Vorkurs Programmierung

Sebastian Mai
Kai Dannies

2012

Fachschaftsrat der Fakultät für Informatik der Otto-von-Guericke-Universität Magdeburg

Dieses Heft wurde vom Fachschaftsrat der Fakultät für Informatik der Otto-von-Guericke-Universität Magdeburg für den Vorkurs für Programmierung 2012 produziert.

Inhaltsverzeichnis

1 UNIX

Viele der Grundprinzipien von UNIX sind in heutigen Betriebssystemen noch vorhanden und mit einigen dieser Betriebssysteme wirst du im laufe eures Studiums noch zu tun haben.

1.1 Verzeichnisstruktur

In Unix wird alles, egal ob es sich um eine Netzwerkschnittstelle, die Festplatte oder eine Musikdatei handelt als Datei repräsentiert. Alle diese Dateien lassen sich im sogenannten „Verzeichnissbaum“ des Betriebssystems finden. Das wohl wichtigste Verzeichnis in UNIX ist `/` – das Wurzelverzeichnis. In diesem Verzeichnis liegen alle anderen Verzeichnisse.

An dieser Stelle lernst du die ersten beiden Kommandos für das Terminal kennen: `ls` und `cd`. Sie sind die wichtigsten Kommandos für die Navigation durch den Verzeichnissbaum von Unix. `cd` steht für „change directory“ und lässt uns das Verzeichnis wechseln. Gibt man also `cd /` ins Terminal ein, so wechselt man ins Wurzelverzeichnis. `ls` steht für „list search“, es liefert eine Auflistung der Dateien im aktuellen Ordner. Da in Unix alles eine Datei ist, sieht man auch die Unterverzeichnisse des aktuellen Verzeichnisses. Die Verzeichnisse im Wurzelverzeichnis sind bei den meisten Betriebssystemen (fast) gleich und jedes dieser Verzeichnisse erfüllt die selbe Bedeutung.

Außerdem gibt es noch Kurzschreibweisen für bestimmte Verzeichnisse. `.` bezeichnet das aktuelle Verzeichnis, `..` das Elternverzeichnis und `~` das eigene Homeverzeichnis. Dateien und Verzeichnisse die mit `.` beginnen sind versteckt. Man kann sie anzeigen, wenn man `ls` mit dem Parameter `-a` benutzt.

1.2 Arbeiten im Textmodus

Unix stammt aus Zeiten, in denen man einen Rechner mit mehreren „Terminals“ bedient hat. Ist man an so einem Terminal angemeldet, sieht man erstmal die Ausgabe des sog. „Command Line Interpreters“ – `/bin/sh`. Dieses Programm ist im wesentlichen dafür zuständig, andere Programme aufzurufen. Hinter dem Kommando ls versteckt sich zum Beispiel ein Aufruf des Programms `/bin/ls`. Außerdem gibt es noch einige zusätzliche Kommandos, wie `help` die direkt zum Command Line Interpreter gehören.

Das vermutlich wichtigste Kommando in Unix ist `man`. Mit `man` lassen sich die sogenannten Manpages zu einem Programm anzeigen. In der Manpage stehen alle wichtigen Informationen, die man zu einem Programm braucht. `man man` gibt zum Beispiel eine Hilfeseite zur Benutzung dieser Manpages aus.

Verzeichnis	Bedeutung
`/`	das Wurzelverzeichnis
`/bin`	Ausführbare Programme, z.B. die Shell /bin/sh
`/boot`	der Betriebssystemkern und was zum hochfahren benötigt wird
`/dev`	„Devices“ zum Beispiel Festplatten, USB-Geräte, der Zufallszahlengenerator ..
`/etc`	Konfigurationsdateien für das Betriebssystem
`/home`	Das Elternverzeichnis aller Nutzerverzeichnisse
`/lib`	Bibliotheken für andere Programme
`/opt`	manuell installierte Programme
`/proc`	Systemressourcen
`/root`	Das Heimatverzeichnis des „root“-Nutzers
`/sbin`	das „bin“ für Programme die nur root ausführen darf
`/tmp`	Temporäre Dateien
`/usr`	„unix system resources“ Dateien die für alle Nutzer relevant sind
`/var`	Variabler Inhalt, in `/var/log` verstecken sich diverse Log-Files

Tabelle 1.1: Slash und seine Kinder

Befehl	Bedeutung
`cat`	gibt den Inhalt einer Datei aus
`cd`	Verzeichnis wechseln
`cp`	kopiert eine Datei
`date`	zeigt das aktuelle Datum und die Uhrzeit an
`echo`	gibt die Eingabe zurück
`exit`	aktuelle Terminalsession beenden
`gedit`	ruft einen Texteditor auf
`grep`	Durchsuchen einer Datei
`java`	ein Javaprogramm ausführen
`javac`	Javaquellcode in Bytecode übersetzen
`ls`	zeigt den Inhalt des aktuellen Verzeichnisses
`man`	zeigt eine Hilfeseite an
`mkdir`	legt ein Verzeichnis an
`mv`	verschiebt eine Datei
`passwd`	eigenes Passwort ändern
`pwd`	Zeigt das Verzeichnis mit komplettem Pfad an
`rm`	Datei löschen
`ssh`	eine Shell auf einem anderen Rechner öffnen
`tar`	Dateiarchive packen und entpacken
`unzip`	zip-Archive öffnen
`wget`	herunterladen einer Datei
`yes`	wiederholt die Eingabe

Tabelle 1.2: Wichtige Befehle

1.2.1 Befehlssyntax

Um einen Befehl richtig ausführen zu können benötigt dieser eine bestimmte Form. Ein typischer Unixbefehl könnte so aussehen: `ls -lBh ~/`. ls ist der ausgeführte Befehl, `-l`, `-B` und `-h` sind Parameter für das Programm `ls` und `~/` der Pfad zum Verzeichnis, dessen Dateien `ls` auflisten soll. `~` ist eine abgekürzte Schreibweise für das eigene home-Verzeichnis. Genau so werden die Parameter und der richtige Aufruf von ls auch in der Manpage beschrieben. Mehrere Parameter können oftmals auch zusammengefasst werden, so dass `ls -l -a ..` die selbe Bedeutung hat wie `ls -la ..`

1.2.2 Befehle verknüpfen

In Unix können auch mehrere Befehle verknüpft werden. Dazu gibt es mehrere nützliche Operatoren.

- `command &` führt das Kommando im Hintergrund aus
- Mit `command1 && command2` kann man mehrere Kommandos hintereinander ausführen z.B. `mkdir foo && ls -l && cd foo/`
- Mit `command1 | command2` kann man die Ausgabe von Kommando1 als Eingabe in Kommando2 verwenden z.B. `ls -l | grep foo`

Man kann mit diesen Verknüpfungen aber auch sehr viel Schabernack treiben, man sollte davon nicht mehr benutzen als man benötigt, da man so unnötige Fehler vermeiden kann.

Auch diese Operatoren sind oft nützlich:

- `command > file` schreibt die Ausgabe der Befehls in die angegebene Datei
- `command >> file` hängt die Ausgabe hinten an die Datei an

1.2.3 Vom Textmodus zum Graphikmodus und wieder zurück

Wie man zum Beispiel in den Rechnerpools der Fakultät sieht, besitzen moderne Unixoide Betriebssysteme nicht mehr ausschließlich den Textmodus. Man kann auch wie gewohnt in Fenstern bunte Bildchen anschauen.

Im Gegensatz zu Windows, kann man hier allerdings Programme nicht wirklich von der Shell losgelöst starten, die Ein- und Ausgaben sind nur für den User nicht mehr sichtbar. Besonders deutlich wird wenn man zum Beispiel `firefox` über ein Terminal startet – ob Webbrowser, Spiel oder Programmierumgebung – jedes Programm hängt an der Shell.

