



# AIIT-Matura

ahme17

April 2022

# Inhaltsverzeichnis

<b>1 Grundlagen der Informatik</b>	<b>6</b>
1.1 Zahlensysteme und Datentypen . . . . .	6
1.1.1 Meistverwendete Stellenwertsysteme . . . . .	6
1.1.2 Primitive Datentypen . . . . .	9
1.2 Sequenz, Alternation und Iteration . . . . .	11
1.2.1 Sequenz . . . . .	11
1.2.2 Alternation . . . . .	12
1.2.3 Iteration . . . . .	13
1.3 Grundlegende Begriffe . . . . .	14
1.3.1 Personal Computer (PC) . . . . .	14
1.3.2 Betriebssystem (OS) . . . . .	14
1.3.3 Disc Operating System (DOS) . . . . .	14
1.3.4 Central Processing Unit (CPU) . . . . .	15
1.3.5 Random Access Memory (RAM) . . . . .	15
1.3.6 Read Only Memory (ROM) . . . . .	16
1.3.7 Hard Disc Drive (HDD) . . . . .	16
1.3.8 Solid State Drive (SSD) . . . . .	16
1.3.9 Basic Input/Output System (BIOS) . . . . .	16
1.3.10 New Technology File System (NTFS) . . . . .	16
1.3.11 Extended File System Version 4 (ext4) . . . . .	16
1.3.12 Local Area Network (LAN) . . . . .	16
1.3.13 Wide Area Network (WAN) . . . . .	17
1.4 Zufallszahlen . . . . .	18

1.4.1	In C . . . . .	18
1.4.2	In Java . . . . .	18
<b>2</b>	<b>Strukturierte Programmierung</b>	<b>20</b>
2.1	Aufbau C-Programm . . . . .	20
2.1.1	Aufbau . . . . .	20
2.1.2	Ablauf . . . . .	20
2.2	Eingabe/Ausgabe, EVA Prinzip . . . . .	21
2.2.1	EVA - Prinzip . . . . .	21
2.2.2	printf . . . . .	21
2.2.3	fgets . . . . .	21
2.2.4	sscanf . . . . .	21
2.2.5	Beispiel . . . . .	22
2.3	Felder/Zeiger . . . . .	23
2.3.1	Zeiger . . . . .	23
2.3.2	Felder . . . . .	24
2.4	User-Interface / Menü . . . . .	25
2.5	Funktionen . . . . .	26
2.5.1	Beispiel . . . . .	26
2.5.2	Funktions- und Variablenamen Kriterien . . . . .	26
2.5.3	Funktions- und Variablenamen Best Practice . . . . .	26
2.6	Verzweigungen . . . . .	27
2.6.1	Arten . . . . .	27
2.6.2	if (Schleife)- else . . . . .	27
2.6.3	switch - case . . . . .	27
2.7	Schleifen . . . . .	28
2.7.1	verschiedene Schleifen . . . . .	28
2.7.2	do - while . . . . .	28
2.7.3	while . . . . .	28
2.7.4	for . . . . .	28

2.8	Zufallszahlen . . . . .	29
2.9	Beispiel mit Funktionen, Feld befüllen, Menü, Ausgabefunktion und Struktogrammen . . .	29
2.9.1	Quellcode . . . . .	29
2.9.2	Struktogramme . . . . .	31
<b>3</b>	<b>Objektorientierte Programmierung</b>	<b>33</b>
3.1	Programmiersprache Java (nicht unbedingt zu können) . . . . .	33
3.2	Klasse und Objekt . . . . .	34
3.2.1	Attribute (Eigenschaften) . . . . .	35
3.2.2	Methoden . . . . .	35
3.2.3	Erzeugen eines Objekts . . . . .	36
3.3	Datenkapselung . . . . .	37
3.4	Zugriffsschutz . . . . .	38
3.4.1	Geheimnisprinzip . . . . .	38
3.4.2	Zugriffsmodifikatoren . . . . .	38
3.4.3	Getter- und Setter-Methoden . . . . .	39
3.5	Klassendiagramm . . . . .	40
3.5.1	Objektorientiertes Design mit dem UML Klassendiagramm . . . . .	40
3.6	Konstruktoren . . . . .	43
3.7	Design Patterns . . . . .	44
3.7.1	Immutable . . . . .	44
3.7.2	Rechnerklasse . . . . .	46
3.7.3	Datenhaltungsklassen . . . . .	47
3.8	Vererbung . . . . .	48
3.8.1	Interfaces . . . . .	49
3.8.2	Abstrakte und konkrete Klassen . . . . .	50
3.9	Collections . . . . .	51
3.9.1	ArrayList . . . . .	52
3.9.2	LinkedList . . . . .	53
<b>4</b>	<b>Visualisierungen / Graphische Oberflächen</b>	<b>54</b>

4.1	Rahmenfenster / JFrame . . . . .	54
4.2	Dialogfenster / JDialog . . . . .	54
4.3	Layouts . . . . .	54
4.3.1	Übersicht . . . . .	54
4.3.2	Schachteln von Layouts . . . . .	55
4.4	Java-Swing . . . . .	55
4.4.1	Übersicht . . . . .	55
4.4.2	Wissenwertes . . . . .	56
4.5	Listenmodell . . . . .	57
4.6	Tabellenmodell . . . . .	58
4.7	Handler-Methoden . . . . .	59
4.8	Swing-Worker . . . . .	59
4.9	Beispiel . . . . .	60
<b>5</b>	<b>Networking</b>	<b>61</b>
5.1	Ethernet . . . . .	61
5.1.1	Hardware . . . . .	61
5.1.2	Topologie . . . . .	62
5.2	ISO-OSI-Modell . . . . .	62
5.3	Möglicher Aufbau einer Internetverbindung . . . . .	63
5.4	Namensauflösung . . . . .	63
5.5	Grundbegriffe: . . . . .	63
5.6	Client-Server-Prinzip . . . . .	64
5.6.1	Client . . . . .	64
5.6.2	Server . . . . .	65
5.6.3	Client-Programm . . . . .	65
5.6.4	Server-Programm . . . . .	67
5.7	Socket . . . . .	69
5.8	Dienste . . . . .	69
5.9	Port . . . . .	69

5.10	Codierung . . . . .	70
5.10.1	Codieren . . . . .	70
5.11	Protokolle zur Anwendung in der AT / Mech / IT . . . . .	71
<b>6</b>	<b>Datenschutz / Security</b>	<b>72</b>
6.1	Datensicherheit . . . . .	72
6.1.1	Hash-Funktionen . . . . .	72
6.1.2	Cyclic Redundancy Check (CRC) . . . . .	73
6.2	Verschlüsselung . . . . .	74
6.2.1	Symmetrische Verschlüsselung . . . . .	74
6.2.2	Assymetrische Verschlüsselung . . . . .	74

# Grundlagen der Informatik

## 1.1 Zahlensysteme und Datentypen

### 1.1.1 Meistverwendete Stellenwertsysteme

- Dezimalsystem ... Basis = 10
- Binärsystem (auch Dualsystem) ... Basis = 2
- Oktalsystem ... Basis = 8
- Hexadezimalsystem ... Basis = 16 (nach 9 wird A, B, C, D, E, F verwendet)

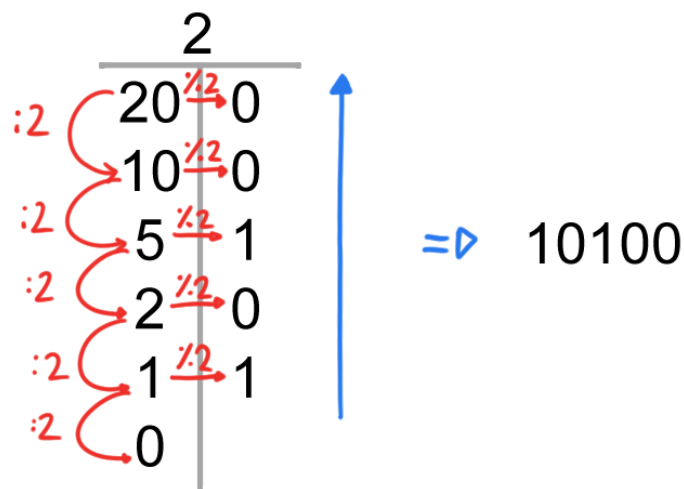


Abbildung 1.1: Trick fürs Umrechnen von Dezimal in ein anderes Stellenwertsystem

Hexadezimal	Oktal	Base64
A 5	2 4 5	C I
1 0 1 0 0 1 0 1	0 1 0 1 0 0 1 0 1	0 0 0 0 1 0 1 0 0 1 0 1

Abbildung 1.2: Trick fürs Umrechnen von einem Stellenwertsystem in Binär

Was ist ein Stellenwertsystem? -  $174 = 1 * 10^2 + 7 * 10^1 + 4 * 10^0$

Was ist kein Stellenwertsystem? - Die römischen Ziffern zum Beispiel.  $174 = CLXXIV$



**Einerkomplement und Zweierkomplement**

**Einerkomplement** - eine binäre Zahl wird invertiert. Wird verwendet um einzelne Bits zu löschen.

**Zweierkomplement** - eine binäre Zahl wird invertiert und mit 1 addiert. Wird verwendet um negative Zahlen darzustellen. Die vorderste Stelle gibt an ob es sich um eine negative (1) oder positive (0) Zahl handelt. Dadurch braucht man aber z.B. um -7 bis 7 darzustellen 4 Bits. (Rechnen mit Zweierkomplement kommt nicht zur Matura)

(PS: beim addieren von 1 und 1 muss man eine 1 eine Stelle weiter addieren. Wissen wahrscheinlich eh alle, ich schreibe nur auf damit ich mich nicht wieder wunder)

**Programm**

Es folgt ein Programm, welches eine Zahl von Stellenwertsystem 7 ins Stellenwertsystem 13 umrechnet.

```
#include <stdio.h>
#include <math.h> // fuer pow(i, n) = i^n

int main() {

    char base7[] = "163240"; // { '1', '6', '3', '2', '4', '0', '\0' }
    printf("Base_7: %s\n", base7);
    // char array -> int array
    int base7_int[sizeof(base7)-1];
    for(int i = 0; i < sizeof(base7)-1; i++) {
        base7_int[i] = base7[i] - '0'; // ASCII 0 != int 0
    }

    // convert to base 10
    int base10 = 0;
    for(int i = 0; i < sizeof(base7)-1; i++) {
        base10 += (int)pow(7,i) * base7_int[sizeof(base7) - 2 - i];
    }
    printf("Base_10: %d\n", base10);

    // convert to base13 (in reverse order)
    char base13[100];
    int base13_int[100];
    int division = base10;

    int pos = 0; // final number of digits
    while(division > 0) {
        base13_int[pos] = division % 13;
        division = division / 13;
        pos++;
    }

    // reverse order and replace e.g. correctly display numbers
    for(int i = 0; i < pos; i++) {
        int current_digit = base13_int[pos-i-1];
        if(current_digit <= 9) {
            base13[i] = current_digit + '0';
            // ASCII '0' = 48 + 1 = 49 -> ASCII '1'
        } else {
            base13[i] = current_digit - 10 + 'A';
            // 11 - 10 = 1 + 65 ('A') = 66 -> ASCII 'B'
        }
    }

    printf("Base_13: %s\n", base13);

    return 0;
}
```

## 1.1.2 Primitive Datentypen

### Deklaration, Definition und Initialisierung

#### Deklaration:

Eine **Deklaration** sagt dem Compiler, dass es eine Variable oder Funktion gibt diese aber **nicht anlegt**.

```
extern int i;
```

Bei Variablen muss man das Schlüsselwort extern verwenden, da sie sonst angelegt wird und somit auch definiert wird.

```
void foo();
```

Eine Deklaration bei einer Funktion. (Funktionskopf)

#### Definition:

Eine **Definition** passiert nach einer Deklaration und legt eine Variable oder Funktion **im Speicher an**.

```
int i;
```

```
void foo() { };
```

#### Initialisierung:

Eine **Initialisierung** passiert nach einer Definition und weist einer Variable **einen Wert zu**. Muss aber zur gleichen Zeit passieren wie die Definition sonst ist es nur eine Zuweisung.

```
int i = 1;
```

#### !!! ACHTUNG !!! KEINE INITIALISIERUNG:

```
int i;  
i = 1;
```

`i = 1` ist eine Zuweisung heißt hier geschieht keine Initialisierung.

Eine Funktion kann nicht initialisiert werden, da man einer Funktion nicht beim Definieren einen Wert zuweisen kann.

## C

- char - 1 Byte, Zweierkomplement
- int - 2 Byte (bei 16-Bit Systemen) / 4 Byte (bei 32-Bit Systemen), Zweierkomplement
- float - 4 Byte (wird nur verwendet für Effizienz bei Microcontrollern oder wenn man, IEEE 754 Speicherplatzprobleme hat)
- double - 8 Byte, IEEE 754

Mit `sizeof()` kann man die Größe eines Datentyps herausfinden.

Um mehrere Variablen mit unterschiedlichen Datentypen zusammenzufassen verwendet man **Strukturen**:

- struct (jede Komponente hat seinen eigenen Speicherplatz)
- union (braucht nur so viel Speicherplatz wie der größte Komponent im union)
- enum (Aufzählungen von Konstanten, d.h. mehrere defines zusammengefasst zu einem Thema)
- typedef (kann Datentypen neue Namen vergeben)

Strukturen müssen immer am Anfang des Programms **deklariert** werden. Dadurch kann das Programm unübersichtlich werden, wenn Funktionen die zur Struktur gehören erst an einer anderen Stelle definiert werden. (union und typedef kommen nicht hab sie nur hinzugefügt zur vollständigkeits halber.)

## Java

- boolean, Zweierkomplement
- char - 2 Byte, Zweierkomplement
- byte - 1 Byte, Zweierkomplement
- short - 2 Byte, Zweierkomplement
- int - 4 Byte (Java ist systemunabhängig daher bleibt int gleich groß), Zweierkomplement
- long - 8 Byte , Zweierkomplement
- float - 4 Byte IEEE 754
- double - 8 Byte IEEE 754

Das Java Equivalent zu Strukturen wären die Klassen, doch diese können noch viel mehr, als nur mehrere Datenelemente in einem zusammenzufassen, sondern auch Methoden (Funktionen in C) für diese Klasse definieren und ist somit übersichtlicher als in C.

## Modifizierer

- short - 2 Byte
- long - 4 Byte

## (nicht in Java)

Bei 16 oder 32-Bit Systemen wird immer short int - 2 Byte und long int - 4 Byte groß sein, obwohl int entweder 2 Byte oder 4 Byte groß ist. (kann man auch für float und double verwenden)

$\text{short int} \leq \text{int} \leq \text{long int}$

$\text{short int} < \text{long int}$

- signed - mit Vorzeichen
- unsigned - ohne Vorzeichen

**(nicht in Java)**

Ein Integer ist normalerweise immer signed und reicht von  $-2^{32}$  bis  $+2^{32} - 1$  mit unsigned reicht ein Integer von 0 bis  $2^{32} - 1$  (bei einem 32-Bit System). (kann nicht bei float oder double verwendet werden! ... und auch nicht bei char...)

