

Grundlagen der Programmierung

Vorlesungsskript für das erste Semester

Wirtschaftsinformatik

Andreas de Vries und Volker Weiß

Version: 22. September 2019

Dieses Skript unterliegt der *Creative Commons License* 4.0
(<http://creativecommons.org/licenses/by/4.0/deed.de>)



Inhaltsverzeichnis

| | | |
|----------|--|------------|
| 1 | Grundlegende Sprachelemente von Java | 6 |
| 1.1 | Einführung | 6 |
| 1.2 | Das erste Programm: Ausgabe eines Textes | 9 |
| 1.3 | Elemente eines Java-Programms | 12 |
| 1.4 | Strings und Ausgaben | 16 |
| 1.5 | Variablen, Datentypen und Operatoren | 19 |
| 1.6 | Eingaben | 26 |
| 1.7 | Zusammenfassung | 32 |
| 2 | Strukturierte Programmierung | 37 |
| 2.1 | Logik und Verzweigung | 38 |
| 2.2 | Iterationen: Wiederholung durch Schleifen | 47 |
| 2.3 | Arrays | 53 |
| 2.4 | Zusammenfassung | 59 |
| 3 | Subroutinen: Funktionen und statische Methoden | 62 |
| 3.1 | Die Idee von Subroutinen | 62 |
| 3.2 | Funktionen in der Mathematik | 63 |
| 3.3 | Funktionen mit Lambda-Ausdrücken | 64 |
| 3.4 | Statische Methoden | 70 |
| 3.5 | Rekursion | 82 |
| 4 | Objektorientierte Programmierung | 94 |
| 4.1 | Modellierung und Programmierung von Objekten | 94 |
| 4.2 | Mehrere Objekte und ihre Zustände: Fallbeispiel Konten | 112 |
| 4.3 | Vererbung | 116 |
| 4.4 | Datenkapselung und Sichtbarkeit von Methoden | 122 |
| 4.5 | Exceptions | 124 |
| 4.6 | Zusammenfassung | 130 |
| A | Spezielle Themen | 132 |
| A.1 | ASCII und Unicode | 132 |
| A.2 | Binärsystem und Hexadezimalsystem | 133 |
| A.3 | javadoc, der Dokumentationsgenerator | 138 |
| A.4 | jar-Archive: Ausführbare Dateien und Bibliotheken | 140 |
| A.5 | Zeit- und Datumfunktionen in Java | 141 |
| A.6 | Formatierte Ausgabe mit HTML | 145 |
| A.7 | Call by reference und call by value | 147 |
| A.8 | enums | 150 |
| A.9 | Psychologie und Logik: Der Wason-Test | 154 |

| | |
|--------------------------------------|------------|
| <i>Grundlagen der Programmierung</i> | 3 |
| Literaturverzeichnis | 156 |
| Index | 157 |

Vorwort

Das vorliegende Skript umfasst den Stoff der Vorlesung *Grundlagen der Programmierung* des Bachelor-Studiengangs Wirtschaftsinformatik am Fachbereich Technische Betriebswirtschaft in Hagen. Es ist die inhaltliche Basis für die in den Praktika gestellten Aufgaben und die Prüfungen.

Zwar ist das Skript durchaus zum Selbststudium geeignet, auch für Interessierte anderer Fachrichtungen oder Schülerinnen und Schüler, denn es werden keine besonderen Kenntnisse der Programmierung vorausgesetzt. Hervorzuheben ist aber die Erkenntnis: Schwimmen lernt man nicht allein durch Anschauen und Auswendiglernen der Bewegungsabläufe, sondern man muss selber ins Wasser! Ähnlich lernt man nämlich 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 und mit Hilfestellungen gelöst werden.

Das Konzept. Dem Konzept des vorliegenden Skripts liegt die Auffassung zu Grunde, dass die Fähigkeit zur Analyse und Innovation von Prozessen eine der wichtigsten Potenziale unserer Gesellschaft ist. Der Begriff des *Prozesses* spielt sowohl in der Betriebswirtschaft als auch in den Ingenieurwissenschaften eine wichtige Rolle, jede Wertschöpfung oder jede Leistungserstellung wird durch sie erst ermöglicht. Auch in der Informatik sind Prozesse von jeher wesentlicher Bestandteil, wenn auch unter den Namen „Algorithmen“, „Prozeduren“ oder „Routinen“. Die Programmierung nimmt dabei eine zentrale Schlüsselfunktion ein, denn sie zwingt dazu, die durch die jeweilige Programmiersprache vorgegebenen Anweisungsmöglichkeiten so präzise zu formulieren, dass ein Computer sie ausführen kann.

Wenn Programmierung, welche Programmiersprache soll man als Einstieg wählen? 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 oder nur abgeschwächt vorkommen. Insbesondere ist 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 Sprachenfamilie der „C-Syntax“ — also Sprachen wie C, C++, Java, C#, aber auch Interpretersprachen wie JavaScript und PHP — am meisten verwendet wird, 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 zu einem großen Arsenal standardmäßig bereitgestellter Pro-

gramme und Softwarekomponenten geworden, die die *Java API* (*API = Application Programming Interface*) bilden. Einen vollständigen und detaillierten Überblick kann man natürlich nur allmählich und nach 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)
- *Strukturierte Programmierung.* (Verzweigungen, logische Operatoren, Schleifen, Erstellen einfacher Algorithmen)
- *Programmierung von Subroutinen.* (Funktionen, statische Methoden und Rekursionen)
- *Objektorientierte Programmierung.* (Objekte, Objektattribute und -methoden, Klassendiagramme nach UML, Assoziationen und Vererbungen)

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 [2], das Handbuch *Java-Programmierung* von Krüger und Hansen [5], der zweibändige *Grundkurs Programmieren in Java* von Dietmar Ratz *et al.* [6] und schließlich der umfassende Überblick *Java ist auch eine Insel* von Christian Ullenboom [8].

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

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

Daneben existiert ein Wiki unter [wikibooks.org](http://de.wikibooks.org/wiki/Java_Standard):

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

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

Hagen,
im September 2019

Andreas de Vries, Volker Weiß

1

Grundlegende Sprachelemente von Java

Kapitelübersicht

| | | |
|-----|--|----|
| 1.1 | Einführung | 6 |
| 1.2 | Das erste Programm: Ausgabe eines Textes | 9 |
| 1.3 | Elemente eines Java-Programms | 12 |
| 1.4 | Strings und Ausgaben | 16 |
| 1.5 | Variablen, Datentypen und Operatoren | 19 |
| 1.6 | Eingaben | 26 |
| 1.7 | Zusammenfassung | 32 |

1.1 Einführung

Java ist eine objektorientierte Programmiersprache. Viele ihrer Fähigkeiten und einen großen Teil der Syntax hat sie von C und C++ geerbt. Im Gegensatz zu diesen Sprachen ist Java jedoch nicht primär darauf konzipiert, möglichst kompakte und optimal auf die zugrunde liegende Hardware angepasste Programme zu erzeugen, sondern soll die 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 vereint also bewährte Konzepte früherer Sprachen. Siehe dazu auch den Stammbaum in Abb. 1.1.

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*, da eine Eiche vor seinem Bürofenster stand. Als später entdeckt wurde, dass es bereits eine Programmiersprache mit diesem Namen 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. So war es purer Zufall, dass in dieser Phase 1993 das World Wide Web (WWW) explosionsartig populär wurde. Bei Sun wurde sofort das Potential von Java erkannt, interaktive („dynamische“) Web-Seiten zu erstellen. So wurde dem Projekt schlagartig neues Leben eingehaucht.

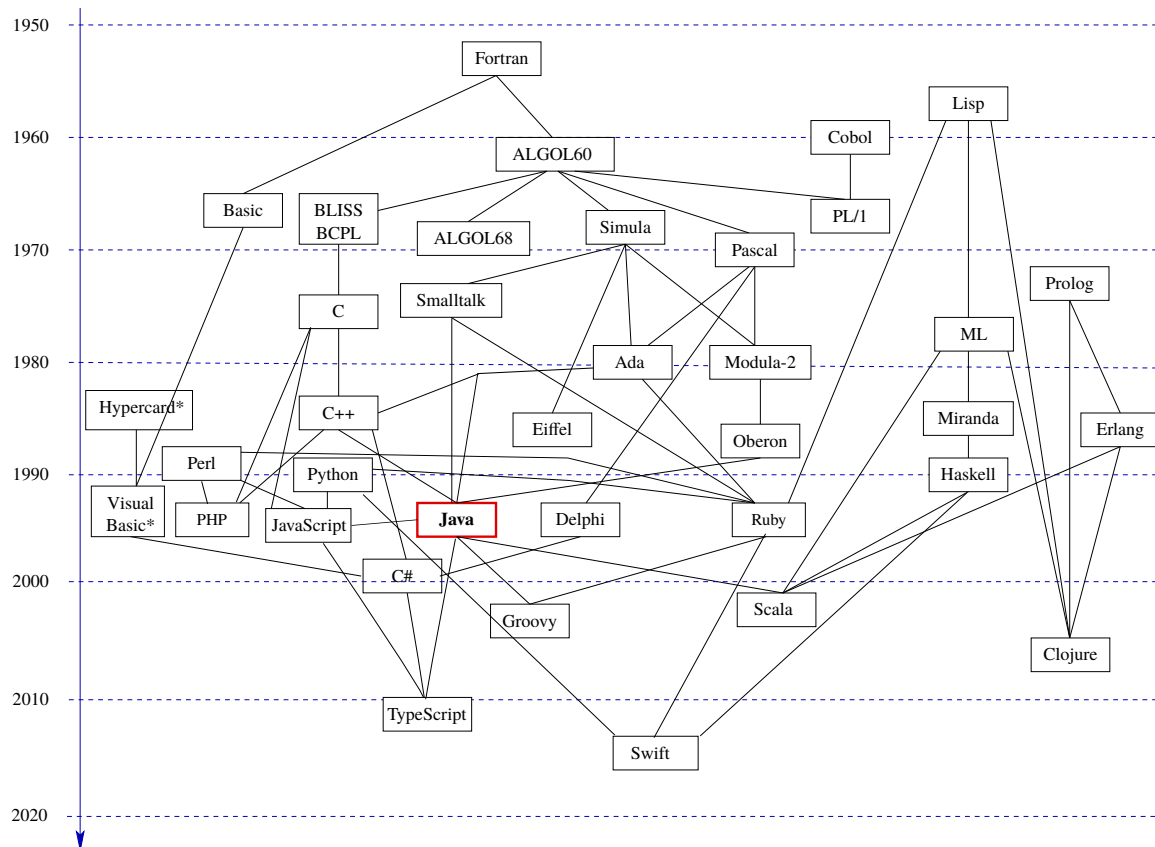


Abbildung 1.1. Stammbaum der gängigen Programmiersprachen (*Visual Basic ist eine nur auf Windows-Systemen lauffähige Programmiersprache, Hypercard eine Makrosprache von Apple MacIntosh).

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 konnte. Formal eingeführt wurde Java auf einer Konferenz von Sun im Mai 1995. Eine beispiellose Erfolgsgeschichte begann. Innerhalb weniger Jahre wurde Java zu einer der weitestverbreiteten Programmiersprachen.

1.1.1 Wie funktioniert Java?

Es gibt im wesentlichen zwei Klassen von Programmiersprachen, Compilersprachen und Interpretersprachen. Bei ersterer wird ein „Übersetzungsprogramm“, der *Compiler*, verwendet, um

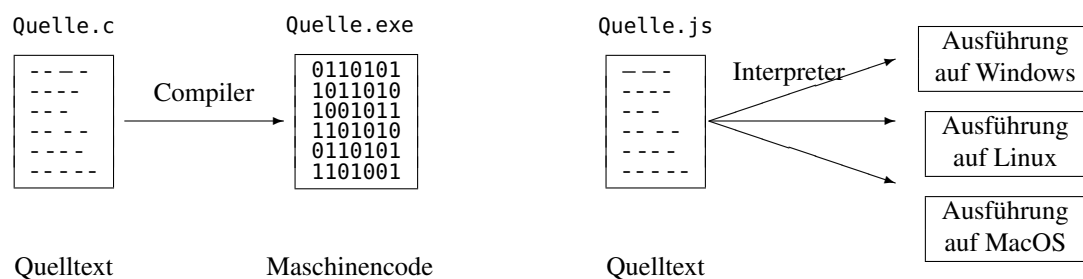


Abbildung 1.2. Wirkungsweisen eines Compilers und eines Interpreters, hier skizziert für C und JavaScript.

aus dem Quelltext eine Datei zu erstellen, die von dem jeweiligen Betriebssystem ausgeführt werden kann; unter Windows z.B. haben solche ausführbaren Dateien meist die Endung `.exe` (für *executable*). Bei einer Interpretersprache dagegen wird der Quelltext von einem *Interpreter* sofort in den Arbeitsspeicher (RAM, *random access memory*) des Computers übersetzt und

ausgeführt. Beispiele für Compilersprachen sind C oder C++, Beispiele für Interpretersprachen sind 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, denn ein Interpreter geht das Programm zeilenweise durch und muss jede Anweisung erst übersetzen, bevor er sie ausführt.

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, es können also höchstens noch Laufzeitfehler (z.B. nicht verarbeitbare Eingabedaten, Division durch 0) oder logische Fehler enthalten sein. Bei einer Interpretersprache dagegen läuft bei einem Programmierfehler das Programm einfach nicht, und die Ursache ist nicht näher eingegrenzt. 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, MacOS), auf der das Programm laufen soll, muss eine eigene Compilerdatei erstellt werden; bei Interpretersprachen kann der Quelltext 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 Quelltext eines Java-Programms steht in einer Textdatei mit der

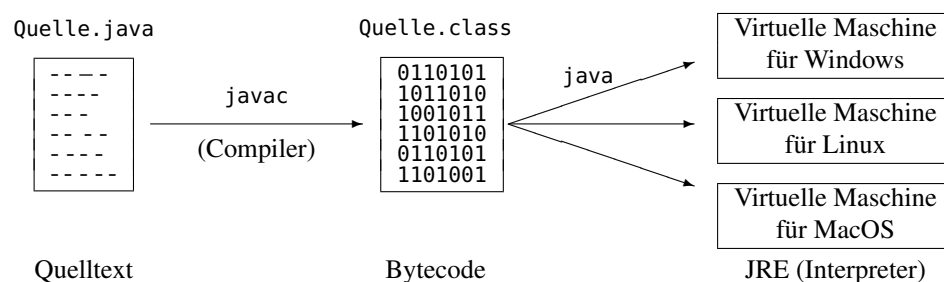


Abbildung 1.3. Vom Quellcode zur Ausführung eines Java-Programms (JRE = Java Runtime Environment)

Endung `.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 (JVM)*. 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. 1.3 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 JDK-Umgebung für das Betriebssystem, unter dem Sie arbeiten möchten. (JDK = *Java Development Kit*) Es beinhaltet die notwendigen Programme, insbesondere den Compiler `javac`, den Interpreter `java` und die Virtuelle Maschine. Das JDK ist als freie Software erhältlich unter

Nach erfolgter Installation¹ 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> oder <http://www.eclipse.org>

benutzt. Die IDE's sind geeignet für die fortgeschrittene Programmierung mit grafischen Bedienungselementen oder für größere Software-Projekte. Wir werden zunächst die einfache Programmerstellung mit einem Texteditor verwenden, um die Prinzipien der Kompilier- und Programmstartprozesse zu verdeutlichen. Später kann man mit etwas komfortableren Editoren wie dem *Java-Editor* (<http://www.javaeditor.org/>) für Windows oder dem plattformunabhängigen *jEdit* (<http://www.jedit.org>) arbeiten.

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 1.4 auf dem Bildschirm aus.



Abbildung 1.4. Ergebnis der Applikation Willkommen.

¹ Installationshinweise siehe <http://haegar.fh-swf.de/Java/HowTos/Installation-OpenJDK.pdf>

1.2.1 Wie kommt die Applikation ans Laufen?

Wie in der Einleitung beschrieben (Abb. 1.3 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
```

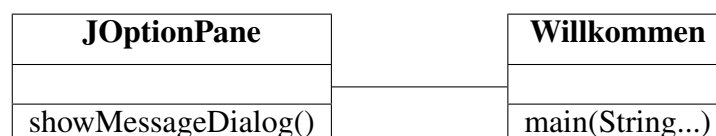
ausgeführt werden.

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, sogenannte *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. 8), die einzelnen Anweisungen auszuführen. Er arbeitet sie „sequenziell“, d.h. der Reihe nach, ab (Zeile 6).
4. Die Applikation besteht aus einer einzigen 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 ein „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 13 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

`Raketel`, 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: Ein Klassenname beginnt mit einem Großbuchstaben, enthält keine Sonderzeichen oder Umlaute und besteht aus einem zusammenhängenden Wort. Verwenden Sie einen „sprechenden“ Namen, der ausdrückt, was die Klasse bedeutet.

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 auf jeden Fall die folgende Konstruktion beinhalten:

```

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

```

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

Man kann übrigens auch statt „`String... args`“ auch drei Punkte für die eckige Klammer verwenden und „`String[] args`“ schreiben, was für deutsche Tastaturen etwas weniger angenehm ist.

```

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

```

In diesem Skript, in der Vorlesung und in den Praktika werden beide Versionen verwendet. Welche Version Sie selbst benutzen, bleibt Ihnen überlassen.

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. Vergleichen wir Deklarationen und Anweisungen, so stellen wir fest, dass Deklarationen die Struktur des Programms festlegen, während die Anweisungen den zeitlichen Ablauf definieren:

| Ausdruck | Wirkung | Aspekt |
|-------------|----------------------|---------------------|
| Deklaration | 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.

Merkregel 2. 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*, die Bestandteile der Programmiersprache sind und eine bestimmte vorgegebene Bedeutung haben. Die in Java reservierten Wörter sind in Tabelle 1.1 zusammengefasst.

Reservierte Wörter

| | | | | | | |
|------------|-----------|------------|-----------|--------------|----------|--------|
| 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 | var | void | volatile | while |

Literale

| | | | | | | |
|-------|------|------|--------------|---------------|---------------|-----------|
| false | true | null | 0, 1, -2, 5L | 1.2, .0, 2.0f | 'A', 'b', ' ' | "Abc", "" |
|-------|------|------|--------------|---------------|---------------|-----------|

In Java nicht verwendete reservierte Wörter

| | | | | | | |
|----------|-------|-------|--------|---------|------|-------|
| byvalue | cast | const | future | generic | goto | inner |
| operator | outer | rest | | | | |

Tabelle 1.1. Die reservierten Wörter in Java

Dazu gehören auch sogenannte *Literale*, z.B. die Konstanten **true**, **false** und **null**, deren Bedeutung wir noch kennen lernen werden. Allgemein versteht man unter einem Literal einen konstanten Wert, beispielsweise also auch 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.

Allgemein bezeichnet man die Regeln der Grammatik einer Programmiersprache, nach denen Wörter aneinander gereiht werden dürfen, mit dem Begriff *Syntax*. In der Syntax einer Programmiersprache spielen die reservierten Wörter natürlich eine Schlüsselrolle, weshalb man sie auch oft „Schlüsselwörter“ nennt.

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.:

```

1  /* Dies ist ein Kommentar, der
2     sich über mehrere Zeilen
3     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.

1.3.4 Programmierstil: Strukturierung durch Einrücken

Ein Leerzeichen trennt verschiedene Wörter, jedes weitere Leerzeichen direkt dahinter hat jedoch keine Wirkung. Ebenso werden in Java werden Leerzeilen und Tab-Stops vom Compiler nicht verarbeitet. Diese Zeichen heißen *Whitespace* oder *Leerraum*.

Wozu dann extra Leerraum? Das folgende Programm lässt sich compilieren und ausführen:

```

1 public class Willkommen{public static void main(String... args) {javax.
2 swing.JOptionPane.showMessageDialog(null, "Hallo Welt!");}}
```

Der Compiler kann es lesen, aber können Sie das auch? Fügen wir geeignet Leerraum ein, so ergibt sich das folgende Programm:

```

1 public class Willkommen{
2     public static void main(String... args) {
3         javax.swing.JOptionPane.showMessageDialog(null, "Hallo Welt!");
4     }
5 }
```

Das ist auf jeden Fall leichter lesbar. Zusätzlicher Leerraum kann (und soll!) also zur Strukturierung und Lesbarkeit des Programms verwendet werden.

1.3.5 import-Anweisung

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

bzw. \javax\swing (bei Windows-Systemen) gespeichert.² 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.

1.3.6 * Dialogfenster mit anderen Icons

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 macOS.

² Sie 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.

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.).

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

| Escape-Zeichen | Bedeutung | Beschreibung |
|---------------------|-------------------|--|
| <code>\uxxxx</code> | Unicode | das Unicode-Zeichen mit Hexadezimal-Code <code>xxxx</code> („Codepoint“); z.B. <code>\u222B</code> ergibt \int |
| <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

aufgelistet. Man kann also auch Anführungszeichen ausgeben:

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

ergibt als Ausgabe **"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| = \u2113");
```

die Ausgabe

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

Eine Auflistung der möglichen Unicode-Zeichen und ihrer jeweiligen Hexadezimalcodes finden Sie im Web zum Beispiel unter <http://www.isthisthingon.org/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` und `println` des Standard-Ausgabestroms `System.out`. Die Methode `print` gibt den eingegebenen String unverändert (und linksbündig) im Konsolenfenster aus. `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);
```

Die `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). Das folgende Programm fasst die Ausgabemöglichkeiten noch einmal zusammen:

```
1 import javax.swing.*;  
2  
3 public class Ausgaben {  
4     public static void main(String[] args) {  
5         // Ausgabe im Konsolenfenster:  
6         System.out.print("Dies ist eine ");  
7         System.out.println("Ausgabe mit " + (4+3) + " Wörtern.");  
8  
9         // Ausgabe in eigenem Fenster:  
10        JOptionPane.showMessageDialog(  
11            null, "Die ist eine Ausgabe \nmit " + 7 + " Wörtern."  
12        );  
13    }  
14 }
```

1.4.3 Dokumentation der API

Eine *API* (*application programming Interface*: „Schnittstelle für die Programmierung von Anwendungsprogrammen“) ist allgemein eine Bibliothek von Programmen und Routinen („Methoden“), mit denen über selbsterstellte Programme auf Funktionen des Betriebssystems zugegriffen werden kann. Jede Programmiersprache muss eine API für das jeweilige Betriebssystem bereitstellen, auf dem die erstellten Programme laufen sollen. Die API ist also eine Zwischenschicht zwischen dem Betriebssystem und dem Programmierer.

Die API von Java stellt dem Programmierer im Wesentlichen Programme (meist Klassen) 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, `JOptionPane` zur Programmierung von fensterbasierten Dialogen, oder `System` für systemnahe Funktionen.

Die Java-API besitzt eine detaillierte Dokumentation, die online verfügbar ist:

<https://docs.oracle.com/en/java/javase/13/docs/api/>

In Abbildung 1.5 ist ein Ausschnitt der Dokumentation für die Klasse `Math` gezeigt. Generell kann man eine Klasse über das Suchfeld eingeben. 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

OVERVIEW MODULE PACKAGE **CLASS** USE TREE DEPRECATED INDEX HELP Java SE 13 & JDK 13

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD SEARCH: X

arithmetic operations consistently produce correct results, which in some cases means the operations will not overflow the range of values of the computation. The best practice is to choose the primitive type and algorithm to avoid overflow. In cases where the size is int or long and overflow errors need to be detected, the methods `addExact`, `subtractExact`, `multiplyExact`, and `toIntExact` throw an `ArithmeticException` when the results overflow. For other arithmetic operations such as `divide`, `absolute value`, `increment by one`, `decrement by one`, and `negation`, overflow occurs only with a specific minimum or maximum value and should be checked against the minimum or maximum as appropriate.

Since:
1.0

Field Summary

| Modifier and Type | Field | Description |
|-------------------|-----------------|---|
| static double | <code>E</code> | The double value that is closer than any other to <i>e</i> , the base of the natural logarithms. |
| static double | <code>PI</code> | The double value that is closer than any other to <i>pi</i> , the ratio of the circumference of a circle to its diameter. |

Method Summary

| Modifier and Type | Method | Description |
|-------------------|----------------------------|---|
| static double | <code>abs(double a)</code> | Returns the absolute value of a double value. |
| static float | <code>abs(float a)</code> | Returns the absolute value of a float value. |
| static int | <code>abs(int a)</code> | Returns the absolute value of an int value. |

Abbildung 1.5. Ausschnitt aus der API-Dokumentation für die Klasse `Math`.

`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 diesem Skript werden generell nur die wichtigsten Elemente einer Klasse oder eines Interfaces beschrieben. Sie sollten daher beim Programmieren stets auch in der API-Dokumentation 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 für die Programmierung in Java bereit. Sie liefert die Konstanten

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

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)      // ex (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. 1.6: Je nach Datentyp der Operanden ergibt der (scheinbar!)

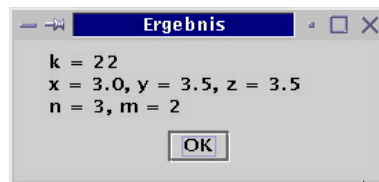


Abbildung 1.6. Die Applikation Arithmetik.

gleiche 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. 1.7. Dort kann dann ein *Wert* gespeichert werden.

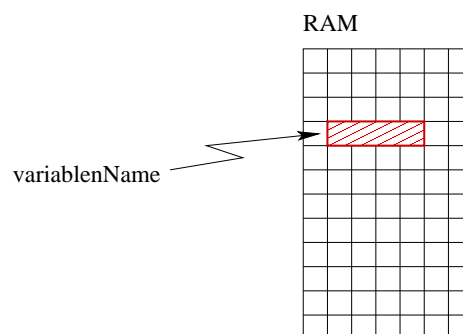


Abbildung 1.7. Die Deklaration der Variable *variablenName* bewirkt, dass im Arbeitsspeicher (RAM) eine bestimmte Anzahl von Speicherzellen reserviert wird. Die Größe des reservierten Speichers (= Anzahl der Speicherzellen) ist durch den Datentyp bestimmt. Vgl. Tabelle 1.3 (S. 22)

Es ist nicht ein 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, Variablenamen (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 3. 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.

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 zer-

| 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

stört 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 4. 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

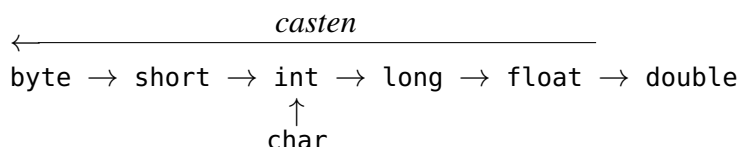
```
((Datentyp)) Ausdruck
```

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` → `double`) oder `int z = 'z';` (`char` → `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.

| 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.

