Stand: 3.12.2007

Einführung in die Informatik III

(für Studierende des 3. Fachsemesters)

Pflichtvorlesung für die Diplomstudiengänge "Informatik" und "Automatisierungstechnik in der Produktion" sowie wählbare Vorlesung für weitere Studiengänge, z. B. für Mathematik, Softwaretechnik, Lehramt Informatik usw.

Universität Stuttgart, Wintersemester 2007/2008

Dozent: Volker Claus

Bitte beachten Sie: Für die Richtigkeit der Inhalte und insbesondere der Programme wird keine Garantie übernommen.

Gliederung

0. Vorbemerkungen, Voraussetzungen
1. Objektorientierte Programmierung
2. Funktionales Programmieren
3. Maschinennahe (abstrakte) Programme
4. Nebenläufigkeit, S/T-Netze

Einordnung und Inhalte

In den Vorlesungen "Einführung in die Informatik I und II" wurde der Rohstoff „Information“ und seine Darstellung, Verarbeitung, Speicherung und Übertragung zusammen mit den Grundlagen, um Informationsprozesse durchführen zu können, vorgestellt. Im Vordergrund stand eine „sequenzielle imperative Denkweise“, d. h., Variablen werden als Behälter mit erlaubten Werten, Operatoren und einem Verfallsdatum aufgefasst, auf denen Algorithmen schrittweise, einzeln und nacheinander mit Hilfe nur eines Prozessors arbeiten. Datentypen können durch gängige Operationen zu sehr komplexen Gebilden zusammengesetzt werden, Verweise auf zugehörige Behälter, aber auch auf andere Programmeinheiten sind zulässig, die Zusammenfassung zu einem Modul (Paket), bestehend aus Spezifikation und Implementation, und deren Archivierung und Wiederverwendung erleichtert die Erstellung neuer Programme. Hinzu kamen Syntax-, Komplexitäts- und Verifikationsfragen.

In der hieran anschließenden Veranstaltung „Einführung in die Informatik III“ geht es um die Vermittlung andersartiger Informatik-Denkweisen "Paradigmen"), konkret um

- objektorientiertes Programmieren,
- funktionales Programmieren,
- systemnahes Programmieren,
- Nebenläufigkeit.

Hierzu werden neben den notwendigen Grundlagen neue Sprachen und Kalküle eingeübt und zwar Java, Scheme, abstrakter Assembler und gemeinsame Variable bzw. Kommunikationskanäle.

Rahmen der gesamten Veranstaltung

Die Veranstaltung dauert 16 Wochen (in der Zeit vom 15.10.07 bis 12.2.08). Sie besteht aus (eine „Vorlesungs- oder Übungsstunde“ = 45 Zeit-Minuten):
Vorlesung: 3 Vorlesungsstunden pro Woche (insgesamt 48 Vorlesungsstunden),
Übungen: 2 Übungsstunden pro Woche (insgesamt 30 Übungsstunden).
In den Übungen sind Programme in den Sprachen Ada, Java und Scheme anzufertigen.
Die Zusatzveranstaltungen und begleitende Programmierübungen, die Sie aus dem ersten Studienjahr gewohnt waren, werden nicht mehr fortgesetzt. Wir erwarten, dass Sie evtl. fehlendes Wissen und fehlende Fertigkeiten selbst ausgleichen werden.

Zeit und Ort im Wintersemester 2007/2008:
Vorlesung: Montag und Dienstag 14:00 bis 15:30 Uhr, Hörsaal 38.01, manchmal 14-tägige Termine, Details: siehe zusätzlich verteilte Informationen oder Web-Seite.
Dozent: Prof. Dr. Volker Claus.
Übungen und Programmierübungen: Je nach Übungsgruppe. Wöchentlich (14 Termine und ein Zusatztermin im Oktober). Es gibt zwei zur Übung zählende Zwischenklausuren (3.12.07 und 22.01.08).
Betreuung der Übungen: Dipl.-Inf. Sascha Riexinger.
Klausur: geplant am Mittwoch, 5.3.08, vermutlich um 10 Uhr im großen Hörsaal 53.01.

Anforderungen an Sie (beachten Sie: *Anwesenheit reicht nicht!*)
Regelmäßige Mitarbeit, insbesondere Einüben und Festigen des Stoffs der Vorlesung durch Nacharbeit und Übungen. Studentischer Aufwand für die Lehrveranstaltung wöchentlich rund 11 Zeitstunden (5 SWS, ca. 7,5 ECTS-Punkte). Wird diese Zeit regelmäßig (!) in der Vorlesungszeit aufgewendet, so reichen weitere 40 Stunden zur Prüfungsvorbereitung aus. Gesamter Zeitaufwand (einschl. der Vorlesungs- und Übungsstunden): rund 215 Zeitstunden. (Messen Sie Ihre Zeiten!)
Die Teilnehmer(innen) sind aufgefordert, den Kontakt zu den Betreuern und zu den Tutor(inn)en zu suchen. Hierfür können z.B. E-Mails und die Sprechstunden und die direkte Ansprache in und nach den Lehrveranstaltungen genutzt werden.
Wir bemühen uns darum, alle Hörer(innen) auf eine erfolgreiche Prüfung vorzubereiten. Defizite zeigen sich meist schon frühzeitig, vor allem beim Abschneiden in den Übungen und in den Zwischenklausuren. Leider haben wir nicht genügend Personal, um auf jede(n) einzelne(n) einzugehen. Überwachen Sie sich daher verstärkt selbst, lassen Sie Ihr Studium nicht schleifen und gehen Sie **sofort** in die Sprechstunden des Mitarbeiters oder des Dozenten, sobald Ihre Leistungen nicht mehr über den Minimalanforderungen liegen, d. h., sobald Sie erkennen, dass Sie weniger als 50 % der Übungsaufgaben alleine lösen können. *Eine große Gefahr liegt im schleichenden Ausklinken aus den Veranstaltungen!* Denn der Stoff baut stets auf dem bereits Gelernten auf, und so steigern sich einzelne Versäumnisse schnell zu großen Wissenslücken.

Literatur:

Abelson, Harold, and Gerald J, Sussman with J. Sussman, “Structure and Interpretation of Computer Programms”, MIT, 2. Auflage, 1996 (es gibt auch eine deutsche Ausgabe)

Bishop, Judy, "Java lernen", Pearson-Studium, Addison-Wesley, 2. Auflage, 2003

Boles, D.: "Programmieren spielend gelernt mit dem Java-Hamster-Modell", Teubner-Verlag, Wiesbaden, 2006

Cousineau, Guy and Michel Mauny, „The Functional Approach to Programming“, Cambridge University Press, ISBN: 0521576814 (1998)

Goos, Gerhard und Wolf Zimmermann, „Vorlesungen über Informatik Band 1: Grundlagen und funktionales Programmieren“, Springer, ISBN: 3540244050 (2005)

Goos, Gerhard und Wolf Zimmermann, „Vorlesungen über Informatik Band 2: Objekt-orientiertes Programmieren und Algorithmen“, Springer, ISBN: 3540244034 (2006)

Heinisch, Cornelia, Müller-Hofmann, Frank, und Goll, Joachim, "Java als erste Programmiersprache", Teubner-Verlag, Wiesbaden, 2007

Nicholson, John, "Java Programming", http://cc.usu.edu/~jnicholson/CS3410-Fall2007/

Ottmann, T., und Widmayer, P., „Algorithmen und Datenstrukturen“, Spektrum Verlag, Heidelberg, 4. Auflage 2002

Scott, Michael L., „Programming Language Pragmatics“, Morgan Kaufmann Publishers, ISBN: 0126339511 (2000)

Thiermann, Peter, „Grundlagen der funktionalen Programmierung“, Teubner-Verlag, ISBN: 3519021374 (1994)

Wilson, Paul, "An introduction to Scheme and its Implementation", online-Buch: ftp://ftp.cs.utexas.edu/pub/garbage/cs345/schintro-v14/schintro_toc.html

Weitere Literatur wird bei Bedarf genannt.

Konkrete Wissens-Inhalte, die vermittelt werden:

In dieser Vorlesung werden unter anderem vermittelt:

Kalküle und Sprachen:

- Umgang mit der Programmiersprache Java
- Umgang mit der Programmiersprache Scheme
- Register- und Stackmaschinen
- Attributierte Grammatiken
- Stellen-Transitions-Netze

Algorithmen:

- Wiederholung: Baumsortieren und Sortieren durch Mischen
- Median in linearer Zeit
- Teilworterkennung (substring-problem)
- Anagramme
- Beliebige genaue Wurzelberechnung (mit Pellscher Gleichung)
- Schneeflocken-Zeichnungen (nach Koch)
- Peterson-Algorithmus (softwaremäßiger wechselseitiger Ausschluss)

0. Vorbemerkungen, Voraussetzungen

0.1 Hinweise, Voraussetzungen

0.2 Formalismen, Erinnerungen

0.3 Bemerkungen zur Softwareerstellung

0.1 Hinweise, Voraussetzungen

1. Sie finden die Folien der Vorlesung auf der Webseite, manchmal leider erst nach der jeweiligen Vorlesung.
2. Die Vorlesung erfolgt überwiegend an der Tafel. Die dort entwickelten Aussagen werden durch die Folien unterstützt.
3. Die Übungsaufgaben und ihre Lösungen gehören zum Stoff der Vorlesung dazu! Insbesondere sind sie auch Prüfungsstoff.
4. Programmieren in Ada wird erwartet. Wir werden zwar nur selten Programme in Ada schreiben, aber die Denkweise des imperativen Programmierens wird vorausgesetzt (und zwar deutlich mehr als Anfänger wissen).

0.2 Formalismen, Erinnerungen

1. Die Backus-Naur-Form (= BNF = kontextfrei) ist Ihnen bestens bekannt, Sie können sie lesen und Sie können hierin die Syntax einer Sprache formulieren.
2. Auch die Erweiterung EBNF kennen Sie (zusätzlich { }, [ ], ' ' und ggf. Unterstreichungen für Sonderwörter.
3. Alle elementaren Datentypen sind Ihnen geläufig einschl. der Unterschiede von z. B. short, integer, long, double usw. (oder entsprechende Formulierungen).
4. Sie kennen Strings, Felder, Records, Pakete (package, Modul), Generizität, Overloading, Exceptions, Blöcke, ...
5. Sie kennen diverse Beispiele für Algorithmen.

Sollten sie wenig Programmierpraxis haben, so nutzen Sie von Anfang an die Programmierung in Java (und später in Scheme), um sich ein "handwerkliches Fundament" zu erwerben, das später unverzichtbar ist.

Über die Rechnerlandschaft, Plattformen usw. werden wir kaum noch reden. Wenn Sie hier unsicher sind, so lassen Sie sich Vieles zeigen und vollziehen Sie es dann noch einmal ganz alleine nach!

Imperatives Programmieren bezieht sich meist auf eine virtuelle Maschine mit Speicherplätzen, die durch Indizes, Anordnungen oder Verzeigerung dynamisch miteinander verknüpft werden. Das "Wie" und die konkreten Abläufe spielen hier ein besondere Rolle. Sie sollten sich über das Modell, das Sie hierbei im Kopf haben, klar sein.

0.3 Ungeordnete Bemerkungen zur Softwareerstellung

a) Was ist Software?

b) Was ist Programmieren im Kleinen?

c) Was ist Programmieren im Großen?
(Komplex, viel Sichten, verzahnt, viele Dokumente, viele Ersteller, Architektur, Schichten/Hierarchien, ...)

d) Erstellung großer Software
Teams, Tools, Umgebungen, Kommunikation, Versionen, viele Abhängigkeiten $\Rightarrow$ Management

e) Einige Begriffe: verständliche Architektur, lesbare Teile, Gruppieren, Spezifizieren, Verifizieren, Testen und Reviews, Wartung und Wiederverwendung, Modelle, Schnittstellen, Interaktion, eingebettete Systeme, ...

f) Was ist Qualität der Software? (QoS) Bewertung, Oberflächen, Benutzungsschnittstellen, ...

g) Software-Entwicklungsphasen und ihre Besonderheiten

h) Echtzeitsysteme, Anwendungssysteme, Informationssyst.

i) Methoden (des Vorgehens, der Darstellung, der Analyse usw., ihre Wirksamkeit, ihre Unterstützung durch Tools, direkte Zugriffsmöglichkeiten, konsistente Dokumente usw.)

j) Qualität der Prozesse

k) Architekturen (frameworks, components, libraries, ...)

Siehe hierzu einschlägige Bücher und Vorlesungen!

1. Objektorientierte Programmierung (ooP)

1.1 Prinzipien

1.2 Java, Teil 1

1.3 Beispiele (Mischen, Median)

1.4 Java, Teil 2

1.5 Textverarbeitung: Erkennung von Teilwörtern

1.6 Java, Teil 3

1.7 Hinweise zu objektorientierten Sprachen

1.1 Prinzipien der objektorientierten Programmierung

(Zum Teil Wiederholung aus der Grundvorlesung:)
Klasse
Variablen/Zustand
Methoden
Instanzen
Initiierung, Konstruktor
Vererbung
Kommunikation
Parameter und Modifikatoren
(Klassen-) Bibliotheken

[Zu diesem Abschnitt 1.1 vergleiche Abschnitt 4.6 der Grundvorlesung!]

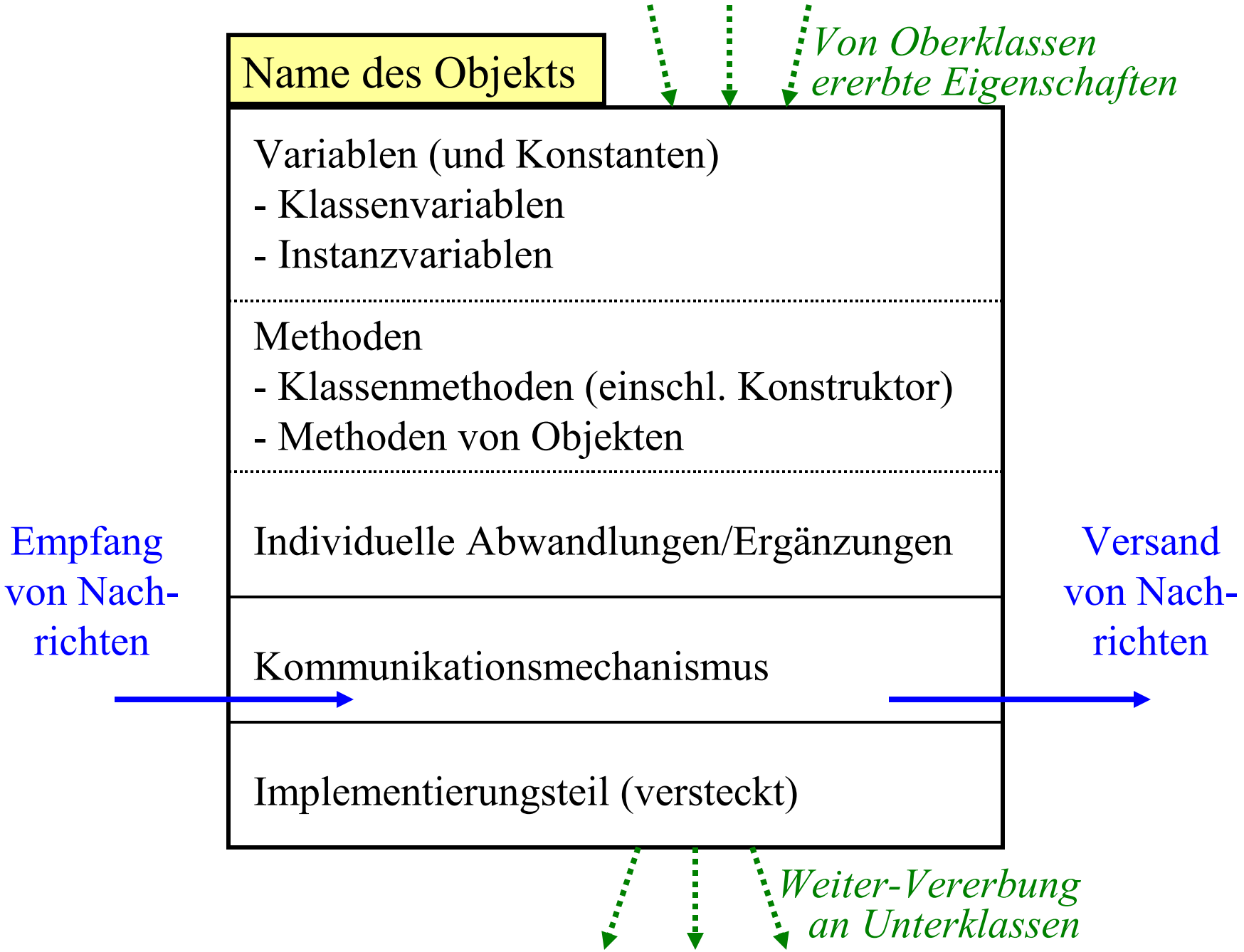
Der objektorientierte Ansatz geht auf Arbeiten aus dem Jahre 1969 von Alan Kay zurück, der (auch hierfür) 2003 den Turing Award erhielt. Erste Ideen finden sich bereits in der Programmiersprache SIMULA 65, die ein einfaches Klassenkonzept mit sog. Koroutinen besitzt. Der Prototyp der ooP ist Smalltalk 80, eine objektorientierte Sprache auf Basis funktionaler Programmierung, die in den 70er Jahren in der Firma Xerox entstand und später im Projekt 'Squeak' weiterentwickelt wurde. Als erste rein objektorientierte Sprache mit kommerziellem Einsatz gilt Eiffel (Bertrand Meyer, 1988). Als Weiterentwicklung der relativ maschinennahen Sprache C hat sich seit 1983 C++ einen großen Marktanteil erobert, den sie sich seit kurzem immer mehr mit den Nachfolgesprachen Java und C# teilt.

Grundidee der Objektorientierung ist es, funktionsfähige Einheiten zu kapseln; diese bieten Strukturen und Funktionalitäten an, von denen man aber nicht angibt, wie sie realisiert sind. Man trennt also das "Was" vom "Wie". Mit dieser Grundidee lassen sich vor allem sehr umfangreiche Systeme leichter und sicherer modellieren, entwickeln, zusammenfügen, implementieren, anpassen und pflegen.

Objekte sind in sich geschossene Einheiten, die

- wie Moduln aufgebaut sind: Es gibt ein Schema, genannt "Klasse", das vor allem aus "Attributen" (das sind die einzelnen Datenstrukturen der Variablen) und "Methoden" (das sind die algorithmischen Teile) besteht; diese werden mit Parametern und Modifikatoren versehen. Ein Objekt ist eine Instanz (oder ein "Exemplar" oder eine "Ausprägung") einer Klasse, das bei seiner Erzeugung ("new") durch einen Konstruktor iniziiert wird.
- einen individuellen Zustand besitzen (Speicherzustand der Klassen- und Instanzvariablen),
- miteinander kommunizieren können; dies geschieht durch Nachrichtenaustausch ("message passing"),
- durch Vererbung ihre Eigenschaften an neue Objekte bzw. Klassen weitergeben können.

Mit Hilfe vordefinierter Klassen können gewisse Bereiche (z. B. komplexe Zahlen) komplett beschrieben und verwendet werden.


Name des Objekts
Von Oberklassen ererbte Eigenschaften
Variablen (und Konstanten)
- Klassenvariablen
- Instanzvariablen
Methoden
- Klassenmethoden (einschl. Konstruktor)
- Methoden von Objekten
Individuelle Abwandlungen/Ergänzungen
Kommunikationsmechanismus
Implementierungsteil (versteckt)
Empfang von Nach-richten
Versand von Nach-richten
Weiter-Vererbung an Unterklassen

Prinzipien der objektorientierten Programmierung (ooP):

1. Es gibt nur Objekte. Jedes Objekt ist eindeutig identifizierbar über seinen Namen.
2. Alles wird über Klassen, Instanzbildung, Zustände, Methoden, Nachrichten und Vererbung realisiert.
3. Objekte handeln in eigener Verantwortung (und sie geben nur bekannt, *was* sie bearbeiten, niemals, *wie* sie dies tun).
4. Klassen werden in Bibliotheken aufbewahrt und stehen allen Programmen und Klassendefinitionen zur Verfügung.
5. Programmieren bedeutet, Klassen festzulegen, hieraus Objekte zu erzeugen und diesen Aufgaben zu übertragen, indem man ihnen geeignete Nachrichten schickt, bzw. indem man ihre Methoden "aufruft". Die Auswertung der Objekte erfolgt hierbei erst zur Laufzeit (Polymorphie, dynamische Bindung der Objekte an Variable).

Wenn "alles" Objekte sind, so sind konsequenterweise auch Klassen, Nachrichten und die (formalen) Parameter Objekte.

Eine Methode der Klasse *"Klasse"* ist "new". Diese Methode erzeugt aus der Klasse eine Instanz, also ein konkretes Objekt. Jede Klasse ist eine Unterklasse der Klasse *"Klasse"* und hat somit diese Methode ererbt, kann also Instanzen von sich selbst erzeugen.

Eine konkrete Nachricht, die ein Objekt A an ein Objekt B schickt, besitzt meist aktuelle Parameter. Diese sind ebenfalls Objekte. Ebenso erwartet das Objekt A, dass das Objekt B ihm eine Nachricht mit konkreten Objekten als aktuellen Parametern zurückschickt (mittels "return").

Hinweise zur Literatur: Zum objektorientierten Programmieren gibt es heute sehr viele Bücher. Die objektorientierte Vorgehensweise wird mittlerweile auch in den Schulen (und gerne mit Java) vermittelt, sodass viele Erstsemester bereits entsprechende Programmiererfahrungen besitzen.

Objektorientierung wird im künftigen Bachelorstudium im ersten Semester unterrichtet werden. Standardlehrbücher sind zum Beispiel mit den Autorennamen Balzert, Doberkat, B. Meyer, Oesterreich, Rumbaugh, Sneed usw. verbunden.

Daneben gibt es eine Fülle von Einführungen in die Objektorientierung, die sich oft aber nur als Einführung in eine spezielle Sprache, insbesondere Java, und dann meist nur in eine aktuelle Version (z. B. JDK 1.5 oder höher), erweisen. Bei dieser Literatur ist es wichtig, sich rechtzeitig immer auf die nächste Version einzustellen.

Manche Einführungsbücher in die Informatik starten z.B. mit der Sprache Java und erläutern hieran wichtige Prinzipien. Beispiele sind:

Balzert, H., "Lehrbuch Grundlagen der Informatik", Spektrum Akademischer Verlag, Heidelberg, 2. Auflage, 2004

Gumm, H., Sommer, M., "Einführung in die Informatik", Oldenbourg Verlag, München, 7. Auflage, 2006

Horstmann, C.S., "Computing Concepts with Java 2 Essentials", Wiley, 2nd edition, 2006

Küchlin, W., Weber, A., "Einführung in die Informatik - objektorientiert mit Java", Springer, 3. Auflage, 2005

usw.

Im Bereich "Datenstrukturen und Algorithmen" weisen diverse Lehrbücher bereits im Titel auf Java hin, z. B.:

Lang, H.W., "Algorithmen in Java", Oldenbourg, München 2002,
Goodrich, M.T., Tamassia, R., "Data Structures and Algorithms in Java", Wiley, 2. Auflage, 2001
Saake, G., Sattler, K.U., "Algorithmen und Datenstrukturen, eine Einführung in Java", dpunkt-Verlag, 3. Auflage, 2006
Sedgewick, R., "Algorithmen in Java", Pearson Studium, 3. Auflage, 2003
Weiss, M.A., "Date Structures and Algorithm Analysis in Java", Addison Wesley, 2nd edition, 2007

(Dieser Hinweis unterstreicht zum einen, dass die Leser die Algorithmen konkret ausprobieren können, zum anderen ist er auch ein Verkaufsargument, um Praktiker anzusprechen; andere Bücher, die auf eine Sprachabhängigkeit verzichten, sind allein deshalb nicht besser oder schlechter - doch dies müssen Sie im Einzelfall selbst beurteilen, wenn Sie sich in einer Buchhandlung umsehen.)

1.2 Java, Teil 1

1.2.1 Erstes Beispiel

1.2.2 Grundbausteine der Sprache

1.2.3 Syntax von Java

1.2.4 Typen

1.2.5 Anweisungen

1.2.6 Methoden

1.2.7 Klassen (auch: String und Hüllenklassen)

1.2.8 Modifikatoren

1.2.9 Ein- und Ausgabe

1.2.1.a Erstes Beispiel (*das* Standardbeispiel)

Es soll nur der Text "Hello World!" ausgegeben werden.
Hierzu definieren wir eine Klasse, die dieses tut.

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

Die "main"-Methode wird als erste durchgeführt, welche den Text "Hello World!" ausgibt. Man könnte auch ein Objekt dieser Klasse erzeugen und eine zugehörige Initialisierung (= den Konstruktor, der die Anweisungen dieser main-Methode enthält) starten. Also:

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

Zuerst wird die main-Methode "new Hello()" ausgeführt, d. h., es wird ein Objekt der Klasse Hello erzeugt. Dieses startet stets mit seinem Konstruktor "Hello ()", wobei "Hello World!" ausgedruckt und das Programm beendet wird. Das Ergebnis ist also das gleiche, aber die Objektorientierung kommt hier klarer zum Ausdruck.

Wir betrachten erenut das erste der beiden Programme. Dies kann man nun modifizieren, indem man den Text zum Beispiel 5 Mal ausgeben lässt. Die for-Schleife ist syntaktisch anders als in Ada und hat auch eine andere Bedeutung, weil z.B. die Laufvariable im Schleifenrumpf verändert werden darf.

```
public class Hello {

        public static void main (String[ ] args) {

                for (int i = 1; i <= 5; i = i+1)

                        System.out.println("Hello World!");

        }

}
```

Veränderung der Laufvariablen im Rumpf der Schleife:

```
public class Hello {
        public static void main (String[ ] args) {
                for (int i = 1; i <= 5; i = i+2)  {
                        i = i-1;
                        System.out.println("Hello World!");
                }
        }
}
```

Dies liefert die gleiche Ausgabe, wie das vorherige Programm.

1.2.1.b Zweites Beispiel

x_1, x_2, x_3, ... bezeichnet man als eine Folge. Wenn eine solche Folge in einem Programm gegeben ist, kann man sie ab dem ersten bis zu einem n-ten Element ausdrucken. Wir wollen nun eine *Folge von Zahlen* betrachten.
Dies formulieren wir als Datentyp. Hierzu brauchen wir:

- das aktuell betrachtete Element (wir nennen es "aktuell"),
- ein erstes Element (wir nennen es "elem1"),
- eine Vorschrift, wie man den Nachfolger zu einem Element ermittelt (wir nennen diesen Algorithmus "naechstesElem"),
- eine Ausgabeanweisung für ein Element (dies nennen wir "drucke"),
- eine Ausgabeanweisung für die ersten n Elemente der Folge (wir nennen dies "druckeFolge (n)").

Wir sind nun nicht an einem einzelnen Programm interessiert, sondern an einem Schema, das man in anderen Programmen verwenden kann, ohne dass man angeben muss, wie die Folge aufgebaut ist. Formulierung in Ada-ähnlicher Darstellung:

```
schema Zahlenfolge is
begin
  integer elem1, aktuell := 0;
  function naechstesElem (integer x) return integer is begin ... end;
  procedure drucke (integer x) is begin ... end;
  procedure druckeFolge (positive n) is
      begin  drucke(elem1); aktuell := elem1;
            for i in 2..n loop
                  aktuell := naechstesElem(aktuell);
                  drucke(aktuell);
            end loop;
      end;
  end;
end;
```

Dies lässt sich direkt nach Java übertragen, wobei man einige Ersetzungen vornehmen muss:

"begin ... end" wird zu "{...}".

"is" weglassen.

"function" und "procedure" werden einheitlich als Funktionen (sog. "Methoden") aufgefasst. Hier wird zunächst der Ergebnisdatentyp angegeben (wenn es wie bei Prozeduren keinen gibt, so schreibt man "void"), dann folgt der Name, dann die Parameterliste.

"schema" wird zu "class".

"integer" und "positive" werden zu "long" (oder "int").

Die for-Schleife wird dargestellt durch for (int i=2; i <= n; i++). Hierbei ist i eine Integer-Laufvariable, die mit dem Wert 2 beginnt; sie wird wiederholt, solange i <= n ist; nach jedem Schleifendurchlauf wird i um 1 erhöht ("i++").

":=" wird zu "=".

Dann gibt es noch Besonderheiten, siehe übernächste Folie.

```
class Zahlenfolge {
    long elem1, aktuell;
    zahlenfolge( ) {elem1=0; aktuell=0;}
    long naechstesElem (long x) { return aktuell; }
    void drucke (long x) {
        System.out.println (x);
    }
    void druckeFolge (long n) {
        drucke(elem1);
        aktuell = elem1;
        for (int i = 2; i <= n; i++) {
            aktuell = naechstesElem(aktuell);
            drucke(aktuell);
        }
    }
}
```

Beachten Sie das Layout! In Java muss man sich an „gute Regeln“ halten!

Die Besonderheiten sind:

1) Das Ausdrucken auf das Standard-Ausgabegerät erledigt eine spezielle Methode print, die in einer Klasse mit Namen "System.out" definiert ist und die direkt im Programm genutzt werden darf. Soll anschließend zur nächsten Zeile übergegangen werden, so schreibt man println.

2) Man muss mindestens einen "Konstruktor" angeben, der den Namen der Klasse trägt. Diese spezielle Methode dient dazu, ein konkretes Objekt dieser Klasse zu bilden. Bei uns lautet er: zahlenfolge( ) {elem1=0; aktuell=0;}

3) Es darf nichts undefiniert bleiben. Daher haben wir den Rumpf ("body") der Methode naechstesElem vorläufig als { return aktuell; } angegeben.

Eine Klasse ist also ein Schema, aus dem konkrete Objekte gebildet werden. Möchte man ein solches Objekt anlegen mit dem Namen "Folge1", so schreibt man die Deklaration

```
Zahlenfolge Folge1 = new Zahlenfolge( );
```

Das reservierte Wort new bewirkt, dass eine Instanz (= eine konkrete Kopie) der Klasse Zahlenfolge erzeugt und sofort die Anweisungen des Konstruktor-Rumpfs zahlenfolge ausgeführt wird (Initialisierung). Der Konstruktor ist somit eine Methode ohne Ergebnistyp (er ist also vom Typ "void", aber man lässt das Wort "void" in der Definition des Konstruktors weg).

Eine Instanz einer Klasse nennt man "Objekt".

Ein mächtiges Hilfsmittel ist die Vererbung. Hierzu gehen wir nun zu einer speziellen Folge von Zahlen über.

Wir betrachten Folge der "Dreieckszahlen" $x_1 = 1$ und
$x_j = x_{j-1} + j$ für $j = 2, 3, ...,$ also die Folge 1, 3, 6, 10, 15, ...

Dies ist eine Folge von Zahlen. Wir können also die bereits definierte Klasse Zahlenfolge übernehmen. Hinzufügen müssen wir die "Nummer" j als neue Variable. Weiterhin müssen wir die Berechnung des nächsten Elements aus einem Element x anpassen, also neu definieren:

"Erhöhe Nummer und addiere diese Zahl zu x." , also:

```
nummer = nummer +1;
return ( x + nummer );
```

In Java erweitert (reservierte Wort: extends) man nun die Klasse Zahlenfolge zur Klasse Dreiecksfolge ("Unterklasse") und notiert nur die Neuerungen und Veränderungen:

```
class Dreiecksfolge extends Zahlenfolge {

    long nummer;
    dreiecksfolge( ) {
        nummer = 1; elem1 = 1; aktuell = 1;
    }
    long naechstesElem (long x) {
        nummer = nummer +1;
        return (x + nummer);
    }
    long nummer_der_Dreieckszahl ( ) {
        return  nummer;
    }
}
```

```
class Dreiecksfolge extends Zahlenfolge {

    long nummer;
    dreiecksfolge( ) {
        nummer = 1; elem1 = 1; aktuell = 1;
    }
    long naechstesElem (long x) {
        nummer = nummer +1;
        return (x + nummer);
    }
    long nummer_der_Dreieckszahl ( ) {
        return  nummer;
    }
}
```

Vererbt werden: elem1, aktuell, drucke, druckeFolge

Neue Variable

Konstruktor

Umdefinierte Methode

Neue Methode

Um diese beiden Klassen nun in einem Programm zu nutzen, fügen wir eine weitere Klasse mit irgendeinem Namen (wir wählen willkürlich "Probieren") wie folgt hinzu (die Zeile mit *main* kennen Sie schon bzw. ignorieren Sie einfach):

```
class Probieren {
    public static void main (String[ ]  args) {
        Zahlenfolge f = new Zahlenfolge( );
        f.drucke(5);  f.druckeFolge(5);
        Dreiecksfolge d = new Dreiecksfolge( );
        d.drucke(5);  d.druckeFolge(5);
    }
}
```

Ausgabe: 5 0 0 0 0 0 5 1 3 6 10 15

Neben der Vererbung spielt das Überladen eine wichtige Rolle. Wie in der Grundvorlesung beschrieben, dürfen Funktionsnamen mehrfach verwendet werden, sofern durch den Kontext eindeutig klar ist, welche Funktion jeweils gemeint ist. In Java wird dies durch den Vektor der Argumentdatentypen festgestellt.

Zum Beispiel könnten wir bei der Dreiecksfolge statt mit 1 mit einer anderen Zahl z beginnen wollen. In diesem Fall müssten wir elem1 und aktuell mit z initialisieren.

In Java kann man einfach einen weiteren Konstruktor in diese Klasse einfügen, der genau dies bewirkt.

Wir ändern also Dreiecksfolge ab, indem wir eine weitere Konstruktor-Methode hinzufügen:

```
class Dreiecksfolge extends Zahlenfolge {
    long nummer;
    dreiecksfolge( ) {
        nummer = 1; elem1 = 1; aktuell = 1;
    }
    dreiecksfolge(long z) {
        nummer = 1; elem1 = z; aktuell = z;
    }
    ...
}
```

In der Klasse Probieren kann man dann zum Beispiel schreiben:

```
Dreiecksfolge dd = new Dreiecksfolge(7);
dd.druckeFolge(5);
```

was die Ausgabe 7 9 12 16 21 bewirkt.

