Einführung in die Informatik

Jochen Hoenicke

Software Engineering
Albert-Ludwigs-University Freiburg

Sommersemester 2014

Teil I

Organisation

Organisation

- Die Vorlesung ist Mittwochs 14–16 (c.t).
- Die Übungen sind Dienstag 16–18, Mittwoch 16–18, Freitags 12–14
- Die Übungen starten in der nächsten Woche

Anmeldung zu den Übungen

- Übungen können in Gruppen von zwei Personen bearbeitet werden.
- Bitte Gruppe in der Liste eintragen.
- Außerdem muss sich jeder mit seinem Rechenzentrumszugang auf der Daphne-Seite anmelden: `https://daphne.informatik.uni-freiburg.de/ss2014/EinfuehrungInformatikSS2014/`

Die Klausur findet am Freitag, den 5. September, 14 Uhr statt.

Übungsgruppen

In den Übungsgruppen

- sollen Fragen zur Vorlesung geklärt werden,
- Übungsaufgaben/Musterlösung besprochen werden,
- Fragen zum neuen Übungsblatt gestellt werden,
- Probleme mit den Aufgaben besprochen und gelöst werden.

Eine Sprache lernt man nicht durch Theorie; man muss sie benutzen.

Übungen sind nicht verpflichtend, aber ohne sie kann man nicht Programmieren lernen.

Daphne und Subversion

Die Übungen werden über Daphne verwaltet.

- Abgabe geschieht elektronisch über Subversion.
- Programme werden durch Jenkins auf Syntaxfehler geprüft.
- Abgabezeitpunkt wird automatisch überwacht.

Empfohlene Literatur

Die Vorlesung folgt dem Buch:

Dietmar Ratz, Jens Scheffler, Detlef Seese, Jan Wiesenberger, Grundkurs Programmieren in Java, Hanser Verlag, 2011.

`http://www.grundkurs-java.de/`

Teil II

Unser erstes Programm

Unser erstes Programm

Unser erstes Java-Programm soll $3 + 4$ ausrechnen.

```
public class Berechnung {

    public static void main(String[] param) {

        int i;

        i = 3 + 4;

        System.out.println(i);

    }
}
```

Schreiben, Kompilieren, Ausführen

- Das Programm kann mit einem Editor (z.B. notepad++, nano, vim, emacs, eclipse, idea) geschrieben werden.
- Es muss unter dem Namen `Berechnung.java` gespeichert werden.
- Anschließend kann man auf der Konsole (unter Windows startet man dazu cmd) das Programm kompilieren und ausführen:

```
> cd Verzeichnis
> javac Berechnung.java
> java Berechnung
7
```

Achtung

JDK muss installiert sein und der Pfad gesetzt sein (siehe Forum-Link).

Rahmen

Unser eigentliches Programm wird in einen Rahmen gepackt:

```
public class Berechnung {
    public static void main(String[] param) {
        ...
    }
}
```

- Die erste Zeile enthält den Namen des Programms (Berechnung).

Achtung

Der Name muss mit dem Namen der .java-Datei übereinstimmen.

- Die zweite Zeile ist notwendig, damit das Programm ausführbar ist.
- Die genaue Bedeutung dieser Zeilen wird später erklärt. Für das erste, können Sie die Zeilen einfach übernehmen.

Variablen

Mit der Zeile

```
int i;
```

wird ein Speicherbereich für eine Ganzzahl-Variable mit dem Namen `i` reserviert.

- Variablen dienen zum Speichern von Zwischenergebnissen.
- Eine Variable kann unterschiedliche Typen haben: ganze Zahlen (`int`), Kommazahlen (`double`), Zeichenketten (`String`), usw.
- `int` steht für "integral number" (Ganzzahl)
- Die Grösse ist begrenzt: `int` kann nur Zahlen zwischen -2147483648 und 2147483647 speichern.
  Statt `int` kann man auch `long` schreiben für Zahlen zwischen -9223372036854775808 und 9223372036854775807.

Zuweisungen

Mit der Zeile

```
i = 3 + 4;
```

wird $3 + 4$ „berechnet“ und in der Variablen `i` gespeichert.

Achtung

Das Gleichheitszeichen ist in Java nicht symmetrisch. Die Zuweisung

```
3 + 4 = i;
```

führt zu einem Syntaxfehler.
Links muss eine Variable stehen (genauer: ein Name für ein Speicherfeld), rechts muss eine Berechnung stehen.

Ausgabe

Mit der Zeile

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

wird das Ergebnis auf die Konsole ausgegeben.
Man kann auch Zeichenketten ausgeben:

```
    System.out.print("Das Ergebnis ist: ");
    System.out.println(i);
```

oder

```
    System.out.println("Das Ergebnis ist: " + i);
```

gibt die Zeile

```
Das Ergebnis ist: 7
```

aus.

Teil III

Java Syntax

Lexikographische Struktur

Lexikographische Struktur

Ein Java-Programm besteht aus vielen Zeichen.
Mehrere Zeichen bilden zusammen eine Einheit, ähnlich einem Wort in einer natürlichen Sprache. Wir unterscheiden:

- Bezeichner: z.B. `i`, `main`, `param`.
  Damit werden Variablen, Klassen, Methoden, Felder bezeichnet.
- Literale: `3`, `3.14`, `'a'`, `"Hallo Welt"`, `true`, `null`
- Reservierte Wörter: `class`, `int`, `public`, `static`, `void`
- Sonderzeichen und Operatoren: `{`, `}`, `[`, `]`, `(`, `)`, `+`, `=`
  Es gibt auch zusammengesetzte Operatoren: `!=`, `<=`, `++`, `>>>=`
- Kommentare und Leerräume
  Diese haben keine Bedeutung für den Computer.

Bezeichner

Syntax

$$\begin{aligned} \textit{Bezeichner} &::= \textit{JavaLetter}\,\textit{JavaLetterOrDigit}^* \\ \textit{JavaLetter} &::= \texttt{A} \mid \ldots \mid \texttt{Z} \mid \texttt{a} \mid \ldots \mid \texttt{z} \mid \texttt{\_} \mid \texttt{\$} \mid \text{any Unicode letter} \\ \textit{JavaLetterOrDigit} &::= \textit{JavaLetter} \mid 0 \mid \ldots \mid 9 \mid \text{any Unicode digit} \end{aligned}$$

Bezeichner bestehen auf Buchstaben, Ziffern, Unterstrich oder Dollar und dürfen nicht mit einer Ziffer beginnen. Z.B.

- `i`, `Uebung01`, `Uebung_01`
- `Übung01` (gefährlich, da Kodierung kaputt gehen kann)
- `\u00dbbung01` (sichere Variante, aber nicht lesbar)
- `Uebung$01` (aber Dollarzeichen sind für interne Zwecke reserviert)

Keine Bezeichner sind:

- `1Uebung`, `Uebung 1`, `Uebung-1`
- `class`, `int` (reservierte Wörter)

Literale

Literale sind Konstanten wie ganze Zahlen, Zeichen, Zeichenketten

- 0, 123, -3421 (ganze Zahlen)
- `3.14159`, `6.62606957e-34` (Fließkommazahlen)
- `'a'`, `'@'` (Zeichen)
- `"Das Ergebnis ist:"` (Zeichenketten)
- `true`, `false` (Boolesche Konstanten)
- `null` (Null-Objekt)

Mit \ man Zeichen wie ", \ und ' in Zeichenketten ausdrücken:

- `'\''`
- `'\\'`
- `"Ich sagte: \"Hallo\""`

Reservierte Wörter

Folgende Schlüsselwörter sind in Java reserviert:

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

Wenn der Compiler für Ihr Programm merkwürdige Fehlermeldungen liefert, überprüfen Sie, ob Sie eine Variable nach einem Schlüsselwort benannt haben.
Die Schlüsselwörter `const` und `goto` sind zwar reserviert, aber nicht in Java eingebaut.

Kommentare

Es gibt drei Arten von Kommentaren:

- Mit `//` wird der Rest der Zeile als Kommentar interpretiert

```
// ein einzeiliger Kommentar
```

- Alles zwischen `/*` und `*/` wird als Kommentar gelesen. Kommentare können nicht geschachtelt werden.

```
/* Dies ist ein Kommentar.
 * Er kann ueber mehrere Zeilen gehen */
/* /* // */ dies ist kein Kommentar
// dies aber */ und dies
```

- Alles zwischen `/**` und `*/` wird als `javadoc`-Kommentar gelesen.

```
/** Berechnung der Formel 3 + 4. */
public class Berechnung {
  /** Die Hauptmethode zur Berechnung von 3+4.  Sie wird
   *  beim Starten automatisch aufgerufen.
   *  @param params Kommandozeilenargumente (ignoriert).
   */
  public static void main(String[] params) {
```

Grundstruktur eines Programms

Klassen

Die kleinste Einheit eines Java-Programms ist eine Klasse.

```
public class Berechnung {
    ...
}
```

Das Schlüsselwort `public` besagt, dass die Klasse von fremden Programmteilen benutzt werden darf.
Eine als `public` definierte Klasse muss in einer Datei mit dem gleichen Namen und der Endung `.java` gespeichert werden.
Der Kompiler (`javac`) schreibt das Kompilat in eine Datei mit der Endung `.class`. Diese kann dann vom Interpreter (`java`) ausgeführt werden.

Package

Ein größeres Projekt wird gewöhnlich in Paketen organisiert. Es gibt auch Unterpakete, die mit Punkt getrennt werden.

```
package java.util;

public class ...
```

Jedes Paket bekommt sein eigenes Unterverzeichnis, das genau so heißen muss, wie das Paket.

Coding Conventions

Paketnamen sollen nur aus Kleinbuchstaben bestehen. Der Paketname soll mit dem umgekehrten Domainnamen beginnen, damit er weltweit eindeutig ist. Zum Beispiel:

```
package de.uni_freiburg.informatik.ultimate.smtinterpol;
```

Wenn das Programm nicht zur Veröffentlichung gedacht ist, reicht es aber lokal eindeutige Namen zu wählen.

Import

Klassen aus fremden Pakete können in eigenen Programmteilen benutzt werden, indem sie importiert werden. Man kann eine Klasse importieren, oder alle Klassen eines Pakets.

```
package ...;
import Prog1Tools.*;         // importiert alle Klassen
import Prog1Tools.IOTools; // importiert nur IOTools
public class ...
```

Es werden immer alle Klassen aus dem Paket `java.lang` inkludiert, z.B. `System`.

Main-Methode

Programmanweisungen stehen für gewöhnlich in Methoden (auch Prozeduren oder Funktionen genannt). Beim Starten eines Programms wird die `main`-Methode aufgerufen:

```
public class Berechnung {
    public static void main(String[] param) {
        // ...Anweisungen...
    }
}
```

Die Schlüsselwörter `static void` und das Argument `String[] param` werden wir später erklären.

Coding-Conventions

Programme sollen ordentlich eingerückt und umgebrochen werden.

```
public class Berechnung{public static void
main(String[]param){int i;i=3+4;System.out
.println(i);}}
```

Das Programm ist schwer zu lesen und zu warten.

Coding-Convention

- Maximal eine Anweisung pro Zeile
- Einrückung von Code in geschweiften Klammern um vier Zeichen
- Leerzeichen wie im Schreibmaschinenkurs
  Ausnahme: Üblicherweise kein Leerzeichen nach Punkt und zwischen Funktionsname und Parameter, z.B. `System.out.println(i);`

JavaDoc

Programme sollten immer mit JavaDoc-Kommentaren versehen werden: ein Kommentar für die Klasse und ein Kommentar für jede Methode.

```
/** Berechnung der Formel 3 + 4. */
public class Berechnung {
  /** Die Hauptmethode zur Berechnung von 3+4.  Sie wird
   *  beim Starten automatisch aufgerufen.
   *  @param params Kommandozeilenargumente (ignoriert).
   */
  public static void main(String[] params) {
```

Das Tool javadoc erzeugt aus diesen Kommentaren eine HTML-Dokumentation, die für andere Benutzer sehr hilfreich sein kann.
Siehe z.B. `http://docs.oracle.com/javase/7/docs/api/`
Die Kommentare müssen eine gewisse Konvention einhalten, z.B. sollte der erste Satz eine kurze und prägnante Beschreibung sein.

Variablendeklarationen

Variablendeklaration

UNI FREIBURG

Syntax

$$\textit{Deklaration} ::= \textit{Datentyp}\ \textit{Bezeichner}\ (\ ,\ \textit{Bezichner})^*\ ;$$

Eine Variablendeklaration reserviert einen Speicherbereich für ein oder mehrere Variablen. Zum Beispiel reserviert

```
int i, j;
```

Speicher für zwei Ganzzahl(Integer)-Variablen `i` und `j`.
Eine Deklaration wird wie (fast) alle Anweisungen mit einem Semikolon abgeschlossen.

Datentypen

UNI FREIBURG

Syntax

$$\textit{Deklaration} ::= \textit{Datentyp}\ \textit{Bezeichner}\ (\ ,\ \textit{Bezichner})^*\ ;$$

Datentyp kann sein:

- `byte`, `short`, `int`, `long`: Ganzzahlen mit unterschiedlichen Wertebereichen (byte: $-128 \ldots 127$, short: $-32768 \ldots 32767$, int: $-2^{31} \ldots 2^{31} - 1$, long: $-2^{63} \ldots 2^{63} - 1$).
- `char`: einzelnes Unicodezeichen. Dies wird auch wie eine Ganzzahl zwischen 0 und 65535 behandelt, z.B. `'A' == 65`.
- `boolean`: Wahrheitswerte, also `true` und `false`.
- `float`, `double`: Fließkommazahlen mit verschiedenen Wertebereichen und Genauigkeit.
- `String`: Zeichenketten
- ...: Man kann auch seine eigenen Typen bauen (später).

Zuweisungen

Zuweisungen

Syntax

$$\textit{Zuweisung} ::= \textit{LValue} = \textit{Ausdruck} ;$$

$$\textit{LValue} ::= \textit{Bezeicher} \mid \ldots$$

Eine Zuweisung weist einer Variablen (genauer: einem LValue) das Ergebnis einer Berechnung (Ausdruck) zu.
Auch eine Zuweisung wird mit einem Semikolon abgeschlossen.
Die Zuweisungen werden in der Reihenfolge ausgeführt, in der sie im Programm stehen. Es ist keine mathematische Gleichheit.

```
int x;
x = 5;       // x ist jetzt 5
x = 7;       // x ist jetzt 7
x = x + 1; // x ist jetzt 8 (7 + 1)
```

Ausdrücke

Syntax

$$
\begin{aligned}
\textit{Ausdruck} ::= & \ \textit{Ausdruck}\ \textit{Binär-Operator}\ \textit{Ausdruck} \\
| & \ \textit{Unär-Operator}\textit{Ausdruck} \\
| & \ (\textit{Datentyp})\ \textit{Ausdruck} \\
| & \ (\textit{Ausdruck}) \\
\textit{Binär-Operator} ::= & \ \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{==} \mid \texttt{!=} \mid \texttt{>=} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{<} \mid \ldots \\
\textit{Unär-Operator} ::= & \ \texttt{+} \mid \texttt{-} \mid \texttt{!} \mid \texttt{\~{}} \mid \texttt{++} \mid \texttt{--} \mid
\end{aligned}
$$

Ein Ausdruck (engl. Expression) ist eine Berechnung. Die Syntax ist noch größer als oben angegeben.
Man kann Ausdrücke Klammern, ansonsten gilt Punkt- vor Strichrechnung.

Beispiel: Ausdruck

```
int x;
boolean b;
x = 5 + 4 * 3;     // x ist jetzt 17
x = 2 * (x + 1); // x ist jetzt 36
x = -x + 1;        // x ist jetzt -35
b = (x > 0);       // b ist jetzt false
b = (x != 0);      // b ist jetzt true
x = x / 4;         // x ist jetzt -8 (rundet)
x = x % 3;         // x ist jetzt -2 (Rest)
```

Typkonvertierungen

Typkonvertierungen `(Datentyp)Ausdruck` wandeln einen Wert in einen anderen Typ um. Dabei kann Präzision verloren gehen oder ein Überlauf auftreten.
Beispiel:

```
short s; int i; double d;
d = 123456.789
i = (int) d;    // i ist jetzt 123456
s = (short) i; // s ist jetzt -7616
```

Typkonvertierungen passieren auch implizit aber nur, wenn keine Präzision verloren geht:

```
i = s; // okay
d = s; // okay
s = d; // Compiler-Fehler
s = i; // Compiler-Fehler
```

Operationen: Integer

- +, −, ∗: Verhalten sich (außer bei Überlauf) wie man erwarten würde.
- /: Division, rundet zu Null. Division durch Null führt zu einem Programmabsturz.
- %: Rest bei der Division.
- Unäres +,−: Wie man erwartet.
- Unäres ~: vertauscht 0 und 1 in der Binärdarstellung (not).
- &, |, ^: Logisch Und bzw. Oder bzw. Exklusiv-Oder auf der Binärdarstellung.
- <<,>>,>>>: Shiften auf der Binärdarstellung.

Operationen: Fließkommazahlen

Bei Fließkommazahlen rundet die Division nicht und es gibt auch keinen Programmabsturz beim Teilen durch 0.
Aber Vorsicht:

```
double d,e;
d = 7 / 5;      // d ist jetzt 1.0
d = 7.0 / 5.0; // d ist jetzt 1.4
```

Ob / rundet hängt davon ab, ob mindestens ein Operand eine Fließkommazahlen ist.
Integer werden automatisch bei Bedarf nach Fließkommazahlen konvertiert.
Umgekehrt muss man eine explizite Typkonvertierung verwenden.

```
int i;
double d;
i = 7;          // i ist jetzt 7
d = i;          // d ist jetzt 7.0
d = d / 5;      // d ist jetzt 1.4
i = (int) d; // i ist jetzt 1
```

Operator + und Zeichenketten

Der Operator + ist für ganze Zahlen, Fließkommazahlen und Zeichenketten definiert.

1. Ist einer der Operanden eine Zeichenkette, so wird der andere automatisch in eine Zeichenkette konvertiert und die Zeichenketten aneinandergehängt.
2. Ist einer der Operanden eine Fließkommazahl, so werden Fließkommazahlen addiert.
3. Ist einer der Operanden vom Type `long` so wird mit dem Typ `long` addiert.
4. Ansonsten wird mit dem Typ `int` addiert.

Zahlen können nicht mit `(String)i` in Zeichenketten konvertiert werden. Stattdessen kann man die Funktion `String.valueOf(i)` verwenden oder `"" + i`.

Integer, Short, Byte

Die Arithmetischen Operationen gibt es nicht für `byte` und `short`. Es wird implizit auf `int` konvertiert:

```
short s1, s2;
int i;
i = s1 + s2;  // implizite Konvert. nach int
s1 = s1 + s2; // Compiler-Fehler!
s1 = (short) (s1 + s2); // so funktioniert es
```

Die Typen `byte` und `short` sollten nur gebraucht werden, wenn Speicherplatz wichtig ist (z.B. in großen Feldern, später).

Long-Literale

Eine Zahl, z.B. 123456, wird in Java als `int`-Konstante gesehen. Für große Zahlen kann man ein `L` anhängen um auszudrücken, dass es ein `long` sein soll.

```
long l;
l = 1234567890123456789;  // Compilerfehler
l = 1234567890123456789L; // richtig
l = 1 << 40;  // falsch; keine Warnung!
l = 1L << 40; // richtig
```

Der Ausdruck `1L << 40` berechnet 2^{40} (eine Eins mit 40 Nullen in der Binärdarstellung).

Fließkomma-Literale

Eine Fließkommazahl wird als `double` (hohe Genauigkeit) interpretiert, `float` (einfache Genauigkeit) wird mit `F` gekennzeichnet.

```
System.out.println(3.1415926535897932);
// druckt 3.141592653589793
System.out.println(3.1415926535897932F);
// druckt 3.1415927
System.out.println((double) 0.1F);
// druckt 0.10000000149011612}
```

Zeichen (Character)

Der Typ `char` steht für ein Zeichen, ist aber gleichzeitig eine Zahl, die nach `int` konvertiert werden kann.

```
System.out.println('a');                    // druckt a
System.out.println((int)'a');               // druckt 97
System.out.println((char)97);               // druckt a
System.out.println((char)('A' + 1));// druckt B
System.out.println((char)('A' - 1));// druckt @
```

Die Zahlen/Zeichenzuordnung folgt Unicode (ISO 10646).
Weil die Zeichen im lateinischen Alphabet aufeinanderfolgen, kann man leicht den Buchstabenwert bestimmen.

```
char c; int i;
i = c - 'A' + 1; // Buchstabenwert (A=1..Z=26)
```

Vergleichsoperatoren

Die Operatoren < (kleiner),<= (kleiner oder gleich), > (größer), >= (größer oder gleich),== (gleich), != (ungleich) vergleichen zwei Werte und liefern einen Wahrheitswert.

```
boolean b;
b = 3 <= 5; // liefert true
b = 5 > 5;  // liefert false
b = 5 == 5; // liefert true
b = 5 != 5; // liefert false
b = 3.14159 <= Math.PI; // liefert true
b = 3.14159 == Math.PI; // liefert false
```

Vergleichsoperationen

UNI FREIBURG

Achtung

Auf Zeichenketten funktionieren ==, != nicht immer. Auf Fließkommazahlen können durch unterschiedliche Rundungsfehler zwei Zahlen verschieden werden.

```
boolean b; double d;
b = IOTools.readString() == "test"; // falsch
d = 1000000.1;
b = (d - 1000000 == 0.1); // liefert false
```

Operatoren auf Wahrheitswerten

Die Operatoren `&&`, `||`, `!`, `^` stehen für logisches UND, ODER, NICHT und XOR.

```
boolean b;
b = true && true; // liefert true
b = false && b;   // liefert false
b = b || true;    // liefert true
```

Die Operanden werden (wie immer) von links nach rechts ausgewertet. Steht bei `||` oder `&&` das Ergebnis bereits nach dem ersten Operanden fest, wird der zweite *nicht* ausgewertet (Kurzschlussauswertung).

```
b = false && IOTools.readInteger("x: ") == 5;
```

Hier fragt das Programm nicht nach `x`, da das Ergebnis sowieso `false` ist.

Der ?:-Operator

Es gibt einen ternären Operator

$$\textit{Ausdruck} ::= \textit{Ausdruck}\,?\,\textit{Ausdruck}\,:\,\textit{Ausdruck}$$

zum Beispiel

```
int x,y,a,m;
a = x >= 0 ? x : -x // liefert abs(x)
m = x >= y ? x : y  // liefert max(x,y)
```

- Der Operator entspricht dem if-then-else-Operator in funktionalen Programmiersprachen.
- Der erste Ausdruck muss den Typ `boolean` haben, die anderen beiden Ausdrücke müssen den gleichen Typ haben. Wenn der erste Ausdruck zu `true` auswertet, wird der zweite Ausdruck ausgewertet, ansonsten der dritte. Das Ergebnis der Auswertung wird zurückgegeben.
- Es gilt auch hier Kurzschlussauswertung: der Teil der nicht gebraucht wird, wird auch *nicht* ausgewertet.

Zuweisungen mit Operator

Man kann die meisten Operatoren mit einer Zuweisung kombinieren. Die Anweisung

```
i += j;
```

ist eine Abkürzung für

```
i = i + j;
```

Zuweisungen dürfen in Java auch als Ausdrücke (ge/miss)braucht werden.

```
i = (j = 4) + (k = 7);
```

weist `j` den Wert 4, `k` den Wert 7 und `i` den Wert 11 zu.
Das sollte aber nur in Ausnahmefällen verwendet werden.

Pre-/Postinkrement Operatoren

Man kann eine Variable gleichzeitig auslesen und um eins inkrementieren.
Der Ausdruck `i++` (Postinkrement) liest `i` aus, inkrementiert `i`, rechnet aber mit dem alten Wert. Zum Beispiel

```
int i, j;
i = 1;
j = (i++) + (i++) + (i++) + (i++) + (i++);
```

addiert die Zahlen von 1 bis 5.
Der Ausdruck `++i` (Preinkrement) inkrementiert `i`, liest `i` aus und rechnet mit dem neuen Wert. Zum Beispiel

```
i = 1;
j = (++i) + (++i) + (++i) + (++i) + (++i);
```

addiert die Zahlen von 2 bis 6.
Analog dekrementieren `i--` und `--i` den Wert von `i`.
Die Anweisung `i++;` kann als Abkürzung für `i = i + 1;` benutzt werden.

Operator-Präzedenzen

Name	Operator	Assoz.	
Methodenaufruf	`( )`	-	höchste
Komponentenzugriffe	`.`, `[ ]`	links, -	Präzedenz
Postfix-Operatoren	`++`, `--` (Postin-/-dekrement)	-	
Unäre Operatoren	`+`,`-`,`~`,`!`	-	
	`++`, `--` (Prein-/-dekrement)	-	
Typkonvertierung	(*Datentyp*)	-	
Multiplikation	`*`,`/`,`%`	links	
Addition	`+`,`-`	links	
Schiebeoperation	`<<`,`>>`,`>>>`	links	
Relationen	`<=`,`<`,`>=`,`>`, `instanceof`	(links)	
Gleichheit	`==`,`!=`	links	
Bitweises UND	`&`	links	
Bitweises ODER	`|`	links	
Bitweises XOR	`^`	links	
Logisches UND	`&&`	links	
Logisches ODER	`||`	links	
Ternär	`?:`	rechts	
Zuweisung	`=`, `+=`, `-=`, `*=`, `/=`, `%=`,	rechts	niedrigste
	`&=`, `|=`, `^=`, `<<=`, `<<=`, `>>=`, `>>>=`		Präzedenz

Deklaration mit Initialisierung

Eine Deklaration kann mit einer Zuweisung zu einer Anweisung kombiniert werden:

```
int i = 4;
```

reserviert den Speicherbereich für `i` und initialisiert `i` auf 4.
Es geht auch mit mehreren Variablen:

```
int i = 4, j = i + 1, k = j + 1;
```

reserviert den Speicherbereich für `i`,`j` und `k` und initialisiert die Variablen auf 4, 5 bzw. 6.

Methodenaufrufe

Methodenaufrufe

Syntax

Methodenaufruf ::= *Primary* . *Bezeichner* (*Ausdruck* (, *Ausdruck*)*)

Ein Methodenaufruf ist eine Anweisung, oder ein Ausdruck (wenn die Methode einen Wert zurückliefert).
Wir haben bereits Beispiele gesehen:

```
String name = IOTools.readString("Name: "); // 1
System.out.println("Hallo " + name);         // 2
```

In 1 besagt `IOTools`, wo die Methode zu finden ist, `readString` ist der Name der Methode und `"Ihr Name: "` das Argument oder Parameter. Die Methode `readString` gibt ihren Parameter als Prompt aus, liest eine Zeichenkette ein und liefert diese zurück.
Was genau `IOTools` und `System.out` sind, werden wir erst klären können, wenn wir die objektorientierte Programmierung einführen.

IOTools

Die Klasse `IOTools` hält verschiedene Methoden bereit, um Werte einzulesen:

```
int i = IOTools.readInteger();
double j = IOTools.readDouble("j: ");
boolean b = IOTools.readBoolean("b: ");
System.out.println("" + i + j + b);
```

Der Parameter ist optional und ist ein Prompt der ausgegeben wird.
Die Funktionen warten bis eine Eingabe vom Benutzer kommt.

```
25
j: 1e2
b: ja
Eingabefehler java.lang.NumberFormatException: For input strin
Bitte Eingabe wiederholen...
b: true
25100.0true
```

Objekt-Methoden

Man kann auch Methoden auf Objekten, z. B. auf Zeichenketten, aufrufen.
Siehe `http://docs.oracle.com/javase/6/docs/api/java/lang/String.html`.

```
String s = "Hallo Welt";
int i = s.length();                 // i ist 10
s = s.substring(1,4);               // s ist "all"
s = s.replaceAll("l", "ba"); // s ist "ababa"
boolean b = s.equals("ababa"); // true
```

Blöcke

Block

Syntax

$$\textit{Anweisung} ::= \{\ \textit{Anweisung}^*\ \}$$

Mehrere Anweisungen können zu einem Block zusammengefasst werden, indem man sie in { und } einklammert.

```
int j;
{
    int i = 4;
    j = i + 1;
}
{
    int i = j;
    System.out.println(i); // gibt 5 aus.
}
```

Variablen „leben“ nur in dem Block, in dem sie deklariert wurden.

Verzweigungen

If-Anweisung

Bisher wurden unsere Programme immer von vorne nach hinten ausgeführt. Manchmal will man aber in Abhängigkeit von Werten andere Operationen ausführen.

```
int a = readInteger();
int b = readInteger();
if (b == 0) {
  System.err.println("oo");
} else {
  System.err.println(a / b);
}
```

Verschachtelte If-Anweisung

Man kann die if-Anweisung auch verschachteln.

```
int a = readInteger();
int b = readInteger();
if (b == 0) {
  if (a < 0) {
    System.err.println("-oo");
  } else {
    System.err.println("oo");
  }
} else {
  System.err.println(a / b);
}
```

Syntax einer If-Anweisung

UNI FREIBURG

Syntax

If-Anweisung ::= `if (`*Ausdruck*`)` *Anweisung* (`else` *Anweisung*)$^?$

- Der Ausdruck muss vom Typ `boolean` (Wahrheitswert) sein. Wenn der Ausdruck wahr ist, wird die erste Anweisung ausgeführt, ansonsten die zweite. Die zweite Anweisung kann auch weggelassen werden.
- Eine Anweisung kann auch ein Block sein (also geschweifte Klammern). Eine einzelne Anweisung kann auch ohne geschweifte Klammern stehen; dies ist aber gefährlich, da man schnell vergisst die Klammern hinzuzufügen, wenn man die Anweisung erweitert.
- Man sollte zur besseren Lesbarkeit die Unteranweisungen immer in neuen Zeilen schreiben und richtig einrücken.

Switch-Anweisung

Wenn man mehrere Verzweigungen braucht, bietet sich eine Switch-Anweisung an.

```
  int tage = readInteger("Wieviele Tage sind seit dem "
      + "1. Januar 2000 vergangen? ");
  System.out.print("Heute ist ");
  switch (tage % 7) {
  case 0:
    System.out.println("Samstag");
    break;
  case 1:
    System.out.println("Sonntag");
    break;
...
  case 6:
    System.out.println("Freitag");
    break;
  default:
    System.out.println("etwas schiefgelaufen.");
    break;
  }
```

Switch-Anweisung (2)

- Eine Switch-Anweisung hat die Form

```
  switch (Ausdruck) {
(
  case Konstante:
    Anweisung*
    break;
)*
  default:
    Anweisung*
    break;
  }
```

- Switch-Anweisungen funktionieren mit ganzen Zahlen und Aufzählungstypen. Seit Java 7 auch mit Zeichenketten.
- Wenn man das `break` vergisst führt Java den nächsten Fall gleich mit aus.

Schleifen

Schleifen

Manchmal möchte man eine Anweisung mehrmals wiederholen.
Java bietet drei verschiedene Arten von Schleifen an:

- Die While-Schleife.
  Die Laufbedingung wird vor jeder Ausführung des Schleifenrumpfs geprüft.
- Die For-Schleife.
  Variante der While-Schleife bei der normalerweise die Anzahl der Ausführungen im Voraus feststeht.
- Die Do-While-Schleife
  Die Laufbedingung wird nach jeder Ausführung des Schleifenrumpfs geprüft.

Syntax einer While-Schleife

UNI FREIBURG

Syntax

While-Schleife ::= `while` (*Ausdruck*) *Anweisung*

Der Ausdruck ist die Laufbedingung, die vor jedem Schleifendurchlauf geprüft wird. Liefert sie den Wahrheitswert `true`, so wird der Schleifenrumpf (die Anweisung) ausgeführt. Anschließend wird erneut die Laufbedingung geprüft.
Wenn die Laufbedingung `false` liefert beendet die Schleife und die nächste Anweisung nach der Schleife wird ausgeführt.