## IEEE 754 Norm

Die IEEE 754 Norm wird verwendet um zu definieren wie **Gleitkommazahlen** binär abgespeichert werden. Dabei wird die Zahl in mehrere Teile aufgeteilt und in einzelnen in Bits abgespeichert wie man hier sehen kann:

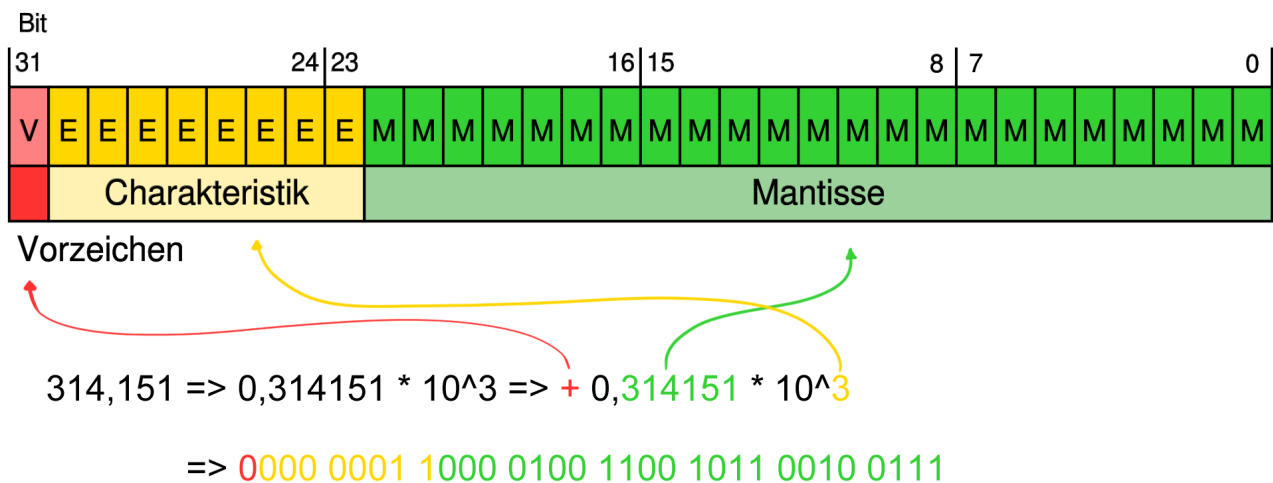


Abbildung 1.3: Abspeicherung von Gleitkommazahlen

0000 0001 1000 0100 1100 1011 0010 0111 ist eigentlich falsch das wird euch auch jeder online IEEE 754 Converter sagen, aber in HA seiner Welt ist das angeblich richtig. Wer sich gerne bilden möchte kann ja bei der folgenden **Webseite** vorbeischaun, dort wird es relativ leicht erklärt wie die Umwandlung eigentlich funktioniert und man kann auch seine eigenen Werte eingeben. <https://www.uibk.ac.at/mechatronik/mikroelektronik/lehre/webapps/ieee754.html.de>

## 1.2 Sequenz, Alternation und Iteration

(Die Flußdiagramme sind laut HA kompletter blödsinn, hab aber auch keine Ahnung wies richtig kehrt also denkts eich anfoch weg)

### 1.2.1 Sequenz

Hintereinander auszuführende Programmanweisungen.

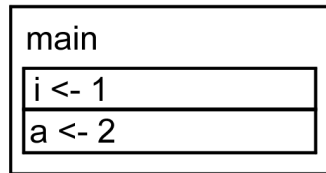


Abbildung 1.4: Eine Sequenz als Structogramme dargestellt

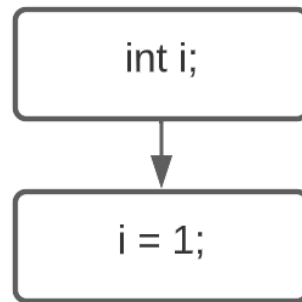


Abbildung 1.5: Eine Sequenz als Flußdiagramme dargestellt

### 1.2.2 Alternation

- if - else (Verzweigung)
- switch - case (Mehrfache Verzweigung)

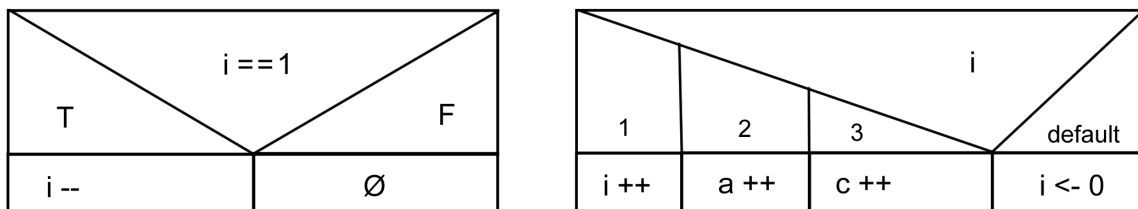


Abbildung 1.6: Alternationen als Structogramme dargestellt

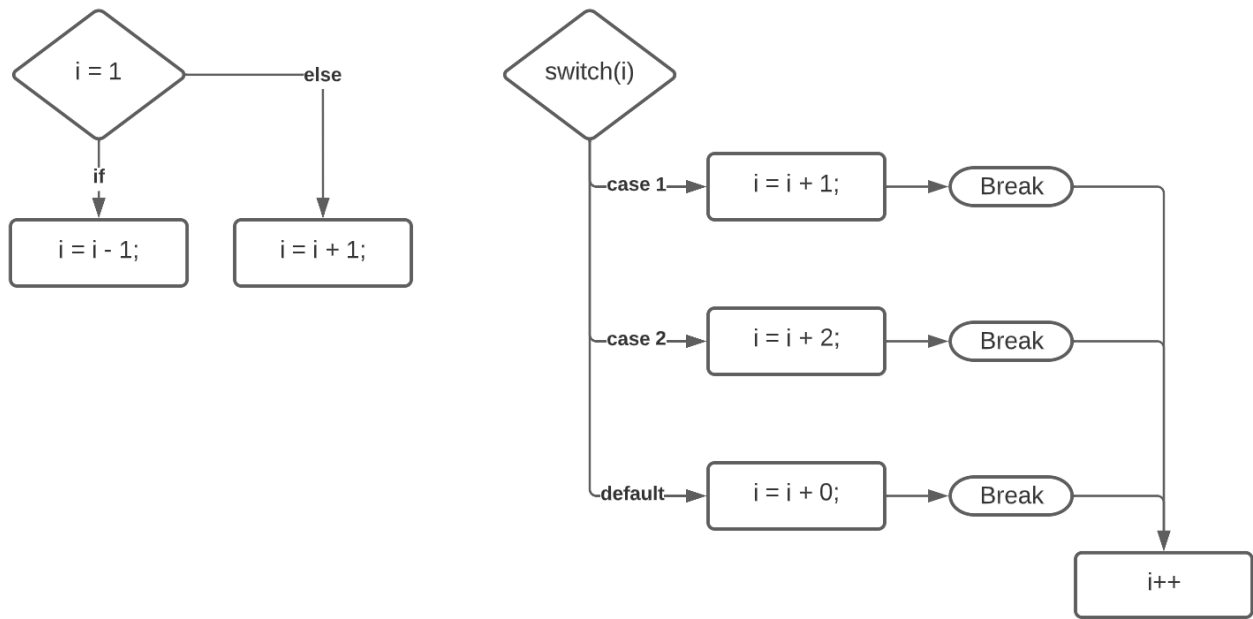


Abbildung 1.7: Alternationen als Flußdiagramme dargestellt

### 1.2.3 Iteration

- for (Zählschleife)
- while (Kopfgesteuerte Schleife)
- do - while (Fußgesteuerte Schleife)
- for - each (Mengenschleife)

Anweisungen innerhalb einer Schleife:

- continue (bricht aktuelle Iteration ab beginnt aber bei einer neuen wieder)
- break (bricht Schleife ganz ab)

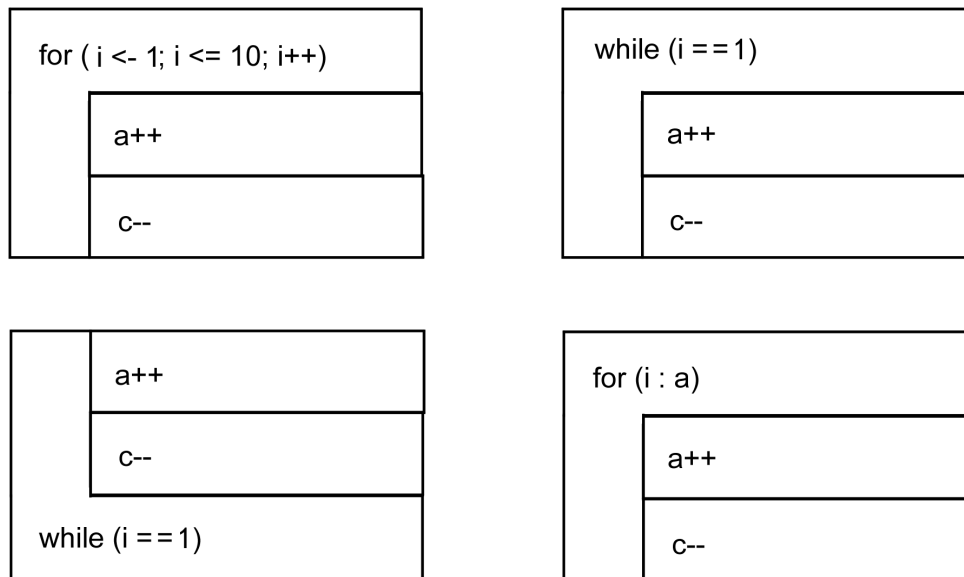


Abbildung 1.8: Iterationen als Structogramme dargestellt

## 1.3 Grundlegende Begriffe

(Laut HA kommen die nicht was auch bedeutet, dass er sie nicht kontrolliert hat. Also verwendet diese Definitionen auf eigene Gefahr.)

### 1.3.1 Personal Computer (PC)

Dieser Begriff wurde 1981 eingeführt als der erste PC (IBM PC) veröffentlicht wurde, denn zuvor waren Computer nicht für den Allgemeinverbraucher designed sondern nur für Computerexperten, Techniker und Wissenschaftler.

### 1.3.2 Betriebssystem (OS)

Ein Betriebssystem verwaltet das Zusammenspiel der Hardware- und Software-Komponenten eines Computers, sodass dieser mit einer benutzerfreundlichen Oberfläche verwendet werden kann.

### 1.3.3 Disc Operating System (DOS)

DOS oder auch MS-DOS (Microsoft - Disc Operating System) war das erste Betriebssystem für Microsoft PCs. DOS besitzt keine Grafische Oberfläche (Graphical User Interface - GUI) und wird daher als Text (Terminal User Interface - TUI) ausgegeben. Windows ist ein Virus und auch eine Weiterführung von DOS.



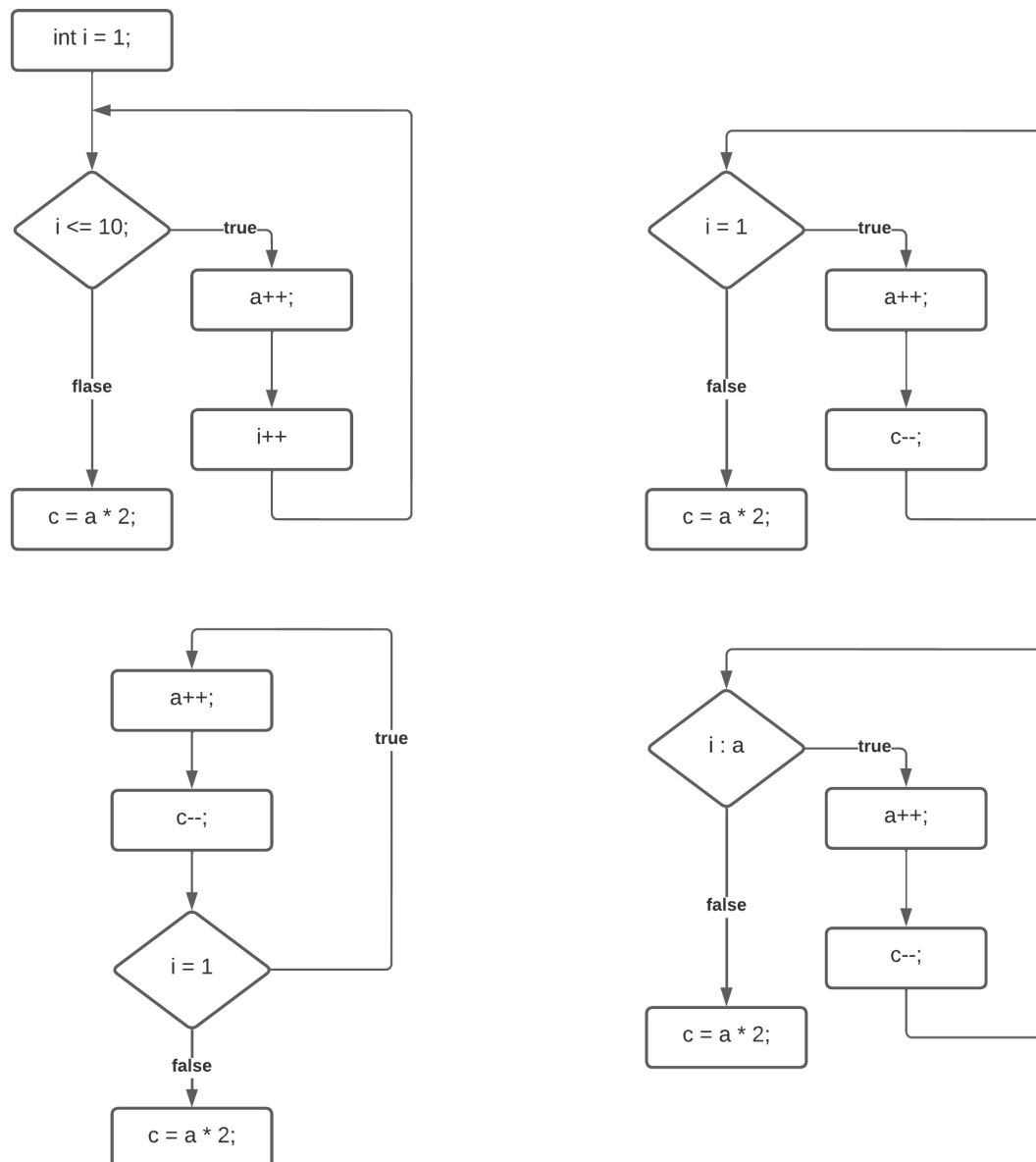


Abbildung 1.9: Iterationen als Flußdiagramme dargestellt

### 1.3.4 Central Processing Unit (CPU)

Die CPU ist der Hauptprozessor eines Computers und die dafür zuständig Befehle abzuarbeiten. Die Leistung einer CPU sind abhängig vom Anzahl der Kerne und der Taktfrequenz.

### 1.3.5 Random Access Memory (RAM)

Der Arbeitsspeicher wird verwendet um temporär Daten zu speichern welche die CPU gerade braucht um diese später schnell aufrufen zukönnen.

### 1.3.6 Read Only Memory (ROM)

Dieser Festwertspeicher ist nur dazu gedacht gelesen zu werden und wurde früher für das BIOS verwendet.

### 1.3.7 Hard Disc Drive (HDD)

Ein Festplattenlaufwerk ist ein Speichermedium, welches Daten auf einer Scheibe speichert in dem es diese an bestimmten Stellen magnetisiert.

### 1.3.8 Solid State Drive (SSD)

Eine Solid-State-Disk besteht aus Halbleitern und ist um einiges schneller beim Schreiben und Lesen von Daten als eine HDD. Sie ist auch kleiner von der Bauart und ist unempfindlich zu Erschütterungen. SSDs wurden ab 2007 erhältlich für die Allgemeinheit.

### 1.3.9 Basic Input/Output System (BIOS)

Die Aufgabe des BIOS ist den PC nach dem Einschalten funktionsbereit zu machen und anschließend das Betriebssystem zu starten. Seit ca. 2010 wird das BIOS schrittweise von UEFI abgelöst.

### 1.3.10 New Technology File System (NTFS)

NTFS ist mittlerweile das standard Dateisystem für Windows. Davor wurde das Dateisystem FAT (File Allocation Table) für Windows Systeme verwendet. Dieses System wird heutzutage hauptsächlich für externe Datenträger wie USB-Sticks verwendet.

