ulm university universität
uulm

Java Grundlagen

Skript zum Kurs der Programmierstarthilfe
Fakultät für Ingenieurwissenschaften und Informatik
Universität Ulm

Guido de Melo und das Programmierstarthilfe-Team

2009

Über dieses Skript

Dieses Skript über die Programmiersprache Java wurde für die Programmierstarthilfe an der Universität Ulm unter Verwendung von Studiengebühren geschrieben. Es richtet sich insbesondere an Programmierneulinge und kann auch unabhängig von den Materialien der Programmierstarthilfe zum Selbststudium verwendet werden. Thematisch beginnt das Skript bei den absoluten Grundlagen und leitet den Leser sehr ausführlich durch die Programmierkenntnisse die ein Informatikstudent im ersten Semester vermittelt bekommt.

Fehler gefunden?

Wenn du in diesem Skript einen Fehler findest oder etwas von der Formulierung her schwer verständlich findest, dann melde dich bitte bei uns. Wir sammeln alle Fehler und verbessern die kommende Version des Skripts. Schicke einfach eine Mail an `guido.de-melo@uni-ulm.de`.

Fassung 29. März 2009, enthält Korrekturen aus WS 2008/09
Revision 256
© 2009 das Programmierstarthilfe-Team

Name	Kapitel	sonstiges
Guido de Melo	1, 2, 4, 5, 6, 7, 8	LaTeXen, Bilder, Endredaktion
Benjamin Erb	8, 9, B, D	Korrekturen
Peter Lobner	7, 8	Korrekturen
Robert Schmitz	3, 4, C	IDEs, Korrekturen
Finn Steglich	4, 5, 6	Korrekturen
Juliane Wessalowski		Korrekturen
Marcus Bombe	0, 5, 8, A	Syntaxübersicht, Korrekturen

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 2.0 License. To view a copy of this license, visit
`http://creativecommons.org/licenses/by-nc-sa/2.0/de/` or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.
Satz: PDF-LaTeX 2_ε

Inhaltsverzeichnis

0 Einleitung

Hallo Informatik-Studenten und Java-Interessierte,

dieses Skript behandelt die Grundlagen der Programmiersprache Java. Es wurde von den Veranstaltern der Programmierstarthilfe der Universität Ulm geschrieben und behandelt ziemlich genau den Stoff, der im ersten Semester in der Vorlesung Praktische Informatik zum Thema Java behandelt wird. Es handelt sich folglich nicht um ein umfassendes Werk über Java welches jedes Thema in allen Details behandelt - für diesen Zweck gibt es schon hunderte andere Bücher. Viel mehr ist es eben eine Zusammenfassung der Grundlagen. Aus diesem Grund ist dieses Skript auch so geschrieben, dass auch Anfänger damit umgehen können und Spaß am Programmieren entwickeln können. Hierzu klären wir in der Einleitung erst einmal alles Grundsätzliche was man noch vor der ersten eigenen Zeile Java-Code wissen sollte. Im Nachfolgenden kommen wir dann zu teilweise aufeinander aufbauende Kapitel mit den Java-Grundlagen. Ein etwas größerer Anhang rundet das Skript ab und enthält Hilfsmittel und zudem einen Ausblick auf Themen jenseits der Grundlagen.

Das Skript wurde für die Programmierstarthilfe an der Universität Ulm geschrieben. Ursprünglich wurden die Java-Grundlagen auf den Aufgabenblättern der Programmierstarthilfe vermittelt, allerdings sind wir der Meinung eine Trennung von Theorie und Aufgaben erhöht die Lesbarkeit und somit auch die Benutzbarkeit sowohl von Aufgabenblättern als auch des Skriptes. Damit Aufgaben und hiermit zusammenhängendes Wissen allerdings trotzdem zusammenfinden, verweisen die Aufgabenblätter der Programmierstarthilfe auf die einzelnen Kapitel und Unterkapitel dieses Skriptes. Das Skript hingegen wurde bewusst so geschrieben, dass es unabhängig von der Programmierstarthilfe und den Aufgabenblättern verwendet werden kann.

Viel Spaß mit der Programmiersprache Java!
Euer Programmierstarthilfe-Team

0.1 Zweck der Einleitung

Wir besprechen in dieser Einleitung alle grundlegenden Dinge, die man wissen sollte noch bevor man überhaupt Befehle der Programmiersprache Java erlernt. Beginnen werden wir mit einer Klärung des Begriffs des Programmierens, dann werden wir uns ansehen, was Java eigentlich für eine Programmiersprache ist und welche Voraussetzungen es gibt, um Java zu verwenden. Sobald wir das wissen, sehen wir uns ein kleines Programm in Java an und besprechen Zeile für Zeile was dort eigentlich steht und was das bedeutet. Es gilt weiterhin zu erfahren, wie man sein eigenes Programm am Besten eingibt und wie man den Computer dazu bringt dieses Programm dann zu starten. Zum Ende der Einleitung behandeln wir zudem noch, was man bei Problemen tun kann und wie man verständliche Programme produziert.

Diese Einleitung ist zudem nicht nur für absolute Anfänger interessant. Auch wer schon einmal programmiert hat und daher schon die ein oder andere Erfahrung mitbringt wird in dieser Einleitung Wichtiges erfahren können. Daher sei es jedem nahe gelegt die Einleitung zumindest einmal aufmerksam zu lesen um dann bei auftretenden Fragen oder Problemen während des Programmierens hierher zurückkehren zu können um nochmals nach schauen zu können, wie man ein Java-Programm übersetzt und ausführt oder was bei Fehlern zu tun ist.

Noch bevor wir allerdings all das oben beschriebene angehen, müssen wir zunächst noch über zwei häufig auftretende Irrtümer sprechen um mit falschen Vorstellungen von vorne herein aufzuräumen.

0.2 Häufige Irrtümer

0.2.1 Irrtum Nr. 1: Informatiker gleich Programmierer

Wenn man beispielsweise auf der nächsten Familienfeier seinen Verwandten erzählt, man habe sich nun für Informatik eingeschrieben, so kommt bei den meisten sofort die Vorstellung auf „Informatiker? Das sind doch die, die Computerprogramme schreiben!“. Diese Vorstellung ist größtenteils falsch, denn Informatiker und Programmierer sind unterschiedliche Berufsgruppen. Der Programmierer ist derjenige, der die Programme schreibt. Zum beruflichen Programmierer wird man beispielsweise über die Ausbildung zum Fachinformatiker.

Informatik hingegen studiert man an einer Hochschule und die Hochschulen behaupten hin und wieder sogar, dass Programmierkenntnisse für einen Abschluss in einem Informatikstudiengang nicht zwingend erforderlich sind. Diese Aussage mag einem wenig helfen, wenn man gerade versucht, die Prüfung zur praktischen Informatik zu bestehen. Richtig ist aber trotzdem, dass man als Informatiker hauptsächlich nicht programmiert und der Computer für den Informatiker nur ein Werkzeug darstellt. Informatiker beschäftigen sich vielmehr mit strukturierten, systematischen Planungen, Überwachung der Ausführung und Verifizierung der Ergebnisse, im Kontext von Informationen. Das sind dann zum Beispiel die Planung oder Leitung von Software-Projekten oder die Anfertigung von Spezifikationen, die andere Hardware oder Software in einem Projekt einzuhalten hat.

Aber auch wenn der Informatiker später eher wenig programmieren muss, ist ein genaues Verständnis über die Funktionsweise von Computerprogrammen bei sehr viele Aufgaben unerlässlich. Aus diesem Grund bekommt man auch die Grundlagen einer Programmiersprache im Studium vermittelt, denn wie könnte man besser ein genaues Verständnis erwerben als durch selbst zu programmieren? Beschäftigen wir uns in diesem Zuge mit dem zweiten Irrtum, dem man möglicherweise aufsitzen könnte:

0.2.2 Irrtum Nr. 2: Programmieren lernen heißt 3D-Spiele bauen

Wenn man nun als Informatikstudent aus oben genannten Gründen eine Programmiersprache beigebracht bekommt, produziert man deshalb noch lang keine 3D-Spiele. Statt dessen reichen die Grundlagen um textbasierte Programme zu schreiben, das bedeutet, dass wenn das Programm gestartet wird kein buntes Fenster aufgeht und man auf irgendetwas klicken kann. Aber das ist auch überhaupt nicht das Ziel, wenn es darum geht die Grundlagen zu erarbeiten. Es geht beispielsweise darum zu erlernen, wie man ein Programm so schreibt, dass es die einzelnen Arbeitsschritte sinnvoll hintereinander ausführt und zwar gerade so und sooft wie die Situation es erfordert. Wir werden zum Beispiel lernen, wie man Befehle nur unter gewissen Bedingungen ausführt oder Dinge sooft tut, solange nicht eine andere Bedingung eintritt. Auch werden wir uns mit Datenstrukturen beschäftigen. Wir werden klären, wie der Computer Informationen abspeichert, wie wir vorgehen müssen wenn wir diese Informationen verändern wollen aber auch wie wir komplexere Informationen, das heißt aus mehreren einfachen Informationen zusammengesetzte Informationen, in unserem Programm abbilden.

Für all diese Fragen ist eine schöne Ausgabe der Ergebnisse wenig interessant. Folgender Vergleich mag das verdeutlichen: Wenn jemand ein Auto konstruieren soll, so geht es erstmal darum, dass dieses Auto fahren kann und sich sicher lenken lässt – die Farbe des Lacks

und das Design des Interieurs ist erstmal absolut nebensächlich. Will man also unbedingt 3D-Spiele basteln, so wird man sich dies in der Freizeit aneignen müssen. Dennoch wären auch hierfür die hier vermittelten Grundlagen der erste Schritt, denn so sehr ein Auto ohne Lack und Verkleidungsteile fahren kann, so wenig wird man einen Eimer Farbe dazu überreden können, einen von A nach B zu bringen.

0.3 Programmieren?

Nachdem nun schon sooft der Begriff *Programmieren* gefallen ist, gilt es nun endlich diesen Begriff und den ganzen Prozess des Computer-Programm-Erstellens mal etwas näher zu beleuchten. Dabei ist für uns zum einen von Interesse, was eigentlich im Computer selbst passiert, aber auch, wie wir von einer Aufgabe zu einem Programm kommen.

Von Prozessoren, Maschinencode und Hochsprachen

Sprechen wir erstmal davon, was der Prozessor des Computers für Fähigkeiten beherrscht. Der Prozessor kennt nämlich nur ganz grundlegende Operationen wie zum Beispiel die Grundrechenarten und das Lesen sowie Schreiben im Arbeitsspeicher. Alle Programme die man auf dem Computer ausführt bestehen ausschließlich aus tausenden von solchen grundlegenden Operationen. Diese werden in Maschinencode gespeichert, wobei ein solcher Maschinencode einfach die Hintereinanderreihung von Nummern ist und jede Nummer für eine Operation steht. Unter Windows haben Dateien die Maschinencode enthalten häufig die Dateiendung `.exe`. Man kann Programme auch quasi in Maschinencode schreiben. Dazu gibt man schlicht eine Liste von Abkürzungen ein wobei jede Abkürzung direkt eine Operation in Maschinencode entspricht. Dies nennt man dann Assemblersprache und ein Programm namens Assembler übersetzt diese dann in Maschinencode, indem jede Abkürzung eins zu eins in die Nummer der Operation umgeschrieben wird, für die sie steht. Das lernt man in der Technischen Informatik kennen, und wird in der Praxis zum Beispiel benötigt, um dem Computer Anweisungen zu geben, den Kern des Betriebssystems in den Speicher zu laden und dann auszuführen.

Allerdings ist es nahezu unmöglich komplexere Programme in einer Assemblersprache zu schreiben. Daher wurden die sogenannten Hochsprachen erfunden. Eine der bekanntesten Hochsprachen ist C und dessen Erweiterung C++. Eine solche Hochsprache wird oft einfach nur Programmiersprache genannt. Die Befehle einer Programmiersprache sind so gebaut, dass man durch ihre Kombination wesentlich Aufgabenorientierter dem Computer Anweisungen geben kann und nicht mehr gezwungen ist alles in Grundrechenarten oder Speicherzugriffe herunter zu brechen. Zu einer Datei, die Anweisungen in einer Hochsprache enthält, sagt man Quelltext. Allerdings ist ein solcher Quelltext nicht mehr direkt vom Prozessor ausführbar, denn Befehle aus der Hochsprache steht häufig für eine Aneinanderreihung von vielen Maschinenbefehlen, oft in Abhängigkeit zu vorhergehenden Befehlen der Hochsprache. Aus diesem Grunde muss man ein Übersetzungsprogramm – der sogenannte Compiler – verwenden, um aus dem in einer Hochsprache geschriebenen Quelltext wieder Maschinencode zu erzeugen.

Vom Problem zum Programm

Nachdem wir nun eine Vorstellung davon haben, was mit unserem Quelltext passiert nachdem wir ihn geschrieben haben, müssen wir noch klären, wie wir überhaupt zum Quelltext kommen. Normalerweise sind die Aufgaben, die ein von uns geschriebenes Programm erfüllen soll nämlich aus der realen Welt und daher müssen wir diese Aufgaben erst einmal

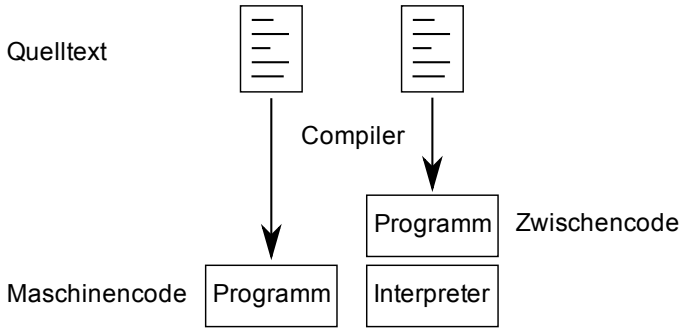


Abbildung 0.1: Vom Quelltext zum fertigen Programm: Je nach Programmiersprache wird der Quelltext vom Compiler direkt in Maschinencode übersetzt oder zunächst in einen Zwischencode übersetzt, welcher dann von einem Interpreter ausgeführt wird.

genau betrachten und in einen Algorithmus fassen. Ein Algorithmus ist eine aufs genauste definierte Abfolge von elementaren Grundschritten. So wird aus der Aufgabenstellung „Einkaufen“ der Algorithmus „Einkaufswagen holen - Solange noch Artikel auf der Einkaufsliste stehen: Artikel suchen und in Wagen legen, dann den Artikel von der Liste streichen - an der Kasse bezahlen“. „An der Kasse zahlen“ kann man wiederum zerlegen in „Waren aufs Band legen - Geldbeutel hervorholen - von der Kassiererin genannte Geldmenge übergeben“. Offensichtlich hängt unser Algorithmus also auch davon ab, was wir als elementare Grundschritte ansehen. Wenn wir nun eine Aufgabe in Maschinencode zu lösen hätten, müssten wir alles soweit zerlegen bis die gesamte Aufgabe nur noch aus einer Aneinanderreihung von Grundrechenarten besteht. Da wir allerdings eine Hochsprache verwenden, wird uns ein Teil der Arbeit abgenommen, denn wir müssen die Aufgabe nur soweit auseinander nehmen, bis alle Bestandteile als Befehle der Hochsprache oder als bereits fertige programmierte Unteraufgaben dastehen. Diese Unteraufgaben können sowohl von einem selbst programmiert sein oder auch schon von der Programmiersprache zur Verfügung gestellt werden. Zum Beispiel stellen Hochsprachen im Allgemeinen immer eine einfache Möglichkeit bereit, eine Textzeile auf dem Bildschirm auszugeben. Also wäre es nicht nötig sich darüber Gedanken zu machen, wie man nun die einzelnen Buchstaben einer solchen Textzeile nacheinander an die Grafikkarte schickt.

Wir stellen also fest, Programmieren bedeutet, ein gegebenes Problem zu zerlegen und durch bereits bestehende Bausteine darzustellen. Dazu ist es natürlich unerlässlich diese Bausteine kennen und verwenden zu lernen. Und genau diese Kenntnis um die Bausteine, aber auch die Denkweise wie man die Bausteine kombinieren muss, um eine gegebene Aufgabe zu lösen, das macht das Programmieren-Lernen aus. Zu Beginn dieses Lernens werden die meisten oft Ärger mit dem Compiler haben, denn der Compiler besteht darauf, dass der Quelltext vollkommen korrekt eingegeben wurde und absolut kein Spielraum für Interpretationsmöglichkeiten vorhanden ist. Dies liegt daran, dass der Compiler natürlich nicht wissen kann wie es gemeint ist, wenn es mehrere Möglichkeiten gibt den Quelltext zu verstehen. Und statt es einfach irgendwie zu machen und durch Willkür unberechenbare Programme zu erzeugen, wird stattdessen bei jeder kleinsten Ungereimtheit ein Fehler ausgegeben und das Übersetzen abgebrochen. Das mag einen am Anfang verzweifeln lassen und frustrieren, aber bereits nach kurzer Zeit kann man Fehler schnell korrigieren oder macht sie erst gar nicht. Man muss sich aber in jedem Fall im Klaren sein, dass Lernen durch Versuch und Irrtum mit Sicherheit ein Bestandteil des Programmieren-Lernens ist.

0.4 Allgemeines über Java

Nun wissen wir grob, was es mit dem Programmieren auf sich hat und sollten uns nun endlich Java zuwenden. Java ist im Vergleich zu C und C++ eine recht junge Hochsprache und erfreut sich zudem großer Beliebtheit. Java hat im Übrigen überhaupt nichts mit JavaScript zu tun. Die beiden Sprachen haben komplett verschiedene Anwendungsbereiche und sind sehr verschieden.

Java hat eine Besonderheit im Gegensatz zu klassischen Hochsprachen: Ein in Java geschriebenes Programm ist in seinem fertigen Zustand auf dutzenden Plattformen lauffähig, das heißt ein Java Programm kann unverändert auf verschiedenen Betriebssystemen und Prozessorarchitekturen ausgeführt werden. Ein Programm in C++ kann dies beispielsweise nicht, da der Compiler es beim Übersetzen direkt in den Maschinencode der Zielplattform übersetzt. Java erreicht diese Plattformunabhängigkeit, indem eine Zwischenebene eingeführt wurde. Der Compiler in Java erzeugt keinen Maschinencode aus dem Quelltext (Dateiendung .java), sondern erzeugt sogenannten Bytecode (Dateiendung .class). Bytecode kann man sich als eine Art Maschinencode vorstellen, der einem konkreten Prozessor unabhängig ist. Allerdings ist Bytecode daher nicht direkt von einem Prozessor ausführbar. Stattdessen wird der Bytecode von der Java Runtime Environment (JRE) eingelesen und ausgeführt und diese Runtime Environment ist wiederum für eine konkrete Plattform erzeugt worden aber auf allen Plattformen vom Funktionsumfang identisch. Das bedeutet, dass jeder der Java-Programme ausführen will eine solche Runtime Environment benötigt. Woher man die Runtime Environment und auch den Java-Compiler bekommt, klären wir am Ende des Abschnittes.

Eine weitere Eigenschaft von Java ist, dass die Erfinder von Java viele funktionierende Konzepte aus C++ und anderen Sprachen übernommen haben. So ist Java eine objektorientierte Programmiersprache. Was das bedeutet werden wir allerdings erst in einem der späteren Kapitel klären. Allerdings haben die Entwickler von Java bewusst nicht alle Elemente aus C++ übernommen. Beispielsweise unterstützt Java absichtlich keine sogenannte Zeigerarithmetik, das ist die Fähigkeit die Speicheraddresse von Daten direkt zu manipulieren. Statt dessen kümmert sich die Java Runtime Environment um die Speicherverwaltung komplett selbstständig. So wird auch das Aufräumen des Speichers nicht dem Programmierer überlassen. Der sogenannte Garbage Collector sucht nach nicht mehr verwendeten Daten und gibt die Speicherbereiche dieser Daten wieder an das Betriebssystem zurück. Man ist Aufgrund solcher Eigenschaften von Java allerdings im Allgemeinen nicht beim Programmieren eingeschränkt; Java nimmt einem lediglich die Arbeit ab.

Java kommt mit einer sehr umfangreichen Klassenbibliothek daher. Unter einer Klassenbibliothek versteht man eine Ansammlung von kleinen Unterprogrammen, welche man verwenden kann um viele häufig wiederkehrende Aufgaben zu erledigen und diese somit nicht selbst programmieren muss. So ist Java schon von Haus aus in der Lage mit Netzwerken und insbesondere mit dem Internet umzugehen. Auch beinhaltet Java Komponenten für graphische Benutzeroberflächen. Diese Klassenbibliothek ist Bestandteil der Java Runtime Environment und somit auf jedem Computer automatisch verfügbar, der Java-Programme ausführen kann.

Woher bekommt man nun aber den Java Compiler und die Runtime Environment? Für Windowsbenutzer gilt folgendes: Beides ist unter http://java.sun.com erhältlich. Im Downloadbereich muss man hierzu „Java SE“ auswählen. SE steht für Standart Edition. Der Compiler ist enthalten im Java SE Development Kit, kurz JDK. Hierin ist auch schon die passende Runtime Environment, kurz JRE, enthalten. Das bedeutet, dass man die Runtime Environment nur für Rechner herunterladen muss auf denen man ausdrücklich Java-Programme nur ausführen, nicht aber schreiben möchte. Zudem kann man sich hier auch die Dokumentation der Klassenbibiothek herunterladen. Wie man damit arbeitet wird im

Anhang besprochen, ist aber im Moment noch sehr nebensächlich. Wichtig ist, dass man sich das aktuelle JDK herunterlädt und installiert, da dieses den Compiler für Java enthält.

Linuxbenutzer können das JDK auch von `http://java.sun.com` bekommen oder besorgen sich das JDK über ihre Distribution bzw. Paketverwaltung. MacOS-Benutzer erhalten das JDK direkt von Apple.

0.5 Editoren

Nachdem man nun den Java-Compiler auf seinem Rechner hat, müssen wir noch besprechen, wie man Java-Quelltext am einfachsten eingibt. Hierbei gibt es mehrere Möglichkeiten: Zum einen kann man *Plain Text*-Editoren verwenden, zum anderen gibt es auch einige IDEs für Java. Was man unter diesen Begriffen zu verstehen hat, klären wir jetzt.

Einfache Texteditoren

Der simpelste Weg Java-Quelltext zu erzeugen, ist einen einfachen Texteditor wie zum Beispiel Notepad zu verwenden. Wichtig ist allerdings, dass es sich um einen Editor handelt, der die Daten als reinen Text abspeichert (Dateiendung unter Windows häufig .txt). Open Office Writer oder Microsoft Word sind hingegen nicht geeignet, denn die hiermit produzierten Dateien enthalten zusätzliche Informationen wie Schriftart, -farbe, -größe und jede Menge Informationen zum Dokument selbst. Mit diesen zusätzlichen Informationen kann der Java Compiler allerdings überhaupt nichts anfangen und die Quelldatei auch als solche nicht erkennen. Wichtig ist beim Abspeichern zudem, dass die Datei die Dateiendung .java haben muss, damit der Compiler mit ihr arbeitet. Im Übrigen gibt es mehrere wesentlich geeignetere Texteditoren als Notepad. Ein zum Programmieren geeigneter Texteditor sollte nämlich Zeilennummern anzeigen können. Dies ist sehr nützlich bei der Fehlersuche, dazu aber später. Zudem ist es sehr hilfreich, wenn der Texteditor die Befehle farblich hervorhebt. Dies nennt sich Syntax Highlighting.

Entwicklungsumgebungen

Außer einfachen Texteditoren gibt es allerdings auch noch die sogenannten IDEs. IDE steht für Integrated Development Environment, zu deutsch: Entwicklungsumgebung; also ein Programm in welches schon Hilfsmittel für die Programmierung eingebaut sind. Solche IDEs helfen einem die Übersicht bei mehreren geöffneten Quelltexten zu behalten und können häufig auch kleinere Aufgaben automatisiert für den Programmierer erledigen. Angenehm ist insbesondere, dass IDEs im Allgemeinen einen Button haben, welcher die aktuelle Datei automatisch compiliert und ausführt. Zudem können sie oft auf Fehler beim Programmieren hinweisen.

Eine Liste mit Editoren und IDEs und vieles zur Entscheidungshilfe haben wir für dich im Anhang gesammelt.

Vorteile von einfachen Texteditoren

Obwohl die Vorteile für eine IDE auf der Hand liegen, raten wir dem Programmieranfänger, zumindest die ersten Wochen mit einem einfachen Texteditor zu arbeiten anstatt eine IDE zu verwenden. Dies hat zwei Gründe: Zum einen ist die IDE nochmal ein Programm, das einem neu ist und möglicherweise für zusätzliche Probleme sorgt. Viel wichtiger aber ist der zweite Grund: Dadurch, dass IDEs dafür sorgen, dass man einige Standardzeilen

nicht mehr selbst tippen muss, hat man keine Chance eben jene Zeilen ohne IDE schreiben zu lernen oder einen Fehler in diesen zu finden. Vielen ist es schon passiert, dass sie diese Zeilen zwar kannten, in der Prüfung dann allerdings nicht zu Papier haben bringen können, da sie diese Zeilen nie haben selber tippen müssen. Und Programmieren ohne Übung funktioniert eben nicht.

0.6 Das erste Programm: Hello World

Schauen wir uns nun endlich mal ein Java-Programm im Quelltext an, nachdem wir jetzt so viel Theoretisches besprochen haben. Das nachfolgende Programm gibt „Hello World“ aus. Wir werden gleich im Detail besprechen, was die einzelnen Zeilen bedeuten. Man muss sich dies allerdings nicht gleich merken können, viele Details haben erst später eine größere Bedeutung und schließlich kann man immer hierher zurückblättern. Kommen wir zum Quelltext:

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

Der Quelltext besteht aus fünf Zeilen. Das erste was auffällt ist, dass einige Zeilen eingerückt sind. Diese Einrückung dient ausschließlich der Lesbarkeit und ist für den Java Compiler irrelevant. Mehr zum Thema Einrücken wird am Ende der Einleitung besprochen. Grundlegend ist allerdings folgendes: In der ersten Zeile steht etwas geschrieben und dahinter öffnet sich eine geschweifte Klammer. Das Gegenstück zu dieser Klammer befindet sich in Zeile Fünf. Hier wird der sogenannte Block wieder geschlossen. Alle Zeilen dazwischen sind um mindestens eine Ebene eingerückt, um zu verdeutlichen, dass sich diese Zeilen im Inneren dieses Blocks von Zeile eins bis fünf befinden. Selbiges gilt für Zeile drei, welche sich im Inneren des Blocks von Ende Zeile zwei bis zur vierten Zeile befindet. Aus diesem Grund ist Zeile drei nochmals eingerückt. Kommen wir zum Inhaltlichen:

In Zeile eins wird die Klasse namens Hello begonnen. Die Klasse ist öffentlich. Das bedeutet, dass andere Programmteile diese Klasse mit benutzen dürfen. Genaueres folgt im Kapitel zur Objektorientierung. Wichtig ist, dass wenn die im Quelltext beschriebene öffentliche Klasse `Hello` heißt, auch die Datei die diesen Quelltext enthält `Hello.java` heißen muss, ansonsten meldet der Compiler einen Fehler. Alles weitere spielt sich im Inneren dieser Klasse ab.

In der zweiten Zeile wird eine sogenannte Methode mit dem Namen `main` angegeben. Methoden sind Funktionseinheiten die man wieder verwenden kann. Zum Beispiel wäre `Koche Kaffee` eine durchaus öfters wiederverwendbare Funktionseinheit für meinen persönlichen Butler. Mehr zu Methoden wird im Kapitel Methoden behandelt. Auch diese Methode hier ist öffentlich, kann also aus anderen Programmteilen heraus verwendet werden. Insbesondere wird diese Methode von der Java Runtime Environment ausgeführt, wenn die in Bytecode compilierte Datei Hello.class ausgeführt werden soll. Die Methode `main` ist nicht nur öffentlich, sondern auch statisch. Was das bedeutet, kann allerdings auch erst im Kapitel über Methoden erklärt werden. Das dritte Wort in der Zeile heißt void und steht für den Rückgabewert. Auch dies wird im Kapitel über Methoden behandelt. Soviel aber vorweg: Wenn ich mir eine Methode schreibe, die mir beispielsweise etwas ausrechnet so erwarte ich von ihr auch, dass sie mir das Ergebnis dahin liefert, von wo ich diese Methode aus verwendet habe. Die Methode main hingegen ist keine Hilfsfunktion sondern der Ausgangspunkt des ganzen Programms. Wenn die Methode main zu Ende ist, also ihre schließende Klammer erreicht wird, so ist das Programm auch automatisch zu

Ende. Deshalb liefert die Methode main an niemanden etwas zurück und genau dafür steht das Wörtchen void.

