Schriftliche Hausarbeit zur Abschlussprüfung der erweiternden Studien für Lehrer im Fach Informatik

VIII. Weiterbildungskurs in Zusammenarbeit mit der Fernuniversität Hagen

Eingereicht dem Amt für Lehrerausbildung, Fuldatal

Mit Java in die Oberstufe

Eine Handreichung für den Umstieg von Pascal auf Java

Verfasser: Franz Beslmeisl
Gutachter: Rolf Zimmermann

Vorwort

Diese Examensarbeit soll Ihnen als Informatik-Lehrer[1] Mut machen, Ihren Unterricht einmal nicht mit Pascal zu beginnen, sondern mit Java. Ein solcher Schritt bringt immer die Unannehmlichkeit mit sich, dass Neues gelernt werden muss. Manches scheint deshalb schwieriger und die Mühe nicht wert.

Wenn Sie allerdings innerlich den Entschluss schon gefasst haben, mit Java arbeiten zu wollen, wenn Sie sich schon darauf freuen und nur noch die unbestimmte Angst haben, es könnte im Unterricht etwas schief gehen, dann können Sie beim Durchgehen dieses Textes Sicherheit sammeln. Der Text ist dabei eine Art Tutor, der Sie (mit einer Java-Brille) durch die im schulischen Alltag auftretenden Stationen führt. Wegen Ihres eigenen Erfahrungsschatzes werden Sie manches Kapitel problemlos überspringen oder den Wunsch verspüren, ein Thema weiter auszubauen. Werfen Sie dann bitte einen Blick in die Literaturliste oder die Dokumente auf der beiliegenden CD. (Auf der CD befindet sich auch dieser Text mit seinen Programmbeispielen[2] und einige frei erhältliche Werkzeuge zur Programmierung in Java.)

Noch eine Anmerkung zur Übersetzung englischer Begriffe: Es ist unmöglich, das Englische so zu verwenden, dass es jeder Leser als angemessen empfindet. Auf dem Gebiet der Informatik finde ich es angenehm, existierende deutsche Begriffe zu verwenden und nicht existierende oder zwanghaft eingedeutschte lieber englisch zu belassen. Ein Beispiel: Hätte man immer schon Array zum Array gesagt, so könnte man jetzt Feld zum Field sagen. Weil man aber seit jeher Feld zum Array sagt, muss man jetzt Attribut zum Field sagen.

Übrigens ist das Wort *Klasse* im Zweifel als die deutsche Übersetzung von `class` zu verstehen. Eine Menge von Schülern nenne ich bevorzugt *Kurs*.

Letzte Korrektur am 13. Mai 2006

[1] unerheblich, welcherlei Geschlechts, Hautfarbe, Religion... Verstehen Sie die neutrale Form bitte neutral!
[2] Die Querverweise sind dort verlinkt.

Inhaltsverzeichnis

Einleitung

Als *Sun Microsystems* 1995 die erste Version von Java zur Verfügung stellte, stieß die neue Programmiersprache schnell auf große Begeisterung und – damit einher gehend – auf weite Verbreitung: An den Universitäten wird die objektorientierte Programmierung (OOP) fast ausschließlich mit Java gelehrt. Das Internet ist voller Java-Applets in allen nur denkbaren Einsatzgebieten, winzig aber leistungsstark. Als die NASA 1997 den Mars-Pathfinder entwickelte, wurde die Steuerung in Java programmiert, nicht nur weil man das Gerät damit von jedem gewöhnlichen PC (auch außerhalb Houstons) steuern konnte, sondern weil die Entwicklungszeit unschlagbar kurz war und der Entwicklungserfolg praktisch sicher[3].

In den Schulen merkt man von solcher Begeisterung nicht viel. Nur wenige Fachkollegen vertreten den Standpunkt, Java *solle* Unterrichtssprache in der Schul-Informatik sein, bestenfalls meint man, es *könne*. Die Gründe für diese Zurückhaltung sind verschiedener Natur:

- Es bestehen kleine syntaktische Unterschiede zu Pascal. Wer sehr wenig Zeit hat, sieht hier schon eine Hemmschwelle, deren Überwindung nicht wirklich Vorteile verspricht. Schließlich hat man mit Delphi[4] eine objektorientierte Erweiterung von Pascal, die scheinbar alles nötige kann.

  Pragmatisch betrachtet ist das nicht ganz falsch. Die Einarbeitung in Java ist auch nicht wegen der Syntax anspruchsvoll, sondern wegen der sehr sauberen objektorientierten Konzepte, die deutlich über die Möglichkeiten von Delphi hinaus gehen. Wenn man nicht schon mit der objektorientierten Denkweise aufgewachsen ist, hilft Java sehr bei der Erarbeitung der wesentlichen Ideen[5].

  Delphi ist in diesem Punkt nicht einfacher, sondern verführerischer. Erst fängt man klein an und freut sich, dass mit Hilfe der vorhandenen Klassen alles so schnell geht. Wenn die Projekte aber interessanter und größer werden, wird aus Programmieren *mit Objekten* nicht etwa *objektorientiertes Programmieren*, sondern *schlampiges Programmieren mit Objekten*. Man verzichtet auf Daten-Kapselung, verwendet globale Variable. Methoden werden nicht als Teil der Klasse, sondern als Prozedur im Hauptprogramm realisiert. Wenn OOP später Thema wird, theoretisiert man, ohne die Vorteile jemals richtig gefühlt zu haben.

- Die sichtbaren Komponenten einer graphischen Oberfläche (GUI) sind mit Delphi viel leichter handhabbar, Ereignisse sind etwas von Anfang an nicht Hinterfragtes. In Java braucht man Layoutmanager und muss sich dauernd irgendwo anmelden, um Ereignisse zu bekommen.

  Dieses Argument ist besonders im Anfängerunterricht der 11. Klasse sehr gewichtig, geradezu ausschlaggebend. Schließlich will man den Anfänger nicht mit fortgeschrittenen Konzepten quälen. Aber auch für Java gibt es Entwicklungsumgebungen, die diese technischen Details so lange selbständig in die Hand nehmen, wie sie den Anfänger nicht interessieren. (Übrigens auf eine sehr transparente Weise. Bei Delphi in Schülerhand werden ja oft wesentliche Dateien inkonsistent, so dass nach einem kleinen Fehler trotz vorhandenem Quelltext ein Projekt neu angelegt werden muss.)

  In diesem Text finden Sie deshalb ein Kapitel über die Verwendung von NetBeans, einer Entwicklungsumgebung (IDE), die Delphi in nichts nachsteht und frei erhältlich ist. In den anderen Kapiteln wird darauf verwiesen, das GUI aber klassisch programmiert. In der 12. Klasse muss man nämlich das Ereigniskonzept von Java durchaus nicht mehr als umständlich und unangenehm empfinden, sondern kann es als klar gegliedertes und kluges Beispiel für den Einsatz von Objekten und das Senden von Nachrichten zwischen diesen in den Unterricht aufnehmen.

- Der Java-Compiler erzeugt keine echte Maschinensprache. Deshalb sind Java-Programme etwa um den Faktor 5 langsamer. Der Geschwindigkeitsnachteil tritt im Unterrichtseinsatz jedoch nicht zu Tage. Wenn man wirklich einmal etwas programmiert, das in Java merklich langsam läuft, dann läuft es wahrscheinlich auch in Pascal oder C langsam.

- In Java kann man nicht leicht auf Hardware-Ebene programmieren. Wer also die Feinheiten einer Graphikkarte ausnützen will, hat es schwer, an diese heranzukommen. Aber auch das ist für schulische Zwecke natürlich kein Nachteil.

[3] Auch das Hubble-Teleskop wird mittlerweile mit Java-Applets gesteuert. Beim aktuellen Mars-Mobil *Spirit* wurde die Technik noch ausgebaut. Es wird davon ausgegangen, dass sich die Entwicklungszeit beim Einsatz von Java im Vergleich zu den bisherigen Technologien halbiert.

[4] Gemeint ist natürlich die Sprache *Pascal mit Objekten*. Aber meist wird das Werkzeug Delphi wegen der engen Zusammenarbeit mit der Sprache mit dieser identifiziert.

[5] Diese Aussage ist persönlich gefärbt.

Nun zu den verlockenden Vorteilen von Java:

- Alles was es in Pascal gibt, gibt es in ähnlicher Form auch in Java. Die Syntaxunterschiede muss niemand fürchten.
- Die Grundlagen imperativer Programmierung kann man in Java lehren, ohne über Objekte sprechen zu müssen. (Strings sind hier eine kleine Ausnahme, die der Anfänger aber nicht bemerkt.)
- Die Sprache Java ist noch strenger definiert als Pascal. Die Situation, dass man manche Konstrukte verbieten muss, weil man nicht sicher sein kann, wie sie der Compiler behandelt, tritt nicht mehr auf. Bei allem, was der Compiler akzeptiert, ist in der *Java Language Specification* definiert, was genau passiert.
- Java ist streng typisiert und regt zu sauberem, modularisiertem Programmieren an.
- Java-Programme laufen ohne Änderung auf allen Rechnern, die eine Java-Umgebung zur Verfügung stellen. Java-Umgebungen gibt es für Windows, Macintosh, Linux und verschiedene andere Unixe.
- Die Java-Umgebung und mehrere beeindruckende IDEs sind kostenlos erhältlich. Gewissenhafte Lehrer und Schüler dürfen also ein reines Gewissen behalten, auch wenn sie für die Werkzeuge nichts bezahlt haben.
- Man kann fast alles auch in Form eines Applets programmieren und somit in eine HTML-Seite einbinden. Erklärende Texte und die Funktionalität des Programms sind also trefflich kombinierbar.

  Dieser Punkt ist einer der überzeugendsten! Schüler haben Homepages und lieben es, ihre eigenen Werke dort auszustellen. Auch als Lehrer freut man sich gelegentlich, kleine Übungsapplets für die Schüler bereitstellen zu können, die dann ohne weitere Installation im Browser laufen.
- Java-Programme sind oft um den Faktor 100 kleiner als Delphi-Programme gleicher Funktionalität. Dieser Vorteil verringert sich erst bei großen Projekten, geht aber nie ganz verloren.
- Die Standard-Pakete ermöglichen es, problemlos in viele moderne Themenbereiche einzudringen: Graphik (mit allgemeinen affinen Transformationen in Matrixdarstellung), Sound (alle gängigen Formate, z. B. MIDI), Multithreading, Verschlüsselung, Serialisation, Datenbanken, Internet-Protokolle und vieles andere; alles auf musterhaft durchdachtem und ausgearbeitetem Niveau. Zum Thema RMI (Remote Method Invocation) etwa finden Sie in diesem Text ein Beispiel zur Aufteilung einer Aufgabe auf mehrere Rechner im Netz (Grid-Computing).

1 Erste Schritte

In diesem Kapitel wird gezeigt, welche Schritte zu unternehmen sind, damit man auf einem Rechner einigermaßen angenehm Java-Programme erstellen und ausführen kann. Dazu sollte man ausreichend verstehen, was auf diesem Weg alles passiert.

Es ist für den Anfang ganz lehrreich und gar nicht anstrengend, die drei Entwicklungsschritte Editieren, Übersetzen und Ausführen/Testen selber anzustoßen. Entwicklungsumgebungen (IDEs) nehmen einem zwar jede Unannehmlichkeit ab, aber eben auch eine ganze Menge Verständnis fürs Detail. Lassen wir uns also mit der Installation der wunderbaren IDE NetBeans wenigstens bis zum Ende des Kapitels Zeit.

Zuerst muss man natürlich auf dem Rechner...

1.1 Java installieren

Java-Umgebungen für verschiedene Betriebssysteme bekommt man kostenlos beim Erfinder Sun unter der Adresse `http://java.sun.com/j2se/1.4/download.html`.

Verschiedene Java-Versionen unterscheiden sich weniger in den beiliegenden Werkzeugen (wie etwa dem Compiler) als vielmehr in den stets wachsenden Bibliotheken, die in `jar`-Dateien vorliegen und mit ihren Tausenden Klassen die ganze vorhandene Funktionalität bereitstellen. Dies ist auch der Grund, warum Java-Programme so kurz sind: Für alle Standardprobleme werden ausgereifte und stark optimierte Lösungen angeboten. Der Programmierer muss wirklich nur noch *sein* Problem lösen.

Es werden immer zwei Sorten zum Download angeboten: Das Java-Runtime-Environment (JRE) ist eine Teilmenge der Vollausstattung (J2SE) und reicht aus, um Java-Programme auszuführen. Will man aber selber programmieren, so braucht man die Standardedition (J2SE). Es handelt sich dabei jeweils um selbstentpackende Programm-Dateien, die man nur einmal starten muss. Man kann dann während der Installation wählen, in welches Verzeichnis alles abgelegt wird. Im folgenden wollen wir davon ausgehen, dass J2SE installiert wurde und als Installationspfad `c:\Programme\java` angegeben wurde.

1.2 Die Dokumentation

Als Programmierer braucht man auch unbedingt die Dokumentation der Java-Umgebung mit ihren vielen Paketen. Der Entwicklungszyklus funktioniert zwar technisch auch ohne sie aber nicht intellektuell. Weil die Dokumentation so groß ist, wird sie unabhängig von der Java-Umgebung zum Download angeboten als `zip`-Datei, die man einfach an eine beliebige Stelle der Festplatte kopiert. Gelesen wird die Dokumentation mit einem gewöhnlichen Webbrowser, den man am besten so einstellt, dass er beim Start gleich die Wurzel-Seite der Dokumentation anzeigt.

Wenn wir davon ausgehen, dass wir die `zip`-Datei ausgepackt haben nach `c:\Books\java\docs`, so ist die Wurzel der Hilfe zu finden bei `c:\Books\java\docs\api\index.html`.

In der Dokumentation werden nicht die Sprachelemente erläutert; wie man eine Schleife programmiert, muss man also an anderer Stelle nachlesen (im vorliegenden Text etwa). Die Dokumentation gibt genaue Auskunft darüber, wie die vielen mitgelieferten Klassen wirken und wie sie zusammenwirken. Den Aufbau der Dokumentation und die Denkweise, die darin zum Ausdruck kommt, sollte man sich als Lehrer im Lauf der Zeit aneignen. Der Abstraktionsgrad der Texte ist nach der erlittenen Eingewöhnungsphase gut geeignet, zu sauber objektorientiertem Denken anzuregen. Die meisten Schüler hassen die Texte leider schon allein deswegen, weil sie in Englisch verfasst sind.

Die Dokumentation ist übrigens dadurch entstanden, dass die Programmierer der Klassen im Quelltext stets auf eine definierte Weise Kommentare hinterlassen haben, die am Ende von einem eigenen Programm zu der riesigen Gesamtheit zusammengestellt wurde. Dieser Hilfetext-Compiler heißt `javadoc`, liegt der Installation bei und kann auch für eigene Projekte verwendet werden.

Entwicklungsumgebungen wie NetBeans machen die Benutzung der Dokumentation besonders einfach. Man setzt den Cursor auf einen Methodennamen, drückt Alt-F1 und bekommt sofort die Erklärung. Trotzdem sollte man die Dokumentation auch regelmäßig als Ganzes mit dem Browser durchsuchen, um nicht die strukturellen Zusammenhänge aus dem Auge zu verlieren. Richten Sie die oben genannte Seite also wenigstens als einen Favoriten Ihres Browsers ein.

1.3 Editieren

Für den Anfang schadet es nicht, wenn man einen einfachen Texteditor verwendet. Die farbliche Hervorhebung der nachfolgenden Zeilen ist nicht Teil der Datei, sondern wird von manchen Editoren automatisch angeboten. Schreiben wir also folgendes kleines Programm:

```
public class Beispiel{
  public static void main(String[] args){
    System.out.println("\n\t\tDiese Ausgabe stammt von Beispiel.java");
  }
}
```

Da der Klassenname `Beispiel` ist, muss die Datei abgespeichert werden als `Beispiel.java`[1]. Das Verzeichnis auf der Festplatte ist beliebig, nehmen wir also z. B. `c:\Programme\test\`. Den weiteren Gang zeigt die DOS-Eingabeaufforderung[2] in Abbildung 1.1.

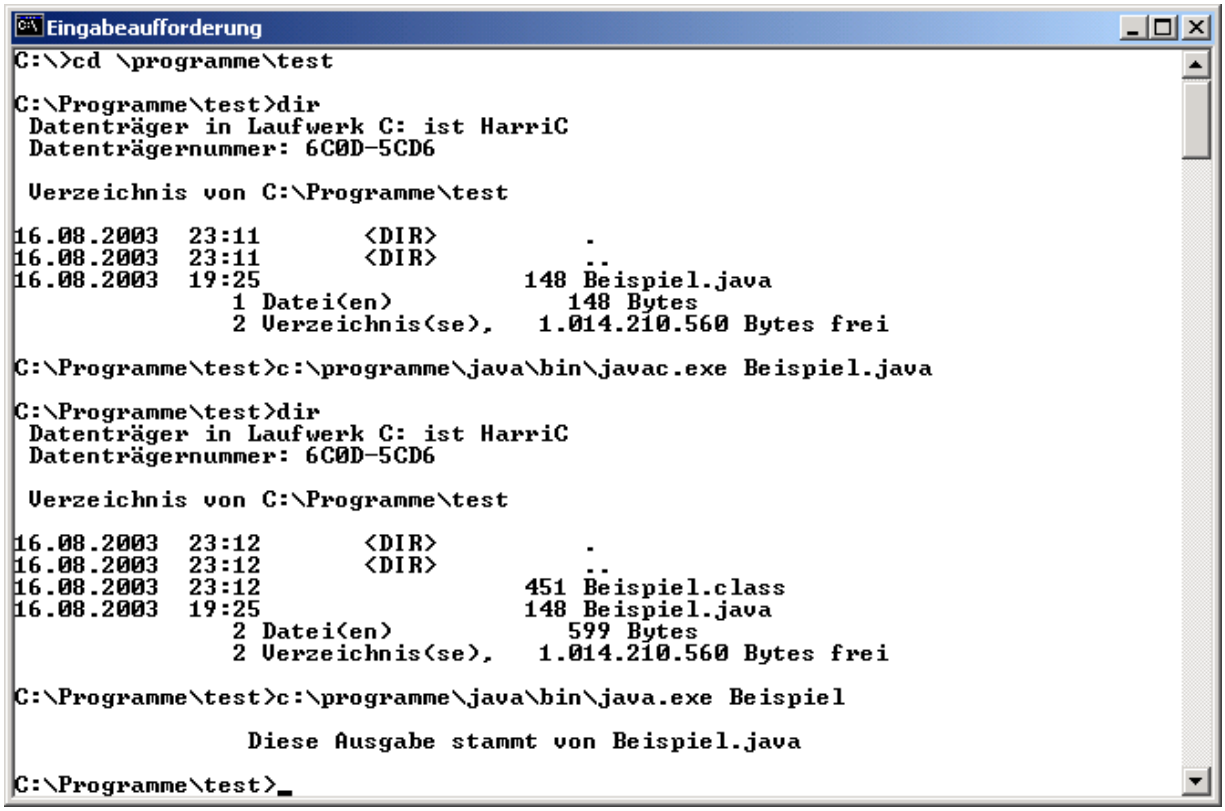

Abbildung 1.1: Ein Entwicklungszyklus in der Konsole (hier DOS-Box mit ungewöhnlicher Färbung)

[1] Notepad hat in manchen Windows-Versionen die unangenehme Eigenschaft, dass er beim Abspeichern die Endung `.txt` anfügt, was zudem der Explorer bei entsprechender Einstellung nicht einmal anzeigt. In diesem Fall ist es am besten, die Datei im Explorer mit der korrekten Endung `.java` anzulegen (mit Datei-Neu-Textdatei), diese dann durch Doppelklick zu öffnen, Notepad als Anzeigeprogramm zu wählen, und diesen während der ganzen Programmierarbeit nicht mehr zu schließen. Viel besser geht es mit dem Freeware-Editor *Crimson* auf der CD `//progs/cedt360r.exe`.

[2] Unter Linux nennt man diese *Konsole*. Der Ablauf ist ganz ähnlich.

1.4 Kompilieren

Das Programm muss nun compiliert werden. Zu diesem Zweck öffnet man eine Konsole (also etwa die besagte DOS-Eingabeaufforderung) und macht erst einmal das Programmverzeichnis zum aktuellen Verzeichnis mit `cd \Programme\test`. Das Bild zeigt, dass die einzige vorhandene Datei unser Programmtext `Beispiel.java` ist.

Der Compiler `javac` wird aus dem aktuellen Verzeichnis heraus nicht gefunden, muss aber aus diesem Verzeichnis heraus aufgerufen werden! Entweder setzt man jetzt den Pfad so, dass er gefunden wird, oder man gibt den Pfad des Compilers einfach jedesmal mit an. Damit das keine übermäßige Tipparbeit wird, sollte man `doskey` starten, damit man im Wiederholungsfall leicht mit den Cursortasten zu früheren Befehlen wechseln kann[3].

Wir starten den Compiler und sagen ihm, welche Datei er übersetzen soll. In unserem Fall geht das mit `c:\programme\java\bin\javac.exe Beispiel.java`. Windows unterscheidet zwar nicht Groß- und Kleinschreibung, aber `javac` tut es! Deshalb ist es wichtig, dass `Beispiel.java` genau so geschrieben wird[4].

Da wir keine Fehlermeldungen bekommen haben, war der Compiliervorgang erfolgreich, d. h. die ausführbare Datei `Beispiel.class` wurde erzeugt.

1.5 Ausführen

Was ist eine Virtual Machine?

Wenn ein Java-Programm fertig übersetzt und lauffähig ist, dann liegt es als eine oder mehrere `class`-Dateien[5] vor, bei uns `Beispiel.class`. Der Inhalt ist die Maschinensprache für einen Prozessor, den es nicht gibt, nämlich für einen Java-Prozessor. Diese Maschinensprache unterscheidet sich also von derjenigen für einen 80x86-Prozessor, wie man sie in einer `exe`-Datei findet.

Damit die `class`-Datei auch z. B. auf einem Intel-Prozessor funktioniert, muss dieser ein Programm ausführen, das einen Java-Prozessor emuliert (imitiert). Der Intel-Prozessor wird also für jeden Java-Befehl mehrere Intel-Befehle ausführen, die zusammen das gleiche bewirken, wie der Java-Befehl. Das ist ein Grund, warum Java-Programme so vergleichsweise langsam sind. Das Emulationsprogramm nennt man eine JVM (Java Virtual Machine). Es gibt natürlich auch JVMs für andere Prozessoren, wie etwa den Motorola-Prozessor eines Macintosh-Rechners. Die JVM ist für den Programmierer der Garant dafür, dass sein Java-Programm genau das tut, was er programmiert hat, unabhängig von dem Rechner, auf dem es läuft.

Wo ist sie?

Im einfachsten Fall ist das ganz uninteressant. Eine weit verbreitete Art von Java-Programmen sind nämlich Applets (ebenfalls `class`-Dateien), die in eine HTML-Seite eingebunden sind. Damit diese laufen können, muss der Browser eine JVM haben. Die großen Browser (InternetExplorer, Netscape-Navigator, Mozilla) beinhalten eine JVM, um die man sich nicht kümmern muss, weil sie automatisch aufgerufen wird, wenn der Browser auf eine HTML-Seite trifft, die ein Applet beinhaltet.

Will man jedoch auch Java-Programme starten, die wie unser `Beispiel` keine Applets sind oder ist man mit der JVM des Browsers nicht zufrieden[6], so macht das gar nichts. Bei der oben genannten Installation wurde nämlich auch die neueste Sun-JVM installiert. Diese ersetzt zum einen automatisch die JVM des Browsers und steht zum anderen im gleichen Verzeichnis zur Verfügung wie der Compiler, bei uns also unter `c:\Programme\java\bin\java.exe`

[3] Ab Windows2000 erübrigt sich `doskey`, in Linux sowieso.
[4] Unter Linux muss auch `javac` genau so, also klein, geschrieben werden.
[5] Wenn es mehrere sind, verpackt man sie meist in eine `jar`-Datei (siehe Abschnitt 5.11).
[6] Bei der Microsoft-JVM wurde oft unterstellt, dass sie bewusst Mängel beinhalte, um die Java-Technologie als schlecht funktionierend darzustellen.

Wie ruft man sie auf?

Unser Programm ist fertig übersetzt als `c:\Programme\test\Beispiel.class` und kann nun gestartet werden mit `c:\Programme\java\bin\java.exe Beispiel`. Die Dateinamenserweiterung `.class` darf nicht mit angegeben werden. Dabei ist `java.exe` der Startbefehl für die JVM. Diese nimmt dann als Argument den Namen (ohne Erweiterung) der `class`-Datei entgegen. Die JVM muss im gleichen Verzeichnis gestartet werden, in dem sich das Java-Programm `Beispiel.class` befindet. Aber das ist ja beim aktuellen Zustand der Konsole kein Problem.

Übrigens kann man GUI-Programme auch mit `javaw` statt mit `java` starten. Die Eingabeaufforderung wartet dann nicht auf das Ende der JVM sondern kehrt sofort vom Aufruf zurück[7]. Dies ist dann von Vorteil, wenn man die JVM gar nicht aus einer Konsole starten will, sondern z. B. aus einer Batch-Datei. Die Konsole erscheint dann erst gar nicht, während sie mit `java` so lange sichtbar ist, wie das Programm läuft.

1.6 Verteilen von fertigen Programmen

Wenn man ein Projekt vollendet hat und der Meinung ist, es könnte jemanden interessieren, dann muss man sich leider noch damit auseinander setzen, dass nicht auf allen Rechnern eine aktuelle Java-Umgebung installiert ist. Das führt zu zwei nicht zu unterschätzenden Problemen:

- Die Erfahrung zeigt zwar, dass fast alle Schüler einen eigenen Rechner mit Windows-Betriebssystem besitzen oder wenigsten Zugang zu einem solchen haben. Von denen, die auch einen Internetzugang haben, nimmt es aber fast niemand auf sich, die ganze J2SE herunterzuladen. Es ist deshalb unerlässlich, einen Klassensatz an CDs anzufertigen und zu verteilen. Für die Schüler eines Informatikkurses tut man das natürlich gleich zu Beginn des Kurses. Ist die Anwendergruppe jünger (wie z. B. bei einem Übungsprogramm Bruchrechnen), so sollte man auch die Installation in der Klasse vorführen oder die Eltern mit einbinden.

  Das Verteilen der J2SE birgt ein lizenzrechtliches Problem. Es handelt sich zwar um kostenlose Software, jedoch muss jeder Nutzer die Nutzungsbedinugnen akzeptieren, was nur beim Download von der *Sun*-Site möglich ist. Die Firma *Sun* zeigt sich aber nach meinen Erfahrungen kulant, wenn es um schulische Zwecke geht. Eine E-Mail sollte genügen.

- Auch der Aufruf der JVM von der Befehlszeile ist nicht jedermanns Sache. Wenn möglich, sollte das Programm also die Form eines Applets haben, das in eine HTML-Seite eingebunden ist. Die Arbeit wird dann nämlich vom Browser erledigt. Wenn ein Applet nicht möglich ist, sollte man wenigstens eine Batch-Datei beilegen, die durch Doppelklick gestartet werden kann. Der Aufruf der JVM wird dadurch vor dem Anwender versteckt. Auf diese Problematik wird in Kapitel 5.11 noch näher eingegangen.

1.7 Alles in einem: NetBeans

NetBeans (ehemals Forté) ist eine sehr komfortable Entwicklungsumgebung, die in ihrer jeweils aktuellsten Version kostenlos von `www.netbeans.org` herunter geladen werden kann. Die Möglichkeiten sind überwältigend, was sich aber zuerst einmal in einer Größe von knapp 100 MB auf der Festplatte niederschlägt. Weitere Voraussetzungen sind eine vollständige Installation der J2SE (wie in den vorherigen Abschnitten besprochen) und ein Rechner mit mindestens 500 MHz und 128 MB Speicher.

Verwendungsmöglichkeiten

NetBeans bietet eine umfangreiche Funktionalität nicht nur für die Programmierung in Java, sondern auch für alles was damit zusammenhängt. Man kann damit problemlos den ganzen Unterricht der 11. Klasse bestreiten:

[7] In Linux gibt es kein `javaw`. Prozesse können mit `&` in den Hintergrund geschickt werden.

- Die Baumstruktur der Projektverwaltung steht auch für HTML zur Verfügung mit Syntaxhervorhebung und -vervollständigung. Im Kontextmenü hat man dann mit View die Möglichkeit, die Datei von einem Browser anzeigen zu lassen.
- Legt man das HTML-Projekt in einem sogenannten Webmodul an, kann man dem Browser die Seiten auch mittels eines Servers schicken (nicht, wie oben, als Datei von der Festplatte). Der dafür nötige Webserver (der selber natürlich auf die Festplatte zugreift) ist Teil des integrierten Tomcat-Servers, der automatisch gestartet wird, wenn man im Kontextmenü Execute wählt. Damit können dann auch Formularinhalte verarbeitet werden.

  Die Übertragung erfolgt wie üblich über das HTTP-Protokoll. Dieses wird aufgezeichnet von einem integrierten HTTP-Monitor. Damit kann man sich jederzeit anschauen, was zu welcher Zeit, in welcher Reihenfolge vom Server zum Client und umgekehrt übertragen wurde. Das Protokoll wird klar und anschaulich.

  Der Vorteil dieser Herangehensweise ist, dass man auf dem eigenen Rechner ohne weitere Installation eines vollständigen WAMP- oder LAMP-Pakets die benötigte Funktionalität bekommt. Damit können Schüler dann auch zu Hause problemlos üben. Eine genauere Beschreibung dieser sehr interessanten Möglichkeiten würde über den Rahmen dieses Textes hinaus gehen. Lesen Sie dazu bitte die Datei `//code/tomcat/readme.txt` und nutzen Sie die ausgearbeiteten Beispiele.
- Natürlich hat man Syntaxhervorhebung auch in Java-Dateien. Die erzeugten Texte werden auf Tastendruck kompiliert oder ausgeführt. Während man Texte eintippt, läuft im Hintergrund eine Dauerkompilation, die Syntaxfehler farbig unterringelt. Außerdem wird Syntaxvervollständigung angeboten, die – während man aus den zur Verfügung stehenden Methoden und Attributen auswählt – gleich noch die Hilfetexte einblendet. Auch die Debugging-Möglichkeiten sind fesselnd.

Installation und erster Start

Die Installation ist sehr einfach: es handelt sich um eine riesige, selbstentpackende Programm-Datei, die lediglich einmal gestartet werden muss. Man macht nichts falsch, wenn man alle danach auftretenden Fragen bejaht. Falls das Installationsprogramm nach dem Java-Verzeichnis fragt, geben Sie bitte das Verzeichnis des J2SE an. Ein anderes Java-Verzeichnis würde nicht funktionieren und existiert wahrscheinlich sowieso nicht auf dem Rechner. Gegen Ende der Installation kann man auswählen, welcher Webbrowser verwendet werden soll. Da ist der interne Browser *nicht* empfehlenswert, weil er keine Applets ausführen kann (für die Hilfetexte reicht er aber allemal). Wählen Sie dort am besten den IE, Netscape oder Mozilla.

Mit NetBeans arbeitet man immer in einem Projekt. Dieses legt man einmal am Anfang an (Menü Project – Project Manager). Danach hat man im Fenster Filesystems eine neue, leere Verzeichniswurzel. Alle Verzeichnisse, die man hier einhängt, liegen danach im `classpath`; wir hängen aber nur eines ein. Das reicht gewöhnlich auch für große Projekte.

