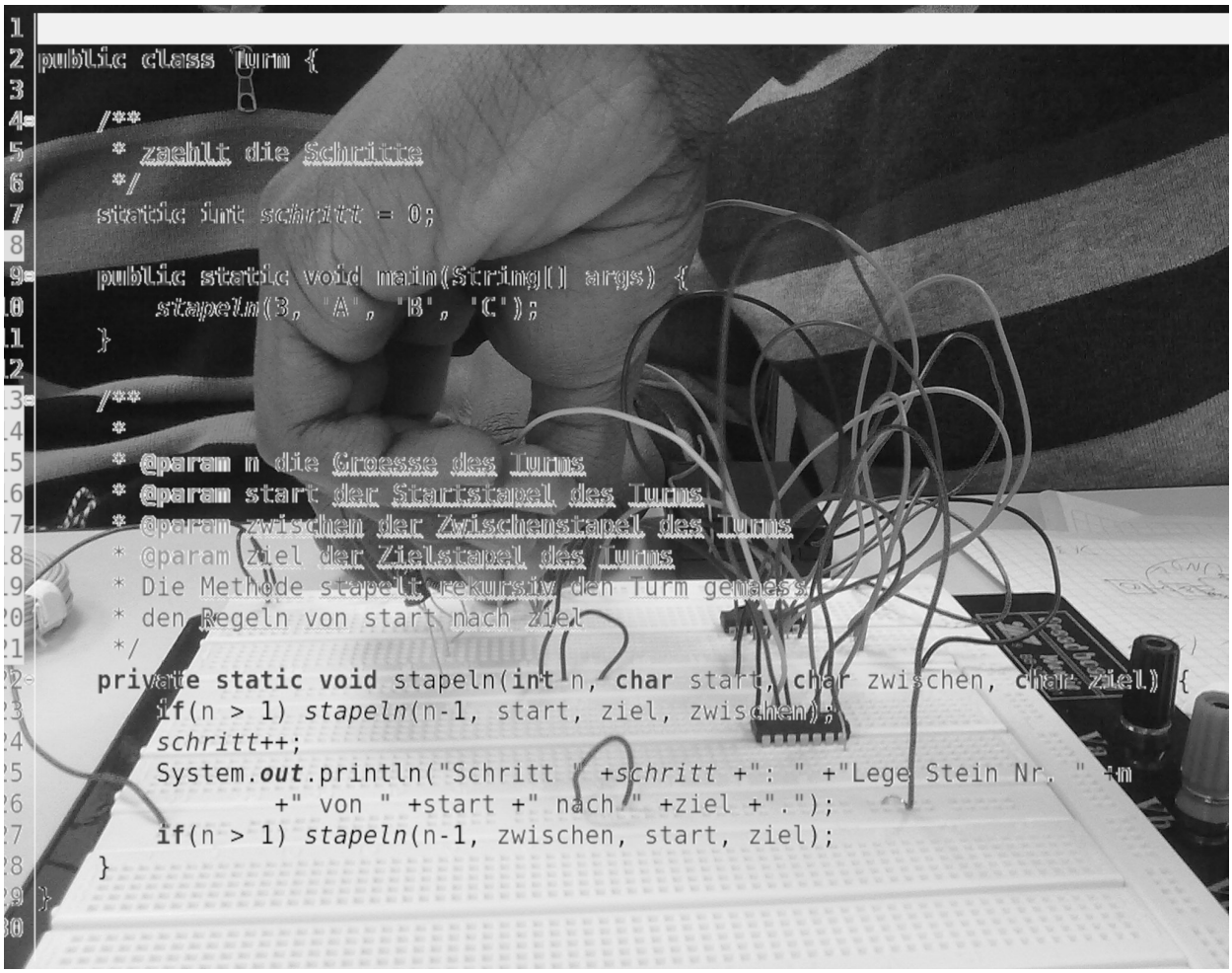
Einblicke in die Informatik

Skript zum Informatikunterricht von C. Jost am Studienkolleg der TU Darmstadt



Christof Jost

Version: 6. April 2025

Inhaltsverzeichnis

Vorwort

Zum Gebrauch dieses Skripts Dieses Skript enthält sicher einige Fehler. Der Autor übernimmt keine Haftung für die Richtigkeit der Angaben. Informationen über Fehler, sonstige Verbesserungsvorschläge und Kritik bitte an christof.jost„aatt“tu-darmstadt.de (statt „aatt“ schreiben Sie natürlich @).

Das Skript enthält Links zu Websites mit weitergehenden Informationen, Videos, Bildern und teilweise interaktiven Animationen. Links zu externen Inhalten im www sind durch blaue Schrift gekennzeichnet. Zum Zeitpunkt der Verlinkung waren die Ziele der Links frei von illegalen und anstößigen Inhalten. Der Autor kann aber die Linkziele nicht permanent überprüfen und übernimmt daher keinerlei Haftung bezüglich der verlinkten Seite und bittet bei Entdeckung unangemessener Linkziele um eine Mitteilung. Interne Links innerhalb des Skripts sind durch violette Schrift gekennzeichnet.

Wichtige Definitionen, Zusammenfassungen und dergleichen befinden sich in grau unterlegten, nummerierten Boxen. Aufgaben sollen Ihnen helfen, zu überprüfen, ob Sie das Gelesene verstanden haben. Sie haben unterschiedliche Schwierigkeitsstufen: Einige sind in Sekunden beantwortbar, andere erfordern, ein aufwändiges Programm zu schreiben. Weiterführende Informationen zu einem bestimmten Thema sind mit der Überschrift *Exkurs* gekennzeichnet und in kleinerer Schrift gesetzt. Sie können überlesen werden, ohne das Verständnis des übrigen Texts zu gefährden.

Das Skript dient zur Ergänzung des Unterrichts von C. Jost. Es ist nicht als alleinige Literatur für ein Selbststudium geeignet und kann den Unterricht nicht ersetzen. Beachten Sie, dass insbesondere der Unterrichtsgang, aber auch die einzelnen Inhalte im Skript nicht 1:1 denen im Unterricht entsprechen. Das gilt umso mehr für den Unterricht der Kolleg:innen am Studienkolleg.

1 Informatik und Computer

1.1 Informatik und ihre Teilgebiete

Definition der Informatik Informatik ist ein Kofferwort aus den Wörtern *Information* und *Automatik*. Informatik ist also die Wissenschaft von der automatischen Verarbeitung von Information. Für die automatische Verarbeitung braucht man Maschinen, diese nennt man *Rechner* oder *Computer*. Wenn Sie im englischen Sprachraum Informatik studieren würden, würde das Fach *Computer Science* heißen. Dies unterstreicht, dass die Beschäftigung mit Computern integraler Bestandteil des Fachs Informatik ist. Es gibt Autoren, die die Ansicht vertreten, Informatik sei die Wissenschaft von der (Verarbeitung der) Information und der Computer sei nur eines von vielen denkbaren Hilfsmitteln, wie das Mikroskop in der Biologie. So, wie die Biologie nicht die Wissenschaft von den Mikroskopen sei, sei Informatik nicht die Wissenschaft von den Computern. Diese Ansicht ist nicht haltbar.

Teilgebiete der Informatik Die Informatik umfasst unstrittig die folgenden Teilgebiete:

- Praktische Informatik. Die Praktische Informatik beschäftigt sich beispielsweise mit Programmierung, Datenbanken, Betriebssystemen oder Rechnernetzwerken. Da solche zentralen Themen, die jedem geläufig sind, zur Praktischen Informatik zählen, wird Informatik manchmal zu Unrecht auf dieses Teilgebiet verengt. Grundlegende Programmierkenntnisse sind in allen technischen Studiengängen, oft aber auch in der Physik und anderen Naturwissenschaften und in der Mathematik gefragt. Daher liegt hier der Schwerpunkt des Informatikkurses am Studienkolleg Darmstadt.

- Technische Informatik. Die Technische Informatik beschäftigt sich mit den hardwareseitigen Grundlagen der Informatik und ist somit das Grenzgebiet der Informatik zur Elektrotechnik. Der Technischen Informatik, insbesondere der Boolesche Algebra, ist das Kapitel 4 dieses Skripts gewidmet.

- Theoretische Informatik. Die Theoretische Informatik beschäftigt sich beispielsweise mit der Frage, welche Fragen überhaupt durch Berechnungen beantwortet werden können oder wie effizient oder schnell Probleme gelöst werden können. Die Theoretische Informatik lehnt sich in der Fragestellung und der Methodik eng an die Mathematik an. Am Studienkolleg streifen wird die Theoretische Informatik bei der Beschäftigung mit Komplexität in Kapitel 6.

Ohne Anspruch auf Vollständigkeit könnte man auch über die folgenden weiteren Teildisziplinen der Informatik sprechen:

- Angewandte Informatik. Darunter könnte z.B. die Bioinformatik, die Computerlinguistik oder die Wirtschaftsinformatik fallen oder auch die Inhalte des Studiengangs *Computational Engineering*, letztlich alles, was Informatik auf andere Fächer anwendet. Die große Breite, die der Begriff abdeckt, macht ihn aber ein wenig willkürlich. Sie wenden Informatik an, wenn Sie im www surfen, eine E-Mail oder eine WhatsApp-Nachricht verschicken oder einen Text am Computer verfassen. Ist das alles schon Angewandte Informatik?

- In jüngerer Zeit wird das Gebiet der *künstlichen Intelligenz* immer wichtiger. Es hat unzweifelhaft viel mit Informatik zu tun. Nicht geklärt ist aber, ob es ein Teilgebiet der Informatik ist oder in ein anderes Teilgebiet der Informatik einsortiert werden müsste.

1.2 Hardware

Der Prozessor. Der Prozessor oder *CPU* für *Central Processing Unit* ist das „Gehirn“ des Computers, er erledigt die eigentliche Rechenarbeit. Herkömmliche Prozessoren rechnen sehr gut mit ganzen Zahlen, womit man bisher den Hauptteil der Aufgaben gut erledigen kann. Für die Aufbereitung der Daten für die Darstellung am Bildschirm benötigt man viele Berechnungen mit Gleitkommazahlen. Dafür gibt es spezielle Grafikprozessoren oder *Graphics Processing Units, GPUs*, die diese Aufgabe besser erledigen können. Manche Anwendungen, die viele Gleitkommazahlberechnungen erfordern, benutzen die GPU, obwohl es gar nicht um graphische Daten geht. Dazu gehören beispielsweise Anwendungen der künstlichen Intelligenz (*KI*). Mittelfristig ist zu erwarten, das Rechner mit speziellen Prozessoren für die Anwendungen der KI ausgestattet werden.

Speicher. Um Daten verarbeiten zu können, muss man diese speichern können. Dafür benötigt man entsprechende Speicherbausteine. Es gibt *persistente* oder *nichtflüchtige* Speicher, die ihre Inhalte auch bei abgeschaltetem Strom behalten und nicht persistente oder *flüchtige* Speicher, die ohne elektrische Energieversorgung „alles vergessen“. Da der Zugriff auf persistente Speicher in der Regel viel langsamer ist als der Zugriff auf nicht persistente Speicher, sind in allen üblichen Rechnern beide Arten von Speichern verbaut. Beim *Booten* (oder deutsch: *Hochfahren*) eines Rechners werden im wesentlichen Daten für das Betriebssystem vom langsamen, persistenten Speicher in den schnellen, nicht persistenten Speicher zu übertragen, um damit arbeiten zu können.

Persistente Speicher waren früher meistens Festplattenlaufwerke, kurz auch einfach *Festplatten* genannt, englisch *Hard Disk Drives, HDD*, heute sind die schnelleren Solid State Drives, SSDs der Standard. Der schnelle, nicht persistente Speicher wird im Deutschen meistens Arbeitsspeicher genannt, auch der Ausdruck *RAM* für *Random Access Memory* ist üblich. Allerdings betont der Begriff Random Access Memory („Speicher mit wahlfreiem Zugriff “), dass die zu speichernden Daten willkürlich auf die einzelnen Speicherzellen verteilt werden können, ohne dass das einen Einfluss auf die Verarbeitungsgewschindigkeit der Daten hat. Dies ist bei den modernen SSDs, im Gegensatz zu den alten Festplatten, aber auch der Fall, so dass dieser Namen die Speicherarten nicht mehr treffend unterscheidet.

Weitere Komponenten Neben Prozessor und Speicher besitzt ein Computer noch viele weitere Komponenten. Zu nennen wären hier u.a. die Schnittstellen (z.B. USB, HDMI) zum Anschluss von Peripherie, vor allem zum Anschluss von Ein- und Ausgabegeräten wie Tastatur, Maus oder Bildschirm. Wichtig ist auch die Energieversorgung über ein Netzteil und, bei tragbaren Geräten, einen Akkumulator. Das Netzteil muss, zusammen mit weiteren Bausteinen die Versorgung durch 230 V Wechselspannung aus der Steckdose in verschiedene für die einzelnen Komponenten notwendigen Spannungen umwandeln. Alle Komponenten eines Computers benötigen Gleichspannung von wenigen Volt.

2 Erste Schritte in Java

2.1 Grundlagen

2.1.1 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, z.B. auch Befehle für einen Prozessor, müssen auf diese Form gebracht werden. 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, siehe Abbildung 2.1. Diese Übersetzung erledigt ein Programm, das man *Compiler* nennt. Statt *übersetzen* sagt man meistens *kompilieren*.

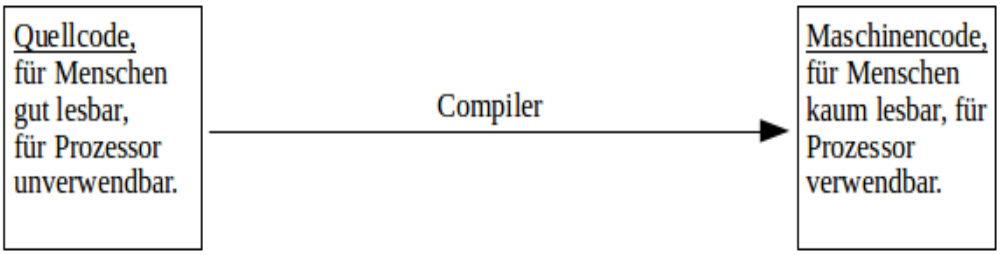


Abbildung 2.1: Codeumwandlung

2.1.2 Bytecode bei Java

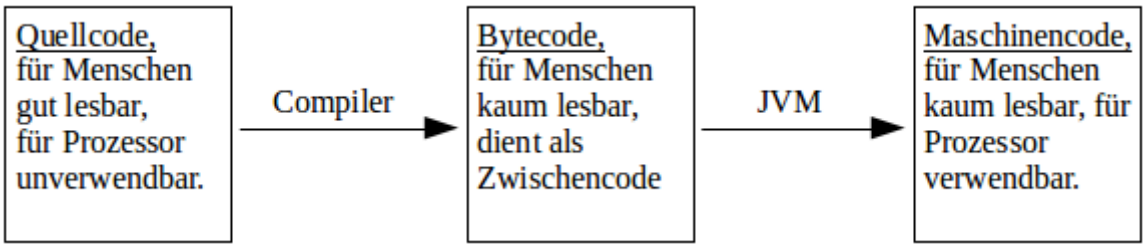


Abbildung 2.2: Codeumwandlung bei Java

Liegt der Quellcode in einer Programmiersprache vor, dann kann man ihn mit einem Compiler für einen bestimmten Rechnertyp kompilieren. Rechnertypen sind hier in der Praxis meistens Betriebssysteme: Sie können das Programm z.B. für Windows oder für Linux kompilieren.

Besser wäre es, man könnte das fertig kompilierte Programm auf allen Rechnern benutzen. Hier hat Java sene Vorteile. Java-Programme werden in *Bytecode* kompiliert. Der Bytecode ist der Maschinencode für die *Java Virtual Machine*, kurz *JVM*. Die JVM ist ein virtueller Computer, der als Software auf einem tatsächlichen Rechner installiert werden kann. Jeder Computer, der die JVM istalliert hat,

kann Java-Bytecode ausführen. Die JVM gehört auf den meiste Computern zur Standardausstattung. Man könnte den Bytecode auch als einen Zwischencode erklären, der dann von der JVM in die Maschinensprache des betreffende Computers überführt wird, wie in Abbildung 2.2 gezeigt. Java-Quellcode wird in Dateien mit der Endung *.java* abgespeichert, Bytecodedateien haben die Endung *.class*.

2.1: Exkurs: Bedeutung von Java in der Computerwelt

Geräte wie eine Waschmaschine werden mit einfachen Prozessoren gesteuert. Die Programme dafür sind relativ einfach und wurden daher lange mit nur wenigen Hilfsmitteln mehr oder weniger direkt in Maschinencode erstellt. Mit der Zeit konnten die Waschmaschinen aber immer mehr und die Erstellung der Programme wurde komplizierter. Daher sollte anfangs der 90er-Jahre des 20. Jahrhunderts eine Plattform geschaffen werden, die es ermöglichen sollte, mit demselben Programm unterschiedliche Prozessoren anzusprechen: Die Java-Technologie. Die beteiligten Entwickler hatten angeblich eine Vorliebe für starken Java-Kaffee, weshalb ihr Projekt diesen Namen bekam.
Etwa zur selben Zeit wurde das World Wide Web immer wichtiger. Neben reinem Text als Inhalt sollten auch zunehmend kleine Programme für Animationen und dergleichen zur Verfügung gestellt werden. Diese mussten aber plattformunabhängig funktionieren, denn der Anbieter einer Webseite muss mit Besuchern mit völlig verschiedenen Systemen rechnen. Hier konnte Java gute Dienste tun und wurde dadurch sehr beliebt.
Java ist bis heute eine der beliebtesten Programmiersprachen. Sie wird in vielen Universitäten, z.B. auch an der Technischen Universität Darmstadt, in den Einführungssemestern im Informatikstudium verwendet. Professionelle Werkzeuge zur Java-Entwicklung sind kostenlos erhältlich. Auch wenn die Bedeutung von Java in Webbrowsern weitgehend verschwunden ist, ist ihr wichtigster Vorzug die Plattformunabhängigkeit. Sie wird daher vor allem dann eingesetzt, wenn unterschiedliche Systeme miteinander verknüpft werden sollen.

2.1: Aufgabe: 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. 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?
4. 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?

2.1.3 Entwicklungsumgebungen

Um ein Programm zu schreiben und zu testen, benötigt man ein paar Hilfsmittel:

- Man braucht einen *Editor*, eine Art primitive Textverarbeitung, um den Programmcode überhaupt zu schreiben.
- Einen Compiler, der den in der Programmiersprache geschriebenen Code vor der Ausführung in Maschinencode oder – wie bei Java – einen Zwischencode übersetzt, fachsprachlich *kompiliert*.
- Man benötigt eine Umgebung, in der man das Programm oder auch nur Teile davon problemlos testen kann.

Am besten, man installiert eine *integrierte Entwicklungsumgebung*, neudeutsch *IDE* von *Integrated Development Environment*. Die IDE bringt einen Editor mit, der unter anderem durch durch farbliche Kennzeichnungen hilft, den Überblick zu behalten, außerdem integriert sie gegebenenfalls den Compiler und die Laufzeitumgebung. Damit braucht man nur noch die IDE, um Programme zu erstellen. Es gibt viele IDEs für die Java-Entwicklung. Am Studienkolleg setzen wir *Eclipse* ein.

2.2: Exkurs: Weitere Entwicklungsumgebungen

Statt Eclipse wird für die Java-Entwicklung oft die IDE IntelliJ IDEA der Firma JetBrains verwendet. Beliebst ist auch die IDE Netbeans, die genau wie Eclipse ein Open-Source-Projekt ist. Weitere IDEs für Java möchten Anfängern einen unkomplizierten Einstieg bieten. In diesem Zusammenhang wäre BlueJ zu nennen oder der Java-Editor. Eine Sonderrolle nimmt Greenfoot ein: Hier kommt mit der IDE eine Symmlung einfacher Spiele in Java, die man verändern und weiter entwickeln kann.
Wir werden im Studienkolleg Eclipse verwenden. Ich rate Ihnen, auf Ihrem privaten Rechner ebenfalls Eclipse zu verwenden, damit Sie sich nicht in mehrere IDEs einarbeiten müssen. Ausnahmen:

- Sie können bereits Java programmieren und benutzen eine andere IDE.
- Sie haben keinen Laptop oder sonstigen vollwertigen Computer und müssen alles auf einem Tablet erledigen und können dort nur Online-IDEs benutzen.

Beachten Sie aber, dass Sie während der Klausuren auf jeden Fall auf den gestellten Systemen des Studienkollegs arbeiten, auf denen ausschließlich Eclipse läuft. Sie sollten daher mindestens im Unterricht mit Eclipse arbeiten.

2.2: Aufgabe: Tools für die Java-Programmierung auf dem eigenen Rechner

Eclipse findet man unter https://www.eclipse.org/. Installieren Sie Eclipse auf Ihrem privaten Rechner. Beachten Sie die Lizenzbedingungen.

2.1.4 Hallo Welt!

Die erste Aufgabe in einer neuen Programmiersprache ist traditionell, den Text *Hallo Welt!* auszugeben, um einen ersten Eindruck von der Syntax zu bekommen. Dafür erzeugt man in Eclipse ein neues Java-Projekt (File -> New -> JavaProject) und gibt dem Projekt einen Namen, z.B. *Test1*. In diesem Projekt legt man einen Klasse an(File -> New ->Class oder, kürzer, die grüne Schaltfläche mit dem „C“). Eine Klasse ist, zumindest für diesen Einstieg, eine Datei mit Java-Code. Der Name der Klasse sollte gemäß den Java-Konventionen mit einem Großbuchstaben beginnen. Er darf keine Leerzeichen enthalten. Als Beispiel wähle ich *MeineErsteKlasse*. Außerdem muss man im selben Fenster das Package benennen, das ist der Ordner, in dem die Datei abgelegt wird. Package-Namen sollen mit Kleinbuchstaben beginnen. Wählen Sie z.B. *test*. Unter der Frage „Which method stubs would you like create?“ sollte man das Häkchen bei *public static void main(String[] args)* setzen, andernfalls muss man diese Zeile, die in jedem Java-Programm vorkommt, selbst tippen. Dann bestätigt man durch Betätigen der Schaltfläche *Finish* und erhält den folgenden Code:

Listing 2.1: Meine Erste Klasse

```
1 package test;
2
3 public class MeineErsteKlasse {
4
5     public static void main(String[] args) {
6         // TODO Auto-generated method stub
7     }
8 }
```

Die erste Zeile ist die Überschrift der Klasse, deren Inhalt mit der öffnenden geschweiften Klammer beginnt. Leerzeilen sind für den Programmablauf unerheblich und dienen nur der Übersichtlichkeit. Zeile 3 ist die Überschrift, fachsprachlich *Signatur* der *main()-Methode*, mit der jedes Java-Programm gestartet wird. Die Bedeutung der einzelnen Begriffe erfolgt später. Die eigentliche Methode, man spricht auch vom *Rumpf* der Methode, beginnt dann mit der öffnenden geschweiften Klammer. Die Zeichen // beginnen einen Kommentar. Kommentare sind Notizen für den Programmierer und beeinflussen nicht die Programmausführung. Die schließende Klammer in der nächsten Zeile beendet die main()-Methode, die weitere schließende Klammer beendet die Klasse.

2.3: Exkurs: default Package

Auf die Angabe eines Packages kann verzichtet werden, wenn man beim Erstellen des Projekts den Haken bei *Create module-info.java file* entfernt oder nachträglich das angelegte module-info.java file löscht, bevor man die Klasse anlegt. Die Klasse wird dann ganz oben im Verzeichnis *src* (für englisch *source* für *Quelle*, *Quellcode*) angelegt. Bei größeren Projekten sollte man das nicht tun, da man damit auf Strukturierung verzichtet. Bei Projekten aus einer oder wenigen Klassen ist das in Ordnung. Ich werde oft so vorgehen und die weiteren Beispiele kommen ohne Package aus.

2.3: Aufgabe: Hallo Welt!

Löschen Sie den Kommentar aus der Klasse und fügen Sie statt dessen die folgende Anweisung ein: *System.out.println("Hallo Welt!");*. Speichern (Diskettensymbol in Eclipse) und starten Sie (grüner Pfeil) das Programm. Das Programm sollte den Text *Hallo Welt!* ausgeben. Wenn es nicht funktioniert, versuchen Sie, die Fehlermeldung in Eclipse zu verstehen und den Fehler zu beheben.

2.4: Aufgabe: Ausgabeanweisung und String-Konkatenation

In der folgenden Aufgabe lernen Sie eine Variante der Ausgabeanweisung kennen und lernen einiges über das Aufbauen von Strings.
a) Schreiben Sie folgende main()-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 }
```

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 }
```

Was ist der Unterschied zwischen *println* und *print*?
b) Wie wird der Ausdruck *2 * 2 + 3* in den folgenden main()-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 }
```

c) Schreiben Sie folgende 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 Bedeutungen hat das Zeichen „+“? Was passiert bei *+(2*2+3)* ohne die Klammern?

2.5: Aufgabe: Klammern

In der Java-Programmierung kamen bereits im ersten einführenden Beispiel drei Arten von Klammern vor, in der fortgeschrittenen Programmierung kommt eine vierte Art dazu. Um über Programmierung sprechen zu können und um zu verstehen, was andere Menschen über Programmierung sagen, müssen Sie die Klammern benennen können. Lernen Sie also die folgenden Begriffe!

- () sind *runde Klammern*. Sie sind die häufigsten Klammern. Wenn man einfach *Klammern* sagt, dann sind normalerweise runde Klammern gemeint.
- [ ] sind eckige Klammern.
- {} sind *geschweifte Klammern*.
- Die *spitzen Klammern* <> kamen bisher noch nicht vor.

Den Ausdruck $2 + (3 \cdot 5)$ kann man als „Zwei plus in (runden) Klammern drei mal fünf“ lesen, da so klar ist, von wo bis wo der geklammerte Ausdruck geht. Wenn das nicht klar ist, dann muss man „Zwei plus runde Klammer auf drei mal fünf runde Klammer zu“ lesen. Die Zeile *public static void main(String*[ ] *args) {* liest man „public static void main runde Klammer auf String eckige Klammer auf eckige Klammer zu args runde Klammer zu geschweifte Klammer auf“.

2.2 Methoden

2.2.1 Strukturieren mit Methoden

Wozu Methoden? Sie werden am Studienkolleg Programme mit mehreren Hundert Zeilen Code schreiben. Ein Profi-Programmierer schreibt Code für Programme, die aus Millionen Zeilen Code bestehen, denken Sie z.B. an ein ausgefeiltes Textverarbeitungsprogramm. Es ist nicht möglich, solche Programme als Abfolge einzelner Anweisungen zu erstellen. Ein Programm muss innerhalb verschiedener Ebenen strukturiert werden. Die unterste Ebene der Struktur ist die Zusammenfassung von Anweisungen zu *Methoden*. Hier ein erstes Beispiel:

Listing 2.2: Erste Methode

```
1  public class BeispielMethode {
2
3      public static void main(String[] args) {
4          halloSagen();
5          halloSagen();
6      }
7
8      private static void halloSagen() {
9          System.out.println("Hallo Welt!");
10     }
11 }
```

Das Programm beginnt, wie jedes Java-Programm, mit der main()-Methode. In der main()-Methode wird die Methode *halloSagen()* in der Zeile 4 *aufgerufen*. Wäre diese Methode nicht in den Zeilen 8 bis 10 *deklariert*, würde der Methodenaufruf zu einem Compilerfehler führen. So aber springt die Programmausführung von Zeile 4 in die Zeile 8 und arbeitet dort die Anweisungen in der Methode ab, hier also die Ausgabeanweisung in Zeile 9. Nach der Beendigung der Methode in Zeile 10 springt die Ausführung zurück in die *aufrufende Methode*, hier die main()-Methode und arbeitet dort die nächste Zeile, die Zeile 5, ab. Diese Zeile enthält wieder einen Aufruf der Methode halloSagen(), so dass die

Methode abermals abgearbeitet wird. Nach dem Rücksprung in die aufrufende main()-Methode endet das Programm mit dem Ende der main()-Methode in Zeile 6. Insgesamt gibt das Programm also zweimal den Text *Hallo Welt!* aus.

2.6: Aufgabe: Deklaration und Aufruf einer Methode

Der obige Text enthält die wichtigen Fachbegriffe *Methodendeklaration* und *Methodenaufruf*. Stellen Sie sicher, dass Sie die Begriffe verstanden haben und angeben können, wo sich im obigen Programm die Deklaration und die Aufrufe der Methode *halloSagen()* befinden.

Methoden mit Parametern Manche Methoden benötigen zusätzliche Informationen, um ihre Aufgabe zu erledigen. Diese zusätzlichen Informationen werden als *Parameter* übergeben. Sie möchten z.B. nicht allen Ihren Freund:innen „Hallo Welt!“ zurufen, sondern jeden mit Namen begrüßen. Dies leistet das folgende Programm:

Listing 2.3: Methode mit Parameter

```
1  public class BeispielMethode {
2
3      public static void main(String[] args) {
4          halloSagen("Tarzan");
5          halloSagen("Isolde");
6      }
7
8      private static void halloSagen(String name) {
9          System.out.println("Hallo " +name +"!");
10     }
11 }
```

Die Deklaration der Methode halloSagen() enthält nun einen *formalen Parameter* des Typs *String* und dem Bezeichner *name*. Beim Aufruf der Methode muss jeweils ein *aktueller Parameter* angegeben werden, der dann konkret als Parameter verwendet wird. Das Programm gibt dann die Zeilen „Hallo Tarzan!“ und „Hallo Isolde!“ aus.

2.4: Exkurs: Falscher Freund: Aktueller Parameter

Das englische Wort *actual* ist ein falscher Freund. Es ähnelt dem deutschen Wort *aktuell*, muss aber korrekt mit dem deutschen Wort *tatsächlich* übersetzt werden. Die Fachbegriffe in der Informatik kommen meistens aus der englischen Sprache, so auch der englische Begriff *actual parameter*, der mit *tatsächlicher Parameter* oder *konkreter Parameter* ins Deutsche hätte übersetzt werden müssen. Unglücklicherweise wurde er mit *aktueller Parameter* übersetzt.

2.7: Aufgabe: formaler und aktueller Parameter einer Methode

Der obige Text enthält die wichtigen Fachbegriffe *formaler Parameter* und *aktueller Parameter*. Stellen Sie sicher, dass Sie die Begriffe verstanden haben und angeben können, wo im obigen Programm formale und aktuelle Parameter vorkommen.

Eine Methode kann auch mehrere Parameter haben. Die Methode *addieren()* im folgenden Programm hat zwei Parameter des Typs *int*. Der Typ *int* steht für *integer* und bezeichnet eine ganze Zahl.

Listing 2.4: Methode mit zwei Parametern des Typs int

```
1  public class BeispielMethode {
```

```
2
3       public static void main(String[] args) {
4           addieren(20, 22);
5           addieren(−5, 2);
6       }
7
8       private static void addieren(int summand1, int summand2) {
9           System.out.println(summand1 + summand2);
10      }
11  }
```

2.8: Aufgabe: Methode addieren()

Nennen Sie die Ausgabe des obigen Programms. Wenn Sie nicht sicher sind, probieren Sie es aus.

Methoden mit Rückgabe Die obige Methode gibt die Summe der beiden Parameter aus. Häufig hat man Situationen, in denen man ein Ergebnis nicht ausgeben, sondern weiter verarbeiten möchte. Dass muss die Methode eine *Rückgabe* machen. Die Rückgabe wird durch das Schlüsselwort *return* eingeleitet. In der Deklaration der Methode muss angegeben werden, dass die Methode eine Rückgabe macht und von welchem Typ die Rückgabe ist. Dieser *Rückgabetyp* in der Methodendeklaration lautete in den bisherigen Beispielen stets *void*. Dieses englische Wort bedeutet *leer* und somit, dass die Methode keine Rückgabe hat. Im folgenden Programm hat die Methode *addieren()* den Rückgabetyp *int*, d.h. die Methode gibt eine ganze Zahl zurück.

Listing 2.5: Methode mit Rückgabe

```
1   public class BeispielMethode {
2
3       public static void main(String[] args) {
4           addieren(20, 22);
5           addieren(−5, 2);
6       }
7
8       private static int addieren(int summand1, int summand2) {
9           return summand1 + summand2;
10      }
11  }
```

Lässt man das Programm laufen, passiert gar nichts. Die Methode *addieren()* wird zwar zweimal aufgerufen und rechnet dabei eine Summe aus. Diese wird aber niemals ausgegeben, daher sehen wir kein Ergebnis. Wir könnten die Ausgabe in der main()-Methode unterbringen:

Listing 2.6: Methode mit Rückgabe, Ausgabe in der main()-Methode

```
1   public class BeispielMethode {
2
3       public static void main(String[] args) {
4           System.out.println(addieren(20, 22));
5           System.out.println(addieren(−5, 2));
6       }
7
8       private static int addieren(int summand1, int summand2) {
9           return summand1 + summand2;
```

```
10      }
11  }
```

2.9: Aufgabe: Ausgabe und Rückgabe

Erklären Sie den Unterschied zwischen *Ausgabe* und *Rückgabe*.

2.5: Exkurs: Methoden, Prozeduren, Funktionen

In anderen Programmiersprachen wird manchmal statt des Begriffs *Methode* der Begriff *Funktion* verwendet. In manchen Programmiersprachen werden nur Methoden mit Rückgabe *Funktion* genannt, Methoden ohne Rückgabe werden dann *Prozedur* genannt. In diesem Skript wird, die in Java üblich, nur der Begriff *Methode* verwendet.

2.2.2 Ein erster Blick auf die Objektorientierung

Die *imperative Programmierung* sieht ein Programm als Ansammlung von Befehlen, die gegebenenfalls in Methoden gruppiert werden können. Bei großen Programmen führt das zu kompliziertem, kaum wartbarem Code. Wir Menschen sind es gewohnt, in unserer Umgebung *Objekte* zu identifizieren. Diese Objekte haben bestimme Eigenschaften, in der Programmierung meistens *Attribute* oder *Instanzvariablen* genannt und können bestimmte Dinge tun, in der Programmierung meistens durch Methoden, die ein Objekt ausführen kann, dargestellt. In der *objektorientierten Programmierung* werden in einem Programm Objekte erschaffen. Diese tun etwas. Sie können z.B. die Erschaffung anderer Objekte initiieren, selbst etwas tun oder andere Objekte dazu bringen, etwas zu tun. In der objektorientierten Programmierung sind Klassen Baupläne für Objekte. Man kann dann beliebig viele Objekte nach diesem Bauplan erschaffen.

Beispiel: Sie schreiben ein Programm für die Verwaltung der Studierenden am Studienkolleg. Dann würden Sie eine Klasse *Student* programmieren. Attribute wären beispielsweise der Name, der Vorname, das Geburtsdatum, die Adresse oder die abgelegten Prüfungen mit den jeweiligen Noten. Man müsste Methoden vorsehen, um Prüfungen abzulegen oder die Adresse ändern zu können. Sie können dann für jede:n Student:in im Studienkolleg ein Objekt der Klasse erschaffen und darin die betreffenden Werte speichern.

Die Objektorientierung lohnt sich, wenn man mehrere, von der Art gleiche, im Detail aber verschiedene Objekte hat wie im obigen Beispiel die Studierenden. Bei unseren Programmen war das bisher nicht der Fall und diese Programme waren nicht objektorientiert. Allerdings ist Java eine objektorientierte Programmiersprache, und man muss die Objektorientierung mit dem Schlüsselwort *static* explizit abschalten, wenn man sie nicht haben möchte. Möchte man im Beispiel 2.6 in der Methode *addieren()* das Wort *static* los werden, muss man das Programm objektrorientiert schreiben. Das könnte so aussehen:

Listing 2.7: Objektrorientierte Methode