### 1.3.11 Extended File System Version 4 (ext4)

ext4 ist im Moment das standard Dateisystem für Linux Geräte. Es hat das mittlerweile veraltete ext3 abgelöst.

### 1.3.12 Local Area Network (LAN)

Lokales Netzwerk beschreibt ein Rechnernetz, welche mit Kabel verbunden sind.

#### Ethernet

Ethernet ist ein Hardware Protokoll, welches Geräten ermöglicht via Kabel miteinander zu kommunizieren.

#### Wireless Local Area Network (WLAN)

Ein Lokales Netzwerk, welches Geräten ermöglicht drahtlos miteinander verbunden zu sein

### **1.3.13 Wide Area Network (WAN)**

Gleich wie LAN erstreckt sich aber über einen größeren Bereich.

## 1.4 Zufallszahlen

### 1.4.1 In C

- `rand()` - berechnet aus der vorherigen Zufallszahl die nächste
- `srand()` - setzt die erste Zufallszahl

Wenn man nur den **`rand()`** - Befehl benutzt, wird man immer die **gleichen Zufallszahlen** bekommen, weil der vorherige Wert beim Berechnen der ersten Zufallszahl immer 0 wäre.

Meistens(bzw fast immer) verwendet man **`srand(time(NULL))`**. **`time(NULL)`** gibt die Millisekunden seit einem bestimmten Zeitpunkt aus (1.1.1970) und ist somit immer anders, außer man startet das Programm in derselben Sekunde zweimal.

### 1.4.2 In Java

In Java gibt es zwei Möglichkeiten eine Zufallszahl zu generieren.

#### Klasse Random

In Java kann man die Klasse Random einbinden mit welcher man einen Zufallszahlengenerator erzeugen kann.

```
import java.util.Random;

public class Zufall {

    public static void main(String[] args) {

        Random rnd = new Random();
        for(int i = 1; i <= 10; i++) {
            System.out.println(rnd.nextInt(101));
        }
    }
}
```

Es werden 10 ganzzahlige Zufallszahlen im Bereich von 0 bis 100 ausgegeben. Wenn man **`rnd.nextInt()`** schreiben würde, würden Zahlen im kompletten Integerbereich  $2^{32}$  ausgegeben werden.

## Klasse Math

Wenn man die Klasse Math einbindet hat man die Möglichkeit die Funktion **Math.random()** aufzurufen mit welcher man Zufallszahlen zwischen 0.0 und 1.0 generieren kann.

```
import java.lang.Math;

public class MathZufall {

    public static void main(String args[])
    {
        for (int i = 1; i <= 10; i++) {
            int rand = (int)(Math.random() * 100);
            System.out.println(rand);
        }
    }
}
```

Es werden 10 ganzzahlige Zufallszahlen im Bereich von 0 bis 100 ausgegeben.

# Strukturierte Programmierung

## 2.1 Aufbau C-Programm

### 2.1.1 Aufbau

- Präprozessorbefehl - Funktionen - int main
- Präprozessorbefehle: Includes und defines; z.B: #include stdio.h
- Funktionen: siehe Kapitel Funktionen
- main/Menü: immerwiederkehrendes Menü; von der main aus werden die ganzen Funktionen aufgerufen

### 2.1.2 Ablauf

```
gcc:  
#include <stdio.h>  
#define N 120
```

```
1 ... Pr prozessor : alle include ... Inhalt der Datei wird ins Listing kopiert ,  
                        define ... Macro -> N wird mit 120 ersetzt
```

```
2 ... Compiler -> Objekt Datei (main.o)
```

```
3 ... Linker -> main.exe
```

## 2.2 Eingabe/Ausgabe, EVA Prinzip

### 2.2.1 EVA - Prinzip

- EVA ... Eingabe - Verarbeitung - Ausgabe
- Eingabe ... fgets
- Verarbeitung ... sscanf
- Ausgabe ... printf

### 2.2.2 printf

- um etwas an die Konsole auszugeben
- syntax: printf("Zeichenkette");
- oder: printf("ganze Zahl: %d", (int)ganzeZahl);
- oder: printf("Fließkommazahl: %lf", (double)fkZahl);
- Formatierung einer Zahl: printf("%08.3lf",fkZahl);
- .3 steht für die Anzahl der nachkommastellen
- 8 steht für den für die Zahl verwendeten Platz
- wenn man 0 davor schreibt wird der restliche Platz mit Nullen ausgefüllt

### 2.2.3 fgets

- um etwas in die Konsole einzugeben
- syntax: fgets(string, sizeof(string), stdin);
- string ... name der Zeichenkette
- sizeof(string) ... größe der Zeichenkette string
- stdin... Ort wo man den String herbekommt, also von einer Eingabe

### 2.2.4 sscanf

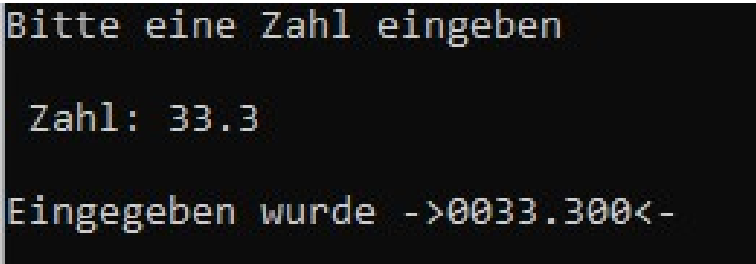
- um einen String in eine Zahl umzuwandeln
- syntax: int rückgabewert = sscanf(string, "%lf", x);
- string ... name der Zeichenkette, dessen Inhalt man umwandeln möchte
- "%lf" ... in welchen Datentyp man umwandeln möchte
- x ... in welche Variable der Wert hineingespeichert werden soll
- Rückgabewert = Anzahl der erfolgreich umgewandelten Zahlen, also normalerweise 1

### 2.2.5 Beispiel

```
printf(" Bitte eine Zahl eingeben \n\n");

do{
    printf(" Zahl: ");
    fgets(s, sizeof(s), stdin);
    n = sscanf(s, "%lf", &x);
} while (n < 1);

printf("\nEingegeben wurde ->%08.3lf<-\n", x);
```



```
Bitte eine Zahl eingeben

Zahl: 33.3

Eingegeben wurde ->0033.300<-
```

Abbildung 2.1: Beispiel01



## 2.3 Felder/Zeiger

### 2.3.1 Zeiger

Ein Zeiger ist eine Variable, dessen Wert eine Speichervariable ist.

#### Syntax

```
int *x;
```

```
int y;
```

Der Wert von x ist eine Speicheradresse. Diese Adresse zeigt auf eine int-Variable oder sie hat den Wert NULL wenn sie auf nichts zeigt.

$x = \&y$  - mit dem `&` - Operator wird die Adresse von Variablen bestimmt, mit diesem Schritt wird also erreicht, dass der Pointer x den Wert der Adresse von y hat und somit auf y zeigt

$y = y + 1$  ist das gleiche wie  $(*x) = (*x) + 1$

#### Nutzen von Zeigern

1. Übergabe als Funktionsparameter: Wenn Werte in Paramtern in der Funktion verändert werden, ist es auch nach dem Aufrufen der Funktion sichtbar
2. `char s[80];` s ist ein Zeiger und zeigt auf `s[0]`
3. hat man einen Zeiger auf eine Struktur, so kann man mit (Pfeiloperator) auf die Elemente dieser zurückgreifen
4. als Funktionspointer

#### Beispiel

Aufgabenstellung: Von einem Quader (Seite a, b und c) sollen Volumen und Oberfläche berechnet werden. Es soll als Fehler -1 gemeldet werden, wenn eine Seite negativ ist, -2 wenn zwei Seiten negativ sind bzw. -3 wenn alle Seiten negativ sind.

```
int compute(int a, int b, int c, int *volume, int *perimeter)
{
    int error=0;
    if(a<0)
    {
        error--;
    }
    if(b<0) { error--; }
    if(c<0) { error--; }

    if(error==0)
    {
        *volume = a * b * c;
        *perimeter = 2*a*b + 2*a*c + 2*b*c;
    }
}
```

```

    }
    else
    {
        *volume=0;
        *perimeter=0;
    }

    return error;
}

```

Aufrufen der Funktion

```

    int volumen;
    int umfang;

    if (compute(10,15,20, &volumen, &umfang) != 0){
        printf("FEHLER");
    }

```

### 2.3.2 Felder

Ein Feld ist prinzipiell ähnlich wie ein Pointer, nur dass ein Feld auf eine Reihe von Variablen mit dem gleichen Datentyp zeigt

#### Syntax

```
int numbers[100];
```

int ist der Datentyp der Daten im Feld, numbers der Name und 100 ist die Größe des Feldes, man kann also 100 ganzzahlige Zahlen in diesem Feld speichern.

numbers[3] = 3; so kann man auf einen bestimmten Wert des Feldes zugreifen bzw. einen Wert zuweisen

#### Nutzen von Feldern

1. Übergabe als Funktionsparameter: Wenn Werte in Parametern in der Funktion verändert werden, ist es auch nach dem Aufrufen der Funktion sichtbar
2. Man kann mehrere Werte in einem Feld speichern und muss somit nicht so viele verschiedene Variablen anlegen

#### Beispiel

```

void fillNaturalNumbers(int numbers[], int n)
{
    int i=0;
    while(i<n)
    {
        numbers[i]=i+1;
        i++;
    }
}

```

## 2.4 User-Interface / Menü

```
int printMenu(){
    int menu;
    printf("0 ... Programm beenden\n");
    printf("1 ... Natuerliche Zahlen\n");
    printf("2 ... Fibonacci Zahlen\n");
    printf("3 ... Zahlen ausgeben\n");
    printf("\n Wahl [0,3] = ");
    sscanf("%d",&menu);
    return menu;
}

int main(){
    int menu;
    int numbers[N];
    do{
        menu = printMenu();
        switch(menu){
            case 1: fillFibonacci(numbers,N);
                    break;
            case 2: fillNaturalNumbers(numbers,N);
                    break;
            case 3:
                printNumber(numbers,N);
                break;
        }
    } while(menu != 0);
}
```

[language = java]

## 2.5 Funktionen

Funktionen bestehen aus

- Funktionskopf: Datentyp, Name, Parameterliste
- Funktionsrumpf: Anweisungen und Vereinbarungen

### 2.5.1 Beispiel

Beispiel einer Funktion:

```
int Addiere(zahl1, zahl2){  
    int ergebnis = zahl1+ zahl2;  
    return ergebnis;  
}
```

Aufrufen der Funktion:

```
int ergebnis = Addiere(3,5);
```

### 2.5.2 Funktions- und Variablennamen Kriterien

- nur Ziffern, Zeichen und underline
- darf nicht mit Ziffern anfangen
- keine Schlüsselwörter

### 2.5.3 Funktions- und Variablennamen Best Practice

- Variablen und Funktionen Klein
- CamelCase schreibweise
- Zweck aus Name ersichtlich

## 2.6 Verzweigungen

### 2.6.1 Arten

- if - else
- switch - case

### 2.6.2 if (Schleife)- else

```
if(Bedingung){  
Anweisungen  
}else{  
Anweisungen  
}
```

Wenn if-Bedingung erfüllt ist, wird if Anweisung ausgeführt, und wenn nicht, dann else - Anweisung

Weitere Anwendung - siehe Beispiel in Unterkapitel Zeiger

### 2.6.3 switch - case

```
switch(ganzeZahl) {  
case 0:  
{Anweisung}  
break;  
case 1:  
case 2:  
case n:  
}
```

Es wird von einer ganzzahligen Variable (ganzeZahl) ausgegangen und je nachdem, welchen Wert diese hat, wird die entsprechende Anweisung ausgeführt.

Also wenn die ganze Zahl 0 beträgt, wird die Anweisung im case 0 ausgeführt.

Weitere Anwendung - siehe Beispiel in Unterkapitel Menü

## 2.7 Schleifen

### 2.7.1 verschiedene Schleifen

- Fußgesteuerte Schleife: do - while
- Kopfgesteuerte Schleife: while
- Zählschleife: for

### 2.7.2 do - while

- führt den do - Block, solange eine Abfrage im While erfüllt ist, aus

```
printf(" Bitte eine Zahl zwischen 0 und 100 eingeben \n\n");

do{
    printf(" Zahl: ");
    fgets(s, sizeof(s), stdin);
    n = sscanf(s, "%lf", &x);
} while (n < 1 || x < 0 || x > 100);

printf("\nEingegeben wurde ->%08.3lf<-\n", x);
```

der do - Block wird solange ausgeführt, bis der Rückgabewert kleiner als 1 ist und solange die Zahl, die eingegeben wurde nicht zwischen 0 und 100 ist

### 2.7.3 while

- Anweisungsblock wird solange die Bedingung erfüllt ist, also einen Wert ungleich 0 hat, wiederholt

```
while(Bedingung){ Anweisungen }
```

### 2.7.4 for

- for(Initialisierung; Bedingung; Reinitialisierung) {}
- Initialisierung: wird beim ersten Aufruf der Funktion durchgeführt
- Bedingung muss erfüllt werden um in die Reinitialisierung zu kommen, sonst wird die schleife geskip-  
ped
- Reinitialisierung: wird bei jedem weiteren Aufruf der Funktion durchgeführt

#### Beispiel

```
for(int i = 1; i != 5; i++){
    printf("%d",i);
}
```

Schleife zählt bis 5 und gibt bei jedem Durchlauf die aktuelle Durchlaufnummer an

## 2.8 Zufallszahlen

- `rand()` : berechnet aus der vorherigen Zufallszahl die nächste
- `srand()`: setzt die erste Zufallszahl
- Wenn man nur den `rand` - Befehl benutzt, wird man immer die gleichen Zufallszahlen bekommen, weil der vorherige Wert beim Berechnen der ersten Zufallszahl immer 0 wäre
- meistens(bzw fast immer) verwendet man `srand(time(NULL))`
- `time(NULL)` gibt die Sekunden seit einem bestimmten Zeitpunkt aus (1.1.1970) und ist somit immer anders, außer man startet das Programm in derselben Sekunde zweimal

## 2.9 Beispiel mit Funktionen, Feld befüllen, Menü, Ausgabefunktion und Struktogrammen

### 2.9.1 Quellcode

```
#include <stdio.h>
#define N 120

void fillNaturalNumbers(int numbers[], int n)
{
    int i=0;
    while(i<n)
    {
        numbers[i]=i+1;
        i++;
    }
}

void fillFibonacci(int numbers[], int n)
{
    int i;
    if (n==1)
    {
        numbers[0]=1;
    }
    if (n>1)
    {
        numbers[0]=1;
        numbers[1]=1;
        for (i=2; i<n; i++)
        {
            numbers[i] = numbers[i-1] + numbers[i-2];
        }
    }
}

void printNumber(int numbers[], int n)
{
```

```

int i;
for(i=0; i<n; i++)
{
    printf("%3d ", numbers[i]);
    if (((i+1)%10)==0)
    {
        printf("\n");
    }
}

int printMenu()
{
    int menu;
    printf("0 ... Programm beenden\n");
    printf("1 ... Natuerliche Zahlen\n");
    printf("2 ... Fibonacci Zahlen\n");
    printf("3 ... Zahlen ausgeben\n");
    printf("\n Wahl [0,3] = ");
    scanf("%d",&menu);
    return menu;
}

int main()
{
    int menu;
    int numbers[N];
    do
    {
        menu = printMenu();
        switch(menu)
        {
            case 1:
                fillFibonacci(numbers,N);
                break;
            case 2:
                fillNaturalNumbers(numbers,N);
                break;
            case 3:
                printNumber(numbers,N);
                break;
        }
    }
    while(menu != 0);
}