Allerdings kann die Methode main durchaus Informationen vom Programmstart übergeben bekommen. Diese Informationen heißen Parameter. So wird beim Start eines Programms mit einer Datei - beispielsweise Doppelklick auf eine Videodatei - dem Programm - in diesem Fall dann der Videoplayer - mit diesem Parameter mitgeteilt mit welcher Datei es gestartet wurde, so dass das Programm gerade dann auch diese Datei bearbeitet. Diese Startparameter werden von der Runtime Environment in eine Liste von Zeichenketten geschrieben. Zeichenketten werden in Java als `String` bezeichnet und werden uns schon im ersten Kapitel wieder begegnen. Die eckigen Klammern hinter dem String bedeuten, dass es mehr als nur eine Zeichenkette sein kann. Dies wird im Kapitel zum Thema Arrays behandelt. Wir nennen diese Liste von Zeichenketten `args`. Das ist eine beliebig gewählte Bezeichnung die für *Arguments* steht. Etwaige an unser Programm übergebene Parameter stehen damit als Liste von Zeichenketten unter der Bezeichnung args zur Verfügung. Da wir im Folgenden aber nichts mehr mit diesem args anstellen, kann uns absolut egal sein, ob und welche Parameter wir übergeben bekommen. Schließlich soll das Programm ja nur „Hello World“ ausgeben und genau das macht die nächste Zeile:

In Zeile drei verwenden wir schon die erste Methode aus der mitgelieferten Klassenbibliothek von Java. Wir übergeben die Zeichenkette „Hello World“ an die Methode println, welche tatsächlich die übergebene Zeichenkette in der Textkonsole ausgibt. Da die Methode println allerdings nicht in der Klasse Hello programmiert wurde, sondern wo ganz anders steht, müssen wir zu Beginn der dritten Zeile erstmal dem Compiler sagen wo er diese Methode finden kann. Tatsächlich befindet sich diese Methode in der Klasse `System` in der Klassenbibliothek und dort wiederrum innerhalb von `out`, daher kommt dieser längere Name zustande. Interessant ist zudem, dass die Zeile mit einem Semikolon beendet wurde. Das liegt daran, dass alle Befehle in Java mit einem Semikolon enden müssen und Zeile drei ist gerade der Befehl „Führe die Methode println mit dem Übergabewert Hello World aus“. Die Zeilen eins und zwei haben kein Semikolon, denn sie enthalten keinen unmittelbaren Java Befehl, sondern definieren eine Java Klasse und eine Methode. Für was diese Begriffe stehen erfahrt ihr in einem späteren Kapitel.

Wie bereits zu Beginn des Abschnittes gesagt, ist es überhaupt nicht notwendig, all diese Details auf Anhieb zu kennen oder zu verstehen. Die zugrunde liegenden Themen werden ausführlich in den Kapiteln dieses Skriptes behandelt und möglicherweise ist es nach einigen Kapiteln nochmals interessant, hierher zurückzukehren und im Detail nachzuvollziehen was da eigentlich steht. Das wichtigste an diesem Abschnitt ist nämlich Zeile eins und zwei plus die schließenden Klammern in Zeile vier und fünf. Denn jedes Java-Programm wird in der Methode main gestartet und daher spielt sich zu Beginn alles im Bereich zwischen Zeile zwei und vier ab. Mit anderen Worten: Jedes Java-Programm enthält diese ersten beiden Zeilen. Es ist zu Beginn allerdings vollkommen ausreichend zu akzeptieren, dass es so ist. Die genaue Bedeutung der ersten beiden Zeilen ist in den ersten Kapiteln erst einmal nebensächlich.

0.7 Übersetzen und Ausführen

Wir haben uns nun angesehen, wie ein Java-Quelltext aussehen kann. Es stellt sich direkt die Frage, wie man vom Quelltext zum Programm kommt und dieses ausführen kann. Wie in den Abschnitten über Programmieren und Java beschrieben, ist es dazu zunächst notwendig, dass der Quelltext durch den Compiler in Bytecode übersetzt wird. Der Bytecode kann dann wiederum von der Java Runtime Environment ausgeführt werden. Dieser Abschnitt beschreibt, wie diese beiden Schritte mithilfe der Kommandozeile umgesetzt werden. Auch

wenn man später durch die Benutzung von IDEs dies per Mausklick erledigt, ist es wichtig diese Schritte auch manuell durchführen zu können.

0.7.1 Die Kommandozeile

Zunächst einmal einige Worte zur Kommandozeile: In der Kommandozeile gibt man den vom Computer auszuführenden Befehl direkt per Tastatur ein. Die Kommandozeile ist komplett textbasiert und war lange vor graphischen Benutzeroberflächen da. Unter Windows ist die Kommandozeile ein Relikt aus älteren Tagen, Stichwort MS-DOS. Unter unixbasierten Bestriebssystemen ist die Kommandozeile nach wie vor ein beliebtes weil sehr mächtiges Werkzeug. Bevor wir uns dem Übersetzen und Ausführen widmen, betrachten wir die grundlegendsten Funktionen der Kommandozeilen unter Windows sowie Linux und MacOS:

Unter Windows startet man die Kommandozeile durch Start -> Ausführen -> cmd. Im nun erscheinenden Fenster befindet man sich stets in einem Ordner auf der Festplatte, dessen Name und Ort links vom > angezeigt wird. Gibt man hier nun zum Beispiel `notepad` ein und drückt Enter, so wird Notepad geöffnet. Mit dem Befehl `cd` wechselt man Verzeichnisse, etwa wechselt man mit `cd C:\Programme` in den Programme-Ordner und mit `cd zweiundvierzig` in den Unterordner namens zweiundvierzig des aktuellen Ordners, sofern ein solcher Unterordner existiert. Mit `cd ..` wechselt man in den übergeordneten Ordner. Mit `dir` kann man sich ansehen, welche Dateien im aktuellen Ordner liegen. Sollten einem diese Befehle neu sein, so empfiehlt es sich, damit ein wenig herum zu experimentieren. Mit der Tabulatortaste kann man zudem angefangene Datei- und Ordnernamen vervollständigen, was einem Tipparbeit spart. Mit `exit` beendet man die Kommandozeile.

Unter unixbasierten Betriebssystemen heißt die Kommandozeile meist Terminal oder Bash. Wenn man diese startet, befindet man sich ebenfalls stets in einem Ordner auf der Festplatte. Auch hier gibt es den `cd` Befehl, um zwischen Ordnern hin und her wechseln zu können. Um die Dateien in einem Ordner anzuzeigen, nutzt man den Befehl `ls`. Die Tabulatortaste kommt hier sogar noch weit mehr zum Einsatz, denn sie kann neben Datei- und Ordnernamen auch Befehle vervollständigen und schnelles doppeltes Drücken der Tabulatortaste zeigt einem die möglichen Alternativen, wenn es mehr als nur einen Datei- oder Ordnernamen gibt, zu dem man das eingetippte vervollständigen könnte. Auch hier beendet man die Kommandozeile mit `exit`.

0.7.2 javac und java

Wenn man nun den Quelltext mit dem Compiler übersetzen möchte, so muss man diesen zunächst in einer Datei abspeichern. Diese Datei muss genau so heißen wie die öffentliche Klasse in ihr und trägt die Dateiendung `.java`. Beginnt der Quelltext also mit `public class Beispiel {`, so muss die Datei also `Beispiel.java` heißen, unter Linux sogar unter Beachtung der Groß- und Kleinschreibung. Sobald diese Datei vorliegt, öffnet man die Kommandozeile und wechselt mit `cd` in den Ordner, in welchem sie liegt. Nun sagt man dem Java-Compiler, welcher auf den Namen `javac` hört, dass er diese Datei übersetzen soll. Dazu gibt man dann `javac Beispiel.java` ein. Sofern keine Fehler im Quelltext sind, erzeugt der Compiler für jede Klasse eine `.class`-Datei, in unserem Fall daher `Beispiel.class`. Einen Text gibt der Compiler nur aus, wenn es Probleme gab; siehe hierzu den nächsten Abschnitt. Sollte man später einmal Quelltexte haben, die aus mehreren Klassen bestehen und sich über mehrere Dateien erstrecken, so merkt der Compiler das von selbst, und bearbeitet diese Dateien gleich von selbst mit.

Diese vom Compiler erzeugten `.class`-Dateien enthalten nun das Programm im Bytecode. Im Gegensatz zu einer `.exe`-Datei werden diese Dateien nicht selbst ausgeführt, etwa durch Doppelklick, sondern mit der Java Runtime Environment gestartet. Die Java Runtime Environment ist in der Kommandozeile über den Befehl `java` zu erreichen. Hierzu ist zu beachten, wenn man nun `Beispiel.class` starten möchte, dass man nur `java Beispiel` eingibt und auf das `.class` verzichtet. Sofern `Beispiel` eine öffentliche Klasse war, die wie oben im Hello World eine main-Methode enthält, wird diese dann ausgeführt und das Programm ab dieser Stelle abgearbeitet bis es entweder selbst zu Ende ist oder ein Fehler auftritt.

An dieser Stelle hat man nun alle grundlegendsten Informationen zusammen. Jetzt wäre die richtige Gelegenheit um das Hello World-Programm aus dem vorhergehenden Abschnitt abzutippen und nach dieser Anleitung zu übersetzen und auszuführen.

0.8 Übersetzungsfehler und andere Tücken

Leider geht am Anfang vieles nicht reibungslos. So bricht der Compiler die Übersetzung des Quelltextes augenblicklich ab, wenn ein Fehler oder auch nur eine Ungenauigkeit im Quelltext ist. Auch bricht die Java Runtime Environment sofort die Ausführung des Programms ab, wenn ein vom Programmierer nicht abgefangener Fehler auftritt, etwa dass durch Null geteilt wird. Zu diesen sogenannten Laufzeitfehlern gibt es im Anhang einen eigenen Abschnitt. In diesem Abschnitt behandeln wir Fehler welche während oder schon vor dem Übersetzen auftreten.

0.8.1 Aus Compilerfehlern schlau werden

Sobald man einen fehlerhaften Quelltext übersetzt, meldet sich der Compiler mit einem Fehler. Glücklicherweise kann einem der Compiler häufig schon eine kurze Auskunft über den Fehler geben. Hat man bei Hello World beispielsweise vergessen, das Semikolon als Ende der dritten Zeile zu setzen, so meldet sich der Compiler unter Windows wie folgt zu Wort:

```
1 C:\Arbeit\Programmierstarthilfe> javac Hello.java
2 Hello.java:3: ';' expected
3         System.out.println("Hello World")
4                                           ^
5 1 error
```

Unter Linux sieht das ähnlich aus:

```
user@Notebook:~$ javac Hello.java
----------
1. ERROR in Hello.java (at line 3)
  System.out.println("Hello World")
                                   ^
Syntax error, insert ";" to complete BlockStatements
----------
1 problem (1 error)
```

Wir sehen also, dass der Compiler uns nicht nur sagt, dass irgend etwas nicht geht, sondern auch in welcher Datei der Fehler vorliegt und insbesondere auch in welcher Zeile, die in der Windowsversion mit einem Doppelpunkt vom Dateinamen abgetrennt ist. Zudem enthält die Fehlermeldung auch eine Information, was der Compiler bemängelt. In diesem Fall hat er eben das Semikolon am Ende der dritten Zeile nicht vorgefunden. Nicht immer sind alle Compilerfehlermedlungen hilfreich. Es kommt öfters vor, dass ein kleiner Fehler wie ein vergessenes Semikolon dafür sorgt, dass der Compiler nicht nur den eigentlichen Fehler angibt, sondern auch die Folgezeilen für fehlerhaft hält. Erhält man also besonders viele Fehler, so ist es sinnvoll zunächst vergessene Zeichen und Tippfehler zu korrigieren und dann nachzusehen, ob andere Fehler damit auch verschwunden sind.

Im Übrigen meldet der Compiler sich nicht nur bei Tippfehlern, also sogenannten syntaktischen Fehlern, welche im Quelltext zu finden sind. Der Compiler macht bereits einige inhaltlichen Prüfungen. So würde der Compiler es nicht zulassen, dass man versucht, eine Methode aufzurufen, welche gar nicht existiert. Zudem kümmert sich der Java-Compiler auch um die sogenannte Typsicherheit. Habe ich in meinem Quelltext etwa eine Ganzzahl-Variable definiert und versuche eine Kommazahl darin zu speichern, würden mir Informationen verloren gehen. Deshalb beschwert sich der Compiler schon mit `possible loss of precision`, sobald es auch nur denkbar wäre, dass eine solche Situation auftritt. So würde der Compiler einen Fehler ausgeben, wenn ich einer Ganzzahl-Variable den Wert der Wurzel einer beliebigen Zahl zuweisen will, denn es gibt Zahlen für die die Wurzel eben eine Kommazahl ist.

0.8.2 Problem: Änderungen werden nicht übernommen

Es passiert einem gerade am Anfang hin und wieder, dass man zwar eine Änderung im Quelltext macht, aber dann feststellt, dass diese nicht im Programm angekommen ist. In diesem Fall sollte man zunächst überprüfen, ob man die neue Version des Quelltextes gespeichert hat und die Änderung nicht nur im offenen Editor steht. Zudem muss man die Datei natürlich jedes mal aufs neue mit dem Compiler übersetzen. Hier bei ist zu beachten, dass der Compiler nur dann eine neue `.class`-Datei erzeugt, wenn die neue Version des Quelltextes keine Fehler aufweist. Ansonsten werden nur Fehlermeldungen ausgegeben und wenn man nun die `.class`-Datei ausführt, startet man noch die vorhergehende Version. Insbesondere mit dem JCreator und der Funktion „Kompilieren und Ausführen“ passiert einem dies öfters. Sollte sich nach einer Quelltextänderung das Programm nicht verändert zeigen, so sollte man nach Compilerfehlern suchen.

0.8.3 Problem: Eine Datei wird nicht gefunden

Wenn die Befehle `java` und `javac` grundsätzlich funktionieren aber das Übersetzen nicht startet oder beim Starten des Programms mit `java` die Fehlermeldung NoClassDefFound-Error erscheint, so sollte man zunächst überprüfen, ob man sich im richtigen Ordner befindet und ob die Dateien, die man verwenden möchte, dort auch befinden. Zudem sollte

man kontrollieren, ob der Aufruf des Compilers mit dem ganzen Dateinamen erfolgt ist, also auch mit dem `.java` am Ende, etwa `javac Beispiel.java`. Im Gegensatz dazu muss man beim Starten der class-Dateien mit `java` die Dateiendung weglassen, etwa `java Beispiel`. Gibt man statt dessen `java Beispiel.class` ein, so versucht Java nicht die Klasse Beispiel zu starten sondern die nicht existente Klasse `class` im nicht existenten Paket `Beispiel`. Daher wird dann der Fehler `NoClassDefFoundError` ausgegeben.

0.8.4 Problem: Der Befehl java und/oder javac wird nicht gefunden.

Wenn die Kommandozeile meldet, dass der Befehl `java` bzw. `javac` nicht gefunden wird, so kann das daran liegen, dass auf dem verwendeten Computer keine Java Runtime Environment (JRE) bzw. kein Java SE Development Kit (JDE) installiert ist. Werden die Befehle trotzdem nicht gefunden, obwohl ein JRE bzw JDK installiert ist, so kann die Ursache sein, dass der Ordner in dem sich die Dateien zu den Befehlen `java` bzw `javac` befinden nicht in der sogenannten Path-Umgebungsvariable eingetragen ist. In dieser Umgebungsvariable stehen die Ordner, in welcher die Kommandozeile die Befehle sucht.

Unter Windows XP fügt man den Ordner wie folgt zur Path-Variable hinzu: Als Benutzer mit Administratorrechten wählt man Start -> Einstellungen -> Systemsteuerung -> System -> Reiter Erweitert -> Umgebungsvariablen. Hier fügt man den Pfad zu java/javac mit Semikolon getrennt an die bestehende Path-Variable an. Auf keinen Fall darf man diese Variable nur ersetzen. Der Pfad zu java/javac ist häufig von der Art `C:\Programme\jdk-<versionsnummer>\bin`; der Eintrag für Path sollte dann diese Form haben:

```
bisherige_Einträge;C:\Programme\jdk-<versionsnummer>\bin
```

0.9 Kommentieren, Einrücken und Formatieren

Nachdem wir jetzt soviel gelernt haben, ist es sinnvoll uns Notizen machen zu können - insbesondere im Quelltext. Notizen im Quelltext werden Kommentare genannt und können in Java in zwei verschiedenen Arten eingebaut werden. Zuerst schauen wir uns einzeilige Kommentare an:

```
1 tu_irgendwas(); // Dies ist ein Kommentar
2 // und wird vom Java-Compiler ignoriert
```

Der Rest der Zeile wird ab dem doppelten Schrägstrich ignoriert, in der nächsten Zeile geht es aber ganz normal weiter. Möchte man längere Kommentare machen, so kann man auch einen Kommentarblock erstellen, das geht dann so:

```
1 /* Das hier ist ein Kommentarblock, er wird durch
2 Schraegstrich Stern begonnen. Auch hier steht noch ein
3 Kommentar, weil der Block noch nicht geschlossen ist.
4 Schliessen kann man einen Kommentarblock so: */
```

Der Kommentarblock kann auch nur über einen Teil einer Zeile gehen. Es wird nur genau der Part zwischen /* und */ ignoriert. Im Übrigen sind Kommentarblöck nicht schachtelbar. Steht ein /* innerhalb eines Kommentarblocks so wird es als Bestandteil des Kommentars angesehen und das erste */ schließt den gesamten Block. Ein weiteres vorhandenes */ erzeugt dann einen Fehler, weil zu ihm kein /* gefunden wird.

Nachdem wir nun wissen, wie man kommentiert, müssen wir auch noch darüber sprechen, warum man kommentiert. Es gibt zwei häufige Verwendungsgründe für Kommentare: Zum einen ist das das Auskommentieren von Quelltextzeilen. Möchte man testweise wissen, wie sich das Programm verhält, wenn eine gewisse Anweisung nicht ausgeführt wird, so stellt man einfach // vor die entsprechende Zeile. So muss man entsprechende Zeilen nicht löschen, nur weil man sie testweise einmal nicht braucht. Natürlich kann man mit /* und */ auch viele Anweisungen auf einmal ignorieren lassen.

Die wichtigere Anwendung von Kommentaren ist allerdings das Hinterlassen von Notizen, wie der Quelltext funktioniert und warum Etwas an einer Stelle so gemacht wurde, wie es eben da steht. Zu Beginn unterschätzt man die Wichtigkeit von Kommentaren komplett. Man darf aber nicht vergessen, dass man nicht nur für andere kommentiert sondern hauptsächlich für sich selbst. Schon nach ein paar Tagen kann man komplett vergessen haben wie eine komplizierte Methode arbeitet und ist auf die eigenen Kommentare angewiesen. Beim Kommentieren ist es wichtig, dass man nicht das Was sondern das Warum kommentiert. So ist es überhaupt nicht nützlich, wenn man etwa neben eine Addition als Kommentar schreibt „hier werden zwei Variablen addiert“, denn dies erkennt man von selbst. Viel besser ist es, wenn man schreibt, warum die Variablen addiert werden, also etwa „die Summe ist die Anzahl aller Einträge in der Liste, weiter muss nicht eingelesen werden“. Nur durch solche Kommentare kann man später nachvollziehen, warum man die Addition überhaupt macht. Sehr hilfreich ist es zudem, wenn man die verwendeten Variablen nicht `a` oder `b` nennt, sondern zum Beispiel `zeilennummer` oder `max_anzahl_benutzer`. Durch diese sogenannten „sprechenden Variablen“ wird der Quelltext wesentlich nachvollziehbarer. Die zusätzliche Tipparbeit wird durch weniger Frust bei der Fehlersuche fürstlich entlohnt.

Eine weitere Maßnahme zur Übersicht ist das Einrücken und Formatieren des Quelltextes. Schon beim Hello World haben wir kennen gelernt, dass man alles was innerhalb eines Blocks passiert, um eine Ebene tiefer einrückt. Hierbei ist eine Einrücktiefe von vier Leerzeichen üblich. Das konsequente Einrücken sorgt dafür, dass man mehr Überblick über den Quelltext hat, da schon durch die Einrücktiefe klar ist, welche Quelltextzeilen von welchen abhängen. Zudem kommt es gerade am Anfang häufig vor, dass man Klammern vergisst. Dies wird dann durch das konsequente Einrücken wesentlich schneller sichtbar. Man kann minutenlang Fehler suchen, die man durch ordentlich eingerückten Quelltext innerhalb von Sekunden entdecken würde.

Zudem sollte man alles was man an Quelltext schreibt stets nach den selben Regeln formatieren, etwa dass eine schließende geschweifte Klammer immer in einer neuen Zeile steht und sie die selbe Einrücktiefe hat wie das Element, zu dem die öffnende Klammer gehört. Ob man sich bei diesen Formatierungsregeln gänzlich an unsere Beispiele hält oder es anders macht, ist nebensächlich. Wichtig ist, dass man es konsequent macht. Diese Disziplin wird damit belohnt, dass einem Tippfehler und vergessene Zeichen sofort ins Auge springen, wenn man sich erstmal an eine Formatierungsart gewöhnt hat.

1 Datentypen

In jeder Sprache gibt es Arten von Elementen, in Sprachen für Menschen nennen wir z.B. verschiedene Wörter Verben und Nomen, in Programmiersprachen haben wir außer Schlüsselworten der Sprache selbst verschiedene Arten von Daten mit denen gearbeitet wird.

1.1 Konventionen und Ausgabe

In Java wie in allen imperativen Programmiersprachen verwendet man viele Variablen. Diese brauchen immer einen Namen. Aus der Schule sind wir gewohnt sehr kurze Namen wie x, y oder vielleicht ϕ zu verwenden. In Programmen macht dies keinen Sinn, denn es wird schwer den Sinn noch zu erkennen, wenn so kurze Namen verwendet werden. Am besten ist es sprechende Namen zu nehmen: `datum`, `sekunden` oder `seitenlayout` sind einfacher zu verstehen.

Namen dürfen beliebig lang sein und müssen mit einem Buchstaben beginnen. Das sind `A-Z`, `a-z` und auch _ und `$`. Nach dem ersten Buchstaben dürfen sie auch Zahlen enthalten. Der Unterstrich und das Dollarzeichen sollen nicht verwendet werden, sie wurden früher für automatisch erzeugten Code genutzt und sind nur aus historischen Gründen erlaubt.

Eine allgemeine Konvention ist, alle Variablen mit einem kleinen Buchstaben beginnen zu lassen und wenn es ein Satz wird die Anfangsbuchstaben der Worte groß zu schreiben. Also z.B. `printServer12`, `userInterface` oder auch `iteratorFactory`.

Ausgaben werden mit dem Befehl `System.out.print()` gemacht. Was zwischen den Klammern steht wird ausgegeben. Schreibt man `System.out.println()` so wird zusätzlich ein Zeilenumbruch ausgegeben.

```
1 System.out.println("Hallo Welt");
2 // gibt Hallo Welt aus
```

1.2 Variablen und Zahlen

Damit unsere Programme sinnvolle Dinge tun benötigen sie meist Daten mit denen sie etwas machen können. Diese Daten werden in einem Speicherbereich in einem Programm abgelegt.

Bevor man mit einer Variable arbeiten kann muss man sie erst einmal deklarieren, d.h. anlegen:

```
1 int meinErsterInteger;
```

Der Typ ist hier `int`. Es gibt ganze Zahlen, die wir mit dem Wort `int` markieren und Fließkommazahlen, für die wir `double` nehmen. In die Variable kommt eine Zahl, in dem wir eine Zuweisung machen:

```
1 meinErsterInteger = 5;
```

Bei einer Fließkommazahl würde im Programm `5.0` stehen. Deklaration und Zuweisung kann auch in einer Zeile geschehen:

```
double meinErsterDouble = 5.0;
```

Im Gegensatz zu Variablen in der Mathematik muss eine Variable in einem Programm immer einen Wert enthalten. Wenn man eine Variable deklariert und ihr nichts zuweist ist sie unbelegt und kann nicht verwendet werden, denn der Compiler entdeckt diesen Fehler.

Rechnen mit Zahlen

Mit Zahlen kann nun gerechnet werden. Es gibt die gewohnten Rechenoperationen +, −, ⋆ und /. In unserem Beispiel rechnen wir eine einfache Formel aus.

```
int i = 6;
System.out.println(5 * 3 + (i / 2));
// gibt 18 aus
```

Sehr häufig muss man in einem Programm eine Variable immer wieder um 1 erhöhen. Das kann man auf drei Arten machen:

```
int i = 0;
i = i + 1; // i ist 1
i += 1; // i ist 2
i++; // i ist 3
```

Durch − wird eine Variable entsprechend um eins verringert. Die Schreibweisen += und ++ sind nur Abkürzungen.

Modulo

Zusätzlich zu den üblichen Grundrechenarten gibt es die Modulo-Operation `%`. Diese entspricht dem Teilen mit Rest aus der Grundschule und liefert den Rest beim Teilen.

```
System.out.println(5 % 2);
// gibt 1 aus
```

Beim Teilen von ganzen Zahlen (`int`) bekommt man nur die Anzahl ganzer Teiler als Ergebnis.

```
System.out.println(5 / 2);
// gibt 2 aus
```

Damit bleibt man bei allen Rechenoperationen auf ganzen Zahlen im selben Zahlenbereich.

1.3 Wahrheitswerte

Bisher haben wie `int` für ganze Zahlen und `double` für rationale Zahlen kennen gelernt. Ein wichtiger Typ ist `boolean`, der nur die Wahrheitswerte wahr oder falsch (`true` oder `false`) aufnehmen kann. Immer wenn man einen Vergleich macht, kommt als Ergebnis ein boolean heraus. Im nächsten Kapitel werden wir sehen wie man damit den Ablauf eines Programms steuern kann.

<	kleiner
<=	kleiner oder gleich
==	gleich
>=	größer oder gleich
>	größer
!=	ungleich

Es ist auch möglich, mehrere Vergleiche auf einmal zu machen. Wenn wir wissen wollen ob das Alter einer Person zwischen 18 und 21 liegt, können wir das so bestimmen:

```
1 boolean ergebnis = (alter < 21) && (alter > 18);
```

Es gibt also Befehle für Arithmetik mit Wahrheitswerten wie mit Zahlen. Nur sind es weniger.

- logisches Nicht: `!a` ergibt `false`, wenn `a` wahr ist und `true`, wenn `a` `false` ist
- logisches Und: `a && b` ergibt `true`, wenn sowohl `a` als auch `b` wahr sind
- logisches Oder: `a ||b` ergibt `true`, wenn mindestens einer der beiden Ausdrücke `a` oder `b` wahr ist
- exklusives Oder: `a ^b` ergibt `true`, wenn beide Ausdrücke einen unterschiedlichen Wahrheitswert haben

```
1 System.out.println((!false) && true );
2 // gibt true aus
```

1.4 Strings

Auch mit Text muss oft in Programmen gearbeitet werden. Dazu gibt es den Datentyp `String`. Wir beschränken uns vorerst auf eine sehr einfache Operation, nämlich das Zusammenhängen von Strings.

```
1 String text = "Hallo";
2 String mehr = text + " Welt";
```

Der String „mehr“ enthält nun den Text „Hallo Welt“.

1.5 Der Rest: byte, short, char, long, float

Es gibt noch mehrere Datentypen, auf die noch nicht eingegangen worden ist. Für Zahlen gibt es neben `int` auch `long`. Die Wertebereiche dieser Typen sind verschieden, ein `long` kann 64 Bit ganze Zahlen enthalten, ein `int` nur 32 Bit. Bei Fließkommazahlen ist es ähnlich, hier gibt es noch `float`s, die auch 32 Bit haben und damit weniger Werte enthalten können als `double` mit 64 Bit. Bei der Zuweisung muss man Java sagen was für einen Typ eine Zahl hat, falls man `long` oder `float` verwendet, indem man ein `l` oder `f` an die Konstante anhängt.

```
1 long grosseGanzeZahl = 21474836481;
2 float nichtSoooGross = 1.0f;
```

Es gibt auch Datentypen für Zahlen die deutlich kleiner als `int` sind. In eine `short`-Variable passen 16 Bit Werte, in ein `byte` nur 8 Bit.