Aus Ada kennen Sie Sichtbarkeits- und Schutzmechanismen. Diese gibt es in Java ebenso. Man kann Klassen, Methoden usw. mit Zusätzen ("Modifier" genannt) versehen, z.B. bei Methoden:

public: relativ frei verfügbar / benutzbar von außerhalb des Pakets (= einer festgelegten Umgebung).

protected: diese Methode lässt sich nur aus dem gleichen Paket oder aus Unterklassen aufrufen.

private: nur in derselben Klasse aufrufbar.

kein Zusatz: die Methode heißt dann "friendly"; sie kann nur von Objekten von Klassen des gleichen Pakets aufgerufen werden.

Wir möchten nun eine andere Zahlenfolge darstellen, z. B. beginnend mit Null die Quadratzahlen 0, 1, 4, 9, 16, 25, ...

Wir wissen nun schon, wie dies geht:

- Wir nummerieren hier ab 0 durch (nummer = 0).
- Wir brauchen das Start-Element "elem1"=0.
- Wir berechnen naechstesElem nach der Formel
  $(n+1)^2 = n^2 + 2n + 1 = n^2 + 2*(n+1) - 1$.

Also:

```
long naechstesElem (long x) {
        nummer = nummer +1;
        return (x + 2*nummer-1);
    }
```

Wir erhalten:

```
class Quadratzahlen extends Zahlenfolge {
        long nummer;
        quadratzahlen( ) {
                nummer = 0; elem1 = 0; aktuell = 0;
        }
        long naechstesElem (long x) {
                nummer = nummer +1;
                return (x + 2*nummer-1);
        }
        long nummer_der_Quadratzahl ( ) {
                return  nummer;
        }
}
```

```
class Probieren {
    public static void main (String[ ]  args) {
        Zahlenfolge f = new Zahlenfolge( );
        f.drucke(5);
        f.druckeFolge(5);
        Quadratzahlen d = new Quadratzahlen( );
        d.drucke(10);
        d.druckeFolge(10);
    }
}
class Zahlenfolge {
    long elem1, aktuell;
    zahlenfolge( ) {elem1=0; aktuell=0;}
    long naechstesElem (long x) {
        return aktuell;
    }
    void drucke (long x) {
        System.out.println (x);
    }
    void druckeFolge (long n) {
        drucke(elem1);
        aktuell = elem1;
        for (int i = 2; i <= n; i++) {
            aktuell = naechstesElem(aktuell);
            drucke(aktuell);
        }
    }
}
```

```
class Quadratzahlen extends
                        Zahlenfolge {
    long nummer;
    quadratzahlen( ) {
        nummer = 0;
        elem1 = 0;
        aktuell = 0;
    }
    long naechstesElem (long x) {
        nummer = nummer +1;
        return (x + 2*nummer-1);
    }
    long nummer_der_Quadratzahl ( ) {
        return  nummer;
    }
}
```

Dies als Java-Programm laufen lassen!
Die Ausgabe lautet:

5 0 0 0 0 0 10 0 1 4 9 16 25 36 49 64 81

Machen Sie sich klar, welche Programmteile welche Ausgabe bewirken und dass die Ausgabe somit korrekt ist.

Wir möchten nun ein Java-Programm schreiben, das die Folge der Fibonaccizahlen ausgibt: $F_0 = 0$, $F_1 = 1$, $F_{n+2} = F_{n+1} + F_n$.

Wir gehen nach dem gleichen Schema vor: Statt elem1 brauchen wir nun elem0 und elem1 (wir nummerieren ab "0"). aktuell und nummer bleiben wie in "Dreiecksfolge" bzw. "Quadratzahlen". Aber "naechstesElem" ist jetzt neu zu formulieren. Das Problem dabei ist: Wir brauchen zwei Parameter an Stelle von einem. In die Unterklasse lässt sich dies leicht einfügen, aber wir müssen auch die Verwendung von "naechstesElem" in der übergeordneten Methode "druckeFolge" korrigieren.

Überlegen Sie sich die hier auftretenden Schwierigkeiten. Kann man durch Vererbung in einfacher Weise die bisherigen Methoden nutzen oder ist es besser, die bisherigen Methoden in der Unterklasse zu überschreiben?

Anmerkung:

Die relativ einfach aufgebaute Sprache Java wird durch die Fülle von Klassen, die in Bibliotheken abgelegt sind, und durch viele Hilfswerkzeuge und Entwicklungsumgebungen so unüberschaubar, dass man bereits von einem Berufsfeld des "Java-Programmierens" sprechen kann.

Wir gehen auf die professionellen Möglichkeiten nicht ein, sondern beschränken uns auf die grundlegenden Ideen und wie diese genutzt werden, um einige Probleme schön, elegant, effizient, korrekt, ... zu lösen. Insbesondere durch die Anwendungen in Netzen und die Integration in Produkte (embedded systems) wurde Java stark verbreitet. Jede(r) von Ihnen wird daher irgendwann sicher erneut mit dieser Sprache (oder einer Nachfolgeversion) konfrontiert werden.

1.2.2 Grundbausteine der Sprache

Java ist prinzipiell so aufgebaut wie andere Sprachen auch: Es gibt Grundsymbole (Literale), aus denen sich elementare Strukturen und Anweisungen ergeben, die durch geeignete Daten-/Kontrollstruktur-Operationen zu größeren Einheiten zusammengefügt werden. Die syntaktische Struktur wird durch die EBNF festgelegt.

Es folgt nun über 3 Doppelstunden ein "integrierter" Kurz-Programmierkurs in Java. Die Programme sollten Sie mit JDK 1.6, Eclipse oder ähnlichen Werkzeugen bearbeiten.

1.2.2.1 Zeichensatz und Zeichen (siehe Grundvorlesung 2.4.7)

Als Zeichensatz verwendet Java den 16-stelligen Unicode (ISO-Standard 10646). Hiervon wird in Westeuropa meist nur der 8-stellige Latin1-Zeichensatz ausgenutzt .

Grundelement ist also das Zeichen (character).
Die in Java übliche Unterteilung der Zeichen lautet:
Ziffern: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
Trennzeichen: ,(das Komma), ., ;, (,), [,], {, }
Leerzeichen = die Latin1-Zeichen mit den Nummern 32, 9 und 12: Zwischenraum (space, Nummer 32), Tabulatorzeichen (HT, Nummer 9) und Seitenumbruch (FF, Nummer 12).
Zeilenende: Die Latin1-Zeichen mit den Nummern 10 und 13: Zeilenvorschub (LF, 10) und Wagenrücklauf (CR, 13). Die Folge LF CR wird als ein Zeilenende ("eol") aufgefasst.
Operationszeichen: +, -, *, /, &, <, >, =, |, ^, %, !, ?, ~, :
Buchstaben: die üblichen Buchstaben (zum Schreiben von Text in irgendeiner Sprache) sowie $ und _

Darstellung der Zeichen: Jedes Zeichen (= Element des Unicode) wird wie üblich in Apostrophe eingeschlossen. Das Zeichen A wird also als 'A' dargestellt. Will man das Apostroph, die Anführungsstriche oder nicht auf der Tastatur vorhandene Zeichen darstellen, so muss man (wie in Scheme) das Zusatzzeichen \ ("Escape-Sequenz") verwenden. Beispiele hierfür:

\'	bezeichnet das Apostroph '
\"	bezeichnet die Anführungsstriche "
\\	bezeichnet \ (backslash oder Rückwärtsquerstrich)
\u	beginnt die hexadezimale Darstellung eines Unicode-Zeichens. \u00A5 ist also das Unicodezeichen mit der 16-stelligen Bit-Repräsentation 0000000010100101
\b	bezeichnet das Zurücksetzen um ein Zeichen (Backspace)
\r	bezeichnet den Wagenrücklauf (carriage return)
\n	bezeichnet den Zeilenvorschub (line feed)
\t	bezeichnet das Tabulatorzeichen

Beispiele: '0' '§' ' ' 'Œ' 'λ' '\"' '\"' '\\' '\u0010' '\u3' 'Щ' '∩'
(Hinweis: Escape-Sequenzen dürfen überall im Programm auftauchen; sie werden schon zur Compilezeit ersetzt.)

1.2.2.2 Zeichenketten

Aus Zeichen werden Zeichenketten (Strings) zusammengesetzt. Wie üblich werden diese als Folge von Zeichen, die in Anführungsstriche eingeschlossen sind, dargestellt. Für Apostroph, Anführungsstriche und nicht gängige (auf der Tastatur nicht vorhandene) Zeichen ist der Backslash \ zu verwenden. Beispiele für Zeichenketten:

"Informatik in Stuttgart" "" (leere Zeichenkette)

"Der Java-Standardtext heißt: \"Hello World!\""

"Die \"Informatik in Stuttgart\""

"Um das Apostroph \' darzustellen, muss man \\\' schreiben."

"Lottozahlen\n 5\t17\t20\t28\t41\t42\nliefert 6 Zahlen und 3 Zeilen"

(Dagegen liefert "\u0022" einen Fehler, weil zuerst \u0022 zum Zeichen " ausgewertet wird und dann """ nicht sinnvoll interpretiert werden kann. Was ist aber mit "\\u0022" und "\\\u0022" ? Im Zweifel ausprobieren.)

1.2.2.3 Reservierte Wörter in Java (im Jahre 2007)

Folgende 50 Wörter ("reserved words" oder "keywords") haben eine feste Bedeutung in Java und dürfen nicht anders verwendet oder umdefiniert werden:

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

Zur Festlegung von Eigenschaften und Verwendung von Daten, Methoden oder Klassen dienen hierbei die reservierten Wörter

abstract	final	native	private
protected	public	static	strictfp
synchronized	transient	volatile	

In der Regel schreiben wir in dieser Vorlesung die reservierten Wörter mit blauer Farbe oder wir unterstreichen sie, damit man sie klar erkennen kann. Hierdurch wird die Struktur eines Programms besser sichtbar.

Kurz gefasste Auflistung der Bedeutungen der reservierten Wörter. Hiervon kennen Sie das Meiste schon aus Ada.

abstract	die jeweilige Klasse oder die jeweilige Methode bleibt ohne Implementierung (z. B. abstrakte Klasse in Java)
assert	Ausnahmefall, falls Bedingung hinter assert false ist
boolean	elementarer Datentyp der Wahrheitswerte
break	sofortiges Beenden einer Kontrollstruktur
byte	8-Bit langer Datentyp der ganzen Zahlen $-2^7..2^7-1$
case	Fall in einer Fallunterscheidung (switch)
catch	Abfangen einer Ausnahmesituation
char	elementarer Datentyp der Zeichen
class	Beginn einer Klassendefinition
const	[noch ohne Bedeutung, erst künftig vorgesehen]
continue	Weitermachen beim Schleifenanfang
default	Standardfall in einer Fallunterscheidung
do	Schleife mit Abbruchprüfung nach dem Schleifenrumpf

double	64-Bit langer Datentyp der reellen Zahlen
else	Beginn des alternativen Falls einer if-Anweisung
enum	zur Definition eines eigenen Aufzählungstyps
extends	Bildung einer Unterklasse durch Vererbung
final	Keine Unterklasse erlaubt bzw. Implementierung oder Wert nicht in Unterklassen veränderbar
finally	Abschluss gewisser Ausnahmen (catch ... finally ...)
float	32-Bit langer Datentyp der reellen Zahlen
for	Laufschleife
goto	[noch ohne Bedeutung, erst künftig vorgesehen]
if	Alternative, Beginn der if-Anweisung
implements	Implementierung einer Schnittstelle durch eine Klasse
import	Einbindung einer anderen Klasse
instanceof	Test, ob ein Objekt zu einer Klasse gehört
int	32-Bit langer Datentyp der ganzen Zahlen $-2^{31}..2^{31}-1$

interface	Beginn einer Schnittstellendefinition
long	64-Bit langer Datentyp der ganzen Zahlen
native	wird für Methoden, die in einer anderen Programmiersprache geschrieben sind, verwendet
new	erzeugt ein Objekt eines Referenzdatentyps
package	Paket-Zusammenfassung
private	Verstecken von Details, Verbieten von Zugriffen
protected	Schutz vor Zugriffen
public	Klasse ist erzeugbar und erweiterbar im gleichen Paket oder durch Importieren
return	sofortiges Verlassen einer Methode
short	16-Bit-Datentyp der ganzen Zahlen $-2^{16}..2^{16}-1$
static	Zuordnung einer Methode usw. zu einer Klasse
strictfp	erzwingt, dass Gleitpunktoperationen mit einer bestimmten Darstellung durchgeführt werden, um auf verschiedenen Plattformen stets gleiche Ergebnisse zu erhalten
super	Verweis auf die Oberklasse

switch	Beginn einer Fallunterscheidung
synchronized	Synchronisationsvorschrift bei Nebenläufigkeit
this	Verweis auf die eigene Klasse
throw	Übergang zu einer Ausnahmebehandlung
throws	Angabe, dass woanders definierte Ausnahmen in dieser Klasse oder Methode ausgelöst werden können
transient	bezeichnet die Komponenten, die beim Speichern eines Objekts nicht gespeichert werden sollen
try	Beginn eines Blocks, in dem bestimmte Ausnahmen auftreten können (die in einem anschließenden catch- plus eventuellem finally-Block behandelt werden)
void	Attribut einer Methode ohne Rückgabewert
volatile	Variablen, die durch nebenläufige Prozesse verändert werden können
while	Schleife mit Abbruchprüfung vor dem Schleifenrumpf (vgl. do)

1.2.2.4 Bezeichner in Java (= identifier)

Bezeichner beginnen mit einem Buchstaben. Beachten Sie: $ oder _ sind am Anfang eines Bezeichners ebenfalls erlaubt. Danach kann eine beliebige Folge von Buchstaben (einschließlich $ und _) und Ziffern stehen. Bezeichner dürfen beliebig lang sein. Einschränkungen:

Als Bezeichner dürfen ***nicht*** verwendet werden:
- die reservierten Wörter (siehe 1.2.2.3),
- die Wörter (spezielle "Konstanten") true, false, null.

Auch die drei Wörter (für wichtige vordefinierte Klassen) Object, String, System sollten Sie nicht als Bezeichner verwenden, da hierbei leicht Konflikte und somit schwer aufzufindende Fehler entstehen können.

Empfehlung: Verwenden Sie immer nur aussagekräftige Bezeichner. Setzen Sie '$' und '_' nur ein, wenn es sinnvoll ist.

Wichtig: Java ist "case sensitive", d.h., Java unterscheidet Groß- und Kleinschreibung! Reservierte Wörter und Namen von Paketen müssen z. B. mit kleinen Buchstaben geschrieben werden. "class" ist also ein reserviertes Wort, während "Class" als Bezeichner für eine Variable oder eine Klasse verwendet werden kann.

Empfehlungen, an die sich fast alle Java-Programmierer halten: In Java beginnen Bezeichner für Methoden in der Regel mit einem kleinen, Bezeichner für Klassen mit einem großen Buchstaben. Auch Variablen beginnen mit einem kleinen Buchstaben. Innerhalb eines Variablennamens benutzt man große Buchstaben zum deutlichen Abgrenzen.

Beispiel: int ersteZahl, zweiteZahl;

Konstanten schreibt man meist nur mit Großbuchstaben.

1.2.2.5 Literale (vgl. Grundvorlesung 3.2.2)

Literale bezeichnen Konstanten elementarer Datentypen oder Zeichenketten. Elementare Datentypen (in Java meist "primitive" Datentypen genannt) sind [in Klammern die zugehörigen Java-Schlüsselwörter]:

- Boolesche Werte (boolean): false, true. (Default: false)
- Zeichen (char). (Default: \u0000)
- ganze Zahlen (byte, short, int, long), auch mit Vorzeichen, auch oktale und hexadezimale Darstellung ist möglich.
- Gleitpunktzahlen (float, double), auch mit Vorzeichen.

Weiterhin zählt die Null-Referenz null zu den Literalen.

Noch vorzustellen sind die numerischen Literale (ihr Default-Wert ist stets die Null) und deren Darstellung in Java:

byte (8-Bit-Darstellung): ganze Zahlen von -128 bis +127,

short (16-Bit-Darstellung): ganze Zahlen von -32768 bis +32767,

int (32-Bit-Darstellung): ganze Zahlen von -2^{31} bis $+2^{31}-1$,

long (64-Bit-Darstellung): ganze Zahlen von -2^{63} bis $+2^{63}-1$.

Gibt man nichts an, so ist "int" gemeint. 64-Bit-Darstellungen müssen am Ende ein großes oder kleines L besitzen. Oktale und hexadezimale Darstellungen sind erlaubt; sie beginnen mit 0 und dürfen nur die Ziffern 0 bis 7 verwenden, bzw. sie beginnen mit 0x und dürfen dann die Ziffern 0 bis 9 sowie die Buchstaben A bis F (oder auch a bis f) enthalten. Beispiele:

17 0 -5 144L 789123654L 0xA0F -0xFABEL
056 00 0xaffe

float (32-Bit-Darstellung) und double (64-Bit-Darstellung) sind Darstellungen für reelle Zahlen. Sie werden stets dezimal notiert. Ein Dezimalpunkt und/oder ein Exponent "e <ganze Zahl>" sind zulässig (E statt e ist erlaubt). Float-Literale erhalten ein nachgestelltes f oder F, double-Literale ein nachgestelltes d oder D; d darf auch fehlen. Eine Ziffernfolge ohne eines dieser Zusatzmerkmale wird als ganze Zahl interpretiert. Zum Beispiel ist 45 ganzzahlig, 45f dagegen eine Gleitpunktzahl.

Beispiele für double-Zahlen:
1.23 1.23e3 -9e-9 23D 15. 0.123 .123 0.123d .123d

Beispiele für float-Zahlen:
1.23f 1.23e3F -9e-9f 23f -0.123f .123F

float umfasst den Bereich bis maximal $\pm 10^{38}$, double bis $\pm 10^{308}$.

1.2.2.6 Kommentare (vgl. Grundvorlesung 3.2.2)

Es gibt drei Ausprägungen von Kommentaren in Java.

// ...	Der Kommentar endet automatisch mit dem Zeilenende.
/* ... */	Der Kommentar zwischen /* und */ darf beliebig lang sein.
/** ... */	Wie /* ... */, doch der Kommentar kann von Java-Werkzeugen weiter verarbeitet werden, und zwar zur Übernahme in eine dokumentierte Fassung des Programms. Hier darf man auch Steuerzeichen einstreuen; bitte selbst ausprobieren.

1.2.2.7 Operatoren in Java

+, -, *, /	Übliche Operatoren auf Zahlen (/ bezeichnet zugleich die ganzzahlige Division)
%	Modulo-Bildung ganzer Zahlen
<<, >>, >>>	bitweises Verschieben nach links, nach rechts und nach rechts mit Nullextension
&, |, ^	bitweise Boolesches and, or und xor

Schreibt man hinter jeden der obigen 11 Operatoren ein "=", so wird der Operator mit einer Zuweisung gekoppelt gemäß:

y += 8 bedeutet das Gleiche wie y = y + 8

<, <=, >, >=, ==, !=	die üblichen Vergleichsoperatoren
!, &&, ||	logisches not, and und or
++, --	Inkrement und Dekrement jeweils um 1
=	ist der Zuweisungsoperator (in Ada: ":=")
+	Konkatenieren von Strings.

Operatoren treten in Ausdrücken auf. Meist werden sie inorder verwendet. Man muss daher Prioritäten angeben. Die Liste der von unten nach oben wachsenden Prioritäten lautet:

nachgestellte Postfixoperatoren:	++ -- . [ ] (<Parameterliste>)
einstellige Präfixoperatoren:	+ - ++ -- ! ~
Referenz und casting:	new (<Datentyp>)
Mult./Div:	* / %
zweistellige Addieroperatoren:	+ -
Verschieben (bitweise):	<< >> >>>
Vergleich/Element:	< > <= >= instanceof
Gleichheit/Ungleichheit:	== !=
and (bitweise):	&
exor (bitweise):	^
or (bitweise):	|
and (boolean):	&&
or (boolean):	||
Bedingung im Ausdruck:	? :
alle 12 Zuweisungsoperatoren:	= += -= *= *(usw.)*

1.2.2.8 Ausdrücke

Ausdrücke bestehen aus Werten, Variablen, Funktionen und Operatoren. Sie besitzen einen (Ergebnis-) Typ.

Variablen werden in der Form

<Datentyp> <Liste von Bezeichnern für Variablen> ;

deklariert. Initialisierung ist zulässig. Beispiele:

```
int hundert = 100; long zahl;
double kontoBestand, prozente, zusatzZins;
String vorgegebenerText, gesuchterText;
```

Konstanten sind die Literale (1.2.2.5) oder selbst definierte Werte. Die Definition erfolgt nach dem Schema:

final <Datentyp> <Bezeichner> = <Wert>;

Beispiele:

```
final double EULER = 2.7182818284d;
final float PI = 3.1415926f;
```

Ausdrücke werden wie üblich entsprechend der Stelligkeiten und Typen aus Operatoren/Funktionen und Konstanten/Variablen gebildet (Funktionen werden als Methoden definiert).

Auch Zuweisungen sind (Zuweisungs-)Ausdrücke x = y*(a+b). Daher ist auch x = q = r = y*(a+b) ein (Zuweisungs-) Ausdruck (sog. Mehrfachzuweisung). Ebenso ist

x -= q += z = (3+2)*8

zulässig. Die Auswertung der Zuweisungsoperatoren "-=", "+=" und "=" erfolgt von rechts nach links. Interpretiert man diesen Ausdruck als Anweisung (Wertzuweisung), so ist er gleichbedeutend mit der folgenden Folge von Anweisungen ("=" ist der normale Zuweisungsoperator):

z = (3+2)*8; q = q + z; x := x - q;

Autoinkrement und Dekrement: ++ und --.

Diese Operatoren können vor oder hinter einer Variablen stehen:

i++ bedeutet: Verwende zunächst den Wert von i und erhöhe anschließend den Wert von i um 1.

++i bedeutet: Erhöhe zuerst den Wert von i um 1 und verwende diesen Wert.

Analog für i-- und --i.

Beispiel:

```
int a = 1;              // a besitzt anschließend den Wert 1.
int b = ++a;            //Anschließend sind a gleich 2 und b gleich 2.
int c = b++;            //Anschließend sind c gleich 2 und b gleich 3.
int d = b--  +  ++a;    //Danach sind a gleich 3, d gleich 6, b gleich 2.
int e = ++c - d++;      //Danach sind c gleich 3, e gleich -3, d gleich 7.
```

Wenn # ein Operator ist, so bedeutet x #= e den Ausdruck x = x # e . Dabei wird die Adresse von x aber nur einmal berechnet. Diese Semantik führt zusammen mit Seiteneffekten zu schwer verstehbarem Code; zugleich ist die obige Aussage, x#=e sei gleich x=x#e, nur noch für einfache Variablen richtig (Beispiel siehe nächste Folie).

Auch das Autoinkrement i++ (mit der Bedeutung i = i+1) kann je nach Programmumgebung andere Berechnungen auslösen, da innerhalb eines Ausdrucks stur von links nach rechts unter Beachtung der Prioritäten ausgewertet wird. Ein Beispiel ist:

```
a[i] = a[i] * ++i;        dies ist äquivalent zu:
                          i = i+1; a[i] = a[i-1] * i;
a[i] = ++i * a[i];        dies ist äquivalent zu:
                          i = i+1; a[i] = a[i] * i;
```

(Hier lauern beliebig viele Fehlerquellen.)

Beispiel: Gleiche Deklaration, zwei gleiche Zuweisungen?

```
int i = 0; int [ ] a = {0, 1, 2, 3, 4};
int g(int k) {
      i++; return (k+2);
}
```

Zuweisung 1: Die Adresse von a[g(i)] wird zweimal berechnet:

```
a[g(i)] = a[g(i)] + 5;    /* setzt a[3] auf den Wert a[2]+5 (== 7),
                             anschließend hat i den Wert 2. */
```

Zuweisung 2: Die Adresse von a[g(i)] wird nur einmal berechnet:

```
a[g(i)] += 5;             /* setzt a[2] auf den Wert a[2]+5 (== 7),
                             anschließend hat i den Wert 1. */
```

In Java (wie auch in C und in funktionalen Sprachen) werden Anweisungen zunächst wie Ausdrücke behandelt. Ein Ausdruck wird zur Anweisung durch das Anfügen eines ";". So ist i++ zunächst ein Ausdruck, der zu einer Wertzuweisung durch Anhängen eines Semikolons wird: i++;

Auch die Alternative *if b then x else y* kann man als dreistelligen Operator auf Ausdrücken auffassen mit der Bedeutung: Ist b erfüllt, so sei x das Ergebnis, anderenfalls y.

In Java gibt es diesen dreistelligen Operator für Ausdrücke in der Form **b ? x : y**

Beispiel: (a < b) ? b : a
berechnet das Maximum von a und b, und

(a < b) ? ((c < a) ? c : a) : ((c < b) ? c : b)
berechnet das Minimum der drei Zahlen a, b und c.

1.2.3 Syntax von Java **(etwas veraltet, nicht vollständig, selbst aktualisieren!)**

Für das Folgende gibt es (wie stets) keine Garantie für Fehlerfreiheit.
EBNF, d.h.: Eckige Klammern = ein- oder keinmal. Geschweifte Klammern = beliebig oft (inkl. keinmal). Runde Klammern dienen zum Zusammenfassen. Anführungszeichen klammern Terminalzeichen ein. Terminalzeichen sind zusätzlich in blau geschrieben. Kursives ist selbsterklärend. Nichtterminalzeichen stehen hier nicht in spitzen Klammern. Startsymbol der EBNF ist compilation unit. Den Abschluss jeder Regel bildet ein Punkt. Intervalle x..y bedeuten, dass jeder Wert von x bis y hier stehen darf. Es gibt noch Einschränkungen, die hier nicht sichtbar sind. Es folgen die wichtigsten 50 Regeln:

compilation_unit ::=
    [package_statement] { import_statement } { type_declaration }.

package_statement ::= "package" package_name ";" .

import_statement ::= "import" ((package_name "." "*" ";") |
    (class_name | interface_name) ";") .

type_declaration ::=
    [d_comment] (class_declaration | interface_declaration) ";".

d_comment ::= "/**" *Folge von Zeichen ohne */* "*/" .

```
class_declaration  ::=
        {modifier}  "class"  identifier ["<" identifier {"," identifier} ">" ]
        [ "extends" class_name ] [ "implements"  interface_name
        { "," interface_name } ]  "{"  {field_declaration}  "}"  .
interface_declaration  ::= { modifier }  "interface" identifier
         [  "extends" interface_name  { "," interface_name }  ]
        "{"  { field_declaration }  "}"  .
field_declaration  ::= ( [ d_comment ]
        ( method_declaration | constructor_declaration |
variable_declaration ) ) | static_initializer | ";"  .
method_declaration ::= {modifier} type identifier "(" [ parameter_list ] ")"

        { "["  "]" } [ "throws" identifier {"," identifier} ]
        ( statement_block  |  ";"  ) .
constructor_declaration  ::=  {modifier} identifier "(" [ parameter_list ] ")"

        [ "throws" identifier {"," identifier} ] statement_block  .
statement_block  ::=  "{"  { statement }  "}"  .
variable_declaration  ::=  { modifier }  type  variable_declarator
```

```
variable_declarator  ::=  identifier { "["  "]" } [ "=" variable_initializer ] .
variable_initializer  ::= expression |
        ( "{" [variable_initializer {"," variable_initializer} [ "," ] ] "}" ) .
static_initializer  ::= "static" statement_block  .
parameter_list  ::= parameter  { "," parameter }  .
parameter  ::=  type  identifier  { "["  "]" }  .
statement  ::=  variable_declaration  |  ( expression  ";"  )  |
statement_block  |  if_statement  |  do_statement  |          while_statement
| for_statement  |  assert_statement  |
        try_statement  |  switch_statement  |
        ( "synchronized"  "(" expression ")" statement )  |
        ( "return" [ expression ] ";" ) |  ( "throw" expression ";" )  |
        ( identifier  ":" statement )  |  ( "break"  [ identifier ]  ";" )  |
        ( "continue"  [ identifier ]  ";" )  |  ";"  .
if_statement ::= "if"  "(" expression  ")" statement  [ "else" statement ] .
do_statement ::= "do" statement  "while"  "(" expression  ")"  ";"  .
while_statement ::= "while"  "(" expression  ")" statement  .
```

```
assert_statement ::= "assert" logical_expression [ ":" string ] ";"
for_statement ::= "for"  "("  ( variable_declaration  |  ( expression ";"  )  |
";"  )  [ expression ]  ";"  [ expression ]  ")" statement  .
try_statement ::= "try" statement
        { "catch"  "(" parameter  ")" statement } [ "finally" statement ]  .
switch_statement ::= "switch"  "(" expression ")"
        "{" { ( "case" expression ":"  |  "default" ":" )  statement } "}" .
expression  ::= numeric_expression  | testing_expression  |
        logical_expression  | string_expression  | bit_expression  |
        casting_expression  | creating_expression  | literal_expression  |
        "null"  |  "super"  |  "this"  | identifier  |  (  "(" expression ")"  )  |
        ( expression  (  (  "("  [ arglist ]  ")"  )  |  (  "[" expression "]"  )  |
        (  "." expression )  |  (  "," expression )  |
        (  "instanceof"  ( class_name | interface_name )  )  )  )  .
numeric_expression  ::=  (  (  "-"  |  "++"  |  "--"  )  expression )  |
        ( expression  (  "++"  |  "--"  )  )  | ( expression  (  "+"  |  "+="  |
        "-" | "-=" | "*" | "*=" | "/" | "/=" | "%" | "%=" )  expression  )  .
```

```
testing_expression  ::= expression
        (  ">"  |  "<"  |  ">="  |  "<="  |  "=="  |  "!="  )  expression  .
logical_expression  ::=  ( expression  ( "&"  |  "&="  |  "|"  |  "|="  |  "^"  |
        "^="  |  ( "& &" )  |  "||="  |  "%"  |  "%="  ) expression ) |
        ( "!" expression )  |  ( expression "?" expression  ":" expression ) |

        "true"  |  "false"  .
string_expression  ::=  ( expression  (  "+"  |  "+="  )   expression  )  .
bit_expression  ::=  (  "~" expression ) |
        ( expression  (  ">>="  |  "<<"  |  ">>"  |  ">>>"  )  expression  )  .
casting_expression  ::=  "(" type  ")" expression  .
creating_expression  ::=  "new" ( ( classe_name "(" [ arglist ] ")" ) |
                ( type_specifier [ "[" expression "]" ] { "["  "]" } ) |
                                ( "(" expression ")" )  )  .
literal_expression  ::=  integer_literal  | float_literal  | string  | character  .
arglist  ::=  expression  { "," expression }  .
type  ::=  type_specifier  { "["  "]" }  .
type_specifier ::=  "boolean" | "byte" | "char" | "short" | "int" | "float" |
        "long"  |  "double"  | class_name | interface_name  .
```

modifier ::= "public" | "private" | "protected" | "static" | "final" |
"native" | "synchronized" | "abstract" | "transient" | "volatile".

package_name ::= identifier | (package_name "." identifier) .

class_name ::= identifier | (package_name "." identifier) .

interface_name ::= identifier | (package_name "." identifier) .

integer_literal ::= (("1..9" {"0..9"}) | " 0 " {"0..7"} |
("0" "x" "0..9 a..f" {"0..9 a..f"})) ["l" | "L"] .

float_literal ::= (decimal_digits "."
[decimal_digits] [exponent_part] [float_type_suffix]) |
("." decimal_digits [exponent_part] [float_type_suffix]) |
(decimal_digits [exponent_part] [float_type_suffix]) .

decimal_digits ::= "0..9" { "0..9" } .

exponent_part ::= ("e" | "E") ["+" | "-"] decimal_digits .

float_type_suffix ::= "f" | "F" | "d" | "D" .

character ::= " *Element des unicode-Zeichensatzes* " .

string ::= " " " {character} " " " .

identifier ::= "a..z,$,_" {"a..z,$,_,0..9, *unicode-Zeichen oberhalb 00C0*"} .

Hinweise:

Statt a..f darf in der Regel für integer integral auch A..F stehen.
Das Gleiche gilt für a..z in der Regel für identifier.
("a..z,$,_ ,0..9") ist als Vereinigungsmenge von Intervallen und Elementen zu lesen.)

Man kann zusätzliche Nichtterminalzeichen einführen, z. B.:

decimal_digit ::= " 0..9 "

separator ::= "," | "." | ";" | "(" | ")" | "[" | "]" | "{" | "}"

operator_character ::= "+" | "-" | "*" | "/" | "%" | "&" | "|" | "!" | "^" |
"?" | "~" | "!" | "<" | ">" | "=" | ":"

Obige Syntax orientiert sich an der ursprünglichen Definition von Java (1995). Für Java 1.5 ist sie aber unvollständig. Hierfür gibt es eine umfangreichere Syntax mit feiner gegliederten Nichtterminalen. In unseren Regeln fehlen die später hinzugefügten Sprachelemente, z.B. Regeln für enum. *Schema hierfür:*

enumDeclaration ::= "enum" identifier [implementsList] enumBody .

enumBody ::= "{" enumConstant { "," enumConstant }
[";" { classOrInterfaceBodyDeclaration }] "}" .

enumConstant ::= identifier [arglist] [classOrInterfaceBody] .

implementsList ::= implements classOrInterfaceType {"," classOrInterfaceType} .