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)`.

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,

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

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 gelesen natürlich völlig unsinnige Ausdruck bei Anmerkung (6),

$$\text{ausgabe} = \text{ausgabe} + \text{"\nx = "} + x \dots ;$$

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 Kombinierte Operatoren

Es gibt eine ganze Reihe von Operatoren, die häufig verwendete Anweisungen verkürzt darstellen. Sie heißen *kombinierte Operatoren* und können Zuweisungsoperatoren oder Operatoren mit einem Operanden (*unäre Operatoren*) sein. Wir geben Sie hier tabellarisch an.

| Zuweisungs- operator | Beispiel | Bedeutung | Wert für c, wenn int c=11, x=4 |
|-------------------------|----------|------------|--|
| += | c += x; | c = c + x; | 15 |
| -= | c -= x; | c = c - x; | 7 |
| *= | c *= x; | c = c * x; | 44 |
| /= | c /= x; | c = c / x; | 2 |
| %= | c %= x; | c = c % x; | 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 |
|----------|---------------|----------|--|
| ++ | präinkrement | ++c | erhöht erst c um 1, und führt dann den Rest der Anweisung mit dem neuen Wert aus |
| ++ | postinkrement | c++ | führt erst die gesamte Anweisung aus und erhöht <i>danach</i> den Wert von c um 1 |
| -- | prädekrement | --c | erniedrigt erst c um 1, und führt dann den Rest der Anweisung mit dem neuen Wert aus |
| -- | postdekrement | c-- | 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, notwendige Daten einzugeben. Wir werden in diesem Abschnitt mehrere Möglichkeiten kennenlernen, Eingaben in einer Applikation zu verarbeiten.

1.6.1 Eingaben über die Konsole mit `System.console().readLine()`

Eine der einfachsten Möglichkeiten, in Java Eingaben einzulesen, ist der Befehl

```
System.console().readLine();
```

Er bewirkt, wenn das Programm von der Konsole gestartet wird, dass der Programmablauf der Applikation stoppt und auf eine Eingabe des Anwenders wartet, bis die Return-Taste gedrückt wird. Die Tastatureingaben werden als String eingelesen, also dem allgemeinsten Datentyp, der Tastatureingaben umfasst, und können in eine Variable gespeichert und dann weiterverarbeitet werden. Ein einfaches Beispiel ist das folgende Programm:

```
1 public class ErsteEingaben {
2     public static void main(String[] args) {
3         System.out.print("Gib einen Text ein: ");
4         String text = System.console().readLine();
5         System.out.println("Du hast eingegeben: \"" + text + "\"");
6     }
7 }
```

Hier wird die Benutzereingabe in Zeile 4 in der Variablen `text` gespeichert und in der nächsten Anweisung einfach ausgegeben. (Beachten Sie in Zeile 5 dabei die Umklammerung des Eingabetextes durch Anführungszeichen mittels der Escape-Sequenz `\"`!)

1.6.2 Eingabefenster mit einem Textfeld

Eingaben können etwas bedienungsfreundlicher als über die Konsole in Java auch über Dialogfenster programmiert werden. Eine einfache Möglichkeit dazu bietet die Methode `showInputDialog` der Klasse `JOptionPane` aus dem Paket `javax.swing`, die mit der Anweisung

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

aufgerufen wird (falls `eingabe` vorher als String deklariert wurde). Die Wirkung dieser Anweisung ist hierbei, dass sie den weiteren Ablauf des Programms solange anhält, bis der Anwender mit der Maus auf `OK` oder `Abbrechen` geklickt oder die return-Taste gedrückt hat. Ein einfaches Beispiel für diese Art der Eingabe ist das folgende Programm:

```
1 import javax.swing.JOptionPane;
2
3 public class EingabenMitFenster {
4     public static void main(String[] args) {
5         String eingabe = JOptionPane.showInputDialog("Gib einen Text ein:");
6         System.out.println("Du hast eingegeben: \"" + eingabe + "\"");
7     }
}
```

```
8 }
```

Hier ist zu beachten, dass die Klasse `JOptionPane` (wie immer) zu importieren ist (Zeile 1) und dass die Variable im Unterschied zu dem obigen Programm nun `eingabe` heißt, nicht `text`. Drückt man bei der Eingabe auf `Abbrechen`, so wird die Eingabe gar nicht eingelesen und die Ausgabe unseres Programms lautet:

```
Du hast eingegeben: "null"
```

Probieren Sie es einmal aus!

1.6.3 Eingabedialoge mit mehreren Eingabefeldern

Wie kann man mehrere Dateneingaben in einem Dialogfenster programmieren? Eingaben über Dialogfenster mit mehreren Eingabefeldern zu erstellen, 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`, ein so genanntes *Array*, also eine durchnummerierte Liste vom Datentyp `JTextField[]`. Dieser Datentyp ist 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]};
```

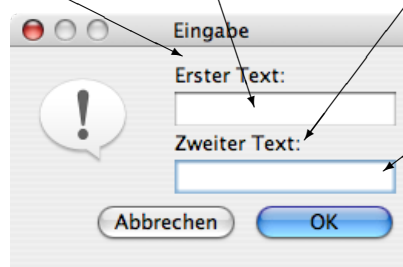


Abbildung 1.8. Das durch den „Eingabeblock“ erzeugte Dialogfenster

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

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

Die Wirkung dieser Anweisung ist hier wieder, dass sie den weiteren Ablauf des Programms solange anhält, bis der Anwender mit der Maus auf `OK` oder `Abbrechen` 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 `OK` oder `Abbrechen` gedrückt hat, wird in der Variablen `click` gespeichert, bei `OK` hat sie den Wert 0, bei `Abbrechen` den Wert 2:

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

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. Ein einfaches Beispiel für diese Art der Eingabe ist das folgende Programm:

```

1 import javax.swing.JOptionPane;
2 import javax.swing.JTextField;
3
4 public class MehrereEingaben {
5     public static void main(String[] args) {
6         // Eingabefelder aufbauen:
7         JTextField[] feld = {new JTextField(), new JTextField()};
8         Object[] msg = {"Erster Text:", feld[0], "Zweiter Text:", feld[1]};
9         // Dialogfenster anzeigen:
10        int click = JOptionPane.showConfirmDialog(null, msg, "Eingabe", 2);
11        // Eingaben ausgeben:
12        System.out.println(
13            "Du hast eingegeben: \"" + feld[0].getText() + ", " + feld[1].getText() + "\"";
14        );
15    }
16 }
```

1.6.4 Zwei eingegebene 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);

        // Konkatination 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 1.9 auf dem Bildschirm aus.

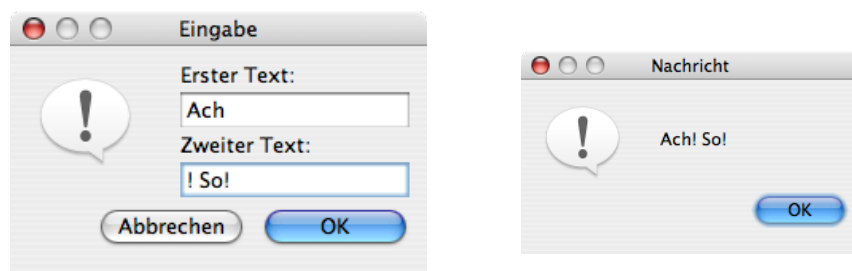


Abbildung 1.9. Die Applikation StringAddition. Links der Eingabedialog, rechts die Ausgabe.

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`.

Konkatination 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 nennt sie auch *Konkatination*. So ergibt die Konkatination der Strings **"Ach"** und **"! So!"** z.B.

"Ach" + "! So!" \mapsto **"Ach! So!"**.

1.6.5 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
        );
    }
}
```

1.6.6 Konvertierung von Strings in Zahlen

In großen Teilen ist dieses Programm identisch mit unserem Programm zur Stringaddition (Abb. 1.9). 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 Stringwert muss in den Integerwert 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 <input type="text" value="13"/> und <input type="text" value="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

1.6.7 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, wie in Tabelle 1.6

| Datentyp | boolean | char | byte | short | int | long | float | double |
|------------|----------------|-------------|-------------|--------------|------------|-------------|--------------|---------------|
| Hüllklasse | Boolean | Character | Byte | Short | Integer | Long | Float | Double |

Tabelle 1.6. Die primitiven Datentypen und ihre Hüllklassen (Wrapper Classes)

aufgelistet.

1.6.8 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');
```

Beispielsweise 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(',', '.', '.') );
```

1.7 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 mehrzeiliger 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.
- Das Casting kann von einem „kleineren“ in einen „größeren“ Datentyp implizit geschehen, z.B. bei der Anweisung

```
double x = 1;
```

(das Literal 1 ist vom Datentyp **int**!), oder muss explizit durch den *cast*-Operator von einem „größeren“ in einen „kleineren“ Datentyp programmiert werden:

```
int n = (int) Math.PI;
```

(Math.PI $\approx \pi$ ist **double**!)

- 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 über die Konsole kann mit


```
String ein = System.console().readLine();
```

 programmiert werden.
- Ebenso kann eine Eingabe mit


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

 mit der Klasse `JOptionPane` aus dem Paket `javax.swing` durchgeführt werden.
- Eine Eingabe von mehreren 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 müssen die Klassen `JOptionPane` und `JTextField` aus dem Paket `javax.swing` vor der Klassendeklaration importiert werden.

- Ein eingegebener Wert ist zunächst immer ein `String`. Um ihn als Zahl zu speichern, muss er konvertiert werden. Die Konvertierung eines Strings in einen **int** bzw. **double**-Wert geschieht mit den `parse`-Methoden aus den jeweiligen Hüllklassen, also `Integer.parseInt("...")` bzw. `Double.parseDouble("...")`.
- Die Konvertierung zwischen den primitiven Datentypen ist demgegenüber das Casting, siehe oben.
- 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 Operanden 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.

2

Strukturierte Programmierung

Kapitelübersicht

| | | |
|-------|---|----|
| 2.1 | Logik und Verzweigung | 38 |
| 2.1.1 | Die Verzweigung | 38 |
| 2.1.2 | Logische Bedingungen durch Vergleiche | 39 |
| 2.1.3 | Logische Operatoren | 40 |
| 2.1.4 | Die if-else-if-Leiter | 43 |
| 2.1.5 | Gleichheit von double-Werten | 43 |
| 2.1.6 | Abfragen auf Zahlbereiche | 43 |
| 2.1.7 | Bitweise Operatoren | 45 |
| 2.1.8 | * Bedingte Wertzuweisung mit dem Bedingungsoperator | 47 |
| 2.2 | Iterationen: Wiederholung durch Schleifen | 47 |
| 2.2.1 | Die while -Schleife | 47 |
| 2.2.2 | Die Applikation <i>Guthaben</i> | 48 |
| 2.2.3 | Umrechnung Dezimal- in Binärdarstellung | 49 |
| 2.2.4 | Die for -Schleife | 50 |
| 2.2.5 | Die do/while -Schleife | 52 |
| 2.2.6 | Wann welche Schleife verwenden? | 53 |
| 2.3 | Arrays | 53 |
| 2.3.1 | Was ist ein Array? | 54 |
| 2.3.2 | Lagerbestände | 55 |
| 2.3.3 | Die for-each-Schleife | 57 |
| 2.3.4 | Mehrdimensionale Arrays | 58 |
| 2.4 | Zusammenfassung | 59 |

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. In der Sprache 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 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

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

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

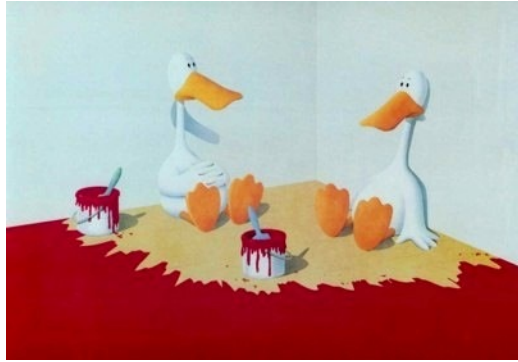


Abbildung 2.1. Für einen korrekt funktionierenden Algorithmus spielen Logik und zeitliche Reihenfolge der Einzelschritte eine wesentliche Rolle. ©1989 Michael Bedard „The Failure of Marxism“

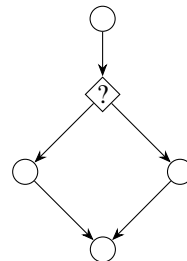
Auf diese Weise wurde das Wort Algorithmus zu einem Synonym für Rechenverfahren.

2.1 Logik und Verzweigung

2.1.1 Die Verzweigung

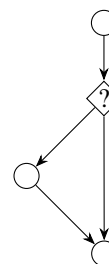
Die *Verzweigung* (auch: *Selektion*, *Auswahlstruktur* oder **if**-Anweisung) eine Anweisung, die abhängig von einer Bedingung zwischen alternativen Anweisungsfolgen auswählt. In Java hat sie die folgende Syntax:

```
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 *Boole'schen 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 5. 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 6. Alle algebraischen Vergleichsoperatoren können im Allgemeinen nur zwei Ausdrücke von elementarem Datentyp vergleichen. Zwei Strings sollten dagegen nur mit der Methode `equals` verglichen werden:

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

Der Vergleichsoperator `==` funktioniert bei Strings nicht immer! Entsprechend sollte man auch den Ungleichheitsoperator `!=` bei Strings nicht verwenden, sondern

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

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.

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 (^ oder !=)

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. Für Boole'sche Werte ist das äquivalent zu dem Ungleichheitsoperator `!=`. Im Unterschied zum XOR `^` ist dieser Operator jedoch nicht für bitweise Operationen geeignet, die wir im Folgenden betrachten werden.

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.

Manchen fällt es leichter, sich Wahrheitstabellen mit 0 und 1 zu merken, wobei $0 = \text{false}$ und $1 = \text{true}$ gilt, siehe Tab. 2.2. Diese „Mathematisierung“ von **false** und **true** hat die verblüffende Konsequenz, dass wir mit den logischen Operatoren `&&`, `||` und `^` algebraisch rechnen können, wenn wir sie wie folgt durch die Grundrechenarten ausdrücken:

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

| <i>a</i> | <i>b</i> | <i>a && b</i> | <i>a b</i> | <i>a != b</i> | <i>!a</i> |
|----------|----------|-----------------------|---------------|---------------|-----------|
| 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.

| <i>a</i> | <i>b</i> | <i>a && b</i> | <i>a b</i> | <i>a != b</i> | <i>!a</i> |
|----------|----------|-----------------------|---------------|---------------|-----------|
| 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.

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 *Boole'schen 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

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

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 $a1$, $a2$ und $a3$ 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 geprueft: ");
    }
    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. 137).

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]';
        }

        JOptionPane.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 \wedge 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 \wedge , 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.

| 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

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

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

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

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

nur noch den kleineren Speicherbereich zu nutzen. Hier kann schon eine einzige Wertzuweisung einer **long**-Variablen 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 Iterationen: Wiederholung durch Schleifen

Eine der wichtigsten Eigenschaften von Computern ist ihre Fähigkeit, sich wiederholende Tätigkeiten immer und immer wieder auszuführen. Eine wichtige Struktur zur Programmierung von Wiederholungen sind die „Iterationen“ oder Schleifen. 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.

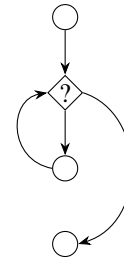
2.2.1 Die `while`-Schleife

Bei einer Iteration 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 7. 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. Ihre Die Syntax lautet

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

Hier bezeichnet *start* also die einmalig zum Start der Schleife durchzuführenden Initialisierungen von Variablen, *check* die vor jeder Iteration zu überprüfende Schleifenbedingung und *update* stets am Ende des Schleifenrumpfs auszuführenden Anweisungen. Die `for`-Schleife eignet sich besonders zur Implementierung einer *Zählschleife*, also für Wiederholungen, die durch einen Index oder einen Zähler gesteuert werden und bei denen bereits zum Schleifenbeginn klar ist, wie oft sie ausgeführt werden.

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

Beispielsweise kann man eine Zählvariable `i` von einem Startwert bis zu einen Endwert wie folgt hochzählen:

```
for (int i = 0; i < 10; i++) {
    System.out.print(i + ", ");
}
```

oder runterzählen:

```
for (int i = 10; i > 0; i--) {
    System.out.print(i + ", ");
}
```

In diesem Schleifenkonstrukt werden also im Gegensatz zu den `while`-Schleifen die für die Verwaltung einer Schleife wichtigen Anweisungen übersichtlich im Schleifenkopf aufgelistet. Auch mit einer `for`-Schleife kann man eine Endlosschleife programmieren, wenn nämlich die Update-Anweisung (und die Anweisungen im Schleifenrumpf) nicht dazu führen, dass die Schleifenbedingung falsch wird. Beispielsweise: `for(int i=10; i>0; i++) ...`

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 8. 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 verwenden?

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 9. 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.

2.3 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.

| 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

2.3.1 Was ist ein Array?

Ein *Array*⁴ (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. 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:

```
bestand = zugang1 + zugang2 + ... + 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 10. 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:

⁴ *array* – engl. für: Anordnung, Aufstellung, (Schlacht-)Reihe, Aufgebot

```
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

```
<Typ>[] a;      oder      <Typ> a[];
```

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

```
a = new <Typ> [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:

```
<Typ>[] a = {wert0, ..., wertn};  oder  <Typ>[] a = new <Typ>[] {wert0, ..., wertn};
```

(2.6)

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

2.3.2 Lagerbestände

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

```
import javax.swing.*;

/**
 * berechnet den Lagerbestand 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 `JTextArea` deklariert und sofort erzeugt. Die Klasse `JTextArea` 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!)

2.3.3 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 11. 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. (2.6) auf S. 55.

2.3.4 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 | | | |
| ↓ | | | |
| A | B | C | ← i |
| 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 Schiffeversenken 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,⁵ man braucht dann genau so viele Indizes wie man Dimensionen hat.

2.4 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;
}
```

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).

⁵ In der Physik und den Ingenieurwissenschaften ist der Begriff des „Tensors“ geläufig, der ist als mehrdimensionales Array von Zahlen darstellbar.

Schleifen

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

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

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

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

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:

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

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[] $a = \{wert_0, wert_1, \dots, wert_n\};$

- 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.

3

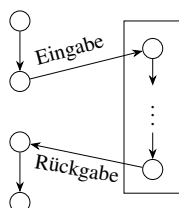
Subroutinen: Funktionen und statische Methoden

Kapitelübersicht

| | | |
|-------|--|----|
| 3.1 | Die Idee von Subroutinen | 62 |
| 3.2 | Funktionen in der Mathematik | 63 |
| 3.3 | Funktionen mit Lambda-Ausdrücken | 64 |
| 3.3.1 | Funktionen höherer Ordnung und Funktionalgebra | 67 |
| 3.4 | Statische Methoden | 70 |
| 3.4.1 | Eine erste Applikation mit statischen Methoden | 73 |
| 3.4.2 | Statische Methoden: Die Applikation Pythagoras | 75 |
| 3.4.3 | Vergleich von Funktionen und Methoden | 78 |
| 3.4.4 | Zusammenfassung | 80 |
| 3.5 | Rekursion | 82 |
| 3.5.1 | Architektur einer terminierenden Rekursion | 83 |
| 3.5.2 | Rekursion als Problemlösungsstrategie „Divide and Conquer“ | 85 |
| 3.5.3 | Rekursion versus Iteration | 87 |
| 3.5.4 | Klassifikation von Rekursionen | 91 |

3.1 Die Idee von Subroutinen

Ein wichtiges Mittel zur Strukturierung komplexer Softwaresysteme ist die „Modularisierung“ in kleinere Einheiten. Man spricht bei diesen Einheiten üblicherweise von *Subroutinen* oder *Unterprogrammen*. Die Idee ist dabei, einen ganzen Block von Anweisungen unter einem Namen zusammenzufassen, so dass er von einem anderen Programm verwendet werden kann, indem er mit diesem Namen „aufgerufen“ wird:



Die Subroutine startet dann zur Laufzeit des Hauptprogramms, führt ihre Anweisungen aus und übergibt dem Hauptprogramm das Ergebnis ihrer Anweisungen als *Rückgabe*, das dann damit seine Anweisungen weiterführt. Meist bekommt eine Subroutine sogenannte *Parameter*

als *Eingabe*, die sie zur Berechnung des Ergebnisses benötigt. Die entsprechenden englischen Ausdrücke lauten:

| | | | |
|-------------|--------------|------------------|---------------|
| Aufruf | Eingabe | Parameter | Rückgabe |
| <i>call</i> | <i>input</i> | <i>parameter</i> | <i>return</i> |

In vielen Programmiersprachen, so auch in Java, werden solche Subroutinen als Funktionen oder Methoden programmiert. Mit diesen beiden eng verwandten Begriffen werden wir uns in diesem Kapitel beschäftigen.

3.2 Funktionen in der Mathematik

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:

$$f : D \rightarrow W, \quad x \mapsto f(x). \quad (3.1)$$

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

$$f : \mathbb{N} \rightarrow \mathbb{N}, \quad x \mapsto x^2. \quad (3.2)$$

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

$$f : \mathbb{R} \rightarrow \mathbb{R}, \quad x \mapsto x^2. \quad (3.3)$$

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

$$f : \mathbb{R} \rightarrow [-1, 1], \quad x \mapsto \sin x. \quad (3.4)$$

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 12. 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:

$$f : \mathbb{R}^2 \rightarrow \mathbb{R}, \quad (x, y) \mapsto \sqrt{x^2 + y^2}. \quad (3.5)$$

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 13. 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.

3.3 Funktionen mit Lambda-Ausdrücken

Eine Möglichkeit, Funktionen in Java zu programmieren, sind die sogenannten „Lambda-Ausdrücke“. Als einführendes Beispiel betrachten wir das folgende Programm, das zwei Funktionen

$$f : \mathbb{R} \rightarrow \mathbb{R}, \quad x \mapsto 3x^2 \quad \text{und} \quad \text{pythagoras} : \mathbb{R}^2 \rightarrow \mathbb{R}, \quad (x, y) \mapsto \sqrt{x^2 + y^2} \quad (3.6)$$

implementiert, einzelne Werte einsetzt und für f eine Wertetabelle ausgibt:

```

1 import java.util.function.Function;
2 import java.util.function.BiFunction;
3
4 public class ErsteFunktionen {
5     public static Function<Double,Double> f = x -> 3*x*x;
6
7     public static BiFunction<Double,Double,Double> pythagoras =
8         (x,y) -> Math.sqrt(x*x + y*y);
9
10    public static void main(String... args) {
11        System.out.println("f(2) = " + f.apply(2.0));
12        System.out.println("pythagoras(3,4) = " + pythagoras.apply(3.,4.));

```



```

13
14     System.out.println("\nWertetabelle:\nx \t f(x)");
15     for (double x = 0; x <= 10; x += 0.5) {
16         System.out.printf("%1$.1f \t %2$.2f \n", x, f.apply(x));
17     }
18 }
19 }

```

Hier wird die Funktion $f(x) = 3x^2$ in Zeile 5 definiert. Sie wird als **public static** außerhalb der main-Methode deklariert mit Hilfe des Datentyps `Function`, der sich in dem Paket `java.util.function` befindet und in Zeile 1 importiert wurde. Dabei muss der Definitionsbereich D und der Wertebereich W der Funktion $f: D \rightarrow W$ in spitzen Klammern angegeben werden:

D W
 \downarrow \downarrow
`Function<Double,Double> f;`

Zu beachten ist hierbei, dass in den spitzen Klammern keine primitiven Datentypen stehen können, sondern nur deren Hüllklassen, siehe Tabelle 1.6 auf Seite 32. Möglich sind allerdings Arrays von primitiven Datentypen.

Die Definition der Funktion selbst geschieht mit Hilfe eines sogenannten Lambda-Ausdrucks. Ein allgemeiner *Lambda-Ausdruck* besteht aus einer Parameterliste mit Datentypen in runden Klammern, dem Pfeil-Symbol `->` und dem Funktionsrumpf:

$(\underbrace{\langle T1 \rangle x, \langle T2 \rangle y}_{\text{Parameterliste}}) \rightarrow \underbrace{x*x + y*y}_{\text{Funktionsrumpf}};$

Die Datentypen $T1$ und $T2$ können dabei auch Arrays sein, auch Arrays von primitiven Datentypen. Kann der Compiler den Datentyp der Parameter erkennen (zum Beispiel durch die Deklaration mit `Function` und den Datentypen in den spitzen Klammern), so können die Datentypen weggelassen werden. Hat man zusätzlich nur einen Parameter, so können auch die Klammern weggelassen werden:

```
x -> x*x + 1;
```

Man kann ebenso das reservierte Wort **return** verwenden, allerdings muss der Funktionsrumpf dann in geschweifte Klammern gesetzt werden:

```
(x) -> {return x*x + 1;;}
```

Diese Variante eignet sich für Funktionen, die kompliziertere Algorithmen implementieren. Die Syntax in Java für Lambda-Ausdrücke ist insgesamt recht vielfältig, eine Übersicht ist in Tabelle 3.1 aufgelistet. Grundsätzlich bleibt dabei natürlich das Format (Parameterliste) `->` {Funktionsrumpf} erhalten. Beispiele für syntaktisch korrekte Lambda-Ausdrücke sind also:

```
(int a, int b) -> a + b;
```

```
(int a, int b) -> {return a + b;}
```

```
(String s) -> System.out.println(s);
```

```
() -> 42;
```

| Syntax des Lambda-Ausdrucks | | Regel |
|---|-----------|--|
| Parameterlisten | | |
| (int x) | -> x + 1; | Parameterliste mit einem Parameter und expliziter Typangabe |
| int x | -> x + 1; | Falsch: Parameterliste mit Typangabe stets in Klammern! |
| (x) | -> x + 1; | Parameterliste mit einem Parameter ohne explizite Typangabe |
| x | -> x + 1; | Parameterliste mit einem Parameter ohne explizite Typangabe: Klammern dürfen weggelassen werden |
| (int x, short y) | -> x + y; | Parameterliste mit zwei Parametern und expliziten Typangaben |
| int x, short y | -> x + y; | Falsch: Parameterliste mit Typangaben stets in Klammern! |
| (x, y) | -> x + y; | Parameterliste mit zwei Parametern ohne explizite Typangaben |
| (x, short y) | -> x + y; | Falsch: keine Mischung von Parametern mit und ohne explizite Typangabe! |
| () | -> 42; | Parameterliste darf leer sein |
| Funktionsrümpfe | | |
| (x,y) -> x*x - y*y; | | Rumpf mit nur einem Ausdruck |
| x -> { ...Anweisungen; ... return wert; }; | | Rumpf als Block mit mehreren Anweisungen und abschließender return-Anweisung; geeignet für Funktionen, die komplexe Algorithmen implementieren |
| (int x) -> return x + 1; | | Falsch: die return-Anweisung nur innerhalb eines Blocks mit geschweiften Klammern {...}! |

Tabelle 3.1. Syntax von Lambda-Ausdrücken in Java. Die rötlich hinterlegten Ausdrücke sind falsch. Modifiziert nach [4]

(Die beiden letzten Lambda-Ausdrücke sind jedoch keine Funktionen, sondern ein Consumer und ein Supplier.)

