

Grundlagen der Programmierung

Vorlesungsskript für Wirtschaftsinformatiker des ersten Semesters

Andreas de Vries und Volker Weiß

Version: 19. Juli 2014

Dieses Skript unterliegt der *Creative Commons License* 3.0
(<http://creativecommons.org/licenses/by-nc/3.0/deed.de>)



Inhaltsverzeichnis

1	Grundlegende Sprachelemente von Java	7
1.1	Einführung	7
1.1.1	Wie funktioniert Java?	8
1.1.2	Installation der Java SDK-Umgebung	9
1.1.3	Erstellung von Java-Programmen	9
1.1.4	Die 4 Programmarten von Java	9
1.2	Das erste Programm: Ausgabe eines Textes	10
1.2.1	Wie kommt die Applikation ans Laufen?	11
1.2.2	Ein erster Überblick: So funktioniert das Programm	11
1.2.3	Die Klassendeklaration und Klassennamen	12
1.2.4	Der Programmstart: Die Methode main	12
1.2.5	Die import -Anweisung des swing-Pakets	13
1.2.6	Dialogfenster	13
1.3	Elemente eines Java-Programms	14
1.3.1	Anweisungen und Blöcke	14
1.3.2	Reservierte Wörter und Literale	15
1.3.3	Kommentare	15
1.3.4	import-Anweisung	16
1.4	Strings und Ausgaben	16
1.4.1	Strings	16
1.4.2	Möglichkeiten der Ausgabe	17
1.4.3	Dokumentation der API	18
1.5	Variablen, Datentypen und Operatoren	20
1.5.1	Variablen, Datentypen und Deklarationen	21
1.5.2	Der Zuweisungsoperator =	21
1.5.3	Datenkonvertierung mit dem <i>Cast</i> -Operator	23
1.5.4	Operatoren, Operanden, Präzedenz	23
1.5.5	Spezielle arithmetische Operatoren	25
1.6	Eingaben	26
1.6.1	Strings addieren	26
1.6.2	Eingabedialoge	27
1.6.3	Zahlen addieren	28
1.6.4	Konvertierung von Strings in Zahlen	29
1.6.5	Einlesen von Kommazahlen	30
1.6.6	Weitere Eingabemöglichkeiten	31
1.7	Zusammenfassung	31
2	Prozedurale Programmierung	35
2.1	Logik und Verzweigung	35
2.1.1	Die Verzweigung	35
2.1.2	Logische Bedingungen durch Vergleiche	36

2.1.3	Logische Operatoren	39
2.1.4	Die <code>if-else-if</code> -Leiter	42
2.1.5	Gleichheit von <code>double</code> -Werten	42
2.1.6	Abfragen auf Zahlbereiche	42
2.1.7	Bitweise Operatoren	43
2.1.8	Bedingte Wertzuweisung mit dem Bedingungsoperator	45
2.2	Wiederholung durch Schleifen	45
2.2.1	Die <code>while</code> -Schleife	46
2.2.2	Die Applikation <i>Guthaben</i>	46
2.2.3	Umrechnung Dezimal- in Binärdarstellung	47
2.2.4	Die <code>for</code> -Schleife	48
2.2.5	Die <code>do/while</code> -Schleife	50
2.2.6	Wann welche Schleife?	50
2.3	Statische Methoden	51
2.3.1	Mathematische Funktionen	51
2.3.2	Was ist eine Methode?	52
2.3.3	Eine erste Applikation mit statischen Methoden	55
2.3.4	Statische Methoden: Die Applikation Pythagoras	57
2.4	Arrays	59
2.4.1	Was ist ein Array?	59
2.4.2	Lagerbestände	61
2.5	Die <code>for-each</code> -Schleife	63
2.6	Mehrdimensionale Arrays	63
2.7	Zusammenfassung	64
3	Objektorientierte Programmierung	68
3.1	Die Grundidee der Objektorientierung	68
3.1.1	Was charakterisiert Objekte?	68
3.1.2	Objekte als Instanzen ihrer Klasse	69
3.1.3	Kapselung und Zugriff auf Attributwerte	69
3.1.4	Abfragen und Verändern von Attributwerten: <code>get</code> und <code>set</code>	70
3.2	Klassendiagramme der UML	70
3.3	Programmierung von Objekten	71
3.3.1	Der Konstruktor	71
3.3.2	Die Referenz <code>this</code>	72
3.3.3	Typische Implementierung einer Klasse für Objekte	72
3.3.4	Objektelemente sind nicht statisch	73
3.3.5	Lokale Variablen und Attribute	74
3.3.6	Speicherverwaltung der Virtuellen Maschine	74
3.4	Erzeugen von Objekten	75
3.4.1	Deklaration von Objekten	75
3.4.2	Erzeugung von Objekten: der <code>new</code> -Operator	76
3.4.3	Aufruf von Methoden eines Objekts	76
3.4.4	Besondere Objekterzeugung von Strings und Arrays	76
3.5	Fallbeispiel DJTools	77
3.5.1	Version 1.0	77
3.5.2	Version 2.0: Formatierte Zeitangaben	80
3.6	Zeit- und Datumfunktionen in Java	84
3.6.1	Die Systemzeit	84
3.6.2	<code>GregorianCalendar</code>	85
3.6.3	Datumsformatierungen	86

3.7	Objekte in Interaktion: Fallbeispiel Konten	88
3.8	Öffentlichkeit und Privatsphäre	91
3.8.1	Pakete	91
3.8.2	Verbergen oder Veröffentlichen	92
3.9	Vererbung	94
3.9.1	Überschreiben von Methoden	95
3.9.2	Die Klasse <code>Object</code>	95
3.9.3	Die Referenz super	96
3.9.4	Fallbeispiel: Musiktitel	96
3.10	Schnittstellen (<i>Interfaces</i>)	97
3.11	Zusammenfassung	100
4	Datenstrukturen: Collections	103
4.1	Notation und Grundlagen	103
4.2	Bedeutung sortierter Datenstrukturen	104
4.3	Sortierung in Java	106
4.3.1	Das Interface <code>Comparable</code>	106
4.3.2	Das Interface <code>Comparator</code>	108
4.4	Theoretische Konzepte für Datenstrukturen	111
4.4.1	Die drei Grundfunktionen einer Datenstruktur	111
4.4.2	Verkettete Listen (<i>Linked Lists</i>)	112
4.4.3	Bäume	113
4.5	Java Collections	116
4.5.1	Listen	119
4.5.2	Mengen	119
4.5.3	Maps (Zuordnungen / Verknüpfungen)	120
4.5.4	Wann welche Klasse verwenden?	122
4.6	Statische Methoden der Klasse <code>Collections</code>	122
4.7	Zusammenfassender Überblick	123
A	Spezielle Themen	124
A.1	ASCII und Unicode	124
A.2	Das Binärsystem	124
A.2.1	Umrechnung vom Dezimal- ins Binärsystem	127
A.2.2	Binärbrüche	127
A.3	javadoc, der Dokumentationsgenerator	128
A.3.1	Dokumentationskommentare	129
A.4	jar-Archive: Ausführbare Dateien und Bibliotheken	130
A.4.1	Manifest-Datei	130
A.5	Formatierte Ausgabe mit HTML	131
A.5.1	HTML-Ausgabe von Wahrheitstabellen	132
A.6	Call by reference und call by value	133
A.7	enums	136
A.7.1	Zusammenfassung	139
	Literaturverzeichnis	140

Vorwort

Das vorliegende Skript umfasst den Stoff der Vorlesung *Grundlagen der Programmierung* im Rahmen des Bachelor-Studiengangs Wirtschaftsinformatik am Fachbereich Technische Betriebswirtschaft in Hagen. Es ist die inhaltliche Basis für die obligatorisch zu belegenden Praktika.

Zwar ist das Skript durchaus zum Selbststudium geeignet, auch für Interessierte anderer Fachrichtungen oder Schüler, da keine besonderen Kenntnisse der Programmierung vorausgesetzt werden. Hervorzuheben ist aber die Erkenntnis: Schwimmen lernt man nicht allein durch Anschauen und Auswendiglernen der Bewegungsabläufe, sondern man muss ins Wasser! Ähnlich lernt man das Programmieren nur durch ständiges Arbeiten und Ausprobieren am Rechner. So trifft auch auf das Erlernen einer Programmiersprache das chinesische Sprichwort zu:

Was man hört, das vergisst man.

Was man sieht, daran kann man sich erinnern.

Erst was man tut, kann man verstehen.

Entsprechend kann dieses Skript nicht den Praktikumsbetrieb einer Hochschule ersetzen, in dem Aufgaben selbständig, aber unter Anleitung gelöst werden.

Das Konzept. Die Straffung der Bachelor-Studiengänge gegenüber den traditionellen Diplomstudiengängen macht eine inhaltliche Modifikation auch der klassischen Programmierveranstaltungen im Rahmen des Informatikstudiums notwendig. Dies haben wir als Chance begriffen, das Thema der Datenstrukturen, deren theoretischer Betrachtung bisher ein breiter Raum eingeräumt wurde, direkt durch die praktische Arbeit mit ihnen einzuführen, also gerade in einer Programmierveranstaltung. Mit dem Collection-Framework von Java lässt sich dieses Vorhaben sehr gut umsetzen.

Natürlich ist der knappen Zeit der Veranstaltung Tribut zu zollen, so dass wichtige Bereiche der Informatik hier nicht angesprochen werden können, so das Gebiet der Datenbanken, ohne die das heutige öffentliche und wirtschaftliche Leben nicht mehr funktionieren würde, oder das der Netzwerke, die insbesondere durch die Etablierung des Internet seit Mitte der 1990er Jahre zu einer Revolutionierung der Informationsflüsse unserer Gesellschaft geführt hat und weiter führen wird. Diese Inhalte werden in späteren Veranstaltungen des Studiums behandelt, für die diese Vorlesung die Grundlagen legt.

Wenn Programmierung, welche Programmiersprache wählt man? Es gibt eine große Auswahl, so dass man Kriterien dazu benötigt. Eines dieser Kriterien ist, dass ein Programmieranfänger zunächst mit einer Compilersprache lernen sollte, da einerseits die Suche nach Ursachen für (nicht nur am Anfang) auftretende Fehler durch die Trennung von Syntax- und Laufzeitfehlern einfacher ist, andererseits wichtige Konzepte wie Typisierung in Interpretersprachen nicht vorkommen und daher ein Erlernen einer solchen Sprache nach Kenntnis einer Compilersprache viel einfacher ist als umgekehrt. Ein weiteres Kriterium ist sicherlich die Verbreitung einer Sprache; da zur Zeit ohne Zweifel die Familie der „C-Syntax“ — also Sprachen wie C, C++, Java, C#, aber auch Interpretersprachen wie JavaScript und PHP — die meist verwendete ist, sollte es auch eine derartige Sprache sein.

Die derzeit beste Wahl erscheint uns daher Java. Java ist der wohl optimale Kompromiss aus modernem Konzept, praktikabler Erlernbarkeit und vielfältiger Erweiterbarkeit in speziellere Anwendungsgebiete. Java ist in der relativ kurzen Zeit, die es existiert, zu dem großen Arsenal standardmäßig

bereitgestellter Programme und Softwarekomponenten geworden, die *Java API*. Einen vollständigen und detaillierten Überblick kann man nur allmählich mit intensiver Arbeit am Rechner bekommen.

Inhaltliches Ziel dieses Skripts ist die Vermittlung des Stoffs folgender Themen:

- *Grundlegende Sprachelemente von Java.* (Ausgaben, primitive Datentypen und Strings, Eingaben)
- *Einführung in die prozedurale Programmierung.* (Verzweigungen, logische Operatoren, Schleifen, statische Methoden, Erstellen einfacher Algorithmen)
- *Einführung in objektorientierte Programmierung mit Java.* (Objekte, Objektattribute und -methoden, Klassendiagramme nach UML, Interfaces) Insbesondere soll ein Einblick in die Mächtigkeit gewährt werden, die das Paradigma der Objektorientierung liefert. Die Sicht auf Objekte vereinfacht und „lokalisiert“ viele komplexe Probleme radikal, so dass manches Mal komplizierte Lösungsalgorithmen in einfache Teilalgorithmen zerfallen.
- *Anwendungsorientierte Einführung in Datenstrukturen.* (Verkettete Listen, Bäume, Mengen, Zuordnungen)

Literatur und Weblinks. Die Auswahl an Literatur zu Java ist immens, eine besondere Empfehlung auszusprechen ist fast unmöglich. Unter den deutschsprachigen Büchern sind die folgenden zu erwähnen, die dieses Skript ergänzen: das umfassende Lehrwerk *Einführung in die Informatik* von Gumm und Sommer [5], das *Handbuch der Java-Programmierung* von Guido Krüger [7], geeignet als Nachschlagewerk, und *Java ist auch eine Insel* von Christian Ullenboom [9], ein Überblick über fast alle Bereiche von Java. Sehr gründlich und detailliert sind das *Java 5 Programmierhandbuch* von Ulrike Böttcher und Dirk Frischalowski [1], sowie der zweibändige *Grundkurs Programmieren in Java* von Dietmar Ratz, Jens Scheffler und Detlef Seese [8].

Als Weblink sind die englischsprachigen Tutorials von Sun zu empfehlen, sie geben Überblick über verschiedene Themen von Java („trails“):

<http://docs.oracle.com/javase/tutorial/>

Daneben entwickelt sich aktuell ein interessanter Wiki unter wikibooks.org:

http://de.wikibooks.org/wiki/Java_Standard

Gerade zum Einstieg, aber auch in den fortgeschrittenen Themen ist es eine gute Ergänzung zu dem vorliegenden Skript.

Hagen, den 19. Juli 2014

Andreas de Vries, Volker Weiß

Kapitel 1

Grundlegende Sprachelemente von Java

1.1 Einführung

Java ist eine objektorientierte Programmiersprache. Viele ihrer Fähigkeiten und einen großen Teil der Syntax hat sie von C++ geerbt. Im Gegensatz zu C und C++ ist Java jedoch nicht darauf ausgelegt, möglichst kompakte und extrem leistungsfähige Programme zu erzeugen, sondern soll den Programmierer vor allem dabei unterstützen, sichere und fehlerfreie Programme zu schreiben. Daher sind einige der Fähigkeiten von C++ nicht übernommen worden, die zwar mächtig, aber auch fehlerträchtig sind. Viele Konzepte älterer objektorientierter Sprachen (wie z.B. Smalltalk) wurden übernommen. Java ist also keine wirklich neue Programmiersprache, sondern es vereint bewährte Konzepte früherer Sprachen. Siehe dazu auch den Stammbaum in Abb. 1.1.

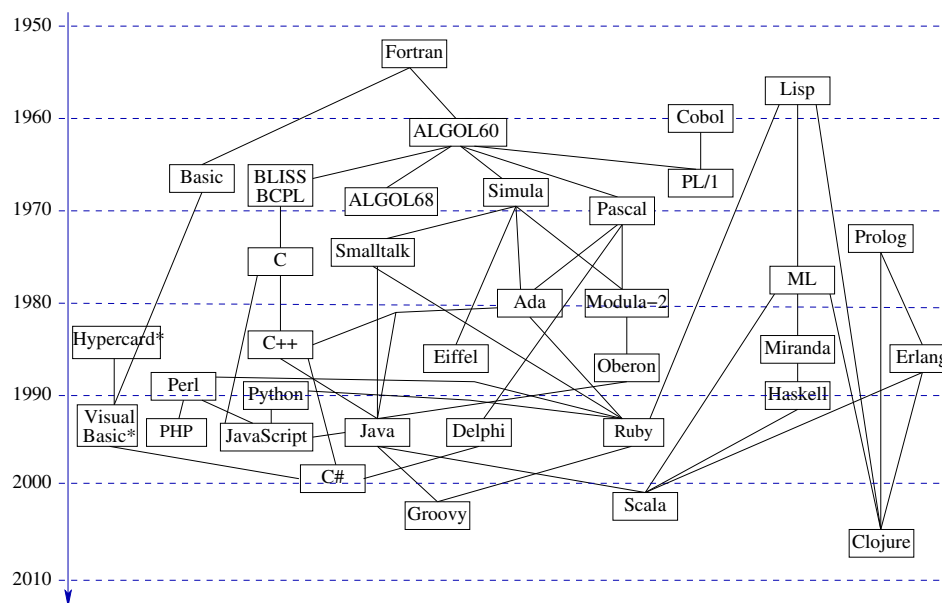


Abbildung 1.1: Stammbaum der gängigen Programmiersprachen (*Visual Basic ist eine nur auf Windows-Systemen lauffähige Programmiersprache, Hypercard eine Makrosprache von Apple-MacIntosh).

Java wurde ab 1991 bei der Firma Sun Microsystems entwickelt. In einem internen Forschungsprojekt namens *Green* unter der Leitung von James Gosling sollte eine Programmiersprache zur Steuerung von Geräten der Haushaltselektronik entstehen. Gosling nannte die Sprache zunächst *Oak*, nach einer Eiche vor seinem Büfenster. Als später entdeckt wurde, dass es bereits eine Programmiersprache *Oak* gab, wurde ihr angeblich in einem Café der Name *Java* gegeben. *Java* ist ein umgangssprachliches Wort für Kaffee.

Da sich, entgegen den Erwartungen der Strategen von Sun, der Markt für intelligente Haushaltsgeräte nicht so schnell und tiefgreifend entwickelte, hatte das Green-Projekt große Schwierigkeiten.

Es war in Gefahr, abgebrochen zu werden. Es war purer Zufall, dass in dieser Phase 1993 das World Wide Web (WWW) explosionsartig bekannt wurde. Bei Sun wurde sofort das Potential von Java erkannt, interaktive („dynamische“) Web-Seiten zu erstellen. So wurde dem Projekt schlagartig neues Leben eingehaucht.

Seinen Durchbruch erlebte Java, als die Firma Netscape Anfang 1995 bekanntgab, dass ihr Browser Navigator 2.0 (damals Marktführer) Java-Programme in Web-Seiten ausführen kann. Formal eingeführt wurde Java auf einer Konferenz von Sun im Mai 1995. Eine beispiellose Erfolgsgeschichte begann. Innerhalb weniger Jahre wurde Java zu einer weitverbreiteten Programmiersprache.

1.1.1 Wie funktioniert Java?

Es gibt im wesentlichen zwei Möglichkeiten, selbst erstellte Programme auf einem Computer auszuführen: Man kann den Quellcode der jeweiligen Programmiersprache *compilieren* oder ihn *interpretieren*. Im ersten Fall wird ein „Übersetzungsprogramm“, der *Compiler*, verwendet, um aus dem Quelltext eine Datei zu erstellen, die von dem jeweiligen Betriebssystem ausgeführt werden kann; unter Windows z.B. haben solche ausführbaren Dateien die Endung `.com` oder `.exe` (für *compiled* oder *executable*). Im zweiten Fall wird der Quelltext von einem *Interpreter* sofort in den Arbeitsspeicher (RAM, *random access memory*) des Computers übersetzt und ausgeführt.

Beispiele für Compilersprachen sind Pascal, C oder C++. Beispiele für Interpretersprachen sind Basic, Perl, PHP oder JavaScript. Ein Vorteil von Compilersprachen ist, dass ein kompiliertes Programm schneller („performanter“) als ein vergleichbares zu interpretierendes Programm abläuft, da die Übersetzung in die Maschinensprache des Betriebssystems ja bereits durchgeführt ist. Ein Interpreter dagegen geht das Programm zeilenweise durch und übersetzt es.

Ein weiterer Vorteil der Compilierung aus der Sicht des Programmierers ist, dass ein Compiler Fehler der *Syntax* erkennt, also Verstöße gegen die Grammatikregeln der Programmiersprache. Man hat also bei einem kompilierten Programm die Gewissheit, dass die Syntax korrekt ist. Bei einer Interpretersprache dagegen läuft bei einem Programmierfehler das Programm einfach nicht, und die Ursache ist nicht näher eingegrenzt. Es können sowohl Syntaxfehler, aber auch logische oder Laufzeitfehler (z.B. Division durch 0) vorhanden sein. Compilersprachen ermöglichen also eine sicherere Entwicklung.

Andererseits haben Compilersprachen gegenüber Interpretersprachen einen großen Nachteil bezüglich der Plattformunabhängigkeit. Für jede Plattform, also jedes Betriebssystem (Windows, Linux, Mac OS), auf der das Programm laufen soll, muss eine eigene Compilerdatei erstellt werden; bei Interpretersprachen kann der Source-Code von dem ausführenden Rechner sofort interpretiert werden, (vorausgesetzt natürlich, der Interpreter ist auf dem Rechner vorhanden).

Welche der beiden Ausführprinzipien verwendet Java? Nun, wie bei jedem vernünftigen Kompromiss — beide! Der Source-Code eines Java-Programms steht in einer Textdatei mit der Endung

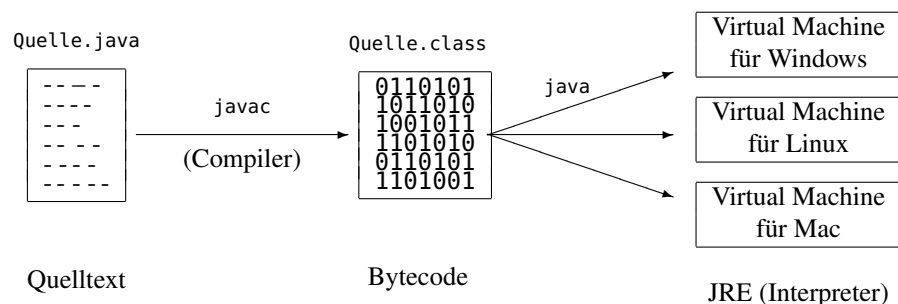


Abbildung 1.2: Vom Quellcode zur Ausführung eines Java-Programms (JRE = Java Runtime Environment)

`.java`. Er wird von einem Compiler namens `javac` in eine Datei umgewandelt, deren Namen nun die Endung `.class` trägt. Sie beinhaltet jedoch nicht, wie sonst bei kompilierten Dateien üblich, ein

lauffähiges Programm, sondern den so genannten *Bytecode*. Dieser Bytecode läuft auf keiner realen Maschine, sondern auf der *virtuellen Maschine (J)VM*. Das ist de facto ein Interpreter, der für jedes Betriebssystem den Bytecode in den RAM des Computers lädt und das Programm ablaufen lässt. Die Virtuelle Maschine wird durch den Befehl `java` aufgerufen und führt den Bytecode aus. Abb. 1.2 zeigt schematisch die einzelnen Schritte vom Quellcode bis zur Programmausführung. Die JVM ist Teil der *Java Runtime Environment (JRE)*, die es für jedes gängige Betriebssystem gibt.

1.1.2 Installation der Java SDK-Umgebung

Ein Java-Programm wird auch *Klasse* genannt. Um Java-Programme zu erstellen, benötigt man die Java 2 SDK-Umgebung für das Betriebssystem, unter dem Sie arbeiten möchten. (SDK = *Software Development Kit*) Es beinhaltet die notwendigen Programme, insbesondere den Compiler `javac`, den Interpreter `java` und die Virtuelle Maschine. Das J2SE ist erhältlich im Internet direkt von SUN unter

<http://java.sun.com>.

(Die Beschreibung der Installation ist z.B. [7] §2.1 oder unter <http://haegar.fh-swf.de> beschrieben.)

Nach erfolgter Installation wird die Datei mit der Eingabe des Zeilenkommandos

```
javac Klassenname.java
```

compiliert. Entsprechend kann sie dann mit der Eingabe

```
java Klassenname
```

ausgeführt werden (ohne die Endung `.class`!).

1.1.3 Erstellung von Java-Programmen

Java-Programme werden als Text in Dateien eingegeben. Hierzu wird ein Texteditor oder eine „integrierte Entwicklungsumgebung“ (*IDE = Integrated Development Environment*) wie Netbeans oder Eclipse,

<http://www.netbeans.org> <http://www.eclipse.org>

benutzt. Die IDE's sind geeignet für die fortgeschrittene Programmierung mit grafischen Bedienungselementen oder größeren Software-Projekten. Wir werden zunächst die einfache Programmerstellung mit einem Texteditor verwenden, um die Prinzipien des Kompilier- und Programmstartprozesses zu verdeutlichen. Später kann man mit etwas komfortableren Editoren wie dem *Java-Editor (javaeditor.org)* für Windows oder dem plattformunabhängigen *jEdit (jedit.org)* arbeiten.

1.1.4 Die 4 Programmarten von Java

Eine ganz neue Eigenart von Java ist es, dass es mehrere Arten von Programmen gibt, die erstellt werden können: Applikationen, Applets, Servlets und Midlets.

- *Applikationen*. Eine Applikation (lat. für „Anwendung“) ist ein eigenständiges Programm, das nur die virtuelle Maschine als Interpreter braucht und „stand-alone“ lauffähig ist — ähnlich wie in anderen Programmiersprachen üblich.
- *Applets*. Ein Applet (engl. für „kleine Applikation“, „Applikationchen“) ist ein Programm, das nur als Unterprogramm eines Webbrowsers läuft. Applets waren bei ihrer Einführung Mitte der 1990er Jahre eine enorme Innovation für das Web, sie ermöglichten Interaktivität und Animationen auf den bis dahin statischen Webseiten. Applets werden als Programme automatisch über das Web verteilt und innerhalb eines Webbrowsers, also „lokal“ ausgeführt. Insbesondere können also rechenintensive Algorithmen auf dem Rechner des Anwenders ablaufen und belasten nicht den Webserver.

- *Servlets*. Ein Servlet ist ein Java-Programm, das ausschließlich als Unterprogramm eines Web-servers läuft. Es ist somit das Gegenstück zu einem Applet, das innerhalb eines Browsers (dem „Web-Client“) läuft. Ein Servlet erweitert die Funktionalität eines Web-servers.

Die möglichen Anwendungen von Servlets sind vielfältig. Sie ermöglichen webbasierte Lösungen für sichere und vertrauliche Zugänge zu einer Website, für Zugriffe auf Datenbanken und für individuelle interaktive Erstellung ganzer HTML-Dokumente.

Eng verwandt mit Servlets sind JSP (Java Server Pages), Java-Programme, die in HTML-Seiten eingebettet sind und bei ihrer Aktivierung durch den Browser Servlets erzeugen.

- *Midlets*. Ein Midlet ist ein Java-Programm, das auf Mobilgeräten wie Handys oder Handhelds ablaufen kann. Das Wort Midlet ist abgeleitet von MID = *mobile information device*. Bei Programmen für Mobilgeräte ist zu beachten, dass solche Endgeräte nur wenig Speicher, eine geringe Performanz, eine kleine Tastatur und nur temporäre Verbindungen zum Netz haben. Midlets sind auf diese Besonderheiten abgestimmt. Sie sind den Applets ähnlich, es steht ihnen jedoch eine weit eingeschränkte Klassenbibliothek zur Verfügung. Eine Entwicklungsumgebung bietet SUN mit der J2ME (Java 2 Micro Edition), die das J2SDK benötigt. Entsprechend ist die Virtuelle Maschine abgespeckt, sie heißt KVM (das „K“ steht für kilo, da der benötigte Speicher im Kilobyte-Bereich liegt, aktuell ≤ 80 KB). Weitere Informationen finden Sie unter:

<http://java.sun.com/j2me/> oder <http://java.sun.com/products/midp/>.

Keine Programmiersprache ermöglicht standardmäßig einen so breiten Funktionsumfang, wie ihn Java mit diesen Programmarten liefert.

1.2 Das erste Programm: Ausgabe eines Textes

Wir beginnen mit einer einfachen *Applikation*, die eine Textzeile ausgibt. Dieses (mit einem beliebigen Texteditor) erstellte Programm ist als die Datei `Willkommen.java` abgespeichert und lautet:

```
1 import javax.swing.JOptionPane;
2 /**
3  * Ausdruck von 2 Zeilen in einem Dialogfenster
4  */
5 public class Willkommen {
6     public static void main(String[] args) {
7         JOptionPane.showMessageDialog(null, "Willkommen zur\nJava Programmierung!");
8     }
9 }
```

Das Programm gibt nach dem Start das Fenster in Abbildung 1.3 auf dem Bildschirm aus.



Abbildung 1.3: Ergebnis der Applikation Willkommen.

1.2.1 Wie kommt die Applikation ans Laufen?

Wie in der Einleitung beschrieben (Abb. 1.2 auf S. 8), muss zunächst aus der `Willkommen.java`-Datei (dem „Source-Code“ oder Quelltext) eine `Willkommen.class`-Datei im Bytecode mit dem Compiler `javac` erstellt werden, und zwar mit der Eingabe des Zeilenkommandos

```
javac Willkommen.java
```

Entsprechend kann sie dann mit der Eingabe

```
java Willkommen
```

ausgeführt werden.

1.2.2 Ein erster Überblick: So funktioniert das Programm

Bevor wir die einzelnen Programmbestandteile genauer betrachten, sollten wir uns die grobe Struktur einer Java-Applikation verdeutlichen.

1. Ein Java-Programm ist stets eine *Klasse* und beginnt mit den Schlüsselworten `public class` sowie dem Namen der Klasse (Zeile 5).
2. Ein Java-Programm ist in Einheiten gepackt, die *Blöcke*, die von geschweiften Klammern { ... } umschlossen sind und oft vorweg mit einer Bezeichnung versehen sind (z.B. `public class Willkommen` oder `public static void main(...)`). Zur Übersichtlichkeit sind die Blöcke stets geeignet einzurücken.
3. Der Startpunkt jeder Applikation ist die Methode `main`. Hier beginnt der Java-Interpreter (die „virtuelle Maschine“, siehe S. 9), die einzelnen Anweisungen auszuführen. Er arbeitet sie „sequenziell“, d.h. der Reihe nach, ab (Zeile 6).
4. Die Applikation besteht aus einer Anweisung (Zeile 7): Sie zeigt einen Text in einem Fenster an.
5. Um Spezialfunktionen zu verwenden, die von anderen Entwicklern erstellt wurden, kann man mit der Anweisung `import` bereits programmierte Klassen importieren. Eine Klasse ist in diesem Zusammenhang also so etwas wie „Werkzeugkasten“, aus dem wir fertig programmierte „Werkzeuge“ (hier die Methode `showMessageDialog` aus `JOptionPane`) verwenden können. Eine der wenigen Klassen, die wir nicht importieren müssen, ist die Klasse `System`.

Halten wir ferner fest, dass die Grobstruktur unserer kleinen Applikation durch das folgende Diagramm dargestellt werden kann.



Es ist ein *Klassendiagramm*. Es stellt grafisch dar, dass unsere Klasse `Willkommen` die Methode `main` hat und die Klasse `JOptionPane` „kennt“, deren Methode `showMessageDialog` sie verwendet.

1.2.3 Die Klassendeklaration und Klassennamen

In der Zeile 5 beginnt die *Klassendeklaration* der Klasse `Willkommen`. Jedes Java-Programm besteht aus mindestens einer Klasse, die definiert ist von dem Programmierer. Die reservierten Worte **public class** eröffnen die Klassendeklaration in Java, direkt gefolgt von dem *Klassennamen*, hier `Willkommen`. *Reservierte Wörter* sind von Java selber belegt und erscheinen stets mit kleinen Buchstaben. Eine Liste aller reservierten Wörter in Java befindet sich in Tabelle 1.1 auf Seite 15 und am Anfang des Indexverzeichnisses des Skripts.

Es gilt allgemein die Konvention, dass alle Klassennamen in Java mit einem Großbuchstaben beginnen und dass jedes neue Wort im Namen ebenfalls mit einem Großbuchstaben beginnt, z.B. `BeispielKlasse` oder `JOptionPane` (*option* = Auswahl, *pane* = Fensterscheibe; das J kennzeichnet alle so genannten „Swing-Klassen“).

Der Klassenname ist ein so genannter *Identifizier* oder *Bezeichner*. Ein Bezeichner ist eine Folge von alphanumerischen Zeichen (*Characters*), also Buchstaben, Ziffern, dem Unterstrich (`_`) und dem Dollarzeichen (`$`). Ein Bezeichner darf nicht mit einer Ziffer beginnen und keine Leerzeichen enthalten. Erlaubt sind also

`Rakete1`, oder `$wert`, oder `_wert`, oder `Taste7`.

Demgegenüber sind `7Taste` oder `erste Klasse` als Klassennamen *nicht* erlaubt. Im übrigen ist Java schreibungssensitiv (*case sensitive*), d.h. es unterscheidet strikt zwischen Groß- und Kleinbuchstaben — `a1` ist also etwas völlig Anderes als `A1`.

Merkregel 1. Eine Klasse in Java muss in einer Datei abgespeichert werden, die genau so heißt wie die Klasse, mit der Erweiterung `„.java“`. Allgemein gilt folgende Konvention: Klassennamen beginnen mit einem Großbuchstaben und bestehen aus einem Wort. Verwenden Sie „sprechende“ Namen, die andeuten, was die Klasse bedeutet. Im Gegensatz dazu werden Methodennamen und Variablen (siehe unten) klein geschrieben.

In unserem ersten Programmbeispiel heißt die Datei `Willkommen.java`.

1.2.4 Der Programmstart: Die Methode `main`

Wie startet eine Applikation in Java? Die zentrale Einheit einer Applikation, gewissermaßen die „Steuerzentrale“, ist die *Methode* `main`. Durch sie wird die Applikation gestartet, durch die dann alle Anweisungen ausgeführt werden, die in ihr programmiert sind. Die Zeile 6,

```
public static void main(String[] args) (1.1)
```

ist Teil jeder Java-Applikation. Java-Applikationen beginnen beim Ablaufen automatisch bei `main`. Die runden Klammern hinter `main` zeigen an, dass `main` eine *Methode* ist. Klassendeklarationen in Java enthalten normalerweise mehrere Methoden. Für Applikationen muss davon *genau eine* `main` heißen und so definiert sein wie in (1.1). Andernfalls wird der Java-Interpreter das Programm nicht ausführen. Nach dem Aufruf der Methode `main` mit dem „Standardargument“ `String[] args` kommt der *Methodenrumpf* (*method body*), eingeschlossen durch geschweifte Klammern (`{ ... }`). Über den tieferen Sinn der langen Zeile `public static void main (String[] args)` sollten Sie sich zunächst nicht den Kopf zerbrechen. Nehmen Sie sie (für's erste) hin wie das Wetter!

Eine allgemeine Applikation in Java muss also in jedem Fall die folgende Konstruktion beinhalten:

```

public class Klassenname {
    public static void main( String[] args ) {
        Deklarationen und Anweisungen;
    }
}

```

Das ist also gewissermaßen die „Minimalversion“ einer Applikation. In unserem Beispiel bewirkt die Methode `main`, dass der Text „Willkommen zur Java-Programmierung“ ausgegeben wird.

1.2.5 Die `import`-Anweisung des `swing`-Pakets

Im Gegensatz zu dem Paket `java.lang` muss eine Klasse aus einem anderen Pakete ausdrücklich mit ihrem Paketpfad angegeben werden. Im allgemeinen muss man das jedes Mal tun, wenn man die Klasse oder eine ihrer Methoden verwenden will. Man kann aber auch ein Paket importieren.





Mit der `import`-Anweisung in Zeile 1 wird die Klasse `JOptionPane` eingelesen und für das Programm benutzbar. `JOptionPane` ist eine Standardklasse von Java aus dem Paket `javax.swing` und stellt *Dialogboxen*, also Fenster zur Ein- und Ausgabe, zur Verfügung. Die Methode `showMessageDialog` der Klasse `JOptionPane` in Zeile 7 öffnet eine Dialogbox namens *message dialog*, die den nach der Konstanten `null` in Anführungszeichen eingeschlossenen Text anzeigt, vgl. Abb. 1.3. Das reservierte Wort `null` ist an dieser Stelle ein Parameter, der im Wesentlichen die Position des Dialogfensters in der Mitte des Bildschirms festlegt. Die Titelzeile des Dialogfensters enthält einfach den String „Message“.

1.2.6 Dialogfenster

Es gibt noch eine weitere Version der Methode `showMessageDialog`, die nicht mit zwei, sondern mit vier Parametern aufgerufen wird und mit denen man die erscheinenden Symbole variieren kann, beispielsweise durch die Anweisung

```
JOptionPane.showMessageDialog(null, "Hallo?", "Frage", JOptionPane.PLAIN_MESSAGE);
```

Der dritte Parameter (hier: „Frage“) bestimmt den Text, der in der Titelleiste des Fensters erscheinen soll. Der vierte Parameter (`JOptionPane.PLAIN_MESSAGE`) ist ein Wert, der die Anzeige des Message-Dialogtyp bestimmt — dieser Dialogtyp hier zeigt kein Icon (Symbol) links von der Nachricht an. Die möglichen Message-Dialogtypen sind in der folgenden Tabelle aufgelistet:

Message-Dialogtyp	int-Wert	Icon	Bedeutung
<code>JOptionPane.ERROR_MESSAGE</code>	0		Fehlermeldung
<code>JOptionPane.INFORMATION_MESSAGE</code>	1		informative Meldung; der User kann sie nur wegklicken
<code>JOptionPane.WARNING_MESSAGE</code>	2		Warnmeldung
<code>JOptionPane.QUESTION_MESSAGE</code>	3		Fragemeldung
<code>JOptionPane.PLAIN_MESSAGE</code>	-1		Meldung ohne Icon

(Statt der langen Konstantennamen kann man auch kurz den entsprechenden `int`-Wert angeben.) Beispielsweise erscheint durch die Anweisung

```
JOptionPane.showMessageDialog(null, "Hallo?", "Frage", JOptionPane.QUESTION_MESSAGE);
```

oder kürzer

```
JOptionPane.showMessageDialog(null, "Hallo?", "Frage", 3);
```

ein Dialogfenster mit einem Fragezeichen als Symbol, dem Text "Hallo?" im Fenster und dem Text "Frage" in der Titelzeile. Das genaue Aussehen der Symbole hängt von dem jeweiligen Betriebssystem ab, unter Windows sehen sie anders als unter Linux oder OS X.

1.3 Elemente eines Java-Programms

1.3.1 Anweisungen und Blöcke

Jedes Programm besteht aus einer Folge von *Anweisungen* (*instruction, statement*), wohldefinierten kleinschrittigen Befehlen, die der Interpreter zur Laufzeit des Programms ausführt. Jede Anweisung wird in Java mit einem Semikolon (;) abgeschlossen.

Wenn man Deklarationen und Anweisungen vergleicht, so stellt man fest, dass Deklarationen die Struktur des Programms festlegen, während die Anweisungen den zeitlichen Ablauf definieren:

Ausdruck	Wirkung	Aspekt
Deklarationen	Speicherreservierung	statisch (Struktur)
Anweisung	Tätigkeit	dynamisch (Zeit)

Anweisungen werden zu einem *Block* (*block, compound statement*) zusammengefasst, der durch geschweifte Klammern ({ und }) umschlossen ist. Innerhalb eines Blockes können sich weitere Blöcke befinden. Zu beachten ist, dass die dabei Klammern stets korrekt geschlossen sind. Die Blöcke eines Programms weisen also stets eine hierarchische logische Baumstruktur aufweisen.

Empfehlung 1. *Zur besseren Lesbarkeit Ihres Programms und zur optischen Strukturierung sollten Sie jede Zeile eines Blocks stets gleich weit **einrücken**. Ferner sollten Sie bei der Öffnung eines Blockes mit { sofort die geschlossene Klammer } eingeben.*

Auf diese Weise erhalten Sie sofort einen blockweise hierarchisch gegliederten Text. So können Sie gegebenenfalls bereits beim Schreiben falsche oder nicht gewollte Klammerungen (Blockbildungen!) erkennen und korrigieren. Falsche Klammerung ist keine kosmetische Unzulänglichkeit, das ist ein echter Programmierfehler.

Es gibt zwei verbreitete Konventionen die öffnende Klammer zu platzieren: Nach der einen Konvention stellt man sie an den Anfang derjenigen Spalte, auf der sich der jeweils übergeordnete Block befindet, und der neue Block erscheint dann eingerückt in der nächsten Zeile, so wie in dem folgenden Beispiel.

```
public class Willkommen
{
    ...
}
```

Die andere Konvention zieht die Klammer zu der jeweiligen Anweisung in der Zeile darüber hoch, also

```
public class Willkommen {
    ...
}
```

In diesem Skript wird vorwiegend die zweite der beiden Konventionen verwendet, obwohl die erste etwas lesbarer sein mag. Geschmackssache eben.

1.3.2 Reservierte Wörter und Literale

Die Anweisungen eines Programms enthalten häufig *reservierte Wörter* oder *Schlüsselwörter*, die Bestandteile der Programmiersprache sind und eine bestimmte vorgegebene Bedeutung haben. Die in Java reservierten Wörter sind in Tabelle 1.1 zusammengefasst.

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

Literale

false	true	null	0, 1, -2, 2.0	"Abc"	
-------	------	------	---------------	-------	--

In Java nicht verwendete reservierte Wörter

byvalue	cast	const	future	generic	goto
inner	operator	outer	rest	var	

Tabelle 1.1: Die reservierten Wörter in Java

Dazu gehören auch sogenannte *Literale*, die Konstanten **true**, **false** und **null**, deren Bedeutung wir noch kennen lernen werden. Allgemein versteht man unter einem *Literal* einen konstanten Wert, beispielsweise eine Zahl wie 14 oder 3.14, ein Zeichen wie **'a'** oder einen String („Text“) wie **"Hallo!"**.

Daneben gibt es in Java reservierte Wörter, die für sich gar keine Bedeutung haben, sondern ausdrücklich (noch) nicht verwendet werden dürfen.

1.3.3 Kommentare

Ein *Kommentar* (*comment*) wird in ein Programm eingefügt, um dieses zu dokumentieren und seine Lesbarkeit zu verbessern. Insbesondere sollen Kommentare es erleichtern, das Programm zu lesen und zu verstehen. Sie bewirken keine Aktion bei der Ausführung des Programms und werden vom Java-Compiler ignoriert. Es gibt drei verschiedene Arten, Kommentare in Java zu erstellen, die sich durch die Sonderzeichen unterscheiden, mit denen sie beginnen und ggf. enden müssen.

- `// ...` Eine Kommentartyp ist der *einzeilige Kommentar*, der mit dem Doppelslash `//` beginnt und mit dem Zeilenende endet. Er wird nicht geschlossen. Quellcode, der in derselben Zeile vor den Backslashes steht, wird jedoch vom Compiler ganz normal verarbeitet.
- `/* ... */` Es können auch mehrere Textzeilen umklammert werden: `/*` und `*/`, z.B.:

```
/* Dies ist ein Kommentar, der  
sich über mehrere Zeilen  
erstreckt. */
```

- `/** ... */` Die dritte Kommentartyp `/** ... */` in den Zeilen 2 bis 4 ist der *Dokumentationskommentar* (*documentation comment*). Sie werden stets direkt vor den Deklarationen von Klassen, Attributen oder Methoden (wir werden noch sehen, was das alles ist...) geschrieben, um sie zu beschreiben. Die so kommentierten Deklarationen werden durch das Programm `javadoc`-Programm automatisch verarbeitet.

In Java werden Leerzeilen, genau wie Leerzeichen und Tab-Stops, vom Compiler nicht verarbeitet. Ein Leerzeichen trennt verschiedene Wörter, jedes weitere Leerzeichen direkt dahinter hat jedoch keine Wirkung. Zusätzliche Leerzeichen können und sollten also verwendet werden zur Strukturierung und Lesbarkeit des Programms.

1.3.4 import-Anweisung

Generell können fertige Java-Programme zur Verfügung gestellt werden. Dazu werden sie in Verzeichnissen, den sogenannten *Paketen* gespeichert. Beispielsweise sind die Klassen des wichtigen Swing-Pakets `javax.swing` in dem Verzeichnis `/javax/swing` (bei UNIX-Systemen) bzw. `\javax\swing` (bei Windows-Systemen) gespeichert.¹ Insbesondere sind alle Programme des Java-API's in Paketen bereitgestellt.

Um nun solche bereitgestellten Programme zu verwenden, müssen sie ausdrücklich importiert werden. Das geschieht mit der `import`-Anweisung. Beispielsweise wird mit

```
import javax.swing.*;
```

Diese Anweisung muss stets am Anfang eines Programms stehen, wenn Programme aus dem entsprechenden Paket verwendet werden sollen. Das einzige Paket, das nicht eigens importiert werden muss, ist das Paket `java.lang`, welches die wichtigsten Basisklassen enthält. Die `import`-Anweisungen sind die einzigen Anweisungen eines Java-Programms, die *außerhalb* eines Blocks stehen.

1.4 Strings und Ausgaben

1.4.1 Strings

Der in die Anführungszeichen (") gesetzte Text ist ein *String*, also eine Kette von beliebigen Zeichen. In Java wird ein String stets durch die doppelten Anführungszeichen eingeschlossen, also

" ... ein Text "

Innerhalb der Anführungszeichen kann ein beliebiges Unicode-Zeichen stehen (alles, was die Tastatur hergibt ...), also insbesondere Leerzeichen, aber auch „Escape-Sequenzen“ wie das Steuerzeichen `\n` für Zeilenumbrüche oder `\"` für die *Ausgabe* von Anführungszeichen (s.u.).

Escape-Sequenzen und Unicode

Zur Darstellung von Steuerzeichen, wie z.B. einen Zeilenumbruch, aber auch von Sonderzeichen aus einer der internationalen Unicode-Tabellen, die Sie nicht auf Ihrer Tastatur haben, gibt es die *Escape-Sequenzen*: Das ist ein Backslash (`\`) gefolgt von einem (ggf. auch mehreren) Zeichen. Es gibt mehrere Escape-Sequenzen in Java, die wichtigsten sind in Tabelle 1.2 aufgelistet. Man kann

Escape-Zeichen	Bedeutung	Beschreibung
<code>\uxxxx</code>	Unicode	das Unicode-Zeichen mit Hexadezimal-Code <code>xxxx</code> („Code-Point“); z.B. <code>\u222B</code> ergibt ∫
<code>\n</code>	line feed LF	neue Zeile. Der Bildschirmcursor springt an den Anfang der nächsten Zeile
<code>\t</code>	horizontal tab HT	führt einen Tabulatorsprung aus
<code>\\</code>	<code>\</code>	Backslash <code>\</code>
<code>\"</code>	<code>"</code>	Anführungszeichen <code>"</code>
<code>\'</code>	<code>'</code>	Hochkomma (Apostroph) <code>'</code>

Tabelle 1.2: Wichtige Escape-Sequenzen in Java

also auch Anführungszeichen ausgeben:

```
JOptionPane.showMessageDialog( null, "\"Zitat Ende\"" );
```

¹Sie werden die Verzeichnisse und Dateien jedoch auf Ihrem Rechner nicht finden: Sie sind in kompakte Dateien komprimiert und haben in der Regel die Endung `.jar` für *Java Archive* oder `.zip` für das gebräuchliche Zip-Archiv.

ergibt **"Zitat Ende"**. Auch mathematische oder internationale Zeichen, die Sie nicht auf Ihrer Tastatur finden, können mit Hilfe einer Escape-Sequenz dargestellt werden. So ergibt beispielsweise

```
JOptionPane.showMessageDialog(null, "2 \u222B x dx = x\u00B2\n\u2115 = \u00D7");
```

die Ausgabe

$$\begin{array}{l} 2 \int x dx = x^2 \\ |\mathbb{N}| = \aleph \end{array}$$

Eine Auflistung der möglichen Unicode-Zeichen und ihre jeweiligen Hexadezimalcodes finden Sie im WWW unter

<http://haegar.fh-swf.de/Applets/Unicode/>

1.4.2 Möglichkeiten der Ausgabe

In unserer Applikation ist der Ausgabetext ein String. Durch die Methode

```
JOptionPane.showMessageDialog(null, "...")
```

kann so ein beliebiger String auf dem Bildschirm angezeigt werden. Ein spezieller String ist der leere String `" "`. Er wird uns noch oft begegnen.

Weitere einfache Möglichkeiten zur Ausgabe von Strings im Befehlszeilenfenster („Konsole“) bieten die Anweisungen `print`, `printf` und `println` des Standard-Ausgabestroms `System.out`. Die Methode `print` gibt den eingegebenen String unverändert (und linksbündig) im Konsolenfenster aus, `printf` erzeugt eine formatierte Ausgabe. `println` steht für *print line*, also etwa „in einer Zeile ausdrucken“, und liefert dasselbe wie `print`, bewirkt jedoch nach der Ausgabe des Strings einen Zeilenumbruch. Beispiele:

```
System.out.print("Hallo, Ihr 2");  
System.out.println("Hallo, Ihr" + 2);  
System.out.printf("Hallo, Ihr %1$d", 2);
```

Der `printf`-Befehl wird hierbei mit einem sogenannten *Format-String*, der eine Art Schablone ist, in die Werte der nachfolgenden Argumente durch einen *Format-Spezifizierer* (*format specifier*) mit der Syntax

%Argument-Index \$ Konversionszeichen

eingesetzt werden. Der Argument-Index zwischen `%` und `$` gibt hierbei die Stelle des einzusetzenden Arguments in der Argumentliste an, beginnend bei `%1$` für das erste Argument. Das Konversionszeichen bestimmt die Formatierung, beispielsweise:

Konversions- zeichen	Formatierte Ausgabe als ...
s	String
d	ganze Zahl (Dezimalzahl)
o	Oktalzahl
x bzw. X	Hexadezimalzahl (mit Klein- bzw. Großbuchstaben)
f	Fließkommazahl (optional mit festen Nachkommastellen: <code>%1\$f.3</code>)
e	Fließkommazahl in wissenschaftlicher Schreibweise (mit Exponent)

Die Syntax des Format-Strings ist für den Anfänger nicht ganz übersichtlich und ist in der Klasse `Formatter`² im Paket `java.util` beschrieben. Eine zusammenfassende Beispielapplikation folgt, die alle drei Ausgabemöglichkeiten ausführt.

²<http://java.sun.com/j2se/1.5.0/docs/api/java/util/Formatter.html#syntax>

```
import javax.swing.*;

public class Ausgaben {
    public static void main(String[] args) {
        // Ausgabe im Konsolenfenster:
        System.out.print("Dies ist eine ");
        System.out.println("Ausgabe mit " + (4+3) + " Wörtern.");

        // formatierte Ausgabe im Konsolenfenster:
        System.out.printf("Dies ist %1$s Ausgabe mit %2$d Wörtern.\n", "eine", 4+3);
        System.out.printf("pi = %1$.3f \n", 3.141592653589793);

        // Ausgabe in eigenem Fenster:
        JOptionPane.showMessageDialog(
            null, "Die ist eine Ausgabe \nmit " + 7 + " Wörtern."
        );
    }
}
```

Die drei print-Befehle geben auf der Konsole allerdings nur die ASCII-Zeichen aus, Umlaute oder Sonderzeichen können nicht ausgegeben werden. Die Ausgabe allgemeiner Unicode-Zeichen gelingt nur mit Hilfe von JOptionPane (bzw. anderen Swing-Klassen).

1.4.3 Dokumentation der API

Eine *API (application programming Interface: „Schnittstelle für die Programmierung von Anwendungsprogrammen“)* ist allgemein eine Bibliothek von Routinen („Methoden“), mit denen man über selbsterstellte Programme auf Funktionen des Betriebssystems oder einer Benutzeroberfläche zugegriffen werden kann. Jede Programmiersprache muss eine API für das jeweilige Betriebssystem bereitstellen, auf dem die erstellten Programme laufen sollen. Man nennt die Erstellung von Software auch *Implementierung* der Schnittstelle.³

Die API von Java stellt dem Programmierer im Wesentlichen Programme (je nach Art „Klassen“ und „Interfaces“ genannt) zur Verfügung, gewissermaßen „Werkzeugkästen“, mit deren Werkzeugen verschiedene Funktionen ermöglicht werden. Das wichtigste dieser „Werkzeuge“ ist die `main`-Methode, die das Ablaufen einer Applikation ermöglicht. Weitere wichtige Klassen sind beispielsweise `Math` für mathematische Funktionen oder `JOptionPane` zur Programmierung von fensterbasierten Dialogen.

Die Java-API besitzt eine detaillierte Dokumentation, die auf dem Webaufttritt von SUN online unter dem Link *API Specifications* zugänglich ist:

<http://docs.oracle.com/javase/6/docs/api/>

In Abbildung 1.4 ist ein Ausschnitt der Dokumentation für die Klasse `Math` gezeigt. Generell besteht eine Dokumentationsseite aus drei Bereichen, links unten sind die Klassen und Interfaces der API aufgelistet, links oben die Pakete (die „Verzeichnisse“ der Klassen und Interfaces). In dem Hauptfenster erscheint nach dem Anklicken einer der Klassen deren Beschreibung.

Man erkennt in dem Hauptfenster den unteren Teil der allgemeinen Beschreibung und die beiden für den Programmierer stets sehr wichtigen Elemente einer Klasse, ihre „Attribute“ (*fields*) und Methoden. Die Attribute der Klasse `Math` beispielsweise sind die beiden Konstanten `E` und `PI`, während die alphabetisch ersten Methoden die Funktionen `abs(a)` den Absolutbetrag des Parameters `a` und `acos(a)` den arccos des Parameters `a` darstellen.

In dem vorliegenden Skript werden generell nur die wichtigsten Elemente einer Klasse oder eines Interfaces beschrieben. Sie sollten daher beim Programmieren stets auch in der API-Dokumentation

³*implementare* – lat. „auffüllen“

JVM™ 2 Platform Standard Ed. 5.0

[All Classes](#)

Packages

- [`java.applet`](#)
- [`java.awt`](#)
- [`java.awt.color`](#)
- [`java.awt.datatransfer`](#)
- [`javax.swing`](#)
- [`management.factory`](#)
- [`ManagementPermission`](#)
- [`ManageReferralControl`](#)
- [`ManagerFactoryParameters`](#)
- [`Manifest`](#)
- [`Map`](#)
- [`Map.Entry`](#)
- [`MappedByteBuffer`](#)
- [`MARSHAL`](#)
- [`MarshalException`](#)
- [`MarshaledObject`](#)
- [`MaskFormatter`](#)
- [`Matcher`](#)
- [`MatchResult`](#)
- [`Math`](#)
- [`MathContext`](#)
- [`MatteBorder`](#)
- [`MBeanAttributeInfo`](#)
- [`MBeanConstructorInfo`](#)
- [`MBeanException`](#)
- [`MBeanFeatureInfo`](#)
- [`MBeanInfo`](#)
- [`MBeanNotificationInfo`](#)
- [`MBeanOperationInfo`](#)
- [`MBeanParameterInfo`](#)
- [`MBeanPermission`](#)
- [`MBeanRegistration`](#)
- [`MBeanRegistrationException`](#)
- [`MBeanServer`](#)

The exact result may be returned. For exact results large in magnitude, one of the endpoints of the bracket may be infinite. Besides accuracy at individual arguments, maintaining proper relations between the method at different arguments is also important.

Therefore, most methods with more than 0.5 ulp errors are required to be *semi-monotonic*: whenever the mathematical function is non-decreasing, so is the floating-point approximation, likewise, whenever the mathematical function is non-increasing, so is the floating-point approximation. Not all approximations that have 1 ulp accuracy will automatically meet the monotonicity requirements.

Since: JDK 1.0

Field Summary	
static double	<u>E</u> The double value that is closer than any other to e, the base of the natural logarithms.
static double	<u>PI</u> The double value that is closer than any other to pi, the ratio of the circumference of a circle to its diameter.

Method Summary	
static double	<u>abs</u> (double a) Returns the absolute value of a double value.
static float	<u>abs</u> (float a) Returns the absolute value of a float value.
static int	<u>abs</u> (int a) Returns the absolute value of an int value.
static long	<u>abs</u> (long a) Returns the absolute value of a long value.
static double	<u>acos</u> (double a) Returns the arc cosine of an angle, in the range of 0.0 through pi.
static double	<u>asin</u> (double a) Returns the arc sine of an angle, in the range of -pi/2 through pi/2.
static double	<u>atan</u> (double a) Returns the arc tangent of an angle, in the range of -pi/2 through pi/2.