1.2.4 Typen

1.2.4.1 Datentypen

Es gibt die *primitiven Datentypen* (in Klammern die Zahl der Bytes, die für ihre Realisierung erforderlich sind; Zahlen werden intern im Zweierkomplement dargestellt):

boolean (1), char (2), byte (1), short (2),
int (4), long (8), float (4), double (8),

die Typen char (für Zeichen, siehe 1.2.7.3) und boolean (für die Wahrheitswerte), die selbstdefinierten *Aufzählungstypen* der einfachen Form

enum <Name> { <Liste der Elemente> } ;

und *Referenz-Datentypen* zu array, class und interface (diese werden als Verweis / Zeiger / Adresse / Referenz auf den Speicherbereich, in dem ihre Komponenten stehen, realisiert).

1.2.4.2 Variablen

Variablen werden wie in der imperativen Programmierung als Behälter eingesetzt. Jede Variable speichert einen elementaren Wert oder einen Verweis auf einen Speicherbereich. Variablen müssen vor ihrer Verwendung deklariert werden. Dabei ist ihr Wert entweder undefiniert oder sie können explizit initialisiert werden.

In Java muss jede Variable, bevor sie erstmals gelesen wird, einen Wert erhalten haben. Dies wird bereits zur Übersetzungszeit überprüft, allerdings nur auf oberster Typhierarchie, s. u.

Die Deklaration erfolgt durch Angabe des Datentyps gefolgt von einer Liste der Variablen und eventuell der initialisierenden Zuweisung. Für die Initialisierung von Referenzdatentypen ist der Operator new (oder die Konstante null) zu verwenden.

Beispiele:

```
short a, b, c;  char x = 'B';
float r;  float pi = 3.1415926;  double q = -4.443e-13;

Dreiecksfolge dd = new Dreiecksfolge(7);
Dreiecksfolge d = new Dreiecksfolge( );

float[ ] zeugnisnoten = new float[11] ;
float[ ] noten = null;
float[ ] zn = {1.0, 1.3, 1.7, 2.0, 2.3, 2.7, 3.0, 3.3, 3.7, 4.0, 5.0};
```

Der new-Operator initialisiert zugleich die Komponenten mit den Standardwerten ("default", siehe 1.2.2.5). Ansonsten wird einer Variablen kein Anfangswert automatisch zugewiesen. Der Wert von noten[1] ist also undefiniert (und dies erkennt der Compiler nicht), während zn[1] gleich 1.3 ist.

Typanpassung:

Java ist im Prinzip streng typisiert. Variablen müssen einen Typ haben und behalten diesen; Operatoren verlangen, dass ihre Argumente den zugehörigen Typ besitzen. Daher muss man in der Regel den Typ eines Wertes explizit anpassen (diese Anpassung nennt man casting):

(<gewünschter Typ>) <Variable>

Beispiel: float a, b; int z; ...; a = b + (float) z; ...

Nur in folgenden Fällen erfolgt eine implizite Anpassung (sog. "conversion") in Java:
wenn Zahlen von ganzzahligen nach reellwertigen Werten umgewandelt werden und hierbei die Genauigkeit des Wertes nicht verringert wird (int liefert den ganzzahligen Anteil)
oder wenn in einem Operator zwei verschiedene numerische Werte benutzt werden, wobei die Umwandlung immer zum genaueren Datentyp hin erfolgt.

Beispiele:

int i = 3; int j = 5; float k = 1.5f; double d = 2.5D;

i = (int) k (liefert 1)	d = d/j (liefert 0.5D)
k = j (liefert 5.0)	i = d/j Fehler, wegen der
d = i/j (liefert 0.0D)	Genauigkeit
d = ((double)i) /j (liefert 0.6D)	i = (int)d/j (liefert 0)

1/3 liefert den Wert 0 (/ ist hier die ganzzahlige Division)

1/3.0 liefert 0.33333333333D

(weil: 3.0 ist vom Typ double und die ganze Zahl 1 wird dorthin umgewandelt).

Empfehlung: Stets explizite Typanpassung verwenden!

1.2.4.3 Felder / arrays

Wenn T ein Datentyp ist, so ist T[ ] der Feld-Datentyp über T. Die einzelnen Komponenten besitzen die Indizes 0, 1, 2, ... Felder sind dynamisch, d.h., sie werden erst zur Laufzeit angelegt, und die obere Feldgrenze kann daher durch einen Ausdruck angegeben werden (die untere Grenze ist immer 0). Ist die obere Grenze festgelegt, kann man sie nicht mehr verändern.

```
int t = 12; int v = t+t; ...
float[ ] werte = new float[(t*t-v)/13];
```

erzeugt somit unter dem Namen "werte" ein Feld von 9 float-Elementen, die mit 0.0 initialisiert sind.

Die eckigen Klammern dürfen auch rechts vom Bezeichner stehen: `float werte [ ] = new float[(t*t-v)/13];`

Erzeugen eines Feldes:

- durch Auflisten der Elemente
  int[ ] q = {7, 3, 5, 7, 9} (das 5-elementige Feld (7,3,5,7,9))
- durch den new-Operator
  int[ ] r = new int[<Integer-Ausdruck>];
- durch null (das Feld bleibt zunächst unspezifiziert).

Die Länge (=Anzahl der Elemente) eines Feldes zählt ***nicht*** zum Datentyp array. Es gibt also keine mehrdimensionalen Felder, vielmehr muss man

<Datentyp> [ ] [ ] [ ] feldname = ...

für das gewohnte dreidimensionale Feld schreiben. Andererseits kann man nun auch Dreiecksmatrizen und andere Strukturen definieren. Zum Beispiel:

```
float [ ] [ ] rechteck_matrix = new float [30] [20];

int [ ] [ ] pascal_dreieck = {{1}, {1,1 }, {1,2,1}, {1,3,3,1},
        {1,4,6,4,1}, {1,5,10,10,5,1}, {1,6,15,20,15,15,6,1}};
```

Für die Variable pascal_dreieck sind also genau die Indizes [0] [0], [1] [0], [1] [1], [2] [0], [2] [1], [2] [2], [3] [0], ... zulässig.

Mit jeder Dimension des array-Datentyps ist stets die Anzahl der Elemente (= Länge) als Attribut verbunden. Man schreibt <Variable>.length.

Zum Beispiel gilt:
rechteck_matrix.length hat den Wert 30,
rechteck_matrix[i].length hat den Wert 20 (für jedes i von 0 bis 29),
pascal_dreieck.length hat den Wert 7,
pascal_dreieck[0].length hat den Wert 1,
pascal_dreieck[3].length hat den Wert 4.

Beispiel: Zufallszahlen zählen (vgl. Buch [J. Bishop], 6.1.3).

```
class Haeufigkeiten {
    Haeufigkeiten ( ) {
        int bis = 30;
        int treffer [ ] = new int [bis];
        int zufallszahl;
        for (int i = 0; i < 1000; i++) {
            // Math.random ist eine Methode zur Erzeugung einer Zufallszahl
            zufallszahl = (int) (Math.random ( ) * bis);
            treffer[zufallszahl] ++;
        }
        for (int i = 0; i < bis; i++)
            System.out.println(i + ": " + treffer[i]);
    }
    public static void main (String [ ]args) {
        new Haeufigkeiten ( );
    }
}
```

Ein Beispiellauf ergab folgende Ausgabe für die Häufigkeiten der Zahlen von 0 bis 29 bei 1000 Versuchen:

0: 27	15: 34
1: 38	16: 34
2: 41	17: 30
3: 33	18: 28
4: 29	19: 31
5: 27	20: 29
6: 30	21: 34
7: 39	22: 32
8: 40	23: 42
9: 24	24: 31
10: 34	25: 27
11: 28	26: 42
12: 41	27: 41
13: 32	28: 35
14: 27	29: 40

Experimentieren Sie mit Varianten dieses Programms. Untersuchen Sie, wie "zufällig" die Ausgabe-Verteilung der ersten 30 Zahlen wirklich ist. Ergänzen Sie Berechnungen zum Mittelwert, zur mittleren Abweichung hiervon usw.

1.2.5 Anweisungen

Alle Anweisungen dürfen eine Marke besitzen in der Form

<Bezeichner> : <Anweisung>

Typen von Schleifen:

for ([<Initialisierungen>]; [<Bedingung>]; [<Veränderungen>])
<Anweisung>

while (<Bedingung>) <Anweisung>

do <Anweisung> while (<Bedingung>)

Siehe Grundvorlesung, Kontrollstrukturen: Es gibt also die Laufschleife (for), die while-Schleife (beginnend mit while) und die repeat-Schleife (beginnend mit do und der Wiederholungsbedingung nach while am Ende). Bei der for-Schleife darf die Laufvariable auch neu deklariert werden.

Beispiele (mit folgender Deklaration):

```
float [ ] quadrat = new float [20];   int k = 1; int m = 15;

for (int i = 0; i < 20; i++)  quadrat[i] = float(i*i);
```

Ergebnis: Diese Schleife legt 0, 1, 4, 9, ..., 361 im Feld quadrat ab.

```
while (k < 20)  {
        quadrat[k] = quadrat[k] + quadrat[k-1];
        k++;
}
```

Ergebnis: verändert die Werte von quadrat nach einer Art Fibonacci-Schema (stellen Sie fest, welche Werte berechnet werden).

```
do {quadrat[m] = float(3*m); m -=1;} while (m > 20)
```

Ergebnis: verändert die Werte von quadrat[15] in 45 und von m in 14. Die do-while-Schleife wird mindestens einmal durchgeführt.

Lesen Sie sich nochmals die Beschreibung von Schleifen in der Grundvorlesung Abschnitt 2.1.5, A8 durch! Die for-Schleife in Java entspricht der (ziemlich unstrukturierten) allgemeinen for-Schleife (A8d). Wie dort geben wir auch hier ein Äquivalent zur for-Schleife an und empfehlen, die for-Schleife in Java stets nur wie eine Laufschleife zu verwenden.

Beachte: Das Java-Konstrukt

for (init_1, init_2, ..., init_i; <Bedingung>;
verändern_1, ..., verändern_k) <Anweisung>

ist gleichbedeutend mit

```
{ init_1; init_2; ..., init_i;
  while (<Bedingung>); { <Anweisung>;
        verändern_1; ...; verändern_k}
}
```

Sprachelemente, um den Kontrollfluss zu beeinflussen (hinzu kommt noch return zum Beenden von Methoden siehe 1.2.6).

break; beendet die innerste, break umgebende Anweisung switch, for, while oder do-while (ähnlich zu exit in Ada; man verlässt die laufende Anweisung).

continue [<Marke>]; beendet den laufenden Schleifendurchlauf. continue ist nur innerhalb von Schleifen zulässig. Der Rest des Schleifenrumpfs wird übersprungen; man bleibt aber innerhalb der Schleife. Wird eine Marke angegeben, so ist die Schleife gemeint, die mit dieser Marke benannt wurde.

if-Anweisung: Die ein- und zweiseitige *Alternative* (= if-Anweisung) hat die gewohnte Bedeutung. Der Ausdruck muss einen Wahrheitswert liefern. Die von uns angegebene Syntax in 1.2.3

```
if_statement ::= "if" "(" expression ")" statement [ "else" statement ].
```

ist mehrdeutig, da zum Beispiel für

```
if (x!=0) if (x<0) a=1; else a=2;
```

unklar ist, zu welchem if das else gehört. Prinzipiell legt man hierfür das letzte if fest. Obige Anweisung bedeutet also

```
if (x!=0) { if (x<0) a=1; else a=2; }
```

Kurze Aufgabe: Wäre folgende Syntax eindeutig?

```
"if" "(" expression ")" statement
["else" "if" "(" expression")" statement ]
["else" statement ]
```

Für geschachtelte if-Anweisungen sollte man daher stets { ... } oder notfalls die switch-Anweisung verwenden.

Die switch-Anweisung realisiert *Fallunterscheidungen*. Allerdings entspricht die Semantik nicht der Fallunterscheidung von Ada. In Java muss hinter switch ein Ausdruck vom Ergebnistyp byte, short, int, char (oder ein selbstdefinierter enum-Typ) stehen. Der Wert dieses Ausdruckes sei x. Es werden die case-Fälle der Reihe nach durchgegangen, bis der erste Fall für x zutrifft oder bis man auf default trifft. Die zugehörige Anweisung wird ausgeführt und danach alle folgenden Anweisungen (!) in der switch-Anweisung. Will man nur eine Anweisung ausführen, so muss man explizit ein "break" einfügen (break beendet den Rumpf der switch-Anweisung, s. o.). Häufiges Schema:

```
switch (<spezieller_Ausdruck_siehe_oben>) {
        case <Konstante_1>: <Anweisungsfolge_1>
        case <Konstante_2>: <Anweisungsfolge_2>
        ...
        case <Konstante_k>: <Anweisungsfolge_k>
        default: <Anweisungsfolge>
}
```

Beispiel: Addiere 1 modulo 4

```
switch (k) {
        case 0:  k = 1; break;
        case 1:  k = 2; break;
        case 2:  k = 3; break;
        default: k = 0; break;
}
```

Beispiel: Umrechnen von Noten in Text

```
switch (note) {
    case 1: System.out.print("sehr ");
    case 2: System.out.print("gut"); break;
    case 3: System.out.print("befriedigend"); break;
    case 4: System.out.print("ausreichend"); break;
    case 5: System.out.print("mangelhaft, ");
    default: System.out.print("gehen Sie zur Beratung"); break;
}
```

Noch ein Beispiel:

```
for (int [ ] q = {0, 2, 1, 3}, int k = 0; k < q.length; k++)  {
    switch (q[k])  {
            case 0: System.out.print("Wenn hinter ");
            case 1: System.out.print("\b Fliegen Fliegen");
                    if (k==2) break;
            case 2: System.out.print(" fliegen,"); break;
            default: System.out.println(" nach.");
    }
}
```

\b ist der "Backspace", d.h., das Zeichen davor wird gelöscht.

In Java gibt es auch "Blöcke". Syntax:

statement_block ::= "{" { statement } "}"

Anweisungen werden in Java mit einem ";" abgeschlossen, sodass in der Syntax keine Trennzeichen zwischen den Anweisungen explizit auftreten.

Einen Einfluss auf die Speicherverwaltung haben die Blöcke in Java nicht.

Da jedes statement wieder ein Block sein kann, lassen sich Blöcke beliebig schachteln.

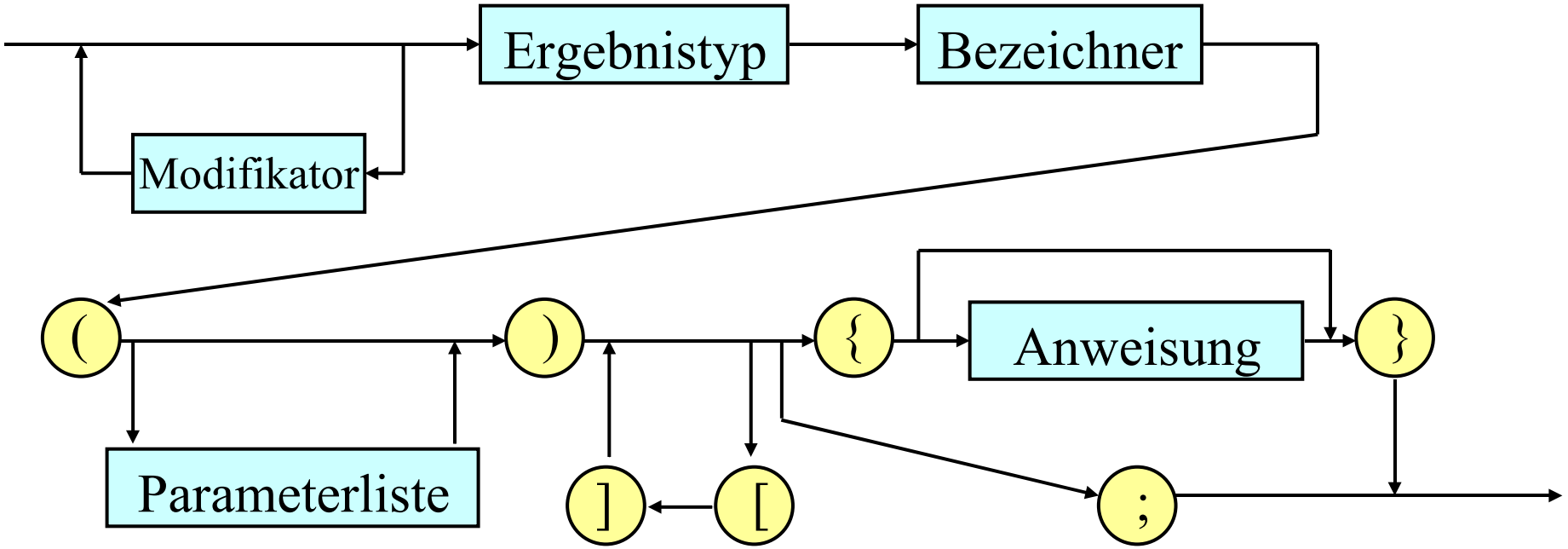
Blöcke dienen zum einen dem Zusammenfassen von Anweisungen, zum anderen definieren sie die Lebensdauer bzw. den Sichtbarkeitsbereich von Bezeichnern, vgl. 1.11.5 der Grundvorlesung. Wird zum Beispiel in einem Block eine Variable deklariert, so ist sie sichtbar ab der Deklarationsstelle bis zum Ende des Blockes (das Ende ist durch "}" gegeben). Die Begriffe global und lokal: wie üblich.

In Java braucht man zwischen Lebensdauer und Sichtbarkeit nicht zu unterscheiden, da es in Java verboten ist, eine Variable in einem Unterblock umzudeklarieren.

1.2.6 Methoden (Algorithmen/Funktionen)

Die Deklaration einer Methode hat den syntaktischen Aufbau

{ modifier } type identifier "(" [parameter_list] ")"
{ "[" "]" } (statement_block | ";")



Beispiel Fakultätsfunktion (iterativ berechnet)

```
long fak (int a) {
      if (a == 0) return 1L;
      else {
            long ergebnis = 1L;
            for (int i = 1; i <= a; i++)
                  ergebnis *= (long) i;
            return ergebnis;
      }
}
```

Beispiel Fakultätsfunktion (rekursiv berechnet)

```
long fak (int a) {
      if (a == 0) return 1L;
      else return (long) a * fak(a-1);
}
```

Beispiele ggT und Maximum

```
int ggT (int a, int b) {
        if (b == 0) return a;
        else return ggT(b, a % b);
}
```

```
int max (int a, int b, int c) {
        return max(a, max(b,c));
}
int max (int x, int y) {
        if (x > y) return x;
        else return y;
}
```

Überladen ist in Java erlaubt, wenn sich die Funktionen in ihren Argumenten in Anzahl und/oder Typ unterscheiden

```
return (x < y) ? y : x
```

genügt, vgl. 1.2.2.8

Beispiel Suchen in einem Feld: Suche den Schlüssel s in einem long array a in den Grenzen von links bis rechts (siehe Grundvorlesung 6.5.1); zurückgegeben wird der Index bzw. links-1.

```
int n = ...;  ...
long [ ] feld = new long [n]; ...
```

```
int suchen (long [ ] a, int links, int rechts, long s) {
      int mitte;
      while (links <= rechts)  {
            mitte = (links+rechts)/2;
            if (a[mitte] == s)  return mitte;
            else   if (a[mitte] < s)  links = mitte + 1;
                  else rechts = mitte - 1;
      };
      return (links - 1);
}
```

Aufruf im Programm: suchen(feld,0,n-1,schluessel).

Methoden brauchen keinen Ergebnistyp zu besitzen. In diesem Fall lautet der Ergebnistyp void. Beachten Sie, dass eine Methode stets ein Klammerpaar besitzt, auch wenn keine Parameter übergeben werden.
Java überprüft, dass jeder mögliche Zweig in einer Methode auf ein return mit einem Ausdruck des zugehörigen Ergebnistyps stößt. Im Falle des Ergebnistyps void muss entweder das Ende der Methode oder ein parameterloses return erreicht werden.

Weiterhin darf jede Methode Modifikatoren besitzen. Diese lauten (Erläuterung siehe später in 1.2.8 oder Literatur):

public, protected, private	regeln den Zugriff auf die Methode
static	Methode ist eine Klassenmethode
abstract	Die Implementierung erfolgt erst in Unterklassen.
final	nicht in Unterklassen veränderbar
synchronized	Synchronisation von Prozessen
native	Plattformabhängigkeit

Parameterübergabe in Java

(1) Formale Parameter werden als lokale Variable der Methode aufgefasst. Beim Aufruf ("call") der Methode werden sie mit den Werten der aktuellen Parameter ("Übergabeparameter") initialisiert.

(2) Die Übergabe erfolgt call-by-value.

call-by-value heißt hier:
Ist der Parametertyp primitiv, so wird nur der Wert (nicht aber eine Referenz auf eine Variable) übergeben. Ist der aktuelle Parameter eine solche Variable, so kennt die Methode diese Variable nicht und kann daher ihren Wert auch nicht verändern.
Ist der Parametertyp ein Referenzdatentyp, so wird die Adresse des Objekts übergeben und die Methode kann die Komponenten des Objekts verändern.
(Vgl. Abschnitt 4.2 der Grundvorlesung.)

Wir verwenden zur Illustration das Standardbeispiel "vertausche zwei Variableninhalte".

```
void vertausche (float X, float Y) {
        float H = X;
        X = Y;
        Y = H;
}
int a = 2; int b = 7;
vertausche (a, b);
```

Hier werden nur die Werte 2 und 7 übergeben. Nach dem Methodenaufruf haben die Variablen a und b unverändert die Werte 2 bzw. 7.
Wir betrachten daher nun den Referenzdatentyp "array" als formalen Parameter und vertauschen zwei seiner Komponenten

```
void vertausche (float[ ] X, int i, int j) {
        float H = X[i];
        X[i] = X[j];
        X[j] = H;
}
float [ ] feld = {3.0, 5.0, 8.0, 9.0};
int a = 1; int b = 3;
vertausche (feld, a, b);
```

Hierdurch wird das array feld = (3.0, 5.0, 8.0, 9.0) in (3.0, 9.0, 8.0, 5.0) abgeändert. Man beachte, dass X die Referenz auf das Objekt feld beim Aufruf erhält und dass diese Referenz auch nicht verändert wird, also dem call-by-value-Prinzip unterliegt. Jedoch kann die Methode nun auf die Komponenten des Objekts feld zugreifen, weil dessen Adresse in X steht.

Anderes Standardbeispiel: Gerade/Ungerade:

```
boolean gerade (int a) {
        if (a == 0) return true;
        else return ungerade(a-1);
}
boolean ungerade (int a) {
        if (a == 0) return false;
        else return gerade(a-1);
}
```

Hinweis: In Java benötigt man keine gesonderte Vorab-Spezifikation wie in Ada. Java ermittelt zuvor alle auftretenden Namen und übersetzt oder interpretiert den Text erst anschließend.

1.2.7 Klassen

1.2.7.1 Aufbau von Klassen in Java

In einer Klasse werden Datenstrukturen (mit Selektoren bzw. Regelung des Zugriffs auf Komponenten), Algorithmen (Methoden), erforderliche Variablen und Angaben zur Initialisierung (Konstruktor) zusammengefasst.

Ein Objekt ist eine Instanz (= mit Werten versehene Kopie) einer Klasse. Es wird mittels new erzeugt, wobei ein Speicherbereich und eine Referenz auf den Anfang des Speicherbereichs angelegt werden und die Initialisierung durchgeführt wird.

Ein Java-Programm ist eine Menge von Klassendefinitionen. Eine dieser Klassen muss ausgezeichnet werden (in Java durch "main"), damit das Programm eindeutig gestartet werden kann.

Aufbau einer Klasse:

class_declaration ::=

{modifier} "class" identifier ["<" identifier {"," identifier} ">"]

["extends" class_name] ["implements" interface_name
{ "," interface_name }] "{" {field. declaration} "}"

In Java soll ein Klassenname stets mit einem großen Buchstaben beginnen.

modifier: sinnvoll sind nur "public" und alternativ "abstract" (= keine Instanziierung) oder "final" (keine Unterklassen möglich).

extends gibt bei Vererbung die vererbende Oberklasse an.

Ein "interface" ist im Wesentlichen das, was man in Ada als Spezifikation, allerdings mit Mehrfachvererbung, bezeichnet. implements bedeutet, dass diese Klasse eine konkrete Implementierung einer oder mehrerer solcher Spezifikationen bildet.

field_declaration ist eine Variablen-, Methoden- oder Konstruktordeklaration.

Beispiel (vergleiche 4.6 der Grundvorlesung):

```
class Punkt {
      private float x;
      private float y;
      Punkt () {x = 0.0f; y = 0.0f;}
      Punkt (float a, float b) {x = a; y = b;}
      float ausgabeX ( ) {return x;}
      float ausgabeY ( ) {return y;}
      String ausgabe ( ) {
            return "X-Koordinate: " + x + ", Y-Koordinate: " + y;
      }
      private boolean diagonal () {return x*x == y*y;}
      void verschieben (float diffx, float diffy)
            { x = x + diffx; y = y + diffy; }
      void verschieben (float v) { verschieben (v, v); }
      void strecken (float s) {x *= s;  y *= s; }
}
```

```
class Kreis {
    private float radius;
    private Punkt mitte;
    Kreis () {einheitsKreis (0.0f, 0.0f, 1.0f);}
    void einheitsKreis (float a, float b, float c)
        {mitte.x = a; mitte.y = b; radius = c; }
    float q (float z) {return z*z;}
    Punkt ausgabePunkt ( ) {return mitte;}
    float ausgabeR ( ) {return radius;}
    String ausgabe ( ) {
        return "Mittelpunkt: (" + mitte.x + ", " + mitte.y + "),
        Radius: " + radius; }
    private boolean aufRand (float a, float b)
       {return q(radius) == q(mitte.x-a)+q(mitte.y-b);}
    void verschieben (float diffx, float diffy)
       {mitte.x += diffx; mitte.y += diffy; }
    void vergroessern (float s) {radius *= s; }
}
```

In der Grundvorlesung haben wir gefordert, dass Programm-einheiten Parameter besitzen dürfen; insbesondere können diese Parameter auch Datentypen oder Algorithmen sein. In Java ist dies zulässig. Man gibt nach dem Klassennamen Bezeichner in spitzen Klammern für Typparameter an, die zunächst nicht festgelegt werden (Generizität). Beispiel: Folge von Elementen:

```
class Folge <Element> {
    Element inhalt;
    Folge<Element> rest;
    Folge<Element> (Element a) {inhalt = a; rest = null;}
    Folge<Element> (Element a, Folge<Element> b)
            {inhalt = a; rest = b;}
}
```

Man sieht zugleich, dass Klassen rekursiv verwendet werden können, um z.B. Listen, Bäume oder Graphen zu definieren.

Ähnlich wie in Ada müssen die konkreten Typen bei Variablendeklarationen angegeben werden. Will man einer Variablen X eine Folge von Zeichenketten zuordnen, so schreibt man

Folge <String> X = new Folge <String> ("Wetter");

Dies erzeugt eine Variable vom Typ Folge aus Zeichenketten, die mit der Zeichenkette "Wetter" und rest = null initialisiert wurde. Analog:

Folge <Integer> X = new Folge <Integer> (25);

X = new Folge <Integer> (51, X); ...

Als Typen müssen Klassennamen verwendet werden. Die primitiven Typen müssen daher durch die ihnen zugrunde liegenden Klassen ("Hüllenklassen", siehe 1.2.7.4) ersetzt werden. Die Hüllenklasse von int ist Integer, weshalb im Beispiel Integer und nicht int steht.

1.2.7.2 Programmstart mittels main:

Genau eine Klasse des Java-Programms enthält eine Methode der Form

public static void main (String[ ] args) { <Rumpf> }

oder mit drei Punkten im Parameterteil (ab Version JDK 1.5)

public static void main (String ... args) { <Rumpf> }

Als Parameter für main dienen sog. Kommandozeilenparameter, die wir zunächst ignorieren. Mit dieser main-Methode wird das Programm vom Java-System gestartet.

Beachten Sie, dass "main" eine Klassenmethode ist und daher den Modifikator "static" besitzt. Static-Attribute können nur static-Attribute aufrufen bzw. verwenden, siehe 1.2.8.

Hinweis: Java besitzt bei gewissen Java-Umgebungen diverse Zusatzregeln. Zum Beispiel muss das Programm genau unter dem Klassennamen, der das Programm beschreibt, abgelegt werden (in der Form <Name>.java), sodann muss man die Kommentare des Compilers bei Fehlern interpretieren lernen, weiterhin muss man beim Ablaufenlassen des Programms oft hinter "java" ein " -classpath " einfügen und darf nun nur noch den Klassennamen (ohne Erweiterung) angeben usw.

Java ist auch nicht für Anfängerprogramme konzipiert, sondern entfaltet seine Mächtigkeit erst, wenn man die bereits vorhandenen Klassen im eigenen Programm einsetzt. Die vordefinierten Klassen und die großen Klassenbibliotheken von Java bilden daher einen riesigen Baukasten, aus dem man ständig immer komplexere Programme zusammenstecken kann.

1.2.7.3 Character

Die Darstellung der Datentypen char und String hatten wir schon in 1.2.2.1 und 1.2.2.2 besprochen, doch kann man mit den dortigen Informationen noch keine Textverarbeitung durchführen. Beim Typ char muss man klären, wie man diesen Typ durch-laufen kann. Beispiel: eine Codierungstabelle der um 3 Zeichen im Alphabet verschobenen Buchstaben, die in Ada lauten würde:

```
H: Character array ('a'..'z') of Character;        -- mit der Initialisierung:
for i in 'a'..'z' loop
        H(i) := H(Character'Val(Character'Pos(i)+3));
end loop;
```

Weiterhin sind alle zulässigen Operationen auf char anzugeben.

Prinzip: Der Datentyp char in Java wird mit den natürlichen Zahlen von 0 bis 2^{16}-1 = 65535 gleichgesetzt. Dies sind die vorzeichenlosen ganzen int-Zahlen, die mit 16 Bit darstellbar sind. Allerdings ist Java stark typisiert, weshalb char und int nicht einfach addiert werden dürfen. Vielmehr muss zuvor stets eine Typumwandlung (casting, 1.2.4.2) erfolgen. Für char-Variablen a und c sind also a=a+1, c++ oder a=*c nicht erlaubt, wohl aber a = (char) ((int) a * (int) c). Beispiel:

```
public class Test_Char1 {
   static int q = 105; static char a = '\u0041'; static char c = (char) 66;
   public static void main (String [] args) {
      System.out.print(q + ", " + (char) q + ", " + a + ", " + c + "; ");
      System.out.print( (char) (9+(int)a) + ", " + (char) ((int) c -10) );
      for (int i = 40; i < 67; i += 2)
         System.out.print(", " + (char)i + " -> " + (char)(i+3));
   }
}
```

liefert:

105, i, A, B; J, 8, (-> +, * -> -, , -> /, . -> 1, 0 -> 3, 2 -> 5, 4 -> 7, 6 -> 9, 8 -> ;, : -> =, < -> ?, > -> A, @ -> C, B -> E

1.2.7.4 Hüllenklassen (wrapper classes)

Die Variablen der primitiven Datentypen speichern ihre Werte unmittelbar, d.h., die zugehörigen Methoden oder Eigenschaften sind nicht zu sehen. Die vollständigen Datentypen (Klassen) existieren natürlich auch. Man nennt sie Hüllenklassen (wrapper classes) und sie können mit folgenden Namen angesprochen werden (beachte: die primitiven Typen beginnen mit einem kleinen, die Klassen mit einem großen Buchstaben):

Primitiver Datentyp	Zugehörige (vordefinierte) Hüllenklasse
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Hüllenklassen sind erforderlich, wenn man nicht die Werte eines primitiven Datentyps, sondern die Objekte übergeben will/muss oder wenn die zugrunde liegende Klasse gefordert ist (1.2.7.1). In diesen Klassen sind zugleich alle zulässigen Operationen und Attribute gespeichert, zum Beispiel die größte und kleinste darstellbare Zahl eines numerischen Datentyps oder die Umwandlung in ein Objekt einer anderen Klasse (vgl. die folgenden Folien).

1.2.7.5 Strings (Zeichenketten), Klassen für Zeichenketten

In Ada ist ein String im Wesentlichen ein array aus Zeichen. In Java ist es eine Folge von Zeichen (Zeichenkette, Sequenz). In Java gibt es drei vordefinierte Klassen für Zeichenketten, die voneinander unabhängig, insbesondere also keine Ober- oder Unterklassen voneinander sind:

String	Objekte dieser Klasse können nach ihrer Initialisierung nicht mehr verändert werden. *(im Paket java.lang)*
StringBuffer	Veränderbare Zeichenketten bzgl. Länge und Inhalt. *(im Paket java.lang)*
StringTokenizer	Besitzt zusätzliche Methoden, um eine Zeichenkette in Teile ("token") zu zerlegen. *(im Paket java.util)*

Methoden der Klasse String:

- String concat (String s): hängt s an den vorhandenen String an. Statt u.concat(v) kann man auch u + v schreiben.
- Konstruktor String (char [] c), der aus einem Feld von Zeichen die Folge der Zeichen als String liefert.
- public int length () liefert die Länge des aktuellen Strings.
- public int indexOf (char b) liefert die Position, an der im aktuellen String zum ersten Mal das Zeichen b steht (oder -1, falls b im String nicht vorkommt).
- public char charAt (int i) liefert das i-te Zeichen des Strings (beachte, dass die Nummerierung von 0 bis length-1 geht).
- public boolean equals (... q) liefert true, falls q ein Objekt der Klasse String ist und den gleichen Text wie der aktuelle String besitzt. Anderenfalls wird false geliefert.

Weitere Methoden der Klasse String:

- boolean startsWith (String s), prüft, ob der aktuelle String mit dem String s beginnt.
- boolean endsWith (String s), prüft, ob der aktuelle String mit dem String s endet..
- String substring (int i, int j), liefert den Teilstring des aktuellen Strings von der Position i bis zur Position j. Falls i = j ist, so wird ein String der Länge 1 geliefert; ist i > j, so wird der leere String zurückgegeben.
- indexOf (String s), liefert den kleinsten Index, ab dem s ein Teilstring des aktuellen Strings ist; falls s nicht im aktuellen String als Teilstring vorkommt, wird -1 zurückgegeben.

