Informatik-GK

Unterrichtsskript

Thomas Klein

Januar 2024

Inhaltsverzeichnis

I Einführung in die Informatik 6

1 Grundlagen der Informatik 7

1.1 Was ist Informatik? 7

1.2 Bits und Bytes 9

1.3 Die hexadezimale Schreibweise für Bytes 11

1.4 Der ASCII-Code 12

1.5 Digitalisierung 13

2 HTML 14

2.1 Aufbau einer Webseite 14

2.2 Formatierung von Text 17

2.3 Tags mit Attributen 17

2.4 Listen und Aufzählungen 18

2.5 Tabellen 21

2.6 Bilder 22

3 CSS 24

3.1 CSS-Dateien verlinken 24

3.2 HTML-Elemente stylen 24

3.3 CSS-Klassen 26

3.4 Das Box-Modell 27

3.5 `div`- und `span`-Elemente 28

3.6 Verschachtelte CSS-Selektoren 29

3.7 Layout mit CSS-Grid 30

4 Internetprotokolle 35

4.1 Rechnernetze und Kommunikationsprotokolle 35

4.2 Die TCP/IP-Protokollfamilie 36

4.2.1 Das Internet Protocol und der Domain Name Service 36

4.2.2 Das Transfer Control Protocol 37

4.2.3 Das TCP-IP-Referenzmodell 37

4.3 Das Client-Server-Modell 38

4.4 Protokolle der Anwendungsschicht 38

II Einführung in die Programmierung 40

1 Schnelleinstieg in Java 41

1.1 Hallo Welt 41

1.2 Das User-Interface (UI) 42

1.3 Die `onAction`-Methode 43

2 EVA 45

2.1 Datentypen 45

2.2 Variablen 46

2.3 Methoden und Anweisungen 48

3 Methoden 50

3.1 Parameter und Rückgabewerte 50

3.2 Darstellung als Struktogramm 53

4 if-else 54

5 Arrays 58

5.1 Array, Länge, Index 59

5.2 Deklarieren und Initialisieren von Array-Variablen 60

5.3 Zugriff auf die Elemente und die Länge eines Arrays 61

6 Schleifen 63

6.1 Die `FOR`-Schleife 64

6.2 Die `WHILE`-Schleife 66

6.3 `break` und `continue` 67

III Algorithmik und Objektorientierung 68

1 Klassen und Objekte 69

1.1 Klassen als zusammengesetzte Datentypen 69

1.2 Deklaration einer Klasse 71

1.3 Objekte einer Klasse erzeugen und verwenden 71

2 Der Konstruktor einer Klasse 73

2.1 Was ist ein Konstruktor? 73

2.2 Das Schlüsselwort `this` 74

3 Verwalten von Objekten in Arrays 76
3.1 Arrays von Objekten 76
3.2 Hinzufügen neuer Objekte zu einem Array 77
3.3 Entfernen von Objekten aus einem Array 77
3.4 Zufällige Auswahl in einem Array 79

4 Klassen mit Methoden 80

5 Das Geheimnisprinzip 83
5.1 Die Schlüsselwörter `private` und `public` 83
5.2 Getter- und Setter-Methoden 84
5.3 Der Sinn hinter dem Geheimnisprinzip 85
5.3.1 Ausgangslage: Ein Routenplaner 85
5.3.2 Anwendung des Geheimnisprinzips 86
5.3.3 Lösungsmöglichkeit 1: Getter-Methoden 87
5.3.4 Lösungsmöglichkeit 2: Die Funktionalität verlagern 87
5.3.5 Vergleich der beiden Lösungsmöglichkeiten 88

6 Modellieren mit UML 89
6.1 Darstellung von Klassen in UML 89
6.2 Assoziationen und Aggregationen 89
6.3 Multiplizitäten 91

7 Suchalgorithmen 94
7.1 Die sequentielle Suche 94
7.2 Die binäre Suche 95

8 Sortieralgorithmen 97

9 Die Laufzeit eines Algorithmus 100

10 Rekursion 103

11 Rekursion vs. Iteration 108

IV Datenbanken 110

1 Entitäten und Relationen 112
1.1 Daten und Informationen 112
1.2 Das ER-Modell 113
1.3 ER-Diagramme 114
1.4 Kardinalitäten und Optionalitäten 114

2 Das Relationenmodell 116
2.1 Relationen und Schlüssel 116
2.2 Überführen des ERD in das Relationenmodell 117

3 Abfragen mit SQL 120
3.1 Der SELECT-Befehl 120
3.2 Formeln und Aggregatfunktionen 121
3.3 Sortierung und Limitierung 122
3.4 Gruppieren 123
3.5 Joins 124

