servlets laufen in einer Umgebung, dem Servletcontainer
- Servletcontainer liefert die Laufzeitumgebung für das Servlet
- Spezialform eines Servlets ist das `HttpServlet`
- für dynamische Webseiten

### 18.5.2 Lebenszyklus eines Servlets

1. Servletcontainer lädt und initialisiert das Servlet; Aufruf der Methode `init(...)`
2. Servlet bearbeitet eine oder mehrere Anfragen; Aufruf der Methode `service(...)`
3. Servletcontainer zerstört das Servlet; Aufruf der Methode `destroy(...)`

- für `HttpServlet`s ist die `service(...)` Methode bereits implementiert
- `init(...)` und `destroy(...)` sind passend vordefiniert
- diese Methode ruft tatsächlich eine der folgenden Methoden auf
  - ◆ `doGet (HttpServletRequest req, HttpServletResponse resp)`
  - ◆ `doPost (HttpServletRequest req, HttpServletResponse resp)`

---

## 18.5.3 Beispiel

```
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWorldServlet extends HttpServlet {
    public void doGet (HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

        String title = "HelloWorldServlet";

        // Methode um Daten an Browser zu schicken
        // setContentType setzt den MIME Type der Daten,
        // die an den Browser geschickt werden sollen
        resp.setContentType("text/html");
        PrintWriter out = resp.getWriter();

        // Schreibe Daten als Antwort
        // mit den println Aufrufen wird ein HTML-Dokument erzeugt
        // und an den Webserver uebergeben
        out.println("<html><head><title>");
        out.println(title);
        out.println("</title></head><body>");
        out.println("<h1>" + title + "</h1>");
        out.println("<p>Hello World!");
        out.println("You are calling form: " + req.getRemoteHost());
        out.println("<br>"+ counter);
        out.println("</body><html>");
        counter++;
        out.close();

    }
    private int counter = 0;
}
```

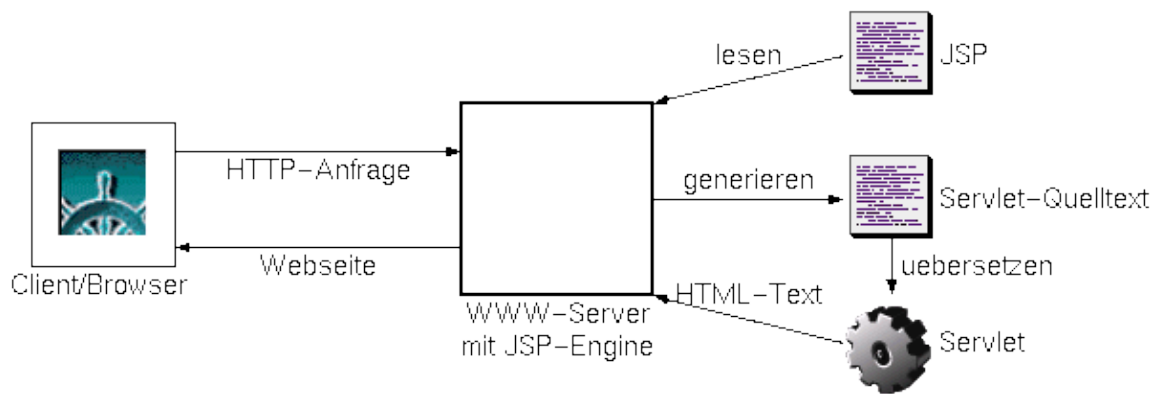
---

## 18.6 Java Server Pages

### 18.6.1 JSP Überblick

- Trennung von request/response Verarbeitung, Logik und Darstellung
- JSP-Seite besteht aus Template-Text, in der Regel HTML, und JSP-Elementen
- Server braucht eine JSP-Engine z.B. Tomcat um JSP Seiten zu verarbeiten
- JSP Technologie basiert auf Servlets
- JSP-Seite wird in ein Servlet umgewandelt, das dann ausgeführt wird

## Single page



### 18.6.2 JSP Elemente

1. **Direktiven** für Informationen über die Seite z.B.  
definiert seitenabhängige Attribute  
für Includedateien
2. **Action Elemente** führen Aktionen durch z.B.  
`<jsp:useBean>`  
macht eine JavaBean in der Seite bekannt  
`<jsp:getProperty>`  
holt einen Property Wert einer JavaBean und fügt ihn der Antwort hinzu  
`<jsp:setProperty>`  
setzt einen JavaBean Property Wert
3. **Scripting Elemente** fügen Java-Code hinzu z.B.  
`<% ... %>`  
enthält Java-Code  
`<%= ... %>`  
Ausdrücke, deren Ergebnis der Antwort hinzugefügt werden soll  
`<%! ... %>`  
Deklarationen für Variablen und Methoden

### 18.6.3 Aufbau einer JSP Seite

```
<%@ page import="java.util.*"%>

<html>
<head>
<title> Hello World </title>
</head>
<body>
  <h1>Hello World!</h1>
  es ist jetzt <%= new Date().toString() %>
</body>
</html>
```

### 18.6.4 JavaBean

- JavaBean oder Bean <sup>§</sup> ist eine Javaklasse, die nach bestimmten Konventionen aufgebaut ist
- kapselt die Funktionalität einer JSP
- eine Bean wird von einem Serverprozess erzeugt
- der JSP-Engine muss der Typ der Bean mitgeteilt werden z.B.

```
<jsp:useBean id="myclock" class="MyClock">
```

- die dazugehörige Klasse MyClock.java

## ▶ 18.6.5 JavaBean Properties

- die Datenelemente einer Bean entsprechen den *Properties*
- für jedes Datenelement müssen in der Bean Setter und Getter definiert werden nach folgendem Beispiel:

```
public String getUsername() {...}
public void setUsername(String s) {...}
```

- ein Beispiel für eine Bean mit Datenelementen und zugehörigen Methoden
- Setzen der Datenelemente in der JSP–Seite erfolgt mit:

```
<jsp:useBean
  id="theBean"
  scope="session"
  class="UserInfoBean">
  <jsp:setProperty name="theBean" property="*" />
</jsp:useBean>
```

- JSP § zum Abfragen der Datenelemente aus der obigen Bean

## ▶ 19 AWT

### ▶ 19.1 GUI–Elemente

#### ▶ 19.1.1 Komponenten

- **Component** = Basisklasse für alle Bestandteile graphischer Oberflächen
- Zwei Gruppen: Dialogelemente und Container
- **Dialogelemente** = konkrete Knöpfe, Textfelder, Scrollbars etc.
- **Container** = rechteckiger Rahmen um untergeordnete Komponenten

#### ▶ 19.1.2 Dialogelemente

- AWT bietet Klassen für fertige Dialogelemente

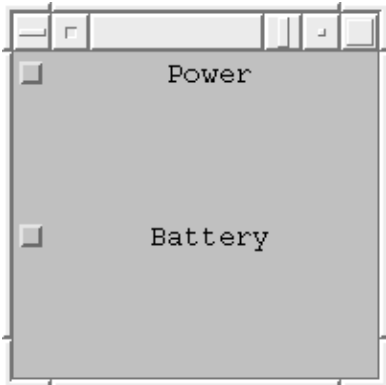
Button Schaltfläche mit Beschriftung



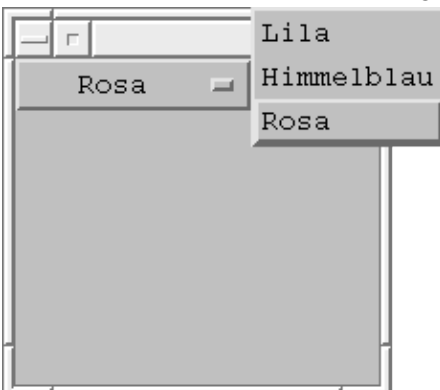
Label feststehender Text (passiv)