Wir werden diese Methoden beim Beispiel "Erkennen von Teiltexten" in 1.5 benutzen.

Methoden der Klasse StringBuffer:

- StringBuffer append (String s) hängt s an den vorhandenen StringBuffer an.
- StringBuffer insert (int i, String s) fügt den String s ab der Position i in den aktuellen StringBuffer ein. (Der StringBuffer wird also auch um die Länge von s verlängert.)
- void setCharAt (int i, char c) überschreibt das Zeichen, das an der Position i steht, mit dem Zeichen c.
- Konstruktor StringBuffer (String s) initialisiert einen neuen StringBuffer mit der Zeichenkette s.
- Konstruktor StringBuffer ( ) erzeugt einen StringBuffer ohne Inhalt.
- int length ( ) liefert die aktuelle Länge des StringBuffers.
- public int indexOf (char b), public char charAt (int i), public boolean equals (... q) und manche andere Methoden verhalten sich wie die gleich benannten Methoden der Klasse String.

Die Klasse StringTokenizer fasst einen String als Folge von Teilstrings auf, die im ursprünglichen String durch Trennzeichen abgegrenzt sind. Es gibt viele Methoden hierfür, z. B.:

- StringTokenizer (String s) Konstruktor zur Erzeugung einer Folge von Zeichenketten aus s, die durch ' ', '\r', '\n' oder '\t' getrennt sind (diese Trennzeichen treten im erzeugten Objekt nicht mehr auf).
- StringTokenizer (String s, String trenn) Konstruktor zur Erzeugung eines Objekts aus s, wobei die Menge der Trennzeichen genau die in trenn enthaltenen Zeichen sind.
- public String nextToken( ) gibt den Teilstring bis zum nächsten Trennzeichen zurück.

Durch den Konstruktor legt man die Menge der Trennzeichen fest, nach denen ein String zerlegt werden soll; dies lässt sich aber im Laufe der Bearbeitung wieder ändern. Bei Texten könnte man den Punkt als Trennzeichen verwenden und sich dann von Satz zu Satz mittels nextToken( ) voran arbeiten. Zur Analyse von Java-Programmen könnte man die Trennzeichen '{', '}', ';' verwenden usw.

1.2.7.6 Umwandeln in Strings ("toString" und "valueOf")

Jede Klasse K sollte sich selbst präsentieren können oder jedes Objekt sollte seinen Zustand in lesbarer Form ausgeben können. Daher findet sich in jeder gängigen Java-Klasse eine aus der Klasse Object vererbte oder umdefinierte Methode der Form toString( ).

Beispiel: In der (Hüllen-) Klasse Integer könnte es folgende Methode toString geben, die allerdings nicht korrekt arbeitet (warum?):

```
static String toString (int x) {
    if (x < 0) return "-" + natToString(-x);
    else return natToString(x);
}
static String natToString (int x) {
    if (x == 0) return "";
    else return natToString (x/10) + (x % 10);
}
```

(*Hinweis:* Um dies zu testen, füge man zum Beispiel

```
public static void main (String[] args) {
    int z = 489125; int y = -9888678; String S = toString(286*5041);
    System.out.print(S + "  " + toString(z) + "  " + toString(y));
}
```

hinzu, lege dies alles in eine Klasse, übersetze sie und führe sie aus.)

Die (Hüllen-) Klasse Integer enthält also unter anderem:

- einen Konstruktor Integer (int z), um ein Zahl-Objekt mit dem Wert z (in 32-Bit-Darstellung) zu initialisieren,
- einen Konstruktor Integer (String s), um ein Objekt mit der Zahl, die durch den Text s dargestellt ist, in 32-Bit-Darstellung zu initialisieren,
- eine Klassenmethode String toString (int x), um eine int-Zahl in einen Text umzuwandeln,
- eine Klassenmethode Integer valueOf (String s), um aus dem Text s das zugehörige Integer-Objekt zu erhalten.

usw. Dies gilt analog auch für andere Klassen. Beispiel:

```
String langzahl = Long.toString(12345678909876543 21);
Long z = Integer.valueOf (langzahl);
```

(Bitte selbst in Handbüchern weiter recherchieren.)

1.2.7.7 Pakete (package)

Klassen, die inhaltlich oder funktional zu einem gleichen Gebiet gehören, fasst man gerne als Pakete zusammen, zum Beispiel die Klasse "Math". Hierdurch lassen sich Klassen auch schützen und innerhalb eines Pakets verwendete Namen kollidieren nicht mit extern benutzten Namen.

Pakete sind hierarchisch angeordnet, d.h., Pakete enthalten oft weitere Pakete. Pakete realisieren also Verzeichnisse (directories). Der Zugriff erfolgt wie gewohnt über die Punktnotation.

Ein großes Paket ist java . In java gibt es das Paket java.lang . Hierin ist Math eine Klasse, auf die also über java.lang.Math zugegriffen werden kann.

Will man Teile eines Pakets nutzen, so kann man entweder den kompletten Zugriffspfad angeben oder man kann die benötigten Teile mittels

import <Name des Zugriffspfads>

importieren. Will man alle Teile eines Pakets nutzen, so schreibt man

import <Name des Zugriffspfads des Pakets> . *;

Hierbei werden allerdings keine Unter-Pakete mit-importiert; diese muss man alle einzeln importieren.

1.2.8 Modifikatoren

static

Bei der Einführung in die objektorientierte Programmierung wurde zwischen Klassen- und Objekt-bezogenen Konstanten, Variablen und Methoden (diese 3 Begriffe fasst man unter "Attribute" zusammen) unterschieden, siehe Bild in 1.1. In einer Klasse "Geometrie" wäre zum Beispiel die Konstante PI für alle Objekte gleich. Man macht sie daher zu einer Klassen-Konstanten. In Java macht der Modifikator "static" aus einem Attribut ein Klassen-Attribut. Der Zugriff erfolgt über den Klassennamen und die Punktnotation.

Für das Klassen-Attribut ist eine Instanz (also ein Objekt mit seinen Objekt-Attributen) aber nicht sichtbar. Daher können static-Attribute nur static-Attribute verwenden oder aufrufen.

private (nicht für Klassen zugelassen)

Wenn man möchte, dass ein Attribut nicht von außen, also weder von einer Oberklasse noch von einer Unterklasse noch von fremden Objekten aus zugreifbar sein soll, so gibt man ihm den Modifikator "private".

public

Der Modifikator public macht diese Klasse oder ihr jeweiliges Attribut für alle Objekte und Klassen zugreifbar, die diese Klasse importieren können.

< keine Angabe >

Steht vor einer Klasse oder einem ihrer Attribute kein Modifikator, so ist die Klasse bzw. ihr Attribut für alle Objekte zugreifbar, die im gleichen Paket wie die Klasse stehen.

protected (nicht für Klassen zugelassen)

Diesen Modifikator verwendet man bei Vererbungen für Attribute. protected-Attribute können innerhalb der Klasse, in allen Unterklassen und in dem gesamten Paket genutzt werden. Dies gilt auch für Unterklassen, die via import in einem Paket deklariert worden sind.

abstract

Von dieser Klasse oder diesem Attribut lassen sich keine Instanzen bilden; zugehörige Attribute sind also erst in Unterklassen definiert.

final

Von dieser Klasse dürfen keine Unterklassen gebildet werden bzw. dieses Attribut darf in Unterklassen nicht verändert oder überschrieben werden.

1.2.9 Ein- und Ausgabe (weitere Details siehe z.B. [J.Bishop])

Eingabe und Ausgabe sind in Java Teil der Interaktionen, die ein Programm mit seiner Umwelt verbindet. Diese finden über (nebenläufige) Datenströme statt. Ein Datenstrom ist hierbei eine Folge von Daten (in der Regel eine Folge von Zeichenketten), die aus einer Quelle stammen. Eine Quelle kann eine Datei, ein Sensor, der Internetanschluss, ein Fenster auf dem Bildschirm, die Tastatur, eine Diskette usw. sein.

Ein- und Ausgabe werden somit von Methoden kontrolliert, die von gewissen Klassen der Sprache Java bereit gestellt werden. Diese Klassen sind vordefiniert und müssen "nur noch geeignet" zusammengestellt werden. (Wir betrachten hier nur den einfachsten Fall "Tastatur und Bildschirm".)

Vordefiniert für die Ausgabe sind die Methoden "print" und "println", die sich in der Klasse "System" im Objekt "out" befinden. Wir erreichen sie durch System.out.print bzw. System.out.println . Wenn wir an eine dieser Methoden eine Zeichenkette übergeben, so wird diese auf dem Bildschirm angezeigt, wobei println am Ende der Ausgabe zusätzlich zur nächsten Zeile übergeht. Die Zeichenkette darf aus mehreren Zeichenketten und Variablen bestehen, die miteinander durch "+" verbunden sind. (Der Operator "+" bewirkt hierbei die Aneinanderreihung oder Konkatenation der einzelnen Zeichenketten zu einer gesamten Zeichenkette, siehe 1.2.7.5).

Beispiel für eine Bildschirmausgabe:

```
public class Ausgabebeispiel {
    static String text = "Heute " + "ist der ";
    static short tag = 30;
    static String s = text + tag + ". Oktober";
    public static void main(String[] args) {
        System.out.println("Hello" + '\n' + "World!");
        System.out.println();
        System.out.println(s + ".");
    }
}
```

Zugehörige Ausgabe (Escape-Sequenz '\...' beachten, 1.2.2.1):

```
Hello
World!

Heute ist der 30. Oktober.
```

Dem System.out entspricht ein System.in. Die Eingabe ist aber komplizierter. Es gibt eine abstrakte Klasse InputStreamReader. Diese erhält "System.in" als Parameter und schließt den Datenstrom an "System.in" an. Dieses Objekt geben wir an die Klasse BufferedReader. Mit deren Methode readLine wird dann eine gepufferte Eingabe (bis zum Drücken der return-Taste auf der Tastatur) möglich. Die erforderlichen Klassen sind alle im Paket java.io enthalten. Die Eingabe speichern wir in einem String (hier mit dem Bezeichner zeile). Schema:

```
import java.io.*;
BufferedReader Eingabestrom =
    new BufferedReader (new InputStreamReader(System.in));
String zeile = Eingabestrom.readLine();
```

Da Fehler bei der Eingabe auftreten können, müssen alle an der Eingabe irgendwie beteiligten Methoden die Ausnahmebehandlung "IOException" ereichen können (mittels throws).

Beispiel: Solange von der Tastatur zeilenweise lesen und am Bildschirm ausgeben, bis in einer Zeile nur eine Null steht:

```
import java.io.*;
class Anzeigen {
   String zeile;
   Anzeigen() throws IOException {
      do {
         BufferedReader Eingabestrom =
            new BufferedReader(new InputStreamReader(System.in));
         zeile = Eingabestrom.readLine();
         System.out.println(zeile);
      } while (! zeile.equals("0"));
   }
   public static void main(String[] args) throws IOException {
      new Anzeigen();
   }
}
```

Nun können wir zwar eine Zeichenkette in eine String-Variable einlesen, aber wir können noch nicht die einzelnen Bestandteile lesen. Dies sollte man selbst programmieren, wobei es diverse Hilfsfunktionen in den existierenden Klassen gibt.

Besteht zeile nur aus einer Gleitkommazahl, so kann man diese wie folgt extrahieren und einer float-Variablen zahl zuweisen:

```
zahl = Float.valueOf(zeile).floatValue();
```

Das hilft aber noch nicht, wenn mehrere Zahlen oder Werte verschiedener Datentypen in einer Zeile stehen oder wenn Informationen über mehrere Zeilen gehen. Hierfür sollten Sie sich eigene Methoden schreiben, die Sie dann an geeigneten Stellen in Ihr Programm importieren und "in einfacher Weise" nutzen können. Siehe Java-Literatur, auch und vor allem bzgl. Ein-/Ausgabe in Dateien.

1.3 Beispiele

Vorbemerkung: Wie überträgt man ein bestehendes imperatives Programm nach Java, wenn man mit Objekten eigentlich gar nicht arbeiten möchte?

Wir haben dies bereits kennen gelernt: Man definiere irgendeine Klasse mit leerem Konstruktor und schreibe das imperative Programm in den Rumpf von public static void main (...) {**...**}. Dabei verwende man nur static-Attribute. Hierdurch wird kein Objekt erzeugt, sondern es wird nur auf der Klassen-Ebene ein Programm ausgeführt. Alternativ kann man das imperative Programm auch in den Konstruktor der Klasse stecken und im Rumpf von main nur new <Klasse> schreiben, wie wir dies in 1.2.1.a mit der Klasse Hello demonstriert haben.

"Fast" nach dieser Idee, die den Bezug zur bisherigen Programmierung herstellt, werden nun zwei Beispiele vorgestellt.

1.3.1.a Zufällige Erzeugung beliebig großer Felder

Zufallszahlen (besser "Pseudozufallszahlen") sind in der Umgebung eclipse bereits enthalten (vgl. 1.2.4.3, Math.random( )) oder werden durch import java.util.*; einem Programm zugänglich.
[Hier ist die Klasse Random definiert. Man erzeugt sich zunächst ein neues Objekt r dieser Klasse mit irgendeinem Anfangswert, z.B. 51: Random r = new Random(51), und erhält dann mittels r.nextInt() die jeweils nächste ganzzahlige Zufallszahl.
Da man bei jedem Programmlauf nicht immer die gleiche Folge verwenden möchte, sollte man einen zufälligen Startwert wählen, typischerweise die aktuelle Systemzeit

```
System.currentTimeMillis(),   also:
Random r = new Random(System.currentTimeMillis());
```

Die nächste Zahl transformiert man in das Intervall $0 \leq x < max$ durch Math.abs(r.nextInt()) % max , wobei man die Absolutfunktion abs aus der ebenfalls vordefinierten Klasse Math verwendet.]

Verwendung in einer Methode zur Erzeugung eines Felds von ganzzahligen Zufallszahlen mittels "import java.util.*;":

```
public int[] erzeugeZufallsfeld (int anzahl, int max) {
    int[] a = new int [anzahl];
    Random r = new Random(System.currentTimeMillis());
    for (int m = 0; m < anzahl; m++)
        a[m] = Math.abs(r.nextInt( )) % max;
                                    //Zufallszahl von 0 bis Max-1
    return a;                       //Math.abs = Absolutbetrag
}
```

Diese Methode kann z. B. wie folgt benutzt werden, um ein Feld von 150 Zahlen zwischen 0 und 1999 zu erzeugen:

```
int n = 150;
int[] Z = new int [n];
Z = erzeugeZufallsfeld (n, 2000);
```

1.3.1.b Sortieren durch Mischen.

Man schreibe ein Java-Programm, welches ein Feld ganzer Zahlen durch Mischen sortiert.

Hierbei greift man auf das Verschmelzen von bereits sortierten Feldern zurück, siehe Grundvorlesung "Einführung in die Informatik II". Die Anzahl der Vergleiche ist auch im schlechtesten Fall durch $n \cdot \log(n)$ nach oben beschränkt. Wir erläutern das Verfahren nicht noch einmal, sondern übertragen den Algorithmus aus der Grundvorlesung direkt nach Java.

Zunächst geben wir ein zu sortierendes Feld mit 15 Zahlen fest vor. Anschließend wird die Veränderung beschrieben, die nötig ist, um ein Feld aus n Zufallszahlen zwischen 0 und max-1 zu erzeugen.

```
import java.util.*;            // notwendig für das Objekt Random, s.u.
/**
 * VC, Uni Stuttgart, 22.10.07,
 * vergleiche Grundvorlesung Sommersemester 2007, Abschnitt 10.5.3
 **/

public class Sort {
   /**
    * In main die Zahlenfolge festlegen, „sortieren“ aufrufen, ausdrucken.
    * Ungünstigen Fall wählen: 2^4-1 = 15 Elemente im Feld.
    **/
   public static void main (String [] args) {
      int[] Z = {3, 6, 1, 23, 39, 64, 5, 22, 7, 2, 23, 17, 8, 7, 10};
      sortieren_durch_Mischen (Z, 0, Z.length-1);
      for (int m = 0; m < Z.length; m++)
         System.out.println (((m < 10) ? "  " : "") + m + " :  " + Z[m]);
   }
```

```
/**
 * Das Zahlenfeld A[links..rechts] nach der Größe aufsteigend sortieren.
 **/
public static void sortieren_durch_Mischen (int [] A, int links, int rechts) {
    if (rechts <= links) return;
    else {              //rekursiv absteigen und dann bottom up verschmelzen
        int mitte = (links + rechts)/2;
        sortieren_durch_Mischen (A, links, mitte);
        sortieren_durch_Mischen (A, mitte+1, rechts);
        verschmelze (A, links, mitte, rechts);
    }
}

/**
 * Verschmelze zwei sortierte Folgen A[links..mitte] und
A[mitte+1..rechts]
 * zu einer sortierten Folge B[0..rechts-links]; anschließend B
 * zurückspeichern nach A
 **/
```

```
    private static void verschmelze (int [] A, int links, int mitte, int rechts) {
        assert (links <= mitte) && (mitte < rechts);
        int [] B = new int [rechts-links+1];
        int i = links;
        int j = mitte+1;
        int k = 0;                                  // stets das kleinere Element nach B
        while ((i <= mitte) && (j <= rechts)) {
            if (A[i] <= A[j]) {B[k] = A[i]; i ++;}
            else {B[k] = A[j]; j ++;}
            k ++;
        }                   // restliche Elemente des verbliebenen Feldes anhängen
        if (i > mitte)
            while (j <= rechts) { B[k] = A[j]; j ++; k ++; }
        else
            while (i <= mitte) { B[k] = A[i]; i ++; k ++; }
        for (int m = 0; m < k; m++)  A[links+m] = B[m];      //zurück nach A
        return;
    }
}
```

Ausgabe des obigen Programms (das Feld Z = {3, 6, 1, 23, 39, 64, 5, 22, 7, 2, 23, 17, 8, 7, 10} wurde dort fest vorgegeben):

```
 0 :  1
 1 :  2
 2 :  3
 3 :  5
 4 :  6
 5 :  7
 6 :  7
 7 :  8
 8 :  10
 9 :  17
10 :  22
11 :  23
12 :  23
13 :  39
14 :  64
```

Wir übernehmen nun obige Methode
int [ ] erzeugeZufallsfeld (int anzahl, int max),
um das Programm mit beliebigen Feldern testen zu können.

Füge also die Klassenmethode

```
static int[] erzeugeZufallsfeld (int anzahl, int max) {
    int[] a = new int [anzahl];
    Random r = new Random(System.currentTimeMillis());
    for (int m = 0; m < anzahl; m++)
        a[m] = Math.abs(r.nextInt( )) % max;
                                        //Zufallszahl von 0 bis max-1
    return a;                           //Math.abs = Absolutbetrag
}
```

hinzu und ersetze die Zeile

```
int[] Z = {3, 6, 1, 23, 39, 64, 5, 22, 7, 2, 23, 17, 8, 7, 10};
```

durch ein zufällig erzeugtes Feld mit n Elementen:

```
int n = 47;
int[] Z = new int [n];
Z = erzeugeZufallsfeld (n, 2000);
```

Dann erhält man für n = 47 zum Beispiel folgende Ausgabe (mit modifiziertem Druckbild, da hier 5 Ergebnisse in einer Zeile stehen):

```
 0 :  39      1 :  40      2 :  67      3 :  118     4 :  132
 5 :  153     6 :  221     7 :  222     8 :  225     9 :  262
10 :  263    11 :  268    12 :  337    13 :  349    14 :  411
15 :  415    16 :  426    17 :  504    18 :  527    19 :  539
20 :  558    21 :  645    22 :  700    23 :  725    24 :  841
25 :  848    26 :  877    27 :  889    28 :  899    29 :  903
30 :  928    31 :  974    32 :  1004   33 :  1157   34 :  1182
35 :  1212   36 :  1299   37 :  1307   38 :  1452   39 :  1470
40 :  1483   41 :  1508   42 :  1621   43 :  1641   44 :  1697
45 :  1793   46 :  1922
```

1.3.2 Median in linearer Zeit

1.3.2.1 Aufgabenstellung:

Finde zu n (long-) Zahlen den Median; dies ist das Element, das nach dem Sortieren an der Stelle $(n+1)/2$ steht.

Definition: "Sort(k)"

Gegeben ist eine Folge von n ganzen Zahlen $A_1, A_2, ..., A_n$.
Das Element, das nach dem Sortieren an der Position k steht, bezeichnen wir mit Sort(k) der Folge A_i.

Eindeutige Charakterisierung des Wertes Sort(k):

$|\{ j \mid A_j \leq Sort(k) \}| \geq k$ und zugleich $|\{ j \mid A_j > Sort(k) \}| \leq n-k$.

Anmerkung: Sind alle A_i paarweise verschieden, so gilt statt "≥ k" und "≤ n-k" die Gleichheit "= k" und "= n-k"

Es gilt: Median = Sort((n+1)/2).

Zur Illustration: 145 Zahlen. Welches ist das Element Nr. 73?

2301, 4892, 8197, 7823, 6541, 2639, 7891, 6883, 9211, 6738,
3371, 10892, 4394, 13823, 11741, 2663, 4852, 3197, 7623,
7841, 6383, 10512, 6938, 4092, 8144, 7823, 6741, 2639, 7391,
6884, 9291, 6735, 5171, 10892, 4994, 13623, 12742, 2662,
4432, 3857, 5623, 10395, 2394, 1823, 1751, 2263, 4152, 3647,
7635, 7741, 6383, 1022, 6938, 4992, 8744, 4823, 6641, 7739,
5191, 6294, 4971, 7035, 6631, 11542, 4794, 1373, 15542,
2362, 4412, 3707, 5323, 5371, 4892, 4294, 1373, 11940, 2664,
4252, 3737, 7913, 7221, 6373, 11512, 6928, 4492, 2144, 7433,
6641, 12799, 7341, 6284, 9201, 4735, 5441, 10852, 4984,
12223, 11741, 2632, 2432, 3657, 5629, 10355, 4394, 1823,
1751, 7263, 4452, 6647, 8645, 7641, 6383, 1322, 3938, 4022,
8441, 4323, 6941, 7832, 5121, 6354, 4931, 7235, 6431, 9542,
1794, 3273, 4542, 2662, 4812, 2707, 8323, 6484, 9251, 3795,
5071, 6362, 4812, 2747, 5422, 5371, 1592, 4294, 2723, 6242.

1.3.2.2 Verfahren, um Sort(k) für n Elemente zu bestimmen

Vorgehen 1: Sortiere die Folge und nimm anschließend das Element an der Stelle k.
Zeitaufwand hierfür $O(n \cdot \log(n) + n) = O(n \cdot \log(n))$.
(Den Median von 1 Billionen Elementen zu finden, erfordert dann ungefähr 40 Billionen Vergleiche.)

Vorgehen 2: Führe einen Teilsortierschritt durch und mache dann mit dem Bereich, in dem der Median liegen muss, rekursiv weiter. Dies ist nur im Mittel ein O(n)-Verfahren; im schlimmsten Fall dauert es $O(n^2)$ Schritte (wie bei Quicksort).

Vorgehen 3: Berechne den Median der Fünfer-Mediane, führe mit diesem Wert einen Teilsortierschritt durch und mache mit dem Bereich, in dem der Median liegen muss, genau so weiter. Dies erfordert nur O(n) Schritte, allerdings ist die Konstante relativ groß, sodass es sich nur für sehr große n lohnt.

1.3.2.3 Wir betrachten das Verfahren 2.

Was ist ein Teilsortierschritt (= "Quicksortschritt") ?

Teilsortierschritt: Ergebnis sind 2 Teilfolgen, deren Elemente alle kleiner gleich p oder alle größer gleich p sind.

```
public void Teilsortier (long [] A, int L, int R) {
long p, hilf; int i = L; int j = R;
if (i < j) {
 p := A[(i+j)/2];
  while (i <= j) {
     while (A[i] < p)  i ++;
     while (A[j] > p)   j --;
     if (i <= j) { hilf = A[i]; A[i] = A[j]; A[j] = hilf; i++; j--; }
    }
  }
}        // suche weiter in A[L..j] oder in A[j+1..R]
```

Das Vorgehen ist nun offensichtlich:

Es sei n = A.length die Zahl der Elemente, für die Sort(k) bestimmt werden soll, $1 \leq k \leq n$. Es sei *min* eine kleine Zahl, bis zu der es einfacher ist, das k-te Element direkt zu bestimmen (in der Praxis: *min* = 5).

Falls höchstens *min* Elemente vorliegen (also $n \leq min$), so berechne man Sort(k) direkt, anderenfalls führe man einen Teilsortierschritt durch; hierbei wird das Feld A in ein Feld mit h (= j - L + 1) Elementen und in ein Feld mit (n-h) Elementen zerlegt. Falls $k \leq h$ ist, so suche man im ersten Feld nach dem k-ten Element, anderenfalls im zweiten Feld nach dem (k-h)-ten Element.

Zur Implementierung: Wir wählen alle Variablen für die Berechnung global. Anfangs ist L=0 und R=A.length-1.

```
public class Sortk {                          // k, min, A werden fest gewählt
  static int min = 5; static int L, R, k;
  static long [] A;
  static void iteriereTeilsortieren() {    // k, A, L und R sind hier global
    long p, hilf;  int h;  int i = L;  int j = R;
    if (R - L > min)  {
      p = A[(i+j)/2];
      while (i <= j) {
        while (A[i] < p)  i ++;
        while (A[j] > p)  j --;
        if (i <= j) { hilf = A[i]; A[i] = A[j]; A[j] = hilf; i++; j--; }
      }
      h = j - L + 1;
      if (h < k) {k = k - h; L = j + 1;} else R = j;
      iteriereTeilsortieren();
    }
    else direktFinden();
  }
```

```
static void direktFinden() {          // k, A, L und R sind global
    long m;                           // Bubblesort (es geht noch besser!)
    for (int i = L; i < R; i++)
        for (int j = i+1; j <= R; j++)
            if (A[j-1] > A[j]) {m=A[j]; A[j]=A[j-1]; A[j-1]=m;}
    if ((L <= R) && (1 <= k) && (k <= R-L+1))  {
        System.out.println("Das gesuchte Element ist "+A[L+k-1]+".");
        }
    else System.out.println("Falsche Eingabewerte.");
}
public static void main (String [] args) {
    long [] B = {23, 10, 15, 12, 1, 25, 9, 2, 24, 13, 16, 19, 28, 6, 8,
                 27, 3, 4, 21, 0, 26, 11, 18, 14, 17, 20, 7, 30, 29, 22, 5};
    A = B; L = 0; R = A.length-1; k = (A.length+1)/2;
    System.out.println(A.length+" Elemente, gesucht ist das "+k+"-te.");
    iteriereTeilsortieren ();
    }
}
```

Alle Initialisierungen wurden hier in die Methode main verlagert.

1.3.2.4 Verfahren 3: "Median der Fünfer-Mediane".

Ein schnelles rekursives Verfahren, um Sort(k) zu bestimmen:

Man iteriert Teilsortierschritte, wählt aber als Element p stets ein Element, welches garantiert, dass die Teilfolge, die weiter untersucht werden muss, höchstens $a \cdot n$ Elemente besitzt mit $0 < a \leq 7/10$.

Dies erreicht man, indem man die mittleren Elemente von Teilfolgen der Länge 5 nimmt und deren Median bestimmt.

Es folgt die genaue Beschreibung des Vorgehens:

Der Kopf der Methode, um Sort(k) zu berechnen, sei

public long **fuenfMed**(long [] A, int k).

A[links..rechts] sei ein Feld mit m=rechts-links+1 Elementen.

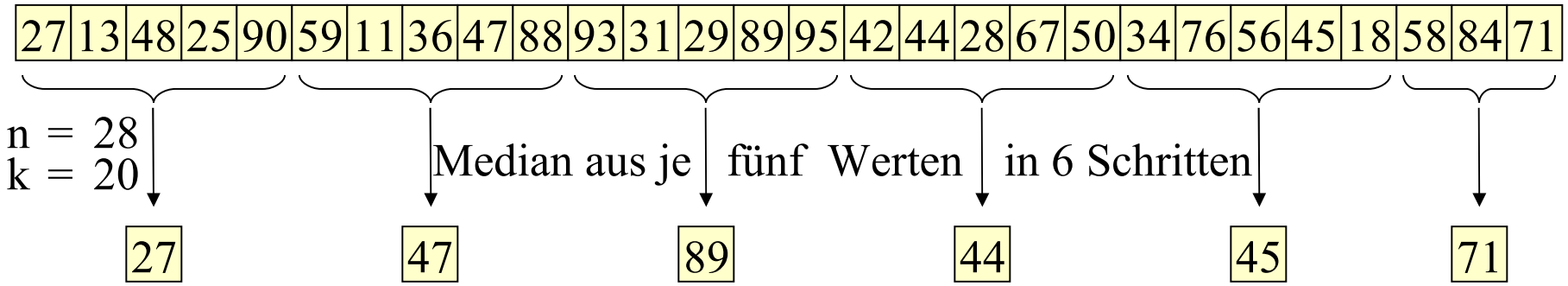
Falls m <= 5 ist, dann bestimme das k-te Element direkt.

Anderenfalls bilde B[0..(m-1)/5] aus A, wobei B[i] das mittlere Element der fünf Elemente A[links+5*i],...,A[links+5*i+4] für i = 0, ..., (m-1)/5-1 und B[m/5] das mittlere Element der Elemente A[links+5*(m-1)/5)],...,A[rechts] ist. Rufe rekursiv für anzahl = (m-1)/5 +1 auf:

p = **fuenfMed** (B, (anzahl+1)/2)

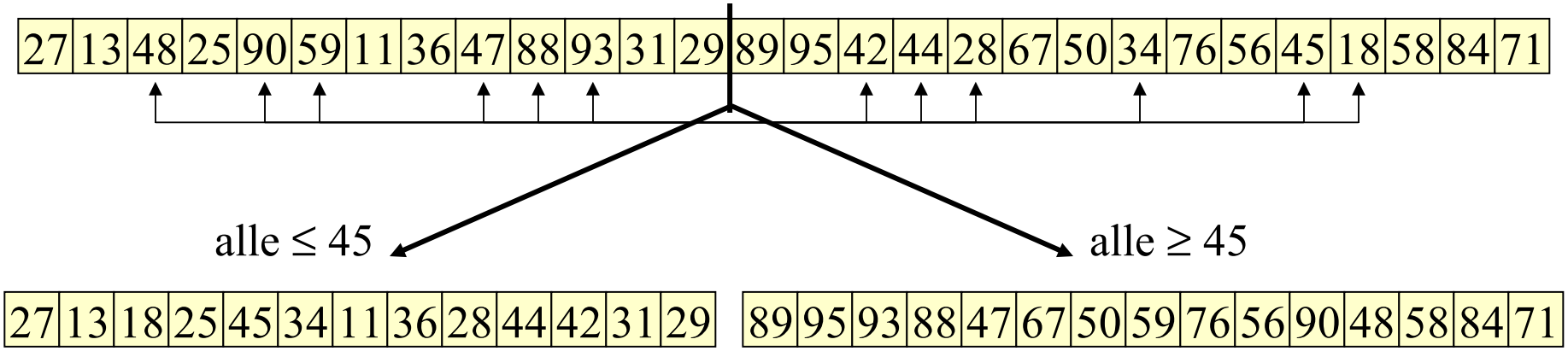
Dies liefert den "Median der Fünfer-Mediane". Nun führe mit diesem p einen Teilsortierschritt durch und rufe dann rekursiv **fuenfMed** mit dem Teil des Feldes auf, in dem Sort(k) liegen muss.

Gegeben ist eine ungeordnete Folge mit 28 Werten. Suche den Wert Sort(20).



Nun aus diesen ⌈n/5⌉-tel = 6 Werten Sort((⌈n/5⌉+1)/2) bestimmen. Ergebnis: 45.

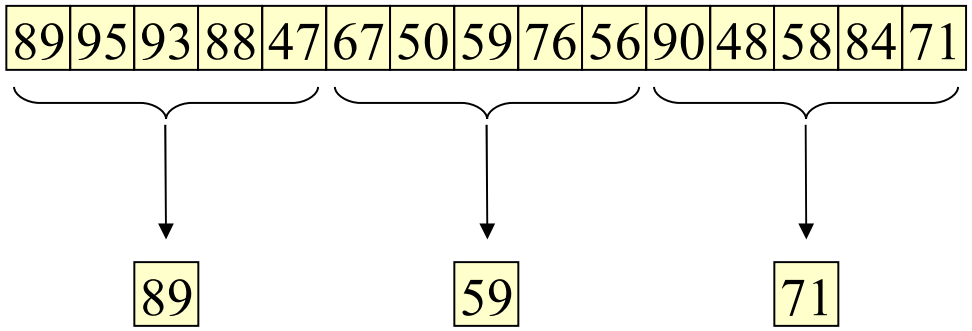
Mit dieser 45 einen Teilsortierschritt in der ursprünglichen Folge durchführen.



Dies sind 13 Elemente. Wegen 20 > 13 muss man also rechts weitersuchen.

Suche *hier* das Element Sort(20-13) = Sort(7).

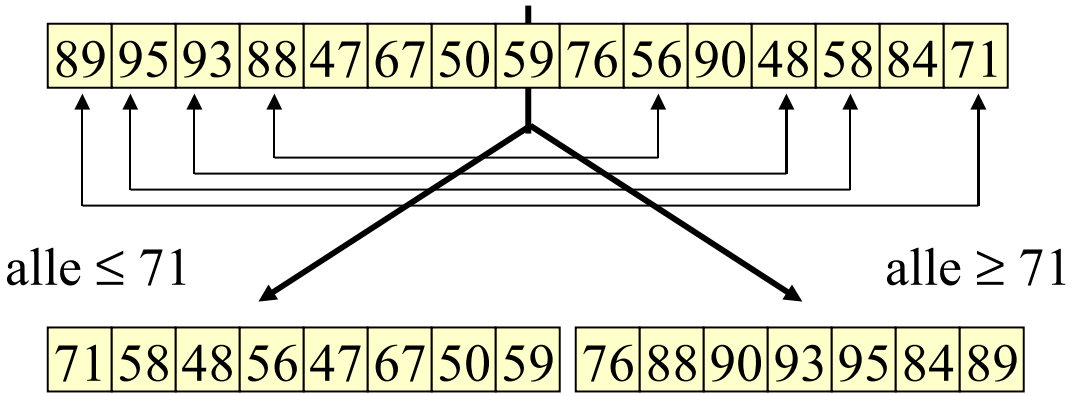
Weitermachen mit der Folge aus 15 Werten. Suche hier den Wert Sort(7).