```



## 2.9.2 Struktogramme

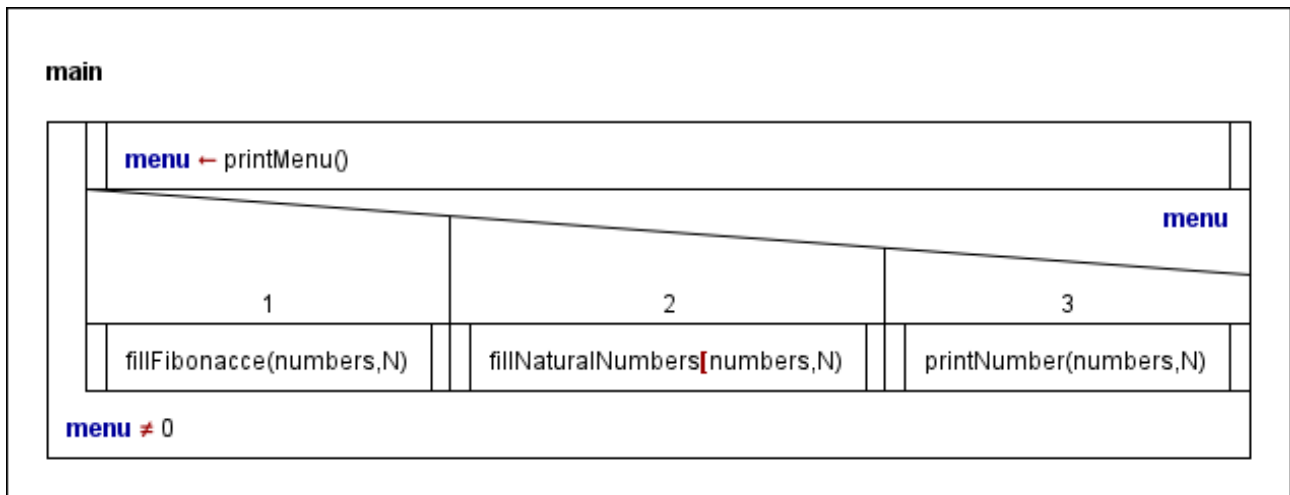


Abbildung 2.2: main

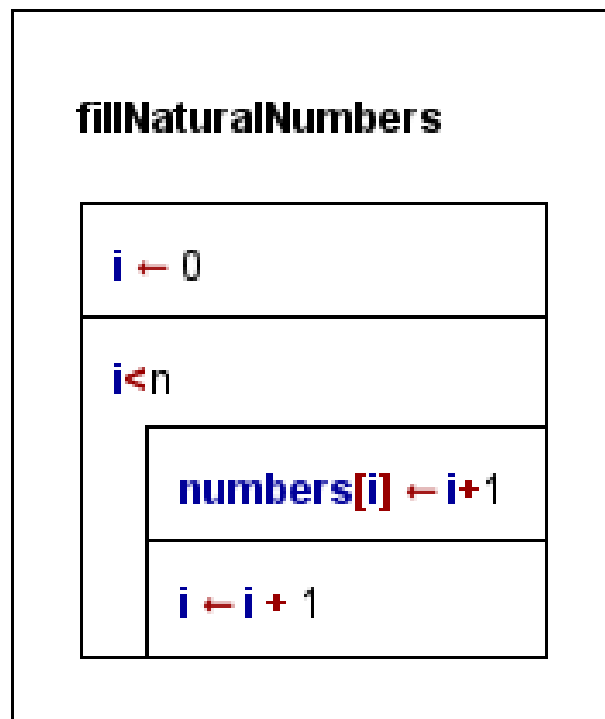


Abbildung 2.3: fillNaturalNumbers

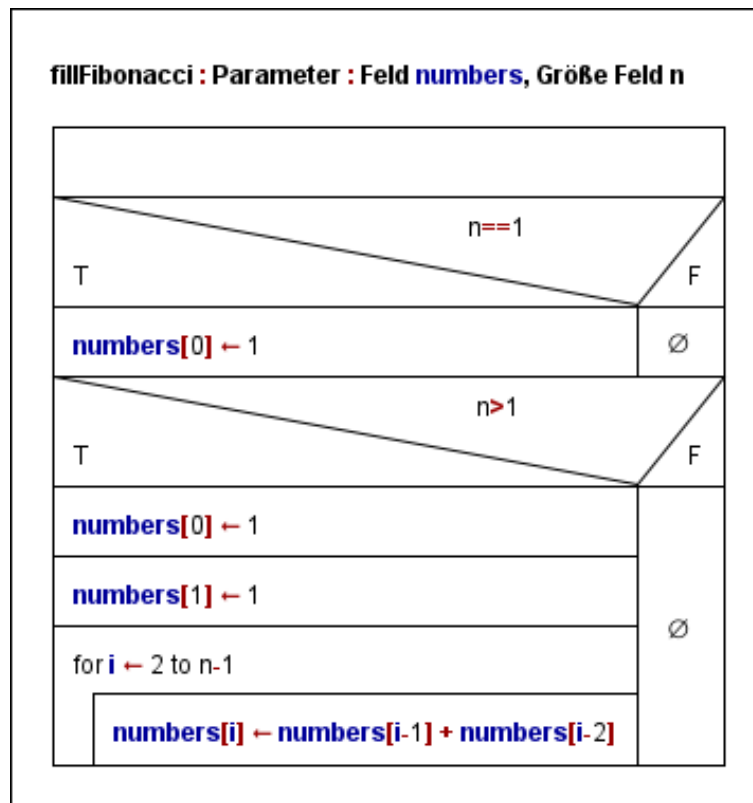


Abbildung 2.4: fillFibonacci

# Objektorientierte Programmierung

Objektorientierte Programmierung (kurz OOP) ist ein Programmierparadigma (Programmierstil), der dem menschlichen Denken ähnlich ist. In der OOP wird alles durch Objekte beschrieben. Die OOP führte die Software-Industrie aus ihrer Krise und machte es möglich robustere, fehlerärmere und besser wartbare Programme zu erstellen.

## 3.1 Programmiersprache Java (nicht unbedingt zu können)

Die Programmiersprache wurde 1995 von Sun Microsystems (2010 von Oracle aufgekauft) veröffentlicht. Java Programme werden nicht auf Betriebssystemebene ausgeführt, sondern in der Java Virtual Machine weitestgehend unabhängig vom darunterliegenden Betriebssystem ausgeführt. Die Java Virtual Machine ist Teil der Java Laufzeitumgebung, die für nahezu alle Betriebssysteme verfügbar ist. Man kann sagen, dass Java größtenteils plattformunabhängig ist. Zum Entwickeln von Java Programmen wird das JDK (Java Development Kit) benötigt, das alle Werkzeuge zur Entwicklung von Java Programmen enthält. Die Java-Syntax ist der von C/C++ sehr ähnlich.

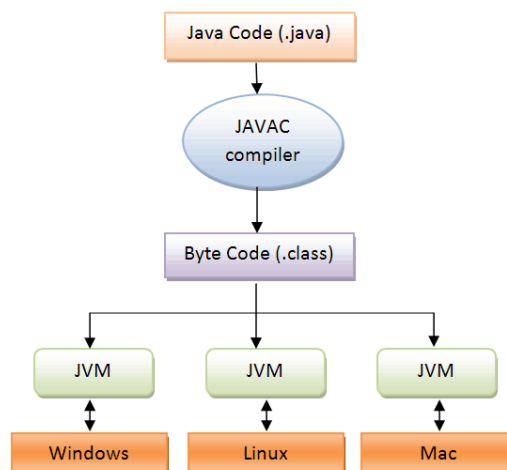


Abbildung 3.1: JVM

## 3.2 Klasse und Objekt

Eine der wichtigsten Ideen der objektorientierten Programmierung ist die Trennung zwischen Konzept und Umsetzung z.B. zwischen einem Bauplan und seinem Bauteil. In der OOP manifestiert sich diese Unterscheidung in den Begriffen Objekt und Klasse. Ein Objekt ist dabei ein tatsächlich existierendes „Ding“ aus der Anwendungswelt des Programms. Eine Klasse ist dagegen die Beschreibung eines oder mehrerer ähnlicher Objekte. Eine Klasse beschreibt mindestens drei wichtige Dinge:

- Wie ist das Objekt zu bedienen?
- Welche Eigenschaften hat das Objekt und wie verhält es sich?
- Wie wird das Objekt hergestellt?

Klassen ähneln den Strukturen aus C. Ein großer Vorteil der Klassen ist es, dass die Funktionen (in der OOP Methoden genannt), die mit den Datenelementen der Klasse arbeiten, ein Teil der Klasse sind.

### 3.2.1 Attribute (Eigenschaften)

Objekte besitzen verschiedene Eigenschaften, auch Datenelemente oder Instanzvariablen genannt.

### 3.2.2 Methoden

Methoden bestimmen das Verhalten von Objekten und somit auch das Verhalten des gesamten Programms. Sie werden in der Klassendefinition angelegt und haben Zugriff auf alle Attribute des aktuellen Objekts. Methoden sind den Funktionen anderer Programmiersprachen ähnlich, arbeiten aber immer mit den Attributen des aktuellen Objekts.

Die Syntax der Methodendefinition in Java ähnelt der von C/C++:

```
{ Modifier }  
Typ Name ([ Parameter ] )  
{  
    { Anweisung ; }  
}
```

#### Main-Methode

Eine Methode **main()** muss jede Java-Anwendung besitzen. Sie stellt den Einstiegspunkt in die Ausführung einer Java-Anwendung dar. Aufbau der Main-Methode:

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

#### Überladen von Methoden

In Java ist es erlaubt Methoden zu überladen, das heißt zwei Methoden einer Klasse können denselben Namen haben. Der Compiler unterscheidet die Varianten anhand der Anzahl und Typen ihrer Parameter.

Eine Klassendefinition wird durch das Schlüsselwort **class** eingeleitet. Darauf folgt eine beliebige Menge an Attributen und Methoden. Folgender Code ist ein Beispiel für eine einfache Klassendefinition:

```
public class Hund  
{  
    public String name;  
    public int rasse;  
    public int geburtsjahr;  
  
    public int getAlter()  
    {  
        return 2022 - geburtsjahr;  
    }  
}
```

### 3.2.3 Erzeugen eines Objekts

Objekte werden mit dem **new**-Operator erzeugt (Ausnahme Strings und Arrays). Dabei wird zuerst eine Variable vom Typ der Klasse deklariert. Dieser Variable wird dann das neu erzeugte Objekt zugewiesen.

```
Hund hund1;  
hund1 = new Hund();
```

In den Attributen der Klasse stehen zunächst nur die Standardwerte ihrer Datentypen. Der Zugriff auf die Attribute oder Methoden des Objekts erfolgt mittels Punktnotation:

```
hund1.name = "Rex";
```

### 3.3 Datenkapselung

Die Zusammenfassung von Methoden und Variablen bezeichnet man als Datenkapselung. Datenkapselung stellt eine wichtige Eigenschaft objektorientierter Programmiersprachen dar. Kapselung hilft vor allem, die Komplexität der Bedienung eines Objekts zu reduzieren. Um eine Lampe anzuschalten, muss man nicht viel über den inneren Aufbau des Lichtschalters wissen. Sie vermindert aber auch die Komplexität der Implementierung, denn undefinierte Interaktionen mit anderen Bestandteilen des Programms werden verhindert oder reduziert.

## 3.4 Zugriffsschutz

### 3.4.1 Geheimnisprinzip

Als Geheimnisprinzip wird ein Programmierkonzept bezeichnet, das darauf abzielt, alle Daten so weit wie möglich vor unkontrolliertem Zugriff von außen zu schützen. Es wird durch Datenkapselung realisiert.

In der Objektorientierung ist es ein Gebot von Sicherheit und Stabilität, dass jede einzelne Klasse den Zugriff auf ihre Strukturen nur durch genau definierte Schnittstellen zulässt. Um dies zu erreichen, müssen Methoden und Attribute so weit wie möglich vor unkontrollierten Aufrufen von außerhalb der deklarierenden Klasse geschützt werden. Man spricht auch von einer Einschränkung der Sichtbarkeit der jeweiligen Elemente. Dies geschieht i.a. dadurch, dass Zugriffe auf Methoden und Felder nur für die Bereiche erlaubt werden, die aus programmtechnischen Notwendigkeiten heraus auch wirklich den Zugriff benötigen. D.h., dass z.B. der Aufruf einer Methode aus einer anderen Klasse heraus nur dann erlaubt wird, wenn deren Ausführung an dieser Stelle des Programms auch tatsächlich notwendig ist.

Auf diese Weise wird erreicht, dass eine Klasse nur ein genau definiertes Verwendungsspektrum besitzt und falsche Zugriffe vermieden werden, etwa durch andere Entwickler im Projekt oder bei einer späteren Wiederverwendung der Klasse in einem anderen Kontext. Geschieht dies nicht, drohen bei der Komplexität moderner, objektorientierter Programme leicht Fehlfunktionen, Programmabstürze oder gar das Einschleusen von Fremdcode, etc.

### 3.4.2 Zugriffsmodifikatoren

Java stellt zum Einstellen der Sichtbarkeit drei Zugriffsmodifikatoren bereit, die in der u.a. Tabelle angegeben sind. Die Angabe package-default bezeichnet hierbei die Deklaration von Methoden und Attributen ohne gesondert angegebenen Zugriffsmodifikator. Für jeden Modifikator ist angegeben in welchen Programmteilen das mit dem Modifikator ausgezeichnete Element sichtbar ist und verwendet werden kann ('x') und in welchen nicht. ('-').

Modifikator	Klasse	Package	Unterklasse	global
public	x	x	x	x
protected	x	x	x	-
package-default	x	x	-	-
private	x	-	-	-

Abbildung 3.2: Zugriffsschutz



### 3.4.3 Getter- und Setter-Methoden

Häufig werden in Klassen die Attribute mit dem Modifier **private** versehen, damit von außen nicht auf sie zugegriffen werden kann. Dennoch braucht man manchmal ein Attribut in einer anderen Klasse oder möchte es verändern. Für diesen Fall gibt es Getter- und Setter- Methoden.

Beispiel:

```
public class Mensch{

    //Attribut
    private String name;

    //Konstruktor
    public Mensch(String name){
        this.name = name;
    }

    //Setter Methode
    public void setName(String neuename){
        this.name = neuename;
    }

    //Getter Methode
    public String getName(){
        return this.name;
    }
}
```

## 3.5 Klassendiagramm

Eine Möglichkeit der visuellen Darstellung von Klassen und deren Zusammenhänge ist das sogenannte UML Klassendiagramm. UML steht für Unified Modeling Language.

### 3.5.1 Objektorientiertes Design mit dem UML Klassendiagramm

Eine Klasse besteht aus drei Bestandteilen. Jede Klasse hat einen Namen, Eigenschaften und Methoden.

Im UML Klassendiagramm werden diese drei Bestandteile durch waagrechte Striche voneinander getrennt.

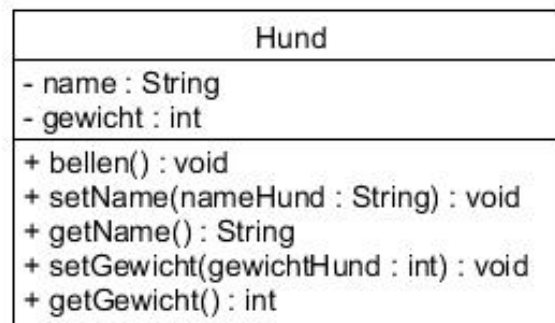


Abbildung 3.3: Klassendiagramm

Ganz oben steht der Name der Klasse.

Der mittlere Teil enthält die Klassen-Attribute. Jedes Attribut hat einen Datentyp, den man getrennt durch einen Doppelpunkt hinter den jeweiligen Attributnamen schreibt.

Die Methoden werden samt Parameterliste und Rückgabewert im unteren Teil des Klassendiagramms aufgeführt. Der Datentyp des Rückgabewertes steht hinter dem Doppelpunkt.