Die While-Schleife

Eine While-Schleife wird solange ausgeführt, wie die Laufbedingung der Schleife erfüllt ist. Zum Beispiel

```
int x = readInteger("x:"), root = 0;
while (root * root < x)
    root++;
}
System.out.println("Die Wurzel ist " + root);
```

Das Programm berechnet die Quadratwurzel (aufgerundet).

Dreieckszahlen

Wir wollen jetzt die Zahlen von 0 bis $(n-1)$ addieren. (Richtige Informatiker fangen beim Zählen mit 0 an!)
`http://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD831.html`

```
int n = readInteger("n: ");
int sum = 0;
int i = 0;
while (i < n) {
  sum += i;
  i++;
}
System.out.println(sum);
```

Die Variable `i` ist eine Laufvariable, die von 0 bis `n - 1` hochläuft. Bei `i == n` wird die Schleife abgebrochen.

Die For-Schleife

Es kommt recht häufig vor, dass eine Laufvariable von einem zu einem anderen Wert läuft. Java, C und andere Sprachen bieten hierfür eine „schönere“ Syntax an.

```
int n = readInteger("n: ");
int sum = 0;
for (int i = 0; i < n; i++) {
  sum += i;
}
System.out.println(sum);
```

For-Schleife ist While-Schleife

Eine For-Schleife

```
for (A; B; C) {
  D
}
```

ist nur eine Abkürzung für

```
{
  A
  while (B) {
    D
    C
  }
}
```

Normalerweise deklariert und initialisiert `A` eine Variable (die nur lokal in der Schleife lebt), `B` prüft ob diese Variable noch in den Grenzen ist und `C` inkrementiert (oder dekrementiert) die Variable.

Die Do-While-Schleife

Die Do-While-Schleife prüft die Laufbedingung erst nach dem Schleifenrumpf. Der Unterschied ist also, dass die Laufbedingung vor der ersten Ausführung des Schleifenrumpfs noch nicht gelten muss.

```
int n = IOTools.readInteger("n:");
int i = 0;
do {
    System.out.println(i);
    i++;
} while (i < n);
```

Das Programm gibt die Zahlen von 0 bis `n-1` aus. Wenn aber $n = 0$ ist, gibt das Programm trotzdem die Zahl 0 aus.
Oftmals ist es besser und richtiger eine While-Schleife zu verwenden.

Die Break-Anweisung

Manchmal möchte man erst in der Mitte der Schleife die Schleife beenden. Dafür gibt es das Kommando `break`.

```
int sum = 0;
while (true) {
    int n = IOTools.readInteger
        ("Zahl (0 bricht ab): ");
    if (n == 0) {
        break;
    }
    sum += n;
}
```

Das Kommando break verlässt sofort die umschließende Schleife oder Switch-Anweisung ohne die restlichen Kommandos auszuführen.

Die Break-Anweisung mit Label

Wenn man nicht die innerste Schleife verlassen will, kann man das mit einem Label markieren.

```
aussen:
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        if (f(i) == f(j)) {
            System.out.println("f ist nicht
                injektiv");
            break aussen;
        }
    }
}
```

Der Block der mit dem Label markiert wurde, muss keine Schleife sein.

Die Continue-Anweisung

Mit der Anweisung `continue` kann man einen neuen Schleifendurchlauf starten. Es wird dann wieder die Laufbedingung geprüft.

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (j == i)
            continue;
        System.out.println(i
            + " ist ungleich " + j);
    }
}
```

- Bei For-Schleifen wird das Inkrement noch ausgeführt.
- Auch ein `continue` kann mit einem Label versehen werden. Das Label muss aber an einer Schleife stehen.

Teil IV

Methoden in Java

Binomialkoeffizient

In einer Übungsaufgabe sollte der Binomialkoeffizient berechnet werden:

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$$

Beispielprogramm

Möglicher Rechenweg: Berechne $n!$, $k!$, $(n-k)!$ und dividiere.

```
public class Binomial {
    public static void main(String[] param) {
        int n = IOTools.readInteger("n: ");
        int k = IOTools.readInteger("k: ");
        long nf = 1L, kf = 1L, nkf = 1L;
        for (int i = 1; i <= n; i++)
            nf *= i;
        for (int i = 1; i <= k; i++)
            kf *= i;
        for (int i = 1; i <= n-k; i++)
            nkf *= i;
        System.err.println((nf / kf / nkf));
    }
}
```

Codeduplikation

In dem Programm gibt es drei Schleifen, die die Fakultät berechnen.

- Wir müssen dreimal fast den gleichen Code schreiben.
- Wenn wir dabei einen Fehler machen, müssen wir ihn dreimal korrigieren.
- Der Code ist schwieriger zu lesen

Wie können wir die Duplikation der Schleife vermeiden?

Methoden

Man kann gemeinsamen Code in Methoden (auch Funktionen oder Prozeduren genannt) auslagern.

```
public class Binomial {
    public static long fakultaet(int n) {
        long fak = 1L;
        for (int i = 1; i <= n; i++)
            fak *= i;
        return fak;
    }
    public static void main(String[] param) {
        int n = IOTools.readInteger("n: ");
        int k = IOTools.readInteger("k: ");
        System.err.println(
           fakultaet(n) / fakultaet(k) /
              fakultaet(n - k));
    }
}
```

Syntax von Methoden

Die folgende Zeile leitet eine neue Methode ein.

```
public static long fakultaet(int n) {
```

- `public` bedeutet, dass sie von außen aufgerufen werden kann.
- `static` wird später erklärt.
- `long` ist der Typ des Rückgabewertes der Methode
- `fakultaet` ist der Name der Methode. Der kann beliebig gewählt werden.
- `int n` ist der Parameter der Methode

Die Methode nimmmt als Eingabe einen Parameter `n` vom Type `int` und liefert als Rückgabe einen Wert vom Typ `long`.

Methodenkörper

```
public static long fakultaet(int n) {
    long fak = 1L;
    for (int i = 1; i <= n; i++)
        fak *= i;
    return fak;
}
```

- Zwischen den geschweiften Klammern stehen Anweisungen.
- Eine Methode kann mehrfach aufgerufen werden. Die Anweisungen werden jedesmal neu ausgeführt.
- Jede Methode „sieht“ ihre eigenen Variablen. Die Variable `n` der Methode `fakultaet` ist nicht gleich der Variable `n` in `main`.
- Die Parameter werden auf den Wert gesetzt, der beim Aufruf angegeben wird. Z.B. setzt der Aufruf `fakultaet(n-k)` den Wert von `n` in Methode `fakultaet` auf das Ergebnis der Berechnung `n-k`.
- Der Befehl `return` beendet die Methode und setzt den Rückgabewert.

Die Methode `main`

Auch `main` ist eine Methode.

- Der Rückgabetyp ist `void`. Dieser spezielle Typ steht für keinen Wert.
- Eine Methode mit dem Rückgabetyp `void` braucht kein `return`. Man kann aber die `return`-Anweisung (ohne Wert) nutzen, um die Methode vorzeitig zu verlassen.
- Der Parameter ist `String[] param`. Wir werden später sehen, dass das für ein Feld von Zeichenketten (also beliebig viele Zeichenketten) steht.

Ausführungsreihenfolge

Wenn eine Methode aufgerufen wird, wird die aktuelle Methode unterbrochen und später weiter ausgeführt.

```
public class Binomial {
    public static long fakultaet(int n) {
        System.err.println("in fakultaet n = "+n);
        long fak = 1L;
        for (int i = 1; i <= n; i++)
            fak *= i;
        System.err.println("in fakultaet fak = "+fak);
        return fak;
    }
    public static void main(String[] param) {
        int n = IOTools.readInteger("n: ");
        int k = IOTools.readInteger("k: ");
        System.err.println("in main n = " + n);
        long b = fakultaet(n)/fakultaet(k)/fakultaet(n-k);
        System.err.println("in main n = " + n);
        System.err.println("in main b = " + b);
    }
}
```

Parameter und Rückgabewerte

Eine Methode kann (fast) beliebig viele Parameter haben, aber nur maximal einen Rückgabewert.
Zum Beispiel:

```
public static long binomial(int n, int k) {
    return fakultaet(n) / fakultaet(k) /
        fakultaet(n-k);
}
```

Call by Value

Der Parameter wird als Wert übergeben. Wenn Methoden ihre Parameter ändern hat das auf den Aufrufer keine Auswirkung:

```
public static long fakultaet(int n) {
    long fak = 1L;
    while (n > 0)
        fak *= (n--);
    return fak;
}
...
    int  n = 5;
    long f = fakultaet(n);
    System.err.println(n);
```

Hier ändert fakultaet zwar den Parameter `n`, aber das `n` des Aufrufers wird nicht geändert.

Dokumentation

Man sollte zu jeder Methode eine Dokumentation schreiben, die die Methode und ihre Parameter erklärt und die Sonderfälle erläutert.

```
/**
 * Berechnet die Fakult&auml;t von n.
 * @param n  eine nichtnegative ganze Zahl.
 *   Um &Uuml;berl&auml;ufe auszuschlie&szlig;en
 *   muss <code>n &lt;= 20</code> gelten.
 * @returns Die Fakult&auml;t von n.
 */
public static long fakultaet(int n) {
```

Man kann eingeschränkte HTML-Befehle in der Dokumentation benutzen (deshalb muss man aber auch $<$ als `<` schreiben).

Rekursive Methoden

In der Mathematik definiert man Fakultät auch oft induktiv durch

$$0! = 1 \qquad n! = n \cdot (n-1)! \text{ (falls } n > 0)$$

Hier wird bei der zweiten Definition die Fakultät von $n-1$ benutzt um die Fakultät von n zu berechnen (man nennt das Rekursion).
Das geht auch in der Programmiersprache Java:

```
public long fakultaet(int n) {
    if (n == 0)
        return 1;
    return n * fakultaet(n - 1);
}
```

Binomial als Rekursive Methode

Auch mehrfache Aufrufe sind möglich.

$$\binom{n}{k} = \begin{cases} 1 & \text{if } k = 0 \text{ or } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{if } 0 < k < n \end{cases}$$

```
public long binomial(int n, int k) {
    if (k == 0 || k == n)
        return 1;
    return binomial(n-1, k-1)
         + binomial(n-1, k);
}
```

Teil V

Referenzdatentypen

Felder (Arrays)

Felddatentypen

Die bisher vorgestellten Datentypen können nur einen Wert speichern (Ausnahme: Zeichenkette). Um mehrere Werte zu speichen gibt es Feldtypen (array types).

Feldtyp

Zu jedem Datentyp `T` gibt es einen Datentyp mit dem Namen `T[]`, der mehrere Elemente vom Typ T speichern kann. `T[]` ist ein Feldtyp.

Den Typ `String[]` haben wir schon als Parameter der `main`-Funktion gesehen. Der Parameter enthält die Kommandozeilenargumente.

Lesen von Feldern

Das folgende Programm gibt die Kommandozeilenargumente aus:

```
public class PrintArgs {
    public static void main(String[] params) {
        for (int i = 0; i < params.length; i++) {
            System.out.println("Argument "+i+"
               ist "+params[i]);
        }
    }
}
```

```
> java PrintArgs Hallo Welt
Argument 0 ist Hallo
Argument 1 ist Welt
```

Lesen von Feldern (2)

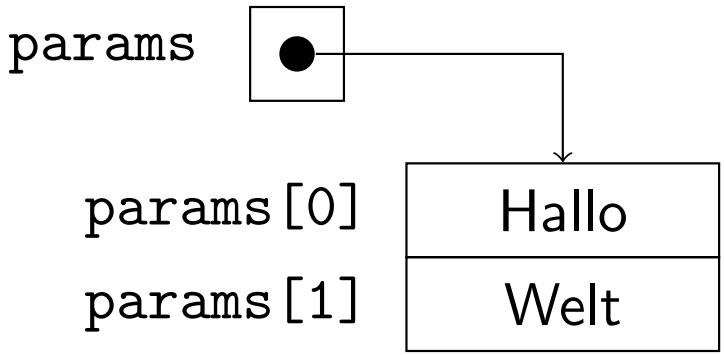
```
for (int i = 0; i < params.length; i++) {
    System.out.println("Argument " + i
                       + " ist " + params[i]);
}
```

Sei `params` eine Variable, die einen Feldtyp hat.

- `params.length` gibt die Länge des Felds (also die Anzahl der gespeicherten Werte) zurück.
- `params[i]` gibt das `i`-te Element des Feldes zurück. Man nennt `i` auch den Feldindex.
  Die Felder werden von 0 bis `params.length - 1` nummeriert.

Felder sind Referenztypen

Die Variable `params` enthält in Wirklichkeit nur einen Zeiger (auch Referenz genannt) auf das Feld. Das Feld selbst kann irgendwo im Speicher liegen.



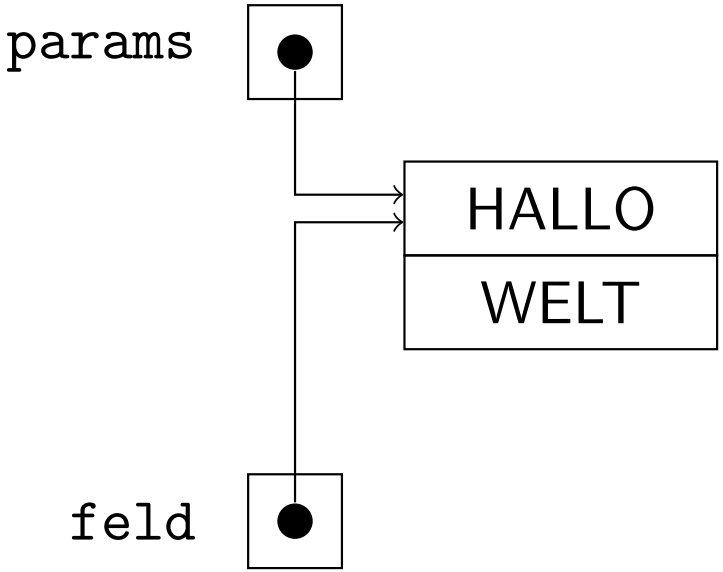
Bei einem Methodenaufruf wird nur der Zeiger übergeben, das Feld selbst wird nicht kopiert.

Schreiben auf Felder

```
public class PrintArgs {
    public static void macheGross(String[] feld){
        for (int i = 0; i < feld.length; i++)
            feld[i] = feld[i].toUpperCase();
    }
    public static void main(String[] params) {
        macheGross(params);
        for (int i = 0; i < params.length; i++)
            System.out.println(params[i]);
    }
}
```

Der Ausdruck `feld[i]` darf auch auf der linken Seite einer Zuweisung stehen. Dadurch wird der Inhalt des Feldes geändert.

Ablauf des Programms



- Beim Aufruf von `macheGross` wird der Zeiger von `params` nach `feld` kopiert.
- Durch `feld[0] = feld[0].toUpperCase()` wird der Inhalt des Feldes geändert.
- Nach dem Ende der Methode gibt es `feld` nicht mehr. Die Parameter sind aber geändert.

Erzeugen von Feldern

Felder müssen erzeugt werden und dabei wird auch die Länge des Feldes festgelegt:

```
int[] meinFeld = new int[10];
```

Der Ausdruck `new int[10]` erzeugt ein Feld mit exakt 10 Elementen, die jeweils ein `int` speichern können. Der Zeiger auf dieses Feld wird der Variable `meinFeld` zugewiesen.

- Die Länge des Feldes kann nicht geändert werden, allerdings kan man `meinFeld` ein neues Feld mit zum Beispiel mehr Elementen zuweisen.
- Wenn man ein Element lesen will, dass nicht vorhanden ist (zu großer oder negativer Feldindex), stürzt das Programm mit einer Fehlermeldung ab (ArrayIndexOutOfBoundsException).
- Initial werden alle Elemente auf 0 initialisiert. Referenztypen werden auf den Wert `null` initialisiert.

Nullreferenz

Alle Referenztypen (also auch Felder) können in Java den Wert `null` haben. Dieser steht für uninitialisiert.

```
int[] meinFeld = null;
```

- Der Zeiger `null` zeigt auf einen verbotenen Speicherbereich an dem kein Feld liegt.
- Jeder Zugriff auf einen Index oder auf die Länge des Feldes, das den Wert `null` hat, führt zu einem Programmabsturz (NullPointerException).

Syntax für Feldzugriffe

Syntax

$$\textit{Ausdruck} ::= \textit{Ausdruck}\,[\textit{Ausdruck}]$$

$$\textit{LValue} ::= \textit{Ausdruck}\,[\textit{Ausdruck}]$$

Wenn `feld` ein Ausdruck vom Typ `T[]` und `i` ein Ausdruck vom Typ `int` ist, dann ist `feld[i]` sowohl ein *Ausdruck*, als auch ein *LValue* vom Typ `T`.

- In beiden Fällen werden zunächst `feld` und `i` ausgewertet. Das liefert einen Zeiger auf ein Feld und eine Zahl (den Feldindex).
- Als Ausdruck (Lesezugriff) hat `feld[i]` den Wert des `i`-ten Eintrags in diesem Feld.
- Als LValue (links von einer Zuweisung) wird dem `i`-ten Eintrag des Felds der Wert auf der rechten Seite zugewiesen.
- Wenn `feld` zu `null` auswertet, bricht in beiden Fällen das Programm mit einem Fehler ab (NullPointerException). Ebenso, wenn es keinen `i`-ten Eintrag gibt (ArrayIndexOutOfBoundsException).

Felder und Schleifen

Oft möchte man Code für alle Einträge in einem Feld ausführen, z.B.

```
for (int i = 0; i < params.length; i++)
    System.out.println(params[i]);
```

Java hat für diesen Fall noch eine zweite Syntax für die For-Schleife:

```
for (String param : params)
    System.out.println(param);
```

Hier wird eine Variable `param` deklariert und dieser nach und nach jeder Eintrag des Feldes `params` zugewiesen. Jedesmal wird dann der Schleifenkörper ausgeführt.

Allerdings gibt es *keine* Abkürzung für

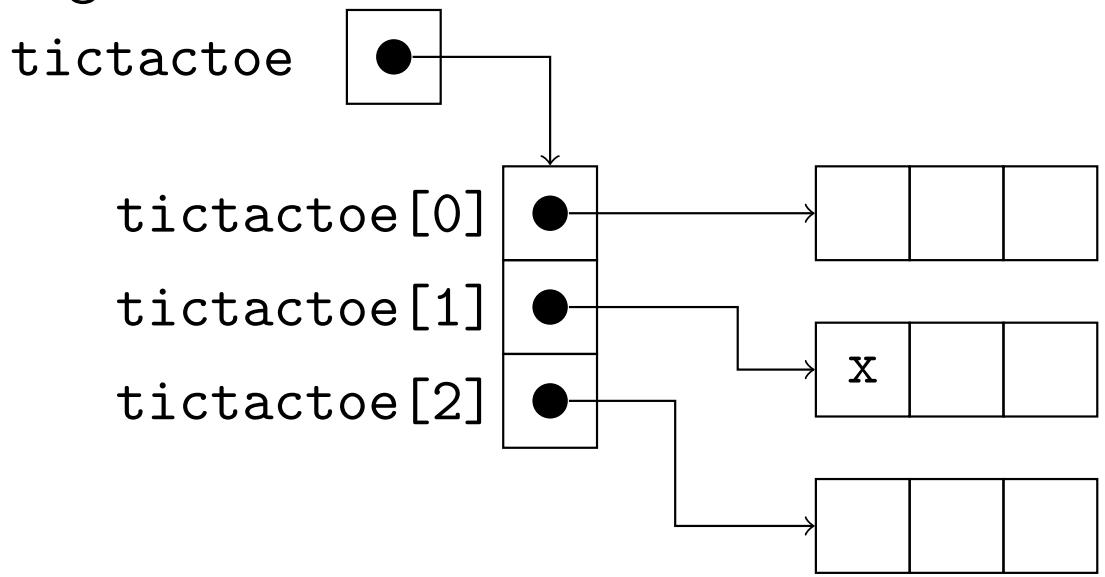
```
for (int i = 0; i < feld.length; i++)
    feld[i] = feld[i].toUpperCase();
```

Mehrdimensionale Felder

Man kann Feldtypen auch schachteln, z.B.

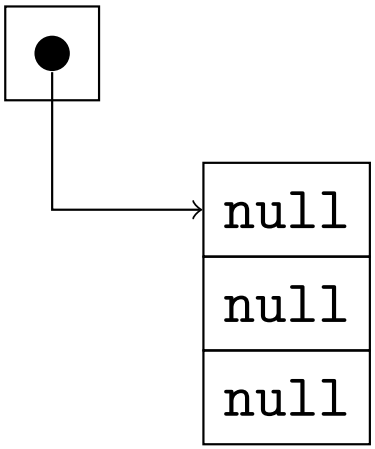
```
char[][] tictactoe = new char[3][3];
tictactoe[1][0] = 'x';
```

Hierbei ist `char[][]` ein Feld, das Werte vom Typ `char[]` speichert, also Zeiger auf `char`-Felder.



Erzeugen mehrdimensionaler Felder (1)

Der Ausdruck `new char[3][]` erzeugt ein Array mit drei Elementen vom Typ `char[]`, die mit `null` gefüllt werden:

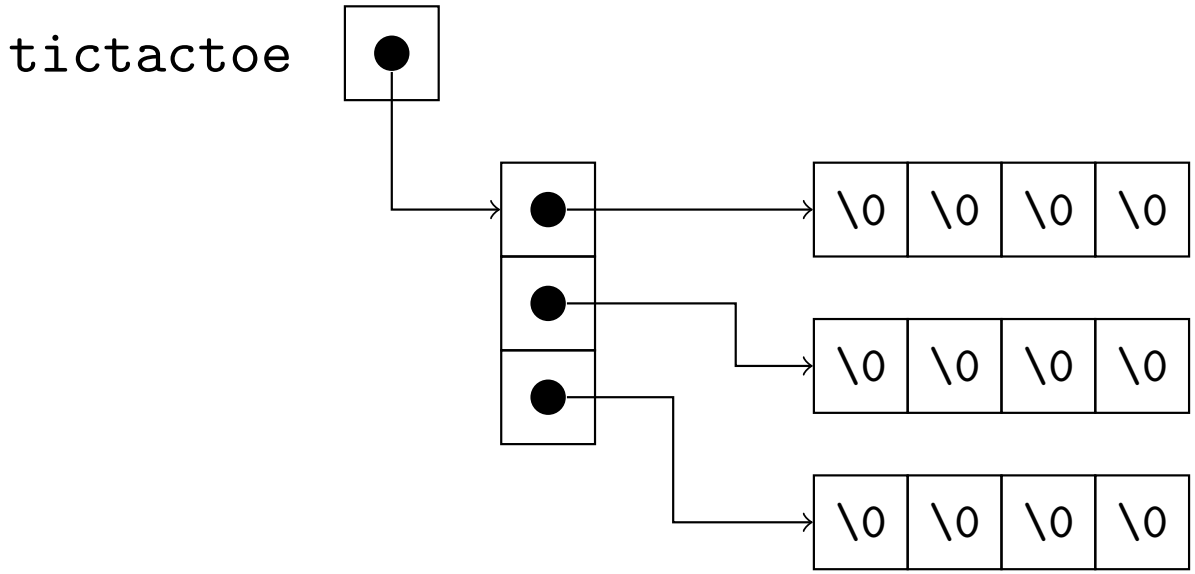


Achtung

Logisch wäre zwar `new char[][3]`, denn `char[]` ist der Typ der Argumente. Das ist aber syntaktisch falsch. Bei `new` entspricht die Reihenfolge der eckigen Klammern der Reihenfolge beim Feldzugriff.

Erzeugen mehrdimensionaler Felder (2)

Der Ausdruck `new char[3][4]` erzeugt ein Array mit drei Elementen vom Typ `char[]`, die mit neuen Arrays mit jeweils vier Elementen vom Typ `char` initialisiert werden.

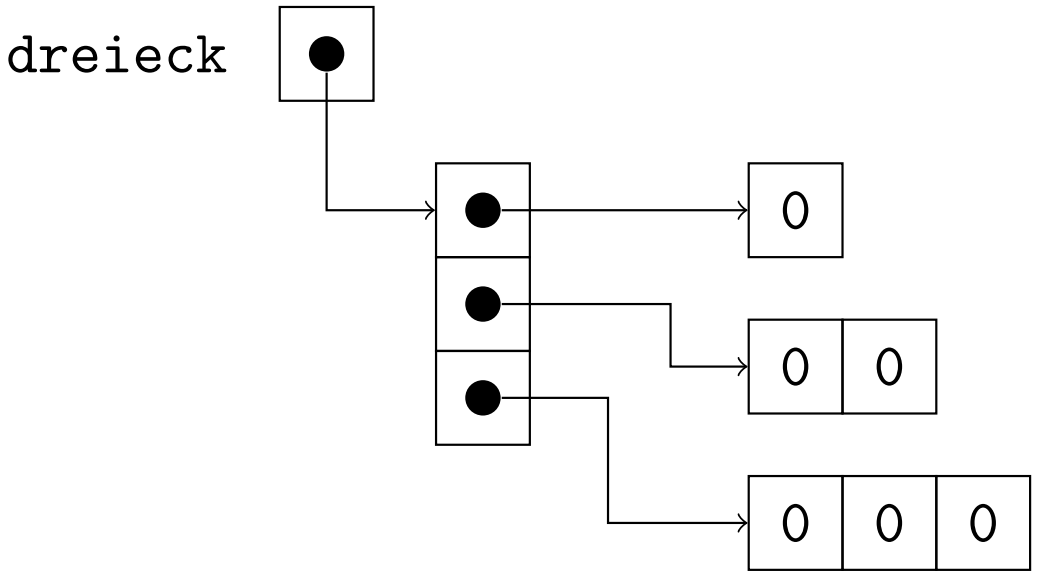


Der Ausdruck `new char[][4]` ist syntaktisch nicht erlaubt.

Erzeugen mehrdimensionaler Felder (3)

Bei mehrdimensionalen Feldern, gibt es aber keinen Grund, warum die Unterfelder gleich lang sein müsssen.

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



Klassen

Klassen

Ein weiterer Referenztyp sind Klassen. Damit kann man mehrere Werte mit unterschiedlichem Typ zusammenfassen.

```
public class Adresse {
    public String vorname;
    public String nachname;
    public String strasse;
    public int hausnummer;
    public int postleitzahl;
    public String ort;
}
```

- Der obige Java-Code deklariert einen neue Klasse (einen Typ) mit dem Namen `Adresse`, die aus mehreren Komponenten `vorname`, `nachname`, usw. besteht.
- Eine Instanz oder Objekt von diesem Typ speichert mehrere Werte, einen für jede Komponente.
- Das Schlüsselwort `public` erlaubt es jedem, die Komponente zu lesen und zu schreiben.

Wohin kann man Klassen schreiben?

- Die Klasse Adresse kann in einer Datei mit dem Namen `Adresse.java` gespeichert werden.
- Alternativ kann man auch innere Klassen verschachtelt in einer Klasse definieren:

```
public class Adressverwaltung {
    public static class Adresse {
        ...
    }

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

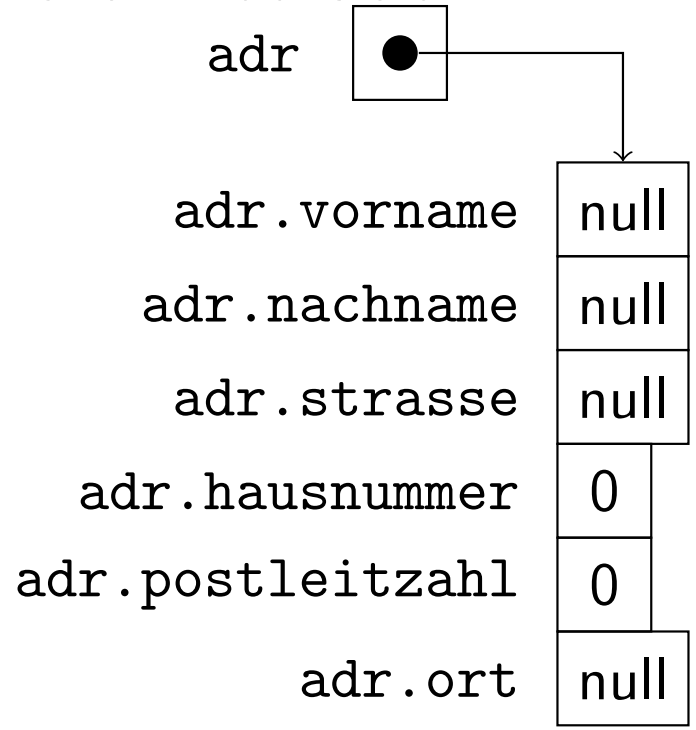
Das sollte man aber nur machen, wenn die Klasse nur von der Adressverwaltung gebraucht wird.

Objekt erzeugen

Die folgende Anweisung erzeugt ein neues Objekt vom Typ Adresse und weist sie der Variablen adr zu.

```
Adresse adr = new Adresse();
```

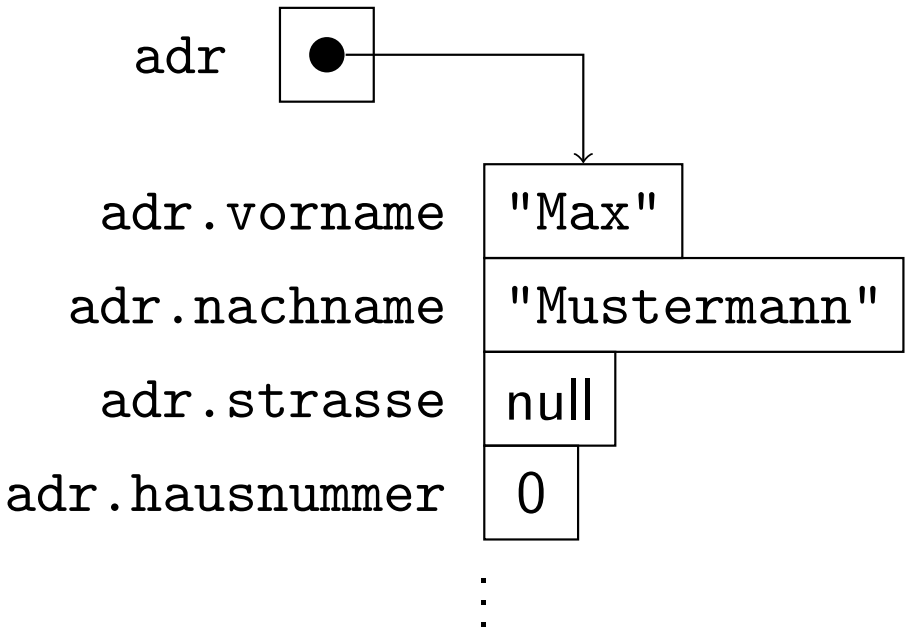
Auch hier wir in adr eine Referenz auf das eigentliche Objekt geschrieben. Das Objekt selbst kann irgendwo im Speicher liegen. Das Objekt wird mit Nullen initialisiert.



Objekte schreiben

Mit der Punkt-Notation kann auf die Komponenten eines Objekts zugegriffen werden.

```
Adresse adr = new Adresse();
adr.vorname = "Max";
adr.nachname = "Mustermann";
System.out.println(adr.vorname + " " +
                   adr.nachname);
```



Syntax der Punktnotation

Syntax

$$\textit{Ausdruck} ::= \textit{Ausdruck} \,.\, \textit{Bezeichner}$$

$$\textit{LValue} ::= \textit{Ausdruck} \,.\, \textit{Bezeichner}$$

Wenn `adr` ein Ausdruck vom Typ `Adresse` ist und `vorname` ein Komponente vom Typ `Adresse`, dann ist `adr.vorname` sowohl ein *Ausdruck*, als auch ein *LValue*.