Ein String wird immer mit doppelten Anführungszeichen angegeben. Manchmal braucht man ein einzelnes Zeichen, dann nimmt man ein `char`.

```
1 char buchstabe = 'a';
```

Dabei müssen einfache Anführungszeichen verwendet werden.

1.6 Typecasts

Nun kann es sein, dass man mit verschiedenen Datentypen auf einmal arbeiten muss. Z.B. möchte man für ein Zeichen das zehnte Zeichen danach bestimmen. Das heißt zu einem Buchstaben wie „c“ 10 zu addieren. Das geht in Java so:

```
1 char zeichen = 'c';
2 int ganzzahl = 10;
3 zeichen += ganzzahl;
```

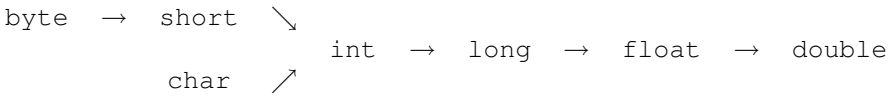
Damit nachher in `zeichen` immer noch ein `char` ist und keine Zahl muss Java den Typ automatisch anpassen. Diese Anpassung nennt man Typecast. In Java werden ganz automatisch Typecasts ausgeführt, sogenannte „implizite Typcasts“, wenn unterschiedliche Datentypen miteinander verrechnet werden. Außerdem werden implizite Typecasts verwendet, wenn eine Zahl in ihrem Wertebereich erweitert wird, zum Beispiel, wenn eine ganze Zahl (Typ `int`) einer Fließkommazahlen (Typ `double`) zugewiesen wird.

```
1 double kommazahl = ganzzahl;
```

Solche impliziten Typecasts nimmt Java nur vor, wenn dadurch der Wertebereich erweitert wird, nicht aber in Gegenrichtung. Im folgenden Beispiel passt maximal die Zahl 127 in das `byte`.

```
1 byte winzig = 1000; // Fehler
```

Das Schaubild unten zeigt mögliche implizite Typecasts in Pfeilrichtung an. Es ist natürlich auch möglich, einige Stufen zu überspringen und beispielsweise `char` direkt auf `double` zu casten.



Manchmal möchte man gegen die Pfeilrichtung casten. Dafür gibt es die Möglichkeit, den Zieldatentyp explizit anzugeben, daher auch der Name „expliziter Typcast“. Man trägt diesen einfach in Klammer ein. Beim expliziten Casten ist Vorsicht geboten, denn der Wertebereich wird unter Umständen verkleinert. Wenn beispielsweise eine Fließkommazahl auf `int` gecastet wird gehen die Nachkommastellen verloren (kein Runden, es wird abgeschnitten).

```
1 int neue_ganzzahl = (int) 3.1415; // ist also 3
```

2 Kontrollstrukturen

Bis jetzt haben wir die grundlegenden Datentypen kennen gelernt und mit ihnen gerechnet. Als nächstes führen wir Fallunterscheidungen ein, denn oft müssen je nach Bedarf einmal die einen, ein anderes Mal andere Anweisungen durchgeführt werden.

2.1 Blöcke

In Java kann man mehrere Anweisungen die zusammen gehören in einem Block zusammenfassen. Ein Block verhält sich dann wie eine einzelne Anweisung, die mehrere Schritte macht.

```
1 {
2   Anweisung;
3   Anweisung;
4 }
```

2.2 if-Anweisung

Nun kommen wir zu den Fallunterscheidungen. Wir unterscheiden mehrere Fälle. Am einfachsten ist der Fall, bei Bedarf etwas zu machen und sonst einfach nicht. Im dem kleinen Programmteil unten wird erst Anweisung1 ausgeführt. Dann wird der boolsche Ausdruck in der Klammer angesehen und wenn er wahr ist wird AnweisungX ausgeführt. Anweisung2 wird dann wieder in jedem Fall ausgeführt.

```
1 Anweisung1;
2 if(boolscher-Ausdruck)
3   AnweisungX;
4 Anweisung2;
```

An einem konkreten Beispiel sieht das so aus:

```
1 if(true)
2   rabatt = 20;
```

Es macht natürlich keinen Sinn dort je nach Fall `true` oder `false` hin zu schreiben und jedes Mal das Programm neu zu kompilieren. Statt dessen können wir einen Vergleich einsetzen, denn der liefert als Ergebnis einen boolean.

```
1 if(alter < 14)
2   rabatt = 20;
```

Die Variable `alter` wird vorher im Programm gesetzt oder vom Benutzer eingegeben. Je nach Alter bekommen Leute dann Rabatt.

Statt einer einzelnen Anweisung können wir nun auch einen ganzen Block einsetzen um mehr Dinge erledigen zu können.

```
if(alter < 14) {
  rabatt = 20;
  druckenMitBildchen = true;
  familientarif = true;
}
```

Der Block beginnt direkt nach dem `if` und endet an der Klammer `}`. Im Vorfeld sind natürlich die Variablen `rabatt`, `druckenMitBildchen` und `familientarif` sinnvoll definiert worden.

Ein häufiger Fehler ist, direkt nach einem `if` ein Semikolon zu schreiben. Das `if` erwartet nach dem boolschen Ausdruck eine Anweisung, die mit einem Semikolon abgeschlossen wird. Läßt man die Anweisung zwischen Klammer und Semikolon weg, so ergänzt Java automatisch eine leere Anweisung, die nichts tut. Die Anweisung nach dem „falschen“ Semikolon wird dann immer ausgeführt, da sie nun nichts mehr mit dem `if` zu tun hat.

2.3 if mit mehreren Möglichkeiten

Hin und wieder stößt man an Stellen, an denen ein einzelnes `if` nicht ausreicht. Zum Beispiel möchten wir ein Programm schreiben, dass je nach Alter ausgibt was für Fahrzeuge man fahren darf. Es sollen je nach Alter kein bis alle Möglichkeiten ausgegeben werden. Dafür können wir natürlich mehrere `if`-Abfragen machen:

```
if(alter >= 9)
  System.out.println("darf Rad auf Strasse fahren");
if(alter >= 15)
  System.out.println("darf Mofa fahren");
if(alter >= 18)
  System.out.println("darf Auto fahren");
```

Falls man zwei verschiedene Fälle hat die sich ausschließen, kann man zu der Bedingung im `if` für beide Fälle Anweisungen angeben. Durch einen `else`-Teil geben wir an, was geschehen soll, falls der boolsche Ausdruck des `if` falsch ist.

```
if(boolscher-Ausdruck)
  AnweisungFallA;
else
  // falls boolean false war
  AnweisungFallB;
```

Sowie die Möglichkeit, mehrere Fälle abzufragen.

```
if(boolscher-Ausdruck)
  AnweisungFallA;
else if(boolscher-Ausdruck)
  AnweisungFallB;
else
  AnweisungFallC;
```

Das folgende Beispiel zeigt uns, wie man bestimmen kann ob eine Zahl positiv, negativ oder Null ist. Man fragt also drei Fälle ab die sich gegenseitig ausschließen.

```
if(zahl > 0)
  System.out.println("Zahl positiv");
else if(zahl < 0)
  System.out.println("Zahl negativ");
else
  System.out.println("Zahl ist Null");
```

Das Dangling-else-Problem

In vielen Sprachen kann man Quelltext schreiben der mehrdeutig ist. Es stellt sich die Frage wie die folgende Zeile zu verstehen ist.

```
if (a == 1) if (b == 1) Anweisung1; else Anweisung2;
```

Es gibt zwei Möglichkeiten. Java versteht diese Zeile so, dass das `else` zum letzten (bzw. inneren) `if` gehört. Der vollständig geklammerte Quelltext dazu sieht so aus:

```
if (a == 1) {
  if (b == 1) {
    Anweisung1;
  } else {
    Anweisung2;
  }
}
```

Die andere Möglichkeit es zu interpretieren wäre, dass das `else` zum ersten (bzw. äußeren) `if` gehört. Das bedeutet, dass man Klammern setzen muss, wenn Java den Quelltext so verstehen soll.

```
if (a == 1) {
  if (b == 1) {
    Anweisung1;
  }
} else {
  Anweisung2;
}
```

Es empfielt sich also, auf uneindeutige Konstruktionen zu verzichten, und der Lesbarkeit zur Liebe ein paar Klammern mehr zu setzen als nötig.

2.4 switch-Anweisung

Für Mehrfachverzweigungen ist oft einfacher statt vielen if-else ein switch zu schreiben, mit dem mehrere Fälle unterschieden werden können.

```
switch(variable) {
  case Konstante1:
    Anweisungen;
    break;
  case Konstante2:
    Anweisungen2;
```

```
    break;
    .
    .   // mehr Faelle
    .
  case KonstanteN:
    AnweisungenN;
    break;
  default:  // was sonst geschehen soll
    Anweisungen;
}
```

Je nach dem was für eine Zahl in der Variable steht wird der entsprechende Fall ausgeführt. Durch das `break` danach geht es dann nach dem `switch` weiter. Falls kein Fall zutrifft wird getan was bei `default` steht.

Es ist möglich das `break` wegzulassen. Dann werden nach unten so lange die nächsten Anweisungen ausgeführt bis ein `break` kommt oder das `switch` zu Ende ist. Damit kann man mehrere Fälle vereinen wie man auch gleich sehen kann.

Das Beispiel nimmt eine Zahl und gibt je nach Fall eine kleine Beschreibung aus.

```
// n sei ein int
switch(n) {
  case 1:
    System.out.println("Die Zahl ist 1.");
    break;
  case 2:
  case 4:
  case 8:
    System.out.println("Die Zahl ist 2, 4, oder 8.");
    System.out.println("(Eine Zweierpotenz!)");
    break;
  case 3:
  case 6:
  case 9:
    System.out.println("Die Zahl ist 3, 6, oder 9.");
    System.out.println("(Ein Vielfaches von 3!)");
    break;
  case 5:
    System.out.println("Die Zahl ist 5.");
    break;
  default:
    System.out.print("Die Zahl ist 7 oder ");
    System.out.println("ausserhalb des Bereichs 1 - 9.");
}
```

3 Schleifen

Oft sollen Computer Dinge erledigen, die sich wiederholen. In manchen Fällen wäre dies dadurch zu erreichen, immer wieder die gleiche Anweisung einzutippen. Aber diese Lösung wäre nicht unbedingt elegant und auch nur dann möglich, wenn wir schon während des Programmierens wüssten, wieviele Durchläufe wir benötigen (was im Allgemeinen nicht der Fall ist). Schleifen schaffen Abhilfe: Mit ihnen kann man einen bestimmten Codeblock so oft ausführen lassen, wie man möchte, bzw. die *Schleifenbedingung* erfüllt ist.

3.1 for-Schleifen

Am einfachsten ist es, wenn bekannt ist, wie oft eine Anweisung ausgeführt werden soll. Dann verwendet man beispielsweise eine `for`-Schleife, auch *Zähl-Schleife* genannt.

```
1  for(int i=1; i<10; i=i+1)
2    Anweisung;
3
4  // oder mit einem Block
5  for(int i=1; i<10; i=i+1) {
6    Anweisungen;
7  }
8
9  // oder in Kurzschreibweise
10 for(int i=1; i<10; i++)
11   Anweisung;
```

Diese Schleife zählt von 1 bis 9 und führt dabei jedes mal die Anweisungen aus. Eine Besonderheit bei diesem Schleifentyp ist der Inhalt der Klammer nach dem `for`: Anders als bei allen anderen Schleifentypen besteht er nicht aus einem einzigen Teil, sondern aus drei Teilen, die jeweils durch ein `;` getrennt werden. Der erste Teil (im obigen Beispiel `int i=1`) deklariert und initialisiert die so genannte *Zählvariable*. Der zweite Teil (`i<10`) gibt die Bedingung an, die gelten muss, damit die Schleife ausgeführt wird und der dritte und letzte Teil (`i=i+1`) legt fest, wie die Zählvariable erhöht, oder, wenn rückwärts gezählt werden muss, verkleinert werden soll. Je nach Bedarf kann man, statt in Einerschritten, natürlich auch in Zweier- oder beliebigen anderen Schritten zählen, indem man den dritten Anweisungsteil in der Klammer entsprechend anpasst.

3.2 while-Schleifen

Falls man nicht genau wissen kann, wie oft etwas getan werden soll gibt es in der Regel eine Bedingung, die gilt so lange noch etwas getan werden muss. Oder in Code gesagt:

```
1  while(Bedingung) {
2    Anweisungen;
3  }
```

An einem Beispiel soll dies nun klarer werden. Man lernt so lange wie man noch Prüfungen schreiben muss. Sobald man keine Prüfungen mehr hat, muss man nicht mehr für sie büffeln.

```
boolean pruefungen = true;
while(pruefungen) {
  // lernen
  // hier pruefungen auf false setzen
}
```

3.3 do-while-Schleife

Es kann nun vorkommen, dass ein Programmstück ausgeführt werden muss, bevor man die Bedingung im `while` prüft. Mit der `while`-Schleife müsste man dann Code doppelt schreiben.

```
Anweisung1;
while(Bedingung) {
  Anweisung1;
}
```

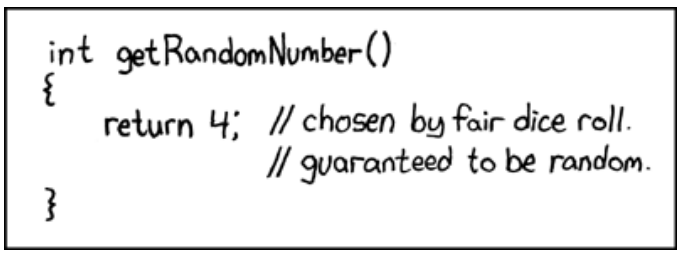
Das ist nicht sinnvoll, denn identischer Code, der mehrfach vorhanden ist, kann auch mehr Fehler enthalten. Zum Glück gibt es aber dafür eine Konstruktion, die uns die Verdopplung erspart.

```
do {
    Anweisung1;
} while(Bedingung);
```

Bei der `do-while`-Schleife wird zuerst der Teil nach dem `do` abgearbeitet, bevor die Bedingung überprüft wird. Im folgenden Beispiel verwenden wir eine Methode `wuerfle()` um uns eine Zufallszahl erzeugen zu lassen, die zwischen 1 und 6 liegt. Falls es keine 4 war, soll weiter gewürfelt werden.

```
int ergebnis;
do {
  ergebnis = wuerfle();
} while(ergebnis != 4);
```

Der folgende Quelltext macht das Selbe, nur etwas effizienter:



Randall Munroe (`http://xkcd.com/221`)

Häufiger Fehler

Wie beim `if` darf auch bei Schleifen kein Semikolon so gesetzt sein, dass dadurch der Körper der Schleife nicht ausgeführt wird. Im Beispiel wird eine `for`-Schleife unwirksam.

```
for(int i=1; i<10; i++); // FEHLER: leere Anweisung 10x
  Anweisung; // wird 1x ausgefuehrt
```

Im folgenden Beispiel mit einer `while`-Schleife sorgt das falsch gesetzte Semikolon sogar für eine Endlosschleife.

```
while(Bedingung); { // FEHLER: Endlosschleife
  Anweisung; // wird NIE ausgefuehrt
}
```

Bei einer `do-while`-Schleife muss nach dem `while` ein Semikolon gesetzt werden. Der auszuführende Block steht nämlich schon davor zwischen `do` und `while`.

3.4 Verschachtelte Schleifen

Um kompliziertere Aufgaben zu lösen kann es sein, dass man mehrere Schleifen ineinander schachteln muss. Wir möchten als Beispiel ein Dreieck aus Sternchen ausgeben.

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

Es bietet sich an eine Schleife zu machen, die 5 Zeilen erzeugt. Doch wie können je Zeile die richtige Anzahl an Sternchen ausgegeben werden? Es fällt auf, dass es so viele Sternchen sind, wie die Zeile von oben als Nummer links stehen hat. In der ersten Zeile eins, in der zweiten zwei und schließlich in der fünften fünf. Wir können dazu einfach noch eine Schleife machen, welche die Zählvariable der äußeren Schleife verwendet.

```
for(int i=1; i<6; i++) { // Zeilennummer
  for(int j=1; j<=i; j++) { // eine Zeile mit i Sternen
    System.out.print('*');
  }
  System.out.println(); // und der Zeilenumbruch
}
```

4 Arrays

Beginnen wir mit einer Definition[1]: Ein Array ist eine Datenstruktur, in der eine Reihe von Elementen vom gleichen Typ über einen Index zugegriffen werden kann. Die Größe lässt sich nicht ändern.

4.1 Eindimensionale Arrays

Ein Array kann man sich wie ein Regal mit vielen Fächern vorstellen. Es gibt eine feste Anzahl an Fächern, und jedes Fach hat eine Nummer. In jedes Fach kann nur eine einzelne Sache gelegt werden, und all diese Sachen müssen vom selben Typ sein.

Möchte man zum Beispiel alle Teilnehmer einer Vorlesung erfassen, so kann man dazu ein Array anlegen, das so viele Elemente (Fächer) enthält, wie es Teilnehmer gibt.

```
1 int[] teilnehmer; // Variable deklarieren
2 teilnehmer = new int[120]; // Array mit 120 Teilnehmern
3
4 // das gleiche fuer 5000 Leute in einem Schritt
5 int[] teilnehmer = new int[5000];
```

Der Typ der Variablen ist hier `int[]`, das heißt ein Array aus Daten vom Typ `int`. Jetzt sollte auch endlich klar sein, was `String[] args` heißt, was wir immer hinter `main` schreiben, nämlich einfach eine Variable vom Typ `String[]` mit Namen args, also ein Array bestehend aus `String`s.

Um das Array anzulegen, schreibt man das Schlüsselwort `new`, den Datentyp, hier also `int` und die Anzahl der Elemente, die dieses Array groß sein soll.

Auf die einzelnen Elemente eines Arrays greift man nun über ihren Index zu. Der Index entspricht einfach der Nummer des Fachs im Regal. Java nennt das erste Fach aber 0 und **nicht** 1, also aufpassen, denn dies ist ein beliebter Fehler. Wir füllen nun die fiktiven Matrikelnummern einiger Teilnehmer in das Array.

```
1 teilnehmer[0] = 123456;
2 teilnehmer[1] = 123457;
3 teilnehmer[2] = 123458;
4 // usw.
```

Wenn ein Array nicht so viele Elemente hat, kann man sie beim Erzeugen des Arrays auch direkt angeben. Dann muss man die Größe nicht hinschreiben, Java ermittelt sie aus der Anzahl der Elemente.

```
1 int[] primzahlen = {2, 3, 5, 7, 11, 13};
```

Um auf die Elemente zuzugreifen, spricht man sie mit dem Index an. Hier müssen wir immer daran denken, dass der Index bei Null beginnt!

[1] Nach `http://en.wikipedia.org/wiki/Array`

```
1 // gibt 7 aus
2 System.out.println(primzahlen[3]);
```

Das geht natürlich auch mit einer Schleife, denn wir wollen ja nicht nur die Tipparbeit der vielen Variablen vermeiden, sondern auch weniger Ausgaben schreiben.

```
1 for(int i=0; i<6; i++){
2   System.out.println(primzahlen[i]);
3 }
```

Manchmal möchten wir einfach alle Elemente ausgeben oder mit allen Elementen etwas berechnen. Da wir vielleicht im Laufe der Zeit die Größe eines Arrays in unserem Programm von Version zu Version ändern wollen, müssten wir jedes mal überall in den Schleifen die Zahl austauschen bis zu der gezählt wird. Um das zu vermeiden, kann man mit `arrayname.length` die Anzahl der Array-Einträge ermitteln.

```
1 for(int i=0; i<primzahlen.length; i++){
2   // von 0 bis 5, denn primzahlen.length == 6
3   System.out.println(primzahlen[i]);
4 }
```

Eine Warnung noch zur Arbeit mit Schleifen und Arrays: Der Index beginnt bei 0, je nach Abbruchbedingung der Schleife kann es sein, dass man nicht alle Elemente bearbeitet. `arrayname.length` liefert für die Primzahlen oben 6 als Ergebnis. Würde im Vergleich `i <= primzahlen.length` stehen statt `i < primzahlen.length`, so würde Java versuchen eine siebte Primzahl auszugeben. Unser Array endet aber schon mit der sechsten und deshalb würde eine Fehlermeldung ausgegeben werden und das Programm abstürzen.

Sobald es darum geht, größere Mengen an Daten des selben Typs zu verarbeiten, sind Arrays unglaublich praktisch. Und vor dieser Aufgabe steht man häufiger als vielleicht vermutet, so z.B. beim Ausrechnen von Reihen aus der Mathematik, dem Zeichnen von Kurven, Erfassung von Leuten (Accounts, Bankkonten, Bibliothek, ...).

4.2 Mehrdimensionale Arrays

Man kann auch Arrays verwenden, die mehr als nur eindimensional sind. Wenn wir z.B. eine 3×3-Matrix benötigen, legen wir diese wie folgt an und befüllen sie.

```
1  int[][] matrix = new int[3][3];
2  matrix[0][0] = 1;
3  matrix[0][1] = 2;
4  matrix[0][2] = 3;
5  matrix[1][0] = 4;
6  matrix[1][1] = 5;
7  matrix[1][2] = 6;
8  matrix[2][0] = 7;
9  matrix[2][1] = 8;
10 matrix[2][2] = 9;
```

Wir können dann auf die Elemente mit `matrix[i][j]` zugreifen. Um mit zweidimensionalen Arrays zu arbeiten, benötigt man in der Regel ineinander geschachtelte Schleifen. Um z.B. alle Elemente der obigen Matrix auszugeben, schreibt man Folgendes:

```
1  for(int i=0; i<matrix.length; i++){ // zeilenweise
2    for (int j=0; j<matrix[i].length; j++) { // spaltenweise
3      System.out.print(" " + matrix[i][j]);
4    }
5    System.out.println(); // Zeilenumbruch
6  }
7  // Ausgabe:
8  // 1 2 3
9  // 4 5 6
10 // 7 8 9
```

Hier sieht man gut, dass mehrdimensionale Arrays in Java einfach Arrays von Arrays sind. Das bedeutet, die Matrix enthält ein Array dessen Elemente die Zeilen sind. Jede Zeile besteht dann wieder aus einem eindimensionalen Array, das die eigentlichen Werte enthält. Um die Länge einer Zeile zu erhalten, verwenden wir somit wieder die Arraylänge des Arrays der Zeile, z.B. `matrix[0].length`, denn `matrix[0]` ist ja wieder ein Array.

5 Methoden

Um die wachsende Komplexität länger werdender Programme zu beherrschen und den Quelltext übersichtlich und strukturiert zu halten, verwendet man sogenannte Methoden. Diese sind Einheiten von Quelltext, die wir einmal schreiben und dann an verschiedenen Orten im Programm verwenden können.

Methoden stellen das Konzept der Funktionen aus der Mathematik dar: Nehmen wir an, wir möchten das Quadrat einer Zahl ausrechnen, so würden wir das so schreiben:

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

Dabei kann man nun noch definieren, für welche Zahlen diese Funktion gelten soll, zum Beispiel nur auf natürlichen Zahlen. Dann bildet sie auch in die natürlichen Zahlen ab ($\mathbb{N} \to \mathbb{N}$). Wir können nun diese Methode in Java deklarieren:

```
1 public static int f(int x){
2   return x * x;
3 }
```

Diese Methode heißt `f` und hat einen Parameter vom Typ `int` namens `x`.

Direkt vor dem Methodennamen steht der so genannte Rückgabetyp (in diesem Fall `int`). Eine Variable dieses Typs wird nach einem Aufruf mit `return` an die aufrufende Stelle zurückgegeben. Ein Aufruf der Methode könnte so in der `main`-Methode stehen:

```
1 int ergebnis = f(5); // ergebnis ist nun 25
```

In diesem Fall wird `f` ausgeführt, die Methode gibt dann eine Zahl vom Typ `int` zurück und diese wird als Ergebnis der Variablen `ergebnis` zugewiesen.

Vor dem Methodennamen stehen noch zwei Worte. Das Schlüsselwort `static` benötigt man, damit die Methode aus der `main`-Methode aufgerufen werden kann. Das Wort `public` könnte auch weggelassen werden. Beide Attribute sind später für die Objektorientierung wichtig, und wir werden sie dann erklären.

Soll die Methode von reellen auf reelle Zahlen ($\mathbb{R} \to \mathbb{R}$) abbilden, so muss statt `int` ein Zahlentyp wie `float` oder `double` verwendet werden:

```
1 public static double f(double x){
2   return x * x;
3 }
```

5.1 void als Rückgabetyp

Manchmal ergibt sich der Fall, dass eine Methode etwas tun soll, das kein Ergebnis liefert. Soll z.B. eine längere Meldung ausgegeben werden, so würde es keinen Sinn machen, danach eine Zahl oder einen String zurückzugeben. In diesem Fall schreibt man `void` als Rückgabetyp vor den Methodennamen.

```
1 public static void gib_fehler_aus(String fehler) {
2   System.out.print("Es ist ein Fehler aufgetreten: ");
3   System.out.println(fehler);
4 }
```

5.2 Mehrere Parameter

Natürlich kann eine Methode auch mit mehr als einem Parameter arbeiten. Dabei darf jeder Parameter einen anderen Datentyp haben. Schreiben wir eine einfache Funktion zum Potenzieren. Sie soll das n-fache einer reellen Zahl ausrechnen, wobei $n \in \mathbb{N}$ ist.

$$power(x, n) = x^n$$

Die Methode dazu nimmt zwei Parameter, ein `double` und ein `int` und multipliziert dann das x n-Mal.

```
1 public static double power(double x, int n){
2   double ergebnis = 1;
3   for (int i=1; i<=n; i++)
4     ergebnis = ergebnis * x;
5   return ergebnis;
6 }
```

Innerhalb der Methode gibt es eine Variable `result`, die wir für Zwischenergebnisse verwenden und dann zurückgeben. Sie ist außerhalb der Methode nicht vorhanden (siehe auch das Kapitel zu *Gültigkeit*). Durch die Schleife wird nur multipliziert, wenn n größer gleich 1 ist.

Die Inhalte der Variablen die man einer Methode übergibt werden von ihr kopiert, so dass sich außerhalb der Methode eine Variable nicht ändert, falls in der Methode mit ihr gearbeitet wird. Diesen Mechanismus werden wir im Kapitel zur Objektorientierung ausführlicher begegnen.

5.3 Überladen von Methoden

Es dürfen beliebig viele Methoden in einem Programm vorkommen, die alle den gleichen Namen haben, sofern sie sich ihren Parametern unterscheiden. Wir können die obige Methode auch nur für ganze Zahlen und sogar für reele Exponenten schreiben. Beim Aufruf erkennt dann Java anhand der Datentypen der Parameter, welche Methode aufgerufen werden soll.

```
1 public static int power(int x, int n){
2   // aehnlich wie oben
3 }
4 public static double power(double x, double n){
5   // Die Methode ist so geschrieben, dass sie auch
6   // mit reelen Exponenten umgehen kann.
7 }
```

Wenn man mehrere Methoden hat, die gleich heißen und sich nur an den Parametern unterscheiden, so sagt man, sie haben verschiedenen Signaturen. Die Signatur ist der Name der Methode zusammen mit den Datentypen ihrer Parameter und ihrem Rückgabetyp. Auch mit einer unterschiedlichen Anzahl an Parametern oder unterschiedlicher Reihenfolge der Parameter haben zwei Methoden unterschiedliche Signaturen.

Wenn wir mehrere Methoden mit gleichen Namen aber verschiedenen Signaturen haben, so heißen die Methoden überladen. Allerdings dürfen sich Methoden nicht nur an ihrem Rückgabewert unterscheiden. Zum Beispiel ist die Methode `System.out.println` überladen und kann mit Variablen beliebigen Typs aufgerufen werden oder sogar ohne Parameter.

5.4 Aufruf von Methoden

Eine Methode, die sogenannte `main`-Methode kennen wir schon. Da das Programm zu Ende ist, wenn die `main`-Methode zu Ende ist, kann sie natürlich auch niemandem etwas zurückgeben und hat daher den Rückgabetyp `void`. Wenn man ein Programm startet, möchte man diesem Programm oft mitteilen, welche Datei es öffnen soll, oder man möchte im Programm gleich von Start an gewisse Optionen setzen können, zum Beispiel, dass keine Plugins geladen werden sollen. Solche Aufruf-Argumente werden bei Start eines Java-Programms von der Java-Runtime-Environement in ein String-Array geschrieben und dann der `main`-Methode übergeben. Aus diesem Grund hat die `main`-Methode als Parameter ein String-Array, welches häufig `args` genannt wird.

