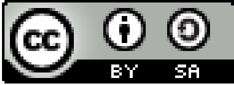
Informatikskript für den Unterricht von C. Jost am Studienkolleg für ausländische Studierende der Technischen Universität Darmstadt

Christof Jost

12. Oktober 2015

Lizenz

Dieses Werk bzw. dessen Inhalt steht unter einer Creative Commons Namensnennung - Weitergabe unter gleichen Bedingungen 3.0 Unported Lizenz. Autor: Christof Jost.



Vorbemerkungen

Es gibt mit Sicherheit etliche Fehler in diesem Skript. Dies gilt umso mehr, als dass das Werk eben grundlegend überarbeitet wurde und weite Teile komplett neu geschrieben sind und noch keine Korrektur erfahren haben. Ich bitte, die Fehler an die Adresse cjost„at“stk.tu-darmstadt.de mitzuteilen. Der Autor übernimmt keinerlei Verantwortung für die Korrektheit der Inhalte. Die Links zu (anderen) www-Dokumenten in der elektronischen Version wurden nach bestem Wissen und Gewissen erstellt. Der Autor übernimmt dennoch keine Haftung für die Inhalte verlinkter Seiten. Beachten Sie, dass der Lehrgang viel mehr Übungen enthält, als hier im Skript wiedergegeben sind. Zum Selbststudium ist daher das Skript als alleinige Literatur ungeeignet.

Die Informatik ist die Wissenschaft von der Verarbeitung von Information, insbesondere die automatische Verarbeitung von Information mittels Rechnern. Die Informatik teilt sich auf in die Teilgebiete technische Informatik, praktische Informatik, angewandte Informatik und theoretische Informatik. Dieses Skript soll eine kleine Einführung in die technische Informatik geben und außerdem grundlegendes Wissen zum Programmieren in Java vermitteln, was zum Bereich praktische Informatik zählt. Beachten Sie, dass praktische Informatik nicht nur Programmieren ist. Zur praktischen Informatik gehören weitere Bereiche wie Datenbanken, Betriebssysteme oder Netzwerke. Angewandte Informatik wird in diesem Lehrgang nur gestreift, theoretische Informatik wird nicht behandelt.

Inhaltsverzeichnis

1 Hardware **5**
1 Netzteil . . . 5
2 Prozessor . . . 5
3 Speicher . . . 6
3.1 Festplatte . . . 6
3.2 Arbeitsspeicher . . . 6

2 Programmieren mit Java – Teil1 **7**
1 Grundbegriffe der Programmierung . . . 7
1.1 Compiler . . . 7
1.2 Bytecode bei Java . . . 8
1.3 Entwicklungsumgebungen . . . 9
2 Erste Programme in Java . . . 10
2.1 Wie beginnt ein Java-Programm? . . . 10
2.2 Ausgabeanweisungen . . . 10
2.3 Variablen, Datentypen, Zuweisungsoperator . . . 13
2.4 Methoden ohne Rückgabe . . . 14
2.5 Kommentare . . . 17
3 Objektorientierte Programmierung . . . 18
3.1 Einführung . . . 18
3.2 Aufbau einer Java-Klasse . . . 19
3.3 Java-Konventionen . . . 21
3.4 Typen . . . 22
3.5 Methoden mit Rückgabe . . . 23
3.6 Zugriffmodifizierer . . . 26
4 Kontrollstrukturen . . . 27
4.1 Export und Import in Eclipse . . . 27
4.2 if und if-else . . . 27
4.3 Die while-Schleife . . . 32
4.4 Zufall . . . 34
4.5 Die for-Schleife . . . 35

5 Arrays . . . 39
5.1 Wozu Arrays? . . . 39
5.2 Ein Beispiel für Arrays: Vektoren . . . 40

3 Grundlagen der Technischen Informatik 45
1 Zahlensysteme . . . 45
1.1 Römische Zahlen – ein Additionssystem . . . 45
1.2 Das Dezimalsystem – ein Stellenwertsystem . . . 46
1.3 Das Dualsystem – ein Stellenwertsystem . . . 47
1.4 Weitere Stellenwertsysteme . . . 49
1.5 Vorzeichendarstellung im Dualsystem . . . 51
1.6 Gleitkommazahlen . . . 54
2 Boolesche Algebra und Schaltnetze . . . 56
2.1 Boolesche Funktionen von einer Variablen . . . 56
2.2 Boolesche Funktionen von zwei Variablen . . . 57
2.3 Gatter . . . 58
2.4 Vollständige Operatorensysteme . . . 60
2.5 Umformungen Boolescher Ausdrücke . . . 62
2.6 Kanonische Normalformen . . . 65
2.7 Minimale Normalformen und Karnaugh-Diagramme . . . 68
2.8 Ein Beispiel mit Don't-Cares . . . 73
2.9 Der Carry-Ripple-Addierer . . . 75

4 Programmieren in Java – Teil 2 81
1 Ein genauerer Blick auf Variablen . . . 81
1.1 Initialisierung von Variablen und Sichtbarkeit von Variablen . . . 81
1.2 Variablen mit gleichem Namen und this-Operator . . . 82
2 Algorithmik am Beispiel Sortierung . . . 84
2.1 Einführung . . . 84
2.2 Selectionsort . . . 85
2.3 Bubblesort . . . 86
2.4 Insertionsort . . . 87
3 Rekursion . . . 88
3.1 Ein schlechtes Beispiel zur Einführung: Fakultät . . . 88
3.2 Ein besseres Beispiel: Türme von Hanoi . . . 90
3.3 Quicksort . . . 92
3.4 Mergesort . . . 94

5 Anhang 97
1 Literatur zu Java und Programmierung . . . 97

Kapitel 1

Hardware

Vorbemerkung: Ein Computer darf nur bei ausgestecktem Netzstecker und nur von oder mit fachkundigen Personen (Elektro-Ingenieur, Informatiker, Elektriker) geöffnet und gewartet werden.

1 Netzteil

Das Wort „Netz“ im Kompositum „Netzteil“ bezieht sich auf das Stromnetz, nicht auf das Datennetz. Das Netzteil wird über Kabel und Netzstecker an das 230-Volt-Wechselstromnetz angeschlossen. Die verschiedenen Komponenten des Computers benötigen Gleichstrom bei verschiedenen, aber immer wesentlich niedrigeren Spannungen (3 – 24 Volt). Aufgabe des Netzteils ist es, aus den 230 Volt Wechselspannung durch Transformatoren, Gleichrichter und anderen Bauteilen diese Versorgungsspannungen zu erzeugen.

Wartbarkeit: Da das Netzteil mit den potentiell tödlichen 230V des Stromnetzes verbunden wird, ist eine Manipulation oder fehlerhafte Reparatur eines Netzteils gefährlich. Eine Reparatur durch Laien muss unter allen Umständen unterbleiben.

2 Prozessor

Der Prozessor ist das Gehirn des Computers. Er erledigt die eigentliche Rechenarbeit. Er sitzt in einem Sockel auf der Hauptplatine (englisch: Motherboard). Den eigentlichen Prozessor sieht man dabei nicht, da er unter einem Lüfter zur Kühlung sitzt.

Wartbarkeit: Das Austauschen eines Prozessors gehört zu den anspruchsvolleren Reparaturaufgaben. Der Prozessor muss zu den anderen Komponenten passen, daher lässt sich durch Austauschen des Prozessors in einem alten Computer meist keine große Verbesserung der Rechenleistung erzielen. Prozessoren haben selten Defekte. Daher ist der Austausch des Prozessors im Normalfall keine Arbeit für den Benutzer.

3 Speicher

3.1 Festplatte

Die Festplatte dient dem dauerhaften Speichern großer Datenmengen. „Dauerhaft“ bedeutet, dass die Daten beim Abschalten des Computers nicht verloren gehen, sondern nach dem Einschalten wieder zur Verfügung stehen. Festplatten bestehen aus rotierenden Scheiben, auf deren Oberflächen durch magnetische Veränderungen Information gespeichert wird. Eine moderne Festplatte hat eine Speicherkapazität von etwa 1 TB (Terabyte) oder 1000 GB (Gigabyte).

Wartbarkeit: Da Festplatten mechanisch funktionieren, sind sie fehleranfällig, so dass das Austauschen notwendig werden kann. Das Austauschen einer Festplatte ist einfach. Aber: Der Aufwand, anschließend wieder alle Daten herzustellen, ist groß, insbesondere, wenn das Betriebssystem betroffen ist.

3.2 Arbeitsspeicher

Der Festplattenzugriff ist viel zu langsam, um mit den Daten von der Festplatte direkt zu arbeiten. Die Daten, mit denen momentan gearbeitet wird, werden daher von der Festplatte in den Arbeitsspeicher geladen. Der Zugriff auf Daten vom Arbeitsspeicher ist ca. tausendmal schneller als der Zugriff auf die Festplatte. Dennoch kann der Arbeitsspeicher die Festplatte nicht ersetzen, denn der Arbeitsspeicher kann Daten nicht dauerhaft speichern. Er verliert beim Abschalten des Stroms sein Gedächtnis. Außerdem ist der Arbeitsspeicher zu klein und dabei zu teuer, um alle Daten zu halten. Ein moderner Computer verfügt über etwa 8 GB Arbeitsspeicher. Der Arbeitsspeicher sitzt in Steckplätzen auf der Hauptplatine.

Wartbarkeit: Oftmals gibt es in Computern freie Steckplätze für zusätzlichen Arbeitsspeicher. Das Vergrößern des Arbeitsspeichers kann einen älteren, langsamen Computer wieder geeigneter für moderne, aufwändige Software machen. Das Austauschen oder Vergrößern des Arbeitsspeichers ist einfach.

Kapitel 2

Programmieren mit Java – Teil1

1 Grundbegriffe der Programmierung

1.1 Compiler

Ein Computer „versteht“ nur binäre Daten, also Daten, die aus zwei Zeichen, üblicherweise schreibt man für diese „0“ und „1“, bestehen. Alle Zahlen und andere Daten, die ein Prozessor verarbeiten soll, müssen auf diese Form gebracht werden. Auch Befehle für einen Prozessor haben diese Form. Man spricht von „Maschinensprache“. Verschiedene Prozessoren benutzen dabei verschiedene Maschinensprachen, das heißt, es gibt unterschiedliche Befehle und selbst die gleichen Befehle haben nicht unbedingt denselben Code aus 0 und 1. Da diese Ketten aus Nullen und Einsen für einen Menschen schwer zu merken und zu lesen sind, programmiert man Computer in einer Programmiersprache. Die Befehle einer Programmiersprache nennt man auch Anweisungen. Sie lehnen sich meistens an die englische Sprache an und sind besser zu merken und zu verstehen. Das in einer Programmiersprache geschriebene Programm, „Quellcode“ genannt, muss dann in Maschinensprache übersetzt werden. Diese Übersetzung erledigt ein Programm, das man „Compiler“ nennt. Statt „übersetzen“ sagt man meistens „kompilieren“. So wie es

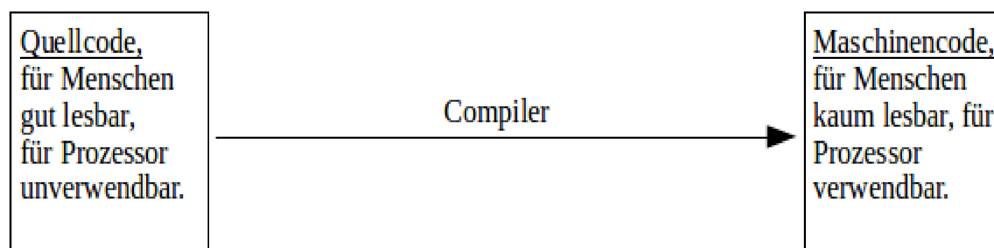


Abbildung 2.1: Codeumwandlung

Dolmetscher gibt, die von Deutsch nach Griechisch übersetzen, und andere, die für die Übersetzung von Arabisch in Farsi zuständig sind, gibt es Compiler, die verschiedene Programmiersprachen in Maschinensprache für verschiedene Rechner mit unterschiedlichen Betriebssystemen oder Prozessoren kompilieren. Haben Sie beispielsweise ein Programm in der Programmiersprache „Fortran“ geschrieben und wollen es auf einem Apple-Mac-Computer laufen lassen, dann brauchen Sie einen Compiler, der aus Fortran-Quellcode den Maschinencode für einen Apple-Mac-Computer erzeugt.

1.2 Bytecode bei Java

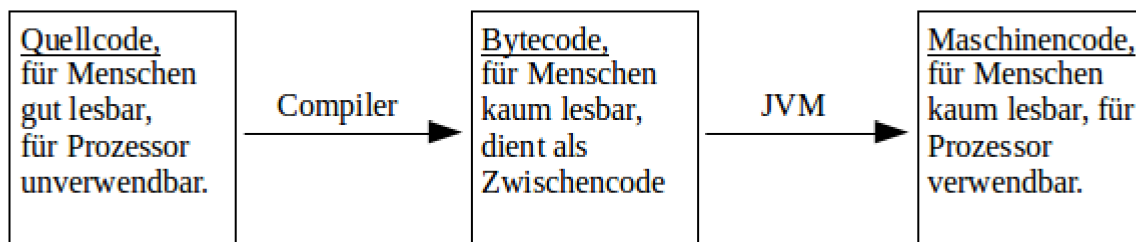


Abbildung 2.2: Codeumwandlung bei Java

Hat man ein Programm z.B. in der Programmiersprache C geschrieben und möchte es sowohl auf einem Rechner mit dem Betriebssystem Windows als auch auf einem Apple Mac verwenden, so muss man den Quellcode mit zwei verschiedenen Compilern kompilieren, um ein Programm in Maschinensprache für Windows und ein Programm für Apple Mac zu erhalten. Java vermeidet dieses Problem mit Hilfe eines Zwischencodes, den man „Bytecode“ nennt. Ein in Java geschriebenes Programm wird durch einen Compiler in Bytecode überführt. Ein Computer, der die „Java Virtual Machine“, JVM, installiert hat, setzt diesen Bytecode dann in Maschinensprache für den betreffenden Computer um. Die Anpassung für verschiedene Computer erfolgt durch die JVM. Da die JVM heute auf fast allen Computern zur Standardsoftware gehört, kann also Java-Bytecode praktisch auf allen Rechnern ausgeführt werden. Java-Quellcode wird in Dateien mit der Endung „.java“ abgespeichert, Bytecodedateien haben die Endung „.class“.

Aufgabe 2.1 *Quellcode, Bytecode, Maschinencode*

1. *Sie sind Programmierer. Ein Kunde hat ein Programm für übliche Rechner mit dem Betriebssystem Windows bei Ihnen in Auftrag gegeben. Sie haben in C programmiert. Was bekommt der Kunde von Ihnen: Quellcode oder Maschinencode?*

2. *Sie haben ein Java-Programm geschrieben und wollen es verkaufen. Was verkaufen Sie: Quellcode, Bytecode oder Maschinencode?*

3. *Java wird gerne verwendet, um kleine Animationen auf Webseiten zu programmieren. Warum ist Java dafür gut geeignet? Hier ein Beispiel:* `http://staff.fim.uni-passau.de/~kreuzer/Spielesammlung/Zauberwuerfel/applets/RubiksCube.html`

4. *Ein Freund soll morgen bei Herrn Jost ein Java-Programm abgeben, das er aber einfach nicht zum Laufen bringt. Sie möchten ihm helfen. Was muss er Ihnen schicken: Quellcode, Bytecode oder Maschinencode?*

5. *Ein gutes Argument für die Verwendung von Java ist die Plattformunabhängigkeit, also die Tatsache, dass Java-Bytecode auf allen Rechnern mit JVM läuft. Welche Nachteile könnten sich daraus aber ergeben?*

1.3 Entwicklungsumgebungen

Wenn Sie ein Java-Programm erstellen wollen, dann müssen Sie das Programm zunächst schreiben. Dafür brauchen Sie einen Editor, eine Art einfaches Textverarbeitungsprogramm. Der Quellcode wird in einer Datei abgespeichert. Aus dieser wird dann durch Aufruf des Compilers Bytecode erstellt. Dann wird das Programm ausprobiert und dann in aller Regel nachgebessert. Das ständige Arbeiten mit den drei Programmen Editor, Compiler und JVM ist zeitaufwändig. Daher benutzt man zum Programmieren „integrierte Entwicklungsumgebungen“ kurz „IDE“ von englisch „Integrated Development Environment“, die einen Editor beinhalten und aus denen der Compiler und die JVM aufgerufen werden können. Man arbeitet also nur noch mit der IDE. Eine sehr beliebte IDE für Java ist Eclipse, die wir auch verwenden werden. Ferner wird auch oft NetBeans verwendet. Für Lerner werden manchmal auch die überschaubareren IDEs BlueJ oder JavaEditor empfohlen.

Die Entwicklungsumgebung Eclipse hat die Eigenschaft, dass sie bereits bei der Eingabe des Quellcodes versucht, diesen zu kompilieren. Man muss also nicht selbst die Kompilierung starten. Der Vorteil ist, dass ein „Compilerfehler“, das heißt Code, der nicht kompiliert werden kann, sofort gemeldet wird.

Aufgabe 2.2 *Installation der Programmierwerkzeuge*

1. *Installieren Sie das Java Development Kit, JDK, auf Ihrem privaten Computer.* `http://www.oracle.com/technetwork/java/javase/downloads/index.html`

2. *Installieren Sie Eclipse auf Ihrem privaten Computer.* `http://www.eclipse.org/` *Links zu den Downloadseiten finden Sie auch auf den Seiten des Studienkollegs unter „Lernmaterialien“ oder in Moodle. Diese Programme dürfen i.A. kostenlos genutzt werden, beachten Sie für die Details aber die Lizenzbedingungen.*

2 Erste Programme in Java

2.1 Wie beginnt ein Java-Programm?

Ein Java-Programm besteht aus einer oder mehreren Quellcode-Dateien. Jede solche Quellcode-Datei beginnt üblicherweise mit „`public class`“, gefolgt von einem Namen. Eine Datei könnte also so beginnen: „`public class MeinErstesProgramm`“. Der Name darf keine Leerzeichen beinhalten. Soll der Name dennoch, wie im Beispiel, aus mehreren Worten bestehen, so kennzeichnet man die Wortanfänge durch Großbuchstaben. Der erste Buchstaben sollte ein Großbuchstaben sein. Nach dieser „Überschrift“' folgt eine öffnende geschweifte Klammer, also „{“. Ganz am Ende der Datei steht dann die schließende geschweifte Klammer „}“.

Der Eintstiegspunkt eines Java-Programms liegt in der main()-Methode, die durch die Zeile `public static void main(String[] args)` eingeleitet wird. Diese Zeile kommt also in jedem Programm gewöhnlich einmal vor. Was diese Zeile genau bedeutet, wird später erklärt und soll erst einmal so auswendig gelernt werden. Nach dieser Zeile folgt wiederum eine öffnende geschweifte Klammer, die am Ende der Methode geschlossen wird.

Wichtig! Nach jeder öffnenden geschweiften Klammer wird der weitere Programmtext um vier Stellen nach rechts eingerückt. Die Einrückung endet nach dem Schließen der geschweiften Klammer. Eclipse nimmt diese Einrückungen normalerweise automatisch vor. Sollte dies aus irgendwelchen Gründen einmal nicht funktionieren, so kann man in Eclipse den Befehl „Source → Correct Indentation“ aufrufen.

Mehrere Dateien eines Programms werden in Eclipse zu einem „Projekt“ zusammengefasst. Möchten Sie in Eclipse also ein neues Programm beginnen, so müssen Sie unter „File → new → Java Project“ ein neues Projekt erstellen und in diesem eine Datei, indem Sie den Button „New Java Class“ betätigen. Das Verzeichnis, in dem Eclipse die Projekte abspeichert, wird „Workspace“ genannt (siehe Abbildung 2.3).

2.2 Ausgabeanweisungen

Das folgende Programm schreibt einen kurzen Text („Hallo Welt“) in ein Fenster, die „Konsole“.

Listing 2.1: class MeinErstesProgramm

```
1 public class MeinErstesProgramm{
2     public static void main(String[] args){
3         System.out.println("Hallo Welt");
4     }
5 }
```

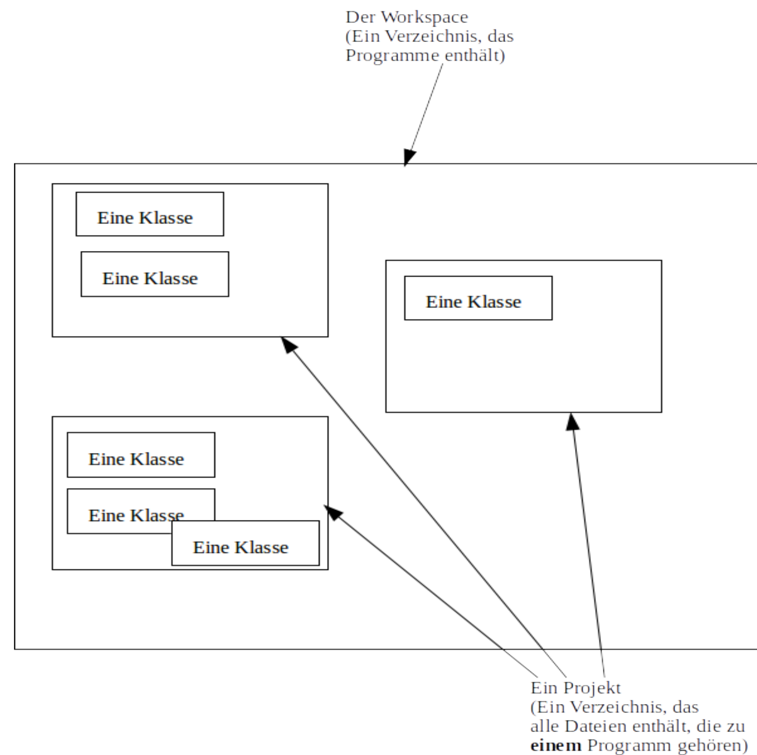


Abbildung 2.3: Verzeichnisstruktur beim Erstellen von Java-Programmen mit Eclipse

Die dritte Zeile stellt die Ausgabeanweisung dar. Experimentieren Sie in den folgenden Aufgaben mit der Ausgabeanweisung. Dabei bleiben die ersten beiden Zeilen und die letzten beiden Zeilen (mit den „}“) immer gleich.

Aufgabe 2.3 *Ausgabeanweisung*

1. *Schreiben Sie folgende Methoden und probieren Sie sie aus:*

```
1 public static void main(String[] args){
2       System.out.println ("1. Zeile");
3       System.out.println ("2. Zeile");
4       System.out.println ("3. Zeile");
5       System.out.println ("4. Zeile");
6 }
```

und

```
1 public static void main(String[] args){
2       System.out.print ("1. Zeile");
3       System.out.print ("2. Zeile");
```

```
4       System.out.print ("3. Zeile");
5       System.out.print ("4. Zeile");
6   }
```

Was ist der Unterschied zwischen „println“ und „print“?

2. *Wie wird der Ausdruck „2 * 2 + 3 “ in den folgenden Methoden ausgewertet?*

```
1   public static void main(String[] args){
2       System.out.println ("2*2+3");
3   }
```

und

```
1   public static void main(String[] args){
2       System.out.println (2*2+3);
3   }
```

3. *Schreiben Sie folgenden Methoden und probieren Sie sie aus:*

```
1   public static void main(String[] args){
2       System.out.println ("2*2+3 ist " +(2*2+3) + "Stimmt's?"
            );
3   }
```

Welche Rolle spielt das Zeichen „+“? Was passiert, wenn man bei „`+(2*2+3)`“ die Klammern weg lässt?

Die Verarbeitung von Textbausteinen, „Strings“ genannt, benötigt man oft und funktioniert nicht nur in Zusammenhang mit „System.out.println()“!

Aufgabe 2.4 *Klammern*
In der Programmierung mit Java kommen die Zeichen { } ( ) [ ] vor. Diese haben eine wichtige Bedeutung. Lernen Sie die folgenden Vokabeln:

„{ }“ heißen „geschweifte Klammern“. Das Zeichen „{“ wird gelesen als „geschweifte Klammer auf“, das Zeichen „}“ wird gelesen als „geschweifte Klammer zu“.

„()“ heißen „runde Klammern“. Das Zeichen „(“ wird gelesen als „runde Klammer auf“, das Zeichen „)“ wird gelesen als „runde Klammer zu“.

„[]“ heißen „eckige Klammern“. Das Zeichen „[“ wird gelesen als „eckige Klammer auf“, das Zeichen „]“ wird gelesen als „eckige Klammer zu“.

Wird einfach nur „Klammer“ gesagt, ist normalerweise eine runde Klammer gemeint.

2.3 Variablen, Datentypen, Zuweisungsoperator

Betrachen Sie das folgende kurze Programm. Tippen Sie es ab. Probieren Sie es aus. Was macht das Programm?

Listing 2.2: class BMI Version 1

```
1 public class BMI {
2     public static void main(String[] args) {
3             double groesse = 1.72;
4             double idealmasse = 23*groesse*groesse;
5             System.out.println("Sie sind "+groesse +" gross.");
6             System.out.println("Ihre Idealmasse betraegt " +
                  idealmasse +" kg.");
7     }
8 }
```

In diesem Programm (Listing 2.2) kommen zwei Variablen vor: „groesse“ und „idealmasse“. Bei der Einführung einer Variablen muss in Java der Datentyp der Variable angegeben werden. Beide Variablen in diesem Beispiel sind vom Datentyp „double“. Der Datentyp „double“ gibt an, dass die Variablen Werte einer Zahl mit Dezimalpunkt speichern. Beachten Sie, dass in Java als Dezimaltrenner der Punkt, nicht das Komma, dient. Diese Einführung einer Variablen durch Angabe des Datentyps und des Namens der Variablen nennt man „Deklaration“ der Variablen. Der Bezeichner (Name) einer Variablen beginnt üblicherweise mit einem Kleinbuchstaben.

Nach der Deklaration folgt das Zeichen „=“. Dieses Zeichen ist der Zuweisungsoperator. Er sagt, dass die Variable, die links davon steht, den Wert des Ausdrucks rechts davon bekommt. Die erste Zuweisung eines Werts an eine Variable nennt man „Initialisierung“ der Variablen.

Aufgabe 2.5 *Variablen*

1. *In einer Klasse befinden sich die Variablendeklarationen und Variableninitialisierungen* `double x = 2.0` *und* `double y = 3.0`.

   a) Welche Werte speichern die Variablen nach der Anweisung `x = y`*?*

   b) Welche Werte speichern die Variablen, wenn statt `x = y` *die Anweisung* `y = x` *erfolgt?*

2. *Deklarieren Sie in der Klasse „BMI“ eine weitere Variable vom Typ „double“ mit dem Bezeichner „masse“ und initialisieren Sie diese mit einem vernünftigen Wert.*

3. *Ändern Sie das Programm so, dass auch der Wert der Variablen „masse“, der die tatsächliche Masse [1] darstellen soll, ausgegeben wird.*

---

[1] Beachten Sie, dass man bei der Körpermasse in der deutschen Umgangssprache meistens von „Gewicht“ spricht. In der Physik bedeutet „Gewicht“ aber dasselbe wie „Gewichtskraft“. Die Gewichtskraft

2.4 Methoden ohne Rückgabe

Bisher ist das Programm eine Aneinanderreihung von Anweisungen. Bei komplizierten Programmen wird das sehr unübersichtlich. Daher kann man mehrere Anweisungen zu einer „Methode“ zusammenfassen.

Die Klasse „BMI“ besitzt in der Fassung nach den Aufgaben 2.5 drei Ausgabeanweisungen hintereinander. Deklariert man eine Methode „`static void ausgeben()`“, dann kann man diese drei Anweisungen, durch den Methodenaufruf „`ausgeben()`“ ersetzen. Die drei Ausgabeanweisungen finden sich statt dessen in der Deklaration der Methode.

Ein erster Versuch, der aber so nicht kompilierbar ist, könnte so lauten:

Listing 2.3: class BMI Version 2, fehlerhaft

```
1  public class BMI {
2
3      public static void main(String[] args){
4          double groesse = 1.72;
5          double idealmasse = 23*groesse*groesse;
6          double masse = 72;
7          ausgeben();
8      }
9
10     static void ausgeben(){
11         System.out.println("Sie sind " + groesse +" gross.");
12         System.out.println("Ihre Idealmasse betraegt " +
               idealmasse +" kg.");
13         System.out.println("Ihre tatsaechliche Masse betraegt "
               +masse +" kg.");
14     }
15 }
```

Die erste Zeile einer Methodendeklaration, hier `static void ausgeben()` nennt man die „erweiterte Methodensignatur“, manchmal auch kurz und etwas ungenau einfach „Methodensignatur“. An diesem Beispiel kann man erkennen, dass es sich bei der Zeile `public static void main(String[] args)` ebenfalls um eine erweiterte Methodensignatur handelt, die zusammen mit den weiteren Zeilen bis zum nächsten „}“ eine Methodendeklaration bildet. Es wird eine Methode mit dem Bezeichner (Namen) „main“ deklariert.