4 Relationale Algebra 127
4.1 Projektionen und Selektionen 127
4.2 Kreuzprodukt und Join 129

5 Datenschutz und Datensicherheit 130
5.1 Datenschutz 130
5.2 Datensicherheit 133

V Theoretische Informatik 134

1 Formale Sprachen 136
1.1 Alphabet, Wort, Sprache, Grammatik 136
1.2 Reguläre und kontextfreie Sprachen 139
1.3 Syntax-Diagramme 140

2 Endliche Automaten 142
2.1 Akzeptoren 142
2.2 Schreibende Automaten (Mealy-Automaten) 144
2.3 Beschreibung realer Automaten 145
2.4 Reguläre Ausdrücke 145

3 Berechenbarkeit 148
3.1 Computer, Probleme, Algorithmen und Berechenbarkeit 148
3.2 Turingmaschinen 149
3.3 Die Hypothese von Church 150
3.4 Die universelle Turingmaschine 151
3.5 Probleme als partielle Funktionen 152
3.6 Nicht-berechenbare Probleme 154
3.7 Das Halteproblem 156

4 Grundlagen der Komplexitätstheorie 158
4.1 Laufzeit und O-Notation 158
4.2 Klassifizierung von Problemen 162
4.3 P=NP? 163

I

Einführung in die Informatik

1 Grundlagen der Informatik

In diesem einführenden Kapitel besprechen wir zunächst die Grundlagen der Informatik, deren Kenntnis zwar nicht explizit im Kerncurriculum gefordert wird, aber absolut notwendig ist, um die Inhalte des Unterrichts verstehen zu können. Wir klären, was ein Computer eigentlich ist, wie er (grob) funktioniert, wie Daten im Computer gespeichert und bearbeitet werden und weitere grundlegende Dinge.

1.1 Was ist Informatik?

Es gibt verschiedene Definitionen der Wissenschaft Informatik. Wir verwenden die folgende

Definition 1.1 Informatik *ist die Wissenschaft von der automatischen und systematischen Verarbeitung und Darstellung von Daten zur Gewinnung von Informationen, insbesondere unter dem Einsatz von Computern.*

Bemerkung 1.2 Informatik hat also nur am Rande mit Computern zu tun; man kann durchaus Informatik betreiben ohne jeglichen Computereinsatz.[1] #

Nun bedürfen die Begriffe »Daten«, »Informationen« und »Computer« noch einer Definition. Beginnen wir mit der Unterscheidung zwischen Daten und Informationen:

Definition 1.3 *Unter* **Daten** *(Singular:* **Datum***) versteht man verschiedene Symbole wie Buchstaben und Ziffern.*

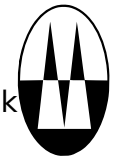
Damit aus Daten **Informationen** *werden, muss man wissen, was diese Symbole bedeuten sollen.*

Beispiel 1.4 Die Zahl 19821213 ist ein Datum, aber keine Information, wenn man nicht weiß, wie die Zahl zu interpretieren ist. Die Zahl wird zu einer Information, wenn man weiß, dass es sich dabei um die Angabe eines Geburtstags handelt, nämlich der 13.12.1982. #

Es bleibt die große Frage, was eigentlich ein Computer sein soll:

Definition 1.5 *Wir bezeichnen etwas als* **Computer***, wenn es beliebige* **Programme** *ausführen kann, die veränderbar sind.*

[1] Es macht nur weniger Spaß :)



Beispiel 1.6 Ein Computer zeichnet sich also dadurch aus, dass sein Einsatzgebiet jederzeit durch eine Änderung der Software verändert werden kann. So kann ein Computer Bilder verarbeiten, Musikstücke abspielen, Maschinen steuern oder gegen einen Menschen Schach spielen wenn ein entsprechendes Programm vorhanden ist.

In diesem Sinne sind also Toaster, Wecker, Waschmaschinen und Festnetz-Telefone keine Computer, da sie nur genau die Aufgabe erfüllen können, für die sie gebaut wurden. Es ist nicht möglich, einer Waschmaschine Schachspielen beizubringen.[2] #

Nun stellt sich die neue Frage, was ein »Programm« sein soll:

Definition 1.7 *Ein* **Programm** *oder* **Algorithmus** *ist eine Abfolge von klar definierten* **Anweisungen***, die nacheinander ausgeführt werden und aus einer* **Eingabe** *Schritt für Schritt eine* **Ausgabe** *generieren (***EVA-Prinzip***: Eingabe-Verarbeitung-Ausgabe).*

Beispiel 1.8 Der folgende Algorithmus untersucht, ob in einer Gruppe von Personen ein Arzt anwesend ist.

