Algorithmen und Programmierung II

Prof. Dr. Erich Ehses

FH Köln
Campus Gummersbach

Sommersemester 2014

Inhaltsverzeichnis

Kapitel 1

Einleitung

The purpose of the Smalltalk project
is to provide computer support
for the creative spirit
in everyone.
Daniel Ingalls, 1981

Im ersten Semester haben Sie die Grundlagen von Java und C kennengelernt. C war die Grundlage für die Einführung in die *prozedurale Programmierung*. Die prozedurale Programmierung sieht ein Programm als eine Abfolge von Befehlen. Diese Befehle sind der Verständlichkeit und auch der Wiederverwendbarkeit halber in Prozeduren (in C Funktionen genannt) gegliedert. Der Entwurf eines prozeduralen Programms verlangt den Entwurf von Prozeduren und das Verständnis ihrer Interaktion. Typische Darstellungsformen für die prozedurale Programmierung sind Ablaufdiagramme und Aufrufdiagramme.

Java hat demgegenüber zum Ziel die *objektorientierte Programmierung* zu unterstützen. Es ermöglicht aber immer noch ein rein prozedurales Vorgehen. Um Sie nicht im Erlernen der Ähnlichkeiten und Unterschiede zu C zu überfordern, haben Sie bisher vorwiegend prozedurale Programme gesehen.

Einige von Ihnen werden zu Beginn diesen Semesters einen Schock erleben: Die objektorientierte Programmierung ist anders als Sie dachten! Das Merkmal der objektorientierten Programmierung ist *nicht*, dass Variablen und Methoden in Klassen deklariert werden! Objektorientierung hat auch nicht zum bloßen Ziel, ein lauffähiges Programm „hinzubekommen“, das gerade das macht, was es soll. **Das Ziel der Softwareentwicklung besteht darin, ein korrektes, effizientes und verständliches System zu entwickeln, das als Basis für spätere Weiterentwicklung brauchbar ist.**

Die Vorgehensweise der objektorientierten Programmierung fasst ein Programm als eine Menge von Objekten auf. In Java werden gleichartige Objekte durch Klassen beschrieben.[1] Die Objekte verfügen über Methoden, die ihr Verhalten bestimmen. Klassendiagramme beschreiben wie Klassen und ihre Objektinstanzen zusammenhängen. Im Unterschied zur prozeduralen Programmierung ist der Ablauf eines Programms dagegen nur mit großer Mühe zu erkennen. Der prozedurale Programmierer kann ein objektorientiertes Programm kaum verstehen.

Um Objektorientierung zu verstehen, müssen Sie umlernen!

Der erste Teil der Vorlesung besteht darin, Sie in das objektorientierte Vorgehen einzuführen. Dabei werden auch ein paar neue Sprachkonstrukte von Java eingeführt.

[1] Das gilt für die meisten objektorientierten Sprachen. Es gibt aber auch Objektorientierung ohne Klassen!

Im zweiten Teil werden wir uns mit der systematischen Entwicklung von Algorithmen beschäftigen und einige Algorithmen beispielhaft näher kennenlernen. Gleichzeitig dient dieser Teil auch der Vertiefung und Einübung der Programmierkenntnisse.

In diesem Semester kommen zwar noch ein paar neue Spracheigenschaften hinzu; das Schwergewicht wird sich aber von der Definition der Sprache Java stärker auf ihren Gebrauch verlagern. Dabei werden Sie Vorgehenweisen kennenlernen, die an keine besondere Programmiersprache gebunden sind.

In diesem einführenden Kapitel gebe ich Ihnen einen groben Überblick über das, was Sie in diesem Semester erwartet.

1.1 Softwarequalitäten

Der Nutzen und die Kosten der Softwareentwicklung werden entscheidend durch die Eigenschaften der erstellten Software bestimmt. Diese Eigenschaften nennt man *Softwarequalitäten*. Man unterscheidet die Softwareeigenschaften, die für den Softwareanwender erkennbar sind (*externe Qualitäten*), von den Eigenschaften, die nur für den Softwareentwickler von unmittelbarem Interesse sind (*interne Qualitäten*).

Die folgende Liste stellt einige wichtige externe Softwarequalitäten zusammen:

Korrektheit Die Software muss bei korrekter Eingabe die vorgegebenen funktionalen Anforderungen erfüllen und die gewünschten Ergebnisse liefern.

Robustheit Die Software soll auch bei fehlerhafter Eingabe ein angemessenes Verhalten zeigen.

Bedienbarkeit Die Nutzung des Systems muss möglichst einfach sein. Soweit wie möglich sollten die Häufigkeit von Benutzerfehlern und ihre Auswirkungen minimiert werden.

Anpassbarkeit Da sich äußere Gegebenheiten mit der Zeit verändern, muss es möglich sein, ein Softwaresystem mit vertretbarem Aufwand an diese Änderungen anzupassen.

Portierbarkeit Ein Produkt sollte einfach auf anderen Hardwareplattformen und Systemumgebungen übertragbar sein.

Effizienz Ein System muss den äußeren Randbedingungen genügen. Dies sind insbesondere die Systemkosten, die Kosten der Systemvoraussetzungen (Hardware), das geforderte Laufzeitverhalten (Laufzeiteffizienz) und die Kosten der Systemeinführung.

Diese Punkte betreffen die Softwarequalitäten, die für den Endbenutzer erkennbar sind. Erreicht werden viele dieser Qualitäten, durch die internen Softwarequalitäten, die nur in den Quelltexten und den Softwaredokumenten unmittelbar sichtbar werden.

Wichtige interne Eigenschaften sind:

Modularität Ein großes System ist modular aufgebaut, wenn es sich einfach in Teilkomponenten zerlegen lässt. Bei kleinen Programmen bietet Modularität keinen unmittelbar erkennbaren Vorteil. Größere System lassen sich jedoch nur dann vernünftig und kostengünstig entwickeln, wenn von Anfang an auf eine modulare Struktur geachtet wird.

Verifizierbarkeit Ein System ist verifizierbar, wenn sich seine Korrektheit einfach zeigen lässt. Die Verifikation hat mit der planmäßigen Entwicklung der Software zu tun.

Testbarkeit Ein System ist testbar, wenn es so aufgebaut ist, dass mögliche Programmierfehler einfach aufzufinden und zu korrigieren sind. Das Testen ist die Überprüfung, ob man nicht doch etwas übersehen hat.

Verwendung von Standardkomponenten Der Einsatz von (gekauften oder selbst entwickelten) Basiskomponenten ist die Voraussetzung für die effiziente Entwicklung großer Systeme.

1.2 Der Softwareentwicklungsprozess

Große Softwaresysteme werden in einem Prozess entwickelt, an dem mehrere Personen beteiligt sind, und der sich über Jahre hinziehen kann. Die damit verbundenen Planungs- und Managementprobleme werden heute immer noch nicht zufriedenstellend beherrscht. Die Fortschritte in der *Softwaretechnik* werden nämlich durch erhöhte Anforderungen an den Umfang und an die Funktionalität der Software infrage gestellt. Insbesondere gibt es aber auch zu wenige qualifizierte Entwickler.

Die anerkannten Methoden der Softwaretechnik lassen sich ganz grob vereinfacht in zwei Bereiche unterteilen: in *Programmierung im Großen* und in *Programmierung im Kleinen*.

1.2.1 Programmierung im Großen

Durch die Einführung eines Softwaresystems werden in der Regel bestehende Methoden der Informationsverarbeitung verändert oder ersetzt. Daher fängt Softwareentwicklung – besser sollte man sagen *Systementwicklung* – bei der *Analyse* bestehender Systeme an. Aus dieser *Ist-Analyse* entwickelt sich ein *Forderungskatalog*, der die *Anforderungen* an ein mögliches neues System beschreibt. Erst danach kann man entscheiden, ob es überhaupt möglich und sinnvoll ist, eine computerbasierte Lösung zu suchen.

Die Qualität dieser Vorarbeiten entscheidet häufig über Erfolg oder Misserfolg eines Projekts. Wir können in dieser Vorlesung nicht näher auf dieses Gebiet eingehen. Sie sollten aber für die Programmierung einen wichtigen Punkt mitnehmen:

Merksatz:

Software ist immer in vorgegebene äußere Abläufe eingebettet. Softwarequalitäten lassen sich nur auf dem Hintergrund der äußeren Anforderungen und Randbedingungen beurteilen.[2]

Die verschiedenen Vorgehensmodelle für die Softwareentwicklung werden in dem Fach Softwaretechnik behandelt.

Die Erfahrungen der Softwareentwicklung zeigen auch, dass es nicht möglich ist, im Voraus alle Eigenschaften eines Systems vorauszubestimmen. Ein wichtiger Grund liegt auch darin, dass sich die Anforderungen oft schon während der Entwicklung verändern. Dies hat wichtige Konsequenzen für die Art und Weise wie Software strukturiert sein soll.

[2] Diese Aussage gilt übrigens auch für die Beurteilung und Auswahl der zu verwendenden Programmiersprache.

Merksatz:

Software muss so strukturiert werden, dass sie leicht veränderbar ist. Veränderbarkeit und die nötige Wartbarkeit erreicht man nur dadurch, dass man ein System so aufbaut, dass einzelne Teile für sich verändert werden können, ohne dass gleichzeitig andere Softwarekomponenten beeinflusst werden.

Die objektorientierte Programmierung stellt eine Leitlinie für die Zerlegung eines großen Systems dar. Die Programmiersprache Java fördert. diese Methodik dadurch, dass sie Sprachkonstrukte anbietet, die die objektorientierte Zerlegung unterstützen.

In diesem Kurs werde ich die Gesichtspunkte der Programmierung im Großen nicht behandeln, sie sind ja Bestandteil des Faches Softwaretechnik. Ich versuche jedoch, Ihnen durch die aufeinander aufbauenden Programmbeispiele einige Aspekte zu verdeutlichen.

1.2.2 Programmierung im Kleinen

Unter Programmierung im Kleinen versteht man die Methoden, die für die Entwicklung korrekter und effizienter Softwarekomponenten nötig sind. Auch wenn sie für das Gelingen großer Projekte nicht so entscheidend sind wie die Gesichtspunkte der Programmierung im Großen, so ist ihre Bedeutung jedoch nicht unerheblich: Programmierfehler werden schließlich im Kleinen gemacht! Viele Systemüberlegungen lassen sich nicht verstehen, wenn man nicht vorher eine solide Grundlage erworben hat.

Ein großer Teil dieses Semesters wird sich mit unterschiedlichen Aspekten der Programmierung im Kleinen befassen.

Leider fehlt dabei die Zeit, um die systematische Entwicklung korrekter Software angemessen zu besprechen. Ich habe die Erfahrung gemacht, dass Studenten auch nur schwer dazu zu bewegen sind, Programmierung mit formalen Methoden zu verbinden. Trotzdem werde ich zu Beginn des Algorithmenteils einen kurzen Einblick geben. Ich glaube nämlich, dass die dort vermittelte Denkweise des Denkens in logischen, invarianten *Aussagen* über Programmeigenschaften, das Verständnis und die Programmierfähigkeit deutlich verbessert.

1.2.3 Objektorientierte Programmierung

Der zentrale Gesichtspunkt der objektorientierten Methodik besteht darin, dass man große Systeme in eine Anzahl miteinander verbundener und untereinander agierender Objekte zerlegen kann. Das Neue an der Objektorientierung – im Unterschied zu anderen Verfahren der Modularisierung – ist, dass sie es erlaubt, diese Objekte beliebig auszutauschen und erst zur Laufzeit dynamisch festzustellen, welche Klasse die Operationen eines Objekts implementiert.

Nach allgemeiner Definition (hier beziehe ich mich auf die Wikipedia[3]) ist das Paradigma der Objektorientierung durch vier wichtige Merkmale gekennzeichnet.

Abstraktion Jedes Objekt im System kann als ein abstraktes Modell eines Akteurs betrachtet werden, der Aufträge erledigen, seinen Zustand berichten und ändern und mit den anderen Objekten im System kommunizieren kann, ohne offen legen zu müssen, wie diese Fähigkeiten implementiert sind (vgl. abstrakter Datentyp (ADT)).

[3] `http://de.wikipedia.org`

Kapselung Objekte können den internen Zustand anderer Objekte nicht in unerwarteter Weise lesen oder ändern. Ein Objekt hat eine Schnittstelle, die darüber bestimmt, auf welche Weise mit dem Objekt interagiert werden kann. Dies verhindert das Umgehen von Invarianten des Programms.

Polymorphie Verschiedene Objekte können auf die gleiche Nachricht unterschiedlich reagieren. Die Auswahl der Methode, die auf eine Nachricht reagiert, erfolgt zur Laufzeit, Dies nennt man auch *späte Bindung*.

Vererbung Neue Klassen können auf der Basis bereits vorhandener Klassen definiert werden. Dabei übernehmen sie die Eigenschaften der Ausgangsklasse, können diese aber auch erweitern und in der Auswirkung verändern (*override, überschreiben*). In Java wird durch die Vererbung gleichzeitig eine Typbeziehungen zwischen Klassen hergestellt.

Im ersten Semester wurde der Schwerpunkt auf das Erlernen wichtiger Syntaxkonstrukte gelegt. Dabei wurde die Bedeutung von Modularisierung und Datenkapselung Wert betont. Der Aspekt der Abstraktion wurde ebenfalls angesprochen. Von Anfängern wird die Wichtigkeit von Abstraktion aber oft erheblich unterschätzt. Auch Polymorphie und Vererbung unterstützen die abstrakte Formulierung von Problemlösungen.

Die wichtigsten neuen Inhalte von Java simd:

- Typkonzept und Schnittstellen
- Vererbung, späte Bindung und Polymorphie

Zu Anfang habe ich den Unterschied von prozeduraler und objektorientierter Programmierung angesprochen. Ich will dieses Thema hier nochmals aufgreifen.

Vielleicht haben Sie bisher Objektorientierung nur darauf bezogen, dass Daten und Methoden in Klassen eingekapselt sind. Wenn eine Programmiersprache nur diesen Aspekt unterstützt, nennt man sie *objektbasiert* aber eben nicht objektorientiert. Für die Objektorientierung ist zusätzlich nötig, dass eine Klasse eine definierte Schnittstelle hat und, dass es möglich ist, die Verwandtschaft verschiedener Schnittstellen exakt zu definieren. Kurz, *Objektorientierung unterstützt die abstrakte Beschreibung des Verhaltens von Objekten.* Dies wird erreicht, indem man in einer Variablen Objekte unterschiedlichen Typs speichern kann. Dadurch wird es möglich, in einer Datenstruktur (z.B. einem Feld) Objekte von verschiedenen Klassen zu speichern (dies ist ein Sonderfall von *Polymorphie*). Ebenso ist es möglich allgemeine Algorithmen für nicht genau festgelegte Objekttypen zu schreiben. In der Konsequenz legt der Programmtext nicht mehr eindeutig fest, was bei dem Aufruf einer Operation (Methode) zu geschehen hat.

Da Sie zunächst C und dann die weniger objektorientierten Bestandteile von Java kennengelernt haben, trifft für Sie vielleicht auch das folgende Zitat von Bruce Eckel [Eck1999] zu, der sich auf den Übergang von C zur objektorientierten Programmierung in C++ bezieht.

> Man erkennt leicht die Vorteile, die die Code-Organisation durch die gemeinsame Gruppierung von Datenstrukturen und auf diesen operierenden Funktionen, erfährt. Die meisten Programmierer, die Erfahrung mit C haben, sehen die Nützlichkeit sofort, da sie genau das zu tun versuchen, sobald sie eine Softwarebibliothek aufbauen. ...

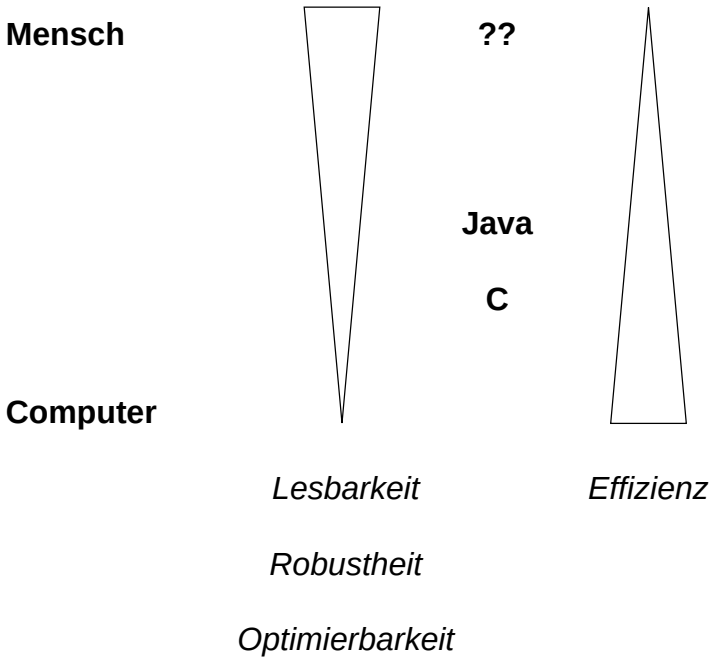

Abbildung 1.1: Programmiersprachen stehen zwischen den Kommunikationsgewohnheiten des Menschen und der Architektur der Computerhardware. Man scheint beim Programmieren auf einer höheren Abstraktionsebene den größeren Komfort und die größere Fehlersicherheit mit Laufzeitnachteilen bezahlen zu müssen. Das ist nur zum Teil der Fall! Größerer Abstraktionsgrad bedeutet auch mehr Spielraum für automatische Optimierung!

> Man bleibt leicht auf der objektbasierten Ebene stehen: Sie ist leicht verständlich und man hat ohne große geistige Anstrengung bereits viele Vorteile. Man erhält leicht das Gefühl, dass man neue Datentypen erzeugt – man macht Klassen und Objekte und sendet Nachrichten an diese Objekte und alles ist schön und nett.
>
> Aber das täuscht! Wenn man hier aufhört, verpasst man den größten Teil der Sprache, nämlich den Sprung zur wahren objektorientierten Programmierung. Diesen Sprung schafft man nur mit virtuellen Funktionen [d.h. Methoden E.E.]. [4]
>
> [Methoden] erweitern das Typkonzept mehr als das bloße Einkapseln von Code in Klassen. ...
>
> Die Eigenschaften einer prozeduralen Sprache können auf der algorithmischen Ebene verstanden werden. Dagegen versteht man [Methoden] nur aus der Sicht des Softwareentwurfs.[5]

Die Abbildung 1.1 will deutlich machen, dass verschiedene Programmiersprachen an unterschiedlicher Stelle hinsichtlich der Nähe zur Computerarchitektur oder der Nähe zur menschlichen Denkweise angesiedelt sind. C ist ziemlich maschinennah. Man kann in C die Eigenschaften der Hardware effizient ausnutzen. Java hingegen ist deutlich problemorientierter. Dadurch ist Java für maschinennahe Programmierung nicht geeignet. Gleichzeitig hat Java meist (nicht immer) ein etwas schlechteres Laufzeitverhalten als C. Dies

[4] Die „normalen“ Funktionen in C++ sind genauso wenig objektorientiert wie die Funktionen in C. Nur durch den Zusatz `virtual` kann man erreichen, dass eine C++-Funktion sich wie eine Java-Methode verhält.

[5] Dies bedeutet, dass man in objektorientierten Programmen nicht so sehr den Ablauf, als vielmehr die Struktur einer Anwendung programmiert!

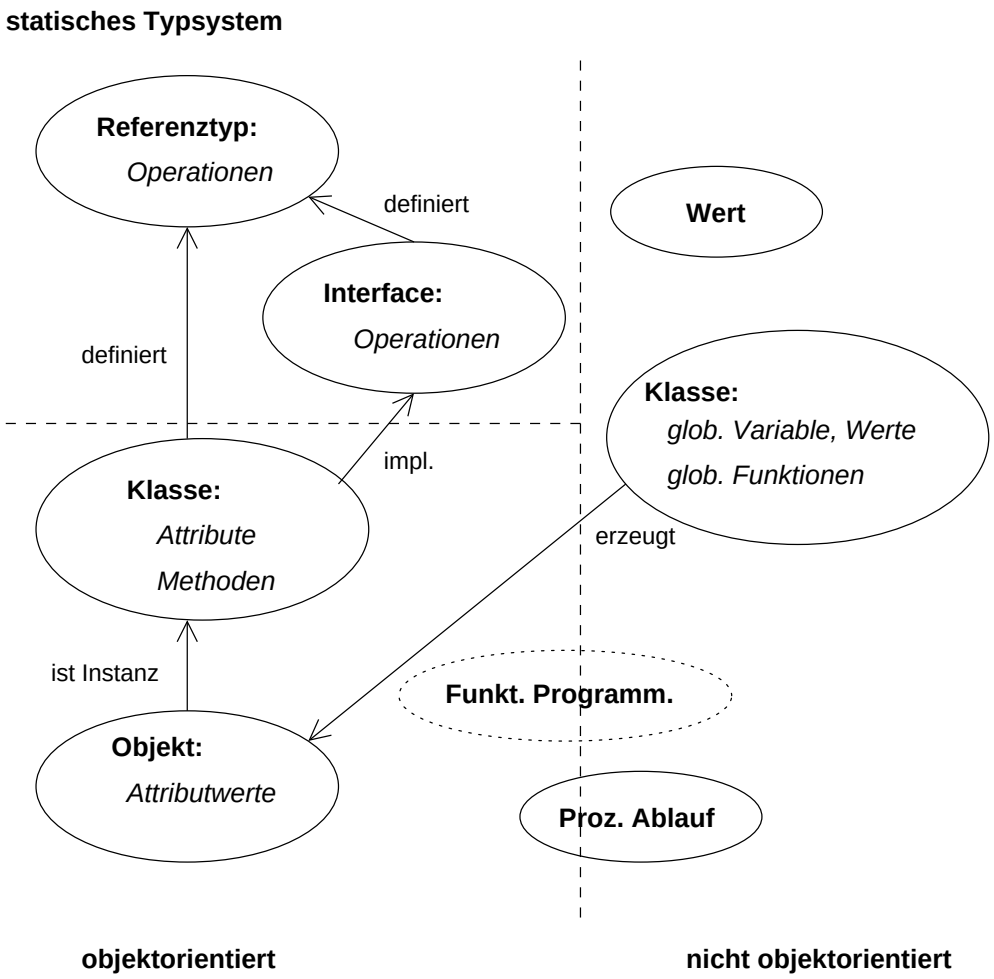

Abbildung 1.2: Java verbindet Objektorientierung mit prozeduralen Anteilen (rechts) und mit einem statischen Typsystem für Objektreferenzen (oben links). Dadurch erreicht Java gute Ausführungseigenschaften. Die gestrichelte Ellipse „Funktionale Programmierung“ weist in die Zukunft: Dann soll es möglich sein, neben der prozeduralen Programmierung von Abläufen auch die Vorteile der funktionalen Programmierung besser zu nutzen.

ist der Preis, den man für eine erheblich geringere Fehleranfälligkeit zahlen muss.

Die Abbildung 1.2 gibt einen groben Überblick über die Struktur von Java. Java ist eine aus der Praxis entwickelte Sprache. Als solche enthält sie kein „reines“ Modell der Objektorientierung. Richtig angewendet ist es kein Problem, die prozeduralen Eigenschaften mit der Objektorientierung in Java zu verbinden. Allerdings wird das Erlernen der Sprache dadurch erschwert. Im ersten Teil des Skripts werde ich fast nur auf Objektorientierung eingehen. Im zweiten Teil (Algorithmen und Datenstrukturen) kommt dann wieder stärker die prozedurale Programmierung ins Spiel.

Eine zweite Bruchlinie der Abbildung 1.2 grenzt das statische Typsystem von Java von der Objektorientierung ab (das Typsystem für elementare Wertdaten habe ich in der Abbildung nicht dargestellt). Objektorientierung und Typsystem bilden keinen Gegensatz aber zwei voneinander unabhängige Elemente.

Objektorientierte Sprachen ohne statisches Typsystem sind leichter zu erlernen als Java. Objektorientierung und Typsystem zielen nämlich in verschiedene Richtung. Objektorientierung will möglichst allgemeine Verwendung von Softwareeinheiten und *größtmögli-*

che Flexibilität erreichen. Ein Typsystem stellt dagegen eine *Einschränkung* der erlaubten Operationen dar. Das Typsystem ermöglicht dem Compiler viele Fehler vorab zu entdecken. Die Deklaration von Datentypen für Variable und für Funktionssignaturen erhöht den Dokumentationswert eines Programms. Dafür bezahlt man den Preis, dass der Umgang mit den Typregeln nicht ganz einfach zu erlernen ist.

1.3 Prinzipien der objektorientierten Programmierung

Einerseits erscheint Objektorientierung sehr „natürlich“, oder zumindest wird Objektorientierung so verkauft. Andererseits finden viele Studentinnen und Studenten objektorientierte Programmierung schwer zu verstehen und zu beherrschen. Wie kann das sein? Woran liegt das?

1.3.1 Ein Paradigmenwechsel

Für jemand, der zunächst die prozedurale Programmierung erlernt hat, sind objektorientierte Programme anfangs schwer zu verstehen. Sie oder er hat den Eindruck, dass einzelne Klassen und Methoden so gut wie nichts tun als ein paar Werte weiterzureichen und Methoden anderer Klassen aufzurufen.

Dies liegt an dem Unterschied zwischen dem prozeduralen und dem objektorientierten Programmierparadigma! Nach der prozeduralen Methodik beschreibt man das Programm als einen Ablauf, d.h. eine Prozedur, die nach und nach die Daten verändert. Das Programm ist entsprechend modular in Prozeduren (auch Funktionen genannt) gegliedert.

Gemäß der objektorientierten Programmierung, beschreibt man ein Programm aber als die Zusammenarbeit von vielen spezialisierten Objekten. Die Objekte (beschrieben durch ihre Klassen) kümmern sich immer nur um sich selbst. Sie interagieren mit anderen Objekten, indem sie diesen per Methodenaufruf Nachrichten senden.

Merksatz:

> *In einem objektorientierten Programm sollte man nicht nach dem prozeduralen Ablauf suchen – er ist in einem guten Programm kaum zu erkennen. Man sollte vielmehr versuchen, jedes Objekt aus sich heraus zu verstehen.*
>
> *Objektorientierung ist keine Erweiterung des prozeduralen Ansatzes! Es ist eine fundamental andere Herangehensweise!*

Unter anderem ist dieser Paradigmawechsel in der Forderung nach Wiederverwendbarkeit und Verständlichkeit begründet. Eine Klasse implementiert eben nicht einen komplexen Ablauf oder einen komplexen Algorithmus. Sie hat vielmehr die Aufgabe, ein einfaches Konzept so einfach wie möglich zu beschreiben. Die Funktionsweise des Gesamtsystems ergibt sich nicht so sehr aus dem *Inhalt* einzelner Klassen, als aus dem sinnvollen *Zusammenspiel* der Objekte verschiedener Klassen. Bei großen Systemen ist es nötig, Entwurfsmuster für die Kooperation von Klassen anzuwenden und wo möglich auf vorfabrizierte Frameworks zurückzugreifen.

1.3.2 Es gibt verschiedene Gründe für das Schreiben einer Klasse

Ein besonderes Problem besteht darin, dass Anfänger beim Lernen falsche Schwerpunkte setzen und sich eine schlechte Programmiermethodik aneignen. Ein Grund dafür ist, dass

Vorlesungsbeispiele und Praktikumsaufgaben leicht den falschen Weg zeigen. Beispiele sind immer nur dazu da, bestimmte Aspekte zu verdeutlichen. Sie zeigen nicht immer, worum es bei der Programmierung wirklich geht.

Man schreibt Klassen für unterschiedliche Aufgabenstellungen. Je nach Aufgabe spielen immer wieder andere Gesichtspunkte eine Rolle. Für das Praktikum geht es darum, sich etwas von der irrealen Übungssituation zu lösen und so zu tun, als hätte man eine wirkliche Aufgabe.

Was sind nun die wichtigsten Szenarien?

1. Manchmal genügen ein paar Anweisungen um ein einfaches Problem zu lösen. Das ist die typische Praktikumssituation, die aber so in der Praxis nicht vorkommt.
2. Praktische Probleme bestehen meist darin, dass man Teile einer großen Anwendung verändert oder erweitert. Hierbei kommt es darauf an, die „Philosophie“ des vorhandenen Systems zu verstehen.
3. Es ist nicht einfach, ein großes System von Grund auf zu entwickeln. Damit spätere Erweiterungen und Veränderungen noch möglich sind, ja damit das System überhaupt fertig wird, ist es nötig, eine verständliche und einheitliche Struktur zu entwickeln. Hierfür wurde die Objektorientierung geschaffen. Ihre Vorteile zeigen sich aber nur, wenn Projektleiter, Softwarearchitekten und Entwickler verstehen, worum es geht.
4. Die Anforderungen an Systeme ändern sich mit der Zeit. Dies führt zu ständigen Änderungen an Softwaresysteme. Dies wiederum bewirkt, dass viele frühe Entwurfsentscheidungen obsolet werden. Kurz, die Struktur der Software verschlechtert sich mehr und mehr. Diesem „Verschleißprozess“ kann nur durch eine gleichzeitige permanente Strukturverbesserung (*refactoring*) entgegengewirkt werden.
5. Ein Sonderfall der Systementwicklung ist die Entwicklung von Bibliotheksklassen. Hierbei kommt es in ganz besonderem Maße darauf an, Gesichtspunkte wie Vollständigkeit und Wiederverwendbarkeit zu beachten.

Diese verschiedenen Aspekte sind der Grund, dass ich in der Vorlesung manchmal auf die Gesichtspunkte des Schreiben von Bibliotheken eingehe auch wenn das vermutlich nicht das von Ihnen angestrebte Berufsbild ist. Aber es ist auf jeden Fall hilfreich, wenn man bei der Verwendung einer Bibliothek versteht, von welchen Konzepten die Bibliotheksentwickler ausgingen.

Objektorientierte Software ist „große“ Software. Gleichzeitig können Sie im AP-Praktikum aber nur kleine Projekte bewältigen. Um diesen Widerspruch zu lösen, sind die Aufgaben des 2. Semesters fast alle so gestellt, dass Sie *nur* noch Teile in vorgefertigter Software ergänzen müssen. Dies mag manchmal etwas unbefriedigend sein. Mit Sicherheit ist es nicht immer einfach. Es entspricht aber den Anforderungen der beruflichen Praxis und es dient dem Verständnis der Objektorientierung. Schließlich haben Sie in der Schule auch das Lesen vor dem Schreiben gelernt.

1.4 Überblick

Zunächst werden in dem Kapitel **Wiederholung und Vertiefung** anschließend an ein paar Bemerkungen zum Paketkonzept, zur Projekterstellung in Eclipse die Ausnahmebehandlung und die Struktur einer Standardklasse vorgestellt.

Die Ausnahmebehandlung von Java stellt eine äußerst wichtige Spracheigenschaft zur Entwicklung robuster Softwareeinheiten dar. Sie wird in der virtuellen Maschine und in der vorhandenen Java-Bibliothek selbst ausgiebig genutzt, so dass man nicht daran vorbeikommt.

Der Abschnitt über die Eigenschaften der Klasse `Object` geht auf das grundlegende Verhalten von Java-Klassen ein.

Das Kapitel **Objektorientierung** beschreibt in vertiefter Form alle wichtigen Eigenschaften der Objektorientierung in Java. Zunächst wird näher auf das der Objektorientierung in Java zugrunde liegende abstrakte Typkonzept eingegangen. Daraus ergibt sich die Konsequenz, dass in der Objektorientierung Schnittstellen wichtiger sind als Klassen. Schnittstellen bestimmen die Struktur eines Softwaresystems und Klassen beschreiben seine Implementierung im Detail. Die Formulierung abstrakter Typbeziehungen ermöglicht die Entwicklung von polymorphen und wiederverwendbaren Softwareeinheiten.

Das Thema Objektorientierung wird durch ein paar Hinweise im Kapitel **Programmierregeln abgerundet.**

Neben der Beherrschung von Entwurfsverfahren und der Beherrschung einer Programmiersprache und ihrer Entwurfsparadigmen ist die Kenntnis von grundlegenden Algorithmen und Datenstrukturen sehr wichtig.

Zunächst werden in dem Kapitel **Korrekte und effiziente Algorithmen** einige grundlegenden Merkmale von Algorithmen angesprochen. Als erstes gehört dazu der Begriff der Korrektheit. Es wird gezeigt, dass es für ein bestimmtes Problem immer verschiedene korrekte algorithmische Lösungen gibt. *Ein* Unterscheidungsmerkmal für verschiedene Algorithmen ist ihre *Komplexität*. Mit der Komplexität [6] eines Algorithmus (hier verwendet man auch den Begriff *Effizienz*) ist der Bedarf an externen Ressourcen gemeint (vor allem Laufzeit und Speicherplatz). Zur Beschreibung des Laufzeitverhaltens wird die O-Notation eingeführt.

Softwaresysteme beschreiben aber nicht nur Algorithmen und Abläufe sondern stellen immer auch Daten und ihre Zusammenhänge (möglichst effizient) dar.

Viele Datenstrukturen versuchen in der Anordnung der Daten die logische Struktur der Anwendung zu beschreiben. Beispiele sind Stadtpläne oder elektronische Schaltpläne. In diesen Fällen beschreibt die Datenstruktur den logischen Zusammenhang der Daten. In den genannten Beispielen ist dies die topologische Beziehung, d.h. die Verbindung zwischen Ortspunkten oder Schaltungspunkten. Auf der anderen Seite gibt es Datenstrukturen, die vorwiegend der bequemen Verwaltung und der effizienten Unterstützung wichtiger Funktionen dienen (z.B. effizientes Suchen). Im Kapitel **Datenstrukturen** wird nur diese zweite Ebene, nämlich die der grundlegenden Datenstrukturen behandelt. Auch wenn es für die meisten Datenstrukturen bereits fertige Implementierungen in den gängigen Klassenbibliotheken gibt, so ist ihre Kenntnis jedoch die unverzichtbare Grundlage für die Entwicklung von selbst zu entwickelnden spezialisierten Algorithmen.

Im Kapitel **Suchen und Sortieren** werden die Grundlagen von Algorithmen anhand von Algorithmen zum Suchen und Sortieren vertieft. Die angesprochenen Algorithmen sollten unbedingt jedem Informatiker bekannt sein. Ihre Behandlung dient auch dazu, Übung bei der Entwicklung und Bewertung von Algorithmen zu bekommen.

Abschließend werden Algorithmen behandelt, mittels der man *Verzeichnisse* (englisch *map*, *key - value pairs* oder *dictionary*) effizient implementieren kann. Dabei wird zum Teil auch auf vorher behandelte Datenstrukturen und Algorithmen zurückgegriffen. Die

[6] laut Webster's bedeutet das Adjektiv *complex* auch „hard to separate, analyze or solve"

Idee der *hash map* (deutsch *Streutabelle, assoziativer Speicher*) zeigt einen weiteren Ansatz zur Suche.

Das Skript ist als ein parallel zur Vorlesung lesbarer Text gestaltet. Zum Zweck des Nachschlagens und der Klausurvorbereitung müssen Sie sicher Ihre eigenen Anmerkungen und Hervorhebungen anbringen. Zusätzlich sollten Sie nach Möglichkeit aber auch andere Literatur hinzuziehen. Ich empfehle Ihnen unbedingt, sich der im Netz frei verfügbaren amerikanischen Texte zu bedienen. Erste Einstiegspunkte finden Sie auf meiner Homepage (`http://www.gm.fh-koeln.de/ehses`).

Die Themenfolge der Vorlesung ist von verschiedenen Faktoren, u.a. auch davon abhängig wie und wann einzelne Themen im Praktikum behandelt werden. Es kann daher leicht sein, dass sie sich von der Gliederung des Skriptes unterscheidet.

Kapitel 2

Wiederholung und Vertiefung

2.1 Das Paketkonzept

2.1.1 Einleitung

Haben Sie sich schon einmal die Java-Online-Dokumentation angeschaut? Wenn nicht, sollten Sie das unbedingt nachholen! Dies ist nämlich nicht nur deshalb wichtig, weil Sie nur so interessante und brauchbare Software entwickeln können, sondern auch deshalb, um an den Beispielen zu erfahren, wie man große Software strukturieren und dokumentieren kann. Jedenfalls werden Sie dabei bemerken, dass die Java-Klassen in Paketen organisiert sind. Diese Paketeinteilung ist aber nicht nur ein Ordnungsprinzip der Java-Bibliothek, sondern ist grundsätzlich für die Programmierung in Java von Bedeutung.

Bis jetzt haben wir nur wenige Klassen aus der Java-Bibliothek benutzt. Aber nehmen wir mal als ein ganz primitives Beispiel die Aufgabe, ein *Hello World* Programm zu schreiben. Wenn wir das wirklich ausführlich schreiben, sieht das Programm so aus:

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

Wir können Java-Programme natürlich einfacher schreiben. Schließlich hilft der Compiler uns aus hier mit ein paar Vereinfachungen.

In Java gilt die Regel, dass nicht nur alles in einer Klasse stehen muss, sondern dass jede Klasse Bestandteil eines Paketes (*package*) ist.

Definition:

> *Jede Java-Klasse gehört zu einem **Paket**. Die Paketzugehörigkeit wird durch eine **Package-Anweisung**, die sich auf die gesamte Datei auswirkt, festgelegt. Fehlt die Package-Anweisung, so gehören die Klassen der Datei zu dem anonymen* User-Paket. *Paketnamen sind durch Punkte* `"."` *untergliedert und durch einen Punkt von dem Klassennamen getrennt.*

Im obigen Beispiel werden die beiden Klassen `System`, und `String` angesprochen. Sie gehören beide zu dem Paket `java.lang`.

Das Paket `java.lang` enthält einige grundlegende Systemklassen. Da es so fundamental ist, braucht man bei der Verwendung von Klassen aus `java.lang` den Paketnamen nicht anzugeben.

Merksatz:

Für Klassen aus dem Paket `java.lang` *ist keine Paketangabe, also auch keine Import-Anweisung nötig.*

Bei Klassen aus anderen Paketen muss aber entweder der Paketname mit aufgeführt werden, oder es muss eine Import-Anweisung für das Paket verwendet werden.

Definition:

*Die **Import-Anweisung** legt fest, dass für eine Klasse oder für alle Klassen eines bestimmten Pakets die Angabe des Paketnamens unterbleiben kann. Sie stellt eine Abkürzung dar.*

Es gibt zwei unterschiedliche Arten des Import. In der ersten Form, der Import-Anweisung wird genau eine Klasse *importiert* und damit bekannt gemacht. In der zweiten Form, werden durch eine Anweisung alle Klassen eines Pakets importiert.

Package-Anweisung: `package` *Paketname*

Import-Anweisung: `import` *Paketname* `.` *Klassenname*
| `import` *Packetname* `.*`

Es ist zu beachten, dass Paketnamen mehrere durch Punkt (`.`) getrennte Namensbestandteile enthalten dürfen. Der Punkt ist Teil des Namens und hat ansonsten auf der Java-Sprachebene keine besondere Bedeutung. Entgegen der vielleicht naheliegenden Vermutung gibt es in Java nämlich keine Hierarchie von Unter- und Oberpaketen. Trotzdem geht man bei der Namensgebung von der inhaltlichen Verwandtschaft von Paketen aus. Der Punkt ist auch wichtig bei der Suche nach Class- und Java-Dateien (siehe unten).

Das Paketkonzept dient in erster Linie dazu, größere Software zu strukturieren. Wie leicht würde es sonst vorkommen, dass Sie einen Klassennamen verwenden, der irgendwo im System bereits benutzt wird. Oder noch schlimmer: Beim *Mischen* zweier Softwarebibliotheken stellen Sie fest, dass in beiden Bibliotheken einige Klassen denselben Namen tragen.

Die mit der Java-Entwicklungsumgebung ausgelieferten Pakete (vgl. Tabelle 2.1) sind in drei Grobbereiche gegliedert: `java` für grundlegende Systempakete, `javax` für erweiterte Pakete (wie z.B. Java-Swing) und andere Namen, wie `omg.CORBA` für Pakete von dritter Seite. Hier werden wir uns nur mit den wichtigsten Bibliotheken aus dem Bereich `java` auseinandersetzen.

Neben der systematischen Gliederung der Software dient das Paketkonzept auch dazu, Class-Files (also den Bytecode meines Java-Programms) zu lokalisieren. Im Normalfall handelt es sich dabei um eine einfache Verzeichnishierarchie. Der Class-File der Klasse `a.b.Klasse` befindet sich im Verzeichnis a und darunter dann in b in der Datei Klasse.class. Das oberste Verzeichnis der Paketnamen-Hierarchie muss sich in einem Verzeichnis befinden, das in der Umgebungsvariable `CLASSPATH` mit aufgelistet ist.

Paket	Beispielklasse	Bemerkung
`java.lang`		elementare Klassen
	`String`	Stringdatentyp (unveränderlich)
`java.util`		Algorithmen / Datenstrukturen
	`ArrayList`	Datenbehälter
	`Scanner`	Texteingabe
`java.io`		Ein-/Ausgabeklassen
	`FileReader`	Einlesen von Textdatei
`javax.swing`		Klassen für GUI-Programmierung
	`JButton`	Schaltknopf

Tabelle 2.1: Wichtige Pakete der Java-Bibliothek

2.1.2 Umgang mit Paketen auf der Kommandozeile

Das Ganze soll durch das folgende Beispiel etwas deutlicher werden. Dabei gehe ich davon aus, dass das augenblickliche Verzeichnis in `CLASSPATH` enthalten ist. Meine Hello-World-Datei sieht jetzt wie folgt aus:

```
/*
 * Das ist die Datei Nachricht.java mit der Klasse
 * nachrichten.Nachricht .
 * Sie liegt im Unterverzeichnis nachrichten.
 */
package nachrichten;

/**
 * Die Klasse speichert eine Meldung, die bei
 * Bedarf mehrfach ausgegeben werden kann.
 */
public class Nachricht {
    private String text;

    /**
     * Erzeugt ein Objekt, das eine feste Nachricht
     * enthaelt.
     * @param text Text der Meldung.
     */
    public Nachricht(String text) {
        this.text = text;
    }

    /**
     * Schreibt mehrfach die Meldung des Objekts
     * in die Standardausgabe.
     * @param anzahl Anzahl der Meldungszeilen.
     */
    public void schreibeMehrfach(int anzahl) {
        for (int i=0; i < anzahl; i++)
            System.out.println(text);
    }
}

/*
 * Dies ist die Datei Test.java
 * mit der Klasse Test
 * (gehoert nicht zu dem Paket)
 * Sie liegt direkt im Arbeitsverzeichnis.
```

```
 */
//import nachrichten.Nachricht;   // moegliche
//import nachrichten.*;           // Importanweisungen

class Test {
    public static void main(String[] argv) {
        nachrichten.Nachricht meldung =
            new nachrichten.Nachricht(argv[0]);
        meldung.schreibeMehrfach(5);
    }
}
```

Wir haben also jetzt zwei Dateien, nämlich Nachricht.java und Test.java. Die Datei Test.java liegt in unserem aktuellen Arbeitsverzeichnis; die Datei Nachricht.java liegt in dem Unterverzeichnis nachrichten. javac hat die Default-Regel, dass die erzeugten Classfiles in dem Verzeichnis des dazu gehörenden Quelltextes abgelegt werden.

Beim Übersetzen und bei der Programmausführung sollten Sie unbedingt weiter im Arbeitsverzeichnis bleiben und nicht in Unterverzeichnisse für Pakete „absteigen“. Unsere Befehle könnten wie folgt aussehen:

```
javac nachrichten/Nachricht.java
javac Test.java
```

Alternativ können wir uns auch entscheiden, die generierten Classfiles in einem eigenen Verzeichnisbaum, z.B. classes abzulegen. In diesem Fall müssen wir zweierlei tun, nämlich zunächst das Wurzelverzeichnis für diesen Verzeichnisbaum anlegen und dann bei der Übersetzung dieses Verzeichnis auch angeben:

```
mkdir classes
javac -d classes nachrichten/Nachricht.java
javac -d classes Test.java
```

Auch in diesem Fall findet sich die Paketstruktur in der Verzeichnisstruktur der Classfiles wieder. Wir haben nämlich jetzt insgesamt die folgenden Dateien (WORK steht für das aktuelle Arbeitsverzeichnis:

- WORK/Test.java
- WORK/nachrichten/Nachricht.java
- WORK/classes/Test.class
- WORK/classes/nachrichten/Nachricht.java

Preisfrage: Wie können wir jetzt das Programm starten? Dazu gibt es zwei Möglichkeiten. Entweder wir passen den CLASSPATH an oder wechseln in das Unterzeichnis classes. In beiden Fällen sieht der Aufruf ganz normal aus:

```
java Test Hallooo
```

Wir können aber jetzt auf die Idee kommen, auch die Main-Klasse `Test` in des Paket `nachrichten` zu legen. Zuerst werden wir dazu die Klassendefinition verändern.

```
/*
 * Dies ist die Datei Test.java.
 * Sie enthaelt die Klasse nachrichten.Test .
 * (gehoert zu dem Paket nachrichten).
 */
package nachrichten;

class Test {
    public static void main(String[] argv) {
        Nachricht meldung =
            new Nachricht(argv[0]);
        meldung.schreibeMehrfach(5);
    }
}
```

Wie Sie sehen, wird die Klasse dadurch sogar etwas einfacher. Es ist jetzt nämlich nicht mehr nötig, die Paketzugehörigkeit der Klasse `Nachricht` anzugeben.[1] Die Klassen befinden sich ja in demselben Paket. Sie sollten nicht vergessen, jetzt auch die Datei Test.java in das Unterverzeichnis nachrichten zu verschieben.

Wenn Sie das Beispiel testen, sollten Sie vielleicht vor der Ausführung zunächst alle Classfiles löschen und dann die Java-Dateien neu übersetzen. So verhindern Sie, dass Sie mit „Altlasten“ zu tun haben. Nach der Übersetzung sollten die Dateien Nachricht.class und Test.class beide in dem Unterverzeichnis nachrichten oder classes/nachrichten liegen.

Der Aufruf `java Test Hallo` wird jetzt nicht funktionieren. Wahrscheinlich erscheint bei Ihnen eine Meldung, wie:

```
Exception in thread "main" java.lang.NoClassDefFoundError:
    Test
```

Der Grund liegt ganz einfach darin, dass es keine Klasse `Test` mehr gibt. Die frühere Klasse `Test` heißt nämlich jetzt `nachrichten.Test`, da der Paketname ein Bestandteil des vollständigen Namens ist.

Der richtige Aufruf macht jetzt etwas mehr Tipparbeit:

```
    java nachrichten.Test Hallo
```

2.1.3 Projekte und Pakete in Eclipse

Auf der Kommandozeile gestaltet sich der Umgang mit den Paketregeln sicher etwas umständlich. Das ist einer der wichtigen Gründe zu einer integrierten Entwicklungsumgebung, wie Eclipse oder Netbeans, zu wechseln. Entwicklungsumgebungen haben ihre eigenen Regeln für den Umgang mit Paketen. Dabei werden Pakete übersichtlich dargestellt. Auch die Verwendung von Paketen wird dann so einfach, dass es keinen Grund mehr gibt, Pakete nicht zu verwenden.

[1] Allerdings ist die Paketangabe immer erlaubt, auch dann wenn es sich um eine Klasse im aktuellen Paket handelt.

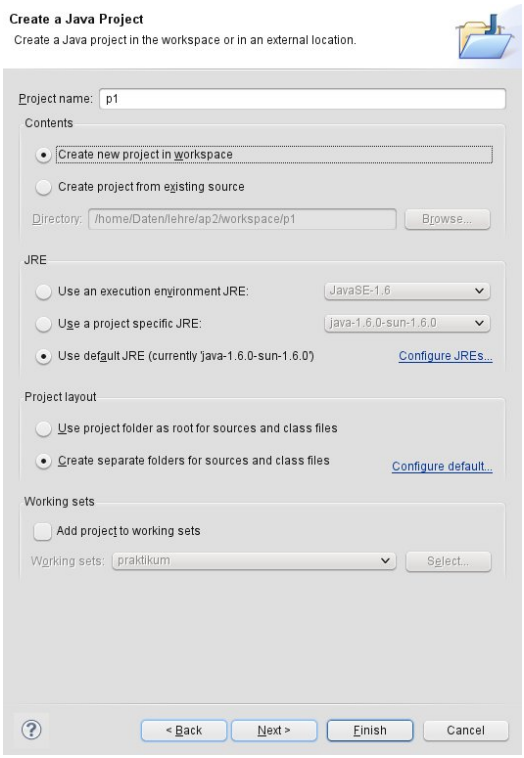

Abbildung 2.1: Maske zur Erzeugung eines neuen Java-Projekts

Übersicht über die Verzeichnisorganisation

Eclipse ordnet alle Java-Dateien Projekten zu. Zunächst ist jedes Projekt durch ein eigenes Verzeichnis im Arbeitsbereich (*workspace*) dargestellt. Die Projektverzeichnisse enthalten die Quelltexte und die Classfiles in einer Dateiorganisation, die der Paketstruktur entspricht. Projekte erlauben es, die Dateien zwecks Übersichtlichkeit auf unterschiedliche Folder zu verteilen. Projekte enthalten Verweise auf benötigte externe Bibliotheken.

Eclipse nutzt nicht die Umgebungsvariablen `CLASSPATH` und `SOURCEPATH` sondern definiert pro Projekt in der internen Datei `.classpath` eigene Pfadregeln. Die Datei kann nicht direkt editiert werden, aber sie zeigt ganz deutlich, wie die Projektorganisation zu verstehen ist:

```
<?xml version="1.0" encoding="UTF-8"?>
<classpath>
    <classpathentry kind="src" path="src"/>
    <classpathentry kind="src" path="test"/>
    <classpathentry kind="con"
        path="org.eclipse.jdt.launching.JRE_CONTAINER"/>
    <classpathentry kind="con"
        path="org.eclipse.jdt.junit.JUNIT_CONTAINER/4"/>
    <classpathentry kind="output" path="bin"/>
</classpath>
```

Sie sehen hier drei verschiedene Pfadangaben. Unter den angegebenen Verzeichnissen befindet sich die Verzeichnishierarchie der Paketstruktur. Die Angabe mehrerer Verzeich-

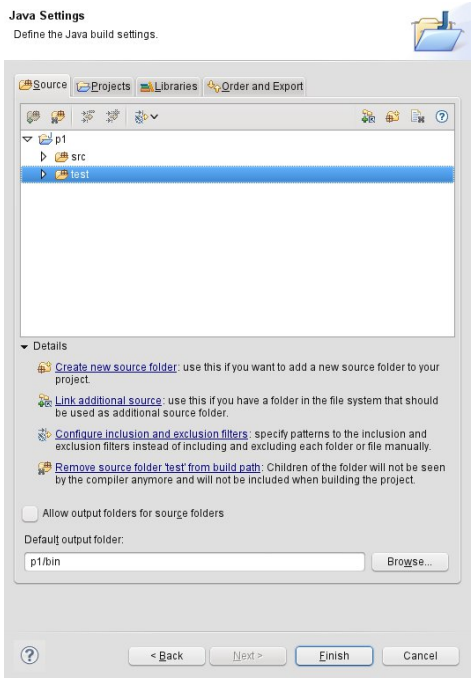

Abbildung 2.2: Quellpfade

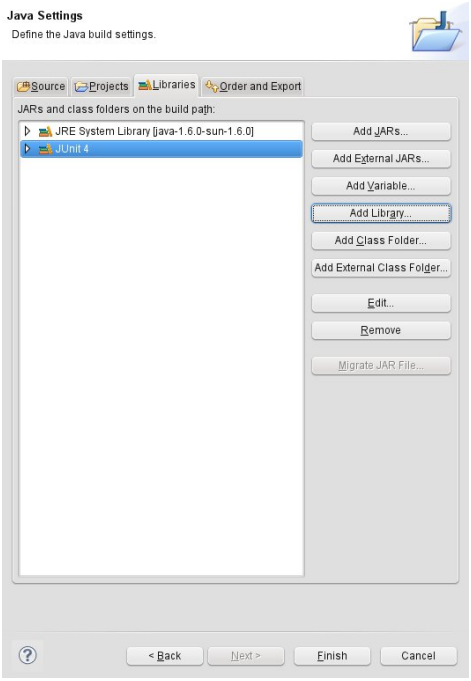

Abbildung 2.3: Bibliotheken

nisse – hier `src` und `test` – erlaubt es logisch getrennte Teile des Systems auch in der Verzeichnisstruktur zu trennen. Wir können so Quelltexte und Testdateien auseinander halten.

`"con"` benennt die Pfade für benötigte Bibliotheken. Dabei benutzt Eclipse hier nicht die eigentlichen Pfadangaben, sondern es bezieht sich auf Eclipse-eigene Variablen. Variablen dienen, dazu, dass die Projektstruktur unabhängig von dem Speicherort des Projekts wird. Die Inhalte der Variablen können durch die Voreinstellungen von `Eclipse` verändert werden.

Die Java-Bibliothek wird bei der Projekterzeugung von Eclipse automatisch eingesetzt. Der `JUNIT_CONTAINER` wurde bei der Projekterzeugung ausgewählt, um das Testen mit JUnit zu ermöglichen. Bei der Steuerung über die Kommandozeile müsste dieser Pfad in `CLASSPATH` stehen.

`"bin"` benennt das Verzeichnis für die erzeugten Classfiles. Dieser Verweis steht sonst auch im `CLASSPATH`.

Projekterzeugung

Der erste Schritt beim Arbeiten mit Eclipse ist das Anlegen eines neuen Projekts. Im einfachsten Fall ist dazu nur der Projektname anzugeben. Hier soll aber gezeigt werden, wie ein etwas komplexeres Projekt – wie es zum Beispiel im AP-Praktikum verlangt wird – aufgesetzt wird.

Zunächst wird über die Menüfunktion „New Java Project“ die Maske für die Projekterzeugung aufgerufen (s. Abb.2.1).

Def Projektname *p1* ist bereits eingetragen. Ebenso ist hier ausgewählt, dass wir (zunächst) ein neues Projekt erzeugen (alternativ hätten wir ein geeignet exportiertes Pro-

Abbildung 2.4: Leeres Projekt

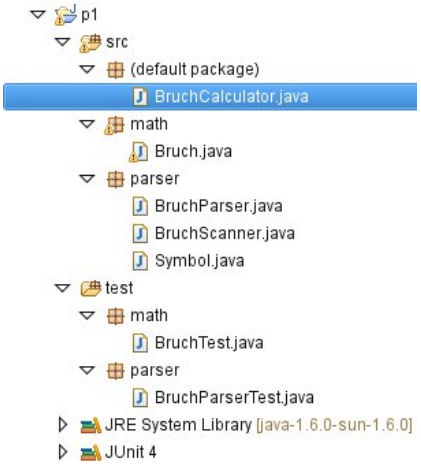

Abbildung 2.5: Projekt mit Paketen und Quellen

jekt „wiederbeleben“ können. Die automatisch ausgewählte JRE lautet bei Ihnen eventuell etwas anders. Das Projektlayout, sollte automatisch die Variante „create separate folders for source and class files“ ausgewählt haben.

Es ist jetzt wichtig, dass Sie jetzt mittels „next“ die nächste Seite zur genaueren Definition der Projekteigenschaften auswählen (man kann diese Projekteigenschaften auch später über das Projekt ändern). In den Projekteigenschaften, definieren wir die bereits angesprochenen Pfade.

Die Seite (siehe Abb. 2.2) „Define the Java build settings“ enthält mehrere Tabs. Aufgeschlagen ist „Source“. Die Abbildung zeigt den Zustand, nachdem zusätzlich zu dem schon in der vorigen Seite angesprochenen Quellverzeichnis „src“ ein weiteres Quellverzeichnis „test“ erzeugt wurde. Die Erzeugung weiterer Quellverzeichnisse kann über verschiedene Wege veranlasst werden (rechte Maustaste, Kommandoleiste des Fenster, Details).

Wir haben jetzt zwei unterschiedliche Verzeichnisse, in denen man die eigentlichen Quelltexte von den Quellen der Testklassen trennen kann.

Zusätzlich sehen Sie, dass am unteren Ende der Seite unter „Default output folder:“ das Verzeichnis `p1/bin` eingetragen ist.

Im nächsten Schritt legen wir die Pfade für Bibliotheken fest. Dazu wählen wir den Reiter „Libraries“ aus.

Hier (Abb. 2.3 ist der gewünschte Zustand dargestellt, nachdem wir die JUnit-Bibliothek ausgewählt haben. Die Java-Bibliothek wurde zuvor schon automatisch gesetzt. Wenn wir das Testframework JUnit nutzen wollen, muss die JUnit-Bibliothek ausgewählt werden. Sie ist bereits Bestandteil von Eclipse, muss aber noch über den Befehl „Add-Library“, die Auswahl von JUnit und darunter die Auswahl von JUnit-4 aktiviert werden.

Nachdem dies geschehen ist, können wir mit „Finish“ die Projekterzeugung abschließen. Wir haben damit ein leeres Projekt (s. Abb. 2.4. Dieses können wir füllen, indem wir per Hand die nötigen Pakete und Dateien erzeugen. Alternativ können wir über „Import“ den Inhalt einer Archivdatei oder eines Verzeichnisbaums importieren. **Dabei müssen wir aber darauf achten, dass die importierte Verzeichnisstruktur mit der Projektstruktur übereinstimmt.** Wenn nicht, müssen wir entweder die Projekteinstellungen ändern, oder die Daten in die gewünschte Verzeichnisstruktur überführen. Das Endergebnis sieht

dann vielleicht wie in der Abbildung 2.5 aus.

2.1.4 Paketsichtbarkeit

Pakete haben die Aufgabe, die Gliederung großer Software zu unterstützen. Dementsprechend wird das Paketkonzept auch bei den Sichtbarkeitsregeln von Java beachtet. Bisher kennen Sie die folgenden Sichtbarkeitsbereiche:

`public`**:** Klassen und Klassenelemente sind überall sichtbar.

`private`**:** Klassenelemente sind *nur* in der aktuellen Klasse sichtbar.

***lokal*:** Die Namen von Methodenparametern und die Namen von lokalen Variablen sind nur innerhalb der betreffenden Methode sichtbar.

Durch Pakete wird mit dem Konzept der Paketsichtbarkeit ein weiterer auf das aktuelle Paket beschränkter Sichtbarkeitsbereich eingeführt. In Zusammenhang mit Vererbung wird später noch `protected` hinzukommen

Definition:

*Die **Paketsichtbarkeit** erstreckt sich auf das momentane Paket. Sie gilt dann, wenn kein explizites Sichtbarkeitsattribut (wie* `public`*,* `private` *oder* `protected`*) angegeben ist. Klassen selbst sind entweder ausdrücklich als* `public` *deklariert, oder aber sie haben implizit bei fehlender Sichtbarkeitsangabe die Paketsichtbarkeit (oft auch Defaultsichtbarkeit genannt). Die später noch zu besprechenden geschachtelten Klassen können wie die Elemente einer Klasse auch* `private` *oder* `protected` *sein.*

Im folgenden Beispiel kann man sich vorstellen, dass Teile eines Paketes nur interne Hilfsklassen darstellen:

```
/*
 * Datei: Wichtig.java
 * Diese Datei enthaelt neben der oeffentlichen
 * Klasse Wichtig die paketinterne
 * Klasse HilfeBeiDiesUndDas.
 */
package paket;

/**
 * Diese Klassen stellt allgemein verwendbare
 * wichtige Dienste bereit.
 */
public class Wichtig {
   private int geheim = 17; // private Variable
   int intern = 19;         // im Paket sichtbar
   private HilfeBeiDiesUndDas hilfe;

   public void methode() {
      ...{
   }
}

/**
 * Diese Klasse wird nur von Klassen des aktuellen
```

```
 * Paketes benoetigt.
 */
class HilfeBeiDiesUndDas {
    void methode() {
        ...
    }
}
```

Um Missverständnissen vorzubeugen, möchte ich hier allerdings nochmals betonen, dass der Begriff *Sichtbarkeit* sich immer nur auf die Verwendung von *Namen* und nicht auf Objekte bezieht. Objekte können ja in Variablen unterschiedlicher Sichtbarkeit zugänglich sein oder als Funktionsresultate zur Verfügung stehen. Es ist daher auch kein Widerspruch, wenn Methoden von Klassen, deren Namen bloße Paketsichtbarkeit haben, ihrerseits öffentlich sichtbar sind. So *besitzt* zum Beispiel die Klasse `HilfeBeiDiesUndDas`, wie jede andere Klasse auch, eine öffentliche Methode `toString`.

2.1.5 Import von Klassenelementen

Klassenfunktionen und Klassenvariablen müssen bekanntlich immer den Namen der Klasse vorangestellt haben. Viele Leute finden dies lästig. Es ist auch nicht so schön, wenn man eine mathematische Berechnung wie folgt schreiben muss:

```
double wert = Math.sqrt(Math.PI) * Math.sin(alpha);
```

schöner ist es, wenn man wie in C direkt Funktions- und Konstantennamen verwenden kann. Man erreicht dies, indem man im Programm-Kopf ein sogenanntes *Static Import* aufführt;

```
import static java.lang.Math.*;
...

double wert = sqrt(PI) * sin(alpha);
```

Neben der Wildcard-Form des Beispiels kann man auch gezielt einzelne Namen importieren. In jedem Fall muss der vollständige Paketname angegeben werden (Es gibt kein Import aus dem Defaultpaket).

static-import: `import static` *Paketname* **.** *Klassenname* **.** Elementname
| `import static` *Packetname* **.** *Klassenname* **.** `*`

2.2 Der prozedurale Aspekt von Java

In diesem Kapitel geht es nicht darum, dass man in Java prozedural programmieren *kann* – man kann das fast in jeder Sprache. Es geht darum. dass Java (leider) nicht durchgängig objektorientiert ist. Dies macht sicht an einigen Stellen, manchmal durchaus etwas verwirrend, bemerkbar:

– Die einfachen Datenelemente (numerische und boole'sche Typen) sind keine Objekte.

– Klassenfunktionen und Klassenvariable sind nicht objektorientiert.

– Java folgt auch sonst nicht der Regel „everything is an object", z.B. sind auch Methoden und Klassen.[2]

– Das statische Typsystem von Java hat an einigen Stellen Probleme mit den dynamischen Eigenschaften der Objektorientierung. Die Syntax für Typdeklarationen und die genauen Regeln des Typsystems sind inzwischen vielleicht die größten Handicaps der Sprache.

Für alle diese Punkte gibt es nachvollziehbare Gründe. Aber sicher kann man heute auch einige der frühen Entwurfsentscheidungen kritisieren. Sie lassen sich aber nicht mehr rückgängig machen, ohne dass große Teile des bestehenden Java-Codes obsolet würden.

2.2.1 Klasseneigenschaften

Eine Klasse wird in der Regel dazu verwendet, Objekte zu erzeugen. Sie *kann* aber auch einfach dazu dienen, globale Variable und Funktionen zu beschreiben.

Wie Sie wissen, muss in Java jeglicher Code in einer Klasse stehen. Das zwingt nicht im geringsten zur Objektorientierung. Es führt aber zumindest dazu, dass in der Regel die Namen von Funktionen und globalen Variablen besser lesbar werden, da sie jetzt mit dem Klassennamen verknüpft sind.

Merksatz:

Klassenelemente erkennt man daran, dass sie mit dem Vorsatz `static` *versehen sind.*

Bei der Erzeugung von Objekten wird der Konstruktor aufgerufen, der die nötige Initialisierung vornimmt. Klassen können über einen oder über mehrere Static-Blocks verfügen. Der darin enthaltene Code wird beim Laden der Klasse ausgeführt.

Static-Block : `static` { *Anweisungen* }

Fortgeschrittenes Feature: Zu jeder Klasse gehört das Klassenobjekt *Klasse*`.class`.

Wenn man eine Klasse ausschließlich für globale Funktionen usw. vorsieht, macht man das durch einige Zusätze, wie den privaten Konstruktor deutlich. Das folgende Beispiel zeigt alle hier nur kurz angesprochenen Eigenschaften.

```
import java.util.Scanner;

public final class Console {
    private Console() {}
```

[2] Allerdings gibt es für Klassen immerhin „Klassenobjekte", die eine ganze Reihe von Operationen auf Klassen bieten (*Reflection*. Schwerer wiegt, dass es (bisher) keine Objekte für Methoden und Funktionen gibt, so dass die funktionale Programmierung sehr erschwert wird.

```
    private static final Scanner in;
    static { // Static Block
        Locale.setDefault(Locale.GERMANY);
        in = new Scanner(System.in);
    }

    public static int nextInt() {
        return in.nextInt();
    }

    ...
}
```

2.2.2 Typparameter bei Behälterklassen

Bisher haben Sie zum Speichern von Daten ausschließlich Arrays verwendet, Das ist die effizienteste Methode. Sie ist aber auch nicht ohne ihre eigenen Schwierigkeiten. Der Umgang mit Arrays ist doch relativ mühsam und primitiv.

Die Java-Bibliothek enthält ein umfassendes System von Alternativen. Je nach Anwendung übernehmen unterschiedliche Klassen die Rolle von Arrays. In vielen Fällen wwerden für jeden Bedarf verschiedene Implementierungen angeboten:

- Für das Speichern von Daten in fester Reihenfolge verwendet man Listenklassen wie `ArrayList`.
- Für das Festhalten, welche Elemente zu einer Menge gehören, verwendet man Mengenklassen, wie `HashSet`.
- für Zuordnungen verwenden man Map-Klassen, wie `HashMap`.
- für nebenläufige Anwendungen gibt es besondere Warteschlangen, wie `ArrayBlockingQueue`.

Die Aufzählung lässt sich noch weiter fortsetzen. Wir werden später noch besprechen, wie man solche Behälterklassen effizient implementiert. Hier geht es zunächst um die Frage, wie das Java-Typkonzept damit umgeht.

Als Beispiel vergleichen wir ein Java-Array mit der Klasse `ArrayList`. Schauen wir uns zunächst mal die Typregeln für Arrays an. Wenn wir ein Array deklarieren und erzeugen, müssen wir angeben, welchen Typ wir für die Arrayelemente vorsehen. Der Compiler garantiert uns dann, dass wir in dem Array stets die richtigen Daten vorfinden.

```
int[] a = new int[10]
a[0] = 7;  // muss ein int sein!
int x = a[0]; // ist garantiert ein int
```

Bei Behälterklassen, ist eine ähnliche Typangabe erst seit Java 1.5 möglich, aber auch sinnvoll. Wenn wir sie „vergessen“ wird sich unser Programm (aus Gründen der Aufwärtskompatibilität) zwar noch übersetzen lassen, wir werden aber vermutlich eine Compilerwarnung erhalten.

```
ArrayList<String> b = new ArrayList<String>();
a.add("Hello"); // muss ein String sein !
a.add("World");
String w = a.get(1); // ist garantiert ein String
```

Sie erkennen, dass die Typangabe in spitzen Klammern hinter dem Klassennahmen steht.

ParametrisierterTyp: *Grundtyp<Liste von Typparametern>*

Als Elemente von Behälterklassen und auch als Typparameter dürfen nur Referenzdaten verwendet werden. Auch wenn Java nicht vollständig objektorientiert ist, so werden Objekte doch deutlich bevorzugt! Am Ende dieses Kapitels ist dargestellt, wie man Wertdaten in Objekte packen und damit auch in den Systemklassen speichern kann.

Es bleibt noch anzumerken, dass einige Klassen mehrere Parameter besitzen. assoziative Zuordnungen sind hierfür ein gutes Beispiel:

```
HashMap<String, Person> mitarbeiter =
    new HashMap<String, Person>();

mitarbeiter.put("Maier",
    new Person("Maier", "Fritz", "Programmierer", 1985));
mitarbeiter.put("Mueller",
    new Person("Mueller", "Karin", "Testerin", 1986));

int jahr = mitarbeiter.get("Mueller").geburtsJahr();
Person m = mitarbeiter.get("Maier");
System.out.println("hey " + m.vorname());
```

2.2.3 Eigene Klasse mit Typparameter

Natürlich können Sie auch Klassen mit Typparametern schreiben. Dabei muss durch die Syntax zunächst erklärt werden, welche Namen Typparameter sind. Das geschieht einfach durch Angabe des Parameters (in spitzen Klammern) unmittelbar hinter dem Klassennamen. Für den Rest der Klasse können Sie diesen Namen dann wie einen Typnamen verwenden.

Als Beispiel schreibe ich eine ganz primitive Behälterklasse, die ein Array so kapselt, dass man am Ende Daten anhängen kann. Das Array soll sich bei Bedarf vergrößern. Alle unnötigen Dinge sind weggelassen.

```
public class Array<T> { // Typparameter T
    private Object[] array = new Object[10];
    private int size = 0;

    public void add(T x) {
        if (size == array.length) {
            Object[] neu = new Object[2 * size];
            for (int i = 0; i < size; i++)
                neu[i] = array[i];
            array = neu;
        }
```

```
        array[size] = x;
        size += 1;
    }

    @SuppressWarnings("unchecked")
    public T get(int index) {
        if (index >= size) throw
            new ArrayIndexOutOfBoundsException();
        return (T) array[index];
    }
}
```

Die gute Nachricht dieses Beispiels ist, dass im Großen und Ganzen keine besonderen Regeln für die Typparameter gelten. Wohl ist anzumerken, dass der Compiler nichts über den später eingesetzten Typ weiß. Dementsprechend kann man mit Variablen von einem generischen Typ nur die Methoden der Klasse `Object` aufrufen. Dies kann man ändern, das ist aber in diesem Semester nicht unser Thema.

Aber warum habe ich dem Array `array` den Typ `Object` gegeben und nicht, wie das sicher korrekter wäre den Typparameter `T` eingesetzt? Die Antwort ist: ja, dass wäre von der Softwaretechnik her korrekt, aber das Typkonzept von Java erlaubt es nicht, ein Array mit einem Typparameter zu erzeugen.

Der Grund ist, dass Arrays zur Laufzeit erzeugt werden und dass Java verlangt, dass dann der Typ des Arrays genau bekannt ist. Die Typinformation soll nämlich in dem Array-Objekt gespeichert werden, damit bei seiner Verwendung Laufzeit-Typprüfungen möglich sind.

Mit Typparametern geht das aber nicht. Später eingesetzte konkrete Typen werden nämlich nicht in den Objekten gespeichert. Diesen Umstand nennt man *Typlöschung*. Natürlich ist das absolut inkonsequent: Bei Arrays speichert man den Typ, bei parametrisierten Objekten nicht, sondern sagt einfach, dass es beliebige Objektreferenzen sein können.

Allerdings gibt es auch eine einfache Entschuldigung für diesen Fehler. Als nämlich die Typparameter in Java 1.5 eingeführte, sah man keine andere Möglichkeit, die erlaubt, bestehende Java-Programme in neuen JVMs unverändert auszuführen. Und an den Arrays konnte man natürlich auch nichts mehr ändern.

Die Annotation `@SuppresWarnings` dient dazu eine Warnung durch den Compiler zu unterdrücken. Der Compiler ist nämlich mit Recht der Auffassung, dass die angegebene Typangabe nicht überprüfbar ist.

Merksatz:

Auch in modernen Programmiersprachen ist nicht alles logisch und gut.

Es bleibt aber festzuhalten, dass der Benutzer der Klasse `Array` davon nichts merkt. Der Compiler sorgt dafür, das keine Typfehler vorkommen. Nur innerhalb der Klasse ist dieser Schutz nicht optimal. Aber dafür ist dieser Code ja auch in einer Klasse gekapselt.

2.2.4 Elementare Wertdaten und Objektreferenzen

Bekanntlich werden in Java Zahlen und die boole'schen Werte unmittelbar in den Variablen als Werte gespeichert, wohingegen Objekte nur als Referenz in einer Variablen auftreten. Die Objekte selbst werden auf dem Heap verwaltet.

Die einfachere Behandlung der Wertdatentypen trägt zur Effizienz von Java-Programmen bei. Gleichzeitig führt sie aber dazu, dass Zahlen sich eben nicht wie Objekte verhalten. Sie gehorchen anderen Regeln als Objekte und teilen nicht die Vorteile der Objektorientierung.

Die „Kur“, die man sich in Java für dieses Problem ausgedacht hat, besteht aus zweierlei:

1. Man hat jedem Wertdatentyp eine entsprechende „Verpackung“ (wrapper class) zugeordnet: `int` zu `Integer`, `char` zu `Character`, `long` zu `Long`, `double` zu `Double` usw. in der Regel einfach eine Klasse, deren Namen mit einem Großbuchstaben beginnt.
2. Es gibt Methoden, mit denen man Wertdaten in Objekte verpacken und wieder auspacken kann.
3. Der Java-Compiler ruft ab Java 1.5 die Umwandlung bei Bedarf automatisch auf (*autoboxing*).

Das folgende Beispiel illustriert die Situation. Die in der Java-Bibliothek enthaltenen Behälterlassen, können bequem Objektreferenzen verwalten. In Objekten dieser Klassen lassen sicher aber keine elementaren Daten speichern. Es gibt aber einen Umweg. Ehe man Zahlen in Behälterobjekte steckt, müssen Sie verpackt werden. Nach dem Entnehmen müssen sie wieder ausgepackt werden.[3]

```
import java.util.ArrayList;
Scanner in = ...
...
// liest Zahlen in eine Liste ein:
ArrayList<Integer> a = new ArrayList<Integer>();
einlesen:
while (true) {
    int zahl = in.nextInt();
    if (zahl < 0) break einlesen;  // Schleife beenden
    a.add(Integer.valueOf(zahl));
}
```

Der Vorteil von `ArrayList`-Objekten gegenüber Arrays besteht u.a. darin, dass wir uns keine Gedanken über den erforderlichen Speicherbedarf machen müssen. Außerdem stehen uns so eine Reihe von Operationen zur Verfügung, die wir sonst erst mal programmieren müssten.

Dies war zunächst das Einpacken. Das Auspacken wird durch den folgenden Abschnitt dargestellt, in dem die Summe aller gespeicherten Zahlen berechnet wird[4]:

```
int summe = 0;
for (Integer x : a) {
    int zahl = x.intValue();
    summe += zahl;
}
```

Immer dann, wenn wir ein `int` in ein Objekt packen, rufen wir also `valueOf` aus; wenn wir es auspacken, verwenden wir `intValue`. Selbstredend gibt es für die anderen Datentypen ähnliche Operationen.

[3] Das Beispiel enthält die erweiterte Syntax für die Break-Anweiung. Man kann hinter `break` den Namen einer Markierung angeben. Die markierte Kontrollstruktur wird verlassen.

[4] Das Foreach „funktioniert“ auch bei Behältern.

Merksatz:

Verwenden Sie niemals die Konstruktoren der Verpackungsklassen `Integer`, `Double` *usw. Diese erzeugen nämlich immer neue Objekte. Dagegen nutzen* `Integer.valueOf` *und die gleichnamigen Funktionen der anderen Klassen, da wo es Sinn macht, einen effizienten Caching-Mechanismus. Da Verpackungsobjekte unveränderlich sind, genügt es, wenn es z.B. die 1 ein einziges mal als Objekt gibt. Autoboxing macht es automatisch richtig.*

Das als *Autoboxing* bekannte Compilerfeature erlaubt uns beim Programmieren so zu tun als seien Zahlen bereits Objekte. Das Beispiel sieht mit Autoboxing ein Stück einfacher aus. Wir sollten aber nicht vergessen, dass zur Ausführungszeit genau die gleichen Ein- und Auspackaktionen ausgeführt werden müssen wie zuvor.

```
ArrayList<Integer> a = new ArrayList<Integer>();
einlesen:
while(true) {
    int zahl = in.nextInt();
    if (zahl < 0) break einlesen;
    a.add(zahl);
}
 ...
int summe = 0;
for (int x : a) summe += x;
```

Die letzte Zeile hätten wir auch schreiben können als

```
for (Integer x : a) summe + = x;
```

2.3 Die Ausnahmebehandlung von Java

Man muss davon ausgehen, dass es in jedem größeren Programm Fehlern enhalten sind. Das Java-System unterscheidet drei verschiedene Arten von Fehlern:

1. Fatale Fehler, die auf die Hardware oder auf die virtuelle Java-Maschine zurückzuführen sind. Solche Fehler dürften eigentlich nie auftreten. Wenn sie trotzdem vorkommen, ist in der Regel eine erfolgreiche Programmausführung nicht mehr möglich.

2. Fehler, die auf äußeren Ursachen beruhen. Dazu gehören Fehler bei der Datenübertragung und die meisten Fehler, die mit dem Dateisystem zu tun haben, z.B. der Versuch eine Datei zu lesen, für die kein Leserecht vorliegt. Häufig ist es sinnvoll, sich für diese Fehler eine gute Fehlerbehandlung auszudenken.

3. Programmierfehler. Natürlich sollten solche Fehler eigentlich nicht vorkommen. Es wäre aber weltfremd zu erwarten, dass keine Fehler passieren. Viel besser ist es, mit Fehlern zu rechnen und dafür zu sorgen, dass Fehler leicht erkannt, lokalisiert und dann hoffentlich auch leicht korrigiert werden können.

Manchmal ärgert man sich etwas darüber, dass die Ausnahmebehandlung von Java einen zu besonderen Maßnahmen zwingt. Man sollte dann aber auch daran denken, dass richtige

Ausnahmebehandlung zwar Aufwand bedeutet und Zeit kostet, dass sie letztlich aber die viel größere Zeit einspart, die man bei einem undisziplinierten Programmierstil in die Fehlersuche stecken muss.

Merksatz:

Die Aufgabe der Ausnahmebehandlung besteht vor allem in der genauen Meldung *von Fehlern. Es ist nicht ihre Aufgabe „Programmabstürze“ zu vermeiden.*

2.3.1 Der Ausnahmemechanismus behebt die Unzulänglichkeiten älterer Verfahren

Wenn man ein Java-Programm einigermaßen professionell entwickelt, wird man in seinem Programmtext die nötigen Fehlerabfragen eingebettet haben. Als Beispiel will ich hier eine Funktion betrachten, die das größte Element eines Feldes finden soll. Zunächst soll die genaue Behandlung des Fehlers offen gelassen werden:

```
static int maxElement(int[] feld) {
    if (feld.length == 0) {
        // FEHLER !!!
    }
    else {
        int m = feld[0];
        for (int x : feld) {
            if (x > m) m = x;
        }
        return m;
    }
}
```

Die Funktion findet das größte Element eines Feldes von ganzen Zahlen. Der Algorithmus (Else-Teil) sollte für Sie unbedingt nachvollziehbar sein. Nur, was soll man machen, wenn die Funktion `maxElement` mit einem leeren Feld, bei dem die Länge gleich 0 ist, aufgerufen wurde? [5]

Es gibt mehrere Möglichkeiten, die aber alle ihre Probleme haben:

1. Sie geben einen **„unmöglichen“ Wert** als Funktionsresultat zurück. Nachteile: man muss aufpassen, dass man mit dem unmöglichen Wert nicht weiterrechnet; nicht immer gibt es unmögliche Werte.
2. Sie geben einen speziellen **Fehlercode** zurück. Nachteile: ein normales Funktionsresultat ist nicht möglich, man muss den Fehlercode nochmals abfragen.
3. Sie geben eine **Fehlermeldung** aus. Nachteile: die Fehlermeldung kann untergehen, es ist nicht immer erkennbar, wo der Fehler aufgetreten ist, Sie müssen unsinnige Funktionsresultate zurückgeben.
4. Sie bewirken einen **Programmabbruch** nebst Fehlermeldung. Nachteil: die Methode darf nicht von einem Programm aufgerufen werden, dass auch bei Fehlern möglichst weiterarbeiten soll (Server).
5. Sie verwenden die Methode der **Ausnahmebehandlung**. Nachteile: keine der angesprochenen Nachteile, da die Fehlerbehandlung an passender Stelle unabhängig von der Fehlererkennung vorgenommen werden kann.