Mit dem Befehl `ssh` kann man auch eine Shell auf einem anderen Unix-Rechner öffnen und sich Beispielsweise von Zuhause im Rechnerpool der FIN einloggen und dort arbeiten – sogar mit Programmen die man lokal gar nicht installiert hat. Mit `ssh -x` kann man auch Graphikmodus-Programme benutzen. Im Gegensatz

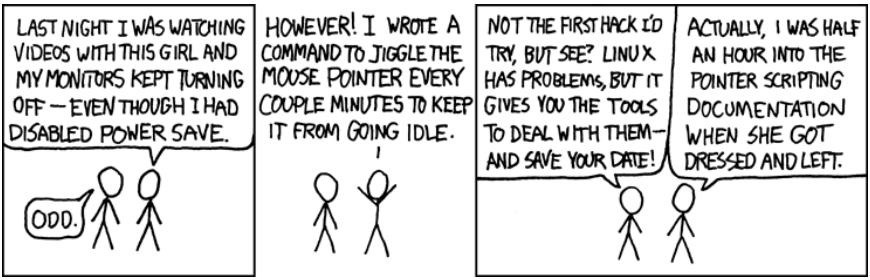

Abbildung 1.1: xkcd.com/196

zum Remotedesktop werden hier allerdings wesentlich weniger Daten übertragen und der Computer bleibt (einigermaßen) schnell.

1.3 Aufgaben

1. lade dir mit `wget` ... das Dateiarchiv für die erste Aufgabe herunter, entpacke es mit `tar -xf`.
2. sieh nach wie spät es ist mit `date` und rufe mit `man date` die Manpage zu `date` auf
3. lege in Europa einen Ordner 'Großbritanien' an und verschiebe 'york' hinein
4. finde Atlantis und lösche es
5. lege einen weiteren Ordner mit Städten für Australien an, lösche den Ordner wieder
6. trage die Einwohnerzahl von Magdeburg in der Datei europa/deutschland/-magdeburg ein
7. gib den Inhalt der Datei europa/deutschland/magdeburg mit `cat` aus
8. schau dir die Dateien in deinem Homeverzeichnis an
9. öffne eine ssh-Verbindung zu `trex.cs.uni-magdeburg.de` (anmelden kann man sich dort mit seinem FIN-Account.

2 Die Grenzen eines Computers

Der Computer kann eigentlich nur eine Sache: sehr schnell rechnen. Die Programme, die diese Fähigkeit ausnutzen lassen einen Computer oft sehr viel klüger erscheinen als er aussieht. Ein sehr berühmtes Programm, welches genau diese Tatsache demonstriert wurde in den 70er Jahren von Joseph Weizenbaum geschrieben: ELIZA. Es ist ein Programm, welches mit einem menschlichen Nutzer kommunizieren kann. Hier ein Beispiel davon:

ELIZA: Is something troubling you ?

YOU: Yes, actually I try to setup an example with you.

ELIZA: I understand.

YOU: Can you help me with this?

ELIZA: You like to think I help you – don't you ?

YOU: Yes I do.

ELIZA: You say yes you do ?

YOU: Yes I do.

ELIZA: Do you say yes you do for some special reason ?

YOU: Because I want you to help me.

ELIZA: Suppose you got I to help you soon.

YOU: I don't understand you.

ELIZA: Does that trouble you ?

YOU: Yes, a little bit

ELIZA: You seem to be quite positive.

YOU: You talk nonsense.

ELIZA: Oh, I talk nonsense ?

YOU: Yes you do.

ELIZA: We were discussing you – not me.

Unter `http://www.masswerk.at/elizabot/` kannst du es selber ausprobieren wenn du willst. Auch dieses Programm zeigt keinerlei Intelligenz, obgleich die Antworten meistens Sinn machen. Der Computer macht nichts anderes als eine Antwort, basierend auf den Informationen die er in einer Datenbank hat, auszurechnen die wahrscheinlich eine sinnvolle Antwort auf die Aussage des Nutzers ist. Damit der Computer so etwas kann, musst du als Programmierer ihm beibringen, wie er mit simplen Rechenoperationen dieses Ergebnis erreicht.

2.1 Arbeitsweise eines Rechners

Ein Rechner versteht nur einen sehr begrenzten Satz von Befehlen. Diese Befehle sind zum Beispiel Laden von oder Speichern in eine Speicheradresse, Addieren von zwei Werten oder das Vergleichen von zwei Werten. Mit Hilfe solcher sehr einfachen Anweisungen müssen alle Programme geschrieben werden. Als ob das noch nicht schwer genug wäre, versteht der Rechner nur die Eingabe von Einsen und Nullen.

1100 0000 0000 0000

1001 1010 1011 1000

1001 1010 1100 0000

0000 0000 0000 0000

0000 0000 0000 0000

1001 1000 1100 0000

1100 1111 1111 1011

Den obenstehenden Maschinencode kann ein Rechner lesen und interpretieren. Da dieser Code aber weder gut lesbar noch einfach zu schreiben ist, entwickelten Informatiker im Laufe der Jahre verschiedene Abstraktionen um das Programmieren einfacher zu machen.

Eine Abstraktionsebene ist Assembler. Ein Beispiel für ein Assemblerprogramm:

```
ORG 100h
push cs
pop ds
mov ah, 09h
mov dx, Meldung
int 21h
int 20h

Meldung: db "Hello␣World"
         db "$"
```

Das Programm macht nichts außer einem "Hello World" auf der DOS-Konsole auszugeben. Das ist zwar besser lesbar als der Maschinencode aber immer noch

nicht gut lesbar. Deshalb gibt es eine weitere Abstraktionsebene. Diese Abstraktionsebene sind die höheren Programmiersprachen wie C, C++, Java, Python und so weiter. Auf dieser Abstraktionsebene programmieren die meisten Programmierer. In diesem Vorkurs werden wir uns mit der Sprache Java beschäftigen.

3 Hello World!

3.1 Wozu „Hello World!“?

Fast alle Bücher, Tutorials, Vorkurshefte und sonstige Programmierbeispiele beginnen mit einem einfachen Programm, das nichts tut als „Hello World!“ auf der Konsole auszugeben.

Im Grund behandelt man mit dem „Hello World!“-Programm zwei Fragen:

- Was und an welchen Stellen kann ich im Quelltext ändern, um ein tolleres Programm zu erhalten?
- Wie komme ich von der Quelltext-Datei zum laufenden Programm?

3.1.1 Bestandteile des Hello-World-Progammes

In den meisten Programmiersprachen gibt es eine Art „Rahmen“ der die eigentliche Programmlogik umgibt. In Java sieht das Hello-World-Programm wie folgt aus:

```
/*
 * Vorkurs fuer Informatik
 * Hello.java
 *
 */
public class Hello {
  public static void main(String args[]) {
    System.out.println("Hello␣World!"); // gibt 'Hello World' aus.
  }
}
```

Die erste Zeile enthält unter anderem den Namen des Programms nämlich „Hello“. Die Datei mit dem Quelltext muss demzufolge `Hello.java` heißen. `public class` bedeutet, dass wir eine nach außen sichtbare Klasse definieren. Warum man das so machen muss ist ersteinmal egal.