Das Plus und Minus vor den Attributs- und Methodennamen gibt die Sichtbarkeit an. Plus steht für public, Minus für private und ein Rautezeichen steht für protected.

### Statische Methoden und Attribute

Statische Methoden und Attribute werden im Klassendiagramm mit Hilfe eines Unterstrichs gekennzeichnet.

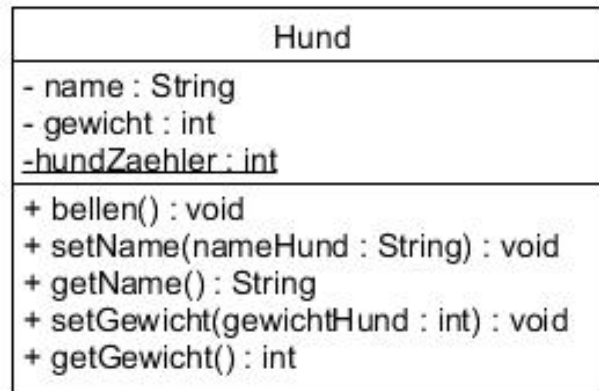


Abbildung 3.4: Klassendiagramm

### Arrays im Klassendiagramm

Um Arrays darzustellen werden hinter das Attribut eckige Klammern gesetzt. In den eckigen Klammern wird die Kapazität festgelegt. Können beliebig viele Elemente in das Array geschrieben werden schreibt man einen \* in die Klammern.

Hinter den eckigen Klammern kann man in geschwungenen Klammern festlegen, ob das Array geordnet ist oder ob jedes Element nur ein einziges mal gespeichert werden kann.

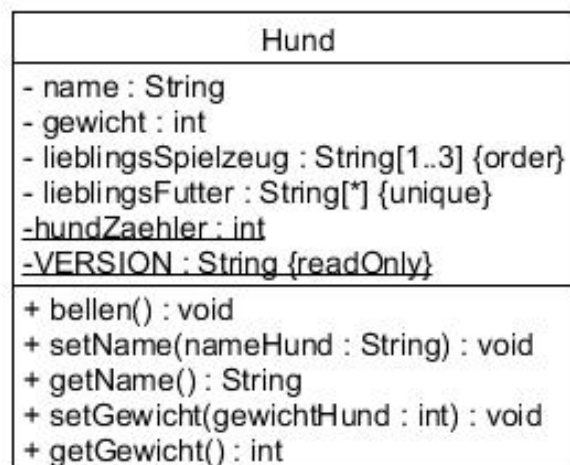


Abbildung 3.5: Klassendiagramm

### Konstanten im Klassendiagramm

Konstanten werden in Java mit Hilfe des Schlüsselwortes `final` deklariert und im UML Klassendiagramm mit dem Zusatz `readOnly` versehen.

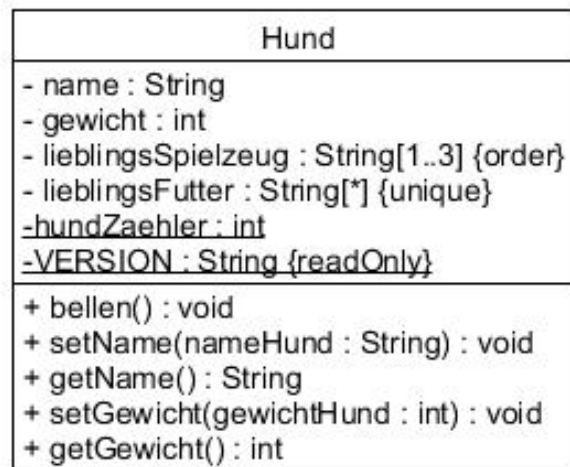


Abbildung 3.6: Klassendiagramm

### Vererbung im Klassendiagramm

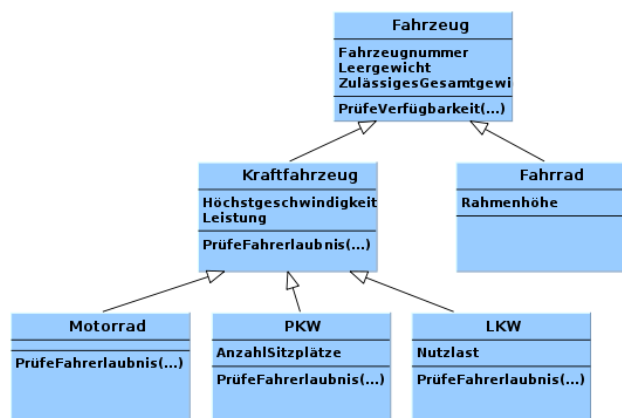


Abbildung 3.7: Klassendiagramm

## 3.6 Konstruktoren

Ein Konstruktor ist eine spezielle Methode, die bei der Initialisierung eines Objekts aufgerufen wird. Konstruktoren erhalten den Namen der Klasse, zu der sie gehören, haben keinen Rückgabewert, können beliebig viele Parameter haben und überladen werden. Jede Klasse hat einen Konstruktor. Wird der Konstruktor nicht programmiert, erstellt der Java-Compiler automatisch einen parameterlosen Standardkonstruktor.

Konstruktoren haben die Aufgabe ein Objekt nach seiner Erzeugung in einen gültigen Zustand zu bringen.

```
public class Hund {  
  
    private final String name;  
    private final String rasse;  
    private final int alter;  
  
    public Hund(String name, String rasse, int alter) {  
        this.name = name;  
        this.rasse = rasse;  
        this.alter = alter;  
    }  
  
    public static void main(String[] args) {  
        Hund hund1 = new Hund("Rex", "Malteser", 5);  
    }  
}
```

## 3.7 Design Patterns

Design-Patterns (oder Entwurfsmuster) stellen Lösungen für konkrete Programmierprobleme dar.

### 3.7.1 Immutable

In Java werden Objekte als unveränderlich (immutable) bezeichnet, wenn diese nach ihrer Instanzierung nicht mehr verändert werden können. Die Membervariablen sind nicht öffentlich zugänglich und werden im Konstruktor oder statischen Initialisierern gesetzt. Auf sie kann ausschließlich lesend zugegriffen werden.

#### **Vorteile:**

- Schutz vor Manipulation: Ein unveränderbares Objekt kann an jede Klasse gesendet werden, ohne dass diese das Objekt ändern kann.
- Verbessert die Performance:
- Nebenläufigkeit ohne Synchronisation möglich: Es können mehrere Threads erstellt werden, diese können gleichzeitig auf das unveränderbare (immutable) Objekt zugreifen.

Ein bekanntes Beispiel für eine unveränderliche Klasse wäre die Klasse String.

Um ein wirklich unveränderliches Objekt zu erhalten, muss die Klasse selbst, ihre Membervariablen und Methoden als final deklariert werden. Dies hat jedoch den Nachteil, dass die Klasse nicht abgeleitet werden kann. Würde man die Klasse nicht als final deklarieren, könnten Tochterklassen von ihr abgeleitet werden, die zwar nicht die privaten Membervariablen der Basisklasse verändern können, aber selbst veränderbare Elemente hinzufügen könnten. Dies widerspricht der Idee des Immutable Design Pattern und daher sollte die Klasse als final deklariert werden.

**Beispiel**

```

public class Hund {

public final class Immutable
{
    // Die Deklaration als final ist aus 2 Gruenden sinnvoll:
    // 1 – Zufaelliges ändern ist nicht möglich
    // 2 – Compiler-Fehler wenn Initialisierung vergessen wurde
    private final byte monat;
    private final byte[] werte;

    public Immutable(byte monat, byte[] werte)
    {
        // da ein byte-Wert uebergeben wird, muss nicht geklont werden
        this.monat = monat;

        // veraenderbare Argumente muessen geklont werden koennen
        this.werte = (byte[]) werte.clone();
    }

    public final byte getMonat()
    {
        return monat; // byte-Wert muss nicht geklont werden
    }

    public final byte getWert(int index)
    {
        // nur der nicht veraenderbare byte-Wert wird zurueckgegeben, nicht das Array-
        return werte[index];
    }
}

```

Wie im oberen Programmbeispiel zu sehen ist, sind alle Membervariablen privat und werden ausschließlich im Konstruktor gesetzt (schreibend zugegriffen). Werden veränderliche Objekte an den Konstruktor übergeben, dann müssen diese, wie in Zeile 15, kopiert (geklont) und erst danach der Membervariable zugewiesen werden.

Weiterhin darf nicht auf eine Membervariable lesend zugegriffen werden, wenn diese ein veränderliches Objekt ist. Daher wird in Zeile 25 auch nicht auf das Array selbst, sondern auf den angeforderten byte-Wert lesend zugegriffen. Dieser ist unveränderlich.

### 3.7.2 Rechnerklasse

```
public class QuaderRechner {

    //Eingangsgroessen
    private final double laenge;
    private final double breite;
    private final double hoehe;
    //Ausgangsgroessen
    private double volumen;
    private double oberflaeche;
    private double raumdiagonale;

    public QuaderRechner(double laenge, double breite, double hoehe)
        throws Exception {
        this.laenge = laenge;
        this.breite = breite;
        this.hoehe = hoehe;
        if (laenge <= 0.0) {
            throw new Exception(" Fehler: Die Laenge muss groesser als 0 sein!");
        }
        if (breite <= 0.0) {
            throw new Exception(" Fehler: Die Breite muss groesser als 0 sein!");
        }
        if (hoehe <= 0.0) {
            throw new Exception(" Fehler: Die Hoehe muss groesser als 0 sein!");
        }
        rechnen();
    }

    private void rechnen() {
        volumen = laenge * breite * hoehe;
        oberflaeche = 2 * (laenge * breite + laenge * hoehe + breite * hoehe);
        raumdiagonale = Math.sqrt(hoehe * hoehe + (laenge * laenge + breite * breite));
    }

    public double getVolumen() {
        return volumen;
    }

    public double getOberflaeche() {
        return oberflaeche;
    }

    public double getRaumdiagonale() {
        return raumdiagonale;
    }
}
```



### 3.7.3 Datenhaltungsklassen

```
public class Schueler {  
  
    private final String nachname, vorname, klassennamen;  
    private final int knr, jahrgang;  
  
    public Schueler(String nachname, String vorname, String klassennamen, int knr, int jahrgang) {  
        this.nachname = nachname;  
        this.vorname = vorname;  
        this.klassennamen = klassennamen;  
        this.knr = knr;  
        this.jahrgang = jahrgang;  
    }  
  
    public String getNachname() {  
        return nachname;  
    }  
  
    public String getVorname() {  
        return vorname;  
    }  
  
    public String getKlassennamen() {  
        return klassennamen;  
    }  
  
    public int getKnr() {  
        return knr;  
    }  
  
    public int getJahrgang() {  
        return jahrgang;  
    }  
}
```

Abbildung 3.8: Datenhaltungsklasse vom Schüler

```
public static void main(String[] args) {  
  
    List<Schueler> klasse = new ArrayList<>();  
    klasse.add(new Schueler( nachname: "Uller", vorname: "Lucas", klassennamen: "5AHME", knr: 27, jahrgang: 2017));  
    klasse.add(new Schueler( nachname: "Gutmann", vorname: "Werner", klassennamen: "Lehrer", knr: 2, jahrgang: 1965));  
    klasse.add(new Schueler( nachname: "Maier", vorname: "Michaela", klassennamen: "Sontiges", knr: 332, jahrgang: 1990));  
    System.out.println(klasse+"\n");  
}
```

Abbildung 3.9: Daten in die Datenhaltungsklasse speichern als Minimal Beispiel

Die Daten könne dann in einem Listenmodel ausgegeben werden - da ist Wichtig **ToString** zu verwenden da sonst keine Formatierung herscht.

Bei einem Tabelmodel ist das nicht notwendig da jeder Wert in eine eigenes Feld gespeichert wird. (Siehe auch Programm von Wurzinger in der 3 Klasse)

## 3.8 Vererbung

Vererbung bietet die Möglichkeit Eigenschaften vorhandener Klassen auf neue Klassen zu übertragen. Man unterscheidet dabei zwischen einfacher Vererbung, bei der eine Klasse von maximal einer anderen Klasse abgeleitet werden kann, und Mehrfachvererbung, bei der eine Klasse von mehr als einer anderen Klasse abgeleitet werden kann. In Java gibt es lediglich Einfachvererbung, allerdings gibt es mit Interfaces eine eingeschränkte Form der Mehrfachvererbung.

Um eine neue Klasse aus einer bestehenden abzuleiten, ist im Kopf der Klasse mit Hilfe des Schlüsselworts `extends` ein Verweis auf die Basisklasse anzugeben.

**Beispiel:**

```
public class Vec2D {
    private final int x;
    private final int y;

    public Vec2D(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }
}

public class Vec3D extends Vec2D{

    private final int z;

    public Vec3D(int x, int y, int z) {
        super(x, y); // Konstruktor der Oberklasse aufrufen
        this.z = z;
    }

    public int getZ() {
        return z;
    }
}
```

### 3.8.1 Interfaces

#### Definition

Ein Interface ist eine besondere Form einer Klasse, die ausschließlich abstrakte Methoden enthält. Anstelle des Schlüsselworts `class` wird ein Interface mit dem Bezeichner `interface` deklariert. Alle Methoden eines Interface sind implizit abstrakt und öffentlich. Neben Methoden kann ein Interface auch Konstanten enthalten, die Definition von Konstruktoren ist allerdings nicht erlaubt.

#### Implementierung eines Interface

Durch das bloße Definieren eines Interface wird die gewünschte Funktionalität aber noch nicht zur Verfügung gestellt, sondern lediglich beschrieben. Soll diese von einer Klasse tatsächlich realisiert werden, muss sie das Interface implementieren. Dazu erweitert sie die `class`-Anweisung um eine `implements`-Klausel, hinter der der Name des zu implementierenden Interface angegeben wird.

#### Beispiel:

```
public interface Groesse
{
    public int getLaenge();
    public int getHoehe();
    public int getBreite();
}

public class Auto2 implements Groesse
{
    public String name;
    public int    erstzulassung;
    public int    leistung;
    public int    laenge;
    public int    hoehe;
    public int    breite;

    public int getLaenge()
    {
        return this.laenge;
    }

    public int getHoehe()
    {
        return this.hoehe;
    }

    public int getBreite()
    {
        return this.breite;
    }
}
```

## Verwenden eines Interface

Nützlich ist ein Interface immer dann, wenn Eigenschaften einer Klasse beschrieben werden sollen, die nicht direkt in seiner normalen Vererbungshierarchie abgebildet werden können. Hätten wir beispielsweise `Groesse` als abstrakte Klasse definiert, ergäbe sich eine sehr unnatürliche Ableitungshierarchie, wenn Autos, Fußballplätze und Papierblätter daraus abgeleitet wären. Durch Implementieren des `Groesse`-Interface können wir die Verfügbarkeit der drei Methoden `laenge`, `hoehe` und `breite` dagegen unabhängig von ihrer eigenen Vererbungslinie garantieren.

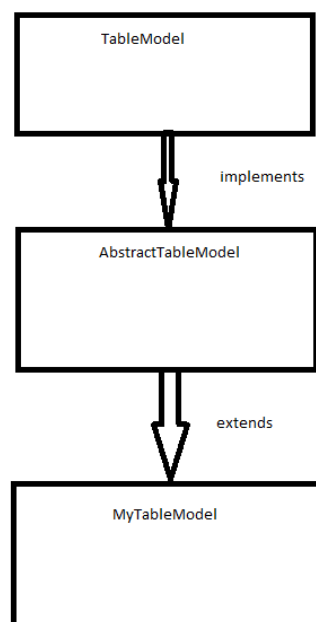
### 3.8.2 Abstrakte und konkrete Klassen

In Java ist es möglich, abstrakte Methoden zu definieren. Im Gegensatz zu den konkreten Methoden enthalten sie nur die Deklaration der Methode, nicht aber ihre Implementierung.