[5] Eine ähnliche Frage ergibt sich, wenn die übergebene Array-Referenz gleich `null` ist.

Die Verfahren 3 und 4 werden in kleinen Programmen häufig verwendet. Da kleine Programme sehr übersichtlich sind, weiß man bei bei der Entdeckung des Fehlers genau, worum es geht, und kann daher leicht die entsprechende Ausgabe generieren. Bei Klassen und Funktionen, die innerhalb eines größeren Programms mehrfach verwendet werden (dies gilt vor allem für Bibliotheksklassen), sind diese Verfahren jedoch absolut unbrauchbar.

Die Verfahren 1 und 2 stellen den Standard der Fehlerbehandlung in der Programmiersprache C dar.[6] Ich will die Problematik von C an einem kleinen Beispiel erläutern. Da die Funktion `maxElement` bereits einen Wert zurück gibt, scheidet das Verfahren 2, nämlich die Rückgabe einer speziellen Fehlerkennung aus. Wir können aber mal dazu übergehen, die kleinste `int`-Zahl als unmöglich zu erklären und für die Kennzeichnung des Fehlerfalls zu verwenden. Wie sieht dann die Funktion und die dazu gehörende Fehlerbehandlung aus?

```
/**
 * SCHLECHTES BEISPIEL!
 *
 * Bestimmt den groessten Feldinhalt.
 * Das Feld muss mindestens die Laenge 1 haben.

 * @param feld Feld von ganzen Zahlen.
 * @return groesster Wert.
 *         Integer.MIN_VALUE, wenn das Feld 0 Elemente hat.
 */
static int maxElement(int[] feld) {
    if (feld.length == 0) {
        return Integer.MIN_VALUE; // FEHLER !!
    }
    else {
        int m = feld[0];
        for (int x : feld) {
            if (x > m) m = x;
        }
        return m;
    }
}
```

Bis auf die notwendige genaue Kommentierung, sieht diese Lösung bis hierher ganz gut aus. Wir dürfen aber auch nicht vergessen, dass wir bereits einen kleinen Preis bezahlt haben. Wir bekommen bei einem Feld, das nur Elemente gleich der kleinsten `int`-Zahl hat, eine irreführende Fehlerrückgabe.

Aber nun zur Anwendung:

```
    int[] meinFeld;
    // der Feldinhalt wird hier irgendwie ermittelt.
    int r = maxElement(meinFeld);
    if (r == Integer.MIN_VALUE) {
        System.err.println("zuwenig Werte " +
            "fuer die Bestimmung des Maximums");
        System.exit(1);
    }
    // wenn man hierhin kommt ist alles ok
    // weitere Anweisungen.
```

[6] Das gilt nur im *alten* C. C++ hat eine richtige Ausnahmebehandlung.

Eigentlich wollten wir nur wissen, wie groß der maximale Feldinhalt ist. Dies sagt uns auch der einfache Aufruf von `maxElement`. Die Regel, dass man alle Fehler behandeln sollte, führt aber dazu, dass wir an der Stelle des Aufrufs fünf weitere Zeilen für die Behandlung des Fehlerfalls schreiben mussten. gewissenhafte C-Programmierer fragen in ihren Programmen tatsächlich alle möglichen Fehler konsequent ab. Sicher gehört dazu aber auch ein großes Maß an Disziplin.

Häufig kommt noch hinzu, dass an der Stelle des Funktionsaufrufs der Fehler auch noch nicht richtig behandelt werden kann. Man braucht aber trotzdem die Fehlerabfrage und muss dann den Fehler auf eine andere Art weitermelden. Den allermeisten Programmierern ist dieser Aufwand zu hoch. Sie verzichten dann vollständig auf eine Fehlerbehandlung. Die Konsequenz zeigt sich unter anderem in der schlechten Qualität von Software.[7]

Warum ist es denn schlimm, Fehler nicht zu behandeln? Wenn ein Programm korrekt ist, wird ein fehlerhafter Aufruf schließlich nie auftreten. Das ist wahr. Wenn Sie aber doch einen Fehler in Ihrem Programm hatten – und das ist der Normalfall –, wird der Fehler nicht erkannt und es wird einfach mit einem falschen Ergebnis weitergerechnet. Möglicherweise wird dieser Fehler nicht oder sehr spät erkannt. Unter Umständen kann daraus großer Schaden entstehen. In jedem Fall kostet es dann viel Aufwand herauszufinden, worin die eigentliche Fehlerursache bestand.

Sagen Sie nicht, dass das angegebene Beispiel künstlich konstruiert ist. Ich kenne viele gute C-Programme, bei denen mehr als die Hälfte des Programmtextes aus Fehlerabfragen besteht. Und ich kenne noch mehr C-Programme, die so gut wie keine Fehlerbehandlung kennen. Oder wussten Sie, dass die Funktionen `scanf` und `printf` eine Fehlerkennung zurückgeben?

Schauen wir uns daher mal die Alternative der Ausnahmebehandlung in Java an.

Die Java-Ausnahmebehandlung geht von dem Grundsatz aus, dass die *Erkennung* und die *Behandlung* eines Fehlers getrennt zu formulieren sind. Der Grundsatz der Sicherheit verlangt, dass unbedingt alle Fehler erkannt werden. Java sieht einen grundsätzlichen Mechanismus vor, mit dem der Fehler so gemeldet ist, dass alle erforderliche Information weitergegeben wird. Wenn der Entwickler weiß, wie auf ein mögliches Fehlverhalten zu reagieren ist, kann er dann Rahmen der Fehlerbehandlung geeignete Maßnahmen ergreifen.

Zunächst die *Fehlererkennung*. Viele Fehler werden bereits von der virtuellen Maschine festgestellt, dann wird das Auslösen einer Fehlermeldung ebenfalls von der virtuellen Maschine übernommen. Im Programm wird ein Fehler meist bei der Überprüfung von Voraussetzungen für den weiteren Programmablauf erkannt. Nun geht es darum, ein Objekt zu erzeugen, dass die für eine angemessene Fehlerbehandlung notwendige Information transportiert. Dies bedeutet zunächst, dass wir für die möglichen Fehlerarten passende *Fehlerklassen* festlegen. Dazu haben wir grundsätzlich zwei verschiedene Möglichkeiten:

1. Wir können eine bereits vorhandene Ausnahmeklasse der Java-Bibliothek (oder eine bereits von uns vorher definierte Klasse) verwenden. Wenn die Bedeutung des erkannten Fehlers mit der Bedeutung der vorhandenen Fehlerklasse übereinstimmt, ist dies die beste Lösung.

2. Wenn es noch keine passende Fehlerklasse gibt, müssen wir selbst eine solche Klasse definieren. Allerdings müssen wir auch hier einen Bezug zur Java-Bibliothek

[7] Nicht vergessen, gehen auch die meisten Einbruchsmöglichkeiten in Rechnersysteme auf fehlende Fehlerprüfungen zurück!

herstellen, indem wir festlegen, in welchen Zweig der Fehlerhierarchie unser neuer Fehler einzuordnen ist.

Wenn die Funktion `maxElement` vor die Aufgabe gestellt wird, die größte von 0 Zahlen zu bestimmen, so ist diese Aufgabe nicht lösbar, da es einfach keine solche Zahl gibt. Die Java-Bibliothek verwendet in ähnlichen Fällen die Ausnahme `IllegalArgumentException`.

Diese vordefinierte Klasse `IllegalArgumentElementException` aus dem Paktet `java.lang` ist etwa wie folgt definiert:

```
public class
IllegalArgumentException extends RuntimeException {
    public IllegalArgumentException() {
        super();
    }
    public IllegalArgumentException(String message) {
        super(message);
    }
}
```

Die Bedeutung von `extends` und `super` hat mit Vererbung zu tun, dies wird unten näher erläutert. Hier nur so viel: Die Klasse `IllegalArgumentException` gehört in der Fehlerhierarchie zu dem Bereich der Klasse `RuntimeException`, diese weiter in den Bereich der Klasse `Exception` und diese (wie alle Fehlerklassen) zu dem Bereich der Klasse `Throwable`.

Die gesamte Funktionalität aller Fehlerklassen ist in der Klasse `Throwable` festgelegt. Dazu gehört, dass man optional dem Fehlerkonstruktor eine Nachricht mitgeben kann. Mit der Funktion `getMessage` kann man später an anderer Stelle dann wieder auf diese Nachricht zugreifen. Eine andere wichtige Funktion, die alle Fehlerklassen verstehen, ist `printStackTrace` zur Ausgabe der Stelle wo der Fehler aufgetreten ist.

Die neue Funktion `maxElement` sieht jetzt so aus:

```
static int maxElement(int[] feld) {
    if (feld.length == 0)
        throw new IllegalArgumentException(
            "need at least 1 value");
    int m = feld[0];
    for (int x : feld) {
        if (x > m) m = x;
    }
    return m;
}
```

Wenn jetzt `maxElement` aufgerufen wird, und ein Fehler auftritt, wird das Programm mit einer automatischen Fehlermeldung beendet, bei der die Klasse des Fehlers, der mitgegebene Text „need at least 1 value“ und die Stelle an der der Fehler erzeugt wurde (*stack trace*) ausgegeben werden. Diese automatische Ausgabe sieht etwa so aus:

```
java.lang.IllegalArgumentException("need at least 1 value");
    at Test.maxElement(Test.java:6)
    at Test.main(Test.java:25)
Exception in thread "main"
```

Sie mögen einwenden, dass diese Ausgabe nicht schön aussieht, oder nicht Endbenutzeradäquat ist. Das mag sein. Aber, Sie haben ja auch nicht gesagt, was im Fehlerfall passieren soll. Trotzdem gibt Ihnen Java immerhin eine ziemlich genaue Angabe über den aufgetretenen Fehler. Solange es sich um das Entdecken einfacher Programmierfehler handelt, ist das meiner Meinung sehr hilfreich und auch ausreichend. Jedenfalls ist es besser als das oft zu beobachtende einfache „Abfangen“ von Ausnahmen.

Merksatz:

Versuchen Sie immer, mögliche Fehlersituationen zu erkennen und dabei eine entsprechende Ausnahme zu erzeugen. Eine Fehlerbehandlung muss aber nur dann erfolgen, wenn es wichtige Gründe dafür gibt. Es macht wenig Sinn, Programmierfehler „zu behandeln“.

Wenn Sie den Programmabbruch nicht wollen, müssen Sie an geeigneter Stelle den Fehler abfangen. Dies kann irgendwo „oberhalb“ von dem Aufruf von `maxElement` geschehen. Beispielsweise könnte das entsprechende Programmfragment so aussehen:

```
public static void main(String[] argv) {
    Scanner in = new Scanner(System.in);
    try {
        // Einlesen eines Feldes und seiner Werte
        System.out.println("Wieviele Werte: ");
        int n = in.nextInt();
        int[] f = new int[n];
        for (int i = 0; i < n; i++) {
            System.out.print(i + "-ter Wert: ");
            f[i] = in.nextInt();
        }
        int m = maxElement(f);
        System.out.println("Das Maximum lautet: " +m);
    }
    catch (IllegalArgumentException e) {
        System.err.println(
            "Es wurden keine Werte eingegeben, " +
            "daher konnte kein Maximum berechnet werden.");
    }
    catch (Exception e) {
        System.err.println("fataler Fehler: " + e);
        e.printStackTrace();
    }
}
```

Das auszuführende Programmstück wird in einen Try-Block eingeschlossen, an den sich mehrere Catch-Blöcke anschließen. Diese Catch-Blöcke enthalten einen „Übergabeparameter“ vom Typ eines der möglichen Fehler. Falls das Programmstück anstandslos läuft, passiert weiter nichts. Falls jedoch ein Fehler aufgetreten ist und ein passender Catch-Block gefunden wurde, geht dort die Programmausführung weiter. Unbekannte Fehler werden an eine eventuell höhere Ebene weitergereicht. Nach der Fehlerbehandlung, die ja auch eine neue Benutzereingabe veranlassen kann, wird das Programm normal fortgesetzt. [8]

Auf den ersten Blick erscheint Ihnen die Notwendigkeit für den Try-Block nicht ganz einleuchtend. Geht es denn nicht nur darum, zu sagen, was im Fall eines bestimmten Fehlers

[8] Wenn Sie der Auffassung sind, dass es besser wäre, vor dem Aufruf sicherzustellen, dass das Feld wenigstens ein Element hat, so stimme ich Ihnen zu, dass dies der bessere Programmierstil ist. Mir ist jetzt hier nur kein besseres Beispiel eingefallen.

passieren soll? Im Prinzip schon, nur gehört zur Definition eines bestimmten Fehlers nicht nur die Angabe über die Art des Fehlers sondern auch die die Angabe, wo innerhalb des Programmablaufs der Fehler auftreten konnte. Genau diese Zuordnung wird durch try und catch ausgedrückt: „Fange hier den Fehler xyz auf, der in diesen Programmanweisungen auftreten kann."

2.3.2 Einzelheiten der Fehlerbehandlung in Java

Die folgende Syntaxdarstellung zeigt alle Varianten zum Erzeugen, Weiterreichen und Abfangen von Ausnahmen. Es enthält zusätzlich zu den bisher besprochenen Formen, den Finally-Block.

Throw-Anweisung : `throw` *Konstruktoraufruf*

Try-Catch-Block : `try` *Block*
`(` `catch` `(` *Parameter* `)` *Block* `)*`
`(` `finally` *Block* `)?`

Methodenkopf : *Typ Name*(*Parameterliste*)
`(` `throws` *Liste von Ausnahmeklassen* `)?`

Die meisten Ausnahmen, mit denen Sie bisher zu tun hatten, wurden von dem Java-System erzeugt. In einigen Fällen durch die Java-Bibiliothek, in anderen durch die virtuelle Maschine selbst. Bestimmt ist Ihnen schon eine `NullPointerException` begegnet. Diese Ausnahme wird immer dann erzeugt, wenn Sie über eine Null-Referenz eine Methode aufrufen wollen. Eine andere wichtige Ausnahme ist `ArrayIndexOutOfBoundsException`. Sie wird erzeugt, wenn versucht wird, auf einen nicht vorhandenen Arrayplatz zuzugreifen. Viele andere Ausnahmen werden durch die Methoden der Java-Bibliothek erzeugt. In der Bibliothek wird natürlich die angegebene Syntax der Throw-Anweisung genutzt. Sie sollten es sich ebenfalls zur Gewohnheit machen, auch in Ihren eigenen Programmen Fehlerzustände abzufragen und mit `throw` zu melden.

Abhließend soll der gesamte Ablauf durch die Abb. 2.6 dargestellt werden. Dabei wird auch noch auf den Finally-Block eingegangen.

Die Abb. 2.6 entspricht dem folgenden fiktiven Programmcode (hier uninteressante Anweisungen sind weggelassen):

```
void m1() {
    ...
    try {
        ...
        m2();
        ...
    }
    catch (XYException e) {
        ...
    }
    finally {
        ...
```

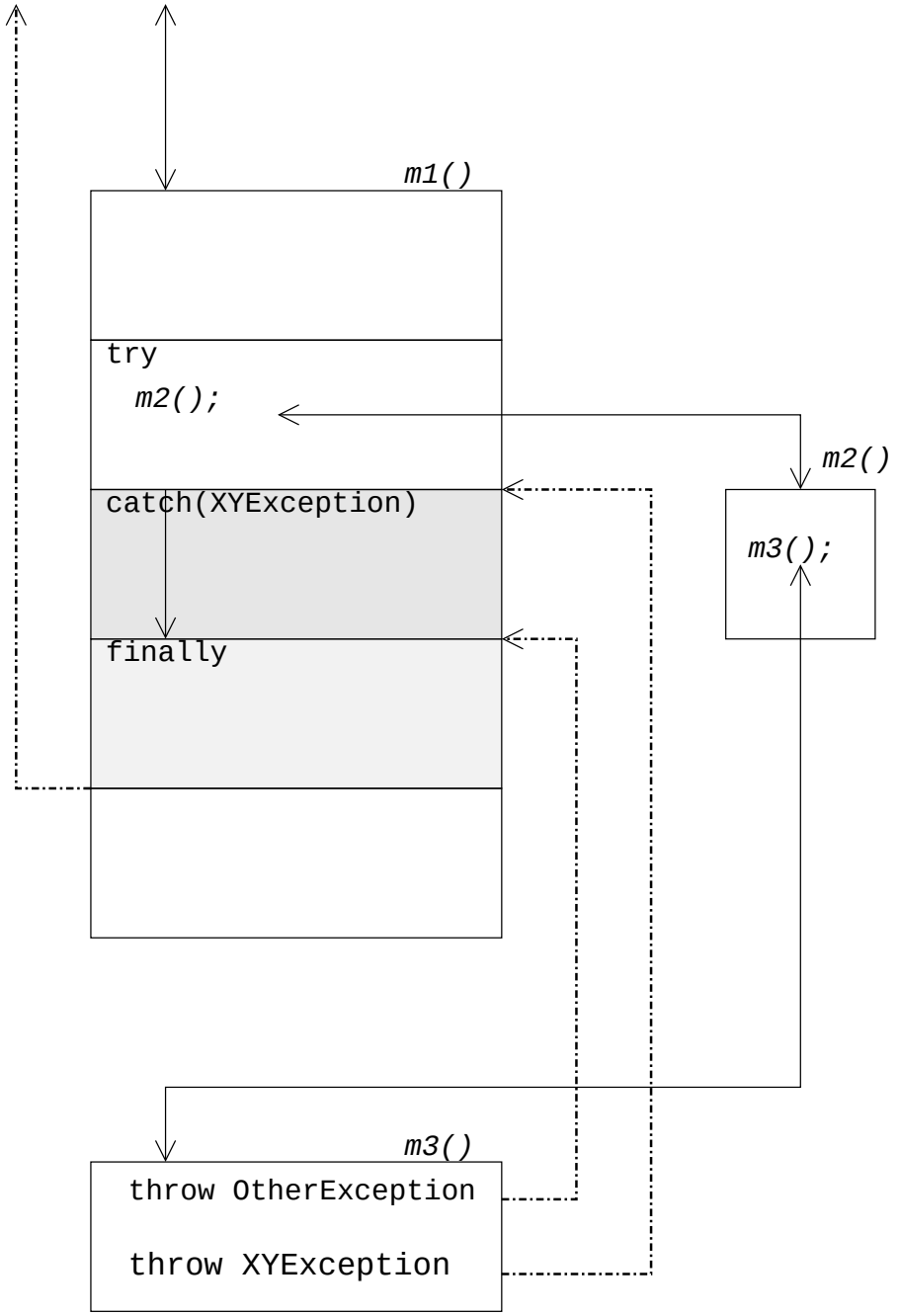

Abbildung 2.6: Darstellung des Programmflusses im Normalfall als durchgezogene Linie. Ein Methodenaufruf ist durch eine Linie mit Doppelpfeil dargestellt. Der nach unten zeigende Pfeil steht für den Aufruf, das obere Ende für den Rücksprung. Im Fehlerfall wird der normale Ablauf verlassen; es findet nur noch ein Rücksprung länges der gestrichelten Linie dar. Es sind zu unterscheiden: Programmabschnitte ohne besondere Vorkehrungen, Try-Blöcke, Catch-Blöcke und Finally-Blöcke. Wird der Finally-Block für einen nicht abgefangenen Fehler ausgeführt, wird nach seiner Ausführung der Ablauf des Fehlerzustandes fortgesetzt.

```
    }
    ...
}

void m2() {
    ...
    m3();
    ...
}

void m3() {
    ...
    if (fehler1) throw new OtherException();
    ...
    if (fehler2) throw new XYException();
    ...
}
```

An dem Beispiel können Sie sehen, dass Methoden, die kein Auffangen von Fehlern vorsehen, einfach übersprungen werden. Ein Finally-Block kann auf zwei Arten verlassen werden. Liegt kein Fehler vor, wird die Programmausführung normal fortgesetzt; liegt jedoch ein Fehler vor, so wird weiter in den aufrufenden Methoden nach einer Fehlerbehandlung gesucht.

Für den Finally-Block haben wir die drei folgenden Fälle:

- Das Programm läuft „glatt“, ohne Ausnahme durch. In diesem Fall wird der Try-Block vollständig abgearbeitet. Danach wird der Finally-Block ausgeführt. Anschließend werden die restlichen Anweisungen der Methode ausgeführt.
- Es tritt die behandelte Ausnahme „Ausnahme“ auf. Ab der Stelle wo die Ausnahme auftritt wird der Try-Block verlassen. Anschließend wird der Catch-Block für „Ausnahme“ ausgeführt. Schließlich wird der Finally-Block mit den „Aufräumarbeiten“ ausgeführt. Danach wird mit den auf den Finally-Block folgenden Anweisungen fortgefahren.
- Es tritt eine hier nicht behandelte Ausnahme auf. Auch hier wird der Try-Block sofort verlassen. Anschließend wird auch hier der Finally-Block ausgeführt. Der Fehlerzustand bleibt weiter bestehen. Die aktuelle Methode wird verlassen und in der aufrufenden Methode wird nach einer Fehlerbehandlung gesucht.

Der Finally-Block enthält also Anweisungen, die immer ausgeführt werden müssen. Eine sinnvolle Anwendung kann zum Beispiel darin bestehen, geöffnete Dateien zu schließen oder nicht mehr benötigte temporäre Dateien zu löschen.

Definition:

Der Finally-Block enthält Anweisungen, die unabhängig von dem Auftreten von Ausnahmen in einem Try-Block immer ausgeführt werden müssen. Der Finally-Block kann auch ohne Catch-Block direkt auf einen Try-Block folgen.

2.3.3 Die Java-Fehlerhierarchie

Es wurde bereits angesprochen, dass Java verschiedene Ausnahmearten kennt. Die möglichen Ausnahmen eines Java-Programms sind in einer Hierarchie von Klassen ange-

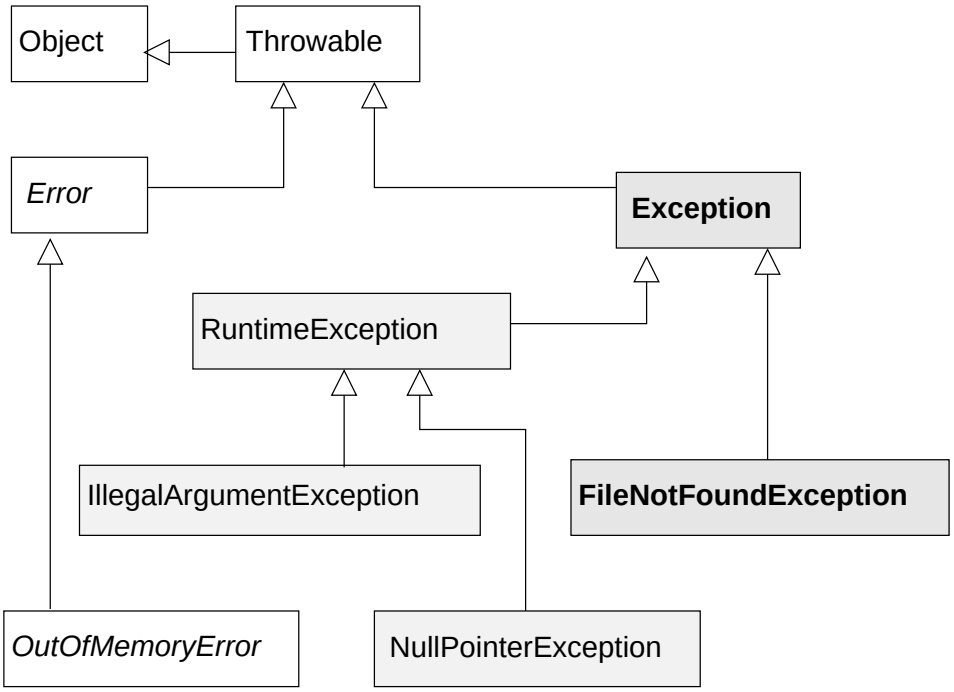

Abbildung 2.7: Hierarchie der Java-Fehlerklassen. Die hellgrau unterlegten Klassen bezeichnen ungeprüfte Ausnahmen. Geprüfte Ausnahmen sind fett gedruckt auf dunkelgrauem Grund. Fehlerklassen (`Error`) sind schräg gestellt.

ordnet (vgl. Abb. 2.7), deren Wurzel die Klasse `Throwable` ist. Unmittelbare Unterklassen von Throwable sind die Klassen `Exception` und `Error`. Die Klasse `RuntimeException` ist wiederum eine Unterklasse von `Exception`.

Hinsichtlich der Behandlung von Exceptions geht Java von den unterschiedlichen möglichen Fehlerursachen aus. Dies schlägt sich dann sowohl in der Einordnung innerhalb der Exception-Hierarchie als auch innerhalb eines Java-Programms nieder:

- Unterklassen der Klasse `Error` werden im Allgemeinen für fatale Systemfehler benutzt, die man nicht sinnvoll behandeln kann. Ein typisches Beispiel, ist `VirtualMachineError`, der dann auftritt, wenn der Java-Interpreter fehlerhaft ist.

- Unterklassen der Klasse `RuntimeException` sind werden als Ausnahmen mit programminterner Ursache angesehen. In aller Regel sind dies dann Programmierfehler. Da man der Fehlerart nicht immer die Ursache ansehen kann, können solche Fehler aber auch auf andere Ursachen, wie z.B. falsche Benutzereingabe, zurückzuführen sein. Je nach dem Einzelfall muss man den Fehler im Programm beachten oder nicht.

- Unterklassen von `Exception` – mit Ausnahme von `RuntimeException` – gelten als *checked Exceptions* oder *geprüfte Ausnahmen*. Java verlangt zwingend, dass jede Methode, die eine solche Ausnahme werfen kann, in ihrem Methodenkopf (als Teil der Methodensignatur) den Namen der Ausnahme in einer Throws-Klausel aufführt. Der Name „checked exception“ ergibt sich daraus, dass der Compiler *überprüft* und nachvollzieht wo solche Ausnahmen entstehen können. Der Sinn der oft lästigen Compilerprüfung besteht darin, dass solche Ausnahmen im Allgemeinen

nie ausgeschlossen werden können (z.B. ein Fehler bei einer Datenübertragung). Die erzwungene Angabe im Methodenkopf soll die Robustheit eines Programms erhöhen.

Beachten Sie die Syntax der Throws-Klausel in der weiter oben stehenden Syntaxübersicht. Dabei wird dem Methodenkopf einfach das Schlüsselwort `throws` gefolgt von einer Liste der eventuell geworfenen checked-Exceptions angehängt. Wie auch sonst üblich, können generelle Namen, wie `Exception`, für alle untergeordneten Klassen stehen.

Der Grund für eine geprüfte Ausnahme sollte immer in der Methodendokumentation angegeben sein (`@throws`-Zeile). Auch bei ungeprüften Ausnahmen ist dies sinnvoll, wenn durch den Kommentar Information über mögliche Fehler beim *Aufruf* der Methode ausgedrückt wird. Es macht keinen Sinn, zu dokumentieren, was alles im Fall eines Fehlers bei der internen *Implementierung* der Methode passieren könnte.

Das folgende abschließende Beispiel zeigt eine Methode, die den Inhalt einer Textdatei in eine Liste einliest. Die dabei auftretende geprüfte Ausnahme ist `FileNotFoundException`. `main` übernimmt von der Kommandozeile einen Dateinamen, liest die Datei ein und gibt den Inhalt auf dem Bildschirm aus.

```
import java.io.FileNotFoundException; // Eingabeklassen
import java.io.FileReader;
import java.util.ArrayList; // speichert die Daten
import java.util.List;      // Schnittstelle fuer Listen
import java.util.Scanner;   // Texteingabe

/**
 * Die Klasse FilePrinter ist eine Anwendung, die den
 * Inhalt einer Datei auf dem Bildschirm ausgibt.
 */
public class FilePrinter {
    /**
     * Liest den Inhalt einer Textdatei Zeile fuer
     * Zeile in eine Liste ein.
     *
     * @param name Dateiname.
     * @return Liste mit den Textzeilen der Datei.
     * @throws FileNotFoundException wenn die Datei
     *         nicht geoeffnet werden kann.
     */
    public static List<String> readFile(String name)
        throws FileNotFoundException
    {
        Scanner file = new Scanner(new FileReader(name));
        List<String> lines = new ArrayList<String>();
        try {
            while (hasNextLine(file))
                lines.add(file.nextLine());
        }
        finally {
            // egal was passiert, muss eine geoeffnete
            // Datei wieder geschlossen werden.
            if (file != null) file.close();
        }
        return lines;
    }

    /**
     * Erfragt den Namen einer Datei und gibt ihren
     * Inhalt nebst Zeilennummern auf der Konsole aus.
```

```
     */
    public static void main(String[] argv) {
        Scanner in = new Scanner(System.in);
        while (true) {
            System.out.print("Dateiname: ");
            String name = in.nextLine();
            try {
                List<String> contents = readFile(name);
                String format =
                    " %" +
                    numDigits(contents.size()) +
                    "d  %s%n";
                int lineNr = 1;
                for (String line : contents)
                    System.out.printf(format, lineNr++,
                        line);
                break;
            }
            catch(FileNotFoundException e) {
                System.out.printf(
                    "Datei %s nicht gefunden. ", name);
                System.out.println("..erneut eingeben!");
            }
        }
    }

    /**
     * Berechnet die Anzahl der Ziffern einer positiven
     * ganzen Zahl.
     * @param n ganze Zahl
     * @return Anzahl der Ziffern von n
     * @throws ArithmeticException wenn n negativ ist.
     */
     public static numDigits(int n) {
         if (n < 0)
             throw new ArithmeticException(
                 n + " is not a positive number");
         return n == 0 ? 1 : (int) Math.log10(n) + 1;
     }
}
```

Beachten Sie auch, dass die in `numDigits` geworfene Ausnahme nicht aufgefangen wird. In einem korrekten Programm kann sie nicht auftreten. Beim Entwickeln des Beispiels bekam ich aufgrund eines Fehlers eine `NullPointerException`. Natürlich wurde diese auch nirgendwo „behandelt“.

Das notwendige Auffangen oder Weiterreichen von checked Exceptions ist manchmal lästig, so dass Anfänger sehr oft dazu neigen, den Compiler mit einem ganz einfachen Schema „ruhig“ zu stellen, indem sie einfach die Ausnahme auffangen und stillschweigend ignorieren. *Unterdrücken Sie Fehlermeldungen nicht!* Das kostet Sie viel Zeit, da Sie sich damit der Möglichkeit berauben, zu erfahren wo Sie einen Fehler gemacht haben.

Mir erzählen jedes Jahr einige Studenten im Praktikum, dass sie lange nach einem Fehler gesucht und ihn nicht gefunden haben. Oft konnte ich dann den Fehler in wenigen Minuten lokalisieren, indem ich einfach die Catch-Blöcke im Programm der Studenten entfernt habe.

```
    // schlechter Stil!
    try {
        ...
```

```
}
catch (Exception noliTurbareCirculosMeos) {
  System.out.println("'Es ist ein Fehler passiert."');
}
```

Wenn Sie nicht wissen, was Sie mit der Ausnahme anfangen sollen, ist es (vor allem wenn die Klasse Bestandteil einer Bibliothek werden soll) am besten, die Ausnahme mit `throws` weiterzureichen. Wenn Sie sie aber auffangen, weil Sie wissen, dass es sich um einen schwerwiegenden Fehler handeln muss, dann sollten Sie zumindest das Programm abbrechen *und* die komplette Fehlerinformation ausgeben, wie das in dem Beispiel durch die Methode `reportFatalError` geschieht.

Eine andere sinnvolle Variante kann sein, die geprüfte Ausnahme in ein Error-Objekt oder eine ungeprüften Ausnahme einzupacken und erneut zu werfen. Dabei bleibt die Fehlerinformation erhalten und kann auch später noch abgefragt werden.

```
// brauchbare Methodik fuer fatale Fehler
try {
    // die hier auftretenden Fehler duerften nie
    // vorkommen
    ...
}
catch (Exception fatalerFehler) {
    throw new Error(fatalerFehler);
}
```

Das erneute Werfen eines Fehlers ist auch immer dann sinnvoll, wenn die ursprüngliche Fehlermeldung für Programmebenen an die der Fehler gemeldet wird, keine sinnvolle Bedeutung hat. Dies ist z.B. dann der Fall, wenn die ursprünglich geworfene Ausnahme nur mit Implementierungsdetails einer internen Softwareschicht zu tun hat. Dann sollte der Fehler abgefangen und mit einer sinnvollen Ausnahmeklasse erneut gemeldet werden.

```
// Verbergen interner Fehler
try {
    ...
}
catch (LowLevelException internerFehler) {
    throw new HighLevelException("neue Erklaerung");
}
```

2.4 Grundregeln für den Entwurf einer Klasse

Die Diskussion der Ausnahmebehandlung zeigt, dass die Programmiersprache Java nicht nur von Sprachdefinitionen im engeren Sinn bestimmt ist, sondern, dass sie darüber hinaus eine Vielzahl von Konventionen über den Gebrauch von bestimmten Mechanismen umfasst. Einige der wichtigsten Regeln betreffen die Art und Weise wie man eine Klasse definieren sein sollte.

2.4.1 Die Klasse `java.lang.Object`

Die Klasse `Object` ist direkt oder indirekt die Oberklasse aller Klassen. Die genauere Erklärung einer Oberklasse erfolgt im nächsten Kapitel. Hier genügt die Konsequenz dieser

Eigenschaft: *Die Klasse* `Object` *definiert Operationen und Methoden für alle Objekte*

Operationen die auf der Identität eines Objekts beruhen

Sie kennen die Regel, dass man Strings mit `equals` und nicht mit == vergleichen soll. Die Erklärung ist, dass == zwei Referenzen vergleicht, aber nicht aussagt, ob es sich um die gleichen Strings handelt. `equals` vergleicht dagegen bei Strings die Zeichenfolgen und darauf kommt es in der Regel an.

Die Definition von `equals` bestimmt was bei der jeweiligen Klasse unter *Objektidentität* zu verstehen ist. Dabei spielt es eine ganz wesentliche Rolle, ob es sich um veränderliche oder um unveränderliche Objekte handelt. Wir können hier schon mal festhalten, dass bei veränderlichen Objekten in aller Regel, die bereits in der Klasse `Object` vordefinierte Methode, die (genau wie == auf „Pointergleichheit“ prüft), korrekt ist. Bei unveränderlichen Objekten sollte dagegen `equals` durch eine eigene Methode überschreiben werden.

Um das richtig zu erläutern, will ich etwas weiter ausholen. In Java müssen wir unterscheiden zwischen der *logischen Bedeutung* eines Objekts und seiner *Implementierung* im Programm.

Definition:

> *Zwei Objekte sind* ***identisch****, wenn sie in allen ihren Eigenschaften übereinstimmen (*principium identitatis indiscernibilium*).*[9]

In der Informatik muss der Identitätsbegriff etwas weiter gefasst werden um zweitrangige Unterschiede in der internen Darstellung von Objekten auszuschalten. So gelten die Zahlen `1.5` und `1.5f` nicht nur mathematisch, sondern auch in einem Computerprogramm als identisch, auch wenn sie intern unterschiedlich gespeichert sind. Identität ist nicht leicht zu definieren, wenn auch zufällige Unterschiede infolge von Rechenungenauigkeiten berücksichtigt werden müssen.

In der Anwendung versteht man unter Identität immer die logische Identität und schaltet auf bloßen Implementierungsdetails beruhende Unterschiede aus. Das heißt, dass die Objektidentität nicht einfach alleine durch das Java-System geprüft werden kann. Gleichheit ist ein Konzept, das grundsätzlich für alle Objekte definiert ist, dessen Details aber von Objektklasse zu Objektklasse verschieden zu präzisieren sind.

Dies geschieht, indem eventuell die in der Klasse `Object` definierte Methode

```
public boolean equals(Object vergleichsObjekt)
```

passend überschrieben wird

In der Abbildung 2.8 sind zwei unterschiedliche Gleichheitsdefinitionen dargestellt. Bei *unveränderlichen* Wertobjekten (z.B. der Klasse `Bruch`) ist die Gleichheit über die Werte der Attribute zu definieren. Bei *veränderlichen* Objekten ist die Objektidentität fast immer durch die eindeutige Objektreferenz bestimmt, da ja ansonsten die Gleichheit zweier Objekte keine verlässliche Eigenschaft wäre. Zwei veränderliche Instanzen können sich dagegen in der Zukunft unterschiedlich verändern und sind also nicht identisch. Dieser

[9] Diese Definition ist unmittelbar einleuchtend. Dementsprechend lässt sich auch kein Autor angeben. Sie war bereits in der Antike bekannt. G.W. Leibniz hat sie zur Grundlage umfassender philosophischer Betrachtungen gemacht.

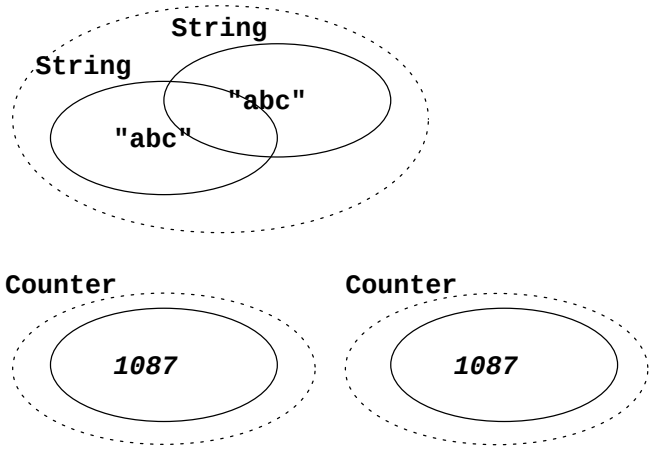

Abbildung 2.8: Gepunktete Linien umranden logische Objekte, durchgezogene Ellipsen stehen für Javaobjekte. Die beiden String-Objekte stellen eigentlich nur einen einzigen String dar. Es ist mehr oder weniger zufällig, ob Java einen oder zwei getrennte Speicherbereiche verwendet. Die beiden Zählerobjekte haben momentan den gleichen Zählerstand. Sie sind aber trotzdem verschiedene Objekte.

„Normalfall“ der Objektorientierung ist durch die Klasse `Object` als der Default vorgegeben.

Die von Zahlen her bekannte Gleichheitsbeziehung == sollte nicht mit dem objektorientierten Identitätsvergleich verwechselt werden. Die Operation == prüft die Übereinstimmung von Objektreferenzen. Es ist eine effiziente, aber auch eine auf sehr niedrigem Abstraktionsniveau angesiedelte Operation.[10].

Bei der Klasse `String` kann man sich die Unterschiede von `equals` und == klar machen:

```
String a = "abcdef";
String b = "abcdef";
String c = "a" + a.substring(1,6); // c = "abcdef"

boolean true1 = a.equals(b);
boolean true2 =  a == b;   // macht der Compiler so

boolean true3 = a.equals(c);
boolean false1 = a == c;  // kann der Compiler nicht
```

Die Namen der boole'schen Variablen stehen für die Ergebnisse der Vergleichsoperationen. Die Ergebnisse von `equals` entsprechen den Erwartungen; die Ergebnisse von == sagen nur etwas über den Compiler und über die virtuelle Maschine aus.[11]

Ein Sonderfall stellt die Prüfung dar, ob ein Ausdruck wirklich eine Objektreferenz liefert oder ober er den Wert `null` hat. Hier kann man `equals` nicht aufrufen. Es bleibt also nur der direkte Vergleich der Referenzen. Die folgenden Zeilen zeigen die Varianten des Vergleichs mit `null`.

[10] Die Java-Notation ist irreführend. Die natürlich aussehende Operation == sollte bei Objekten nur mit großer Vorsicht verwendet werden! Die Sprache Scala hat das anders gemacht. Dort ruft man mit == die Methode `equals` auf. Für den einfachen Vergleich zweier Referenzen gibt es den speziellen Operator `eq`

[11] Manchmal, wenn man *genau* weiß, was man tut, bevorzugt man == aus Optimierungsgründen.

```
variable = new XYZObject();
null == variable      // ergibt false
null.equals(variable) // Fehler: NullPointerException
variable == null      // ergibt false
variable.equals(null) // ergibt false
```

Das letzte Ergebnis ist eine Anforderung an die Definition von `equals`: wenn das Argument `null` ist, muss `false` zurückgegeben werden.[12]

Die vollständige Auflistung, der bei der Definition einer Klasse immer zu berücksichtigenden Methoden, ist:

1. `boolean equals(Object obj)`
   Diese Operation dient dem Vergleich zweier Objekte.Sie liefert `true`, wenn die beiden Objekte als gleich gelten (Identität). Das Objekt, d.h. seine Klasse legt fest, wie der Vergleich durchzuführen es. Es *kann* sich dabei um den Vergleich der Objektinhalte handeln. Dies ist sinnvoll bei Objekten die unveränderliche Werte darstellen (z.B. `String` oder `Integer`). Es kann aber auch sein, dass `equals` exakt den Vergleich von == ausführt. Dies ist bei veränderlichen Objekten angebracht. Für Klassen, die die Methode `equals` nicht überschreiben, ist durch die Klasse `Object` bereits der Vergleich der Referenzen vorgegeben.

2. `int hashCode()`
   Diese Operation ordnet jedem Objekt eine ganze Zahl zu. Für Objekte, die gemäß `equals` gleich sind, muss `hashCode` denselben Wert liefern. Ansonsten sollten die Werte möglichst breit gestreut sein, damit sie charakteristisch für das jeweilige Objekt sind. Die Methode der Klasse `Object` gibt vermutlich die Speicheradresse des Objekts zurück. Die Methode der Klasse `String` gibt eine Zahl zurück, die sich aus den Unicode-Werten der Zeichen des Strings ergibt. Jede Klasse, die ein eigenes `equals` besitzt, muss auch `hashCode` überschreiben. Die Bedeutung von `hashCode` liegt in der erheblichen Beschleunigung von Suchverfahren.

3. `String toString()`
   Diese Operation ordnet jedem Objekt einen möglichst eindeutigen aussagekräftigen String zu. Für zwei Objekte, die gemäß `equals` gleich sind, sollte derselbe String zurückgegeben werden. Die Methode der Klasse `Object` gibt einen String zurück, der sich aus dem Namen der Klasse und der Adresse des Objekts zusammensetzt. Die Methode der Klasse `String` gibt den String selbst zurück.

Die drei angesprochenen Methoden bilden eine Einheit. Bei den im folgenden Abschnitt besprochenen Klassen von unveränderlichen Wertobjekten sollten stets alle drei Methoden überschrieben werden. Bei anderen Klassen ist das in der Regel nicht nötig.

Die Feststellung des Klassenobjekts

Die Methode `Class getClass()` gibt das Klassenobjekt der Klasse eines Objekts zurück. Ein Klassenobjekt ist eine Instanz der Klasse `Class`. Es ermöglicht im Programm die Eigenschaften einer Klasse abzufragen. Zum Beispiel erhält man für jedes in der Variable `abc` referierte Objekt durch den Aufruf `abc.getClass().getName()` den Namen der Klasse. Die weiteren Möglichkeiten werden hier nicht besprochen. Sie

[12] Das unsymmetrische Verhalten von `equals` ist Ursache für viele Programmierfehler. Dies geht darauf zurück, dass die `null`-Referenz ein sehr unglückliches Konzept ist.

gehören zu den fortgeschrittenen Java-Elementen, die mit dem Begriff *reflection* bezeichnet werden.

Methoden zur Thread-Synchronisation

Java ist eine der Programmiersprachen, die auf Sprachebene das Konzept der Nebenläufigkeit unterstützt. Die Klasse `Object` definiert für jedes Objekt die Methoden `wait`, `notify` und `notifyAll`.

Sonstige Methoden

Die Methode `clone` kann genutzt werden, um Objektkopien zu erzeugen. Die Methode `finalize` wird aufgerufen, wenn ein Objekt von der Speicherbereinigung (garbage collection) entfernt wird. Sie kann dann eventuell nötige „Aufräumarbeiten“ erledigen.

2.4.2 Wertobjekte

Die am einfachsten zu gebrauchenden Objekte sind Objekte, die ihren Wert nie ändern. Ihr Verhalten ähnelt insofern dem der einfachen Werttypen von Java. Als vollwertige Objekte sind sie aber flexibler zu verwenden.

Die wichtigsten Eigenschaften sind:

- Eine Wertobjekt ist ein *unveränderliches Objekt*, d.h. es gibt keine Methode, mit der der Wert eines Objektes verändert werden kann.
- *Verknüpfungsoperationen* auf Wertobjekten liefern *neue* Objekte.
- Bei Wertobjekten sind die Methoden `equals`, `hashCode` und `toString` so überschrieben, dass ihr Ergebnis nur von dem gespeicherten Wert und nicht von der Adresse des Objekts abhängt.
- Es macht bei Wertobjekten keinen Unterschied, ob ein bestimmter Wert einmalig in einem einzigen Objekt gespeichert ist, oder ob es mehrere Objekte mit demselben Wert gibt.
- In Anwendungen mit Nebenläufigkeit und in verteilten Systemen bilden Wertobjekte Grundbausteine auf die verschiedene Threads gleichzeitig zugreifen können, oder die sicher über Nachrichten ausgetauscht werden können.

Bei Klassen für Wertobjekte sollte durch `final` verhindert werden, dass weitere Klassen abgeleitet werden können. Andernfalls kann man nämlich nicht mehr für die Unveränderbarkeit garantieren.

Am Beispiel der Klasse `Bruch` soll die Definition einer Klasse für Wertobjekte gezeigt werden.[13].

[13] Natürlich ist auch die Klasse `java.lang.String` ein sehr gutes Beispiel.

Beispielhafte Klasse `Bruch`

Um das Beispiel nicht zu lang werden zu lassen, soll `Bruch` der folgenden Schnittstelle genügen:

```
    Bruch(int zaehler, int nenner);  // Konstruktor
    Bruch div(Bruch bruch);          // Division
    int zaehler();                   // Zaehler
    int nenner();                    // Nenner
    double doubleValue();            // Gleitkommadarstellung
    boolean equals();                // Identitaet
    int hashCode();                  // Hashzahl
    String toString();               // String
```

Der Konstruktor und die Methode `div` sollen eine `ArithmeticException` werfen, wenn das Resultat kein gültiger Bruch ist (Division durch 0).

Die Spezifikation wird durch einen Test dokumentiert, der für das Testframework JUnit geschrieben wird. Hier wird die Beschreibung des Tests zunächst ausgelassen. Sie wird dann im nächsten Abschnitt nachgeholt.

Beim Entwickeln der Lösung sollten wir Schritt für Schritt vorgehen. Im ersten Schritt reicht es, wenn man den Compiler „glücklich“ macht. Dann kann man sich einen Testfall nach dem andern vornehmen, bis alle Testfälle erfüllt sind. Abschließend sollte man vergleichen, ob die Implementierung und der Test vollständig sind, und ob man mit mit der Struktur der Klasse zufrieden ist.

Hier gebe ich als Beispiel ein mögliches Endergebnis an.

```
/**
 * Die Klasse Bruch beschreibt Bruchobjekte.
 * Alle Methoden garantieren, dass ein Bruch niemals den
 * Nenner 0 hat.
 * Diese Implementierung speichert Brueche intern immer in
 * ihrer gekuerzten Form.
 */
public final class Bruch {
    private final int zaehler;
    private final int nenner;

    /**
     * Erzeugt einen Bruch aus Zaehler und Nenner.
     * Der Nenner darf nicht 0 sein.
     *
     * @param zaehler Zaehler des Bruchs.
     * @param nenner Nenner des Bruchs.
     * @throws ArithmeticException wenn der Nenner 0 ist.
     */
    public Bruch(int zaehler, int nenner) {
        if (nenner == 0)
            throw new ArithmeticException(
                "Der Nenner darf nicht 0 sein!");
        this.zaehler = gekuerzterZahler(zaehler, nenner);
        this.nenner = gekuerzterNenner(zaehler, nenner);
    }

    /**
     * Gibt eine neuen Bruch zurueck, der gleich dem
     * Quotienten von diesem Bruch und dem Argument ist.
     *
```

```
     * @param dividend durch diesen Bruch wird dividiert.
     * @return Ergebnis der Division
     * @throws ArithmeticException wenn der Dividend 0 ist.
     */
    public Bruch div(Bruch dividend) {
        return new Bruch(
            this.zaehler * dividend.nenner,
            this.nenner * dividend.zaehler);
    }

    /**
     * Gibt den Bruch als Dezimalzahl zurueck.
     * @return Dezimalwert
     */
    public double doubleValue() {
        return (double) zaehler / (double) nenner;
    }

    /**
     * Gibt den Bruch in gekuerzter Form als
     * String zurueck.
     * DIES IST HIER KEINE GUTE LOESUNG !!
     */
    public String toString() {
        return zaehler + "/" + nenner;
    }

    /* siehe java.lang.Object. */
    public boolean equals(Object that) {
        if (that instanceof Bruch) {
            Bruch b = (Bruch) that;
            return this.zaehler == b.zaehler &&
                   this.nenner == b.nenner;
        }
        return false;
    }

    /* siehe java.lang.Object. */
    public int hashCode() {
        return zaehler + nenner << 7;
    }

    /**
     * Gibt den Zaehler des gekuerzten Bruches
     * zurueck.
     * Der Nenner ist beim Aufruf !=  0.
     * Am Ende ist der Nenner immer > 0.
     */
    private int gekuerzterZaehler(int z, int n) {
        ...
        return ...;
    }

    /**
     * Gibt den Nenner des gekuerzten Bruches
     * zurueck.
     * Der Nenner ist beim Aufruf !=  0.
     * Am Ende ist der Nenner immer > 0.
     */
    private int gekuerzterNenner(int z, int n) {
        ...
        return ...;
    }
```

```
}
```

Die Art und Weise wie in der Klasse gekürzt wird, ist vom Ablauf her nicht ganz optimal. Ich habe diese Form gewählte um die funktionale Bedeutung deutlich zu machen und um ganz einfach durch `final` auszudrücken, dass die Instanzvariablen nicht verändert werden können. Das ist zwar für unveränderliche Objekte nicht zwingend nötig, dient aber der Lesbarkeit.

Eine vollständige Klasse `Bruch` sollte natürlich weitere Methoden enthalten. Zusätzlich sollte auch dafür gesorgt werden, dass die Größe zweier Brüche verglichen werden kann. Wie dies sinnvoll geschieht, wird im nächsten Kapitel erläutert.

Sonstige Klassen

Andere Klassen haben weniger strenge Regeln, als dies für Wertobjekte gilt. Man sollte sich aber strikt an die Regel halten, die Veränderung des Objektzustandes und die Abfrage des aktuellen Zustandes zu trennen. Das heißt, Methoden die den Zustand eines Objekts verändern, haben keine Rückgabe und Methoden, die den Zustand erfragen, sollten nichts verändern. Nach Möglichkeit sollte eine Klasse so „gestrickt“ sein, dass grundsätzlich jederzeit jede Methode aufgerufen werden kann. Wenn bestimmte Aufrufe nicht möglich sind, sollte eine Ausnahme erzeugt werden (z.B. `IllegalStateException` oder `NoSuchElementExcption`).

2.5 Die Test-Zuerst Methode

Bei der Entwicklung von Klassen stellen sich zwei miteinander verbundene Fragen:

1. Wie können wir unmissverständlich spezifizieren, was eine Klasse und ihre Methoden tun sollen?
2. Wie können wir überprüfen, ob eine Klasse die Spezifikation erfüllt.

Wenn wir ganz genau sind, ist die Antwort auf beide Fragen sehr schwierig – im Grunde genommen ist damit ein in der Praxis bisher nicht gelöstes Problem angesprochen. In Softwaretechnik werden Sie sicher mehr über dieses Thema lernen.

Hier wollen wir nur ein paar bescheidene Schritte in diese Richtung gehen. Zunächst wollen wir uns hier nicht mit der Softwareentwicklung im Großen, sondern nur mit der Entwicklung kleiner Programmeinheiten, nämlich Klassen, befassen.

Zum anderen haben wir nicht den Anspruch, dass die entwickelte Klasse mit *Sicherheit* korrekt ist. Trotzdem soll aber mit realistischer Wahrscheinlichkeit erreicht werden, dass die fertige Klasse unseren Anforderungen entspricht.

Diese Ziele erreichen wir mit der „Test-Zuerst Methode“ (*test first*). Wir gehen so vor, dass wir für jede Klasse eine eigene Testklasse entwickeln. Da es sich hierbei um das Testen einer kleinen Programmeinheit (unit) handelt, spricht man auch von Unit-Test[14]

[14] Genau genommen ist die Test-Zuerst Methode kein Test, sondern eine Entwicklungsmethode.

- Zunächst entwickeln wir für die neue Klasse einen Test. Darin erzeugen wir Objekte, rufen Methoden auf und formulieren unsere Erwartung an die Resultate der Methoden.
- Als nächstes entwickeln wir unsere Klasse. Im ersten Schritt schreiben wir nur soviel auf, dass die Klasse durch den Compiler übersetzt werden kann.
- Wenn wir den Test durch JUnit ausführen, erhalten wir vermutlich eine Reihe von negativen Testergebnissen (es ist wichtig zu sehen, dass der Test zunächst scheitert).
- Durch Ergänzen der neuen Klasse sorgen wir nach und nach dafür, dass alle Tests erfüllt werden.
- Wenn wir die Klasse erweitern wollen, erweitern wir zunächst den Test und dann die Klasse.
- Wenn wir feststellen, dass in der Klasse Fehler versteckt sind, die der Test nicht erkennt, verbessern wir zunächst den Test so, dass der Fehler erkannt wird. Anschließend verbessern wir die Klasse.
- Nach jeder Veränderung der Klasse führen wir den automatischen Test aus.

Hier ist leider nicht der Platz, die Softwareentwicklung mit der Test-Zuerst Methode in allen Details zu beschreiben. Es soll aber wenigstens angegeben werden, wie die Testspezifikation in JUnit3 und in JUnit4 aussieht.

Grundsätzlich wird in beiden Fällen eine Testdatei erstellt. In der Sprache von JUnit heißt diese auch *TestFall* oder *test case*. Diese Datei enthält eine Anzahl von *Testmethoden*. Die Testmethoden testen die einzelnen Funktionalitäten der Klasse. Es kommt darauf an, die Testmethoden möglichst einfach zu halten und jeweils möglichst nur eine Grundfunktion zu testen. Nur so erhält man durch den Test auch aussagekräftige Ergebnisse, die schnell zum Auffinden des Fehlers führen.

Die Methode soll am Beispiel einer Klasse `Bruch` erläutert werden. Wir kennen zwar bereits die zu testende Klasse. Normalerweise hätten wir aber erst den folgenden Test geschrieben, um uns über die Aufgaben der Klasse klar zu werden.

2.5.1 Testspezifikation in JUnit4

Da das Testframework JUnit schon geraume Zeit existiert, ist es kein Wunder, dass inzwischen auch Veränderungen erfolgt sind. Eclipse bietet zwei Versionen zur Auswahl, das ältere JUnit3 und das neuere JUnit4. Da JUnit4 insgesamt etwas einfacher zu verstehen ist, verwende ich nur noch diese Version.

Ein grundsätzliches Merkmal von JUnit4 ist, dass Testmethoden durch Annotationen gekennzeichnet sind. Annotationen erkennt man an dem vorangestellten `@`. Eine Annotation ist ein Element von Java, das Zusatzinformationen zu einer Klasse und zu Teilen der Klasse liefern kann. Diese Information ist dann vom Compiler oder sogar (bei JUnit zur Ausführungszeit auswertbar. Von sich aus haben Annotationen aber keine Wirkung. Hier verwenden wir sie nur als Teil eines „Kochrezepts“.

Das folgende Beispiel zeigt eine Klasse zum Test unserer Klasse `Bruch`.

```
import static org.junit.Assert.*;
```

```
import org.junit.Before;
import org.junit.Test;

class BruchTest {
    private Bruch bruch;

    @Before
    public void initialisiereBruch() {
        bruch = new Bruch(-4, -6);
    }

    @Test
    public void konstruktor() {
        assertEquals(2, bruch.zaehler());
        assertEquals(3, bruch.nenner());
    }

    @Test
    public void testEquals() {
        Bruch bruch2 = new Bruch(2, 3);
        assertTrue(bruch.equals(bruch2));
        assertTrue(bruch2.equals(bruch));
        assertFalse(bruch.equals(null));
        assertFalse(bruch.equals("2/3"));
        assertFalse(bruch.equals(new Bruch(-2, 3)));
    }

    @Test(timeout=1000)
    public void divisionsTest() {
        Bruch dividend  = new Bruch(1, 3);
        Bruch quotient = new Bruch(2, 1);
        assertEquals(quotient, bruch.div(dividend));
    }

    @Test(expected=ArithmeticException.class)
    public void illegalerKonstruktor() {
        new Bruch(2, 0);
    }

    @Test
    public void testToString() {
        assertEquals("2/3", bruch.toString());
        assertEquals("2", new Bruch(2, 1).toString());
    }

    @Test
    public void testDoubleValue() {
        assertEquals(2./3., bruch.doubleValue(), 1e-12);
    }
}
```

Hier sind eine Reihe von Anmerkungen angebracht:

- Die Klasse wird eingeleitet von Importanweisungen: ein statisches Import für die Klasse `Assert` (ermöglicht den bequemen Aufruf der Methoden des Frameworks), und Imports für die beiden hier verwendeten Annotationen `Before` und `Test`.
- Ein Test besteht aus einer Reihe von Testläufen. In jedem Testlauf wird zunächst die durch `@Before` gekennzeichnete Methode (hier: `initialisiereBruch`)

zum Aufsetzen der Testumgebung und dann eine der Testmethoden ausgeführt. Dies wird (im Normalfall) solange fortgeführt, bis alle Testmethoden einmal ausgeführt worden sind. Beim Auftreten eines Fehlers oder einer Ausnahme, wird der jeweilige Testlauf beendet, die anderen Tests werden aber trotzdem ausgeführt. Mit der separaten Initialisierung erreicht man, dass die einzelnen Tests unabhängig voneinander ausgeführt werden. Bei Auftreten eines Fehlers in der Intialisierung kann allerdings keine Testmethode ausgeführt werden!

- Die Testklasse kann eigene Instanzvariable erhalten. Wichtig ist, dass diese nicht einen Konstruktor (auch nicht implizit), sondern nur in der Initialisierungsmethode (`@Before`) initialisiert werden.
- Die Testmethoden erkennt man an der Annotation `@Test`.
- In einer Testmethode werden gegebenenfalls noch weitere Testvorbereitungen getroffen. Dann wird eine oder mehrere der gemeinsam zu testenden Methoden aufgerufen. Schließlich wird das Ergebnis mit einer der `assert..`-Methoden überprüft.
- Die Angabe `timeout=` gibt die maximale Laufzeit des Test in Millisekunden vor. Bei Überschreiten der Zeit wird der Test als gescheitert abgebrochen. Damit erreicht man, dass auch bei Endlosschleifen in einzelnen Methoden der Gesamttest durchgeführt werden kann.
- Die Angabe `expected=` benennt eine Ausnahmeklasse. Damit ist es möglich festzustellen, ob eine Methode bei fehlerhaftem Aufruf die erwarteten Ausnahme wirft. Wird die angegebene Ausnahme nicht geworfen, ist der Test gescheitert.
- Es gibt noch weitere Annotationen, deren Bedeutung Sie den `JUnit`-Hilfen entnehmen können. Viele dieser Annotationen werden nur selten verwendet. Häufiger sieht man vielleicht die Annotation `@Ignore`. Damit werden Testfälle für noch nicht fertigstellte Methoden außer Kraft gesetzt.[15]
- Falls nötig, können Sie auch Hilfsmethoden schreiben, die nicht als Bestandteil des Tests gekennzeichnet sind. Diese werden dann nicht automatisch, sondern nur bei direktem Aufruf ausgeführt.

Eine Testklasse hat nicht mehr Rechte als jede andere Klasse. Sie kann also auch nicht auf private Klassenelemente zugreifen. Es lässt sich nur das Verhalten eines Objekts, nicht aber die exakte Implementierung testen!

Die Methode `konstruktor` verlangt, dass Zähler und Nenner der Variablen `bruch` die richtigen gekürzten Werte haben. Zur Prüfung wird die Methode `assertEquals` aufgerufen. Sie enthält als erstes Argument den erwarteten Wert und als zweites Argument die Abfrage des Objekt gelieferten Wert. Stimmen beide Werte überein, ist der Test erfolgreich. Andernfalls wird die Testmethode abgebrochen und das Ergebnis wird in JUnit festgehalten. Um das Testresultat lesbarer zu gestalten, können die Testmethoden optional einen erläuternden Text enthalten.

Auch die Methode `divisionsTest` verwendet `assertEquals`. Es ist aber zu beachten, dass hier nicht elementare Zahlen sondern komplette Bruchobjekte verglichen werden. Dabei wird die Methode `equals` der Klasse `Bruch` aufgerufen. Der Test ist nur dann sinnvoll, wenn sichergestellt ist, dass `equals` korrekt arbeitet.

[15] Da der Compiler nichts von den Test-Annotationen weiß, werden die zu ignorierenden Methoden trotzdem übersetzt.

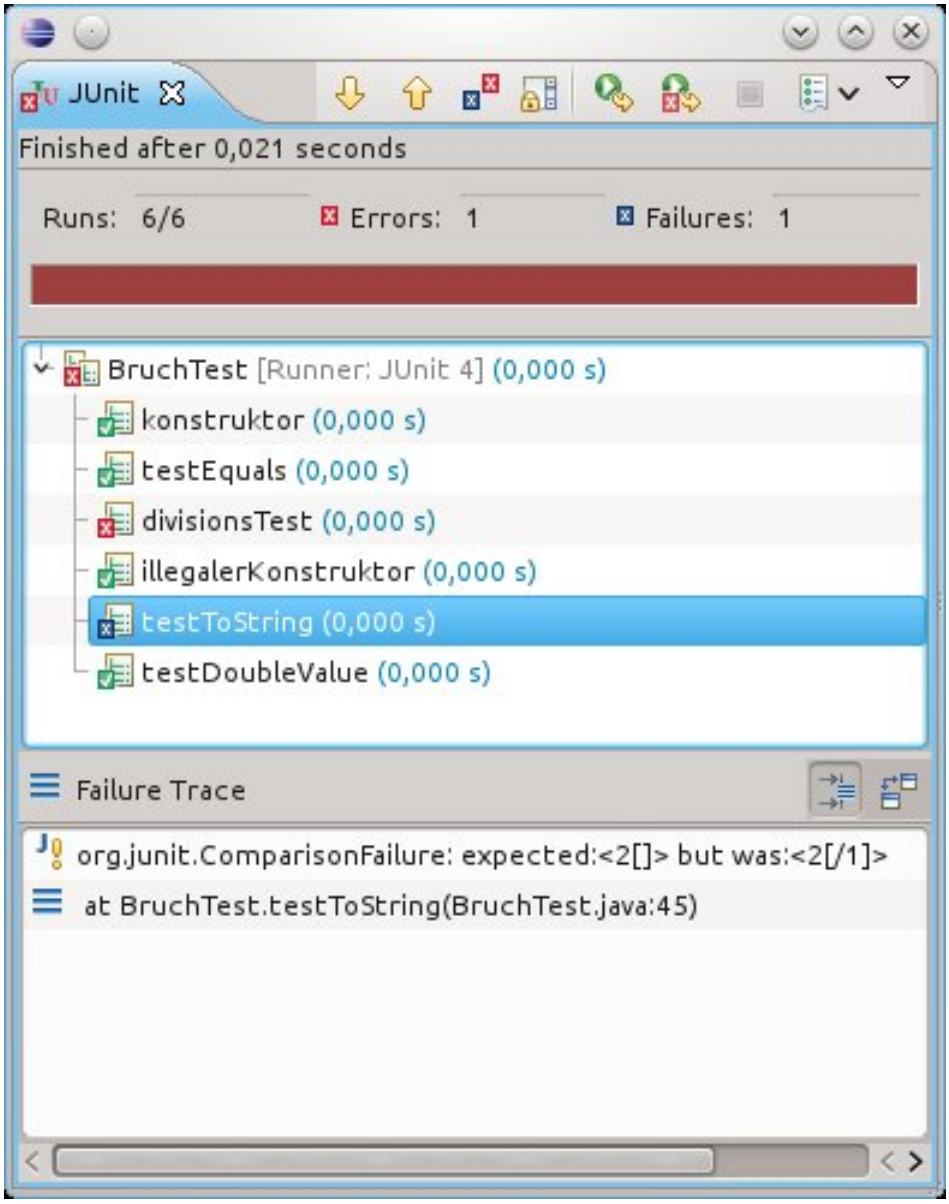

Abbildung 2.9: Dieser Test der Klasse `Bruch` war in zwei Punkten nicht erfolgreich. Beim Test der Division ist eine Ausnahme aufgetreten (nicht detailliert dargestellt). Der Test der Methode `toString` hat für die ganze Zahl 2 nicht das erwartete Ergebnis gebracht.

Gleichzeitig ist bei `divisionsTest` ein Timeout von 1 s angegeben (auch wenn das hier vielleicht nicht so sinnvoll ist).

Die Methode `illegalerKonstruktor` demonstriert den Umgang mit erwarteten Ausnahmen.

In `testDoubleValue` enthält der Aufruf `assertEquals` einen weiteren Parameter. Hier muss man nämlich bei Gleitkommazahlen eine Genauigkeitsschranke für die Überprüfung der Gleichheit festlegen. `1e-12` bedeutet, dass das Ergebnis in mindestens 12 Stellen mit der Erwartung übereinstimmen soll.

Schließlich ist in Abbildung 2.9 dargestellt, wie ein negatives Testergebnis aussehen könnte. Obwohl in einem der Tests eine Ausnahme aufgetreten ist (wenn man den Punkt auswählt, bekommt man nähere Information), wurden weitere Tests durchgeführt. Die meisten Tests sind erfolgreich. Für den Test von `toString` erhalten wir eine Fehlermeldung. Die detaillierte Meldung beschreibt das nicht der Erwartung entsprechende Ergebnis und gibt einen Verweis auf die Zeile der Testmethode. Durch Mausklick kommt man in Eclipse bequem zu dieser Stelle und kann dann gegebenenfalls einen Haltepunkt für den Debugger setzen.[16]

2.6 Fehlersuche mit dem Debugger

Seit es Programmiersprachen und Compiler gibt, gibt es auch Debugger. Seltsamerweise werden Debugger, von denjenigen, die sie am meisten benötigen, am wenigsten genutzt. Das wollen wir ändern.

Zunächst eine Worterklärung und eine kleine Anekdote. Das englische Wort *bug* bedeutet „Käfer“. Die Arbeit eines *debuggers* kann man insofern mit einem Ungezieferentferner vergleichen. Laut einer uralten Legende hat der Begriff einen weit zurückliegenden Ursprung. Man hatte nämlich in einem Computerprogramm mal wieder einen Fehler, der sich partout nicht finden ließ. Schließlich, und das war damals auch nicht so außergewöhnlich, verdächtigte man die Hardware. Damit hatte man dann auch Erfolg. Die Computerpionierin Murray Grace Hopper entdeckte nämlich zwischen den Schaltkontakten eines Relais eine tote Motte. Der „Bug“ war entdeckt und nachdem er entfernt war (der Rechner war jetzt „entbugged“), lief das Programm wieder anstandslos.

Beim modernen Debugging geht es nicht um die Hardware, sondern um die Software. Es gibt aber Parallelen. Das Innenleben eines geraden ausgeführten Programms ist für uns genauso in einer Blackbox versteht wie die Hardware unseres Computers. Es mag zwar sein, dass man in manchen Fällen die Fehlerursache allein an der fehlerhaften Funktionalität erkennen kann, aber ein professionelles Vorgehen nutzt alle technischen Möglichkeiten um einem Fehler auf den Sprung zu kommen. Die Fehlersuche beginnt zunächst mit einem Testprogramm, welches häufig den Fehler schon in einen engen Bereich eingrenzen kann. Umso kleiner dieser Bereich ist umso einfacher ist die Fehlersuche.

Ehe ich im folgenden auf ein paar Einzelheiten eingehe, will ich betonen, dass Debugging aber auch noch weitere Vorteile bietet:

- Wie angesprochen, erlaubt Debugging das Aufspüren von Fehlern.
- Die Visualisierung des Programmablaufs durch einen Debugger kann das Verständnis einzelner Programmabschnitte und Algorithmen verbessern.

[16] In diesem Skript wird Debugging nicht besprochen. Ich gebe aber in der Vorlesung einige Hinweise.

– Ein Debugger kann auch dazu beitragen die Grundmechanismen der objektorientierten Programmierung besser zu verstehen.

Gemäß meiner Vorbemerkung dürfte klar sein, dass Debugging nicht auf Java beschränkt ist. Praktisch jede Programmiersprache bietet diese Möglichkeit. Debugging erfordert auch nicht zwingend eine integrierte Entwicklungsumgebung. Kommandozeilendebugger bieten in der Regel die identische (manchmal) sogar bessere Funktionalität, allerdings in der Regel bei schlechterer Bedienbarkeit. Für C können Sie z.B. den GNU-Debugger gdb und für Java den Debugger jdb benutzen.

2.6.1 Das Beispielprogramm

Fehlersuche kommt eigentlich nur in größeren Programmen vor. Hier im Skript kann ich aber nur kleine Beispiele zeigen. Mein Testszenario enthält also nur zwei Klassen. Eine JUnit-Testklasse, `StackTest` die definiert, was meine Klasse können soll und die Klasse `Stack`, die den Stack implementiert. Um das Beispiel nicht zu lang werden zu lassen, habe ich auch alle Kommentare weggelaasen.

```
import static org.junit.Assert.*;
import java.util.NoSuchElementException;
import org.junit.*;

public class StackTest {
    private Stack<Integer> intStk;
    private Stack<String> strStk;

    @Before
    public void create() {
        strStk = new Stack<String>();
        intStk = new Stack<Integer>();
        for (int i = 1; i <= 7; i++)
            intStk.push(i);
    }
    @Test
    public void init() {
        assertTrue(strStk.isEmpty());
        assertFalse(intStk.isEmpty());
    }
    @Test
    public void identity() {
        strStk.push("abc").push("def");
        assertSame("def", strStk.pop());
        assertFalse(strStk.isEmpty());
    }
    @Test
    public void intStack() {
        for (int i = 7; i >= 1; i--)
            assertEquals(Integer.valueOf(i), intStk.pop());
        assertTrue(intStk.isEmpty());
    }
    @Test
    public void testToString() {
        assertEquals("[1,2,3,4,5,6,7]", intStk.toString());
        assertEquals("[]", strStk.toString());
    }
    @Test(expected=NoSuchElementException.class)
    public void emptyPop() {
        strStk.pop();
```

```
    }
}
```

Man kann erkennen, dass die eigentliche Stackklasse parametrisiert sein soll, so dass man Stacks mit unterschiedlichen Datentypen erzeugen kann. Gleichzeitig demonstriert das Beispiel auch, dass das oben angesprochene Autoboxing die Illusion erzeugt, das nicht objektorientierte `int` und das objektorientierte `Integer` seien dasselbe. In der Methode `intStack` ließ sich dieser Unterschied aber nicht verbergen. Da es `assertEquals` sowohl für Objekte wie für elementare Datentypen gibt, musste ich mich entscheiden. Die gewählte Variante macht aus der Zahl `i` ein Objekt und lässt dieses mit dem Ergebnis von `pop` vergleichen. Die andere Variante, hätte das Resultat in ein `int` verwandelt:

```
assertEquals(i, intStack.pop().intValue());
```

Das Ergebnis ist in beiden Fällen gleich.

Als nächstes schauen wir uns noch kurz die Stackklasse an. Diese Variante enthält noch einen Fehler, den wir weiter unten finden werden.

```
// ENTHAELT EINEN FEHLER !!!
import java.util.NoSuchElementException;

public class Stack<T> {
    private int top;
    private T[] data = newArray(3);

    public Stack<T> push(T x) {
        checkStorage();
        data[top] = x;
        top += 1;
        return this;
    }
    public T pop() {
        if (isEmpty())
            throw new NoSuchElementException();
        top -= 1;
        T r = data[top];
        data[top] = null;
        return r;
    }
    public boolean isEmpty() {
        return top == 0;
    }
    @Override
    public String toString() {
        StringBuilder b = new StringBuilder("[");
        if (!isEmpty()) b.append(data[0]);
        for (int i = 1; i < top; i++)
            b.append(",").append(data[i]);
        return b.append("]").toString();
    }
    private void checkStorage() {
        if (top == data.length) {
            T[] newData = newArray(2 * data.length);
            for (int i = 0; i < data.length; i++)
                    newData[i] = data[i];
        }
    }
```

```
    @SuppressWarnings("unchecked")
    private T[] newArray(int size) {
        return (T[]) new Object[size];
    }
}
```

Im Großen und Ganzen enthält die Klasse wenig Besonderes. In der Methode `newArray` ist die bereits angesprochene Schwierigkeit beim Erzeugen von Arrays in generischen Klasse gekapselt. Die Methode `checkStorage` sorgt dafür, dass der Stack immer genug Speicher hat.

Die wichtigste neue Methode ist `toString`. Sie ersetzt die in der Klasse `Object` bereits vorgegebene Methode durch einen für unseren Fall besseren Mechanismus. Dies ist besonders beim Testen im Debugger hilfreich, da so unsere Stack-Objekte auch im Debugger immer sinnvoll dargestellt werden.

Anmerkung:

Wissen Sie wass ein `StringBuilder` ist (s. 1. Semester)? Da String-Objekte unveränderlich sind, enthält die Bibliothek für die wenigen Fälle, wo man per Programm Strings aufbaut diese Klasse. Die Verwendung von `StringBuilder` ist erheblich effizienter als die Verknüpfung von String-Objekten mittels `+`.

2.6.2 Schrittweises Ausführen eines Programms

Zu dem Debugging eines Programms gehören (in Eclipse und ähnlichen IDE's) mehrere Dinge.

1. Zunächst muss dafür gesorgt sein, dass der Compiler die für das Debugging erforderliche Information bereitstellt. Das ist insbesondere die Information über Klassen und Klassenelemente (die ist in Java immer vorhanden) und die Information über lokale Variablen und die Zuordnung von Zeilennummer und Binärcode. Ist diese Information vorhanden, kann der Debugger so tun, als würde der Quelltext unmittelbar ausgeführt. Unter *Preferences-Java-Compiler* kann man sich davon überzeugen, ob die richtigen Einstellungen vorgenommen wurden.

2. Man kann große Programme nicht Schritt für Schritt ausführen (dafür ist der Computer da). Es ist daher nötig festzulegen, an welcher Stelle die „normale" Ausführung in den „Debug-Modus" übergehen soll. Dies geschieht durch die Festlegung von Haltepunkten (*breakpoint*). Haltepunkte können in Eclipse über verschiedene Wege gesetzt werden, Zum Beispiel über das Kontextmenü am linken (grauen) Rand des Editorfensters, oder einfach durch Doppelklick in diesen Bereich.

3. Anstelle das Programm „normal" auszuführen, muss es im Debug-Modus gestartet werden. Diese geschieht über das Kontext-Menü des Package Explorer („debug as"), über das Run-Menü oder durch Drücken des Käfer-Symbols in der Hauptmenüleiste.

4. Vor dem ersten Anhalten schlägt Eclipse vor, in die Debug-Perspektive zu wechseln. Was man auch unbedingt tun sollte. Evtl. empfiehlt es sich diese Perspektive für die eigenen Zwecke nochmals etwas anzupassen und dann zu speichern.

5. Zur Debug-Perspektive gehören mehrere wichtige Views, die im Folgenden noch besprochen werden.

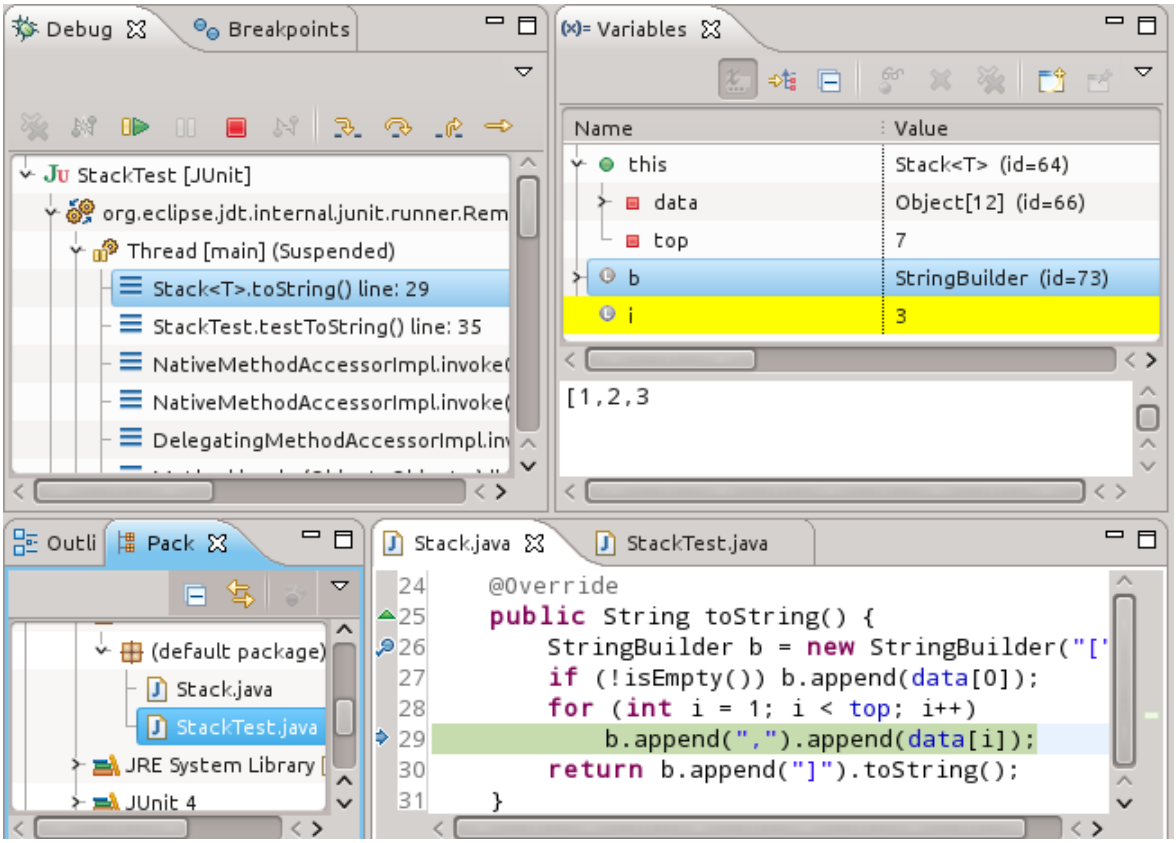

Abbildung 2.10: Darstellung einer Debug-Perspective.