Halten wir fest: Wenn ein Java-Programm läuft, dann werden Methoden abgearbeitet. Eine Methode kann andere Methoden aufrufen. Jedes Java-Programm wird gestartet durch den Aufruf der Methode mit der erweiterten Signatur `public static void main(String[] args)`.

---

wird in der Einheit N (Newton) gemessen und nicht wie die Masse in kg. Um die Begriffe wie im Physikunterricht zu verwenden, ist hier meistens von der „Masse“ die Rede.

Warum aber läuft das Programm (Listing 2.3) nicht in seiner jetzigen Fassung? Das Problem ist, dass die Variablen „groesse“, „idealmasse“ und „masse“ in der main()-Methode deklariert werden und daher nur dort bekannt sind. Ihre Verwendung in der Methode „ausgeben()“ ist nicht möglich. Man sagt, diese Variablen sind „lokale Variablen“.

Eine Verbesserung könnte darin bestehen, dass die Deklarationen der Variablen vor die Methoden geschrieben werden. Dadurch werden die Variablen zu „globalen Variablen“, die in der gesamten Klasse verwendet werden können. Das Programm sieht dann so aus:

Listing 2.4: class BMI Version 3

```
1  public class BMI{
2      static double groesse;
3      static double idealmasse;
4      static double masse;
5
6      public static void main(String[] args){
7          groesse = 1.72;
8          idealmasse = 23*groesse*groesse;
9          masse = 72;
10         ausgeben();
11     }
12
13     static void ausgeben(){
14         System.out.println("Sie sind " + groesse +" gross.");
15         System.out.println("Ihre Idealmasse betraegt " +
               idealmasse +" kg.");
16         System.out.println("Ihre tatsaechliche Masse betraegt "
               +masse +" kg.");
17     }
18 }
```

Nun sollte das Programm wieder kompilierbar und lauffähig sein. Bei der Deklaration der globalen Parameter benötigt man hier das Schlüsselwort „static“, das auch in den Methodendeklarationen vorkommt. Dessen Bedeutung wird später erklärt. Beachten Sie unbedingt, dass in der main()-Methode nun keine Deklaration mehr erfolgt, das heißt, vor den Variablen wird nicht deren Typ genannt. Würden Sie beispielsweise `double groesse = 1.72;` schreiben, dann gibt es danach zwei Variablen mit dem Bezeichner „groesse“, eine globale und eine lokale. Das kann zu seltsamen, schwer zu findenden Fehlern führen.

Aufgabe 2.6 *Methode erstellen*
Ergänzen Sie die Klasse „BMI“ um eine Methode „variablenInitialisieren()“. Die main()-Methode soll dann so lauten:

```
public static void main(String[] args){
    variablenInitialisieren();
    ausgeben();
}
```

Hinweis: Sie benötigen globale Variablen.

Greift man immer zu dieser Lösung, dann kann dies aber Probleme machen. Große Programme können dann damit beginnen, dass 20, 40 oder noch viel mehr globale Variablen deklariert werden. Dies verschlechtert die Lesbarkeit eines Programms, da die Deklaration und die Verwendung der Variablen stark getrennt wird. Daher ist eine andere Verbesserung des fehlerhaften Programms günstiger:

Listing 2.5: class BMI Version 4

```
public class BMI {

        public static void main(String[] args) {
                double groesse = 1.72;
                double idealmasse = 23*groesse*groesse;
                double masse = 72;
                ausgeben(groesse, idealmasse, masse);
        }

        static void ausgeben(double dieGroesse, double
            dieIdealmasse, double dieMasse){
                System.out.println("Sie sind " + dieGroesse +"
                    gross.");
                System.out.println("Ihre Idealmasse betraegt " +
                    dieIdealmasse +" kg.");
                System.out.println("Ihre tatsaechliche Masse
                    betraegt " +dieMasse +" kg.");
        }
}
```

Aufgabe 2.7 *Einige Fachbegriffe*
In diesem Abschnitt wurden sehr viele Fachbegriffe eingeführt. Sie müssen diese beherrschen, wenn Sie das weitere Skript und den weiteren Unterricht verstehen möchten. Überlegen Sie sich, ob Sie die folgenden Begriffe definieren können, und lesen Sie, wenn notwendig, noch einmal nach: Methode, Methodendeklaration, Methodenaufruf, erweiterte Signatur einer Methode, main()-Methode, lokale Variable, globale Variable, Parameter einer Methode, formaler Parameter, aktueller Parameter.

2.5 Kommentare

Quellcode muss verständlich sein, auch für einen Programmierer, der den Code nicht selbst geschrieben hat. Das ist Alltag in jeder Software-Firma. Daher müssen Quelltextdateien kommentiert werden. Ein Kommentar ist ein Text, der vom Compiler nicht ausgewertet wird und nur zur Erklärung dient.

Kommentare können eingeleitet werden durch zwei Schrägstriche, also //. Dann geht der Kommentar bis ans Ende dieser Zeile. Ab der nächsten Zeile geht der normale Programmcode weiter, der vom Compiler ausgewertet wird.

Längere Kommentare werden durch /* eingeleitet und */ beendet. Eine Sonderform ist der Javadoc-Kommentar. Er wird eingeleitet durch /** und beendet durch */. Aus javadoc-Kommentar kann automatisch ein html-Dokument (wie oft bei www-Seiten verwendet) erzeugt werden. Beispiele können Sie unter docs.oracle.com/javase/8/docs/api finden. Dort finden Sie Dokumentationen für die Bibliotheksklassen. Das sind Programmbausteine, die man in die eigenen Programme einbauen kann.

Wenn Sie programmieren, dann sollten Sie folgendermaßen kommentieren:

- Jede Klasse bekommt einen Javadoc-Kommentar, der in einem oder zwei Sätzen beschreibt, was die Klasse macht. Dieser Kommentar soll außerdem den „Tag"(englisch: Etikett) „@author", gefolgt vom Autorennamen, und den Tag „@version", gefolgt von einer Versionsnummer und/oder dem Datum der Klassenversion enthalten.
- Jede Methode erhält einen Javadoc-Kommentar, der in ein oder zwei Sätzen erklärt, was die Methode macht. Jeder Parameter der Methode wird durch einen „@param"-Tag erläutert.
- Variablen können, wenn nötig, durch einige Stichworte erläutert werden.
- besonders komplizierte Programmteile können zusätzlich kommentiert werden.

Das Programm aus Listing 2.5 könnte so kommentiert werden:

Listing 2.6: class BMI Version 5 (wie Version 4, aber kommentiert

```
1  /**
2   * @author Christof Jost
3   * @version 2015-09-08
4   * In der Klasse werden Groesse und Masse durch eine
5   * Variableninitialisierung gegeben. Fuer die Groesse
6   * wird die ideale Masse ausgerechnet und die
7   * gegebenen und berechneten Werte ausgegeben.
8   */
9  public class BMI {
10
```

```
11        /**
12         * Die main-Methode initialisiert die Variablen und
13         * macht eine Ausgabe
14         */
15        public static void main(String[] args) {
16                double groesse = 1.72;
17                double idealmasse = 23*groesse*groesse;
18                double masse = 72;
19                ausgeben(groesse, idealmasse, masse);
20        }
21
22        /**
23         * Die Methode organisiert die Ausgabe.
24         * @param dieGroesse die Koerpergroesse in m
25         * @param dieIdealmasse die Idealmasse in kg
26         * @param dieMasse die tatsaechliche Masse in kg
27         */
28        static void ausgeben(double dieGroesse, double
             dieIdealmasse, double dieMasse){
29                System.out.println("Sie sind " + dieGroesse +"
                     gross.");
30                System.out.println("Ihre Idealmasse betraegt " +
                     dieIdealmasse +" kg.");
31                System.out.println("Ihre tatsaechliche Masse
                     betraegt " +dieMasse +" kg.");
32        }
33 }
```

Aufgabe 2.8 *Kommentare*
Kommentieren Sie die Methode „variablenInitialisieren()“ aus Aufgabe 2.6.

3 Objektorientierte Programmierung

3.1 Einführung

Computerprogramme werden gemäß eines „Programmierparadigmas“ geschrieben, das heißt, sie folgen einer bestimmten Grundidee des Programmierstils. Beispiele für Programmierparadigmen sind die imperative Programmierung und die objektorientierte Programmierung. In der imperativen Programmierung wird ein Programm als eine Folge von

Anweisungen aufgefasst. Wenn dabei einige Anweisungen zu einer „Prozedur“ oder „Methode“ zusammengefasst werden, bleibt das dennoch „imperative Programmierung“. Das, was Sie bisher gemacht haben, war imperative Programmierung.

Das heute wichtigste Programmierparadigma ist die objektorientierte Programmierung. Programmiersprachen, die Konstrukte anbieten, welche die objektorientierte Programmierung besonders gut unterstützen, nennt man „objektorientierte Programmiersprachen“. Java ist eine objektorientierte Programmiersprache. Bei der objektorientierten Programmierung erzeugt man „Objekte“. Ein Objekt hat bestimmte Eigenschaften und kann bestimmte Dinge machen. In den meisten objektorientierten Programmiersprachen, auch in Java, werden Objekte erzeugt, indem man einen „Bauplan“ programmiert und davon Objekte, auch „Instanzen“ genannt, erzeugt. In Java ist ein solcher „Bauplan“ eine Klasse.

In Java ist die objektorientierte Programmierung der Normalfall. Programmiert man globale Variablen oder Methoden, die sich **nicht** auf Objekte der Klasse beziehen, sondern auf die Klasse selbst, dann muss man diese mit dem Wort „static“ kennzeichnen. „static“ benötigt man also insbesondere, wenn man mit Java nicht objektorientiert, sondern imperativ programmiert, wie das bisher der Fall war.

Aufgabe 2.9 *„static“*
Wenn man mit Java objektorientiert programmiert, braucht man „static“ fast nie, außer einmal. An welcher Stelle?

Ein Vorteil der objektorientierten Programmierung ist ein relativ übersichtlicher Code. Besonders wichtig ist, dass man die Klassen (also Baupläne), die jemand anderes programmiert hat, in den eigenen Programmen verwenden kann. Man muss die Klasse nicht bis ins Detail verstehen, um sie verwenden zu können. Damit können große Programme von mehreren Programmierern gemeinsam programmiert werden.

3.2 Aufbau einer Java-Klasse

Eine Java-Klasse bei der objektorientierten Programmierung besteht normalerweise aus drei Teilen:

1. Attribute, auch Instanzvariablen genannt. Attribute sind die globalen Variablen der objektorientierten Programmierung. Als Attribute werden die Teile und Größen programmiert, die die Objekte der Klasse **haben** sollen.

   Beispiel: Sie möchten eine Klasse „Rechteck“ programmieren, die der Bauplan für Rechteck-Objekte sein soll, die auf dem Bildschirm dargestellt werden können. Ein Rechteck **hat** eine Breite, also ist es sinnvoll, in der Klasse „Rechteck“ ein Attribut „breite“ zu programmieren.

   Soll das Rechteck auf einem Bildschirm dargestellt werden, wird die Breite am besten in Pixel gemessen. Dann sollte das Attribut den Typ „int“ besitzen. „int“ ist der in Java üblichste Typ für eine ganze Zahl.

2. Konstruktoren. Ein Konstruktor ist eine spezielle Methode. Der Konstruktor legt fest, mit welchen Eigenschaften ein Objekt sein „Leben“ beginnt. Ein Konstruktor hat denselben Namen wie die Klasse. Das Schlüsselwort „void“ fehlt. Der Aufruf eines Konstruktors erfolgt nicht über den Namen der Methode, sondern mit dem Schlüsselwort „new“, gefolgt vom Namen.

   Beispiel: Eine Klasse „Rechteck“ besitzt einen Konstruktor mit der erweiterten Signatur `public Rechteck()`. Der Konstruktoraufruf lautet `new Rechteck()`. Der Konstruktor liefert daraufhin ein Objekt der Klasse. Um mit diesem Objekt arbeiten zu können, muss man ihm einen Namen geben, das heißt eine Variable einführen, die auf dieses Objekt zeigt. Der Typ dieser Variable ist der Name der Klasse. Insgesamt sieht eine Programmzeile, in der ein Objekt erzeugt und einer Variablen zugewiesen wird, beispielsweise so aus: `Rechteck meinRechteck = new Rechteck()`.

   Programmiert man in einer Klasse keinen Konstruktor, dann kann man dennoch Objekte der Klasse erzeugen. Java fügt automatisch einen Standardkonstruktor hinzu. Die Attribute haben dann Standardwerte, beispielsweise „0“ bei Zahlen.

3. Danach folgen die übrigen Methoden. Sie beschreiben, was ein Objekt der Klasse machen kann. Beispielsweise könnte ein Rechteck seine Breite ändern. Da es in der objektorientierten Programmierung mehrere Objekte geben kann, die nach dem selben Bauplan (also Klasse) erstellt wurden, muss gesagt werden, welches Objekt diese Methode ausführen soll. Dies geschieht durch Nennung einer Variablen, die auf dieses Objekt verweist, gefolgt von einem Punkt und dem Namen der Methode. Beispiel: `meinRechteck.setBreite(130)`.

Aufgabe 2.10 *Attribute, Konstruktoren, Methoden*
Gegeben ist die folgende Klasse „Rechteck“:

```
1  import java.awt.Rectangle;
2  //eine Klasse aus der Bibliothek wird verwendet
3
4  /**
5   * @author Christof Jost
6   * @version 2015-09-10
7   * erzeugt ein Rechteck zur Darstellung auf einem Zeichenbrett
8   */
9
10 public class Rechteck {
11
12         int breite;
13         int hoehe;
14         int xPosition;
```

```
15        int yPosition;
16        String farbe;
17        Zeichenbrett zeichenbrett;
18        //auf diesem Zeichenbrett wird das Rechteck dargestellt
19
20        public Rechteck(Zeichenbrett dasZeichenbrett) {
21            breite = 100;
22            hoehe = 200;
23            xPosition = 200;
24            yPosition = 200;
25            farbe = "lila";
26            zeichenbrett = dasZeichenbrett;
27            zeichnen();
28        }
29
30        void zeichnen(){
31            zeichenbrett.zeichne(this, farbe, new Rectangle
                  (xPosition, yPosition,breite, hoehe));
32        }
33
34        void setBreite(int neueBreite){
35            breite = neueBreite;
36            zeichnen();
37        }
38 }
```

1. *Wie viele Attribute besitzt die Klasse? Wie heißen diese?*
2. *In welcher Zeile beginnt der Konstruktor, in welcher Zeile endet er?*
3. *Wie viele Methoden besitzt die Klasse? Wie heißen diese?*

3.3 Java-Konventionen

Wenn Sie mit Java programmieren, dann müssen Sie bestimmte Konventionen einhalten. Die Einhaltung der Konventionen ist nicht notwendig für das Funktionieren des Programms. Das Einhalten der Konventionen ist aber wichtig, damit andere Programmierer Ihre Programme verstehen können und Sie die die Programme anderer Programmierer verstehen können. Halten Sie sich an folgende Konventionen:

- Einrückungen: Mit jeder öffnenden geschweiften Klammer „{“ wird um weitere vier Stellen eingerückt. Mit jeder schließenden geschweiften Klammer „}“ wird die Einrückung um vier Stellen vermindert. In Eclipse wird dies im Normalfall automatisch

erreicht. Ansonsten markieren Sie die Stelle mit falschen Einrückungen und wählen Sie in Eclipse „Source → Correct Indentation“.

- Klassennamen sind ein Nomen im Singular und beginnen mit einem Großbuchstaben.
- Eine Klasse beginnt mit den Attributen, gefolgt von den Konstruktoren, schließlich kommen die Methoden.
- Namen von Attributen und anderen Variablen beginnen mit einem Kleinbuchstaben
- Namen von Methoden beginnen mit einem Kleinbuchstaben. Soweit sinnvoll, sind Methodennamen Verben im Infinitiv.
- Sie sollten lange, sprechende Namen benutzen. Bestehen Namen aus mehreren Wörtern, dann wird jedes neue Wort mit einem Großbuchstaben begonnen (Leerzeichen führen zu einem Fehler!).

Aufgabe 2.11 *Klassen erstellen*

1. *Ergänzen Sie die Klasse “Rechteck“ aus Aufgabe 2.10 um Methoden zum Setzen der anderen Attribute. Nehmen Sie sich die Methode `setBreite()` zum Vorbild.*
2. *Lassen Sie sich die Klassen „Kreis“, „Dreieck“ und „Zeichenbrett“ geben und kopieren Sie diese zusammen mit Ihrer Klasse „Rechteck“ in dasselbe Projekt. Erstellen Sie eine Klasse „Main“ mit einer main()-Methode und programmieren Sie die main()-Methode so, dass ein Bild gemalt wird. Das Bild könnte z.B. ein Haus mit Fenstern und Tür und einem Kamin zeigen, dazu die Sonne. Sie müssen dafür gegebenenfalls in den Klassen „Kreis“ und „Dreieck“ weitere Methoden programmieren.*
3. *Erstellen Sie in dem Projekt eine neue Klasse mit dem Namen „Bild“. Fügen Sie der Klasse eine Methode `void malen()` hinzu, die ein Bild malt. Sie können das Bild aus Teil 2 kopieren. Fügen Sie eine weitere Methode `void malen2()` hinzu, die ein anderes Bild malt, z.B. mit anderen Farben. Ändern Sie die main()-Methode ab: Die main()-Methode soll nicht mehr die einzelnen Anweisungen für das Malen eines Bilds enthalten, sondern ein Objekt der Klasse „Bild“ erzeugen und darauf die Methoden „malen()“ und „malen2()“ aufrufen.*

3.4 Typen

In Java gibt es zwei Arten von Typen: „Primitive“ Typen, die in der Sprache angelegt sind, und „Objekttypen“, die programmiert wurden. Bisher wurden zwei primitive Typen benutzt: „double“ für Bruchzahlen und „int“ für ganze Zahlen. Insgesamt gibt es in Java

acht primitive Typen. Ein Objekttyp ist beispielsweise der Typ „Zeichenbrett“ in Aufgabe 2.10. Er existiert, weil eine Klasse „Zeichenbrett“ programmiert wurde. Es gibt daher beliebig viele Objekttypen.

In Java gibt es vier primitive Typen für ganze Zahlen:

- `byte` für ganze Zahlen zwischen –128 und +127
- `short` für ganze Zahlen zwischen –32768 und +32767
- `int` für ganze Zahlen zwischen –2147483648 und +2147483647
- `long` für ganze Zahlen zwischen –9223372036854775808 und +9223372036854775807

`byte` und `short` werden normalerweise nicht auf „echten“ Computern verwendet, sondern nur auf Kleingeräten, bei denen der Speicherplatz knapp ist. Meistens benutzt man `int`. `long` benutzt man, wenn der Zahlenbereich von `int` zu klein ist.

Es gibt zwei primitive Typen für Bruchzahlen:

- `float` für positive und negative Zahlen mit Beträgen bis $3,4 \cdot 10^{38}$
- `double` für positive und negative Zahlen mit Beträgen bis $1,7 \cdot 10^{308}$

`double` deckt nicht nur einen größeren Bereich ab, sondern teilt in kleinere Stufen ein, ist also genauer. Allerdings verbraucht eine Zahl vom Typ `double` doppelt soviel Speicherplatz wie eine Variable vom Typ `float`. Bei Programmen für PCs wird praktisch immer `double` verwendet.

Weitere primitive Typen sind:

- `boolean` für die Speicherung der Werte `true` (wahr) und `false` (falsch)
- `char` für die Speicherung eines Zeichens, z.B. eines Buchstabens

Ein wichtiger Objekttyp ist der Typ `String`. Eine Variable vom Typ `String` verweist auf ein Textobjekt, also eine Zeichenkette. Dieser Typ hat eine Sonderrolle: Er kann jederzeit benutzt werden, ohne dass die Bibliotheksklasse „String“ importiert werden muss oder im Projekt explizit vorhanden sein muss. Er fühlt sich daher wie ein primitiver Typ an.

3.5 Methoden mit Rückgabe

Die bisher betrachteten Methoden waren im Prinzip Aufforderungen an ein Objekt, etwas zu tun. Zum Beispiel in Aufgabe 2.10 die Methode

```
void setBreite(int neueBreite){
        breite = neueBreite;
        zeichnen();
}
```

Eine solche Methode, die nur oder fast nur die Aufgabe hat, den Wert eines Attributs zu setzen, nennt man „Setter-Methode“ oder kurz „Setter“ von englisch „to set“, das bedeutet „setzen“.

Andere Methoden sind keine Aufforderungen an ein Objekt etwas zu tun, sondern Fragen an ein Objekt, auf die eine Antwort erwartet wird. Beispielsweise könnte man ein Objekt der Klasse „Rechteck“ aus der Aufgabe 2.10 fragen: „Welche Breite hast du?“ Man erwartet dann eine Antwort vom Typ „int“, da die Breite von diesem Typ ist. Die Methode könnte so aussehen:

```
int getBreite(){
        return breite;
}
```

„int“ gibt dabei an, das die Antwort der Methode von diesem Typ sein wird. Bei Methoden, die keine Antwort liefern, wie wir sie bisher hatten, steht an dieser Stelle „void“, das bedeutet „leer“. Die Antwort wird durch die Anweisung „return“ gegeben. Eine Methode mit Rückgabe **muss** eine return-Anweisung besitzen. Nach der return-Anweisung ist die Methode zu Ende. Der Aufruf einer Methode mit Rückgabe erfolgt normalerweise im Rahmen einer Zuweisung, da die Antwort gespeichert werden soll, also z.B. `int breiteDesRechtecks = meinRechteck.getBreite()`

Eine solche Methode wie im vorigen Beispiel, die den Wert eines Attributs zurück gibt, nennt man „Getter-Methode“ oder kurz „Getter“ von englisch „to get“, das bedeutet „bekommen“, da man bei Aufruf dieser Methode den Wert des Attributs bekommt.

In der folgenden Aufgabe geht es um die Eingabe mit einer vorgefertigten graphischen Benutzeroberfläche, „Textfenster“ genannt. Eine graphische Benutzeroberfläche nennt man auch kurz „GUI“ von englisch „graphical user interface“. Dabei spielen Methoden mit Rückgabe eine Rolle. Diese Benutzeroberfläche wird im Rahmen dieses Kurses noch oft gebraucht. Die Programmierung eigener GUIs in Java soll nicht verfolgt werden, da dieses Wissen vermutlich nicht von allen Studierenden gebraucht wird, da die graphische Ausgabe in anderen Programmiersystemen als Java ganz anders funktionieren kann. Sie können aber auch hier einen Vorteil der objektorientierten Programmierung erkennen: Auch wenn Sie nicht alle Konstrukte im Quellcode der Klasse „Textfenster“ verstehen, werden Sie die Klasse benutzen können.

Aufgabe 2.12 *Eingabe mit einer GUI*

1. *Laden Sie sich von der Seite* `http://www.stk.tu-darmstadt.de/tkurs_stk/lernmaterialient_stk/index.de.jsp` *oder aus der Moodle-Platform Ihres Kur-*

ses die Klasse „Textfenster“ herunter und kopieren Sie diese in ein neues Java-Projekt.

2. *Erstellen Sie im selben Java-Projekt die folgende Klasse:*

```
1 public class Main {
2     public static void main(String[] args) {
3             Textfenster meinFenster = new Textfenster();
4 
5         double eingabe = meinFenster.doubleEingeben("Bitte
              geben Sie eine Zahl ein.");
6         meinFenster.schreiben("Sie haben die Zahl " +
              eingabe +" eingegeben.");
7     }
8 }
```

Probieren Sie nun das Programm aus.

3. *Schreiben Sie nun das Programm „BMI“ (siehe z.B. Listing 2.6) neu. Die Größe und die Masse sollen aber jetzt nicht im Programm fest eingegeben sein, sondern mit einem „Textfenster“-Objekt eingegeben werden. Gehen Sie folgendermaßen vor: Erstellen Sie ein neues Projekt mit drei Klassen: „Textfenster“, „BMI“ und „Main“. Die Klasse BMI soll drei Attribute besitzen: Ein Attribut „fenster“ vom Typ „Textfenster“ und die Attribute „groesse“ und „masse“ vom Typ „double“. Fügen Sie zur Klasse „BMI“ eine oder mehrere Methoden hinzu, die die Eingabe, die Berechnung und die Ausgabe durchführen. Die Klasse „Main“ soll die main()-Methode enthalten. In dieser Methode soll ein Objekt der Klasse „BMI“ erzeugt werden. Auf diesem Objekt sollen die Methoden aufgerufen werden, damit der Benutzer die Größe und die Masse eingegeben kann und dann über Größe, Masse und die Idealmasse informiert wird.*

4. *Ihre Programme laufen bisher nur in der Entwicklungsumgebung „Eclipse“. Befreien Sie Ihre Programme! Beispiel: Das BMI-Programm aus Aufgabe 3. Wählen Sie File → Export. Im sich öffnenden Fenster wählen Sie → RunnableJARfile → Next. Unter „Launch Configuration“ geben Sie an, in welcher Klasse Ihres Programms sich die main()-Methode befindet, in diesem Beispiel also in der Klasse „Main“. Unter „Export Destination“ geben Sie an, wo das fertige Programm gespeichert werden soll. Probieren Sie dann aus, dass so erzeugte Programm durch einen Doppelklick zu starten.*

3.6 Zugriffmodifizierer

Programmiert man Klassen für ein großen Softwareprojekt, an dem viele Programmierer beteiligt sind, dann kann folgendes Problem auftreten: In einer verbesserten Version einer Klasse ändern Sie die Funktionsweise einer Methode. Nun hat aber ein anderer Programmierer, der Ihre Klasse verwendet, diese Methode benutzt und benötigt exakt die alte Funktionalität.

Um solche Probleme zu vermeiden, erlaubt man anderen Klassen nur auf bestimmte Methoden und Attribute zuzugreifen. Diese kennzeichnet man mit dem Schlüsselwort „public“. Diese Methoden und Attribute müssen ihre Funktionsweise und Bedeutung in allen Versionen behalten, damit die anderen Programmierer, die sie benutzen, sich darauf verlassen können.

Methoden und Attribute, die nur innerhalb der Klasse benutzt werden sollen und geändert werden können, werden statt dessen mit dem Schlüsselwort „private“ gekennzeichnet. Die Benutzung von außerhalb der Klasse ist dann nicht möglich und führt zu einer Fehlermeldung; „public“ und „private“ sind Zugriffsmodifizierer.

Bei großen Programmen werden die Klassen auf verschiedene Unterverzeichnisse, „Pakete“ genannt, verteilt. Gibt man keinen Zugriffsmodifizierer an, dann ist der Zugriff innerhalb desselben Pakets erlaubt. Man nennt dies den „Standardzugriffmodifizierer“. Bei unseren kleinen Programmen, bei denen es nur ein Paket gibt, ist daher das Weglassen des Zugriffsmodifizierers praktisch gleich, als ob überall „public“ stehen würde. Ein weiterer Zugriffmodifizierer lautet „protected“, der aber im Rahmen dieses Kurses nicht gebraucht wird.

Im Rahmen dieses Kurses sollten Sie sich folgenden Gebrauch der Zugriffsmodifizierer angewöhnen:

- Die main()-Methode muss „public“ sein.
- Setzen Sie alle Klassen auf „public“. Private Klassen sind sehr spezielle Konstrukte, die wir hier nicht behandeln.
- Setzen Sie Methoden, die nur eine „Codezusammenfassung“ sind, welche nur in derselben Klasse benutzt werden, auf „private“, die Methoden, die in anderen Klassen benutzt werden, auf „public“.
- In Lehrbüchern wird oft empfohlen, alle Attribute auf „private“ zu setzen und den Zugriff aus anderen Klassen gegebenenfalls mit Settern und Gettern, die „public“ sind, zu regeln. Im Rahmen der kleinen Programme, die wir hier schreiben, ist dies oftmals zu kompliziert. Setzen Sie daher die Zugriffsmodifizierer der Attribute wie bei den Methoden je nach Notwendigkeit „private“ oder „public“.

4 Kontrollstrukturen

4.1 Export und Import in Eclipse

Mittlerweile haben Sie Ihre ersten Programme geschrieben. Daher ist es an der Zeit, bevor es mit den Kontrollstrukturen losgeht, ein paar wichtige Funktionen in Eclipse zu zeigen.

Sie müssen Programme mit Ihrem USB-Stick nach Hause bringen und wieder mit bringen. Dafür haben Sie drei Möglichkeiten:

- Die umständliche Methode: Sie speichern alle Klassen in einem oder mehreren Textdokumenten. Am anderen Computer erstellen Sie ein neues Projekt. Darin erstellen Sie Klassen mit den Namen der Klassen des vorhanden Projekts. Danach kopieren Sie den Quelltext aus dem Textdokument in jede Klasse.

- Die schnellste, einfachste Methode: Benutzen Sie einen Ordner auf Ihrem USB-Stick als Eclipse-Workspace, dann haben Sie immer alles dabei. Vergessen Sie aber nicht, den Workspace von Zeit zu Zeit zu zu kopieren, weil man einen USB-Stick verlieren kann. Manchmal führt dieses Vorgehen zu Problemen, wenn die Java- und Eclipse-Installationen auf den verschiedenen Rechnern sehr unterschiedlich sind.