- In beiden Fällen wird zunächst `adr` ausgewertet. Das liefert einen Zeiger auf ein Objekt.
- Als Ausdruck (Lesezugriff) hat `adr.vorname` den Wert der Komponente `vorname` in diesem Objekt.
- Als LValue (links von einer Zuweisung) wird dem Komponente `vorname` des Objekts der Wert auf der rechten Seite zugewiesen.
- Wenn `adr` zu `null` auswertet, bricht in beiden Fällen das Programm mit einem Fehler ab (NullPointerException).

Felder und Klassen

Man kann Felder und Klassen auf verschiedene Weise kombinieren.

- Man kann ein Feld von Objekten erzeugen:

```
Adresse[] adressListe = new Adresse[10];
adressListe[0] = new Adresse();
```

- man kann in einer Klasse Felder oder andere Klassen als Komponenten haben.

```
public class Brief {
    public String[] anhaenge;
    public Adresse absender;
    public Adresse empfaenger;
    ...
}
```

Teil VI

Objektorientierte Programmierung

Programmierparadigmen

Im Laufe des letzten Jahrhunderts wurden verschiedenen Programmierparadigmen vorgeschlagen:

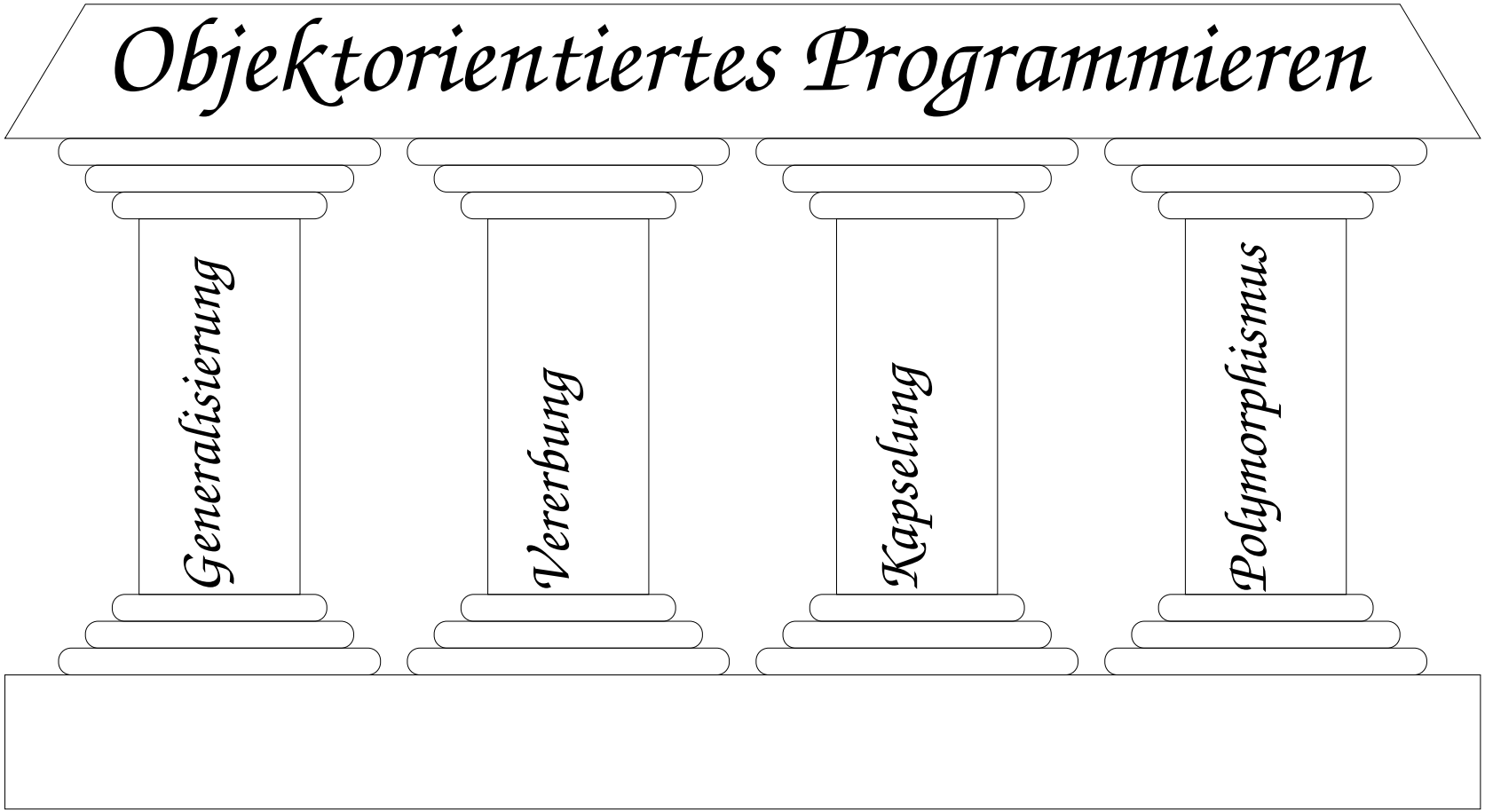
- *Imperative Programmierung*: Hintereinander Ausführung von Befehlen, Schleifen. Zur Strukturierung werden Teile in eigene Methoden ausgelagert.
- *Logische Programmierung*: Das Problem wird durch logische Formeln beschrieben. Die Programmiersprache enthält Automatismen zum Lösen der Formeln mit Lösungssuche.
- *Funktionale Programmierung*: Das Programm wird in Funktionen geschrieben. Diese sollten nach Möglichkeit wenig oder keine Seiteneffekte haben. Es gibt keine Variablen.
- *Objektorientierte Programmierung*: Das Programm wird in Klassen unterteilt. Die Objekte dienen nicht nur als Datenspeicher, sondern haben eigene „Intelligenz“.

Philosophie der Objektorientierung

Everything is an object — *Alan Kay, Erfinder von Smalltalk*

- Objekte speichern die Daten (in Form von Komponenten).
- Objekte enthalten Programmcode (in Form von Methoden).
- Objekte kommunizieren miteinander (in Form von Methodenaufrufen).
- Objekte sind Instanzen von Klassen.

Die Grundpfeiler der Objektorientierten Programmierung



Generalisierung / Abstraktion

Objekte werden in Klassen kategorisiert.

Beispiel: Dateien

Auf einem Rechner liegen viele Dateien. Auch wenn sie alle verschieden sind, haben sie Gemeinsamkeiten.
Dateien enthalten eine Folge von Zeichen. Sie können gelesen oder geschrieben werden.
Die Dateien sind Objekte, die zur selben Klasse (Datei) gehören.

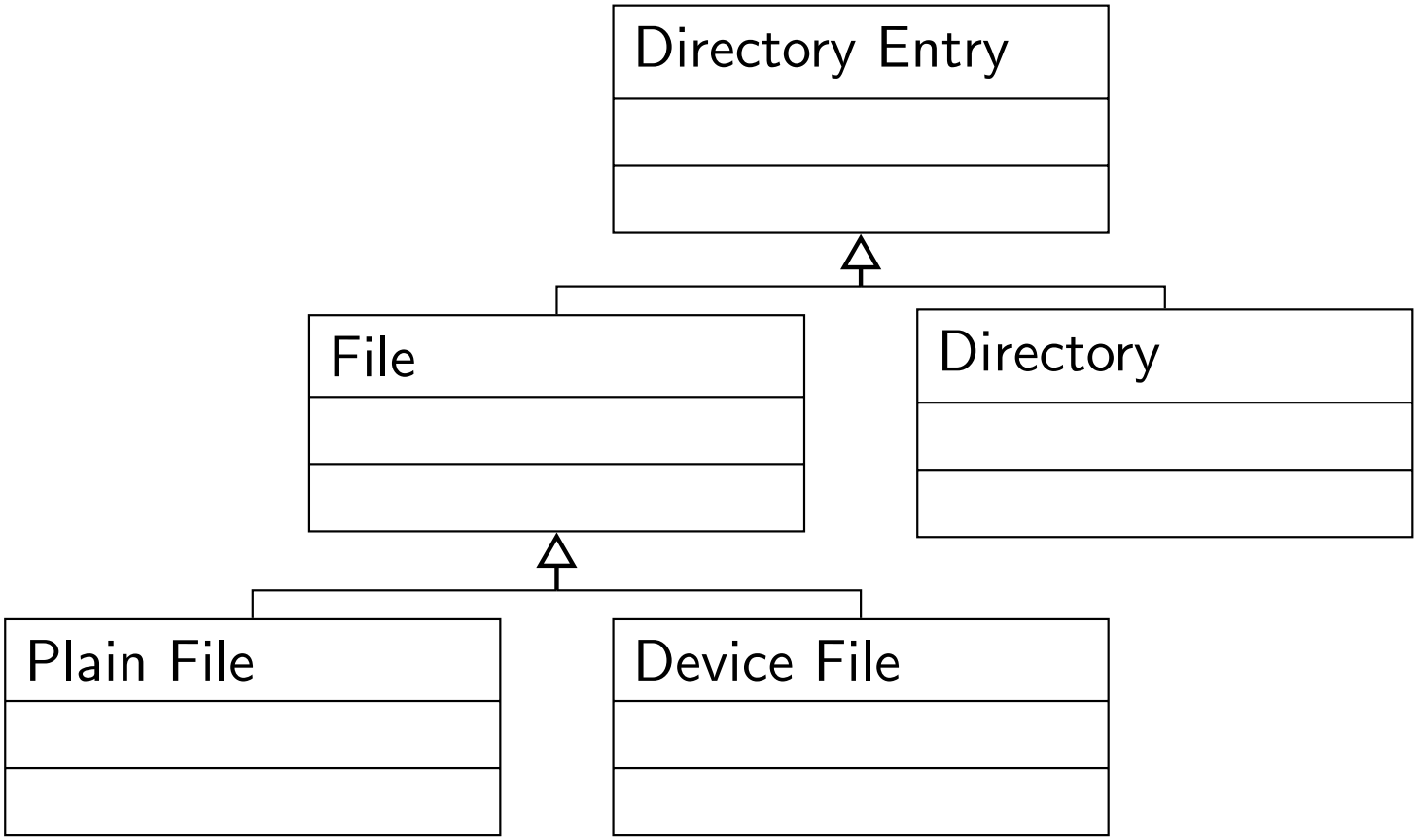
Kapselung

Kapselung verbietet anderen Programmteilen Zugriff auf das Innenleben

- Wohldefinierte Schnittstelle erleichtert die Zusammenarbeit in großen Programmierprojekten.
- Innenleben kann später geändert werden ohne die anderen Programmteile ändern zu müssen.
- Mehrere Objekte können die gleiche Schnittstelle aber verschiedenes Innenleben haben.

Vererbung

Klassen können hierarchisch geordenet werden. Subklassen erben Attribute von Superklassen.



Polymorphismus

Polymorphie (wörtlich: Vielgestalt) erlaubt es verschiedenen Objekten, die gleiche Schnittstelle aber verschiedenes Verhalten zu haben.

- eine gemeinsame Superklasse definiert die Schnittstelle in Form einer Methode. Sie kann auch ein Standardverhalten vorgeben.
- Die Subklassen können die Methode überschreiben (Englisch: override), um ihr eigenes Verhalten zu definieren.

Klassen

Die Objekte werden in Klassen kategorisiert.

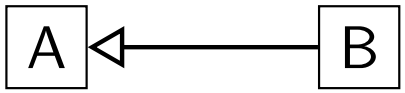
- Eine Klasse definiert die Daten (Komponenten), aus denen die Objekte zusammengesetzt sind.
- Eine Klasse definiert auch Methoden, um diese Daten zu manipulieren (die Schnittstelle der Objekte).
- Alle Objekte einer Klasse haben die gleichen Komponenten und die gleichen Methoden, aber verschiedene Datenwerte.
- Die Daten und Methoden können öffentlich (public) oder privat (private) sein. Die interne Struktur soll privat sein, um eine Kapselung zu erreichen.

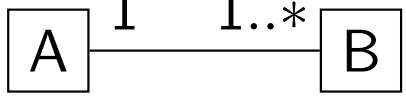
Klassendiagramme

Um Objekte in Klassen zu kategorisieren und Abhängigkeiten zu visualisieren bieten sich Klassendiagramme aus der UML an.

- Eine Klasse wird dargestellt durch eine Box

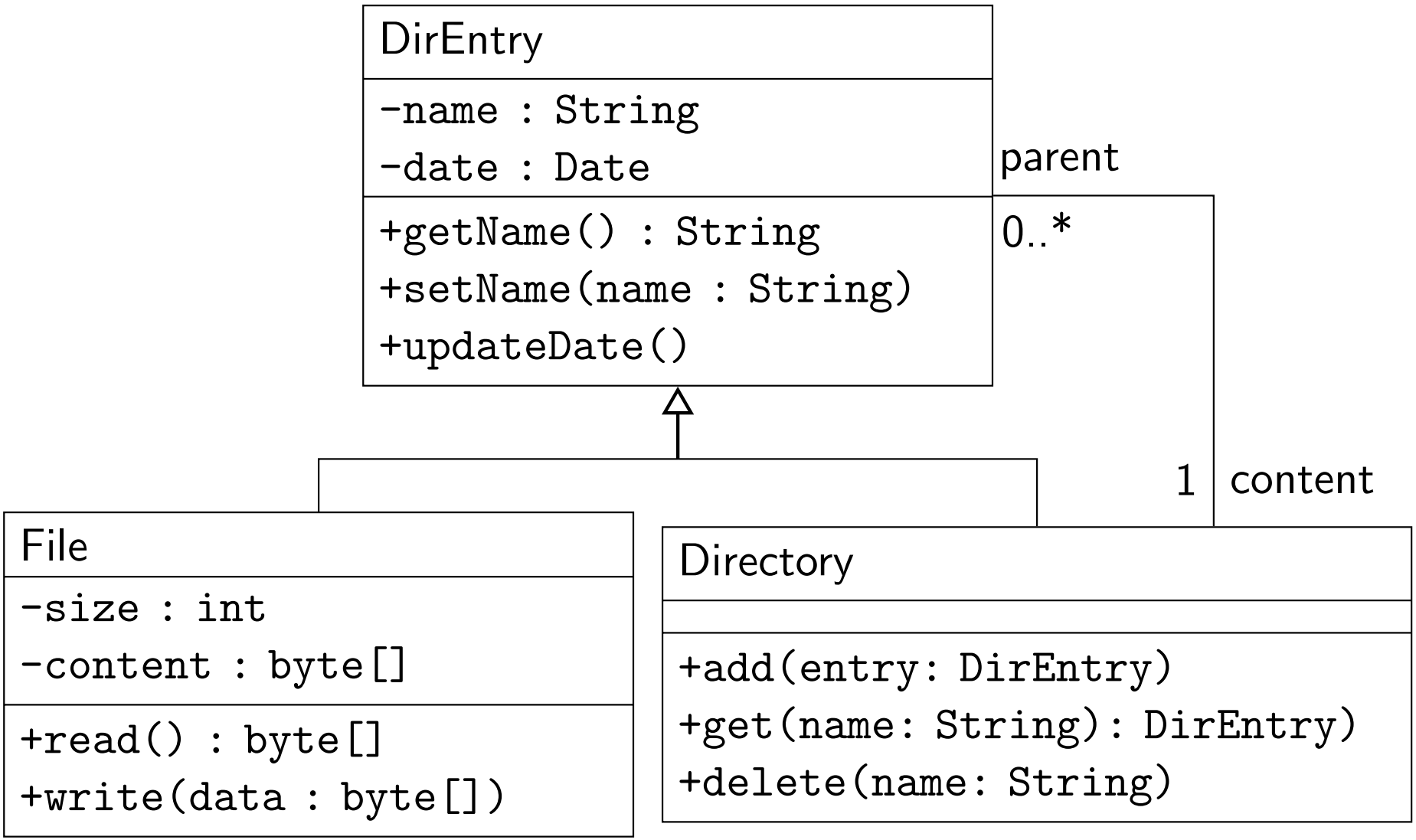
KlassenName
`-komponente: int`
`-interneMethode() : int` `+methode(param : int): int`

- Mit dem Pfeil 

wird eine Vererbung angezeigt. Klasse B erbt von Klasse A.
- Mit einer Kante 

wird eine Relation zwischen Klassen angezeigt. Einem A-Objekt ist mindestens ein B-Objekte zugeordnet; einem B-Objekt ist genau ein A-Objekt zugeordenet.

Beispiel: Klassendiagramm



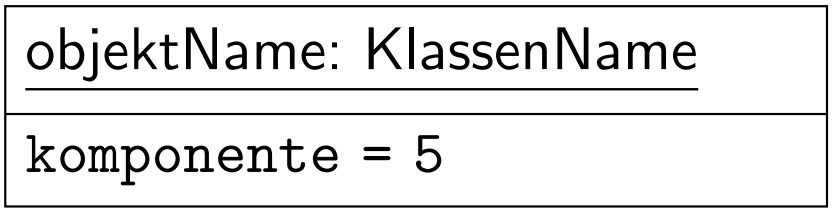
Objektdiagramme

Zu einer Klasse kann es viele Objekte geben.

Beispiel

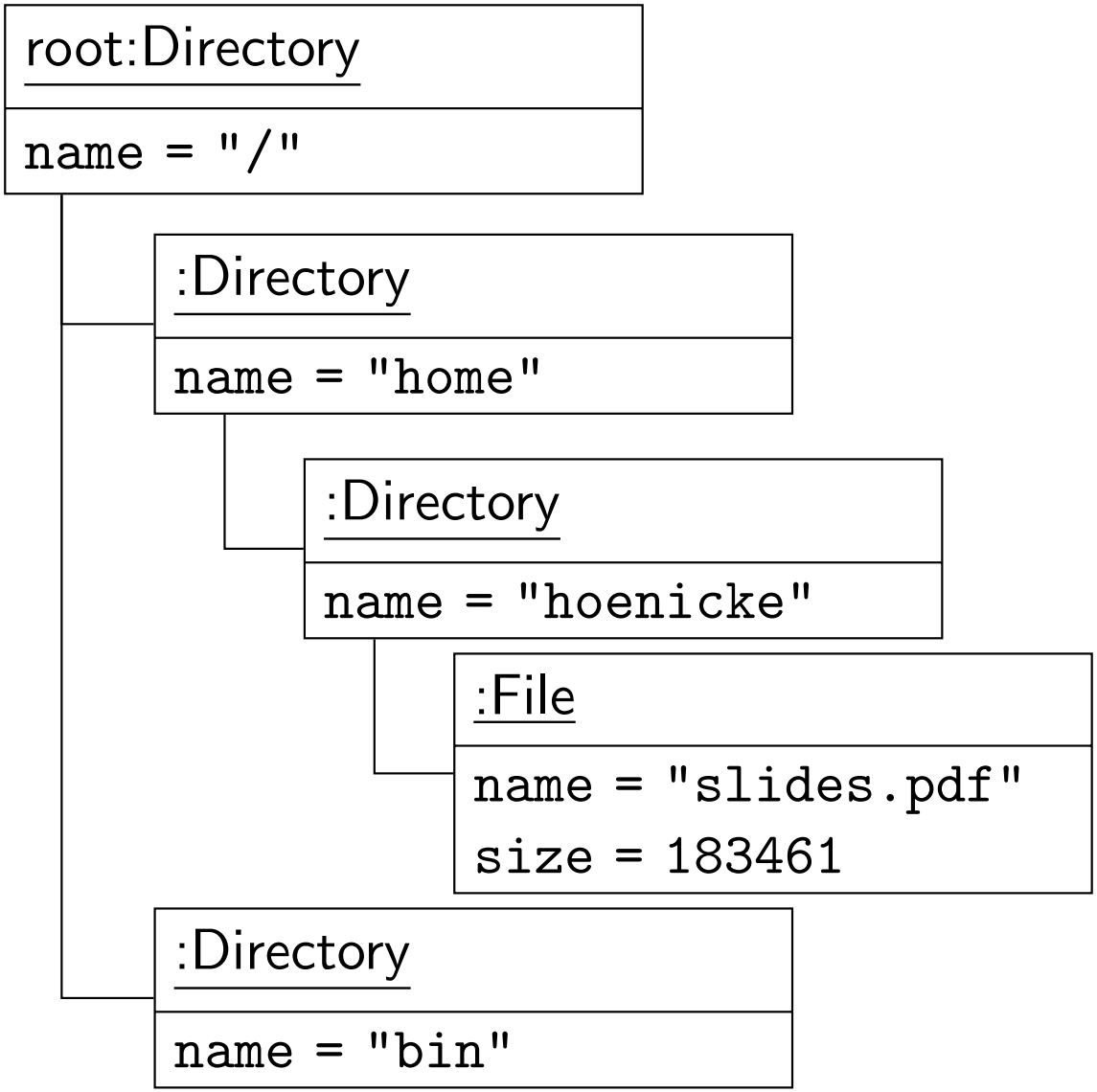
Die Klasse `File` hat viele Objekte, nämlich eins für jede Datei.

Objekte werden durch einen Doppelpunkt und einem Unterstrich gekennzeichnet:



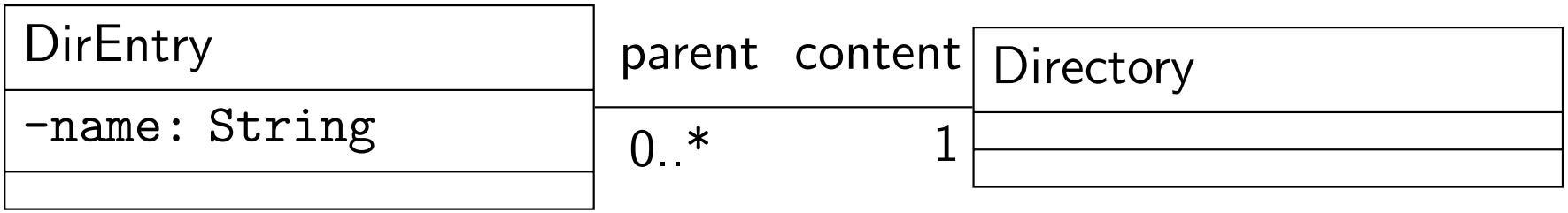
Im Gegensatz zu Klassen müssen Objekte nicht immer einen Namen haben.

Beispiel: Objektdiagramm



Relationen

Relationen lassen sich in Java nicht direkt ausdrücken.



Man kann Sie durch Attribute (Komponenten) modellieren, zum Beispiel:

Allerdings müsste man noch dokumentieren, dass für ein Directory `dir` gilt: `dir.content[i].parent == dir`.
Manchmal braucht man auch die eine Richtung der Relation nicht und kann sie weglassen.

Klassen in Java

Java-Klassen

```
public class ClassName {
    private String component1;
    private int component2;

    private String internalMethod() {
        ...
    }
    public int publicMethod() {
        ...
    }
}
```

- Achtung: kein `static` benutzen.
- Mit den Schlüsselwörtern `private` oder `public` wird angezeigt, ob die Komponente bzw. Methode privat oder öffentlich ist.

Instanzmethoden

Eine Instanzmethode ist eine Methode, die nicht mit `static` gekennzeichnet ist.

```
public class DirEntry {
    private String name;

    public void setName(String n) {
        this.name = n;
        // alternativ:  name = n;
    }
}
```

- Eine Instanzmethode ist in einer Klasse definiert,
- gehört aber zu einem Objekt.
- Mit `this` kann die Methode auf dieses Objekt zugreifen.
- Wenn es eindeutig ist, kann `this.` auch weggelassen werden.

Aufruf von Instanzmethoden

Auf die Instanzmethode eines Objektes wird ebenfalls mit der Punktnotation zugegriffen.

```
DirEntry entry = someEntry;
entry.setName("home");
```

Hier ist `entry.setName` die Instanzmethode `setName` des Objektes `entry`.

Wenn man aus eine Instanzmethode von einer anderen des gleichen Objekts aufrufen will, kann man `this` benutzen:

```
this.setName("abc");
```

Man kann `this.` auch weglassen.

Beispiele für Methodenaufrufe

Wir haben diese Notation bereits kennengelernt, ohne darauf näher einzugehen:

- `System.out.println("Hello")` ruft die Methode `println` des Objektes `System.out` auf. Letzteres Objekt entspricht der Standardausgabe (normalerweise Konsolenausgabe).
- `"abc".toUpperCase()` ruft die Methode `toUpperCase` einer Zeichenkette auf. In der Tat ist auch `String` eine Klasse und alle Zeichenketten sind Objekte.

Validierung durch Instanzmethoden

Instanzmethoden können benutzt werden, um Felder zu setzen und dabei auf gültige Werte zu validieren:

```
/**
 * Set the file name.
 * @param n the new filen name.
 * @returns true if successful,
 *          false otherwise.
 */
public boolean setName(n : String) {
    // Do not allow '/' in a file name.
    if (n.indexOf('/') != -1)
        return false;
    this.name = n;
    return true;
}
```

Rekursive Instanzmethoden

Auch Instanzmethoden können rekursiv sein.

```
public class DirEntry {
    String name;
    DirEntry parent;
    public String getAbsolutePath() {
        if (parent == null)
            return "/";
        return parent.getAbsolutePath()
                + "/" + name;
    }
}
```

Zwei Sichten auf Instanzmethoden

Es gibt zwei Sichtweisen:

1. Jedes Objekt einer Klasse hat seine eigene Instanzmethode. Diese unterscheiden sich dadurch, dass `this` ein anderes Objekt bezeichnet.
2. Es gibt nur eine Methode, nämlich die der Klasse. Das Objekt `this` ist nur ein versteckter Parameter der Methode. Beim Methodenaufruf schreibt man diesen Parameter vor den Methodennamen statt in die Klammern.

Ohne Polymorphie (später) gibt es keine Möglichkeit, diese Sichtweisen zu unterscheiden. Man kann sich aussuchen, welche man intuitiver findet.

Statische Komponenten und Methoden

statische Komponenten und Methoden

Was bedeutet das Schlüsselwort `static`?

```
public class Test {
    public static int wert;

    public static void main(String[] param) {
        wert = 3 + 4;
        System.out.println(wert);
    }
}
```

- Eine statische Komponente gehört nicht zum Objekt, sondern zur Klasse.
- Wenn es mehrere Objekte gibt, hat die Komponente nur einen Wert.
- Eine statische Methode kennt kein `this`.

Lesen und Schreiben von statischen Feldern

Folgender Programmcode

```
Test objekt = new Test();
objekt.wert = 5;
```

hat den gleichen Effekt wie:

```
Test.wert = 5; // besser!
```

- Die statische Komponente `wert` ist eine Komponente von der Klasse `Test`, nicht von dem Objekt.
- Den Ausdruck `Test.wert`, kann man überall benutzen um diese Komponente zu lesen oder zu schreiben.
- In der Klasse `Test` kann man auch kurz `wert` schreiben.

Aufrufen von statischen Methoden

Eine statische Methode kann ebenfalls von überall aufgerufen werden:

```
Test.main(new String[4]);
```

- Diese Methode ist keinem Objekt zugeordnet, daher gibt es kein `this`.
- Man kann auf statische Komponenten und Methoden normal zugreifen.
- Man kann keine Instanzmethoden aufrufen (außer man hat ein Objekt).
- Aus einer Instanzmethode kann man eine statische Methode ohne Klassennamen aufrufen.

Beispiele für statische Methoden und Komponenten

- Alle Methoden im ersten Teil der Vorlesung waren statisch!
- Die Bibliotheksklasse `Math` definiert viele nützliche statische Funktionen. Zum Beispiel, `Math.pow(double x, double y)`, um Potenzen zu berechnen.
- `Math.PI` ist ein Beispiel für eine statische Komponente, die (eine Approximation von) π enthält. Die Komponente ist auch als `final` markiert, d.h., sie darf nicht überschrieben werden.
- `System.out` ist eine statische Komponente (ebenfalls konstant).
- `IOTools.readInteger()` ist eine statische Methode.

Wann benutzt man statische Methoden/Komponenten?

In folgenden Fällen benutzt man statische Methoden/Komponenten:

- Die `main`-Methode muss statisch sein, da beim Programmstart noch kein Objekt existiert.
- Für Hilfsfunktionen, die kein Objekt benötigen (z.B. die Methoden in `Math`.
- Um Konstanten wie `Math.PI` zu definieren.

Man sollte nach Möglichkeit keine nicht konstanten statischen Felder benutzen.

Konstruktoren

Konstruktoren

Bisher haben wir neue Objekte so erzeugt:

```
Test object = new Test();
```

Tatsächlich ist Test eine Konstruktor, d.h. eine spezielle Methode.

Standardmäßig ist die Methode leer, man kann aber eine eigene schreiben:

```
public class Test {
    private int x;
    public Test() {
        x = IOTools.readInteger("x:");
    }
    public static void main(String[] param) {
        Test object = new Test();
        System.err.println(object.x);
    }
}
```

Klasseninitialisierung

Die Aufgabe eines Konstruktors ist es die Klasse zu initialisieren.

- Jedes Objekt muss über einen Konstruktor erzeugt werden.
- Der Konstruktor initialisiert zunächst alle Felder.
- Dann wird der Programmcode im Konstruktor ausgeführt.

Konstruktoren

```
public class Test {
    private int x;
    public Test() {
        x = IOTools.readInteger("x:");
    }
    ...
}
```

- Konstruktoren haben keinen Rückgabetyp (nicht einmal `void`).
- Konstruktoren können `public` oder `private` sein. Man kann so verhindern, dass andere Programmteile Objekte erzeugen.
- Konstruktoren können auch Parameter haben.
- Eine Klasse kann mehrere Konstruktoren haben.
- Diese können sich gegenseitig mit `this()` aufrufen, aber nur in der ersten Zeile!

Beispiel

```
public class Value {
    public int v;
    public Value() {
        this(-1);
    }
    public Value(int i) {
        v = i;
    }
}
```

- Der zweite Konstruktor kann zum Beispiel mit `x = new Value(5)` aufgerufen werden.
- Der erste Konstruktor kann mit `x = new Value()` aufgerufen werden.
- Der erste Konstruktor ruft den zweiten mit `-1` auf.

Initialisierte Felder

Man kann den Komponenten Initialwerte geben:

```
public class Car {
    public String model;
    public int tires = 4;

    public Car(String model, int tires) {
        this.model = model;
        this.tires = tires;
    }
    public Car(String model) {
        this.model = model;
    }
}
```

- Wird eine Variable nicht initialisiert, hat sie den Wert 0 (oder null).
- Die Variable wird am Anfang des Konstruktors initialisiert.

Erweiterte Initialisierung

Man kann sogar beliebigen Code hinzufügen, der am Anfang jedes Konstruktors ausgeführt wird.

```
public class Car {
    public String model;
    public int tires = 4;
    {
        System.out.println("model: " + model
            + " tires: " + tires);
    }

    public Car(String model, int tires) {
        this.model = model;
        this.tires = tires;
    }
    public Car(String model) {
        this.model = model;
    }
}
```

Der this()-Aufruf

Im Allgemeinen ist es lesbarer gemeinsame Initialisierungen explizit in einen Konstruktor zu schreiben. Die anderen Konstruktoren können diesen dann aufrufen.

```
public class Car {
    public String model;
    public int tires = 4;

    public Car(String model, int tires) {
        this(model);
        this.tires = tires;
    }
    public Car(String model) {
        System.out.println("model: " + model
            + " tires: " + tires);
        this.model = model;
    }
}
```

Statische Initialisierung

Auch statische Felder können Initialwerte haben. Man kann auch hier beliebigen Code einfügen.