```
1  public class Beispiel{
2
3      public static void main(String[] args) {
4          Beispiel b = new Beispiel();
5          System.out.println(b.addieren(20, 22));
6          System.out.println(b.addieren(−5, 2));
7      }
8
9      private int addieren(int summand1, int summand2) {
10         return summand1 + summand2;
11     }
12 }
```

In der Deklaration der Methode fehlt in der *Signatur* (Zeile 9) das Schlüsselwort *static*. Daher gehört die Methode nun zu Objekten der Klasse *Beispiel* und muss auf einem solchen Objekt aufgerufen werden.

Daher wird in der main()-Methode in Zeile 4 zuerst ein solches Objekt mit dem Bezeichner *b* erzeugt. Die Methode wird dann in den Zeilen 5 und 6 auf diesem Objekt aufgerufen, indem man das Objekt, auf dem die Methode aufgerufen wird, angibt, gefolgt von einem Punkt und dem Methodenaufruf.

Da wir nicht mehrere Objekte der Klasse benötigen, hat die objektorientierte Version keine Vorteile, außer dass wir das Schlüsselwort *static* weglassen können. Dafür haben wir größeren Aufwand zur Erzeugung des Objekts und dem Aufruf der Methode. Im weiteren Verlauf werden wir in den Beispielen mal die eine, mal die andere Version verwenden.

2.2.3 Methodensignaturen

Signatur und erweiterte Signatur Die erweiterte Signatur der Methode *addieren()* im vorigen Beispiel 2.7 lautet

Listing 2.8: erweiterte Signatur

```
9    private int addieren(int summand1, int summand2)
```

Die Signatur im engeren Sinne lautet nur *addieren(int summand1, int summand2)*. Der Zugriffsmodifizierer und der Rückgabtyp gehören also zur erweiterten Signatur, nicht aber zur Signatur. Achtung: Der Gebrauch der Begriffe ist nicht streng. Oft wird einfach nur von der *Signatur* gesprochen, auch wenn gegebenenfalls die erweiterte Signatur gemeint ist.

Bezeichner, Parameterliste und Rückgabetyp Schauen wir uns die Teile der erweiterten Signatur an: *addieren* ist der *Bezeichner* oder *Name* der Methode. Danach kommt in den runden Klammern die Parameterliste, in vielen Fällen auch nur ein Parameter oder gar kein Parameter. Beachten Sie, dass die Klammern geschrieben werden müssen, auch wenn eine Methode keinen Parameter hat. Andernfalls erkennt der Compiler den Text nicht als Methodensignatur. Vor dem Bezeichner steht der *Rückgabetyp*, in diesem Beispiel *int* für eine ganze Zahl. Wenn eine Methode keine Rückgabe besitzt, dass muss als Rückgabetyp *void* angegeben werden.

Zugriffsmodifizierer Ganz vorne in der erweiterten Signatur steht der *Zugriffsmodifizierer*. *public* bedeutet, dass die Methode von allen Stellen des Programms aus verwendet werden kann. Im Gegensatz dazu kann eine Methode mit dem Zugriffsmodifizierer *private* nur innerhalb derselben Klasse aufgerufen werden. Dies ist wichtig bei großen Softwareprojekten: Eine Methode mit dem Zugriffsmodifizierer *private* kann jederzeit verändert oder auch wieder gelöscht werden, da dies nur Auswirkungen innerhalb derselben Klasse hat, was nachvollzogen werden kann. Eine Methode mit dem Zugriffsmodifizierer *public* kann aber in völlig anderen Klassen, die vielleicht ein anderer Programmierer geschrieben hat, verwendet werden. Eine spätere Änderung kann Probleme verursachen, daher sollte man nur dann Methoden *public* setzen, wenn es nötig ist. So lange wir aber am Studienkolleg nur kleine Programme schreiben, die nur aus einer oder wenigen Klassen bestehen und immer nur eine Programmierer:in haben, spielt das keine Rolle. Außer *public* und *private* gibt es noch den Standard-Zugriffsmodifizierer, wenn man gar nichts schreibt und den Zugriffsmodifizierer *protected*. Sie unterscheiden sich von *public* und *private*, wenn es in größeren Projekten weitere Ebenen der Strukturierung des Projekts gibt.

Erweiterte Signatur der *main()*-Methode Wir können nun einen genaueren Blick auf die main()-Methode werfen, mit der jedes Java-Programm beginnt.

Listing 2.9: Signatur der main()-Methode

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

Sie hat den Zugriffsmodifizierer *public*. Das muss so sein, denn zum Programmstart wird die Methode immer von außerhalb aufgerufen. Sie ist statisch, denn unmittelbar nach dem Start des Programms existieren keine Objekte, auf denen eine Methode aufgerufen werden könnte. Sie hat den Rückgabetyp *void*, das heißt, sie hat keine Rückgabe. Das ist klar: Endet die main()-Methode, dann endet das Programm. Es wäre niemand mehr da, um die Rückgabe entgegen zu nehmen. Der Bezeichner lautet *main*. Es gibt einen Parameter des Typs *String*[ ], das ist eine Sammlung von Strings, also Zeichenketten.

Der Bezeichner des Parameter heißt *args* von englisch *arguments*, deutsch *Argumente*, ein anderes, im Deutschen eher unübliches Wort für *Parameter*. Der Bezeichner des Parameters muss nicht *args* heißen, Sie können einen anderen Namen verwenden. Alle anderen Teile der erweiterten Signatur der main()-Methode müssen genau so sein.

Wir werden den Parameter der main()-Methode nicht verwenden. Er findet Verwendung, wenn ein Java-Programm, zum Beispiel für die Verbindung unterschiedlicher Systeme, die Ergebnisse eines anderen Programms beim Start entgegen nehmen muss.

Überladen Zwei verschiedene Methoden in einer Klasse haben meistens verschiedene Bezeichner. Manchmal aber kann es sinnvoll sein, dass zwei verschiedene Methoden denselben Bezeichner haben. Stellen Sie sich z.B. vor, Sie müssen in einem Programm oft die Summe von zwei oder drei Zahlen bilden. Dann ist es sinnvoll, beide Methoden, die eine mit zwei formalen Parametern, die andere mit drei formalen Parametern, *addieren()* zu nennen. Dies, man nennt es *Überladen (des Methodenbezeichners)* ist möglich, denn beim Aufruf der Methode kann der Compiler an der Anzahl der aktuellen Parameter erkennen, ob er die Methode mit zwei oder die Methode mit drei formalen Parametern verwenden muss. Im Gegensatz dazu sind mehrere Methoden in einer Klasse, die sich nur im Rückgabetyp oder dem Zugriffsmodifizierer unterscheiden, nicht erlaubt. Wenn man den Begriff *Signatur* exakt verwendet, dann kann man sagen: Verschiedene Methoden innerhalb derselben Klasse müssen sich in ihrer Signatur unterscheiden.

2.3 Variablen

2.3.1 Primitive Typen

Variablen: Deklaration, Initialisierung, Zugriff Eine *Variable* dient der Aufnahme von Daten. Die Daten, die eine Variable repräsentiert, können verändert werden. Java ist eine stark typisierte Sprache. Bevor man eine Variable verwenden kann, muss man diese *deklarieren*. Dabei wird festgelegt, Daten welchen Typs die Variable speichern kann. Beispiel:

Listing 2.10: Programm mit Variablen für primitive Typen

```
1  public class Beispiel{
2
3      public static void main(String[] args) {
4          int a;
5          a = 1;
6          int b = 1;
7          int c = 3;
8          int d = 42;
9
10         System.out.println(a);
11         System.out.println(b);
12         System.out.println(c);
13         System.out.println(d);
14         d = 2;
15         System.out.println(d);
16         d = d + 1;
17         System.out.println(d);
18
19         System.out.println(a == b);
20         System.out.println(a == c);
21         System.out.println(c == d);
22         System.out.println(a != c);
23     }
24 }
```

In Zeile 4 wird die Variable *a* deklariert. Ihr Typ wird dabei auf *int*, also ganze Zahlen, festgelegt. In Zeile 5 wird diese Variable *initialisiert*, d.h., sie erhält erstmals einen Wert. Würde man versuchen, mit einer Variable vor ihrer Initialisierung zu arbeiten, dann würde man einen Fehler erhalten. Das Zeichen= ist der Zuweisungsoperator. Er weist der Variablen links den Wert des Ausdrucks rechts zu. In der Zeile 6 wird die Variable *b* deklariert und sofort initialisert, was sehr häufig so gemacht wird. In den Zeilen 7 und 8 wird ebenso mit den Variablen *c* und *d* verfahren.

In den Zeilen 10 bis 13 werden die Werte der Variablen ausgegeben. Man erhält, jeweils auf einer Zeile, die Werte 1, 1, 2, und 42. In der Zeile 14 wird der Variablen d der neue Wert *2* zugewiesen, der in dann in der nächsten Zeile ausgegeben wird. In der Zeile 16 wird der Variablen *d* abermals ein neuer Wert zugewiesen. Der Ausdruck rechts des Zuweisungsoperator wird zu 3 ausgewertet, dieser Wert wird dann der Variablen d zugewiesen. Man sieht hier gut, dass der Zuweisungsoperator kein Gleichheitszeichen ist: Der Ausdruck $d = d + 1$ macht in der Mathematik, in der man = als Gleichheitszeichen auffassen würde, wenig Sinn, denn diese Gleichung hätte keine Lösung.

Test auf Gleichheit und Ungleichheit Möchte man Variablen oder andere Ausdrücke auch Gleichheit testen, dann muss das Symbol == zum Einsatz kommen, das in den Zeilen 19 bis 21 benutzt wird. Die Ausgaben dieser Zeilen lauten *true*, *false* und *true*. Die Werte *true* und *false* sind die einzigen Werte des Typs *boolean*. Das Zeichen *!=* in Zeile 22 ist der Test auf Ungleichheit. Die Zeile liefert die Ausgabe *true*, da *a* ungleich *c* ist.

Alle primitiven Typen in Java In Java gibt es acht *primitive Typen*. Das sind Typen, die von Anfang an in der Programmiersprache angelegt sind.

Die primitiven Typen in Java sind:

- Der Typ *int* fpr *integer*, deutsch *Ganzzahl*. Er steht für eine ganze Zahl zwischen −2147483648 und +2147483647.
- Der Typ *boolean* mit den Werten *true* und *false*.
- Der Typ *float* für *floating point number*, deutsch *Gleitkommazahl*, eine Zahl mit Komma.
- Der Typ *double*. Er steht für wie *float* für eine Zahl mit Komma, deckt aber einen größeren Zahlenbereich mit einer feineren Unterteilung ab. Eine Variable des Typs *double* benötigt doppelt soviel Speicher (8 Byte statt 4 Byte) wie eine Variable des Typs *float*, daher der Name.
- Der Typ *char* von englisch *character*, *Zeichen*. Eine Variable dieses Typs kann ein einzelnes Zeichen speichern.
- Die Typen *long*, *short* und *byte* stehen wie *int* für ganze Zahlen. Dabei umfasst *long* einen größeren Zahlenbeeich als *int* für die Fälle, in denen *int* nicht ausreicht. *short* und *byte* dagegen haben nur kleine Wertebereiche, benötigen dafür aber nicht viel Speicher. Eine Variable des Typs *byte* belegt nur 1 Byte, kann aber nur Werte zwischen −128 und +127 darstellen.

2.10: Aufgabe: Computer und Waschmaschinen

Wenn Sie eine Java-Programm für Ihren Computer schreiben, dann ist es üblich, ganze Zahlen mit dem Typ *int* darzustellen. Er umfasst einen Zahlenbereich, der für die meisten Anwendungen groß genug ist und der Speicher Ihres Computers kommt mit dem Speicherbedarf zurecht.
Eine Waschmaschine hat nur wenig Speicher eingebaut. Diskutieren Sie, welchen Typ Sie für die Speicherung der Temperatur des Waschgangs verwenden würden.

2.3.2 Objekttypen

In objektorientierter Programmierung kann man weitere Typen erschaffen, diese Typen sind dann nicht primitiv. Der Typ *String* ist ein Beispiel für einen nicht primitiven Typen, man spricht auch von einem *Objekttypen*. Der Typ String kann nur verwendet werden, weil in Ihrer Java-Installation eine Klasse *String* vorhanden ist. Wäre das nicht der Fall, müssten Sie die Klasse *String* und viele andere Klassen selbst programmieren. Moderne Programmierung funktioniert nur, weil viele solche Bausteine in einer *Bibliothek* bereits vorliegen und nicht erst selbst programmiert werden müssen.

Ein ähnliches Programm wie das obige, diesmal aber mit Variablen des Typs String, also einem Objekttypen, kommt im folgenden Beispiel:

Listing 2.11: Programm mit Variablen für Objekttypen

```
1  public class Beispiel{
2
3      public static void main(String[] args) {
4          String a;
5          a = "Hallo";
6          String b = "Hallo";
7          String c = "Welt!";
8          String d = "Studienkolleg Darmstadt";
9
10         System.out.println(a);
11         System.out.println(b);
12         System.out.println(c);
13         System.out.println(d);
14         d = "Technische Uni DA";
15         System.out.println(d);
16         d = d + 1;
17         System.out.println(d);
18
19         System.out.println(a.equals(b));
20         System.out.println(a.equals(c));
21         System.out.println(d.equals("Technische Uni DA1"));
22         System.out.println(!a.equals(c));
23     }
24 }
```

Vieles ist gleich, es gibt aber auch Unterschiede: Das Zeichen + bedeutet in Zusammenhang mit Strings die Konkatenation, also die Verkettung. Daher liefert Zeile 17 die Ausgabe *Technische Uni DA1*, weil in der vorigen Zeile an den String *Technische Uni DA* die *1* angehängt wird. Die Zahl *1* wird dafür automatisch in den String *1* umgewandelt.

Die Operatoren == und *!*= funktionieren bei Objekttypen nicht immer wie gewollt. Warum das so ist wird in Abschnitt 3.4, speziell hier erklärt. Man verwendet bei Objekttypen daher normalerweise die Methode *equals*. Sie liefert die erwarteten Ergebnisse. Auf Ungleichheit prüft man, indem man einen Ausdruck mit der *equals()*-Methode mit dem Ausrufungszeichen *!* vor dem Ausdruck negiert.

2.4 Methoden und Variablen im Zusammenspiel

2.4.1 String-Methoden

length() Die Methode *length()*, aufgerufen auf einem Objekt des Typs String, gibt die Länge des Strings zurück. Beispiel:

Listing 2.12: Beispiel für die Methode length()

```
1      public class Beispiel{
```

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

Das Programm gibt die Zahl *13* aus, weil das Wort *Studienkolleg* 13 Buchstaben lang ist. Wem die Zeile 4 in diesem Programm zu kompakt ist, kann Variablen verwenden:

Listing 2.13: Beispiel für die Methode length() mit Variablen

```
1   public class Beispiel{
2
3       public static void main(String[] args) {
4           String wort = "Studienkolleg";
5           int laenge = wort.length();
6           System.out.println(laenge);
7       }
8   }
```

In Zeile 4 dieser Programmversion wird eine Variable des Typs *String* mit dem Bezeichner*wort* deklariert und der String *Studienkolleg* dieser Variablen zugeordnet. In Zeile 5 wird eine Variable des Typs *int* mit dem Bezeichner *laenge* deklariert und die Rückgabe der Methode *length()*, aufgerufen auf *wort*, dieser Variablen zugewiesen. In Zeile 6 wird schließlich der Wert dieser Variablen, also *13*, ausgegeben.

substring() Im Wort *Studienkolleg* steckt das Wort *die*. In der Informatik wird gewöhnlich ab der Zahl 0 gezählt. Somit ist *S* der 0. Buchstabe des Worts *Studienkolleg*, die Buchstaben *die* sind der 3., 4. und 5. Buchstaben. Das folgende Programm holt diesen Teil des Wort heraus und gibt ihn aus:

```
1       public class Beispiel{
2
3       public static void main(String[] args) {
4           String wort = "Studienkolleg";
5           String teil = wort.substring(3, 6);
6           System.out.println(teil);
7       }
8   }
```

Maßgeblich ist die Methode *substring()*. Sie nimmt zwei Parameter des Typs *int* entgegen, nämlich die Nummer des 1. Buchstabens des Substrings und die Nummer des 1. Buchstabens nach dem Substring und gibt den gewünschten Substring zurück. Auf den ersten Blick scheint es willkürlich, dass der Start-Index inklusiv, der End- Index jedoch exklusiv ist. Auf den zweiten Blick bemerkt man aber, dass das sinnvoll ist. So sieht man beispielsweise sofort, dass der ausgewählte Substring die Länge 3 hat, weil $6 - 3 = 3$ ist.

2.6: Exkurs: Variante der Methode substring()

Möchte man aus dem String *Darmstadt* den Substring *stadt* gewinnen, dann könnte man auf den String den Methodenaufruf *substring(4, 9)* anwenden. Es funktioniert aber auch *substring(4)*. Wir nur ein Parameter angegeben, dann wird für den zweiten Parameter die Länge des Strings verwendet. Es gibt also zwei Versionen der Methode *substring()*, eine mit einem Parameter, eine andere mit zwei Parametern. In anderem Worten: Die Methode *substring()* ist *überladen*.

Blöcke und Sichtbarkeit von Variablen Ein *Block* ist eine Bereich, der durch geschweifte Klammern eingeschlossen ist. Eine Methode beginnt und endet mit geschweiften Klammern und definieren dadurch einen Block. Viele Kontrollstrukturen (siehe Kapitel 3 legen ebenfalls Blöcke fest. Eine Variable ist *sichtbar*, d.h. benutzbar, in dem *Block*, in dem sie deklariert wurde. Eine Variable, die in einer Methode deklariert wurde, dazu gehören auch die Parameter, sind nur in der Methode sichtbar. Man sagt die Variablen ist *lokal (bezüglich der Methode)*. Eine Variable, die in einer Klasse außerhalb von Methoden oder anderen Blöcken deklariert wurde und somit in der gesamten Klasse sichtbar ist, heißt *globale*

Variable. In der objektorientierten Programmierung nennt man (nicht statische) globale Variablen, die die Eigenschaften der Objekte speichern, auch *Attribute* oder *Instanzvariablen*.

2.4.2 Aufgaben

Auf der Seite https://codingbat.com/java finden Sie Aufgaben zur Erstellung von Methoden. Sie müssen bei diesen Aufgaben kein vollständiges Programm mit einer *main*-Methode erstellen. Der Ablauf ist der folgende: Sie schreiben eine Methode. Wenn Sie den Button *Go* drücken, wird Ihre Methode genommen, in ein Programm mit einer *main*-Methode eingebaut und von dieser *main*-Methode aufgerufen. Dies geschieht mehrmals mit verschiedenen Parametern. Die Rückgabe, die Ihre Methode dabei liefert, wird mit einer erwarteten Rückgabe verglichen, um Ihnen Rückneldung geben zu können, ob Ihre Methode korrekt arbeitet.

2.11: Aufgabe: Codingbat, String 1

Auf der Seite https://codingbat.com/java finden Sie Aufgaben zur Erstellung von Methoden. Lösen Sie folgende Aufgaben:

1. helloName
2. makeAbba
3. makeTags
4. makeOutWord
5. extraEnd
6. withoutEnd
7. nonStart
8. left2
9. right2
10. nTwice

2.7: Exkurs: charAt()

Wenn Sie bei den Aufgaben auf *Codingbat* zur Bearbeitung von Strings einen einzelnen Buchstaben aus dem String holen möchten, z.B. den dritten Buchstaben, dann können Sie das mit dem Methodenaufruf *substring(3, 4)* auf dem entprechendem String-Objekt tun. Eine Alternative ist der Methodenaufruf *charAt(3)*. Beachten Sie, dass die Methode *substring()* ein Objekt des Typs *String* zurück gibt, die Methode *charAt()* jedoch den primitiven Typen *char*.

integer-Division Das Zeichen für die Division in Java ist /. Wendet man die Division auf ganze Zahlen (*int*) an, dann erhält man ein Ergebniss des Typs *int*. Eventuelle Nachkommastellen werden abgeschnitten. $8/3$ liefert also 2, $-99/5$ liefert -19.

2.12: Aufgabe: noch mehr Codingbat, String 1

Die folgenden Aufgaben können Sie nun auch lösen:

1. firstHalf
2. middleTwo
3. middleThree

Die weiteren Aufgaben in Codingbat String 1 erfordern die *bedingte Anweisung* mit dem Schlüsselwort *if*, das später in Abschnitt 3.1.1 behandelt wird. Sie können versuchen, die Aufgaben zu lösen und dabei den Umgang mit *if* lernen, Sie können aber auch später auf diese Aufgaben zurück kommen, falls Sie noch mehr üben müssen.

2.5 Ein- und Ausgabe

2.5.1 Ausgabe mit einem GUI-Element

Das folgende Programm iste eine Variation des Hello-World-Programms:

Listing 2.14: Ausgabe mit einer Message-Box

```
1       import javax.swing.JOptionPane;
2
3   public class Beispiel {
4
5       public static void main(String[] args) {
6           JOptionPane.showMessageDialog(null, "Hallo Welt!");
7       }
8   }
```

Es liefert eine Box mit der Ausgabe wie in Abbildung 2.3 gezeigt. *JOptionPane* ist eine Klasse, die die Methode *showMessageDialog* zur Verfügung stellt. Die Methode wird nicht auf einem Objekt der Klasse, sondern der Klasse selbst aufgerufen. Sie ist also eine statische Methode. Die Methode nimmt zwei Parameter entgegen: Der erste Parameter ist das Mutter-Fenster, wenn die Methode innerhalb einer größeren *GUI* (von englisch *Graphical User Interface, graphische Benutzeroberfläche*). Wird das Mutter-Fenster geschlossen, das könnte z.B. das Hauptfenster einer Anwendung sein, dann muss auch dieses Fenster geschlossen werden. Um solche Funktionalutäten zu ermöglichen, muss Message-Dialog-Fenster sein Mutter-Fenster kennen. In unserer Anwendung gibt es kein Mutter-Fenster, daher geben wir dafür *null*, den Verweis auf ein nicht vorhandenes Objekt, an. Der zweite Parameter ist des Typs *String* und gibt den anzugebenden Text an.

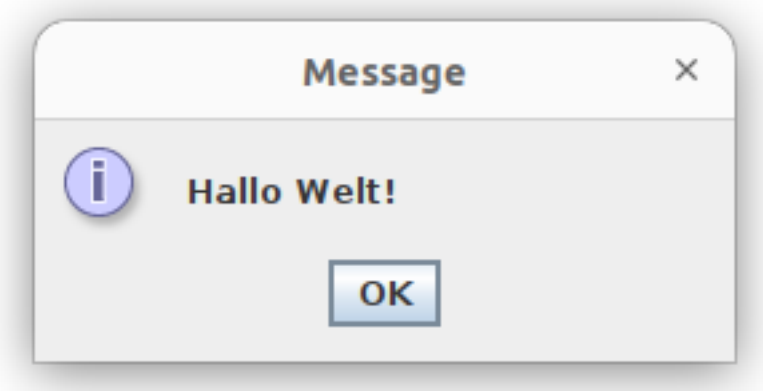


Abbildung 2.3: Ausgabe mit einer Message-Box

Bevor man die Klasse JOptionPane benutzen kann, muss sie importiert werden. Dies geschieht in der Zeile 1. Fast alle Klassen aus der Bibliothek muss man importieren. Wichtige Ausnahmen sind die Klassen *String*, die Sie bereits kennen, und *Math*, die wichtige mathematische Funktionen und Konstanten zur Verfügung stell, weil sie sehr oft gebraucht werden. Die Zeile für den Import müssen Sie sich nicht merken: Ohne diese Zeile meldet Eclipse einen Fehler. Ein Rechtsklick auf das rote Fehlersymbol an der Zeile liefert einige Vorschläge zur Lösung des Problems. Der erste Vorschlag ist normalerweise, den fehlenden Import hinzuzufügen. Wenn Sie diesen anklicken, wird die Zeile automatisch eingefügt.

2.5.2 Eingaben

Im folgenden Programm wird der Benutzer:in zwei Fragen gestellt, die diese durch eine Eingabe beantworten soll. Die Antwort wird jeweils in einer Variablen gespeichert und dann wieder ausgegeben.

Listing 2.15: Eingabe auf zwei Arten

```
1   import java.util.Scanner;
```

```
 2  import javax.swing.JOptionPane;
 3
 4  public class Eingaben {
 5
 6      public static void main(String[] args) {
 7          Scanner sc = new Scanner(System.in);
 8          System.out.println("Wie alt bist du?");
 9          int alter = sc.nextInt();
10          sc.close();
11          System.out.println(alter);
12
13          String geschwister = JOptionPane.showInputDialog(null, "Wie viele Geschwister
                hast du?");
14          JOptionPane.showMessageDialog(null, geschwister);
15      }
16  }
```

In Zeile 7 wird ein Objekt der Bibliotheksklasse *Scanner* erzeugt, nachdem in Zeile 1 der notwendige Import vorgenommen wurde. In Zeile 9 wird dann zur Eingabe aufgefordert. Die Benutzer:in gibt in die Konsole, also das Fenster, in dem auch die Standard-Ausgabe erfolgt, die Eingabe ein. In der folgenden Zeile 10 wird das Scanner-Objekt geschlossen. Diese Anweisung ist nicht unbedingt nötig, wenn sie weggelassen wird, verursacht dies aber unnötige Prozessorlast.

Eine Alternative ist die Zeile 13. Hier erfolgt die Eingabe mit der Klasse JOptionPane und deren Methode *showInputDialog()* wie in Abbildung 2.4 gezeigt.

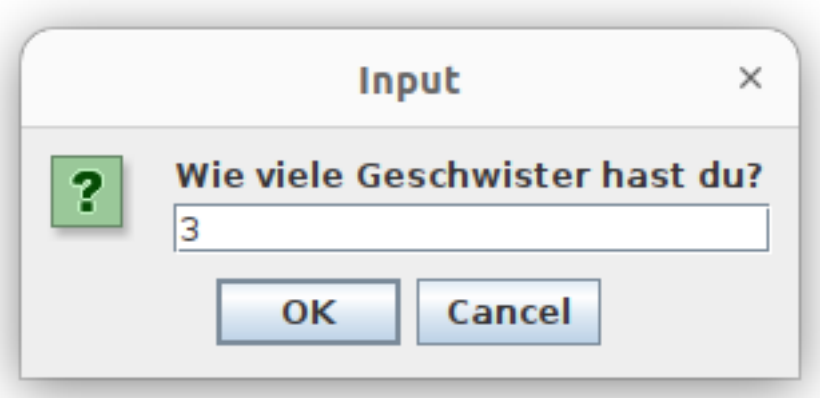


Abbildung 2.4: Eingabe mit einer Input-Dialog-Box

Ein Objekt der Scanner-Klasse kann außer ganze Zahlen mittles der Methode *nextDouble()* auch Zahlen mit Komma entgegennehmen oder mit der Methode *nextLine()* einen String. Die Methode *showInputDialog()* gibt die Eingabe grundsätzlich als String zurück. Würde man im obigen Beispiel mit der eingegebenen Zahl rechnen wollen, müsste man den String *3* zunächst in die ganze Zahl (int) *3* umwandeln. Das könnte beispielsweise mit der folgenden Zeile geschehen, die Umwandlung in eine Zahl des Typs *double* funktioniert analog.

Listing 2.16: Umwandlung eines String in eine ganze Zahl

```
1  int bruederUndSchwestern = Integer.parseInt(geschwister);
```

2.8: Exkurs: Fehler abfangen mit try/catch

Sie sollen natürlich fehlerlose Programme schreiben. Aber auch ein eigentlich fehlerloses Programm kann abstürzen. Beispiele:

- Ihr Programm kann versuchen, eine Datei auf einen Datenträger zu schreiben, aber der Datenträger ist voll.
- Ihr Programm kann versuchen, Daten über eine Netzwerkverbindung auszutauschen, aber die Netzwerkverbindung steht nicht zur Verfügung.
- Sie versuchen, mit der obigen Zeile einen String in einen int umzuwandeln, aber der String, den eine Benutzer:in eingegeben hat, enthält Buchstaben.
- Sie benutzen die Methode *nextInt()* eines Objekts der Scanner-Klasse, aber die Benutzer:in gibt keine ganze Zahl ein.

Anweisungen, die versagen können, werden in einen *try-Block* geschrieben. Auf diesen muss eine *catch*-Anweisung folgen, in der beschrieben wird, was getan werden soll, wenn die Anweisungen im try-Block nicht ausgeführt werden können. Auf diese Weise kann ein Programmabsturz verhindert werden. Beispiel:

Listing 2.17: Beispiel try/catch

```
1  public static void main(String[] args) {
2      Scanner sc = new Scanner(System.in);
3
4      System.out.println("Wie alt bist du?");
5      try {
6          int alter = sc.nextInt();
7          sc.close();
8          System.out.println(alter);
9      }
10     catch (Exception e) {
11         System.out.println("Du haettest eine Zahl eingeben sollen, du Depp!");
12         e.printStackTrace();
13     }
14 }
```

Die Anweisung in Zeile 11 ist klar, Zeile 12 gibt Auskunft über den Fehler und durch welche Methodenaufrufe er verursacht wurde, um gegebenenfalls zusätzliche Informationen für die Programierer:in oder die Benutzer:in zur Verfügung zu stellen.

3 Strukturierung

3.1 Kontrollstrukturen

3.1.1 Bedingte Anweisung und Verzweigung

Bedingte Anweisung Dividiert man die Masse in Kilogramm einer Person durch die Größe dieser Person zum Quadrat, dann erhält man den *Body-Mass-Index*, kurz *BMI* dieser Person. Der BMI hat theoretisch die Einheit kg/m^2, wird aber – die Physiklehrer:innen der Studienkollegs mögen mir verzeihen – meist einfach als einheitenlose Zahl angegeben. Mit dem BMI kann man einfache Aussagen über Über-, Normal- oder Untergewicht einer Person machen. Ob diese wirklich hilfreich sind, ist umstritten, aber ein gutes Beispiel an dieser Stelle gibt der BMI her.

Das folgende Programm fragt die Benutzer:in nach Größe und Gewicht. (Die Physiklehrer:innen mögen mir nochmal verzeihen: Physikalisch korrekt wäre *Masse*, aber in der Umgangssprache spricht man meistens von Ǵewicht, was in der Physik eher für die *Gewichtskraft* steht.) Damit wird der BMI ausgerechnet und bewertet, ob die Person übergewichtig ist oder nicht.

Listing 3.1: Ein Programm mit bedingten Anweisungen

```
1  import java.util.Scanner;
2
3  public class Person {
4
5      double groesse;
6      double gewicht;
7
8      void eingeben() {
9          Scanner scanner = new Scanner(System.in);
10         System.out.println("Bitte geben Sie Ihre Groesse in Meter ein.");
11         groesse = scanner.nextDouble();
12         System.out.println("Bitte geben Sie Ihr Gewicht in Kilogramm ein.");
13         gewicht = scanner.nextDouble();
14         scanner.close();
15     }
16
17     void berechnenUndAusgeben() {
18         double bmi = gewicht/(groesse*groesse);
19         System.out.println("Ihr BMI betraet " +bmi +".");
20         if (bmi > 25) {
21             System.out.println("Sie haben Uebergewicht.");
22         }
23         if (bmi >= 18.5 && bmi <= 25) {
24             System.out.println("Sie haben Normalgewicht.");
25         }
26         if (bmi < 18.5) {
27             System.out.println("Sie haben Untergewicht.");
28         }
29     }
30
31     public static void main(String[] args) {
```

```
32          Person p = new Person();
33          p.eingeben();
34          p.berechnenUndAusgeben();
35      }
36  }
```

Neu ist hier die *bedingte Anweisung* mit dem Schlüsselwort *if*. Nach dem *if* kommt in runden Klammern ein Ausdruck, der zu einem Booleschen Wert, also *true* oder *false* auswertet. Die Anweisungen, die in der dann folgenden geschweiften Klammer stehen, werden nur ausgeführt, wenn der Boolesche Ausdruck zu *true* ausgewertet wurde.

Neu sind auch die Operatoren >= und <= für *größer oder gleich* bzw. *kleiner oder gleich* und der Ausdruck &&, der zwei Ausdrücke, die zu einem Booleschen Ausdruck auswerten, mit dem *logischen UND* verknüpft.

3.1: Exkurs: Debugger verwenden

Ein Fehler in einem Programm nennt man manchmal *Bug*, wörtlich übersetzt heißt das *Wanze*. Der Ausdruck kommt aus der Frühzeit der Computer, als diese mit *Relais*, das sind elektromagnetische Schalter, gearbeitet haben. Die Computerspezialistin Grace Hopper reparierte einen Computer durch das Entfernen einer Wanze aus einem Relais.
Ein *Debugger* ist ein Hilfsmittel in einer IDE zum Finden von Fehlern. Wenn Sie einen Fehler in einem Programm suchen, dann kann es sinnvoll sein, das Programm Schritt für Schritt laufen zu lassen und bei jedem Schritt zu beobachten, ob die Variablen die erwarteten Werte angeben und ob das Programm die Anweisungen in der erwartete Reihenfolge abarbeitet. Eine solche Analyse können Sie in Eclipse so durchführen:

- Setzen Sie durch einen Doppelpunkt am linken Rand des Codefensters einen Stopppunkt.
- Starten Sie das Programm nicht am üblichen Play-Button, sondern am Bug-Symbol links daneben. Die Frage nach dem Umschalten in die Debug-Ansicht beantowrten Sie mit *sitsch*, also *ja*. Das Programm läuft nun bis zum von Ihnen gesetzten Stopppunkt. Im Variablen-Fenster sehen Sie die Werte der verschiedenen Variablen.
- Mit den gelben Pfeilen *Step Into*, *Step Over* und *Step Return* können Sie durch das Progeramm gehen. *Step Into* und *Step Over* gehen jeweils einen Schritt weiter. Sie unterschieden sich aber, wenn die betreffende Anweisung ein Methodenaufuf ist: *Step Over* behandelt das Abarbeiten der Methode als eine Anweisung und geht in die nächste Zeile. *Step Into* hingegen geht in die aufgerufene Methode und Sie können diese Schritt für Schritt abarbeiten. *Step Return* verlässt eine Methode und begibt sich zurück an die Stelle beim Methodenaufruf.

Vermutlich hilft diese Anleitung aleine nicht weiter. Fragen Sie mich im Unterricht oder scheuen Sie sich eines der zahlreichen YouTube-Videos zum Thema an.

Verzweigung Eine andere Version der Methode *berechnenUndAusgeben()* könnte so aussehen:

Listing 3.2: Eine Methode mit Verzweigungen