- Wenn die einfache Methode nicht funktioniert, dann ist die folgende Methode besser als die umständliche Methode: Klicken Sie File → Export → Java → JAR file → Next. Dann wählen Sie, welche Projekte Sie exportieren möchten, indem Sie die entsprechenden Auswahlboxen markieren. Bei den Auswahlboxen unten wählen Sie „Export Java source files and resources“ da sonst die Quelldateien, auf die es ankommt, nicht exportiert werden. Unter „Select the export destination“ wählen Sie Ordner und Name der zu speichernden Datei. Klicken Sie dann „Finish“.

  Um die Datei wieder zu importieren, legen Sie zuerst ein neuen Projekt an. Gehen Sie dort in das Verzeichnis „src“ für die Quellcode-Dateien. Klicken Sie jetzt File → Import → General → Archieve File → Next. Wählen Sie dann die vorher exportierte Datei aus und klicken Sie „Finish“.

Die Funktion „Export“, die eben beschrieben wurde, braucht man noch für eine viel wichtigere Aufgabe: Bisher laufen Ihre Programme immer unter der IDE „Eclipse“. Ihre Kunden, die Ihre Software kaufen, möchten aber nicht Eclipse installieren. Sie möchten, dass man Ihr Programm einfach klickt und es läuft. Außerdem möchten Sie nicht, dass Ihre Kunden in Eclipse immer Ihren Quellcode sehen.

4.2 if und if-else

Als Beispiel für dieses Konstrukt soll wieder das Programm zur Bestimmung des Body-Mass-Index herhalten, und zwar so wie in Aufgabe 2.12, Teilaufgabe 3 erstellt.

Ein solches Programm soll dem Benutzer nicht nur seine Größe, Masse, und Idealmasse mitteilen, sondern warnen, wenn die Person zu schwer ist. Der Hinweis auf zu große Masse könnte, wenn es Attribute „masse“ und „idealmasse“ gibt, beispielsweise durch folgende Methode erfolgen:

Listing 2.7: Beispiel für „if“

```
1 void warnenZuSchwer(){
2     if(masse > idealmasse) {
3         System.out.println("Ihre Masse ist " +(masse -
              idealmasse) + " kg zu gross.");
4     }
5 }
```

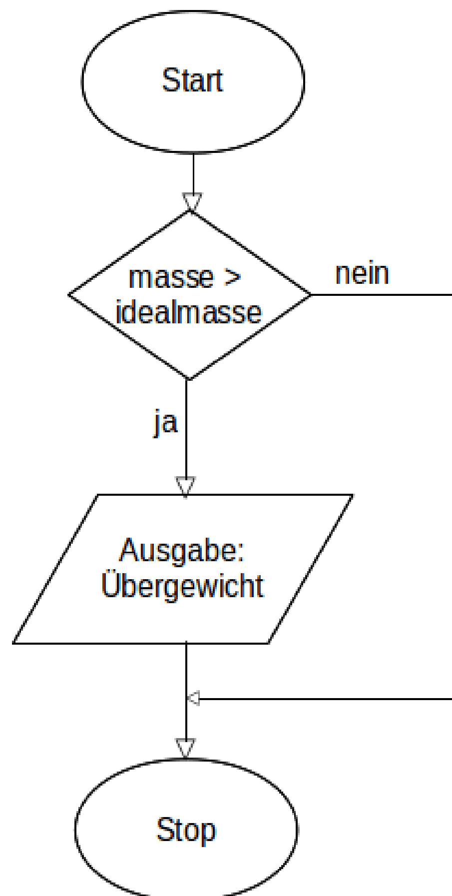


Abbildung 2.4: Ein Flussdiagramm mit „if“

Hinter „if“ (das hießt auf deutsch „falls“) steht immer eine runde Klammer. In dieser runden Klammer steht ein „Boolescher Ausdruck“, das heißt ein Ausdruck, der wahr (eng-

lisch: true) oder falsch (englisch: false) sein kann. Der Boolesche Ausdruck kann auch eine Boolesche Variable sein.

Methoden mit „if“ können kompliziert werden. Gerne stellt man die Algorithmen daher durch Flussdiagramme dar. Das Flussdiagramm würde hier aussehen wie in Abbildung 2.4. Details der Ausgabe wurden dabei nicht übernommen.

Möchte man auch eine Warnung ausgeben, wenn die Masse zu klein ist, dann könnte man eine zweite Bedingung mit „if“ verwenden:

Listing 2.8: Beispiel für eine Methode mit zwei„if“

```
1 public void warnen(){
2     if(masse > idealmasse) {
3         System.out.println("Ihre Masse ist " +(masse -
             idealmasse) + " kg zu gross.");
4     }
5     if(masse < idealmasse) {
6         System.out.println("Ihre Masse ist " +(idealmasse -
             masse) + " kg zu klein.");
7     }
8 }
```

Aufgabe 2.13 *if*

1. *Erweitern Sie die Methode so, dass wenn Masse und Idealmasse übereinstimmen die Ausgabe „Gratulation. Sie haben Ihre Idealmasse.“ erfolgt. Achtung! Sie testen auf Gleichheit mit dem Zeichen ==, denn das Zeichen = ist bereits für den Zuweisungsoperator vergeben.*

2. *Zeichnen Sie das Flussdiagramm für die Methode aus Teil 1.*

„if“ sorgt dafür, dass die Anweisungen in den geschweiften Klammern nur ausgeführt werden, wenn die Bedingung war ist. Manchmal benötigt man folgende Funktionalität: Wenn die Bedingung richtig ist, soll eine Anweisung ausgeführt werden, wenn die Bedingung nicht richtig ist, soll eine andere Anweisung ausgeführt werden. Dafür eignet sich das if-else-Konstrukt, für das Sie in Listing 2.9 ein Beispiel finden.

Das Zeichen % steht dabei für den Modulo-Operator. Er ist definiert für zwei Werte vom Typ „int“ und gibt den Divisionsrest. Beispiel: Dividiert man 23 durch 4, so erhält man 5 mit dem Rest 3. Der Ausdruck „23 % 4“ liefert daher den Wert „3“. Eine Zahl a ist durch eine Zahl b ohne Rest teilbar, wenn der Ausdruck „a % b == 0“ zu „true“ (wahr) ausgewertet wird.

Listing 2.9: Beispiel für eine Methode mit „if-else“

```
1 public void geradeOderUngerade(int zahl){
```

```
2       if(zahl % 2 == 0){
3           System.out.println("Die Zahl ist gerade.");
4       }
5       else{
6           System.out.println("Die Zahl ist ungerade.");
7       }
8   }
```

Das zugehörige Flussdiagramm sehen Sie in Abbildung 2.5 Beachten Sie: „if“ kann ohne „else“ auftreten, aber niemals „else“ ohne „if“.

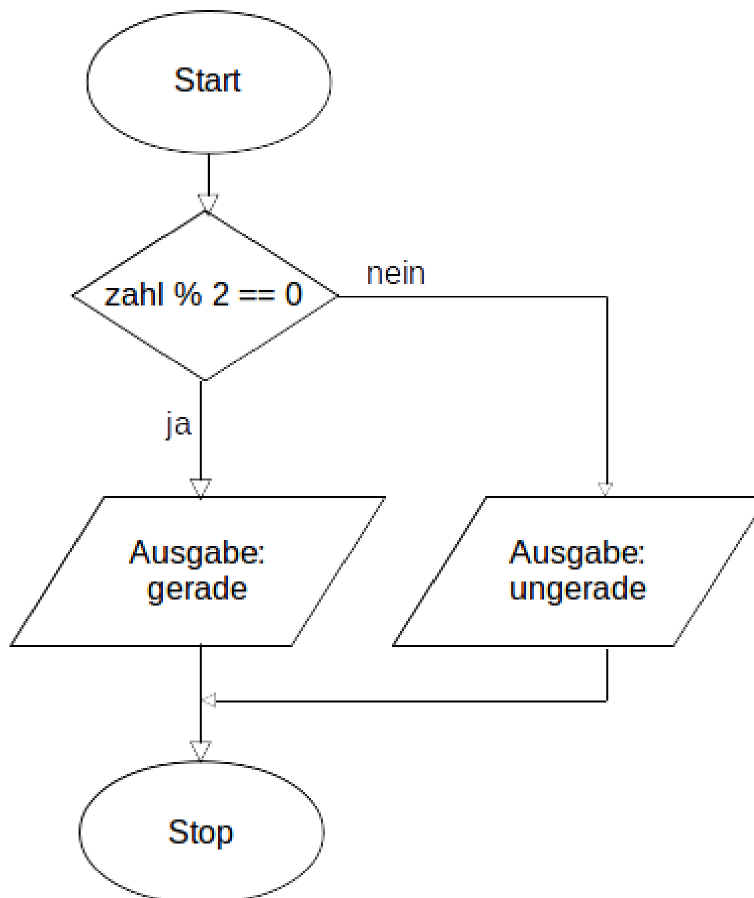


Abbildung 2.5: Ein Flussdiagramm mit „if-else“

In einem „if“-Zweig, das heißt in der geschweiften Klammer, kann ein weiteres „if“ stehen. Genauso kann in einem „else“-Zweig ein weiteres „if“ stehen. Beispielsweise kann das Programm aus Aufgabe 2.13 auch umgesetzt werden wie in Listing 2.10.

Listing 2.10: Beispiel für eine Methode mit „if-else“ ineinander

```
1  public void warnen(){
2      if(masse > idealmasse) {
3          System.out.println("Ihre Masse ist " +(masse -
               idealmasse) + " kg zu gross.");
4      }
5      else{
6          if(masse < idealmasse) {
7                  System.out.println("Ihre Masse ist " +(
                       idealmasse - masse) + " kg zu klein.");
8          }
9          else{
10             System.out.println("Gratualtion! Sie haben Ihre
                   Idealmasse.");
11         }
12     }
13 }
```

Aufgabe 2.14 *Flussdiagramm erstellen*
Erstellen Sie das Flussdiagramm zum Programm in Listing 2.10.

Die weiteren Aufgaben erfordern verschiedene Vergleichsoperatoren. Daher sollen hier die bereits bekannten Vergleichsoperatoren zusammen mit weiteren dargestellt werden.

== gleich

!= ungleich

< kleiner

<= kleiner oder gleich

> größer

>= größer oder gleich

Aufgabe 2.15 *Verbessertes BMI-Programm*

1. *Schreiben Sie nochmal ein Programm, das Auskunft über die Masse und die Idealmasse einer Versuchsperson gibt. Das Programm soll, wie in Aufgabe 3, objektorientiert sein. Die Eingabe von Größe und Masse soll über ein Objekt der Klasse „Textfenster“ erfolgen.*

   Das Programm soll aber nicht so starr arbeiten wie bisher. Der Body-Mass-Index (BMI) soll ausgerechnet werden. Es gilt $bmi = masse/(groesse * groesse)$. Dabei wird die Größe in Meter und die Masse in Kilogramm eingesetzt. Es gilt:

   - *Personen mit einem BMI > 30 sind stark übergewichtig*

- *Personen mit* $30 >= BMI > 25$ *sind leicht übergewichtig*
- *Personen mit* $25 >= BMI > 18,5$ *sind normalgewichtig*
- *Personen mit* $18,5 >= BMI > 16$ *sind leicht untergewichtig*
- *Personen mit* $16 >= BMI$ *sind stark untergewichtig*

2. *Schreiben Sie eine weitere Version des BMI-Programms. Es soll gleich funktionieren wie in Teil 1. Es soll aber kein „else“ verwendet werden. Wenn Sie auf „else“ verzichten möchten, dann benötigen Sie den Operator &&. Er stellt die logische UND-Verknüpfung dar. Beispiel: if(bmi <= 25 && bmi >=18.5){...} Je nachdem, wie Sie die Aufgabe lösen, kann auch der ODER-Operator hilfreich sein. Er lautet in Java ||.*

Aufgabe 2.16 *Quadratische Gleichung*

Schreiben Sie ein Programm, das die Lösungen einer quadratischen Gleichung $ax^2 + bx + c = 0$ *ausgibt. Der Benutzer soll a, b und c eingeben können. Sie benötigen die Wurzelfunktion für die Lösung. Diese wird in Java durch die statische Methode „sqrt()“ in der Bibliotheksklasse „Math“ zur Verfügung gestellt. Diese Klasse ist automatisch importiert. Die Methode benötigt einen Parameter vom Typ „double“ und gibt dann dessen Wurzel zurück. Anwendungsbeispiel:* `double d = Math.sqrt(b*b - 4*a*c);`

Bevor Sie programmieren, denken Sie über folgende Fragen nach: Wann hat eine solche Gleichung zwei Lösungen, eine Lösung oder keine Lösung? Was passiert, wenn a = 0 ist und wie möchten Sie mit diesem Fall umgehen?

4.3 Die while-Schleife

Gibt ein Benutzer beim Programm „BMI“ eine negative Masse ein, dann ist sicher ein Fehler aufgetreten. Das Programm bemerkt dies aber nicht und arbeitet weiter. Es wird dann ein negativer BMI ausgerechnet, der dann dazu führt, dass das Programm „starkes Untergewicht“ meldet.

Aufgabe 2.17 *Schlechte Lösung Warum löst ein Code wie der folgende das Problem nicht?*

```
1 public void eingeben(){
2     masse = fenster.doubleEingeben("Bitte die Masse in kg
          eingeben.");
3     if (masse <= 0) {
4         masse = fenster.doubleEingeben("Bitte die Masse in kg
              eingeben.");
5     }
6 }
```

Möchte man das Problem mit „if“ lösen, dann benötigt man Rekursion, die in Kapitel 4, Abschnitt 3 erklärt wird. Besser geht es mit einer while-Schleife. „while“ bedeutet auf deutsch „solange“. Der Teil in den geschweiften Klammern hinter „while“ wird solange wiederholt, wie die Bedingung in den runden Klammern wahr ist. Im obigen Code-Beispiel wird also durch das „if“ die Eingabeaufforderung nur einmal wiederholt, wenn beim ersten Mal eine falsche Eingabe gemacht wurde. Macht der Benutzer noch einmal eine falsche Eingabe, dann wird mit der falschen Eingabe weiter gearbeitet. Ersetzt man „if“ durch „while“, dann wird die Eingabeaufforderung so oft wiederholt, bis eine korrekte Eingabe gemacht wird. Ein solches Konstrukt mit Wiederholungen hießt „Schleife“ (englisch: loop). Der Code sollte also so aussehen:

Listing 2.11: Beispiel für eine while-Schleife

```
1 public void eingeben(){
2     masse = fenster.doubleEingeben("Bitte die Masse in kg
          eingeben.");
3     while (masse <= 0) {
4         masse = fenster.doubleEingeben("Bitte die Masse in kg
              eingeben.");
5     }
6 }
```

Von der while-Schleife gibt es in vielen Programmiersprachen eine Variante, auch in Java: die do-while-Schleife. Hier kommt nach dem Schlüsselwort „do“ (englisch für „mache, tu“) zuerst der zu wiederholende Code in geschweiften Klammern, dann erst folgt „while“ mit der Bedingung in runden Klammern. In diesem Fall wird der Code in geschweiften Klammern auf alle Fälle einmal durchgeführt. Er wird dann solange wiederholt, wie die Bedingung in den runden Klammern wahr ist. Im obigen Beispiel ist diese Variante besser: Einmal muss der Benutzer sicher die Masse eingeben. Ob die Eingabe wiederholt werden muss, hängt von deren Richtiigkeit ab. Der Code wird dadurch etwas kürzer:

Listing 2.12: Beispiel für eine do-while-Schleife

```
1 public void eingeben(){
2     do {
3         masse = fenster.doubleEingeben("Bitte die Masse in kg
              eingeben.");
4     }
5     while (masse <= 0);
6 }
```

4.4 Zufall

Für viele Anwendungen, z.B. Spiele, benötigt man Zufallszahlen. Ein Computer soll aber bei gleichen Eingaben immer dieselben Ausgaben produzieren. Das „Ausdenken“ von Zufallszahlen kann er daher nicht. Allerdings genügen in den meisten Fällen sogenannte „Pseudozufallszahlen“, deren Erstellung über komplizierte Funktionen von Größen wie der aktuellen Uhrzeit abhängen und damit schwer vorhersagbar sind. Im weiteren Text ist von „Zufallszahlen“ die Rede, auch wen es sich exakt nur um „Pseudozufallszahlen“ handelt. Möchte man in Java Zufallszahlen benutzen, ist es am einfachsten, ein Objekt der Klasse „Random“ zu erzeugen und sich die Zufallszahlen von diesem produzieren lassen. Die Klasse „Random“ muss zu Beginn der Klasse, in der diese Klasse verwendet wird, durch die Anweisung `import java.util.Random;` hinzugefügt werden. Dann kann ein Objekt dieser Klasse erzeugt werden. Objekte der Klasse „Random“ verfügen über die Methode „nextInt()“ mit einem Parameter vom Typ „int“. Die Methode gibt eine Zahl vom Typ „int“ zurück. Diese ist größer oder gleich 0 und kleiner als der Parameter. Beispiel:

```
1 import java.util.Random;
2
3 public class Zufall {
4     public static void main(String[] args){
5         Random zufall = new Random();
6         int zahl = zufall.nextInt(6);
7         System.out.println(zahl);
8     }
9 }
```

Dieses Programm gibt ein zufällige ganze Zahl zwischen 0 und 5 aus.

Aufgabe 2.18 *Zufallszahlen*

1. *Programmieren Sie einen elektronischen Würfel, d.h. ein Programm, dass ganze Zufallszahlen zwischen 1 und 6 ausgibt.*

2. *Schreiben Sie ein Programm, das immer zwei Zufallszahlen x und y ausgibt. Dabei soll gelten* $x + y \leq 20$. *Hinweis: Es ist keine akzeptable Lösung, x und y einfach aus dem Bereich von 0 bis 10 zu wählen, um* $x + y \leq 20$ *sicher zu stellen.*

3. *Schreiben Sie ein Rechenübungsprogramm für Schulanfänger. Das Programm stellt dem Benutzer eine Additionsaufgabe. z.B. „13 + 4 = ?“. Gibt der Benutzer das richtige Ergebnis ein, wird er dafür gelobt. Gibt er ein falsches Ergebnis ein, muss er die Aufgabe nochmal lösen. Die Ergebnisse sollen nicht größer als 20 sein.*

4.5 Die for-Schleife

Bei Aufgabe 2.18, Teilaufgabe 3 ist störend, dass nur eine Aufgabe gestellt wird. Besser wäre es, dass der Benutzer 15 Aufgabe lösen soll. Dies kann man durch eine while-Schleife erreichen, z.B. so wie in dieser Methode:

Listing 2.13: Feste Anzahl von Wiederholungen mit „while“

```
1 public void wiederhole15mal(){
2     int n = 0;
3     while (n < 15){
4         System.out.println("Wiederholung");
5         n++;
6     }
7 }
```

Die Anweisung `++` ist der Inkrementoperator. Er vergrößert die Variable um 1. Die Anweisung `n = n + 1` hätte dieselbe Wirkung.

Aufgabe 2.19 *Rechentrainer mit 15 Aufgaben*
Schreiben Sie das Programm aus Aufgabe 2.18, Teilaufgabe 3 neu, aber so, dass 15 Aufgaben gestellt werden. Sie haben dann zwei while-Schleifen: Eine äußere Schleife, die die 15 Aufgaben organisiert und eine innere Schleife, die die Richtigkeit des Ergebnisses kontrolliert und bei falschem Ergebnis die wiederholte Eingabe des Ergebnisses verlangt.

Eine Benutzung der while-Schleife wie in Listing 2.13 ist nicht gut, denn Sie Steuerung der 15 Wiederholungen ist auf drei Zeilen verteilt: Zeile 2 initialisiert die Zählvariable n. Zeile 3 enthält die Schleifenbedingung und in Zeile 5 wird die Zählvariable inkrementiert. Es ist schwer, hier den Überblick zu behalten und es wird noch schwerer, wenn in der Schleife mehr Anweisungen stehen.

Daher wird statt einer while-Schleife in diesem Fall besser eine for-Schleife verwendet, bei der diese Information in einer einzigen Zeiele steht. Das sieht dann so aus:

Listing 2.14: Feste Anzahl von Wiederholungen mit „for“

```
1 public void wiederhole15mal(){
2     for (int n = 0; n < 15; n++){
3         System.out.println("Wiederholung");
4     }
5 }
```

Aufgabe 2.20 *for-Schleife*

1. *Schreiben Sie eine Methode, die folgende Ausgabe erzeugt:*
   1 2 3 4 5 6 7 8 9 10

Die Methode soll nur einen Ausgabebefehl beinhalten, nämlich `System.out.print(i +"\t");` *Hinweis: "\t" fügt einen Tabulator ein.*

2. *Schreiben Sie eine Methode, die folgende Ausgabe erzeugt:*
   7 14 21 28 35 42 49 56 63 70
   Wie oben soll die Methode nur einen Ausgabebefehl enthalten, der pro Aufruf nur eine Zahl ausgibt.

Alles was man mit einer for-Schleife machen kann, kann man auch mit einer while-Schleife machen. Umgekehrt könnte man auch alles mit der for-Schleife erledigen. Man braucht also nur eine der beiden Schleifenarten.

Man benutzt beide Schleifenarten, um möglichst übersichtlichen Code zu erhalten. Es gelten dabei die folgenden Regeln:

- Ist zu Beginn der Schleife die Anzahl der Wiederholungen bekannt, dann wird eine for-Schleife benutzt.
- Ist zu Beginn der Schleife die Anzahl der Wiederholungen nicht bekannt, dann wird eine while-Schleife benutzt. Dies ist z.B. dann der Fall, wenn die Anzahl der Wiederholungen von einer Benutzereingabe abhängt, siehe z.B. die Listings 2.11 und 2.12.

Aufgabe 2.21 *Rechentrainer: Schleifen überarbeiten*
Überarbeiten Sie den Rechentrainer aus Aufgabe 2.19: Überlegen Sie, ob zwei while-Schleifen in diesem Programm gut sind oder ob nicht besser eine oder beide durch eine for-Schleife ersetzt werden sollten und führen Sie diese Veränderung durch.

In einer Methode können mehrere Schleifen vorkommen. Die Schleifen können hintereinander vorkommen oder ineinander vorkommen. Aufgaben, die ineinander liegende Schleifen erfordern, sind meistens schwerer zu lösen. Manchmal ist ist es sinnvoll, eine solche Methode in zwei Methoden aufzuteilen, so dass in jeder Methode nur eine Schleife vorkommt. Auf alle Fälle müssen Sie solche Konstrukte üben.

Aufgabe 2.22 *Aufgaben mit ineinander liegenden Schleifen*

1. *Schreiben Sie eine Methode, die mit nur einem Ausgabebefehl die folgende Ausgabe erzeugt:*
```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
1 * 6 = 6
```

*1 * 7 = 7*
*1 * 8 = 8*
*1 * 9 = 9*
*1 * 10 = 10*
*2 * 1 = 2*
*2 * 2 = 4*
*2 * 3 = 6*
*2 * 4 = 8*
*2 * 5 = 10*
*2 * 6 = 12*
*2 * 7 = 14*
*2 * 8 = 16*
*2 * 9 = 18*
*2 * 10 = 20*
*3 * 1 = 3*
... und so weiter bis
*10 * 10 = 100*

2. *Schreiben Sie eine Methode mit einer Schleife, die folgende Bildschirmausgabe erzeugt:*
   xxxxxxxxxx
   xxxxxxxxxx
   xxxxxxxxxx
   xxxxxxxxxx
   xxxxxxxxxx
   xxxxxxxxxx
   a) In Ihrer Methode darf nur einmal die Methode `System.out.println(''xxxxxxxxxx'')` vorkommen, keine weiteren Ausgabemethoden.
   b) Schreiben Sie eine weitere Methode, die dieselbe Ausgabe erzeugt. In Ihrer Methode darf nur einmal die Methode `System.out.print(''x'')` vorkommen, außerdem einmal `System.out.println()` zur Erzeugung der Zeilenumbrüche. Dafür dürfen Sie nicht nur eine, sondern zwei Schleifen verwenden.

3. *Schreiben Sie eine Methode mit zwei Schleifen, die folgende Bildschirmausgabe erzeugt:*
   x
   xx
   xxx
   xxxx
   xxxxx
   xxxxxx

xxxxxxx
xxxxxxxxx
Hinweis: In Ihrer Methode darf nur einmal die Methode `System.out.print("x")` *vorkommen, außerdem einmal* `System.out.println()` *zur Erzeugung der Zeilenumbrüche.*

4. *Schreiben Sie eine Methode mit zwei Schleifen, die folgende Bildschirmausgabe erzeugt:*
   xxxxxxxxxxx
   xxxxxxxxx
   xxxxxxx
   xxxxx
   xxx
   x
   Hinweis: In Ihrer Methode darf nur einmal die Methode `System.out.print("x")` *vorkommen, außerdem einmal* `System.out.println()` *zur Erzeugung der Zeilenumbrüche.*

5. *Schreiben Sie je eine möglichst kurze Methode, die folgende Bildschirmausgaben erzeugen:*
   a)
   *
   ***
   *****
   ****
   ****
   ****
   b)
   ****
   ****
   ****
   *****
   ***
   *

c)

```
*
  *
    *
      *
        *
          *
```

d)

```
          *
        *
      *
    *
  *
*
```

5 Arrays

5.1 Wozu Arrays?

Ein Array, im Deutschen auch manchmal „Feld“ genannt, ist ein Tupel gleichartiger Objekte oder primitiver Datentypen. Die Verwendung eines Arrays anstelle vieler einzelner Variablen kann Vorteile haben, z.B.

- können mehrere Variablen zusammen gehören, z.B. die Komponenten eines Vektors. Diese Zusammengehörigkeit wird bei Verwendung eines Arrays deutlich, bei Verwendung einzelner Variablen nicht unbedingt.
- will man in einer Schleife mehrere Variablen gleichartig verarbeiten. Dies geht mit einem Array.
- braucht man manchmal sehr viele Variablen desselben Typs. Man will sie nicht alle einzeln deklarieren.
- entscheidet sich manchmal erst zur Laufzeit des Programms, nicht schon bereits beim Programmieren, wie viele Variablen man benötigt. Dieses Problem kann man mit Arrays lösen.

5.2 Ein Beispiel für Arrays: Vektoren

Ein Vektor im $\mathbb{Q}^3$ ist ein Tripel von rationalen Zahlen[2] Eine Klasse für Vektoren, die eingegeben und ausgegeben werden können, könnte so aussehen:

Listing 2.15: Vektoren mit einzelnen Variablen

```
public class Vektor{
    double komponente1;
    double komponente2;
    double komponente3;
    Textfenster fenster;

    /**
     * Konstruktor
     * @param dasFenster Jeder neue Vektor bekommt im
     *  Konstruktor als Parameter ein Textfenster
     *  gegeben, das in den Methoden "eingeben()" und
     *  "ausgeben()" benutzt wird.
     */
    public Vektor(Textfenster dasFenster){
        fenster = dasFenster;
    }

    public void eingeben(){
        komponente1 = fenster.doubleEingeben("Komponente 1");
        komponente2 = fenster.doubleEingeben("Komponente 2");
        komponente3 = fenster.doubleEingeben("Komponente 3");
    }

    public void ausgeben(){
    fenster.schreiben("(");
    fenster.schreiben(komponente1);
    fenster.schreiben(" ");
    fenster.schreiben(komponente2);
    fenster.schreiben(" ");
    fenster.schreiben(komponente3);
    fenster.schreiben(")");
```

[2] In der Mathematik spielt der Vektorraum $\mathbb{R}^3$ eine viel größere Rolle, muss aber in der Computertechnik meistens durch den $\mathbb{Q}^3$ ersetzt werden, da die Darstellung der reellen Zahlen im Computer Probleme bereitet. Tatsächlich werden auch die rationalen Zahlen $\mathbb{Q}$ im Computer nicht perfekt dargestellt. Mehr dazu erfahren Sie im Kapitel „Grundlagen der Technischen Informatik“, Abschnitt 1.6 ab Seite 54.

```
32          }
33  }
```

Aufgabe 2.23 *Vektor eingeben und ausgeben Erstellen Sie ein Projekt. Übernehmen Sie darin die in Listing 2.15 gezeigt Klasse „Vektor“ und die Ihnen bekannte Klasse „Textfenster“. Schreiben Sie eine main()-Methode, in der eine Instanz der Klasse „Textfenster“ und eine Instanz der Klasse „Vektor“ erzeugt wird und auf dem Vektor die Methoden „eingeben()“ und „ausgeben()“ aufgerufen werden.*

Bei den Methoden „eingeben()“ und „ausgeben()“ stört, dass fast derselbe Text mehrmals eingegeben werden muss, weil drei gleichartige Attribute existieren. Möchte man Methoden hinzufügen, um mit den Vektoren zu rechnen, wird das Problem noch unangenehmer.