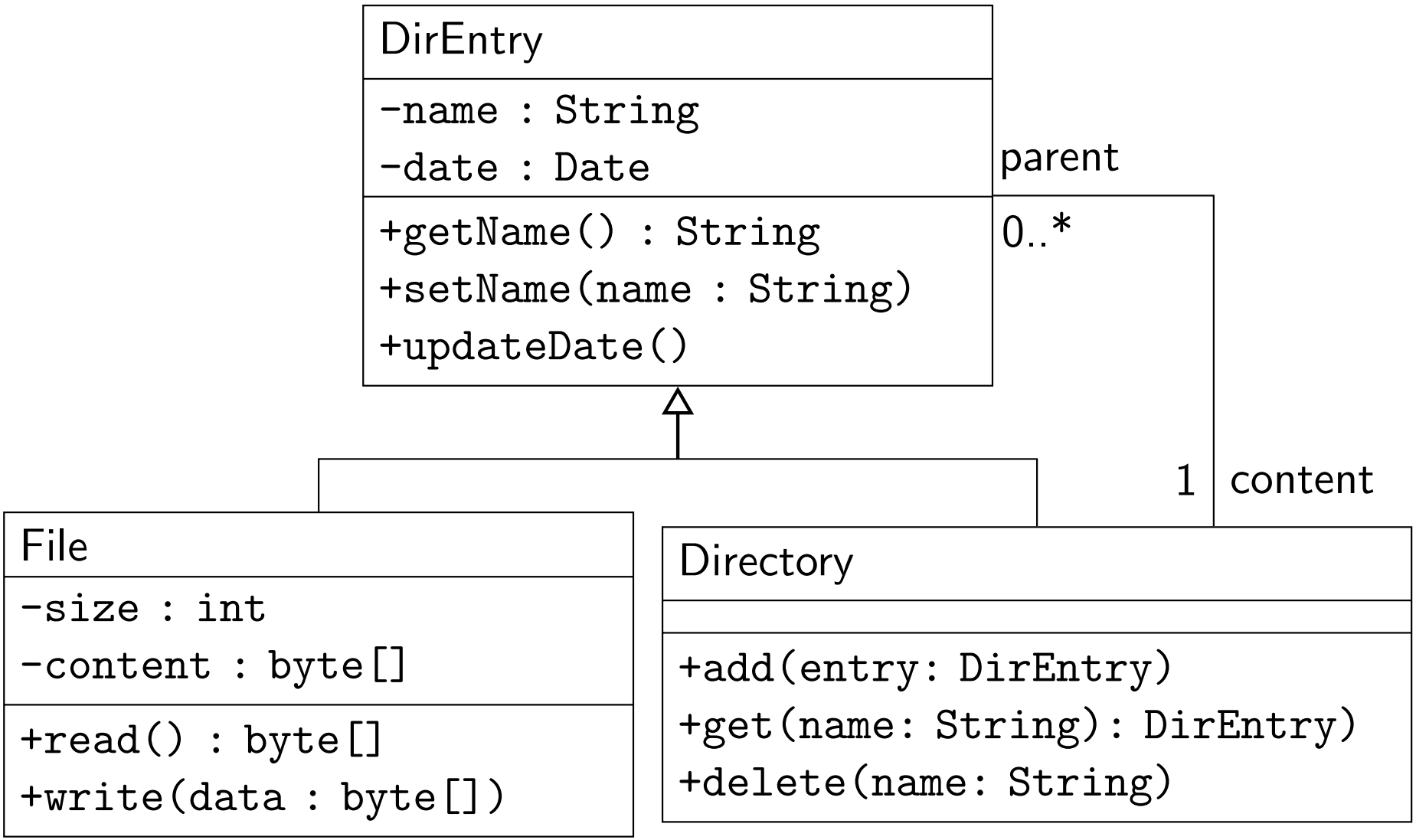
```
public class Car {
    final static Car PHANTOM = new Car();
    static {
        PHANTOM.tires = 0;
    }

    ...
}
```

Achtung

Ein `static`-Block oder ein Initialwert kann Seiteneffekte haben, die andere Klassen laden und deren `static`-Blöcke ausführen. Dann kann man sich nicht auf die Reihenfolge verlassen.

Vererbung

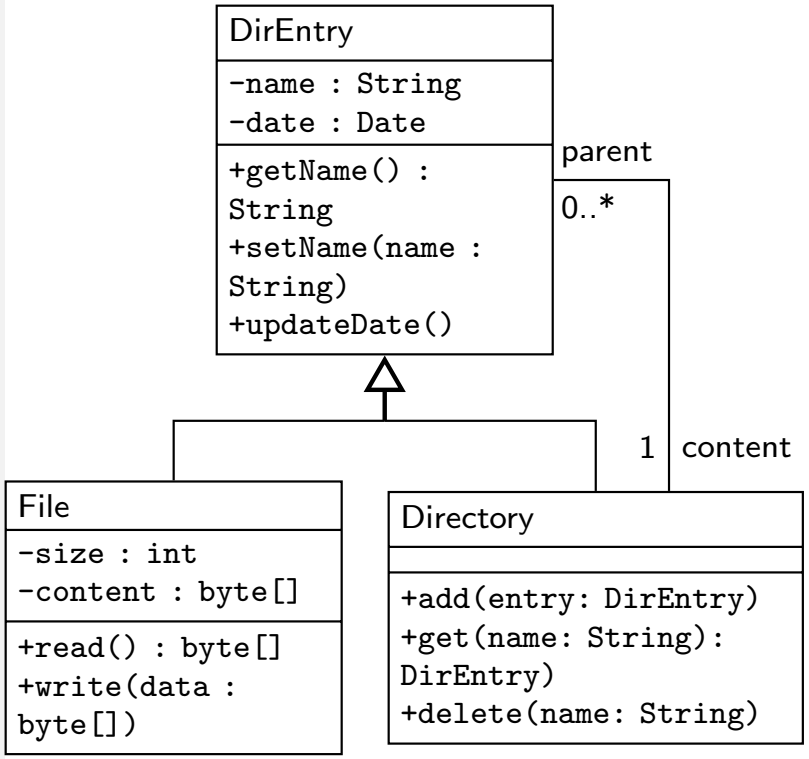
Vererbung in Java



Vererbung in Java

In Java wird die Vererbung mit dem Schlüsselwort `extends` ausgedrückt.

```
public class DirEntry {
    private String name;
    public String getName()
      ...
}

public class File extends DirEntry {
    private byte[] content;
    ...
}

public class Directory extends
    DirEntry {
    private DirEntry[] content;
    ...
}
```



Wann benutzt man Vererbung?

- `File extends DirEntry`, heißt dass `File` von `DirEntry` erbt.
- Man sollte es nur benutzen, wenn es eine "ist ein"-Beziehung gibt:
  Eine Datei ist ein Directoryeintrag.
  Ein Verzeichnis ist ein Directoryeintrag.
- Die Subklasse (die erbende Klasse) ist *spezieller* als die Superklasse.
- Jeder Code der einen Directoryeintrag erwartet, kann jetzt mit einer Datei oder einem Verzeichnis aufgerufen werden.

Automatische Typkonvertierung

Eine Datei ist ein Directoryeintrag. Daher kann man folgendes schreiben:

```
Directory root = new Directory();
Directory home = new Directory();
// Directory erbt Methode von DirEntry
home.setName("home");
// root.add erwartet DirEntry, aber
// es gibt automatische Konvertierung
root.add(home);
DirEntry entry = home;      // okay
//Directory home = entry; // Compiler-Fehler
Directory home = (Directory) entry; //okay
File home = (File) entry; // Laufzeitfehler
```

Die Klasse `Object`

Jede Klasse erbt automatisch von `java.lang.Object`.

- `Object` ist die Superklasse von allen Klassen und sogar von Feldern (Arrays).
- `Object` definiert ein paar Methoden, die jedes Objekt hat, zum Beispiel
  - `String toString()`: gibt eine lesbare Repräsentation des Objekts aus.
  - `boolean equals(Object o)`: testet ob zwei Objekte gleich sind.
  - `Class getClass()`: liefert den Laufzeittyp des Objekts.

Compile-Typ und Laufzeittyp

Weil ein `File` ein `DirEntry` ist, ist die folgende Zuweisung okay:

```
DirEntry entry = new File();
```

Die Variable `entry` hat den Typ `DirEntry`. Sie speichert eine Referenz auf ein Objekt, das den Typ `DirEntry` hat. `DirEntry` ist der Compile-Typ, denn er steht schon beim Kompilieren des Programms fest.
Zur Laufzeit kann die Variable aber eine Referenz auf ein Objekt vom Typ `File` enthalten.

- Der Laufzeittyp muss nicht mit dem Compile-Typ übereinstimmen.
- Der Laufzeittyp ist immer eine Subklasse vom Compile-Typ.

Der Operator `instanceof` kann benutzt werden, um den Laufzeittyp zu überprüfen. Außerdem hat jedes Objekt die Methode `getClass()`, die den Laufzeittyp zurück gibt.

Der Operator `instanceof`

Der Operator `instanceof` erwartet einen Ausdruck (vom Typ `Object`) und einen Klassentyp:

Syntax

Ausdruck ::= *Ausdruck* `instanceof` *Klassentyp*

Der Operator liefert ein Boolesches Ergebnis:

- `true`, falls der Laufzeittyp ein Subtyp von dem Klassentyp ist.
- `false` sonst, oder falls der Ausdruck `null` ist.

Der Klassentyp kann auch ein Feldtyp (array type) sein.

Explizite Typkonvertierung

Weil ein Directoryeintrag eine Datei oder ein Verzeichnis sein kann, erlaubt der Compiler Typkonvertierungen.
Mit `instanceof` kann man sich vergewissern, ob ein Objekt einen Typ hat.

```
void test(DirEntry entry) {
    if (entry instanceof Directory) {
        Directory dir = (Directory) entry;
        ...
    } else if (entry instanceof File) {
        File file = (File) entry;
        ...
    }
}
```

Wenn bei einer Typkonvertierung der Typ nicht stimmt, stürzt das Programm ab (ClassCastException).

Konstruktoren in Subklassen

```
public class DirEntry {
    String name;
    public DirEntry(String name) {
      this.name = name;
    }
}

public class File extends DirEntry {
    private byte[] content;
    public File(String name, byte[] content) {
        super(name);
        this.content = content;
    }
    public File(String name) {
        this(name, new byte[0]);
    }
}
```

Konstruktoren und Vererbung

Jeder Konstruktor muss entweder

- einen anderen Konstruktor der gleichen Klasse `this(...)` aufrufen,
- oder einen Konstruktor der Superklasse `super(...)` aufrufen.
- Wenn es nicht explizit angegeben ist, wird `super()` aufgerufen.

Initialisierung von Klassen

Die Initialisierung läuft wie folgt:

1. Das Objekt wird mit Nullen initialisiert,
2. der Super-Konstruktor wird aufgerufen,
3. die Felder werden initialisiert und expliziten Initialisierungen werden ausgeführt.
4. der eigentliche Konstruktor wird ausgeführt.

Ruft ein Konstruktor einen anderen Konstruktor mit `this(...)` auf, kümmert sich der aufgerufene Konstruktor um den zweiten und dritten Punkt.

Das Schlüsselwort `final`

Mit dem Schlüsselwort `final` hat drei Bedeutungen:

- Vor einer Klasse (z.B. `public final class String`) verbietet es Subklassen zu erzeugen.
  Weil die Klasse `String` final ist, gibt es einen Compiler-Fehler, wenn man von dieser Klasse ableitet.
- Vor einer Methode verbietet es Subklassen, diese Methode zu überschreiben.
- Vor einer Komponente verbietet es den Wert zu ändern. Damit werden Konstanten definiert (z.B.
  `public static final double PI=3.141592653589793`).
  Konstanten sind meist statisch. Nicht statische final Komponenten kann man benutzen, um ein Objekt unveränderlich (immutable) zu machen. `String` ist ein Beispiel für ein unveränderliches Objekt.

Polymorphie

Motivation

Wir wollen unsere Verzeichniseinträge mit Icons ausstatten

- Es soll eine Methode geben, die das Icon zurückgibt.
- Dateien haben andere Icons als Verzeichnisse.
- Wenn man den Inhalt eines Verzeichnis ausgibt, will man keine Fallunterscheidung machen.

Datei-Icons

```
public class DirEntry {
    public Icon getIcon() {
        return UNKNOWN;
    }
}
public class Directory
    extends DirEntry {
    public Icon getIcon() {
        return DIRECTORY;
    }
}
```

```
public class File
    extends DirEntry {
    public Icon getIcon() {
        return FILE;
    }
}
void test() {
    DirEntry entry =
        new Directory();
    Icon i = entry.getIcon();
}
```

Welche `getIcon()`-Methode wird in `test` aufgerufen?

Polymorphie

- Die erbende Klasse kann eine Methode der Superklasse überschreiben (Englisch: override).
- Welche Methode aufgerufen wird, hängt davon ab, welchen Laufzeittyp das Objekt hat.
- Die überschreibende Methode kann das Originalmethode aufrufen.

```
public Icon getIcon() {
  if (!hasIcon)
    return super.getIcon();
  ...
}
```

Abstrakte Methoden und Klassen

In unserem Beispiel gibt es keine Objekte vom Typ DirEntry.
Man kann das durch das Schlüsselwort `abstract` explizit machen.

```
public abstract class DirEntry {
    public abstract Icon getIcon();
}
public class Directory
    extends DirEntry {
    public Icon getIcon() {
        return DIRECTORY;
    }
}
```

- Eine abstrakte Methode darf nur in einer abstrakten Klasse sein.
- Die erbenden Klassen müssen die Methode überschreiben (es sei denn, sie sind selbst abstrakt)
- Eine abstrakte Methode hat keinen Code (Implementierung).
- `new` darf nicht auf abstrakten Klassen benutzt werden.

Interfaces (Schnittstellen)

Der Extremfall einer abstrakten Klasse ist ein Interface.

- In einem Interface sind alle Methoden abstrakt und öffentlich. Man kann `public abstract` weglassen.
- Alle Komponenten sind statisch, konstant und öffentlich (`public static final`).
- Eine Klasse kann nur von einer anderen Klasse erben, aber von beliebig viele Interfaces.

Bei einem Interface spricht man in Java nicht von erben oder erweitern, sondern von implementieren. Es wird das Schlüsselwort `implements` statt `extends` benutzt.

Wann benutzt man Interfaces?

Anwendungszwecke für Interfaces:

- Um mehrere austauschbare Komponenten zu erlauben. Alle Komponenten implementieren das gleiche Interface.
  Beispiel: Ein Schachprogramm könnte ein Interface zur Anbindung von Schach-KIs definieren. Dann kann man die KI einfach austauschen.
- Um Rückmeldungen zu geben.
  Beispiel: Eine Methode die länger läuft, könnte eine Objekt als Parameter erwarten, dass ein Interface `ProgressListener` implementiert. Diese Interface enthält eine Methode `progress(double percentage)`, die z.B. einen Fortschrittsbalken anzeigt.

Teil VII

Erweiterte Programmkonstrukte

Code-Konventionen

Konventionen

Konventionen dienen dazu, den Code lesbarer zu machen. Dazu gehört

- richtige Einrückung,
- Zeilenumbrüche,
- konsequente Bennenung von Variablen/Klassen/Methoden,
- Kommentare, insbesondere JavaDoc-Kommentare.

Einrückung

Die generellen Regeln der Einrückung sind:

- Blöcke (durch { und } gekennzeichnet) werden eingerückt.
  Alle Blöcke werden immer gleichweit eingerückt.
- Bei mehrzeiligen Befehlen, werden die folgenden Zeilen eingerückt.

Bei größeren Projekten sollte abgesprochen werden,

- wie weit eingerückt wird (4 Spaces, 2 Spaces, 8 Spaces?),
- ob Tabs benutzt werden sollen und wie breit ein Tab ist,
- ob geschweifte Klammern in extra Zeilen kommen; werden sie eingerückt?

Viele Entwicklungsumgebungen können automatisch einrücken. Man muss allerdings dem Tool die Konventionen mitteilen.

Zeilenumbrüche

- Generell gilt, eine Anweisung pro Zeile.
- Auch bei kurzen Anweisungen, z.B. `i++;`
- Ein einzeiliger Schleifenrumpf oder Then/Else-Teil kommen immer in eine eigene Zeile.
- Lange Zeilen sollten zusätzlich umgebrochen werden.

Namenskonventionen

Die Groß-/Kleinschreibung von Bezeichnern soll helfen Typen von Variablen zu unterscheiden.

- Bezeichner in Java benutzen üblicherweise `DieCamelCaseKonvention`.
- Klassennamen haben großen Anfangsbuchstaben.
- Methoden- und Komponentennamen und Variablen haben kleinen Anfangsbuchstaben.
- Konstanten werden komplett großgeschrieben.
- Paketnamen werden komplett kleingeschrieben.
- Methodennamen fangen üblicherweise mit einem Verb an.
- Der Bezeichner sollte kurz, knapp und präzise sein.
- Je globaler ein Bezeichner ist, desto besser sollte er gewählt sein.

Aufzählungstypen

Aufzählungstypen (Enumerations)

Ein Aufzählungstyp kann nur endlich viele Werte annehmen.

Beispiel

```
public enum CardSuit = {
    HEARTS, DIAMONDS, CLUBS, SPADES;
}
```

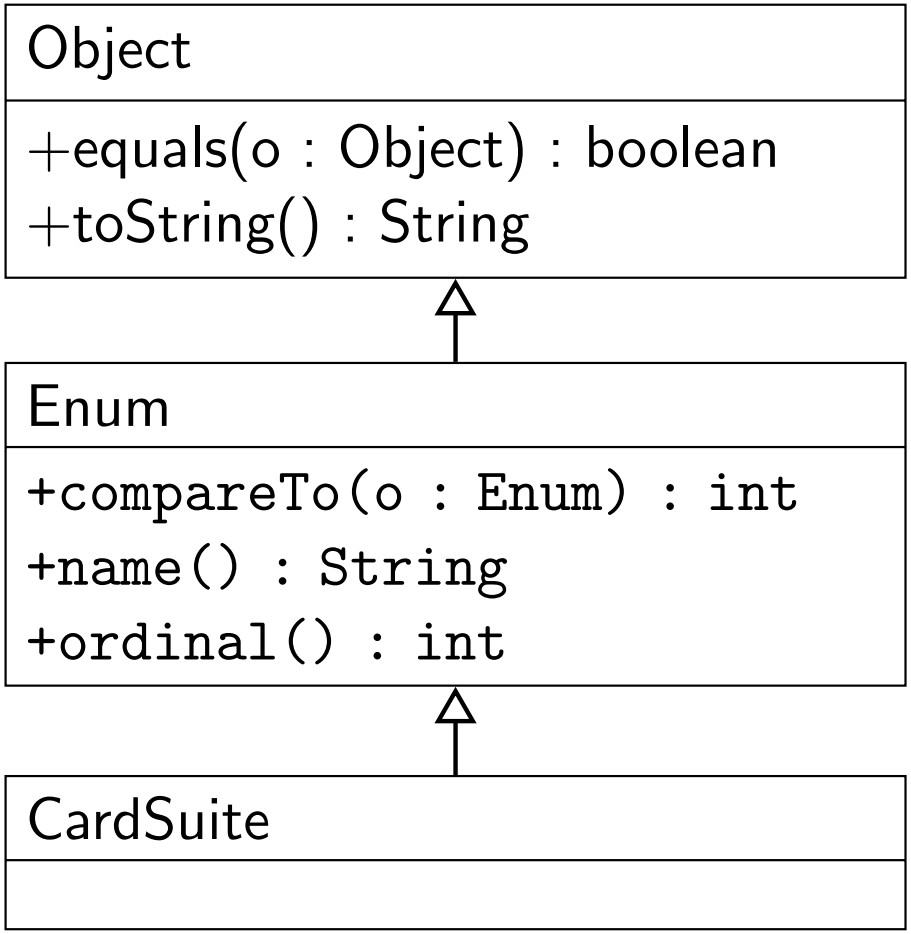
Ein Aufzählungstyp ist eine Klasse. Es gibt nur vier Objekte von dieser Klasse. Auf diese kann mit `CardSuit.HEARTS` ... zugegriffen werden.

Klassenhierarchie

Alle Aufzählungstypen erben von der Klasse `java.lang.Enum` die Methoden

- `ordinal()` gibt die Ordinalzahl zurück, 0 für die erste, usw.
- `name()` gibt den Namen zurück
- `toString()` gibt ebenfalls den Namen aus.

Aufzählungstypen haben keinen (öffentlichen) Konstruktor; man kann keine weiteren Werte erzeugen.



Aufzählungstypen und switch

Aufzählungstypen können in switch verwendet werden:

```
CardSuit suit = card.getSuit();
switch(suit) {
case HEARTS:
case DIAMONDS:
    System.out.println("rot");
    break;
case CLUBS:
case SPADES:
    System.out.println("schwarz");
    break;
}
```

Der Compiler warnt sogar, wenn man einen Fall vergessen hat.

Methoden in Aufzählungstypen

Man kann auch eigene Funktionen in Aufzählungstypen hinzufügen:

```
public enum CardSuit = {
    HEARTS, DIAMONDS, CLUBS, SPADES;

    public String getColor() {
        switch(this) {
        case HEARTS: case DIAMONDS:
            return "red";
        default:
            return "black";
        }
    }
}
```

Exceptions (Ausnahmen)

Fehler in Java

In Java werden Fehler durch spezielle Objekte, Exceptions, gekennzeichnet.
Wir haben schon einige kennengelernt:

- NullPointerException
- ArrayIndexOutOfBoundsException
- ArithmeticException

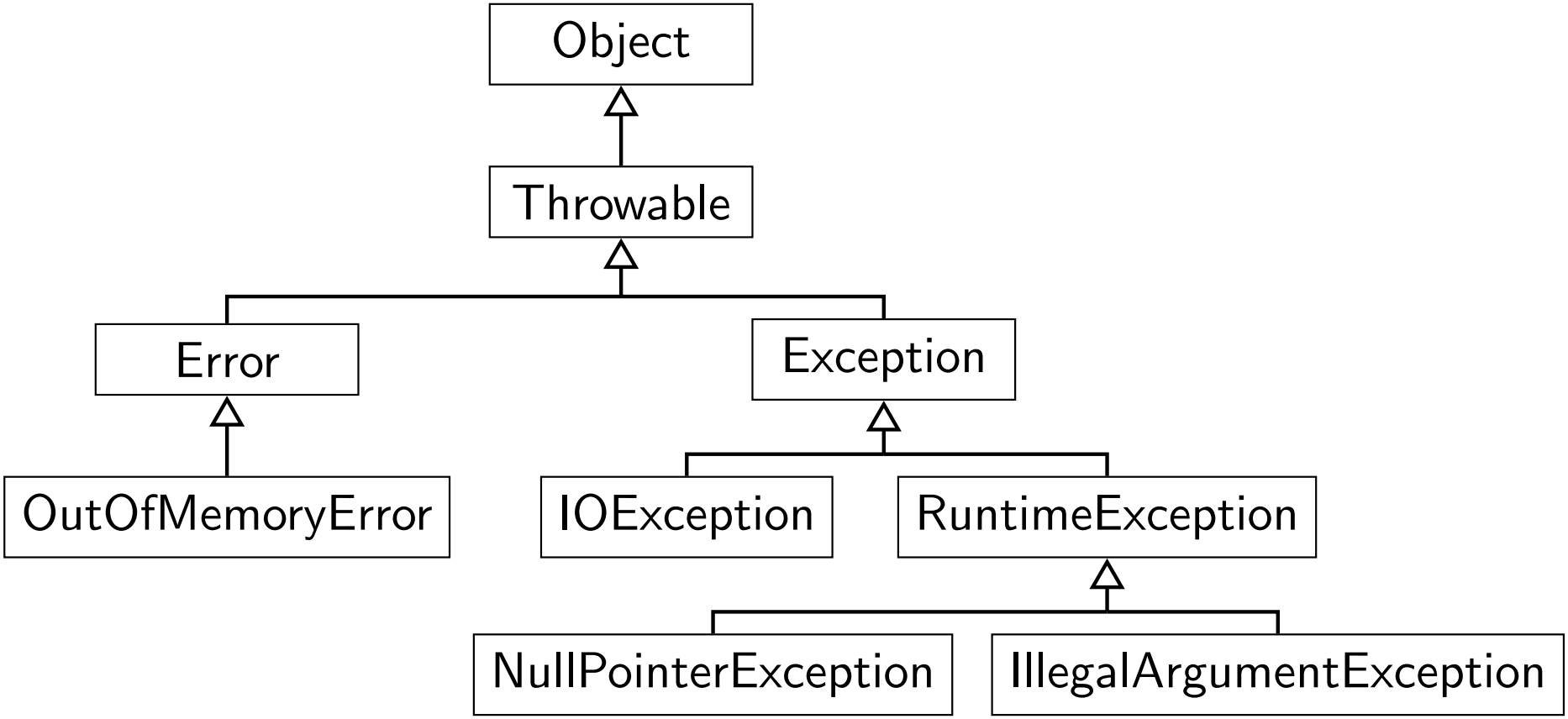
Fehlerausgabe

Wenn ein Fehler auftritt, wird normalerweise das Programm beendet und ein „Stacktrace“ ausgegeben, der besagt, wo der Fehler passiert ist:

```
Exception in thread "main" java.lang.OutOfMemoryError: Java
    heap space
at de.uni_freiburg.informatik.ultimate.smtinterpol.convert.
    Clausifier.run(Clausifier.java:1955)
at de.uni_freiburg.informatik.ultimate.smtinterpol.convert.
    Clausifier.addFormula(Clausifier.java:2024)
at de.uni_freiburg.informatik.ultimate.smtinterpol.smtlib2.
    SMTInterpol.assertTerm(SMTInterpol.java:907)
at expriments.TestSMTInterpol.first_example(TestSMTInterpol.
    java:56)
at expriments.TestSMTInterpol.main(TestSMTInterpol.java:25)
```

- Erste Zeile gibt an, welcher Fehler aufgetreten ist.
- Zweite Zeile benennt die Klasse, Methode, Datei und Zeilennummer.
- Die Zeilen darunter geben an, wer die Methode aufgerufen hat.

Klassenhierarchie der Exceptions



Eigene Exceptions

Man kann in Java seine eigenen Exceptions bauen

```
public class MyException extends Exception
    public MyException(String message,
                       Throwable cause) {
        super(message, cause);
    }
    public MyException(String message) {
        super(message);
    }
}
```

Der Konstruktor nimmt bis zu zwei Parameter

- `String message` ist die Nachricht die am Ende ausgegeben werden soll.
- `Throwable cause` kann eine andere Exception benennen, die diesen Fehler verursacht hat. Es wird dann auch die andere Exception ausgegeben.

Exceptions werfen

In Java kann man das Kommando `throw` benutzen, um einen Fehler zu werfen:

```
public long factorial(int n) {
    if (n < 0)
        throw new IllegalArgumentException
            ("argument must not be negative");
    ...
}
```

Normalerweise wird `throw` mit *newException* benutzt. Man kann aber auch eine gefangene Exception erneut werfen.

Fehler fangen

Man möchte oft nicht, dass das Programm abstürzt, sondern stattdessen eine sinnvolle Fehlermeldung ausgeben und weitermachen.
Fehler können in Java mit folgender Syntax gefangen werden:

```
try {
    ... oeffne Datei ...
    ... lese Datei ...
} catch (FileNotFoundException ex) {
    System.out.println("Die Datei wurde nicht
       gefunden!");
}
... mache normal weiter ...
```

Wie werden Exceptions gefangen

Wenn eine Exception geworfen wird,

- wird zunächst der innerste `try`-Block der die Exception fängt gesucht.
- Gibt es in der Methode keinen, wird in der aufrufende Methode gesucht.
- Wenn der Block gefunden wurde und es einen passenden `catch`-Block gibt, wird mit dem Catch-Block weitergemacht.
- Gibt es keinen `try`-Block im gesamten Programm, wird das Programm beendet und die Exception ausgegeben.
- Wenn keine Exception geworfen wird, wird nach dem `try`-Block der `catch`-Block übersprungen.

Exception-Hierarchie

Ein `catch`-Block fängt alle Exception-Objekte von dem entsprechenden Typ *und alle Subklassen*.

Man kann also mit

```
catch (Exception ex) {
    ...
}
```

Alle `Exceptions`, auch `RuntimeException`, `NullPointerException`, `IOException` fangen.

Error, Exception, und RuntimeException

- Alle Subklassen von `Error` bezeichnen Fehler, die man nicht sinnvoll abfangen kann.
  Zum Beispiel: `OutOfMemoryError` (nicht genügend Speicher), `NoClassDefFoundError` (.class-Datei liegt nicht auf dem Rechner).
- Alle Subklassen von `RuntimeException` können jederzeit auftreten.
- Alle anderen Subklassen von `Exception` müssen explizit deklariert werden. Insbesondere `IOException`.

```
void readFile(String filename)
    throws IOException {...
```

Wenn eine Methode, die die Exception nicht deklariert eine andere aufruft, die sie deklariert, muss sie die Exception fangen und verarbeiten.

Teil VIII

Programmieren und Datenstrukturen

Algorithmen

Algorithmus

Ein Algorithmus ist eine Vorschrift zur Lösung eines Problems.
Folgende Eigenschaften sind essentiell

- Finitheit: Die Vorschrift (also das Programm) muss endlich sein, darf zu jedem Zeitpunkt nur endlich viel Speicher benutzen,
- Terminiertheit: Ein Algorithmus darf nur endlich viele Schritte insgesamt durchführen.
- Effektivität: Für jede Anweisung eines Algorithmus muss klar sein, was sie macht und wie man sie ausführt.

Weitere wünschenswerte Eigenschaften:

- Determiniertheit: Das Ergebnis ist immer das selbe.
  Determinismus: Der Rechenweg ist fest vorgegeben.

Beispiele

- Ein Kochrezept ist finit. Es erfüllt Terminiertheit. Die Effektivität hängt von der Vorbildung des Kochs/der Genauigkeit der Beschreibung ab. Determiniertheit ist oft nicht gegeben (... schmecken Sie mit Salz und Pfeffer ab).
- Algorithmus zum Multiplizieren von mehrstelligen Dezimalzahlen: Finitheit, Terminiertheit, Effektivität und Determiniertheit sind gegeben. Determinismus nicht unbedingt.
  Tatsächlich ist der Begriff eine Abwandlung des Namens des Gelehrten Muhammed al-Chwarizmi, der in seinem Lehrbuch das „Über das Rechnen mit indischen Ziffern“ (um 825) solche Algorithmen vorgestellt hat.

Euklidischer Algorithmus

Einer der ältesten Algorithmen stammt von Euklid (3. Jahrhundert v.u.Z)

Die Elemente des Euklid, Buch VII, Proposition 2

Es soll der größte gemeinsame Teiler von AB und CD, die nicht teilerfremd sind bestimmt werden.
[...] Wenn CD nicht Teiler von AB ist, subtrahiert man, von den beiden Zahlen AB und CD ausgehend, immer die kleinere von der größeren bis die entstandene Zahl Teiler der ihr vorhergehenden ist, der dann der größte gemeinsame Teile von AB und CD ist.

Der Rest der Proposition gibt ein Beispiel und beweist die Korrektheit des Algorithmus.