$n = 15$
$k = 7$

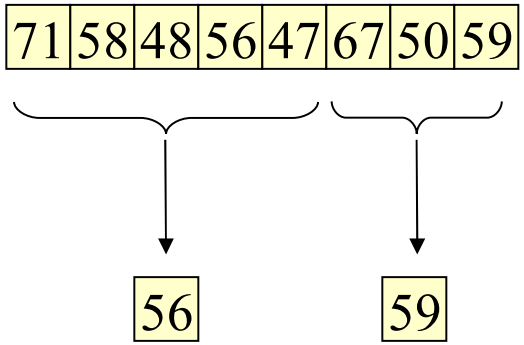
Aus diesen $\lceil n/5 \rceil$-tel $= 3$ Werten Sort($(\lceil n/5 \rceil+1)/2$)=Sort(2)=71 direkt bestimmen.

Mit dieser 71 einen Teilsortierschritt in der obigen Folge durchführen.



Dies sind 8 Elemente. Wegen $k = 7 \leq 8$ muss man also links weitersuchen nach dem Element Sort(7).

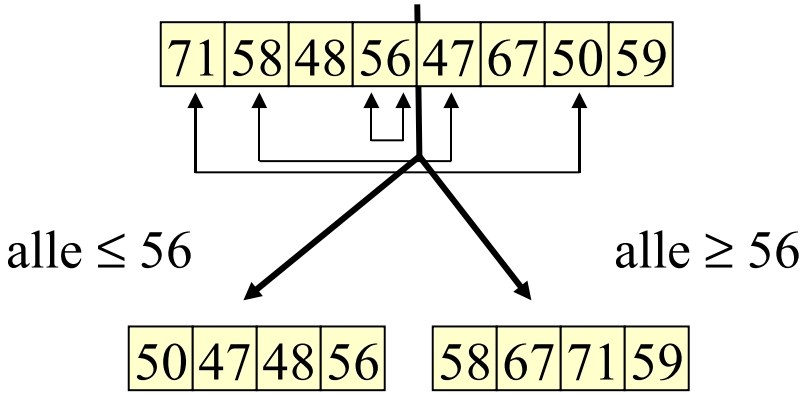
Weitermachen mit der Folge aus 8 Werten. Suche hier den Wert Sort(7).



n = 8
k = 7

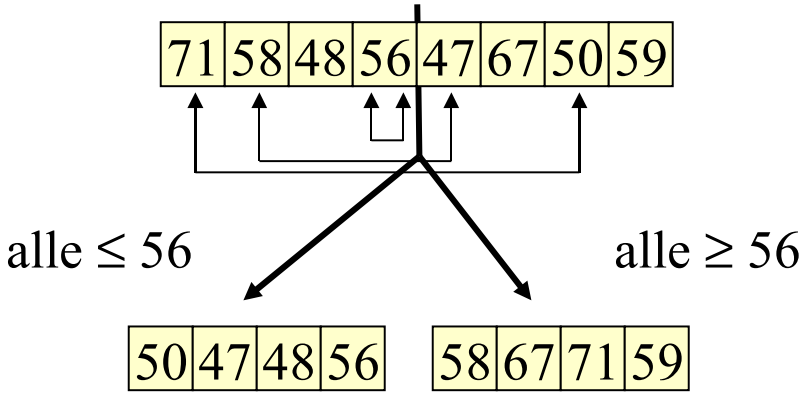
Aus diesen ⌈n/5⌉-tel = 2 Werten Sort((⌈n/5⌉+1)/2)=Sort(1)=56 direkt bestimmen.

Mit dieser 56 einen Teilsortierschritt in der obigen Folge durchführen.



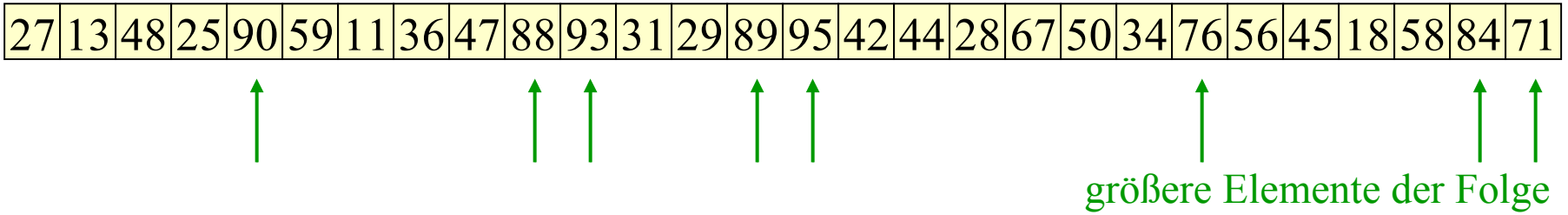
Links stehen 4 Elemente. Wegen k = 7 > 4 muss man also rechts weitersuchen nach dem Element Sort(7-4) = Sort(3).

Da rechts aber nur noch 4 Elemente stehen, also weniger als 5, können bzw. müssen wir Sort(3) = 67 direkt bestimmen.
Somit ist 67 das 20. Element der ursprünglichen Folge.



Links stehen 4 Elemente. Wegen $k = 7 > 4$ muss man also rechts weitersuchen nach dem Element Sort(7-4) = Sort(3).

Nun können wir noch die Charakterisierung von Sort(k) demonstrieren, indem wir die Probe machen, ob das gesuchte Element tatsächlich die Zahl 67 gewesen ist. Wir laufen also mit Sort(k) = 67 durch das Feld. Es muss gelten: $|\{j \mid A_j \leq \text{Sort}(k)\}| \geq k$ und $\{j \mid A_j > \text{Sort}(k)\}| \leq n\text{-}k$.



Es sind n = 28 und k = 20 sowie
$|\{j \mid A_j \leq \text{Sort}(k)\}| = 20 \geq k$ und $|\{j \mid A_j > \text{Sort}(k)\}| = 8 \leq n - k = 8$.
Die Charakterisierung von Sort(20) ist erfüllt und 67 ist somit tatsächlich das 20. Element der sortierten Folge.

Nun kommen wir zur Komplexität dieses Verfahrens.

Behauptung: Nimmt man als Pivot-Element p den Median der Fünfer-Mediane und sind alle Elemente der Folge verschieden, so besitzt nach dem Teilsortierschritt jedes der beiden Teilfelder mindestens $3 \cdot n/10 - 1$ und höchstens $7 \cdot n/10 + 1$ Elemente.

Beweis: Wir denken uns die Fünfer-Mediane $m_1, m_2, ..., m_{n/5}$ geordnet (Pivot-Element $p = m_r$ mit $r = n/10$). Für jedes i sei m_i der Median von fünf Elementen e_i, f_i, m_i, g_i, h_i der ursprünglichen Folge, dann ist o.B.d.A. $e_i \le f_i \le m_i \le g_i \le h_i$. Wichtig: Zu m_i gibt es zwei Elemente e_i und f_i, die kleiner oder gleich m_i sind. Wegen $m_1 \le m_2 \le ... \le m_r$ sind $e_1, ..., e_r, f_1, ..., f_r, m_1, ..., m_{r-1}$ kleiner oder gleich $p = m_r$. Dies sind genau $3n/10 - 1$ Elemente. Sind alle Elemente verschieden, so liegen diese im linken Teilfeld. Analog für die Elemente, die größer oder gleich m_r sind. Daraus folgt die Behauptung. Erläuternde Skizze:

e_1	e_2	e_3	...	e_{r-1}	e_r	e_{r+1}	e_{r+2}	...	$e_{n/5}$
f_1	f_2	f_3	...	f_{r-1}	f_r	f_{r+1}	f_{r+2}	...	$f_{n/5}$
m_1	m_2	m_3	...	m_{r-1}	m_r	m_{r+1}	m_{r+2}	...	$m_{n/5}$
g_1	g_2	g_3	...	g_{r-1}	g_r	g_{r+1}	g_{r+2}	...	$g_{n/5}$
h_1	h_2	h_3	...	h_{r-1}	h_r	h_{r+1}	h_{r+2}	...	$h_{n/5}$

Farbig unterlegt sind jeweils $3 \cdot n/10 - 1$ Elemente. (Untersuchen Sie den Fall, dass gleiche Elemente in der Folge auftreten.)

1.3.2.5 Zeitkomplexität: Nun können wir die Zahl der Vergleiche T(n) dieses Verfahrens berechnen. Wir ersetzen hier (nicht ganz korrekt) $\lceil n/5 \rceil$ durch n/5.

Das Verfahren verläuft folgendermaßen:

Berechne zu jedem Fünferblock der Folge den Median (in 6 Schritten!) und bilde die Folge dieser Fünfer-Mediane (dies sind n/5 Elemente).

Berechne rekursiv von dieser Folge den Median p.

Führe mit diesem Element p auf der ursprünglichen Folge einen Teilsortierschritt durch.

Stelle fest, in welcher Teilfolge das Element Sort(k) liegen muss und fahre mit dieser Folge rekursiv fort, bis weniger als 6 Elemente übrig sind (dann bestimme das gesuchte Element direkt).

$$T(n) \leq 6 \cdot n/5 + T(n/5) + n + T(7 \cdot n/10)$$

Zahl der Vergleiche ≈ Zeit

Für die Anzahl der Vergleiche gilt also höchstens

$T(n) = 11 \cdot n/5 + T(n/5) + T(7 \cdot n/10)$ mit $T(5) = 6$.

Die Lösung dieser Gleichung ist bekanntlich von der Form

$$T(n) = a \cdot n + b$$

für geeignete Zahlen a und b (weil $1/5 + 7/10 < 1$ ist).

Aufgabe für Sie: Überzeugen Sie sich, dass das Verfahren tatsächlich den Median findet und berechnen Sie die genauen Konstanten a und b. (Einfach einsetzen und Gleichung lösen.)

Hinweis. Es gilt: $T(n) < 22 \cdot n$, in der Praxis $T(n) <\approx 16 \cdot n$. Genaue Berechnung: siehe Literatur oder Vorlesung EA^2 im 6. Semester.

Durch Modifizierung des Verfahrens kann sogar $T(n) < 8 \cdot n$ erreicht werden.

Aufgabe:

a) Schreiben Sie das zugehörige Java-Programm.

b) Berechnen Sie die Erwartungswerte für a und b experimentell mit diesem Programm.

Hinweis: Zur Berechnung des Medians finden Sie ein Ada-Programm, in dem allerdings "die Berechnung der mittleren Elemente von Teilfolgen der Länge 5" nicht korrekt ausprogrammiert wurde, unter: http://www.fmi.uni-stuttgart.de/fk/lehre/ss07/eaa/

1.3.2.6 Hinweis zum Programm

```
class Sort_k {
   long[] A = new long[];  int k;
   Sort_k (long[] Z, int k) { ... }     // kopiere Z nach A
   public long fuenfMed (long [] A, int k) {
      int m = (A.length+4)/5;
      if (m > 5 ) {              // B ermitteln und dann rekursiv weiter rechnen
         long[] B = new long[m];  ...;
         return fuenfMed (B,(m+1)/2); }
      else { ... }               // direkt berechnen
   }
   int trennindex;
   public void quicksortschritt (long[] Z, long pivot, int L, int R) { ... }
   long p = fuenfMed (A, (A.length+1)/2);
   quicksortschritt(A,p,links,rechts);   // trennindex) wird mitberechnet
   if trennindex >= k {if k == 0 { ... } else {...; new Sort_k (A[links..trennindex-1], k); ... }}
   else {k = k-trennindex; if k == 0{ ... } else {new Sort_k (A[trennindex..rechts], k); ... }
   ....
   public static void main ...    // Feld und k einlesen,  new Sort_k hiermit starten
}
```

1.3.2.7 Die "Paarweise Ungleichheit"

Hinweis auf ein ähnlich aussehendes Problem, das

EDP = element distinctness problem:

Gegeben ist eine Folge $A_1, A_2, ..., A_n$. Stelle fest, ob die Elemente paarweise verschieden sind.

Nahe liegendes Verfahren: Folge sortieren und danach in einem Durchlauf feststellen, ob ein Element doppelt vorkommt. Aufwand: $O(n \cdot \log(n))$ Vergleiche. Mit dem Algorithmus zur Lösung des Plateau-Problems (siehe Grundvorlesung Kapitel "Semantik" 7.3.2) kann man dann sogar mit dem gleichen Aufwand für jedes $1 \leq k \leq n$ berechnen, wie viele verschiedene Elemente es gibt, die genau k-mal in der Folge enthalten sind. Das Problem EDP ist also nicht schwerer zu lösen als das Sortieren einer Folge. Es *könnte* aber sein, dass es auch zum EDP einen Linearzeit-Algorithmus gibt. Diese Frage ist bisher noch ungelöst.

Zur Illustration: 176 Zahlen. Gibt es zwei gleiche Zahlen?

5361, 2892, 1192, 7923, 6541, 2639, 7491, 6883, 9211, 7888, 6738,
5336, 10812, 4394, 13823, 11841, 2663, 4852, 3197, 234, 7623, 898,
1841, 6383, 10512, 6937, 4092, 7981, 8144, 7823, 6741, 2637, 7391,
4884, 549, 9291, 6735, 459, 5171, 10892, 4994, 13623, 12742, 2762,
9432, 3857, 5623, 10395, 9544, 2394, 1822, 1451, 2263, 4152, 3647,
4635, 7741, 6393, 7872, 1022, 6938, 4992, 8744, 4823, 6541, 7739,
8891, 559, 6294, 4971, 3451, 7035, 6631, 11542, 4794, 1473, 15542,
12362, 4412, 3707, 5323, 5391, 4191, 4294, 1373, 671, 11940, 2664,
4252, 3737, 7913, 7221, 6373, 6711, 11512, 6928, 4492, 2144, 7433,
6641, 12799, 7341, 3805, 7019, 6284, 9201, 4735, 5441, 10852,
4984,
12223, 11741, 2632, 3618, 646, 2432, 3657, 5629, 10355, 4394,
1823,
1751, 7263, 4452, 6647, 8645, 7641, 2674, 2383, 12322, 3938, 4022,
8441, 4323, 6941, 7832, 5121, 9340, 6354, 4931, 7235, 6031, 9542,
1794, 3273, 4542, 2662, 4822, 2524, 2707, 8323, 6484, 9251, 3795,
5071, 6362, 4812, 2747, 5422, 5372, 10432, 1592, 4694, 2723, 6242,
5232, 1842, 7131, 6429, 5023, 6734, 8601, 9945, 6766, 3010, 7824.

1.4 Java, Teil 2

1.4.1 Listen (Objekt-Referenzen)

1.4.2 Einfache Operationen mit Objekten

1.4.3 Zusicherungen und Ausnahmen

1.4.4 Interfaces

1.4.5 abstrakte Klassen

1.4.6 Threads

1.4.1 Listen, Referenzen (vgl. Grundvorlesung Kapitel 3)

In Ada haben wir für dynamische Datenstrukturen explizite Referenzen (Pointer, Zeiger) verwendet. Datenstrukturen können stattdessen auch rekursiv definiert werden, wie dies z.B. in der BNF geschieht, wenn man eine Folge von Größen des Typs <element> definiert:

<liste> ::= null | < element> <liste>

In dieser Weise geht man auch in Java vor, wobei man wie üblich Selektoren verwenden muss, und definiert

class Liste {Element name; Liste naechstes;}

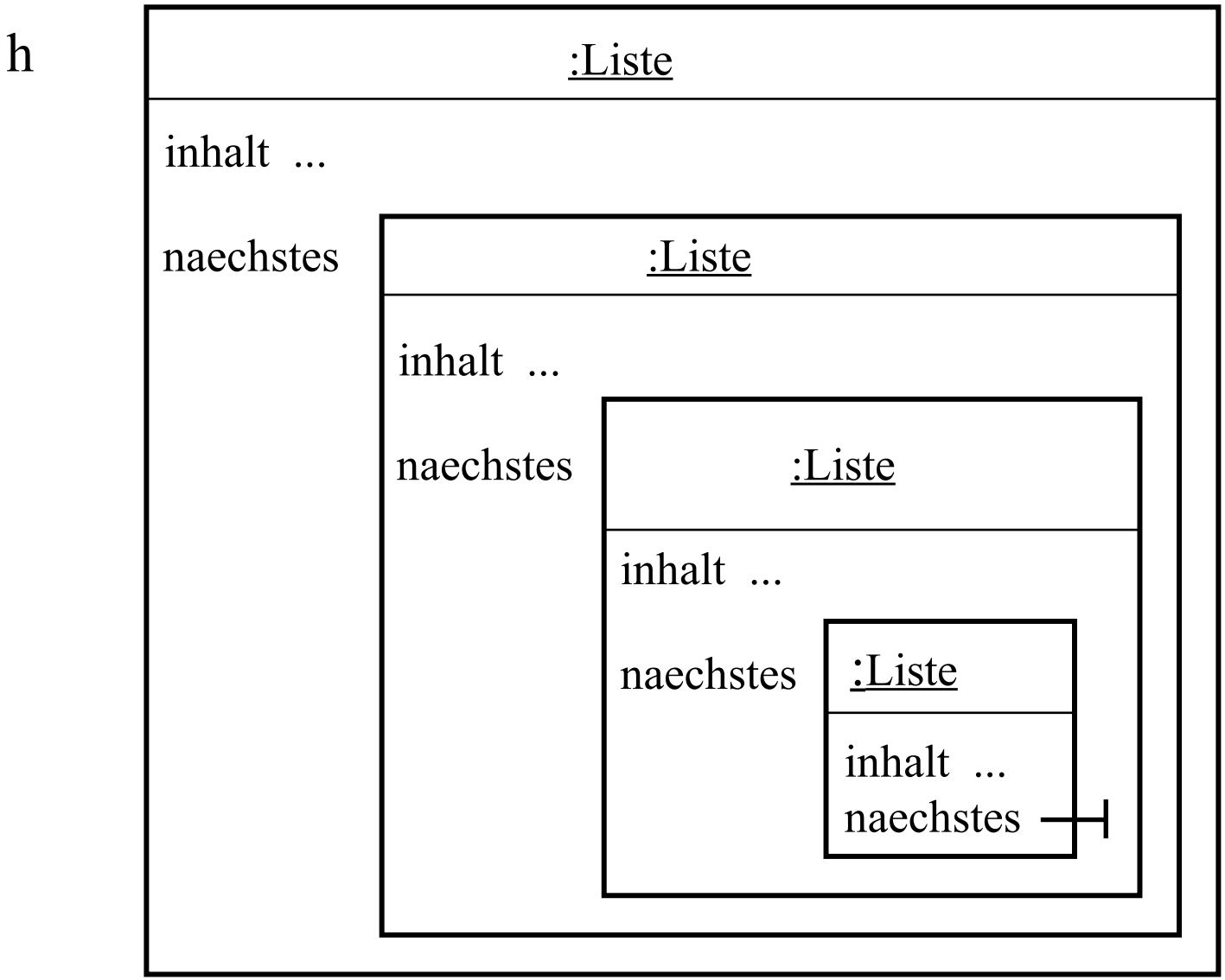
Den Fall "null" behandelt man bei der Initialisierung mit dem Konstruktor oder mit einer anderen Wertzuweisung.

Vollständige Beschreibung solcher Objekte in Java:

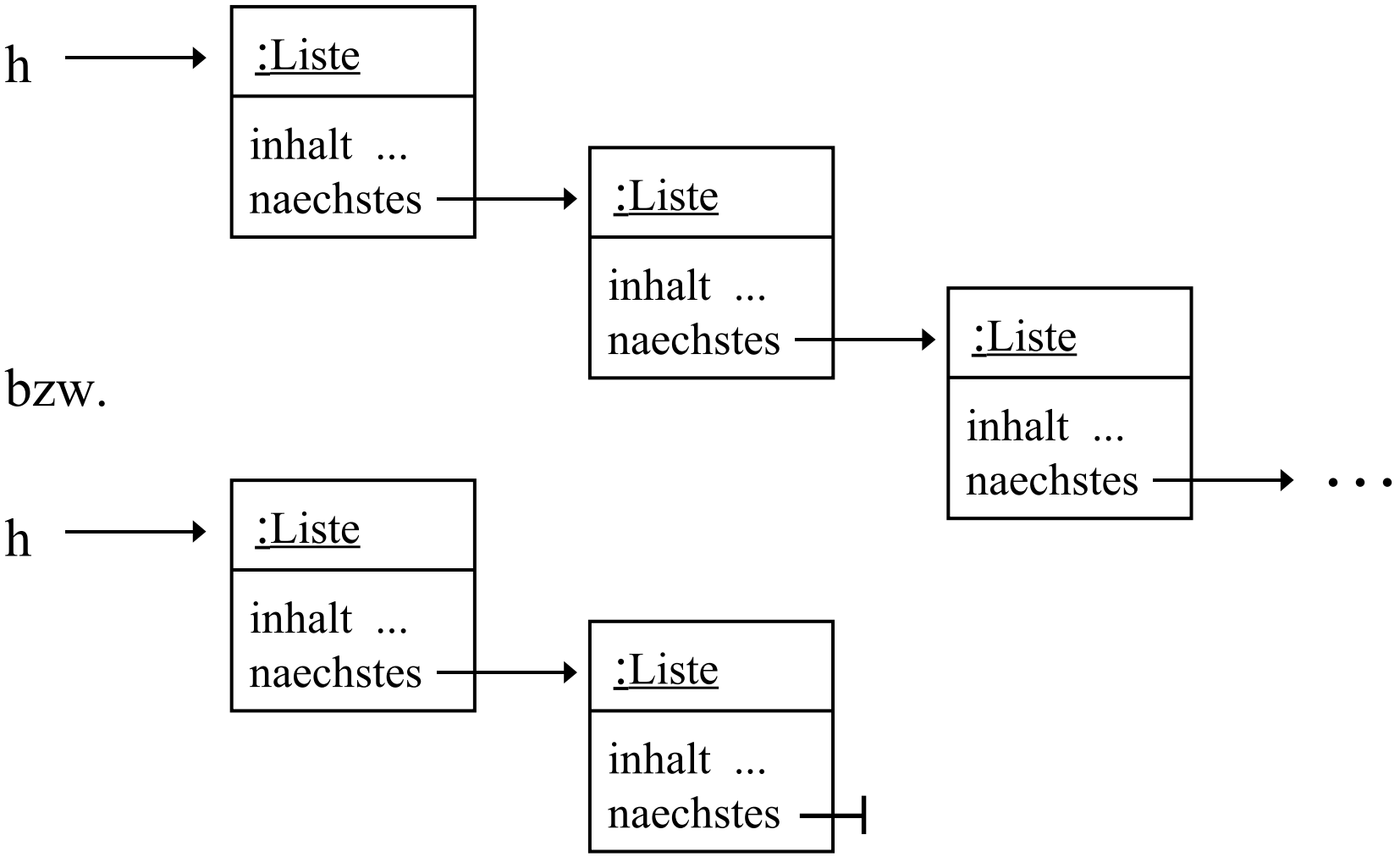
```
class Liste {
        Element inhalt;
        Liste naechstes;
        Liste (Element e, Liste f) {
                inhalt = e;
                naechstes = f;
        }
}
```

Die Rekursion zeigt sich daran, dass im Objekt der Klasse Liste erneut ein Objekt der Klasse Liste (mit Selektor "naechstes") enthalten ist. Dies führt zu einer prinzipiell unendlichen Struktur, die durch ein begrenzendes Objekt (bei Listen: null) zu einer endlichen Struktur wird.

Liste h; kann man sich wie folgt veranschaulichen
(:Liste steht für "Klasse mit Namen Liste"):



Liste h;
kann man sich auch mit Hilfe von Zeigern veranschaulichen:



Beispiel: Monatsnamen ausgeben

Die zwölf Monatsnamen sollen in einer Liste gespeichert und in umgekehrter Reihenfolge wieder ausgegeben werden.

```
public class Listenbeispiel1 {                    // VC, Informatik III, Beispiel für Listen
   String Monatsnamen [] = new String [12];
   public void setzeNamen () {                    // Anstelle einer Eingabe
      Monatsnamen [0] = "Januar";
      Monatsnamen [1] = "Februar";
      Monatsnamen [2] = "März";
      Monatsnamen [3] = "April";
      Monatsnamen [4] = "Mai";
      Monatsnamen [5] = "Juni";
      Monatsnamen [6] = "Juli";
      Monatsnamen [7] = "August";
      Monatsnamen [8] = "September";
      Monatsnamen [9] = "Oktober";
      Monatsnamen [10] = "November";
      Monatsnamen [11] = "Dezember";
   }
```

```
class Monatsliste {    // rekursive Listenstruktur
    String name;
    Monatsliste folgemonat;
    Monatsliste(String wort,Monatsliste L) {
        name = wort;
        folgemonat = L;
    }
}
Listenbeispiel1 () {     // Konstruktor, in dem hier das gesamte Programm steckt
    Monatsliste monate = null;
    setzeNamen ();
    for (int i=0; i<=11; i++) {
        monate = new Monatsliste (Monatsnamen[i], monate);
    }
    Monatsliste naechster = monate;
    while (naechster != null) {  // Liste durchlaufen, bis "null" erreicht wird.
        System.out.println (naechster.name);
        naechster = naechster.folgemonat;
    }
}
public static void main (String args []){
    new Listenbeispiel1();
}
}
```

Als Ausgabe erhält man:

Dezember
November
Oktober
September
August
Juli
Juni
Mai
April
März
Februar
Januar

Analog kann man nun Bäume und allgemeine Graphen darstellen.

Wir setzen für die Übungen gewisse Kenntnisse voraus:
Zu Listen und ihren Operationen siehe Grundvorlesung 3.5,
zu Keller und Schlange siehe Grundvorlesung 3.5.4 und 3.5.5,
zu Bäumen siehe Grundvorlesung 3.7 und 8.2,
zu Graphen siehe Grundvorlesung 3.8 und 8.8.

Binäre Bäume kann man z. B. wie folgt darstellen:

```
class BinBaum {
        Element inhalt;
        Binbaum links, rechts;
        BinBaum (Element e) {
                BinBaum (e, null, null);
        }
        BinBaum (Element e, BinBaum l, BinBaum r) {
                inhalt = e;
                links = l;
                rechts = r;
        }
}
```

1.4.2 Einfache Operationen mit Objekten

new <Typ>	Erzeugt ein neues Objekt der Klasse <Typ> und gibt die Referenz auf dieses Objekt zurück. Beachten Sie, dass "new" zuerst alle Variablen erzeugt und erst danach den Konstruktor ausführt, siehe hierzu "class Lager" in 1.4.6.
this	Bezeichnet das aktuelle Objekt, in dem sich dieses Schlüsselwort befindet.
<Typ> v	Die Variable v kann als Wert nur Referenzen auf Objekte der Klasse <Typ> besitzen.
v1 = v2	Der neue Wert von v1 wird gleich dem Wert von v1. Wenn v1 und v2 Variablen für Objekte sind, so referenzieren v1 und v2 anschließend das gleiche Objekt.

v1 == v2	Prüft, ob die Werte der Variablen v1 und v2 gleich sind. Im Falle von Objekten bedeutet dies: Es wird geprüft, ob v1 und v2 das gleiche Objekt referenzieren. (Es wird also nicht die inhaltliche Gleichheit zweier Objekte geprüft.)

Möchten Sie von einem Objekt eine Kopie anlegen, so müssen Sie für die jeweilige Klasse eine eigene Methode schreiben, z. B. "clone ()". Sie sollten dann zuleich eine Boolesche Methode "equals (x)" definieren, um zu entscheiden, ob zwei Objekte bzgl. ihrer Inhalte gleich sind. Es sollte also Folgendes realisiert werden:

v1.equals(v2)	Prüft, ob die Objekte, auf die v1 und v2 verweisen, inhaltlich übereinstimmen.
v1 = v2.clone()	Kopiert das Objekt, auf das v2 zeigt, und weist v1 die Referenz aiuf dieses neue Objekt zu. (Oder: v1 = (<Typ von v1>) v2.clone())

1.4.3 Korrektheit (Zusicherungen und Ausnahmen)

1.4.3.1 Zusicherungen (siehe 7.1.5 Grundvorlesung)

Eine Zusicherung in einem Programm ist eine prädikatenlogische Formel (oder eine Bedingung), die genau an dieser Stelle erfüllt sein muss. In Java schreibt man hierfür

```
assert < Bedingung > [: <Text>];
```

Beispiel:

```
int ggT (int a, int b) {
   assert (b > 0) : "zweiter Parameter ist nicht positiv";
   ...   // weiter programmieren wie üblich
}
```

Bedeutung: Trifft die Bedingung zu, wird die nächste Anweisung durchgeführt; trifft sie aber nicht zu, wird der Text (falls er fehlt: eine Standardsystemmeldung) ausgegeben.

"assert" dient vor allem der korrekten Programmentwicklung: Man streut ins Programm genau die Bedingungen ein, die hier entsprechend der Semantik gelten müssen. Diese lässt man auch im fertigen Programm stehen; sie belasten die Laufzeit nicht, da man alle Zusicherungen beim Programmstart mittels "-da" ausschalten, aber bei fehlerhaftem Verhalten mittels "-ea" für erneutes Testen wieder einschalten kann.

Zusicherungen sind vor allem wichtig, wenn die Werte eines Objekts oder aller Objekte einer Klasse Nebenbedingungen erfüllen müssen oder wenn man sicher sein will, dass eine Methode am Ende ihrer Berechnung auch tatsächlich das geforderte Ergebnis liefert. Zugleich führt man durch die Zusicherungen auch (wenigstens in Ansätzen) einen Korrektsheitsbeweis mit sich, wie dies in der Grundvorlesung (in Kapitel 7) behandelt wurde.

1.4.3.2 Ausnahmen ("exception", siehe 4.1.8 Grundvorlesung)

Die Sicht in Java ist die Folgende: Wenn man in einem Programmteil auf eine Ausnahme (= ein Ereignis, das den normalen Programmablauf stört) treffen könnte, so schließt man diesen Programmteil in

```
try { ...}
```

ein ("try-Block"). Die möglichen Ausnahmen können anschließend in catch-Blöcken abgefangen werden. Schema hierfür (der finally-Block darf nur folgen, falls k > 0 ist):

```
try {
   ... if (<Bedingung>)
      throw new <Ausnahmekonstruktor> (<Parameterliste>);
   ...}
catch (<Ausnahmetyp1> <Bezeichner1>)
   {<Anweisungsfolge zur Behandlung der Ausnahme1>}
catch (<Ausnahmetyp2> <Bezeichner2>)
   {<Anweisungsfolge zur Behandlung der Ausnahme2>}
...
catch (<Ausnahmetyp_k> <Bezeichner_k>)
   {<Anweisungsfolge zur Behandlung der Ausnahme_k>}
[finally {<Anweisungsfolge>}]
```

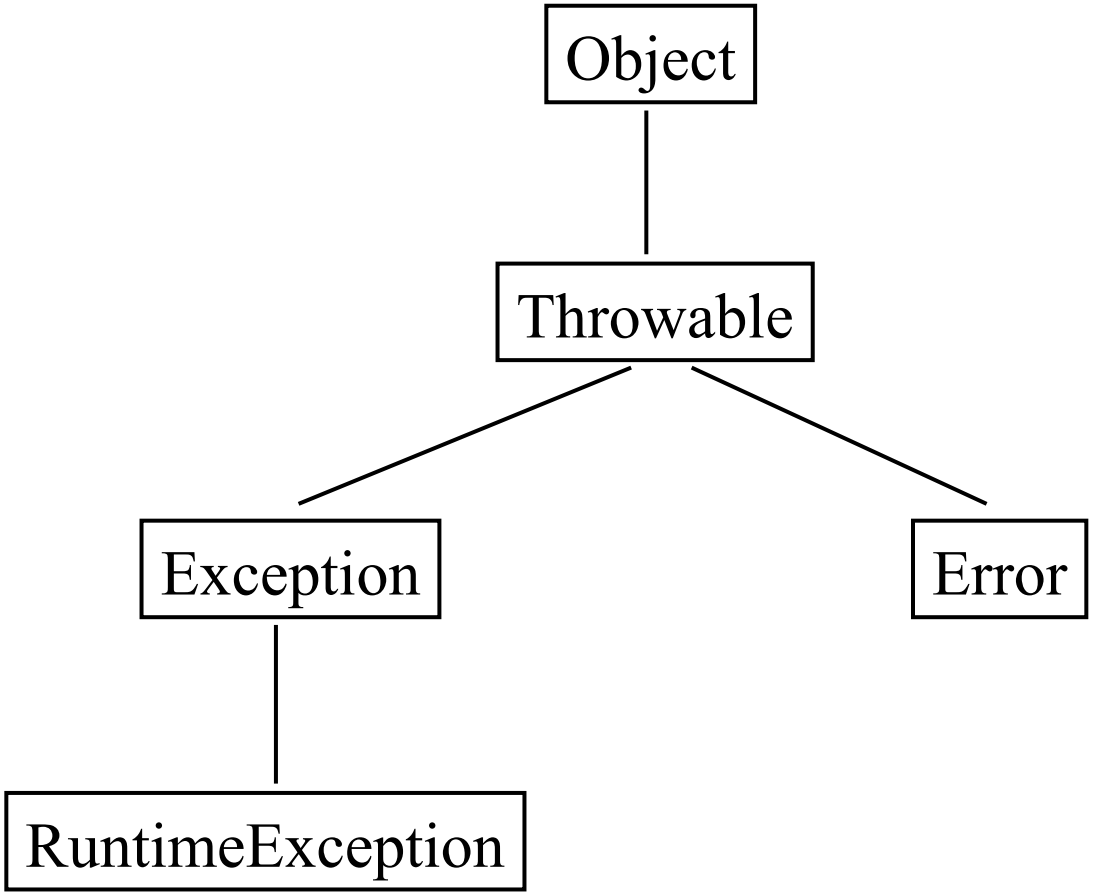
Bedeutung dieser Konstruktion:
Eine Ausnahme wird in Java als Objekt aufgefasst, die von Ereignissen (Situationen, Bedingungen) "geworfen" wird. Dies kann zum einen durch das Programm selbst geschehen, zum andern kann das Java-Laufzeitsystem der Auslöser (Division durch 0, Überschreiten von Indexgrenzen, Speicher knapp, ...) sein.
Das geworfene Ausnahmeobjekt wird dabei in der Regel neu erzeugt, daher steht meist "new" vor der Ausnahme. Dem Ausnahmeobjekt können (entsprechend dem Konstruktor in der zugehörigen Klasse) aktuelle Parameter mitgegeben werden.
Daraus folgt: Eine Ausnahme muss irgendwo als Klasse deklariert werden und an der Stelle, wo sie geworfen wird, sichtbar sein.
Die üblichen Ausnahmen sind vordefiniert (wie in Ada) und werden, falls sie eintreten, automatisch bei der Ausführung des Programms vom Laufzeitsystem "geworfen". Man kann sie in einem catch-Block "fangen" und auf diese Weise die vordefinierte Methode abändern und die konkrete Ausnahmebehandlung selbst festlegen. Andere Ausnahmen, die man selbst neu einführt, muss man als Klasse definieren.