Eine Klasse, die mindestens eine abstrakte Methode enthält, wird selbst als abstrakt angesehen und muss ebenfalls mit dem Schlüsselwort `abstract` versehen werden. Abstrakte Klassen können nicht instanziiert werden, da sie Methoden enthalten, die nicht implementiert wurden. Stattdessen werden abstrakte Klassen abgeleitet und in der abgeleiteten Klasse werden eine oder mehrere der abstrakten Methoden implementiert. Eine abstrakte Klasse wird konkret, wenn alle ihre Methoden implementiert sind. Die Konkretisierung kann dabei auch schrittweise über mehrere Vererbungsstufen erfolgen.

#### Beispiel `AbstractTableModel`

Ein Tabellenmodell muss das Interface `TableModel` implementieren. Damit man nicht alle abstrakten Methoden des Interface selbst konkret ausprogrammieren muss, gibt es die abstrakte Klasse `AbstractTableModel`. In dieser Klasse sind die meisten abstrakten Methoden bereits implementiert. Wenn man sein eigenes Tabellenmodell verwenden will muss man es lediglich von `AbstractTableModel` ableiten und die abstrakten Methoden `getRowCount`, `getColumnCount` und `getValueAt` implementieren.



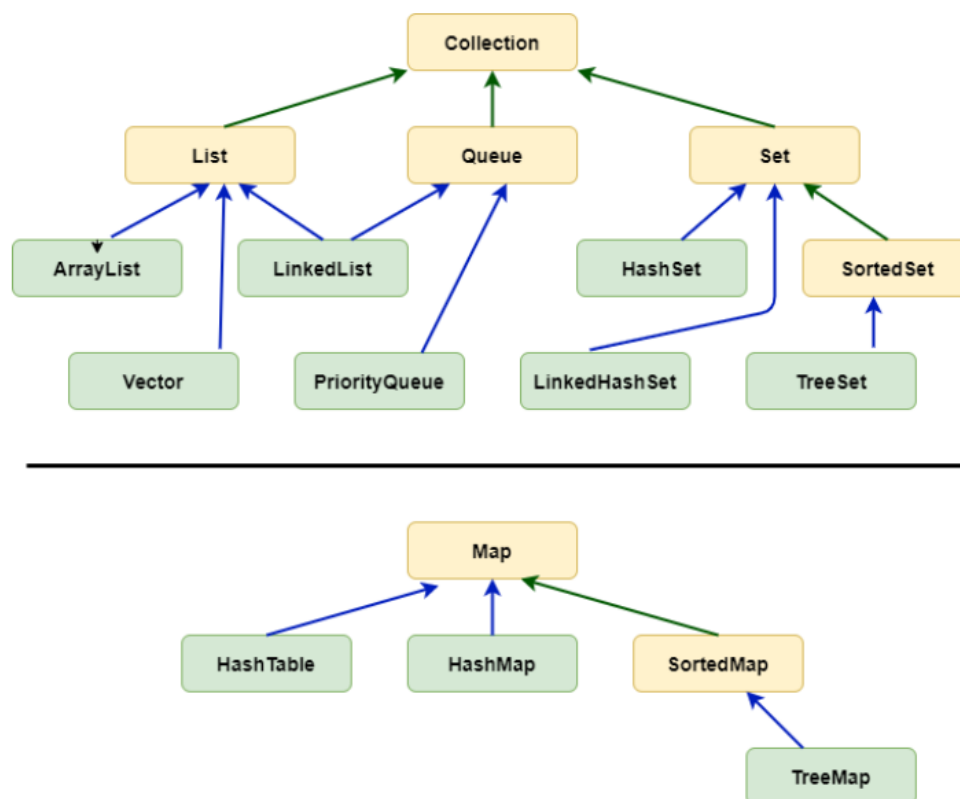
## 3.9 Collections

Eine Collection ist eine „Sammlung“ von Objekten gleichen Typs.

Die Klassen des Java Collection Framework in `java.util` stellen vier Familien von abstrakten Datentypen und Funktionen zur Verfügung:

- **Listen:** geordnete Datenstrukturen auf die man wie auf Felder mit einem numerischen Index zugreifen kann. Im Unterschied zu Feldern (Array) erlauben sie das listentypische Einfügen an einer beliebigen Stelle.
- **Sets:** Eine Implementierung mathematischer Mengen. Objekte können nur einmal in einer Menge vorkommen. Man kann prüfen ob bestimmte Objekte in einer Menge enthalten sind. Eine Reihenfolge oder Ordnung in der Menge ist nicht relevant.
- **Verzeichnisse (Map):** Verzeichnisse können als verallgemeinerte Felder angesehen werden. Felder erlauben einen direkten zugriff mit einem numerischen Index. Verzeichnisse erlauben die Wahl einer beliebigen Zugriffskriteriums. Man kann zum Beispiel Personen nach ihrem Nachnamen verwalten und eine Person mit Hilfe eines gegebenen Nachnamens aufrufen.
- **Warteschlangen(Queue):** Warteschlangen sind Listen die nach dem FIFO Prinzip (First In , First Out) aufgebaut sind. Sie verfügen über keinen wahlfreien Zugriff.

Vererbungshierarchie Collections:



### 3.9.1 ArrayList

Bei einer ArrayList werden die Objekte intern in einem dynamischen Array gespeichert. Vorteilhaft an einer ArrayList ist es, dass direkt Zugriffe auf einzelne Elemente performant sind. Allerdings ist das Hinzufügen und Löschen von Elementen nicht besonders effizient, weil ein neues Array erstellt werden muss und alle Elemente kopiert werden müssen.

Eine ArrayList sollte man verwenden, wenn die Anzahl der Lesezugriffe größer als die Häufigkeit des Hinzufügens oder Entfernens ist.

#### Erstellen einer ArrayList

In den spitzen Klammern wird angegeben, welcher Objekttyp in der Liste gespeichert wird. Primitive Datentypen können nicht gespeichert werden, deshalb muss man die entsprechende Wrapper-Klasse verwenden.

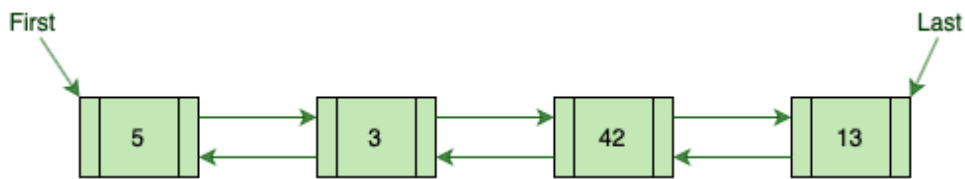
```
List<Integer> list = new ArrayList<>;
```

#### Wichtige Methoden

- Element ausgeben: `liste.get()`
- Element hinzufügen: `list.add()`
- Element ändern: `list.set()`
- Element entfernen `list.remove()`
- Größe zurückgeben: `list.size()`

### 3.9.2 LinkedList

Eine LinkedList verwendet intern eine doppelt verkettete Liste. Jedes "Kettenglied" besitzt einen Zeiger auf das vorherige und das nächste Element.



Das Hinzufügen und Löschen von Elementen geht deutlich schneller im Vergleich zur ArrayList. Dafür ist die ArrayList bei Direktzugriffen schneller. Werden Elemente häufig hinzugefügt oder gelöscht ist die LinkedList besser als die ArrayList.

#### Erstellen einer LinkedList

```
List<Integer> list = new LinkedList<>;
```

#### Wichtige Methoden

- Element ausgeben: `liste.get()`
- Element hinzufügen: `list.add()`
- Element ändern: `list.set()`
- Element entfernen `list.remove()`
- Größe zurückgeben: `list.size()`

## Visualisierungen / Graphische Oberflächen

### 4.1 Rahmenfenster / JFrame

Ein JFrame ist die Grundlage für ein GUI. GUI ist die Abkürzung für „Graphical User Interface“. Ein GUI kann je nach Anforderung unterschiedlichst angepasst werden.

### 4.2 Dialogfenster / JDialog

Dialogfenster werden verwendet, um zum Beispiel einer Liste oder Tabelle einen Wert hinzufügen zu können.

### 4.3 Layouts

Um die verschiedenen Elemente aus Java-Swing richtig anordnen bzw. anpassen zu können, werden diese in ein Layout gelegt. Die Layouts dienen meist zur richtigen Positionierung der einzelnen Elemente.

#### 4.3.1 Übersicht

##### **Border Layout**

Ausrichtung: Norden, Süden, Osten, Westen, Zentrum

##### **Box Layout**

Controls werden in x und y Richtung zentriert

##### **Flow Layout**

Controls werden in x-Richtung zentriert



## Grid Layout

Der Platz wird in Zeilen und Spalten aufgeteilt. Die einzelnen Controls werden auf die ganze Größe der Felder aufgezogen.

## Grid Bag Layout

Kann im Prinzip alles, was andere Layouts können und lässt sich sehr leicht anpassen. Das Grid Bag Layout ist jedoch ineffizient und sollte deshalb nur verwendet werden, wenn es wirklich benötigt wird bzw. kein anderes Layout den Anforderungen gerecht wird.

### 4.3.2 Schachteln von Layouts

Bei den meisten Rahmenfenster ist es aufgrund komplexerer Anforderungen nötig, verschiedene Layouts zu schachteln. Der Aufbau einer solchen Schachtelung ist im nachfolgenden Beispiel noch ersichtlich.

## 4.4 Java-Swing

Java-Swing ist die Umgebung, in der die Benutzeroberfläche mittels "Klicksi-Klacksi" zusammengebaut werden kann. Hierfür werden verschiedenste Controls, Containers etc. verwendet.

### 4.4.1 Übersicht

- Panel  
Ein Panel ist eine rechteckige leere Fläche, der ein Layout zugewiesen werden kann. Panels werden auch bei der Schachtelung von Layouts vermehrt verwendet.
- Scroll Pane  
Ein Container, dessen Inhalt scrollbar gemacht wird bzw. mit vertikalen und horizontalen Scroll-Bars versehen wird.
- Label  
Kann einen Text, Beschriftung etc. darstellen
- Button  
Eine Schaltfläche, die mit Hilfe von Handler-Methoden eine zugewiesene Aktion ausführt
- Combo Box  
Ein Dropdown-Menü mit dem man verschiedenste Dinge auswählen kann.

Um eine Combo Box zu befüllen wird in der Regel der Befehl *addItem* benutzt. Oft muss diese aber auch mit Elementen einer Enum-Klasse gefüllt werden. Hierfür wird eine for-Schleife verwendet wie unten dargestellt. Das im unteren Beispiel veranschaulichte Enum soll verschiedene Obstsorten zur Auswahl bereitstellen.

```
private void fillComboBox()  
{  
    cbObst.removeAllItems();  
    for (Obst obst : Obst.values())  
        cbObst.addItem(obst);  
}
```

```
}
```

- Text Field  
Hier kann eine Ein- oder Ausgabe bewerkstelligt werden.
- Formatted Field  
Textfeld mit formatierter bzw. persönlich angepasster Darstellung
- Table  
Eine Tabelle, die mit Hilfe eines Tabellenmodells realisiert werden kann. Ist mit Zeilen und Spalten aufgebaut.
- Menu Bar  
Erzeugt eine Menüleiste die angepasst werden kann.
- JFileChooser  
Wird eingesetzt um auf Dateien zuzugreifen. Wird meistens beim Speichern oder Laden von Daten genutzt. Der JFileChooser öffnet ein Dialogfenster, dass Zugriff auf die Dateien gewährt. Eingeliedert wird der JFileChosser meist in einer Handler-Methode. Zur Veranschaulichung dient der darunter stehende Code.

```
private void onSave(java.awt.event.ActionEvent evt) {
    JFileChooser fc = new JFileChooser();
    if(fc.showSaveDialog(this) == APPROVE_OPTION){
        try {
            positionen.save(fc.getSelectedFile());
        } catch (Exception ex){
            JOptionPane.showMessageDialog(this, ex, "FEHLER", JOptionPane.ERROR_MESSAGE);
        }
    }
}

private void onLoad(java.awt.event.ActionEvent evt) {
    JFileChooser fc = new JFileChooser();
    if(fc.showOpenDialog(this) == APPROVE_OPTION){
        try {
            positionen.open(fc.getSelectedFile());
        } catch (Exception ex){
            JOptionPane.showMessageDialog(this, ex, "FEHLER", JOptionPane.ERROR_MESSAGE);
        }
    }
}
```

#### Erklärung:

Die beiden Handler-Methoden *onSave* und *onLoad* dienen grundsätzlich zum Speichern und Laden einer Tabelle. Es wird ein neues Objekt des Typs JFileChosser erzeugt. Mit *showSaveDialog(this)* wird das Dialogfenster zum Speichern bzw. mit *showOpenDialog* zum Laden geöffnet. *Positionen* ist ein Objekt der Datenhaltungsklasse und mit *.getSelectedFile* wird die ausgewählte Datei zum Speichern bzw. Laden übernommen.

#### 4.4.2 Wissenwertes

Wenn man in einem JLabel ein gewisses Format haben möchte, wie zum Beispiel einen Zeilenumbruch, ist das via html Code möglich.

Das geht so:

```
<html>1. Zeile<br>2. Zeile</html>\\
```

—> mit `<html>` bzw. `</html>` wird die Eingabe geöffnet bzw. geschlossen

—> mit `<br>` erzeugt man einen Zeilenumbruch

Wenn man in einem Textfield den vorgegeben Text löschen möchte bzw. ein leeres Feld zur Eingabe haben möchte muss man die Größe händisch eingeben. Ansonsten wird das Textfeld automatisch ganz klein.

Das geht so: Rechtsklick auf das Textfeld → Properties → preferred size → [länge,breite] ([100,20])

Um im Rahmenfenster einen Einzug zu erzeugen, also einen Rand, ist eine Border nötig.

Das geht so: Rechtsklick auf JLabel → Border → empty Border → Insets: [5,5,5,5]

Um null-Pointer Exceptions zu vermeiden (das Programm kann die Werte aus den Textfeldern nicht auslesen) muss man den FormattedTextFields den Value 0 bzw. bei double Zahlen 0.0 zuweisen.

Einfach unter den Properties der Text Felder den Punkt „value“ suchen, dann auf die drei Punkte, danach „Costum Code“ auswählen und 0 eintragen.

## 4.5 Listenmodell

Für in Listenmodell wird eine Java-Klasse erstellt. Mittels extends `AbstractListModel<Integer>` werden durch Vererbung alle benötigten Methoden verfügbar.

- `getSize()` liefert die Größe der Liste bzw. die Anzahl der Elemente
- `getElementAt()` liefert die Position eines Elements der Liste

```
public class TestListModel extends AbstractListModel<Integer>
{
    @Override
    public int getSize()
    {
        return Groesse der Liste zB 100;
    }

    @Override
    public Integer getElementAt(int index)
    {
        return index + 1;
    }
}
```

## 4.6 Tabellenmodell

Ein Tabellenmodell wird meist öfter genutzt als das einer Liste, da es durch die Spalten der Tabelle vielfältiger ist. Hierfür wird ebenfalls eine Java-Klasse erstellt.

`getRowCount()`, `getColumnCount()` und `getValueAt()` werden automatisch eingefügt bzw. müssen immer vorhanden sein. Die Override-Methode `getColumnName()` kann bei Bedarf mit Alt+Einfügen eingefügt werden.