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

Markiert den Anfang der Main-Funktion des Programmes. In der Main-Funktion stehen die Befehle, die nach Programmstart als erstes abgearbeitet werden.

```
System.out.println("Hello␣World!");
```

Das ist der Befehl der „Hello World“ ausgibt. `System.out.println` ist der Befehl der Text ausgibt, „Hello World!“ der Text den wir ausgeben wollen.
Die beiden schließenden (geschweiften) Klammern sind das Ende der Main-Funktion und das Ende des Programmes. Wenn du ein anderes Programm schreiben willst, musst du lediglich die „`System.out.println()`“-Zeile durch andere Befehle ersetzen. Damit bleiben allerdings noch einige Stellen des Programmcodes unerklärt:

// leitet einen Einzeiligen Kommentar ein. Der Rest der Zeile, beginnend mit den beiden Schrägstrichen wird beim Übersetzen des Programmes ignoriert. Mehrzeilige Kommentare beginnen mit /* und enden mit */. Der Übersicht halber führt man oftmals die Reihe mit Sternen von unten bis oben fort.

3.1.2 Das Programm ausführen

Um das Programm auszuführen, musst du zuerst den Quelltext des Programms in Bytecode umwandeln lassen und dann die Java-Virtual-Maschine (JVM) aufrufen, die den Bytecode aufsführt. Für das Übersetzen des Quelltextes in Bytecode ist der Java-Compiler zuständig Unter UNIX ruft man den Java-Compiler aus dem Terminal mit `javac Hello.java` auf. Der Compiler hat nun, sofern das Programm keine Fehler enthält eine Datei mit dem Namen „Hello.class“ erzeugt (oder eine ältere Version überschrieben) – diese Datei enthält den Bytecode. Die JVM kannst du mit dem Befehl `java` aufrufen. `java Hello` führt den Bytecode in Hello.class aus.

3.2 Aufgaben

1. Sieh dir den Quelltext in Hello.java an.
2. Kompiliere das Programm und führe es aus.
3. Ändere den ausgegebenen Text.
4. Ändere den Namen des Programms.
5. Führe das geänderte Progamm aus.

Normalerweise bearbeitet man Quelltext nicht in einem einfachen Texteditor sondern einer weit praktischeren Entwicklungsumgebung. Für Java bietet sich als Entwicklungsumgebung das Programm `eclipse` an.

- Starte Eclipse und erstelle ein neues Projekt für das Hello-World-Programm.
- Füge eine Quelltextdatei mit dem Hello-World-Programm zum Projekt hinzu und führe es aus.

4 Variablen und Datentypen

Computerprogramme weisen im Wesentlichen drei relativ unabhängige Komponenten auf: Eingabe, Verarbeitung und Ausgabe. Bis jetzt haben wir uns nur mit der Ausgabe beschäftigt. Die einfachste Form der Eingabe kennen wir auch schon: Wir setzen Werte im Quelltext. Ein Beispiel dafür ist unser „Hello World“-Proramm, indem wir festlegen das genau „Hello World“ und nicht „foobar“ ausgeben werden soll.

Um komplexere Eingaben, Verarbeitung und die Ausgabe dieser veränderlichen Daten zu ermöglichen brauchen wir eine Möglichkeit Daten abzuspeichern und zu bearbeiten, die vor dem Programmaufruf noch nicht bekannt sind. Genau dies erlaubt das Konzept der Variablen. Jede Variable ist ein Platzhalter für einen dieser unbekannten Werte.

Ersetzen wir im „Hello World“-Programm die Zeile

```
System.out.println("Hello␣World");
```

durch die folgenden drei Zeilen:

```
String text;
text = "Hello␣World";
System.out.println(text);
```

Das neue Programm tut dasselbe wie das ursprüngliche „Hello World“-Proramm, ist allerdings besser als das alte, denn die Ausgabe ist jetzt von der Eingabe (im Quelltext) getrennt.

4.1 Befehle

Wie wir sehen ist das „Hello World“-Programm länger geworden. Um Variablen zu manipulieren braucht man nämlich Befehle. Diese Befehle kann man daran erkennen, dass sie mit einem Semikolon enden. Abgearbeitet werden sie in der Reihenfolge in der man sie liest. Im Quelltext oben stehen drei Befehlsaufrufe:

Die Deklaration einer Variablen, eine Wertzuweisung und ein Funktionsaufruf. Das sind drei der wichtigsten Befehlsarten.

4.1.1 Deklaration und Definition

Um mit Variablen arbeiten zu können braucht man zwei Schritte: Deklaration und Definition.

„Sei p eine Primzahl” entspricht in etwa dem, was man dem Compiler bei der Deklaration einer Variable sagt. Um Werte digital abspeichern zu können muss man wissen, um was für Daten es sich handelt. Eine Zahl muss anders abgespeichtert werden, als eine Zeichenkette wie „Hello World“.

Java-Code	Beispiel	Bedeutung
`int`	`42`	Ganzzahlen
`double`	`3.1415`	Fließkommazahlen
`char`	`'q'`	Buchstaben
`boolean`	`false`	Wahrheitswerte
`String`	`"Foobar"`	Zeichenketten

Tabelle 4.1: Datentypen in Java

Glücklicherweise du dir als Programmierer kaum noch Gedanken darum machen, wie die Daten intern repräsentiert werden. Um auszudrücken um was für Daten es sich handelt gibt es sogenannte Datentypen. Eine kleine Auswahl der verfügbaren Typen findest du in der Tabelle 4.1.

Der Befehl für die Deklaration einer Variablen sieht so aus:

```
Typname variablenname;
```

Beachtet, das Variablennamen üblicherweise kleingeschrieben werden.

Jetzt kennt der Compiler den Typ der Variablen. Damit man auch etwas mit Variablen anfangen kann braucht die Variable noch einen Wert – die Wertzuweisung heißt Definition der Variablen.
Eine Zuweisung sieht so aus:

```
variablenname = Wert;
```

Wichtig ist dabei vor allem eins: Das „=“ (der Zuweisungsoperator) hat eine (Lese-)Richtung. Der Wert den die Variable bekommt steht immer rechts! Man kann in einer Zuweisung auch den (alten) Wert der Variable oder den einer anderen Variablen verwenden: `p = 2 * p + q;`.

Außerdem gibt es noch zwei Kurzschriebweisen, die oft verwendet werden:

```
x += y; // entspricht x = x + y;
        // Das funktioniert auch mit *=, /= und -=

x++; // entspricht x = x + 1;
x--; // x = x - 1;
```

Deklaration und Definition können auch in einer Anweisung gemacht werden:

```
String text = "Hello World";
```

Beachtet, das der Datentyp von rechter- und linker Seite der Zuweisung übereinstimmen müssen. Es kann allerdings auch sein, dass ihr das gar nicht wollt. Um den Typ einer Variablen zu ändern gibt es sogenannte Casts. Integrale Datentypen wie `int` oder `char` könnt ihr ineinander umwandeln, indem ihr den eingeklammerten Namen des Zieldatentyps vor die Variable schreibt:

```
char c = (char) intvariable;
```

Das `(char)` sorgt dann dafür, dass aus der Ganzzahl ein Buchstabe gemacht wird. Für komplexere Typänderungen gibt es Funktionen. Manchmal musst du diese Funktionen auch selbst schreiben. Ein schwieriges Beispiel für so eine Typumwandlung wäre: Mach aus einer Zahl das zugehörige Zahlwort.

4.1.2 Funktionsaufrufe

Funktionen sind vorgefertigte Unterprogramme, die einen bestimmten Zweck erfüllen. Eine Funktion kennt ihr schon: `System.out.println()`. Diese Funktion tut etwas und gibt dann die Kontrolle wieder an den nächsten Befehl ab. Es gibt aber auch Funktionen, die einen Wert berechnen.

Diesen Wert nennt man Rückgabewert. Die Funktion `Math.random()` hat als Rückgabewert einen zufälligen Wert.