Wie wir wissen, wird die `main`-Methode automatisch aufgerufen. Wie ruft man aber nun die anderen Methoden auf? Hierzu muss man lediglich den Methodennamen hinschreiben und die Parameter in Typ und Anzahl korrekt übergeben. Wenn eine Methode einen Rückgabetyp hat, so speichert man diesen dadurch, dass man den Methodenaufruf - und somit den Rückgabewert - einfach einer Variable geeigneten Typs zuweist. Sehen wir uns hierfür ein Beispiel an:

```
1  public class Methoden {
2
3    // von oben
4    public static double power(double x, int n){
5      double ergebnis = 1;
6      for (int i=1; i<=n; i++)
7        ergebnis = ergebnis * x;
8      return ergebnis;
9    }
10
11   public static void main(String[] args) {
12     int exponent = 5;
13     double rueckgabe = power(2.0, exponent);
14     System.out.println(rueckgabe);
15   }
16 }
```

In der `main`-Methode wird die `power`-Methode mit den Argumenten `exponent` und `2.0` aufgerufen. Die `power`-Methode berechnet darauf hin 2 hoch 5. Das Ergebnis hiervon, nämlich 32, wird von der `power`-Methode zurück gegeben und in der `main`-Methode in der Variablen `rueckgabe` gespeichert und in der darauffolgenden Zeile ausgegeben. In diesem Beispiel wird die Variable `rueckgabe` nirgendwo sonst benötigt, daher können wir auch folgende Vereinfachung in der `main`-Methode vornehmen:

```
1  public static void main(String[] args) {
2    System.out.println(power(2.0, 5));
3  }
```

Diese `main`-Methode tut genau das Gleiche wie oben. Aber anstatt extra eine eigene Variable für den Rückgabewert der `power`-Methode anzulegen, übergeben wir hier den Rückgabewert der `power`-Methode direkt an die `println`-Methode. Solche Aufrufe werden also von innen nach außen ausgewertet. Bei Bedarf kann man auch tiefere Verschachtelungen vornehmen. In folgendem Beispiel wird 2 hoch 5 mit 3 hoch 3 potenziert, also 32 hoch 27 berechnet, und dann ausgegeben.

```
1  public static void main(String[] args) {
2    System.out.print(power(power(2.0, 5),power(3.0, 3)));
3  }
```

6 Gültigkeit

Mittlerweile haben wir die Grundlagen der imperativen Programmierung behandelt. Vielleicht hast du schon einmal festgestellt, dass nicht auf jede Variable die deklariert wurde von überall im Programm aus zugegriffen werden kann.

6.1 Blöcke

Variablen sind immer nur in einem bestimmten Teil eines Programms gültig. Dies nennt man den *Gültigkeitsbereich*. Der Gültigkeitsbereich wird durch Blöcke bestimmt, dazu dienen die geschweiften Klammern `{}`. Man spricht auch von einer Ebene in der eine Variable gilt.

Dies ist durchaus sinnvoll, denn wenn wir uns vorstellen ein längeres Programm zu schreiben, so müssten wir sehr aufpassen keine Variablen doppelt zu verwenden. Sollten wir z.B. in einer Schleife eine Zählvariable `i` verwendet haben, so müssten wir entweder später nirgends mehr `i` verwenden oder doch zumindest sicherstellen, dass immer richtige Startwerte in `i` stehen. Blöcke ermöglichen uns eine geschickte Strukturierung von Programmen.

Das folgende Beispiel ist zwar unsinnig, es zeigt aber, dass Blöcke allein die Gültigkeitsbereiche von Variablen bestimmen.

```
1  public static void main(String[] args) {
2    { // Block 1
3      int a = 5;
4      System.out.println(a);
5      // gibt 5 aus
6    }
7    { // Block 2
8      float a = 4.0f; // geht, weil in neuem Block
9      System.out.println(a);
10     // gibt 4.0 aus
11   }
12 }
```

Der obere Block enthält eine Variable `a`, die im unteren Block neu deklariert wird. Java „vergisst“ am Ende des ersten Blocks die Variable `a`, so dass sie noch einmal deklariert werden kann. Wären die Klammern der Blöcke nicht, würde eine Fehlermeldung ausgegeben werden, da man nicht zweimal eine Variable mit dem selben Namen deklarieren darf.

Eine Variable ist immer von der Initialisierung bis zum Ende der Ebene gültig. Ihre Lebensdauer geht von dem Zeitpunkt ihrer Erzeugung bis zu dem Zeitpunkt, an dem sie im Speicher gelöscht wird.

Wie wir vom Beispiel schon indirekt ableiten können ist es nicht möglich eine Variable aus einem Block zu verwenden, der auf der gleichen Ebene liegt. Wir werden in den folgenden Beispielen sehen, dass Blöcke andere Blöcke enthalten. In diesem Fall sind die Variablen aus der äußeren Ebene auch in den enthaltenen Blöcken zugreifbar.

```
1  { // aussen
2    int a = 1;
```

```
3     int b = 2;
4     { // innen
5       System.out.println(a);
6       // gibt 1 aus
7       int b = 3;
8       System.out.println(b);
9       // gibt 3 aus
10    }
11  }
```

Im Beispiel greifen wir auf die Variable `a` aus dem äußeren Block zu. Wenn aber in einem inneren Block eine Variable deklariert wird, die gleich heißt wie eine Variable in einem äußeren Block, so greifen wir automatisch auf die innerste Variable zu. Darum erhalten wir auch 3 als Ausgabe.

6.2 Blöcke im Kleinen

Bei Schleifen und bedingten Anweisung verwenden wir automatisch Blöcke. Variablen, die innerhalb dieser Blöcke angelegt werden, sind außerhalb nicht mehr gültig.

```
1  for(int i=1; i<10; i++) {
2    System.out.println(i);
3    // gibt Zahlen von 1 bis 9 aus
4  }
5  System.out.println(i); // Fehler
```

Innerhalb der Schleife ist die Variable `i` gültig, doch nach dem Block der Schleife kann nicht mehr auf sie zugegriffen werden, und das Beispiel liefert einen Fehler.

6.3 Blöcke bei Methoden

Auch Methoden verwenden die `{}` Klammern. Deshalb sind in ihnen deklarierte Variablen nur innerhalb gültig. Betrachten wir noch einmal die Methode `power` aus dem Kapitel über Methoden.

```
1  public static double power(double x, int n){
2    double ergebnis = 1;
3    for (int i=1; i<=n; i++) {
4      ergebnis = ergebnis * x;
5    }
6    return ergebnis;
7  }
```

Innerhalb der Methode sind die Variablen `x`, `n`, `ergebnis` und `i` gültig. Dabei gilt `i` nur innerhalb der Schleife, die anderen Variablen gelten in der ganzen Methode.

Methoden werden meist nicht in einander geschachtelt aufgeschrieben, d.h. ihre Blöcke sind alle auf der gleichen Ebene und Variablen sind unter den verschiedenen Methoden nicht zugreifbar. Da ein Methodenrumpf ebenso einen Block darstellt, sind innerhalb von Methoden nur die Variablen bekannt, die der Methode übergeben oder innerhalb des Methodenrumpfs deklariert wurden.

6.4 Blöcke bei Klassen

Auch wenn wir bisher noch nicht mit Klassen gearbeitet haben verwenden wir doch immer den Block der Klasse unseres aktuellen Programms.

```
1  class Gueltigkeit {
2    public static void main(String[] args) {
3      int c = 0;
4      System.out.println(c);
5    }
6  }
```

Die Klasse `Gueltigkeit` besitzt einen Block in dem sie die Methode `main` und in diesem Beispiel keine Variablen besitzt. Im Kapitel zur Objektorientierung werden wir Variablen auch innerhalb von Klassen einführen, die dann natürlich nur innerhalb des Blocks der Klasse gelten können.

6.5 Beispiel

Im folgenden Beispiel gibt es mehrere Gültigkeitsbereiche. Der erste Block erstreckt sich über die ganze Klasse, deshalb nennt man ihn auch Klassenebene. In den beiden Methoden gibt es jeweils eine Methodenebene. Innerhalb der oberen Methode gibt es zusätzlich noch eine Schleifenebene.

```
1  class Gueltigkeit {                                |Klasse
2    /* wird im naechten Kapitel erklaert */         |
3    static int a = 3;                               |
4                                                    |
5    public static double power(double x, int n){   ||power
6      double ergebnis = 1;                          ||
7      for (int i=1; i<=n; i++) {                    |||Schleife
8        ergebnis = ergebnis * x;                    |||
9      }                                             |||
10     return ergebnis;                              ||
11   }                                               ||
12                                                   |
13   public static int mal2(int zahl) {              ||mal2
14     int a = 2;                                    ||
15     return a * zahl;                              ||
16   }                                               ||
17                                                   |
18   public static void main(String[] args) {        ||main
19     int b = 5;                                    ||
20                                                   ||
21     double ergebnis = power(2.7135 ,b);           ||
22     ergebnis = ergebnis + a;                      ||
23     System.out.println(ergebnis);                 ||
24   }                                               ||
25 }                                                 |
```

In der Klassenebene ist `a` gültig. In `power` sind `x`, `n`, `ergebnis` und innerhalb der Schleife `i` gültig. In `main` sind `args`, `b` und eine andere Variable namens `ergebnis` gültig.

Die Variable `a` wird automatisch zu einem `double` konvertiert, wenn sie zum Ergebnis addiert wird. Die `|` markieren die Bereiche und gehören natürlich nicht zum eigentlichen Programm.

In der Methode `mal2` gibt es eine Variable `a`, die lokal den Wert 2 hat. Das `a`, das zu Klasse gehört hat den Wert 3, wird aber innerhalb der Methode überdeckt, wo eine gleichnamige Variable `al` mit dem Wert 2 deklariert wird.

7 Rekursion

Das Wort Rekursion kommt aus dem Lateinischen (*recurrere* – zurücklaufen) und bezeichnet in der Informatik[1] bzw. Mathematik eine Funktion, die sich selbst zur Lösung wiederverwendet. Rekursion ist ein Lösungsstrategie für eine Vielzahl von Problemstellungen. Meist führt Rekursion zu kompakten und gut lesbaren Lösungen.

Machen wir ein Beispiel. In der Mathematik gibt es eine Vorgänger- und eine Nachfolger-Funktion um geordnete Mengen zu definieren, z.B. kann man Eigenschaften der natürlichen Zahlen so beschreiben. Eine Zahl hat einen Vorgänger, der um eins kleiner ist als die Zahl selbst. Die folgende Funktion gibt für eine beliebige Zahl x die Aufrufe der Vorgänger-Funktion $v(x)$ an.

$$v(x) = \begin{cases} 0, & falls \quad x = 0 \\ v(x-1)+1, & sonst \end{cases} \quad \forall x \in \mathbb{N}$$

Für 0 ist $v(x)$ auch 0, danach ist der Wert der Funktion immer gleich dem Wert der Funktion eine Stelle links versetzt plus eins. Die Vorgänger-Funktion wird nach Größe der Zahl x-Mal aufgerufen.

Im Grund ist jede Rekursion eine Fallunterscheidung. Es gibt einen Fall für die *Abbruchbedingung* (auch Basisfall genannt) und einen *Rekursionsfall*. Im Beispiel ist $x = 0$ die Abbruchbedingung. Es ist sinnvoll sich bei Problemen die man mit Rekursion lösen möchte zuerst die Abbruchbedingung zu überlegen.

Der Rekursionsfall tritt meist öfters ein, man nennt dies Rekursionsschritte. Im Beispiel wird die gleiche Funktion f mit veränderten Parametern $(x-1)$ nochmals aufgerufen und dann ihr Ergebnis um eins erhöht.

Die Formel lässt sich natürlich auch als Methode umsetzen.

```
1 int v(int x ){
2   if(x == 0) { // Basisfall
3     return 0;
4   } else {
5     return v(x-1) +1; // Rekursionsfall
6   }
7 }
```

Als einfaches Beispiel soll nun $v(2)$ berechnet werden. Mathematisch sieht das so aus.

$$\begin{aligned} v(2) &= v(1)+1 \\ &= (v(0)+1)+1 \\ &= (0+1)+1 \\ &= 2 \end{aligned}$$

Man schreibt zur Veranschaulichung die verschiedenen Aufrufe und neuen Argumente immer in eine Zeile unterhalb der aktuellen Zeile und anschließend die Rückgabewerte entweder direkt an die Stelle des Aufrufs oder rechts daneben.

[1] In der Informatik sagt man auch zum Scherz „Rekursion – siehe Rekursion“

Was wir auf dem Papier machen muss auch von Java getan werden. Java merkt sich an welcher Stelle ein Aufruf passiert ist und speichert dies in einem sogenannten Callstack. Beim Aufsteigen aus der Rekursion kann Java so die richtigen Ergebnisse an den richtigen Stellen einsetzen.

7.1 Formen von Rekursion

Es gibt verschiedene Formen von Rekursion. Das Beispiel oben nutzt lineare Rekursion, ein Aufruf erzeugt eine Kette von Aufrufen und terminiert dann. Diese Art der Rekursion lässt sich in der Regel durch Schleifen ersetzen.

Bei kaskadenförmiger Rekursion folgen mehrere Aufrufe parallel, diese Form ist oft elegant, kann aber einen exponentiellen Berechnungsaufwand haben. Ein gutes Beispiel hierfür ist die Berechnung der Fibonacci-Zahlen. Diese sind wie folgt definiert:

$fib_0 = 0$
$fib_1 = 1$
$fib_n = fib_{n-1} + fib_{n-2}$

Um fib_5 zu berechnen erhält man dann keine Kette sondern den folgenden Aufrufbaum.

fib_5
$= fib_4 + fib_3$
$= (fib_3 + fib_2) + (fib_2 + fib_1)$
$= ((fib_2 + fib_1) + (fib_1 + fib_0)) + ((fib_1 + fib_0) + 1)$
$= (((fib_1 + fib_0) + 1) + (1 + 0)) + ((1 + 0) + 1)$
$= (((1 + 0) + 1) + (1 + 0)) + ((1 + 0) + 1)$
$= 5$

Bei wechselseitiger Rekursion existieren zwei Funktionen, die sich gegenseitig aufrufen.

Schließlich gibt es noch die verschachtelte Rekursion bei der die Parameter für den Aufruf selbst durch Rekursion bestimmt werden. Diese Form lässt sich schwer nachvollziehen. Sie wird z.B. bei der Ackermann-Funktion verwendet.

7.2 Gefahren bei Rekursion

Die größte Gefahr bei Rekursion ist es, den Basisfall falsch zu konstruieren. Dies hat oft den Effekt, dass das Programm dann nicht terminiert.

So elegant manche Lösungen bei Rekursion auch geschrieben werden können, man sollte nicht vergessen, jeder Aufruf benötigt Speicher und Rechenzeit!

8 Objektorientierung

Nachdem wir nun die Grundlagen der imperativen Programmierung kennengelernt haben kommen wir zu einem neuen Konzept, der Objektorientierung. An sich ist dies ein einfaches Konzept: Wenn wir bisher eine Variable angelegt haben, so musste sie immer von einem bestimmten Typ sein. Um uns herum gibt es deutlich mehr Typen als die uns bisher bekannten Zahlen, Booleans oder Strings. Diese werden wir jetzt kennen lernen.

Bei Gegenständen die uns tagtäglich begegnen sprechen wir allgemein von Objekten. Menschen, Gebäude, Gegenstände - all dies sind für uns Objekte, welche einem bestimmten Typ zuordbar sind. So hat z.B. die Universität mehrere Gebäude, aber auch ein Reihenhaus oder eine Villa ist ein Gebäude. Ein Typ kann also verschiedene Ausprägungen haben.

Bei der Objektorientierung bedient man sich dieser Sichtweise und versucht existierende Aufgaben und Probleme in Objekte umzuwandeln, mit denen dann gearbeitet werden kann. Dieses Konzept ist etwas abstrakter und erfordert vom Programmierenden zunächst eine neue Sichtweise auf Probleme, ermöglicht aber auf Dauer intuitivere und übersichtlichere Programme sowie eine Reihe weiterer Vorteile.

Eher selten programmiert man aber Gebäude, schon eher möchte man Bücher-, CD- oder DVD-Sammlungen verwalten, Fotos katalogisieren oder Finanzen erfassen.

8.1 Klassen als Typen

Als Beispiel möchten wir nun unsere Finanzen erfassen. Dazu möchten wir Konten bei verschiedenen Banken verwalten. Ein Konto hat mehrere Eigenschaften, eine Nummer, einen Inhaber, einen Kontostand und die Bankleitzahl. Bisher kennen wir eine Reihe von Datentypen von Java, `int`, `double` oder `String`. Es wäre möglich mehrere Arrays anzulegen und immer unter dem selben Index alle Attribute eines Kontos zu speichern.

```
1 String[] inhaber = new String[10];
2 double[] guthaben = new double[10];
3 int[] nummer = new int[10];
4 int[] blz = new int[10];
```

Das funktioniert zwar, allerdings hängen die zusammengehörigen Daten hier nur indirekt über ihren Index zusammen. Wir hätten also gerne einen Datentyp, in dem ein `String` ein `double` und zwei `int`s gespeichert werden können. Um dies zu realisieren, deklarieren wir eine neue Klasse, die dann diese Attribute enthält. Dazu müssen wir eine neue Datei anlegen, welche den Quelltext dieser Klasse enthält.

```
1 // Datei Konto.java
2 public class Konto {
3   String inhaber;
4   double guthaben;
5   int nummer;
6   int blz;
7 }
```

Es ist eine Konvention, den Namen von Klassen immer groß zu schreiben. Bei zusammengesetzten Namen für Klassen wird jeweils der erste Buchstabe der Teilwörter mit einem Großbuchstaben begonnen, also beispielsweise `BankAutomatKartenLeser`.

Unsere Klasse `Konto` enthält die Attribute Inhaber, Guthaben, Nummer und BLZ. Diese haben als Datentyp dann `String`, `double` bzw. `int`.

8.2 Objekte

Sobald eine Klasse deklariert ist können wir Objekte von ihrem Typ anlegen. Dazu müssen wir das Schlüsselwort `new` verwenden. Sobald eine Klasse deklariert ist können wir Objekte von ihrem Typ anlegen. Dabei müssen wir das Schlüsselwort `new` verwenden.

```
1 Konto giro = new Konto();
2 // im Vergleich dazu das Anlegen eines ints
3 int informatikerZahl = 42;
```

Nun enthält die Variable namens `giro` ein Konto mit allen Attributen, die in der Klasse definiert sind. Dies ist ganz ähnlich zum Deklarieren eines `int` aus dem Kapitel Datentypen. Vorne steht in beiden Fällen der Datentyp, danach der Variablenname. Es folgt das = und dann der Wert, welcher der Variable zugewiesen wird. Bei dem `int` die Zahl, beim Konto wird durch das Schlüsselwort `new` ein Objekt angelegt. Wir initialisieren nun die verschiedenen Attribute. Dazu schreiben wir einfach `Variablenname.Attribut` um auf ein Attribut zuzugreifen. Wie bei allen Variablen ist es auch hier bei den Objekten Konvention, Variablennamen mit einem Kleinbuchstaben zu beginnen.

```
1 giro.inhaber = "Max Muster";
2 giro.guthaben = 122.77;
3 giro.nummer = 1234567;
4 giro.blz = 10050000;
```

Damit haben wir nun ein Objekt vom Typ Konto in der Variablen, welches verschiedene Attribute hat. Es ist jetzt einfacher möglich viele Konten zu haben:

```
1 Konto[] konten = new Konto[100];
```

Wenn wir mit Objekten arbeiten wollen geht das nun recht einfach. Zunächst legen wir ein weiteres Konto an und vergleichen dann die Guthaben.

```
1 Konto spar = new Konto();
2 spar.inhaber = "Max Muster";
3 spar.guthaben = 1523.17;
4 spar.nummer = 98765;
5 spar.blz = 25050180; // Hannover
6
7 if(giro.guthaben < spar.guthaben) {
8   System.out.println("Sparkonto voller als Girokonto.");
9 }
```

8.3 Methoden

Zu einer Klasse kann es eine Reihe von Methoden geben. Die Objektorientierung erlaubt es uns, Methode direkt an die Klasse zu koppeln. Das ist sehr praktisch, so muss man

sich nicht merken, welche Methoden auf welchen Objekten arbeiten, man kann einfach die Methode eines Objekts aufrufen.

Bisher waren alle unsere Methoden `static`. Das bedeutet, sie sind immer aufrufbar, ohne dass von der Klasse ein Objekt erzeugt wurde. Zum Beispiel muss man kein Objekt der Java-Klasse `Math` erzeugen, um `Math.sqrt(2)` aufzurufen, denn die Methode `sqrt()` ist statisch.

Nun wollen wir nicht-statische Methoden einführen. Das sind Methoden, die fest an ihr Objekt geknüpft sind. Wir können diese Methode nur dann aufrufen, wenn auch ein Objekt existiert.

```
1  // Datei Konto.java
2  public class Konto {
3    String inhaber;
4    double guthaben;
5    int nummer;
6    int blz;
7    double limit;
8
9    // eine nicht-statische Methode
10   public boolean abheben(double betrag) {
11     if(betrag < 0) { // Abheben von negativen
12       return false; // Betraegen geht nicht
13     }
14     else if((guthaben - betrag) > limit) {
15       guthaben = guthaben - betrag;
16       return true; // Abheben erfolgreich
17     }
18     else {
19       return false; // Limit ueberzogen
20     }
21   }
22 }
```

Die Klasse ist um das Attribut `limit` erweitert worden, über das die Bank festlegen kann, wie weit das Konto überbucht werden kann. Bei einem Limit von -500 könnten wir das gesamte Guthaben und dann noch 500 abheben, danach liefert die Methode `false`. Auch eine Abhebung durch einen negativen Betrag wird abgefangen.

Diese Methode ist nicht statisch, das bedeutet, sie lässt sich nur aufrufen, wenn man vorher ein Objekt angelegt (und sinnvollerweise gefüllt) hat:

```
1  giro.abheben(200);
```

Konstruktor

Bisher haben wir Objekte angelegt und danach von Hand jedes Attribut mit Werten gefüllt. Dies geht auch schneller und eleganter in dem man einen sogenannten Konstruktor nutzt. Jede Klasse besitzt automatisch eine Methode, die beim Erstellen eines Objektes aufgerufen wird. Zunächst macht diese Methode nichts weiter, als das Objekt anzulegen. Den Konstruktor erkennt man daran, dass er genauso wie die Klasse heißt. Als einzige Methode in Java hat er keinen Rückgabewert, nicht einmal `void`, denn er erzeugt ja bereits das Objekt, wodurch der Rückgabetyp dieser Methode eindeutig und implizit definiert ist.

Meist nimmt ein Konstruktor die verschiedenen Werte für die Attribute entgegen, die ein Objekttyp besitzt. Es ist auch möglich, mehrere Konstruktoren zu haben und zum Beispiel den Fall abzudecken, dass ein Attribut unbekannt ist und daher mit dem Standardeintrag gefüllt werden soll.

```
1  public class Konto {
2    String inhaber;
3    double guthaben;
4    int nummer;
5    int blz;
6    double limit;
7
8    // Konstruktor 1: Mit Guthaben
9    Konto(String i, double g, int n, int b, double l) {
10     inhaber = i;
11     guthaben = g;
12     nummer = n;
13     blz = b;
14     limit = l;
15   }
16
17   // Konstruktor 2: Kein Guthaben
18   Konto(String i, int n, int b, double l) {
19     inhaber = i;
20     guthaben = 0.0;
21     nummer = n;
22     blz = b;
23     limit = l;
24   }
25 }
```

Wenn jetzt ein Objekt vom Typ `Konto` erzeugt werden soll, kann direkt im Konstruktor schon alles übergeben werden. Um zum Beispiel das Girokonto von oben anzulegen, reicht Folgendes aus:

```
1 // Konstruktor 1
2 Konto giro = new Konto("Max Muster", 122.77, 1234567,
3   10050000, 0);
```

Java ergänzt automatisch einen leeren Konstruktor, wenn man eine Klasse schreibt ohne selbst einen oder mehrere Konstruktoren anzugeben. Sobald man aber selbst einen Konstruktor schreibt, wird der leere Standardkonstruktor nicht mehr automatisch von Java erzeugt. Das bedeutet für unser obiges Beispiel, dass `new Konto()` nun nicht mehr funktioniert. Jetzt können nur noch unsere selbst geschriebenen Kontruktoren verwendet werden.

8.4 Gültigkeit und statische Attribute

Im vorletzten Kapitel haben wir uns mit der Gültigkeit von Variablen befasst. Betrachten wir nun wie es sich mit der Gültigkeit von Attributen verhält. Wann immer man ein Objekt anlegt werden automatisch die Attribute des Objekts erzeugt. Ihr Gültigkeitsbereich ist innerhalb des Objekts, durch die Punktnotation (`Objektname.attributname`) sind Attribute aber auch außerhalb des Objekts zugreifbar.

Attribute müssen immer in der Klasse deklariert werden. Wenn sie bei der Deklaration keinen Wert zugewiesen bekommen, so werden sie mit einem Standardwert initialisiert. Für numerische Werte ist das der Wert 0, für `boolean`-Werte `false` und für alle anderen Datentypen, also auch selbst erstellte, `null`. Es ist keineswegs garantiert, dass alle Java Versionen auf allen Geräten die selben Standardwerte für primitive Datentypen nutzen. Deshalb ist es guter Stil, Attributen einen Startwert zuzuweisen, entweder bei der Deklaration, oder im Konstruktor. Sonst könnte ein lesender Zugriff auf ein nicht initialisiertes Objekt einen Laufzeitfehler verursachen, welcher ohne weitere Behandlung das Programm abstürzen lässt.

Wir haben bereits gesehen, dass eine Klasse statische Methoden besitzen kann, die auch zugreifbar sind, wenn keine Objekte existieren. Es gibt auch statische Attribute. Durch das statische Attribut `anzahl` möchten wir speichern, wie viele Konten wir insgesamt angelegt haben.

```
1  public class Konto {
2    static int anzahl;
3    String inhaber;
4    double guthaben;
5    // Startwert bei Deklaration
6    int nummer = 123;
7    int blz;
8    double limit;
9
10   Konto(String i, double g, int n, int b, double l) {
11     inhaber = i;
12     guthaben = g;
13     nummer = n;
14     blz = b;
15     limit = l;
16     anzahl++;
17   }
18 }
```

Nun wäre es ungeschickt, immer wenn wir ein Konto anlegen die Anzahl an Konten von Hand hochzuzählen. Das kann der Konstruktor automatisch für uns erledigen.

Wie oben gesagt, greifen wir auf Objektattribute, also oben beispielsweise `inhaber` oder `nummer`, mit `Objektname.attributname` zu. Auf statische Attribute, sogenannte Klassenattribute, können wir zudem auch durch `Klassenname.attributname` zugreifen. Wir brauchen also, genau wie auch bei statischen Methoden, nicht extra ein Objekt der Klasse zu erstellen. Diese Unabhängigkeit von einem Objekt bewirkt allerdings auch, dass alle Objekte einer Klasse und die Klasse selbst auf genau dem selben statischen Attribut arbeiten. Ändert man ausgehend vom Beispiel oben in einem Objekt vom Typ Konto das Attribut `anzahl`, dann ist dieses Attribut auch bei allen anderen Objekten vom Typ Konto und bei der Klasse Konto selbst auf den neuen Wert geändert, da die Objekte keine eigene Kopie von `anzahl` haben.

8.5 public und private

Jedes Attribut einer Klasse und jede Methode kann eine der Eigenschaften `public` oder `private` erhalten. Hat sie keine davon, gilt sie standardmäßig als `public` (erst bei Benutzung von Paketen ergibt sich hier ein Unterschied).

Wenn ein Attribut oder eine Methode `private` ist, kann man es bzw. sie außerhalb der Klasse nicht benutzen, bei `public` schon. Diese Eigenschaft nutzt man vor allem, um dem Anwender, der die Objekte (außerhalb der Klasse) erzeugt, zu verbieten, Attribute auf beliebige Werte zu setzen.