Abbildung 1.4: Ausschnitt aus der API-Dokumentation für die Klasse `Math`.

nachschlagen. Sie sollten sich sogar angewöhnen, beim Entwickeln von Java-Software mindestens zwei Fenster geöffnet haben, den Editor und die Dokumentation im Browser.

Mathematische Funktionen und Konstanten

Zusammengefasst stellt die Klasse `Math` also wesentliche Konstanten und mathematische Funktionen und Operationen für die Programmierung in Java bereit. Sie liefert die Konstanten

$$\begin{array}{ll} \text{Math.E} & // \text{ Euler'sche Zahl } e = \sum_{k=0}^{\infty} \frac{1}{k!} \approx 2,718281828\dots \\ \text{Math.PI} & // \text{ Kreiszahl } \pi = \sum_{k=0}^{\infty} \frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right) \approx 3,14159\dots \end{array}$$

und u.a. die mathematischen Funktionen

```
Math.abs(x)      //  $|x|$ 
Math.acos(x)     //  $\arccos x$ 
Math.asin(x)     //  $\arcsin x$ 
Math.atan(x)     //  $\arctan x$ 
Math.atan2(x,y)  //  $\arctan(x/y)$ 
Math.cos(x)      //  $\cos x$ 
Math.exp(x)      //  $e^x$  (e hoch x)
Math.log(x)      //  $\ln x$  (natürlicher Logarithmus)
Math.max(x,y)    // Maximum von x und y
Math.min(x,y)    // Minimum von x und y
Math.pow(x,y)    //  $x^y$  (x hoch y)
Math.random()    // Pseudozufallszahl z mit  $0 \leq z < 1$ 
Math.round(x)    // kaufmännische Rundung von x auf die nächste ganze Zahl
Math.sin(x)      //  $\sin x$ 
Math.sqrt(x)     //  $\sqrt{x}$ 
Math.tan(x)      //  $\tan x$ 
```

1.5 Variablen, Datentypen und Operatoren

Zu Beginn des letzten Jahrhunderts gehörte das Rechnen mit Zahlen zu denjenigen Fähigkeiten, die ausschließlich intelligenten Wesen zugeschrieben wurden. Heute weiß jedes Kind, dass ein Computer viel schneller und verlässlicher rechnen kann als ein Mensch. Als „intelligent“ werden heute gerade diejenigen Tätigkeiten angesehen, die früher eher als trivial galten: Gesichter und Muster erkennen, Assoziationen bilden, sozial denken. Zukünftig werden Computer sicherlich auch diese Fähigkeiten erlernen, zumindest teilweise. Wir werden uns aber in diesem Skript auf die grundlegenden Operationen beschränken, die früher als intelligent galten und heute als Basis für alles Weitere dienen. Fangen wir mit den Grundrechenarten an.

Wie jede Programmiersprache besitzt auch Java die Grundrechenoperationen. Sie bestehen wie die meisten Operationen aus einem „Operator“ und zwei „Operanden“. Z.B. besteht die Operation $2+3$ aus dem Operator $+$ und den Operanden 2 und 3. Allgemein kann man eine Operation wie folgt definieren.

Definition 1.1. Eine Operation ist gegeben durch einen Operator und mindestens einen Operanden. Hierbei ist ein *Operator* ein Zeichen, das die Operation symbolisiert, meist eine zweistellige Verknüpfung, und die Argumente dieser Verknüpfung heißen *Operanden*. \square

Diese Definition einer Operation erscheint trivial, sie hat es jedoch in sich: Wendet man denselben Operator auf verschiedene Datentypen an, so kommen möglicherweise verschiedene Ergebnisse heraus. Wir werden das in der nächsten Applikation sehen.

```
1 import javax.swing.JOptionPane;
2 /**
3  * Führt verschiedene arithmetische Operationen durch
4  */
5 public class Arithmetik {
6     public static void main( String[] args ) {
7         int m, n, k;    // ganze Zahlen
8         double x, y, z; // 'reelle' Zahlen
9         String ausgabe = "";
10        // diverse arithmetische Operationen:
11        k = - 2 + 6 * 4;
12        x = 14 / 4;  y = 14 / 4.0;  z = (double) 14 / 4;
13        n = 14 / 4;  m = 14 % 4;
14        // Ausgabestring bestimmen:
15        ausgabe = "k = " + k;
16        ausgabe = ausgabe + "\nx = " + x + ", y = " + y + ", z = " + z;
17        ausgabe = ausgabe + "\nn = " + n + ", m = " + m;
18        JOptionPane.showMessageDialog(
19            null, ausgabe, "Ergebnis", JOptionPane.PLAIN_MESSAGE
20        );
21    }
22 }
```

Das Programm führt einige arithmetische Operationen aus. Auf den ersten Blick etwas überraschend ist die Ausgabe, siehe Abb. 1.5: Je nach Datentyp der Operanden ergibt der (scheinbar!) gleiche

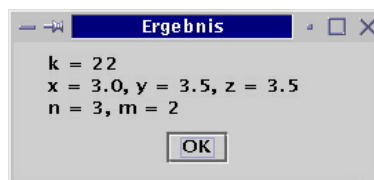


Abbildung 1.5: Die Applikation Arithmetik.

Rechenoperator verschiedene Ergebnisse.

1.5.1 Variablen, Datentypen und Deklarationen

Die Zeile 7 der obigen Applikation,

```
int m, n, k;
```

ist eine *Deklaration*. Der Bezeichner *k* ist der Name einer *Variablen*. Eine Variable ist ein Platzhalter, der eine bestimmte Stelle im Arbeitsspeicher (RAM) des Computers während der Laufzeit des Programms reserviert, siehe Abb. 1.6. Dort kann dann ein *Wert* gespeichert werden. Es ist nicht ein

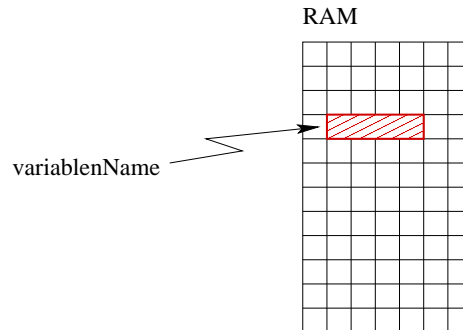


Abbildung 1.6: Die Deklaration der Variable *variablenName* bewirkt, dass im Arbeitsspeicher (RAM) eine bestimmte Anzahl von Speicherzellen reserviert wird. Die Größe des reservierten Speichers (= Anzahl der Speicherzellen) ist durch den Datentyp bestimmt. Vgl. Tabelle 1.3 (S. 22)

beliebig großer Wert möglich, denn dafür wäre ein unbegrenzter Speicherplatz notwendig: Die Menge der möglichen Werte wird durch den *Datentyp* festgelegt. In unserem Beispiel ist dieser Datentyp *int*, und der Wertebereich erstreckt sich von -2^{31} bis $2^{31} - 1$.

Deklarationen werden mit einem Semikolon (;) beendet und können über mehrere Zeilen aufgesplittet werden. Man kann mehrere Variablen des gleichen Typs in einer einzigen Deklaration durch Kommas (,) getrennt deklarieren, oder jede einzeln. Man könnte also in unserem Beispiel genauso schreiben:

```
int m;  
int n;  
int k;
```

Ein Variablenname kann ein beliebiger gültiger Bezeichner sein, also eine beliebige Zeichenfolge, die nicht mit einer Ziffer beginnt und keine Leerzeichen enthält. Es ist üblich, Variablennamen (genau wie Methodennamen) mit einem kleinen Buchstaben zu beginnen.

Variablen müssen stets mit ihrem Namen und ihrem *Datentyp* deklariert werden. In Zeile 7 besagt die Deklaration, dass die Variablen vom Typ *int* sind.

1.5.2 Der Zuweisungsoperator =

Eine der grundlegendsten Anweisungen ist die Wertzuweisung. In Java wird die Zuweisung mit dem so genannten *Zuweisungsoperator* (*assign operator*) = bewirkt. In Zeile 11,

```
k = - 2 + 6 * 4;
```

bekommt die Variable *k* den Wert, der sich aus dem Term auf der rechten Seite ergibt, also 22. (Java rechnet Punkt vor Strichrechnung!) Der Zuweisungsoperator ist gemäß Def. 1.1 ein Operator mit zwei Operanden.

Merkregel 2. Der Zuweisungsoperator = weist der Variablen auf seiner linken Seite den Wert des Ausdrucks auf seiner rechten Seite zu. Der Zuweisungsoperator = ist ein binärer Operator, er hat zwei Operanden (die linke und die rechte Seite).

Alle Operationen, also insbesondere Rechenoperationen, müssen stets auf der rechten Seite des Zuweisungsoperators stehen. Ein Ausdruck der Form $k = 20 + 2$ ist also sinnvoll, **aber eine Anweisung** $20 + 2 = k$ **hat in Java keinen Sinn!**

Datentypen und Speicherkonzepte

Variablenamen wie m oder n in der Arithmetik-Applikation beziehen sich auf bestimmte Stellen im Arbeitsspeicher des Computers. Jede Variable hat einen *Namen*, einen *Typ*, eine *Größe* und einen *Wert*. Bei der Deklaration am Beginn einer Klassendeklaration werden bestimmte Speicherzellen für die jeweilige Variable reserviert. Der Name der Variable im Programmcode verweist während der Laufzeit auf die Adressen dieser Speicherzellen.

Woher weiß aber die CPU, wieviel Speicherplatz sie reservieren muss? Die acht sogenannten *primitiven Datentypen* bekommen in Java den in Tabelle 1.3 angegebenen Speichergrößen zugewiesen.

Datentyp	Größe	Werte	Bemerkungen
boolean	1 Byte	true, false	
char	2 Byte	'\u0000' bis '\uFFFF'	Unicode, 2^{16} Zeichen, stets in Apostrophs (!)
byte	1 Byte	-128 bis +127	$-2^7, -2^7 + 1, \dots, 2^7 - 1$
short	2 Byte	-32 768 bis +32 767	$-2^{15}, -2^{15} + 1, \dots, 2^{15} - 1$
int	4 Byte	-2 147 483 648 bis +2 147 483 647	$-2^{31}, -2^{31} + 1, \dots, 2^{31} - 1$ Beispiele: 123, -23, 0x1A, 0b110
long	8 Byte	-9 223 372 036 854 775 808 bis +9 223 372 036 854 775 807	$-2^{63}, -2^{63} + 1, \dots, 2^{63} - 1$ Beispiele: 123L, -23L, 0x1AL, 0b110L
float	4 Byte	$\pm 1.4\text{E-}45$ bis $\pm 3.4028235\text{E}+38$	$[\approx \pm 2^{-149}, \approx \pm 2^{128}]$ Beispiele: 1.0f, 1.F, .05F, 3.14E-5F
double	8 Byte	$\pm 4.9\text{E-}324$ bis $\pm 1.7976931348623157\text{E}+308$	$[\pm 2^{-1074}, \approx \pm 2^{1024}]$ Beispiele: 1.0, 1., .05, 3.14E-5

Tabelle 1.3: Die primitiven Datentypen in Java

Wird nun einer Variable ein Wert zugewiesen, z.B. durch die Anweisung

```
k = - 2 + 6 * 4;
```

der Variablen k der berechnete Wert auf der rechten Seite des Zuweisungsoperators, so wird er in den für k reservierten Speicherzellen gespeichert. Insbesondere wird ein Wert, der eventuell vorher dort gespeichert war, überschrieben. Umgekehrt wird der Wert der Variablen k in der Anweisung

```
ausgabe = "k = " + k;
```

der Zeile 15 nur aus dem Speicher gelesen, aber nicht verändert. Der Zuweisungsoperator zerstört nur den Wert einer Variablen auf seiner *linken* Seite und ersetzt ihn durch den Wert der Operationen auf seiner *rechten* Seite.

Eine Variable muss bei ihrem ersten Erscheinen in dem Ablauf eines Programms auf der linken Seite des Zuweisungsoperators stehen. Man sagt, sie muss *initialisiert* sein. Hat nämlich eine Variable keinen Wert und steht sie auf der rechten Seite des Zuweisungsoperators, so wird ein leerer Speicherplatz verwendet. In Java ist die Verarbeitung einer nichtinitialisierten Variablen ein Kompilierfehler. Würde man die Anweisung in Zeile 11 einfach auskommentieren, so würde die folgende Fehlermeldung beim Versuch der Kompilierung erscheinen:


```

/Development/Arithmetik.java:11: variable k might not have been initialized
ausgabe = "k = " + k;
                ^

```

1 error

Sie bedeutet, dass in Zeile 11 des Programms dieser Fehler auftritt.

Merkregel 3. *Eine Variable muss initialisiert werden. Zuweisungen von nichtinitialisierten Werten ergeben einen Kompilierfehler. Eine Variable wird initialisiert, indem sie ihre erste Anweisung eine Wertzuweisung ist, bei der sie auf der linken Seite steht.*

In der Applikation Arithmetik wird beispielsweise die Variable `ausgabe` direkt bei ihrer Deklaration initialisiert, hier mit dem leeren String:

```
String ausgabe = "";
```

1.5.3 Datenkonvertierung mit dem *Cast-Operator*

Man kann in einem laufenden Programm einen Wert eines gegebenen Datentyps in einen anderen explizit konvertieren, und zwar mit dem *Cast-Operator* `()`: Durch

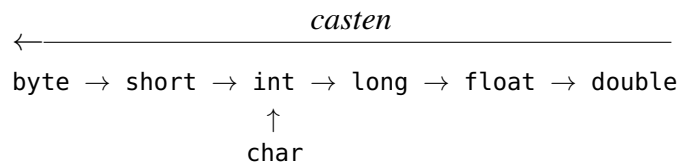
$$(\langle \text{Datentyp} \rangle) \text{ Ausdruck}$$

wird der Wert des Ausdrucks in den Typ umgewandelt, der in den Klammern davor eingeschlossen ist. Beispielsweise castet man die Division der beiden `int`-Werte 14 und 3 in eine Division von `double`-Werten, indem man schreibt `(double) 14 / 3`. In diesem Falle ist die Wirkung exakt wie `14.0 / 3`, man kann jedoch mit dem Cast-Operator auch Werte von Variablen konvertieren. So könnte man durch die beiden Zeilen

```
double x = 3.1415;
int n = (int) x;
```

erreichen, dass die Variable `n` den `int`-Wert 3 erhält. Der Cast-Operator kann nicht nur in primitive Datentypen konvertieren, sondern auch zwischen komplexen Datentypen („Objekten“, wie wir später noch kennen lernen werden).

Der Cast-Operator bewirkt eine sogenannte *explizite Typumwandlung*, sie ist zu unterscheiden von der *impliziten* Typumwandlung, die „automatisch“ geschieht.



Ein solcher impliziter Cast geschieht zum Beispiel bei den Anweisungen `double x = 1;` (`int` \rightarrow `double`) oder `int z = 'z';` (`char` \rightarrow `int`).

1.5.4 Operatoren, Operanden, Präzedenz

Tauchen in einer Anweisung mehrere Operatoren auf, so werden sie nach einer durch ihre *Präzedenz* (oder ihrer *Wertigkeit*) festgelegten Reihenfolge ausgeführt. Eine solche Präzedenz ist z.B. „Punkt vor Strich“, wie bei den Integer-Operationen in Zeile 14: Hier ist das Ergebnis wie erwartet 22. In Tabelle 1.4 sind die arithmetischen Operatoren und ihre Präzedenz aufgelistet.

Arithmetische Ausdrücke in Java müssen in einzeiliger Form geschrieben werden. D.h., Ausdrücke wie „a geteilt durch b“ müssen in Java als `a / b` geschrieben werden, so dass alle Konstanten, Variablen und Operatoren in einer Zeile sind. Eine Schreibweise wie $\frac{a}{b}$ ist nicht möglich. Dafür werden Klammern genau wie in algebraischen Termen zur Änderung der Reihenfolge benutzt, so wie in `a * (b + c)`.

Operator	Operation	Präzedenz
()	Klammern	werden zuerst ausgewertet. Sind Klammern von Klammern umschlossen, werden sie von innen nach außen ausgewertet.
*, /, %	Multiplikation, Division, Modulus	werden als zweites ausgewertet
+, -	Addition, Subtraktion	werden zuletzt ausgewertet

Tabelle 1.4: Die Präzedenz der arithmetischen Operatoren in Java. Mehrere Operatoren gleicher Präzedenz werden stets von links nach rechts ausgewertet.

Operatoren hängen ab von den Datentypen ihrer Operanden

In Anmerkung (3) sind die Werte für y und z auch wie erwartet, 3.5. Aber $x = 3$ — Was geht hier vor? Die Antwort wird angedeutet durch die Operationen in Anmerkung (3): Der Operator `/` ist mit zwei Integer-Zahlen als Operanden nämlich eine *Integer-Division* oder eine *ganzzahlige Division* und gibt als Wert eine Integer-Zahl zurück. Und zwar ist das Ergebnis genau die Anzahl, wie oft der Nenner ganzzahlig im Zähler aufgeht, also ergibt $14 / 4$ genau 3. Den Rest der ganzzahligen Division erhält man mit der modulo-Operation `%`. Da mathematisch

$$14/4 = 3 \text{ Rest } 2,$$

ist $14 \% 4$ eben 2.

Um nun die uns geläufige Division reeller Zahlen zu erreichen, muss mindestens einer der Operanden eine **double**- oder **float**-Zahl sein. Die Zahl 14 wird zunächst stets als **int**-Zahl interpretiert. Um sie als **double**-Zahl zu markieren, gibt es mehrere Möglichkeiten:

$$\text{double } s = 14.0, \quad t = 14., \quad u = 14d, \quad v = 1.4e1; \quad (1.2)$$

String als Datentyp und Stringaddition

Wie die dritte der Deklarationen in der Arithmetik-Applikation zeigt, ist ein String, neben den primitiven Datentypen aus Tab. 1.3, ein weiterer Datentyp. Dieser Datentyp ermöglicht die Darstellung eines allgemeinen Textes.

In der Anweisung in Anmerkung (6)

$$\text{ausgabe} = \text{"k = "} + k; \quad (1.3)$$

wird auf der rechten Seite eine neue (!) Operation „+“ definiert, die *Stringaddition* oder *Konkatenation* (Aneinanderreihung). Diese Pluszeichen kann nämlich gar nicht die übliche Addition von Zahlen sein, da der erste der beiden Operanden `"k = "` ein String ist! Die Konkatenation zweier Strings **"text1"** und **"text2"** bewirkt, dass ihre Werte einfach aneinander gehängt werden,

$$\text{"text1"} + \text{"text2"} \mapsto \text{"text1text2"} \quad (1.4)$$

Hier ist die Reihenfolge der Operanden natürlich entscheidend (im Gegensatz zur Addition von Zahlen). Die Stringaddition **"2" + "3"** ein anderes Ergebnis liefert als die Addition $2+3$:

$$\text{"2"} + \text{"3"} \mapsto \text{"23"}, \quad 2 + 3 \mapsto 5. \quad (1.5)$$

Auch hier, ähnlich wie bei der unterschiedlichen Division von **int** und **double**-Werten, haben zwei verschiedene Operatoren dasselbe Symbol. Das steckt hinter der Definition 1.1.

Eine Eigentümlichkeit ist aber noch nicht geklärt: Was wird denn für k eingesetzt? Das ist doch ein `int`-Wert und kein String. Die Antwort ist, dass der zur Laufzeit in dem Speicherbereich für k stehende Wert (also 22) automatisch in einen String umgeformt wird,

$$\text{"k" = " + 22} \mapsto \text{"k" = " + "22"} \mapsto \text{"k = 22"} \quad (1.6)$$

Nach Ausführung der Anweisung (6) steht also in der Variablen `ausgabe` der String `"k = 22"`.

Mit unserem bisherigen Wissen über den Zuweisungsoperator gibt nun der mathematisch gelesenen natürlich völlig unsinnige Ausdruck bei Anmerkung (6),

```
ausgabe = ausgabe + "\nx = " + x ... ;
```

einen Sinn: An den alten Wert von `ausgabe` wird der String `"..."` angehängt und als neuer Wert in den Speicherplatz für `ausgabe` gespeichert.

1.5.5 Spezielle arithmetische Operatoren

Es gibt eine ganze Reihe von Operatoren, die häufig verwendete Anweisungen verkürzt darstellen. Wir geben Sie hier tabellarisch an.

Zuweisungsoperator	Beispiel	Bedeutung	Wert für c, wenn <code>int c=11, x=4</code>
<code>+=</code>	<code>c += x;</code>	<code>c = c + x;</code>	15
<code>-=</code>	<code>c -= x;</code>	<code>c = c - x;</code>	7
<code>*=</code>	<code>c *= x;</code>	<code>c = c * x;</code>	44
<code>/=</code>	<code>c /= x;</code>	<code>c = c / x;</code>	2
<code>%=</code>	<code>c %= x;</code>	<code>c = c % x;</code>	3

So bewirkt als die Anweisung `c += 4;`, dass der beim Ausführen der Anweisung aktuelle Wert von c, beispielsweise 11, um 4 erhöht wird und den alten Wert überschreibt.

Daneben gibt es in Java den *Inkrementoperator* `++` und den *Dekrementoperator* `--`. Wird eine Variable c um 1 erhöht, so kann man den Inkrementoperator `c++` oder `++c` anstatt der Ausdrücke `c = c+1` oder `c += 1` verwenden, wie aus der folgenden Tabelle ersichtlich.

Operator	Bezeichnung	Beispiel	Bedeutung
<code>++</code>	präinkrement	<code>++c</code>	erhöht erst c um 1, und führt dann den Rest der Anweisung mit dem neuen Wert aus
<code>++</code>	postinkrement	<code>c++</code>	führt erst die gesamte Anweisung aus und erhöht danach den Wert von c um 1
<code>--</code>	prädekrement	<code>--c</code>	erniedrigt erst c um 1, und führt dann den Rest der Anweisung mit dem neuen Wert aus
<code>--</code>	postdekrement	<code>c--</code>	führt erst die gesamte Anweisung aus und erniedrigt erst dann den Wert von c um 1

Besteht insbesondere eine Anweisung nur aus einem dieser Operatoren, so haben Prä- und Postoperatoren dieselbe Wirkung, also: `c++;` \iff `++c;` und `c--;` \iff `--c;`. Verwenden Sie also diese Operatoren nur dann in einer längeren Anweisung, wenn Sie das folgende Programmbeispiel sicher verstanden haben!

Da die Operatoren `++` und `--` nur einen Operanden haben, heißen sie „unär“ (*unary*).

Überlegen Sie, was in dem folgenden Beispielprogramm als `ausgabe` ausgegeben wird.

```
import javax.swing.JOptionPane;
/** Unterschied präinkrement - postinkrement. */
public class Inkrement {
    public static void main( String[] args ) {
        int c;
        String ausgabe = "";
```

```

// postinkrement
c = 5;
ausgabe += c + ", ";
ausgabe += c++ + ", ";
ausgabe += c + ", ";
// präinkrement
c = 5;
ausgabe += c + ", ";
ausgabe += ++c + ", ";
ausgabe += c;
// Ausgabe des Ergebnisses:
JOptionPane.showMessageDialog(
    null, ausgabe, "prä- und postinkrement", JOptionPane.PLAIN_MESSAGE
);
}
}

```

(Antwort: "5, 5, 6, 5, 6, 6")

1.6 Eingaben

Bisher haben wir Programme betrachtet, in denen mehr oder weniger komplizierte Berechnungen durchgeführt wurden und die das Ergebnis ausgaben. Der Anwender konnte während des Ablaufs dieser Programme die Berechnung in keiner Weise beeinflussen. Eine wesentliche Eigenschaft benutzerfreundlicher Programme ist es jedoch, durch Eingaben während der Laufzeit ihren Ablauf zu bestimmen. In der Regel wird der Benutzer durch Dialogfenster aufgefordert, die Notwendigen Daten einzugeben. Man nennt diese Art von Kommunikation auch *Interaktion*.

1.6.1 Strings addieren

Unsere nächste Java-Applikation wird zwei Eingabestrings über die Tastatur einlesen und sie *konkateniert* (= aneinandergehängt) ausgeben.

```

import javax.swing.*; // (1)

/**
 * Addiert 2 einzugebende Strings
 */
public class StringAddition {
    public static void main( String[] args ) {
        // Eingabefelder aufbauen:
        JTextField[] feld = {new JTextField(), new JTextField()}; // (2)
        Object[] msg = {"Erster Text:", feld[0], "Zweiter Text:", feld[1]};
        // Dialogfenster anzeigen:
        int click = JOptionPane.showConfirmDialog(null, msg, "Eingabe", 2);

        // Konkatenation der eingegebenen Texte:
        String ausgabe = feld[0].getText() + feld[1].getText(); // (3)

        // Ausgabe des konkatenierten Texts:
        JOptionPane.showMessageDialog(
            null, ausgabe, "Ergebnis", JOptionPane.PLAIN_MESSAGE // (4)
        );
    }
}

```

Das Programm gibt nach dem Start das Fenster in Abbildung 1.7 auf dem Bildschirm aus.

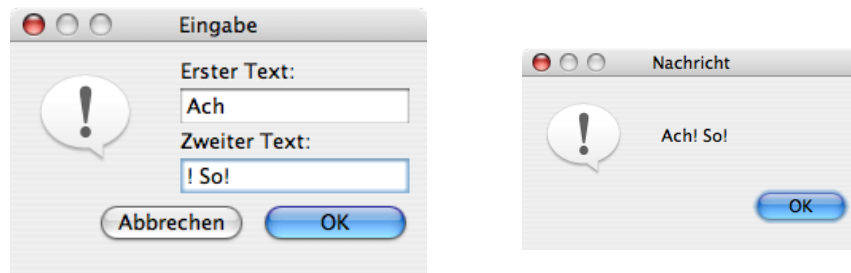


Abbildung 1.7: Die Applikation StringAddition. Links der Eingabedialog, rechts die Ausgabe.

Wir bemerken zunächst, dass diesmal mit `import javax.swing.*` das gesamte Swing-Paket importiert wird, insbesondere also die Klasse `JOptionPane`. Ferner wird die Klasse `StringAddition` deklariert, der Dateiname für die Quelldatei ist also `StringAddition.java`.

1.6.2 Eingabedialoge

Eingaben über Dialogfenster zu programmieren, insbesondere mit mehreren Eingabefeldern, ist im Allgemeinen eine nicht ganz einfache Aufgabe. In Java geht es relativ kurz, man benötigt allerdings drei Anweisungen (Anmerkung (2)), den „Eingabeblock“:

```
// Eingabefelder aufbauen:
JTextField[] feld = {new JTextField(), new JTextField()};
Object[] msg = {"Erster Text:", feld[0], "Zweiter Text:", feld[1]};
// Dialogfenster anzeigen:
int click = JOptionPane.showConfirmDialog(null, msg, "Eingabe", 2);
```

(1.7)

Wir werden diese drei Anweisungen hier nur insoweit analysieren, um sie für unsere Zwecke als flexibles Dialogfenster einsetzen zu können. Um genau zu verstehen, wie sie funktionieren, fehlt uns zur Zeit noch das Hintergrundwissen. Die ersten zwei Anweisungen bauen die Eingabefelder und die dazugehörigen Texte auf. Beides sind Deklarationen mit einer direkten Wertzuweisung. Die erste Variable ist `feld`, und es ist ein so genanntes *Array*, eine durchnummerierte Liste vom Datentyp `JTextField[]`. Dieser Datentyp ist allerdings nicht primitiv, sondern eine Klasse, die im Paket `javax.swing` bereit gestellt wird. Das Array wird in diesem Programm mit zwei Textfeldern gefüllt, jeweils mit dem Befehl `new JTextField()`. Das erste Textfeld in diesem Array hat nun automatisch die Nummer 0, und man kann mit `feld[0]` darauf zugreifen, auf das zweite entsprechend mit `feld[1]`.

In der zweiten Anweisung wird ein Array vom Typ `Object[]` deklariert, wieder mit einer direkten Wertzuweisung. Hier werden abwechselnd Strings und Textfelder aneinander gereiht, ihr Zusammenhang mit dem später angezeigten Dialogfenster ist in Abb. 1.8 dargestellt.

```
JTextField[] feld = {new JTextField(), new JTextField()};
Object[] msg = {"Erster Text:", feld[0], "Zweiter Text:", feld[1]};
```

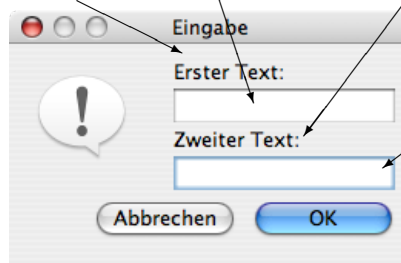


Abbildung 1.8: Das durch den „Eingabeblock“ erzeugte Dialogfenster

Mit der dritten Anweisungszeile schließlich wird das Dialogfenster am Bildschirm angezeigt:

```
int click = JOptionPane.showConfirmDialog(null, msg, "Eingabe", 2);
```

Die Wirkung dieser Anweisung ist interessanterweise, dass sie den weiteren Ablauf des Programms solange anhält, bis der Anwender mit der Maus auf oder geklickt hat. Alle weiteren Anweisungen der Applikation, also ab Anweisung (3), werden also erst ausgeführt, wenn der Anwender den Dialog beendet hat. Man spricht daher von einem „modalen Dialog“. Danach kann man mit der Anweisung

```
feld[n].getText()
```

auf den vom Anwender in das $(n + 1)$ -te Textfeld eingetragenen Text zugreifen. Der Text ist in Java natürlich ein String.

Ob der Anwender nun oder gedrückt hat, wird in der Variablen click gespeichert, bei hat sie den Wert 0, bei den Wert 2:

$$\text{OK} \Rightarrow \text{click} = 0, \quad \text{Abbrechen} \Rightarrow \text{click} = 2. \quad (1.8)$$

Zunächst benötigen wir diese Information noch nicht, in unserem obigen Programm ist es egal, welche Taste gedrückt wird. Aber diese Information wird wichtig für Programme, deren Ablauf durch die beiden Buttons gesteuert wird.

Konkatenation von Strings

In Anmerkung (3),

```
String ausgabe = feld[0].getText() + feld[1].getText();
```

wird die Variable ausgabe deklariert und ihr direkt ein Wert zugewiesen, nämlich die „Summe“ aus den beiden vom Anwender eingegebenen Texten, in Java also Strings. Natürlich kann man Strings nicht addieren wie Zahlen, eine Stringaddition ist nichts anderes als eine Aneinanderreihung. Man nenn sie auch *Konkatenation*. So ergibt die Konkatenation der Strings "Ach" und "! So!" z.B.

```
"Ach" + "! So!"  ⇨  "Ach! So!".
```

1.6.3 Zahlen addieren

Unsere nächste Java-Applikation wird zwei Integer-Zahlen über die Tastatur einlesen und die Summe ausgeben.

```
import javax.swing.*;

/**
 * Addiert 2 einzugebende Integer-Zahlen.
 */
public class Addition {
    public static void main( String[] args ) {
        int zahl1, zahl2,          // zu addierende Zahlen          // (1)
            summe;                 // Summe von zahl1 und zahl2
        String ausgabe;
        // Eingabefelder aufbauen:
        JTextField[] feld = {new JTextField(), new JTextField()};
        Object[] msg = {"Erste Zahl:", feld[0], "Zweite Zahl:", feld[1]};
        // Dialogfenster anzeigen:
        int click = JOptionPane.showConfirmDialog(null, msg, "Eingabe", 2);
```

```

// Konvertierung der Eingabe von String nach int:
zahl1 = Integer.parseInt( feld[0].getText() );           // (2)
zahl2 = Integer.parseInt( feld[1].getText() );

// Addition der beiden Zahlen:
summe = zahl1 + zahl2;                                   // (3)

ausgabe = "Die Summe ist " + summe;                     // (4)
JOptionPane.showMessageDialog(
    null, ausgabe, "Ergebnis", JOptionPane.PLAIN_MESSAGE
);
}
}

```

1.6.4 Konvertierung von Strings in Zahlen

In großen Teilen ist dieses Programm identisch mit unserem Programm zur Stringaddition (Abb. 1.7). bis auf die Deklaration der **int**-Variablen. Das Einlesen der Werte geschieht hier genau wie im zweiten Programm, nur dass der Informationstext in der Dialogbox jetzt etwas anders lautet: Der Anwender wird aufgefordert, ganze Zahlen einzugeben.

Nun passiert bei der Eingabe aber Folgendes: Der Anwender gibt ganze Zahlen ein, die Textfelder `feld[0]` und `feld[1]` aber, denen der Wert übergeben wird, liefern mit `feld[n].getText()` einen Wert vom Typ `String`. Wie passt das zusammen?

Kurz gesagt liegt das daran, dass über Textfelder genau genommen nur `Strings` eingelesen werden können. Schließlich ist *jede* Eingabe von der Tastatur ein `String`, d.h. auch wenn jemand 5 oder 3.1415 eintippt, so empfängt das Programm immer einen `String`. Der feine aber eminent wichtige Unterschied zwischen 3.1415 als `String` und 3,1415 als `Zahl` ist vergleichbar mit dem zwischen einer Ziffernfolge und einer `Zahl`. Als `Zahlen` ergibt die Summe $5 + 3,1415$ natürlich 8,1415 — als `Strings` jedoch gilt

"5" + "3.1415" = "53.1415"!

`Zahlen` werden durch `+` arithmetisch addiert (was sonst?), aber `Strings` werden konkateniert.

Unser Programm soll nun jedoch `Integer`-`Zahlen` addieren, empfängt aber über die Tastatur nur `Strings`. Was ist zu tun? Die Eingabestrings müssen *konvertiert* werden, d.h. ihr `String`wert muss in den `Integer`wert umgewandelt werden. Java stellt für diese Aufgabe eine Methode bereit, `Integer.parseInt`, eine Methode der Klasse `Integer` aus dem Paket `java.lang`. In der Zeile der Anmerkung (2),

```
zahl1 = Integer.parseInt( feld[0].getText() );
```

gibt die Methode `parseInt` den nach **int** konvertierten `String` aus dem Textfeld als **int**-Wert an das Programm zurück, der dann sofort der **int**-Variablen `zahl1` zugewiesen wird. Entsprechend wird in der darauf folgenden Zeile der Wert des zweiten Textfeldes konvertiert und an die Variable `zahl2` übergeben.

Die Zeile der Anmerkung (3)

```
summe = zahl1 + zahl2;
```

besteht aus zwei Operatoren, `=` und `+`. Der `+`-Operator addiert die Werte der beiden `Zahlen`, der Zuweisungsoperator `=` weist diesen Wert der Variablen `summe` zu. Oder nicht so technisch formuliert: Erst wird die Summe `zahl1 + zahl2` errechnet und der Wert dann an `summe` übergeben.

In Anmerkung (4) schließlich wird das Ergebnis mit einem Text verknüpft, um ihn dann später am Bildschirm auszugeben.

```
ausgabe = "Die Summe ist " + summe;
```

Was geschieht hier? Die Summe eines Strings mit einer Zahl? Wenn Sie die Diskussion der obigen Abschnitte über Strings, Ziffernfolgen und Zahlen rekapitulieren, müssten Sie jetzt stutzig werden. String + Zahl? Ist das Ergebnis eine Zahl oder ein String?

Die Antwort ist eindeutig: String + Zahl ergibt — String! Java konvertiert automatisch ...! Der +-Operator addiert hier also nicht, sondern konkateniert. Sobald nämlich der erste Operand von + ein String ist, konvertiert Java alle weiteren Operanden *automatisch* in einen String und konkateniert.

Gibt beispielsweise der Anwender die Zahlen 13 und 7 ein, so ergibt sich die Wertetabelle 1.5 für die einzelnen Variablen unseres Programms.

Zeitpunkt	feld[0]	feld[1]	zahl1	zahl2	summe	ausgabe
vor Eingabe	—	—	—	—	—	—
nach Eingabe von 13 und 7	"13"	"7"	—	—	—	—
nach Typkonvertierung (2)	"13"	"7"	13	7	—	—
nach Anmerkung (3)	"13"	"7"	13	7	20	—
nach Anmerkung (4)	"13"	"7"	13	7	20	"Die Summe ist 20"

Tabelle 1.5: Wertetabelle für die Variablen im zeitlichen Ablauf des Programms

Konvertierung von String in weitere Datentypen

Entsprechend der Anweisung `Integer.parseInt()` zur Konvertierung eines Strings in einen **int**-Wert gibt es so genannte *parse*-Methoden für die anderen Datentypen für Zahlen, die alle nach dem gleichen Prinzip aufgebaut sind:

Datentyp.parseDatentyp(... Text ...)

Beispielsweise wird durch `Double.parseDouble("3.1415")` der String "3.1415" in die **double**-Zahl 3.1415 konvertiert. Hierbei sind `Integer` und `Double` sogenannte *Hüllklassen* (*wrapper classes*). Zu jedem der primitiven Datentypen existiert solch eine Klasse.

1.6.5 Einlesen von Kommazahlen

Ein häufiges Problem bei der Eingabe von Kommazahlen ist die kontinentaleuropäische Konvention des Dezimalkommas, nach der die Zahl π beispielsweise als 3,1415 geschrieben. Gibt ein Anwender jedoch eine Zahl mit einem Dezimalkomma ein, so entsteht bei Ausführung der `parseDouble`-Methode ein Laufzeitfehler (eine *Exception*) und das Programm stürzt ab.

Eine einfache Möglichkeit, solche Eingaben dennoch korrekt verarbeiten zu lassen, bietet die Methode `replace` der Klasse `String`. Möchte man in einem String `s` das Zeichen 'a' durch 'b' ersetzen, so schreibt man

```
s = s.replace('a', 'b');
```

Beipielsweise ergibt `s = "Affenhaar".replace('a','e');` den String "Affenheer". Mit Hilfe des `replace`-Befehls kann man eine Eingabe mit einem möglichen Komma also zunächst durch einen Punkt ersetzen und erst dann zu einem **double**-Wert konvertieren:

```
double x;
...
x = Double.parseDouble( feld[0].getText().replace(',', '.', ' ') );
```

1.6.6 Weitere Eingabemöglichkeiten

Eingabefenster mit einem Textfeld

Benötigt man nur eine einzige Eingabe, so gibt es eine etwas kürzere Möglichkeit als den Eingabeblock, nämlich die Methode `showInputDialog` der Klasse `JOptionPane`, die mit der Anweisung

```
eingabe = JOptionPane.showInputDialog( "... Eingabeaufforderung ...");
```

(falls `eingabe` vorher als `String` deklariert wurde) aufgerufen wird. Mit dem Eingabeblock wäre die entsprechende Eingabe also `eingabe = feld[0].getText()`, d.h. das Aufbauen des Dialogfensters und das Abgreifen der Eingabe geschieht in einer einzigen Anweisung.

Scanner-Eingaben über die Konsole

Eine Möglichkeit, Eingaben über das Konsolenfenster einzulesen, bietet die folgende Konstruktion mit der `Scanner`-Klasse:

```
Scanner sc = new Scanner(System.in);
System.out.print("Eingabe: ");
String ein = sc.next();
```

Möchte man mehrere Eingaben einlesen, so werden einfach weitere `sc.next()`-Anweisungen angegeben.

args-Eingaben über die Konsole

Eingaben, die bereits beim Start der Applikation einzugeben sind, können über die `args`-Parameter der `main`-Methode durch die folgenden Anweisungen eingelesen werden:

```
public class ArgsEingabe {
    public static void main(String[] args) {
        String ein1 = "", ein2 = "";
        if ( args.length > 0 ) ein1 = args[0];
        if ( args.length > 1 ) ein2 = args[1];
    }
}
```

Um auf diese Weise zwei Eingaben zu machen, muss die Applikation mit den Daten aufgerufen werden, etwa:

```
java ArgsEingabe Hallo Ihr
```

Die verschiedenen Eingabestrings werden also einfach durch Leerzeichen getrennt hinter den Klassennamen geschrieben. Die Verallgemeinerung dieser Konstruktion für noch mehr Eingaben ist offensichtlich. Diese eher technische Eingabeart eignet sich sehr gut für den Programmierer, um während der Entwicklung eines Programms verschiedene Eingaben verarbeiten zu lassen.

1.7 Zusammenfassung

Applikationen, Strings und Ausgaben

- Ein Java-Programm ist eine Klasse.

- Eine Klasse in Java muss in einer Datei abgespeichert werden, die genau so heißt wie die Klasse, mit der Erweiterung „.java“. (Ausnahme: Wenn mehrere Klassen in einer Datei definiert werden.)
- Klassennamen beginnen mit einem Großbuchstaben und bestehen aus einem Wort. Verwenden Sie „sprechende“ Namen, die andeuten, was die Klasse bedeutet. Im Gegensatz dazu werden Methoden- und Variablennamen klein geschrieben.
- Java ist schreibungssensitiv (*case sensitive*), d.h. es wird zwischen Groß- und Kleinschreibung unterschieden.
- Eine Applikation benötigt die Methode **main**; sie heißt daher auch „main-Klasse“.
- Die Methode **main** ist der Startpunkt für die Applikation. Sie enthält der Reihe nach alle Anweisungen, die ausgeführt werden sollen.
- Die Syntax für eine Applikation lautet:

```
public class Klassenname {
    public static void main( String[] args ) {
        Deklarationen und Anweisungen;
    }
}
```

- Kommentare sind Textabschnitte, die in den Quelltext eingefügt werden und vom Compiler nicht beachtet werden. Es gibt drei Kommentararten:
 - `// ...` einzeiliger Kommentar
 - `/* ... */` Es können auch mehrerzeiliger Kommentar
 - `/** ... */` Dokumentationskommentar (wird von javadoc automatisch verarbeitet)
- Ein String ist eine Kette von Unicode-Zeichen, die von Anführungszeichen (") umschlossen ist. Beispiele: `"aBC ijk"`, oder der leere String `""`.
- Die Ausgabe eines Strings `"...Text..."` geschieht mit der Anweisung


```
javax.swing.JOptionPane.showMessageDialog( null, "... Text ... " );
```

 bzw. einfach mit


```
JOptionPane.showMessageDialog( null, "... Text ..." );
```

 wenn die Klasse `JOptionPane` mit `import javax.swing.JOptionPane;` vor der Klassendeklaration importiert wurde.
- Eine weitere Ausgabe im Befelszeilenfenster („Konsole“) geschieht durch die Anweisungen

```
System.out.println("...");    oder    System.out.print("...");
```

Der Unterschied der beiden Anweisungen besteht darin, dass `println` nach der Ausgabe des Strings einen Zeilenumbruch ausführt.

Variablen, Datentypen und Operatoren

- Der Datentyp für Strings ist in Java die Klasse `String`.
- In Java gibt es acht primitive Datentypen,
 - `boolean` für Boolesche Werte,
 - `char` für einzelne Unicode-Zeichen,
 - `byte`, `short`, `int` und `long` für ganze Zahlen,
 - `float` und `double` für reelle Zahlen

Die Datentypen unterscheiden sich in ihrer Speichergröße und in ihrem Wertebereich.

- Werte können in Variablen gespeichert werden. Eine Variable ist ein Name, der ein zusammenhängendes Wort ohne Sonderzeichen ist und auf einen bestimmten Speicherbereich verweist. Variablen müssen deklariert werden, d.h. sie müssen vor einer Wertzuweisung einem eindeutigen Datentyp zugeordnet sein.
- Variablen müssen initialisiert werden, bevor man sie benutzen kann. Verwendung nichtinitialisierter Variablen in einer Zuweisung auf der rechten Seite führt zu einer Fehlermeldung.
- Ein Operator ist durch seine Wirkung auf seine Operanden definiert. Insbesondere kann ein Operator zwar dieselbe Bezeichnung wie ein anderer haben, aber eine ganz andere Wirkung besitzen. Z.B. bedeutet `+` für zwei Zahlen eine Addition, für zwei Strings eine Konkatenation, oder `/` für zwei `int`-Werte eine ganzzahlige Division, für mindestens einen `double`-Wert aber eine Division reeller Zahlen.
- Operatoren haben eine Rangfolge (Präzedenz), die bestimmt, welcher der Operatoren in einer Anweisung zuerst bearbeitet wird. Haben Operatoren den gleichen Rang, wird von links nach rechts (bzw. bei Klammern von innen nach außen) bewertet.
- Java kennt die Rechenoperationen `+`, `-`, `*`, `/`, `%`.
- Die Division zweier `int`- oder `long`-Werte ist in Java automatisch eine ganzzahlige Division, d.h. `14/4` ergibt `3`.
- Der `cast`-Operator konvertiert primitive Datentypen in andere primitive Datentypen. Z.B. wird (`int`) `3.14` zum integer-Wert `3`. Der `cast`-Operator kann auch komplexe Datentypen (Objekte) in andere komplexe Datentypen konvertieren.
- Mit den modifizierten Zuweisungsoperatoren `+=`, `-=`, `*=`, `/=`, `%=` wird der Wert der Variablen auf der linken Seite durch ihren Wert vor der Anweisung um die entsprechende Operation mit dem Wert auf der rechten Seite aktualisiert.
- Der Inkrementoperator `++` erhöht den Wert der direkt vor oder hinter ihm stehenden Variablen um 1, der Dekrementoperator `-` erniedrigt ihn um 1. Stehen die Operatoren allein in einer Anweisung, so ist ihre Wirkung gleich, also `c++;` \iff `++c;` und `c-;` \iff `-c;`.
- Die Klasse `Math` in dem Standardpaket `java.lang` (welches man nicht importieren muss) liefert die gängigen mathematischen Funktionen, z.B. `sin` und `cos` oder `pow`. Man verwendet sie durch den Aufruf `Math.sin(3.14)` oder `Math.pow(Math.PI, 0.5)`. Eine Liste der Funktionen und der für sie zu verwendenden Syntax (welche Parameter mit welchen Datentypen erforderlich und welcher Datentyp des Ergebnisses zu erwarten ist) finden in der API-Dokumentation, online unter java.sun.com.

- Die Klasse `DecimalFormat` aus dem Paket `java.text` ermöglicht mit der Anweisung

```
java.text.DecimalFormat variable = java.text.DecimalFormat( "#,##0.00" );
```

die Ausgabe einer **double**-Zahl x mit genau zwei Nachkommastellen mittels

```
variable.format( x );
```

Eingaben

- Eine Eingabe von Daten kann mit dem „Eingabeblock“ geschehen:

```
// Eingabefelder aufbauen:
JTextField[] feld = {new JTextField(), new JTextField()};
Object[] msg = {"Erster Text:", feld[0], "Zweiter Text:", feld[1]};
// Dialogfenster anzeigen:
int click = JOptionPane.showConfirmDialog(null, msg, "Eingabe", 2);
```

Dazu muss das Paket mit `import javax.swing.*;` vor der Klassendeklaration importiert werden.

- Ebenso kann eine Eingabe mit

```
String ein = JOptionPane.showInputDialog( "... Text ..." );
```

oder über das Konsolenfenster mit der `Scanner`-Klasse und ihrer Methode `next()` durchgeführt werden.

- Ein eingegebener Wert ist zunächst immer ein `String`. Um ihn als Zahl zu speichern, muss er konvertiert werden. Die Konvertierung von einem `String` in einen **int** bzw. **double**-Wert geschieht mit den Methoden `Integer.parseInt("...")` bzw. `Double.parseDouble("...")`
- Zur Darstellung des Ablaufs eines Programms, in dem Variablen verschiedene Werte annehmen, eignen sich Wertetabellen.
- Der Operator `+` hat abhängig von seinen Operanden verschiedene Wirkung. Sei `op1 + op2` gegeben. Dann gilt:
 - Ist der erste Operand `op1` ein `String`, so versucht der Java-Interpreter, den zweiten Operand `op2` in einen `String` zu konvertieren und konkateniert beide `Strings`.
 - Sind *beide* Operanden Zahlen (also jeweils von einem der primitiven Datentypen **byte**, **short**, **int**, **long**, **float**, **double**), so werden sie addiert. Hierbei wird der Ergebniswert in den „größeren“ der beiden Operandentypen konvertiert, also z.B. „**int** + **double**“ ergibt „**double**“.
 - Ist `op1` eine Zahl und `op2` ein `String`, so kommt es zu einem Kompilierfehler.

Kapitel 2

Prozedurale Programmierung

In diesem Kapitel beschäftigen wir uns mit den Grundlagen von Algorithmen. Ein *Algorithmus*, oder eine *Routine*, ist eine exakt definierte Prozedur zur Lösung eines gegebenen Problems, also eine genau angegebene Abfolge von Anweisungen oder Einzelschritten. Im Rahmen der Objektorientierung bildet ein Algorithmus eine „Methode“.

Bei einem Algorithmus spielen die zeitliche Abfolge der Anweisungen, logische Bedingungen (Abb. 2.1) und Wiederholungen eine wesentliche Rolle. Das Wort „Algorithmus“ leitet sich ab von

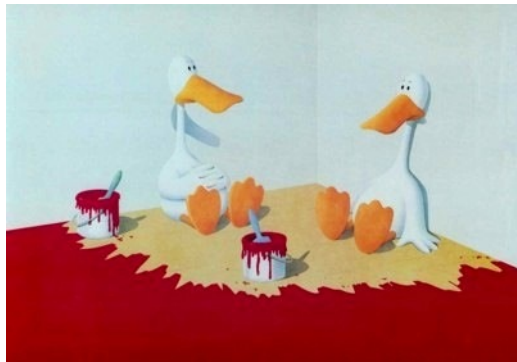


Abbildung 2.1: Für einen korrekt funktionierenden Algorithmus spielen Logik und zeitliche Reihenfolge der Einzelschritte eine wesentliche Rolle. ©1989 Michael Bedard „*The Failure of Marxism*“

dem Namen des bedeutenden persisch-arabischen Mathematikers Al-Chwarizmi¹ (ca. 780 – ca. 850). Dessen mathematisches Lehrbuch *Über das Rechnen mit indischen Ziffern*, um 825 erschienen und etwa 300 Jahre später (!) vom Arabischen ins Lateinische übersetzt, beschrieb das Rechnen mit dem im damaligen Europa unbekannten Dezimalsystem der Inder.² Auf diese Weise wurde das Wort Algorithmus zu einem Synonym für Rechenverfahren.

2.1 Logik und Verzweigung

2.1.1 Die Verzweigung

Die *Verzweigung* (auch: *Selektion*, *Auswahlstruktur* oder *if-Anweisung*) eine Anweisung, die abhängig von einer Bedingung zwischen alternativen Anweisungsfolgen auswählt. In Java hat sie die folgende Syntax:

¹<http://de.wikipedia.org/wiki/Al-Chwarizmi>

²George Ifrah: *The Universal History of Numbers*. John Wiley & Sons, New York 2000, S. 362

```
if ( Bedingung ) {
    Anweisungen;
} else {
    Anweisungen;
}
```

Falls die Bedingung wahr ist (d.h. *Bedingung* = **true**), wird der Block direkt nach der Bedingung (kurz: der „**if**-Zweig“) ausgeführt, ansonsten (*Bedingung* = **false**) der Block nach **else** (der „**else**-Zweig“). Der **else**-Zweig kann weggelassen werden, falls keine Anweisung ausgeführt werden soll, wenn die Bedingung falsch ist:

```
if ( Bedingung ) {
    Anweisungen;
}
```

Die Bedingung ist eine logische Aussage, die entweder wahr (**true**) oder falsch (**false**) ist. Man nennt sie auch einen *Booleschen Ausdruck*. Ein Beispiel für eine Verzweigung liefert die folgende Applikation:

```
import javax.swing.*;
/** bestimmt, ob ein Test bei eingegebener Note bestanden ist.
 */
public class Testergebnis {
    public static void main( String[] args ) {
        double note;
        String ausgabe;

        // Eingabeblock:
        JTextField[] feld = {new JTextField()};
        Object[] msg = {"Geben Sie die Note ein (1, ..., 6):", feld[0]};
        int click = JOptionPane.showConfirmDialog(null, msg, "Eingabe", 2);

        // Eingabe vom Typ String nach int konvertieren:
        note = Double.parseDouble( feld[0].getText() );           // (1)

        // Ergebnis bestimmen:
        if ( note <= 4 ) {
            ausgabe = "Bestanden";
        } else {
            ausgabe = "Durchgefallen";
        }

        JOptionPane.showMessageDialog(null, ausgabe);
    }
}
```

Diese Applikation erwartet die Eingabe einer Note und gibt abhängig davon aus, ob die Klausur bestanden ist oder nicht.

2.1.2 Logische Bedingungen durch Vergleiche

Durch die **if**-Anweisung wird eine Entscheidung auf Grund einer bestimmten *Bedingung* getroffen. Bedingungen für zwei Werte eines primitiven Datentyps können in Java durch *Vergleichsoperatoren* gebildet werden:

Algebraischer Operator	Operator in Java	Beispiel	Bedeutung
=	==	x == y	x ist gleich y
≠	!=	x != y	x ist ungleich y
>	>	x > y	x ist größer als y
<	<	x < y	x ist kleiner als y
≥	>=	x >= y	x ist größer gleich y
≤	<=	x <= y	x ist kleiner gleich y

Merkregel 4. Der Operator == wird oft verwechselt mit dem Zuweisungsoperator =. Der Operator == muss gelesen werden als „ist gleich“, der Operator = dagegen als „wird“ oder „bekommt den Wert von“.

Eine wichtige Einschränkung muss man bei den algebraischen Vergleichsoperatoren erwähnen:

Merkregel 5. Alle algebraischen Vergleichsoperatoren können nur zwei Ausdrücke von elementarem Datentyp vergleichen. Zwei Strings können jedoch nur mit der Methode equals verglichen werden:

```
string1.equals(string2).
```

Der Vergleichsoperator == funktioniert bei Strings nicht! Entsprechend kann man auch den Ungleichheitsoperator != bei Strings nicht verwenden, stattdessen muss

```
!string1.equals(string2)
```

verwendet.

Das folgende Beispielprogramm verwendet mehrere if-Anweisungen, um zwei eingegebene Zahlen zu vergleichen.

```
import javax.swing.*;

/**
 * Benutzt if-Anweisungen und Vergleichsoperatoren
 */
public class Vergleich {
    public static void main( String[] args ) {
        String eingabe1, eingabe2;    // Eingabestrings
        int zahl1, zahl2;             // zu vergleichende Zahlen
        String ausgabe = "";          // (1)

        // Eingabeblock:
        JTextField[] feld = {new JTextField(), new JTextField()};
        Object[] msg = {"Geben Sie zwei Zahlen ein:", feld[0], feld[1]};
        int click = JOptionPane.showConfirmDialog(null, msg, "Eingabe", 2);

        // Speichern der Zahlen als String:
        eingabe1 = feld[0].getText();
        eingabe2 = feld[1].getText();
        // Konvertierung der Eingabe von String nach int:
        zahl1 = Integer.parseInt( eingabe1 );
        zahl2 = Integer.parseInt( eingabe2 );
    }
}
```

```

// Bestimmung des Ausgabestrings durch Vergleiche:
if ( zahl1 == zahl2 ) {
    ausgabe += zahl1 + " == " + zahl2;           // (2)
}
if ( zahl1 != zahl2 ) {
    ausgabe += "\n" + zahl1 + " != " + zahl2;
}
if ( zahl1 < zahl2 ) {
    ausgabe += "\n" + zahl1 + " < " + zahl2;
}
if ( zahl1 > zahl2 ) {
    ausgabe += "\n" + zahl1 + " > " + zahl2;
}
if ( zahl1 <= zahl2 ) {
    ausgabe += "\n" + zahl1 + " <=" + zahl2;
}
if ( zahl1 >= zahl2 ) {
    ausgabe += "\n" + zahl1 + " >=" + zahl2;
}

if ( eingabe1.equals( eingabe2 ) ) {
    ausgabe += "\n" + eingabe1 + " ist gleich " + eingabe2;    // (3)
}

if ( !eingabe1.equals( eingabe2 ) ) {
    ausgabe += "\n" + eingabe1 + " ist nicht gleich " + eingabe2; // (4)
}

// Ausgabe des Ergebnisses:
JOptionPane.showMessageDialog(
    null, ausgabe, "Vergleichsergebnisse", JOptionPane.PLAIN_MESSAGE );
}

```

In der main-Methode werden, vor dem Eingabeblock, fünf Variablen deklariert, und zwar drei vom Typ String und zwei Variablen vom Typ **int**. Insbesondere wird die Variable `ausgabe` in Anmerkung (1) deklariert und direkt der *leeren String* zugewiesen: die Variable wird initialisiert.