Wie packt man nun Funktionen in einen Befehl? Interessiert uns der Rückgabewert der Funktion, so benutzen wir die Funktion in der rechten Seite einer Zuweisung:

```
zufallszahl = Math.random();
```

Interessiert uns der Rückgabewert nicht, so lassen wir die linke Seite der Zuweisung einfach weg:

```
Math.random();
```

Viele Funktionen, beispielsweise `System.out.println()`, sind in der Lage Daten weiterzuverarbeiten. Dafür muss der Funktion allerdings mitgeteilt werden, welche Daten gemeint sind.

Um diese Daten an die Funktion zu übergeben gibt es die Klammern nach dem Funktionsnamen. Dort kann man die entsprechenden Werte eintragen.

Sollen der Funktion mehrere Werte übergeben werden, so müssen diese durch Kommata getrennt werden – die Reihenfolge der Parameter entscheidet dabei darüber, welcher Wert wofür benutzt wird.

```
funktion(parameter1, parameter2 /* . . . */);
```

4.2 Einlesen von Werten

Um Werte einzulesen braucht man eine Variable vom Typ `Scanner`.

Mit dem Scanner kann man nicht nur von der Standarteingabe, sondern auch aus Dateien, Netzwerkverbindungen und anderen Quellen Daten einlesen. Deswegen muss man Angeben, dass der Scanner die Daten aus `System.in` beziehen soll.

Danach kann man mit `scannervariable.next()` die nächste Eingabe abholen. Außerdem muss man um den Scanner benutzen zu können mit `import java.util.Scanner;` ein weiteres Paket einbinden in dem der Scanner enthalten ist.
Im Quelltext sieht das so aus:

```
import java.io.*; // Pakete fuer ein und Ausgaben
import java.util.Scanner; // Paket fuer den Scanner
```

```
public class Hello {
        public static void main(String args[]) {
                Scanner scanner;
                // Variable: klein-s, Typ: GROSS-S.
                scanner = new Scanner(System.in);
                // Legt einen Scanner an, der System.in ueberwacht
                String name = scanner.next();
                System.out.println("Hello␣" + name + "!");
        }
}
```

Damit ist unser „Hello World“-Programm, schon richtig gut, denn es kann nun auf Nutzereingaben reagieren.

4.3 Aufgaben

1. Füge eine neue Textausgabe ein, die den Nutzer auffordert seinen Namen einzugeben, lass das Programm antworten mit „Hello *Name*“.

2. Schreibe ein Java Programm, das den Ape-Faktor einer Person ausgibt, nachdem man Größe und Armspanne eingegeben hat. (Der Ape-Faktor ist die Differenz zwischen Größe und Armspanne einer Person)

3. Schreibe ein Programm, das den ASCII-Code eines eingegeben Buchstaben ausgibt.

Zusatz: Schreibe ein Programm, das einen Buchstaben mit der Caesar-Verschlüsselung (mit Variabler Verschiebeweite) verschlüsselt.

5 Schleifen und Arrays

5.1 Schleifen

Mit dem bisher gelernten Wissen sollte folgende Aufgabe für dich leicht zu lösen sein: Schreibe ein Programm, das die Zahlen von 1 bis 12 ausgibt. Etwas schwieriger wird die Aufgabe, wenn du die Zahlen von 1 bis einer Million ausgeben sollst oder die Obergrenze von einer Nutzereingabe abhängt. An dieser Stelle kommen Schleifen ins Spiel: Sie führen einen Anweisungsblock solange aus, bis eine bestimmte Bedingung erfüllt (oder nicht mehr erfüllt) ist. Java kennt, wie viele andere Programmiersprachen auch drei verschiedene Arten von Schleifen: Die *for*-Schleife, die *while*-Schleife und die *do-while*-Schleife.

5.1.1 Die FOR-Schleife

Diese Schleife ist die, die man vermutlich am häufigsten sieht. Im Struktogramm wird sie wie in Abbildung 5.1 verwendet.

Für diese Art von Schleife braucht man eine explizite Laufvariable, die man inkrementieren kann. Im einfachsten Fall kann man dafür einen Integer benutzen. Das untenstehende Codebeispiel verdeutlicht die Benutzung einer *for*-Schleife in Java.

```
public class ForSchleife {
  public static void main(String[] args) {
    for(int i = 1; i <= 12; i=i+1){
      System.out.println(i);
    }
  }
}
```

Eine *for*-Schleife braucht drei Informationen über die Laufvariable: Ihren Initialwert, hier mit *int i = 1* geschehen, einen bool'schen Ausdruck wann die Schleife abbrechen soll, hier mit $i <= 12$ angegeben. Das bedeutet, dass die Schleife nicht mehr ausgeführt wird, sobald i diese Bedingung nicht mehr erfüllt, also 13 erreicht.

Am Ende benötigt man noch eine Aktion, die nach jedem Schleifendurchlauf gemacht werden soll, in diesem Fall wird mit *i=i+1* die Laufvariable nach jedem

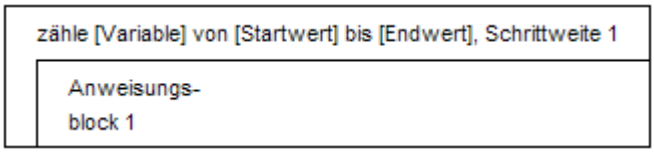

Abbildung 5.1: Die *for*-Schleife in einem Struktogramm

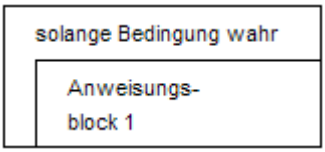

Abbildung 5.2: Eine *while*-Schleife in einem Struktogramm

Schleifendurchlauf inkrementiert.

5.1.2 Die WHILE-Schleife

Der Unterschied zu der *for*-Schleife ist im Wesentlichen, dass keine konkrete Initialisierung benötigt und auf eine Behandlung der Variable nach jedem Schleifendurchlauf verzichtet wird. Im Struktogramm hat die *while*-Schleife auch eine eigene Abbildung (5.2).

In Java definiert der folgende Code eine *while*-Schleife:

```
public class IfThenElse {
  public static void main(String[] args) {
    int i = 1;
    while(i<=12){
      System.out.println(i);
      i = i+1;
    }
  }
}
```

Der Code hat den gleichen Effekt wie der bei der im letzten Abschnitt gezeigten *for*-Schleife. Die Initialisierung findet hier jedoch vor der Schleife statt; die Inkrementierung der Variable findet in der Schleife statt.

5.1.3 Die DO-WHILE Schleife

Diese Schleife unterscheidet sich nicht wesentlich von der *while*-Schleife. Im Gegensatz zu dieser wird die Bedingung aber erst am Ende geprüft und nicht am Anfang. Auch diese Schleife hat eine Entsprechung in einem Struktogramm (Abbildung 5.3.

Der Javacode ähnelt ebenfalls sehr der *while*-Schleife:

```
public class IfThenElse {
  public static void main(String[] args) {
    int i = 1;
    do{
      System.out.println(i);
      i = i+1;
    }while(i<=12);
  }
}
```

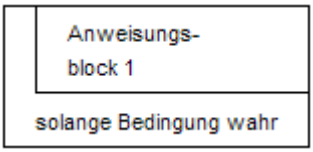

Abbildung 5.3: Die *do-while*-Schleife in einem Struktogramm

Diese Schleife läuft also, unabhängig davon ob die Bedingung am Anfang erfüllt ist oder nicht, mindestens ein mal durch.