- `getRowCount()` liefert die Anzahl der Reihen
- `getColumnCount()` liefert die Anzahl der Spalten
- `getValueAt()` hier können einzelnen Positionen Werte zugewiesen werden bzw. der Wert an einer Position bestimmt werden
- `getColumnName()` weißt den Spalten ihre Namen zu. Hierfür wird am Anfang ein Feld von Strings mit den Spaltennamen erzeugt und danach auf dieses zugegriffen.

```
public class TestTableModel extends AbstractTableModel
{
    private final static String[] colNames = {"Aepfel", "Birnen", "Zwetschken", "Dat

    @Override
    public int getRowCount()
    {
        return 3;
    }

    @Override
    public int getColumnCount()
    {
        return 5;
    }

    @Override
    public Object getValueAt(int row, int column)
    {
        if (row == 1 && column == 3)
            return "Kartoffel";
        return String.format("%04d/%04d", row + 1, column + 1);
    }

    @Override
    public String getColumnName(int column)
    {
        return colNames[column];
    }
}
```

## 4.7 Handler-Methoden

Handler Methoden erwecken Schaltflächen zum Leben.

Sie weisen zum Beispiel einem Button eine Funktion zu, die erfüllt soll wenn er gedrückt wird.

Um eine Handler Methode zu erzeugen sind folgende Schritte durchzuführen:

Rechtsklick auf den Button → Events → actionPerformed → Namen eingeben → Enter

Handler-Methoden unterscheiden sich von Beispiel zu Beispiel und müssen je nach Anforderung individuell angepasst werden. Die meist genutzte Methode ist dennoch *onEnd* oder wie man sie auch immer nennt, um das Programm zu beenden. Das beenden wird mit dem einfachen Befehl *dispose()* bewerkstelligt.

## 4.8 Swing-Worker

Der Swing-Worker wird benötigt, damit das Programm bei längeren Ladezeiten des Ergebnisses trotzdem bedienbar bleibt. Ansonsten könnte man in der Wartezeit nichts anfangen.

Im Swing-Worker dürfen nur Dinge stehen, die nichts mit der GUI zu tun haben.

### Erklärung

- Swing-Worker mit den Parametern `Integer, Object`
- Alt+Einfügen → ImplementMethod → `doInBackground`
- Alt+Einfügen → OverrideMethod → `done`
- In `doInBackground` wird wie der Name schon verrät der im Hintergrund auszuführende Prozess geschrieben
- in „`done()`“ kann zum Beispiel ein Ergebnis einer Rechnung, einer Variable zugewiesen werden
- mit „`tfErgebnis.setValue(ergebnis)`“ wird der Wert der Variable `ergebnis` dem Textfeld `tfErgebnis` zugewiesen (damit die GUI das Ergebnis anzeigt) –> Muss nicht immer sein (Programmabhängig) 2 Exceptions werden gefangen (immer gleich)
- In `JOptionPane` muss man dann noch 2 Änderungen vornehmen, statt „`this`“ gehört „`null`“ und auf die Zeile „`e.getCause()`“ darf nicht vergessen werden da ansonsten die falsche Exception gefangen werden könnte
- `.execute();` zum ausführen des Swingworkers

```
new SwingWorker<Integer , Object>()
{
    @Override
    protected Integer doInBackground() throws Exception
    {
        //Im Hintergrund auszufuehrender Prozess
    }

    @Override
    protected void done()
    {
```

```
try
{ // Beispielcode
  final int ergebnis = get();
  tfErgebnis.setValue(ergebnis);
}
catch (InterruptedException e){}
catch (ExecutionException e)
{
  JOptionPane.showMessageDialog(null, e.getCause().getMessage(),
    "Fehler aufgetreten", JOptionPane.ERROR_MESSAGE);
}
}.execute();
```

## 4.9 Beispiel

**Beispiel für das Erstellen einer GUI mit Hilfe von geschachtelten Layouts**

# Networking

## 5.1 Ethernet

**def.:** "Ethernet ist die Norm, die die Schichten des Internets beschreibt und wird auch als ein paketorientiertes Protokoll angesehen."

### 5.1.1 Hardware

#### BNC

- veraltet
- $v = 10\text{Mbit/s}$

#### Twisted Pair

- modern
- besteht aus 4 verdrehten Adernpaare (8 Leitungen)
- "Western Stecker" (RJ 45)
  - $v = 10\text{Mbit/s}$
  - $v = 100\text{Mbit/s}$  ; max. 110m!
  - $v = 1\text{Gbit/s}$
  - $v = 10\text{Gbit/s}$

#### Glas

- $>1\text{Gbit/s}$
- hohe Reichweite

### 5.1.2 Topologie

#### Bus

- geht ein Kabel kaputt, so kommt das gesamte Netz zum Stillstand
- Hardware: Switch (Jeder Zweig volle Bandbreite, keine Kollisionen)

#### Stern

- Hardware: ein elektrischer Verstärker, sogenannter Hub (Bandbreite wird auf alle Geräte aufgeteilt)

#### Ring

- Ein Ringnetzwerk ist eine Netzwerktopologie, bei der jeder Knoten mit genau zwei anderen Knoten verbunden ist und einen einzigen kontinuierlichen Pfad für Signale durch jeden Knoten bildet - einen Ring. Daten werden von Knoten zu Knoten übertragen, wobei jeder Knoten auf dem Weg jedes Paket verarbeitet.

## 5.2 ISO-OSI-Modell

Damit über das Internet ein Informationsaustausch gelingen kann, bedarf es Protokollen. Diese Protokolle werden in den jeweiligen Paketen angewandt und in einander verschachtelt. Die maximale Paketgröße beträgt 1500 Bytes an Nutzdaten. Dieser Paketschachtelung wird wie folgt beschrieben:

- **Erste Schicht: Ethernet(MAC)** ist eindeutig und 6 Byte groß. Beispiel: *E4-B9-7A-DE-00-94*
- **Zweite Schicht: IP (IPv4, IPv6)** Mit IPv4 sind 2 hoch 32, also 4.294.967.296 Adressen darstellbar. Aufgrund der Knappheit der IPv4-Adressen wurden IPv6 ins Leben gerufen.
- **Dritte Schicht: TCP/IP oder UTP/IP**  
*TCP/IP:* sorgt dafür, dass Pakete in der richtigen Reihenfolge und vollständig ankommen. Es wird eine Verbindung zwischen Sender und Empfänger aufgebaut und die Pakete werden nummeriert, somit ist kein Datenverlust möglich. Nahezu alle Arten an Datenaustausch außer dem Streaming basieren auf diesem verbindungsorientierten Protokoll.  
*UDP/IP:* Ziel ist es, mit maximaler Geschwindigkeit Pakete von A nach B zu schicken. Auf Datenverlust wird keine Rücksicht genommen. Ein typischer Anwendungsfall ist das Streaming.



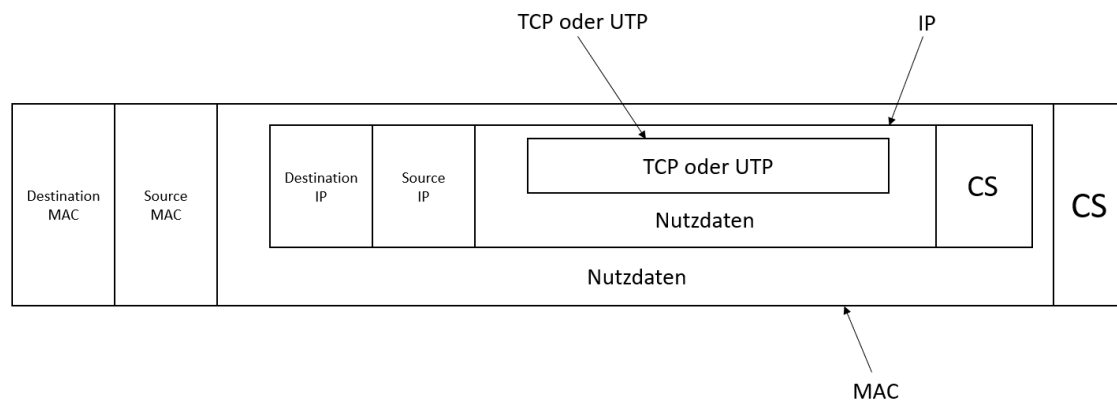


Abbildung 5.1: Protokollstapel

### 5.3 Möglicher Aufbau einer Internetverbindung

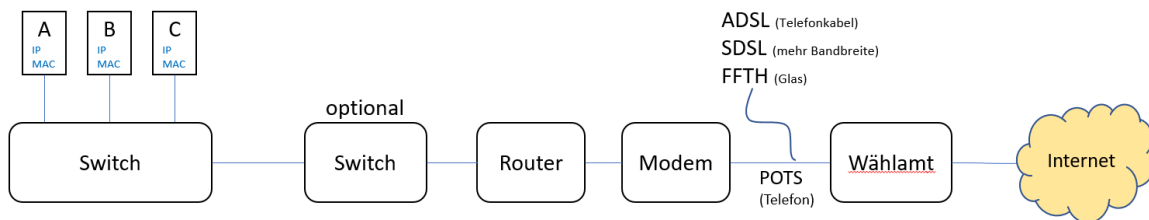


Abbildung 5.2: Netzwerkaufbau

### 5.4 Namensauflösung

Als Namensauflösung bezeichnet man Verfahren, die es ermöglichen, Namen von Rechnern beziehungsweise Diensten in vom Computer bearbeitbare, meist numerische Adressen zu übersetzen.

1. URL wird vom Benutzer eingegeben
2. Über das DNS (Domain Name System) erfolgt die Umwandlung (Namensauflösung) von einer URL in eine IP-Adresse.

### 5.5 Grundbegriffe:

- **LAN** ... Local Area Network
- **WAN** ... Wide Area Network
- **ISO-OSI (Schichten)modell** ... Protokollstapel, mit dem sich die Kommunikation zwischen Systemen beschreiben und definieren lässt. Es besitzt sieben einzelne Schichten (Layer) mit jeweils klar voneinander abgegrenzten Aufgaben.
- **ARPA-Net** ... Vorgänger vom Internet
- **MAC-Adresse** ... eindeutig, 6 Byte

- **IP-Adresse** Es benötigt ausschließlich der Router eine IP-Adresse um ins Internet zu gelangen.
  1. Wem gehört das Netz?
  2. Wem gehört der Rechner

**Beispiel:**

12 .	7 . 132 . 51
Netzadresse	Hostadresse(Rechneradresse)

Abbildung 5.3: Beispiel IP-Adresse

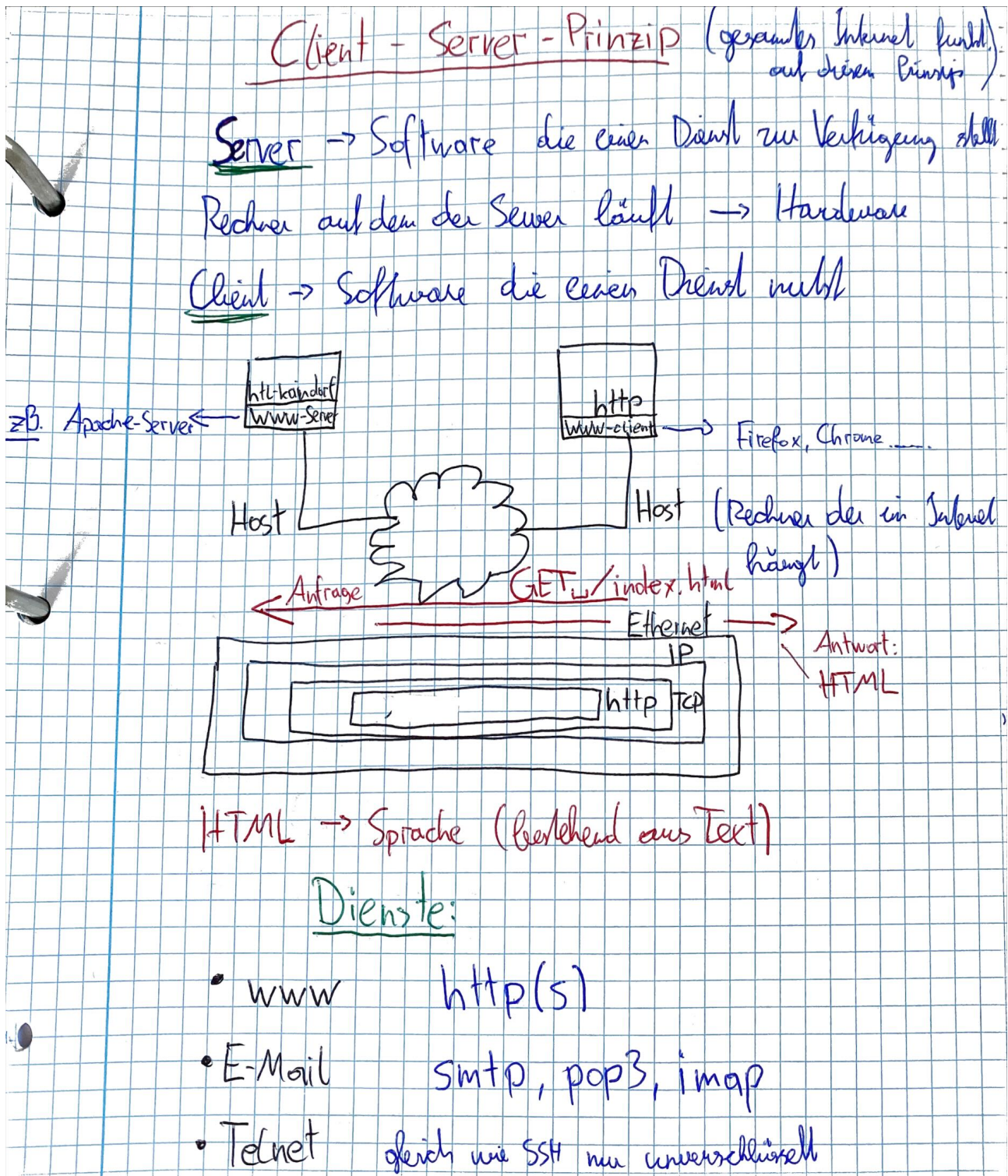
Grundsätzlich gibt es drei Adressbereiche von IP-Adressen:

- öffentliche, routbare IP-Adressen
  - private, nicht routbare IP-Adressen
  - reservierte IP-Adressen
- **DHCP (Dynamic Host Configuration Protocol)** ... ermöglicht die Zuweisung der Netzwerkkonfiguration an Clients durch einen Server.
  - **Backbone** ... Netz der riesigen Leitungen, zB. Unterseekabel zwischen Kontinenten

## 5.6 Client-Server-Prinzip

### 5.6.1 Client

Der Client fragt einen bestimmten Dienst beim dafür zuständigen Server an. Die Initiative für die Kommunikation geht immer vom Client aus. Er formuliert die Anfrage, sendet sie lokal oder über das Netzwerk an den Server und interpretiert die erhaltene Antwort.



### 5.6.2 Server

Ein Server (engl. „Diener“) ist in der Informatik ein Dienstleister, der in einem Computersystem Daten oder Ressourcen zur Verfügung stellt. Das Computersystem kann dabei aus einem einzelnen Computer oder einem Netzwerk mehrerer Computer bestehen.