Eleganter kann man eine solche Aufgabe mit einem Array bewältigen. Ein Vektor kann damit als ein Array von drei Variablen des Typs „double“ dargestellt werden. Nennt man das Array, also das Tupel „komponente“, dann heißen die einzelnen Komponenten des Arrays komponente[0], komponente[1] und komponente[2]. Statt von „Komponenten“ wie beim Vektor spricht man bei einem Array meistens von „Elementen“. Ein Array wird deklariert durch den Typ der Elemente gefolgt von einem leeren eckigen Klammerpaar, also [], dann folgt der Name der Arrayvariablen. Für ein Array „komponente“ lautet die Deklaration also `double[] komponente;`. Ein Array ist immer ein Objekt, das zunächst durch den Aufruf des new-Operators erzeugt werden muss. Dieser Aufruf lautet in diesem Beispiel `komponente = new double[3];`. Dieser Aufruf kann z.B. im Konstruktor einer Klasse „Vektor“ auftauchen, kann aber auch direkt auf die Deklaration folgen wie unten gezeigt. Die Zahl in den eckigen Klammern gibt an, aus wie vielen Elementen das Array besteht. Beachten Sie, dass die Elemente von 0 an durchnummeriert werden. Ein Element komponente[3] gibt es daher nicht. Wenn die Komponenten des Vektors komponente[1], komponente[2] und komponente[3] heißen sollen, dann muss der new-Operator in der Form `komponente = new double[4]` aufgerufen werden, wobei dann das Element komponente[0] nicht benutzt wird. Die Klasse Vektorrechnung könnte unter Verwendung eines Array so aussehen wie in Listing 2.16 gezeigt, wobei hier noch eine Methode für die Vektoraddition ergänzt wurde. Sie sehen, dass die Methoden „eingeben()“ und “ausgeben()“ so eleganter sind.

Listing 2.16: Vektor mit Array

```
1  public class Vektor {
2      double[] komponente = new double [3];
3      Textfenster fenster;
4
5      /**
6       * Konstruktor
```

```
7        * @param dasFenster Jeder neue Vektor bekommt im
8        * Konstruktor als Parameter ein Textfenster
9        * gegeben, das in den Methoden "eingeben()" und
10       * "ausgeben()" benutzt wird.
11       */
12     public Vektor(Textfenster dasFenster){
13         fenster = dasFenster;
14     }
15
16     public void eingeben(){
17         for (int i = 0; i < 3; i++){
18             komponente[i] = fenster.doubleEingeben("Komponente
                 " +(i+1));
19         }
20     }
21
22     public void ausgeben(){
23         fenster.schreiben("(");
24         for(int i = 0; i < 3; i++){
25             fenster.schreiben(komponente[i] +" ");
26         }
27         fenster.schreiben(")\n");
28     }
29
30     public Vektor addieren(Vektor summand){
31         Vektor summe = new Vektor(fenster);
32         for (int i = 0; i < 3; i++){
33             summe.komponente[i] = komponente[i] + summand.
                 komponente[i];
34         }
35         return summe;
36     }
37 }
```

Eine Klasse mit einer main()-Methode, die Gebrauch von der Klasse „Vektor“ macht, um eine Vektoraddition durchzuführen, könnte aussehen wie in Listing 2.17 gezeigt.

Listing 2.17: Klasse mit main()-Methode für die Vektoraddition

```
1 public class Main {
2     public static void main(String[] args) {
3         Textfenster f = new Textfenster();
4         Vektor u = new Vektor(f);
```

```
5           Vektor v = new Vektor(f);
6           u.eingeben();
7           v.eingeben();
8           u.ausgeben();
9           f.schreiben("+ ");
10          v.ausgeben();
11          f.schreiben("= ");
12          u.addieren(v).ausgeben();
13      }
14  }
```

Aufgabe 2.24 *Rechnen mit Vektoren*

Testen Sie Ihre Lösungen, indem Sie in der main()-Methode Beispiel berechnen.

1. *Ergänzen Sie die Klasse „Vektor“ um eine Methode, um einen Vektor mit einer Zahl multiplizieren zu können.*
2. *Ergänzen Sie die Klasse „Vektor“ um eine Methode, die das Skalarprodukt zweier Vektoren ausrechnet.*
3. *Falls Sie das Kreuzprodukt bereits behandelt haben: Erweitern Sie die Klasse um eine Methode für das Kreuzprodukt.*

Man kann auch mehrdimensionale Arrays bilden. Man kann sie auch als Arrays von Arrays auffassen. Ein zweidimensionales Array könnte z.B. so erzeugt werden: `double[][] matrix = new double[3][4];`.

Aufgabe 2.25 *Gauß-Verfahren*

1. *Schreiben Sie ein Programm, dass ein Gleichungssystem aus drei Gleichungen mit drei Unbekannten nach dem Gauß-Verfahren löst. Benutzen Sie ein zweidimensionales Array. Bemerkung: Der Fall von drei Gleichungen mit drei Unbekannten kommt häufig vor, z.B. bei Statikaufgaben in zwei Dimensionen wie in Ihrem Physikkurs.*
2. *Verbessern Sie Ihr Programm: Die Anzahl der Gleichungen und Variablen darf vom Benutzer gewählt werden.*

Kapitel 3

Grundlagen der Technischen Informatik

1 Zahlensysteme

1.1 Römische Zahlen – ein Additionssystem

Bei den römischen Zahlen gibt es mehrere leicht unterschiedliche Systeme. Abweichungen von dieser Darstellung zu anderen sind daher möglich. Bei den römischen Zahlen gelten die Symbole aus Tabelle 3.1.

Tabelle 3.1: römische Ziffern

römische Ziffer	I	V	X	L	C	D	M
Bedeutung in indo-arabischen Zahlen	1	5	10	50	100	500	1000

Alle weiteren Werte werden durch Addition, teilweise auch durch Subtraktion, erzeugt. Bei der Addition stehen immer die höherwertigen Zeichen links. So ist beispielsweise das Jahr 2010 in römischen Zahlen MMX. Es wird vermieden, viermal hintereinander dasselbe Symbol zu benutzen, indem man eine Ziffer subtrahiert statt addiert. Das Zeichen, das subtrahiert werden muss, kann dadurch erkannt werden, dass es vor einem höherwertigen Zeichen steht. So bedeutet beispielsweise IX die Zahl 9, XL die Zahl 40. Es stehen niemals zwei Zeichen zur Subtraktion hintereinander, die Zahl 8 wird also VIII geschrieben, nicht etwa IIX. Es werden nur die Ziffern, die Zehnerpotenzen symbolisieren, also I, X, oder C, zur Subtraktion voran gestellt. 95 wird also XCV geschrieben, nicht VC. Ein Zeichen wird nur den beiden nächst höheren zur Subtraktion vorangestellt, also I kann vor V und X stehen, X vor L und C, C vor D und M, nicht aber z.B. I vor C. 99 wird also XCIX geschrieben, nicht IC.

Aufgabe 3.1 *Römische Zahlen*
Schreiben Sie das heutige Datum und Ihr Geburtsdatum in römischen Zahlen.

Die Nachteile des römischen Zahlensystems sind offensichtlich: Zum einen lassen sich nicht so einfach beliebig große Zahlen darstellen. Zwar gibt es auch Ziffern für 5000 und 10000, aber wenn man noch größere Zahlen haben möchte, muss man ständig neue Ziffern erfinden. Noch problematischer ist das Rechnen. Es gibt keine einfachen Methoden, bei denen diese Zahlendarstellung beim Rechnen hilft. Warum wurde dieses Zahlensystem dann hier behandelt? Zur Abschreckung! Es soll Ihnen klar werden, wie vorteilhaft Stellenwertsysteme sind im Gegensatz zu Additionssystemen.

1.2 Das Dezimalsystem – ein Stellenwertsystem

Tabelle 3.2: Additionstabelle im Dezimalsystem

+	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	6	7	8	9	10	11
3	3	4	5	6	7	8	9	10	11	12
4	4	5	6	7	8	9	10	11	12	13
5	5	6	7	8	9	10	11	12	13	14
6	6	7	8	9	10	11	12	13	14	15
7	7	8	9	10	11	12	13	14	15	16
8	8	9	10	11	12	13	14	15	16	17
9	9	10	11	12	13	14	15	16	17	18

Das Dezimalsystem, auch Zehnersystem genannt, ist ein Stellenwertsystem. Es ist das heute üblichste Zahlensystem und ist wahrscheinlich in Indien entstanden und hat sich über den arabischen Raum auf der ganzen Welt verbreitet. Man spricht im Deutschen daher von „arabischen Zahlen“, manchmal auch „indischen Zahlen“ oder „indisch-arabischen Zahlen“, wenn man von Zahlen im Zehnersystem spricht. Stellenwertsystem bedeutet, dass der Wert einer Ziffer nicht nur von deren Symbol, sondern auch von deren Stellung abhängig ist. Beispielsweise bedeutet in der Zahl „110“ die erste Eins „hundert“, die zweite aber „zehn“. In „203“ bedeutet die Zwei „zwei Hunderter“, die Null bedeutet „keine Zehner“ und die Drei bedeutet „drei Einer“. Generell gilt: die n-te Stelle von rechts gezählt hat den Stellenwert 10^{n-1}. Ein Stellenwertsystem braucht eine Ziffer 0, die bei den römischen

Zahlen keine Rolle spielt. Das Dezimalsystem besitzt zehn Ziffern: 0, 1, 2, 3, 4, 5, 6, 7, 8 und 9.

Für die Addition gelten die üblichen Regeln der „schriftlichen Addition“: Zur Addition beliebiger Zahlen genügt die Kenntnis der Additionstabelle, welche die Ergebnisse der Addition aller beliebigen Kombinationen der Ziffern des Dezimalsystems angibt. Man kann damit jeweils die Ziffern derselben Stelle addieren. Ergibt sich ein zweistelliges Ergebnis, so muss dessen Zehnerstelle bei der nächsten Stelle mit berücksichtigt werden. Die Additionstabelle für das Dezimalsystem ist in Tabelle 3.2 angegeben. Für die Multiplikation benötigt man entsprechend die Multiplikationstabelle 3.3, außerdem, um die Überträge zu verarbeiten, wenn die Multiplikation zweier Ziffern ein zweistelliges Ergebnis liefert, die Additionstabelle. Dieses Verfahren ist als „schriftliche Multiplikation“ bekannt, das hier als bekannt voraus gesetzt wird.

Tabelle 3.3: Multiplikationstabelle im Dezimalsystem

·	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

1.3 Das Dualsystem – ein Stellenwertsystem

Das Dualsystem, auch Zweiersystem genannt, ist die Grundlage elektronischer Rechner, denn es besitzt zwei Ziffern, nämlich 0 und 1. Diese können durch „elektrische Spannung angelegt“ und „elektrische Spannung nicht angelegt“, kurz „an“ oder „aus“ repräsentiert werden. Wie bei allen Stellenwertsystemen hat die n-te Stelle von rechts den Stellenwert b^{n-1}, wobei b für die Basis steht, im Falle des Dualsystems also zwei. Die Stellenwerte sind also 1, 2, 4, 8, 16 usw. Zahlen im Dualsystem können durch eine tiefgestellte „2“ rechts gekennzeichnet werden. Möchte man die Zahl 11101_2 ins Dezimalsystem umrechnen, so

ergibt sich von rechts nach links: $1 \cdot 1 + 0 \cdot 2 + 1 \cdot 4 + 1 \cdot 8 + 1 \cdot 16 = 29$. Für die Umrechnung vom Dezimalsystem ins Dualsystem kann man folgendermaßen vorgehen: Man dividiert fortlaufend durch 2 und notiert die Divisionreste, bis man null erreicht. Die Divisionsreste liefern die Dualzahl. Als Beispiel soll die Umrechnung der Zahl 42 dienen.

$$\begin{aligned}42 &= 2 \cdot 21 + 0\\ 21 &= 2 \cdot 10 + 1\\ 10 &= 2 \cdot 5 + 0\\ 5 &= 2 \cdot 2 + 1\\ 2 &= 2 \cdot 1 + 0\\ 1 &= 2 \cdot 0 + 1\end{aligned}$$

Daraus ergibt sich $42 = 101010_2$. Eine andere Möglichkeit besteht darin, die höchste passende Zweierpotenz zu finden, die von der auszudrückenden Zahl abzuziehen und mit dem Rest genauso zu verfahren, bis es schließlich keinen Rest mehr gibt. Beispiel: 100 soll in eine Dualzahl umgerechnet werden. Die höchste in 100 passende Zweierpotenz ist 64. $100 - 64 = 36$. Die höchste in 36 passende Zweierpotenz ist 32. $36 - 32 = 4$. Die höchste in 4 passende Zweierpotenz ist 4. $4 - 4 = 0$. Also muss die 64er-Stelle und die 32er-Stelle besetzt sein, nicht aber die 16er- und die 8er-Stelle; die 4er-Stelle ist besetzt; die 2er-Stelle und die 1er Stelle sind nicht besetzt. Es ergibt sich $100 = 1100100_2$.

Aufgabe 3.2 *Umrechnung ins Dualsystem*
Rechnen Sie 42 und 100 mit der jeweils anderen Methode ins Dualsystem um.

Man kann auch Dualbrüche schreiben. So bedeutet $0{,}1_2$ die Zahl 0,5 im Dezimalsystem. Die Ziffern rechts des Kommas haben den Stellenwert 2^{-1}, 2^{-2}, 2^{-3}, 2^{-4} usw. Die Dualzahl $101{,}1101_2$ kann man folgendermaßen ins Dezimalsystem umrechnen:
$1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 + 1 \cdot 0{,}5 + 1 \cdot 0{,}25 + 0 \cdot 0{,}125 + 1 \cdot 0{,}0625 = 5{,}8125$

Umgekehrt kann man einen Dezimalbruch in einen Dualbruch umwandeln: In 0,43 passt 0,5 nicht, aber 0,25, Rest ist 0,18. Hier passt 0,125, Rest ist 0,055. 0,0625 passt nicht, aber 0,03125 usw. Es ergibt sich $0{,}43 \approx 0{,}01101..._2$. Es kann passieren, dass ein endlicher Dezimalbruch bei der Umrechnung zu einem periodischen Dualbruch wird oder umgekehrt. Analog zum anderen oben geschilderten Verfahren könnte man auch so vorgehen:

$$\begin{aligned}0,43 \cdot 2 &= 0 + 0,86\\ 2 \cdot 0,86 &= 1 + 0,72\\ 2 \cdot 0,72 &= 1 + 0,44\\ 2 \cdot 0,44 &= 0 + 0,88\\ 2 \cdot 0,88 &= 1 + 0,76\end{aligned}$$

Auch hier ergibt sich $0{,}43 \approx 0{,}01101..._2$.

Zum Addieren von Dualzahlen braucht man wiederum eine Additionstabelle. Diese ist in diesem Fall aber erfreulich kurz, wie man an Tabelle 3.4 sieht.

Tabelle 3.4: Additionstabelle im Dualsystem

+	0	1
0	0	1
1	1	10

Damit kann man folgendermaßen addieren:

1. Summand	1 1 0 0 1 0 0
2. Summand	1 0 1 0 1 0
Übertrag	1 1
Ergebnis	1 0 0 0 1 1 1 0

Aufgabe 3.3 *Kontrolle*
Rechnen Sie die obigen Zahlen ins Dezimalsystem um und kontrollieren Sie so, dass die Rechnung korrekt ist!

Die Multiplikationstabelle 3.5 ist genauso kurz.

Tabelle 3.5: Multiplikationstabelle im Dualsystem

·	0	1
0	0	0
1	0	1

Aufgabe 3.4 *Multiplizieren im Dualsystem*
Rechnen Sie die die Zahlen 12 und 23 ins Dualsystem um und berechnen Sie im Dualsystem deren Produkt.

1.4 Weitere Stellenwertsysteme

Man kann zu jeder Basis $b > 1$ ein Stellenwertsystem angeben. Das System hat b Ziffern, die erste ist „0“, die letzte $b - 1$. Die Stellenwerte sind die Potenzen von b, also b^0, b^1, b^2 usw. Umrechnungen vom und ins Dezimalsystem können analog wie beim Dualsystem erfolgen.

Aufgabe 3.5 *Das Dreiersystem als ein weiteres Beispiel eines Stellenwertsystems*

1. *Rechnen Sie die Zahlen 30 und 42 ins Dreiersystem um.*

2. *Stellen Sie die Additionstabelle für das Dreiersystem auf und addieren Sie die in Teil a erhaltenen Zahlen. Rechnen Sie das Ergebnis vom Dreiersystem ins Dezimalsystem um und kontrollieren Sie, ob das Ergebnis korrekt ist.*

3. *Stellen Sie die Multiplikationstabelle für das Dreiersystem auf und multiplizieren Sie die in Teil a erhaltenen Zahlen. Rechnen Sie das Ergebnis ins Dezimalsystem um und kontrollieren Sie, ob es korrekt ist.*

Außer dem Dualsystem spielt für Computer noch das Hexidezimalsystem (Sechzehnersystem) eine Rolle. Es ist nämlich ein guter Kompromiss, was die Verständlichkeit für Computer und Menschen betrifft: Die Zahlen sind kürzer und leichter zu merken als beim Dualsystem, was dem Menschen entgegenkommt. Die Umrechnung ins Dualsystem ist denkbar einfach, was dem Computer entgegenkommt. Das Hexadezimalsystem benötigt 16 Ziffern. Diese sind 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Dabei gilt $A_{16} = 10$, $B_{16} = 11$, $C_{16} = 12$, $D_{16} = 13$, $E_{16} = 14$, $F_{16} = 15$. Die Stellenwerte sind von rechts nach links 1, 16, 256, 4096 usw. Die Umrechnung vom Hexadezimalsystem ins Dezimalsystem funktioniert nach den obigen Verfahren. Die Umrechnung zwischen Dualsystem und Hexadezimalsystem kann entsprechend durchgeführt werden, geht aber bedeutend einfacher. Es gilt Tabelle 3.6 für die Umrechnung der Ziffern des Hexadezimalsystems:

Will man nun die Hexadezimalzahl $A2C_{16}$ ins Dualsystem umrechnen, so muss man nur die Darstellungen der obigen Ziffern aneinander reihen: $A2C_{16} = 101000101100_2$. Dies funktioniert auch umgekehrt. Gegebenenfalls muss dafür die Dualzahl so mit führenden Nullen ergänzt werden, dass sie eine durch vier teilbare Anzahl von Ziffern besitzt: $1001011011_2 = 001001011011_2 = 25B_{16}$.

Aufgabe 3.6 *Umrechnung bei „verwandten“ Zahlensystemen*

1. *Erklären Sie, wann eine solche einfache Umrechnung zwischen zwei verschiedenen Stellenwertsystemen funktioniert.*

2. *In welches System könnte man Zahlen im Dualsystem ebenfalls problemlos umrechnen?*

3. *In welches System könnte man Zahlen im Dreiersystem problemlos umrechnen?*

4. *Stellenwertsysteme lassen sich zu beliebigen Basen größer 1 erzeugen. Warum hat sich das Dezimalsystem bei Menschen durchgesetzt?*

5. *Zusatzaufgabe für Fans von Donald Duck: Welches Zahlensystem dürfte sich langfristig in Entenhausen durchsetzen?*

Tabelle 3.6: Umrechnung hexadezimal – dual

hexadezimale Ziffer	**Dualzahl in vierstelliger Schreibweise**
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

1.5 Vorzeichendarstellung im Dualsystem

Zahlen werden im Computer als Dualzahlen dargestellt. Allerdings fehlt nach dem bisher Gesagten eine Vorzeichendarstellung. Es ist keine Lösung, negative Zahlen mit einem Minuszeichen zu versehen, denn dies wäre die Einführung eines dritten Zeichens. Das Dualsystem wurde aber eben deswegen gewählt, weil die Coputertechnik prinzipiell dual ist – Spannung ein und Spannung aus. Also muss das Vorzeichen ebenfalls durch 0 und 1 dargestellt werden. Dafür sind drei Möglichkeiten üblich. Allen diesen Darstellungen ist gemeinsam, dass die Stelle ganz links das Vorzeichen der Zahl bestimmt. Daraus folgt, dass bei diesen Darstellungen die Anzahl der Ziffern festgelegt sein muss, damit man weiß, welches die „Stelle ganz links“ ist, um sie korrekt berücksichtigen zu können. Dies macht die Benutzung von führenden Nullen notwendig. In diesem Abschnitt werden Dualzahlen mit Vorzeichen genauso wie bei nicht negativen Dualzahlen durch eine tiefgestelle 2 ge-

kennzeichnet. Die richtige Interpretation ergibt sich aber erst aus dem Text, da klar sein muss, welche Darstellung zur Anwendung kommt.

Die erste Möglichkeit, negative Zahlen darzustellen, besteht darin, die Stelle ganz links als Vorzeichen zu interpretieren, den Rest als Betrag. „1“ ganz links bedeutet dabei negativ, „0“ positiv. So wäre z.B. bei achtstelligen Dualzahlen in dieser Darstellung $00011001_2 = +25$ und $10011001_2 = -25$. Bei einer Dualzahl in der Darstellung Vorzeichen und Betrag mit n + 1 Stellen (also z.B. von 0 bis n durchnummeriert) gilt für die Umrechnung in eine Dezimalzahl $<a_n, a_{n-1}, \ldots, a_1, a_0> = (-1)^{a_n} \cdot (a_{n-1} \cdot 2^{n-1} + ... + a_1 \cdot 2^1 + a_0 \cdot 2^0)$. Nachteilhaft an dieser Darstellung ist, dass es zwei Darstellungen für die Zahl 0 gibt, nämlich (wiederum bei achtstelligen Zahlen) 00000000_2 und 10000000_2. Problematischer ist allerdings, dass keine normale Rechnung möglich ist mit diesen Zahlen. So ergäbe beispielsweise 3 + (–2) nicht 1, sondern $00000011_2 + 10000010_2 = 10000101_2 = -5$, denn $3 + 2 = 5$, aber das Vorzeichen von –2 taucht im Ergebnis auf. Will man korrekte Algorithmen für das Rechnen mit Zahlen in dieser Darstellung haben, so muss man komplizierte Fallunterscheidungen berücksichtigen.

Aufgabe 3.7 *Vorzeichen und Betrag*
Es geht im folgenden um vierstellige Dualzahlen in der Darstellung mit Vorzeichen und Betrag.

1. *Wie viele verschiedene Zahlen lassen sich darstellen?*
2. *Wie lauten die kleinste und die größte darstellbare Zahl?*
3. *Stellen Sie die Zahlen 3 und –3 und 8 und –8 dar.*

Eine weitere Möglichkeit, negative ganze Zahlen darzustellen, ist das Einerkomplement. Hier beginnt jede positive Zahl mit einer 0. Die betragsgleiche negative Zahl erhält man, indem man jede „0“ durch „1“ ersetzt und jede „1“ durch „0“. Als Beispiel sollen vierstellige Dualzahlen in dieser Darstellung dienen. Die Zahl 5 lautet dann 0101_2, die Zahl –5 entsprechend 1010_2. Allgemein gilt bei einer solchen Zahl mit n + 1 Stellen

$<a_n, a_{n-1}, \ldots, a_1, a_0> = a_n \cdot (-2^n + 1) + a_{n-1} \cdot 2^{n-1} + ... + a_1 \cdot 2^1 + a_0 \cdot 2^0$. Eine große Verbesserung gegenüber dem obigen Vorschlag ist aber nicht eingetreten: Wieder gibt es zwei Darstellungen von 0, nämlich (wieder am Beispiel vierstelliger Zahlen) 0000_2 und 1111_2. Dass auch die zweite Darstellung für „0“ steht, kann man aus den Stellenwerten nachrechnen: $1 \cdot (-2^3 + 1) + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = -7 + 4 + 2 + 1 = 0$. Man kann auch argumentieren, dass eine Zahl X plus ihr Einerkomlement $\overline{X}$ immer eine Folge von Einsen ergibt. Diese Folge von Einsen muss die Zahl 0 repräsentieren, da eine Zahl plus ihr negatives stets 0 ergeben muss. Auch das Hauptproblem bleibt: $0011_2 + 1101_2 = 0000_2$ durch stellenweise Addition (die fünfte Stelle, die sich als Übertrag ergeben würde, wird weggelassen) entspricht der Rechnung 3 + (–2) = 0, was wiederum leider falsch ist. Um in der Einerkomplementdarstellung richtige Additionen ausführen zu können, müssen wiederum Fallunterscheidungen getroffen werden.

Aufgabe 3.8 *Einerkomplement*
Es geht im Folgenden um vierstellige Dualzahlen in der Einerkomplementdarstellung.

1. *Wie viele verschiedene Zahlen lassen sich darstellen?*

2. *Wie lauten die kleinste und die größte darstellbare Zahl?*

3. *Stellen Sie die Zahlen 3 und –3 und 8 und –8 dar.*

Die heute übliche Methode, ganze Zahlen darzustellen, ist das Zweierkomplement. Hier ist bei nicht negativen Zahlen die Stelle ganz links null, die weiteren Stellen haben die üblichen Stellenwerte. Die Darstellung einer negativen Zahl erhält man, wenn man von der betragsgleichen positiven Zahl ausgeht, davon das Einerkomplement bildet und anschließend 1 addiert. Ein Übertrag nach der letzten Stelle wird dabei ignoriert. Es sollen wieder vierstellige Dualzahlen als Beispiel betrachtet werden. Es gilt: $3 = 0011_2$. Um die Zahl –2 in diese Darstellung zu bringen, geht man folgendermaßen vor: $+2 = 0010_2$. Das Einerkomplement dieser Darstellung ist 1101_2. Dazu muss nun 1 addiert werden, dies liefert 1110_2. Dass diese Umrechnung funktioniert, kann man sich folgendermaßen klar machen: Addiert man eine Dualzahl X und ihr Einerkomplement $\overline{X}$, so erhält man stets eine Folge von Einsen: 11...1. Addiert man dazu 1, so erhält man durch fortgesetzte Überträge stets 0, wobei der letzte Übertrag, der in einer zusätzlichen Stelle „1“ liefern würde, ignoriert wird. Als Stellenwerte ergeben sich $<\mathrm{a_n}, \mathrm{a_{n\text{-}1}}, \ldots, \mathrm{a_1}, \mathrm{a_0}> = a_n \cdot (-2^n) + a_{n-1} \cdot 2^{n-1} + ... + a_1 \cdot 2^1 + a_0 \cdot 2^0$. Man könnte damit die Zahl 1101_2 folgendermaßen ins Dezimalsystem umrechnen: $1101_2 = 1 \cdot (-2^3) + 1 \cdot 2^2 + 0 \cdot 2^1 + 2 \cdot 2^0 = -3$. Die Stellenwerte kann man natürlich auch für die Umrechnung in anderer Richtung benutzen.

Beim Zweierkomplement gibt es nur eine Darstellung der Zahl 0, nämlich $00...0_2$. Auch die Addition funktioniert ohne Probleme, solange das Ergebnis im darstellbaren Zahlenbereich bleibt. Die Rechnung 3 + (–2) = 1 lautet in Zweierkomplementdarstellung mit vier Stellen $0011_2 + 1110_2 = 0001_2$. Dies entspricht dem richtigen Ergebnis 1. Wegen dieser Vorzüge wird heute zur Darstellung ganzer Zahlen in Rechnern intern praktisch ausschließlich das Zweierkomplement benutzt.

Aufgabe 3.9 *Zweierkomplement*
Es geht im Folgenden um vierstellige Dualzahlen in Zweierkomlementdarstellung.

1. *Wie viele verschiedene Zahlen lassen sich darstellen?*

2. *Wie lauten die kleinste und die größte darstellbare Zahl?*

3. *Stellen Sie die Zahlen 3 und –3 und 8 und –8 dar.*

Beim Programmieren in Java gibt es unter anderem die Typen int und long für ganze Zahlen. Bei der Deklaration einer Variable vom Typ int werden 4 Byte = 32 bit für die

Speicherung des Werts im Speicher reserviert. Die Stellenwerte sind von rechts nach links 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32 768, 65536, 131072 usw. Die vorletzte (also 31.) Ziffer von links hat den Stellenwert $2^{30} = 1073741824$. Die letzte hat den Stellenwert $–2^{31} = –2147483648$. Die kleinste Zahl ist damit $10000000000000000000000000000000_2 = –2147483648$. Die größte ist $01111111111111111111111111111111_2 = 2^{31} –1 = 2147483647$.