Für unser Konto bedeutet dies, das wir selbst bestimmen wollen wie das Guthaben geändert werden darf. Darum machen wir das Attribut `private` und erweitern die Klasse um eine Methode zum Einzahlen auf das Konto, die einen Sicherheitsmechanismus enthält.

```
1  public class Konto {
2    String inhaber;
3    private double guthaben;
4    int nummer;
5    int blz;
6    double limit;
7
8    // Setzen der Attributwertes
9    public void einzahlen(double betrag) {
10     if(betrag > 0.0) {
11       guthaben = guthaben + betrag;
12     }
13   }
14
15   // Auslesen des Attributwertes
16   public double getGuthaben() {
17     return guthaben;
18   }
19   // Konstruktor usw.
20 }
```

Die Methode `einzahlen(double betrag)` ist `public`, damit sie von außerhalb der Klasse verwenden kann. Wenn wir auf das Guthaben zugreifen wollen um es auszulesen müssen wir dafür eine Methode schreiben.

Wenn man gerade mit der Objektorientierung anfängt, mag einem der Nutzen von public und privat eher gering erscheinen. Tatsächlich ist es aber so, dass die Einschränkung der Sichtbarkeit eine wichtige Maßnahme ist, um die Komplexität von Programmen beherrschen zu können. Machen wir hierzu ein Beispiel:

An einem Fernseher werden lediglich die Anschlüsse nach außen geführt, die der Anwender auch benutzen können soll, beispielsweise der Stromanschluss und das Antennenkabel oder der Anschluss für externe Lautsprecher. Es wäre unsinnig auch andere elektrische Verbindungen nach Außen zu führen, etwa die Kontakte von verbauten Chips oder die Kabelverbindungen der einzelnen Komponenten. Und selbst wenn es genau dokumentiert wäre, welche Anschlüsse benutzt werden dürften und welche nicht, so wäre es doch eine massive Fehlerquelle und viele Benutzer würden den Fernseher nicht bedienen können oder sogar versehentlich zerstören, weil sie das Stromkabel am Signalprozessor angeschlossen haben. Deshalb haben Fernseher auch Gehäuse und nur die sinnvollen Anschlüsse werden aus dem Gehäuse geführt. Die restliche Komplexität wird im Inneren des Gerätes verborgen.

Genau so verhält es sich auch mit unseren Klassen, nur dass wir die Klassen dafür verwenden, Daten zu verwalten und auf ihnen zu arbeiten. Sollen Andere mit den von uns geschriebenen Klassen arbeiten, so ist es sinnvoll, nur diejenigen Methoden und Attribute `public` zu setzen, die man auch tatsächlich verwenden soll. Dadurch verhindert man Fehlbedienung, wie bei unserem Kontobeispiel, wo man verhindern will, dass man das Guthaben einfach ohne jedes Limit verringern kann. Gleichzeitig reduzieren wir auch die

Komplexität für den Nutzer unserer Klasse. So zeigen Entwicklungsumgebungen bei der Verwendung einer Klasse nur die öffentlichen Teile der Klasse an. Durch sinnvolle Bezeichnung der öffentlichen Methoden und Attribute erhält der Verwender so schon einen guten Überblick über den Funktionsumfang einer Klasse. Hierbei kann dem Verwender auch egal sein, wenn es zahlreiche Hilfsmethoden und interne Attribute gibt, welche durch die Kennzeichnung als `private` – wie beim Fernseherbeispiel – nicht nach außen geführt werden.

Man nennt dieses Konzept Kapselung oder auch Geheimnisprinzip. Es stellt eine der größten Stärken der objektorientierten Programmierung dar. Die Konsequenz aus diesem Konzept ist, dass der Nutzer einer Klasse sich nicht um den internen Ablauf der Methoden kümmern muss und somit auch, falls nötig, eine spätere Änderung der Implementierung der Klasse möglich ist.

8.6 Referenzen

Was geschieht wenn zwei Variablen auf das selbe Objekt verweisen? Vergleichen wir einfache Datentypen mit Objekten.

```
1  int a = 5;
2  int b = a;
3  b = 3;
4  // a=5 und b=3
5
6  Konto giro = new Konto("Max Muster", 122.77, 1234567, ↩
       10050000, 0);
7  Konto spar = new Konto("Max Muster", 1523.17, 98765, ↩
       25050180, 0);
8  spar = giro;
9  System.out.println(spar.blz);
10 // gibt 10050000 aus
```

Bei den einfachen Datentypen wurden von Java zwei `int` angelegt. Als `a` an `b` zugewiesen wurde, wurde eine Kopie der Wertes, also der Zahl, in `b` abgelegt.

Bei Objekten wird aber in der Variable nicht die Zahl gespeichert sondern nur ein Verweis auf das Objekt im Speicher. Das Objekt selbst kann sehr groß sein, das heißt viel Speicher benötigen, weil es viele Attribute enthält. Wenn bei jeder Zuweisung das ganze Objekt umkopiert werden müsste, wäre das sehr aufwändig. Deshalb benutzt Java Verweise, man spricht von Referenzen.

Wenn nun `giro` an `spar` zugewiesen wird, so wird einfach dessen Referenz kopiert und an `spar` übergeben.

Eine besondere Referenz ist die `null`-Referenz. Mit dem Schlüsselwort `null` erzeugt man eine leere Referenz, die auf keine Daten zeigt. Setzt man alle Referenzen auf ein Objekt auf `null`, so löscht Java die zugehörigen Daten aus dem Speicher. Das automatische Löschen nennt man *garbage collection*, denn die nicht mehr benützten Objekte entsprechen (Daten-)Müll, der nicht mehr gebraucht wird und von Zeit zu Zeit von Java weggeräumt werden muss um neuen freien Speicher zu erhalten.

Ein Objekt wird auch automatisch gelöscht, falls es keine Referenzen mehr darauf gibt. Wenn wir ein Giro- und ein Sparkonto anlegen, dann haben wir zwei Objekte im Speicher geschaffen, wie auf der linken Hälfte des Bildes zu sehen ist. Sobald aber `giro` an `spar` zugewiesen wird, so verweist keine Referenz mehr auf das `spar`-Objekt und es wird gelöscht.

Achtung: Versucht man über die `null`-Referenz auf Eigenschaften oder Methoden zuzugreifen, kommt es zu einem Laufzeitfehler, genau so als ob das Objekt erst gar nicht initialisiert worden wäre:

```
1 Konto giro = new Konto("Max Muster", 122.77, 1234567, ↩
      10050000, 0);
2 giro = null; // giro wird geloescht
3 System.out.println(giro.blz);
4 // Null-Pointer-Fehler und Programmabsturz
```

9 Einfache abstrakte Datentypen

In der Informatik existieren neben den *primitiven Datentypen*, die wir bereits in den ersten Kapiteln kennengelernt haben und beispielsweise zur Speicherung einzelner Zahlenwerte dienen, noch sogenannte *abstrakte Datentypen* zur Repräsentation von komplexeren Daten oder Mengen davon. Diese abstrakten Datentypen sind erweiterte Datentypen, die für viele Algorithmen und Problemlösungen benötigt werden oder erst eine effiziente Problemlösung ermöglichen. Im Grunde sind abstrakte Datentypen zunächst Beschreibungen, wie diese Datenstrukturen aufgebaut sind, zu was sie dienen und wie sie funktionieren. Allerdings existieren in vielen Programmiersprachen wie auch in Java bereits vorimplementierte erweiterte Datentypen.

Doch in in einem Informatikstudium sollte man zumindest einmal selbst einfache abstrakte Datentypen implementiert haben, um ein tieferes Verständnis hierfür zu entwickeln. Ein einfacher abstrakter Datentyp ist die sogenannte Liste, die ähnlich einem Array mehrere Elemente eines bestimmten Typs aufnehmen kann. Anders als bei einem Array kann eine Liste aber eine variable Anzahl von Elementen aufnehmen und somit kann auch die Länge der Liste dynamisch geändert werden, was bei einem Array nicht möglich ist.

Weitere wichtige Datentypen sind unter anderem Stapel (stacks), Schlangen (queues), Bäume (trees), Halden (heaps) oder assoziative Arrays (hash maps). Das Thema der abstrakte Datentypen wird später im Studium in der Vorlesung *Algorithmen und Datenstrukturen* noch vertieft.

9.1 Listen

Listen bestehen aus einzelnen Listenelementen, die miteinander verbunden werden. Diese Listenelemente sind Objekte und enthalten im Wesentlichen zwei wichtige Instanzvariablen: Eine Variable von dem Datentyp, für den die Liste ausgelegt ist sowie eine Referenz auf das nächste Listenelement in der Liste. Hier ein Beispiel für Listenelemente einer Liste von Integer-Werten:

```
1 public class ListenElement {
2   int inhalt;
3   ListenElement nachfolger;
4
5   public ListenElement(int inhalt) {
6     this.inhalt = inhalt;
7   }
8 }
```

Diese Beispielklasse enthält zusätzlich noch einen Konstruktor, dem man gleich den Wert des Listenelements beim Instanziieren übergeben kann. So könnte man zwei Listenelemente erzeugen:

```
1 ListenElement erstesElement = new ListenElement(22);
2 ListenElement zweitesElement = new ListenElement(43);
```

Bisher wurden nur einzelne Elemente und keine wirkliche Liste erzeugt, da noch keine Reihenfolge der Elemente festgelegt wurde. Hierfür wird das zweite Element als Nachfolger des ersten referenziert:

```
erstesElement.nachfolger = zweitesElement;
```

Per Definition ist dasjenige Listenelement das letzte, welches keinen Nachfolger mehr besitzt, also das Listenelement, bei dem die Nachfolger-Referenz `null` ist.

Um später komfortabler mit der Liste zu arbeiten, bietet es sich an, eine Listenklasse zu erstellen, die das erste Listenelement kapselt und somit Zugriff auf die Liste in Form eines einzelnen Objektes bietet:

```
public class Liste {
  ListenElement listenKopf;

  public Liste(ListenElement listenKopf) {
    this.listenKopf = listenKopf;
  }

  public int laenge() {
    //Muss noch implementiert werden...
  }

  public void fuegeHinzu(ListenElement element) {
    //Muss noch implementiert werden...
  }

  //...
}
```

Die obenstehende zwei-elementige Beispielliste würde man nun so erzeugen:

```
Liste meineListe = new Liste(new ListenElement(22));
meineListe.fuegeHinzu(new ListenElement(43));
meineListe.fuegeHinzu(new ListenElement(12));
```

Wie man leicht sieht, hat die Einführung einer eigenen Klasse für die Liste einen großen Vorteil. Viele Operationen wie das Zählen aller Elemente oder die Aufnahme eines neuen Elementes in die Liste beziehen sich vorerst auf die ganze Liste und noch nicht konkret auf einzelne Elemente. Die Listenklasse kann dann intern Operationen delegieren oder auf den einzelnen Listenelementen operieren. Die die bisherige Listenimplementierung zeigt Abbildung 9.1.

Eine typische Operation auf Listen ist nun das Iterieren über die komplette Liste. Während bei einem Array genau feststeht, wie viele Elemente enthalten sind, ist hier das Iterieren mithilfe einer `for`-Schleife einfach. Bei einer Liste bietet sich jedoch eine andere Schleife an, nämlich eine `while`-Schleife. Und als Wiederholungsbedingung wird gefordert, dass das aktuelle Element nicht das letzte Element ist. In Java sieht das in etwa so aus:

```
ListenElement aktuellesElement = meineListe.listenKopf;
while(null != aktuellesElement){
  //etwas mit dem aktuellen Element machen, danach...
  aktuellesElement = aktuellesElement.nachfolger;
}
```

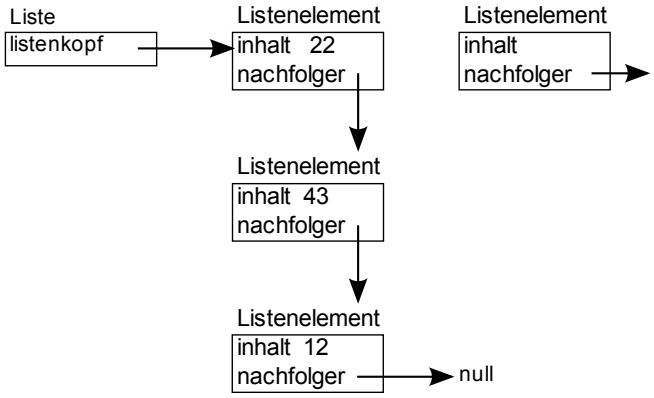


Abbildung 9.1: Die bisherige Listenimplementierung

Es wird eine temporäre Variable `aktuellesElement` vom Typ `ListenElement` erzeugt, die anfangs auf den Kopf der Liste zeigt. In der `while`-Schleife kann nun mit dem jeweiligen Element gearbeitet werden. Am Ende des Schleifenkörpers wird nun die Referenz des aktuellen Elements auf seinen Nachfolger geschoben. Erst beim letzten Element wird den Wert dieser Variable dann `null` und die Schleife wird verlassen, da die Wiederholungsbedingung nicht mehr erfüllt ist.

Bei vielen weiteren Operationen, wie dem Einfügen, Überschreiben oder Löschen von Elementen in der Liste gibt es zwei wichtige Dinge zu beachten. Es muss sichergestellt werden, dass nach der jeweiligen Operation der Rest der Liste weiterhin intakt ist. So sollen zum Beispiel nach dem Löschen eines Elements nicht alle Element hinter dem gelöschten Element weiterhin in der Liste hängen und nicht mit entfernt werden. Außerdem müssen Ausnahmefälle sorgsam beachtet werden. Das sind Operationen am Kopf sowie am Ende der Liste, bei denen das Vorgehen eventuell angepasst werden muss.

Wie bei vielen abstrakten Datentypen gibt es auch bei den Listen diverse Variationen. So gibt es auch Listen mit Listenelementen, die ihren Nachfolger und ihren Vorgänger verlinken, oder auch Listen, bei denen das letzte Element wieder auf den Listenkopf zeigt, und somit einen Ring erzeugt.

9.2 Bäume

Eine weitere wichtige abstrakte Datenstruktur in der Informatik sind Bäume. Sie sind mit linearen Listen verwandt, jedoch können die Elemente eines Baumes mehr als einen Nachfolger besitzen. Bei Bäumen spricht man meist nicht mehr von Elementen, sondern von Knoten. Eine typische Baumstruktur ist der Binärbaum. Bei ihm besitzt jeder Knoten zwei Nachfolger – jeweils einen linken und einen rechten nachfolgenden Knoten. Eine entsprechende Implementierung für Knoten eines Baumes für Integer-Werte könnte wie folgt aussehen:

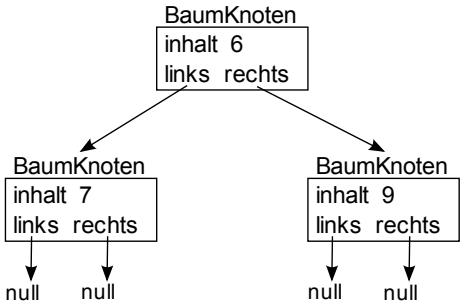
```
1 public class BaumKnoten {
2   int inhalt;
3   BaumKnoten linkerNachfolger;
4   BaumKnoten rechterNachfolger;
5
```

```
6   public BaumKnoten(int inhalt) {
7     this.inhalt = inhalt;
8   }
9 }
```

Der oberste Knoten, der selbst nicht Nachfolger eines anderen Knotens ist, wird auch Wurzel bezeichnet. Diejenigen Knoten im Baum, die keine Nachfolger mehr haben, bei denen also die Referenzen auf die Nachfolger mit dem Wert `null` belegt sind, heißen außerdem Blätter.

Ein einfacher Baum lässt sich so wie folgt erzeugen:

```
1 //Erzeugen der Knoten
2 BaumKnoten wurzel = new BaumKnoten(6);
3 BaumKnoten linkerKnoten = new BaumKnoten(7);
4 BaumKnoten rechterKnoten = new BaumKnoten(9);
5
6 //Einhaengen der Nachfolgeknoten
7 wurzel.linkerNachfolger = linkerKnoten;
8 wurzel.rechterNachfolger = rechterKnoten;
```



Hier ist der Knoten mit 6 der Wurzelknoten, die Knoten 7 und 9 stellen Blattknoten dar.

Wie schon bereits bei den Listen bietet sich auch bei den Bäumen die Erstellung einer Rahmenklasse an, um komfortabel auf der Datenstruktur arbeiten zu können. Hier wird nun anstatt des Kopfes der Liste, also dem ersten Listenelment der Wurzelknoten im Baumobjekt gespeichert. Es ist sinnvoll, baumspezifische Methoden wie das Traversieren des Baums oder typische Operationen wie das Einfügen, Suchen oder Löschen von Knoten ebenfalls in der Rahmenklasse zu implementieren.

```
1  public class Baum {
2    BaumKnoten wurzelKnoten;
3
4    public Liste(BaumKnoten wurzelKnoten) {
5      this.wurzelKnoten = wurzelKnoten;
6    }
7
8    public void fuegeHinzu(BaumKnoten knoten) {
9      //Muss noch implementiert werden...
10   }
```

```
11
12    public boolean enthaeltWert(int wert) {
13      //Muss noch implementiert werden...
14    }
15
16    //...
17  }
```

Nun folgt ein Beispiel für die Nutzung dieser Klasse:

```
1  //Anlegen eines neuen Baum-Objektes mit der Wurzel 6
2  Baum meinBaum = new Baum(new BaumKnoten(6));
3
4  //Hinzufuegen der Knoten 7 und 9
5  meinBaum.fuegeHinzu(new BaumKnoten(7));
6  meinBaum.fuegeHinzu(new BaumKnoten(9));
7
8  //Suche nach dem Wert 7 im Baum
9  boolean sucheErfolgreich = meinBaum.enthaeltWert(7);
10 //liefert true
```

Während sich zur Iterierung über Listenelemente Schleifen angeboten haben, da man somit die einzelnen Elemente nacheinander durchlaufen konnte, funktioniert nun dieses Konzept für Bäume nicht mehr, da jeder Knoten in einem Binärbaum nicht nur einen, sondern auch zwei oder eventuell gar keinen Nachfolger besitzen kann. Hier greift man auf das Konzept der Rekursion zurück, um einen Baum vollständig zu traversieren. Es gibt verschiedene Möglichkeiten, wie Bäume rekursiv traversiert werden können, alle haben jedoch gemeinsam, dass mithilfe einer Methode ein Knoten ausgewertet und zusätzlich die Methode rekursiv mit dessen Nachfolgeknoten aufgerufen wird. Dieses Konzept funktioniert, da jeder Knoten in einem Baum zugleich auch eine Wurzel eines Teilbaumes ist, nämlich dem Baum bestehend aus diesem Knoten und all seinen nachfolgenden Knoten. Solche rekursiven Methoden werden immer auf dem Wurzelknoten eines Baumes aufgerufen. Rekursiv werden dann die Nachfolgeknoten der Wurzel aufgerufen, und so weiter, bis schließlich die Methodenaufrufe die Blätter erreichen und die Rekursion beenden.

Folgendes Code-Beispiel ermöglicht das Aufsummieren aller Werte in einem Integer-Binärbaum. Bei einem echten Einsatz sollten solche Methoden im Übrigen in die entsprechende Baumklasse eingefügt werden.

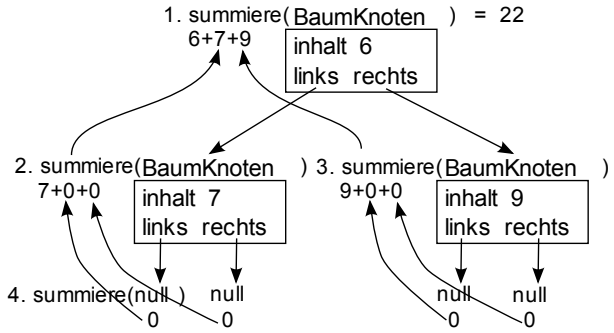
```
1  public static int summiere(BaumKnoten knoten) {
2    if(wurzel==null) {
3      return 0;
4    } else {
5      return (knoten.inhalt
6        + summiere(knoten.linkerNachfolger)
7        + summiere(knoten.rechterNachfolger));
8    }
9  }
```

Gestartet wird diese rekursiver Summierung wie folgt:

```
1  int summe = summiere(meinBaum.wurzelKnoten);
```

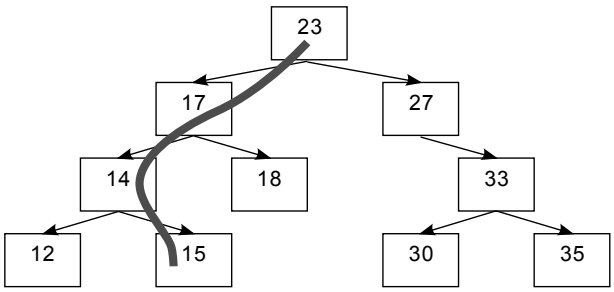
Die folgende Abbildung zeigt wie die Inhalte des Baums aufsummiert werden. Zuerst wird die Wurzel bearbeitet, da sie nicht `null` ist werden dann mit Schritt 2 und 3 die Kinder

summiert. Im vierten Schritt werden dann alle Blätter summiert, wobei nur eines aus Platzgründen angeschrieben ist. Die Ergebnisse werden schließlich nach oben hin aufsummiert.



Besonders mächtig werden Bäume als Datenstruktur dann, wenn sie zusätzlich mit einer Ordnungsrelation versehen werden. So könnte man beispielsweise für einen Binärbaum für Integerwerte festlegen, dass jeder rechte Nachfolgeknoten eines Knotens immer einen größeren Zahlenwert enthalten muss als der aktuelle Knoten, und der linke Nachfolgeknoten eine Zahl kleiner oder gleich dem aktuellen Knoten. Diese Eigenschaft muss außerdem für alle nachfolgenden Knoten ebenfalls gelten. Einen solchen Baum bezeichnet man dann als Suchbaum, da er sich zur effizienten Suche auf Datenmengen eignet.

Die Effizienz dieser Suche basiert darauf, das ausgehend vom Wurzelknoten die rekursive Suchmethode sich immer genau für einen der beiden möglichen Teilbäume entscheiden muss, um dort die Suche fortzusetzen solange der Wert noch nicht gefunden wurde, da im anderen Teilbaum alle Werte zu groß beziehungsweise zu klein sind. Das heißt bei der Suche bewegt sich der Algorithmus nur entlang eines Astes, und besucht nicht den kompletten Baum mit allen Elementen. Dies ist vergleichbar mit der Namenssuche in einem Telefonbuch. Schlägt man bei der Suche nach „Wagner“ das Telefonbuch in der Mitte auf und findet dort Namen mit dem Anfangsbuchstaben „M“, so steht fest, dass man nur noch in der hinteren Hälfte des Buches weitersuchen muss und die vordere Hälfte gar nicht mehr genauer betrachten muss. Hier ein Beispiel eines Suchbaumes und der Suche nach dem Wert 15:



Die Suche beginnt bei 23, da der gesuchte Wert kleiner ist muss er im linken Teilbaum liegen. Der Nachfolgeknoten 17 ist noch zu groß, also wird die Suche im linken Teilbaum

von 17 fortgesetzt. Der Knoten 14 ist zu klein, darum muss sein rechter Teilbaum den gesuchten Wert enthalten. Einen weiteren Schritt später ist der Zielknoten erreicht.

Das Einführen einer Ordnungsrelation auf einem Binärbaum erfordert es allerdings, die Baumoperationen anzupassen, um die Ordnung zu erhalten. So muss zum Beispiel beim Einfügen eines neuen Elementes zunächst die richtige Position dafür im Baum gefunden werden, da das neue Element nicht einfach irgendwo angehängt werden darf. Dies könnte ansonsten die Ordnung zerstören.

A Exceptions

In allen Computerprogrammen kommt es manchmal zu Fehlern und Ausnahmesituationen. Bisher stürzten unsere Programme immer ab, wenn so etwas aufgetreten ist. Bei größeren Programmen, mit denen auch andere Menschen umgehen können sollen, ist das aber nicht wünschenswert. Bei gravierenden Fehler sollte zumindest eine Fehlermeldung ausgegeben werden und versucht werden zu retten was zu retten ist. Bei kleinen Fehlern oder Ausnahmesituationen kann das Programm unter Umständen sogar weiterarbeiten. Nehmen wir als Beispiel, dass der Benutzer eine Zahl eingeben soll, statt dessen aber Text eingibt. Das Programm nun abstürzen zu lassen wäre völlig überzogen, statt dessen ist es vollkommen ausreichend diese Ausnahmesituation durch erneutes erfragen der Zahl zu lösen.

Hierzu muss man mit Fehlern und Ausnahmesituation aber zunächst umgehen können. In Java sind hierfür aber bereits geeignete Mittel integriert. Kommt es in einem Java Programm zu einem Fehler, so wird eine sogenannte Exception ausgelöst. Solange kein Teil des Programms den Fehler abfängt und eine geeignete Fehlerbehandlung durchführt, stürzt das Programm einfach mit der Ausgabe dieser Exception ab. Wenn man mit Objekten arbeitet, ist dies zum Beispiel oft die `NullPointerException`, wenn eine Referenz die man verwenden will auf null zeigt. Oder schauen wir uns dieses Beispiel mit Arrays an:

```
1 public class MeineKlasse{
2   public static void main(String[] args{
3     // Erstellt a[0], a[1] und a[2]
4     int[] a = new int[3];
5
6     // Fehler! a[42] gibt es nicht
7     System.out.println(a[42]);
8   }
9 }
```

Beim Ausführen dieses kleinen Programms bekommen wir dann von Java folgendes zu hören:

```
1 Exception in thread "main" java.lang.↩
     ArrayIndexOutOfBoundsException: 42
2     at MeineKlasse.main(MeineKlasse.java:4)
```

Java beschwert sich also darüber, dass es eine Ausnahme (Exception) gab. Der Name der Ausnahme ist `ArrayIndexOutOfBoundsException`, auf Deutsch also etwa „der Array-Index liegt außerhalb der Grenzen“. Dies ist wenig verwunderlich, da wir mit 42 wirklich weit daneben lagen. Die zweite Zeile gibt noch an in welcher Klasse und Methode das Problem ausgelöst wurde und in den Klammern ist der Dateiname und die Zeile vermerkt, wo das Problem auftrat. Dies ist sehr nützlich für die Fehlersuche.

A.1 Fehler abfangen mit try-catch

Nun wollen wir betrachten, wie wir solche Fehler abfangen können und unser Programm geeignet reagieren lassen. Hierzu benötigen wir die Schlüsselworte `try` und `catch`.`try` steht am Anfang eines Blocks, der eine Exception werfen kann. Sollte dies geschehen, so

bricht die Ausführung im `try`-Block ab und Java überprüft, ob direkt nach dem `try`-Block ein `catch`l-Block für diese spezielle Exception existiert. Wenn dem so ist, wird dieser Block als Fehlerbehandlung ausgeführt:

```
1 try {
2   /* Code der die BeispielException werfen kann */
3 }
4 catch(BeispielException ex){
5   /* Fehlerbehandlung */
6 }
```

Schauen wir uns dies an einem Beispiel an. Wir lesen mit `parseInt(String)` aus einem übergebenen String einen Integer aus. Sollte der String keinen Integer enthalten, so wirft diese Methode eine `NumberFormatException`:

```
1 String str = "42";
2 try {
3   int ganzzahl = Integer.parseInt(str);
4   ganzzahl = ganzzahl * ganzzahl;
5   System.out.println(ganzzahl);
6 }
7 catch(NumberFormatException ex) {
8   System.out.println("Kein Integer!");
9 }
```

Solange der String wie in diesem Beispiel einen Integer enthält, wird dieser einfach quadriert und dann ausgegeben. Der `catch`-Block wird ignoriert. Nun stellen wir uns vor, `str` sei „foo“. In diesem Fall wird der Aufruf der `parseInt`-Methode nicht erfolgreich sein. Es wird eine `NumberFormatException` geworfen und von Zeile 3 wird direkt zur Zeile 8 gesprungen und die Ausgabe lautet „Kein Integer!“. Zeilen 4 und 5 werden nicht ausgeführt. Dieses Überspringen des verbleibenden `try`-Blocks sollte man bei der Verwendung von `try` beachten.