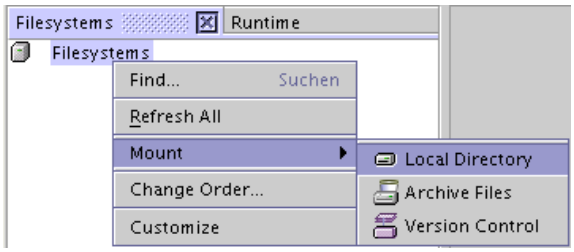

Abbildung 1.2: Einhängen eines lokalen Verzeichnisses

Abbildung 1.2 zeigt als Ergebnis eines Rechtsklicks das Kontextmenü zum Mount eines Local Directory. Mit dem nachfolgenden Dialog navigiert man zu dem Verzeichnis, in dem die Dateien des Projekts abgespeichert werden sollen. Für Packages dürfen später Unterverzeichnisse angelegt werden, aber das werden wir nicht brauchen.

Wenn Sie das weiter oben schon verwendete Verzeichnis eingebunden haben, finden Sie in Filesystems die Datei `Beispiel.java`. Der Inhalt wurde sofort analysiert und wird beim Aufklappen des Knotens sichtbar. Die Einzelheiten im Umgang mit NetBeans ergeben sich bei einigem Herumprobieren von selbst. Für den Anfang erwähnenswert ist lediglich, dass man eine neue Datei am einfachsten erzeugt durch Rechtsklick auf den Verzeichnisknoten und Auswahl von New - All Templates - Java Classes - Empty Java File. Mit einem Tastendruck auf F9 wird kompiliert, mit F6 wird das Programm ausgeführt.

Zum Schluss noch ein Detail zur Dokumentation. Wenn man innerhalb eines Quelltextes die Taste Alt-F1 drückt, so soll die Hilfe zu dem Element angezeigt werden, das sich unter dem Cursor befindet. Das ist ein sehr allgemein angelegtes Konzept, das auch auf Hilfetexte aus dem eigenen Projekt zugreift. Alle noch nicht bekannten Hilfetexte – und die Standard-Java-Hilfe gehört dazu, weil sie nicht Bestandteil der J2SE-Installation ist – müssen einmal für das aktuelle Projekt bekannt gemacht werden. Dazu ruft man im Menü Tools den Javadoc Manager auf (siehe Abbildung 1.3).

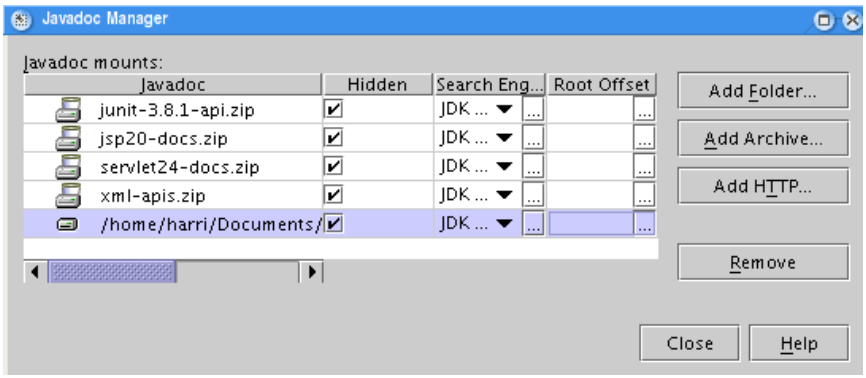

Abbildung 1.3: Verfügbarmachen der Dokumentation

Nachdem man mit Add Folder das Verzeichnis aufgesucht hat, in das man die Java-Dokumentation ausgepackt hat (in den bisher genannten Schritten wäre das `c:\Books\java\docs\api`), ist die Hilfe im aktuellen Projekt auf komfortable Weise verfügbar. Es wird zu allen bekannten Klassen schon beim Schreiben des Quelltextes Syntaxvervollständigung und ein kleiner Einblick in den Hilfetext angeboten.

Es ist recht ärgerlich, bei jedem neuen Projekt die Dokumentation einhängen zu müssen. Wenn man es aber gleich beim ersten Projekt richtig anstellt, wird die Standarddokumentation bei jedem weiteren automatisch eingehängt. Und das geht so:

- Einmal einhängen wie oben beschrieben.
- Im Menü Tools – Options – IDE Configuration – System – Filesystems Settings den entsprechenden Eintrag suchen und markieren.
- Die Spalte rechts davon ist grau und bietet oben im Kopf die Möglichkeit der Erweiterung durch Klick auf «.
- Nach dem Aufklappen erkennt man, dass der Eintrag *Project*-weit gilt. Klick auf *User* (oder als Administrator auf *Default*) macht den Eintrag allgemein gültig.
- Das Options-Fenster wieder schließen. In Zukunft ist die Standarddokumentation immer bekannt.

2 Java als erste Programmiersprache

Dieses Kapitel listet das Übungsmaterial auf, das sich auf der CD befindet und größtenteils im weiteren Text beschrieben wird. Es soll eine Orientierung darüber bieten, welche Themen man sich als Lehrer der Reihe nach aneignen sollte, um mit Java nicht in eine unterrichtliche Sackgasse zu geraten. Die gezeigten Beispiele gehen im Detail gelegentlich über das hinaus, was man im Unterricht tun will und sind deshalb hauptsächlich als Anregung zu verstehen. Es wird davon ausgegangen, dass NetBeans installiert ist (wie in Kapitel 1.7 beschrieben), was aber nicht heißen soll, dass alle Programme eine graphische Oberfläche (GUI) haben müssen. Jedenfalls können die angegebenen Verzeichnisse in ein jeweils neues Netbeans-Projekt eingebunden werden und bilden dort eine zusammengehörige Menge von Dateien, die leicht bearbeitet werden können.

2.1 Sprachgrundlagen in der 11. Klasse

Programmieren lernt man durch Programmieren. Beim Anfängerunterricht geht es erst einmal um die benötigten Grundlagen, also Sequenz, Entscheidung, Wiederholung und Verschachtelung, sowie Funktionsaufrufe. Hier kann man problemlos die Programme in Java umschreiben, die man früher in Pascal geschrieben hat. Beachten Sie bitte die Beispiele auf der CD:

- In `//code/tomcat` wird gezeigt, wie man NetBeans schon im ersten Halbjahr der 11. Klasse vorteilhaft einsetzen kann, wenn die Schüler noch gar nicht programmieren.
- In `//code/noobj` finden Sie kleine Beispiele, die ganz ohne Objekte auskommen, etwa so, wie in klassischem Pascal. Für die Eingabe von der Konsole liegt die Hilfsklasse `LineInput` bereit. Töne können leicht über die MIDI-Schnittstelle erzeugt werden (siehe `Tonfolge.java`), was wegen der Klangqualität gewöhnlich den ganzen Kurs begeistert. Der Quelltext solcher Hilfsklassen muss übrigens nicht bekannt gemacht werden, das Vorhandensein der `class`-Dateien reicht aus. Dass man (fremde) Routinen aufruft, gehört ohnehin zu den wichtigsten Konzepten der Programmierung.
- Die Verzeichnisse `//code/nimm` und `//code/netbeans` enthalten Beispiele zu einfachen Programmen mit GUI-Oberfläche, wie man sie auch mit Delphi erstellt. Wie man mit NetBeans vorgeht, wird in Kapitel 3 gezeigt. Das Kapitel sollten Sie lesen, bevor Sie GUI-Programme erstellen. Besonders hinzuweisen ist auf die Möglichkeit, GUI-Programme als Applet auszuführen (z. B. `EinmalEinsApplet`). Damit kann das Programm in eine HTML-Seite eingebunden werden.

2.2 Objektorientierung in der 12. Klasse

Ab der 12. Klasse werden nicht nur Objekte verwendet, sondern Klassen selber erzeugt. Die Gedankengänge die hier nötig sind, werden in Übungen vorgestellt:

- Im Verzeichnis `//code/obj` befinden sich einige kleine Beispiele mit Objekten.
- Ein kleines Projekt ohne Listener wird in Kapitel 4 vorgestellt und befindet sich ausgearbeitet im Verzeichnis `//code/sortdemo`.
- Ein größeres Projekt mit Listenern wird in Kapitel 5 besprochen. Die zugehörigen Verzeichnisse sind `//code/krug1` und `//code/krug2`.
- Schließlich befindet sich noch ein umfangreiches Projekt, an dem eine größere Zahl von Rechnern zusammen arbeiten kann, im Verzeichnis `//code/rmi`. Der begleitende Text ist das Kapitel 6

3 Übung: Arbeiten mit NetBeans

In diesem Abschnitt soll exemplarisch gezeigt werden, wie man ein altmodisches kleines Pascal-Programm auf Konsolen-Ebene umwandelt in ein altmodisches kleines Java-Programm auf Konsolen-Ebene. Danach soll dieses dann optisch aufbereitet werden zu einem GUI-Programm. Dabei wollen wir die Entwicklungsumgebung NetBeans verwenden und sehen, dass sie einen ähnlichen Bedienungskomfort bietet wie Delphi.

Wir nehmen als Beispiel ein (bei Schülern wenig) bekanntes Nimm-Spiel. Dabei liegt eine bestimmte Anzahl von Objekten auf dem Tisch, von denen die beiden Spieler abwechselnd mindestens eines und höchstens eine vorher vereinbarte Anzahl weg nehmen. Gewonnen hat, wer das letzte Objekt weg nimmt.

Die Motivation dieses Spiels erwächst aus der Tatsache, dass die Schüler eine ganze Weile brauchen, bis sie die Strategie durchschauen und danach sehr stolz sind, wenn sie eine „Maschine gebaut“ haben, die optimal spielt und gegen einen unbedarften Gegner fast ausnahmslos gewinnt. Dass das Spiel in der Konsole sehr altmodisch aussieht, stört da nicht.

3.1 Das einfache Nimm-Spiel in Pascal

Da bei so kleinen Programmen die Ähnlichkeit zwischen Pascal und Java sehr augenscheinlich ist, sei hier auch einmal das Pascal-Programm aufgeführt. Wie man größere Pascal-Projekte systematisch in Java-Klassen umwandelt, ist in Anhang D dargelegt.

```
program Nimm;
var k, m, n, s : integer;
begin
  write('Wie viele Steine am Anfang? ');
  readln(k);
  write('Zu nehmen zwischen 1 und ');
  readln(m);
  write('0 - Sie beginnen   1 - Ich beginne: ');
  readln(s);
  repeat
    if s=0 then
      repeat
        write('Nehmen Sie 1 bis ',m,' Steine: ');
        readln(n);
      until (n>0) and (n<=m)
    else begin
      n:=k mod (m+1);
      if n=0 then n:=1;
      writeln('Ich nehme ',n,' Stein(e).');
    end;
    k:=k-n; if k<0 then k:=0;
    writeln('Es bleiben ',k,' Stein(e).');
    writeln;
    s:=1-s;
  until k<=0;
  if s=0 then writeln('Ich gewinne!')
  else writeln('Sie gewinnen.');
  readln;
end.
```

3.2 Das einfache Nimm-Spiel in Java

In Java kann man statt `write` oder `writeln` die Befehle `System.out.print` bzw. `System.out.println` verwenden. Diese sind zwar im Grunde nur für die Fehlersuche vorgesehen und haben in einem fertigen

Programm nichts zu suchen, so lange man aber Programme in der Konsole laufen lässt, erfüllen sie ihren Zweck.

Anders sieht es da schon mit dem Pendant zu `readln` aus. Das gibt es nämlich nicht, so dass man selber tätig werden muss. Die verwendete Klasse `LineInput` ist ein ausreichend nützliches Eigenprodukt, das fast das gleiche leistet. (Der Quelltext ist auf der CD im Verzeichnis `//code/pas2java/LineInput.java`). Schüler müssen den Inhalt nicht kennen, es reicht vielmehr, einfach die `class`-Datei im aktuellen Verzeichnis zu haben.

```
public class Nimm{
  public static void main(String[] args){
    System.out.print("Wie viele Steine am Anfang? ");
    int k=LineInput.readInt();
    System.out.print("Zu nehmen zwischen 1 und ");
    int m=LineInput.readInt();
    System.out.print("0 - Sie beginnen   1 - Ich beginne: ");
    int s=LineInput.readInt();
    int n;
    do{
      if(s==0)
        do{
          System.out.print("Nehmen Sie 1 bis "+m+" Steine: ");
          n=LineInput.readInt();
        } while(n<1 || n>m);
      else{
        n=k%(m+1);
        if(n==0) n=1;
        System.out.println("Ich nehme "+n+" Stein(e).");
      }
      k=k-n; if(k<0) k=0;
      System.out.println("Es bleiben "+k+" Stein(e).\n");
      s=1-s;
    } while(k>0);
    if(s==0) System.out.print("Ich gewinne!");
    else System.out.print("Sie gewinnen.");
    LineInput.readString();
  }
}
```

Man sieht: Die Unterschiede zu Pascal sind minimal:

- Blöcke werden nicht mit `begin` und `end` gebildet sondern mit `{` und `}`.
- Alle Anweisungen enden mit dem Semikolon, auch `if`-Zweige, auf die ein `else` folgt.
- Statt `repeat-until` verwendet man `do-while`, was aus Gründen der englischen Sprache zur Negierung der Abbruchbedingung führt.
- Variable deklariert man bei Bedarf, wobei man darauf achtet, dass solche, die innerhalb eines Blocks deklariert werden, auch nur innerhalb des Blocks zugreifbar sind (`n` musste deshalb relativ weit oben deklariert werden).

3.3 Eine Oberfläche mit NetBeans

Nun wollen wir die Annehmlichkeiten von NetBeans nutzen, um relativ schnell eine GUI-Oberfläche zu erstellen. Dazu muss, wie in Abschnitt 1.7 beschrieben, in Filesystems das Programm-Verzeichnis eingehängt werden.

Aus dem Kontextmenü unserer Verzeichniswurzel wählen wir New – JFrame Form. Beim folgenden Wizard geben wir als Name `NimmFrame` an und klicken gleich auf Finish. Damit erhalten wir die Möglichkeit, mit der Maus das Aussehen des neuen Programms zu gestalten. Alles was wir tun, schlägt sich in automatisch generiertem Quelltext in `NimmFrame.java` nieder. Einige zusätzliche Informationen, die der Editor nicht verlieren soll, werden in `NimmFrame.form` gespeichert. (Das Programm ist auch ohne diese Datei kompilier- und ausführbar, aber der Editor würde „vergessen“, was seine Eintragungen bedeuten). Wir arbeiten mit den folgenden vier Bereichen:

1. Den meisten Platz nimmt in der Mitte das Formular ein. Den Mausmodus-Anzeiger lassen wir die ganze Zeit in der Stellung „Pfeil“.
2. Rechts oben findet man alle verwendbaren Komponenten in der Palette. Klickt man eine solche Komponente an, so wird diese als nächstes verwendet.
3. Gleich darunter hat man im Inspector einen Komponentenbaum, der am Anfang nur unseren `JFrame` beinhaltet und seinen zugeordneten Standard-Layoutmanager, also `BorderLayout`. Das ändern wir aber sofort durch Rechtsklick und Auswahl von `AbsoluteLayout`. Klickt man an eine Stelle des Formulars, so erscheint dort die vorher ausgewählte Komponente. (Das geht aber nur, weil wir `AbsoluteLayout` gewählt haben. Dazu gleich mehr.)
4. Rechts ganz unten bekommt man in Properties die Eigenschaften und möglichen Events der gerade aktuellen Komponente angezeigt und kann auf diese auch einwirken.

Der Spieler legt anfangs die Anzahl der vorhandenen Steine fest und wie viele bei jedem Zug maximal genommen werden dürfen. Dann drückt er den Start-Button und erhält im großen Textbereich einen einzeiligen Hinweis, wie viele Steine vorhanden waren und wie viele der Rechner davon weg genommen hat (der Hinweis kann rein symbolisch geschehen). Er kann nun selber eine Zahl eingeben und Enter drücken. Nachdem in einer weiteren Hinweiszeile auch seine Steine entfernt wurden, kommt automatisch der Rechner wieder an die Reihe. Am Ende wird noch vermerkt, wer gewonnen hat. Abbildung 3.1 zeigt das Aussehen des Fensters.

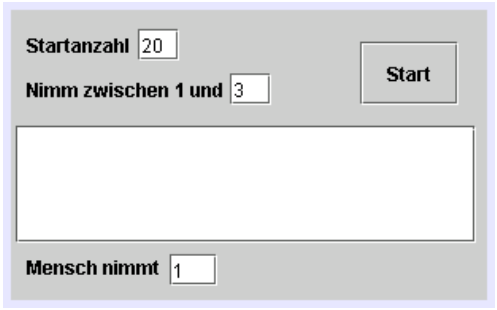

Abbildung 3.1: Aufbau des GUI

Die Oberfläche besteht aus 3 `JLabels`, 3 `JTextFields`, 1 `JButton`, 1 `JScrollPane` und 1 `JTextArea`. Die `JScrollPane` ist so groß, wie die `JTextArea` und muss als erstes platziert werden. Die `JTextArea` kommt dann innen hinein. Dadurch ist sie bei Bedarf scrollfähig. Sollte es dabei Schwierigkeiten geben, so kann man im Inspector Komponenten auch ausschneiden und an anderer (logischer) Stelle einfügen. Versuchen Sie selber, das Aussehen so ähnlich wie in Bild 3.1 hinzubekommen. Die Ausrichtung mehrerer Elemente erreicht man durch Mehrfachauswahl (mit Hilfe der Strg-Taste) und eine entsprechende Einstellung im Abschnitt Layout der Properties.

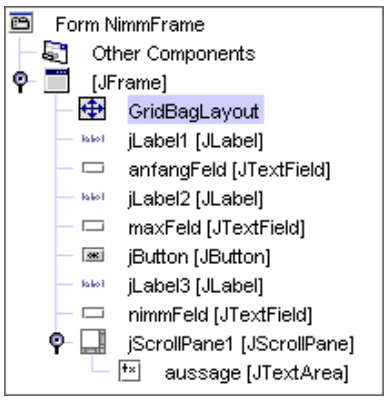

Abbildung 3.2: Baumansicht des GUI-Aufbaus im Inspector

Außer den Komponenten, die im Programm nicht mehr angesprochen werden, sollten Sie aussagekräftige Namen vergeben durch Rechtsklick im Inspector und Auswahl von Rename. Wie Abbildung 3.2 zeigt, sind das `anfangFeld`, `maxFeld`, `aussage` und `nimmFeld`.

In der Komponente aussage wurde columns auf 40, editable auf false und font auf die Festbreitenschrift DialogInput 12 Plain gesetzt.

(Übrigens: Other Components bietet eine zweite Baumstruktur an, in der man unsichtbare Komponenten ablegen kann. Solche müssen dann nicht – wie in Delphi – an sinnloser Stelle mitten im Formular herum liegen. Wir machen von dieser Möglichkeit hier keinen Gebrauch.)

Als letzten Schritt sollten wir uns noch ein paar Gedanken zum Layoutmanager machen. Ein gut gemachtes Java-Programm sollte sich nicht auf absolute Maße von Komponenten verlassen. Wenn der Anwender die Systemschrift größer stellt, weil er schlecht sieht, dann müssen die Komponenten größer werden dürfen ohne das Programm unbedienbar zu machen. Seit jeher werden in Java deshalb Layoutmanager verwendet, die eine bestimmte anordnungslogische Strategie verfolgen und deshalb leider nicht ganz intuitiv verwendbar sind.

Wir wollten uns nicht um `BorderLayout` oder ähnliches kümmern, sondern intuitiv (mit der Maus) arbeiten und haben deshalb den GUI-Editor verwendet, mit dem man die Komponenten einfach platziert und skaliert, wie man es in Delphi zu schätzen gelernt hat. Dazu mussten wir `AbsoluteLayout` verwenden (sonst hätten wir uns ziemlich über das Verhalten des Editors gewundert!). Das erfüllt aber nicht den Standard. Ein sehr leistungsstarker Layoutmanager, der optisch fast das gleiche leistet (und anordnungslogisch noch viel mehr), heißt `GridBagLayout` und ist sehr unangenehm zu programmieren.

Erfreulicherweise bietet NetBeans hier die unerhörte Möglichkeit, das eine in das andere umzuwandeln und das ohne große Einbußen in der gewünschten Anordnung. Man muss nur im Inspector auf `AbsoluteLayout` rechts-klicken und im Kontextmenü `GridBagLayout` auswählen. Um zu erkennen, welche Arbeit einem gerade abgenommen wurde, durchsuche man den Quelltext nach `GridBagConstraints`.

3.4 Code implementieren mit NetBeans

Bis hier her hat NetBeans im Hintergrund Quellcode produziert, der im Programm nachher das tut, was wir durch Aktionen an seiner Oberfläche graphisch hergestellt haben. Damit wir diesen (im folgenden Autocode genannten) Quellcode nicht versehentlich zerstören, wurden gewöhnliche einzeilige Kommentare eingefügt, die alle mit einer bestimmten Buchstabenfolge beginnen und dadurch als „nicht für die Augen des Programmierers bestimmt“ markiert gelten. NetBeans zeigt diese speziellen Kommentare nicht an, färbt aber die so markierten Bereiche im Editor blau und verhindert deren Änderungen durch den Programmierer.

Der Autocode hat Vorteile, insbesondere didaktische:

- Es gibt keine versteckten Dateien mit compiler-relevanten Inhalten. Der Quelltext in der `java`-Datei ist vollständig. (Einige Einstellungen, die NetBeans für das „eigene Verständnis“ braucht, werden in der `.form`-Datei gespeichert. Für die Compilierung des Programms ist diese Datei aber irrelevant.)
- Auf Grund der gefärbten Darstellung ist immer klar, welcher Teil des Quelltextes vom Programmierer selber stammt.
- Mit Hilfe des Autocodes kann man neue Techniken lernen, in unserem Beispiel etwa, wie man eine `JTextArea` in eine `JScrollPane` bindet.
- Beim Durchschauen des Autocodes fallen dem Interessierten viele Möglichkeiten ein, wie der Code effizienter gestaltet werden kann, wenn man – im Gegensatz zur Entwicklungsumgebung – weiß, welchen Zweck er hat.

Nun kommt aber auch der Moment, wo wir selber programmieren müssen: Nach dem Aussehen kommt die Funktion!

Wenn der Spieler auf den Start-Button drückt, muss das Spiel initialisiert werden. Eine Methode hierfür wird angelegt, wenn man im Form Editor den Button markiert und dann in den Properties im Bereich Events den Event `actionPerformed` anklickt. Durch Drücken von *Enter* gelangt man in den Quelltext. Warum die Methode einen so seltsamen Namen hat (und nicht einfach `actionPerformed` heißt), kann man im Autocode nachvollziehen[1].

```
private void startButtonActionPerformed(java.awt.event.ActionEvent evt){
  nochda=Integer.parseInt(anfangFeld.getText());
```

[1] Es werden *anonymous classes* verwendet.

```
    if(nochda<1) nochda=1;
    maxNimm=Integer.parseInt(maxFeld.getText());
    if(maxNimm<1) maxNimm=1;
    anfangFeld.setEditable(false);
    maxFeld.setEditable(false);
    steine=new char[nochda];
    for(int i=0; i<nochda; i++) steine[i]='-';
    aussage.setText(anzeigeEvtl(""));
    computerZieht();
  }
```

Das Innere der Methode muss man natürlich selber schreiben. Die Variablen `nochda`, `maxNimm` und `steine` sind Attribute, die man selber anlegen muss. `nochda` merkt sich, wie viele Steine noch nicht weggenommen wurden (am Anfang alle). `maxNimm` ist der Zahlenwert aus dem entsprechenden `JTextField`.

Das Umwandeln der Strings in Zahlenwerte übernimmt die Methode `Integer.parseInt`. Das kann schief gehen, wenn etwa der Inhalt des `JTextFields` gar keine vernünftige Zahl ist. In diesem Fall wird eine `NumberFormatException` erzeugt, was wir hier der Kürze halber aber nicht berücksichtigen (obwohl es mit einiger Wahrscheinlichkeit vorkommt[2]).

Beide Textfelder können während des Spiels (wegen `setEditable(false)`) nicht mehr verändert werden. `steine` repräsentiert alle Zeichen, die für Steine angezeigt werden: Ein ‚-' bedeutet, dass der entsprechende Stein noch nicht weggenommen wurde, ‚#' wird verwendet, um anzuzeigen, dass der Computer ihn genommen hat und ‚+' dass der Mensch ihn genommen hat.

Die beiden Methoden `computerZieht` und `anzeigeEvtl` wurden nach Bedarf selber programmiert. Letztere hat den Seiteneffekt, dass sie die beiden Textfelder wieder editierbar macht, wenn das Spiel zu Ende ist. Das ist kein guter Stil aber kurz.

```
  private void computerZieht() {
    if(nochda==0) return;
    int n=nochda%(maxNimm+1);
    if(n==0) n=1;
    nochda-=n;
    for(int i=0; i<n; i++) steine[nochda+i]='#';
    aussage.append(anzeigeEvtl("Ich habe gewonnen!"));
    nimmFeld.requestFocus();
    nimmFeld.selectAll();
  }
  private String anzeigeEvtl(String wennkeine){
    String s=nochda+" ";
    for(int i=0; i<steine.length; i++) s=s+steine[i];
    s=s+"\n";
    if(nochda==0){
      s=s+wennkeine+"\n";
      anfangFeld.setEditable(true);
      maxFeld.setEditable(true);
    }
    return s;
  }
```

Wenn der Anwender am Zug ist und im untersten `JTextField` *Enter* drückt, muss eine weitere Methode aufgerufen werden. Sie wird auf die gleiche Weise angelegt, wie der Start-Button (Rechts-Klick auf den Event `actionPerformed`).

```
  private void nimmFeldActionPerformed(java.awt.event.ActionEvent evt){
    if(nochda==0) return;
    int n=Integer.parseInt(nimmFeld.getText());
    if(n<1) n=1;
    if(n>maxNimm) n=maxNimm;
    if(n>nochda) n=nochda;
    nochda-=n;
    for(int i=0; i<n; i++) steine[nochda+i]='+';
    aussage.append(anzeigeEvtl("Du hast gewonnen."));
```

[2] In früheren Versionen war diese Exception „abfangpflichtig". Da sich aber viele Programmierer beschwert hatten über den umständlichen aber unvermeidlichen Code, wurde die `NumberFormatException` in die Ahnenreihe von `RuntimeException` gestellt, so dass dem Programmierer jetzt freigestellt ist, ob er ihr Auftreten für wahrscheinlich hält und abfängt. (Mehr zu diesem Thema in Abschnitt A.6.)

```
        computerZieht();
    }
```

3.5 Abwägung der Vor- und Nachteile

Entwicklungsumgebungen wie NetBeans oder Delphi erleichtern dem Anfänger die Erstellung von GUI-Programmen, was viel Motivation mit sich bringt. Möchte man aber eigene Komponenten einsetzen, die von vorhandenen abstammen und sich in Einzelheiten anders verhalten, so kann man diese vielleicht einigermaßen zügig programmieren (wie etwa im Krugspiel in Kapitel 5.4) aber die Einbindung in die GUI geht nicht einfach durch Ziehen mit der Maus. Die selbst programmierte Komponente ist nämlich nicht ohne weiteres in die Komponentenleiste zu bekommen. Das gilt sowohl für Delphi als auch für NetBeans[3].

In Delphi behilft man sich üblicherweise dadurch, dass man eben nicht ernsthaft eigenständige Klassen für die Komponenten erzeugt, sondern das Programm als *die Komponente* betrachtet. Damit ist es dann nicht so schlimm, wenn man sich bei Bedarf einen fremden `Canvas` holt und darauf herumzeichnet, so fremd ist der ja dann nicht.

Diesen Weg kann man natürlich auch mit NetBeans beschreiten, indem man die `paint`-Methode der Hauptkomponente so überschreibt, dass sie erst einmal das tut, was sie sonst auch tun würde (`super.paint(...)`) und danach mit `getGraphics` auf die Zeichenflächen der anderern Komponenten zugreift und den Rest zeichnet. Man verzichtet also bei der GUI auf ein strenges objektorientiertes Konzept, vielleicht eine lässliche Sünde bei den winzigen Programmierprojekten in der Schule, falls man sich ihrer wenigstens bewusst ist.

Auf der CD (`//code/netbeans/Nimm.java`) befindet sich noch eine hübschere Variante des Nimmspiels, die diesen (unschönen) Weg aufzeigt. Im gleichen Verzeichnis finden Sie übrigens noch weitere kleine Projekte, die in diesem Text nicht besprochen werden. Es wird z. B. gezeigt, wie man Zufallszahlen erzeugt oder einen Timer einsetzt. Lesen Sie bitte jeweils die `readme`-Dateien.

[3] Für NetBeans müsste man eine regelrechte *Java-Bean*, für Delphi eine `dll`-Datei erzeugen. Beides würde den Rahmen des Schulunterrichts sprengen.

4 Übung: Demonstration von Sortieralgorithmen

Die Veranschaulichung der Arbeitsweise von Sortieralgorithmen ist eine vielfach gelöste Aufgabe. Immer wieder aber kommt es vor, dass man eine alte Idee verbessert, was natürlich eine gewisse Programmierarbeit mit sich bringt. In dieser Übung werden deshalb einige Gedanken zusammen gefasst, die diese Arbeit für die Zukunft erleichtern. Dabei soll es nicht in erster Linie um Feinheiten der Programmierung gehen[1] als vielmehr um Überlegungen zur Aufteilung der Aufgaben unter verschiedenen Klassen. (Das Projekt befindet sich ausgearbeitet im Verzeichnis `//code/sortdemo`.)

4.1 Aufteilung des Problems in Klassen

Die Aufgaben müssen so unter den Klassen aufgeteilt werden, dass auch für die Zukunft Erweiterbarkeit erwartet werden kann. Für einen Sortieralgorithmus zu Demonstrationszwecken (nennen wir ihn abkürzend `Algorithmus`) reicht es, wenn er ganze Zahlen aufsteigend sortieren kann. Ein `Demonstrator` hingegen ist etwas, das über eine zu sortierende Zahlenmenge verfügt und Zugriffe oder Vertauschungen innerhalb dieser Menge geeignet darstellt. Dass der `Demonstrator` die Zahlenmenge besitzt und nicht der `Algorithmus` ist deswegen wichtig, weil sonst der `Algorithmus` Aktionen in der Zahlenmenge vornehmen könnte, ohne dass sie dargestellt werden, was der Idee des `Demonstrator`s widersprechen würde.

Natürlich muss jeder `Algorithmus` auf eine Zahlenmenge Einfluss nehmen können, um sie zu sortieren. Im vorliegenden Fall scheint es deshalb angemessen zu sagen, jeder `Algorithmus` besitzt einen `Demonstrator` und dieser bietet Zugriff auf die Zahlenmenge. Da einem beim Begriff `Demonstrator` gleich unterschiedlichste Ideen in den Sinn kommen, die schrecklich wenig miteinander zu tun haben, fassen wir seine Fähigkeiten nicht in einer abstrakten Vorfahrklasse zusammen, sondern in einem Interface. Beim `Algorithmus` ist die Sache einfacher: Er muss die Möglichkeit bieten, einen `Demonstrator` entgegen zu nehmen (`setDemonstrator`) und sich diesen in einem den Nachfahren zugänglichen Attribut zu merken. Außerdem muss er die Methode `sortiere` anbieten. Da `Algorithmus` selber noch nicht auf eine bestimmte Weise sortiert, sind die Klasse und die `sortiere`-Methode `abstract`.