Aufgabe 3.10 *Umrechnungsprogramm*

1. *Schreiben Sie ein kleines Java-Programm, das die Zahl 2147483640 einer Variable vom Typ int zuweist. In einer for-Schleife soll der Wert dieser Variable ausgegeben werden und die Variable um 1 erhöht werden. Die Schleife soll mindestens zehn Durchläufe machen. Was passiert, wenn die maximale durch eine Zahl vom Typ int darstellbare Zahl überschritten wird? Warum?*

2. *Möchte man in Java programmieren und dabei möglichst wenig Speicher belegen, kann man für sehr kleine ganze Zahlen den Datentyp „byte“ statt „int“ verwenden. Eine Variable des Typs byte beansprucht, wie der Name vermuten lässt, genau ein Byte. Was ist die kleinste und die größte durch byte darstellbare Zahl?*

3. *Für etwas größere Zahlen kann man den Datentyp „short“ benutzen. Der kleinste Wert, den eine Variable vom Typ short annehmen kann, ist –32768, der größte +32767. Wie viele bit, wie viele Byte Speicherplatz beansprucht eine Variable vom Typ short?*

4. *Eine Variable vom Typ long wird in 8 Byte abgespeichert. Was ist die kleinste und die größte Zahl, die in einer Variablen vom Typ long abgespeichert werden kann? (Hinweis: Die Aufgabe kann nur mit einem guten Taschenrechner gelöst werden!).*

1.6 Gleitkommazahlen

Rationale Zahlen können durch Dualbrüche wiedergegeben werden, wie im Abschnitt 1.3. Dies könnte man im Prinzip auch ausdehnen auf negative Zahlen. Die so erhaltenen Zahlen heißen Festkommazahlen. Sie sind aber unüblich. Im Normalfall werden rationale Zahlen durch Gleitkommazahlen dargestellt, da diese eine in etwa gleiche relative Genauigkeit über große Bereiche ermöglichen.

Bevor die Gleitkommazahlen im Dualsystem behandelt werden, sollen kurz Gleitkommazahlen im Dezimalsystem betrachtet werden, da diese vertrauter sind. Eine Gleitkommazahl ist z.B. $-4,711 \cdot 10^{-5}$. –4,711 heißt dabei Mantisse und –5 Exponent. Verschiedene Gleitkommazahlen können denselben Zahlenwert repräsentieren, so gilt z.B. $-4,711 \cdot 10^{-5} = -47,11 \cdot 10^{-6} = -0,4711 \cdot 10^{-4}$. Man normalisiert die Darstellung, indem man festlegt, dass es genau eine Stelle vor dem Komma gibt und diese ist nicht 0.

Die normalisierte Darstellung der obigen Zahl lautet also $-4,711 \cdot 10^{-5}$. Allgemein also: $(-1)^s x \cdot 10^{\pm y}$. Der Wert s kann 0 oder 1 sein und legt das Vorzeichen fest.

Im Dualsystem hat man dann entsprechend $(-1)^s \cdot 1,a \cdot 2^{\pm y}$. Die Mantisse hat immer eine 1 vor dem Komma stehen, denn bei der Normalisierung wurde festgelegt, dass die Ziffer vor dem Komma nicht 0 ist. Die einzige von 0 verschiedene Ziffer im Dualsystem ist aber 1. Problematisch ist, dass der Exponent vorzeichenbehaftet ist. Dies vermeidet man, indem man eine feste Zahl b vorgibt, wobei b üblicherweise um 1 kleiner ist als eine Zweierpotenz, z.B. 7. Dann kann die Gleitkommazahl dargestellt werden als $(-1)^s \cdot 1,a \cdot 2^{c-b}$. Um eine konkrete Zahl auszudrücken, muss man s, a und c angeben, denn b ist festgelegt. Für c müssen sinnvollerweise so viele Stellen reserviert werden, dass c etwa doppelt so groß werden kann wie b. Wenn b also beispielsweise 7 ist, dann wird man für c vier Stellen reservieren, um Zahlen zwischen 0 und 15 darstellen zu können. Damit kann der Exponent Werte zwischen –7 und +8 annehmen. Allerdings sind der kleinste Wert von c, also 0, und der größte Wert von c, im Beispiel also 15, für spezielle Zwecke reserviert, wie Darstellung der Zahl 0, $\pm\infty$, ungültige Zahl. Damit nimmt c effektiv nur die Werte von 1 bis 14 an, somit liegt der Exponent zwischen –6 und +7. Die Anzahl der Stellen von c bestimmt den Wertebereich einer Gleitkommazahl. Die Anzahl der Stellen von a bestimmt die Genauigkeit. Je mehr Stellen a hat, desto feiner ist die Unterteilung. Üblicherweise werden s, a und c in der Reihenfolge s, c, a angegeben.

Beispiel: Gegeben sei die achtstellige duale Gleitkommazahl 10011101 (siehe auch Tabelle 3.7). Die erste Stelle s legt das Vorzeichen fest, die nächsten vier Stellen c den Exponenten, die letzten drei die (Nachkommastellen der) Mantisse a. Da die vier Stellen von c 16 verschiedene Werte zulassen (effektiv 14, da der kleinste und größte reserviert ist), muss b = 7 festgelegt werden. Stelle Nummer 1 zeigt, dass es sich um eine negative

Tabelle 3.7: Beispiel Gleitkommazahl

Stelle	1	2	3	4	5	6	7	8
Ziffer	1	0	0	1	1	1	0	1
Bedeutung	Vorzeichen	c				a		

Zahl handelt. Stelle Nummer 6 sagt, dass die 1. Nachkommastelle der Mantisse mit dem Stellenwert 1/2 gesetzt ist, Nummer 7 sagt, dass die 2. Nachkommastelle mit dem Stellenwert 1/4 nicht gesetzt ist und Nummer 8 sagt, dass die dritte Nachkommastelle mit dem Stellenwert 1/8 gesetzt ist. Im Dezimalsystem lautet die Mantisse also 1,625. Der Wert von c in den Stellen 2 bis 5 beträgt $2^1 + 2^0 = 3$. Um daraus den Exponenten zu erhalten, muss man b abziehen, also 3 – 7 = –4. Nun ist $2^{-4} = 1/16$. Insgesamt ergibt sich also

$-1,625 \cdot 2^{-4} = -0,1015625 = -1,015625 \cdot 10^{-1}$ als Dezimalzahl.

Aufgabe 3.11 *Genauigkeit von Gleitkommazahlen*
Was ist die nächst kleinere Zahl, die sich in der geschilderten Form mit acht Stellen als duale Gleitkommazahl darstellen lässt? Was geschieht, wenn man eine Zahl dazwischen darstellen muss?

In Java kann man zur Darstellung von rationalen Zahlen die Datentypen „float“ oder „double“ verwenden. Eine Variable vom Typ float belegt vier Byte, also 32 bit. Von diesen benötigt man eines für das Vorzeichen, 23 für die (Nachkommastellen der) Mantisse, acht für den Exponenten. Der Wert von b beträgt 127. Damit lassen sich Daten im Bereich $\pm 3,4028 \cdot 10^{38}$ darstellen.

Der Typ double belegt acht Byte, das sind 64 bit, und zwar eines für das Vorzeichen, 52 für die (Nachkommastellen der) Mantisse und elf für den Exponenten. Der Wert von b beträgt 1023. Damit lassen sich Werte im Bereich $\pm 1,7977 \cdot 10^{308}$ darstellen. Darüber hinaus ist die Unterteilung genauer, da die Mantisse bei double länger ist als bei float.

2 Boolesche Algebra und Schaltnetze

2.1 Boolesche Funktionen von einer Variablen

Eine Funktion ordnet jedem Element aus einem Definitionsbereich D genau ein Element aus einem Wertebereich W zu. Die Elemente aus D nennt man Argumente, die zugeordneten Elemente aus W Funktionswerte.

In der Analysis betrachtet man Funktionen mit $D \subseteq \mathbb{R}$ und $W \subseteq \mathbb{R}$. Beispiel: f: $\mathbb{R} \to \mathbb{R}$ x $\mapsto x^3 - 2$. Oft auch geschrieben als $y = x^3 - 2$. Eine Funktion kann auch von mehreren Variablen abhängen, dann gilt also $D \subseteq \mathbb{R}^n$, z.B. f: $\mathbb{R}^2 \to \mathbb{R}$ (x, y) $\mapsto \sqrt{x^2 + y^2}$ oder $z = \sqrt{x^2 + y^2}$

Für Computer interessant sind Boolesche Funktionen. Hier spielt die Menge $B = \{0; 1\}$ eine große Rolle. B wird Binärraum genannt. Bei Booleschen Funktionen gilt immer W=B. Für den Definitonsbereich D gilt D=B^n mit $n \in \mathbb{N}^*$, im einfachsten Fall, nämlich n = 1, also D = B.

Da B und damit auch B^n für jedes endliche n nur endlich viele Elemente haben, gibt es für ein gegebenes n nur endlich viele Boolesche Funktionen. Aus demselben Grund kann man auch alle Zuordnungen aufzählen.

Alle Booleschen Funktionen einer Variablen sind in Tabelle 3.8 zu finden.

Die Funktionen werden nummeriert, indem man die Wertetabelle von unten nach oben als Dualzahl liest.[1]

Die Funktionen F_0 und F_3 sind eher uninteressant, da sie unabhängig vom Wert von x immer dasselbe ergeben. F_2 ist die Identität und als solche auch nicht besonderes inter-

[1]Die Nummerierung in der Literatur ist nicht einheitlich. Oft wird die Wertetabelle auch von oben nach unten gelesen und die sich so ergebende Dualzahl als Nummer der Funktion verwendet.

Tabelle 3.8: Alle Booleschen Funktionen einer Variablen

X	F_0	F_1	F_2	F_3
0	0	1	0	1
1	0	0	1	1

essant. Die einzig interessante Boolesche Funktion von einer Variablen ist F_1. F_1 ist die Boolesche Funktion, die als Funktionswert immer den anderen Wert liefert, also die Negation, da sie immer nicht den x-Wert liefert. Man schreibt für die Negation auch $F_1 = \neg x$ oder $F_1 = \bar{x}$ gelesen „nicht x“.

2.2 Boolesche Funktionen von zwei Variablen

Es gibt 16 Funktionen F_i von zwei Variablen X_1 und X_2:

Tabelle 3.9: Alle Booleschen Funktionen von zwei Variablen

$\mathbf{X_1}$	$\mathbf{X_2}$	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	**0**	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	**1**	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	**0**	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	**1**	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

16 Funktionen sind auf den ersten Blick eine ganze Menge, aber wenn man genauer hinschaut, sind die Booleschen Funktionen von zwei Variablen sehr überschaubar. Die Funktionen F_0 und F_{15} sind trivial, des weiteren sind die Funktionen F_{12} und F_3 eher uninteressant, da sie nur von X_1 abhängen. Es ist nämlich $F_{12} = X_1$ und $F_3 = \bar{X}_1$(der Überstrich bedeutet Negation). Genauso sind die Funktionen F_{10} und F_5 uninteressant, da sie in gleicher Weise nur von X_2 abhängen. Es bleiben zehn Funktionen, die eine nähere Untersuchung verdienen. Von diesen sind fünf die Negationen der anderen fünf.

F_{14} ist die ODER-Funktion, geschrieben $F_{14} = X_1 \vee X_2$ oder $F_{14} = X_1 + X_2$. Die Funktion heißt ODER-Funktion, weil die Variable X_1 **oder** X_2 gleich 1 sein muss, damit der Funktionswert 1 ist.

F_8 ist die UND-Funktion, geschrieben $F_8 = X_1 \wedge X_2$ oder $F_8 = X_1 \cdot X_2$ oder kurz $F_8 = X_1 X_2$. Die Funktion heißt UND-Funktion, weil beide Variablen X_1 **und** X_2 gleich 1 sein müssen, damit der Funktionswert 1 ist.

Aufgabe 3.12 *Negationen von UND und ODER*

1. *Die Negation der Oder-Funktion wird NOR-Funktion genannt, von englisch „not or“, übersetzt „nicht oder“ oder besser „negiertes Oder“. Welche der Funktionen aus der obigen Tabelle ist die NOR-Funktion?*

2. *Die Negation der UND-Funktion heißt entsprechend NAND-Funktion. Welche Funktion aus der Tabelle ist die NAND-Funktion?*

F_9 ist die Äquivalenzfunktion, geschrieben $X_1 \leftrightarrow X_2$. Sie ist genau dann 1, wenn beide Variablen denselben Wert haben. F_{11} ist die Funktion X_1 IMPLIZIERT X_2, geschrieben $X_1 \rightarrow X_2$. wenn X_1 gleich 1 ist, dann muss auch X_2 gleich 1 sein, damit der Funktionswert 1 ist. Für den Fall $X_1 = 1$ und $X_2 = 0$ ist der Funktionswert 0. F_{13} ist die Funktion x_2 IMPLIZIERT X_1, geschrieben $X_2 \rightarrow X_1$ .Wenn X_2 gleich 1 ist, dann muss auch X_1 gleich 1 sein.

Aufgabe 3.13 *Negationen verschiedener Funktionen*

1. *Finden Sie die Negationen zu den Funktionen F_{11} und F_{13}!*

2. *Finden Sie die Antivalenzfunktion $X_1 \not\leftrightarrow X_2$. Diese Funktion ist die Negation der Äquivalenzfunktion F_9.*

2.3 Gatter

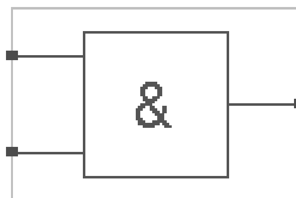


Abbildung 3.1: UND-Gatter

Elektronische Bauteile, die Boolesche Funktionen realisieren, nennt man *Gatter*. Abbildung 3.1 zeigt das Symbol für ein UND-Gatter. Es realisiert die UND-Funktion. Dabei sind die beiden Anschlüsse links die Eingänge. Sie entsprechen den Variablen, von denen die Funktion abhängt. Der Anschluss rechts ist der Ausgang. Er liefert den Funktionswert. Der Wert „0“ wird in der Elektronik durch „Spanung aus“ ausgedrückt, der Wert „1“ durch „Spannung ein“.

Das Symbol in Abbildung 3.2 steht für einen Inverter, auch Negationsglied genannt. Er realisiert die Negation. Abbildung 3.3 steht für ein ODER-Gatter. Das Größer-oder-gleich-Zeichen beim ODER-Gatter soll ausdrücken, dass dieses Gatter am Ausgang dann

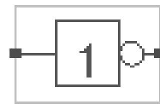


Abbildung 3.2: Inverter

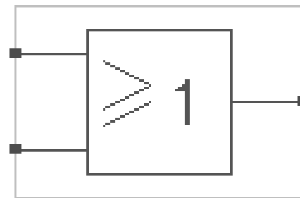


Abbildung 3.3: ODER-Gatter

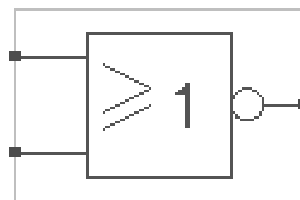


Abbildung 3.4: NOR-Gatter

das Signal „1“ liefert, wenn die Summe der Eingangssignale größer oder gleich 1 ist, wenn nämlich ein Eingangssignal „1“ ist oder beide Eingangssignale „1“ sind.

Abbildung 3.4 zeigt das Symbol des NOR-Gatters. Der Kreis am Ausgang steht für die Negation. Das Symbol des NAND-Gatters (Abbildung 3.5) sieht entsprechend aus.

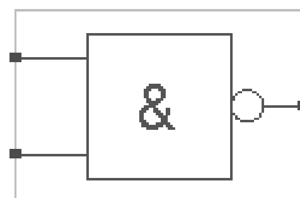


Abbildung 3.5: NAND-Gatter

Das Symbol eines Äquivalenzgatters ist in Abbildung 3.6 gezeigt. Dabei drückt das Gleichheitszeichen aus, dass der Ausgang genau dann das Signal „1“ liefert, wenn die Signale an den beiden Eingängen gleich sind.

Die Negation der Äquivalenzfunktion ist die Antivalenzfunktion. Ein Antivalenzgatter liefert genau dann am Ausgang den Wert „1“, wenn die beiden Eingangssignale unterschiedlich sind, also „0“ und „1“ oder „1“ und „0“. Man könnte die Antivalenzfunktion auch so ausdrücken: Entweder der erste Eingang ist „1“, oder der zweite Eingang ist „1“, nicht

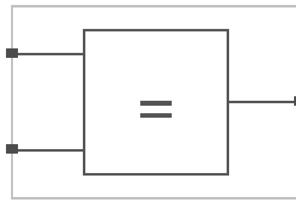


Abbildung 3.6: Äquivalenz-Gatter

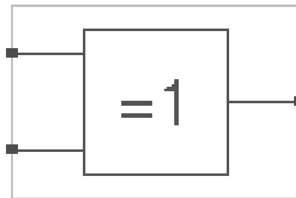


Abbildung 3.7: XOR

aber beide gleichzeitig, dann ist der Ausgang „1“. Daher sagt man statt „Antivalenzfunktion“ meistens „Exklusive ODER-Funktion“, kurz XOR. Statt von Antivalenzgatter spricht man meistens von einem XOR-Gatter. Dessen Schaltsymbol ist in Abbildung 3.7 zu sehen. „=1“ soll dabei ausdrücken, dass der Ausgang genau dann „1“ ist, wenn die Summe der Eingangssignale 1 ist.

Aufgabe 3.14 *Hades*
Besorgen Sie sich das Programm „Hades“, entweder von hier oder von den Seiten des Studienkollegs oder von der Moodle-Seite Ihres Kurses außerdem die Anleitung zu diesem Programm (Links wie oben). Bringen Sie das Programm auf Ihrem Rechner zum laufen. Es sollte durch einen einfachen Doppelklick starten. Das Programm dient dem Aufbau logischer Schaltungen aus Gattern und wird in den nächsten Wochen im Unterricht und für Hausaufgaben gebraucht werden.

2.4 Vollständige Operatorensysteme

Aufgabe 3.15 *Anzahl Boolescher Funktionen*

1. *Wie viele Boolesche Funktionen von drei Variablen gibt es?*

2. *Wie viele Boolesche Funktionen von vier Variablen gibt es?*

3. *Wie viele Boolesche Funktionen von n Variablen gibt es? $n \in N^*$*

Die Anzahl von Booleschen Funktionen von drei und mehr Variablen ist unüberschaubar. Man kann sie nicht alle einzeln kennen lernen wie die Funktionen von zwei Variablen.

Das ist aber auch nicht nötig, denn alle Booleschen Funktionen von drei und mehr Variablen lassen sich durch geeignete Kombinationen von Booleschen Funktionen von zwei Variablen darstellen. Auf den Beweis sei an dieser Stelle verzichtet. Man benötigt also zum Aufbauen von Schaltungen, die Boolesche Funktionen von beliebig vielen Variablen realisieren, nur Gatter mit zwei Eingängen. Dass es dennoch Gatter mit mehr als zwei Eingängen gibt, kann komfortabel sein, ist aber nicht unbedingt nötig.

Man benötigt aber nicht einmal alle Funktionen von zwei Variablen. So lassen sich mit der UND-Funktion, der ODER-Funktion und der Negation, einer Booleschen Funktion einer Variablen, alle Booleschen Funktionen von zwei Variablen darstellen und damit, nach dem oben gesagten, auch alle Boolesche Funktionen von n Variablen mit $n \in \mathbb{N}^*$. Man sagt, UND- ODER-Funktion und Negation bilden ein *vollständiges Operatorensystem*. Anders gesagt: Bei Schaltungen kommt man mit UND-, ODER-Gattern und Negationsgliedern (Inverter) aus.

Beispiel: Abbildung 3.8 zeigt eine Schaltung aus einem Negationsglied (Inverter) und einem ODER-Gatter. Diese Schaltung realisiert die Funktion F_{11}, X1 IMPLIZIERT X2. Die Elemente X1 und X2 sind Schalter in Hades, die die Eingänge repräsentieren, das Element F11 ist eine LED, die den Ausgang der Schaltung repräsentiert.

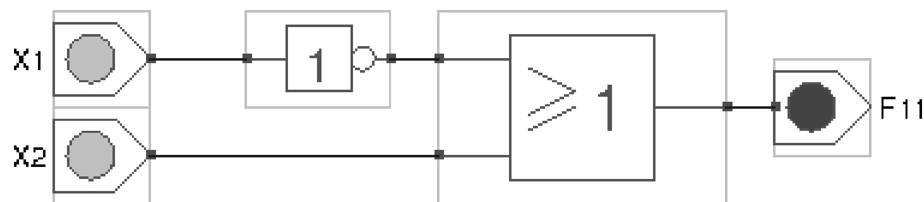


Abbildung 3.8: F11 aus NICHT und ODER

Aufgabe 3.16 *Vollständigkeit des UND-ODER-NICHT-Systems*
Verwenden Sie für diese Schaltungen nur UND-, ODER- und Negationsglieder!

1. *Bauen und testen Sie mit Hades Schaltungen, welche die Funktionen F_1 (NOR-Funktion), F_2 (Negation von X_2 IMPLIZIERT X_1), F_4 (Negation von X_1 IMPLIZIERT X_2), F_7 (NAND-Funktion) und F_{13} (X_2 IMPLIZIERT X_1) realisieren.*

2. *Bauen und testen Sie mit Hades Schaltungen, welche die Funktionen F_6 (Antivalenz-Funktion, auch XOR-Funktion genannt) und F_9 (Äquivalenzfunktion) realisieren.*

Das Operatorensystem, das nur aus der NAND-Funktion besteht, ist ebenfalls vollständig. Man könnte dies wiederum zeigen, indem man alle Booleschen Funktionen von zwei Variablen nur durch die NAND-Funktion darstellt, so wie dies im vorigen Aufgabenblock für das Operatorensystem UND-ODER-Negation gemacht wurde. Einfacher geht

es aber so: Wir wissen bereits, dass das UND-ODER-Negation-System vollständig ist. Wenn es gelingt zu zeigen, dass sich die UND-Funktion, die ODER-Funktion und die Negation allein durch NAND-Funktionen darstellen lassen, dann ist gezeigt, dass auch das NAND-System vollständig ist.

Aufgabe 3.17 *Vollständigkeit des NAND- und des NOR-Systems*

1. *Bauen Sie in Hades eine Negationsschaltung, eine UND-Schaltung und eine ODER-Schaltung auf und verwenden Sie dafür nur NAND-Gatter.*

2. *Zeigen Sie, dass auch das NOR-System vollständig ist.*

2.5 Umformungen Boolescher Ausdrücke

Für die folgenden Ausführungen bezeichne „+“ den ODER-Operator, „·“ den UND-Operator und der Überstrich die Negation, wie bereits in Abschnitt 2.2 dargelegt. Wie in der Arithmetik kann der Operator „·“ weggelassen werden, außerdem wird wie in der Arithmetik die Vereinbarung „Punkt vor Strich“ getroffen, d.h. im Ausdruck (AB) + C kann die Klammer weggelassen werden, nicht aber im Ausdruck A(B + C), wobei A, B und C Boolesche Variablen oder Ausdrücke sind. Dann gelten folgende Regeln für die Umformung Boolescher Ausdrücke. Die Liste ist nicht vollständig und kann dies auch gar nicht sein. Sie ist auch nicht minimal, d.h. einige der genannten Regeln lassen sich aus anderen herleiten.

Kommutativgesetze
1. $A \cdot B = B \cdot A$
2. $A + B = B + A$

Idempotenzgesetze
3. $A \cdot A = A$
4. $A + A = A$

Assoziativgesetze
5. $(A \cdot B) \cdot C = A \cdot (B \cdot C)$, daher schreibt man einfach $A \cdot B \cdot C$
6. $(A + B) + C = A + (B + C)$, daher schreibt man einfach $A + B + C$

Distributivgesetze
7. $A \cdot (B + C) = A \cdot B + A \cdot C$
8. $A + B \cdot C = (A + B) \cdot (A + C)$

Existenz der Identitätelemente
9. $1 \cdot A = A$
10. $0 + A = A$

Existenz der inversen Elemente
11. $A \cdot \bar{A} = 0$
12. $A + \bar{A} = 1$

Doppelte Negation
13. $\bar{\bar{A}} = A$

14. $0 \cdot A = 0$
15. $1 + A = 1$

De Morgansche Regeln
16. $\overline{A \cdot B} = \bar{A} + \bar{B}$
17. $\overline{(A + B)} = \bar{A} \cdot \bar{B}$

Verallgemeinerte De Morgansche Regeln
18. $\overline{\prod_{i=1}^{n} A_i} = \sum_{i=1}^{n} \bar{A}_i$
19. $\overline{\sum_{i=1}^{n} A_i} = \prod_{i=1}^{n} \bar{A}_i$

Die Regeln 1 bis 17 können durch die entsprechenden Wahrheitstafeln bewiesen werden, wenn man die Wahrheitstafeln für die UND-, ODER- und NICHT-Funktion vorgibt. Für die Regeln 18 und 19 braucht man darüber hinaus die Beweismethode der vollständigen Induktion, die Sie im Mathematik-Kurs lernen werden. Als Beispiel ist die Regel 16 in Tabelle 3.10 bewiesen. Man sieht, dass der Ausdruck links des Gleichheitszeichens in Regel 16 sich für jede mögliche Variablenbelegung genau gleich auswertet wie der Ausdruck rechts des Gleichheitszeichens, womit bewiesen ist, dass die beiden Ausdrücke gleich sind.

Der Inversionssatz von Shannon ist eine weitere Verallgemeinerung der verallgemeinerten De Morganschen Regeln. Hier kommen UND- und ODER-Operatoren gemischt vor: Gegeben sei ein Boolescher Ausdruck, der Negationen, UND- und ODER-Verknüpfungen enthält. Dieser Ausdruck kann negiert werden, indem man jedes nicht negierte Auftreten einer Variablen negiert und bei jedem negierten Auftreten einer Variable die Negation entfernt und alle ODER-Operatoren durch UND-Operatoren ersetzt und umgekehrt.

Weiter sind folgende Beziehungen von Nutzen: Durch die Schaltung in Abbildung 3.8 wurde gezeigt, dass gilt $A \to B = \bar{A} + B$. Außerdem gilt für die Äquivalenzfunktion $A \leftrightarrow B = AB + \bar{A}\bar{B}$ und für die Antivalenzfunktion $A \not\leftrightarrow B = \bar{A}B + A\bar{B}$.

Lernen Sie die Regeln nicht stumpf auswendig. Das Können der Regeln kommt mit ihrer Verwendung. Nur die De Morganschen Regeln und die Regel 8 sollten sie explizit lernen, da sie nicht offensichtlich sind. Regel 8 ist so unangenehm, weil die Regel in der Arithmetik, dem Rechnen mit Zahlen **nicht** gilt!

Tabelle 3.10: Beweis von Regel 16

Variablenbelegung		linke Seite		rechte Seite		
			Auswertung linke Seite			Auswertung rechte Seite
A	B	A·B	$\overline{AB}$	$\bar{A}$	$\bar{B}$	$\bar{A}+\bar{B}$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

Hier ein Beispiel, wie eine zunächst kompliziert aussehende Boolesche Funktion mittels der obigen Regeln vereinfacht werden kann. Über den Gleichheitszeichen stehen die Nummern der verwendeten Regeln.

$F(A,B,C)$
$= (A+B)(C+\bar{A})(C+\bar{C})B+\bar{\bar{C}}\bar{C}$
$\stackrel{13}{=} (A+B)(C+\bar{A})(C+\bar{C})B+C\bar{C}$
$\stackrel{11}{=} (A+B)(C+\bar{A})(C+\bar{C})B+0$
$\stackrel{2}{=} 0+(A+B)(C+\bar{A})(C+\bar{C})B$
$\stackrel{10}{=} (A+B)(C+\bar{A})(C+\bar{C})B$
$\stackrel{12}{=} (A+B)(C+\bar{A})\cdot 1\cdot B$
$\stackrel{1}{=} 1\cdot(A+B)(C+\bar{A})B$
$\stackrel{9}{=} (A+B)(C+\bar{A})B$
$\stackrel{1}{=} B(A+B)(C+\bar{A})$
$\stackrel{7}{=} (BA+BB)(C+\bar{A})$
$\stackrel{3}{=} (BA+B)(C+\bar{A})$
$\stackrel{9}{=} (BA+1B)(C+\bar{A})$
$\stackrel{1}{=} (BA+B1)(C+\bar{A})$
$\stackrel{7}{=} B(A+1)(C+\bar{A})$
$\stackrel{2}{=} B(1+A)(C+\bar{A})$
$\stackrel{15}{=} B\cdot 1(C+\bar{A})$

$\overset{1}{=} 1 \cdot B(C + \bar{A})$
$\overset{9}{=} B(C + \bar{A})$
$\overset{7}{=} BC + B\bar{A}$
$\overset{1}{=} BC + \bar{A}B$
$\overset{2}{=} \bar{A}B + BC$

Der letztlich erhaltene Funktionsausdruck ist recht einfach. Außer auf Kürze wurde auch auf die alphabetische Reihenfolge der Variablen geachtet. Die ausführliche Umformung soll die Nachvollziehbarkeit erhöhen. Der Fortgeschrittene wird mehrere Umformungen gleichzeitig durchführen.

2.6 Kanonische Normalformen

Definition: Ein Boolescher Ausdruck, der keine anderen Verknüpfungen enthält als UND-Verknüpfungen, heißt *Konjunktionsterm*. Beispiele: $\bar{X}_0X_1X_2X_3$ ist ein Konjunktionsterm, nicht aber $X_0X_1X_3 + X_4$, da hier auch eine ODER-Verknüpfung enthalten ist. Auch $\overline{X_0X_1}X_2X_3$ ist kein Konjunktionsterm, da hier die NAND-Verknüpfung $\overline{X_0X_1}$ enthalten ist.