Beachten wir noch den Anfang des `catch`-Blocks: Der Typ der Exception wird angegeben, da innerhalb eines `try`-Blocks theoretisch verschiedene Exceptions auftreten können. So könnten mehrere `catch`-Blöcke untereinander stehen, der eine fängt beispielsweise eine `NumberFormatException`, der nächste eine `NullPointerException` und beide `catch`-Blöcke reagieren nur auf ihre individuell zu fangende Exception. Zudem wird nach dem Typ noch der gefangenen Exception ein Name zugewiesen, in unseren Beispielen lautet er `ex`. Dies ist nützlich, da man aus einer Exception noch mehr Informationen gewinnen kann als nur die Tatsache, dass sie aufgetreten ist. Mit `ex.getMessage()` wird die automatisch generierte Fehlermeldung angezeigt, `ex.printStackTrace()` kann man alle beteiligten Klassen und Methoden und Zeilennummern einsehen. In großen Programmen ist es sehr wichtig, dass man diese Informationen protokolliert, so dass man Fehler nicht nur feststellen sondern auch finden und beheben kann.

Verwendungshinweise für try-catch

Es nützt nichts, überall im Programm `try-catch`-Anweisungen zu haben, schließlich sollte in den allermeisten Situationen niemals eine Exception geworfen werden. Statt dessen sollten `try-catch`-Anweisungen nur ganz gezielt um einzelne Codefragmente gesetzt werden, von den man tatsächlich versteht wann sie einen Fehler werfen könnten. Manche Exceptions sollten überhaupt nie gefangen werden. Hierzu gehört zum Beispiel die `ArrayIndexOutOfBoundsException`. Wird diese geworfen, so spricht das immer

für einen Programmierfehler der zu beheben ist anstatt ihn mit `try-catch` zu verbergen. Selbiges gilt auch für `NullPointerExceptions`: Man sollte stets vorher prüfen ob eine Referenz auf null zeigt anstatt hinterher mit `catch(NullPointerException e)` die Scherben zusammenzukehren. `NumberFormatExceptions` hingegen abzufangen ist insbesondere um Benutzereingaben herum sehr sinnvoll.

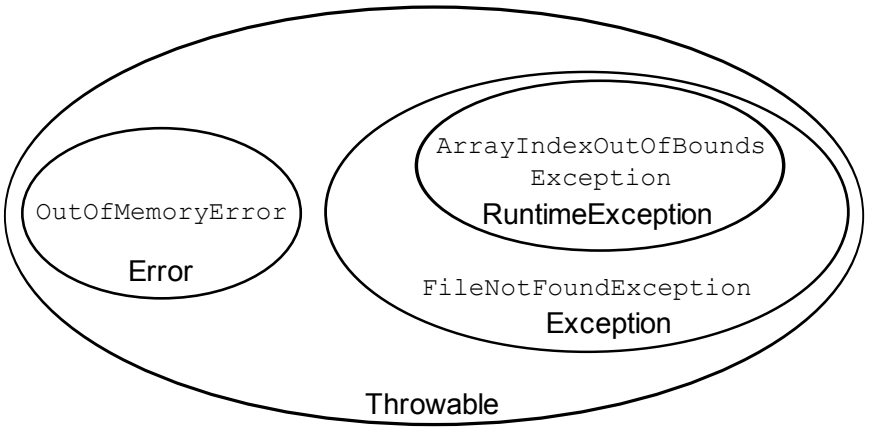


Abbildung A.1: Arten von Fehlern

Wie man in Abbildung A.1 sieht, sind alle hier behandelten Fehler vom Typ Throwable. Es gibt innerhalb dieser Menge Errors und Exceptions, RuntimeExceptions sind ein spezieller Fall von Exceptions.

Diesen Tipps liegt die Unterscheidung zwischen checked und unchecked Exceptions zu Grunde. unchecked Exceptions *können* behandelt werden (oder wir lassen das Programm abstürzen), checked Exceptions *müssen* behandelt werden. Möchte man genau sehen welche Exceptions nun zur Teilmenge der RuntimeExceptions gehört, so empfiehlt sich ein Blick in die API, wie er weiter hinten im Anhang erklärt ist.

checked Exceptions

Es gilt: Alle Exceptions, die keine RuntimeExceptions sind, müssen explizit behandelt werden, da es sonst zu einem Compilerfehler kommt. Das liegt daran, dass diese so genannten checked Exceptions eben wie der Name schon sagt vom Compiler überprüft werden. Das bedeutet, dass der Compiler sicherstellt, dass wo immer eine solche Exception auftreten kann, der Programmcode damit umgehen kann, entweder mit einem `try-catch`-Block oder durch explizites weiterreichen der Exception (siehe nächster Abschnitt). Sinn hiervon ist es, dass evtl. fehleranfälliger Code wie beispielsweise das Einlesen aus einer Datei immer so verwendet werden muss, dass das Programm nicht automatisch abstürzt, wenn ein Fehler auftritt (hier zum Beispiel ungenügende Rechte, nicht-existente Datei, ...). Es soll so programmiert werden dass für diese wahrscheinlichen Fehlerquellen vorgesorgt wird oder zumindest der Programmierer die Möglichkeit ihres Auftretens zur Kenntnis nehmen muss. Sehen wir uns hier ein Beispiel an:

```
1 public class MyClass{
2   public static void main(String[] args){
3     java.io.FileInputStream f = new java.io.↩
          FileInputStream("input.txt");
4   }
5 }
```

Dies erzeugt folgenden Compilerfehler:

```
1 MyClass.java:3: unreported exception java.io.↩
     FileNotFoundException;
2   must be caught or declared to be thrown
3   java.io.FileInputStream f = new java.io.FileInputStream(↩
       "input.txt");
```

Die Lösung ist ein `try-catch`-Block um den Programmteil (`must be caught`) oder die Exception muss explizit an den Aufrufer weitergegeben werden (`declared to be thrown`; siehe nächster Abschnitt).

unchecked Exceptions

Alle Exceptions, die vom Typ |RuntimeException| sind, heißen unchecked Exceptions da vom Compiler nicht überprüft wird ob sie auftreten können. Dies bedeutet, dass Programmcode, welcher RuntimeExceptions erzeugen kann, nicht in einem `try-catch`-Block stehen muss. Dies ist außerordentlich praktisch, da fast jeder Programmcode RuntimeExceptions produzieren kann und sonst ein Programm fast nur noch `try-catch`-Blöcken bestehen würde. Beispielsweise wär es sonst notwendig um jeden Array-Zugriff einen `try-catch`-Block zu setzten, da ja der Index außerhalb der Grenzen liegen könnte. Auch kann jeder beliebige Objektzugriff dazu führen, dass man theoretisch auf einer Null-Referenz arbeitet.

In den aller meisten Fällen sollten Exceptions vom Typ `RuntimeException` nicht behandelt werden. Das liegt daran, dass es sich fast immer um Programmierfehler handelt die immer behoben werden sollten. Allerdings gibt es auch hier Außnahmen. Immer dann, wenn man es als Programmierer nicht selbst in der Hand hat ob die Variableninhalte den Ansprüchen genügen, etwa weil sie direkt vom Benutzer eingegeben werden oder von einem anderen Programm übermittelt werden, ist es oft sinnvoll auch RuntimeExceptions zu behandeln. Das beste Beispiel hierfür ist das Beispiel von oben mit dem Parsen einer Zahl aus einem String betrachtet, welches eine `NumberFormatException` (welche zum Typ `RuntimeException` gehört) produzieren kann. Hier ist ein `try-catch`-Block wichtig.

Errors

Der Vollständigkeit halber seien noch Errors erwähnt. Errors werden immer dann von der Virtual Machine erzeugt, wenn massive Probleme auftreten die dann auch fast immer vom Programm nicht gelöst werden können. Etwa wenn die Virtual Machine selbst am abstürzen ist (`InternalError`) oder falls kein Arbeitsspeicher mehr verfügbar ist (`OutOfMemoryError`). Errors sollten normalerweise nicht behandelt werden.

A.2 Exceptions erzeugen und weiterreichen

Im vorherigen Abschnitt haben wir uns damit beschäftigt auftretende Fehler an Ort und Stelle zu behandeln. Genauer: Wir haben Exceptions mit `try-catch`-Blöcken genau an den Stellen behandelt, an denen wir die möglicherweise fehlerverursachende Methoden aufgerufen haben. Nun bleiben noch zwei Punkte zu klären. Erstens: Wie reicht man Exceptions an den Aufrufer der eigenen Methode weiter und wie erzeugt man überhaupt Exceptions?

Exceptions weiterreichen

Stellen wir uns vor, wir programmieren uns eine Zugriffsmethode die wir immer in unserem Programm bemühen, wenn wir Lese- oder Schreibzugriff auf eine beliebige Datei in einem Projektverzeichnis benötigen, zum Beispiel die tatsächlichen Arbeitsdaten unseres Programms oder die Konfigurationsdatei. Diese Zugriffsmethode verwendet dann mit Sicherheit selbst wieder Methoden die Exceptions produzieren können (vgl. Abschnitt „checked Exceptions“).

Allerdings ist es überhaupt nicht möglich innerhalb der Zugriffsmethode sinnvoll auf Exceptions zu reagieren! Handelt es sich um einen Schreibfehler bei der Konfigurationsdatei die beim Beenden automatisch gespeichert wird ist die Reaktion ganz anders als wenn zum Beispiel Arbeitsdaten nicht geöffnet werden können weil die entsprechenden Rechte fehlen. Wie in einer solchen Situation zu reagieren ist, muss der Aufrufer der Zugriffsmethode entscheiden. Die Exception soll also nicht in der Zugriffsmethode durch einen geeigneten `catch`-Block behandelt werden, sondern erst beim Aufrufer der Zugriffsmethode bearbeitet werden. Hierzu muss die Zugriffsmethode die Exception weiterreichen. Dies passiert mit dem Schlüsselwort `throws` in der Methodensignatur:

```
1 void zugriff(/*Parameter*/) throws FileNotFoundException {
2   ...
3   //ohne try-catch:
4   FileInputStream f = new FileInputStream(dateiname);
5   ...
6 }
```

Nun ist der Aufrufer der Zugriffsmethode selbst wieder in der Pflicht die an ihn weitergereichte Exception mit einem `try-catch`-Block zu fangen oder kann sie auch wieder an dessen Aufrufer weiterreichen. Die könnte dann so aussehen:

```
1 try{
2   zugriff(/*Parameter*/);
3 }
4 catch(FileNotFoundException e){
5   System.out.print(/*Situationsabhaengige Meldung */);
6 }
```

Das ist auch insbesondere dann wichtig, wenn man im Team Software entwickelt oder Programmfunktionen schreibt, die von anderen wiederverwendet werden. Zum einen sollte man eine klare Regelung treffen, wer auf welcher Ebene für welche Fehlerbehandlung zuständig ist. Zum anderen sollte man sich beim Schreiben allgemeiner Programmfunktion darüber im Klaren sein, welche Fehler man den weiteren Benutzern weiterreichen möchte, und was intern behandelt werden soll.

Durch das Schlüsselwort `throws` wird also die Ausnahmebehandlung so lange an den jeweiligen Aufrufer zu delegieren, bis eine sinnvolle Behandlung möglich ist. Wie wir gesehen haben ist das insbesondere wichtig, wenn Hilfsmethoden geschrieben werden, die selbst an Ort und Stelle gar nicht entscheiden können wie eine geeignete Reaktion auszusehen hat.

Soll eine Methode mehrere Exceptions weiterreichen können, so werden diese einfach mit Komma getrennt aufgezählt:

```
1 void method() throws ExceptionA, ExcB, ExcC {...}
```

Hierbei ist allerdings zu beachten, dass jede dieser einzelnen Exceptions auch vom Aufrufer behandelt werden müssen. Dies kann aufwendig werden. Zwar kann man anstatt sie einzeln zu behandeln auch mit `catch(Exception e){...}` alle auf einen Streich erwischen, dies ist allerdings sehr schlechter Stil, da eine differenzierte Betrachtung der Fehler so gar nicht mehr möglich ist.

Exceptions erzeugen

Da wir nun wissen, wie Exceptions weitergereicht werden, bleibt noch die Frage zu klären, wie Exceptions überhaupt ausgelöst werden. Hierzu gibt es das Schlüsselwort `throw` (ohne s am Ende!) um Objekte vom Typ `Throwable` an den Aufrufer weiterzureichen. In der Praxis sollte man aber nie direkt die Klasse Throwable erweitern, sondern immer von Exception und davon abgeleiteten Klassen erben, da Throwable als Oberklasse von allen Errors und Exceptions für die virtuelle Maschine eine besonder Klasse darstellt. Beispiel:

```
1 static float division(float a, float b){
2   if (b == 0)
3     throw new ArithmeticException("Div. durch Null!");
4   return a/b;
5 }
```

Das Beispiel ist nicht sonderlich nützlich, da die Division `a/b` diese Exception sowieso weiterreichen würde, wäre `b null`, aber es demonstriert die Verwendung gut. Die hier verwendete `ArithmeticException` ist vom Typ `RuntimeException`, also eine unchecked Exception. Deswegen ist es nicht nötig `throws ArithmeticException` in die Methodensignatur zu schreiben. Anders verhält es sich mit checked Exceptions, hier ist es Pflicht diese mit `throws` in der Signatur anzugeben. Beispiel:

```
1 void do_something() throws Exception{
2   ...
3   if (everything_ok == false)
4     throw new Exception("Something's broken!");
5   ...
6 }
```

Dieses Beispiel ist allerdings verbesserungswürdig. Zum einen ist es nicht sinnvoll eine generische Exception weiterzureichen sondern statt dessen sollte eine passende existierende Exception gewählt werden (hierzu sei die API empfohlen, siehe auch weiter hinten im Anhang den Abschnitt zur API) oder man erstellt eine eigene Exception die von Exception oder RuntimeException erbt (wie das geht steht im Anhang zu Vererbung). Zum anderen ist natürlich in der Anwendung dann drauf zu achten, dass die Fehlermeldungen konkreter sind.

B Weiterführende Konzepte und Ausblick

B.1 Objektorientierung

Wie bereits im Kapitel über Klassen und Objekte zu sehen war, ist die Objektorientierung ein alternatives Konzept zur rein imperativen Programmierung. Bisher wurde jedoch nur ein Teil davon vorgestellt, nämlich die Klassen sowie das Konzept der Kapselung. Weitere sehr wichtige Konzepte der objektorientierten Programmierung sind die Vererbung sowie die Polymorphie. Die Vererbung ermöglicht es, aus einer Klasse neue Unterklassen abzuleiten, die dann zunächst die gleichen Merkmale wie die eigentliche Klasse aufweisen. Diese Unterklassen lassen sich nun jedoch um neuen Funktionalitäten erweitern, oder es besteht die Möglichkeit, von der Oberklasse geerbte Methoden neu zu implementieren, was man auch unter Polymorphie versteht.

B.1.1 Vererbung

Um bei jeder neuen Klasse nicht wieder bei Null anfangen zu müssen, kann man das Konzept der Vererbung nutzen. Eine bestehende Klasse baut dabei auf einer Basisklasse auf. Dazu gibt es das Schlüsselwort `extends`. Die erbende (oder erweiternde) Klasse erhält dabei alle Eigenschaften und Methoden der Basisklasse, die nicht als `private` gekennzeichnet sind. Deswegen bietet es sich meist an, Klassen möglichst allgemein zu schreiben und sich Gedanken zu machen, wo Gemeinsamkeiten wiederverwendet werden können. Zum Beispiel könnte man in einem Projekt Klassen für Mitarbeiter und Abteilungsleiter benötigen. Die Klasse `Mitarbeiter` hat dann Attribute für Personalnummer, Namen, Vornamen und das Gehalt. Schreibt man nun die Klasse für die Abteilungsleiter, kann man von der Mitarbeiter-Klasse erben. Damit hat dann die Klasse der Abteilungsleiter automatisch alle Methoden und Attribute, die auch in der Mitarbeiter-Klasse definiert sind. Um sie nun um ein Array der Mitarbeiter zu erweitern, wird dies in der Abteilungsleiter-Klasse hinzugefügt. In Java sieht das folgendermaßen aus:

```
1 public class Mitarbeiter {
2   public String name,vorname;
3   public int personalNummer;
4   public float gehalt;
5 }
6
7 public class Abteilungsleiter extends Mitarbeiter {
8   public Mitarbeiter[] untergebene;
9 }
```

Objekte beider Klassen kann man auch gemeinsam unter dem Basisdatentyp verwalten. Z.B. lassen sich in einem Array vom Typ `Mitarbeiter` auch `Abteilungsleiter` speichern. Beim Zugriff hat man dann jedoch nur die Attribute zur Verfügung, die in der Mitarbeiter-Klasse definiert sind. Wie bei den primitiven Datentypen spricht man auch hier von Casten.

```
1 public static void main(String[] args) {
2   Mitarbeiter[] mitarbeiter = new Mitarbeiter[10];
3   mitarbeiter[0] = new Mitarbeiter();
```

```
4   mitarbeiter[1] = new Abteilungsleiter();
5
6   System.out.println(mitarbeiter[0].gehalt);
7
8   //Abteilungsleiter erbt Gehalts-Attribut
9   System.out.println(mitarbeiter[1].gehalt);
10
11  //Funktioniert nicht, da Array von Basisklasse
12  System.out.println(mitarbeiter[1].untergebene);
13
14  //durch Cast jedoch moeglich
15  System.out.println(
16    ((Abteilungsleiter)mitarbeiter[1]).untergebene);
17 }
```

Generell werden auch alle (`public`) Methoden der Basisklasse mit vererbt. Will man in einer erbenden Klasse diese Methoden für den Datentyp spezieller umsetzen, kann man auch die Methoden der Basisklasse überschreiben. Dazu schreibt man einfach eine Methode mit gleichem Namen und den gleichen Übergabeparametern (inkl. Reihenfolge). Mit dem Schlüsselwort `super` kann man aber auch die Implementierung der Basisklasse nutzen und erweitern. Angenommen, Mitarbeiter erhalten ihr normales Gehalt ausgezahlt, Abteilungsleiter jedoch zusätzlich eine Umsatzbeteiligung. Will man dies umsetzen, bietet es sich an, die Methode zum Ausgeben des Gehalts in der Klasse `Mitarbeiter` zu schreiben, sie jedoch in der `Abteilungsleiter`-Klasse überschreiben.

```
1  public class Mitarbeiter {
2    //...
3
4    public int getGehalt() {
5      return this.gehalt;
6    }
7  }
8
9  public class Abteilungsleiter extends Mitarbeiter {
10   //...
11
12   public int getGehalt() {
13     int bonus = ... //Bonus berechnen
14     return super.getGehalt()+bonus;
15   }
16 }
```

Die Methode zum Ausgeben des Gehalts aus der Klasse `Mitarbeiter` wird hier von der Klasse `Abteilungsleiter` überschrieben, über die `super`-Referenz ist das Ergebnis der Methode aus der Basisklasse jedoch trotzdem verfügbar und wird hier noch um den Bonus ergänzt und zurückgegeben.

In einigen Programmiersprachen gibt es auch das Konzept der Mehrfachvererbung. Dabei erbt eine Klasse Attribute und Methoden von mehreren Basisklassen. Einigen muss man sich dabei, was passiert, wenn in den verschiedenen Basisklassen mehrere gleichnamige Attribute oder Methoden existieren. In Java gibt es keine Mehrfachvererbung. Man kann jedoch Interfaces (siehe Abschnitt unten) nutzen, um ähnliches zu erreichen.

B.1.2 Abstrakte Klassen

Der Modifikator `abstract` ermöglicht es in Java, Methoden und Klassen als abstrakt zu definieren. Dies bedeutet, dass von der Klasse keine direkten Instanzen angelegt werden können. Dafür ermöglicht dies, von konkreten Klassen zu abstrahieren und zunächst gewisse Grundeigenschaften zu definieren, ohne dass direkt Objekte der Klasse erzeugt werden können. Erst wenn Unterklassen die abstrakte Klasse erweitern und die als abstrakt definierten Methoden implementieren, können auch konkrete Objekte erzeugt werden. Abstrakte Klassen können, aber müssen nicht zwingend abstrakte Methoden enthalten. Abstrakte Methoden definieren nur die Methodensignatur und enthalten keine Implementierung des Rumpfes. Genauso können abstrakte Klassen bereits fertige Methoden enthalten, die dann geerbt werden können.

Da all dies sehr abstrakt klingen mag, hier nun ein anschaulicheres Beispiel mit Fahrzeugen. Angenommen, es sollen verschiedene Fahrzeugtypen in Java nachgebaut werden, PKWs, Busse und Fahrräder. All diese Gegenstände teilen bestimmte Eigenschaften untereinander, unterscheiden sich jedoch auch. Sie besitzen alle ein Gewicht und und eine Anzahl von Plätzen. PKW und Bus haben überdies auch noch einen Motor. Neben den unterschiedlichen Merkmalen gibt es auch gemeinsame und verschiedene Aktionen. So können alle drei Objekte fahren, jedoch kann ein Fahrrad nicht anhalten, um Personen an der Haltestelle einzusammeln.

Der Sinn der Vererbung und abstrakter Klassen ist nun, so viele Gemeinsamkeiten wie möglich in die allgemeinere Klasse zu packen, und nur die unterschiedlichen oder speziellen Merkmale in den spezielleren Klassen zu implementieren. Zunächst wird hierfür eine Klasse `Fahrzeug` definiert, die abstrakt ist. Sie beschreibt konkret, das jedes Fahrzeug Plätze und Mitfahrer hat, diese ein- und aussteigen können. Außerdem definiert sie abstrakt, dass jedes Fahrzeug fahren und anhalten können muss. Wie das konkret passiert, ist nicht definiert, Jede Klasse, die jedoch von `Fahrzeug` erbt, muss nun diese Methode implementieren. Eine Ausnahme hierzu stellen wiederum abstrakte Klassen dar. Erbt eine abstrakte Klasse von einer anderen abstrakten Klasse, so muss sie die Methoden nicht unbedingt implementieren, sondern kann die Abstraktheit der Klassen quasi weitergeben.

```
1  public abstract class Fahrzeug {
2    //Eigenschaften
3    int mitfahrer;
4    int plaetze;
5
6    //Abstrakte Methoden
7    abstract public void fahren();
8    abstract public void anhalten();
9
10   //Implementierte Methoden
11   public void einsteigen(int einsteigendePersonen) {
12     if(mitfahrer + einsteigendePersonen <= plaetze) {
13       mitfahrer = mitfahrer + einsteigendePersonen;
14     }
15   }
16
17   public void aussteigen() {
18     if(mitfahrer<0) {
19       mitfahrer--;
20     }
21   }
22 }
```

Es ist nun nicht möglich, mit `new Fahrzeug()` neue Fahrzeuge anzulegen, da diese Klasse abstrakt ist und dies die Erzeugung von Objekten verhindert. Nun könnte man aber konkrete Unterklassen anlegen:

```
1 public class Fahrrad extends Fahrzeug {
2   public void anhalten() {
3     betaetigeBremse();
4   }
5
6   public void fahren() {
7     treteInDiePedale();
8   }
9 }
```

Die Methoden zum Ein- und Aussteigen werden automatisch von der Klasse `Fahrzeug` geerbt, während die Methoden zum Anhalten und Bremsen zwingend implementiert werden müssen, da sie abstrakt sind und somit bei einer nicht abstrakten Klasse implementiert sein müssen.

Als nächstes wird eine Klasse `MotorFahrzeug` erstellt, sie stellt die Oberklasse aller Fahrzeuge mit Motor dar:

```
1 public abstract class MotorFahrzeug extends Fahrzeug {
2   public void anhalten() {
3     bremsen();
4   }
5
6   public void fahren() {
7     gasGeben();
8   }
9 }
```

Die entstandene Klasse `MotorFahrzeug` enthält zwar keine abstrakten Methoden mehr, da aber die Klasse als abstrakt definiert wurde, können hiervon noch immer keine Instanzen erzeugt werden. Möglich wird dies erst durch abgeleitete, bestimmte Klassen:

```
1 public class PKW extends MotorFahrzeug {
2   //leer, alles noetige wird geerbt
3 }
```

```
1 public class Bus extends MotorFahrzeug {
2   public void halteAnBushalteStelle() {
3     anhalten();
4     einsteigen();
5     fahren();
6   }
7 }
```

Der Grundgedanke bei Vererbung ist die Realisierung von Hierarchien, hauptsächlich von *is-a*-Beziehungen, also in diesem Fall: Ein Fahrrad ist ein Fahrzeug. Ein motorisiertes Fahrzeug ist ein Fahrzeug. Ein Auto und ein Bus sind jeweils ein motorisiertes Fahrzeug. Somit lassen sich nun nicht nur von allen drei Typen Objekte erzeugen, genauso können auch Sammlungen von Fahrzeugen gespeichert werden. In einem Array von Fahrzeugen lassen sich PKWs, Busse und Fahrräder gemeinsam ablegen.

B.1.3 final-Modifikator

Ein gegensätzliches Konzept zum `abstract`-Modifikator stellt das Schlüsselwort `final` dar. Hiermit verhindert man bei der Deklaration von Methoden und Attributen, dass diese von erbenden Klassen überschrieben werden können. Das bedeutet, dass erbende Klassen diese Methoden oder Attribute weiterhin besitzen, jedoch müssen sie die geerbte Variante übernehmen und können sie nicht überschreiben. Es ist sogar möglich, Klassen als `final` zu definieren. Das führt dazu, dass von dieser Klasse nicht mehr geerbt werden kann, eine andere Klasse kann also die finale Klasse nicht mehr erweitern. Ein Beispiel hierfür ist die die Standard-Klasse `String`. Somit ist es nicht mehr möglich, eine eigene `String`-Klasse zu schreiben, die von der Standard-String-Klasse erbt. Auch in Blöcken kann `final` bei Variablen verwendet werden. Dies hat zur Konsequenz, dass nach erstmaliger Zuweisung eines Wertes keine weiteren Zuweisungen mehr durchgeführt werden. Auf diese Weise werden auch Konstanten in Java realisiert:

```
1  public class Mensch
2  {
3    //Unveraenderliche Merkmale
4    final int geburtsjahr;
5    final boolean geschlecht;
6
7    //Konstanten fuer Geschlechter
8    public static final boolean MAENNLICH = false;
9    public static final boolean WEIBLICH = false;
10
11   public Mensch(boolean geschlecht, int geburtsjahr) {
12     this.geschlecht = geschlecht;
13     this.geburtsjahr = geburtsjahr;
14     //Ab jetzt sind Merkmale nicht mehr aenderbar
15   }
16
17   public static void main(String[] args) {
18     Mensch alice = new Mensch(WEIBLICH,1987);
19     Mensch bob = new Mensch(MAENNLICH,1989);
20   }
21 }
```

B.1.4 Interfaces

In Java ist es nicht möglich, von mehreren Klassen zu erben. Die Entscheidung, auf Mehrfachvererbung zu verzichten, wurde von den Java-Entwicklern wohl überlegt getroffen, da Mehrfachvererbung nicht nur komplex und damit fehlerträchtig ist, sondern auch schnell zu ungewollten Mehrdeutigkeiten führen kann. Als Ersatz dafür ist es in Java dafür möglich, mehrere Interfaces zu implementieren, womit die fehlende Mehrfachvererbung sinnvoll ausgeglichen wird. Interfaces sind abstrakte Definition von Methoden, die ein Klasse implementieren muss, um dem Interface gerecht zu werden. Damit sind Interfaces vergleichbar mit abstrakten Klassen, die nur abstrakte Methoden enthalten, mit der Ausnahme, dass Klassen eben mehrere Interfaces implementieren, und nicht nur eine Klass erweitern können. Während man bei erbenden Klassen von einer ist-ein-Beziehung spricht, entspricht das Implementieren von Interfaces einer verhält-sich-wie-Beziehung. Ein Interface steht also für bestimmte Eigenschaften und Merkmale, für die es aufgrund der vorgegebenen Methoden steht. Anders als bei abstrakten Klassen lassen sich bei Interfaces keine Methoden direkt implementieren, die Interfaces definieren lediglich die Methodensignatur und

sind abstrakt — und das implizit, das heißt sie müssen nicht als abstrakt deklariert werden, weil sie es automatisch sind.