Nachdem man sich darüber klar geworden ist, welche Methoden gebraucht werden, kann man die Dateien `Demonstrator.java` und `Algorithmus.java` in kürzester Zeit aufschreiben und compilieren. Die Funktionalität muss nämlich noch nicht implementiert werden, es reicht vielmehr zu wissen, was die Methoden am Ende tun werden. Dieses gehört auch sofort ausreichend kommentiert, und sei hier für `Demonstrator` kurz angedeutet (vgl. Bild 4.1):

- `nimmZahlen` nimmt ein `int[]` entgegen. Wird `null` übergeben, sollte der Demonstrator automatisch eine Defaultmenge benützen.
- `mischen` mischt die Zahlenmenge.
- `gibZahl` gibt die Zahl an einer anzugebenden Position der Menge zurück.
- `vertauscheZahlen` vertauscht die Zahlen an zwei anzugebenden Positionen.
- `anzahl` gibt die Größe der Zahlenmenge zurück. Ein `Algorithmus` wird stets wissen wollen, wie viele Zahlen zu sortieren sind. Er muss aber nicht derjenige sein, der sie anfangs auch übergeben hat.
- `anzZugriffe` gibt an, wie oft die Methode `gibZahl` seit dem letzten Mischvorgang aufgerufen wurde.
- `anzVertauschungen` tut das entsprechende für die Methode `vertauscheZahlen`.
- `verlangsame` bietet die Möglichkeit, den `Demonstrator` zu bremsen. (Ein `Demonstrator`, bei dem dies keinen Sinn hat, kann den Aufruf ja gegebenenfalls in einer leeren Methode ignorieren.)
- `korrekt` stellt die Zahlenmenge in ihrem aktuellen Zustand dar und gibt an, ob sie korrekt, also aufsteigend sortiert ist.

[1] Programmieraspekte werden ausführlicher in Kapitel 5 dargestellt.

- `markierePlatz` bietet einem freundlichen `Algorithmus` die Möglichkeit eine Position hervorzuheben. Anzugeben sind die Position und eine weitere Zahl, die für die Art und Weise der Hervorhebung steht. Was der `Demonstrator` damit anstellt, ist ihm überlassen. Wenn Hervorhebungen bei einem bestimmten `Demonstrator` keinen Sinn haben, kann er es auch lassen.

4.2 Implementation und Nachfahren

Damit ist ein wichtiger Zwischenschritt erreicht! Ab hier können nun nämlich mehrere Personen verschiedene Seiten des Projekts unabhängig bearbeiten. Einer kann etwa `SelectionSort` als Nachkomme von `Algorithmus` programmieren und kompilieren. Dazu muss es noch nicht einmal einen bestimmten `Demonstrator` geben. In jedem `Algorithmus` wird nämlich nur ein Feld des Typs `Demonstrator` verwendet und dieses Interface ist schon kompiliert.

Auf der anderen Seite kann jemand schnell einen `BlindDemonstrator` programmieren, der nur das nötigste tut und gar nichts demonstriert. Er muss z. B. bei `gibZahl` die richtige Zahl liefern, aber er muss daneben nichts darbieten. Dieser `BlindDemonstrator` implementiert alle Methoden, die in `Demonstrator` aufgelistet sind und *ist* damit ein `Demonstrator`, der an einen `Algorithmus` übergeben werden kann.

Mit diesen beiden nicht abstrakten Klassen kann man dann erstmals ein kleines Testprogramm schreiben, wie etwa `SortKonsole`, das einen `BlindDemonstrator` erzeugt, ihn einem `SelectionSort` übergibt und diesen auffordert zu sortieren. (Da `SortKonsole` keinen Zugriff mehr auf `BlindDemonstrator` braucht, nachdem dieser an einen Algorithmus übergeben wurde, fehlt in Bild 4.1 der Besitz anzeigende Pfeil zwischen den beiden.)

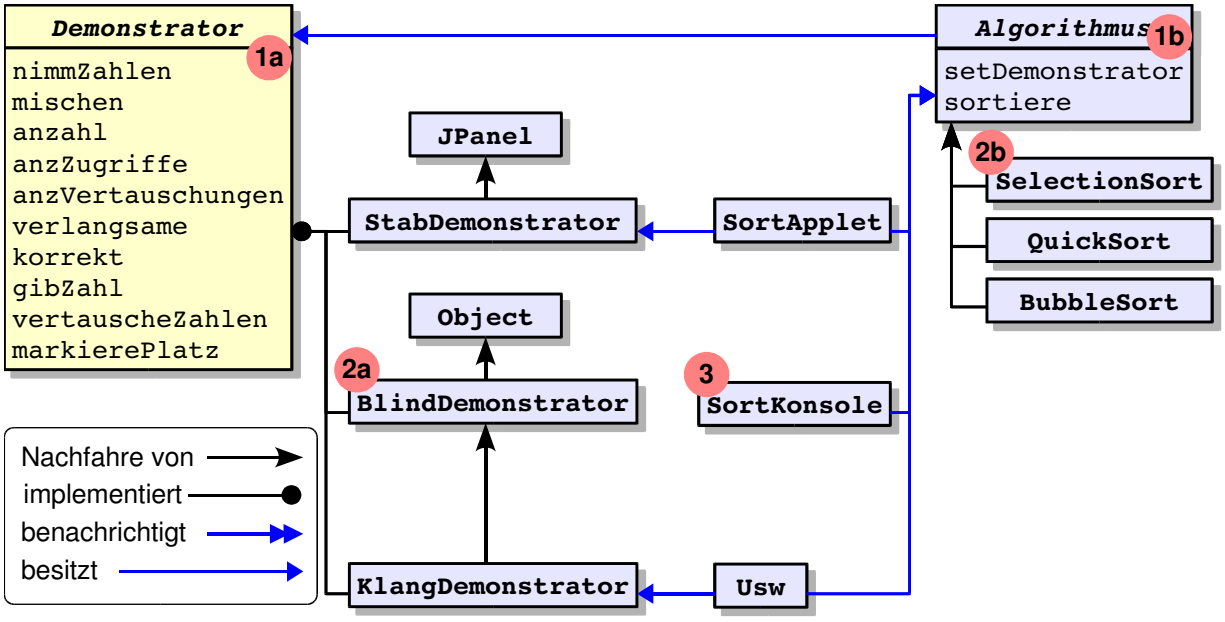

Abbildung 4.1: Die Klassen zu SortDemo

4.3 Verfeinerung und Erweiterung

Nachdem man nun ein zusammen arbeitendes Gerüst von Klassen geschaffen hat, kann man sich anspruchsvollen Erweiterungen hingeben. Der erste ordentlich aussehende `Demonstrator` heißt `StabDemonstrator` und soll Nachfahre eines `JPanel`s sein, einer Komponente also, die man in ein GUI-Programm einbinden kann. Die Zahlenmenge wird als verschieden lange senkrechte Striche dargestellt. Beim Zugriff auf eine Zahl oder bei Vertauschung werden sie kurz eingefärbt.

An dieser Stelle wird klar, warum es klug war, `Demonstrator` nicht als abstrakten Vorfahren sondern als Interface einzurichten. In Java gibt es keine Mehrfachvererbung[2] und wir wollen unsere Klasse von

[2] Die Entwickler nennen gute Gründe für diese Designentscheidung.

`JPanel` ableiten, um die ganze Funktionalität zu erben!

Ein Programm, das diese neue Komponente zur Anzeige verwendet, ist `SortApplet`. Im gleichen Vezeichnis befindet sich noch ein etwas ausgefallener `KlangDemonstrator`, der einen Sortiervorgang „musikalisch“ darstellt, was aber ein gewisses tonales Talent vom Zuhörer verlangt. Wenn Sie den ausprobieren wollen, können Sie leicht in `SortKonsole` statt des `BlindDemonstrators` den `KlangDemonstrator` verwenden. Alles arbeitet (objektorientierterweise) problemlos zusammen. Die Dateien `Usw` und `BubbleSort` sind Erweiterungen, die dem Leser überlassen werden.

5 Übung: Ein Spiel mit Krug-Objekten

In diesem Kapitel werden wir das Verhalten von Krügen nachbilden und damit ein Umschütt-Spiel programmieren. Im Gegensatz zu Kapitel 4 wird hier auf viele javaspezifische Feinheiten eingegangen: Die wichtigsten Methoden der Klasse `Object`, programmatische Erzeugung des GUI, Erläuterung des Listener-Konzepts, Bündelung mehrerer Klassen in einer `jar`-Datei, Erstellung eines Applets.

Die Mathematik und der Stammbaum der Klasse `Krug` sind sehr einfach. Da jedoch nicht auf einen mächtigen Vorfahren zurück gegriffen sondern alles selber programmiert wird, können Schüler mit diesem Beispiel schon recht deutlich erfahren, worauf man bei objektorientierter Denkweise achten muss. Die Klasse wird *vorerst* nicht mit dem Ziel entworfen, einen bestimmten Dienst in einem größeren Zusammenhang zu leisten. Wenn wir sie ordentlich entwerfen, wird sie solche Dienste automatisch tun können. Fehler und Mängel in der Entwurfsphase sind aber auch lehrreich und nachher ganz leicht zu verbessern.

Während der Programmierphase bekommt die Klasse eine eigene `main`-Methode, damit sie allein ausführbar ist. Wenn sie ausreichend getestet ist, wird diese `main`-Methode wieder entfernt, denn ein `Krug` ist kein ausführbares Hauptprogramm.

5.1 Definition und Programmierung von `Krug`

Die Eigenschaften und Fähigkeiten eines Kruges sind folgende

1. Ein Krug hat ein nicht negatives Volumen und eine nicht negative Füllung, die höchstens so groß sein kann wie das Volumen. Der Wertebereich ist quasikontinuierlich (Typ `double`).
2. Zwei Krüge sind gleich, wenn sie das gleiche Volumen haben (siehe `equals`).
3. Die Kopie eines Kruges hat das gleiche Volumen und die gleiche Füllung (siehe `clone`).
4. Ein Krug kann Füllung und Volumen mitteilen (siehe `getFüllung`, `getVolumen` und `toString`).
5. Ein Krug kann einem anderen seine ganze eigene Füllung anbieten (siehe `schütteIn`).
6. Wenn ein Krug ein Flüssigkeitsangebot erhält, nimmt er so viel davon an, wie sein Restvolumen zulässt (siehe `nimm`).

Da es in den Java-Paketen keinen annähernd brauchbaren Vorfahren gibt, muss man die Klasse `Krug` direkt von `Object` ableiten. Hier wird eine mögliche Lösung genannt, an Hand derer man die Untiefen studieren kann. Entwickelt man `Krug` mit Schülern, so wird man zuerst einmal über die nötigen Attribute und Methoden diskutieren müssen. Eine wichtige Vorgabe sind die Methoden `equals`, `clone` und `toString`, weil man bei ordentlicher Programmierung verpflichtet ist, „sich an die Verträge zu halten“, die einem die Vorfahr-Methoden aufzwingen. Diese Dinge werden im Anschluss an die Lösung gleich noch diskutiert.

```
public class Krug implements Cloneable{
  protected double volumen;
  protected double füllung;

  public Krug(double v, double f){
    volumen=v;
    setFüllung(f);
  }

  public double getVolumen(){return volumen;}

  public double getFüllung(){return füllung;}

  public double nimm(double angebot){
    double menge=getVolumen()-getFüllung();
    if(angebot<menge) menge=angebot;
    füllung+=menge;
    return menge;
  }
```

```
  public void schütteIn(Krug krug){
    double menge=krug.nimm(getFüllung());
    füllung-=menge;
  }

  void setFüllung(double neu){
    if(neu<0) neu=0;
    else if(neu>volumen) neu=volumen;
    füllung=neu;
  }

  public String toString(){
    return getFüllung()+".."+getVolumen();
  }

  public Object clone(){
    Object ret;
    try{ret=super.clone();}
    catch(CloneNotSupportedException ex){ret=null;}
    return ret;
  }

  public boolean equals(Object obj){
    if(!(obj instanceof Krug)) return false;
    return ((Krug)obj).getVolumen()==getVolumen();
  }

  public int hashCode(){return (int)getVolumen();}

  //Die folgende Methode wird nach der Testphase entfernt
  public static void main(String[] args){
    Krug s8=new Krug(8,8);
    Krug s5=new Krug(5,0);
    Krug s3=new Krug(3,0);
    //Auftrag: Erzeuge zwei Krüge mit Inhalt 4
    System.out.println(s8+" "+s5+" "+s3);
    s8.schütteIn(s5); System.out.println(s8+" "+s5+" "+s3);
    s5.schütteIn(s3); System.out.println(s8+" "+s5+" "+s3);
    s3.schütteIn(s8); System.out.println(s8+" "+s5+" "+s3);
    s5.schütteIn(s3); System.out.println(s8+" "+s5+" "+s3);
    s8.schütteIn(s5); System.out.println(s8+" "+s5+" "+s3);
    s5.schütteIn(s3); System.out.println(s8+" "+s5+" "+s3);
    s3.schütteIn(s8); System.out.println(s8+" "+s5+" "+s3);
  }
}
```

Zur genannten Lösung treten wahrscheinlich einige grundsätzliche und einige Fragen zur Syntax auf.

- In `equals` ist festzulegen, wann zwei Krüge gleich sind. Überschreibt man diese Methode nicht, so gelten zwei Krüge nur dann als gleich, wenn sie „beide“ ein und derselbe Krug sind. Das wäre für Krüge eine übertrieben strenge Sichtweise. Für zwei Krüge mit den Attributen `volumen` und `füllung` ist es angemessener, sie gleich zu nennen, wenn sie gleiches Volumen haben.

  Zusammen mit `equals` sollte man `hashCode` überschreiben. Damit können Instanzen der neuen Klasse effizienter verwaltet werden. In der Schule kann man dies verschweigen. Die Lösung beinhaltet die überschriebene Methode jedoch.

- In `clone` ist festzulegen, wie ein Krug kopiert wird. Da die `clone`-Methode des Vorgängers `Object` aber in so einfachen Fällen wie dem unseren alles schon richtig macht, brauchen wir nur festlegen, dass Klonierung der inneren Logik unserer Klasse nicht widerspricht. Das tun wir, indem wir `Cloneable` implementieren (siehe Kopfzeile). Wenn nichts Prinzipielles dagegen spricht, so sollte man die Herstellung einer Kopie immer ermöglichen. Da die `clone`-Methode von `Object` jedoch nicht `public` sondern nur `protected` ist, überschreiben wir die Methode und erweitern die Zugriffsstufe auf `public`. Damit kann überall eine Kopie von einem `Krug` gemacht werden. Bei dieser Gelegenheit entfernen wir auch gleich `throws CloneNotSupportedException` aus der Deklaration der Methode. Diese Exception kann nämlich nicht mehr auftreten, weil wir ja `Cloneable` implementieren. Im Unterricht kann man die `clone`-Methode so lange verschweigen, wie Kopiervorgänge nicht gebraucht

werden. In der Lösung ist sie jedoch aufgeführt.

Da die Klonierung etwas Grundsätzliches ist, wurde die Methode `clone` schon in die Wurzel der Objekthierarchie `Object` aufgenommen. Das hat zur Folge, dass als Rückgabetyp ebenfalls `Object` definiert werden musste. Deshalb kann man leider die Kopie eines Kruges `k1` nicht mit `Krug k2=k1.clone()` erzeugen, weil der Compiler zur Übersetzungszeit glauben muss, dass die `clone`-Methode ein `Object` und nicht einen `Krug` erzeugt. Ein `Object` ist an `k2` nicht zuweisbar, außer im höchst unwahrscheinlichen Fall, dass das `Object` zur Laufzeit in Wirklichkeit ein `Krug` ist, aber da ist der Compiler lieber vorsichtig und gibt eine Fehlermeldung aus. Da wir in der geschilderten Situation wissen, was der Compiler nicht wissen kann[1], nämlich dass die `clone`-Methode von `Krug` einen `Krug` zurück liefert, können wir den Compiler dann mit einem Cast beruhigen, also `Krug k2=(Krug)k1.clone()`. Zur Laufzeit wird übrigens nochmal getestet, ob `k1.clone()` wirklich ein `Krug` ist. Es könnte ja sein, dass wir den Compiler auf Grund eines Denkfehlers beruhigt haben.

Doch das erklärt nicht den etwas seltsamen Aufbau der Methode mit `try` usw. Der Grund dafür ist, dass die ganze `clone`-Fähigkeit eben schon bei `Object` implementiert ist. Man muss im Grunde nur immer die `clone`-Methode des Vorfahren aufrufen, was im Beispiel mit `super.clone()` ja auch geschieht. Weil aber ein Vorfahr der aktuellen Klasse prinzipiell etwas dagegen haben könnte, ein Duplikat zuzulassen, steht es ihm frei, eine `CloneNotSupportedException` zu erzeugen. Diese muss abgefangen werden, was in unserem Beispiel im `try-catch`-Block passiert. Der Vorfahr von `Krug` ist `Object` und erzeugt diese Exception nur dann, wenn `Krug` nicht das Interface `Cloneable` implemeniert. Dass `Krug` dies tut, ist dem Compiler zwar bekannt, aber er kann nicht wissen, dass dies der Grund dafür ist, dass die `CloneNotSupportedException` niemals kommen wird[2]. Der insgesamt etwas seltsame Eindruck, den die `clone`-Methode macht, liegt also im Grunde an ihrer frühen Stellung im Ahnenverzeichnis und ihren trotzdem starken Fähigkeiten.

Die `clone`-Methode liefert eine neue Instanz der Klasse des geklonten Objekts. Alle Attribute werden kopiert, was bei primitiven Typen ein tatsächliches Kopieren bedeutet, bei Objekten aber nur eine Kopie des Zeigers. Das Orginal und sein Klon teilen sich also Attribut-Objekte, die eigentlich nur dem Original gehören. Braucht man eine vollständig unabhängige Kopie, so müssen in der `clone`-Methode des Originals nach dem Aufruf von `super.clone()` alle Attribut-Objekte ersetzt werden durch Kopien (die man wieder durch `clone`-Aufrufe dieser Objekte bekommt).

- In der `toString`-Methode hat ein Objekt die Möglichkeit, eine textliche Aussage über sich zu machen. Hat man einen Krug `k`, und „gibt ihn aus“ mit `System.out.println("sagwas "+k)`, so wird `k.toString()` aufgerufen[3]. Überschreibt man die Methode nicht, so beinhaltet die Aussage lediglich die Speicheradresse bei der das Objekt liegt. Für einen Krug ist das eine ziemlich nutzlose Aussage. Lieber sollte ein Krug sagen, welche Füllung und welches Volumen er hat.
- `nimm` bekommt ein Angebot und liefert als Rückgabewert, wieviel von dem Angebot tatsächlich angenommen wurde.
- In `equals` kommt der `instanceof`-Operator vor. Er hat den Wert `true`, wenn der linke Operand (ein Objekt) den Typ des rechten Operanden (eine Klasse) hat. Das `!` ist lediglich die Verneinung. Eine Zeile weiter muss dann `obj` zu einem `Krug` gecastet werden, damit die Methode `getVolumen` aufgerufen werden kann.

5.2 Der Nachfahre See

In unserer Krug-Welt fehlen noch Dinge wie ein Ausguss und ein Zapfhahn. Beide kann man sich in einem See vereinheitlicht denken. Aus einem See kann man praktisch beliebig viel Wasser entnehmen (und es ist immer noch welches da) und man kann praktisch beliebig viel Wasser hinein schütten (und es ist immer noch Platz).

So ein See ist aber auch nichts wesentlich anderes als ein gigantischer Krug. Deshalb sollte es möglich sein, die Klasse `See` von der Klasse `Krug` abzuleiten. Wenn nur wenige Änderungen nötig sind, ist unser Konzept schlüssig.

[1] Im speziellen Fall von `clone` könnte der Compiler den Rückgabetyp schon wissen. Dazu müsste die `clone`-Methode aber eine Spezialbehandlung erfahren, bei allen anderen Methoden schaut der Compiler nämlich nur den Rückgabetyp an.

[2] Um dies zu ändern, hätte man der `CloneNotSupportedException` eine Sonderstellung unter allen Exceptions geben müssen.

[3] Diese syntaktische Auffälligkeit gibt es nur bei der Aneinanderhängung von Strings und wurde deshalb in die Sprache aufgenommen, weil Missverständnisse im Grunde nicht möglich sind und das Konstrukt halt doch recht angenehm ist.

1. Füllung und Volumen eines Sees sind unendlich.
2. Alle Seen sind gleich. Im Anwendungsfall wird aber gewöhnlich nur einer benötigt.
3. Wenn ein See ein Flüssigkeitsangebot erhält, nimmt er alles an.
4. Wenn ein See ein Flüssigkeitsangebot macht, so ist dieses stets groß genug, den Empfänger zu füllen.

Der erste Punkt scheint etwas problematisch. Da Füllung und Volumen aber `double`-Attribute sind, kann man ihnen den Wert ∞ zuweisen. Zugriff auf diesen ungewöhnlichen Wert erhält man in der Hilfsklasse `Double`[4].

Im Detail erkennt man, dass ein See nur fast ein Krug mit unendlichem Volumen und Inhalt ist. Die kleinen Probleme ergeben sich aber nur aus den Besonderheiten beim Rechnen mit unendlichen Werten. $\infty - \infty =$`NaN` (not a number), also nicht definiert. Da diese Differenz in `Krug` auftreten kann, müssen bei Verwendung von Seen folgende kleine Änderungen vorgenommen werden. (Man sieht übrigens, dass Seen einfachere Gebilde sind als Krüge.)

```
public class See extends Krug{
  private static final String SEE="See";

  public See(){
    super(Double.POSITIVE_INFINITY, Double.POSITIVE_INFINITY);
  }

  public double nimm(double angebot){
    return angebot;
  }

  public void schütteIn(Krug krug){
    krug.nimm(krug.getVolumen());
  }

  public String toString(){
    return SEE;
  }
}
```

Auch hier soll wieder eine kleine Diskussion folgen

- Der Konstruktor hat keine Argumente, weil alle Seen gleich sind. Mit `super` wird der Konstruktor des Vorfahren `Krug` aufgerufen und ein voller Krug mit unendlichem Volumen erzeugt.
- `nimm` teilt dem Anbieter mit, dass das ganze Angebot angenommen wird. In Wirklichkeit wird das Angebot aber mathematisch völlig ignoriert. Der alte Wert der Füllung (∞) ist ja immer noch richtig.
- `schütteIn` bietet sicherheitshalber so viel an, dass selbst ein leerer Empfänger voll wird. Dass der Empfänger den Namen `krug` und den Typ `Krug` hat, darf nicht darüber hinweg täuschen, dass `krug` zur Laufzeit auch ein Nachfahre von `Krug` sein kann, wie etwa ein `See`. Mathematisch richtig wäre es auch gewesen, ∞ anzubieten.

  Der Rückgabewert von `krug.nimm` wird übrigens wieder großzügig ignoriert.
- `toString` hätte nicht überschrieben werden müssen. Die Ausgabe `Infinity..Infinity` des Vorfahren wäre ebenfalls korrekt. Das Wort `See` ist aber vielleicht anschaulicher. Die Verwendung einer Konstante verhindert, dass bei jedem Aufruf von `toString` der String `See` neu erzeugt wird.
- Zum Testen der Klasse `See` sollte man die `main`-Methode (die ruhig in `Krug` bleiben darf) so ändern, dass sie auch ein Objekt vom Typ `See` erzeugt und dieses verwendet.

5.3 Verwendung in einem Spiel

Krüge und Seen kann man überall dort verwenden, wo ihre Eigenschaften den gewünschten Sinn erfüllen: Als Tankstellen und Autotanks in einer Rennsimulation oder als Grundlage für die Entwicklung eines Backtracking-Algorithmus, der selbständig die kürzeste Lösung zu einem Problem der Art in der

[4] So eine Hilfsklasse gibt es für alle primitiven Datentypen.

`main`-Methode findet oder, wie es hier geschehen soll, in einem Spiel, das dem Benutzer eine graphische Oberfläche zur Verfügung stellt, auf der er selber herumprobieren kann (s. Abbildung 5.1).

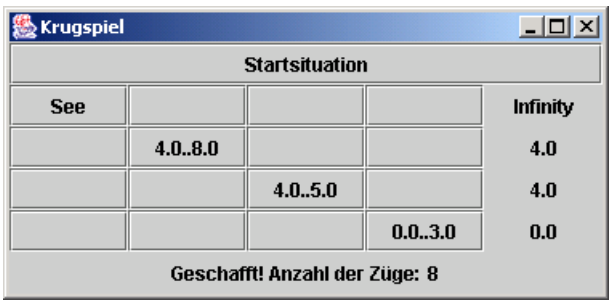

Abbildung 5.1: GUI-Darstellung des Spiels

Das Aussehen ist sehr unspektakulär. Schaut man sich aber den folgenden Programmtext an, so wird man mit Freude eine Verschönerung des Aussehens auf später verschieben. (Das liegt im wesentlichen daran, dass diese Version nicht in mehrere nach Funktion aufgeteilte Klassen zerlegt ist. Das wird der Inhalt der nachfolgenden Übung sein.)

Wie spielt man nun mit diesem seltsamen Ding? Z. B. bedeutet `4.0..8.0`, dass wir einen Krug mit Füllung 4 und Volumen 8 haben. Klickt man auf den Button links davon, so geht der ganze Inhalt in den See, klickt man rechts davon, so verlieren wir Füllung 1 an den dann vollen Krug mit Volumen 5. Allgemein ausgedrückt gibt die Zeile an, welcher Krug abgibt und die Spalte, welcher aufnimmt. Die Hauptdiagonale dient nur der Anzeige (kann aber auch gedrückt werden[5]). Die Anzeigenspalte aus `JLabels` dient dazu, dem Spieler sein Füllungsziel anzuzeigen.

Der Konstruktor

Im Konstruktor wird das Aussehen definiert. Wegen der vielen Details, denen sich der Anfänger ungeschützt ausgesetzt sieht, ist das der unangenehmste Teil des Programms und der Grund, warum man diesen Teil entweder vorgeben oder mit NetBeans erstellen sollte (siehe dazu Kapitel 3). Eine zuvor eingeschobene Unterrichtssequenz zu dem Thema ist ebenfalls sehr hilfreich. Genaueres zum Thema Oberflächengestaltung findet man im Tutorial auf der CD unter `//books/tutorial.zip`.

Das Hauptfenster vom Typ `KrugSpiel` ist ein direkter Nachfahre von `JFrame`, einer der wenigen Klassen, die Hauptfenster sein können. Der Anzeigebereich hat standardmäßig ein `BorderLayout`, von dem wir aber `WEST` und `EAST` nicht verwenden, sondern nur `NORTH` für einen Reset-Button, `CENTER` für ein `JPanel` mit `JButton`-Matrix nebst Sollanzeige aus `JLabels` und `SOUTH` für ein `JPanel` mit zwei `JLabels`, die zusammen die Statuszeile darstellen. Das Standardlayout eines `JPanels` ist `FlowLayout`, also eine Anordnung der zugefügten Elemente nebeneinander. Für die Statuszeile ist das genau richtig (das erste für den Text, das zweite für die Zahl). Für die Buttonmatrix musste ein `GridLayout` der Größe 4×5 angefordert werden.

Mit all diesen Vorbemerkungen sollte es möglich sein, den Konstruktor des Spiels zu verstehen. Der ruft mit `super` den Konstruktor des Vorfahren `JFrame` auf und übergibt ihm einen String, der, wie man in der Hilfe zu `JFrame` erfährt, in der Kopfzeile angezeigt wird. Mit `this.start` ist das Attribut des `KrugSpiels` gemeint, mit `start` der Parameter des Konstruktors. Mit `setDefaultCloseOperation` wird festgelegt, was passiert, wenn der Benutzer das Fenster schließen will. Wir legen fest, dass dann das Programm beendet werden soll. `this.getContentPane` fordert von unserem Fenster den Anzeigebereich an, in den wir dann mit `add` unsere drei (!) Komponenten hinzufügen, einen `JButton` und zwei `JPanels` (die aber selber wieder Komponenten enthalten und je nach Manager anordnen).

Bei allen erzeugten Buttons findet man noch den Befehl `einbutton.addActionListener(this)`. Damit wird `this`, also unser Spiel als `ActionListener` beim jeweiligen Button angemeldet und von diesem benachrichtigt, falls er gedrückt wurde. Die Benachrichtigung erfolgt dadurch, dass der gedrückte Button die Methode `actionPerformed` unseres Spiels aufruft. Dazu muss unser Spiel natürlich diese Methode erst mal haben. Weil es sie hat, darf sich unser Spiel als `ActionListener` bezeichnen, was es

[5] Es hat keine Auswirkung, außer dass man einen unnötigen Zug macht. Das ist zweifellos ein Makel, den man beheben sollte und auch leicht beheben kann!

ganz am Anfang (bei `implements...`) auch tut. Der Compiler testet wegen dieses `implements...`, ob wir wirklich die geforderte Methode[6] besitzen und erlaubt uns, uns als `ActionListener` anzumelden.

Im Array `merk` werden gleich zu Anfang Kopien der anfänglichen Füllungen erstellt. Dies ist nötig, damit das Spiel einen Reset anbieten kann. Für die vielen nicht ausdrücklich erwähnten Kleinigkeiten, empfehle ich die Hilfe, wenn der wache[7] Menschenverstand einmal nicht reichen sollte.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class KrugSpiel extends JFrame implements ActionListener{
  Krug[] ist;              //die Krüge in ihrem jeweils aktuellen Zustand
  double[] merk;           //merkt sich die anfänglichen Füllungen
  double[] soll;           //die Füllungen, die erreicht werden sollen
  int anzahl;              //die Anzahl der gemachten Züge
  JLabel fortschrittLabel; //der Text in der Statuszeile
  JLabel anzahlLabel;      //die Zahl daneben
  JButton[][] gießen;      //zweidimensionales Array von JButtons
  JLabel[] sollLabel;      //die letzte Spalte mit der Anzeige von soll
  JButton reset;           //für die Wiederherstellung des Anfangszustands
  private static final String L_WEITER="Anzahl der Züge:";
  private static final String L_FERTIG="Geschafft! "+L_WEITER;

  public KrugSpiel(Krug[] start, double[] soll){
    super("Krugspiel");
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.ist=start;
    int anz=start.length;
    merk=new double[anz];
    for(int i=0; i<anz; i++) merk[i]=start[i].getFüllung();
    this.soll=soll;
    Container c=this.getContentPane();//hat BorderLayout
    //erzeuge den Reset-Button
    reset=new JButton("Startsituation");
    reset.addActionListener(this);
    c.add(reset, BorderLayout.NORTH);
    //erzeuge die Anzeige der benötigten Schritte
    JPanel p;
    p=new JPanel();//FlowLayout ist Default
    fortschrittLabel=new JLabel();
    p.add(fortschrittLabel);
    anzahlLabel=new JLabel();
    p.add(anzahlLabel);
    c.add(p, BorderLayout.SOUTH);
    //Erzeuge die Matrix
    p=new JPanel(new GridLayout(anz, anz+1));
    c.add(p, BorderLayout.CENTER);
    gießen=new JButton[anz][anz];
    sollLabel=new JLabel[anz];
    for(int zeile=0; zeile<anz; zeile++){
      for(int spalte=0; spalte<anz; spalte++){
        JButton b=new JButton();
        b.addActionListener(this);
        p.add(b);
        gießen[zeile][spalte]=b;
      }
      sollLabel[zeile]=new JLabel();
      sollLabel[zeile].setHorizontalAlignment(JLabel.CENTER);
      p.add(sollLabel[zeile]);
    }
    reset();
    pack();
    show();
  }
```

[6] Gefordert im Interface `ActionListener`; siehe die Hilfe dazu.
[7] Kein Druckfehler, „sch“ nicht beabsichtigt!