Euklidischer Algorithmus in Java

Man kann diesen Algorithmus direkt als Java-Programm implementieren:

```
public int computeGCD(int ab, int cd) {
    if (ab <= 0 || cd <= 0)
        throw new IllegalArgumentException();
    while (ab % cd != 0) {
        if (ab > cd) {
            int new = ab - cd;
            ab = cd;
            cd = new;
        } else
            cd -= ab;
    }
    return cd;
}
```

Euklidischer Algorithmus in Java

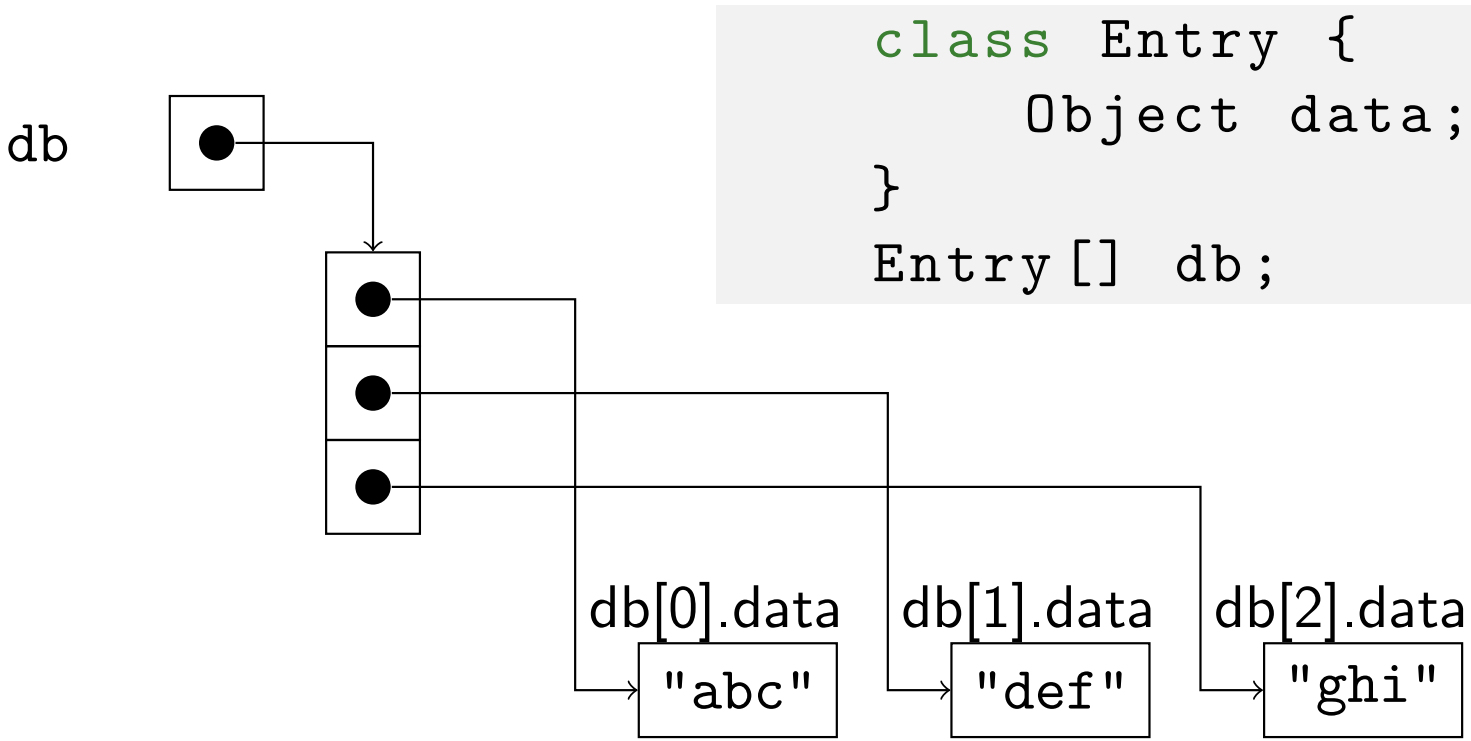
Leicht verbesserte Variante:

```
public int computeGCD(int ab, int cd) {
    if (ab < 0 || cd < 0)
        throw new IllegalArgumentException();
    while (cd != 0) {
        int new = ab % cd;
        ab = cd;
        cd = new;
    }
    return ab;
}
```

Verkettete Listen

Dynamische Datenstrukturen

Objekte in Java werden als Referenzen (Zeiger) gespeichert.



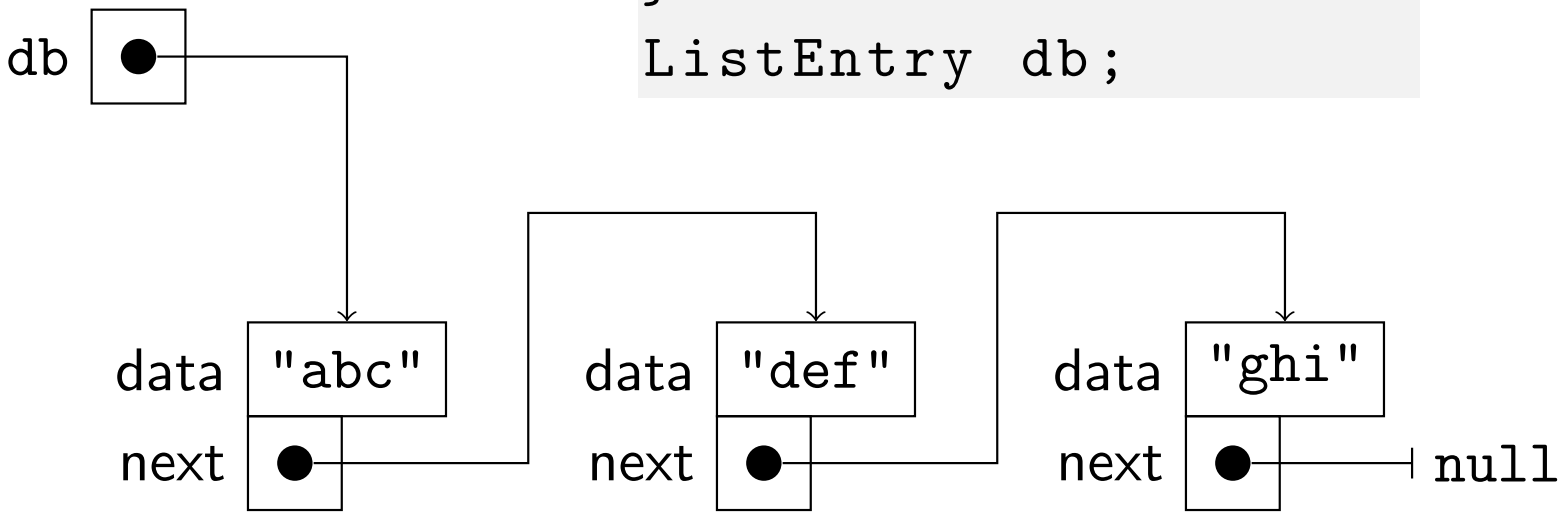

```
class Entry {
    Object data;
}
Entry[] db;
```

Eine weitere Möglichkeit mehrere Objekte zu speichern, ist sie zu verketten.

Verkettete Liste

Eine Komponente im Objekt ist eine Referenz auf das nächste Objekt.

```
class ListEntry {
    Object    data;
    ListEntry next;
}
ListEntry db;
```



Einfach verkette Listen

Das Beispiel auf der letzten Folie zeigte eine einfach verkettete Liste. Eine Implementierung ist:

```
public class List
    ListEntry head;

    /** Erzeugt eine neue leere Liste. */
    public List() {
      head = null;
    }

    /** Prueft, ob die Liste leer ist. */
    public boolean isEmpty() {
      return head == null;
    }
    ...
```

Einfach verkette Listen (2)

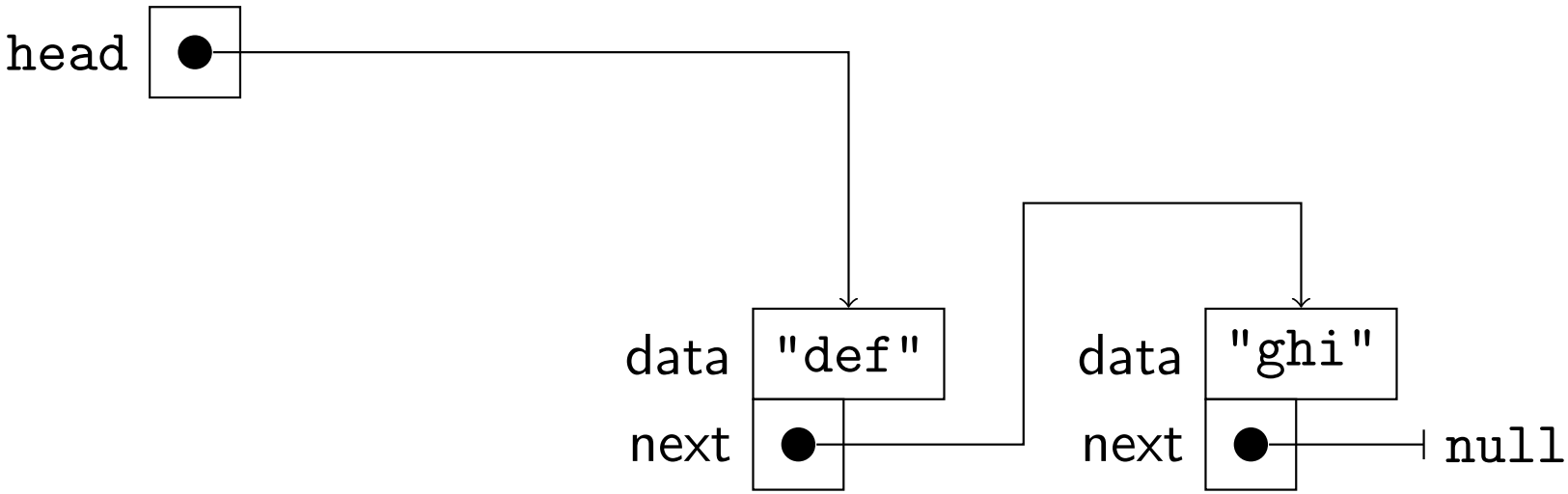
Um ein Element in eine Liste einzufügen, kann man es leicht als erstes Element einfügen.

```
//public class List
    ...
    /** Fuegt ein neues Element ein. */
    public void add(Object data) {
        ListEntry newEntry = new ListEntry();
        newEntry.data = data;
        newEntry.next = head;
        head = newEntry;
    }
    ...
```

Man kann aber auch an einer anderen beliebigen Stelle einfügen (siehe Übung).

Einfügen in Listen

Wir wollen `"abc"` in die Liste `["def","ghi"]` einfügen.

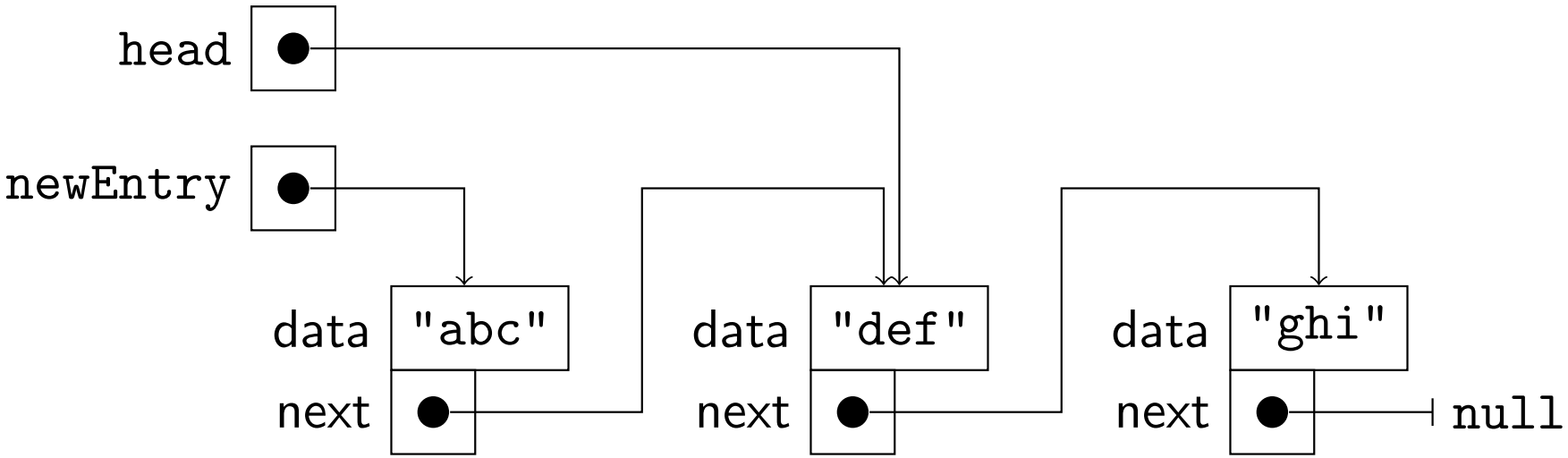


Einfügen in Listen

Wir wollen "abc" in die Liste ["def","ghi"] einfügen.

```
ListEntry newEntry = new ListEntry();
newEntry.data = "abc";
newEntry.next = head;
```

Die ersten drei Zeilen erzeugen einen neuen Listeneintrag



Einfügen in Listen

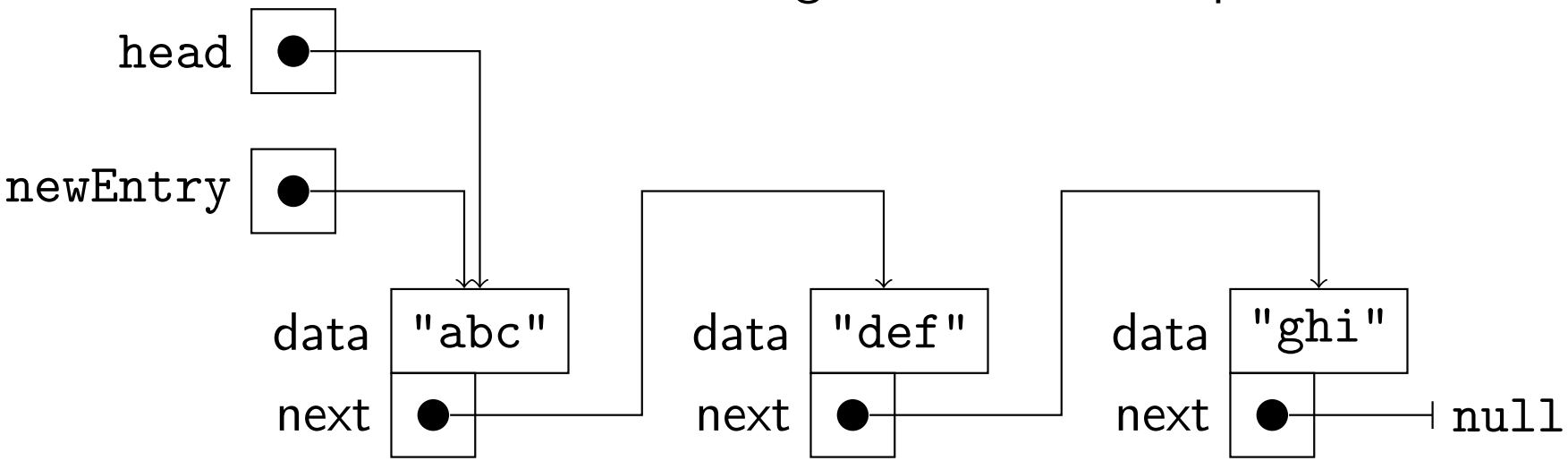
Wir wollen "abc" in die Liste ["def","ghi"] einfügen.

```
ListEntry newEntry = new ListEntry();
newEntry.data = "abc";
newEntry.next = head;
```

Die ersten drei Zeilen erzeugen einen neuen Listeneintrag

```
head = newEntry;
```

Die letzte Zeile macht diesen Eintrag zum neuen Startpunkt.



Iterieren

Um über eine Liste zu laufen, kann man sich durch die next-Referenzen hangeln:

```
//public class List
...
/** Gibt alle Elemente als String zurueck */
public String toString() {
    String content="";
    String comma="";
    for (ListEntry e = head;
         e != null; e = e.next){
        content += comma + e.toString();
        comma = ",";
    }
    return "[" + content + "]";
}
```

Löschen

Um ein Element zu löschen, muss es zunächst gefunden werden. Daher bauen wir folgenden Algorithmus:

1. Suche zu löschendes Element in der Liste. Merke dabei den Vorgänger.
2. Entferne das Element aus der Liste.

Löschen – Teil 1

Zunächst durchlaufen wir die Liste und merken uns dabei den Vorgänger.

```
//public class List
...
/** Entfernt das Element data aus der Liste. */
public boolean remove(Object data) {
    ListEntry prev = null;
    ListEntry elem = head;
    while (elem != null
            && !elem.data.equals(data)) {
        prev = elem;
        elem = elem.next;
    }  ...
```

Ist `elem == null`, so wurde das Element nicht gefunden. Andernfalls zeigt `elem` auf das zu löschende Element, `prev` auf den Vorgänger oder `null`, wenn das erste Element gelöscht werden soll.

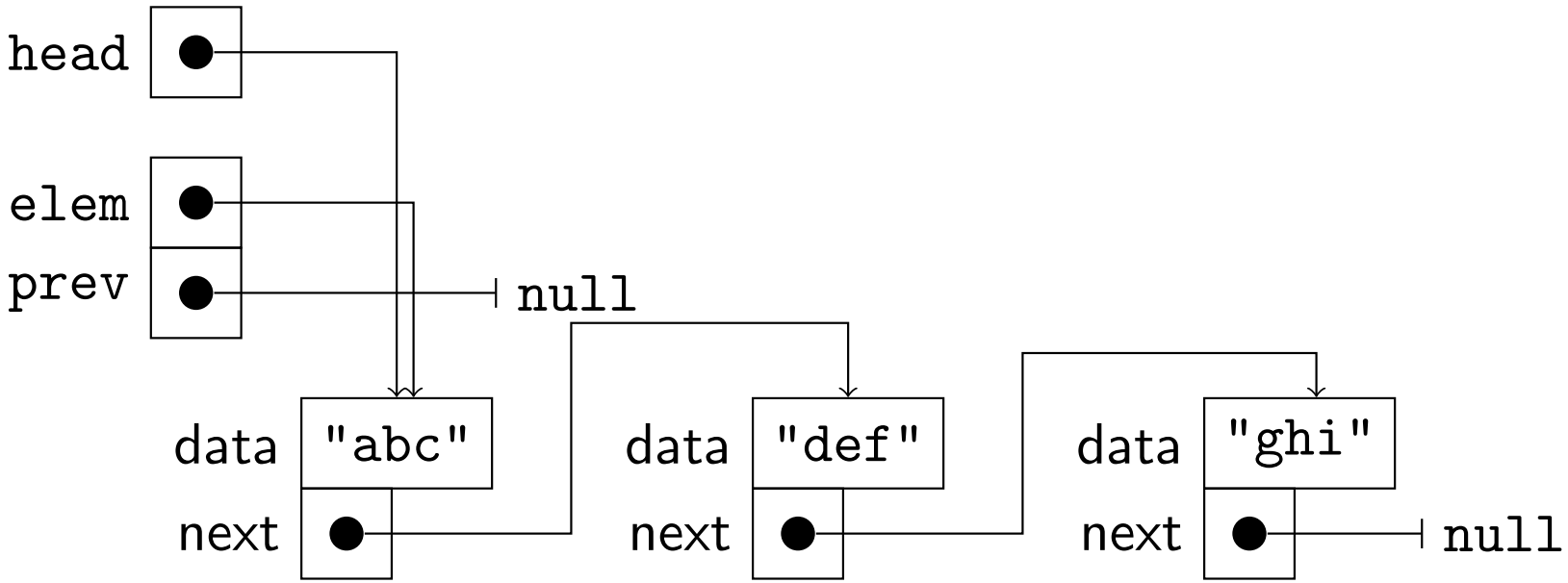
Löschen – Teil 2

```
...//boolean remove(Object data)
    if (elem == null) {
        // data wurde nicht gefunden.
        return false;
    } else if (prev == null) {
        // erstes Element soll geloescht werden.
        assert head == elem;
        head = elem.next;
        return true;
    } else {
        // haenge elem aus der Liste aus.
        assert prev.next == elem;
        prev.next = elem.next;
        return true;
    }
}
```

Löschen – Animation

Zunächst löschen wir `"def"`:

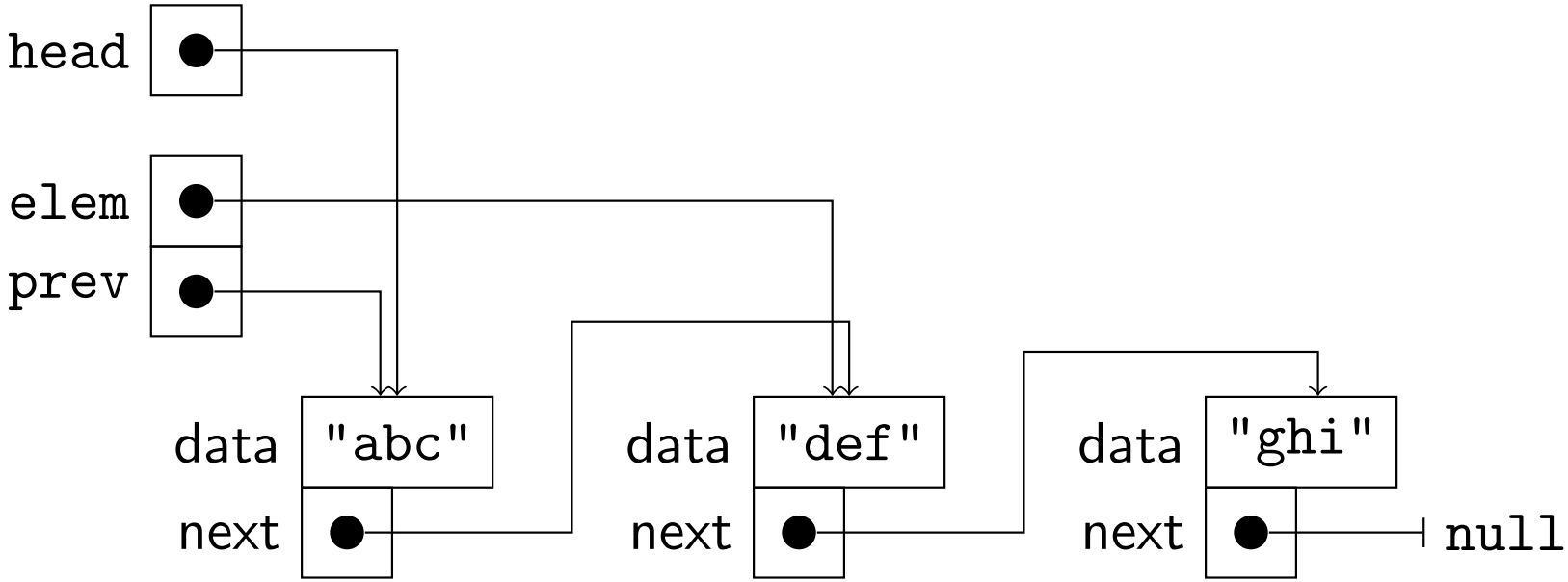
Initialisierung



Löschen – Animation

Zunächst löschen wir `"def"`:

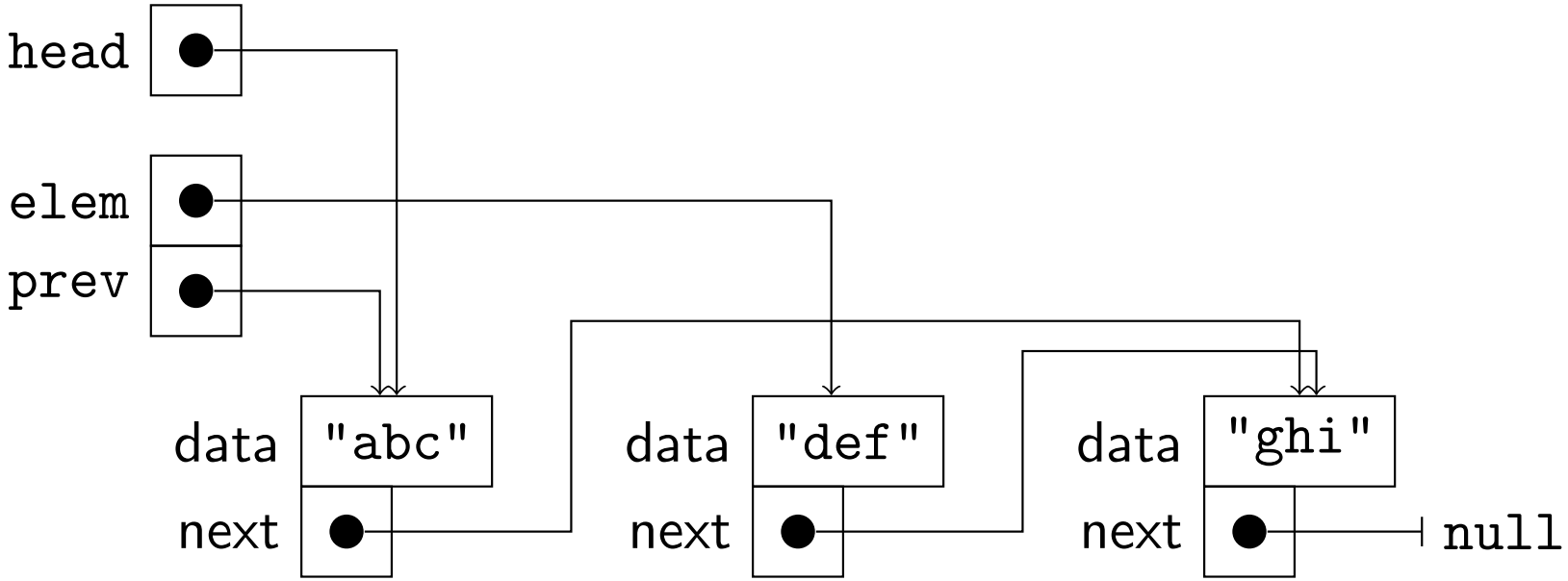
Suche `elem` mit `elem.data="def"`.



Löschen – Animation

Zunächst löschen wir `"def"`:

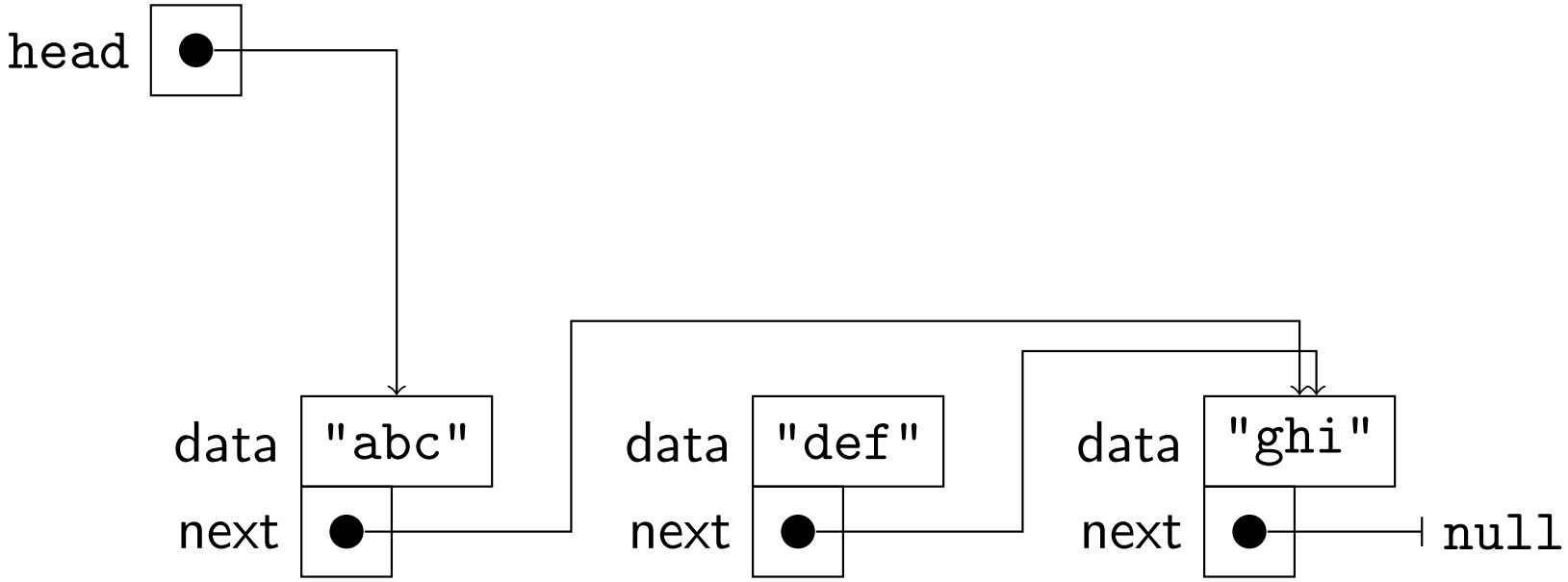
Aushängen (else-Teil).



Löschen – Animation

Zunächst löschen wir `"def"`:

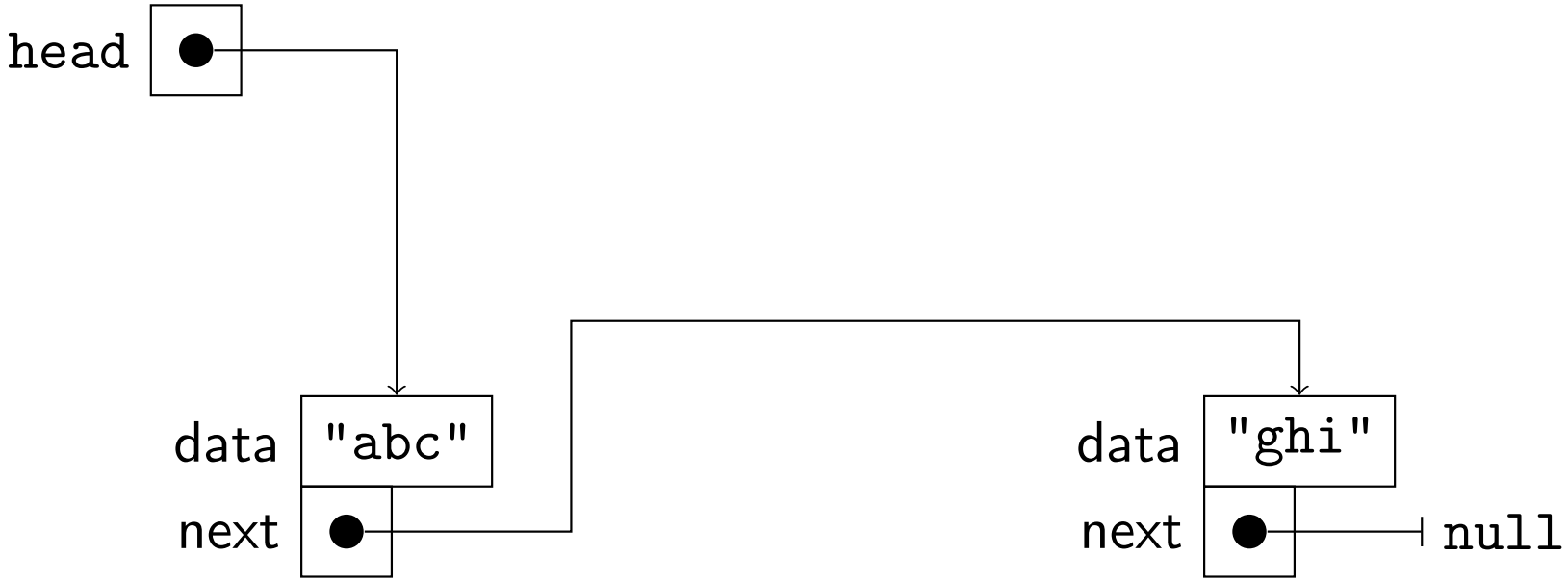
Methode gibt `true` zurück.