Angenommen, es sollen die Studentenausweise und Girokonten in Java modelliert werden. Obwohl beides grundlegend verschiedene Objekte sind, teilen sie sich trotzdem bestimmte Eigenschaften. Hier ist der Einsatz eines Interfaces ideal, welches wie folgt aussehen könnte:

```
1 public interface Konto {
2   public float erfrageWert();
3 }
```

Natürlich kann ein Interface auch mehrere Methoden definieren. Im Beispiel müssen nun die Klassen zur Implementierung des Interfaces `Konto` eine Methode `public float erfrageWert()` besitzen. Welchen Code die Implementierungen besitzen, bleibt den Klassen überlassen, nur die Signatur der Methode muss mit der angegebenen im Interface übereinstimmen. Damit ist sichergestellt, dass alle Klassen, die das Interface Konto implementieren — was auch immer sie darstellen — eine Möglichkeit bieten, den Wert zu erfragen.

```
1  public class GiroKonto implements Konto {
2    int kontoNr;
3    float kontostand = 0;
4
5    public GiroKonto(int kontoNr) {
6      this.kontoNr = kontoNr;
7    }
8
9    //Methode wird vom Interface Konto gefordert!
10   public float erfrageWert() {
11     return kontostand;
12   }
13
14   public void zahleEin(float Betrag) {
15     //...
16   }
17
18   public float zahleAus(float Betrag) {
19     //...
20   }
21 }
```

Würde eine Klasse bei der Klassendeklaration angeben, `Konto` zu implementieren, ohne die Methode `erfrageWert()` zu implementieren, so würde dies sofort zu einem Compilerfehler führen.

Obwohl nun die Klasse für den Studentenausweis komplett anders als die Klasse des Girokontos ist, so enthalten beide Klassen die Methode `public float erfrageWert()`, um dem Interface `Konto` gerecht zu werden.

```
1  public class StudentenAusweis implements Konto {
2    int matrikelNr;
3    float guthaben = 0;
4
5    public StudentenAusweis(int matrikelNr) {
6      this.matrikelNr = matrikelNr;
```

```
7    }
8
9    //Methode wird vom Interface Konto gefordert!
10   public float erfrageWert() {
11     return guthaben;
12   }
13
14   public void ladeKarteAuf(float Betrag) {
15     //...
16   }
17
18   public void kopiere(int anzahlKopien) {
19     //...
20   }
21
22   public void bezahleInMensa(float preis) {
23     //...
24   }
25 }
```

Ähnlich wie Klassen lassen sich auch Interfaces vererben, das heißt ein Interface kann mithilfe von `extends` einen existierendes Interface um neue Methoden erweitern.

B.2 Weiterführendes zu Datentypen

B.2.1 Autoboxing

In den vorherigen Kapiteln wurden bereits die verschiedenen Datentypen von Java vorgestellt. Neben den Klassen gibt es in Java noch verschiedene primitive Datentypen, wie zum Beispiel `int` oder `char`. Klassen sind die Basis für komplexere Datentypen, wie sie in Java zum Teil vorgefertigt mitliefert werden, man denke an `String`. Genauso lassen sich mithilfe von Klassen aber auch eigene Datentypen definieren, wie anhand der Listen und Listenelemente im Kapitel über abstrakte Datenstrukturen gesehen. Instanzen solcher Datentypen sind bekanntermaßen immer Objekte. Doch wie sieht es mit den primitiven Datentypen aus, sind sie auch Objekte?

In Java gibt es eine Möglichkeit, auch primitive Datentypen als Objekte zu behandeln. Hierfür existieren sogenannte Wrapper-Klassen, die die jeweiligen primitiven Werte in Form eines Objektes kapseln. Das heißt, diese Klassen enthalten jeweils nur ein einziges Attribut mit dem primitiven Datentyp. Diese Wrapperklassen sind wichtig, da bei der generischen Programmierung oder den vorgefertigten Datenstrukturen der Collections-API nur Klassen als Typen verwendet werden können, und keine primitiven Datentypen (siehe auch die nächsten Abschnitte zur Generics und der Collections-API von Java).

```
1 // Integerwert als primitiver Datentyp
2 int i = 12;
3
4 // Integerwert als Objekt
5 Integer j = new Integer(12);
6
7 // Abfrage des primitiven Wertes des Objektes
8 int x = j.intValue();
```

Neu seit Java 5 ist das sogenannte Autoboxing, dass dem Programmierenden die Arbeit abnimmt, primitive Typen und Wrappertypen explizit umzuwandeln, da Java diese Umwandlung von dem einen in den anderen Typ automatisch durchführt.

```
1  int x = 36;
2
3  // bisher:  Integer y = Integer.valueOf(37);
4  Integer y = 37;
5
6  // bisher: y = Integer.valueOf(y.intValue()-1);
7  y--;
8
9  // bisher: x == y.intValue()
10 if(x == y)
11   System.out.println("identisch");
```

Die Nutzung der Wrapper-Klassen bietet jedoch auch ein gewisses Gefahrenpotenzial bei unvorsichtiger Verwendung. Angenommen es werden zwei Objekte vom Typ Integer erzeugt, und die Werte sollen nun verglichen werden. Während bei primitiven Datentypen der Vergleichsoperater die Werte vergleicht, werden nun bei zwei Objekten die Referenzen verglichen. Es handelt sich nun zwar um zwei Objekte, die den gleichen numerischen Wert speichern, aber eben nicht um das selbe Objekt, es wurden schließlich zwei verschiedene Objekte instanziiert.

```
1  // primtive Datentypen
2  int x = 12;
3  int y = 12;
4  boolean z = (x==y); //true
5
6  // Integer-Objekte
7  Integer a = new Integer(12);
8  Integer b = new Integer(12);
9
10 // FALSCH: Prueft Objektreferenzen auf Gleichheit
11 boolean c = (a == b); //false
12
13 // RICHTIG: Prueft Objekte auf Gleichheit
14 boolean d = (a.equals(b)); //true
```

Für jeden primitiven Datentyp existiert nun eine entsprechende Wrapperklasse, selbsterklärend heißen diese wie folgt: `Byte`, `Short`, `Integer`, `Long`, `Double`, `Float`, `Boolean`, `Character` sowie `void`.

Abschließend sei noch erwähnt, dass nur dann auf die Wrapperklassen zurückgegriffen werden sollte, wenn diese auch wirklich benötigt werden. Da bei jeder Erzeugung nicht nur der primitive Wert, sondern jeweils noch ein dazugehöriges Objekt erstellt werden muss, sind geboxte Variablen weniger performant.

B.2.2 Enum

Ein Enum (*enumerated type*) ist ein Datentyp, der dazu dient, Aufzählungen zu ermöglichen. Angenommen, es soll ein Kartenspiel implementiert werden. Hierfür wird eine Klasse Karte benötigt, und jedes Objekt dieser Klasse muss zu einer der vier Farben (Kreuz,

Herz, Pik, Karo) sowie einem der möglichen Werte (2 bis 10, Bube, Dame, König, Ass) gehören.

Nun könnte man sich natürlich darauf einigen, dass jeder mögliche Wert durch eine Zahl repräsentiert wird, und dann mit den Zahlen als Stellvertreter gearbeitet wird, allerdings ist die Nutzung von Enums viel praktischer. Zunächst werden die beiden Typen deklariert. In diesem Fall wird das in eigenen Dateien durchgeführt, Enums können allerdings auch innerhalb von Klassen deklariert werden.

```
1 public enum Farbe {
2   KREUZ,HERZ,PIK,KARO
3 }
4
5 public enum Wert {
6   ZWEI,DREI,VIER,FUENF,SECHS,SIEBEN,ACHT,NEUN,ZEHN,
7   BUBE,DAME,KOENIG,ASS
8 }
```

Wie bei konstanten Werten ist es auch bei Enum-Werten gebräuchlich, den Namen komplett groß zu schreiben.

Eine Klasse `Karte` könnte nun so aussehen:

```
1 public class Karte {
2   final Wert wert;
3   final Farbe farbe;
4
5   public Karte(Farbe farbe, Wert wert) {
6     this.wert = wert;
7     this.farbe = farbe;
8   }
9 }
```

Um ein komplettes Deck zu erzeugen, nutzt man nun die Möglichkeit eines Enums, über alle möglichen Werte zu iterieren. Über die Methode `values()` des Enums lässt sich ein Array mit den beinhalteten Werten erzeugen. Dies wird hier genutzt, um die Anzahl aller Karten zu bestimmen:

```
1 int i = 0;
2 int s = Wert.values().length * Farbe.values().length;
3 Karte[] deck = new Karte[s];
4
5 // Ueber alle Moeglichkeiten iterieren
6 for(Wert wert : Wert.values()) {
7   for(Farbe farbe : Farbe.values()) {
8     deck[i++] = new Karte(farbe, wert);
9   }
10 }
```

Im diesem Beispiel wurde eine `foreach`-Schleife benutzt, sie wird im nächsten Kapitel genauer vorgestellt. Die Werte eines Enums besitzen neben ihrem Namen zusätzlich noch einen numerischen Wert, dieser entspricht wie bei einem Array die Position. Das heißt der erste Wert bekommt eine 0, der nächste eine 1 und so weiter. Abgefragt werden kann dieser Wert mit `ordinal()`. Somit könnten wir den Wert zweier Karten wie folgt vergleichen:

```
1 public static int vergleicheKarten(Karte k1, Karte k2) {
```

```
2   return k2.wert.ordinal() - k1.wert.ordinal();
3 }
```

Ist der Rückgabewert negativ, so ist Karte 1 mächtiger, bei positivem Wert ist es die zweite Karte, und bei einer Null sind die Karten wertgleich.

Eine Besonderheit an Java ist, dass enums objektähnliche Eigenschaften haben und unter anderem auch eigene Methoden besitzen können. Dies soll anhand eines Beispiels erläutert werden. Angenommen, es soll ein Enum für bestimmte Farben erzeugt werden. Normalerweise werden die Farben über den Rot- Grün- und Blau-Anteil definiert. Dieser besteht gewöhnlich aus 8 Bit, das heißt je Farbkanal kann ein Wert zwischen 0 und 255 zugewiesen werden. Im Web ist es dagegen üblich, die Farbe als eine einzige Hexadezimalzahl anzugeben. Hierfür werden jeweils die RGB-Werte als Hexadezimalzahl aneinandergereiht und eine Raute vorgestellt. Grün entspricht somit #00ff00 (R: 0->00, G: 255->ff, B: 0->00). Nun soll der Enum-Typ bestimmte Farben definieren und alternativ noch im Webformat ausgeben können:

```
1  public enum WebFarben {
2    // Deklaration der Enum-Werte
3    ROT(255,0,0),
4    GRUEN(0,255,0),
5    SCHWARZ(0,0,0),
6    WEISS(255,255,255),
7    GRAU(127,127,127);
8
9    // Attribute der einzelnen Farben
10   final int r;
11   final int g;
12   final int b;
13
14   // Enum-Konstruktor
15   WebFarben(int rot, int gruen, int blau) {
16     r = rot;
17     g = gruen;
18     b = blau;
19   }
20
21   // Ausgabe als Hex
22   public String toHex() {
23     // Umrechnung mit Hilfe von format()
24     return String.format("#%02x%02x%02x", r,g,b);
25   }
26 }
```

Nun ließe sich der entsprechende Hex-Wert zu der Farbe einfach über die Methode ausgeben lassen:

```
1 String hexGrau = GRAU.toHex(); // ergibt #7f7f7f
```

B.2.3 Generics

Seit Version 5 unterstützt Java Generics, was nun ermöglicht, in Java generisch zu programmieren. Dieses sehr mächtige Konzept scheint auf den ersten Blick meist nicht ganz

verständlich, bringt aber viele Vorteile und verbessert auch die Sicherheit und Fehleranfälligkeit von Programmen während der Entwicklung. Durch Generics ist es möglich, Variablen für Typen einzuführen. An dieser Stelle sei auf die Datenstruktur Liste verwiesen, die im Kapitel der abstrakten Datentypen entwickelt wurde. Die Elemente der Listenklasse konnten nur Werte vom Typ `int` aufnehmen. Bei vielen Datenstrukturen möchte man jedoch ganz allgemeine Containerklassen haben, in die man beliebige Datentypen speichern können soll. Soll nun eine Namensliste verwaltet werden, so müsste der Inhalt der bisherigen ListenElement-Klasse auf `String` geändert werden. Für eine Ergebnisliste eines 100-Meter-Laufs wären wiederum Float-Werte nötig. Die bisherige Listenklassen ist also wenig wiederverwendbar. Nun könnte man auf die Idee kommen, den Datentyp des Feldes inhalt auf `Object` zu ändern, da in Java jeder beliebige Wert und jedes Objekt auch automatisch vom Typ Object ist, da in Java `Object` die Superklasse jeder Klasse ist. Dies funktioniert tatsächlich:

```
1  public class ListenElement {
2    Object inhalt;
3    ListenElement nachfolger;
4
5    public ListenElement(Object inhalt) {
6      this.inhalt = inhalt;
7    }
8  }
```

Diese Änderung hat jedoch einen gewaltigen Nachteil. Man verliert die Kontrolle darüber, welchen Inhalt die Liste eigentlich besitzt. Angenommen man nutzt die `Object`-Liste nun für Integer-Werte und man möchte dann die Summe aller Elemente berechnen:

```
1  // Liste anlegen
2  Liste meineListe = new Liste(new ListenElement(22));
3  meineListe.fuegeHinzu(new ListenElement(43));
4
5  // Ooopps, ich bin gar kein Integer
6  meineListe.fuegeHinzu(new ListenElement("Autsch"));
7  meineListe.fuegeHinzu(new ListenElement(13));
8
9  int summe = 0;
10 ListenElement aktuellesElement = meineListe.listenKopf;
11 while(null != aktuellesElement) {
12   summe = summe + (Integer) aktuellesElement.inhalt;
13   aktuellesElement = aktuellesElement.nachfolger;
14 }
```

Dieser Programmcode lässt sich ohne Fehler kompilieren, erst bei der Ausführung kommt es zum Problem, denn es kommt zu einer `ClassCastException`. Man konnte nämlich problemlos auch einen `String` in die Liste einfügen, da auch dieser vom Typ `Object` ist, und somit ein gültigen Wert besitzt. Doch beim Auslesen wird ein Objekt vom Typ `Integer` erwartet. Da keine `Objects` addiert werden können, müssen die Elemente zunächst zu `Integer` gecastet werden. Genau hier bricht nun das Programm ab, denn das dritte Element der Liste enthält gar keinen `Integer`, sondern einen `String`. Dieser lässt sich jedoch nicht zu einem `Integer` casten. Das Problem dieser Variante ist es, dass man sich selbst Einschränkungen auferlegt („Ich benutze nur Integer als Listenwerte“), die jedoch nirgendwo überprüft werden. Stellt man sich nun vor, dass man auch noch im Team programmiert oder das geschriebene Programm von Dritten weiterverwendet wird, werden solche Einschränkungen nicht nur völlig wirkungslos, sondern auch gefährlich, da sie erst zur Laufzeit zu Problemen führen.

Abhilfe schaffen hier Generics, die quasi als Platzhalter für den benutzten Typ stehen, und erst bei der Kompilierung mit einem echten Wert ersetzt werden:

```
1  public class ListenElement<T> {
2    T inhalt;
3    ListenElement<T> nachfolger;
4
5    public ListenElement(T inhalt) {
6      this.inhalt = inhalt;
7    }
8  }
```

Überall, wo bisher `Object` stand, steht nun T als Platzhalter (T als Platzhalter ist zwar keine Pflicht, aber üblich, da es für Type steht). Außerdem steht nun hinter dem Klassennamen in spitzen Klammern ebenfalls ein T.

Ebenso die Klasse `Liste`:

```
1  public class Liste<T> {
2    ListenElement<T> listenKopf;
3
4    public Liste(ListenElement<T> listenKopf) {
5      this.listenKopf = listenKopf;
6    }
7
8    public int laenge() {
9      // Muss noch implementiert werden...
10   }
11
12   public void fuegeHinzu(ListenElement<T> element) {
13     // Muss noch implementiert werden...
14   }
15
16   //Ueberladene Methode fuer einfacheres Hinzufuegen
17   public void fuegeHinzu(T wert) {
18     fuegeHinzu(new ListenElement<T>(wert));
19   }
20
21   //...
22 }
```

Wird nun ein Objekt vom Typ `ListenElement` oder `Liste` erzeugt, so muss explizit angegeben werden, durch was das T ersetzt werden soll. Das vorherige Beispiel sähe nun so aus:

```
1  //Liste anlegen
2  Liste<Integer> meineListe =
3      new Liste<Integer>(new ListenElement<Integer>(22));
4  meineListe.fuegeHinzu(43);
5  meineListe.fuegeHinzu(13);
6
7  int summe = 0;
8  ListenElement<Integer> aktuellesElement =
9      meineListe.listenKopf;
10 while(null != aktuellesElement){
```

```
11    summe = summe + aktuellesElement.inhalt;
12    aktuellesElement = aktuellesElement.nachfolger;
13  }
```

Beim Kompilieren würde nun das T durch `Integer` ersetzt werden, und das Einfügen eines Strings wäre nicht mehr möglich. Außerdem fällt nun das Casten auf `Integer` weg, da die Listenwerte vom Typ `Integer` sind, und nicht mehr vom Typ `Object`. Ganz ähnlich werden Generics auch in allen Klassen der Collections-API eingesetzt, sodass die dortigen Datenstrukturen für beliebige Objekte einsetzbar sind.

Generics lassen sich noch weitaus komplexer nutzen, jedoch wird an dieser Stellen nicht mehr vertiefend darauf eingegangen.

B.2.4 Collections

Im Kapitel zu abstrakten Datentypen wurden bereits zwei grundlegende Datenstrukturen vorgestellt. In Java sind viele wichtige Datenstrukturen bereits fertig implementiert, sodass man direkt darauf zurückgreifen kann, ohne diese selbst neu programmieren zu müssen. Diese Datenstrukturen sowie ein Reihe von Hilfsklassen wurden bei Java in der Collections-API gesammelt und stehen seit Version 5 zur Verfügung.

Datenstrukturen

Die allgemeinen Datenstrukturen, die die Collections-API in Java zur Verfügung stellt, lässt sich wie folgt in einer Tabelle darstellen:

	Interface		
Implementierung	`Set`	`List`	`Map`
Hashtabelle	`HashSet`		`HashMap`
Variables Array		`ArrayList`	
Baum	`TreeSet`		`TreeMap`
Verkettete Liste		`LinkedList`	
Hashtabelle und verkettete Liste	`LinkedHashSet`		`LinkedHashMap`

Ein `Set` ist eine Sammlung von Daten, die keine Duplikate enthält. Sie ist somit vergleichbar mit dem mathematischen Mengenbegriff. Eine `List` ist eine geordnete Sammlung von Daten, auf die über eine Positionsindex zugegriffen werden kann. Eine `Map` (im Deutschen manchmal auch als assoziatives Array bezeichnet) ist eine Sammlung von Schlüssel-Wert-Paaren, wobei jeder Schlüssel eindeutig sein muss, also nur einmal verwendet werden darf. Ein Beispiel für eine Map wäre eine Datensammlung aller Studenten, wobei die Matrikelnummer der Schlüssel und der Name der Wert wäre.

Eine Hashtabelle ist eine Datenstruktur, bei der die Elemente über einen Hashcode referenziert werden, ein variables Array ist ein Array, dessen Größe sich zur Laufzeit ändern kann. Ein Baum ist eine geordnete Datenstruktur, die ihre Daten baumartig verwaltet (siehe auch Binärbaum aus dem Kapitel Abstrakte Datentypen). Eine verkettete Liste ist eine Liste, deren Elemente ihren jeweiligen Nachfolger kennen (im Falle einer doppelt verketteten Liste auch ihren Vorgänger). Natürlich lassen sich solche Strukturen auch mischen. So lassen sich Elemente sowohl in einer Hashtabelle einordnen, als auch innerhalb einer verketteten Liste verlinken. Dies verbindet die Vorteile beider Konzepte. In diesem Fall könnte über die Hashtabelle ein Eintrag relativ schnell gefunden, und seine nächsten n Nachfolger dann mithilfe der verketteten Liste aufgerufen werden. Entsprechend erhöht sich dann aber auch der Aufwand der Operationen wie Einfügen oder Löschen von Elementen.

Aus den gewünschten Eigenschaften der Datensammlung sowie den vorhandenen Implementierungen lassen sich nun konkrete Datenstrukturen der Java Collections-API wählen. Sollen zum Beispiel alle Studierenden mit ihrer Matrikelnummer gespeichert werden, so bietet sich wie bereits oben erwähnt eine `Map` an. Doch soll nun eine `TreeMap`, eine `HashMap` oder gar eine `LinkedHashMap` verwendet werden? Dies hängt ganz vom Einsatzzweck ab, denn jede Implementierung hat seine eigene Vor- und Nachteile. So lässt sich bei einer `HashMap` zu einer vorhandenen Matrikelnummer schneller der zugehörige Student finden, als mit einer `TreeMap`. Bei der `HashMap` wird direkt die Nummer als Hashcode benutzt und somit in konstanter Laufzeit auf den Datensatz zugegriffen. Bei der `TreeMap` bilden die Matrikelnummern jedoch einen Rot-Schwarz-Baum (eine besonderer binärer Suchbaum), so dass das Auffinden eines Datensatzes bei gegebener Nummer den Aufwand $O(\log(n))$ benötigt. Will man dagegen alle Studierenden mit den Matrikelnummern 60000-70000 auflisten, ist die `HashMap` völlig ungeeignet, weil hier zunächst alle Schlüssel gesammelt und sortiert werden müssen. Bei der `TreeMap` ist dies hingegen einfacher möglich durch die Baumordnung. Würde man nun eine `LinkedHashMap` verwenden, deren Schlüssel nicht nur gehasht sind, sondern zusätzlich in einer sortierten und verketteten Liste stünden, wäre dies die ideale Wahl. Wie dieses Beispiel zeigt ist die Wahl der richtigen Datenstruktur nicht immer einfach, und neben den oben genannten allgemeinen Datenstrukturen bietet die Java Collections-API noch sehr viele weitere Datenstrukturen mit spezielleren Eigenschaften an – unter anderem Stacks, Queues und nebenläufige Datenstrukturen.

Hilfsklassen

Die Collections-API beinhaltet außerdem eine Vielzahl von Hilfsklassen, die Algorithmen für typische Operationen auf Datenstrukturen bereitstellen. Angenommen das Kartendeck aus dem vorherigen Enum-Abschnitt soll gemischt werden. Mithilfe der Collections-API lässt sich dies elegant erledigen. Allerdings funktioniert die Misch-Funktion nur mit Listen, sodass das Array zunächst in eine Liste umgewandelt werden muss. Auch dies erledigt die API selbstständig:

```
1 Karte[] deck;
2 // ...
3
4 // Array in Liste von Karten umwandeln
5 List<Karte> deckListe = Arrays.asList(deck);
6 // Mischen
7 Collections.shuffle(deckListe);
8 // deckListe ist nun eine zufaellige Liste der Karten
```

Eine andere typische Aufgabe ist das Suchen und Sortieren in Datenstrukturen. Auch hier bietet die Collections-API bereits fertige Methoden an.

```
1 // Anlegen einer Namensliste
2 List<String> namen = new ArrayList<String>();
3 namen.add("Bob");
4 namen.add("Eve");
5 namen.add("Alice");
6 // Alphabetische Sortierung (da vom Typ String)
7 Collections.sort(namen);
8 // Liste namen nun sortiert
```

Zur Sortierung setzt Java übrigens intern Merge Sort ein. Sobald eine Datenstruktur eine Ordnung besitzt, also sortiert ist, lässt sich darauf eine binäre Suche starten, die die Po-

sition des gesuchten Elements zurückgibt. Wird ein Element nicht gefunden, so wird `-1` zurückgegeben:

```
Collections.binarySearch(namen, "Eve"); // ergibt 2
Collections.binarySearch(namen, "Benjamin"); // ergibt -1
```

Die Collections-API besitzt noch viele weitere Hilfsfunktionen, wie zum Beispiel: Umdrehen von Reihenfolgen, Suche von Minimal- und Maximalwerten, Kopieren von Datenstrukturen, Erzeugen von nur lesbaren Kopien einer Datenstruktur und vielem mehr.

Iteratoren & foreach-Schleifen

Durch die Collections-API wurden außerdem Iteratoren eingeführt, die einen vereinfachten geordneten Zugriff auf Datenstrukturen ermöglichen sollen. Intern sind Iteratoren ein Interface, dass implementiert werden muss, glücklicherweise wurde dies bei allen wichtigen Datenstrukturen aus der Collections-API bereits gemacht. Der Iterator besitzt im wesentlichen für das Lesen zwei wichtige Methoden: Mit `next()` wird das nächste Element in der Datenstruktur zurückgegeben, mit `hasNext()` lässt sich abfragen, ob noch weitere Element vorhanden sind.

Außerdem besitzt Java seit Version 5 eine `foreach`-Schleife, die intern auf das Iteratoren-Konzept zurückgreift. Eine `foreach`-Schleife ist eine spezielle `for`-Schleife, die über eine Datenstruktur iteriert, und nacheinander Zugriff auf alle enthaltenen Elemente bietet, ohne dass sich der Programmierer um die Schleifen-Verwaltung kümmern muss.

Eine `foreach`-Schleife enthält im Schleifenkopf zunächst eine Deklaration, von welchem Typ die Werte sind und unter welcher Variable sie im Schleifenkörper existieren sollen. Dann folgt ein Doppelpunkt und die Referenzierung der Datenstruktur, über die iteriert werden soll. Foreach-Schleifen funktionieren auch mit Arrays. Hier nun ein paar vergleichende Beispiele:

```
// Array anlegen
int[] meinArray = {2,1,3};

// Liste anlegen
List<Integer> meineListe = new ArrayList<Integer>();
meineListe.add(2);
meineListe.add(1);
meineListe.add(3);

// Array mit for-Schleife durchlaufen
for (int i = 0; i < meinArray.length; i++) {
  System.out.println(meinArray[i]);
}

// Array mit foreach-Schleife durchlaufen
for (int x : meinArray) {
  System.out.println(x);
}

// Liste mit for-Schleife durchlaufen
for (Iterator it = meineListe.iterator();it.hasNext();) {
  System.out.println(it.next());
}

```

```
25 // Liste mit foreach-Schleife durchlaufen
26 for (int x : meineListe) {
27   System.out.println(x);
28 }
```

Alle vier Varianten erzeugen die gleiche Ausgabe.

B.3 Grafische Benutzeroberflächen

Java besitzt von Haus aus die Möglichkeit, grafische Benutzeroberflächen zu erzeugen. Da Java eine plattformunabhängige Sprache ist, die auf vielen verschiedenen Betriebssystemen und Architekturen ausführbar ist, wurde hierfür das Abstract Window Toolkit (AWT) eingeführt, das als Abstraktionsschicht zwischen einem Fenstermodell für den Programmierenden und der eigentlichen grafischen Benutzerschnittstelle des Betriebssystems dient. Im Klartext heißt dies, dass sich mithilfe von AWT zum Beispiel auf gleiche Weise ein Fenster erzeugen und mit Inhalt füllen lässt, unabhängig davon, ob das Programm nun unter Windows, Linux oder auf einem Mac ausgeführt wird. Der entsprechende Quellcode des Javaprogramms ist identisch und je nach laufendem Betriebssystem erzeugt dann die Java Virtual Machine ein entsprechendes Fenster des jeweiligen Betriebssystems.

Neben den nötigen Klassen zur Realisierung von Fenstern und Dialogen enthält das Package `java.awt` außerdem fertige Widgets, also Elemente grafischer Benutzeroberflächen wie Buttons, Textfelder, Eingabefelder, Listen oder Menüs.