Was passiert in der Anweisung von Anmerkung (2)? Die beiden Zahlen `zahl1` und `zahl2` werden auf Gleichheit geprüft. Sind sie gleich, so wird die Anweisung `ausgabe += zahl1 ...` durchgeführt. Wir wissen, dass der `+=`-Operator an den aktuellen String den Wert an der rechten Seite anhängt (man könnte also auch schreiben: `ausgabe += zahl1 + ...`). Somit erhält die Variable `ausgabe` den Wert

`ausgabe + zahl1 + " == " + zahl2`

zugewiesen. Nehmen wir also an, `zahl1` und `zahl2` haben beide den Wert 123, so wird also dem *vor* der Anweisung leeren String `""` der String `"123 == 123"` hinzugefügt: Da `ausgabe` vom Typ String ist, werden durch die `+`-Operatoren sowohl `zahl1` als auch `zahl2` automatisch in Strings konvertiert, d.h. sie `+`-Operatoren konkatenieren. Nach der Zuweisung hat damit die Variable `ausgabe` den Stringwert

`"123 == 123"`

Entsprechend wächst die Länge des Strings `ausgabe`, je mehr Bedingungen der folgenden `if`-Anweisungen zutreffen. In der Zeile von Anmerkung (3) wird ein neuer Operator verwendet, `+=` (*addition assignment operator*):

`ausgabe += eingabe1; bewirkt: ausgabe += eingabe1;`

Für unser Beispiel wird der String also schließlich den Wert

```
"123 == 123\n123 <= 123\n123 >= 123 ist gleich 123"
```

haben, was am Bildschirm ausgegeben wird als

```
123 == 123
123 <= 123
123 >= 123
123 ist gleich 123
```

2.1.3 Logische Operatoren

Logische Operatoren werden benutzt, um logische Aussagen zu verknüpfen oder zu negieren. Technisch gesehen verknüpfen sie Boolesche Werte (**true** und **false**) miteinander. Java stellt die Grundoperationen UND, ODER, XOR („exklusives ODER“ oder „Antivalenz“) und NICHT zur Verfügung und bietet darüber hinaus die Möglichkeit, das Auswertungsverhalten der Operanden zu beeinflussen.

Logisches AND (&&)

Der &&-Operator führt die logische UND-Verknüpfung durch. Er ergibt **true**, wenn seine beiden Operanden **true** ergeben, ansonsten **false**.

Logisches OR (||)

Der ||-Operator führt die logische ODER-Verknüpfung durch. Er ergibt **true**, wenn einer seiner beiden Operanden **true** ergibt, ansonsten **false**.

Das Exklusive Oder XOR (^)

Der XOR-Operator ^ führt die logische Verknüpfung des Exklusiven ODER durch. Hierbei ist $A \wedge B$ **true**, wenn A und B *unterschiedliche* Wahrheitswerte haben, und **false**, wenn sie gleiche Wahrheitswerte haben.

Logische Negation (!)

Der Negations-Operator ! wird auf nur einen Operanden angewendet. Er ergibt **true**, wenn sein Operand **false** ist und umgekehrt.

Die Wahrheitstabellen in Tab. 2.1 zeigen die möglichen Ergebnisse der logischen Operatoren. Die erste Spalte ist beispielsweise folgendermaßen zu lesen: Wenn Aussage a den Wert **false** besitzt und b ebenfalls, dann ist $a \&\& b$ **false**, $a \parallel b$ ist **false**, usw.

a	b	$a \&\& b$	$a \parallel b$	$a \wedge b$	$\neg a$
false	false	false	false	false	true
false	true	false	true	true	–
true	false	false	true	true	false
true	true	true	true	false	–

Tabelle 2.1: Wahrheitstabellen für verschiedene logische Operatoren.

Manchen fällt es leichter, sich Wahrheitstabellen mit 0 und 1 zu merken, wobei 0 = **false** und 1 = **true** (Tab. 2.2). Diese „Mathematisierung“ von **false** und **true** hat eine wunderschöne Konsequenz.

a	b	$a \&\& b$	$a \parallel b$	$a \wedge b$	$\neg a$
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Tabelle 2.2: Wahrheitstabellen für verschiedene logische Operatoren.

Man kann nämlich auf diese Weise mit den logischen Operatoren $\&\&$, \parallel und \wedge algebraisch rechnen, wenn man sie wie folgt durch die Grundrechenarten ausdrückt:

$$a \&\& b = a \cdot b, \quad a \parallel b = a + b - ab, \quad a \wedge b = (a + b) \% 2, \quad \neg a = (a + 1) \% 2, \quad (2.1)$$

mit $a, b \in \{0, 1\}$. (Probieren Sie es aus!) Daher spricht man völlig zurecht von einer „Algebra“, und zwar nach ihrem Entdecker von der *Booleschen Algebra*. Dass Computer mit ihrer Binärlogik überhaupt funktionieren, liegt letztlich an der Eigenschaft, dass die Wahrheitswerte eine Algebra bilden. Man erkennt an (2.1) übrigens direkt, dass

$$\neg a = a \wedge 1 \quad \text{für } a \in \{0, 1\}. \quad (2.2)$$

Anwendung im Qualitätsmanagement

Betrachten wir als eine Anwendung das folgende Problem des Qualitätsmanagements eines Produktionsbetriebs. Es wird jeweils ein Los von 6 Artikeln, aus dem eine Stichprobe von drei zufällig ausgewählten Artikeln durch drei Roboter automatisiert qualitätsgeprüft wird. Die Artikel in dem Los sind von 0 bis 5 durchnummeriert. Das Programm soll so aufgebaut sein, dass es drei Zahlen a_1 , a_2 und a_3 zufällig aus der Menge $\{0, 1, \dots, 5\}$ bestimmt, wobei der Wert von a_1 die Nummer des Artikels beschreibt, den Roboter Nr. 1 prüft, der Wert von a_2 den von Roboter Nr. 2 zu prüfenden Artikel, und a_3 den von Roboter Nr. 3 zu prüfenden Artikel. Werden für (a_1, a_2, a_3) beispielsweise die drei Werte $(3, 0, 5)$ bestimmt, also

$$(a_1, a_2, a_3) = (3, 0, 5),$$

so soll Roboter 1 den Artikel 3 prüfen, Roboter 2 den Artikel 0 und Roboter 3 den Artikel 5. Für den Ablauf der Qualitätsprüfung ist es nun notwendig, dass die Werte von a_1 , a_2 und a_3 unterschiedlich sind, denn andernfalls würden mehrere Roboter einen Artikel prüfen müssen. Falls also nicht alle Zahlen unterschiedlich sind, soll das Programm eine Warnung ausgeben.

Zufallszahlen mit `Math.random()`. Um das Programm zu schreiben, benötigen wir zunächst eine Möglichkeit, eine Zufallszahl aus einem gegebenen Zahlbereich zu bekommen. In Java geschieht das am einfachsten mit der Funktion `Math.random()`. Das ist eine Funktion aus der Klasse `Math`, die einen zufälligen Wert z vom Typ **double** von 0 (einschließlich) bis 1 ausschließlich ergibt. Mit der Anweisung

```
double z = Math.random();
```

erhält die Variable z also einen Zufallswert aus dem Einheitsintervall $[0, 1[$, also $0 \leq z < 1$. Was ist zu tun, um nun daraus eine ganze Zahl x aus dem Bereich $[0, 6[$ zu bilden? Zwei Schritte sind dazu nötig, einerseits muss der Bereich vergrößert werden, andererseits muss der **double**-Wert zu einem **int**-Wert konvertiert werden. Der Bereich wird vergrößert, indem wir den z -Wert mit 6 multiplizieren, also $x = 6 \cdot z$. Damit ist x ein zufälliger Wert mit $0 \leq x < 6$. Daraus kann durch einfaches Casten ein ganzzahliger Wert aus der Menge $\{0, 1, \dots, 5\}$ gemacht werden. Zu einer einzigen Zeile zusammengefasst können wir also mit


```
int x = (int) ( 6 * Math.random() );
```

eine Zufallszahl x mit $x \in \{0, 1, \dots, 5\}$ erhalten. Auf diese Weise können wir also den drei Zahlen a_1 , a_2 und a_3 zufällige Werte geben.

Fallunterscheidung mit logischen Operatoren. Wie wird nun geprüft, ob eine Warnmeldung ausgegeben werden soll? Da wir schon in der Alltagssprache sagen: „Wenn nicht alle Zahlen unterschiedlich sind, dann gib eine Warnung aus“, müssen wir eine **if**-Anweisung verwenden. Um die hinreichende Bedingung für die Warnmeldung zu formulieren, überlegen wir uns kurz, welche Fälle überhaupt eintreten können.

1. Fall: *Alle drei Zahlen sind unterschiedlich.* Als logische Aussage formuliert lautet dieser Fall:

```
a1 != a2 && a1 != a3 && a2 != a3    ist wahr.
```

2. Fall: *Genau zwei Zahlen sind gleich.* Als logische Aussage ist das der etwas längliche Ausdruck

```
(a1 == a2 && a1 != a3 && a2 != a3) ||  
(a1 != a2 && a1 == a3 && a2 != a3) ||  
(a1 != a2 && a1 != a3 && a2 == a3)    ist wahr.
```

⇔

```
(a1 == a2 && a1 != a3) ||  
(a1 != a2 && a1 == a3) ||  
(a2 == a3 && a1 != a3)    ist wahr.
```

3. Fall: *Alle drei Zahlen sind gleich.* Mit logischen Operatoren ausgedrückt:

```
a1 == a2 && a1 == a3 && a2 == a3    ist wahr.
```

Für unsere Warnmeldung benötigen wir nun jedoch gar nicht diese genaue Detaillierung an Fallunterscheidungen, also die Kriterien für die genaue Anzahl an übereinstimmenden Zahlen. Wir müssen nur prüfen, ob *mindestens* zwei Zahlen übereinstimmen. Die entsprechende Bedingung lautet:

```
a1 == a2 || a1 == a3 || a2 == a3    ist wahr.
```

Das deckt die Fälle 2 und 3 ab. Entsprechend ergibt sich das Programm wie folgt.

```
public class Qualitaetspruefung {  
    public static void main(String[] args) {  
        // 3 zufällig zur Prüfung ausgewählte Artikel aus {0, 1, ..., 5}:  
        int a1 = (int) ( 6 * Math.random() );  
        int a2 = (int) ( 6 * Math.random() );  
        int a3 = (int) ( 6 * Math.random() );  
  
        if( a1 == a2 || a1 == a3 || a2 == a3 ) {  
            System.out.print("WARNUNG! Ein Artikel wird mehrfach geprüfert: ");  
        }  
        System.out.println("(" + a1 + ", " + a2 + ", " + a3 + ")");  
    }  
}
```

2.1.4 Die if-else-if-Leiter

Möchte man eine endliche Alternative von Fällen abfragen, also mehrere Fälle, von denen nur einer zutreffen kann, so kann man mehrere **if**-Anweisungen verschachteln zu einer sogenannten *kaskadierten Verzweigung* oder „**if-else-if**-Leiter“ (*if-else-ladder*). Beipielsweise kann man zur Ermittlung der (meteorologischen) Jahreszeit abhängig vom Monat den folgenden Quelltextausschnitt („*Snippet*“) verwenden:

```
int monat = 5; // Mai, als Beispiel
String jahreszeit = "";
if (monat == 12 || monat == 1 || monat == 2) {
    jahreszeit = "Winter";
} else if (monat == 3 || monat == 4 || monat == 5) {
    jahreszeit = "Frühling";
} else if (monat == 6 || monat == 7 || monat == 8) {
    jahreszeit = "Sommer";
} else if (monat == 9 || monat == 10 || monat == 11) {
    jahreszeit = "Herbst";
} else {
    jahreszeit = "- unbekannter Monat -";
}
System.out.println("Im Monat " + monat + " ist " + jahreszeit);
```

2.1.5 Gleichheit von double-Werten

Prüft man zwei double-Werte auf Gleichheit, so kann es zu unerwarteten Ergebnissen führen. Beispielsweise ergibt der Vergleich ($0.4 - 0.3 == 0.1$) den Wert `false`, wie man schnell durch folgendes Quelltextfragment selber überprüfen kann:

```
double x = 0.4, y = 0.3;
System.out.println(x - y == 0.1);
```

Ursache ist die Tatsache, dass Werte vom Datentyp `double` als binäre Brüche gespeichert werden, und speziell 0.4 und 0.3 aber keine endliche Binärbruchentwicklung besitzen (vgl. S. 127).

2.1.6 Abfragen auf Zahlbereiche

Betrachten wir nun ein kleines Problem, bei dem wir zweckmäßigerweise mehrere Vergleiche logisch miteinander verknüpfen. Für eine eingegebene Zahl x soll ausgegeben werden, ob gilt

$$x \in [0, 10[\cup [90, 100]$$

```
import javax.swing.*;
/** bestimmt, ob eine eingegebene Zahl x in der Menge [0,10[∪[90,100] ist. */
public class Zahlbereich {
    public static void main( String[] args ) {
        double x;
        boolean istInMenge;
        String ausgabe;

        // Eingabeblock:
        JTextField[] feld = {new JTextField()};
        Object[] msg = {"Geben Sie eine Zahl ein:", feld[0]};
        int click = JOptionPane.showConfirmDialog(null, msg, "Eingabe", 2);

        // Eingabe nach double konvertieren:
        x = Double.parseDouble( feld[0].getText() );
```

```

// Booleschen Wert bestimmen:
istInMenge = ( x >= 0 ) && ( x < 10 );
istInMenge |= ( x >= 90 ) && ( x <= 100 );
// Wert für Variable ausgabe bestimmen:
if ( istInMenge ) {
    ausgabe = "Zahl " + x + '\u2208 [0,10[ \u222A [90,100]';
} else {
    ausgabe = "Zahl " + x + '\u2209 [0,10[ \u222A [90,100]';
}
OptionPane.showMessageDialog(null,ausgabe);
}
}

```

Zu beachten ist, dass nun zur Laufzeit des Programms nach der Eingabe der Wert der Booleschen Variable `istInMenge` auf `true` gesetzt ist, wenn eine der beiden `&&`-verknüpften Bedingungen zutrifft.

Abfragen auf einen Zahlbereich für `int`-Werte

Möchte man in Java überprüfen, ob ein `int`-Wert n in einem bestimmten Bereich von a bis b liegt, also ob $n \in [a, b]$, so kann man sich die Darstellung des Zweierkomplements (der Darstellung von `int`- und `long`-Werten) zunutze machen und den Wahrheitswert mit nur einem Vergleich durchführen:

$$(a \leq n \ \&\& \ n \leq b) \iff (n - a - 0x80000000 \leq b - a - 0x80000000) \quad (2.3)$$

Hierbei ist $0x80000000 = -2^{31} = -\text{Integer.MAX_VALUE} - 1$. Damit ist der Wert $x - a - 0x80000000$ für jeden Integerwert x stets positiv.

„Short-Circuit“-Evaluation und bitweise Operatoren

Java stellt die UND- und ODER-Verknüpfungen in zwei verschiedenen Varianten zur Verfügung, nämlich mit Short-Circuit-Evaluation oder ohne.

Bei der *Short-Circuit-Evaluation* eines logischen Ausdrucks wird ein weiter rechts stehender Teilausdruck nur dann ausgewertet, wenn er für das Ergebnis des Gesamtausdrucks noch von Bedeutung ist. Falls in dem Ausdruck $a \ \&\& \ b$ also bereits a falsch ist, wird zwangsläufig immer auch $a \ \&\& \ b$ falsch sein, unabhängig von dem Resultat von b . Bei der Short-Circuit-Evaluation wird in diesem Fall b gar nicht mehr ausgewertet. Analoges gilt bei der Anwendung des ODER-Operators.

Die drei Operatoren `&`, `|` und `^` können auch als bitweise Operatoren auf die ganzzahligen Datentypen `char`, `byte`, `short`, `int` und `long` angewendet werden.

2.1.7 Bitweise Operatoren

Neben den drei Operatoren `&`, `|` und `^`, die auch auf Boolesche Operanden wirken, gibt es noch weitere bitweise Operatoren, die auf die ganzzahligen Datentypen `char`, `byte`, `short`, `int` und `long` angewendet werden. Einen Überblick über sie gibt Tabelle 2.3.

Die logischen Verknüpfungsoperationen (`&`, `|`, `^`) wirken also jeweils auf jedes einzelne Bit der Binar­darstellung (bei `char` der Unicode-Wert des Zeichens). Z.B. ergibt $25 \wedge 31 = 6$, denn

$$\begin{array}{r}
 25_{10} = 11001_2 \\
 \wedge 31_{10} = 11111_2 \\
 \hline
 00110_2
 \end{array} \quad (2.4)$$

Will man die Schiebeoperatoren auf `long`-Werte anwenden, so ist Sorgfalt geboten, denn das Auftreten nur eines `int`-Wertes genügt, um auch das Ergebnis nach `int` zu zwingen und damit nur noch den kleineren Speicherbereich zu nutzen. Hier kann schon eine einzige Wertzuweisung einer `long`-Variablen

³Für $x \in \mathbb{R}$ ist $\lfloor x \rfloor$ definiert als die größte natürliche Zahl n , für die gilt $n \leq x$. Z.B. $\lfloor \pi \rfloor = 3$, $\lfloor -\pi \rfloor = -4$, $\lfloor 5 \rfloor = 5$.

Operator	Bezeichnung	Bedeutung
\sim	Einerkomplement	$\sim a$ entsteht aus a , indem alle Bits von a invertiert werden
$ $	Bitweises ODER	$a b$ ergibt den Wert, der durch die ODER-Verknüpfung der korrespondierenden Bits von a und b entsteht
$\&$	Bitweises UND	$a \& b$ ergibt den Wert, der durch die UND-Verknüpfung der korrespondierenden Bits von a und b entsteht.
\wedge	Bitweises XOR	$a \wedge b$ ergibt den Wert, der durch die XOR-Verknüpfung der korrespondierenden Bits von a und b entsteht.
\ll	Linksschieben (mit Vorzeichen)	$a \ll b$ ergibt den Wert, der durch Schieben aller Bits von a um b Positionen nach links entsteht. Das höchstwertige Bit (also wg. der Zweierkomplementdarstellung: das Vorzeichen) erfährt keine besondere Behandlung. Es gilt also $a \ll b = 2^b a$ für $b > 0$, solange a und $2^b a$ noch im darstellbaren Bereich des Datentyps liegt. Der Schiebeoperator gibt nur int - oder long -Werte zurück. Für int -Werte gilt stets $a \ll -b = a \ll (32 - b)$, für long -Werte $a \ll -b = a \ll (64 - b)$.
\gg	Rechtsschieben mit Vorzeichen	$a \gg b$ ergibt den Wert, der durch Schieben aller Bits von a um b Positionen nach rechts entsteht. Falls das höchstwertige Bit gesetzt ist (a also negativ ist, wg. Zweierkomplement), wird auch das höchstwertige Bit des Resultats gesetzt. Es gilt daher $a \gg b = \lfloor a/2^b \rfloor$ für $b > 0$, solange a und $\lfloor a/2^b \rfloor$ noch im darstellbaren Bereich des Datentyps liegt. ³ Der Schiebeoperator gibt nur int - oder long -Werte zurück. Für int -Werte gilt $a \gg -b = a \gg (32 - b)$, für long $a \gg -b = a \gg (64 - b)$.
\ggg	vorzeichenloses Rechtsschieben	$a \ggg b$ ergibt den Wert, der durch Schieben aller Bits von a um b Positionen nach rechts entsteht, wobei das höchstwertige Bit bei $b \neq 0$ immer auf 0 gesetzt wird. Es gilt also $a \ggg b = \lfloor a /2^b \rfloor$ für $b > 0$, solange $\lfloor a /2^b \rfloor$ noch im darstellbaren Bereich des Datentyps liegt. ³ Der Schiebeoperator gibt nur int - oder long -Werte zurück. Für int -Werte gilt $a \ggg -b = a \ggg (32 - b)$. für long -Werte entsprechend $a \ggg -b = a \ggg (64 - b)$.

Tabelle 2.3: Bitweise Operatoren

mit einem **int**-Wert ausreichen. Man sollte daher Wertzuweisungen durch Konstante explizit durch das Suffix **L** als **long** ausweisen, also z.B.

```
long a = 0L, b = 0xffffL, c = Long.parseLong("3");
```

Bitmasken

Oft werden die bitweisen Operatoren für Bitmasken zur effizienten Speicherung und Filterung von Information in einem binären Datentyp verwendet. Eine *Bitmaske* bezeichnet eine mehrstellige Binärzahl, mit der Informationen aus einer anderen Binärzahl gefiltert oder gespeichert werden können. Verwendet wird dieses Prinzip z. B. bei der Netzmaske, Tastenmaske oder beim Setzen von Parametern. Z.B. findet man mit dem bitweisen UND und der Linksschiebung leicht heraus, ob bei einer gegebenen (ganzen) Zahl n das Bit Nummer i gesetzt ist:

```
int mask = 1 << i;
int gesetzt = n & mask; // ergibt genau das i-te Bit von n
```

Beispielsweise kann man für einen Punkt aus einer Bilddatei `file` (hier der Klasse `java.io.File`) leicht den RGB-Code gewinnen. Das ist ein **int**-Wert, in dessen erstem (rechten) Byte der Blauanteil des Punktes angegeben ist, im zweiten Byte der Grünanteil und im dritten der Rotanteil. Entsprechend kann man eine Bitmaske für den Grünanteil bilden, indem nur die Bits des zweiten Bytes gesetzt sind, also `0x0000FF00`.

```
int i=100, j=100, rot, gruen, blau;
java.awt.Image.BufferedImage image = javax.imageio.ImageIO.read( file );
int rgb = image.getRGB(i,j);
rot    = (rgb & 0x00FF0000) >> 16;
gruen  = (rgb & 0x0000FF00) >> 8;
blau   = (rgb & 0x000000FF);
```

Oder man könnte einen eingegebenen Lottotipp „6 aus 49“ in einer einzigen **long**-Variable speichern, denn da $49 < 64$, kommt man mit 6 bits zur Darstellung einer einzelnen Tippzahl aus, also für einen ganzen Tipp 36 bits — **long** speichert aber 64 bits.

```

1 import javax.swing.*;
2
3 /** Speichert 5 Zahlen <= 49 in einer long-Variable.*/
4 public class LottotippMitBitmaske {
5     public static void main(String[] args) {
6         String output = "";
7         long tipp = 0, bitmaske;
8
9         // Dialogfenster:
10        JTextField[] feld = {
11            new JTextField("7"), new JTextField("12"), new JTextField("27"),
12            new JTextField("34"), new JTextField("35"), new JTextField("49")
13        };
14        Object[] msg = {
15            "Geben Sie Ihren Tipp für 6 aus 49 ein:",
16            feld[0], feld[1], feld[2], feld[3], feld[4], feld[5]
17        };
18        int click = JOptionPane.showConfirmDialog(null, msg, "Eingabe", 2);
19
20        // Codieren der Eingaben in tipp:
21        for (int i = 0; i < feld.length; i++) {
22            // jede Tippzahl (von 1 bis 49) benötigt 6 bits:
23            tipp |= Long.parseLong(feld[i].getText()) << (6*i);
24        }
25
26        output += "Ihr Tipp: ";
27        // Decodieren der Eingaben in tipp mit Bitmaske:
28        for (int i = 0; i < feld.length; i++) {
29            bitmaske = 0x3fL << 6*i; // 0x3f = 63 = 6 bits; wichtig ist das L!
30            output += ((tipp & bitmaske) >> 6*i) + ", ";
31        }
32        output += " (codiert " + tipp + ", 0x3f = " + 0x3fL + ")";
33        System.out.println(output);
34    }
35 }

```

2.1.8 Bedingte Wertzuweisung mit dem Bedingungsoperator

Möchte man bedingungsabhängig einer Variable einen Wert zuweisen, so kann man in Java den Bedingungsoperator `(bedingung) ? wert1 : wert2;` verwenden. Beispielsweise kann man kurz zur Bestimmung des Minimums zweier Zahlen n und a schreiben:

```
int min = (n<a) ? n : a; (2.5)
```

Zu beachten ist, dass der Datentyp der Variablen auf der linken Seite kompatibel zu *beiden* Werten rechts vom Fragezeichen sind. So darf keine der beiden Variablen n oder a in unserem Beispiel vom Typ **double** sein.

2.2 Wiederholung durch Schleifen

Eine der wichtigsten Eigenschaften von Computern ist ihre Fähigkeit, sich wiederholende Tätigkeiten immer und immer wieder auszuführen. Zur Programmierung von Wiederholungen gibt es im Wesentlichen zwei Strukturen, die „Iteration“ durch Schleifen und die „Rekursion“ durch Selbstaufrufe. Auf

Rekursionen werden wir hier zunächst nicht eingehen, wir werden die für viele etwas leichter zu verstehenden Schleifenkonstrukte kennen lernen. Es gibt in den meisten Programmiersprachen drei Schleifenkonstrukte, die jeweils bestimmten Umständen angepasst sind, obwohl man im Prinzip mit einer einzigen auskommen kann. In Java ist diese allgemeine Schleifenstruktur die **while**-Schleife.

2.2.1 Die **while**-Schleife

Bei einer Wiederholungsstruktur oder *Schleife* wird eine bestimmte Abfolge von Anweisungen (der *Schleifenrumpf*) wiederholt, solange eine bestimmte Bedingung (die *Schleifenbedingung*) wahr ist. In Java wird eine Schleifenstruktur durch die folgende Syntax gegeben:

```
while ( Bedingung ) {  
    Anweisung i1;  
    ...  
    Anweisung in;  
}
```

Um das Prinzip der **while**-Schleife zu demonstrieren, betrachten wir den folgenden logischen Programmausschnitt („Pseudocode“), der einen Algorithmus zur Erledigung eines Einkaufs beschreibt:

```
while ( es existiert noch ein nichterledigter Eintrag auf der Einkaufsliste ) {  
    erledige den nächsten Eintrag der Einkaufsliste;  
}
```

Man macht sich nach kurzem Nachdenken klar, dass man mit dieser Schleife tatsächlich alle Einträge der Liste erledigt: Die Bedingung ist entweder wahr oder falsch. Ist sie wahr, so wird der nächste offene Eintrag erledigt, und die Bedingung wird erneut abgefragt. Irgendwann ist der letzte Eintrag erledigt - dann ist aber auch die Bedingung falsch! Die Schleife wird also nicht mehr durchlaufen, sie ist beendet.

Merkregel 6. *In einer Schleife sollte stets eine Anweisung ausgeführt werden, die dazu führt, dass die Schleifenbedingung (irgendwann) falsch wird. Ansonsten wird die Schleife nie beendet, man spricht von einer Endlosschleife. Außerdem: Niemals ein Semikolon direkt nach dem Wort **while**! (Einzige Ausnahme: *do/while*, s.u.)*

2.2.2 Die Applikation *Guthaben*

Die folgende Applikation verwaltet ein Guthaben von 100 €. Der Anwender wird solange aufgefordert, von seinem Guthaben abzuheben, bis es aufgebraucht ist.

```
import javax.swing.*;  
  
/** Verwaltet ein Guthaben von 100 €. */  
public class Guthaben {  
    public static void main( String[] args ) {  
        int guthaben = 100;  
        int betrag = 0;  
        String text = "";
```

```

while ( guthaben > 0 ) {
    text = "Ihr Guthaben: " + guthaben + " \u20AC";
    text += "\nAuszahlungsbetrag:";
    // Eingabeblock:
    JTextField[] feld = {new JTextField()};
    Object[] msg = {text, feld[0]};
    int click = JOptionPane.showConfirmDialog(null, msg, "Eingabe", 2);
    betrag = Integer.parseInt( feld[0].getText() );
    guthaben -= betrag;
}

text = "Ihr Guthaben ist aufgebraucht!";
text += "\nEs betr\u00E4gt nun " + guthaben + " \u20AC.";
JOptionPane.showMessageDialog(null, text);
}
}

```

In diesem Programm werden in der main-Methode zunächst die verwendeten Variablen deklariert (bis auf diejenigen des Eingabeblocks). Dabei ist die Variable guthaben der „Speicher“ für das aktuelle Guthaben, das mit 100 € initialisiert wird. Die beiden Variablen betrag und text werden später noch benötigt, sie werden zunächst mit 0 bzw. dem leeren String initialisiert.

Als nächstes wird eine **while**-Schleife aufgerufen. Sie wird solange ausgeführt, wie das Guthaben positiv ist. Das bedeutet, dass beim ersten Mal, wo das Guthaben durch die Initialisierung ja noch 100 € beträgt, der Schleifenrumpf ausgeführt wird. Innerhalb der Schleife wird zunächst die Variable text mit der Angabe des aktuellen Guthabens belegt, die dann als Informationszeile in dem Eingabeblock verwendet wird. Nachdem der Anwender dann eine Zahl eingegeben hat, wird diese zu einem **int**-Wert konvertiert und vom aktuellen Guthaben abgezogen. Sollte nun das Guthaben nicht aufgebraucht oder überzogen sein, so wird die Schleife erneut ausgeführt und der Anwender wieder zum Abheben aufgefordert. Nach Beendigung der Schleife wird schließlich eine kurze Meldung über das aktuelle Guthaben ausgegeben und das Programm beendet.

Beachtung verdient der Unicode **\u20AC** für das Euro-Zeichen € (je nach Editor kann es damit nämlich beim einfachen Eintippen Probleme geben.) Entsprechend ergibt **\u00E4** den Umlaut ‚ä‘. Vgl. <http://haegar.fh-swf.de/Applets/Unicode/index.html>

2.2.3 Umrechnung Dezimal- in Binärdarstellung

Es folgt eine weitere, etwas mathematischere Applikation, die eine **while**-Schleife verwendet. Hierbei geht es um die Umrechnung einer Zahl in Dezimaldarstellung in die Binärdarstellung.

```

import javax.swing.*;

/**
 * Berechnet zu einer eingegebenen positiven Dezimalzahl
 * die Binärdarstellung als String.
 */
public class DezimalNachBinaer {
    public static void main( String[] args ) {
        int z = 0; // Dezimalzahl
        int ziffer = 0; // Binärziffer
        String binaer = "";
        String ausgabe = ""; // Binärzahl
    }
}

```



```

// Eingabeblock:
JTextField[] feld = {new JTextField()};
Object[] msg = {"Geben Sie eine positive Zahl ein:", feld[0]};
int click = JOptionPane.showConfirmDialog(null, msg, "Eingabe", 2);
z = Integer.parseInt( feld[0].getText() );

while ( z > 0 ) {
    ziffer = z % 2;
    // Ziffer vor den bisher aufgebauten String setzen:
    binaer = ziffer + binaer;
    z /= 2;
}

ausgabe += feld[0].getText() + " lautet binär " + binaer;
JOptionPane.showMessageDialog(null, ausgabe);
}
}

```

Die Schleife besteht aus drei Schritten: Zunächst wird die Ziffer (entweder 0 oder 1) von dem aktuellen Wert der Zahl z bestimmt, danach wird diese Ziffer vor den aktuellen String `binaer` gesetzt, und schließlich wird z ganzzahlig durch 2 dividiert. Dies wird solange wiederholt, wie z positiv ist (oder anders ausgedrückt: solange *bis* $z = 0$ ist).

2.2.4 Die for-Schleife

Eine weitere Schleifenkonstruktion ist die **for**-Schleife oder *Zählschleife*. Sie eignet sich für Wiederholungen, die durch einen Index oder einen Zähler gesteuert werden und bei der bereits zum Schleifenbeginn klar ist, wie oft sie ausgeführt werden. Die Syntax der **for**-Schleife lautet

```

for ( int Zähler = Startwert; Zähler <= Maximum; Aktualisierung ) {
    Anweisungsblock;
}

```

Im Englischen beschreibt man den Anfangsteil in den runden Klammern kurz mit

```

for ( start; check; update ) {
    ...;
}

```

Als Beispiel betrachten wir die folgende Applikation, in der der Anwender fünf Zahlen (z.B. Daten aus einer Messreihe oder Noten aus Klausuren) eingeben kann und die deren Mittelwert berechnet und ausgibt. Nebenbei lernen wir, wie man **double**-Variablen initialisieren kann, Eingabefelder vorbelegt und Dezimalzahlen für die Ausgabe formatieren kann.

```

import javax.swing.*;
import java.text.DecimalFormat;

/**
 * Berechnet den Mittelwert 5 einzugebender Daten.
 */
public class Mittelwert {
    public static void main( String[] args ) {
        int max = 5; // maximale Anzahl Daten
        double mittelwert, wert = 0.0, gesamtsumme = .0; // (1)
        String ausgabe = "";
    }
}

```



```

// Eingabeblock:
JTextField[] feld = {
    new JTextField("0"), new JTextField("0"), new JTextField("0"),
    new JTextField("0"), new JTextField("0") // (2)
};
Object[] msg = {"Daten:", feld[0], feld[1], feld[2], feld[3], feld[4]};
int click = JOptionPane.showConfirmDialog(null, msg, "Eingabe", 2);

for ( int i = 0; i <= max - 1; i++ ) {
    // Konvertierung des nächsten Datenwerts von String nach double:
    wert = Double.parseDouble( feld[i].getText() ); // (3)
    // Addition des Datenwerts zur Gesamtsumme:
    gesamtsumme += wert;
}

// Berechnung des Mittelwerts:
mittelwert = gesamtsumme / max; // (4)
// Bestimmung des Ausgabeformats:
DecimalFormat zweiStellen = new DecimalFormat("#,##0.00"); // (5)
// Ausgabe des Ergebnisses:
ausgabe = "Mittelwert der Daten: ";
ausgabe += zweiStellen.format( mittelwert ); // (6)
JOptionPane.showMessageDialog(null, ausgabe);
}
}

```

Wie Sie sehen, findet das Hochzählen des Zählers (hier `i`) nicht wie bei der `while`-Schleife im Anweisungsblock statt, sondern bereits im Anfangsteil der Schleife.

Programmablauf

Zunächst werden in der `main`-Methode die Variablen initialisiert. In Anmerkung (1) sind zwei Arten angegeben, wie man Variablen `double`-Werte zuweisen kann, nämlich mit `wert = 0.0` oder auch mit `.0`; eine weitere wäre auch mit `0.0d`.

In Anmerkung (2) werden Textfelder erzeugt, die bei der Anzeige eine Vorbelegung haben, hier mit `"0"`. Erst nach OK-Klicken des Eingabefeldes wird dann die `for`-Schleife ausgeführt.

Innerhalb der `for`-Schleife werden die Werte der Textfelder in (3) nach `double` konvertiert und direkt auf die Variable `gesamtsumme` aufaddiert. Nach Beendigung der Schleife ist der Wert in `gesamtsumme` also die Gesamtsumme aller in die Textfelder eingegebenen Werte. Danach wird der Mittelwert in (4) berechnet.

Formatierung von Zahlen

In der Zeile von Anmerkung (5) wird das Ergebnis unserer Applikation mit Hilfe der Klasse `DecimalFormat` auf zwei Nachkommastellen, einem Tausendertrenner und mindestens einer Zahl vor dem Komma formatiert. In dem „Formatierungsmuster“ bedeutet `#`, dass an dieser Stelle eine Ziffer nur angezeigt wird, wenn sie ungleich 0 ist, während für 0 hier auch eine 0 angezeigt wird. Mit der Anweisung (6)

```
zweiStellen.format( x );
```

wird der `double`-Wert `x` mit zwei Nachkommastellen dargestellt; sollte er Tausenderstellen haben, so werden diese getrennt, also z.B. 12.123.123,00. Andererseits wird mindestens eine Stelle vor dem Komma angegeben, also 0,00023.

Etwas verwirrend ist, dass man bei der Formatierungsanweisung das Muster in der englischen Notation angeben muss, also das Komma als Tausendertrenner und den Punkt als Dezimaltrenner — bei der Ausgabe werden dann aber die Landeseinstellungen des Systems verwendet.

2.2.5 Die **do/while**-Schleife

Die **do/while**-Schleife ist der **while**-Schleife sehr ähnlich. Allerdings wird hier Schleifenbedingung *nach* Ausführung des Schleifenblocks geprüft.

Die Syntax lautet wie folgt.

```
do {  
    Anweisungsblock;  
} while ( Bedingung );
```

Merkregel 7. Bei einer **do/while**-Schleife wird der Schleifenrumpf stets mindestens einmal ausgeführt. Bei der Syntax ist zu beachten, dass nach der Schleifenbedingung ein Semikolon gesetzt wird.

Als Beispiel betrachten wir die folgende Applikation, die eine kleine Variation der „Guthabenverwaltung“ darstellt.

```
import javax.swing.*;  
/** Verwaltet ein Guthaben von 100 €.  
 */  
public class GuthabenDoWhile {  
    public static void main( String[] args ) {  
        double guthaben = 100.0, betrag = .0;    // (1)  
        String text = "";  
  
        do {  
            text = "Ihr Guthaben: " + guthaben + " \u20AC";  
            text += "\nAuszahlungsbetrag:";  
            // Eingabeblock:  
            JTextField[] feld = {new JTextField()};  
            Object[] msg = {text, feld[0]};  
            int click = JOptionPane.showConfirmDialog(null, msg, "Eingabe", 2);  
            betrag = Double.parseDouble( feld[0].getText() );  
            guthaben -= betrag;  
        } while ( guthaben > 0 );  
  
        text = "Ihr Guthaben ist aufgebraucht!";  
        text += "\nEs betr\u00E4gt nun " + guthaben + " \u20AC.";  
        JOptionPane.showMessageDialog(null, text);  
    }  
}
```

Hier wird der Schleifenrumpf, also die Frage nach der Auszahlung, mindestens einmal ausgeführt — sogar wenn das Guthaben anfangs nicht im Positiven wäre ...

2.2.6 Wann welche Schleife?

Obwohl wir mit unserem Beispielprogramm eine Schleife kennengelernt haben, die in allen drei möglichen Varianten **while**, **for** und **do/while** implementiert werden kann, ist die **while**-Konstruktion die allgemeinste der drei. Mit ihr kann jede Schleife formuliert werden.

Merkregel 8. Die **for**- und die **do/while**-Schleife sind jeweils ein Spezialfall der **while**-Schleife: Jede **for**-Schleife und jede **do/while**-Schleife kann durch eine **while**-Schleife ausgedrückt werden (umgekehrt aber nicht unbedingt).

Betrachten Sie also die **while**-Schleife als *die* eigentliche Schleifenstruktur; die beiden anderen sind Nebenstrukturen, die für gewisse spezielle Fälle eingesetzt werden können.

Schleife	Verwendung
while	prinzipiell alle Schleifen; insbesondere diejenigen, für die die Anzahl der Schleifendurchläufe beim Schleifenbeginn nicht bekannt ist, sondern sich dynamisch innerhalb der Schleife ergibt
for	Schleifen mit Laufindex, für die die Anzahl der Wiederholungen vor Schleifenbeginn bekannt ist
do/while	Schleifen, die <i>mindestens einmal</i> durchlaufen werden müssen und bei denen die Schleifenbedingung erst nach dem Schleifenrumpf geprüft werden soll.

Tabelle 2.4: Die Schleifenkonstruktionen von Java und ihre Verwendung

2.3 Statische Methoden

2.3.1 Mathematische Funktionen

Um zu verstehen, was eine Methode ist, muss man den Begriff der mathematischen *Funktion* kennen. Betrachten wir eine einfache mathematische Funktion, z.B.

$$f(x) = x^2.$$

Diese Funktion heißt f , sie hat eine Variable namens x , den „Parameter“ der Funktion, und sie hat auf der rechten Seite eine Art „Anweisungsteil“.

Was besagt diese Funktionsdefinition? Aufmerksam Lesenden wird ein Mangel an dieser Definition auffallen: Es ist kein *Definitionsbereich* angegeben, und ebenso kein *Wertebereich*! Wir wissen also gar nicht, was x eigentlich sein darf, und was f für Werte annimmt. Mathematisch korrekt geht man her, bezeichnet den Definitionsbereich mit D , den Wertebereich mit W und schreibt allgemein eine Funktion ausführlicher:

$$f : D \rightarrow W, \quad x \mapsto f(x). \quad (2.6)$$

D und W sind zwei Mengen. Unsere spezielle Funktion $f(x) = x^2$ ist z.B. definiert für $D = \mathbb{N}$ und $W = \mathbb{N}$,

$$f : \mathbb{N} \rightarrow \mathbb{N}, \quad x \mapsto f(x) = x^2. \quad (2.7)$$

Das ist eine mathematisch korrekte Funktionendefinition. Wir können nun für x natürliche Zahlen einsetzen:

$$f(1) = 1, \quad f(2) = 4, \quad f(3) = 9, \quad \dots$$

Was passiert bei einer Änderung des Definitionsbereichs D ? Wenn wir für die Variable x nun nicht nur natürliche Zahlen einsetzen wollen, sondern *reelle Zahlen* \mathbb{R} , also Kommazahlen wie 0,5 oder π , so müssen wir unsere Funktion erweitern zu

$$f : \mathbb{R} \rightarrow \mathbb{R}, \quad x \mapsto f(x) = x^2. \quad (2.8)$$

Setzen wir beispielsweise für $x \in \mathbb{R}$ die Werte $x = -1, 5, 0, 0,5, 1, 2$ ein, so erhalten wir:

$$f(-1,5) = 2,25, \quad f(0) = 0, \quad f(0,5) = 0,25, \quad f(1) = 1, \quad f(2) = 4.$$

Die Funktion $f : \mathbb{R} \rightarrow \mathbb{N}, x \mapsto f(x) = x^2$ ist nicht definiert! (Preisfrage: Warum?) Die Funktion $f : \mathbb{N} \rightarrow \mathbb{R}, x \mapsto f(x) = x^2$ dagegen ist sehr wohl definiert. (Warum das?) Ein anderes Beispiel für eine mathematisch korrekt definierte Funktion ist

$$f : \mathbb{R} \rightarrow [-1, 1], \quad x \mapsto f(x) = \sin x. \quad (2.9)$$

Hierbei ist $D = \mathbb{R}$ die Menge der reellen Zahlen und $W = [-1, 1]$ ist das reelle Intervall der Zahlen x mit $-1 \leq x \leq 1$.

Merkregel 9. Eine mathematische Funktion ist eine Abbildung f von einem Definitionsbereich D in einen Wertebereich $W \subset \mathbb{R}$, in Symbolen

$$f : D \rightarrow W, \quad x \mapsto f(x).$$

Hierbei wird jedem Wert $x \in D$ genau ein Wert $f(x) \in W$ zugeordnet. Der Anweisungsteil $f(x)$, der Definitionsbereich D und der Wertebereich W sind dabei nicht unabhängig voneinander.

Eine Funktion kann auch mehrere Parameter haben. Betrachten z.B. wir die folgende Definition:

$$f : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}, \quad x \mapsto f(x, y) = \sqrt{x^2 + y^2}. \quad (2.10)$$

Hier ist nun $D = \mathbb{R} \times \mathbb{R} = \mathbb{R}^2$, und $W = \mathbb{R}$. Mit ein wenig Mühe rechnet man direkt nach:

$$f(0, 0) = 0, \quad f(1, 2) = f(2, 1) = \sqrt{5}, \quad f(3, 4) = 5, \quad \text{usw.}$$

Merkregel 10. Der Definitionsbereich D kann auch mehrere Parameter erfordern, in Symbolen: $D \subset \mathbb{R}^n$, mit $n \in \mathbb{N}, n > 1$. Man spricht dann auch von einer „mehrdimensionalen Funktion“. Der Wertebereich einer reellwertigen Funktion allerdings kann nur eindimensional sein, $W \subset \mathbb{R}$. Man schreibt allgemein:

$$f : D \rightarrow W, \quad (x_1, \dots, x_n) \mapsto f(x_1, \dots, x_n).$$

Der Vollständigkeit sei angemerkt, dass es durchaus Funktionen geben kann, deren Wertebereich ebenfalls mehrdimensional ist, also $W \subset \mathbb{R}^m$. Solche Funktionen sind beispielsweise „Vektorfelder“, die in der Physik oder den Ingenieurwissenschaften eine wichtige Rolle spielen. Wir werden sie hier jedoch nicht näher betrachten.

2.3.2 Was ist eine Methode?

Eine Methode im Sinne der (objektorientierten) Programmierung ist eine Verallgemeinerung einer mathematischen Funktion: Einerseits können Definitions- und Wertebereich allgemeiner sein, es müssen nicht unbedingt Zahlen zu sein; sie können sogar leer sein. Andererseits kann der Anweisungsteil ein ganzer Block von Anweisungen sein, es braucht nicht nur eine mathematische Operation zu sein.

Merkregel 11. Eine Methode ist ein Block von Anweisungen, der gewisse Eingabeparameter als Input benötigt und nach Ausführungen der Anweisungen einen Wert zurückgibt. Dabei kann die Menge der Eingabeparameter oder die Rückgabe auch leer sein.

Eine Methode ist so etwas wie eine Maschine oder eine „Black Box“, die mit Eingabedaten (einer „Frage“) gefüttert wird, der Reihe nach vorgeschriebene Anweisungen ausführt und eine Rückgabe („Antwort“) als Ergebnis ausgibt, siehe 2.2.

Eine Methode besteht strenggenommen aus drei Teilen: Dem *Methodennamen*, der Liste der *Eingabeparameter* und dem *Methodenrumpf*. Der Methodenrumpf enthält einen Block von einzelnen Anweisungen, die von geschweiften Klammern $\{ \dots \}$ umschlossen sind. Hinter jede Anweisung schreibt man ein Semikolon.

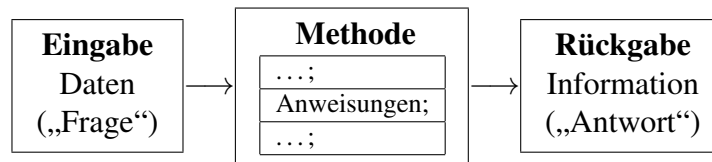
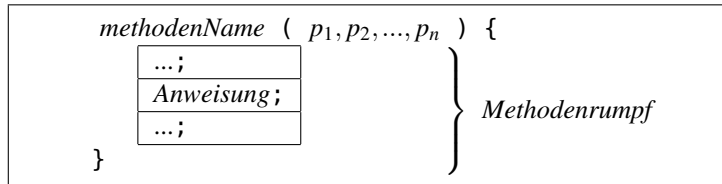


Abbildung 2.2: Schema einer Methode



Der Methodenname legt den Namen der Methode fest, hier beispielsweise *methodenName*; die Liste der Parameter besteht aus endlich vielen (nämlich n) Parametern p_1, \dots, p_n , die von runden Klammern umschlossen werden. Diese Parameter können prinzipiell beliebige Daten sein (Zahlwerte, Texte, Objekte). Wichtig ist nur, dass die Reihenfolge dieser Parameter in der Methodendeklaration bindend festgelegt wird.

Es kann Methoden geben, die gar keine Parameter haben. Dann ist also $n = 0$, und wir schreiben einfach

methodenName()

Der Methodenrumpf schließlich enthält die Anweisungen und Verarbeitungsschritte, die die Parameterdaten als Input verwenden. Er wird von geschweiften Klammern $\{ \dots \}$ umschlossen.

Beispiele

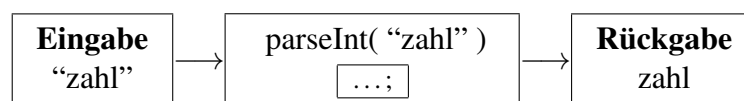
Wir haben bereits einige Methoden verwendet. Betrachten wir sie mit unseren neuen Erkenntnissen über Methoden.

- Die Methode `showMessageDialog` der Klasse `JOptionPane`. Sie erwartet 2 Eingabeparameter, die Konstante `null` und einen String. Sie liefert keinen Rückgabewert. (Daher ist die Methode eigentlich keine „Frage“, sondern eine „Bitte“).

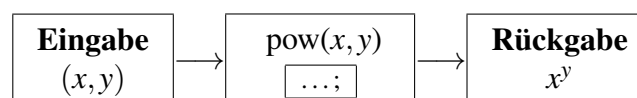


Die Punkte (...) deuten Anweisungen an, die wir nicht kennen (und auch nicht zu kennen brauchen).

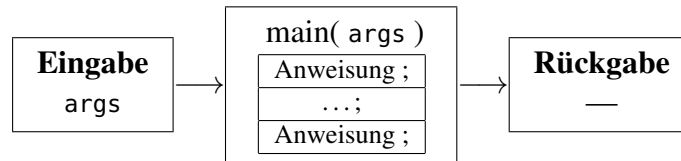
- Die Methode `parseInt` der Klasse `Integer`. Sie erwartet einen String als Eingabeparameter und gibt einen `int`-Wert zurück.



- Die Methode `pow` der Klasse `Math`. Sie erwartet zwei Parameter vom Typ `double` und gibt die Potenz als Rückgabe zurück.



- Die Methode `main`. Die haben wir bereits mehrfach selber programmiert. Es ist eine Methode, die einen Parameter (`args`) vom Datentyp `String[]` erwartet⁴ und liefert keinen Rückgabewert.



Merkregel 12. Eine Methode hat als Eingabe eine wohldefinierte Liste von Parametern mit festgelegter Reihenfolge und kann ein Ergebnis als Rückgabe liefern. Der Aufruf einer Methode bewirkt, dass ihr Algorithmus ausgeführt wird. Die Kombination

`<Rückgabetyyp> methodenName(<Parametertypen>)`

wird auch als **Signatur** der Methode bezeichnet.

Methodendeklarationen

Wir unterscheiden zwei Arten von Methoden: Diejenigen ohne Rückgabewert und diejenigen mit Rückgabewert. Genau wie Variablen müssen auch Methoden deklariert werden. Die Syntax lautet wie folgt.

Methode mit Rückgabe

```

public static <Rückgabetyyp> methodenName(<Datentyp> p1, ..., <Datentyp> pn) {
    ...;
    return Rückgabewert;
}
  
```

Der Datentyp *vor* dem Methodennamen gibt an, welchen Datentyp die Rückgabe der Methode hat. Entsprechend muss in der Eingabeliste jeder einzelne Parameter wie jede Variable deklariert werden. Zwischen zwei Parametern kommt stets ein Komma (,).

Eine Methode mit Rückgabewert muss am Ende stets eine Anweisung haben, die den Wert des Ergebnisses zurück gibt. Das geschieht mit dem reservierten Wort **return**. Nach der **return**-Anweisung wird die Methode sofort beendet.

Methode ohne Rückgabe

Soll eine Methode keine Rückgabe liefern, verwendet man das reservierte Wort **void** (engl. „leer“).

```

public static void methodenName(<Datentyp> p1, ..., <Datentyp> pn) {
    ...;
}
  
```

Hier wird keine **return**-Anweisung implementiert. (Man kann jedoch eine leere **return**-Anweisung schreiben, also `return;`.)

So deklarierte Methoden heißen *statische Methoden*. Fast alle Methoden, die wir bislang kennen gelernt haben, sind statisch: die `main()`-Methode, die Methode `showMessageDialog` der Klasse `JOptionPane`, sowie alle Methoden der Klassen `Math` und `System` sind statisch. (Wir werden später im Zusammenhang mit Objekten auch nichtstatische Methoden kennen lernen.)

⁴Das ist ein so genanntes „Array“ von Strings, s.u.

Das reservierte Wort **public** ist im Allgemeinen zwar nicht zwingend erforderlich für Methoden, es ermöglicht aber einen uneingeschränkten Zugriff von außen (siehe S. 92). In der Regel sind statische Methoden stets „öffentlich“.

Merkregel 13. Eine Methode ohne Rückgabewert muss mit **void** deklariert werden. Eine Methode mit Rückgabewert muss mit dem Datentyp des Rückgabewerts deklariert werden; sie muss als letzte Anweisung zwingend eine **return**-Anweisung haben.

Aufruf einer Methode

Alle Aktionen und Tätigkeiten des Softwaresystems werden in Methoden programmiert. Methoden werden während des Ablaufs eines Programms „aufgerufen“. Ein solcher Aufruf (*call*, *invocation*) geschieht einfach durch Angabe des Namens der Methode, mit Angabe der Eingabeparameter. Wir werden oft die Aufrufstruktur mit einem Diagramm wie in Abb. 2.3 darstellen.

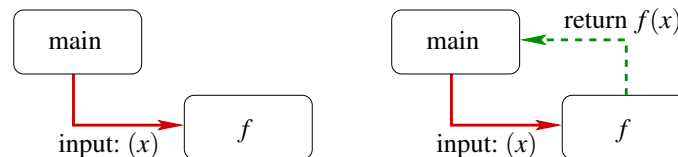


Abbildung 2.3: Diagramm eines Aufrufs einer Methode f mit Rückgabe, aus der main -Methode mit der Anweisung $y = f(x)$. Zunächst bewirkt der Aufruf mit dem Eingabeparameter x , dass der zur Laufzeit aktuelle Wert von x , das „Argument“ (siehe S. 58), in die Funktion f eingesetzt wird (linkes Bild). Während der Berechnungszeit wartet die main -Methode, erst wenn der Wert $f(x)$ zurück gegeben ist (rechtes Bild), werden weitere Anweisungen der main -Methode ausgeführt.

2.3.3 Eine erste Applikation mit statischen Methoden

Betrachten wir zunächst ein einfaches Beispiel zweier statischer Methoden, die in einer Applikation deklariert und aufgerufen werden.

```
import javax.swing.*;
/** Gibt das Quadrat einer ganzen Zahl und
 * die Wurzel der Quadratsumme zweier reeller Zahlen aus.*/
public class ErsteMethoden {

    public static int quadrat( int n ) {           // (1)
        int ergebnis;                             // (2)
        ergebnis = n*n;
        return ergebnis;                          // (3)
    }

    public static double hypotenuse( double x, double y ) { // (4)
        double ergebnis;                          // (5)
        ergebnis = Math.sqrt( x*x + y*y );
        return ergebnis;
    }

    public static void main( String[] args ) {
        String ausgabe = "";
        ausgabe += "5^2 = " + quadrat(5);           // (6)
        ausgabe += "\n 5 = " + hypotenuse(3.0, 4.0);
    }
}
```

```

        JOptionPane.showMessageDialog( null, ausgabe );
    }
}

```

Wo werden Methoden deklariert?

Bei einer ersten Betrachtung der Applikation `ErsteMethoden` fällt auf, dass die Deklaration der Methoden nicht innerhalb der `main`-Methode geschieht, sondern direkt in der Klasse auf gleicher Ebene wie sie.

Merkregel 14. *Methoden werden stets direkt in Klassen deklariert, niemals in anderen Methoden. Die Reihenfolge der Deklarationen mehrerer Methoden in einer Klasse ist beliebig.*

Man verwendet zum besseren Überblick über eine Klasse und die in ihr deklarierten Methoden oft ein Klassendiagramm, in der im untersten von drei Kästchen die Methoden aufgelistet sind:

ErsteMethoden
<code>int</code> <code>quadrat(int)</code> <code>double</code> <code>hypothense(double, double)</code>

Zur Verdeutlichung setzen wir, ähnlich wie bei der „echten“ Deklaration im Quelltext, den Datentyp der Rückgabe vor den Methodennamen und in Klammern die Datentypen der einzelnen Parameter (aber ohne die Parameternamen).