6. Der weitere Debug-Ablauf ist durch die Menüleiste des Debug-View zu steuern.

7. Wichtig: Wenn das Programm nicht vollständig ausgeführt wird, sollte es beendet werden (roter Knopf im Debug-Fenster). Andernfalls, wird Ihr Rechner nach kurzer Zeit nicht mehr reagieren!

Die Abbildung 2.10 stellt eine Ansicht des Debug-Fensters dar. Vermutlich sieht das auf Ihrem Rechner erzeugte Fenster anders aus. Ich habe wegen beschränktem Platz nur die wichtigsten Views angezeigt.

In der linken oberen Ecke sehen Sie die „Debug-View“. Der Fensterrahmen enthält mehrere Bedienelemente für die Steuerung des Ablaufs. Sie erkennen:

Resume: weitere Ausführung bis zum nächsten Haltepunkt

Suspend (grau): Unterbrechen der Ausführung

Terminate: Beendet den Debugging-Lauf

Step Into: Ausführung der nächsten Zeile; wenn dort ein Methodenaufruf steht, wird in die Methode gesprungen.

Step Over: Ausführung der nächsten Zeile

Step Return: Führe die aktuelle Methode bis zum Ende aus.

Set Next Statement: Hier kann man festlegen, dass im nächsten Schritt zurück zu einer bestimmten Methode im Aufruf-Stack gesprungen wird.

Drop-to-Frame: Dieser Schalter ist hier nicht sichtbar. Er bewirkt, dass die Ausführung zum Beginn der aktuellen Methode zurückspringt. Lokale Variable werden entfernt. Aber Vorsicht: Änderungen an externen Objekte bleiben erhalten (Seiteneffekt).

Unterhalb befindet sich ein Fenster zur Darstellung des Stackframes, genauer zur Angabe, wo sich der Ablauf des Programms gerade befindet. Nachdem zunächst das Programm und der aktuelle Thread dargestellt sind [17] Anschließend sind die Methodenaufrufe dargestellt. Zuoberst findet sich die gerade ausgeführte Methode, nebst Zeilennummer. Darunter die Methode, von der sie aufgerufen wurde usw.

In der verkleinerten Darstellung, sind ein paar Dinge nicht dargestellt. Hier verweise ich Sie auf die Hilfe und auf die Vorlesung.

Das rechte obere Fenster „Variables“ stellt die Inhalte der verfügbaren Variablen dar. Instanzvariable werden sichtbar, wenn man die This-Referenz aufklappt. Die jeweils ausgewählte Variable wird im unteren Teil des Fensters kompakt ausgegeben. Die Ausgabe erfolgt dabei über die Methode `toString`. Wenn Sie eigene Objekte lesbar dargestellt haben wollen, müssen Sie diese Methode passend überschreiben. Sie können auch die Variablen der Methoden sehen, von denen aus die aktuellen Methode aufgerufen wurde. Dazu müssen Sie halt nur die entsprechende Methode im Stackframe auswählen.

In der unteren Hälfte der Abbildung sind die bekannten Editor-Fenster angezeigt. Im Editor ist die aktuelle Zeile (vor ihrer Ausführung) grün markiert. Sie können nachvollziehen, dass bis zu dieser Stelle (bei dem Wert `i = 3`) drei Array-Werte nebst Trennzeichen in dem `StringBuilder` eingefügt wurden.

2.6.3 Einfaches Szenario zur Fehlersuche

Wenn Sie die oben angegebene Testdatei als JUnit-Test ausführen, erhalten Sie für alle Testfälle einen roten Punkt. Für alle Fälle enthalten wir (Abb. 2.11) die identische Fehlermeldung, dass in der Methode `push` eine Ausnahme geworfen wurde. Es ist übrigens kein Wunder, dass überall die gleiche Fehlermeldung kommt. Wie man erkennt, resultiert sie nämlich aus der Ausführung der Methode `create`, die vor jedem Test ausgeführt wurde (siehe `@Before`). Natürlich sagt uns der Test bereit, wo der Fehler ungefähr liegen könnte. Aber schauen wir zunächst mit dem Debugger genauer nach.

Wir starten JUnit erneut im Debug-Modus. Da wir keinen Haltepunkt gesetzt haben, erhalten wir wieder dieselbe Statistik, ohne dass der Debugger anhält. Wir müssen erst einen Haltepunkt setzen. Dies können wir z.B. in der Methode `push` tun, in der der Fehler aufgetaucht ist. Da wir aber nur an dem Auftreten der Exception interessiert sind, ist es noch besser nur für diesen Fall ein Anhalten vorzusehen. Die Ansicht „Breakpoints“ enthält einen Schalter mit der sich Exception-Haltepunkte definieren lassen. Ich habe (s. Abbildung 2.12) einen Haltepunkt für eine beliebige Ausnahme (`Exception` und alle Unterklassen) definiert. Wichtig ist, dass der Haltepunkt auch greift, wenn die Ausnahme bereits im Programm aufgefangen wird (*caught*).

[17] Sie wissen, dass ein Programm mehrere Abläufe (Threads) enthält, auch wenn Sie nur einen einzigen programmiert haben? Ihr einziger Programmthread trägt immer den Namen `[main]`.

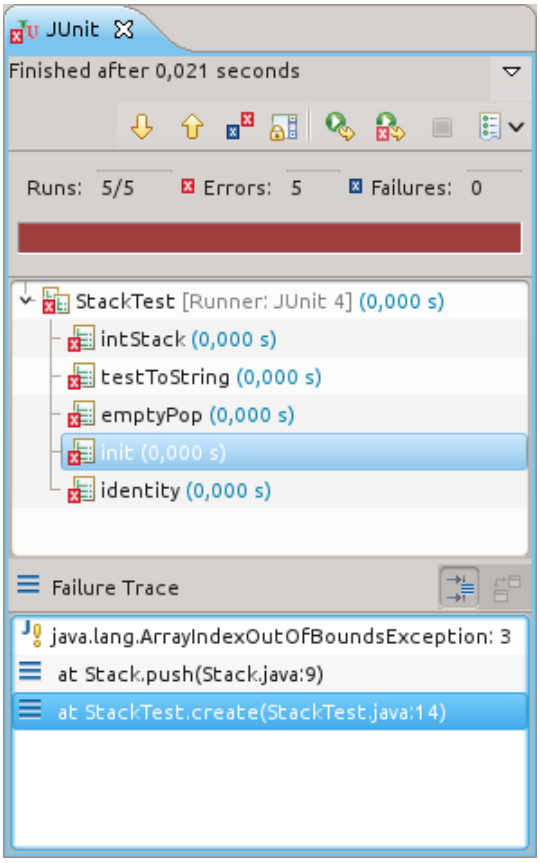

Abbildung 2.11: JUnit erkennt eine Ausnahme.

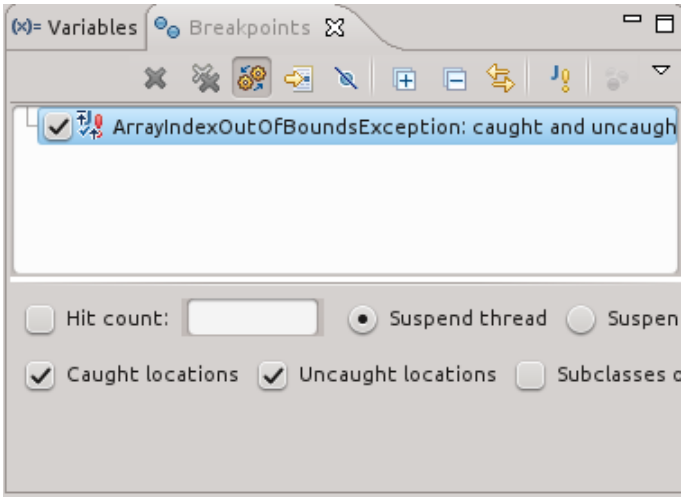

Abbildung 2.12: Eigenschaften eines Exception Haltepunkts.

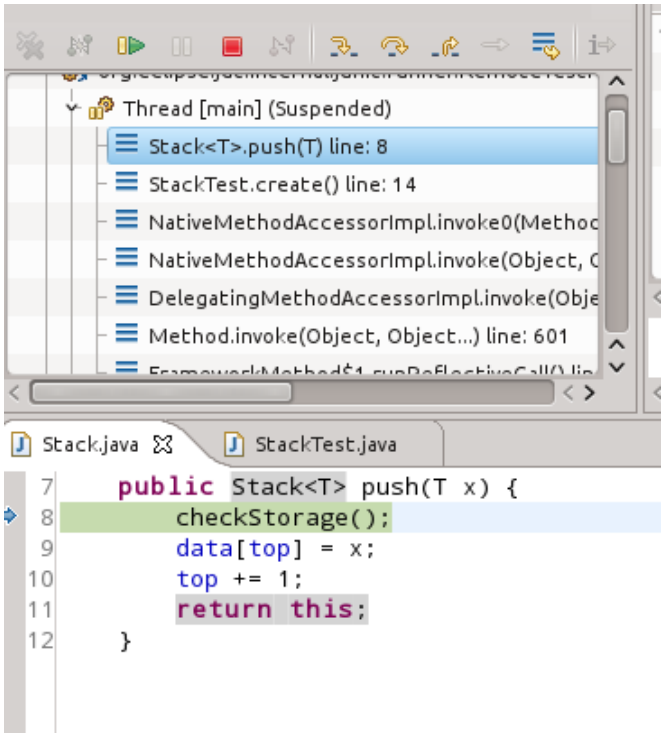

Abbildung 2.13: Zurückgehen zum Anfang der Methode (Drop to Frame).

Nachdem wir den Haltepunkt definiert haben, können wir den Debugger neu starten und wie erwartet hält er genau an der Stelle an, an der die Ausnahme geworfen wird. Wir erkennen auch, dass hier ein Problem besteht. Das Feld `data` ist nämlich voll und es passt somit kein weiteres Datenelement hinein. Offensichtlich ist in der vorherigen Zeile (`checkStorage`) etwas schief gelaufen. Nach meiner Meinung sollte diese Methode nämlich dafür sorgen, dass stets genug Speicher da ist.

Jetzt sind wir also schon über die Fehlerstelle hinausgelaufen. Wenn ein Fehler erkannt wird, muss er ja schon passiert sein. Wir könnten jetzt einen Haltepunkt am Anfang von `push` setzen, den Debugger neu starten und uns dann wieder bis zu der Fehlerstelle durchklicken. Das ist mühsam. Es geht viel einfacher. Durch Drücken von „Drop to Frame“ (blaue Linien) können wir den Zustand der Methodenausführung wieder zu ihrem Anfang zurücksetzen. Und dann werden wir einfach mittels „Step Into“ in die Methode `checkStorage` springen.

Jetzt können wir die folgenden Anweisungen Schritt für Schritt ausführen. Letztlich muss uns aber irgendwann auffallen, dass hier zwar ein neues Array passender Größe erzeugt wird und dass auch alle Daten kopiert werden. Es fehlt aber, das Speichern der Referenz des neuen Arrays in der Instanzvariable `data` (Abb. 2.14).

Der Debugger bietet noch weitere Möglichkeiten (bedingte Haltepunkte, Watchpoints, Auswertung von Ausdrücken). Schauen Sie auch das Video-Tutorial an: `http://eclipsetutorial.sourceforge.net/debugger.html`.

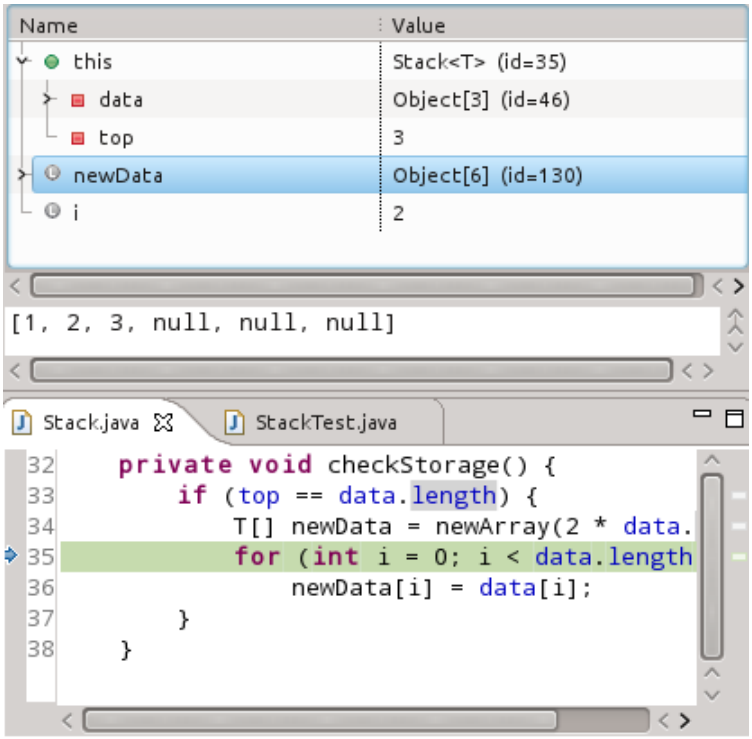

Abbildung 2.14: Hier fehlt was.

Kapitel 3

Objektorientierung

What's in a name? that which we call a rose
By any other name would smell as sweet.
William Shakespeare, Romeo and Juliet

Namen haben von sich aus keine feste Bedeutung. Sie müssen erst deklariert, also erklärt werden. In diesem Kapitel geht es insbesondere um die Bedeutung, die Variablen-, Klassen- und Typnamen haben können.

3.1 Ausdrücke und Typen

Die Begriffe *Ausdruck*, *Typ* und *Wert* spielen in jeder Programmiersprache eine wichtige Rolle. Wie Sie noch sehen werden, unterscheidet sich jedoch das Typkonzept der Objektorientierung stark von dem Typkonzept der prozeduralen Programmierung in C (und in Java). Das Typkonzept von Java versucht die Forderung der starken Typprüfung durch den Compiler mit der Forderung der Objektorientierung nach einem flexiblen Laufzeitverhalten zu vereinen. Das ist nicht immer einfach und Java ist dadurch sicher schwerer zu erlernen als objektorientierte Sprachen, die anstelle von Typdeklarationen eine Typprüfung zur Laufzeit vornehmen. Daher will auch ich Ihnen bereits bekannte Begriffe nochmals aufgreifen, wiederholen und vertiefen.

3.1.1 Werttypen und Referenztypen

Nach meiner Erfahrung gibt es immer wieder große Mißverständnisse beim Verständnis der Rolle des Datentyps. Vermutlich liegt das daran, dass Sie im ersten Semester die Bedeutung des Datentyps aus einer Perspektive kennengelernt haben, die für die Programmiersprache C und auch für den seinerzeit behandelten Stoff angemessen ist, die aber ungeeignet ist, die objektorientierte Programmierung zu verstehen.

Tabelle 3.1 stellt die unterschiedlichen Aufgaben einer Typangabe dar. So wie die elementaren Datentypen in den Programmiersprachen C und Java implementiert sind, ist es zwingend notwendig, dass jede Variable mit der korrekten Typinformation versehen ist.[1] Nur so kann der Compiler für die Variable die erforderlichen Speicherbedarf feststellen und nur nur so ist festgelegt, wie die Daten der Variablen kodiert sind.

[1] Die Typ*information* ist zwingend notwendig – nicht die Typ*angabe*. In modernen Programmiersprachen (auch in C++) kann der Compiler diese Information aus dem Typ der Variableninitialisierung herleiten.

Typ	Größe	max. Wert	Kodierung	Operationen
`short`	2	37267	2-Kompl.	+, -, *, /
`int`	4	2147483647	2-Kompl.	+, -, *, /
`double`	8	$1,8 \cdot 10^{308}$	IEEE 754	+, -, *, /
`Object`	4	—	Adresse	`equals,`
`String`	4	—	Adresse	`equals, charAt,`
`Stack`	4	—	Adresse	`equals, push,`

Tabelle 3.1: Aussage der Typangabe

Bei den in der Tabelle angeführten Referenzdatentypen `Object`, `String` und `Stack` können Sie erkennen, dass hier der Typname kein Unterscheidungsmerkmal für Speicherbedarf und Kodierung darstellt. Es ist wirklich so, wie es der folgende Merksatz ausdrückt:

Merksatz:

In einer rein objektorientierten Sprache (die auch Zahlen durch Objekte darstellt), braucht man keine statische Typangabe.

Wozu braucht eine objektorientierte Sprache dann noch Typinformation? Die rechte Spalte der von Tabelle 3.1 gibt die Antwort. Alle Wertdatentypen verstehen im Großen und Ganzen dieselben Operationen.[2] Es gibt auch hier Unterschiede (so ist die Shiftoperation << nicht für Gleitkommazahlen definiert). Diese Unterschiede werden aber oft als zweitrangig angesehen.

Bei den Referenzdatentypen ist es dagegen der Normalfall, dass unterschiedliche Objekte über unterschiedliche Methoden verfügen. Die Typangabe reduziert sich auf die (auch von der Informatiktheorie geforderte) Angabe der für die Datenelemente definierten Operationen.

Definition:

*Ein **abstrakter Datentyp** definiert eine Menge von Objekten zusammen mit den für sie definierten Operationen.*

Wir werden weiter unten, bei der Definition von Datenstrukturen nochmals auf abstrakte Datentypen zurückkommen. Hier können wir zunächst den Unterschied zu einem durch einen Klasse definierten *konkreten Typ* festhalten.

Definition:

*Ein **konkreter Datentyp** definierte die Kodierung von Daten und beschreibt die darauf definierten Operationen.*

An dieser Stelle wollen wir festhalten: Objekte werden durch Klassen, d.h. durch konkreten Datentypen, beschrieben; Variablen werden durch den abstrakten Typ deklariert, der die Menge der erlaubten Operationen festlegt. In Java werden konkrete Typen durch (konkrete) Klassen und abstrakte Typen durch Klassen (konkret oder abstrakt) und durch Interfaces beschrieben.

Der Compiler, der den Typ einer Variablen kennt, kann Typfehler feststellen. Er stellt sicher, dass in einer Variablen nur Referenzen von Objekten gespeichert sind, die mit dem

[2] In C haben wir nur numerischen Typen, in Java spielt allerdings `Boolean` eine Sonderrolle.

angegebenen Typ verträglich sind und dass nur Methoden aufgerufen werden, die durch den Typ angegeben sind.

Viele Programmiersprachen entlasten den Programmierer von der Pflicht, den Typ einer Variablen anzugeben. Dass bringt eine Reihe von Vorteilen mit sich und vermeidet die oft komplexen Regeln eines starren Typsystems. Man erkauft sich sich dies aber nicht nur mit dem Fehlen einer frühen Typprüfung sondern auch mit einer erheblich verminderten Ausführungseffizienz. In der Konsequenz findet sich diese Vorgehensweise vorwiegend in Skriptsprachen in denen es mehr auf Einfachheit und Flexibilität und weniger auf Effizienz ankommt.

3.1.2 Historischer Hintergrund

Ursprünglich wurden Datentypen in Programmiersprachen benötigt, um eine effiziente Übersetzung in Maschinencode zu ermöglichen. Damit waren zunächst die elementaren Datentypen für ganze Zahlen, Gleitkommazahlen und für Felder von Zahlen gemeint.

Die Zahlentypen unterscheiden sich hinsichtlich ihrer Kodierung und hinsichtlich ihres Speicherbedarfs. Sonst stellen sie aber alle das gleiche Konzept dar. Für alle Zahlen sind die gleichen Operationen erlaubt, Zahlen können automatisch oder aufgrund einer ausdrücklichen Umwandlungsfunktion von einer Darstellung in eine andere umgewandelt werden[3].

Relativ früh wurden auch schon Programmiersprachen entwickelt, die einen bequemeren Umgang mit Daten erlauben. Da frühe Computer fast stets an den Grenzen ihrer Leistungsfähigkeit betrieben wurden, wurden die dafür nötigen Performanceeinbußen aber im Allgemeinen nicht hingenommen.

Das begann sich erst zu ändern, als Anfang der 1980er die ersten Arbeitsplatzrechner verfügbar wurden. Da diese jetzt einem einzigen Nutzer uneingeschränkt zur Verfügung standen, war es möglich, einen Teil ihrer Leistungsfähigkeit für Programmier- und Bedienkomfort zu opfern.

Eine führende Rolle spielte damals das Forschungszentrum Xerox-Parc in Palo Alto (Kalifornien). Hier wurden die Konzepte von graphischen Entwicklungs- und Arbeitsumgebungen in Verbindung mit objektorientierter und funktionaler Programmierung entwickelt. Alan Kay, einer der Pioniere, war von der Idee getrieben, das Programmieren einfacher zu machen.

Die Arbeiten von Adele Goldberg, Alan Kay, Dan Ingalls und anderen mündeten in der Programmiersprache Smalltalk 80.

Smalltalk 80 unterscheidet sich grundlegend von prozeduralen Sprachen wie C. Es ist konsequent objektorientiert "*everything is an object*". Smalltalk enthielt von Anfang an eine mächtige interaktive Programmierumgebung und eine umfassende Bibliothek mit einer durchgehenden Klassenhierarchie. Smalltalk-Programme werden in Bytecode übersetzt und in einem speziellen Repository verwaltet. Fast alle Begriffe und die meisten Entwurfsmuster der objektorientierten Programmierung haben ihren Ursprung in der Smalltalk-Welt.[4]

Auch wenn Java viele Konzepte von Smalltalk übernommen hat, so unterscheidet sich Smalltalk doch in vielen Punkten. Die wichtigsten Eigenschaften sind:

[3] Genau genommen, kennt Java nicht verschiedene abstrakte Zahlentypen sondern nur unterschiedliche konkrete Realisierungen des abstrakten Konzepts der Zahl

[4] Auch die Entwicklung von Eclipse wurde stark durch Erfahrungen mit Smalltalk beeinflusst.

- Alles sind Objekte (auch Zahlen und boolesche Werte) und Instanzen von Klassen.
- Objekte werden zur Laufzeit erzeugt.
- Nicht mehr benötigte Objekte werden automatisch entfernt (*garbage collection*)..
- Objekte können nicht auf den Inhalt fremder Objekte zugreifen.
- Objekte senden sich Botschaften (*message*)
- Empfängerobjekte (*receiver*) empfangen die Botschaften.
- Ein Objekt reagiert auf eine Botschaft mittels einer Methode (*method*) seiner Klasse.
- Falls das Objekt nicht über eine passende Methode verfügt, wird eine Fehlermeldung erzeugt.
- Der Name „Objektorientierung“ kommt daher, dass alles über Objekte geregelt wird: Objekte speichern die Daten und entscheiden, wie mit Botschaften verfahren wird.
- Smalltalk Variablen kennen keine Typdeklaration.

Smalltalk ist das Vorbild für viele objektorientierte Sprachen.[5] Java wurde auch stark aus anderer Richtung, wie Simula und C++, beeinflusst. Als Besonderheiten von Java sind die folgenden Punkte zu nennen:

- Objekte werden durch Referenzen angesprochen. Referenzen werden meist (aber nicht zwingend!) durch die Speicheradresse des referierten Objekts kodiert.
- Die Java-Unterscheidung zwischen Wert- und Referenzdatentypen ist ein Kompromiss, der dem Streben nach Effizienz geschuldet ist.[6]
- Die Typdeklaration von Java fördert eine effiziente Übersetzung, eine frühe Fehlererkennung, und die Unterstützung durch Programmierwerkzeuge. Sie ist aber nicht für die Programmlogik notwendig.
- Das Typsystem von Java versucht, die starren Typregeln des Compilers mit den flexibleren Anforderungen der Objektorientierung in Einklang zu bringen. Das bringt große Vorteile, die aber auch mit komplizierten Sprachkonstrukten erkauft werden.[7]

In den folgenden Abschnitten will ich versuchen, Ihnen ein paar Aspekte des Java-Typsystems zu erläutern.

3.1.3 Grundbegriffe

Die im folgenden aufgeführten Grundbegriffe über Ausdrücke und Typen sollten Ihnen nicht unbekannt sein. Wegen der großen Bedeutung einer genauen Ausdrucksweise will ich sie aber hier nochmals wiederholen.

Die wichtigsten Elemente eines Typsystems sind *Werte*, *Variable* und *Ausdrücke*.

[5] Dazu gehört auch die durch Apple-Geräte bekannt gewordene Programmiersprache Objective-C.

[6] Es ist derzeit geplant, die Wertdatentypen mit Java 10 aufzugeben.

[7] Auch die Smalltalk-Entwickler hatten zunächst vor, der Sprache Typdeklarationen hinzuzufügen. Sie fanden aber keine befriedigende Lösung.

Definition:

*Ein **Ausdruck** ist:*

1. *ein **Literal**. Ein Literal ist ein direkt angegebener Wert. Beispiele für Literale sind Zahlen, Strings und die Wahrheitswerte* `true` *und* `false`.
2. *der Zugriff auf den Inhalt einer **Variablen**. Eine **Variable** ist ein Platzhalter für einen elementaren Wert oder für eine Objektreferenz. Der **Typ** einer Variablen legt fest, welche Werte oder welche Referenzen in der Variablen gespeichert werden können. Mittels der Typinformation entscheidet der Compiler ob eine Operation erlaubt ist.*[8]
3. *Der Aufruf einer **Methode** oder einer **Funktion**,* [9] *die ein Ergebnis zurückliefern.*
4. *die **Verknüpfung** von Ausdrücken durch Operatoren.*
5. *der **New-Ausdruck** zur Erzeugung eines Objekts.*
6. *die explizite **Typangabe** durch type cast. Bei elementaren Wertdaten bewirkt die Typangabe eine Umwandlung des Ausdruckswertes in einen äquivalenten Wert des angegebenen Typs. Bei Referenzen dient die Typangabe zunächst dazu, andernfalls vom Compiler verbotene Operationen zu ermöglichen. Der Compiler garantiert die Typsicherheit durch das Erzeugen eines Befehls zur Typprüfung.*[10].

Ein Ausdruck hat einen *Wert* und einen *Typ*. Hinsichtlich des Wertes muss man grundsätzlich zwischen den elementaren Werten und den Referenzen unterscheiden. Referenzen verweisen auf Objekte, deren Verhalten durch ihre *Klasse* definiert ist. Anders als die elementaren Werte werden Objekte in einem eigenen Speicherbereich, dem *Heap* verwaltet.

Da es in diesem Semester in erster Linie um Objektorientierung geht, werde ich häufig so tun, als gäbe es in Java nur Objekte und Objektreferenzen.

Die Unterscheidung zwischen *Wert* und *Objektreferenz* ist sehr technisch. Die Unterscheidung ist vom Standpunkt der Programmierung kontraproduktiv. Sie ist nur unter dem Gesichtspunkt der Performanceoptimierung zu rechtfertigen. Ich werde daher auch deshalb häufig die Wertdatentypen einfach "vergessen", wenn ich über Typkonzepte schreibe.

Anmerkung:

Werte sind unveränderliche Objekte, auch wenn die elementaren Datentypen in Java nicht die Eigenschaften „richtiger" Objekte besitzen. Bei unveränderlichen Objekten spielt es keine Rolle, ob man von der Objektreferenz oder von dem Objekt selbst redet. Es spielt dabei auch keine Rolle, ob ein Ausdruck nur eine weitere Referenz oder wirklich ein neues Objekt erzeugt. Bei veränderlichen Objekten ist das anders! Neuere objektorientierten Sprachen kennen keine Wertdatentypen. Es gibt jedoch keine objektorientierte Programmiersprache ohne Referenzen.

[8] In Java kann das bedeuten, dass der Compiler mögliche und sinnvolle Operationen verbietet, wenn er nicht die ausreichende Typinformation hat.

[9] Da sich Klassenfunktion wie die Funktion in C und nicht wie Methoden in Java verhalten, verwende ich nicht die Namen *statische Methode* oder *Klassenmethode*. Da in diesem Fall die Bedeutung klar ist, verwende ich dagegen häufig auch nur den Namen *Funktion*.

[10] Häufig findet nennt man die Typangabe *Typumwandlung* – dieser Begriff ist falsch oder *Typanpassung* – dieser Begriff ist ungenau. Der englische Begriff *type cast* – dieser Begriff ist sehr anschaulich, lässt aber ebenfalls den Mechanismus offen

3.1.4 Aufgaben des Typsystems

Bei der prozeduralen Programmierung gewährleistet die Typdeklaration von Variablen, dass sparsam mit dem Arbeitsspeicher umgegangen werden kann, und dass der Compiler bereits alle Maschinenoperationen exakt bestimmen kann. Die Programmiersprache C und ähnliche Sprachen gehorchen dem Motto „what can be done at compile time, shall not be deferred to runtime“.

Die Objektorientierung unterscheidet sich aber wesentlich dadurch von der prozeduralen Programmierung, dass sie nicht alle Entscheidungen durch den Compiler vorweg nehmen lässt. Vielmehr sollen die Objekte, die allerdings immer erst zur Laufzeit existieren, die entscheidende Rolle spielen.

Jedes Objekt hat einen Typ, der in Java durch die Klasse des Objekts beschrieben ist. Dieser Typ legt fest, wie sich das Objekt verhält. Der wichtigste Aspekt dieses Verhaltens ist die Frage, welche Methoden man mit dem Objekt aufrufen kann. Typfehler, wie das Aufrufen einer nicht vorhandenen Methode, sollten allerspätestens bei der Ausführung erkannt werden. Ebenso sollten auch Fehler, wie der Zugriff auf nicht alloziierte Feldelemente, erkannt werden. Am Ende sollen *alle Typfehler* erkannt werden, damit eine Programmiersprache als sicher gelten kann. Java ist sicher; C ist eine unsichere Sprache.

Java versucht möglichst viele Fehler bereits bei der Übersetzung zu erkennen. Viele sichere Programmiersprachen, wie z.B. die weit verbreiteten Skriptsprachen, führen die Typprüfung aber erst zur Laufzeit durch. Beide Vorgehensweisen haben ihre Vor- und Nachteile.

Zwecks Realisierung der Objektorientierung werden auch in Java wichtige Entscheidungen auf die Laufzeit verschoben. An erster Stelle ist hier die Entscheidung zu nennen, durch welche Methode eine Operation ausgeführt werden soll (späte Bindung).[11]. In Java können Typprüfungen bei Bedarf auf die Laufzeit verschoben werden.

Trotzdem besteht der Java-Standard auf der Typdeklaration von Variablen und auf der Typprüfung durch den Compiler. Dies bringt in der Regel weder Speicher- noch Laufzeitvorteile.[12] Es gibt aber einige andere positive Elemente einer Typkenntnis und Typprüfung im Compiler:

- Typangaben erhöhen die Lesbarkeit eines Programms. Sie fördern gleichzeitig einen überlegten Programmierstil.
- Durch die Typprüfung können viele Fehler frühzeitig erkannt werden.
- Die Typangabe hilft bei der automatischen Analyse von Programme. Sie ist die Grundlage vieler Hilfestellungen durch integrierte Entwicklungsumgebungen.
- Im Einzelfall kann der Compiler aufgrund der Typangabe auch Optimierungen vornehmen.

Man spricht bei der Typprüfung durch den Compiler von einer *statischen* Typprüfung. Eine Folge eines statischen Typsystems ist die Forderung, dass die Typen von Variablen und von Funktionsresultaten genau angegeben werden müssen.[13]

[11] Bei C wurde man sagen, welche Funktion durch einen Funktionsaufruf ausgeführt wird.

[12] Bei elementaren Datentypen ergibt die Typkenntnis allerdings einen tatsächlichen Laufzeitgewinn.

[13] Es gibt auch Programmiersprachen, in denen der Compiler Typinformation nach Möglichkeit aus dem Kontext ableitet (*type inference*).

3.1.5 Statischer und dynamischer Typ einer Objektreferenz

Ein zentraler Aspekt der Objektorientierung in Java ist das Zusammenspiel zwischen der statischen Typprüfung durch den Compiler und der Realisierung des objektorientierten Verhaltens zur Laufzeit. Damit verbunden ist die Unterscheidung zwischen dem *statischen Typ* und dem *dynamischen Typ* eines Ausdrucks.

Definition:

*Der **statische Typ** eines Ausdrucks ergibt sich aus dem Typ von Variablen und dem Typ von Funktionsresultaten sowie aus den Signaturen der Verknüpfungsoperatoren. Die Typregeln einer Programmiersprache definieren, welche Zuweisungen erlaubt sind, welche Operationen ausgeführt werden dürfen und welche Argumente übergeben werden dürfen. Das Einhalten der statischen Typregeln wird durch den Compiler geprüft.*

Definition:

*Der **dynamische Typ** eines Ausdrucks ist der Typ seines Ergebnisses. Bei den Werttypen von Java stimmt der dynamische Typ eines Ausdrucks immer mit dessen dynamischen Typ überein. Bei Referenzdatentypen ist dies anders. Der Typ eines Objekts ist durch seine Klasse bestimmt. Die von einem Ausdruck erzeugten Referenzen, verweisen immer auf ein Objekt, dessen Klasse mit dem statischen Typ des Ausdrucks* verträglich *aber nicht unbedingt identisch ist.*

Die Regeln der Typverträglichkeit basieren auf den Beziehungen zwischen den Klassen und Schnittstellen von Objekten und Variablen. Sie werden weiter unten besprochen. Kurz gesagt, bedeutet Typverträglichkeit, dass man einer Variablen eines Obertyps ein Objekt eines Untertyps zuweisen darf.

In einem Java-Programm ist man oft darauf angewiesen dem Compiler klarzumachen, welchen genauen Typ ein Ausdruck hat. Der Programmierer, der den Ablauf und den Hintergrund des Programms kennt, hat oft genaueres Wissens über den dynamischen Typ eines Ausdrucks als in dem – dem Compiler allein zugänglichen – statischen Typ zum Ausdruck kommt. Wenn diese genaue Information nötig ist, um z.B. eine Methode aufrufen zu können, muss das Programm sie dem Compiler mittels einer Typangabe mitteilen.

Definition:

*Eine **Typangabe** ist eine Operation, die auf eine Referenz wirkt. Weder die Referenz noch das referierte Objekt werden durch den Cast verändert. Vielmehr wird dem Compiler Information über den Typ des Ausdrucks vermittelt. Zur Laufzeit wird überprüft, ob diese Information sich mit dem dynamischen Typ des Objekts verträgt. Ist dies nicht der Fall, wird eine* `ClassCastException` *geworfen. Die Typangabe von Referenzen ist nicht mit der syntaktisch identischen Typkonvertierung von numerischen Werten zu verwechseln!*

Definition:

*Es gibt zwei unterschiedliche Arten von Typbeziehungen von Referenztypen, nämlich die Zuweisung zu einen allgemeineren Obertyp, auch **Typerweiterung** oder **Aufwärtsanpassung (up cast)** genannt, und die Zurodnung zu einen spezielleren Untertyp, namens **Typverengung** oder **Abwärtsanpassung (down cast)**. Die Behandlung einer Referenz als zu einem Obertyp gehörend, ist völlig problemlos. Dabei geht allerdings statische Typinformation verloren. Der umgekehrte Weg erfor-*

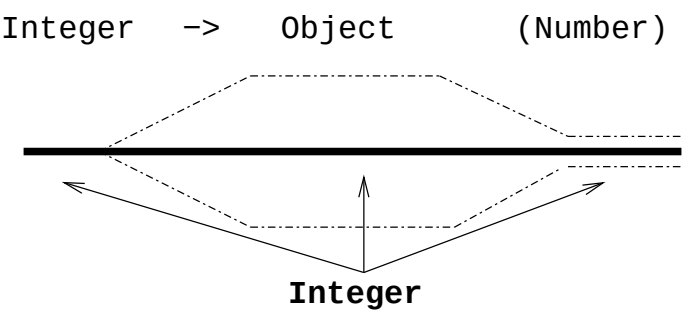

Abbildung 3.1: In der Abbildung ist der dynamische Typ eines Objekts durch eine dick gezeichnete Linie und der statische Typ durch eine gestrichelte Linie dargestellt. Die Abbildung zeigt die Typerweiterung (durch Zuweisung) von `Integer` nach `Object`, gefolgt von einer Typverengung nach `Number`. `Number` ist ein Obertyp von `Integer`, könnte aber auch für andere Datentypen, wie `Double`, stehen. Entscheidend ist, dass der statische Typ immer den dynamischen Typ miteinschließt.

> *dert die Angabe des gewünschten Typs durch einen Cast-Ausdruck. Der Compiler prüft, ob Objekte diesen Typs aufgrund der Typregeln überhaupt vorkommen können. Die endgültige Typprüfung erfolgt dann zur Laufzeit. Bei der Abwärtsanpassung wird durch die Typangabe dem Compiler die bei einer vorhergehenden Aufwärtsanpassung verloren gegangene Information wieder vermittelt. Das Laufzeitverhalten der beteiligten Objekte bleibt in jedem Fall völlig unverändert.*

Die Abbildung 3.1 verdeutlicht den Zusammenhang zwischen dynamischem und statischem Typ bei der Typerweiterung. Wir gehen von folgendem Beispiel aus. Dazu müssen Sie wissen, dass `Number` ein Obertyp von `Integer` und `Object` der allgemeinste Obertyp aller Java-Typen ist.

```
Integer intVariable = Integer.valueOf(7);
Object objVariable = intVariable;
Number numVariable = (Number) objVariable;
```

Zunächst wird ein `Integer`-Object in einer Variablen gleichen Typs gespeichert. Statischer und dynamischer Typ sind identisch. Anschließend wird die Objektreferenz in die Variable `objVariable` übertragen. In dieser Variablen können grundsätzlich beliebige Objektreferenzen gespeichert werden. In der Abbildung 3.1 ist dies durch die aufgeweiteten gestrichelten Linien angedeutet. Die durchgehende dickere Linie bezeichnet den gleichbleibenden dynamischen Typ des Objekts. Die Typangabe `(Number)` ermöglicht das Speichern des Inhalts von `objVariable` in einer Variablen vom Typ `Number`. Der Typ `Number` lässt Referenzen auf `Integer` und auf andere Zahlenobjekte zu. Er ist aber erheblich enger gefasst als `Object`.

Natürlich ist es auch möglich, nach einer Typangabe das Objekte wieder in einer Variablen vom Typ `Integer` zu speichern.

Die Abbildung 3.2 illustriert den Fehler in der folgenden Anweisungsfolge.

```
Integer intVariable = Integer.valueOf(7);
Object objVariable = intVariable;
String strVariable = (String) objVariable;
```

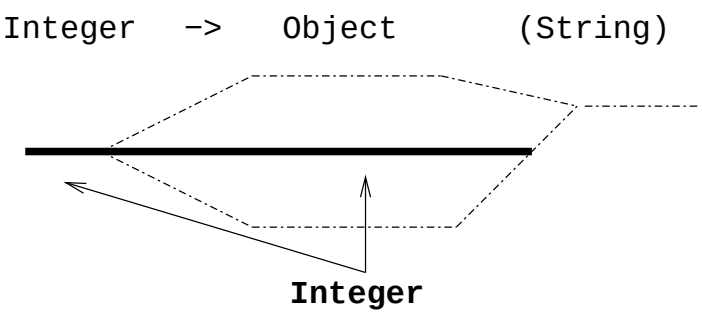

Abbildung 3.2: Diese Abbildung zeigt eine auf eine Typerweiterung folgende fehlerhafte Typverengung. Wenn in einer Variablen vom Typ `Object` eine Objektreferenz vom Typ `Integer` gespeichert ist, so lässt sich diese nicht als `String` ansprechen oder speichern. Die Typangabe `(String)` führt zu der Ausnahme `ClassCastException`.

Für den Compiler sieht diese Anweisungsfolge ebenfalls korrekt aus, da der Compiler ja nur die statischen Typdeklaration betrachtet und nicht die Inhalte der Variablen nachhält.[14]

Die gestrichelten Linien schränken auch hier wieder die möglichen Objektreferenzen ein. Die Typverengung nach `String` ist für `Integer`-Referenzen nicht möglich. Das Resultat ist ein Fehlerzustand mit `CastClassException`.

Diese Beispiele bezogen sich auf Referenzdatentypen. Es soll noch einmal betont werden, dass für die elementaren Wertdatentypen andere Regeln gelten. Dort bewirkt die Tyangabe immer eine Umwandlung der Zahlendarstellung. Den Begriff Typerweiterung kann man bei Zahlen so verstehen, dass man einen Typ mit einem größeren Darstellungsbereich wählt. Bei der Umwandlung in einen kleineren Darstellungsbereich können Fehler auftreten. Diese werden jedoch von Java nicht erkannt. Konkret bedeutet dies, dass die Umwandlung von `double` zu `int` zu unsinnigen Zahlen führen kann. Hier ein paar Beispiele:

```
(byte) 3000        -> -72
(int) 1e30         -> 2147483647
(float) 1e100      -> Infinity
```

Auch hier gilt wieder gilt das Fazit, dass die Wertdatentypen sozusagen die Low-Cost Variante der objektorientierten Referenzdaten sind.

Man kann sich den Unterschied zwischen statischem und dynamischem Typ auch auf eine etwas andere Art merken:

Merksatz:

> *Der statische Typ kann als **Schnittstellentyp** angesehen werden. Mit ihm wird ein Obertyp für ein Ausdruckresultat festgelegt. Der statische Typ sagt dem Compiler, **welche Methoden die entsprechenden Objekte kennen** (welche Botschaften sie verstehen). Der dynamische Typs kann dagegen als **Implementierungstyp** angesehen werden, der festlegt, **wie ein Objekt implementiert** ist, d.h. durch welche Klasse es erzeugt wurde und durch welche Methode es eine Operation ausführt. **Das Wesen der Objektorientierung besteht darin, dass der Schnittstellentyp allgemeiner als***

[14] Natürlich ist es denkbar, dass ein Compiler diesen und ähnliche andere Fehler entdecken kann. Er wäre aber sicher überfordert, wenn die Zuweisungen in unterschiedlichen Programmteilen stünden. Mit Sicherheit könnte er den Fehler nicht entdecken, wenn er von Einzelheiten des Programmablaufs abhängt.

der Implementierungstyp definiert sein kann. *Diese Typregeln ermöglichen es Code, der für einen bestimmten Schnittstellentyp geschrieben wurde, für unterschiedlich implementierte Objekte zu verwenden (Polymorphie).*

Eine Eselsbrücke für die Unterscheidung von statischem und dynamischen Typ ist: Ein statischer Typ ist etwas das „feststeht“ und daher bereits vom Compiler geprüft werden kann; ein dynamischer Typ „steht nicht fest“ und kann erst zur Laufzeit überprüft werden.

Die Unterscheidung vom statischem und dynamischen Typ (Schnittstellentyp und Implementierungstyp) ist auch hilfreich bei der Unterscheidung und Bewertung von Programmiersprachen.

Streng statisch getypte nicht-objektorientierte Programmiersprachen kennen keine Unterscheidung zwischen statischem und dynamischem Typ. Der Vorteil ist, dass die Typprüfung ausschließlich im Compiler erfolgt, und zur Laufzeit keine Typkonflikte auftreten können. Der Nachteil ist, dass es absolut unmöglich ist, Wiederverwendung in größerem Stil zu realisieren. Das klassische Beispiel ist die von Niklas Wirth für die Informatikausbildung entwickelte Programmiersprache *Pascal*.

Unvollständig getypte Programmiersprachen definieren ebenfalls statische Typregeln, die durch den Compiler geprüft werden. Auch sie kennen keine dynamische Typprüfung zur Laufzeit. Um eine hinreichende Flexibilität und Wiederverwendbarkeit zu ermöglichen, erlauben sie die Umgehung der Typprüfung (ungeprüfte Typangabe). Die meisten älteren Programmiersprachen folgen diesem Konzept (*Fortran*, *C*, *C++*). Der Vorteil liegt in der Kombination von teilweise erreichter Typsicherheit und maximaler Laufzeiteffizienz.[15] Der große Nachteil liegt in den hohen Kosten, die durch schwer zu entdeckende Fehler entstehen. [16]

Dynamisch getypte Programmiersprachen kennen keine Typdeklaration und keinen statischen Typ. Schnittstellen können nur durch Kommentare definiert werden und der Compiler kann keine Typprüfung vornehmen. Dafür wird aber eine exakte Laufzeitüberprüfung vorgenommen. Der Vorteil ist die bequeme und flexible Programmierung. Der Nachteil ist die relativ geringe Laufzeiteffizienz und die späte Fehlererkennung. Beispiele sind objektorientierte Programmiersprachen wie Smalltalk, die Skriptsprachen sowie die meisten nicht-prozeduralen Sprachen.

Streng getypte, objektorientierte Programmiersprachen enthalten ein statisches *und* ein dynamisches Typkonzept. Sie garantieren, dass alle Typfehler erkannt werden und ermöglichen gleichzeitig die Formulierung polymorpher, vielseitig nutzbarer Algorithmen. Im Vergleich zu Programmiersprachen ohne dynamisches Typkonzept werden geringe Laufzeitnachteile in Kauf genommen. Moderne optimierende Laufzeitcompiler versuchen, dieses Manko auf ein Minimum zu reduzieren. Beispiele sind *Java* und *C#*. Moderne Sprachkonzepte (z.B. parametrisierte Datentypen) versuchen die Notwendigkeit der dynamischen Typprüfung zu verringern. Hierbei ist natürlich auch die richtige Programmiertechnik von Bedeutung. Es ist Sache des Programmierers zu entscheiden, welche Rolle er der statischen Typsicherheit beimisst, bzw. welche Flexibilität und polymorphe Wiederverwendbarkeit er benötigt.

[15] Die ebenfalls von Niklas Wirth entwickelte Sprache Modula 2 erlaubt die kontrollierte Außerkraftsetzung der Typprüfung.

[16] C++ ermöglicht die dynamische Typprüfung mittels `dynamic_cast`.

Dynamisch getypte Programmiersprachen eignen sich besonders gut für die schnelle Entwicklung von Prototypen. Auch mag der Aufwand zur statischen Typdeklaration lästig erscheinen. Neben der größeren Fehlersicherheit liegt aber ein unschätzbarer Vorteil in der verbesserten Analysierbarkeit und sicheren Veränderbarkeit des Programms.

3.2 Ein Softwaresystem ist durch Schnittstellen gegliedert

In diesem Abschnitt will ich anhand einiger Beispiele zeigen, was eine Schnittstelle ist, wie eine reine Schnittstelle in Java durch ein Interface realisiert wird und welche Vorteile sich daraus ergeben. Ich werde dabei zeigen, dass der Begriff der Klasse deutlich komplexer ist als der der Schnittstelle.

3.2.1 Eine Klasse erfüllt zwei Aufgaben

Sie haben vielleicht noch die Definition für eine Klasse im Ohr: „Eine Klasse beschreibt die Struktur und das Verhalten einer Menge von Objekten.“, oder anders ausgedrückt: „Eine Klasse ist dazu da, Objekte zu erzeugen. Sie deklariert deren Instanzvariable und definiert deren Methoden.“

Diese Definitionen sind nicht falsch; sie benennen die *Hauptaufgabe* einer Klasse, nämlich die Beschreibung und Erzeugung von Objekten. Zusätzlich hat eine Klassendeklaration aber noch eine weitere Aufgabe.

Diese zweite Aufgabe folgt aus der im letzten Kapitel besprochenen statischen Typprüfung durch den Compiler. Eine Klasse dient nämlich auch dazu, einen Typ für ihre Objekte zu definieren. In dynamisch getypten Sprachen gibt es diese Rolle nicht.[17]

Betrachten wir als ein Beispiel eine ganz einfache Klasse und fragen wir uns, welche Möglichkeiten sie zur Verfügung stellt. Die Klasse `ElapsedTime` kann in der angegebenen Form genutzt werden, um das Zeitverhalten von Algorithmen zu messen.

```
/**
 * This is a simple stop watch.
 * It measures the total elapsed time between
 * <tt>start</tt> and <tt>stop</tt> in seconds.
 * <p>
 * Caveat: The accuracy of the measurement depends on the
 * underlying operation system.
 */
public class ElapsedTime {
    private long startTime;
    private boolean running = false;

    /**
     * (Re-)start the measurement.
     * It is not allowed to start a clock that already
     * has been started.
     * Time is always reset to zero.
     *
     * @throws IllegalStateException if the
     * clock is already running.
     */
    public void start() {
```

[17] Bei den dynamisch getypten Sprachen, wird bei jedem Methodenaufruf erst zur Laufzeit geprüft, ob die Methode vorhanden ist.

```
        if (running)
            throw new IllegalStateException("already running"
                );
        running = true;
        startTime = System.currentTimeMillis();
    }

    /**
     * Stops the measurement and returns
     * the total elapsed time since start.
     * It is not allowed to stop a clock that
     * has already been stopped.
     *
     * @return time in seconds.
     * @throws IllegalStateException if the clock
     *         is not running.
     */
    public double stop() {
        if (!running)
            throw new IllegalStateException("not running");
        long endTime = System.currentTimeMillis();
        running = false;
        return 0.001 * (endTime - startTime);
    }
}
```

Ich behaupte, dass mit dieser Klassendefinition verschiedene Aspekte gelöst sind:

1. Die Dokumentation der Klasse verrät mir, was ich damit anfangen kann, welche *Operationen* ich mit einem Objekt ausführen kann, und welche Information ich dabei erhalte.
2. Die Klasse sagt mir (und dem Compiler), wie ich *Variablen anlegen* kann und welche Operationen ich damit ausführen kann.
3. Die Klasse sagt mir (und dem Compiler) wie ich *Objekte erzeugen* kann und durch welche Methoden diese die angegebenen Operationen ausführen.

Für Entwickler, die eine bestimmte Klasse nutzen wollen, ist oft nichts wichtiger als ein genauer Kommentar, der exakt die Funktionalität einer Klasse beschreibt. Der Entwickler, der die Klasse verwendet, erwartet dabei, dass die Implementierung (Punkt 3) diesen Kommentaren entspricht.

Für den Compiler und für die formale Betrachtung der Programmiersprache sind nur die beiden letzten Punkte wichtig, nämlich die Festlegung eines *Typs* mit dem man Variablen anlegen kann und mit dem man bestimmte Operationen ausführen kann, und die Beschreibung von *Objekten* die konkrete Werte speichern und konkrete Methoden ausführen.

Definition:

*Ein (abstrakter) statischer **Typ** beschreibt eine Menge von Operationen. Eine (konkrete) **Klasse** erzeugt Objekte. Eine Klasse definiert also den dynamischen Typ ihrer Objekte. Sie definiert aber auch einen statischen Typ, der z.B. bei der Deklaration von Variablen verwendet werden kann.*

Der Unterschied zwischen Typ und Klasse ist leicht zu erkennen:

```
ElapsedTime t;        // ElapsedTime ist eine Typangabe.
t = new ElapsedTime() // Die Klasse erzeugt ein Objekt.
/*
 * t ist eine Variable mit dem Typ ElapsedTime.
 * t enthaelt die Referenz auf ein Objekt.
 */
t.start();
/*
 * start() ist eine Operation des Typs ElapsedTime
 * das Objekt fuehrt eine Methode der Klasse ElapsedTime aus.
 */
```

Vielleicht finden Sie es verwirrend, dass `ElapsedTime` zwei verschiedene Bedeutungen haben soll, je nachdem, wo es gerade steht, oder unter welchem Aspekt man eine bestimmte Programmzeile betrachtet.

Diese Unterscheidung ist ganz zentral für das Verständnis der Programmierung in Java. Im Übrigen macht der Compiler selbst diesen Unterschied. Er selbst interessiert sich nämlich bei der *Übersetzung* der Anweisung `t.start();` nur für die statische Typinformation. Dagegen interessiert sich die *Ausführung* der Anweisung durch die virtuelle Maschine nur noch für die konkrete Implementierung der Methode in der Klasse `ElapsedTime`.

3.2.2 Das Interface-Konzept von Java

Obwohl man im Zusammenhang mit Java und Objektorientierung vielleicht zuerst an den Begriff der Klasse denkt, gibt es in Java ein deutlich wichtigeres und einfacheres Konzept, nämlich die Beschreibung eines *reinen Typs* durch ein Java-Interface. Für unser Beispiel sieht das so aus:

```
/**
 * An IClock-Object measures time differences between
 * <code>start</code> and <code>stop</code> in seconds.
 * <p>
 * Different implementations may have different
 * interpretations on what they understand as time
 * difference.
 */
public interface IClock {
    /**
     * (Re-)starts the measurement.
     * Time is always reset to zero.
     *
     * @throws IllegalStateException if the clock is
     * already running.
     */
    public void start();

    /**
     * Stops the measurement and returns the time
     * since start.
     * This interface does not specify how this
     * time difference is measured
     * Wether it is the elapsed time or the netto
     * processor time or some other time measure
     * is left to the implementing class.
     *
     * @return time in seconds.
     * @throws IllegalStateException
```

```
     *         if the clock is not running.
     */
    public double stop();
}
```

Wenn Sie diese Schnittstellenbeschreibung sehen, sollte Ihnen sofort klar sein, was sie *nicht* ist. Man kann mit ihr ganz sicher keine Objekte erzeugen. Es gibt weder irgendwelche Angaben über eventuell notwendige Instanzvariable noch eine Angabe über die Ausführung von Methoden!

Definition:

*Ein **Interface** der Programmiersprache Java beschreibt einen statischen Typ, das heißt die Schnittstelle von Objekten. Mit einem Interface kann man Variablen deklarieren aber keine Objekte erzeugen. Ein Interface enthält nur abstrakte Methoden als Deklaration der Operationssignaturen und unter Umständen auch die Definition von Konstanten (*`static final`*).*

Schauen wir uns das folgende Beispiel an, das ohne die Kenntnis einer Klasse auskommt.

```
/**
 * Wir untersuchen hier das Zeitverhalten eines Teils eines
 * Algorithmus.
 */
double responseTime(IClock t) {
    tueWasOhneZeitMessung();
    t.start();
    tueWasMitZeitMessung();
    double result = t.stop();
    tueWiederWasOhneZeitMesung();
    return result;
}
```

Der Vorteil der Methode `responseTime` ist, dass sie nicht festlegt, zu welcher Klasse der Parameter `t` gehört. Das ist ihr nämlich völlig egal. Wichtig ist nur, dass das übergebene Objekt über die beiden Methoden `start` und `stop` verfügt. Genau das wird durch die Schnittstelle `IClock` garantiert.

Es stellt sich die Frage, wie wir diese Methode sinnvoll aufrufen können. Natürlich müssen wir dabei ein konkretes Objekt erzeugen.

```
System.out.println(
    "Zeit: " + responseTime(new ElapsedTime()));
```

Das Beispiel ist nur *fast* perfekt. Es funktioniert so nicht, weil der Java-Compiler keinen Zusammenhang zwischen dem Interface `IClock` und der Klasse `ElapsedTime` erkennt und daher eine Fehlermeldung ausgibt.

Naürlich sehen *Sie* schon den Zusammenhang: Die Klasse `ElapsedTime` implementiert alle Methoden der Schnittstelle `IClock` mit der exakt passenden Signatur. Das ist genau das, was man meint, wenn man sagt, dass die Klasse eine Schnittstelle *implementiert*. Allerdings muss man das in Java bereits bei der Deklaration der Klasse ausdrücken, damit der Compiler diese Behauptung auch prüfen kann. Die verbesserte Form der Klasse `ElapsedTime` sieht wie folgt aus (dabei lasse ich die Kommentare weg):

```
public class ElapsedTime implements IClock {
    private long startTime;
    private boolean running = false;

    public void start() {
        if (running)
            throw new IllegalStateException("already running"
                );
        running = true;
        startTime = System.currentTimeMillis();
    }

    public double stop() {
        if (!running)
            throw new IllegalStateException("not running");
        long endTime = System.currentTimeMillis();
        running = false;
        return 0.001 * (endTime - startTime);
    }
}
```

Nach dieser verbesserten Klassendeklaration ist der oben angegebene Aufruf von `responeTime(new ElapsedTime())` erlaubt.

Merksatz:

Bei der Implementierung einer Schnittstelle durch eine konkrete Klasse müssen grundsätzlich alle dort definierten Methoden implementiert werden. Es können aber auch weitere Methoden zur Klasse hinzugefügt werden.

3.2.3 Vorteile der Verwendung einer Interface-Einheit

Gut, werden Sie sagen, das leuchtet mir soweit ein. Aber wir hätten das Ganze auch ohne Verwendung der Schnittstelle `IClock` hingekriegt. Warum sollen wir uns die unnötige Schreibarbeit machen, nur damit ein Purist einen „reinen“ Typ sieht.

Der Einwand ist berechtigt. Ich muss Ihnen daher sofort mindestens einen Vorteil nennen. Später werden Sie noch weitere Vorteile von Schnittstellen kennen lernen.

Bleiben wir bei dem angesprochenen Beispiel.[18] Die Java-API erlaubt uns nur die Messung der tatsächlich vergangenen Zeit, nämlich der *elapsed time*. Über den Umweg der C-Schnittstelle der virtuelle Maschine (JNI = Java Native Interface) können wir aber beliebige C-Funktionen, wie z.B. die Funktion `clock`, aufrufen. `clock` verspricht laut Dokumentation die Angabe einer Zeit, die mit der echten Prozessorzeit zu tun hat.[19]

Wir brauchen also eine Klasse `CPUTime`. Da es auch dabei um Zeitmessungen geht, werden wir exakt dieselbe Schnittstelle wie bei der Klasse `ElapsedTime` verwenden. Dies drücken wir dadurch aus, dass wir auch hier die Schnittstelle `IClock` implementieren.

```
/**
 * Objects of this class measure differences in processor
    time.
```

[18] Inzwischen ist als weiterer Grund hinzugekommen, dass neuere Java-Bibliotheken auch andere Methoden zur Zeitmessung kennen.

[19] Die Erfahrung zeigt, dass sich die Funktion `clock` auf verschiedenen Betriebssystemen unterschiedlich verhält.

```
 */
public class CPUTime implements IClock {
    private double startTime = -1;

    /* Hiermit wird die C-Funktion deklariert */
    private static native double clock();

    public void start() {
        if (startTime >=  0)
            throw new IllegalStateException();
        startTime = clock();
    }

    public double stop() {
        if (startTime < 0)
            throw new IllegalStateException();
        double difference = clock() - startTime;
        startTime = -1;
        return 1E-3 * difference;
    }

    static {
        System.loadLibrary("clock");
    }
}
```

Sehen Sie den realen Vorteil, der Verwendung *einer* Schnittstelle für zwei Klassen? Wir können jetzt die Methode `responseTime` und alle Methoden, bei denen es nur auf die Grundfunktionalität von `start` und `stop` ankommt, ohne weiteres mit allen möglichen konkreten Implementierungen aufrufen. Am Beispiel:

```
System.out.println("CPU-Zeit: " +
    responseTime(new CPUTime());
System.out.println("echte Zeit: " +
    responseTime(new ElapsedTime());
```

Das hier beobachtete Verhalten ist ein erstes Beispiel von *Polymorphie*. Ehe ich näher auf diesen Begriff eingehe, will ich ein paar weitere Beispiele angeben.

Anmerkung:

Das Schlüsselwort `native` bei der Deklaration der Funktion `clock` sagt aus, dass diese in C geschrieben ist. Wie dies genau zu geschehen hat, soll hier nicht besprochen werden. Hinweise dazu finden sich in der Standard Java-Dokumentation in dem Kapitel JNI (Java Native Interface). Der C-Quelltext ist im Anhang aufgeführt.

3.2.4 Polymorphe Algorithmen

Gehen wir einmal zu einem anderen Beispiel über, dass Sie bereits kennen, nämlich zu der Klasse `Bruch` zur Darstellung von Brüchen. Bei diesem Beispiel kann man fragen, wie man ein Array von Brüchen sortieren kann.

Es ist klar, dass es möglich sein muss. Wenn Sie sich aber die bisherigen Beispiele für die Klasse `Bruch` anschauen, werden Sie feststellen, dass da noch etwas fehlt. Um Brüche (oder irgend etwas anderes) sortieren zu können, müssen wir unbedingt wissen, nach welchem Kriterium die Brüche angeordnet werden sollen. Konkret: Wir brauchen eine Vergleichsfunktion für Brüche.

Es ist nicht schwer, eine solche Vergleichsfunktion zu schreiben. Der Einfachheit halber nehme ich an, dass die Klasse `Bruch` so aufgebaut ist, dass Brüche stets gekürzt sind und dass der Nenner immer positiv ist. Wir können dann z.B. eine Funktion `groesserGleich` wie folgt schreiben:

```
public class Bruch {
    ...
    public boolean groesserGleich(Bruch b) {
        return zaehler*b.nenner >= nenner*b.zaehler;
    }
    ...
}
```

Jetzt müssen wir nur noch einen Sortieralgorithmus für Brüche schreiben, der diese Methode benutzt.

Doch halt! Ehe wir das machen, sollten wir in der Java-Klassenbibliothek nachschauen, ob es nicht vielleicht schon einen passenden Algorithmus gibt. Tatsächlich finden wir im Paket `java.util` die Klasse `Arrays`, die gleich mehrere Sortierverfahren anbietet. Für unsere Zwecke ist die folgende Funktion geeignet:

```
/**
 * Sorts the specified array of objects into ascending
 * order, according to the natural ordering of its
 * elements.
 * All elements must implement the <code>Comparable</code>
 * interface.
 *
 * @param a the array to be sorted.
 * @throws ClassCastException if the array contains
 *         elements that are not mutually comparable
 *         (e.g. strings and integers).
 */
 public static void sort(Object[] a)
```

Obwohl `sort` für ein Array von `Object` definiert ist, können wir es auch für ein Array von `Bruch` aufrufen. Wenn wir diese fertige Methode für Brüche verwenden wollen, müssen wir gemäß der Spezifikation aber dafür sorgen, dass alle Arrayelemente, also unsere Brüche, die Schnittstelle `Comparable` implementieren. Diese Schnittstelle findet sich bereits in `java.lang`. Ich lasse hier mal den langen Klassen- und Methodenkommentar des Originals beiseite und gebe nur eine verkürzte Form an.

```
public interface Comparable<T> {
    /**
     * Vergleicht das Argument mit dem Empfaengerobjekt.
     * Das Ergebnis ist positiv, gleich 0, oder negativ,
     * je nachdem das Empfaengerobjekt groesser, gleich,
     * oder kleiner als das Argument ist.
     *
     * @return Vergleichsergebnis.
     * @throws ClastCastException wenn der Vergleich nicht
     *         moeglich ist.
     */
     public int compareTo(T other);
}
```

Was müssen wir tun? Wir müssen dafür sorgen, dass die Klasse `Bruch` die Schnittstelle `Comparable` und damit die Methode `compareTo` implementiert . Danach ist das Sortieren ganz einfach:

```
Bruch[] feld = new Bruch[10];
feld[0] = new Bruch(1, 2);
...
feld[9] = new Bruch(1, 6);
Arrays.sort(feld);
```

Ich gebe Ihnen hier die notwendige Erweiterung der Klasse `Bruch` an.

```
public class Bruch implements Comparable<Bruch> {
   ...
   public int compareTo(Bruch that) {
       return zaehler*that.nenner - nenner*that.zaehler;
   }
}
```

Hier ist das Beispiel mit Typparametern geschrieben. Wenn Sie diese nicht mögen, können Sie das auch so schreiben:

```
public class Bruch implements Comparable {
   ...
   public int compareTo(Object other) {
       Bruch that = (Bruch) other;
       return zaehler*that.nenner - nenner*that.zaehler;
   }
}
```

Wenn Sie die beiden Lösungen vergleichen, erkennen Sie, dass der Typparameter in der ersten Implemntierung dem Compiler ermöglicht, die nötige Typprüfung vorzunehmen. In der zweiten Lösung ist dies durch die Laufzeitprüfung mittels Typangabe ersetzt.

Das Beispiel zeigt, dass die Verwendung von reinen Schnittstellen in Form von Interface-Einheiten tatsächlich Vorteile bietet. Die Verwendung der Methode `compareTo` ist nicht auf das Sortieren beschränkt. Zum Beispiel ermöglicht sie auch die binäre Suche und, nicht zu vergessen, auch die Anwendung in beliebigen noch zu schreibenden Algorithmen, in denen es nur auf den Größenvergleich zwischen Brüchen ankommt.

3.2.5 Polymorphe Datenstrukturen

Eine grundlegende Aufgabe von Software ist der Aufbau von Datenstrukturen. Weiter unten beschäftigen wir uns mit unterschiedlichen Realisierungen. Hier soll erst einmal nur auf Arrays eingegangen werden.

Arrays sind ein Spezialfall des Speicherns von Daten in einem *Behälter*. Mit dem Begriff Behälter ist gemeint, dass man diesem Behälter (Array) Daten hinzufügen kann und dass man später wieder auf alle oder gezielt auf einzelne Elemente zugreifen kann. Wie das im Einzelnen geschieht, kann verschieden sein. In manchen Anwendungen werden Daten unter einem Suchbegriff abgelegt, über den sie später wieder gefunden werden können (Java-Bibliothek: Interface `java.util.Map`). Bei einem Array stehen die Datenelemente an einem bestimmten numerierten Feldplatz.

Auf die unterschiedliche Art des Zugriffs kommt es hier nicht an. Es geht nur um das Problem, dass Objekte, die zu unterschiedlichen Klassen gehören, in ein und demselben Behälter gespeichert werden können.

Im letzten Abschnitt hatten wir gesehen, dass eine Schnittstelle es ermöglicht Objekte verschiedener Klassen gleich zu behandeln. Es sollte Sie nicht wundern, dass dies auch der Weg ist, wie wir Objekte verschiedener Klassen in einen Behälter oder ein Feld bekommen.

Schauen wir uns als Beispiel das folgende Problem an. Wir schreiben ein Programm zur Verwaltung elektronischer Schaltungen und elektronischer Bauelemente wie Kondensator, Widerstand, Transistor, integrierter Schaltkreis und andere. Die Bauelemente sind untereinander so verschieden, dass wir für jeden Typ eine eigene Klasse schreiben. Für den Zweck der Erzeugung einer Stückliste sollen alle Bauelemente in einem Feld gespeichert werden.

Bis hierher haben wir also eine Reihe von Klassen, wie `Widerstand`, `Transistor` oder `Kondensator`. Um ein Feld für die Stückliste erzeugen zu können, brauchen wir jetzt noch einen Datentyp – genauer eine Schnittstelle – die von allen diesen drei Klassen implementiert wird.

In der einfachsten Form können wir unser Beispiel dann wie folgt schreiben:

```
public interface Bauelement {
}

public class Widerstand implements Bauelement {
    ...
}

public class Transistor implements Bauelement {
    ...
}

public class Kondensator implements Bauelement {
    ...
}
```

Die Klassen und die Schnittstelle müssen in verschiedenen Dateien mit dem richtigen Namen stehen!

Innerhalb unserer Stücklistenverwaltung kann die Anwendung der Klassen etwa wie folgt aussehen:

```
void verarbeiteStueckliste(Bauelement[] elemente) {
    for (Bauelement x : elemente) {
        x.methodenAufruf(); // ???
    }
}

void testStueckliste() {
    Bauelement[] a = new Bauelement[3];
    a[0] = new Transistor(...);
    a[1] = new Widerstand(...);
    a[2] = new Kondensator(...);
    verarbeiteStueckliste(a);
}
```

Ich hoffe, das Beispiel (mit dem total willkürlich konstruierten `testStueckliste`) vermittelt Ihnen die Idee um was es geht. In gewisser Hinsicht ist, das Beispiel ein bisschen ähnlich zu dem weiter oben diskutierten Sortieren von Brüchen. Wir haben wieder eine Methode, die mit unterschiedlichen Datentypen zurecht kommen soll. Der Unterschied besteht nur darin, dass jetzt in dem Array, zur gleichen Zeit *verschiedenartige Objekte* sind.

Sie haben aber sicher die drei Fragezeichen „???“ in der dritten Zeile des Beispiels bemerkt? Hier haben wir – und der Compiler – ein Problem: Woher wissen wir, dass der Aufruf `x.methodenAufruf` erlaubt ist? Die Antwort lautet: So wie die Schnittstelle `Bauelemente` definiert ist, ist dieser Aufruf nicht erlaubt!

Wenn wir eine gemeinsame Schnittstelle für unterschiedliche Klassen definieren wollen, müssen wir auch unbedingt festlegen, welchen Operationen für alle Objekte definiert sein sollen.

Eine etwas realistischere Deklaration von `Bauelemente` könnte so aussehen:

```
public interface Bauelement {
    public String getBezeichnung();
    public double getPreis();
}
```

Natürlich ist damit allen Bauelement-Klassen (wie `Transistor`) die Aufgabe auferlegt, diese Methoden richtig zu implementieren. Dafür können wir jetzt aber auch etwas sinnvollere Anwendungen angeben, wie z.B. eine Methode zur Berechnung der Kosten der Bauelemente.

```
double berechneGesamtKosten(Bauelement[] elemente) {
    double summe = 0.0;
    for (Bauelement x : elemente) {
        summe += x.getPreis();
    }
    return summe;
}
```

Anmerkung:

Mit einer Variablen einer beliebigen Schnittstelle kann man auch alle in der Klasse `Object` deklarierten Methoden aufrufen. Beispiele sind `toString` und `equals`. Dies folgt aus der Tatsache, dass jede Klasse von der Klasse `Object` abgeleitet ist.

3.2.6 Die Erweiterung einer Interfaceeinheit

Zunächst einmal ist festzuhalten, dass eine Klasse beliebig viele Schnittstellen implementieren kann: Man muss nur alle Interfacenamen im der Implements-Klausel des Klassenkopfes auflisten und alle geforderten Methoden implementieren. Es ist auch kein Problem, wenn eine Methode von mehreren Schnittstellen gleichzeitig gefordert wird.

Konzepte der Alltagswelt werden in Java am besten durch Interfaces ausgedrückt (Tier, Vogel, Haus); schließlich können wir auch bei diesen abstrakten Begriffen zunächst nur ein paar allgemeine Aussagen darüber treffen, was man mit den entsprechenden Objekten anfangen kann.

Die alltäglichen Konzepte stehen oft aber in einer besonderen Hierarchiebeziehung. Die

Begriffe *Tier* und *Vogel* sind verwandt, da der Begriff Tier ein Oberbegriff zu Vogel ist. Dies bedeutet, dass alles, was man über ein Tier sagen kann, auch auf einen Vogel zutrifft.

Übertragen auf die Java-Schnittstellen führt dies zu der *Erweiterungsbeziehung* von Schnittstellen.

Definition:

Ein Java-Interface kann im Kopf hinter dem Schüsselwort `extends` *eine Liste von Interface-Namen aufführen (Extends-Klausel). Damit wird ausgesagt, dass alle Deklarationen der angeführten Schnittstellen durch die neue Schnittstelle übernommen werden.*

Das folgende Beispiel soll die Situation verdeutlichen:

```
public interface ITime {
    public double getTime();
}

public interface ISimulationTime
        extends ITime, Comparable<ITime> {
    public void setRate(double factor);
    public double getRate();
}
```

Objekte, die der Schnittstelle `ISimulationTime` genügen, verfügen über die Methoden `getTime`, `compareTo` sowwie über `setRate` und `getRate` (und natürlich über `toString` usw.).[20]

Ein Objekt kann in einer Variablen gespeichert werden, deren Typ durch seine Klasse direkt oder indirekt implementiert wurde.

Ein gutes objektorientiertes System zeichnet sich dadurch aus, dass seine Schnittstellen sehr sorgfältig entworfen sind. Dazu gehört, dass man sich jedes Mal genau überlegt, was zu den wesentlichen Eigenschaften einer Schnittstelle gehört. In unserem Beispiel ist eine entprechende Verbesserung möglich:

```
// bessere Loesung, da Zeiten immer vergleichbar sind.
public interface ITime extends Comparable<ITime> {
    public double getTime();
}

public interface ISimulationTime extends ITime {
    public void setRate(double factor);
    public double getRate();
}
```

Die Beziehung zu `Comparable` wird also gewissermaßen vererbt. Es schadet aber auch nicht (und ändert überhaupt nichts), wenn man Schnittstellen wiederholt aufführt. Dies gilt auch für das wiederholte Hinschreiben von Methoden. Manchmal macht man das, um in den Methodenkommentaren auf Besonderheiten hinzuweisen.

[20] Wir kommen hier nahe an die Grenze der einfachen Verwendungen von Typparametern. Bei Typparametern gelten nicht die normalen Regeln für Ober- und Untertypen. Die gezeigten Beispiele sind also nur gültig, wenn wir hier, <ITime¿ als Datentyp für die Operation `compareTo` autritt.

```
// moegliche Loesung
public interface ITime extends Comparable<ITime> {
    public double getTime();
}

public interface ISimulationTime
        extends ITime, Comparable<ITime> {
    public void setRate(double factor);
    public double getRate();
    public int compareTo(ITime that);

    /**
     * gibt eine virtuelle Zeit zurueck.
     * @return Zeit in msec.
     */
     public double getTime();
}
```

Die Stärke von Schnittstellen kommt natürlich auch bei der Definition von Methodenschnittstellen und von Datenstrukturen zum Tragen.

3.3 Die Ableitung von Klassen

Im Folgenden soll das Konzept der Vererbung, d.h. der Ableitung von Klassen genauer diskutiert werden. In diesem Zusammenhang bedeutet Vererbung, dass sowohl die Schnittstelle einer allgemeineren Klasse als auch ihre Implementierung an eine abgeleitete Klasse weitergegeben wird.

Vererbung ist eine mächtige Technik, die aber auch mit vielen Fallstricken und Fehlerquellen behaftet ist, so dass sie immer mit großer Vorsicht verwendet werden sollte! Im Unterschied zu der bisher besprochenen Anwendung von Interface-Einheiten, verletzt die Vererbung nämlich das Geheimnisprinzip! Dies liegt einfach daran, dass es bei der Vererbung von Klassen ja um die Vererbung der *Implementierung* der Oberklasse. Um die abgeleitete Klasse richtig schreiben zu können, muss ich in der Regel einige Implementierungsdetails der Oberklasse kennen; wenn die Oberklasse geändert wird, wird die Unterklasse oft nicht mehr korrekt arbeiten. Natürlich gilt auch hier, dass es der Programmierstil den Unterschied zwischen guter und schlechter Verwendung von Vererbung macht.

3.3.1 Die Ableitung von Klassen ist eine Schnittstellenbeziehung

Nachdem ich Ihnen Interfaces als neues Konzept vorgestellt habe, geht es jetzt stärker um die Klasse, d.h. die Beschreibung der Implementierung von Objekten. Wie anfangs betont, ist die Klasse *auch* eine Schnittstelle. Daher kann man annehmen, dass man auch durch eine Klasse gemeinsame Eigenschaften mehrerer Klassen beschreiben kann.

Dies ist tatsächlich so und es ist auch ein wichtiger Aspekt der Klassenvererbung. Die Vererbung von Klassen hat weitere Eigenschaften. Dies soll in diesem einleitenden Abschnitt verdeutlicht werden.

Bei der Vererbungsbeziehung von Klassen verwendet man ebenso wie bei der Erweiterung von Schnittstellen das Schlüsselwort `extends`. Gehen wir zum Beispiel davon aus, dass wir eine Klasse `GeneralTimer` haben. Wie sieht dann eine abgeleitete Klasse `SpecialTimer` aus?

```
public class SpecialTimer extends GeneralTimer {
   ... // das Innenleben
}
```

Für das Innenleben der neuen Klasse wollen wir uns erst einmal nicht interessieren, sondern nur fragen, wie es mit den Typeigenschaften der Klassen bestellt ist.

Definition:

Die Vererbungsbeziehung zwischen Klassen wird in Java durch das Schlüsselwort `extends` *ausgedrückt. Die Klasse von der abgeleitet wird, nennt man* ***Oberklasse (superclass)*** *(manchmal auch* ***Basisklasse****) und die abgeleitete Klasse nennt man* ***Unterklasse (subclass)****. Man spricht bei der Vererbungsbeziehung auch von einer* ***Spezialisierung*** *der Oberklasse durch die Unterklasse oder umgekehrt von einer* ***Generalisierung*** *der Unterklasse durch die Oberklasse.*

Wenn wir nun Oberklasse und Unterklasse vergleichen, so ist es selbstverständlich, dass beide sowohl Typen darstellen, als auch gleichzeitig Objekte beschreiben. Ein Unterschied ergibt sich hinsichtlich der Verträglichkeit von Objekten mit den jeweiligen Typen.

In Pseudocode kann man die möglichen Beziehungen wie folgt beschreiben:

```
Oberklasse var1 = new Oberklasse();   // erlaubt
Unterklasse var2 = new Unterklasse(); // erlaubt

Unterklasse var3 = new Oberklasse();  // nicht erlaubt
Oberklasse var4 = new Unterklasse();  // auch erlaubt !!!
```

Die Initialisierung der Variablen `var1` und `var2` bietet nicht Neues. Es überrascht auch nicht, dass einer Variablen der Unterklasse kein Objekt der Oberklasse zugewiesen werden darf. Schließlich stellt die Unterklasse eine Spezialisierung dar, bei der eventuell Operationen möglich sind, die in der Oberklasse nicht implementiert wird.

Die letzte Beziehung nämlich die Zuweisung eines Objekts der Oberklasse zu einer Variablen vom Typ der Unterklasse stellt eine wichtige Eigenschaft von Klassen dar.

Merksatz:

Eine Oberklasse ist ein Obertyp der Unterklasse, damit ist der Typ der Unterklasse mit dem Typ der Oberklasse verträglich.

3.3.2 Die Syntax des Klassenkopfes

Die Ableitung von einer vorhandenen Klasse wird in Java durch die extends-Deklaration im Klassenkopf ausgedrückt. Zusätzlich zur Vererbung kann die Information über die Implementierung einer oder mehrerer Schnittstellen angegeben werden. Jede Klasse ist genau von einer einzigen Oberklasse abgeleitet. Wenn im Klassenkopf keine Oberklasse angegeben ist, wird vom Compiler die Klasse `Object` als Oberklasse eingesetzt.

Klassenkopf :
(*Sichtbarkeitsangabe*)?
(final | abstract)?
class *Name*
(extends *Name*)?
(implements *Namensliste*)?

Da an dieser Stelle erstmals die fast vollständige Syntax des Klassenkopfes angegeben ist (ein paar Details werden später erläutert), will ich die fünf Elemente des Klassenkopfes hier aufzählen (der fehlende Punkt betrifft nur den Spezialfall der geschachtelten Klasse).