Mit folgendem Code wird ein einfaches Fenster mit zwei Buttons, einem Textfeld sowie einem Label erzeugt.

```
1  import java.awt.*;
2
3  // Fenster erbt von Frame
4  public class MeinFenster extends Frame {
5    public MeinFenster() {
6      // Konstruktor von Frame aufrufen
7      // Parameter ist Fenstertitel
8      super("Mein erstes Fenster");
9
10     // Textlabel erzeugen
11     Label halloLabel = new Label("Hallo Welt");
12     // Button erzeigen
13     Button exitButton = new Button("Beenden");
14
15     // fliessendes Layout waehlen
16     this.setLayout(new FlowLayout());
17
18     // Widgets auf Fenster platzieren
19     this.add(halloLabel);
20     this.add(exitButton);
21
22     // Fenstergroesse setzen
23     setSize(300,100);
24     // Fenster anzeigen
25     setVisible(true);
26   }
27
```

```
28    public static void main(String[] args) {
29      // Fenster erzeugen
30      new MeinFenster();
31    }
32  }
```

Durch das Aufrufen von `new Fenster()` in der Main-Methode wird das Fenster erzeugt und angezeigt. Was nun noch fehlt, ist eine Möglichkeit, auf Eingabe zu reagieren, also eine Interaktion mit der Benutzeroberfläche zu ermöglichen. In Java wird hierfür ein Konzept verwendet, dass als Observer-Pattern bekannt ist. Sogenannte Listener registrieren sich bei der Komponente. Sobald auf der Komponente nun eine Aktion ausgeführt wird, beispielsweise ein Mausklick auf eine Button-Komponente, so informiert die Komponente alle seine Listener über das Ereignis. Je nach Widget gibt es verschiedene Listener, die für bestimmte Ereignisse zuständig sind. Diese Listener sind in Form von Interfaces vorhanden. Will man nun einen eigenen Listener schreiben, so muss die Methoden des Listener-Interfaces implementieren. Um dem Beenden-Button aus dem Beispiel einen Sinn zu geben, muss zunächst ein entsprechender Listener geschrieben werden, der auf einen Klick auf den Button wartet. Hierfür wird ein `ActionListener` als Vorlage verwendet:

```
1  class FensterSchliesser implements ActionListener {
2    // Methode wird bei einer Aktion aufgerufen
3    public void actionPerformed(ActionEvent e) {
4      System.exit(0);
5    }
6  }
```

Nun kann die Fenster-Klasse angepasst werden, indem der Beenden-Button nicht nur erzeugt wird, sondern auch ein neuer Listener gestartet wird, welcher sich beim Button registriert:

```
1  // Button erzeugen
2  Button exitButton = new Button("Beenden");
3
4  // Listener erzeugen
5  ActionListener machZu = new FensterSchliesser();
6
7  // Listener beim Button anmelden
8  exitButton.addActionListener(machZu);
```

Wird nun auf den Beenden-Button geklickt, so meldet der Button dieses Ereignis an alle seine registrierten Listener. In diesem Fall ist das nur der `FensterSchliesser`. Dieser ruft dann seine `actionPerformed`-Methode auf, welche wiederum als Aktion das Fenster schließt.

B.4 Thread-Programmierung mit Java

Alle bisherigen Programme waren nicht nebenläufig. Das heißt zu jedem Zeitpunkt befand sich das jeweilige Programm in einem einzigen, ganz bestimmten Abschnitt, und hat Zeile für Zeile den Code ausgeführt (wobei es durch Methodeaufrufe eventuell in Methoden gesprungen und wieder zurückgekehrt ist). Bei der Programmierung mit sogenannten Threads (engl. für Faden) ist es möglich, mehrere Programmabschnitte parallel, also zur gleichen Zeit, auszuführen. Dadurch lassen sich bestimmte Probleme effizienter lösen. Außerdem

unterstützen neuere Prozessorarchitekturen direkt das parallele Ausführen mehrerer Operationen in einem Rechenzyklus, wodurch das Thema Multi-Threading in Zukunft immer wichtiger sein wird. Leider ist Thread-Programmierung ein komplexes Thema, das zu den bisherigen Fehlerquellen beim Programmieren noch weitere hinzufügt. Insbesondere dann, wenn mehrere Threads sich Ressourcen teilen, also zum Beispiel gemeinsam auf die gleichen Variablen zugreifen, muss der Programmierende besonders aufpassen.

Glücklicherweise bietet Java bereits von Haus aus einige Funktionalitäten zum Thema Mutli-Threading an und ermöglicht so direkt nebenläufiges Programmieren. Anhand eines zugegebenermaßen nicht sonderlich sinnvollen Beispiels soll nun Thread-Programmierung mit Java vorgestellt werden und auf die mögliche Gefahren aufmerksam gemacht werden.

Angenommen, es soll eine Situation nachgebaut werden, bei der zwei Personen gleichzeitig mehrmals würfeln, und eine dritte Person die Gesamtzahl aller Augen zählt. Natürlich könnte man nun das Gleichzeitige simulieren, in dem man beide Personen immer abwechselnd nacheinander würfeln lässt, doch hier soll dies nun echt zeitgleich passieren. Zunächst wird eine Klasse angelegt, die die Person repräsentiert, die die Ergebnisse aufsummiert:

```
1  public class ZaehlendePerson {
2    private int summe = 0;
3
4    public void uebergebeErgebnis(int hinzu) {
5      summe = summe + hinzu;
6    }
7  }
```

Diese Person stellt keinen Thread dar, da sie nur von den würfelnden Personen aufgerufen und somit auf Aufforderung etwas tut, und nicht ständig im Hintergrund selbst arbeitet. Dies tun jedoch die würfelnden Personen. Sie stellen ein Thread. Hierfür können sie in Java entweder von der Klasse `java.lang.Thread` erben, oder das Interface `Runnable` implementieren. Dadurch muss die Klasse nun die Methode `public void run()` implementieren. Sie enthält den Code, den der Thread ausführen soll. Durch das Erben von Thread erwirbt die Klasse außerdem verschiedene fertige Funktionen zur Verwaltung des Threads, zum Beispiel die Methode `start()` zum Starten des Threads. Somit sieht die Klasse für die würfelnden Personen wie folgt aus:

```
1  public class WuerfelndePerson extends Thread {
2    ZaehlendePerson zaehler;
3    int wieOft;
4
5    // Im Konstruktor wird festgelegt, wer das Ergebnis
6    // uebergeben bekommt und wie oft gewuerfelt wird
7    public WuerfelndePerson(ZaehlendePerson zaehler, int ↩
         wieOft) {
8      this.zaehler = zaehler;
9      this.wieOft = wieOft;
10   }
11
12   // Der Inhalt der Run-Methode wird als Thread ↩
         ausgefuehrt
13   public void run() {
14     for(int i = 0;i<wieOft;i++) {
15       // Wuerfeln von 1..6
16       int zufallszahl = 1+(int) (Math.random()*6);
17
18       // Wert an Zaehler uebergeben
```

```
19          zaehler.uebergebeErgebnis(zufallszahl);
20        }
21      }
22    }
```

Die würfelnden Personen müssen wissen, wie oft sie würfeln sollen, und an wen sie die Ergebnisse übergeben werden sollen, dies wird in zwei Attributen gespeichert und im Konstruktor festgelegt. Die Run-Methode definiert außerdem den Programmcode, welcher als Thread ausgeführt werden soll. Das Szenario ausführen könnte man nun so:

```
1  public static void main(String[] args) {
2    // Zaehlende Person anlegen
3    ZaehlendePerson zaehlendePerson = new ZaehlendePerson();
4
5    // Wuerfelende Personen anlegen
6    // Threads laufen noch nicht!
7    WuerfelndePerson persA = new
8      WuerfelndePerson(zaehlendePerson,10);
9    WuerfelndePerson persB = new
10     WuerfelndePerson(zaehlendePerson,10);
11
12   // Beiden Personen Start mitteilen (beide Threads ↩
       starten)
13   persA.start();
14   persB.start();
15 }
```

Wichtig ist hierbei, dass durch das alleinige Erzeugen der Threadobjekte die Threads noch nicht direkt starten. Hierfür gibt es eine eigene Methode `start()`, welche von der Klasse Thread geerbt wurde, die den Thread startet und somit die Methode `run()` zur Ausführung bringt. Nun können beide Personen gleichzeitig würfeln und die dritte Person zählt alle Ergebnisse. Alle? Leider nicht unbedingt, weil sich im Programm ein typischer Fehler nebenläufiger Programmierung eingeschlichen hat. Angenommen, die bisherige Gesamtzahl würde 20 betragen, und die Personen A und B würden ein weiteres Mal würfeln und zufälligerweise genau zeitgleich ihre neuen Ergebnisse übermitteln wollen. Person A hat eine 1 gewürfelt, Person B eine 5. Person B ist aus Zufall etwas schneller und ruft zuerst die Methode `zaehler.uebergebeErgebnis(5)` auf. Hier wird der alte Wert 20 um 5 erhöht und wieder zurückgespeichert. Aber noch bevor der Wert zurückgespeichert wurde, könnte auch schon Person A die Methode aufgerufen haben. Da das neue Ergebnis von B noch nicht zurückgespeichert wurde, ist auch für A der alte Wert 20, und wird um ein 1 erhöht. In der Zwischenzeit ist B mit dem Methodenaufruf fertig und der Wert ist nun 25. Person A beendet nun auch den Aufruf und speichert 20+1=21 als neuen Wert, wodurch das Ergebnis von B überschrieben wird und verloren geht. Das Problem der Methode `uebergebeErgebnis()` ist, dass sie keine atomare Operation darstellt, das heißt, dass sie eigentlich mehrere Befehle ausführt (alten Wert laden, hinzuaddieren, neuen Wert speichern) und deswegen nicht threadsicher ist, wenn mehrere Thread zeitgleich darauf zugreifen. Eine einfache Lösung in Java ist es, die Zugriffe auf diese Methode zu synchronisieren. Dadurch wird festgelgt, dass immer nur ein Thread gleichzeitig zugreifen darf. Kommen weitere Threads und wollen auf eine synchronisierte Methode zugreifen, müssen sie sich quasi in eine virtuelle Warteschlange anstellen und warten, bis sie an die Reihe kommen und den exklusiven Zugriff bekommen. Sind sie mit dem Methodenaufruf fertig, geben sie den Schutz frei und der nächste Thread ist an der Reihe. Diese Funktionalität wird in Java durch den Methoden-Modifier synchronized angewandt, folglich müsste die Methode `uebergebeErgebnis()` entsprechend angepasst werden:

```
1 public class ZaehlendePerson {
2   private int summe = 0;
3
4   // synchronized: Exklusiver Zugriff
5   public synchronized void uebergebeErgebnis(int hinzu) {
6     summe = summe + hinzu;
7   }
8 }
```

Dies war nun ein kleines Beispiel für Thread-Programmierung mit Java. Es gibt noch viele weitere interessante Techniken, aber auch einige Probleme mehr bezüglich der nebenläufigen Programmierung, auf die an dieser Stelle nicht weiter eingegangen werden kann.

C Integrierte Entwicklungsumgebungen

C.1 Notepad + Compiler

Im Prinzip benötigt man zum Programmieren nicht mehr als einen einfachen Texteditor (z.B. Windows Notepad) und einen Compiler. Für Übungsaufgaben und sehr kleine Programme mag das auch vollkommen in Ordnung sein, wird der Quellcode umfangreicher, erweist sich die Notepad/Compiler-Kombination jedoch schnell als unkomfortabel und unübersichtlich. Abhilfe schaffen hier sogenannte „Integrierte Entwicklungsumgebungen“ oder einfach IDEs (engl.: integrated development environment).

Einige geeignete Texteditoren zum Programmieren sind:

- Notepad++ (Windows, `notepad-plus.sourceforge.net`, Freie Software)
- SciTE (Windows/Linux, `www.scintilla.org/SciTE.html`, Freie Software)
- gedit (Linux, `gnome.org/projects/gedit`, Freie Software)
- gvim (Windows/Linux/Mac, `vim.org`, Freie Software)

C.2 Vorteile von IDEs

Die meisten IDEs sind wahre Alleskönner und besitzen Features wie Projekt-Manager, Debugger, Compiler und meist sehr leistungsfähige Texteditoren. Die Editoren verfügen im Normalfall über viel praktische Zusatzfunktionen, die dem Programmierer das Leben einfacher machen sollen, beispielsweise Syntaxhervorhebung, Syntaxüberprüfung, Autovervollständigung oder automatische Code-Formatierung.

Viele IDEs bieten zudem noch die Möglichkeit, die eigene Funktionalität durch Plugins zu erweitern. So ist es beispielsweise nicht ungewöhnlich, dass IDEs gleichzeitig mehrere Programmiersprachen unterstützen.

C.2.1 Projekt-Management

Da in Java jede Klasse in einer eigenen Datei gespeichert werden muss, neigen Java-Projekte dazu, aus sehr vielen Dateien zu bestehen. Aus diesem Grund verfügen Java-IDEs über Datei- und Projekt-Manager, die dabei helfen, den Überblick zu behalten und zudem sicherstellen, dass die in Java vorgeschriebenen Namenskonventionen eingehalten werden (Klassenname = Dateiname, Packagename = Verzeichnisname). In vielen IDEs gibt es zudem noch nützliche Zusatzfunktionen: Klickt man in Eclipse IDE beispielsweise bei gedrückter STRG-Taste auf den Namen einer Klasse, so springt der Texteditor automatisch an die Stelle im Code, an der die Klasse deklariert wurde. Das Selbe funktioniert natürlich auch bei Funktionen oder Variablen. Außerdem kann man sich zu jeder Klasse eine Gliederung (Outline) anzeigen lassen, also eine Auflistung aller in der Klasse deklarierten Attribute und Methoden.

Zum Projekt-Management gehört auch das sogenannte Refactoring. Darunter versteht man das Umstrukturieren von Programmteilen, also z.B. das Umbenennen von Klassen. Bei größeren Projekten stellt man nämlich ab und zu fest, dass für eine Klasse, eine Methode

oder eine Variable ein ungünstiger Namen gewählt wurde – beispielsweise weil der Name wenig aussagekräftig ist oder, im schlimmsten Fall, sogar in die Irre führt. Aussagekräftige und verständliche Namen sind für fehlerfreie und ordentliche Programmierung jedoch unabdingbar, vor allem, wenn mehrere Leute an einem Projekt arbeiten.

Das Umbenennen von Klassen oder Methoden von Hand ist allerdings recht mühsam, weil man alle Dateien durchforsten muss, um zu prüfen ob im Code die umbenannte Klasse/-Methode verwendet wird. Die Refactoring-Funktionen in IDEs nehmen einem diese Arbeit nicht nur ab, sondern stellen auch sicher, dass man alle Vorkommen erwischt hat und beim Compilieren keine Überraschungen erlebt.

C.2.2 Debugger

Eine weitere, sehr nützliche, Funktion ist der Eingangs bereits erwähnte Debugger. Der Debugger ist dazu da, dem Programmierer auf der Suche nach Bugs – also Programmierfehlern – behilflich zu sein. Er ermöglicht es, ein Programm Schritt für Schritt – also Anweisung für Anweisung – auszuführen, so dass der Programmierer prüfen kann, ob es sich tatsächlich so verhält, wie er es geplant hat. Dabei kann er zu jedem Zeitpunkt sogar nachsehen, welche Werte sich in den einzelnen Variablen des Programmes befinden.

C.2.3 Texteditor

Das Herzstück einer guten IDE ist jedoch der Texteditor. Die Standardfunktionen sind Syntaxhervorhebung, Syntaxprüfung, Code-Folding und Autovervollständigung.

Unter Syntaxhervorhebung (oder Syntaxhighlighting) wird das farbliche Hervorheben von Schlüsselwörtern und Befehlen der Programmiersprache verstanden. Diese aufgearbeitete Darstellung des Textes vereinfacht das Querlesen von Code enorm und hilft dadurch schnell die Struktur dahinter zu verstehen.

Die Syntaxprüfung ist mit einer automatischen Rechtschreibungs- und Grammatik-Prüfung in Schreibprogrammen vergleichbar. So markieren die Editoren Tippfehler und ungültige Satzkonstruktionen und bieten, wenn möglich, sogar Vorschläge zur Behebung des Problems an, sogenannte QuickFixes. Genau wie bei der Auto-Korrektur in einem Schreibprogramm, kann die Syntaxprüfung jedoch ausschließlich formale Fehler finden, die Suche nach logischen und inhaltlichen Fehlern bleibt nach wie vor Aufgabe des Programmierers.

Code-Folding ist ein weiteres nettes Feature zur Verbesserung der Übersichtlichkeit im Texteditor. Man versteht darunter das Verstecken oder „Einklappen“ einzelner Blöcke um Programmcode – ähnlich, dem Ein- und Ausklappen von Verzeichnisinhalten im Windowsexplorer. Meistens existieren zu diesem Zweck kleine Icons am linken Rand des Editorfensters.

Bei großen Programmen mit vielen Klassen verliert man relativ leicht den Überblick über die einzelnen Klassen und die zugehörigen Methoden. Deswegen verfügen viele Editoren über eine kontextsensivite Autovervollständigung, die dem Benutzer bei Bedarf eine Auflistung der verfügbaren Methoden und Attribute anzeigt, so dass dieser bequem den richtigen Eintrag aussuchen kann. Damit ist die Autovervollständigung eine der praktischsten Funktionen, denn sie erspart das Tippen langer Methodennamen und vor allem die Suche in den Klassendokumentationen nach den richtigen Methoden-Signaturen.

C.3 Nachteile von IDEs

Trotz allem gibt es unter den Programmierern auch die eher puristischen Strömungen, die – wenn überhaupt – nur ungerne mit IDEs arbeiten und die IDE-Befürworter voller Häme als Klickibunti-Programmierer bezeichnen, die ohne ihre tollen Features wie Autovervollständigung und Syntaxprüfung angeblich hilflos seien.

Tatsächlich ist nicht ganz von der Hand zu weisen, dass die Nutzung von IDEs zu fehlender Übung im Umgang mit Programmierbibliotheken führen kann, da nun mal kein Bedarf besteht, sich die genauen Namen und Signaturen der einzelnen Methoden zu merken, wenn die Autovervollständigung sie einem per Knopfdruck in einer Liste zur Auswahl anbietet. Nicht von ungefähr scheitern viele Studenten in ihrer Praktischen-Informatik-Prüfung an der Deklaration einer Main-Methode! Besonders am Anfange der Programmierer-Laufbahn ist es also sicherlich empfehlenswert, die dicken IDEs mal beiseite zu lassen und einen schlanken Syntaxhighlighter wie Notepad++ zu verwenden.

C.4 Welche IDE soll ich benutzen?

Die Frage, ob man eine IDE verwenden sollte oder nicht (und wenn ja, welche?), wird nicht selten mit einigem Eifer ausgefochten. Die Einen legen mehr Wert auf den Komfort beim Programmieren und die Anderen halten derartige Hilfsmittel für unnötigen Luxus und "coden" lieber alles von Hand – am besten per Kommandozeilen-Tool auf einem Linuxrechner. Letzten Endes bleibt die Wahl der geeigneten IDE natürlich Geschmackssache und hängt zudem vom Umfang des Projektes ab.

Eine kleine Liste der beliebtesten Java-IDEs soll an dieser Stelle natürlich nicht fehlen:

- Eclipse IDE (http://www.eclipse.org)
- NetBeans IDE (http://www.netbeans.org)
- JCreator (http://www.jcreator.com)
- JBuilder (http://www.codegear.com/products/jbuilder)
- XCode (Max OS X, `developer.apple.com`, nativen Softwareentwicklung für Mac OS X, proprietär)
- ...

Wir empfehlen dir, mehrere IDEs auszuprobieren und dir erst einmal eine eigene Meinung zu bilden.

D API - Dokumentation

Wie man vielleicht bei den ersten Programmierschritten selbst schnell bemerkt, benötigt man im Programmieralltag häufig bereits bestehende Methoden, Klassen oder Programmteile. Da es sehr viele Probleme oder Aufgaben gibt, die bei der Programmierung häufiger auftreten und gelöst werden müssen, gibt es bei Java bereits eine große Anzahl von fertigen Klassen und Funktionalitäten. Auch Programmierende oder Entwicklerteams häufen mit der Zeit viele wiederverwendbare Programmmodule und Klassen an. Da die Existenz der Klassen allein aber noch nicht ausreicht, gibt es dazu meist eine ausführliche Dokumentation. Dort werden die einzelnen Pakete, Klassen und Methoden vorgestellt und ihre Verwendung erklärt. Für angehende Java-Programmierende gehört es somit zum Grundwissen, mit dieser API-Dokumentation umgehen zu können und dort relevanten Informationen für sich finden zu können. Ebenso sollte klar sein, wie man selbst solche Kommentare schreibt und anwendet.

Denn auch bei der Entwicklung von größeren Softwareprojekten später ist es notwendig, mit der Dokumentation von fremden Klassen und Methoden umgehen zu können. Da sich solche Dokumentationsübersichten automatisch aus beliebigen Java-Code erzeugen lassen, wenn dort die Kommentare in einem bestimmten Stil formatiert sind, werden sie häufig auch zur Dokumentation von Firmensoftware, freien Projekten, etc. verwendet.

Javadoc-Informationen werden als Kommentare in Java-Klassen integriert. Zusätzlich zu den gewöhnlichen Java-Blockkommentaren werden Javadoc-Abschnitte jedoch durch einen zweiten öffnenden Stern gekennzeichnet:

```
1 /**
2  * Javadoc Kommentar
3  */
```

Mit einem solche Kommentarblock lassen sich jeweils Klassen, Methoden, Konstanten oder Variablen beschreiben. Zusätzlich zu der Möglichkeit, durch einen Text zu Beginn des Kommentars das jeweilige Konstrukt zu beschreiben, besteht außerdem die Möglichkeit, auf einige vorgefertigte Beschreibungen zurückzugreifen, welche jeweils mit einem @-Zeichen deklariert werden.

```
1  /**
2   * Eigene Klasse fuer mathematische Funktionen.
3   * @author Ada Lovelace
4   * @see java.lang.Math
5   * @version 1.0
6   */
7  public class MeineMatheKlasse {
8    /**
9     * Eulersche Zahl
10    * @see http://en.wikipedia.org/wiki/E_(mathematical_constant)
11    */
12   public static final double EULER_ZAHL = 2.718281828459045;
13
14   /**
15    * Fuehrt eine Division durch.
16    * @param a Dividend
17    * @param b Divisor
```

```
18      * @return Quotient
19      * @throws ArithmeticException Ausnahmefehler
20      *         bei Division durch Null
21      */
22    public static float dividiere(float a, float b)
23        throws ArithmeticException {
24      if(b == 0) {
25        throw new ArithmeticException("Division durch Null");
26      } else {
27        return a/b;
28      }
29    }
30  }
```

Solche Informationen werden einerseits von vielen Entwicklungsumgebungen interpretiert, sodass dem Programmierenden bei der Entwicklung zusätzliche Informationen zur Verfügung stehen. Andererseits besteht auch die Möglichkeit, aus den Javadoc-Informationen einer oder mehrerer Klassen eine Dokumentation im HTML-Format zu generieren, welche auch durch die offizielle API-Dokumentation von Java bekannt ist: `http://java.sun.com/javase/6/docs/api/`

Einleitung

Grundkonstrukt für `Klassenname.java`:

```
public class Klassenname{
    public static void main (String[] args){
        /*CODE*/
    }
}
```

Compilieren: `javac Klassenname.java`
Ausführen: `java Klassenname`

Kommentare:

```
// Kommentar bis zum Zeilenende
/* Beliebig langer Kommentar */
```

Datentypen

Variablendeklaration für primitive Datentypen: `typ name = wert;` Mögliche Typen sind **`byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`**. Beispiele: **`int`** `anzahl = 3;` **`char`** `zeichen = 'x';` **`float`** `pi = 3.14f;`
Zeichenketten: `String name = "Hans";` **Ausgabe:** `System.out.println("Hallo" + name);` (`+` fügt zusammen)
Typecasts ohne möglichen (theoretischen) Informationsverlust automatisch, sonst explizit auf diese Weise:
`(zieltyp) variable;` Beispiel: **`float`** `pi = 3.14f;` **`int`** `ganzzahl = (`**`int`**`) pi; // nun ist ganzzahl gleich 3`

Rechenoperationen (für geeigente a und b)
Grundrechenarten: `a+b, a-b, a*b, a/b`
Achtung: `4/3` ergibt `1`, `4.0/3.0` ergibt `1.333...`
Modulo Rechnung: `a%b`, Beispiel: `10%7` ist `3`
Inkrementierung, Dekrementierung: `a++, b--`

Logikoperationen für boolsche Ausdrücke a,b:
Vergleiche: `a==b, a!=b, a>=b, a<=b, a>b, a<b`
Und: `a&&b`, Oder: `a||b`, Nicht: `!a`, XOR: `a^b`
Beispiel: `(5 >= 2) && !`**`false`** ergibt **`true`**

Kontrollstrukturen

If-Abfragen mit optionalem `else if` und `else`:

```
if (boolscher Ausdruck){
    /*CODE falls Ausdruck true war*/
} else if (zweiter boolscher Ausdruck){
    /*ANDERNFALLS wenn zweiter Ausdruck true*/
} else {
    /*ANDERNFALLS*/
}
```

Switch (mit optionalem default-Teil) für Fallunterscheidungen
Achtung: **`break`** auslassen führt auch nachfolgende **`case`** aus!

```
switch(integervariable){
    case Konstante_1: /*CODE*/; break;
    case Konstante_2: /*CODE*/; break;
       ...               ...
    case Konstante_N: /*CODE*/; break;
    default: /*ANDERNFALLS*/ }
```

Schleifen

for-Schleife werden zum n-maligen Wiederholen verwendet:
`for``(start; bedingung; inkrementierung) {/*CODE*/}`
Beispiel: **`for`**`(`**`int`**` i=0; i<10; i++)`
`{System.out.println(i);}`

(do-)while-Schleifen werden stets unter einer Bedingung ausgeführt: **`while`**`(bedingung) { /*CODE*/ }`
bzw: **`do`** `{ /*CODE*/ }` **`while`**`(bedingung);`
Beispiel: **`while`**`(bier_nicht_leer()) { trinke(); }`

Arrays

Arrays sind Regale mit nummerierten Fächern für genau einen Datentyp: `typ[] name = new typ[n];` Erzeugt die Fächer `name[0]` bis `name[n-1]` Beispiel: **`int`**`[] lotto = `**`new int`**`[6];` Erzeugt `lotto[0]` bis `lotto[5]`
Mehrdim. Arrays: **`int`**`[][] matrix = `**`new int`**`[n][m];` **Länge** eines Arrays: `lotto.length` ist `6`, `matrix[].length` ist `m`

Methoden

Methoden bekommen beliebig viele Argumente übergeben und haben genau einen Rückgabewert.
Struktur: `[`**`static`**`] [rückgabetyp] name(arg1typ name1, arg2typ name2,...) {/*CODE*/}`
Static: Methoden die nicht von einem Objekt abhängen/auf ihm arbeiten. **Ohne static:** Methoden die auf Objektattributen arbeiten
Rückgabetyp: beliebiger Datentyp, **`void`** steht für "kein Rückgabetyp". Die Rückgabe erfolgt in der Methode mit **`return`** `variable`
Beispiel: **`static double`** `berechneKreis(`**`int`**` radius){ `**`return`**` (radius * radius * 3.1415); }`

Objektorientierung

Objekte sind selbstgebaute Datentypen mit Attributen und Methoden. Sie werden in Klassen beschrieben. Beispiel:

```
public class Quader{
  int laenge, hoehe, breite;                    /*Attribute*/
  public Quader(int l, int h, int b)            /*Konstruktor*/
    {laenge = l; hoehe = h; breite = b;}
  public int volumen(){ return laenge*hoehe*breite;} /*Objektmethode*/
}
```

Der **Konstruktor** wird zum Erzeugen des Objektes benutzt. Der leere Konstruktor ist implizit vorhanden, solange keine eigenen Konstruktoren (wie in diesem Fall) implementiert sind Verwendung: `Quader myQuader = `**`new`**` Quader(1, 5, 5);`
Zugriff auf Objektattribute sofern diese nicht `private` sind über `objektname.attributname` Beispiel: `myQuader.laenge`
private (statt **`public`**) verwendet man, um den direkten Lese/Schreib-Zugriff zu unterbinden, etwa damit man keine negativen Zahlen eintragen kann. Der Zugriff muss dann über geeignete Methoden realisiert werden. Auch verwendet man **`private`** für interne Methoden um die Komplexität der eigenen Klasse zu verbergen und sie somit übersichtlicher bei der Verwendung zu machen.
Methoden arbeiten auf Daten ihres Objektes. Beispiel: `myQuader.volumen()` ist `25`. Statische Methoden können dies nicht, sind dafür aber unabhängig von Objekten. Zugriff hierbei mit `Klassenname.methode()`. Ein statisches Attribut (Klassenattribut) wird nur einmal angelegt und alle Objekte einer Klasse arbeiten auf der selben Kopie. Zugriff hierbei mit: `Klassenname.attribut`
Seiteneffekte: Der Name eines Objekt ist nur ein fester Zeiger, wo das Objekt im Speicher liegt (Referenz). `Quader myQuader2 = myQuader` führt dazu, dass sowohl `myQuader` als auch `myQuader2` auf das selbe Objekt zeigen. z.B. bei Übergabeparameter an Methoden muss dies beachtet werden, da nur die Referenz kopiert wird, aber nicht das dahinter liegende Objekt.