In der Zeile von Anmerkung (1) beginnt die Deklaration der Methode `quadrat`. Wir erkennen sofort den typischen Aufbau einer Methode mit Rückgabewert. Wie wir es schon aus den Implementierungen der `main`-Methode kennen, werden zunächst Variablen deklariert (2), und dann Anweisungen ausgeführt. In Anmerkung (3) wird der berechnete Wert von `ergebnis` als Rückgabewert an den aufrufenden Prozess (hier: die `main`-Methode) zurück gegeben. Beachten Sie, dass der Datentyp des Rückgabewerts hinter dem Wort `return` exakt zu dem Datentyp passen muss, der vor der Methode steht:

```

static int quadrat( int n ) {
    ...;
    return ergebnis;
}

```

Aufruf statischer Methoden

Eine statische Methode wird unter Angabe des Namens der Klasse, in der sie deklariert ist, aufgerufen:

Klassenname.methodenname (...);

Statische Methoden heißen daher auch *Klassenmethoden*. Der Klassenname kann auch weggelassen werden, wenn die Methode in derselben Klasse deklariert ist, in der sie aufgerufen wird. Daher funktioniert der Aufruf der Methode `quadrat` in der Zeile von Anmerkung (6) und derjenige von `hypothense` in der Zeile danach.

Mit der kurzen Anweisung in (6) wird also die Methode `berechneQuadrat` mit dem Wert 5 aufgerufen, die das Eingabefenster anzeigt und den Wert des Quadrats von 5 an `main` zurück gibt. Der Aufruf einer Methode entspricht also dem „Einsetzen“ von Werten in mathematische Funktionen.

2.3.4 Statische Methoden: Die Applikation Pythagoras

In der folgenden Applikation sind zwei Methoden (neben der obligatorischen **main**-Methode) implementiert.

```
import javax.swing.*;
/** Liest 2 Zahlen ein und gibt die Wurzel ihrer Quadratsumme aus.*/
public class Pythagoras {

    public static JTextField[] einlesen(String text) {           // (1)
        // Eingabeblock:
        JTextField[] feld = { new JTextField(), new JTextField() };
        Object[] msg = {text, feld[0], feld[1]};
        int click = JOptionPane.showConfirmDialog(null, msg, "Eingabe", 2);
        return feld;                                           // (2)
    }

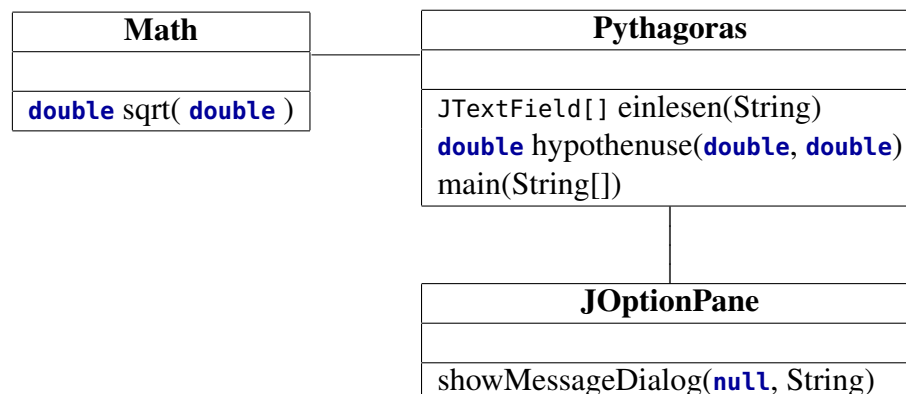
    public static double hypotenuse( double x, double y ) {    // (3)
        return Math.sqrt( x*x + y*y );                       // (4)
    }

    public static void main( String[] args ) {
        double a, b, c;
        JTextField[] feld = einlesen("Gib 2 Zahlen ein:");    // (5)
        a = Double.parseDouble( feld[0].getText() );
        b = Double.parseDouble( feld[1].getText() );
        c = hypotenuse( a, b );                                // (6)

        String ausgabe = "f( " + a + ", " + b + " ) = " + c;
        JOptionPane.showMessageDialog( null, ausgabe );
    }
}
```

Klassendiagramm

Bevor wir uns dem Ablauf des Programms widmen, betrachten wir die beteiligten Klassen des Programms und ihre Beziehung zueinander anhand eines Klassendiagramms.



Jeder Klasse sind ihre jeweiligen verwendeten Methoden zugeordnet.

Grober Ablauf und Methodenaufrufe

Betrachten wir zunächst die **main**-Methode. In (5) wird die erste Anweisung ausgeführt, es wird die Methode `einlesen` aufgerufen. Ohne deren Deklaration zu kennen kann man dennoch sehen, dass die Methode keinen Eingabeparameter erwartet und einen (komplexen) Datentyp `JTextField[]` zurückgibt, der den Inhalt mehrerer Textfelder enthält. Diese Werte werden in der Variablen `feld` gespeichert.

Danach werden die **double**-Variablen *a* und *b* mit den konvertierten Werten aus den Textfeldern beschickt.

Entsprechend sieht man in (6), dass *hypothenuse* eine Funktion ist, die mit den beiden eingelesenen **double**-Zahlen *a* und *b* aufgerufen wird und einen **double**-Wert zurück gibt. Die beiden Eingaben und das Ergebnis der Methode *hypothenuse* werden dann ausgegeben.

Die Methoden *einlesen* und *hypothenuse*

In der Zeile von Anmerkung (1) beginnt die Deklaration der Methode *einlesen*. Wir erkennen sofort den typischen Aufbau einer Methode mit Rückgabewert. Wie wir es schon aus den Implementierungen der *main*-Methode kennen, werden zunächst Variablen deklariert (hier: *feld* und *msg*) und dann Anweisungen ausgeführt (hier nur eine: der Eingabeblock, d.h. ein Eingabefenster anzeigen). In Anmerkung (2) werden die Inhalte der beiden Textfelder als Rückgabewert an den aufrufenden Prozess (hier: die *main*-Methode) zurück gegeben. Auch hier passt der Datentyp des Rückgabewerts hinter dem Wort **return** exakt zu dem Datentyp, der vor der Methode steht:

```
static JTextField[] einlesen(String text) {  
    ...;  
    return feld;  
}
```

Mit der kurzen Anweisung in (5) wird also die Methode *einlesen* aufgerufen, die das Eingabefenster anzeigt und die Inhalte der Textfelder an *main* zurück gibt. Wie in Abb. 2.4 dargestellt, ist die

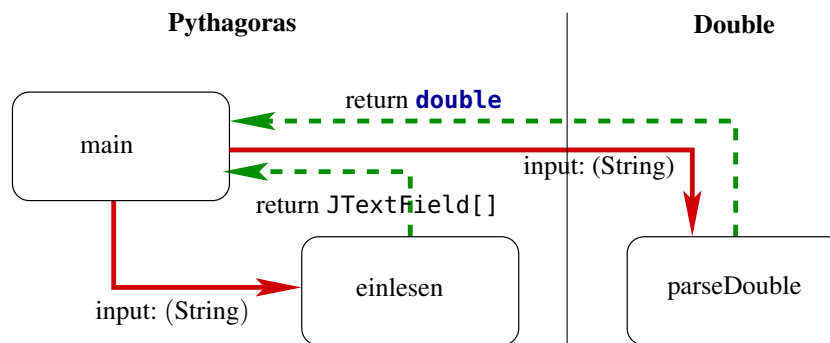


Abbildung 2.4: Aufrufstruktur der Anweisungen *feld = einlesen(...)*; und *a = Double.parseDouble(...)* in der Applikation *Pythagoras*, Anmerkung (5). Über den Methoden sind die Klassen aufgeführt, zu denen sie jeweils gehören.

main-Methode der Startpunkt, in dem *einlesen* mit dem String *text* als Eingabeparameter aufgerufen wird; da *einlesen* als *JTextField[]* deklariert ist (Anmerkung (1)), wartet die *main*-Methode auf ihren Rückgabewert. *einlesen* baut ein Eingabefenster auf. Erst wenn der Anwender seine Eingaben abgeschickt hat, gibt *einlesen* die Textfelder an *main* zurück, und die nächsten Anweisung in der *main*-Methode können ausgeführt werden.

Entsprechendes gilt für die Aufrufstruktur der Anweisung in (6), siehe Abb. 2.5. Hier wird die Methode *hypothenuse* mit zwei Werten (*a, b*) aufgerufen. Mit der Deklaration in (3) heißen diese beiden Parameter innerhalb der Methode *hypothenuse* nun jedoch *x* und *y*! Hier werden als die Werte der Variablen (*a, b*) für die Variablen (*x, y*) eingesetzt. Allgemein werden in der Informatik die Variablen bei der Deklaration einer Methode *Parameter* genannt, während Variablen oder auch konkrete Werte beim Aufruf *Argumente* heißen. In obigem Programm sind *x* und *y* also Parameter, *a* und *b* dagegen Argumente. (Das ist ein ganz wichtiger Punkt! Wenn Sie dies nicht ganz verstehen, so sehen Sie sich noch einmal den Abschnitt 2.3.1 ab Seite 51 über Funktionen und Variablen an.)

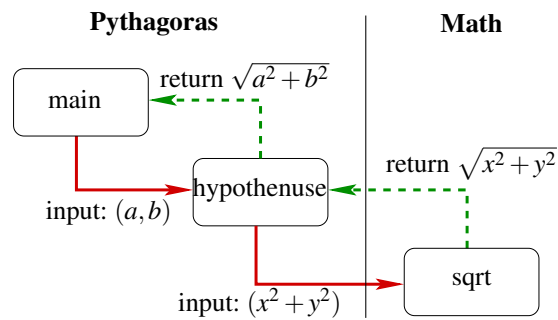


Abbildung 2.5: Aufrufstruktur bei der Anweisung `c = hypethenuse(a,b)` in der Applikation Pythagoras, Anmerkung (6).

Lokale Variablen

Sieht man sich den Methodenaufruf in Anmerkung (6) an, so könnte man auf die Frage kommen: Kann der Aufruf `hypotenuse(a,b)` überhaupt funktionieren? Denn die *Deklaration* der Methode in Anmerkung (3) hat doch die Variablennamen `x` und `y` verwendet, muss man daher nicht statt `a` und `b` eben `x` und `y` schreiben?

Die Antwort ist: Die Namen der Variablen bei Deklaration und Aufruf einer Methode sind vollkommen unabhängig. Entscheidend bei einem Methodenaufruf ist, dass Anzahl, Datentyp und Reihenfolge der Parameter übereinstimmen. Da bei der Deklaration von `hypotenuse` der erste Parameter (`x`) vom Typ `double` ist, muss es der erste Parameter beim Aufruf (`a`) auch sein, und entsprechend müssen die Datentypen von `y` und `b` übereinstimmen.

Logisch gesehen entspricht die Deklaration einer Methode der Definition einer mathematischen Funktion $f(x)$ mit Definitionsbereich (\leftrightarrow Eingabeparameter mit Datentypen) und Wertebereich (\leftrightarrow Datentyp des Rückgabewerts) sowie der Funktionsvorschrift, z.B. $f(x) = x^2$ (\leftrightarrow Algorithmus in der Methode). Daher entspricht der Aufruf einer Methode dem Einsetzen *eines Wertes* in die Funktion. Ein übergebener Wert wird auch *Argument* (siehe S. 58) genannt. Man ruft Methoden also stets mit *Werten* (der richtigen Datentypen) auf, nicht mit „Variablen“ oder „Parametern“.

Wichtig ist in diesem Zusammenhang, dass alle Variablen des Programms *lokale Variablen* sind: Sie gelten nur innerhalb derjenigen Methode, in der sie deklariert sind. So ist beispielsweise der Gültigkeitsbereich von `x` und `y` ausschließlich die Methode `hypotenuse`, und der für `a`, `b` und `c` ausschließlich die Methode `main`.

Und was ist mit der Variablen `feld`? Sie ist zweimal deklariert, in der Methode `einlesen` und in `main`. Auch hier gilt: ihr Gültigkeitsbereich ist auf ihre jeweilige Methode beschränkt, sie „wissen“ gar nichts voneinander. Im Gegensatz dazu wäre eine Deklaration zweier gleichnamiger Variablen innerhalb einer Methode nicht möglich und würde zu einem Kompilierfehler führen.

2.4 Arrays

Arrays sind eine der grundlegenden Objekte des wichtigen Gebiets der so genannten „Datenstrukturen“, das sich mit der Frage beschäftigt, wie für einen bestimmten Zweck mehr oder weniger komplexe Daten gespeichert werden können. Die Definition eines Arrays und seine Erzeugung in Java sind Gegenstand der folgenden Abschnitte.

2.4.1 Was ist ein Array?

Ein *Array*⁵ (auf Deutsch manchmal: *Reihung* oder *Feld*) ist eine Ansammlung von Datenelementen desselben Typs, die man unter demselben Namen mit einem so genannten *Index* ansprechen kann.

⁵ *array* – engl. für: Anordnung, Aufstellung, (Schlacht-)Reihe, Aufgebot

Dieser Index wird manchmal auch „Adresse“ oder „Zeiger“ (engl. *pointer*) des Elements in dem Array bezeichnet.

Stellen Sie sich z.B. vor, Sie müssten die Zu- und Abgänge eines Lagers für jeden Werktag speichern. Mit den Techniken, die wir bisher kennengelernt haben, hätten Sie nur die Möglichkeit, eine Variable für jeden Tag anzulegen: zugang1, zugang2, usw. Abgesehen von der aufwendigen Schreibarbeit wäre so etwas sehr unhandlich für weitere Verarbeitungen, wenn man beispielsweise den Bestand als die Summe der Zugänge der Woche wollte:

$$\text{bestand} = \text{zugang1} + \text{zugang2} + \dots + \text{zugang5};$$

Falls man dies für den Jahresbestand tun wollte, hätte man einiges zu tippen ...

Die Lösung für solche Problemstellungen sind Arrays. Wir können uns ein Array als eine Liste nummerierter Kästchen vorstellen, so dass das Datenelement Nummer i sich in Kästchen Nummer i befindet.

array	E	i	n	_	A	r	r	a	y
	↑	↑	↑	↑	↑	↑	↑	↑	↑
Index	0	1	2	3	4	5	6	7	8

Hier haben wir Daten des Typs **char**, und jedes Datenelement kann über einen Index angesprochen werden. So ist beispielsweise **'A'** über den Index 4 zugreifbar, also hier `array[4]`. Zu beachten ist: Man fängt in einem Array stets bei 0 an zu zählen, d.h. das erste Kästchen trägt die Nummer 0!

Merkregel 15. In Java beginnen die Indizes eines Arrays stets mit 0.

Die folgende Illustration zeigt ein Array namens `a` mit 10 Integer-Elementen. Das können z.B. die jeweilige Stückzahl von 10 Aktien sein, die sich im gesamten Portfolio `a` befinden.

11	3	23	4	5	73	1	12	19	41
<code>a[0]</code>	<code>a[1]</code>	<code>a[2]</code>	<code>a[3]</code>	<code>a[4]</code>	<code>a[5]</code>	<code>a[6]</code>	<code>a[7]</code>	<code>a[8]</code>	<code>a[9]</code>

Ein Array ist eine Datenstruktur mit so genanntem *Direktzugriff* (engl. *random access*): Auf alle Komponenten kann direkt und gleichschnell zugegriffen werden. In Java wird ein Array, z.B. `zugang`, wie folgt deklariert:

double[] zugang; oder **double** zugang[];

Die eckigen Klammern weisen darauf hin, dass es sich um ein Array handelt. (Es ist auch möglich, das Array mit **double**[] zugang zu deklarieren.) In Java sind Arrays Objekte. Sie müssen mit dem **new**-Operator erzeugt (instanciiert) werden, wie in der folgenden Zeile dargestellt:

zugang = **new** double[5];

Bei der Erzeugung eines Arrays muss also auch gleich seine Größe angegeben werden, also die Anzahl der Elemente des Arrays. Dieser Wert kann sich für das erzeugte Objekt nicht mehr ändern. Für unser Array `zugang` sind es also fünf Elemente vom Typ **double**. Man kann die beiden Zeilen auch in eine zusammenfassen:

double[] zugang = **new** double[5];

Um im Programm nun auf die einzelnen Elemente zuzugreifen, setzt man den Index in eckigen Klammern hinter den Namen der Variablen. Der Zugang des vierten Tages wird also mit

zugang[3] = 23.2;

angesprochen. Doch halt! Wieso [3], es ist doch der vierte Tag? (Nein, das ist kein Druckfehler!)

Ein Array braucht als Elemente nicht nur elementare Datentypen zu haben, es können allgemein Objekte sein, d.h. komplexe Datentypen. Allgemein wird ein Array *a* von Elementen des Typs *<Typ>* durch

$$\text{<Typ>}[] \text{ a}; \quad \text{oder} \quad \text{<Typ>} \text{ a}[];$$

deklariert. Bei der Erzeugung eines Array-Objekts muss seine Länge angegeben werden. Die Erzeugung eines Array-Objekts der Länge *länge* geschieht durch

$$\text{a} = \text{new } \text{<Typ>} \text{ [länge];}$$

länge muss hierbei ein **int**-Wert sein. Alternativ kann direkt bei der Deklaration eines Arrays die Erzeugung durch geschweifte Klammern und eine Auflistung der Elementwerte geschehen:

$$\text{<Typ>}[] \text{ a} = \{\text{wert}_0, \dots, \text{wert}_n\}; \quad \text{oder} \quad \text{<Typ>}[] \text{ a} = \text{new } \text{<Typ>}[] \{\text{wert}_0, \dots, \text{wert}_n\}; \quad (2.11)$$

Wir haben diese Konstruktion sogar bereits kennen gelernt: unser Eingabeblock besteht aus einem Array von JTextFields und Objects, die so erzeugt wurden.

2.4.2 Lagerbestände

Das folgende Beispielprogramm berechnet nach Eingabe von Zu- und Abgängen den aktuellen Lagerbestand.

```
import javax.swing.*;

/**
 * berechnet den Lagersbestand nach Eingabe von Zu- und Abgängen.
 */
public class Lagerbestand {
    public static final int ANZAHL_TAGE = 5; // (1)
    public static void main( String[] args ) {
        String ausgabe = "";
        double[] zugang = new double[ ANZAHL_TAGE ];
        double bestand = 0;

        // Eingabeblock:
        JTextField[] feld = { new JTextField(), new JTextField(),
                               new JTextField(), new JTextField(), new JTextField() };
        Object[] msg = {"Zu-/ Abgänge:", feld[0], feld[1], feld[2], feld[3], feld[4]};
        int click = JOptionPane.showConfirmDialog( null, msg, "Eingabe", 2);
        // aktuellen Bestand berechnen:
        for ( int i = 0; i < zugang.length; i++ ) { // (2)
            zugang[i] = Double.parseDouble( feld[i].getText() );
            bestand += zugang[i];
        }

        // Ausgabe der Arraywerte:
        ausgabe = "Tag\tZu-/ Abgang\n";
        for ( int i = 0; i < zugang.length; i++ ) {
            ausgabe += i + "\t" + zugang[i] + "\n";
        }
        ausgabe += "---\t" + "---\n";
        ausgabe += "Bestand\t" + bestand;

        // Ausgabebereich erstellen:
        JTextArea ausgabeBereich = new JTextArea(ANZAHL_TAGE + 3, 10); // (3)
        JScrollPane scroller = new JScrollPane(ausgabeBereich); // (4)
        ausgabeBereich.setText( ausgabe ); // (5)
    }
}
```

```

        JOptionPane.showMessageDialog(
            null, scroller, "Bestand", JOptionPane.PLAIN_MESSAGE    // (6)
        );
    }
}

```

Konstanten in Java

Das Schlüsselwort **final** in Anmerkung (1) indiziert, dass die deklarierte Variable

ANZAHL_TAGE

eine Konstante ist: Sie wird bei ihrer Deklaration initialisiert und kann danach nicht wieder verändert werden. Manchmal werden Konstanten auch „read-only“-Variablen genannt. Der Versuch, eine **final**-Variable zu verändern, resultiert in einer Fehlermeldung des Compilers.

Da eine Konstante allgemein gelten soll, insbesondere nicht nur in einer einzelnen Methode, wird sie üblicherweise als statische (also objektunabhängige) Variable direkt in einer Klasse deklariert, also außerhalb der Methoden. Auf diese Weise ist eine Konstante üblicherweise eine *Klassenvariable* und ihr Gültigkeitsbereich ist die gesamte Klasse und ihre Methoden, im Gegensatz zu den lokalen Variablen (mit Deklarationen *in* Methoden).

Array-Grenzen und das Attribut length

So praktisch der Einsatz von Array ist, so sehr muss man aufpassen, keinen Laufzeitfehler zu verursachen: denn falls ein Index zu groß oder zu klein gewählt ist, so stürzt das Programm ab und wird beendet, es erscheint eine Fehlermeldung der Art

java.lang.ArrayIndexOutOfBoundsException: 5

Einen solchen Fehler kann der Compiler nicht erkennen, erst zur Laufzeit kann festgestellt werden, dass ein Index nicht im gültigen Bereich ist.

Die zur Laufzeit aktuelle Größe eines Arrays `array` ermittelt man mit der Anweisung

`array.length`

`length` ist ein Attribut des Objekts `array`. Mit `zugang.length` in Anmerkung (2) wissen wir also, wieviel Zugänge in dem Array gespeichert sind. Der Index `i` der **for**-Schleife in Anmerkung (2) läuft also von 0 bis 4 und berechnet jeden einzelnen Array-Eintrag.

Will man allgemein die Einträge eines Arrays `array` bearbeiten, so sollte man die **for**-Schleife verwenden und den Laufindex mit `array.length` begrenzen:

```

for ( int i = 0; i < array.length; i++ ) {
    ... Anweisungen ...
}

```

Auf diese Weise ist garantiert, dass alle Einträge von `array` ohne Programmabsturz durchlaufen werden. Man sollte sich daher diese Konstruktion angewöhnen und den maximalen Index stets mit `array.length` bestimmen.

Übrigens ist das Attribut `length` ein **final**-Attribut, es ist also für ein einmal erzeugtes Array eine Konstante und kann also vom Programmierer nachträglich nicht mehr verändert werden.

Ausgaben mit JTextArea und JScrollPane

In der Zeile von Anmerkung (3) wird ein Objekt `ausgabeBereich` der Klasse `TextArea` deklariert und sofort erzeugt. Die Klasse `TextArea` gehört zum Paket `javax.swing` und ist eine GUI-Komponente, die es ermöglicht, mehrere Zeilen Text auf dem Bildschirm auszugeben. Die Parameter in den Klammern geben vor, dass das Objekt `ausgabe` `ANZAHL_TAGE + 3` Zeilen und 10 Spalten Text hat.

In Anmerkung (4) wird ein Objekt `scroller` der Klasse `JScrollPane` instanziiert. Die Klasse `JScrollPane` aus dem `javax.swing`-Paket liefert eine GUI-Komponente mit Scroll-Funktionalität. Damit ist es möglich, durch den ausgegebenen Text zu scrollen, so dass man alle Zeilen lesen kann, auch wenn das Fenster zu klein ist. In Anmerkung (5) schließlich wird der String `ausgabe` mit der Methode `setText` in das Objekt `ausgabeBereich` gesetzt.

Wichtig ist, dass bei der Ausgabe nun nicht `ausgabeBereich` angegeben wird, sondern das Scrollpanel `scroller`. (Ersteres würde auch funktionieren, aber eben nicht scrollbar!)

2.5 Die for-each-Schleife

Um ein Array (und auch eine „Collection“, s.u.) elementweise zu durchlaufen, gibt es in Java eine Alternative zur **for**-Schleife, die „verbesserte“ *for-Schleife (enhanced for-loop)* oder „for-each-Schleife“. Ihre Syntax lautet:

```
for ( <Typ> element : array ) {  
    Anweisungsblock;  
}
```

Hierbei bezeichnet `<Typ>` den Datentyp der Elemente des Arrays `array`, also muss `array` vorher als `Typ[] array` deklariert worden sein. Diese Schleife liest man „für jeden `<Typ>` in `array`“ oder „for each `<Typ>` in `array`“.

Im Gegensatz zur normalen **for**-Schleife wird hier im Schleifenkopf nicht die Variable für den Index deklariert, sondern diejenige für die Werte der einzelnen Elemente. Die Variable wird automatisch mit dem ersten Element des Arrays initialisiert, bei jedem weiteren Schleifendurchlauf trägt sie jeweils den Wert des nächsten Elements. Die Schleife beendet sich automatisch, wenn alle Elemente des Arrays durchlaufen sind. Diese Schleife ist sehr praktisch, wenn man ein Array durchlaufen will und die Indexwerte nicht benötigt. Die zweite **for**-Schleife in der Applikation `Lagerbestand` könnte man also auch wie folgt schreiben:

```
for (double element : zugang) {  
    ausgabe += element + "\n";  
}
```

Entsprechend ergibt das eine Liste der Zugangszahlen, allerdings ohne den laufenden Index `i`.

Eine wichtige Einschränkung der Verwendungsmöglichkeiten ist die folgende Regel.

Merkregel 16. *In Java kann man sich mit der for-each-Schleife nur die aktuellen Werte eines Arrays ablesen, jedoch nicht verändern! Eine Modifikation von Arraywerten kann nur mit Hilfe des Indexes geschehen, also in der Regel mit der **for**-Schleife, oder durch eine Initialisierung wie in Gl. (2.11) auf S. 61.*

2.6 Mehrdimensionale Arrays

Mehrdimensionale Arrays sind Arrays in Arrays. Ein zweidimensionales Array kann man sich vorstellen als eine Tabelle oder Matrix, in der eine Spalte ein Array von Reihen ist, und eine Reihe wiederum ein Array von Zellen. Z.B. stellt das Array


```
String[][] tabelle = {
    {"A", "B", "C"},
    {"D", "E", "F"}
};
```

eine 3×2 -Tabelle dar:

j ↓			$\leftarrow i$
A	B	C	
D	E	F	

Auf einen bestimmten Wert des Arrays greift man entsprechend mit mehreren Indizes zu, also

`tabelle[i][j]`.

Hierbei bezeichnet i die Nummer der Zeile (beginnend bei 0!) und j die Spalte, wie in der Tabelle angedeutet. D.h. in unserem Beispiel:

`System.out.println(tabelle[0][2]);` ergibt *C*.

Man kann i und j auffassen als Koordinaten, die jeweils den Ort einer Datenzelle angeben, so ähnlich wie A-2 oder C-5 Koordinaten eines Schachbretts (oder des Spielfeldes von Schiffe-versenken oder einer Excel-Tabelle usw.) sind.

Entsprechend kann man höherdimensionale Arrays bilden. Ein dreidimensionales Array ist ein Array von Arrays von Arrays von Elementen; man kann es sich vorstellen als einen Quader von Würfeln als Zellen. Jede Zelle kann durch 3 Koordinaten angesprochen werden,

`quader[i][j][k]`

Zwar kann man sich Arrays mit mehr als 3 Dimensionen nicht mehr geometrisch vorstellen, aber formal lassen sie sich mit entsprechend aufbauen,⁶ man braucht dann genau so viele Indizes wie man Dimensionen hat.

2.7 Zusammenfassung

Logik und Verweigung

- Um den Programmablauf abhängig von einer Bedingung zu steuern, gibt es die Verzweigung (**if**-Anweisung, Selektion).
- Die **if**-Anweisung (Verzweigung) kann Anweisungsblöcke unter einer angebbaren Bedingung ausführen. Die Syntax lautet:

```
if ( Bedingung ) {
    Anweisung i1;
    ...
    Anweisung im;
} else {
    Anweisung e1;
    ...
    Anweisung em;
}
```

⁶Die Physiker kennen den Begriff des Tensors, der ist als mehrdimensionales Array darstellbar.

Falls die Bedingung wahr ist, wird **if**-Zweig ausgeführt, ansonsten der **else**-Zweig. Der **else**-Zweig kann weggelassen werden.

- Wir haben in diesem Kapitel Vergleichsoperatoren (<, >, <=, >=, ==, !=) kennen gelernt.
- Zu beachten ist der Unterschied zwischen einer Zuweisung ($x = 4711$) und einem Vergleich ($x == 4711$).
- Die Präzedenz der Vergleichsoperatoren ist stets geringer als diejenige des Zuweisungsoperators oder der numerischen Operatoren +, -, usw.
- Die logischen Operatoren && (UND), || (ODER) und ^ (XOR) sowie ihre „Short-Circuit-Varianten“ & und | werden verwendet, um Bedingungen zu verknüpfen. Man kann sie entweder über ihre Wahrheitstabellen (Tab. 2.1) definieren, oder in ihrer numerischen Variante über die Grundrechenarten (2.1).

Schleifen

- In Java gibt es drei Schleifenstrukturen. Ihre Syntax lautet jeweils:

```
while (Bedingung) {  
    Anweisungsblock;  
}
```

```
for (start; check; update) {  
    Anweisungsblock  
}
```

```
do {  
    Anweisungsblock;  
} while (Bedingung);
```

Statische Methoden

- Eine mathematische Funktion ist korrekt definiert mit ihrem Definitionsbereich D , ihrem „Anweisungsteil“ $f(x)$ und ihrem Wertebereich W . In Symbolen:

$$f : D \rightarrow W, \quad x \mapsto f(x).$$

Es gilt dann für die Variable $x \in D$ und für den Wert $f(x) \in W$.

- Der Definitionsbereich einer Funktion kann mehrdimensional sein, $D \subset \mathbb{R}^n$, $n \geq 1$. Der Wertebereich ist immer eine reelle Zahl, $W \subset \mathbb{R}$.
- Anweisungen können als Block zu einer Methode zusammengefasst werden. Eine Methode ist die objektorientierte Verallgemeinerung einer Funktion.
- Die Deklaration einer Methode besteht aus dem Methodennamen, der Parameterliste und dem Methodenrumpf:

```
public static Datentyp des Rückgabewerts methodName(Datentyp  $p_1$ , ..., Datentyp  $p_n$ ) {  
    Anweisungen;  
    [ return Rückgabewert; ]  
}
```

Die Parameterliste kann auch leer sein ($n = 0$). Der Rückgabewert kann leer sein, dann ist die Methode **void**. Ist er es nicht, so muss die letzte Anweisung der Methode eine **return**-Anweisung sein.

- Variablen, die in Methoden deklariert sind, gelten auch nur dort. Sie sind lokale Variablen. Daher ist die Namensgebung von Variablen in verschiedenen Methoden unabhängig voneinander, innerhalb einer Methode darf es jedoch keine Namensgleichheit von Variablen geben.

Arrays

- Ein Array ist eine indizierte Ansammlung von Daten gleichen Typs, die über einen Index („Adresse“, „Zeiger“) zugreifbar sind. Man kann es sich als eine Liste nummerierter Kästchen vorstellen, in der das Datenelement Nummer i sich in Kästchen Nummer i befindet.

11	3	23	4	5	73	1	12	19	41
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

Ein Array ist also eine Datenstruktur mit Direktzugriff (*random access*). In Java beginnen die Indizes eines Arrays mit 0.

- Ein Array a von Elementen des Typs *Datentyp* wird deklariert durch

Datentyp[] a ; oder *Datentyp* a [];

Bei der Erzeugung eines Array-Objekts muss seine Länge angegeben werden:

$a = \text{new } \text{Datentyp}[\text{länge}];$

Hierbei ist *länge* die Größe des Arrays, d.h. die Anzahl seiner Elemente. Alternativ kann direkt bei der Deklaration eines Arrays die Erzeugung durch geschweifte Klammern und eine Auflistung der Elementwerte geschehen:

Datentyp[] $a = \{\text{wert}_0, \text{wert}_1, \dots, \text{wert}_n\};$

- Will man die Einträge eines Arrays a sukzessive bearbeiten, so sollte man die for-Schleife verwenden und den Laufindex mit `array.length` begrenzen:

```
for ( int i = 0; i < a.length; i++ ) {  
    ... Anweisungen ...  
}
```

Will man dagegen die Elemente eines Arrays *array* vom Typ *Typ*[] durchlaufen und benötigt die Indizes nicht, so kann man die „for-each-Schleife“ (oder „verbesserte“ for-Schleife) verwenden:

```
for ( <Typ> element : array ) {  
    Anweisungsblock;  
}
```

- Man kann mehrdimensionale Arrays, also Arrays in Arrays in Arrays ... programmieren. Eine Tabelle ist als zweidimensionales Array darstellbar.

Konstanten

- Konstanten werden in Java durch das Schlüsselwort **final** deklariert,

final *Datentyp* *variable* = *wert*;

Der Wert einer **final**-Variable kann nicht mehr geändert werden. Üblicherweise werden Konstanten statisch als Klassenvariablen deklariert.

Weitere Ausgabemöglichkeit

- Mit den Anweisungen

```
JTextArea ausgabeBereich = new JTextArea(Zeilen, Spalten);  
JScrollPane scroller = new JScrollPane(ausgabeBereich);  
ausgabeBereich.setText( text );
```

kann man einen Textbereich erzeugen, der *Zeile* mal *Spalte* groß ist, bei der Ausgabe nötigenfalls von Scroll-Balken umrahmt ist und den (auch mehrzeiligen) String *text* enthält. Mit

```
JOptionPane.showMessageDialog(null,scroller,"Ausgabe",JOptionPane.PLAIN_MESSAGE);
```

wird dieser Textbereich ausgegeben.

Kapitel 3

Objektorientierte Programmierung

3.1 Die Grundidee der Objektorientierung

3.1.1 Was charakterisiert Objekte?

Die Welt, wie wir Menschen sie sehen, besteht aus Objekten. Das sind

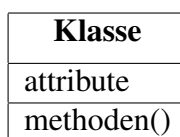
- Gegenstände, wie Autos, Schiffe und Flugzeuge,
- Lebewesen wie Pflanzen, Tiere oder Menschen,
- abstrakte Konstrukte wie Staaten, Absprachen, Konzerne oder Konten.

Ein Objekt kann z.B. ein konkretes Auto (der Beetle), eine individuelle Person (Hans Meier), die mathematische Zahl π oder ein Vertrag (Zahlungsvereinbarung zwischen Bank X und Person Y vom 19. Juli 2014) sein. Der VW Beetle ist als Objekt charakterisiert z.B. durch seine Farbe, sein Baujahr, die Anzahl seiner Sitzplätze und seine Höchstgeschwindigkeit, sowie seine Methoden beschleunigen, bremsen und hupen. Die Zahl π ist einfach durch ihren Wert charakterisiert, ihre Methoden sind die mathematischen Operationen $+$, $-$, \cdot , $/$.

Bei all diesen Objekten können wir also die folgenden drei Beobachtungen festhalten:

1. Wir versuchen ständig, die Objekte um uns herum zu *kategorisieren*, zu sinnhaften Einheiten zusammen zu fassen: *Begriffe* zu bilden. Täten wir das nicht, so hätten wir gar keine Möglichkeit, darüber zu sprechen. Wir sagen ja nicht (immer): „Das Objekt da gefällt mir!“ und zeigen darauf, sondern sagen: „Das *Haus* da gefällt mir.“ Wir haben aus dem Objekt die Kategorie oder den Begriff „Haus“ gemacht. Ähnlich denken wir zwar oft an das Individuum „Tom“, haben jedoch auch die Vorstellung, dass er eine „Person“ ist. (Im Grunde haben wir bereits in der Sekunde, in der wir den unbestimmten Artikel benutzen, eine Kategorisierung vorgenommen: Das ist *ein* Haus, Tom ist *eine* Person.)
2. Jedes Objekt ist *individuell*, es hat stets mindestens eine *Eigenschaft*, die es von anderen Objekten der gleichen Kategorie unterscheidet. Das konkrete Haus da unterscheidet sich (u.a.) von anderen Häusern dadurch, dass es die Hausnummer 182 hat. Ein Objekt „Haus“ hat also die Eigenschaft „Hausnummer“. Die Person heißt Tom, hat also die Eigenschaft „Name“.
3. Die Objekte verändern sich: Ein Lebewesen wird geboren und wächst, Tom reist von *A* nach *B*, der Saldo meines Kontos vermindert sich (leider viel zu oft). Objekte erfahren *Prozesse* oder führen sie selber aus. Genau genommen verändern diese Prozesse bestimmte Eigenschaften der Objekte (wie beispielsweise Alter, Größe, Aufenthaltsort, Saldo).

Die Theorie der Objektorientierung fasst diese drei Beobachtungen zu einem Konzept zusammen. Sie verwendet allerdings eigene Bezeichnungen: Statt von Kategorien spricht sie von *Klassen*¹, Eigenschaften heißen *Attribute*, und Prozesse werden von *Methoden* ausgeführt. Entsprechend wird ein Objekt durch das folgende Diagramm dargestellt:



(In der englischsprachigen Literatur sind im Zusammenhang mit Java auch die Begriffe *data* und *methods* üblich, siehe [3].)



Fassen wir ganz formal zusammen:

Definition 3.1. Ein *Objekt* im Sinne der Theorie der Objektorientierung ist die Darstellung eines individuellen Gegenstands oder Wesens (konkret oder abstrakt, real oder virtuell)² aus dem zu modellierenden *Problembereich* der realen Welt. Ein Objekt ist eindeutig bestimmt durch seine *Attribute* (Daten, Eigenschaften) und durch seine *Methoden* (Funktionen, Verhalten). □

3.1.2 Objekte als Instanzen ihrer Klasse

Wir haben festgestellt, dass eine Klasse ein Schema, ein Begriff, eine Kategorie oder eine Schablone ist, in welches man bestimmte individuelle Objekte einordnen kann. So ist „Tom“ *eine* Person, „Haldener Straße 182“ ist *ein* Haus. Oft kann man eine Klasse aber auch als eine Menge gleichartiger Objekte auffassen: In der (reellen) Mathematik ist eine Zahl ein Element der Menge \mathbb{R} , also ist π ein Objekt der Klasse \mathbb{R} . Oder ein Buch ist ein spezielles Exemplar des endlichen Buchbestandes der Bibliothek, d.h. „Handbuch der Java-Programmierung“ von G. Krüger ist ein Objekt der Klasse „Buchbestand der Hochschulbibliothek“.

Ein Objekt verhält sich zu seiner Klasse so etwa wie ein Haus zu seinem Bauplan. Ein Objekt ist immer eine konkrete Ausprägung einer Klasse. Man sagt auch, ein Objekt sei eine *Instanz* seiner Klasse. Beispielsweise ist π eine Instanz der (geeignet definierten) Klasse „Zahl“, oder Tom eine Instanz der Klasse „Person“.

3.1.3 Kapselung und Zugriff auf Attributwerte

Ein weiterer wesentlicher Aspekt der Objektorientierung ist, dass die Attribute von den Methoden vor der Außenwelt *gekapselt* sind. Das bedeutet, sie können (im allgemeinen) nur durch die Methoden verändert werden. Die Methoden (aber nicht unbedingt alle, es gibt auch „objektinterne“ Methoden) eines Objektes stellen die Schnittstellen des Objektes zur Außenwelt dar.

¹ Aristoteles war sicher der Erste, der das Konzept der Klasse gründlich untersucht hat: Er sprach von „der Klasse der Fische und der Klasse der Vögel“.

² Wenn es da ist und du es sehen kannst, ist es *real*. Wenn es da ist und du es nicht sehen kannst, ist es *transparent*. Wenn es nicht da ist und du es (trotzdem) siehst, ist es *virtuell*. Wenn es nicht da ist und du es nicht siehst, ist es *weg*. Roy Wilks

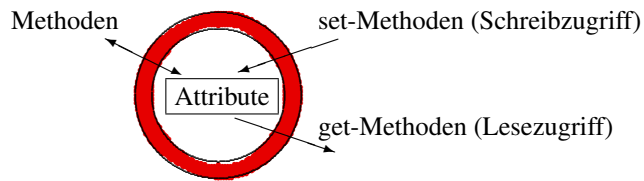


Abbildung 3.1: Objekte sind Einheiten aus Attributen und Methoden. Die Attribute sind gekapselt, sie können (i.a.) nur über die Methoden verändert werden.

Auf den ersten Blick erscheint die Kapselung von Attributen umständlich. Warum sollte man den Zugriff auf ihre Werte nicht direkt gestatten? Es gibt hauptsächlich zwei Gründe für das Paradigma der Kapselung, die sich vor allem in komplexen Softwaresystemen auswirken. Einerseits kann der Programmierer über die Methoden den Zugriff genau steuern; manche Attribute wie z.B. Passwörter soll ein Benutzer der Objekte vielleicht gar nicht sehen dürfen oder nicht verändern können, und so werden keine get- bzw. set-Methoden geschrieben.

Andererseits wird über die Methoden für spätere Anwender eine eindeutige und saubere Schnittstelle beschrieben, die es ihnen gestattet, die verwendeten Objekte als *Black Boxes* zu sehen, deren Inneres sie nicht zu kennen brauchen.

3.1.4 Abfragen und Verändern von Attributwerten: get und set

Es ist in der Objektorientierung allgemein üblich, den lesenden und schreibenden Zugriff auf Attributwerte über „get“- und „set“-Methoden zu ermöglichen. (Oft auch „getter“ und „setter“ genannt.) Die get-Methoden *lesen* dabei die Werte der Attribute, die set-Methoden dagegen *schreiben*, also verändern sie. Jedes einzelne Attribut hat dabei im Allgemeinen seine eigenen get- und set-Methoden.

Das Schema ist sehr einfach: Lautet der Name des Attributs `attribut`, so nennt man seine get-Methode einfach `getAttribut()`, und seine set-Methode `setAttribut(wert)`. Die get-Methode wird ohne Parameter aufgerufen und liefert den Wert des Attributs zurück, ihr Rückgabetyp ist also der Datentyp des Attributs. Umgekehrt muss der set-Methode als Parameter der Wert übergeben werden, den das Attribut nach der Änderung haben soll, sie gibt jedoch keinen Wert zurück, ist also **void**.

3.2 Klassendiagramme der UML

Zur Strukturierung und zur besseren Übersicht von Klassen und ihren Objekten verwendet man Klassendiagramme. Die hier gezeigten Klassendiagramme entsprechen der UML (*Unified Modeling Language*). UML ist eine recht strikt festgelegte Modellierungssprache, deren „Worte“ Diagramme sind, die wiederum aus wenigen Diagrammelementen („Buchstaben“) bestehen. Sie hat insgesamt neun Diagrammarten. Ihr Ziel ist die visuelle Modellierung von Strukturen und Prozessen von Software. Sie ist der Standard der modernen Software-Entwicklung. Mit UML wird es Personen, die keine oder wenig Programmierkenntnisse haben, ermöglicht, ihre Anforderungen an Software zu modellieren und zu spezifizieren.

Eines der Diagramme der UML ist das Klassendiagramm. Hier wird jede Klasse als ein Rechteck dargestellt, mit dem Namen der Klasse im oberen Teil, allen Attributen („Daten“) im mittleren Teil, und den Methoden im unteren Teil.

Person
String vorname String nachname double groesse
String getVorname() void setVorname(String name) String getNachname() void setNachname(String name) double getGroesse() void setGroesse(double meter) String toString()

Oben ist also eine Klasse Person modelliert, die aus drei Attributen nachname, vorname und groesse besteht und eine ganze Reihe eigener Methoden besitzt.

Durch ein Klassendiagramm wird also anschaulich dargestellt, aus welchen Attributen und Methoden ein Objekt der Klasse sich zusammen setzt. Wir werden sowohl bei dem Entwurf als auch bei der Beschreibung von Klassen stets das Klassendiagramm verwenden.

In unserer Beispielklasse Person ist neben den get- und set-Methoden für die drei Attribute noch eine spezielle Methode toString() vorgesehen. Sie dient dazu, die wesentlichen Attributwerte, die das Objekt spezifizieren, in einem lesbaren String zurück zu geben. Was „wesentlich“ ist, bestimmt dabei der Programmierer, wenn es aus den Zusammenhang heraus Sinn macht, kann er auch durchaus bestimmte Attributwerte gar nicht anzeigen, sie also verbergen.

3.3 Programmierung von Objekten

Hat man das Klassendiagramm, so ist die Implementierung eine (zunächst) einfache Sache, die nach einem einfachen Schema geschieht. Zunächst wird die Klasse nach dem Eintrag im obersten Kästchen des Klassendiagramms genannt, als nächstes werden die Attribute aus dem zweiten Kästchen deklariert (normalerweise ohne Initialisierung), und zum Schluss die Methoden. Sie bestehen aus jeweils nur einer Anweisung, get-Methoden aus einer **return**-Anweisung, die den aktuellen Wert des jeweiligen Attributs zurück gibt, set-Methoden aus einer, die den übergebenen Wert in das jeweilige Attribut speichert.

Die get- und set-Methoden werden normalerweise **public** deklariert, damit sie „von außen“ (eben öffentlich) zugänglich sind.

3.3.1 Der Konstruktor

Eine Besonderheit ist bei der Implementierung von Objekten ist der sogenannte *Konstruktor*. Ein Konstruktor wird ähnlich wie eine Methode deklariert und dient der Initialisierung eines Objektes mit all seinen Attributen. Er heißt genau wie die Klasse, zu der er gehört. Im Unterschied zu anderen Methoden wird bei ihnen *kein* Rückgabewert deklariert. Konstruktoren dürfen eine beliebige Anzahl an Parametern haben, und es kann mehrere geben, die sich in ihrer Parameterliste unterscheiden („überladen“). Ein Konstruktor hat also allgemein die Syntax:

```
public Klassenname ( Datentyp p1, ..., Datentyp pn ) {
    Anweisungen;
}
```

Bei einer **public** deklarierten Klasse sollte auch der Konstruktor **public** sein. Wird in einer Klasse kein Konstruktor explizit deklariert, so existiert automatisch der „Standardkonstruktor“ Klassenname(), der alle Attribute eines elementaren Datentyps auf 0 initialisiert (bzw. auf "" für **char** und **false** für **boolean**) und Attribute eines Objekttyps auf **null** (das „Nullobjekt“) gesetzt.

3.3.2 Die Referenz **this**

Das Schlüsselwort **this** dient dazu, auf das Objekt selbst zu verweisen (gewissermaßen bedeutet es „ich“ oder „mein“). Es ist also eine Referenz und daher von der Wirkung so wie eine Variable, die auf ein Objekt referenziert. Entsprechend kann man also auf eine eigene Methode oder ein eigenes Attribut mit der Syntax

this.methode() oder **this**.attribut

verweisen. Die **this**-Referenz kann im Grunde immer verwendet werden, wenn auf objekt eigene Attribute oder Methoden verwiesen werden soll. Praktischerweise lässt man sie jedoch meistens weg, wenn es keine Mehrdeutigkeiten gibt.

Manchmal allerdings gibt es Namenskonflikte mit gleichnamigen lokalen Variablen, so wie in dem Konstruktor oder den **set**-Methoden unten. Hier sind die Variablenname doppelt vergeben — einerseits als Attribute des Objekts, andererseits als Eingabeparameter des Konstruktors bzw. der Methoden, also als lokale Variablen. Damit der Interpreter genau weiß, welcher Wert nun welcher Variablen zuzuordnen ist, *muss* hier die **this**-Referenz gesetzt werden. (Übrigens: im Zweifel „gewinnt“ immer die lokale Variable gegenüber dem gleichnamigen Attribut, d.h. im Konstruktor wird ohne **this** stets auf die Eingabeparameter verwiesen.)

Nun könnte man einwenden, dass man einen Namenskonflikt doch vermeiden könnte, indem man die lokale Variable einfach anders nennt. Das ist selbstverständlich richtig. Es ist aber beliebte Praxis (und oft auch klarere Begriffsbezeichnung), gerade in **set**-Methoden und Konstruktoren die Eingabvariablen genau so zu nennen wie die Attribute, denen sie Werte übergeben sollen.

3.3.3 Typische Implementierung einer Klasse für Objekte

Insgesamt kann man aus unserem Klassendiagramm also die folgende Klasse implementieren:

```
/** Diese Klasse stellt eine Person dar.*/
public class Person {
    // -- Attribute: -----
    String vorname;
    String nachname;
    double groesse;

    // -- Konstruktor: -----
    /** Erzeugt eine Person.*/
    public Person(String vorname, String nachname, double groesse) {
        this.vorname = vorname;
        this.nachname = nachname;
        this.groesse = groesse;
    }

    // -- Methoden: -----
    /** Gibt den Vornamen dieser Person zurück.*/
    public String getVorname() {
        return vorname;
    }

    /** Gibt den Nachnamen dieser Person zurück.*/
    public String getNachname() {
        return nachname;
    }

    /** Verändert den Nachnamen dieser Person.*/
    public void setNachname(String nachname) {
        this.nachname = nachname;
    }
}
```



```

    /** Gibt die Größe dieser Person zurück.*/
    public double getGroesse() {
        return groesse;
    }
    /** Verändert die Größe dieser Person.*/
    public void setGroesse(double groesse) {
        this.groesse = groesse;
    }
    /** Gibt die Attributwerte dieser Person formatiert zurück.*/
    public String toString() {
        return (vorname + " " + nachname + ", " + groesse + " m");
    }
}

```

Da die Klasse öffentlich ist, müssen wir sie in einer Datei speichern, die ihren Namen trägt, also `Person.java`. In den meisten Fällen wird eine Klasse, aus der Objekte erzeugt werden, als **public** deklariert (vgl. §3.8, S. 91).

Man sollte der Klasse und ihren öffentlichen Attributen und Methoden jeweils einen javadoc-Kommentar `/**...*/` direkt voranstellen, der kurz deren Sinn erläutert. Mit dem Befehl `javadoc Person.java` wird dann eine HTML-Dokumentation erzeugt, in der diese Kommentare erscheinen.

Wann sollte man set- und get-Methoden schreiben und wann nicht? Es gibt kein ehernes Gesetz, wann set- und get-Methoden zu schreiben sind. Im Allgemeinen werden get-Methoden zu allen Attributen deklariert außer denen, die verborgen bleiben sollen (Passwörter usw.), und set-Methoden nur dann *nicht*, wenn man sich von vornherein sicher ist, dass der entsprechende Attributwert sich nach Erzeugung des Objekts nicht mehr ändern wird. In unserem obigen Beispiel gehen wir davon aus, dass sich der Vorname der Person während der Laufzeit des Programms nicht ändern wird, daher haben wir keine set-Methode vorgesehen. Demgegenüber kann sich der Nachname durch Heirat bzw. die Größe durch Wachstum ändern, also wurden ihre set-Methoden implementiert. Objekte aus Klassen, für die alle Attribute set- und get-Methoden besitzen, heißen in Java auch POJO (*plain old Java objects*).

3.3.4 Objektelemente sind nicht statisch

Wir haben in den vorangegangenen Kapiteln ausschließlich mit statischen Methoden gearbeitet. Das sind Methoden, die nicht einzelnen Objekten gehören und daher auch ohne sie verwendet werden, sondern allgemein der Klasse gehören. Aus diesem Grund haben sie auch Klassenmethoden.

Ganz anders verhält es sich bei Objektmethoden. Da jedes Objekt einer Klasse stets individuelle Werte für seine Attribute hat, benötigt es auch eigene Methoden, um die Werte zu lesen oder zu verändern. Diese Methoden sind *nicht statisch*, d.h. man lässt das Wörtchen **static** einfach weg. Eine solche Methode „gehört“ jedem einzelnen Objekt heißt daher auch *Objektmethode*.

Das Schlüsselwort **static** bei der Deklaration von Elementen einer Klasse, also Attribute und Methoden, bewirkt, dass sie direkt zugreifbar sind und kein Objekt benötigen, um verwendet werden zu können. Sie heißen daher auch *Klassenattribute* und *Klassenmethoden*. Daher konnten wir bislang auch ohne Kenntnis von Objekten Methoden programmieren.

Eine für Objekte allerdings nur selten gewollte Eigenschaft statischer Attribute ist es, dass es sie *nur einmal* im Speicher gibt. Damit ist der Wert eines statischen Attributs für alle Objekte der Klasse gleich. Mit anderen Worten, die Objekte teilen sich *dasselbe* Attribut.

Statische Methoden sind Methoden, die (neben lokalen Variablen) nur auf statische Attribute wirken können, also auf Klassenvariablen, von denen es zu einem Zeitpunkt stets nur einen Wert geben kann.

Methoden in Klassen, aus denen Objekte instanziiert werden, sind in der Regel *nicht* statisch. Der Grundgedanke der Objektorientierung ist es ja gerade, dass die Daten nur individuell für ein Objekt gelten.

Merkregel 17. Klassen, von denen Objekte instanziiert werden sollen, haben normalerweise keine statischen Attribute und Methoden. Generell kann von statischen Methoden nicht auf Objektattribute zugegriffen werden.

3.3.5 Lokale Variablen und Attribute

Ein Aufrufparameter in einer Methodendeklaration hat, genau wie eine innerhalb einer Methode deklarierte Variable, seinen Geltungsbereich nur innerhalb dieser Methode. Solche Variablen heißen *lokale Variable*. Lokale Variablen gelten („leben“) immer nur in der Methode, in der sie deklariert sind.

Demgegenüber sind Attribute *keine* lokalen Variablen. Sie sind Variablen, die in der gesamten Klasse bekannt sind. Deswegen gelten sie auch in jeder Methode, die in der Klasse deklariert ist. Eine Übersicht gibt Tabelle 3.1.

Variablenart	Ort der Deklaration	Gültigkeitsbereich
lokale Variable	in Methoden (also auch Eingabeparameter!)	nur innerhalb der jeweiligen Methode
Attribute	in der Klasse, außerhalb von Methoden	im gesamten erzeugten Objekt, also auch innerhalb aller Methoden
statisches Attribut (Klassenvariable)	außerhalb von Methoden	in der gesamten Klasse, ggf. für alle daraus erzeugten Objekte, insbesondere innerhalb aller Methoden der Klasse

Tabelle 3.1: Unterschiede lokaler Variablen und Attribute.

3.3.6 Speicherverwaltung der Virtuellen Maschine

Werte statischer Attribute, nichtstatischer Attribute und lokaler Variablen werden im Arbeitsspeicher der Virtuellen Maschine (JVM) jeweils in verschiedenen Bereichen gespeichert. Der Speicher ist aufgeteilt in die *Method Area*, den *Stack* und den *Heap*. In der Method Area werden die statischen Klassenvariablen und die Methoden (mit ihrem Bytecode) gespeichert. Der Stack ist ein „LIFO“-Speicher („last in, first out“). In ihm werden die Werte der lokalen Variablen der Methoden gespeichert. Auf diese Weise wird garantiert, dass stets nur eine lokale Variable zu einem bestimmten Zeitpunkt von dem Prozessor verwendet wird: Es ist die, die im Stack ganz oben liegt. Ist eine Methode verarbeitet, werden die Werte ihrer lokalen Variablen gelöscht (genauer: ihre Speicheradressen im Stack werden freigegeben).

Method Area		Stack	Heap
Methoden (Bytecode)	statische Variablen	Werte lokaler Variablen	Objekte Attribute

Abbildung 3.2: Der Speicheradressraum der JVM

Ebenso befinden sich statische Variablen in der Method Area. Genaugenommen existieren **static**-Elemente bereits bei Aufruf der Klasse, auch wenn noch gar kein Objekt erzeugt ist. Auf eine **static**-Variable einer Klasse ohne instanziiertes Objekt kann allerdings nur über eine **static**-Methode zugegriffen werden.

Die Objekte und ihre Attribute werden dagegen im Heap gespeichert. Der Heap (eigentlich: „garbage collected heap“) ist ein dynamischer Speicher.³ Für Details siehe [4, §18].

Merkregel 18. Da Objekte erst in der Methode **main** selbst angelegt werden, kann **main** nicht zu einem Objekt gehören. Das bedeutet, dass **main** stets eine Klassenmethode ist und als **static** deklariert werden muss. Ebenso muss sie **public** sein, damit der Java-Interpreter, der die Applikation starten und ausführen soll, auch auf sie zugreifen kann.