```
  public void reset(){
    int anz=ist.length;
    if(anz!=soll.length) soll=new double[anz];
    for(int i=0; i<anz; i++){
      if(soll[i]<0 || soll[i]>ist[i].getVolumen())
        soll[i]=ist[i].getFüllung();
      ist[i].setFüllung(merk[i]);
      sollLabel[i].setText(String.valueOf(soll[i]));
      setAnzeigeText(i);
    }
    anzahl=0;
    anzahlLabel.setText("0");
    fortschrittLabel.setText(L_WEITER);
  }

  public void actionPerformed(ActionEvent e){
    Object button=e.getSource();
    if(button==reset) {reset(); return;}
    int anz=ist.length;
    int zeile, spalte=0;
    such: for(zeile=0; zeile<anz; zeile++)
      for(spalte=0; spalte<anz; spalte++)
        if(button==gießen[zeile][spalte]) break such;
    if(zeile<anz && spalte<anz){
      ist[zeile].schütteIn(ist[spalte]);
      setAnzeigeText(zeile);
      setAnzeigeText(spalte);
      anzahl++;
      anzahlLabel.setText(String.valueOf(anzahl));
      boolean fertig=true;
      for(int i=0; i<anz; i++) fertig&=ist[i].getFüllung()==soll[i];
      fortschrittLabel.setText(fertig?L_FERTIG:L_WEITER);
    }
  }

  private void setAnzeigeText(int n){
    gießen[n][n].setText(ist[n].toString());
  }

  public static void main(String[] args){
    new KrugSpiel(
      //new Krug[]{new Krug(8,8), new Krug(5,0), new Krug(3,0)},
      //new double[]{4, 4, 0}
      new Krug[]{new See(), new Krug(8,0), new Krug(5,0), new Krug(3, 0)},
      new double[]{Double.POSITIVE_INFINITY, 4, 4, 0}
    );
  }
}
```

Die Methoden `reset` und `setAnzeige`

Zur Wiederherstellung des Anfangszustands mitten im Spiel werden in `reset` die Füllungen von `ist` so gesetzt, wie es anfangs in `merk` festgehalten wurde. Eventuell unbrauchbare Soll-Werte werden so geändert, dass das Spiel nicht abbrechen muss. Der Text der `sollLabel` wird gesetzt und die Statuszeile zurückgesetzt.

Der Text der Buttons in der Hauptdiagonale wird in `setAnzeige` erstellt. Da die `toString`-Methode von `Krug` und `See` ausreichend schön ist, wird sie hier verwendet. Wenn eine andere Anzeigeform gewünscht wird, so kann hier ein etwas größerer Aufwand betrieben werden.

Die `actionPerformed`-Methode

Wie schon erwähnt, wird diese Methode von allen Buttons aufgerufen, bei denen unser `KrugSpiel` angemeldet ist, also wirklich von allen. Deshalb muss erst einmal herausgefunden werden, von welchem Button

der Aufruf kommt. Das ist deshalb möglich, weil beim Aufruf ein Informationspäckchen mitgeliefert wird, ein sogenannter `ActionEvent`, von dem wir den Aufrufer erfahren können.

Falls es der Reset-Button ist, wird lediglich `reset()` aufgerufen. Andernfalls wird die Buttonmatrix durchsucht. Dann wird umgefüllt, die Diagonalanzeige und die Statuszeile aktualisiert und schließlich noch getestet, ob der Sollzustand schon erreicht ist.

Die `main`-Methode

Hier wird einfach mit `new` eine Instanz des `KrugSpiel`s erzeugt. Ungewöhnlich mag es erscheinen, dass der Zeiger, den `new` zurückliefert, nicht irgendwo abgespeichert wird, aber das ist nicht nötig, weil wir damit ja nichts mehr vorhaben. Beachtenswert ist auch die Tatsache, dass nach diesem kurzen `new`-Befehl das Programm nicht sofort zu Ende ist, obwohl die Methode `main` jetzt fertig ist. Der Grund dafür ist, dass die graphische Oberfläche in einem eigenen Thread läuft. Erst wenn alle Threads eines Programms beendet sind, endet das Programm. Den graphischen Thread beendet man durch Schließen des Fensters.

Falls die kurze Schreibweise nicht ganz verständlich ist, hier noch einmal eine etwas ausführlichere, die das gleiche tut.

```
public static void main(String[] args){
  Krug[] krugarray=new Krug[4];
  krugarray[0]=new See();
  krugarray[1]=new Krug(8,0);
  krugarray[2]=new Krug(5,0);
  krugarray[3]=new Krug(3,0);
  double[] ziel=new double[4];
  ziel[0]=Double.POSITIVE_INFINITY;
  ziel[1]=ziel[2]=4;
  KrugSpiel ks=new KrugSpiel(krugarray, ziel);
}
```

Da der Konstruktor allgemein gehalten ist (was auch zu einigen schwieriger verständlichen Zeilen geführt hat), ist hier die Anzahl der verwendeten Krüge beliebig. Es müssen also nicht genau vier sein!

5.4 Mehr Objekte – mehr Ordnung

Wir können hier mit einem gewissen Stolz behaupten, dass wir ein komplexes Gefüge aufgetrennt haben in mehrere, übersichtlichere Teile. Schließlich gibt es die Krüge ganz unabhängig vom Rest des Spiels. Hat man ihre Funktionsweise verstanden und implementiert, so ist der Geist frei von ihnen und kann sich den weiteren Problemen widmen, dem Spiel eben.

Das `KrugSpiel` selbst ist aber noch ein heilloses Durcheinander, was daran liegt, dass wir uns anfangs über den Begriff *Spiel* nicht genügend Gedanken gemacht haben. Es *hat* so unterschiedliche Elemente wie Krüge, Buttons und Anzeigelabels und *tut* so unterschiedliche Dinge wie schütten, Buttons heraussuchen, Spielstand testen und Anzeigen setzen. Für den Zweck dieses kleinen Spiels mag der aktuelle Entwicklungsstand hinnehmbar sein. Aber wenn wir nicht objektorientiert denken wollten, hätten wir auch gleich noch den ganzen Krugmechanismus im `KrugSpiel` belassen können.

In den folgenden Abschnitten dieses Kapitels soll nun das vorliegende Geflecht begrifflich ordentlich auseinander getrennt werden. Wir werden feststellen, dass das Denken auf eine inspirierende Weise klarer wird – zu dem Preis, dass die Gesamtheit des Programmtextes größer wird. Aber die Klarheit steigt überproportional!

Als Folge der objektorientierten Denkweise in diesem Abschnitt werden mehrere, semantisch scharf getrennte Klassen programmiert. Einige davon sind sichtbare Komponenten, die man geeignet zur Darstellung bringen muss, was aus dem Programm heraus auch ganz einfach geht, wenn man sich mit der Technik vertraut gemacht hat. Mit Entwicklungsumgebungen wie Delphi oder NetBeans, die man mit Schülern gern von Anfang an einsetzt, ist die Platzierung eigener Oberflächenkomponenten aber gar nicht so leicht, weil sie ja nicht in der Komponentenleiste auftauchen. Deswegen ist dieser Abschnitt ein aus Schülersicht fortgeschrittener.

5.5 Objekte senden Nachrichten

Eine Objekten zu Grunde liegende Hauptidee ist, dass sie für sich existieren und funktionieren sollten. Wenn sie aufeinander einwirken, dann durch gegenseitige Nachrichten. Als Nachricht bezeichnet man den Aufruf einer Methode des einen Objektes durch ein anderes. Wenn also z. B. ein Krug die `nimm`-Methode eines anderen Kruges aufruft, so hat er ihm eine „`nimm`-Nachricht gesendet“.

Dass es eine solch enge Zusammenarbeit von zwei Krügen geben würde, war zur Programmierzeit absehbar und deshalb leicht zu implementieren. Was tut man aber bei der Programmierung von `Krug`, wenn man auch anstrebt, fremde Klassen zu benachrichtigen, so fremd, dass man sie zur Programmierzeit noch gar nicht kennt. Es könnte doch sein, dass ein `Krug` später einmal einen `Wirt` oder eine `Kasse` benachrichtigen soll, wenn er von einem `See` befüllt wurde.

In Java verwendet man für diesen weiter greifenden Nachrichtentyp das Event/Listener-Modell. Damit wird die Idee „Nachricht“ abstrahiert und auf das Wesentliche reduziert:

- Der Sender stellt einen Anmeldemechanismus zur Verfügung, an den sich jeder Interessent (*Listener*) wenden kann.
- Der Sender benachrichtigt alle Listener, wenn etwas passiert ist, das sie interessiert. Wenn es Einzelheiten mitzuteilen gibt, werden diese beim Aufruf in Form eines Informationspäckchens übergeben. Ein solches Informationspäckchen nennt man üblicherweise *Event*.
- Welche Methoden ein Listener zur Verfügung zu stellen hat, ist öffentlich in einem Interface definiert.

Im bisherigen `KrugSpiel` wurde dieser Mechanismus schon eingesetzt: Das Interface `ActionListener` legt fest, dass ein Interessent die Methode `actionPerformed` haben muss, um sich `ActionListener` nennen zu dürfen. Als solcher kann er sich dann bei einem Button durch dessen `addActionListener`-Methode anmelden. Wird der Button gedrückt, so ruft er von allen angemeldeten `ActionListenern` die Methode `actionPerformed` auf und übergibt einen `ActionEvent` mit den Details[8].

Der strukturelle Makel an der bisherigen Lösung ist, dass `KrugSpiel` eine größere Funktionalität besitzt, als es der Name nahe legen würde. Wir wollen deshalb die Aufgaben genau analysieren und auf mehrere, spezifischere Klassen verteilen. Abbildung 5.2 zeigt eine mögliche Aufgabenverteilung. Blaue

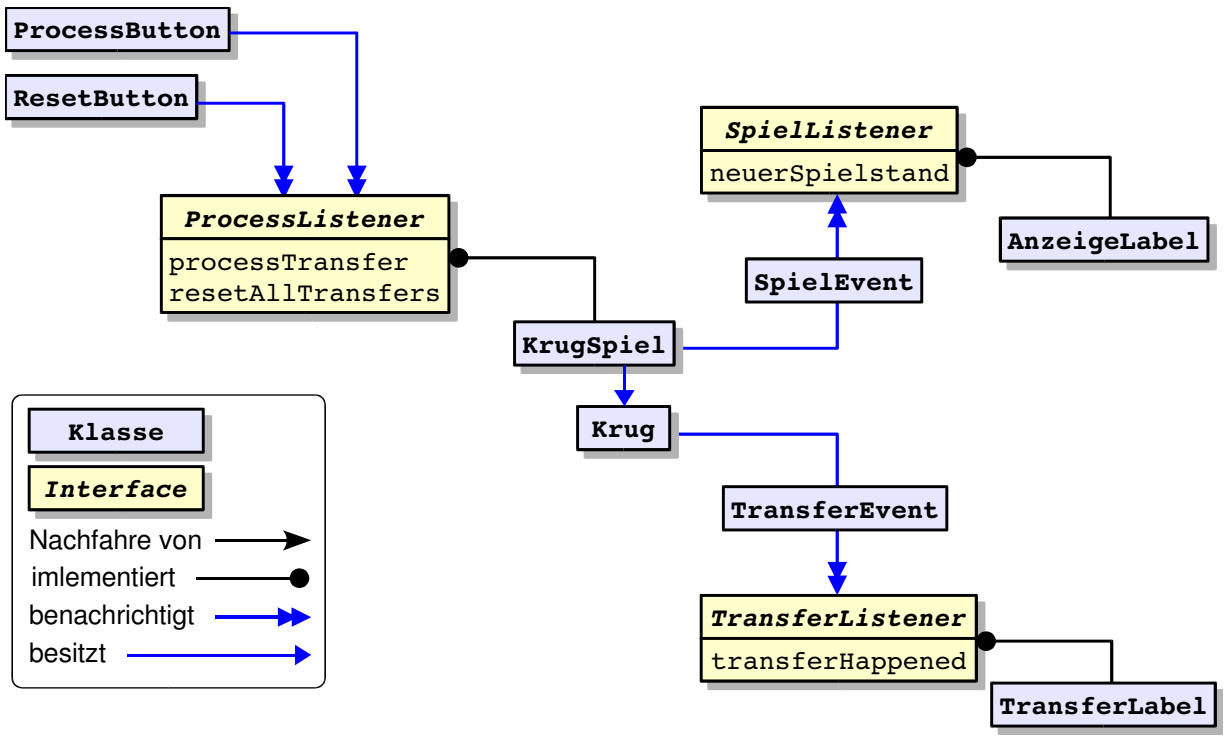

Abbildung 5.2: Interfaces und Nachrichten im verbesserten Krugspiel

Doppelpfeile zeigen an, wer an wen Nachrichten sendet. (Nicht schattierte Klassen sind die dabei gesendeten Events.) Der Sender hat einen Zeiger auf den Empfänger und ruft Empfänger-Methoden auf. In

[8] Ein Detail ist z.B., ob auch die Shift-Taste gedrückt war.

den drei Fällen wo die Bindung nicht ganz innig[9] ist, stellt der Empfänger die aufrufbaren Methoden über ein Interface zur Verfügung.

5.6 Nachrichten von einem `Krug`

Bisher konnte ein `Krug` Methoden eines anderen `Krug`es aufrufen. Für Nachrichten an fremde (auch noch nicht existierende) Klassen, bauen wir nun einen Event-Mechanisums ein.

- Interessant ist ein Krug durch seine Umschüttvorgänge; wir wollen sie `TransferEvent`s nennen. Mitzuteilen ist bei jedem solchen Vorgang, welcher `Krug` abgibt, welcher andere `Krug` aufnimmt und wieviel.
- Um über einen Schütt-Vorgang benachrichtigt werden zu können, muss ein Objekt die Methode `transferHappened` haben und darf sich dann `TransferListener` nennen; wir legen dies im gleichnamigen Interface fest. Ein `Krug` ruft diese Methode gegebenenfalls auf und übergibt dabei auch einen `TransferEvent` (mit den oben genannten Informationen).
- Um auch tatsächlich benachrichtigt zu werden, muss ein `TransferListener`[10] beim interessanten `Krug` angemeldet werden. In `Krug` steht deshalb die Methode `addTransferListener` zur Verfügung. (Auch `removeTransferListener` sollte nicht fehlen.) Alle angemeldeten `TransferListener` werden in einem `Vector` gesammelt.
- Üblicherweise (aber nicht zwingend) lagert man auch den Teil in eine Methode aus, der die Listener benachrichtigt. Diese Methode nennen wir `fireTransferEvent`.

Folgendes muss bei `Krug` hinzugefügt werden:

```
private ArrayList listeners=new ArrayList();

public void addTransferListener(TransferListener g){
  listeners.add(g);
}
public void removeTransferListener(TransferListener g){
  listeners.remove(g);
}

protected void fireTransferEvent(Krug partner, double delta){
  TransferEvent te=new TransferEvent(this, partner, delta);
  for(int i=0, n=listeners.size(); i<n; i++)
    ((TransferListener)listeners.get(i)).transferHappened(te);
}
```

Die Methode `fireTransferEvent` muss dann aber auch noch von überall her aufgerufen werden, wo Änderungen des Kruginhalts auftreten können, das sind in `Krug` die Methoden

```
public void schütteIn(Krug krug){
  double menge=krug.nimm(getFüllung());
  füllung-=menge;
  this.fireTransferEvent(krug, -menge);
  krug.fireTransferEvent(this, menge);
}

void setFüllung(double neu){
  if(neu<0) neu=0;
  else if(neu>volumen) neu=volumen;
  füllung=neu;
  neu=neu-füllung;
  this.fireTransferEvent(this, neu);
}
```

In `See` geht das entsprechend wieder einfacher

```
public void schütteIn(Krug krug){
  double menge=krug.nimm(krug.getVolumen());
  this.fireTransferEvent(krug, -menge);
```

[9] Der Begriff wird im Laufe der Diskussion klarer werden.
[10] Abkürzende Sprechweise für „Objekt, das das Interface `TransferListener` implementiert“.

```
    krug.fireTransferEvent(this, menge);
  }

  void setFüllung(double neu){
    fireTransferEvent(this, füllung);
  }
```

Was ein `TransferListener` ist, wird im gleichnamigen Interface festgelegt.

```
public interface TransferListener{
  void transferHappened(TransferEvent te);
}
```

Ein `TransferEvent` ist ein kleiner Behälter für die genannten drei Informationen. Da Events etwas sehr Alltägliches sind, sollte es nicht verwundern, dass es schon einen Vorfahren gibt. Die einzige Funktionalität aber, die von diesem übernommen wird, ist das Merken des Event-Erzeugers und das ist bei uns ja immer der erste der beiden beteiligten Krüge.

```
import java.util.EventObject;

public class TransferEvent extends EventObject{
  private Krug partner;
  private double menge;

  public TransferEvent(Krug source, Krug partner, double menge){
    super(source);
    this.partner=partner;
    this.menge=menge;
  }

  public Krug getKrug(){return (Krug)super.getSource();}
  public Krug getPartner(){return partner;}
  public double getMenge(){return menge;}
}
```

5.7 Interessenten für Nachrichten von einem `Krug`

Denkbar wäre hier z. B. eine graphische Komponente, die einen Krug mit wechselnder Füllung zeichnet, oder eine Komponente, die ein Warnsignal spielt, wenn durch einen Schütt-Vorgang ein Grenzwert über- oder unterschritten wird. Allen solchen Interessenten von `TransferEvent`s ist gemeinsam, dass sie die Methode `transferHappened` haben müssen und sich damit als `TransferListener` ausweisen.

Da es uns ums Prinzip geht, wählen wir eine sehr einfache Komponente: Ein `TransferLabel` hat die Aufgabe, den aktuellen Stand eines bestimmten Kruges anzuzeigen und wird nachher, wie schon in der ersten Programmversion, wieder auf der Diagonalen des Spiels zu sehen sein. Es soll ein Nachfahre von `JLabel` sein, aber eben ein solcher, der die `transferHappened`-Methode besitzt. Genau beim Aufruf dieser Methode soll sich auch die Anzeige anpassen.

Was ist nun daran besser als in der alten Version, wo die Anzeige ja auch bei jedem Schütt-Vorgang geändert wurde und zudem noch das schon existierende `JLabel` ohne Probleme verwendet werden konnte? Der Vorteil ist die entfallene „Innigkeit“! In der alten Version war das `JLabel` fest ins Hauptprogramm[11] integriert, und dieses hat eine Spezialmethode des `JLabels` (nämlich `setText`) aufgerufen. Wollte man stattdessen eine tonerzeugende Komponente, so hätte man das Hauptprogramm ändern müssen. Das ist jetzt nicht mehr der Fall, denn wir haben bei unserem Abstraktionsprozess erkannt, dass jeder denkbare Interessent zufrieden gestellt werden kann, wenn er die Methode unseres Interfaces implementiert.

```
import javax.swing.JLabel;

public class TransferLabel extends JLabel implements TransferListener{
  public TransferLabel(){
    super(" ");
    setHorizontalAlignment(JLabel.CENTER);
  }
```

[11] Wenn es ein Hauptprogramm gibt und dieses einen Großteil der Funktionalität übernimmt, dann hat man wahrscheinlich nicht sauber klassifiziert.

```
    public void transferHappened(TransferEvent te){
      setText(te.getKrug().toString());
    }
  }
```

Damit wird auch der Begriff *Interface* oder *Schnittstelle* klar. Ein Interface legt fest, wie die Methoden heißen, auf die sich mehrere kommunikationswillige Objekte geeinigt haben, noch bevor es auch nur eines von ihnen geben muss.

5.8 Verwaltung des Spielstands

Der Zweig rechts oben in Abbildung 5.2 hat die gleiche Struktur wie der soeben besprochene von rechts unten im gleichen Bild. Es geht dabei um die Wiedergabe des Spielstands. Bisher wurde dies ebenfalls durch ein `JLabel` bewerkstelligt. Aber wieder kann man ahnen, dass später vielleicht eine Lautuntermalung dazu kommen soll deretwegen man nicht gleich das ganze „Hauptprogramm“ ändern will. Die Lösung sei als Übung empfohlen und befindet sich auch auf der CD in `//code/krug2/`.

Beachtenswert ist hier die Tatsache, dass der Spielstand natürlich nicht Sache der Krüge sein kann. Dafür ist das `KrugSpiel` verantwortlich, das aber eine innige Beziehung zu den Krügen hat – pragmatischer ausgedrückt: Das Krugspiel hat Krüge, aber das liegt ja auch in der Natur der Sache.

5.9 Steuerung des Krugspiels

Bis jetzt war das `KrugSpiel` bei allen Buttons als `ActionListener` angemeldet. Wenn dann von einem Button die `actionPerformed`-Methode des `KrugSpiel`s aufgerufen wurde, so musste das `KrugSpiel` erst einmal ermitteln, ob es sich um den Reset-Button oder einen Umfüll-Button handelte. Im Falle eines Umfüll-Buttons musste zusätzlich ermittelt werden, welche Krüge beteiligt sind. Dies sind Aufgaben, die das `KrugSpiel` nicht haben sollte.

Das Krugspiel hat Zugriff auf alle Krüge und kennt den Spielstand. Daneben sollte es keine Verwaltungsaufgaben erfüllen, sondern nur Schütt-Befehlen gehorchen. Diese gibt es in zweierlei Ausprägung: *Zurücksetzen des Spiels* und *Schütte einen bestimmten Krug in einen anderen bestimmten Krug*

Um den Leser nicht zu langweilen und dem Lernenden einen Fortschritt zu ermöglichen, soll hier gezeigt werden, dass man nicht wegen jeder Kleinigkeit von Nachricht auch immer gleich einen Event verwenden muss (obwohl man es natürlich mit voller Berechtigung auch tun könnte); die Nummern der beteiligten Krüge werden direkt übergeben. Wir programmieren dafür wieder zwei neue Sorten von Buttons (`ResetButton` und `ProcessButton`), aber auch eine Uhr könnte einen solchen Vorgang hervorrufen. In der realen Welt gibt es viele Auslöser!

Das `KrugSpiel` stellt sich für Klassen, die steuernd Einfluss nehmen wollen, als `ProcessListener` dar, also wieder über ein Interface. Damit verleihen wir der Idee Ausdruck, dass es auch noch andere Spiele geben könnte, die ebenfalls einfach nur dieses Interface implementieren müssten, um (z. B.) von unseren Buttons gesteuert werden zu können. Immerhin ist das denkbar für alle Spiele mit durchnummerierten Feldern. Die beiden übertragenen Nummern bedeuten dann die Nummern der an einem Zug beteiligten Felder.

Für das Zurücksetzen des Spiels muss keine Information übertragen werden – der Aufruf der Methode `resetAllTransfers` genügt. Für einen Umschütt-Vorgang sind nur zwei ganze Zahlen zu übermitteln, nämlich die Nummern der beteiligten Krüge im Spiel. Da beide Male die Aktion beim Drücken des Buttons stattfinden soll, kann man den Aufruf der Interface-Methode aus der Methode `fireActionPerformed`, die den Nachfahren zur Verfügung steht, durchführen.

Wir sehen uns den `ProcessButton` an. Der `ResetButton` ist noch einfacher (siehe CD `//code/krug2/`).

```
import javax.swing.JButton;
import java.awt.event.ActionEvent;

public class ProcessButton extends JButton{
  protected int a;
  protected int b;
```

```
  protected ProcessListener proclistr;

  public ProcessButton(ProcessListener p, int a, int b){
    proclistr=p;
    this.a=a;
    this.b=b;
  }

  protected void fireActionPerformed(ActionEvent e){
    proclistr.processTransfer(a, b);
    super.fireActionPerformed(e);
  }
}
```

Der wesentliche Teil von `KrugSpiel` ist natürlich die Implementation der beiden Interface-Methoden.

```
import java.util.ArrayList;//dynamische Arrays mit beliebigem Inhalt

public class KrugSpiel implements ProcessListener{
  ArrayList krüge=new ArrayList();//Krüge im Anfangszustand
  ArrayList start=new ArrayList();//Füllungen im Anfangszustand
  ArrayList soll =new ArrayList();//Sollfüllung
  int schritte;
  private ArrayList listeners=new ArrayList();

  public KrugSpiel(Krug[] kr, double[] so){
    for(int i=0; i<kr.length; i++) addKrug(kr[i], so[i]);
  }

  public void addKrug(Krug k, double s){
    if(s<0) s=0;
    if(s>k.getVolumen()) s=k.getVolumen();
    krüge.add(k);
    start.add(new Double(k.getFüllung()));
    soll .add(new Double(s));
  }

  public void resetAllTransfers(){
    schritte=0;
    for(int i=0; i<krüge.size(); i++)
      getKrug(i).setFüllung(getStart(i));
    fireSpielEvent();
  }

  public void processTransfer(int a, int b){
    getKrug(a).schütteIn(getKrug(b));
    schritte++;
    fireSpielEvent();
  }

  public int getAnzahl(){return krüge.size();}

  public Krug getKrug(int i){return (Krug)krüge.get(i);}

  public double getSoll(int i){
    return ((Double)soll.get(i)).doubleValue();
  }

  public double getStart(int i){
    return ((Double)start.get(i)).doubleValue();
  }

  public int getSchritte(){return schritte;}

  public boolean isFertig(){
    boolean fertig=true;
    for(int i=0; i<krüge.size(); i++)
      fertig&=getKrug(i).getFüllung()==getSoll(i);
    return fertig;
```

```
  }

  public void addSpielListener(SpielListener s){listeners.add(s);}

  public void removeSpielListener(SpielListener s){listeners.remove(s);}

  protected void fireSpielEvent(){
    SpielEvent e=new SpielEvent(this, getSchritte(), isFertig());
    for(int i=0; i<listeners.size(); i++)
      ((SpielListener)listeners.get(i)).neuerSpielstand(e);
  }
}
```

5.10 Das Hauptprogramm

Da wir bei unseren „Aufräumarbeiten“ die Klasse `KrugSpiel` von allem befreit haben, was dem Namen nach nicht hinein gehört, brauchen wir jetzt ein eigenes Hauptprogramm, das die im Spiel benötigten Klassen erzeugt und miteinander in Beziehung setzt („verdrahtet“). Dies wird geleistet in der `main`-Methode der Klasse `Main`, die selber keine weitere Funktionalität zur Verfügung stellt.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Main{
  public static void guiAufbau(Container c){
    KrugSpiel ks=new KrugSpiel(
      new Krug[]{new See(), new Krug(8,0), new Krug(5,0), new Krug(3, 0)},
      new double[]{Double.POSITIVE_INFINITY, 4, 4, 0}
    );
    //erzeuge den Reset-Button
    ResetButton reset=new ResetButton("Startsituation", ks);
    c.add(reset, BorderLayout.NORTH);
    //erzeuge die Anzeige der benötigten Schritte
    AnzeigeLabel al=new AnzeigeLabel();
    ks.addSpielListener(al);
    c.add(al, BorderLayout.SOUTH);
    //Erzeuge die Matrix
    int anz=ks.getAnzahl();
    JPanel p=new JPanel(new GridLayout(anz, anz+1));
    c.add(p, BorderLayout.CENTER);
    for(int zeile=0; zeile<anz; zeile++){
      for(int spalte=0; spalte<anz; spalte++)
        if(zeile==spalte){
          TransferLabel tl=new TransferLabel();
          ks.getKrug(zeile).addTransferListener(tl);
          p.add(tl);
        } else{
          ProcessButton pb=new ProcessButton(ks,zeile,spalte);
          p.add(pb);
        }
      JLabel soL=new JLabel(ks.getSoll(zeile)+"");//+"" kleiner Trick!
      soL.setHorizontalAlignment(JLabel.CENTER);
      p.add(soL);
    }
    ks.resetAllTransfers();
  }

  public static void main(String[] args){
    JFrame f=new JFrame("Krugspiel");
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    Container c=f.getContentPane();//hat BorderLayout
    guiAufbau(c);
    f.pack();
    f.show();
```

```
    }
}
```

5.11 Das Programm in einem Stück weitergeben

Irgendwann kommt der Moment, da man sein Programm an andere Leute verteilen will. Diese Leute müssen auf ihrem Rechner mindestens das aktuelle JRE installiert haben (wie das geht, steht auf Seite 3). In den restlichen Abschnitten dieses Kapitels werden verschiedene Komfortstufen bereitgestellt, die alle ausbaubar sind. Für den Anfang reichen aber die hier geschilderten Vorgehensweisen.

Im Moment besteht unser Programm aus mehreren `class`-Dateien, die sich sauber aufgeräumt in einem Verzeichnis befinden. Bei der Weitergabe könnte aber leicht eine vergessen werden oder verloren gehen. Deshalb packen wir alle diese Dateien in eine `jar`-Datei (**J**ava **Ar**chiv). Das Format ist dasselbe wie bei einer `zip`-Datei. Damit das Archiv weiß, welche `class`-Datei die beim Start benötigte `main`-Methode hat, wird noch automatisch eine sogenannte Manifest-Datei beigelegt. Das ist dann aber schon alles. Folgendes ist zu tun:

- Eine Textdatei `manifest.txt` erzeugen mit dem Inhalt

  ```
  Main-Class: Main
  ```

  Das `Main` nach dem Doppelpunkt ist der Name der Klasse mit der `main`-Methode und die hatten wir ja `Main.java` genannt, was uns jetzt kurz ein wenig durcheinander bringt. (Nach dem Eintrag muss übrigens einmal Enter gedrückt worden sein. Ohne Zeilenschaltung geht es nicht.)

- Eine Konsole öffnen und in das Verzeichnis gehen, wo `manifest.txt` und die `class`-Dateien liegen.

- Mit dem Befehl `c:\Programme\java\bin\jar.exe cmf manifest.txt Spiel.jar *.class` wird das `jar`-Tool aufgerufen, es wird ihm aufgetragen, eine Datei (**f**ile) zu erzeugen (**c**reate) und eine **m**anifest-Datei beizufügen. Die entstehende Datei soll `Spiel.jar` heißen und sie soll (außer der Manifestdatei) alle Dateien der Art `*.class` aufnehmen.

- Die `jar`-Datei kann nun auf einen beliebigen java-fähigen Rechner übertragen und gestartet werden. Auf unserem Rechner geht das mit `c:\Programme\java\bin\javaw -jar Spiel.jar`

  Es wird also wieder ganz normal `java` oder `javaw` aufgerufen mit dem Schalter `-jar`. Der Rest geht automatisch.

5.12 Ein Applet in eine HTML-Seite einbinden

Das ist für den Benutzer schon angenehmer, weil er nicht wissen muss, wie man eine Konsole aufmacht und `java` startet. Ein Durchschnittsbenutzer ist ja gewöhnlich nicht einmal in der Lage, ein Programm überhaupt zu finden, wenn es auf der Festplatte so versteckt ist wie `java.exe`.

Stattdessen gibt man ihm eine Diskette mit einer `html`- und einer `jar`-Datei. (Was diesmal in der `jar`-Datei drin sein muss, wird weiter unten besprochen.) Wenn er nun mit seinem Browser die `html`-Datei anschaut, läuft das Applet automatisch.

Was ist nun aber ein Applet? So etwas wie ein Fenster nur ohne Rahmen, weil es ja im Browser angezeigt wird. Die Programmierung eines Applets geht fast genau so wie die Programmierung eines eigenständigen `JFrame`s. Für den Schulgebrauch könnte man sich geradezu auf den Standpunkt stellen, dass es reicht, ausschließlich Applets zu programmieren.

Um die aufkommende Freude nicht im Keim zu ersticken, fangen wir diesmal hinten an und nehmen an, dass wir schon alles programmiert und in eine `jar`-Datei verpackt haben[12]. Dann brauchen wir nur noch in eine `html`-Datei dort, wo das Spiel erscheinen soll, die folgenden Zeilen (ein so genanntes `applet`-Tag) einfügen:

```
<applet code="SpielApplet" archive="Spiel.jar" width=200 height=100>
  <param name="einparameter"     value="einwert">
  <param name="andererparameter" value="andererwert">
</applet>
```

[12] Aus der Sicht des vorangehenden Abschnitts fehlt noch die Datei `SpielApplet.class`.

In der selben `html`-Datei bringen wir auch sonst noch alles unter, was den Leser zu unserem Spiel interessieren könnte, z. B. die Spielregeln.