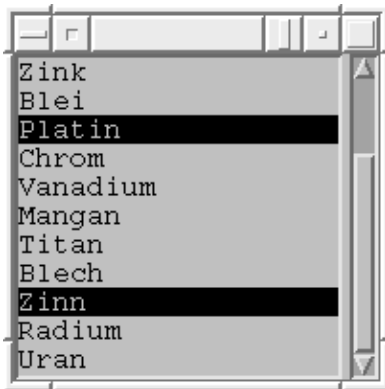
Checkbox Knopf mit Text



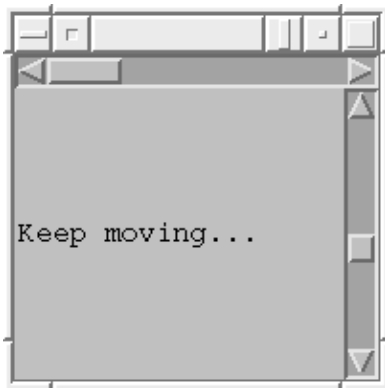
Choice Auswahl aus fester Menge



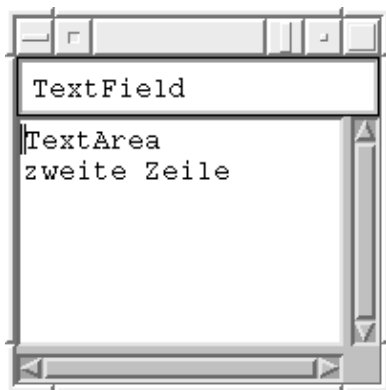
List scrollbare Auswahlliste



Scrollbar Schieber (quer oder senkrecht)



TextField Text-Eingabezeile  
TextArea Text-Eingabefeld (mehrzeilig)



### 19.1.3 Container

- Komponenten in Container einbauen mit `Container.add(Component c)`
- Hierarchischer Aufbau = Baum
- Frame = Konkrete Klasse für Container *mit Dekoration*
- Größe festlegen mit `Component.setSize`
- Sichtbar machen mit `Component.setVisible`

- Beispiel ClickMeButton.java produziert folgendes Fenster:



---

## ▶ 19.1.4 Ableiten als Konstruktionsmittel

- Neue Klasse von vordefinierter Basisklasse ableiten
- Explizite Konstruktion des Frames kann entfallen
- Beispiel
- Vergleich der Ansätze:
  - Aggregation  
GUI-Objekt enthält Frame-Objekt, richtet Methodenaufrufe an dieses
  - Ableiten  
GUI-Objekt ist von Frame abgeleitet, reicht Methodenaufrufe per dynamischem Binden weiter

---

## ▶ 19.2 Plazieren von GUI-Elementen

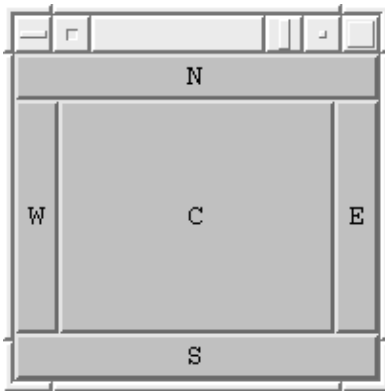
### ▶ 19.2.1 Layout

- "**Layout**" = Art der Anordnung von Komponenten in einem Container
- Wird von bestimmten Objekten geregelt: "LayoutManager"
- Jedem Container ist ein Layoutmanager zugeordnet
- Voreingestellter Layoutmanager für Frame: BorderLayout
- Layoutmanager können ausgetauscht werden

---

### ▶ 19.2.2 BorderLayout für Frames

- Methode Container.add ist überladen mit zweitem Parameter zu `add(String, Component)`
- String-Parameter wird von jedem Layoutmanager passend interpretiert
- BorderLayout akzeptiert z.B. Himmelsrichtungen ("N", "W", "E", "S" und "C" für *Center*)
- Beispiel mit fünf Buttons produziert folgendes Fenster:



## 19.2.3 Layout-Manager

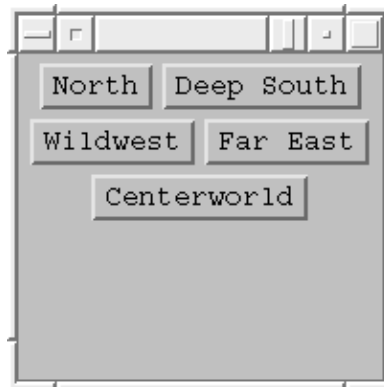
- AWT gibt verschiedene Layoutmanager vor

BorderLayout

wie oben

FlowLayout Komponenten wie Schriftzeichen zeilenweise von oben nach unten

Beispielprogramm



GridLayout festes Raster einstellbarer Größe

Beispielprogramm



GridBagLayout sehr flexibel, kompliziert

- Neue Layoutmanager können definiert werden

## 19.3 Dialogereignisse

### 19.3.1 Events

- Dialogereignisse werden als "Events" sichtbar

- Alle Klassen zum Eventhandling im Package `java . awt . event`
  - Events werden vom AWT generiert und zugestellt
  - Klassifiziert nach Auslöser (Maus, Tastatur, Änderung am Fenster, Focus, ...)
  - Einzelheiten in `Event`-Objekte verpackt, Auswertung nach Bedarf
- 

### ▶ 19.3.2 Low-Level- und semantische Events

- "Low-Level-Events": direkt vom Fenstersystem (Mausbewegungen, Tastendrucke, Focus, ...)
  - "Semantische Events" beziehen sich auf bestimmte Dialogelemente (Button, Scrollbar, ...)
  - Interessant sind i.d.R. semantische Events, seltener low-level-Events
- 

### ▶ 19.3.3 Listener und Callbacks

- Verarbeitung von Events über vorbereitete Objekte ("Listener"), z.B. `ActionListener` für `ActionEvents`
  - Listener werden *vorab* an Dialogelemente gebunden ("registriert"), z.B. mit `Button.addActionListener(ActionListener)` an Buttons
  - AWT ruft bei Events bestimmte Methoden ("Callbacks") im Listener, z.B. `actionPerformed(ActionEvent)`
  - AWT definiert Listener-Interfaces mit erforderlichen Methoden
  - Beispiel
- 

### ▶ 19.3.4 Adapter

- Einige Listener-Interfaces deklarieren zahlreiche Methoden, z.B. `MouseListener`
  - AWT bietet Default-Implementierungen als "Adapter", z.B. `MouseAdapter`
  - Adapter implementieren alle erforderlichen Methoden mit leeren Rümpfen
  - Adapter ableiten zu sinnvollen Listnern, dabei *nur interessante* Methoden redefinieren
  - Beispiel
- 

### ▶ 19.3.5 Innere Klassen

- Sprachmittel wie früher verwendet
  - Vereinfacht die Anordnung von Listnern
  - Beispiel
-

## ▶ 19.3.6 Anonyme Klasse

- Von Listnern wird meist nur *ein* Objekt gebraucht
- Klassendefinition und Objekt–Allokieren zusammenfassen
- Syntax

```
new Baseclass()
{
  // redefined & additional elements
}
```

- Gültigkeitsbereiche sind geschachtelt
- Beispiel

## ▶ 19.4 Zeichnen

### ▶ 19.4.1 Graphische Primitive

- Canvas ("Leinwand") = Zeichenfläche für frei gestaltbare Grafiken
- Komponente neben vordefinierten Dialogelementen (Button, TextArea, ...)
- Basisklasse Canvas produziert leere Fläche
- Abgeleitete Benutzerklassen produzieren sinnvolle Darstellungen