Meist werden Methoden als **static** deklariert, um sie anderen Klassen zur Verfügung zu stellen, ohne Objekte der betreffenden Klasse zu erzeugen. Wir haben bereits in Abschnitt 2.3.2 (S. 54) solche Methoden deklariert.

3.4 Erzeugen von Objekten

Die im vorigen Abschnitt implementierte Klasse `Person` ist zwar ein kompilierbares Programm — aber keine lauffähige Applikation! Denn es fehlt die `main`-Methode. Wir werden nun eine Applikation in einer eigenen Datei programmieren, die im selben Verzeichnis wie die Datei `Person.java` steht.

```
/** Erzeugt und verändert Objekte der Klasse Person.*/
public class PersonApp {
    public static void main( String[] args) {
        String ausgabe = "";
        Person tom = new Person("Tom", "Katz", 1.82);           // (1)
        Person jerry;                                           // (2)
        jerry = new Person("Jerry", "Maus", 1.56);              // (3)
        ausgabe += tom + '\n' + jerry;

        jerry.setGroesse(1.46);

        ausgabe += '\nJerry ist jetzt ' + jerry.getGroesse() + " m groß.";
        javax.swing.JOptionPane.showMessageDialog(null, ausgabe);
    }
}
```

Die Ausgabe dieser Applikation ist in Abb. 3.3 dargestellt.



Abbildung 3.3: Die Ausgabe der Applikation `PersonApp`.

3.4.1 Deklaration von Objekten

Technisch gesehen ist eine Klasse der Datentyp für ihre Objekte, der sich aus den einzelnen Datentypen der Attribute zusammen setzt. Man bezeichnet daher eine Klasse auch als *komplexen Datentyp*. Ein Objekt kann daher aufgefasst werden als ein Speicherplatz für einen komplexen Datentyp. Eine Variable einer Klasse ist ein bestimmter Speicherplatz im RAM des Computers. Sie wird deklariert, indem vor ihrem ersten Auftreten ihr Datentyp erscheint, egal ob dieser komplex oder primitiv ist:

³Leider hat der Begriff „Heap“ in der Informatik auch die Bedeutung eines auf gewisse Weise geordneten logischen Baumes; der Heap in der Virtuellen Maschine ist damit jedoch nicht zu verwechseln!

```
double zahl;  
Klassenname variable;
```

In Anmerkungen (1) und (2) der Applikation PersonApp werden zwei Objekte tom und jerry der Klasse Person deklariert. Beachten Sie: Wie üblich werden Variablennamen von Objekten klein geschrieben, ihre Klassen groß!

3.4.2 Erzeugung von Objekten: der **new**-Operator

Der Initialisierung einer Variablen von einem primitiven Datentyp entspricht die *Erzeugung* eines Objekts. Sie geschieht wie in Anmerkung (3) mit dem **new**-Operator und der Zuordnung der Variablen:

```
variable = new Klassenname();
```

Erst der **new**-Operator erzeugt oder „instanziert“ ein Objekt. Nach dem **new**-Operator erscheint stets der Konstruktor. Er trägt den Namen der Klasse des erzeugten Objekts und initialisiert die Attribute.

Bei der Deklaration (2) reserviert die Variable jerry zunächst einmal Speicherplatz (im so genannten Heap). Erst mit **new** und dem Konstruktor wird dieser Platz auch tatsächlich von dem Objekt „besetzt“: erst jetzt ist es da!

Ähnlich wie bei Variablen eines primitiven Datentyps kann man Deklaration und Erzeugung („Initialisierung“) einer Variablen für ein Objekt in einer Zeile schreiben, also:

```
Klassenname variable = new Klassenname();
```

wie in Anmerkung (1).

Merkregel 19. *Objekte werden stets mit dem **new**-Operator erzeugt. Eine Ausnahme sind Strings in Anführungszeichen, wie „Hallo“; sie sind Objekte der Klasse String und werden automatisch erzeugt. Der **new**-Operator belegt („allokiert“) Speicherplatz für das erzeugte Objekt. Nach dem Operator steht immer der **Konstruktor** der Klasse, der die Attributwerte des Objekts initialisiert. Je nach Definition des Konstruktors müssen ihm eventuell Parameter mitgegeben werden.*

Technisch werden Objekte von der Virtuellen Maschine in einem eigenem Speicherbereich des RAM gespeichert, dem so genannten „Heap“.

3.4.3 Aufruf von Methoden eines Objekts

Allgemein gilt die folgende Syntax für den Aufruf der Methode `methode()` des Objekts `objekt`:

```
objekt.methode()
```

Der Unterschied zu einer statischen Methode ist also, dass die Variable des Objekts vorangeschrieben wird, und nicht der Klassenname. Wird innerhalb einer Methode eines Objekts eine weitere Methode von sich selbst, also desselben Objekts, aufgerufen, so wird die Variable des Objekts entweder weggelassen, oder es wird das Schlüsselwort **this** verwendet, also **this.eigeneMethode()**. (Bedenken Sie, dass das Objekt ja „von außen“ erzeugt wird und es selber seine eigenen Variablennamen gar nicht kennen kann!)

3.4.4 Besondere Objekterzeugung von Strings und Arrays

In Java gibt es eine große Anzahl vorgegebener Klassen, aus denen Objekte erzeugt werden können, beispielsweise die Klasse `TextField`, mit der wir schon intensiv gearbeitet haben.

Für zwei Klassen spielen in Java werden die Objekte auf besondere Weise erzeugt. Einerseits wird ein Objekt der Klasse `String` durch eine einfache Zuweisung ohne den `new`-Operator erzeugt, also z.B. `String text = "ABC";`. Andererseits wird ein Array zwar mit dem `new`-Operator erzeugt, aber es gibt keinen Konstruktor, also `double[] bestand = new double[5];` oder `char[] wort = {'H', 'a', 'l', 'l', 'e'}`;

3.5 Fallbeispiel DJTools

3.5.1 Version 1.0

Als Fallbeispiel werden wir in diesem Abschnitt das Problem betrachten, eine Liste zu spielender Musiktitel per Eingabe zu erweitern und die Gesamtspieldauer zu berechnen.

Problemstellung

Es ist eine Applikation zu erstellen, die die Eingabe eines Musiktitels ermöglicht, diesen an eine vorhandene Liste von vorgegebenen Titeln anhängt. Ausgegeben werden soll eine Liste aller Titel sowie deren Gesamtspieldauer in Minuten und in Klammern in Sekunden.

Analyse

Bevor wir mit der Programmierung beginnen, untersuchen wir das Problem in einer ersten Phase, der *Analyse*. Im *Software Engineering*, der Lehre zur Vorgehensweise bei der Erstellung von Software, werden in dieser Phase das Problem genau definiert, die notwendigen Algorithmen beschrieben und Testfälle bestimmt. Üblicherweise ist eines der Ergebnisse der Analyse eine *Spezifikation* der zu erstellenden Software, ein *Anforderungskatalog* oder ein *Lastenheft*, und zwar in Übereinkunft zwischen Auftraggeber (der das zu lösende Problem hat) und Auftragnehmer (der die Lösung programmieren will).

Eine Analyse des Problems sollte der eigentlichen Programmierung stets vorangehen. In unserem Fallbeispiel werden wir uns darauf beschränken, anhand der Problemstellung die wesentlichen Objekte und ihre Klassendiagramme zu entwerfen und einen typischen Testfall zu erstellen.

Beteiligte Objekte. Die einzigen Objekte unserer Anwendung sind die Musiktitel. (Man könnte ebenfalls die *Liste* der Musiktitel als ein Objekt sehen, wir werden diese aber einfach als Aneinanderreihung von Titeln in der *main*-Methode implementieren.) Ein Musiktitel besteht dabei aus den Daten *Interpret*, *Name* und *Dauer*. Hierbei sollte die Dauer praktischerweise in der kleinsten Einheit angegeben werden, also in Sekunden; nennen wir also die Dauer aussagekräftiger *Sekundenzeit*. Ferner sehen wir eine Objektmethode *getMinuten()* vor, die die Sekundenzeit in Minuten zurückgeben soll. Das ergibt also das folgende Klassendiagramm.

Titel
name
interpret
sekundenzeit
toString()
getSekunden()
getMinuten()

Testfalldefinition. Als Testfälle verwenden wir willkürlich folgende drei Musiktitel. Hierbei berechnen sich die Minuten einfach durch Multiplikation der Sekundenzeit mit 60. Zu beachten ist dabei, dass bei der Gesamtspieldauer die Sekunden aufaddiert werden und daraus der Wert der Minuten

abgeleitet wird. Bei der Summierung der einzelnen Minutenwerte gibt es bei einer nur näherungsweisen Speicherung der unendlichen Dezimalbrüche, wie sie prinzipiell im Computer geschieht, zu Rundungsfehlern.

Name	Interpret	Sekundenzeit	Minuten
Maneater	Nelly Furtado	197	3,28 $\bar{3}$
Catch	Blank & Jones	201	3,35
Talk	Coldplay	371	6,18 $\bar{3}$
Gesamtspieldauer		769	12,82

Entwurf

Der *Entwurf (design)* ist im Software Engineering diejenige Phase des Entwicklungsprozesses, in der die direkten Vorgaben für die Implementierung erstellt werden, also die genauen Beschreibungen der Klassen und Algorithmen. Ein *Algorithmus* ist so etwas wie ein Rezept, eine genaue Beschreibung der einzelnen Arbeitsschritte. Er bewirkt stets einen Prozess, der Daten verändert oder Informationen erzeugt. Algorithmen werden normalerweise in eigenen Methoden ausgeführt, und zwar in Objektmethoden, wenn sie den einzelnen Objekten gehören, oder in statischen Methoden, wenn sie keinem einzelnen Objekt zugeordnet werden können.

Pseudocode. In der Entwurfsphase werden Algorithmen meist in Pseudocode angegeben. *Pseudocode* ist die recht detaillierte aber knappe Beschreibung der Anweisungen. Pseudocode kann auf Deutsch oder auf Englisch geschrieben werden. Ziel ist es, dem Programmierer den Algorithmus genau genug darzustellen, so dass er ihn in eine Programmiersprache umsetzen kann. Im Pseudocode stehen üblicherweise nicht programmiersprachliche Spezifika wie beispielsweise Typdeklarationen.

Der einzige Algorithmus unseres Musiktitelproblems ist die Berechnung in Minuten, in Pseudocode lautet er einfach

```
getMinuten() {  
    zurückgeben sekundenzeit / 60;  
}
```

Der Algorithmus benötigt keine Eingabe von außen, sondern verwendet das Objektattribut `sekundenzeit` des Musiktitels.

Implementierung

Die *Implementierung* ist diejenige Phase des Software Engineerings, in der die Ergebnisse des Entwurfs, also die Klassendiagramme und Algorithmen, in Programme umgesetzt werden. Insbesondere müssen nun technische Details wie die Festlegung der Datentypen festgelegt werden. Aus dem obigen Klassendiagramm ergibt sich die folgende Klasse.

```
/** Diese Klasse stellt einen Musiktitel dar. */  
public class Titel {                                // (1)  
    // Attribute: -----  
    String name;                                    // (2)  
    String interpret;  
    int sekundenzeit;  
  
    // Konstruktor: -----  
    public Titel(String name, String interpret, int sekundenzeit) { // (3)  
        this.name = name;  
        this.interpret = interpret;  
        this.sekundenzeit = sekundenzeit;  
    }  
}
```

```

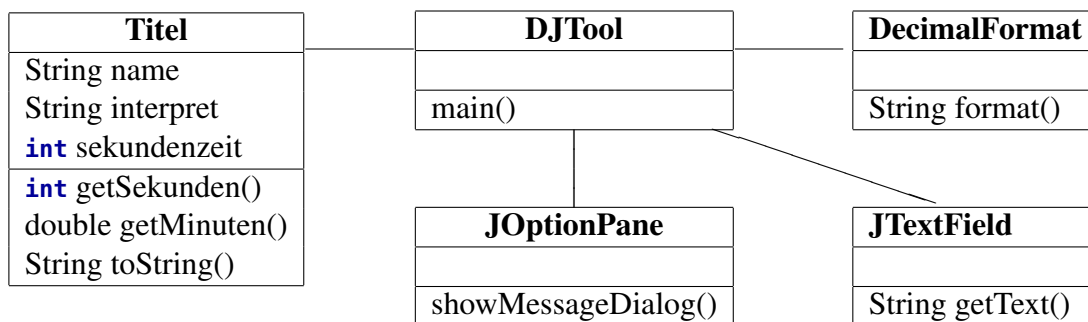
// Methoden: -----
public int getSekundenzeit() {           // (3)
    return sekundenzeit;
}

public double getMinuten() {
    return (double) sekundenzeit / 60.0;
}

public String toString() {
    return interpret + ": " + name + " (" + sekundenzeit + " sec)";
}
}

```

Das DJTool als Applikation. Die Applikation können wir nun nach der Struktur des foldenden Klassendiagramms erstellen.



Dem Diagramm ist zu entnehmen, dass die Klasse DJTool die Klassen Titel, JOptionPane, JTextField und DecimalFormat (sowie System) kennt und deren Methoden verwenden kann.

```

import java.text.DecimalFormat;
import javax.swing.*;
public class DJTool {
    public static void main(String[] args) {
        // lokale Variablen:
        int dauer = 0;
        double spieldauer = 0;
        String ausgabe = "";
        DecimalFormat f2 = new DecimalFormat("#.##0.00");

        // Standardtitel vorgeben:
        Titel titel1 = new Titel( "Maneater", "Nelly Furtado", 197)
        Titel titel2 = new Titel( "Catch", "Blank & Jones", 201)
        JTextField[] feld = {new JTextField(), new JTextField(), new JTextField()};
        Object[] msg = {"Titel:", feld[0], "Interpret:", feld[1], "Dauer (in sec):", feld[2]}
        int click = JOptionPane.showConfirmDialog(null, msg, "Eingabe", 2);

        // Konvertierung String -> int:
        dauer = Integer.parseInt( feld[2].getText() );

        // Erzeugen des eingegebenen Musiktittels:
        Titel titel3 = new Titel (feld[0].getText(), feld[1].getText(), dauer );

        // Berechne gesamte Spieldauer:
        dauer = titel1.getSekundenzeit() + titel2.getSekundenzeit()
                + titel3.getSekundenzeit();
        spieldauer = titel1.getMinuten() + titel2.getMinuten() + titel3.getMinuten();
    }
}

```



```

// erzeuge Ausgabestring:
ausgabe = titel1 + "\n" + titel2 + "\n" + titel3
ausgabe += "\n\nGesamtspieldauer: " + f2.format(spieldauer) + " min ("
ausgabe += dauer + " sec)";

// Ausgabe auf dem Bildschirm:
JOptionPane.showMessageDialog( null, ausgabe, "Titelliste", -1 );
}

```

3.5.2 Version 2.0: Formatierte Zeitangaben

Ein erster Schritt ist vollbracht, wir haben eine Applikation programmiert, die die Eingabe eines Musiktitels gestattet und aus einer Liste von Titeln die Dauer berechnet und ausgibt. Einen Wermutstropfen hat unser Programm allerdings: Wir bekommen die Zeiten nur in Darstellung voller Sekunden bzw. der Minuten als Kommazahl. Das ist nicht ganz so praktisch und widerspricht auch unserer alltäglichen Darstellung von Zeitangaben.

Neue Problemstellung

Es soll eine Applikation erstellt werden, die als Eingabe eine durch oder formatierte Zeit erwartet und alle Zeitangaben in diesem Format ausgibt.

Analyse

Da es sich um eine Modifikation eines bestehenden Programms handelt, ist natürlich die erste Frage: Was können wir von den bestehenden Programmen wieder verwenden, was müssen wir ändern, und was muss neu programmiert werden? Wenn wir uns das Klassendiagramm der ersten Version betrachten, sehen wir, dass wir die Klassen übernehmen können. Wir müssen wahrscheinlich einige Details ändern, doch die Klassen bleiben. Der einzig neue Punkt ist das Format der Zeit. Der hat es in sich, wie man schon bei der Zusammenstellung von Testfällen feststellt.

Testfälle. Zunächst gilt: Die Testfälle aus unserer ersten Analyse sollten wir beibehalten. Sie sind in jeder Phase der Entwicklung schnell zu testen und zum Erkennen grober Fehler allemal geeignet. Allerdings sollten nun die Dauern nicht in Sekunden oder Minuten als Kommazahl ein- und ausgegeben werden, sondern im Format „(h:)m:s“,

Name	Interpret	Sekundenzeit	Minuten
Maneater	Nelly Furtado	197	3:17
Catch	Blank & Jones	201	3:21
Talk	Coldplay	371	6:11
Gesamtspieldauer		769	12:49

Wie kann man nun die Zeiten 3:17, 3:21 und 6:11 addieren? Die Lösung lautet: Umrechnen der Zeiten in Sekunden, einfache Addition der Sekundenzeiten, und Umrechnung der Sekundenzeit in Stunden, Minuten und Sekunden. Also

$$„3:17“ = 0 \cdot 3600 + 3 \cdot 60 + 17 = 197, \quad „3:21“ = 0 \cdot 3600 + 3 \cdot 60 + 21 = 201,$$

und

$$„6:11“ = 0 \cdot 3600 + 6 \cdot 60 + 11 = 371,$$

also $197 + 201 + 371 = 769$ Sekunden. Die Umrechnung in Stunden geschieht dann einfach durch die ganzzahlige Division mit Rest,

$$769 \div 3600 = 0 \text{ Rest } 769.$$

Entsprechend wird der Rest dieser Division umgerechnet in Minuten:

$$769 \div 60 = 12 \text{ Rest } 49.$$

Das ergibt dann also 0:12:49.

Entwurf

Die Klassenstruktur unseres DJTools können wir weitgehend übernehmen. Wir haben aber nun jedoch einen ganz neuen Datentyp, die Zeit. Im Grunde ist eine Zeit auch nur *eine* Zahl, nur eben mit besonderen Eigenschaften. Warum also nicht eine Zeit als Objekt auffassen?

Was sind die Attribute eines Objekts Zeit? Wenn wir uns die Zeit 10:23:47 hernehmen, so hat sie die drei Attribute Stunde, Minute und Sekunde. Auf der anderen Seite müssen wir zum addieren zweier Zeiten zweckmäßigerweise auf die Sekundenzeit ausweichen. Speichern wir also ein Objekt Zeit mit der Sekundenzeit als Attribut und zeigen es mit toString im Format (h:)m:s an. Insgesamt erhalten wir das um die Klasse Uhrzeit erweiterte Klassendiagramm in Abbildung 3.4. Die Beziehungslini-

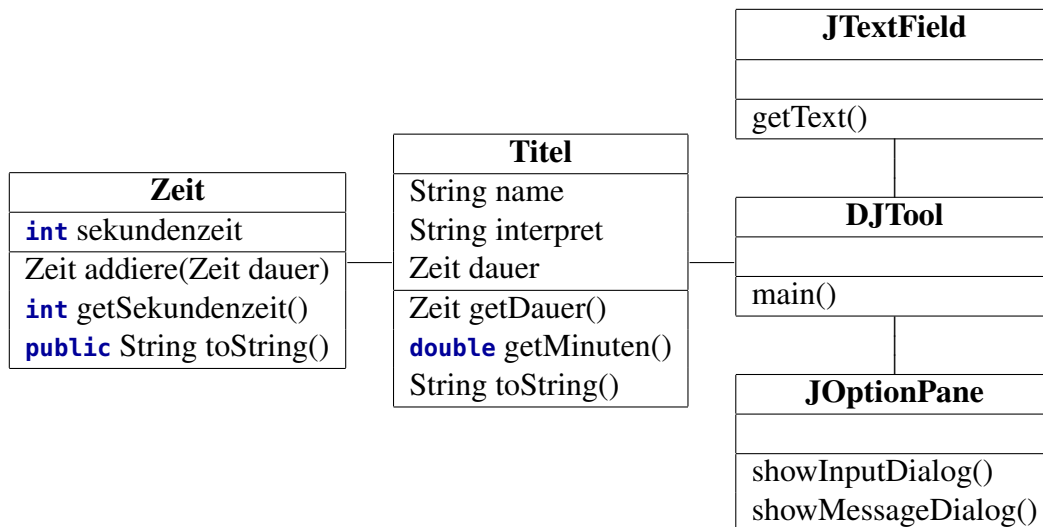


Abbildung 3.4: Klassendiagramm der DJTools (ohne die Klasse System, die stets bekannt ist)

en in diesem Diagramm besagen also, dass die Klasse **DJTool** die Klassen **Titel**, **JOptionPane** und **JTextField** kennt und deren Methoden verwenden kann. Ferner benutzt **Titel** die Klasse **Zeit**, die Berechnungen mit dem Format (h:)m:s ermöglicht.

In diesem Diagramm haben wir auch die Datentypen der Attribute und der Rückgabewerte der Methoden aufgeführt.

Implementierung

Neben leichten Modifizierungen unserer „alten“ Klassen können wir nun die Resultate der Entwurfsphase in Java codieren.

```

import javax.swing.*;
public class DJTool {
    public static void main ( String[] args ) {
        // lokale Variablen:
        Zeit spieldauer = null;
        String ausgabe = "";
    }
}
  
```

```

// Standardtitel vorgeben:
Titel titel1 = new Titel( "Maneater", "Nelly Furtado", "3:17")
Titel titel2 = new Titel( "Catch", "Blank & Jones", "3:21")
JTextField[] feld = {
    new JTextField("Talk"),new JTextField("Coldplay"),new JTextField("6:11")
};
Object[] msg = {"Titel:", feld[0], "Interpret", feld[1], "Dauer:", feld[2]}
int click = JOptionPane.showConfirmDialog(null, msg, "Eingabe", 2);
// Erzeugen des eingegebenen Musiktitels:
Titel titel3 = new Titel(feld[0].getText(),feld[1].getText(),feld[2].getText());

// Berechne gesamte Spieldauer:
spieldauer = titel1.getDauer().addiere(titel2.getDauer());
spieldauer = spieldauer.addiere(titel3.getDauer());

// erzeuge Ausgabestring:
ausgabe = titel1 + "\n" + titel2 + "\n" + titel3
ausgabe += "\n\nGesamtspieldauer: " + spieldauer;

// Ausgabe auf dem Bildschirm:
JOptionPane.showMessageDialog( null, ausgabe, "Titelliste", -1 );
}
}

```

Die Klasse Titel muss entsprechend modifiziert werden:

```

/** Diese Klasse stellt einen Musiktitel dar. */
public class Titel {
    String name;
    String interpret;
    Zeit dauer; // (1)

    /** Erzeugt einen Musiktitel, dauer wird in dem Format (h):m:s erwartet.*/
    public Titel(String name, String interpret, String dauer) { // (2)
        this.name = name;
        this.interpret = interpret;
        this.dauer = new Zeit(dauer);
    }

    /** gibt die Dauer dieses Titels als Zeit zurück.*/
    public int getDauer() { // (3)
        return dauer;
    }

    /** gibt die Attribute dieses Titels formatiert zurück.*/
    public String toString() {
        return interpret + ": " + name + " (" + dauer + ")";
    }

    /** gibt die Dauer in Minuten zurück.*/
    public double getMinuten() {
        return (double) dauer.getSekundenzeit / 60.0;
    }
}

```

Und schließlich benötigen wir die Klasse Zeit.

```

import java.util.Scanner;
/** Diese Klasse erzeugt aus Strings im Format (h:)m:s Objekte, die eine
 * Zeitspanne darstellen. Sie ist ein Datentyp, der eine Addition seiner
 * Objekte ermöglicht.
 */
public class Zeit {
    int sekundenzeit;

    /** erzeugt eine Zeit der Länge sekunden.*/
    public Zeit(int sekunden) {                                // (1)
        this.sekundenzeit = sekunden;
    }

    /** erzeugt eine Zeit aus einem String im Format (h:)m:s.*/
    public Zeit(String dauer) {                                // (2)
        // dauer scannen:
        Scanner sc = new Scanner(dauer).useDelimiter("\\D"); // (3)
        int stunden = sc.nextInt();                             // (4)
        int minuten = sc.nextInt();
        if ( sc.hasNext() ) { // dauer wurde im Format h:m:s übergeben ...
            sekunden = sc.next();
        } else { // dauer wurde im Format m:s (ohne Stunden) übergeben ...
            sekunden = minuten;
            minuten = stunden;
            stunden = 0;
        }
        this.sekundenzeit = 3600 * stunden + 60 * minuten + sekunden;
    }

    /** Addiert die übergebene dauer zu dieser Zeit und gibt die Summe zurück.*/
    public Zeit addiere(Zeit dauer) {
        return new Zeit( sekundenzeit + dauer.getSekundenzeit() );
    }

    public int getSekundenzeit() {
        return sekundenzeit;
    }

    public String toString() {
        String ausgabe = "";
        int sekunden = sekundenzeit;
        int stunden = sekunden / 3600;
        sekunden %= 60;
        int minuten = sekunden / 60;
        sekunden %= sekunden;
        if ( stunden != 0 ) {
            ausgabe += stunden + ":";
        }
        ausgabe += minuten + ":" + sekunden;
        return ausgabe;
    }
}

```

Zu beachten ist, dass die Klasse Zeit zwei Konstruktoren besitzt. Der eine erwartet die Sekundenzahl, der andere einen mit (h:)m:s formatierten String. Wie normale Methoden kann man also auch Konstruktoren mehrfach deklarieren, wenn sie verschiedene Eingabeparameter (Datentypen und/oder Anzahl) besitzen.

Zur Umwandlung des formatierten Strings wird hier die Klasse Scanner verwendet. Mit der Methode `useDelimiter("\\D")` wird ihm mitgeteilt, dass jedes nichtnumerische Zeichen als Trennzeichen aufgefasst wird. Damit wäre also auch eine Zeitangabe „3.42.57“ möglich. Möchte man dagegen

Zahleingaben im englischen Format mit dem Punkt als Dezimaltrenner erlauben, so kann man auch konkret den Doppelpunkt als Trennzeichen vorsehen, also `useDelimiter(":")`.

3.6 Zeit- und Datumfunktionen in Java

Datumsfunktionen gehören zu den schwierigeren Problemen einer Programmiersprache. Das liegt daran, dass einerseits die Kalenderalgorithmen nicht ganz einfach sind⁴, andererseits die verschiedenen Ländereinstellungen („*Locale*“) verschiedene Formatierungen erfordern.

Basis der heute allgemein üblichen Zeitangaben ist der Gregorianische Kalender⁵. Er basiert auf dem Sonnenjahr, also dem Umlauf der Erde um die Sonne. Die Kalenderjahre werden ab Christi Geburt gezählt, beginnend mit dem Jahr 1 nach Christus. Ein Jahr 0 gibt es übrigens nicht, die 0 ist erst ein halbes Jahrtausend nach Einführung dieses Systems in Indien erfunden worden. Der Gregorianische Kalender wurde 1582 von Papst Gregor XIII eingeführt und ersetzte den bis dahin gültigen und in wesentlichen Teilen gleichen Julianischen Kalender, der auf Julius Cäsar (100 – 44 v. Chr.) zurück ging und im Jahre 46 v. Chr. eingeführt wurde.

Der Gregorianische Kalender gilt in katholischen Ländern seit dem 15.10.1582 (= 4.10.1582 Julianisch), wurde aber erst später in nichtkatholischen Ländern eingeführt. Insbesondere im Deutschen Reich mit seinen vielen kleinen Fürstentümern und Ländern ergab sich in dieser Zeit erhebliche Konfusion, da in den katholischen Ländern das neue Jahr bereits begann, wenn in den protestantischen noch der 21.12. war; daher der Ausdruck „zwischen den Jahren“. Erst 1700 wurde im ganzen Deutschen Reich der Gregorianische Kalender einheitlich verwendet. Großbritannien und Schweden gingen erst 1752 zu ihm über, das orthodoxe und das islamische Ost- und Südosteuropa sogar erst im 20. Jahrhundert (Russland 1918, Türkei 1926). Seit der zweiten Hälfte des 20. Jahrhunderts gilt der Gregorianische Kalender im öffentlichen Leben der gesamten Welt.⁶

Die Datums- und Kalenderfunktionen in Java sind recht komplex und nicht ganz einheitlich strukturiert. Wir werden uns hier ganz pragmatisch nur auf die Systemzeit als Basis der Zeitmessung eines Computers und zwei Klassen beziehen, die für die meisten Belange ausreichen, die Klasse `GregorianCalendar` für die eigentlichen Zeit- und Kalenderfunktionen, sowie die Klasse `SimpleDateFormat` für die länderspezifische Formatierung der Ausgaben.

3.6.1 Die Systemzeit

Die Zeitmessung eines Computers basiert auf der *Systemzeit*. Diese Systemzeit wird gemessen in den Millisekunden seit dem 1.1.1970 0:00:00 Uhr als willkürlich festgelegten Nullpunkt. Entstanden ist das Prinzip der Systemzeit im Zusammenhang mit dem Betriebssystem UNIX in den 1970er Jahren, dort wurde sie aufgrund der geringen Speicherkapazität der damaligen Rechner in Sekunden gemessen. (Der Datentyp `int` mit 32 bit ermöglicht damit die Darstellung eines Datums vom 13.12.1901, um 21:45:52 Uhr bis zum 19.1.2038, um 4:14:07 Uhr. Demgegenüber reicht der Datenspeicher 64 bit für `long`, um die Millisekunden bis zum Jahre 292 278 994 darzustellen — ich bezweifle, dass die Menschheit diesen Zeitpunkt erreichen wird.)

In Java ermöglicht die Methode `currentTimeMillis()` der Klasse `System` das Messen der aktuellen Systemzeit. Sie gibt als **long**-Wert die Differenz der Millisekunden zurück, die die Systemzeit des

⁴Hauptsächliche Ursache ist, dass die Länge eines Tages von 24 Stunden nicht glatt aufgeht in der Länge eines Jahres von 365,2422 Tagen (ein „tropisches Jahr“). Dieser „Inkommensurabilität“ trägt man Rechnung durch Einführung von Schaltjahren (jedes 4. Jahr, außer bei vollen Jahrhunderten, die nicht durch 400 teilbar sind), die dadurch alle 3323 Jahre bis auf einen Fehler von immerhin nur 1 Tag reduziert wird. Hinzu kommt die Bestimmungen von Wochentagen, die auf der Periode 7 beruht, die einen Zyklus von 28 Jahren ergibt, an dem die Wochentage und das Datum sich exakt wiederholen.

⁵*calendae* — lat. „erster Tag des Monats“

⁶Thomas Vogtherr: *Zeitrechnung. Von den Sumerern bis zur Swatch*. Verlag C.H. Beck, München 2001, §§9&10; siehe auch <http://www.pfeff-net.de/kalend.html>

Attribut-Index	Bedeutung/Werte	Standardwert
ERA	v. Chr. oder n. Chr. (AD,BC)	AD
YEAR	Jahreszahl	1970
MONTH	Monat (0–11)	JANUARY
WEEK_OF_YEAR	Kalenderwoche (1–53)	1
DAY_OF_MONTH	Tagesdatum (1–31)	1
DAY_OF_WEEK	Wochentag	MONDAY
WEEK_OF_MONTH		0
DAY_OF_WEEK_IN_MONTH		1
AM_PM	<i>ante/post meridiem</i>	AM (vormittags)
HOUR, HOUR_OF_DAY	(0–11), (0–23)	0
MINUTE, SECOND, MILLISECOND	(0–59), (0–59), (0–999)	0
ZONE_OFFSET	Zeitzone (0–82800000)	–

Tabelle 3.2: Die Indizes der Attribute eines Kalenderobjekts. Die Attributeindizes sind statisch, d.h. man kann sie entweder über `obj.ABC` oder `GregorianCalendar.ABC` ansprechen. Die Zeitzone wird angegeben als Differenz zur Weltzeit GMT (*Greenwich Mean Time*) in Millisekunden (MEZ = +3600000).

Computers vom 1.1.1970, 0:00 Uhr trennt. (1 Millisekunde = 10^{-3} sec.) Mit der Anweisung vor Anmerkung (2),

```
jetzt = System.currentTimeMillis();
```

wird also der Wert dieser Systemzeit der Variablen `jetzt` zugewiesen. Da wir nur die Sekundenanzahl benötigen und die Methode `setStart` des Objekts `beleg` entsprechend einen Parameter des Typs `int` erwartet, muss der Wert von `jetzt` erst durch 1000 dividiert und dann wie vor Anmerkung (3) gecastet werden.

Das Messen der Systemzeit kann auch zur Bestimmung der Laufzeit von Anweisungsfolgen verwendet werden. Wer es sogar noch genauer wissen möchte, kann es mit der Methode `System.nanoTime()` versuchen, sie misst die Anzahl der Nanosekunden (1 Nanosekunde = 10^{-9} sec) ab einem vom Rechner abhängigen Zeitpunkt und eignet sich zur Messung von Zeitdifferenzen, die nicht länger als 292 Jahre (2^{63} nsec) sein dürfen. Eine Laufzeitmessung könnte also wie folgt durchgeführt werden:

```
long startzeit = System.nanoTime();
// ... zu messende Anweisungen ...
long laufzeit = System.nanoTime() - startzeit;
```

3.6.2 GregorianCalendar

Die Klasse `GregorianCalendar` befindet sich im Paket `java.util` und liefert das fast auf der ganzen Welt verwendete Kalendersystem, den Gregorianischen Kalender. Für Tage vor dem 15.10.1582, dem Einführungsdatum des Gregorianischen Kalenders, verwendet sie das Julianische System. (Genau genommen können historische Daten damit jedoch nur bis zum 4.1. des Jahres 4 korrekt dargestellt werden, da erst ab diesem Zeitpunkt das endgültige Julianische System eingesetzt wurde; vor dem Jahre 46 v. Chr. existierte der Julianische Kalender noch nicht einmal ...)

Erzeugt man ein Objekt mit dem Standardkonstruktor, so wird die zur Laufzeit aktuelle Systemzeit dargestellt. Will man eine andere Systemzeit darstellen, so kann man diese mit der Methode `setTimeInMillis` setzen, oder man kann für die einzelnen Attribute die Methode

```
set(int field, int value)
```

tun, wobei `field` der Index des zu verändernden Attributes ist, und `value` sein neuer Wert. Die Indizes der wichtigsten Attribute sind aus der Tabelle 3.2 auf S. 85 ersichtlich. Alle Attribute sind von der Klasse `Calendar` des gleichen Pakets `java.util` geerbt, eine vollständige Auflistung findet sich also in der Java API bei ihr.

Es gibt Varianten der set-Methode, die jeweils Änderungen mehrerer Attribute gleichzeitig erlauben, nämlich

```
set(int jahr, int mon, int tag), (3.1)
```

```
set(int jahr, int mon, int tag, int stunde, int min), (3.2)
```

```
set(int jahr, int mon, int tag, int stunde, int min, int sec) (3.3)
```

wobei stunde stets die *hour of day* ist, also die im mitteleuropäischen Raum übliche Stunde des 24-Stunden-Formats. Entsprechend kann man mit der Methode

```
get( int field )
```

den Wert des jeweiligen Attributs bekommen. Mit

```
datum.get( GregorianCalendar.DAY_OF_MONTH )
```

erhält man also das Tagesdatum. Vorsicht ist geboten bei dem Index für den Monat, hier liefert die get-Methode einen Wert von 0 bis 11, für eine übliche Ausgabe des Monatswertes muss man also dabei stets 1 dazu addieren. Eine weitere wichtige Methode ist

```
add( int field, int amount )
```

mit der man Zeiten addieren kann. Z.B. ergeben die folgenden Anweisungen, dass das aktuelle Datum um exakt 5 Tage zurück gestellt wird:

```
GregorianCalendar zeit = new GregorianCalendar();  
zeit.add( GregorianCalendar.DAY_OF_MONTH, -5 );
```

3.6.3 Datumsformatierungen

SimpleDateFormat

Die Klasse SimpleDateFormat des Pakets java.text ermöglicht eine relativ einfache Formatierung von Datumsangaben gemäß der Landeseinstellung („*Locale*“). Ein Datumsformat kann mit Hilfe eines Formatierungsstrings (*date and time pattern string*) vorgegeben werden. Dabei haben bestimmte Buchstaben eine feste Bedeutung, siehe Tab. 3.3. Diese Buchstaben werden in einem Formatierungsstring durch ihre entsprechenden Werte ersetzt, alle anderen Zeichen bleiben unverändert. Ein Stringabschnitt, der von Apostrophs '...' eingeschlossen ist, wird ebenfalls unverändert ausgegeben. Die Anzahl, mit der ein Formatierungsbuchstabe in dem Formatierungsstring erscheint, entscheidet in der Regel über die Länge der Ausgabe: Taucht ein Formatierungsbuchstabe einmal auf, so wird er als Zahl mit der entsprechenden Stellenanzahl dargestellt, ansonsten wird ggf. mit führenden Nullen aufgefüllt; vier gleiche Formatierungsbuchstaben bedeuten, dass ein Text in voller Länge dargestellt wird. Speziell beim Monat z.B. bedeuten MMM bzw. MMMM, dass der Monat 11 als „Dez“ bzw. „Dezember“ ausgegeben wird. Ein Formatierungsstring "yyyy/MM/dd" ergibt also z.B. „2005/06/28“, dagegen "dd-M-yy" „28-6-04“; entsprechend ergibt "k:mm' Uhr'" genau „23:12 Uhr“.

POSIX-Formatstring

Eine spezielle Kategorie der POSIX⁷ sprintf-Formatierung mit den Präfixen 't' oder 'T' für Formatierung ausschließlich in Großbuchstaben ist in Java durch die statische Methode format der Klasse String (genauer durch die Klasse Formatter) übernommen. Sie wird aufgerufen gemäß

⁷POSIX (*P*ortable *O*perating *S*ystem *I*nterface for *U*niX) ist ein gemeinsam von der IEEE und der *Open Group* für Unix entwickelte standardisierte API für C, also eine Schnittstelle zwischen Applikationen und dem Betriebssystem. Der (inter)nationale Standard trägt die Bezeichnung DIN/EN/ISO/IEC 9945.

Letter	Date or Time Component	Beispiele
G	Era designator	n.Chr., AD
y	Year	1996; 96
M	Month in year	July; Jul; 07
w	Week in year	27
W	Week in month	2
D	Day in year	189
d	Day in month	10
F	Day of week in month	2
E	Day in week	Tuesday; Tue
a	Am/pm marker	PM
H	Hour in day (0-23)	0
k	Hour in day (1-24)	24
K	Hour in am/pm (0-11)	0
h	Hour in am/pm (1-12)	12
m	Minute in hour	30
s	Second in minute	55
S	Millisecond	978
z	General time zone	Pacific Standard Time; PST; GMT-08:00
Z	Time zone (RFC 822)	-0800

Tabelle 3.3: Formatierungsbuchstaben für SimpleDateFormat. Alle anderen Buchstaben in einem Formatierungsstring werden unverändert ausgegeben.

```
String s = String.format("%1$txxx", zeit) bzw. "%1$Txxx"
```

wobei zeit ein Objekt von GregorianCalendar ist und das Suffix die Formatierung festlegen. Die Suffixtypen sind ähnlich, aber nicht vollkommen identisch denen der durch GNU date und POSIX strptime(3c) definierten. Beispielsweise ist 'L' für die Angabe der Millisekunden vorgesehen, POSIX sieht nur Sekunden vor.

Die folgenden Konversionszeichen sind zur Formatierung von Uhrzeiten vorgesehen:

'H'	Stunde des Tages, formatiert als zweistellige Zahl, ggf. durch eine führende Null ergänzt, also 00 - 23. 00 ist Mitternacht.
'k'	Stunde des Tages, ohne führende Null formatiert, also 0 - 23. 0 ist Mitternacht.
'M'	Minute, zweistellig formatiert, also 00 - 59.
'S'	Sekunde, zweistellig formatiert, also 00 - 60 ("60" ist ein spezieller Wert, um Schaltsekunden zu ermöglichen.)
'L'	Millisekunde, dreistellig formatiert, Bereich 000 - 999.
'N'	Nanosekunde, neunstellig formatiert, also 000000000 - 999999999.
'z'	gemäß RFC 822 formatierte numerische Zeitonenabweichung von GMT (<i>Greenwich Mean Time</i>), z.B. -0800.
'Z'	Die Abkürzung für die Zeitzone

Folgende Konversionszeichen werden zur Formatierung von Kalenderdaten verwendet:

'B'	der von der Ländereinstellung („Locale“) abhängige volle Monatsname, z.B. "Januar", "Februar".
'b'	der von der Ländereinstellung („Locale“) abhängige abgekürzte Monatsname, z.B. "Jan", "Feb".
'A'	der von der Ländereinstellung („Locale“) abhängige Wochentag, z.B. "Sonntag", "Montag"
'a'	die von der Ländereinstellung („Locale“) abhängige Kurzform des Wochentags, z.B. "Son", "Mon"
'Y'	Jahr, auf (mindestens) 4 Stellen formatiert, z.B. 0092 entspricht 92 n. Chr. im Gregorianischen Kalender.
'y'	Die letzten beiden Ziffern des Jahres, z.B. 00 - 99.
'j'	Tag des Jahres, dreistellig formatiert, z.B. 001 - 366 für den Gregorianischen Kalender. 001 ist der erste Tag des Jahres.
'm'	Monat, zweistellig formatiert, Bereich 01 - 13, wobei "01" der erste Monat ist des Jahres ist und "13" ein spezieller Wert um Schaltmonate in Mondkalendern zu ermöglichen.
'd'	Tag im Monat, zweistellig formatiert, also 01 - 31, "01" ist der erste Tag des Monats.
'e'	Tag im Monat, ohne führende Nullen formatiert, also 1 - 31, "1" ist der erste Tag des Monats.

Folgende Konversionszeichen werden für kombinierte Zeitangaben verwendet:

'R'	Uhrzeit als 24-Stunden-Uhr gemäß "%tH:%tM" formatiert
'T'	Uhrzeit als 24-Stunden-Uhr gemäß "%tH:%tM:%tS" formatiert
'r'	Uhrzeit als 12-Stunden-Uhr gemäß "%tI:%tM:%tS %Tp".
'D'	Datum, formatiert als "%tm/%td/%ty".
'F'	Gemäß ISO 8601 vollständiges Datum, formatiert als "%tY-%tm-%td".
'c'	Datum und Uhrzeit werden als "%ta %tb %td %tT %tZ %tY" formatiert, z.B. "Sun Jul 20 16:17:00 EDT 1969".

Näheres finden Sie in der API-Dokumentation zu der Klasse `Formatter` im Paket `java.util`:

<http://java.sun.com/j2se/1.5.0/docs/api/java/util/Formatter.html#ddt>

Beispiel

Die folgende kleine Applikation erzeugt ein Objekt `jetzt` der Klasse `GregorianCalendar`, das die aktuelle Systemzeit des Rechners repräsentiert, und gibt verschiedene Daten mit unterschiedlichen Formaten aus.

```
import java.util.GregorianCalendar;
import java.text.SimpleDateFormat;
import javax.swing.JOptionPane;

/**
 * Datums- und Zeitfunktionen.
 *
 * @author de Vries
 */
public class ZeitTest {
    public static void main ( String[] args ) {
        String ausgabe;

        GregorianCalendar jetzt = new GregorianCalendar();
        SimpleDateFormat datum1 = new SimpleDateFormat( "d.M.yyyy, H:mm:ss 'Uhr, KW 'w" );
        SimpleDateFormat datum2 = new SimpleDateFormat( "EEEE', den 'd.MMMM.yyyy' um 'H:mm:ss,SSS' Uhr'" );

        // Ausgabe & Ende:
        ausgabe = "Datum heute: " + jetzt.get(jetzt.DAY_OF_MONTH);
        ausgabe += "." + (jetzt.get(jetzt.MONTH) + 1);
        ausgabe += "." + jetzt.get(jetzt.YEAR);
        ausgabe += ", KW " + jetzt.get(jetzt.WEEK_OF_YEAR);
        ausgabe += ", " + jetzt.get(jetzt.HOUR_OF_DAY);
        ausgabe += ":" + jetzt.get(jetzt.MINUTE);
        ausgabe += ":" + jetzt.get(jetzt.SECOND);
        ausgabe += "," + jetzt.get(jetzt.MILLISECOND);
        ausgabe += "\n\noder: " + datum1.format( jetzt.getTime() );
        ausgabe += "\n\noder: " + datum2.format( jetzt.getTime() );
        ausgabe += String.format("\n\noder: %1$tA, %1$td.%1$tm.%1$tY, %1$tT,%1$tL Uhr", jetzt);

        jetzt.add(jetzt.SECOND, -3601);
        ausgabe += "\n\n - 3601 sec: " + datum1.format(jetzt.getTime());

        JOptionPane.showMessageDialog(null, ausgabe, "Datum", -1);
    }
}
```

3.7 Objekte in Interaktion: Fallbeispiel Konten

Dieses Kapitel befasst sich mit einer wesentlichen Möglichkeit, die objektorientierte Programmierung bietet: die Interaktion von Objekten. Die mögliche Erzeugung individueller Objekte mit ihren jeweils eigenen „privaten“ Eigenschaften führt dazu, dass jedes Objekt eine Art Eigenleben hat. Wir werden das am Beispiel von Bankkonten in diesem Kapitel erarbeiten.

Die Problemstellung

Es soll eine Applikation *Bankschalter* programmiert werden, mit dem es möglich sein soll, einen Betrag von einem Konto auf ein anderes zu überweisen. Es sollen 5 Konten eingerichtet werden,

jeweils mit einem Startguthaben von 100 €.

Die Analyse

Sequenzdiagramm

Betrachten wir zunächst zur Klärung des Sachverhalts das folgende Diagramm, ein so genanntes Sequenzdiagramm. Es stellt den zeitlichen Ablauf einer Überweisung dar. Zunächst wird eine Überwei-

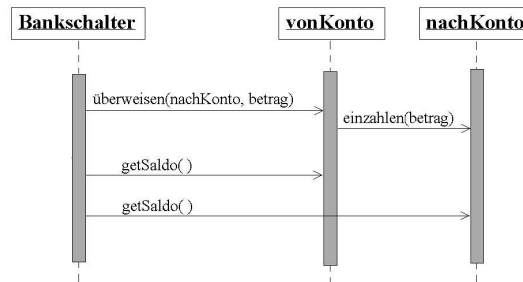


Abbildung 3.5: Sequenzdiagramm. Es zeigt die Methodenaufrufe („Nachrichten“), die die verschiedenen beteiligten Objekte ausführen.

sung ausgeführt, in dem die Methode ueberweisen des Kontos vonKonto mit dem zu überweisenden Betrag aufgerufen wird; das bewirkt, dass sofort vonKonto die Methode einzahlen von nachKonto mit dem Betrag aufruft.

Klassendiagramm

Zunächst bestimmen wir die notwendigen Klassen. Aus der Anforderung der Problembeschreibung und aus dem Sequenzdiagramm ersehen wir, dass das Softwaresystem bei einer Transaktion mehrere Objekte benötigt: die Konten der Bank, die wir als Exemplare einer Klasse sehen, die Klasse Konto.

Als nächstes müssen wir die Attribute und Methoden der einzelnen Objekte bestimmen, soweit wir sie jetzt schon erkennen können. Die „Oberfläche“ des Systems wird durch die Klasse *Bankschalter* dargestellt, sie wird also die Applikation mit der Methode *main* sein.

Für ein Konto schließlich kennen wir als identifizierendes Merkmal die Kontonummer, aber auch der Saldo (Kontostand) des Kontos. Als Methoden nehmen wir wieder diejenigen, die wir nach der Anforderung implementieren müssen. D.h., zusammengefasst erhalten wir das UML-Diagramm in Abb. 3.6

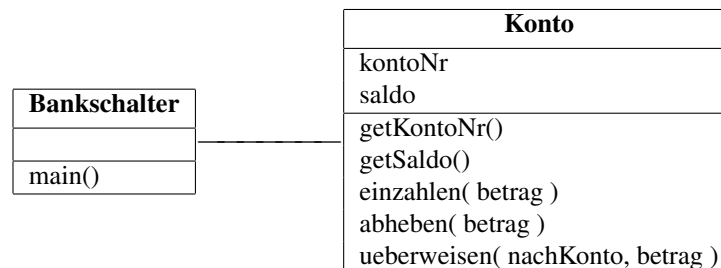


Abbildung 3.6: Klassendiagramm des Kontosystems gemäß der UML

Der Entwurf

Da unser System algorithmisch betrachtet sehr einfach ist, ergänzt der Entwurf unsere Analyse nicht viel. Neben den trivialen get-Methoden, die jeweils lediglich die Attributwerte zurückgeben, bewirken die drei Methoden eine Änderung des Saldos um den Betrag:

- einzahlen(betrag) vergrößert den Saldo des Kontos um den Betrag, saldo += betrag;
- abheben(betrag) verringert den Saldo um den Betrag, saldo -= betrag;
- ueberweisen(nachKonto, betrag) verringert den Betrag des Kontos und ruft einzahlen(betrag) von nachKonto auf:

```
nachKonto.einzahlen( betrag )
```

Die Implementierung

Bei der Implementierung geht es zunächst noch darum, die von der Analyse in UML noch offen gelassenen (normalerweise technischen) Details zu erarbeiten. In diesem Beispiel zu beachten sind die Typdeklarationen der Variablen. Wir werden die zwei Klassen wie immer in verschiedenen Dateien abspeichern. Die erste Datei lautet Konto.java

```
/** Die Klasse implementiert Konten. */
public class Konto {
    // Attribute: -----
    int kontoNr;
    double saldo;

    // Konstruktor: -----
    public Konto( int kontoNr ) {
        this.kontoNr = kontoNr;           // (1)
        saldo = 100;    // Startguthaben 100 Euro
    }

    // Methoden: -----
    public int getKontoNr() {
        return kontoNr;
    }

    public double getSaldo() {
        return saldo;
    }

    public void einzahlen( double betrag ) {
        saldo += betrag;
    }

    public void abheben( double betrag ) {
        saldo -= betrag;
    }

    public void ueberweisen( Konto nachKonto, double betrag ) {
        saldo -= betrag;
        nachKonto.einzahlen( betrag );
    }
}
```

Wir können die Methoden aus den UML-Diagrammen erkennen, ebenso die Attribute. Die zweite Klasse, die wir erstellen, schreiben wir in eine andere Datei Bankschalter.java:

```
import javax.swing.*;
/** erzeugt die Konten erzeugt und ermöglicht Überweisungen. */
public class Bankschalter {
    public static void main( String[] args ) {
        final int anzahlKonten = 5;
```

```

// erzeuge Array von Konten mit Kontonummern 0, 1, ..., 4:
Konto konto[] = new Konto[ anzahlKonten ];
for ( int i = 0; i < konto.length; i++ ) {
    konto[i] = new Konto(i);
}
Konto vonKonto, nachKonto;
double betrag;
String ausgabe;

// Eingabeblock:
JTextField[] feld = {new JTextField(), new JTextField(), new JTextField("0")};
Object[] msg = { "Überweisung von Konto", feld[0], "nach Konto", feld[1],
    "Betrag:", feld[2]};
int click = JOptionPane.showConfirmDialog(null, msg, "Eingabe", 2);

while ( click == 0 ) { // 'OK' geklickt?
    vonKonto = konto[ Integer.parseInt( feld[0].getText() ) ];
    nachKonto = konto[ Integer.parseInt( feld[1].getText() ) ];
    betrag = Double.parseDouble( feld[2].getText() );

    // Überweisung durchführen:
    vonKonto.ueberweisen( nachKonto, betrag );

    // Anzeigen:
    ausgabe = "Kontostand von Kontonr. " + vonKonto.getKontoNr();
    ausgabe += ": " + vonKonto.getSaldo() + " Euro \n";
    ausgabe += "Kontostand von Kontonr. " + nachKonto.getKontoNr();
    ausgabe += ": " + nachKonto.getSaldo() + " Euro"

    // neues Eingabefenster:
    feld[2].setText("0");
    click = JOptionPane.showConfirmDialog(null, msg, "Eingabe", 2);
}
}

```

Zusammenfassend können wir zu Objekten also festhalten:

- Objekte führen ein Eigenleben: Durch ihre privaten Attribute befinden sie sich in individuellen Zuständen. Ein einsichtiges Beispiel sind Bankkonten. Ein wesentliches Attribut eines Kontos ist der Saldo, also der Kontostand. Natürlich hat jedes individuelle Konto (hoffentlich!) seinen eigenen Saldo, d.h. er ist gekapselt oder „privat“ und nur über wohldefinierte Methoden, z.B. `einzahlen()` oder `abheben()`, veränderbar.
- Nun können zwei Konten auch interagieren, beispielsweise bei einer Überweisung. Die Überweisung muss, wenn man sie programmieren will, sauber (also *konsistent*) den Saldo des einen Kontos verkleinern und entsprechend den des anderen vergrößern. Die Summe („Bilanz“) über alle Konten muss konstant bleiben, sie ist eine *Erhaltungsgröße*.

3.8 Öffentlichkeit und Privatsphäre

3.8.1 Pakete

Mit *Paketen* kann man Klassen gruppieren. Die Erstellung eines Pakets in Java geschieht in zwei Schritten.

1. Zunächst erstellen Sie in Ihrem aktuellen Projektverzeichnis (dem CLASSPATH) ein Verzeichnis mit dem Namen des Pakets („*Paketname*“); in dieses Verzeichnis werden die `.java`-Dateien aller Klassen gespeichert, die zu dem Paket gehören sollen.

2. Die Klassen des Pakets müssen mit der Zeile beginnen:

```
package Paketname;
```

Üblicherweise werden Paketnamen klein geschrieben (awt, swing,...).

3.8.2 Verbergen oder Veröffentlichen

Manchmal ist es sinnvoll, Klassen nicht allgemein verwendbar zu machen, sondern sie nur teilweise oder gar nicht zur Verfügung zu stellen. Es ist sogar äußerst sinnvoll, nur das was notwendig ist „öffentlich“ zu machen, und alles andere zu verbergen, zu „kapseln“ wie man sagt. Warum?

Was verborgen ist, kann auch nicht separat verwendet werden, d.h. der Programmierer braucht sich nicht Sorgen machen, dass seine Programmteile irgendwo anders etwas zerstören (was insbesondere in komplexen Systemen großer Firmen ein echtes Problem sein kann). Andererseits kann kein anderes Programm oder Objekt auf verborgene Attribute zugreifen und dort unbeabsichtigte Effekte bewirken. Kapselung (*implementation hiding*) reduziert also auf zweifache Weise die Ursachen von Programm-Bugs. Das Konzept der Kapselung kann gar nicht überbetont werden.

In der Regel wird man einen differenzierten Zugriff auf Objektteile gestatten wollen. Man wird einige Attribute und Methoden öffentlich zugänglich machen und andere verbergen. Die öffentlichen Teile definieren dann die Schnittstelle nach außen.

Java verwendet drei Schlüsselwörter, um die Grenzen in einer Klasse anders zu ziehen, die Zugriffsmodifikatoren (*access specifiers*).

- **public**: Zugriff uneingeschränkt möglich;
- **protected**: alle Unterklassen („erbende“ Klassen) und Klassen desselben Pakets haben Zugriff;
- **private**: Zugriff nur innerhalb der Klasse.

Diese Zugriffsmodifikatoren (*access specifiers*) bestimmen also die Sichtbarkeit des Elements, das darauf folgend definiert wird. Lässt man vor der Definition den Zugriffsmodifikator weg, so erhält das Element den Defaultzugriff *friendly*: Er erlaubt anderen Klassen im gleichen Paket den Zugriff, außerhalb des Pakets jedoch erscheint es als **private**.