1. Bei top level Klassen ist die Sichtbarkeitsangabe `public` möglich. Fehlt diese, so gilt die Paketsichtbarkeit. Bei geschachtelten Klassen, kann die Sichtbarkeitsangabe auch `private` oder `protected` sein.
2. Das Schlüsselwort `final` bedeutet, dass von dieser Klasse keine weitere Klasse abgeleitet werden darf. Das Schlüsselwort `abstract` kennzeichnet eine abstrakte Klasse, die (wie ein Interface) abstrakte Methoden enthalten darf. Von abstrakten Klassen kann keine Instanz gebildet werden. `final` und `abstract` dürfen nicht gleichzeitig im Kopf einer Klasse auftreten.
3. In jedem Klassenkopf muss das Schlüsselwort `class` gefolgt von dem Namen der Klasse stehen.
4. Jede Klasse ist von einer Oberklasse abgeleitet. Durch den extends-Teil wird der Name dieser Klasse angegeben. Fehlt diese Angabe, so ist die Klasse `Object` die Oberklasse. Klassen in deren Klassenkopf `final` steht, können hier nicht eingesetzt werden.
5. Die optionale Angabe `implements` gefolgt von einer durch Komma getrennten Liste von Schnittstellennamen, benennt die durch die Klasse implementierten Schnittstellen.

Als Beispiel ist hier der Kopf der Datei String.java aus der Java-Klassenbibliothek angegeben (die vielen Kommentare sind weggelassen):

```
package java.lang;

public final class String extends Object
    implements Serializable, Comparable, CharSequence
{
    // Der Koerper der Klasse String
}
```

Dieser Kopf sagt uns schon eine ganze Menge über die Klasse `String` aus:

– Da die Klasse mittels `final` als nichtableitbar deklariert ist, kann man keine weitere Klasse ableiten. Diese Vorkehrung wurde deshalb getroffen, damit das in der Klasse `String` implementierte Konzept der Unveränderlichkeit von Strings nicht umgangen werden kann (Wertobjekt!).

- Da die Klasse von der Klasse `Object` abgeleitet ist, verfügt sie über alle dort definierten Methoden. Einige dieser Methoden gelten unverändert weiter, andere sind in der Klasse `String` überschrieben. Ein Beispiel ist die Methode `equals` mit der die Gleichheit zweier Strings festgestellt werden kann.
- Da die Klasse die Schnittstelle `Comparable` implementiert, verfügt sie über eine Methode `compareTo` mit der sich zwei String-Objekte verleichen lassen (alphabetische Reihenfolge). Dadurch ist zum Beispiel die Methode `Arrays.sort` in der Lage, ein Feld von Strings zu sortieren.
- Da die Klasse die Schnittstelle `Serializable` implementiert, ist es möglich, String-Objekte in einer Datei zu speichern oder über das Netz zu verschicken.
- Die Schnittstelle `CharSequence` sagt aus, dass Methoden wie `charAt` oder `length` vorhanden sind, dass man einen String quasi als konstantes Stringarray behandeln kann. Diese Schnittstelle gibt es ab der Java-Release 1.4.

Der Klassenkopf sagt nichts darüber aus, ob die Klasse eventuell noch weitere Methoden deklariert.

Definition:

Der (vollständige) Typ eines Objekts ist durch seine Klasse festgelegt. Die Menge der erlaubten Operationen umfasst alle Methoden, die in der Klasse oder in einer ihrer Oberklassen definiert sind. Alle Oberklassen und alle implementierten Interfaces stellen Obertypen der Klasse dar.

Aus der Deklaration der Klasse `String`

```
class String implements Comparable, Serializable ...
```

ergeben sich die (echten) Obertypen `Object`, `Comparable` und `Serializable`. String-Objekte können in Variablen eines dieser Typen oder in einer Variablen des Typs `String` gespeichert werden.

3.4 Die Implementierungs-Eigenschaften der Vererbung

Eine Klasse beschreibt einen Typ und eine konkrete Implementierung.

Merksatz:

Durch die Ableitung einer Unterklasse aus einer Oberklasse wird eine neue erweiterte Klasse gebildet. Der Typ der Unterklasse ist eine Spezialisierung (Untertyp) des Typs der Oberklasse und daher mit diesem verträglich. Die konkrete Implementierung der Objekte der Unterklasse wird teilweise von der Oberklasse übernommen. Sie kann jedoch auch von dieser abweichen, soweit die Festlegungen der Schnittstelle beachtet werden.

Der durch die Klasse definierte Typ ist durch die Schnittstelle beschrieben. Eine abgeleitete Klasse erweitert den Typ ihrer Oberklasse, indem sie unter Umständen weitere Operationen hinzufügt. Dies hat zur Folge, dass (syntaktisch) jede Variable der Oberklasse auch Referenzen auf ein Objekt der Unterklasse enthalten darf.

Merksatz:

*Eine korrekt abgeleitete Klasse muss dem von Barbara Liskov aufgestellten **Substitutionsprinzip** genügen. Dieses Prinzips sagt aus, dass überall dort, wo Variablen der Oberklasse stehen, Objekte einer Unterklasse verwendet werden können.*

Zunächst erscheint dieser Merksatz trivial, da ja bei bloßer Erweiterung der Funktionalität der Oberklasse, kein anderes Verhalten ihrer normalen Methoden zu erwarten ist. In Wirklichkeit liegt jedoch hier genau ein Problem, da nicht automatisch gewährleistet ist, dass die Funktionsweise der Unterklasse sich mit den Typangaben der Oberklasse verträgt.

Neben der Erweiterung der Schnittstelle bewirkt die Vererbung nämlich auch eine Erweiterung und in der Regel auch eine Veränderung der Implementierung.

3.4.1 Vererbung von Datenfeldern

Zunächst schauen wir uns die Vererbung von Datenfeldern (Attributen) an.

Definition:

Ein Objekt einer abgeleiteten Klasse enthält alle in der Oberklasse definierten Variablen. *Die eigene Klasse kann weitere Variablen definieren. Dies ist unabhängig davon, wie die Sichtbarkeit der Variablen definiert ist. Methoden der Unterklasse können unmittelbar nur auf Variablen zugreifen, die sichtbar sind. Damit ist der direkte Zugriff auf private Variable der Oberklasse verboten. Der Zugriff auf Variable, die in der Oberklasse* `public` *oder* `protected` *deklariert sind, ist immer erlaubt. Wenn sich die Unterklasse im selben Paket wie die Oberklasse befindet, ist zusätzlich der Zugriff auf Variable mit Paketsichtbarkeit gestattet. Definiert die Unterklasse eine Variable mit demselben Namen wie eine Variable der Oberklasse, so **verdeckt** ihre Definition die Variable der Oberklasse. Die Variable der Oberklasse kann aber immer noch durch die Form* `super.`*Name angesprochen werden.*

Das folgende Beispiel dient dazu, die Definition zu erläutern.

```
class Superclass {
    private int privat;
    protected int geschuetzt;
    int paketsichtbar;
    public int oeffentlich;
    protected int wirdVerdeckt;
}

class Subclass extends Superclass {
    private int neueVariable;
    protected int wirdVerdeckt;

    /* ein Objekt enthaelt:
     *   privat, geschuetzt, paketsichtbar,
     *   offentlich, super.wirdVerdeckt
     *   neueVariable, wirdVerdeckt
     *   also insgesamt 7 int-Variable.
     */

    void methode() {
        /*
         * die Methode kann ansprechen:
         *   geschuetzt
```

```
            *   oeffentlich
            *   super.wirdVerdeckt
            *   neueVariable
            *   this.wirdVerdeckt (== wirdVerdeckt)
            *   paketsichbar, wenn im selben Paket
            * also insgesamt 5 oder 6 Variable des Objekts
            *
            * privat: kann nicht angesprochen werden
            * paketsichbar: ausserhalb des Pakets kein Zugriff
            * wirdVerdeckt oder this.wirdVerdeckt meint die
            *   neue Variable
            * super.wirdVerdeckt meint die verdeckte Variable.
            */
    }
}
```

Natürlich bedeutet, die Auflistung dieser Möglichkeiten nicht, dass man sie auch alle anwenden sollte! Auch bei der Vererbung sollte man sich soweit es geht an das Geheimnisprinzip halten. Dies bedeutet konkret, dass in der Regel nicht auf Variable der Oberklasse zugegriffen werden sollte, bzw. dass in der Regel alle Variablen privat sein sollten. Die selten verwendete Ausnahme `protected` gibt den eindeutigen Hinweis, dass diese Variable von der abgeleiteten Klasse benutzt werden sollte.

3.4.2 Vererbung und Überschreiben von Methoden

Definition:

Objekte der abgeleiteten Klasse verfügen über alle Methoden der Oberklasse soweit diese nicht überschrieben oder verdeckt wurden. *Zusätzlich kann die Unterklasse weitere Methoden definieren. Innerhalb der Methoden der abgeleiteten Klasse kann nicht direkt auf private Methoden der Oberklasse zugegriffen werden. Private Methoden und Methoden die* `final` *deklariert sind, können nicht überschrieben werden.*

In der Definition werden die Begriffe *Überschreiben* und *Verdecken* verwendet. Verdecken wird weiter unten im Zusammenhang mit statischen Klassenelementen beschrieben. Das Überschreiben ist der zentrale Mechanismus der objektorientierten Erweiterung einer Klasse.

Zunächst soll der Begriff *Signatur* näher beleuchtet werden.

Definition:

Unter der ***Signatur*** *einer Operation versteht man die Anzahl und die Typen der Parameter und im Allgemeinen auch den Ergebnistyp.* [21] *In Java gilt die Regel, dass bei festgelegtem Namen und festgelegter Parameterliste der Ergebnistyp eindeutig sein muss. Zusätzlich ist durch die Signatur festgelegt, welche geprüften Ausnahmen von der Operation geworfen werden können.*

```
double division(int a, int b)
```

hat die Signatur

[21] Die Java-Sprachdefinition zählt den Rückgabetyp nicht zur Signatur.

```
division: int, int -> double
```

In Java ist es nicht erlaubt, im selben Sichtbarkeitsbereich eine zweite Methode der folgenden Form zu definieren:

```
int division(int a, int b)
    // Ergebnis muesste double lauten!
```

Nach dieser kurzen Auffrischung des Begriffs der Signatur, können wir nun definieren, was Überschreiben bedeutet:

Definition:

*Eine Methode einer abgeleiteten Klasse **überschreibt** eine Methode der Oberklasse, wenn sie denselben Namen und dieselbe Signatur besitzt. Wenn ein Objekt der Unterklasse die überschriebene Operation ausführen soll, so wird stets die Methode der Unterklasse ausgeführt. Dies ist unabhängig davon, wie das Objekt im angesprochen wird. Es ist zum Beispiel gleichgültig, ob dies über eine Variable der Unterklasse oder über eine Variable vom Typ der Oberklasse geschieht. Innerhalb von Methoden der Unterklasse kann man überschriebene Methoden der Oberklasse mit* `super.`*Name aufrufen.*

Auch hier soll ein kleines Beispiel die Situation verdeutlichen:

```
class Superclass {
    public void methode(int x) {
        System.out.println("Superclass.methode-1");
    }
    public void methode(int a, int  b);
        System.out.println("Superclass.methode-2");
    }
}

class Subclass extends Superclass {
    public void methode(int a) {
        System.out.println("Subclass.methode-1");
    }
    public boolean sonstwas() {
        super.methode(4);
        return true;
    }
}
```

Welche Methoden implementiert nun ein Objekt von `Subclass` und welche Methoden, werden im konkreten Fall aufgerufen?

```
Subclass sub = new Subclass();
sub.methode(3);      // "Subclass.methode-1"
sub.methode(1,2);    // "Superclass.methode-2"
sub.sonstwas();      // "Sublcass.sonstwas"
                     // und "Superclass.methode-1"

Superclass oben = sub;
oben.methode(3)      // wie oben
oben.methode(1,2)    // wie oben
oben.sonstwas();     // Compiler-Fehler !!
```

```
((Subclass)oben).sonstwas();  // wie oben

Superclass s = new Superclass();
s.methode(3);       // "Superclass.methode-1" !!
```

Beachten Sie bitte das letzte Beispiel. Formal sieht der Aufruf `oben.methode(3)` exakt identisch aus, wie der Aufruf `s.methode(3)`. In beiden Fällen wird eine Methode desselben Namens (und derselben Signatur) mit einer Variablen vom Typ `Superclass` aufgerufen. Die Ergebnisse sind jedoch verschieden, da die Methoden von Objekten von verschiedenen Klassen ausgeführt werden. In der Variablen `oben` wird ein Objekt der Klasse `Subclass` angesprochen, in der Variablen `s` jedoch eins der Klasse `Superclass`.

Merksatz:

Der Typ einer Variablen oder eines Ausdrucks entscheidet, welche Operationen erlaubt sind. Die Klassenzugehörigkeit eines Objekts entscheidet, nach welcher Methode diese Operation ausgeführt wird. Die Methode wird objektorientiert *ausgewählt!*

Die Liste der erlaubten Aufrufe ist noch nicht ganz vollständig. Da die Klasse `Superclass` von der Klasse `Object` abgeleitet ist, kommen alle dort definierten Methoden hinzu:

```
if (sub.equals(oben)) ...
System.out.println(sub.toString());
oben.wait();
// usw.
```

Anmerkung:

Ich versuche, wenn es geht, die Ableitung von Klassen zu vermeiden. Wenn man jedoch die Ableitung von Klassen vorsehen will, muss man die Oberklasse geeignet entwerfen und so ausführlich dokumentieren, dass keine überraschenden Fehler auftreten! Wenn ich erstmals eine Klasse schreibe, markiere ich sie in der Regel mit dem Schlüsselwort `final` um darauf hinzuweisen, dass ich mir keine Gedanken gemacht habe, ob eine sinnvolle Ableitung möglich ist. In der Regel bevorzuge ich die intensive Nutzung von Schnittstellen. In einigen Fällen ist auch die im folgenden Abschnitt besprochene Verwendung von abstrakten Klassen sinnvoll.

3.4.3 Konstruktoren

Konstruktoren spielen in Java eine Sonderrolle. Sie sind keine Methoden, auch wenn sie formal so ähnlich aussehen. Ihre Aufgabe ist die Initialisierung eines neuen Objekts. Da dies die wichtigste Aufgabe einer Klasse ist, gilt die folgende Regel;

Merksatz:

Konstruktoren werden nicht vererbt.

Es kommt noch eine zweite Regel hinzu, die die Initialisierung des geerbten Teils des Objekts beschreibt:

Merksatz:

Jeder Konstruktor muss den Konstruktor der Oberklasse aufrufen. Der Konstruktor der Oberklasse wird angesprochen durch das Schlüsselwort `super` *gefolgt von der Liste der Parameter (oder einem Paar leerer Klammern). Der Aufruf von* `super` *ist die erste Aktion in einem Konstruktor.*

Schließlich gibt es noch eine Regel, die den Gebrauch des Defaultkonstruktors beschreibt.

Merksatz:

Wenn kein anderer Konstruktor definiert wird, wird automatisch ein parameterloser Defaultkonstruktor erzeugt. Dieser ruft intern den Defaultkonstruktor der Oberklasse auf. Wenn in einem Konstruktor kein Aufruf eines Konstruktors der Oberklasse steht, wird automatisch deren Defaultkonstruktor aufgerufen. Wenn die Oberklasse keinen parameterlosen Konstruktor besitzt muss einer der Konstruktoren der Oberklasse mittels `super` *aufgerufen werden.*

Das folgende etwas unsinnige Beispiel soll die Konstruktoraufrufe verdeutlichen:

```
public class Bruch // extends Object {
    private int zaehler;
    private int nenner;

    public Bruch(int zaehler, int nenner) {
        super(); // nicht notwendig
        ...
    }
    protected final int getZaehler() {
        return zaehler;
    }
    ...
}

public class PositiverBruch extends Bruch {
    public PositiverBruch(int zaehler, int nenner) {
        super(zaehler, nenner); // zwingend notwendig
        if (getZaehler() < 0)
            throw new IllegalArgumentException();
    }

    ...
}
```

3.4.4 Generatorfunktionen

Definition:

Eine ***Generatorfunktion*** *ist eine Klassenfunktion zum Erzeugen von Instanzen der Klasse.*

Gemäß dieser Definition hat eine Generatorfunktion eine ähnliche Aufgabe wie ein Konstruktor.

Java erlaubt es, dass man mehrere Konstruktoren mit unterschiedlichen Parameterlisten definiert. Dies ist mitunter aber immer noch unbefriedigend:

- Die Unterscheidung gelingt nur, wenn die Konstruktoren sich in Anzahl oder Typ der Parameter unterscheiden.
- Die unterschiedliche Funktion der Konstruktoren lässt sich nicht im Namen ausdrücken.
- Der Konstruktor legt die Klasse des erzeugten Objekts genau fest. Es ist nicht möglich dies von übergebenen Argumenten abhängig zu machen.
- Ein Konstruktor legt immer ein *neues* Objekt an. Es ist nicht möglich, einfach ein bereits vorhandenens Objekt zu verwenden (*caching*).

Alle diese Defizite können mittels Generatorfunktionen gelöst werden. Dabei bleibt es dem Programmierer überlassen zu entscheiden, ob er zusätzlich zu den Generatorfunktionen auch weiterhin öffentliche Konstruktoren anbietet, oder ob er diese nur für die interne Objekterzeugung – hier sind Konstruktoren unabdingbar – vorsieht. Es ist aber zu beachten, dass die Ableitung von einer Klasse nur möglich ist, wenn auf den Konstruktor zugegriffen werden kann!

Merksatz:

Soll die Ableitung von einer Klasse möglich sein, dann muss der Konstruktor der Oberklasse in der Unterklasse zugänglich sein. Dies bedeutet in der Regel, dass er `public` *oder* `protected` *sein sollte. Bei paketinternen Klassen, kann auch die Paketsichbarkeit vorgesehen sein.*

Das Gesagte sollte an dem folgenden – etwas konstruierten – Beispiel für die Klasse Bruch deutlich werden.

```
public class Bruch // extends Object {
    private int zaehler;
    private int nenner;
    private static final BRUCH ZERO = new Bruch(0, 1);

    protected Bruch(int zaehler, int nenner) {
        ...
    }

    /** gibt das passenste Bruchobjekt zurueck */
    public static Bruch fromInt(int zahl) {
        if (zahl == 0)
            return ZERO;
        else if (zahl > 0)
            return new PositiverBruch(zahl) :
        else
            return new Bruch(zahl);
     }

    /** gibt das passenste Bruchobjekt zurueck */
    public static
    Bruch fromZaehlerNenner(int zaehler, int nenner) {
        if (zaehler == 0 && nenner != 0)
            return ZERO;
        boolean positiv = zaehler >= 0 == nenner > 0;
        return positiv ?
            new PositiverBruch(zaehler, nenner) :
            new Bruch(zaehler, nenner);
     } ...
}
```

Mögliche Aufrufe können dann so aussehen:

```
Bruch a = Bruch.fromInt(-3);
Bruch b = Bruch.fromZaehlerNenner(7, 8);
Bruch c = Bruch.fromInt(0);
```

Beachten Sie, dass bei der Erzeugung eines positiven Bruchs ein Objekt der Klasse `PositiverBruch` zurückgegeben wird. Natürlich muss weiterhin als statischer Typ der allgemeinere Typ der Oberklasse gewählt werden.

Das Beispiel mit der Rückgabe von `ZERO` verdeutlicht die Möglichkeit, Objekte mehrfach zu verwenden. Das ist bei der Klasse `Bruch` dann möglich, wenn sie unveränderlich ist.

In der Praxis werden Sie viele Anwendungen entdecken wo Generatorfunktionen sinnvoll einzusetzen sind. Generatorfunktionen sind ein Spezialfall der *Erzeugungsmuster*[22]

Gute Beispiele für Generatorfunktionen finden sich auch in der Java-Bibliothek. Die Verpackungsklasse `Integer` verfügt über verschiedene Konstruktoren. Anstatt diese aufzurufen, verwendet man aber, wie schon im vorigen Kapitel angemerkt, die Generatorfunktion `valueOf`. Dann hat man auch die Gewährt, dass optimaler Gebrauch von bereits vorhandenen Objekten gemacht wird.

```
Integer x = Integer.valueOf(3); // macht Integer aus Zahl
String y = String.valueOf(42);  // macht String aus Zahl
```

3.5 Abstrakte Klassen

Nach den bisherigen Diskussionen sollte klar sein, dass Klassensysteme durch Schnittstellen „zusammengehalten“ und durch Klassen implementiert werden. Wenn Sie dem in der letzten Anmerkung formulierten Ratschlag folgen, werden Sie sich fragen, wie man es erreichen kann, dass der für eine bestimmte Schnittstelle festgelegte Code teilweise wiederverwendet werden kann.

Da die richtige Verwendung wichtiger ist, als die Kenntnis der einfachen Syntax, will ich etwas ausholen und ein kleines Beispiel beschreiben.

3.5.1 Motivation

Als Szenario wähle ich ein Computerspiel, das ein Gebäude verwendet, in dem die Spielerpersonen durch Türen gehen müssen, um andere Räume zu erreichen. Diese Türen mögen ganz unterschiedliche Eigenschaften haben, sie mögen abschließbar sein oder auch nicht, einige Türen schließen sich wieder automatisch und so weiter.

Um die Software übersichtlich zu halten, empfiehlt es, sich von der folgenden Schnittstelle auszugehen:

```
public interface IDoor {
    public void open();
    public void close();
    public boolean isOpen();
}
```

[22] Vergleiche die Literatur zu Entwurfsmustern, wie [Gam1995].

Eine einfachste Implementierung einer Tür könnte wie folgt aussehen:

```
public final class SimpleDoor implements IDoor {
    private open = false;

    public void open() {
        open = true;
    }
    public void close() {
        open = false;
    }
    public boolean isOpen() {
        return open;
    }
}
```

Sie mögen sich nun fragen, was man mit dieser simplen Klasse anfangen soll. Schließlich ist in keiner Weise vorgesehen, dass eine entsprechende graphische Anzeige erfolgt, oder dass zumindest eine einfache Meldung ausgegeben wird. Ich habe aber die Klasse nicht nur aus didaktischen Gründen so kurz gehalten. Vielmehr will ich mich bewusst auf das abstrakte Modell einer Tür beschränken, ohne mich schon zu früh auf bestimmte Ausprägungen und graphische Darstellungen festzulegen. Vielleicht überzeugt Sie das Argument, dass ich so eine größere Wiederverwendbarkeit erreiche. Zum Beispiel muss eine selbstschließende Tür durch ein etwas anderes Modell beschrieben werden. Ihre Graphik wird aber vielleicht genauso wie die der einfachen Tür aussehen.[23]

Ich greife jetzt den Gedanken auf, dass wir mitkriegen wollen, wenn die Tür sich öffnet oder schließt. Wir wollen in unserem Programm aber nicht gezwungen sein, immer wieder alle Türen abzufragen, sondern wir wollen einfach eine Nachricht bekommen, wenn sich an der Tür was ändert.

Was ich im Folgenden beschreibe, entspricht dem Beobachter-Muster (*observer pattern*). In dem Modell gibt es zwei Rollen, nämlich den Beobachter (Schnittstelle `IObserver`) und das beobachtete Objekte (Schnittstelle `ISubject`). In einfachster Form sehen die beiden Schnittstellen wie folgt aus:

```
public interface IObserver {
    /**
     * Diese Methode wird vom Subjekt bei jeder Aenderung
        aufgerufen.
     *
     * @param subject der Ausloeser der Nachricht.
     */
    public void update(ISubject subject);
}

public interface ISubject {
    /**
     * Hiermit kann sich ein Beobachter anmelden.
     * Er wird in der Folge, durch Aufruf von
     * update() benachrichtigt.
     *
     * @param observer
```

[23] Dieses Vorgehen steht, etwas im Widerspruch zum Yagni-Prinzip („Your Aren't Gonna Need It"). Danach implementiert man zunächst nur die einfachste Methode, die funktioniert, und implementiert erst bei Bedarf eine komplexere aber flexiblere Struktur. Hier will ich aber gerade andeuten, wie man zu einer flexiblen Struktur kommt.

```
     */
    public void register(IObserver observer);
}
```

Um die Funktionsweise zu verdeutlichen, gebe ich eine einfache Anwendung an, die eine Tür steuert, aber dann auch Meldungen ausgibt.

```
class DoorObserver implements IObserver {
    public void update(ISubject subject) {
        IDoor door = (IDoor)subject;
        if (door.isOpen())
            System.out.println("Die Tuer wurde geoeffnet");
        else
            System.out.println(
                "Die Tuer wurde geschlossen");
    }

    public static void main(String[] args){
        DoorObserver b = new DoorObserver();
        ObservableDoor d =
            new ObservableDoor(new SimpleDoor());
        d.register(b);
        d.open();
        d.open();       // bewirkt nichts
        d.close();
        d.close();      // bewirkt nichts
    }
}
```

In dem Beispiel ist vorausgesetzt, dass wir bereits eine Klasse `ObservableDoor` besitzen. Diese Klasse soll sich wie eine normale Tür verhalten, nur, dass jede Änderung an die registrierten Beobachter gemeldet ist. Die Klasse `ObservableDoor` „weiß“ bewusst nicht, wie eine Tür geöffnet oder geschlossen wird. Ich will mich hier noch nicht festlegen. Die konkret verwendete Tür wird daher als Parameter im Konstruktor übergeben. Für den einfachen Testfall reicht die primitive `SimpleDoor`. In der Anwendung habe ich vorausgesetzt, dass das mehrfache Öffnen einer bereits offenen Tür nichts bewirkt.

Wie implementieren wir nun die Klasse `ObservableDoor`? `ObservableDoor` soll wie jede andere Tür anwendbar sein, muss also die Schnittstelle `IDoor` implementieren, als beobachtbares Subjekt aber auch die Schnittstelle `ISubject`.

```
public final class ObservableDoor
        implements IDoor, ISubject {

    private IDoor door;

    public ObservableDoor(IDoor door) {
        this.door = door;
    }

    public void open() {
        door.open();
        // benachrichtigen aller Beobachter
    }

    public void close() {
        door.close();
```

```
        // benachrichtigen aller Beobachter
    }

    public boolean isOpen() {
        return door.isOpen();
    }

    public void register(IObserver observer) {
        // observer registrieren
    }
}
```

Alle Operationen für die Tür sind so implementiert, dass für jede Operation die Methode eines anderern, nämlich des im Konstruktor angegebenen Tür-Objekts aufgerufen wird. Man spricht in diesem Zusammenhang auch von *Komposition* oder von *Delegation*. Speziell das Muster dieses Beispiel kann man auf einer höheren Abstraktionsebene auch als *Dekoration* bezeichnen: Die hinzugeschaltete Klasse `ObservableDoor` „dekoriert“ andere Türen in dem Sinne als alle ihre Aktionen an die Beobachter gemeldet werden. Komposition ist neben der Vererbung das andere Idiom zu Wiederverwendung von Programmcode. Es hat unter anderm den Vorteil, dass es besser dem Geheimnisprinzip entspricht (es verwendet nur öffentliche Methoden). Außerdem ist es in der Anwendung flexibler als die Vererbung (bei der Vererbung hätten wir nur eine Art von Tür beobachtbar gemacht).

Die besonderen Methoden, die das Konzept des beobachteten Subjekts ausmachen, habe ich jedoch noch nicht implementiert. Der Grund ist folgender: Die Möglichkeit beobachtet zu werden, ist nicht auf Türen beschränkt, es handelt sich um ein ganz allgemeines Konzept. Es wäre daher sowohl konzeptionell als auch vom Schreibaufwand her unsinnig, diese Funktionalität speziell für die Tür zu implementieren.

Eine Lösung besteht darin, dass wir eine abstrakte Klasse definieren, die alle Eigenschaften eines beobachtbaren Objekts festlegt. Warum abstrakt? Weil es sich bei dem Begriff „beobachtbar“ um einen abstrakten Begriff handelt! Die Aufgabe der Klasse soll nur darin bestehen, alle beobachtbaren Objekte auf gleichartige Art und Weise zu unterstützen.

3.5.2 Aufgaben und Eigenschaften abstrakter Klassen

Das letzte Beispiel zeigt einen der häufigsten Anwendungen von Abstrakten Klassen. Wir benötigen nämlich für unterschiedliche verwandte Klassen eine Implementierung, die einen bestimmten gemeinsamen Teilaspekt realisiert, so dass sich eine geeignete Modularisierung anbietet.

Merksatz:

> ***Eine Klasse soll ein bestimmtes, eng umrissenes Konzept ausdrücken**. Wenn, mehrere verschiedene Konzepte auftreten (Tür, beobachtbar), sollten diese Funktionalitäten auf zwei verschiedene Klassen verteilt werden.*

Hier folgt also jetzt die vollständige Definition der Klasse `AbstractSubject`[24].

```
import java.util.ArrayList;

public abstract class AbstractSubject implements ISubject {
```

[24] In der Java-Klassenbibliothek, gibt es die ähnlich definierte abstrakte Klasse `Observable` und die Schnittstelle `Observer`

```
    private ArrayList<IObserver> clients =
        new ArrayList<IObserver();

    public final void register(IObserver client) {
        clients.add(client);
    }

    /**
     * Diese Methode muss von der Unterklasse
     * bei jeder Zustandsaenderung aufgerufen werden.
     */
    protected final void fireSubjectChanged() {
        for (IObserver client : clients)
            client.update(this);
    }
}
```

Beachten Sie, dass einerseits mittels `final` ausgedrückt wird, dass `register` und `fireSubjectChanged` nicht überschrieben werden sollen und, dass mit `protected` ausgesagt wird, dass eine abgeleitete Klasse `fireSubjectChanged` eventuell aufrufen sollte.

Nach dieser Vorbereitung können wir die Implementierung von `ObservableDoor` abschließen.

```
public final class ObservableDoor
    extends AbstractSubject implements IDoor
{
    private IDoor door;

    public ObservableDoor(IDoor door) {
        this.door = door;
    }

    public void open() {
        if (!door.isOpen() {
            door.open();
            fireSubjectChanged();
        }
    }

    public void close() {
        if (door.isOpen() {
            door.close();
            fireSubjectChanged();
        }
    }

    public boolean isOpen() {
        return door.isOpen();
    }
}
```

Zusammenfassend kann man die „Dienstleistung“ einer abstrakten Klasse wie folgt darstellen:

Definition:

*Eine **abstrakte Klasse**, definiert notwendige Methoden für alle abgeleiteten Klassen. Zusätzlich kann sie Methoden definieren, die von den abgeleiteten Klassen auf-*

gerufen werden sollen. Diese Methoden sind in der Regel als `protected` *deklariert. Wenn die vordefinierten Methoden nicht mehr überschrieben werden sollen, sollten sie als* `final` *gekennzeichnet sein.*

Die abstrakte Klasse kann auch ***abstrakte Methoden*** *definieren, die dann von den abgeleiteten Klassen zu überschreiben sind. Eine Klasse, die einer abstrakten Klassen abgeleitet ist, und nicht alle abstrakten Methode*[25] *vollständig implementiert, muss als abstrakte Klasse deklariert sein.*

Von abstrakten Klassen kann man keine Objekte erzeugen.

Zur Klarstellung sei der Begriff einer konkreten Klasse nochmals definiert.

Definition:

Eine ***konkrete Klasse*** *ist eine Klasse, die ohne den Zusatz* `abstract` *definiert ist. Eine konkrete Klasse muss alle Operationen implementieren, die von den implementierten Schnittstellen von abstrakten Oberklassen gefordert sind, sofern diese nicht schon bereits in einer Oberklasse implementiert wurden.*

Eine konkrete Klasse gilt (in der Regel) dazu, Objekte zu erzeugen.

Der Begriff der *abstrakten Methode* bedarf noch einer kurzen Erläuterung. Sie kennen zwar das Prinzip schon, dass nämlich in einem Interface *nur* abstrakte Methoden stehen. In einer abstrakten Klassen müssen diese Methoden jedoch ausdrücklich gekennzeichnet werden.

Definition:

Eine ***abstrakte Methode*** *enthält einen Methodenkopf dem anstelle des Methodenkörpers ein Semikolon folgt. Abstrakte Methoden stehen nur in Interface-Einheiten und in abstrakten Klassen. In abstrakten Klassen muss der Kopf durch das Schlüsselwort* `abstract` *gekennzeichnet sein.*

Da eine abstrakte Methoden aufgrund der fehlenden Implementierung eigentlich keine Methode darstellt, spreche ich lieber von der Deklaration einer ***Operation****.*

Wie gesagt, *muss* eine Klasse abstrakt sein, wenn sie selbst eine abstrakte Methode enthält, oder wenn eine Oberklasse oder eine implementierte Schnittstelle eine abstrakte Methode definiert, für die es keine Implementierung gibt. Dabei ist zu berücksichtigen, dass alle Vererbungsaktionen transitiv sind, d.h. ein Objekt kennt alle Variablen und Methoden der Oberklasse und deren Oberklasse usw.

Das folgende Beispiel verdeutlich die Verwendung von abstrakten Methoden an Hand der Volumenberechnung von Rotationskörpern. Das Volumen ist gleich $\pi \int_{x_0}^{x_1} f(x)^2 dx$. Diese Information können wir wie folgt in einer abstrakten Klasse darstellen. Das Integral wird dabei durch eine einfache Trapezregel implementiert:

```
public abstract class AbstractRotationalBody {
    private final double x0, x1, dx;
    private final int nSteps;

    protected AbstractRotationalBody(double x0, double x1,
            int nSteps) {
```

[25] Abstrakte Methoden sind entweder in eienr abstrakten Klasse deklariert, oder durch ein Interface vorgegeben.

```
        this.x0 = x0;
        this.x1 = x1;
        this.nSteps = nSteps;
        dx = (x1 - x0) / nSteps;
    }

    public double volume() {
        double v = 0.5 * (contour(x0) + contour(x1));
        for (int i = 1; i < nSteps; i++) {
            double y = contour(x0 + i * dx);
            v += y * y;
        }
        return Math.PI * v * dx;
    }

    protected abstract double contour(double x);
}
```

Die abstrakte Methode `contour` ist nötig, damit man die Methode `volume` überhaupt formulieren kann.

Um das Beispiel zu vervollständigen, will ich eine abgeleitete Klasse angeben. Ihrer Phantasie sind keine Grenzen gesetzt, wenn Sie weitere Rotationskörper implementieren wollen.

```
public class Sphere extends AbstractRotationalBody {
    private final double radius;

    public Sphere(double radius) {
        super(-radius, radius, 10000);
        this.radius = radius;
    }

    public double contour(double x) {
        return Math.sqrt(radius * radius - x * x);
    }
}
```

Das sogenannte *Collection-Framework* der Klassenbibliothek definiert eine Reihe von Schnittstellen und Klassen, die Ansammlungen von Daten beschreiben. Ein Beispiel ist die Klasse `ArrayList`. Joshua Bloch, der Autor dieser Klassen, hat große Mühe darauf verwandt, diese Klassen so zu gestalten, dass der Benutzer der Bibliothek maximale Flexibilität in ihrer Verwendung erhält. Dabei hat er intensiv und gut überschaubar Schnittstellen, abstrakte Klassen und Delegation eingesetzt. Die Vererbungshierarchie der Klasse `ArrayList` sieht (vereinfacht) so aus:

```
interface Collection
    interface List
        abstract class AbstractList
            class ArrayList
```

`Collection` ist hier die allgemeinste Schnittstelle. Sie definiert Operationen, die auf Datenbehältern selbst dann sinnvoll sind, wenn keine bestimmte Reihenfolge der Inhalte festgelegt ist. `List` fügt dem weitere Operationen hinzu, die bei einer festen Reihenfolge von Objekten sinnvoll sind. Die abstrakte Klasse `AbstractList` beschreibt alles, was von der konkreten Implementierung des Datenbehälters nicht abhängt. Die Klasse

`ArrayList` speichert (wie ihr Name sagt) die Daten in einem Array. Die als Alternative vorhandene Klasse `LinkedList` speichert die Daten als verkette Liste[26].

3.6 Weitere Klasseneigenschaften

In diesem Abschnitt werden die statischen Klassenelemente und die beim Programmieren graphischer Oberflächen häufig verwendeten geschachtelten Klassen besprochen.

3.6.1 Statische Klassenelemente

Definition:

*Unter **statischen Klassenelementen** (vergleiche 1. Semester) verstehen wir Variablen und Funktionen („statische Methoden"), deren Deklaration mit dem Zusatz* `static` *versehen ist. Statische Elemente gehören nicht zu einem Objekt, sondern zur Klasse. Sie sind grundsätzlich über den Namen der Klasse anzusprechen. Innerhalb der definierenden Klasse kann der Klassenname entfallen. Statische Klassenelemente gehorchen den üblichen Sichtbarkeitsregeln.*

Da die abgeleitete Klasse eine Erweiterung der Oberklasse darstellt, können die Elemente der Oberklasse auch über den Namen der Unterklasse angesprochen werden. Trotzdem sind diese Elemente nur ein einziges Mal vorhanden. Wenn eine Unterklasse ein Element gleichen Namens (bei Funktionen zusätzlich gleicher Signatur) wie ein Element der Oberklasse definiert, so wird das Element der Oberklasse *verdeckt*.

Im Grunde genommen passiert mit den statischen Klassenelementen bei der Vererbung nichts Besonderes. Es geht hierbei nicht um Objektorientierung. Statische Elemente können nicht in einer Schnittstelle definiert werden. Eine Ausnahme ist nur die Definiton von Konstanten (`static final`).

Das folgende Beispiel zeigt ein paar typische Situationen.

```
class Oberklasse {
    static int var1 = 5;
    static int getVar1() { return var1; }
}

class Unterklasse extends Oberklasse {
    static int var1 = 6;
    static int getSumme() {
        return var1 + Oberklasse.getVar1();
    }
}
```

In diesem Beispiel haben wir zwei verschiedene Variable und zwei verschiedene Funktionen. Sie können mit den folgenden eindeutigen Namen angesprochen werden:

```
Oberklasse.var1
Oberklasse.getVar1() == Unterklasse.getVar1()
Unterklasse.var1
Unterklasse.getSumme()
```

[26] Das Konzept der verketteten Liste werden Sie später kennenlernen

Innerhalb einer Klasse kann der Name der Klasse weggelassen werden. Der Name der Oberklasse kann vor einem ererbten Element weggelassen werden, wenn dieses nicht verdeckt ist. In dem Beispiel ist die Variable `var1` in der Unterklasse verdeckt.

Da statische Elemente nichts mit Objekten zu tun haben, verfügen sie nicht über die `this`-Referenz. Es ist sehr schlechter (und zu Fehlern führender) Stil statische Elemente über Objektreferenzen anzusprechen.

Statische Funktionen sind das Gegenstück zu den Funktionen von C. Wenn eine Klasse *nur* statische Funktionen enthält, spielt sie die Rolle eines Moduls, das mehrere Funktionen zu einer Einheit zusammenfasst. In diesem Fall sollte die Erzeugung von Objekten und die Ableitung einer Klasse ausdrücklich verboten werden. Ein Beispiel für eine solche Klasse ist `java.lang.Math`. Die Deklaration sieht etwa so aus:

```
package java.lang;

public final class Math { // keine Ableitung !
    private Math() { }    // keine Objekte

    public static native double sin(double x);
    ...
}
```

An dieser Stelle möchte ich auch den *Static-Block* zur Initialisierung statischer Variabler oder allgemein zum Ausführen von Anweisungen, die beim Laden einer Klasse ausgeführt werden sollen, angeben:

Static-Block : `static`{*Anweisungen* }

3.6.2 Geschachtelte Klassen

Viele Datemstrukturen und Klassen benötigen für ihre Implementierung Hilfsklassen, die außerhalb nicht sichtbar sein sollen. Natürlich kann man dieses Problem mit dem Pakektkonzept ganz gut lösen. Bei kleinen Hilfsklassen bietet Java aber eine noch einfachere Lösung. Java erlaubt nämlich, dass man eine Klasse innerhalb einer anderen Klasse deklariert. In der in diesem Abschnitt beschriebenen Form bedeutet das nur, dass diese *geschachtelte Klasse* (englisch: *nested class*) zum Namensraum der umfassenden Klasse gehört.

```
class UmfassendeKlasse {
    ...
    static class Geschachtelte Klasse {
       ...
    }
    ...
}
```

Das Schlüsselwort `static` ist hier wichtig. Es bedeutet, dass die geschachtelte Klasse zwar in der umfassenden Klasse liegt, aber nicht unmittelbar auf die Instanzvariablen der Objekte der umfassenden Klasse zugreifen kann (im Unterschied zu dem Konzept des nächsten Abschnitts).

Innerhalb von `UmfassendeKlasse` kann die `GeschachtelteKlasse` ganz normal mit ihrem Namen angesprochen werden. Außerhalb geht das jedoch nicht, da heißt sie `UmfassendeKlasse.GeschachtelteKlasse`, wie man das bei Komponenten einer Klasse erwartet.

Wie bei anderen Komponenten kann auch bei geschachtelten Klassen die äußere Sichtbarkeit eingeschränkt (oder erweitert werden (`private`, `protected`, `public`). Die Sichtbarkeitseinschränkungen beziehen sich nicht auf die umfassende Klasse, sondern immer nur auf die „Außenwelt“. Die Methoden der umfassenden Klasse haben also immer uneingeschränkten Zugriff auf die Methoden und Attribute der geschachtelten Klasse. Das ist ja auch sinnvoll.

Man darf zwar in geschachtelten Klassen nicht auf Komponenten eines Objekts der umfassenden Klasse zugreifen, aber der Zugriff auf `statische` Komponenten der Klasse ist erlaubt (die brauchen ja kein Objekt).

3.6.3 Innere Klassen

Mit dem Begriff *innere Klasse* (englisch: *inner class*) ist eine geschachtelte Klasse gemeint, die auch auf die Variablen des zugehörigen Objekts der umfassenden Klasse, bzw. wenn sie in einer Methode deklariert ist, sogar auf die lokalen Variablen der Methode zugreifen kann. In der inneren Klasse angesprochene lokale Variable müssen allerdings als `final` deklariert sein. Hinsichtlich der Feinheiten von inneren Klassen muss ich Sie auf die Java-Sprachdefinition verweisen.

```
class UmfassendeKlasse {
    private Object abc;
    ...
    private class Inner {
        ...
        Object getAbc() {
            return abc:
        }
    }
}
```

3.6.4 Anonyme Klassen

Oft ist eine Klassen nötig, um eine einzige Methode zu definieren, die bei einer bestimmten Aktionen aufgerufen werden soll. Mit einer solchen Methodendefinition ist ein nicht unerheblich syntaktischer Ballast verbunden.

Das folgende Beispiel zeigt wie man in der Java-Schnittstelle einen graphischen Ende-Knopf (Quit-Button) implementieren kann. Zunächst in Form einer geschachtelten Klasse:

```
...
static class Quit implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
}

public void init() {
```

```
    ...
    JButton q = new JButton("quit");
    q.addActionListener(new Quit());
    frame.add(q);
    ...
}
...
```

Die Logik dieses Programmstücks ist, dass mittels `addActionListener` zunächst dem Objekt des Ende-Knopfes ein Objekt übergeben wird, das für die Ausführung der Knopf-Aktion zuständig ist. Dadurch, dass die Klasse `JButton` weiß, dass dieses Objekt vom Typ `ActionListener` ist, weiß der Endeknopf, dass er, wenn der gedrückt wird, `actionPerformed` aufgerufen kann.

So wie es dargestellt ist, funktioniert die Methode und so wird das auch manchmal programmiert. Wenn man aber viele von solchen „kleinen“ Klassen hat, so kann das doch ganz schön umständlich sein. Java bietet deshalb die Möglichkeit einfach ein Objekt zu erzeugen ohne einen Namen für dessen Klasse zu deklarieren. Da die Klasse nur ein einziges Mal bei der Erzeugung des Listener-Objekts angesprochen wird, ist ein Name auch nicht nötig. Allerdings muss bei der Beschreibung dieser *anonymen Klasse* ein Typ angegeben werden. Dies kann durch den Namen einer Schnittstelle oder einer Superklasse geschehen.

Definition:

> *Eine **anonyme Klasse** ist eine innere Klasse ohne Namen. Eine anonyme Klasse dient nur der Erzeugung von Objekten. Der Typ dieser Objekte ist die die Angabe der Oberklasse (oder der einer Schnittstelle definiert, Häufig werden innere Klasse in Java verwendet, wenn man Funktionsobjekte erzeugen will, die nur dazu da sind eine bestimmte Funktion auszuführen.*

Das Beispiel soll hier reichen. Genauere Details müssen Sie bei Bedarf in der Java-Dokumentation nachlesen:

```
    ...
    q.addActionListener(
        new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.exit(0);
            }} );
    frame.add(q);
    ...
```

Auch wenn es in der Regel so wie hier aussieht, so ist es durchaus erlaubt, innerhalb der anonymen Klasse mehr als eine Methode und auch Instanzvariable zu deklarieren.

3.7 Zusammenfassung

3.7.1 Polymorphie und späte Bindung

Wir haben verschiedene Möglichkeiten diskutiert, mit denen sich die Wiederverwendung fördern lässt:

Interface. Ein Interface definiert einen statischen Typ. Damit können wir Datenstrukturen und Methoden definieren, die Objekte unterschiedlichen Typs speichern und verarbeiten können, solange nur der Typ des Objekts ein Untertyp des Schnittstellentyps ist ist. Die Ableitungsbeziehung von Schnittstellen erlaubt uns, Konzepte unterschiedlichen Abstraktionsniveaus zu bilden und zu verwalten.

Klasse. Auch Klassen definieren Schnittstellen und auch für Klassen gibt es Ableitungen, mit denen wir eine konzeptionelle Hierarchie aufbauen können. Diese Möglichkeit ist bei Klassen jedoch nicht so ausgeprägt wie bei Schnittstellen, da Klassen nur einfache Vererbung erlauben. Die Hauptaufgabe einer Klasse besteht in der Beschreibung und Erzeugung von Objekten. Die Ableitung von Klassen ist neben der Spezialisierung eines Typs gleichzeitig eine Wiederverwendung von vorhandener Implementierung.

Abstrakte Klasse. Eine abstrakte Klasse gibt für konkrete Klassen eine gemeinsame Teilimplementierung vor. Auch abstrakte Klassen beschreiben einen Typ. In der Regel empfiehlt es sich jedoch hier, diesen Typ durch ein Interface auszudrücken (auch abstrakte Klassen werden nur einfach vererbt). In guten Klassenhierarchien (z.B. dem Collection-Framework) ergänzen abstrakten Klassen die jeweiligen Schnittstellen, indem sie die Implementierung der Schnittstelle vereinfachen.

Delegation. Wiederverwendung wird auch erreicht, indem man Aufgaben an andere Objekte delegiert. Häufig hat die aufrufende Klasse (wie in dem Beispiel der Türen) dieselbe Schnittstelle wie die aufgerufene Klasse. Diese Situation entspricht in der Regel dem Entwurfsmuster *Dekoration*. Die aufrufende Klasse kann jedoch auch eine ganz andere Schnittstelle haben, d.h. die Delegation erfordert keine Typverwandschaft der beteiligten Klassen. Im Unterschied zur Vererbung von Klassen kann die Klassenbeziehung bei der Delegation sogar dynamisch konfiguriert werden (etwa über Parameter des Konstruktors).

Die automatische Wiederverwendung des Codes durch Ableitung einer Klasse ist ein Plus der Objektorientierung. Der eigentliche Vorteil der Objektorientierung besteht aber in der *Polymorphie*.

Definition:

*Unter **Polymorphie** (griechisch: poly-morphein = viele Formen, Vielfalt) versteht man die Möglichkeit, Objekte unterschiedlicher Art gleichartig anzusprechen.*

Die Definition fasst einige Fälle von Polymorphie zusammen.

- Eine Variable eines Typs T kann Objekte von allen Untertypen von T speichern.
- Der Aufruf einer Operation ist für eine Objektreferenz möglich, wenn die Signatur durch den statischen Typ der Referenz erlaubt ist. Die Ausführung geschieht durch die Methode, die in der Klasse des Objekts vorgesehen ist.
- Manchmal versteht man unter Polymorphie auch die Tatsache, dass verschiedene Methoden einer Klasse denselben Namen tragen dürfen, solange sich ihre Signatur unterscheidet (statische Polymorphie). Diese Art von Polymorphie ist jedoch keine Besonderheit der Objektorientierung.

Sicher haben Sie sich schon einmal gefragt, wie ein C-Compiler es zustande bringt einzelne Funktionsaufrufe richtig auszuführen. Genau genommen wird diese Aufgabe durch den letzten Schritt der Compilierung eines C-Programms, durch den *Linker*, ausgeführt. Der Linker fasst die verschiedenen Teile eines Programms zu einer ausführbaren Einheit zusammen und stellt eine Verknüpfung zwischen Funktionsaufrufen und Funktionen her, indem er an der Stelle des Aufrufs die Funktionsadresse einträgt. Das Verbinden von Aufruf und Funktion nennt man *Bindung* (*binding*). Den eben beschriebenen Fall nennt man *frühe* Bindung.

In Java ist die Situation in mehrfacher Hinsicht anders. Zunächst gibt es den separaten Linker nicht. Java sieht nämlich vor, dass Klassen erst zur Laufzeit, bei Bedarf, geladen werden. Dafür ist ein besonderes Objekt, der sogenannte Klassenlader (*class loader*) zuständig. Dieses Objekt überträgt den eingelesenen Bytecode in eine interne Form, führt Sicherheitsprüfungen durch und bindet soweit wie möglich Klassen und Methoden.

Die vollständige Bindung von Aufruf zu Methode gelingt in Java aber nur in wenigen Fällen, nämlich nur dann wenn diese Beziehung absolut feststeht. Dies gilt z.B. für den Aufruf von statischen Funktione. Der Aufruf einer statischen Funktionen enthält ja Klassen- und Methodenname und berücksichtigt nur statische Typeigenschaften. Ein weiteres wichtiges Beispiel ist der Aufruf von privaten Methoden. Auch in Java spricht man in diesem Fall von früher Bindung, da die Bindung vor der Ausführung des entsprechenden Aufrufs erfolgt.

Frühe Bindung kann jedoch bei dem Aufruf der Operation eines Objekts nicht immer korrekt vorgenommen werden. Dies liegt daran, dass währender der Übersetzung, also vor der Ausführung des Programms, zwar der *Typ* der Objekte bekannt ist, die eigentlichen *Objekte* und ihre jeweilige *Klasse* jedoch nicht. Es bleibt nichts anderes übrig, als die Entscheidung darüber, welche Methode die Ausführung der Operation übernimmt, bis zum Zeitpunkt des Aufrufs selbst zu verschieben, da erst dann mit Sicherheit das aufgerufene Objekt bekannt ist. Diese aufgeschobene Bindung heißt *späte Bindung* (*late binding*).

Der Java-Compiler veranlasst frühe Bindung für:

- **Klassenfunktion:** Es wird der Bytecode `invokestatic` erzeugt.
- **Konstruktoren, private Methoden, Super-Aufrufe:** Es wird der Bytecode `invokespecial` erzeugt.

Die Bytecodes der Methodenaufrufe bei später Bindung sind `invokevirtual` und `invokeinterface`. Die Unterschiede zwischen diesen beiden Aufrufen sind rein implementierungsbedingt.[27]

Wie funktioniert die späte Bindung zur Laufzeit? Nun, es gibt verschiedene Implementierungsmöglichkeiten. Für das Verständnis genügt es, wenn wir uns den folgenden Ablauf vorstellen:

1. Der Methodenname, ihre Signatur und die Referenz des Empfängerobjekts stehen fest.
2. Aus der Objektreferenz erhalten wir das Objekt und daraus die Klassenbeschreibung.

[27] Die JVM kann `invokevirtual` einfacher effizient realisieren. Daher *kann* es kleine Performacevorteile bringen, wenn man Variablen eher mit Klassentypen als mit Interfacetypen deklariert. Dies aber etwas der Forderung nach möglichst abstrakter Formulierung eines Programms.

3. In der Klassenbeschreibung suchen wir nach der Methode.
4. Wir führen die Methode aus, nachdem wir ihr die Objektreferenz (wird zu `this`) und die Argumente des Aufrufs übergeben haben.

Naürlich müssen dazu alle nötigen Informationen über die Klasse bereits in ihrem Classfile enthalten sein – sie sind es! Der dritte Schritt wird durch die virtuelle Maschine in der Regel effizienter als durch einfache Suche durchgeführt. Einige Techniken verwenden dazu Methodennummerierungen und Methodentabellen, die die Suche zu einem einfachen indizierten Zugriff werden lassen. Aber das sind Implementierungsdetails der virtuellen Maschine.

Anmerkung:

Die frühe Bindung ist genau genommen eine Optimierung der späten Bindung, die den Suchaufwand zur Laufzeit vermeidet. Moderne virtuelle Maschinen besitzen einen optimierenden Hotspot-Compiler, der auch aus dem Programmkontext erkennen kann, ob frühe Bindung möglich ist und in diesem Fall eventuell sogar den Aufruf vollständig durch den Programmcode der Methode ersetzt (inlining).

3.7.2 Zusammenfassung der Syntax

Auf den folgenden Seiten werden die wichtigsten Konstrukte der Objektorientierung in Java zusammengefasst.

Übersetzungseinheit

Eine Übersetzungeinheit ist eine Datei, die eine oder mehrere Top-Level Klassen und Interfaces enthält. *Eine* Klasse/Interface davon darf als `public` deklariert sein. Der Name der Datei muss gleich dem Namen dieser öffentlichen Klasse/Interface plus der Endung `.java` sein.

Die erste Anweisung einer Übersetzungseinheit ist die optionale **Package-Anweisung**.

Package-Anweisung : `package` *Paketname* ;

Fehlt die Package-Anweisung, so gehört die Klasse zum Defaultpaket. Für ernsthafte Java-Projekte sollte man unbedingt eine sinnvolle Paketstruktur definieren.

Anschließend können mehrere Import-Anweisungen folgen. Die Import-Anweisungen führen Paketnamen für einzelne Klassen oder für alle Klassen des Pakets ein.

Import-Anweisung : `import` *Paketname* `.*`
| `import` *Paketname* `.` *Klasse*
| `import static` *Paktet* `.` *Klasse* `.*`
| `import static` *Paket* `.` *Klasse* `.` *Methode*

Anschließend folgen die Klassen- und Interface-Deklarationen. Top-Level Klassen und Interfaces tragen die Modifikatoren `public` oder Default (Paketsichtbarkeit).

Interface

Interfaces definieren die Operationen und Konstanten einer Schnittstelle. Sie werden verwendet um den statischen Typ von Variablen, Parametern und Rückgabewerten zu deklarieren.

Interface-Deklaration :
Sichtbarkeitsangabe `interface` *Name*
`(extends` *Liste von Interface-Namen* `)?`
{
Deklaration von Konstanten
Deklaration von abstrakten Methoden
}

Ein Interface definiert abstrakte Methoden und Konstanten. Ein Interface kann vorhandene Interfaces erweitern (`extends`). Dadurch gehören alle Methoden und Konstanten der angegebenen Interfaces mit zu dem neuen Interface. Gleichzeitig definiert die Erweiterung von Interfaces eine Typbeziehung. Wenn T_2 eine Erweiterung der Schnittstelle T_1 ist, so kann jeder Ausdruck vem Typ T_2 in einer Variablen vom Typ T_1 gespeichert werden.

Mit der Deklaration von Konstanten sollte man sehr zurückhaltend umgehen. Es ist schlechter Stil, Interfaces zu reinen Konstantensammlungen zu degradieren. [28]

Die Syntax der Deklaration von *Konstanten* und *abstrakten Methoden* folgt weiter unten.

Abstrakte Klasse

Abstrakte Klassen definieren einen Typ und Methoden und Variable, die im Allgemeinen für die Implementierung des Typs benötigt werden. Methoden, die jede konkrete Klasse definieren soll (z.B. weil sich sonst die Methoden der abstrakten Klasse nicht sinnvoll definieren lassen), werden als *abstrakt* deklariert. Von einem Interface unterscheidet sich eine abstrakte Klasse dadurch, dass sie praktisch wie eine Klasse deklariert ist; von einer Klasse unterscheidet sie sich dadurch, dass sie abstrakte Methoden enthalten kann (bzw. nicht alle Methoden einer implementierten Schnittstelle oder erweiterten abstrakten Klasse implementieren muss).

Abstrakte Klasse :
Sichtbarkeitsangabe `abstract class` *Name*
`(extends` *Oberklasse* `)?`
`(implements` *Liste von Interface-Namen* `)?`
{
alles was in einer Klasse stehen darf
Deklaration von abstrakten Methodenn
}

[28] Oft verwendet man zur Definition von Konstanten besser Enum-Klassen.

Abstrakte Methode

Eine *abstrakte Methode* definiert den Namen einer Operation und ihre Signatur.[29]

Abstrakte Methode :
Sichtbarkeitsangabe `abstract` *Typ Name*
`(` (*Parameterliste*)`?)` `;`

In einer abstrakten Klasse dient dass Schlüsselwort `abstract` zur Unterscheidung von „normalen“ Methoden. In einem Interface ist diese Unterscheidung nicht nötig. Man lässt `abstract` dort in der Regel weg.

Klasse

Eine *Klasse* definiert den Typ und die Implementierung einer Menge von Objekten. Als Ausnahme gibt es auch Klassen, die nur statische Elemente enthalten.

Klasse :
Sichtbarkeitsangabe `(final)?class` *Name*
`(extends` *Oberklasse* `)?`
`(implements` *Liste von Interface-Namen* `)?`
{
Klassenvariable
Klassenfunktionen
statische Blöcke
Instanzvariable
Instanzmethoden
Konstruktoren
Geschachtelte Klassen
}

Das Schlüsselwort `final` verbietet die weitere Ableitung von einer Klasse.

Statische Deklarationen

Die Deklarationen, die sich auf die Klasse (das Klassenobjekt) und nicht auf die von der Klasse erzeugten Objekte beziehen, sind mit dem Schlüsselwort `static` gekennzeichnet. Das folgende Beispiel listet alle Möglichkeiten auf:

```
static final int
    ANSWER_TO_EVERYTHING = 42;      // Konstante
static int laufendeNr;              // Klassenvariable
```

[29] Der Begriff *Methode* ist hier irreführend, da man mit einer Methode die Art der Ausführung durch ein Objekt meint. Genauer wäre wohl *Deklaration einer Operation*. Das klingt aber vielleicht etwas zu kompliziert.

```
static int getLaufendeNr() { ... }  // Klassenfunktion
static {                            // Block
    laufendeNr = 7;
}
```

Die statischen Blöcke enthalten Anweisungen, die beim Laden der Klasse ausgeführt werden. Sie dienen in der Regel der Initialisierung von statischen Variablen. Da, wo es möglich ist, bevorzugt man natürlich die Initialisierung durch direkte Zuweisung (diese wird ebenfalls beim Laden der Klasse ausgeführt).

Statische Blöcke verfügen über keine This-Referenz.

Ein Sonderfall ist die Deklaration von (absoluten) Konstanten mittels `static final`. Das Auftreten dieser Konstanten wird durch den Compiler direkt durch den Wert ersetzt.

Klasseneigenschaften gelten auch in Unterklassen. Dies sieht so aus wie Vererbung. Klasseneigenschaften können jedoch höchstens *verdeckt* aber nicht überschrieben werden. Sie können weder als `final` noch als `abstract` deklariert werden. Wenn man Klassenelemente anspricht, sollte man dies immer ausdrücklich über den Namen der Klasse tun. Schließlich haben Klassenelemente nichts mit der Objektorientierung zu tun!

Eine geschachtelte Klasse, die `static` deklariert ist, ist zwar in den Namensraum der umgebenden Klasse eingebettet, hat aber keinen besonderen Bezug zu den Objekten der Klasse.

Objekteigenschaften

Die nicht-statischen Deklarationen einer Klasse beziehen sich alle auf Objekte.

- Der **Konstruktor** initialisiert ein neues Objekt. Konstruktoren werden nicht vererbt. Wenn es keinen Konstruktor gibt, wird automatisch der Defaultkonstruktor erzeugt. Jeder Konstruktor muss als erste Anweisung den Konstruktor der Oberklasse aufrufen. Wenn vorhanden, wird eventuell automatisch der Defaultkonstruktor der Oberklasse aufgerufen. Der Aufruf des Konstruktors der Oberklasse erfolgt über `super(...)` der Aufruf eines anderen Konstruktors der Klasse erfolgt über `this(...)`.
- Eine **Variable**, die als `final` deklariert ist, kann nur direkt als Initialisierung oder einmalig im Konstruktor einen Wert erhalten. Sie kann später nicht mehr verändert werden. Das heißt jedoch nicht, dass ein von ihr referiertes Objekt nicht verändert werden kann.
- Eine **Methode**, die als `final` deklariert ist, kann nicht in einer abgeleiteten Klasse überschrieben werden.
- Variable können in abgeleiteten Klassen verdeckt werden. Auf die Variablen der Oberklasse, kann mittels `super.`*Name* zugegriffen werden. Das **Verdecken von Variablen** wird vom Compiler aufgelöst.
- Methoden können in abgeleiteten Klassen überschrieben werden. Auf die Methoden der Oberklasse, kann mittels `super.`*Name* zugegriffen werden. Das **Überschreiben einer Methode** macht die späte Bindung nötig, d.h. unabhängig vom statischen Typ gewährleistet die späte Bindung, dass die richtige Methode des Objekts aufgerufen wird. Die späte Bindung ist mit die wichtigste Eigenschaft einer objektorientierten Programmiersprache.

– Eine **innere Klasse** (d.h. eine nicht-statisch geschachtelte Klasse) kann sich auf die Eigenschaften eines Objekts der umgebenden Klasse beziehen. Mehrdeutigkeiten können durch *Klassenname*`.this` vermieden werden.

– Innere Klassen können auch innerhalb einer Methode deklariert werden. Der wichtigste Anwendungsfall ist die `anonyme Klasse`. Es können nur solche Variable der umgebenden Methode referiert werden, die als `final` deklariert sind.

Anonyme Klasse

Anonyme Klassen werden dann verwendet, wenn ein Objekt nur für die Ausführung einer einzigen Operation gebraucht wird, z.B. ein `Comparator`, der die von einer Sortierfunktion benötigte Vergleichsfunktion `compare` definiert.

Die anonyme Klasse hat keinen Namen. Sie wird nur bei der Erzeugung eines Objekts aufgeführt. Ihre Methoden implementieren oder überschreiben Methoden einer Schnittstelle oder Oberklasse.

Erzeugung eines Objekts einer anonymen Klasse:
`new` *Oberklasse oder Interface* `(...)`
Klassenkörper

Hinter `new` steht der Name der Schnittstelle, die implementiert wird oder der Name der Klasse von der abgeleitet wird. In den danach folgenden Klammern stehen eventuell Argumente für den Konstruktor der Oberklasse.

Das folgende Beispiel zeigt die absteigende Sortierung eines Arrays von Strings.

```
String[] feld = { "Hallo", ... };
Comparator<String> cmp = new Comparator<String>() {
    public int compare(String a, String b) {
        return b.compareTo(a);
    }};
Arrays.sort(feld, cmp) {
```

Die anonyme Klasse ist ein Beispiel, in dem *nur* der Name der Schnittstelle, aber nicht der Name der Klasse bekannt ist.

Oft lässt man bei der Verwendung der anonymen Klasse sogar die Objektvariable weg. Dann sieht das Beispiel wie folgt aus.[30]

```
String[] feld = { "Hallo", ... };
Arrays.sort(feld, new Comparator<String>() {
    public int compare(String a, String b) {
        return b.compareTo(a);
    }});
```

[30] Das Beispiel sieht immer noch etwas kompliziert aus. Die Java-Entwickler wollen in den nächsten Jahren eine mächtigere und gefälligere Variante einführen.

Kapitel 4

Programmierregeln

... we can expect good programming style to remain
the combination of sound principles, talent and work,
and the fight against poor style is never ending.
Peter Naur, 1992[1]

Dieses Kapitel stellt keinen Vorlesungsstoff im engeren Sinne dar. **Das Kapitel ist nicht direkt prüfungsrelevant.**

Trotzdem ist das Kapitel nicht unwichtig. Erst durch Beachtung dieser Regeln lernt man programmieren!

4.1 Grundregeln für den Aufbau einer Klasse

Die Regeln folgen ein paar zentralen Grundprinzipien, wie *Geheimnisprinzip*, *Lesbarkeit* und *Lokalität* (man muss nur dass kennen, was gerade eine Rolle spielt).

1. Überlege zunächst, was die Klasse tun soll! [2] Formuliere diese Erwartungen, indem du ein Testprogramm schreibst (am besten direkt mit JUnit).
2. Entwickle eine Klasse Schritt für Schritt. Teste jeweils den erreichten Stand!
3. Eine Klasse bildet eine Einheit von äußerer Schnittstelle (Typ) und innerer Implementierung.
4. Das Geheimnisprinzip erfordert, dass es keine Operation gibt, die Auskunft über die Art der Implementierung gibt.
5. Eine Klasse sollte genau *ein* Konzept beschreiben. Die Aufgabe der Klasse sollte in einem kurzen Klassenkommentar beschreibbar sein.
6. Instanzvariable sind dazu da, den Zustand eines Objekts zu beschreiben. Mache nicht aus Bequemlichkeit temporäre Variable zu Instanzvariablen!
7. Methoden sollen kurz und verständlich sein.
8. Führe lokale Variable nur da ein, wo sie benötigt werden .

[1] Zitat aus *Java's insecure Parallelism* von Per Brinch Hansen, 1999

[2] Eine gute Methode der Softwaretechnik besteht darin, die Aufgaben einer Klasse auf einem *responsiblity chart* festzuhalten.

9. Verwende aussagekräftige und sinnvolle Namen. Wenn sich die Bedeutung eines Programmobjekts ändert, dann soll man auch seinen Namen entsprechend ändern.

10. Unterstütze die Lesbarkeit durch Schnittstellen-Kommentare und sinnvolle Formatierung.

11. Lösche alles, was nicht gebraucht wird !!!

4.2 Fortgeschrittene Regeln für das Zusammenspiel von Klassen

In der Literatur findet man eine viele von Hinweise zur guten Gestaltung der Softwarearchitektur. Diese Regeln sind auch für den Entwurf einer einfachen Klasse hilfreich. Daher habe ich hier ein paar dieser Regeln zusammengestellt.

Jedes Objekt erfüllt eine bestimmte Aufgabe. Es hat eine Schnittstelle, die beschreibt, welche Operationen es ausführen kann und Methoden, die die Art der Ausführung der Operationen beschreiben. Beides wird festgelegt durch die Klasse, mit der das Objekt erzeugt wird. Klassen können von allgemeineren Oberklassen abgeleitet sein. Zusätzlich gibt es in Java Schnittstellen (`interface`) mit der bestimmte Operationsschnittstellen definiert werden können, ohne dass eine Implementierung angegeben wird. Klassen können sich auf Schnittstellen beziehen (`implements`) und Schnittstellen können vorhandene Schnittstellen erweitern (`extends`).

Schnittstellen und Klassen (im Java-Jargon: Typen) dienen u.a. dazu, Variable zu deklarieren. Durch den Typ wird angegeben, welche Operationen man mit der Variablen ausführen kann und welche Objektreferenzen in der Variablen gespeichert werden können. Ganz allgemein gilt, dass in einer Variablen eines allgemeineren Typs Objekte einer spezielleren Klasse gespeichert werden können.

1. Das *Substitutionsprinzip* von Barbara Liskov besagt, dass überall da wo eine Schnittstelle vom Typ T verwendet wird (z.B. durch eine Variable) ein beliebiges Objekt einer Klasse eingesetzt werden kann, die diese Schnittstelle implementiert. In der Regel sollte der Benutzer nur die Schnittstelle von Objekten kennen müssen, aber nicht deren Klasse. Eine objektorientierte Sprache wie Java bietet die Voraussetzung für die Anwendung dieses Prinzips. Das Substitutionsprinzip ist aber nur dann erfüllt, wenn man sich in der Entwicklung auch konsequent danach richtet. Insbesondere ist es nicht zulässig, dass eine Klasse die Bedeutung von Methoden komplett umdefiniert oder sogar die Menge der verwendbaren Methoden einschränkt. Richtig ist der Satz: „Eine Ente ist ein Vogel, der schwimmen und tauchen kann“. Falsch ist dagegen: „Ein Auto ist ein Vogel, der keine Federn, keine Flügel und keine Beine hat und auch nicht fliegen aber dafür fahren kann“.

2. Das *Prinzip der einfachsten Lösung* (Kent Beck und viele andere): „Man sollte die einfachste Struktur implementieren, die ein gegebenes System löst.“ Die einfachste Struktur ist nicht unbedingt die kürzeste, sondern diejenige die am besten verständlich ist. Eine Konsequenz dieser Regel ist, dass man keine Funktionalität implementiert, die (noch) nicht benötigt und nicht getestet wird.

3. Das *offen-geschlossen-Prinzip (open closed principle)* von Bertrand Meyer besagt: „Softwareeinheiten (Klassen, Module usw.) sollen offen für Erweiterungen aber geschlossen für Veränderungen sein“. Software sollte so strukturiert sein, dass man

im Fall einer Erweiterung nicht die vorhandene Funktionalität ändern muss, sondern nur neue Einheiten hinzuzufügen braucht. Erweiterbarkeit wird in erster Linie durch die Verwendung von allgemeinen Schnittstellen und in zweiter Linie durch die Ableitung von spezialisierten Klassen erreicht.

Die Anwendbarkeit dieser Regel erfordert einen kleinen Zusatz, denn nicht immer ist vorhandene Software so strukturiert, dass sie eine einfache Erweiterung erlaubt. Dann sollte man zunächst die Software so umstrukturieren, dass die Erweiterbarkeit möglich wird, ohne dabei schon ihre Funktionalität zu verändern (*Refactoring*). Erst nachdem man getestet hat, dass alles wie bisher funktioniert, sollte man dann mit der Erweiterung beginnen.

4. Das *Prinzip der umgekehrten Abhängigkeit (dependancy inversion principle)* von Robert Martin besagt, dass „Programmcode einer höheren Ebene nicht von der Codierung einer niedrigeren Ebene abhängen sollte, statt dessen sollten beide von einer gemeinsamen Abstraktion abhängen."

   Die Kenntnis dieses Prinzip ist deshalb wichtig, weil damit den Prinzipien der prozeduralen Programmierung – die Sie ja schon gut kennen – widersprochen wird. In der prozeduralen Programmierung ruft eine allgemeine Funktion, „Unterprogramme" auf, die Teilaufgaben erledigen. Die allgemeine Funktion ist von den Unterprogrammen abhängig. In einem objektorientierten System sollte die höchste, abstrakte Ebene nicht „wissen" wie ihre Aufgaben im Detail erledigt werden. Man erreicht dies dadurch, dass man abstrakte Schnittstellen verwendet. Eine besondere Ausprägung des Prinzips sind *Frameworks* in denen die allgemeine Struktur einer Anwendung bereits als Klassenstruktur vorgefertigt ist, und nur noch anwendungsspezifische Klassen hinzugefügt werden.

Im Grunde sagen alle Regeln mehr oder weniger dasselbe aus: *Man sollte ein Softwaresystem logisch gemäß den verwendeten Konzepten gliedern und man sollte Abhängigkeiten zwischen Klassen möglichst vermeiden oder möglichst schwach gestalten.*

Aus den im Literaturverzeichnis angegeben Büchern möchte ich zwei besonders hervorheben. Das Standarwerk zu Entwurfsmustern [Gam95] gibt konkrete „Bauanleitungen" für verschiedene Probleme. Das Buch „Refactoring" [Fow2000] von Martin Fowler zeigt überzeugend den Weg, wie man vorhandene Software so weiterentwickelt, dass sie mit der Zeit nicht schlechter, sondern besser wird.

4.3 Hinweise zur Optimierung

Es ist nicht einfach allgemein gültige Optimierungsregeln anzugeben, da eine Quelltextoptimierung wissen muss, welche Konstrukte effizient übersetzt und ausgeführt werden und welche nicht. Viele scheinbar ineffiziente Konstrukte werden zur Laufzeit automatisch optimiert. Es ist nur selten möglich im Voraus zu sagen, wieviel eine einzelne Optimierungsmaßnahme wirklich bringt.

Optimierung steht zu einem großen Teil im Widerspruch zu einer guten Programmiertechnik.

1. *Optimiere ein Programm nur, wenn es nötig ist.* Untersuche, für welche Programmteile sich die Optimierung lohnt (Performance-Analyse). Teste die Ergebnisse.
2. *Optimiere zunächst auf algorithmischen Ebene.* Hier ist der mögliche Gewinn mit Abstand am größten!

3. *Benutze Standardbibliotheken.* Standardbibliotheken verwenden meist hochgradig optimierte Algorithmen und Implementierungen. Beachte die Performance-Hinweise der Dokumentation!

4. *Vermeide das unnötige Erzeugen von temporären Objekten.* Ein typisches Beispiel ist die Stringkonkatenierung durch den Operator +. Erheblich günstiger ist die Methode `append` aus der Klasse `StringBuilder`.[3]

5. *Je spezifischer und statischer ein Programm ist, um so effizienter wird es in der Regel ausgeführt.* Allerdings ist es dann gleichzeitig weniger allgemein zu verwenden.

6. *Verwende Exception-Mechanismen nur für wirkliche Ausnahmen.* Das Erzeugen einer Ausnahme ist sehr aufwändig, da dabei ein komplettes Stacktrace generiert werden muss. Ausnahmen sollten niemals zur Steuerung der Programmlogik missbraucht werden.

7. *Der Zugriff auf lokale Variable ist sehr effizient.* Es kann sich lohnen, innerhalb einer Methode eine Instanzvariable temporär in eine lokale Variable zu kopieren.

8. *Betriebssystemaufrufe sind langsam.* Man sollte so programmieren, dass ihre Zahl möglichst gering gehalten wird. Zum Beispiel erreicht man dies bei der Ein- und Ausgabe durch die Verwendung von puffernden Klassen, wie `BufferedReader`.

9. Ein einfaches, oft verwendetes Verfahren der Optimierung besteht darin, einmal ermittelte Ergebnisse, die häufig abgefragt werden, vorzuhalten (caching).

Die Liste kann gar nicht vollständig sein, da viele Regeln von ganz speziellen Bedingungen abhängen. Das Entscheidende ist: *Es ist nicht anzustreben, ein Programm in seine absolut effizienteste Form zu bringen. Dies würde einen nicht vertretbaren Aufwand für zum Teil kaum merkliche Effekte bedeuteten. Es ist vielmehr entscheidend, die wenigen laufzeitkritischen Programmteile herauszufinden und diese gezielt zu optimieren.* Es gibt Werkzeuge (Profiler), die diese Suche unterstützen.

In C führt oft allein, das Anschalten der automatischen Optimierung des Compilers mit `-O2` zu einer Halbierung der Laufzeit. In Java kann man bei lang laufenden Anwendungen oft einen ähnlichen oder sogar noch besseren Effekt durch die Verwendung der VM Option `-server` erreichen.

Der am Ende der Aufzählung aufgeführte Ratschlag, häufig benötigte Ergebnisse zu speichern, kann mitunter zu einer erheblichen Laufzeitverbesserung führen. Probieren Sie einfach einmal das folgende Testprogramm aus, das auf dem ineffizienten naiv rekursiven Fibonacci-Algorithmus beruht:

```
import java.util.Arrays;

public class FibCache {
    private static final long UNKNOWN = -1;
    private static long[] cache = new long[93];
    static {
        cache[0] = 0;
        cache[1] = 1;
        Arrays.fill(cache, 2, cache.length, UNKNOWN);
    }

    /**
```

[3] Achtung: bis 1.4 die Klasse `StringBuffer` verwenden!

```
     * Computes Fibonacci-function.
     *
     * @param n argument (0<= n <= 92)
     * @returns n-th Fibonacci-number
     * @throws IllegalArgumentException if n<0 or n>92
     */
    public static long fib(int n) {
        try {
            if (cache[n] == UNKNOWN)
                cache[n] = fib(n-1) + fib(n-2);
            return cache[n];
        }
        catch (ArrayIndexOutOfBoundsException e) {
            throw new IllegalArgumentException(
                      "illegal: " + n);
        }
    }

    public static void main(String[] args) {
        System.out.println(fib(Integer.parseInt(args[0])));
    }
}
```

Wenn Sie das Programm testen wollen, müssen Sie es einfach wie folgt aufrufen:

```
java FibCache <zahl>
```

Für `<zahl>` können Sie eine Zahl zwischen 0 und 92 einsetzen. Auch wenn das bei einmaliger Ausführung noch nicht der schnellste Algorithmus ist, so ist der Unterschied zur ursprünglichen Version (ohne Zwischenspeicher) doch gigantisch. Die Laufzeit dieses Verfahrens, ist im ungünstigsten Fall $O(n)$. Bei wiederholtem Aufruf sinkt die Laufzeit weiter, da bereits vorher gefundene Ergebnisse, in $O(1)$ gefunden werden.[4]

[4]Die O-Notation wird in 5.3 genauer besprochen. $O(1)$ bedeutet, dass die Laufzeit unabhängig von dem Parameter n ist, $O(n)$ bedeutet, dass die Laufzeit linear mit n ansteigt.

Kapitel 5

Korrekte und effiziente Algorithmen

Es gibt keinen Königsweg zur Mathematik.
Euklid

Das Schreiben von Computerprogrammen ist zwar oft mühsam, wirft auf der anderen Seite aber meist keine besonders schwierigen Fragen auf. Ab und zu stößt man aber doch auf Probleme, die sich nicht ganz so einfach lösen lassen oder man erlebt, dass es jemand anderem mit einer „genialen“ Idee gelingt, Programmabläufe extrem zu beschleunigen. Dabei wird klar, dass man Programmabläufe nicht immer einfach nur so „hinschreiben“ kann, sondern dass der Ablauf vorher genau überlegt werden muss.

Ein Verfahren, das genau die Reihenfolge und die Bedeutung einzelner Schritte eines Programmablaufs beschreibt, nennt man *Algorithmus*.

Der Begriff Algorithmus geht zurück auf den persischen Mathematiker Mohammed al-Khowârizmî (780-850, Baghdad), der Rechenverfahren für den Umgang mit Dezimalzahlen aufschrieb.[1] Als die Methode auch in Europa bekannt wurde, insbesondere auch durch Leonardo da Pisa (1170-1240), besser bekannt als Fibonacci, war dies einer der frühen Anstöße für den Aufschwung der Naturwissenschaften.

Für Algorithmen ist typisch, dass die Rechenvorschrift genau Schritt für Schritt angegeben ist. In diesem Sinne hat bereits Euklid (ca. 300 BC in Alexandria) Algorithmen formuliert. Einer der bekanntesten Algorithmen ist sein Verfahren zur Bestimmung des größten gemeinsamen Teilers zweier Zahlen, das noch heute *euklidischer Algorithmus* heißt. Ein anderes bekanntes algorithmisches Verfahren ist der Primzahlalgorithmus „Sieb des Eratosthenes“ von Eratosthenes (276 BC, Cyene - 194 BC, Alexandria[2]). Eratosthenes ist vor allem bekannt, weil er als erster ziemlich genau den Erdumfang ermittelt hat.