Die `param`-Zeilen dienen dazu, dem Applet Parameter mitzugeben, was aber in unserem Beispiel nicht verwendet wird. (Wie man innerhalb eines Applets auf Parameter zugreift, wird im Tutorial auf der CD `//books/tutorial.zip` ausführlich erklärt.) Bemerkenswert ist, dass unsere `jar`-Datei nun als selbständig lauffähiges Programm *und* als Applet verwendet werden kann.

Wir müssen uns jetzt natürlich auch einmal ansehen, wie man ein Applet programmiert. Auf Grund der Vorarbeit ist nicht mehr viel zu tun.

```
import javax.swing.JApplet;

public class SpielApplet extends JApplet{
  public void init(){
    Main.guiAufbau(getContentPane());
  }
}
```

Ein `JApplet` ist im Aufbau einem `JFrame` sehr ähnlich. Beide haben einen Anzeigebereich für graphische Komponenten, den man mit `getContentPane` bekommt. Das was man bekommt, ist beide Male ein `Container` mit `BorderLayout`. Da wir das Füllen dieses `Containers` in der Klasse `Main` schon ausgelagert haben in eine eigene Methode `guiAufbau`, können wir diese Methode jetzt wieder verwenden und diesmal das `JApplet` statt des `JFrame`s mit den graphischen Komponenten befüllen lassen.

Dass dies innerhalb der Methode `init` geschieht, ist eine Eigenheit von Applets. Sie haben drei besondere Methoden:

- Die `init`-Methode wird vom Browser genau einmal aufgerufen, wenn er die Seite mit dem Applet das erstemal lädt. In diese Methode schreibt man üblicherweise den Code zum optischen Aufbau des Applets.
- Die `start`-Methode wird vom Browser kurz vor der Darstellung des Applets aufgerufen, damit es etwaige spezielle Aktionen starten kann. In dieser Methode kann man z. B. weitere Threads oder ein Dauermusikstück starten. Unser Applet hat keine Aktionen, die zu starten wären, deshalb überschreiben wir `start` nicht sondern lassen es leer, wie es im Vorfahren definiert ist.
- Die `stop`-Methode wird vom Browser aufgerufen, wenn das Applet seine speziellen Aktionen wieder stoppen soll. Das ist z. B. dann der Fall, wenn der Benutzer die Seite mit dem Applet verlässt. Ein ordentliches Applet sollte dann nämlich wieder Ruhe geben. (Kehrt der Anwender zur Seite zurück, wird nur die `start`-Methode wieder aufgerufen.)

5.13 Applet oder Application?

Ein selbständiges Programm mit einer `main`-Methode nennt man Application. Aber wozu braucht man so etwas überhaupt, wenn es doch schon Applets gibt? Ihre Existenzberechtigung haben beide Bauformen auf Grund ihrer speziellen Eigenheiten.

- Ein Applet ist nur innerhalb einer `html`-Seite lauffähig. Eine Application braucht keine solche Seite.
- Ein Applet hat sehr eingeschränkte Rechte. Es kann nicht auf die Festplatte oder den Drucker zugreifen[13] und deshalb auch keinen Schaden anrichten. Eine Application darf alles und kann auch schaden.
- Ein Applet nutzt den Speicher, der dem Browser vom Betriebssystem zur Verfügung gestellt wurde. Bei ausuferndem Speicherbedarf kann ein Applet Probleme bekommen. Für eine Application gelten auch hier schwächere Einschränkungen.

In der Schule wird man in den meisten Fällen Applets bevorzugen, weil die Präsentation und Dokumentation „gleich nebenan“ im `html`-Dokument nieder gelegt werden kann. Schüler stellen ihre Erzeugnisse auch meist mit Stolz ins Internet. Soll ein Programm aber Dateien lesen, schreiben oder ausdrucken, so wird man es als Application ausführen.

[13] Außer man erteilt diese Rechte ausdrücklich durch eine so genannte Policy.

6 Übung: Eine verteilte Anwendung

In diesem Kapitel werden mehrere mächtige Konzepte von Java in einem Projekt zusammen geführt: Es sollen `Jobs` auf mehrere Rechner (Arbeitssuchende) im Netz verteilt werden. Die Ergebnisse werden beim Arbeitgeber gesammelt und ausgewertet.

Das ganze Projekt ist nicht groß aber ziemlich komplex. Sie finden es auf der CD unter `//code/rmi/`. Am besten kopieren Sie das ganze Verzeichnis auf mindestens zwei Rechner im lokalen Netzwerk (es geht auch in zwei verschiedenen Verzeichnissen eines Rechners, aber dann ist die Arbeitsweise nicht beeindruckend). Öffnen Sie die Dateien `arbeitssuchend` und `arbeitgebend` (in Windows mit `.bat`) und passen Sie sie an Ihr Netzwerk an (die codebase meint immer den eigenen Rechner). Geben Sie beim einen `arbeitssuchend` an der Konsole ein, beim anderen `arbeitgebend`. (Falls Sie nur einen Rechner verwenden, öffnen Sie zwei Konsolen). Letzterer sollte eine Menge von Primzahlzwillingen auflisten.

Wenn man den Schülern alle Details dieses Projekts vermitteln will, handelt es sich um ein ordentliches Problem zur objektorientierten Programmierung mit mehreren Klassen und Interfaces. Man kann sich jedoch auch auf Teile beschränken und etwa nur verschiedene `Jobs` programmieren. Die genaue Funktionsweise bleibt dann verborgen.

6.1 Was RMI automatisch leistet

RMI muss eine Menge Leistungen erbringen, um die sich der Programmierer nicht kümmern will. Deshalb sind die folgenden Erläuterungen recht umfangreich und scheinen kompliziert. Die Programmierung ist dann aber sehr einfach. Zur Erleichterung des Verständnisses sollen erst einmal einige Begriffe geklärt werden, auf die bisher noch nicht eingegangen wurde:

- Klassen werden in Java *dynamisch gebunden*, d. h. sie werden erst zur Laufzeit zu einem funktionierenden Ganzen vereinigt. Für das Kompilieren reicht es deshalb aus, `Job` in einem Interface zu definieren. Die Rechner, die nachher einen `Job` ausführen, können sogar schon gestartet werden, auch wenn es noch gar keine `Job`-Klasse gibt (also eine Klasse, die `Job` implementiert). Solche können auch noch viel später programmiert werden und sind trotzdem innerhalb des entworfenen Konzeptes lauffähig.
- Durch den Einsatz von *RMI* (Remote Method Invocation) entsteht für den Programmierer an entscheidender Stelle der Eindruck, als würde er nur einen einzigen Rechner programmieren. Die Übergabe von Parametern an die Methoden entfernter Objekte sowie die Rückgabe der Ergebnisse wird von RMI übernommen.

  Die ganze im Hintergrund sich abspielende Funktionalität bekommt `ArbeitsSuchend`, weil es ein Nachkomme von `UnicastRemoteObject` ist und ein Interface implementiert, das Nachkomme des Interfaces `Remote` ist. Außerdem braucht man auf dem Arbeitgeber-Rechner eine Art Statthalter, mit dem stellvertretend zu kommunizieren ist. Der Statthalter wird mit dem gerade für diesen Zweck vorhandenen Compiler `rmic` aus der Klasse `ArbeitsSuchend` erstellt und heißt automatisch `ArbeitsSuchend_Stub`.

  Der Sinn dieses Konstruktes ist letztendlich, dass der Arbeitgeberrechner mit dem Stub spricht, wenn er dem Arbeitssuchenden etwas mitteilen will. Der Stub sorgt dafür, dass die Kommunikation über das Netzwerk funktioniert. Wenn man bedenkt, dass nicht nur Strings sondern evtl. ganze Objekte übertragen werden müssen, versteht man, warum solch ein technischer Aufwand nötig ist. Man beachte, dass nur das Verständnis des ganzen schwierig ist. Die Programmierung ist ziemlich einfach, wie wir nachher noch sehen werden.
- Nun zur Übertragung von Objekten. Objekte haben Attribute und Methoden. Die Methoden sind in der Klasse (`class`-Datei) niedergelegt, die Attribute hat aber jedes Objekt (genauer: jede Instanz) selber. Wer die Klasse hat, braucht nur noch die Instanzattribute, um daraus wieder ein Objekt herstellen zu können. Der Mechanismus, der die Instanzattribute herausholt und entweder auf einen Datenträger schreibt, oder, wie hier, über das Netz überträgt, heißt *Serialisation*. Die Wiederherstellung eines Objektes aus solchen Daten und der `class`-Datei, nennt man *Deserialisation*. Der Mechanismus ist ein Kapitel für sich; eine sehr schwierige Aufgabe wurde da höchst beeindruckend

gelöst. Im Moment reicht es aber, die Existenz eines solchen Mechanismus zur Kenntnis zu nehmen und sich zu freuen, dass das alles automatisch und problemlos geht. (Serialisierbar sind übrigens alle Objekte, die das Interface `Serializable` implementieren. Das ist nur bei einer sehr kleinen Zahl von Objekten nicht der Fall und dort gibt es ernsthafte Gründe dagegen. Man kann vereinfachend annehmen, dass im wesentlichen alles serialisierbar ist.)

Abbildung 6.1 zeigt, was der Reihe nach passiert, wenn der entfernte Rechner B in Bereitschaft geht und auf Anfragen wartet. Im Bild schon gestartet ist eine sogenannte RMI-Registry (grün), das ist ein Namensserver, der auf Anfrage serialisierte Daten überträgt. Dazu gehören Stubs (Kommunikations-Stellvertreter), Parameter und Rückgabewerte von Methoden.

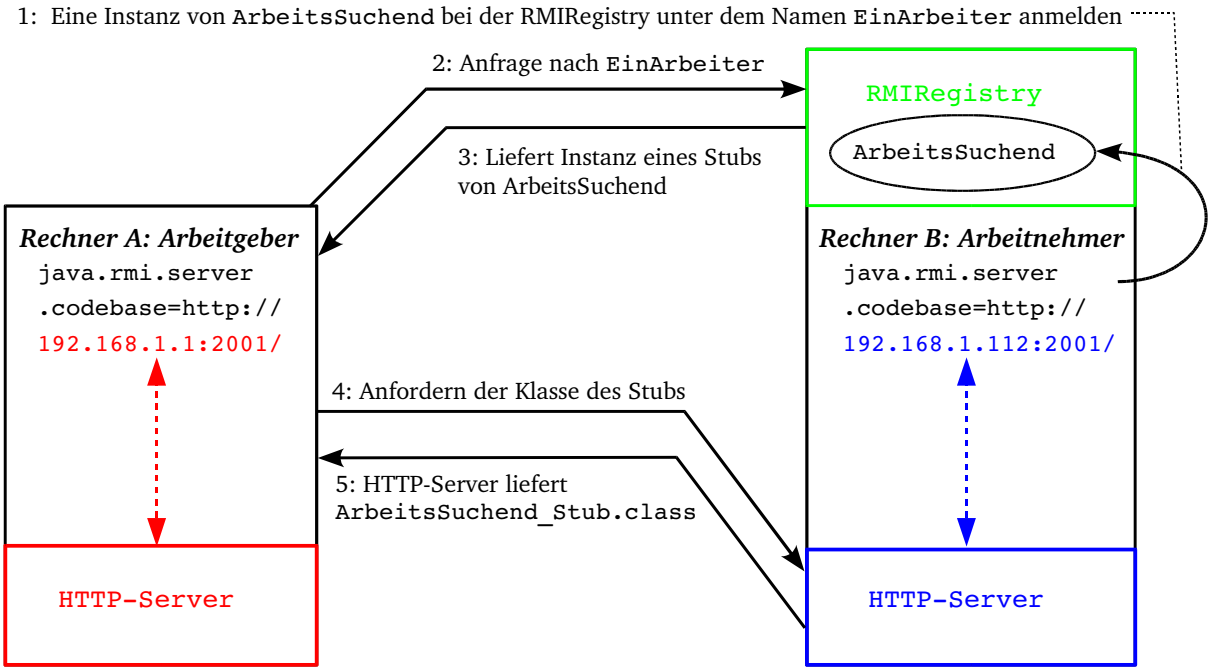

Abbildung 6.1: Erster Teil: Aufbau einer RMI-Kommunikation

1. Eine Instanz der Klasse `ArbeitsSuchend` wird erzeugt und bei der RMI-Registry unter dem Namen `EinArbeiter` angemeldet. Beim Programmstart kann man für die JVM (Java Virtual Machine) die Umbegungsvariable `java.rmi.server.codebase` angeben. In unserem Beispiel wird das nicht nötig sein, aber vorerst ist das Verständnis des allgemeinen Falles einfacher. Die RMI-Registry bekommt diese Adresse mitgeteilt.

2. Der Arbeitgeberrechner A fordert bei der RMI-Registry einen `ArbeitsSuchenden` an um ihn in einer Variable des Typs `Arbeiter` abzulegen. (`Arbeiter` ist das oben erwähnte Interface, das von `ArbeitsSuchend` implementiert wird. Es definiert die Methoden von `ArbeitsSuchend`, die aus der Ferne zur Verfügung stehen.) Die Methoden des Objektes, das A dann bekommt, werden von A aufgerufen, als wäre es ein Objekt im eigenen Speicher.

3. Die RMI-Registry schickt aber „nur“ einen serialisierten Stub der Klasse. (Das macht nichts, weil dieser ja alle Methoden aus dem Interface `Arbeiter` zur Verfügung stellt.) Um aus den serialisierten Stub-Daten wieder ein Stub-Objekt herstellen, braucht A die Datei `ArbeitsSuchend_Stub.class`. In unserem Beispiel wird A diese Datei im eigenen Verzeichnis haben, und die beiden nächsten Schritte werden nicht nötig sein. Im allgemeinen Fall kann es aber sein, dass die Stub-Klasse nicht vorliegt. Also muss sie beschafft werden.

4. Da A mit dem serialisierten Stub auch dessen Codebase erhält, kann er sich an diese wenden und die `class`-Datei anfordern. Dort muss ein HTTP- oder FTP-Server laufen, der diese liefern kann. Gewöhnlich kann man nicht davon ausgehen, dass auf jedem an unserem Mechanismus beteiligten Rechner ein Webserver läuft. Deshalb müssten wir selber einen aufsetzen. Wie schon gesagt, entfällt aber in unserem Fall der blaue Teil.

5. Die `class`-Datei des Stubs (und evtl. noch weitere benötigte Klassen) werden an A übertragen.

Damit hat A alles was nötig ist, um bei B Methoden aufzurufen. Schauen wir uns in Abbildung 6.2 noch an, was dabei passiert.

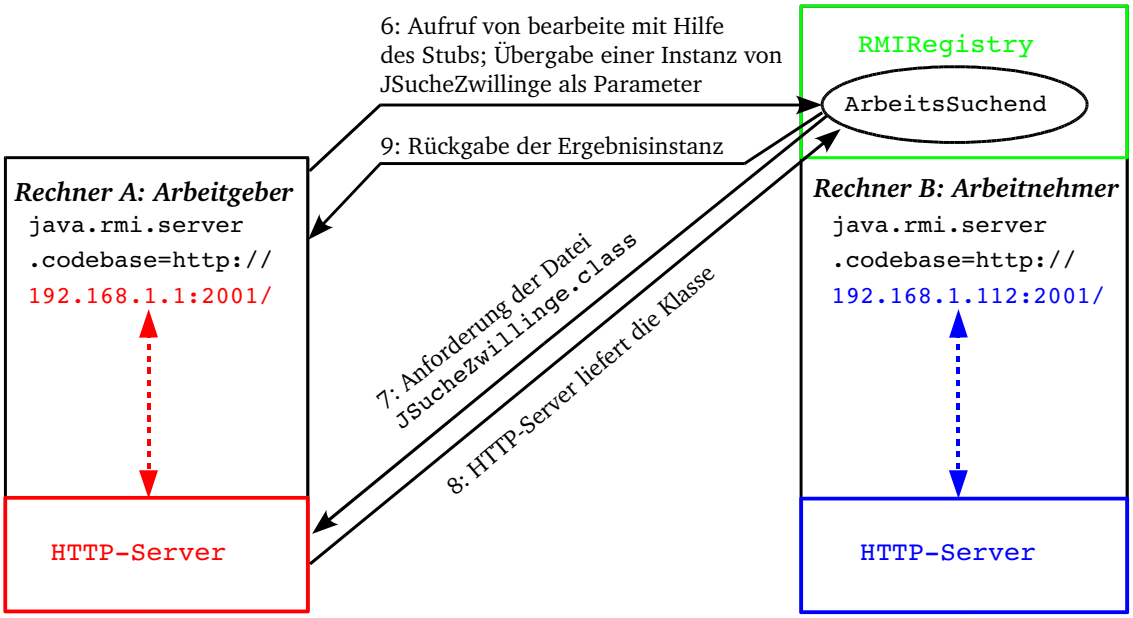

Abbildung 6.2: Zweiter Teil: Aufruf einer entfernten Methode

6. A ruft bei seiner Stub-Instanz (die in einer Variable des Typs `Arbeiter` abgelegt ist) die in `Arbeiter` definierte entfernte Methode `bearbeite` auf und gibt ihr als Parameter eine Klasse mit, die `Job` implementiert, z. B. ein `JSucheZwillinge`. Da diese Klasse `Job` implementiert, *ist* sie ein `Job`. Übertragen wird natürlich wieder nur das Ergebnis der Serialisierung der bei A vorliegenden Instanz von `JSucheZwillinge`.

   An dieser Stelle könnte die Frage aufkommen, wie sich B verhält, wenn von mehreren Rechnern gleichzeitig oder kurz hintereinander `Jobs` zur Bearbeitung angeliefert werden. Auf B wurde zwar nur ein `ArbeitsSuchende`r erzeugt und an der RMIRegistry angemeldet, aber der erzeugt für jeden Arbeitsauftrag einen eigenen Thread. Mehrere Arbeitsaufträge werden also quasiparallel bearbeitet.

7. Der `ArbeitsSuchende` auf B braucht nun aber auch die Datei `JSucheZwillinge.class`, und die hat er nicht, weil wir uns ja vorbehalten haben, die `Job`-Klassen erst später zu programmieren, wenn B schon lange läuft. Jedenfalls wollen wir nicht mit jedem neuen Job zu B laufen und die Klassendatei dort ablegen.

   Wo soll er nun nach dieser Datei suchen? Beim Programmstart auf A haben wir glücklicherweise auch die Möglichkeit, die Umgebungsvariable `java.rmi.server.codebase` anzugeben. Die RMI-Registry kennt diese und hat sie auch B mitgeteilt, als dessen (entfernte) Methode `bearbeite` aufgerufen wurde. B fragt also dort nach der gewünschten Datei.

8. Auf A lassen wir einen kleinen Webserver auf Port 2001 laufen, der diese Anfrage befriedigt. (Die beiden für den Webserver benötigten Klassen findet man in der RMI-Dokumentation von Sun.) Wir konnten also auf den blauen Webserver verzichten, nicht aber auf den roten.

9. Wenn der `ArbeitsSuchende` fertig ist, wird das Ergebnisobjekt des `Jobs` an den Arbeitgeber zurück geliefert.

6.2 Zusammenwirken von Interfaces und Klassen

In diesem Abschnitt wird der ganze Mechanismus noch einmal aus Programmierersicht besprochen. Die Dateien befinden sich auf CD im Verzeichnis `//code/rmi` und sind ausreichend dokumentiert. Die folgenden Dateien müssen auf allen Rechnern verfügbar sein, die Arbeit annehmen oder verteilen:

- Im Interface `Job` wird vereinbart, dass sich eine Klasse dann `Job` nennen darf, wenn sie die parameterlose Methode `losgehts` implementiert, die ein beliebiges `Object` als Ergebnis zurück liefert. `Job`

ist eine Erweiterung von `Serializable`. Klassen, die `Job` implementieren, müssen nämlich immer auch `Serializable` sein.

Eine Instanz von `Job` muss schon alle für den `Job` nötigen Parameter besitzen und für die Abarbeitung mitbringen. Die Parameter sollen nämlich Parameter des `Jobs` sein, nicht solche der Methode `losgehts`.

- Im Interface `Arbeiter` wird definiert, welche Methoden der (oder die) entfernten Objekte anbieten. `Arbeiter` muss Nachkomme von `Remote` sein, damit der RMI-Mechanismus die Zusammenhänge versteht. In unserem Beispiel wird definiert, dass ein `Arbeiter` die Methode `bearbeite(Job j)` haben muss.
- `ArbeitsSuchend` ist eine Klasse, die das Interface `Arbeiter` implementiert. Ein `ArbeitsSuchender` ist also ein `Arbeiter`, weil er die verlangte Methode hat. Interessant ist, was diese Methode tut! Sie ruft nur die Methode `losgehts` des `Jobs` auf, den Sie als Parameter erhält. Was in dieser Methode zu tun ist, wird sich erst zur Laufzeit herausstellen. Es wird jedenfalls auf dem Rechner getan, der den `ArbeitsSuchenden` besitzt (im Bild also Rechner B).

  Bei der Kompilierung von `ArbeitsSuchend` entsteht die Datei `ArbeitsSuchend.class`. Einzig von dieser Datei wird aber auch noch ein Stub gebraucht. Durch den Aufruf `rmic ArbeitsSuchend` entsteht die Datei `ArbeitsSuchend_Stub.class`. (Es entsteht auch `ArbeitsSuchend_Skel.class`, ein Relikt aus alten RMI-Zeiten, das gelöscht werden darf.)

  Damit nicht zu viele Klassen entstehen, wurde in `ArbeitsSuchend` auch noch eine `main`-Methode untergebracht, die die beim Start einmal anfallende Verwaltungsarbeit erledigt: RMI-Registry erstellen, Instanz von `ArbeitsSuchend` erzeugen und anmelden. Ein anderer Ort wäre genauso gut möglich gewesen.
- `Arbeiter.policy` ist eine Textdatei, die die Rechte der JVM auf Rechner B einschränkt. Die auszuführende Arbeit könnte dem Rechner ja schaden. Für den Anfang ist aber alles erlaubt.
- `arbeitssuchend` oder `arbeitssuchend.bat` sind Batch-Dateien mit dem Aufruf der JVM, der ja hier wegen der Umgebungsvariablen und der Policy-Datei und den Parametern nicht mehr ganz einfach aussieht.

In unserem Beispiel kann jeder Rechner Arbeit annehmen (Rechner B in den Abbildungen). Wenn er auch Arbeit verteilen soll (Rechner A), so werden weitere Dateien gebraucht.

- Eine Klasse mit `main`-Methode, die einen Webserver startet, sich nach `ArbeitsSuchenden` umsieht, das zu erledigende Problem in mehrer Teile zerlegt und verteilt und die Ergebnisse einsammelt. Ein Beispiel für eine solche Klasse ist `PSucheZwillinge`, die einen größeren Zahlenbereich von mehreren Rechnern nach Primzahlzwillingen durchsuchen lässt.
- Ein `Aufpasser` sorgt dafür, dass der Arbeitgeber nicht selber auf die Abarbeitung einer Teilaufgabe warten muss. Die nächste Teilaufgabe soll ja schon an den nächsten Rechner gegeben werden, bevor der erste fertig ist. Erst dann hat man einen Geschwindigkeitsvorteil. Der Arbeitgeber startet deswegen für jede Teilaufgabe einen eigenen Wartethread, einen `Aufpasser`.

  Der Aufpasser bekommt bei seiner Erzeugung einen `Job`, die Adresse eines `ArbeitsSuchenden`, dem er diesen `Job` geben kann und ein Objekt, das am Ergebnis dieses `Jobs` interessiert ist. Das wird meist der Arbeitgeber selber sein. Ein Interessent muss jedenfalls als solcher erkennbar sein, indem er die Methode `nimmErgebnis` aus dem Interface `Aufpasser.Interessent` implementiert. Der Aufpasser wartet auf das Ergebnis des `Jobs` und gibt es dann dem Interessenten. Danach stirbt der Aufpasserthread.
- Mindestens eine Klasse, die das Interface `Job` implementiert, z. B. `JSucheZwillinge`.
- `arbeitgebend` oder `arbeitgebend.bat` sind wieder Batch-Dateien mit dem etwas komplizierten Aufruf der JVM.

6.3 Diskussion

Dieses letzte Kapitel sollte noch einmal die Leistungsfähigkeit von Java aufzeigen. Dass die dahinter steckende Technologie aufwändig und schwierig zu durchschauen ist, liegt an der Komplexität des Problems und lässt sich nur dadurch vermeiden, dass man sich nicht hinein vertieft. Mit Schülern muss man die Hintergründe aber auch gar nicht bis ins letzte Detail erarbeiten. Wenn ein grober Überblick über die

wenigen wichtigen Zeilen eines der Beispiele gegeben ist, kann man eigene Projekte ganz ähnlich entwerfen; der Arbeitnehmer-Teil ist jedenfalls allgemein genug gehalten um viele Aufgaben zu lösen. Man muss nur ein Arbeitgeberprogramm und evtl. mehrere Jobs programmieren. Das Projekt ist übrigens völlig symmetrisch. Jeder Rechner kann einem anderen einen Job geben und gleichzeitig den Job eines anderen abarbeiten.

Denkbare Anwendungsmöglichkeiten sind Netzwerkspiele, Netzwerkabfragen oder alle Probleme, die einen einzigen Rechner in Zeit- oder Platznot bringen, wie etwa größere Simulationen. Da der Arbeitnehmer als Application läuft, können seine Rechte sehr großzügig sein (im Beispiel hat er die maximal möglichen Rechte). Damit sind auch Fernsteueraufgaben möglich, wie etwa das Löschen aller Verzeichnisse einer Arbeitsgruppe. Was passiert, hängt nur vom gegebenen Job ab.

A Die Grundkonstrukte von Java

In diesem Kapitel werden die wichtigsten Elemente der Sprache Java vorgestellt. Die Aufstellung ist nicht vollständig und für einen Programmieranfänger ohne Unterweisung auch nicht geeignet, das Programmieren zu erlernen. Jemandem, der irgendeine (imperative) Programmiersprache beherrscht, sollte der Text aber die kleinen Eigenheiten von Java schnell vermitteln können.

Java ist eine Sprache mit sehr präzisen Regeln, die man aber nicht alle sofort kennen muss um die ersten Programme schreiben zu können. Im Zweifelsfall wird man immer im definierenden „Gesetzestext“ nachsehen; das ist die von *Sun Microsystems* formulierte *Language Specification*. Als einführender Text ist diese aber zu anspruchsvoll. Deshalb wurden viele Bücher über Java geschrieben, die die Sachverhalte sympathischer aber nicht bis in die letzten Einzelheiten darstellen. Dieser Abschnitt ersetzt kein solches Buch, reicht aber vielleicht für das schnelle Nachschlagen.

Das Lesen trägt nur dann Früchte, wenn man schon Fragen hat, und diese tauchen auf, wenn man schon eine Aufgabe hat. Nehmen Sie also am besten Ihre alten Pascal-Übungen und versuchen Sie sie in Java umzuschreiben. Wie man das systematisch bewerkstelligt, finden Sie in Kapitel 3 und Anhang D.

A.1 Ein Beispiel mit Parametern

Das folgende kleine, ziemlich sinnlose Beispielprogramm nimmt zwei Eingaben entgegen und entscheidet daraus, wie die Ausgabe auszusehen hat. Das Ganze spielt sich auf der Kommandozeile ab.

Nach dem Kompilieren kann es gestartet werden mit `...\java Aussage Wurst lecker` und wird den Text `Wurst schmeckt lecker.` produzieren. Bemerkenswert ist also die Tatsache, dass `Aussage` ein Argument von `java` ist und die letzten beiden Worte Argumente von `Aussage` sind, die als zwei Array-Elemente an das Java-Programm übergeben werden.

```
public class Aussage{
  /* Das Folgende ist die Deklaration eines konstanten
     Stringarrays. (dies hier ist ein mehrzeiliger Kommentar) */
  static final String[] IGITT={//ein Array von Strings, direkt gegeben
    "Karotte", "Koriander", "Melone"
  };

  static boolean isBäh(String s){
    for(int i=0; i<IGITT.length; i++){
      if(s.equals(IGITT[i])) return true;
    }
    return false;
  }

  public static void main(String[] viele){
    if(isBäh(viele[0])) //nur eine Anweisung im then-Zweig
      System.out.println(viele[0]+"? Ist ja schrecklich!");
    else
      System.out.println(viele[0]+" schmeckt "+viele[1]+".");
  }
}
```

Das Programm muss, wie schon erwähnt, in reiner Textform in der Datei `Aussage.java` abgespeichert werden. Zeilen und Einrückungen dienen der Übersichtlichkeit. Blöcke werden mit geschweiften Klammern gebildet; der größte Block ist die Klassendefinition selbst. Es gibt zwei Arten von Kommentaren: `//` bis Zeilenende bilden einen Kommentar. Über mehrere Zeilen sich erstreckende Kommentare schließt man zwischen `/*` und `*/` ein. Java unterscheidet Groß- und Kleinschreibung.

Eine Klasse hat meist Attribute (fields) wie `IGITT`[1] und Methoden wie `main(...)` und `isBäh(...)`. (In nicht objektorientierten Sprachen werden letztere auch Prozeduren und Funktionen genannt). Bezeich-

[1] `IGITT` ist auch ein Array, was aber nicht ausschlaggebend ist für den Begriff „field“.

ner wie `isBäh`[2] dürfen alle Unicode-Zeichen enthalten, die in irgend einer Sprache Buchstaben oder Zahlen sind. Für Klassennamen sollte man aber auf Umlaute und sonstige Nicht-ASCII-Buchstaben verzichten, weil man nicht sicher sein kann, dass das Betriebssystem den Dateinamen nicht umcodiert.

Alle Attribute und Methoden sind `static`, weil wir im Moment noch keine Objekte programmieren. Damit ein Programm aufgerufen werden kann, muss die zugehörige Klasse als Einstiegspunkt die `static`-Methode `main` haben. Diese nimmt beliebig viele Strings entgegen, die alle in einem String-Array abgespeichert werden (oben `viele` genannt). Die Indizierung beginnt in Java immer mit 0. `IGITT.length` hat aber den Wert 3, deshalb die Verwendung von <. Die Tatsache, dass es sich nicht nur um einen String, sondern um eine ganze Menge Strings handelt, wird durch die eckigen Klammern ausgedrückt.

Die Attribute `static`, `final` und `public` werden später erklärt. `boolean` bedeutet, dass die Methode einen Wahrheitswert zurückliefert, `void` bedeutet, dass die Methode nichts zurückliefert (Prozeduren sind Funktionen, die nichts zurück liefern). Die ungewöhnliche `for`-Schleife wird noch erklärt.

A.2 Bezeichner, Schlüsselwörter, Literale

Bezeichner (Identifier) sind Namen. Java unterscheidet Groß- und Kleinschreibung. Namen beginnen immer mit einem Buchstaben, können aber auch Zahlen und den Unterstrich _ enthalten. Buchstaben sind übrigens in Unicode recht weit gefasst. Im Beispiel sind `main`, `String`, `viele` Namen, möglich wäre aber auch $\tau\varepsilon\chi$. Zu beachten ist die Konvention, dass Klassen und Interfaces mit Großbuchstaben beginnen, Attribute und Methoden aber mit Kleinbuchstaben. Einzelne Großbuchstaben dürfen in den Bezeichnern dort verwendet werden, wo ein neues Wort beginnt, z. B. in `isBäh`. Konstanten werden vollständig in Großbuchstaben benannt.

Schlüsselwörter (Keywords) sind Wörter, die nicht mehr verwendet werden dürfen, weil sie schon eine besondere Bedeutung in der Sprache haben. In Java sind dies die Wörter in Tabelle A.1.