```
1  void berechnenUndAusgeben() {
2      double bmi = gewicht/(groesse*groesse);
3      System.out.println("Ihr BMI betraegt " +bmi +".");
4      if (bmi > 25) {
5          System.out.println("Sie haben Uebergewicht.");
6      }
7      else {
8          if(bmi >= 18.5) {
9              System.out.println("Sie haben Normalgewicht.");
10         }
11         else {
12             System.out.println("Sie haben Untergewicht.");
13         }
14     }
15 }
```

Hier wird die Kontrollstruktur der *Verzweigung* verwendet: In Zeile 4 findet sich wieder das Schlüsselwort *if* und die Anweisung in Zeile 5 wird nur ausgeführt, wenn die Bedingung in Zeile 4 zu *true* auswertet. Wird die Bedingung hingegen zu *false* ausgewertet, wird nicht nichts gemacht, sondern etwas anderes: In Zeile 7 beginnt ein *else-Zweig*, der mit der schließenden geschweiften Klammer in Zeile 14 endet. Der else-Zweig enthält eine weitere Verzweigung.

3.1: Aufgabe: Verzweigung oder mehr Bedingungen

Vergleichen Sie die beiden Versionen der Methode *berechnenUndAuswerten()* in Listing 3.1 und Listing 3.2. Welche ist kürzer? Welche ist einfacher zu verstehen? Welche gefällt Ihnen besser? Welche kann schneller ausgeführt werden?

3.2: Exkurs: Notwendigkeit von Blöcken

Nach einem *if* oder *else* steht meistens ein *Codeblock*, der durch geschweifte Klammern zusammen gefasst wird. Ohne die geschweiften Klammern gilt das *if* oder *else* immer nur für eine Anweisung. Allerdings sind die Blöcke im Listung 3.2 auch immer nur eine Anweisung lang. Das heißt, man könnte diese geschweiften Klammern weglassen:

Listing 3.3: Code ohne Blöcke

```
1  void berechnenUndAusgeben() {
2      double bmi = gewicht/(groesse*groesse);
3      System.out.println("Ihr BMI betraegt " +bmi +".");
4      if (bmi > 25)
5          System.out.println("Sie haben Uebergewicht.");
6      else if (bmi >= 18.5)
7          System.out.println("Sie haben Normalgewicht.");
8      else
9          System.out.println("Sie haben Untergewicht.");
10 }
```

Das ist kürzer, man kann es aber auch weniger übersichtlich finden. Gefahr besteht, wenn zusätzliche Anweisungen eingefügt werden und die dann notwendigen Klammern vergessen werden.

Algorithmik Mit zunehmender Komplexität der Programme wird es schwieriger, einfach wild zu programmieren. Man muss die Programme vorher planen. *Softwareengeneering* beschäftigt sich mit dem Planen von großen Programmen, deren Aufteilen in Komponenten und das Zusammenspiel von diesen. Die Programme im Rahmen dieses Kurses sind so klein, dass Softwareengeneering kein Thema ist.

Aber auch die Erstellung einer einzelnen Methode kann kompliziert sein. Dasselbe Problem kann auf verschiedene Arten lösen. Die beiden Versionen der Methode *berechnenUndAuswerten()* in Listing 3.1 und Listing 3.2 sind ein erstes Beispiel dafür. Man spricht von unterschiedlichen *Algorithmen* zur Lösung des Problems. Sie müssen in der Lage sein, eine Methode zu planen und dafür Algorithmen notieren zu können. Es gibt verschiedene Arten, Algorithmen zu notieren, z.B.

- *Pseudocode*. Dabei schreibt man die Methode in Schritten auf, nicht unähnlich der Programmierung. Jedoch kümmert man sich nicht um die Syntax einer spezifischen Programmiersprache und fasst größere, aber einfache Teile kompakt zusammen.
- *Programmablaufpläne*, kurz *PAP*, die im folgenden thematisiert werden.
- *Nassi-Shneiderman-Diagramme*.

Der in Abbildung 3.1 gezeigte Programmablaufplan untersucht, wie viele Lösungen eine quadratische Gleichung mit den Koeffizienten a, b und c (die vorher mit initialisiert wurden) besitzt und das Resultat ausgibt.

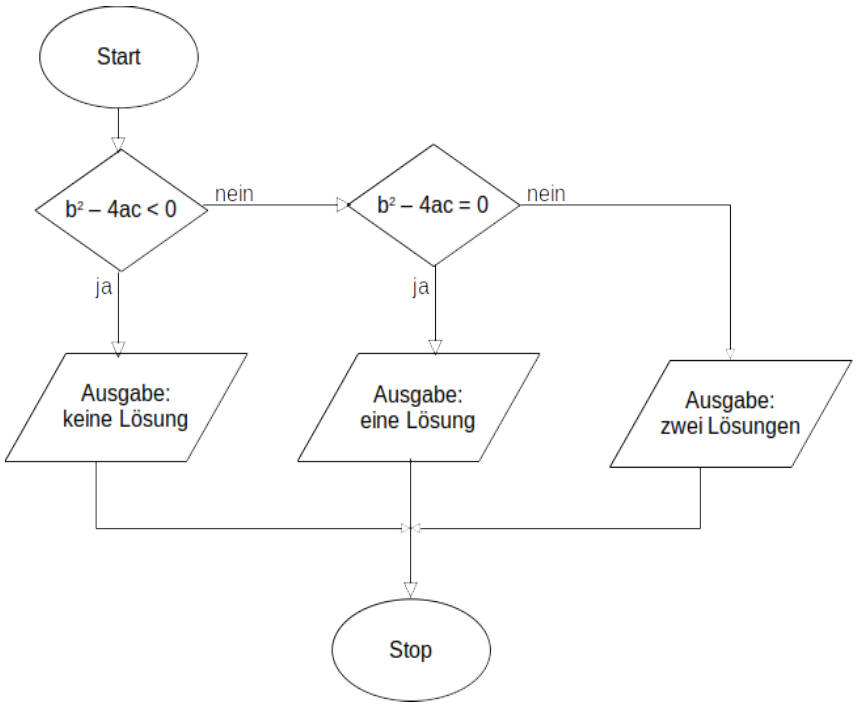


Abbildung 3.1: Programmablaufplan für einen Algorithmus mit Verzweigungen

3.2: Aufgabe: Programmablaufpläne

1. Überlegungen Sie sich, wie der PAP für eine bedingte Anweisung aussieht.
2. Zeichnen Sie die PAP für die Methode *berechnenUndAuswerten()* in Listing 3.1 und Listing 3.2.
3. Statt nur Untergewicht, Normalgewicht und Übergewicht kann man auf Grundlage des Body-Mass-Index auch starkes Untergewicht (BMI kleiner als 16), leichtes Untergewicht (BMI größer als 16, kleiner als 18,5), Normalgewicht (BMI zwischen 18,5 und 25), leichtes Übergewicht (BMI größer als 25 bis 30) und starkes Übergewicht (BMI größer als 30) unterscheiden. Erstellen Sie die PAP für zwei Programmversionen, die eine nur mit bedingten Anweisungen, die andere mit Verzweigungen, und schreiben Sie die Programme.

3.3: Aufgabe: Codingbat

1. Die Aufgaben aus Codingbat - String 1, die Sie bisher nicht lösen konnten, weil bedingte Anweisungen notwendig waren und gegebenenfalls Verzweigungen hilfreich sind, können Sie nun in Angriff nehmen.
2. Die Aufgaben aus Codingbat - Warmup 1 können jetzt von Ihnen gelöst werden. Gegebenenfalls ist neben dem *UND-Operator* &&, den Sie schon kennen gelernt haben, auch der *ODER-Operator* || hilfreich. Viele Aufgaben erfodern den Umgang mit dem Datentyp *boolean* in den Parametern oder der Rückgabe, was Sie unbedingt üben sollten.

3.4: Aufgabe: Schaltjahr

Ein Schaltjahr ist ein Jahr, in dem es den 29. Februar gibt. Es gelten folgende Regeln:

- Wenn die Jahreszahl nicht durch 4 teilbar ist, dann ist das Jahr kein Schaltjahr.
- Wenn die Jahreszahl durch 400 teilbar ist, dann ist das Jahr ein Schaltjahr.
- Wenn die Jahreszahl durch 4, aber nicht durch 100 teilbar ist, dann ist das Jahr ein Schaltjahr.
- Wenn die Jahreszahl durch 100, aber nicht durch 400 teilbar ist, dann ist das Jahr ein Schaltjahr.

Erstellen Sie einen PAP und schreiben Sie ein Programm, bei dem die Benutzer:in die Jahreszahl eingibt und das Programm ausgibt, ob es sich um eine Schaltjahr handelt oder nicht. Um das Programm zu schreiben, benötigen Sie den *Modulo-Operator*, der in Java % lautet. Er operiert auf zwei ganzen Zahlen und leifert den Divisionsrest der beiden Zahlen. Beispiel: 17%5 ergibt 2, da die Division 17/5 den Wert 3 ergibt und dabei den Divisionrest 2 übrig lässt. Eine ganze Zahl n kann mit der folgenden Zeile auf Teilbarkeit durch die ganze Zahl i geprüft werden:

Listing 3.4: Test auf Teilbarkeit

```
1    if(n % i == 0)
```

3.5: Aufgabe: Briefporto

Das Porto, also der Wert der Briefmarke, die man auf einen Brief kleben muss, hängt vom Gewicht, der Länge, der Breite und der Dicke des Briefs ab. Die genauen Regeln finden Sie hier. Schreiben Sie ein Programm, bei dem die Benutzer:in die Daten des Briefs eingeben kann und dann erfährt, welches Porto nötig ist.

Weitere Aufgaben

3.3: Exkurs: switch-case-Anweisung

Wenn man auf viele verschiedene Werte des Typs *int* unterschiedlich reagieren möchte, kann die *switch-case-Anweisung* gegebenfalls besser geeignet sein als Konstruktionen mit bedingten Anweisungen oder Verzweigungen. Die folgende Methode gibt ein Beispiel für deren Anwendung.

```
1  static void noteInWorten(int note) {
2      switch(note) {
3      case 1: System.out.println("sehr gut");
4      break;
5      case 2: System.out.println("gut");
6      break;
7      case 3: System.out.println("befreidigend");
8      break;
9      case 4: System.out.println("ausreichend");
10     break;
11     case 5: System.out.println("mangelhaft");
12     break;
13     case 6: System.out.println("ungenuegend");
14     break;
15     default: System.out.println("keine Note im deutschen Notensystem");
16     }
17 }
```

3.6: Aufgabe: Quadratische Gleichung

Schreiben Sie ein Programm, bei dem die Benutzer:in die Werte der Koeffizienten a, b und c einer quadratischen Gleichung $ax^2 + bx + c = 0$ eingeben kann und dann erfährt, ob und gegebenenfalls welche Lösungen es gibt.

3.1.2 Schleifen

Kopfgesteuerte while-Schleife Was, wenn die Benutzer:in eine offenkundig falsche Eingabe tätigt, z.B. für eine Größe einen negativen Wert eingibt? Ein gutes Programm sollte erkennen, dass diese Eingabe nicht sinnvoll sein kann und nachfragen. Die folgende Methode unternimmt einen Versuch, das Problem anzugehen.

Listing 3.5: Ein erster Verusch, falsche Eingaben zu verhindern

```
1 static void groesseEingeben() {
2     System.out.println("Bitte geben Sie Ihre Groesse in Meter ein!");
3     double groesse = scanner.nextDouble();
4     if (groesse < 0) {
5         System.out.println("Falsche Eingabe. Bitte nochmal!");
6         groesse = scanner.nextDouble();
7     }
8     return groesse;
9 }
```

Diese Lösung ist aber nicht besonders gut, denn wenn die Benutzer:in abermals eine falsche Eingabe tätigt, geht diese durch. Mit mehreren bedingten Anweisungen könnte man mehrere falsche Eingaben abfangen, aber viele wären gut? Abhilfe schafft eine neues Konstrukt, die *while-Scleife*. Bei allen Schleifen, englisch *loop*, wird etwas wiederholt. In unserem Beispiel sollte die Eingabe so lange wiederholt werden, bis sie vernünftig ist, wie das das folgende Beispiel leistet.

Listing 3.6: Eine gute Lösung, um eine falsche Eingabe zu verhindern

```
1 static double groesseEingeben() {
2     System.out.println("Bitte geben Sie Ihre Groesse in Meter ein!");
3     double groesse = scanner.nextDouble();
4     while (groesse < 0) {
5         System.out.println("Falsche Eingabe. Bitte nochmal!");
6     }
7     groesse = scanner.nextDouble();
8     return groesse;
9 }
```

3.7: Aufgabe: PAP der while-Schleife

Zeichnen Sie den PAP für die Methode in Listing 3.6.

3.4: Exkurs: Ein erster Blick auf die Rekursion

Man kann eine Methode wie die in Listing 3.6 auch ohne Schleife realisieren, wenn man *Rekursion*, die in Kapitel 5 ausführlich behandelt wird, nutzt. Eine rekursive Methode ist eine Methode, die sich selbst aufruft. Die folgende Methode übernimmt dieselbe Aufgabe wie die Methode in Listing 3.6, aber ohne while-Schleife, statt dessen mit einer Rekursion.

```
1 static double groesseEingebenRekursiveVersion() {
2     System.out.println("Bitte geben Sie Ihre Groesse in Meter ein!");
3     double groesse = scanner.nextDouble();
4     if (groesse >= 0) return groesse;
5     else return groesseEingebenRekursiveVersion();
6 }
```

Fußgesteuerte while-Schleife Die while-Schleife wie in Listing 3.6 ist eine *kopfgesteuerte* while-Schleife, weil die Steuerung der Schleife in Zeile 4 über dem *Schleifenrumpt* in den Zeilen 5 bis 9 steht. Eine ähnliche Methode mit einer *fußgesteuerten* Schleife finden Sie in Listing 3.7.

Listing 3.7: Fußgesteuerte Schleife

```
1 static double groesseEingebenFussgesteuert() {
2     double groesse;
3     do {
4         System.out.println("Bitte geben Sie Ihre Groesse in Meter ein!");
5         groesse = scanner.nextDouble();
6     }
7     while (groesse < 0);
8     return groesse;
9 }
```

Der Rumpf einer fußgesteuerten Schleife wird mindestens einmal durchlaufen, während der Rumpf einer kopfgesteuerten Schleife vielleicht auch niemals durchlaufen wird. Einmal muss die Eingabe getätigt werden, das spricht in diesem Anwendungsfall für die fußgesteuerte Schleife. Allerdings vergibt man sich die Möglichkeit, verschiedene Texte für die erste Eingabeaufforderung und die Wiederholung nach einer falschen Eingabe zu verwenden.

3.8: Aufgabe: Sichtbarkeit lokaler Variablen

Listing 3.7 könnte man scheinbar noch kürzer machen, wenn man die die Zeile 2 weg lässt und die Deklaration der Variablen *groesse* in Zeile 5 zusammen mit der Wertzuweisung vornimmt. Erklären Sie, warum das nicht geht.

3.5: Exkurs: Zufallszahlen

In der folgenden Aufgabe benötigen Sie Zufallszahlen. Ein Computer kann sich nur schwer Zufallszahlen „ausdenken“, denn der Ablauf eines Programms ist weitgehend determiniert, das heißt, es liefert bei denselben Eingaben immer dieselbe Ausgabe.
Die Zufallszahlen, die ein Objekt der Klasse *Random* produziert, sind *Pseudozufallszahlen*. Sie basieren auf einer Folge von Zahlen, wobei diese Folge von einer bei der Erzeugung des Random-Objekts gewählten Zahl, genannt *Seed* (englisch für *Samen*), abhängt. Der Seed wird, wenn er benötigt wird, aus dem Datum und der Uhrzeit berechnet. Um vorhersagen zu können, welche Pseudozufallszahlen geliefert werden, müsste man außer der genauen mathematischen Definition der Folge auch auf die Sekunde genau wissen, wann das Objekt erzeugt wird. Da dies schwierig zu erreichen ist, sind die Pseudozufallszahlen für die meisten Anwendungen, z.B. in Spielen, ausreichend. Echte Zufallszahlen müssen durch spezielle Hardware, die ein physikalisches Experiment durchführt, das vermessen wird, erzeugt werden.

Die folgende Aufgabe beinhaltet einen *mehrzeiligen Kommentar*. Statt für jede Zeile Kommentar // zu schreiben, kann man einen Kommentar mit dem Zeichen /* beginnen und dem Zeichen */ beenden

3.9: Aufgabe: Ratespiel

Programmieren Sie ein Ratespiel: Der Computer generiert eine Zufallszahl zwischen 0 und 100, die der Benutzer erraten muss. Nach jedem Versuch sagt das Programm der Benutzer:in, ob die geratene Zahl zu klein oder zu groß war. Am Ende, wenn die Zahl erraten wurde, graturliert das Programm zum Erfolg. Im folgenden Listing 3.8 sehen Sie einen Startpunkt für diese Aufgabe. Die Methode *zahlAusdenken()* ist fertig, die Methode *zahlErrraten()* müssen Sie implementieren. In der Methode *zahlAusdenken()* wird zunächst ein Objekt der Bibliotheksklasse *Random* erzeugt. Auf diesem wird dann die Methode *nextInt()* aufgerufen, die eine Zahl zwischen 0 und der als Pararmeter übergebenen Obergrenze zurück gibt. 0 ist inklusiv, die Obergrenze exklusiv.

Listing 3.8: Ratespiel

```
1   import java.util.Random;
2
3  public class ZahlenRaten {
4
5      public static void main(String[] args) {
6          int zahl = zahlAusdenken();
7          zahlErraten(zahl);
8      }
9
10     private static int zahlAusdenken() {
11         Random zufall = new Random();
12         int zahl = zufall.nextInt(101);
13         return zahl;
14     }
15
16     /**
17      * In dieser Methode soll die Benutzer:in die Zahl,
18      * die sich der Computer "ausgedacht" hat erraten.
19      * Benutzen Sie ein while-Schleife, die so lange
20      * laeuft, bis die Zahl korrekt erraten wurde.
21      */
22     private static void zahlErraten(int zahl) {
23         // TODO Auto-generated method stub
24     }
25 }
```

Die for-Schleife Das folgende Programm zählt von 1 bis 10:

Listing 3.9: Zählen mit einer while-Schleife

```
1  public class Schleife {
2
3      public static void main(String[] args) {
4          zaehlen(10);
5      }
6
7      private static void zaehlen(int ende) {
8          int i = 1;
9          while (i <= ende) {
```

```
10              System.out.println(i);
11              i++;
12          }
13      }
14  }
```

Ungünstig an diesem Programm ist, dass die Steuerung der Wiederholungen auf drei Zeilen verteilt ist: In Zeile 8 wird die Zählvariable i deklariert und initialisiert, in Zeile 9 wird kontrolliert, ob es (weitere) Ausführungen des Schleifenrumpfs gibt und in Zeile 11 wird die Zählvariable *inkrementiert*, d.h. um *1* erhöht. Die Zeile $i = i + 1$ würde auch funktionieren. In großen Programmen können die drei Stellen, mit denen der Ablauf der Schleife kontrolliert wird, sehr weit ausienander liegen, so dass man leicht die Übersicht verlieren kann.

Besser ist daher die folgende Version der Methode *zaehlen()*, die statt der bekannten while-Schleife eine *for-Schleife* benutzt. Die drei Angaben zur Steuerung der Schleife stehen nun alle kompakt in Zeile 16.

Listing 3.10: Zählen mit einer for-Schleife

```
1  private static void zaehlen2(int ende) {
2      for(int i = 1; i <= ende; i++) {
3          System.out.println(i);
4          }
5  }
```

Alles, was man mit for-Schleifen programmieren kann, kann man auch mit while-Schleifen programmieren und umgekehrt. Um aber übersichtlichen Code zu bekommen gilt: Eine for-Schleife ist besser, wenn zu Beginn der Schleife bereits bekannt ist, wie viele Wiederholungen nötig sind. Wenn das nicht bekannt ist, ist die while-Schliefe besser. Beispiel: Der Benutzer soll eine Länge in Metern eingeben. Sie möchten mit einer Schleife bei Eingabe von negativen Zahlen die Eingabe wiederholen.

3.1.3 Aufgaben

3.10: Aufgabe: for-Schleifen

1. Implementieren Sie eine Methode, die die folgende Ausgabe erzeugt:
   1 2 3 4 5 6 7 8 9 10
   Das Programm soll nur eine Ausgabeanweisung beinhalten, nämlich
   System.out.print(i + "\t"); Dabei fügt +"\t" einen Tabulator ein.

2. Implementieren Sie eine Methode, die die folgende Ausgabe erzeugt:
   7 14 21 28 35 42 49 56 63 70
   Das Programm nur eine Ausgabeanweisung enthalten, die pro Aufruf eine Zahl ausgibt.

3.11: Aufgabe: Aufgaben aus der Zahlentheorie

1. Implementieren Sie eine Methode *teilerBerechnen()*, die einen Parameter des Typs *int* entgegen nimmt und alle Teiler dieser Zahl ausgibt.
2. Implementieren Sie eine Methode *gemeinsameTeilerBerechnen()*, die zwei Parameter des Typs *int* entgegen nimmt und alle gemeinsamen Teiler dieser Zahlen ausgibt.
3. Implementieren Sie eine Methode *ggT()*, die zwei Parameter des Typs *int* entgegen nimmt und den größten gemeinsamen Teiler der beiden Zahlen zurück gibt. Geben Sie den PAP des von Ihnen benutzen Algorithmus an.
4. Implementieren Sie eine Methode *kgV()*, die zwei Parameter des Typs *int* entgegen nimmt und das kleinste gemeinsame Vielfache der beiden Zahlen zurück gibt. Geben Sie den PAP des von Ihnen benutzen Algorithmus an.
5. Implementieren Sie eine Methode *quersummeBerechnen()*, die eine Zahl des Typs *int* entgegen nimmt und die Quersumme dieser Zahl zurück gibt. Falls Sie dem recht komplizierten Wikipedia-Artikel nicht folgen können, hier ein Beispiel für die Quersumme: Die Quersumme von 365 ist 14, da $3 + 6 + 5 = 14$ ist.

3.12: Aufgabe: Codingbat

1. Die Aufgaben String 2 erfordern den Einsatz einer Schleife. Diese Aufgaben können Sie gut lösen.
2. Die Aufgaben Warmup 2 erfordern ebenfalls eine Schleife und sind etwas abwechlungsreicher als die die Aufgaben unter *String 2*. Allerdings erfordern die Aufgaben *arrayCount9*, *arrayFront9*, *array123*, *array667*, *noTriples* und *has271* auch den Einsatz von Arrays, die wir im nächsten Kapitel 3.2 behandeln. Diese Aufgaben sollten Sie zurück stellen.

3.13: Aufgabe: Verschachtelte for-Schleifen

Implementieren Sie eine Methode mit einer Schleife, die folgende Ausgabe erzeugt:
xxxxxxxxxx
xxxxxxxxxx
xxxxxxxxxx
xxxxxxxxxx
xxxxxxxxxx
xxxxxxxxxx
In Ihrem Programm darf nur einmal *System.out.print("X");* vorkommen, außerdem einmal *System.out.println()* zur Erzeugung der Zeilenumbrüche. Sie benötigen zwei Schleifen, die ineinander liegen, man sagt auch *verschachtelt* sind.

3.14: Aufgabe: Verschachtelte Schleifen in Codingbat

In den Aufgaben String 3 können Sie verschachtelte Schleifen vertiefen.

3.15: Aufgabe: Rechentrainer

, Schreiben Sie einen Rechentrainer für 6-jährige Kinder:

1. Implementieren Sie eine Methode, die zwei Zufallszahlen erstellt, und damit eine Additionsaufgabe der Art $3 + 14 =$? stellt. Achten Sie darauf, dass die Summe der Zahlen maximal 20 ergibt, aber die Summanden dennoch größer als 10 sein können. Die Benutzer:in soll dann die Lösung eingeben. Ist diese falsch, wird die Aufgabe erneut gestellt, ist sie richtig, gibt es ein Lob.
2. Erweitern Sie das Programm so, dass nicht nur eine Aufgabe, sondern zehn Aufgaben gelöst werden müssen. Sie können dafür die Methode aus Teil 1 mit einer Schleife (*while* oder *for*?) ergänzen, oder aber eine weitere Methode implementieren, die die Methode aus Teil 1 zehn Mal auruft.

3.16: Aufgabe: Kleines Einmaleins

Implementieren Sie eine Methode mit nur einer Ausgabeanweisung, die das kleine Einmaleins ausgibt, also
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
...
1 * 10 = 10
2 * 1 = 2
2 * 2 = 4
...
10 * 10 = 100

3.17: Aufgabe: Weitere verschachtelte for-Schleifen

Implementieren Sie je eine Methode (also insgesamt vier Methoden), die die folgenden Ausgaben erzeugen:

```
     X        XXXXXX            X            X X X X X X
    XX         XXXXX          X X             X X X X X
   XXX  und     XXXX  und    X X X     und     X X X X
  XXXX           XXX        X X X X            X X X
 XXXXX            XX       X X X X X           X X
XXXXXX             X      X X X X X X           X
```

3.6: Exkurs: Java-Programm ohne Eclipse

Bisher erstellen Sie Ihre Programme nicht nur in Eclipse, sondern lassen die Programme auch nur in Eclipse laufen. Wenn Sie ein nützliches Programm weitergeben, vielleicht gar verkaufen möchten oder vielleicht auch nur Ihren Eltern zeigen möchten, was Sie gelernt haben, dann ist da ein Problem: Sie möchten von Ihren Kunden nicht verlangen, dass sie zunächst Eclipse installieren. Außerdem möchten Sie im Allgemeinen nicht Ihren Quellcode, sondern nur das lauffähige Programm weitergeben.
Um aus Eclipse ein Standalone-Programm zu erzeugen, gehen Sie folgendermaßen vor:

- Wählen Sie *File -> Export -> Java -> Runnable JAR file* und betätigen Sie dann die Schaltfläche *next*.
- Im dann erscheinenden Fenster müssen Sie die *Lauch configuration* und die *Export destination* angeben. Die *Launch configuration* gibt die Klasse an, die die *main()*-Methode enthält, die den Programmstart ausführen soll. Die *Export destination* gibt an, wo und unter welchem Namen auf Ihrem Computer das lauffähige Programm gespeichert werden soll. Betätigen Sie nach dem Eintragen dieser Werte die Schaltfläche *Finish*.

Sie erhalten dann am zuvor festgelegten Ort eine Datei mit der Erweiterung *.jar*. Diese Datei können Sie kopieren und weitergeben. Sie sollte auf allen Computern, die eine (genügend neue) Java Virtual Machine, JVM, installiert haben, laufen. Die JVM ist auf vielen Rechnern standardmäßig installiert. ist das nicht der Fall, dann muss sie installiert werden. Downloads finden Sie auf https://www.java.com/de/, Linux-Nutzern steht auch https://openjdk.org/ zur Verfügung.
Wenn Ihr Programm keine Konsole nutzt, also insbesondere keine Ausgaben mit *System.out.println()* oder *System.out.print()* und keine Ausgaben mit einem Scanner-Objekt, sondern auf GUI-Elementen baisert, dann können Sie das Programm mit einem Doppelklick starten. Benötigt Ihr Programm eine Konsole, dann muss es aus einer Konnsole gestartet werden. Die übliche Windows-Konsole, das CMD-Fenster, erhält man durch gleichzeitiges Drücken der *Windows*-Taste und der Taste *R*. Unter Linux drücken Sie gleichzeitig die Tasten *STRG* (englisch *CTRL*), *ALT* und *T*. In der Konsole gehen Sie in das Verzeichnis, in dem die eben angefertigte .jar-Datei steht (benutzen Sie dafür den Befehl *cd* (das steht für englisch *change directory*) und geben Sie dann den Befehl *java -jar MeinProgramm.jar* ein. *MeinProgramm* ersetzten Sie dabei natürlich mit dem von Ihnen zuvor gewählten Namen Ihrer Datei.

3.2 Datenstrukturen

3.2.1 Arrays

Arrays dienen der Zusammenfassung gleichartiger Variablen. Bei der Deklaration einer Array-Variablen kommt hinter dem Typ eine öffnende und eine schließende eckige Klammer, um zu kennzeichnen, dass es sich nicht um eine einzelne Variable handelt, sondern um eine Array-Variable. Im unten stehenden Listing 3.11 wird in Zeile 2 eine Array-Variable vom Typ *String* mit dem Namen *enten* deklariert.

Listing 3.11: Beispiel mit Array

```
1  public static void main(String[] args) {
2          String[] enten = {"Donald", "Daisy", "Dagobert"};
3          for(int i = 0; i < 3; i++) {
4              System.out.println(enten[i]);
5          }
6      }
```

Die Variable *enten* verweist auf ein, ebenfalls in Zeile 2, neu erzeugtes String-Array der Größe drei. Statt der Zeile 2 könnte man auch die folgenden Zeilen schreiben:

Listing 3.12: Erzeugung und Befüllung eines Arrays

```
1      String[] enten = new String[3];
2      enten[0] = "Donald";
3      enten[1] = "Daisy";
4      enten[2] = "Dagobert";
```

Bei dieser Version sieht man, wie zuerst ein Array der Größe drei erzeugt wird. Es enthält unmittelbar nach der Erzeugung die Werte *null*, also Verweise auf das nicht vorhandene Objekt. In den folgenden Zeilen werden dann die Strings "Donald", "Daisy"bzw. "Dagobert"den jeweiligen Array-Elementen zugeordnet. Man sieht, wie die eckigen Klammern eingesetzt werden, um das jeweilige Element innerhalb des Arrays anzusprechen, genauso, wie das in Zeile 4 in Listing 3.11 mit der Nummer *i* geschieht.

Man muss sich dieses Array vorstellen als ein Speicherplatz für drei Variablen des Typs *String*. Abbildung 3.2 verdeutlicht die Situation. Die Plätze eines Arrays werden immer mit der Nummer 0 beginnend nummeriert, so dass ein Array der Größe drei Plätze mit den Nummern 0, 1 und 2 besitzt.

In der Schleife in den Zeilen 3 bis 6 in Listing 3.11 werden die Inhalte der einzelnen Array-Elemente ausgegeben.

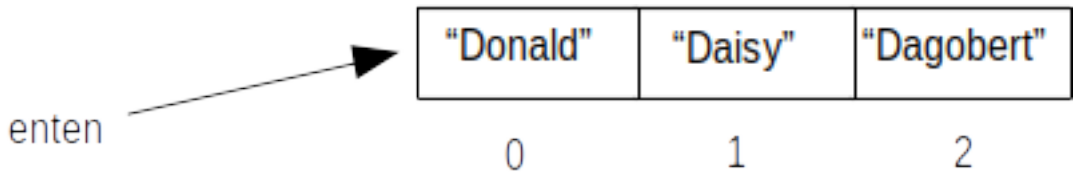


Abbildung 3.2: Ein Array

Vorteile von Arrays Ein ähnliches Programm kann man auch mit drei einzelnen String-Variablen, die man beispielsweise *ente1, ente2, ente3* nennen könnte, realisieren. Welche Vorteile bieten Arrays?

- In diesem Programm ist der wichtigste Vorteil, dass man durch die Variablen mit einer Schleife durchgehen kann. Die Zeilen 3 bis 6 in Listing 3.11 funktionieren nicht mit einzelnen Variablen *ente1, ente2, ente3* nicht, da hier die Ziffern 1, 2, und 3 nur als Zeichen, nicht als Nummern aufgefasst werden.
- Ein weiterer Vorteil ist die Kürze der Zeile 2 in Listing 3.11. Einzelne Variablen würden immer eine umständlichere Fassung wie in Listing 3.12 erfordern. Dieser Vorteil kann bei großen Arrays wichtig werden: oft hat man Hunderte oder Tausende gleichartiger Variablen, die in Schleifen verarbeitet werden. Die Programmierer:in will nicht Tausende von Variablen einzeln erzeugen.
- Ein Array betont die Zusammengehörigkeit der Variablen. Stärker als im Beispiel mit den Enten wiegt dieser Vorteil, wenn man beispielsweise die drei Koordinaten eines Punkt im Raum zu einem Array zusammenfasst oder die Komponenten eines Vektors.
- Ein ganz wichtiger Vorteil eines Arrays ist, dass seine Größe nicht notwendig zur Programmierzeit festgelegt werden muss, sondern auch erst zur Laufzeit des Programms erfolgen kann. Das Beispiel in Listing 3.13 verdeutlicht dies.

Listing 3.13: Festlegung der Array-Größe zur Laufzeit