### ▶ 19.4.2 paint und repaint

- AWT ruft Methode `Canvas.paint(...)` zur Darstellung einer Zeichnung
- `paint` = "Callback"–Methode
- AWT entscheidet selbst, wann `paint` notwendig ist (*nicht* explizit aufzurufen!)
- Um Neudarstellung auszulösen: `Component.repaint()` benutzen
- Beispiel

### ▶ 19.4.3 Graphik–Kontext

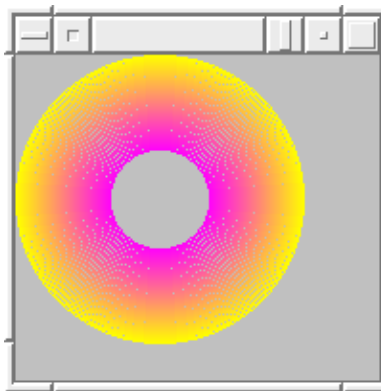
- AWT liefert Graphikkontext als Parameter für `paint`
- Graphikkontext hat Typ `Graphics`
- `Graphics`–Objekt repräsentiert die Zeichenfläche
- Graphik–Operationen richten sich an `Graphics`–Objekt

## 19.4.4 Farben

- Farben = Objekte des Typs `Color`
- Konstruktoren erlauben Definition beliebiger Schattierungen durch Mischung aus Primärfarben, z.B. gleichwertig

```
new Color(0x00FFAAAA)
new Color(255, 170, 170)
new Color(1.0, 0.65, 0.65)
```

- Grundfarben als Konstanten vordefiniert, z.B. `Color.pink`
- Wiedergabe auf konkreter Maschine hängt vom Graphik-Subsystem ab
- Ausgabe des Beispielprogramms:



## 19.4.5 Schriften

- Schriften = Objekte des Typs `Font`
- Konstruktoren erlauben Vorgabe gewünschter Eigenschaften

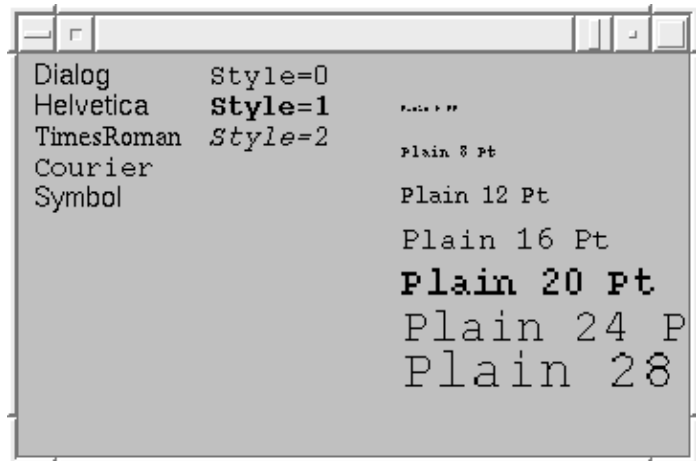
```
new Font(String type, int attr, int pts)
```

- Argumentwerte:

type=	attr=	pts=
"Dialog"	Font.PLAIN	8 10 12 14...
"Helvetica"	Font.ITALIC	
"Courier"	Font.BOLD	
"TimesRoman"	Font.ITALIC + Font.BOLD	
"DialogInput"		

- Grundvorrat wird als verfügbar vorausgesetzt
- Wiedergabe auf konkretem System hängt von den tatsächlich installierten Schriften ab

- Ausgabe des Beispielprogramms:



## 20 Applets

### 20.1 HTML–Quelltext

- HTML beschreibt Inhalt und Struktur eines Dokumentes, aber *nicht*: Darstellung
- Inhalt = Klartext, Struktur = Markups
- Beispiel
- Als HTML–Quelltext

```
<html>
  <head>
    <title>Titelzeile</title>
  </head>
  <body>
    <h1>&Uuml;berschrift</h1>
    Inhalt der Webseite mit einem Bild:
    <br>
    <img src=smiley.gif>
  </body>
</html>
```

### 20.2 HTTP–Protokoll Webserver

- Webserver und Webbrowser ("Client") kommunizieren über HTTP
- Browser = aktiv, schickt Requests an Server
- Server = passiv, liefert Daten als Responses an Browser zurück
- Auszug aus einem Serverprotokoll

```
192.168.0.7 - - [27 Jul 1998 18:25:19 GMT] "GET /plainpage.html HTTP/1.0" 200 -
192.168.0.7 - - [27 Jul 1998 18:25:19 GMT] "GET /smiley.gif HTTP/1.0" 200 -
```

### 20.3 HTTP–Request

- HTTP ist standardisiert
- Requests haben festen Aufbau, ebenso Responses
- Beispiel für expliziten Dialog mit einem Server:

```
$ telnet 192.168.0.7 22722
Trying 192.168.0.7...
Connected to lupo.localnet.
Escape character is '^]'.
GET /plainpage.html HTTP/1.0

HTTP/1.0 200 OK
Server: NetForge/0.40
Date: Mon, 27 Jul 1998 18:27:38 GMT
Content-Type: text/html
Last-Modified: Mon, 27 Jul 1998 18:23:55 GMT
Content-Length: 195

<html>
  <head>
    <title>Plain vanilla page</title>
  </head>
  <body>
    <h1>Header</h1>
    Text giving page content. Here is an image:
    <br>
    <img src=smiley.gif alt="smile!">
  </body>
</html>
Connection closed by foreign host.
```

---

## 20.4 Webseite mit Applet-Tag

- Webseiten enthalten Komponenten, wie Bilder, Klänge, Applets
- Applet = Mini-Anwendung
- Applet-Bytecode kommt vom Server, wird im Browser ausgeführt
- Webseite mit Applet
- Webseite im HTML-Quelltext

```
<html>
  <head>
    <title>Titelzeile</title>
  </head>
  <body>
    <h1>&Uuml;berschrift</h1>
    Inhalt der Webseite mit einem Applet:
    <br>
    <applet code=Smiley.class height=64 width=64>
    </applet>
  </body>
</html>
```

---

## 20.5 Statisches Applet

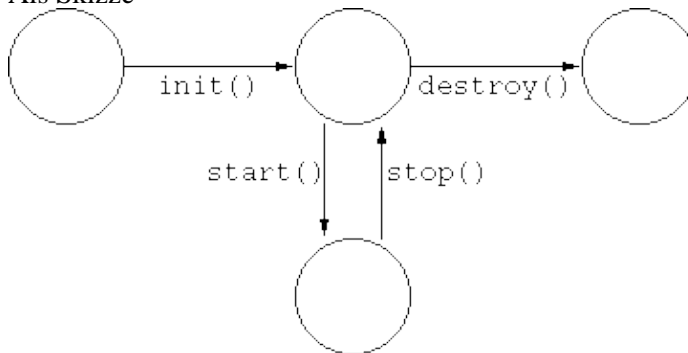
- Applet = Javaprogramm in besonderem Kontext
- Abgeleitet von Basisklasse Applet
- Methode `paint (Graphics )` produziert Darstellung
- Quelltext eines Applets

## 20.6 Lifecycle eines Applets

- Vier verschiedene Aufrufe durch den Browser:

`init()`      1× beim ersten Start des Applets  
`start()`      wenn das Applet in der Webseite sichtbar wird  
`stop()`        wenn das Applet in der Webseite verschwindet  
`destroy()`    1× am Ende