Tabelle A.1: Schlüsselwörter in Java

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

Literale sind das, was man hinschreibt, wenn man Werte direkt eingibt, also Strings in Anführungszeichen, Zahlen, die Wahrheitswerte `true` und `false`, sowie das Symbol für die Nichtzuordnung von Objekten `null`.

A.3 Datentypen

In Java gibt es zwei Sorten von Datentypen: die primitiven Werttypen aus Tabelle A.3 und die Referenztypen. Der maßgebliche Unterschied ist, dass Werttypen wegen ihrer geringen Größe von höchstens 8 Byte bei Aufrufen von Methoden selber übergeben werden. Von Objekten, die durch Klassen, Interfaces oder Arrays definiert sind, wird hingegen nur ein Zeiger übergeben – das Objekt selber bleibt in seiner vollen Größe dort im Speicher liegen, wo der Zeiger hin zeigt. Für den Programmierer ist es wesentlich zu wissen, mit welcher Art er es zu tun hat, weil Referenzobjekte nach der Rückkehr von Methoden unter Umständen verändert sein können, Wertvariablen aber nicht.

In Java werden alle Variablen initialisiert, sobald sie eingerichtet worden sind. Zahlen sind anfangs immer 0, Wahrheitswerte sind `false`, Referenzen sind `null`. Der Compiler versucht zwar trotzdem den Programmierer zu zwingen, Werte zuzuweisen, bevor diese verwendet werden, es gibt aber Situationen, in denen der Compiler nicht nachprüfen kann, ob schon eine Zuweisung stattgefunden hat.

[2] Der Leser verzeihe mir das niedrige sprachliche Niveau aber der Name sollte kurz sein, aus zwei Worten bestehen und einen Umlaut enthalten.

Integertypen

Es gibt die ganzzahligen, vorzeichenbehafteten Typen `byte`, `short`, `int`, `long` (siehe Tabelle A.3). Berechnungen mit Integerwerten erfolgen mit mindestens 32 Bit Genauigkeit, und sogar mit 64 Bit, wenn mindestens ein `long` beteiligt ist. Ein `long`-Literal gibt man mit nachfolgendem `L` an. Der Compiler nimmt evtl. nötige Umwandlungen in die größeren Typen automatisch vor, weil keine Information verlorengehen kann. Will man in einen kleineren Typ umwandeln, könnte dies mit Datenverlust verbunden sein, weil einfach nur die unteren Bits behalten werden. So eine Umwandlung muss deshalb vom Programmierer angefordert werden. Man nennt das einen Cast. (`byte b=1+1` führt zu einer Fehlermeldung, besser `byte b=(byte)(1+1)`. Üblicherweise verwendet man aber `byte` nur in Arrays. Im genannten Fall nimmt man immer `int`. Der Cast ist dann nicht nötig.)

Der Typ `char`

Ein `char` ist eine 16-Bit-Zahl, repräsentiert aber ein Unicodezeichen. Literale werden in einfachen Anführungszeichen angegeben (z. B. `char bst='a';`). Mögliche Literale sind natürlich auch die Unicodesequenzen (siehe Tabelle A.3) und einige aus C bzw. ASCII bekannte Escape-Sequenzen, die in Tabelle A.2 aufgeführt sind.

Tabelle A.2: einige (Unicode-)Escape-Sequenzen

Escape	Unicode-Escape	Bedeutung
`\b`	`\u0008`	BS (Backspace)
`\t`	`\u0009`	HT (horizontal Tab)
`\n`	`\u000a`	LF (Line Feed)
`\f`	`\u000c`	FF (Form Feed)
`\r`	`\u000d`	CR (Cariage Return)
`\"`	`\u0022`	" (double quote)
`\'`	`\u0027`	' (single quote)
`\\`	`\u005c`	\ (backslash)
`\000` bis `\377`	`\u0000` bis `\u00ff`	oktaler Wert

Bei Rechenoperationen mit `char` werden diese in ein `short` konvertiert. Dabei werden einfach die Bits kopiert, wodurch aus dem stets positiv interpretierten `char` negative Zahlen entstehen können.

Gleitpunkttypen

Java kennt die Typen `float` und `double` (siehe Tabelle A.3). Der Standardtyp ist `double`, alle Funktionen der Mathematikbibliothek haben ihn als Ergebnis. Braucht man das Funktionsergebnis als `float`, so benutzt man einen Cast; braucht man ein `float`-Literal, so hängt man an die Zahl ein F an. (Es besteht keine Gefahr der Verwechslung mit dem F der Hexadezimaldarstellung, weil diese bei Gleitpunktzahlen nicht verwendet werden kann.) Das Ergebnis einer Rechnung ist nur dann ein `float`, wenn alle Operanden diesen Typ haben und gegebenenfalls Integertypen (außer `long`) vorkommen.

Tabelle A.3: Primitive Typen in Java

Typ	Größe	Wertebereich	Default
`byte`	8 bit	$-128 \ldots 127$	`0`
`short`	16 bit	$-32768 \ldots 32767$	`0`
`int`	32 bit	$-2147483648 \ldots 2147483647$	`0`
`long`	64 bit	$-9223372036854775808 \ldots 9223372036854775807$	`0L`
`char`	16 bit	`\u0000`...`\uffff`	`\u0000`
`float`	32 bit	$\pm 3.40282e38$	`0.0F`
`double`	64 bit	$\pm 1.79769e308$	`0.0D`
`boolean`	1 bit	`false`, `true`	`false`

Arrays

In anderen Programmiersprachen oft Felder genannt, aber dieser Begriff wird in Java schon verwendet für Attribute. Arrays haben *einen* Namen aber *viele* Elemente, auf die über einen Index zugegriffen werden kann. Dieser geht bei n Elementen immer von 0 bis $n-1$. Der Index kann sich maximal im positiven Bereich eines `int` bewegen, wodurch ein Array also höchstens 2^{31} Elemente haben kann. Nach der Reservierung des nötigen Speicherplatzes kann die Größe nicht mehr verändert werden. Die Größe eines Arrays bekommt man mit `arrayname.length`.

Arrays sind Referenztypen, die als von der Klasse `Object` abgeleitet betrachtet werden können. Mehrdimensionale Arrays gibt es im strengen Sinne nicht, Arrays von Arrays sind aber erlaubt bis zu beliebiger Schachtelungstiefe.

```
//12 ints in einer Reihe werden angelegt und mit 0 initialisiert
int[] monatsTage = new int[12];
monatsTage[0] = 31;  //Zuweisung
int i = monatsTage.length;  //i = 12
//7 Strings anlegen und sofort initialisieren
String[] tagesKürzel = {"Mo","Di","Mi","Do","Fr","Sa","So"};
//Tabelle einiger Quadratzahlen als Array of Array
int[][] quad = new int[4][2];
quad[0][0] =  7; quad[0][1] = 49;
quad[1][0] =  3; quad[1][1] =  9;
quad[2][0] = -2; quad[2][1] =  4;
quad[3][0] =  9; quad[3][1] = 81;
```

Konstanten

Konstanten werden durch das Attribut `final` deklariert. Einer Konstanten kann nur einmal ein Wert zugewiesen werden. Versucht man ihn zu ändern, so bekommt man eine Fehlermeldung. Üblicherweise erfolgt die Zuweisung sofort bei der Deklarierung. Es ist außerdem zweckmäßig, immer das Attribut `static` mitzuverwenden, da es ausreicht, eine Konstante einmal im Speicher vorliegen zu haben (zu `static` später mehr). Konstantennamen werden nach Konvention vollständig groß geschrieben:

```
final int ENDE=65537;
static final String ERRMSG="Ein Fehler ist aufgetreten:";
```

A.4 Ausdrücke (Expressions)

Im Gegensatz zu C oder C++ ist die Auswertungsreihenfolge in Java nicht abhängig vom Compiler sondern für jeden Fall festgelegt. Ein Ausdruck aus mehreren Teilen wird, wenn die Teile gleiche Priorität haben, von links nach rechts ausgeführt. Alle Operatoren mit Prioritäten (P) sind in Tabelle A.4 zusammengefasst.

Die erste Spalte gibt neben der Priorität auch die Assoziativität der Operatoren an. Diese ist wichtig, wenn ein Operand zwischen mehreren Operatoren gleicher Priorität steht. ← steht für „linksassoziativ“ und bedeutet, dass der Operand stärker links gebunden ist als rechts. So wird `a+b-c` genau so ausgewertet wie `(a+b)-c`, weil `b` zwischen zwei gleich starken Operatoren steht, die aber beide ← sind. `mannschaft.name[17]` hingegen wird eben nicht abwegig als `(mannschaft.name)[17]` interpretiert (17. Mannschaftsname), sondern als `mannschaft.(name[17])`, wie man es vernünftigerweise erwartet. Es wäre sehr unpraktisch, wenn man hier Klammern setzen müsste. Da die beiden Operatoren '`.`' und '`[`' die Assoziativität → haben, muss man es nicht. Bei Operatoren verschiedener Priorität treten diese Probleme natürlich nicht auf. Der stärkere siegt.

A.5 Anweisungen (Statements)

In diesem Abschnitt werden die Kontrollstrukturen `if`, `switch`, `for`-Schleife, `while`-Schleife und `do`-Schleife besprochen. Da alle diese Anweisungen im Ausführungsteil nur einen Befehl zulassen, braucht

Tabelle A.4: Java-Operatoren mit jeweiliger Priorität

P	Bedeutung	Beispiel	Operanden
→15	Klammern ()	`3*(2+4)`	beliebig
	Arrayzugriff []	`name[3*i]`	`int`
	Punktoperator	`name.length`	beliebig
	Postfix-Operatoren	`i++ i--`	arithmetisch
→14	Prefix-Operatoren	`++i --i`	arithmetisch
	Vorzeichen	`-5`	arithmetisch
	bitweise Negation ~	`~i`	`int`
	logische Negation !	`leer=!isFull(pool)`	`boolean`
→13	Typkonvertierung	`(int)3.14`	beliebig
	Instantiierung	`new Double(1.2)`	`Object`
←12	Multiplikation	`2*3.7`	arithmetisch
	Division	`2/3.7`	arithmetisch
	Divisonsrest	`zwei=12%5`	arithmetisch
←11	Additon	`2+3.7`	arithmetisch
	Subtraktion	`2-3.7`	arithmetisch
	Verkettung	`name="Ha"+"ns"`	`String`
←10	Shift links	`zwanzig=5<<2`	`int`
	Shift rechts (Vorz. rein)	`minuszwei=-5>>2`	`int`
	Shift rechts (0 rein)	`eins=-1>>>31`	`int`
←9	kleiner, kleinergleich?	`nein=3<3; ja=3<=3`	arithmetisch
	größer, größergleich?	`ja=4>3; nein=3>=4`	arithmetisch
	Typvergleich	`ja=bmw instanceof Auto`	`Object`
←8	identische Werte?	`ja=2.3==2.3`	primitiv
	verschiedene Werte?	`nein=2.3!=2.3`	primitiv
	dasselbe Objekt?	`ja=bmw==bmw`	`Object`
	verschiedenes Objekt?	`nein=bmw!=bmw`	`Object`
←7	bitweises AND	`fünf=0xd5 & 0x0f`	`int`
	logisches AND	`ja=1<2 & 3==3`	`boolean`
←6	bitweises XOR	`fünf=0x08 ^ 0x0d`	`int`
	logisches XOR	`ja=1<2 ^ 1==2`	`boolean`
←5	bitweises OR	`sieben=6|3`	`int`
	logisches OR	`ja=1<2 | 1==2`	`boolean`
←4	abkürzendes AND	`nein=2<1 && nixmehr()`	`boolean`
←3	abkürzendes OR	`ja=1<2 || nixmehr()`	`boolean`
→2	bedingte Zuweisung;	`zwei=5<6?2:3; drei=5==6?2:3`	`boolean`, beliebig
→1	(Operation und	`*= /= += -=`	Variable, arithmetisch
	anschließende	`%= <<= >>= >>>=`	Variable, `int`
	effiziente)	`&= ^= |=`	Variable, `int`/`boolean`
	Zuweisung	`x=2+a`	beliebig

man eine Möglichkeit, mehrere Befehle zu einem einzigen zusammenzufassen. Dies wird erreicht durch Blockbildung mit geschweiften Klammern „{“ und „}“. Man kann also überall, wo nur ein Befehl erlaubt ist, mehrere durch geschweifte Klammern gruppierte Befehle einsetzen. Eine Besonderheit von Blöcken muss noch genannt werden: Variable, die in einem Block deklariert werden, sind außerhalb nicht mehr definiert. Ein Zugriff darauf ist nicht möglich. Braucht man z. B. eine Schleifenvariable auch ausserhalb, so muss man sie vor der Schleife deklarieren. Nun zu den einzelnen Anweisungen:

Die Anweisungen `if` und `if...else`

Die `if`-Anweisung gibt es mit optionalem `else`-Zweig. Die Syntax an Hand eines Beispiels sieht so aus:

```
if(i%2==0) System.out.println("gerade Zahl");
```

reagiert, wenn `i` geradzahlig ist. Mit `else`-Zweig wird auch ein ungeradzahliges `i` angezeigt

```
if(i%2==0) System.out.println("gerade Zahl");
else System.out.println("ungerade Zahl");
```

Nach `if` stehen also immer runde Klammern mit einem `boolean`-Ausdruck. Ist dieser `true`, so wird der folgende Befehl(s-Block) ausgeführt. Ein eventueller `else`-Block wird nur ausgeführt, wenn der `if`-Ausdruck `false` ist. Bei `if-else`-Ketten ist zu beachten, dass sich ein `else` immer auf das ihm nächste `if` bezieht. Wenn man das nicht so will, muss man geeignet klammern.

Die beiden folgenden für den Compiler völlig identischen Beispiele verdeutlichen dies. Der Programmierer wollte Kaffee trinken, wenn es nicht heiß ist. Wenn es heiß ist, und im Kühlschrank gibt es Eistee, dann möchte er diesen trinken. Wenn es keinen gibt, dann halt nicht. Kaffee wäre aber sicher ganz falsch.

```
if(heiß)
  if(kühlschrank.hat(eistee))
    trink(eistee);
else trink(kaffee);
```

Der Compiler versteht das aber, weil ihm Einrückungen nichts bedeuten, regelrecht so

```
if(heiß)
  if(kühlschrank.hat(eistee))
    trink(eistee);
  else trink(kaffee);
```

Es wird also nur getrunken, wenn es heiß ist, nötigenfalls sogar Kaffee. Um das gewünschte Verhalten zu erzielen, muss man schreiben

```
if(heiß){
  if(kühlschrank.hat(eistee))
    trink(eistee);
  }
else trink(kaffee);
```

Eine besonders häufig gebrauchte Verwendung für `if` ist die bedingte Zuweisung

```
if(gewonnen) punkte=100; else punkte=0;
```

Für diesen Fall gibt es in Java erfreulicherweise den sehr effizienten `?`-Operator, mit dem das selbe Beispiel so lautet

```
punkte=gewonnen?100:0;
```

Die `switch`-Anweisung

Diese Anweisung wird bevorzugt, wenn unter vielen Alternativen auszuwählen ist. Der Nachteil ist, dass die Alternativen konstant sein müssen und nur die Typen `boolean`, `byte`, `char`, `short`, `int` oder `long` verwendet werden dürfen. Als weitere Besonderheit ist zu beachten, dass, wenn ein `case` zutreffend war, alle nachfolgenden auch als zutreffend betrachtet werden. Ist der Programmfluss also einmal durch ein `case` in den konditionellen Teil eingedrungen, so bleibt er auch drin (der Amerikaner nennt das „fall-through“). Will man das verhindern, so muss man den eingeschlagenen Zweig und damit den ganzen `switch` durch ein `break` beenden. Beachten Sie im folgenden Beispiel, dass `c` ein `char` ist, also einem Buchstaben ähnlich im Programmtext verwendet wird, in Wirklichkeit aber eine Zahl ist, wie es die `switch`-Anweisung verlangt.

```
char c=getcharfromkeyboard();//frei erfundene Methode
switch(c){
  case '0':
  case '1':
  case '2':
  case '3':
  case '4':
  case '5':
  case '6':
  case '7':
  case '8':
  case '9': System.out.println("Das war eine Ziffer."); break;
  case '+': case '-': System.out.println("additives Zeichen");
  break;
  case '*':
  case '/': System.out.println("multiplikatives Zeichen"); break;
```

```
    default : System.out.println("dieses Zeichen habe ich nicht erwartet.");
  }
```

Das Beispiel zeigt auch den Sonderfall `default`. Dieser Zweig ist optional und wird dann eingeschlagen, wenn kein anderer `case` zutreffend war. Beachten Sie außerdem, dass sich der Compiler natürlich nicht um die optische Formatierung des Programmtextes kümmert; das Beispiel ist unübersichtlich aber logisch richtig.

Die `for`-Schleife

Diese Anweisung wird für Schleifen am häufigsten verwendet und ist auch am besten darauf zugeschnitten. Das folgende Beispiel gibt alle Elemente eines Arrays aus:

```
for(int j=0, n=array.length; i<n; i++){
  System.out.println("Die "+j+". Zeile ist ein");
  System.out.println(array[j]);
}
```

In der Definition der Schleife stehen zwei trennende Semikolons, die immer vorhanden sein müssen, auch wenn die Sektionen dazwischen gelegentlich leer bleiben.

Im ersten Bereich macht man initialisierende Schritte, die genau einmal am Beginn der Schleife ausgeführt werden. Meist handelt es sich um Wertzuweisungen an Variablen. Sind mehrere nötig, darf man sie durch Komma getrennt alle hier angeben, obwohl es natürlich ebenso gut möglich ist, diese Befehle vor die Schleife zu schreiben. Man kann aber seine Gedanken klarer ausdrücken, wenn man zeigt, dass es sich hier um eine Initialisierung handelt.

Im zweiten Bereich steht ein `boolean`-Ausdruck. Die Schleife wird so lange abgearbeitet, wie dieser `true` ist. Diese Bedingung ist eine Eingangsbedingung. Komplizierte Bedingungen formuliert man mit den üblichen logischen Verknüpfungsoperatoren. Fehlt diese Bedingung, so gilt die Eingangsbedingung als stets erfüllt („defaults to `true`“).

Im letzten Bereich steht üblicherweise ein Inkrement- oder Dekrement-Befehl, der am Ende der Schleife abgearbeitet wird. Auch das dient wieder nur der Schönheit des Programms, man könnte solche Befehle ja auch einfach ans Ende der Schleife setzen. Bei Verwendung von `continue` (siehe weiter unten) ist die vorhandene Notation aber tatsächlich von Vorteil.

Ein weiteres Detail ist noch die Deklaration der Variablen im Initialisierungsteil der Schleife. Solche Variablen sind nur innerhalb der Schleife zugreifbar. Braucht man ihren Wert nachher noch, so muss die Variable schon vor der Schleife deklariert werden.

Warum die `for`-Schleife so beliebt ist, zeigen folgende Beispiele, die so in Pascal nicht möglich wären:

```
for(int j=0; j<101; j=j+5) System.out.println(j+" ist Vielfaches von 5.");
for(float j=3.14; j<4; j=j+0.1f) System.out.println("noch ein Stück "+j);
for(int j=0; ; j=(j+1)%7) System.out.println("0<="+j+"<7");//endlos
```

Die `while`-Schleife

Dieses Konstrukt ist zwar wegen der Mächtigkeit der `for`-Schleife überflüssig, wurde aber aus Gründen der Tradition anderer Sprachen in Java übernommen.

```
int i=0;
while(i<10){
  System.out.println("Diese Zeile kommt 10 mal.");
  i++;
}
```

ist gleichbedeutend mit der folgenden `for`-Schleife:

```
for(int i=0; i<10; i++) System.out.println("Diese Zeile kommt 10 mal.");
```

Die `do...while`-Schleife

Dieses Konstrukt ist die einzige Schleifenart, bei der die Bedingung erst am Ende abgefragt wird, also mindestens ein Schleifendurchlauf erfolgt.

```
int i=irgendwas();
do{
  i--;
  System.out.println("Diese Zeile kommt mindestens einmal");
} while(i>0);
```

`break` und `continue` ohne Label

Beide Anweisungen unterbrechen auf ihre Weise den gewöhnlichen Programmlauf. Ein `break` ohne Label darf nur innerhalb von Schleifen oder einer `switch`-Anweisung verwendet werden. Das Programm wird dann am Ende dieser einschließenden Anweisung fortgesetzt. Mit der `continue`-Anweisung verhält es sich ähnlich. Ohne Label darf sie nur innerhalb einer Schleife verwendet werden. Diese wird aber nicht abgebrochen, vielmehr wird sofort der nächste Durchgang eingeleitet. Der Rest der Schleife wird also im aktuellen Durchgang ignoriert.

`break` und `continue` mit Label

Ein Label wird dann verwendet, wenn man die Funktionsweise von `break` und `continue` ausdehnen will auf weiter außen liegende Schleifen. `break aussen` springt dann ans Ende der mit `aussen` belabelten Schleife. `continue aussen` versetzt sie unverzüglich in den nächsten Schritt.

```
aussen: for(int i=0; i<10; i++){
  ...
  for(int j=0; j<i; j++){
    if(...) break aussen;
    ...
  }
}/*aussen ist hier zu Ende - das Programm geht
hier weiter, wenn die Schleife zu Ende ist
oder wenn der break auftrat*/
```

Eine Besonderheit bei `break` ist, dass im belabelten Fall die umgebende Struktur nicht mehr unbedingt eine Schleife sein muss. Es kann ein beliebiger durch `{` und `}` eingeschlossener Block sein, der an seinem Anfang ein Label hat.

Zusammenfassend kann man sagen, dass die beiden Befehle für eine strukturierte Sprache wie Java ein kleiner Ersatz für Sprungbefehle sind. In Java ist zwar auch das Schlüsselwort `goto` reserviert, wird aber nicht verwendet.

A.6 Ausnahmebehandlung (Exceptions)

In einem Programmlauf können unvorhergesehene Situationen auftreten. Das Programm bricht ab, wenn der Speicher nicht reicht, weil z. B. das Betriebssystem keinen mehr zur Verfügung stellt. Das Programm bricht nicht gleich ab, wenn ein Lesevorgang von Diskette nicht zu Ende geführt werden kann, weil ein Sektor der Diskette defekt ist. Oder eine Benutzereingabe, die eigentlich eine Zahl sein sollte, enthält einen Buchstaben und ist dadurch nicht weiter brauchbar; die Eingabe sollte wiederholt werden.

Wollte man so programmieren, dass auf jede mögliche Inkonsistenz richtig reagiert wird, so wäre das Programm nur noch eine Ansammlung von Sicherheitsabfragen. Ein klarer Algorithmus würde darin hoffnungslos verschwinden. In Java gibt es deshalb, wie in vielen modernen Programmiersprachen, eine Programmstruktur, die einen problematischen Programmteil in eine Art geschütztes Umfeld stellt. Wenn alles gut geht, kann man die Schutzumgebung ignorieren, wenn etwas schief geht, übernimmt sie die Kontrolle und biegt die Sache wieder so gut wie möglich hin (wenn sie gut programmiert ist).

Gravierende Probleme, die unrettbar zum Programmabbruch führen, können übrigens auch damit nicht gelöst werden. In Java spricht man dann von einem Error. Was in Java „behandelt“ wird, nennt man Exception.

Wir betrachten als Beispiel einen Stack von `int`-Zahlen. Die Größe unseres primitiven Stacks kann nur am Anfang festgelegt werden, wodurch man natürlich das Risiko eingeht, dass er überläuft, wenn man zu viel darin ablegt (push). Außerdem besteht immer die Gefahr, dass man mehr herausholen will (pull),

als drin ist. Dieses Problem kann man noch vermeiden, indem man gewissenhaft programmiert und gut testet. Es gibt aber auch Situationen, die man nicht beherrschen kann, oder solche, wo man absolut sicher gehen will oder muss.

Unser Stack hat nun jedenfalls Methoden `push(int)` und `pull()`, die eine `Exception` erzeugen (das englische Verb ist `throw`).

```
import java.util.EmptyStackException;//gibt es schon
import java.lang.RuntimeException;

class FullStackException extends RuntimeException{
  //Diese Klasse tut genau dasselbe wie RuntimeException.
  //Wir wollen aber diesen Namen, weil er aussagekräftiger ist.
  public FullStackException(String sagwas){
    super(sagwas);
  }
}

public class IntStackWithExceptions{
  int[] stack;
  int sp=-1;//StackPointer

  public IntStackWithExceptions(int größe){stack=new int[größe];}

  public void push(int was) throws FullStackException{
    if(sp+1>=stack.length) throw new FullStackException(
      "Aber hallo, der Stack ist schon voll mit "+(sp+1)+" Elementen!");
    stack[++sp]=was;
  }

  public int pull() throws EmptyStackException{
    if(sp<0) throw new EmptyStackException();
    return stack[sp--];
  }

  public boolean stackEmpty(){return sp<0;}
}
```

Da es in Java schon einen Stack gibt (`java.util.Stack`) und die dort definierte `EmptyStackException` für unsere Zwecke gut passt, können wir diese gleich übernehmen. Für den Überlauf müssen wir selber eine programmieren, `FullStackException`[3]. Die Erzeugung von Exceptions ist meist recht einfach, da sie eigentlich alle dasselbe tun: Sie haben die besagte Wirkung und können eine Nachricht als `String` tragen. Diese Eigenschaften hat schon die Urmutter `Exception`, von der alle abstammen. Man kann sich also auf diese berufen (`super(...)`). Man sollte sich jedoch bemühen, dass die neu erfundene Exception an einem semantisch vernünftigen Platz im Stammbaum steht. Hier nun ein Programm, das unseren Stack verwendet:

```
public class TestExceptions{
  /* In dieser Test-Klasse kann man Zahlen eingeben:
     positive Zahlen werden gepusht
     bei Eingabe von 0 wird gepullt und angezeigt
     bei negativen Zahlen hört das Programm auf
  */
  public static void main(String[] args){
    IntStackWithExceptions stack=new IntStackWithExceptions(5);
    int z;
    while((z=LineInput.readInt())>=0){
      if(z>0)
        try{//hier beginnt der try-catch-Block
          stack.push(z);
        }catch (FullStackException e){
          System.out.println("Beim push ist etwas schief gegangen:");
          System.out.println(e.getMessage());
        }//hier endet er
      else
        try{//hier beginnt ein try-catch-finally-Block
          System.out.println(stack.pull());
```

[3] Unser Stack kann überlaufen, der im Package `util` mitgelieferte nicht.

```
            }catch (java.util.EmptyStackException e){
              System.out.println("Beim pull ist etwas schief gegangen:");
              System.out.println(e.getMessage());
            }finally{
              System.out.println("So oder so ist hiermit das pull zu Ende.");
            }//hier endet er
        }
      }
    }
```

Die problematischen Stellen sind eingeschlossen in einen `try`-Block. Wie das Wort schon vermuten lässt, handelt es sich hierbei vorerst nur um einen Versuch der Durchführung. Tritt dabei eine Exception auf, so wird kein weiterer Befehl des `try`-Blocks mehr ausgeführt, sondern sofort in den zuständigen `catch`-Block gesprungen, der die dort angegebene Exception abfängt. Es dürfen übrigens mehrere solche hintereinander aufgeführt werden, falls mehrere verschiedene Exceptions auftreten können. Die Reihenfolge ist aber wichtig, weil die `catch`-Blöcke der Reihe nach durchsucht werden und nur der erste für die Exception zutreffende ausgeführt wird. Tritt keine Exception auf, so wird auch kein `catch`-Block abgearbeitet. Nach den `catch`-Blöcken kann noch ein `finally`-Block folgen, der auf jeden Fall abgearbeitet wird, egal ob nun eine Exception aufgetreten ist oder nicht. Auf einen `try`-Block muss mindestens ein `catch`-Block oder der `finally`-Block folgen. Nach der Abarbeitung eines `catch`-Blocks (und des evtl. nachfolgenden `finally`-Blocks) ist die `try`-Struktur als Ganzes beendet. Das Programm läuft an der Stelle danach weiter.

Auch was Exceptions angeht, ist Java sehr konsequent. Der Compiler passt auf, dass alle „geworfenen“ Exceptions auch behandelt werden. Deshalb muss eine Methode, die eine Exception produziert, dies im Methodenkopf deklarieren mit dem `throws`-Hinweis. Ruft nun ein anderer Programmteil eine solche Methode auf, so muss er die dort potentiell geworfenen Exceptions behandeln oder sie selber deklarieren. Zu dieser sehr einleuchtenden Philosophie wurde aber eine gewaltige Ausnahme geschaffen: Die Klasse `RuntimeException` und alle ihre Abkömmlinge müssen nicht behandelt werden, weil es einfach nicht zumutbar wäre, z. B. jede Indexüberschreitung bei Feldzugriffen abzusichern. Es besteht auch tatsächlich eine große Wahrscheinlichkeit, dass man solche Probleme durch aufmerksames Programmieren vermeidet. Die beiden Exceptions des obigen Beispiels sind Abkömmlinge von `RuntimeException` und hätten nicht behandelt werden müssen.

A.7 Packages

Java-Klassen lassen sich in Gruppen (Packages) zusammenfassen. Dies ist zum einen praktisch, wenn man in logisch zusammengehörigen Programmstücken Zusammengehörigkeit anzeigen will, zum anderen unumgänglich, wenn es mehrere Klassen gleichen Namens gibt. Aus letzterem Grund muss sich die Gruppierung in Packages auch in einer geeigneten Verzeichnisstruktur wiederspiegeln.

Die augenfällige Eigenschaft aller Klassen in einem Package ist, dass von außerhalb des Packages nur die `public` deklarierten Klassen und Interfaces sichtbar (d. h. zugreifbar) sind. Innerhalb der „Familie“ kann jede Klasse auf jede andere zugreifen.

Als Beispiel wollen wir einmal annehmen, dass wir für ein Graphik-Projekt die Klassen `Rectangle`, `Circle`, `Color` usw. brauchen. Allein weil die Klasse `Rectangle` schon existiert, muss es einen Weg geben, dem Compiler zu sagen, ob er die unsere oder die schon vorhandene nehmen soll. Wir schreiben also in die Datei `Rectangle.java` als *erste Anweisung* `package graphics` und speichern die Datei im Unterverzeichnis `graphics` ab. Damit ist das `graphics`-Package erzeugt und man kann zwischen `java.awt.Rectangle` und `graphics.Rectangle` unterscheiden.

Die Lage könnte theoretisch noch komplizierter werden, wenn unser Rechner über das Internet mit vielen anderen Rechnern verbunden ist und von dort auch auf Klassen zugreifen wollte, die vielleicht schon wieder den gleichen Namen haben, also auch `graphics.Rectangle` aber auf einem anderen Rechner. Wodurch ist dann definiert, von welchem Rechner die genannte Klasse zu holen ist? Dazu wurde vereinbart, dass alle großen Organisationen ihren eigenen Namen in den Klassennamen einfließen lassen, wodurch dieser dann eindeutig wird. Und zwar soll wie bei Internet-URLs vorgegangen werden, aber vom Allgemeinen zum Besonderen (also genau umgekehrt wie in den URLs). Wenn es also, einmal angenommen, mehrere Entwicklungsteams von `graphics`-Paketen in der Firma IBM gibt, so sollte z. B. das Team um Herrn Smith in ihrer Datei `Rectangle.java` notieren: `package COM.ibm.smithteam.graphics`.

Lässt man die `package`-Anweisung weg, so gehört die Klasse übrigens auch zu einem Package, nämlich jenem ohne Namen. Wenn man vom namenlosen Package spricht, nennt man es das Default-Package.