5.1.4 Gefahren von Schleifen

Jeder Code, den du bisher geschrieben hast, läuft bei korrekter Syntax durch und das Programm wird irgendwann beendet sein. Bei Schleifen ist diese Tatsache nicht mehr garantiert. Der folgende Code hat zwar eine völlig korrekte Syntax, wird aber bei der Ausführung eine niemals terminierende Endlosschleife sein:

```
public class IfThenElse {
  public static void main(String[] args) {
    int i = 1;
    while(i<=12){
      System.out.println(i);
    }
  }
}
```

Hier wurde vergessen, die Variable, die in der Bedingung geprüft wird, zu ändern. Die Bedingung $i \leq 12$ wird damit immer wahr sein und damit wird die Schleife nie verlassen. Wenn ihr also ein Programm schreibt, es laufen lasst und es hört einfach nicht auf zu rechnen, dann ist es möglicherweise solche eine Endlosschleife die das Problem erzeugt. Es ist eure Aufgabe als Programmierer, darauf zu achten das jede Schleife terminiert.

Im letzten Kapitel hast du bereits das Schlüsselwort *break;* kennengelernt. Es wurde dazu verwendet um sofort aus einer *switch*-Anweisung herauszuspringen. Ein *break;* in einer Schleife hat die gleiche Funktion: Die Ausführung der Schleife wird sofort beendet und der Code wird nach der Schleife fortgesetzt.

```
public class IfThenElse {
  public static void main(String[] args) {
    for (int i = 1; i <= 12; i++){
      System.out.println(i);
      if (i==6){
        break;
      }
    }
    System.out.println("Nach␣der␣Schleife");
  }
}
```

Das Ergebnis des obenstehenden Codes ist, dass nur die Zahlen von eins bis sechs ausgegeben werden und die Schleife danach mittels des *break;* verlassen wird, unabhängig von dem Wahrheitswert der Abbruchbedingung der Schleife.

Für Schleifen gibt es noch ein zweites derartiges Schlüsselwort: *continue;*. Im Gegensatz zu *break;* bricht es nicht die komplette Schleife ab sondern lediglich den aktuellen Schleifendurchlauf. Der untenstehende Code wird also alle Zahlen zwischen eins und zwölf ausgeben außer der 6:

```
public class IfThenElse {
  public static void main(String[] args) {
    for (int i = 1; i <= 12; i++){
      if (i==6){
        continue;
      }
      System.out.println(i);
    }
    System.out.println("Nach␣der␣Schleife");
  }
}
```

5.1.5 Aufgaben

1. Implementiere eine Schleife, die eine Integer-Zahl von der Konsole liest und alle natürlichen Zahlen kleiner oder gleich der gelesenen Zahl ausgibt. Implementiere das Programm mit Hilfe einer
   - *for*-Schleife
   - *while*-Schleife
   - *do-while*-Schleife
2. Verändere das Programm mit der *for*-Schleife, sodass es nur noch die geraden Zahlen ausgibt. Verändere dabei
   - nur den Anweisungsblock in der Schleife.
   - nur den Schleifenkopf.
3. Schreibe ein Programm das von der Konsole zwei Integer n und m liest. Gib alle Zahlen zwischen 1 und $n \cdot m$ in n Spalten und m Zeilen ausgibt. (Hinweis: Es ist möglich eine Schleife in einer Schleife zu definieren)

4. Schreibe ein Programm das für eine eingegebene Zahl prüft, ob sie eine Primzahl ist. (Hinweis: Das Sieb des Eratosthenes bietet einen möglichen Algorithmus)

5. Schreibe ein Programm welches alle Primzahlen bis zu einer von dem Nutzer eingegebenen Obergrenze ausgibt.

5.2 Arrays

Du weißt bereits was Variablen und Datentypen sind. Wenn man beispielsweise die Höchsttemperatur eines Tages speichern möchte, kann man sich dafür ein *double* deklarieren, in dem man diesen Wert speichern kann. Jetzt könnte man zum Beispiel die Aufgabe bekommen, die Höchsttemperaturen von jedem Tag der Woche zu speichern. Man kann dafür natürlich 7 Variablen anlegen. Wenn man den Zeitraum für das Speichern der Temperaturen aber ausweitet, zum Beispiel auf ein Jahr, dann kann auch das anstrengend werden.

Arrays sind ein Weg um die Speicherung solcher gleichartigen Datenmengen einfach umzusetzen. Man kann sich ein Array wie eine lange Reihe von Schließfächern vorstellen: in jedem Schließfach gibt es einen Inhalt, und mit der Nummer des Schließfachs kann man auf den Inhalt zugreifen. Die Verwendung von Arrays in Java zeigt das folgende Codebeispiel:

```
public class IfThenElse {
  public static void main(String[] args) {
    double[] temperature = new double[365];
    temperature[42] = 13.37;
    System.out.println(temperature[42]);
    System.out.println(temperature[23]);
  }
}
```

In der ersten Zeile wird der Container deklariert: Es wird Platz für 365 *double*-Werte reserviert und jeder der Werte wird mit einem Standardwert, in diesem Fall 0.0, initialisiert. In der zweiten Zeile befülle ich den 42. Platz des Containers mit einem Wert: 13.37, der in der dritten Zeile wieder ausgegeben wird. Die Ausgabe in der vierten Zeile wird 0.0 ausgeben, da kein anderer Wert diesem Container zugewiesen wurde.

Bemerkung: Die Initialisierung mit einem Standardwert ist eine Eigenheit von Java. Viele andere Programmiersprachen, zum Beispiel C++, würden an dieser Stelle einen (mehr oder weniger) zufälligen Wert zurückgeben.

Ähnlich wie das Schachteln von Schleifen, können auch Arrays "geschachtelt" werden. Wenn man sich ein eindimensionales Array als Reihe eines Schachbretts vorstellt, dann ist das komplette Brett ein zweidimensionales Array. Ein dreidimensionales Array kann man sich als viele Schachbretter übereinander vorstellen. Die Vorstellung von Dimensionen größer als drei überlasse ich an der Stelle lieber

den Mathematikern. Die Deklaration eines mehrdimensionalen Arrays kann wie folgt aussehen:

```
public class IfThenElse {
  public static void main(String[] args) {
    String[][] chessBoard = new String[8][8];
  }
}
```

Natürlich muss man auch bei Arrays auf Dinge achten. So kann man in Java bei einem Array, das mit n Feldern initialisiert wurde auf die Felder 0 bis $n-1$ zugreifen. Ein Zugriff auf einen Wert außerhalb dieses Bereichs führt zu einem Absturz des geschriebenen Programms. Ein weiterer Nachteil von Arrays ist es, dass man eine Größe beim Programm spezifizieren muss. Datenstrukturen, die diesen Nachteil umgehen wirst du im Laufe des ersten Semesters kennenlernen.

Eine häufig genutzte Funktion von Arrays ist das bestimmen der Größe um zum Beispiel die Anzahl der Schleifendurchläufe zu bestimmen:

```
public class IfThenElse {
  private const int ARRAY_SIZE = 8;
  public static void main(String[] args) {
    int[] example = new int[ARRAY_SIZE];
    for (int i = 0; i < example.length; i++{
      //do something
    }
  }
}
```

Insbesondere beim Einlesen der Größe des Arrays, wenn man also die Größe des Arrays nicht von vornherein kennt, ist diese Funktion sehr hilfreich.

5.2.1 Aufgaben

1. Schreib ein Programm, das für ein Array von Integern oder Floats die Summe aller ihrer Elemente ausgibt.

2. Schreib ein Programm, welches zwei gleich große Arrays von Integern oder Floats entgegennimmt und die Elementweise Summe in ein neues Array schreibt.