Das Thema Algorithmen überspannt einen riesigen Anwendungsbereich; schließlich handelt es sich dabei nicht nur um ein Thema der Informatik. In allen Anwendungsbereichen kennt man algorithmische Verfahren. Die Formulierung eines Algorithmus ist nicht ohne genaue Kenntnis der Eigenheiten der Anwendung möglich. Beispiele sind Verfahren der numerischen und der symbolischen Mathematik, Computergeometrie und Computergraphik, Graphentheorie, Kryptographie, die Methoden der Künstlichen Intelligenz usw. In diesem Kurs können wir natürlich nicht alle diese Bereiche ansprechen, sondern werden uns auf grundlegende Methoden im Umgang mit Datenstrukturen beschränken. Ich werde hier das *Suchen* und *Sortieren* in Feldern und den Aufbau von effizienten Verzeichnissen durch *Binärbäume* und durch *Hashtabellen* besprechen. In der Fachliteratur zu Algorithmen (z.B. Sedgewick [Sed1992]) finden Sie eine Einführung in eine größere Zahl von

[1] Dezimalzahlen wurden ab 300 vor der Zeitrechnung in Indien entwickelt.
[2] Alexandria war bis zur Christianisierung das beherrschende Wissenschaftszentrum der Antike.

algorithmischen Verfahren.

Algorithmen haben zwei sehr wichtige Aspekte: *Korrektheit* und *Effizienz*. Im ersten Abschnitt zeige ich zunächst an einem Beispiel, welche Rolle ein systematisches Vorgehen für die Entwicklung korrekter Algorithmen spielt. Danach bespreche ich, wie sich die Effizienz eines algorithmischen Verfahrens beschreiben lässt.

In den darauffolgenden Abschnitten werden diese Aspekte bei den verschiedenen Standardalgorithmen zum Suchen, Sortieren und zum Aufbau von Zuordnungen vorgestellt. Dabei werden Sie sehen, dass Effizienzüberlegungen keineswegs durch immer schneller werdende Hardware obsolet werden. Es ist nämlich so, dass es in sehr vielen Bereichen inzwischen ausgefeilte algorithmische Verfahren gibt, die bei großen Problemen um Größenordnungen schneller sind als die „naiven“ Verfahren, auf die Sie (oder ich) selbst ohne weiteres kommen.

Das Ziel des zweiten Teils der Vorlesung ist erreicht, wenn Sie anschließend einige wichtige Datenstrukturen und Algorithmen kennen, wenn Sie gelernt haben, Algorithmen zu bewerten und wenn Sie einige Wege zur Entwicklung von Algorithmen eingeübt haben.

Ein Algorithmus ist am einfachsten mit einem Rezept zu vergleichen. Dort wird beschrieben, wie man ausgehend von einer Menge von Zutaten nach mehreren Arbeitsgängen endlich das gewünschte Gericht „zaubert“. Die einzelnen Arbeitsschritte bestehen darin, dass der Zustand der Zutaten solange verändert wird, bis er den Zielvorstellungen entspricht. So gehen Kartoffeln durch das Schälen vom Rohzustand in den geschälten Zustand über, und – nach einigen Zwischenzuständen, wie dem „gewaschenen Zustand“ – durch das anschließende Kochen in den gewünschten Zustand „gekocht“.

Ähnlich sind auch in der Software Algorithmen als Abfolge von Zustandsveränderungen aufzufassen. Der momentane *Zustand* eines Programmablaufs lässt sich durch die Wertebelegung der Variablen definieren. Der in diesem Kurs verwendete Begriff des Algorithmus ist eng mit dem Paradigma der imperativen Programmierung verbunden.

Definition:

> *Ein **Algorithmus** ist eine Vorschrift, nach der durch eine Folge von mehreren Zustandsveränderungen aus einen **Anfangszustand** der gewünschte **Endzustand** hervorgebracht wird. Dabei sind die einzelnen Zustandsveränderungen durch den Algorithmus genau definiert, sie verlaufen deterministisch, und der Algorithmus terminiert nach endlich vielen Schritten.*

Es gibt zwei besonders herausragende Eigenschaften eines Algorithmus: die *Korrektheit* und die *Komplexität*.

Definition:

> *Unter der **Korrektheit** eines Algorithmus verstehen wir die Übereinstimmung seines Verhaltens mit der Aufgabenstellung (Spezifikation). Mit **Laufzeitkomplexität** bezeichnen wir die Abhängigkeit der Anzahl der Arbeitsschritte von der Problemgröße.*

Bei mathematischen Problemen ist es möglich, die Aufgabenstellung durch eine Formel genau zu präzisieren. In solchen Fällen kann die Korrektheit des algorithmischen Verfahrens formal untersucht werden. Ich will Ihnen hier an dem überschaubaren Beispiel der ganzzahligen Potenzierung zeigen, dass eine sorgfältige und formal durchdachte Entwicklung eines Algorithmus einem unüberlegten Drauflosprogrammieren hinsichtlich Korrektheit und Effizienz weit überlegten ist.

5.1 Ein negatives Beispiel

Um die Aufgabenstellung einzugrenzen, gebe ich Ihnen hier zunächst die Signatur der zu realisierenden Potenzfunktion vor. Natürlich ist eine solche Funktion praktisch stets schon fertig verfügbar, so dass Sie eigentlich nie in die Verlegenheit kommen werden, wirklich einmal eine Potenzfunktion selbst zu programmieren.[3] Andererseits können Sie an diesem einfachen Beispiel Probleme, die auch in anderen Anwendungen auftauchen, ganz gut erkennen.

```
double static power (double base, int exponent );
// power berechnet base^exponent
```

Angenommen, Sie geben dieses Problem einem Programmierer. Der kommt nach ein paar Tagen zurück und präsentiert Ihnen seine Lösung (Sie ist in einer Klasse `Math` eingebettet). Um zu prüfen, ob er seine Aufgabe auch korrekt erledigt hat, schreiben Sie mit JUnit ein kleines Testprogramm:

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class MathTest {
    private static final EPS = 1E-10;

    @Test
    public void testPower() {
        assertEquals(2.0, Math.power(2.0, 1), EPS);
        assertEquals(4.0, Math.power(2.0, 2), EPS);
        assertEquals(1.5, Math.power(1.5, 1), EPS);
    }
}
```

Als Ergebnis liefert JUnit den Erfolg anzeigenden grünen Balken. Alles ist in Ordnung, Sie bezahlen den Programmierer und setzen die Funktion in einem Ihrer Softwareprojekte ein. Ihr System funktioniert leider auf Dauer nicht einwandfrei. Als sich der Verdacht immer mehr verdichtet, dass die `power`-Funktion falsch sein könnte, schauen Sie sich einmal den Quelltext an:

```
public static double power (double base, int exponent ) {
// falsche Fassung !!
    double result = 0.0;
    for (int i = 1; i <= exponent ; i++)
        result += base;
    return result;
}
```

Klar, das konnte nicht funktionieren! Wenn Sie genau hinschauen, sehen Sie, dass diese Funktion, aufgrund eines „dummen“ Fehlers nichts anderes macht, als das Produkt der Parameter `base` und `exponent` zu ermitteln.

Aber, hat der Programmierer Sie tatsächlich betrogen, indem er Ihnen eine fehlerhafte Lösung verkauft hat? – Ich meine, das ist nicht unbedingt klar. Immerhin hat seine Funktion Ihren Abnahmetest bestanden. Außerdem haben Sie ihm auch nicht genau erklärt,

[3] siehe: `java.lang.Math.pow`

was Sie unter *Potenzierung* verstehen, so dass der Programmierer sich immer noch damit herausreden kann, dass er genau das gemacht hat, was Sie von ihm verlangt hatten.

Natürlich ist dieses Primitivbeispiel künstlich aufgebauscht. Aber sagen Sie nicht, dass es völlig praxisfern sei! Nach meinem Eindruck entsteht ein Großteil der Kosten der Softwareentwicklung erst in der Endphase eines Softwareprojekts, wenn sich herausstellt, dass die Anforderungen des Auftraggebers von den Softwareentwicklern nicht richtig verstanden und damit auch nicht richtig umgesetzt wurden.

Es stellt sich demnach die Frage, ob es nicht Grundtechniken gibt, mit denen man auf Anhieb (oder wenigstens in den meisten Fällen) korrekte Software entwickeln kann.

Unabhängig von den Methoden der nächsten Abschnitte, gibt es aber einen wichtigen Hinweis in Richtung Testbarkeit der Software.

Merksatz:

Testen kann die Anwesenheit von Fehlern aufdecken, aber nicht beweisen, dass die Software korrekt ist. Entwickeln Sie die Testverfahren gemeinsam (oder vor) der Software. Achten Sie dabei darauf, dass die Tests alle Sonderfälle abdecken. Bei Veränderungen des Algorithmus sollten Sie seine Korrektheit immer wieder anhand der Tests untersuchen.

5.2 Die Entwicklung korrekter Software

Aus dem Beispiel sollten Sie die wichtige Lehre ziehen, dass man sich auf bloßes Testen allein nicht verlassen kann.

In jedem Fall müssen wir zuerst einmal die Aufgabenstellung genauer beschreiben. Dazu gibt es unterschiedliche Verfahren. Ein Verfahren besteht darin, dass man für jeden Programmabschnitt angibt, welche logische Beziehung zwischen den Variablen nach der Berechnung bestehen soll, vorausgesetzt sie erfüllten vor Eintritt in den Programmabschnitt die vorgesehenen Voraussetzungen. Die Voraussetzungen nenne ich *Vorbedingung* Q, das Ziel *Nachbedingung* R.

Definition:

Ein Programmfragment $\{Q\}S\{R\}$[4] *ist* ***korrekt****, wenn für alle anfänglichen Variablenbelegungen, die mit der Bedingung* Q *verträglich sind, nach Ausführen des Programmabschnitts* S *eine Variablenbelegung vorliegt, für die die geforderte Zielbedingung* R *erfüllt ist.*

Die Aufgabenstellung unseres Beispiels kann ich damit wie folgt formulieren:

```
public static double power (double base, int exponent) {
    double result;
    // Q: bei negativem exponent muss base != 0 sein!
    ... Programmstueck S
    // R: result = base^exponent
    return result;
}
```

[4] In der Fachliteratur ist es üblich nicht die Programmanweisungen sondern die Zusicherungen in geschweifte Klammern einzuschließen.

Vorausgesetzt, das Programmstück S ist korrekt, dann erfüllt die Funktion ihren Zweck. Aber was bedeutet `baseˆexponent`? Das ist eine Operation aus dem Problembereich – in diesem Fall aus der Mathematik. Zuerst müssen wir also mathematisch definieren, was potenzieren heißt. Dies kann durch die folgenden Gleichungen geschehen:

$$x^{-y} = \frac{1}{x^y} \qquad \text{falls } x \neq 0 \tag{5.1}$$

$$x^y = \begin{cases} 1 & \text{falls } y = 0 \\ x \cdot x^{y-1} & \text{falls } y > 0 \end{cases} \tag{5.2}$$

Es gibt noch viele weitere Beziehungen. Zur Spezifikation der Aufgabenstellung und auch zur korrekten Lösung reichen aber diese Formeln bereits aus. Wie wir gleich sehen werden, benötigt man für ein *effizientes* Programm noch eine weitere Formel:

$$x^y = (x \cdot x)^{\frac{y}{2}} \qquad \text{für gerade } y \tag{5.3}$$

Der entscheidende Ansatzpunkt für ein Lösungsverfahren ist durch die Formel 5.2 gegeben, die es erlaubt, die Problemstellung rekursiv zu vereinfachen. Leider ist diese Formel nur auf positive Exponenten anwendbar. Hier hilft die Formel 5.1, mit deren Hilfe man allgemeine Aufrufe auf das Lösungsverfahren mit positiven Exponenten reduzieren kann. Die Technik der Reduktion eines Problems auf ein einfacheres Teilproblem wird in verschiedenen Varianten bei anderen Algorithmen immer wieder auftauchen. Man nennt das Prinzip auch *schrittweise Verfeinerung*.

Definition:

*Die Methode der **schrittweisen Verfeinerung** verlangt, dass man ein gegebenes Problem solange in Teilprobleme zerlegt, bis diese für sich zu lösen sind.*

Bei der schrittweisen Verfeinerung handelt es sich um eine top-down Vorgehensweise.

```
public static double power (double base, int exponent) {
    // bei negativem exponent muss base != 0 sein!

    double result;
    if ( exponent >= 0 ) {
        // Q1: exponent >= 0
        result = posPower(base, exponent);
    }
    else {
        // Q2: exponent < 0, base != 0
        // nach Formel 4.1:
        result = 1.0 / posPower(base, -exponent);
    }

    // R: result = baseˆexponent
    return result;
}

private static double posPower(double base,int exponent) {
    double result;
    // Q: exponent >= 0
    ... Programmstueck S
    // R: result = baseˆexponent
    return result;
}
```

Der mit dieser Verfeinerung erzielte Fortschritt besteht darin, dass die verbleibende Aufgabe jetzt *nur* noch darin besteht, die Potenzierung mit positiven Exponenten zu lösen.

Für die systematische Entwicklung der Fallunterscheidung (if) können Sie sich merken, dass sie dazu dient, *auf verschiedenen Wegen das gleiche Ziel zu erreichen*. Dabei hängt die Wahl des Weges von einer logischen Abfrage ab, deren Ergebnis bei der Formulierung der beiden Teilwege ausgenutzt wird. Dies habe ich in dem Beispiel durch die beiden Kommentare *Q1* und *Q2* ausgedrückt.

Zur Lösung des verbleibenden Teilproblems braucht man jetzt nur noch die eigentliche Potenzdefinition der Formel 5.2 zu verwenden:

```
private static
double posPower(double base, int exponent) {
    // Q: exponent >= 0

    double result;
    if ( exponent == 0 ) {
        // Q1: exponent = 0
        // nach Formel 5.2:
        result = 1.0;
    }
    else {
        // Q2: exponent > 0
        // nach Formel 5.2:
        result = base * posPower(base, exponent - 1);
    }

    // R: result = base^exponent
    return result;
}
```

Das sieht einfach aus. Ist es aber auch richtig? Wenn Sie das Programm Schritt für Schritt mit den mathematischen Formeln vergleichen, sehen Sie, *dass das Programm korrekt sein muss, wenn die Berechnung nach endlichen vielen Schritten abgeschlossen ist*. Schließlich haben wir nur solche Zustandsänderungen vorgenommen, die durch die Formel 5.2 gedeckt sind- Natürlich gilt dies nur, wenn die Vorbedingung erfüllt ist und `posPower` nicht mit negativen Exponenten aufgerufen wird. In dem Beispiel habe ich darauf verzichtet, die Vorbedingung zu überprüfen. Tatsächlich ist das auch eine Entwurfsentscheidung. Zweifellos kann man hier gegebenenfalls eine Ausnahme werfen und so die Robustheit des Algorithmus erhöhen.

Ein korrektes Programm muss in endlicher Zeit zu seinem Ergebnis kommen (terminieren). In diesem Fall nennt man es auch *total korrekt*. Wurde wie hier nur nachgewiesen, dass es, *falls* es terminiert, ein korrektes Ergebnis liefert, so bezeichnet man dies als *partielle Korrektheit*. Um zu beweisen, dass unser Programm total korrekt ist, müssen wir also noch beweisen, dass das Programm auch terminiert.

5.3 Die Terminierung und die Effizienz eines Algorithmus

Wir können die Terminierung der Potenzfunktion beweisen, indem wir zeigen, dass die Funktion `posPower(x,n)` genau n rekursive Aufrufe ausführt. Bei jedem rekursiven Aufruf ist der Übergabeparameter `exponent` um 1 kleiner als beim vorherigen Aufruf. Mit einem anfänglich positiven `exponent` muss daher stets nach genau n Schritten ein Aufruf mit `exponent`$= 0$ erfolgen, bei dem die rekursive Aufruffolge endet.

Merksatz:

Ein rekursiver Algorithmus terminiert garantiert, wenn bei jedem rekursiven Aufruf die Problemgröße reduziert wird und wenn er über eine Abbruchbedingung verfügt, bei der kein weiterer rekursiver Aufruf mehr erfolgt.

Die angegebene Bedingung ist *hinreichend* aber nicht *notwendig* für die Terminierung eines Algorithmus. Wenn die angegebene Bedingung nicht gilt, kann der Algorithmus trotzdem nach endlichen vielen Schritten terminieren. Es dann aber auch sein, dass es unmöglich ist, die Terminierung wirklich zu beweisen (*Halteproblem*). Die genaue Auseinandersetzung mit der Terminierung eines Algorithmus gehört in den Problemkreis *Berechenbarkeit*. Diese ist eine Bestandteil der Gebiete *Algorithmik* und *Theoretische Informatik*.

Ihnen sollten unbedingt die beiden Teile einer rekursiven Problemlösung, nämlich der *rekursive Aufruf* und die *Abbruchbedingung*, im Gedächtnis haften bleiben.

In der Praxis ist man nicht nur daran interessiert, *ob* ein Programm terminiert, sondern vor allem *wie lange* es rechnet. Die genaue Rechenzeit hängt allerdings nicht nur von dem Algorithmus, sondern von einer Anzahl weiterer Faktoren ab, die der Programmierer nicht im Griff hat. In erster Linie sind dies die Leistungsfähigkeit des Computers, die Qualität des Compilers, das Aufsetzen des Programms durch das Betriebssystem und, bei einem Multitasking-System, die Rechnerauslastung zum Zeitpunkt der Berechnung.

Der einzige nur von der Aufgabenstellung und nicht von der Ablaufumgebung abhängige Problemparameter, der für die Laufzeit entscheidend ist, ist die sogenannte *Problemgröße*. Ein Programm, das viele oder schwer zu behandelnde Eingabewerte zu verarbeiten hat, wird nämlich sicher meist eine größere Laufzeit erfordern, als wenn nur wenige Eingaben zu verarbeiten wären. In unserem Beispiel wird die Anzahl der rekursiven Aufrufe durch den Betrag des Exponenten bestimmt. Diese Zahl, die ich hier n nenne, bestimmt die Problemgröße. Unsere Aufgabe besteht darin, eine Formel anzugeben, die die tatsächliche Laufzeit T als Funktion $T(n)$ in Abhängigkeit dieser Größe n ausdrückt.

Wenn man tatsächliche Zeitmessungen anstellt, wird man feststellen, dass die Abhängigkeit der Rechenzeit von der Eingabe eine sehr komplizierte Funktion sein kann. Meist ist man aber nur an relativ einfachen Angaben interessiert, mit denen man insbesondere das asymptotische Verhalten für Eingaben, die große Rechenzeit erfordern, abschätzen kann. Dieses asymptotische Verhalten wird durch die O-Notation beschrieben.

Wenn man sagt, dass die Laufzeit eines Problems $O(n^2)$ sei, so bedeutet das, dass der führende Term in der Formel für die Laufzeit proportional zu n^2 ist.

Definition:

*Die **O-Notation** gibt den führenden Term in der Formel für die Laufzeit eines Algorithmus in Abhängigkeit der Problemgröße an. Konstante Faktoren werden weggelassen. Formal gilt [Sed1992]: Eine Funktion $f(N)$ ist in $O(g(N))$, mit einer (möglichst einfachen) positiven Funktion $g(N)$, wenn es Konstanten c_0 und N_0 gibt, so dass $|f(N)| < c_0\, g(N)$ für alle $N > N_0$ gilt.*

Die vollständige Formel für die Laufzeit $T(n)$ mag z.B. etwa so aussehen (Einheiten habe ich hier weggelassen):

$$T(n) = 0.00012n^2 + (0.059 + 0.04\log(n))n + 0.003$$

$T(n)$ ist in der Komplexitätsklasse $O(n^2)$. Die Schreibweise $O(n^2)$ rechtfertigt sich auch dadurch, dass für das Grenzwertverhalten gilt:[5]

$$\lim_{n\to\infty} \frac{T(n)}{n^2} = 0.00012$$

Zumindest dann, wenn N gegen ∞ geht, gilt die Definitonsgleichnung für einen Faktor c_0, der größer als 0.00012 ist. Ab $N_0 = 50000$ kann man $c_0 = 0.00013$ setzen. Ab $N_0 = 5000$ gilt $c_0 = 0.0002$ und ab 500 immerhin $c_0 = 0.00074$. Man kann dies auch so ausdrücken: Bei kleineren Werten von N_0 erhalten wir grundsätzlich ungünstigere (größere) Werte für c_0. Wir dürfen die asymptotischen Formeln daher nicht auf zu kleine Problemgrößen ausdehnen. Wenn es in der praktischen Anwendung aber (überhaupt) um Laufzeitverhalten geht, dann nur bei relativ großen Problemen. Für diese nimmt die Aussagekraft der asymptotischen Formel zunehmend zu.[6]

Sie erkennen daraus, dass genau genommen zwar jede der asymptotischen Formeln erst ab einem bestimmend N_0 wirklich eine obere Abschätzung für die Laufzeit gestattet, dass man aber die O-Notation mit einem gewissen Fehlerspielraum für praktisch verwertbare Laufzeitabschätzungen verwenden kann. Man kann nämlich ausgehend von der Kenntnis der Laufzeit für eine bestimmte Problemgröße ungefähr die Laufzeit für größere Probleme abschätzen. Solange man dabei die Grenzen des asymptotischen Verfahrens beachtet, lässt sich so das Laufzeitverhalten verschiedener Algorithmen vergleichen.[7]

Im Folgenden betrachten wir zwei typische Fälle. Andere Fälle kann man sich ähnlich herleiten.

Zunächst nehmen wir an, dass ein Laufzeitverhalten von $O(n^2)$ vorliegt (zum Beispiel ein direktes Sortierverfahren). Wir wissen dann, dass die Laufzeit ungefähr als $T = a\,n^2$ berechnet werden kann. Um die unbekannte Konstante a zu ermitteln, müssen wir eine Messung, etwa mit der Problemgröße n_0, durchführen. Wir messen dabei die Zeit T_0. Unser Ziel ist herauszufinden, wie groß die Zeit T_1 für die Problemgröße n_1 ist.

Für die Berechnung dividieren wir die beiden Zeitbeziehungen durcheinander und erhalten:

$$\frac{T_1}{T_0} = \frac{a\,{n_1}^2}{a\,{n_0}^2} = \left(\frac{n_1}{n_0}\right)^2$$

und erhalten damit einen Ausdruck für T_1 der nur noch von dem Verhältnis der Problemgrößen und der Zeit T_0 abhängt:

$$T_1 = T_0 \left(\frac{n_1}{n_0}\right)^2$$

Nehmen wir also mal an, wir haben herausgefunden, dass $n_0 = 1000$ Zahlen in $T_0 = 0.01$ Sekunden sortiert werden. Dann werden 100000 Zahlen in der $100^2 = 10000$ fachen Zeit also in 100 Sekunden sortiert.

Bei vielen Algorithmen tauchen Logarithmen in der Laufzeitabschätzungen auf. Dadurch wird die Berechnung aber nicht wirklich komplizierter. Da in den Formeln nur Verhält-

[5] Diese Form sagt, dass die Laufzeit nicht nur durch eine quadratische Kurve beschränkt ist, sondern dass sie sogar asymptotisch durch diese beschrieben ist. Dies beschreibt man exakter mit $\Theta(n^2)$.

[6] Diese angegebenen Werte kann man mit einer Mathematiksoftware wie Maple oder Mathematica ermitteln.

[7] Ganz exakte Daten lassen sich nur durch Laufzeitmessungen (Benchmarks) ermitteln.

nisse von Logarithmen auftauchen, ist die Berechnung unabhängig von der Basis der Logarithmen, so dass man (bei geeigneten Verhältnissen) nicht mal einen Taschenrechner braucht.

Effiziente Sortierverfahren haben eine Komplexität von $O(n \log n)$. Wir erhalten für die Zeit:

$$T_1 = T_0 \frac{n_1}{n_0} \frac{\log n_1}{\log n_0}$$

Wenn wir das letzte Beispiel für einen effizienten Algorithmus durchrechnen, verwenden wir am besten Zehnerlogarithmen.[8] Wir müssen nur wissen, dass $\lg 10^n = n$ ist. Dann haben wir

$$T_1 = T_0 \frac{100000}{1000} \frac{5}{3} = 1,7\,\mathrm{s}$$

Das Ergebnis spricht für sich: Bei dem einfachen Algorithmus steigt die Laufzeit erheblich schneller an, als bei einem effizienten Verfahren. Hier bin ich von der unrealistischen Annahme ausgegangen, dass beide Algorithmen bei 1000 Elementen gleich schnell sind.

Gemäß ihrer Definition stellt die O-Notation immer eine obere Grenze für die Laufzeit dar (*worst case*). Oft verwendet man die Schreibweise aber auch zur ungefähren Beschreibung des durchschnittlichen (*average case*) oder des besten (*best case*) Laufzeitverhaltens. In dem Beispiel der Potenzierung – so weit wir es bisher behandelt haben – macht das keinen Unterschied. In allen Fällen ist die Laufzeit $O(n)$. Wenn nichts anderes gesagt ist, ist immer der ungünstigste Fall gemeint.

5.4 Die Entwicklung einer effizienteren Lösung

Das Programm, das wir gerade entwickelt haben, ist eine korrekte und akzeptable Lösung des Potenzierungsproblems. Es ist jedoch noch lange nicht die beste Lösung! Dies können Sie durch etwas Nachdenken leicht herausfinden.

Angenommen wir sollten 2^{16} berechnen. Wir könnten dies so tun:

$$2^{16} = \underbrace{2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2}_{15 \text{ Multiplikationen}}$$

So rechnet die Funktion `posPower`. Es geht jedoch besser:

$$\begin{aligned} 2^{16} &= (2 \cdot 2)^{16/2} &&= 4^8 \\ &= (4 \cdot 4)^{8/2} &&= 16^4 \\ &= (16 \cdot 16)^{4/2} &&= 256^2 \\ &= (256 \cdot 256)^{2/2} &&= 65536^1 = 65536 \end{aligned}$$

Vergleichen Sie einmal! Nach der ersten Rechenmethode müssen 15 Multiplikationen ausgeführt werden. Nach dem zweiten Verfahren sind aber nur 4 Multiplikationen erforderlich!

Zugegeben, man muss erst einmal auf so eine Idee kommen. Und selbst dann wird die zweite Lösung sicher deutlich komplizierter als das erste Programm sein. Es gibt leider kein Kochrezept, wie man für jedes Problem die beste Lösung findet.

[8] Wenn wir es mittels es mit Zweierpotenzen zu tun haben, nehmen mir Logarithmen zur Basis 2.

Merksatz:

Die Entwicklung korrekter und effizienter Lösungen erfordert das nötige problemspezifische Wissen und ein großes Maß an Kreativität. Bei der Umsetzung dieses Wissens benötigt man fundierte Kenntnisse der wichtigsten Lösungsmethoden.

Die Anwendung der Formel 5.3 führt zu der folgenden letzten Fassung der rekursiven Formulierung der Potenzfunktion.

```
private static
double posPower(double base, int exponent) {
    // Q: exponent >= 0

    double result;
    if (exponent == 0) {
        // Q1: exponent = 0
        // nach Formel 5.2:
        result = 1.0;
    }
    else if (exponent % 2 == 0) {
        // Q2: exponent > 0 && exponent ist gerade
        // nach Formel 5.3:
        result = posPower(base*base, exponent/2);
    }
    else {
        // Q3: exponent > 0 && exponent ist ungerade
        // nach Formel 5.2:
        result = base * posPower (base, exponent - 1);
    }

    // R: result = base^exponent
    return result;
}
```

Von der (partiellen) Korrektheit dieser Version können Sie sich leicht durch Vergleich mit den angegebenen Formeln überzeugen. Wenn man nun die Anzahl der rekursiven Aufrufe und damit die Laufzeit untersucht, wird es jetzt etwas komplizierter. Hier ist es nützlich, die oben angesprochene Unterscheidung in best case, average case und worst case vorzunehmen.

Der einfachste Fall – und gleichzeitig der best case – liegt vor, wenn n eine Zweierpotenz ist, also z.B. 16 oder 32. In diesem Fall wird bis zum Ende jedes Mal die zweite Alternative ausgeführt. Für die Anzahl der rekursiven Aufrufe (es fehlt jeweils der eine Schritt für die 1) gilt etwas vereinfacht:

n	1	2	4	8	$\cdots$	n
$T(n)$	0	1	2	3	$\cdots$	$\log_2 n$

Der gegenteilige Fall – der worst case – liegt vor, wenn $n = 2^m - 1$ ist, d.h., wenn n um 1 kleiner als eine Zweierpotenz ist. In diesem Fall hat man erst den Fall einer ungeraden Zahl zu behandeln, dann eine gerade Zahl, dann wieder eine ungerade und so weiter. Offensichtlich ist die Rechenzeit beinahe doppelt so groß wie im günstigsten Fall.

$$T(n)_{\text{worst case}} \approx 2T(n)_{\text{best case}}$$

Ein konstanter Faktor kommt in der O-Notation nicht zum Ausdruck. Damit gilt $T = O(\log n)$. Natürlich ist damit auch die durchschnittliche Laufzeit proportional zu n^2.

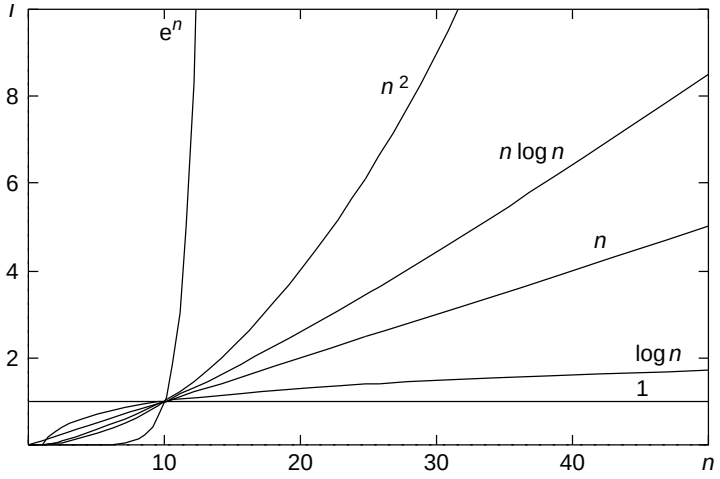

Abbildung 5.1: Die Abbildung gibt an, wie sich die Laufzeit T in Abhängigkeit der Problemgröße n bei unterschiedlichen funktionalen Zusammenhängen verhält. Die verschiedenen Kurven sind so angepasst, dass sie alle bei $n = 10$ den Wert 1 ergeben.

In Abbildung 5.1 können Sie erkennen, dass verschiedene funktionale Abhängigkeiten zu sehr unterschiedlichem Laufzeitverhalten führen. Sie können erkennen, dass vor allem die e^n-Kurve sehr schnell zu hohen Werten führt, während auf der anderen Seite $\log n$ so langsam ansteigt, dass man dabei beinahe von konstanter Zeit sprechen kann.

Bei kleinen Zahlen ist die O-Notation unbrauchbar. Es kann sein, dass der „schlechtere“ Algorithmus erheblich schneller ist als der asymptotisch bessere. Wenn eine Berechnung jedoch häufig für große Werte auszuführen ist, kann der Vorteil des besseren Verfahrens aber gigantisch werden. Bei großen n sind die oben angesprochenen Ungenauigkeiten der asymptotischen Abschätzung daher auch nicht wirklich von Bedeutung.

Die Potenzfunktion ist kein typisches Beispiel für rechenintensive Probleme. Dies können Sie schon daran erkennen, dass bei großen Exponenten schnell die computerintern realisierten Zahlenbereiche überschritten werden. Auch wenn Sie in diesem Kurs keine Beispiele hierfür kennenlernen, so kommen rechenintensive Probleme, bei denen Effizienzüberlegungen eine wichtige Rolle spielen, in der Praxis aber ziemlich häufig vor. Sie zeichnen sich meist dadurch aus, dass mit sehr vielen Datenelementen zu rechnen ist (Visualisierung, Computerspiele, Simulation, Bildverarbeitung usw.).

Ganz besonders aufwändig ist die Lösung von Problemen, deren Laufzeit proportional zu einer Exponentialfunktion ansteigt. Die Exponentialfunktion e^n hat nämlich die Eigenschaft, dass sie für $n \to \infty$ schneller als jedes Polynom ansteigt. Bei solchen $O(e^n)$-Algorithmen wird die Grenze der praktischen Berechenbarkeit schon bei relativ kleinen Problemgrößen n erreicht. Leider gehören fast alle Verfahren der Künstlichen Intelligenz, genauso wie sehr viele Optimierungsprobleme zu dieser Komplexitätsklasse. Meist besteht der einzige praktikable Lösungsweg darin, vereinfachte Verfahren – *Heuristiken* – zu finden, mit denen sich ein brauchbares, suboptimales Ergebnis in akzeptabler Zeit finden lässt.

5.5 Korrekte Iteration und Schleifeninvariante

Man kann jedes iterative Problem auch rekursiv lösen und umgekehrt jeden rekursiven Algorithmus in einen äquivalenten iterativen Algorithmus umformen. Manchmal sind re-

kursive Programme besser zu verstehen und zu verifizieren: Dies gilt vor allem dann, wenn die Aufgabenstellung selbst rekursiv formuliert ist. Dadurch, dass bei jedem rekursiven Aufruf „neue“ Variable zur Verfügung stehen, ist die Überprüfung der Korrektheit rekursiver Funktionen einfach. Bei der Verifikation von Iterationsschleifen muss man eine kompliziertere Technik verwenden. (Die Iteration ist „logisch“ komplizierter als die Rekursion, da in einer Iterationsschleife die Variablenwerte wiederholt verändert werden). Trotzdem werden natürlich viele Algorithmen iterativ formuliert, so dass wir uns die dabei nötige Vorgehensweise genauer ansehen sollten. Als Beispiel betrachte ich hier die systematische Entwicklung einer iterativen Fassung von `posPower`. Zunächst schreibe ich den Rahmen für diese Prozedur auf:

```
private static
double posPower(double base, int exponent) {
    // Q: exponent >= 0
    double result = 1.0;
    int i = 0;
    ...
    while (...) {
       ...
    }
    // R: result = base^exponent
    return result;
}
```

Da ich am Ende einen Wert zurückgeben muss, habe ich hier bereits die Variable `result` eingeführt, die schließlich das Funktionsresultat enthalten soll. Die Potenzierung soll mit einer While-Schleife gelöst werden. Hierbei sind die nötige Initialisierung der Variablen, die Schleifenbedingung und die Schleifenanweisung zunächst noch offen.

Bei den bisherigen rekursiv formulierten Beispielen können wir an jeder Stelle genau sagen, welche Variablenbelegungen möglich sind und wie die Werte sich zueinander verhalten. Dies scheint bei einer Schleife zunächst anders zu sein, da sich die Werte der Variablen ja bei jedem Schleifendurchlauf ändern können, so dass wir (was eigentlich auch der Sinn einer Schleife ist) jedes Mal (anscheinend) ganz andere Verhältnisse haben. Der entscheidende Trick zum Verständnis eines iterativen Programmteils besteht daher darin, eine logische Aussage zu finden, die unverändert während des gesamten Schleifendurchlaufs gilt. Wenn sich aus dieser Aussage am Ende auch noch das richtige Ergebnis ableiten lässt, nennt man diese Aussage *Schleifeninvariante*.

Definition:

> *Die **Schleifeninvariante** ist eine logische Aussage, die beim ersten Eintritt in die Schleife, bei jedem wiederholten Eintritt in die Schleife und nach dem Verlassen der Schleife gilt. Aus der Schleifeninvariante und aus der negierten Schleifenbedingung (nach dem Verlassen der Schleife gilt die Invariante weiter, die Schleifenbedingung jedoch nicht) muss die Zielaussage folgen.*

Die Abbildung 5.2 zeigt, wie man sich das Konzept der Schleifeninvarianten bildlich veranschaulichen kann, indem man sie mit dem Begriff der Richtung eines Weges vergleicht. Die grau unterlegten *Aussagen* geben Auskunft über den jeweiligen Zustand – wo man sich befindet. Die auf hellem Untergrund gedruckten *Anweisungen* bewirken eine Änderung des Zustandes. Dabei soll sich hier einerseits etwas ändern, nämlich die Distanz zum Ziel soll verringert werden. Andererseits muss aber auch etwas gleichbleiben, bei der Zielsuche muss man nämlich die richtige Richtung beibehalten. Genauso wie bei einer Wanderung in unbekanntem Gelände ein Kompass stets die unveränderliche Richtung

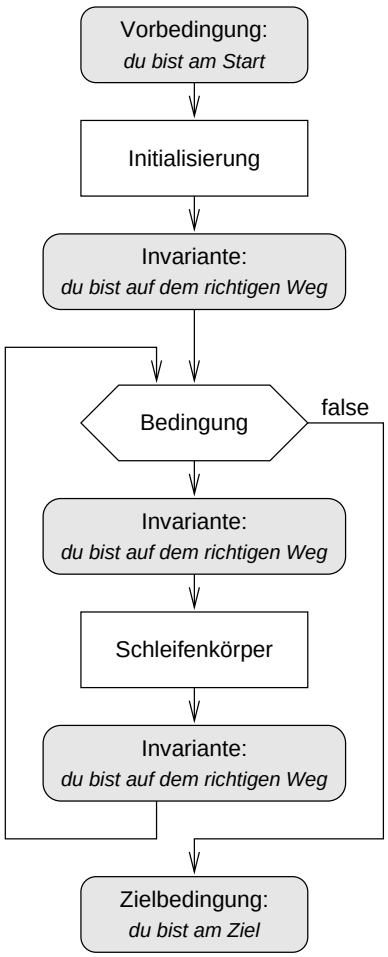

Abbildung 5.2: Das Ablaufdiagramm zeigt, dass Zusicherungen (grau unterlegte Flächen) stets zwischen einzelnen Programmaktionen stehen. Wenn jede Zusicherung für sich erfüllt ist, *muss* nach Verlassen der Schleife die Zielbedingung erreicht sein.

zum Ziel zeigt, dient eine Invariante dazu, die Entwicklung einer Schleife in die richtige Richtung zu lenken.

Gleichzeitig macht die Darstellung auch das Terminierungsproblem deutlich. Der Algorithmus terminiert, wenn man sich in jedem Schleifendurchlauf um ein merkliches Stück – genauer: mindestens um eine festgelegte Minimalstrecke – dem Ziel nähert. Darüber macht die Schleifeninvariante keine Aussage. *Wenn* die Schleife jedoch terminiert, dann hat man mit Sicherheit das Ziel erreicht.

Es ist nicht immer einfach, eine Schleifeninvariante zu finden – in diesem Kurs fehlt für eine systematische Einführung in diese Technik einfach die Zeit (in [Gri1983] wird sehr ausführlich in diese Methodik eingeführt). Die Schleifeninvariante findet man meist, indem man die Aussage der Zielbedingung abschwächt. Die wichtigsten Abschwächungen entstehen durch:

- Umwandeln einer Konstanten der Zielaussage in eine Variable
- Einführung von zusätzlichen Variablen in die Zielaussage
- Weglassen von Und-verknüpften Bedingungen

Das Auffinden der Invarianten ist eine kreative Tätigkeit, die zu verschiedenen brauchbaren Lösungen führen kann. Die mehr oder weniger geschickte Wahl der Schleifeninvarianten bestimmt die Qualität des daraus abgeleiteten Algorithmus. In unsererem Beispiel wähle ich zum Finden der Invarianten die Methode des *Umwandelns einer Konstanten in eine Variable*. Damit kommen wir bei der Potenzierung zu einer einfachen $O(n)$-Formulierung. Wir gehen von der folgenden Aussage aus:

```
// Invariante: result = base ^ i
```

Hinter dieser Formulierung verbirgt sich die *Schleifenidee*: In der Variablen `result` ist das vorläufige Ergebnis nach i Schleifendurchläufen gespeichert. Bei der Wahl der Invarianten ist wichtig, dass sie leicht erfüllt werden kann (hier geschieht dies mit: `result = 1; i = 0;`) und dass nach dem Durchlaufen der Schleife aus der Invarianten die Zielbedingung folgt (hier, wenn `i=exponent`).

Nachdem wir jetzt eine Invariante haben, will ich Ihnen nun zeigen, wie wir daraus das Programm Schritt für Schritt *ableiten* können. Zunächst müssen wir dafür sorgen, dass die Schleifeninvariante *vor* dem Eintritt in die While-Schleife gilt. Dies ist die Aufgabe der Initialisierung.

Die richtige Initialisierung der Variablen `result` und `i` habe ich schon angegeben. Die Schleifenbedingung ergibt sich automatisch daraus, dass nach der Schleife die Bedingung R gelten soll.

```
private static
double posPower(double base, int exponent) {
    // Q: exponent >= 0
    double result = 1.0;
    int i = 0;
    // I: result = base ^ i;
    while (i != exponent) {
        ...
    }
    // hier gilt: I && exponent == i
    // daraus folgt:
    // R: result = base^exponent
    return result;
}
```

Beachten Sie, dass bei diesem Programmteil bereits garantiert ist, dass nach dem Verlassen der While-Schleife in der Variablen `result` das richtige Ergebnis steht – vorausgesetzt, wir halten uns bei der Formulierung des Schleifenkörpers an die Invariante.

In die Schleife gehört etwas, das uns dem Ziel näherbringt. In der einfachsten Form ist dies das Erhöhen der Variablen `i`. Die Schleife könnte also so aussehen:

```
while(i != exponent) {
    i++;
}
```

Ist dies jedoch richtig? – Natürlich nicht! Man sofort sehen, dass die Invariante nach dem Erhöhen von `i` nicht mehr gelten kann. Um die Invariante zu „retten“, müssen wir auch eine andere Variable der Invariante verändern. Nach der Formel 5.2 sollten wir die Variable `result` mit `base` multiplizieren, um so das größer gewordene `i` zu berücksichtigen – hier sehen Sie das, indem Sie die zweite Teilformel von 5.2 rückwärts lesen. Dann gilt

jedenfalls auch nach der Schleife wieder die Invariante, so dass das Programm insgesamt korrekt ist (quod erat demonstrandum).

```
private static
double posPower(double base, int exponent) {
    // Q: exponent >= 0
    double result = 1.0;
    int i = 0;
    // I: result = base ^ i;
    while (i != exponent) {
        result *= base;
        i++;
    }
    // hier gilt: I && exponent == i
    // daraus folgt:
    // R: result = base^exponent
    return result;
}
```

Die Abschätzung der Programmlaufzeit von Schleifen ist oft sehr einfach. Meist lässt sich nämlich – dies gilt vor allem für Zählschleifen – die genaue Anzahl der Schleifendurchläufe angeben. Der Zeitaufwand für Schleifen, die in „Einerschritten“ durchlaufen werden, ist gleich $O(n)$.

5.6 Zusammenfassung der Grundzüge der Algorithmenentwicklung

Die Beispiele dieses Kapitels dienten dazu, auf ein paar Grundelemente der Entwicklung von Algorithmen hinzuweisen:

1. Die Entwicklung eines Algorithmus beginnt mit dem genauen Verständnis der Aufgabenstellung. Man muss die *Vorbedingungen* definieren, die es ermöglichen, dass die geforderte *Zielbedingung* erreicht werden kann.
2. Komplexe Probleme lassen sich nur lösen, indem man sie in einfachere Teilprobleme zerlegt.
3. Probleme auf großen Datenmenge oder mit einer großen Zahl von Berechnungsschritten erfordern Wiederholung. Wiederholung wird durch *Iteration* oder durch *Rekursion* ausgedrückt.
4. Schwierigere Abläufe lassen sich als ein schrittweises Annähern an das Ziel des Verfahrens auffassen.
5. Iterative Algorithmen lassen sich am besten mit einer *Schleifeninvarianten* verstehen. Die Schleifeninvariante ist eine Bedingung die vor, in und nach der Schleife gilt. Sie beschreibt die eigentliche Idee des Verfahrens.
6. Wenn ein Algorithmus sein Ziel nicht erreichen kann, muss er dies deutlich machen. In Java geschieht dies durch das Erzeugen einer Ausnahme.

Zur Beschreibung von Algorithmen braucht man eine formale Sprache, z.B. eine Programmiersprache. Welche Sprache man dazu verwendet, ist ziemlich egal. Meist „vergisst“ man völlig die konkrete Syntax irgendeiner Programmiersprache und verwendet

einen (manchmal selbstdefinierten) *Pseudocode* der einfach nur dazu da ist, die algorithmischen Abläufe verständlich auszudrücken. Dabei sind Details einer konkreten Programmiersprache oft nur störend.

Ich werde ab und zu Pseudocode zur Beschreibung von Algorithmen verwenden. Neben Pseudocode gibt es noch andere Verfahren zur Beschreibung von Algorithmen, wie Nassi-Sneidermann-Diagramme (Struktogramme) und Ablaufdiagramme. Abgesehen von einer ersten Einführung in algorithmisches Denken haben diese Diagramme aber heute keine Bedeutung mehr, da man mit ihnen komplexe Abläufe nicht adäquat darstellen kann. Ich persönlich glaube auch, dass sie dazu verführen, den konkreten Ablauf eines Teilalgorithmus für wichtiger zu halten, als die dahinterliegengen Idee, die sich eher in einer beschreibenden Aussage (z.B. Invariante) erklären läßt.

Kapitel 6

Datenstrukturen

Nel mezzo del cammin di nostra vita
Mi ritrovai per una selva oscura
Ché la diritta via era smarrita.[1]
Dante Alighieri, L'Inferno

In diesem Kapitel werden die grundlegenden dynamischen Datenstrukturen vorgestellt. An den Beispielen möchte ich Ihnen gleichzeitig wichtige Aspekte der Softwareentwicklung vermitteln:

1. Für große Projekte ist entscheidend, dass die Gesamtaufgabe in Teilaufgaben zergliedert werden kann, die unabhängig voneinander gelöst werden können. Diese Aufteilung funktioniert nur dann, wenn die Schnittstellen zwischen den Teilbereichen auf ein Mindestmaß reduziert werden.

2. Ein Aspekt der Aufteilung in Teilbereiche ist, grundlegende Datenstrukturen und Algorithmen aus der eigentlichen Anwendungsebene in grundlegendere Bibliotheken und Frameworks zu verlagern.

3. Dabei spielen *Abstrakte Datentypen* (ADT) eine wichtige Rolle. Sie zeichnen sich dadurch aus, dass sie einen in sich abgeschlossenen Teilbereich nach außen hin nur durch eine implementierungsunabhängige Schnittstelle darstellen.

4. Bei der Softwareentwicklung werden immer wieder bestimmte Grundelemente und Entwurfsmuster eingesetzt. In diesem Kapitel bespreche ich grundlegende Datenstrukturen, die jeder Informatiker kennen sollte.

5. Wenn immer möglich, wird man professionell entwickelten Bibliothekslösungen den Vorzug vor Eigenentwicklungen geben. Der richtige Einsatz von Bibliotheken setzt allerdings ein Grundverständnis der elementaren Datenstrukturen und Algorithmen und ihrer Eigenschaften voraus.

Ich will Ihnen zunächst die Grundidee des Programmierens von Datentypen näher erläutern. Ich betone dabei die Bedeutung der Trennung der Spezifikation des *Verhaltens* eines Datentyps von der *Repräsentation* seiner internen Datenstruktur und internen Funktionsweise. Während die Schnittstelle eine Zusage an den Benutzer einer Klasse macht, wird die Programmierung der Funktionalität von einer genauen Festlegung der Bedeutung der Instanzvariablen (z.B. in einer *Klasseninvarianten*) gesteuert.

[1] Auch bei Datenstrukturen führt der direkte (lineare) Weg nicht immer zum Ziel – dann befindet man sich „in einem dunklen Wald".

Nach einer allgemeinen Einführung bespreche ich beispielhaft einige Datentypen, an denen die Grundprinzipien der Entwicklung nach dem Modell des abstrakten Datentyps dargestellt werden.

Ich beschreibe zunächst, ausgehend von der abstrakten Idee sequentieller Datenstrukturen, den konkreten Datentyp `SinglyLinkedList` als Alternative zu Feldern.

An dem Beispiel der Datentypen `Stack` und `Queue` zeige ich Ihnen dann, wie man in Java die Schnittstelle abstrakter Datentypen so durch eine Schnittstelle beschreiben kann, dass durch konkrete Klassen unterschiedliche Implementierungen der Schnittstelle vorgenommen werden können. Abschließend werde ich Ihnen die etwas komplexere Datenstruktur des Binärbaums vorstellen.

6.1 Was sind abstrakte Datentypen?

Softwaresysteme können zwei grundverschiedene Arten von Klassen enthalten, nämlich *anwendungsbezogene Klassen*, die ein Modell der Wirklichkeit darstellen und *implementierungsbezogene Klassen*, die ausschließlich (oder vorwiegend) nur für die computerinterne Realisierung des Modells da sind.

Wiederverwendung, vielseitige Verwendung von Algorithmen, kurz wirtschaftliche und verständliche Software erhalte ich nur, wenn ich fähig bin mich von einzelnen konkreten Aufgaben und Realisierungen zu lösen. Kurzum: Ich muss in der Lage sein Software *abstrakt* zu formulieren.

Klassen können in Java zur Formulierung von benutzerdefinierten Datentypen verwendet werden. Anwendungsbezogenen Klassen verstehen Sie vielleicht eher als die Beschreibung einer Menge von realen Objekten denn als Definition eines Datentyps. Es fällt Ihnen aber bestimmt nicht schwer, in Schnittstellen der Java-Bibliothek wie `List`, `Set` oder `Map` eine Erweiterung der Datentypen der Programmiersprache zu sehen.

Merksatz:

> *Ein objektorientiertes System besteht aus Klassen, die reale Objekte darstellen und bei deren formaler Beschreibung vordefinierte und benutzerdefinierte Datentypen Verwendung finden.*

Es ist sinnvoll eigene Datentypen so zu definieren, dass die Operationen in syntaktischer und semantischer Hinsicht dem vom Benutzer erwarteten Verhalten entsprechen. Häufig entsteht die Benutzererwartung aus der Übertragung des Verhaltens bekannter Konzepte. In vielen Programmiersprachen ist es daher möglich die vordefinierten Operatoren (wie + und $\star$) mit neuen Bedeutungen zu versehen. Wenn wir zum Beispiel in der Programmiersprache *Scala*[2] einen eigenen Datentyp für beliebig genau darstellbare rationale Zahlen einführen würden, sollten wir unbedingt die Addition und die Multiplikation in der von den reellen und ganzen Zahlen her bekannten Syntax einführen. Die Anwendung von einer nach diesem Prinzip konstruierten Klasse `Bruch` wird dann so aussehen:

```
val a = Bruch(3,4)          // a = 3/4
val b = 2 * a               // b = 2 * 3/4 = 3/2
println(b + Bruch(1,2))     // Ausgabe: 2
```

[2] Scala ist eine objektorientierte Sprache, die über Java hinausgehende Konzepte erprobt. Der Scala-Compiler erzeugt Java-Bytecode. Scala Programme werden in der JVM ausgeführt und nutzen die Java-Bibliothek.

Die Klasse `Bruch` ist intuitiv zu benutzen, da sie auf bereits bekannten Konzepten aufbaut.

In dem Eingehen auf die Benutzererwartungen drückt sich das Prinzip der *syntaktischen Stetigkeit* aus, das bei der Einführung neuer Konzepte beachtet werden sollte: *Neue Konzepte sollen möglichst als eine natürliche Erweiterung bekannter Konzepte und Vorstellungen formuliert werden*. Bei Beachtung der syntaktischen Stetigkeit wird die Anwendbarkeit neuer Klassen deutlich erleichtert und damit auch die Gefahr der fehlerhaften Benutzung erheblich verringert.

In Scala wird die syntaktische Stetigkeit insbesondere durch die Möglichkeit des Überladens von Operatoren und durch automatische Konvertierung erreicht. Das gibt es in Java nicht, da automatische Darstellungsumwandlung auch die Verständlichkeit eines Programms erschweren kann. Trotzdem kann man auch in Java das Prinzip der syntaktischen Stetigkeit sinnvoll anwenden, indem man sich nämlich an die Stilregeln und Vorgaben der Standard-Bibliotheken hält.

Ganz grundsätzlich gehören zu einem Datentyp (wie z.B. `int` oder `double`) zwei Dinge: eine rechnerinterne *Repräsentation* und eine Menge von *Operationen*, die auf den Objekten dieses Typs ausgeführt werden können.

Viele Programmierer halten die Implementierung einer Klasse für wichtiger als die genaue Definition der Schnittstelle eines Datentyps. Für die langfristige Verwendung eines Systems von Klassen ist das jedoch der falsche Ansatz. Denken Sie nur daran, dass Sie bei der Verwendung eines eingebauten Datentyps (z.B. `double`) nicht daran interessiert sind, wie dessen Operationen intern implementiert sind. Sie interessieren sich nur dafür, wie man diese Operationen ansprechen kann.

Es gibt einige weitere Gründe, die Schnittstelle und die Implementierung eines Datentyps streng zu trennen. Dazu zählen unter anderem eine bessere Lesbarkeit, eine bessere Testbarkeit und eine bessere Veränderbarkeit der Implementierung:

- Eine Beschreibung eines Datentyps, die das *Verhalten* dieses Datentyps beschreibt, ist verständlicher als eine Beschreibung, die auf den Details der Realisierung beruht. Als ein Beispiel können Sie die Gebrauchsanleitung eines Automobils nehmen: Dort wird Ihnen z.B. erklärt, was mit Ihnen und dem Automobil geschieht, wenn Sie auf das Gaspedal treten (das Fahrzeug beschleunigt), aber es wird nicht erklärt, wie dies im Detail durch die Motorsteuerung und die Physik der Verbrennung zustande kommt.

- Erfolgreiches Testen von größeren Softwaresystemen ist nur möglich, wenn die Software in Komponenten zerlegt werden kann, die für sich getestet werden können. Die genaue Beschreibung des Verhaltens eines Datentyps ermöglicht den separaten Test. Da Datentypen aufeinander aufbauen (zur Definition der rationalen Zahlen brauchen Sie die ganzen Zahlen), muss man sich bei dem Test komplexerer Komponenten auf die Korrektheit der elementareren Komponenten verlassen können.

- Wenn die Schnittstelle eines Datentyps unabhängig von seiner Realisierung definiert wurde, kann die konkrete Realisierung geändert werden, ohne dass dies Auswirkungen auf die Verwendung des Datentyps hat. Nur durch eine solche Trennung der einzelnen Komponenten eines großen Systems ist eine vernünftige Weiterentwicklung des Systems möglich.

- Kurz gesagt, eine abstrakte Definition einer Schnittstelle ermöglicht es, dass ein

Nutzer jederzeit zwischen unterschiedlichen Implementierungen frei auswählen kann. Gleichzeitig ermöglicht eine abstrakte Beschreibung, dass eine Implementierung für viele Nutzer brauchbar ist.

Wir können die bisher herausgearbeiteten Begriffe in einer Definition zusammenfassen.

Definition:

*Unter einem **Datentyp** versteht man die Zusammenfassung von Wertebereichen und Operationen zu einer Einheit. Die Realisierung von Datentypen besteht in den beiden formal getrennten Teilen der **abstrakten Schnittstelle** und der **konkreten Repräsentation**. Soweit ein Datentyp nur durch die abstrakte Schnittstelle festgelegt ist, heißt er **abstrakter Datentyp**. Im Unterschied dazu bezeichne ich einen Datentyp, der über eine konkrete Repräsentation verfügt, als **konkreten Datentyp**.*

6.1.1 Die abstrakte Schnittstelle eines Datentyps

Innerhalb eines Computerprogramms muss jede Operation genau festgelegt sein. Wenn man aber Datentypen auf mehrere verschiedenen Arten implementieren kann, so stellt sich die Frage, wie man ihr Verhalten exakt beschreiben soll, ohne auf eine konkrete Implementierung einzugehen. Da sich die Mathematik schon immer mit solchen abstrakten Fragestellungen befasst hat, ist es für die Informatik naheliegend, dort nach brauchbaren Verfahren zu suchen.

Die Grundoperationen der Gleitkommazahlen sind die Rechenoperationen. Ihre Bedeutung ist Ihnen aus dem Mathematikunterricht bekannt. Im Unterricht wurden Ihnen diese Operationen durch anschauliche Beispiele beigebracht. Daneben gibt es in der Mathematik aber auch den Ansatz alle mathematischen Begriffe streng formal zu definieren. Die formale Definition von mathematischen Verknüpfungen ist die Aufgabe der *Algebra*. Dort geschieht die Beschreibung von Operationen dadurch, dass man für jede Operation die *Signatur* angibt, die ausdrückt zu welchen Mengen die Operanden und das Resultat der Operation gehören und dass man das Verhalten der Operationen durch einige grundlegende Gleichungen oder *Axiome* beschreibt. Für die Plusoperation kann man z.B. die folgenden (unvollständigen) Aussagen treffen. Erstens die Signatur:

$$+: \quad \mathcal{R} \times \mathcal{R} \longrightarrow \mathcal{R}$$

und zweitens einige Eigenschaften

$$\begin{aligned} a + (b + c) &= (a + b) + c \\ a + (-a) &= 0 \\ 0 + a &= a \\ a + b &= b + a \end{aligned}$$

Diese Beschreibung ist noch unvollständig, da insbesondere eine umfassende Definition der reellen Zahlen fehlt. Sie drückt aber bereits wichtige Eigenschaften der Addition aus (kommutative Gruppe), mit denen sich viele mathematische Verfahren erklären lassen.

Diese Vorgehensweise, Datentypen auf diese algebraische Art – durch Angabe der Signaturen und der Axiome – zu beschreiben, bezeichnet man auch als *algebraische Spezifikation abstrakter Datentypen*. Bei Anwendungen, bei denen ein großes Gewicht auf die Gewährleistung der Korrektheit der Software gelegt wird, werden solche formalen Verfahren vermehrt eingesetzt. Ich nehme aber an, dass Ihnen dieses streng formale Herangehen der Mathematik zu abstrakt und zu schwierig erscheint.

Das Mathematikbeispiel zeigt, dass man sich fragen muss, wie man abstraktes Verhalten am besten beschreibt. Wenn uns die exakte formale Spezifikation zu kompliziert ist, bleibt uns nur eine *informelle Spezifikation* durch eine möglichst genaue textuelle Beschreibung (etwa als Kommentar) übrig. Diese Beschreibung kann relativ kurz sein, wenn sie sich auf das Vorverständnis des Lesers bezieht. In dieser kurzen Form sind Spezifikationen oft leicht verständlich, haben jedoch auch den Nachteil, dass Missverständnisse nicht ganz auszuschließen sind.

Uns bleiben letztlich verschiedene Möglichkeiten der Festlegung von Datentypen. In den folgenden Abschnitten werde ich Ihnen anhand von Beispielen zeigen, wie Datentypen informell beschrieben werden können und wie die Signatur des Datentyps in Java durch eine Schnittstelle festgelegt werden kann. Doch zunächst möchte ich noch auf ein wichtiges Problem bei der Implementierung eines Datentyps eingehen.

6.1.2 Die konkrete Repräsentation eines Datentyps

Ich habe bisher betont, dass eine sinnvolle Definition eines Datentyps eine möglichst abstrakte Schnittstellenbeschreibung erfordert. Diese Forderung hat auch einige Konsequenzen für die Realisierung eines Datentyps:

- In der Schnittstelle dürfen keine darstellungsabhängigen Begriffe und Größen auftauchen. Für die Realisierung einer Schnittstelle durch Klassen bedeutet dies, dass der direkte Zugriff auf Instanzvariable verhindert werden muss (*Datenkapselung*).
- Die Schnittstelle soll ein einfaches, darstellungsunabhängiges Verhalten zeigen. Das heißt unter Anderem, dass ein einmal definiertes Objekt immer in einem gültigen Zustand ist, auf den jederzeit möglichst alle Elementfunktionen und Operationen anwendbar sind.

Der zweite Punkt ist noch etwas erklärungsbedürftig. Ich will daher am Beispiel von typischen Fehlerquellen bei einfachen Variablen zeigen, worum es mir geht.

Wenn Sie eine einfache Java-Variable verwenden, ist die Forderung nach dem jederzeit gültigen Zustand zum Beispiel dann verletzt, wenn mit der Variablendefinition noch keine Initialisierung verbunden ist. So ist in dem folgenden Beispiel die Verwendung der Variablen `x` korrekt, die von `y` jedoch falsch:

```
int x;
int y;
x = 7;
x = x + y;
```

Hier liegt das Problem darin, dass mit einer Variablendefinition ohne Initialisierung noch kein gültiges Objekt geschaffen wird. Für ein nicht initialisiertes Objekt ist zunächst außer der Wertzuweisung keine andere Operation erlaubt. Deshalb macht es wenig Sinn, Variable zunächst nur zu deklarieren und erst später zu initialisieren.

Aus diesem Grund wird auch bei der Definition einer Klasseninstanz immer ein Konstruktor aufgerufen, der – wenn er richtig definiert ist – das erzeugte Objekt in einen gültigen Zustand versetzt.

Funktionen, die nur den Zustand eines Objekts erfragen, ohne diesen Zustand zu verändern, stellen kein zusätzliches Problem dar. Wenn ein Objekt verändert wird, kann

es jedoch sein, dass danach andere Methoden nicht mehr wie vorgesehen funktionieren (z.B. wenn ich an meinem Auto rumbastele, läuft es nachher nicht mehr).

In einfachen Beispielprogrammen sind Fehler gut zu erkennen. Bei großen Programmen geht jedoch die Übersicht leicht verloren. Da die Fehlererkennung des Compilers bei Fehlern in der Verwendung von Variablen nicht greift, führt dies zu nur schwer zu lokalisierenden Programmfehlern.

Um auch bei großer Software auch noch den Überblick über das Verhalten von Objekten zu behalten, stellt man an die Zustandsveränderung eines Objekts die Forderung, dass sie wieder zu einem gültigen Objektzustand führt. Durch diese Forderung wird sowohl die Benutzung eines Objekts, als auch die Implementierung der einzelnen Methoden erleichtert, da man bei ihrer Beachtung stets davon ausgehen kann, dass beim Methodenaufruf ein gültiger Objektzustand vorliegt.

Natürlich muss irgendwie festgelegt sein, was ein gültiger Zustand ist. Diese Festlegung geschieht durch die *Klasseninvariante* oder *Repräsentationsinvariante*.

Definition:

*Die **Klasseninvariante** einer Klasse ist eine logische Aussage bezüglich der erlaubten Werte der Instanzvariablen der Klasse.*

Ich werde Ihnen den Umgang mit Klasseninvarianten an den kommenden Beispielen noch näher erläutern. Hier will ich zunächst das grobe Schema erklären.

Mit dem Begriff der Klasseninvarianten wird die unterschiedliche Rolle der verschiedenen Arten von Konstruktor und Methoden deutlich:

- Der *Konstruktor* erzeugt aus einem völlig undefinierten Zustand ein Objekt, das der Klasseninvariante genügt.
- Eine *Abfragefunktion* setzt die Gültigkeit der Klasseninvariante voraus. Da die Abfrage den Zustand des Objekts nicht verändert, gilt die Klasseninvariante nach Beendigung der Abfrage automatisch weiter.
- Eine *zustandsverändernde Funktion* setzt ebenfalls die Gültigkeit der Klasseninvariante voraus. Während der Ausführung der Funktion darf die Klasseninvariante zeitweise verletzt sein. Nach der Ausführung muss die Klasseninvariante jedoch unbedingt wieder gelten.

Die Forderung nach der Beibehaltung der Klasseninvariante ist mit der wichtigste Grund für die Forderung nach einer vollständigen Datenkapselung. Wenn Sie nämlich den ungeprüften Zugriff auf die Instanzvariablen freigeben, können Sie innerhalb der Funktionen der Klasse nicht mehr gewährleisten, dass die Klasseninvariante stets erfüllt ist. (Java besitzt mit `synchronized` ein besonderes Sprachkonstrukt, das nur dazu da ist die Beachtung der Klasseninvariante auch in nebenläufigen Programmen (Threads) zu schützen.)

Die Klasseninvariante ist *kein Bestandteil der Schnittstelle*, sondern eine *darstellungsabhängige Aussage zur Gewährleistung eines sinnvollen Verhaltens*. Sie ist also kein Bestandteil eines abstrakten Datentyps, aber ein wichtiger interner Teil einer konkreten Implementierung. Ich werde Ihnen bei den Beispielen dieses Kapitels stets eine Klasseninvariante angeben. Die angeführten Beispiele gehorchen natürlich stets den hier angegebenen Forderungen.

6.2 Sequentielle Datenstrukturen

6.2.1 Die abstrakte Beschreibung von Sequenzen

Unter einer Sequenz versteht man die Realisierung einer Folge von Objekten. Aus der Mathematik kennen Sie Zahlenfolgen, die häufig durch besondere Bildungsregeln definiert sind, wie z.B.:

$$a_0 = 1,\ a_1 = 2,\ a_2 = 4,\ a_3 = 8,\ a_4 = 16,\ \ldots,\ a_i = 2^i$$

Das Typische an einer Folge ist, dass die einzelnen Folgenglieder nummeriert sind und in einer festen Reihenfolge stehen.

Datenstrukturen in prozeduralen und prozedural-objektorientierten Programmiersprachen haben die Eigenschaft,dass man die darin enthaltenen Werte ständig nach Belieben ändern kann, wohingegen mathematische Gebilde unveränderlich sind.[3] Auf der Java Seite kennen Sie Arrays als einen Weg zur Implementierung von Sequenzen. Die wesentliche Verwandtschaft zwischen Folgen und Feldern besteht darin, dass ihre Elemente in einer bestimmten Reihenfolge stehen. Diese Eigenschaft wird meist mit dem Begriff *Sequenz* oder *sequentielle Datenstruktur* benannt.

Definition:

*Eine **Sequenz** ist eine Ansammlung von Datenelementen, die in einer bestimmten (linearen) Reihenfolge stehen.*

Wenn Sie einmal Ihre Erfahrungen im Umgang mit Klassen und mit dem Feldkonzept von Java zusammennehmen, können Sie sofort einige Operationen aufführen, die für Sequenzen unbedingt nötig sind:

- Ein *Konstruktor*.
- Die *Abfrage* und die *Veränderung* einzelner Elemente.
- Das *Einfügen* und das *Löschen* von Elementen.
- Das Erfragen der *Anzahl* der Elemente.

Ich denke mit diesen Operationen haben wir eine ungefähre Vorstellung von dem abstrakten Konzept der sequentiellen Struktur. Im nächsten Abschnitt wird gezeigt, dass man Sequenzen auch anders als durch Arrays realisieren kann.

6.2.2 Alternativen zu Feldern

Felder sind für viele Fälle die effizienteste Implementierung von Sequenzen. Sie sind genauso organisiert wie der Computerspeicher, so dass die Hardware effizient auf beliebige Elemente zugreifen kann. Die hardwarenahe Organisation ist der Grund für die Effizienz. Sie hat aber einige grundsätzlichen Nachteile:

- Die Größe eine Feldes lässt sich zur Laufzeit nicht verändern.

[3] Wegen besserer Ausnutzbarkeit moderner Rechnerarchitekturen kommen aber unverändliche funktionale Datenstrukturen immer mehr in Mode.

- Will man ein neues Element an beliebiger Stelle in die Sequenz einfügen oder aus ihr herausnehmen, muss man alle nachfolgenden Feldelemente verschieben.
- Alle Feldelemente müssen vom gleichen Datentyp sein.[4]
- Die Inhalte eines Feldes gehen beim Programmende verloren. Felder eignen sich nicht dazu Daten permanent zu speichern.

Neben Arrays sind die beiden wichtigsten sequentiellen Datenstrukturen *verkettete Listen* und *Dateien*. Leider gibt es nicht die ideale Datenstruktur, so dass jede ihre besonderen Vor- und Nachteile hat:

- Die Nachteile von Feldern wurden eben bereits dargestellt. Ihr Vorteil ist der effiziente Zugriff auf das n-te Element. Felder ermöglichen zudem eine sehr gute Speicherausnutzung, wenn die Anzahl der Datenelemente vorab bekannt ist.
- Der Vorteil von verketteten Listen besteht darin, dass sich zur Laufzeit sehr einfach die Organisation (Einfügen und Löschen) und die Größe einer Liste verändern lässt. Um auf ein bestimmtes Element zuzugreifen, muss man jedoch vom Anfang her alle davor liegenden Elemente durchlaufen, da von jedem Element aus immer nur ein Verweis auf das nachfolgende Element gegeben ist.
- Bei Feldern und Listen geht der Inhalt beim Programmende verloren.
- Dateien dienen in erster Linie dazu Daten permanent zu speichern. Dazu werden Dateien auf externen Medien (Festplatte) gespeichert. Der Zugriff auf externe Speichermedien ist relativ langsam. Die effiziente Ausführung und Organisation von Dateizugriffen ist Sache des Betriebssystems. Das Betriebssystem ist dabei unter anderem auch für die Gewährleistung der Datensicherheit und des Zugriffsschutzes zuständig.

6.2.3 Verkettete Listen

Die Effizienzvorteile von Feldern und ihre mangelnde Flexibilität kommen beide daher, dass bei Feldern die Reihenfolge der Elemente *implizit* in der Reihenfolge der Speicherzellen ausgedrückt wird. Es ist kein besonderer Aufwand nötig um auszudrücken, dass `a[3]` auf `a[2]` folgt, da durch den Compiler dafür gesorgt wird, im Speicher unmittelbar aufeinander folgen. Diese implizite Codierung der Reihenfolge vermeidet Speicher- und Laufzeitnachteile. Gleichzeitig ist es dadurch aber schwierig neue Feldelemente einzufügen.

Es gibt eine sehr große Zahl von Anwendungen, in denen die Nachteile des Einfügens gar keine oder nur eine geringe Rolle spielen. Denken Sie nur an mathematische Aufgabenstellungen, in denen Felder zur Darstellung von Vektoren und Matrizen verwendet werden.

Auf der anderen Seite gibt es aber auch viele Computerprogramme, in denen die dynamische Veränderbarkeit der Datenstruktur sehr wichtig ist. Ein Beispiel sind die Systeme der Computergraphik, in denen geometrische Objekte durch Listen von geometrischen Grundelementen beschrieben werden und in denen ganze Szenen wiederum aus Listen

[4] Bei Objekten gilt diese Forderung nicht so streng. Hier müssen die Inhalte aber immerhin mit dem Basistyp des Feldes verträglich sein.

von Objekten aufgebaut sind. Solche Systeme bestehen aus einer Vielzahl von Datenstrukturen, deren Größe und Struktur sich während des Programmablaufs permanent verändert.

Bei der Implementierung von Sequenzen durch verkettete Listen wird die Reihenfolge der Elemente *explizit* ausgedrückt. Entsprechend der Terminologie der Graphentheorie (siehe dazu auch den Abschnitt über Binärbäume) nennt man die Datenelemente einer verketteten Liste auch *Knoten* und die Verbindungen, die die Reihenfolge der Knoten beschreiben, *Kanten*. Obwohl es grundsätzlich auch andere Möglichkeiten gibt, werden Kanten meist durch die Speicheradressen der Zielknoten dargestellt. In Java sind dies die Objektreferenzen. Da Kanten zwei Knoten verbinden, nennt man sie oft auch *Verkettung*, *Zeiger* oder *Link*.

Das kennzeichnende Merkmal einer verketteten Liste ist die Anreihung ihrer Elemente in Form einer Kette. Im Minimalfall ist jedes Listenelement ausschließlich vom Anfang der Liste, dem *Kopf* der Liste, erreichbar. Diese Form der Liste heißt *einfach verkettete Liste*. Dabei enthält jeder Listenknoten die Referenz des unmittelbar nachfolgenden Knotens. Durch diese Art der Datenorganisation ist es möglich, jeden Knoten ausgehend vom Listenanfang zu erreichen. Diese „Bewegung“ durch die Liste kann bei der einfach verketteten Liste nur in einer Richtung erfolgen. Es gibt jedoch auch die Form der *doppelt verketteten Liste*, in der jeder Knoten zusätzlich die Referenz seines Vorgängerknotens enthält und in der auch „Rückwärtsbewegungen“ einfach ausführbar sind.

Anmerkung:

In Programmiersprachen, die Funktionale Programmierung unterstützen, findet man eine besonders effiziente Implemntierung von unveränderlichen einfach verketteten Listen. Dabei wird eine Liste durch die Referenz des Anfangsknotens dargestellt. Jeder Listenknoten enthält einen Wert (*head*) und eine Restliste (*tail*). Solche Sprachen sind *Ruby*, *Scala*, *Clojure* und viele andere.

Die Abbildung 6.1 veranschaulicht den Aufbau von verketteten Listen Die eigentlichen Datenelemente der Liste (die Knoten) sind durch Ovale dargestellt. Die Pfeile symbolisieren Referenzen (die Kanten), mit denen die Verkettung der Datenelemente ausgedrückt wird. Bei der einfach verketteten Liste enthält das eigentliche Listenobjekt eine Referenz auf das erste Knotenobjekt. Dieses enthält die Referenz auf den nachfolgenden Knoten und so weiter. Zur Optimierung des Zugriffs auf des letzte Listenelement enthält das Listenobjekt manchmal auch die Referenz des letzten Elements (hier gestrichelt gezeichnet). Die untere Abbildung zeigt die Struktur der doppelt verketteten Liste.

Implementierung des Listentyps durch eine Klasse

Nachdem ich Ihnen eine erste Vorstellung von verketteten Listen vermittelt habe, will ich jetzt konkreter werden, indem ich zunächst eine Klasse `SinglyLinkedList`[5] entwickle, die wichtige Grundfunktionen zur Verfügung stellt. Anschließend gehe ich kurz auf mögliche Anwendungen dieser Klasse ein.

Ich will mit Ihnen hier nur die einfachste Form von einfach verketteten Listen besprechen. Selbst dabei muss ich mich auf einige wenige Grundoperationen beschränken. „Richtige“ Listenklassen enthalten sehr viel mehr Operationen. Die Vielzahl von Listenfunktionen ist für die flexible Verwendbarkeit des Listentyps notwendig, bei ihrer Implementierung werden jedoch keine Grundmuster verwendet, die nicht auch hier erklärt werden.

[5] In der Javaklassenbibliothek beschreibt die Schnittstelle `java.util.List` das allgemeine Konzept der sequentiellen Anordnung; die Klasse `java.util.LinkedList` ist eine fertige Implementierung.

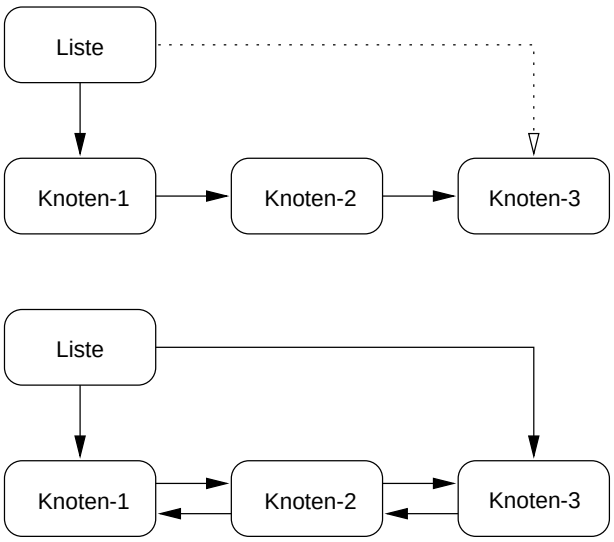

Abbildung 6.1: Oben ist eine einfach verkettete Liste und unten eine doppelt verkettete Liste dargestellt. Doppelt verkettete Listen erleichtern und beschleunigen einige Listenoperationen.

Hinweis:
Die Implementierung von Listenoperationen ist ein beliebtes Klausurthema!

Die Operationen lassen sich in vier Kategorien einordnen:

1. Jeder Datentyp verfügt über Operationen zum Erzeugen von Datenelementen. Zu diesen *Standardoperationen* zählen auch Funktionen zum Kopieren eines Objekts und zum Vergleich zweier Listenobjekte (`equals`). Hier will ich nur den parameterlosen Konstruktor definieren.

2. Die wichtigste Aufgabe einer Liste besteht darin Datenelemente zu speichern. Es muss daher Operationen zum *Einfügen* neuer Elemente und zum *Löschen* bestehender Elemente geben.

3. Es soll Methoden geben, mit denen sich der aktuelle Zustand der Liste erfragen lässt.

4. Schließlich muss die Möglichkeit bestehen, alle Listenelemente nacheinader abzuarbeiten. Dies wird in Java durch das Iterator-Konzept gelöst. Ich werde auch erklären, wie es möglich ist, die Iteration durch Listen mit der Foreach-Schleife durchzuführen.

Der Zugriff auf ein Element, das – ähnlich wie bei einem Feld – an einer bestimmten Position steht, wurde bewusst weggelassen. Er ist nicht schwer zu implementieren, aber er wird relativ langsam sein, da zum Auffinden des n-ten Listenelements nacheinander genau n Knoten aufgesucht werden müssen, um an die richtige Stelle zu kommen.[6]

Die Signaturen der Methoden der Klasse `SinglyLinkedList` orientieren Sie sich an der Klasse `LinkedList` aus dem Paket `java.util`. Dort finden sich neben vielen weiteren Methoden auch solche für den direktem Zugriff auf das n-te Listenelement.

[6] Auch dies könnte ein Klausurthema sein.

Wie die Abbildung 6.1 nahelegt, besteht die Listenabstraktion aus zwei eng miteinander verbundenen Klassen (später kommt noch eine dritte Klasse dazu), der Klasse `SinglyLinkedList` zur Repräsentation der gesamten Liste und der Klasse `SinglyLinkedList.Node` zur Repräsentation eines einzelnen Knotens. Die Klasse `Node` dient ausschließlich dazu, die Listenwerte zu speichern. Die Klasse `Node` soll daher außerhalb der Klasse `SinglyLinkedList` nicht verfügbar sein.

Dies können wir dadurch erreichen, dass wir `Node` als private geschachtelte Klasse implementieren. Wegen der engen Verbindung zur Klasse `SinglyLinkedList` formuliere ich für die Klasse `Node` keine eigene Klasseninvariante.

Hinweis: **Die folgenden Implementierungsbeispiele verwenden (im Unterschied zu den Klassen der Javabibliothek) keine Typparameter.**

```
/** Node dient zum Verwalten der Eintraege
 */
private static class Node {
    Node(Object value, Node next) {
        this.value = value; this.next = next;
    }
    Object value;
    Node next;
}
```

Die Schnittstelle der Klasse `SinglyLinkedList` ist einfach anzugeben. Der einzige Dateninhalt eines `SinglyLinkedList`-Objekts ist die in der privaten Instanzvariablen `first` gespeicherte Referenz des ersten Listenknotens. Die Funktion `iterator` wird später besprochen.