Definition: Ein Boolescher Ausdruck, der keine anderen Verknüpfungen enthält als ODER-Verknüpfungen, heißt *Disjunktionsterm*. Beispiel: $\bar{X}_0 + X_1 + X_2 + X_3$ ist ein Disjunktionsterm, nicht aber $X_0X_1 + X_3 + X_4$, da hier auch eine UND-Verknüpfung enthalten ist.

Aufgabe 3.18 *Konjunktionsterme und Disjunktionsterme*

1. *Gibt es Boolesche Ausdrücke, die gleichzeitig Konjunktionsterm und Disjunktionsterm sind?*

2. *Ist* $\overline{X_0 + X_1} + X_2 + X_3$ *ein Disjunktionsterm?*

Definition: Ein Boolescher Ausdruck, der eine Disjunktion (ODER-Veknüpfung) von Konjunktionstermen ist, heißt *disjunktive* Normalform, kurz *DNF*. Beispiel: $A\bar{C}D + \bar{A}\bar{B}CD + ABC\bar{D}$.

Definition: Ein Boolescher Ausdruck, der eine Konjunktion (UND-Verknüpfung) von Disjunktionstermen ist, heißt *konjunktive Normalform*, kurz *KNF*. Beispiel: $(A+B+C+D)(A+\bar{B}+\bar{C})(A+\bar{D})$

Jede Boolesche Funktion lässt sich als DNF schreiben. Jede Boolesche Funktion lässt sich als KNF schreiben. Dies wird weiter unten klar werden. Es gibt von einer Booleschen Funktion immer mehrere Darstellungen als DNF und mehrere Darstellungen als KNF. Dies kann man sich leicht klar machen: Die Funktionen $F_1 = A\bar{C}D + \bar{A}\bar{B}CD + ABC\bar{D}$ und $F_2 = A\bar{C}D + \bar{A}\bar{B}CD + ABC\bar{D} + A\bar{A}B$ sind gleich, da der letzte Konjunktionsterm in F_2 stets 0 liefert und daher keinen Einfluss hat.

Aufgabe 3.19 *KNF ist nicht eindeutig*
Geben Sie zwei verschiedene KNFs an, welche die gleiche Boolesche Funktion repräsentieren.

Definition: Ein *Minterm* ist ein Konjunktionsterm, in dem alle n Variablen einer betrachteten Booleschen Funktion genau einmal auftreten. Beispiel: Es sollen Boolesche Funktionen der Variablen A, B, C und D betrachtet werden. Dann ist $A\bar{B}CD$ ein Minterm, nicht aber $AC\bar{D}$, da die Variable B nicht vorkommt. $\bar{A}BC\bar{C}D$ ist kein Minterm, weil die Variable C zweimal vorkommt.

Definition: Eine DNF, deren Konjunktionsterme ausschließlich Minterme sind und in der kein Minterm mehrmals vorkommt, heißt *kanonische disjunktive Normalform*, kurz *KDNF*.

Von jeder Booleschen Funktion lässt sich eine Darstellung als KDNF erzeugen. Zwei KDNF der gleichen Booleschen Funktion können sich nur in der Reihenfolge der Minterme und der Reihenfolge der Variablen innerhalb der Minterme unterscheiden. Man sagt daher, die KDNF ist eindeutig bis auf[2] Vertauschung.

Beispiele: Es werden Boolesche Funktionen der drei Variablen A, B und C betrachtet. Die Funktion $F_1 = A\bar{B}\bar{C} + A\bar{B}C + AB\bar{C}$ ist in kanonischer disjunktiver Normalform. Die Funktion $F_2 = A\bar{B}\bar{C} + A\bar{B}C + AB\bar{C} + \bar{B}AC$ ist gleich wie die Funktion F_1. F_2 ist aber nicht in KDNF, denn der zweite und der vierte Minterm sind bis auf Vertauschung gleich. Daher enthält diese Darstellung denselben Minterm zweimal. Die Funktion $F_3 = A\bar{B}C + AB\bar{C} + \bar{B}\bar{C}A$ ist in KDNF. Sie beschreibt dieselbe Funktion wie F_1 (und damit auch wie F_2), denn F_3 lässt sich aus F_1 durch Vertauschungen erzeugen.

Die Darstellung als KDNF kann dazu genutzt werden, zwei Boolesche Funktionen auf Gleichheit zu überprüfen. Dafür werden beide Funktionen auf KDNF gebracht. Stimmen die beiden Darstellungen dann bis auf Vertauschungen überein, so handelt es sich um gleiche Funktionen. Ist dies nicht der Fall, so sind die Funktionen ungleich.

Konstruktion der KDNF: Die KDNF einer Booleschen Funktion lässt sich aus deren Wahrheitstafel konstruieren. Zu jeder „1“ in der Wahrheitstafel gehört ein Minterm. In diesem taucht jede mit „0“ belegte Variable negiert auf, jede mit „1“ belegte Variable ohne Negation. Die Disjunktion aller Minterme ergibt die KDNF. Beispiel: Gegeben ist eine Boolesche Funktion F von den drei Variablen A, B und C durch ihre Wahrheitstafel in Tabelle 3.11

Zu der ersten „1“ gehört der Minterm $\bar{A}\bar{B}\bar{C}$. Zur zweiten „1“ gehört der Minterm $\bar{A}\bar{B}C$. Zur dritten „1“ gehört der Minterm ABC. Die Funktion F hat also die Darstellung als KDNF $F = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + ABC$.

Definition: Ein *Maxterm* ist ein Disjunktionsterm, in dem alle n Variablen einer betrachteten Booleschen Funktion genau einmal auftreten. Beispiel: Es sollen Boolesche Funktionen der Variablen A, B, C und D betrachtet werden. Dann ist $A + \bar{B} + C + D$ ein

[2]bis auf, englisch except for, französisch sauf. Bitte finden Sie heraus, was „bis auf“ in Ihrer Muttersprache heißt!

Tabelle 3.11: Beispielfunktion für die KDNF

A	B	C	F
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Maxterm, nicht aber $A + C + \bar{D}$, da die Variable B nicht vorkommt. $\bar{A} + B + C + \bar{C} + D$ ist kein Maxterm, weil die Variable C zweimal vorkommt.

Definition: Eine KNF, deren Disjunktionsterme ausschließlich Maxterme sind und in der kein Maxterm mehrmals vorkommt, heißt *kanonische konjunktive Normalform*, kurz *KKNF*.

Von jeder Booleschen Funktion lässt sich eine Darstellung als KKNF erzeugen. Zwei KKNF der gleichen Booleschen Funktion können sich nur in der Reihenfolge der Maxterme und der Reihenfolge der Variablen innerhalb der Maxterme unterscheiden. Man sagt daher, die KKNF ist eindeutig bis auf Vertauschung. Die Darstellung einer Booleschen Funktion als KKNF kann genauso wie die Darstellung als KDNF genutzt werden, um die Gleichheit zweier Funktionen zu prüfen.

Konstruktion der KKNF: Es soll die KKNF gefunden werden von derjenigen Booleschen Funktion, die bereits als Beispiel für die Konstruktion einer KDNF gedient hat. Deren Wahrheitstafel ist in Tabelle 3.12 hier noch einmal gegeben, ergänzt um die Spalte $\bar{F}$, der Negation von F.

Zuerst wird die KDNF von $\bar{F}$ erstellt. Diese lautet $\bar{F} = \bar{A}B\bar{C} + \bar{A}BC + A\bar{B}\bar{C} + A\bar{B}C + AB\bar{C}$. Nach Regel 19 der Umformungen gilt $\overline{\sum_{i=1}^{n} A_i} = \prod_{i=1}^{n} \bar{A}_i$. Angewandt auf $\bar{F}$ ergibt sich $\bar{\bar{F}} = \overline{(\bar{A}B\bar{C})} \cdot \overline{(\bar{A}BC)} \cdot \overline{(A\bar{B}\bar{C})} \cdot \overline{(A\bar{B}C)} \cdot \overline{(AB\bar{C})}$. Benutzt man Regel 13 $\bar{\bar{F}} = F$ und Regel 18 $\overline{\prod_{i=1}^{n} A_i} = \sum_{i=1}^{n} \bar{A}_i$ der Umformungen, so erhält man $F = (\overline{\bar{A}} + \bar{B} + \bar{\bar{C}})(\bar{\bar{A}} + \bar{B} + \bar{C})(\bar{A} + \bar{\bar{B}} + \bar{\bar{C}})(\bar{A} + \bar{\bar{B}} + \bar{C})(\bar{A} + \bar{B} + \bar{\bar{C}})$. Wendet man nun abermals Regel 13 an, um die vielen doppelten Negationen los zu werden, so erhält man $F = (A + \bar{B} + C)(A + \bar{B} + \bar{C})(\bar{A} + B + C)(\bar{A} + B + \bar{C})(\bar{A} + \bar{B} + C)$. Dies ist die gesuchte KKNF von F.

Tabelle 3.12: Beispielfunktion für die KKNF

A	B	C	F	$\bar{F}$
0	0	0	1	0
0	0	1	1	0
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	0

Der Weg, um von der KDNF von $\bar{F}$ auf die KKNF von F zu kommen, lässt sich auch sehr kurz beschreiben: Wendet man auf die KDNF von $\bar{F}$ den Inversionssatz von Shannon an, so erhält man die KKNF von F.

Aufgabe 3.20 *Länge von KDNF und KKNF*
Bei der Booleschen Funktion, die als Beispiel diente für die Konstruktion von KDNF und KKNF, war die KDNF kürzer als die KKNF. Ist das bei jeder Booleschen Funktion so?

2.7 Minimale Normalformen und Karnaugh-Diagramme

Kanonische Normalformen sind gut geeignet, um die Gleichheit zweier Boolescher Ausdrücke zu überprüfen. Da sie aber sehr lang sind, sind sie jedoch unhandlich. Es ist aufwändig, eine Boolesche Funktion in ihrer Darstellung als kanonische Normalform als Schaltung aufzubauen. Andererseits ist die Formulierung als DNF oder KNF recht übersichtlich. Zum Aufbau von Schaltungen geht man daher gerne von möglichst kurzen DNF oder KNF aus. Eine kürzest mögliche Darstellung einer Booleschen Funktion als DNF oder KNF wird *minimale DNF* bzw. *minimale KNF* genannt.

Gehen wir wieder von der Beispielfunktion aus Tabelle 3.11 aus.

Als KDNF wurde gefunden $F = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + ABC$. Der erste und der zweite Konjunktionsterm unterscheiden sich nur im Auftreten der Variablen C. Dies gibt Anlass zur Anwendung von Regel 7 der Umformungen. Man erhält $F = (\bar{A}\bar{B})(\bar{C} + C) + ABC$. Mit den Regeln 12, 1 und 9 kommt man schließlich auf $F = \bar{A}\bar{B} + ABC$. Dies ist eine minimale DNF von F. Die Funktion F hat keine andere minimale DNF.

Wenn man ausgehend von der KDNF eine minmale DNF sucht, kann es passieren, dass Regel 7 auf verschiedene Paare von Konjunktionstermen angewandt werden kann.

Es kann dann sein, dass man zum Schluss auf die gleiche minimale DNF kommt, oder aber man kommt auf verschiedene minimale DNF. Es kann aber auch passieren, dass man eine DNF findet, die sich nicht weiter vereinfachen lässt, die aber nicht minimal ist. Daher braucht man Verfahren, die einen sicher zu allen möglichen minimalen DNF einer Booleschen Funktion führen. Ein solches Verfahren nutzt Karnaugh-Diagramme. Das Karnaugh-Diagramm für die Beispielfunktion sieht so aus:

A (rechte zwei Spalten), C (mittlere zwei Spalten), B (untere Zeile)

	1 [0]	1 [1]	0 [5]	0 [4]
B	0 [2]	0 [3]	1 [7]	0 [6]

Die rechten vier Kästchen stehen für die Terme mit A, die linken vier Kästchen für die Terme mit $\bar{A}$. Die unteren vier Kästchen stehen für die Terme mit B, die unteren vier Kästchen für die Terme mit $\bar{B}$. Die mittleren vier Kästchen stehen für die Terme mit C, die äußeren vier Kästchen für die Terme mit $\bar{C}$.

Aufgabe 3.21 *Minterme im Karnaugh-Diagramm*
Man kann sagen, im Karnaugh-Diagramm steht jedes Kästchen für einen Minterm. Finden Sie jedes Kästchen zu allen Mintermen
$\bar{A}\bar{B}\bar{C}, \bar{A}\bar{B}C, \bar{A}B\bar{C}, \bar{A}BC, A\bar{B}\bar{C}, A\bar{B}C, AB\bar{C}, ABC$

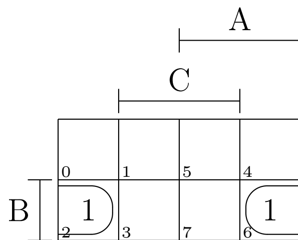
Nun geht es darum, Zweiergruppen nebeneinander oder übereinander zu finden, bei denen beide Kästchen mit einer „1" besetzt sind. Im Beispiel gibt es eine solche Zweiergruppe:

A (rechte zwei Spalten), C (mittlere zwei Spalten), B (untere Zeile); Zweiergruppe umrandet: Kästchen 0 und 1

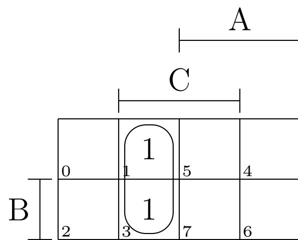
	(1 [0]	1) [1]	0 [5]	0 [4]
B	0 [2]	0 [3]	1 [7]	0 [6]

In dieser Zweiergruppe gilt $\bar{A}$ und $\bar{B}$, während die Variable C einmal negiert, einmal ohne Negation auftaucht. Die Variable C fällt daher in der Beschreibung dieser Zweiergruppe weg. Diese Zweiergruppe repräsentiert den Term $\bar{A}\bar{B}$. In der minimalen DNF von F muss aber auch die einzelne „1" auftauchen, daher ergibt sich $F = \bar{A}\bar{B} + ABC$. Dieses Ergebnis wurde oben bereits gefunden

Bei den Karnaugh-Diagrammen von drei Variablen muss man sich den linken mit dem rechten Rand verbunden vorstellen. Man müsste das Karnaugh-Diagramm also anstatt auf ein flaches Papier auf einen Zylindermantel („Röhre") aufzeichnen. Daher ist auch eine solche Zweiergruppe ist erlaubt, die den Term $B\bar{C}$ repräsentiert:

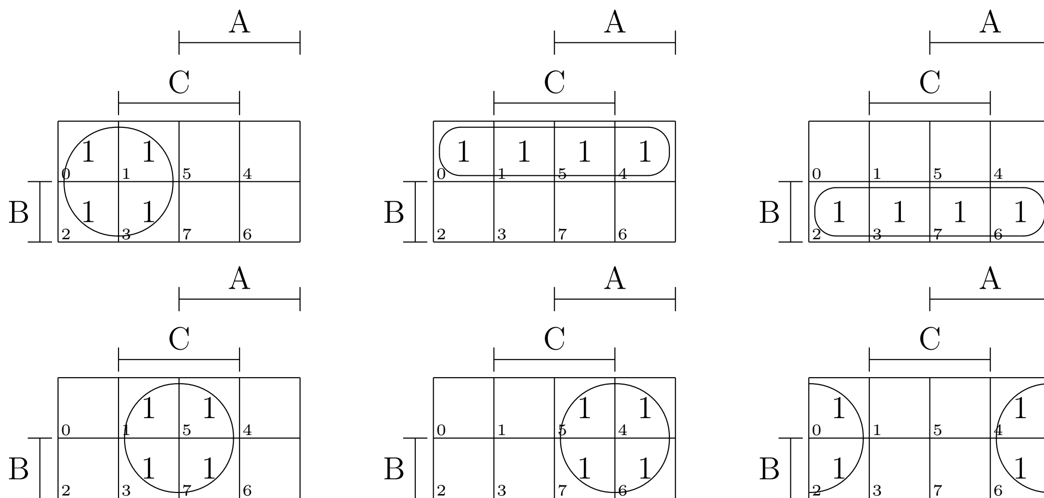


Eine Zweiergruppe kann auch senkrecht angeordnet sein. Die Zweiergruppe im folgenden Beispiel repräsentiert den Term $\bar{A}C$.



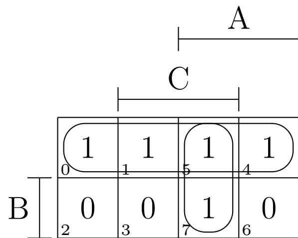
Statt Zweiergruppen können auch Vierergruppen, entweder als Block oder als Zeile, gefunden werden, in denen alle Kästchen mit einer „1“ besetzt sind. Diese stehen für einen Term mit nur einer Variablen.

Statt Zweiergruppen kann man auch Vierergruppen finden. Die Vierergruppen repräsentieren Terme aus nur einer Variablen, also $A, B, C, \bar{A}, \bar{B}$, oder $\bar{C}$. Die Vierergruppen sehen so aus:



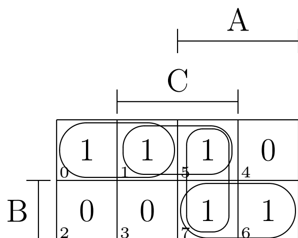
Aufgabe 3.22 *Terme einer Variablen bei Funktionen von drei Variablen*
Ordnen Sie die Vierergruppen den richtigen Termen aus einer Variablen zu.

Die minimale DNF der Funktion, die durch dieses



Karnaugh-Diagramm gegeben ist, hat die minimale DNF $F = \bar{B} + AC$. Beachten Sie, dass die „1“ aus dem Minterm 7, ABC, nicht einzeln genommen wird, da man dann drei Variablen bräuchte und die DNF lauten würde $F = \bar{B} + ABC$. Das ist aber länger als die oben gegebene DNF. $F = \bar{B} + ABC$ ist also eine DNF, sie ist aber nicht minimal.

Das folgende Beispiel weist eine zusätzliche Schwierigkeit auf.



Man kann keine Vierergruppe finden. Man findet vier Zweiergruppen, die die Terme $\bar{A}\bar{B}$, $\bar{B}C$, AC und AB repräsentieren. Die DNF $F = \bar{A}\bar{B} + \bar{B}C + AC + AB$ ist aber nicht minimal, denn die beiden Einsen des Terms $\bar{B}C$ stecken bereits in den Termen $\bar{A}\bar{B}$ und AC. Daher wird dieser Term nicht benötigt. Eine minimale DNF lautet daher: $F = \bar{A}\bar{B} + AC + AB$.

Genauso gut kann man aber auch den Term AC weglassen, muss dann den Term $\bar{B}C$ aufnehmen. Daraus ergibt sich die DNF $F = \bar{A}\bar{B} + \bar{B}C + AB$. Diese enthält auch drei Konjuktionsterme mit je zwei Variablen, ist also genau gleich lang wie die vorher gefundene minimale DNF. Das heißt, auch diese DNF ist minimal.

Die Darstellung einer Funktion als minimale DNF ist also nicht eindeutig: Bei manchen Funktionen lassen sich mehrere minimale DNF finden. Das gleiche gilt für die minmalen KNF. Im Gegensatz dazu sind KDNF und KKNF einer Funktion (bis auf Vertauschung) eindeutig.

Ein Karnaugh-Diagramm für eine Boolesche Funktion von vier Variablen sieht zum Beispiel so aus:

			D	D, B	B
		1 (0)	0 (1)	0 (5)	0 (4)
	C	1 (2)	0 (3)	1 (7)	1 (6)
A	C	1 (10)	1 (11)	1 (15)	1 (14)
A		1 (8)	0 (9)	0 (13)	0 (12)

Dabei muss man sich wieder den linken mit dem rechten Rand verbunden vorstellen, außerdem den unteren mit dem oberen Rand. Das Diagramm müsste also anstatt auf ein flaches Papier auf einen Torus („Ring“) aufgezeichnet werden. Außer Zweier- und Vierergruppen kann es jetzt auch Achtergruppen geben, nämlich die obere, untere, linke oder rechte Hälfte, oder die beiden mittleren Zeilen oder die beiden mittleren Spalten. Einzelne Einsen stehen für Terme mit vier Variablen, Zweiergruppen für Terme mit drei Variablen, Vierergruppen für Terme mit zwei Variablen, Achtergruppen für Terme mit einer Variablen.

Aufgabe 3.23 *Minimale DNF und minimale KNF*
Wie viele minimale DNF gibt es für die Funktion, die durch das obige Karnaugh-Diagramm definiert ist? Wie lautet diese bzw. wie lauten diese?

Karnaugh-Diagramme für fünf Variablen erhält man, wenn man in der dritten Dimension weiter baut. Man hat also zwei Karnaugh-Diagramme für vier Variablen übereinander. Dabei steht eine Ebene für die fünfte Variable E, die andere für diese Variable negiert, $\bar{E}$.

Ein Karnaugh-Diagramm für sechs Variablen besteht aus vier solcher Ebenen. Die beiden mittleren stehen dabei für die Variable E, die beiden unteren für die sechste Variable F. Man muss sich die Unterseite mit der Oberseite verbunden vorstellen.

Spätestens bei sieben und mehr Variablen sind Karnaugh-Diagramme nicht mehr geschickt. Man muss dann auf andere Verfahren zur Erstellung minimaler DNF zurück greifen, z.B. das Quine-McClusky-Verfahren. In der Praxis werden heute aber solche Aufgaben ohnehin durch Computerprogramme erledigt. Kein Informatiker oder Elektrotechniker entwirft heute noch minimale DNF für komplizierte Funktionen mit Papier und Bleistift.

Aufgabe 3.24 *Minimale KNF*
Entwickeln Sie ein Verfahren, um alle minimalen KNF einer Booleschen Funktion zu finden. Gehen Sie ähnlich vor wie bei der KKNF: Erstellen Sie eine minimale DNF für

$\bar{F}$. Dabei ist ein Karnaugh-Diagramm hilfreich. Aus der minimalen DNF von $\bar{F}$ kann dann mit Hilfe des Inversionssatzes von Shannon eine minimale KNF von F erzeugt werden.

2.8 Ein Beispiel mit Don't-Cares

An einer Straße soll eine Ampelanlage aufgestellt werden, damit Fußgänger sicher die Straße überqueren können. Dafür sind fünf Ampellichter zu steuern: Grün und Rot auf der Fußgängerampel und Grün, Gelb und Rot auf der Autoampel. Alle Lichter werden von demselben Taktgeber gesteuert. Der Taktgeber beginnt bei 0, zählt alle vier Sekunden um eins weiter bis 12 und beginnt dann wieder bei 0. Während der Taktzeiten 0, 1, 2 und 3 ist die Fußgängerampel grün, sonst, d.h. bei 4 bis 12, ist sie rot.

Tabelle 3.13: Fußgängerampel in Abhängigkeit des Takts

Takt	A	B	B	D	F_1	F_2
0	0	0	0	0	1	0
1	0	0	0	1	1	0
2	0	0	1	0	1	0
3	0	0	1	1	1	0
4	0	1	0	0	0	1
5	0	1	0	1	0	1
6	0	1	1	0	0	1
7	0	1	1	1	0	1
8	1	0	0	0	0	1
9	1	0	0	1	0	1
10	1	0	1	0	0	1
11	1	0	1	1	0	1
12	1	1	0	0	0	1

Die Autoampel zeigt bei Taktzeit 5 rot und gelb gleichzeitig, bei 6, 7, 8, 9 und 10 zeigt sie grün, bei 11 zeigt sie gelb, bei 12, 0, 1, 2, 3 und 4 zeigt sie rot. Der Taktgeber hat die vier Ausgänge A, B, C und D, die für die Stellenwerte $2^3 = 8$, $2^2 = 4$, $2^1 = 2$ bzw. $2^0 = 1$ stehen. Der Taktgeber zählt also von 0000_2 bis 1100_2 und beginnt dann wieder von vorne. Die Ausgänge des Taktgebers sind die Eingänge der fünf Schaltnetze, welche die Ampellichter steuern.

Das grüne Licht der Fußgängerampel soll durch die Boolesche Funktion F_1 repräsentiert werden, das rote Licht der Fußgängerampel durch die Boolesche Funktion F_2. Diese hängen wie in der Tabelle 3.13 gezeigt von den Variablen A, B, C und D ab.

Stellt man das zu F_1 gehörende Karnaugh-Diagramm auf, so stellt sich die Frage, was bei den drei Positionen eingetragen werden soll, die nicht vergeben sind. Hier wird ein „–“ oder ein X eingetragen.

F_1

			D	D, B	B
		1 (0)	1 (1)	0 (5)	0 (4)
	C	1 (2)	1 (3)	0 (7)	0 (6)
A	C	0 (10)	0 (11)	X (15)	X (14)
A		0 (8)	0 (9)	X (13)	0 (12)

Beim Finden von Gruppen kann ein solches Zeichen wahlweise als 0 oder als 1 betrachtet werden, je nachdem, was von Vorteil ist, um wenige, große Gruppen zu erhalten. Da es für die Richtigkeit der erhaltenen Gleichung egal ist, ob man ein „X“ nun als „0“ oder „1“ auffasst, nennt man dieses Zeichen ein „Don't-Care“ von englisch „macht nichts“. Hier ist es offenkundig von Vorteil, alle Don't-Cares als „0“ zu betrachten, um eine minimale DNF von F_1 zu erhalten. Diese lautet dann $F_1 = \bar{B}\bar{A}$.

Beim Karnaugh-Diagramm von F_2 ist es günstiger, alle „X“ als „1“ aufzufassen. Man erhält dann $F_2 = AB$.

F_2

			D	D, B	B
		0 (0)	0 (1)	1 (5)	1 (4)
	C	0 (2)	0 (3)	1 (7)	1 (6)
A	C	1 (10)	1 (11)	X (15)	X (14)
A		1 (8)	1 (9)	X (13)	1 (12)

Um die Steuerung für die Fußgängerampel zu bauen, kann man aber auch auf separate Schaltungen für F_1 und F_2 verzichten. Statt dessen kann man die einfachere von den beiden, das ist F_2 bauen, und sich für F_1 zunutze machen, dass gilt $F_1 = \bar{F}_2$. Damit kann man, wenn man F_2 hat, F_1 mit einem einzigen Schaltelement zusätzlich, nämlich einem Inverter, erhalten.

Aufgabe 3.25 *Ampel*
Finden Sie einfache Schaltungen für die drei Lichter der Autoampel, indem Sie die Karnaugh-Diagramme zeichnen und die Don't-Cares geschickt einbeziehen.

2.9 Der Carry-Ripple-Addierer

Um zu demonstrieren, dass man Boolesche Algebra und damit digitale elektronische Bausteine zum Rechnen, also zum Aufbau eines Computers brauchen kann, soll eine Schaltung zum Addieren von Zahlen aufgebaut werden. Die Summanden sollen dabei als Dualzahlen vorliegen, das Ergebnis wird als Dualzahl präsentiert. In realen Rechenmaschinen, z.B. Ihrem Taschenrechner, sind weitere Komponenten eingebaut, welche die Ein- und Ausgabe in den vertrauten Dezimalzahlen ermöglichen.

Tabelle 3.14: Wertetabelle eines Halbaddierers

1. Summand A_0	2. Summand B_0	Übetrag $Ü_0$	Stelle S_0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Um zwei Dualzahlen zu addieren, braucht man zunächst eine Schaltung, um die Einerstellen der beiden Summanden zu addieren. Diese Schaltung braucht zwei Eingänge, je einen für die Einerstelle jedes Summanden. Es können folgende Fälle auftreten: $0+0=0$, $0+1=1$, $1+0=1$ und $1+1=10_2$. Die Schaltung muss also zwei Ausgänge haben, je eine für jede Stelle, da es möglich ist, dass das Ergebnis zweistellig ist. Bei den eigentlich einstelligen Ergebnissen muss die linke Stelle dann eine führende Null sein, also $0+0=00$, $0+1=01$ und $1+0=01$. Die Schaltung realisiert also zwei Boolesche Funktionen von zwei Variablen. Die Funktionen sollen „S“ für die Stelle und „Ü“ für den Übertrag heißen. Die Wertetabelle der beiden Funktionen ist in Tabelle 3.14 angegeben. Der Index „0“ soll dabei bedeuten, dass es sich um die Stelle mit dem Wer 2^0, also um die Einerstelle, handelt. Es muss untersucht werden, welche Funktionen das sind. Die Einerstelle des Summanden, also S, ist 0, wenn die Einerstellen der beiden Summanden beide 0 sind oder beide 1 sind. Die Einerstelle ist 1, wenn die Einerstellen der beiden Summanden verschieden sind. Dies ist die Antivalenzfunktion F_6. Die Berechnung der Einerstelle wird daher durch ein Antivalenzgatter realisiert. Die Zweierstelle der Summe, also Ü, ist nur dann 1, wenn die Einerstellen des ersten *und* des zweiten Summanden 1 sind. Dies ist die

UND-Funktion F_8. Die Schaltung für die Addition der Einerstellen sieht also aus, wie in Abbildung 3.9 gezeigt.