Das geworfene Ausnahmeobjekt muss nun "gefangen" werden. Dies erledigt das Java-Laufzeitsystem, es sei denn, wir deklarieren den Block, in dem die Ausnahme stattfinden kann, als "try-Block" und fügen anschließend catch-Blöcke für gewisse Ausnahmen an. Hier wird dann eine eigene Ausnahmebehandlung durchgeführt. Dadurch wird die Kontrolle dann an eine aufzurufenden Methode (oder mittels return an die Stelle, wo die Ausnahme stattfand) weitergegeben.

Im Anschluss an den try- oder einen catch-Block-Bereich wird auf jeden Fall der finally-Block ausgeführt. Dies dient vor allem dazu, irgendwelche letzten "Aufräumarbeiten" noch ausführen zu können, bevor das Programm normal beendet oder abgebrochen wird.

Empfehlung: Probieren Sie diese Konstruktionen an selbst gewählten Beispielen aus. Machen Sie sich kundig, welche Ausnahmen in Java vordefiniert sind.

Eine Ausnahme-Klasse wird in Java als Unterklasse zur Klasse "Exception" definiert, die wiederum eine Unterklasse der Klasse "Throwable" (= "kann geworfen werden") ist. Die Klassen-hierarchie lautet:



Auf die Klasse "Error", in der Fehler, die zum Programmab-bruch führen sollten, behan-delt werden, gehen wir hier nicht ein.

Künstliches Beispiel: "AusnahmeTest"

Suche, ob der ggT von a und b im Feld zahlen enthalten ist. Hierzu sollen die Ausnahmen, dass man den ggT wegen $b \leq 0$ nicht bilden möchte (Ausnahme: NichtPositiv) und dass eine Zahl nicht im Feld zahlen vorkommt (Ausnahme: NichtGefunden), verwendet werden. Analysieren Sie das Beispielprogramm und machen Sie sich die anschließend aufgelistete Ausgabe klar.

```
class AusnahmeTest {  // Layout wegen Platzproblemen nur z.T. vorschriftsmäßig.

   static class NichtPositiv extends Exception {
      NichtPositiv (int j) {
         System.out.println ("Zahl " + j + " ist nicht positiv. Abbruch. ");
      }
   }

   static class NichtGefunden extends Exception {
      NichtGefunden ( ) { return; }
      NichtGefunden (int d) {System.out.println(d);}
   }
```

```
public static void main (String [] args) {
    int anfangA = 48; int b = 30; int endeB = 36;
    int x, y, r, i;   int [] zahlen = {1, 2, 4, 7, 9, 10, 12};
    while (b <= endeB) {
        x = anfangA; y = b;
        try {
            if (y <= 0) throw new NichtPositiv (y);
            System.out.print("Es geht los mit " + x + " und " + y);
            while (y != 0) {r = x%y; x = y; y = r;}
            System.out.println("; ihr ggT lautet " + x + ".");
            i = 0;   while ((i < zahlen.length) && (zahlen[i] != x)) i++;
            if (i == zahlen.length) throw new NichtGefunden (x);
            System.out.println("Index: " + i + ".");
        }
        catch (NichtPositiv e1) {
        System.out.println ("Der ggT von " + x + " und " + y + " wurde nicht berechnet.");
        }
        catch (NichtGefunden e) {
        System.out.println ("Der Wert der Exception " + e + " kommt im Feld nicht vor. ");
        }
        finally { b ++;
            if (b > endeB) System.out.println ("Ende.");
            else System.out.println("Neues b = " + b + ".");
} } } }
```

```
// Ausgabe (mit zahlen = {1, 2, 4, 7, 9, 10, 12}):
Es geht los mit 48 und 30; ihr ggT lautet 6.
6
Der Wert der Exception Ausnahme$NichtGefunden kommt im Feld nicht vor.
Neues b = 31.
Es geht los mit 48 und 31; ihr ggT lautet 1.
Index: 0.
Neues b = 32.
Es geht los mit 48 und 32; ihr ggT lautet 16.
16
Der Wert der Exception Ausnahme$NichtGefunden kommt im Feld nicht vor.
Neues b = 33.
Es geht los mit 48 und 33; ihr ggT lautet 3.
3
Der Wert der Exception Ausnahme$NichtGefunden kommt im Feld nicht vor.
Neues b = 34.
Es geht los mit 48 und 34; ihr ggT lautet 2.
Index: 1.
Neues b = 35.
Es geht los mit 48 und 35; ihr ggT lautet 1.
Index: 0.
Neues b = 36.
Es geht los mit 48 und 36; ihr ggT lautet 12.
Index: 6.
Ende.
```

Das Beispiel zeigt: Man kann Ausnahmen wie Abweichungen vom gewünschten Programmablauf behandeln und umgeht dadurch unübersichtliche if-else-Strukturen.

Hinweis: Wenn finally verwendet wird, muss zuvor mindestens eine catch-Anweisung stehen.

Beachte: Der finally-Block wird auf jeden Fall durchgeführt, auch wenn der try-Block ohne eine Ausnahme durchlaufen wurde. Hier kann man zum Beispiel ein "return" hinschreiben, wenn der try-Block selbst in einer gerufenen Methode steht, usw.

Da gibt es noch ein Problem: Die Ausnahme muss ja nicht in der jeweiligen Anweisung selbst erfolgen, sondern sie könnte in einem gerufenen Unterprogramm stattfinden. Jene Methode muss dann aber die zuständige Ausnahmebehandlung kennen. Um dies sicherzustellen, muss man die Ausnahmen, die möglicherweise von der gerufenen Methode geworfen werden müssen oder können, in ihrem Kopf (unmittelbar hinter der Parameterliste) nach dem Schlüsselwort "throws" angeben:

```
public void machen (int i) throws heute, gestern { ... }
```

bedeutet also, dass im Methodenrumpf { ... } die Ausnahmen "heute" und "gestern" vorkommen und möglicherweise geworfen werden und die Methode "machen" daher die entsprechenden Aufrufe vorsehen muss.

1.4.4 Interfaces

Abstrakte Datentypen sind in der Grundvorlesung behandelt worden. In Java kann man die Abstraktion auf zwei Arten einsetzen: in Interfaces und abstrakten Klassen.

Ein Interface ist eine Auflistung von Spezifikationen von Methoden.

```
interface <Name des Interface> {
    <Folge von Konstanten und Methodenspezifikationen>
}
```

Somit ist ein Interface wie eine "abgemagerte" Klasse aufgebaut.

Ein Interface muss auch implementiert werden. Dies geschieht durch eine zugehörige Klasse:

```
class <Name der Klasse> implements <Name des Interface> {
    <alle Methoden des Interface müssen hier auftreten>
    <alle sonstigen Möglichkeiten einer Klasse können hier
     zusätzlich genutzt werden>
}
```

Das Besondere besteht darin, dass ein Interface von mehreren verschiedenen Klassen implementiert werden kann.

Objekte vom Typ des Interface sind zugelassen. Man kann auch Felder vom Typ des Interface bilden, deren Komponenten Objekt der verschiedenen implementierenden Klassen sein dürfen usw.

1.4.5 Abstrakte Klassen und Methoden (vgl. Ada)

Eine *abstrakte Klasse* dient vorwiegend dem Entwicklungsprozess eines Programms. Von einer abstrakten Klasse dürfen keine Objekte erzeugt werden, vielmehr stellt sie einen Baustein in der Hierarchie der Vererbung dar.

Eine *abstrakte Methode* wird nur spezifiziert; sie muss aber später auf irgendeiner Ebene der Vererbungshierarchie konkret ausprogrammiert werden.

Man sagt auch, eine abstrakte Klasse sei eine Klasse mit mindestens einer abstrakten Methode.

Die Ideen sind aus der Grundvorlesung bekannt; wir gehen hier nicht näher darauf ein.

1.4.6 Threads (leichtgewichtige Prozesse)

Viele Programmteile lassen sich eine gewisse Zeit lang unabhängig von einander ausführen. Man verteilt sie dann auf verschiedene Prozessoren und startet sie dort. Hin und wieder müssen Synchronisationen durchgeführt werden, weil zum Beispiel Daten von einem Prozess an einen anderen zu übergeben sind; siehe Kapitel 1 dieser Vorlesung.

Solche Programmteile, die nebenläufig zueinander ablaufen können, nennt man "Thread" (engl.: Faden). Der Ablauf des Gesamtprogramms wird als eine Linie aufgefasst, die sich an gewissen Stellen in diverse Fäden aufspaltet, die später wieder zusammenlaufen (können), bzw.: Vom Hauptprozess zweigen diverse Seitenprozesse ab, die miteinander direkt oder über gemeinsame Variable kommunizieren können.

Wir demonstrieren dies an dem in der Grundvorlesung mehrfach benutzten Erzeuger-Verbraucher-System, siehe dort Kapitel 13.

Zur Erläuterung: Es gibt eine vordefinierte Klasse "Thread", die unter anderem die (statischen) Methoden "start" (ein thread wird gestartet), "run" (ein thread läuft ab), "sleep" (ein thread wartet), "yield" (Kontrolle an andere threads abgeben) besitzt. Die Methode "start" stößt auch "run" an. Um bei der Ausführung einer Methode oder einzelner Anweisungen nicht gestört zu werden, verwendet man das Schlüsselwort synchronized. Zum Beispiel bedeutet im Anweisungsteil

synchronized (<Objekt A>) {<Anweisungsfolge>}

dass das Objekt A, während die <Anweisungsfolge> abläuft, für jeden Zugriff durch andere Objekte gesperrt ist. Im Falle

synchronized <Methode>

(vor der Definition der <Methode>) darf die <Methode> später zu jedem Zeitpunkt von höchstens einem Objekt ausgeführt werden.

***Beispiel:* Erzeuger-Verbraucher-System.** Zuerst das Lager, wobei wir nur Lagerinhalt[1] bis Lagerinhalt[max] und LagerInhalt[0] für den Fehlerfall "Leer" verwenden:

```
class Lager {
    int Anzahl, Lagerkapazitaet;
    Lager(int max) {Anzahl = 0; Lagerkapazitaet = max;}
    long [] LagerInhalt = new long [Lagerkapazitaet+1];
    boolean istLeer( ) {return Anzahl == 0;}
    boolean istVoll( ) {return Anzahl >= Lagerkapazitaet;}
    synchronized void speichern (long a) {
        if (! istVoll()) {Anzahl ++; LagerInhalt[Anzahl] = a;}
    }
    synchronized long entnehmen ( ) {
        long x = LagerInhalt[Anzahl];
        if (! istLeer()) Anzahl --;
        return x;
    }
}
```

Vorsicht Fehler!

Hinweis: Obige Klasse wird bei

new Lager(50)

anders initialisiert, als man denken könnte. Bei "new" werden zuerst alle Deklarationen ausgeführt und erst anschließend (!) wird der Konstruktor ausgeführt. Man erhält also gar kein Feld mit dem Indexbereich 0..49.
Eclipse erkennt dies nicht, obwohl in Java keine nicht initialisierte Variable verwendet werden soll.
Man muss also das Feld LagerInhalt im Konstruktor erzeugen und dort die Länge "max" festlegen.

Dies ist im Folgenden ausgeführt worden. Wir wechseln zugleich zu englischen Begriffen und benennen das Lager nun mit "Store".

Korrekte Fassung

```
class Store {
    int Anzahl, Lagerkapazitaet;
    long [] LagerInhalt;
    Store(int max) {Anzahl = 0; Lagerkapazitaet = max;
        LagerInhalt = new long [Lagerkapazitaet+1];
        LagerInhalt[0] = -500L;  // "0" ist ein Fehlerfall; er sollte besser
    }                                // durch eine Ausnahme abgefangen werden.
    boolean istLeer( ) {return Anzahl == 0;}
    boolean istVoll( ) {return Anzahl >= Lagerkapazitaet;}
    synchronized void speichern (long a) {
        if (! istVoll()) {Anzahl ++; LagerInhalt[Anzahl] = a;}
    }
    synchronized long entnehmen ( ) {
        long x = LagerInhalt[Anzahl];
        if (! istLeer()) Anzahl --;  // Dieser Fall darf eigentlich
        return x;                        // nicht eintreten!
    }
}
```

Erzeuger (dieser arbeitet unendlich lange weiter; immer wenn das als Parameter an "meinLager" übergebene Lager nicht voll ist, kann er ein Element vom Typ long dort einspeichern):

```
class Producer extends Thread {
    Lager  meinLager;        //Referenz auf ein Objekt vom Typ Lager
    Producer (Lager x ) {meinLager = x;}
    public void run ( ) {
        for ( ; true ; )
            if (! meinLager.istVoll( )) erzeuge_und_speichere( );
            else yield( );
    }
    public void erzeuge_und_speichere( ) {
        long erzeugt = <hier problemspezifisch etwas einsetzen>;
        do {continue;} while (meinLager.istVoll());
        meinLager.speichern(erzeugt);
    }
}
```

Verbraucher (dieser arbeitet unendlich lange; immer wenn das als Parameter an "meinLager" übergebene Lager nicht leer ist, kann er ein Element vom Typ long dort entnehmen):

```
class Consumer extends Thread {
    Lager  meinLager;
    Consumer (Lager x) {meinLager = x;}
    public void run () {
        for ( ; true ; )
            if (! meinLager.istLeer( )) konsumieren( );
            else yield( );
    }
    public void konsumieren( ) {  // Warten, bis Lager nicht leer ist
        do {continue;} while (meinLager.istLeer());
        long product = meinLager.entnehmen();
        <hier problemspezifisch "product" verarbeiten>
    }
}
```

Nun fehlt noch eine Testumgebung für das Erz.-Verbr.-System:

```
class TestEVS {
    static int anzahlProdukte;
    static Store schuppen;
    TestEVS() {anzahlProdukte = 10;}
    public static void main (String [] args) {
        schuppen = new Store(10);
        Producer firma1 = new Producer(schuppen);
        Producer firma2 = new Producer(schuppen);
        Consumer mensch1 = new Consumer(schuppen);
        Consumer mensch2 = new Consumer(schuppen);
        firma1.start();  firma2.start();
        mensch1.start();  mensch2.start();
        < Füge noch Anweisungen für Ausgabe und Terminieren hinzu>
    }
}
```

Dieses System endet nicht. Man muss daher in die Klassen und in TestEVS noch Terminierungsmöglichkeiten einbauen, z. B.:

```
/* Programm für Threads: Erzeuger-Verbraucher-System */
import java.util.*;       // notwendig für das Objekt Random

class TestEVS2 {
   static TestEVS2 test;
   static int anzahlProdukte;
   Store schuppen;
   TestEVS2 () {anzahlProdukte = 12;}
   public static void main (String [] args) {
      test = new TestEVS2();
      test.ausfuehren();
   }
   void ausfuehren() {
      schuppen = new Store(5);
      Producer firma1 = new Producer(schuppen);
      Producer firma2 = new Producer(schuppen);
      Consumer mensch1 = new Consumer(schuppen);
      Consumer mensch2 = new Consumer(schuppen);
      firma1.start();  firma2.start();
      mensch1.start();  mensch2.start();
   }
}
class Consumer extends Thread {
   Store  meinLager;
   Consumer (Store x) {meinLager = x;}
   public void run () {
      for (int i = 0; i < TestEVS2.anzahlProdukte; i++)
         if (! meinLager.istLeer()) konsumieren();
         else yield();
   }
   public void konsumieren() {
      if (! meinLager.istLeer()) {
         long product = meinLager.entnehmen();
         System.out.println(this + " konsumierte " + product);
      }
      else  yield();
   }
}
```

```
class Producer extends Thread {
   Store  meinLager;     //Referenz auf ein Objekt vom Typ Lager
   Random r;
   Producer (Store x ) {meinLager = x;
      r = new Random(System.currentTimeMillis());
      }
   public void run () {
      for (int i = 0; i < TestEVS2.anzahlProdukte; i++)
         if (! meinLager.istVoll()) erzeuge_und_speichere();
         else yield();
   }
   public void erzeuge_und_speichere() {
      long erzeugt = Math.abs(r.nextLong()) % 1000000;
      if (! meinLager.istVoll())  {
         meinLager.speichern(erzeugt);
         System.out.println(this + " erzeugte die Zahl " + erzeugt);
      }
      else yield();
   }
}
class Store {
   int Anzahl, Lagerkapazitaet;
   long [] LagerInhalt;
   Store(int max) {Anzahl = 0; Lagerkapazitaet = max;
      LagerInhalt = new long [Lagerkapazitaet+1];
      LagerInhalt[0] = -500L;
   }
   boolean istLeer( ) {return Anzahl == 0;}
   boolean istVoll( ) {return Anzahl >= Lagerkapazitaet;}
   synchronized void speichern (long a) {
      if (! istVoll()) {Anzahl ++; LagerInhalt[Anzahl] = a;}
   }
   synchronized long entnehmen ( ) {
      long x = LagerInhalt[Anzahl];
      if (! istLeer()) Anzahl --;
      return x;
   }
}
```

Das Feld LagerInhalt hat Länge 5. Eigentlich sollen genau 12 Zahlen ("anzahlProdukte") erzeugt und konsumiert werden, das passiert aber nicht (warum??). Eine mögliche Ausgabe lautet:

```
Thread[Thread-0,5,main] erzeugte die Zahl 178639
Thread[Thread-0,5,main] erzeugte die Zahl 862324
Thread[Thread-0,5,main] erzeugte die Zahl 261709
Thread[Thread-0,5,main] erzeugte die Zahl 976684
Thread[Thread-1,5,main] erzeugte die Zahl 178639
Thread[Thread-1,5,main] erzeugte die Zahl 862324
Thread[Thread-2,5,main] konsumierte 178639
Thread[Thread-3,5,main] konsumierte 862324
Thread[Thread-0,5,main] erzeugte die Zahl 550312
Thread[Thread-2,5,main] konsumierte 976684
Thread[Thread-2,5,main] konsumierte 261709
Thread[Thread-2,5,main] konsumierte 862324
Thread[Thread-2,5,main] konsumierte 178639
Thread[Thread-3,5,main] konsumierte 550312
Thread[Thread-0,5,main] erzeugte die Zahl 121662
Thread[Thread-0,5,main] erzeugte die Zahl 1447
Thread[Thread-0,5,main] erzeugte die Zahl 299690
Thread[Thread-0,5,main] erzeugte die Zahl 94397
Thread[Thread-2,5,main] konsumierte 261709
Thread[Thread-2,5,main] konsumierte 94397
Thread[Thread-2,5,main] konsumierte 299690
Thread[Thread-2,5,main] konsumierte 1447
Thread[Thread-2,5,main] konsumierte 121662
Thread[Thread-1,5,main] erzeugte die Zahl 261709
Thread[Thread-1,5,main] erzeugte die Zahl 976684
Thread[Thread-1,5,main] erzeugte die Zahl 68650
Thread[Thread-1,5,main] erzeugte die Zahl 550312
Thread[Thread-1,5,main] erzeugte die Zahl 121662
Thread[Thread-1,5,main] erzeugte die Zahl 1447
```

Modifizieren Sie dieses Programm, sodass tatsächlich genau "anzahlProdukte" viele Zahlen erzeugt und auch korrekt konsumiert werden!

Auf die Abarbeitungsreihenfolge der Prozesse hat man keinen Einfluss. Es ist aber ein gewisses Schema zu erkennen. Untersuchen Sie dieses genau.

Bei Testläufen trat auch der Fall "-500" auf, d. h., auch der Fall Anfang = 0 ist möglich. Überlegen Sie, warum dies geschehen kann.

Frage:
Ist es Zufall, dass mehrere Zahlen zweimal erzeugt wurden? Gehen Sie das Programm genau durch und versuchen Sie, diese Frage zu klären.

1.5 Erkennung von Teilwörtern (substring-problem)

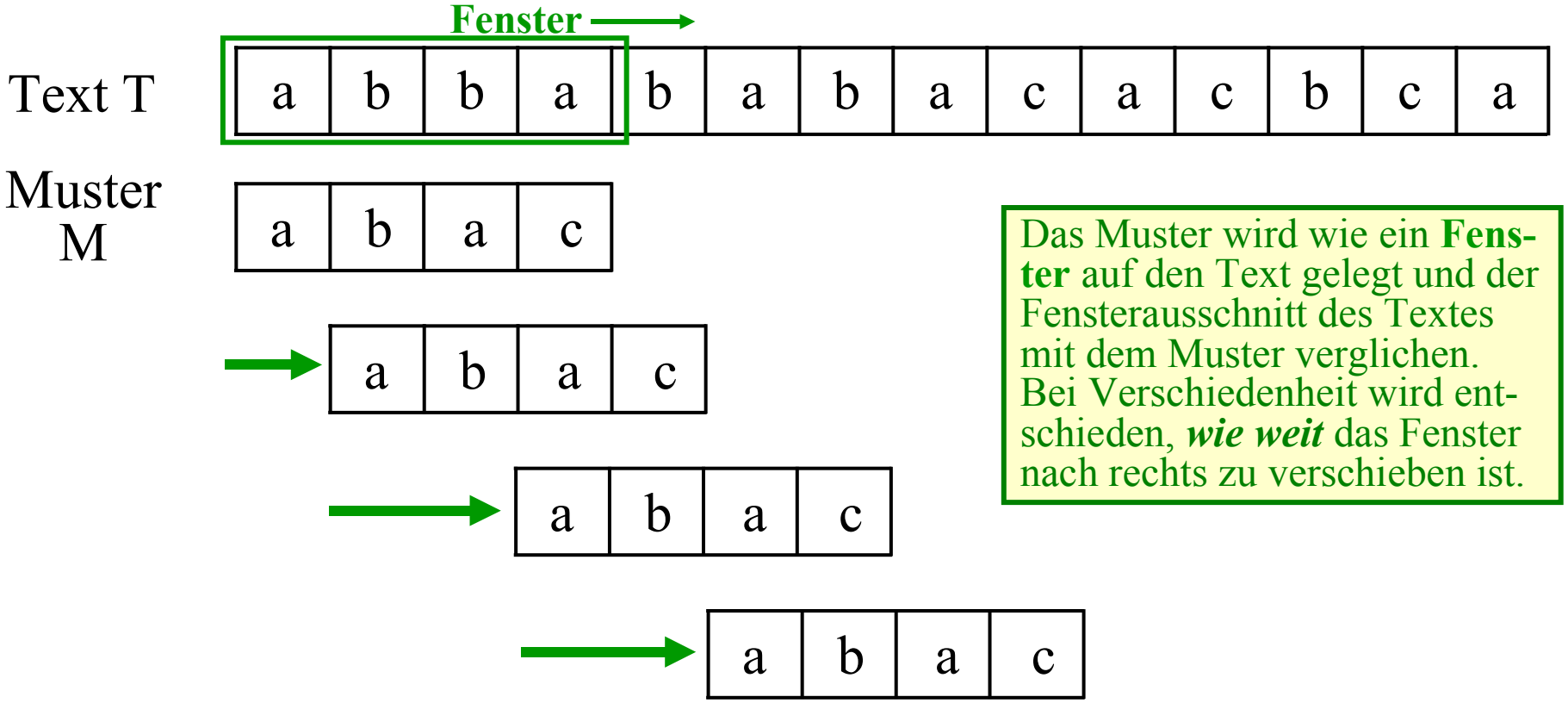
1.5.1 Aufgabenstellung:

Gegeben sind: String T (= der Text)
String M (= das Muster).
Stelle fest, ob M als Teiltext (substring) in T vorkommt.

Definitionen: Es sei $\Sigma = \{a_1, a_2, ..., a_r\}$ ein Alphabet mit r Zeichen. Σ^* ist die Menge der Wörter über Σ mit der Operation "Konkatenation" (= Hintereinanderschreiben) und dem leeren Wort ε. Jedes Wort $w \in \Sigma^*$ lässt sich als $w = w_1 w_2 ... w_k$ schreiben mit $w_i \in \Sigma$; $k \geq 0$ heißt Länge von w.

u heißt Präfix oder Anfangswort von w, wenn es ein Wort v mit $w = uv$ gibt. u heißt Suffix oder Endwort von w, wenn es ein Wort v mit $w = vu$ gibt. u heißt Teilwort ("substring") von w, wenn es Wörter v_1 und v_2 mit $w = v_1 u v_2$ gibt. Falls zusätzlich $u \neq w$ gilt, so heißt u echtes Präfix, Suffix bzw. Teilwort.

Die Grundidee bei der Teilworterkennung besteht darin, ein Muster M links auf den Text T zu legen, gewisse Zeichen miteinander zu vergleichen und, sobald zwei verschiedene Zeichen vorliegen, das Muster nach rechts zu verschieben und erneut Vergleiche durchzuführen.



Das Muster wird wie ein **Fenster** auf den Text gelegt und der Fensterausschnitt des Textes mit dem Muster verglichen. Bei Verschiedenheit wird entschieden, ***wie weit*** das Fenster nach rechts zu verschieben ist.

1.5.2 Eine nahe liegende Lösung

Um festzustellen, ob ein Wort $M = M_0...M_{m-1}$ Teilwort von $T = T_0...T_{n-1}$ ist, vergleiche die Wörter buchstabenweise:

```
for (int i = 0; i <= n-m; i++) {
    j = 0;
    while ((j < m) && (T_{i+j} = M_j))  j++;
    if (j == m) return true;
}
return false;
```

In der Praxis wird man die Position i als Ergebnis zurückgeben, ab der M ein Teilwort von T ist, bzw. "-1", falls M kein Teilwort von T ist. Dies führt sofort zu folgendem Java-Programm:

```
class Teilwort {
   static String Text = "text.....";   static String Muster = "....";
   public static int substring (String T, String M) {
      int LT = T.length(); int LM = M.length(); int j;
      if (LM <= LT) {
         for (int i = 0; i <= LT-LM; i++) {
            j = 0;
            while ((j<LM) && (T.charAt(i+j)==M.charAt(j)))  j++;
            if (j == LM) return i;              // gefunden ab Position i
      }  }
      return -1;                                 // nicht als Teilwort enthalten
   }
   public static void main (String [] args) {
      String T = Text; String M = Muster; int Position = substring(T,M);
      System.out.print("Das Muster ist im Text ");
      if (Position < 0) System.out.print("nicht");
      else System.out.print("ab Position " + (Position+1));
      System.out.println(" enthalten.");
   }
}
```

Ausgabe dieses Programms: Das Muster ist im Text ab Position 5 enthalten.

In diesem Programm wird nur einmal nach dem Teilwort gesucht. Man kann die Methode substring aber leicht so abwandeln, dass *jedes* Auftreten des Teilwortes Muster notiert und (als String oder als Feld) zurückgegeben wird. Man kann im Falle i = 0 bzw. i = n - m = LT-LM zusätzlich ausgeben, dass das Muster M Präfix bzw. Suffix des Textes T ist.

Aufwand dieses Verfahrens: **O(n·m)** für n=|T| und m=|M|.

Der schlechteste Fall tritt auch ein, zum Beispiel für die Wörter
T = aaaa...aa = a^n und M = aa...ab = $a^{m-1}b$.

Kann man dieses Verfahren beschleunigen?

Ja, auf mehrfache Arten. Drei werden im Folgenden vorgestellt.

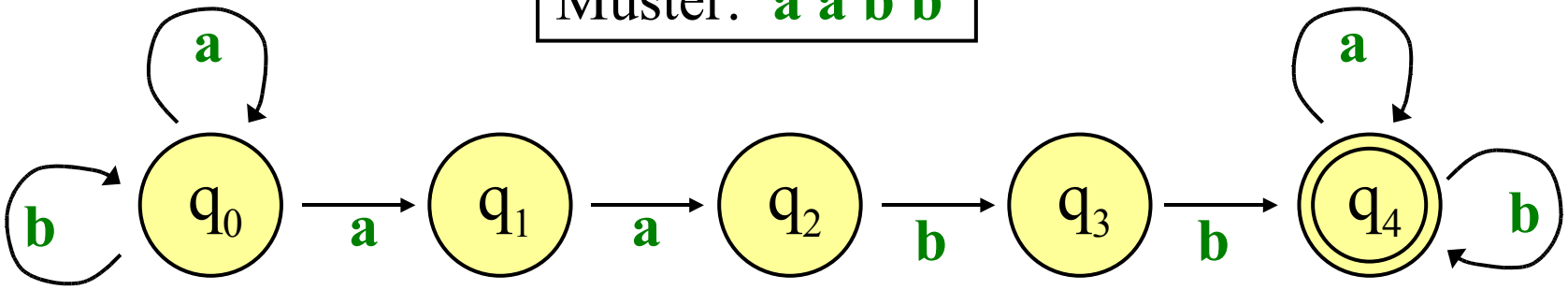
1.5.3 Vorgehen der Theoretischen Informatik

Idee: Baue zu dem Muster M einen endlichen Akzeptor A mit der Eigenschaft: A gelangt bei Eingabe von T genau dann in einen Endzustand, wenn M Teilwort von T ist.

Der Akzeptor darf nichtdeterministisch sein; man kann ihn ja anschließend in einen deterministischen umwandeln.

$A = (Q, \Sigma, \delta, q_0, F)$ besteht aus der Zustandsmenge Q, dem Eingabalphabet Σ, der Überführungsrelation $\delta \subseteq Q \times \Sigma \times Q$, dem Anfangszustand q_0 und der Endzustandsmenge $F \subseteq Q$. Für praktische Zwecke muss δ eine Abbildung von $Q \times \Sigma$ nach Q sind, d. h., A muss dann deterministisch sein. Die Umwandlung liefert im Falle, dass A zu einem Muster M gehört, stets einen deterministischen Akzeptor mit der Zustandszahl $|Q| = m+1$, wobei m die Länge des Musters M ist.

Wir erläutern dies an einem Beispiel.



Nichtdeterministischer endlicher Akzeptor A mit der Zustandsmenge $Q = \{q_0, q_1, q_2, q_3, q_4\}$, dem Eingabealphabet $\Sigma = \{a, b\}$, dem Anfangszustand q_0 und der Endzustandsmenge $F = \{q_4\}$.

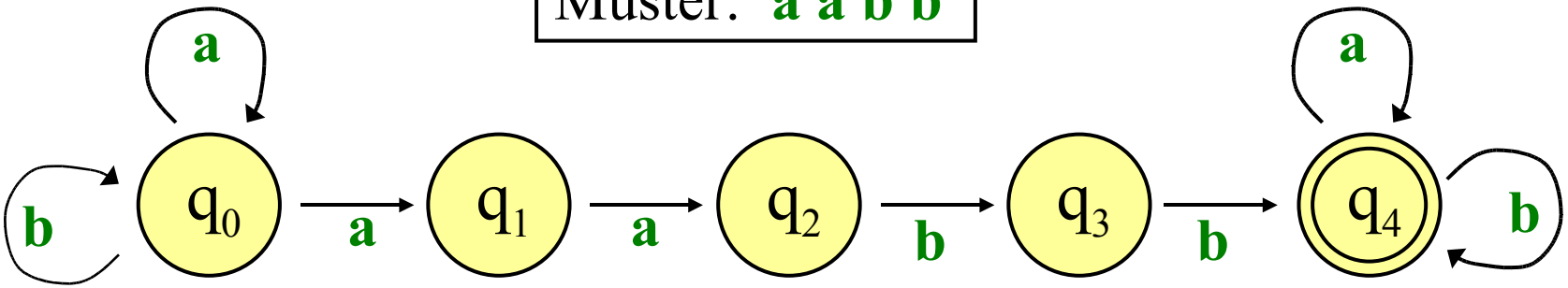
Das Umwandeln von A in einen deterministischen endlichen Akzeptor À' erfolgt mit Hilfe der "Potenzmengenkonstruktion":

Die neue Zustandsmenge wird die Potenzmenge 2^Q von Q. Bilde für jedes $Z \subseteq Q$ und $x \in \Sigma$ die Überführungsfunktion

$\delta'(Z,x) = \{q' \mid \exists\, q \in Z$, sodass es einen Übergang $q \xrightarrow{x} q'$ in A gibt$\}$.