- Als Skizze



## 20.7 Animiertes Applet

- Applet kann Thread starten, um Darstellung fortlaufend zu ändern
- Thread ruft periodisch `repaint()`, `repaint()` mündet in `paint()`
- Quelltext eines Applets
- Webseite mit animiertem Applet

## 20.8 Appletparameter

- Applet-Tag erlaubt Applet-Parameter
- Parameter sind Paare aus "Namen" und "Wert" (beides Strings)
- Beispiel
- HTML-Quelltext mit Appletparametern:

```
<html>
```

```

<head>
  <title>Titelzeile</title>
</head>
<body>
  <h1>&Uuml;berschrift</h1>
  Inhalt der Webseite mit einem Applet, das mit
  unterschiedlichen Parameterwerten initialisiert wird:
  <br>
  <applet code=ColorSmiley.class height=64 width=64>
    <param name=color value=ffff00>
  </applet>
  <applet code=ColorSmiley.class height=64 width=64>
    <param name=color value=c0ff00>
  </applet>
  <applet code=ColorSmiley.class height=64 width=64>
    <param name=color value=ffc000>
  </applet>
</body>
</html>

```

---

## ▶ 20.9 Zugriff vom Applet auf Parameter

- Basisklasse `Applet` bietet Methode `getParameter`
- Liefert Wert des Parameters mit gegebenem Namen
- Beispiel

---

## ▶ 20.10 Applet vs. Application

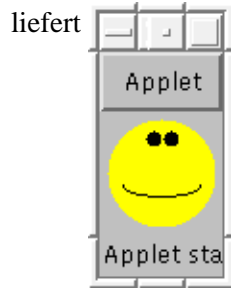
- Javaprogramm kann als Applet *und* Applikation konstruiert werden
- Methode `main()` für Start als "Stand-alone"-Programm
- Basisklasse `Applet` für Einsatz als Applet
- Beispiel

---

## ▶ 20.11 JDK Appletviewer

- Webbrowser verschiedener Hersteller, Versionen
- Java-VM unterschiedlicher Java-Versionen
- JDK liefert Programm `appletviewer` zum Testen von Applets
- Start mit Webseite, beispielsweise

```
appletviewer appletpage.html
```





## 20.12 Security


- Applet-Bytecode wird vom Server geliefert, Browser startet blind
- Sicherheit: Kein Zugriff außerhalb des Browsers = "Sand Box"
- Webseite mit fragwürdigem Applet
- HTML-Quelltext


```
<html>
  <head>
    <title>Titelzeile</title>
  </head>
  <body>
    <h1>&Uuml;berschrift</h1>
    Inhalt der Webseite mit einem Applet,
    <applet code=Footprint.class height=1 width=1>
    </applet>
    <br>das versucht auf das lokale Filesystem zu schreiben.
  </body>
</html>
```


## Material


 Niemeyer, Peck: "Learning Java" (O'Reilly)



 Flanagan: "Java in a Nutshell" (O'Reilly)



 JDK Documentation

 The Java Language Specification

 The Java Tutorial

 Java FAQs List

  Eckel: Thinking in Java

  Java as a First Language

# Seitenverzeichnis

## Vorlesung Programmieren

### 1 Einführung

- 1.1 Erstes Beispiel
  - 1.1.1 Problemstellung
  - 1.1.2 Lösungsidee
  - 1.1.3 Algorithmus
  - 1.1.4 Programm
  - 1.1.5 Übersetzen
  - 1.1.6 Ausführen
- 1.2 Programmiersprachen
  - 1.2.1 Merkmale
  - 1.2.2 Generationen
  - 1.2.3 Eigenschaften
  - 1.2.4 Syntax
  - 1.2.5 Semantik
  - 1.2.6 Pragmatik
- 1.3 Verarbeitungsmodell
  - 1.3.1 Compiler und Maschinencode
  - 1.3.2 Java-Compiler und VM
  - 1.3.3 Ressourcenbedarf
  - 1.3.4 Laufzeitbibliotheken
- 1.4 Java-Quelltext
  - 1.4.1 Forderung an Quelltext
  - 1.4.2 Dateinamen
  - 1.4.3 Layout
  - 1.4.4 Kommentare
  - 1.4.5 Identifizier
  - 1.4.6 Programmrahmen
  - 1.4.7 Java-Versionen
  - 1.4.8 Software
- 1.5 Entwicklungsprozess
  - 1.5.1 Software-Lifecycle
  - 1.5.2 Problemanalyse
  - 1.5.3 Entwurf
  - 1.5.4 Implementierung
  - 1.5.5 Test
  - 1.5.6 Wartung
  - 1.5.7 Kosten

### 2 Grundbausteine

- 2.1 Numerische Ausdrücke
  - 2.1.1 Numerale
  - 2.1.2 Grundrechenarten
  - 2.1.3 Ganzzahlige Division
  - 2.1.4 Geschachtelte Ausdrücke
- 2.2 Operatoren
  - 2.2.1 Priorität
  - 2.2.2 Unäre Operatoren
  - 2.2.3 Assoziativität
  - 2.2.4 Syntax und Operatorentabelle
  - Komplette Operatorentabelle
- 2.3 Variablen und Wertzuweisung
  - 2.3.1 Variablen
  - 2.3.2 Definition

2.3.3 Wertzuweisung	_____
2.3.4 Konstanten	_____
2.3.5 <i>L- und R-Values</i>	_____
2.3.6 Anweisungsarten	_____
2.4 Numerische Typen	_____
2.4.1 Gleitkommawerte	_____
2.4.2 Notation	_____
2.4.3 Polymorphismus von Operatoren	_____
2.4.4 Genauigkeit vs. Geschwindigkeit	_____
2.4.5 Implizite Typkonversion	_____
2.4.6 Explizite Typkonversion ( <i>Typecast</i> )	_____
2.4.7 Konversion numerischer Typen	_____
2.4.8 Wertebereiche	_____
2.4.9 Bereichsüberschreitung	_____
2.5 Bibliotheksfunktionen	_____
2.5.1 Vordefinierte Algorithmen	_____
2.5.2 Notation	_____
2.5.3 Argument und Ergebnis	_____
2.5.4 Signaturen	_____
2.5.5 Laufzeiten	_____
2.5.6 Dokumentation	_____
2.6 Fehler	_____
2.6.1 Fehlerquellen	_____
2.6.2 Syntaxfehler	_____
2.6.3 Statische Semantik	_____
2.6.4 Dynamische Semantik	_____
Exkurs: Programmparameter	_____
1 Konstanten im Quelltext	_____
2 Kommandozeilen-Argumente	_____
3 Tastatureingabe	_____
3 Kontrollstrukturen	_____
3.1 Sequenzen	_____
3.2 Alternativen ( <i>if/else</i> )	_____
3.2.1 Idee	_____
3.2.2 Bedingung	_____
3.2.3 Vergleichsoperatoren	_____
3.2.4 Beispiele für implizite Typkonversion	_____
3.2.5 Beispiele für <i>if</i> -Anweisungen	_____
3.2.6 Zweiseitige Alternativen	_____
3.2.7 Geschachtelte Alternativen	_____
3.2.8 <i>Dangling Else</i>	_____
3.2.9 <i>if</i> -Kaskade	_____
3.3 Wahrheitswerte	_____
3.3.1 Datentyp <i>boolean</i>	_____
3.3.2 Relationale Operatoren	_____
3.3.3 Logische Operatoren	_____
3.3.4 Wahrheitstabellen	_____
3.3.5 Blöcke als Anweisungsgruppe	_____
3.3.6 Teilweise und vollständige Auswertung	_____
3.3.7 Operorentabelle	_____
3.3.8 Boole'sche Variablen	_____
3.4 Schleifen ( <i>while</i> )	_____
3.4.1 <i>while</i> -Schleife	_____
3.4.2 Beispiel: Wertebereich abzählen	_____