So viel zu den Namen und Ablageorten von Klassen in Packages. Die Folge ist, dass man nun einen gehörigen Schreibaufwand hat, z. B. `java.awt.Rectangle rect=new java.awt.Rectangle()`. Wenn man immer die selbe Sorte Rectangle braucht, kann man dies aber dem Compiler von Anfang an mitteilen durch die `import`-Anweisung. Man kann schreiben `import java.awt.Rectangle`, wenn man aus dem Paket `java.awt` nur die `Rectangle`-Klasse braucht, oder `import java.awt.*`, wenn man so viele Klassen aus diesem Paket braucht, dass die Einzelauflistung nicht mehr praktikabel ist. Im weiteren Programmtext reicht dann die Zeile `Rectangle rect=new Rectangle()`.

Die `import`-Anweisungen folgen im Quelltext nach einer etwaigen `package`-Anweisung. Bei Verwendung eines `*` werden alle Klassen des genannten Packages importiert, aber nicht die Klassen von etwaigen Unterpackages. Mit `import java.awt.*` werden also nicht die Klassen von `java.awt.event.*` importiert.

Automatisch importiert werden übrigens auch ohne eine entsprechende Anweisung die Klassen des Packages `java.lang`, des Default-Packages und des aktuellen Packages[4]. Sollte bei mehreren `import`-Anweisungen eine Klasse durch ihren Kurznamen nicht mehr eindeutig identifizierbar sein, so bekommt man eine Compiler-Fehlermeldung und muss dann eben doch den vollständigen Namen verwenden.

Da sich alle Klassen eines Packages in einem entsprechenden Unterverzeichnis befinden, muss man Compiler und Interpreter auch mitteilen, wo sie das Stammverzeichnis dieser Unterverzeichnisse finden können. Dies geschieht mit der Option `-classpath "..."`. In den Anführungszeichen kann man dann mehrere Einstiegspunkte als Suchpfad angeben. Das aktuelle Verzeichnis und das Installationsverzeichnis der Java-Bibliothek braucht man aber nie anzugeben. Diese beiden befinden sich immer automatisch im Classpath.

Da Java-Klassen oft in `jar`-Dateien komprimiert vorliegen (diese haben intern das `zip`-Format), dürfen im Classpath auch die Namen von `zip`- oder `jar`-Dateien angegeben werden. Diese müssen aber die Verzeichnisstruktur der enthaltenen Klassen beinhalten, falls Packages gezipt wurden.

Im Beispiel:

```
javac -classpath "c:\java\meineklassen;c:\java\projekte\funktion.jar" Test.java
```

wurde der Compiler aus dem Verzeichnis heraus gestartet, in dem sich `Test.java` befindet. Er sucht nun zuerst einmal die Datei `Test.java` in den Verzeichnissen, die im Classpath angegeben sind. (Die Reihenfolge wird beachtet.) Dort findet er sie nicht. Er sucht jedoch grundsätzlich noch weiter im aktuellen Verzeichnis, wo er die Datei auch findet. (Wäre sie hier auch nicht, so ginge er noch die Java-Bibliothek durch, würde sie dort nicht finden und endete mit einer Fehlermeldung.) Nun kann `Test.java` compiliert werden. Alle Referenzen in `Test.java` auf weitere Klassen werden in der gleichen Weise gesucht und hoffentlich auch gefunden. Sollte es aus irgendwelchen Gründen wichtig sein, dass das aktuelle Verzeichnis als erstes durchsucht wird, so kann man das Kürzel „`.`“ als erstes in den Classpath aufnehmen.

[4] *Default* ist das Package ohne `package`-Anweisung im Wurzelverzeichnis des Projekts. *Aktuell* ist das Package, in dem sich die gerade programmierte Datei befindet.

B Klassen, Interfaces, Vererbung

Im vorigen Abschnitt wurden die wichtigsten Grundkonstrukte von Java vorgestellt. Bei einigen Erklärungen und Beispielen war es aber bereits nötig, Klassen zu programmieren und zu verwenden. Auf diesbezügliche Einzelheiten konnte dabei nicht eingegangen werden, weil ja zuerst die Sprache in ihren Grundzügen bekannt sein muss, bevor man über schwierigere Konzepte diskutieren kann. Dies soll nun oberflächlich nachgeholt werden.

Für eine tiefer gehende Darlegung fehlt hier der Platz. Über diesen Themenbereich liest man üblicherweise mehrere Bücher und macht sich im Verlauf größerer Projekte seine eigenen Gedanken.

B.1 Klassen

Den Begriff *Klasse* umschreibt man am besten als Bauplan. Der Name sollte eine Aussage über die Funktionalität machen. Im Bauplan wird dann beschrieben, mit welchen Befehlen diese Funktionalität verwirklicht wird und welche Variablen dazu benötigt werden. Ein Java-Programm besteht in den allermeisten Fällen aus mehreren oder vielen Klassen. Das „Java-Programm“ selbst ist dann oft nur noch eine kleine Klasse, deren Programmtext die vielen anderen Klassen verwendet und in ein sinnvolles Ganzes zusammenbindet.

Es gibt Klassen (Baupläne), nach denen man Objekte erstellen kann, die eigenständig existieren aber alle die gleiche Funktionalität haben. Man nennt die Objekte in diesem Zusammenhang auch Instanzen. Man könnte z. B. die Klasse `Auto` programmieren, die festlegt, dass ein `Auto` eine Variable `tank` hat, in der abgespeichert ist, wieviel Benzin noch vorhanden ist und eine Methode `tanken(int liter)`, die es erlaubt, den Wert der Variable `tank` zu vergrößern. Außerdem gäbe es noch viele andere Methoden und Variablen, die die Funktionalität eines `Auto`s beschreiben. Mit Hilfe dieses Bauplans kann man dann verschiedene Autos (Instanzen) erzeugen (`Auto meinBmw=new Auto(); Auto deinAudi=new Auto();`), die alle ihren eigenen Tank haben (`meinBmw.tank` bzw. `deinAudi.tank`).

Es gibt aber auch Klassen, bei denen es keinen Sinn hat, einzelne Instanzen zu erzeugen. So gibt es z. B. in Java die Klasse `Math`, mit oft gebrauchten mathematischen Methoden wie `sqrt`. Um sie zu verwenden, braucht man keine verschiedenen Instanzen von `Math`; es gibt ja nur die eine Mathematik. Um also die Wurzel von 5 auszurechnen, schreibt man dann einfach `x=Math.sqrt(5)`. Hier wird im Bauplan von `Math` nachgesehen, wie eine Wurzel berechnet wird.

Bei der Deklaration von Klassen hat man einige Optionen. Alles was in eckigen Klammern steht, kann auch weggelassen werden. Die einzelnen Optionen werden im folgenden erklärt.

[`public`][`abstract`][`final`] `class Name` [`extends VorfahrName`][`implements InterfaceListe`]

- Ist eine Klasse `public`, so ist sie von überall her verfügbar, ansonsten nur aus Klassen des selben Package.
- `abstract` macht man eine Klasse, wenn man verhindern will, dass sie instantiiert wird. Wenn man z. B. die Klasse `Säugetier` definiert mit der Methode `säugen`, dann erben die Nachfahren `Paarhufer` und `Affe` diese Funktionalität. Trotzdem sollte es verhindert werden, dass ein `Säugetier` erzeugt wird, denn so etwas gibt es nicht.
- Eine `final`-Klasse kann keine Nachfahren haben. Wenn man sicher ist, dass keine Nachfahren mehr benötigt werden, dann sollte man die Klasse `final` machen, weil der Compiler dann einige Optimierungen vornehmen kann.
- Mit `extends` gibt man den Namen des Vorfahren an, von dem sich die Klasse ableitet. Die Klasse übernimmt (erbt) von ihm seine gesamte Funktionalität[1]. Erweitern oder ändern kann man diese Funktionalität dadurch, dass man einzelne Methoden neu programmiert. Gibt man keinen Vorfahren an, so wird implizit die Klasse `Object` als Vorfahre verwendet. Diese ist Vorfahre von allen anderen Klassen.

[1] Es besteht kein Zugriff auf Attribute und Methoden, die im Vorfahren `private` deklariert sind.

- nach `implements` kann eine durch Kommata getrennte Liste von Interface-Namen folgen, die die Klasse implementiert. Näheres dazu im entsprechenden Abschnitt.

B.2 Zugriffsstufen

Bei der Verwendung von Feldern (Variablen) und Methoden (Funktionen, Prozeduren) unterscheidet man vier Zugriffsstufen: `private`, `protected`, package und `public`. Die Stufe „package“ erreicht man dadurch, dass man keines der drei anderen Schlüsselwörter angibt.

Zugriffsstufen dienen dem Schutz von Variablen und Methoden gegen unbefugten Zugriff bzw. Aufruf. In den folgenden Erklärungen werden nur Variablenzugriffe verwendet. Für Methodenaufrufe gelten analoge Regeln.

- Die strengste Zugriffsstufe ist `private`. Auf eine so deklarierte Variable kann nur von der deklarierenden Klasse selbst (oder von ihren Instanzen) zugegriffen werden. Man verwendet diese Stufe für Arbeitsvariablen, die für außenstehende Klassen bedeutungslos sind oder bei denen die Gefahr besteht, dass ihre Veränderung von außen die Funktionalität der Klasse stören könnte.
- Etwas großzügiger ist die Package-Stufe. Auf derart deklarierte Variablen haben alle Klassen desselben Package Zugriff. Selbst die Nachfahren müssen zum selben Package gehören, wenn sie Zugriff haben sollen.
- `protected` umfasst die beiden bisher genannten Stufen und nimmt noch die Nachfahren hinzu, auch wenn sie nicht zum gleichen Package gehören.
- Die einfachste Stufe ist natürlich `public`. Auf solche Variablen darf von überall her zugegriffen werden.

Die folgende Tabelle fasst noch einmal zusammen, welche Stufe welcher Klasse Zugriff ermöglicht:

	`public`	`protected`	package	`private`
nur die Klasse selber	×	×	×	×
aus dem gleichen Package	×	×	×	
Nachfahren	×	×		
alle anderen	×			

B.3 Konstruktoren

Bevor eine Instanz einer Klasse verwendet werden kann, muss sie erzeugt und initialisiert werden, d. h. es muss wenigstens genug Speicher für alle Felder (Variablen) reserviert werden. Alle Werte werden anfangs auf 0 bzw. `null` gesetzt. Dies zu erledigen ist die Aufgabe des Konstruktors. Außerdem kann er noch weitere Initialisierungsschritte unternehmen, die aber dann ausdrücklich programmiert werden müssen. Der Konstruktor wird immer dann aufgerufen, wenn ein Objekt mit `new` erzeugt wird, z. B. `Stack meinStack=new Stack(50)`.

Man programmiert einen Konstruktor wie eine gewöhnliche Methode, allerdings ohne Angabe eines Rückgabetyps und mit dem gleichen Namen wie die Klasse.

```
Zugriffsstufe KlassenName(ParameterListe)
```

- Es können mehrere Konstruktoren gleichen Namens angegeben werden. Welcher zu verwenden ist, erkennt der Compiler an der Parameterliste. Man nennt dies *Überladen* von Konstruktoren.
- Programmiert man eine Klasse ganz ohne Konstruktor, weil man außer der Speicherreservierung nichts will, so ergänzt der Compiler automatisch den Default-Konstruktor mit keinem Parameter, der genau das tut. Hat man mindestens einen Konstruktor angegeben, so ergänzt der Compiler den Default-Konstruktor nicht! Wenn man ihn dann trotzdem braucht, muss man ihn selber programmieren.
- Bei der Initialisierung von Objekten ist es notwendig, dass die von den Vorfahren geerbten Felder auch initialisiert werden. Deshalb muss als erster Befehl stets ein Konstruktor des Vorfahren aufgerufen werden, z. B. mit `super(50)`. Tut man das nicht, so ergänzt der Compiler diesen Aufruf

stillschweigend. Da er aber nicht wissen kann, welchen Konstruktor des Vorfahren er verwenden soll, nimmt er den Default-Konstruktor. Der muss dann natürlich vorhanden sein, sonst gibt es eine Fehlermeldung. Am besten und klarsten ist es deshalb, immer selber diesen Aufruf zu programmieren. Dabei kann man dann auch wählen, welchen Konstruktor des Vorfahren man aufrufen will.

- Es gibt eine kleine Einschränkung zum vorigen Punkt. Es besteht die Möglichkeit, aus einem Konstruktor einen anderen aufzurufen. Z. B. kann man aus dem Default-Konstruktor (ohne Parameter) einen anderen (mit einem Parameter) aufrufen mit `this(10);`. Dann wird nicht vorher `super()` aufgerufen. Im anderen Konstruktor passiert es aber dann doch.

B.4 Felder

In Java heißen die Variablen *Felder*. Im Deutschen führt das zu Missverständnissen, weil man dabei gern an Arrays denkt, was aber nicht gemeint ist. In diesem Fall muss dann auch der Deutsche „Array“ sagen. Felder haben entweder einen primitiven Typ, bei dem der Wert direkt abgespeichert wird oder einen Referenztyp, bei dem ein Zeiger auf ein bestimmtes Objekt abgespeichert ist. Man deklariert Felder wie folgt:

`Zugriffsstufe` [`static`][`final`][`transient`][`volatile`] `typ variablenName` [`=...`]

- Wenn man mehrere Instanzen einer Klasse hat, die sich in einem Wert unterscheiden, so brauchen diese Instanzen alle ein eigenes Feld für diesen Wert. Man macht dann dieses Feld *nicht* `static`. Werte, die in allen Instanzen gleich sind (Konstanten) oder über die Gesamtheit der Instanzen etwas aussagen (z. B. ihre Anzahl) wird man aber `static` machen. `static`-Felder beziehen sich also auf die Klasse, nicht-`static`-Felder auf einzelne Instanzen.
- Der Zusatz `final` bewirkt, dass eine Zuordnung nicht mehr geändert werden kann, dass also das Feld eine Konstante ist. Bei Konstanten ist es meist sinnvoll, sie auch `static` zu machen. Der Name eines `final`-Feldes wird üblicherweise immer vollständig in Großbuchstaben geschrieben.
- Der Zusatz `transient` wird sehr selten gebraucht. Er verhindert, dass das Feld beim Abspeichern eines Objektes („Serialisation“) mitgespeichert wird. Auch beim Wiederherstellen („Deserialisation“) wird es ignoriert.
- Noch spezieller ist die `volatile`-Eigenschaft. Sie hat nur bei Verwendung mehrerer Threads eine Bedeutung und wird hier nicht besprochen.

B.5 Methoden

Funktionen und Prozeduren heißen in Java *Methoden*. Alle Methoden haben einen Rückgabetyp. Wird nichts zurückgegeben, so ist der Rückgabetyp `void`. Hat die Methode keine Parameter, so muss ein leeres Klammerpaar angegeben werden. Wie schon bei den Konstruktoren ist Überladen möglich, d. h. es können mehrere Methoden des gleichen Namens deklariert werden. Der Compiler erkennt an den Parametern, welche er verwenden muss. Methoden werden wie folgt deklariert:

`Zugriffsstufe` [`static`][`abstract`][`final`][`native`][`synchronized`] `typ name(ParameterListe)`
[`throws ExceptionListe`]

- Wie schon bei den Variablen ist eine `static`-Methode eine, die sich nicht auf einzelne Instanzen oder ihre Variablen bezieht, sondern auf die Klasse selbst.
- Eine Methode ist `abstract`, wenn nur ihr Kopf aufgeführt ist, sie aber nicht implementiert ist. Man wählt dies, wenn man bei einer Klasse schon weiß, welchen Namen die Methode bei den Nachfahren haben soll, aber noch nicht weiß, wie die Nachfahren die gewünschte Funktionalität im einzelnen herstellen. Eine Klasse mit einer `abstract`-Methode ist natürlich nicht funktionsfähig und daher nicht instantiierbar. Deshalb muss eine solche Klasse auch `abstract` deklariert sein.
- Soll eine Methode von den Nachfahren nicht mehr überschrieben und dadurch abgeändert werden können, so macht man sie `final`. Der Compiler kann dann einige Optimierungen vornehmen.

- Manchmal ist es nötig, sehr eng mit dem Betriebssystem, auf dem Java aufsetzt, zusammenzuarbeiten, weil z. B. ungewöhnliche Hardware gesteuert werden muss. Dann muss vielleicht mit einer fremden Programmbibliothek in einer anderen Sprache kommuniziert werden. Da hier nicht mehr nur Java-Code ausgeführt wird, muss so eine Methode als `native` deklariert werden. Programme mit solchen Methoden laufen dann auch nicht mehr auf allen Betriebssystemen.
- In Java ist es möglich, Programme parallel oder quasi-parallel ablaufen zu lassen. Man spricht von „Threads“, was man als „Handlungsstränge“ übersetzen könnte. Will man verhindern, dass ein zweiter Thread in eine Methode eintritt, die der erste noch nicht verlassen hat, macht man die Methode `synchronized`. Darauf soll hier nicht weiter eingegangen werden.
- Der Typ einer Methode gibt an, was sie zurückliefert, wenn sie fertig ist.
- In einer Methode können Exceptions auftreten oder explizit erzeugt werden (siehe Abschnitt A.6). Diese müssen im Kopf der Methode mit `throws`, durch Kommata getrennt, aufgelistet werden.

B.6 Überschreiben, verdecken

Ein Nachfahre erbt alle Felder und Methoden, auf die ihm sein Vorfahre Zugriff gewährt, außer er überschreibt eine Methode oder verdeckt ein Feld. Der Zugriff wird geregelt mittels der vier früher genannten Zugriffsstufen.

Eine Methode zu überschreiben bedeutet, sie beim Nachfahren nochmals neu zu programmieren, damit sie den Besonderheiten des Nachfahren besser gerecht wird. Braucht man die Vorfahr-Methode auch, so steht sie mit `super.methodenname(...)` zur Verfügung.

Auch Felder, die eine Klasse schon geerbt hat, kann sie nochmals neu deklarieren. Dadurch wird das Feld des Vorfahren verdeckt, d. h. bei Verwendung des Feldnamens wird immer auf das neu deklarierte Feld zugegriffen. (Dieses kann übrigens auch einen anderen Typ haben als beim Vorfahren.) Braucht man doch einmal das Feld des Vorfahren gleichen Namens, so kann man darauf mit `super.feldname` zugreifen.

B.7 Interfaces

Ein Interface hat etwa den gleichen Sinn wie eine Bescheinigung. Eine Bescheinigung hat einen aussagekräftigen Namen und ist eine Liste von Fähigkeiten, die beherrscht werden müssen, um zum Besitz der Bescheinigung zu berechtigen. Wenn jemand diese Bescheinigung besitzt, kann ohne weitere Prüfung davon ausgegeangen werden, dass er die geforderten Fähigkeiten hat. Wir definieren als Beispiel das Interface `Pianist`.

```
public interface Pianist{
  void spiele(Klavier k, Werk w);
  void verbeugeDich();
}
```

Wenn nun die Klasse `Taschenrechner` diese beiden Methoden implementiert und dies auch in ihrer Kopfzeile anzeigt mittels `implements Pianist`, dann kann sie außer als `Taschenrechner` auch als `Pianist` verwendet werden. Schließlich kann man bei ihr die beiden geforderten Methoden aufrufen und dabei die geforderten Parameter übergeben. Wie gut die beiden Methoden programmiert wurden, interessiert den Compiler natürlich nicht. Selbst wenn die Methoden leer sind und niemals einen Ton produzieren, so funktioniert doch der Aufruf und das Programm bleibt lauffähig.

Die pianistischen Fähigkeiten des `Taschenrechner`s kann man natürlich einerseits direkt ansprechen, wie etwa in

```
Taschenrechner t=new Taschenrechner(); t.spiele(...);
```

Damit hat man aber nicht den Interface-Mechanismus ausgenutzt. Üblicherweise ist es an der betreffenden Stelle im Programm nämlich tatsächlich egal, welches Objekt nun die in `Pianist` definierten Fähigkeiten mitbringt. Deshalb reduziert man z. B. den Taschenrechner auf seine pianistischen Fähigkeiten mit

```
Pianist p=new Taschenrechner(); p.spiele(...);
```

Der Compiler erlaubt dies, weil er leicht nachprüfen kann, dass `Taschenrechner` das Interface `Pianist` implementiert. Wenn man danach auf die Idee kommt, `p` wieder als `Taschenrechner` einzusetzen[2], dann erlaubt der Compiler das nicht mehr ohne weiteres. Nach der Zuordnung an eine Variable des Typs `Pianist` geht er nämlich nur noch davon aus, dass das abgespeicherte Ding die Fähigkeiten von `Pianist` hat, nicht mehr und nicht weniger. Falls der Programmierer es besser weiß, muss er schreiben

```
Pianist p=new Taschenrechner(); p.spiele(...);
((Taschenrechner)p).addiere(...);
```

Damit ist der Compiler beruhigt. Zur Laufzeit wird aber trotzdem nochmal getestet, ob das Ding in `p` wirklich ein `Taschenrechner` ist. Der Programmierer könnte sich ja geirrt haben und während des Programmlaufs kann einer Variablen gewöhnlich viel passieren.

[2] Wer auf diese Idee kommt, hat wahrscheinlich einen größeren Fehler bei der Anlage seiner Klassenstruktur gemacht. Normalerweise sollte sowas nicht nötig sein!

C Bits und Bytes

C.1 Die Standardbasen 2, 8, 10 und 16

Die folgende Tabelle ist eine Gegenüberstellung der für unsere Zwecke wichtigsten Stellenwertsysteme: dezimal, binär (dual), hexadezimal und oktal. Es ist hilfreich, den Wert einer Zahl (*die Zahl*) von ihrer

Tabelle C.1: Zahlensysteme

(10)	(2)	(16)	(8)	(10)	(2)	(16)	(8)
dez	bin	hex	okt	dez	bin	hex	okt
0	0	0	0	16	10000	10	20
1	1	1	1	17	10001	11	21
2	10	2	2				
3	11	3	3	31	11111	1f	37
4	100	4	4	32	100000	20	40
5	101	5	5				
6	110	6	6	64	1000000	40	100
7	111	7	7				
8	1000	8	10	100	1100100	64	144
9	1001	9	11				
10	1010	a	12	127	1111111	7f	177
11	1011	b	13	128	10000000	80	200
12	1100	c	14				
13	1101	d	15	196	11000100	c4	304
14	1110	e	16				
15	1111	f	17	255	11111111	ff	377

Schreibweise zu unterscheiden. Hundert ist eine Zahl und wird geschrieben als 100_{10}, 1100100_2, 202_7, 64_{16}. Man gibt die Basis als Index bei, wobei vereinbart wird, dass die Basis *immer* im Dezimalsystem angegeben wird – sonst würde es lustig!

Für die für uns wichtigen Basen 10, 16 und 8 verwenden wir auch die in Java üblichen Schreibweisen:

dezimal	wie gewohnt	`0`,`1`,`19`,`64`
oktal	mit führender Null	*0*, *01*, *023*, *0100*
hexadezimal	mit `0x` vorne	`0x0`, `0x1`, `0x13`, `0x40`

Für die Basis 2 stellt Java leider keine eigene Schreibweise bereit.

C.2 Negative Zahlen

Bei 8 Bits hat man 256 Möglichkeiten, sie zu setzen oder nicht zu setzen. Das kann man, wie oben, dazu verwenden, die Zahlen 0 bis 255 zu codieren. Braucht man auch negative Zahlen, so ist das hauptsächlich eine Interpretationsfrage. Am weitesten verbreitet sind die drei folgenden Codierungen: Vorzeichen-Betrag-, 2-Komplement- und Exzess-Darstellung. Welche man verwendet, hängt von der Praktikabilität in der jeweiligen Anwendung ab.

Bei der Vorzeichen-Betrag-Methode wird das oberste Bit zur Darstellung des Vorzeichens verwendet. Dadurch hat man die Besonderheit der doppelten Null (±0). Der Bereich erstreckt sich zwischen −127 bis +127. Mit der Zweierkomplement-Darstellung kann man die Zahlen −128 bis +127 codieren. Bei 0 bis 127 ist das oberste Bit 0. Alle Zahlen mit gesetztem obersten Bit gelten als negativ. Bei der Exzess-Darstellung muss eine bestimmte Zahl abgezogen werden, um die tatsächliche Zahl zu erhalten. Tabelle C.2 stellt die drei Formen gegenüber. In einer Java-VM wird die 2-Komplement-Darstellung angewandt.

Tabelle C.2: Darstellung negativer Zahlen

Zahl	±-Betrag	2-Komplement	Exzess-127
128	—	—	11111111
127	01111111	01111111	11111110
2	00000010	00000010	10000001
1	00000001	00000001	10000000
0	00000000	00000000	01111111
−0	10000000	—	—
−1	10000001	11111111	01111110
−2	10000010	11111110	01111101
−127	11111111	10000001	00000000
−128	—	10000000	—

Eine Vorzeichen-Betrags-Zahl wird negiert, indem man das oberste Bit umdreht, eine 2-Komplement-Zahl, indem man alle Bits umdreht und dann 1 addiert, bei einer Exzess-m-Zahl muss man ein bisschen rechnen.

C.3 Binäre Operatoren

Es gibt einige grundsätzliche Operationen, die (fast) jeder Prozessor beherrscht. Da immer nur höchstens zwei Operanden verwendet werden, sind die folgenden Rechnungen besonders einfach:

Addition:	00111001	Summand	57
Java +	00011101	Summand	29
	111 1	Übertrag (Carry)	1
	01010110	Summe	86

Die Subtraktion lässt sich zurückführen auf die Addition einer negativen Zahl. Bei dieser Gelegenheit wird auch klar, warum die 2-Komplement-Codierung für negative Zahlen so praktisch ist: Damit das Ergebnis stimmt, muss man nur den Übertrag außerhalb der letzten Stelle ignorieren.

Subtraktion:	01101101	Summand	109
Java -	11110110	Summand (neg.)	−10
	11111	Übertrag (Carry)	1
	$_1$01100011	Summe	99

Diesen einfachen Mechanismus beherrschen die meisten Prozessoren mit ihrem Subtrakt-Befehl, sodass der Programmierer nicht selbst die 2-Komplemente erstellen muss. Ein größeres Problem ist die Bereichsüberschreitung bei der Addition zweier Zahlen mit gleichem Vorzeichen. Da kann es passieren, dass die Summe zweier positiver Zahlen so groß ist, dass das Ergebnis das oberste Bit setzt, was dann fälschlicherweise aber konsequenterweise als negative Zahl interpretiert wird. Analoges passiert bei der Addition zweier negativer Zahlen, deren Summe das höchste Bit löscht, sodass das Ergebnis positiv aussieht.

Die Multiplikation führt man durch, wie man es auf dem Papier machen würde. Damit sich Überträge nicht unübersichtlich anhäufen, sollte man aber die Summe nicht erst am Schluss bilden, sondern nach jedem Schritt eine Zwischensumme. Das folgende Beispiel zeigt die Überträge nicht und bildet nur einmal die Summe am Schluss. Alle Zahlen sind als positiv zu interpretieren.

Multiplikation:	010110 · 101110	22 · 46
Java *	010110	88
	000000	132
	010110	
	010110	
	010110	
	000000	
	01111110100	1012

Auch die Division wird wie auf dem Papier durchgeführt. Wo man im Dezimalsystem aber abschätzen muss, wie oft der Divisor in den Dividenden passt, gibt es bei den Dualzahlen nur die Frage ob er passt

oder nicht. Es ist also pro Divisionsschritt nur ein Vergleich nötig. Auch im folgenden Beispiel sind alle Zahlen als positiv zu interpretieren.

```
Division: 110110 : 101 = 001010      54 : 5 = 10 Rest 4
Java /    0
          11
           0
          110
          101
            11
             0
            111
            101
             100
```

Das Ergebnis des bitweisen AND-Operators hat dort eine 1, wo beide Operanden eine 1 haben, sonst 0. Das Ergebnis des bitweisen OR-Operators hat dort eine 0, wo beide Operanden eine 0 haben, sonst 1. Das Ergebnis des bitweisen XOR-Operators hat dort eine 1, wo die beiden Operanden verschieden sind, sonst 0.

```
AND:    01101101        OR:     01001101        XOR:    01101001
Java &  11110110        Java |  11100100        Java ^  11110100
        01100100                11101101                10011101
```

Bei den logischen Schiebe-Operationen werden von links oder von rechts eine bestimmte Anzahl von Nullen eingeschoben, bei den arithmetischen von links das höchste Bit, so dass das Vorzeichen erhalten bleibt. Schieben nach links um eine Stelle verdoppelt den Wert; Schieben nach rechts halbiert ihn. Der zweite Operand gibt an, um wie viele Stellen geschoben wird. Mehr als 32 bei `int` bzw. 64 bei `long` ist also nicht sinnvoll.

```
SHL:    11001001        SHR:    11001001        ASR:    11001001
Java << 00000011        Java >>> 00000011       Java >> 00000011
        01001000                00011001                11111001
```

Schließlich hat man noch manchmal einen NOT-Operator, der einfach alle Bits umdreht. Man spricht auch vom Komplement oder genauer vom 1-Komplement.

```
NOT:    11001001
Java ~  00110110
```

C.4 Gleitpunktzahlen

Bisher haben wir nur ganze Zahlen (Integer-Typen) betrachtet. Viele mathematische Probleme (auch einfacher Art) benötigen jedoch einen viel größeren oder kleineren Zahlenbereich als dies mit ganzzahligen Typen möglich ist. Wir untersuchen hier die weit verbreiteten Gleitpunktzahlen[1] (Floating-Point-Numbers) nach dem ANSI/IEEE 754 Standard. Er wurde 1985 vereinbart und findet z. B. bei Intel seit dem 80286 Verwendung. Wir beschränken uns auf die 32-Bit-Zahlen. Die Idee ist mit 64-Bit oder noch mehr die gleiche.

Die 32 Bits werden wie folgt aufgeteilt:

- Bit 31 ist das Vorzeichen.
- die 8 Bits mit Nummern 30 bis 23 stehen für den Exponenten zur Basis 2.
- die restlichen 23 Bits 22 bis 0 stehen für die Mantisse.

Am Beispiel der Zahl $z = 3,125$ sei das erläutert: In Dualdarstellung ist $z = 11,001_2 = 11,001_2 \cdot 2^0$. Dafür gibt es aber noch viele andere Darstellungen. Man braucht nur das Komma zu verschieben und den Exponenten anzugleichen: $z = 0,11001_2 \cdot 2^{10_2} = 110,01_2 \cdot 2^{-1}$. Um die Darstellung eindeutig zu machen,

[1] Der deutschere Begriff dafür ist „Fließkommazahlen“, da diese aber beim Programmieren immer mit dem Punkt als Trennzeichen einzugeben sind, bleiben wir gleich bei der internationalen Version.

wird die Zahl *normalisiert*, d. h. Komma und Exponent in der Weise verändert, dass genau eine 1 vor dem Komma steht, also $z = 1,1001_2 \cdot 2^1$. Weil sich alle Zahlen (außer 0, dazu später) so darstellen lassen, kann man nochmal ein Bit sparen. Die 1 vor dem Komma ist ja sicher, muss also nicht aufgeschrieben, sondern nur dazugedacht werden.