```
1  public static void main(String[] args) {
2      Scanner sc = new Scanner(System.in);
3      String[] enten;
4      System.out.println("Wie viele Enten moechtest du eingeben?");
5      int n = sc.nextInt();
6      enten = new String[n];
7
8      for(int i = 0; i < enten.length; i++) {
9          enten[i] = sc.next();
10     }
11     sc.close();
12
13     for(int i = 0; i < enten.length; i++) {
14         System.out.println(enten[i]);
15     }
16 }
```

In Zeile 3 wird eine Variable des Typs String-Array deklariert, aber nicht initialisiert. Erst in der Zeile 6 wird dieser Variablen ein neu erzeugtes Array zugeordnet, nachdem zuvor der Benutzer gefragt wurde, wie groß das Array sein soll. Da man zur Programmierzeit die Größe des Arrays nicht kennt, kann man diese Größe in den Schleifen nicht mit einer konkreten Zahl festlegen. Man könnte hier die Variable n verwenden. Oft wurde aber das Array in einer anderen Methode erzeugt und eine solche

Variable existiert nicht im Sichtbarkeitsbereich der Schleife. Daher ist es am besten, man verwendet das Attribut *length* des Array-Objekts wie in Listing refArray3.

Wichtige Methoden in der Klasse *Arrays* In der Bibliotheksklasse *java.util.Arrays*, die man mit *import java.util.Arrays;* einbinden kann, sind nützliche Methoden, um mit Arrays zu arbeiten. Hier sind einige Beispiele. Der Bezeichner *enten* steht hier stellvertretend für den Bezeichner eines Arrays. In den Beispielen wird angenommen, dass es sich um ein Arrays von Strings handelt, die Methode funktionieren aber auch für andere Arrays.

- *Arrays.toString(enten)* überführt das Array in einen String und gibt diesen zurück.,Z.B. für eine Ausgabe mit *System.out.println(Arrays.toString(enten))*.
- *Arrays.fill(enten, "Daisy")* setzt alle Array-Elemente auf den Wert *Daisy*.
- *Arrays.sort(enten)* sortiert das Array aufsteigend, sofern die Elemente geordnet werden können. Bei Array-Elementen des Typs String erfolgt die Sortierung alphabetisch.
- *Arrays.copyOfRange(enten, int from, int to)* erzeugt ein neues Array, befüllt dieses mit den Elementen des Arrays enten zwischen den Indizes from (inklusiv) und to (exklusiv) und gibt dieses zurück.

Weitere Informationen über die Bibliotheksklasse *Arrays* finden Sie hier beispielsweise hier: https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/util/Arrays.html.

3.2.2 Aufgaben

3.18: Aufgabe: Vertauschen im Array

1. Erstellen Sie eine Methode, die in einem Array zwei Elemente vertauscht. Die Methode soll zwei Parameter vom Typ *int* besitzen. Das sollen die Nummern der Array-Elemente sein, die vertauscht werden. Sie können die Methode in ein Programm einbetten ähnlich wie in Listing 3.11.
2. Vermutlich haben Sie in Teil 1 eine weitere Variable verwendet, um den Wert eines Array-Elements zwischenzuspeichern, wenn sein alter Wert überschrieben wird. Bei Zahlen, also z.B. Werten des Typs *int*, geht es auch ohne eine solche weitere Variable. Probieren Sie das. Hinweis: Verbrauchen Sie nicht zu viel Zeit für diese Aufgabe. Entweder, Sie haben die richtige Idee und können die Aufgabe in wenigen Minuten lösen, oder aber Sie finden die Lösung in Stunden nicht.

3.19: Aufgabe: Codingbat, Arrays

Auf der Seite https://codingbat.com/java finden Sie Aufgaben zu Arrays.

- In *Array 1* finden Sie Aufgaben, die ganz grundlegend den Umgang mit Arrays vermitteln. Es werden keine Schleifen benötigt. Sie sollten damit anfangen.
- In *Array 2* finden Sie Aufgaben, die den Einsatz einer Schleife erfordern. Sie sind im Schwierigkeitsgrad überschaubar und Sie sollten diese Aufagben als nächstes anpacken
- In *Array 3* finden Sie Aufgaben, die mehrere Schleifen erfordern. Sie sind ähnlich schwierig wie die Aufgaben unten.

3.20: Aufgabe: Gefangene

König Jostus I hat in seinem Gefängnis 100 Gefangene, jeder in seiner Zelle. Zu seinem Geburtstag möchte er einige freilassen. Sein königlicher Hofmathematiker hat sich dafür folgendes Verfahren ausgedacht:
Im 1. Durchgang werden alle Türen geöffnet. Im zweiten Durchgang wird jede zweite Tür wieder geschlossen. Im dritten Durchgang wird der Zustand jeder 3. Tür verändert, das heißt, wenn sie offen war, geschlossen, wenn sie geschlossen war, geöffnet. Im 4 Durchgang wird der Zustand jeder 4 Tür verändert usw. Die Gefangenen, deren Zellen nach dem 100. Durchgang offen stehen, dürfen gehen.
Schreiben Sie ein Programm, das ermittelt, welche Türen nach dem 100. Durchgang offen stehen.
Anleitung:

- Erstellen sie ein Attribut *zellen* vom Typ *boolean[]* mit der Größe 101. (Code: *boolean[] zellen = new boolean[101];*) Das Array-Element 0 wird nicht verwendet, die Array-Elemente 1 bis 100 repräsentieren die Zellentüren. *false* bedeutet geschlossen, *true* bedeutet offen.
- In einer for-Schleife werden die 100 Verfahren durchgegangen.
- In der for-Schleife für die 100 Verfahren ist eine weitere for-Schleife, um in jedem Verfahren die 100 Türen durchgehen zu können. Tipp: Das Zeichen *!*, das in Java für *NICHT* steht, kann verwendet werden, um den Zustand einer Zellentür zu verändern. Beispiel: Die Zeile *zelle[5] = !zelle[5];* verändert den Zustand von Zellentür 5.

3.21: Aufgabe: Gefangene, Varianten

An der obigen Aufgabe sollen Sie den Umgang mit Arrays üben. Hier sind Arrays aber nicht unbedingt notwendig.

1. Man kann das Problem ohne Array lösen, indem man die Lage der Schleifen zueinander vertauscht. In der äußeren Schleife wird die Nummer der Tür durchgegangen, in der inneren Schleife werden die Durchgänge sumuliert. Auf diese Weise betrachtet man immer nur eine Tür und braucht daher kein Array. Programmieren Sie diese Variante.
2. Eine extrem kurze Lösung des Problems, ebenfalls ohne Array, kann man erhalten, wenn man die Mathematik dahinter erfasst. Versuchen Sie das.

3.22: Aufgabe: Wörterbuch

Programmieren Sie ein Wörterbuch, z.B. Deutsch – Spanisch. Zuerst soll das Wörterbuch erstellt werden, d.h. die Benutzer:in gibt ein Wort und anschließend seine Übersetzung ein. Das Wort wird in einem Array von Strings gespeichert, die Übersetzung in einem weiteren Array von Strings.
Dann soll man Wörter eingeben können und das Programm gibt, vorausgesetzt das Wort existiert im Wörterbuch, dessen Übersetzung aus. Die Übersetzung kann gefunden werden, weil das Wort und seine Übersetzung die gleiche Nummer in ihrem jeweiligen Array besitzen: Ist beispielsweise *Hund* das fünfte Wort im Array *deutsch*, dann sollte im Array *spanisch* das fünfte Wort *perro* lauten.

3.23: Aufgabe: Sieb des Eratosthenes

Das *Sieb des Eratosthenes* ist ein Algorithmus zum Finden aller Primzahlen bis zu einer gegebenen Obergrenze. Mit Bleistift und Papier kann man ihn folgendermaßen durchführen:

- Schreiben Sie alle natürlichen Zahlen von 2 bis zur gewählten Obergrenze auf.
- Kreisen Sie die erste nicht durchgestrichene Zahl (beim 1. Durchgang also die Zahl *2*) ein und streichen Sie alle weiteren Vielfache dieser Zahl (beim 1. Durchgang also *4, 6, 8 ...*, denn diese sind sicher keine Primzahlen). Dieser Schritt wird so lange wiederholt, bis alle Zahlen bearbeitet, also entweder eingekreist oder gestrichen sind.
- Am Ende sind genau die Primzahlen eingekreist.

1. Führen Sie den Algorithmus schriftlich durch.
2. Erstellen Sie einen Programmablaufplan für diesen Algorithmus.
3. Schreiben Sie ein Programm, das die Primzahlen bis zu einer von der Benutzer:in gegebenen Obergrenze ausgibt. Gehen sie folgendermaßen vor: Erstellen Sie ein Array von Booleans. Die Größe des Arrays soll um eins größer sein als die Obergrenze. Die Elemente eines neu erzeugten Arrays von Booleans besitzen zu Beginn alle den Wert *false*. Fassen Sie das als *nicht gestrichen* auf. Gehen Sie in einer Schleife von *2* bis zur Obergrenze durch und streichen Sie alle (weiteren) Vielfachen der Schleifenvariablen, indem Sie das Aray-Element mit dieser Nummer auf *true* setzen. Geben Sie am Ende die Nummern derjenigen Array-Elemente aus, die den Wert *false* besitzen.

3.24: Aufgabe: Mastermind

Programmieren Sie das Spiel Mastermind, das auch unter weiteren Namen, zum Beispiel SuperHirn, bekannt ist. Das Spiel soll textbasiert, d.h. ohne Grafik, arbeiten.
Bei Mastermind denkt sich ein Spieler, der hier durch den Computer ersetzt wird, eine Liste (Array) von vier Farben aus. Die Farben stammen aus der Menge rot, orange, gelb, grün, blau, lila. Eine mögliche Liste wäre zum Beispiel (0. orange, 1. blau, 2. lila, 3. blau). Der andere Spieler, hier in der Computervariante der einzige Spieler, muss die Liste von Farben erraten. Der Ratende bekommt nach jedem Rateversuch Rückmeldung, wie gut er geraten hat: Für die Farbe an der 0. Position wird geschaut:

- Ist an der 0. Position der Lösungsliste tatsächlich diese Farbe, gibt es einen schwarzen Punkt.
- Ist die Farbe zwar nicht an der 0. Position, aber woanders in der Lösungsliste, gibt einen weißen Punkt.
- Ist die Farbe gar nicht in der Lösungsliste, gibt es keinen Punkt.

Für die weiteren Farben der Rateliste wird genauso verfahren. Der Ratende bekommt die Anzahl schwarzer und weißer Punkte mitgeteilt, bevor er das nächste Mal rät. Beispiel: Wenn die Lösungliste (0. orange, 1. blau, 2. lila, 3. blau) lautet und die Rateliste (0. blau, 1. orange, 2. lila, 3. grün), dann wird die Rückmeldung schwarz, weiß, weiß gegeben.

1. Erstellen Sie eine Javaklasse *Mastermind*. Importieren Sie die Klassen *Random* und *Scanner*, die Sie benötigen, damit sich der Computer eine zufällige Farbliste „ausdenken“ kann bzw. um die Benutzereingaben entgegennehmen zu können.
2. Erstellen Sie drei Arrays von Strings. Die erste enthält die möglichen Farben, die zweite soll die zu erratende Farbliste aufnehmen, die dritte die vom Benutzer geratene Farbliste aufnehmen.
3. Erstellen Sie eine Methode *kombiAusdenken()*, die eine zufällige Liste von vier Farben zurück gibt.
4. Erstellen Sie eine Methode ohne Parameter *raten()*, die die Benutzer:in auffordert, vier Farben einzugeben. Die Methode soll eine Liste der eingegebenen Farben zurück geben.
5. Erstellen Sie eine Methode *bewerten()*, die das Array mit den zu ratenden Farben und das Array mit den geratenen Farben als Parameter entgegen nimmt. Die Methode gibt entsprechend den Regeln die Bewertung des Rateversuchs aus.
6. Erstellen Sie eine *main()*-Methode, die, unter Verwendung der obigen Methoden, eine Runde *Mastermind* mit der Benutzer:in spielt.

3.2.3 Listen und Maps

Listen Arrays sind sehr grundlegende Konstrukte, die es in vielen Programmiersprachen gibt. Viele Programmiersprachen bieten aber auch komfortablere *Listen* an. Der Vorteil solcher Lis- ten im Vergleich zu einem Array ist, dass die Programmierer:in sich nicht um die Größe kümmern muss, die automatisch verwaltet wird. In Java gibt es mehrere Arten von Listen, z.B. die Klasse *ArrayList*. Diese Klasse kann nach dem Import *import java.util.ArrayList;* verwendet wer- den. *ArrayList<String> liste = new ArrayList<>();* erstellt eine neue Liste. In der spitzen Klammer steht der Typ, der in der Liste gespeichert werden kann. Folgende Methoden, hier an konkreten Beispielen gezeigt, von ArrayList sind wichtig:

- *add("Dagobert")* fügt ein neues Element ans Ende der Liste an.
- *set(3, "Daisy")* setzt das dritte Element der Liste auf *"Daisy"*.
- *get(2)* liefert das zweite Element der Liste zurück.
- *remove(2)* löscht das zweite Element aus der Liste. Die Indizes der nachfolgenden Listenelemente verkleinern sich dadurch um eins.
- *liste.size()* gibt die Größe der Liste zurück.

Die Methoden *set()*, *get()* und *remove()* verursachen einen Laufzeitfehler, wenn es die betreffenden Listenelemente nicht gibt, weil die Liste zu kurz ist. Eine ArrayList kann leider keine primitiven Typen aufnehmen. Die Zeile *ArrayList<int> liste = new ArrayList<>();* führt zu einem Fehler. Zu jedem primitiven Typen gibt es aber einen Bibliothekstypen, der an dieser Stelle verwendet werden kann. Die Klassen, die die primitiven Typen repräsentieren, nennt man *Wrapper-Klassen*, von englisch *to wrap*, das bedeutet *einwickeln*. Jede Wrapper-Klasse verpackt sozusagen einen primitiven Typen. Die Wrapper-Klassen heißen meistens genau wie die primitiven Typen, nur dass am Anfang ein Großbuchstaben steht. Einzige Ausnahme: Die zum primitiven Typen *int* gehörende Wrapper-Klasse heißt *Integer*. Die obige fehlerhafte Zeile müsste also korrekt so lauten: *ArrayList<Integer> liste = new ArrayList<>();*

3.25: Aufgabe: Listen

Alle Aufgaben mit Arrays können Sie auch mit Listen lösen.

Maps Ein Array oder eine Liste ordnet einer natürlichen Zahl ein Datum[1] zu. Dieses Datum kann ein Datum eines primitiven Typs, also z.B. eine andere Zahl oder eine Boolesche Variable, oder ein Objekt, z.B. ein String, sein. Die natürliche Zahl, die die Nummer des Array- oder Listenelements angibt, nennt man *Schlüssel* oder auch englisch *Key*. Eine *Map* ist eine Liste, bei der der Key aber keine Zahl, sondern ein Objekt ist. Oft ist der Key vom Typ String und die Inhalte sind ebenso vom Typ String. Dann wird also einem Wort ein anderes Wort zugeordnet, so, wie das in einem Wörterbuch der Fall ist. Daher heißt diese Datenstruktur in anderen Programmiersprachen *Dictionary*, z.B. in *Python*. Das folgende Programm demonstriert die Verwendung einer Map.

```
1  import java.util.HashMap;
2
3  public class Woerterbuch {
4
5      public static void main(String[] args) {
6          HashMap<String, String> woerter = new HashMap<>();
7          woerter.put("Hund", "perro");
8          woerter.put("Katze", "gato");
9          woerter.put("Maus", "perro");
10
11         System.out.println("Hund auf spanisch ist");
12         System.out.println(woerter.get("Hund"));
13     }
14 }
```

In Zeile 1 erfolgt der notwendige Import. In Zeile 6 wird die Map erzeugt. Dabei muss in der spitzen Klammer angegeben werden, von welchem Typ die Keys und die Werte der Maps sind, in diesem Fall beides vom Typ *String*. Die Methode *put()* in den Zeilen 7 – 9 erezugt jeweils ein Schlüssel-Wert-Paar in der Map. Mit der Methode *get()*, die in Zeile 12 verwendet wird, kann man den Wert, der dem Key *Hund* zugeordnet ist, erhalten.

3.26: Aufgabe: Maps

Die Aufgabe, ein Wörterbuch zu erstellen, siehe oben, geht natürlich viel einfacher mit einer Map als mit zwei Arrays. Realisieren Sie diese Variante des Wörterbuchs.

3.7: Exkurs: Hashing

Um mit einer Map effizient arbeiten zu können, muss der Computer dem einem Key zugeordneten Wert schnell finden können. Um diese Aufgabe zu lösen, gibt es verschiedene Möglichkeiten und daher verschiedene Typen von Maps, z.B. *HashMap* oder *TreeMap*. Beim Hashing wird der Ort, an dem ein Wert gespeichert wird, seine Speicheradresse, aus dem zu speichernden Wert berechnet. Das klingt abstrakt, das Konzept ist aber altbekannt: Wenn Sie in einem Lexikon (aus Papier, ein Buch!) das Wort *Studienkolleg* suchen, dann möchten Sie dafür nicht das ganze Lexikon durchlesen müssen. Sie wissen aber, wo das Wort stehen muss: Sie finden vielleicht die Wörter *Studienkameradin* und *Studienkommission*. Dazwischen muss *Studienkolleg* stehen. Wenn es da nicht steht, dann ist das Wort nicht in diesem Lexikon. Der Ort, an dem ein Wort stehen muss, wird auch dem Wort selbst berechnet.

[1] Das Wort *Datum* bezeichnet fast immer eine Zahlenkette zur Bezeichnung eines Tags, z.B. *09.11.1918*. Dies ist hier nicht gemeint. *Datum* ist hier der selten gebrauchte Singular des Worts *Daten*.

3.3 Objektorientierte Programmierung

3.3.1 Idee der Objektorientierung

Neue Typen In der objektorientierten Programmierung erschafft man neue Typen. In der Sprache Java sind zunächst nur die primitiven Typen, siehe Abscnitt 2.3.1 vorhanden. Wie haben aber auch schon die Typen *String*, *Scanner*, *Random*, *JOptionPane*, *ArrayList*, und *HashMap* benutzt. Diese Typen sind *Objekttypen*. Man kann Objekte dieser Typen erzeugen, weil in der Bibliothek jeweils eine Klasse *String*, *Scanner*, *Random* ... vorhanden ist, die jemand zuvor programmiert hat. Wenn wir Typen brauchen, die es in der Bibliothek nicht gibt, dann müssen wir diese selbst programmieren. Damit sind wir in der objektorientierten Programmierung.

Klassen als Baupläne für Objekte Bisher war ein Java-Klasse eine Datei, die Quellcode aufnimmt. Unsere bisherigen Programme bestanden aus einer Klasse. In der objektorientierten Programmierung ist eine Klasse ein Bauplan für Objekte. Hat man eine Klasse, dann kann man beliebig viele *Objekte*, auch *Instanzen* genannt, dieser Klasse erzeugen. Eine Klasse besteht üblicherweise aus drei Teilen:

1. Die Attribute, auch Instanzvariablen genannt. Die Attribute speichern die Eigenschaften eines Objekts.

2. Die Konstruktoren. Konstruktoren sind spezielle Methoden, die mit dem Schlüsselwort *new* aufgerufen werden, um eine neue Instanz dieser klasse zu erzeugen. Der Bezeichner eines Konstruktors ist immer gleich wie der Konstruktor der Klasse. Ein Konstruktor hat keinen Rückgabetypen wie andere Methoden, auch nicht *void*.

3. Die (weiteren) Methoden der Klasse. Sie legen fest, was eine Instanz der Klasse tun kann.

Wenn man programmiert, dann arbeitet man immer mit Daten, die durch geeignete Operationen verarbeitet werden. In der imperativen Programmierung stehen die Daten und die Operationen, die auf diesen Daten durchgeführt werden, gegebenenfalls sehr weit voneinander entfernt im Programmcode, was sehr unübersichtlich sein kann. In der objektorientierten Programmierung wird versucht, die Daten in Gestalt der Attribute und die auf ihnen durchgeführten Operationen in Gestalt der Methoden zu Klassen zusammen zu fassen, um so übersichtlicheren Code zu erhalten.

Wichtig an diesem Vorgehen ist auch, dass man mit den Klassen arbeiten kann, ohne sie im Detail zu kennen. Wir arbeiten schon lange mit den Kassen *Scanner*,*Random* und vielen weiteren, haben aber niemal in ihren Quellcode geschaut. Man sieht: Das konzept der *Bibliothek* funktioniert viel besser in der objektorientierten Programmierung mit ihren abgeschlossenen Klassen. Damit aber nicht genug: Große Software-Projekt werden in Teams von vielen Programmierer:innen erstellt. Hier ist hilfreich, dass ein großes Projekt aus verschiedenen Klassen aufgebaut ist und nicht jede Programmierer:in jede Klasse genau kennen muss.

3.3.2 Grundlagen der Objektorientierung anhand des Beispiels *Bruch*

In Java gibt es primitive Typen für ganze Zahlen und für Kommazahlen. Es gibt aber keinen Typen für Bruchzahlen, also für Objekte der Art $\frac{5}{8}$. Um mit Brüchen rechnen zu können, benötigen wir also eine Klasse *Bruch*. Diese soll nun erstellt werden.

Der erste Teil einer Klasse sind die Attribute. Wir müssen uns überlegen, welche Attribute unsere Klasse *Bruch* benötigt. Auf die notwendigen Attribute kommt man am besten, indem man eine Frage mit dem Verb *haben* stellt: Was **hat** ein Bruch? Ein Bruch **hat** einen *Zähler* und einen *Nenner*. Zähler und Nenner sind ganze Zahlen. Damit haben wir ein guten Anfang für die Klasse Bruch. Teil 1 von 3 ist geschafft!

Listing 3.14: Beginn der Klasse Bruch

```
1 public class Bruch {
2
3     //1. Attribute
```

```
4       int zaehler, nenner;
5
6       //2. Konstruktoren
7
8       //3. Methoden
9   }
```

Wenden wir uns Teil 2 zu. Möchte man einen neuen Bruch erzeugen, dann könnte man Zähler und Nenner dieses Bruchs als Parameter angeben. Ein Konstruktor für die Klasse Bruch könnte so aussehen:

Listing 3.15: Konstruktor der Klasse Bruch, erster Versuch

```
1  //2. Konstruktoren
2  public Bruch(int neuerZaehler, int neuerNenner) {
3       zaehler = neuerZaehler;
4       nenner = neuerNenner;
5  }
```

Unschön sind die Namen der Parameter. Es wäre besser, sie würden einfach *zaehler* und *nenner* heißen. Die Zeile 2 müsste dann *zaheler = zaehler;* lauten. Dabei soll *zaehler* links das Attribut mit diesem Namen, *zaehler* rechts den Parameter mit diesem Namen. Das funktioniert aber so nicht, da der Java-Compiler nicht wissen kann, was mit *zaehler* jeweils gemeint ist. Er würde beide Male annehmen, es sei der Parameter gemeint und würde eine Warnung ausgeben, da das Ersetzen eines Parameterwerts durch sich selbst sinnlos ist. Abhilfe schafft das Schlüsselwort *this*, von englisch *dies*. Dieses Schlüsselwort bezeichnet das aktuelle Objekt („dieses Objekt“), mit dem man gerade arbeitet. Mit *this.zaehler* wir daher das Attribut *zaehler* angesprochen. Damit sieht die verbesserte Version des Konstruktors so aus:

Listing 3.16: Konstruktor der Klasse Bruch, verbesserte Version

```
1  //2. Konstruktoren
2       public Bruch(int zaehler, int nenner) {
3           this.zaehler = zaehler;
4           this.nenner = nenner;
5       }
```

Der Nenner eines Bruchs darf nicht 0 sein. Wenn Sie das stört, benötigt Sie das *try/catch-Konstrukt* aus Exkurs 2.8, um damit umgehen zu können.

Wir können nun unserer Klasse um eine *main()*-Methode ergänzen, in der wir einen Bruch erzeugen und diesen Ausgeben. Das sieht dann so aus:

Listing 3.17: Klasse Bruch mit main()-Methode

```
1  public class Bruch {
2
3       //1. Attribute
4       int zaehler, nenner;
5
6       //2. Konstruktoren
7       public Bruch(int zaehler, int nenner) {
8           this.zaehler = zaehler;
9           this.nenner = nenner;
10      }
11
12      //3. Methoden
13
14      public static void main(String[] args) {
15          Bruch bruch1 = new Bruch(5, 8);
16          System.out.println(bruch1);
```

```
17      }
18  }
```

Die Ausgabe ist aber recht enttäuschend: In meinem Fall lautet sie *Bruch@4c98385c*. Dies besagt, dass es sich beim Objekt, auf das die Variable *bruch1* verweist, um ein Objekt der Klasse *Bruch* handelt und das diese Variable auf die Speicheradresse *4c98385c* verweist. Schöner wäre eine Ausgabe wie *5/8*.

toString()-Methode Alle Klassen in Java *erben* (englisch: *inherit*) von der Klasse *Object*. Das bedeutet, sie haben alle automatisch die Methoden der Klasse *Object*. Dazu gehört die Methode *toString()*, die immer dann automatisch verwendet wird, wenn ein Objekt als String dargestellt werden soll, wie in Zeile 16 des obigen Programms der Fall. Dort ist festgelegt, dass von einem Objekt Standardmäßig die Klasse und die Adresse ausgegeben werden. Möchte wir ein anderes verhalten der Methode *toString()*, dann müssen wir diese *überschreiben*, englisch *override*. Das bedeutet, das wir eine Methode *toString()* mit dem gewünschten Verhalten in unserer Klasse *Bruch* anlegen. Die Klasse sieht dann so aus:

Listing 3.18: Klasse Bruch mit main()- und toString()-Methode

```
1  public class Bruch {
2
3      //1. Attribute
4      int zaehler, nenner;
5
6      //2. Konstruktoren
7      public Bruch(int zaehler, int nenner) {
8          this.zaehler = zaehler;
9          this.nenner = nenner;
10     }
11
12     //3. Methoden
13     @Override
14     public String toString() {
15         return zaehler +"/" +nenner;
16     }
17
18     public static void main(String[] args) {
19         Bruch bruch1 = new Bruch(5, 8);
20         System.out.println(bruch1);
21     }
22 }
```

Jetzt bekommen wir die gewünschte Ausgabe *5/8*. Überschriebene Methoden werden üblicherweise mit der Annotation *@Override* gekennzeichnet.

Methode *multiplizieren()* Die folgend Klasse *Bruch* ist ergänzt um eine Methode *multiplizieren()*. In der *man()*-Methode werden zwei Brüche erzeugt und ausgegeben, dann werden sie miteinander multipliziert und das Ergebnis ausgegeben.

Listing 3.19: Klasse Bruch mit multiplizieren()-Methode

```
1
2  public class Bruch {
3
4      //1. Attribute
5      int zaehler, nenner;
6
7      //2. Konstruktoren
8      public Bruch(int zaehler, int nenner) {
```

```
 9           this.zaehler = zaehler;
10           this.nenner = nenner;
11       }
12
13       //3. Methoden
14       @Override
15       public String toString() {
16           return zaehler +"/" +nenner;
17       }
18
19       public Bruch multiplizieren(Bruch faktor) {
20           return new Bruch (this.zaehler*faktor.zaehler, this.nenner*faktor.nenner);
21       }
22
23       public static void main(String[] args) {
24           Bruch bruch1 = new Bruch(5, 8);
25           System.out.println(bruch1);
26           Bruch bruch2 = new Bruch(2, 3);
27           System.out.println(bruch2);
28           Bruch bruch3 = bruch1.multiplizieren(bruch2);
29           System.out.println(bruch3);
30       }
31   }
```

3.27: Aufgabe: Klasse *Bruch* fertig stellen

1. Bisher müssen die Brüche, mit denen gerechnet werden soll, fest im Programm eingegeben werden. Erstellen Sie einen weiteren Konstruktor ohne Parameter, in dem die Benutzer:in Zähler und Nenner eingeben kann.
2. Ergänzen Sie die Klasse *Bruch* um eine Methode zum dividieren durch einen adneren Bruch.
3. Ergänzen Sie Methoden, um Brüche addieren und subtrahieren zu können.
4. Erstellen Sie eine Methode, die den Kehrbruch des Bruchs zurück gibt, also aufgerufen auf *5/8* den Bruch *8/5* zurück gibt.
5. Erstellen Sie eine Methode, die den Bruch kürzt, also beispielsweise aufgerufen auf 10/24 den Bruch 5/12 zurück gibt. Modifizieren Sie die Methode dann auch noch so, dass es keine negativen Nenner gibt. Aus $10/-24$ soll also $-5/12$ werden und aus $-10/-24$ soll $5/12$ werden.

3.3.3 Weitere Konzepte der Objektorientierung am Beispiel *Vektor*

Ähnlich wie wir eben eine neue Klasse *Bruch* implementiert haben, um mit Bruchzahlen rechnen zu können, können wir eine Klasse Vektor erstellen. Ein Anfang ist im folgenden Listing gegeben, wobei an diesem Beispiel einige neue Dinge gezeigt werden.

Listing 3.20: Klasse Vektor

```
1
2  public class Vektor {
3
```

```
4       /**
5        * Attribut der Klasse; es nimmt die Koordinaten eines Vektors
6        * im dreidimensionalen Raum auf
7        */
8       private double[] koordinaten = new double[3];
9
10      /**
11       * Getter fuer das Attribut koordinaten
12       * @return das Attribut koordinaten
13       */
14      public double[] getKoordinaten() {
15          return koordinaten;
16      }
17
18      /**
19       * Setter fuer das Attribut koordinaten
20       * @param koordinaten die neuen Koordinaten
21       */
22      public void setKoordinaten(double[] koordinaten) {
23          this.koordinaten = koordinaten;
24      }
25
26      public static void main(String[] args) {
27          Vektor v = new Vektor();
28          System.out.println(v);
29      }
30  }
```

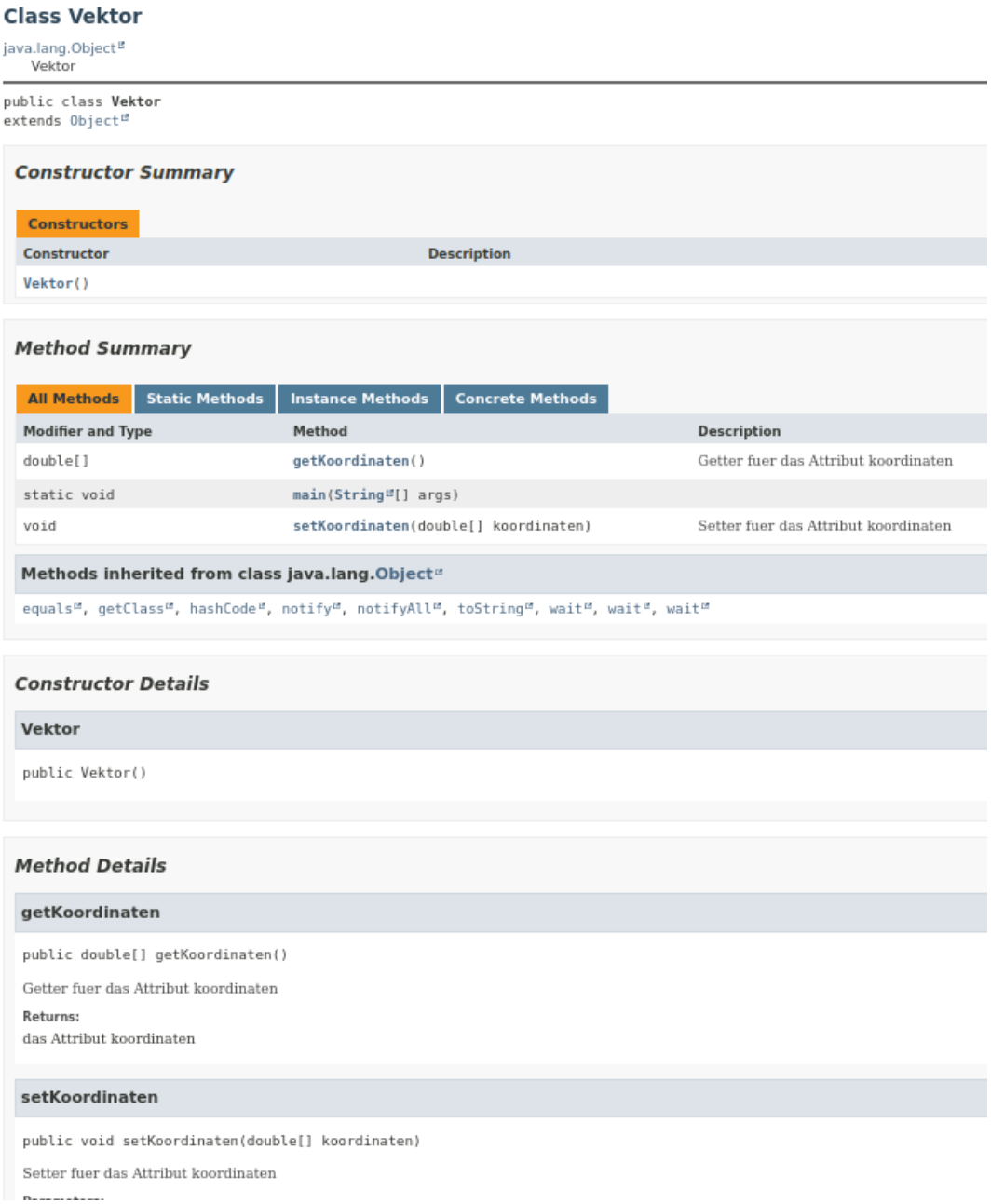
Folgende Dinge sind neu:

- Die Klasse enthält vor jedem Attribut und vor jeder Methode Kommentare, die mit /** statt dem üblichen /* für einen mehrzeiligen Kommentar eingeleitet werden.
- Es existieren zwei Methoden *getKoordinaten()* und *setKoordinaten()*, deren Zweck zunächst unklar ist.
- Die Klasse enthält keinen Konstruktor. Dennoch kann in der *main()*-Methode ein Konstruktor ohne Parameter aufgerufen werden und das so erzeugte Objekt scheint normal zu funktionieren.

Schauen wir uns diese Neuerungen der Reihe nach an. Die speziellen Kommentare dienen der Dokumentation. Wenn Sie eine Klasse in einem großen Projekt oder gar für einen Programm-Bibliothek erstellen, Dann müssen Sie die Funktionsweise für diese Klasse für Ihre Kolleg:innen oder die Benutzer:innen der Bibliothek dokumentieren. Eclipse (und andere IDEs) könne aus solchen *Javadoc*-Kommentaren ein Dokument erstellen, das bestimmten Normierungen genügt. Einen Ausschnitt eines so erzeugten Dokuments zeigt die Abbildung 3.3. Eine Javadoc können Sie ezeugen, indem Sie die Kommentare anlegen und dann in der Klasse auf *File -> Export -> Java -> Javadoc* gehen.

Die Methoden *getKoordinaten()* und *setKoordinaten()* heißen *Getter-Methode* bzw. *Setter-Methode* oder kurz *Getter* bzw. *Setter* für das Attribut *koordinaten*. Sie ermöglichen es, lesend bzw. schreibend von außerhalb der Klasse *Vektor* auf das Attribut *koefizienten* zuzugreifen, denn diese Methode sind public, das Attribut aber ist private. Man könnte natürlich einfach das Attribut public setzen und auf die beiden Methoden verzichten. Das hat aber zwei Nachteile:

- Vielleicht möchten Sie die Zugriffe feiner steuern, z.B. nur Lesezugriff, aber keinen Schreibzugriff von außerhalb geben. Dann könnten Sie den Setter weglassen oder stärker beschränken und nur den Getter public lassen.



Class Vektor

java.lang.Object
Vektor

public class **Vektor**
extends Object

Constructor Summary

Constructors

Constructor	Description
Vektor()	

Method Summary

All Methods | **Static Methods** | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method	Description
double[]	getKoordinaten()	Getter fuer das Attribut koordinaten
static void	main(String[] args)	
void	setKoordinaten(double[] koordinaten)	Setter fuer das Attribut koordinaten

Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Details

Vektor

public Vektor()

Method Details

getKoordinaten

public double[] getKoordinaten()

Getter fuer das Attribut koordinaten

Returns:
das Attribut koordinaten

setKoordinaten

public void setKoordinaten(double[] koordinaten)

Setter fuer das Attribut koordinaten

Abbildung 3.3: Ausschnitt aus der Javadoc der Klasse Vektor

- Eventuell bauen Sie die Klasse später um. Sie könnten beispielsweise statt karthesischer Koordinaten Kugelkoordinaten als Attribute verwenden. Wenn dann eine Kollegin bereits das alte Attribut in einer anderen Klasse verwendet, dann wird deren Klasse plötzlich nicht mehr funktionieren, weil Sie in Ihrer Klasse eine Änderung vorgenommen haben. Wenn Sie das Attribut niemals mit *public* frei gegeben haben, kann die Kollegin das Attribut nicht verwendet haben. Den Getter können Sie aber weiterhin beibehalten, indem Sie die kartheischen Koordinaten aus de Kugelkoordinaten berechnen.

Die Benutzung von Gettern und Settern ist bei kleinen Projekten einer Person etwas lästig, in Profi-Programmierung aber vollkommen üblich. Sie müssen in der Lage sein, Getter und Setter für Attribute zu erstellen.

Die Klasse *Vektor* enthält keinen expliziten Konstruktor, dennoch kann man, wie das Beispiel zeigt, Instanzen der Klasse erzeugen. Jede Klasse besitzt einen Konstruktor ohne Parameter. Entweder, in der Klasse ist ein solcher Konstruktor explizit angelegt, oder aber es liegt implizit der *Standardkonstruktor* vor. Der Standardkonstruktor ist ein Konstruktor ohne Parameter und mit leerem Rumpf. Die Klasse

Vektor verhält sich daher so, als ob folgender Konstruktor vorhanden wäre:

Listing 3.21: Expliziter Standardonstruktor der Klasse Vektor