Den größten Zugriffsschutz bietet also **private**, danach *friendly* und dann **protected**. Der Zugriffsschutz ist am geringsten (nämlich komplett aufgehoben!) bei **public**. In Java sind Zugriffsmodi-

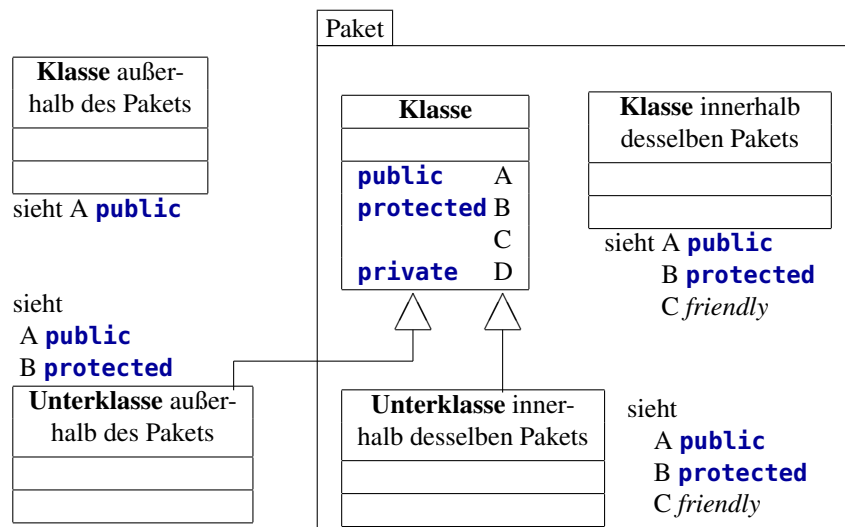


Abbildung 3.7: Die Zugriffsbereiche von Java

fiktoren erlaubt für Klassen, Schnittstellen, Methoden, Konstruktoren und Attribute. Die möglichen Zugriffsmodifikatoren sind durch die folgende Tabelle bestimmt:

Element	private	<i>friendly</i>	protected	public
Attribut	x	x	x	x
Methode	x	x	x	x
Konstruktor	x	x	x	x
Klasse		x		x
Schnittstelle		x		x

Beispiel. Die Wirkung der Zugriffsmodifikatoren sehen wir am einfachsten an Hand unseres Personenbeispiels. Deklarieren wir nämlich das Attribut `name` als **private**, also

```
// Person.java
public class Person {
    private String name;
    // Konstruktor: -----
    public Person ( String name ) {
        this.name = name;
    }
}
```

so stellen wir fest, dass sich die `main`-Klasse nun nicht mehr compilieren lässt! Stattdessen erscheint eine Fehlermeldung. Das Attribut `name` von `Person` ist also jetzt gekapselt, es ist von einer anderen Klasse („da draußen“) nicht mehr zugänglich.

Was tun, wenn wir dennoch auf den Namen zugreifen möchten, sei es um ihn zu lesen, oder um ihn sogar zu verändern? Nun, erinnern wir uns an Abbildung 3.1. Eine der Grundideen der Objektorientierung ist es, die Attribute durch Methoden zu schützen — genau das tun wir nun! Wir definieren also zwei Methoden in `Person`, nämlich `getName()` und `setName()`.

```
// Person.java
public class Person {
    // Attribut: -----
    private String name;
    // Konstruktor: -----
    public Person (String name) {
        this.name = name;
    }
    // Methoden: -----
    public String getName() {
        return name;
    }
    void setName(String name) {
        this.name = name;
    }
}
```

Beachten Sie die unterschiedlichen Zugriffsbereiche der beiden Methoden: Gelesen werden kann der Name von jeder Klasse aus, geändert werden dagegen nur von „freundlichen“ Klassen, also Klassen aus dem gleichen Paket wie `Person`. Ferner bemerken wir, dass der Konstruktor auch *friendly* ist: Ein Objekt der Klasse `Person` kann also nur von einer Klasse aus demselben Paket erzeugt werden.

Nun können wir die `main`-Klasse anpassen, so dass sie Namen lesen und schreiben kann.

```
// PersonenTest.java
import javax.swing.*;

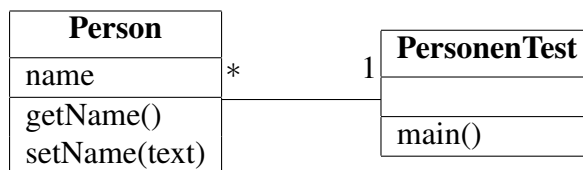
public class PersonenTest {
```

```

public static void main (String[] args) {
    Person subjekt1, subjekt2;
    String ausgabe = "";

    // Objekte:
    subjekt1 = new Person( "Tom" );    // erzeugt Objekt Tom
    subjekt2 = new Person( "Jerry" ); // erzeugt Objekt Jerry
    ausgabe += "1. Momentaufnahme: ";
    ausgabe += "subjekt1 = " + subjekt1.getName() + ", ";
    ausgabe += "subjekt2 = " + subjekt2.getName() + "\n";
    subjekt2.setName("Tina");          // aendert Namen
    ausgabe += "2. Momentaufnahme: ";
    ausgabe += "subjekt1 = " + subjekt1.getName() + ", ";
    ausgabe += "subjekt2 = " + subjekt2.getName() + "\n";
    // Ausgabe:
    JOptionPane.showMessageDialog( null, ausgabe );
}
}

```



Attribute, Methoden und Konstruktoren können also alle Zugriffsmodifikatoren erhalten, während für Klassen und Schnittstellen nur *friendly* und **public** erlaubt sind.

3.9 Vererbung

Beim Modellieren der Klassen eines Systems trifft man häufig auf Klassen, die der Struktur oder dem Verhalten nach anderen Klassen ähneln. Es ist sehr vorteilhaft, gemeinsame Struktur- und Verhaltensmerkmale (d.h.: Attribute und Methoden) zu extrahieren und sie in allgemeineren Klassen unter zu bringen. Beispielsweise kann man eine Anleihe und eine Aktie als Unterklassen einer allgemeinen Klasse Wertpapier auffassen. Ein solches Vorgehen führt zu einer *Vererbungs-* oder *Generalisierungshierarchie*.

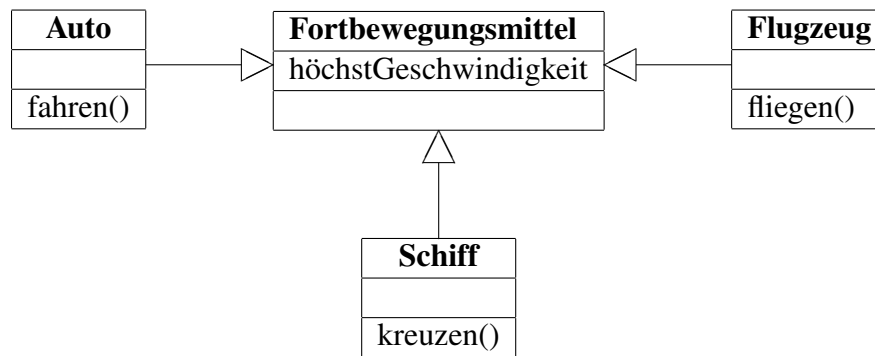
Umgekehrt ist die Vererbung eine Möglichkeit, bereits vorhandene Klassen zu erweitern. Man kann dann die Attribute und Methoden der allgemeinen Klasse übernehmen und neue Merkmale hinzufügen. Beispielsweise ist eine Vorzugsaktie eine Aktie mit dem Merkmal „kein Stimmrecht“.

Die Vererbung ist eine „ist-ein“-Relation („*is-a*“), also eine Vorzugsaktie *ist ein* Wertpapier, oder ein Auto *ist ein* Fortbewegungsmittel.

In der UML wird eine Vererbung durch eine Verbindungsgerade zwischen zwei Klassen, deren eines Ende zur allgemeinen Klasse hin mit einem hohlen Dreieck als Pfeil versehen ist.



Angewandt auf das Beispiel von Fortbewegungsmitteln wird das Prinzip der Vererbung deutlich.



Es ist zu erkennen, dass die Superklasse die Subklasse *generalisiert* oder *verallgemeinert*. Also ist „Fortbewegungsmittel“ die Superklasse von „Auto“, „Auto“ wiederum ist eine Subklasse von „Fortbewegungsmittel“. Insbesondere sieht man, dass die Generalisierung eine sehr strenge Beziehung ist, denn alle Attribute und Methoden der Superklasse sind gleichzeitig Attribute und Methoden *jeder ihrer* Subklassen. Ein Fortbewegungsmittel hat beispielsweise das Attribut „Höchstgeschwindigkeit“; damit hat jede Subklasse eben auch eine „Höchstgeschwindigkeit“. Man sagt, dass ein Objekt einer Subklasse die Attribute und Methoden der Superklasse *erbt*. Entsprechend ist damit der Ursprung des Begriffs *Vererbung* in diesem Zusammenhang geklärt.

Andererseits gibt es Attribute und Methoden, die spezifisch für eine Subklasse sind und daher nicht geerbt werden können. Zum Beispiel kann nur ein Schiff kreuzen, während ein Flugzeug fliegt und ein Auto fährt.

Ganz abgesehen von diesen technischen Aspekten hat diese Kategorisierung von Klassen den Vorteil, dass die einzelnen individuellen Objekte eines Systems, ihre Klassen, deren Klassen, usw., geordnet werden. Das allein ist allerdings schon grundlegend für unser Verständnis der Welt.

Wie sieht eine Vererbung in Java aus? Klassen werden in Java mit dem Schlüsselwort **extends** von einer anderen Klasse abgeleitet.

3.9.1 Überschreiben von Methoden

Wir haben bisher das Überladen von Methoden kennen gelernt, durch das mehrere Methoden mit gleichem Namen, aber unterschiedlichen Parameterlisten in einer Klasse deklariert werden können. Dem gegenüber steht das *Überschreiben* (*overwrite*) einer Methode: wird eine Methode einer Subklasse mit demselben Namen und derselben Parameterliste deklariert wie eine Methode der Superklasse, so ist die Superklassenmethode „überschrieben“. Ein Beispiel ist die Methode `toString` in unserer Beispielklasse `Titel`: sowohl die Subklasse `Electronica` als auch die Subklasse `Symphonie` erben sie, aber da in jeder Subklasse in jeder Subklasse diese Methode (mit den erweiterten Attributen) neu deklariert wird, ist die `Titel`-Methode damit überschrieben.

3.9.2 Die Klasse `Object`

Jede Klasse in Java ist, direkt oder indirekt, abgeleitet von einer Superklasse, der Klasse `Object`. Das ist automatisch so, auch ohne dass man explizit **extends** `Object` zu schreiben braucht. Man sagt auch, die Klasse `Object` ist die Wurzel (*root*) der allgemeinen Klassenhierarchie von Java.

Alle Klassen in Java erben also insbesondere die öffentlichen Methoden von `Object`, die wichtigsten sind `clone`, `equals`, `getClass`, `hashCode` und `toString`. Einige dieser Methoden kennen wir bereits, eine genaue Beschreibung dieser Methoden kann man in der Java-API-Dokumentation nachschlagen. Diese Methoden können in der Subklasse überschrieben werden, in der Regel wird das aber nur für `toString`, in Einzelfällen `equals` und `hashCode` gemacht. Zu beachten ist, dass wenn man die `equals`-Methode überschreibt, entsprechend auch die `hashCode`-Methode überschreiben sollte, um beide Methoden konsistent zu halten: Sind zwei Objekte gleich nach `equals`, so sollten sie insbesondere gleiche Hash-Codes liefern.

3.9.3 Die Referenz **super**

In Java kann innerhalb einer Methode das gerade betroffene („lebende“) Klassenobjekt durch die Referenz **this** angesprochen werden. Mit der Referenz **super** wird in einer abgeleiteten Klasse auf die vererbende Mutterklasse, die Superklasse referenziert. Insbesondere bei Überschreibungen von Methoden aus der Superklasse kann somit auf die originale Methode zurückgegriffen werden. Angenommen, die Klasse `ZahlendesMitglied` überschreibt die gegebene Methode `setNummer` aus der

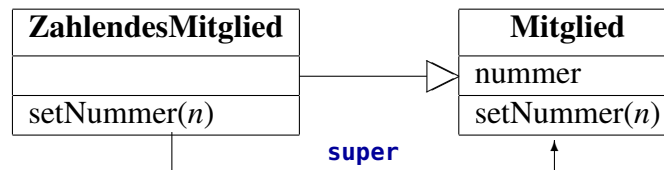


Abbildung 3.8: Wirkung der Referenz **super.setNummer(n)**

Superklasse `Mitglied`, indem die Nummer nur gesetzt wird, wenn die Anzahl `n` der Mitglieder größer als 3000 ist:

```
public void setNummer(int nummer) {
    if ( nummer < 3000 ) {
        System.err.println("Fehler!");
    } else {
        super.setNummer(n);
    }
}
```

Zahlende Mitglieder müssen also eine Nummer größer als 3000 haben. Der folgende Codeausschnitt zeigt die Zuweisung von Variablen zu den erzeugten Objekten:

```
Mitglied m1 = new Mitglied(); m1.setNummer(412); // OK!
ZahlendesMitglied m2 = new ZahlendesMitglied(); m2.setNummer(413); // Fehler!
ZahlendesMitglied m3 = new ZahlendesMitglied(); m3.setNummer(3413); // OK!
```

Ohne die **super**-Referenz hätte man in der Überschreibung der Methode `setNummer` keine Möglichkeit, auf die Methode der Superklasse zu referenzieren.

3.9.4 Fallbeispiel: Musiktitel

Ein Beispiel für eine Vererbung sind die beiden Ableitungen `Symphonie` und `Electronica` der Klasse `Musiktitel` der DJTool-Applikation:

```
/**
 * Diese Klasse stellt einen Electronica-Titel dar.
 */
public class Electronica extends Titel {
    /** DJ dieses Titels.*/
    String dj;
    /** Mix, aus dem dieser Titel stammt.*/
    String mix;

    /** Erzeugt einen Electronica-Titel.*/
    public Electronica(
        String name,String interpret,String dj,String mix,String dauer
    ) {
        super(name, interpret, dauer); // ruft den Konstruktor der Superklasse auf
        this.dj = dj;
        this.mix = mix;
    }
}
```



```

    }

    /** gibt den DJ dieses Titels zurück.*/
    public String getDj() {
        return dj;
    }

    /** gibt den Mix dieses Titels zurück.*/
    public String getMix() {
        return mix;
    }

    /** gibt die charakteristischen Attribute dieses Titels zurück.*/
    public String toString() {
        return dj+" feat. "+interpret+": "+name+", "+mix+" ("+"dauer + ")";
    }
}

```

und

```

/**
 * Diese Klasse stellt eine Symphonie als speziellen Musiktitel dar.
 */
public class Symphonie extends Titel {
    /** Komponist dieser Symphonie.*/
    String komponist;

    /** Erzeugt eine Symphonie.*/
    public Symphonie(String komponist, String name, String interpret, String dauer) {
        super(name, interpret, dauer); // ruft den Konstruktor der Superklasse auf
        this.komponist = komponist;
    }

    /** gibt den Komponisten dieser Symphonie zurück.*/
    public String getKomponist() {
        return komponist;
    }

    /** gibt die charakteristischen Attribute dieser Symphonie zurück.*/
    public String toString() {
        return komponist + ": " + name + ", " + interpret + " (" + dauer + ")";
    }
}

```

Die Klassen, von der eine gegebene Klasse abgeleitet ist, erkennt man stets oben in der durch javadoc erzeugten API-Dokumentation.

3.10 Schnittstellen (*Interfaces*)

Im Sinne der Objektorientierung spielen Objektmethoden die Rolle von Schnittstellen, also die Kontaktpunkte zur Welt außerhalb des Objekts. Im übertragenen Sinne wären also Schnittstellen von Menschen beispielsweise die Augen oder der Mund, mit ihren entsprechenden Funktionalitäten des Sehens, oder des Schmeckens und des Sprechens, durch die der Kontakt zur Außenwelt hergestellt wird. Um eine Schnittstelle zu beschreiben, ist es wichtig, den jeweiligen Input oder Output der Tätigkeit darzustellen, nicht jedoch die konkrete Realisierung, oder wie man objektorientiert spricht: ihre „Implementierung“. Das Auge mit seiner Methode „sehen“ als Schnittstelle ist also dadurch beschrieben, dass Informationen über das Licht (Input) verarbeitet werden, der Mund mit seinen Methoden „schmecken“ und „sprechen“ dadurch, dass die Information chemischer Moleküle (Input) verarbeitet

wird und Worte als Schall entstehen. Wie das geschieht, ist nicht mehr Zuständigkeit der Schnittstellenbeschreibung, es gibt Menschen, die Rot und Grün nicht unterscheiden können oder verschiedene Sprachen sprechen.

Im technischen Bereich spielen Schnittstellen eine wesentliche Rolle im Zusammenhang mit Normierungen. Betrachten wir als Beispiel ein Elektrohaushaltsgerät. Es sind (mindestens) zwei Schnittstellen für ein solches Gerät notwendig, ein Stecker für die Stromentnahme und ein Schalter zum Ein- und ggf. Ausschalten. Wie diese Funktionalitäten (objektorientiert: „Methoden“) Stromentnahme und Einschalten konkret realisiert (objektorientiert: „implementiert“) werden, ist geräteabhängig, das Einschalten einer Kaffeemaschine ist anders als dasjenige eines Toasters oder eines Kühlschranks.

Merkregel 20. Eine Schnittstelle ist definiert durch ihr „Protokoll“, d.h. durch strikte Festlegung ihrer Ein- bzw. Ausgabeformate, also den Rahmen der Kommunikation mit der Außenwelt. Eine Schnittstelle ist wie eine Steckdose, in die ein Stecker, die „Implementierung“, passen muss. Eine Schnittstelle ermöglicht auf diese Weise einen beliebigen Austausch aller sie implementierenden Komponenten, also aller Komponenten, die dieses Protokoll erfüllen.

In Java definiert man über eine Schnittstelle diejenigen Methoden, die jede Klasse enthalten muss, die die Schnittstelle implementiert. Eine Schnittstelle wird in Java wie eine Klasse deklariert, nur wird das Schlüsselwort **interface** anstatt **class** verwendet, und alle deklarierten Methoden haben keinen Methodenrumpf:

```
[public] interface Schnittstelle {  
    [public] Rückgabetyt methode1(Parameterliste);  
    :  
    [public] Rückgabetyt methode2(Parameterliste);  
}
```

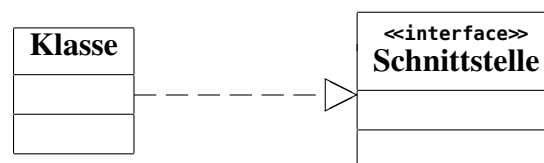
Eine Schnittstelle und ihre Methoden können wie üblich als **public**, **protected** oder **private** deklariert werden, allerdings nicht als **static**.⁸

Eine Klasse, die eine Schnittstelle implementiert, erhält nach der Deklaration ihres Namens das Schlüsselwort **implements** und den Namen der Schnittstelle:

```
public class Klasse implements Schnittstelle {  
    :  
}
```

In diesen Klassen müssen alle Methoden der Schnittstelle deklariert werden, ansonsten ist sie nicht kompilierbar. Sollen mehrere Schnittstellen implementiert werden, so werden diese nach **implements** durch Kommata getrennt aufgelistet.

Als Klassendiagramm wird die Beziehung einer Schnittstelle und einer sie implementierenden Klasse ähnlich wie eine Vererbung dargestellt:



Die einzigen Unterschiede sind, dass eine Implementierung durch eine gestrichelte Linie dargestellt und die Schnittstelle selber mit dem Wörtchen **<<interface>>** in doppelten Dreiecksklammern etikettiert wird.

⁸Der Grund für die Nicht-Implementierbarkeit statischer Methoden ist subtil und liegt vereinfacht gesagt darin, dass man eine solche Methode im Allgemeinen mit dem Namen der Schnittstelle aufrufen müsste, und damit ist die Referenz nicht mehr klar; die Objektmethoden dagegen gehören einem vorher erzeugten Objekt einer bestimmten implementierenden Klasse und sind somit wohlreferenziert. Aus einem ähnlichen Grund können Schnittstellen auch keine Konstruktoren besitzen, denn von ihr kann kein Objekt erzeugt werden.

Beispiel 3.2. In dem folgenden Quelltext werden zwei Schnittstellen Auge und Mund deklariert, sowie zwei Klassen Mensch und Human, die sie implementieren (Abbildung 3.9). In der Applikation

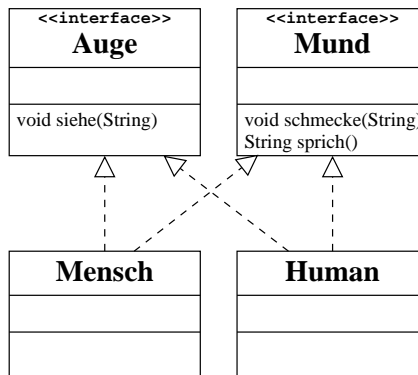


Abbildung 3.9: Die Schnittstellen Auge und Mund und die sie implementierenden Klassen Mensch und Human.

TestInterface werden sie auf verschiedene Weisen verwendet.

```

interface Auge {
    public void siehe(String licht);
}

interface Mund {
    public void schmecke(String molekuele);
    public String sprich();
}

class Mensch implements Auge, Mund {
    public void siehe(String licht) {
        System.out.println("Ich sehe " + licht);
    }

    public void schmecke(String molekuele) {
        System.out.println("Ich schmecke " + molekuele);
    }

    public String sprich() {
        return "Ich sage, was ich denke.";
    }
}

class Human implements Auge, Mund {
    public void siehe(String licht) {
        System.out.println("I see " + licht);
    }

    public void schmecke(String molekuele) {
        System.out.println("I taste " + molekuele);
    }

    public String sprich() {
        return "I say what I think.";
    }
}

public class TestInterface {
    /** diese statische Methode ist für jede Schnittstelle Mund definiert.*/
    public static void schreie(Mund mund) {
        System.out.println(mund.sprich().toUpperCase());
    }
}
  
```

```

}

public static void main(String[] args) {
    Mensch du = new Mensch();
    Human you = new Human();
    du.siehe("Laub im Sommer");
    du.schmecke("Erdbeeren");
    schreie(du);
    you.siehe("summer leaves");
    schreie(you);

    // erzeuge ein Objekt vom Typ Mund (kann allerdings nicht sehen!):
    Mund mund = new Mensch();
    mund.schmecke("Gurken");
    System.out.println(mund.sprich());
}
}

```

In der main-Methode wird zunächst ein Objekt `du` der Klasse `Mensch` und ein Objekt `you` der Klasse `Human` erzeugt und deren Methoden sowie die statische Methode `schreie` der Applikationsklasse `TestInterface` aufgerufen. Danach wird ein Objekt `mund` vom Typ `Mund` (!) erzeugt, allerdings implementiert als `Mensch`. Der Vorteil: man kann nun die Methode `schreie` aufrufen, die ein Objekt vom allgemeinen Typ `Mund` erwartet; Nachteil: das Objekt `mund` kann nicht sehen, es hat keine Methode `siehe`, obwohl es durch die Klasse `Mensch` implementiert ist! □

Ein großer Vorteil von Schnittstellen in Java ist, dass man eine Schnittstelle als Parameter von Methoden übergeben kann, die die wohldefinierten Methoden verwenden. Ein Beispiel ist die obige Methode `schreie` der Klasse `TestInterface`.

In Java oft verwendete Schnittstellen sind der `ActionListener`, der für die Registrierung von Eingabeaktionen wie Mausklicks oder Drücken der Return-Taste zuständig ist und für den die nach der Aktion aufgerufene Methode implementiert werden muss, das Interface `Runnable` zur Implementierung eines Threads, oder `Comparable` zur Implementierung sortierbarer Objekte.

Eine Art Zwischending zwischen Schnittstellen und Klassen sind *abstrakte Klassen*, die wie folgt mit dem Schlüsselwort **abstract** beginnen und sowohl Attribute und Methoden (wie üblich, statisch und mit beliebigen Zugriffsmodifikatortypen) als auch abstrakte, also noch zu implementierende Methoden besitzen:

```

[public] abstract class Klasse {
    [public|protected] abstract Rückgabetyp methode(Parameterliste);
    :
}

```

Aus solchen Klassen kann kein Objekt erzeugt werden, der Programmierer ist gezwungen, eine eigene Sub-Klasse per `extends` zu vererben und sämtliche abstrakten Methoden zu implementieren. Im Gegensatz zu einer Schnittstelle kann man in einer abstrakten Klasse also gewisse Klassenstrukturen vordefinieren, also Attribute und Standard-Methoden. Zu beachten ist, dass Methoden und Attribute, auf die die ererbenden Sub-Klassen zugreifen können sollen, nicht `private` deklariert werden dürfen, sondern engstenfalls `protected`.

3.11 Zusammenfassung

Objekte

- Ein *Objekt* ist die Darstellung eines individuellen Gegenstands oder Wesens (konkret oder abstrakt, real oder virtuell) aus dem zu modellierenden *Problembereich* der realen Welt.

- Ein Objekt ist eindeutig bestimmt durch seine *Attribute* und durch seine *Methoden*.

Klasse
attribute
methoden()

- Das Schema bzw. die Struktur eines Objekts ist seine *Klasse*.
- Zur Darstellung und zum Entwurf von Klassen wird das Klassendiagramm gemäß der UML verwendet.
- Die Attributwerte sind durch die Methoden *gekapselt*. Der Zugriff auf sie geschieht nur durch öffentliche Methoden, insbesondere get-Methoden für lesenden und set-Methoden für schreibenden Zugriff.
- Eine Klasse, aus der Objekte erzeugt werden sollen, wird in der Regel in einer eigenen Datei *Klasse.java* implementiert.
- Die Syntax für eine Java-Klasse lautet:

```
/** Kommentar.*/
public class Klassenname {
    Deklarationen der Attribute;
    Deklaration des Konstruktors;
    Deklarationen der Methoden;
}
```

Das reservierte Wort **public** kann hier vor **class** weg gelassen werden. Üblicherweise sind die Attribute *nicht* **public**, die Methoden jedoch sind es.

- Die Prozesse, die ein Objekt erfährt oder auslöst, werden in Methoden ausgeführt. Die Deklaration einer Methode besteht aus dem Methodennamen, der Parameterliste und dem Methodenkörper:

```
public Datentyp des Rückgabewerts methodName ( Datentyp p1, ..., Datentyp pn ) {
    Anweisungen;
    [ return Rückgabewert; ]
}
```

Die Parameterliste kann auch leer sein ($n = 0$). Der Rückgabewert kann leer sein (**void**). Ist er es nicht, so muss die letzte Anweisung der Methode eine **return**-Anweisung sein. Normalerweise sind die Methoden eines Objekts *nicht statisch*.

- Attribute gelten objektweit, also sind im gesamten Objekt bekannt, lokale Variablen jedoch nur innerhalb ihrer Methoden.
- Das Schlüsselwort **this** ist eine Referenz auf das aktuelle Objekt. Mit ihm kann auf objekt-eigene Attribute und Methoden verwiesen werden.
- In jeder Klasse, aus der Objekte erzeugt werden, sollte ein Konstruktor deklariert werden, das ist eine spezielle Methode, die genauso heißt wie die Klasse, ohne Rückgabebetyp deklariert wird und die Attributwerte initialisiert. Es können mehrere Konstruktoren deklariert werden, die sich in ihrer Parameterliste unterscheiden.
- Üblicherweise wird eine öffentliche Methode `toString()` deklariert, die die wesentlichen Attributwerte eines Objekts als String zurück gibt.

- Die Erzeugung von Objekten geschieht durch den **new**-Operator, direkt gefolgt von dem Konstruktor.
- Die Methode `methode()` des Objekts `objekt` wird aufgerufen durch `objekt.methode()`. Wird innerhalb einer Objektmethode eine Methode des eigenen Objekts aufgerufen, so wird entweder die Variable ganz weggelassen oder es wird das Schlüsselwort **this** verwendet.

Schnittstellen und abstrakte Klassen

- Mit Hilfe von Schnittstellen (*interfaces*) und abstrakten Klassen kann festgelegt werden, welche Methoden (mit Namen und Eingabeparametern) von einer implementierenden bzw. erbenenden Klasse definiert werden müssen.
- Schnittstellen und abstrakte Klassen sind so eine Art Vertrag: Auch wenn der Programmierer die genaue Implementierung nicht kennt, kann er doch die Schnittstellenmethoden aufrufen und seine Programme kompilieren. Solche Programme ablaufen lassen oder konkrete Objekte erzeugen kann er freilich erst mit (nicht-abstrakten) Klassen.
- Während eine Schnittstelle nur die Strukturen der zu implementierenden Methoden („Signatur“) beschreibt, können abstrakte Klassen daneben auch normale Attribute und Methoden definieren.
- Abstrakte Klassen verwendet man daher, wenn für spätere Programmierer bereits eine „Standard-Implementierung“ vorgegeben werden soll, Schnittstellen dagegen, wenn spätere Programmierer größtmögliche Freiheit bei der Implementierung haben sollen (z.B., weil genaue Implementierungen noch gar nicht bekannt sind oder die implementierenden Programme auf jeweils verschiedene Formate oder Sprachen spezialisiert sein sollen).

Kapitel 4

Datenstrukturen: Collections

Ein zentrales Problem jeder Entwicklung von Software ist die Verarbeitung und Speicherung von *Daten*. Bisher kennen wir dazu zwei wichtige Konzepte: Einerseits können wir einzelne Daten in Variablen speichern, entweder von einem primitiven Datentyp für Zahlen bzw. Zeichen, oder, für komplexe *Datensätze* (*records*), als Objekte, die ja strukturierte Einheiten letztendlich primitiver Daten sind. Das zweite Konzept sind Arrays, mit denen wir mehrere gleichartige Datensätze speichern und per Index zugreifbar machen.

In der betriebswirtschaftlichen Praxis wichtige zu verarbeitende Daten sind beispielsweise Bilanzzahlen oder Umsatzzahlen, die man entweder als **int**-Werte (in Cent) abspeichert, wenn Summen und Differenzen exakt zu berechnen sind, oder als **double**-Werte, wenn eine hohe Genauigkeit im Dezimalstellenbereich erforderlich ist, beispielsweise bei der Zinseszinsrechnung. Ansammlungen gleichartiger Datensätze, wie beispielsweise die Zugangsmengen eines Lagerbestands, kann man in einem Array speichern.

Ein Array ist die grundlegendste und einfachste *Datenstruktur*. Datenstrukturen werden allgemein dazu verwendet, insbesondere große Mengen an Daten strukturiert und effizient zu speichern. Die einzig optimale Datenstruktur gibt es leider nicht, eine gute Einsetzbarkeit hängt von dem jeweiligen konkreten Problem ab. Zur Speicherung eines alphabetisch sortierten Telefonbuchs eignen sich ganz andere Datenstrukturen als zum Beispiel solche zur Verwaltung einer Druckerwarteschlange. Entsprechend unterscheiden sich die Algorithmen zum Einfügen, Suchen oder Löschen von Datensätzen, abhängig von der Datenstruktur.

In der Theoretischen Informatik haben sich im Laufe der zweiten Hälfte des letzten Jahrhunderts einige wichtige Konzepte der Datenstrukturierung entwickelt, die heute als Standards in Einführungen zur Informatik gelehrt werden. In Java sind diese Datenstrukturen durch das *Collections Framework* recht einfach implementierbar. Wir werden das ausnutzen und anhand der Collections die theoretischen Konzepte der Datenstrukturierung gleichzeitig mit ihrer Implementierung durchgehen.

4.1 Notation und Grundlagen

Definition 4.1. Für eine reelle Zahl x bezeichnet $\lfloor x \rfloor$ die größte ganze Zahl kleiner oder gleich x , oder formaler:

$$\lfloor x \rfloor = \max\{n \in \mathbb{Z} \mid n \leq x\} \quad (4.1)$$

Die $\lfloor \dots \rfloor$ -Zeichen heißen „untere Gaußklammern“ (*floor brackets*). Entsprechend heißen die Zeichen $\lceil \dots \rceil$ „obere Gaußklammern“ (*ceiling brackets*), und für eine reelle Zahl x bezeichnet $\lceil x \rceil$ die kleinste ganze Zahl größer oder gleich x , oder formaler:

$$\lceil x \rceil = \min\{n \in \mathbb{Z} \mid n \geq x\} \quad (4.2)$$

□

Die unteren Gaußklammern $\lfloor x \rfloor$ einer positiven Zahl $x > 0$ bewirken, dass ihre Nachkommastellen abgeschnitten werden, also $\lfloor \pi \rfloor = 3$, $\lfloor 3 \rfloor = 3$, die oberen Gaußklammern $\lceil x \rceil$, dass auf die nächste ganze Zahl aufgerundet wird, also $\lceil \pi \rceil = 4$, $\lceil 4 \rceil = 4$. Für negative Zahlen jedoch gilt es genau anders herum, beispielsweise $\lfloor -5,43 \rfloor = -6$, $\lfloor -\pi \rfloor = -4$, oder $\lceil -\pi \rceil = -3$.

Für einen **double**-Wert x entspricht $\lfloor x \rfloor$ genau der Wirkung der Methode `Math.floor()`, und $\lceil x \rceil$ derjenigen von `Math.ceil()`.

4.2 Bedeutung sortierter Datenstrukturen

Beispiel 4.2. (*Ein Telefonbuch*) Da ein Telefonbuch aus gleich strukturierten Datensätzen der Form (Name, Vorname, Straße, Telefonnummer) besteht, einem sogenannten *Tupel*, liegt die Idee nahe, es als ein Array aus Objekten der Klasse `Eintrag` zu speichern, also:

```
public class Eintrag {
    String name;
    String vorname;
    String strasse;
    long nummer;
}
```

Mit der Deklaration `Eintrag[] telefonbuch;` in der `main`-Methode einer Applikation könnte man dann durch `telefonbuch[i]` auf den Eintrag Nummer i zugreifen, und mit den entsprechenden `get`- und `set`-Methoden könnte man dann auf die Daten zugreifen. Wenn man nun bei jedem neuen Eintrag darauf achtet, dass er gemäß der alphabetischen Sortierung nach dem Namen in das Array eingefügt wird, so ist das Array jederzeit sortiert. \square

Was ist der Vorteil eines sortierten Verzeichnisses? Nun, die Suche nach einem bestimmten Eintrag ist sehr effizient. So findet man einen Namen im Telefonbuch sehr schnell, auch wenn es viele Einträge hat. Versuchen Sie im Unterschied dazu einmal, eine bestimmte Telefonnummer in dem Telefonbuch Ihrer Stadt zu finden! Der große Vorteil liegt darin, dass bei einem von links nach rechts aufsteigend sortierten Verzeichnis `verzeichnis` mit $n = \text{verzeichnis.length}$ Einträgen das „binäre Suchen“ nach s angewandt werden kann:

```
public static int binaerSuchen(char s, char[] verzeichnis) {
    int mitte, links = 0, rechts = verzeichnis.length - 1;

    while( links <= rechts ) {
        mitte = (links + rechts) / 2;
        if ( s > verzeichnis[mitte] ) { // rechts weitersuchen ...
            links = mitte + 1;
        } else if ( s < verzeichnis[mitte] ) { // links weitersuchen ...
            rechts = mitte - 1;
        } else { // Suche erfolgreich!
            return mitte;
        }
    }
    return -1;
}
```

Diese Methode gibt die Indexposition des Suchbuchstaben in dem Character-Array `verzeichnis` an, wenn er darin enthalten ist, und den Wert `-1`, wenn nicht. Ein Array ist ein *Verzeichnis mit direktem Zugriff* (*random access*), da man über den Zeigerindex, beispielsweise i , direkt auf jeden beliebigen Eintrag `array[i]` lesend und schreibend zugreifen kann.

Satz 4.3. (*Suche in einem Verzeichnis*) In einem Verzeichnis mit direktem Zugriff auf n Einträge bei bekanntem n benötigt man zur vollständigen Suche eines gegebenen Suchbegriffs s ...

... maximal $\lceil \log_2 n \rceil + 1$ Vergleiche des Suchbegriffs in einem sortierten Verzeichnis,

... im ungünstigsten Fall n Vergleiche des Suchbegriffs in einem unsortierten Verzeichnis.¹

Eine Suche heißt dabei „vollständig“, wenn der Suchbegriff entweder in dem Verzeichnis existiert und seine Position gefunden wird, oder aber nicht vorhanden ist und dies sicher nachgewiesen wird.

Beweis. Ist das Verzeichnis sortiert, so kann man die folgende Variante des binären Suchens verwenden:

```
public static int binaerSuchen2(char s, char[] verzeichnis) {
    int links = 0, rechts = verzeichnis.length - 1;
    int pos = (links + rechts)/2;

    while( links < rechts ) {
        if ( s > verzeichnis[pos] ) { // rechts weitersuchen ...
            links = pos + 1;
        } else { // links weitersuchen ...
            rechts = pos;
        }
        pos = (links + rechts)/2;
    }
    if ( s != verzeichnis[pos] ) pos = -1;
    return pos;
}
```

Es kann maximal $\lceil \log_2 n \rceil$ Iterationen geben, so dass bei n Einträgen nur noch einer übrig bleibt; da aber nicht sicher ist, ob dieser Eintrag dem Suchbegriff entspricht, muss noch die Gleichheit überprüft werden, also gibt es $\lceil \log_2 n \rceil + 1$ Iterationen der binären Suche. In jeder Iteration wird jeweils genau ein Vergleich des Suchbegriffs durchgeführt.

In einem unsortierten Verzeichnis ist ein ungünstiger Fall, wenn der Suchbegriff sich nicht im Verzeichnis befindet. Dann muss jeder einzelne Eintrag auf Gleichheit geprüft werden, d.h. man benötigt n Vergleiche. Q.E.D.

Beispiel 4.2 (Fortsetzung) Sucht man in einem kleinen Telefonbuch mit $n = 3$ Einträgen der Namen

{Bach, Mozart, Tschaikowski},

nach dem Eintrag Albinoni, so benötigt man mit `binaerSuchen2` also $\lceil \log_2 3 \rceil + 1 = 3$ Vergleiche, um herauszufinden, dass er sich nicht im Telefonbuch befindet, wie die Wertetabelle 4.1 zeigt. □

links	rechts	pos	links < rechts	s > verzeichnis[pos]	s != verzeichnis[pos]
0	2	1	j		
	1	0	j	n	
	0	0	n	n	
					j

Tabelle 4.1: Wertetabelle für die Telefonbuchsuche mit $s = \text{"Albinoni"}$, mit drei Verzeichnisvergleichen.

¹Zwar ist für ein Verzeichnis mit einer Hash-Tabelle, bei dem die Position eines Eintrags abhängig von seinem Wert berechnet wird, die *mittlere* Laufzeit einer vollständigen Suche sogar konstant [6, S. 524ff], im ungünstigsten Fall jedoch ist sie linear.

Man spricht beim binären Suchen von einem „Algorithmus mit logarithmischer Laufzeit“, während das Suchen in einem unsortierten Verzeichnis im schlimmsten Fall „lineare Laufzeit“ benötigt. Der Unterschied zwischen den beiden Laufzeitklassen macht sich insbesondere für große Werte von n bemerkbar:

n	10	100	10 000	1 000 000
$\lceil \log_2 n \rceil + 1$	5	8	15	21

In einem unsortierten Telefonbuch mit 1 Mio Einträgen müssten Sie im schlimmsten Fall 1 Mio Namen nachschauen, für ein sortiertes dagegen wären Sie *spätestens* nach 21 Vergleichen fertig!

4.3 Sortierung in Java

Soll ich chronologisch oder alphabetisch antworten?

Filmzitat aus *Sherlock Holmes* (2010)

Um Objekte sortieren zu können, muss für ihre Klasse zunächst eine *Ordnung* existieren, d.h., dass zwei beliebige Objekte o_1 , o_2 der Klasse verglichen werden können und einer der Beziehungen $o_1 < o_2$, $o_1 == o_2$ oder $o_1 > o_2$ genügen. Ein mathematisches Beispiel für Mengen mit einer Ordnung sind die reellen Zahlen.

Erstellen wir eine eigene Klasse, so ist nicht immer eine natürliche Ordnung der Objekte dieser Klasse gegeben. Ein Beispiel ist die folgende einfache Klasse `Kreis`:

```
public class Kreis {
    double radius;

    public Kreis(double radius) {
        this.radius = radius;
    }
}
```

Ein `Kreis` ist also allein durch seinen Radius definiert. Würden wir nun mehrere Objekte mit verschiedenen Radien in ein Array (oder eine Collection, s.u.) packen, hätten wir keine Chance, sie irgendwie zu sortieren.

4.3.1 Das Interface Comparable

Comparisons are easily done

Once you've had a taste of perfection

Katy Perry, „*Thinking of You*“ (2009)

Implementiert eine Klasse `T` das Interface `Comparable<T>`, so wird sie mit einer Ordnung ausgestattet. Dazu muss die Methode `int compareTo(o)` deklariert werden, so dass

$$\text{compareTo}(T\ o) = \begin{cases} 1 & \text{für } \text{this} > o, \\ 0 & \text{für } \text{this.equals}(o), \\ -1 & \text{für } \text{this} < o. \end{cases}$$

Statt 1 bzw. -1 können hier auch beliebige positive bzw. negative Integerwerte verwendet werden. Obwohl nicht zwingend vorgeschrieben, sollte entsprechend die Methode `equals` überschrieben werden, so dass sie mit der Ordnungsrelation konsistent bleibt. Das wiederum sollte parallel mit einer entsprechenden Änderung der Standardmethode `int hashCode` einhergehen, denn eines der „ungeschriebenen Gesetze“, bzw. ein Kontrakt in Java lautet:

Merkregel 21. Wird die equals-Methode überschrieben, so muss die hashCode-Methode überschrieben werden, so dass beide konsistent bleiben. D.h., sind zwei Objekte gleich gemäß equals, so müssen sie denselben Hashcode haben (nicht notwendig umgekehrt).

Hintergrund ist, dass die hashCode-Methode den Hashcode eines Objekts berechnet, d.i. eine ganze Zahl, die eine Art Prüfziffer des Objekts darstellt und für viele effizienten Speicherungen in Java verwendet wird, insbesondere bei HashSet oder HashMap.

Betrachten wir als Beispiel unsere Klasse Kreis. Eine naheliegende Ordnung ist, einen Kreis k1 größer als einen anderen Kreis k2 zu nennen, wenn sein Radius größer ist, und umgekehrt. Demnach wären zwei Kreise *als Objekte* gleich, wenn sie gleichen Radius haben. In Java könnte diese

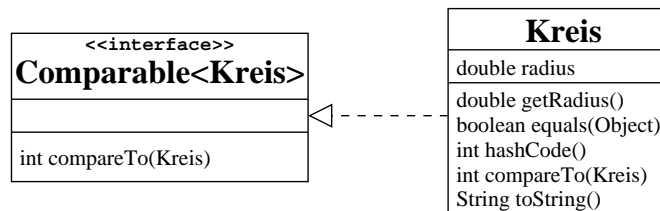


Abbildung 4.1: Die Klasse Kreis, die Comparable implementiert.

Ordnungsrelation durch die folgende Klassendeklaration realisiert werden, in dessen main-Methode beispielhaft drei Kreise erzeugt werden (Abbildung 4.1):

```

1 public class Kreis implements Comparable<Kreis> {
2     public static final double GENAUIGKEIT = 1e-8; // Genauigkeit 10-8
3     private double radius;
4
5     public Kreis(double radius) {
6         this.radius = radius;
7     }
8
9     public String toString() {
10        return "S("+radius+")";
11    }
12
13    public double getRadius() {
14        return radius;
15    }
16
17    public boolean equals(Object o) {
18        if (o instanceof Kreis) {
19            return compareTo((Kreis) o) == 0; // => konsistent mit compareTo!
20        }
21        return false;
22    }
23
24    public int hashCode() {
25        return (int) (1e5 * this.getRadius()); // konsistent mit equals!
26    }
27
28    public int compareTo(Kreis p) {
29        double diff = this.getRadius() - p.getRadius();
30        if (diff > GENAUIGKEIT) return 1;
31        if (diff < -GENAUIGKEIT) return -1;
32        return 0;
33    }
34
35    public static void main(String[] args) {
  
```

```

36     Kreis[] k = {
37         new Kreis(Math.sqrt(2)), new Kreis(1.), new Kreis(1. + GENAUIGKEIT/2)
38     };
39
40     java.util.Arrays.sort(k);
41
42     String txt = "", txt2 = "\nhashCode: ";
43     for (int i = 0; i < k.length; i++) {
44         for (int j = i+1; j < k.length; j++) {
45             txt += "\nk" + i + "=" + k[i] + ", k" + j + "=" + k[j] +
46                 ", k" + i + ".equals(k" + j + ")=" + k[i].equals(k[j]) +
47                 ", k" + i + ".compareTo(k" + j + ")=" + k[i].compareTo(k[j]);
48         }
49         txt2 += "k" + i + "=" + k[i].hashCode() + ", ";
50     }
51     javax.swing.JOptionPane.showMessageDialog(null, txt + txt2, "Kreise", -1);
52 }
53 }

```

Die Ausgabe des Programms lautet:

```

k0=S(1.0), k1=S(1.0000000005), k0.equals(k1)=true, k0.compareTo(k1)=0
k0=S(1.0), k2=S(1.4142135623730951), k0.equals(k2)=false, k0.compareTo(k2)=-1
k1=S(1.0000000005), k2=S(1.4142135623730951), k1.equals(k2)=false, k1.compareTo(k2)=-1
hashCode: k0=100000, k1=100000, k2=141421,

```

Man beachte die (wegen der internen Binärdarstellung generelle!) Schwierigkeit, die Gleichheit von **double**-Werten zu prüfen, da durch Rechnungen normalerweise nur Näherungswerte bestimmt werden. In diesem Beispiel wird das Problem gelöst, indem eine Konstante GENAUIGKEIT definiert wird, die die Genauigkeit vorgibt, mit der zwei **double**-Werte auf Gleichheit geprüft werden.

In dieser Implementierung sind equals und compareTo miteinander konsistent, denn this.equals(p) ist true genau dann, wenn this.compareTo(p) gleich 0 ist. Entsprechend wird der dritte Kreis nicht mehr in die sortierte Menge aufgenommen, er ist ja gleich dem zweiten. Generell muss die equals-Methode als Eingabe ein allgemeines Objekt erwarten. Um zu überprüfen, ob die Klasse dieses Objekts überhaupt von der Klasse Kreis ist, verwendet man das reservierte Wort **instanceof**.

Die Sortierung eines Arrays wird hier durch die statische Methode der Klasse Arrays durchgeführt. Sie sortiert aufsteigend nach der in der compareTo-Methode definierten Ordnung. (Insbesondere braucht man die Methode nicht selber zu programmieren!)

Hätten wir die Koordinaten (x,y,z) des Mittelpunktes als Attribute zu unserer Klasse Kreis hinzugefügt und würden zwei Kreise gleich nennen, wenn ihre Radien *und* ihre Mittelpunkte gleich sind (ggf. im Rahmen einer gewissen Genauigkeit), so müssten wir die equals- und die hashCode-Methode anpassen, nicht aber die compareTo-Methode.

4.3.2 Das Interface Comparator

Das Interface Comparable ist sehr praktisch, wenn man den Objekten einer Klasse ein festes und eindeutiges Sortierkriterium geben will. Manchmal möchte man jedoch Objekte nach einem anderen, oder nach Bedarf vielleicht auch nach verschiedenen Sortierbegriffen ordnen. Beispielsweise möchte ein Logistiker Containerkisten mal nach ihrem Gewicht, ein anderes Mal nach ihrem Volumen sortieren. Für solche Zwecke verwendet man in Java das Interface Comparator. Klassen, die einen Comparator<T> implementieren, müssen die Methode compare(T p, T q) deklarieren, die jeweils 1, 0 oder -1 zurück gibt, abhängig davon, ob p größer, gleich oder kleiner als q ist.

Betrachten wir dazu als Beispiel die Klasse Kiste, die zwei Comparatoren verwendet, GewichtSort und VolumenSort. Der erste vergleicht die Gewichte zweier Kisten miteinander, der zweite ihre Volumina. Beide sind so implementiert, dass sie eine Sortierung in *absteigender* Reihenfolge ermöglichen,

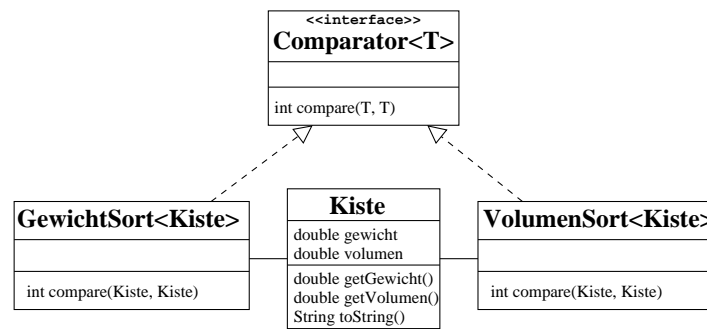


Abbildung 4.2: Die Klasse Kiste, die zwei die Schnittstelle Comparator implementierende Klassen verwendet.

also die schweren bzw. die großen Kisten zuerst (Abbildung 4.2). Um mehrere Objekte der Klasse Kiste mit ihnen zu sortieren, muss zunächst ein Comparator-Objekt erzeugt werden und dieser mit der zu sortierenden Liste von Kisten von der statischen Methode `sort` der Klasse `Collections` aufgerufen werden, also z.B.

```
GewichtSort gs = new GewichtSort();
Collections.sort(liste, gs);
```

Entsprechend der `compare`-Methode des Comparators `GewichtSort` wird die Liste dadurch sortiert.

```

1  import java.util.*;
2
3  /** Comparator zur absteigenden Sortierung von Kisten nach ihrem Gewicht.*/
4  class GewichtSort implements Comparator<Kiste> {
5      public int compare(Kiste p, Kiste q) {
6          double diff = p.getGewicht() - q.getGewicht();
7          if ( diff < 0 ) return -1;
8          if ( diff > 0 ) return 1;
9          return 0;
10     }
11 }
12
13 /** Comparator zur absteigenden Sortierung von Kisten nach ihrem Volumen.*/
14 class VolumenSort implements Comparator<Kiste> {
15     public int compare(Kiste p, Kiste q) {
16         double diff = p.getVolumen() - q.getVolumen();
17         if ( diff < 0 ) return 1;
18         if ( diff > 0 ) return -1;
19         return 0;
20     }
21 }
22
23 /** Stellt eine Kiste mit gegebenem Gewicht und Volumen dar.*/
24 public class Kiste {
25     private double gewicht;
26     private double volumen;
27
28     public Kiste(double gewicht, double volumen) {
29         this.gewicht = gewicht;
30         this.volumen = volumen;
31     }
32
33     public String toString() {
34         return "(" + gewicht + " kg, " + volumen + " m^3)";
35     }
36
37     public double getGewicht() {
38         return gewicht;

```

```

39     }
40
41     public double getVolumen() {
42         return volumen;
43     }
44
45     public static void main(String[] args) {
46         Kiste[] k = {
47             new Kiste(Math.sqrt(2), 3.0), new Kiste(1., 4.0), new Kiste(1., 1.5)
48         };
49
50         GewichtSort gs = new GewichtSort();
51         VolumenSort vs = new VolumenSort();
52
53
54         String txt = "";
55         for (int i = 0; i < k.length; i++) {
56             for (int j = i+1; j < k.length; j++) {
57                 txt += "\nk" + i + "=" + k[i] + ", k" + j + "=" + k[j] +
58                     "\n gs.compare(k"+i+", k"+j+") = " + gs.compare(k[i],k[j]) +
59                     ", vs.compare(k"+i+", k"+j+") = " + vs.compare(k[i],k[j]);
60             }
61         }
62
63         ArrayList<Kiste> liste = new ArrayList<Kiste>(k.length);
64         for (int i = 0; i < k.length; i++) {
65             liste.add(k[i]);
66         }
67         txt += "\nliste=" + liste;
68         Collections.sort(liste, gs); // sortiere Kisten nach Gewicht
69         txt += "\nliste=" + liste;
70         Collections.sort(liste, vs); // sortiere Kisten nach Volumen
71         txt += "\nliste=" + liste;
72         javax.swing.JOptionPane.showMessageDialog(null, txt, "Kisten", -1);
73     }
74 }

```

Die Ausgabe dieses Programms lautet:

```

k0=(1.4142135623730951 kg, 3.0 m^3), k1=(1.0 kg, 4.0 m^3)
gs.compare(k0, k1) = 1, vs.compare(k0, k1) = 1
k0=(1.4142135623730951 kg, 3.0 m^3), k2=(1.0 kg, 1.5 m^3)
gs.compare(k0, k2) = 1, vs.compare(k0, k2) = -1
k1=(1.0 kg, 4.0 m^3), k2=(1.0 kg, 1.5 m^3)
gs.compare(k1, k2) = 0, vs.compare(k1, k2) = -1
liste=[(1.4142135623730951 kg, 3.0 m^3), (1.0 kg, 4.0 m^3), (1.0 kg, 1.5 m^3)]
liste=[(1.0 kg, 4.0 m^3), (1.0 kg, 1.5 m^3), (1.4142135623730951 kg, 3.0 m^3)]
liste=[(1.0 kg, 4.0 m^3), (1.4142135623730951 kg, 3.0 m^3), (1.0 kg, 1.5 m^3)]

```

Hierbei ist die Klasse `ArrayList<Kiste>` aus dem Paket `java.util` fast dasselbe wie das Array `Kiste[]`, nur erstens müssen Objekte davon (wie üblich) mit dem Konstruktor erzeugt werden, zweitens werden neue Elemente mit der Methode `add` eingefügt, und drittens wird eine `ArrayList` mit der `sorts`-Methode der Klasse `Collections` sortiert (nicht über die von Arrays). Die `ArrayList` ist eine der fundamentalen Datenstrukturen von Java, eine der *Collections*. Wie sie genau definiert ist und sich zu den anderen *Collections* verhält, werden wir in den nächsten Abschnitten lernen.

4.4 Theoretische Konzepte für Datenstrukturen

4.4.1 Die drei Grundfunktionen einer Datenstruktur

Wir bezeichnen eine allgemeine Ansammlung von strukturierten Daten allgemein als *Datenstruktur* (*data structure*) oder *Kollektion* (*collection*). Manchmal finden Sie in der Literatur auch den Begriff *Container*, im Umfeld von Java hat sich allerdings *Collection* etabliert.

Neben der Suche eines gegebenen Eintrags sind zwei weitere grundlegende Funktionen einer allgemeinen Datenstruktur das Einfügen (*insert* oder *add*) und das Löschen (*delete* oder *remove*) einzelner Einträge. Ist die Datenstruktur ein sortiertes Array, so muss eine Routine zum Einfügen sinnvollerweise die Sortierung beachten. Wollen wir in das Telefonbuch aus Beispiel 4.2 (Fortsetzung) den Eintrag Beethoven einfügen, so müssen wir zunächst die Stelle finden, an die der Eintrag kommen soll (hier also $i=1$), dann „Platz schaffen“, indem alle Einträge danach um einen Platz nach rechts verschoben werden, und schließlich in die „frei“ gewordene Stelle den neuen Eintrag speichern. Die Suche nach der Position geschieht am effizientesten mit dem binären Suchen, benötigt also logarithmische Laufzeit, das „Platz-schaffen“ benötigt schlimmstenfalls n Aktionen, hat also lineare Laufzeit, das Einfügen benötigt nur eine Operation. Insgesamt erhalten wir also für eine Einfügeroutine eines sortierten Arrays ungünstigenfalls die Anzahl

$$T_{\text{insert}}^{\text{max}}(n) = \underbrace{\lceil \log_2 n \rceil + 1}_{\text{binäre Suche}} + \underbrace{n}_{\text{Platz schaffen}} + \underbrace{1}_{\text{speichern}} \approx n \quad (4.3)$$

für sehr große n , also sehr große Arrays. Das Einfügen in ein sortiertes Array erfordert also lineare Laufzeit im ungünstigsten Fall.