```
public class SinglyLinkedList {

    /** Default Konstruktor
     */
    public SinglyLinkedList()

    /**
     * Prueft ob die Liste leer ist.
     * @return true, wenn Liste leer
     */
    public boolean isEmpty()

    /**
     * Fuegt ein Element am Anfang der Liste hinzu.
     * @param obj einzufuegendes Objekt
     */
    public void addFirst(Object obj)

    /**
     * Gibt das erste Listenelement zurueck.
     * @return 1. Element
     * @throws NoSuchElementException wenn Liste leer
     */
    public Object getFirst()

    /**
     * Entfernt 1. Element aus der Liste.
     * @return das erste Listenelement
     * @throws NoSuchElementException wenn Liste leer
     */
```

```
    public Object removeFirst()

    /**
     * Prueft ob ein Element ein bestimmtes Element
     * in der Liste vorhanden ist
     * @param obj zu suchendes Objekt
     * @return true, wenn gefunden
     */
    public boolean contains(Object obj)

    /**
     * Fuegt ein Element am Ende der Liste hinzu
     * @param obj einzufuegendes Objekt
     */
    public void addLast(Object obj) {

    /** Gibt einen Iterator auf die Liste zurueck
     */
    public Iterator iterator()

    /**
     * Loescht das erste Vorkommen von obj aus der liste.
     * @param obj zu loeschendes Objekt
     * @return true wenn Einrag entfernt wurde; sonst false
     */
    public boolean remove(Object obj)
}
```

Ehe ich mit der Besprechung der Implementation beginne, will ich eine Klasseninvariante formulieren, die die genaue Bedeutung der Instanzvariablen `first` festlegt.

Zunächst erscheint es klar, dass die Variable `first` auf das erste Listenelement zeigt und dass in jedem Knoten (Klasse `Node`) die Variable `next` auf das nachfolgende Element zeigt.

Aber was ist, wenn es kein nachfolgendes (oder keine erstes) Element gibt? Hier sind in der Praxis tatsächlich verschieden Lösungen üblich:

- Es erscheint am Einfachsten, einfach den Wert `null` einzusetzen, wenn man ausdrückt, dass es keinen weiteren Knoten gibt. Wegen ihrer Einfachheit werde ich hier auch diesen Weg gehen. Allerdings ist er auch nicht ohne Nachteile. Da `null` keine Methoden „versteht“, zwingt es dazu, auf viele Sonderfälle zu achten.
- Wir können einen besonderen Knotentyp (z.B. namens `Nil`) für den Endknoten vorsehen. Das ist insbesondere dann eine sehr gute Lösung, wenn Knoten selbst über eigene Methoden verfügen. Viele (streng) objektorientierte Sprachen, setzen auf diese Idee.
- Wir können einen kreisförmige (zirkuläre) Liste aufbauen, bei der der letzte Knoten wieder auf den ersten Knoten zeigt. Dazu benötigen wir aber auch für den Fall einer leeren Liste einen solchen Knoten. Kurz man verwendet einfach einen „Kopfknoten“ der vor dem ersten Listenknoten steht, auf den er verweist. Gleichzeitig zeigt der letzte Knoten auf den Kopfknoten. Diese Variante ist in der Implementierung der Klasse `LinkedList` in der Java-Bibliothek gewählt. Sie vermeidet praktisch alle Sonderfälle.

Die Diskussion zeigt, dass es nötig ist bei der Implementierung von Datenstrukturen gilt, einige grundsätzliche Entscheidungen zu treffen. Nicht nur die Effizienz sondern auch der

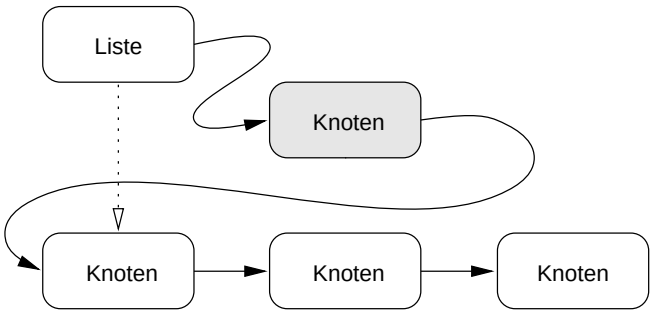

Abbildung 6.2: Beim Einfügen eines neuen Knotens (grau unterlegt) am Anfang einer Liste muss der Knoten richtig „eingekettet“ werden. Genau der umgekehrte Vorgang läuft beim Löschen des ersten Knotens ab: Hier wird der Anfangszeiger auf den zweiten Knoten gesetzt.

Programmmieraufwand hängen von diesen Festlegungen ab. Auf jeden Fall sollte man die Entwurfsentscheidungen dokumentieren. Sie sind bei der Realisierung einer jeden Methode zu beachten!

Klasseninvariante:

In der Klasse `SinglyLinkedList` *gilt:* `first = null`*, oder aber* `first` *enthält die Referenz des ersten Listenknotens. Dessen* `next`*-Feld enthält die Referenz des nächsten Knotens usw. Der letzte Knoten ist dadurch gekennzeichnet, dass sein* `next`*-Feld gleich* `null` *ist. Die Felder* `value` *der Listenknoten können beliebige Objektreferenzen enthalten. Diese spielen für die Implementierung der Listenoperationen keine Rolle.*

Wir haben damit eine zunächst einfache aussehende Wahl getroffen. Wir sollten uns aber im Klaren sein, dass die Festlegung von `null` für das Listenende uns immer wieder zur Beachtung von Sonderfällen zwingt. Dies liegt einfach darin, dass `null` eben kein Objekt ist.

Merksatz:

Wenn in einer Datenstruktur oder in einem Algorithmus `null` *eine wichtige Rolle spielt, gilt es stets die Sonderrolle der* `null` *zu beachten.*

Die Aufgabe des Konstruktors besteht darin die einfachste gültige Liste zu erzeugen. Dies ergibt eine Variablenbelegung, die der leeren Liste entspricht. Wie Sie an der Klasseninvariante sehen, brauchen wir dazu nur `first` auf `null` zu setzen. Gleichzeitig ist die Funktion `isEmpty` durch die Invariante vollständig bestimmt.

```
private Node first = null;

/**
 * Prueft, ob die Liste leer ist
 * @return true, wenn Liste leer
 */
public boolean isEmpty() {
    return first == null;
}
```

Die Abbildung 6.2 verdeutlicht das durch die Funktion `addFirst` durchzuführende Einfügen eines neuen Elements am Anfang der Liste. Diese Aufgabe können Sie dadurch

lösen, dass Sie zunächst einen neuen Knoten erzeugen, darauf dessen `next`-Feld auf den (bisher) ersten Knoten zeigen lassen (die Referenz findet sich in `first`) und schließlich in der Variablen `first` die Referenz des neuen Knotens speichern. Bei der Erzeugung des neuen Knotens nutzen wir den Konstruktor der Klasse `Node`.

```
/**
 * Fuegt ein Element am Anfang der Liste hinzu
 * @param obj einzufuegendes Objekt
 */
public void addFirst(Object obj) {
    first = new Node(obj, first);
}
```

Lassen Sie uns prüfen, ob die Invariante erfüllt ist. Laut dieser enthält `first` vor dem Einfügen entweder den Wert `null` oder die Referenz auf den ersten Knoten. Im Fall der leeren Liste wird der neu eingefügte Knoten zum letzten Knoten. Er enthält also hier den Wert `null`. Andernfalls enthält der die Referenz auf den zuvor ersten Knoten, der jetzt zu seinem Nachfolgeknoten wird. Schließlich zeigt in beiden Fällen `first` am Ende auf den ersten Knoten.

Entgegen der Vorbemerkung wegen den Problemen mit `null` brauchten diese bisher nicht beachtet zu werden. Bei `addFirst` wird ja höchstens die `null` in den neuen Knoten kopiert. Das ist ja auch so gewollt, wenn die Liste vorher leer war. Bei den weiteren Algorithmen sollten wir aber immer an mögliche Sonderfälle denken.

Es gibt einerseits die Funktion `getFirst` zur Abfragen des ersten Elements einer Liste und die Funktion `removeFirst` zum Entfernen des ersten Elements. Mittels `first` ist der Zugriff auf das erste Listenelement ganz einfach. Die Funktion `removeFirst` führt in umgekehrter Reihenfolge die Aktionen beim Einfügen eines Knotens aus. Vergleichen Sie wieder die Abbildung 6.2.

```
/**
 * Gibt das erste Listenelement zurueck.
 * @return 1. Element
 * @throws NoSuchElementException wenn Liste leer
 */
public Object getFirst() {
    if (first == null) return new NoSuchElementException();
    return first.value;
}
/**
 * Entfernt 1. Element aus der Liste.
 * @return das erste Listenelement
 * @throws NoSuchElementException wenn Liste leer
 */
public Object removeFirst() {
    if (first == null) throw new NoSuchElementException();
    Object result = first.value;
    first = first.next;
    return result;
}
```

Vergleichen wir wieder die Invariante. Es ist nicht gesagt, dass es einen ersten Knoten gibt. Wenn die Liste leer ist, können wir auch keinen ersten Wert zurückgeben. Wir legen fest, dass eine Ausnahme geworfen wird.

Da wir bei `getFirst` nichts ändern, brauchen wir auch nichts weiter zu beachten. Wir

wissen aber, dass in `first.value` die erste Objektreferenz der Liste steckt.

Bei `removeFirst` soll der erste Knoten entfernt werden. Wir erreichen dies, indem wir `first` einfach auf den Nachfolgeknoten, des zu entfernenden Knotens setzen. Dies ist gemäß der Invariante der zweite Listenknoten oder `null`. Um den nicht mehr verwendeten Listenknoten brauchen wir uns – wie Sie wissen – nicht zu kümmern. Sobald der Knoten nicht mehr über eine Referenz erreichbar ist, kann die virtuelle Maschine bei Bedarf den Speicherplatz wieder verwenden.

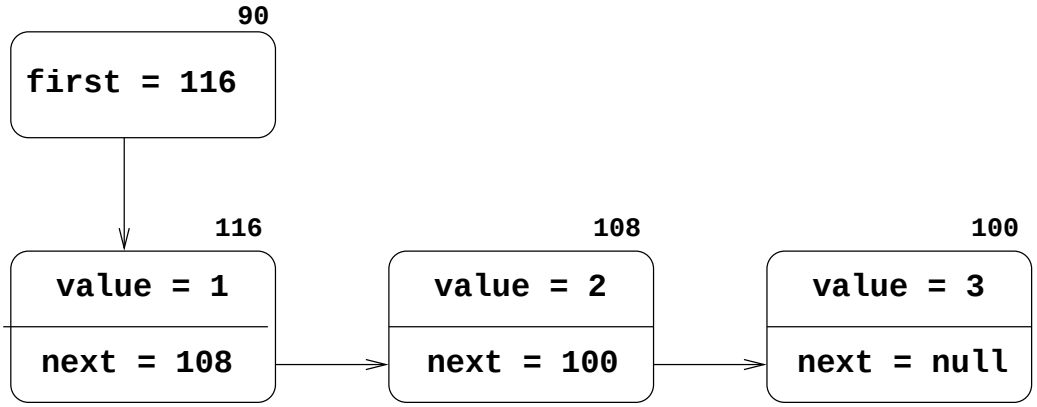

Abbildung 6.3: In dieser Abbildung ist beispielhaft die Repräsentation der Liste der Zahlen 1, 2, 3 dargestellt. Die am linken oberen Rand der Objekte angegebenen Speicheradressen sind willkürlich gewählt.

Bei einer einfach verketteten Liste ist nur das erste Listenelement *unmittelbar* verfügbar. Sobald andere Elemente angesprochen werden, oder wenn an anderer Stelle ein Element eingefügt werden soll, muss zuerst die richtige Stelle gesucht werden. Das Schema ist bei allen Listenoperationen gleich, dass nämlich in einer Schleife eine Hilfsvariable sukzessive auf aufeinander folgende Knoten gesetzt wird. Das Grundmuster kann sowohl durch eine While-Schleife als auch durch eine For-Schleife realisiert werden.[7]

```
// 1. While-Schleife
Nodep = first;
while (p != null) {
    // tue was mit p
    p = p.next;
}

// 2. For-Schleife
for (Node q = first; q!=null; q=q.next) {
    // tue was mit q
}
```

Bevor ich den Algorithmus zur Suche nach einem Konten hinschreibe, will ich noch eine Hilfsfunktion angeben, die wir zum sicheren Prüfen auf Gleichheit verwenden können.

```
/**
 * Vergleicht zwei Objekte auf Identitaet.
 * <tt>null</null> wird korrekt behandelt.
 * @param a erstes Objekt
 * @param b zweites Object
```

[7] Natürlich kann man die While-Schleife durch Rekursion ersetzen.

```
  * @return <tt>true></tt> bei Gleichheit
  */
private static boolean eq(Object a, Object b) {
    return a == null ? b == null : a.equals(b);
}
```

Die Funktion `contains` ist ein typisches Beispiel für die Verwendung dieses Schemas. Dabei wird das Grundmuster geringfügig abgeändert, da ja nach dem Auffinden des gesuchten Wertes die Suche abgebrochen werden kann.

Ich gebe zu, dass das nicht die effizientes Lösung ist. Weiter oben hatte ich ja bereits eine andere Lösung angegeben. Es kann auch sein, dass ich gelegentlich den korrekten Umgang mit `null` vergesse. Es ist halt wirklich ein Problem! In vielen Fällen behilft man sich auch damit, dass man (per Kommentar) `null` als Inhalt verbietet.

```
/**
 * Prueft ob ein Element ein bestimmtes Element
 * in der Liste vorhanden ist.
 * @param obj zu suchendes Objekt
 * @return true, wenn gefunden
 */
public boolean contains(Object obj) {
    Node p = first;
    while(p != null && !eq(p.value, obj))
        p = p.next;
    return p != null;
}
```

Beachten Sie zunächst wie mit der Funktion `eq` das Problem des Vergleichs mit `null` gelöst wird.

Machen Sie sich anhand der Abbildung 6.3 klar, wie die Funktion `contains` funktioniert. Angenommen die Aufgabe besteht darin festzustellen, ob die Zahl 2 in der Liste enthalten ist. Der Parameter `value` hat also den Wert 2. Durch die Initialisierung erhält die Zeigervariable `p` den anfänglichen Wert 116, sie „zeigt“ damit auf den Knoten mit dem Wert 1. Die Schleifenbedingung in der While-Anweisung ergibt den Wert `true`, da `p != null` erfüllt ist und außerdem auch `value` nicht gleich dem Knotenwert des Objekts bei 116, nämlich 1, ist. Als nächstes wird die Schleifenanweisung ausgeführt, die dem Zeiger `p` den neuen Wert 108 zuweist. Der erneute Test der Schleifenbedingung ergibt `false` (Wert gefunden), die Schleife wird verlassen, und da der Zeiger `p` ungleich `null` ist, lautet der Resultatwert der Funktion `true`.

Wenn Sie dagegen versuchen festzustellen, ob die Zahl 4 in der Liste enthalten ist, so wird auch der zweite Test der Schleifenbedingung `true` ergeben. Die Variable `p` erhält im Schleifenkörper den neuen Wert 100, und nach einem weiteren Test wird `p` schließlich auf den Wert `null` gesetzt. Erst danach wird die Schleife verlassen, da jetzt der erste Teil der Schleifenbedingung (Und-Verknüpfung!) nicht mehr erfüllt ist. Da jetzt `p` gleich `null` ist, ist der durch die Return-Anweisung bestimmte Resultatwert gleich `false`.

Bei einigen Listenfunktionen tritt ein kleines Problem auf, dass ich Ihnen am Beispiel der `addLast`-Funktion verdeutlichen will. Ein erster schematischer Ansatz könnte so aussehen:

```
void addLast (Object value) {
    // letzten Knoten suchen
    Node p = first;
```

```
    while (p != null) {
        p = p.next;
    }
    // Problem:
    //    jetzt ist der letzte Knoten bereits
    //    vorbei !!
    ???
}
```

In dem Beispiel können Sie erkennen, dass die gesamte Liste durchlaufen wird und damit auch das letzte Listenelement gefunden wird. Die While-Schleife wird aber erst verlassen, nachdem die Zeigervariable `p` den Wert `null` erhalten hat (daran erkennt man das Listenende) und dadurch wieder die Speicheradresse des letzten Knotens „vergessen“ hat. *Eine* Lösung des Dilemmas, dass diese schematische Suche über das Ziel hinausschießt, besteht darin, dass die Schleifenbedingung so abgeändert wird, dass „vorausschauend“ das Ende der Liste gesucht wird:

```
Node p = first; // Vorsicht: ist p != null?
while (p.next != null) {
    p = p.next;
}
```

Dies ist oft die einfachste Lösung. Man muss dabei jedoch darauf achten, dass *vor* Eintritt in die Schleife abgefragt werden muss, ob die Schleife überhaupt ein Element enthält. In dem nicht auszuschließenden Fall (beachten Sie die Invariante der Klasse `List`!), dass `first` gleich `null` ist, wird das Programm sonst abstürzen.

```
/**
 * Fuegt ein Element am Ende der Liste hinzu.
 * @param obj einzufuegendes Objekt
 */
public void addLast(Object obj) {
    Node newNode = new Node(obj, null);
    if(first == null) {
        // noch kein Element vorhanden
        // somit wird dies der erste Eintrag
        first = newNode;
    }
    else {
        // letzten Knoten suchen
        Node p = first;
        while(p.next != null)
            p = p.next;
        // ans Ende anhaengen
        p.next = newNode;
    }
}
```

Die zweite Idee zur Realisierung von `addLast` besteht darin die While-Schleife zunächst so zu übernehmen wie bisher. Da die Schleife über das Ziel hinausschießt, verwenden wir eine zweite Variable `q`, die immer einen Schritt hinter `p` „hinterherläuft“. Wenn wir den Sonderfall der leeren Liste separat behandeln, können wir für die Suche nach dem letzten Element wieder bei dem 2. Knoten anfangen. `q` kann also zunächst auf den ersten Knoten zeigen.[8]

[8] Bei anderen Algorithmen oder bei einer anderen Formulierung des Algorithmus kann man aber auch mit

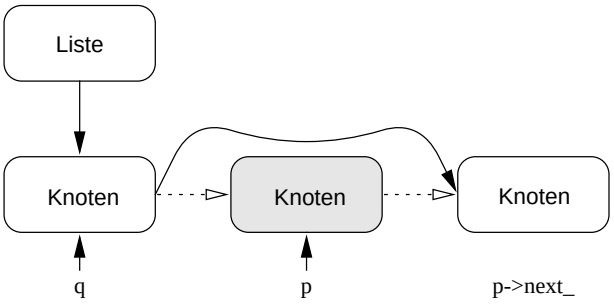

Abbildung 6.4: Der Knoten, auf den der Zeiger `p` zeigt, soll entfernt werden. Dazu wird der `next`-Zeiger des Vorgängerknotens `q` so „umgebogen“, dass er auf den Nachfolger von `p` zeigt.

Diese `addLast`-Funktion sieht so aus:

```
public void addLast(Object obj) {
    Node newNode = new Node(obj, null);
    if(first == null)
        first = newNode;
    else { // first != null
        Node q = first;
        Node p = first.next;
        while(p != null) {
            q = p;
            p = p.next;
        }
        q.next = newNode;
    }
}
```

Genau das gleiche Schema können Sie verwenden, wenn Sie eine Funktion schreiben, die *vor* einem bestimmten Knoten einen neuen Knoten einfügt oder wenn Sie einen Knoten mit einem vorgegebenen Inhalt löschen wollen.

Machen Sie sich das folgende Programmstück mit Hilfe der Abbildung 6.4 klar, die den Normalfall des Löschens eines Knotens darstellt, der nicht der erste Knoten der Liste ist. Das Löschen des ersten Knotens muss nämlich wieder etwas anders behandelt werden. Genauso wie beim Anfügen wird dabei anstelle von `q` die Instanzvariable `first` verändert.

```
/**
 * Loescht das erste Vorkommen von obj aus der liste.
 * @param obj zu loeschendes Objekt
 * @return true wenn Einrag entfernt wurde; sonst false
 */
public boolean remove(Object obj) {
    if (first != null) {
        if (eq(first.value, obj) {
            first = first.next;
            return true;
        }
        Node q = first;
        Node p = first.next;
        while(p != null) {
            if (eq(p.value, obj) { // gefunden
```

`q = null` anfangen.

```
                q.next = p.next;
                return true;
            }
            q = p;
            p = p.next;
        }
    }
    return false;
}
```

Wir haben jetzt die Grundfunktionen von Listen auf der Basis von einer *einfach verketteten Liste* besprochen. In der Praxis wird mindestens ebenso häufig die *doppelt verkettete Liste* verwendet. Sie hat zwar den Nachteil, dass der Speicherbedarf pro Listenknoten etwas größer ist und dass bei Veränderungen immer die doppelte Anzahl von Zeigern verändert werden muss, andererseits gestaltet sich aber das Löschen eines Elements erheblich einfacher. Die Listenklasse `LinkedList` der Java-Bibliothek verwendet eine Variante der doppelt verketteten Liste.

Die Java-Bibliothek ist so strukturiert, dass sie zunächst die möglichen Listenoperationen in der Schnittstelle `List` festlegt und dann unterschiedliche Implementierungen anbietet. Darunter ist auch `ArrayList` als Implementierung auf der Basis eines Arrays (das bei Bedarf vergrößert wird). Die Listenklassen bieten grundsätzlich alle die gleiche Funktionalität. Die Entscheidung für eine bestimmte Variante sollte davon abhängen, welche Operationen am häufigsten vorkommen (einerseits dynamische Erweiterung andererseits Zugriff auf ein bestimmtes Element).

Eine mögliche Variante der Implementierung einer doppelt verketteten Liste, verwendet einen zusätzlichen Knoten, der gleichzeitig als Anfang und Ende der Liste dient (die Liste ist dann praktisch kreisförmig angeordnet). Diese Variante wird auch in der Java-Bibliothek verwendet. Die wichtigsten Deklarationen sehen wie folgt aus:

```
public class DoublyLinkedList {
    private static class Node {
        private Object value;
        private Node next;
        private Node previous;
    }

    private Node head;

    public DoublyLinkedList() {
        head = new Node();
        head.next = head;
        head.previous = head;
    }
```

Versuchen Sie selbst einmal, die Listenoperationen anders als hier dargestellt selbst zu implementieren. Dabei sollten zyklische und doppelt verkettete Listen keine große Herausforderung sein.

Anwendungen von Listen

In den einleitenden Überlegungen zu sequentiellen Datenstrukturen bin ich bereits auf die Vor- und Nachteile von Listen eingegangen. Abgesehen von der Effizienz der Datenzugriffe können Listen grundsätzlich überall anstelle von Feldern verwendet werden. Der

wichtigste Vorteil von verketteten Listen gegenüber Feldern besteht darin, dass die Anzahl der Elemente nicht beschränkt ist.

In dem folgenden Beispiel beschreibe ich, wie Sie mit der Klasse `SinglyLinkedList` das einfache Problem lösen können den Mittelwert von einer Anzahl ganzer Zahlen zu bilden. Allerdings ist das zunächst nur die Motivation für die noch zu beschreibende bessere Lösung mittels Iterator.

```
public static SinglyLinkedList einlesen() {
    SinglyLinkedList liste = new SinglyLinkedList();
    while(true) {
        System.out.print("Bitte geben Sie eine Zahl ein" +
            " (0 zum Beenden) :");
        int n = in.nextInt();
        if (n == 0) break;
        liste.addLast(Integer.valueOf(n));
    }
    return liste;
}

/** Bildet den Mittelwert aus den Daten einer Liste
 */
public static double mittelwert1(SinglyLinkedList liste) {
    int summe = 0;
    int anzahl = 0;
    while (!liste.isEmpty()) {
        summe += (Integer) liste.removeFirst();
        anzahl++;
    }
    return (double) summe/(double) anzahl;
}

public static void main (String[] args) {
    SinglyLinkedList lst = einlesen();
    System.out.println("Mittelwert1: " + mittelwert1(lst));
}
```

Das Beispiel sollte nicht schwer zu verstehen sein.[9] Durch die „Verpackung“ der Listenoperationen in einer Klasse brauchen Sie zum Verständnis des Programms nicht zu wissen, was bei den einzelnen Listenoperationen genau passiert.

6.2.4 Iteratoren

Wenn Sie das Beispiel genau durchlesen, werden Sie feststellen, dass die Funktion `mittelwert` nicht optimal arbeitet. Bei diesem Algorithmus wird die eingegebene Liste bei der Mittelwertbildung zerstört. Leider gibt es in *unserer* Klasse `SinglyLinkedList` keine andere Möglichkeit, um an die Werte der Liste heranzukommen. Daran können Sie sehen, dass die hier vorgestellte Klasse noch bei weitem nicht den Ansprüchen an eine gut gestaltete Schnittstelle genügt. Die fehlenden Teile betreffen insbesondere auch die Möglichkeit des flexiblen Ansprechens der einzelnen Datenelemente.

Wie ich einleitend bemerkt habe, ist eine verkettete Liste nur ein Sonderfall der umfassenden Gruppe von sequentiellen Datenstrukturen. Sammlungen moderner Entwurfsmuster

[9] Wenn wir eine Operation `size()` für die Anzahl der Listenelemente hätten, bräuchten wir diese jetzt nicht zu zählen.

enthalten für lineare Datenstrukturen ein allgemeines Iterator-Konzept, dass man anwenden kann ohne etwas über die konkrete Realisierung der Datenstruktur zu wissen.

Schauen Sie sich diesen Mechanismus an unserem Mittelwert-Beispiel an:

```
/** Bildet den Mittelwert mit Iterator
 */
public static double mittelwert2(List liste) {
    int summe = 0;
    int anzahl = 0;
    for (Iterator iter = liste.iterator(); iter.hasNext();) {
        Integer i = (Integer)iter.next();
        summe += i.intValue();
        anzahl++;
    }
    return (double) summe/(double) anzahl;
}
```

Durch die Funktion `iterator` wird ein Objekt erzeugt, das dazu verwendet werden kann, alle Elemente einer Liste nacheinander aufzusuchen. Es verfügt über die Funktionen `hasNext` um zu prüfen, ob man bereits am Ende der Liste ist, und über die Funktion `next`, die einen Verweis auf das laufende Element zurückgibt und gleichzeitig zum nächsten Element weiterschaltet. Das sonst in einer For-Schleife übliche Erhöhen des Schleifenzählers im dritten Teil des Schleifenkopfes (`i++`) entfällt hier.

Iteratoren gehören nach diesen Vorüberlegungen unbedingt zu allgemein verwendbaren Behälterklassen. Das folgende Listing zeigt, was Sie tun müssen um unsere Listenklasse um einen Iterator zu erweitern.

```
/**
 * Beispiel-Implementation einer Iterator Klasse
 */
class ListIterator implements Iterator {
    private Node currentNode; // aktueller Knoten

    /**
     * Erzeugt ein Iterator-Objekt.
     * @param firstNode Referenz des ersten Knotens
     */
    ListIterator(Node firstNode) {
         currentNode = firstNode;
    }

    /**
     * Gibt die Referenz auf den Inhalt des
     * aktuellen Knotens zurueck und rueckt einen
     * Knoten vor.
     * @return aktueller Knotenwert
     */
    public Object next() {
        if (node == null)
            throw new NoSuchElementException();
        Object result = node.value;
        node =  node.next;
        return result;
    }

    /** Gibt es noch weitere Knoten ?
     * @return true wenn ja
     */
```

```
    public boolean hasNext() {
        return node != null;
    }

    public void remove(){
        throw new UnsupportedOperationException();
    }
}
```

Die Methode `remove` wurde nicht implementiert. Sie ist bei der einfach verketteten Liste etwas komplizierter und hier auch nicht so wichtig. Die Schnittstelle `Iterator` erlaubt dies, wenn man die angegebene Ausnahme wirft.

Die Bibliotheksklasse lässt sich mit Typparametern und unter Ausnutzung des Autoboxing noch einfacher verwenden. Zusätzlich wird auch die For-Each-Schleife verwendet. Der wirkliche Mechanismus hinter dieser Syntax ist allerdings dann auch wieder die Iteratorklasse.

```
import java.util.List;
import java.util.LinkedList;

public class BeispielFuerGenericList {
    public static List<Integer> einlesen() {
        List<Integer> liste =
            new LinkedList<Integer>();
        while(true) {
            System.out.print("Bitte geben Sie eine Zahl ein"
                + " (0: Ende) :");
            int n = in.nextInt();
            if (n == 0) break;
            liste.addLast(n);
        }
        return liste;
    }

    /** Bildet den Mittelwert aus den Daten einer Liste
     */
    public static double mittelwert1(List<Integer> liste) {
        int summe = 0;
        for (int x : liste) summe+ = x;
        return (double) summe / (double) liste.size();
   }

    public static void main (String[] args) {
       List<Integer> lst = einlesen();
       System.out.println("Mittelwert1: " +
                          mittelwert1(lst));
    }
}
```

Es ist keine schwierige Übung auch die Klasse `SinglyLinkedList` mit der Foreach-Schleife zu versehen. Dazu muss die Klasse nur die Schnittstelle `Iterable` aus `java.lang` implementieren. Zusätzlich können Sie dann auch gleich einen Typparameter einführen.

Das folgende Listing deutet das Vorgehen an:

```
import java.util.Iterator;
```

```
public class SinglyLinkedList<T> implements Iterable<T> {
   private Node<T> first;

   ... die ueblichen Methoden

   /**
    * Implementiert Iterable<T>
    */
   public Iterator<T> iterator() {
       return new ListIterator<T>(first);
   }
}

class Node<T> { ... wie gehabt (T fuer Object) }
class ListIterator<T> { ... ebenso }
```

6.2.5 Sequentielle Datenstrukturen in der Java-Bibliothek

Ab Java 1.2 verfügt Java über eine genau überlegte Klassenstruktur für Datenstrukturen. Das bedeutet aber auch, dass das vor dieser Version nicht so war. Allerdings sind die einmal definierten Klassen später nicht entfernt worden. Ein Beispiel ist die Klasse `Vector`, die längst durch modernere Variante ersetzt werden kann.

Die entsprechende Teil der Java-Bibliothek nennt sich „Collection Framework“. Die darin enthaltenen Klassen befinden sich in dem Paket `java.util`. Wie schon der Name ausdrückt, geht es um Behälterklassen im Allgemeinen und nicht nur um sequentielle Strukturen.

Dieser Abschnitt betriff die folgenden Schnittstellen und Klassen:

Collection: Das Interface `Collection` ist der Obertyp für alle Datenbehälter. Es definert u.a. Methoden zur Abfrage der Größe oder zum Hinzufügen von Elementen.

List: Das Interface `List` beschreibt den Typ der sequentiellen Strukturen.

Iterator: Dieses Interface legt die Grundlage für das Iteratorkonzept.

ListIterator: Objekte dieser Schnittstelle verfügen über zusätzliche Iteratorfunktionen die nur bei Listen einen Sinn machen.

Iterable: Dieses Schnittstelle befindet sich in `java.lang`. Sie dient zur Kennzeichnung, dass ein Behälter über einen Iterator verfügt, der mittels der Methode `iterator` erzeugt werden kann.

ArrayList: Die Klasse `ArrayList` implementiert das Listenkonzept mit einem Arrays.

LinkedList: Die Klasse `LinkedList` implementiert das Listenkonzept mit einer doppelt verketteten Liste.

6.3 Stacks und Queues

In diesem Abschnitt kehren wir wieder zu der Darstellung von Datenstrukturen zurück. Inzwischen kennen Sie die grundlegenden Hilfsmittel zur Realisierung sequentieller Strukturen, nämlich Felder und verkettete Listen. In diesem Abschnitt zeige ich, wie die beiden

wichtigen Datenstrukturen *Stack* und *Queue* durch eine abstrakte Schnittstelle beschrieben werden können und wie man für eine Schnittstelle unterschiedliche Realisierungen formulieren kann.

Die Kunst des Programmierens besteht darin, dass man das passende Abstraktionsniveau findet. Eine gute Grundregel besagt, dass man stets die einfachste Datenstruktur wählen soll, in der sich das Problem formulieren lässt. Hinter dieser Regel versteckt sich ein Sonderfall der allgemeineren Maxime des *defensiven Programmierstils*.

Merksatz:

Setze möglichst einfache Hilfsmittel ein und baue ein Programme so auf, dass Fehler und Unzulänglichkeiten möglichst einfach entdeckt werden können!

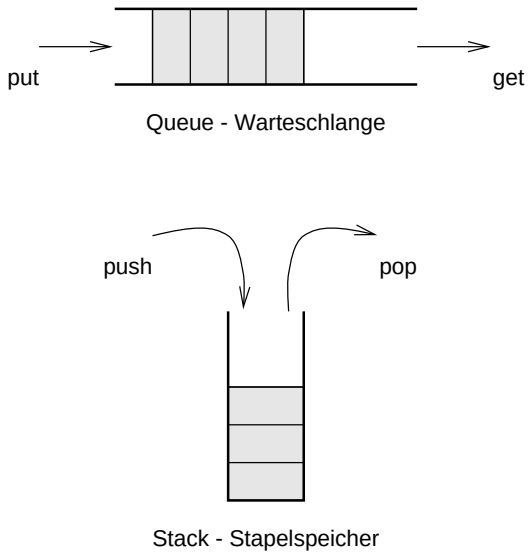

Abbildung 6.5: Die Datenstruktur *Queue* entspricht einem einfachen Zwischenpuffer, bei dem an dem einen Ende neue Daten eingefügt und an dem anderen Ende Daten entnommen werden. Dagegen ist bei einem *Stack* nur ein Ende der Datenstruktur unmittelbar ansprechbar: Dort werden neue Daten abgelegt und gespeicherte Daten entnommen, ähnlich wie bei einem Stapel, bei dem man nur auf das obere Ende zugreifen kann.

Es gibt zwei besonders eingeschränkte Varianten von Behälterklassen, die so häufig verwendet werden, dass man dafür eigene Begriffe geschaffen hat: *Warteschlangen* (*Queues*) und *Stapelspeicher* (*Stacks*). Ich gebrauche im Folgenden die deutschen und die englischen Bezeichnungen nebeneinander. Manchmal gebraucht man für Stacks auch den Namen Kellerspeicher. In der Abbildung 6.5 sind die beiden Datenstrukturen bildlich dargestellt. Bei der folgenden Besprechung der beiden Datentypen stelle ich jeweils die abstrakte Schnittstelle und zwei unterschiedliche Implementierungen vor.

Die meisten Java-Programmierer beachten bei dem Entwurf größerer Systeme den folgenden Merksatz, dem wir hier auch folgen wollen:

Merksatz:

Gehe bei dem Systementwurf zunächst von Schnittstellen aus und nicht von Klassen.

6.3.1 Die Schnittstelle Stack

Wenn es darum geht, die Reihenfolge von Daten umzudrehen, oder auch nur darum, sie – unabhängig von irgendeiner Reihenfolge – zu speichern, so ist ein Stack die geeignete Wahl.

Ein Stack verfügt – bis auf einfache Abfrageoperationen – nur über die beiden Operationen *push*: „Schiebe Daten auf den Stack“ und *pop*: „Hole die Daten wieder vom Stack“. Dabei kommen die Datenelemente in der umgekehrten Reihenfolge wieder heraus, als sie hinein „gepushed“ wurden. Man nennt Stacks daher auch LIFO-Stacks (“last in first out“).

Vielleicht wird der Begriff durch ein Beispiel noch etwas deutlicher. Sie kennen alle den Schreibtisch von Sachbearbeitern in Behörden und Ämtern. Häufig befindet sich auf der linken Seite des Tisches ein Stapel von zu erledigenden Vorgängen. Sobald der Sachbearbeiter (oder die Sachbearbeiterin) dazu kommt, nimmt er sich den nächsten Vorgang vom Stapel und bearbeitet ihn. Darauf nimmt er sich den nächsten Vorgang und so weiter. Formulare, die zwischendurch zur Bearbeitung eintreffen, legt er einfach zuoberst auf den Stapel. Dieses Prinzip hat zu Folge, dass zwar nichts vergessen wird, dass jedoch Formulare, die am unteren Ende des Stapels liegen, häufig lange unbearbeitet bleiben, da später eingehende Vorgänge bei diesem Stapelprinzip immer wieder vorgezogen werden.

Wie Sie schon an diesem Beispiel erkennen können, gibt es ganz verschiedene konkrete Realisierungen des abstrakten Konzepts eines Stapelspeichers. Ohne irgendeine Implementierung anzugeben, können wir aber in Java Stacks nicht exakt beschreiben. Immerhin können wir in Java jedoch die Aufrufschnittstellen der Stack-Operatonen – also ihre Signatur – festlegen, ohne dabei bereits eine konkrete Implementierung angeben zu müssen. Die Aufrufschnittstellen sind in Java nichts anderes als der öffentlich sichtbare Teil der Klassendeklaration. Normalerweise gehört zu dieser Schnittstellenbeschreibung immer auch eine Implementierung. Da wir eine solche Implementierung noch nicht festlegen wollen, verwenden wir hier das Interface-Konzept von Java.

Die Spezifikation einer so formulierten abstrakten Stackklasse sollte neben den notwendigen Deklarationen stets genügend Kommentare enthalten, die die Bedeutung der Operationen erläutern.

Die Darstellung der Stackklassen greift auf Typparameter zurück. Das Gegenstück können Sie dann bei der Besprechung der Queueklassen sehen.

```
/**
 * Schnittstelle fuer Stapelklassen.
 * @param T Typ der zu speichernden Objekte
 */
public interface Stack<T> {
     /**
      * Speichert x oben auf dem Stapel.
      * Vorbed.: Stapel nicht voll!
      * @param x zu speicherndes Objekt
      */
    public void push(T x);

    /**
     * Entfernt das oberste Stapelelement
     * und gibt es zurueck.
     * Vorbed.: Stapel nicht leer!
     * @return oberstes Stapelobjekt
     */
    public T pop();
```

```
    /**
     * Gibt das oberste Element zurueck,
     * ohne es zu Loeschen.
     * @return oberstes Stapelobjekt
     */
    public T peek();

    /**
     * Ist Stack voll?
     * @return true, wenn Speicher erschoepft
     */
    public boolean isFull();

    /**
     * Ist Stack leer?
     * @return true, wenn Stack leer
     */
    public boolean isEmpty();
}
```

Sie werden sich vielleicht fragen, wozu an dieser Stelle eine solche abstrakte Schnittstelle, die ja kein konkretes Programmelement festlegt, überhaupt gut ist.

Die Notwendigkeit ergibt sich einerseits aus dem Wunsch das *Verhalten* von Objekten *abstrakt* ohne Angabe einer konkreten Implementierung zu beschreiben. Andererseits aber auch aus der strengen Typprüfung von Java.

Die abstrakte Deklaration ist m Grunde ist es dasselbe wie eine Normierung oder Standardisierung. Auch Normen für Schrauben- und Papiermaße stellen nichts Konkretes zur Verfügung. Gleichzeitig ermöglicht eine Norm, wie die DIN-Norm für Papierformate, das genaue Layout eines Textes festzulegen, ohne dass man wissen muss, auf welchem konkreten Papier der Text schließlich gedruckt wird.

In dem *Collection-Frameworks* des Bibliothek-Pakets `java.util` finden Sie viele gute Beispiele für Interfaces.

Schließlich kann ich Algorithmen schreiben, die ihre Daten über einen Stack bekommen und die nicht wissen müssen, wie dieser Stack implementiert ist. Als ein (schematisches) Beispiel können Sie eine Methode nehmen, die über einen Stack Arbeitsaufträge bekommt, die sie abarbeiten soll.

```
void erledigeAuftraege(Stack<ArbeitsAuftrag> s) {
    while ( !s.isEmpty() ) {
        ArbeitsAuftrag naechsterAuftrag = s.pop();
        naechsterAuftrag.erledigen();
    }
}
```

Wir können also Funktionen schreiben, die mit einem *bereits vorhandenen* Stack umgehen können, ohne zu wissen, wie er realisiert ist. Wenn wir dagegen ein Stackobjekt erzeugen wollen, müssen wir uns unbedingt auf einen konkreten Datentyp festlegen.

Um konkrete Stacks zu erzeugen, müssen wir also konkrete Klassen definieren, die die Stackfunktionen implementieren.

Es gibt viele unterschiedliche Realisierungen von Stacks. Wichtige praktische Gesichtspunkte bei der Auswahl eines konkreten Verfahrens sind die Laufzeiteffizienz und die effiziente Nutzung des Speichers. Im folgenden stelle ich zwei unterschiedliche Imple-

mentationen vor, nämlich `ArrayStack` und `ListStack`.

Zunächst will ich noch ganz kurz die verschiedenen Stackfunktionen kommentieren. Die beiden Funktionen `push` und `pop` stellen die eigentliche Funktionalität des Stack dar. Sie sind daher für eine Stackklasse zwingend erforderlich. Die anderen Funktionen sind demgegenüber zweitrangig.

Die beiden Abfragefunktionen `isFull` und `isEmpty` sind dazu da, dass der Anwender der Klasse `Stack` unerlaubte Zugriffe vermeiden kann. `isFull` ist nötig bei Stackimplementationen, die nur über einen beschränkten Speicherplatz verfügen.

6.3.2 Anwendungen von Stacks

Grundsätzlich können Sie Stacks überall da verwenden, wo es nötig ist, einige Daten (zeitweise) zu speichern – zumindest dann, wenn Sie keinen besonderen Wert auf die Reihenfolge der Datenelemente legen. Als ein Beispiel führe ich hier die Ein- und Ausgabe von Daten an.

```
public static void eingabe(Stack<Integer> s) {
    boolean weiter = true;
    while (weiter) {
        System.out.print("Eingabe: ");
        int zahl = in.nextInt();
        if (zahl != 0)
            s.push(zahl);
        else
            weiter = false;
    }
}

/** Gibt die Daten eines Stacks aus
 */
public static void ausgabe(Stack<Integer> s) {
    while (!s.isEmpty()) System.out.print(s.pop() + " ");
    System.out.println();
}
```

Beachten Sie, dass es ohne weiteres möglich ist diese Funktionen separat vorzuübersetzen, *ohne* dass dabei irgendeine Festlegung über die Art der Implementierung des Stacks getroffen wurde. An anderer Stelle, z.B. in `main` kann dann später ein Objekt erzeugt werden.

```
public static void main (String[] args) {
    System.out.print("Bitte Werte fuer Stack eingeben: ");
    Stack<Integer> s = new ListStack<Integer>();
    eingabe(s);
    System.out.print("Das war: ");
    ausgabe(s);
}
```

Ein Stack ist sicher nicht die ideale Datenstruktur für die Mittelwertberechnung, bei der die benötigten Daten ja nicht „verbraucht“ werden sollten. Es gibt aber eine ganze Reihe von Anwendungen, in der genau dieser temporäre Charakter des Speicherns gewollt ist.

Das wichtigste Beispiel von Stackstrukturen in einem Computer ist die Verwaltung von *Aktivierungsrekords* einer Funktion. Im ersten Semester haben Sie bei der Behandlung re-

kursiver Funktionen gelernt, dass jeder rekursive Funktionsaufruf eine Instanz der Funktion bildet, die über einen eigenen Satz von Übergabeparametern und lokalen Variablen verfügt. Die Inhalte dieser Variablen werden in dem Aktivierungsrekord gespeichert, der daneben noch weitere Informationen (insbesondere die Rücksprungadresse) enthält. Der Begriff Stack rechtfertigt sich hier dadurch, dass die Aktivierungsrekords genau wie beim Stapelspeicher hintereinander angelegt und in umgekehrter Reihenfolge wieder entfernt werden. Aktivierungsrekords werden nicht durch eine eigene Stackklasse, sondern durch die Prozessorhardware verwaltet.

Die Tatsache, dass Aktivierungsrekords stackartig verwaltet werden, ermöglicht es, rekursive Funktionsaufrufe durch die explizite Verwendung eines Stacks (der Parameter und Variable zwischenspeichert) zu eliminieren. Da einige ältere Programmiersprachen keine rekursiven Aufrufe erlaubten, war diese Methode häufig nötig. Allerdings führt sie meist zu schwer verständlichen Programmen, so dass man bei es „richtig“ rekursiven Problemen fast immer bei der rekursiven Formulierung belässt (die häufig auch schneller als die Lösung mit dem programmierten Stack ist). Es gibt jedoch auch einige Bereiche, in denen explizite Stack-Strukturen auch heute noch eingesetzt werden. Mit die wichtigsten Beispiele sind die syntaktische Analyse von formalen Texten und die Berechnung arithmetischer Ausdrücke. Ihnen ist vielleicht das Prinzip eines UPN-Taschenrechners bekannt, der nach dem Prinzip der „umgekehrten polnischen Notation“ arbeitet: UPN-Taschenrechner speichern Zwischenresultate auf einem Stack.

6.3.3 Stack-Implementation mit einer Liste

Auch wenn Stackoperationen so einfach sind, dass man die nötigen Listenoperationen explizit programmieren kann, greife ich hier auf die Bibliotheksklasse `LinkedList` zurück. Dabei wird nochmals deutlich, dass es neben der Vererbungsbeziehung die *Nutzungsbeziehung* gibt. Es ist ganz wichtig, die beiden Begriffe auseinanderzuhalten. Ein Stack ist ein anderes Konzept als eine Liste, daher wäre hier Vererbung nicht angebracht. Allerdings realisiert er seine Funktionalität mit Hilfe der Listenoperationen. Dies wird dadurch erreicht, dass die Stackklasse eine Liste als Komponente enthält.

Definition:

*Unter **Komposition** versteht man den Aufbau eines Objektes aus Teilobjekten anderer Klassen.*

Merksatz:

*Das Prinzip der Komposition von Klassen ist mindestens ebenso wichtig wie die Vererbung. Die Realisierung vieler Methoden geschieht dabei mittels **Delegation**, also der Weiterleitung des Methodenaufrufs an die elementarere Klasse.*

```
import java.util.LinkedList;

public class ListStack<T> implements Stack<T> {
    private LinkedList<T> data;

    public ListStack() {
        data = new LinkedList<T>();
    }
    public void push(T x) {
        data.addFirst(x);
    }
```

```
    public T peek() {
        return data.getFirst();
    }
    public T pop() {
        return data.removeFirst();
    }
    public boolean isFull() {
        return false;
    }
    public boolean isEmpty() {
        return data.isEmpty();
    }
}
```

Durch die Verwendung der Klasse `LinkedList` ist dieses mal auch die Klasseninvariante extrem einfach:

Klasseninvariante:

Die Klasse `ListStack` *speichert die Datenelemente in der Variablen* `data` *von Typ* `LinkedList`. *Das zuletzt eingefügte Element findet sich am Anfang dieser Liste.*

Vielleicht denken Sie, dass sich so eine einfache Klasse gar nicht lohnt, da man ja die nötigen Funktionen der Klasse `LinkedList` jederzeit unmittelbar aufrufen kann und so denselben Zweck erreicht. Sie haben insofern recht, als unter Umständen eine zu weitgehende Modularisierung ein Programm auch unnötig aufblähen kann. Andererseits sollten Sie aber auch bedenken, dass mit die wichtigste Funktion eines Programms darin besteht, dass es für Menschen lesbar ist. Die bessere Lesbarkeit eines Programmabschnitts wirkt sich nicht nur bei der Weiterentwicklung durch andere Kollegen aus, sondern sie führt meist auch zu korrekteren Programmen.

6.3.4 Stack-Implementation mit einem Array

Arrays stellen hinsichtlich Laufzeit- und Speicheranforderungen die günstigste Datenstruktur dar.

Auch hier wird die Entwurfsentscheidung durch die Klasseninvariante festgelegt.

Klasseninvariante:

Die Klasse `ArrayStack` *speichert die Datenelemente in dem Feld* `data`. *Die Variable* `top` *enthält den Index des niedrigsten freien Stapelplatzes. Die Indizes der gültigen Stapelelemente sind* 0 *bis* `top`-1.

Nach der Festlegung der Klasseninvariante ist die Implementierung wieder weitgehend festgelegt.

```
public class ArrayStack<T> implements Stack<T> {
    private int top;
    private final T[] data;

    public ArrayStack(int n) {
        top = 0;
        data = (T) new Object[n];
    }
    public void push(T x) {
```

```
        if(isFull()) // Phantasie-Exception
            throw new StackFullException();
        data[top] = x;
        top++;
    }
    public T pop() {
        if (isEmpty()) throw new NoSuchElementException();
        top--;
        T result = data[top];
        data[top] = null;
        return result;
    }
    public T peek() {
        if (isEmpty()) throw new NoSuchElementException();
        return data[top-1];
    }
    public boolean isFull() {
        return top == data.length;
    }
    public boolean isEmpty() {
        return top == 0;
    }
}
```

Die Stackklasse ist so einfach, dass sich beinahe jeder Kommentar erübrigt. Die hier gewählte fest Arraygröße ist sicher unproblematisch, wenn man im voraus die maximale Stackgröße angeben kann. Andernfalls sollte man bei einer „richtigen“ Stackklasse dafür sorgen, dass bei Bedarf ein größeres Datenarray erzeugt wird.

Die von mir nicht ausformulierte `StackFullException` hat nur Kommentarwert.

Die einfachen Arrayoperationen verführen dazu, auf eine eigene Stackklasse zu verzichten und im Programm einfach Anweisungen wie

```
feld[sp++] = datenelement;
```

einzustreuen. Ein wichtiger Vorteil der objektorientierten Lösung ist neben der besseren Lesbarkeit aber die einfache Änderbarkeit von Implementierungsdetails, wie die automatische Vergrößerung des Arrays.

6.3.5 Die abstrakte Schnittstelle Queue

Ähnlich wie der Stack verfügt die Datenstruktur *Queue* über zwei grundlegende Operationen: Ich nenne sie *put* und *get*. Mit *put* werden Datenelemente in die Queue eingefügt; mit *get* werden die Daten wieder entnommen.

Im Unterschied zum Stack bleibt bei der Queue die Reihenfolge der Daten unverändert. Sie heißt daher auch „FIFO“-Stack (“first In first out “). Eine andere deutsche Bezeichnung ist *Warteschlange*.

Wenn der Sachbearbeiter, den wir für die Erläuterung des Stacks herangezogen haben, gemäß dem Queue-Prinzip arbeitet, wird es nicht so leicht passieren, dass Akten lange liegen bleiben. Er wird nach wie vor immer von seinem Stapel den obersten Vorgang zur Bearbeitung auswählen. Nur wird er jetzt neueingehende Unterlagen jeweils *unter* den Stapel schieben, so dass die Reihenfolge der Eingänge beibehalten bleibt.

Es gibt viele unterschiedliche Realisierungen von Warteschlangen. Wie bei Stacks besteht

der wichtigste praktische Unterschied darin, ob die Größe der Warteschlange beschränkt ist (*bounded queue*) oder nicht. Eine unbeschränkte Warteschlange wird meist durch eine Liste implementiert; für eine beschränkte Warteschlange bietet sich ein Vektor an.[10]

In der folgenden Klassenschnittstelle wurden wie beim Stack die Funktionen `isFull` und `isEmpty` aufgenommen, **Bei diesem Beispiel werden keine Typparameter verwendet.**

```
public interface Queue {
    /**
     * Haengt x an die Queue an.
     * Vorbed.: Queue nicht voll!
     * @param x anzuhaengendes Objekt
     * @throws QueueFullException
     */
    public void put(Object x);

    /**
     * Entnimmt das erste Element der Queue.
     * Vorbed.: Queue nicht leer!
     * @return erstes Element.
     * @throws NoSuchElementException, wenn Liste leer.
     */
    public Object get();

    /**
     * Ist Queue voll?
     * @return true, wenn ja
     */
    public boolean isFull();

    /**
     * Ist Queue leer?
     * @return true, wenn ja
     */
    public boolean isEmpty();
}
```

6.3.6 Anwendungen von Queues

Queues können ähnlich wie Stacks zur temporären Datenspeicherung benutzt werden. Das beim Stack angeführte Beispiel der Mittelwertberechnung können wir demnach mit einer Queue genauso hinschreiben.

Der wichtigste Unterschied zwischen Stack und Queue besteht in der Reihenfolge der Datenelemente, die bei einer Queue anders als beim Stack nicht umgekehrt wird. Dies bedeutet einerseits, dass eine Queue *nicht* verwendet werden kann, wenn es um die iterative Formulierung rekursiver Verfahren geht. Andererseits können wir Queues da verwenden, wo die ursprüngliche Reihenfolge der Datenelemente beibehalten werden muss.

Eine ganz wichtige Anwendung sind Datenpuffer zum Zwischenspeichern von Elementen eines Datenstroms. Die Notwendigkeit der Zwischenspeicherung entsteht im allgemeinen durch unterschiedliche (asynchrone) Verarbeitungsgeschwindigkeiten von Datensender und Datenempfänger. Als ein Beispiel können Sie sich die Dateneingabe in ein Programm

[10] Beschränkte Warteschlangen machen in vielen Anwendungen wirklich Sinn, so dass man ganz bewusst keine Vergrößerung der Datenstruktur zulässt, obwohl das einfach möglich wäre.

vorstellen. Das Senden der Daten erfolgt durch die Tastatur mit der Tippgeschwindigkeit des Anwenders, das Empfangen geschieht durch das Anwendungsprogramm, das gemäß seinen Verarbeitungszeiten die Eingaben verarbeitet. Damit keine Daten verlorengehen, während das Programm nicht in der Lage ist neue Daten anzunehmen (z.B. während das Betriebssystem andere Aufgaben durchführt), müssen die Daten in einer Queue zwischengespeichert werden. Die Klassen `BufferedReader`, `BufferedWriter`, `BufferedInputStream` und `BufferedOutputStream` aus der Java-Bibliothek, verwenden intern den Queue-Mechanismus um die Effizienz der Ein-/Ausgabe zu erhöhen.

Innerhalb von reinen Softwareanwendungen sind Queues unter anderem für bestimmte Arten der Traversierung (dem Durchlaufen) von Datenstrukturen nötig. Am Ende dieses Kapitels zeige ich Ihnen, wie Sie mittels einer Queue eine Baumstruktur ebenenweise durchlaufen können. Diese Art der Baumtraversierung wird auch bei effzienten Lösungen von Suchproblemen angewendet.

6.3.7 Queue-Implementation mit einer Liste

Die Implementierung einer Queue mittels einer verketteten Liste ist sehr ähnlich zu der Stackimplementierung. Der einzige Unterschied besteht darin, dass wir neue Elemente nicht einfach am Listenanfang einfügen können, sondern dass wir sie am Ende der Liste anfügen müssen.

```
import java.util.LinkedList;

public class ListQueue implements Queue {
    private final LinkedList data;

    public ListQueue() {
        data = new LinkedList();
    }
    public void put( Object x) {
        data.addLast(x);
    }
    public Object get() {
        return data.removeFirst();
    }
    public boolean isFull() {
        return false;
    }
    public boolean isEmpty() {
        return data.isEmpty();
    }
}
```

Ich denke zu diesem Beispiel ist weiter nichts zu sagen. Entsprechend einfach ist auch wieder die Klasseninvariante:

Klasseninvariante:

Die Klasse `ListQueue` *speichert die Datenelemente in der Reihenfolge des Einfügens in einem Listenobjekt. Die Liste wird über die Variable* `data` *referiert.*

Wenn Sie anstelle der Bibliotheksklasse die Klasse `SinglyLinkedList` verwenden, bietet es sich an, darüber nachzudenken wie man `addLast` effizienter machen kann. Das geht nämlich indem man in einer Instanzvariablen `last` die Referenz des letzten

Listenknoten speichert (siehe Abbildung 6.1). Natürlich ist das in der Bibliotheksklasse (die ist ja auch doppelt verkettet) schon bedacht.

6.3.8 Queue-Implementierung mit einem Array

Die Arraylösung ist nicht ganz so einfach, wie es zunächst den Anschein hat.

Ein Arrayobjekt hat stets eine genau definierte Länge. Daher liegt es nahe, mit Arrays eine beschränkte Queue fester Größe zu realisieren. Die maximale Anzahl von Elementen wird durch den Konstruktor festgelegt.

```
public class ArrayQueue implements Queue {
    private final Object[] data;
    private int size;
    private int in;
    private int out;

    public boolean isFull() {
        return size == data.length;
    }

    public boolean isEmpty() {
        return size == 0;
    }
}
```

In dem Listing haben Sie wahrscheinlich erkannt, dass das Feld `data` dazu dient, die Datenelemente zu speichern. Dabei gibt es jedoch ein Problem, das an dem folgenden Beispiel erkennbar wird:

```
Queue v = new ArrayQueue q(20);

for(int i=0; i<20; i++)
    q.put(Integer.valueOf(i));
for(i=0; i<19; i++)
    System.out.println(q.get());
q.put(Integer.valueOf(5)); // geht das jetzt?
```

Nachdem 20 Elemente in q eingefügt und danach 19 Elemente entnommen wurden, müsste eigentlich Platz für neue Elemente sein. Nur wo kommen diese Elemente hin?

Nach der ersten For-Schleife hat das Feld `data` doch folgenden Inhalt (natürlich sind das alles Integer-*Objekte*:

```
data[0] = 0, data[1] = 1, ..., data[19] = 19
```

Nachdem in der zweiten For-Schleife 19 Elemente entnommen wurden, sind die ersten 19 Felder von `data` wieder verfügbar. Nur wie merkt man sich das?

Die Abbildung 6.6 gibt die Grundidee der Variablenbelegung einer Queue wieder, die nach mehrmaligen Zugriffen so entstanden sein könnte. Die in der Abbildung nicht aufgeführte Variable `size` hat den Wert 5.

Dieses Diagramm legt die Formulierung einer brauchbaren Klasseninvariante nahe. Die Klasse verwendet zwei Indizes um den Zugriff zu dem Datenfeld zu verwalten. Der Index

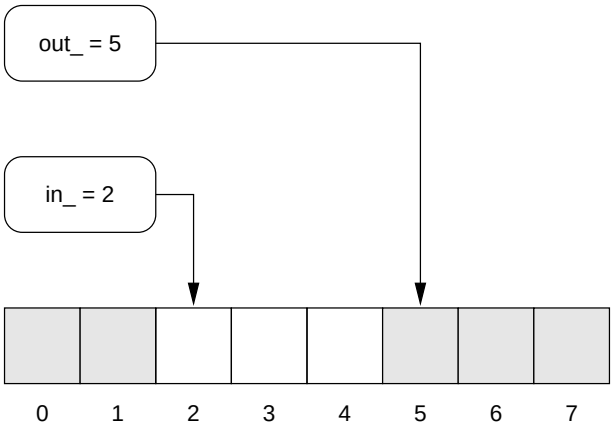

Abbildung 6.6: Die Realisierung eines zirkulären Puffers. Die hellen Felder sind frei und die grauen Felder sind mit Daten belegt. Die Reihenfolge der belegten Datenfelder ist: 5, 6, 7, 0, 1.

`in` zeigt immer auf das nächste freie Datenfeld, während `out` immer auf das nächste anstehende Ausgabefeld zeigt. Wenn jedoch beide Variable auf dasselbe Feld zeigen, tritt ein logischer Widerspruch auf. Zu seiner Auflösung wird die Variable `size` herangezogen. Bei leerer Queue (`size` $= 0$) ist der `out`-Index ungültig: Das Feld ist frei. Ist dagegen die Warteschlange nicht leer und `size` $=$ `data.length`, so ist der `in`-Index ungültig: Das Feld ist mit einem gültigen Wert belegt.

Das Weiterzählen der beiden Indizes wird so gestaltet, dass sie immer in das Datenfeld zeigen. Sobald ein Index von `data.length`-1 zum nächstfolgenden Feld geht, muss er zurück auf 0 gesetzt werden. Dies erreicht man am einfachsten durch Bestimmung des Teilerrestes modulo `data.length`.

Nach diesen Überlegungen können wir die etwas kompliziertere Klasseninvariante hinschreiben.

Klasseninvariante:

> *Die Datenelemente der Klasse* `ArrayQueue` *werden in dem Array* `data` *gespeichert. Die Anzahl der aktuell gespeicherten Daten beträgt* `size`*. Das nächste zu entnehmende Element befindet sich in* `data[out]`*, und das nächste Einfügen erfolgt bei* `data[in]`*. Die Datenelemente sind in der Reihenfolge von zyklisch aufsteigenden Indizes gespeichert. Auf den letzten Index folgt der Index* 0*.*

Zusätzlich zur Beachtung der Klasseninvariante ist bei dieser Klasse das Einhalten der Vorbedingungen von `put` und `get` ganz wichtig. Sonst genügt ein einziger falscher Aufruf, um das gesamte System durcheinanderzubringen.

```
public ArrayQueue(int n) {
    size = 0;
    in = 0;
    out = 0;
    data = new Object[n];
}

/**
 * Fuegt ein Objekt ein.
 * <p>
 * Vorbed.: Die Queue ist nicht voll!
 * @param x einzufuegendes Objekt
```

```
  * @throws IllegalStateException wenn voll
  */
 public void put(Object x) {
     if(!isFull()) {
         size++;
         data[in] = x;
         in = (in+1) % data.length;
     }
     else throw new IllegalStateException();
 }

 /**
  * Entnimmt ein Objekt,
  * <p>
  * Vorbed.: Die Queue ist nicht leer!
  * @return entnommenes Objekt
  * @throws NoSuchElementException wenn leer
  */
 public Object get() {
     if(!isEmpty()) {
         size--;
         Object result = data[out];
         data[out] = null;
         out = (out+1) % data.length;
         return result;
     }
     throw new NoSuchElementException();
 }
```

6.3.9 Stack und Queue in der Java-Bibliothek

In der ersten Java-Version gab es bereits eine Klasse `Stack`. Diese Klasse passt genauso wenig wie ihre Oberklasse `Vector` in das modernere Konzept des Collection Frameworks.

Sie sollten in die folgenden Schnittstellen und Klassen aus `java.util` kennen:

Queue: Das Interface `Queue` legt die Operationen für eine Warteschlange fest. Die Namen der Operationen sind anders als in meinen Beispielen. Es werden mehrere Varianten für jede Operation definiert.

Deque: Das Interface `Deque` (*double ended queue*, gesprochen: „deck“) beschreibt zusammenfassend die Operationen von Stack und Queue. Es beschreibt genau das, was sein Name sagt: Eine sequentielle Datenstruktur, auf die man nur über die beiden Enden zugreifen kann. Sie erweitert das Interface `Queue`.

Klassen: Es gibt eine große Anzahl von Klassen, die die angesprochenen Schnittstellen implementieren. Dazu gehört auch die Klasse `LinkedList`. Einige Implementierungen sind speziell auf Multithreading zugeschnitten. Man findet diese Klassen in dem Paket `java.util.concurrent`

6.4 Binärbäume – Nichtlineare Datenstrukturen

6.4.1 Grundbegriffe

Die beiden letzten Datenstrukturen – Queue und Stack – zeichneten sich durch besondere Einfachheit aus. Diese Einfachheit erlaubt eine besonders effiziente Implementierung.

Auf der anderen Seite besteht aber auch das Bedürfnis für komplexere als die bisher besprochenen sequentiellen und linearen Datenstrukturen.

Die bisher behandelten Strukturen waren Ansammlungen von Datenelementen, die in einer festgelegten Reihenfolge angeordnet sind. Für die Begriffsbildungen und für die Algorithmen waren die Datentypen und Eigenschaften der Datenelemente eher zweitrangig. Wichtig war allein die lineare Anordnung der Daten. Auf der anderen Seiten gibt es Datenstrukturen mit viel komplizierteren Verbindungen zwischen den Datenelementen. In der allgemeinen Form nennen wir solche Strukturen *Graphen*.

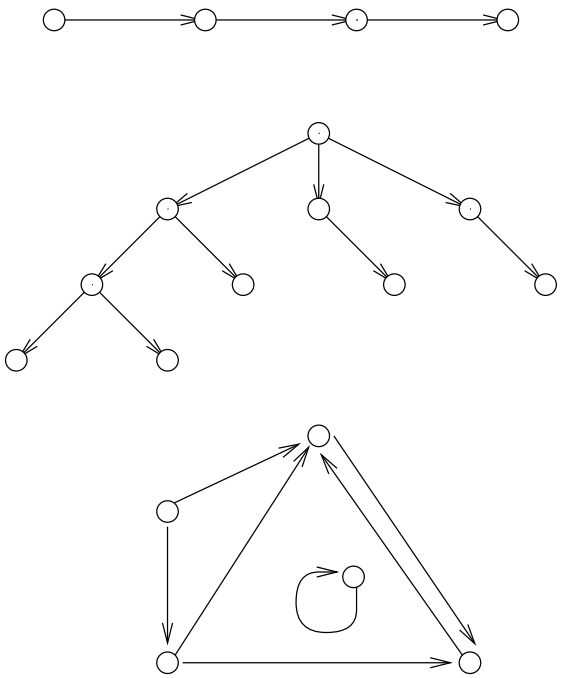

Abbildung 6.7: Verschiedene Formen von Graphen. Von oben nach unten: lineare Liste, Baum und allgemeiner Graph.

Die in den Strukturen enthaltenen Datenelemente werden *Knoten* genannt. Die Verbindungen zwischen den Knoten – die in den bisherigen Beispielen die Reihenfolge festlegten – nennt man *Kanten*. Eine Datenstruktur, in der Knoten auf beliebige Weise durch Kanten verbunden sind, nennt man *Graph*.

Definition:

*Ein **Graph** besteht aus einer Menge von **Knoten** und aus einer Menge von **Kanten**. Jede Kante verbindet genau zwei Knoten.*

Graphen spielen in der Informatik eine besondere Rolle, da man mit ihnen aufgrund ihrer großen Allgemeinheit sehr viele Sachverhalte modellieren kann. Wegen dieser Bedeutung hat man auch eine große Zahl von Algorithmen entwickelt, die auf effiziente Art und Weise auf Graphen arbeiten.

Die Graphentheorie und der Bereich der Graphalgorithmen ist so umfassend und komplex, dass ich hier nicht näher darauf eingehen kann.[11]

Hier will ich mich auf eine sehr wichtige Teilmenge von Graphen beschränken, die zwar etwas komplexer als die bisherigen linearen Strukturen, aber trotzdem noch sehr überschaubar und entsprechend auch sehr anwendungsrelevant ist, nämlich auf *Bäume* und insbesondere auf *Binärbäume*.

Je nachdem, ob die Kanten eines Graphen eine Richtung angeben, spricht man von *gerichteten* oder *ungerichteten* Graphen. Da wir in den folgenden Beispielen Kanten immer durch Referenzen ausdrücken, beschränke ich mich auf gerichtete Graphen. Wo nötig, wird die Richtung der Kanten durch Pfeile ausgedrückt.

Die Graphentheorie hat eine Reihe von Begriffen geschaffen, die in vielen Informatikanwendungen immer wieder vorkommen.

Definition:

*Ein **Weg** ist eine Folge von Kanten. Ein Knoten B ist von einem anderen Knoten A aus **erreichbar**, wenn es einen Weg von A nach B gibt. Ein Graph heißt **verbunden**, wenn es mindestens einen Knoten gibt, von dem jeder andere Knoten erreicht werden kann. Ein Weg, der auf den Ausgangsknoten zurückführt, heißt **Kreis**. Die **Entfernung** zweier Knoten ist gleich der minimalen Länge des dazwischen liegenden Weges, wobei die Länge eines Weges gleich der Anzahl der Kanten ist.*

Nach diesen Grundbegriffen, kommen wir zur Definition eines Baumes.

Definition:

*Ein **Baum** ist ein Graph, der einen ausgezeichneten Knoten, **Wurzel** genannt, besitzt, von dem aus jeder andere Knoten über genau einen Weg erreicht werden kann. Die Entfernung eines Knotens von der Wurzel bestimmt seine **Ebenennummer**. Die maximale Ebenennummer heißt **Höhe** des Baums.*

Viel Strukturzusammenhänge sind baumartig – man sagt auch hierarchisch – gegliedert. Ein bekanntes Beispiel sind Stammbäume, wie der in Abbildung 6.8, der die Anfänge der Welt schildert.

In Analogie zu der Stammbaumterminologie spricht man bei Bäumen auch von *Elternknoten* und von *Kindknoten*. Gemäß dem Sprachgebrauch der Botanik nennt man einen Endknoten (ein Knoten, der keine Kinder besitzt) auch *Blatt* oder *Blattknoten*.

Ich vermute, dass die Baumstruktur in vielen Zusammenhängen nur deshalb auftaucht, weil sie besonders einfach zu überschauen ist. So sind die Verantwortlichkeiten in Unternehmen fast immer baumartig strukturiert, obwohl es dafür keinen absolut zwingenden Grund gibt.

Bei den folgenden Beispielen werde ich mich auf eine besondere Art von Bäumen konzentrieren, nämlich auf sogenannte *Binärbäume*, die sich dadurch auszeichnen, dass ein Knoten höchstens zwei Nachfolgeknoten (Kinder) hat.

Definition:

*Ein **geordneter Baum** ist ein Baum, in dem die Reihenfolge der Kindknoten genau festgelegt ist. Ein **Binärbaum** ist ein geordneter Baum, in dem jeder Knoten höchstens zwei Kinder hat.*

[11] Vielleicht werden Sie ja das Fach „Algorithmik“ hören. Dort werden Sie mehr zu Graphalgorithmen lernen.

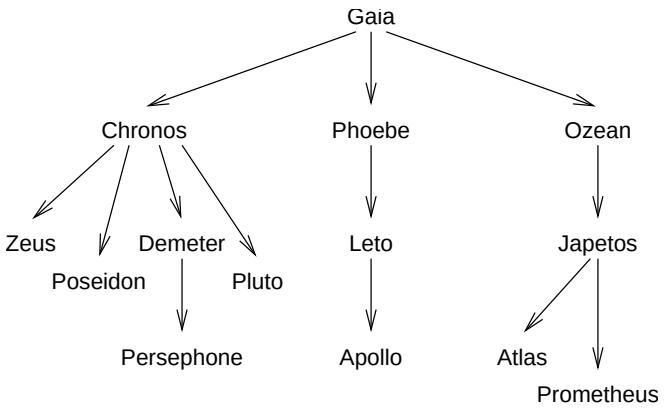

Abbildung 6.8: Ein typischer Stammbaum

Eine typische Eigenschaft von Bäumen und von Binärbäumen ist der rekursive Charakter der Datenstruktur. Ein Knoten eines Baumes ist nämlich entweder ein Blatt oder seinerseits die Wurzel eines Baumes. Aufgrund dieser rekursiven Struktur ergibt sich für fast alle Baumfunktionen eine rekursive Definition und auch eine rekursive Programmstruktur.

Ähnlich wie bei anderen Datenstrukturen gibt es auch für Graphen und Bäume eine Vielzahl unterschiedlicher Beschreibungsarten. Die wichtigste computerinterne Darstellung erfolgt durch dynamische Datenstrukturen und Zeiger. Zunächst will ich aber auch kurz ansprechen, wie Bäume durch lineare Strukturen dargestellt werden können.

6.4.2 Lineare Repräsentation von Binärbäumen

Auf den ersten Blick erscheint es etwas widersinnig, dass man nichtlineare Strukturen linear darstellen soll. Das Besondere an Bäumen ist doch gerade, dass an einem Knoten *mehrere* Nachfolger „hängen“ können, so dass man meinen sollte, dass eine lineare Darstellung nicht möglich ist.

Trotzdem, ich behaupte, dass man ohne eine lineare Darstellung von nichtlinearen Datenstrukturen nicht auskommt. Der Grund ist ganz einfach: Die meisten Kommunikations- und Speichermedien arbeiten streng sequentiell!

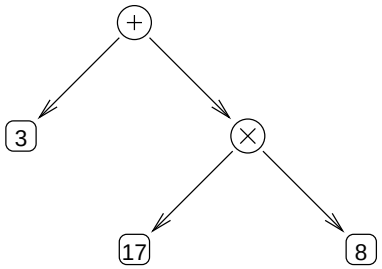

Abbildung 6.9: Baumdarstellung eines mathematischen Ausdrucks (abstrakter Syntaxbaum)

Ein ganz einfaches Alltagsbeispiel für die sequentielle Darstellung von Bäumen ist die Schreibweise für arithmetische Ausdrücke, die „normale“ mathematische Formelsprache. Wenn Sie nämlich einmal genauer darüber nachdenken, werden Sie bemerken, dass eine Formel, wie $3 + 17 \cdot 8$, eigentlich genau dem Binärbaum entspricht, der in der Abbildung 6.9 dargestellt ist. Dass es sich bei einer Formel nicht um eine einfache Sequenz handelt, erkennen Sie hier daran, dass in dieser Formel das Pluszeichen nicht die unmittelbar daneben stehenden Zahlen 3 und 17, sondern vielmehr das Ergebnis des linken „Teilbaums“ $17 \cdot 8$ mit der 3 verknüpft. Die lineare Notation eines eigentlich hierarchischen Zusammenhangs ist nicht immer einfach zu interpretieren: Bei der Interpretation von Ausdrücken müssen Sie unbedingt wissen, wie die Priorität der Operatoren definiert ist, dass also „Punktrechnung“ stets vor „Strichrechnung“ durchzuführen ist.

Ganz genauso wie in der Arithmetik erfordern auch andere Serialisierungen von nichtlinearen Strukturen ein festgelegtes Protokoll, das eine eindeutige Darstellung definiert.[12]

6.4.3 Binärbäume als dynamische Datenstruktur

Bei Listen haben wir gesehen, wie man mit Referenzen explizit die Reihenfolge der Datenelemente ausdrücken kann. Im Vergleich zur impliziten Speicherung in Feldern bieten solche dynamischen Datenstrukturen eine deutlich höhere Flexibilität. Es ist also nur konsequent, wenn wir auch Binärbäume ähnlich implementieren. Dabei sollten Sie jedoch nicht vergessen, dass es auch andere Möglichkeiten gibt.

Das entscheidende Datenelement von Bäumen sind die Knoten, an denen die Verzweigungen angebracht sind. Da aufgrund der rekursiven Definition von Bäumen an jedem Knoten wieder ein Baum „hängt“, empfiehlt es sich die Funktionen, die auf Bäumen definiert sind, einer Knotenklasse zuzuordnen.

Um den rekursiven Charakter der Baumstruktur zu betonen, beschreibe ich Bäume hier durch Baumknoten, die durch die einfache Klasse `TreeNode` beschrieben werden (eine perfekte Baumklasse ist anders und erheblich komplexer aufgebaut). Wie bei der Liste habe ich auch hier keine vollständige Liste der möglichen Funktionen angegeben. Wie Sie an der folgenden Zusammenstellung der Struktur und der Methodenschnittstelle sehen, habe ich nämlich hier schon ein besonderes Verhalten vorgeschrieben, nämlich dass sich Bäume ausdrucken lassen.

Anmerkung:

Die hier vorgestellte Implementierung von Binärbäumen ist bei weitem nicht die einzige Möglichkeit. In der Praxis findet man viele unterschiedliche Varianten.

```
// Beschreibung der Schnittstelle. Keine korrekte Syntax!
//
public abstract class TreeNode {
    protected TreeNode left;
    protected TreeNode right;

    protected TreeNode()

    protected TreeNode(TreeNode left, TreeNode right)

    /**
```

[12] Eine große praktische Bedeutung hat das sequentielle Abspeichern von hierarchischer (baumartiger) Information in XML-Dateien.

```
     * Gibt die Hoehe des Baums ab diesem Knoten zurueck
     */
    public int hoehe()

    /**
     * Gibt die Anzahl aller Knoten ab root zurueck
     */
    public int anzahl(TreeNode root)

    // Rekursives Durchlaufen des Baumes:

    public static String preOrder(TreeNode n)
    public static String inOrder(TreeNode n)
    public static String postOrder(TreeNode n)

    /* Alternative Implementierung von postOrder()
     */
    public String postOrder()

    /**
     * Ebenenweises Durchlaufen des Baumes:
     */
    public void levelOrder()
}
```

Wie Sie an den Instanzvariablen sehen, hat ein `TreeNode`-Objekt keinen wirklichen Inhalt. Es enthält ausschließlich zwei Zeiger, nämlich einen auf den linken Teilbaum und einen auf den rechten Teilbaum. Fehlende Teilbäume werden – analog zum Vorgehen bei Listen – durch `null` gekennzeichnet. Mit diesen Festlegungen ist es möglich, solche Funktionen, die nur auf der Baumstruktur beruhen und keinen besonderen Bezug auf Knotenwerte erfordern, zu formulieren. Dateninhalte und weitere Funktionen können später durch Vererbung den Baumknoten hinzugefügt werden. Bei den Baumfunktionen haben wir von der folgenden Festlegung auszugehen:

Klasseninvariante:

Die Zeiger `left` *und* `right` *sind entweder gleich* `null` *oder sie zeigen auf einen Knoten, der vom Typ* `TreeNode` *oder von einem abgeleiteten Typ ist.*

Die Funktionen `anzahl` und `hoehe` sind typisch für Algorithmen auf Bäumen. Entsprechend der rekursiven Definition der Datenstruktur werden Baumalgorithmen rekursiv definiert. Das Grundprinzip besteht darin, die Funktion rekursiv für den rechten und den linken Teilbaum auszuführen und die Teilergebnisse – entsprechend der vorgegebenen Funktionalität – geeignet zusammenzufassen. Ein einfaches Beispiel ist die Funktion `anzahl`, in der einfach die Anzahl der Knoten im linken Teilbaum, die Anzahl der Knoten im rechten Teilbaum und eine Eins für den Knoten selbst zusammengezählt werden.

Die Beispiele sind im Detail bewusst unterschiedlich implementiert um Ihnen zu zeigen, dass es mehr als eine Möglichkeit für die Formulierung von Baumalgorithmen gibt. Natürlich sollte man sich bei einer richtigen Bibliotheksklasse auf einen einheitlichen Stil festlegen.

Betrachten wir zunächst die Bestimmung der Anzahl aller Knoten mittels der Methode `anzahl`. Dies ist selbst die Methode eines Knotens. Folglich wird nach dieser Methode der kleinste Baum die Größe 1 haben. Vor einem (rekursiven[13]) Aufruf der Methode für

[13] Genau genommen kann man nicht wissen, ob eine Methode rekursiv ist.

die Kindknoten müssen wir prüfen, ob es überhaupt Kinder gibt.[14]

```
public int anzahl() {
    int result = 1;
    if (this.left != null)
        result += this.left.anzahl();
    if (this.right != null)
        result += this.right.anzahl();
    return result;
}
```

Mit der Funktion `anzahlStatic` ist derselbe Sachverhalt durch eine Klassenfunktion beschrieben. Diese erhält die Referenz auf den zu untersuchenden Baum als Parameter. Dieser Parameter kann natürlich auch `null` sein. Daher kann das Ergebnis jetzt auch 0 lauten. Die rekursiven Funktionsaufrufe sind immer möglich.

```
public static int anzahlStatic(TreeNode root) {
    if (root == null) return 0;
    return 1 +
        anzahlStatic(root.left) +
        anzahlStatic(root.right);
}
```

Die Bestimmung der Höhe eines Teilbaums sieht ähnlich aus. Anstelle der Summierung muss hier jedoch die maximale Höhe der beiden Teilbäume berechnet werden (die Funktion `Math.max` bestimmt das Maximum zweier Zahlen).

Die Funktion `hoehe` ist wieder als Methode formuliert (Zur Wiederholung: Was ist ein bedingter Ausdruck?).