Es genügt natürlich, alle die $Z \subseteq Q$ zu betrachten, die von dem neuen Startzustand $\{q_0\}$ aus erreichbar sind. Endzustände sind alle Z mit $Z \cap F \neq \emptyset$. Illustration an obigem Beispiel:

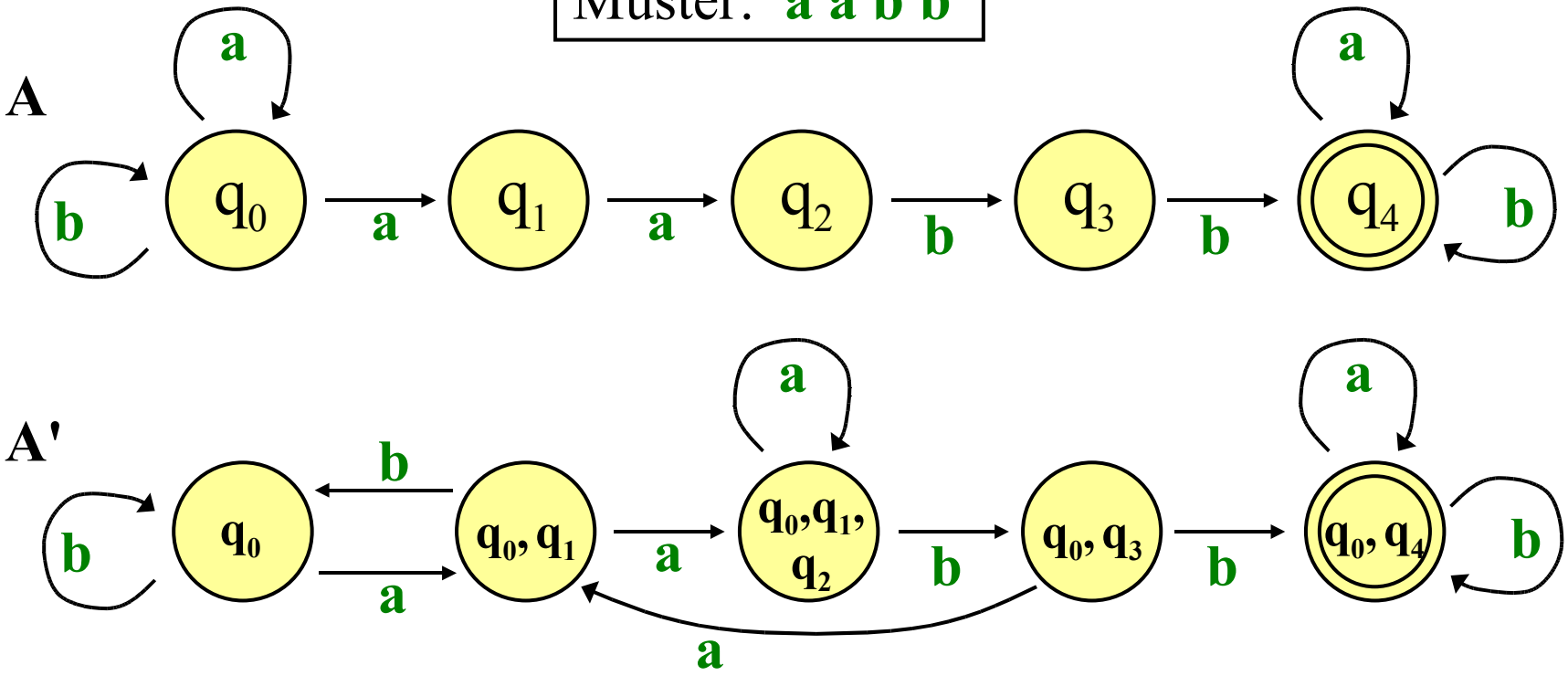
Muster: **a a b b**



	a	b
$\{q_0\}$	$\{q_0,q_1\}$	$\{q_0\}$
$\{q_0,q_1\}$	$\{q_0,q_1,q_2\}$	$\{q_0\}$
$\{q_0,q_1,q_2\}$	$\{q_0,q_1,q_2\}$	$\{q_0,q_3\}$
$\{q_0,q_3\}$	$\{q_0,q_1\}$	$\{q_0,q_4\}$
$\{q_0,q_4\}$	$\{q_0,q_1,q_4\}$	$\{q_0,q_4\}$
$\{q_0,q_1,q_4\}$	$\{q_0,q_1,q_2,q_4\}$	$\{q_0,q_4\}$
$\{q_0,q_1,q_2,q_4\}$	$\{q_0,q_1,q_2,q_4\}$	$\{q_0,q_3,q_4\}$
$\{q_0,q_3,q_4\}$	$\{q_0,q_1,q_4\}$	$\{q_0,q_4\}$

Hinweis:
Hier können die vier Endzustände $\{q_0,q_4\}$, $\{q_0,q_1,q_4\}$, $\{q_0,q_3,q_4\}$ und $\{q_0,q_1,q_2,q_4\}$ zu einem einzigen Endzustand $\{q_0,q_4\}$ zusammengefasst werden (sog. "äquivalente" Zustände, siehe Vorlesung Theoretische Informatik 1).
Dann erhält man:

Muster: **a a b b**

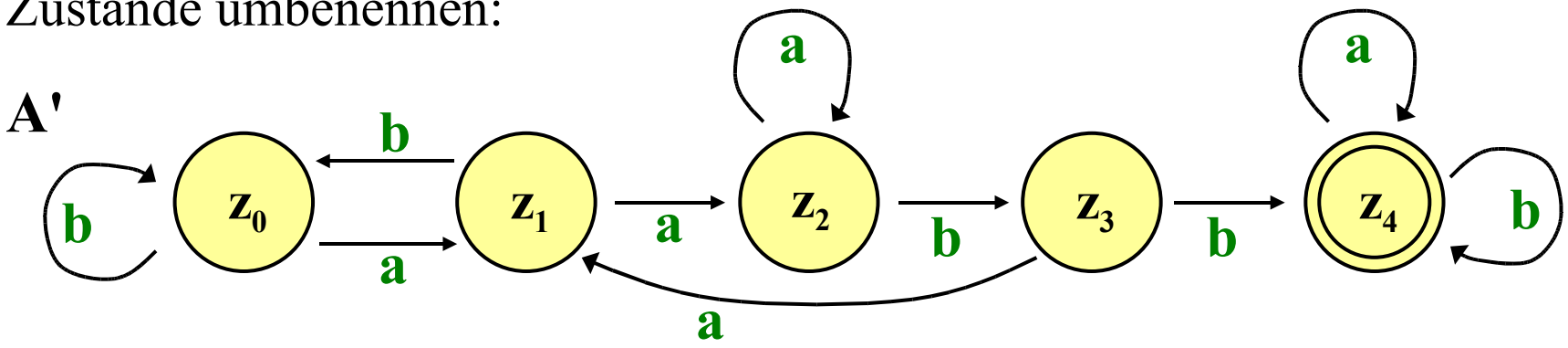


Aus dem nichtdet. Akzeptor A wird somit der deterministische Akzeptor $A' = (Q', \{a, b\}, \delta', \{q_0\}, \{\{q_0,q_4\}\})$, der bis auf δ' dem Akzeptor A strukturell ähnelt. Die Zustandsmenge Q' lautet: $\{\{q_0\}, \{q_0,q_1\}, \{q_0,q_1,q_2\}, \{q_0,q_3\}, \{q_0,q_4\}\}$.

δ'	a	b
$\{q_0\}$	$\{q_0,q_1\}$	$\{q_0\}$
$\{q_0,q_1\}$	$\{q_0,q_1,q_2\}$	$\{q_0\}$
$\{q_0,q_1,q_2\}$	$\{q_0,q_1,q_2\}$	$\{q_0,q_3\}$
$\{q_0,q_3\}$	$\{q_0,q_1\}$	$\{q_0,q_4\}$
$\{q_0,q_4\}$	$\{q_0,q_4\}$	$\{q_0,q_4\}$

Muster: **a a b b**

Zustände umbenennen:



Vorgehen für ein Muster M der Länge m:

(1) Zustandsmenge $\{z_0, z_1, ..., z_m\}$ mit Anfangszustand z_0 und Endzustand z_m.

(2) Konstruiere die Überführungstabelle δ' aus M wie oben skizziert (d. h., A' ist nun berechnet).

(3) Gib den Text T buchstabenweise in den so entstandenen Akzeptor A' ein. Wird der Endzustand z_m erreicht, so melde dies (d.h., genau bis hierhin ist das Muster M erstmalig im Text T enthalten).

δ'	a	b
z_0	z_1	z_0
z_1	z_2	z_0
z_2	z_2	z_3
z_3	z_1	z_4
z_4	z_4	z_4

Aufwand dieses Verfahrens: **$O(n + k \cdot m^2)$** Schritte.

Gegeben: Eingabealphabet Σ mit $|\Sigma| = k$ und Muster M der Länge m.

(1) Zustandsmenge $\{z_0, z_1, ..., z_m\}$ mit Anfangszustand z_0 und Endzustand z_m sowie Σ.

(2) Konstruiere die Überführungstabelle δ' aus M wie skizziert (d. h., A' ist nun berechnet).

(3) Gib den Text T der Länge n buchstabenweise in den so entstandenen Akzeptor A' ein. Wird der Endzustand erreicht, so melde dies (d.h., genau hier ist das Muster M erstmalig im Text T enthalten).

m+1 + 2 + k Schritte

$k \cdot m$ Tabelleneinträge δ' erstellen; jeder Tabelleneintrag kostet (nur im Spezialfall der Wörter!) bis zu 2m Schritte, um die Teilmenge zu bilden, sowie m Schritte, um den soeben konstruierten Zustand zu erkennen. Gesamt: $\leq 3 \cdot k \cdot m^2$ Schritte

$\leq$ n+1 Schritte

Kann man dieses Verfahren beschleunigen? (Ja.)

Wenn man es beschleunigen will, aber das Lesen von links nach rechts des Akzeptors beibehalten möchte, so kann dies nur in Punkt (2) geschehen. Man muss versuchen, den Umweg über die Potenzmenge zu vermeiden und stattdessen die Überführungsfunktion δ' direkt konstruieren. Hierbei kann man die Zustandsmenge auch durch die "Zustände" z_u ersetzen, wobei u alle Präfixe des Musters M durchläuft. (Betrachten Sie die Zustände im obigen Beispiel: Der Zustand z_i entspricht dort genau dem gelesenen Präfix von M der Länge i.)

Diese Überlegung führt zu einem Verfahren, das Knuth, Morris und Pratt vor 30 Jahren vorgestellt haben.

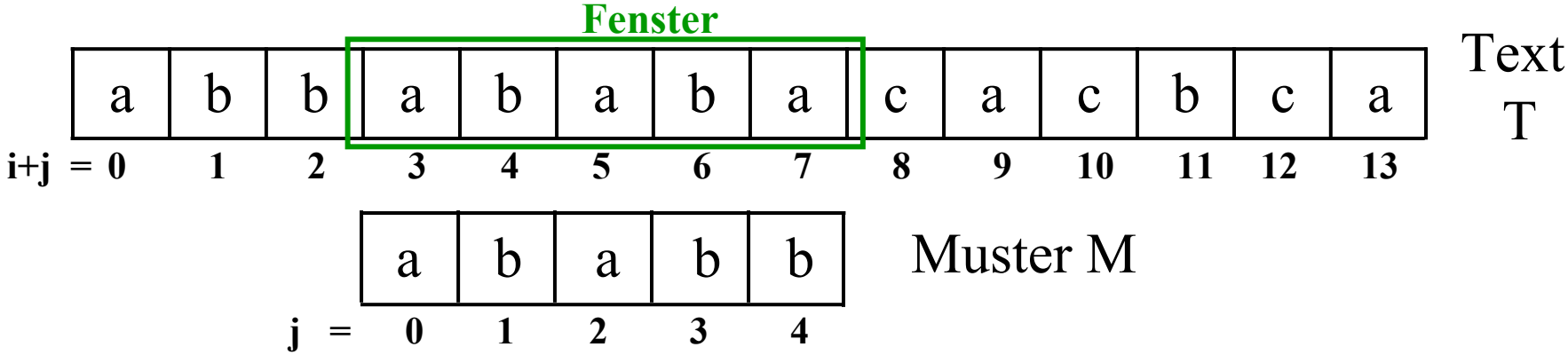
1.5.4 Der KMP-Algorithmus

KMP steht für die Autoren "Knuth, Morris und Pratt" des Artikels "Fast pattern matching in strings", SIAM Journal on Computing, 6, no. 1, Seiten 323-350, 1977.

Wir betrachten wieder die "nahe liegenden Lösung". Dort wird das Muster zeichenweise mit dem Fenster des Textes verglichen: $T_{i+j} = M_j$ für $j = 0, 1, 2, ...$

Falls dies ungleich ist, wird das Muster nur um eine (!) Position verschoben (d. h.: $i = i+1$) und die Überprüfung erfolgt erneut. Man könnte aber oft um mehrere Positionen verschieben; denn die Information, dass man bei der Ungleichheit $T_{i+j} \neq M_j$ bereits weiß, dass $T_{i+k} = M_k$ für $k = 0, 1, ..., j-1$ gilt, wird bei dem nahe liegenden Verfahren weggeworfen!

Betrachte als Beispiel eine Situation T[i+j] ≠ M[j] :



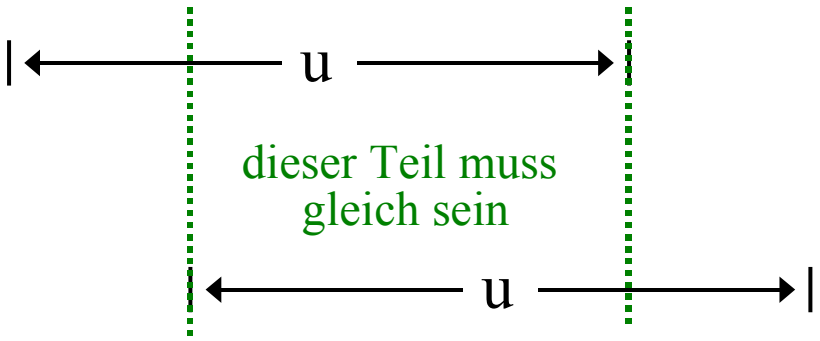
i=3, j=0: T[i+j] = T[3] = M[j] = M[0] = a
i=3, j=1: T[i+j] = T[4] = M[j] = M[1] = b
i=3, j=2: T[i+j] = T[5] = M[j] = M[2] = a
i=3, j=3, T[i+j] = T[6] = M[j] = M[3] = b
i=3, j=4: T[i+j] = T[7] ≠ M[j] = M[4] = b

Genau hier weiß man, dass im Fenster gilt (i=3, j=4):
T[3..6] = T[i..i+j-1] = M[0..j-1] = M[0..3]. Man kann sofort das Muster um 2 Positionen nach rechts verschieben, weil
T[5..6] = T[i+2..i+2+k] = M[0..k] = M[0..1] für k= 0, ..., j-1-2 gilt und daher nicht noch einmal untersucht werden muss!

Allgemein gilt:
Es seien T[i..i+m-1] das Fenster des Textes T und M[0..m-1] das Muster. Für die ersten k Zeichen des Musters ($k \geq 0$) mögen Fenster und Muster übereinstimmen:

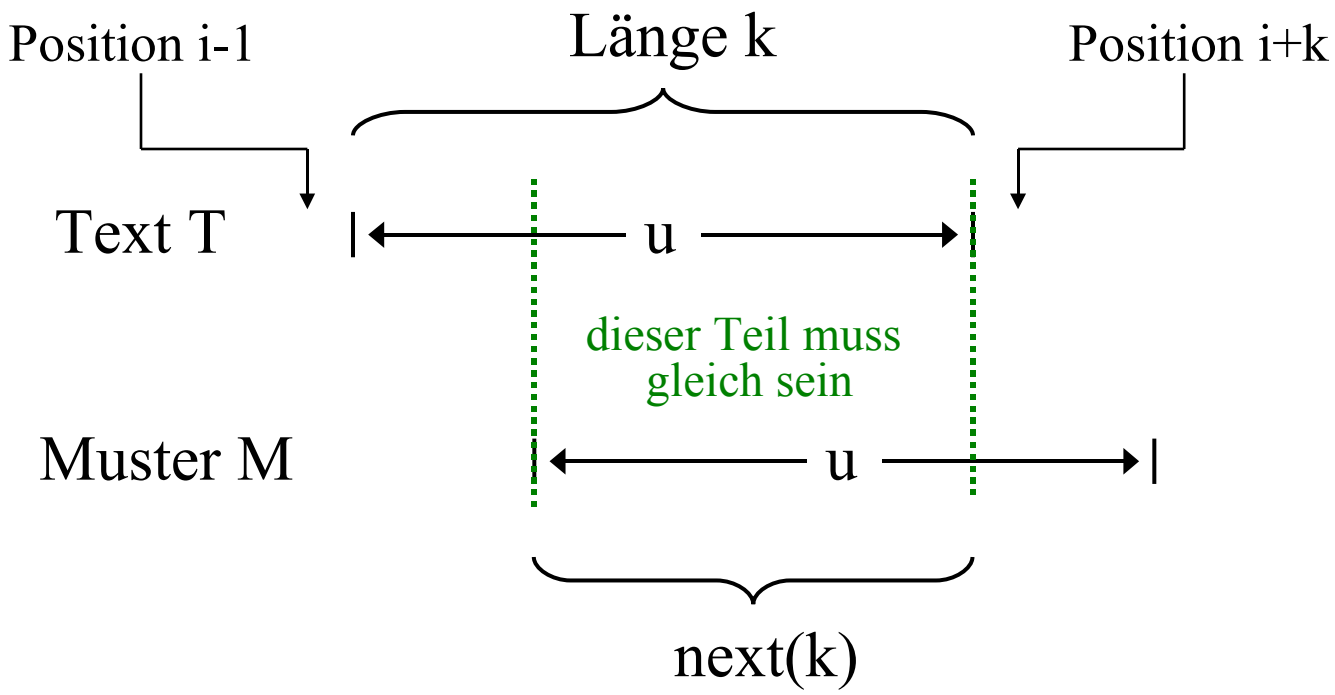
$T[i..i+m-1] = u\ a\ v$ mit $|u| = k$, $a \in \Sigma$ und $|v| = m-k-1$
$M[0..m-1] = u\ b\ v'$ mit $b \in \Sigma$, $a \neq b$ und $|v'| = m-k-1$.

u wurde gelesen, und daher kann man das Muster nun so weit verschieben, bis erstmals wieder eine Übereinstimmung von u des Musters mit u des Textes (ohne das erste Zeichen von u) entsteht. Dies ist nur vom Anfangswort u des Musters M abhängig und kann vorab berechnet werden.



Gesucht wird also das längste echte Suffix von u, das gleich einem Präfix von u ist
(das Suffix ε ist möglich).

Sei u also das Präfix von M der Länge k ≥ 0, für das T und M übereinstimmen, dann ist



Dieses Präfix der Länge next(k) ist das neue übereinstimmende Präfix für T und M, wobei ab der Textposition i+k weiter getestet wird. k entspricht j im Algorithmus. Die Indizierung erfolgt in Java aber ab 0. Daher j ist also durch next(j-1) zu ersetzen und i ist entsprechend um j - next(j-1) zu erhöhen.

Wir betrachten ein Beispiel (wir setzen formal next(0) = 0 für unseren Algorithmus; beachten Sie, dass next(1) stets 0 ist):

M = a b a b c a b a a
u ist das Anfangswort von M der Länge k:

u	Länge k	next(k)
ε	0	0
a	1	0
ab	2	0
aba	3	1
abab	4	2
ababc	5	0
ababca	6	1
ababcab	7	2
ababcaba	8	3
ababcabaa	9	1

Der bisherige Teil des Algorithmus

```
for (int i = 0; i <= n-m; i++) {
    j = 0;
    while ((j < m) && (T_{i+j} = M_j))  j++;
    if (j == m) return i;
}
return -1;
```

muss also nun ersetzt werden durch:

```
j = 0;
for (int i = 0; i <= n-m; ) {
    while ((j < m) && (T_{i+j} = M_j))  j++;
    if (j == m) return i;
    if (j == 0) i++;
    else { i += j - next[j-1]; j = next[j-1]; }
}
return -1;
```

Aufwand:

Wir nehmen an, wir hätten schon gezeigt, dass sich die Funktion next in O(m) Schritten berechnen lässt.

In den Wertzuweisung der Blöcke wird
entweder i um mindestens 1 erhöht
(beachten Sie: $j > next(j)$ für $j > 0$ ist, siehe unten)
oder
das Muster M wird um mindestens 1 nach rechts verschoben
(dies geschieht in der while-Schleife).
Hierzu gehören jeweils höchstens 2 Abfragen von Bedingungen.

Folglich können insgesamt höchstens $3 \cdot (n+m)$ Bedingungen und Wertzuweisungen ausgeführt werden, d.h., das KMP-Verfahren ist von der Größenordnung **O(n+m)**.

Nun müssen wir noch die Funktion next berechnen.

Zu berechnen ist für jedes k = 0, 1, 2, ..., m der Wert

next(k) = Länge des längsten Präfix von M, das ein Suffix von M[0..k-1] ist (für k> 0)

mit next(0) = 0.

Die Idee, um next zu berechnen, beruht darauf, jedes Anfangswort darauf zu testen, ob es im weiteren Verlauf im Anfangswort bis zur Länge k erneut vorkommt. Dabei kann man von links nach rechts vorgehen, ohne immer wieder an den Anfang zurückkehren zu müssen, wie man aus der nachfolgenden Folgerung schließen kann.

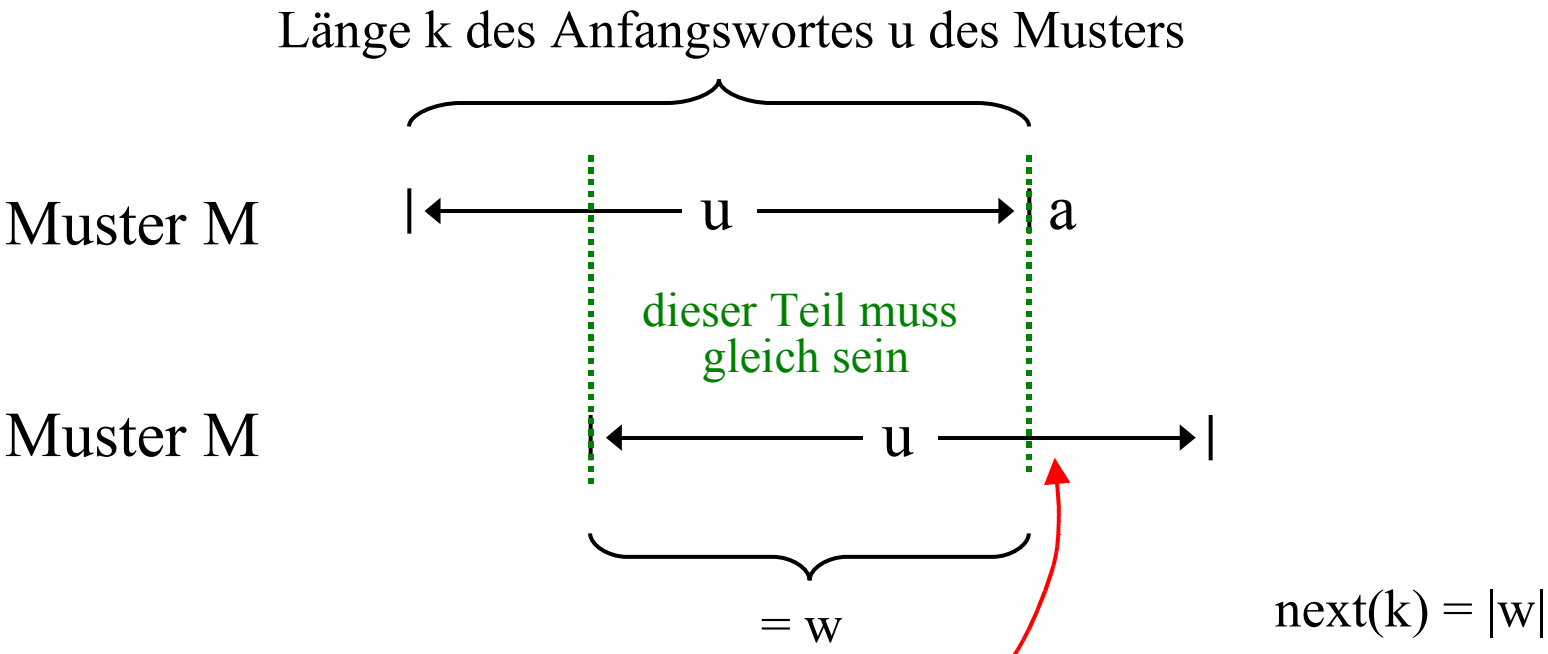
Folgerung: Für $k > 0$ gilt stets: $k \geq next(k) + 1 \geq next(k+1)$.

Beweis: Annahme: w ist längstes Präfix und zugleich Suffix von u. Also gilt für k=|u|: next(k) = |w|. Dann existieren x und y mit u = wx und u = yw. Betrachte den nächsten Buchstaben a im Muster und das Anfangswort ua der Länge k+1. Wenn v längstes Präfix und zugleich Suffix von ua ist, so muss gelten: ua = vx' = y'v. Wenn v länger als w ist, so muss v = wz mit |z| > 0 sein und es folgt: ua = wzx' = y'wz. Wäre w mindestens um 2 Zeichen länger als v, d.h. z = cdz', so wäre wc oder eine Verlängerung wc... ein Präfix von u, das zugleich Suffix von u ist, von der Länge > |w| = next(k). Dies ist ein Widerspruch zur Annahme, dass w längstes Präfix und Suffix von u ist. Daraus folgt der zweite Teil der Ungleichungen. Der erste Teil folgt hieraus und aus der Tatsache, dass next(1) = 0 ist.
(Hierbei sind $a, b, c, d \in \Sigma$ und $u, v, w, x, x', y, y', z, z' \in \Sigma^*$)

Beispiel für das Muster a b a b c a b a a. Schiebe das Muster unter sich durch und ermittle maximale Gleichheit bis zur Länge k (also nur bei den fett gedruckten Teilwörtern!):

a b a b c a b a a	next(1) = 0
a b a b c a b a a	
a b a b c a b a a	next(2) = 0
a b a b c a b a a	
a b a b c a b a a	next(3) = 1
a b a b c a b a a	
a b a b c a b a a	next(4) = 2
a b a b c a b a a	
a b a b c a b a a	next(5) = 0
a b a b c a b a a	
a b a b c a b a a	next(6) = 1
a b a b c a b a a	
a b a b c a b a a	next(7) = 2
a b a b c a b a a	
a b a b c a b a a	next(8) = 3
a b a b c a b a a	
a b a b c a b a a	next(9) = 1
a b a b c a b a a	(k= 9 wird nicht benötigt)

Skizze zur iterativen Berechnung von next(k):



Falls w = ε ist, so ist next(k) = 0. Der nächste Buchstabe nach dem Anfangswort u sei a. Steht dort auch a, so ist next(k+1) = next(k) + 1, weil wa dann längstes Präfix und Suffix von ua ist. Steht dort kein a, so verkürzt sich das Präfix/Suffix und man muss zu dem Anfangswort der Länge next(k-1) übergehen. Diese Idee ist im folgenden Java-Programm umgesetzt worden.

Das zugehörige Java-Programm lautet mit obigem Muster:

```
class TeilwortKMP {
   static String Text = "bababcababaabababcababababcbabababcabaabcaab";

   static String Muster = "ababcabaa";  // im Text ab Position 27 enthalten
   static int[] next;
   public static int substring (String T, String M) {
      int LT = T.length(); int LM = M.length(); int j;
      if (LM <= LT) {
         j = 0;
         for (int i = 0; i <= LT-LM; ) {
            while ((j<LM) && (T.charAt(i+j)==M.charAt(j)))  j++;
            if (j == LM) return i;              // gefunden ab Position i
            if (j == 0)  i++;
            else { i += j - next[j-1]; j = next[j-1]; }
      }  }
      return -1;                                // nicht als Teilwort enthalten
   }
```

```
    public static void main (String [] args) {
        String T = Text; String M = Muster;
        next = bildeNext(M);                    // Testausdruck zur Überprüfung
        for (int i=0; i<M.length(); i++) System.out.println(i+"  "+next[i]);
        int Position = substring(T,M);
        System.out.print("Das Muster ist im Text ");
        if (Position < 0) System.out.print("nicht");
        else System.out.print("ab Position " + (Position+1));
        System.out.println(" enthalten.");
    }
    private static int[] bildeNext(String M) {
        int j = 0; int [] neu = new int[M.length()];
        neu[0] = 0;
        for (int i = 1; i < M.length(); ) {
          if (M.charAt(i) == M.charAt(j)) {
              neu[i] = j+1; i++; j++;
              }
          else  if (j > 0)  j = neu[j-1];
                else {neu[i] = 0; i++;}
        }
        return neu;
    }
}
```

Dieses Programm liefert folgende Ausgabe
(vgl. Beispiel vier Folien zuvor):

```
0  0
1  0
2  1
3  2
4  0
5  1
6  2
7  3
8  1
Das Muster ist im Text ab Position 27 enthalten.
```

Hinweis 1: Natürlich kann man dieses Verfahren nochmals etwas beschleunigen; denn wir werfen ja die Information, welches Zeichen wir T[i+j] an Position i+j im Text gelesen haben, weg. Dann erhält aber die Funktion next einen weiteren Parameter $a \in \Sigma$ und benötigt nun den $|\Sigma|$-fachen Speicherplatz und auch den $|\Sigma|$-fachen Zeitaufwand zur Berechnung. Dies ist in der Praxis nur dann sinnvoll, wenn man sehr lange Texte durchsuchen muss.

Hinweis 2: In der Literatur finden Sie leichte Modifikationen des hier angegebenen Algorithmus; insbesondere ersetzt man (wie in der Methode bildeNext) i+j durch einen einzigen Zähler, der nur über die Positionen des Textes läuft.

Hinweis 3: Das Besondere am KMP-Algorithmus ist, dass der Text stets nur von links nach rechts durchlaufen wird. Man kehrt also im Text niemals an eine frühere Position links von der aktuellen Position zurück.

1.5.5 Der Boyer-Moore-Algorithmus

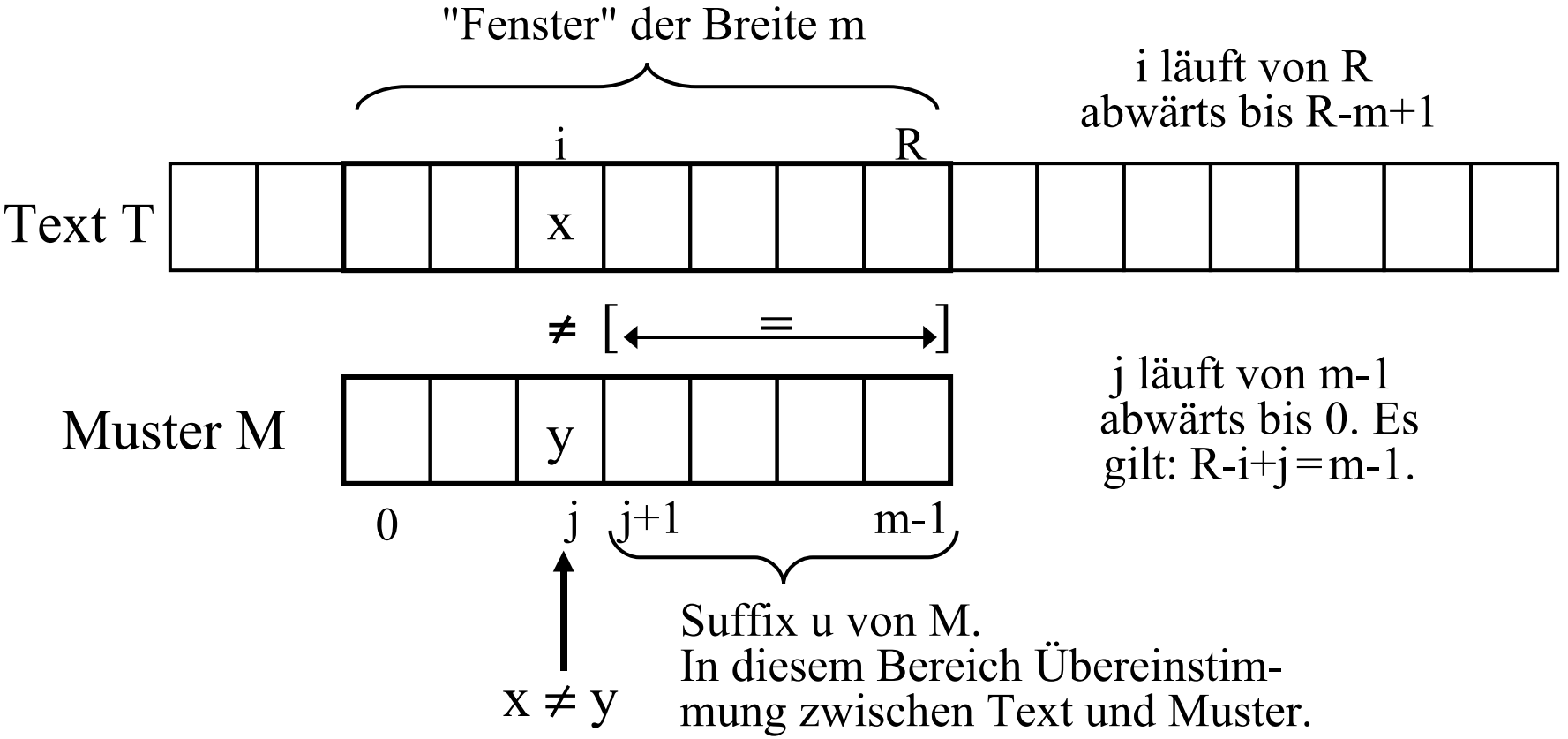
Eine andere Idee lautet: Vergleiche das Muster mit dem Text nicht von vorne nach hinten, sondern *von hinten nach vorn*.

Der Vorteil zeigt sich bei Zeichen, die im Text, aber nicht im Muster vorkommen. In $T = (a^{m-1}c)^s$ und $M = aa...a = a^m$ verschiebt man M bei jedem Vergleich "ist c gleich a?" sofort um m Positionen, sodass man insgesamt nur s = n/m Vergleiche benötigt, um festzustellen, dass M kein Teilwort von T ist.