### 5.6.3 Client-Programm

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
```

```

import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.net.InetAddress;
import java.net.Socket;

public class SimpleClient
{
    public static String sendRequestAndRecieveResponse(
        String host, int port, String request)
        throws Exception
    {
        #Namensaufl sung
        #getName = factory Methode (statische Methode die Objekt erzeugt
        final InetAddress adr = InetAddress.getByHost(host); #InetAddress unterscheidet
        System.out.println(" IP: " + adr.getHostAddress());
        #Namensaufl sung      URI —> IP-Adresse
        #final InetAddress adr2 = InetAddress.getByHost("185.26.156.88");
        #System.out.println("URL: " + adr2.getHostAddress());

        #Verbindung aufbauen
        try (final Socket socket = new Socket(adr, port)) #Sobald man es schafft ein Ob
        {
            #System.out.println("Gewonnen ==> Verbindung ge ffnet");
            final BufferedWriter writer
                = #BUffert einige Daten zusammen und dann schickt er mehrere Bytes e
                new BufferedWriter(
                    new OutputStreamWriter(socket.getOutputStream(), "utf8")); #
            writer.write(request);
            writer.flush(); #schickt alle bsiher gesammelten Bytes s

            final BufferedReader reader
                = new BufferedReader(
                    new InputStreamReader(socket.getInputStream(), "utf8"));

            String line;
            final StringBuilder builder = new StringBuilder();
            while((line = reader.readLine()) != null)
                builder.append(line).append("\n");

            return builder.toString();

            #System.out.println(" Antwort: " + reader.readLine());
        }
        #System.out.println("==> Verbindung geschlossen");
        #socket.close(); nicht verwenden, da Autocloseable besser ist
    }

    public static void main(String[] args)
    {
        try
        {
            final String antwort =

```

```

        SimpleClient.sendRequestAndRecieveResponse("127.0.0.1", 4711, "GET 7")
        System.out.println(antwort);
    }
    catch (Exception ex)
    {
        ex.printStackTrace();
    }
}
}

```

#### 5.6.4 Server-Programm

```

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketTimeoutException;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

```

#4 Schritte des Servers

#horchen, anfrage lesen, antwort generieren, antwort versenden

```

public abstract class SimpleServer
{
    private final int port;
    private ServerSocket serverSocket = null;
    private HandleRequestThread handleRequestThread = null; #in JAVA ein Objekt = null
    private final ExecutorService exe = Executors.newWorkStealingPool(); #Dienst dem
                                                                    #mit runnable oder callable kann man ebenso e

    public SimpleServer(int port)
    {
        this.port = port;
    }
    # Ports gehen von 2^16 = 65365
    # Port 80 —> www (http(s))
    # >1064 reservierte Ports
    #Warten auf eine TCP-Verbindung
    private Socket listen() throws Exception
    {
        # .accept() —> wenn client versucht verbindung herzustellen, dann nimm an und
        return serverSocket.accept(); #serverSocket bekommt Port übergeben
    }

    #Lesen des Requests vom Client
    private String readRequest(Socket socket) throws IOException

```

```

{
    final BufferedReader reader = new BufferedReader( #Reader(char) von InputStrea
        new InputStreamReader (socket.getInputStream(), "utf8"));
    return reader.readLine(); #erste Zeile machen
}

#Berechnen des Response aus dem Request nur zu Testzwecken
protected abstract String createResponse(String request);

/*{
    return "Die Anfrage war: " + request; #"Echo Sever"
}*/

#Response zum Client retour senden
private void sendResponse(Socket socket, String response)
    throws Exception
{
    final BufferedWriter writer = new BufferedWriter( # OutputStream(bytes) schreib
new OutputStreamWriter(socket.getOutputStream(), "utf8"));
    writer.write(response);
    writer.flush(); #geht ins www ( wird abgeschickt)
    socket.shutdownOutput();
}

private void handleRequest()
    throws Exception
{
    final Socket socket = listen(); #4 Schritte des Servers
    exe.submit( () -> #Executor ausführen lassen mittels submit
        #Callable ist mit Exception, Runnable ohne und submit braucht
        #Lambda vereinfacht und vermeidet Boilerplatecode
    {
        final String
            request = readRequest(socket),
            response = createResponse(request);
            sendResponse(socket, response);
            return null;
        }
    );
}

public void start() throws Exception
{
    if (serverSocket == null)
    {
        serverSocket = new ServerSocket(port);
        serverSocket.setSoTimeout(1000); #ich warte max. 1000ms im accept
        handleRequestThread = new HandleRequestThread(); #Nur Thread->Objekt erzeugt und
        handleRequestThread.start(); #Fork -> startet zweiten Thread

    }
}

```

```

public void stop() throws Exception
{
    if (serverSocket != null)
    {
        handleRequestThread.interrupt(); #Aufforderung sich zu beenden
        handleRequestThread.join(); #Zeit ablaufen lassen bis das interrupt fertig ist
        serverSocket.close();
        serverSocket = null;
    }
}
private class HandleRequestThread extends Thread
{
    @Override
    public void run()
    {
        while (!isInterrupted()) # schaut ob eine Bitte ansteht sich zu schließen (
        {
            try
            {
                handleRequest();
            }
            catch (SocketTimeoutException ignore) {}
            catch (Exception ex)
            {
                ex.printStackTrace();
            }
        }
    }
}

```

## 5.7 Socket

## 5.8 Dienste

- **www** http(s)
- **E-Mail** smtp, pop3, imap
- **SSH** auf ein Terminal auf entferntem Rechner zugreifen
- **Telnet** gleich wie SSH nur unverschlüsselt
- **FTP(s)** Files transferieren

## 5.9 Port

Den Port kann man mit der Türnummer oder der Stiege bei einer Adresse vergleichen.

**Beispiel:** 1.2.3.4:80

In diesem Beispiel ist die Nummer 80 die Portnummer dieser IP-Adresse.



## 5.10 Codierung

### 5.10.1 Codieren

- Umwandlung von Zeichen in Bytes = von Textdatei zu Binärdatei

#### Decodieren

- Umwandlung von Bytes in Zeichen = von Binärdatei zu Textdatei

#### Dateien

Abfolge von Bits und Bytes auf Datenträger. Besteht aus Pfad, Name und Extension.

#### Dateisystem

Ist das Inhaltsverzeichnis der Dateien (FAT, FAT32, NTFS, ExFAT)

#### Streams

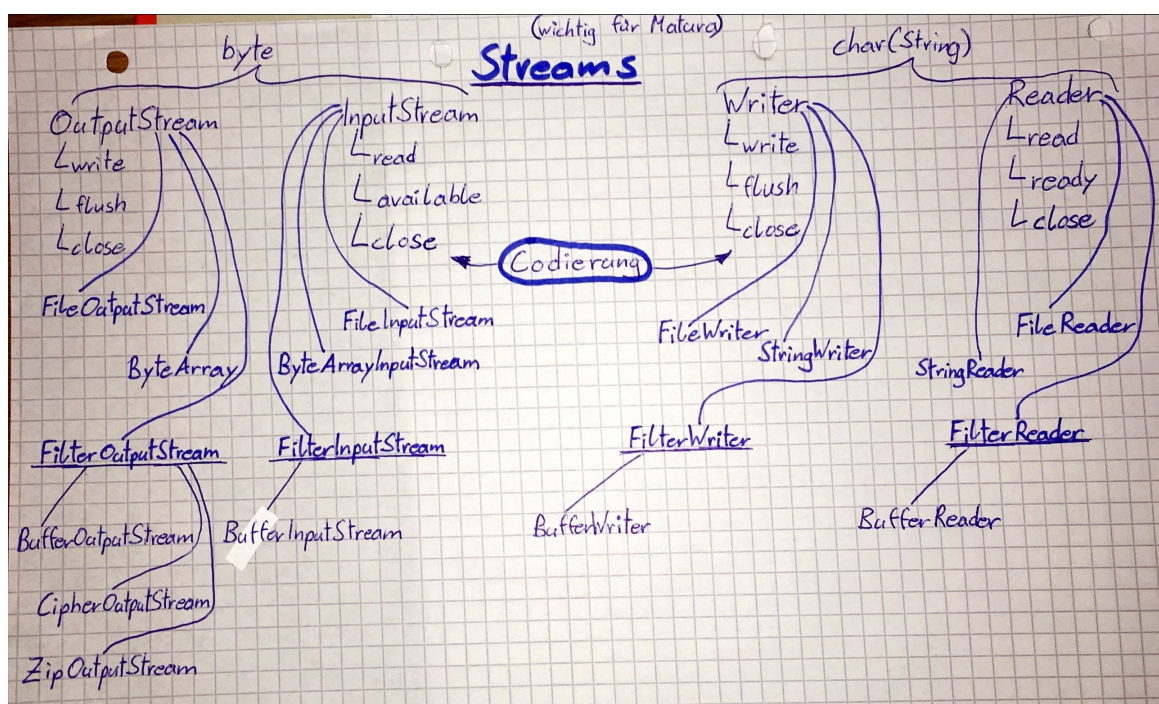


Abbildung 5.4: Arten von Streams

#### Arten von Streams

**OutputStream (Klasse) Methoden:**



- write
- flush
- close

**InputStream (Klasse) Methoden:**

- read
- available
- close

**Nutzen** Mithilfe von OutputStream und InputStream kann man Daten des Typ BYTE lesen.

**Writer (Klasse)**

**Reader (Klasse)**

**Nutzen** Mithilfe von Writer und Reader kann man Daten des Typs char lesen.

## 5.11 Protokolle zur Anwendung in der AT / Mech / IT

- Einfache Textprotokolle
- Modbus over IP
- JSON

# Datenschutz / Security

## 6.1 Datensicherheit

### 6.1.1 Hash-Funktionen

Der Zweck einer Hash-Funktion ist, aus einem Set an Daten einen „**Fingerabdruck**“ zu erstellen. Dieser Fingerabdruck wird meist **Hash** oder **Message Digest** genannt.

Jede Hash-Funktion hat folgende Charakteristiken:

- Die Länge des erzeugten Hashes hängt nicht von den Input-Daten, sondern nur vom verwendeten Algorithmus ab
- Der gleiche Algorithmus erzeugt bei den gleichen Daten immer einen gleichen Hash
- Die Änderung eines einzigen Bits der Daten führt zu einem völlig anderen Hash
- Es ist extrem schwierig die Daten zu berechnen aus denen ein Hash erzeugt wurde. Die Rechendauer nimmt dabei exponentiell mit der Länge des Hashes zu.

Da die Größe eines Hashes meist kleiner ist als die Größe der Eingangsdaten gehen Informationen verloren (die Entropie steigt). Dadurch gibt es theoretische mehrere Datensets, die den gleichen Hash erzeugen. Die Wahrscheinlichkeit so ein Datenset zu finden ist jedoch klein genug um ignoriert werden zu können.

Aus diesen Eigenschaften ergeben sich zwei Haupteinsatzgebiete für Hashes:

#### 1. Vergleichen von Daten

Lädt man z.B. große Dateien aus dem Internet herunter gibt es ein Risiko, dass Bitflips auftreten und die Datei somit nicht mehr dem Original entspricht. Das kann dazu führen, dass die Datei entweder direkt unbrauchbar ist, oder, im schlimmsten Fall, erst später Probleme verursacht.

Aus diesem Grund kann man in den meisten Fällen auf Downloadseiten (z.B. hier) eine Prüfsumme / einen Hash der Datei finden. Berechnet man nun die Prüfsumme der lokalen Datei mit dem gleichen Algorithmus wie den angegebenen Hash kann man überprüfen ob die Datei exakt dem Original entspricht.

#### 2. Speicherung von Passwörtern

Da jedes Gerät, das mit dem Internet verbunden ist hackbar ist, sollte man sensible Daten nie als Klartext speichern. Hat man einen Account auf einem Server wird beim Einloggen nicht das Passwort an den Server gesendet, sondern nur der Hash des Passworts. Stimmt der Hash überein weiß der Server, dass das Passwort auch übereinstimmen muss ohne das Passwort zu kennen.

Dadurch, dass ein gleiches Passwort immer zum gleichen Hash führt weiß man, dass das Passwort „12Passwort“ immer zum gleichen Hash führt. Und wenn man diesen Hash findet weiß man automatisch das Passwort dieses Benutzers und kann darauf hoffen, dass es auch für andere Dienste verwendet wird. Es gibt riesige Datenbanken, in denen für verschiedene Algorithmen die Hashes von oft verwendeten Passwörtern berechnet und gespeichert werden. Man sollte also längere und kompliziertere Passwörter verwenden, da die Wahrscheinlichkeit, dass diese bereits in einer solchen Datenbank vorhanden sind kleiner ist. Und vor allem sollte man nicht immer das gleiche Passwort verwenden.

Relevante Videos: Hashing, Hash databases

Die bekanntesten Hash-Algorithmen sind **MD5** und die **SHA**-Algorithmen.

- **MD5** steht für Message Digest version 5 und ist einer der ersten Hash-Algorithmen. Er erzeugt einen 32-bit Hash und es wurden bereits Methoden gefunden, den Algorithmus umzukehren. Er sollte also nicht mehr für sicherheitskritische Anwendungen verwendet werden.
- **SHA**, oder Secure Hash Algorithm, ist eine Gruppe von mehreren Algorithmen wie **SHA-128**, **SHA-256** oder **SHA-512**. Diese erzeugen jeweils 128-, 256- und 512-Bit Hashes. Dieser Algorithmus ist nach wie vor sicher und Stand der Technik.

### Implementierung in Java

```
import java.security.MessageDigest;

public class Allt {
    public String hash (String input) {

        MessageDigest md = MessageDigest.getInstance("SHA-512");

        // digest() method is called to calculate the message
        // digest of the input string returned as an array of bytes
        byte[] messageDigest = md.digest(input.getBytes());

        String hex = "";
        for (byte i : byteArray) {
            hex += String.format("%02X", i);
        }

        return hex;
    }
}
```

Die Methode `digest()` kann, je nach Länge des Inputs, längere Zeit dauern. Es ist also in vielen Anwendungen ratsam diese in einen eigenen Thread zu verlagern.

### 6.1.2 Cyclic Redundancy Check (CRC)

Ein CRC ist, wie ein Message Digest, ein Algorithmus, der eine beliebig große Menge an Bytes auf eine kleinere Prüfsumme reduziert. Anders als ein Message Digest, ist der CRC jedoch darauf ausgelegt **Fehler bei der Datenübertragung zu erkennen**. Dafür wird vor dem Senden die Prüfsumme der Nachricht

berechnet und an die Nachricht angehängt. Der Empfänger berechnet dann die Prüfsumme der empfangenen Daten und vergleicht sie mit der mitgesendeten.

Aufgrund mathematischer Tricks kann man garantieren, dass eine unterschiedliche Nachricht (bis zu einer bestimmten Größe) immer eine andere Prüfsumme erzeugt und damit sogenannte **Hash Collisions** verhindert werden kann. Hash-Funktionen auf der anderen Seite sind darauf optimiert, einen möglichst kleinen **Output Bias** zu haben. Das heißt, dass man vom erzeugten Hash wenig bis kaum Informationen über die ursprüngliche Nachricht ablesen kann.

Die meist verwendeten Algorithmen sind **CRC16** und **CRC32**.

## 6.2 Verschlüsselung

### 6.2.1 Symmetrische Verschlüsselung

Die symmetrische Verschlüsselung ist die einfachste Form der Kryptografie. Dabei verwendet man zum Verschlüsseln und Entschlüsseln der Nachricht den gleichen Schlüssel. Eine der einfachsten Algorithmen ist die **Caesar-Verschlüsselung**. Dabei verschiebt man alle Buchstaben der Nachricht um eine bestimmte Anzahl an Stellen im Alphabet. Diese Anzahl ist dabei der Schlüssel, den man benötigt um die Nachricht zu entschlüsseln.

Der große Nachteil der symmetrischen Verschlüsselung ist der **Austausch des Schlüssels**. Zu Beginn ist die Kommunikation noch unverschlüsselt, sodass man den Schlüssel beim erstmaligen Austausch abfangen kann.

### 6.2.2 Asymmetrische Verschlüsselung

Diese Art der Verschlüsselung verwendet eine sogenannte **Einwegfunktion**. Realisiert wird das oft mit der Multiplikation von Primzahlen. 3259 und 5431 zu multiplizieren ist kein Problem für einen Computer. Herauszufinden aus welchen Faktoren das Ergebnis 17699629 ist jedoch sehr zeitaufwändig. In der Praxis nutzt man Zahlen mit mehreren hundert Stellen, deren Primfaktorzerlegung mehrere Jahre dauern würde.

Die asymmetrische Verschlüsselung besteht immer aus einem **privaten Schlüssel** und einem **öffentlichen Schlüssel**. Damit ist es möglich eine Nachricht mit dem öffentlichen Schlüssel zu verschlüsseln, die nur mehr mit dem privaten Schlüssel entschlüsselbar ist.