```
public int hoehe() {
    int l = this.left == null  ? 0 :
        this.left.hoehe();
    int r = this.right == null ? 0 :
        this.right.hoehe();
    return Math.max(l, r) + 1;
}
```

Bei der Listenklasse hatten wir die Klasse `Node` für die Knoten und die Klasse `SinglyLinkedList` für den Zugriff auf die Datenstruktur von außen aus. Das können wir bei Bäumen auch so machen. Dann würde man vielleicht ein Methode wie `anzahl` oder `hoehe` in einer entsprechenden Klasse `Tree` implementieren. Dazu muss man aber auf jeden Fall eine zusätzliche Methode schreiben, die die eigentlichen Knoten verbirgt und die Rekursion erstmalig aufruft.

```
// Beispiel fuer Teile einer Tree-Klasse;
class Tree {
    private TreeNode root;

    public int anzahlKnoten() {
        return root == null ? 0 : root.anzahl();
    }
    ...
}
```

[14] Das ist mal wieder ein Problem mit `null`.

6.4.4 Rekursives Traversieren von Binärbäumen

Es ist häufig nötig, alle Datenelemente einer Datenstruktur nacheinander aufzusuchen, um entweder ihre Werte zu erfragen und eventuell auszugeben, oder aber um einzelne oder alle Werte zu verändern. Bei sequentiellen Datenstrukturen ist die Reihenfolge der Werte mehr oder weniger explizit festgelegt. Dies ist bei Bäumen – und damit auch bei Binärbäumen – anders. Da jeder Knoten im zwei Nachfolgerknoten haben kann, gibt es keine eindeutig festgelegte Reihenfolge. Die Reihenfolge der Knoten wird erst durch den verwendeten Algorithmus festgelegt. Die Aufgabe des vollständiges Durchlaufens eines Binärbaums ist im Kern nichts anderes als die Umwandlung der Baumstruktur in eine sequentielle Struktur. Diese Umwandlung kann, aber muss nicht, umkehrbar sein.

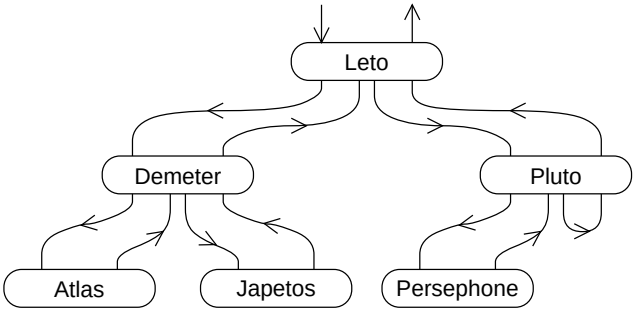

Abbildung 6.10: Rekursive Traversierung eines Binärbaums. Der Weg, der die einzelnen Knoten aufsucht, entspricht genau der Baumstruktur. Dabei wird jeder innere Knoten genau dreimal berührt. Daraus ergeben sich die im Text besprochenen drei verschiedenen Möglichkeiten der rekursiven Traversierung.

Die naheliegendste Formulierung des Traversierungsproblems geht von der rekursiven Struktur eines Binärbaums aus. An jedem Knoten sind genau drei Objekte gegeben: der Wert des Knotens selbst, der linke Teilbaum und der rechte Teilbaum. Bei einem rekursiven Durchlaufen eines Binärbaums ergibt sich der in der Abbildung 6.10 gezeigte Weg, der darauf beruht, dass stets ganze Teilbäume vollständig verarbeitet werden ehe andere Knoten an die Reihe kommen. Wenn wir voraussetzen, dass bei einem geordneten Baum der linke Teilbaum stets vor dem rechten Teilbaum abzuarbeiten ist, ergeben sich drei verschiedene Reihenfolgen, je nachdem ob wir den Wert des Knotens vor den beiden Teilbäumen, zwischen den beiden Teilbäumen oder nach den Teilbäumen verarbeiten. In der Abbildung 6.10 können Sie diese drei Möglichkeiten daran sehen, dass jeder innere Knoten genau dreimal „besucht“ wird, so dass sich die Frage stellt, wann die Verarbeitung des Knotenwerts erfolgen soll. Die drei Alternativen heißen *Preorder-Traversierung*, *Inorder-Traversierung* und *Postorder-Traversierung* (*pre* = *vor*, *in* = *dazwischen*, *post* = *nach*).

Ich muss hier nochmals betonen, dass die hier dargestellten Funktionen, die einen Baum auf dem Bildschirm ausgeben, nur *ein* Beispiel für die Verwendung der Traversierungsoperationen sind. In fast allen etwas komplexeren Anwendungen (wie Compilerbau, Computergraphik oder Datenbanken), gibt es viele weitere Beispiele.

In unserem speziellen Fall sehen die drei Funktionen wie folgt aus. Sie erkennen, dass die Art des Traversierens sich aus der Stellung der Knotenoperation ergibt.

```
public static String preOrder(TreeNode n) {
    if (n == null) return "";
    return
```

```
        " " + n.toString() +
        preOrder(n.left) +
        preOrder(n.right);
}

public static String inOrder(TreeNode n) {
    if (n == null) return "";
    return
        inOrder(n.left) +
        " " + n.toString() +
        inOrder(n.right);
}

public static String postOrder(TreeNode n) {
    if (n == null) return "";
    return
        postOrder(n.left) +
        postOrder(n.right) +
        " " + n.toString();
}
```

Auch hier ist es natürlich möglich, anstelle der Formulierung durch eine statische Funktion eine normale Methode zu verwenden. Zum Beispiel:

```
public String postOrder() {
    String s = "";
    if (left != null) s = s  + left.postOrder();
    if (right != null)  s = s + right.postOrder();
    return s + " " + this.toString();
}
```

Im nächsten Abschnitt bespreche ich als weitere Traversierungsart das ebenenweise Durchlaufen des Baums (*Levelorder-Traversierung*), die im Unterschied zu den hier besprochenen Verfahren nicht rekursiv definiert ist.

Zunächst will ich Ihnen eine konkrete Anwendung für die rekursiven Baumtraversierung zeigen. Die Klasse `TreeNode` ist eine abstrakte Schnittstelle. Daher lässt sich mit ihr allein noch kein ablauffähiges Programm schreiben. Wir müssen vielmehr zunächst eine konkrete Klasse definieren. Als einfaches Beispiel greife ich hier die Darstellung von arithmetischen Ausdrücken durch Binärbäume auf. Dabei gehe ich davon aus, dass in einem arithmetischen Ausdruck Maloperationen, Plusoperationen und numerische Werte vorkommen. Diese drei Arten von Objekten werden durch drei verschiedene Klassen ausgedrückt.

```
public class PlusNode extends TreeNode {
    public PlusNode(TreeNode l, TreeNode r) {
        super(l, r);
    }

    public String toString() {
        return "+";
    }
}

public class MultNode extends TreeNode {
    public MultNode (TreeNode l, TreeNode r) {
        super(l, r);
    }
```

```
    public String toString() {
        return "*";
    }
}

public class NumNode extends TreeNode {
    private double value;

    public NumNode(double value) {
        super(null, null);
        this.value = value;
    }

    public String toString() {
        return String.valueOf(value);
    }
}
```

Häufig werde ich gefragt, wie denn nun ein kompletter Baum in einem Programm aufgebaut werden soll, wo doch ein Konstruktor immer nur einen einzigen Knoten erzeugt. Darauf gibt es zwei Antworten. Die wichtigste Antwort besteht darin, dass ein Programm, das mit Baumstrukturen arbeitet, Bäume nach und nach aufbaut. Im nächsten Kapitel werden Sie Beispiele für dieses Vorgehen sehen. Die andere Möglichkeit besteht darin, dass man die nötigen Konstruktoraufrufe geeignet schachtelt. Zum Beispiel könnte der Ausdruck aus Abbildung 6.9, nämlich $3 + 17 \cdot 8$, so erzeugt werden:

```
TreeNode e =
    new PlusNode(
        new NumNode(3),
        new MultNode(
            new NumNode(17),
            new NumNode(8)));
```

Egal wie der Baum aufgebaut wurde, hier setze ich einfach voraus, dass die Baumstruktur existiert und ausgegeben werden soll. Es ist interessant, die Ausgaben zu vergleichen, die mit den verschiedenen Traversierungsarten aus diesem arithmetischen Ausdruck $3+17\cdot 8$ erzeugt werden. Wir erhalten bei der Preoder-Traversierung: `+ 3 * 17 8`, bei Inorder: `3 + 17 * 8` und bei Postorder `3 17 8 * +`.

Die Inorder-Darstellung scheint die „normale“ Ausgabe des Ausdrucks zu ergeben. Das ist jedoch nicht ganz richtig. Versuchen Sie doch nur einmal den Ausdruck $(3+17)\cdot 8$ als Baum aufzufassen und auszugeben! Sie werden auch dabei dieselbe Inoder-Ausgabe wie vorher bei $3 + 17 \cdot 8$ erhalten. Um die Inorder-Darstellung eindeutig zu machen, benötigen wir Klammern, mit denen wir darstellen können, auf welche Werte die Operatoren anzuwenden sind.

Die beiden anderen Ausgaben sind auch ohne Klammerung stets eindeutige Darstellungen von arithmetischen Ausdrücken. Diesen Sachverhalt hat der polnische Mathematiker Lukasiewicz erkannt. Nach ihm heißt die Preorder-Form *polnische Notation*. Davon ist dann später der Name *umgekehrt polnische Notation* für die Postorder-Form abgeleitet worden. Sie sehen also, dass die verschiedenen Schreibweisen für arithmetische Ausdrücke nur verschiedene sequentielle Darstellungen eines Binärbaums sind. Wenn ein arithmetischer Ausdruck als Binärbaum gegeben ist, lässt er sich mit den rekursiven Baumtraversierungen leicht in den verschiedenen Formen ausgeben. Dies wird im Übrigen auch von dem Java-Compiler bei jeder Programmübersetzung durchgespielt. Während ein Java-

Programme nämlich nur die Inorder-Form von arithmetischen Ausdrücken enthält, kennt der Java-Bytecode nur die einfacher zu interpretierende Postorder-Form.

6.4.5 Ebenenweise Baumtraversierung mittels Warteschlange

Wenn Bäume in einer anderen Reihenfolge durchlaufen werden sollen, kommt man nicht darum herum, über andere Verfahren nachzudenken. Ein solches Beispiel ist das ebenenweise Durchlaufen des Baumes (*Levelorder*), das in Abbildung 6.11 dargestellt ist. Ein Beispiel für die Levelorder-Traversierung ist die nach der Hierarchiestufe geordnete Ausgabe der Namen der Manager einer Firma. Die Namen sollen also etwa in der Reihenfolge: Vorstandsvorsitzender, Vorstandsmitglieder, Bereichsleiter usw., ausgegeben werden.

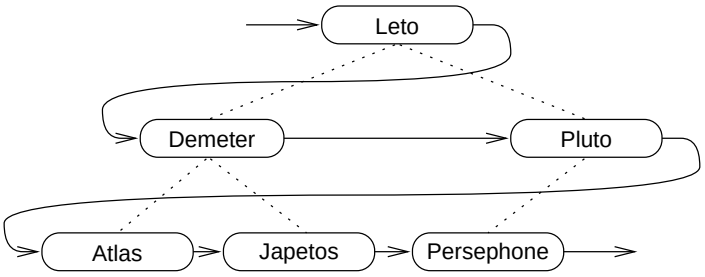

Abbildung 6.11: Ebenenweise Traversierung eines Binärbaums. Da der Weg nicht der gestrichelt dargestellten Baumstruktur entspricht, ist diese Traversierungsart nicht durch rekursives Durchlaufen des Baumes zu erreichen.

Die Traversierung eines Baumes in der Reihenfolge der Ebenen ist – wie Sie an dem Managerbeispiel sehen – nicht an Binärbäume gebunden. Da wir uns hier aber nur Binärbäume näher angesehen haben, will ich das Verfahren auch an diesem Beispiel erläutern.

Das Verfahren zur Programmierung der Levelorder-Traversierung benötigt eine Variable `p`, die immer auf den aktuell zu verarbeitenden Knoten zeigt. Da dieser Knoten auf jeden Fall vor seinen Kindknoten (und auch vor eventuellen Geschwisterknoten) ausgegeben werden soll, erfolgt zunächst die Ausgabe des Knotenwertes. Danach geht es darum festzulegen, welche Knoten weiter zu verarbeiten sind. Dabei soll die Reihenfolge im Baum von links nach rechts und von oben nach unten beibehalten werden. Dies wird durch eine Warteschlangenstruktur erreicht, die ja gemäß dem FIFO-Prinzip die Reihenfolge beibehält. Da zunächst Knoten einer niedrigen Ebenennummer und dabei auch von links nach rechts eingeordnet werden, werden sie schließlich auch in dieser Reihenfolge aus der Warteschlange entnommen und verarbeitet. Die Warteschlange spielt also hier die Rolle eines „Auftragsbuches“, in dem die Referenzen der zu bearbeitenden Knoten gespeichert sind. Die Initialisierung erfolgt durch das Eintragen des Wurzelknotens, und die Bearbeitungsschleife wird beendet, sobald alle Aufträge erledigt sind und die Queue leer ist.

Als ein Beispiel für diese Strategie nehmen Sie einmal an, dass beim Durchlaufen des Baums der Abbildung 6.11 gerade der Knoten „Demeter“ erreicht worden ist. Zu diesem Zeitpunkt befindet sich nur der Knoten „Pluto“ in der Warteschlange (genaugenommen: die Referenz von „Pluto“). Das Wort „Demeter“ wird jetzt ausgegeben und die beiden Kindknoten von „Demeter“ werden in die Warteschlange eingefügt, die daraufhin die Knoten „Pluto“, „Atlas“ und „Japetos“ enthält. Als nächster Knoten kommt also „Pluto“ an die Reihe, der schließlich noch „Persephone“ der Warteschlange hinzufügt. Da die jetzt in der Queue befindlichen Knoten „Atlas“, „Japetos“ und „Persephone“ keine Kind-

knoten besitzen, werden keine neuen Knoten mehr hinzugefügt, so dass die Traversierung nach der Ausgabe der drei Knotennamen terminiert.

Bei der Traversierung kann es vorkommen, dass sehr viele Knoten zu speichern sind, nämlich mindestens alle Knoten einer Ebene. Daher ist es wohl angebracht, zur Implementierung der Warteschlange eine verkettete Liste zu wählen.

Wir können die in diesem Kapitel besprochene `ListQueue` benutzen; oder wir halten uns an die Java-Bibliothek. Ich verwende hier die Schnittstelle `Queue` und die Klasse `LinkedList`.

```
import java.util.Queue;
import java.util.LinkedList;
...

public void levelOrder() {
    // Initialisierung der Warteschlange
    // mit dem Wurzelknoten:
    Queue<TreeNode> q = new LinkedList<TreeNode>();
    q.add(this);
    // alle Teilbaeume aus q bearbeiten:
    while (!q.isEmpty()) {
        // naechsten Knoten holen:
        TreeNode p = q.remove();
        // Knoten bearbeiten:
        System.out.print(p.toString() + " ");
        // linkes Kind vormerken:
        if (p.left != null) q.add(p.left);
        // rechtes Kind vormerken:
        if (p.right != null) q.add(p.right);
    }
}
```

Das hier beschriebene Verfahren der ebenenweisen Baumtraversierung bildet die Grundlage für einige algorithmische Verfahren wie zum Beispiel das Problemlösen nach der Methode der Breitensuche. Bei der Breitensuche – einer der wichtigsten Problemlösungsstrategien der Künstlichen Intelligenz – wird ein Suchbaum so lange ebenenweise durchlaufen, bis der gesuchte Knoten gefunden ist. Wegen der Ähnlichkeit zur Levelorder-Traversierung ist klar, dass die Breitensuche mit Hilfe einer Warteschlange implementiert wird.

Man kann Stacks zur Eliminierung von Rekursion kann. Dies gilt natürlich auch für die Verfahren der rekursiven Baumtraversierung. Die hier angegebenen Funktion `preorder2` (bei der ich mal wieder zur Methoden-Form gewechselt bin) zeigt, dass dies fast genauso aussieht wie `levelOrder`. Die anderen Traversierungen sind allerdings etwas komplizierter. Versuchen Sie es einmal!

```
import java.util.Deque;
import java.util.LinkedList;
...

public void preOrder2() {
    Deque<TreeNode> s = new LinkedList<TreeNode>();
    s.addFirst(this);
    while (!s.isEmpty()) {
        // naechsten Knoten holen:
        TreeNode p = s.removeFirst();
        // Knoten bearbeiten:
        System.out.print(this + " ");
```

```
            // rechtes Kind vormerken:
            if (p.right != null) s.addFirst(p.right);
            // linkes Kind vormerken:
            if (p.left != null) s.addFirst(p.left);
        }
    }
```

Auch bei diesem Beispiel verwende ich wieder die Java-Bibliothek. Hier ist die Klasse `LinkedList` durch die Schnittstelle `Deque` auf Operationen am Anfang und Ende der Liste beschränkt (in diesem Fall auf den Stack).

Kapitel 7

Suchen und Sortieren

Due credit must be paid to the genius of the designers of ALGOL 60
who included recursion in their language
and enabled me to describe my invention[1]
so elegantly to the world.
C.A.R. Hoare, The Emperor's Old Clothes

In diesem Kapitel werden wir ein paar der wichtigsten Algorithmen behandeln, die jeder Informatiker kennen sollte. Dazu gehört das effiziente Suchen, das Sortieren und die effiziente Programmierung von Datenstrukturen, die beliebige Daten unter einem Suchschlüssel abspeichern. Das Kapitel ist so aufgebaut, dass bei allen Verfahren zunächst eine algorithmenunabhängige Schnittstelle diskutiert wird. Anschließend wird zunächst ein einfaches aber meist ineffizientes Lösungsverfahren dargestellt und danach werden dann ein oder mehrere bessere Verfahren besprochen.

Grundsätzlich findet Suche und Sortieren in unterschiedlich implementierten Datenbeständen statt. In diesem Kapitel werden nur Algorithmen besprochen, die auf Feldern, auf verketteten Listen und auf Binärbäumen operieren.

7.1 Effizientes Suchen in Feldern

In sehr vielen Computeranwendungen muss häufig nach bestimmten Inhalten gesucht werden. Der bei der Suche verwendete Algorithmus kann einen großen Einfluss auf das Laufzeitverhalten eines Programms haben.

Im Folgenden vergleiche ich zwei verschiedene (bekannte) Verfahren der Suche in einem Feld. Die erste Methode ist die *lineare Suche* mit einem Laufzeitverhalten von $O(n)$. Die zweite Methode ist die *binäre Suche* der Klasse $O(\log n)$. Im Unterschied zur linearen Suche setzt die binäre Suche jedoch ein sortiertes Feld voraus. Sortieren, das im nächsten Abschnitt besprochen wird, ist also oft die Voraussetzung für effiziente Suche.

Bei der Beschreibung von Algorithmen, die auf Feldern operieren, ist es nötig, Aussagen über ganze Feldbereiche zu machen. Um solche Formulierungen halbwegs lesbar zu halten, verwende ich eine besondere Schreibweise. Durch den Ausdruck `a[i:j]` gebe ich an, dass der gesamte Bereich von `a[i]` bis zu (und einschließlich) `a[j]`) gemeint ist. Die Aussage, dass die Zahl m größer ist als alle Inhalte des n-elementigen Feldes `a`, lautet in dieser Schreibweise einfach `a[0:n-1] < m`.

[1] Gemeint ist der Sortieralgorithmus *Quicksort*.

In diesem Kapitel stehen die algorithmischen Verfahren und nicht die objektorientierte Programmierung im Vordergrund. Deshalb werden alle Suchverfahren und Sortierverfahren als statische Funktionen formuliert.

Es gibt aber doch eine Fragestellung, die auf die besonderen Lösungsmöglichkeiten in einer objektorientierten Programmiersprache wie Java abzielt. Die Frage ergibt sich, wenn wir uns daran machen, eine richtige Algorithmenbibliothek zu schreiben. Das Problem besteht darin, dass Java eine streng getypte Sprache ist, die für die Formulierung eines Algorithmus die genaue Angabe des Datentyps verlangt, dass andererseits aber Algorithmen zum Suchen und Sortieren für Daten von vielen verschiedenen Datentypen benötigt werden. Zur Lösung dieses Problems gibt es mehrere Möglichkeiten:

1. Für jeden Datentyp schreiben wir einen speziellen Algorithmus. Technisch ist damit kein Problem verbunden – aber die Bibliothek kann nie vollständig sein, da wir ja unser Java-Programm immer wieder um neue Datentypen erweitern werden.

2. Wir nutzen das Konzept der Template-Programmierung. Dabei werden aus einer vorgegebenen Schablone für das Suchverfahren automatisch konkrete Algorithmen für den jeweiligen Datentyp generiert. Diese Variante wird von der Programmiersprache C++ angeboten.

3. Wir formulieren spezielle Suchverfahren für die einfachen Wertdatentypen und einen generellen Algorithmus für alle Referenzdatentypen. Die von `Object` abgeleiteten Referenzdatentypen verfügen alle über eine Methode `equals` mit der sich die Gleichheit zweier Objekte feststellen lässt. Damit ist es ohne weiteres möglich, einen ganz allgemeinen Suchalgorithmus zu schreiben, der für alle Objekte funktioniert. Die binäre Suche und alle Sortierverfahren benötigen aber zusätzlich die Möglichkeit Vergleiche („größer“ oder „kleiner“) ausführen zu können. Aus guten Gründen sieht die Klasse `Object` aber keine entsprechende Vergleichsoperation vor (warum nicht?). Vergleiche sind jedoch möglich, wenn die entsprechenden Objekte die Schnittstelle `Comparable` implementieren. Wie Sie sich erinnern, definiert `Comparable` die Operation `compareTo`, die einen solchen Vergleich ausführt. Das Verfahren wird bei einigen Java-Klassen verwendet. Es ist immer dann sinnvoll, wenn es eine („natürliche“) Anordnung der Objekte gibt, die man in ihrer Klasse festlegen kann. Der Nachteil ist, dass in der Klasse der Objekte nur eine einzige Anordnung festgelegt werden kann.

4. Eine flexiblere Möglichkeit der Festlegung der Reihenfolge liefern Algorithmen, die die Durchführung von Vergleichen einem speziellen Objekt übertragen. Jedes Vergleichsobjekt verfügt über eine Methode `compare` mit der zwei Datenobjekte verglichen werden. Die Schnittstelle aller Vergleichsobjekte ist in dem Interface `java.util.Comparator` definiert. Je nach Vergleichsobjekt können die Daten unterschiedlich angeordnet sein (nach Größe, nach Alter, nach Gewicht usw.). Auch dieser Ansatz wird in Java viel verwendet.

5. Gleichzeitig lassen sich die Suchalgorithmen in Java durch Typparameter beschreiben. Damit sind aber ein paar kompliziertere Sachverhalte verbunden, so dass ich hier nicht darauf eingehe.

Da es mehr auf die Algorithmen als auf die Datenstrukturen ankommt, werde ich wiederholt die Algorithmen auch anhand der elementaren Datentypen erläutern.

Das folgende Listing stellt die hier verwendeten Signaturen von Suchfunktionen zusammen:

```
/* Generelle Aufgabenspezifikation der Suche:
 * suche in dem n-elementigen Feld a nach dem Wert key
 *   wenn gefunden: Resultat = Feldindex i von key
 *                             0 <= i < n
 *   sonst:         Resultat kleiner als 0
 */

/**
 * Finde die Position einer ganzen Zahl in einem Feld.
 * @param a int-Feld.
 * @param key zu suchende Zahl.
 * @return wenn gefunden die Position, sonst negative Zahl.
 */
public static int search(int[] a, int key)

/**
 * Finde die Position eines Objekts in einem Feld.
 * @param a Feld.
 * @param key zu suchendes Objekt.
 * @return wenn gefunden die Position, sonst negative Zahl.
 */
public static int search(Object[] a, Object key)

/**
 * Finde die Position eines Objekts in einem Feld.
 * @param a Feld.
 * @param key zu suchendes Objekt.
 * @param cmp Comparator-Objekt, zum Groessenvergleich.
 * @return wenn gefunden die Position, sonst negative Zahl.
 */
public static int search(Object[] a, Object key,
       Comparator cmp)
```

Die Schnittstelle `Comparator` aus `java.util` sieht so aus:

```
public interface Comparator<T> {
    /**
     * Vergleicht zwei Objekte a und b
     * @param a 1. Objekt.
     * @param b 2. Objekt.
     * @return 0: a == b, negativ: a < b, positiv: a > b
     * @throws ClassCastException, bei falschem Objekttyp.
     */
     int compare(T a, T b);
}
```

Die Schnittstelle ist, wie es sich gehört, mit Typparametern geschrieben.

7.1.1 Die lineare Suche

Die lineare Suche ist so einfach, dass sie praktisch von jedem Programmierer immer wieder neu „entdeckt“ wird. Sie zeigt auch ein weiteres Beispiel für die Verwendung einer Schleifeninvariante. Beachten Sie, dass sich diese Funktion mit der Methode `equals` ganz allgemein für beliebige Referenztypen hinschreiben lässt.

```
public static final int NOT_FOUND = -1;
```

```
public static int search(int[] a, int key) {
    // Invariante: key ist nicht in a[0:i-1]
    // Ziel der Schleife: i=a.length oder a[i]=key
    int i = 0;
    while (i != a.length && key != a[i]) {
        // key nicht in a[0:i]
        i += 1;
    }
    // Ziel: key ist nicht in a[0:i-1]
    //       und (i=a.length oder a[i]=key)
    return i < a.length ? i : NOT_FOUND;
}

/**
 * Sucht die Position eines Objekts in einem Array.
 * <p>
 * Vorbedingung: das Objekt <tt>key</tt> darf nicht
 *     <tt>null</tt> sein.
 * @param key zu suchendes Objekt
 * @return index der Position oder -1
 */
public static int search(Object[] a, Object key) {
    // Invariante: key ist nicht in a[0:i-1]
    // Ziel der Schleife: i=a.length oder a[i]=key
    int i = 0;
    while (i != a.length && !key.equals(a[i])) {
        // key nicht in a[0:i]
        i += 1;
    }
    // Ziel: key ist nicht in a[0:i-1]
    //       und (i=na.length oder a[i]=key)
    return i < a.length ? i : NOT_FOUND;
}
```

Anmerkung:

Der angegebene Algorithmus für Objekte hat die Vorbedingung, dass nicht nach `null` gesucht werden darf. Diese Einschränkung ist etwas meiner Faulheit geschuldet. Wenn Sie wollen, verwenden Sie die im letzten Kapitel vorgestellte Funktion `eq`.

Bei der Lektüre der Zielbedingungen sollten Sie daran denken, was Sie von `search` erwarten. Dazu gehört auch die Überlegung, was bei dem mehrfachen Vorkommen eines Datenelements `key` getan werden soll. In der Zielbedingung wird gefordert, dass der kleinste passende Index (entsprechend dem „ersten“ Element) zurückgegeben wird.

An diesem Beispiel erkennen wir wieder, dass die Invariante eine Abschwächung der Zielbedingung darstellt. Es wurde nämlich nur eine der beiden Und-verknüpften Aussagen der Zielbedingung zur Invarianten gewählt. Dadurch, dass der weggelassene Teil als Abbruchbedingung verwendet wird, ist garantiert, dass das Ziel nach dem Schleifenende auch erreicht ist.

Die Ausführungszeit der linearen Suche hängt davon ab, nach wie vielen Schritten das gesuchte Datenelement `key` gefunden wird. Im ungünstigsten Fall wird `key` nicht gefunden. Dann sind n Vergleiche nötig, und der Aufwand für diesen schlechtesten Fall ist daher $O(n)$. Im Normalfall dürfte das gefundene `key` in der Mitte des Feldes liegen. Auch in diesem durchschnittlichen Fall ist der Algorithmus also $O_{avg}(n)$.

Abschließend sollten Sie sich an diesem einfachen Beispiel auch schon mit der Verwendung eines Comparator-Objekts vertraut machen.

```
public static int search( Object[] a, Object key,
       Comparator cmp) {
    // Invariante: key ist nicht in a[0:i-1]
    // Ziel der Schleife: i=a.length oder a[i]=key
    int i = 0;
    while (i != a.length && cmp.compare(key, a[i]) != 0) {
        // key nicht in a[0:i]
        i++;
    }
    // Ziel: key ist nicht in a[0:i-1]
    //       und (i=a.length oder a[i]=key)
    return i < a.length ? i : NOT_FOUND;
}
```

7.1.2 Die binäre Suche

Das Ergebnis für den Suchaufwand, das wir in dem letzten Abschnitt ermittelt hatten, stimmt eigentlich sehr pessimistisch. Es sagt aus, dass die Suchzeit in einer Tabelle linear mit der Tabellengröße ansteigt.

Wenden wir das einmal auf ein alltägliches Beispiel an. Angenommen, Sie benötigen für die Suche nach einem Schlagwort über objektorientierte Programmierung in einem Glossar mit 100 Begriffen 1–2 Sekunden. Wie lange brauchen Sie um das Wort „trivial“ im Duden nachzuschlagen, vorausgesetzt, der Duden enthält ca. 200000 Schlagworte? — Die Rechnung ist ganz einfach: Da die Suchzeit proportional mit der Größe der Tabelle anwächst, benötigen wir 2000 bis 4000 Sekunden, also ungefähr 1 Stunde, um das Wort „trivial“ zu finden. Machen Sie mal einen Versuch!

Natürlich werden Sie das Wort erheblich schneller finden. Sie werden aber auch sicher nicht suchen, indem Sie alle Schlagworte ab „A“ durchlesen (oder suchen Sie immer noch?). Vielmehr werden Sie in der Regel den Duden irgendwo in der Mitte aufschlagen, sehen, dass dort Worte, wie „Karriere“ stehen und dann werden Sie schlagartig entscheiden, dass „trivial“ weiter hinten stehen muss, wo Sie dann schließlich weitersuchen.

Sie sollten sich klar machen, dass diese effiziente Schlagwortsuche an eine wichtige Voraussetzung gekoppelt ist! Ihre Entscheidung, nach dem ersten Vergleich nur noch im letzten Teil des Dudens weiterzusuchen, ist nur deshalb vernünftig, weil Sie wissen, dass die Schlagworte alphabetisch aufsteigend sortiert sind. Wäre dem nicht so, würde dieses Verfahren nicht funktionieren.

Die Analogie der Schlagwortsuche können wir auch bei Computer-Algorithmen nutzen. Wir müssen nur voraussetzen, dass das Feld (aufsteigend) sortiert ist und dass der Datentyp der Feldelemente einen Größenvergleich ermöglicht. Die Grundidee besteht dann darin, dass wir den Suchbereich, der durch die beiden Variablen `i` und `j` begrenzt wird, möglichst schnell verkleinern, indem wir das mittlere Element dieses Intervalls zum Vergleich heranziehen. Dadurch ist es möglich, wie bei der Schlagwortsuche, das Suchintervall bei jedem Vergleich in etwa zu halbieren. Wie Sie leicht herausfinden, sollte die Laufzeit $O(\log_2 n)$ sein.

```
public static int search(Object[] a, Object key,
       Comparator cmp) {
    /* Vorbedingung:
     *     n>=0 und a ist aufsteigend sortiert
     * Ziel der Schleife:
     *     key ist nicht in a[0:i-1]
```

```
     *     und (i=n oder a[i]=x)
     * Invariante:
     *          (i<=j oder key ist nicht in a[0:n-1])
     *     und key ist nicht in a[0:i-1]
     *     und key ist nicht in a[j:n-1]
     */
    int i = 0;           // untere Intervallgrenze
    int j = a.length-1;  // obere  Intervallgrenze
    int m = (i + j) / 2; // Intervallmitte
    while (i <= j && cmp.compare(key, a[m]) != 0) {
        if(cmp.compare(key, a[m]) < 0)
            j = m - 1;
        else
            i = m + 1;
        m = (i + j) / 2;
    }
    if (i <= j)
        return m;  // gefunden
    else
        return -(i+1);  // nicht gefunden
}
```

Eigentlich ist der Algorithmus[2] nicht viel komplizierter als die lineare Suche. Wenn Sie sehr häufig in einer Tabelle suchen, lohnt es sich sogar, vorher die gesamte Tabelle zu sortieren – allerdings nur dann, wenn sich die Tabelle nicht ständig ändert.

Beachten Sie bitte den für den Fall des Nichtfindens angegebenen Rückgabewert. Wenn Sie zurückblättern auf Seite 182 werden Sie feststellen, dass ich da geschrieben habe, dass wenn das gesuchte Objekt nicht gefunden wurde, eine negative Zahl zurückgegeben werden soll. Bei der linearen Suche bin ich dieser (selbst gestellten) Forderung gerecht geworden, indem ich bei nicht gefundenem Wert einfach eine -1 zurückgegeben habe. Dies hätte man in der binären Suche auch so machen können. Die hier gewählte Form ist etwas besser (und entspricht auch der Lösung der Java-Bibiliothek). Ich gebe nämlich als Information zurück, an welche Stelle der gesuchte Wert in dem (sortierten) Feld gehört. Genauer: der Rückgabewert ist gleich dem negativen Feldindex *vor* dem der gesuchte Wert stehen müsste minus 1. Die Subtraktion ist nötig, da sonst eine Verwechselung mit der Rückgabe „Schlüssel wurde bei Index 0 gefunden“ möglich wäre. Überlegen Sie selbst, wozu diese Information verwendet werden kann und warum man das nicht auch bei der linearen Suche macht!

Wenn ich in der Vorlesung die binäre Suche besprochen habe, werde ich häufig von Studenten gefragt, ob nicht noch eine weitere Verbesserung möglich ist. Die Studenten sagen, dass sie bei der Suche in einem Telefonbuch oder einem Duden nicht stur die Mitte aufschlagen um zu entscheiden, wo weiterzusuchen ist, sondern dass sie vorher anhand des Suchbegriffs abschätzen, wo das gesuchte Wort in etwa zu finden sein sollte.

Dieser Einwand ist berechtigt. Man kann die Intervallsuche tatsächlich durch eine solche Abschätzung nochmals deutlich verbessern. Der Algorithmus ist in der Literatur als *Interpolationssuche* bekannt. Der Suchaufwand ist dabei im Mittel $O(\log(\log n))$, d.h. er ist beinahe unabhängig von der Tabellengröße.

Anmerkung:

Wie findet man heraus, dass die Laufzeit der binären Suche gleich $O(\log n)$ ist? Die Anweisungen vor der While-Schleife werden höchstens einmal ausgeführt. Dagegen werden die

[2] Wenn Sie meinen, dass man den Algorithmus einfacher und übersichtlicher schreiben kann, will ich nicht widersprechen. Meine Implementierung hebt die Ähnlichkeit zur linearen Suche hervor.

Anweisungen der Schleifenanweisungen wiederholt. Wenn sie oft genug ausgeführt werden, wird fast die gesamte Laufzeit auf diesen Bereich entfallen. Die entscheidende Frage ist: „Wie oft wird die Schleife bei der Suche in einem Feld der Länge n höchstens ausgeführt?“. Eine unüberlegte Antwort könnte lauten n-Mal, da wir ja eine Schleife über die Elemente eines Feldes der Länge n haben. Diese Antwort ist falsch! Wenn wir genauer hinsehen, erkennen wir, dass die Variablen i und j die den noch zu bearbeitenden Bereich eingrenzen, so definiert sind, dass sich bei jedem Schleifendurchlauf ihre Differenz $j - i$ halbiert und dass spätestens wenn $i = j$ ist, die Suche abgebrochen wird. Die Anzahl der möglichen Durchläufe ist daher gleich der Anzahl der möglichen (ganzzahligen) Halbierungen einer Zahl n. Nach der Mathematik (Potenzrechnung und Logarithmen) ist dies $\log_2 n$.

7.2 Effizientes Sortieren in Feldern

In diesem Abschnitt bespreche ich zwei Möglichkeiten, wie man Felder sortieren kann. Genauso wie beim Suchen ist das eine Verfahren in dem Sinne „naiv“, dass man einfach darauf kommen kann. Die Laufzeit dieses Verfahrens ist $O(n^2)$. Andere bessere Verfahren – z.B. Quicksort, Mergesort oder Heapsort – laufen dagegen (im Mittel, z.T. auch im ungünstigsten Fall) mit $O(n \log n)$.

Auch hier lege ich keinen Wert auf eine besondere objektorientierte Darstellung. Objektorientierung kann dann sinnvoll eingesetzt werden, wenn man Behälterklassen definiert, die *garantiert* sortiert sind, bei denen also bei jedem neuen Einfügen eine richtige Reihenfolge hergestellt wird. Für diese Anwendung sind die hier besprochenen Methoden jedoch nicht besonders gut geeignet. Dieses Problem wird aber weiter unten aufgegriffen.

7.2.1 Worum geht es beim Sortieren?

Die allgemeine Schnittstelle einer Sortierfunktion für Felder von Objekten kann etwa so aussehen:

```
public static void sort(Object[] a, Comparator cmp)
```

Aber wie sollen wir jetzt bei der Implementierung vorgehen? Aus den bisherigen Beispielen sollten Sie gelernt haben, dass es keinen Sinn macht, mit dem Programmieren anzufangen, solange Sie sich noch keine Klarheit über die Aufgabe, d.h. die Spezifikation des Problems verschafft haben.

Sortieren gehört erfreulicherweise zu den wenigen Problemen, die wir einfach und exakt spezifizieren können. Im folgenden verwende ich in Kommentaren und Spezifikationen die üblichen Operatoren (z.B. <) und nicht die umständliche Comparator-Schreibweise.

```
// Q: a ist ein Feld mit n Elementen und A = a
Sort(a, n);
// R: a enthaelt exakt dieselben Elemente wie A
// und a[i] <= a[j] fuer 0 <= i < j < n
```

Die Vorbedingung Q sagt unter anderem aus, dass mit `A` die ursprünglichen Werte des Feldes `a` benannt werden sollen (`A` ist *keine* Programmvariable). hier und generell in der Besprechung von Sortieralgorithmen verwende ich das Symbol n zur Bezeichnung der Feldgröße, d.h. der Anzahl der zu sortierenden Objekte. Sie wissen natürlich, dass man

die Feldgröße in Java durch den Ausdruck `a.length` berechnen kann. Sortieren bedeutet nach dieser Spezifikation, dass erstens alle Elemente des ursprünglichen Feldes nachher noch vorhanden sind, und dass zweitens diese Elemente aufsteigend geordnet sind. Man kann das auch so ausdrücken: *ein sortiertes Feld ist eine geordnete Permutation des Feldes*.

Oft ist es übersichtlicher, wenn man die verbale Spezifikation durch halbformale Ausdrücke hinschreibt. Dadurch wird nämlich die logische Struktur der Aussage deutlicher. Dazu definieren wir für die elementaren Teilaussagen sogenannte Prädikatsfunktionen, das heißt Funktionen, die genau die Wahrheitswerte *wahr* und *falsch* annehmen können. Natürlich werden diese Funktionen *nicht programmiert*, da es sich ja um reine Hilfskonstrukte für die Programmspezifikation handelt. Zusammen mit diesen Hilfsdefinitionen lautet unser bisheriges Programmstück:

```
// permutation(A, a):
//    a ist Permutation von A
// geordnet(a):
//    wenn a n Elemente hat, gilt
//    fuer 0<=i<j<n: a[i] <= a[j]
//
// Q: a[0:n-1] = A[0:n-1]
Sort(a, n);
// R: permutation(A[0:n-1], a[0:n-1])
//    && geordnet(a[0:n-1])
```

Solche Überlegungen sind kein rein akademisches Spiel, sondern führen oft durch die damit verbundene Strukturierung der Gedanken zu der richtigen Programmidee. Häufig gibt die Problemstellung nämlich bereits Hinweise über den Lösungsweg. So auch hier. Man kann nämlich die beiden oben angeführten Ziele mit unterschiedlichen Mitteln erreichen:

1. Die Bedingung, dass ein sortiertes Feld eine Permutation des Ausgangsfeldes ist, ist automatisch erfüllt, wenn die einzige Veränderung von Inhalten des Feldes `a` das Vertauschen zweier Feldinhalte ist.
2. Man erhält ein geordnetes Feld, wenn man eine *geeignete* Folge von Vertauschungen durchführt.

Nach dem, was wir bereits gelernt haben, können wir die eine Teiloperation – die Vertauschung – sofort als Prozedur aufschreiben:

```
Object temp = a[i];
a[i] = a[j];
a[j] = temp;
```

Wenn Sie auf Effizienz keinen Wert legen, kann ich Ihnen auch sofort einen korrekten Sortieralgorithmus angeben:

Sortieralgorithmus 0: *Probiere der Reihe nach alle Permutationen des Feldes* `a` *aus. Überprüfe dann, ob diese Elemente vollständig geordnet sind, ob also gilt:* `geordnet(a[0:n-1])`. *Ist dies der Fall, so hast du das sortierte Feld gefunden.*

Also was hält Sie ab, das Problem nach diesem Algorithmus zu lösen? — Er scheint doch ganz einfach zu sein.

Aber lassen Sie uns vorher noch überlegen, wie groß die zu erwartende Laufzeit sein wird. Es wäre nämlich nicht gut, wenn wir Aufwand in die Entwicklung eines unbrauchbaren Verfahrens steckten.

Wenn wir Glück haben, ist die Laufzeit exzellent. Dann ist das Feld bereits sortiert! Wir müssen uns dann nur durch ein einmaliges Durchgehen des ganzen Feldes davon überzeugen, dass das Feld auch wirklich sortiert ist. Es gilt also:

$$T_{\text{best case}} = O(n)$$

Aber was ist mit dem ungünstigsten Fall? In dem Fall müssen wir mit unserem Algorithmus alle Permutationen durchprobieren (und jedes Mal überprüfen ob das Feld sortiert ist), bis wir zu guter Letzt ein geordnetes Feld erhalten. Aus der Mathematik wissen Sie, dass eine Folge von n Zahlen $n!$ Permutationen besitzt. Weiter wissen wir, dass wir für jede dieser $n!$ Permutationen $O(n)$ Vergleichsoperationen zur Überprüfung der richtigen Reihenfolge durchführen müssen und ebenso mindestens eine Vertauschung, um von einer zur nächsten Permutation zu kommen. Damit kommen wir auf ca. $n \cdot n!$ Operationen. Für die grobe O-Notation nähert man die Fakultätsfunktion durch die Exponentialfunktion an und lässt alle Faktoren weg, die für das genäherte Zeitverhalten keine Rolle spielen. Dabei bleibt hier nur noch die Exponentialfunktion übrig (auch eine kompliziertere Formel führt am Ende zu keinen anderen Konsequenzen):

$$T_{\text{worst case}} = O(e^n)$$

Da die Exponentialfunktion schon bei kleinen Werten von n stark ansteigt, sind Algorithmen mit exponentiellem Laufzeitverhalten nur sehr eingeschränkt brauchbar. Wir sollten uns also ein anderes Verfahren überlegen.

7.2.2 Ein einfacher Sortieralgorithmus

Viele Algorithmen gehen davon aus, dass man das sortierte Feld nach und nach aufbaut. Dadurch, dass man Feldelemente, die bereits endgültig festgelegt wurden, nicht mehr zu verändern braucht, sollte die Anzahl der nötigen Schritte viel kleiner als bei der Permutationsmethode sein. Die Lösungsidee lässt sich formal exakt in der Schleifeninvarianten darstellen. Da ich voraussetze, dass wir das Feld `a` nur durch Vertauschungen verändern, beachte ich die Permutationsbedingung nicht mehr besonders, sondern führe nur noch die Forderung der geordneten Reihenfolge auf. Damit lässt sich ein erster Sortieralgorithmus in groben Zügen so schreiben:

```
public static void sort(Object a[], Comparator cmp) {
    int n = a.length; // zur besseren Verstaendlichkeit
    for (int i = 0 ; i < n; i++) {
       // Inv: geordnet(a[0:i-1])
       // .. fehlt noch
       // geordnet(a[0:i])
    }
    // R: geordnet(a[0:n-1])
}
```

Bis auf das fehlende Teilstück ist dieser Algorithmus bestimmt korrekt. Sie sehen, dass bei der Beendigung der Schleife `i=n` gilt. Wenn die Invariante wirklich bis zuletzt erfüllt ist, gilt automatisch die Aussage R.

Damit der Algorithmus nicht zu kompliziert wird, können wir den Begriff *geordnet* dahingehend einengen, dass in dem jeweiligen Feldabschnitt bereits die endgültigen Werte stehen, so dass dieser Abschnitt bereits fertig sortiert ist[3].

Es bleibt also nur noch die Frage, wie das fehlende Programmstück aussehen muss. Beim ersten Eintritt in die Schleife ist die Laufvariable gleich 0. Die Invariante sagt aus, dass der Feldabschnitt `a[0:-1]` fertig sortiert ist. Da dieser Abschnitt anfangs (wegen der unzulässigen Grenzen) leer ist, ist die Bedingung trivialerweise erfüllt. Nach dem ersten Durchlaufen des Schleifenkörpers soll dagegen bereits `a[0:0]` fertig sortiert sein. Dies bedeutet, dass in `a[0]` das minimale Feldelement steht! Als nächstes sollten wir aber noch den allgemeinen Fall überlegen. Die Feldelemente `a[0:i-1]` sind jetzt fertig sortiert. Es geht darum, aus dem Teilfeld `a[i:n-1]` das Element auszuwählen, das nach `a[i]` gehört. Dieses gesuchte Feldelement ist das Minimum des Restfeldes `a[i:n-1]`!

Bei diesem Verfahren, wird bei dem i-ten Schleifendurchlauf das Feldelement für die i-te Feldposition ausgewählt. Der Algorithmus heißt daher auch *Sortieren durch direkte Auswahl* oder *selection sort*.

Nach dem, was wir bisher überlegt haben, sieht die nächste Verfeinerung so aus:

```
public static void sort(Object a[], Comparator cmp) {
    int n = a.length;
    for (int i = 0 ; i < n; i++) {
        // fertig-geordnet(a[0:i-1])
        int j;
          // !! hier fehlt was !!
        // i<=j<n und a[j] = Minimum(a[i:n-1])
          // vertausche a[i] und a[j];
        // fertig-geordnet(a[0:i])
    }
    // R: fertig-geordnet(a[0:n-1])
}
```

Das Problem ist inzwischen auf eine Teilaufgabe reduziert, die so einfach ist, dass man sofort die endgültige Lösung angeben kann. Um sie von anderen Sortierverfahren zu unterscheiden, habe ich die Methode `selectionSort` genannt.

```
public static
void selectionSort(Object[] a, Comparator comp) {
    int n = a.length;
    for (int i = 0 ; i < n; i++) {
        // fertig-geordnet(a[0:i-1])
        int j = i;
        for (int k=i+1; k < n; k++ )
            if (comp.compare(a[k], a[j]) < 0) j = k;
        // i<=j<n und a[j] = Mininum(a[i:n-1])
        Object t = a[i];
        a[i] = a[j];
        a[j] = t;
        // fertig-geordnet(a[0:i])
    }
    // R: fertig-geordnet(a[0:n-1])
}
```

[3] auch ein anderer Algorithmus „Sortieren durch Einfügen“ erfüllt die Invariante — nur sind dabei die Werte noch nicht endgültig festgelegt

Und wie ist es hier mit der Effizienz? Die Komplexität von iterativen Algorithmen lässt sich meist dadurch angeben, dass man einfach die Anzahl der Durchläufe von ineinander geschachtelten Schleifen miteinander multipliziert. Damit kommt man hier zu dem Ergebnis:

$$T_{\text{selection sort}} = O(n^2)$$

Diese Aussage gilt nur für die innerste Schleife, in der nur Vergleichsoperationen auftreten. Die Zahl der Vertauschungen ist proportional zu n, da Vertauschungen nur in der äußeren Schleife stattfinden. In Java (vor allem dann, wenn man Vergleiche mit einem Comparator ausführt) benötigen Vergleiche mehr Rechenzeit als die relativ effiziente Vertauschung von Objektreferenzen. In anderen Programmiersprache, z.B. in C oder in C++, kann es aber sein, dass bei der Vertauschung große Datenmengen bewegt werden, dann kann dieser Algorithmus relativ günstig sein, da er die Zahl der Vertauschungen minimiert. Allerdings gilt dies nur dann, wenn n nicht zu groß ist. Bei sehr großen n wird immer die stärker anwachsende Zahl der Vergleiche die Laufzeit bestimmen.

Wenn Sie wollen, können Sie die Lesbarkeit eines Algorithmus durch sinnvolle Modularisierung erheblich erhöhen.

Eine Regel, die eine Modularisierung fördert besagt, dass man Innerhalb einer Methode am besten keine Kommentare verwendet, sondern lieber die entsprechende Anweisungsfolge in eine eigene Methode steckt.

Schauen Sie sich das Beispiel an.

```
public static
void selectionSort(Object[] a, Comparator comp) {
    for (int i = 0; i < a.length; i++) {
        // fertig-geordnet(a[0:i-1])
        swap(a, i, minIndex(a, i, comp));
        // fertig-geordnet(a[0:i])
    }
    // R: fertig-geordnet(a[0:n-1])
}
/**
 * Gibt den Index des kleinsten Elements zurueck.
 * Die Suche beginnt bei dem Index <tt>start</tt>.
 * @param a Feld
 * @param start der Anfangsindex
 * @param comp das Comparatorobjekt
 */
private static
int minIndex(Object[] a, int start, Comparator comp) {
    int min = start;
    for (int i = start + 1; i < a.length; i++)
        if (comp.compare(a[i], a[min]) < 0) min = i;
    return min;
}
/**
 * Vertauschen zweier Feldinhalte.
 * @param a Feld
 * @param idx1 Index des 1. Elements
 * @param idx2 Index des 2. Elements
 */
private static void swap(Object[] a, int idx1, int idx2) {
    Object t = a[idx1];
    a[idx1] = a[idx2];
    a[idx2] = t;
}
```

Es gibt eine viele ähnliche Suchverfahren, deren Laufzeit $O(n^2)$ ist. Ein Beispiel ist der Algorithmus „Sortieren durch direktes Einfügen“ (*insertion sort*). Er benötigt nur halb soviel Vergleiche wie die direkte Auswahl (allerdings ist die Zahl der Vertauschungen höher). Von den einfachen Verfahren (zu denen auch das noch schlechtere *bubble sort* gehört) ist er der beste. Auf diese Verfahren gehe ich hier nicht ein. Stattdessen lernen Sie im nächsten Abschnitt einen erheblich effizienteren Sortieralgorithmus kennen.

7.2.3 Ein schneller Sortieralgorithmus

Hier soll jetzt nicht näher auf weitere einfache Sortieralgorithmen eingegangen werden. Sie unterscheiden sich in ein paar Details, sind aber alle in ihrem Laufzeitverhalten von $O(n^2)$. Da es mehrere Verfahren gibt, die sich mit $O(n \log n)$ verhalten, spielen die einfachen Verfahren in der Praxis keine Rolle.

Die Grundidee bei dem gerade besprochenen Algorithmus ist, in jedem Schritt den sortierten Bereich um ein Element zu vergrößern. Diese Eigenschaft von `selectionSort` erkennt man am deutlichsten in der folgenden rekursiv formulierten Varianten.

Anmerkung:

Vorsicht: Das Folgende ist kein exaktes Java (und auch kein C) sondern eine vereinfachte frei erfundene Pseudocode-Schreibweise. Pseudocode wird immer da verwendet, wo es nicht so sehr auf die Programmiersprache ankommt (und das ist bei Algorithmen fast immer so). Um den Pseudocode-Charakter zu unterstreichen, habe ich hier auch deutsche Namen gewählt.

```
// Feld a von i0 bis i1 sortieren
void sortiere(a[i0:i1]) {
    if (i1 > i0) {
       j = indexVonMinimum(a[i0+1,i1])
       vetausche(a[i0], a[j]);
       sortiere(a[i0+1:i1]);
    }
}
```

Weiter vereinfacht kann man diese rekursive Formulierung auch so ausdrücken:

```
void sortiere(a[i0:i1]) {
    if (i1 > i0) {
       // teile a so auf, dass gilt
       // a[i0] <= a[i0+1:i1]
       sortiere(a[i0+1:i1]);
    }
}
```

Der Algorithmus beruht also darauf, dass man bei jeder Wiederholung das Problem auf ein um 1 kleineres Teilproblem reduziert. Die Idee, einen effizienteren Algorithmus zu bekommen, geht davon aus, dass es viel günstiger ist, das Feld in zwei etwa gleich große Teilfelder zu zerlegen, die man dann getrennt sortiert. Der Grund, warum das günstiger sein sollte, liegt in dem $O(n^2)$-Verhalten der „einfachen“ Sortieralgorithmen. Wenn der Aufwand für das Sortieren eines Feldes aus n Elementen proportional zu n^2 ist, ist nämlich der Aufwand für das Sortieren einer Hälfte des Feldes nur $n^2/4$ und zum getrennten Sortieren beider Hälften $n^2/2$. Indem man diese Grundidee rekursiv weiterverfolgt,

sollte sich eine beträchtliche Verbesserung erreichen lassen.[4]

Damit das funktionieren kann, muss man aber zuerst dafür sorgen, dass durch getrenntes Sortieren das richtige Resultat erreicht wird. Es gibt zwei unterschiedliche schnelle Sortierverfahren, die beide auf der rekursiven Unterteilung des Feldes beruhen, aber einen entgegengesetzten Weg wählen, wie man zu einem korrekt sortierten Feld kommt. Die eine Möglichkeit besteht darin, *vor* der Rekursion das Feld so zu verändern, dass am Ende alles richtig sortiert ist. Die andere Möglichkeit führt sofort die Rekursion durch und ordnet *am Ende* die beiden Hälften so um, dass das Gesamtfeld sortiert ist.

```
void sortiere(a[i0:i1]) {
    if (i0 < i1) {
       // Algorithmus1: passende Vorbereitung (Quicksort)
       sortiere(a[i0:mitte]);
       sortiere(a[mitte+1:i1]);
       // Algorithmus2: passende Nachbereitung (Mischen)
    }
}
```

Sortieren durch Mischen (*merge sort*), besteht von der Grundidee her darin, ein Feld in zwei Hälften zu zerlegen, die beiden Hälften getrennt zu sortieren (natürlich rekursiv nach demselben Verfahren) und die sortierten Hälften am Ende zusammen zu mischen. Mischen wird in Praktikum und Vorlesung behandelt.

Das hier besprochene Verfahren heißt *Quicksort*. Es beruht, darauf, dass man *vor* der rekursiven Feldzerlegung die Feldelemente so umordnet, dass nach dem Sortieren der beiden Bestandteile, das gesamte Feld richtig sortiert ist.

Dazu muss man das Feld so in zwei Hälften aufteilen, dass alle Elemente der unteren Teilhälfte kleiner oder gleich den Elementen der oberen Teilhälfte sind. Dies ist eine Aufgabe, die man (im Prinzip) in $O(n)$ lösen kann. Doch stellen wir das zuerst einmal zurück, und beginnen wir mit der Formulierung der Grundidee.

```
void sortiere(a[i0:i1]) {
    if (i0 < i1) {
        mitte = zerlege(a[i0,i1]);
        // a[i0:mitte-1] <= a[mitte] < a[mitte+1:i1]
        sortiere(a[i0:mitte-1]);
        // fertig_sortiert(a[i0:mitte])
        sortiere(a[mitte+1:i1]);
        // fertig_sortiert(a[i0:i1])
    }
}
```

In der Java-Form verwende ich wieder die üblichen Konventionen und auch englisch geschriebene Funktionsnamen. Im nächsten Listing sehen Sie eine erste Fassung des fertigen Java-Programms. Nur die Funktion `partition`, die den Teilalgorithmus *zerlege* realisiert, ist noch nicht angegeben. Aufgrund der Tatsache, dass der Quicksortalgorithmus jeweils auf nur einem Teil eines Feldes operiert, müssen wir hier auch eine entsprechende Funktion für das Sortieren eines Teilfeldes definieren. Es spricht natürlich nichts dagegen, auch diese Schnittstelle öffentlich zu machen.

```
public static void quickSort(Object[] a, Comparator comp) {
```

[4] Eine andere Überlegung geht davon aus, dass man ein Feld mit n-Elementen n mal um 1 verkleinern, aber nur $\log n$ mal halbieren kann.

```
    sort(a, 0, a.length-1, comp);
}

public static void quickSort(Object a[], int i0, int i1,
       Comparator comp) {
        // i0, i1 = Intervallanfang, -ende
    if (i0 < i1) {
        int mitte = partition(a, i0, i1, comp);
        // a[i0:i-1] <= a[mitte] < a[mitte+1:i1]
        quickSort(a, i0, mitte-1, comp);
        // fertig_sortiert(a[i0:mitte])
        quickSort(a, mitte+1, i1, comp);
        // fertig_sortiert(a[i0:i1])
    }
}
```

Vergleichen Sie diese Java-Formulierung mit dem Pseudocode! Sie sehen, dass sich nicht viel geändert hat.

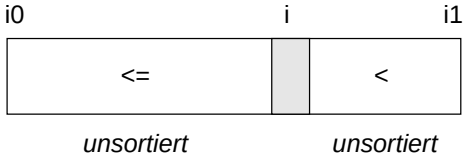

Abbildung 7.1: Hier ist dargestellt wie nach Ausführen der Funktion `partition` die Feldelemente so vertauscht sind, dass `a[i]` bereits den endgültigen Wert enthält und die Bereiche kleinerer und größerer Elemente trennt, so dass in dem weiteren Verfahren die beiden Teile unabhängig voneinander sortiert werden können.

Die Funktion `partition` enthält das, was wir noch nicht gelöst haben: die Zerlegung eines Feldes. Nach ihrem Aufruf hat sie das Feld `a` so verändert, dass die in dem letzten Beispiel angegebene Bedingung gilt. Sie sagt aus, dass `a[i]` endgültig festgelegt ist und dass alle links davon befindlichen Elemente auf jeden Fall links bleiben, ebenso wie die Elemente des rechten Feldabschnitts rechts von `a[i]` bleiben. In der Abbildung 7.1 ist diese Situation graphisch dargestellt.

Anhand der weiteren Kommentare ist klar, dass `quickSort`, vorausgesetzt es terminiert, das richtige Ergebnis liefert.

Für die Terminierung ist nur ausschlaggebend, dass bei erneuten rekursiven Aufrufen das jeweilige $i1 - i0$ mindestens um 1 kleiner ist als bei dem vorherigen Aufruf. Wie Sie sehen, ist dies gewährleistet, da – egal wie `partition` funktioniert – wenigstens ein Element, nämlich `a[i]`, aus der Rekursion herausfällt.

Die Terminierung lässt sich auch so zeigen, dass man sich klar macht, dass das Sortieren durch Auswahl ein ungünstiger Spezialfall des Quicksortalgorithmus ist. Diesen Spezialfall erhalten wir nämlich dann, wenn `partition` immer den Ergebniswert $i0$ hat. In dem Fall ist nämlich `a[i0]` nach dem Aufruf das Minimum von `a[i0:i1]`. Der erste rekursive Aufruf von `sort` erhält ein Teilfeld der Länge 0 (also ein leeres Feld) und der zweite Aufruf ein Feld der Länge $i1 - i0$, also um 1 weniger als der Feldabschnitt, der übergeben wurde.

Solange `partition` die oben angegebene Forderung erfüllt, ist die konkrete Ausgestaltung, nämlich welcher Wert zurückgegeben wird, für die Korrektheit unwichtig. Wie man sieht, ist die günstige Wahl dieses Wertes jedoch entscheidend für die Effizienz des Sortierverfahrens: Der gerade besprochene ungünstige Rückgabewert führt zu $O(n^2)$.

Das bestmögliche Resultat von `partition` ist $i = (i1 + i0)/2$, so dass das untersuchte Feldsegment in zwei gleich große Teile geteilt wird. Eine genauere Analyse ergibt für diesen Fall (wenn man voraussetzt, dass der Aufwand für `partition` proportional zu $i1 - i0 + 1$ ist), dass der Laufzeitbedarf von der Größenordnung $O(n \log n)$ ist.

Dieses optimale Verhalten ist aber nur dann effizient zu erhalten, wenn wir auf Anhieb wissen, welche Elemente von `a[i0:i1]` in die linke bzw. in die rechte Hälfte gehören. Dies kann z.B. so geschehen, dass wir eine Zahl auswählen und festlegen, dass alle Feldelemente, die kleiner oder gleich dieser Zahl sind, in die linke Hälfte gehören und die Feldelemente, die größer als diese Zahl sind, in die rechte Hälfte kommen. Der Aufwand, die optimale Zahl zu finden, ist jedoch so groß, dass sich dann der Algorithmus nicht mehr lohnt. Die Alternative besteht darin, dass wir einfach auf Gutdünken ein Feldelement auswählen und anhand dieses Elements (auch *Pivotelement* genannt) die Zuordnung zur linken und zur rechten Hälfte bestimmen.

In der Literatur findet man ganz unterschiedliche Möglichkeiten das Pivotelement auszuwählen. Eine ganz primitive Variante besteht darin, einfach das erste Element des Feldintervalls zum Pivot zu „ernennen“. Für den durchschnittlichen Fall ist diese Wahl so gut wie jede andere. Sie hat jedoch einen gravierenden Nachteil. Wenn nämlich das Feld bereits exakt oder annähernd sortiert sein sollte, führt diese Wahl zu dem schlechtesten Sortierverhalten, dem Worst-Case, in dem Quicksort nicht mehr schneller als die direkte Auswahl ist. Die Standardliteratur verwirft daher diese Lösung mit dem Hinweis, dass halbwegs fertig sortierte Felder eher die Regel als die Ausnahme sind.

In Lehrbüchern findet man am häufigsten die Wahl des mittleren Feldelements – bei einem bereits sortierten Feld ist das die beste Wahl. Ausführlichere Darstellungen zeigen, wie man einen besseren Schätzwert für den Median, *Pseudomedian* genannt, effizient berechnen kann.

Allen diesen Verfahren ist gemein, dass es einen ungünstigen Fall gibt, der das jeweilige Verfahren schlecht aussehen lässt. Grundsätzlich lässt sich bei Quicksort daran auch nichts ändern. Erstaunlicherweise gibt es jedoch ein ganz einfaches sehr robustes Verfahren, das hinsichtlich des Worst-Case Verhaltens (zumindest statistisch) allen anderen Verfahren überlegen ist. Nach diesem Verfahren wird das Pivot-Element einfach *zufällig* ausgewählt. Dieses Verfahren ist insofern sehr robust, als es grundsätzlich keine Feldanordnung geben kann, bei der es immer versagt. Das quadratische Worst-Case-Verhalten ist zwar nicht ausgeschlossen, bei großen Feldern statistisch jedoch sehr (in Wirklichkeit sehr, sehr, sehr, ...) unwahrscheinlich.

Da dieses Zufallsverfahren auch einfach zu programmieren ist, habe ich es hier verwendet. Wegen ihrer großen Robustheit finden Zufallsverfahren in den letzten Jahren zunehmend Anwendung.

```
private static void swap(Object[] a, int i, int j) {
    Object t = a[i]; a[i] = a[j]; a[j] = t;
}

private static int partition(Object[] a, int i0, int i1,
        Comparator comp) {
    int rdm = i0 + (int)(Math.random()*(i1-i0));
    swap(a, rdm, i0);
    int pivot = i0;
    i0++;
    // Invariante (I0, I1 = Anfangswerte):
    //   a[I0+1:i0-1]<=a[pivot]<a[i1+1:I1]
    while (i0 <= i1) {
```

```
        if(comp.compare(a[i0], a[pivot]) <= 0)
            i0++;
        else if (comp.compare(a[pivot], a[i1]) < 0)
            i1--;
        else {
                // a[i0] > a[pivot] >= a[i1]
            swap(a, i0, i1);
                // a[i0] <= a[pivot] < a[i1]
            i0++;
        }
    }
    // jetzt ist i0 = i1 + 1 und daher
    //   a[I0+1:i1] <= a[pivot] < a[i1+1:I1]
    // bringe pivot an die richtige Stelle
    if (i1 != pivot) swap(a, pivot, i1);
    return i1;
}
```

Bei dem dargestellten Verfahren wird also ein beliebiges Feldelement als Pivotelement festgelegt. Es wird anschließend an den Intervallanfang getauscht, damit es zunächst aus dem Weg ist und die weitere Zerlegung nicht stört. Am Ende des Algorithmus wird es dann an die richtige Stelle getauscht.

Die beiden Variablen `i0` und `i1` dienen als Laufvariable, die Kandidaten für eine Vertauschung aussuchen. Wenn `i0>i1` ist, ist der Aufteilungsvorgang beendet. Innerhalb der Schleife wird abgefragt, ob man `i0` oder `i1` erhöhen bzw. erniedrigen kann, oder ob man hat zwei Tauschpartner gefunden hat: In diesem Fall wird getauscht und `i0` wird erhöht (Sie können auch `i1` gleichzeitig erniedrigen. So bleibt das Verfahren aber etwas übersichtlicher). Schließlich wird am Ende das Pivotelement auf die Stelle `i1` getauscht. Damit hat man immer für wenigstens ein Feldelement den endgültigen Platz gefunden. Die Terminierung des Algorithmus ist leicht zu beweisen, da in jedem Durchlauf die Differenz der beiden Indizes vermindert wird.

Die in der Literatur angegebenen Varianten der `partition`-Funktion sind oft geringfügig optimiert. Häufig wird dabei das Verfahren unmittelbar in die `quickSort`-Funktion integriert. Dabei muss man nur etwas sorgfältiger auf die passende Variablenbenennung achten:

```
public static void quickSort(Object[] a, int i0, int i1,
        Comparator comp) {
    if (i0 < i1) {
        int rdm = i0 + (int)(Math.random()*(i1-i0));
        int t = a[rdm]; a[rdm] = a[i0]; a[i0] = t;
        int pivot = i0;
        int l = i0 + 1;
        int r = i1;
        while (l <= r) {
            if (comp.compare(a[l], a[pivot]) <= 0)
                l++;
            else if (comp.compare(a[pivot], a[r]) < 0)
                r--;
            else {
                temp = a[l]; a[l] = a[r]; a[r] = temp;
                l++; r--;
            }
        }
        if (r != pivot) {
            Object t = a[pivot]; a[pivot] = a[r]; a[r] = t;
        }
```

```
            quickSort(a, i0, r - 1, comp);
            quickSort(a, r + 1, i1, comp);
        }
    }

    public static void quickSort(Object[] a, Comparator comp) {
        quickSort(a, 0, a.length - 1, comp);
    }
```

Ich möchte hier aber ausdrücklich festhalten, dass solche Optimierungen letztlich kaum zu einer messbaren Laufzeitverbesserung führen, und keine wirkliche Optimierung des Verfahrens darstellen.

Das Zeitverhalten von Quicksort (egal in welcher Form) ist durch die folgenden Beziehungen gegeben:

$$\begin{aligned} T_{\text{best case}} &= O(n \log n) \\ T_{\text{average case}} &= O(n \log n) \\ T_{\text{worst case}} &= O(n^2) \end{aligned}$$

Das typische Merkmal an Quicksort ist, dass seine Leistungsfähigkeit nur statistisch – im durchschnittlichen Fall – erkennbar wird. Eine Analyse, die nur den ungünstigsten Fall betrachtet, würde ihn wahrscheinlich als relativ langsamen Algorithmus verwerfen, obwohl er in der Praxis fast unschlagbar ist.

Andere effiziente Sortierverfahren sind aber nicht besser als Quicksort, wenn Sie auf die üblichen Sortieraufaben in Feldern angewendet werden. Für andere Fälle, gibt es jedoch bessere Verfahren. Ein Beispiel ist das nächsten Abschnitt diskutierte ständige Einsortieren neuer Werte. Ein anderes Beispiel besteht darin, eine sehr Datenmenge zu sortieren, die so groß ist, dass der gesamte Datenbestand nicht gleichzeitig im Hauptspeicher Platz hat. Für diesen Zweck – das externe Sortieren – wurden ebenfalls eine Reihe von effizienten Verfahren entwickelt. Für alle Anwendungsbereiche hat man $O(n \log n)$-Algorithmen gefunden.

Mit dem Algorithmus *Proxmapsort* gibt es sogar ein superschnelles Verfahren, dessen Laufzeit im Mittel proportional zur Feldgröße ist. Dabei gilt allerdings die besondere Voraussetzung, dass den Feldinhalten einigermaßen gleichmäßig verteilte Zahlenwerte zugeordnet werden können.

7.3 Effiziente Verzeichnisse

Bei den bisherigen Beispielen von Feldern und Behälterklassen (z.B. verkettete Liste) stand im Vordergrund das einfache Speichern von Daten. In diesem und in dem folgenden Abschnitt will ich Ihnen Algorithmen vorstellen, die normalerweise beim Aufbau von Tabellen oder Verzeichnissen verwendet werden.

Ein Verzeichnis (englisch *dictionary* oder *map*) ist eine Datenstruktur, in der die Datenelemente unter einem *Schlüsselbegriff* abgelegt sind. Man kann ein Verzeichnis auch als eine Verallgemeinerung eines Feldes ansehen, in dem die Feldelemente nicht über ganzzahlige Feldindizes sondern auch über beliebige nichtnumerische Schlüsselbegriffe angesprochen werden können. Man nennt daher Verzeichnisse oft auch *assoziative Speicher* oder *assoziative Felder*.

Grundsätzlich können Schlüsselbegriffe beinahe jeden beliebigen Datentyp haben, der Vergleichsoperationen kennt. In den allermeisten Fällen, reicht es aus, wenn man sich

bei den Schlüsselbegriffen auf Zeichenketten vom Typ `String` beschränkt. Da dies vielleicht etwas konkreter und damit leichter zu verstehen ist, werde ich mich in den Beispielen zunächst darauf beschränken. Sie werden aber sehen, dass der Datentyp des Suchschlüssels in Java praktisch keinen Unterschied macht.

Neben dem Schlüsselbegriff ist in einer Tabelle immer auch ein Inhaltselement oder sogar eine komplexe Struktur von Inhalten gespeichert – dies sind die *Werte*, die unter dem Schlüssel abgelegt sind. Der Datentyp des Inhaltselements ist absolut nicht eingrenzbar.

Der objektorientierten Vorgehensweise folgend, soll hier eine abstrakte Schnittstellenbeschreibung für Tabellen in Form einer Schnittstelle angegeben werden. Wie üblich habe ich mich auch hier wieder an der Java-Bibliothek orientiert, ohne allerdings deren Vollständigkeit anzustreben.

```
/**
 * Schnittstelle fuer 'Dictionaries', d.h. ein Verzeichnis
 * von beliebigen Informationselementen, auf die ueber
 * einen Suchbegriff zugriffen wird.
 * Dictionary beschreibt eine Teilmenge der Schnittstelle
 * java.util.Map.
 */
public interface Dictionary {
    /**
     * Eintragen eines neuen Objekts.
     * @param key Suchschluessel.
     * @param value zu speicherndes Objekt.
     */
    public void put(String key, Object value);

    /**
     * Befindet sich 'key' im Dictionary?
     * @param key Suchschluessel.
     * @return true wenn Schluessel vorhanden.
     */
    public boolean includes(String key);

    /**
     * Suchen des unter 'key' gespeicherten Objekts.
     * @param key Suchschluessel.
     * @return gespeichertes Objekt oder null.
     */
    public Object get(String key);

    /**
     * Erlaubt die bequeme Ausgabe aller Elemente.
     * Fuer andere Operationen auf allen Elemente
     * braucht man einen Iterator.
     */
    public String toString();
}
```

Zu der Funktion `toString` ist eine Anmerkung angebracht. Zu jeder allgemein definierten Tabelle gehört die Möglichkeit, dass man bei Bedarf eine Funktion auf alle Elemente anwenden kann. Bei Feldern ist dies einfach über eine Zählschleife möglich. Es gibt aber keine sinnvolle Methode, mit der wir über die Erzeugung aller möglichen Schlüsselwerte alle Tabellenelemente erreichen können. Die beste Möglichkeit dies zu realisieren, ist die Verwendung (und Implementierung) eine Iterators.[5] Da dessen Implementierung aber aus-

[5] Am besten gleich zusammen mit der Implementierung von `Iterable`.

schließlich technische und keinerlei algorithmische Probleme enthält, habe ich hier keine solche allgemeine Schnittstelle definiert. (Es kann zu Übungszwecken aber gut sein, dies selbst zu versuchen.)

Andererseits ist die Implementierung von `toString` aber auch für sich schon sinnvoll. Außerdem kann ich damit zeigen, wie durch Inorder-Traversierung die Elemente eines Binärbaums in sortierter Reihenfolge aufgesucht werden.

7.3.1 Binärbäume als Datenstruktur für Tabellen

Bei dem Vergleich von Feldern und Listen hatten wir als Vorzug von Listen die bessere Möglichkeit der dynamischen Veränderung hervorgehoben. Aus den dabei angeführten Argumenten geht hervor, dass Tabellen, die sich in unvorhersehbarer Weise und auch häufig ändern, eigentlich besser durch dynamische Datenstrukturen, wie verkettete Listen, implementiert werden sollten.

In der Diskussion der Suchverfahren haben Sie gesehen, dass eine effiziente Suche nur möglich ist, wenn die Daten auf besondere Art und Weise – aufsteigend sortiert bei der binären Suche – angeordnet sind. Von daher liegt es nahe, als grundlegende Datenstruktur aufsteigend sortierte verkettete Listen zu verwenden.

Ehe wir anfangen so etwas zu programmieren, sollten wir uns aber zunächst Gedanken über das zu erwartende Laufzeitverhalten machen. Binäre Suche erfordert einen Aufwand von $O(\log n)$. Dabei muss immer wieder (also $\log n$-mal) auf verschiedene Feldplätze, die nicht unbedingt nahe beieinander liegen, zugegriffen werden. Diese wahlfreien Adressierungen stellen bei Feldern kein Problem dar, da ein solcher Zugriff in konstanter Zeit $O(1)$ möglich ist. Bei verketteten Listen ist das jedoch anders! Hier „kostet“ ein *wahlfreier* Zugriff auf ein beliebiges Listenelement einen Aufwand von $O(n)$!

Die Essenz dieser Überlegung ist ganz einfach: Verkettete Listen sind streng lineare Strukturen; daher sind auf verketteten Listen auch nur lineare Algorithmen sinnvoll zu implementieren.

Zusammenfassend können wir festhalten, dass es für die Programmierung von effizienten Suchverfahren einerseits günstig ist, dynamische Datenstrukturen zu verwenden (wegen der Veränderbarkeit), dass aber andererseits verkettete Listen nicht gut geeignet sind (wegen der Linearität). Der Ausweg lautet: Wir müssen nichtlineare dynamische Datenstrukturen verwenden! Die einfachste nichtlineare Struktur, die Sie kennengelernt haben, ist der Binärbaum. Und Binärbäume sind tatsächlich gut für die effiziente Speicherung von Verzeichnissen geeignet.

Die Grundidee eines Suchbaums ist in Abbildung 7.2 dargestellt. Die Grundstruktur ist auf einen Blick erkennbar: Der Baum ist insofern alphabetisch geordnet, als *der Schlüssel eines Elternknotens alphabetisch zwischen den Schlüsseln des linken und des rechten Teilbaums liegt.* Dieser letzte Satz ist genau die Klasseninvariante, die die Repräsentation unseres Baumverzeichnisses bestimmt.

Im Idealfall sind der linke und der rechte Teilbaum genau gleich groß. Beim Suchen im Binärbaum können wir an jedem Knoten entweder feststellen, dass wir den Suchbegriff gefunden haben, oder aber wir können bei diesem Idealfall die Menge der noch zu durchsuchenden Begriffe halbieren, indem wir je nach der alphabetischen Stellung des Suchwortes im linken oder im rechten Teilbaum weitersuchen. Das Suchverfahren ist praktisch eine binäre Suche, und seine Laufzeit ist daher von der Größenordnung $O(\log n)$.

Der Suchaufwand hängt in der Praxis davon ab, wie der Binärbaum organisiert ist, ob

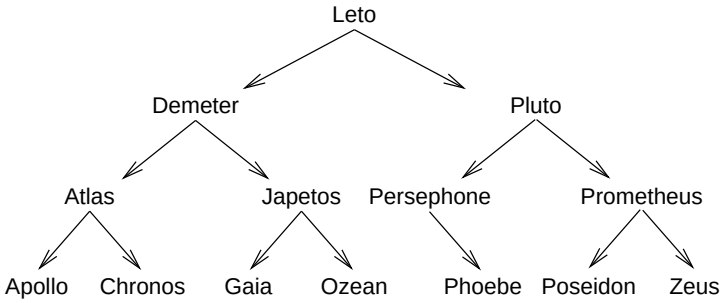

Abbildung 7.2: Ein Suchbaum für den griechischen Götterhimmel. Die besondere Eigenschaft eines Suchbaums ist, dass die Schlüsselbegriffe des linken Teilbaums sämtlich alphabetisch vor dem Knotenschlüssel liegen und die des rechten Teilbaums alle alphabetisch nach dem Knoten liegen.

nämlich der linke und der rechte Teilbaum tatsächlich gleich viele oder ob sie verschieden viele Knoten enthalten. Es gibt auch den Extremfall, dass der jeder Knoten nur ein Kind hat, also an jedem Knoten nur eine Verzweigung nach links oder nach rechts möglich ist. Ein solcher „Baum“ ist nichts anderes als eine lineare Liste – mit einem entsprechend schlechten Zeitverhalten.

Es ist möglich (und sinnvoll), die Baumalgorithmen so zu formulieren, dass der Suchbaum stets *ausgeglichen* ist. Suchverfahren auf ausgeglichenen Bäumen spielen eine große Rolle. Auch Klassen der Java-Bibliothek beruhen auf ausgeglichenen Bäumen. Da effiziente Verfahren zum Ausgleichen von Bäumen etwas komplizierter sind, beschränken wir uns hier auf die einfachen Verfahren, bei denen wir ohne besonderes Ausgleichen einfach „hoffen“, dass bei dem mehr oder weniger zufälligen Einsortieren von Namen halbwegs ausgeglichene Bäume entstehen.

Die Klasse `TreeNode` im letzten Kapitel war eine abstrakte Basisklasse, die keine wirklichen Knoteninhalte darstellte. Für das Suchen und Sortieren kommen wir jedoch an konkreten Inhalten nicht mehr vorbei: Wir könnten hier eine abgeleitete, konkrete Klasse verwenden, dass würde das Programm aber nicht wesentlich vereinfachen. Deshalb wird die Klasse hier von Grund auf neu programmiert.

Knoten des Binärbaums werden durch die geschachtelte Klasse `Node` beschrieben. Zunächst erweitert diese Klasse Baumknoten um die Felder `name` (für den Schlüsselbegriff) und `value` (für den Inhalt).