Für das Löschen eines Eintrags muss man zunächst den Eintrag finden und dann alle nachfolgenden Einträge nach links verschieben. Entsprechend benötigt das Löschen im ungünstigen Falle etwa n Operationen, hat also ebenfalls lineare Laufzeit.

Bei der Implementierung einer allgemeinen Datenstruktur als ein Array ergibt sich nun jedoch ein grundsätzliches technisches Problem: bei der Erzeugung eines Arrays muss bereits seine maximale Größe bekannt sein. Aber schon unser Telefonbuchbeispiel zeigt, dass die maximale Anzahl von Einträgen von vornherein oft gar nicht vorhersagbar ist. (Abgesehen davon muss bei den meisten Programmiersprachen der Index eines Array ein integer-Wert sein; in Java bedeutet das, dass die maximale Größe eines Arrays $2^{31} - 1 = 2\,147\,483\,647$ betragen kann, abhängig von der Größe des Arbeitsspeichers und des Datentyps der Einträge aber eher kleiner ist: für 1 GB Arbeitsspeicher kann ein Array von `char` „nur“ etwa 300 Mio Einträge umfassen.)

Zusammen gefasst ergeben sich also die folgenden Probleme bei der Speicherung eines Verzeichnisses durch ein Array:

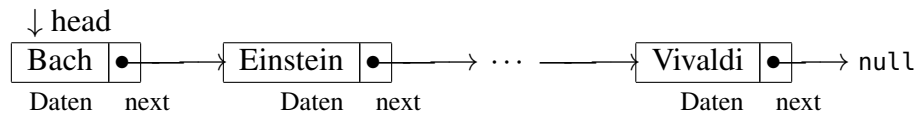
- Für ein Array muss die maximale Anzahl von Einträgen von vornherein bekannt sein. Legt man es „zur Vorsicht“ zu groß an, vergeudet man unnötig wertvollen Speicherplatz, legt man es zu klein an, können Einträge irgendwann nicht mehr gespeichert werden.
- Das Einfügen eines Eintrags insbesondere an den Anfang eines Arrays erfordert das Bewegen von sehr vielen Einträgen, um „Platz zu schaffen“. Für sehr große Arrays kostet das sehr viel Laufzeit.
- Das Löschen von Einträgen, insbesondere am Anfang eines Arrays erfordert das Bewegen sehr vieler Einträge, um die entstandene Lücke zu schließen und kostet daher ebenfalls sehr viel Laufzeit.

Welche Alternativen zu Arrays als Datenstrukturen? Wir werden die bedeutendsten zunächst theoretisch beschreiben und erste Erfahrungen mit ihren Konzepten sammeln. Jede dieser alternativen Datenstrukturen hat jeweils ihre Vor- und Nachteile, und ihr Einsatz hängt von der konkreten Art des jeweils zu lösenden Problems ab.

4.4.2 Verkettete Listen (*Linked Lists*)

Die erste Datenstruktur, die wir neben dem Array betrachten wollen, ist die verkettete Liste. Sie ist als theoretisches Konzept radikal anders als ein Array und eine rein dynamische Datenstruktur. Sie basiert wesentlich auf „Zeigern“.

Eine *verkettete Liste* (*linked list*) besteht aus *Knoten* (*node*), einem Datensatz, der den eigentlichen Datenteil (*data*) und einen *Zeiger* (*pointer*) enthält, der auf einen weiteren Knoten oder auf das Nullobjekt `null` verweist. Der erste Knoten einer verketteten Liste heißt *Kopf* (*head*), der letzte Knoten verweist stets auf `null`. Der auf den nächsten Knoten verweisende Zeiger heißt „next“.

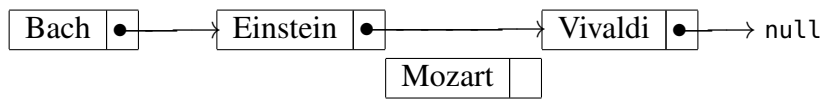


In einer verketteten Liste haben wir nur auf den Kopf der Liste direkten Zugriff, die weiteren Knoten erreichen wir nur, indem wir den Zeigern folgen. Im Gegensatz zu einem Array ist eine verkettete Liste also *kein* Verzeichnis mit direktem Zugriff.

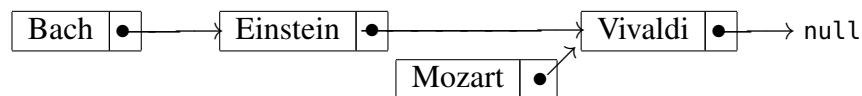
Betrachten wir die drei Operationen Einfügen, Suchen und Löschen eines Knotens bei verketteten Listen. und deren Laufzeiten. Nehmen wir dazu beispielhaft die folgende verkettete Liste:



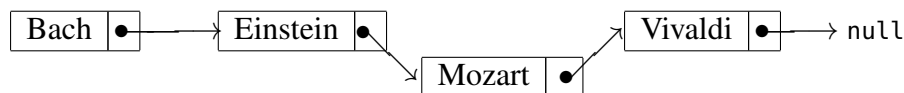
Wir möchten nun den Knoten `Mozart` hinter den Knoten `Einstein` einfügen. Wie muss man vorgehen? Die Ausgangssituation kann man wie folgt darstellen:



Folgen wir den Zeigern, beginnend beim Kopf der Liste, so suchen wir den Knoten `Einstein`, indem wir bei jedem angelangten Knoten den Datenteil mit dem Eintrag Einstein vergleichen. Sind die beiden Daten nicht gleich, so haben wir ihn noch nicht gefunden und folgen dem Zeiger zum nächsten Knoten in der Liste, ansonsten war die Suche erfolgreich. In unserer Beipielliste sind wir also schon beim zweiten Schritt am Ziel der Suche. Jetzt kopieren wir den Zeiger des Knotens `Einstein` als next-Zeiger für den einzufügenden Eintrag:

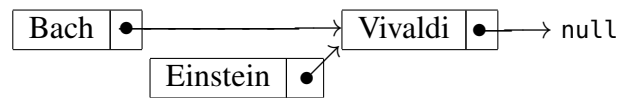


In diesem Zwischenschritt referenzieren also *zwei* Zeiger auf den Knoten `Vivaldi`! Abschließend wird der Zeiger von `Einstein` auf den neuen Knoten „umgebogen“, so dass die Liste die folgende Gestalt hat:



Wie gewünscht ist also Mozart nach Einstein in unsere Liste eingefügt.

Versuchen wir nun, den Knoten `Einstein` aus unserer Originalliste (4.4) zu löschen. Dazu muss die Liste also so modifiziert werden, dass der Zeiger des Knoten `Bach` auf `Vivaldi` zeigt. Um auf den Knoten `Bach` zuzugreifen, müssen wir wieder beim Kopf starten und die Liste durchlaufen, bis wir den Knoten `Bach` erreichen, indem wir für jeden Zeiger den *vohergehenden* Knoten speichern. Haben wir den zu löschenden Knoten gefunden, so nehmen wir den Zeiger des Vorgängers und lassen ihn auf Einstein zeigen, `Einstein`, also:



In Java wird eine verkettete Liste typischerweise durch die Schnittstelle `Iterator<E>` dargestellt, die die drei Methoden `boolean hasNext()` und `E next()` vorschreibt, wobei `hasNext()` prüft, ob ein Element der Klasse `E` existiert, und `next()` das nächste Element zurück gibt und dessen Zeiger zum nächsten Element folgt.

4.4.3 Bäume

Eine verkettete Liste ist eine sogenannte *lineare Datenstruktur*, jedes Element hat höchstens einen Nachfolger. Eine Verallgemeinerung einer Liste ist ein „Baum“, eine nichtlineare Datenstruktur, in der jedes Element mehrere Nachfolger haben kann. Bäume finden vielfältige Anwendungen, beispielsweise werden sogenannte B*-Bäume häufig zur Indizierung von Datenbanken verwendet. Allgemein gesprochen zeichnet sich ein Baum dadurch aus, dass seine Elemente, die „Knoten“, durch „Kanten“ verknüpft sind. Formal definieren wir:

Definition 4.4. Ein *Baum (tree)* ist eine endliche nichtleere Menge von Elementen, *Knoten (nodes)* genannt, für die gilt:

- a) es gibt genau einen speziell ausgezeichneten Knoten, die *Wurzel (root)* des Baumes;
- b) jeder Knoten zeigt auf eine möglicherweise leere Folge von anderen Knoten, seine *Kinder (children)* oder *Nachfolger*, so dass auf jeden Knoten des Baumes außer der Wurzel genau ein Knoten zeigt, sein *Elternknoten (parent)* oder *Vorgänger*.

Ein Knoten ohne Kinder ist ein *Blatt (leaf)*, ein Knoten, der weder die Wurzel noch ein Blatt ist, heißt *innerer Knoten* des Baumes. Die Verbindung eines Knotens mit seinen Kindern heißt *Kante*. □

Aus Definition 4.4 folgt, dass jeder innere Knoten eines Baumes die Wurzel eines echten Teilbaumes (*subtree*) des Baumes ist.² Üblicherweise implementiert man einen Baum durch Knoten, die als Attribute Zeiger auf weitere Knoten haben. Blätter zeigen demnach auf `null` (je nach Implementierung aber auch auf einen speziellen „Pseudoknoten“, was manche Algorithmen des Baums vereinfacht).

Ein Baum stellt also stets eine Hierarchie seiner Knotenelemente dar, wobei in jeder Hierarchieebene sich die Kinder einer gleichen Generation befinden. Solche hierarchischen Strukturen gibt es sehr häufig in der realen Welt, beispielsweise

- das Organigramm eines Unternehmens,
- die Struktur eines Buches mit Kapiteln, Abschnitten und Unterabschnitten,
- die Unterteilung eines Landes in Bundesstaaten, Bezirke, Kreise und Städte;
- Stammbäume als Darstellung der Nachkommen eines Menschen
- die Gewinner der einzelnen Spiele eines Sportturniers nach dem KO-System;
- die Struktur des Dateiverzeichnisses eines Rechners in Laufwerke, Verzeichnisse, Unterverzeichnisse, Dateien;

²Solche Bäume werden auch „gewurzelte Bäume“ (*rooted trees*) genannt, manchmal werden allgemeinere, so genannte „freie Bäume“ betrachtet [2]. Ferner ist nach Definition 4.4 die Reihenfolge der Kinder eines Knotens wichtig. In der mathematischen Literatur betrachtet man oft Bäume, bei denen die Reihenfolge der Kinder keine Rolle spielt und Bäume als eine spezielle Klasse „zyklenfreier Graphen“ aufgefasst werden.

- die Tag-Struktur eines HTML- oder XML-Dokuments.

In der Mathematik können Klammerungen ebenfalls durch eine Hierarchie dargestellt werden. So können wir beispielsweise den arithmetischen Ausdruck

$$((6 \cdot (4 \cdot 28) + (9 - ((12/4) \cdot 2)))$$

als einen Baum auffassen (Abbildung 4.3b). Übrigens zeichnen Informatiker Bäume in der Regel mit

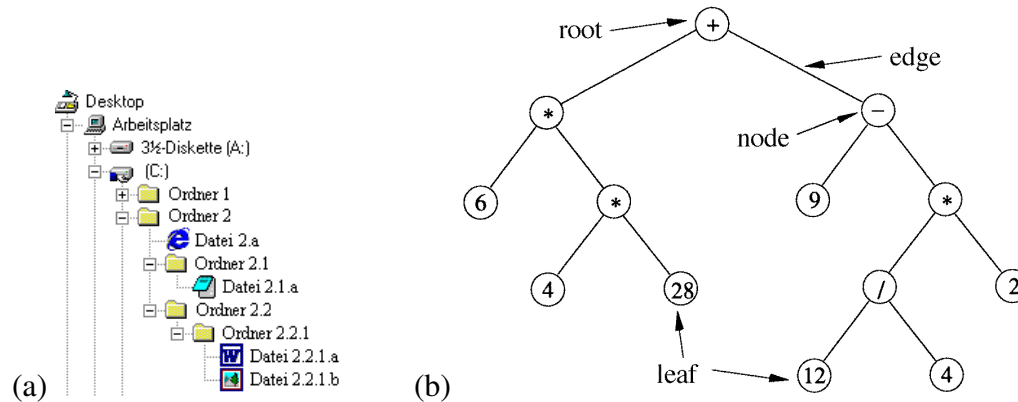


Abbildung 4.3: (a) Ein Verzeichnisbaum. (b) Ein Baum, der den arithmetischen Ausdruck $((6 \cdot (4 \cdot 28) + (9 - ((12/4) \cdot 2)))$ darstellt, mit Wurzel (*root*), Kanten (*edges*) (inneren) Knoten (*nodes*) und Blättern (*leaves*).

der Wurzel nach *oben*, anders als deren biologische Vorbilder.

Definition 4.5. Die *Höhe* h eines Baumes ist die Anzahl der Kanten eines längsten direkten Pfades von der Wurzel zu einem Blatt. □

Die Höhe eines Baumes ist also die größtmögliche Anzahl seiner Generationen oder Hierarchieebenen, wobei die Wurzel die nullte Generation bzw. Ebene ist.

Beispiele 4.6. Die Höhe des Baumes in Abbildung 4.3(a) ist $h = 6$, die Höhe des Baumes in Abbildung 4.3(b) ist $h = 4$. Man überlegt sich leicht, dass ein Baum mit n Knoten und der Höhe $h = n - 1$ eine verkettete Liste sein muss. □

Definition 4.7. Ein Baum heißt *ausgeglichen* (*balanced*), wenn sich die Höhen aller Teilbäume einer Generation um höchstens 1 unterscheiden. □

Beispiele 4.8. Die beiden Bäume in Abbildung 4.3 sind nicht ausgeglichen. So hat der erste der drei Teilbäume von Ordner 2 die Höhe 1, der zweite die Höhe 2 und der dritte die Höhe 3. Entsprechend hat der linke Teilbaum 9 nach dem Knoten - die Höhe 0, der rechte Teilbaum * die Höhe 2. □

Definition 4.9. Ein *binärer Baum* (*binary tree*) ist ein Baum, dessen Knoten maximal zwei Kinder haben (von denen keines, eins oder beide null sein können). □

Der Baum in Abbildung 4.3 (b) ist ein binärer Baum.

Satz 4.10. Die Höhe h eines binären Baumes mit n Knoten beträgt

$$h = \lfloor \log_2 n \rfloor. \quad (4.5)$$

Beweis. Zunächst beobachten wir, dass die Anzahl n an Knoten eines ausgeglichenen Baumes einer Höhe h durch die Ungleichungen

$$1 + 2 + \dots + 2^{h-1} + 1 \leq n \leq 1 + 2 + \dots + 2^{h-1} + 2^h \quad (4.6)$$

beschränkt ist. (Die rechte Ungleichung wird zur Gleichung, wenn der Baum „voll“ ist.) Nun sind sowohl $1 + 2 + \dots + 2^{h-1}$ als auch $1 + 2 + \dots + 2^h$ geometrische Reihen, d.h. es gilt³

$$1 + 2 + \dots + 2^{h-1} = 2^h - 1, \quad 1 + 2 + \dots + 2^h = 2^{h+1} - 1.$$

Damit ergibt (4.6) dann $2^h \leq n \leq 2^{h+1} - 1$, und wegen $2^{h+1} - 1 < 2^{h+1}$ also

$$2^h \leq n < 2^{h+1}.$$

Mit der rechten Seite erhalten wir $h \leq \log_2 n$, und mit der linken $\log_2 n < h + 1$. Logarithmieren der Ungleichungen (was erlaubt ist, da der Logarithmus monoton steigend ist) ergibt daher

$$h \leq \log_2 n < h + 1.$$

Also gilt $h = \lfloor \log_2 n \rfloor$.

Q.E.D.

Heaps

Bäume werden in der Informatik hauptsächlich verwendet, um sortierte Daten zu speichern. Je nach Hintergrund oder Sinn der Sortierung gibt es verschiedene Baumstrukturen. Fast immer werden dazu Binärbäume verwendet. Jeder Knoten muss einen eindeutigen „Schlüssel“ haben, nach dem sortiert werden kann. Ein Binärbaum heißt dann *sortiert*, wenn für jeden seiner Knoten gilt:

1. kein Knoten in seinem linken Teilbaum hat einen größeren Schlüssel,
2. kein Knoten in seinem rechten Teilbaum hat einen kleineren Schlüssel.

Oft ist man aber nur an dem einen Knoten mit dem besten (oder schlechtesten) Schlüssel interessiert. So arbeiten Computer Warteschlangen von durchzuführenden Prozessen häufig nicht nach dem FIFO-Prinzip ab, sondern verarbeiten als nächstes den Prozess mit der höchsten Priorität. Solche Warteschlangen heißen *Priority Queues*. Ein weiteres Beispiel sind Sportturniere, bei denen nur der Gewinner einer Paarung sich für die nächste Runde qualifiziert („KO-System“). Einer der einfachsten Binärbäume für solche Aufgaben ist der Heap,⁴ auf Deutsch oft auch *Halde* genannt.

Definition 4.11. Ein *Heap* ist ein binärer linksvollständiger Baum, bei dem *jeder* Teilbaum als Wurzel einen Knoten mit dem maximalen Schlüssel dieses Teilbaums hat. Ein solcher Heap wird auch *Maximumheap* genannt. Ein *Minimumheap* ist ein binärer linksvollständiger Baum, bei dem entsprechend jeder Teilbaum als Wurzel einen Knoten mit dem minimalen Schlüssel hat. \square

Ein Heap ermöglicht ein schnelles Einfügen von Knoten und ein schnelles Suchen des Maximums (bzw. Minimums). Allerdings gibt es nicht die Möglichkeit einer schnellen Suche nach einem beliebigen Knoten, dafür müssen alle Knoten des Baums sukzessive durchlaufen werden. Ein Heap ist ein „partiell geordneter Baum“. Er kann als ein Array implementiert werden, siehe Abbildung 4.4. Die Wurzel ist dann der Array-Eintrag $a[0]$, und für einen gegebenen Index i eines Eintrags ergeben sich die Indizes $p(i)$ seines Elternelements, seines linken Kindes $l(i)$ und seines rechten Kindes $r(i)$ durch die folgenden Formeln:

$$p(i) = \left\lfloor \frac{i-1}{2} \right\rfloor, \quad l(i) = 2i+1, \quad r(i) = 2(i+1). \quad (4.7)$$

Natürlich existiert ein Elternknoten $p(i)$ nur für $i > 0$. Der Index eines linken Kindes ist stets eine ungerade Zahl, während derjenige eines rechten Kindes stets gerade ist. Wann nun kann man ein gegebenes Array als einen Heap darstellen, und wann nicht? Das formale Kriterium gibt der folgende Satz an.

³Ein Informatiker kann sich die geometrische Reihe $1 + 2 + \dots + 2^{h-1} = 2^h - 1$ leicht klarmachen, indem er sich überlegt, dass die Zahl, die h gesetzten Bits entspricht, genau $2^h - 1$ ist, und jedes gesetzte Bit eine Zweierpotenz darstellt. Beispielsweise ist für $h = 4$ die Summe $\sum_{k=0}^4 2^k = 1 + 2 + 4 + 8 = 1111_2 = 15 = 2^4 - 1$.

⁴Der Begriff „Heap“ als Datenstruktur ist nicht zu verwechseln mit dem gleichnamigen Speicherbereich der Java Virtual Machine, in dem alle Objekte abgelegt werden!

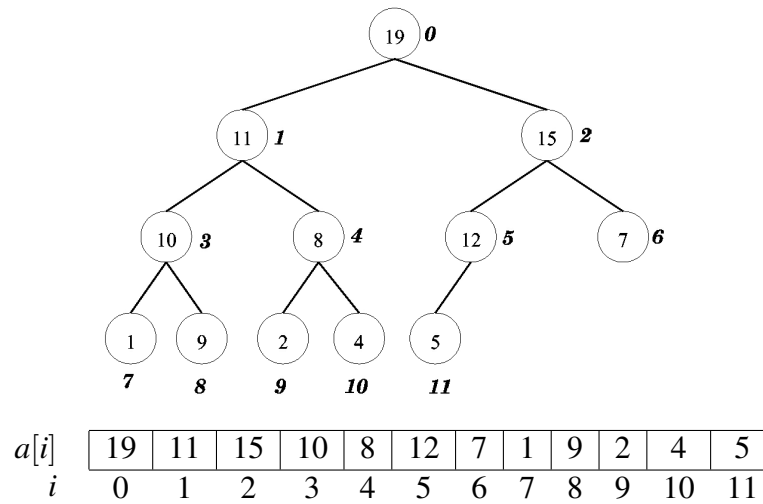


Abbildung 4.4: Ein Heap als Binärbaum (mit durchnummerierten Knoten) und implementiert als ein Array $a[i]$ der Länge 12.

Satz 4.12. Ein Array $a[i]$ lässt sich genau dann als ein (Maximum-)Heap darstellen, wenn die so genannte „Heap-Bedingung“

$$a[p(i)] \geq a[i] \quad (4.8)$$

für alle i erfüllt ist, wobei der Index $p[i]$ durch Gleichung (4.7) gegeben ist. Entsprechend kann man ein gegebenes Array $a[i]$ genau dann als einen Minimumheap darstellen, wenn die Minimumheap-Bedingung

$$a[p(i)] \leq a[i] \quad (4.9)$$

für alle i erfüllt ist.

Die Heap-Bedingungen (4.8) bzw. (4.9) lassen sich grafisch sehr leicht überprüfen, indem man den Baum von der Wurzel an generationenweise von links nach rechts mit den Array-Einträgen $a[0]$, $a[1]$, \dots , $a[n-1]$ auffüllt und dann für jeden Knoten einzeln prüft, ob er größer (bzw. kleiner) gleich als seine Kinder ist.

Wichtige Baumstrukturen in Java

Bäume gibt es in Java im Rahmen des Collection-Frameworks zwar (noch?) nicht, aber eine allgemeine Baumstruktur ist durch die Schnittstelle `TreeModel` gegeben, die im Wesentlichen aus dem Wurzelement `root` besteht, das wiederum vom Typ `TreeNode` ist und maximal ein Elternelement sowie eine Liste von Kindern hat. Wichtige Implementierungen dieser Schnittstellen sind `DefaultTreeModel` mit `DefaultMutableTreeNode`, vor allem von der Swing-Klasse `JTree` zur Darstellung von Verzeichnisbäumen verwendet werden.

4.5 Java Collections

Eine *Collection*, manchmal auch *Container* genannt, ist in Java ein Objekt, das mehrere Elemente zu einer Einheit zusammen fasst. Collections sind von der Java-API bereitgestellte Datenstrukturen, um Daten zu speichern, zu suchen, zu manipulieren und zu kommunizieren. Typischerweise stellen sie Dateneinträge dar, die eine natürliche Gruppe bilden, wie beispielsweise ein Skatblatt als eine Collection von Spielkarten, ein Postfach als Collection von Briefen oder ein Telefonbuch als eine Verknüpfung von Namen und Telefonnummern.

Merkregel 22. Bei Verwendung der Collections werden die als Typparameter auftretenden Klassen stets mit einem der vier Großbuchstaben

- E für Element,
- T für Typ,
- V für Value (Wert) und K für Key (Schlüssel)

benannt. Diese Notation wird auch in der Java-API-Dokumentation verwendet.

Merkregel 23. Daneben wird häufig die „Wildcard“ `<?>` verwendet, die ein Objekt der allgemeinsten Klasse `Object` bezeichnet. Es gibt folgende Varianten:

- `<?>` – steht für den allgemeinen Typ `Object`
- `<? extends Typ>` – steht für alle Unterklassen und die Klasse `Typ`
- `<? super Typ>` – steht für alle Superklassen von `Typ` und die Klasse `Typ` selbst

Eine Hauptschwierigkeit für den Anfänger besteht zumeist darin, die scheinbar erschlagende Fülle an bereitgestellten Interfaces und Klassen zu strukturieren. Beginnen wir mit den Interfaces, denn im Wesentlichen wird die Struktur des Java-Collection-Frameworks durch die Interfaces `Collection`,

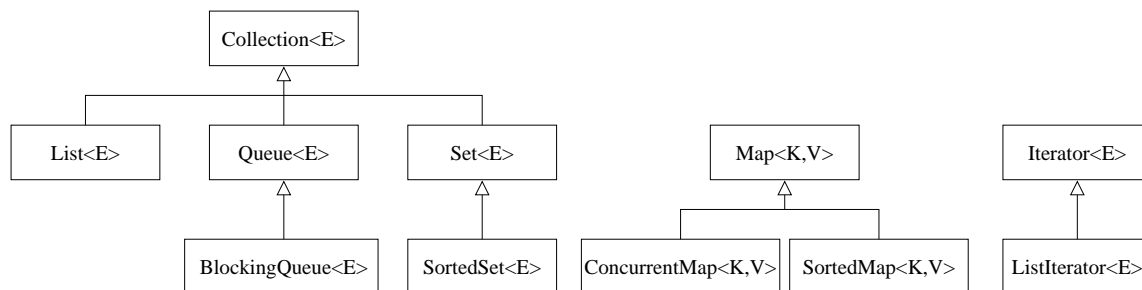


Abbildung 4.5: Wichtige Interfaces des Java Collection Frameworks. Alle befinden sich im Paket `java.util`, bis auf die sich für nebenläufige Zugriffe („Threads“) geeigneten Klassen `BlockingQueue` und `ConcurrentMap` aus dem Paket `java.util.concurrent`.

`Map` und `Iterator` bestimmt (Abbildung 4.5). Das Interface `Collection<E>` beschreibt die Struktur eines allgemeinen Containers von Objekten der Klasse `E` und verlangt u.a. die Implementierung der folgenden Methoden:

- `boolean add(E o)`: fügt das Element `o` ans Ende der Liste ein
- `void clear()`: löscht alle Elemente dieser Collection
- `boolean contains(E o)`: gibt `true` zurück, wenn diese Collection das Objekt `o` enthält
- `boolean isEmpty()`: gibt `true` zurück, wenn diese Collection kein Element enthält
- `boolean remove(E o)`: entfernt das eingegebene Element
- `int size()`: gibt die Anzahl der Element dieser Collection
- `E[] toArray()`: gibt ein Array zurück, das alle Elemente dieser Collection enthält.

Es hat als Subinterfaces `List<E>`, `Queue<E>` und `Set<E>`. Die grundlegenden Interfaces des Java-Collection-Frameworks sind in der folgenden Tabelle aufgelistet:

Interface	Erläuterungen und wichtige zu implementierende Methoden
Iterator<E> (verkettete Liste)	Die Elemente werden in einer verketteten Liste (Sequenz) gespeichert. Üblicherweise wird ein Iterator als Hilfsobjekt zum Durchlaufen der Elemente einer Collection verwendet. boolean hasNext(): prüft, ob ein weiteres Element existiert E next(): gibt das nächste Element zurück und verweist auf dessen nächstes void remove(): entfernt das aktuelle Element
List-Iterator<E> (doppelt verkettete Liste)	Die Elemente werden in einer doppelt verketteten Liste gespeichert, jedes Element hat also einen Nachfolger (next) und einen Vorgänger (previous). ListIterator ist ein Subinterface von Iterator und wird üblicherweise als Hilfsobjekt zum Durchlaufen der Elemente einer Liste (s.u.) verwendet. boolean hasPrevious(): prüft, ob ein Vorgängerelement existiert E previous(): gibt das Vorgängerelement zurück void remove(): entfernt das aktuelle Element
List<E> (indizierte Liste)	Die Elemente werden in einer indizierten verketteten Liste gespeichert und sind wahlweise direkt über einen Index oder sequentiell über einen Iterator zugreifbar. boolean add(E o): fügt das Element o ans Ende der Liste ein void add(int i, E o): fügt das Element o an die Position i der Liste ein E get(int i): gibt das Element auf Position i zurück E set(int i, E e): ersetzt das Element e an Position i in dieser Liste
Queue<E> (Warteschlange)	Die Elemente werden nach dem FIFO-Prinzip (<i>first-in, first out</i>) verarbeitet, das zuerst eingefügte Element wird auch zuerst ausgelesen. boolean offer(E o): fügt das Element o in die Warteschlange ein, wenn möglich E peek(): gibt den Kopf der Warteschlange zurück E poll(), E remove(): gibt den Kopf der Warteschlange zurück und entfernt ihn
Set<E> (Menge)	Eine Menge enthält keine doppelten Elemente, und es können die Mengenoperationen Vereinigung, Durchschnitt und Subtraktion durchgeführt werden: boolean add(E o): fügt das Element o in die Menge ein, wenn es nicht schon vorhanden ist boolean addAll(Collection<E> c): fügt alle Elemente der eingegebenen Collection c in die Menge ein, wenn sie nicht schon vorhanden sind, d.h. this ergibt $this \cup c$. boolean removeAll(Collection<E> c): löscht alle Elemente dieser Menge, die in c sind, d.h. this ergibt $this \setminus c$. boolean retainAll(Collection<E> c): behält nur diejenigen Elemente dieser Menge, die auch in c sind, d.h. this ergibt $this \cap c$.
Map<K,V> (Verknüpfung)	Es werden Schlüssel-Wert-Paare <K,V> gespeichert, wobei jeder Schlüssel eindeutig ist. Map ist kein Subinterface von Collection. V get(K key): gibt den Wert zu dem eingegebenen Schlüssel key zurück V put(K key, V value): fügt den Schlüssel key mit dem Wert value ein Set<K> keySet(): gibt eine Set mit den Schlüsseln zurück

Die Klassen des Collection-Frameworks implementieren diese Interfaces, die wichtigsten sind in Abbildung 4.6 aufgeführt. Sie stellen die verschiedenen Datenstrukturen zur Speicherung von Objekten

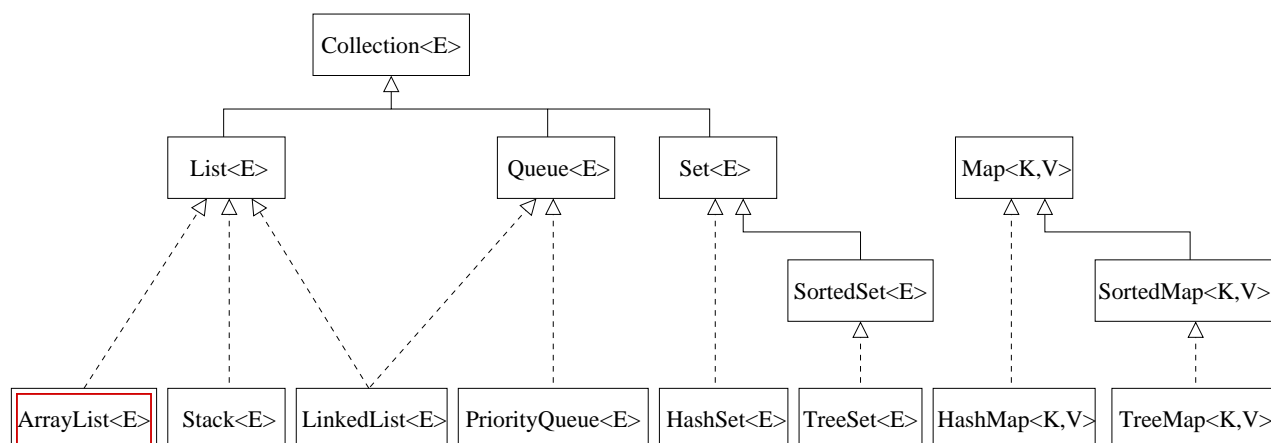


Abbildung 4.6: Wichtige Klassen des Java Collection Frameworks mit ihren Interfaces. Die für die meisten praktischen Fälle geeignete Klasse ist die ArrayList, ein „dynamisches Array“. Alle aufgeführten Klassen befinden sich im Paket java.util.

dar. Wir werden sie detaillierter in den folgenden Abschnitten betrachten.

4.5.1 Listen

Listen sind lineare Collections, oder Sequenzen. Die Elemente liegen einer bestimmten Reihenfolge vor, normalerweise in der Einfügereihenfolge. In Java haben Listen die folgenden speziellen Eigenschaften:

- Eine Liste kann doppelte Elemente enthalten.
- Ein direkter Zugriff über einen Index ist möglich, mit der Methode `get(int)`.

Die Notation einer List in der API-Dokumentation lautet `List<E>`, wobei E für den Datentyp (üblicherweise die Klasse) der Elemente steht. Die am häufigsten eingesetzten Implementierungen des Interfaces List sind nachfolgend aufgeführt:

Klasse	Erläuterungen und wichtige Methoden
Array-List<E>	Ein dynamisches Array, d.h. eine Array mit veränderbarer Größe. <code>void remove(int i)</code> : entfernt das Element an Position i dieser ArrayList und verschiebt die Position aller nachfolgenden Elemente um 1 nach links <code>void trimToSize()</code> : passt die Kapazität dieser ArrayList an die aktuelle Größe an
Linked-List<E>	Eine Datenstruktur, die sich gut eignet, wenn man oft Elemente am Anfang einfügen oder aus der Mitte löschen will. Da LinkedList auch eine Queue implementiert, eignet sie sich auch als FIFO-Speicher. <code>boolean addFirst(E o)</code> : fügt das Element o an den Anfang ein <code>E getFirst()</code> : gibt das erste Element der Liste zurück. <code>boolean offer(E o)</code> : hängt das Element o an das Ende der Liste an. <code>E peek()</code> : gibt das erste Element der Liste zurück, aber entfernt es nicht. <code>E poll()</code> : gibt das erste Element der Liste zurück und entfernt es.
Stack<E>	Ein Stapel, d.h. eine LIFO-Datenstruktur. <code>E peek()</code> : gibt das oberste (zuletzt eingefügte) Element der Liste zurück, aber entfernt es nicht. <code>E pop()</code> : gibt das oberste (zuletzt eingefügte) Element der Liste zurück und entfernt es. <code>E push(E o)</code> : legt das Element o auf den Stapel. <code>int search(E o)</code> : gibt die Position des Elements o im Stapel zurück, d.h. seinen Abstand vom Kopfende (<i>top</i>) des Stapels. Das oberste Element hat die Position 1, je tiefer o im Stapel liegt, desto höher ist seine Position.

4.5.2 Mengen

Wie in der Mathematik ist auch in Java eine Menge (*set*) eine Einheit von Elementen, die nicht doppelt vorkommen. Ein bereits vorhandenes Element wird somit mit `add` nicht erneut eingefügt, liefert allerdings `false` zurück. Die Notation einer Set in der API-Dokumentation lautet `Set<E>`, wobei E für den Datentyp (üblicherweise die Klasse) der Elemente steht. Die folgende Tabelle führt die wichtigsten Implementierungen des Interfaces Set auf.

Klasse	Erläuterungen und wichtige Methoden
HashSet<E>	Eine ungeordnete Menge, deren Speicher intern durch eine Hash-Tabelle verwaltet wird. Sie ist die schnellste Implementierung einer Menge. Da sich allerdings die (interne) Ordnung der Elemente dynamisch ändern kann, eignet sie sich nicht für geordnete Mengen. Zwei Parameter bestimmen eine HashSet, die Kapazität (<i>capacity</i>) <i>c</i> und der Ladefaktor (<i>load factor</i>) α der Hash-Tabelle, d.i. der Quotient der Anzahl der Elemente durch die Kapazität (<i>capacity</i>). Die (erwartete) Anzahl Elemente, also <code>this.size()</code> , sollte von der Größenordnung dem Produkt αc entsprechen. Als Faustregel gilt, dass bei einem Ladefaktor $\alpha = \frac{3}{4}$ die Kapazität etwa dem Doppelten der zu erwarteten Elementezahl der Menge entsprechen sollte. Die Defaultwerte bei HashSet liegen bei $\alpha = \frac{3}{4}$ und <i>c</i> = 16 (d.h. <code>size()</code> \approx 6).
TreeSet<E>	Eine geordnete Menge, implementiert das Interface SortedSet. Die interne Speicherverwaltung wird über einen Baum realisiert. Eine TreeSet ist eine langsamere Implementierung einer Set und sollte <i>nicht</i> für ungeordnete Mengen verwendet werden. <code>E first()</code> : gibt das kleinste Element der Menge zurück <code>E last()</code> : gibt das größte Element der Menge zurück. <code>SortedSet<E> subSet(E from, E to)</code> : gibt die geordnete Teilmenge einschließlich dem Element <i>from</i> und ausschließlich dem Element <i>to</i> zurück. <code>SortedSet<E> headSet(E to)</code> : gibt die Teilmenge aller Elemente der Menge $< to$ zurück. <code>SortedSet<E> tailSet(E from)</code> : gibt die Teilmenge aller Elemente der Menge $\geq from$ zurück.

Linked-HashSet<E>	Eine ungeordnete Menge, die intern die Elemente in ihrer Einfügeordnung als eine doppelt verkettete Liste verwaltet. Ist etwas langsamer als eine HashSet, aber schneller als eine TreeSet.
EnumSet<E>	Eine auf enum-Typen spezialisierte und sehr effizient implementierte Menge. Neben den üblichen Mengen-Methoden implementiert sie die folgenden statischen Methoden: static EnumSet<E> noneOf(Class<E> enumType): gibt eine EnumSet zurück, die eine leere Menge zu dem übergebenen enumTyp darstellt. static EnumSet<E> of(E e1, ..., E eN): gibt eine EnumSet zurück, die die übergebenen N Elemente enthält, wobei $N \geq 1$. Jedes Element $e1, \dots, eN$ muss dabei vom enum-Typ <code>Class<E></code> sein. static EnumSet<E> range(E from, E to): gibt eine EnumSet zurück, die Elemente von <code>from</code> bis <code>to</code> (einschließlich) des enum-Typs <code>Class<E></code> enthalten. Die Ordnung wird durch ihn festgelegt, und mit ihr muss stets <code>from</code> \leq <code>to</code> gelten.

Das folgende Java-Programm erzeugt 50 zufällige Lottotipps 6 aus 49, jeweils als 6-elementige sortierte Menge von Integer-Werten.

```
import java.util.*;

public class Lottotipp extends TreeSet<Integer> {

    public Lottotipp() {
        while( this.size() < 6 ) {
            this.add( (int) (49 * Math.random() + 1) );
        }
    }

    public static void main(String[] args) {
        Lottotipp tipp;
        for (int i = 1; i <= 50; i++) {
            tipp = new Lottotipp(); // erzeuge wieder neuen Tipp
            System.out.println("Tipp " + i + ": " + tipp);
        }
    }
}
```

Da Lottotipp also eine TreeSet aus Integern ist, können erstens keine doppelten Elemente auftreten (da er eine Menge ist) und sind zweitens die Elemente aufsteigend sortiert (da er ein Baum ist). Im Konstruktor wird bei der Erzeugung eines Objekts Lottotipp also solange eine Zufallszahl $\in \{1, 2, \dots, 49\}$ in die Menge eingefügt, bis sie 6 verschiedene Zahlen enthält.

4.5.3 Maps (Zuordnungen / Verknüpfungen)

Eine *Map* oder *Zuordnung* stellt eine Beziehung zwischen einem *Schlüssel* (*key*) und einem *Wert* (*value*) dar. In einer Map müssen die Schlüssel eindeutig sein, es kann also keine zwei gleichen Schlüssel geben, und jeder Schlüssel ist mit einem Wert verknüpft. Die Notation einer Map in der API-Dokumentation lautet

Map<K, V>,

wobei K für den Datentyp (üblicherweise die Klasse) des Schlüssels steht und V für denjenigen des Wertes.

Ein einfaches Beispiel einer Map ist ein Telefonverzeichnis, das einem Namen (Schlüssel) eine (und nur eine) Telefonnummer (Wert) zuordnet; dabei kann es durchaus vorkommen, dass zwei Namen dieselbe Telefonnummer zugeordnet ist.

Die wichtigsten zu implementierenden Methoden des Interfaces Map sind die folgenden:

- V `get(K key)`: gibt den dem Schlüssel `key` zugeordneten Wert zurück.

- `V put(K key, V value)`: ordnet dem Schlüssel `key` den Wert `value` zu.
- `V remove(K key)`: entfernt den Schlüssel `key` und seinen zugeordneten Wert aus der Map.
- `V put(K key, V value)`: ordnet dem Schlüssel `key` den Wert `value` zu.
- `int size()`: gibt die Anzahl der Schlüssel-Wert-Paare der Map zurück.
- `void clear()`: löscht alle Zuordnungen dieser Map.
- `boolean containsKey(K key)`: gibt `true` zurück, wenn die Map den Schlüssel `key` enthält.
- `boolean containsValue(V value)`: gibt `true` zurück, wenn die Map einen oder mehrere Werte `value` enthält.
- `Set<K> keySet()`: gibt eine Menge zurück, die aus allen Schlüsseln der Map besteht.
- `Collection<V> values()`: gibt eine Collection aller Werte der Map zurück.

Die beiden wichtigsten Methoden einer Map sind natürlich `put` zum Aufbau der Map, und `get` zum Auslesen des passenden Schlüssels.

Klasse	Erläuterungen und wichtige Methoden
Hash- Map<K, V>	Eine Map mit ungeordneten Schlüsseln, deren Speicher intern durch eine Hash-Tabelle verwaltet wird. Sie ist die schnellste Implementierung einer Map. Da sich allerdings die (interne) Ordnung der Elemente dynamisch ändern kann, eignet sie sich nicht für geordnete Maps. Zwei Parameter bestimmen eine <code>HashMap</code> , die Kapazität (<i>capacity</i>) c und der Ladefaktor (<i>load factor</i>) α der Hash-Tabelle, d.i. der Quotient der Anzahl der Elemente durch die Kapazität (<i>capacity</i>). Die (erwartete) Anzahl Elemente, also <code>this.size()</code> , sollte von der Größenordnung dem Produkt αc entsprechen. Als Faustregel gilt, dass bei einem Ladefaktor $\alpha = \frac{3}{4}$ die Kapazität etwa dem Doppelten der zu erwarteten Elementezahl der Map entsprechen sollte. Die Defaultwerte bei <code>HashMap</code> liegen bei $\alpha = \frac{3}{4}$ und $c = 16$ (d.h. <code>size() \approx 6</code>).
TreeMap <K, V>	Eine Map mit geordneten Schlüsseln. Sie implementiert das Interface <code>SortedMap</code> . Die interne Speicherverwaltung wird über einen Baum realisiert. Eine <code>TreeMap</code> ist eine langsamere Implementierung einer Map und sollte <i>nicht</i> für ungeordnete Maps verwendet werden. <code>K firstKey()</code> : gibt den kleinsten Schlüssel dieser Map zurück <code>K lastKey()</code> : gibt den größten Schlüssel der Map zurück. <code>SortedMap<K, V> subSet(K from, K to)</code> : gibt die geordnete Teil-Map einschließlich dem Schlüssel <code>from</code> und ausschließlich dem Schlüssel <code>to</code> zurück. <code>SortedMap<K, V> headMap(K to)</code> : gibt die Teil-Map aller Schlüssel der Map <code>< to</code> zurück. <code>SortedMap<K, V> tailMap(K from)</code> : gibt die Teilmenge aller Schlüssel der Menge <code>\geq from</code> zurück.
Linked- HashMap <K, V>	Eine Map mit nicht geordneten Schlüsseln, die intern die Schlüssel in der Reihenfolge ihres Einfügens als eine doppelt verkettete Liste verwaltet. Ist etwas langsamer als eine <code>HashMap</code> , aber schneller als eine <code>TreeMap</code> .
Enum- Map<K, V>	Eine auf Schlüssel eines enum-Typs spezialisierte und sehr effizient implementierte Map.

Ein einfaches Programm für ein Telefonbuch lautet:

```
import java.util.*;
import javax.swing.*;

public class Telefonbuch extends TreeMap<String, Integer> {
    public String ort;

    public Telefonbuch(String ort) {
        this.ort = ort;
    }

    public String toString() {
        return "Telefonbuch " + ort + ":\n" + super.toString();
    }
}
```

```

}

public static void main( String[] args ) {
    Telefonbuch hagen = new Telefonbuch("Hagen");

    //Eintragen:
    hagen.put("Schröder", 2380);
    hagen.put("Weiß", 2371);
    hagen.put("de Vries", 2381);

    JOptionPane.showMessageDialog(
        null, hagen + "\nTel. Weiß:" + hagen.get("Weiß")
    );
}
}

```

4.5.4 Wann welche Klasse verwenden?

Bei der Überlegung, welche der Klassen aus Abbildung 4.6 als Datenstruktur für ein gegebenes Problem geeignet ist, sollte man sich zunächst über das zu verwendende Interface aus Abbildung 4.5 klar werden. Im Wesentlichen muss man sich also gemäß folgender Tabelle entscheiden.

Interface	Kriterien
List<E>	Speicherung als Sequenz, mehrfaches Auftreten gleicher Elemente möglich, indizierter Zugriff
Queue<E>	Speicherung als Sequenz gemäß dem FIFO-Prinzip (Warteschlange, <i>first-in, first-out</i>)
Set<E>	Speicherung als Menge, d.h. jedes Element kann maximal einmal auftreten
Map<K,V>	Speicherung von Schlüssel-Wert-Paaren, wobei ein Schlüssel in der Map eindeutig ist (d.h. alle Schlüssel ergeben eine Set!)

Nach Abbildung 4.6 ergeben sich daraus die konkreten Alternativen der für die jeweilige Problemstellung geeigneten Klasse. Grob kann man dabei von folgenden Daumenregeln ausgehen:

- In den meisten Fällen wird die `ArrayList` die geeignete Datenstruktur sein, also eine lineare Sequenz mit indiziertem Zugriff, oder in anderen Worten ein dynamisches Array.
- Benötigt man entweder eine `Queue` oder eine lineare Liste und kann absehen, dass Elemente aus ihrem Innern oft gelöscht werden müssen oder vorwiegend Durchläufe durch die gesamte Liste stattfinden werden, so ist eine `LinkedList` zu bevorzugen, deren Operationen sind schneller als in einer `ArrayList` (die ist schneller im indizierten Zugriff).
- Will man eine `Set` implementieren, dann wird in den meisten Fällen eine unsortierte `HashSet` die geeignete Wahl sein, sie ist in der Verarbeitung schneller als eine sortierte `TreeSet`.
- Will man eine `Map` implementieren, dann wird in den meisten Fällen eine unsortierte `HashMap` die geeignete Wahl sein, sie ist in der Verarbeitung schneller als eine sortierte `TreeSet`.

4.6 Statische Methoden der Klasse `Collections`

Die Klasse `Collections` (man beachte das „s“ am Ende!) enthält nur statische Methoden, durch die einige nützliche Algorithmen für die Datenstrukturen bereitgestellt werden. Sie ist damit gewissermaßen der „Werkzeugkasten“ für das `Collection`-Framework. Diese Methoden sind „polymorph“, d.h.

sie können für verschiedene Implementierungen eines Interfaces verwendet werden. Wichtige dieser Methoden sind:

- `static <T> int binarySearch(List<T extends Comparable<T> list, T key)`, bzw. `static <T> int binarySearch(List<T> list, T key, Comparator<T> c)`: sucht nach dem Objekt `key` in der sortierten Liste `list`. die Liste muss nach dem durch `Comparable`, bzw. dem `Comparator` gegebenen Sortierkriterium sortiert sein.
- `static int frequency(Collection<? c, Object o)` gibt Anzahl der Elemente der Collection `c` zurück, die gleich dem spezifizierten Objekt `o` sind.
- `static <T> List<T> emptyList()` gibt die leere Liste zurück.
- `static <T> Set<T> emptySet()` gibt die leere Menge zurück.
- `static <K,V> Map<K,V> emptyMap()` gibt die leere Map zurück.
- `static <T> max(Collection<? extends T> coll[, Comparator<? super T> comp])` gibt das maximale Element der Collection `coll` gemäß der natürlichen Ordnung der Elemente zurück [, bzw. gemäß des übergebenen Comparators `comp`].
- `static <T> min(Collection<? extends T> coll[, Comparator<? super T> comp])` gibt das minimale Element der Collection `coll` gemäß der natürlichen Ordnung der Elemente zurück [, bzw. gemäß des übergebenen Comparators `comp`].
- `static void sort(List<T> list[, Comparator<? super T> comp])` sortiert die Elemente der Liste `list` gemäß der natürlichen Ordnung der Elemente [, bzw. gemäß des übergebenen Comparators `comp`].

Daneben gibt es Methoden zum Ersetzen (`replaceAll`) von Listenelementen, zum Umkehren der Reihenfolge (`reverse`), zum Rotieren (`rotate`, eine Art „modulo-Verschiebung“), Mischen (`shuffle`) von Listenelementen. und Vertauschen (`swap`) von Listenelementen.

4.7 Zusammenfassender Überblick

Wir haben in diesem Kapitel grundlegende Datenstrukturen kennen gelernt, zunächst die theoretischen Konzepte einer verketteten Liste als lineare Datenstruktur, sowie Bäume als nichtlineare Datenstruktur. Während sich verkettete Listen gut eignen, um Datenelemente einzufügen oder zu entfernen, eignen sich Bäume zur sortierten Speicherung von Daten.

Im Java Collection Framework werden diese und weitere Konzepte umgesetzt. Die Java Collections umfassen Listen, Queues, Mengen und Maps. Hierbei ist eine Liste in Java stets eine „indizierte“ Liste, d.h. man kann sie sowohl über einen Iterator als klassische verkettete Liste darstellen als auch wie bei einem Array über den Index (mit `get(i)`) auf sie zugreifen.

Es gibt jeweils zwei Arten von Implementierungen von Mengen (Sets) und Maps in Java, einerseits mit unsortierter Speicherung mit Hash-Tabelle (`HashSet`, `HashMap`) oder mit sortierter Speicherung mit Hilfe eines Baumes (`TreeSet`, `TreeMap`).

Abschließend vergleichen wir verschiedene Datenstrukturen mit demjenigen Element, das jeweils am effizientesten zu finden ist:

Datenstruktur	am schnellsten zu findendes Element
Array(List)	das Element nach gegebener Position („random access“)
Stack	das neueste Element
Queue	das älteste Element
(Maximum-) Heap	das größte Element
Minimumheap	das kleinste Element

Anhang A

Spezielle Themen

A.1 ASCII und Unicode

Zeichen werden im Computer durch Zahlen dargestellt. Das geschieht mit Hilfe von Codes.

Ein *Code* ist eine eindeutige Zuordnung eines Zeichenvorrats („Alphabet“) in einen anderen Zeichenvorrat.

In diesem Sinne hat ein Code also nichts mit Verschlüsselung zu tun, denn er ist nicht geheim.¹ Bekannte Beispiele für Codes sind der ASCII-Code und der Unicode:

- ASCII-Code (ASCII = *American Standard Code for Information Interchange*)

ASCII-Zeichensatz										
+	0	1	2	3	4	5	6	7	8	9
30				!	"	#	\$	%	&	'
40	()	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[\]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{		}	~			

(§ Restricted Use)

Diese Tabelle ist wie folgt zu lesen: Nimm die Zahl vor der Zeile und addiere die Zahl über der Spalte; der Schnittpunkt ist der dargestellte Buchstabe. Beispiel: J = 74.

- Unicode ist ein 16-bit-Code, d.h. er besteht aus $2^{16} = 65\,536$ Zeichen. Er umfasst die Schriftzeichen aller Verkehrssprachen der Welt. Ein Auszug ist in Abb. A.1 zu sehen. Die Tabellen sind wie folgt zu lesen: Nimm die Zeichenfolge der Spalte und hänge das Zeichen vor jeder Zeile dahinter; der Schnittpunkt ist der dargestellte Buchstabe. Beispiel: J = 004A. (004A ist eine Hexadezimalzahl und bedeutet hier somit 74; der ASCII-Code ist also im Unicode enthalten.) Weitere Informationen können Sie unter dem Link „Unicode“ bei <http://haegar.fh-swf.de> finden.

A.2 Das Binärsystem

Welche Möglichkeiten gibt es, Zahlen darzustellen? Die uns geläufige Darstellung von Zahlen ist das so genannte *Dezimalsystem* (oder „Zehnersystem“). Es hat als so genannte *Basis* die Zahl 10 und besteht aus einem Ziffernvorrat 0, 1, ..., 9, oder in Symbolen:

¹In der Kryptologie spricht man von „Chiffre“

0000	C0 Controls and Basic Latin								007F	0080	C1 Controls and Latin-1 Supplement								00FF
	000	001	002	003	004	005	006	007			008	009	00A	00B	00C	00D	00E	00F	
0	[NUL]	[DLE]	[SP]	0	@	P	~	p			0	[XXX]	[DCS]	[MPE]	◊	À	Ä	à	ä
1	[SOH]	[DC1]	!	1	A	Q	a	q			1	[XXX]	[PU1]	±	Á	Ñ	á	ñ	
2	[STX]	[DC2]	"	2	B	R	b	r			2	[BPH]	[PU2]	¢	Â	Ô	â	ô	
3	[ETX]	[DC3]	#	3	C	S	c	s			3	[MHC]	[STB]	£	Ã	Ó	ã	ó	
4	[EOT]	[DC4]	\$	4	D	T	d	t			4	[IND]	[CCH]	¤	Ä	Ö	ä	ö	
5	[ENQ]	[NAK]	%	5	E	U	e	u			5	[NEL]	[MW]	¥	Å	Ø	å	ø	
6	[ACK]	[SYN]	&	6	F	V	f	v			6	[SSA]	[SPA]	¦	Æ	Ö	æ	ö	
7	[BEL]	[ETB]	'	7	G	W	g	w			7	[ESA]	[EPA]	§	•	Ç	×	÷	
8	[BS]	[CAN]	(8	H	X	h	x			8	[HTJ]	[XXX]	¨	È	Ø	è	ø	
9	[HT]	[EM])	9	I	Y	i	y			9	[HTJ]	[XXX]	©	É	Ù	é	ù	
A	[LF]	[SUB]	*	:	J	Z	j	z			A	[VTS]	[SC1]	®	Ê	Ú	ê	ú	
B	[VT]	[ESC]	+	;	K	[k	{			B	[PLD]	[SCB]	<	Ë	Û	ë	û	
C	[FF]	[FS]	,	<	L	\	l				C	[PLM]	[ST]	¼	Ï	Ü	ï	ü	
D	[CN]	[GS]	=	-	M]	m	}			D	[RI]	[SNC]	½	Í	Ý	í	ý	
E	[SO]	[RB]	>	N	^	n	~				E	[SBI]	[PM]	¾	Î	Þ	î	þ	
F	[SI]	[US]	/	?	O	_	o	[DEL]			F	[SSJ]	[JPC]	ˆ	İ	ß	ï	ÿ	

Abbildung A.1: Die ersten Tabellen des Unicodes.

Basis = 10, Ziffernvorrat = {0, 1, 2, . . . , 8, 9}

Mit diesen 10 Ziffern kann man jede beliebige Zahl darstellen, entscheidend ist die Stelle, an der eine Ziffer steht. Die (von rechts gesehen) erste Ziffer mit der Potenz steht für das Vielfache von 10^0 , die zweite für das von 10^1 , . . . , und die j -te für das Vielfache von 10^{j-1} . Beispielsweise heißt das für die Ziffernfolge 409:

$$\begin{array}{rcl}
 & 409 & \\
 \text{---} & \text{---} & 9 \cdot 10^0 = 9 \\
 \text{---} & \text{---} & + 0 \cdot 10^1 = 0 \\
 \text{---} & \text{---} & + 4 \cdot 10^2 = 400 \\
 \hline
 & & 409_{10}
 \end{array}$$

Addiert man die Produkte der Ziffern mit ihren jeweiligen 10er-Potenzen, so erhält man wieder 409 — wobei wir hier 409_{10} schreiben, um zu verdeutlichen, dass unsere Basis die 10 ist.