Löschen – Animation

Zunächst löschen wir `"def"`:

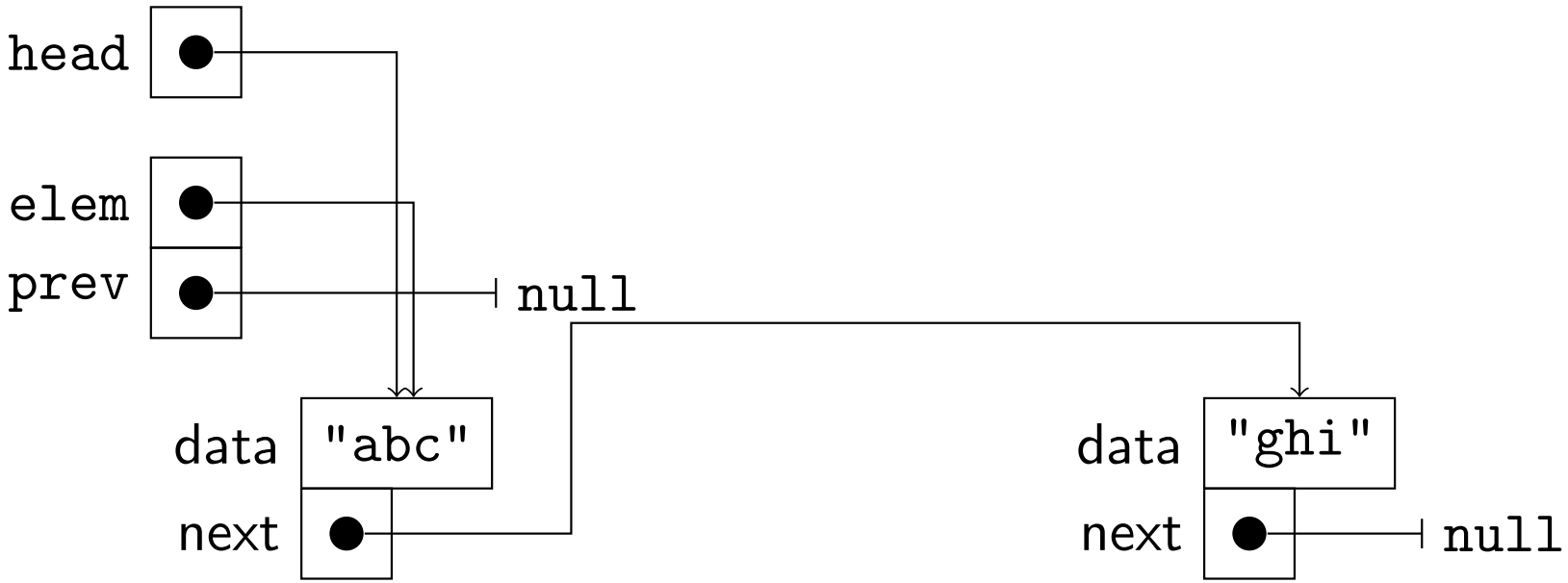
Müllsammlung (garbage collection) läuft.



Löschen – Animation

Jetzt löschen wir "abc":

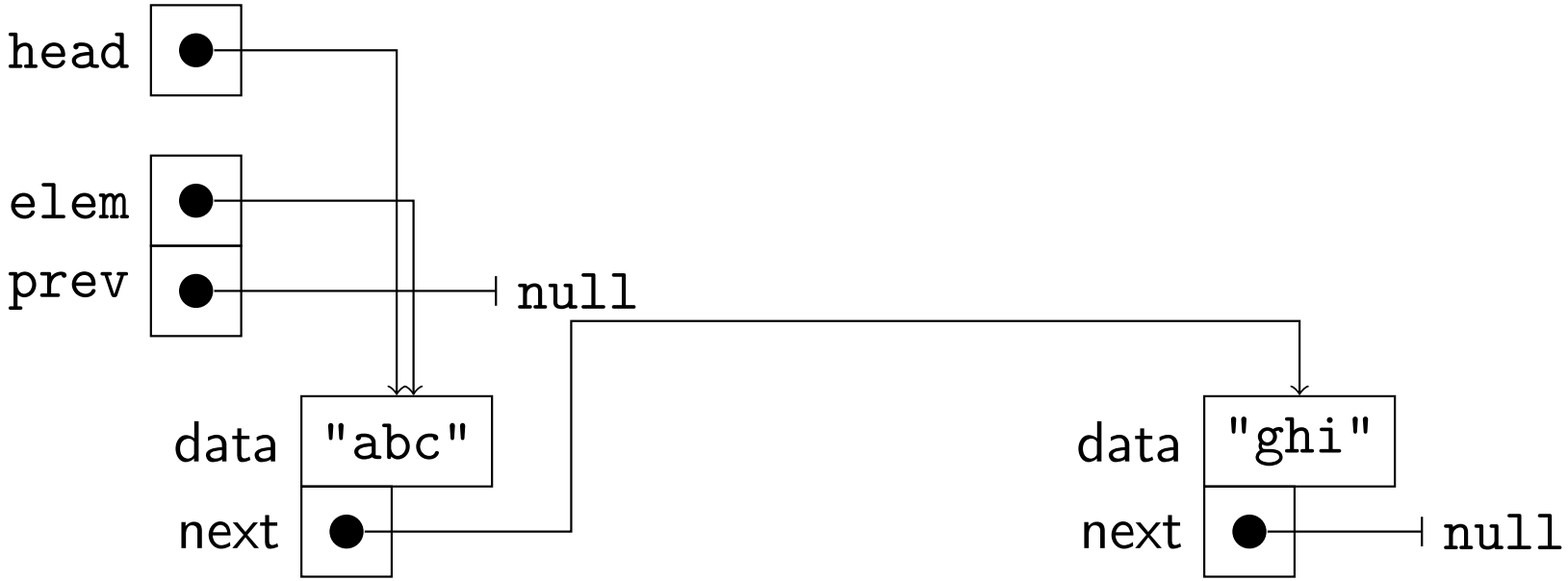
Initialisierung



Löschen – Animation

Jetzt löschen wir `"abc"`:

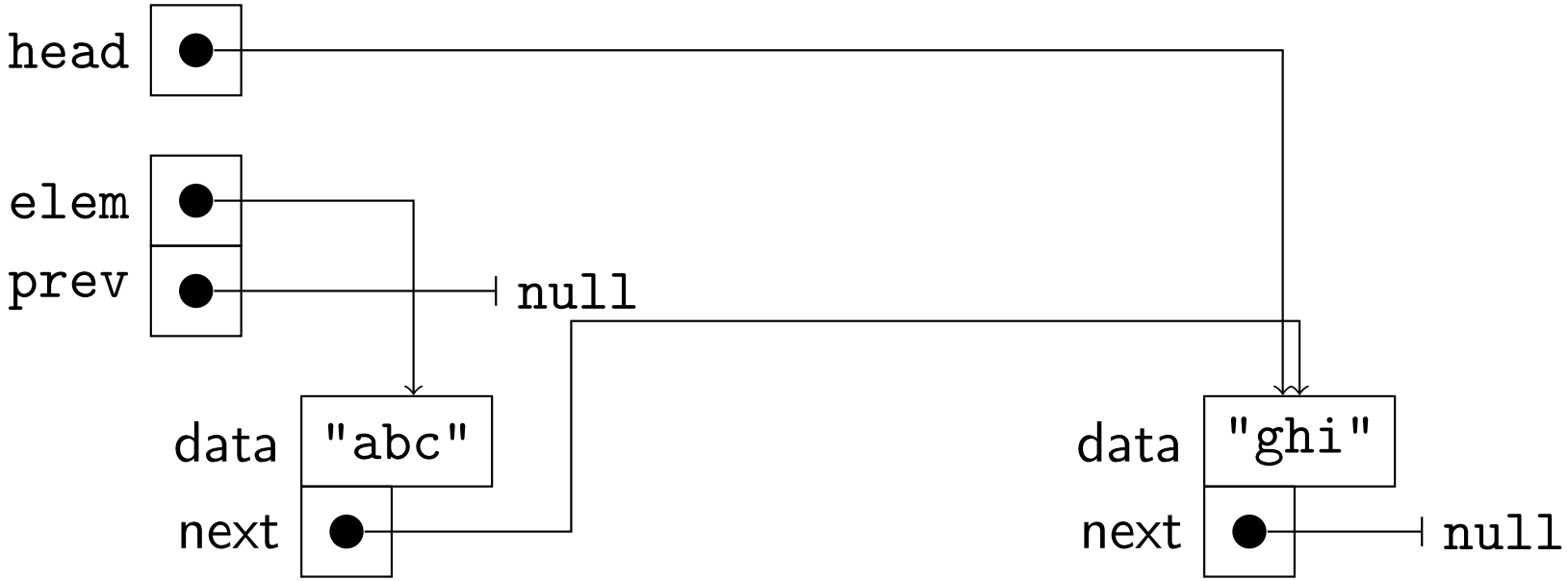
Suche `elem` mit `elem.data="abc"`.



Löschen – Animation

Jetzt löschen wir "abc":

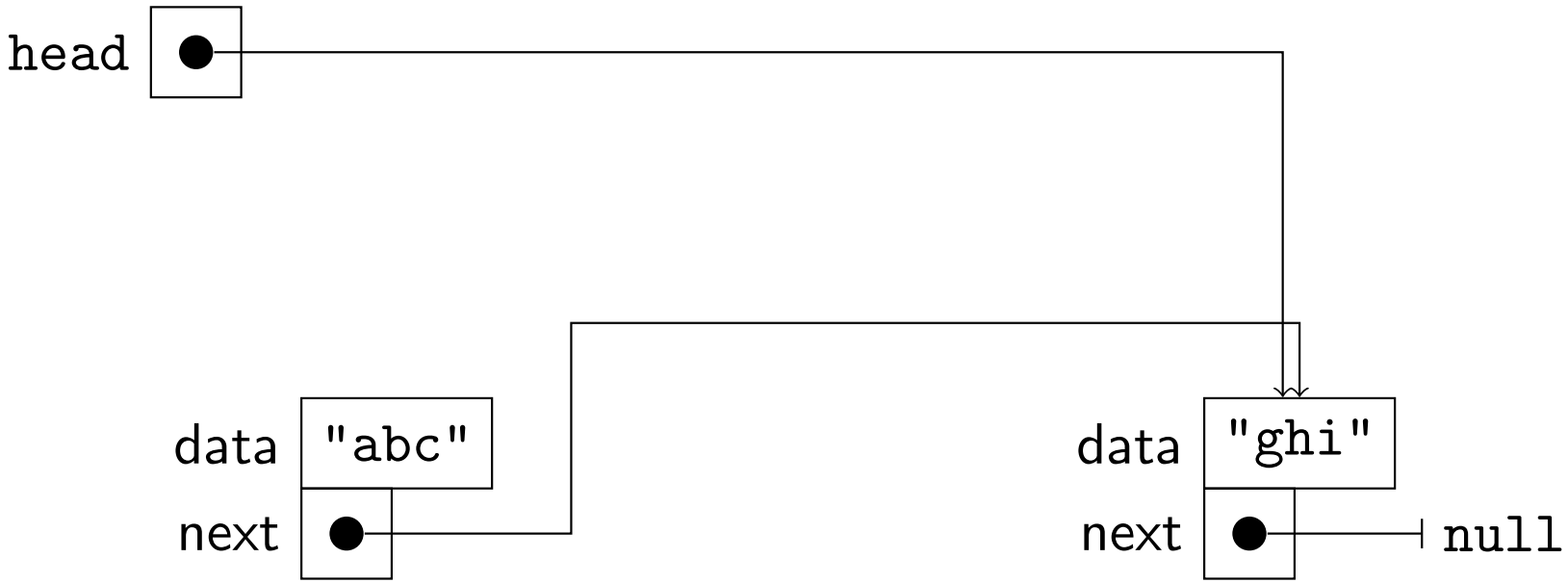
Aushängen (`prev == null`).



Löschen – Animation

Jetzt löschen wir `"abc"`:

Methode gibt `true` zurück.



Löschen – Animation

Jetzt löschen wir `"abc"`:

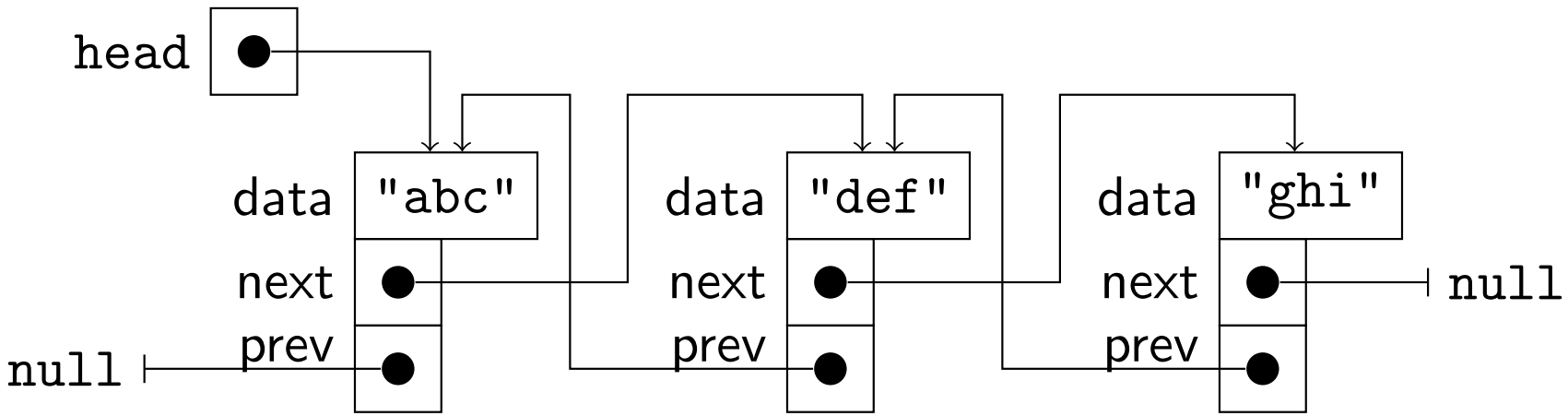
Müllsammlung (garbage collection) läuft.

Doppelt Verkettete Liste

Eine doppelt verkettete Liste speichert zwei Referenzen pro Listeneintrag, einen Vorgänger und einen Nachfolger:

```
class ListEntry {
    Object    data;
    ListEntry prev;
    ListEntry next;
}
```

Das macht Löschen einfacher. Auch kann man die Liste rückwärts durchlaufen.

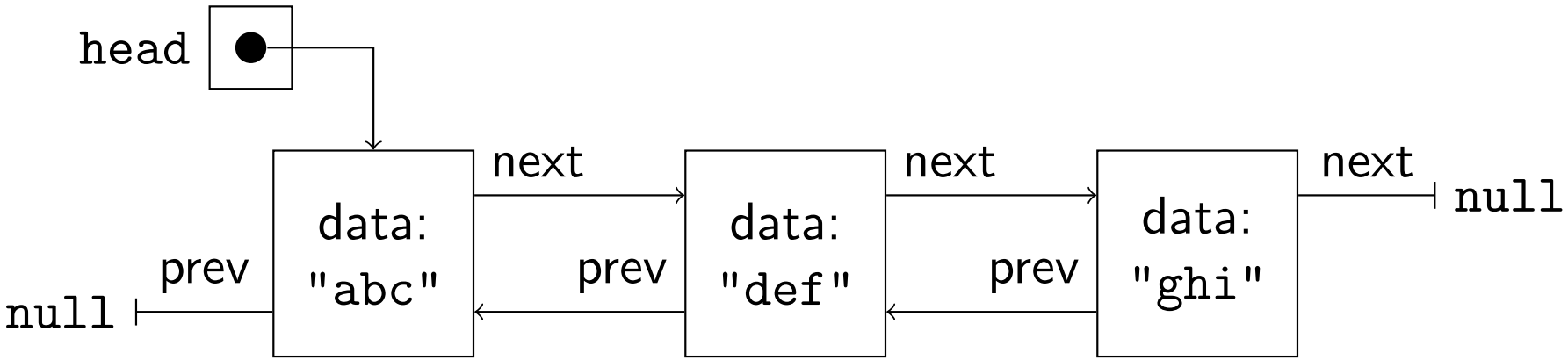


Doppelt Verkettete Liste

Eine doppelt verkettete Liste speichert zwei Referenzen pro Listeneintrag, einen Vorgänger und einen Nachfolger:

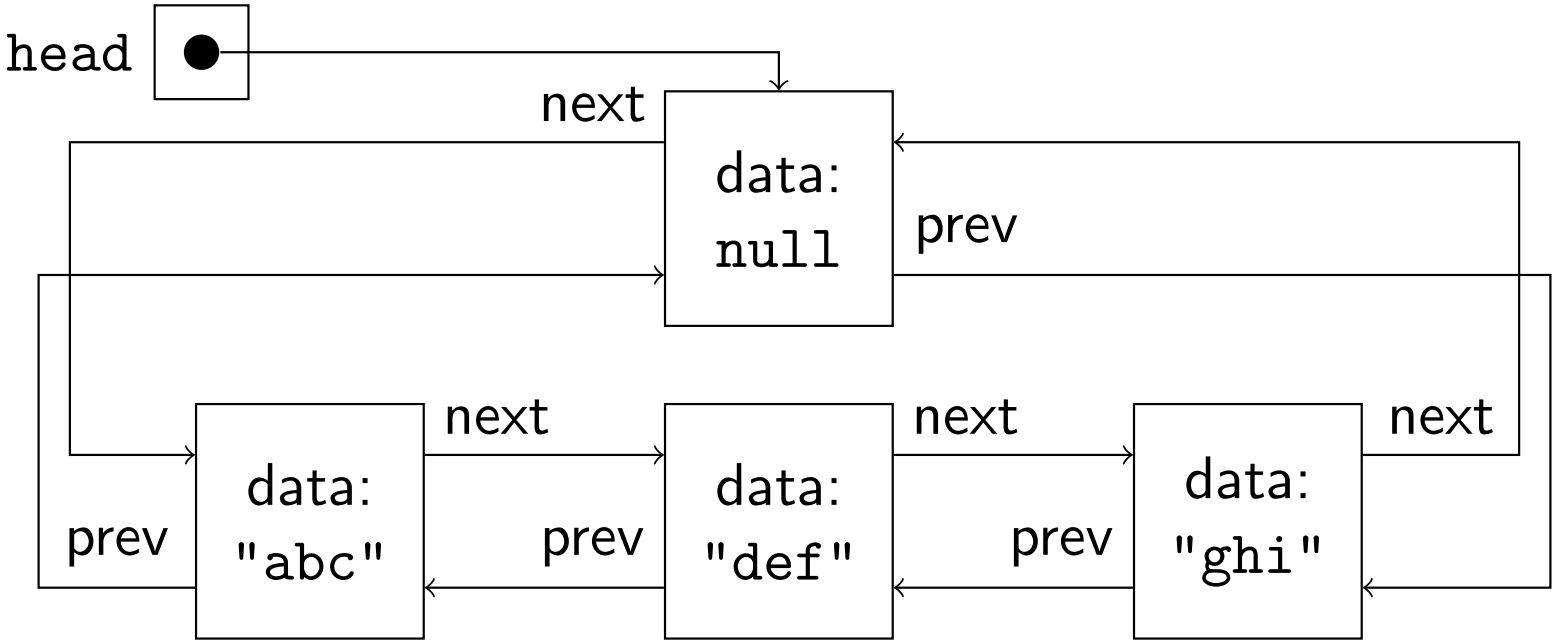
```
class ListEntry {
    Object    data;
    ListEntry prev;
    ListEntry next;
}
```

Das macht Löschen einfacher. Auch kann man die Liste rückwärts durchlaufen.



Blindelement

Statt einer Referenz auf `null` benutzt man bei doppelt verketteten Listen gerne ein Blindelement:



Dadurch kann man sich Fallunterscheidungen sparen. Man spricht hier auch von einer zyklisch verketteten Liste.

Doppelt verkette Liste

Eine Implementierung ist:

```
public class DoublyLinkedList
    ListEntry head;

    /** Erzeugt eine neue leere Liste. */
    public List() {
      head = new ListEntry();
      head.next = head.prev = head;
    }

    /** Prueft, ob die Liste leer ist. */
    public boolean isEmpty() {
      return head.next == head;
    }
    ...
```

Einfügen in doppelt verkettete Listen

Hier fügen wir ein Element hinten in die Liste ein.

```
//public class DoublyLinkedList
    ...
    /** Fuegt ein neues Element ein. */
    public void add(Object data) {
        ListEntry newEntry = new ListEntry();
        newEntry.data = data;
        newEntry.next = head;
        newEntry.prev = head.prev;
        newEntry.prev.next = newEntry;
        newEntry.next.prev = newEntry;
    }
    ...
```

Man kann aber auch vorne einfügen. Dafür muss man nur die zweite und dritte Zeile anpassen.

Iterieren

Beim Iterieren muss man bis zum Blindelement laufen:

```
//public class DoublyLinkedList
...
/** Gibt alle Elemente als String zurueck */
public String toString() {
    String content="";
    String comma="";
    for (ListEntry e = head.next;
         e != head; e = e.next){
        content += comma + e.toString();
        comma = ",";
    }
    return "[" + content + "]";
}
```

Löschen – Teil 1

Auch hier muss ein Element zunächst gefunden werden, um es zu löschen.

```
//public class DoublyLinkedList
...
/** Entfernt das Element data aus der Liste. */
public boolean remove(Object data) {
    ListEntry elem = head.next;
    while (elem != head
           && !elem.data.equals(data)) {
        elem = elem.next;
    }
    ...
```

Ist `elem == head`, so wurde das Element nicht gefunden. Andernfalls zeigt `elem` auf das zu löschende Element.

Löschen – Teil 2

Dank des Blindelements, entfällt im zweiten Teil eine Fallunterscheidung.
Das funktioniert auch mit einelementigen Listen.

```
...//boolean remove(Object data)
    if (elem == head) {
        // data wurde nicht gefunden.
        return false;
    } else {
        // Element aus der Liste loeschen.
        elem.prev.next = elem.next;
        elem.next.prev = elem.prev;
        return true;

    }
}
```

Ausblick: Bibliotheken

Man muss beim Programmieren das Rad nicht immer neu erfinden.

- `java.util.LinkedList` ist eine Implementierung für verkettete Listen in Java.
- Es gibt auch anders implementierte Listen, wie `java.util.ArrayList`. Diese haben Vor- aber auch Nachteile. Um diese einschätzen zu können sollte man wissen, wie verkettete Listen funktionieren.
- In vielen Fällen reicht es aus eine Implementierung aus der Java-Bibliothek zu verwenden. Nur selten braucht man spezielle „handmodellierte“ Datenstrukturen.

Generische Typen

Generische Typen: Motivation

In der letzten Vorlesung hatten wir eine verkettete Liste programmiert. Das funktioniert ganz gut, auch um z.B. Zeichenketten einzufügen:

```
List list = new DoublyLinkedList();
list.add("Hallo"); // impliziete
   Typkonvertierung
```

Wenn wir allerdings eine getFirst()-Methode hinzufügen, muss man beim auslesen den Typ konvertieren.

```
String elem = (String) list.getFirst();
```

Wenn wir den falschen Typ einfügen, meldet der Compiler keine Fehler. Erst beim Auslesen schlägt die Typkonvertierung fehl.

Vergleich: Arrays

Bei Arrays gibt es für jeden Typ einen zugehörigen ArrayTyp, z.B. `String[]` um `String`-Objekte zu speichern.
Können wir etwas ähnliches für Listen tun?
Ja, es gibt generische Datentypen

Generische Liste

In Java kann eine Klasse einen Typparameter haben, der in eckigen Klammern übergeben wird.

```
public class SinglyLinkedList<E> {
    class ListEntry {
        E         data;
        ListEntry next;
    }
    public void add(E data) {...
    public E getFirst() {...
```

- `E` ist immer ein Referenztyp, d.h. er erbt von `Object`.
- `E` kann fast wie ein normaler Typ benutzt werden.
- Es gibt aber Dinge, die nicht funktionieren und zu Warnungen führen, z.B. Typkonvertierung `(E)`.

Benutzung von Generics

Wenn man die Klasse `DoublyLinkedList<E>` benutzen will, muss man angeben was `E` ist (man muss `E` instanziieren):

```
DoublyLinkedList<String> list = new
    DoublyLinkedList<String>();
list.add("Hallo");  // okay
String s = list.getFirst(); // okay
list.add((Integer) 4); // Compilerfehler
```

Der Compiler prüft dann auch, ob die Typen passen.

Wildcards

Wenn einem der Typ der Instanziierung egal ist, kann man auch Wildcards benutzen:

```
    private void printFirst(List<?> list) {
        if (!list.isEmpty())
            Object o = list.getFirst();
            System.out.println(o.toString());
        }
    }
```

Allerdings kann man dan zu solch einer Liste nichts hinzufügen (nur mit Compilerwarnungen).
Man kann auch noch den Typ von `?` genauer eingrenzen: Die Klasse `List<? implements Comparable>` speichert Objekte, die Comparable implementieren.

Vererbungshierarchie

Eine generische Liste `List<String>` erbt nicht von `List<Object>` (siehe Übung). Das hängt damit zusammen, dass `List<String>` die Methode `add(Object data)` nicht implementiert.
Allerdings ist `List<String>` eine Subklasse von

- `List<?>`,
- `List<? extends String>`,
- `List<? implements Comparable>`.
- `List<? super String>`.

Fallstricke

- Java unterstützt keine Laufzeittypen für Generics. Java merkt sich zur Laufzeit nur den Basistyp List, nicht die generischen Parameter.
- Eine Typkonvertierung z.B. auf `List<String>` kann nicht zur Laufzeit überprüft werden.
- Auch `instanceof` funktioniert nicht auf generischen Typen.
- Eine Typkonvertierung auf `E` in der Klasse $List < E >$ kann zur Laufzeit nicht überprüft werden.
- Man kann kein Array vom Typ `E[]` erzeugen.

Kompatibilität

Aus historischen Gründen gibt es in Java zu der Klasse `List<E>` den Typ `List`, der sich wie `List<Object>` verhält. Auch hier gibt es Compilerwarnungen.

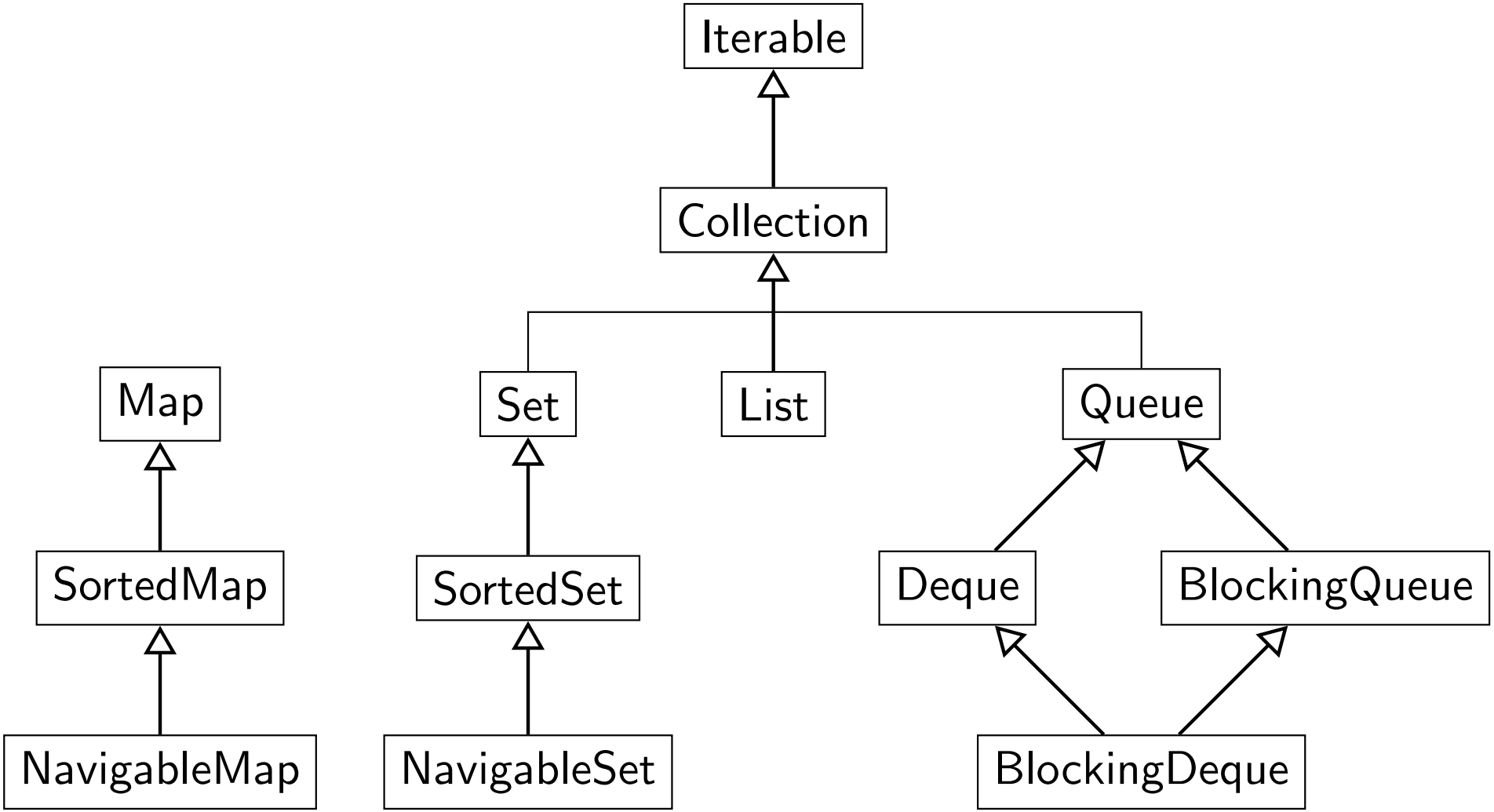
Collection-Bibliothek

Collection-Bibliothek

Die Java-Bibliothek stellt eine Reihe von Klassen zur Verfügung um Daten zu speichern.

- Container-Klassen um Objekte in verketteten Listen, Arrays, oder Bäumen zu speichern.
- Abbildungen (Maps), um Objekte miteinander zu verknüpfen
- Interfaces, die es erlauben, Objekte hinzuzufügen, entfernen, suchen, iterieren.
- Verschiedene Implementierungen, die für unterschiedliche Zwecke geeignet sind.

Collection-Hierarchie: Interfaces



Iterable

Das Interface `Iterable<E>` hat nur eine Methoden:

- `Iterator<E> iterator()` gibt einen Iterator zurück der alle Objekte aufzählt.

Der Iterator hat wiederum drei Methoden:

- `boolean hasNext()`: gibt true zurück, wenn es noch weitere Objekte gibt.
- `E next()`: gibt bei jedem Aufruf das nächste Element zurück. Wirft eine Exception, wenn es kein weiteres Element gibt.
- `void remove()` (optional): löscht das letzte Element aus der Collection.

Damit kann man alle Objekte aufzählen.

Iterable und for-Schleifen

Wenn eine Klasse `Iterable<E>` implementiert gibt es eine „schöne“ Syntax der for-Schleife:

```
Iterable<String> list = ....;
for (String s : liste) {
    ....
}
```

Hier wird der Schleifenrumpf für jedes Element der Liste einmal aufgerufen. Es ist eine Abkürzung für:

```
Iterable<String> list = ....;
Iterator<String> iterator = liste.iterator();
while (iterator.hasNext()) {
    String s = iterator.next();
    ....
}
```

Collection

Das Interface `Collection<E>` hat zusätzliche Methoden:

- `boolean add(E e)` fügt ein Element hinzu.
- `boolean addAll(Collection<? extends E> c)` fügt alle Elemente einer anderen Collection hinzu. r anderen Collection.
- `boolean clear()` löscht alle Elemente.
- `boolean contains(Object o)` prüft ob ein Element enthalten ist.
- `boolean containsAll(Collection<?> e)` prüft ob alle Elemente enthalten sind.
- `boolean isEmpty()` prüft ob die Collection leer ist.
- `boolean remove(Object o)` löscht ein Element.
- `boolean removeAll(Collection<?> c)` löscht alle Elemente eine
- `boolean retainAll(Collection<?> c)` löscht alle Elemente, die nicht in der anderen Collection liegen.
- `int size()` gibt die Anzahl der Elemente zurück.
- `T[] toArray(T[])` gibt ein Array mit allen Elementen zurück.

equals und hashCode

Die Klasse `Object` definiert zwei für Collections relevante Methoden:

- `boolean equals(Object o)` vergleicht das Objekt mit einem anderen Objekt, und gibt `true` zurück, falls die Objekte gleich sind.