```
1 public Vektor() {}
```

3.28: Aufgabe: Klasse *Vektor* fertig stellen

1. Ergänzen Sie die Klasse *Vektor* um eine geeignete *toStrng()*-Methode. Hinweis: Spaltenvektoren sind ohne graphische Ausgabe nicht machbar, beispielsweise können Sie keine großen Klammern um mehrere untereinander stehende Zahlen erzeugen. Arbeiten Sie daher mit Zeilenvektoren.
2. Implementieren Sie einen Konstruktor, der es der Benutzer:in des Programms ermöglicht, einen Vektor einzugeben.
3. Implementieren Sie Methoden, um einen Vektor mit einer Zahl zu multiplizieren, das Skalarprodukt zweier Vektoren und das Kreuzprodukt zweier Vektoren auszurechnen.
4. Implementieren Sie eine Methode, um den Betrag eines Vektors auszurechnen.

3.3.4 Aufgaben

3.29: Aufgabe: Matrizen

1. Implementieren Sie eine Klasse Matrix. Verwenden Sie als Attribut für die Koeffizienten ein Objekt vom Typ *double[][]*. Es handelt sich hier um ein Array von einem Array. Man kann es auch als zweidimensionales Array auffassen.
2. Ergänzen Sie die Klasse um einen Konstruktor zur Eingabe der Dimensionen der Matrix und ihrer Einträge und um eine *toString()*-Methode um die Matrix übersichtlich als String darzustellen.
3. Ergänzen Sie die Klasse um Methoden zur Matrizen-Addition und Matrizen-Multiplikation und um eine Methode, um die Matrix mit einer Zahl zu multiplizieren.

Die weiteren beiden Aufgaben sind sehr aufwändige Projektaufgaben. Sie können diese Aufgaben bearbeiten, wenn Sie etwas zusätzlich machen möchten. Sie werden im Unterricht nicht besprochen.

3.30: Aufgabe: Gerade

Erstellen Sie eine Klasse Gerade, um Geraden im dreidimensionalen Raum darzustellen. Verwenden Sie in Ihrem Projekt die zuvor erstellte Klasse *Vektor* und erstellen Sie in der Klasse *Gerade* zwei Attribute vom Typ *Vektor*, nämlich *stuetzvektor* und *richtungsvektor*. Implementieren Sie mindestens einen geeigneten Konstruktor und eine geeignet *toString()*-Methode. Erweitern Sie dann Ihr Programm um eine Methode, die als Parameter eine weitere Gerade entgegen nimmt und ermittelt, wie diese im Verhältnis zu der Geraden, auf der die Methode aufgerufen wird, liegt (identisch, parallel, schneidend, windschief).

3.31: Aufgabe: LGS

Erstellen Sie eine Klasse *LGS* zur Erzeugung und Lösung eines linearen Gleichungssystems. Benutzen Sie als Attribut ein zweidimensionales Array, das die erweiterte Koeefizientenmatrix des LGS hält.

1. Beginnen Sie mit LGS mit drei Unbekannten und drei Gleichungen, das eindeutig lösbar ist.
2. Erweitern Sie Ihr Programm auf eindeutig lösbare LGS mit je n Gleichungen und Unbekannten.
3. Erweitern Sie Ihr Programm so, dass es unlösbare LGS erkennen kann und dann statt der Lösung ausgibt, dass das LGS nicht lösbar ist.
4. Erweitern Sie Ihr Programm so, dass es auch LGS mit unendlich vielen Lösungen lösen kann und ggf. Lösungen mit Parametern ausgibt. Achtung: Schwer!
5. Erweitern Sie das Programm so, dass die Anzahl der Unbekannten und Gleichungen verschieden sein kann.

3.3.5 Vererbung

Stellen Sie sich vor, Sie möchten eine Software schreiben, mir der ein kleines Theater seine Vorstellungen verwaltet. In der Software gibt es vermutlich eine Klasse *Vorstellung* mit Attributen wie *autor* und *titel*, um das Stück und seine Autor:in zu beschreiben, *ticketpreis* und *laenge* für die Aufführungsdauer in Minuten. Außerdem möchten Sie ein Array von Booleans haben, wobei die Nummern des Arrays die Platznummern repräsentieren und der Wert *false* bedeutet, dass der Platz noch nicht verkauft wurde, während *true* bedeutet, dass der Platz bereits verkauft ist. Der Anfang der Klasse könnte so aussehen:

Listing 3.22: Klasse *Vorstellung* ohne Vererbung

```
1  public class Vorstellung {
2
3      private String autor;
4      private String titel;
5      private double ticketpreis;
6      private int laenge;
7      private boolean sitzplaetze;
8
9      public Vorstellung(String autor, String titel, double ticketpreis,
10             int laenge) {
11         this.autor = autor;
12         this.titel = titel;
13         this.ticketpreis = ticketpreis;
14         this.laenge = laenge;
15     }
16
17     public String getAutor() {
18         return autor;
19     }
20     public void setAutor(String autor) {
21         this.autor = autor;
22     }
23     public String getTitel() {
```

```
24          return titel;
25      }
26      public void setTitel(String titel) {
27          this.titel = titel;
28      }
29      public double getTicketpreis() {
30          return ticketpreis;
31      }
32      public void setTicketpreis(double ticketpreis) {
33          this.ticketpreis = ticketpreis;
34      }
35
36      //... und viele weitere Methoden...
37  }
```

Manchmal finden in dem Theater außerdem auch Rock-Konzerte statt. Die obige Klasse ist dann nur bedingt geeignet: Statt *Autor* und *Titel* des Stücks braucht man ein Attribut für den Namen der Band. Man könnte den Bandnamen unter *Autor* speichern und das Attribut *Titel* einfach nicht verwenden, aber das ist keine gute Lösung. Auch das Attribut *laenge* ist sinnlos, da Sie beim Konzert nicht genau wissen, wie lange es geht, außerdem machen die verkauften und freien Plätze keinen Sinn, da beim Rock-Konzert die Bestuhlung abgeräumt wird. Stattdessen benötigen Sie nun ein Attribut für den Namen der Band und die Anzahl der verkauften Tickets. Die Klasse könnte wie folgt beginnen:

Listing 3.23: Klasse *Konzert* ohne Vererbung

```
1   public class Konzert {
2
3       private String bandname;
4       private double ticketpreis;
5       private int anzahlVerkaufteTickets;
6
7       public Konzert(String bandname, double ticketpreis) {
8           this.setBandname(bandname);
9           this.setTicketpreis(ticketpreis);
10      }
11
12      public String getBandname() {
13          return bandname;
14      }
15
16      public void setBandname(String bandname) {
17          this.bandname = bandname;
18      }
19
20      public double getTicketpreis() {
21          return ticketpreis;
22      }
23
24      public void setTicketpreis(double ticketpreis) {
25          this.ticketpreis = ticketpreis;
26      }
27
28      //... und viele weitere Methoden...
29  }
```

Diese Lösung ist unbefriedigend. Wir benötigen zwei Klassen, weil bei Theatervorstellungen und

Rock-Konzerten nicht alles gleich ist. Aber nun verdoppeln wir einige Dinge wie den Ticketpreis, den es in beiden Klasse gibt. Wenn man später Änderungen an Methoden vornimmt, die etwas mit den Ticketpreis zu tun haben, dann muss man das immer in beiden Klassen machen. Das kann zu ungewollten Unterschieden führen, weil man eine der beiden Änderungen vergisst oder sonst einen Fehler macht. Solche Problem löst man am besten mit *Vererbung*.

Bei der Vererbung realisiert man die Gemeinsamkeiten mehrerer Klassen in einer *Superklasse*. Die einzelnen Klassen *erben* dann diese Gemeinsamkeiten. Um Vererbung zu implementieren, wird das Schlüsselwort *extends*, gefolgt vom Namen der Klasse, von der geerbt werden soll in der Deklaration der erbenden Klasse verwendet.

Eine Superklasse *Veranstaltung* für die Klassen *Vorstellung* und *Konzert* könnte so aussehen:

Listing 3.24: Superklasse *Veranstaltung*

```
1  public class Veranstaltung {
2
3      private double ticketpreis;
4
5      public Veranstaltung(double ticketpreis) {
6          this.ticketpreis = ticketpreis;
7      }
8
9      public double getTicketpreis() {
10         return ticketpreis;
11     }
12
13     public void setTicketpreis(double ticketpreis) {
14         this.ticketpreis = ticketpreis;
15     }
16
17     //... und viele weitere Methoden...
18 }
```

Die Klasse *Vorstellung* würde nun etwas kürzer:

Listing 3.25: Klasse *Vorstellung* mit Vererbung

```
1
2  public class Vorstellung extends Veranstaltung{
3
4      private String autor;
5      private String titel;
6      private int laenge;
7      private boolean sitzplaetze;
8
9      public Vorstellung(String autor, String titel, double ticketpreis,
10             int laenge) {
11         super(ticketpreis);
12         this.autor = autor;
13         this.titel = titel;
14         this.laenge = laenge;
15     }
16
17     public String getAutor() {
18         return autor;
19     }
20     public void setAutor(String autor) {
21         this.autor = autor;
```

```
22      }
23      public String getTitel() {
24          return titel;
25      }
26      public void setTitel(String titel) {
27          this.titel = titel;
28      }
29
30      //... und viele weitere Methoden...
31  }
```

Drei Dinge sind anders als in der vorigen Version der Klasse *Vorstellung* 3.22:

- Die Klassendeklaration wurde um *extends Vorstellung* erweitert, um *Vorstellung* von *Veranstaltung* erben zu lassen.
- Im Konstruktor fehlt die Zeile *this.ticketpreis = ticketpreis*, da es das Attribut *ticketpreis* in dieser Klasse nicht mehr gibt. Stattdessen wird mit *super(ticketpreis)* der Konstruktor der Superklasse *Veranstaltung* aufgerufen.
- Die Methoden, die mit dem Ticketpreis arbeiten, wurden weggelassen, weil diese von der Superklasse geerbt werden. Das sind hier zunächst nur Getter und Setter. Würde man aber das gesamte Programm schreiben, würden sich hier große Vorteile durch die Vererbung ergeben.

3.32: Aufgabe: Klasse *Konzert* mit Vererbung

Ändern Sie die Klasse *Konzert* 3.23 so ab, dass sie von der Klasse *Veranstaltung* 3.24 erbt.

3.8: Exkurs: Abstrakte Klassen

Oftmals gibt es Superklassen, von denen niemals eine Instanz erzeugt werden sollte. Der einzige Zweck solcher Klassen ist es, dass von ihnen geerbt werden kann und dann von ihren Subklassen Instanzen erstellt werden können. In den obigen Beispielen könnte man sich beispielsweise auf den Standpunkt stellen, dass eine *Veranstaltung* entweder eine *Vorstellung* oder ein *Konzert*, niemals aber nur eine Veranstaltung ist. Um Fehler zu vermeiden, kann es vorteilhaft sein, die Instanziierung einer solchen Superklasse explizit zu verbieten. Sie wird dadurch zu einer *abstrakten Klasse*. Das geschieht mit dem Schlüsselwort *abstract* in der Deklaration der Klasse. Man könnte die Klasse *Veranstaltung* als abstrakte Klasse deklarieren durch die Zeile *public abstract class Veranstaltung*.

3.9: Exkurs: Vererbungshierarchie und Mehrfachvererbung

Eine Klasse kann von einer Klasse erben, die von einer weiteren Klasse erbt usw. Somit entsteht eine *Vererbungshierarchie*. Streng genommen legt man bei jeder Vererbung bereits eine Vererbungshierarchie, denn in Java gibt es die Klasse *Object*, von der jede Klasse erbt. In der Klasse *Object* ist beispielsweise die Methode *toString()*, (siehe Abschnitt 3.3.2) grundlegend implementiert.
Mehrfachverbung unterscheidet sich von einer Vererbungshierarchie. Dabei erbt eine Klasse explizit von mehreren Klassen, die nicht voneinander geerbt haben. Mehrfachvererbung gibt es in manchen Sprachen, nicht aber in Java. Mehrfachvererbung kann sehr kompliziert werden, wenn beispielsweise mehrere Superklassen verschiedene Implementierungen von Methoden mit gleicher Signatur besitzen.
Manchmal hört man das Gerücht, Mehrfachvererbung wird in Java durch *Interfaces*, auf die hier nicht eingegangen werden soll, realisiert. Das ist nicht richtig. Interfaces legen Rollen von Instanzen einer Klasse fest. Wenn Sie einen Haushalt mit einem Java-Programm modellieren möchten, dann könnte ein Wachhund die Rolle eines Haustiers haben, an anderer Stelle die Rolle als Alarmanlage. Hier kann das Konzept der Interfaces hilfreich sein, dabei wird aber kein Code vererbt.

3.10: Exkurs: Probleme bei der Vererbung

Die Vererbung geht davon aus, dass es allgemeinere Klassen gibt und speziellere Klassen von den allgemeineren Klassen erben. Vererbung ist oft dann sinnvoll, wenn das Wort *ist* passt: Eine Vorstellung *ist* eine (spezielle) Veranstaltung, ein Konzert *ist* eine (spezielle) Veranstaltung. Daher ist es sinnvoll, eine Klasse *Veranstaltung* zu implementieren und die Klassen *Vorstellung* und *Konzert* von dieser Klasse erben zu lassen. Die allgemeinen Attribute, Konstruktoren und Methoden werden dann in der Superklasse implementiert, die speziellen Attribute, Konstruktoren und Methoden in den jeweiligen Subklassen.
Man könnte nun ein Graphikprogramm schreiben, mit dem man diverse geometrische Figuren zeichnen kann, beispielsweise Kreise, Dreiecke, Rechtecke und Quadrate. Nun *ist* ein Quadrat ein spezielles Rechteck und die Klasse *Quadrat* sollte daher von der Klasse *Rechteck* erben. Das funktioniert aber nicht: Die speziellere Klasse *Quadrat* benötigt nur ein Attribut *Kantenlänge a*, während die allgemeinere Klasse *Rechteck* zwei Attribute *Kantenlänge a* und *Kantenlänge b* benötigt. Wenn, dann müsste die Vererbung umgekehrt sein. Aber auch das macht Probleme, da z.B. eine Methode zum Zeichnen eines Quadrats nicht auf das Rechteck vererbt werden könnte.

3.4 Wertsemantik und Referenzsemantik

3.4.1 Wert- und Referenzsemantik bei Variablen

Zwei mögliche Semantiken Was ist die Bedeutung einer Variablen? Es gibt zwei Möglichkeiten:

- Eine Variable kann einen bestimmten Wert *bedeuten*. Das nennt man *Wertsemantik*, auf englisch *value semantics*.
- Eine Variable kann eine *Referenz* auf eine bestimmte Stelle im Speicher sein, wo dann der betreffende Wert steht. Statt *Referenz* könnte man auch *Zeiger* oder *Verweis* sagen, im Englischen sind die Begriffe *reference* oder *pointer* üblich. Das nennt man *Referenzsemantik*, auf englisch *reference semantics* oder *pointer semantics*.

Semantik bei primitiven Typen Variablen primitiver Typen in Java besitzen Wertsemantik. Das folgende Beispiel zeigt dies.

Listing 3.26: Wertsemantik bei primitiven Typen

```
1 public static void main(String[] args) {
2     int a = 3;
3     int b = 5;
4     System.out.println("a = " +a +" und b = " +b);
5     a = b;
6     System.out.println("a = " +a +" und b = " +b);
7     a = 3;
8     System.out.println("a = " +a +" und b = " +b);
9 }
```

Listing 3.27: Ausgabe des Beispiels Wertsemantik bei primitiven Typen

```
1 a = 3 und b = 5
2 a = 5 und b = 5
3 a = 3 und b = 5
```

Die Ausgabe ist so, wie man es erwartet: Die Werte von a und b sind zunächst 3 und 5, dann wird a gleich b gesetzt, so dass dann a und b beide gleich 5 sind, dann wird a wieder auf 3 gesetzt und a und b besitzen wieder die Werte 3 bzw. 5.

Semantik bei Objekttypen Variablen von Objekttypen besitzen in Java Referenzsemantik. Um ein ähnliches Beispiel wie oben konstruieren zu können, erzeugen witr zunächst eine Klasse *MeinInteger*. Objekte dieser Klasse repräsentieren ganze Zahlen, sind aber Objekte. Die Klasse *MeinInteger* kann so aussehen;

Listing 3.28: MeinInteger

```
1  public class MeinInteger {
2
3      /**
4      * Das Attribut, das den Wert des Integers speichert
5      */
6      int wert;
7
8      /**
9      * Ein Konstruktor
10     * @param wert der Wert des neuen MeinInteger
11     */
12     public MeinInteger(int wert) {
13         this.setWert(wert);
14     }
15
16     /**
17     * Setter fuer Mein Integer
18     * @param wert der neue Wert
19     */
20     public void setWert(int wert) {
21         this.wert = wert;
22     }
23
24     /**
25     * Methode, um ein Objekt in einen String umzuwandeln
26     * Der leere String "" dient der Konverterung des int wert in einen String
27     */
28     @Override
29     public String toString() {
30         return "" + wert;
31     }
32 }
```

Konstruiert man damit nun ein ähnliches Beispiel, dann erlebt man eine kleine Überraschung.

Listing 3.29: Referenzsemantik bei Objekttypen

```
1  public static void main(String[] args) {
2      MeinInteger c = new MeinInteger(3);
3      MeinInteger d = new MeinInteger(5);
4      System.out.println("c = " +c +" und d = " +d);
5      c = d;
6      System.out.println("c = " +c +" und d = " +d);
7      c.setWert(3);
8      System.out.println("c = " +c +" und d = " +d);
9  }
```

Listing 3.30: Ausgabe des Beispiels Referenzsemantik bei Objekttypen

```
1  c = 3 und d = 5
2  c = 5 und d = 5
3  c = 3 und d = 3
```

Die ersten beiden Zeilen sind wie erwartet und gleich wie im Beispiel mit Wertsemantik. In der letzten Zeile hat sich aber plötzlich der Wert von d verändert. Was ist passiert? Im Programm wird in Zeile 5 c gleich d gesetzt. Unter Referenzsemantik bedeutet dies, dass die Variablen c und d ab

nun auf dieselbe Stelle im Speicher verweisen. Man könnte auch sagen, c und d sind ab nun nur zwei verschiedene Namen für dieselbe Speicherstelle. Wird nun in Zeile 7 c geändert, dann ändert sich auch d.

Mit Kenntnis der Referenzsemantik wird auch der Unterschied zwischen == und der Methode $equals()$ bei Objekttypen klar: == prüft, ob zweimal auf die selbe Speicherstelle verwiesen wird. $equals()$ prüft, ob an den beiden Stellen gleiche Objekte gemäß der jeweiligen Implementierung von $equals()$ steht.

Referenzsemantik hat beim Programmieren manchmal Vorteile, oft ist sie aber ein Ärgernis, da ein Verhalten wie im eben vorgestellten Beispiel kontraintuitiv ist und daher leicht zu Fehlern führt. Der Hauptgrund, warum Programmiersprachen mit Referenzsemantik arbeiten, ist der, dass sie leichter technisch umsetzbar ist. Für den Inhalt einer Variable mit Wertsemantik muss der notwendige Speicherplatz für den Wert reserviert werden. Ein Objekt kann aber unterschiedlich viel Speicher benötigen, da es beispielsweise Strings, Arrays oder Listen unterschiedlicher Länge als Attribute besitzen kann.

Immutables Man kann Wertsemantik auch für Objekttypen bekommen, wenn man unveränderliche Variablen, oft englisch *immutables* genannt, verwendet. Das obige Beispiel *MeinInteger* ist effektiv eine Wrapperklasse für den primitiven Typen *int*. Es existiert aber eine Wrapperklasse in der Bibliothek, die Klasse *Integer*. Wiederholt man das obige Beispiel mit dieser Klasse, dann erhält man folgendes:

Listing 3.31: Quasi-Wertsemantik bei Immutables

```
1 public static void main(String[] args) {
2     Integer e = 3;
3     Integer f = 5;
4     System.out.println("e = " +e +" und f = " +f);
5     e = f;
6     System.out.println("e = " +e +" und f = " +f);
7     e = 3;
8     System.out.println("e = " +e +" und f = " +f);
9 }
```

Listing 3.32: Ausgabe des Beispiels Quasi-Wertsemantik bei Immutables

```
1 e = 3 und f = 5
2 e = 5 und f = 5
3 e = 3 und f = 5
```

Hier liegt offenbar Wertsemantik vor, denn der Wert der Variablen f wird nicht verändert. Objekttypen haben in Java aber Refgerenzsemantik. Die Wertsemantik hier wird simuliert. Der Trick ist, dass bei der Zuweisung des Werts *3* zur Variable e in Zeile 7 ein neues Objekt erzeugt wird und die Variable e nun auf die Speicherstelle dieses neuen Objekts zeigt und daher bei Änderung von e der Wert von f nicht verändert wird.

3.33: Aufgabe: Wert- und Referenzsemantik

Ermitteln Sie durch geeignete Beispielprogramme, welche Semantik in folgenden Fällen vorliegt:

1. Variablen vom Typ *double*.
2. Variablen vom Typ *double[]*.
3. Variablen vom Typ *String*.

3.4.2 Call by Value und Call by Reference

Mit der Problematik von *Wertsemantik* und *Referenzsemantik* hängt die Problematik von *Call by Value* und *Call by Reference* bei der Parameterübergabe an Methoden zusammen. Die deutschen Begriffe *Wertparameter* bzw. *Referenzparameter* sind sehr unüblich und werden hier daher nicht verwendet. Wenn eine Methode mit einer Variablen als aktuellen Parameter aufgerufen wird, dann gibt es zwei Möglichkeiten, wie der Parameter der Methode übergeben wird:

- Eine Kopie des Werts des Parameters wird erstellt und diese Kopie wird der Methode übergeben. Das ist *Call by Value*. Wird innerhalb der Methode der Wert des Parameters verändert, dann wird nur diese Kopie verändert, der Wert der Variablen, die als aktueller Parameter beim Methodenaufruf übergeben wurde, bliebt unverändert.
- Eine Referenz auf die Speicherstelle, wo der betreffende Wert der Variablen steht, wird übergeben. Das ist *Call by Reference*. Wird innerhalb der Methode der Wert des Parameters verändert, dann wird auch der Wert der Variablen, die als aktueller Parameter beim Methodenaufruf übergeben wurde, verändert, da der Speicherinhalt verändert wird, wo der Wert dieser Variablen steht.

Das folgende Beispiel illustriert das Problem.

Listing 3.33: Call by Value oder Call by Reference?

```
1  public class Semantik {
2      static int a = 3;
3
4      public static void main(String[] args) {
5          System.out.println("a = " +a);
6          methode(a);
7          System.out.println("a = " +a);
8      }
9
10     private static void methode(int a) {
11         System.out.println("a = " +a);
12         a = 42;
13         System.out.println("a = " +a);
14     }
15 }
```

Listing 3.34: Ausgabe des Beispiels Call by Value oder Call by Reference?

```
1  a = 3
2  a = 3
3  a = 42
4  a = 3
```

Wer Wert der Variablen *a* wid zuerst in der Zeile 5 ausgegeben, dann zweimal in der Methode *methode()* in den Zeilen 11 und 13, wobei man, wenig überschend, die Ausgaben *3*, *3* und *42* erhält. Spannend ist nun, was die Ausgabe in Zeile 7 nach der Rückkehr des Programmablaufs aus der Methode *methode()* liefert. Wenn Java Call by Value macht, dann muss *3* ausgegeben werden, da die Zuweisung der Zahl *42* in der Methode *methode()* dann nur an die Kopie der Variablen erfolgt ist, hier aber das Original ausgegeben wird. Das ist auch der Fall. Man erkennt daran, dass die Parameterübergabe in Java mit Call by Value erfolgt. Bei Call by Reference hätte die Zuweisung in der Methode das Original verändert und die letzte Ausgabe hätte *42* lauten müssen.

Call by Value mit Objekttypen Was passiert, wenn man im obigen Beispiel statt eines Parameters des primitiven Typen *int* einen Parameter eines Objekttypen übergibt wie im folgenden Beispiel? Die Klasse *MeinInteger* ist die Klasse aus Beispiel 3.28.

Listing 3.35: Methodenauruf mit einem Objekttypen als Parameter

```
1  public class Semantik2 {
2      static MeinInteger b = new MeinInteger(3);
3
4      public static void main(String[] args) {
5          System.out.println("b = " +b);
6          methode(b);
7          System.out.println("b = " +b);
8      }
9
10     private static void methode(MeinInteger b) {
11         System.out.println("b = " +b);
12         b.setWert(42);
13         System.out.println("b = " +b);
14     }
15 }
```

Listing 3.36: Ausgabe des Beispiels Methodenauruf mit einem Objekttypen als Parameter