Ein Lambda-Ausdruck erzeugt zunächst nur eine *anonyme* Funktion. Mit Hilfe der Datentypen *Function* und *BiFunction* haben wir oben diesen Ausdrücken eine Funktionsreferenz, also einen Variablennamen gegeben. Solche Funktionsreferenzen werden oft *Rückruffunktionen* oder *Callback-Funktionen* genannt, da sie nicht dort ausgeführt werden, wo sie übergeben werden, sondern in der Umgebung der aufrufenden Funktion, oft in der API, oder in unserem Beispiel in der *main*-Methode.

Welche Datentypen für Funktion gibt es in Java? Die verfügbaren Datentypen stehen in dem Paket *java.util.function* als *funktionale Interfaces* zur Verfügung. Die wichtigsten davon sind in der folgenden Tabelle aufgelistet:

| Datentyp („Funktionales Interface“) | Lambda-Ausdruck | Auswertungsmethode | Bemerkung |
|-------------------------------------|---------------------|-------------------------|--|
| <i>Function</i> <T,R> | x -> x*x - x + 1; | <i>apply</i> (T x) | ein Parameter vom Typ T, Rückgabe vom Typ R |
| <i>BiFunction</i> <T,U,R> | (x,y) -> x*x - y*y; | <i>apply</i> (T x, U y) | zwei Parameter vom Typ T und U, Rückgabe vom Typ R |
| <i>UnaryOperator</i> <T> | x -> x*x - x + 1; | <i>apply</i> (T x) | ein Parameter vom Typ T, Rückgabe vom Typ T |
| <i>BinaryOperator</i> <T> | (x,y) -> x - y; | <i>apply</i> (T x, T y) | zwei Parameter vom Typ T und Rückgabe vom Typ T |
| <i>Consumer</i> <T> | (T x) -> <void>; | <i>accept</i> (T x) | ein Parameter, keine Rückgabe (z.B. durch Aufruf einer void-Methode) |
| <i>Supplier</i> <T> | () -> const; | <i>get</i> () | kein Parameter, Rückgabe vom Typ T |
| <i>Predicate</i> <T> | x -> x < 5; | <i>test</i> (T x) | ein Parameter vom Typ T, Rückgabe vom Typ boolean |

(Die beiden Datentypen *UnaryOperator* und *BinaryOperator* sind spezielle Varianten der Da-

tentypen `Function` und `BiFunction`, in denen alle Datentypen ihrer Parameter und ihrer Rückgaben stets gleich sind.)

Eine deklarierte Funktion kann mit der Methode `apply` ausgewertet werden. Mit anderen Worten werden mit `apply` Werte in eine Funktion eingesetzt. Das folgende Programm wertet zwei Funktionen f und g aus:

```

1 import java.util.function.Function;
2 import java.util.function.BinaryOperator;
3
4 public class Funktionen {
5     public static Function<Integer, Integer> f = (x) -> x*x - 1;
6     public static Function<Integer, Integer> g = (x) -> x*x*x + 2*x - 1;
7     public static BinaryOperator<Function<Integer, Integer>> add = (f,g) ->
8         (x) -> f.apply(x) + g.apply(x);
9
10    public static void main(String... args) {
11        int x = 3;
12        String ausgabe = "f("+x+") = " + f.apply(x);
13        ausgabe += "\ng("+x+") = " + g.apply(x);
14        ausgabe += "\n(f+g)("+x+") = " + add.apply(f,g).apply(x);
15        javax.swing.JOptionPane.showMessageDialog(null, ausgabe, "Funktionen", -1);
16    }
17 }
```

Zu beachten ist, dass in einer Klasse die Funktionreferenzen verschiedene Namen haben müssen, auch wenn sie verschiedene Parameterlisten haben. (Anders als bei Methoden, die wir noch kennenlernen werden, wird für Lambda-Ausdrücke die Signatur der Referenz in Java erst bei ihrem Aufruf ausgewertet.)

3.3.1 Funktionen höherer Ordnung und Funktionalgebra

Eine *Funktion höherer Ordnung* ist eine Funktion, die als Argument eine Funktion erwartet oder deren Ergebnis eine Funktion ist. Beispielsweise kann man damit eine Algebra mit Funktionen programmieren, den sogenannten *Funktionalalkäül* oder *Funktionalgebra*. Man kann zum Beispiel die Summe $f + g$ zweier Funktionen $f, g : \mathbb{R} \rightarrow \mathbb{R}$ definieren, indem man

$$(f + g)(x) = f(x) + g(x) \quad (3.7)$$

definiert. In Java könnte der Additionsoperator für zwei Funktionen also wie folgt aussehen:

```

1 import java.util.function.BinaryOperator;
2 import java.util.function.Function;
3
4 public class Algebra {
5     public static Function<Integer, Integer> f = x -> x*x - 1;
6     public static Function<Integer, Integer> g = x -> x*x + 1;
7
8     public static BinaryOperator<Function<Integer, Integer>> add =
9         (f,g) -> (x -> f.apply(x) + g.apply(x));
10
11    public static void main(String... args) {
12        int x = 3, y = 4;
```

```

13     String ausgabe = "f("+x+") = " + f.apply(x);
14     ausgabe += "\ng("+x+") = " + g.apply(x);
15     ausgabe += "\n(f+g)("+x+") = " + add.apply(f,g).apply(x);
16     javax.swing.JOptionPane.showMessageDialog(null, ausgabe, "Funktionen", -1);
17 }
18 }

```

Hier wird in das funktionale Interface `BinaryOperator<T>` verwendet, das eine Funktion

$$\langle T, T \rangle \rightarrow \langle T \rangle$$

darstellt und zwei Argumente des gleichen Datentyps zu einem neuen Wert verknüpft. In unserem Falle ist in den Zeile 5 und 6 also

$$T = \text{Function}\langle \text{Integer}, \text{Integer} \rangle,$$

d.h. der binäre Operator `add`

$$\text{BinaryOperator}\langle \overbrace{\text{Function}\langle \text{Integer}, \text{Integer} \rangle}^T \rangle$$

verknüpft zwei Funktionen f und g des Typs $T = \text{Function}\langle \text{Integer}, \text{Integer} \rangle$ zu einer neuen Funktion desselben Typs. In Zeile 15 werden daher richtigerweise *zwei* Auswertungen ausgeführt:

$$\text{add.apply}(f,g).\text{apply}(x);$$

Die erste wendet den Operator `add` auf die Funktionen f und g an, danach wird x in die daraus resultierende Funktion $(f + g)$ eingesetzt, wie in Gleichung (3.7) vorgesehen.

Auch die Verkettung mit anderen Rechenoperationen wie Subtraktion, Multiplikation oder Division ist möglich, wie die folgenden Beispiele zeigen.

Beispiel 3.1. (*Gewinnfunktion eines Unternehmens*) Die Preis-Absatz-Funktion $p(x)$ bezeichnet allgemein den Preis pro Mengeneinheit eines Gutes, den ein produzierendes Unternehmen für die Menge x des Gutes am Markt absetzen kann. Typische Preis-Absatz-Funktionen sind monoton fallend. Dann gibt die Umsatzfunktion

$$U(x) = x p(x) \quad (3.8)$$

den Umsatz des Unternehmens bei der abgesetzten Menge x ME (ME = Mengeneinheiten) des Gutes an. (Die Umsatzfunktion wird oft auch Erlösfunktion genannt.) Ist weiters die Kostenfunktion $K(x)$ gegeben, die die Gesamtkosten des Unternehmens zur Herstellung der Menge x ME des Gutes beziffert, so ergibt sich die Gewinnfunktion $G(x)$ durch

$$G(x) = U(x) - K(x). \quad (3.9)$$

Details und Hintergründe dazu sind in [7, §2.5.2] zu finden. Nun sind mit den Gleichungen (3.8) und (3.9) sowohl die Umsatzfunktion als auch die Gewinnfunktion durch Funktionenalgebra aus $p(x)$ und $K(x)$ bestimmbar. D.h. aus der Kenntnis der Preis-Absatz-Funktion $p(x)$ des Gutes und seiner Kostenfunktion $K(x)$ kann das Unternehmen sofort die Gewinnfunktion $G(x)$ bestimmen und daraus die optimal zu produzierende Menge x des Gutes berechnen, die die Gewinnfunktion maximiert. Sind zum Beispiel die Kostenfunktion $K(x)$ und die Preis-Absatzfunktion $p(x)$ für ein Stückgut x durch

$$p : \mathbb{N}_0 \rightarrow \mathbb{R}, \quad x \mapsto 60 - 0,5x, \quad K : \mathbb{N}_0 \rightarrow \mathbb{R}, \quad x \mapsto 10x + 450 \quad (3.10)$$

gegeben, für das x also nur nichtnegative ganzzahlige Werte annehmen kann, so kann man sie in Java einfach durch das folgende Programm implementieren.

```

1 import java.util.function.UnaryOperator;
2 import java.util.function.BinaryOperator;
3 import java.util.function.Function;
4
5 public class Kostenfunktion {
6     /** Preis-Absatz-Funktion.*/
7     public static UnaryOperator<Function<Integer,Double>> U =
8         p -> (x -> x * p.apply(x));
9
10    /** Gewinnfunktion.*/
11    public static BinaryOperator<Function<Integer,Double>> G =
12        (K,U) -> (x -> U.apply(x) - K.apply(x));
13
14    public static void main(String... args) {
15        Function<Integer,Double> p = x -> 60. - 0.5*x;
16        Function<Integer,Double> K = x -> 10.*x + 450.;
17
18        System.out.println("\nWertetabelle der Gewinnfunktion\n\nx \t G(x)");
19        for (int x = 0; x <= 100; x += 5) {
20            System.out.printf("%1$d \t %2$.2f \n",
21                x,
22                G.apply(K, U.apply(p))
23                    .apply(x)
24            );
25        }
26    }
27 }

```

Anhand der in der main-Methode ausgegebenen Wertetabelle erkennt man, dass das Gewinnmaximum bei $x = 50$ liegt. (Wer möchte, kann dieses experimentelle Ergebnis analytisch durch Ableitung der Gewinnfunktion nachrechnen!) \square

Beispiel 3.2. In der Technischen Informatik werden Logikgatter als elektronische Bauelemente von Schaltkreisen betrachtet. Zu ihrer Darstellung werden normierte Symbole verwendet, in Abbildung 3.1 sind die nach IEC gültigen Schaltzeichen aufgelistet. Eine zusammengesetzte

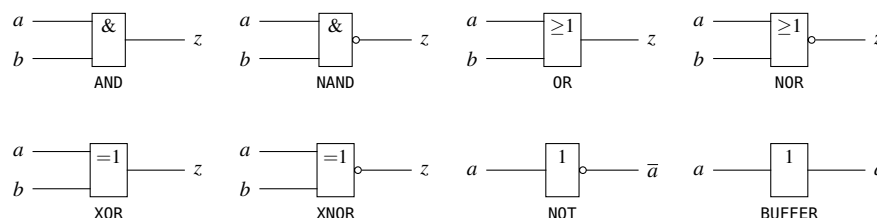


Abbildung 3.1. Die acht Logikgatter nach gültiger IEC-Norm [1, §9.3.4].

logische Schaltung zum Beispiel für ein XOR kann so zum Beispiel aus den Gattern NOT, AND und OR gebildet werden, siehe Abbildung 3.2. Um die Simulation einer solchen Schaltung zu programmieren, kann man die zugrunde liegenden Gatter als Boole'sche Operatoren deklarieren und die Zusammensetzung daraus als neuen Operator. Ein *binärer* Boole'scher Operator f kann nämlich als eine Funktion

$$\{\text{false}, \text{true}\}^2 \rightarrow \{\text{false}, \text{true}\}, \quad (a, b) \mapsto f(a, b)$$

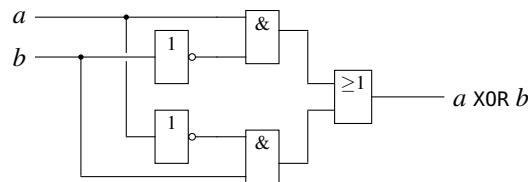


Abbildung 3.2. Schaltung für XOR aus den universellen Gattern NOT, AND und OR.

mit zwei Parametern aufgefasst werden, und ein *unärer* Boole'scher Operator als eine Funktion mit

$$\{\text{false}, \text{true}\} \rightarrow \{\text{false}, \text{true}\}, \quad a \mapsto f(a)$$

mit einem Parameter. Für solche Funktionen, deren Parameterdatentypen gleich dem Datentyp des Wertebereichs sind, gibt es in Java die praktischen Funktionstypen `UnaryOperator<T>` und `BinaryOperator<T>`:

$$\text{UnaryOperator}<T> \ f: <T> \rightarrow <T>, \quad \text{BinaryOperator}<T> \ f: <T>^2 \rightarrow <T>.$$

Sie werden im folgenden Programm für die logischen Operatoren NOT, AND und OR zur Konstruktion des XOR-Gatters verwendet:

```

1 import java.util.function.BinaryOperator;
2 import java.util.function.UnaryOperator;
3
4 public class Schaltalgebra {
5     public static UnaryOperator<Boolean> NOT = (a) -> !a;
6     public static BinaryOperator<Boolean> AND = (a,b) -> a && b;
7     public static BinaryOperator<Boolean> OR = (a,b) -> a || b;
8
9     public static void main(String... args) {
10         BinaryOperator<Boolean> XOR = (a,b) ->
11         OR.apply(
12             AND.apply(a, NOT.apply(b)),
13             AND.apply(NOT.apply(a), b)
14         );
15
16         // Ausgabe:
17         System.out.println("XOR \t| \tfalse\ttrue ");
18         System.out.println("-----");
19         System.out.println("false\t| \t" + XOR.apply(false,false) + "\t" + XOR.apply(false,true));
20         System.out.println("true \t| \t" + XOR.apply(true,false) + "\t" + XOR.apply(true,true));
21     }
22 }

```

Die Reihenfolge der Zusammensetzung im Programm ergibt sich dabei, indem man das Schaltbild „rückwärts“ liest, da man die Klammerung ja von innen nach außen ausführen muss. \square

3.4 Statische Methoden

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 14. 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 3.3.

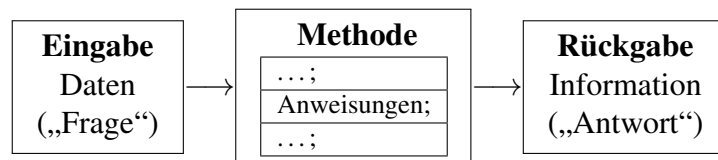
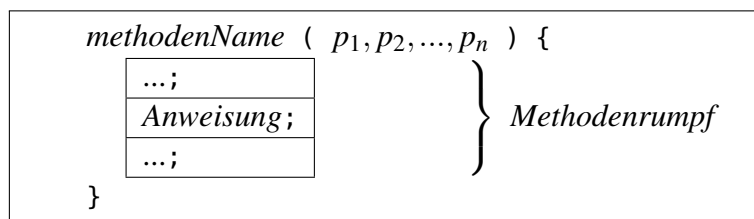


Abbildung 3.3. Schema einer Methode

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 { ... } umschlossen sind. Hinter jede Anweisung schreibt man ein Semikolon.



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 { ... } umschlossen.

Beispiele

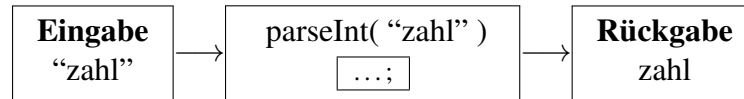
Wir haben bereits einige Methoden verwendet. Betrachten wir sie mit unseren neuen Erkenntnissen über Methoden.

- 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“).

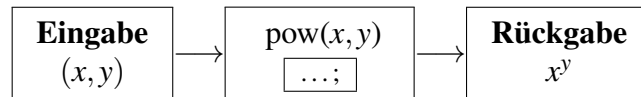


Die Punkte (...) deuten Anweisungen an, die wir nicht kennen (und auch nicht zu kennen brauchen).

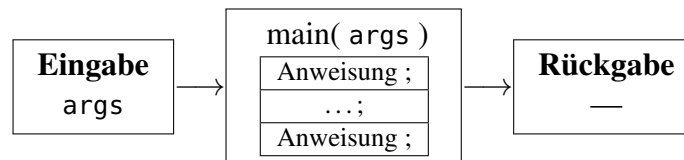
- Die Methode `parseInt` der Klasse `Integer`. Sie erwartet einen `String` als Eingabeparameter und gibt einen `int`-Wert zurück.



- 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[]` erwartet¹ und liefert keinen Rückgabewert.



Merkregel 15. 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

`<Rückgabotyp> methodName(<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 <Rückgabotyp> methodName(<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.

¹Das ist ein so genanntes „Array“ von Strings, s.u.

Methode ohne Rückgabe

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

```
public static void methodName(<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.)

Das reservierte Wort **public** ist im Allgemeinen zwar nicht zwingend erforderlich für Methoden, es ermöglicht aber einen uneingeschränkten Zugriff von außen (siehe S. 123). In der Regel sind statische Methoden stets „öffentlich“.

Merkregel 16. 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. 3.4 darstellen.

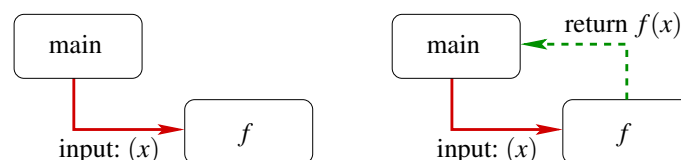


Abbildung 3.4. Diagramm eines Aufrufs einer Methode f mit Rückgabe, aus der `main`-Methode mit der Anweisung $y = f(x)$. Zunächst bewirkt der Aufruf mit dem Eingabeparameter x , dass der zur Laufzeit aktuelle Wert von x , das „Argument“ (siehe S. 77), in die Funktion f eingesetzt wird (linkes Bild). 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.

3.4.1 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 hypothenuse( 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 = " + hypothenuse(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 17. 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 |
|--|
| |
| int quadrat(int) double hypothenuse(double , double) |

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 `hypothenuse` 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.

3.4.2 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 hypothenuse( 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 = hypothenuse( a, b );                                       // (6)
    }
}
```

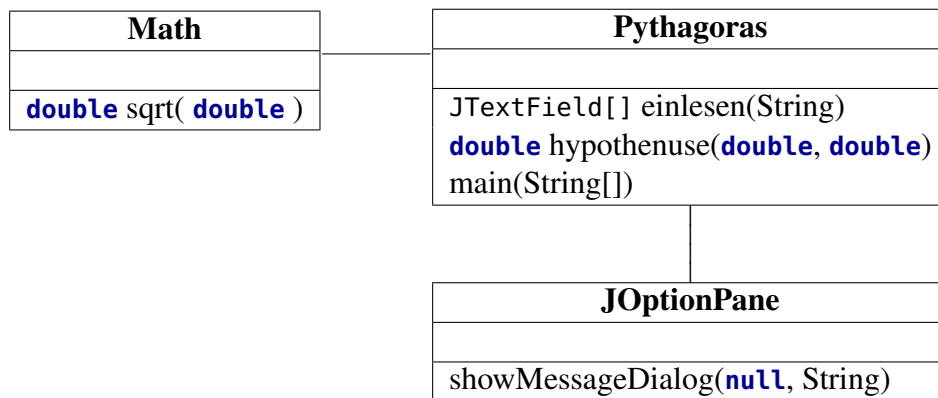
```

        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 `hypotenuse` 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 `hypotenuse` werden dann ausgegeben.

Die Methoden `einlesen` und `hypotenuse`

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. 3.5 dargestellt,

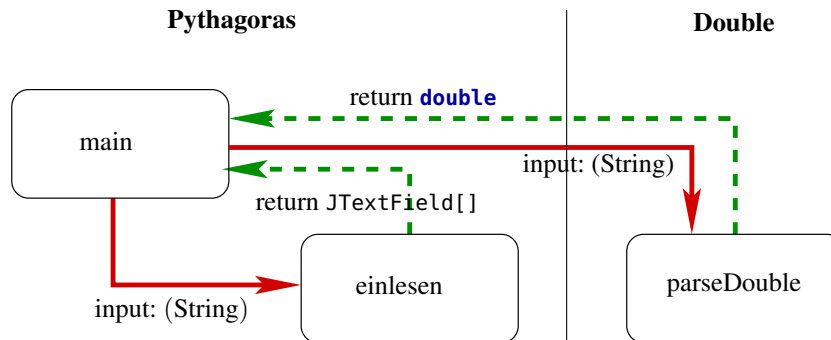


Abbildung 3.5. Aufrufstruktur der Anweisungen `feld = einlesen(...);` und `a = Double.parseDouble(...)` in der Applikation Pythagoras, Anmerkung (5). Über den Methoden sind die Klassen aufgeführt, zu denen sie jeweils gehören.

ist die `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. 3.6. Hier wird

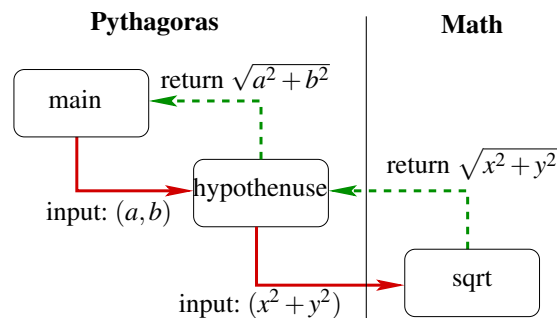


Abbildung 3.6. Aufrufstruktur bei der Anweisung `c = hypethenuse(a,b)` in der Applikation Pythagoras, Anmerkung (6).

die Methode `hypotenuse` mit zwei Werten (a,b) aufgerufen. Mit der Deklaration in (3) heißen diese beiden Parameter innerhalb der Methode `hypotenuse` nun jedoch x und y ! Hier werden als die Werte der Variablen (a,b) für die Variablen (x,y) eingesetzt.

Merkregel 18. Allgemein werden in der Informatik die Variablen bei der Deklaration einer Methode *Parameter* genannt, während die konkreten Werte beim Aufruf einer Methode *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 3.2 ab Seite 63 über Funktionen und Variablen an.)

Lokale Variablen

Sieht man sich den Methodenaufruf in Anmerkung (6) an, so könnte man auf die Frage kommen: Kann der Aufruf `hypothenuse(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 `hypothenuse` 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. 77) 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 `hypothenuse`, 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.

3.4.3 Vergleich von Funktionen und Methoden

Grundsätzlich haben wir in diesem Kapitel zwei Möglichkeiten kennen gelernt, Subroutinen und mathematische Funktionen in Java zu programmieren: einerseits die funktionalen Datentypen (vor allem `Function<D,W>` und `BiFunction<T,U,W>`), andererseits statische Methoden. Beide Möglichkeiten haben jeweils ihre Vorzüge und ihre Nachteile.

- Funktionen:

- Vorteile von Funktionen:

1. Funktionen können elegant als Lambda-Ausdruck implementiert werden.
2. Funktionen können mit einer Variablen als Referenz gespeichert und so als Argument einer Funktion oder Methode übergeben werden.

- Nachteile von Funktionen:

1. Argumente müssen etwas umständlich mit `apply` eingesetzt werden (statt wie in der Mathematik einfach mit runden Klammern).
2. Funktionen in Java können höchstens zwei Parameter (für einen Parameter `Function<D,W>`, für zwei `BiFunction<D1,D2,W>`) oder ein Array von Werten desselben Datentyps übergeben bekommen.
3. Funktionen ermöglichen keine Rekursionen, können sich also nicht selbst aufrufen. (Wir werden im nächsten Abschnitt das Konzept der Rekursion erfahren.)

4. Lambda-Ausdrücke haben in Java nur ein gekapseltes Speicherkonzept, solange sie außerhalb von Methoden deklariert werden; sind sie innerhalb von Methoden deklariert, dürfen ihre Parameternamen nicht bereits als lokale Variablen deklariert sein. (Allerdings kann dieser Nachteil auch als Vorteil gewendet werden, da damit sogenannte Closures programmiert werden können.²)

- Methoden:

- Vorteile von Methoden:

1. Statische Methoden haben ein klares gekapseltes Speicherkonzept, es können höchstens statische Attribute derselben Klasse (z.B. Konstanten) verwendet werden.
2. Argumente werden bei Methodenaufrufen einfach in runden Klammern übergeben.
3. Mit Methoden können Rekursionen programmiert werden.

- Nachteile von Methoden:

1. Methoden können nicht in Referenzvariablen gespeichert und so als Parameter anderen Methoden oder Funktionen übergeben werden; mit statischen Methoden können also nicht Funktionen höherer Ordnung und Funktionalgebren programmiert werden.³
2. Eine Methode muss stets mit einer **return**-Anweisung beendet werden (außer natürlich, wenn es sich um eine **void**-Methode handelt).

Methoden sind in Java von Anfang an ein grundlegendes Konzept gewesen, ganz im Gegensatz zu den erst mit Java 8 eingeführten Funktionen und Lambda-Ausdrücken.⁴ Entsprechend sind Funktionen auch in der Syntax von Java (noch?) nicht so integriert wie Methoden, was das umständliche Einsetzen mit `apply` erklärt. Viel einschneidender ist allerdings das Manko, dass Funktionen zwar andere Funktionen, nicht aber sich selbst aufrufen können, also keine Rekursion ermöglichen. Rekursion jedoch ist eine der zentralen Entwurfsprinzipien der funktionalen Programmierung.

Trotz ihrer Nachteile sollten jedoch nach meiner Ansicht Funktionen möglichst überall dort verwendet werden, wo mathematische Funktionen implementiert werden sollen. Bis auf Rekursionen und die eingeschränkte Anzahl von maximal zwei Parametern kann eine Funktion dasselbe ausführen wie eine statische Methode, man muss halt nur in der Syntax anders denken, also

`Function<D,W> f = ...`

anstatt

`<W> f(<D> x) { ...return ...;}`

schreiben. Wegen der einfacheren Aufrufsyntax sollte man jedoch in als APIs bereitgestellte Programmbibliotheken eher geeignete statischen Methoden implementieren, so dass ein Programmierer, der die Funktionalität verwenden will, nicht umständlich „`f.apply(x)`“ programmieren muss, sondern „`f(x)`“. Zusammengefasst sollte man also intern möglichst Funktionen, nach außen aber eher statische Methoden programmieren. Dieser Gedanke wird anhand des folgenden Beispiels näher erläutert.

² <https://docs.oracle.com/javase/tutorial/java/java00/lambdaexpressions.html#accessing-local-variables>

³ Mit der Objektorientierung, die wir noch kennen lernen werden, kann man grundsätzlich sehr wohl Funktionen- oder Operatoralgebren programmieren: aber eben nicht mit statischen Methoden.

⁴ Die Einführung von Java SE 8 fand 2014 statt, da war Java schon fast 19 Jahre alt.

Beispiel 3.3. Modifizieren wir das Beispiel 3.1 auf S. 68 so, dass in der main-Methode nur die Preis-Absatz-Funktion und die Kostenfunktion definiert werden müssen und damit die Umsatz- und die Gewinnfunktion automatisch berechnet werden, also

$$\text{Geg.: } p(x), K(x) \quad \Longrightarrow \quad \text{berechne } G(U(p), K)(x)$$

so sollten nach den Prinzip „intern Funktionen, extern Methoden“ U und G als statische Methoden implementiert werden, also zum Beispiel:

```
public static Function<Integer,Double> U(Function<Integer, Double> p) {
    return x -> x * p.apply(x);
}
```

Dann kann man $U(p)$ als Funktion in der main-Methode aufrufen:

```
public static void main(String... args) {
    ...
    U(p).apply(x);
    ...
}
```

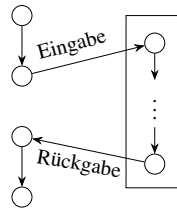
Insgesamt erhalten wir so das folgende Programm:

```
1 import java.util.function.Function;
2
3 public class Kostenfunktion_2 {
4     /** Umsatzfunktion.*/
5     public static Function<Integer,Double> U(Function<Integer,Double> p) {
6         return x -> x * p.apply(x);
7     }
8
9     /** Gewinnfunktion.*/
10    public static Function<Integer,Double> G(
11        Function<Integer,Double> U, Function<Integer,Double> K
12    ) {
13        return x -> U.apply(x) - K.apply(x);
14    }
15
16    public static void main(String... args) {
17        Function<Integer,Double> p = x -> 60. - 0.5*x;
18        Function<Integer,Double> K = x -> 10.*x + 450.;
19
20        System.out.println("\nWertetabelle der Gewinnfunktion\n\nx \t G(x)");
21        for (int x = 0; x <= 100; x += 5) {
22            System.out.printf("%1$d \t %2$.2f \n", x, G(U(p),K).apply(x));
23        }
24    }
25 }
```

□

3.4.4 Zusammenfassung

- Die Idee einer Subroutine ist es, einen Block von Anweisungen als Unterprogramm von dem Hauptprogramm aus aufrufbar zu machen:



Eine Subroutine wird dabei meist mit Parametern als Eingabe aufgerufen, die sie zur Berechnung ihrer Rückgabe als Ergebnis benötigt.

- Eine mathematische Funktion ist vollständig 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, $W \subset \mathbb{R}$.
- Funktionen können in Java außerhalb der main-Methode mit einem Lambda-Ausdruck implementiert werden:

```
public static Function<D,W> f = x -> ...;
```

Die Datentypen D und W müssen dabei eine Klasse (bzw. ein Interface) oder ein Array sein; statt der primitiven Datentypen müssen deren Hüllklassen verwendet werden. Deklariert man Funktionen innerhalb einer Methode, so müssen **public static** weggelassen und bereits vorher deklarierte lokale Variablen beachtet werden.

- Ähnlich können Anweisungen als Block zu einer Methode zusammengefasst werden.
- Die Deklaration einer statischen Methode darf sich nicht innerhalb einer anderen Methode befinden und besteht aus dem Methodennamen, der Parameterliste und dem Methodenkörper:

```
public static Datentyp der Rückgabe methodenName(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.

3.5 Rekursion

Was geschieht eigentlich, wenn eine Methode sich selber aufruft? Ist eine solche Konstruktion überhaupt möglich? Zumindest meldet der Compiler keine Probleme, wenn man die folgende Methode zu implementieren versucht:

```
public static int f(int x) {
    return f(x);
}
```

also mathematisch $f(x) = f(x)$. Versucht man jedoch zum Beispiel, den Funktionswert $f(3)$ berechnen zu lassen, so kommt es zum Absturz des Programms. OK, sind also Selbstaufrufe von Funktionen in der Praxis doch nicht möglich?

Ganz im Gegenteil, Selbstaufrufe heißen in der Informatik *Rekursionen* und bilden ein mächtiges und sehr wichtiges Konstruktionsprinzip für Algorithmen. Allerdings muss man ihre Funktionsweise beherrschen und bei ihrem Einsatz einige Regeln beachten. Rekursionen entsprechen in den Ingenieurwissenschaften den Rückkopplungen technischer Systeme, zum Beispiel bei einem Bildschirm, der die Aufnahmen der Kamera wiedergibt, die ihn gerade filmt, oder bei einem Mikrophon, das seinen gerade selbst aufgenommenen und per Lautsprecher ausgegebenen Schall wieder aufnimmt. Rekursionen gibt es auch in der Kunst und in der Literatur,

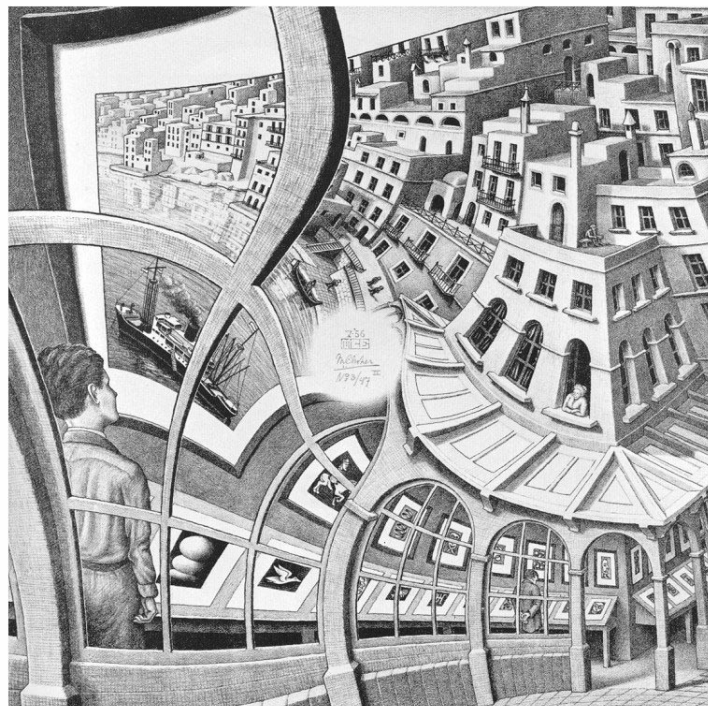
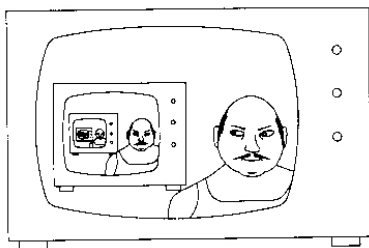


Abbildung 3.7. Rekursionen: Rückkopplung und M. C. Eschers „Kunstdruckgalerie“ (*Prententoonstelling*) von 1956 als *Mise en abyme*. Quellen: [10], Wikimedia.org

dort als *Mise en abyme* (französisch: „in den Abgrund gestellt“) bezeichnet. Hier ist zum Beispiel der Ersteller eines Textes selber Teil dieses Textes, („als mich der Dichter erfand, hatte er ganz was anderes mit mir im Sinn“⁵), oder ein Bild ist Teil eines Motivs in diesem Bild (Abbildung 3.7). (Zur Mathematik des *Mise en abyme* in der Kunst siehe auch die Internetquellen [LdS, Le])

⁵ E.T.A. Hoffmann: *Prinzessin Brambilla*, <https://books.google.de/books?id=bACoDAAQBAJ&pg=PG129>

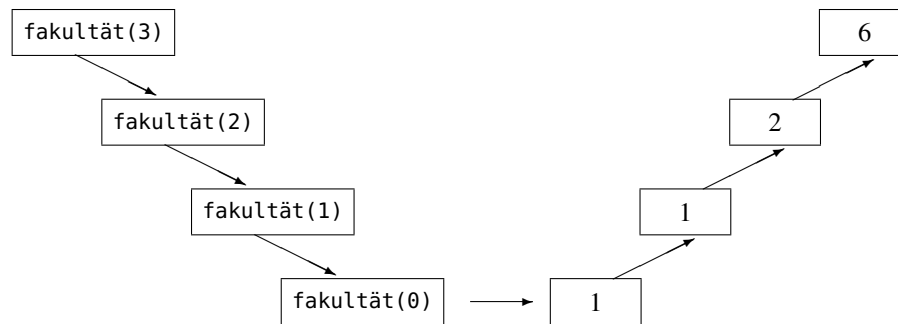


Abbildung 3.8. Aufrufablauf der Fakultätsfunktion anhand eines Aufrufsbaums.

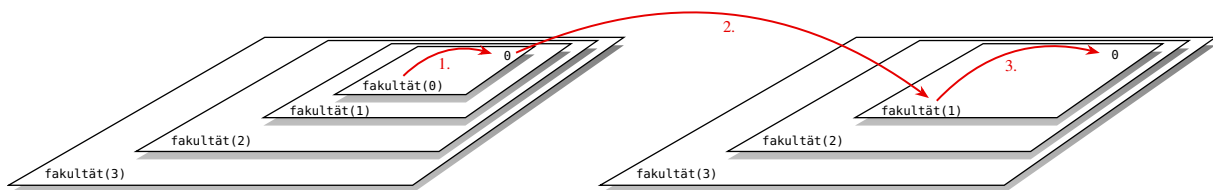


Abbildung 3.9. Aufrufablauf der Fakultätsfunktion anhand eines Stapels „Arbeitsblätter“.

in Abbildung 3.9. Hier wird jeder Rekursionsaufruf als ein neues Arbeitsblatt dargestellt, das auf den bisherigen Stapel kommt, für sich abgearbeitet wird und nach Erledigung „seiner“ Aufgaben sein Ergebnis an das darunterliegende Blatt zurückgibt. Erst beim Basisfall erreicht der Stapel Arbeitsblätter seine größte Höhe und wird danach wieder kleiner. (Dieses Bild entspricht übrigens ziemlich genau dem tatsächlichen Ablauf der Rekursionsaufrufe im Stack des Arbeitsspeichers!) □

Insgesamt können wir daraus die folgenden allgemeinen Beobachtungen ableiten. Es muss (mindestens) einen Basisfall geben, für den die Lösung bekannt ist; hier ist es der Fall $0! = 1$. Für jeden einzelnen Rekursionsschritt muss definiert sein, wie mit dem jeweils zurückgegebenen Ergebnis verfahren werden soll; hier ist es die Multiplikation $n \cdot f(n-1)$. Im Allgemeinen können im Rekursionsschritt auch mehrere Selbstaufrufe geschehen.

Beispiel 3.5. (*Das Maximum eines Arrays finden*) Betrachten wir als ein weiteres Beispiel das Problem, den maximalen Eintrag eines Arrays zu bestimmen. Der Algorithmus besteht hierbei daraus, das Array a von hinten her ($i = a.length - 1$) zu durchlaufen und sich den bisher gefundenen maximalen Eintrag max zu merken. Sind wir am Anfang des Arrays angekommen ($i=0$), hat max den maximalen Eintrag des gesamten Arrays gespeichert. Im Gegensatz zu einer Iteration sind in einer Rekursion aber Werte in lokalen Variablen (hier i und max) nicht in der aufgerufenen oder aufrufenden Methode verfügbar, sondern müssen von der einen in die andere übergeben werden. Für den Aufruf müssen wir also jeweils den aktuell bestimmten Wert max und den aktuellen Index als Parameter übergeben, und dann den frisch berechneten Wert für max per **return** zurückgeben.

```

1 public class Searchmax {
2     /** Findet den maximalen Eintrag des Arrays a ab Index i.*/
3     public static int searchmax(int[] a, int max, int i) {
4         if (a[i] > max) {
5             max = a[i];
6         }
7
8         if (i == 0) { // Basisfall

```

```

9      return max;
10     } else { // Rekursionsschritt:
11         return searchmax(a, max, i - 1);
12     }
13 }
14
15 public static void main(String... args) {
16     int[] a = {3, 7, 5};
17     System.out.println("max: " + searchmax(a, Integer.MIN_VALUE, a.length - 1));
18 }
19 }

```

Der Ablauf des Programms ist in Abbildung 3.10 dargestellt. □

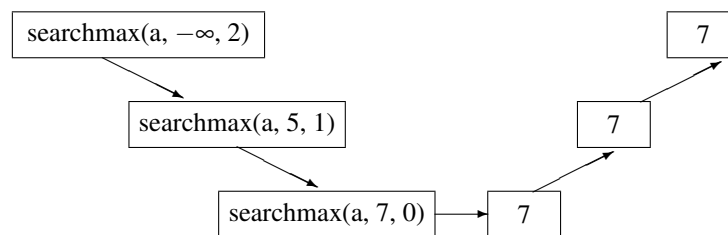


Abbildung 3.10. Aufrufabfolge von searchmax.

3.5.2 Rekursion als Problemlösungsstrategie „Divide and Conquer“

Als Konstruktionsprinzip spielt die Rekursion eine bedeutende Rolle beim Entwurf von Algorithmen, also von Verfahren zur Lösung gegebener Probleme. Die Idee ist dabei wie folgt: Wenn wir ein „großes“ Problem zu lösen haben, aber

- erstens wissen, wie wir es in „kleinere“ Teilprobleme zerlegen können,
- und zweitens, wie „das kleinste“ Problem zu lösen ist,

so ist das Problem auf natürliche Weise mit einer Rekursion lösbar. Hierbei wird „das kleinste“ Problem als Basisfall gelöst, während die Aufteilung in „kleinere“ Teilprobleme im Rekursionsschritt geschieht:

```

sucheLösung(Parameter eines Problems) {
    if (Parameterwerte klein genug) { // Basisfall
        return Lösung;
    } else { // Rekursionsschritt
        teile das Problem in Teilproblem(e);
        sucheLösung(Parameter des Teilproblems);
    }
}

```

Hierbei kann es auch mehrere Basisfälle, also mehrere „kleine“ lösbare Teilprobleme geben, oder auch eine Aufteilung in mehrere Teilprobleme.

Definition 3.6. Die Lösungsstrategie, ein Problem in mehrere Teilprobleme aufzuteilen und jeweils rekursiv zu lösen, heißt *Divide and Conquer* („Teile und beherrsche“). □

Wir wollen diese Problemlösungsstrategie anhand eines Klassikers der Algorithmik erläutern, den Türmen von Hanoi.

Die Türme von Hanoi

Einer Legende nach haben die Mönche in einem Kloster bei Hanoi die schicksalsschwere Aufgabe, einen Stapel 64 gelochter goldener Scheiben zu versetzen, die auf einem von drei Stäben der Größe nach sortiert liegen, die größte liegt unten, die kleinste oben. Aufgrund des beson-

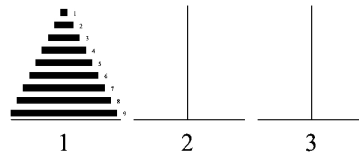


Abbildung 3.11. Türme von Hanoi.

ders schweren Gewichts der Scheiben kann pro Tag nur eine von ihnen auf einen der drei Stäbe bewegt werden, und es darf nie eine größere auf einer kleineren Scheibe liegen. Nehmen wir an, die Scheiben sollen von dem Stapel Nummer 1 auf den Stapel Nummer zwei bewegt werden.

Zur Lösung dieser Aufgabe wollen wir einen Algorithmus erstellen, der eine Liste der nacheinander auszuführenden Züge anzeigt. Die Angabe

$$1 \rightarrow 3$$

bedeutet hierbei: „Bewege die oberste Scheibe von Stapel 1 auf den Stapel 3.“ Für den Fall von nur drei Scheiben lautet eine Anweisungsliste beispielsweise

$$1 \rightarrow 3, \quad 1 \rightarrow 2, \quad 3 \rightarrow 2.$$

Versuchen Sie es selbst einmal für $n = 4$ Scheiben. (Sie sollten dafür mindestens 7 Züge brauchen.)

Wie könnte ein rekursiver Algorithmus aussehen, der eine solche Anweisungsliste zur Lösung des Problems ausgibt? Nach unserer Problemlösungsstrategie benötigen wir dazu (mindestens) einen Basisfall und den Rekursionsschritt. Zum Basisfall: Wie sieht der Algorithmus aus, wenn der Turm aus nur einer Scheibe besteht, die wir von Stapel start nach Stapel ziel versetzen möchten? Dann brauchen wir nur „start \rightarrow ziel“ auszugeben. Für den Rekursi-



Abbildung 3.12. Rekursionsschritt zur Lösung der Türme von Hanoi aus drei Teilschritten: Bewege Turm $(n-1)$ auf 3, dann $1 \rightarrow 2$, und dann Turm $(n-1)$ auf 2.

onsschritt müssen wir überlegen, wie wir einen Turm (n) mit n Scheiben von start nach ziel versetzen, wenn wir wissen, wie wir einen Turm $(n-1)$ mit $(n-1)$ Scheiben von a nach b versetzen. Die Lösung ist dann aber nicht schwer, wie in Abbildung 3.12 skizziert:

$$\text{Turm } (n-1): 1 \rightarrow 3, \quad \text{Scheibe } n: 1 \rightarrow 2, \quad \text{Turm } (n-1): 3 \rightarrow 2. \quad (3.11)$$

Bezeichnen wir den Lösungsalgorithmus mit `turm`, die Anzahl der zu bewegenden Zeilen mit n und die drei Stapelnummern mit `start`, `ziel` und `tmp`, so erhalten wir die folgende Rekursion.

```
public static void turm(int n, int start, int ziel, int tmp) {
    if (n == 1) { // Basisfall: Turm mit nur einer Scheibe
        System.out.println(start + " -> " + ziel);
    } else {
```

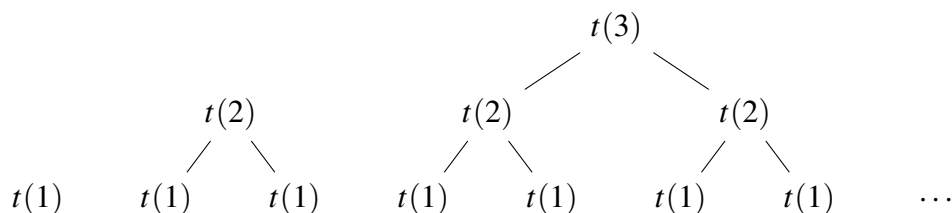


```

    turm(n-1, start, tmp, ziel); // Turm (n-1) temporär auf Stab tmp
    System.out.println(start + " -> " + ziel); // unterste Scheibe auf ziel
    turm(n-1, tmp, ziel, start); // Turm (n-1) von tmp auf ziel
}
}

```

Laufzeitbetrachtung. Wenn pro Tag eine Scheibe bewegt wird, wieviel Tage dauert dann die Versetzung eines Turms von n Scheiben? Um diese Zahl zu bestimmen, machen wir uns zunächst den Aufrufablauf der Rekursion klar. Schreiben wir kurz $t(n)$ für den Aufruf von $\text{turm}(n, \dots)$, so haben wir für $t(1)$, also eine Scheibe, nur einen einzigen Zug (Basisfall!), für $t(2)$ haben wir einen Zug im Rekursionsschritt und zwei Aufrufe $t(1)$, also insgesamt drei Züge:



Da in jedem Rekursionsschritt jeweils zwei rekursive Aufrufe stattfinden, verdoppelt sich pro Rekursionsebene n die Anzahl der Rekursionen. Im Gegensatz zu dem linearen Aufrufablauf der Rekursionen für die Fakultät oder `searchmax` entsteht hier also ein Aufrufbaum, genauer gesagt ein *binärer Baum*. Bezeichnen wir die Anzahl der Züge für n Scheiben mit $f(n)$, so erhalten wir also die folgende Wertetabelle:

| n | 1 | 2 | 3 | 4 | ... | n |
|--------|---|---|---|----|-----|-----------|
| $f(n)$ | 1 | 3 | 7 | 15 | ... | $2^n - 1$ |

(3.12)

d.h.

$$f(n) = 2^n - 1. \quad (3.13)$$

3.5.3 Rekursion versus Iteration

Rekursion ist aus Sicht der Programmierung eine Möglichkeit, einen einmal implementierten Anweisungsblock mehrfach ablaufen zu lassen. Wie die Iteration, also die Programmierung mit Schleifen, ist sie daher eine Kontrollstruktur zur Wiederholung. Tatsächlich weiß man aus der theoretischen Informatik, dass beide Kontrollstrukturen äquivalent sind, d.h. eine Rekursion kann man stets in eine Iteration umformen und umgekehrt [3, §6.1.4]. Als Beispiel betrachten wir unsere Berechnung der Fakultät, die wir oben als Rekursion kennen gelernt haben. Sie kann man äquivalent als `for`-Schleife schreiben:

```

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

```

```

public static int fakultaet(int n) {
    int f = 1;
    for(int i = n; i > 0; i--) {
        f *= i;
    }
    return f;
}

```

Was ist effizienter, Rekursion oder Iteration? Grundsätzlich ist die Anzahl der Rechenschritte von Rekursion und Schleife gleich. Die einzigen Unterschiede bezüglich der Ressourcen Zeit und Speicherplatz ergeben sich durch die Methodenaufrufe bei der Rekursion. Jeder einzelne Methodenaufruf kostet zusätzlichen technischen Verwaltungsaufwand („Overhead“), also zusätzliche Laufzeit; viel bedeutsamer aber ist, dass jeder Aufruf jeweils auch Speicherplatz im Stack der virtuellen Maschine für die lokalen Variablen der Methode benötigt. Pro Rekursionsaufruf steigt der Speicherbedarf um einen konstanten Betrag, oder anders ausgedrückt:

Merkregel 19. Der Speicherbedarf einer Rekursion steigt mit der Rekursionstiefe an, für eine Iteration dagegen ist er viel geringer, meist sogar konstant, also unabhängig von der Iterationsanzahl.

Das lineare Wachstum des Speicherbedarf bezüglich der Rekursionstiefe ist übrigens dafür verantwortlich, dass eine Endlosrekursion wegen Speicherüberlaufs abstürzt, während eine Endlosschleife mit dem immer gleichen Variablen ewig durchlaufen kann.

Trotz dieses Nachteils des höheren Speicherbedarfs haben Rekursionen Vorteile gegenüber Iterationen. Sie sind praktisch immer deutlich klarer und meistens zudem kürzer als ihre äquivalenten iterativen Formulierungen. Sie ermöglichen damit in der Regel leichter wartbaren und sichereren Quelltext, was gerade für komplexe Softwaresysteme von zentraler Bedeutung ist. Entsprechend ist der Entwurf und die Entwicklung von Algorithmen mit Rekursionen einfacher. Eine iterative Version der Türme von Hanoi zum Beispiel (Listing ?? auf Seite ??) muss sich um die Verwaltung der einzelnen Scheiben und die zyklische Verschiebung auf die Stapel kümmern, was die rekursive Variante frei Haus liefert. Ein weiterer immer wichtiger werdender Aspekt ist die „nebenläufige“ oder „verteilte“ Programmierung auf parallelen Rechnerarchitekturen, also Mehrkernprozessoren (*Multicore Processor*) oder Rechnerverbünden (*Cluster*): Rekursionen lassen sehr viel leichter parallelisieren als Iterationen, die grundsätzlich einen sequenziellen Ablauf voraussetzen.

Endrekursionen

Wenn nun jede Rekursion in der Theorie grundsätzlich in eine Iteration umgewandelt werden kann, und des Weiteren eine Iteration effizienter bezüglich der Ressourcen Laufzeit und Speicherplatz umgeht, stellt sich natürlich sofort die Frage: Kann ein Compiler nicht automatisch eine Rekursion in eine Iteration übersetzen? Die Antwort ist: nur für bestimmte Rekursionen, nämlich die sogenannten Endrekursionen.

Eine Rekursion heißt *endrekursiv* (englisch *tail recursive*) oder *endständig rekursiv*, wenn es nur einen Rekursionsaufruf gibt und dieser die letzte Aktion der Methode ist. Ein geeigneter Compiler kann in diesem Falle erkennen, dass die Methode keinen weiteren Speicherplatz benötigt, sondern mit den veränderten Parameterwerten gleich an ihren eigenen Anfang zurückspringen kann: Nichts anderes also als eine Iteration!

Obwohl grundsätzlich möglich, sind nicht alle Compiler auf Erkennung und Umwandlung von Endrekursionen optimiert. Insbesondere formt der Java-Compiler Endrekursionen nicht um. Untersuchen wir dazu etwas genauer, was der Java-Compiler aus einer Rekursion macht wie in dem folgenden Programm macht:

```

1 public class Rekursiv {
2     public static long fakultaet(long n) {
3         if (n == 0) { // Basisfall
4             return 1;
5         } else { // Rekursionsschritt:
6             return n * fakultaet(n - 1);

```



```

7     }
8   }
9 }

```

Betrachten wir mit dem Standardbefehl `javap -c` des Java SDK einmal die Bytecodes der compilierten Klasse. (`javap -c` disassembliert den Bytecode und zeigt den sogenannten Op-Code der einzelnen Befehle.)⁶

```

public static long fakultaet(long);
Code:
  0: lload_0
  1: lconst_0
  2: lcmp
  3: ifne      8
  6: lconst_1
  7: lreturn
  8: lload_0
  9: lload_0
 10: lconst_1
 11: lsub
 12: invokestatic #2          // Method fakultaet:(J)J
 15: lmul
 16: lreturn

```

Man muss nicht alle Anweisungen verstehen um zu erkennen, dass die Methode sich in Anweisung 12 selbst aufruft. Nach der Rückkehr führt sie die Multiplikation in Anweisung 15 aus. Die Java VM muss also nach der Rekursion das Ergebnis verarbeiten. Insbesondere ist die Methode nicht endrekursiv! Für die iterative Variante

```

1 public class Iterativ {
2     public static long fakultaet(long n) {
3         long f = 1;
4         for (long i = n; i > 0; i--) {
5             f *= i;
6         }
7         return f;
8     }
9 }

```

dagegen liefert `javap` die Ausgabe:

```

public static long fakultaet(long);
Code:
  0: lconst_1
  1: lstore_2
  2: lconst_1
  3: lstore     4
  5: lload      4
  7: lload_0
  8: lcmp
  9: ifgt      26

```

⁶ <http://docs.oracle.com/javase/8/docs/technotes/tools/windows/javap.html>

```

12: lload_2
13: lload      4
15: lmul
16: lstore_2
17: lload      4
19: lconst_1
20: ladd
21: lstore      4
23: goto      5
26: lload_2
27: lreturn

```

Hier beginnt die Schleife in Anweisung 5 und endet in Anweisung 23. Die Schleifenbedingung wird in Anweisung 9 geprüft und springt zum Schleifenabbruch zu Anweisung 26.