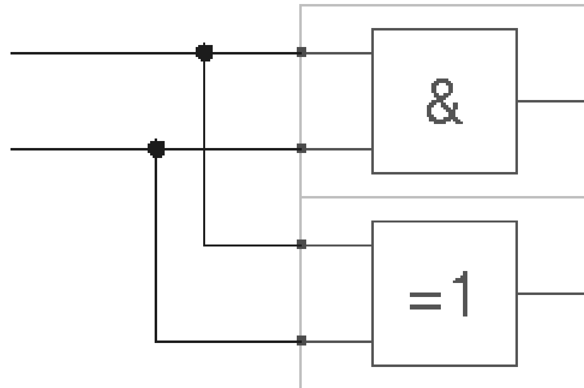


Abbildung 3.9: Halbaddierer

Die Schaltung heißt *Halbaddierer*. Der Name verrät, dass diese Schaltung nicht ausreicht, um ein Addierwerk aufzubauen. Das Problem ist, dass der Halbaddierer nur für die Einerstelle taugt. Bei allen weiteren Stellen müssen nämlich drei Ziffern addiert werden, die betreffende Stelle der beiden Summanden und der Übertrag, der bei der Addition der vorigen Stelle entstanden sein könnte. Eine Schaltung, die das kann, heißt *Volladdierer*. Ein Volladdierer besteht im Prinzip aus zwei Halbaddierern, was ja auch irgendwie lo-

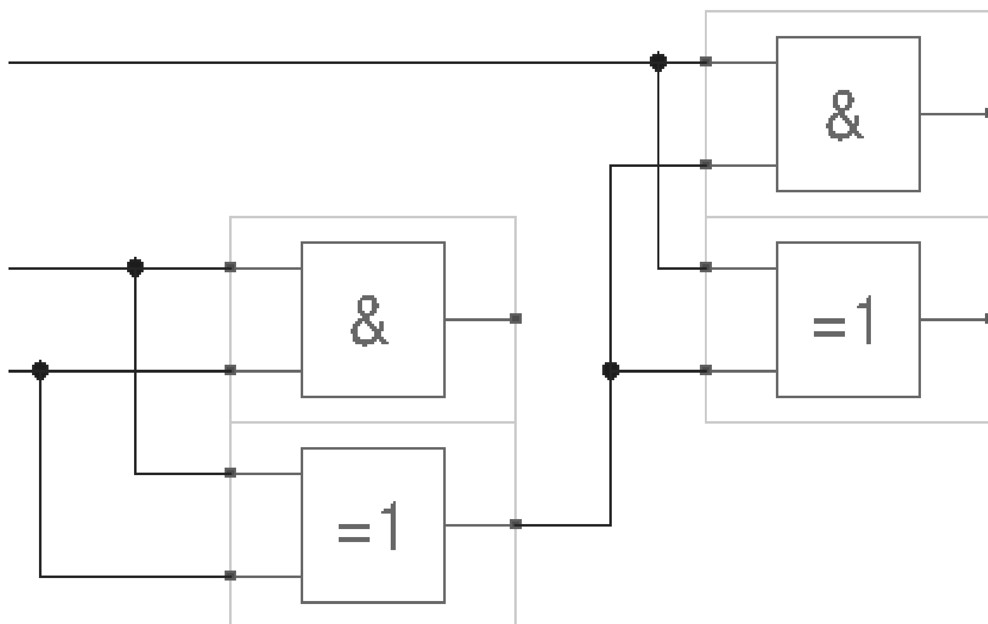


Abbildung 3.10: Volladierer, unfertig

gisch klingt. Im ersten Halbaddierer werden die beiden Stellen der Summanden addiert, zum Ergebnis wird dann im zweiten Halbaddierer der Übertrag addiert. Damit sieht ein Volladdierer zunächst – er ist so noch nicht fertig – aus wie in Abbildung 3.10.

Das Problem an dieser Schaltung ist, dass sie außer –wie beabsichtigt– den drei Eingängen für den 1. Summanden, den 2. Summanden und den Übertrag aus der vorigen Stelle auch drei Ausgänge besitzt. Gefordert sind zwei Ausgänge, ein Ausgang S für sie Stelle und ein Ausgang Ü für den Übertrag in die nächste Stelle. In der Tabelle 3.15 sind die Funktionen Ü und S des Volladdierers angegeben.

Tabelle 3.15: Wertetabelle eines Volladdierers

1. Summand A_n	2. Summand B_n	eingehender Übertrag $Ü_{n-1}$	ausgehender Übertrag $Ü_n$	Stelle S_n
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Aufgabe 3.26 *Stelle S und Übertrag Ü als Boolesche Funktionen*

1. *Wie lässt sich aus A_n, B_n und $Ü_{n-1}$ die Funktion S_n gewinnen?*
2. *Wie lässt sich aus A_n, B_n und $Ü_{n-1}$ die Funktion $Ü_n$ gewinnen?*

Erst weiterlesen, wenn Sie versucht haben, die Aufgabe zu lösen! Mit dieser Lösung sieht ein Volladierer aus wie in Abbildung 3.11 gezeigt.

Aufgabe 3.27 *Addierwerk bauen*

1. *Bauen Sie in Hades ein Addierwerk für zwei Summanden mit je einer (dualen) Stelle. Was brauchen sie? Einen Volladdierer oder reicht ein Halbaddierer? Benutzen Sie, um die Summanden zu erzeugen, zwei Exemplare von „Hex-Switch“ (siehe Abbildung 3.12). Sie brauchen nur den unteren Anschluss, der „0“ liefert, wenn „0“*

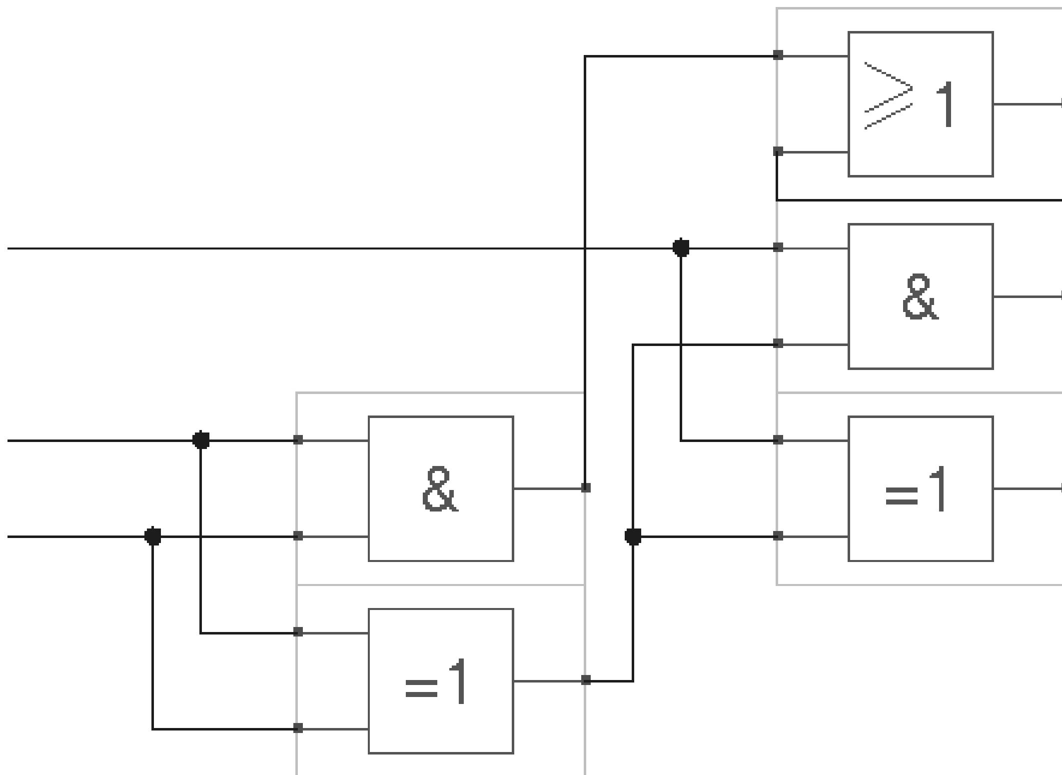


Abbildung 3.11: Volladierer

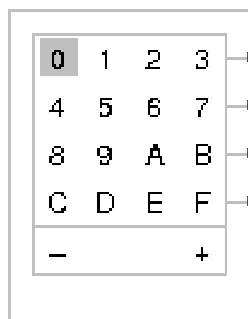


Abbildung 3.12: Hex-Switch

gewählt ist und „1“, wenn „1“ gewählt ist. Benutzen Sie, um das Ergebnis darzustellen, ein Exemplar von „Hex-Display“ (siehe Abbildung 3.13). Das Hex-Display funktioniert nur korrekt, wenn alle Anschlüsse belegt sind und ein definiertes Signal liefern. Fügen Sie deshalb einen Schalter ein, den Sie an die beiden oberen Anschlüsse des Hex-Displays anschließen. Schalten Sie den Schalter aus, so dass diese beiden Anschlüsse auf „0“ gelegt werden.

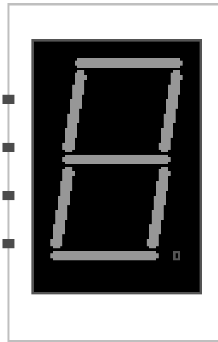


Abbildung 3.13: Hex-Display

2. *Erweitern Sie das Addierwerk für zwei Summanden mit je zwei dualen Stellen. Sie benötigen also nun die beiden nächsten Anschlüsse der beiden Hex switch, außerdem den dritten Anschluss am Hex display. Was brauchen Sie zusätzlich? Einen Volladdierer oder reicht ein Halbaddierer?*

3. *Erweitern Sie das Addierwerk für zwei Summanden mit je drei dualen Stellen.*

4. *Zusatzaufgabe: Erweitern Sie das Addierwerk für zwei Summanden mit jeweils vier dualen Stellen. Sie benötigen ein zweites Hex display für die zweite Stelle des Ergebnisses.*

Nachteil des Carry-Ripple-Addierers
Das in diesem Abschnitt besprochene Addierwerk ist ein Carry-Ripple-Addierer, auch Carry-Chain-Addierer genannt. „Carry“ ist englisch und bedeutet „Übertrag“, „chain“ bedeutet „Kette“. Damit wird beschrieben, dass in jedem Addiererbaustein ein Übertrag anfallen kann, der an den nächsten Addiererbaustein weitergegeben wird. Wenn es auf sehr schnelle Addierwerke ankommt, ist das von Nachteil, denn jedes Gatter braucht einige Sekundenbruchteile (im Bereich von ns) zum Schalten. Erst wenn klar ist, ob die Addition der ersten Stelle einen Übertrag liefert, liegen an den Eingängen des Addierbausteins für die zweite Stelle die richtigen Signale an. Erst nach dessen Schaltzeit kann die Berechnung der dritten Stelle erfolgreich sein usw. Das heißt, die Schaltverzögerungen der einzelnen Addierbausteine addieren sich.

Es gibt daher verschiedene Addierwerke, die diesen Nachteil vermeiden, z.B. den Conditional-Sum-Addierer. Diese Addierwerke sind aber wesentlich komplizierter, das heißt aus mehr Gattern aufgebaut, als ein Carry-Ripple-Addierer für dieselbe Anzahl von Stellen.

Kapitel 4

Programmieren in Java – Teil 2

1 Ein genauerer Blick auf Variablen

1.1 Initialisierung von Variablen und Sichtbarkeit von Variablen

Wir haben es bisher mit drei Arten von Variablen in Java-Programmen zu tun gehabt: Attribute, Parameter von Methoden oder Konstruktoren und lokale Variablen. In der folgenden Klasse kommen alle drei Arten vor. „start“ ist ein Attribut, „ende“ ist ein Parameter und „i“ ist eine lokale Variable.

Listing 4.1: Attribute, Parameter, lokale Variablen

```
public class BeispielVariablentypen {
    int start;
    public void zaehlen(int ende) {
        for (int i = start; i <= ende; i++) {
            System.out.println(i);
        }
    }
}
```

Ruft man die Methode „zaehlen“ auf, so werden die ganzen Zahlen von 0 bis „ende“ ausgegeben. Der Wert des Attributs „start“ beträgt also 0, obwohl dieser Variablen niemals ein Wert zugewiesen wurde. Alle Attribute bekommen bei ihrer Erzeugung automatisch einen Wert; man sagt, sie werden automatisch initialisiert. Attribute vom Typ byte, short, int oder long werden automatisch mit dem Wert 0 initialisiert, solche vom Typ float oder double mit dem Wert 0.0, Attribute vom Typ boolean werden automatisch mit dem Wert „false“ initialisiert. Attribute, die einen Objekttyp repräsentieren, also auch Attribute vom Typ String, werden mit dem speziellen Wert „null“ (nicht zu verwechseln mit 0) initialisiert.

Parameter werden nicht initialisiert, denn sie erhalten ihren jeweiligen Wert beim Aufruf der Methode. Lokale Variablen müssen explizit initialisiert werden. Im obigen Beispiel wird die lokale Variable „i“ mit dem Wert des Attributs „start“ initialisiert, wobei „start“ den Wert 0 hat. Ersetzt man den Kopf der for-Schleife durch eine Version ohne Initialisierung von „i“, also `for(int i; i <= ende; i++) {...}` so erhält man einen Compiler-Fehler („variable i might not have been initialized“). Bei einem Array wäre es oft sehr lästig, alle Elemente des Arrays in einer for-Schleife zu initialisieren. Daher werden lokale Array-Variablen wie Attribute automatisch initialisiert.

Auf ein Attribut kann in der ganzen Klasse zugegriffen werden. Man sagt, ihre Sichtbarkeit erstreckt sich über die gesamte Klasse. Ein Parameter ist nur im Rumpf der betreffenden Methode oder des betreffenden Konstruktors sichtbar. Eine lokale Variable ist nur sichtbar in dem Block, in dem sie deklariert wurde oder, wenn sie im Kopf einer for-Schleife deklariert wurde, im Rumpf dieser for-Schleife. Die folgende Klasse liefert einen Fehler („i cannot be resolved to a variable“), wenn man versucht, sie zu kompilieren, weil auf die Variable i außerhalb der for-Schleife nicht zugegriffen werden kann.

Listing 4.2: Falscher Variablenzugriff

```
1 public class VariablenzugriffAusserhalbDesBlocks {
2     int start;
3     public void zaehlen(int ende) {
4         for (int i = start; i <= ende; i++) {
5             System.out.println(i);
6         }
7         System.out.println("Und am Ende hat i den Wert: " +i);
8     }
9 }
```

1.2 Variablen mit gleichem Namen und this-Operator

Komplizierter werden die Dinge, wenn verschiedene Variablen denselben Namen haben. Deklariert man zwei Attribute mit demselben Namen oder deklariert man in einem Block zwei lokale Variablen mit demselben Namen, so erhält man einen Compilerfehler („duplicate ...“). Man erhält aber keinen Compilerfehler, wenn man einen Parameter oder eine lokale Variable gleich benennt wie ein Attribut.

Aufgabe 4.1 *Variablenzugriff*
Geben Sie die folgende Klasse (Listing 4.3) ein, erzeugen Sie eine Instanz dieser Klasse und probieren Sie die drei öffentlichen Methoden „ausgeben1()“, „ausgeben2()“ und „ausgeben3()“ aus. Welche Ausgabe erhalten Sie und warum?

Listing 4.3: Verschiedene Variablen mit demselben Namen

```
1  public class VerschiedeneVariablenMitGleichemNamen {
2      String meinString = "Ich bin das Attribut!";
3      public void ausgeben1() {
4          System.out.println(meinString);
5      }
6
7      public void ausgeben2() {
8          String meinString = "Ich bin die lokale Variable!";
9          System.out.println(meinString);
10     }
11
12     public void ausgeben3() {
13         parameterSchreiben("Ich bin der Parameter!");
14     }
15
16     private void parameterSchreiben(String meinString) {
17         System.out.println(meinString);
18     }
19 }
```

Bei der vorigen Aufgabe sollten Sie bemerkt haben: Wenn eine lokale Variable denselben Namen besitzt wie ein Attribut, dann spricht dieser Namen im Sichtbarkeitsbereich beider Variablen die lokale Variable an. Es ist allerdings schlechter Programmierstil, eine lokale Variable genauso zu nennen wie ein Attribut. Außerdem sollten Sie erkannt haben: Wenn ein Parameter denselben Namen besitzt wie ein Attribut, dann spricht dieser Name im Sichtbarkeitsbereich beider Variablen den Parameter an. Die Verwendung desselben Namens für einen Parameter und ein Attribut ist häufige Programmierpraxis und kann, geeignet eingesetzt, sehr guter Programmierstil sein, weil es die Lesbarkeit einer Methode vereinfachen kann.

Das Verwenden des selben Namens für ein Attribut und einen Parameter kann aber nur dann sinnvoll sein, wenn man im gemeinsamen Sichtbarkeitsbereich nicht nur den Parameter, sondern auch das Attribut ansprechen kann. Dies gelingt mit dem this-Operator. „this“ von englisch „dies“ zeigt an, dass sich etwas auf das gerade betrachtete Objekt bezieht. Der this-Operator kann dem Namen einer Variablen vorangestellt werden mit der Punktnotation. Damit wird das Attribut für dieses Objekt aufgerufen. Die Zeile `System.out.println(this.meinString);` in der Klasse aus Listing 4.3 führt also immer zu der Ausgabe „Ich bin das Attribut!“, und zwar in der Methode „ausgeben1“ genauso wie in der Methode „ausgeben2“ und der Methode „ausgeben3“.

Aufgabe 4.2 *this-Operator*
Probieren Sie die Zeile

System.out.println(this.meinString);
in den öffentlichen Methoden der vorigen Aufgabe aus.

Die Benennung eines Parameters mit demselben Namen wie ein Attribut ist üblich bei Settern, bei denen ja der Parameter der neue Wert für das Attribut ist. In der Klasse „Rechteck“ aus Aufgabe 2.10 auf Seite 20 sahen die Setter z.B. so aus:

```
void setBreite(int neueBreite){
        breite = neueBreite;
        zeichnen();
}
```

Eigentlich sollte der Parameter kurz und prägnant „breite“ heißen, gerade weil das Attribut, das gesetzt werden soll, genauso heißt. Dies kann mit dem this-Operator nun realisiert werden. Die Methode lautet dann:

```
void setBreite(int breite){
        this.breite = breite;
        zeichnen();
}
```

Machen Sie sich nochmal klar: „this.breite“ bezeichnet das Attribut, „breite“ den Parameter. Die Anweisung `this.breite = breite;` weist dem Attribut den Wert des Parameters zu.

2 Algorithmik am Beispiel Sortierung

2.1 Einführung

Unter einem Algorithmus versteht man eine Lösungsanweisung für ein Problem, die aus endlich vielen Schritten besteht. Einem Computerprogramm, speziell einer Methode, liegt stets ein Algorithmus zu Grunde. Dasselbe Problem lässt sich durch unterschiedliche Algorithmen lösen, wie am Beispiel des Sortierens gezeigt werden soll. Algorithmen können unterschiedlich gut sein. So kann ein Algorithmus ein Problem in weniger Schritten lösen als ein anderer Algorithmus. Dies wird, wenn der Algorithmus als Computerprogramm implementiert wird, im Normalfall eine kürzere Rechenzeit zur Folge haben. Andererseits kann ein Algorithmus schwerer zu implementieren sein, so dass die Implementierung, also das Programmieren, länger dauert oder gar, wenn der Programmierer den Anforderungen nicht gewachsen war, ein fehlerhaftes Programm entsteht.

Aufgabe 4.3 *Sortieren*

1. *Beschriften Sie etwa zehn Zettel mit Zahlen. Mischen Sie und sortieren Sie dann die Zahlen in aufsteigender Reihenfolge. Geben Sie eine Anleitung zum Sortieren,*

so dass jemand anderes nach Ihrem Algorithmus sortieren kann. Gibt es andere Algorithmen zum Sortieren?

2. *Schreiben Sie eine Klasse „MeineZahlen“. Einziges Attribut soll ein Array von Integern, also ein Objekt vom Typ int[] sein. „MeineZahlen“ soll eine Methode „eingeben“ besitzen. Beim Aufruf dieser Methode soll der Benutzer gefragt werden, wie viele Zahlen er eingeben möchte. Die Eingabe soll größer als drei sein. Anschließend wird der Benutzer aufgefordert, die Zahlen einzugeben, die dann gespeichert werden. Weiter soll die Klasse über eine Methode „ausgeben“ verfügen, die die Zahlen des Arrays ausgibt. Testen Sie die Methoden mit einer geeigneten main()-Methode. Die Klasse wird für die weiteren Aufgaben gebraucht.*

3. *Um die Zahlen im Array zu sortieren, muss man oft zwei Zahlen vertauschen. Sie haben zwei Variablen vom Typ int, a und b. Wie lauten die Programmzeilen, die dafür sorgen, dass zum Schluss der alte Wert von b nun in der Variablen a steckt, während die Variable b den alten Wert von a hält?*

4. *Beim vorigen Aufgabenteil haben Sie wahrscheinlich eine dritte Variable benutzt, die vorübergehend einen der Werte aufnimmt und so davor bewahrt, „überschrieben“ zu werden. Geht es auch ohne eine dritte Variable?*

5. *Schreiben Sie eine Methode „zufaelligEingeben“. Hier wird der Benutzer nur gefragt, wie viele Zahlen das Array umfassen soll. Das Array wird dann mit Zufallszahlen gefüllt. Die Erzeugung von Zufallszahlen wurde ab Seite 34 erläutert.*

2.2 Selectionsort

Beim Sortieren der Zettel sind Sie eventuell folgendermaßen vorgegangen: Sie haben den Zettel mit der niedrigsten Zahl gesucht und nach vorne genommen. Dann haben Sie den mit der nächst höheren Zahl gesucht und an die zweite Stelle gesteckt und so weiter. Dieser Algorithmus wird „Selectionsort“ genannt, was soviel bedeutet wie „Sortieren durch Auswählen“, da der jeweils nächste Wert gesucht und ausgewählt wird. Wenn man mit Selectionsort ein Array sortiert, muss man geringfügig anders vorgehen als bei den Zetteln: Hat man den niedrigsten Wert gefunden, kann man den nicht einfach nach vorne stecken, da der Platz anderweitig besetzt ist. Statt dessen muss man die beiden Werte vertauschen.

Aufgabe 4.4 *Implementieren von Sortieralgorithmen*

1. *Erweitern Sie Ihre Klasse „MeineZahlen“ um eine Methode „sortierenDurchAuswaehlen“. Diese sortiert das Array nach dem Algorithmus Selectionsort.*

2. *Schreiben Sie eine Methode „tauschen“, die zwei int-Werte vertauscht. Sie soll die erweiterte Signatur* `private void tausch(int positionA, int positionB)` *besitzen. Benutzen Sie diese Methode in der in Aufgabenteil 1 programmierten Methode.*

Wie gut ist dieser Algorithmus? Er ist nicht sehr kompliziert zu implementieren. Wie schnell ist er, was eine große Rolle spielt, wenn vielleicht hunderttausende Werte sortiert werden müssen? Sind n Werte zu sortieren, so muss man, um den kleinsten Wert zu finden, alle n Werte durchgehen und vergleichen. Danach muss man nur noch ab dem zweiten Eintrag nach dem nächstgrößeren Wert suchen, muss also noch n–1 Werte untersuchen. Insgesamt muss man also $n+(n-1)+(n-2)+(n-3)+...+3+2+1 = \sum_{i=1}^{n} i = \frac{n(n-1)}{2} = \frac{1}{2}(n^2-n)$ Werte untersuchen[1]. Im schlechtesten Fall steht kein Wert, wenn er ausgewählt wird, an der richtigen Stelle. Nach n – 1 Vertauschungen haben n – 1Werte ihren Platz gefunden und dann auch zwangsläufig der letzte Wert, da es keinen falschen Platz mehr für ihn gibt. Geht man davon aus, dass die Untersuchung eines Werts und die Vertauschung dieselben Kosten verursacht, belaufen sich die Gesamtkosten auf $\frac{1}{2}(n^2-n)+n-1 = \frac{1}{2}(n^2+n)-1$ Bei sehr großen n, und nur hier sind diese Laufzeituntersuchungen interessant, ist n gegenüber n^2 vernachlässigbar und auch die „1“ spielt keine Rolle, es gilt also $\frac{1}{2}(n^2+n)-1 \approx \frac{1}{2}n^2$. Weiterhin interessiert man sich nicht für konstante Faktoren wie z.B. $\frac{1}{2}$, denn es ist viel wichtiger, ob der Zeitbedarf eines Algorithmus proportional n, proportional n^2 oder proportional n^3 ist. Insgesamt sagt man daher: Der Zeitbedarf von Selectionsort zur Sortierung von n Einträgen ist von der Ordnung n^2. Im durchschnittlichen Fall ist der Zeitbedarf ebenfalls von der Ordnung n^2, denn es müssen immer $\frac{n(n-1)}{2}$ Werte auf der Suche nach dem jeweils nächsten Wert durchgegangen werden, es sind lediglich weniger Vertauschungen nötig, weil sich der eine oder andere Wert vielleicht zufällig schon an der richtigen Position befindet.

2.3 Bubblesort

Nehmen Sie Ihre Zettel zur Hand und legen Sie sie, nicht sortiert, vor sich hin. Vergleichen Sie den ersten mit dem zweiten. Steht auf dem ersten eine kleinere Zahl als auf dem zweiten, lassen Sie die beiden liegen. Ist aber auf dem zweiten eine kleinere Zahl, so vertauschen Sie die beiden. Verfahren Sie nun genauso mit dem zweiten und dem dritten Zettel, dann mit dem dritten und dem vierten usw., bis schließlich mit dem vorletzten und letzten so verfahren wurde. Beginnen sie anschließend wieder von vorne. Führen Sie das Verfahren so lange durch, bis die Zettel sortiert sind. Dies ist das Sortieren nach dem Algorithmus „Bubblesort“. Der Name von englisch bubbles, Blasen, bezieht sich darauf, dass die großen Zahlen nach oben steigen wie die Blasen in einer Limonade.

[1] Die verwendete Gleichung ist die Gauss'sche Summenformel, die Sie aus dem Mathematikunterricht kennen sollten. Sie wird in der Informatik oft benötigt. Ihr Beweis wird gerne als Beispiel für die Beweismethode der vollständigen Induktion verwendet.

Aufgabe 4.5 *Bubblesort*

1. *Erweitern Sie Ihre Klasse „MeineZahlen“ um eine Methode „sortierenBubbles“. Diese sortiert das Array nach dem Algorithmus Bubblesort.*

2. *Untersuchen Sie das Laufzeitverhalten von Bubblesort: Welcher Ordnung ist Bubblesort? Unterscheiden sich die Ordnung des schlechtesten und des durchschnittlichen Falls?*

3. *Nach einem Durchlauf ist die größte Zahl am richtigen Ort, nach dem zweiten Durchlauf sind die beiden größten Zahlen am richtigen Ort usw. Daher kann man jeden weiteren Durchlauf etwas früher beenden und spart dadurch Zeit. Verbessern Sie Ihren Bubblesort-Algorithmus nach dieser Idee! Ändert sich dadurch die Ordnung des Laufzeitverhaltens?*

2.4 Insertionsort

Legen Sie Ihre Zettel in einer Reihe unsortiert vor sich. Lassen Sie darunter Platz für eine neue Reihe. Nehmen Sie nun den ersten Zettel, egal welche Zahl darauf steht, und legen Sie ihn in die untere Reihe, ganz links. Nehmen Sie nun den zweiten Zettel von oben. Steht dort eine größere Zahl als auf dem ersten, legen Sie ihn rechts vom ersten ab. Ist die Zahl kleiner, schieben Sie den ersten Zettel nach rechts und legen den zweiten links daneben. Nehmen Sie nun den dritten Zettel der oberen Reihe und fügen Sie ihn in der unteren Reihe an der richtigen Position ein. Fahren Sie fort, bis die obere Reihe leer ist und alle Zettel in der unteren Reihe liegen. Die untere Reihe ist nun sortiert. Der verwendete Algorithmus heißt Insertionsort, was soviel bedeutet wie „sortieren durch Einfügen“.

Wenn man Insertionsort implementiert, braucht man ein zweites Array, entsprechend der zweiten Reihe bei den Zetteln. Die Einträge im ersten Array bleiben am einfachsten bestehen.

Aufgabe 4.6 *Insertionsort implementieren*
Erweitern Sie Ihre Klasse „MeineZahlen“ um eine Methode „sortierenDurchEinfuegen“. Diese sortiert das Array nach dem Algorithmus Insertionsort.

Insertionsort hat den Nachteil, dass ein zweites Array gebraucht wird. Der Bedarf an Speicherplatz ist daher doppelt so groß wie bei Selectionsort oder Bubblesort, was bei großen Datenätzen problematisch sein kann. Die Implementierung ist etwas schwieriger als bei SelectionSort oder Bubblesort, weil das Verschieben mehrerer Einträge beim Einfügen einer Zahl nicht ganz leicht ist.

Aufgabe 4.7 *Laufzeit von Insertionsort*
Ist wenigstens die Laufzeit bei Insertionsort kürzer als bei Selectionsort und Bubblesort?