  Eigenschaften `o.equals(o)== true`, `o.equals(null)== false` und `o1.equals(o2)== o2.equals(o1)` und Transitivität sollten gelten.
- `int hashCode()` gibt einen „Fingerabdruck“ des Objekts als `int` zurück. Gleiche Objekte müssen den gleichen Fingerabdruck haben. Verschiedene Objekte sollten (aus Effizienzgründen) verschiedene Fingerabdrücke haben.

  Letzteres ist nicht immer möglich, da es nicht genug Integer gibt. Es sollte aber mit großer Wahrscheinlichkeit gelten.

Collections: Übersicht

Es gibt vier Hauptarten von Collections

- `Set`: Speichert Objekte ohne Duplikate in beliebiger Reihenfolge (oder sortiert in `SortedSet`).
- `List`: Speichert Objekte mit Duplikaten in fester Reihenfolge.
- `Queue`: Speichert Objekte mit Duplikaten (mit oder ohne Reihenfolge) mit schnellem Zugriff auf das erste Element.
  Die Erweiterung `Deque` (double ended queue) erlaubt das schnelle hinzufügen und entfernen sowohl vorne, als auch hinten.
- `Map`: Speichert Assoziation zwischen Schlüssel- und Wertobjekten, mit schnellem Zugriff auf einen Wert anhand des Schlüssels. Obwohl `Map` selbst keine Collection ist, bilden die Schlüssel ein `Set` und die Werte eine `Collection`.

Implementierungen

Zu jedem Interface in der Collection-Hierarchie gibt es ein oder mehrere Implementierungen:

- `Map`: `HashMap` und `LinkedHashMap`
- `SortedMap`/`NavigableMap`: `TreeMap`
- `Set`: `HashSet` und `LinkedHashSet`
- `SortedSet`/`NavigableSet`: `TreeSet`
- `List`: `ArrayList` und `LinkedList`
- `Queue`: `PriorityQueue`.
- `Deque`: `ArrayDeque`, `LinkedList`

Sie unterscheiden sich vor allem in der Effizienz einiger Methoden.

Die Interfaces `Comparator<E>` und `Comparable<E>`

Für `SortedSet` muss man Objekte vergleichen können. Dafür gibt es das Interface `Comparator<E>`, mit einer Methode:

- `int compare(E x, E y)`: Gibt einen negativen Wert zurück, falls $x < y$, Null, wenn x = y und positiven Wert wenn $x > y$ ist. Dabei kann größer/kleiner eine beliebige totale Ordnung sein.

Statt einen `Comparator` kann eine Klasse auch eine natürliche Ordnung auf sich selbst definieren und `Comparable<E>` implementieren:

- `int compareTo(E y)` hat den gleichen Kontrakt wie $compare(this, y)$.

Teil IX

Alapo – Design eines größeren Programms

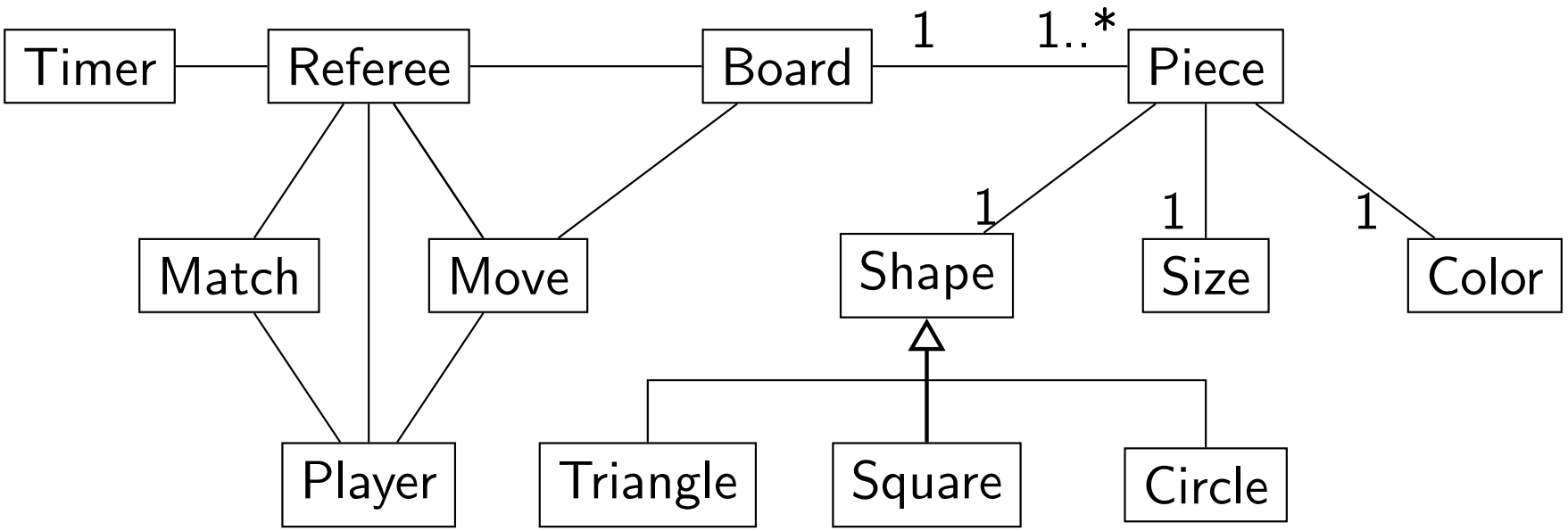
Alapo

Als Programmierübung wollen wir ein Programm schreiben, dass Alapo spielt. Diese Programme sollen am Ende in einem Turnier antreten.

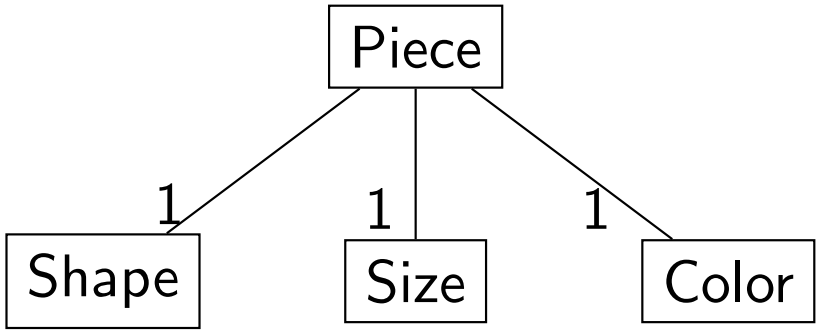
In einer Alapo-*Partie* treten zwei *Spieler* gegeneinander an. Sie berechnen abwechselnd *Züge*, die auf einem *Brett* ausgeführt werden. Auf dem Brett befinden sich *Spielfiguren* von verschiedenen *Formen* (*Dreieck*, *Quadrat*, *Kreis*), *Größen* (klein, groß) und *Farben* (schwarz, weiß). Ein *Schiedsrichter* überprüft diese Züge. Weiter solle der Schiedsrichter die *Bedenkzeit* der Spieler zu überwachen.

Klassen

In der objektorientierten Programmierung werden als erstes die benötigten Klassen aufgeschrieben. Die Hauptwörter der Beschreibung geben einen ersten Anhaltspunkt:



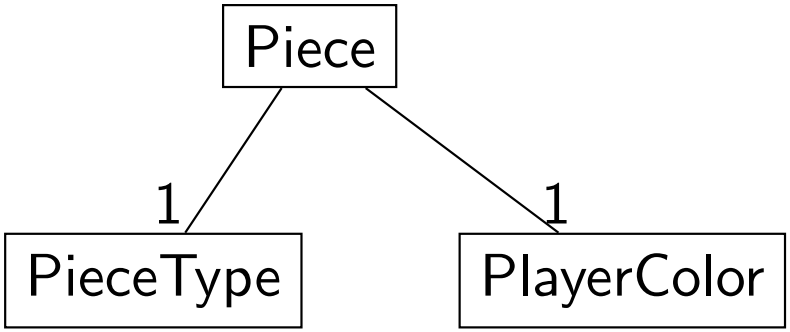
Spielfiguren



Die Klassen für Dreieck, Quadrat und Kreis werden nicht wirklich benötigt. Stattdessen können wir einen Aufzählungstyp benutzen. Auch für Größe und Farbe bieten sich Aufzählungstypen an:

```
enum Shape { TRIANGLE, SQUARE, CIRCLE}
enum Size  { LARGE, SMALL }
enum Color { WHITE, BLACK }
```

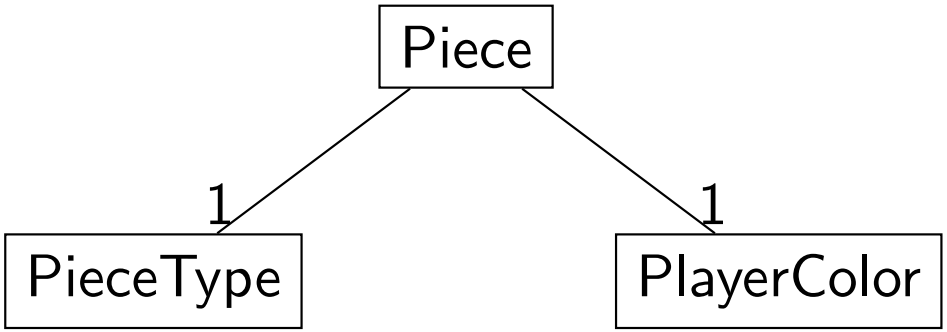
Spielfiguren – Implementierung



In unserer Referenzimplementierung haben wir Shape und Size zusammengefasst.

```
public enum PieceType {
    SMALLTRIANGLE, BIGTRIANGLE,
    SMALLSQUARE, BIGSQUARE,
    SMALLCIRCLE, BIGCIRCLE
}
public enum PlayerColor { WHITE, BLACK }
```

Spielfiguren – Implementierung



Die Klasse Piece enthält die beiden Aufzählungstypen als Komponenten.

```
public class Piece {
    PlayerColor player;
    PieceType pieceType;
    ...
}
```

Spielbrett

Ein Spielbrett muss sich die Figuren merken. Es gibt folgende Möglichkeiten.

- Spielbrett merkt sich eine Liste von Figuren, die Figuren merken sich ihre Position.

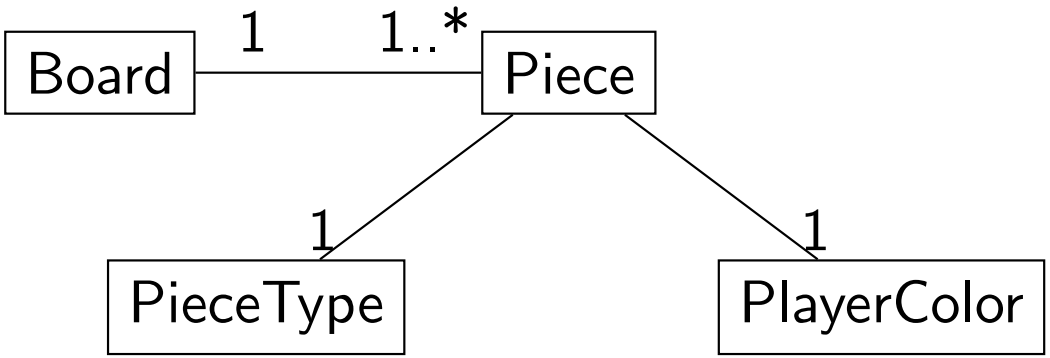
```
class Board { ArrayList<Piece> pieces; ...}
class Piece { int col, row;
    PieceType type; PlayerColor color; ...}
```

- Spielbrett merkt sich die Figuren in einem Array, wobei die Indices die Position angeben.

```
class Board { Piece[][] pieces; ...}
class Piece {
    PieceType type; PlayerColor color; ...}
```

Wir haben uns für die zweite Möglichkeit entschieden.

Spielbrett – Implementierung




```
class Board {Piece[][] pieces; ...}
class Piece {PieceType type; PlayerColor color;}
```

Was soll im Feld `pieces[r][c]` stehen, wenn an der Stelle `(r,c)` keine Figur steht?
Wieder zwei Möglichkeiten.

- Man benutzt die `null`-Referenz.
- Man baut eine Pseudofigur, die ein leeres Feld darstellt.

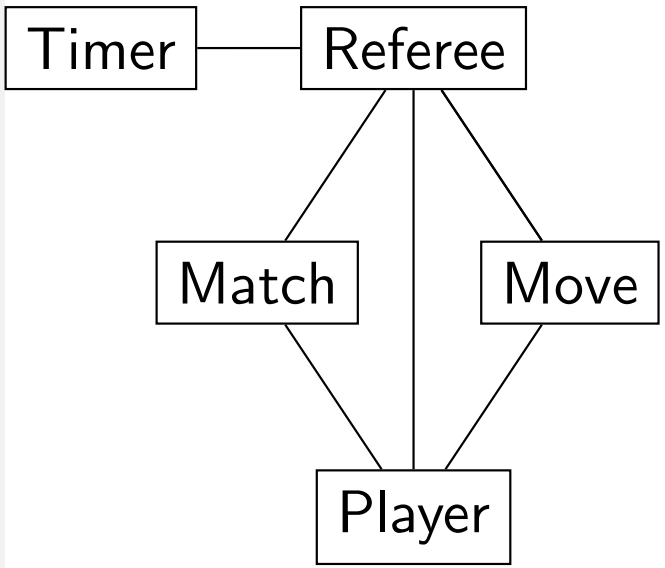
```
class NoPiece extends Piece {...}
```

Wir haben uns für die zweite Möglichkeit entschieden.

Spieler-Interface

Es sollen zwei Spieler in einer Partie gegeneinander antreten.
Wir benutzen ein Interface, um verschiedene Implementierungen eines Spielers zu erlauben. Das Interface muss es ermöglichen, dem Spieler seine Farbe mitzuteilen, die Züge abzufragen und die Züge des Gegners mitzuteilen. Außerdem möchten wir dem Spieler ein Schiedsrichter-Objekt übergeben, über das der Spieler an weitere Funktionalität kommt.

```
public interface Player {
    public void init(
        PlayerColor color,
        Referee referee);

    public Move getMove(
        Move opponentMove);
}
```



Designentscheidungen beim Spieler-Interface

Weil das Interface `Player` von allen Spielprogrammen benutzt werden muss, haben wir versucht die Abhängigkeiten klein zu halten.

- Das `Board` wird nicht von der Schnittstelle referenziert. Jeder darf seine eigene Implementierung für das Spielbrett benutzen.
- Stattdessen ist nur `Move` Teil der Schnittstelle.
- Die Spielerfarbe wird in der `init`-Methode mitgeteilt.
- Die gegnerischen Züge werden bei `getMove` übergeben.
- Es gibt ein Schiedsrichter-Interface über das der Spieler mehr Informationen (z.B. Rechenzeit) bekommen kann.

Die Zug-Klasse

Die Klasse `Move` gehört auch zur Schnittstelle. Deshalb haben wir sie einfach gehalten:

- Der Konstruktor `Move(int oldLine, int oldColumn, int newLine, int newColumn, boolean remis)` erzeugt einen neuen Zug.
  Wenn man Remis anbietet muss man trotzdem einen gültigen Zug mitgeben.
- `getOldLine()`..`getNewColumn()` liefern den entsprechenden Wert.
- `isOfferingRemis()` liefert true, wenn der Zug Remis anbietet.

Objekte der Klasse Move sind unveränderlich (immutable). Man kann die Werte nur im Konstruktor setzen, danach bleiben sie fest. Man kann auch von der Klasse `Move` nicht erben.

Schiedsrichter-Interface

`public interface Referee` stellt weitere Methoden für die Spieler zur Verfügung

- `getRemainingTime()` liefert die Rechenzeit, die das Programm (für alle Züge) noch zur Verfügung hat, zurück. Wir werden wohl zehn Minuten Rechenzeit erlauben.
- `getOpponentRemainingTime()` liefert die Rechenzeit, die das Programm (für alle Züge) zur Verfügung hat zurück.
- `isValid(Move move)` kann benutzt werden, um einen Zug auf Gültigkeit zu testen.
- `writeMessage(String msg)` kann für Debuggingzwecke benutzt werden.

Minimax-Algorithmus

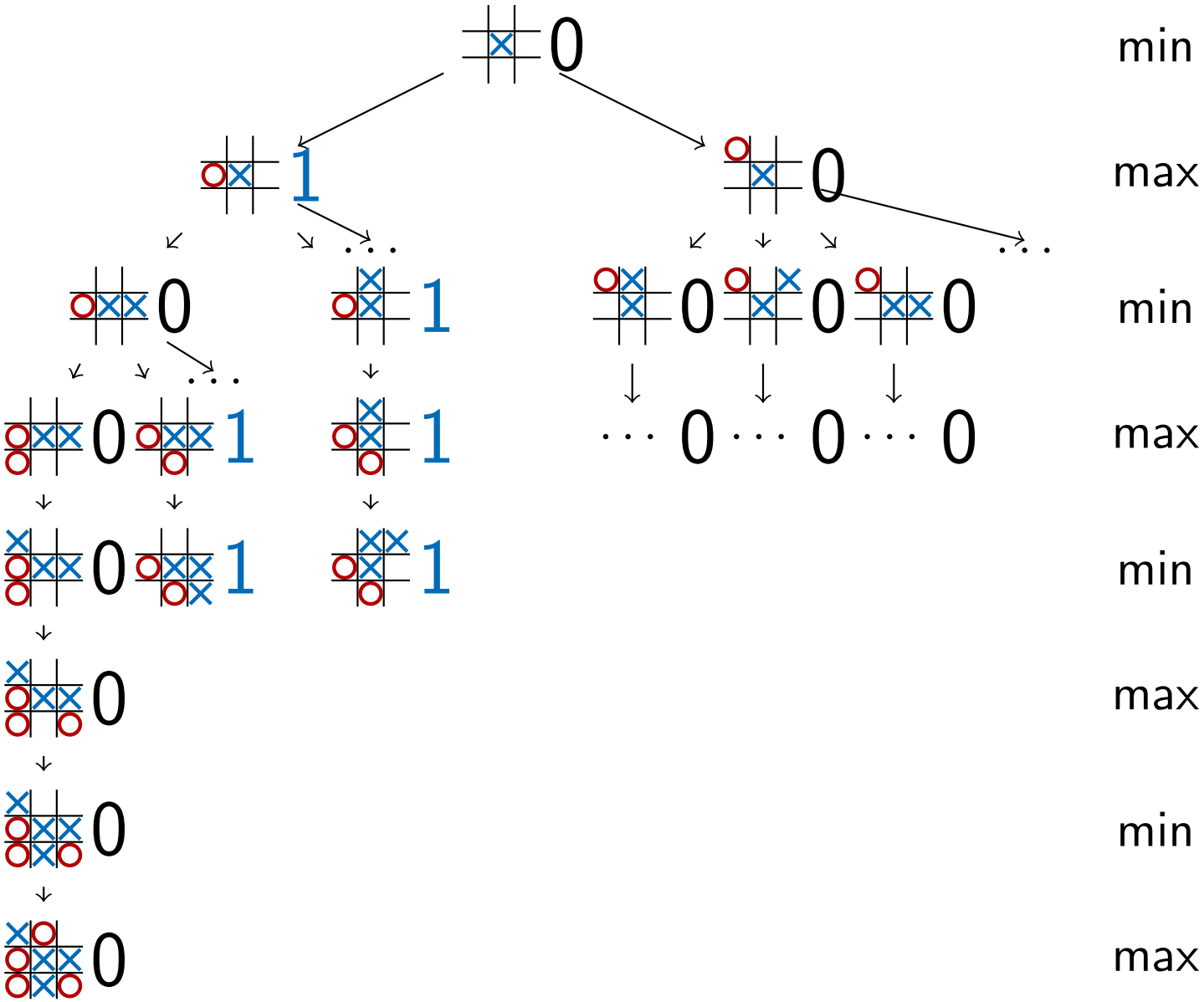
Der Minimax-Algorithmus

Der Minimax-Algorithmus eignet sich für Zwei-Personen-Nullsummenspiele mit perfekter Information (kein Zufall, keine geheimen Züge). Annahmen:

- Zwei Personen ziehen abwechselnd.
- Ein guter Zug für mich ist schlecht für den Gegner (Nullsumme)
- Der Gegner macht immer den für sich besten Zug.

Die Idee ist rekursiv die Züge zu verfolgen.

Beispiel: Tic-Tac-Toe



Pseudo-Code: Minimize (für Kreiszüge)

```
int minimize(Board board) {
  if (board.won(Player.X))
    return 1;
  if (board.isRemis())
    return 0;
  int minimum = 1;
  for (int i = 0; i < 9; i++) {
    Board newB = board.makeMove(Player.O, i));
    if (newB == null)
      continue;
    int value = maximize(newB);
    if (value < minimum)
      minimum = value;
  }
  return minimum;
}
```

Pseudo-Code: Maximize (für Kreuzzüge)

```
int maximize(Board board) {
  if (board.won(Player.O))
    return -1;
  if (board.isRemis())
    return 0;
  int maximum = -1;
  for (int i = 0; i < 9; i++) {
    Board newB = board.makeMove(Player.X, i));
    if (newB == null)
      continue;
    int value = minimize(newB)
    if (value > maximum)
      maximum = value;
  }
  return maximum;
}
```

Minimax mit Negation

Die beiden Funktionen sind sehr ähnlich.

- Negiere die Werte nach jedem Zug: Plus bedeutet aktiver Spieler steht gut, minus bedeutet Gegner steht gut.
- Dann genügt es nur noch zu maximieren.

Pseudo-Code: minimax

```
int minimax(Board board, Player me, Player him) {
  if (board.won(him))
    return -1;
  int maximum = -1;
  for (int i = 0; i < 9; i++) {
    Board newBoard = board.makeMove(me, i));
    if (newBoard == null)
      continue;
    int value = - minimax(newBoard, him, me);
    if (value > maximum)
      maximum = value;
  }
  return maximum;
}
```

Zugtiefe einschränken

- Bisher berechnen wir die Partie bis zum bitteren Ende.
- Für Alapo nicht möglich (jedenfalls nicht in 10 Minuten).
- Wir müssen die Suchtiefe beschränken.

```
int minimaxDepth(Board board, int depth,
                 Player me, Player him) {
  if (depth > MAX_DEPTH)
    return 0; // oder bessere Stellungsbewertung
  ...
  for (int i = 0; i < 9; i++) {
    ...
    int value = - minimax(..., depth + 1, ...);
    ...
```

Optimierungen

- Alpha-Beta-Pruning (siehe Wikipedia): Nicht alle Stellungen sind relevant.
- Zugvorsortierung (gute Züge zuerst probieren).
- Speichere berechnete Stellungen.
- Schrittweises Erhöhen der Zugtiefe.
- Zero-Window-Techniken
- Ruhesuche

Teil X

Softwareentwicklungstechniken – Testen und Debugging

Geschichte

Wikipedia:

> *Die Softwarekrise bezeichnet ein Mitte der 1960er-Jahre auftretendes Phänomen: Erstmal überstiegen die Kosten für die Software die Kosten für die Hardware. In der Folge kam es zu den ersten großen gescheiterten Software-Projekten.*

Techniken/Technologien um Softwareentwicklung zu unterstützen:

- Programmiersprachen
- Kooperationsmodelle für Softwareentwickler (Agile Methoden, ..)
- Debuggen
- Testen
- Verifikation
- ...

Hier: Praktische Einführung in Testen (Test-Driven-Development) und Print-Debugging.

Beispielaufgabe – Primzahltest

Aufgabe: Wir wollen ein Programm schreiben, das entscheidet, ob die eingegebene natürliche Zahl eine Primzahl ist.
Mögliche Eingaben: $[0, MAX\_LONG]$
Mögliche Ausgaben: $\{true, false\}$
Einige Testfälle:

Eingabe	erw. Ausgabe
0	false
1	false
2	true
3	true
4	false
91	false
97	true
100	false
541	true

Testfälle

Ein *Testfall* für eine Methode besteht im einfachsten Fall aus:

- einer Eingabe
- einem erwarteten Resultat

Weiter benötigt man Code, um alle vorhandenen Tests automatisch auf dem Programm ausführen und auswerten zu lassen, die *Testumgebung*.

Arten von Tests

Tests lassen sich einteilen nach der Granularität:

- Komponententest (auch: Unit-Test)
- Integrationstests
- Systemtest
- ...

... aber auch nach anderen Kriterien, z.B. statisch/dynamisch, Blackbox/Whitebox, Regressionstests usw. .

Blackbox Testing

Man versucht, ohne den Code zu betrachten, möglichst „gute“ Testfälle auszusuchen durch:

- Äquivalenzklassenbildung
- Grenzwertanalyse
- ...

Whitebox Testing

Man benutzt den Quellcode.
Man misst die Güte der Testmenge z.B. an

- Pfadüberdeckung
- Anweisungsüberdeckung
- ...

Grundsätze des Testens

(laut Spillner et al.)

- Testen zeigt die Anwesenheit von Fehlern
- Vollständiges Testen ist nicht möglich
- Mit dem Testen frühzeitig beginnen
- Häufung von Fehlern
- Zuhnemende Testresistenz
- Testen ist abhängig vom Umfeld
- Trugschluss: Keine Fehler bedeutet ein brauchbares System

Debugging

Bug nennt man einen Fehler im Programm.

Debugging bezeichnet die Tätigkeit, Bugs zu suchen und zu beheben.

Debugging kann sehr aufwendig sein, es gibt aber hilfreiche Tools.

Debugging – Techniken und Hilfsmittel

- Print Debugging
- Debugger – in IDE oder an der Kommandozeile
- Reverse Debugging
- Profiler
- Delta-Debugger
- ...

Beispiel: Suchbäume als Lösung des Wörterbuchproblems

UNI FREIBURG

Definition

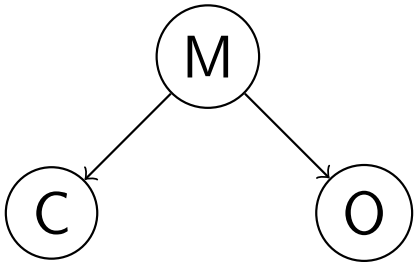
Das Wörterbuchproblem:
Verwalte eine Menge von Einträgen. Jeder Eintrag hat einen Schlüssel, anhand dessen er gefunden werden kann.

Gefordert sind die Operationen

- search(x) Liefere Eintrag mit Schlüssel x zurück, falls vorhanden
- insert(x) Füge Eintrag mit Schlüssel x hinzu
- delete(x) Lösche Eintrag mit Schlüssel x

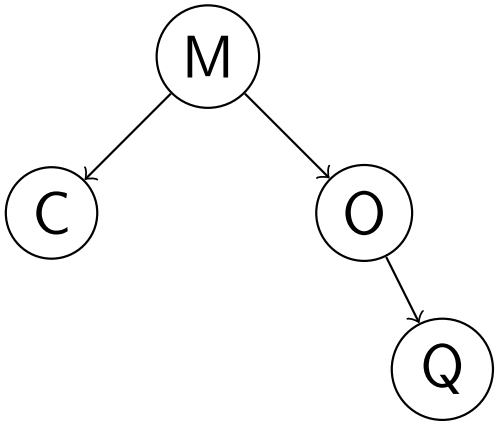
Effiziente Lösung?
(Balancierte) Binärsuchbäume

Binärsuchbäume



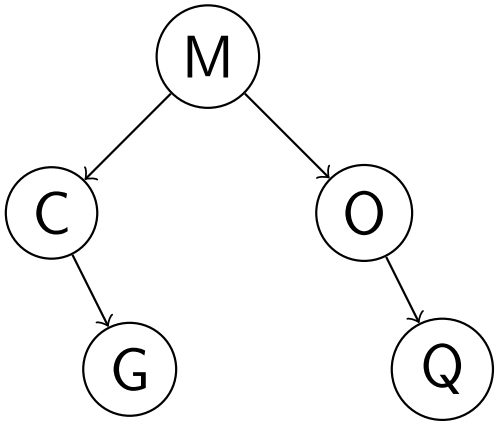

```
t = new Bintree(M)
t.insert(C)
t.insert(O)
t.insert(Q)
t.insert(G)
t.insert(D)
t.insert(E)
t.delete(C)
```

Binärsuchbäume



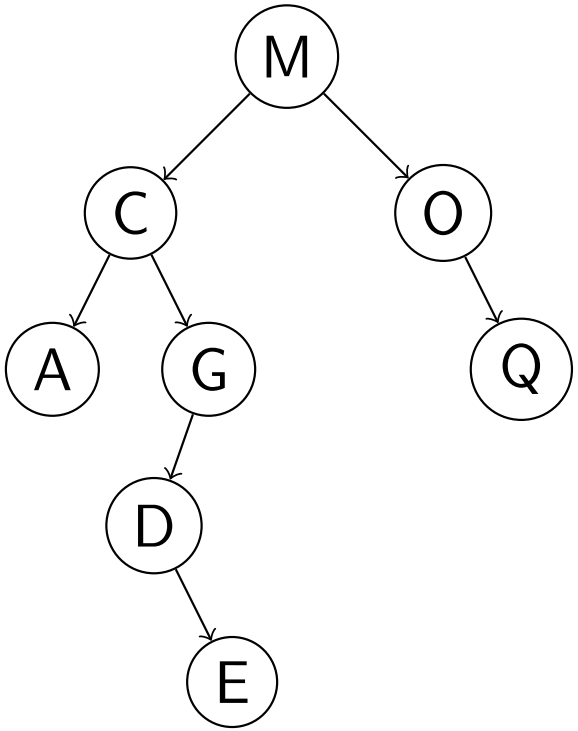

```
t = new Bintree(M)
t.insert(C)
t.insert(O)
t.insert(Q)
t.insert(G)
t.insert(D)
t.insert(E)
t.delete(C)
```

Binärsuchbäume



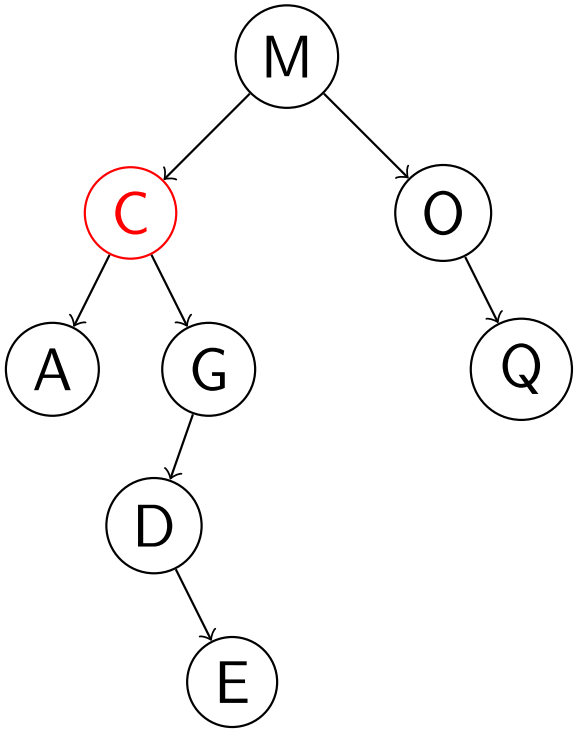

```
t = new Bintree(M)
t.insert(C)
t.insert(O)
t.insert(Q)
t.insert(G)
t.insert(D)
t.insert(E)
t.delete(C)
```

Binärsuchbäume



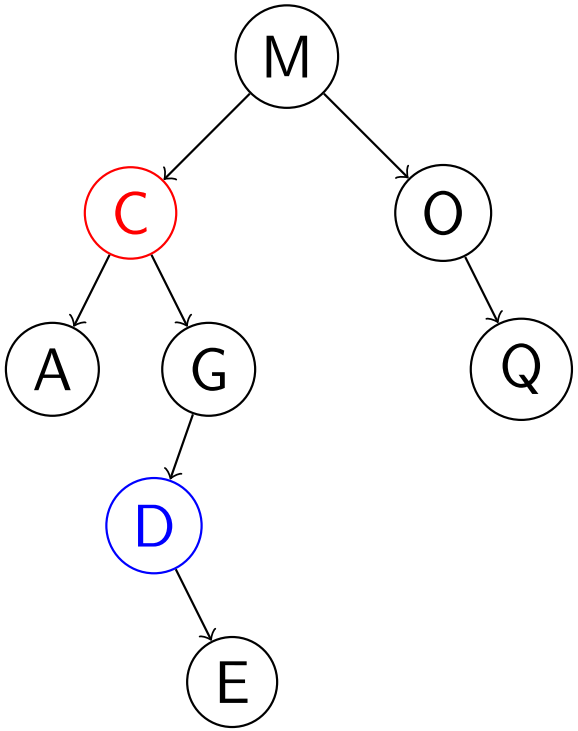

```
t = new Bintree(M)
t.insert(C)
t.insert(O)
t.insert(Q)
t.insert(G)
t.insert(D)
t.insert(E)
t.delete(C)
```

Binärsuchbäume



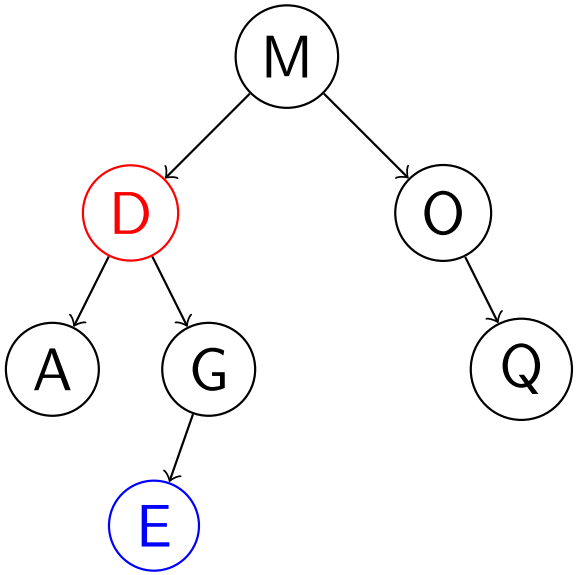

```
t = new Bintree(M)
t.insert(C)
t.insert(O)
t.insert(Q)
t.insert(G)
t.insert(D)
t.insert(E)
t.delete(C)
```

Binärsuchbäume




```
t = new Bintree(M)
t.insert(C)
t.insert(O)
t.insert(Q)
t.insert(G)
t.insert(D)
t.insert(E)
t.delete(C)
```

Binärsuchbäume




```
t = new Bintree(M)
t.insert(C)
t.insert(O)
t.insert(Q)
t.insert(G)
t.insert(D)
t.insert(E)
t.delete(C)
```

Quellen

1. Wikipedia, Artikel *Softwarekrise*, Stand 15.7.2014
2. Spillner, A., Linz, T., *Basiswissen Softwaretest*, korr. Nachdruck d. 3. Aufl. Heidelberg 2007.