Wie lautet nun eine endrekursive Version der Fakultätsmethode? Es muss uns dazu gelingen, die Multiplikation als letzte Aktion in die Aufrufparameter der Methode zu verlegen, da das die einzige Möglichkeit ist, sie an die nachfolgende Rekursion zu übergeben. Wir müssen also einen neuen Aufrufparameter einführen, der die bisher ermittelten Zwischenwerte akkumuliert:

```

1 public class Endrekursiv {
2     public static long fakultaet(long n, long akku) {
3         if (n == 0) { // Basisfall
4             return akku;
5         } else {     // Rekursionsschritt:
6             return fakultaet(n - 1, n*akku);
7         }
8     }
9 }

```

Die letzte Aktion der Methode ist hier nun stets der Rekursionsaufruf, was auch die disassemblierte Version des resultierenden Bytecodes zeigt:

```

public static long fakultaet(long, long);
Code:
 0: lload_0
 1: lconst_0
 2: lcmp
 3: ifne      8
 6: lload_2
 7: lreturn
 8: lload_0
 9: lconst_1
10: lsub
11: lload_0
12: lload_2
13: lmul
14: invokestatic #2          // Method fakultaet:(JJ)J
17: lreturn

```

Wir erkennen, dass der Rekursionsaufruf hier in Anweisung 14 geschieht und die Methode in Anweisung 17 sofort danach verlassen wird. Allerdings hat der Compiler in diesem Falle

(javac version 1.8.0_60 auf Ubuntu Linux) die Endrekursion nicht erkannt und sie nicht in eine iterative Schleife umgewandelt.

Rufen wir zum Laufzeitvergleich die drei Methoden mit der externen Applikation `Main.java` auf,

```

1 public class Main {
2     public static void main(String... args) {
3         long n = 20; // <- maximal möglicher Wert
4         int max = 10000000;
5         long zeit;
6         long[] laufzeit = new long[3];
7
8         System.out.println(n + "! = " + Rekursiv.fakultaet(n));
9         System.out.println(n + "! = " + Iterativ.fakultaet(n));
10        System.out.println(n + "! = " + Endrekursiv.fakultaet(n,1));
11
12        for (int i = 1; i < max; i++) {
13            zeit = System.nanoTime();
14            Rekursiv.fakultaet(n);
15            laufzeit[0] += System.nanoTime() - zeit;
16
17            zeit = System.nanoTime();
18            Iterativ.fakultaet(n);
19            laufzeit[1] += System.nanoTime() - zeit;
20
21            zeit = System.nanoTime();
22            Endrekursiv.fakultaet(n,1);
23            laufzeit[2] += System.nanoTime() - zeit;
24        }
25
26        System.out.println("\nLaufzeiten");
27        System.out.println("rekursiv   : " + laufzeit[0] / 1000000. + " ms");
28        System.out.println("iterativ   : " + laufzeit[1] / 1000000. + " ms");
29        System.out.println("endrekursiv: " + laufzeit[2] / 1000000. + " ms");
30    }
31 }

```

so erhalten wir für die folgende Ausgabe:

| Laufzeiten | |
|-------------|-----------------|
| rekursiv | : 613.694976 ms |
| iterativ | : 447.399841 ms |
| endrekursiv | : 499.465378 ms |

Die Endrekursion ist also deutlich schneller als die nicht-endrekursive, offenbar hat also der JIT-Compiler etwas optimieren können.

3.5.4 Klassifikation von Rekursionen

Die Endrekursion ist nur ein Rekursionsschema von vielen. Sie gehört in die Klasse der *linearen Rekursionen*. Eine Rekursion heißt linear, wenn in jedem Rekursionsschritt genau ein Rekursionsaufruf geschieht. Der Aufrufablauf bildet also eine lineare Kette von Aufrufen. Ein

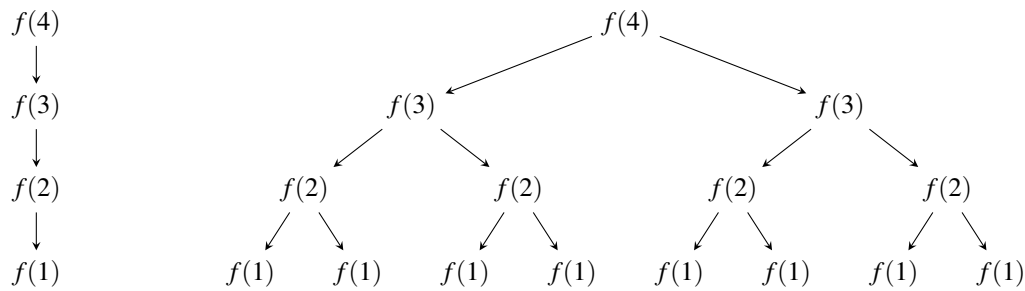


Abbildung 3.13. Lineare und verzweigende Rekursionen. Die lineare Rekursion links ist primitiv, die verzweigende Rekursion rechts hat zwei Rekursionsaufrufe je Rekursionsschritt, wie bei den Türmen von Hanoi.

Beispiel dafür ist unsere obige rekursive Fakultätsberechnung. Wichtige lineare Rekursionen sind speziell die Endrekursionen, die den Rekursionsaufruf als letzte Anweisung des Rekursionsschritts ausführen, und die sogenannten *primitiven Rekursionen* [3, S. 260], die speziell bei jedem Rekursionsschritt den für die Rekursionstiefe wesentlichen Parameter, z.B. n , um 1 senken,

$$f(n, \dots) \rightarrow f(n-1, \dots) \rightarrow \dots \rightarrow f(0, \dots).$$

also für alle n definiert sind [9, §9.3.1].

Lassen wir dagegen mehrere Selbstaufrufe pro Rekursionsebene zu, so sprechen wir von einer *verzweigenden* oder *mehrfachen Rekursion* (*tree recursion*), die einen verzweigten Aufrufbaum erzeugt. Von jedem Knoten gehen dabei genauso viele Äste ab wie Selbstaufrufe im Rekursionsschritt erscheinen. Ein Beispiel dafür ist die Lösung der Türme von Hanoi mit zwei Aufrufen je Rekursionsschritt. Ist mindestens eines der Argumente in einem Rekursionsaufruf selbst wieder ein Rekursionsaufruf, so liegt eine *verschachtelte Rekursion* vor, auch μ -Rekursion (sprich „mü-Rekursion“) genannt. Ein berühmtes Beispiel dafür ist die *Ackermann-funktion*:

```

1  public static long ack(long m, long n) {
2      if (m == 0) {
3          return n + 1;
4      } else if (n == 0) {
5          return ack(m-1, 1);
6      } else {
7          return ack(m-1, ack(m, n-1));
8      }
9  }

```

Der in Herscheid in Südwestfalen geborene Mathematiker Wilhelm Ackermann (1896–1962) entdeckte sie bereits 1927, also 14 Jahre vor Konrad Zuses Konstruktion der Z3, des ersten („turingvollständigen“) Computers. Für weitere Details zu den Rekursionstypen und ihren Beziehungen zu den Berechenbarkeitsklassen siehe [3, §6.1.4, 6.2].

Zusammengefasst ergibt sich daraus die in Abbildung 3.14 dargestellte Hierarchie der verschiedenen Rekursionstypen. In der Theorie spielen die primitiven Rekursionen und die μ -Rekursionen eine bedeutende Rolle. Viele verzweigend rekursive Algorithmen können grundsätzlich in eine primitive Rekursion und somit in eine Zählschleife umgeformt werden, allerdings dann in eine Schleife, die eine exponentiell höhere Schrittzahl benötigt. Eine iterative Lösung des Problems der Türme von Hanoi beispielsweise wäre die folgende Methode:

```

public static void turmIterativ(int n) {
    int zweiHochN = 1 << n; // ( 1 << n ) = 2^n

```


4

Objektorientierte Programmierung

Kapitelübersicht

| | | |
|-------|--|-----|
| 4.1 | Modellierung und Programmierung von Objekten | 94 |
| 4.1.1 | Die Grundidee der Objektorientierung | 94 |
| 4.1.2 | Klassendiagramme der UML | 97 |
| 4.1.3 | Programmierung von Objekten | 98 |
| 4.1.4 | Erzeugen von Objekten | 102 |
| 4.1.5 | Fallbeispiel DJTools, Version 1.0 | 104 |
| 4.1.6 | Version 2.0: Formatierte Zeitangaben | 108 |
| 4.2 | Mehrere Objekte und ihre Zustände: Fallbeispiel Konten | 112 |
| 4.3 | Vererbung | 116 |
| 4.3.1 | Überschreiben von Methoden | 117 |
| 4.3.2 | Die Referenz super | 118 |
| 4.3.3 | Vererbungsstruktur in Javadoc und die Klasse <code>Object</code> | 118 |
| 4.3.4 | Fallbeispiel: Musiktitel | 119 |
| 4.4 | Datenkapselung und Sichtbarkeit von Methoden | 122 |
| 4.4.1 | Pakete | 122 |
| 4.4.2 | Verbergen oder Veröffentlichen | 123 |
| 4.5 | Exceptions | 124 |
| 4.5.1 | Fangen einer Ausnahme | 125 |
| 4.5.2 | Werfen einer Ausnahme | 126 |
| 4.5.3 | Weiterreichen von Ausnahmen | 128 |
| 4.6 | Zusammenfassung | 130 |

4.1 Modellierung und Programmierung von Objekten

4.1.1 Die Grundidee der Objektorientierung

Was charakterisiert Objekte?

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

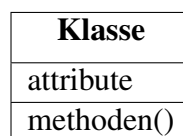
- Gegenstände, wie Autos, Schiffe und Flugzeuge,
- Lebewesen wie Pflanzen, Tiere oder Menschen,
- abstrakte Konstrukte wie Staaten, Absprachen, Konzerne oder Konten.

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 22. September 2019) 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 *fields* und *methods* üblich, so zum Beispiel in der Java API.



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

Fassen wir ganz formal zusammen:

Definition 4.1. 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 *Problemereich* der realen Welt. Ein Objekt ist eindeutig bestimmt durch seine *Attribute* (Daten, Eigenschaften) und durch seine *Methoden* (Funktionen, Verhalten). Attribute werden oft auch *Instanzvariablen* genannt. \square

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“.

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.

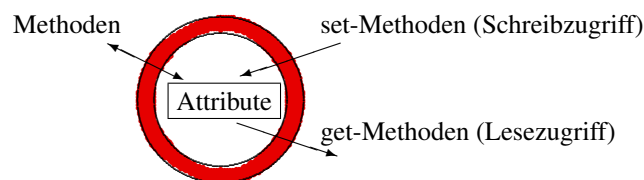


Abbildung 4.1. Objekte sind Einheiten aus Attributen und Methoden. Die Attribute sind gekapselt, sie können (i.a.) nur über die Methoden verändert werden.

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.

²Wenn 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*.

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 Rückgabetyt 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**.

4.1.2 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 „Wörter“ Diagramme sind, die wiederum aus wenigen Diagrammelementen („Buchstaben“) bestehen. Sie hat insgesamt neun Diagrammart. 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.

In einem Klassendiagramm wird eine 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 |
|--|
| - vorname : String - groesse : double |
| + Person(String, groesse) + getVorname() : String + getGroesse() : double + setGroesse(double) : void + toString() : String |

Oben ist also eine Klasse `Person` modelliert, die aus zwei Attributen `vorname` und `groesse` besteht und eine ganze Reihe eigener Methoden besteht. In einem Klassendiagramm werden die Datentypen von Attributen und die Rückgabetypen von Methoden nach einem Doppelpunkt notiert. Soll ein Attribut oder eine Methode „öffentlich“ sichtbar sein, also von einer beliebigen äußeren Klasse aus zugreifbar, so wird ihnen im Klassendiagramm ein Plus `+` vorangestellt. Ein vorangestelltes Minuszeichen dagegen bedeutet, dass das entsprechende Attribut bzw. die Methode nur innerhalb der Klasse sichtbar ist, also verborgen oder „gekapselt“ ist vor Zugriffen äußerer Klassen.

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. Außerdem ist dort der „Konstruktor“ aufgeführt, der genauso heißt wie die Klasse und ohne Doppelpunkt danach notiert. Die Bedeutung eines Konstruktors werden wir weiter unten kennenlernen.

4.1.3 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.

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  $p_1$ , ..., Datentyp  $p_n$ ) {
    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**) gesetzt.

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 Eingabevariablen genau so zu nennen wie die Attribute, denen sie Werte übergeben sollen.

Typische Implementierung einer Klasse für Objekte

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

```

1  /** Diese Klasse stellt eine Person dar.*/
2  public class Person {
3      // -- Attribute: -----
4      /** Der Vorname dieser Person.*/
5      private String vorname;
6      /** Die Größe dieser Person in m. Kann sich zur Laufzeit ändern.*/
7      private double groesse;
8      /** Der Konstruktor, erzeugt ein Objekt mit den Eingabedaten. */
9      public Person(String vorname, double groesse) {
10         this.vorname = vorname;
11         this.groesse = groesse;
12     }
13
14     // -- Methoden: -----
15     /** Gibt den Vornamen dieser Person zurück.*/
16     public String getVorname() {
17         return vorname;
18     }
19     /** Gibt die Größe dieser Person zurück.*/
20     public double getGroesse() {
21         return groesse;
22     }
23     /** Verändert die Größe dieser Person.*/
24     public void setGroesse(double groesse) {
25         this.groesse = groesse;
26     }
27     /** Gibt die Attributwerte dieser Person formatiert zurück.*/
28     public String toString() {
29         return vorname + ", " + groesse + " m";
30     }
31 }
```

Da die Klasse öffentlich ist, müssen wir sie in einer Datei speichern, die ihren Namen trägt, also `Person.java`. In der Regel wird eine Klasse, aus der Objekte erzeugt werden, als **public** deklariert (vgl. §4.4, S. 122).

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*).

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 *Objekt-methode*.

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

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 – oder „Instanzvariablen“, wie sie auch genannt werden – *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 4.1.

| Variablenart | Ort der Deklaration | Gültigkeitsbereich |
|---------------------------------------|---|--|
| lokale Variable | in Methoden (also auch Eingabeparameter!) | nur innerhalb der jeweiligen Methode |
| Attribute / Instanzvariablen | 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 4.1. Unterschiede lokaler Variablen und Attribute.

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 4.2. 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, einem dynamischen Speicher. Für Details siehe [JLS, JVM Specification §2.5].

Merkregel 22. 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.

4.1.4 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.

```

1  /** Erzeugt und verändert Objekte der Klasse Person.*/
2  public class PersonApp {
3      public static void main(String[] args) {
4          String ausgabe = "";
5          Person tom = new Person("Tom", 1.82);           // (1)
6          Person jerry;                                   // (2)
7          jerry = new Person("Jerry", 1.56);             // (3)
8          ausgabe += tom + "\n" + jerry;
9
10         jerry.setGroesse(1.46);
11
12         ausgabe += "\n" +
13             jerry.getVorname() + " ist jetzt " +
14             jerry.getGroesse() + " m groß.";
15         javax.swing.JOptionPane.showMessageDialog(null, ausgabe);
16     }
17 }

```

Die Ausgabe dieser Applikation ist in Abb. 4.3 dargestellt.

| |
|------------------------|
| Tom, 1.82 m |
| Jerry, 1.56 m |
| Jerry ist jetzt 1.46 m |

Abbildung 4.3. Die Ausgabe der Applikation `PersonApp`.

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:

```

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!

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 23. 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“, siehe Abbildung 4.4. Man kann statt des

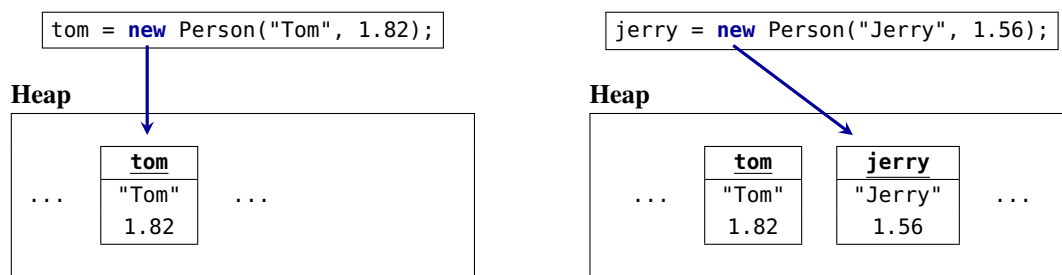


Abbildung 4.4. Speicherung von Objekten durch den **new**-Operator im Heap, einem Bereich des Arbeitsspeichers.

new-Operators übrigens jedes Objekt auch gleich dem „Nullobjekt“ setzen,

```
<Klasse> objekt = null;
```

Hier bezeichnet `<Klasse>` eine beliebige gegebene Klasse. Das reservierte Wort **null** verweist auf dieses leere Nullobjekt, das weder Attribute noch Methoden besitzt.

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 vorge-schrieben 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 weg gelassen, 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!)

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. Einer-seits 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', 'o'}`;

4.1.5 Fallbeispiel DJTools, Version 1.0

Als Fallbeispiel werden wir in diesem Abschnitt das Problem betrachten, eine Liste zu spielen-der 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 Pha-se, 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 könn-te 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 Sekun-denzeit in Minuten zurückgeben soll. Das ergibt also das folgende Klassendiagramm.

| Titel |
|--|
| - name: String - interpret: String - int sekundenzeit |
| + Titel(String, String, int) + toString(): String + getSekunden(): int + getMinuten(): double |

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 |
|-------------------------|------------------|---------------------|----------------|
| Joanne | Lady Gaga | 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 angeben. *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

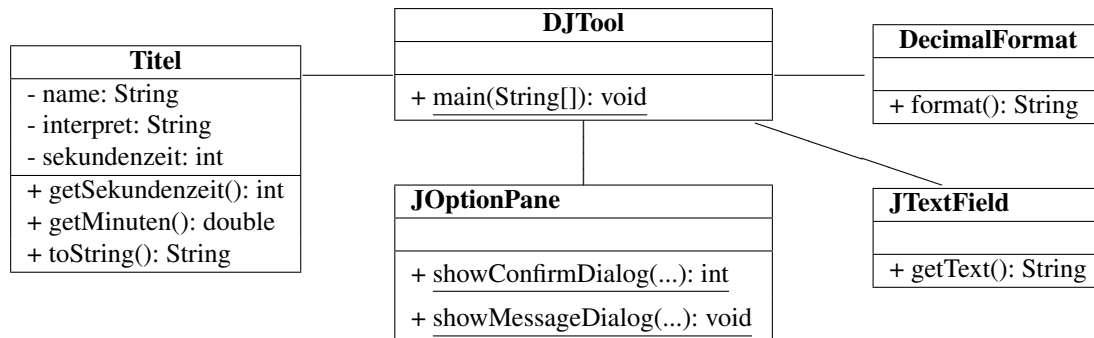
Die *Implementierung* ist diejenige Phase des Software Engineerings, in der die Ergebnisse des Entwurfs, also die Klassendiagramme und Algorithmen, in Programme umgesetzt werden. Insbesondere müssen nun technische Details wie die Festlegung der Datentypen festgelegt werden. Aus dem obigen Klassendiagramm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

```

1  /**
2   * Diese Klasse stellt einen Musiktitel dar.
3   *
4   * @author Andreas de Vries
5   * @version 1.0
6   */
7  public class Titel {
8      /** Name dieses Titels.*
9       private String name;
10     /** Name des Interpreten dieses Titels.*
11     private String interpret;
12     /** Dauer dieses Titels in Sekunden.*
13     private int sekundenzeit;
14
15     /** Erzeugt einen Musiktitel.*
16     public Titel(String name, String interpret, int sekundenzeit) {
17         this.name          = name;
18         this.interpret     = interpret;
19         this.sekundenzeit = sekundenzeit;
20     }
21
22     /** gibt die Dauer dieses Titels in Sekunden zurück.*
23     public int getSekundenzeit() {
24         return sekundenzeit;
25     }
26
27     /** Gibt die Dauer in Minuten zurück.*
28     public double getMinuten() {
29         return (double) sekundenzeit / 60.0;
30     }
31
32     /** Gibt die charakteristischen Attribute dieses Titels zurück.*
33     public String toString() {
34         return interpret + ": " + name + " (" + sekundenzeit + " sec)";
35     }
36 }

```

Das DJTool als Applikation. Die Applikation als System verschiedener Klassen 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 kennt und deren Methoden verwenden kann.

```

1  import java.text.DecimalFormat;
2  import javax.swing.JTextField;
3  import javax.swing.JOptionPane;
4
5  /**
6   * Programm zur Verwaltung von Musiktiteln.
7   * @author Andreas de Vries
8   * @version 1.0
9   */
10 public class DJTool {
11     public static void main ( String args[] ) {
12         // lokale Variablen:
13         int dauer = 0;
14         double spieldauer = 0.0;
15         String ausgabe = "";
16         DecimalFormat f2 = new DecimalFormat("#,##0.00"); // 2 Nachkommastellen
17
18         // Standardtitel vorgeben:
19         Titel titel1 = new Titel("Joanne", "Lady Gaga", 197);
20         Titel titel2 = new Titel("Catch", "Blank & Jones", 201);
21
22         // Eingabe eines weiteren Musiktitels:
23         JTextField[] feld = {
24             new JTextField("Talk"), new JTextField("Coldplay"), new JTextField("371")
25         };
26         Object[] msg = {
27             "Titel:", feld[0], "Interpret:", feld[1], "Dauer (in Sekunden):", feld[2]
28         };
29         int click = JOptionPane.showConfirmDialog(null, msg, "Eingabe", 2);
30
31         dauer = Integer.parseInt(feld[2].getText());
32
33         // Erzeugen des 3. Titels:
34         Titel titel3 = new Titel(feld[0].getText(), feld[1].getText(), dauer);
35
36         // berechne gesamte Spieldauer in Sekunden:
37         dauer = titel1.getSekundenzeit() + titel2.getSekundenzeit() + titel3.getSekundenzeit();
38         // ... in Minuten:

```

```

39     spieldauer = titel1.getMinuten() + titel2.getMinuten() + titel3.getMinuten();
40
41     ausgabe = titel1 + "\n" + titel2 + "\n" + titel3;
42     ausgabe += "\n\nGesamtspieldauer: " +
43         f2.format(spieldauer) + " min (" + dauer + " sec)";
44
45     JOptionPane.showMessageDialog(null, ausgabe, "Titelliste", -1);
46 }
47 }

```

4.1.6 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 `hh:mm:ss` oder `mm:ss` 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,

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

Entsprechend wird der Rest dieser Division umgerechnet in Minuten:

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

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 Zeit erweiterte Klassendiagramm in Abbildung 4.5. Die Beziehungslinien in diesem Diagramm besagen also, dass die Klasse DJTool

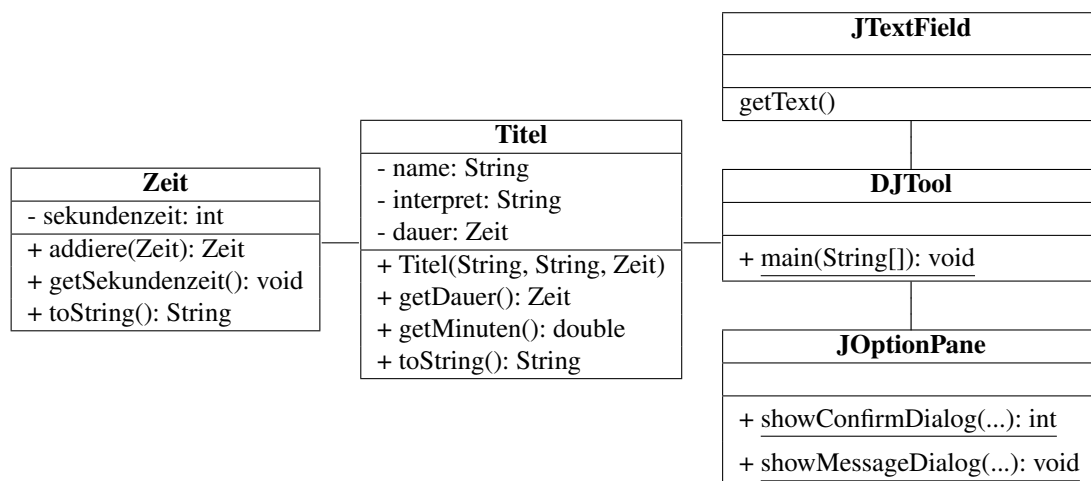


Abbildung 4.5. Klassendiagramm der DJTools (ohne die Klasse System, die stets bekannt ist)

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.

Implementierung

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

```

1 import javax.swing.JTextField;
2 import static javax.swing.JOptionPane.*;
3

```

```

4 /**
5  * Programm zur Verwaltung von Musiktiteln.
6  * @author Andreas de Vries
7  * @version 2.0
8  */
9 public class DJTool {
10     public static void main (String args[]) {
11         Titel[] titel = new Titel[3];
12
13         // Standardtitel vorgeben:
14         titel[0] = new Titel("Joanne", "Lady Gaga", new Zeit("3:17"));
15         titel[1] = new Titel("Catch", "Blank & Jones", new Zeit("3:21"));
16
17         // Eingabe eines weiteren Musiktitels:
18         JTextField[] feld = {
19             new JTextField("Talk"), new JTextField("Coldplay"), new JTextField("6:11")
20         };
21         Object[] msg = {
22             "Titel:", feld[0], "Interpret:", feld[1], "Dauer (mm:ss):", feld[2]
23         };
24         int click = showConfirmDialog(null, msg, "Eingabe", 2);
25
26         // Erzeugen des eingegebenen Titels:
27         titel[2] = new Titel(
28             feld[0].getText(), feld[1].getText(), new Zeit(feld[2].getText())
29         );
30
31         // berechne gesamte Spieldauer:
32         Zeit spieldauer = new Zeit("0:0");
33         for (Titel t : titel) {
34             spieldauer = spieldauer.addiere(t.getDauer());
35         }
36
37         // Ausgabe:
38         String ausgabe = "";
39         for (Titel t : titel) {
40             ausgabe += t + "\n";
41         }
42         ausgabe += "\n\nGesamtspieldauer: " + spieldauer;
43         showMessageDialog(null, ausgabe, "Titelliste", -1);
44     }
45 }

```

Die Klasse Titel muss entsprechend modifiziert werden:

```

1 /**
2  * Diese Klasse stellt einen Musiktitel dar.
3  *
4  * @author Andreas de Vries
5  * @version 2.0
6  */

```

```

7 public class Titel {
8     /** Name dieses Titels.**/
9     private String name;
10    /** Name des Interpreten dieses Titels.**/
11    private String interpret;
12    /** Dauer dieses Titels.**/
13    private Zeit dauer;
14
15    /** Erzeugt einen Musiktitel, dauer wird in dem Format (h:)m:s erwartet.**/
16    public Titel(String name, String interpret, Zeit dauer) {
17        this.name      = name;
18        this.interpret = interpret;
19        this.dauer     = dauer;
20    }
21
22    /** gibt die Dauer dieses Titels als Zeit zurück.**/
23    public Zeit getDauer() {
24        return dauer;
25    }
26
27    /** Gibt die Dauer in Minuten zurück.**/
28    public double getMinuten() {
29        return dauer.getSekundenzeit() / 60.0;
30    }
31
32    /** gibt die charakteristischen Attribute dieses Titels zurück.**/
33    public String toString() {
34        return interpret + ": " + name + " (" + dauer + ")";
35    }
36 }

```

Und schließlich benötigen wir die Klasse Zeit.