3.4.3	Beispiel: Zahlensumme	_____
3.4.4	Beispiel: Collatzfolge	_____
3.4.5	Operatorzuweisungen	_____
3.4.6	Inkrement und Dekrement	_____
3.4.7	Bedingter Operator	_____
3.4.8	Endlosschleifen	_____
3.4.9	Schleifenabbruch mit <code>break</code>	_____
3.4.10	Schleifenkurzschluß mit <code>continue</code>	_____
3.4.11	Geschachtelte Schleifen	_____
3.5	Blöcke	_____
3.5.1	Gültigkeitsbereiche	_____
3.5.2	Lokale und externe Namen	_____
3.5.3	Namenskonflikte	_____
3.5.4	Existenzbereiche	_____
3.6	Zählschleifen ( <code>for</code> )	_____
3.6.1	Syntax und Semantik	_____
3.6.2	Beispiele	_____
3.6.3	Gegenüberstellung mit <code>while</code> -Schleifen	_____
3.7	Verteiler ( <code>switch</code> )	_____
3.7.1	Ersatz für <code>if</code> -Kaskade	_____
3.7.2	Syntax und Semantik	_____
3.7.3	Reihenfolge der <code>case</code> -Klauseln	_____
3.7.4	Duplikate und Lücken	_____
3.7.5	Defaultfall	_____
3.7.6	<i>Fall through</i>	_____
3.7.7	Einschränkungen	_____
3.7.8	Switch als Anweisung	_____
3.8	Algorithmen	_____
3.8.1	Merkmale	_____
3.8.2	Darstellungsformen	_____
3.8.3	Nummerierte Punktliste	_____
3.8.4	Beispiele	_____
3.8.5	Flußdiagramme	_____
3.8.6	Beispiele Flußdiagramme	_____
3.8.7	Struktogramme	_____
3.8.8	Beispiele Struktogramme	_____
4	Textzeichen und Strings	_____
4.1	Textzeichen	_____
4.1.1	Javatyp <code>char</code>	_____
4.1.2	Operationen mit <code>char</code>	_____
4.1.3	Zeichencodes	_____
4.1.4	Zeichensätze	_____
4.1.5	Hexadezimalschreibweise	_____
4.1.6	Ersatzdarstellungen	_____
4.1.7	Bibliotheksmethoden	_____
Exkurs:	Text-I/O	_____
1	Ein- und Ausgabe	_____
2	Beispiel: Lesen eines Textes	_____
3	Beispiel: Text kopieren	_____
4	Beispiel: Zeichen und Zeilen zählen	_____
5	Beispiel: Zwischenraum löschen	_____
6	Beispiel: Textdatei lesen	_____
7	Beispiel: Textdatei schreiben	_____
4.2	Zeichenketten	_____

4.2.1 Motivation	_____
4.2.2 Literale	_____
4.2.3 Javatyp String	_____
4.2.4 Elementzugriff	_____
4.2.5 Methodenaufrufe	_____
4.2.6 Bibliotheksmethoden	_____
4.2.7 Vergleiche	_____
5 Arrays	_____
5.1 Idee	_____
5.1.1 Container	_____
5.1.2 Eigenschaften	_____
5.1.3 Typangabe	_____
5.1.4 Arraytypen	_____
5.2 Referenzsemantik	_____
5.2.1 Definition Arrayvariablen	_____
5.2.2 Arrayvariable und Array	_____
5.2.3 Allokieren	_____
5.2.4 Referenzen	_____
5.2.5 Referenz als Wert	_____
5.2.6 Nicht initialisierte Arrayvariablen	_____
5.2.7 null-Referenz	_____
5.3 Elementzugriff	_____
5.3.1 Indizes	_____
5.3.2 Indexzugriff	_____
5.3.3 Indexfehler	_____
5.3.4 Initialisierung	_____
5.3.5 Array-Literale	_____
5.3.6 Eckige Klammern	_____
5.4 Operationen	_____
5.4.1 Zuweisung	_____
5.4.2 Kopieren	_____
5.4.3 Vergleichen	_____
5.4.4 Abfrage Anzahl Elemente	_____
5.5 Geschachtelte Arrays	_____
5.5.1 Arrays als Elemente	_____
5.5.2 Allokieren	_____
5.5.3 Elementzugriff	_____
5.5.4 Initialisierung	_____
5.5.5 Mehrdimensionales Array	_____
5.6 Anwendungen	_____
5.6.1 Lineare Suche	_____
5.6.2 Aufwand für lineare Suche	_____
5.6.3 Sortieralgorithmen	_____
5.6.4 Voraussetzungen	_____
5.6.5 Idee von <i>Bubble Sort</i>	_____
5.6.6 Algorithmus <i>Bubble Sort</i>	_____
5.6.7 Beispiel <i>Bubble Sort</i>	_____
5.6.8 Binäre Suche	_____
5.6.9 Beispiel binäre Suche	_____
5.6.10 Aufwand für binäre Suche	_____
5.7 Bibliotheksmethoden	_____
5.7.1 Klasse Arrays	_____
5.7.2 Methode <code>arraycopy</code>	_____
6 Klassen	_____

6.1	Klassendefinitionen und Objekte	_____
6.1.1	Ziel	_____
6.1.2	Klassendefinition	_____
6.1.3	Namen von Klassen	_____
6.1.4	Datenelemente	_____
6.1.5	Objekte = Instanzen	_____
6.1.6	Operator new	_____
6.2	Variablen	_____
6.2.1	Definition	_____
6.2.2	Referenzsemantik	_____
6.2.3	Identität vs. Gleichheit	_____
6.2.4	null-Referenz	_____
6.3	Datenelemente	_____
6.3.1	Zugriff	_____
6.3.2	Datenelemente als <i>L-values</i>	_____
6.3.3	Datenelemente unterschiedlicher Objekte	_____
6.3.4	Namensräume	_____
6.4	Methoden	_____
6.4.1	Funktionalität	_____
6.4.2	Definition einer Methode	_____
6.4.3	Methoden und Datenelemente	_____
6.4.4	Kopf und Rumpf	_____
6.4.5	Aufruf	_____
6.4.6	Ablauf eines Aufrufs	_____
6.4.7	Rückkehr	_____
6.4.8	Rumpf als Block	_____
6.5	Argumente und Parameter	_____
6.5.1	Argumente und Parameter	_____
6.5.2	Übergabe	_____
6.5.3	<i>Call Sequence</i>	_____
6.5.4	Primitive Typen als Argumente	_____
6.5.5	Mehrere Parameter	_____
6.5.6	Referenztypen als Parameter	_____
6.5.7	Problematik: Seiteneffekte	_____
6.6	Konstruktoren	_____
6.6.1	Motivation	_____
6.6.2	Syntax	_____
6.6.3	Aufruf	_____
6.6.4	Konstruktoren mit Parametern	_____
6.6.5	Automatisch definierter Konstruktor	_____
6.6.6	Überladen	_____
6.6.7	Aufruf überladener Methoden	_____
6.6.8	Auflösen von Konflikten	_____
6.6.9	Mehrdeutige Aufrufe	_____
6.6.10	Kopier-Konstruktor	_____
6.6.11	Gegenseitiger Aufruf	_____
6.7	Ergebnisrückgabe	_____
6.7.1	Rückgabe eines Wertes	_____
6.7.2	Mehrfache Rückkehrpunkte	_____
6.7.3	Rückgabe einer Referenz	_____
6.7.4	Neues Objekt als Ergebnis	_____
6.7.5	Ergebnislose Rückkehr	_____
6.7.6	Werte- und Referenzsemantik	_____
6.8	Datenkapselung	_____