Diese Idee liegt dem Boyer-Moore-Verfahren zugrunde, welches ebenfalls 1977 vorgestellt wurde: "A fast string searching algorithm", Communications ACM 20 (1977), no.10, Seiten 762-772. Effiziente Varianten und eine genaue Analyse finden Sie im Artikel R.Cole, "Tight bounds on the complexity of the Boyer-Moore pattern matching algorithm", SIAM Journal on Computing 23 (1994), Seiten 1075-1091.

Das Boyer-Moore-Verfahren, einfache Variante.

Vergleichen von rechts. Betrachte eine typische Situation:



Falls $x = T_i \neq M_j = y$ ist, so verschiebe das Muster eine Position nach rechts, d.h., setze R = R+1; i = R; j = m-1;
(Falls x = y ist, i und j erniedrigen usw., bis j = 0 ist.)

Der Algorithmus ist außer im then-Teil der Alternative unverändert:

```
class TeilwortBM1 {
   static String Text = "text.....";   static String Muster = "....";
   public static int substring (String T, String M) {
      int LT = T.length(); int LM = M.length(); int i, j;
      if (LM <= LT) {
         for (int R = LM-1; R < LT; R ++) {
            i = R; j = LM-1;
            while ((j >= 0) && (T.charAt(i) == M.charAt(j))) {i--; j--;}
            if (j == -1) return (i+1);
      }  }
      return -1;
   }
   public static void main (String [] args) {
      String T = Text; String M = Muster; int Position = substring(T,M);
      System.out.print("Das Muster ist im Text ");
      if (Position < 0) System.out.print("nicht");
      else System.out.print("ab Position " + (Position+1));
      System.out.println(" enthalten.");
   }
}
```

(Gleiche Ausgabe wie beim früheren Programm Teilwort.)

Da dieses Verfahren bei einer Ungleichheit R nur um 1 erhöht, arbeitet es im schlechtesten Fall (worst case) ebenfalls nur in O(n·m). Ein Beispiel hierfür sind die Wörter

$$T = aa...aa = a^n \quad \text{und} \quad M = baa...aa = ba^{m-1}.$$

Hier stimmen stets m-1 Zeichen überein; erst wenn man auf das 'b' trifft, wird das Muster um eine Position nach rechts verschoben. Aber: Es ist klar, dass man in dieser Situation das Muster um m Positionen nach rechts schieben könnte, da b in dem bereits überprüften Teil a^{m-1} nicht vorkommt.

Man erkennt, dass mit der gleichen Idee wie beim KMP-Algorithmus eine deutliche Beschleunigung erzielt werden kann. Wir betrachten dies genauer, wobei erst der zweite Ansatz zum Linearzeitverhalten führen wird.

Ansatz 1: "Bad-Character" Heuristik

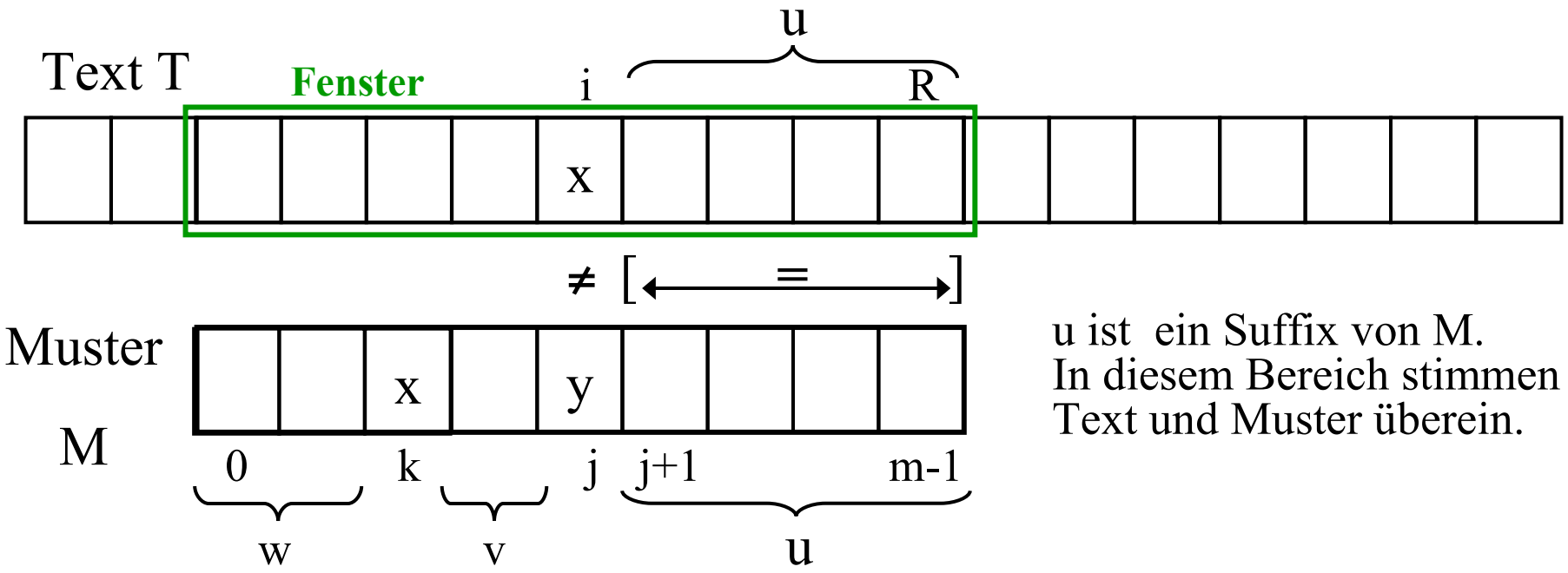
Man vergleiche von rechts nach links den Fensterausschnitt des Textes mit dem Muster M. Sind zwei verglichene Zeichen verschieden und ist x der im Text auftretende Buchstabe, so stelle fest, wo das Zeichen x im Muster vorkommt:

Kommt x nicht im Muster vor, so verschiebe das Muster nach rechts an die Position hinter x.

Kommt x weiter links im Muster vor, so verschiebe das Muster so nach rechts, dass das rechteste dieser (links stehenden) Vorkommen genau unter dasZeichen x zu liegen kommt.

Anderenfalls verschiebe das Muster um eine Position nach rechts.

Illustration: Im Suffix u stimmen T und M überein, und es sei $x \neq y$.



u ist ein Suffix von M.
In diesem Bereich stimmen Text und Muster überein.

Wenn x links von der Verschiedenheit ("mis-match") nicht mehr im Muster vorkommt, dann verschiebe man das Muster um j+1 Positionen nach rechts. Kommt x dort aber vor, so betrachte das rechteste Vorkommen von x an der Position k, d. h., M = wxvyu und x kommt in v nicht vor. Dann verschiebe man das Muster um j-k Positionen nach rechts! Anderenfalls verschiebe das Muster um eine Position.

Algorithmische Betrachtung: Tritt die Situation $x = T_i \neq M_j = y$ ein, so gehe man ab dieser Position j nach links, bis man x oder den Anfang von M erreicht hat, und erhöhe dann R. Ersetze also

```
for (int R = LM-1; R < LT; R ++) {
    i = R; j = LM-1;
    while ((j >= 0) && (T.charAt(i) == M.charAt(j))) {i--; j--;}
    if (j == -1) return (i+1);
}
```

durch

```
int R = LM-1; char x;
while ( R < LT)  {
    i = R; j = LM-1;
    while ((j >= 0) && (T.charAt(i) == M.charAt(j))) {i--; j--;}
    if (j == -1) return (i+1);
    else  { k = j; x = T.charAt(i);      // nun die Position k berechnen
        while ((k >= 0) && (x != M.charAt(k))) k--;
        R = R + (j - k);                 // R neu setzen (= Muster verschieben)
    }                                    // dies ist auch für k=-1 richtig
}
```

Algorithmische Betrachtung:
Man kann vorab leicht aus dem Muster M für jedes Zeichen x eine Tabelle "last" erstellen: Für jedes Zeichen x gibt last[x] die Position des rechtesten Vorkommens im Muster an. Man prüft dann nur, ob last[x] < j ist; falls ja, verschiebt man um (j-last[x]) Positionen, anderenfalls um 1. In Java formuliert:

```
int R = LM-1;
while ( R < LT)  {
   i = R; j = LM-1;
   while ((j >= 0) && (T.charAt(i) == M.charAt(j))) {i--; j--;}
   if (j == -1) return (i+1);
   else  {
      char x = T.charAt(i);
      if (last[x] < j)  R += j - last[x];
      else  R++;                          // R neu setzen
   }
}
```

So nützlich diese Beschleunigung in der Praxis sein mag, so wenig hilft sie aber beim Groß-O-Verhalten. Ein Beispiel hierfür sind wiederum der Text und das Muster

$$T = aa...aa = a^n \quad \text{und} \quad M = baa...aa = ba^{m-1}.$$

Die Verschiedenheit wird erst festgestellt, wenn das 'b' des Musters mit dem Text verglichen wird. Dann wird aber nur um eine Position nach rechts verschoben, da b das erste Zeichen des Musters ist.

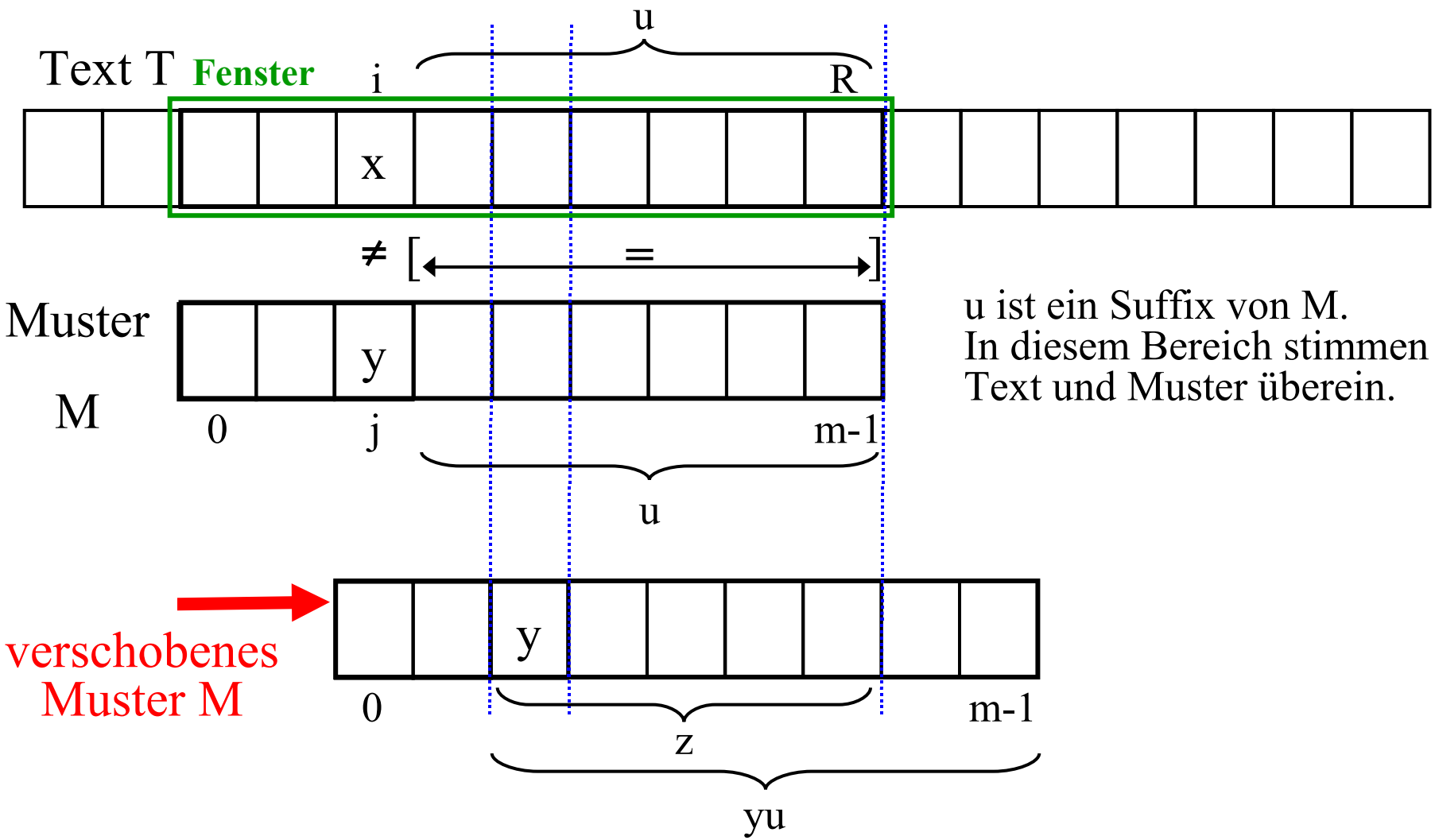
Wir müssen also noch eine andere Information ausnutzen.

Ansatz 2: "Good-Suffix" Heuristik

Stimmen zwei verglichene Zeichen nicht überein, so hat man oft aber bereits eine Übereinstimmung u zwischen einem Suffix des Musters und dem Fensterausschnitt des Textes gefunden. Gibt es ein solches nicht-leeres Suffix u und tritt u im Muster M links noch einmal auf, so verschiebe man das Muster so, dass deren rechtestes Vorkommen genau unter dem Teilwort u am Ende des Fensterausschnittes zu liegen kommt. Ist aber u kein zweites Mal im Muster vorhanden, dann kann man das Muster um (m - |u|) Positionen nach rechts verschieben, eventuell sogar noch etwas weiter.

Wie beim KMP-Verfahren kann man relativ schnell eine Tabelle V der Verschiebungen für jedes Suffix des Musters berechnen.

Illustration: Im Suffix u stimmen T und M überein, und es sei $x \neq y$.



Man erkennt als Bedingung für das **Verschieben um |yu| - |z| Positionen**:

z muss das längste Suffix von yu sein, das zugleich Anfang von yu ist.

Algorithmische Betrachtung: Die Verschiebetabelle V hängt ausschließlich vom Muster M ab. Sie lautet:

Wenn $y = M_j$ das j-te Zeichen (die Nummerierung beginnt ab 0) im Muster M ist, so beträgt die Verschiebung V[j] (falls hier die erste Verschiedenheit zum Text vorliegt, von rechts aus betrachtet) genau k Positionen mit

$V[j] = k = \text{Min}\ \{t \mid t \geq 1 \text{ und } M_j M_{j+1} \ldots M_{m-t-1} = M_{j+t} M_{j+t+1} \ldots M_{m-1}\}$
bzw. V[j] = m-j, falls ein solches t nicht existiert.

Wenn die Verschiebungen V bekannt sind, so erhöhen wir bei einer Verschiedenheit an der Position j die rechte Grenze R also nicht um 1, sondern um V[j].

Somit erhalten wir aus der alten Methode substring mit dieser Modifizierung die neue Methode substringBM:

```
public static int substringBM (String T, String M) {
    int LT = T.length(); int LM = M.length(); int i, j;
    if (LM <= LT) {
        int [] V = new int [LM];
        < hier fehlt noch die Berechnung des Feldes V >
        int R = LM-1;
        while (R < LT) {
            i = R; j = LM-1;
            while ((j >= 0) && (T.charAt(i) == M.charAt(j))) {i--; j--;}
            if (j == -1) return (i+1);
            else R += V[j];
        }
    }
    return -1;
}
```

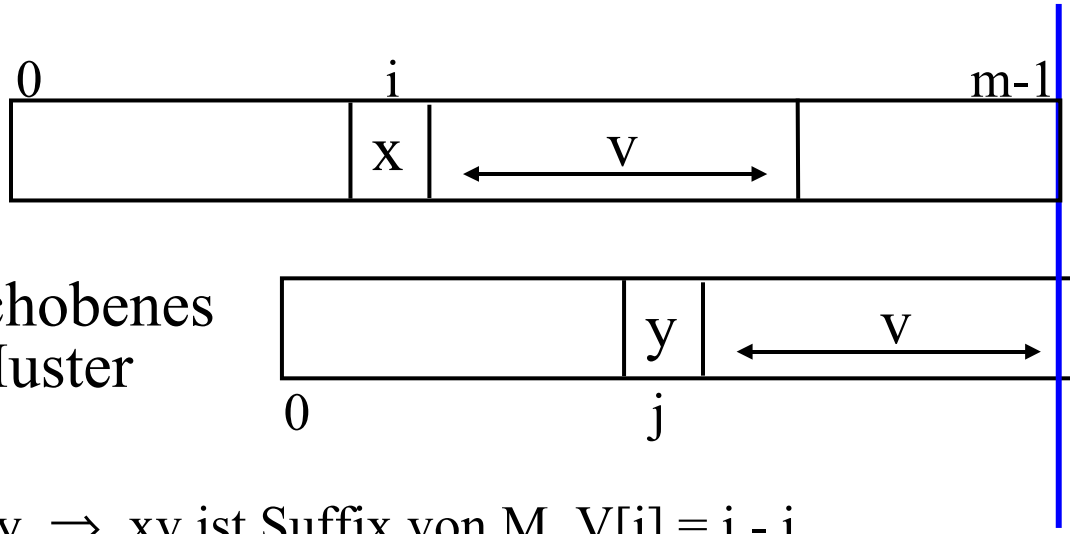
Wie berechnet man das Feld V?
Beispiel: Muster M = "abbabab". m = 7.
Für j = m-1 = 6 muss man prüfen, ob das Suffix "b" ein echtes Anfangswort ≠ ε besitzt, welches Suffix von M ist. Dies ist nicht möglich, da "b" die Länge 1 besitzt, d. h. V[6] = 7 - 6 = 1.

Für j = 5 muss man prüfen, ob das Suffix "ab" ein echtes Anfangswort ≠ ε besitzt, welches Suffix von M ist. Ein solches Anfangswort existiert hier nicht, d. h. V[5] = 7 - 5 = 2.

Für j = 4 muss man prüfen, ob das Suffix "bab" ein echtes Anfangswort ≠ ε besitzt, welches Suffix von M ist. Dies trifft für "b" mit k = 2 zu, d. h. V[4] = 2. Usw.

So erhält man das Feld V = (5,5,2,2,2,2,1).

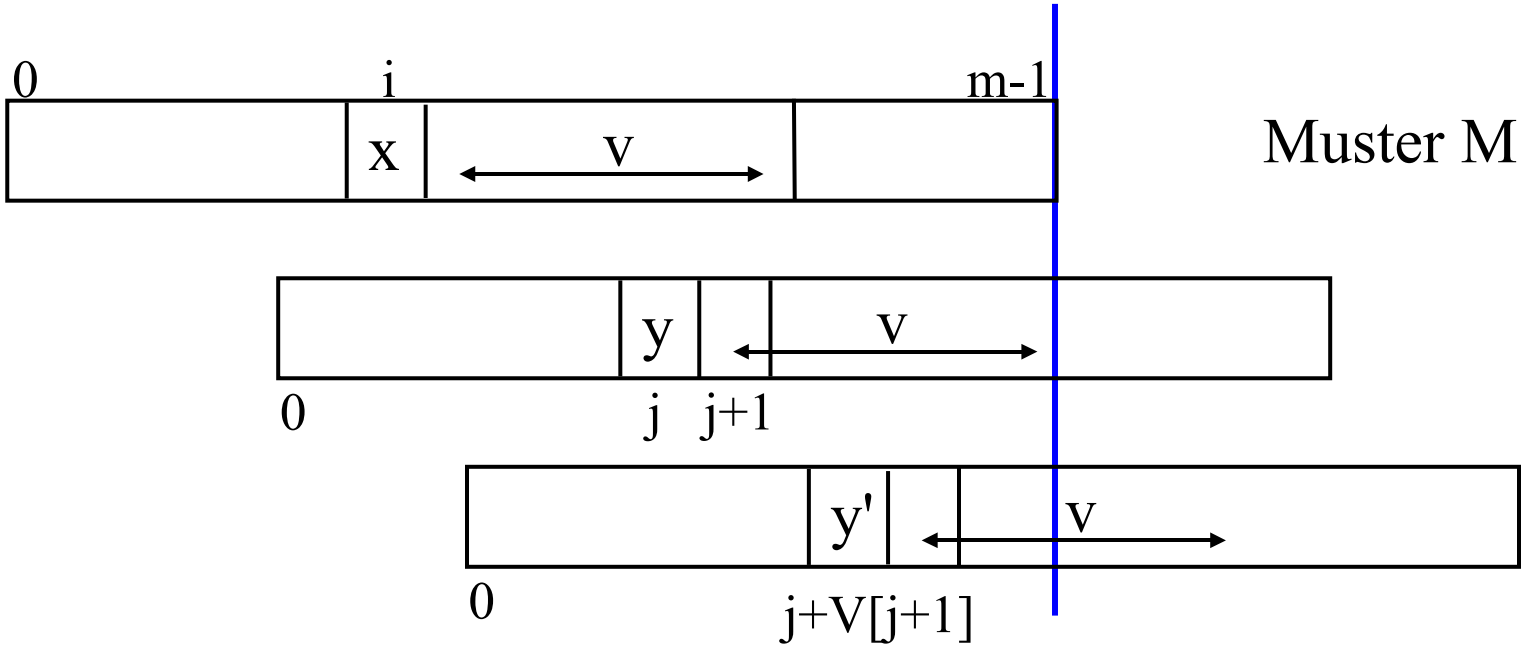
Das Feld V lässt sich rasch mit folgender Überlegung berechnen:



Beachte: v ist Suffix von M

$x = y \Rightarrow$ xv ist Suffix von M, V[i] = j - i

$x \neq y \Rightarrow$ Teste, ob gilt: x = y' = Zeichen links von "verschiebe v" bzgl. j



Dies führt zu folgendem Algorithmus, um das Feld V zu berechnen.

```
int [] V = new int [LM+1]; int i = LM - 1; int j = LM;
V[LM] = 1;                          // beachte, es gilt in der Schleife stets  j > i
while (i >= 0) {
   if ((j = LM) || (T.charAt(i) == M.charAt(j))) {V[i] = j - i; i--; j--; }
   else  j = j+V[j+1];
}
```

Man kann sich überlegen, dass dieses Programmstück nur O(m) Schritte benötigt. (Warum? Es sieht doch nach $O(m^2)$ aus?!)

Es entsteht schließlich folgendes Programm (wobei wir in Java keine gleichen Namen in Unterblöcken deklarieren dürfen, also i und j im Inneren durch i1 und j1 ersetzen):

```
class TeilwortBM2 {
   static String Text = "text.....";   static String Muster = "....";
   public static int substringBM2 (String T, String M) {
      int LT = T.length(); int LM = M.length(); int i, j;
      if (LM <= LT) {
         int [] V = new int [LM+1]; int i1 = LM - 1; int j1 = LM;
         V[LM] = 1;                    // beachte, es gilt in der Schleife stets  j1 > i1
         while (i1 >= 0) {
            if ((j1 == LM) || (T.charAt(i1) == M.charAt(j1)))
               { V[i1] = j1 - i1;  i1--;  j1--; }
            else  j1 = j1+V[j1+1];
         }
         int R = LM-1;
         while (R < LT) {
            i = R; j = LM-1;
            while ((j >= 0) && (T.charAt(i) == M.charAt(j))) {i--; j--;}
            if (j == -1) return (i+1);
            else R = R + V[j];
         }
      }
      return -1;
   }
```

```
    public static void main (String [] args) {
        String T = Text; String M = Muster; int Position = substringBM2(T,M);
        System.out.print("Das Muster ist im Text ");
        if (Position < 0) System.out.print("nicht");
        else System.out.print("ab Position " + (Position+1));
        System.out.println(" enthalten.");
    }
}
```

Konnten Sie dieser Herleitung gut folgen? Dann wird Sie Folgendes überraschen. Gibt man den Text
T = "aabaabbbabaabaaba" und das Muster M = "abaaba" ein, so erhält man die korrekte Ausgabe:

Das Muster ist im Text ab Position 9 enthalten.

Nimmt man hingegen den im Programm angegebenen Text und Muster, so erhält man die falsche Ausgabe:

Das Muster ist im Text nicht enthalten.

Da muss also irgendwo ein (kleiner) Fehler bei der Berechnung des Feldes V sein. Können Sie ihn finden und das Programm korrigieren?

Diese Beschleunigung ist in der Praxis nützlich. Sie hilft auch bei unserem Beispiel

$$T = aa...aa = a^n \text{ und } M = baa...aa = ba^{m-1}.$$

Die Verschiedenheit zwischen Fensterausschnitt und Muster wird erst festgestellt, wenn das 'b' des Musters mit dem Text verglichen wird. Es ist aber V[0] = m, weil es kein echtes Präfix von ba^{m-1} gibt, das zugleich Suffix von ba^{m-1} ist.

Das Groß-O-Verhalten der Suche wird also hier zu O(n).

In der Praxis verschiebt man um das Maximum der Verschiebungen, welche Ansatz 1 und Ansatz 2 liefern.

Aufwand:

Man kann beweisen: Verwendet man die beiden Ansätze im Boyer-Moore-Algorithmus, so ergibt sich ein **O(n+m)**-Verfahren. (Insgesamt werden höchstens 3·(n+m) Vergleiche durchgeführt. Der Beweis ist schwierig, siehe Artikel von Cole.)

In der Praxis zeigt der Boyer-Moore-Algorithmus ein **O(n/m + m)**-Verhalten!

Variante: Suffix with mismatch

Die obige Tabelle V nutzt die Information nicht aus, dass an der j-ten Stelle des Fensterausschnitts ein Zeichen steht, das ungleich M[j] ist. Wenn nun bei der Verschiebung des Musters genau an diese Stelle das Zeichen M[j] zu liegen kommt, so kann man weiter verschieben.

Diese Variante setzt eine genauere Betrachtung beim Erstellen von V voraus, die aber am linearen Verhalten, V aufzubauen, nichts ändert.

1.6 Java (Teil 3)

1.6.1 Pakete

1.6.2 Style Guides

1.6.3 Bemerkungen (nur in der Vorlesung)

1.6.1 Pakete

Das Anlegen eines Pakets mit Namen xyz ist abhängig von Ihrem Javasystem.
Zum Beispiel: Verzeichnis mit Namen xyz anlegen (entweder oberhalb Ihres Workspace oder bei eclipse in einem der Projekte), Einlagern der gewünschten Klassen, übersetzen der Klassen, eventuell Hinzufügen des Verzeichnispfades zum Klassennamen usw.

Die Klassen eines Pakets können Sie verwenden, indem Sie

```
import xyz.*;
```

an den Anfang Ihres Java-Programms schreiben. Wollen Sie hieraus nur eine Klasse mit Namen "eineKlasse" oder ein Unterpaket "unter" benutzen, so schreiben Sie

```
import xyz.eineKlasse;    oder    import xyz.unter.*;
```

Der Name eines Pakets ist der vollständige Name und nicht eine Folge von Unterstrukturen. Der Name

java.lang

bezeichnet also genau dieses Paket und sagt nichts über eine Hierarchie aus; insbesondere bedeutet dieser Name nicht: "gehe in den Ordner java und suche dort nach einem Ordner namens lang". Sie könnten ein Paket

java.lang.heute

einführen, welches aber nicht etwa in einem Verzeichnis, das zu java.lang gehört könnte, zusätzlich eingefügt wird; vielmehr besitzt es nur den Gesamtnamen java.lang.heute unabhängig von sonstigen ähnlich klingenden Namen.
Hier entstehen Namenskonflikte, für die es in Java eindeutige Auflösungsregeln gibt. Klären Sie diese, bevor Sie mit Paketen arbeiten. (Zum Beispiel: Zuerst alle explizit genannten import-Vereinbarungen ohne "*", dann namenslose Pakete der aktuellen Umgebung, danach import-Vereinbarungen mit "*" usw.)

Das Paket java.lang

"lang" ist hier die Abkürzung für "language" (= Sprache).

Dieses Paket ist "vordefiniert" und immer standardmäßig verfügbar, d.h., es muss nicht mittels import hinzugeladen werden. Ohne dieses Paket funktionieren die Basis-Funktio-nalitäten von Java nicht.

In diesem Paket sind enthalten: Die Klasse Object, die Klassenbildung (class), die Klasse System (die aktuelle Umgebung, in der die Java-Programme ablaufen), die Hüllenklassen, die Zeichenketten usw. Eine Auflistung finden Sie auf der folgenden Folie.

Für viele Anwendungen ist die Klasse java.lang.Math hierin hilfreich, die die wechselseitige Umwandlung von Zahlen und diverse numerische und trigonometrische Funktionen umfasst.

Es folgen konkrete Beispiele für Pakete.

Inhalt des Pakets java.lang:

Interfaces: CharSequence, Cloneable , Comparable, Runnable

Klassen: Boolean, Byte, Character, Character.Subset, Character.UnicodeBlock, Class, ClassLoader, Compiler, Double, Float, InheritableThreadLocal, Integer, Long, Math, Number, Object, Package, Process, Runtime, RuntimePermission, SecurityManager, Short, StackTraceElement, StrictMath, String, StringBuffer, System, Thread, ThreadDeath, ThreadGroup, ThreadLocal, Throwable, Void

Exceptions: ArithmeticException, ArrayIndexOutOfBoundsException, ArrayStoreException, ClassCastException, ClassNotFoundException, CloneNotSupportedException, Exception, ExceptionInInitializerError, IllegalAccessException, IllegalArgumentException, IllegalMonitorStateException, IllegalStateException, IllegalThreadStateException, IndexOutOfBoundsException, InstantiationException, InterruptedException, NegativeArraySizeException, NoSuchFieldException, NoSuchMethodException, NullPointerException, NumberFormatException, RuntimeException, SecurityException, StringIndexOutOfBoundsException, UnsupportedOperationException

sowie 20 Error-Klassen.

Das Paket java.util

Dieses Paket umfasst 34 Klassen und 13 Interfaces. Es enthält viele der Datenstrukturen, die Sie in der Grundvorlesung Informatik kennen gelernt haben: Vektoren, Stacks, Bitvektoren, Verzeichnisse, Hashtabellen, Mengen, Listen, sortierte Mengen, Bäume, StringTokenizer usw. Man kann auch eigene Zufallszahlengeneratoren bauen.

Das Paket java.io

Dieses (riesige) Paket realisiert die Ein- und -Ausgabe in Java mit Hilfe der Klassen InputStream, OutputStream, Reader, Writer usw. .

Das Paket java.net dient dem einfacheren Umgang mit Client-Server-Anwendungen im Netz.

Das Paket java.math

Dieses Paket enthält die zwei Klassen BigDecimal und BigInteger, die eine beliebige Genauigkeit beim Umgang mit (rationalen) bzw. ganzen Zahlen erlauben.

1.6.2 Style Guides

Programme sollen gut überschaubar, korrekt, unmittelbar verständlich, selbsterklärend, in sich widerspruchsfrei, leicht anzupassen und zu pflegen usw. sein. Diese Eigenschaften sollten aus dem Programm selbst hervorgehen und nicht auf Grund von Erläuterungen durch andere Personen.

Hierfür wurden Programmierrichtlinien herausgegeben, an die sich die meisten Java-Programmierer (oft erst nach schlechten Erfahrungen mit dem eigenen Programmierstil) auch halten. Hierzu gehören Kommentare, Wahl der Bezeichner, Einrückungen, Reihenfolgen von Sprachkonstrukten, Darstellung von Ausdrücken, "Verbote" (z.B.: shifts in Rechenausdrücken, Sprünge bei Anweisungen, zu viele Schachtelungen), Begrenzung des Programmcodes von Basisfunktionen (auf höchstens 3 Seiten) usw.

Insbesondere bei größeren Programmen, beim Testen, bei Vorgehensmodellen oder bei der Arbeit in Teams sind solche Richtlinien oft unverzichtbar. Zugleich werden sie gekoppelt mit Planungen, Methoden und Dokumentationen, damit z. B. die Kommunikation mit einheitlichen Normen in verständlicher Weise erfolgen und Festlegungen schnell und eindeutig ermittelt werden können.

Zu den aktuellen Empfehlungen suche man im Internet, z.B. nach den "Java Code Conventions". Bei der Suche nach Beispielen für undurchschaubarem Code fahnde man nach Begriffen wie "unmaintainable code", "obfuscation" oder Spaghetti-Code.

Und: Style Guides werden in unseren Vorlesungen des Diplom-Studiengangs Softwaretechnik intensiv behandelt.

1.7 Hinweise zu objektorientierten Sprachen

Die wichtigen Prinzipien der objektorientierten Programmierung finden sich bereits in der Grundvorlesung. Um sie zu erreichen, haben Sie folgende Konzepte kennen gelernt:

- Klassen, Vererbung und die Hierarchie der Klassen (ausgehend von "Object")
- Dynamische Bindung (late binding)
- Polymorphie
- Ausnahmen (exceptions)

Oft fügt man noch "Spezifikationen" (abstrakte Klassen und interfaces), getrennte Übersetzung, Nutzung von Bibliotheken (auch Pakete) und die Verwendung von Mustern hinzu.

Weiterhin gibt man Ziele der ooP an, z. B.:

- Wiederverwendbarkeit
- Adaptionsfähigkeit, Anpassbarkeit
- Portabilität
- Modularität, Kapselung

Sodann gibt man Hinweise zum "Programmieren im Großen". Gerade die objektorientierte Herangehensweise ermöglicht es, kaum überschaubare Probleme in Angriff nehmen und erfolgreich implementieren, warten, einsetzbar machen und meist auch beherrschbar halten zu können.

Mehr hierzu in der Grundlagenvorlesung über Software Engineering und in Veranstaltungen der Praktischen und der Angewandten Informatik.

Objektorientierte Sprachen:

Smalltalk (ab 1980)

C++ (1986), Java (ab 1995), C# (ab 2000)

Eiffel (1988)

Object Pascal, Delphi, Oberon, Oberon-2 (1986-1991)

Hinzuzählen sind weiterhin:
Simula 67, CLOS, Modula-3, Ada 95, Ada 2005, Beta, Visual Objects und etwa 30 weitere weniger verbreitete Sprachen.