```
1  b = 3
2  b = 3
3  b = 42
4  b = 42
```

Das Ergebnis ist anders als im vorigen Beispiel. Effektiv ist das Call by Reference. Streng genommen liegt aber auch hier Call by Value vor. Weil aber die Variable *b* hier eine Objektvariable ist, beinhaltet sie nicht einen Wert, sondern eine Referenz. Die Kopie der Referenz zeigt auf dieselbe Stelle wie die Original-Referenz, daher verändert die Wertzuweisung in Zeile 12 den Wert, der an der referenzierten Speicherstelle steht und daher wird in Zeile 7, nach der Rückkehr aus der Methode, der veränderte Wert ausgegeben. Will man das verhindern, dann müsste man die Zeile 12 durch die Zeile *b = new MeinInteger(42);* ersetzen. Dann wird nicht der alten Referenz ein anderer Wert zugeordnet, sondern ein neues Objekt mit dem anderen Wert erzeugt. Der Originalwert bleibt dann erhalten. Genau das passiert auch, wenn man mit einem Parameter einer unveränderlichen Klasse (Immutable) arbeitet.

3.34: Aufgabe: Semantik beim Methodenaufruf mit einem String-Parameter

Ermitteln Sie durch ein geeignetes Beispielprogramm, ob beim Aufruf einer Methode mit einem Parameter vom Typ String, dem innerhalb der Methode ein neuer Wert zugeordnet wird, das Original erhalten bleibt. Erklären Sie das beobachtete Verhalten.

4 Technische Informatik

4.1 Zahlendarstellungen

4.1.1 Additionssysteme und Stellenwertsysteme

Die römischen Zahlen als ein Beispiel für ein Additionssystem Die *römischen Zahlen* sind die alten, europäischen Zahlen. Für kleinere Zahlen gibt es die Symbole aus Tabelle 4.1. Weitere Zahlen werden

Tabelle 4.1: Römische Ziffern

römische Ziffer	I	V	X	L	C	D	M
Bedeutung als arabischen Zahl	1	5	10	50	100	500	1000

durch Hintereinanderschreiben dieser Ziffern und Addition von deren Werten erhalten. Beispiele finden Sie in der Tabelle 4.2

Tabelle 4.2: Einige römische Zahlen

römische Zahl	II	III	VI	VII	XI	XXVI	MMXXV
Bedeutung als arabischen Zahl	2	3	6	7	11	26	2025

4.1: Exkurs: Abweichungen der römischen Zahlen vom reinen Additionssystem

Die Ziffern in den römischen Zahlen sind von links nach rechts in der Regel nach abnehmender Wertigkeit sortiert. Teilweise, vor allem bei Verwendung römischer Zahlen in der nachrömischen Zeit, wird die vierfache Wiederholung derselben Ziffer vermieden. Das kann man erreichen, indem man man zulässt, dass eine Ziffer links vor einer Ziffer mit fünffachem oder zehnfachem Wert steht. Die niederwertige Ziffer links von einer höherwertigen Ziffer wird dann negativ gewertet. Beispiele finde sich in der Tabelle 4.3.

Tabelle 4.3: Einige römische Zahlen mit Subtraktion (betrifft Exkurs 4.1)

römische Zahl	IV	IX	XL	XLIXI	XC	XCIV	XCIX
Bedeutung als arabischen Zahl	4	9	40	49	90	94	99

Ein solches Zahlensystem ist sehr einfach und naheliegend. Man es ein *Additionssystem*. Additionssysteme haben aber zwei gravierende Nachteile:

1. Für größere Zahlen benötigt man immer weitere Ziffern, wenn die Zahlen nicht sehr lang werden sollen.
2. Additionen können in einem Additionssystem recht gut durchgeführt werden, die Subtraktion macht kaum mehr Probleme. Mutliplikationen und Divisionen und weitere Operationen sind aber nicht gut machbar.

Das Dezimalsystem: Eine Stellenwertsystem Die *arabischen Zahlen* funktionieren völlig anders. Sie bilden kein Additionssystem, sondern ein *Stellenwertsystem*. Es gibt die Ziffern von 0 bis 9. In der Zahl 2025 kommen die Ziffern *2*, *0* und *5* vor. Die Ziffer *2* kommt zweimal vor, da sie aber an zwei unterschiedlichen *Stellen* vorkommt, hat sie jeweils unterschiedliche *Werte*, daher der Name

Tabelle 4.4: Additions- und Multiplikationstabelle 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	7
9	9	10	11	12	13	14	15	16	17	18

·	**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	27	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	46	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	42	56	64	72	81

Stellenwertsystem: Die erste *2* (von links) bedeutet zwei Tausender, der Stellenwert ist also Tausend. Die weitere *2* bedeutet zwei Zehner, der Stellenwert ist also zehn.

Die Stellenwerte in den arabischen Zahlen sind von rechts nach links Einer, Zehner, Hunderter, Tausender, Zehntausender und so weiter. Man könnte auch sagen 10^0, 10^1, 10^2, 10^3, 10^4 und so weiter. Die Zahl *10* spielt als Basis der Stellenwerte eine große Rolle bei den arabischen Zahlen, damit zusammen hängt dann auch, dass es genau zehn Ziffern gibt. Man nennt das arabische Zahlsystem in seiner heutigen Form daher auch das *Dezimalsystem*.

Ein Stellenwertsystem wie das Dezimalsystem vermeidet die Nachteile der Additionssysteme: Man kann alle Zahlen mit einem endlichen Vorrat an Ziffern ausdrücken. Die Länge der Zahlen bleibt dennoch in der Praxis meistens überschaubar. Berechnungen sind einfach möglich. Letztlich muss man nur alle Ziffern mit allen Ziffern addieren und multiplizieren können, um alle Zahlen miteinander addieren und multiplizieren zu können. Anders ausgedrückt, man muss die Additionstabelle und die Multiplikationstabelle 4.4 kennen.

4.2: Exkurs: Herkunft und Schreibrichtung der arabischen Zahlen

Die arabischen Zahlen kamen im frühen Mittelalter mit den Arabern nach Europa. Vorläufer dieses Zahlensystems stammen aber wohl aus Indien, siehe https://de.wikipedia.org/wiki/Arabische_Zahlschrift.
Die Stellenwerte innerhalb einer arabischen Zahl wurden oben entgegen der in europäischen Sprachen üblichen Richtung von rechts nach links angegeben, da nur das sinnvoll möglich ist. Dies die Schreibrichtung der arabischen Sprache. Diesen Einfluss erkennt man auch, wenn man mit arabischen Zahlen rechnet: Beim schriftlichen Addieren beginnt man rechts, damit man einen gegebenenfalls auftretenden Übertrag an die Stelle mit dem nächst höheren Wert nach links weiter geben kann. Vermutlich ist auch die Vertauschung von Zehnerstelle und Einerstelle beim Sprechen der arabischen Zahlen in der deutschen Sprache („Dreiundzwanzig“ für 23) eine Folge davon.

4.1.2 Das Dualsystem

Darstellung von Zahlen im Dualsystem Was ist besonders an der Zahl *10*, dass sie zur Basis des heute weltweit üblichen Zahlensystems wurde? Vermutlich die Tatsache, dass wir Menschen zehn Finger haben und daher im Dezimalsystem ganz gut mit den Fingern rechnen können. Ansonsten kann jede natürliche Zahl ab *2* zur Basis eines Zahlensystems werden.

Das einfachste Stellenwertsystem ist daher das *Zweiersystem*. Statt *Zweiersystem* nennt man es meistens *Dualsystem* oder *Binärsystem*. Im Dezimnalsystem benötigt man zehn Ziffern. Im Dualsystem benötigt man nur zwei Ziffern, *0* und *1*. Die Stellenwerte in Dezimalsystem waren 10^0, 10^1, 10^2, 10^3, 10^4, daher sind sie im Dualsystem 2^0, 2^1, 2^2, 2^3, 2^4, also Einer, Zweier, Vierer, Achter, Sechzehner und so weiter.

Das Dualsystem wird in allen Computern verwendet. Wenn Sie eine Berechnung auf Ihrem Taschenrechner durchführen, dann rechnet der Taschenrechner zunächst Ihre Eingaben ins Dualsystem um, berechnet im Dualsystem das Ergebnis uns rechnet dieses ins Dezimalsystem um, um es Ihnen zu präsentieren. Wenn Sie an Ihrem Computer einen Text schreiben, im Internet surfe oder sonst etwas tun: Alle Aufgaben löst Ihr Computer durch Berechnungen im Dualsystem. Das Dualsystem spielt in der Computerwelt eine so große Rolle, weil Schalter mit zwei Stellungen – Spannung ein / Spannung aus – billig sind, Schalter mit mehr Stellungen, z. B. zwischen *0* und *9*, viel teurer.

Die Zahl 10110_2 bedeutet, umgerechnet ins Dezimalsystem, $1 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 = 22_{10}$. Die tief gestellten Zahlen geben die Basis des jeweiligen Zahlensystems an. Die *10* im Dezimalsystem wird dabei oft weggelassen.

4.1: Aufgabe: Verständnis des Dualsystems

Erklären Sie die folgende Behauptung: *Es gibt 10 Arten von Menschen: Die einen verstehen das Dualsystem, die anderen nicht.*

4.2: Aufgabe: Umrechnung vom Dualsystem ins Dezimalsystem

1. Die größte Dualzahl mit fünf Stellen ist $11111_2 = 31_{10}$. Überlegen Sie, wie man mit den Fingern einer Hand bis 31 Zählen kann. Überlegen Sie, wie weit man mit den Fingern beider Hände zählen könnte.
2. Lösen Sie diese Übungen, bis Sie mindestens 200 Punkte haben und sich auf der Highscore-Liste eintragen können.
3. Erstellen Sie ein Java-Programm, in das eine Dualzahl eingeben werden kann und der Wert der Zahl als Dezimalzahl ausgegeben wird.

Die Umrechnung in umgekehrter Richtung, von einer Dezimalzahl in einen Dualzahl ist etwas schwerer. Für Rechnungen im Kopf bietet sich folgendes Verfahren an, das am Beispiel 23 vorgestellt werden soll.

- Man sucht die größte Zweierpotenz, die maximal so groß ist wie die gegebene Zahl. Für *23* ist das $2^4 = 16$. Die Zahl *16* ist als Dualzahl 10000_2. Nun müssen die Werte für die weiteren Stellen, also die Achter-, Vierer-, Zweier- und Einerstelle dieser Zahl ermittelt werden.
- Von *23* wird *16* subtrahiert. Man erhält *7*. Diese *7* muss nun in die Dualzahl eingebaut werden. Die Zweierpotenz $2^3 = 8$ passt nicht in *7*, daher muss die nächste Ziffer in dieser *0* lauten. Es passt aber die nächst kleinere Zweierpotenz $2^2 = 4$, daher muss an der Viererstelle eine *1* stehen.
- Wir haben nun 10100_2. Dabei sind die letzten beiden Ziffern, die Zweier- und die Einerstelle noch unklar. $10100_2 = 20$, es fehlen also noch *3*. Durch analoges Vorgehen wie bisher kann man feststellen, dass die Zweier- und die Einerstelle *1* sein müssen. Wie erhalten $23_{10} = 10111_2$.

Ein anderes Verfahren funktioniert durch fortwährendes Dividieren durch *2* und Betrachten der Reste:

- $23 = 2 \cdot 11 + 1$
- $11 = 2 \cdot 5 + 1$
- $5 = 2 \cdot 2 + 1$
- $2 = 2 \cdot 1 + 0$
- $21 = 2 \cdot 0 + 1$

Liest man die rechte Spalte in den obigen Rechnungen von unten nach oben, so ergibt sich 10111; das ist das gesuchte Ergebnis.

4.3: Aufgabe: Umrechnung vom Dezimalsystem ins Dualsystem

1. Erklären Sie, wie das zweite der beiden Verfahren zur Umrechnung funktioniert.
2. Lösen Sie diese Übungen, bis Sie mindestens 200 Punkte haben und sich auf der Highscore-Liste eintragen können.
3. Erstellen Sie ein Java-Programm, in das eine Dezimalzahl eingeben werden kann und der Wert der Zahl als Dualzahl ausgegeben wird.

Rechnen im Dualsystem Man kann im dualsystem genauso rechnen wie im Dezimalsystem. Die Additions- und Multiplikationstabellen 4.5 sind erfreulich kurz.

Tabelle 4.5: Additions- und Multiplikationstabelle im Dualsystem

+	**0**	**1**		·	**0**	**1**
0	0	1		**0**	0	0
1	1	10		**1**	0	1

Das folgende Beispiel zeigt die Addition $100_{10} + 41_{10} = 141_{10}$ im Dualsystem.

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

4.3: Exkurs: Das Hexadezimalsysten

Computer arbeiten immer mit dem Dualsystem. Für Menschen ist es unhandlich, da es zu langen, mnemotechnisch schlechten Zahlen führt. Stellen Sie sich vor, Sie müssten die Telefonnummern Ihrer Freunde im Dualsystem auswendig lernen! Die Umrechnung zwischen Dualsystem und Dezimalsystem ist aber relativ rechenintensiv. Als Kompromiss wird an Stellen, an denen Zahlen verwendet werden, die Computersysteme verarbeiten müssen, die aber auch manchmal von Menschen notiert oder diktiert werden müssen Zahlen im *Hexadezimalsystem* verwendet.
Das Hexadezimalsystem ist das Stellenwertsystem zur Basis 16. Die Stellenwerte betragen von rechts nach links $16^0 = 1, 16^1 = 16, 16^2 = 256, 16^3 = 4096$ und so weiter. Man benötigt 16 Ziffern. Außer den Ziffern *0, 1, 2, 3, 4, 5, 6, 7, 8, 9* benutzt man die Buchstaben a, b, c, d, e und *f*. Sie repräsentieren Ziffern, deren Wertigkeit im Dezimalsystem 10, 11, 12, 13, 14 bzw. 15 betragen. Die Zahl $2af_{16}$ bedeutet im Dezimalsystem $2 \cdot 256 + 10 \cdot 16 + 15 \cdot 1 = 512 + 160 + 15 = 687$. Zahlen sind im Hexadezimalsystem nie länger, oft kürzer, als im Dezimalsystem und mnemotechnisch besser als Zahlen im Dualsystem und damit für Menschen besser als Dualzahlen. Für Computer sind die Hexadezimalzahlen besser als Dezimalzahlen, da die Umrechnung kaum Rechenkapazität benötigt. 16 ist eine Zweierpotenz. Jeder Ziffer im Hexadezimalsystem entspricht genau eine der 16 maximal vierstelligen Zahlen im Dualsystem. Die folgende Tabele zeigt dies, wobei die Dualzahlen durch führende Nullen alle auf die Länge vier gebracht wurden.

16er	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
Dual	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

Umrechnungen können dann durch Konkatenation bewerkstelligt werden. Beispiel: $1caffee3e_{16}$ = 0001 1100 1010 1111 1111 1110 1110 0011 1110_2. Die Lücken on der Dualzahl dienen der Kenntlichmachung der Vierergruppen. Die führenden Nullen in der Dualzahl kann man weglassen.

Verwendungen für Hexadezimalzahlen bibt es beispielsweise bei

- IP-Adressen („Internetadressen“) in der Version 6, z.B. 2003:e8:70f:9e0a:f9fa:45e8:8276:dc5c.
- MAC-Adressen, das sind die Hardwareadressen von Netzwerkgeräten, z.B. 34:e1:2d:ef:6c:ee
- Farben werden auf Bildschirmen durch die unterschiedliche Helligkeit der Grundfarben *Rot*, *Grün* und *Blau* der einzelnen Pixel dargestellt. Dabei kann jede Grundfarbe zwischen 0 und 255 eingestellt werden, das ist im Hexadezimalsystem eine Zahl zwischen 0 und ff. Farben im Hexadezimalsystem werden meist durch das Zeichen # gekennzeichnet. Beispielsweise bedeutet #ff0025: Rot auf volle Stärke ($ff_{16} = 255$), kein Grün (00), ein kleiner Blauanteil ($25_{16} = 37$).

4.4: Exkurs: Das Vierersystem in der Biologie

In der DNA treten die Basen Adenin (A), Guanin (G), Cytosin (C) und Thymin (T) auf. MAn kann A, G, C nd T als die vier Ziffern des Vierersystems auffassen und sagen, dass die Erbinformation in der DNA als Zahl im Vierersystem vorliegt.

4.4: Aufgabe: Rechnen im Dualsystem

1. Machen Sie sich ein paar weitere Beispiele von Additionen im Dualsystem.
2. Üben Sie das Multiplizieren im Dualsystem anhand einiger selbst gewählter Beispiele.

Überprüfen Sie Ihre Ergebnisse im Dezimalsystem.

4.1.3 Das Zweierkomplement

Im Dualsystem, wie es bisher eingeführt wurde, können nur nicht negative Zahlen ausgedrückt werden. Das Voranstellen des Zeichens − ist keine Lösung, denn dies wäre ein drittes Symbol neben *0* und *1* und würde die gute Darstellbarkeit der Dualzahlen in elektronischen Systemen stören.

Für die Darstellung ganzer Zahlen wird das *Zweierkomplement* verwendet. Dabei wird das höchstwertige Bit per Vereinbarung mit einem negativen Vorzeichen Versehen. Als Beispiel soll das an einer Zahl mit 8 bit = 1 B (Das Einheitenzeichen B steht für *Byte*) gezeigt werden. Die Stellenwerte im Zweierkomplement sind dann von links nach rechts −128, 64, 32, 16, 8, 4, 2, 1. Die Zahl 10000110_2 bedeutet dann$-128 + 4 + 2 = -122$.

4.5: Aufgabe: Umrechnungen mit dem Zweierkomplement

1. Üben Sie mit dieser Übung das Umrechnen von Zweierkomplementen in Dezimalzahlen mit Vorzeichen.
2. Üben Sie mit dieser Übung das Umrechnen von Dezimalzahlen mit Vorzeichen in Zweierkomplemente.

Das Rechnen im Zweierkomplement funktioniert korrekt, solange der darstellbare Zahlenbereich nicht verlassen wird. Es sind keine Fallunterscheidungen in Abhängigkeit davon, ob kein, ein oder beide Operanden negativ sind. Hierzu einige Beispiele: Die Rechnung $100 + (-50) = 50$ sieht im Zweierkomplement so aus:

1. Summand		0	1	1	0	0	1	0	0
2. Summand		1	1	0	0	1	1	1	0
Übertrag	1	1	0	0	1	1			
Ergebnis	(1)	0	0	1	1	0	0	1	0

Es gibt einen Übertrag in eine neunte Stelle, die es nicht gibt. Dieser Übertrag wird nicht beaachtet. Das Ergebnis $00110010_2 = 50$ ist korrekt.

Die Rechnung $1100 + 50 = -50$ sieht im Zweierkomplement so aus:

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

Auch das ist korrekt. Auch die Addition zweier negativer Zahlen, hier am Beispiel $-35 + (-25) = -60$ gezeigt, funktioniert:

1. Summand		1	1	0	1	1	1	0	1
2. Summand		1	1	1	0	0	1	1	1
Übertrag	1	1	1	1	1	1	1	1	
Ergebnis	(1)	1	1	0	0	0	1	0	0

Die Berechnung $-70 + (-60) = -130$ allerdings muss versagen, weil die kleinste Zahl, die im 8 bit-Zweierkomplement dargestellt werden kann, die Zahl $10000000_2 = -128$ ist.

1. Summand		1	0	1	1	1	0	1	0
2. Summand		1	1	0	0	0	1	0	0
Übertrag	1								
Ergebnis	(1)	0	1	1	1	1	1	1	0

Dieses Ergebnis ist +126 und somit falsch.

4.5: Exkurs: Name *Zweierkomplement*

Das *Zweierkomplement* heißt so, weil bei der Addition einer Zahl und ihres Inversen bezüglich der Addition, also z.B. 60 und −60, alle Stellen ab der ersten *1* (von rechts) sich zu zwei „komplettieren“. Das Ergebnis ist somit *0*, wie es sein sollte.

1. Summand		0	0	1	1	1	1	0	0
2. Summand		1	1	0	0	0	1	0	0
Übertrag	1	1	1	1	1	1			
Ergebnis	(1)	0	0	0	0	0	0	0	0

4.6: Aufgabe: Datentyp Byte

Der Datentyp *Byte* ist eine 8-bit-Zweierkomplementzahl. Betrachten Sie die Ausgabe des folgenden Programms und erklären Sie sie.