6.8.1 Ziel: Reduktion von <u>Abhängigkeiten</u>	_____
6.8.2 Aufbau und Operationen	_____
6.8.3 Schnittstelle	_____
6.8.4 Zugriffsschutz	_____
6.8.5 Kunde, Lieferant und Vertrag	_____
6.9 Assertions	_____
6.9.1 Motivation	_____
6.9.2 Syntax	_____
6.9.3 Wirkungsweise	_____
6.9.4 Compilerversionen	_____
6.9.5 Aktivieren und stilllegen	_____
6.9.6 Was zusichern (und <u>was nicht</u> )?	_____
6.9.7 Kontrollfluss	_____
6.9.8 Klasseninvarianten	_____
6.9.9 Pre- und Postconditions	_____
6.10 Statische Datenelemente	_____
6.10.1 Idee	_____
6.10.2 Zugriff	_____
6.10.3 Beispiel: Objekte zählen	_____
6.10.4 Beispiel: Objekte nummerieren	_____
6.10.5 Beispiel: Einzelobjekte (" <u>Singletons</u> ")	_____
6.10.6 Beispiel Singleton: <u>System.out</u>	_____
6.11 Statische Methoden	_____
6.11.1 Idee	_____
6.11.2 Definition und Aufruf	_____
6.11.3 Beispiel: Hauptprogramm	_____
6.11.4 Beispiel: Bibliotheksfunktionen	_____
6.11.5 Beispiel: Objekte zählen	_____
6.11.6 Beispiel: GGT-Methode	_____
6.11.7 Einschränkungen	_____
6.12 Unified Modelling Language <u>UML</u>	_____
6.12.1 Ziel	_____
6.12.2 Merkmale	_____
6.12.3 Statische Klassendiagramme	_____
6.12.4 Struktur einer Klasse	_____
6.12.5 Beziehungen zwischen Klassen	_____
6.12.6 Rollen und Kardinalitäten	_____
6.12.7 Java-Interpretation	_____
6.12.8 Beispiel	_____
6.13 Wie wird eine Klasse definiert?	_____
6.13.1 Rezept	_____
6.13.2 Beispiel	_____
7 Vererbung	_____
7.1 Interfaces	_____
7.1.1 Schnittstelle vs. Implementierung	_____
7.1.2 Interfaces in Java	_____
7.1.3 Beispiel: Komplexe Zahlen	_____
7.1.4 Interface vs. Klasse	_____
7.2 Implementierung	_____
7.2.1 Konkrete Klasse	_____
7.2.2 Alternative Implementierungen	_____
7.2.3 Variablen- und Objekttyp	_____
7.2.4 Methodenauswahl	_____
7.2.5 Dynamisches Binden	_____

7.2.6	Parameterübergabe	_____
	Interfacetypen als Parameter	_____
	Beispiel: Kopierkonstruktor	_____
	Beispiel: Logischer Vergleich von <u>Objekten</u>	_____
7.2.7	Beispiel: Rüsseltiere	_____
7.2.8	Ergebnisrückgabe	_____
7.3	Konkrete <u>Basisklassen</u>	_____
7.3.1	Ausklammern der <u>Grundfunktionalität</u>	_____
7.3.2	Erweitern der <u>Funktionalität</u>	_____
7.3.3	Lösungswege	_____
7.3.4	Vererbung als <u>Erweiterung</u>	_____
7.3.5	Syntax	_____
7.3.6	Zugriffsschutz <u>protected</u>	_____
7.3.7	Beispiel für <u>protected</u>	_____
7.3.8	Methodenaufruf	_____
	Methoden von <u>Basisklassen</u>	_____
	Methoden abgeleiteter <u>Klassen</u>	_____
7.3.9	Modifikation einer <u>Basisklasse</u>	_____
7.3.10	Dynamisches Binden <u>redefinierter Methoden</u>	_____
7.3.11	Einschränken einer <u>Basisklasse</u>	_____
7.3.12	Statisches Binden	_____
7.3.13	Konstruktoren	_____
	Dynamisches Binden	_____
	Automatischer Aufruf	_____
<u>Basisklassen-Default-Konstruktors</u>		
	Custom-Konstruktor einer <u>Basisklasse</u>	_____
	Beispiel	_____
7.3.14	Bezug auf <u>Basisklasse mit super</u>	_____
7.3.15	Selbstreferenz mit <u>this</u>	_____
7.3.16	Beispiel für Einsatz von <u>this</u>	_____
7.3.17	Ableiten als <u>Konstruktionsprinzip am Beispiel Frame</u>	_____
	Vordefinierte <u>Basisklasse java.awt.Frame</u>	_____
	Redefinition von <u>Frame.paint</u>	_____
	Klasse <u>java.awt.Graphics</u>	_____
	Ableiten von <u>java.awt.Frame</u>	_____
	Kontrollfluß	_____
7.4	Abstrakte <u>Basisklassen</u>	_____
7.4.1	Idee	_____
7.4.2	Eigenschaften	_____
7.4.3	Beispiel: Zähler	_____
7.4.4	Rein abstrakte <u>Basisklassen</u>	_____
7.4.5	Mehrfache Vererbung	_____
7.4.6	Mehrfache Interfaces	_____
7.5	Wurzelklasse <u>Object</u>	_____
7.6	Dynamische <u>Typinformation</u>	_____
7.6.1	Statischer und dynamischer <u>Typ</u>	_____
7.6.2	Abgeleitete <u>Klassen mit zusätzlichen Methoden</u>	_____
7.6.3	Lösung: <u>Fette Basisklassen</u>	_____
7.6.4	Lösung: <u>Dynamische Typinformation mit instanceof</u>	_____
7.6.5	<u>Typecast</u>	_____
7.6.6	Beispiel: <u>equals</u>	_____
7.6.7	Typfehler	_____
8	<u>Exception Handling</u>	_____
8.1	Problematik	_____

8.2 Alternativen	_____
8.2.1 Globale Fehlervariable	_____
8.2.2 Beispiel für Fehlervariable	_____
8.2.3 Fluchtwerte	_____
8.2.4 Beispiel: Fluchtwert	_____
8.2.5 Exceptions	_____
8.3 Java-Sprachmittel	_____
8.3.1 Ablauf	_____
8.3.2 Exception auslösen: <code>throw</code>	_____
8.3.3 Beispiel: Exception auslösen	_____
8.3.4 Fehlerbehandlung <code>catch</code> -Klausel	_____
8.3.5 <code>try</code> -Block	_____
8.3.6 <code>finally</code> -Klausel	_____
8.3.7 Schema für Exceptionhandling	_____
8.4 Schachtelung	_____
8.4.1 Dynamische Schachtelung	_____
8.4.2 Beispiel	_____
8.4.3 Geschachtelte <code>try</code> -Blöcke	_____
8.4.4 Programmabbruch	_____
8.5 Exceptionsignaturen	_____
8.6 Exceptionklassen	_____
8.6.1 Throwables	_____
8.6.2 Elemente der vordefinierten Exceptionklassen	_____
8.6.3 Benutzerdefinierte Exceptionklassen	_____
9 Packages	_____
9.1 Arbeitsweise	_____
9.1.1 Motivation	_____
9.1.2 Packages	_____
9.1.3 Dynamisches Laden von Bytecode	_____
9.1.4 Abbildung ins Filesystem	_____
9.2 Umgang mit Packages	_____
9.2.1 Qualifizierte Namen	_____
9.2.2 <code>import</code> -Klausel	_____
9.2.3 <code>package</code> -Klausel	_____
9.2.4 Default-Package	_____
9.2.5 Namenskollisionen	_____
9.2.6 Assertions	_____
9.3 Zugriffsrechte	_____
9.4 Archivdateien	_____
9.4.1 Zweck	_____
9.4.2 Jar-Archive und das <code>jar</code> -Tool	_____
10 Programmierstil	_____
10.1 Wozu Programmierrichtlinien?	_____
10.2 Programmstruktur	_____
10.3 Layout	_____
10.4 Namen	_____
10.5 Groß- und Kleinschreibung von Namen	_____
10.6 Variablen	_____
10.7 Methoden	_____
10.8 Klassen	_____
10.9 Kommentare	_____
11 Dokumentation	_____
11.1 Doc-Kommentare	_____
11.2 Tags	_____