3. Schreib ein Programm, welches die Größe eines zu erstellenden Arrays vom Nutzer einliest. Fülle dieses Array der Größe n mit den ersten n Primzahlen.

4. Spieglein, Spieglein in meinem Programm, spiegle ein Zweidimensionales Array. (Es sollen sich also die Werte an anderen Stellen wiederfinden.

5. Schreib ein Programm, in dem zu jeder Person mehrere schlechte Wortwitze abgespeichert werden können. Was wäre eine Wortspielkasse, ohne ein tolles Javaprogamm?

6 Modellierung

Modellierung ist bei den meisten Informatikern kein sehr beliebtes Thema. Modellieren bedeutet sich zunächst hinzusetzen und das Programm was man schreiben möchte zu planen. Ich habe in den vergangenen Kapiteln bereits eine Notation dazu eingeführt in Grenzen eingeführt: Struktogramme.

Bislang habt ihr nur Programme geschrieben die sehr leicht zu überblicken und sehr schnell zu schreiben waren. Sobald ihr anfangt größere Programme zu schreiben ist das einfach drauflos schreiben schwieriger und führt zu meist zu schlechteren Ergebnissen wenn ihr vorher nicht euer Programm modelliert. Stell dir vor du sollst eine aus Informatikersicht verhältnismäßig einfache Anwendung wie Minesweeper implementieren. Dann brauchst du eine Oberfläche mit Bedienelementen, du brauchst einen Algorithmus der bei einem Klick ausrechnet welche Felder aufzudecken sind, du brauchst um den Nutzer zufrieden zu machen auch Highscores oder ähnliche Zusätze. Um das Programm effizient und möglicherweise im Team zu schreiben ist eine vorherige Planung oder Modellierung unerlässlich.

Ich möchte hier eine Möglichkeit der Modellierung einführen, die insbesondere für einfache Programme geeignet ist: Struktogramme.

6.1 Struktogramme

Struktogramme sind nicht gut geeignet um große Softwarelösungen zu schreiben, weil sie dazu zu einfach sind. Sie sind aber sehr gut geeignet um einfache Teilprogramme oder Algorithmen zu modellieren. Wesentlich komplexere Arten der Modellierung wirst du im Laufe deines Studiums kennenlernen.

Es gibt nur sehr weniger Elemente, was es einfach macht sie zu lernen. Die Elemente, die standardisiert in der DIN 66261 definiert sind:

- Sequenz
- einfache, zweifache und mehrfache Verzweigung
- *for, while, do-while*-Schleife
- Funktionsaufruf

Für die Schleifen und Verzweigungen kennst du die Darstellungen bereits. Auch für die Sequenz (Bild 6.1) und den Funktionsaufruf (Bild 6.2) sind die Symbole einfach. Man muss diese Symbole jetzt nur kombinieren um sein Programm zu modellieren. Ein Beispiel seht ihr in Bild 6.3.

Dort versteht man, ohne Ahnung von einer konkreten Programmiersprache zu haben, was passiert zumindest syntaktisch passiert. Es bleibt trotzdem schwierig zu verstehen, was dort auf der semantischen Ebene passiert aber dieses Problem wird eine Modellierungsmethode auch nicht lösen können.

Abbildung 6.1: Die Sequenz im Struktogramm

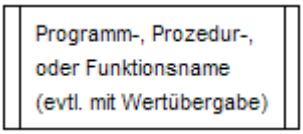

Abbildung 6.2: Der Funktionsaufruf im Struktogramm

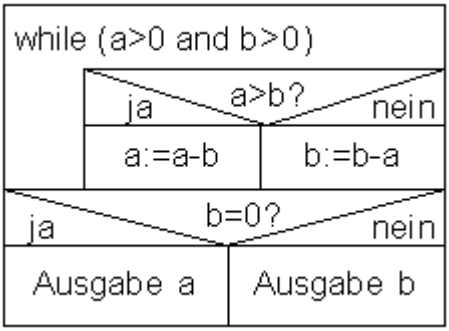

Abbildung 6.3: Ein Beispielalgorithmus in Pseudocode

6.2 Aufgaben

- Finde heraus welches Problem der gegebene Pseudocode löst.
- Schreibe einen Pseudocode, welcher die Weckzeit deines Weckers bestimmt.

7 Funktionen

Bislang sahen alle deine Programme folgendermaßen aus:

```
public class Funktionen {
  public static void main(String[] args) {
    //do something
  }
}
```

Auch hier hast du schon eine Funktion verwendet, möglicherweise ohne es zu wissen: *public static void main(String[] args)* definiert eine Funktion. Diese spezielle Funktion ist eine besondere: Sie deklariert für den Rechner den Einstiegspunkt in unser geschriebenes Programm. Euer Code wird beginnend in dieser Funktion ausgeführt. Ihr habt auch die Möglichkeit weitere Funktionen zu schreiben. Der Syntax dafür ist der folgende:

```
public class Funktionen {
  public static void myFunction() {
    //do something
  }
}
```

Das erste Wort, *public* ist ein Schlüsselwort, welches angibt wer diese Funktion benutzen kann. Ich möchte an dieser Stelle nicht in die Tiefe gehen, aber euch die möglichen Schlüsselwörter nicht vorenthalten:

- *public*
- *private*
- *protected*

Das zweite Schlüsselwort, *static* sei für den Moment verpflichtend. Wenn ihr euch mit Objektorientierung beschäftigt, kann dieses Schlüsselwort ausgelassen werden. Das liegt aber außerhalb des Rahmen dieses Vorkurses.

Das dritte Schlüsselwort, *void*, bezeichnet den Rückgabetyp einer Funktion. *void* bedeutet dabei, dass diese Funktion keinen Rückgabetyp hat. Jede beliebige Bezeichnung eines Datentyps, zum Beispiel *int, bool, String, ...* kann an dieser Stelle stehen. Wenn dort ein Datentyp stehen, muss diese Funktion einen Wert dieses Datentyps zurückgeben:

```
public class Funktionen {
  public static int myFunction() {
    int someInt = 0;
    //do something
    return someInt;
  }
}
```

Das Schlüsselwort für das Zurückgeben eines Wertes ist *return*. Wenn die Funktion *return* erreicht hat, wird kein weiterer Code in der Funktion ausgeführt sondern sofort zurück in aufrufende Funktion gesprungen. In diesem Sinne funktioniert *return* wie ein *break*.

Diese Funktionen können nun genauso mit Code befüllt werden wie ihr bisher die *main*-Funktion getan habt. Ein Beispiel ist das Folgende:

```
public class Funktionen{
  //hier startet das Programm
  public static void main(String[] args){
    int number = readNumberFromConsole();
    bool isPrime = checkPrime(number);
    print(number, isPrime);
  }

  public static int readNumberFromConsole(){
    int number;
    //you already should know the code for this
    return number;
  }

  public static bool checkPrime(int number){
    bool result;
    //you shall figure out this code for yourself
    return result;
  }

  public static void print(int number, bool isPrime){
    if(isPrime)
      System.out.println(number + "␣ist␣eine␣Primzahl");
    else
      System.out.println(number + "␣ist␣keine␣Primzahl");
  }
}
```

Das obenstehende Beispiel demonstriert die Verwendung von Funktionen. Manche Funktionen können Parameter besitzen, also Werte von denen diese Funktion abhängt. Bei der Definition einer Funktion werden diese benötigten Werte in den Klammern definiert. Unsere Funktion *print* hängt beispielsweise von 2 Parametern ab: Einem *int* und einem *bool*.

Eine Funktion wird immer mit ihrem Namen und einer öffnenden und schließenden Klammer hinter dem Namen aufgerufen. In diese Klammern müssen konkrete Werte für alle Parameter, die diese Funktion benötigt übergeben werden. Wenn die Funktion keine Parameter besitzt bleiben diese Klammern leer.