Allein, was hindert uns eigentlich daran, die Basis zu ändern? Warum muss es die 10 sein? Versuchen wir doch einfach einmal, uns eine Zahldarstellung zur Basis 2 zu konstruieren. Das hat natürlich direkten Einfluss auf unseren Ziffernvorrat; hatten wir zur Basis 10 auch 10 Ziffern zur Verfügung, so haben wir zur Basis 2 eben auch nur zwei Ziffern zur Verfügung:

Basis = 2, Ziffernvorrat = {0, 1}

Damit erhalten wir das *Dual*- oder *Binärsystem* (von dem Lateinischen bzw. Griechischen Wort für zwei). Um nun eine Zahl im Binärsystem, z.B. 10_2 , im Dezimalsystem darzustellen, müssen wir wie oben vorgehen, allerdings jetzt mit 2 statt 10:

$$\begin{array}{rcl}
 & 10_2 & \\
 \text{---} & \text{---} & 0 \cdot 2^0 = 0 \\
 \text{---} & \text{---} & + 1 \cdot 2^1 = 2 \\
 \hline
 & & 2_{10}
 \end{array}$$

Mit anderen Worten, 10_2 im Binärsystem entspricht der 2_{10} im Dezimalsystem. Entsprechend berechnen wir 110011001_2 :

$$\begin{array}{r}
 110011001_2 \\
 \begin{array}{l}
 1 \cdot 2^0 = 1 \\
 + 0 \cdot 2^1 = 0 \\
 + 0 \cdot 2^2 = 0 \\
 + 1 \cdot 2^3 = 8 \\
 + 1 \cdot 2^4 = 16 \\
 + 0 \cdot 2^5 = 0 \\
 + 0 \cdot 2^6 = 0 \\
 + 1 \cdot 2^7 = 128 \\
 + 1 \cdot 2^8 = 256 \\
 \hline
 409_{10}
 \end{array}
 \end{array}$$

Ein von dieser Darstellung abgeleitetes und effizienteres Verfahren zur Umrechnung der Binärdarstellung einer ganzen Zahl in das Dezimalsystem ist das folgende: Multipliziere die erste Ziffer (von links) der umzuwandelnden Zahl mit der Basis 2, addiere dazu die nächste Ziffer, multipliziere diese Summe mit 2, usw. . . . Für 110011001_2 ergibt das z. B.

$$\begin{array}{r|cccccccccc}
 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
 + & & & & & & & & & \\
 \hline
 & 1 & 2 & 6 & 12 & 24 & 50 & 102 & 204 & 408 \\
 & 1 & 3 & 6 & 12 & 25 & 51 & 102 & 204 & \boxed{409}
 \end{array} \quad (A.1)$$

Insbesondere für größere Zahlen ist das Verfahren recht schnell. Das Binärsystem ist das einfachste Zahlssystem.² Ein weiteres in der Informatik sehr gebräuchliches Zahlssystem ist das *Hexadezimalsystem*:

$$\text{Basis} = 16, \quad \text{Ziffernvorrat} = \{0, 1, 2, \dots, 8, 9, A, B, C, D, E, F\}.$$

Da nun ein Vorrat von 16 Ziffern benötigt wird, weicht man für die (im Dezimalsystem) zweistelligen Zahlen auf das Alphabet aus. Die Umwandlung einer Hexadezimal- in eine Dezimalzahl geschieht wie gehabt. In Java werden Hexadezimalzahlen in der Notation $0x3A = 3A_{16}$ geschrieben. Beispielsweise ergibt $0x3A$ oder $0x199$:

$$\begin{array}{r}
 0x3A \\
 \begin{array}{l}
 10 \cdot 16^0 = 10 \\
 + 3 \cdot 16^1 = 48 \\
 \hline
 58_{10}
 \end{array}
 \end{array}
 \quad
 \begin{array}{r}
 0x199 \\
 \begin{array}{l}
 9 \cdot 16^0 = 9 \\
 + 9 \cdot 16^1 = 144 \\
 + 1 \cdot 16^2 = 256 \\
 \hline
 409_{10}
 \end{array}
 \end{array}$$

Wir können auch für Hexadezimalzahlen das Schema (A.1) zur Umrechnung ins Dezimalsystem anwenden, nur dass wir nun mit 16 statt mit 2 multiplizieren müssen. So erhalten wir beispielsweise die folgenden Entsprechungen.

Dualsystem	Oktalsystem	Hexadezimalsystem	Dezimalsystem
0b1	01	0x1	1
0b1001010	0112	0x4A	74
0b110011001	0631	0x199	409

(Hierbei bedeutet ein führendes 0b in Java, dass die Zahl im Binärsystem dargestellt ist, eine führende 0 allein dagegen im Oktalsystem, also zur Basis 8!) Nun wissen wir auch, was die Codierung des Unicode bedeutet. Insbesondere ist er eine Verallgemeinerung des ASCII-Codes, wie man z.B. an „J“ erkennt, das im Unicode 004A lautet, also von Hexadezimal- in Dezimaldarstellung gebracht 74, genau wie im ASCII-Code.

²Ein „Unärsystem“ kann es nicht geben, da alle Potenzen der Basis 1 wieder 1 ergeben: $1^j = 1$ für alle j .

A.2.1 Umrechnung vom Dezimal- ins Binärsystem

Wie berechnet man die Binärdarstellung einer gegebenen Zahl im Dezimalsystem? Wir benötigen dazu zwei Rechenoperationen, die *ganzzahlige Division*³ ($()$) und die *Modulo-Operation* ($\%$). Für zwei ganze Zahlen m und n ist die ganzzahlige Division m/n definiert als die größte ganze Zahl, mit der n multipliziert gerade noch kleiner gleich m ist. Einfach ausgedrückt: m/n ist gleich $m \div n$ ohne die Nachkommastellen. Beispielsweise ist

$$13/2 = 6.$$

$m \% n$ ist der Rest der ganzzahligen Division von m durch n , also z.B.

$$13 \% 6 = 1.$$

Insgesamt gilt also

$$m \div n = (m/n) \text{ Rest } (m \% n), \quad (\text{A.2})$$

z.B. $13 \div 2 = 6$ Rest 1. Einige Beispiele:

$$14/3 = 4, \quad 14 \% 3 = 2,$$

$$14/7 = 2, \quad 14 \% 7 = 0,$$

$$14/1 = 14, \quad 14 \% 1 = 0,$$

$$3/5 = 0, \quad 3 \% 5 = 3.$$

Die Umrechnung einer Zahl z von Dezimal- in Binärdarstellung geschieht nach folgender Vorschrift: Erstelle eine Tabelle mit den Spalten z , $z/2$ und $z \% 2$; setze in jeder Spalte jeweils für z den Wert der zu berechnenden Zahl ein; nimm als neuen Wert die Zahl aus der Spalte $z/2$, solange diese echt größer als 0 ist, und wiederhole den Vorgang in der nächsten Zeile. Das Verfahren endet also mit der Zeile, in der in der mittleren Spalte eine 0 steht. Die Binärdarstellung ergibt sich, wenn man die Zahlen in der Spalte $z \% 2$ von unten nach oben aneinander reiht. Berechnen wir z.B. $z = 13$.

z	$z/2$	$z \% 2$	z	$z/2$	$z \% 2$	z	$z/2$	$z \% 2$	z	$z/2$	$z \% 2$
13	6	1	13	6	1	13	6	1	13	6	1
			6			6	3	0	6	3	0
									3	1	1
									1	0	1

Es gilt also $13_{10} = 1101_2$. Entsprechend ergibt sich die Hexadezimaldarstellung aus einer Dezimalzahl, wenn man statt „/ 2“ und „% 2“ stets „/ 16“ und „% 16“ schreibt (und rechnet).

A.2.2 Binärbrüche

Die Berechnung von Binärbrüchen, also binären Kommazahlen, ist ebenso möglich wie diejenige von ganzen Zahlen, allerdings sind die Umrechnungsschemata etwas anders.

Betrachten wir zunächst die Umrechnung von der Binärdarstellung in die Dezimaldarstellung. Auch hier folgt alles aus der Position einer Ziffer, nur werden anstatt der Potenzen 2^k nun die Brüche 2^{-k} , also $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, ... entsprechend der Nachkommastelle k verwendet. Beispielsweise ist

$$0,101_2 = \frac{1}{2} + \frac{1}{8} = \frac{5}{8} = 0,625. \quad (\text{A.3})$$

³Wir verwenden hier das durch die Programmiersprache C populär gewordene Zeichen „/“, das für Integer-Zahlen genau die ganzzahlige Division ergibt.

Für die Umrechnung der Dezimaldarstellung in die Binärdarstellung verwendet man wie im Falle der ganzen Zahlen eine Tabelle, jedoch diesmal mit den Spalten $z - \lfloor z \rfloor$, $2z$ und $\lfloor 2z \rfloor$. Hierbei heißen die Klammern $\lfloor \cdot \rfloor$ die *untere Gaußklammer* oder im Englischen die *floor-brackets*. Für eine reelle Zahl x bezeichnet $\lfloor x \rfloor$ die größte ganze Zahl kleiner oder gleich x , oder formaler:

$$\lfloor x \rfloor = \max\{n \in \mathbb{Z} \mid n \leq x\} \quad (\text{A.4})$$

Die unteren Gaußklammern $\lfloor x \rfloor$ einer positiven Zahl $x > 0$ bewirken also einfach, dass ihre Nachkommastellen abgeschnitten werden, d.h. $\lfloor \pi \rfloor = 3$, $\lfloor 3 \rfloor = 3$. Für negative Zahlen bewirken sie jedoch, dass stets aufgerundet wird, beispielsweise $\lfloor -5,43 \rfloor = -6$ oder $\lfloor -\pi \rfloor = -4$.

Damit ergibt sich der folgende Algorithmus zur Berechnung der Binärdarstellung einer gegebenen positiven Zahl z in ihre Dezimaldarstellung.

1. Erstelle eine Tabelle mit den Spalten $z - \lfloor z \rfloor$, $2z$ und $\lfloor 2z \rfloor$.
2. Setze jeweils für z den Wert der zu berechnenden Zahl ein und berechne die Spaltenwerte.
3. Nimm als neuen Wert für z die Zahl aus der Spalte $2z$, wenn diese nicht 0 ist, und wiederhole den Vorgang bei Schritt 2.

Die Binärdarstellung ergibt sich, wenn man die Zahlen in der Spalte $\lfloor 2z \rfloor$ *von oben nach unten* aneinander reiht. Beispielsweise ergeben sich für $z = 0,625$ oder $z = 0,3$ die folgenden Tabellen.

$z \leftarrow z - \lfloor z \rfloor$	$2z$	$\lfloor 2z \rfloor$
0.625	1.25	1
0.25	0.5	0
0.5	1.0	1
0		

$z \leftarrow z - \lfloor z \rfloor$	$2z$	$\lfloor 2z \rfloor$
0.3	0.6	0
0.6	1.2	1
0.2	0.4	0
0.4	0.8	0
0.8	1.6	1
0.6	1.2	1
...

Damit gilt $0,625_{10} = 0,101_2$ und $0,3_{10} = 0,0\overline{1001}_2$. Die Rückrichtung als Probe gemäß Gleichung (A.3) bestätigt dies. Wie im Dezimalsystem gibt es periodische Brüche (mit Periode ungleich 0) also auch im Binärsystem.

Die einzigen Brüche, die eine endliche Binärbruchentwicklung haben, haben die Form $\frac{n}{2^k}$, oder als Dezimalbruch $0,5^k \cdot n$, für $n, k \in \mathbb{N}$. Damit haben die „meisten“ Zahlen mit endlicher Dezimalbruchentwicklung eine unendliche Binärbruchentwicklung, wie $0,3_{10} = 0,0\overline{1001}_2$. Mathematisch liegt das daran, dass 5 ein Teiler von 10 ist, 5 und 2 aber teilerfremd sind.

A.3 javadoc, der Dokumentationsgenerator

javadoc ist ein Programm, das aus Java-Quelltexten Dokumentationen im HTML-Format erstellt. Dazu verwendet es die öffentlichen Deklarationen von Klassen, Interfaces, Attributen und Methoden und fügt zusätzliche Informationen aus eventuell vorhandenen Dokumentationskommentaren hinzu. Zu jeder Klassendatei `xyz.java` wird eine HTML-Seite `xyz.html` generiert, die über verschiedene Querverweise mit den anderen Seiten desselben Projekts in Verbindung steht. Zusätzlich generiert javadoc diverse Index- und Hilfsdateien, die das Navigieren in den Dokumentationsdateien erleichtern. Aufruf von der Konsole:


```
javadoc [ options ] { package | sourcefile }
```

Will man z.B. nicht nur die öffentlichen Deklarationen dokumentieren, sondern auch die privaten der Klasse `Test.java`, so verwendet man die Option `-private`, also

```
javadoc -private Test.java
```

Mit der Option `-subpackages` kann man alle Klassen und Unterpakete eines Pakets einbinden. Möchte man zusätzlich alle verwendeten Klassen der Java API einbinden, so kann man zur gewünschten Version mit dem Argument `-link` verlinken, beispielsweise für Java 8 durch

```
javadoc -link http://docs.oracle.com/javase/8/docs/api/ -subpackages paket.pfad
```

Ein einfacher Aufruf `javadoc` liefert eine Liste der möglichen Optionen.

A.3.1 Dokumentationskommentare

Bereits ohne zusätzliche Informationen erstellt `javadoc` aus dem Quelltext eine brauchbare Beschreibung aller Klassen und Interfaces. Durch das Einfügen von Dokumentationskommentaren (`/** ... */`) kann die Ausgabe zusätzlich bereichert werden. Er muss im Quelltext immer unmittelbar vor dem zu dokumentierenden Item plziert werden (einer Klassendefinition, einer Methode oder einer Instanzvariable). Er kann aus mehreren Zeilen bestehen. Der Text bis zu einem ersten auftretenden Punkt in dem Kommentar wird später als Kurzbeschreibung verwendet, der Rest des Kommentars erscheint in der Langbeschreibung.

Zur Erhöhung der Übersichtlichkeit darf am Anfang jeder Zeile ein Sternchen stehen, es wird später ignoriert. Innerhalb der Dokumentationskommentare dürfen neben normalem Text auch HTML-Tags vorkommen. Sie werden unverändert in die Dokumentation übernommen und erlauben es damit, bereits im Quelltext die Formatierung der späteren Dokumentation vorzugeben. Die Tags `<h1>` und `<h2>` sollten allerdings möglichst nicht verwendet werden, da sie von `javadoc` selbst zur Strukturierung der Ausgabe verwendet werden.

`javadoc` erkennt des weiteren markierte Absätze innerhalb von Dokumentationskommentaren. Die Markierung muss mit dem Zeichen `@` beginnen und — abgesehen von Leerzeichen — als erstes (von `*` verschiedenes) Zeichen einer Zeile stehen. Jede Markierung leitet einen eigenen Abschnitt innerhalb der Beschreibung ein, alle Markierungen eines Typs müssen hintereinander stehen.

Die folgende Klasse ist ein kleines Beispiel, in dem die wichtigsten Informationen für eine Dokumentation angegeben sind.

```
1  /**
2   * Diese Klasse ist nur ein Test. Ab hier beginnt die Langbeschreibung.
3   *
4   * @author de Vries
5   * @version 1.0
6   */
7  public class JavadocTest {
8      /** Das erste Attribut. Ab hier beginnt die Langbeschreibung.*/
9      public long attribut;
10
11     /** Errechnet  $a^m \bmod n$ .
12      * Dazu wird  $a^m$ -mal  $a$  modulo  $n$  potenziert.
13      * @param a die Basis
14      * @param m der Exponent
15      * @param n der Modulus
16      * @return  $a^m \bmod n$ 
17      * @see #modPow(long,int,long)
18      * @throws IllegalArgumentException wenn Modulus gleich Null
19      */
20     public long modPow (long a, int m, long n) {
```

```

21     if (n == 0) throw new IllegalArgumentException("Teiler ist 0");
22     long y = 1;
23     for (int i = 0; i < m; i++) {
24         y = (y * a) % n;
25     }
26     return y;
27 }
28 }

```

Ein Aufruf mit `javadoc Javadoc.java` liefert das komplette Javadoc. Möchte man für diese Klasse neben den Beschreibungen auch die Angaben von Autor und Version dokumentieren, so ruft man auf:

```
javadoc -author -version JavadocTest.java
```

A.4 jar-Archive: Ausführbare Dateien und Bibliotheken

Man kann Java-Programme in eine Datei komprimieren und somit ganze Programmbibliotheken erstellen, die mit einer kleinen Zusatzinformation versehen auch durch Klicken ausführbar sind. Die resultierenden komprimierten Dateien heißen *Jar-Dateien* oder *Jar-Archive*.

Die allgemeine Syntax zum Anlegen von Jar-Dateien lautet

```
jar cvf Archivname.jar Dateien
```

Beispielsweise erstellt man ein Jar-Archiv mit den beiden Class-Dateien `MeinProg.class` und `Fenster.class` durch

```
jar cvf Demo.jar MeinProg.class Hilfe.class
```

Man kann mit Hilfe des Platzhalters (*wildcard*) `*` auch alle Dateien in das Archiv packen, die eine bestimmte Endung haben, also beispielsweise

```
jar cvf Demo.jar paket/*.class
```

für alle `.class`-Dateien im Verzeichnis `paket`. In ein Jar-Archiv können prinzipiell beliebige Dateien gepackt werden, also auch Textdateien (z.B. für Konfigurationen), Grafikdateien oder Icons. Die Dateien sollten sich in einem Unterverzeichnis desjenigen Paketverzeichnisses befinden, in dem auch die sie verwendenden Java-Klassen liegen.

Der Befehl zur Erzeugung eines Jar-Archivs mit allen Klassen des Verzeichnisses `paket` und allen JPG-Dateien im Unterverzeichnis

```
jar cvf meins.jar paket/*.class paket/icons/*.jpg
```

A.4.1 Manifest-Datei

Für Java-Anwendungen, die aus einem Jar-Archiv gestartet werden kann, muss man noch eine so genannte *Manifest-Datei* bereitstellen. In dieser Datei befinden sich Zusatzinformationen zum Archiv, unter anderem der Name der Startklasse, also die Klasse mit der `main`-Methode. Hierzu wird im Projektverzeichnis ein Unterverzeichnis `meta-inf` an und darin eine Textdatei `manifest.mf` mit der folgenden Zeile:

```
Main-Class: package.Startklasse
```

Die Startklasse wird hier mit vollständigem Paketpfad und ohne Dateiendung angegeben, genau so, wie sie per Konsolenbefehl über den Interpreter `java` aufgerufen würde. Anschließend können Sie das Archiv erzeugen, wobei Sie durch das Flag `m` signalisieren, dass eine Manifest-Datei mitgegeben werden soll:

```
jar cvmf Demo.jar meta-inf/Manifest.mf MeinProg.class Hilfe.class
```

Neben .class-Dateien auch komplette Jar-Archive hinzufügen. Allerdings wird dann eine erweiterte Manifest-Datei erwartet, bei der das Jar-Archiv im Parameter Class-Path definiert wird. Wenn beispielsweise das Archiv jdom.jar dazugehören soll, dann muss die Manifest-Datei folgendermaßen aussehen:

```
Main-Class: MeinProg
Class-Path: jdom.jar
```

Der Aufruf zum Erzeugen des Archivs lautet dann:

```
jar cmf Demo.jar meta-inf/Manifest.mf MeinProg.class Hilfe.class jdom.jar
```

Wenn Sie keine ablauffähige Anwendung verpacken wollen, sondern vielmehr eine Sammlung von Klassen, die von anderen Java-Programmen aufrufbar sein sollen, müssen Sie die Null-Option (0) beim Erzeugen des jar-Archivs verwenden:

```
jar cmf0 Demo.jar meta-inf/manifest.mf MeinProg.class Hilfe.class jdom.jar
```

Abschließend angegeben sei hier nun ein typischer Befehl zur Erzeugung einer ablauffähigen Applikation mit Manifestdatei und Icons, deren Klassen und Dateien sich allesamt im Verzeichnis paket befinden:

```
jar cvfm meins.jar paket/meta-inf/manifest.mf paket/*.class paket/icons/*.jpg
```

A.5 Formatierte Ausgabe mit HTML

In Java ist mit der JOptionPane-Klasse (sowie den meisten anderen Swing-Klassen) eine formatierte Ausgabe mit HTML möglich. HTML ist eine recht einfache Formatierungssprache, in der Webdokumente geschrieben sind und die von einem Browser grafisch interpretiert wird. Grundbausteine von HTML sind die sogenannten *Tags*, die stets paarweise auftreten ein öffnender Tag `<xyz>` von seinem schließenden Partner `</xyz>` geschlossen wird. Dazwischen können Text oder weitere Tags stehen, wobei jedoch streng darauf geachtet werden muss, dass die jeweils inneren Tags vor den äußeren wieder geschlossen werden. Die Tags kann man also ansehen wie Klammern in der Mathematik.

Um mit der showMessageDialog-Methode von JOptionPane eine Ausgabe in HTML zu erzeugen, verwendet man am besten eine String-Variable `ausgabe`, die mit `"<html>"` initialisiert wird und mit `"</html>"` endet, also beispielsweise

```
1 String ausgabe = "<html>";
2 ausgabe += "Das ist jetzt <i>HTML</i>!";
3 ausgabe += "</html>";
4 JOptionPane.showMessageDialog(null, ausgabe);
```

Zu beachten ist, dass zum Zeilenumbruch nun keine Unicode-Angabe `"\n"` mehr möglich ist, sondern der HTML-Tag `"
"` (für *line break*) verwendet werden muss (bewirkt in HTML einen Zeilenumbruch). Er ist einer der wenigen „leeren“ Tags in HTML, der keinen schließenden Tag benötigt.⁴

⁴ Nach XHTML muss der Zeilenumbruch-Tag `
` lauten, doch leider interpretiert Java diesen Tag (noch?) nicht.

Bezüge	Steuerstufe	Steuern in %
866.99	1	0.1
1400.99	2	8.5
1901.99	3	13.9
2402.99	4	17.3
2900.99	5	20.2

Wie die obige Abbildung zeigt, können in HTML auch Tabellen erzeugt werden. Das geschieht durch die folgende Konstruktion:

```
<table border="1">
  <tr>
    <th> ... </th>
    <th> ... </th>
    ...
  </tr>
  <tr>
    <td> ... </td>
    <td> ... </td>
    ...
  </tr>
  ...
</table>
```

Hierbei umschließen die Tags `<tr> ... </tr>` jeweils eine Tabellenzeile (*table row*). Die erste Zeile, die Kopfzeile, besteht aus mehreren Tags `<th> ... </th>` (*table head*) gebildet und so automatisch fett und zentriert formatiert. Die nachfolgenden Tabellenzeilen bestehen aus den Tags `<td> ... </td>`, die jeweils eine einzelne Tabellenzelle (*table data*) darstellen.

Die Hintergrundfarbe einer Tabellenzeile wird in HTML durch die folgende Formatierung geändert:

```
<tr style="background-color:red">
  <td> ... </td>
</tr>
```

und entsprechend in einer einzelnen Zelle mit

```
<td style="background-color:#FF0000"> ... </td>
```

Hierbei ist `#FF0000` der RGB-Code für die Farbe. Da das erste (rechte) Byte den Blauanteil codiert, das zweite den Grünanteil und das dritte (linke) den Rotanteil, also

#12A20F,

bedeutet `#FF0000` dasselbe wie `red`. Weitere Informationen zu HTML finden Sie in dem Online-Tutorial <http://de.selfhtml.de>.

A.5.1 HTML-Ausgabe von Wahrheitstabellen

Die folgende kleine Applikation zeigt die Implementierung und die Wirkung der logischen Operatoren in Form einer in HTML formatierten Tabelle.

```

1 import javax.swing.*;
2
3 /** Zeigt Wahrheitstabellen und demonstriert logische Operatoren.*/
4 public class LogischeOperatoren {
5     public static void main( String[] args ) {
6         boolean a, b;
7         String output = "<html><table border=\"1\">";
8
9         output += "<tr><th> a </th><th> b </th><th> a&&b </th><th> a|b </th>";
10        output += "<th> a^b </th><th> !a </th><th> a&b </th><th> a|b </th></tr>";
11        a = false;
12        b = false;
13        output += "<tr><td>" + a + "</td><td>" + b + "</td><td>" + (a&&b) + "</td><td>" + (a | b);
14        output += "</td><td>" + (a^b) + "</td><td>" + !a + "</td>";
15        output += "<td>" + (a&b) + "</td><td>" + (a | b) + "</td></tr>";
16        a = false;
17        b = true;
18        output += "<tr><td>" + a + "</td><td>" + b + "</td><td>" + (a&&b) + "</td><td>" + (a | b);
19        output += "</td><td>" + (a^b) + "</td><td>" + !a + "</td>";
20        output += "<td>" + (a&b) + "</td><td>" + (a | b) + "</td></tr>";
21        a = true;
22        b = false;
23        output += "<tr><td>" + a + "</td><td>" + b + "</td><td>" + (a&&b) + "</td><td>" + (a | b);
24        output += "</td><td>" + (a^b) + "</td><td>" + !a + "</td>";
25        output += "<td>" + (a&b) + "</td><td>" + (a | b) + "</td></tr>";
26        a = true;
27        b = true;
28        output += "<tr><td>" + a + "</td><td>" + b + "</td><td>" + (a&&b) + "</td><td>" + (a | b);
29        output += "</td><td>" + (a^b) + "</td><td>" + !a + "</td>";
30        output += "<td>" + (a&b) + "</td><td>" + (a | b) + "</td></tr>";
31
32        output += "</table>";
33        output += "<br>";
34        output += "<p>25 & 12 = " + (25 & 12) + "</p>";
35        output += "<p>25 | 12 = " + (25 | 12) + "</p>";
36        output += "<p>25 ^ 12 = " + (25 ^ 12) + "</p>";
37        output += "<p>25L ^ 0x1F = " + (25L ^ 0x1F) + "</p>";
38        output += "<p>'A' ^ 'B' = " + ('A' & 'B') + "</p>";
39        output += "</html>";
40
41        // Tabelle ausgeben, ggf. mit Scrollbalken:
42        JLabel label = new JLabel(output);
43        JScrollPane scroller = new JScrollPane( label );
44        // Größe des Scrollbereichs einstellen:
45        scroller.setPreferredSize(new java.awt.Dimension(300,250));
46        // Textbereich zeigen:
47        JOptionPane.showMessageDialog( null, scroller,
48            "Wahrheitstabellen", JOptionPane.PLAIN_MESSAGE );
49    }
50 }

```

Die Ausgabe besteht aus den *Wahrheitstabellen* der Booleschen Algebra:

A.6 Call by reference und call by value

Eine Variable einer Klasse ist ein bestimmter Speicherplatz im RAM des Computers. Sie kann von einem primitiven Datentyp sein oder von einer bestimmten Klasse, abhängig von ihrer Deklaration:

```

1 int zahl;
2 String eingabe;

```

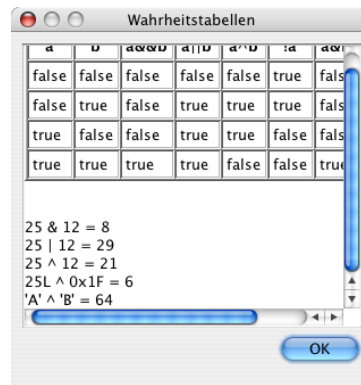


Abbildung A.2: Ausgabe der Applikation LogischeOperatoren.

In Java gibt es jedoch einen fundamentalen Unterschied zwischen Objekten und Daten primitiver Datentypen, also **boolean**, **int**, **double**, Anschaulich gesprochen sind primitive Datentypen die „Atome“, aus denen Objekte zusammengesetzt sind. Objekte bestehen aus Daten (also aus Objekten oder primitiven Daten), während eine Variable eines primitiven Datentyps nur genau einen Wert zu einem bestimmten Zeitpunkt annehmen kann. Außerdem haben Objekte im Unterschied zu primitiven Datentypen Methoden.

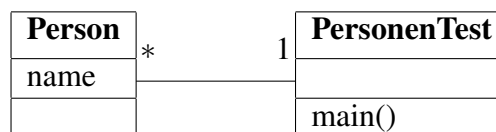
Der Arbeitsspeicher eines Computers ist in eine endliche Anzahl von Speicherzellen aufgeteilt, normalerweise von der Länge 1 Byte. Die (physikalische) Adresse einer Speicherzelle ist im Wesentlichen ihre Nummer. Technisch gesehen belegt eine Variable primitiven Datentyps einen bestimmten Ort im Speicher. Bei Verwendung im Programm wird ihr Wert kopiert und verarbeitet („*call by value*“). Eine Variable, die ein Objekt darstellt, ist dagegen eine „Referenz“ auf den Speicherort, an dem sich das Objekt selbst befindet, also eine Speicheradresse. Bei Verwendung im Programm wird diese Referenz übergeben („*call by reference*“). Variablen, die Objekte darstellen, werden *Referenzen*, *Pointer* oder *Zeiger* genannt. Ein Pointer oder eine Referenz hat also als Wert die Speicheradresse, die das referenzierte Objekt während der Laufzeit hat.

Was der Unterschied von *call by value* und *call by reference* bedeutet, wird am einfachsten klar, wenn man Variablen gleichsetzt. Definieren wir dazu zunächst die Klasse Person.

```

1 // Person.java
2 public class Person {
3     String name;
4     // Konstruktor: ---
5     Person ( String name ) {
6         this.name = name;
7     }
8 }

```



Die folgende main-Klasse verwendet Objekte der Klasse Person.

```

1 // PersonenTest.java
2 import javax.swing.*;
3
4 public class PersonenTest {
5     public static void main (String[] args) {
6         Person subjekt1, subjekt2;
7         int x, y;

```

```

8      String ausgabe;
9
10     // primitive Datentypen:
11     x = 1;
12     y = x;
13     y = 2;
14
15     // Objekte:
16     subjekt1 = new Person( "Tom" ); // erzeugt Objekt mit Namen "Tom"
17     subjekt2 = subjekt1;
18     subjekt2.name += " & Jerry";    // Objektattribut veraendert
19
20     // Ausgabe:
21     ausgabe = "x = " + x + "\ny = " + y;
22     ausgabe += "\n\nString von subjekt1: " + subjekt1.name;
23     ausgabe += "\nString von subjekt2: " + subjekt2.name;
24     JOptionPane.showMessageDialog( null, ausgabe );
25 }
26 }

```

Beachten Sie, dass das Attribut name von Objekt subjekt2 einfach verändert wird, indem die Syntax

objekt.attribut

verwendet wird. Das ist der einfachste Zugriff auf Attribute von Objekten. Dies widerspricht scheinbar der Kapselung der Daten durch Methoden, doch wir sehen weiter unten, dass der Zugriff auf das Attribut beschränkt werden kann. (Hier sei nur schon mal gesagt, dass das Attribut *friendly* ist und daher von Klassen im gleichen Paket gesehen und verändert werden darf.)

Übungsaufgabe. Stellen Sie eine Vermutung an, was die Ausgabe des obigen Programms ist. Überprüfen Sie Ihre Vermutung, indem Sie die beiden Klassen implementieren.

Ein Vergleich aus dem Alltag: Ein Objekt ist wie ein Fernseher, und seine Referenzvariable ist wie die Fernbedienung. Wenn Sie die Lautstärke verändern oder den Sender wechseln, so manipulieren Sie die Referenz, die ihrerseits das Objekt modifiziert. Wenn Sie durch das Zimmer gehen und trotzdem nicht die Kontrolle über das Fernsehen verlieren wollen, so nehmen Sie die Fernbedienung mit und nicht den Fernsehapparat.

In Java sind Referenzvariablen und ihre Objekte eng miteinander verbunden. Die Referenzvariable hat als Datentyp die Klasse des referenzierten Objekts.

Objekt ref;

Die Erzeugung eines Objekts geschieht mit dem **new**-Operator und der Zuordnung der Referenz:

ref = **new** Objekt();

Daher gibt es keinen Grund, die Adresse des referenzierten Objekts zu bekommen, ref hat sie schon intern. (In Java kann man nicht einmal die Adresse bekommen, es gibt keinen Adressoperator).

Merkregel 24. Vermeiden Sie unbedingt Zuweisungen von Objektvariablen außer der Erzeugung mit dem **new**-Operator, also so etwas wie `subjekt1 = subjekt2` in der obigen Klasse `Person`. Damit bewirken Sie lediglich, dass zwei Zeiger auf ein einziges Objekt zeigen. (Das bringt mindestens so viel Verwirrung wie zwei Fernbedienungen, die einen einzigen Fernseher einstellen...)

A.7 enums

Seit Version 5.0 unterstützt Java „Aufzählungstypen“, so genannte *enums* (*enumerated types*). Eine *Enum* oder ein *Aufzählungstyp* ist ein Datentyp bzw. eine Klasse, deren Objekte nur endlich viele vorgegebene Werte annehmen kann. Ein Beispiel ist die Enum Jahreszeit, die nur einen der vier Werte FRUEHLING, SOMMER, HERBST oder WINTER annehmen kann. Die möglichen Werte einer Enum heißen *Enumkonstanten* oder *Aufzählungskonstanten* und können einfache, nur durch ihre Namen gekennzeichnete Variablen, aber auch komplexe Objekte darstellen. Üblicherweise werden sie als Konstanten in Großbuchstaben geschrieben, und es gelten dieselben Bedingungen an ihre Namen wie für allgemeine Variablen in Java; insbesondere dürfen sie nicht mit Ziffern beginnen und nicht einem Schlüsselwort gleichen.

Eine Enum wird wie eine Klasse deklariert, jedoch nicht mit dem Schlüsselwort **class**, sondern mit **enum**. Beispiel:

```
enum Jahreszeit {FRUEHLING, SOMMER, HERBST, WINTER};
```

Diese kurze Anweisung erzeugt in Wirklichkeit eine vollständige Klasse, die neben den Object-Methoden eine statische Methode `values()` bereitstellt und sogar die Schnittstelle `Comparable` implementiert, wobei die Sortierreihenfolge durch die Reihenfolge der Aufzählung festgelegt ist. Eine `enum` kann wie jede Klasse durch selbsterstellte Methoden erweitert werden.

Beispiel A.1. Ein einfaches Beispiel für Aufzählungstypen ist ein Kartenspiel. Dazu deklarieren wir zunächst eine Klasse `Spielkarte`, die als Attribute die enums `Farbe` und `Karte` hat.

```
1 import java.util.*;
2
3 enum Farbe { KARO, HERZ, PIK, KREUZ }
4 enum Karte { SIEBEN, ACHT, NEUN, ZEHN, BUBE, DAME, KOENIG, ASS }
5
6 public class Spielkarte {
7     private final Farbe farbe;
8     private final Karte karte;
9
10    public Spielkarte(Farbe farbe, Karte karte) {
11        this.farbe = farbe;
12        this.karte = karte;
13    }
14
15    public Karte getKarte() { return karte; }
16    public Farbe getFarbe() { return farbe; }
17    public String toString() { return farbe + " " + karte; }
18 }
19
20 class BlattSortierung implements Comparator<Spielkarte> {
21     public int compare(Spielkarte k1, Spielkarte k2) {
22         if ( k1.getFarbe().compareTo(k2.getFarbe()) != 0 ) {
23             return -k1.getFarbe().compareTo(k2.getFarbe());
24         } else {
25             return -k1.getKarte().compareTo(k2.getKarte());
26         }
27     }
28 }
```

Die `toString`-Methode nutzt die `toString`-Methoden von `Farbe` und `Karte` aus. In der folgenden Applikation `Spiel` wird ein Spiel mit den Spielkarten erzeugt, und es kann gemäß Eingabe für n Spieler gegeben werden. (Ohne Eingabe wird ein Skatspiel mit drei Spielern und einem Stock gegeben.)

```
1 import java.util.*;
2
```



```

3 public class Spiel {
4     public final static Comparator sortierung = new BlattSortierung();
5     public ArrayList<Spielkarte> stapel;
6
7     public Spiel() {
8         stapel = new ArrayList<Spielkarte>();
9         for (Farbe farbe : Farbe.values()) {
10             for (Karte karte : Karte.values()) {
11                 stapel.add(new Spielkarte(farbe, karte));
12             }
13         }
14         Collections.shuffle(stapel);
15     }
16
17     public ArrayList<Spielkarte> getStapel() {
18         return stapel;
19     }
20
21     public ArrayList<Spielkarte> geben(int n) {
22         int deckSize = stapel.size();
23         List<Spielkarte> handView = stapel.subList(deckSize-n, deckSize);
24         ArrayList<Spielkarte> blatt = new ArrayList<Spielkarte>(handView);
25         handView.clear(); // loesche gegebene Spielkarten aus Stapel
26         Collections.sort(blatt, sortierung);
27         return blatt;
28     }
29
30     public static void main(String args[]) {
31         int blaetter = 3;
32         int kartenJeBlatt = 10;
33         if ( args.length > 0 ) blaetter = Integer.parseInt(args[0]);
34         if ( args.length > 1 ) kartenJeBlatt = Integer.parseInt(args[1]);
35
36         Spiel spiel = new Spiel();
37         for (int i=0; i < blaetter; i++) {
38             System.out.println(spiel.geben(kartenJeBlatt));
39         }
40
41         int rest = spiel.getStapel().size();
42         if ( rest > 0 ) {
43             System.out.println("Stock: " + spiel.geben(rest));
44         }
45     }
46 }

```

Die Ausgabe ergibt beispielsweise

```

> java Spiel 3 10
[KREUZ ASS, KREUZ NEUN, KREUZ SIEBEN, PIK ASS, PIK BUBE, PIK ZEHN, HERZ ACHT, KARO DAME, KARO ZEHN, KARO SIEBEN]
[KREUZ DAME, PIK ACHT, PIK SIEBEN, HERZ KOENIG, HERZ DAME, HERZ BUBE, HERZ ZEHN, HERZ NEUN, HERZ SIEBEN, KARO KOENIG]
[KREUZ KOENIG, KREUZ BUBE, KREUZ ACHT, PIK KOENIG, PIK DAME, PIK NEUN, HERZ ASS, KARO ASS, KARO BUBE, KARO ACHT]
Stock: [KREUZ ZEHN, KARO NEUN]

```

□

Beispiel A.2. Nehmen wir an, es sollen Daten und Methoden zu einem Aufzählungstypen hinzugefügt werden. Betrachten wir dazu die Planeten unseres Sonnensystems (seit dem 24. August 2006 ohne Pluto⁵). Jeder Planet hat seine Masse M und seinen Radius r , und anhand dieser Daten kann seine Schwerebeschleunigung $g = \frac{GM}{r^2}$ und das Gewicht $F = mg$ eines Objekts der Masse m auf ihm berechnet werden.

⁵ <http://www.iau2006.org/mirror/www.iau.org/iau0603/>

```

1 public enum Planet {
2     MERKUR (3.303e+23, 2.4397e6),
3     VENUS (4.869e+24, 6.0518e6),
4     ERDE (5.976e+24, 6.37814e6),
5     MARS (6.421e+23, 3.3972e6),
6     JUPITER (1.9e+27, 7.1492e7),
7     SATURN (5.688e+26, 6.0268e7),
8     URANUS (8.686e+25, 2.5559e7),
9     NEPTUN (1.024e+26, 2.4746e7);
10
11     /** Newtonsche Gravitationskonstante (m3 kg-1 s-2).*/
12     public static final double G = 6.67300E-11;
13     private final double masse; // in kilograms
14     private final double radius; // in meters
15
16     Planet(double masse, double radius) {
17         this.masse = masse;
18         this.radius = radius;
19     }
20
21     public double schwerebeschleunigung() {
22         return G * masse / (radius * radius);
23     }
24     public double gewicht(double andereMasse) {
25         return andereMasse * schwerebeschleunigung();
26     }
27
28     public static void main(String[] args) {
29         double erdGewicht = 70;
30         if ( args.length > 0 ) erdGewicht = Double.parseDouble(args[0]);
31         double masse = erdGewicht / ERDE.schwerebeschleunigung();
32         for (Planet p : Planet.values())
33             System.out.printf("Gewicht auf dem Planeten %s: %f%n", p, p.gewicht(masse));
34     }
35 }

```

Der Aufzählungstyp Planet enthält nach der Aufzählung die üblichen Klassendeklarationen, also Attribute, Konstruktor und Methoden, und jede Aufzählungskonstante wird mit den dem Konstruktor zu übergebenden Parametern deklariert. Der Konstruktor darf nicht public deklariert werden. In der main-Methode wird als Eingabe ein Erdgewicht (in einer beliebigen Gewichtseinheit) verlangt, für die anderen Planeten berechnet und ausgegeben.

```

$ java Planet 70
Gewicht auf dem Planeten MERKUR: 26,443033
Gewicht auf dem Planeten VENUS: 63,349937
Gewicht auf dem Planeten ERDE: 70,000000
Gewicht auf dem Planeten MARS: 26,511603
Gewicht auf dem Planeten JUPITER: 177,139027
Gewicht auf dem Planeten SATURN: 74,621088
Gewicht auf dem Planeten URANUS: 63,358904
Gewicht auf dem Planeten NEPTUN: 79,682965

```

□

Beispiel A.3. Man kann sogar jeder Aufzählungskonstanten eine eigene Methode geben (eine „konstantenspezifische Methode“). Dafür muss in dem Aufzählungstyp lediglich eine abstrakte Methode deklariert werden. Ein einfaches Beispiel ist die folgende Deklaration eines Operatoren, der die fünf Grundrechenarten definiert.

```

1 public enum Operation {
2     MAL { double eval(double x, double y) { return x * y; } },

```

```

3  DURCH { double eval(double x, double y) { return x / y; } },
4  MOD   { double eval(double x, double y) { return x % y; } },
5  PLUS  { double eval(double x, double y) { return x + y; } },
6  MINUS { double eval(double x, double y) { return x - y; } };
7
8  // Do arithmetic op represented by this constant
9  abstract double eval(double x, double y);
10
11 public static void main(String args[]) {
12     double x = 11;
13     double y = 6;
14     if (args.length >= 2) {
15         x = Double.parseDouble(args[0]);
16         y = Double.parseDouble(args[1]);
17     }
18     for (Operation op : Operation.values())
19         System.out.printf("%f %s %f = %f\n", x, op, y, op.eval(x, y));
20 }
21 }

```

Die Ausgabe lautet:

```

11,000000 MAL 6,000000 = 66,000000
11,000000 DURCH 6,000000 = 1,833333
11,000000 MOD 6,000000 = 5,000000
11,000000 PLUS 6,000000 = 17,000000
11,000000 MINUS 6,000000 = 5,000000

```

□

Im Paket `java.util` gibt es die beiden Set- und Map-Implementierungen `EnumSet` und `EnumMap` speziell für Aufzählungstypen.

A.7.1 Zusammenfassung

- Benötigt man in einer Anwendung Objekte einer Klasse aus einer endlichen Liste von möglichen Werten, z.B. um vorgegebene Optionen darzustellen, so kann man in Java Aufzählungstypen, oder Enums, verwenden. Sie werden wie eine Klasse, allerdings mit dem Schlüsselwort `enum`, deklariert.
- Die möglichen Optionen heißen Aufzählungskonstanten oder Enumkonstanten und sind Objekte des Aufzählungstyps. Für eine Enum können eigene Attribute und entsprechend eigene Konstruktoren erstellt werden, die allerdings nicht `public` sein dürfen. Ebenso können eigene Methoden für Enums deklariert werden.
- Aufzählungstypen implementieren `Comparable` und besitzen eine Sortierordnung, die automatisch durch die Reihenfolge der Aufzählungskonstanten gegeben ist.
- Beim Hinzufügen neuer Aufzählungskonstanten in einer Enum ist keine Neukompilierung etwaiger aufrufenden Klassen nötig, da sie Objekte sind und dynamisch eingebunden werden.
- Durch den vorgegebenen Wertebereich eines Aufzählungstyp können diese Klassen in Java sehr performant gespeichert werden.
- Wann sollte man Enums verwenden? Jedes Mal, wenn man eine vorgegebene Menge an Konstanten benötigt. Sie bieten sich beispielsweise an, wenn man Aufzählungen wie Planeten, Wochentage, Spielkarten usw. benötigt, aber auch andere Mengen von Werten, die man bereits zur Kompilierzeit vollständig kennt, wie Menüauswahlen oder Rundungsmoden.

Literaturverzeichnis

- [1] BÖTTCHER, Ulrike ; FRISCHALOWSKI, Dirk: *Java 5 Programmierhandbuch. Einstieg und professioneller Einsatz*. Frankfurt : Software & Support Verlag, 2005. – <http://www.j2sebuch.de>
- [2] CORMEN, Thomas H. ; LEISERSON, Charles E. ; RIVEST, Ronald L.: *Introduction to Algorithms*. New York : McGraw-Hill, 1990
- [3] DEITEL, Harvey M. ; DEITEL, Paul J.: *Java. How to program*. 3rd Edition. Upper Saddle River : Prentice Hall, 1999
- [4] GOLL, Joachim ; WEISS, Cornelia ; ROTHLÄNDER, Peter: *Java als erste Programmiersprache*. Stuttgart : B.G. Teubner, 2000
- [5] GUMM, Heinz P. ; SOMMER, Manfred: *Einführung in die Informatik*. München : Oldenbourg Verlag, 2008. – ISBN 978–3486587241
- [6] KNUTH, Donald E.: *The Art of Computer Programming. Volume 3: Sorting and Searching*. 3rd Edition. Reading : Addison-Wesley, 1998
- [7] KRÜGER, Guido: *Handbuch der Java-Programmierung*. 3. Aufl. München : Addison-Wesley, 2003. – www.javabuch.de
- [8] RATZ, Dietmar ; SCHEFFLER, Jens ; SEESE, Detlef: *Grundkurs Programmieren in Java. Band 1: Der Einstieg in Programmierung und Objektorientierung*. 2. Aufl. München Wien : Hanser Verlag, 2004
- [9] ULLENBOOM, Christian: *Java ist auch eine Insel*. Bonn : Galileo Press, 2002

Weiterführende Links

- 1. Java API Specification von SUN (auf Englisch) <http://java.sun.com/reference/api/>
- 2. Java Tutorial von SUN (auf Englisch) <http://java.sun.com/docs/books/tutorial/>
- 3. Friedrich Esser, *Java2 Online* <http://www.galileocomputing.de/openbook/java2/>
- 4. Guido Krüger, *Handbuch der Java-Programmierung Online* <http://www.javabuch.de/>
- 5. Christian Ullenbooms Online-Buch <http://www.java-tutor.com/javabuch/>

Index

abstract, 100
boolean, 22
byte, 22
char, 22
class, 11, 12
double, 22
do, 50
enum, 136
extends, 95
false, 22
final , 62
float, 22
for, 48, 63
implements, 98
import, 13
instanceof, 108
interface, 98
int, 22
long, 22, 44
new, 76
null, 13
package, 91
private, 92
protected, 92
public, 11, 12, 92
return, 54
short, 22
static, 73, 75
super, 96
this, 72, 96
true, 22
void, 54
while, 50
0b, 126
0x, 126
<?>, 117
? : , 45
?, 117
ArrayList, 110
Arrays, 108
Collections.sort(), 109
Collections, 110
Comparator, 108
Math.random(), 40
Object, 95
TreeModel, 116
TreeNode, 116
compareTo, 106
equals, 108
hashCode, 107
import, 16
^, 39, 43
|, 39, 43
||, 39
~, 43
&, 39, 43
&&, 39
<<, 43
>>>, 43
>>, 43
%=, 25
*=, 25
++, 25
+=, 25, 38
--, 25
-=, 25
/=, 25
==, 37
abstrakte Klasse, 100
ActionListener, 100
Adresse, 60
Algorithmus, 35, 78
Alphabet, 124
Alternative, 42
AND, 39
Anweisung, 14
API, 18
Applet, 9
Applikation, 9
Argument, 55, 58, 59
Array, 27, 59
ArrayList, 110
Arrays, 108
ASCII, 124
Attribut, 62, 69
Attribute, 69
- und lokale Variablen, 72
Aufruf, 55
Aufzählungstyp, 136
ausgeglichen, 114
Auswahlstruktur, 35
B*-Baum, 113
Bankschalter, 88
Basis, 124
Baum, 113
Bedingung, 36
Bedingungsoperator, 45
Bezeichner, 12
Binärbruch, 127
binärer Baum, 114
Bitmaske, 44
bitweiser Operator, 43

- Blatt, 113
- Block, 14
- Boolesche Algebra, 133
- Boolesche Algebra, 40
- Boolescher Ausdruck, 36
- Bytecode, 9

- call by value, 134
- Cast-Operator, 23
- ceiling bracket, 103
- CLASSPATH, 91
- clone, 95
- Code, 124
- Code-Point, 16
- Collection, 110, 116
- Collections, 122
- Collections.sort(), 109
- Comparable, 100, 106
- Comparator, 108
- compareTo, 106
- Compiler, 8
- Container, 111

- Daten, 69
- Datensatz, 103
- Datenstruktur, 59, 103, 111
- Datentyp, 21
 - komplexer -, 75
 - primitiv, 22
- Deklaration, 21, 75
 - einer Methode, 54
- Dekrementoperator, 25
- Dezimalsystem, 124
- Dialogbox, 13
- direkter Zugriff, 104
- Direktzugriff, 60
- Division, ganzzahlige, 127
- Dokumentationskommentar, 15
- double-Zahl, 24
- Double.parseDouble, 30

- einfügen (verkettete Liste), 112
- Eingabeblock, 27, 61
- Endlosschleife, 46
- enhanced for-loop, 63
- Entwurf, 78
- enum, 136
- EnumMap, 139
- EnumSet, 139
- equals, 37, 95, 108
- Erzeugung eines Objekts, 76
- Escape-Sequenz, 16
- Euler'sche Zahl, 19
- Euro-Zeichen, 47
- Exception, 30

- Feld, 59
- FIFO, 115, 118, 122
- floor bracket, 103
- floor-bracket, 128
- for-each-Schleife, 63
- for-Schleife, 48

- Format-String, 17
- Formatierung von Zahlen, 49
- Funktion, 51

- ganze Zahlen, 127
- ganzzahlige Division, 127
- Gaußklammer, 103, 128
- Generalisierung, 94, 95
- get-Methode, 70
- getClass, 95
- Gosling, James, 7
- Grammatik, 8
- GregorianCalendar, 85
- Gregorianischer Kalender, 84

- Höhe, 114
- Hüllklasse, 30
- Halde, 115
- Handy, 10
- hashCode, 95, 107
- Heap, 74, 76, 115
- Heap-Bedingung, 116
- HTML, 131

- IDE, 9
- Identifizier, 12
- if-else-if-Leiter, 42
- Implementierung, 18, 78
- Index, 59
- Initialisierung, 22
- Inkrementoperator, 25
- instanceof, 108
- Instanz, 69
- instanziieren, 76
- Interaktion, 26
- Interface, 98
- Interpreter, 8
- Iteration, 45
- Iterator, 113

- J2ME, 10
- jar, 16
- jar-Archiv, 130
- java, 9
- Java Runtime Environment, 9
- Java Virtual Machine, 9
- javac, 8
- javadoc, 128
- JOptionPane, 12, 13
- JRE, 9
- JScrollPane, 63
- JSP, 10
- JTextArea, 63
- TextField, 76
- JVM, 9, 74

- Kalender, 84
- Kapselung, 69, 92, 93
- kaskadierten Verzweigung, 42
- Kind, 113
- Klammerung, 14
- Klasse, 9, 11, 69

- Klassenattribut, 73
- Klassendeklaration, 12
- Klassenmethode, 73
- Klassenmethoden, 56
- Klassenvariable, 62
- Knoten, 113
- Kollektion, 111
- Kommazahl, binäre –, 127
- Kommentar, 15
- Konkatenation, 24, 28
- konkatenieren, 26
- Konsistenz - equals und compareTo, 108
- Konsole, 17
- Konstante, 62, 108
- Konstanten, mathematische, 19
- konstantenspezifische Methode, 138
- Konstruktor, 71, 76
- konvertieren, 29
- Kopf (einer verketteten Liste), 112
- Kreiszahl, 19
- KVM, 10

- Laufzeitfehler, 30
- length, 62
- lineare Datenstruktur, 113
- Liste, 112
- Locale, 84, 86
- logische Operatoren, 39
- lokale Variable, 59, 74

- main, 12, 75
- Manifest, 130
- Map, 120
- Math.ceil, 104
- Math.floor, 104
- Math.random(), 40
- mathematische Konstanten, 19
- mehrdimensionale Arrays, 63
- Method Area, 74
- Methode, 12, 52, 69
 - nichtstatische -, 73
 - statische -, 54
- Methoden, 69
- Methodendeklaration, 54
- Methodenname, 52
- Methodenrumpf, 12, 52
- Midlet, 10
- Minimumheap, 115
- modal, 28
- modulo, 127

- Nachfolger (Baum), 113
- nichtstatische Methode, 73
- null, 71

- obere Gaußklammer, 103
- Object, 95
- Objekt, 69
 - Erzeugung, 76
- Objektmethode, 73
- Objektorientierung, 69
- Operand, 21
- Operanden, 20
- Operator, 20, 21
 - logischer, 39
- OR, 39
- Ordnung, 106

- Paket, 16, 91
- Parameter, 51, 52, 58
- parseInt, 29
- partiell geordneter Baum, 115
- pi, 19
- Pointer, 60, 134
- POJO, 73
- polymorphe Methode, 122
- POSIX, 86
- Präzedenz, 23
- print(), 17
- printf(), 17
- println(), 17
- priority queue, 115
- Problembereich, 69
- Pseudocode, 46, 78
- public, 71

- Quelltext, 11

- random access, 60
- random(), 40
- Referenz, 72, 134
- Reihung, 59
- Rekursion, 45
- replace, 30
- reservierte Wörter, 12
- reserviertes Wort, 15
- RGB, 44, 132
- Routine, 35
- Runnable, 100

- Scanner, 83
- Schlüssel, 115
- Schlüsselwort, 15
- Schleife, 46
- Schnittstelle, 98
- Scroll-Balken, 63
- SDK, 9
- Selektion, 35
- Sequenzdiagramm, 89
- sequenziell, 11
- Servlet, 10
- set-Methode, 70
- setText, 63
- Short-Circuit-Evaluation, 43
- showMessageDialog, 13
- Signatur, 54, 102
- Software Engineering, 77
- sortieren, 109
- sortierter Baum, 115
- Source-Code, 11
- Stack, 74
- Standardkonstruktor, 71
- static, 73
- statisch, 73

- statische Methode, 54
- String, 16, 24
- Suche, 105
- Superklasse, 96
- Swing, 12, 13
- System.currentTimeMillis(), 84
- System.nanoTime(), 85
- System.out.print(), 17
- System.out.printf(), 17
- System.out.println(), 17
- Systemzeit, 84

- Tag, 131
- this, 76
- toString, 95
- toString(), 71
- TreeModel, 116
- TreeNode, 116
- Tupel, 104
- Typumwandlung, 23

- ueberladen, 71, 95
- ueberschreiben, 95
- UML, 70
- unär, 25
- Unicode, 16, 124
- untere Gaußklammer, 103, 128

- Variable, 21, 75, 133
 - lokale, 59, 74
- verbesserte for-Schleife, 63
- Vererbung, 94, 95
- Vergleichsoperatoren, 36
- verkettete Liste, 112
- Verzweigung, 35
- virtuell, 69
- Virtuelle Maschine, 9
- VM, 9
- vollständige Suche, 105
- Vorgänger (Baum), 113

- Wahrheitstabelle, 133
- Wahrheitstabellen, 39
- Wert, 21
- Wertigkeit, 23
- Wiederholung, 46
- Wildcard, 117
- Wrapper-Klasse, 30
- Wurzel, 113

- XOR, 39

- Zählschleife, 48
- Zahlen, 124
 - ganze, 127
- Zeiger, 60, 112, 134
- Zugriffsmodifikator, 92
- Zugriffsmodifikator, 92
- Zuordnung, 120
- Zuweisungsoperator, 21