11.3	Generator <code>javadoc</code>	_____
12	I/O (Ein- und Ausgabe)	_____
12.1	Standard-I/O	_____
12.1.1	Problematik	_____
12.1.2	Standard-I/O	_____
12.2	Byteströme	_____
12.2.1	Abstrakte Basisklassen	_____
12.2.2	Methoden zur Ein- und Ausgabe	_____
12.2.3	Beispielprogramm: Standard-I/O kopieren	_____
12.3	I/O-Medien	_____
12.3.1	Medien	_____
12.3.2	File-I/O	_____
12.3.3	Beispielprogramm: Binärdatei kopieren	_____
12.3.4	Block-I/O	_____
12.3.5	Beispielprogramm: Binärdatei blockweise kopieren	_____
12.3.6	Adaptive Blockgrößen	_____
12.4	Filter	_____
12.4.1	Idee	_____
12.4.2	Einordnung in Klassenhierarchie	_____
12.4.3	Beispiele	_____
12.4.4	Gepufferte Ströme	_____
12.4.5	Komprimierte Ströme	_____
12.4.6	Beispielprogramm: Komprimierte Datei lesen und	_____
	<u>erzeugen</u>	
12.4.7	Performance	_____
12.5	Text-I/O	_____
12.5.1	Interne vs. externe Darstellung	_____
12.5.2	Klassen für Text-I/O	_____
12.5.3	Methoden für Text-I/O	_____
12.5.4	Standardmethode <code>toString</code>	_____
12.5.5	Parsermethoden und Wrapperklassen	_____
12.5.6	Brückenklassen	_____
12.6	Package <code>java.io</code>	_____
12.7	Definition neuer I/O-Klassen	_____
12.8	Serialisierung	_____
12.8.1	Ziel	_____
12.8.2	Interface <code>Serializable</code>	_____
12.8.3	Objektströme	_____
12.8.4	Probleme	_____
12.8.5	Beispiel	_____
13	Suchen und Sortieren	_____
13.1	Suchen	_____
13.1.1	Problemstellung	_____
13.1.2	Lineare Suche	_____
13.1.3	Komplexität	_____
13.1.4	Einschätzung der Komplexität	_____
13.1.5	Binäre Suche	_____
13.2	Sortieren	_____
13.2.1	Motivation	_____
13.2.2	Voraussetzung: Vergleichen	_____
13.2.3	Voraussetzung: Vertauschen	_____
13.2.4	Vergleich der Algorithmen	_____
13.3	Einfache Sortieralgorithmen	_____
13.3.1	Bubble Sort	_____

13.3.2	Selection Sort	_____
13.3.3	Insertion Sort	_____
13.3.4	Shellsort	_____
13.3.5	Gegenüberstellung der Ergebnisse	_____
13.3.6	Quicksort	_____
13.4	Beispielprogramme	_____
14	Collection-Framework	_____
14.1	ArrayLists	_____
14.1.1	Problem mit Arrays	_____
14.1.2	ArrayLists vs. Arrays	_____
14.1.3	ArrayListen erzeugen, Element anfügen	_____
14.2	Elementzugriff	_____
14.2.1	Elementtyp Arrays	_____
14.2.2	Elementtyp ArrayList	_____
14.2.3	Wrapperklassen	_____
14.2.4	Umgang mit ArrayLists	_____
	Zugriffsmethode <code>ArrayList.get</code>	_____
	Prüfen des Elementtyps	_____
	Typumwandlung (" <i>type cast</i> ")	_____
14.2.5	ArrayList-Methoden	_____
14.3	Organisation des Collection-Frameworks	_____
14.3.1	Idee	_____
14.3.2	Klassen und Interfaces	_____
14.3.3	Collection-Hierarchie	_____
14.3.4	Allgemeine Methoden	_____
14.3.5	Neue Collections	_____
14.3.6	Ältere Klassen	_____
14.4	Iteratoren	_____
14.4.1	Ziel	_____
14.4.2	Anwendung	_____
14.4.3	Grenzen	_____
14.4.4	Löschen von Elementen	_____
14.4.5	ListIterator	_____
14.5	Maps	_____
14.5.1	Assoziatives Array	_____
14.5.2	Schlüssel und Werte	_____
14.5.3	Maps vs. Collections	_____
14.5.4	Grundlegende Methoden	_____
14.6	HashMap	_____
14.6.1	Einordnung	_____
14.6.2	Beispiel: Telefonbuch	_____
14.6.3	Bestandteile einer Map	_____
14.6.4	Gleichheit von Schlüsseln	_____
14.6.5	Einsatz von HashMaps	_____
14.7	Algorithmen	_____
14.7.1	Idee	_____
14.7.2	Organisation	_____
14.7.3	Beispiele	_____
14.7.4	Größenvergleich	_____
14.7.5	Beispiele	_____
14.8	Gegenüberstellung	_____
14.8.1	Konkrete Containerklassen	_____
14.8.2	Klassen und Interfaces im Collection-Framework	_____
15	Verkettete Listen	_____

- 15.1 Einfach verkettete Listen \_\_\_\_\_
  - 15.1.1 Motivation \_\_\_\_\_
  - 15.1.2 Aufbau \_\_\_\_\_
  - 15.1.3 Bezeichnungen \_\_\_\_\_
  - 15.1.4 Eigenschaften \_\_\_\_\_
- 15.2 Implementierung, 1. Ansatz \_\_\_\_\_
  - 15.2.1 Knotenklasse \_\_\_\_\_
  - 15.2.2 Primitive Zugriffsmethoden \_\_\_\_\_
  - 15.2.3 Probleme \_\_\_\_\_
- 15.3 Implementierung, 2. Ansatz \_\_\_\_\_
  - 15.3.1 Listenoperationen \_\_\_\_\_
  - 15.3.2 Hilfsmethode Vorgängerknoten \_\_\_\_\_
  - 15.3.3 Einfügen \_\_\_\_\_
  - 15.3.4 Löschen \_\_\_\_\_
  - 15.3.5 Probleme \_\_\_\_\_
- 15.4 Implementierung, 3. Ansatz \_\_\_\_\_
  - 15.4.1 Listen- und Knotenklasse \_\_\_\_\_
  - 15.4.2 Aktueller Knoten \_\_\_\_\_
  - 15.4.3 Innere Klassen \_\_\_\_\_
  - 15.4.4 Probleme mit aktuellen Knoten \_\_\_\_\_
- 15.5 Iteratoren \_\_\_\_\_
  - 15.5.1 Idee \_\_\_\_\_
  - 15.5.2 Konsistenzfragen \_\_\_\_\_
- 15.6 Fragen der Semantik \_\_\_\_\_
  - 15.6.1 Ebenen \_\_\_\_\_
  - 15.6.2 Kopieren und Vergleichen \_\_\_\_\_
  - 15.6.3 Cloning \_\_\_\_\_
  - 15.6.4 Schema für `clone()` \_\_\_\_\_
  - 15.6.5 Operationen mit kompletten Listen \_\_\_\_\_
- 15.7 Weitere verkettete Strukturen \_\_\_\_\_
  - 15.7.1 Doppelt verkettete Listen \_\_\_\_\_
  - 15.7.2 Ringe \_\_\_\_\_
- 16 Rekursion \_\_\_\_\_
  - 16.1 Grundlagen \_\_\_\_\_
    - 16.1.1 Rekursion und Iteration \_\_\_\_\_
    - 16.1.2 Beispiel: Zahlensumme \_\_\_\_\_
    - 16.1.3 Beispiel: Fakultätsfunktion \_\_\_\_\_
    - 16.1.4 Beispiel: Fibonacci-Zahlen \_\_\_\_\_
  - 16.2 Programmiersprachliche Umsetzung \_\_\_\_\_
    - 16.2.1 Selbstaufruf \_\_\_\_\_
    - 16.2.2 Aufrufverschachtelung \_\_\_\_\_
    - 16.2.3 Komplexe Aufrufverschachtelung \_\_\_\_\_
  - 16.3 Äquivalenz zwischen Rekursion und Iteration \_\_\_\_\_
    - 16.3.1 Endrekursion (*tail recursion*) \_\_\_\_\_
    - 16.3.2 Schematische Gegenüberstellung \_\_\_\_\_
    - 16.3.3 Anwendung \_\_\_\_\_
    - 16.3.4 Effizienzfrage \_\_\_\_\_
  - 16.4 Beispiel: Türme von Hanoi \_\_\_\_\_
    - 16.4.1 Problemstellung \_\_\_\_\_
    - 16.4.2 Lösungsbeispiel \_\_\_\_\_
    - 16.4.3 Rekursive Lösung \_\_\_\_\_
    - 16.4.4 Implementierung \_\_\_\_\_
    - 16.4.5 Ergebnisse \_\_\_\_\_
  - 16.5 Arten der Rekursion \_\_\_\_\_