Teil XI

Komplexität

Komplexität

Arten von Komplexität:

- Zeitkomplexität: Wie lange dauert es, ein Programm auszuführen?
- Platzkomplexität: Wie viel Speicherplatz benötigt ein Programm bei seiner Ausführung?

Wir beschränken uns hier auf Zeitkomplexität.

Laufzeitbestimmung – Zeit stoppen

```
(..)
    long time = System.nanoTime();
    containsElement(l, i);
    System.out.println(System.nanoTime() - time);
(..)
static boolean containsElement(int[] l, int i) {
  for (int element : l)
    if (element == i)
      return true;
  return false;
}
```

Art/Größe d. Eingabe	1000	10000	100000	1000000	...
Element am Anfang					
Element in der Mitte					
Element am Ende					

Laufzeitbestimmung – Probleme

Probleme bei der „Tabellenmethode“:

- Tabelle ist groß und unhandlich
- Die Messwerte sind abhängig von Ausführungsumgebung, zB.
  - Hardware (vgl. Moore's Law)
  - Programmiersprache/Interpreter/Compiler/...
  - ...
- Maß für abstrakte Algorithmen gewünscht

Lösung:

- Abstrahiere Eingaben zu ihrer Größe/Länge
- Funktion von Eingabe nach Laufzeit
- Gib Laufzeit als „Anzahl der Ausführungschritte“ an
- Abstrahiere von der konkreten Schrittgröße

Verschiedene Analysen

Letzte Folie: Abstrahiere Eingaben zu ihrer Größe/Länge
Problem: unterschiedliche Eingaben gleicher Länge können zu unterschiedlichen Laufzeiten führen (zB. s.h. Tabelle/Experiment)
Verschiedene Analysen:

- Best Case Analyse
- Average Case Analyse
- Worst Case Analyse
- ...

Definition

Wir schreiben $f \in \mathcal{O}(g)$, wenn

$$\exists c \in \mathbb{R}_+ . \exists \varepsilon \in \mathbb{N} . \forall x > \varepsilon : |f(x)| \leq c \cdot |g(x)|$$

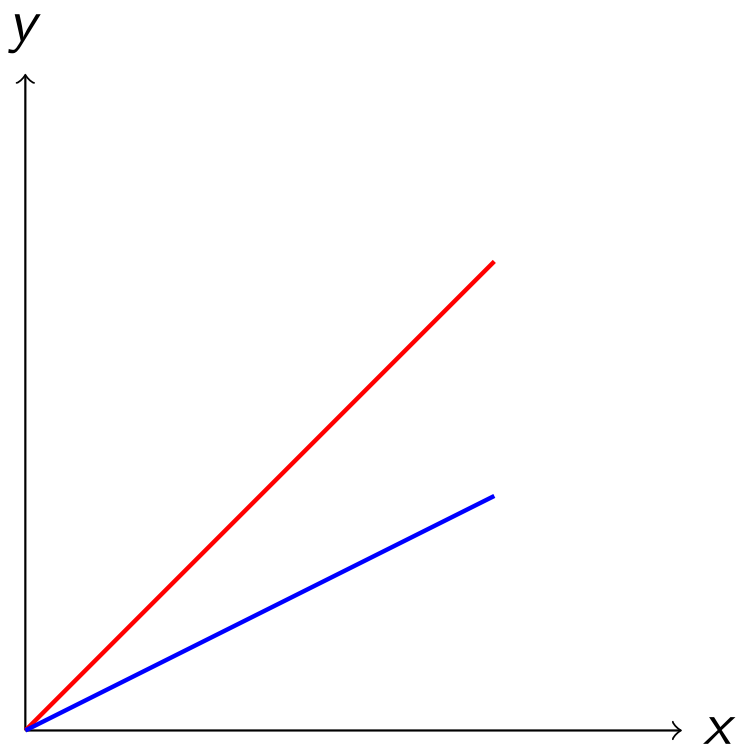
Intuitive Formulierung:
Ab einem bestimmten x-Wert und wenn man g beliebig mit einer Konstante multiplizieren kann, ist f immer durch g nach oben beschränkt.

Anmerkungen:

- Wir schreiben abkürzend $f \in \mathcal{O}(n)$ statt $f \in \mathcal{O}(g), g(n) = n$.
- Man schreibt manchmal auch $f = \mathcal{O}(g)$ statt $f \in \mathcal{O}(g)$.
- Es gibt auch entsprechende Definitionen für die untere Schranke.
- Wir versuchen immer die kleinste obere Schranke anzugeben.
- Man spricht auch von der *asymptotischen Laufzeit* eines Programms/Algorithmus'

Einige Funktionen

$$f \in \mathcal{O}(g) \text{ gdw. } \exists c \in \mathbb{R}_+ . \exists \varepsilon \in \mathbb{N} . \forall x > \varepsilon : |f(x)| \leq c \cdot |g(x)|$$

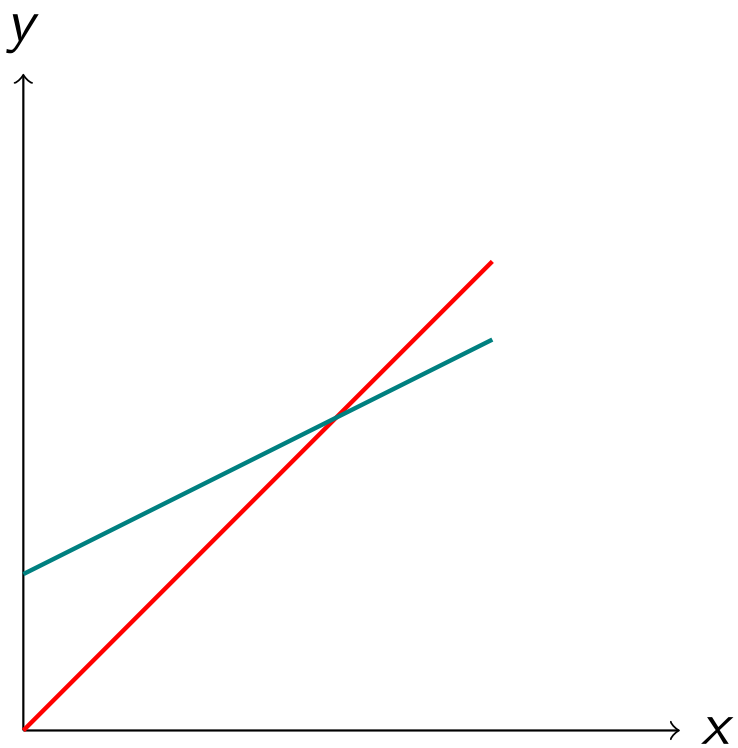


$g(x) = x$

$f_1(x) = \frac{1}{2}x$

Einige Funktionen

$$f \in \mathcal{O}(g) \text{ gdw. } \exists c \in \mathbb{R}_+ . \exists \varepsilon \in \mathbb{N} . \forall x > \varepsilon : |f(x)| \leq c \cdot |g(x)|$$

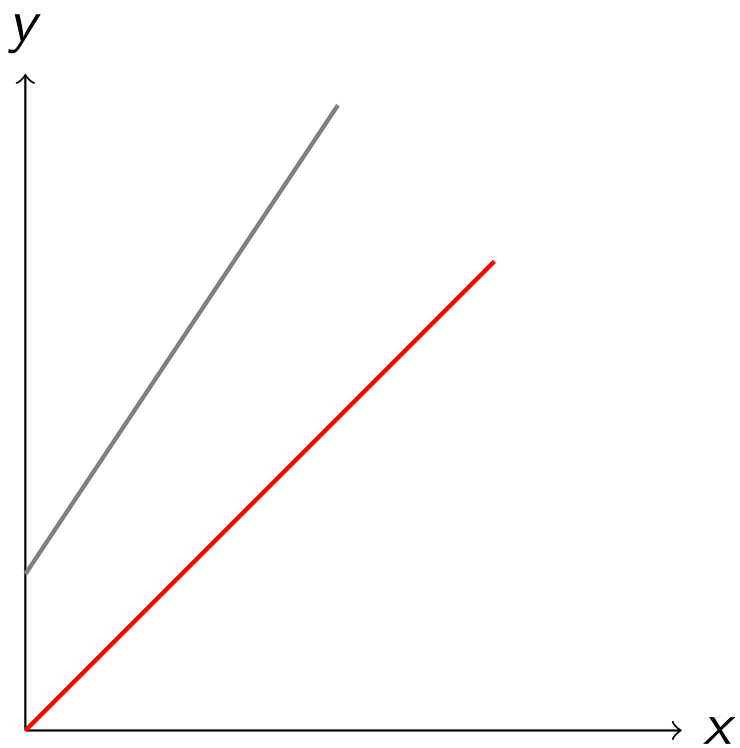


$$g(x) = x$$

$$f_2(x) = \frac{1}{2}x + 1$$

Einige Funktionen

$$f \in \mathcal{O}(g) \text{ gdw. } \exists c \in \mathbb{R}_+ . \exists \varepsilon \in \mathbb{N} . \forall x > \varepsilon : |f(x)| \leq c \cdot |g(x)|$$

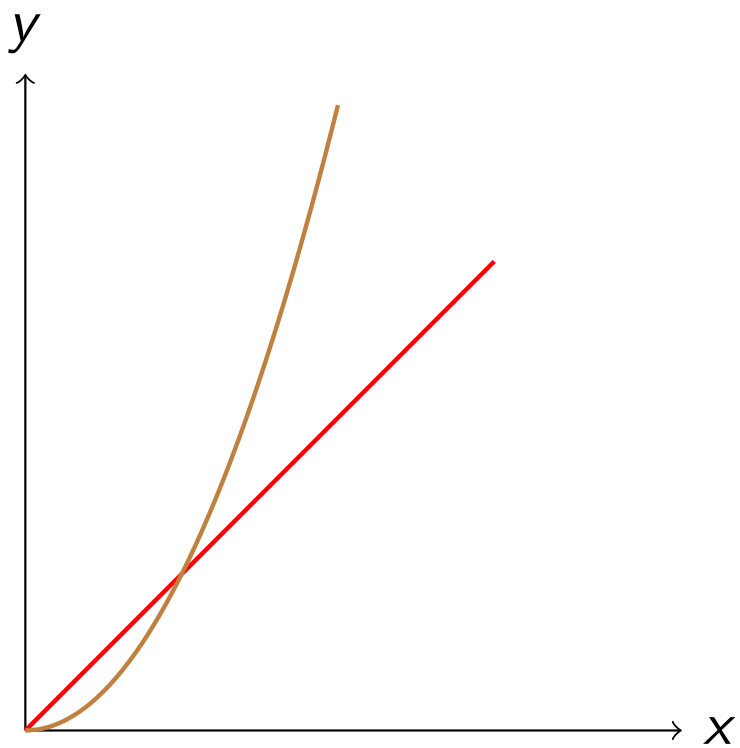


$$g(x) = x$$

$$f_3(x) = \frac{3}{2}x + 1$$

Einige Funktionen

$$f \in \mathcal{O}(g) \text{ gdw. } \exists c \in \mathbb{R}_+ . \exists \varepsilon \in \mathbb{N} . \forall x > \varepsilon : |f(x)| \leq c \cdot |g(x)|$$



$$g(x) = x$$

$$f_4(x) = x^2$$

Rechnung $f_3 \in \mathcal{O}(g)$

$$f_3(x) = \frac{3}{2}x + 1, g(x) = x$$

Zu zeigen:
$\exists c \in \mathbb{R}_+ . \exists \varepsilon \in \mathbb{N} . \forall x > \varepsilon : |f_3(x)| \leq c \cdot |g(x)|$

Wähle $c = 2$, dann ergibt sich als Schnittpunkt der beiden Geraden bei $x = 2$, wähle $\varepsilon = 2$.

Es gilt $\forall x > 2 . \frac{3}{2}x + 1 \leq 2x$.

□

Rechnung $f_4 \notin \mathcal{O}(g)$

$$f_4(x) = x^2, g(x) = x$$

Zu zeigen:

$$\neg(\exists c \in \mathbb{R}_+ . \exists \varepsilon \in \mathbb{N} . \forall x > \varepsilon : |f_4(x)| \leq c \cdot |g(x)|)$$
$$\Leftrightarrow \forall c \in \mathbb{R}_+ . \forall \varepsilon \in \mathbb{N} . \exists x > \varepsilon : |f_4(x)| > c \cdot |g(x)|$$

Wähle c und ε aus $\mathbb{R}_+$ ohne Einschränkung.

Dann ist zu zeigen:
Es existiert ein $x > \varepsilon$ sodass: $f(x) > c \cdot g(x)$, d.h. $x^2 > c \cdot x$.

Gesucht ist also ein $x \in \mathbb{R}_+$ sodass $x > c$ und $x > \varepsilon$.

So ein x existiert, z.B. $x = max(c, \varepsilon) + 1$.

□

Rechenbeispiele, Rechenregeln

Es gilt also zum Beispiel:

Sei $f(n) = m \cdot n + b$ für Konstanten m, b, dann: $f \in \mathcal{O}(n)$ und $f \notin \mathcal{O}(n^2)$

$\mathcal{O}(g) = \mathcal{O}(c \cdot g)$ für beliebige Konstante c, und beliebige Funktion g

$\mathcal{O}(\log_2 n) = \mathcal{O}(\log_c n)$ für beliebige Konstanten c

$f \in \mathcal{O}(n^k)$, falls f ein Polynom vom Grad k ist.

Rechenregeln:

$\mathcal{O}(f) + \mathcal{O}(g) = max(\mathcal{O}(f), \mathcal{O}(g))$

$\mathcal{O}(f) \cdot \mathcal{O}(g) = \mathcal{O}(f \cdot g)$

Gängige Komplexitätsklassen

Klasse	Umgangssprachlich	Beispielprobleme/Anmerkungen
$\mathcal{O}(1)$	Konstantzeit	erstes Element einer Liste ausgeben
$\mathcal{O}(\log(n))$	logarithmische Zeit	in einer sortierten Liste suchen
$\mathcal{O}(n)$	Linearzeit	in unsortierter Liste suchen
$\mathcal{O}(n \cdot log(n))$	„n-log-n“	Liste sortieren
$\mathcal{O}(n^2)$	quadratische Zeit	Zwei Summen ausmultiplizieren
$\mathcal{O}(n^3)$	kubische Zeit	Matrizenmultiplikation
$\mathcal{O}(n^k)$	Polynomialzeit	(jedes k ergibt eine eigene Ordnung)
$\mathcal{O}(k^n)$	Exponentialzeit	Potenzmenge berechnen
	exotischere Klassen	
$\mathcal{O}(n!)$	Alle möglichen Reihenfolgen n Städte zu besuchen.	
$\mathcal{O}(n^n)$		
...		

Beispiel: Element in unsortierter Liste suchen

Gegeben: Liste l von `int`s
Gefragt: Ist `int` i in l enthalten?

```
boolean containsElement(int[] l, int i) {
  for (int element : l) {
    if (element == i)
      return true;
  }
  return false;
}
```

Was können wir über die Laufzeitfunktion $f(n)$ in Abhängigkeit von `n=l.length` aussagen?
Best Case: $f(n) = 1 \in \mathcal{O}(1)$
Average Case: $f(n) = n/2 \in \mathcal{O}(n)$
Worst Case: $f(n) = n \in \mathcal{O}(n)$

Beispiel: In sortierter Liste suchen (binary search)

Gegeben: Sortiertes Array `l`, Zahl `i`, Array-Indizes `index1`, `index2`

Gefragt: Ist `i` in `l` zwischen den Indizes `index1` und `index2` enthalten?

```
boolean contains(int[] l, int i, int index1, int
    index2) {
  if (index1 > index2)
    return false
  int mid = (index2 + index1)/2;
  if (l[mid] == i)
    return true;
  else if (i < l[mid])
    return contains(l, i, index1, mid - 1);
  else
    return contains(l, i, mid + 1, index2);
}
```

Laufzeitanalyse binary search - Beispieldurchlauf

Input: sortiertes Array l = [1, 3, 14, 47, 94, 234, 384, 833, 902] gesuchtes Element: 234

$$\text{index1} = 0,\ \text{index2} = 8,\ \text{mid} = 4,\ \text{l[mid]} = 94$$
$$234 > 94 \implies \text{betrachte nur noch die rechte Hälfte}$$

$$\text{index1} = 5,\ \text{index2} = 8,\ \text{mid} = 6,\ \text{l[mid]} = 384$$
$$234 < 384 \implies \text{betrachte nur noch die linke Hälfte}$$

$$\text{index1} = 5,\ \text{index2} = 6,\ \text{mid} = 5,\ \text{l[mid]} = 234$$
$$234 == 234 \implies \text{liefere } \texttt{true} \text{ zurück}$$

Best Case (das gesuchte Element ist in der Mitte der Liste): $\mathcal{O}(1)$
Worst Case: $\mathcal{O}(\log n)$

Beispiel: Methodenaufrufe

```
static int mult(int m, int n) {
  int result = 0;
  for (int i = 0; i < n; i++)
    result = plus(result, m);
  return result;
}
static int plus(int m, int n) {
  int result = m;
  for (int i = 0; i < n; i++)
    result = inc(result);
  return result;
}
static int inc(int n) {
  return ++n;
}
```

Beispiel: Fibonacci

```
long fiboRek(int n) {
  if (n == 0 || n == 1)
    return n;
  return fiboRek(n - 1) + fiboRek(n - 2);
}
long fiboIt(int n) {
  if (n == 0 || n == 1)
    return n;
  long curNMinus1 = 1, curNMinus2 = 1, curN = 0;
  for (int i = 2; i < n ; i++) {
    curN = curNMinusOne + curNMinusTwo;
    curNMinusTwo = curNMinusOne;
    curNMinusOne = curN;
  }
  return curN;
}
```

Zusammenfassung, Caveats, $\mathcal{O}$-Notation

Die $\mathcal{O}$-Notation

- ist der übliche Weg, um in der Informatik die Komplexität eines Algorithmus' oder eines Problems anzugeben.
- ignoriert multiplikative Konstanten und betrachtet die Laufzeit asymptotisch, d.h. für Eingabegrößen, die gegen ∞ gehen.
- ist nützlich, um eine grobe, plattformunabhängige Laufzeitabschätzung für ein Programm zu geben.
- ist eine Grundlage der Komplexitätstheorie
- ist aber *nicht* immer aussagekräftig im Hinblick auf das tatsächliche Laufzeitverhalten von Programmen – Konstanten können groß werden.

Problemklassen

Im Folgenden betrachten Klassen von Problemen. Gängige Probleme sind zum Beispiel:

1. Enthält eine Sortierte Liste ein bestimmtes Element?
2. Sortiere diese Liste.
3. Gib die kürzeste Route an, die diese Städte besucht.
4. Gibt es eine Route, die diese Städte besucht, und die kürzer als x km ist?
5. Entspricht dieses Program dieser Spezifikation?
6. Läuft dieses Programm unendlich lange?

Probleme 1, 4, 5 und 6 sind *Entscheidungsprobleme*, d.h. Probleme, die ja oder nein zur Antwort haben.

Entscheidungsprobleme

Entscheidunsprobleme sind solche, die „ja“ oder „nein“ zur Antwort haben.

Man kann ein Entscheidungsproblem somit als Menge aller seiner Inputs schreiben, für die die Antwort „ja“ ist.

Zum Beispiel sei das Problem „Gegeben eine Ganzzahl zwischen 0 und 10, ist diese Zahl gerade?“, wir nennen es im Folgenden X.

Dann ist seine Eingabemenge $\mathit{Input}(X) = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.
Und die Menge, die durch das Problem gegeben ist, ist:
$\{x \in \mathit{Input}(X) \mid x \text{ ist gerade}\} = \{0, 2, 4, 6, 8, 10\}$

Die Klassen P und NP

Falls die Laufzeit eines Programms in $\mathcal{O}(n^k)$ liegt, für ein $k \in \mathbb{N}$, sagen wir, es läuft in *Polynomzeit*.

P: Alle Entscheidungsprobleme, die in Polynomzeit lösbar sind.
Beispiele:

NP: Alle Entscheidungsprobleme, bei denen eine -Lösung- in Polynomzeit verifiziert werden kann.
Also Entscheidungsprobleme von der Form „Hat ... eine Lösung“, bei denen man, wenn man einen Lösungsvorschlag hat, in Polynomzeit prüfen kann, ob dieser eine tatsächliche Lösung ist.
Beispiele:

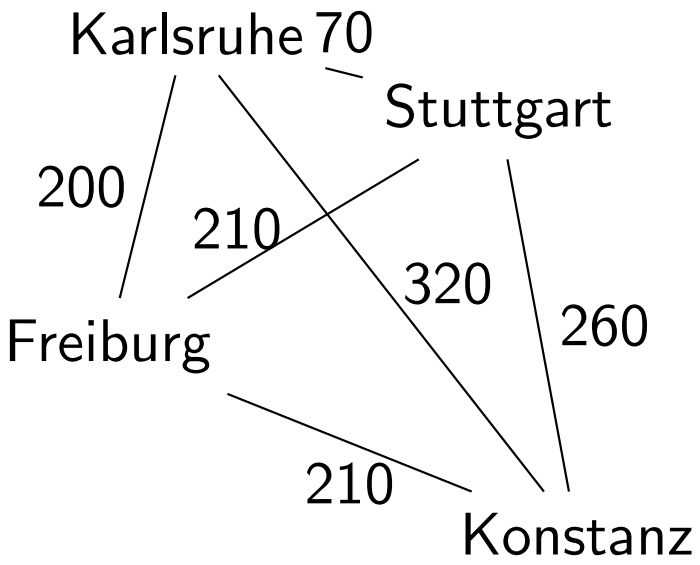
Das Handlungsreisendenproblem (TSP) (als Entscheidungsproblem)

Gegeben: Ein vollständiger Graph mit gewichteten Kanten $G = (V, E, w), E \subseteq V \times V, w : E \to \mathbb{N}$, der Städte und ihre Abstände untereinander (in km) darstellt, sowie eine Kilometerzahl k, die angibt, wie weit man maximal zu fahren bereit ist.

Frage: Gibt es eine Reihenfolge, in der man alle Städte nacheinander abfahren kann, und am Ausgangspunkt ankommen, ohne die gegebene Kilometerzahl zu überschreiten?

Das Handlungsreisendenproblem als Menge:
$\{(G, k) \mid G$ ist ein gewicheteter Graph, k eine Zahl, und G hat eine Rundreise deren Kantensumme kleiner gleich k ist$\}$

Das Handlungsreisendenproblem (TSP) – Beispiel



$k = 750$

TSP $\in$ NP

Behauptung: Das Handlungsreisendenproblem ist in NP.

Nachweis:
Raten: Rate eine Reihenfolge, in der die Städte abgefahren werden sollen.
Verifizieren: Summiere die entsprechenden Kantengewichte und vergleiche die Summe mit k.

Tatsächlich ist TSP sogar NP-vollständig.

Definition

Ein Problem ist *NP-vollständig*, wenn

1. es in NP liegt
2. es *NP-schwierig* ist, d.h. sich jedes Problem in NP darauf in Polynomzeit reduzieren lässt (s. nächste Folie)

Polynomzeitreduktion

UNI FREIBURG

Definition

Seien X, Y zwei Probleme (formuliert als Mengen). Wir sagen X *lässt sich auf Y in Polynomzeit reduzieren*, wenn es eine in Polynomzeit berechenbare Funktion $f : \mathit{Input}(X) \to \mathit{Input}(Y)$ gibt, so dass für jedes $x \in \mathit{Input}(X)$ gilt:

$x \in X$ gdw. $f(x) \in Y$

Wir schreiben dann $X \leq_P Y$.

Wesentliche Folgerung:

Falls gilt $X \leq_P Y$ und Y in P, dann ist auch X in P.

NP-vollständige Probleme

SAT $\{\phi \mid \phi$ist eine erfüllbare aussagenlogische Formel$\}$

HAMPATH $\{G \mid G$ ist ein Graph, in dem es einen Pfad gibt, der jeden Knoten genau einmal besucht$\}$

Graph Färbung $\{(G, k) \mid G = (V, E)$ ist ein Graph, der mit k Farben, so gefärbt werden kann (die Knoten), dass benachbarte Knoten nie die selbe Farbe haben$\}$

.. und viele mehr

P vs. NP

Stand der Forschung:

Man weiß nicht sicher, ob P und NP zusammenfallen, oder nicht.

Bisher sind alle Beweisversuche für die Ungleichheit gescheitert.

Es hat aber auch niemand einen Polynomzeitalgorithmus für ein NP-vollständiges Problem gefunden.

Die Frage ist eines der Milleniumsprobleme.

Falls P=NP: möglicherweise technische Revolution, bisher starke Kryptografie wird schwach, ...

Falls P!=NP: Es würde sich nicht viel ändern, die meisten rechnen damit jetzt schon.

Ausblick: Berechenbarkeitstheorie

Die NP-vollständigen Probleme sind keineswegs die schwierigsten Probleme der Informatik.
Es gibt Probleme, die sicher exponentielle Zeit brauchen.
Es gibt Probleme, die überhaupt nicht lösbar sind.
Die Berechenbarkeitstheorie fragt nicht nach der Komplexität eines Problems, sondern ob es überhaupt in endlicher Zeit von einem Computer lösbar ist.

Zusammenfassung, P vs. NP

- NP ist die Klasse von Problemen, die in Polynomzeit (d.h. meist effizient) verifizierbar sind.
- NP-vollständige Probleme sind solche, die in NP liegen, und die NP-schwierig sind.
- Kann man ein NP-schwieriges Problem in Polynomzeit lösen, so kann man alle Probleme in NP in Polynomzeit lösen, somit P=NP.

Teil XII

Überblick/Zusammenfassung

Vorlesung 1 – Einführung, Struktur von Java Programmen

UNI FREIBURG

- Java – erste Einführung
  - Schreiben, kompilieren, ausführen
  - Variablen – Funktion, Deklaration, Benutzung
  - Zuweisungen – via „=“
  - Ausgaben auf der Textkonsole via
    `void System.out.println(String s)`
- Lexikographische Struktur
  - Bezeichner, Literale, Schlüsselwörter
  - Kommentare
- Programmstruktur in Java
  - Klassen, Methoden
  - Packages, imports
  - Die `main`-Methode
  - Coding Conventions und JavaDoc

Vorlesung 2 – Ausdrücke und Anweisungen

UNI FREIBURG

- Ausdrücke
  - Ausdrücke
  - Typen: int, float, String, boolean
  - Typkonvertierung
  - Operatoren
- Anweisungen
  - Zuweisungen
  - Variablendeklarationen
  - Methodenaufrufe
  - Blöcke
  - Verzweigungen
  - Schleifen

Vorlesung 3 – Methoden, Referenzdatentypen

UNI FREIBURG

- Methoden
  - Syntax von Methodenaufrufen
  - Parameter und Rückgabewert
  - Call by Value
  - Rekursion
- Referenzdatentypen
  - Arrays – Initialisierung, Zugriff
  - Klassen – Felder, Feldzugriff

Vorlesung 4 – Objektorientierung (1)

- Grundideen:
  - Generalisierung
  - Vererbung
  - Kapselung
  - Polymorphismus
- Klassen
- Klassendiagramme
- Objektdiagramme
- Klassen in Java
  - Instanzen, Instanzmethoden, Felder
  - statische Methoden und Felder

Vorlesung 5 – Objektorientierung (2)

- Klassen in Java
  - Konstruktoren
  - Vererbung in Java
  - Typhierarchie, Typkonvertierung
  - Polymorphie, Overriding
  - Interfaces, `abstract`

Vorlesung 6 – Objektorientierung (2)

- Code Konventionen
- Aufzählungstypen (`enums`)
- Exceptions

Vorlesung 7 – Programmieren und Datenstrukturen

UNI FREIBURG

- Algorithmen
  - Definition
  - Eigenschaften
- Datenstrukturen
  - Verkettete Listen (einfach, doppelt)

Vorlesung 8/9/10 – Generics, Bibliothek, Alapo

UNI FREIBURG

- Generische Typen (Generics)
- Das `Collection` Interface
- Die Interfaces `Comparator`, `Comparable`
- Die Standard-Implementierungen der verschiedenen Collections
- Alapo – Vorstellung

Vorlesung 11 – Testen

UNI FREIBURG

- Definition Test
- Arten von Tests
- Kriterien für Testmengen
- Einführung in das Debugging

Vorlesung 12 – Komplexität

- Die $\mathcal{O}$-Notation
  - Motivation
  - Definition
  - Verwendung
- P vs NP