Da z positiv ist, ist Bit 31 auf 0 gesetzt. Dann kommt der Exponent in Exzess-127-Darstellung, d. h. die 1 wird als $1 + 1111111_2 = 10000000_2$ abgespeichert. Von der Mantisse 11001 wird die oberste 1 nicht abgespeichert, also nur 1001. Das sind natürlich nur 4 Bits von 23. Die restlichen sind 0, was eine sehr große Genauigkeit bedeutet. Die 32 Bits sind also mit unserem z wie folgt belegt:

±	$_{30}$Exponent$_{23}$	$_{22}$ Betrag der Mantisse $_{0}$
0	10000000	10010000000000000000000

Erläuterungen und Besonderheiten

- Die Mantisse wird nicht als 2-Komplement gespeichert, sondern in der Form Vorzeichen-Betrag, weil man damit leicht und schnell erkennen kann, welche von zwei Mantissen die größere ist. Dies ist bei Addition und Subtraktion vorrangig, wie wir später noch sehen werden. Bei Multiplikation und Division bringt die 2-Komplement-Darstellung sowieso nichts.
- Auch Exponenten müssen in erster Linie schnell vergleichbar sein, allerdings darf bei diesen Vergleichen das Vorzeichen nicht ausgespart werden. Deshalb wurde die Exzess-Darstellung gewählt. Übrigens sind die Exponenten 00000000 und 11111111 nicht für die oben geschilderte Zahlendarstellung vorgesehen. Diese beiden Werte sind reserviert für besondere Zahlen. Es gibt also nur Exponenten zwischen -126 und $+127$.
- Die Mantisse beinhaltet, wie oben schon gesagt, nur den Nachkommateil der Zahl. Vor dem Komma ist eine 1 zu denken. Man spart dieses Bit und gewinnt dadurch eine Stelle mehr an Genauigkeit.
- Die Zahl 0 wird dargestellt als lauter Nullen in Exponent und Mantisse. Wegen des separaten Vorzeichen-Bits hat man zwei Nullen -0 und $+0$.
- Bei extrem kleinen Zahlen reicht evtl. der kleinste Exponent -126 nicht aus. Statt die Zahl auf Null zu setzen bedient man sich des kleinen Tricks der *Denormalisierung*. Dazu wird bei der Mantisse keine 1 mehr vor dem Komma dazugedacht, wodurch diese dann eben auch kleiner werden kann. Dass dieser Fall vorliegt, wird durch den Exponenten 00000000 ausgedrückt, der aber auch als -126 interpretiert werden muss (siehe Beispiel!).
- Es gibt die beiden „Zahlen“ $-\infty$ und $+\infty$, die z. B. bei der Division durch Null entstehen: $5 : 0 = \infty$, $-5 : 0 = -\infty$. Der Exponent ist dabei 11111111 und die Mantisse 0.
- Als letzte Besonderheit bleibt der Fall, dass der Exponent 11111111 und die Mantisse ungleich 0 ist. Das bedeutet dann, dass das Ergebnis ganz unbrauchbar ist. Es handelt sich hierbei um keine Zahl *NaN* („Not a Number“). NaN's treten z. B. auf bei $0 : 0$, $\sqrt{-1}$, $\arcsin 2$, usw. Die NaN's werden noch unterschieden in die Kategorien *quiet* und *signaling*. Quiet NaN's dürfen in Rechnungen verwendet werden; das Ergebnis ist aber immer NaN. Signaling NaN's werden als dermaßen ungewöhnliches Ergebnis betrachtet, dass es unterbunden werden muss, sie in Rechnungen zu verwenden. Der Prozessor unterbricht den Vorgang. Ist bei einer NaN Bit 22 gesetzt, so handelt es sich um die quiet-Form.
- Bei den 64-Bit-Gleitpunktzahlen sieht alles ganz ähnlich aus. Der Exponent hat jedoch 11 Bits und die Mantisse 52 Bits.

C.5 Rechnen mit Gleitpunktzahlen

Bei Addition und Subtraktion wird evtl. die Zahl mit dem kleineren Betrag an die mit dem größeren angepasst, also denormalisiert. Den kleineren Betrag hat diejenige mit dem kleineren Exponenten oder, falls die Exponenten gleich sind, diejenige mit der kleineren Mantisse. (An dieser Stelle wird klar, warum der Exponent vor der Mantisse abgespeichert wird: Es ist dadurch nämlich nur *ein* Vergleich nötig.)

- Sind die Exponenten verschieden, so wird die Zahl mit dem kleineren denormalisiert, bis ihr Exponent so groß ist wie der große. Der Nachteil ist natürlich, dass die denormalisierte Zahl einige Stellen Genauigkeit verliert (sie wird ja nach rechts geschoben, damit der Exponent wachsen darf).

- Sind die Exponenten (dann) gleich, so können die beiden Mantissen sofort addiert oder voneinander subtrahiert werden. Der Exponent wird einfach übernommen. Das Ergebnis muss lediglich normalisiert werden, falls die Mantisse bei der Rechnung größergleich 2 oder kleiner als 1 geworden ist.

Ziffern, mit denen der Prozessor umgehen muss, obwohl sie nicht in der Zahlendarstellung auftreten (manche Überträge und die hinzugedachte 1) werden in den folgenden Tabellen tiefgestellt eingefügt.

Beispielrechnung:
$1,1101_2 + 0,011_2 = 10,0011_2$
bzw.
$1,8125 + 0,375 = 2,1875$

	±	Exponent	Betrag der Mantisse
	0	01111111	$_1$10100000000000000000000
	0	01111101	$_1$10000000000000000000000
	0	01111111	$_1$10100000000000000000000
denormalisiert	0	01111111	$_0$01100000000000000000000
Summe	0	01111111	$_{10}$00110000000000000000000
normalisiert	0	10000000	$_1$00011000000000000000000

Bei Multiplikation und Division werden beide Mantissen multipliziert bzw. dividiert. Die Exponenten werden addiert bzw. subtrahiert. (Natürlich nicht wörtlich: der Exzess darf ja nicht zweimal auftauchen!) Bleibt die Mantisse nicht im vorgegebenen Rahmen, so wird sie wieder normalisiert.

Beispielrechnung:
$11,101_2 \cdot 0,0011_2 = 0,1010111_2$
bzw.
$3,625 \cdot 0,1875 = 0,6796875$

	±	Exponent	Betrag der Mantisse
	0	10000000	$_1$10100000000000000000000
	0	01111100	$_1$10000000000000000000000
Produkt	0	01111101	$_{10}$10111000000000000000000
normalisiert	0	01111110	$_1$01011100000000000000000

C.6 Weitere Beispiele

Die folgenden Darstellungen sind immer dreigeteilt. Der erste Teil ist stets das Vorzeichenbit, danach kommt der Exponent und zum Schluss die Mantisse. Soll ein Bit gezeigt werden, das in der IEEE 754-Darstellung nicht abgespeichert wird, so wird es tiefgestellt. Wenn kein tiefgestelltes Bit in der Darstellung erscheint, so ist es doch nichtsdestoweniger hinzuzudenken.

Wir subtrahieren als erstes $3221,57 - 0,57$. Der Nachkommaanteil ist binär nicht darstellbar, weil unendlich periodisch, aber das muss uns nicht kümmern, weil die Genauigkeit bei Gleitpunktzahlen nun einmal beschränkt ist.

$$\begin{aligned} 3221,57 &\approx 1{,}10010010101100100011110 \cdot 2^{11} \\ &= 0(138)\ {}_1{,}10010010101100100011110 \\ 0,57 &\approx 1{,}00100011110101110000101 \cdot 2^{-1} \\ &= 0(126)\ {}_1{,}00100011110101110000101 \end{aligned}$$

Bei Addition und Subtraktion wird die Zahl mit dem kleineren Exponenten angepasst (denormalisiert):

$$\begin{aligned} 3221,57 &= 0(138)\ {}_1{,}10010010101100100011110 \\ -0,57 &= 0(138)\ {}_0{,}00000000000_1 00100011110 \\ \hline 3221,00 &= 0(138)\ {}_1{,}10010010101000000000000 \end{aligned}$$

Nun sehen wir uns noch die kleinsten darstellbaren Zahlen an. Die kleinste normalisierte Zahl ist

$$1 \cdot 2^{-126} = 0(1)_1, 00000000000000000000000$$

Wäre der Exponent 0 auch erlaubt, so wäre natürlich die betraglich kleinste darstellbare Zahl

$$\left[1 \cdot 2^{-127} = 0(0)_1, 00000000000000000000000\right]$$

was darauf hinaus liefe, dass die Zahl Null selbst gar nicht darstellbar wäre, weil ja schon alle Bits auf 0 stehen. Tatsächlich wird beim Exponenten 0 aber alles anders interpretiert. Es wird denormalisiert, d. h.

statt der stets hinzuzudenkenden 1 vor dem Komma ist jetzt ausnahmsweise eine 0 vor dem Komma zu denken. Die nächst kleinere Zahl vor $1 \cdot 2^{-126}$ wird dann so gespeichert

$$1 \cdot 2^{-126} \approx 0(0)_0, 11111111111111111111111$$

Der Exponent 0 ist also nicht als $\cdot 2^{-127}$ sondern auch als $\cdot 2^{-126}$ zu interpretieren, weil ja die bisher hinzugedachte 1 entfallen ist.

Dadurch ist es jetzt möglich, noch kleinere Zahlen darzustellen, allerdings mit der Einschränkung, dass immer weniger Bits der Mantisse für die Genauigkeit der Darstellung verwendet werden, weil immer mehr Bits der Mantisse gebraucht werden, um die Kleinheit der Zahl darzustellen. So ist jetzt z. B.

$$\begin{aligned} 1 \cdot 2^{-127} &= 0(0)_0, 10000000000000000000000 \\ 1 \cdot 2^{-128} &= 0(0)_0, 01000000000000000000000 \\ 1 \cdot 2^{-129} &= 0(0)_0, 00100000000000000000000 \\ &\vdots \\ 1 \cdot 2^{-149} &= 0(0)_0, 00000000000000000000001 \end{aligned}$$

Der Übergang zur Zahl Null ist also denkbar fließend. Die Null ist nämlich

$$0 = 0(0)_0, 00000000000000000000000$$

Die größte darstellbare Zahl ($< \infty$) ist übrigens knapp unter $1 \cdot 2^{128}$

$$1 \cdot 2^{128} \approx 0(254)_1, 11111111111111111111111$$

C.7 ASCII und Unicode

Seit 1960 gibt es den ASCII (American Standard Code for Information Interchange). Er dient der Codierung der wichtigsten Zeichen. Das sind 26 Großbuchstaben, 26 Kleinbuchstaben, Ziffern, Satzzeichen und einige Sonderzeichen. Mit 7 Bits gibt es 128 Positionen, die, wie in Tabelle C.3 gezeigt, angeordnet sind.

Wenn ein alter Drucker oder Telex die Bitkombination 65 = `0x41` bekommt, druckt er ein 'A'. Die Zeichen 0 bis 31 und 127 (0 bis `0x1f` und `0x7f`) werden von Druckern nicht auf das Papier gedruckt, sondern steuern sein Verhalten. Code 10 (LF: LineFeed) dreht die Walze um eine Zeile weiter, Code 13 (CR: Carriage Return) fährt den Schlitten nach links zurück[2] und Code 27 (ESC: Escape) dient dazu, dem Drucker zu sagen, dass das was jetzt kommt, eine längere private Mitteilung für ihn ist. Die Bedeutung dieser Kontrollsequenzen unterscheidet sich von Drucker zu Drucker. Z. B. könnte ESC J Z bedeuten, dass ab jetzt der Zeilenabstand $\frac{1}{2}$ Inch betragen soll. ('J' steht für „setze den Zeilenabstand in 180stel Inch", 'Z' ist das Zeichen mit Code 90.)

Da Jahrzehnte lang 1 Byte die Standardgröße war, wurde ASCII auf 8 Bit erweitert durch den ISO-8859-Latin 1-Zeichensatz. Dieser enthält nationale Sonderzeichen wie Umlaute, griechische Buchstaben u.ä. Da gibt es aber Inkompatibilitäten mit den Druckern. Es gibt zwar auf Position 142 das 'Ä', der Drucker war aber vielleicht so programmiert, dass er Code 142 ignoriert und statt dessen die Sequenz ESC A : erwartet.

Man beachte einige Besonderheiten der Anordnung: Den Wert der Ziffern bekommt man z. B. dadurch, dass man von ihrem Code `0x30` ausblendet. Bei den Buchstaben verhält es sich ähnlich. ‚A' liegt auf Position `0x41`, damit man die Lage im Alphabet durch Ausblenden von `0x40` erhalten kann. Die Umschaltung Groß-Klein kann man einfach durch Änderung eines Bits bewerkstelligen.

Unicode ist nach vielen Jahren ein Standard, der endlich auch Buchstaben und Zeichen anderer Sprachen (z. B. Chinesisch) beinhaltet. Das wird möglich durch die 16 Bit-Darstellung, die 65536 Positionen hat. Die untersten 256 Poitionen sind vernünftigerweise wie Latin 1 belegt. Im Moment sind noch nicht alle Positionen vergeben; ein Konsortium bemüht sich, auf dass die Anordnung eine innere Logik habe. Es wurde auch beschlossen, dass ein Bereich von 6400 Positionen niemals standardmäßig belegt sondern freigelassen wird für besondere Erfordernisse einzelner Programme.

Damit man auf einer Tastatur von etwa 100 Tasten all diese Zeichen eingeben kann, gibt es in Java die Notation `\u`*xxxx*, wobei *xxxx* die Position als Hexadezimalzahl ist. In anderen Texten liest man auch `U+`*xxxx*.

[2] In MS-DOS und Windows werden Zeilen immer noch durch diese beiden Zeichen beendet.

Tabelle C.3: ASCII-Standard

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0x0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
0x1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
0x2		!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
0x3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0x4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0x5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0x6	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0x7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Die Übertragung von Textdateien nach dem neuen Unicode-Standard dauert jetzt doppelt so lange wie früher, weil ja alle Zeichen jetzt 16 Bits haben. Das ist meist eine Verschwendung, weil natürlich der Hauptbestandteil von Textdateien in unserer Gegend aus ASCII-Zeichen besteht. Aus diesem Grund wurde die UTF-8-Dateiübertragung definiert, die wir hier kurz besprechen:

Tabelle C.4: UTF-8-Darstellung

Unicode	Bits	UTF-8 (x steht für 0 oder 1)
\u0000...\u007f	7	0xxxxxxx
\u0080...\u07ff	11	110xxxxx 10xxxxxx
\u0800...\uffff	16	1110xxxx 10xxxxxx 10xxxxxx

Damit ist jede ASCII-Zeichenkette auch gültiger Unicode. Bei reinem ASCII braucht man also wieder nur 1 Byte pro Zeichen, dafür braucht man für sehr seltene Zeichen 3 Bytes. Durch Auswertung der oberen Bits eines jeden übertragenen Bytes kann man leicht erkennen, wo ein neues Zeichen beginnt.

D Systematisch von Pascal zu Java

D.1 Umwandlungsmaßnahmen

Die systematische Umwandlung eines Programms von einer in die andere Sprache ist im Grunde ein Unding. Wenn das Programm wichtig ist, programmiert man besser gewissenhaft neu, um die Fähigkeiten der Zielsprache optimal auszunutzen. Wenn es unwichtig ist, programmiert man gar nicht. Die hier gezeigte relativ ausführliche Methode, ein Pascal- in ein Java-Programm umzuwandeln, soll auch nur dazu dienen, die Möglichkeiten von Java schneller zu begreifen, wenn man Pascal schon gut kennt.

- Zu allen Pascal-Datentypen gibt es eine brauchbare Entsprechung in Java (siehe Seite 42). Lediglich `record`-Typen müssen als eigene Klassen ausgeführt werden. Der Klassenname ist dann der Name des `record`-Typs. Die `record`-Variablen werden Attribute.
- In Java bestehen die meisten Programme aus mehreren Klassen, die fast immer aus mehreren Quellcode-Dateien compiliert werden. Das umzuwandelnde Pascal-Programm wird also meist in mehrere Java-Klassen übertragen. Nichtsdestotrotz gibt es immer eine Art Hauptprogramm (mit der `main`-Methode), das den gleichen Namen erhält, wie das ursprüngliche Pascal-Programm. In die `main`-Methode kommt alles, was im Pascal-Hauptprogramm zwischen dem ersten `begin` und dem letzten `end` steht.
- Globale Variablen werden statische Variablen des Java-Hauptprogramms.
- Wenn im Pascal-Programm unterschieden wird zwischen einem Typ `TTyp` und einem Zeiger `PTyp` auf den Typ `TTyp`, dann verwendet man in Java nur die gemeinsame Klasse `Typ`. (Über die Unterscheidung zwischen einer Variablen und dem Zeiger auf eine solche siehe Abschnitt D.2.)
- Funktionen werden statische Methoden des Hauptprogramms. Prozeduren ebenso, aber mit dem Rückgabetyp `void`.
- Dynamische Speicherplatzanforderungen mit `new` werden zu Konstruktoraufrufen.
- Aus `write` und `writeln` wird `System.out.print` und `System.out.println`. Für Eingaben von der Tastatur bedarf es einer Hilfsklasse, wie auf Seite 10 beschrieben. Komplexere Leistungen, wie sie etwa von systemnahen Units erbracht werden, können nicht leicht in Java umgewandelt werden. Hier muss von Fall zu Fall in den vorhandenen Java-Packages gesucht werden.
- Pascal-Units werden zu eigenen Java-Klassen. Die Variablen und Funktionen im `interface`-Teil werden zu Attributen und Methoden mit Zugriffsstufe `public`, diejenigen, die nur im `implementation`-Teil auftauchen, werden zu solchen mit der Zugriffsstufe `private`. Sollte die Unit einen Initialisierungsabschnitt haben (also ein Unit-globales Programmstück zwischen `begin` und `end`), so muss entsprechender Code als `static`-Initializer umgesetzt werden[1].
- Pascal-`nil` ist Java-`null`.

D.2 Paramterübergabe bei Funktionsaufrufen

Standardpascal-Funktionen liefern keine komplexen Datentypen (wie etwa Records) zurück[2], können aber solche als Argumente erhalten. Bei der Parameterübergabe bietet Pascal mehr Möglichkeiten unter Inkaufnahme von größeren Risiken.

Während in Java einfache Typen immer als Wert und Objekttypen immer als Referenz übergeben werden, bietet Pascal durch die optionale Markierung eines Parameters mit `var` die Möglichkeit, diesen als Referenz zu übergeben. Ohne `var` wird er als Wert übergeben, d. h. es wird eine Kopie des Variableninhalts

[1] Dieses Konstrukt ist selten und wurde hier noch nicht erwähnt. Man kann es an die gleichen Stellen eines Quelltextes schreiben wie etwa eine Methodendeklaration. Es handelt sich um einen durch geschweifte Klammern begrenzten Block von Befehlen. Vor dem Block steht das Schlüsselwort `static`. Der Block wird genau einmal abgearbeitet, wenn die Klasse das erste Mal eingeladen wird.

[2] Solche Einschränkungen wurden bei Erweiterungssprachen wie etwa Delphi-Pascal natürlich längst aufgehoben.

erzeugt und dann übergeben, auch wenn diese Kopie viel Speicher benötigt. Der Programmierer hat es also in der Hand, ob er einer Funktion seine Originalvariable anvertraut oder nur eine Kopie. Will man in Java unbedingt nur eine Kopie übergeben, so muss man diese vorher eigens erzeugen, was aber üblicherweise ganz einfach geht mit `einobjekt.clone()`[3].

Problematisch wird die Sache in Pascal dadurch, dass es Zeiger (Referenzen) auch als eigenständigen Typ gibt. Variablen dieses Typs können bei Funktionsaufrufen nun aber ebenfalls mit oder ohne `var`-Kennzeichnung verwendet werden. Im folgenden sei `z` der Name einer Pascal-Variablen mit einem Zeigertyp, d. h. `z` ist im wesentlichen eine Speicheradresse, `z^` hingegen ist der Variablenwert, der im Speicher ab der Adresse `z` abgelegt ist.

1. Eine Prozedur sei definiert als `procedure tuWasMit(w:TKeinZeiger);`

   Beim Aufruf von `tuWasMit(z^)` wird von `z^` eine Kopie angefertigt und unter dem Namen `w` übergeben. Das Original kann von der Prozedur nicht verändert werden.

2. Eine Prozedur sei definiert als `procedure tuWasMit(var w:TKeinZeiger);`

   Hier wird mit `tuWasMit(z^)` im Unterschied zu 1 keine Kopie angefertigt, sondern am Original herumoperiert. Das passiert dem Aufrufer (der ja den Code mit dem entscheidenden `var` gar nicht kennen muss), obwohl er keine Zeigervariable übergibt! Dagegen kann man aber auch in Java nichts tun. Immerhin übergibt man da aber (bewusst, weil gar nicht anders möglich) einen Zeiger und ist gewarnt. (Die meisten Objekte sind so groß, dass man sie bei Methodenaufrufen standardmäßig nicht klont.)

3. Eine Prozedur sei definiert als `procedure tuWasMit(w:TZeiger);`

   Jetzt muss man die Prozedur mit `tuWasMit(z)` aufrufen. Dass am Original herumoperiert werden kann, muss dem Aufrufer klar sein, weil er ja bewusst einen Zeiger übergeben hat. Es ist ihm dann bewusst, wenn er gewohnt ist, einen Zeiger von dem zu unterscheiden, worauf ein solcher zeigt. Diese Unterscheidung sollte also besonders in Pascal niemals als nebensächlich betrachtet werden.

4. Eine Prozedur sei definiert als `procedure tuWasMit(var w:TZeiger);`

   An dieser Stelle geht man in Pascal ernsthaft das Risiko schwer zu findender Fehler ein. Die Prozedur kann dem Aufrufer nicht nur das Original verändern, sondern es ihm völlig wegnehmen und gegen etwas anderes austauschen. Die Prozedur hat nämlich Zugriff auf die Speicherzelle, in der der Aufrufer sich merkt, wo sein Original ist. Ist `tuWasMit` schlecht oder böse programmiert, so kann es durch den Aufruf `tuWasMit(z)` nicht nur passieren, dass `z^` verändert wird (Fälle 2 und 3), sondern z. B. dass `z=nil`, der Inhalt `z^` also unwiederbringlich im Speicher verschollen ist. Dass `z^` damit unter anderem nicht mehr aufgeräumt werden kann, versteht sich von selbst[4].

   Fazit: Es reicht nicht, selber und hier ordentlich zu programmieren. Es müssen alle und überall gut aufpassen. Fehler sind schwer zu finden und können von hinterhältiger Gestalt sein!

D.3 Beispiel zur Umwandlung von Pascal in Java

Eine Pascal-Unit `Kollektion`, die mit Hilfe einer dynamischen verketteten Liste ein beliebig großes Multiset von Integer-Zahlen verwaltet, wird umgeschrieben in Java. Es wäre ein leichtes, die Funktionalität gleich noch ein bisschen zu verbessern, aber darum geht es jetzt nicht. Ein kleines Hilfsprogramm `Zahlen`, das diese Unit verwendet, wird ebenfalls umgeschrieben.

Hier nun erst einmal der ursprüngliche Pascal-Code der Unit

```
unit Kollektion;

interface
  function  istLeer:boolean;
  procedure einfuegen(zahl:integer);
  procedure entfernen(zahl:integer);
  function  wieOftDrin(zahl:integer):integer;

implementation
  type
```

[3] Eine genauere Diskussion dieser Methode finden Sie im Beispiel auf Seite 20.

[4] In Java ist das die Standardmethode, um ein Objekt aufzuräumen. Der Garbage-Collector stellt nämlich fest, dass keine Referenz mehr auf das Objekt zeigt und gibt den belegten Speicher frei.

```
  PKnoten=^TKnoten;
  TKnoten=record
    info:integer;
    next:PKnoten
  end;

var start:PKnoten;

procedure init;
begin
  start:=nil
end;

function istLeer:boolean;
begin
  istLeer:=(start=nil)
end;

procedure einfuegen(zahl:integer);
var element:PKnoten;
begin
  new(element);
  element^.next:=start;
  element^.info:=zahl;
  start:=element;
end;

function wieOftDrin(zahl:integer):integer;
var p:PKnoten;
    n:integer;
begin
  n:=0;
  p:=start;
  while p<>nil do begin
    if p^.info=zahl then n:=n+1;
    p:=p^.next;
  end;
  wieOftDrin:=n;
end;

procedure entfernen(zahl:integer);
var zero:TKnoten;
    vor,lauf:PKnoten;
begin
  zero.next:=start;
  vor :=@zero;
  lauf:=start;
  while lauf<>nil do begin
    if lauf^.info=zahl then begin
      vor^.next:=lauf^.next;
      dispose(lauf);
      lauf:=nil;
      start:=zero.next;
    end
    else begin
      vor :=lauf;
      lauf:=lauf^.next;
    end;
  end;
end;

{Die Kollektion muss vor dem Gebrauch initialisiert werden.}
begin
  init;
end.
```

Beachten Sie bitte, dass die folgende Klasse den (trivialen) Defaultkonstruktor automatisch bekommt, auch wenn er nicht ausdrücklich im Quelltext erscheint. Interessant ist auch die Methode `entfernen`.

Man sieht, wie elegant in Java die Unterscheidung zwischen dem Dereferenzierungs-Operator ^ und dem Adress-Operator @ ausfällt. Ein Objektname, der von keinem Punkt gefolgt ist, bedeutet die Adresse des Objekts. Mit Punkt und weiterem Namensteil wird dereferenziert.

```
public class Kollektion{
  private static Knoten start;

  private static void init(){
    start=null;
  }

  public static boolean istLeer(){
    return start==null;
  }

  public static void einfuegen(int zahl){
    Knoten element=new Knoten();
    element.next=start;
    element.info=zahl;
    start=element;
  }

  public static int wieOftDrin(int zahl){
    int n=0;
    Knoten p=start;
    while(p!=null){
      if(p.info==zahl) n++;
      p=p.next;
    }
    return n;
  }

  public static void entfernen(int zahl){
    Knoten zero=new Knoten();
    zero.next=start;
    Knoten vor =zero;
    Knoten lauf=start;
    while(lauf!=null){
      if(lauf.info==zahl){
        vor.next=lauf.next;
        lauf=null;//so etwas wie dispose ist nicht nötig
        start=zero.next;
      } else{
        vor =lauf;
        lauf=lauf.next;
      }
    }
  }

  //Die Kollektion muss vor dem Gebrauch initialisiert werden.
  static{
    init();
  }
}

class Knoten{
  public int info;
  public Knoten next;
}
```

Hier noch der Quellcode des Hilfsprogramms

```
program Zahlen;
uses kollektion;
var zahl:integer;
    ant :string;
begin
  writeln('Geben Sie ganze Zahlen ein.');
```

```
  writeln('Wenn positiv, werden sie gemerkt.');
  writeln('Wenn negativ, wird die entsprechende positive entfernt.');
  writeln('0 beendet das ganze.');
  repeat
    readln(zahl);
    if zahl<0 then begin
      zahl:=-zahl;
      write('Wollen Sie sie ',zahl,' wirklich entfernen?');
      readln(ant);
      if ant<>'n' then kollektion.entfernen(zahl);
    end
    else if zahl>0 then begin
      write('Wollen Sie sie ',zahl,' wirklich hinzufügen?');
      readln(ant);
      if ant<>'n' then kollektion.einfuegen(zahl);
    end;
    writeln(zahl,' ist ',kollektion.wieOftDrin(zahl),' mal enthalten.');
  until zahl=0;
end.
```

Das zugehörige Java-Programm sieht so aus

```
public class Zahlen{
  public static void main(String[] args){
    System.out.println("Geben Sie ganze Zahlen ein.");
    System.out.println("Wenn positiv, werden sie gemerkt.");
    System.out.println("Wenn negativ, wird die entsprechende "
                      +"positive entfernt.");
    System.out.println("0 beendet das ganze.");
    int zahl;
    String ant;
    do{
      zahl=LineInput.readInt();
      if(zahl<0){
        zahl=-zahl;
        System.out.print("Wollen Sie sie "+zahl+" wirklich entfernen?");
        ant=LineInput.readString();
        if(!ant.equals("n")) Kollektion.entfernen(zahl);
      } else if(zahl>0){
        System.out.print("Wollen Sie sie "+zahl+" wirklich hinzufügen?");
        ant=LineInput.readString();
        if(!ant.equals("n")) Kollektion.einfuegen(zahl);
      }
      System.out.println(zahl+" ist "+Kollektion.wieOftDrin(zahl)
                         +" mal enthalten.");
    } while(zahl!=0);
  }
}
```

Literaturverzeichnis

[1] Hessisches Kultusministerium (Hrsg.). *Lehrplan Informatik, Gymnasialer Bildungsgang, Jahrgangsstufe 11 bis 13*; Wiesbaden; 2003.

[2] Hendrich, Norman. *Java für Fortgeschrittene*; Springer Verlag; Berlin, Heidelberg; 1997

Dieses Buch ist ein lange brauchbarer Begleiter. Immer wieder entdeckt man darin vertiefende Gedanken, die zum fortwährenden Lernprozess passen. Dass das Buch nicht mehr an vorderster Front aktuell ist, ist für den Anfang unbedeutend.

[3] Flanagan, David. *Java In A Nutshell, 2. Auflage*; O'Reilly; Köln; 1998

Der Autor ist berühmt für seinen prägnanten, schnörkellosen Stil. Wer sich lieber an den Rechner setzt und selber ausprobiert, als vorher viel zu lesen, kommt mit diesem Buch schnell in Schwung. Dass es trotzdem über 600 Seiten hat, liegt daran, dass mehr als die Hälfte des Textes einen Abriss der Java-Dokumentation wieder gibt. Auch darin findet man oft wichtige Tipps, aber da der Bastler die Dokumentation sowieso vor sich auf dem Rechner hat, ist das eher Papierverschwendung. Rat: Ausleihen

[4] Flanagan, David. *Java Examples In A Nutshell*; O'Reilly; Köln; 1998

Nochmal 500 höchst prägnante Seiten von dem Autor, der seine Quelltexte so vorbildlich dokumentiert und das zu allen irgendwann interessierenden Themen. Ein sehr empfehlenswertes Buch.

[5] Campione, Mary; Walrath, Kathy. *The Java Tutorial*; Addison Wesley; Reading, Massachusetts; 1999

Das wahrscheinlich beste und ausführlichste Buch um in großer Breite und Tiefe Java zu lernen. Ein nie versiegendes Füllhorn von Details und Ideen. Die HTML-Version finden Sie auf der CD. Es lohnt sich jedoch, etwa alle halbe Jahre auf der Sun-Site nach Updates zu sehen. Es handelt sich nämlich auch um das stets aktuellste Werk.

[6] Arnold, Ken; Gosling, James. *The Java Programming Language, Second Edition*; Addison Wesley; Reading, Massachusetts; 1998

Dieses Buch ist eine gewissenhafte Einführung in Java. Es hat weniger Beispiele als die Bücher von Flanagan und Campione, bietet aber eine tiefer gehende Betrachtung der Sprachkonstrukte und der objektorientierten Philosophie.

[7] Gosling, James; Joy, Bill; Steele, Guy; Bracha, Gilad. *The Java Language Specification, Second Edition*; Addison Wesley; Reading Massachusetts; 2000

Dieses Buch klärt alle Fragen, die über Java-Sprachkonstrukte gestellt werden können, vollständig und endgültig. Es ist nicht leicht zu lesen, weil sehr präzise. Wenn man aber etwas genau wissen will, ist die Lektüre unumgänglich. Es handelt sich in etwa um eine Mischung aus einer Grammatik und einem Gesetzbuch. Die vollständige HTML-Version finden Sie auf der CD.

[8] Lindholm, Tim; Yellin, Frank. *The Java Virtual Machine Specification*; Addison Wesley; Reading, Massachusetts; 2000

Hier wird definiert, wie eine *Java Virtual Machine* (JVM) arbeitet. Eine JVM ist vergleichbar mit einem Prozessor und der Bytecode, den sie verarbeitet ist vergleichbar mit den Maschinenbefehlen, die ein Prozessor verarbeitet. Zum durchgängigen Lesen dieses Buches bedarf es technischen Interesses. Zum gelegentlichen Nachschlagen reicht jedoch die HTML-Version, die auf der CD beiliegt, vollständig aus.

Versicherung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Hilfsmittel angefertigt habe. Sämtliche Stellen, die den benutzten Werken im Wort oder dem Sinn nach entnommen sind, wurden mit Quellenangaben kenntlich gemacht.