16.5.1	Anzahl Aufrufe	_____
16.5.2	Lineare Rekursion	_____
16.5.3	<i>Fat recursion</i>	_____
16.5.4	<i>Compound Recursion</i>	_____
16.6	Rekursive Listenoperationen	_____
16.6.1	Strukturelle Rekursion	_____
16.6.2	Liste ausgeben	_____
16.6.3	Ergebnisrückgabe	_____
16.6.4	Strukturelle Modifikation	_____
16.6.5	Neue Liste als Ergebnis	_____
16.6.6	Implementierung	_____
16.7	<i>Divide-&amp;-Conquer: Quicksort</i>	_____
16.7.1	Begriff	_____
16.7.2	Beispiel: Quicksort	_____
16.7.3	Ablauf von Quicksort	_____
16.7.4	Realisierung	_____
16.7.5	Hilfsfunktion	_____
16.7.6	Verhalten	_____
16.8	<i>Backtracking</i>	_____
16.8.1	n-Damen-Problem	_____
16.8.2	Lösungsidee	_____
16.8.3	Zustand der Berechnung	_____
16.8.4	Implementierung	_____
16.8.5	Hilfsmethode	_____
16.8.6	Ergebnis	_____
16.8.7	Alle Lösungen	_____
Exkurs: Schneeflockenkurve		
1	Idee	_____
2	Ergebnis	_____
3	Bezeichnungen	_____
4	Turtle-Graphik	_____
5	Konfiguration	_____
17	Threads	_____
17.1	Idee	_____
17.1.1	Konzept	_____
17.1.2	Golfplätze	_____
17.1.3	Anwendungen	_____
17.1.4	Threads vs. Prozesse	_____
17.2	Threadklassen	_____
17.2.1	Klasse Thread	_____
17.2.2	Erzeugen von Threadobjekten	_____
17.2.3	Start von Threads	_____
17.2.4	Beenden von Threads	_____
17.2.5	Interface Runnable	_____
17.2.6	Start eines Thread mit Runnable-Objekt	_____
17.2.7	Thread vs. Runnable	_____
17.3	Scheduling	_____
17.3.1	Problem	_____
17.3.2	Thread-Zustände	_____
17.3.3	Zustandswechsel	_____
17.3.4	<i>Time-Slicing</i>	_____
17.3.5	Prioritäten	_____
17.3.6	Selbststeuerung	_____
17.4	Synchronisation	_____

17.4.1	Konkurrierender Zugriff	_____
17.4.2	Problem	_____
17.4.3	Lösungsidee	_____
17.4.4	Objekt-Locks	_____
17.4.5	Synchronized-Methoden und -Blöcke	_____
17.4.6	Konsequenzen	_____
17.5	Kommunikation	_____
17.5.1	Idee	_____
17.5.2	Pipedstreams	_____
17.5.3	Aufbau von Pipedstreams	_____
17.5.4	Beispiel	_____
18	Networking	_____
18.1	Netzwerkgrundlagen	_____
18.1.1	ISO/OSI-Referenzmodell	_____
18.1.2	TCP/IP	_____
18.1.3	IP-Adressen	_____
18.1.4	Domain-Namen	_____
18.2	Sockets	_____
18.2.1	Ports	_____
18.2.2	Client/Server-Modell	_____
18.2.3	Netzwerk-Dienste	_____
18.2.3.1	Echo-Service	_____
18.2.3.2	Telnet-Client	_____
18.2.3.3	Telnet-Server	_____
18.2.3.4	FTP	_____
18.2.3.5	Anonymous Ftp	_____
18.2.3.6	NFS	_____
18.2.3.7	Ping	_____
18.2.4	Sockets	_____
18.2.5	Umgang mit Sockets	_____
18.2.6	Client-Socket	_____
18.2.7	Server-Socket	_____
18.2.8	Mehrfache Requests	_____
18.2.9	Beispielprogramme	_____
18.3	HTTP	_____
18.4	Statische/ dynamische Seiten	_____
18.5	Servlets	_____
18.5.1	Überblick	_____
18.5.2	Lebenszyklus eines Servlets	_____
18.5.3	Beispiel	_____
18.6	Java Server Pages	_____
18.6.1	JSP Überblick	_____
18.6.2	JSP Elemente	_____
18.6.3	Aufbau einer JSP Seite	_____
18.6.4	JavaBean	_____
18.6.5	JavaBean Properties	_____
19	AWT	_____
19.1	GUI-Elemente	_____
19.1.1	Komponenten	_____
19.1.2	Dialogelemente	_____
19.1.3	Container	_____
19.1.4	Ableiten als Konstruktionsmittel	_____
19.2	Plazieren von GUI-Elementen	_____
19.2.1	Layout	_____

19.2.2 BorderLayout für Frames	_____
19.2.3 Layout-Manager	_____
19.3 Dialogereignisse	_____
19.3.1 Events	_____
19.3.2 Low-Level- und semantische Events	_____
19.3.3 Listener und Callbacks	_____
19.3.4 Adapter	_____
19.3.5 Innere Klassen	_____
19.3.6 Anonyme Klasse	_____
19.4 Zeichnen	_____
19.4.1 Graphische Primitive	_____
19.4.2 paint und repaint	_____
19.4.3 Graphik-Kontext	_____
19.4.4 Farben	_____
19.4.5 Schriften	_____
20 Applets	_____
20.1 HTML-Quelltext	_____
20.2 HTTP-Protokoll Webserver	_____
20.3 HTTP-Request	_____
20.4 Webseite mit Applet-Tag	_____
20.5 Statisches Applet	_____
20.6 Lifecycle eines Applets	_____
20.7 Animiertes Applet	_____
20.8 Appletparameter	_____
20.9 Zugriff vom Applet auf Parameter	_____
20.10 Applet vs. Application	_____
20.11 JDK Appletviewer	_____
20.12 Security	_____
Material	_____