```

1 /** Diese Klasse erzeugt Objekte, die eine zeitliche Dauer
2 * darstellen. Sie ist ein Datentyp, der eine Addition von
3 * Zeiten ermöglicht.
4 */
5 public class Zeit {
6     private int sekundenzeit;
7
8     /** Erzeugt eine Dauer der Länge sekunden.**/
9     public Zeit(int sekunden) {
10        this.sekundenzeit = sekunden;
11    }
12
13    /** Erzeugt ein Objekt aus einem String im Format "h:m:s" oder "m:s".**/
14    public Zeit(String dauer) {
15        String[] zeiten = dauer.split(":"); // Trennungszeichen ":"
16        int stunden, minuten, sekunden;
17        if (zeiten.length == 2) {
18            stunden = 0;

```

```

19         minuten = Integer.parseInt(zeiten[0]);
20         sekunden = Integer.parseInt(zeiten[1]);
21     } else {
22         stunden = Integer.parseInt(zeiten[0]);
23         minuten = Integer.parseInt(zeiten[1]);
24         sekunden = Integer.parseInt(zeiten[2]);
25     }
26     this.sekundenzeit = stunden * 3600 + minuten * 60 + sekunden;
27 }
28
29 /** Addiert die eingegebene dauer zu dieser Zeit und gibt das Ergebnis zurück.*/
30 public Zeit addiere(Zeit dauer) {
31     return new Zeit(this.sekundenzeit + dauer.sekundenzeit);
32 }
33
34 public int getSekundenzeit() {
35     return sekundenzeit;
36 }
37
38 /** Gibt die Zeit in dem Format h:m:s zurück.*/
39 public String toString() {
40     String ausgabe = "";
41     int sekunden = sekundenzeit;
42     int stunden = sekunden / 3600;
43     sekunden %= 3600;
44     int minuten = sekunden / 60;
45     sekunden %= 60;
46
47     if (stunden > 0) {
48         ausgabe += stunden + ":";
49     }
50     ausgabe += minuten + ":" + sekunden;
51     return ausgabe;
52 }
53 }

```

Zu beachten ist, dass die Klasse `Zeit` zwei Konstruktoren besitzt. Der eine erwartet die Sekundenanzahl, 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 bei dem zweiten Konstruktor die Methode `split` verwendet, die hier den eingegebenen String an den Zeichen ":" aufteilt und in als Einträge eines String-Arrays speichert.

4.2 Mehrere Objekte und ihre Zustände: Fallbeispiel Konten

Dieses Kapitel befasst sich mit einer wesentlichen Möglichkeit, die objektorientierte Programmierung bietet: die Erstellung mehrerer Objekte derselben Klasse und deren gekapselter „Datenhaushalte“ und separaten Lebenszyklen. Die Erzeugung individueller Objekte mit ihren je-

weils eigenen „geschützten“ (*protected*) oder „privaten“ Eigenschaften führt dazu, dass jedes Objekt eine Art Eigenleben hat. Die zu einem bestimmten Zeitpunkt gegebene Konstellation an Attributwerten eines Objekts wird oft als dessen *Zustand* bezeichnet. Wir werden diese Begriffe am Beispiel von Bankkonten erläutern.

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 Diagramm in Abbildung 4.6, ein so genanntes Sequenzdiagramm. Es stellt den zeitlichen Ablauf einer Überweisung dar. Zunächst wird eine Überweisung ausgeführt, in dem die Methode *ueberweisen*

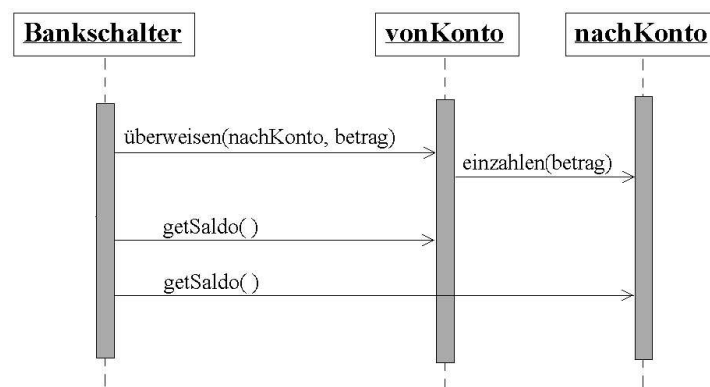


Abbildung 4.6. Sequenzdiagramm. Es zeigt die Methodenaufrufe („Nachrichten“), die die verschiedenen beteiligten Objekte und die Applikation *Bankschalter* ausführen.

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. 4.7

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 zurückge-

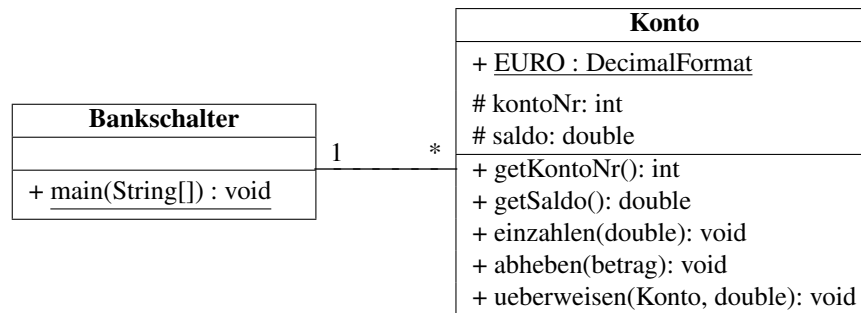


Abbildung 4.7. Klassendiagramm des Kontosystems gemäß der UML

ben, 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`:

```

1 import java.text.DecimalFormat;
2 /** Die Klasse implementiert Konten. */
3 public class Konto {
4     public final static DecimalFormat EURO = new DecimalFormat("#,##0.00 EUR");
5     // Attribute: -----
6     protected int kontoNr;
7     protected double saldo;
8     // Konstruktor: -----
9     public Konto(int kontoNr) {
10         this.kontoNr = kontoNr;
11         saldo = 100; // Startguthaben 100 Euro
12     }
13     // Methoden: -----
14     public int getKontoNr() {
15         return kontoNr;
16     }
17     public double getSaldo() {
18         return saldo;
19     }
20     public void einzahlen(double betrag) {
21         saldo += betrag;
22     }
  
```

```

23     public void abheben(double betrag) {
24         saldo -= betrag;
25     }
26     public void ueberweisen(Konto nachKonto, double betrag) {
27         saldo -= betrag;
28         nachKonto.einzahlen(betrag);
29     }
30     public String toString() {
31         return kontoNr + ": " + EURO.format(saldo);
32     }
33 }

```

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:

```

1  import javax.swing.*;
2  /** Erzeugt die Konten und ermöglicht Überweisungen. */
3  public class Bankschalter {
4      private final static int anzahlKonten = 5;
5      private static String anzeigen(Konto[] konten) {
6          String anzeige = "Kontostände:";
7          for (Konto k : konten) {
8              anzeige += "\n" + k;
9          }
10         return anzeige;
11     }
12
13     public static void main( String[] args ) {
14         // erzeuge Array von Konten mit Kontonummern 0, 1, ..., 4:
15         Konto konto[] = new Konto[anzahlKonten];
16         for ( int i = 0; i < konto.length; i++ ) {
17             konto[i] = new Konto(i);
18         }
19         Konto vonKonto, nachKonto;
20         double betrag;
21         String anzeige = "";
22
23         // Eingabeblock:
24         JTextField[] feld = {new JTextField(), new JTextField(), new JTextField("0")};
25         Object[] msg = {
26             anzeigen(konto), "\nÜberweisung von Konto", feld[0], "nach Konto", feld[1],
27             "Betrag [EUR]:", feld[2]
28         };
29         int click = JOptionPane.showConfirmDialog(null, msg, "Eingabe", 2);
30         while (click == 0) { // 'OK' geklickt?
31             vonKonto = konto[ Integer.parseInt(feld[0].getText()) ];
32             nachKonto = konto[ Integer.parseInt(feld[1].getText()) ];
33             betrag = Double.parseDouble(feld[2].getText());
34             // Überweisung durchführen:
35             vonKonto.ueberweisen( nachKonto, betrag );
36             // Eingabefenster mit aktualisierten Daten:

```

```

37     msg[0] = anzeigen(konto);
38     feld[0].setText(""); feld[1].setText(""); feld[2].setText("0");
39     click = JOptionPane.showConfirmDialog(null, msg, "Eingabe", 2);
40 }
41 }
42 }

```

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*.

4.3 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 allgemeinen 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*.

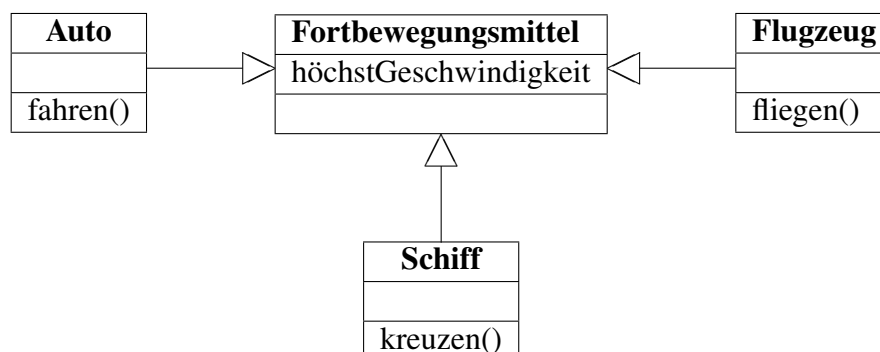
Umgekehrt ist die Vererbung eine Möglichkeit, bereits vorhandene Klassen zu erweitern. Man kann dann die Attribute und Methoden der allgemeinen Klasse übernehmen und neue Merkmale hinzufügen. Beispielsweise ist eine Vorzugsaktie eine Aktie mit dem Merkmal „kein Stimmrecht“.

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.

```

1 public class Subklasse extends Superklasse {
2     ...;                // <-- ggf. neue Attribute
3
4     /** Konstruktor der Subklasse. */
5     public Subklasse (...) {
6         super(...);      // <-- ein Konstruktor der Superklasse
7         ...;             // <-- ggf. Initialisierung der neuen Attribute
8     }
9
10    ...                 // <-- ggf. neue oder überschreibende Methoden
11 }
```

Mit der Referenz **super** wird hier optional ein Konstruktor der Superklasse aufgerufen. Zu beachten ist dabei, dass dieser Aufruf die allererste Anweisung des Subklassen-Konstruktors sein muss.

4.3.1 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.

Das Überladen und das Überschreiben sind spezielle Ausprägungen des in der Programmierung allgemein *Polymorphismus* genannten Konzepts, nach dem ein Methodenname verschiedene Implementierungen innerhalb eines Softwaresystems darstellen kann.

4.3.2 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

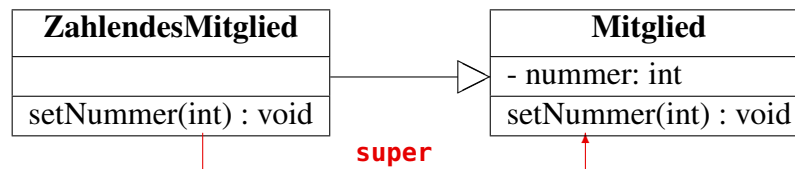


Abbildung 4.8. Wirkung der Referenz `super.setNummer(n)`

ne Methode `setNummer` aus der Superklasse `Mitglied`, indem die Nummer nur gesetzt wird, wenn sie größer als 3000 ist:

```

public void setNummer(int nummer) {
    if (nummer > 3000) {
        super.setNummer(nummer);
    }
}
  
```

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); // nichts passiert
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.

4.3.3 Vererbungsstruktur in Javadoc und die Klasse `Object`

Die Vererbungshierarchie einer Klasse wird auch in der Javadoc-Dokumentation dargestellt. Die Klassen, von der eine gegebene Klasse abgeleitet ist, erkennt man stets im oberen Bereich in der durch javadoc erzeugten API-Dokumentation. Dort erkennt man auch sofort folgendes.

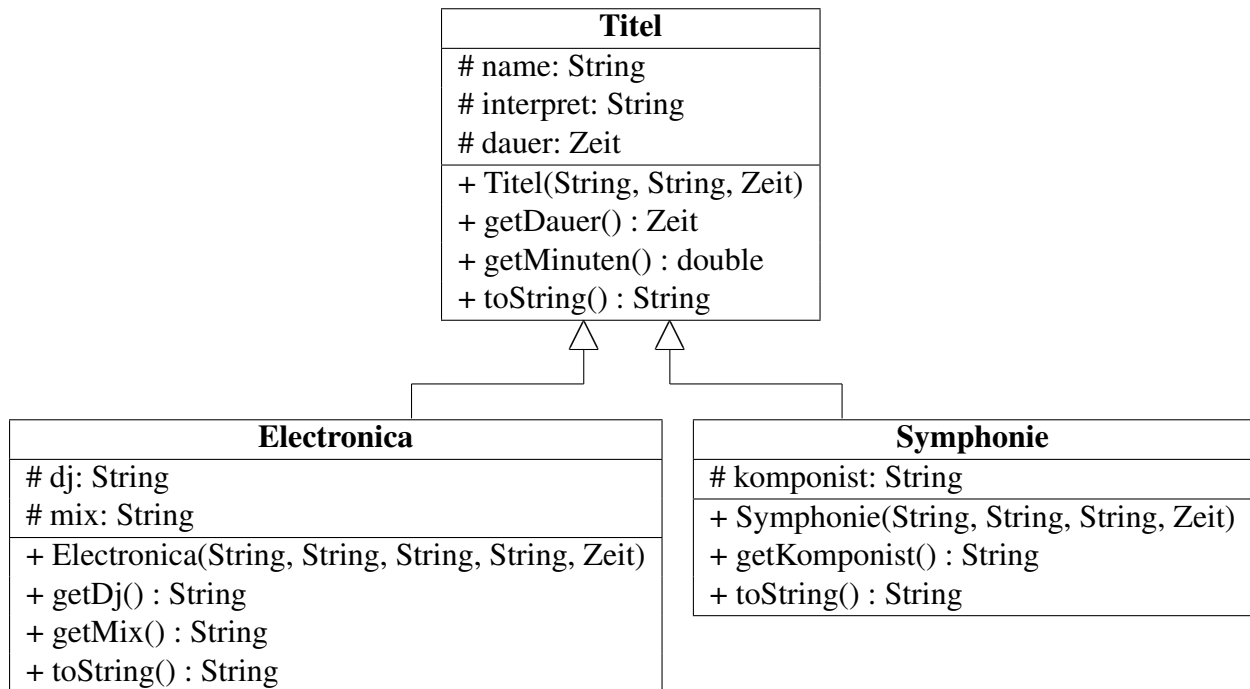
Jede Klasse in Java ist, direkt oder indirekt, abgeleitet von einer Superklasse, der Klasse `Object`. Das ist automatisch so, auch ohne dass man explizit `extends Object` zu schreiben braucht. Man sagt auch, die Klasse `Object` ist die Wurzel (*root*) der allgemeinen Klassenhierarchie von Java.

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.

4.3.4 Fallbeispiel: Musiktitel

Ein Beispiel für eine Vererbung sind die beiden Ableitungen Symphonie und Electronica der Klasse Titel der DJTool-Applikation:



Wir haben hier die Sichtbarkeit der Attribute eines Musiktitels erweitert, sie sind nun **protected** („geschützt“), nicht mehr privat. Der Grund ist, dass damit eine erbende oder „abgeleitete“ Klasse direkt auf sie zugreifen kann. Alternativ kann man natürlich auch öffentliche Get-Methoden implementieren, die dann in den erbenden Klassen verwendet werden können.

In Java implementiert lauten die neuen erbenden Klassen wie folgt:

```

1  /**
2   * Diese Klasse stellt einen Electronica-Titel dar.
3   */
4  public class Electronica extends Titel {
5      /** DJ dieses Titels.*/
6      protected String dj;
7      /** Mix, aus dem dieser Titel stammt.*/
8      protected String mix;
9
10     /** Erzeugt einen Electronica-Titel.*/
11     public Electronica(
12         String name,String interpret,String dj,String mix,Zeit dauer
13     ) {
14         super(name, interpret, dauer); // ruft den Konstruktor der Superklasse auf
15         this.dj = dj;
16         this.mix = mix;
17     }
18

```

```

19  /** Gibt den DJ dieses Titels zurück.**/
20  public String getDj() {
21      return dj;
22  }
23
24  /** Gibt den Mix dieses Titels zurück.**/
25  public String getMix() {
26      return mix;
27  }
28
29  /** Gibt die charakteristischen Attribute dieses Titels zurück.**/
30  @Override
31  public String toString() {
32      return dj+" feat. "+interpret+": "+name+", "+mix+" (" +dauer + ")";
33  }
34 }

```

und

```

1  /**
2   * Diese Klasse stellt eine Symphonie als speziellen Musiktitel dar.
3   */
4  public class Symphonie extends Titel {
5      /** Komponist dieser Symphonie.**/
6      protected String komponist;
7
8      /** Erzeugt eine Symphonie.**/
9      public Symphonie(String komponist, String name, String interpret, Zeit dauer) {
10         super(name, interpret, dauer); // ruft den Konstruktor der Superklasse auf
11         this.komponist = komponist;
12     }
13
14     /** Gibt den Komponisten dieser Symphonie zurück.**/
15     public String getKomponist() {
16         return komponist;
17     }
18
19     /** Gibt die charakteristischen Attribute dieser Symphonie zurück.**/
20     @Override
21     public String toString() {
22         return komponist + ": " + name + ", " + interpret + " (" + dauer + ")";
23     }
24 }

```

Die Klasse Titel lautet mit den Änderungen:

```

1  /**
2   * Diese Klasse stellt einen Musiktitel dar.
3   *
4   * @author Andreas de Vries
5   * @version 3.0
6   */

```



```

7 public class Titel {
8     /** Name dieses Titels.**/
9     protected String name;
10    /** Name des Interpreten dieses Titels.**/
11    protected String interpret;
12    /** Dauer dieses Titels.**/
13    protected Zeit dauer;
14
15    /** Erzeugt einen Musiktitel, dauer wird in dem Format (h:)m:s erwartet.**/
16    public Titel(String name, String interpret, Zeit dauer) {
17        this.name      = name;
18        this.interpret = interpret;
19        this.dauer      = dauer;
20    }
21
22    /** Gibt die Dauer dieses Titels als Zeit zurück.**/
23    public Zeit getDauer() {
24        return dauer;
25    }
26
27    /** Gibt die Dauer in Minuten zurück.**/
28    public double getMinuten() {
29        return dauer.getSekundenzeit() / 60.0;
30    }
31
32    /** Gibt die charakteristischen Attribute dieses Titels zurück.**/
33    public String toString() {
34        return interpret + ": " + name + " (" + dauer + ")";
35    }
36 }

```

Die Titelliste in der Klasse DJTool wird um eine Symphonie und einen Electronica-Titel

```

1 import javax.swing.JTextField;
2 import static javax.swing.JOptionPane.*;
3
4 /**
5  * Programm zur Verwaltung von Musiktiteln.
6  * @author  Andreas de Vries
7  * @version 3.0
8  */
9 public class DJTool {
10    public static void main ( String args[] ) {
11        Titel[] titel = new Titel[5];
12
13        // Standardtitel vorgeben:
14        titel[0] = new Titel("Joanne", "Lady Gaga", new Zeit("3:17"));
15        titel[1] = new Titel("Catch", "Blank & Jones", new Zeit("3:21"));
16        titel[2] = new Symphonie(
17            "Mozart", "Die Prager", "Leonard Bernstein", new Zeit("27:49")
18        );

```

```

19     titel[3] = new Electronica(
20         "You got the love", "Candi Staton", "The Source", "New Voyager Mix", new Zeit("3
21     );
22
23     // Eingabe eines weiteren Musiktitels:
24     JTextField[] feld = {
25         new JTextField("Talk"), new JTextField("Coldplay"), new JTextField("6:11")
26     };
27     Object[] msg = {
28         "Titel:", feld[0], "Interpret:", feld[1], "Dauer (mm:ss):", feld[2]
29     };
30     int click = showConfirmDialog(null, msg, "Eingabe", 2);
31
32     // erzeuge den eingegebenen Titel:
33     titel[4] = new Titel(
34         feld[0].getText(), feld[1].getText(), new Zeit(feld[2].getText())
35     );
36
37     // berechne gesamte Spieldauer:
38     Zeit spieldauer = new Zeit("0:0");
39     for (Titel t : titel) {
40         spieldauer = spieldauer.addiere(t.getDauer());
41     }
42
43     // Ausgabe:
44     String ausgabe = "";
45     for (Titel t : titel) {
46         ausgabe += t + "\n";
47     }
48     ausgabe += "\n\nGesamtspieldauer: " + spieldauer;
49     showMessageDialog(null, ausgabe, "Titelliste", -1);
50 }
51 }

```

4.4 Datenkapselung und Sichtbarkeit von Methoden

4.4.1 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,...).

4.4.2 Verbergen oder Veröffentlichen

Manchmal ist es sinnvoll, Attribute oder Methoden von 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*) oder *Sichtbarkeiten*:

- **public**: Zugriff uneingeschränkt möglich;
- **protected**: alle Unterklassen („erbende“ Klassen) und Klassen desselben Pakets haben Zugriff;
- **private**: Zugriff nur innerhalb der Klasse.

Diese Zugriffsmodifikatoren bestimmen also die Sichtbarkeit des Elements, das darauffolgend 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 Zu-

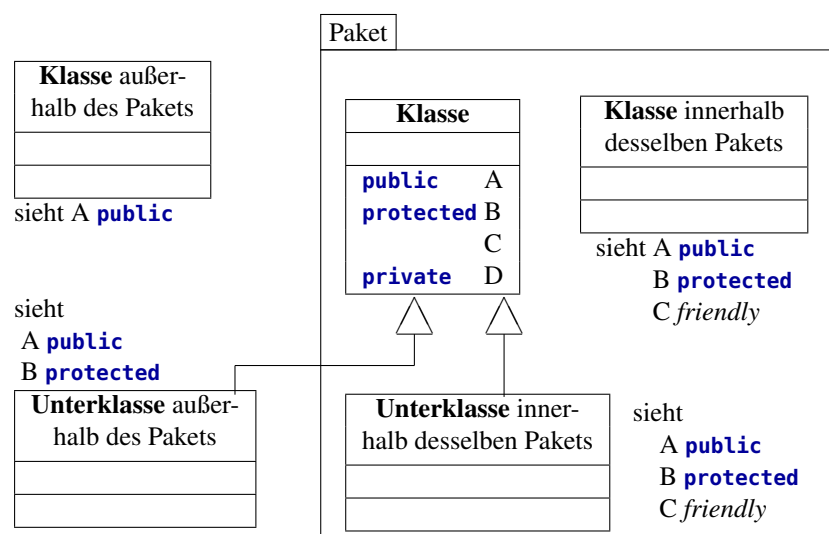


Abbildung 4.9. Die Sichtbarkeiten von Methoden in Java

griffsmodifikatoren erlaubt für Klassen, Schnittstellen, Methoden, Konstruktoren und Attribute. Um die Philosophie der Datenkapselung konsequent umzusetzen und den Zugriff nur über Objektmethoden zu ermöglichen, sollten grundsätzlich Attribute stets **private** oder **protected**

deklariert werden. Ein **public**-Element wird in UML mit + markiert, ein **protected**-Element mit #, und ein **private**-Element -. In Tabelle 4.2 finden Sie die Notationen in einem Klassendiagramm gemäß UML und die Entsprechungen in Java.

| Bedeutung | UML | Java |
|---------------------|---------------------------------------|---|
| öffentlich sichtbar | + | public |
| im Paket sichtbar | ~ | — |
| geschützt sichtbar | # | protected |
| privat sichtbar | - | private |
| Deklaration | <i>attribut : datentyp</i> | <i>datentyp attribut;</i> |
| Objektmethode | <i>m(datentyp, ...) : rückgabotyp</i> | <i>rückgabotyp m(datentyp x, ...)</i> |
| Konstruktor | <i>Konstruktor(datentyp, ...)</i> | <i>Konstruktor(datentyp x, ...)</i> |
| Statische Methode | + <u>main(String[]) : void</u> | public static void main(String[] args) |

Tabelle 4.2. Notation in einem Klassendiagramm in UML und die entsprechende Umsetzung in Java.

4.5 Exceptions

In diesem Abschnitt wenden wir uns dem wichtigen Bereich der *Ausnahmebehandlung* (*exception handling*) zu. Eine *Ausnahme* oder *Exception* zeigt an, dass ein Problem während der Ausführung eines Programms aufgetaucht ist, d.h. ein Laufzeitfehler. Das sind Fehler eines Programms, dessen Syntax formal korrekt ist, die der Compiler also nicht finden kann. Typische Laufzeitfehler sind Division durch 0, Bereichsüberschreitungen von Array-Indizes, Zugriff auf nicht erzeugte Objekte oder fehlerhafte Dateizugriffe. Solche Fehler bringen ein Programm zum Absturz, wenn man sie nicht abfängt. In älteren Programmiersprachen, die dieses Konzept der Ausnahmebehandlung noch nicht kannten, kann es in diesen Fällen zu schweren Speicherbereichsfehlern kommen, die den gesamten Rechner lahmlegen können und damit ein großes Sicherheitsproblem darstellen. Die Ausnahmebehandlung von Java entspricht weitgehend der Ausnahmebehandlung von C++, wie sie A. Koenig und Bjarne Stroustrup bereits 1990 veröffentlichten. Sie wird in Situationen angewandt, in denen das System die die Ausnahme verursachende Fehlfunktion erkennen kann. Diese Erkennungsmethode ist der so genannte *Exception Handler*. Wird eine Ausnahme erkannt, so wird sie „ausgelöst“ oder „geworfen“ (*throw an exception*). Eine geworfene Ausnahme wird von dem Ausnahmebehandler (*exception handler*) „gefangen“ und behandelt (*caught and handled*). Für weitere Informationen sei auf das Tutorial

<https://docs.oracle.com/javase/tutorial/essential/exceptions/>

verwiesen.

Betrachten wir zur Einführung ein einfaches Beispielprogramm, das eine *NullPointerException* auslöst, also einen Zugriff auf ein nicht vorhandenes Objekt:

```

1 package paket;
2
3 public class Exception_1 {
4     public static void main(String... args) {
5         Object objekt = null;
6         System.out.println(objekt.toString());
7     }
8 }
```

Der Compiler erkennt hier kein Problem, der Bytecode wird ohne Weiteres erstellt. Lässt man dieses Programm jedoch ablaufen, so wird es mit der folgenden Fehlermeldung abgebrochen:

```
Exception in thread "main" java.lang.NullPointerException
    at paket.Exception_1.main(Exception_1.java:6)
```

Eine solche Meldung sagt uns bereits eine ganze Menge über den aufgetretenen Laufzeitfehler: In der ersten Zeile steht der Fehlertyp, hier eine `NullPointerException`. Dies ist in Wirklichkeit eine Klasse in Java und in der API unter

<http://docs.oracle.com/javase/8/docs/api/java/lang/NullPointerException.html>

dokumentiert und tritt immer dann auf, wenn auf ein nicht erzeugtes Objekt zugegriffen wird. Die Angaben hinter `at` zeigen die *Aufruferriste* oder den *CallStack*, also welche Methoden in welchen Klassen aufgerufen waren, als der Fehler auftrat. Hier ist es nur eine aufgerufene Methode, die `main`-Methode in der Klasse `Exception_1` im Paket `paket`, und dort in Zeile 6.

In der API-Dokumentation kann man direkt erkennen, dass `Exception` die Superklasse aller Ausnahmen ist. Eine wichtige Subklasse ist `RuntimeException`, die nicht explizit gefangen oder weitergereicht werden muss, und die wiederum als Subklassen `ArrayIndexOutOfBoundsException` und `IllegalArgumentException` hat, siehe Abbildung 4.10. Letztere zeigt üblicherweise

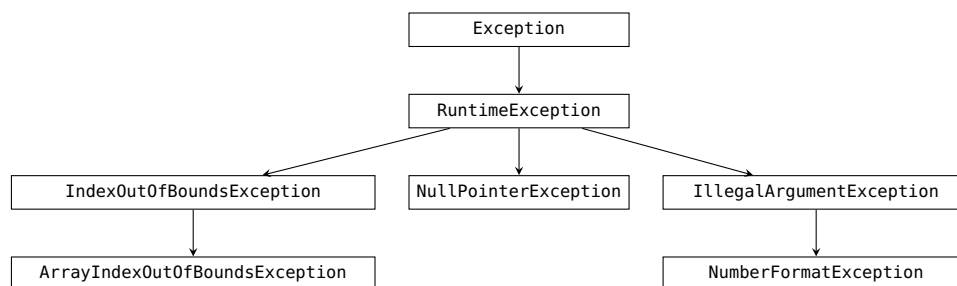


Abbildung 4.10. Wichtige Exceptions in der Exceptionhierarchie von Java.

an, dass eine Methode einen unerlaubten Parameterwert erhalten hat. Eine Subklasse ist `NumberFormatException`, die wir von den Parsefunktionen der Zahlformate kennen und ausgelöst wird, wenn ein String nicht zu einer Zahl umgeformt werden kann.

4.5.1 Fangen einer Ausnahme

Der Programmierer umschließt den Programmteil, der eine Ausnahme erzeugen könnte, mit einem **try**-Block. Ihm können direkt ein oder mehrere **catch**-Blöcke folgen. Jeder dieser **catch**-Blöcke spezifiziert den zu bearbeitenden Ausnahmetyp und enthält Anweisungen, die im Fehlerfall auszuführen sind (den „Exception Handler“).

```

try {
    ... // mache etwas, was schiefgehen könnte ...
}
catch (Ausnahmetyp e) {
    ... // rette mich, wenn es mit Fehler e schiefgeht ...
}
catch (Ausnahmetyp e2) {
    ... // rette mich, wenn es mit dem Fehler e2 schiefgeht
    ...
}
finally {
    ... // und räume am Ende auf (optional).
}
```

Wird der **try**-Block ausgeführt, ohne dass eine Ausnahme ausgeworfen wurde, so wird der **catch**-Block (bzw. die **catch**-Blöcke) übersprungen und die erste Anweisung danach ausgeführt. Optional kann der **finally**-Block definiert werden, in dem alle Anweisungen stehen, die nach dem Verlassen des **try**-Blocks auf jeden Fall ausgeführt werden sollen (z.B. geöffnete Dateien schließen). Falls insbesondere der **try**-Block durch eine der Anweisungen **return**, **continue** oder **break** verlassen wird, werden die Anweisungen des **finally**-Blocks ausgeführt, ehe mit der nächstfolgenden Anweisung (bzw. der aufrufenden Methode) fortgefahren wird.

```

1 package paket;
2 import javax.swing.JOptionPane;
3
4 public class Exception_2 {
5     public static void main(String... args) {
6         String eingabe = "", ausgabe;
7         int zahl;
8
9         try {
10             eingabe = JOptionPane.showInputDialog(null, "Gib eine ganze Zahl ein!");
11             zahl = Integer.parseInt(eingabe);
12             JOptionPane.showMessageDialog(null, "Du hast " + zahl + " eingegeben");
13         } catch (NumberFormatException nfe) {
14             JOptionPane.showMessageDialog(null, "Bitte demnächst eine ganze Zahl eingeben!",
15                 "Fehlermeldung", JOptionPane.WARNING_MESSAGE);
16         } finally {
17             JOptionPane.showMessageDialog(null, "Tschüss!");
18         }
19     }
20 }

```

Sobald eine Ausnahme ausgeworfen wurde, wird der **try**-Block verlassen und der entsprechende **catch**-Block ausgeführt. Insbesondere kann eine Ausnahme keine Objekte des **try**-Blocks verwenden, da dieser nicht mehr aktiv ist. Es kann einen oder auch mehrere **catch**-Blöcke geben, spezifiziert für jeweils eine Ausnahme. Die Definition eines **catch**-Blocks sieht ähnlich aus wie die einer Methode, es wird ein Parameter der Klasse `Exception` oder einer Subklasse davon³ deklariert; allerdings ist der Gültigkeitsbereich der gesamte umschließende Block, darf also weder vorher noch nachher nochmal deklariert werden.

4.5.2 Werfen einer Ausnahme

Soll eine Ausnahme geworfen werden, so geschieht dies mit dem reservierten Wort **throw** und dem **new**-Operator:

```
throw new Ausnahme("Fehlermeldung");
```

Betrachten wir dazu das folgende Programm, das in einer Endlosschleife per Eingabedialogfenster ganze Zahlen zwischen 0 und 7 einliest und den entsprechenden Wochentag ausgibt. Das Programm wirft dabei eine selbsterstellte Ausnahme, wenn es sich um eine zwar um eine ganze Zahl handelt, sie aber nicht aus dem erlaubten Bereich ist. Zudem wird die Schleife abgebrochen, wenn die Taste „Abbrechen“ geklickt wird.

³ Eigentlich kann nach der **catch**-Anweisung allgemein ein Objekt der Klasse `Throwable` oder einer Subklasse davon deklariert werden, das sind neben `Exceptions` auch `Errors`; da aber `Throwables`, die keine `Exceptions` sind, schwere Fehler darstellen, sollten ausschließlich `Exceptions` gefangen werden.

```

1 package paket;
2
3 import static javax.swing.JOptionPane.*;
4
5 class KeinWochentagAusnahme extends NumberFormatException {
6     public KeinWochentagAusnahme(String message) {
7         super(message);
8     }
9 }
10
11 class AbbruchAusnahme extends Exception{}
12
13 public class Exception_3 {
14     public static void main(String... args) {
15         String eingabe = "", ausgabe;
16         int zahl;
17         // Deklariert, erzeugt und initialisiert ein String[7]-Array:
18         String wochentag[] = {"Sonntag", "Montag", "Dienstag", "Mittwoch",
19                               "Donnerstag", "Freitag", "Samstag"};
20
21         while (true) { // Endlosschleife
22             try {
23                 eingabe = showInputDialog(null, "Gib eine Zahl von 0 bis 7 ein!");
24                 if (eingabe == null) { // Anwender klickt Äbbrechen"
25                     throw new AbbruchAusnahme();
26                 }
27                 zahl = Integer.parseInt(eingabe);
28                 if (zahl < 0 || zahl > 7) { // 7 -> 0, s.u.
29                     throw new KeinWochentagAusnahme("Zahl kein Wochentag!");
30                 }
31                 ausgabe = "Tag Nr. " + zahl + " ist " + wochentag[zahl % 7];
32                 showMessageDialog(null, ausgabe);
33             } catch (KeinWochentagAusnahme kwa) {
34                 showMessageDialog(null, "Zahl bitte von 0 bis 7!", "Wochentage", WARNING_MESSA
35             } catch (NumberFormatException nfe) {
36                 showMessageDialog(null, "Bitte eine ganze Zahl!", "Wochentage", WARNING_MESSA
37             } catch (AbbruchAusnahme aa) {
38                 showMessageDialog(null, "Tschüss!");
39                 break; // bricht die Endlosschleife ab
40             }
41         }
42     }
43 }

```

In den Zeilen 25 und 29 wird die **throw**-Anweisung ausgeführt, um jeweils eine Ausnahme auszulösen. Die **throw**-Anweisung spezifiziert ein Objekt, das geworfen werden soll. Dieser Operand kann ein beliebiges Objekt einer von Throwable abgeleiteten Klasse sein. Die beiden direkten Unterklassen von Throwable sind Exception und Error. Errors sind schwerwiegende Systemfehler, die man normalerweise nicht fangen sollte. Ausnahmen oder Exceptions dagegen sind Laufzeitfehler, die der Programmierer meist abfangen möchte.

In unserem Beispiel wird eine generelle Ausnahme abgefangen, indem der Anwender einfach erneut zu einer Eingabe aufgefordert wird. Abgefangen werden hier nur Eingaben, die nicht Zahlen sind. Andererseits wird in Zeile 25 eine Ausnahme geworfen, wenn die Eingabe `null` ist, der Anwender im Dialogfenster also „Abbrechen“ geklickt hat; diese Ausnahme führt zur Meldung „Tschüss!“ und zum Beenden des Programms.

4.5.3 Weiterreichen von Ausnahmen

Im Allgemeinen muss eine ausgelöste Ausnahme nach der Grundregel *Catch-or-throw* behandelt werden, die besagt, dass jede Ausnahme entweder per `try-catch` behandelt oder per `throw` weitergegeben werden muss. Soll eine Ausnahme nicht behandelt, sondern weitergegeben werden, so kann dies einfach dadurch geschehen, dass eine geeignete try-catch-Anweisung nicht verwendet wird. Allerdings muss dann für die betroffene Methode in ihrer Deklaration das reservierte Wort `throws` (mit „s“ und ohne `new`!) mit einer Liste der Ausnahmen angegeben werden, die sie auslösen kann.

```
public <T> methode(...) throws Ausnahme1, Ausnahme2 {
    ...
}
```

Ausnahmen der Klasse `RuntimeException` oder einer ihrer Subklassen unterliegen nicht dieser Regel, sie müssen nicht explizit behandelt oder weitergegeben werden. Der Programmierer kann also in diesem Fall selbst entscheiden, ob er den entsprechenden Fehler behandeln will oder nicht. Im folgenden Beispielprogramm wird eine statische Methode `f2` definiert, die für negative Argumente eine `Exception` wirft und sonst die statische Methode `f1` aufruft; diese wiederum wirft für nichtnegative Argumente eine `IllegalArgumentException`.

```
1 package paket;
2
3 import static java.lang.Math.*;
4 import static javax.swing.JOptionPane.*;
5
6 class NegativesArgument extends Exception {
7     public NegativesArgument(String nachricht) {
8         super(nachricht);
9     }
10 }
11 class AbbruchAusnahme extends Exception{}
12
13 public class Exception_4 {
14     /** Gibt die Wurzel des spezifizierten Wertes zurück, wenn es nicht negativ ist.
15      * @param x eine nichtnegative reelle Zahl
16      * @return die Wurzel von x
17      * @throws IllegalArgumentException wenn x <= 0
18      */
19     public static double f1(double x) throws IllegalArgumentException {
20         if (x <= 0) throw new IllegalArgumentException("Negatives Argument: "+x);
21         return sqrt(x);
22     }
23
24     /** Gibt die Wurzel des spezifizierten Wertes zurück, wenn es positiv ist.
```



```

25      * @param x eine positive reelle Zahl
26      * @return die Wurzel von x
27      * @throws IllegalArgumentException wenn x = 0
28      * @throws NegativesArgument wenn x < 0
29      */
30      public static double f2(double x) throws IllegalArgumentException, NegativesArgument {
31          if (x < 0) throw new NegativesArgument("Negatives Argument: "+x);
32          return f1(x); // <- reicht für x=0 IllegalArgumentException weiter
33      }
34
35      public static void main(String... args) {
36          String eingabe = "";
37
38          while (true) { // Endlosschleife
39              try {
40                  eingabe = showInputDialog(null, "Gib eine Zahl von 0 bis 7 ein!");
41                  if (eingabe == null) { // Anwender klickt Äbbrechen"
42                      throw new AbbruchAusnahme();
43                  }
44                  double x = Double.parseDouble(eingabe); // kann NumberFormatException
werfen
45                  double y = f2(x); // IllegalArgumentException oder NegativesArgument
möglich
46                  showMessageDialog(null, "f2("+x+") = "+y);
47              } catch (NumberFormatException nfe) {
48                  showMessageDialog(null, "Bitte eine Zahl", "Wurzel", WARNING_MESSAGE);
49              } catch (IllegalArgumentException iae) {
50                  iae.printStackTrace();
51              } catch (NegativesArgument na) {
52                  na.printStackTrace();
53              } catch (AbbruchAusnahme aa) {
54                  break; // bricht die Endlosschleife ab
55              }
56          }
57      }
58  }

```

In den **catch**-Anweisungen der Zeilen 50 und 52 wird jeweils der Laufzeit-Stack (*Stack Trace*) auf der Konsole ausgegeben, also der Baum der zum Zeitpunkt des Ausnahmewurfs aufgerufenen Methoden. Für die Eingabe „0“ erfolgt beispielsweise die folgende Ausgabe:

```

java.lang.IllegalArgumentException: Negatives Argument: 0.0
    at paket.Exception_4.f1(Exception_4.java:20)
    at paket.Exception_4.f2(Exception_4.java:32)
    at paket.Exception_4.main(Exception_4.java:45)

```

Es ist also eine `IllegalArgumentException` in `f1` ausgelöst und über `f2` an die `main`-Methode weitergereicht worden. Entsprechend löst die Eingabe von „-1“ eine Ausnahme `NegativesArgument` mit kürzerem Laufzeitstack aus:

```

paket.NegativesArgument: Negatives Argument: -1.0

```

```
at paket.Exception_4.f2(Exception_4.java:31)
at paket.Exception_4.main(Exception_4.java:45)
```

Hier wurde die Methode f1 nämlich gar nicht erst aufgerufen.

4.6 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 Methodenrumpf:

```
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, sind also im gesamten Objekt bekannt, lokale Variablen jedoch nur innerhalb ihrer Methoden bzw. innerhalb des Blocks, in dem sie deklariert werden.
- 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 Rückgabotyp 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 weg gelassen oder es wird das Schlüsselwort **this** verwendet.

Exceptions

- Exceptions (Ausnahmen) sind Laufzeitfehler. In Java sind sie Objekte der Klasse `Exception` oder einer Unterklasse (z.B. `NumberFormatException`).
- Mit einem **try/catch**-Block kann eine im **try**-Block auftretende Exception durch den **catch**-Block gefangen werden. Für jeden Ausnahmetyp kann es einen eigenen **catch**-Block geben.

```
try {  
    ...  
} catch ( Ausnahmetyp ref ) {  
    ...  
} [catch ( Ausnahmetyp2 ref2 ) {  
    ...  
}] [finally {  
    ...  
}]
```

Die Anweisungen im **finally**-Block werden stets durchgeführt, egal ob eine Exception geworfen wurde oder nicht; er kann weggelassen werden.

- Eine Exception kann mit der folgenden Konstruktion „geworfen“ (erzeugt) werden:

throw new Ausnahmetyp ();

- Mit dem reservierten Wort **throws** kann bei der Deklaration einer Methode angegeben werden, welche Ausnahme sie werfen kann, z.B.

```
public static int parseInt(String s) throws NumberFormatException
```



Spezielle Themen

Kapitelübersicht

| | | |
|-------|---|-----|
| A.1 | ASCII und Unicode | 132 |
| A.2 | Binärsystem und Hexadezimalsystem | 133 |
| A.2.1 | Umrechnung vom Dezimal- ins Binärsystem | 135 |
| A.2.2 | Umrechnung zwischen Binär- und Hexadezimalsystem | 136 |
| A.2.3 | Binärbrüche | 136 |
| A.2.4 | Hexadezimalbrüche | 137 |
| A.3 | javadoc, der Dokumentationsgenerator | 138 |
| A.3.1 | Dokumentationskommentare | 139 |
| A.4 | jar-Archive: Ausführbare Dateien und Bibliotheken | 140 |
| A.4.1 | Manifest-Datei | 140 |
| A.5 | Zeit- und Datumfunktionen in Java | 141 |
| A.5.1 | Die Systemzeit | 142 |
| A.5.2 | LocalDateTime und Datumsformatierungen | 143 |
| A.6 | Formatierte Ausgabe mit HTML | 145 |
| A.6.1 | HTML-Ausgabe von Wahrheitstabellen | 146 |
| A.7 | Call by reference und call by value | 147 |
| A.8 | enums | 150 |
| A.8.1 | Zusammenfassung | 154 |
| A.9 | Psychologie und Logik: Der Wason-Test | 154 |

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*)

¹In der Kryptologie spricht man von „Chiffre“

| 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

| 0000 C0 Controls and Basic Latin | | | | | | | | | | 007F 0080 C1 Controls and Latin-1 Supplement | | | | | | | | | | 00FF | | | | | | | | | |
|----------------------------------|-----|-----|-----|-----|-----|-----|-----|-----|--|--|-----|-----|-----|-----|-----|-----|-----|-----|--|------|---|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 000 | 001 | 002 | 003 | 004 | 005 | 006 | 007 | | 0 | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F | | | 0 | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F |
| 1 | 000 | 001 | 002 | 003 | 004 | 005 | 006 | 007 | | 1 | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F | | | 1 | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F |
| 2 | 000 | 001 | 002 | 003 | 004 | 005 | 006 | 007 | | 2 | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F | | | 2 | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F |
| 3 | 000 | 001 | 002 | 003 | 004 | 005 | 006 | 007 | | 3 | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F | | | 3 | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F |
| 4 | 000 | 001 | 002 | 003 | 004 | 005 | 006 | 007 | | 4 | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F | | | 4 | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F |
| 5 | 000 | 001 | 002 | 003 | 004 | 005 | 006 | 007 | | 5 | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F | | | 5 | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F |
| 6 | 000 | 001 | 002 | 003 | 004 | 005 | 006 | 007 | | 6 | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F | | | 6 | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F |
| 7 | 000 | 001 | 002 | 003 | 004 | 005 | 006 | 007 | | 7 | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F | | | 7 | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F |
| 8 | 000 | 001 | 002 | 003 | 004 | 005 | 006 | 007 | | 8 | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F | | | 8 | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F |
| 9 | 000 | 001 | 002 | 003 | 004 | 005 | 006 | 007 | | 9 | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F | | | 9 | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F |
| A | 000 | 001 | 002 | 003 | 004 | 005 | 006 | 007 | | A | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F | | | A | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F |
| B | 000 | 001 | 002 | 003 | 004 | 005 | 006 | 007 | | B | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F | | | B | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F |
| C | 000 | 001 | 002 | 003 | 004 | 005 | 006 | 007 | | C | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F | | | C | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F |
| D | 000 | 001 | 002 | 003 | 004 | 005 | 006 | 007 | | D | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F | | | D | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F |
| E | 000 | 001 | 002 | 003 | 004 | 005 | 006 | 007 | | E | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F | | | E | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F |
| F | 000 | 001 | 002 | 003 | 004 | 005 | 006 | 007 | | F | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F | | | F | 008 | 009 | 00A | 00B | 00C | 00D | 00E | 00F |

(‡ Restricted Use)

Abbildung A.1. Die ersten Tabellen des Unicode.

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 Binärsystem und Hexadezimalsystem

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:

$$\text{Basis} = 10, \quad \text{Ziffernvorrat} = \{0, 1, 2, \dots, 8, 9\}$$

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 $0x199$:

$$\begin{array}{rcl} 0x3A & & 0x199 \\ \begin{array}{l} 10 \cdot 16^0 = 10 \\ + 3 \cdot 16^1 = 48 \end{array} & & \begin{array}{l} 9 \cdot 16^0 = 9 \\ + 9 \cdot 16^1 = 144 \\ + 1 \cdot 16^2 = 256 \end{array} \\ \hline 58_{10} & & 409_{10} \end{array}$$

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.

| Binärsystem | 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 mit einer angefügten Ziffernfolge repräsentiert das Oktalsystem, also Basis 8, und das Präfix 0x das Hexadezimalsystem. 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.

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

$$\boxed{m \div n = (m/n) \text{ Rest } (m \% n)}, \quad (\text{A.2})$$

²Ein „Unärsystem“ kann es nicht geben, da alle Potenzen der Basis 1 wieder 1 ergeben: $1^j = 1$ für alle j .

³Wir verwenden hier das durch die Programmiersprache C populär gewordene Zeichen „/“, das für Integer-Zahlen genau die ganzzahlige Division ergibt.

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 Umrechnung zwischen Binär- und Hexadezimalsystem

Sehr einfach ist die Umrechnung vom Binär- ins Hexadezimalsystem und umgekehrt, wenn man die Binärzahlen als vierstellige Blöcke schreibt, sogenannte *Nibbles* (früher oft „Halbytes“ oder „Tetraden“ genannt):

| Dezimal | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|-------------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| Binär | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 |
| Hexadezimal | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |

(A.3)

Für die Zahl 74 ergibt sich mit den Nibbles beispielsweise aus $0x4A$ die Binärzahl $0b01001010$ und umgekehrt, und entsprechend für 409 aus $0x199$ die Binärzahl $0b000110011001$ und umgekehrt:

| | | | | | |
|--------------|------|------|------|------|------|
| Hexadezimal: | 4 | A | 1 | 9 | 9 |
| Binär: | 0100 | 1010 | 0001 | 1001 | 1001 |

(A.4)

Beachten Sie dabei, dass die Binärzahlen in diesen Fällen mit einer führenden Null zu jeweils vollständigen Nibbles aufgefüllt wurden.

A.2.3 Binärbrüche

Die Berechnung von Dual- oder Binärbrüchen, also Kommazahlen im Binär- oder Dualsystem, 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 $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \dots$ entsprechend der Nachkommastelle k verwendet. Beispielsweise ist

$$0,101_2 = \frac{1}{2} + \frac{1}{8} = \frac{5}{8} = 0,625. \quad (\text{A.5})$$

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:

$$\lfloor x \rfloor = \max\{n \in \mathbb{Z} \mid n \leq x\} \quad (\text{A.6})$$

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 $\lfloor z \rfloor$ 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.5) bestätigt dies. Wie im Dezimalsystem gibt es periodische Brüche (mit Periode ungleich $\overline{0}$) 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.2.4 Hexadezimalbrüche

Ähnlich verhält es sich mit Hexadezimalbrüchen. Die Umrechnung eines Dezimalbruchs in einen Hexadezimalbruch erfolgt wie für einen Binärbruch, nur muss natürlich die Basis 2 nun durch 16 ausgetauscht werden. Zum Beispiel gilt für die Dezimalbrüche 0,625 und 0,3 das folgende Rechenschema:

| $z \leftarrow z - \lfloor z \rfloor$ | $16z$ | $\lfloor 16z \rfloor$ |
|--------------------------------------|-------|-----------------------|
| 0.625 | 10.0 | A |
| 0 | | |

| $z \leftarrow z - \lfloor z \rfloor$ | $16z$ | $\lfloor 16z \rfloor$ |
|--------------------------------------|-------|-----------------------|
| 0.3 | 4.8 | 4 |
| 0.8 | 12.8 | C |
| 0.8 | 12.8 | C |
| ... | ... | ... |

Die Umrechnung von Binär- in Hexadezimalbrüchen geschieht auch wieder über Nibbles nach der Tabelle (A.3), nur müssen jetzt die Binärzahlen mit *hängenden* Nullen zu Nibbles aufgefüllt werden:

| | | | |
|-------------------|------|------|---------------|
| Hexadezimal: 0x0. | A | | |
| Binär: 0, | 1010 | 4 | C C ... |
| | | 0100 | 1100 1100 ... |

(A.7)

In Java können **double**-Werte („Literele“) neben der üblichen Dezimalnotation (also 0.3 oder 3.0e-1) auch in hexadezimaler Darstellung verwendet werden. Hier muss wie für die ganzzahligen Datentypen das Präfix 0x stehen, gefolgt von einer hexadezimalen Ziffernfolge mit dem Punkt als Hexadezimaltrenner und einem abschließenden p oder P, dem *Binärexponentenindikator* und einer vorzeichenbehafteten Ganzzahl [JLS, §3.10.2]. Beispielsweise lässt sich 0,3 wie folgt hexadezimal darstellen:

```
double x = 0x0.4CCCCCCCCCCCCp0;
```

Die Zahl hinter dem Binärexponentenindikator p bestimmt dabei die Position des Gleitkommas in der Binärdarstellung gemäß IEEE754. Mit p0 wird die Zahl davor ohne Gleitkommaverschiebung bewertet, allgemein wird mit $p \pm n$ der Wert zwischen 0x und p mit $2^{\pm n}$ multipliziert.