```
1 public static void main(String[] args) {
2     byte n = 0;
3     for(int i = 0; i < 300; i++) {
4         System.out.println(n);
5         n++;
6     }
7 }
```

4.7: Aufgabe: Andere Ganzzahltypen

Auch die Datentypen *short*, *int* und *long* sind Zweierkomplementzahlen. Ermitteln Sie die jeweils kleinste und größte darstellbare Zahl. Benutzen Sie entweder ein Programm wie in Aufgabe 4.6 oder benutzen Sie die Tastache, dass diese Daentypen 2 B, 4 B bzw. 8 B Speicher belegen.

Ganze Zahlen kann man am Zahlenstrahl darstellen. Der Zahlenstrahl ist nach links und nach rechts jeweils unendlich lang, da es unendlich viele positive und negative ganze Zahlen gibt. Diese perfekte Darstellung der ganzen Zahlen kann ein Programmiersystem nicht leisten, da dafür unendlich viel Speicher notwendig wäre. Im Zweierkomplement degeneriert der Zahlenstrahl zu einem Zahlenkreis, das für

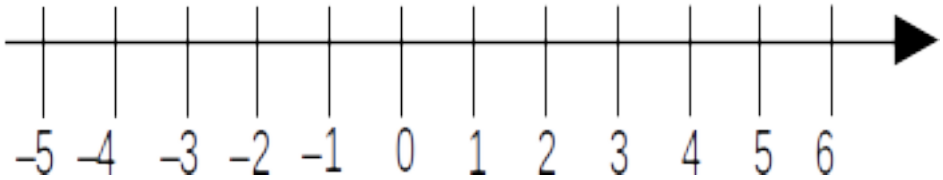


Abbildung 4.1: Zahlenstrahl

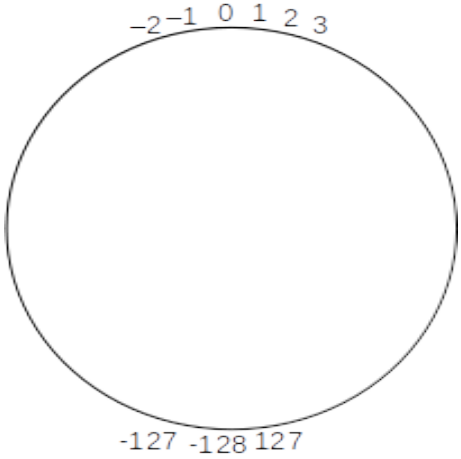


Abbildung 4.2: Zahlenkreis

8-bit-Zweierkomplementzahlen aussieht wie in Abbildung 4.2. In der Abbildung sind aus Platzgründen nur die Zahlen um 0 und um −128 dargestellt. Bewegt man sich im Uhrzeigersinn, dann werden die Zahlen größer, entgegen dem Uhrzeigersinn kleiner. Ausnahme: Geht man von der größten darstellbaren Zahl 127 eins weiter im Uhrzeigersinn, dann landet man bei der kleinsten darstellbaren Zahl −128, die gegenüber der 0 sitzt.

4.6: Exkurs: Ganze Zahlen mit Vorzeichen und Betrag

Viel naheliegender und zunächst einfacher ist es, ganze Zahlen mit Vorzeichen und Betrag darzustellen. Diese Darstellung wurde in den Anfangszeiten der Computer benutzt. Dabei repräsentiert das Bit ganz links das Vorzeichen der Zahl. *1* steht für −, *0* steht für +. Die weiteren Bits stellen den Betrag der Zahl dar.
In diesem System hat die Zahl *0* zwei Darstellungen, nämlich als (+)0 und −0. Das kann zu Problemen beim Test auf Gleichheit führen, wenn (+)0 und −0 nicht als gleich erkannt werden. Außerdem verliert der Zahlenraum eine Zahl. Mit acht Bit könnte man z.B. nur Zahlen zwischen −127 und 128 darstellen. Die Zahl −128 wäre nicht darstellbar.
Das Hauptproblem bei dieser Darstellung der ganzen Zahlen ist aber ein anderes: Die Addition zweier positiver Zahlen funktioniert. Das Vorzeichenbit ist in diesem Fall in beiden Summanden und im Ergebnis jeweils *0*. Für die Addition zweier negativer Zahlen muss aber ganz anders vorgegangen werden. Der Fall, dass eine positive und eine negative Zahl addiert werden, macht sogar zwei weitere Fälle erforderlich, im einen hat die positive Zahl den größeren Betrag und das Ergebnis ist positiv, das Vorzeichnbit muss in diesem Fall *0* sein, im anderen Fall besitzt die negative Zahl den größeren Betrag, dann ist das Ergebnis negativ und das Vorzeichenbit muss auf *1* gesetzt werden. Wegen dieser Probleme beim Rechnen ist diese Zahlendarstellung letztlich viel komplizierter als das Zweierkomplement und wird heute nicht mehr benutzt.

4.1.4 Gleitkommazahlen

Möglichkeiten zur Darstellung gebrochener Zahlen Möchte man Zahlen mit einem Betrag kleiner als *1* oder mit einem Anteil kleiner als *1* darstellen, dann gibt es drei Möglichkeiten:

- Darstellung als Bruch: $\frac{801}{5000000000000000000}$
- Darstellung als Festkommazahl: $0,00000000000000001602$
- Darstellung als Gleitkommazahl: $1{,}602 \cdot 10^{-19}$

In der Festkommazahl steht das Komma an einem festen Ort: Es trennt die Einerstelle von der nächst kleineren Stelle, im Dezimalsystem die Zehntelstelle. In der Gleitkommazahl legt der Exponent fest,

welche Stellen das Komma effektiv trennt. Er lässt diese Position *gleiten*. *Gleitkommazahl* heißt auf englisch *floating point number*. Beachten Sie, dass der Dezimaltrenner im angelsächsischen Sprachraum der Punkt ist, nicht das Komma wie im Deutschen. Da Programmiersysteme in der heutigen Form stark von US-amerikanischen Entwicklungen geprägt sind, ist der Dezimaltrenner in der Computerwelt üblicherweise der Punkt.

Betrachtet man die obigen Darstellungen, dann sieht man, dass die Gleitkommazahl die günstigste Variante ist, während man bei den anderen beiden Varianten viel Speicherplatz darauf verwenden muss, viele Nullen zu speichern. Daher werden gebrochene Zahlen gewöhnlich als Gleitkommazahlen dargestellt.

4.7: Exkurs: Bedeutung von Brüchen

In speziellen Situationen kann die Benutzung von Brüchen von Vorteil sein. Brüche mit Nennern, die andere Primfaktoren als *2* und *5*, da sind die Teiler der Basis *10* des Dezimalsystems, enthalten, können nicht exakt als Fest- oder Gleitkommazahl dargestellt werden, da sie unendlich lang sind. Beispiele: $\frac{1}{3} = 0,\overline{3} = 0,3333\ldots$ oder $\frac{1}{7} = 0,\overline{142857} = 0,142857142857142857\ldots$. Die Speicherung als Fest- oder Gleitkommazahl würde unendlich viel Speichers bedürfen. Die meisten Programmiersprachen stellen keine Typen für Brüche zur Verfügung. Man muss sie dann gegebenenfalls erstellen, wie wir das in Abschnitt 3.3.2 getan haben.

Gleitkommazahlen bieten auch Vorteile, sehr große, aber nicht exakt bekannte Zahlen darzustellen. Statt 299790000 kann man kürzer $2{,}9979 \cdot 10^8$ schreiben, wenn die Nullen in der ersten Darstellung keine Information beinhalten, da sie nur durch eine Rundung entstanden sind.

Die Darstellung einer Gleitkommazahl ist zunächst nicht eindeutig. $1{,}602 \cdot 10^{-19}$, $0{,}1602 \cdot 10^{-18}$ oder $1602 \cdot 10^{-22}$ stellen dieselbe Zahl dar. Um Uneindeutigkeiten zu vermeiden, arbeitet man mit *normierten* Gleitkommzahlen. Jede Gleitkommazahl hat genau eine Darstellung als normierte Gleitkommazahl. Eine Gleitkommazahl ist normiert, wenn sie die folgende Bedingung erfüllt: Die *Mantisse* der Gleitkommazahl, das ist der erste Teil vor der Potenz, hat genau eine Stelle vor dem Komma und diese ist nicht *0*. Von den obigen Darstellungen ist also $1{,}602 \cdot 10^{-19}$ die Darstellung als normierte Gleitkommzahl. $0{,}1602 \cdot 10^{-18}$ ist nicht normiert, weil die Vorkommastelle *0* ist. $1602 \cdot 10^{-22}$ ist nicht normiert, weil es mehrere stellen vor dem Komma gibt.

Gleitkommazahlen in Computersystemen Moderne Prozessoren arbeiten üblicherweise mit zwei Arten von Gleitkommazahlen: *float*, der einen Speicherbedarf von 32 bit=4 B und *double*, der den doppelten Speicherbedarf von 64 bit=8 B besitzt. Viele Programmiersprachen, auch Java, benutzen genau diese Typen, die dann ohne Umwandlung an den Prozessor gegeben werden können.

Die Zahl $-3,25$ wird als float abgespeichert als 11000000010100000000000000000000. Diese 32 bit zerlegt man in drei Gruppen der Länge 1, 8 und 23 also 1 10000000 10100000000000000000000. Wir nennen die erste Gruppe s, die zweite c und die dritte Gruppe a. In unserem Beispiel gilt also $s = 1$, $c = 10000000$ und $a = 10100000000000000000000$.

s legt das Vorzeichen der Zahl fest. *1* steht für $-$, *0* für $+$. Die Gruppe s muss daher genau ein bit lang sein, alles andere würde keinen Sinn ergeben. Die Verteilung der restlichen 31 Bit auf c und a ist durch die Definition von float festgelegt. Diese Aufteilung ist sinnvoll, hätte aber auch geringfügig anders sein können und besitzt damit eine gewisse Willkür.

a speichert die Nachkommastellen der Mantisse. Die Vorkommastelle der Mantisse wird nicht gespeichert. Im Dualsystem kommen nur die Ziffern *0* und *1* vor. Die einzige Vorkommastelle einer normierten Gleitkommazahl ist aber nicht *0*, daher muss sie *1* sein. Unsere Mantisse lautet also $1,10100000000000000000000$. Rechts neben dem Komma in der Dualzahl findet man die Stellenwerte $2^{-1}, 2^{-2}, 2^{-3}, 2^{-4}\ldots$, also $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$ .... Damit gilt $1,10100000000000000000000_2 = 1,625$.

c speichert den Exponenten. Dieser Exponent besitzt ein Vorzeichen. Man könnte ihn daher als Zweierkomplementzahl speichern. Man hat aber einen anderen Weg gewählt: c wird als vorzeichenlose, somit nicht negative Dualzahl interpretiert. Der Exponent der Gleitkommazahl berechnet sich als $c-b$, wobei b in der Definition von float zu $1111111_2 = 127$ festgelegt wurde. In unserem Fall ist $c = 10000000$ und damit $c - b = 10000000_2 - 1111111_2 = 1_2 = 1$. Die Basis der Potenz ist, wie in Computersytemen üblich, *2* und nicht *10*. Es gilt $2^1 = 2$, die Mantisse muss also mit *2* multipliziert werden.

Zusammengefasst: Das Vorzeichenbit $s = 1$ steht für eine negative Zahl. Die Nachkommastellen der Mantisse $a = 10100000000000000000000$ stehen für die Mantisse $1,625$ und der Exponent $c =$

10000000 für eine Multiplikation mit *2*. Somit steht die Zahl 11000000010100000000000000000000 für $-1,625 \cdot 2 = -3,5$.

Allgemein gilt für die Interpretation einer Gleitkommazahl, dass ihr Wert $(-1)^s \cdot 1, a_2 \cdot 2^{c-b}$ beträgt. Der Wert von b und die Aufteilung der Zeichenkette in s, c und a müssen in der Definition der Gleitkommazahl festgelegt sein.

Der Datentyp *double* ist wie folgt festgelegt: das erste der 64 Bit ist das Vorzeichenbit s. Es folgen elf Bits für c. Zusammen mit dem festgelegten Wert für $b = 1111111111_2 = 1023$ kann die Potenz 2^{c-b} berechnet werden. Die restlichen 51 Bits sind die Nachkommastellen der Mantisse.

Wegen der größeren Anzahl der Nachkommastellen der Mantisse können mit *double* feinere Unterteilungen erreicht werden als mit *float*. Der Datentyp *double* ist also im Allgemeinen genauer als *float*. Wegen des größeren Bereichs für den Exponenten können mit *double* größere Zahlen als mit *float* dargestellt werden, außerdem können mit *double* von *0* verschiedene Zahlen, die näher an *0* liegen, dargestellt werden, als das mit *float* möglich ist.

Der minimale und der maximale Wert von c signalsieren bei einer Gleitkommazahl eine besondere Interpretation. Man benötigt diese besondere Interpretationen für die Darstellung der Zahl *0*, für die Darstellung von $+\infty$ und $-\infty$ und *NaN* („Not a Number“) bei einer fehlerhaften Operation, z.B. dem Versuch, die Quadratwurzel aus einer negativen Zahl zu berechnen.

4.8: Aufgabe: Größte Zahl mit *float* und *double*

Ermittlen Sie, was doe jeweils größte Zahl in den Formaten *float* und *double* ist. Schreiben Sie sowohl die Kette aus Einsen und Nullen auf, als auch die Bedeutung als Gleitkommazahl im Dezimalsystem.

Rechnen mit gebrochenen Zahlen im Dualsystem Um Gleitkommazahlen in Formaten wie *float* oder *double* berechnen zu können, muss man im Dezimalsystem gegebene Kommazahlen ins Dualsystem übertragen können. Für den ganzzahligen Anteil ist das klar. Für den Anteil kleiner als *1* kann man wieder prüfen, ob der nächste Stellenwert passt oder nicht ihn dann auf *1* bzw. *0* setzen. Beispiel: $0,625$ soll als Dualzahl wiedergegeben werden. $\frac{1}{2}$ passt. In den dann verbleibenden Rest von $0,125$ passt $\frac{1}{4}$ nicht, aber $\frac{1}{8}$ passt. Der Rest ist dann *0*. Somit gilt $0,625 = 0,101_2$.

Man kann auch den Nachkommateil mit zwei multiplizieren. Sobald eine *1* vor das Komma rutscht, notiert man eine *1*. Rutscht keine *1* vor das Komma, dann notiert man eine *0*. Beispiel: $0,625 \cdot 2 = 1,25$, notiere also *1* und fahre fort mit $0,25 \cdot 2 = 0,5$. Hier entsteht keine *1* vor de Komma, notiere also *0*. $0,5 \cdot 2 = 1$, notiere also eine *1*. Da es keine Nachkommasellen mehr gibt, bricht das Verfahren an dieser Stelle ab und wir haben $0,101_2$ erhalten.

4.8: Exkurs: Periodische Dualzahlen

Alle periodischen Dezimalzahlen ergeben als Dualzahlen ebenfalls periodische Dualzahlen. Z.B. gilt $\frac{1}{3} = 0,\overline{01}_2 = 0,010101\ldots_2$. Nur Brüche mit einem Nenner, der eine Zweierpotenz darstellt, ergeben endliche Dualzahlen. Somit ergibt auch $\frac{2}{5} = 0,4 = 0,\overline{0110}_2 = 0,011001100110\ldots_2$ eine periodische Dualzahl.

4.9: Aufgabe: Umrechnungen mit floats

1. Üben Sie mit dieser Übung das Umrechnen von floats ins Dezimalzahlen.
2. Üben Sie mit dieser Übung das Umrechnen vom Dezimalsystem in floats.

4.9: Exkurs: Gleitkommazahlen und Prozessoren

Das Rechnen mit Gleitkommazahlen ist beträchtlich aufwändiger als das Rechnen mit ganzen Zahlen, da die Zahldarstellungen in ihre Teile zerlegt und interpretiert werden müssen und für die Rechnungen viele Fallunterscheidungen notwendig sind.
Graphikprozesoren müssen viele Gleitkommaoperationen ausführen und sind daher auf diese Aufgabe optimiert. Sie können außerdem in hohem Maße parallel betrieben werden, das heißt, sie können mehrere Berechnungen, die unabhängig voneinander sind, gleichzeitig berechnen.
Dieselben Anforderungen bestehen auch, wenn man künstliche neuronale Netze für das maschinelle Lernen betreiben möchte. Der Boom der künstlichen Intelligenz der letzten Jahre hat daher Graphikprozessoren sehr verteuert und die Aktienkurse der Firmen, die diese herstellen, in große Höhen getrieben.

4.2 Boolesche Algebra uund Schaltnetze

4.2.1 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 nennt man *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 $D \subseteq \mathbb{R}^n$, zum Beispiel
$f : \mathbb{R}^2 \to \mathbb{R} \quad (\mathrm{x}, \mathrm{y}) \mapsto \sqrt{x^2 + y^2}$ 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 4.6 zu finden.

Tabelle 4.6: Alle Booleschen Funktionen einer Variablen

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

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 interessant. 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 $F_1 = \overline{x}$ gelesen *nicht x*.

4.2.2 Boolesche Funktionen von zwei Variablen

Es gibt 16 Funktionen F_i zweier Variablen X_0 und X_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 $F_3 = \overline{X}_1$. Genauso sind die Funktionen F_{10} und F_5 uninteressant, da sie in gleicher Weise nur von X_0 abhängen. Es bleiben zehn Funktionen, die eine nähere Untersuchung verdienen. Von diesen sind fünf die Negationen der anderen fünf.

[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 4.7: Alle Booleschen Funktionen von zwei Variablen

$\mathbf{X_1}$	$\mathbf{X_0}$	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

F_{14} ist die ODER-Funktion, geschrieben $F_{14} = X_0 \vee X_1$ oder $F_{14} = X_0 + X_1$. Die Funktion heißt *ODER-Funktion*, weil die Variable X_0 **oder** X_1 gleich 1 sein muss, damit der Funktionswert 1 ist.

F_8 ist die UND-Funktion, geschrieben $F_8 = X_0 \wedge X_1$ oder $F_8 = X_0 \cdot X_1$ oder kurz $F_8 = X_0X_1$. Die Funktion heißt *UND-Funktion*, weil beide Variablen F_0 **und** F_1 gleich 1 sein müssen, damit der Funktionswert 1 ist.

4.10: Aufgabe: 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 $X_0 \leftrightarrow X_1$. Sie ist genau dann 1, wenn beide Variablen denselben Wert haben. F_{11} ist die Funktion X_1 IMPLIZIERT X_0, geschrieben $X_1 \rightarrow X_0$. Wenn X_1 gleich 1 ist, dann muss auch X_0 gleich 1 sein, damit der Funktionswert 1 ist. Für den Fall $X_1 = 1$ und $X_0 = 0$ ist der Funktionswert 0. F_{13} ist die Funktion X_0 IMPLIZIERT X_1, geschrieben $X_0 \rightarrow X_1$.

4.11: Aufgabe: Negationen verschiedener Funktionen

1. Finden Sie die Negationen zu den Funktionen F_{11} und F_{13}.
2. Finden Sie die Antivalenzfunktion $X_1 \nleftrightarrow X_2$. Diese Funktion ist die Negation der Äquivalenzfunktion F_9.
3. Merkregel: Bei den Booleschen Funktionen von zwei Variablen gilt immer: Die Nummer einer Funktion plus die Nummer ihrer Negationsfunktion gibt immer 15. Überprüfen Sie die Regel anhand der Tabelle 4.7.

4.2.3 Gatter

Elektronische Bauteile, die Boolesche Funktionen realisieren, nennt man *Gatter*. Abbildung 4.3 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 4.4 steht für einen Inverter, auch Negationsglied genannt. Er realisiert die Negation. Abbildung 4.5 steht für ein ODER-Gatter. Das Größer-oder-gleich-Zeichen beim ODER-

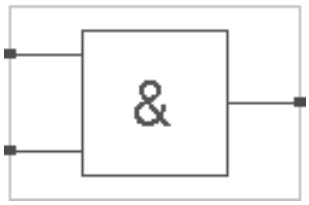


Abbildung 4.3: UND-Gatter

Editor mode. 

Abbildung 4.4: Inverter

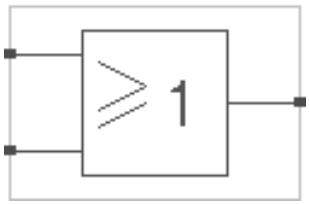


Abbildung 4.5: ODER-Gatter

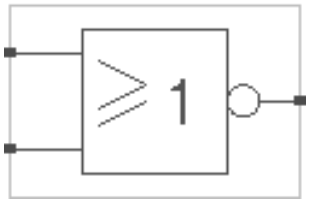


Abbildung 4.6: NOR-Gatter

Gatter soll ausdrücken, dass dieses Gatter am Ausgang dann 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 4.6 zeigt das Symbol des NOR-Gatters. Der Kreis am Ausgang steht für die Negation. Das Symbol des NAND-Gatters (Abbildung 4.7) sieht entsprechend aus.

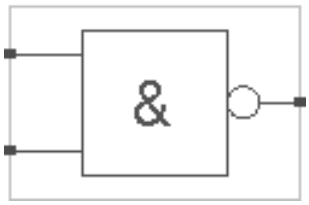


Abbildung 4.7: NAND-Gatter

Das Symbol eines Äquivalenzgatters ist in Abbildung 4.8 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 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 4.9 zu sehen. *=1* soll dabei ausdrücken, dass der Ausgang genau dann *1* ist, wenn die Summe der Eingangssignale 1 ist.

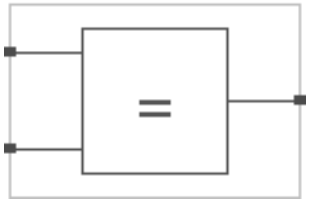


Abbildung 4.8: Äquivalenz-Gatter

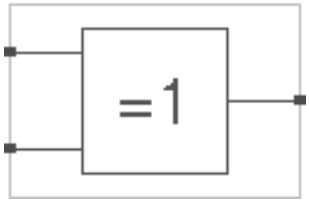


Abbildung 4.9: XOR

4.12: Aufgabe: 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 (von denselben Seiten). 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.

4.2.4 Vollständige Operatorensysteme

4.13: Aufgabe: 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 4.10 zeigt eine Schaltung aus einem Negationsglied (Inverter) und einem ODER-Gatter. Diese Schaltung realisiert die Funktion F_{11}, X_1 IMPLIZIERT X_0. Die Elemente X_0 und X_1 sind Schalter in Hades, die die Eingänge repräsentieren, das Element F_{11} ist eine LED, die den Ausgang der Schaltung repräsentiert.

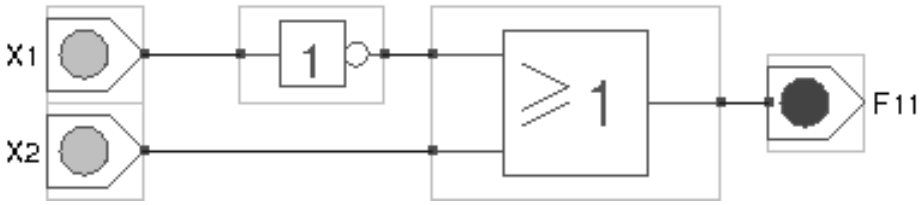


Abbildung 4.10: F_{11} aus NICHT und ODER

4.14: Aufgabe: 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_0 IMPLIZIERT X_1), F_4 (Negation von X_1 IMPLIZIERT X_0), 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. Einfacher geht es aber so: Wir wissen bereits, dass das UND-ODER-NICHT-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.

4.15: Aufgabe: Vollständigkeit des NAND-Systems

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

4.16: Aufgabe: Vollständigkeit des NOR-Systems

1. Bauen Sie in Hades eine Negationsschaltung, eine UND-Schaltung und eine ODER-Schaltung auf und verwenden Sie dafür nur NOR-Gatter.
2. Diskutieren Sie, wie man, nach obiger Vorarbeit, noch einfacher die Vollständigkeit des NOR-Systems zeigen kann und bauen Sie die entsprechende Schaltung in Hades.

4.2.5 Umformungen Boolescher Ausdrücke

Für die folgenden Ausführungen bezeichne + den ODER-Operator, · den UND-Operator und der Überstrich die Negation, wie bereits in Abschnitt 4.2.2 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. $A \cdot \overline{A} = 0$
12. $A + \overline{A} = 1$

Doppelte Negation

13. $\overline{\overline{A}} = A$

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

De Morgansche Regeln

16. $\overline{A \cdot B} = \overline{A} + \overline{B}$
17. $\overline{(A + B)} = \overline{A} \cdot \overline{B}$

Verallgemeinerte De Morgansche Regeln

18. $\overline{\prod_{i=1}^{n} A_i} = \sum_{i=1}^{n} \overline{A}_i$
19. $\overline{\sum_{i=1}^{n} A_i} = \prod_{i=1}^{n} \overline{A}_i$

Die Regeln 1 bis 17 können durch die entsprechenden Wahrheitstafeln bewiesen werden, wenn man die Wahrheitstafeln für die UND-, ODER- und Negation 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 4.8 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

Tabelle 4.8: Beweis von Regel 16

Variablenbelegung		linke Seite		rechte Seite		
			Auswertung linke Seite			Auswertung rechte Seite
A	B	A·B	$\overline{A \cdot B}$	$\overline{A}$	$\overline{B}$	$\overline{A} + \overline{B}$
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

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 4.10 wurde gezeigt, dass gilt $A \to B = \overline{A} + B$. Außerdem gilt für die Äquivalenzfunktion $A \leftrightarrow B = AB + \overline{A}\ \overline{B}$ und für die Antivalenzfunktion $A \nleftrightarrow B = \overline{A}B + A\overline{B}$.

Lernen Sie die Regeln nicht stumpf auswendig. Das Können der Regeln kommt mit ihrer Anwendung. 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!

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.

$$
\begin{aligned}
& F(A,B,C) \\
&= (A+B)(C+\overline{A})(C+\overline{C})B + \overline{\overline{C}}\ \overline{C} \\
&\overset{13}{=} (A+B)(C+\overline{A})(C+\overline{C})B + C\overline{C} \\
&\overset{11}{=} (A+B)(C+\overline{A})(C+\overline{C})B + 0 \\
&\overset{2}{=} 0 + (A+B)(C+\overline{A})(C+\overline{C})B \\
&\overset{10}{=} (A+B)(C+\overline{A})(C+\overline{C})B \\
&\overset{12}{=} (A+B)(C+\overline{A}) \cdot 1 \cdot B \\
&\overset{1}{=} 1 \cdot (A+B)(C+\overline{A})B \\
&\overset{9}{=} (A+B)(C+\overline{A})B \\
&\overset{1}{=} B(A+B)(C+\overline{A}) \\
&\overset{7}{=} (BA+BB)(C+\overline{A}) \\
&\overset{3}{=} (BA+B)(C+\overline{A}) \\
&\overset{9}{=} (BA+1B)(C+\overline{A}) \\
&\overset{1}{=} (BA+B1)(C+\overline{A}) \\
&\overset{7}{=} B(A+1)(C+\overline{A}) \\
&\overset{2}{=} B(1+A)(C+\overline{A}) \\
&\overset{15}{=} B \cdot 1(C+\overline{A}) \\
&\overset{1}{=} 1 \cdot B(C+\overline{A})
\end{aligned}
$$

$\stackrel{9}{=} B(C + \overline{A})$
$\stackrel{7}{=} BC + B\overline{A}$
$\stackrel{1}{=} BC + \overline{A}B$
$\stackrel{2}{=} \overline{A}B + BC$

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.

4.2.6 Kanonische Normalformen

Bisher haben wir Boolesche Funktionen durch einen Funktionsterm angegeben und dann gegebenenfalls die Wahrheitstafel der Funktion betrachtet. Jetzt soll es darum gehen, zu einer gegebenen Wahrheitstafel einen Funktionsterm anzugeben.

Gegeben sei die folgende Wahrheitstafel:

Tabelle 4.9: Beispielfunktion für die Normalformen

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

Um sicher zu stellen, dass bei der Variablenbelegung $A = 0, B = 0, C = 0$ die Funktion den Wert 1 annimmt, kann man den Term $\overline{A}\ \overline{B}\ \overline{C}$ in mit einer ODER-Verknüpfung aufnehmen. Genauso kann man für die zwei weiteren Variablenbelegungen, bei denen die Funktion den Wert 1 annimmt vorgehen und für diese die Terme $\overline{A}\ \overline{B}\ C$ bzw. $A\ B\ C$ aufnehmen. Letztlich erhält man die den Funktionsterm $F = \overline{A}\ \overline{B}\ \overline{C} + \overline{A}\ \overline{B}\ C + A\ B\ C$. Man kann leicht überprüfen, dass dieser Funktionsterm genau die Wahrheitstafel aus der Tabelle 4.9 liefert.

Im Zusammenhang mit dieser mit dieser Art der Darstellung ist es hilfreich, die folgenden Begriffe einzuführen:

- Der Funktionsterm besteht aus ODER-verknüpften UND-Termen. Die ODER-Verknüpfung nennt man auch *Disjunktion*, die UND-Verknüpfung *Konjunktion*, daher kann man sagen, dieser Funktionsterm ist eine Disjunktion von Konjunktionstermen. Die Darstellung einer Funktion als Disjunktion von Konjunktionstermen nennt man *disjunktive Normalform*, kurz *DNF*.
- Alle Konjunktionsterme in dieser Darstellung enthalten alle drei Variablen A, B und C. Solche Konjunktionsterme maximaler Länge nennt man *Minterme*. *Min* bedeutet in diesem Zusammenhang, dass ein solcher Term nur einen Wert 1 in der Wahrheitstafel erzeugt.
- Eine DNF, die ausschließlich aus Mintermen besteht, nennt man *kanonische disjunktive Normalform*, kurz *KDNF*.

4.17: Aufgabe: Minterme

Geben Sie an, welche der Konjunktionsterme in der Funktion $F = \overline{A}\ \overline{B} + ABC$ Minterme sind.

4.18: Aufgabe: DNF, KDNF

Diskutieren Sie, ob die Darstellung $F = \overline{A}\ \overline{B} + ABC$ eine DNF ist und ob es eine KDNF ist.

4.19: Aufgabe: Eindeutigkeit von DNFs

Diskutieren Sie, ob die Funktion $F = \overline{A}\ \overline{B} + ABC$ mit der Funktion aus Beispiel 4.9 übereinstimmt. Diskutieren Sie, inwieweit die Darstellung als DNF bzw. als KDNF eindeutig sind.

Man könnte die Funktion aus Beispiel 4.9 auch ganz anders durch einen Booleschen Term darstellen: Statt zu betrachten, bei welchen Variablenbelegungen die Funktion den Wert 1 annehmen muss, kann man auch schauen, bei welchen Variablenbelegungen die Funktion den Wert 0 annehmen muss. Dies ist z.B. bei der Variablenbelegung $\overline{A}\ B\ \overline{C}$ der Fall. Der Disjunktionsterm $A + \overline{B} + C$ nimmt bei dieser Belegung den Wert 0 an, bei allen anderen Belegungen den Wert 1. Für alle weiteren Nullen in der Wahrheitstafel findet man analoge Disjunktionsterme. Verknüpft man diese konjunktiv, dann erhält man ein Darstellung der Funktion als Disjunktion von Konjunktionstermen, konkret $F = (A + \overline{B} + C)(A + \overline{B} + \overline{C})(\overline{A} + B + C)(\overline{A} + B + \overline{C})(\overline{A} + \overline{B} + C)$. Analog zu den Definitionen in Zusammenhang mit der KDNF sind folgende Begriffe hilfreich:

- Der zuletzt gegebene Funktionsterm ist eine Konjunktion von Disjunktionstermen. Die Darstellung einer Funktion als Konjunktion von Disjunktionstermen nennt man *konjunktive Normalform*, kurz *KNF*.
- Alle Disjunktionsterme in dieser Darstellung enthalten alle drei Variablen A, B und C. Solche Disjunktionsterme maximaler Länge nennt man *Maxterme*. *Max* bedeutet in diesem Zusammenhang, dass ein solcher Term viele Einsen und nur eine Null in der Wahrheitstafel erzeugt.
- Eine KNF, die ausschließlich aus Maxtermen besteht, nennt man *kanonische konjunktive Normalform*, kurz *KKNF*.

4.20: Aufgabe: Maxterme und KNFs

1. Geben Sie einige Beispiele an für Maxterme und für Disjunktionsterme, die keine Maxterme sind. Gehen Sie dabei von Funktionen dreier Variablen A, B und C aus.
2. Geben Sie für die Beispielfunktion mindestens eine Darstellung als KNF an, die keine KKNF ist und diskutieren Sie die Eindeutigkeit der Darstellung einer Funktion als KNF und als KKNF.

4.21: Aufgabe: 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. Diskutieren Sie, von was die Lägen der kanonischen Normalformen abhängen.

Von einer Funktion gibt es im allgemeinen mehrere Darstellungen als DNF und mehrere Darstellungen als KNF. Von einer Funktion gibt es aber genau eine Darstellung als KDNF und genau eine Darstellung als KKNF, wenn man von Vertauschungen der einzelnen Terme und Vertauschungen der Variablen in den Termen absieht. Die Darstellungen einer Funktion als KDNF oder als KKNF sind in diesem Sinne eindeutig. Daher sind diese Darstellungen gut geeignet, zwei Funktionen miteinander zu vergleichen, um gegebenenfalls deren Gleichheit festzustellen. Wenn möglichst kurze Darstellungen einer Funktion gesucht sind, sind die kanonischen Normalformen in aller Regel aber eine schlechte Wahl.

Man kann KDNF und KKNF einer Funktion mit Hilfe des Inversionssatz von Shannon oder der Anwendung der beiden DeMorganschen Regeln ineinander umrechnen. Da die Darstellung einer Funktion als KDNF eher im Einklang mit dem menschlichen Denken ist als die Darstellung als KKNF, kann man das ausnutzen, um die KKNF einer Funktion aus der KDNF zu erstellen.

Alternative 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 4.10 hier noch einmal gegeben, ergänzt um die Spalte $\overline{F}$, der Negation von F.

Tabelle 4.10: Beispielfunktion für die KKNF

A	B	C	F	$\overline{F}$
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

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

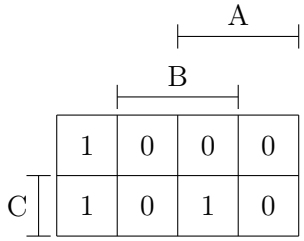
4.2.7 Minimale Normalformen und Karnaugh-Diagramme

Kanonische Normalformen sind gut geeignet, um die Gleichheit zweier Boolescher Ausdrücke zu überprüfen. Da sie aber sehr lang sind, sind sie 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 4.9 aus. Als KDNF wurde gefunden $F = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}C + ABC$. 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 $F = (\overline{A}\,\overline{B})(\overline{C} + C) + ABC$. Mit den Regeln 12, 1 und 9 kommt man schließlich auf $F = \overline{A}\,\overline{B} + ABC$. 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:

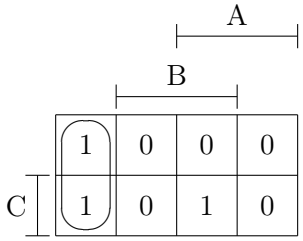


Die rechten vier Kästchen stehen für die Terme mit A, die linken vier Kästchen für die Terme mit $\overline{A}$. Die unteren vier Kästchen stehen für die Terme mit B, die oberen vier Kästchen für die Terme mit $\overline{B}$. Die mittleren vier Kästchen stehen für die Terme mit C, die äußeren vier Kästchen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

4.22: Aufgabe: Minterme im Karnaugh-Diagramm

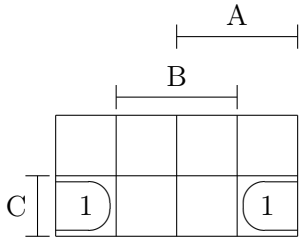
Man kann sagen, im Karnaugh-Diagramm steht jedes Kästchen für einen Minterm. Finden Sie jedes Kästchen zu allen Mintermen
$\overline{A}\ \overline{B}\ \overline{C}, \overline{A}\ \overline{B}C, \overline{A}B\overline{C}, \overline{A}BC, A\overline{B}\ \overline{C}, A\overline{B}C, AB\overline{C}, ABC$

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:

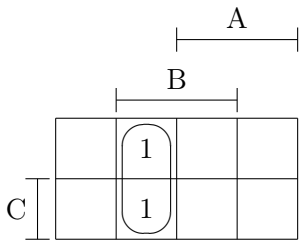


In dieser Zweiergruppe gilt $\overline{A}$ und $\overline{B}$, während die Variable C einmal negiert und einmal ohne Negation auftaucht. Die Variable C fällt daher in der Beschreibung dieser Zweiergruppe weg. Diese Zweiergruppe repräsentiert den Term $\overline{A}\ \overline{B}$. In der minimalen DNF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

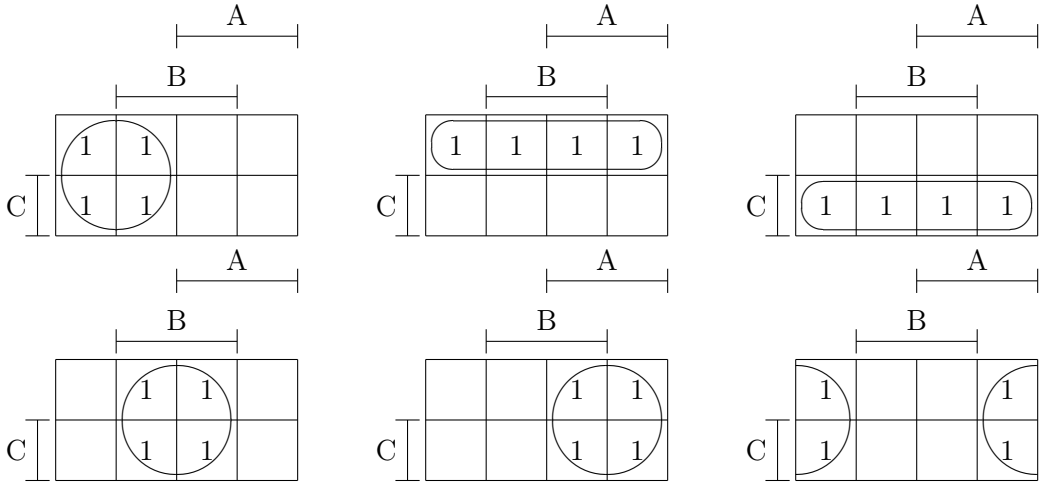
Bei den Karnaugh-Diagrammen von drei Variablen muss man sich den linken mit dem rechten Rand verbunden vorstellen. Man müsste das Karnaugh-Diagramm also anstatt auf ein flaches Papier auf einen Zylindermantel, umgangssprachlich eine *Röhre* aufzeichnen. Daher ist auch eine solche Zweiergruppe erlaubt,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



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



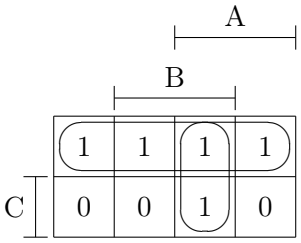
Statt Zweiergruppen kann man auch Vierergruppen finden. Die Vierergruppen repräsentieren Terme aus nur einer Variablen, also $A, B, C, \overline{A}, \overline{B}$, oder $\overline{C}$. Die Vierergruppen sehen so aus:



4.23: Aufgabe: Terme einer Variablen bei Funktionen von drei Variablen

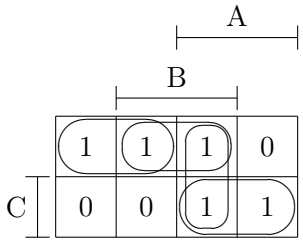
Ordnen Sie die Vierergruppen den richtigen Termen aus einer Variablen zu.

Die minimale DNF der Funktion, die durch dieses



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

Das folgende Beispiel weist eine zusätzliche Schwierigkeit auf.

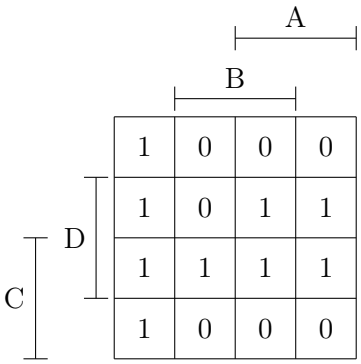


Man kann keine Vierergruppe finden. Man findet vier Zweiergruppen, die die Terme $\overline{A}\ \overline{C}$, $B\overline{C}$, AB und AC repräsentieren. Die DNF $F = \overline{A}\ \overline{C} + B\overline{C} + AB + AC$ ist aber nicht minimal, denn die beiden Einsen des Terms $B\overline{C}$ stecken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

Genauso gut kann man aber auch den Term AC weglassen und muss dann den Term $B\overline{C}$ aufnehmen. Daraus ergibt sich die DNF $F = \overline{A}\ \overline{C} + B\overline{C} + AC$. 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:



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 (umgangssprachlich ein *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.

4.24: Aufgabe: 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 das untere Teildiagramm für die fünfte Variable E, das obere für diese Variable negiert, $\overline{E}$.

Ein Karnaugh-Diagramm für sechs Variablen besteht aus vier solchen Teildiagrammen. 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 zurückgreifen, 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.

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

4.2.8 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.

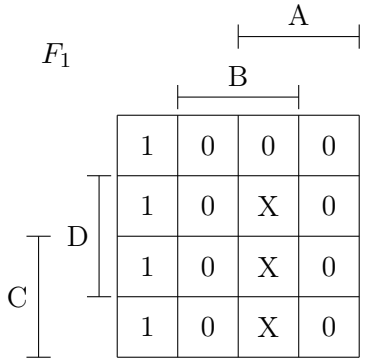
Tabelle 4.11: Fußgängerampel in Abhängigkeit des Takts

Takt	A	B	C	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 4.11 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 X eingetragen.



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, egal*. 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 $F_1 = \overline{B}\ \overline{A}$.

Beim Karnaugh-Diagramm von F_2 ist es günstiger, alle X als *1* aufzufassen. Man erhält dann $F_2 = A + B$.

F_2

C \ D \ A B	00	01	11	10
00	0	1	1	1
01	0	1	X	1
10	0	1	X	1
11	0	1	X	1

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 $F_1 = \overline{F_2}$. Damit kann man, wenn man F_2 hat, F_1 mit einem einzigen zusätzlichen Schaltelement, nämlich einem Inverter, erhalten.

4.26: Aufgabe: Ampel

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

4.2.9 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 4.12: Wertetabelle eines Halbaddierers

1. Summand A_0	2. Summand B_0	Übertrag $\ddot{U}_0$	Stelle S_0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Zur Addition zweier Dualzahlen 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 $1+1 = 10$. 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 $\ddot{U}$ für den Übertrag heißen. Die Wertetabelle der beiden Funktionen ist in Tabelle 4.12 angegeben. Der Index 0 soll dabei bedeuten, dass es sich um die Stelle mit dem Wert 2^0, also um die Einerstelle, handelt. Es muss untersucht werden, welche Funktionen das sind. Die Einerstelle der Summe, 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 4.11 gezeigt.

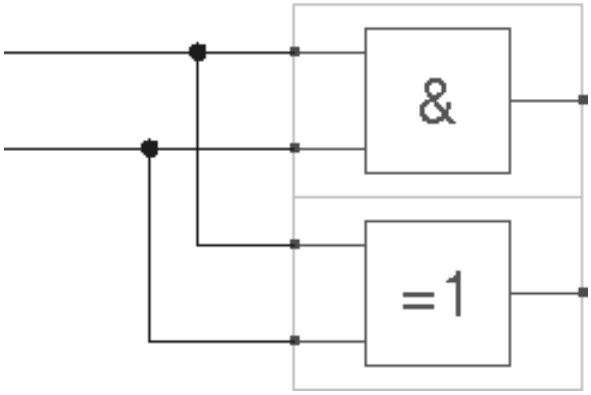


Abbildung 4.11: Halbaddierer

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,

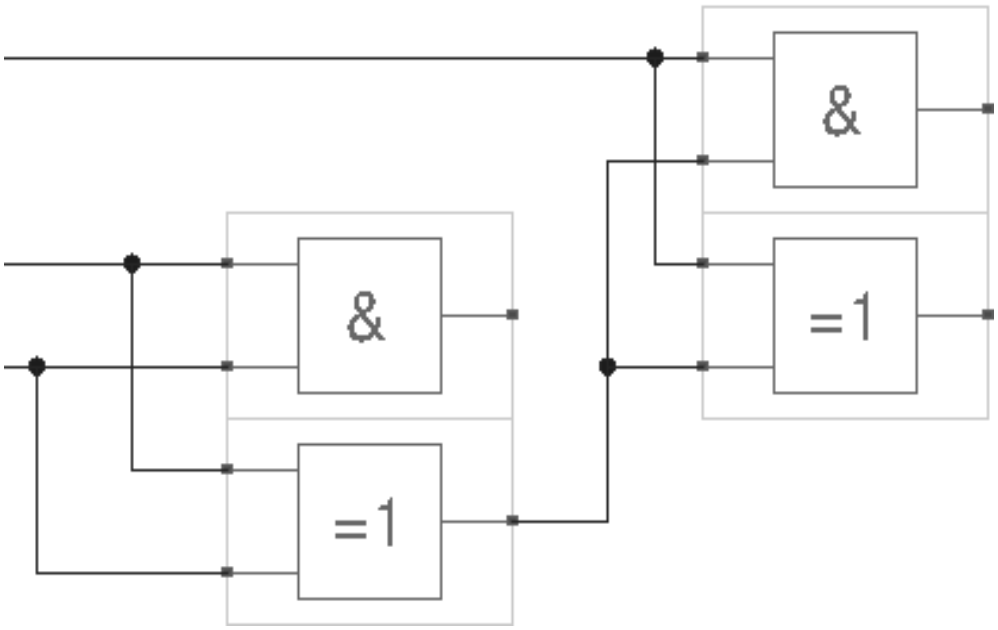


Abbildung 4.12: Volladierer, unfertig

was ja auch irgendwie logisch 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 4.12.

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 die Stelle und ein Ausgang Ü für den Übertrag in die nächste Stelle. In der Tabelle 4.13 sind die Funktionen Ü und S des Volladdierers angegeben.

Tabelle 4.13: Wertetabelle eines Volladdierers

1. Summand A_n	2. Summand B_n	eingehender Übertrag $\ddot{U}_{n-1}$	ausgehender Übertrag $\ddot{U}_n$	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

4.27: Aufgabe: Stelle S und Übertrag Ü als Boolesche Funktionen

1. Wie lässt sich aus A_n, B_n und $\ddot{U}_{n-1}$ die Funktion S_n gewinnen?
2. Wie lässt sich aus A_n, B_n und $\ddot{U}_{n-1}$ die Funktion $\ddot{U}_n$ gewinnen?

Erst weiterlesen, wenn Sie versucht haben, die Aufgabe zu lösen! Mit dieser Lösung sieht ein Volladdierer aus wie in Abbildung 4.13 gezeigt.

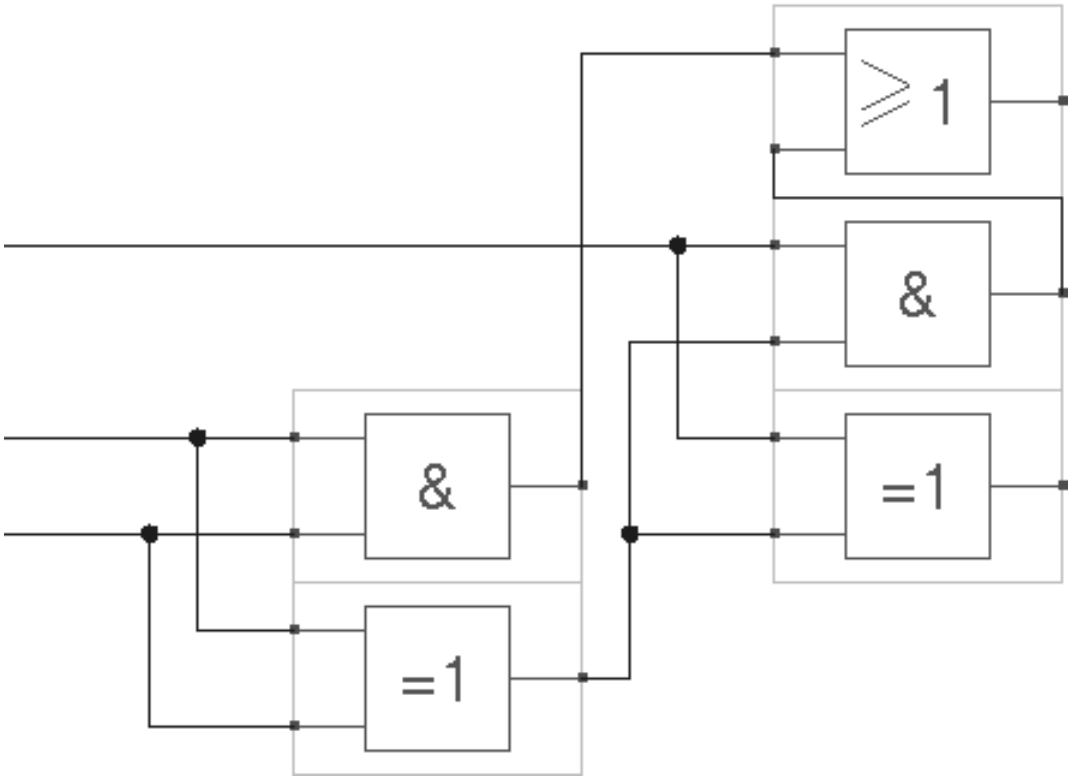


Abbildung 4.13: Volladierer

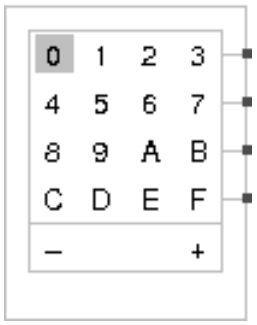


Abbildung 4.14: Hex-Switch

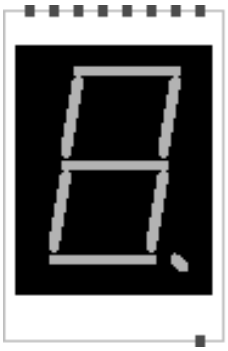


Abbildung 4.15: Hex-Display

4.28: Aufgabe: Addierwerk, eine Stelle

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 4.14). Sie brauchen nur den unteren Anschluss, der *0* liefert, wenn *0* gewählt ist und *1*, wenn *1* gewählt ist. Benutzen Sie, um das Ergebnis darzustellen, ein Exemplar von *Hex-Display* (siehe Abbildung 4.15). 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.

4.29: Aufgabe: Addierwerk mit mehreren Stellen

1. 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?
2. Erweitern Sie das Addierwerk für zwei Summanden mit je drei dualen Stellen.
3. 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.

5 Rekursion

5.1 Grundlegende Beispiele für Rekursion

Definition: Direkte und indirekte Rekursion Eine *rekursive* Methode ist eine Methode, die sich selbst aufruft. Das kann sie entweder direkt machen, dann spricht man von *direkter* Rekursion. Es kann aber auch eine Methode *methode1()* eine Methode *methode2()* aufrufen, die wiederum *methode1()* aufruft, so dass sich die beiden Methoden mittelbar selbst aufrufen. In diesem Fall spricht man von *indirekter* Rekursion. Indirekte Rekursion kann auch über mehr als zwei Methoden laufen.

Ein ganz schlechtes Beispiel zum Einstieg Die folgende Methode ist direkt rekursiv:

Listing 5.1: Rekursion: Ein ganz schlechtes Beispiel

```
1 private static void dummeMethode() {
2     System.out.println("Ich bin eine dumme Methode!");
3     dummeMethode();
4 }
```

Die Methode führt die Ausgabeanweisung aus und ruft sich dann selbst auf. Dabei führt sie abermals die Ausgabeanweisung aus und ruft sich dann selbst aus, und so weiter. Auf diese Weise müsste die Ausgabe unendlich oft getätigt werden. In der Praxis kommt die Ausgabe sehr oft, dann aber bricht das Programm mit einem *StackOverflowError* ab. Das hat folgenden Grund: Wird eine Methode aufgerufen, dann muss sich der Prozess, der das Programm ausführt, merken, an welcher Stelle das Programm nach dem Abarbeiten der Methode in der aufrufenden Methode fortgesetzt werden muss. Wird eine in der aufgerufenen Methode eine weitere Methode aufgerufen, dann muss sich der Prozess eine weitere Rücksprungadresse merken. Die Liste der Rücksprungadressen nennt man *Stapel* oder englisch *Stack*. In unsrem Beispiel wird der Stack immer größer und müsste theoretisch unendlich groß werden. Sobald der Stack größer wird als der Speicherplatz, den der Computer dem Programm zur Verfügung stellt, kommt es zum Programmabbruch mit dem StackOverflowError.

Man könnte das obige Programm verbessern:

Listing 5.2: Rekursion: Ein verbessertes schlechtes Beispiel

```
1 private static void dummeMethode(int n) {
2     System.out.println("Ich bin eine dumme Methode!");
3     if(n>1) {
4         dummeMethode(n-1);
5     }
6 }
```

Ruft man diese Methode mit dem aktuellen Parameter *5* auf, dann wird die Ausgabeanweisung fünf mal ausgeführt. Im Methodenaufruf mit der fünften Ausgabe hat der Parameter *n* den Wert *1*, daher erfolgt kein weiterer Aufruf.

Die n-malige Wiederholung einer Anweisung erreicht man einfacher – für den oder die Programmierer:in genauso wie den Computer – durch eine Schleife. Die Rekursion ist hier nicht sinnvoll. Wir sehen aber an diesem einfachen Beispiel eine funktionierende Rekursion. Weiter erkennt man an diesem Beispiel: Der rekursive Aufruf darf nicht auf alle Fälle stattfinden wie in Listing 5.1, sondern muss beispielsweise in einer bedingten Anweisung stehen wie in Listing 5.2 stehen. Statt in einer bedingten Anweisung kann der rekursive Aufruf auch im *if*- oder *else*-Zweig einer Verzweigung stehen oder in einer Schleife. Bei einer indirekten Rekursion muss das mindestens für einen der rekursiven Aufrufe gelten.

Ein besseres Beispiel: Fiboonacci-Folge Die nullte Fibonacci-Zahl ist *0*, die erste Fibonacchi-Zahl ist *1*. Alle weiteren Fibonacci-Zahlen werden berechnet als die Summe ihrer beiden Vorgänger. Es gilt also

0. Fibonacci-Zahl: 0 (durch Festlegung)
1. Fibonacci-Zahl: 1 (durch Festlegung)
2. Fibonacci-Zahl: $0 + 1 = 1$
3. Fibonacci-Zahl: $1 + 1 = 2$
4. Fibonacci-Zahl: $1 + 2 = 3$
5. Fibonacci-Zahl: $2 + 3 = 5$
6. Fibonacci-Zahl: $3 + 5 = 8$
7. Fibonacci-Zahl: $5 + 8 = 13$
8. Fibonacci-Zahl: $8 + 13 = 21$
9. Fibonacci-Zahl: $13 + 21 = 34$
10. Fibonacci-Zahl: $21 + 34 = 55$
11. Fibonacci-Zahl: $34 + 55 = 89$

Eine Methode zur Berechnung der n-ten Fibonacci-Zahl könnte so aussehen:

Listing 5.3: Methode zur Berechnung der Fibonacci-Zahlen