Wenn Funktionen Werte zurückgeben, kann man diese Werte Variablen zuweisen oder andere Funktionen damit "füttern", wie im folgenden Beispiel:

```
public class Functions{
  //hier startet das Programm
  public static void main(String[] args){
    int number = readNumberFromConsole();
    print(number, checkPrime(number));
  }

  public static int readNumberFromConsole(){
    int number;
    //you already should know the code for this
    return number;
  }

  public static bool checkPrime(int number){
    bool result;
    //you shall figure out this code for yourself
    return result;
  }

  public static void print(int number, bool isPrime){
    if(isPrime)
      System.out.println(number + "␣ist␣eine␣Primzahl");
    else
      System.out.println(number + "␣ist␣keine␣Primzahl");
  }
}
```

7.1 Exkurs: Rekursion

Ein sehr beliebtes Zitat für Rekursion ist: "Um Rekursion zu verstehen, muss man Rekursion verstanden haben". Das trifft das Problem an der Rekursion ziemlich gut. Es ist eigentlich keine schwierige Sache, aber das menschliche Gehirn ist diese Art von denken nicht gewohnt.

Rekursion bedeutet, dass sich eine Funktion immer wieder selbst aufruft. Um eine sich wiederholende Struktur auszunutzen. Ein einfaches Beispiel dafür ist die Fakultät einer Zahl, wo bereits die Definition rekursiv ist:

$$1! = 1 \; n! = n \cdot (n-1)!$$

Wie du siehst, wird die Fakultät benutzt um die Fakultät zu definieren. Die Rekursion hat außerdem noch eine Abbruchbedingung, in diesem Fall wenn $n = 1$. Wenn ihr eine Rekursion programmiert, müsst ihr auch eine Abbruchbedingung programmieren. Der untenstehende Code ist ein Beispiel für die Fakultät die mit Hilfe von Rekursion gelöst wird.

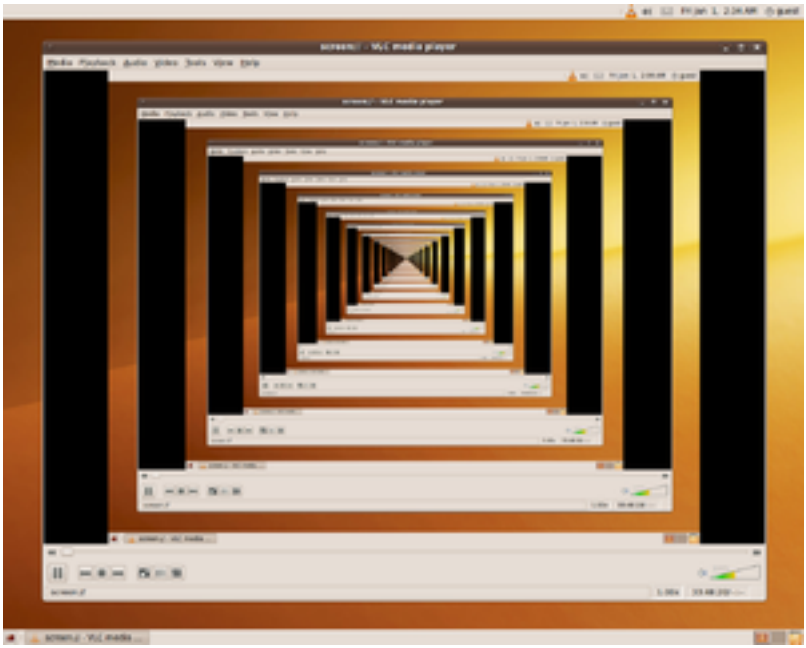

Abbildung 7.1: Grafische Rekursion

```
public class Rekursion{
  public static void main(String[] args){
    long number = 12;
    long result = factorial(number);
    System.out.println(result);
  }

  public static long factorial(long number){
    //Abbruchbedingung
    if(number <= 1){
      return 1;
    }
    //rekursiver Aufruf
    return number*factorial(number-1);
  }
}
```

In der Funktion *factorial* findet ihr beide Elemente, die jede Rekursion enthalten muss. Zum Einen die Abbruchbedingung bei der geprüft wird, ob die Zahl kleiner als 2 ist. Wenn das so ist wird eine Konstante zurückgegeben. Zum Anderen findet ihr den rekursiven Aufruf, bei dem in der *return*-Zeile wieder die Funktion aufgerufen wird.

7.2 Aufgaben

1. Du kennst sicherlich den Bereich in diversen Supermärkten in denen die Einkaufswagen stehen. Mit dem Programm ShoppingCart.java wollen wir diesen Bereich jetzt nachbauen.
   - Mach aus der Ausgabe eine Funktion mit einem aussagekräftigen Namen.
   - Schreibe eine Funktion, die einen Einkaufswagen aus einer zufällig gewählten Reihe wegnimmt.
   - Schreibe eine Funktion, die einen Einkaufswagen in eine zufällge Reihe einfügt.
   - Schreibe eine Funktion, die den Einkaufswagen in der längsten Reihe wieder einreiht.
   - Simuliere den Supermarkt in mehreren Schritten. Wenn du es dir leicht machen willst, nimmt der Kunde den Einkaufswagen in einer Phase mit und bringt ihn sofort zurück. Die Fortgeschrittenen Java-Programmierer unter euch können mehrere Kunden im Markt simulieren.
2. Schreibe ein Programm, dass die n-te Fibonaccizahl ausrechnet.
3. Schreibe ein Programm, dass die Ackermannfunktion berechnet.
4. Schreibe ein Programm, das die Fakultät berechnet ohne Rekursion zu benutzen.

8 Guter Programmierstil

Oder: Wie schreibe ich meinen Code, sodass ihn auch andere lesen können?

"Guter" Code ist unter Programmierern immer ein beliebtes Streitthema, weil jeder dort seine eigene Definition von gutem Code hat. Es gibt also keine allgemeingültigen Regeln, die man bei Code zu beachten hat. Trotzdem gibt es einige Dinge, die man auf jeden Fall machen sollte. Um die Wichtigkeit von sauberem Code zu erkennen, möchte ich zunächst mal die Unterschiede in der Lesbarkeit demonstrieren:

```
public class A{public static void main(String[] args)
{int a=2;int b=3;int c=1;for(int i=0;i<b;i++){c=c*a;}
System.out.println(c);}}
```

Obenstehend ist ein Beispiel ohne jede Formatierung. Der gesamte Code steht einfach in einer Zeile (für die Lesbarkeit habe ich trotzdem Zeilenumbrüche hinzugefügt). Dieser Code würde so kompiliert und mit dem korrekten Ergebnis ausgeführt werden. Probiert einfach mal herauszubekommen, was dieser Code macht.

```
public class A{
public static void main(String[] args){
int a=2;
int b=3;
int c=1;
for(int i=0;i<b;i++){
c=c*a;
}
System.out.println(c);
}
}
```

Dieser Code demonstriert eine der ersten Grundsätze: Jede Anweisung kommt in eine einzelne Zeile. Es erhöht die Übersicht enorm und der Code ist schon viel einfacher zu lesen. Aber es geht natürlich noch besser.

```
public class A{
  public static void main(String[] args){
    int a=2;
    int b=3;
    int c=1;
    for(int i=0;i<b;i++){
      c=c*a;
    }
    System.out.println(c);
  }
}
```