Zum Ausprobieren oder zur Probe selbsterrechneter Umrechnungswerte sei auf die Seite <http://www.arndt-bruenner.de/mathe/scripts/Zahlensysteme.htm> verwiesen.

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 geschützten (*protected*) der Klasse Test.java, so verwendet man die Option -protected, also

```
javadoc -protected 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 vollständige 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 platziert 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  $m$ -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 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

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“

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.

A.5.1 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.)

Die Klasse `LocalDateTime` ist eine für die meisten Zwecke bezüglich Datums- und Zeitangaben ausreichende Klasse. Das Paket `java.time` stellt jedoch zwei weitere wichtige Klassen zur Verfügung: Die Klasse `LocalDate` erlaubt die Speicherung eines Datums ohne Uhrzeiten, und die Klasse `ZonedDateTime` repräsentiert ein Datum mit Uhrzeit und Zeitzone. Die Methoden der beiden Klassen sind ähnlich aufgebaut wie diejenigen von `LocalDateTime`.

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 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 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;
```

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>

A.5.2 LocalDateTime und Datumsformatierungen

Die Klasse `LocalDateTime` befindet sich im Paket `java.time` und liefert das fast auf der ganzen Welt verwendete Kalendersystem ISO-8601, den „proleptischen gregorianischen Kalender“.⁷ Ein Objekt von `LocalDateTime` wird am einfachsten mit der statischen Methode `now()` oder einer der statischen Methoden `of(...)` erzeugt. Mit `now` wird die aktuelle Systemzeit dargestellt, mit den `of`-Methoden erzeugt man ein bestimmtes Datum, zum Beispiel:

```
1  LocalDateTime jetzt = LocalDateTime.now();
2  LocalDateTime damals = LocalDateTime.of(1984, 05, 20, 10,30, 8);
```

Hier repräsentiert `damals` also das Datum 20.05.1984 um 10:30 Uhr und 8 Sekunden. Einige Objektmethode der Klasse `LocalDateTime` ermöglichen das Rechnen mit Zeiten, zum Beispiel die Methode `minusHours(long x)` zur Subtraktion von `x` Stunden:

```
1  LocalDateTime zeit = LocalDateTime.now();
2  zeit = zeit.minusHours(5);
```

Die Klasse `DateTimeFormatter` des Pakets `java.time.format` ermöglicht mit der statischen Methode `ofPattern` 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. A.1. Diese Buchstaben werden in einem Formatierungsstring durch ih-

| Symbol | Date or Time Component | Beispiele |
|--------|------------------------|---------------------------------------|
| G | Era designator | n.Chr., AD |
| y | Year | 1996; 96 |
| M/L | 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 A.1. Formatierungsbuchstaben für `DateTimeFormatter`.

re entsprechenden Werte ersetzt. Ein Stringabschnitt, der von Apostrophs `'...'` eingeschlossen

⁷ Der gregorianische Kalender wurde am 15.10.1582 durch Papst Gregor XIII. in den meisten der damaligen katholischen Länder und den Niederlanden eingeführt. Davor galt in weiten Teilen der alten Welt der julianische Kalender. (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 ...). Der 5.10.1582 julianisch war der 15.10.1582 gregorianisch, d.h. der julianische Kalender hinkt dem Gregorianischen um 10 Tage hinterher. (Durch die leicht unterschiedliche Berechnung der Schaltjahre reduziert sich dieser Unterschied jedoch, je weiter man in die Vergangenheit blickt.) „Proleptisch“ bedeutet hier, dass das gregorianische Kalendersystem einfach auch auf Zeiten vor 1593 angewendet wird, was dann nicht mehr mit den historischen Julianischen Angaben übereinstimmt.

ist, wird 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“.

Daneben kann man auch einen der vordefinierten Standardformater verwenden, zum Beispiel `DateTimeFormatter.ISO_DATE` für das ISO-Format yyyy-mm-dd. Näheres siehe Java API Dokumentation der Klasse `DateTimeFormatter`.

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.

```

1 import java.time.LocalDateTime;
2 import java.time.format.DateTimeFormatter;
3
4 /**
5  * Datums- und Zeitfunktionen.
6  * @author de Vries
7  */
8 public class ZeitTest {
9     public static void main (String[] args) {
10         String ausgabe;
11
12         LocalDateTime jetzt = LocalDateTime.now(), damals = LocalDateTime.of(1964,05,20,10,30);
13         DateTimeFormatter datum1 = DateTimeFormatter.ofPattern("dd.MM.yyyy, H:mm:ss 'Uhr, KW");
14         DateTimeFormatter datum2 = DateTimeFormatter.ofPattern("EEEE', den 'd. MMMM yyyy' um 'H:mm:ss,SSS' Uhr");
15
16         // Ausgabe und Ende:
17         ausgabe = "Datum heute: " + jetzt.getDayOfMonth();
18         ausgabe += "." + jetzt.getMonthValue();
19         ausgabe += "." + jetzt.getYear();
20         ausgabe += ", KW " + jetzt.get(java.time.temporal.IsoFields.WEEK_OF_WEEK_BASED_YEAR);
21         ausgabe += "\n\noder: " + jetzt.format(datum1);
22         ausgabe += "\n\noder: " + jetzt.format(datum2);
23
24         jetzt = jetzt.minusSeconds(3601);
25         ausgabe += "\n\n - 3601 sec: " + jetzt.format(datum1);
26
27         javax.swing.JOptionPane.showMessageDialog(null, ausgabe, "Datum", -1);
28     }
29 }

```


A.6 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.⁸

| 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>
```

⁸ Nach XHTML muss der Zeilenumbruch-Tag `
` lauten, doch leider interpretiert Java diesen Tag (noch?) nicht.

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.6.1 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 Boole'schen Algebra:

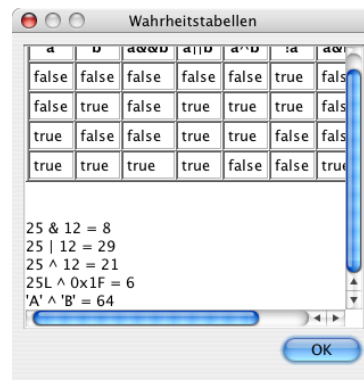


Abbildung A.2. Ausgabe der Applikation LogischeOperatoren.

A.7 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;

```

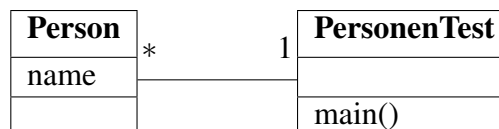
In Java gibt es jedoch einen fundamentalen Unterschied zwischen Objekten und Daten primitiver Datentypen, also **boolean**, **int**, **double**, Anschaulich gesprochen sind primitive Datentypen die „Atome“, aus denen Objekte zusammengesetzt sind. Objekte bestehen aus Daten (also aus Objekten oder primitiven 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 += "\n\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.8 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 Pluto⁹). 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.

```
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 }
```

⁹ <http://www.iau2006.org/mirror/www.iau.org/iau0603/>

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.8.1 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.

A.9 Psychologie und Logik: Der Wason-Test

Der englische Psychologe Peter Wason hat 1966 die Schwierigkeit des abstrakten logischen Denkens durch den Vier-Karten-Test veranschaulicht, indem er der abstrakten Variante eine Version gegenüberstellte, die in eine alltägliche Szene eingekleidet ist,¹⁰ siehe Tabelle A.2. Die

| Abstrakte Version | Alltagstaugliche Version |
|--|---|
| Vier Karten liegen auf dem Tisch. Sie tragen auf der einen Seite einen Buchstaben, auf der Rückseite eine Zahl. Wenn sie auf der einen Seite einen Vokal trägt, dann steht auf der Rückseite eine gerade Zahl. | Vier junge Leute sitzen an einem Tisch in einer Kneipe mit Getränken und dem Ausweis vor sich. In dieser Kneipe gibt es Alkohol erst ab 18 Jahren. |
| Die Karten auf dem Tisch zeigen: E K 4 7 | Sie wissen über die vier Jugendlichen: Alfred trinkt Bier. Ausweis verdeckt. Berta trinkt Limo. Ausweis verdeckt. Was Cindy trinkt, ist unklar. Sie ist älter als 18. Was Dominik trinkt, ist unklar. Er ist noch nicht 18. |
| Sie sollen feststellen, ob auf den Karten die Regel eingehalten wird. Welche Karten müssen Sie umdrehen? | Sie sollen feststellen, ob die Regel eingehalten wird. Welche Ausweise und welche Getränke müssen Sie überprüfen? |

Tabelle A.2. Wasons Vier-Karten-Test

¹⁰ WASON, P. C.: Reasoning. In: FOSS, B. M. (Hrsg.): *New Horizons in Psychology*. Harmondsworth : Penguin, 1966. S. 135–151

beiden Aufgaben sind strukturell identisch. „Vokal“ entspricht „Alkohol“, „Konsonant“ entspricht „kein Alkohol“, „gerade Zahl“ entspricht „älter als 18“ und „ungerade Zahl“ entspricht „noch nicht 18“. Die beiden Aufgaben werden aber dennoch als sehr unterschiedlich schwierig wahrgenommen. Mit der konkreten Fassung hat kaum jemand Schwierigkeiten, die abstrakte Fassung wird von den meisten falsch beantwortet. Im Kern geht es bei den beiden Versionen um das Problem, eine Regel

$$A \rightarrow B,$$

also „aus A folgt B “ oder „wenn A , dann B “, entweder direkt oder durch die äquivalente Aussage $\neg B \rightarrow \neg A$ zu überprüfen, wenn eine der beiden Aussagen A oder B unbekannt sind. A ist hierbei die hinreichende Bedingung, B die notwendige. Anders gesagt: Wenn A gilt, *muss* B gelten, und umgekehrt: wenn B nicht gilt, *darf* A nicht gelten.

| | Abstrakte Version | Alltagstaugliche Version |
|-----|---|---|
| A | „Die Vorderseite der Münze trägt einen Vokal“ | „Die Person trinkt Alkohol.“ |
| B | „Auf der Rückseite der Münze steht eine gerade Zahl.“ | „Die Person ist mindestens 18 Jahre alt.“ |

Von den jeweils vier logisch möglichen Fällen braucht B („mindestens 18“) nicht überprüft zu werden, wenn A falsch ist („trinkt keinen Alkohol“), und A ist egal, wenn B wahr ist.

Die Tatsache, dass die meisten Menschen die abstrakte Variante viel schwieriger finden als die konkrete, weist auf eine grundlegende Eigenschaft der menschlichen Intelligenz hin. Logisches, abstraktes Denken scheint nicht zur Grundausstattung unseres Gehirns zu gehören.¹¹ Wird ein abstraktes logisches Problem jedoch in einen sozialen Kontext gebracht, so ermöglichen die menschlichen Denkstrukturen eine effiziente und schnelle Lösung. Über die tiefere Ursache dieses Phänomens herrscht immer noch Unklarheit in der theoretischen Psychologie. Cosmides und Tooby vermuten, dass ganz allgemein die menschliche Intelligenz sich evolutionsgeschichtlich erst durch solche kontextsensitiven und domänenspezifischen Mechanismen entwickeln und durchsetzen konnte, da sie Vorteile bei der natürlichen Auswahl erbrachte.¹² Allerdings ist dieser Ansatz der Evolutionspsychologie in dieser Konsequenz nicht unumstritten.¹³

Auf der anderen Seite sind Computer gerade genau so konstruiert, dass sie logische Möglichkeiten einer Regel $A \rightarrow B$ formal überprüfen können, den Kontext verstehen sie nicht, er ist ihnen auch egal. Es ist genau die Aufgabe einer Informatikerin oder eines Informatikers, ein kontextsensitives Denken in ein abstraktes Denken zu übertragen.

¹¹ MAURER, B.: *Mathematik. Die faszinierende Welt der Zahlen*. Köln: Fackelträger Verlag, 2015, S. 14f

¹² COSMIDES, L.; TOOBY, J. (1992). 'Cognitive Adaptions for Social Exchange'. In BARKOW, J.; COSMIDES, L.; TOOBY, J.: *The Adapted Mind. Evolutionary Psychology and the Generation of Culture*. New York: Oxford University Press. pp. 163–228.

¹³ DAVIES, P.S.; FETZER, J.H.; FOSTER, T.R. (1995). 'Logical reasoning and domain specificity'. *Biology and Philosophy*. **10**(1), 1–37. doi 10.1007%2FBF00851985. Eine wesentliche Kritik ist, dass die abstrakte Version eine willkürliche logische Regel beschreibt, die alltagstaugliche Version dagegen eine geläufige soziale bzw. rechtliche Norm; im ersten Fall muss man sich also gleichzeitig die Regel *und* deren Einhaltung verdeutlichen, während im zweiten die Überprüfung der Regeleinhaltung deutlich im Vordergrund steht, da die Regel selbst klar und einsichtig ist.

Literaturverzeichnis

- [1] BERGMANN, L. ; SCHAEFER, C. ; RAITH, W. : *Lehrbuch der Experimentalphysik. Band 2. Elektromagnetismus*. 9. Aufl. Berlin New York : Walter de Gruyter, 2006
- [2] GUMM, H. P. ; SOMMER, M. : *Einführung in die Informatik*. München : Oldenbourg Verlag, 2013
- [3] HOFFMANN, D. W.: *Theoretische Informatik*. München : Carl Hanser Verlag, 2009
- [4] KREFT, K. ; LANGER, A. : ‘Effective Java: Let’s Lambda!’. In: *Java Magazin* 12 (2013), S. 21–27
- [5] KRÜGER, G. ; HANSEN, H. : *Java-Programmierung. Das Handbuch zu Java 8*. 8. Aufl. Köln : O’Reilly, 2014. – <http://www.javabuch.de>
- [6] RATZ, D. ; SCHEFFLER, J. ; SEESE, D. ; WIESENBERGER, J. : *Grundkurs Programmieren in Java*. 6. Aufl. München Wien : Hanser Verlag, 2011
- [7] RÖPCKE, H. ; WESSLER, M. : *Wirtschaftsmathematik. Methoden – Beispiele – Anwendungen*. München : Hanser, 2012
- [8] ULLENBOOM, C. : *Java ist auch eine Insel*. Bonn : Rheinwerk Computing, 2016. – <http://www.tutego.de/javabuch/>
- [9] VOSSEN, G. ; WITT, K.-U. : *Grundkurs Theoretische Informatik*. 6. Aufl. Wiesbaden : Springer Vieweg, 2016. – DOI 10.1007/978-3-8348-2202-4
- [10] WIRTH, N. : *Algorithmen und Datenstrukturen*. Stuttgart Leipzig : B.G. Teubner, 1999

Internetquellen

- [API] Java API Specification (auf Englisch): <http://java.sun.com/reference/api/>
- [JLS] Java Language and Virtual Machine Specifications (technische Sprach- und Interpreterspezifikationen, auf Englisch): <http://docs.oracle.com/javase/specs/>
- [Tut] Java Tutorials (auf Englisch): <http://docs.oracle.com/javase/tutorial/>
- [KH] Guido Krüger & Heiko Hansen: *Java-Programmierung* Online <http://www.javabuch.de/>
- [LdS] H. Lenstra & B. de Smit: *Escher and the Droste effect*. <http://escherdroste.math.leidenuniv.nl/> – Mathematisch begründete Vervollständigungen von Eschers Kunstdruckgalerie
- [Le] Jos Ley: *The mathematics behind the Droste effect*. http://www.josleys.com/article_show.php?id=82 – Mathematik zur Rekursion in der bildenden Kunst

Index

boolean, 22
break, 126
byte, 22
catch, 125, 126
char, 22
class, 10, 11
continue, 126
double, 22, 138
do, 52
enum, 150
extends, 117
false, 22
final , 56
finally, 126
float, 22
for, 50, 57
int, 22
long, 22, 46
new, 103
null, 103
package, 122
private, 123
protected, 119, 123
public, 10, 11, 123
return, 65, 72, 126
short, 22
static, 100, 101
super, 118
this, 98, 118
throws, 128
throw, 126, 127
true, 22
try, 125
void, 73
while, 52
->, 65
0b, 135
0x, 135
? : , 47
ArrayIndexOutOfBoundsException, 125
BinaryOperator, 68
Exception, 125
IllegalArgumentException, 125
Math.random(), 41
NullPointerException, 125
NumberFormatException, 125
Object, 118
RuntimeException, 125
!=, 40
import, 15
javap -c, 89
split, 112
^, 40, 45
|, 40, 45
||, 40
~, 45
&, 40, 45
&&, 40
!=, 40
<<, 45
>>>, 45
>>, 45
Throwable, 126
%=, 25
*=", 25
++, 25
+=", 25
--, 25
-=", 25
/=", 25
==, 39
!=, 40
Ackermannfunktion, 92
Adresse, 54
Algebra, 67
Algorithmus, 37, 48, 85, 105
Alphabet, 132
Alternative, 43
AND, 40
anonyme Funktion, 66
Anweisung, 12
API, 5, 17
apply, 67
Argument, 73, 77, 78
Array, 27, 54
ASCII, 132
Attribut, 17, 57, 95
Attribute, 96
– und lokale Variablen, 98
Aufruf, 73
Aufrufbaum, 83
Aufruferliste, 125
Aufzählungstyp, 150
Ausnahme, 124
Auswahlstruktur, 38
Bankschalter, 113
Basis, 133
Basisfall, 83

- Baum, 87
- Bedingung, 38, 39
- Bedingungsoperator, 47
- Bezeichner, 11
- Binärbruch, 136
- binärer Baum, 87
- Binärsystem, 134
- Bitmaske, 46
- bitweiser Operator, 45
- Block, 12
- Boole'sche Algebra, 41, 147
- Boole'scher Ausdruck, 38
- Boole'scher Operator, 69
- Bytecode, 8
- call by value, 148
- Callback-Funktion, 66
- CallStack, 125
- Cast-Operator, 22
- Catch-or-throw, 128
- CLASSPATH, 122
- clone, 118
- Cluster, 88
- Code, 132
- Codepoint, 16
- codeConsumer, 66
- codeSupplier, 66
- Compiler, 7
- Daten, 96
- Datenstruktur, 53
- Datentyp, 20, 21
 - komplexer -, 102
 - primitiver -, 21
- DateTimeFormatter, 143
- Deklaration, 20, 102
 - einer Methode, 72
- Dekrementoperator, 25
- Dezimalsystem, 133
- Direktzugriff, 54
- Divide and Conquer, 85
- Division, ganzzahlige, 135
- Dokumentationskommentar, 14
- double-Literale, 138
- double-Zahl, 24
- Double.parseDouble, 32
- Dualbruch, 136
- Dualsystem, 134
- Eingabe, 63
- Eingabeblock, 27, 55
- Endlosschleife, 48, 126
- Endrekursion, 88
- enhanced for-loop, 57
- Entwurf, 105
- enum, 150
- EnumMap, 153
- EnumSet, 153
- equals, 40, 118
- Erlösfunktion, 68
- Error, 126
- Erzeugung eines Objekts, 103
- Escape-Sequenz, 16
- Euler'sche Zahl, 18
- Euro-Zeichen, 49
- Exception, 32
- exception, 124
- ExceptionHandler, 124
- Feld, 54
- field, 95
- floor-bracket, 137
- for-each-Schleife, 57
- for-Schleife, 50
- Formatierung von Zahlen, 52
- Funktion, 63, 66
- funktionales Interface, 66
- Funktionalkalkül, 67
- Funktionenalgebra, 67
- Funktionsauswertung, 67
- ganze Zahlen, 135
- ganzzahlige Division, 135
- Gaußklammer, 137
- Generalisierung, 116, 117
- get-Methode, 97
- getClass, 118
- Gewinnfunktion, 68
- Gosling, James, 6
- Grammatik, 8
- Gregorianischer Kalender, 141
- höhere Ordnung, Funktion, 67
- Hüllklasse, 32
- Halbbyte, 136
- hashCode, 118
- Heap, 101, 103
- Hexadezimalbruch, 137
- Hexadezimalsystem, 136
- HTML, 145
- IDE, 9
- Identifizier, 11
- IEEE754, 138
- if-else-if-Leiter, 43
- Implementierung, 106
- Index, 54
- Initialisierung, 22
- Inkrementoperator, 25
- Instanz, 96
- instanziieren, 103
- Instanzvariable, 96
- Instanzvariablen
 - und lokale Variablen, 98
- Interface, 66
- Interpreter, 7
- Iteration, 47, 83, 87
- jar, 15
- jar-Archiv, 140

- java, 8
- Java Runtime Environment, 8
- Java Virtual Machine, 8
- Java VM, 89
- javac, 8
- javadoc, 138
- JDK, 8
- JOptionPane, 11
- JRE, 8
- JScrollPane, 57
- JTextArea, 57
- TextField, 104
- JVM, 8, 101

- Kalender, 141
- Kapselung, 96, 123
- kaskadierten Verzweigung, 43
- Kellerspeicher, 93
- Klammerung, 13
- Klasse, 8, 10, 95
- Klassenattribut, 100
- Klassendeklaration, 11
- Klassendiagramm, 10, 74, 97
- Klassenmethode, 75, 100
- Klassenvariable, 56
- kombinierter Operator, 24
- Kommazahl, binäre –, 136
- Kommentar, 14
- Konkatenation, 24, 29
- konkatenieren, 28
- Konsole, 17
- Konstante, 18, 56
- konstantenspezifische Methode, 153
- Konstruktor, 98, 103
- konvertieren, 31
- Kostenfunktion, 68
- Kreiszahl, 18

- Lambda-Ausdruck, 65
- Laufzeitfehler, 32, 124
- Leerraum, 14
- length, 57
- lineare Rekursion, 91
- Literal, 13, 138
- LocalDateTime, 143
- Locale, 141, 143
- Logikgatter, 69
- logischer Operator, 40
- lokale Variable, 78, 100

- main, 11, 101
- Manifest, 140
- Math.random(), 41
- mathematische Konstanten, 18
- mehrdimensionale Arrays, 58
- mehrfache Rekursion, 92
- Mehrkernprozessor, 88
- Method Area, 101
- Methode, 11, 70, 95
 - nichtstatische -, 100
 - statische -, 73
- Methoden, 96
- Methodendeklaration, 72
- Methodenname, 71
- Methodenrumpf, 11, 71
- modal, 28
- modulo, 135
- Multicore Processor, 88

- Nebenläufigkeit, 88
- Nibble, 136, 138
- nichtstatische Methode, 100
- null, 103

- Object, 118
- Objekt, 96
 - Erzeugung, 103
- Objektmethode, 100
- Objektorientierung, 95
- Objektzustand, 113
- Op-Code, 89
- Operand, 19, 21
- Operator, 19, 21, 69
 - kombinierter –, 24
 - logischer –, 40
 - unärer –, 24
- OR, 40

- Paket, 14, 122
- parallele Rechnerarchitektur, 88
- Parameter, 62, 63, 71, 77
- parseInt, 31
- Pfeil-Symbol, 65
- pi, 18
- Pointer, 54, 148
- POJO, 100
- Polymorphismus, 117
- Präfix, 138
- Präzedenz, 23
- primitive Rekursion, 92
- print(), 17
- println(), 17
- Problembereich, 96
- Pseudocode, 48, 105
- public, 98

- Quelltext, 10

- Rückgabe, 62
- Rückruffunktion, 66
- random access, 54
- random(), 41
- Rechnerverbund, 88
- Referenz, 98, 148
- Reihung, 54
- Rekursion, 82, 83
- replace, 32
- reserviertes Wort, 11, 13
- RGB, 46, 146
- Routine, 37

- Schlüsselwort, 13
- Schleife, 47, 87
- Scroll-Balken, 57
- Selektion, 38
- Sequenzdiagramm, 113
- sequenziell, 10
- set-Methode, 97
- setText, 57
- Short-Circuit-Evaluation, 45
- Sichtbarkeit, 123
- Signatur, 67, 72
- Software Engineering, 104
- Source-Code, 10
- split, 112
- Stack, 93, 101
- Stack Trace, 129
- Standardkonstruktor, 98
- Stapelspeicher, 93
- static, 100
- statisch, 100
- statische Methode, 73
- String, 16, 24
- Subroutine, 62
- Superklasse, 118
- Swing, 11
- Syntax, 8, 13
- System.currentTimeMillis(), 142
- System.nanoTime(), 142
- System.out.print(), 17
- System.out.println(), 17
- Systemzeit, 142

- Tag, 145
- Tetrade, 136
- this, 104
- Throwable, 127
- toString, 118
- toString(), 97
- tree recursion, 92
- Tuerme von Hanoi, 86
 - iterative Lösung, 92
- Typumwandlung, 23

- ueberladen, 98, 117
- ueberschreiben, 117
- UML, 97
- Umsatzfunktion, 68
- unär, 25
- unärer Operator, 24
- Unicode, 16, 133
- untere Gaußklammer, 137
- Unterprogramm, 62

- Variable, 20, 102, 147
 - lokale, 78, 100
- verbesserte for-Schleife, 57
- Vererbung, 116, 117
- Vergleichsoperatoren, 39
- verschachtelte Rekursion, 92
- verteilte Programmierung, 88

- verzweigende Rekursion, 92
- Verzweigung, 38
- virtuell, 96
- Virtuelle Maschine, 8
- VM, 8, 89

- Wahrheitstabelle, 147
- Wahrheitstabellen, 40
- Wert, 20
- Wertigkeit, 23
- Whitespace, 14
- Wiederholung, 47, 87
- Wort, reserviertes –, 13
- Wrapper-Klasse, 32

- XOR, 40

- Zählschleife, 50
- Zahlen, 133
 - ganze, 135
- Zeiger, 54, 148
- Zugriffsmodifikator, 123
- Zustand eines Objekts, 113
- Zuweisungsoperator, 21