```
public class TreeDictionary implements Dictionary {

    private static class Node {
        Node(String key, Object value) {
            this.key = key;
            this.value = value;
        }
        Node left = null;
        Node right = null;
        String key;
        Object value;
    }
    private Node root = null;

    public TreeDictionary() {
    }
```

```
    public void put(String key, Object value) {
        root = enterNode(root, key, value);
    }

    public boolean includes(String key) {
        return findNode(root, key) != null;
    }

    public Object get(String key) {
        Node  p = search(root, key);
        return (p == null)? null: p.value;
    }

    public String toString() {
       StringBuilder builder = new StringBuilder("(");
       traverse(builder, root);
       return builder.append(")").toString();
    }
       ...
```

Der ganze Baum ist an der Instanzvariablen `root` aufgehängt. Wie Sie an dem Beispiel sehen, ist die Funktionalität durch statische Hilfsfunktionen realisiert.

Ausgehend von der Idee der binären Organisation eines Verzeichnisses ist auch die Implementierung der Funktionen ziemlich einfach. Obwohl normalerweise alle Baumfunktionen rekursiv definiert werden, ist es möglich, die Hilfsfunktion `findNode` iterativ zu formulieren. Dies liegt daran, dass nicht der gesamte Baum traversiert werden muss. Da ausschließlich abwärts gesucht wird, braucht man sich den Rückweg nicht in einem Rekursionsstack zu speichern.

```
private Node findNode(Node p, String key) {
    while (p != null && !key.equals(p.key)) {
        if (key.compareTo(p.key) < 0)
            p = p.left;
        else
            p = p.right;
    }
    return p;
}
```

Auch die Funktion `enterNode` (Implementierung von `put`) kann iterativ formuliert werden. Ich finde aber, dass die rekursive Lösung einfacher ist. Wie `findNode` ist sie nicht unmittelbar mit dem Baum-Objekt verknüpft, sondern geht von der als Parameter übergebenen Referenz eines Baum-Knotens aus.

```
private static Node enterNode(Node node, String key,
        Object value) {
    if (node == null)
        node = new Node(key, value);
    else {
        int cmp = key.compareTo(node.key);
        if (cmp  == 0)
            node.value = value;
        else if (cmp < 0)
            node.left = enterNode(node.left, key, value);
        else
            node.right = enterNode(node.right, key, value);
    }
```

```
    return node;
}
```

Es bleibt noch die Aufgabe die Methode `toString` und die von ihr aufgerufene Funktion `traverse` zu implementieren. Zunächst ist festzuhalten, dass ich dazu ein Objekt der Klasse `StringBuilder` erzeuge. Wie Sie wissen (sollten), dienen diese dazu nach und nach eine Zeichenkette aufzubauen.

Bei der Erzeugung der Stringdarstellung eines Binärbaums sollten Ihnen sofort die verschiedenen Möglichkeiten der Baumtraversierung einfallen. Die Frage ist nur, welche wir hier verwenden sollen. Die Antwort hängt natürlich davon ab, wie die Ausgabe aussehen soll. Grundsätzlich kann das Dictionary natürlich beliebig ausgeben. Wenn es ohne weiteres möglich ist – so wie hier –, wird man der besseren Lesbarkeit halber aber eine sortierte Ausgabe vorziehen. Dies wird durch die Inorder-Traversierung erreicht (wieso?).

Der Rest der Implementierung hat mit der einigermaßen ansprechend formatierten Darstellung zu tun.

```
private static void traverse(StringBuilder b,
        Node node) {
    if (node != null) {
        traverse(node.left, b);
        comma(b, node.left);
        b.append(node.key).append(":").append(node.value);
        comma(b, node.right);
        traverse(node.right, b);
    }
}

/**
 * Sorgt dafuer, dass nur da ein Komma gesetzt wird,
 * wo es hingehoert.
 */
private static void comma(StringBuilder b,
        Node neighbour) {
    if (neighbour != null) b.append(", ");
}
```

7.3.2 Hashtabellen

Angenommen, Sie bauen für die ca. 100 Mitarbeiter und Mitarbeiterinnen einer Firma ein Personalverzeichnis auf. Dabei verwenden Sie als Suchbegriff eine dreistellige Personalnummer. Es stellt sich die Frage, wie Sie das Mitarbeiterverzeichnis organisieren sollen, wenn die Zugriffszeit über die Personalnummer minimal sein soll.

Vielleicht denken Sie jetzt an eine besonders ausgefeilte Art von Binärbaum oder sonstiger sortierter Datenstruktur. Dabei ist die beste Antwort ganz einfach: Sie speichern die Personaldaten in einem genügend großen Feld (etwa mit 1000 Eintragungen) und verwenden als Feldindex für einen Mitarbeiter einfach seine Personalnummer. Der Aufwand auf bestimmte Personaldaten zuzugreifen ist bei diesem einfachen Verfahren garantiert $O(1)$.

Lassen Sie uns in Gedanken die Organisation des Mitarbeiterverzeichnisses etwas abwandeln. Da es bei Ihrem überschaubaren Mitarbeiterstamm keine zwei Mitarbeiter gleichen Namens gibt, beschließen Sie, die Personalnummern abzuschaffen und stets unmittelbar über den Mitarbeiternamen auf die Personaldaten zuzugreifen. Wie sieht jetzt die Organi-

sation des Mitarbeiterverzeichnisses aus und wie groß ist die mittlere Zugriffszeit?

Nach dem bisher Gelernten werden Sie darauf kommen, dass Sie bei geschickter Organisation ohne weiteres eine Zugriffszeit von $O(\log n)$, aber auch nicht mehr, erreichen können. Der Übergang von Personalnummern zu Namen scheint also zu einem deutlichen Effizienzverlust zu führen.

Das muss aber nicht so sein! Sie brauchen nämlich nur eine einfache mathematische Funktion zu definieren, die für jeden Mitarbeiter, basierend auf seinem Namen, eine eindeutige Zahl (etwa zwischen 0 und 999) liefert. Diese eindeutige Zahl verwenden Sie wieder als Index in der Mitarbeitertabelle – und erhalten wieder eine Zugriffszeit, die unabhängig von der Tabellengröße ist.

Definition:

*Eine Funktion, die aus einem Suchbegriff eine Zahl in einem vorgegebenen Bereich ermittelt, heißt **Hashfunktion**. Eine Hashfunktion, die für die vorkommenden Suchbegriffe eindeutig umkehrbar ist, heißt **perfekte Hashfunktion**. Ein Verzeichnis, auf das mittels einer Hashfunktion zugegriffen wird, heißt **Hashtabelle**. Hashverfahren basieren auf einem Array. Die (aktuelle) Arraygröße wird im folgenden mit N bezeichnet.*

Wenn die Menge der möglichen Suchbegriffe beschränkt ist, gibt es fertige Algorithmen und Bibliotheksfunktionen, die ein perfektes Hashing ermitteln. Bei dem stabilen Mitarbeiterstamm Ihrer Firma ist also so etwas möglich. Da Sie damit jedem Mitarbeiternamen in einer Zeit, die unabhängig von der Mitarbeiterzahl ist, eine eindeutige Zugriffsnummer zuordnen können, ist der Suchaufwand gleich $O(1)$.

Da wir in den hier besprochenen Verzeichnissen zunächst Zeichenketten als Suchbegriffe verwenden, müssen Hashfunktionen ihr Ergebnis aus den ASCII-Werten der einzelnen Zeichen ableiten. Dabei können Sie sich die wildesten funktionalen Zusammenhänge ausdenken. Man könnte zum Beispiel die folgende Funktion verwenden, die im wesentlichen die Buchstabenwerte aufsummiert. Um sicherzustellen, dass die Hashwerte in dem Bereich 0 bis $N-1$ liegen, bilde ich nach jeder Operation den Teilerrest.

```
private int hash(String key) {
    int n = key.length();
    int h = 0;
    for (int i=0; i<n; i++)
        h = (64*h + key.charAt(i)) % N;
    return h;
}
```

Als Beispiel habe ich einmal für einige Namen die Hashwerte bei einer Tabellengröße von $N = 11$ berechnet: Susanne(5), Theodor(10), Konrad(2) und Karl-Otto(0). Wie Sie sehen, ergibt sich eine vollkommen willkürliche, umkehrbare Zuordnung. Wenn dies jetzt Ihr gesamter Mitarbeiterstamm war, haben Sie eine perfekte Hashfunktion gefunden. Ein Problem taucht allerdings auf, wenn Sie Gisela(5) einstellen: Gisela hat dieselbe Hashzahl wie Susanne.

Man *kann* in Java so vorgehen. Allerdings ist es wohl sinnvoller sich an die verbreitete Vorgehensweise der Java-Bibliothek zu halten. Da Hashtabellen in Java-Anwendungen sehr häufig genutzt werden, hat man sich nämlich dazu entschlossen, eine Funktion `hashCode` in die allgemeine Objektschnittstelle aufzunehmen. Natürlich muss dann eventuell eine bestimmte Klasse, die Methode `hashCode` richtig überschreiben. Dabei

muss man sich nur an die Vorgabe halten, dass zwei Objekte, die gemäß `equals` als gleich gelten, auf die selbe Hashzahl abgebildet werden. Außerdem sollten nach Möglichkeit die Hashzahlen möglich gleichmäßig verteilt sein.

In der Klasse `String` ist `hashCode` jedenfalls schon definiert. Allerdings müssen, wir in der Anwendung beachten, dass `hashCode` eine beliebige ganze Zahl zurückgibt. Wir müssen sie also noch auf die Größe unseres Feldes zurecht stutzen. Das kann dann so aussehen:

```
int h = (key.hashCode() & 0x7FFFFFFF) % N;
```

Anmerkung:

Was bedeutet `& 0x7FFFFFFF`? Die Zahl `0x7FFFFFFF` steht für eine 32 bit Zahl. Die Hexadezimalziffer `F` steht für dezimal 15 oder für die Bitfolge `1111`; die Ziffer 7 steht für `0111`. Wenn Sie alles zusammenfügen, haben Sie eine 32 bit Zahl, bei der nur das linkeste Bit 0 ist. Die `&`-Verknüpfung bewirkt ein bitweises Und von allen Bits der beiden Zahlen. Im Ergebnis wird einfach das oberste Bit von `hashCode` zu 0 gesetzt. Damit wird effizient aus einer negativen Zahl eine positive Zahl gemacht. Aber Vorsicht! Sie dürfen dieses Verfahren nicht zur Berechnung des Betrags einer ganzen Zahl verwenden! Warum nicht, haben Sie sicher schon in Theoretische Informatik gelernt.

Natürlich ändert auch die Verwendung der Methode `hashCode` nichts an der Tatsache, dass unterschiedliche Zeichenketten die gleiche Zahl ergeben können.

Den Fall, dass zwei verschiedene Zeichenketten (oder allgemeiner *Schlüsselobjekte*), die gleiche Hashzahl ergeben, nennt man eine *Kollision*.

Definition:

*Eine **Kollision** zwischen zwei Suchschlüsseln liegt dann vor, wenn beiden dieselbe Hashzahl zugeordnet wurde. Eine **Kollisionsbehandlung** ist dazu da, auch im Fall von einer Kollision eine korrekte Arbeitsweise des Verzeichnisses zu gewährleisten.*

Wenn beim Eintragen eines neuen Namens eine Kollision auftritt, so bieten sich verschiedene Möglichkeiten an. Wenn sich Ihr Verzeichnis – so wie ein Mitarbeiterverzeichnis – nur langsam ändert, ermitteln Sie bei jeder Neueinstellung eine perfekte Hashfunktion. Wenn ihr Verzeichnis sich jedoch häufig ändert, verwenden Sie ein Verfahren zum Abspeichern der Tabellenelemente, mit dem Sie in der Lage sind, mit Namenskollisionen zurecht zu kommen, das also eine *Kollisionsbehandlung* enthält.

Ich will Ihnen hier zwei verschiedene Methoden der Kollisionsbehandlung vorstellen. Beide Verfahren kommen ohne Änderung der Hashfunktion aus.

Kollisionsbehandlung durch lineare Verschiebung

Im Idealfall ergibt der Suchbegriff die Hashzahl und diese wiederum den Platz, an dem die Daten zu speichern sind. Sobald eine Kollision vorliegt, wird der Algorithmus versuchen, zwei Namen an dem gleichen Platz zu speichern. Eine solche Kollision kann natürlich erst bei dem zweiten Eintrag festgestellt werden. Die Konsequenz der Kollision ist, dass der zweite Eintrag nicht an seinem idealen Platz abgespeichert werden kann. Es muss vielmehr nach einem neuen Verfahren ein neuer Platz gefunden werden. Im einfachsten Fall sucht man einfach den nächsten freien Platz, d.h. man erhöht die Hashzahl solange,

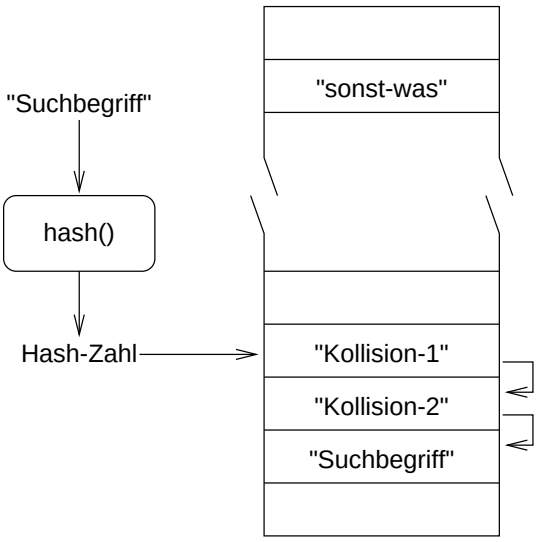

Abbildung 7.3: Bei der Kollisionsbehandlung nach der Methode der linearen Verschiebung wird zunächst mittels der Hashfunktion ein Feldindex errechnet. Findet sich dort nicht der Suchbegriff, wird der Index solange erhöht, bis entweder der Suchbegriff oder ein freier Platz gefunden wird. Im letzten Fall ist man sicher, dass der Suchbegriff nicht in dem Verzeichnis enthalten ist.

bis ein freier Feldplatz gefunden ist. Dieses Verfahren heißt *lineare Verschiebung* (*lineares Rehashing*).

Ein ähnliches Verfahren muss natürlich auch bei dem Aufsuchen der Inhalte ablaufen. In Abbildung 7.3 sind die Zusammenhänge bildlich dargestellt. Dort ist gezeigt, wie die Zeichenkette `"Suchbegriff"` erst nach zwei Versuchen den richtigen Platz gefunden hat.

Die einzelnen Funktionen sind nicht sehr kompliziert. Hier ist gleich die komplette Klasse angegeben.

```
/**
 * Implementiert die Dictionary-Schnittstelle durch eine
 * Hashtabelle mit linearer Verschiebung
 * bzgl. der Funktionsaufrufe s. Dictionary.java
 */

public class HashDictionary implements Dictionary {

    private static class Entry {
        Entry(String key, Object value) {
            this.key = key;
            this.value = value;
        }
        String key;
        Object value;
    }
    private int nEntries;
    private final int N;
    private Entry[] data;

    /* Konstruktoren */
    public HashDictionary() {
        this(73);
    }
```

```
    public HashDictionary(int arraySize) {
        this.N = arraySize;
        nEntries = 0;
        data = new Entry[N];
    }

    /* Funktionen der Schnittstelle Dictionary */

    public void put(String key, Object value) {
        if (!(nEntries  < N-1))
            throw new TableFullException();
        int h = (key.hashCode() & 0x7FFFFFFF) % N;
        while (data[h] != null && !key.equals(data[h].key))
            h = (h + 1) % N;
        if (data[h] == null) {
            data[h] = new Entry(key, value);
            nEntries++;
        }
        else
            data[h].value = value;
    }

    public boolean includes(String key) {
        return findEntry(key) != null;
    }

    public Object get(String key) {
        Entry p  = findEntry(key);
        return (p == null)? null: p.value;
    }

    public String toString() {
        StringBuilder b = new StringBuilder("(");
        boolean first = true;
        for (Entry e : data) if (e != null) {
            if (first) first = false; else b.append(", ");
            b.append(e.key).append(":").append(e.value);
        }
        return b.append(")").toString();
    }

    /* Hilfsfunktion. */

    private Entry findEntry(String key) {
        int h = (key.hashCode() & 0x7FFFFFFF) % N;
        while (data[h] != null && !key.equals(data[h].key))
            h = (h + 1) % N;
        return data[h] == null ? null : data[h];
    }
}
```

Die Beschränkung der Anzahl der Eintragungen durch die Funktion `put` auf $N-1$ bewirkt, dass mindestens ein Arrayplatz leer bleibt. Dadurch wird garantiert, dass die Suche stets abbricht. Gleichzeitig wird natürlich auch ein Überlauf verhindert.

Die Ausnahmeklasse `TableFullException` muss natürlich ebenfalls noch geschrieben werden. Die erheblich bessere Lösung ist aber, im Bedarfsfall das Datenfeld zu vergrößern und neu zu organisieren.

Bei den bisherigen Suchverfahren hing der Aufwand ausschließlich von der Anzahl der Eintragungen ab. Dies ist bei den Hashverfahren anders. Bei dem gerade beschriebenen

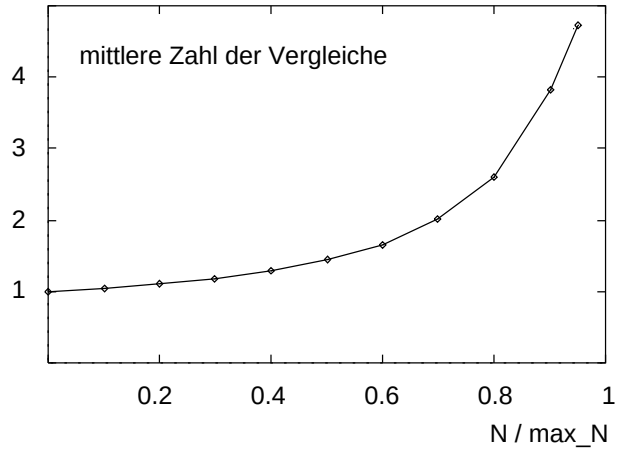

Abbildung 7.4: Die Kurve gibt an, wieviele Vergleiche zum Auffinden eines Inhalts nötig sind. Die unabhängige Variable ist der „Füllfaktor“, d.h. das Verhältnis der Anzahl der gespeicherten Inhalte zur Größe des Datenfeldes.

Verfahren mit linearer Verschiebung spielt die Größe der Hashtabelle die entscheidende Rolle bei der Bestimmung des durchschnittlichen Aufwands. Zusätzlich müssen wir berücksichtigen, dass die Zahl der Eintragungen durch die Tabellengröße begrenzt ist.

Im Idealfall ist der Hashalgorithmus (fast) unschlagbar. Beim Einfügen wird in $O(1)$ der richtige Tabellenplatz gewählt. Dasselbe gilt beim Suchen, wenn der gesuchte Eintrag vorhanden ist. Bei nicht vorhandenem Eintrag muss in jedem Fall überprüft werden, ob eine Kollision vorliegt. Im günstigsten Fall geht auch das in O(1), dann darf allerdings die Tabelle nur etwa bis zu Hälfte gefüllt sein. Bei gefüllter Tabelle ist ein Aufwand von $O(n)$ nötig, um festzustellen, dass ein Suchbegriff nicht vorhanden ist.

Der ungünstigste Fall tritt ein, wenn die Hashfunktion für alle einzutragenden Suchbegriffe die gleiche Zahl ergibt. Dann tritt jedes Mal eine Kollisionsbehandlung ein. Durch den linearen Charakter des Ausweichens auf andere Feldplätze ergibt sich ein Aufwand von $O(n)$, wenn n gleich der Zahl der Eintragungen ist.

Der durchschnittliche Suchaufwand ist natürlich am interessantesten. Es ist möglich, durch statistische Überlegungen eine Formel abzuleiten. Ich habe hier durch ein Beispielprogramm mit willkürlich erzeugten Hashschlüsseln die Kurve der Abbildung 7.4 erzeugt. Die entscheidende Aussage ist, dass nicht die Anzahl der Eintragungen über den Suchaufwand entscheidet, sondern ausschließlich der Füllfaktor, d.h. das Verhältnis der Zahl der Eintragungen zur Größe der Tabelle. Sie erkennen in der Abbildung, dass, solange die Tabelle höchstens zur Hälfte gefüllt ist, im Durchschnitt 1,5 Vergleiche nötig sind, um einen gesuchten Schlüssel zu finden, bzw. dass nur in jedem zweiten Fall eine Kollision auftritt. Sobald die Tabelle zu 90% oder mehr gefüllt ist, schnellt der Suchaufwand jedoch rapide in die Höhe.

Eine effiziente Tabellenverwaltung nach dem Hashverfahren setzt voraus, dass Sie in etwa wissen, wie viele Eintragungen zu erwarten sind. Bei günstig gewählter Größe erhalten Sie auch bei sehr vielen Eintragungen einen sehr schnellen Zugriff. Nachteilig ist nur Vergeudung von Speicherplatz bei nicht gefüllter Tabelle, die bei dynamischen Datenstrukturen, wie Listen und Bäumen, so nicht auftritt. Bei dem Vergleich sollten Sie jedoch auch beachten, dass dynamische Datenstrukturen durch ihre Zeigerfelder an anderer Stelle Speicher vergeuden.

Aufgrund der Kollisionsbehandlung ist das Löschen bei direkter Verschiebung nicht ganz

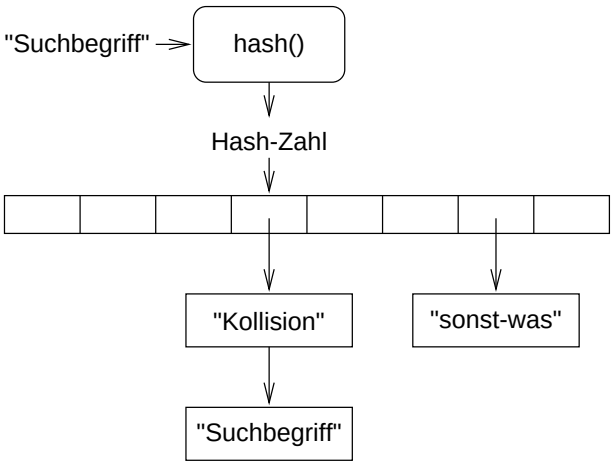

Abbildung 7.5: Das Prinzip der direkten Verkettung der Kollisionslisten. Die Hashfunktion bestimmt bei diesem Verfahren einen Feldindex, mit dem man die Startadresse einer linearen Liste findet, die den gesuchten Begriff enthält bzw. enthalten sollte.

einfach. Dieses Problem tritt bei der im nächsten Abschnitt besprochenen Variante nicht auf.

Kollisionsbehandlung durch direkte Verkettung

Man kann die Grundidee der linearen Verschiebung, nämlich nach der einmaligen Ermittlung der Hashzahl eine lineare Kollisionsbehandlung vorzunehmen, auch anders realisieren. Während bei der linearen Verschiebung die lineare Suche in einem Feld stattfindet, ist es genauso gut möglich, die Daten, die zum gleichen Schlüssel gehören, in einer linearen Liste zu speichern.

Bei der linearen Verschiebung wird der nächste mögliche Tabellenplatz durch das Erhöhen des Indexes gefunden. Umgekehrt sind bei dem *direkte Verkettung* genannten Verfahren alle miteinander kollidierenden Begriffe explizit miteinander verkettet. Die Abbildung 7.5 zeigt das Prinzip.

Bei der Implementation können Sie sich einer eventuell vorhandenen Listenklasse bedienen oder aber die wenigen erforderlichen Operationen neu implementieren. Schwierig ist dies nicht! Die Klassendeklaration unterscheidet sich (natürlich) nur im privaten Teil von der Methode der linearen Verschiebung. Wenn wir die Listenoperationen explizit programmieren, bestehen die einzigen Unterschiede in dem Verkettungsfeld `Entry next`.

```
private static class Entry {
    Entry(String key, Object value, Entry next) {
        this.key = key;
        this.value = value;
        this.next = next;
    }
    String key;
    Object value;
    Entry next;
}
private final int N;
private Entry[] data;
```

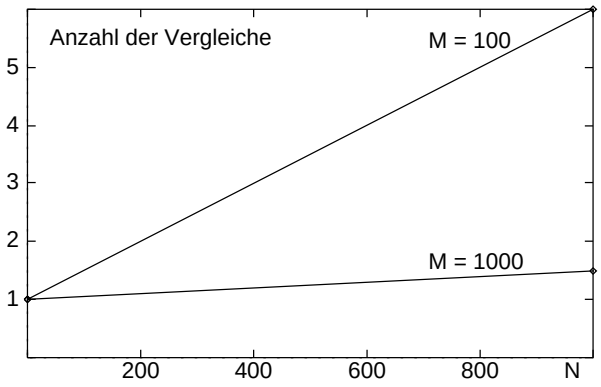

Abbildung 7.6: Beim Verfahren der direkten Verkettung ist der Suchaufwand bei einer Hashtabelle der Größe M und bei N Eintragungen ($N \gg M$) im Durchschnitt ungefähr gleich $1 + N/(2M)$. Wenn man durch automatische Anpassung der Tabellengröße, dafür sorgt, dass stets $N < M$ gilt, erhält man für die Zugriffskomplexität $O(1)$.

Die Funktionen sind alle ganz einfach. Ich gebe Ihnen hier daher nur den Konstruktor und die Funktion zum Eintragen eines neuen Wertes an.

```
public Bucket(int n) {
    N = n;
    data = new Entry[n];
}

public void put(String key, Object value) {
    // die richtige Liste suchen
    int h = (key.hashCode() & 0x7FFFFFFF) % N;
    // ist key verhanden ?
    Entry p = data[h];
    while (p != null && !key.equals(p.key)) p = p.next;
    if (p != null)
        // key da: Wert aendern
        p.value = value;
    else {
        // neues Feld anlegen
        data[h] = new Entry(key, value, data[h]);
    }
}
```

Die direkte Verkettung hat – genauso wie alle Verfahren, die auf dynamischen Datenstrukturen basieren – keine Probleme mit begrenztem Speicherplatz. Allerdings hängt die Laufzeiteffizienz auch hier von der Größe der Hashtabelle ab. Nach einer vernünftigen Faustregel wählt man die Tabellengröße wie bei der direkten Verkettung, so dass nur wenige Kollisionen vorkommen. Dies erreicht man etwa bei einem *Füllfaktor* ($=$ Eintragungen/Tabellengröße) von ca. 0.75. Professionelle Implementierungen vergrößern beim Erreichen dieser Grenze die Tabelle entsprechend. Aber selbst dann, wenn man den Algorithmus so einfach wie hier angegeben implementiert, hat man bei mäßigen Überschreitungen der Tabellengröße immer noch ein annehmbares Laufzeitverhalten.

Es ist leicht einzusehen, dass für eine ganz geringe Anzahl von Eintragungen – wie bei allen Hashverfahren – kaum mehr als ein Vergleich nötig ist. Ist dagegen die Zahl der Eintragungen N groß gegenüber der Größe M der Hashtabelle, die die Anfangsadressen der Kollisionsketten speichert, so sind im Schnitt N/M Vergleiche nötig. Für N gegen ∞

verhält sich der Algorithmus dann nicht besser als lineare Suchverfahren.

Merksatz:

Für eine Hashtabelle mit direkter Verkettung der festen Größe M ist bei einer Zahl von N Eintragungen der durchschnittliche Suchaufwand gleich $1 + N/(2M)$, also von der Größenordnung $O(N/M)$. Wenn man bei dem Überschreiten eines festen Verhältnisse von N/M (in der Regel bei ca. $3/4$) die Tabelle vergrößert, ist ein mittleres Laufzeitverhalten von $O(1)$ gewährleistet.

Abschließend ist noch festzustellen, dass die direkte Verkettung hinsichtlich des Speicherplatzbedarfs ungünstiger ist als die Methode der (linearen) Verschiebung. Dies liegt daran, dass mit dem dynamischen Speicher ein weiterer Overhead an Verwaltungsinformation für jede einzelne Listenzelle verbunden ist.

Die Methode der linearen Verkettung erlaubt auch das einfache Löschen:

```
public void remove(String key) {
    // die richtige Liste suchen
    int h = (key.hashCode() & 0x7FFFFFFF) % N;
    // ist key verhanden ?
    Entry p = data[h];
    Entry q = null;
    while (p != null && !key.equals(p.key)) {
        q = p;
        p = p.next;
    }
    if (p != null)
        // key da: Eintrag entfernen
        if (q == null)
            data[h] = p.next;
        else
            q.next = p.next;
    }
}
```

7.3.3 Dictionary-Klassen der Java-Klassenbibliothek

Die Java-Klassenbibliothek enthält in dem Paket `java.util` mehrere Klassen, die für eine effiziente Verwaltung von Verzeichnissen verwendet werden können.

Typisch für die Klassen der Java-Bibliothek ist, dass man beliebige Objekte als Suchschlüssel verwenden kann. Die einzige Voraussetzung ist, dass die Identität der Schlüsselobjekte nicht von „außen“ verändert werden, da man sonst keine Chance hat, die Inhalte wiederzufinden.

Die Identität ist durch die Methoden `equals` und `hashCode` bestimmt. Schlüsselobjekte müssen wiederauffindbar sein. Also darf sich die Identität nicht ändern.

Bei Objekte, deren Identität durch die Speicheradresse bestimmt ist, und die infolgedessen `equals` und `hashCode` von der Klasse `Object` übernehmen, erfüllen diese Voraussetzung.[6]

[6] Objekte, die durch die Java-Technik der schwachen Referenz (`WeakReference`) angesprochen werden, stellen ein Problem dar. Das soll aber hier nicht näher besprochen werden.

Wenn diese Methoden überschrieben wurden, darf sich ihr Ergebnis (während der Verwendung als Schlüssel) nicht ändern. Dies bedeutet, dass solche Schlüsselobjekte unveränderlich sein sollten. Bei Strings ist dies automatisch gegeben.

Die wichtigsten Interfaces und Klassen sind wie folgt:

- Die Schnittstelle `Map` definiert die möglichen Operationen aller Verzeichnisklassen. Die abgeleitete Schnittstelle `SortedMap` garantiert, dass über die Daten sortiert iteriert wird.
- Die Klasse `TreeMap` implementiert die Realisierung von Verzeichnissen durch Suchbäume. Sie implementiert die Schnittstelle `SortedMap`
- Die Klasse `HashMap` implementiert den Algorithmus der direkten Verkettung. Die Tabelle wird automatisch so angepasst, dass stets eine effiziente Suche gewährleistet ist.
- Die Schnittstellen `Set` und `SortedSet` definieren das Speichern von Mengen, d.h. von Datenstrukturen, in denen jedes Element höchstens einmal enthalten ist. Während eine Menge keine Reihenfolge festlegt, wird mit `SortedSet` eine sortierte Reihenfolge garantiert.
- die Klasse `HashSet` basiert auf `HashMap` ebenso wie `TreeSet` auf `TreeMap` aufbaut.

Die günstigste Klasse zur Realisierung von Verzeichnissen ist die Klasse `HashMap` (die Klasse `Hashtable` ist eine ältere Variante, die etwas ineffizienter ist und auch nur etwas eingeschränkter zu nutzen ist). Wenn es allerdings, darauf ankommt, dass man leicht eine nach den Suchschlüsseln sortierte Ausgabe erzeugen kann, empfiehlt sich die Verwendung der Klasse `TreeMap`.

Zum Traversieren aller gespeicherten Elemente enthalten die Map-Klassen geeignete Iterator-Schnittstellen:

- `keySet().iterator()` liefert einen Iterator über alle Schlüssel (der Iterator läuft bei einer `SortedMap`-Klasse in Sortierreihenfolge).
- `entrySet().iterator()` liefert einen Iterator über alle Inhalte (bei den `SortedMap`-Klassen gilt die sortierte Reihenfolge der Schlüssel).
- `values().iterator()` liefert einen Iterator über alle gespeicherten Werte.
- `iterator()` liefert einen Iterator über alle Elemente einer Menge (bei den `SortedSet`-Klassen gilt die sortierte Reihenfolge).

Mit der For-Each-Schleife sieht die Iteration wieder besonders einfach aus:

```
Map<String, Typ> map = new HashMap<>();
...
for (String key : map.keySet()) {
    ... // Anweisungen mit key
}
```

Zum Abschluss des Skripts ist wurde hier eine Eigenschaft von Java 7 ausgenutzt. Die dort ansatzweise implementierte Typ-Herleitung (type inference) erlaubt es, beim Konstruktoraufruf die Typparameter wegzulassen, da diese ja schon durch den Typ der Variablen `map` bestimmt sind.

Anhang A

Erstellen einer JNI-Implementierung

Hier soll kurz der Aufruf einfache C-Funktionen dargestellt werden. Mit *einfach* ist hier gemeint, dass dabei keinerlei Klasseneigenschaften genutzt werden. Die C-Funktion erhält maximal Werte der elementaren Datentypen und gibt einen solchen Wert zurück.

Auf der Java-Seite braucht man für jede C-Funktion eine Native-Dekalaration, mir der der Name und die Signatur bekannt gemacht werden. In dem Fall des Aufrufs einfacher Funktionen, handelt es sich um eine statische Klassenfunktion.

```
public class CPUTime implements IClock {
    public static native double clock();

    static {
        String path =
            System.getProperty("java.library.path");
        System.out.println(path);
        System.loadLibrary("clock");
    }
}
```

Im nächsten Schritt übersetzen wir die Java-Datei und erzeugen aus dem Classfile die Headerdatei CPUTime.h.

```
javac CPUTime.java
javah CPUTime
```

An dieser Headerdatei fällt zunächst auf, dass die Datei `jmi.h` inkludiert wird. Am interessantesten ist die Deklaration der Funktion `Java_CPUTime_clock`. Durch den Vorsatz `Java_CPUTime` erreicht man, dass der Name innerhalb der C-Welt eindeutig wird.

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class CPUTime */

#ifndef _Included_CPUTime
#define _Included_CPUTime
#ifdef __cplusplus
extern "C" {
#endif
/*
```

```
 * Class:     CPUTime
 * Method:    clock
 * Signature: ()D
 */
JNIEXPORT jdouble JNICALL Java_CPUTime_clock
  (JNIEnv *, jclass);

#ifdef __cplusplus
}
#endif
#endif
```

Wie Sie wissen, gehört zu jeder C-Bibliothek neben der Headerdatei (die haben wir schon) auch eine Implementierungsdatei. Diese müssen wir jetzt per Hand anlegen. Dabei können wir natürlich den vorgefertigten Funktionskopf aus `CPUTime.h` übernehmen. Dabei muss man aber daran denken, für die beiden Parameter Variablen einzuführen! Diese beiden Parameter werden hier nicht benötigt. Sie stellen Referenzen für das Java-Laufzeitsystem und zu der Klasse `CPUTime` dar. Letztere bräuchten wir, wenn wir auf Klassenvariablen zugreifen wollten oder Methoden aufrufen wollten.

Für unsere einfache Anwendung ist nur einigermaßen nützlich zu wissen, dass in der Regel `jdouble` nur ein anderer Name für `double` ist. Ich habe die C-Datei natürlich `CPUTime.c` genannt.

```
#include <time.h>
#include "CPUTime.h"

JNIEXPORT jdouble JNICALL Java_CPUTime_clock(
    JNIEnv *env , jclass c)
{
    return (double) clock() / (double) CLOCKS_PER_SEC;
}
```

Nachdem wir das haben, ist der einfache, relativ systemunabhängige Teil unserer Aufgabe leider zu Ende. Wir müssen schließlich jetzt eine shared library bzw. ein dynamic link library erstellen. Schon diese Namensunterschiede verraten, dass diese Aufgabe auf jedem System anders gelöst wird.

Hier ist zunächst ein Makefile, der auf jedem Unix-System (zumindest unter Linux und unter AIX), dass über den GNU-Compiler verfügt laufen sollte.

```
# Create shared library for Java
# JAVA_HOME muss entsprechend angepasst werden!
JAVA_HOME = /usr/java130
INCS =  -I$(JAVA_HOME)/include -I$(JAVA_HOME)/include/linux

SOURCE = CPUTime.c
LIBRARY = libclock.so

$(LIBRARY): $(SOURCE)
        g++ $(INCS) -shared $(SOURCE) -o $(LIBRARY)
```

Nach diesen Vorbereitungen, laufen meine Beispiele auf der advm1 sofort. Unter meinem Linux-System musste ich aber noch den Suchpfad fuer shared libraries definieren:

```
export LD_LIBRARY_PATH=.
```

Alternativ kann man die erzeugte `.so`-Datei in ein Verzeichnis legen, in dem standardmaessig nach Bibliotheken gesucht wird.

Bei anderen Betriebssystemen oder Compilern aendert sich *nur* die Compileranweisungen. Als Beispiel sei hier die Erzeugung eine DLL (dynamic link library) mit dem MinGW-Compilers (MinGW = „Minimalist's GNU for Windows“ ist Bestandteil der Entwicklungsumgebung Dev-Cpp) angegeben:

```
# Create dll for Java
# JAVA_HOME muss entsprechend angepasst werden!
JAVA_HOME = c:/j2sdk1.4.2
INCS =  -I$(JAVA_HOME)/include -I$(JAVA_HOME)/include/win32
CFLAGS = -Wl --kill-at

SOURCE = CPUTime.c
LIBRARY = clock.dll

$(LIBRARY): $(SOURCE)
    g++ $(CFLAGS) $(INCS) -shared $(SOURCE) -o $(LIBRARY)
```

Anhang B

Glossar

abstract: Das Schlüsselwort `abstract` bezeichnet *abstrakte Methoden* und *abstrakte Klassen*.

Abstrakte Methode: Bei einer abstrakten Methode handelt es sich um eine reine Schnittstellendeklaration, zu der keine Implementierung in der Klasse gehört. Abstrakte Methoden dürfen nur in Interfaces und in abstrakten Klassen stehen.

Abstrakte Klasse: Eine Klasse, die als `abstract` deklariert ist, ist eine abstrakte Klasse. Abstrakte Klassen dienen dazu, das gemeinsame Verhalten mehrerer abgeleiteter Klassen zu beschreiben. Von einer abstrakten Klasse können keine Instanzen gebildet werden. Dafür braucht die abstrakte Klasse nicht alle Methoden ihrer Schnittstelle zu implementieren. Die typische Verwendung abstrakter Klassen besteht darin, ihren Unterklassen gemeinsame Implementierungen Methoden zu bieten. Die Unterklassen müssen sich dann nur noch um ihre jeweiligen Besonderheiten kümmern.

Adresse: Da während der Programmlaufzeit jedes Programmobjekt auf einer festen Stelle im Hauptspeicher des Rechners abgelegt ist, werden Datenobjekte rechnerintern stets über Speicheradressen angesprochen. Speicheradressen gehören zu den Implementierungseigenschaften einer Programmiersprache. Auf der Ebene einer anwendungsbezogenen Sprache wie Java haben Adressen keinen Platz. In der Sprache C ist das aber anders, da C ja bewusst die Verbindung zur Computerhardware abbildet um systemnahe Programmierung zu ermöglichen.

Algorithmus: Genaue Vorschrift wie ein Ergebnis in mehreren aufeinanderfolgenden, genau bestimmten Einzelschritten erreicht werden kann.

Alternative: Anweisungen für Alternativen dienen dazu, Programmverzweigungen in Abhängigkeit von logischen Bedingungen auszudrücken. In Java werden Alternativen durch If-Anweisungen und Switch-Anweisungen ausgedrückt.

Anonyme Klasse: Eine *anonyme Klasse* ist eine Klasse ohne Namen. Anonyme Klassen sind *innere Klassen* (in einer anderen Klasse oder in einer Methode). Ihr einziger Zweck besteht darin, ein Objekt zu erzeugen und dabei eine Methode einer Schnittstelle zu implementieren oder eine Methode einer Oberklasse zu überschreiben. Anonoyme Klassen werden in Java oft zur Definition von Funktionsobjekten verwendet. Funktionsobjekte sind in vielen Anwendungen (z.B. GUI-Programmierung, Bibliotheksklassen) nötig, werden von Java aber nicht direkt unterstützt.

Attribut: Eigenschaft eines Objekts. Häufig wird der Begriff synonym zu *Instanzvariable* verwendet. Es ist aber hervorzuheben, dass mit Attributen, stets von außen sichtbare Eigenschaften gemeint sind.

Ausdruck: Ein Ausdruck ist die Verknüpfung von Werten oder Objekten. Das Ergebnis eines Ausdrucks ist ein Wert oder ein Objekt. Der Typ eines Ausdrucks schränkt die Menge der möglichen Ergebnisse ein. Er wird durch den Compiler ermittelt und hängt von den Typeigenschaften der im Ausdruck verwendeten Operationen und Operanden ab. Der Compiler überprüft, ob die Operationen eines Ausdrucks mit den Typregeln verträglich sind. Die Typangabe mittels Cast kann erst zur Laufzeit geprüft werden.[1]

Automatische Speicherwaltung: Alle Programmobjekte benötigen Speicherplatz. Die Aufgabe diesen Speicherplatz zu reservieren und gegebenenfalls wieder freizugeben, obliegt in einer höheren Programmiersprache dem Compiler und dem Laufzeitsystem. In der Regel werden dazu abgestufte Optimierungen eingesetzt. Für statische Objekte kann Speicherplatz beim Programmstart fest reserviert werden. Der Speicherbedarf für lokale Variable einer Methode wird beim Methodenaufruf garantiert und kann nach Verlassen der Methode wieder freigegeben werden (Laufzeitstack). Daneben gibt es den Speicherbedarf für die mit `new` erzeugten Referenzobjekte (*dynamischer Speicher*). Der Speicherplatz eines Referenzobjekts kann dann neu verwendet werden, wenn das Programm keine gültige Referenz auf dieses Objekt mehr enthält (*garbage collection*). Ältere Programmiersprachen (Pascal, C, C++) verlangen vom Programmierer die explizite Verwaltung des dynamischen Speichers. Damit können zwar Laufzeitvorteile erzielt werden, diese werden aber mit einer sehr großen Fehleranfälligkeit erkauft.

Baumstruktur: Unter einem Baum versteht man einen Graphen, bei dem jeder Knoten ausgehend vom dem ausgezeichneten Wurzelknoten über genau einen Weg erreicht werden kann.

Binden: Unter Binden versteht man im Allgemeinen die Zuordnung eines Namens oder Ausdrucks zu einem Programmobjekt (Speicherplatz, Funktion, Methode). Im engeren Sinne meint man damit, ohne dies besonders zu betonen, die Zuordnung von Funktionsaufruf zu aufgerufener Funktion. Die einzelnen Aktionen des Bindens eines Funktions- oder Methodenaufrufs erfolgen an verschiedenen Stellen des Übersetzungsvorgangs. Der Compiler stellt zunächst fest, zu welchem Schnittstellentype die Funktion gehört und ob der Aufruf und die Verwendung des Ergebnisse mit den Typregeln verträglich ist. Die eigentliche Bindung, nämlich die Vergabe übereinstimmender Adressen in Aufruf und in der Funktion, führt in C der Linker durch (*frühe Bindung*). In objektorientierten Sprachen kann die Bindung sehr oft erst zur Laufzeit vorgenommen werden, weil erst dann das Empfängerobjekt, seine Klasse und seine Methoden bekannt sind (*späte Bindung*). Nur in den Fällen, in denen die implementierende Methode schon zur Übersetzungszeit feststeht (Klassenfunktionen, Konstruktoren, private-Methoden und Aufrufe von Methoden der Oberklasse), kann in Java die Optimierung durch frühen Bindung eingesetzt werden.

Cast: Cast wird meist mit Typkonvertierung oder mit Typanpassung übersetzt. Das ist missverständlich. Es muss zwischen dem Cast bei numerischen Wertdatentypen

[1] Ein weiterer Sonderfall ist die falsche Zuweisung zu einem Arrayelement, die zu einer `ArrayStoreException` führt.

(siehe *Typkonvertierung*) und bei Referenzdatentypen (siehe *Typangabe*) unterschieden werden. Für den nichtnumerischen Wertdatentyp `boolean` gibt es keinen Cast. In der Programmiersprache C ist die Situation anders! Während dort die Wirkung bei numerischen Werten wie in Java ist, besteht die Wirkung bei Zeigern darin, die Typprüfung aufzuheben (*unsicherer Cast*).

class: Das Schlüsselwort `class` leitet die Beschreibung einer *Klasse* ein.

Datei: Eine Datei ist eine unter ihrem Namen auf einem Speichermedium abgelegte (sequentielle) Datenstruktur.

Datenelement: Im weitesten Sinn sind mit diesem Begriff beliebige Programmvariable gemeint. Im engeren Sinn bezieht sich der Begriff auf die Instanzvariablen einer Klasse.

Datentyp: Siehe: *Typ*.

Design by Contract: Unter Design by Contract versteht man eine Programmiermethode, bei der grundsätzlich für jede Methode und Funktion eine mehr oder weniger vollständige formale Spezifikation angegeben wird. Dies geschieht durch die Formulierung von *Vorbedingungen*, die beim Aufruf der Funktion erfüllt sein müssen und von *Nachbedingungen*, die von der Funktion garantiert werden. Die Regeln von Design by Contract verlangen, dass Vorbedingungen durch Vererbung nicht verstärkt werden dürfen, und dass Nachbedingungen nicht abgeschwächt werden dürfen. Zwar ist die Technik der formalen Überprüfung der Korrektheit eines Programms anhand dieser Bedingungen bisher nicht praxistauglich, dafür gibt es aber (auch für Java) sehr gut verwendbare Testwerkzeuge zur Unterstützung von Design by Contract. Vielleicht kommt der größte Gewinn aber auch aus der verbesserten Programmiermethodik und einer guten Dokumentation.

Destruktor: Destruktoren unterstützen den Programmierer bei der expliziten Speicherverwaltung. Da in Java der Speicher automatisch verwaltet wird, gibt es dort auch keine Destruktoren. Die Methode `finalize` kann in wenigen Einzelfällen sinnvoll sein um auch in Java implementierungsnahe „Aufräumarbeiten“ durchzuführen. Allerdings ist zu beachten, dass man in Java nicht wissen kann, ob und wann diese Methdode aufgerufen wird.

Dokumentation: Dokumentation beschreibt das Verhalten eines Programmteils in lesbarer Form. Java unterstützt die Dokumentation von Klassen und von Operationen durch das Werkzeug *javadoc*. In der Regel sollten auch Klassen- und Instanzvariable ausreichend dokumentiert werden. Der Dokumentationswert eines Programms wird erheblich durch aussagekräftige Namen von Variablen, Klassen und Operationen unterstützt. Es ist oft besser, komplexe Teile einer Methode in eine sinnvoll benannte Methode auszulagern als diese Teile *innerhalb* der Methode zu dokumentieren.

Dynamischer Speicher: Die Speicherbereiche, die explizit durch die New-Anweisung dem Programm zugeordnet werden, heißen dynamischer Speicher.

Exemplar: siehe *Instanz*

explizit: Der deutsche Begriff *explizit* bedeutet *ausdrücklich*. Er wird bei der Programmierung häufig so gebraucht, dass man eine bestimmte Operation ausdrücklich angeben muss (oder kann) und dass diese Opereration eben nicht automatisch oder *implizit* ausgeführt wird.

extends: Das Java-Schlüsselwort `extends` taucht im Kopf einer Interface-Deklaration oder einer Klassendeklaration auf. Bei einem Interface bedeutet es, dass die neue Schnittstelle eine oder mehrere andere vorhandene Schnittstellen erweitert. Eine Klassendeklaration bezieht sich mittels `extends` immer auf genau eine Oberklasse. Die Klasse übernimmt damit die Schnittstelle und die Implementierung der Oberklasse. Methoden gleicher Signatur überschreiben die entsprechenden Methoden der Oberklasse. Wenn keine `extends`-Klausel im Klassenkopf vorkommt, bedeutet dies implizit `extends Object`.

Feld: Mit einem Feld ist im Deutschen oft die sequentielle Datenstruktur gemeint, die durch die Speicherung der Datenelementen in aufeinanderfolgenden Speicherzellen ausgedrückt wird (auch *Array* genannt). Vorsicht: In der Java-Sprachbeschreibung wird mit *field* die Komponente einer Klasse oder eines Objekts bezeichnet.

final: Das Schlüsselwort `final` (zu deutsch *endgültig*) wird in Java in verschiedenen Bedeutungen gebraucht. Variablen die als `static final` deklariert sind, bezeichnen Konstanten. `final` ohne den Zusatz `static` steht für unveränderliche Instanzvariable oder unveränderliche lokale Variable. Sie können nur durch die Initialisierungsanweisung (evtl. im Konstruktor) einen Wert erhalten, der dann nicht mehr verändert werden kann. Bei Klassen bedeutet der Zusatz `final`, dass keine Klasse von dieser Klasse abgeleitet werden darf. Bei einer Methode bedeutet `final`, dass die Methode in einer abgeleiteten Klasse nicht überschrieben werden darf.

Frühe Bindung: Damit bezeichnet man die vor der Programmausführung erfolgte Bindung des Funktionsaufrufs an den Funktionscode. In Java erfolgt die frühe Bindung zum Zeitpunkt des Ladens des Classfiles – im Unterschied zur späten Bindung, die erst beim Methodenaufruf erfolgt. Siehe auch *Binden*.

Funktion: Im engeren, mathematischen Sinn bezeichnet eine Funktion die Abbildung einer Liste von Argumenten zu einem Ergebniswert. In diesem engen Sinn hängt das Ergebnis nur von den Argumentwerten ab (in der Informatik wird eine solche Funktion auch *pure function* genannt). In Programmiersprachen, insbesondere in C, bezeichnet man mit Funktion aber auch alle formal als Funktion, d.h. mit Parameter und Rückgabe definierten Einheiten. In Java entspricht den C-Funktionen die *statische Klassenfunktion*.

Graph: Unter einem Graphen versteht man eine Menge von *Knoten*, die durch *Kanten* miteinander verbunden sind.

Hashfunktion: Eine Hashfunktion ordnet einem Suchbegriff (meist eine Zeichenkette) eine Zahl aus einem vorgegebenen Bereich zu. Gute Hashfunktionen zeichnen sich durch eine Gleichverteilung der Ergebniswerte aus. Java sieht vor, dass jede Klasse eine geeignete Methode namens `hashCode` zur Verfügung stellt.

Hashtabelle: Eine Hashtabelle realisiert einen inhaltsadressierten Speicher, der aus dem Wert der Hashfunktion des Schlüsselbegriffs, einen Arrayindex für das Speichern der Daten ermittelt. Wenn unterschiedlichen Schlüsseln die gleiche Hashzahl zugeordnet ist, muss eine *Kollisionsbehandlung* entscheiden, wo die Datenelemente zu speichern oder zu finden sind.

Identität: Jedes Objekt hat eine eigne Identität, die sich in einem Programm durch die eindeutige Objektreferenz ausdrückt. Eine Sonderrolle spielen unveränderliche

Wertobjekte, bei denen bereits der Wert für die Indentität ausreicht. In Java wird die logische Identität durch die Funktion `equals` festgestellt.

implements: Das Schlüsselword `implements` kann im Kopf einer Klassendeklaration stehen. Es ist gefolgt von einer durch Komma getrennten Liste von Schnittstellennamen. Es bedeutet, dass die Klasse alle in den Schnittstellen aufgeführten Operationen implementiert (Ausnahme: abstrakte Klasse). Damit wird gleichzeitig erreicht, dass die Klasse mit den Schnittstellentyp verträglich ist.

implizit: Der deutsche Begriff *impizit* bedeutet *stillschweigend*. Das Gegenteil ist *explizit* (siehe dieses).

Import-Anweisung: Die Import-Anweisung legt fest, dass ein einzelner oder alle Klassennamen eines Pakets ohne Angabe der Paketzugehörigkeit verwendet werden können. Die Variante des Static-Import erlaubt es die Abkürzung so weit zu fassen, dass statische Klassenelemente ohne Angabe des Klassennamens angesprochen werden können. Die Import-Anweisung hat *nichts* mit der Include-Direktive von C zu tun!

Include-Direktive: `include` ist eine Direktive des Präprozessors von C mit der eine komplette Quelltextdatei temporär für die Übersetzung in eine umfassende Datei hineinkopiert wird.

Initialisierung: Unter Initialiserung versteht man die Festlegung des Anfangswertes einer Variablen oder eines Objekts. Im Unterschied dazu, dient eine Zuweisung dazu, Werte zu verändern. Die Initialisierung geschieht bei der Variablendefinition oder im Konstruktor.

instanceof: Der boolesche Ausdruck *Objektreferenz* `instanceof` *Typname* stellt zur Laufzeit fest, ob ein gegebenes Objekt mit der angegebenen Klasse oder Schnittstelle typverträglich ist.

Instanz: Das Wort Instanz kommt vom dem lateinischen *instans* = *gegenwärtig*. Es bedeutet in der Informatik die konkrete Ausgestaltung eines allgemeineren Begriffs, der in verschiedenen Formen präzisiert werden kann. Im Zusammenhang mit Klassen versteht man unter einer *Klasseninstanz* ein aus einer Klasse gebildetes Objekt. Bei einer aktiven Funktionsausführung, bei der konkrete Werte für die lokalen Variablen existieren, spricht man auch von einer *Instanz der Funktion*. Häufig wird im Deutschen anstelle von Instanz der Begriff *Exemplar* verwendet.

Instanzvariable: Instanzvariable sind die Datenelemente eines Objekts, d.h. der Instanz einer Klasse. Instanzvariable beschreiben den Zustand des Objekts. Instanzvariable werden häufig mit den *Attributen* eines Objekts identifiziert. Attribute letztlich immer auf Instanzvariable abgebildet sein; umgekehrt stellen aber nicht alle Instanzvariablen Attribute dar.

Interface: Ein Interface beschreibt eine Schnittstelle, d.h. es deklariert eine Menge von abstrakten Operationen. Zusätzlich können auch Konstanten deklariert werden. Ein Interface definiert einen statischen Typ. Interface-Einheiten können mit `extends` erklären, dass sie die Operationen und Konstanten einer anderen Schnittstelle übernehmen. Es kann festgelegt werden, dass eine Klasse mit dem Interface-Typ verträglich ist, indem man den Namen der Interface-Einheit in der Implements-Klausel aufführt und alle Operationen der Schnittstelle durch Methoden implementiert.

Iteration: Die Formulierung von Wiederholung durch Schleifenanweisungen heißt Iteration. Die wichtigsten Iterationsanweisungen von Java sind die While-Anweisung und die For-Anweisung. For-Anweisungen werden meist da verwendet, wo eine festgelegte Anzahl von Daten bearbeitet werden muss.

Klasse: Eine Klasse beschreibt das Verhalten und die Struktur einer Menge gleichartiger Objekte. Eine Klasse definiert den statischen Typ der Objektschnittstelle, legt das konkrete Verhalten seiner Objekte fest und initialisiert neue Objekte. Es ist wichtig, dass die Schnittstelle einer Klasse möglichst allgemein gehalten ist und nicht von der internen Realisierung abhängt (Geheimnisprinzip).

Klassenfunktion: Klassenfunktionen, oder *statische Funktionen* sind nicht an ein Objekt gebunden und verfügen daher auch nicht über eine `this`-Referenz. Sie werden dazu verwendet, globale Funktionen zu definieren oder um das Verhalten der Klasse als Ganzes zu beschreiben.

Klasseninvariante: Die Klasseninvariante ist eine logische Aussage über die Bedeutung und die Wertebelegung der Variablen eines Objekts, die außerhalb der momentanen Ausführung einer Methode immer erfüllt sein muss. Oft bestehen zwischen verschiedenen Variablen bestimmte einschränkende Beziehungen. Wenn zum Beispiel aus Effizienzgründen die Länge einer verketteten Liste in einer eigenen Variablen gespeichert ist, muss diese redundante Information stets mit der tatsächlichen Länge der Liste übereinstimmen. Das Aufrechterhalten der Klasseninvariante bedeutet nichts anderes, als dass die in der Klasse festgelegte Bedeutung der Variablen während der gesamten Lebensdauer eines jeden Objekts der Klasse gültig ist.

Klassenvariable: Klassenvariable (*statische Variable*) sind in Java Variable, die in der Klasse mit dem Vorsatz `static` deklariert sind. Sie werden pro Klasse nur einmal angelegt und existieren im Unterschied zu den Instanzvariablen, die zu einem einzelnen Objekt gehören, über die gesamte Programmlaufzeit. Sie entsprechen den globalen Variablen von C.

Komplexität: Unter der Komplexität eines Algorithmus versteht man seine Anforderungen an Betriebsmittel, wie Laufzeit, Speicherplatz und externe Ressourcen.

Konstruktor: Ein Konstruktor ist eine Prozedur, die ein neues Objekt initialisiert. Der Aufruf eines Konstruktors geschieht durch einen New-Ausdruck. Der für das Objekt nötige Speicher wird automatisch bereitgestellt. Die Aufgabe des Konstruktors besteht darin, die Instanzvariablen so zu initialisieren, dass die Invariante der Klasse erfüllt ist.

Korrektheit: Mit Korrektheit bezeichnet man die Übereinstimmung eines Programms mit seiner Spezifikation. Ein Programm, das zu zulässigen Eingaben genau die erwarteten Ergebnisse ermittelt, ist korrekt.

Liste: Eine Liste ist eine abstrakte Datenstruktur, die eine Menge von Datenelementen in einer bestimmten Reihenfolge speichert.

Liste, verkettete: Eine *verkettete Liste* ist eine konkrete Implementierung einer abstrakten Liste. Man unterscheidet die *einfach verkettete Liste*, bei der in jedem Listenknoten die Referenz des nachfolgenden Knotens gespeichert ist, von der *doppelt verketteten Liste*, bei der zusätzlich die Referenz des Vorgängerknotens bekannt ist.

Methode: Die konkreten Realisierung einer Schnittstellenoperation eines Objekts heißt *Methode*. Eine Methode legt die *Art und Weise* fest, wie ein Objekt auf eine Nachricht reagiert. Methoden können Parameter und einen Rückgabewert besitzen. Innerhalb einer Methode kann das Empfängerobjekt (mit dem die Methode gerade aufgerufen wurde) mit `this` angesprochen werden. Eine Methode unterscheidet sich von einer Funktion: sie operiert auf einem Empfängerobjekt und der Aufruf der Methode wird nicht durch den Programmtext festgelegt, sondern durch das aktuelle Empfängerobjekt bestimmt.

O-Notation: Die O-Notation bezeichnet in der Mathematik die Angabe des führenden Terms einer Formel. Bei Algorithmen beschreibt man mit ihr die Abhängigkeit der Laufzeit eines Algorithmus von der Problemgröße. Konstante Faktoren werden weggelassen. Beispiele sind $O(n)$, $O(n^2)$ und $O(n \log n)$. In der Literatur wird die Schreibweise mit dem „großen O“ meist für den Laufzeit des ungünstigsten Falls verwendet. Die O-Notation wird aber auch für andere Zusammenhaänge, wie z.B. für die Abhängigkeit des Speicherbedarfs von der Problemgröße verwendet.

Obertyp Wenn ein Typ T_1 ein Obertyp von T_2, so sind alle Operationen von T_1 auch durch T_2 definiert. Ein Interface in Java bezieht sich mittels der Extends-Angaben auf eine Liste von Obertypen. Für eine Klasse bilden die Oberklasse und die implementierten Schnittstellen Obertypen. Die Klasse `Object` ist Obertyp aller Schnittstellen und Klassen. Die Obertypbeziehung ist transitiv.

Objekt: In der allgemeinsten Bedeutung meint man damit ein beliebiges Programmobjekt, das Programmspeicher belegt. Dies sind insbesondere Datenobjekte und Funktionsobjekte. Im engeren, objektorientierten Sinn steht der Begriff Objekt für die Instanz einer Klasse. In der objektorientierten Bedeutung ist ein Objekt durch eine eigene Identität, ein in der Klasse festgelegtes Verhalten und durch seinen besonderen Zustand charakterisiert. Jedes Objekt hat einen durch seine Klasse definierten dynamischen Typ, der sein konkretes Verhalten beschreibt.

Operation: Eine Operation steht für die funktionale Verknüpfung von Operanden oder für die zulässige Nachricht („Methodenaufruf“) an ein Objekt. Jede Operation hat einen Namen und eine Signatur. Objektorientierte Operationen werden durch Methoden implementiert.

Paket: In Java gehört jede Klasse zu einem Paket. Der Paketname besteht aus mehreren durch Punkt getrennten Namensbestandteilen. Der vollständige Name einer Klasse ist der Paketname, gefolgt von einem Punkt und dem eigentlichen Klassennamen. Die Import-Anweisung kann verwendet werden, um die abgekürzte Bezeichnung der Klasse ohne Paketangabe zu ermöglichen. Für das Paket `java.lang` nimmt der Compiler automatisch diesen Import vor. Fehlt bei einer Klasse die Paktetanweisung, so gehört die Klasse zu dem anonymen Paket. Die Paketzugehörigkeit hat auch Konsequenzen für die Sichtbarkeit von Klassen-, Methoden- und Variablennamen (`protected` und Defaultsichtbarkeit).

Persistenz: Persistenz bedeutet so viel wie Dauerhaftigkeit. Programmiersprachen unterstützen unmittelbar nur Programmobjekte, die während eines Programmlaufs innerhalb eines einzigen Computers existieren. Mit Persistenz meint man demgegenüber ein Verhalten, dass diese engen Grenzen sprengt und einem Objekt zu dauerhafter und auch räumlich verteilter Existenz verhilft. Das wichtigste Mittel zur Erreichung von Persistenz ist die Speicherung von Programmdaten auf einem nichtflüchtigen Speichermedium. Java unterstützt die Objektpersistenz unter anderm durch den Mechanismus der *Serialisierung*.

Polymorphie: Unter Polymorphie versteht man, dass eine Variable, Datenstruktur oder Funktion oder Methode Objekte unterschiedlichen Typs speichern oder verarbeiten kann.

private Das Schlüsselwort `private` definiert eine auf die aktuelle Top-Level-Klasse beschränkte Sichtbarkeit.

protected: Das Schlüsselwort `protected` erweitert die Defaultsichtbarkeit innerhalb des aktuellen Pakets auf alle Klassen, die von der aktuellen Klasse direkt oder indirekt abgeleitet sind.

public: Das Schlüsselwort `public` dehnt die Sichtbarkeit auf alle Klassen aus.

Queue: Siehe: *Warteschlange*

Referenzübergabe: Mit Referenzübergabe bezeichnet man einen Mechanismus zur Übergabe von Argumenten an Funktionen. Während bei der *Wertübergabe* der Wert eines Argumentausdrucks übergeben wird, wird bei der Referenzübergabe die Referenz (die Speicheradresse) einer Variablen übergeben.

Rekursion: Funktionen, die sich (direkt oder indirekt) selbst aufrufen, heißen rekursiv. Rekursion ist neben der Iteration ein Mittel um Programmteile wiederholt auszuführen. Die Grundidee der Rekursion besteht darin, dass eine Funktion sich für ein kleineres Teilproblem erneut aufruft. Eine Besonderheit der Rekursion gegenüber der Iteration besteht darin, dass für jeden Aufruf neue lokale Variable erzeugt werden. Damit eine rekursive Funktion terminiert, muß sie unbedingt eine Abbruchbedingung enthalten, bei der die Kette der rekursiven Aufrufe beendet wird. Der Begriff *Rekursion* lässt sich nicht ohne weiteres auf Methoden übertragen, da im Allgemeinen nicht im Voraus bekannt ist, durch welche Methode ein Aufruf ausgeführt wird.

Robustheit: Eine Programmeinheit ist robust, wenn sie auch bei falscher Eingabe ein definiertes Verhalten zeigt. Die Robustheit von Methoden erfordert, dass die Vorbedingungen einer Methode geprüft und im Fehlerfall eine Ausnahme (exception) erzeugt wird.

Schleife: Mit Schleife oder Schleifenanweisung sind Iteration und Iterationsanweisungen gemeint.

Schleifeninvariante: Eine Schleifeninvariante ist eine logische Aussage, die die Bedeutung der im Schleifenkörper vorkommenden Variablen genau beschreibt. Da die Schleifeninvariante während aller Schleifendurchläufe und auch noch nach Verlassen der Schleife gilt, muß aus ihr und aus der nichterfüllten Schleifenbedingung die Zielbedingung folgen.

Schnittstelle: Die Schnittstelle eines Objekts ist definiert durch die Menge der durch das Objekt definierten Operationen. Die Schnittstelle eines Objekts wird durch seine Klasse definiert. Java-Interfaces definieren abstrakte Schnittstellen.

Semantik: Unter der Semantik eines Satzes (oder eines Computerprogramms) versteht man seine Bedeutung. Die Bedeutung entsteht dabei erst durch eine Interpretation, die den einzelnen Symbolen und Beziehungen sinnvolle Inhalte zuordnet.

Smalltalk: In den 70er Jahren wurde in der kalifornischen Forschungseinrichtung Xerox-Parc Smalltalk als eine benutzerfreundliche Programmiersprache für Arbeitsplatzrechner mit graphischer Oberfläche entwickelt. Smalltalk ist die älteste Programmiersprache, die durchgängig nach objektorientierten Grundsätzen aufgebaut ist. Durch Smalltalk wurde Objektorientierung populär.

Spezifikation: Eine Spezifikation ist eine (möglichst) genaue Beschreibung der Aufgabenstellung. Zu einer Spezifikation gehört die Angabe, wie die Übereinstimmung des entwickelten Systems mit den Anforderungen getestet werden kann.

Späte Bindung: Mit später Bindung ist gemeint, dass die Bindung zwischen Aufruf und Ausführung einer Operation erst zur Laufzeit eines Programms erfolgt. Die späte Bindung bildet die Grundlage des objektorientierten Verhaltens. Siehe auch: *Binden*.

Stack, Stapelspeicher: Unter einem Stapelspeicher, auch Stack genannt, versteht man einen Behälter mit zwei verändernden Operationen *push* zum Einfügen und *pop* zum Entfernen, die der Bedingung genügen, dass die Daten in umgekehrter Reihenfolge zur Reihenfolge des Einfügens entnommen werden (LIFO = *last in first out*).

static: Das Schlüsselwort `static` bezeichnet Klassenvariable und Klassenfunktionen. Als Vorsatz vor geschachtelten Klassen, sagt es aus, dass die Klasse den umfassenden Kontext (Objekt oder Methode) nicht kennt.

super: Das Schlüsselwort `super` bezieht sich auf die Eigenschaften eines Objekts, die in der Operklasse (super class) deklariert sind. Wenn Oberklasse und aktuelle Klasse z.B. eine gleichnamige Instanzvariable `var` deklariert haben, so bezeichnet `var` oder `this.var` die Variable der Objektklasse und `super.var` die verdeckte Variable aus einer der Oberklassen. Entsprechend steht `super.method()` für den Aufruf einer überschriebenen Methode. Die Syntax `super(...)` steht für den Aufruf des Konstruktors der Oberklasse.

Syntax: Unter der Syntax oder Grammatik einer Sprache versteht man die Menge der formale Regeln, aus denen sich ableiten lässt, welche Sätze (syntaktisch) richtig gebildet sind. Die Syntax einer Programmiersprache gibt formale Bildungsregeln für den Aufbau von Computerprogrammen vor.

Test: Durch den Test eines Programms können Programmierfehler entdeckt werden. Ein zufriedenstellend abgelaufener Test beweist jedoch nicht, dass das Programm fehlerfrei ist. Die Wirksamkeit von Tests wird enorm erhöht, wenn nicht nur das Gesamtsystem sondern auch alle Komponenten für sich getestet werden. Der Programmtest wird häufig durch die formale Verifikation einzelner kritischer Programmteile ergänzt.

Testgetriebene Entwicklung (test first); Die Maxime, den Test vor der Programmierung zu entwickeln, ist eine moderne Programmiermethodik (kein Testverfahren!). Sie zwingt den Programmierer dazu, zunächst durch die Formulierung von Tests genau festzulegen, was er von der gerade zu entwickelnden Einheit erwartet. Die eigentliche Programmierung besteht dann darin, die neue Programmeinheit so zu entwerfen, dass schließlich der Test erfüllt ist. Dabei wird der Entwickler durch einfach zu nutzende Hilfsmittel, wie JUnit, wirksam unterstützt.

this: Das Schlüsselwort `this` steht für das aktuelle Objekt, das gerade eine Methode ausführt. Wenn es von einer Parameterliste gefolgt ist, bedeutet es den Aufruf eines der Konstruktoren der Klasse (aus einem anderen Konstruktor heraus).

Typ: Ein Typ beschreibt die Eigenschaften und das Verhalten von Datenobjekten. Unter dem Gesichtspunkt der Objektorientierung ist ein Typ durch seine Schnittstelle, d.h. durch die Menge der (öffentlichen) Operationen beschrieben.

Typ, dynamischer: Der dynamische Typ eines Objekts ist durch seine Klasse definiert. Er entscheidet über die späte Bindung von Methoden. Der dynamische Typ kann durch eine Abfrage über `instanceof` überprüft werden. Zur Laufzeit wird überprüft, ob der bei einer Typanpassung angegebene statische Typ auch wirklich mit dem dynamischen Typ des Objekts verträglich ist.

Typ, statischer: Der statische Typ eines Ausdrucks ist durch die Typregeln der Programmiersprache, durch den Typ von Literalen, Variablen und durch den Rückgabetyp von Methoden bestimmt. Der Java-Compiler erlaubt nur solche Zugriffe auf ein Objekt, die durch den statischen Typ erlaubt sind.

Typangabe: Mit Typangabe bezeichnet man den (Cast-) Ausdruck zur Angabe und Überprüfung Referenztyps. Die Notwendigkeit dieser Typangabe ist eine Folge der Objektorientierung. Objektorientierung erlaubt es nämlich Referenzen in Variablen des Obertyp des zugewiesenen Ausdrucks zu speichern. Eine Variable kann so Referenzen auf Objekte eines Untertyps ihres Typs enthalten. Untertypen verfügen in der Regel über mehr Operationen als Obertypen. Daher kann es notwendig sein, die genauere Typinformation anzugeben. Mit der Typangabe kann der Programmierer die Typinformation eines Ausdrucks präzisieren. Der Compiler akzeptiert die Angabe, wenn sie korrekt sein *kann* (der angegebene Typ ist ein Untertyp des Ausdruckstyps). Er erzeugt eine Anweisung, die zur Laufzeit eine exakt prüft, ob die Referenz wirklich den angegebenen (Ober-) Typ hat. Ist diese Typprüfung nicht erfolgreich, wird eine `ClassCastException` geworfen.

Typkonvertierung: Mit Typkonvertierung bezeichnet man die Wirkung der Cast-Operation bei *numerischen* Daten. Sie besteht darin, dass die Darstellung des betroffenen Ausdrucks in die Zahlendarstellung des angegebenen numerischen Typs umgewandelt wird. Grundsätzlich ist die Umwandlung einer jeden Zahlendarstellung in jede andere möglich. Dabei kann mitunter Informationsverlust oder Zahlenüberlauf auftreten. Numerische Werte können nicht in nichtnumerische Werte umgewandelt werden oder umgekehrt. Wenn die Typkonvertierung von einer schwächeren Darstellung in eine umfassendere Darstellung (z.B. von `int` nach `long` erfolgt, kann die Angabe eines Casts unterbleiben. Der Compiler veranlasst in diesem Fall automatisch die nötige Umwandlung. Im Unterschied zur *Typanpassung* bei Referenztypen bewirkt die Cast-Operation keine Typprüfung zur Laufzeit (und damit auch niemals eine Ausnahme). Typprüfung für die nichtobjektorientierten Werttypen wird ausschließlich vom Compiler vorgenommen.

Typlöschung: Mit Typlöschung (type erasure) bezeichnet man den Umstand, dass Typparameter zur Laufzeit nicht bekannt sind. Typparameter dienen in Java nur der genaueren Typprüfung im Compiler. Es ist bei Typparametern nicht möglich, wie bei den „normalen“ Typen Prüfungen zur Laufzeit vorzunehmen. Ebenso ist es nicht möglich Objekte und Arrays mit dem Typ eines Typparameters zu erzeugen. Die Typlöschung zwingt der Programmiersprache Java einige sehr komplizierte Regeln auf. Sie war für die Sprachentwickler aber nötig um die Weiterverwendung von bestehendem Code nicht zu gefährden.

Untertyp: Ein Untertyp erweitert einen Obertyp um weitere Operationen. Die Untertypbeziehung ist transitiv. Siehe auch *Obertyp*.

Vererbung: In einer abgeleiteten Klasse gelten die in der Oberklasse definierten Daten- und Methoden. Variable der Oberklasse können in Java durch Variable der Unterklasse verdeckt (aber nicht ersetzt) werden. Methoden gleichen Namens und gleicher Signatur *überschreiben* (vielleicht besser: *überstimmen*) die gleichartigen Methoden der Oberklasse. Die Übernahme von Elementen der Oberklasse heißt *Vererbung*.

Verhalten: In der objektorientierten Programmierung versteht man unter dem Verhalten eines Objekts die Art und Weise wie der Zustand eines Objekts abgefragt oder verändert werden kann oder wie ein Objekt dazu veranlasst werden kann, weitere Operationen auszuführen – kurz gesagt: Das Verhalten eines Objekts ist durch seine Methoden definiert.

Verifikation: Unter Verifikation versteht man das Bemühen, sich mit formalen Mitteln von der Korrektheit eines Programms zu überzeugen. Verifikation und Test sind zwei sich ergänzende Verfahren zur Gewährleistung der Korrektheit.

Vertrag Funktionen und Methoden stellen insofern einen Vertrag dar, als sie für den Fall, dass Anfangsdaten und Parameter die *Vorbedingung* erfüllen, garantieren, dass das Funktions- oder Methoderesultat und der neue Zustand des Empfängerobjekts eine garantierte *Nachbedingung* erfüllen. Der Vertrag einer Funktion oder Methode wird in Java durch den *Methodenkommentar* beschrieben.

Warteschlange, Queue: Unter einer Warteschlange, auch Queue genannt, versteht man einen Behälter, in dem nur über die *put*-Operation (*enqueue*) Daten abgelegt und aus dem nur über die *get*-Operation (*dequeue*) Daten entnommen werden können. Beim Entnehmen erscheinen die zuvor eingefügten Daten in unveränderter Reihenfolge (FIFO = *first in first out*).

Wertübergabe: Die Wertübergabe (call by value) ist eine Form der Parameterübergabe in Funktionen. Bei der Wertübergabe werden die Werte der aktuellen Parameter in lokale Variable kopiert.

Wiederholung: Die Wiederholung von Programmteilen wird durch Iteration und durch Rekursion ausgedrückt.

Zusicherung: Eine Zusicherung ist eine logische Aussage. Innerhalb eines Computerprogramms werden Zusicherungen häufig formuliert, um die Verifizierbarkeit und Testbarkeit eines Computerprogramms zu erhöhen.

Zustand: Der Zustand eines Programms ist definiert durch die Werte aller zu einem bestimmten Zeitpunkt gültigen Variablen und durch die Position der nächsten auszuführenden Anweisung (Wert des Programmzeigers). Der Zustand eines Objekts ist bestimmt durch die Werte der Instanzvariablen.

Zuweisung: Eine Zuweisung wird in Java durch das Zeichen = ausgedrückt. Sie dient dazu, den Inhalt einer Variable zu verändern. Bei Wertobjekten wird in Java der Wert kopiert, bei Referenzobjekten nur die Objektreferenz.

Literaturverzeichnis

[Blo2001] Joshua Bloch, *Effective Java*
Addison-Wesley 2001
Das Buch beschreibt anhand einiger Idiome wie man die Prinzipien der Objektorientierung in Java umsetzt. Joshua Bloch hat das Collection Framework der Java-Bibliothek entwickelt.

[Boo1991] Grady Booch. *Objektorientierter Entwurf*
Addison-Wesley 1994
Ein grundlegendes Buch zum Verständnis und zur Methodik des objektorientierten Softwareentwurfs. Es wird Ihnen allerdings eher bei realen Problemen, z.B. dem Auffinden von geeigneten Klassen und ihren Beziehungen helfen, als bei den (einfachen) Problemen dieser Vorlesung.

[Eck1999] Bruce Eckel. *Thinking in Java*
http://www.BruceEckel.com
Das Buch (in elektronischer Form frei erhältlich) diskutiert sehr umfassend Sprach- und Entwurfskonzepte von Java

[Fow2000] Martin Fowler, *Refactoring, Improving the Design of Existing Code*
Addison-Wesley 2000
Das Buch beschreibt, wie man Schritt für Schritt existierene Programme verbessert und dabei neue Fehler vemeidet.
Besonderen Wert erhält das Buch auch dadurch, dass einige der vorgestellten Verfahren von modernen Entwicklungsumgebungen automatisiert werden.

[Fut1989] Gerald Futschek. *Programmentwicklung und Verifikation*
Springer-Verlag 1989
Die Methoden der systematischen Programmkonstruktion und der Verifikation werden (soweit das bei dem etwas theoretischen Stoff geht) anschaulich dargestellt. Das Buch wendet sich an professionelle Programmierer.

[Gam1995] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides *Entwurfsmuster*
Addison-Wesley, 1995
Das Buch der *Viererbande* hat sich inzwischen zu *dem* Standardwerk für die professionelle Entwicklung von objektorientierter Software entwickelt. Es zeigt auf, dass es für die verschiedenen Bereiche der Anwendung der Objektorientierung wiederkehrende Muster der Klassen- und Objektbeziehungen gibt. Die Kenntnis dieser Muster ist extrem hilfreich bei der Entwicklung und bei dem Verständnis komplexer Software.

[Gos2000] James Gosling, Bill Joy, Guy Steel, Gilad Bracha. *The Java Language Specification. Second Edition*
Sun Microsystems 2000
Dies ist die verbindliche und größtenteils sogar gut lesbare Festlegung der Sprachregeln von Java. Die derzeitige Ausgabe enthält noch nicht die Spracherweiterungen durch Java 5. Der Text ist auch über die Sun-Website erhältlich.

[Gri1983] David Gries. *The Science of Programming.*
Springer-Verlag 1983
Ein begeisterndes Buch über formale Verfahren zur Entwicklung korrekter Algorithmen. Allerdings nur für Fortgeschrittene und besonders Interessierte.

[KR1977] Brian W. Kernighan, Dennies M. Ritchie. *The C Programming Language*
Hanser Verlag 1978
Das Standard- und Nachschlagewerk zu C

[Schn2000] U. Schneider, D. Werner.
Taschenbuch der Informatik
Hanser-Verlag 2000
Ein sehr umfassendes Nachschlagwerk über alle wichtigen Bereiche der anwendungsorientierten Informatik.

[Sed1992] Robert Sedgewick. *Algorithms in Java*
Addison Wesley 2003
Das Buch beschreibt die wichtigsten Algorithmen, die der Student beherrschen sollte. Die Algorithmen sind sehr anschaulich erläutert. Wer eines der anderen Algorithmen-Bücher von Sedgewick besitzt, kann dies genauso gut benutzen, da Sedgewick von Java praktisch keinen Gebrauch macht.