Zusammengesetze Objekte können nach verschiedenen Attributen sortiert werden. Eine Kartei mit Studenten speichert z.B. Namen, Vornamen, Adresse, Geburtsdatum etc. Um das Geburtsdatum zu erfassen, kann das Geburtsjahr, der Monat und der Tag in ein Formular eingegeben werden. Eventuell wird also gar kein Attribut „Geburtsdatum" gespeichert, sondern drei Attribute „Geburtsjahr", „Monat", „Tag". Will man nun eine Liste mit Studenten nach dem Alter ordnen, so muss man sie zuerst nach dem Tag, dann nach dem Monat und schließlich nach dem Jahr sortieren. Wenn zum Schluss nach dem Jahr sortiert wird, dann darf bei Studenten, die im selben Jahr geboren sind, die vorher erfolgte Sortierung nach dem Monat und dem Tag nicht wieder durcheinander gebracht werden. Einen Sortieralgorithmus, der dies leistet, nennt man „stabil", andernfalls „instabil".

Aufgabe 4.8 *Stabilität*
Untersuchen Sie Ihre Implementierungen von Selectionsort, Bubblesort und Insertionsort auf Stabilität!

3 Rekursion

3.1 Ein schlechtes Beispiel zur Einführung: Fakultät

Hier ist ein Programm, mit dem man die Fakultät einer natürlichen Zahl berechnen kann:

Listing 4.4: Iterative Berechnung der Fakultät

```
1  public class Fakultaet {
2      private int zahl;
3      private Textfenster fenster = new Textfenster();
4
5      public void fakultaetErrechnen() {
6          zahlEingeben();
7          int ergebnis = ausrechnen(zahl);
8          ausgeben(ergebnis);
9      }
10
11     private void zahlEingeben() {
12         zahl = fenster.intEingeben("natuerliche Zahl eingeben");
13         while (zahl < 0) {
14             zahl = fenster.intEingeben("natuerliche Zahl
                   eingeben");
15         }
16     }
17
18     private int ausrechnen(int n) {
```

```
19          int ergebnis = 1;
20          for (int i = 1; i <= n; i++) {
21              ergebnis = ergebnis * i;
22          }
23          return ergebnis;
24      }
25
26      private void ausgeben(int ergebnis) {
27          fenster.schreiben(zahl +"! = " +ergebnis);
28          fenster.zeileUmbrechen();
29      }
30  }
```

Aufgabe 4.9 *Geeigneter Typ für Fakultät*

1. *Probieren Sie das Programm aus. Was erhält man für die Eingabe „0“? Ist das korrekt?*

2. *Was erhält man für die Eingabe „17“? Woran liegt das? Wie könnte man das verbessern? Wie kann man generell verhindern, dass ein falsches Ergebnis ausgegeben wird?*

Kernstück des Programms ist die Methode ausrechnen. Sie benutzt eine Schleife. Eine Methode, die eine Schleife benutzt, nennt man „iterativ“. Man kann die Methode „ausrechnen“ durch die folgende Version, die keine Schleife enthält, ersetzen:

Listing 4.5: Rekursive Bererchnung der Fakultät

```
1  private int ausrechnen(int n) {
2      int ergebnis;
3      if (n == 0) {
4          ergebnis = 1;
5      }
6      else {
7          ergebnis = n * ausrechnen(n - 1);
8      }
9      return ergebnis;
10 }
```

Aufgabe 4.10 *Rekursive Variante von „Fakultät“*

1. *Überzeugen Sie sich, dass auch diese Methode funktioniert.*

2. *Wie oft wird die Methode „ausrechnen“ ausgeführt, wenn man „3“ eingibt? Wie oft wird dabei der if-Zweig, wie oft der else-Zweig durchlaufen?*

Diese Version der Methode „ausrechnen“ ruft sich im Rumpf selbst auf. Dies nennt man „Rekursion“. Die Rekursion setzt sich aber nicht unendlich fort, weil im Rumpf irgendwann die Methode „ausrechnen“ mit dem Parameter „0“ aufgerufen wird. In diesem Aufruf geht die Methode in den if-Zweig, der keinen rekursiven Aufruf enthält. Damit terminiert die Rekursion. Bei der Eingabe „3“ wird zunächst der else-Zweig durchlaufen. 3! ist 3·2!. Im else-Zweig wird die Frage gestellt „was ist die Fakultät von 2?“. Zur Beantwortung erfolgt der dritte Aufruf: Die Fakultät von 2 ist 2·1!. Zur Berechnung von 1! erfolgt der vierte und letzte Aufruf der Methode „ausrechnen“: 1! ist 1·0! und 0! ist 1, wie im if-Zweig festgelegt ist. Zur Berechnung von n! wird die Methode ausrechnen also insgesamt n+1 mal aufgerufen: Einmal „von außerhalb“, dann folgen n rekursive Aufrufe. Von den n+1 Aufrufen laufen n durch den else-Zweig, einer, nämlich der letzte, durch den if-Zweig.

Beim Programmieren einer Rekursion besteht immer die Gefahr, eine endlose Rekursion zu programmieren. Eine rekursive Methode muss ein „if“ enthalten, sonst kann sie nicht terminieren.

Vergleichen wir die iterative mit der rekursiven Version von „ausrechnen“. Die Nachteile der rekursiven Methode sind offensichtlich: Ihr Quelltext ist etwas länger und viel schwerer zu verstehen. Welche Vorteile hat die rekursive Version? Keine! Man braucht ein Loch im Hirn, um die Berechnung der Fakultät rekursiv statt iterativ anzugehen. An diesem Beispiel kann man die Rekursion aber gut verstehen.

Bei anderen Problemen kann ein rekursiver Algorithmus einfacher zu verstehen sein als ein iterativer, z.B. bei den Türmen von Hanoi im nächsten Abschnitt. Weitere Beispiele sind die Sortieralgorithmen Quicksort (Abschnitt 3.3) und Mergesort (Abschnitt 3.4), die schneller arbeiten als die im Abschnitt 2 vorgestellten Sortieralgorithmen und am einfachsten rekursiv formuliert werden.

3.2 Ein besseres Beispiel: Türme von Hanoi

Am Standaort A steht ein Turm. Jeder Stein hat eine andere Größe. Unter jedem Stein liegt ein größerer Stein. Der Turm soll am Standort A abgebaut und am Standort C wieder aufgebaut werden. Es kann immer nur ein Stein gleichzeitig verschoben werden. Es dürfen vorübergehend Steine am Standort B gelagert werden. Es darf niemals – auch nicht am Standort B – ein Stein auf einem kleineren Stein liegen.

Für einen Turm der Höhe 1 ist die Lösung trivial: Man legt den einen Stein von A nach C. Auch für zwei Steine ist die Lösung nicht gerade schwer: Man legt Stein 1 nach B, Stein 2 nach C und dann Stein 1 nach C.

Aufgabe 4.11 *Türme von Hanoi*

Abbildung 4.1: Türme von Hanoi

1. *Lösen Sie die „Türme von Hanoi“ mit drei, vier und fünf Steine hohen Türmen! Sie können als Steine z.B. Münzen mit unterschiedlichem Wert verwenden.*

2. *Formulieren Sie einen Algorithmus zur Lösung der Türme von Hanoi!*

3. *Wie viele Schritte benötigt man, um einen Turm der Höhe 1, einen Turm der Höhe 2, einen Turm der Höhe 3 zu verschieben? Allgemein: Wie viele Schritte benötigt man, um einen Turm der Höhe n zu verschieben? Können Sie einen Turm der Höhe 20 an einem Tag verschieben? Reicht Ihr Leben aus, um einen Turm der Höhe 32 zu verschieben?*

4. *Beweisen Sie die Richtigkeit Ihrer allgemeinen Gleichung aus Aufgabe 3!*

Sicher haben Sie Aufgabe 1 lösen 2 können. Da Sie dabei eine Lösungsstrategie (einen Algorithmus!) entwickelt haben, haben Sie sicher auch keine Angst, die Aufgabe für noch größere Türme zu lösen. Dennoch könnte es sein, dass es Ihnen schwer gefallen ist, Aufgabe 2 zu lösen, d.h. Ihre Lösungsstrategie aufzuschreiben. Das liegt daran, dass der einfachste Algorithmus rekursiv ist, man aber beim Nachdenken über Rekursionen leicht einen Knoten ins Hirn bekommt[2]. Hier ist ein Vorschlag:

Um einen Turm der Höhe n von A nach C umzustapeln, muss man folgendes tun:

1. Wenn n > 1, stapele den Turm der Höhe n–1 (also alles ohne Stein n) von A nach B.

2. Lege den Stein n von A nach C.

3. Wenn n > 1, stapele den Turm der Höhe n–1 von B nach C.

[2] Mein Bruder pflegt zu sagen: „Wenn man Rekursion kapieren will, muss man erst mal Rekursion kapieren, danach ist es leicht.“

Wie stapelt man aber den Turm der Höhe n–1 um, was in diesem Algorithmus zweimal gefordert ist? Ganz einfach: Die Anleitung ist dieselbe, nur muss man n durch n–1 und n–1 durch n–2 ersetzen, außerdem müssen die Namen der Orte A, B, C angepasst werden. Der letzte rekursive Aufruf erfolgt für einen Turm der Höhe $n = 1$. Hier entfallen die Schritte 1 und 3 wegen der Bedingung $n > 1$ und nur Schritt 2 muss ausgeführt werden.

Aufgabe 4.12 *Türme von Hanoi implementieren*
Programmieren Sie die Türme von Hanoi. Das Programm soll eine Ausgabe der Art „Lege Stein 1 von A nach C, lege Stein 2 von A nach B…“ erzeugen. Das Programm besteht im Wesentlichen aus einer Methode, deren Rumpf aus den obigen drei Schritten besteht, d.h. je einem rekursiven Aufruf in Schritt 1 und 3 und einer Ausgabe in Schritt 2. Der Methode muss mitgeteilt werden, wie groß der Turm ist, wie der Startort, der Zwischenort und der Zielort heißen. Das heißt, die Methode hat vier Parameter.

3.3 Quicksort

Quicksort ist ein weiterer Sortieralgorithmus. Er ist ein sogenannter Teile-und-hersche-Algorithmus. Dabei wird das Gesamtproblem in mehrere kleinere Probleme unterteilt und diese weiter unterteilt, bis die kleinen Teilprobleme leicht zu lösen sind. Aus den Lösungen der Teilprobleme wird dann die Lösung des Gesamtproblems gewonnen. Teile-und-herrsche-Algorithmen sind meist rekursiv, so auch in diesem Beispiel.

Nehmen Sie wieder Ihre Zettel mit den Zahlen zur Hand. Sie sollen gemischt sein. Wählen Sie einen, auf dem nicht das Minimum steht. Bilden Sie nun zwei Teildatensätze: In den linken Teildatensatz kommen alle Zettel mit Zahlen, die kleiner sind als die auf dem ausgewählten Zettel, in den rechten die Zettel mit Zahlen größer oder gleich der Zahl auf dem ausgewählten Zettel, also auch der zuerst ausgewählte Zettel selbst. Verfahren Sie anschließend mit den beiden Teildatensätzen genauso, mit den dabei entstandenen Teildatensätzen genauso. Ein Teildatensatz, der nur noch aus Zetteln mit derselben Zahl besteht, wird in Ruhe gelassen. Ein Teildatensatz besteht spätestens dann nur noch aus Zetteln mit derselben Zahl, wenn er nur noch einen einzigen Zettel umfasst. Zum Schluss haben Sie nur noch Teildatensätze mit Zetteln mit derselben Zahl. Diese sind trivialerweise geordnet. Vereint man die Teildatensätze von links nach rechts, so erhält man die Zahlen geordnet.

Beispiel: Die folgenden Zahlen sollen geordnet werden: 3, 2, 5, 7, 3, 9, 1, 6, 4, 2. Wähle eine der Zahlen, die nicht minimal ist, z.B. die 3. Dann entstehen zwei Teildatensätze:
Erster Teildatensatz: 2, 1, 2, zweiter Teildatensatz: 3, 5, 7, 3, 9, 6, 4. Wähle nun im ersten Teildatensatz 2, im zweiten Teildatensatz 5 (Beachten Sie: Die 3 darf hier nicht gewählt werden, da die 3 im zweiten Teildatensatz nun minimal ist!). Damit erhält man vier Teildatensätze: erster Teildatensatz: 1, zweiter Teildatensatz 2, 2, dritter Teildatensatz 3, 2, 3, 1, 2, vierter Teildatensatz 5, 7, 9, 6. Und so weiter, siehe die Abbildung 4.2.

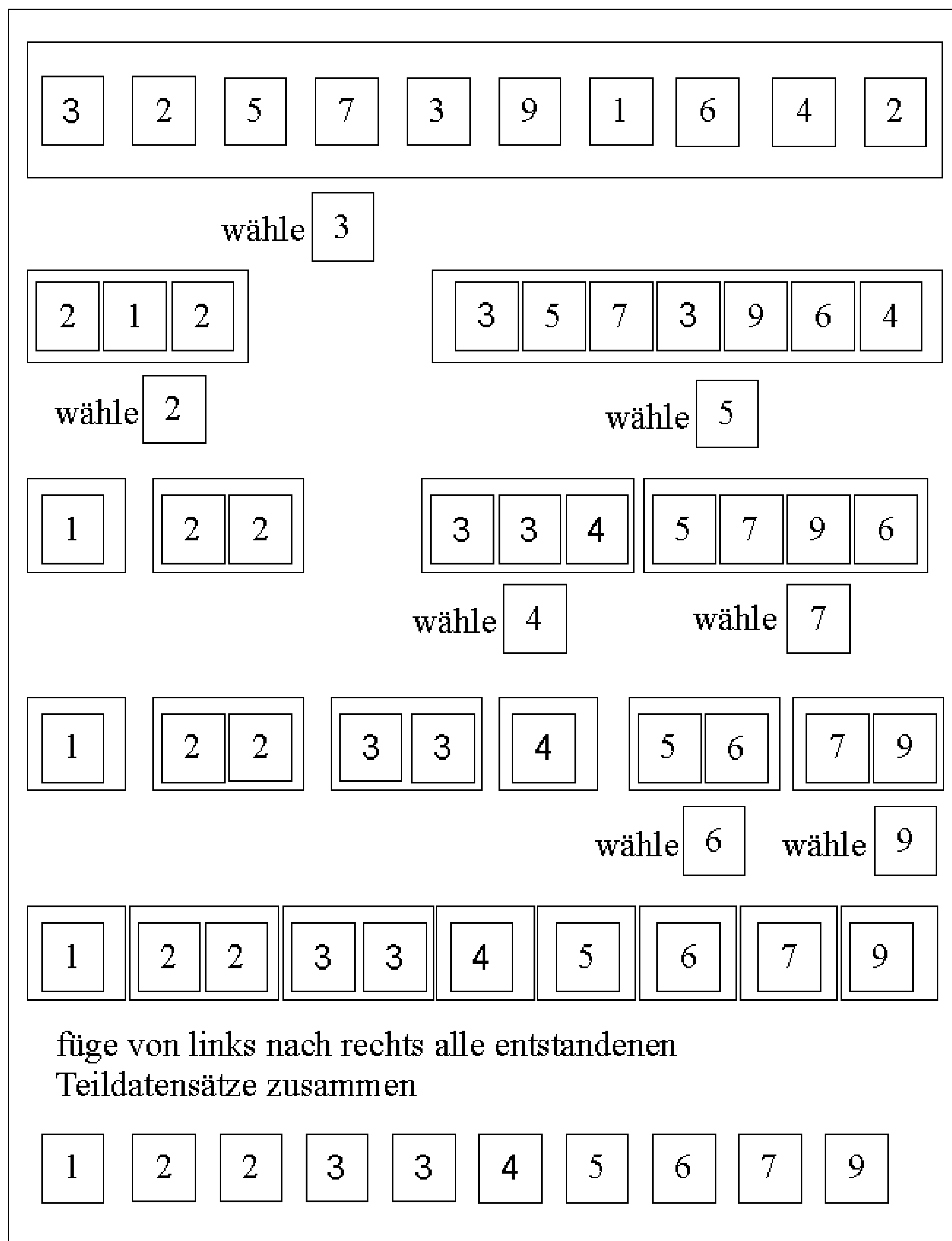


Abbildung 4.2: Quicksort

Aufgabe 4.13 *Implementieren von Quicksort (Zusatzaufgabe, schwer)*
Implementieren Sie Quicksort! Vielleicht müssen Sie sich Anregungen im www holen.

Das durchschnittliche Laufzeitverhalten von Quicksort ist von der Ordnung n log n und ist damit besser als die drei vorher besprochenen Verfahren, die alle der Ordnung n^2 sind. Es lässt sich beweisen, dass kein Sortierverfahren ein besseres Laufzeitverhalten als n log n aufweisen kann. Bei Quicksort handelt es sich also um einen der besten Sortieralgorithmen. Die Schreibweise n log n wirft scheinbar die Frage auf, zu welcher Basis der Logarithmus ist. Bei der Berechnung hat man es mit dem Logarithmus zur Basis 2 zu tun, da der Datensatz im Allgemeinen in zwei kleinere Datensätze unterteilt wird. Da aber bei der Angabe einer Ordnung auf die Konstanten verzichtet wird und sich Logarithmen zu verschiedenen Basen nur um einen konstanten Faktor unterscheiden, wird die Basis nicht angegeben.

Leider ist Quicksort im schlechtesten Fall nur von der Ordnung n^2. Der schlechteste Fall tritt ausgerechnet dann auf, wenn die Daten bereits sortiert sind, denn dann wird typischerweise das zweitkleinste Element zum Trennen in die beiden Teildatensätze verwendet, so dass der linke Teildatensatz nur ein Element besitzt und der rechte alle restlichen. Auf diese Weise muss n mal ein Element abgetrennt werden, wobei dafür jedes Mal alle n Elemente durchgegangen werden müssen. In der Praxis dürfte der Fall, dass ein weitgehend sortierter Datensatz zu sortieren ist, oft vorkommen, was für die Anwendung von Quicksort problematisch ist. Trickreiche Implementierungen, die z.B. einen verbesserten Algorithmus benutzen, um ein Element auszuwählen, nach dem der Datensatz getrennt wird, vermeiden dieses Problem.

Ein weiterer Nachteil von Quicksort ist, dass dieser Algorithmus, genauso wie Selectionsort, nicht stabil ist. Wie bei Selectionsort kann man aber auch bei Quicksort mit etwas Aufwand stabile Varianten programmieren.

3.4 Mergesort

Wie Quicksort ist Mergesort ein Teile-und-herrsche-Algorithmus und damit rekursiv. Die Laufzeit von Mergesort ist auch im schlechtesten Fall von der Ordnung n log n, also der bestmögliche Fall. In der Praxis schneidet es aber im Allgemeinen, d.h. wenn die Datensätze nicht vorsortiert sind, etwas schlechter ab als Quicksort. Mergesort ist stabil. Ein Nachteil ist, dass es im Allgemeinen, ähnlich wie Insertionsort, zusätzlichen Speicherplatz benötigt, um die Daten hin- und her zu schieben, was sich nur unter Aufwand vermeiden lässt.

Bei Mergesort wird der zu sortierende Datensatz zunächst in Teildatensätze zerlegt, die alle genau ein Element besitzen. Einelementige Datensätze sind trivialerweise sortiert. Anschließend werden je zwei der Teildatensätze miteinander verschmolzen. Bei einer ungeraden Anzahl von Teildatensätzen bleibt ein Teildatensatz bestehen. Beim verschmelzen der Teildatensätze wird darauf geachtet, dass der entstehende Datensatz geordnet ist. Das Verschmelzen je zweier Teildatensätze wird solange wiederholt, bis nur noch ein, dann geordneter, Datensatz vorliegt. Ein Beispiel ist in Abbildung 4.3 gezeigt.

Aufgabe 4.14 *Implementieren von Mergesort (Zusatzaufgabe, schwer)*
Implementieren Sie Mergesort! Vielleicht müssen Sie sich Anregungen im www holen.

Mergesort und Quicksort haben gemeinsam, dass sie Teile-und-herrsche-Algorithmen sind, unterschieden sich aber fundamental darin, in welchem Schritt die eigentliche Sortierarbeit verrichtet wird: Quicksort macht die eigentliche Arbeit bei der Teilung des Datensatzes. Die Verschmelzung der Teildatensätze zum Schluss ist dagegen trivial. Die Teilung des Datensatzes bei Mergesort ist trivial. Die eigentliche Arbeit wird beim Verschmelzen der Teildatensätze verrichtet.

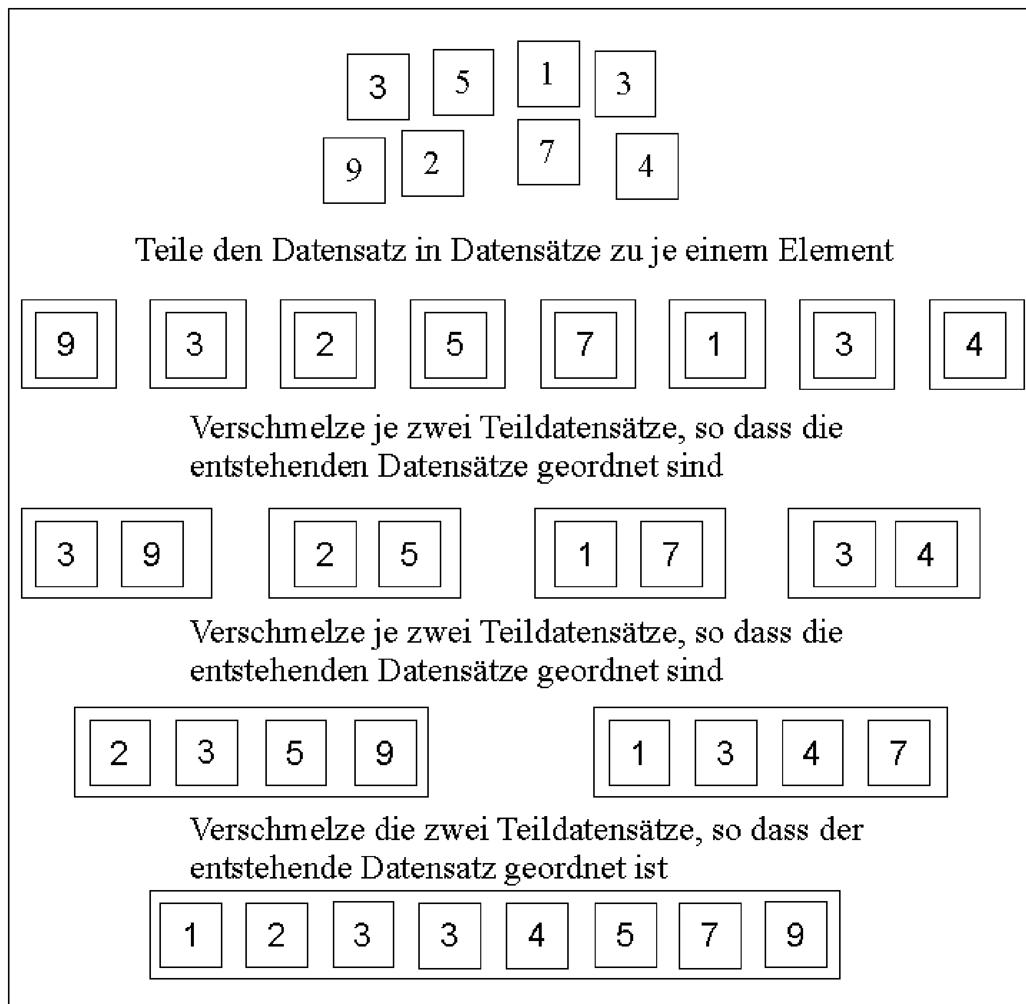


Abbildung 4.3: Mergesort

Kapitel 5

Anhang

1 Literatur zu Java und Programmierung

Zum Thema Programmieren, gerade auch zum Programmieren mit Java, gibt es sehr viele Bücher. Dieser Kurs soll auch ohne Bücher zu absolvieren sein. Die genannten Bücher decken einen größeren Inhalt als meine Lehrveranstaltung ab. Allerdings werden Sie, wenn Sie einen Studiengang gewählt haben, der Kurse in Programmierung enthält (das ist nicht nur Informatik!), auf die Dauer Bücher benötigen. Es lohnt sich also, in der Universitätsbibliothek zu stöbern und das ein oder andere Buch auszuleihen, um es anzuschauen. Ein gekauftes Buch ist natürlich immer besser, weil man Programmieren nicht während einer Leihfrist lernt und in ein eigenes Buch reinschreiben kann. Nachfolgend werden einige Bücher kurz vorgestellt. Aufgrund der angesprochenen Fülle von Büchern ist diese Auswahl zwangsläufig eher zufällig.

1. Sierra, K., Bates, B., Java von Kopf bis Fuß, O'Reilly, 1. deutsche Auflage 2006, ISBN 978-3897214484. Das Buch hat 720 Seiten und kostet 49,90€.
   Für ein Lehrbuch, kein Nachschlagewerk, ist das Buch recht umfangreich. Für den absoluten Anfänger kann dies ein Nachteil sein, für den Fortgeschrittenen ist es ein Vorteil. Es ist betont witzig geschrieben, was man lieben oder hassen kann. Die Autoren gehen davon aus, dass sich der Stoff so leichter merken lässt. Auf alle Fälle erfordert der witzige Schreibstil vom Leser gute Deutschkenntnisse!

2. Schiedermeier, R., Programmieren mit Java, Pearson Studium, 2. Auflage 2010, ISBN 978-3868940312. Das Buch hat 467 Seiten und kostet 39,95€. Für Programmieranfänger geeignet. Es werden zunächst bestimmte Konstrukte vorgestellt, die man auch in der imperativen Programmierung zunächst vorstellen würde, bevor tiefer in die objektorientierten Konzepte eingedrungen wird.

3. Barnes, D.J., Kölling, M., Java lernen mit BlueJ, Pearson Studium, 5. deutsche Ausgabe 2013, ISBN 978-3868949070. Das Buch hat 672 Seiten und kostet 39,95€.

Ein gutes Buch, um einen Einstieg in Java zu bekommen. Es verwendet grundlegend Objektorientierung. Es spricht alle wichtigen Bereiche an, ohne den Anspruch auf Vollständigkeit zu haben, was die Lesbarkeit verbessert. Das Buch baut allerdings auf die Entwicklungsumbebung „BlueJ“ auf. Will man es mit Eclipse nutzen, muss man einige Beispiele anpassen, da man in BlueJ viel ausprobieren kann, ohne eine main-Methode programmieren zu müssen.

4. Kölling, M., Einführung in Java mit Greenfoot: Spielerische Programmierung mit Java, Pearson Studium, 1. Auflage 2010, ISBN 978-3868949025. Das Buch hat 240 Seiten und kostet 29,95€. Dieses Buch benutzt als Grundlage Greenfoot. Es werden vorwiegend von vorgegebenen Grundgerüsten ausgehend Spiele programmiert. Greenfoot liefert neben einer IDE auch viele Bibliotheksfunktionen, die man gut gebrauchen kann. Darin liegt aber auch ein Problem: Man benutzt ständig Greenfoot-Funktionalitäten, die einem sonst nicht zur Verfügung stehen. Etwas überspitzt: Man lernt nicht Java-Programmieren, sondern eher Greenfoot-Programmieren. Das ist für angehende Studenten etwas zu wenig, aber wenn Sie Spaß an dem Konzept haben, können Sie aus diesem Buch trotzdem viel lernen.

5. Poetzsch-Hefter, A., Konzepte objektorientierter Programmierung mit einer Einführung in Java, Springer, 2. überarbeitete Auflage 2009, ISBN 978-3540894704. Das Buch hat 352 Seiten und kostet 29,95€. Das Buch behandelt einen ähnlich großen Stoffumfang wie [3], aber auf nur halb so vielen Seiten! Der Stoff ist also sehr komprimiert, was die Lesbarkeit erschwert, zumal das Buch auch weniger „Programmierrezepte“ vermitteln will, sondern „Programmierkonzepte“, also recht tief geht. Der Schreibstil ist sehr trocken. Alles in allem eher ein Buch für jemanden, der die Grundlagen von Java bereits beherrscht und jetzt vertiefen möchte.

6. Ullenboom, C., Java ist auch eine Insel, Rheinwerk Computing, 11. Auflage 2014, ISBN 978-3836228732. Das Buch hat 1306 Seiten und kostet 49,90€. Dieses Buch ist kein Lehrbuch, sondern eher ein Java-Lexikon für den fortgeschrittenen Programmierer. Sie bekommen das Buch auch kostenlos im Netz: `http://openbook.rheinwerk-verlag.de/javainsel/`

7. Güting, R. H., Dieker, S., Datenstrukturen und Algorithmen, Teubner, 3. Auflage 2004, ISBN 978-3519221210. Das Buch hat 396 Seiten und kostet 34,90€. Wie der Titel bereits ahnen lässt, geht es hier um eine abstraktere Sicht auf Probleme der Programmierung, die im Skript in den Kapiteln 4.2 oder 4.3 nur etwas anklingen.