Dieser Code enthält Einrückungen. Diese Einrückungen demonstrieren einen zusammenhängenden Codeblock. Insbesondere bei mehreren und komplexeren Funktionen hilft dies enorm um auf einen Blick mitzubekommen, welcher Code in welche Funktion / Schleife / Verzweigung gehört.

```
public class Potenzieren{
  public static void main(String[] args){
    int basis=2;
    int exponent=3;
    int ergebnis=1;
    for(int i=0;i<exponent;i++){
      ergebnis=ergebnis*basis;
    }
    System.out.println(ergebnis);
  }
}
```

Im obenstehenden Code habe ich (fast) alle Namen die ich ersetzen kann ersetzt. Jede Variable, jeder Funktionsname und jeder Klassenname sollte widerspiegeln wozu die jeweilige Entität da ist. Eine Ausnahme ist auch im obenstehenden Beispiel zu erkennen: Zählvariablen in Schleifen werden in der Regel mit *i, j, k* usw. benannt. Bei diesem Code kann man sehr leicht erkennen, was gemacht wurde weil man drei einfach Regeln beachtet hat:

- jede Anweisung kommt in eine eigene Zeile
- jeder Inhalt zwischen zwei geschweiften Klammern wird eingerückt
- jeder Name den ihr wählen könnt soll einen Aussagekräftigen Namen bekommen

8.1 Kommentierung

Kommentierung ist den meisten Programmierern ebenfalls ein lästiges Thema. Aber auch hier gilt wieder: Gute Kommentierung macht den Code lesbarer. Die Kommentierung von Code findet auf drei Ebenen statt:

- auf Klassenebene
- auf Funktionsebene
- auf Anweisungsebene im Code

Auf allen drei Ebenen erfüllen die Kommentare unterschiedliche Aufgaben. Auf Klassenebene wird die Frage "Was macht diese Klasse?" beantwortet. Auf Funktionsebene wird die Frage "Wie wird die Aufgabe in der Funktion gelöst?" beantwortet. Auf Anweisungsebene erklärt ein Kommentar, warum genau diese Anweisung durchgeführt wird. Als Faustregel gilt: wenn ihr vor eine Funktion oder eine Anweisung schreiben müsst, was dort passiert ist, dann habt ihr die Funktion oder die verwendeten Variablen nicht gut benannt. Ein Beispiel für Kommentierung:

```
/*
 * Diese Klasse potenziert zwei Integer und gibt das
 * Ergebnis auf der Konsole aus.
 */
public class Potenzieren{

  /*
   * Die Funktion fuehrt das Potenzieren aus, indem sie
   * die Basis mehrfach in einer Schleife multipliziert
   */
  public static void main(String[] args){
    int basis=2;
    int exponent=3;
    //Das Ergebnis wird mit 1 initialisiert, weil x^0=1
    int ergebnis=1;
    for(int i=0;i<exponent;i++){
      ergebnis=ergebnis*basis;
    }
    System.out.println(ergebnis);
  }
}
```

Im obenstehenden Code siehst du zwei verschiedene Arten der Kommentierung. Ein einzeiliger Kommentar wird mit // eingeleitet. Dieser Kommentar geht exakt bis zum nächsten Zeilenumbruch. Ein mehrzeiliger Kommentar wird von */*Kommentar*/* umschlossen. Insbesondere in größeren Projekten im Laufe deines Studiums sollte jede Klasse und jede Funktion einen Kommentar besitzen. Die Anweisungen bekommen nur dann einen Kommentar wenn die Frage nach dem warum beantwortet werden muss.

Insbesondere für die Arbeit mit Kommentaren gilt: Wenn du 4 Programmierer nach dem richtigen Kommentieren fragst wirst du 5 verschiedene Antworten bekommen. Deshalb ist die oben stehende Variante als Vorschlag zu betrachten. Für welchen Stil du dich auch entscheidest: Guter Code braucht Kommentare.

9 Debugging

Wenn ihr Code schreibt, wird es mit einer hohen Wahrscheinlichkeit dazu kommen, dass dieser Code Fehler enthält. Bevor ihr das Programm bei eurem Auftraggeber - während des Studiums euer Tutor oder Betreuer - abgebt, müssen diese Fehler behoben sein. Es gibt zwei Arten von Fehlern: Syntaktische und semantische Fehler. Bei syntaktischen Fehler wird euch der Compiler sagen, dass ihr etwas falsch gemacht habt. Typische Fehler dabei sind, dass man sich bei einer Variable vertippt habt, dass ihr ein Semikolon vergessen habt oder ähnliches. Um diese Fehler zu beheben, müsst ihr nur lernen die Fehlermeldungen des Compilers zu interpretieren. Diese Aufgabe ist mit etwas Übung leicht zu bewältigen.

Schwieriger wird die Aufgabe bei semantischen Fehlern. Semantische Fehler treten erst zur Laufzeit des Programms auf. Entweder werdet ihr dann zur Laufzeit einen Programmabsturz erleiden oder das Programm liefert einfach nicht das Ergebnis was ihr erwartet. Bei einem Programmabsturz wird es wieder eine Fehlermeldung geben, die ihr interpretieren könnt: Es wird eine Exception geworfen. Typische Fehler die zur Laufzeit auftreten sind folgende:

- Zugriff auf eine nicht vorhandene Arrayposition
- Division durch 0
- Zugriff auf ein nicht initialisiertes Objekt

Auch diese Fehler könnt ihr mit Hilfe der Fehlernachrichten sehr schnell lokalisieren. Sie zu lösen ist nur eventuell ein wenig schwieriger, weil ihr nicht sofort wisst, wie die Werte zur Laufzeit zu Stande gekommen sind. Das folgende Codebeispiel verdeutlicht das Problem:

```
public class Div{
  public static void main(String[] args){
    BufferedReader br = new BufferedReader
              (new InputStreamReader(System.in));
    int number1 = Integer.parseInt(br.readLine());
    int number2 = Integer.parseInt(br.readLine());
    int result = number1 / number2;
  }
}
```

Ihr lest zwei Werte von der Konsole. Wenn der Nutzer sich dann einfallen lässt, für den zweiten Wert eine 0 einzugeben werdet ihr eine Exception bekommen, da die Division durch im Rechner 0 nicht möglich ist.

Die einfachste Art diese Art von Fehler zu beheben ist, die Werte der Variablen an dieser Stelle zu überprüfen. Eine sehr einfache Art ist sie einfach aus der Konsole auszugeben bevor die kritische Zeile ausgeführt wird:

```
public class Div{
  public static void main(String[] args){
    BufferedReader br = new BufferedReader
              (new InputStreamReader(System.in));
    int number1 = Integer.parseInt(br.readLine());
    int number2 = Integer.parseInt(br.readLine());
    System.out.println(number1 + "␣" + number2);
    int result = number1 / number2;
  }
}
```

Dann seht ihr, welche Werte den Fehler auslösen und könnt ihn behandeln. Eine zweite Möglichkeit, die von den Compilern der meisten Programmiersprachen bereitgestellt wird sind Breakpoints. Wenn ihr Eclipse benutzt, könnt ihr einfach mit einem Rechtsklick auf die auf den Editorrand neben der Zeile einen Breakpoint setzen. Wenn ihr jetzt unter *Run→Debug* das Programm laufen lasst wird das Programm an jedem Breakpunkt anhalten und euch die Werte aller gesetzten Variablen an der Stelle ausgeben.

Mit diesen Informationen müsst ihr nun das tun was ein Computer nicht kann: Nachdenken. Ihr müsst herausfinden wie der Code verändert werden muss, damit er fehlerfrei läuft und die gewünschten Ergebnisse produziert. Damit schließt sich der Kreis zum ersten Kapitel wieder: Der Rechner kann für euch sehr schnell rechnen aber er kann euch nicht das Denken ersparen.