```
1  private static int fibonnacciBerechnen(int n) {
2      if (n==0) {
3          return 0;
4      }
5      if (n==1) {
6          return 1;
7      }
8      return fibonnacciBeechnen(n−1)+fibonnacciBerechnen(n−2);
9  }
```

5.1: Aufgabe: Fibonacci-Zahlen

1. Erklären Sie, wie die 5. Fibonacci-Zahl berechnet wird. Erklären Sie, wie oft die Methode dabei aufgerufen wird.
2. Schreiben Sie eine Methode, die die Fibonacci-Zahlen berechnet. Benutzen Sie keine Rekursion, sondern eine Schleife. Beurteilen Sie, welche Methode, die rekursive oder die iterative (d.h. die mmit der Schleife), einfacher zu verstehen ist und welche kürzer ist.

5.1: Exkurs: Fibonacchi-Zahlen direkt berechnen

Die Fibonacci-Zahlen sind leichter mit einer Rekursion als mit einer Schleife berechenbar, da die Folge rekursiv definiert ist: Um die 20. Fibonacci-Zahl zu berechnen, muss man zuerst die 19. und die 18. berechnen, dafür die 17. und die 16. und so weiter.
Man kann die n-te Fibonacci-Zahl auch direkt berechnen und zwar mit der folgenden Formel:
$fib(n) = \frac{1}{\sqrt{5}}[(\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n]$. Die Richtigkeit der Formel kann man mit vollständiger Induktion beweisen. Ein anderer Beweis nutzt Eigenwerte und Eigenwerte von Matrizen. Dieser Beweis ist schöner, weil er konstruktiv ist, d.h. die Formel findet, anstatt nur ihre Richtigkeit zu beweisen, nachdem sie gegeben wurde. Die notwendige Mathematik lernt man in den meisten technischen Fächern im Rahmen der Veranstaltung *Mathematik 2*.

5.2: Aufgabe: Fakultät

$n!$ steht für „n Fakultät“. Es gilt $0! := 1$ und für alle weiteren natürlichen Zahlen $n! = n \cdot (n-1)!$. Beispiel: $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot 1 = 120$. Implementieren Sie eine Methode, die einen Parameter vom Typ *int* entgegen nimmt und die Fakultät dieser Zahl zurück gibt. Der Rückgabetyp kann *int* sein, eventuell ist aber *long* besser (Warum?).

5.3: Aufgabe: Summe aufeinanderfolgender Zahlen

Der deutsche Mathematiker Carl Friedrich Gauß sollte als Neunjähriger die Aufgabe $1 + 2 + 3 + \cdots + 98 + 99 + 100$ lösen. Lösen Die diese Aufgabe auf drei Arten. Die Obergrenze soll dabei nicht 100 sein, sondern soll flexibel sein.

- Erstellen Sie eine Methode mit einer Schleife, die die Zahlen bis zur vorgegebenen Obergrenze addiert.
- Erstellen Sie eine Methode, die ohne eine Schleife auskommt und stattdessen eine Rekursion benutzt.
- Benutzen Sie für Ihre Methode die Gaußsche Summenformel.

5.4: Aufgabe: Codingbat: Recursion 1

Lösen Sie die leichteren Aufgaben zur Rekursion auf Codingbat.

5.5: Aufgabe: Palindrom

Ein *Palindrom* ist ein Wort, das vorwärts wie rückwärts gelesen gleich ist. Beispielsweise sind die Namen *ANNA* und *OTTO* Paldindrome oder auch das Wort *RENTNER*. Erstellen Sie eine Methode, die einen String als Parameter entgegen nimmt und zurück gibt, ob es sich um ein Palindrom handelt oder nicht. Die Methode soll rekursiv arbeiten und also keine Schleife beinhalten. Wenn Sie möchten, können Sie *zusätzlich* versuchen, eine entsprechende Methode mit einer Schleife statt einer Rekursion zu implementieren.

5.6: Aufgabe: Pascalsches Dreieck

Dieses Zahlenschema ist das Pascalsche Dreieck, benannt nach dem französischen Mathematiker und Physiker Blaise Pascal:

$$
\begin{array}{ccccccccccccc}
 & & & & & & 1 & & & & & & \\
 & & & & & 1 & & 1 & & & & & \\
 & & & & 1 & & 2 & & 1 & & & & \\
 & & & 1 & & 3 & & 3 & & 1 & & & \\
 & & 1 & & 4 & & 6 & & 4 & & 1 & & \\
 & 1 & & 5 & & 10 & & 10 & & 5 & & 1 & \\
1 & & 6 & & 15 & & 20 & & 15 & & 6 & & 1
\end{array}
$$

Jede Zeile beginnt und endet mit der Zahl 1, dazwischen stehen jeweils die Summen der Zahlen links und rechts über der Zahl. Die Zahlen im Pascalschen Dreieck nennt man man *Binomialkoeffizienten.* Ein *Binom* ist ein Ausdruck der Art $a^n + b^n$. Nimmt man das einfachste Binom dieser Art $a + b$ und setzt es in die n-te Potenz $(a+b)^n$ und multipliziert man diese Potenzen aus, dann ergibt sich

$$
\begin{array}{lccccccc}
(a+b)^0: & & & & 1 & & & \\
(a+b)^1: & & & 1a & & 1b & & \\
(a+b)^2: & & 1a^2 & & 2ab & & 1b^2 & \\
(a+b)^3: & 1a^3 & & 3a^2b & & 3ab^2 & & 1b^3.
\end{array}
$$

Man sieht, dass die Koeffizienten vor den Termen genau die Zahlen aus dem Pascalschen Dreieck sind.
Das Pascalsche Dreieck hat auch eine Bedeutung in der Kombinatorik. Beispiel: Wie viele Möglichkeiten gibt es, aus einer Gruppe von sechs Personen zwei Personen auszuwählen? Wir nummerieren die Zeilen des Pascalschen Dreieck beginnend mit der Nummer 0. Da wir über eine Gruppe von sechs Personen sprechen, suchen wir die sechste Zeile, also die Zeile 1, 6, 15, 20, 15, 6, 1. Die Zalhlen in dieser Zeile nummerieren wir, ebenfalls bei 0 beginnend durch. Wir suchen dann den zweiten Eintrag, das ist 15. Das ist die Antwort auf die oben gestellte Frage.
Erstellen Sie eine Methode, die die Zeilennummer und die Nummer der Zahl innerhalb dieser Zeile als Parameter entgegen nimmt und die Zahl, die an dieser Stelle im Pascalschen Dreieck steht, zurück gibt.

5.2 Kompliziertere Rekursionen

Türme von Hanoi heißt ein Spiel für eine Person. Ein Turm verschieden großer Scheiben liegt am Platz A. Der Turm soll auf Platz C neu aufgebaut werden. Um das Ziel zu erreichen, dürfen zwischenzeitlich auch Schreiben am Platz B liegen. Es darf immer nur eine Scheibe bewegt werden. Es darf niemals eine größere Scheibe auf einer kleineren Scheibe liegen. Die Abbildung 5.1 zeigt die Anfangsstellung.

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.

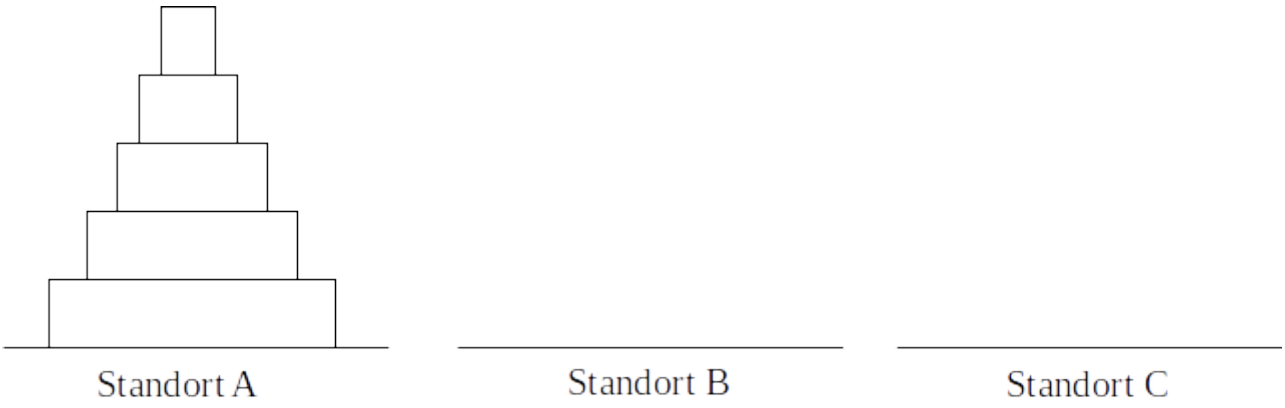


Abbildung 5.1: Türme von Hanoi

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

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.

5.8: Aufgabe: 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 bedeutet, die Methode hat vier Parameter.

Für einen Turm mit vier Steinen sieht das Aufrufschema so aus:

Tabelle 5.1: Aufrufschema für *Türme von Hanoi*

Schritt	**n = 4**	**n = 3**	**n = 2**	**n = 1**
	stapeln(4, a, b, c)			
	1.	stapeln(3, a, c, b)		
		1.	stapeln(2, a, b, c)	
			1.	stapeln(1, a, c, b)
1				**2. a → b**
2			**2. a → c**	
			3.	stapeln(1, b, a, c)
3				**2. b → c**
4		**2. a → b**		
		3.	stapeln(2, c, a, b)	
			1.	stapeln(1, c, b, a)
5				**2. c → a**
6			**2. c → b**	
			3.	stapeln(1, a, c, b)
7				**2. a → b**
8	**2. a → c**			
	3.	stapeln(3, b, a, c)		
		1.	stapeln(2, b, c, a)	
			1.	stapeln(1, b, a, c)
9				**2. b → c**
10			**2. b → a**	
			3.	stapeln(1, c, b, a)
11				**2. c → a**
12		**2. b → c**		
		3.	stapeln(2, a, b. c)	
			1.	stapeln(1, a, c, b)
13				**2. a → b**
14			**2. a → c**	
			3.	stapeln(1, b, a, c)
15				**2. b → c**

5.9: Aufgabe: Codingbat: Recursion 2

Versuchen Sie nun, die schwereren Aufgaben zur Rekursion auf Codingbat zu lösen.

5.10: Aufgabe: Potenzmenge

Die *Potenzmenge* einer Menge ist die Menge aller Teilmengen dieser Mengen. Beispiel: Die Menge {A, B} besitzt die Potenzmenge {{}, {A}, {B}, {A, B}}. Implementieren Sie eine rekursive Methode, die zu einer gegebenen Menge die Potenzmenge ausgibt.
Tipps:

- Die Reihenfolge ist in einer Menge egal. Daher ist im obigen Beispiel z.B. auch {{}, {B}, {A}, {A, B}} eine Lösung.
- Sie können das Problem mit einem Backtracking-Algorithmus lösen, nicht unähnlich wie in der Aufgabe *groupSum* und anderen Aufgaben aus Codingbat, Recursion 2.

5.11: Aufgabe: Alle Permutationen einer Zeichenkette

Implementieren Sie eine Methode, die alle Permutationen einer Zeichenkette ausgibt. Beispielsweise soll für die Zeichenkette *ABC*
ABC
ACB
BAC
BCA
CAB
CBA
ausgegeben werden.
Eine Rekursion muss abbrechen. Daher steht in den meisten rekursiven Methoden der rekursive Aufruf entweder in einem *if*- oder einem *else*-Zweig. Bei diesem Programm ist die einfachste Lösung eine rekursive Methode, bei der der rekursive Aufruf in einer Schleife steht. Auch dies ermöglicht den Abbruch der Rekursion.
Implementieren Sie die Methode mit zwei Parametern, *start* und *fertig*. Beim ersten Aufruf der Methode ist die Zeichenkette, deren Permutationen ausgegeben werden soll, der Parameter *start*, während der Parameter *fertig* zunächst der leere String ist. Die Methode beginnt dann mit einer for-Schleife, deren Schleifenvariable i von 0 bis zur Länge des Strings *start* geht. In der Schleife wird das i-te Zeichen in *start* entfernt und zu *fertig* hinzu gefügt. Mit diesen Werten erfolgt dann der rekursive Aufruf. Nach der Schleife wird geprüft, ob es noch Zeichen in der Zeichenkette *start* gibt. Wenn nein, das heißt alle Zeichen befinden sich nun in der Zeichenkette *fertig*, dann wird die Zeichenkette *fertig* ausgegeben.

6 Komplexität

6.1 Selectionsort

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.

6.1: Aufgabe: 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 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 zwei 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. Erstellen Sie eine Methode *tauschen()* mit zwei Parametern i und j des Typs *int*. Die Methode soll die Elemente an den Positionen i und j im Array *zahlen* vertauschen.
5. 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?
6. 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.

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.

6.2: Aufgabe: Implementieren von Selectionsort

Fügen Sie in Aufgabe 6.1 eine Methode *sortierenDurchAuswaehlen()* zur Klasse *MeineZahlen* hinzu. Diese sortiert das Array nach dem Algorithmus Selectionsort.

6.2 Kriterien zur Bewertung von Sortieralgorithmen

Wie gut ist der Algorithmus Selectionsort? 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 $n + (n-1) + (n-2) + (n-3) + \ldots + 3 + 2 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} - 1 = \frac{1}{2}(n^2 - n) - 1$ Werte untersuchen[1]. Die *1* fehlt in der Summe, da das letzte Element sicher am richtigen Platz ist, wenn alle anderen Elemente am richtigen Platz sind und daher die letzte Suche nach dem Minimum entfällt.

Im schlechtesten Fall steht kein Wert, wenn er ausgewählt wird, an der richtigen Stelle. Nach n – 1 Vertauschungen haben n – 1 Werte 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 verursachen, belaufen sich die Gesamtkosten auf $\frac{1}{2}(n^2 + n) - 1 + n - 1 = \frac{1}{2}n^2 + \frac{3}{2}n - 2$. Bei sehr großen n, und nur hier sind diese Laufzeituntersuchungen interessant, ist n gegenüber n^2 vernachlässigbar und auch die 2 spielt keine Rolle, es gilt also $\frac{1}{2}n^2 + \frac{3}{2}n - 2 \approx \frac{1}{2}n^2$. 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 $\frac{n(n+1)}{2} - 1$ Werte auf der Suche nach den jeweiligen Minima durchgegangen werden, es sind lediglich weniger Vertauschungen nötig, weil sich der eine oder andere Wert vielleicht zufällig schon an der richtigen Position befindet. Ist f(n) eine Funktion, die den Zeitbedarf ausdrückt, um n Zahlen mit dem Algorithmus Selectionsort zu sortieren, dann schreibt man kurz $f(n) \in \mathcal{O}(n^2)$, gelesen „f(n) ist von der Ordnung n Quadrat“ oder auch „f(n) ist von der Komplexität n Quadrat“. Diese Art der Notation nennt man „O-Notation“. Dabei ist $\mathcal{O}(n^2)$ die Menge derjenigen Funktionen von n, die nicht wesentlich schneller wachsen als die Funktion $g(n) = n^2$. Leider ist statt $f(n) \in \mathcal{O}(n^2)$ auch die Schreibweise $f(n) = \mathcal{O}(n^2)$ üblich, obwohl dies eigentlich Quatsch ist, da eine Funktion nicht gleich einer Menge von Funktionen sein kann. Ich ziehe daher die Schreibweise $f(n) \in \mathcal{O}(n^2)$ vor, aber Sie sollten nicht verwirrt sein, wenn Sie an anderer Stelle die andere Schreibweise sehen.

Hier kommt eine exakte Definition für $f(n) \in \mathcal{O}(g(n))$: Es existiert ein $n_0 \in \mathbb{N}$ und ein $c \in \mathbb{R}$, so dass für alle $n > n_0$ gilt $c \cdot g(n) > f(n)$.

[1] Die verwendete Gleichung ist die Gaußsche 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.

6.3: Aufgabe: $\mathcal{O}$-Notation

Prüfen Sie, ob $f(n) \in \mathcal{O}(g(n))$ oder $g(n) \in \mathcal{O}(f(n))$ oder beide Beziehungen oder keine davon gilt für die folgenden Funktionen f und g:

1. $f(n) = 4n, g(n) = n^2$
2. $f(n) = 1000n + 200, g(n) = n^2 - 5$
3. $f(n) = 4n \cdot ln\ n, g(n) = n^2$
4. $f(n) = 4n^3 - 1000, g(n) = n^2 + 1000$
5. $f(n) = 2^n, g(n) = n^3$
6. $f(n) = 4n^3, g(n) = n^3 - 5$

6.4: Aufgabe: Komplexität von Algorithmen

1. Bestimmen Sie die Ordnung eines Programms, das durch ein Array der Größe n durchgeht, um das Minimum zu finden.
2. Bestimmen Sie die Ordnung des Programms aus Aufgabe 3.16, in der das Einmaleins bis $n \cdot n$ ausgegeben wird.
3. In der Aufgabe 3.8 hat sich der Computer eine Zahl „ausgedacht“, die der Benutzer erraten muss und dabei die Hinweise „zu klein“ oder „zu groß“ bekommt erhält. Stellen Sie sich das Programm umgekehrt vor: Der Benutzer denkt sich eine Zahl zwischen 0 und n aus und der Computer muss diese erraten. Wenn er keine Hinweise bekommt, dann muss er alle Zahlen ausprobieren und das Problem ist offensichtlich von der Komplexität $\mathcal{O}(n)$. Wenn der Computer aber korrekte Hinweise „zu klein“ bzw. „zu groß“ bekommt, dann lässt sich die Aufgabe schneller erledigen. Implementieren Sie ein Programm, das möglichst schnell auf diese Weise eine Zahl ermittelt und bestimmen die die Komplexität Ihres Algorithmus’.

Ein anderes wichtiges Kriterium ist die *Stabilität* eines Sortieralgorithmus. Sortieralgorithmen in Tabellenkalkulationen müssen stabil sein. Hier ein einfaches Beispiel: In einer Tabelle sind die Namen Carl Müller, Bernd Schmidt und Albert Schmidt gespeichert. In der ersten Spalte stehen die Vornamen, in der zweiten die Nachnamen. Die Tabelle soll nun nach den Nachnamen sortiert werden. Sind zwei Nachnamen gleich, dann soll der Vornamen entscheiden. Das heißt, die fertig sortierte Liste soll lauten: Carl Müller, Albert Schmidt, Bernd Schmidt. Diese Sortierung kann man erreichen, indem man zuerst nach den Vornamen sortiert, so dass man die Reihenfolge Albert Schmidt, Bernd Schmidt, Carl Müller erhält. Anschließend sortiert man nach den Nachnamen. Nun ist die Frage, ob die Sortierung nach den Nachnamen die bereits erfolgte Sortierung nach den Vornamen korrekt lässt oder zerstört. Je nachdem nennt man den Sortieralgorithmus stabil oder instabil.

6.5: Aufgabe: Stabilität von Selectionsort

Untersuchen Sie, ob Selectionsort stabil ist.

6.3 Weitere einfache Sortierverfahren

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 bubble, Blase, bezieht sich darauf, dass die großen Zahlen nach oben steigen wie die Blasen in einer Limonade.

6.6: Aufgabe: Bubblesort

1. Erweitern Sie Ihre Klasse *MeineZahlen* um eine Methode *sortierenBubble*. 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?
4. Untersuchen Sie, ob Bubblesort stabil ist.

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.

6.7: Aufgabe: 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 Datensätzen 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.

6.8: Aufgabe: Laufzeit uns Sabilität von Insertionsort

Untersuchen Sie Insertionsort hinsichtlich Laufzeit und Stabilität.

6.4 Teile-und-herrsche-Verfahren

Teile und herrsche, englisch *divide and conquer*, beschreibt ursprünglich eine Herrschaftsmethode, die darauf beruht, die Untergebenen in kleine Gruppen mit widerstrebenden Interessen aufzuteilen, so dass sich diese Gruppen untereinander bekämpfen, anstatt sich gemeinsam gegen das herrschende System zu wenden. Der Begriff wird auch oft auf lateinisch, *divide et impera*, verwendet, weil unterstellt wird, dass das römische Reich so seine Herrschaft abgesichert habe. In der Informatik versteht man unter Teile-und-herrsche-Verfahren Algorithmen, die ein großes Problem so lange in kleinere Teilprobleme zerlegen, bis diese schließlich trivial lösbar sind. Anschließend wird die Lösung des großen Problems aus der Lösung der Teilprobleme berechnet. Oft lassen sich solche *Divide-and-Conquer-Algorithmen* am einfachsten mit einer Rekursion implementieren.

Quicksort ist ein Sortieralgorithmus, der nach dem Teile-und-herrsche-Prinzip arbeitet. 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 6.1.

6.9: Aufgabe: Implementieren von Quicksort

Implementieren Sie Quicksort.

Für die Berechnung des Laufzeitverhaltens von Quicksort nehmen wir zunächst den Fall an, dass n Elemente sortiert werden sollen und es sich bei n um eine Zweierpotenz handelt, d.h. $n = 2^k, k \in \mathbb{N}$. Im günstigsten Fall werden die Datensätze immer in zwei gleich große Teildatensätze getrennt, so dass $k = log_2 n$ Teilungen nötig sind. Bei jedem dieser $log_2 n$ Schritte muss man alle Elemente durchgehen, um sie dem richtigen Teildatensatz zuordnen zu können. Somit kommt man auf $n \cdot log_2 n$ Schritte. Logarithmen zu verschiedenen Basen unterscheiden sich durch einen konstanten Faktor. Konstante Faktoren werden in der $\mathcal{O}$-Notation aber vernachlässigt, daher wird die Basis nicht angegeben und es gilt für Quicksort im günstigsten Fall daher, dass dieser Algorithmus von der Ordnung $\mathcal{O}(n\ log\ n)$ ist. Ist n keine Zweierpotenz, dann muss man zur nächsten Zweierpotenz aufrunden. Das ändert aber nichts Grundsätzliches an den eben gemachten Überlegungen. Im statistischen Fall wird ein Datensatz nicht

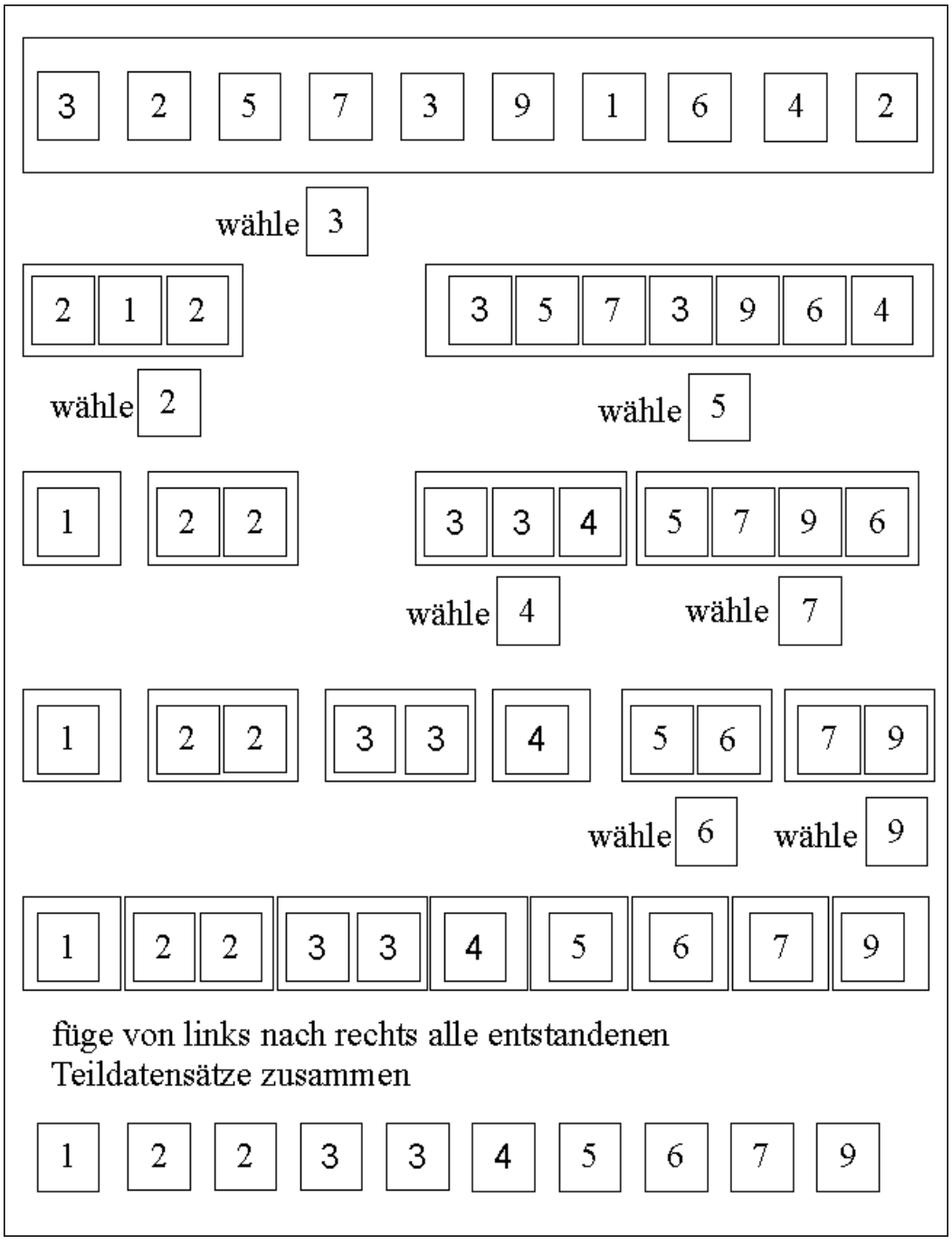


Abbildung 6.1: Quicksort

in zwei gleich große Teildatensätze getrennt. Das verlängert die Laufzeit, da ein größerer Teildatensatz entsteht und für dessen Zerlegung eine längere Kette weiterer Zerlegungen erforderlich ist. Im Sinne der O-Notation ändert dies aber auch nichts: Nimmt man beispielsweise an, dass im Schnitt die Zerlegung in Teildatensätze der Größe $\frac{2}{3}$ und $\frac{1}{3}$ der ursprünglichen Größe vorgenommen werden, dann verlängert sich die Anzahl der Teilungen um einen konstanten Faktor, der aber in der $\mathcal{O}$-Notation nicht berücksichtigt wird. Somit ist auch das durchschnittliche Laufzeitverhalten von Quicksort 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.

Der schlechteste Fall für Quicksort ist der, in dem die Elemente bereits sortiert sind, wenn man außerdem eine einfache Möglichkeit verwendet, dasjenige Element zu bestimmen, anhand dessen die nächsten Teildatensätze erzeugt werden. Beispiel: Es sollen die Zahlen 1, 2, 3, 4, 5, 6, 7, 8 sortiert werden. Man muss ein Element suchen, um den Datensatz in zwei Teildatensätze aufzuteilen. Dieses darf nicht minimal sein. Daher betrachtet man die ersten beiden verschiedenen Elemente, also hier im Beispiel 1 und 2 und wählt davon das größere, also 2. Damit wird der Datensatz aufgeteilt in den Teildatensatz 1 und den Teildatensatz 2, 3, 4 5, 6, 7, 8, 9. In jedem Schritt wird somit immer nur ein Element in einem Teildatensatz abgetrennt und alle anderen bilden den anderen Teildatensatz. Daher muss der Schritt des Aufteilens sehr oft durchgeführt werden.

6.10: Aufgabe: Komplexität von Quicksort

1. Diskutieren Sie, welche Komplexität Quicksort im schlechtesten Fall hat.
2. In Tabellenkalkulationsprogrammen wie LibreOffice-Calc oder auch Excel stehen dem Anwender Sortierfunktionen zur Verfügung. Diskutieren Sie, ob es eine gute Idee ist, diese Sortierfunktionen mit Quicksort zu implementieren.
3. Erörtern Sie, wie man bei Quicksort verhindern kann, dass ausgerechnet das nochmalige Sortieren einer bereits sortierten Liste am längsten dauert.

Wie Quicksort ist *Mergesort* ein Teile-und-herrsche-Algorithmus und wir damit am einfachsten rekursiv implementiert. 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 so lange wiederholt, bis nur noch ein, dann geordneter, Datensatz vorliegt. Ein Beispiel ist in Abbildung 6.2 gezeigt.

6.11: Aufgabe: Implementieren von Mergesort

Implementieren Sie Mergesort.

Mergesort und Quicksort haben gemeinsam, dass sie Teile-und-herrsche-Algorithmen sind, unterscheiden 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.

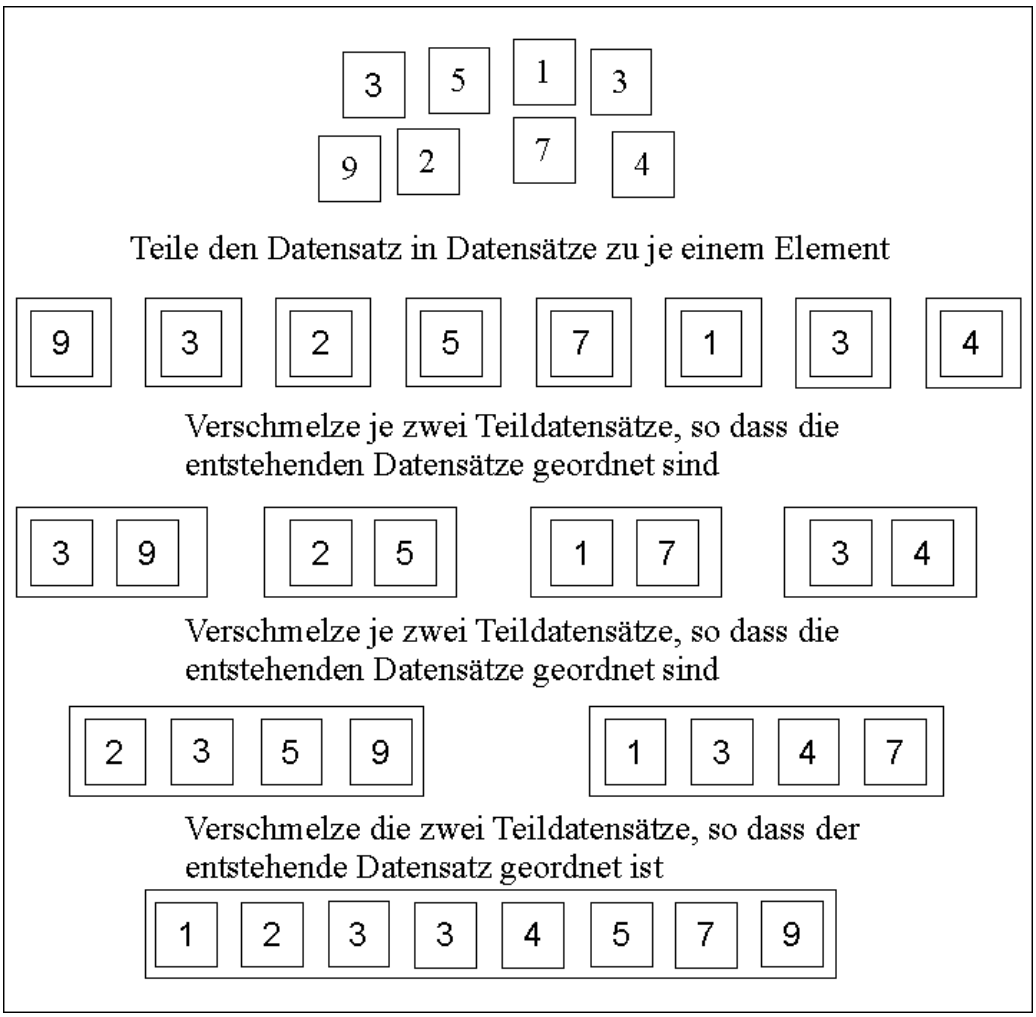


Abbildung 6.2: Mergesort

Für die Berechnung der Komplexität von Mergesort kann man sich klar machen, dass man für jede Verschmelzung zu halb so vielen Teildatensätzen durch alle n Elemente durchgehen muss. Man benötigt $log_2 n$ Verschmelzungen, wiederum vorausgesetzt, dass n eine Zweierpotenz ist. Ohne diese Voraussetzung ändern sich aber nur Konstanten, die nicht berücksichtigt werden. Insgesamt ergibt sich somit, dass Mergesort von der Ordnung $\mathcal{O}(n\ log\ n)$ ist.

Im Gegensatz zu Quicksort hat Mergesort kein besonderes Problem mit vorsortierten Daten. In der Praxis schneidet Mergesort aber im Allgemeinen,d.h. wenn die Datensätze nicht vorsortiert sind, etwas schlechter ab als Quicksort. 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.

7 Anhang

7.1 Verzeichnisse: Literatur, Abbildungen, Tabellen, Exkurse, Aufgaben

Literatur Sie können meinen Informatik-Veranstaltungen am Studienkolleg ohne zusätzliche Literatur folgen. Oft kommen aber Studierende auf mich zu, die Informatik studieren möchten und in der Kollegszeit in Informatik mehr als den hier vorgestellten Stoffe bearbeiten möchten. Andererseits fragen mich auch Studierende, die Probleme mit dem Erlernen von Programmierung haben, nach Literatur, um neben meinem Unterricht einen weiteren Kanal zu haben. Bei allen Unterschieden haben die beiden Gruppen eines gemeinsam: Sie benötigen Literatur zur Programmierung, denn das ist die Haupthürde zu Beginn des Informatik-Studiums, für manche aber eben auch bereits am Kolleg. Daher empfehle ich hier ein paar Bücher zur Java-Programmierung, nicht zu anderen Themen des Informatik.

Es gibt sehr viele Bücher über Java-Programmierung. Die hier gebotene Auswahl ist daher letztlich zufällig. Meine Empfehlung ist daher: Gehen Sie in die Bibliothek und leihen Sie sich einige Bücher aus, die Sie ansprechend finden. Verengen Sie dann die Auswahl auf ein oder zwei Bücher, die Sie durcharbeiten. Praktisch alle Bücher, die Sie finden, werden über den in diesem Skript behandelten Stoff hinaus gehen. Für die angehenden Informatik-Student:innen ist das von Vorteil. Diejenigen, die im eigenen Tempo den Stoff meines Unterrichts lernen möchten, müssen gegebenenfalls gut abgleichen, welche Themen relevant sind. Naturgemäß werden die vorderen Kapitel dieser Bücher eher relevant sein.

1. codingbat.com. Diese Online-Quelle ist kein systematisches Lehrbuch, sondern eine Sammlung kurzer Übungsaufgaben. Zu den Aufgaben werden Test und z.T. Musterlösungen angeboten.

2. Habelitz, Hans-Peter, Programmieren lernen mit Java, Rheinwerk Computing, 2024
   ISBN-10 3367104787, ISBN-13 978-3367104789. Das Buch hat 557 Seiten und kostet 24,90€
   Ein gutes Buch für den Einstieg in Java zu einem günstigen Preis.

3. Sierra, Kathy et al., Java von Kopf bis Fuß, Dpunkt.Verlag GmbH, 2023
   ISBN-10: 3960092067, ISBN-13: 978-3960092063. Das Buch hat 717 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.

4. Christian Ullenboom, Java ist auch eine Insel, Rheinwerk Computing, 2023
   ISBN-10: 383629544X, ISBN-13: 978-3836295444. Das Buch hat 1223 Seiten und kostet 49,90€
   Mit 1223 Seiten extrem viel Buch fürs Geld. Allerdings eher nicht so gut geeignet für Anfänger. Das Buch ist eher ein Nachschlagewerk für Fortgeschrittene.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Liste der Aufgaben

Liste der Exkurse