Eingabe: Eine Gruppe von Personen.

Algorithmus:

(1) Gehe zur ersten Person.

(2) Frage die aktuelle Person, ob sie Arzt ist.

(3) Falls die aktuelle Person mit »Ja« antwortet:

- Ausgabe: Es gibt einen Arzt.
- Beende das Programm.

(4) Falls es noch weitere Personen gibt:

- Gehe zur nächsten Person.
- Gehe zurück zu Schritt (2).

Ansonsten:

- Ausgabe: Es gibt keinen Arzt.
- Beende das Programm.

#

Bemerkung 1.9 Die Programme müssen in einer Sprache aufgeschrieben werden, die der Computer versteht. Solche Sprachen heißen **Programmiersprachen**. #

[2] Zumindest gilt dies in der Theorie. In der Praxis sind in fast allen der genannten Geräte Computer eingebaut, die beispielsweise das Waschprogramm steuern. Der Hintergrund hierfür ist, dass Computer unglaublich billig sind und daher auch in solchen Geräten verbaut werden. Ich persönlich gehe davon aus, dass meine WLAN-fähige Waschmaschine zu Hause durchaus umprogrammiert werden kann und dann eine Partie Schach gegen mich spielen könnte...

1.2 Bits und Bytes

Ein Computer ist eine elektrische Maschine, in der mit rasender Geschwindigkeit elektrische Signale kreisen: Strom an - Strom aus - Strom an - Strom aus - Strom aus - Strom an - .... Ein **Taktgeber** sorgt dafür, dass man zwischen »Strom an« und »3 mal Strom an« unterscheiden kann. Die **Taktfrequenz** wird in der Einheit **Hertz (Hz)** gemessen.[3]

Ein Computer ist also in der Lage, zwischen zwei Zuständen zu unterscheiden: Strom an (1) und Strom aus (0).

Definition 1.10 *Ein* **Bit** *ist eine einzelne 0 oder 1. Es ist die kleinst-mögliche Informationseinheit.*

Beispiel 1.11 Mit Bit-Folgen kann man alles mögliche **kodieren**:

Z.B. könnte die Bitfolge 0001110010101101010 für eine Wegbeschreibung stehen: 00 bedeutet »geradeaus«, 01 bedeutet »nach links abbiegen«, 10 bedeutet »nach rechts abbiegen«, 11 bedeutet »U-Turn«. #

Ein Bit kommt jedoch niemals alleine:

Definition 1.12 *Ein* **Byte** *besteht aus 8 Bits. Alle Daten in Computern werden in Byte-Päckchen verpackt, d.h., es werden niemals einzelne Bits verarbeitet/gespeichert sondern immer ganze Bytes.*

Bemerkung 1.13 Wie bei allen anderen Einheiten auch, können auch Bytes und Taktzahlen mit den Vorsilben

k (Kilo = Tausend): ×1.000

M (Mega = Million): ×1.000.000

G (Giga = Milliarde): ×1.000.000.000

T (Tera = Billion): ×1.000.000.000.000

versehen werden. In einem Smartphone mit einer Taktfrequenz von 2 GHz kreisen also 2 Milliarden Stromsignale pro Sekunde! Eine externe Festplatte mit einem Volumen von 1 TB kann 1 Billion Bytes speichern, also 8.000.000.000.000 Nullen und Einsen! #

Um es noch einmal klarzustellen: Der Computer kann nur mit Bytes umgehen. Alles, was sich nicht als Folge von Bytes darstellen lässt, kann vom Computer nicht bearbeitet

[3] $1\,\mathrm{Hz} = \frac{1}{\mathrm{s}}$

werden. Natürlich soll ein Computer aber auch mit Zahlen und Buchstaben umgehen können:

Definition 1.14 *Eine Zahl im* **Binärsystem** *besteht nur aus den Ziffern* 0 *und* 1*. Diese Ziffern reichen aus, um jede beliebige natürliche Zahl darzustellen:*

1 =	*1*	*6 =*	*110*	*11 =*	*1011*	*16 =*	*10000*
2 =	*10*	*7 =*	*111*	*12 =*	*1100*	*17 =*	*10001*
3 =	*11*	*8 =*	*1000*	*13 =*	*1101*	*18 =*	*10010*
4 =	*100*	*9 =*	*1001*	*14 =*	*1110*	*19 =*	*10011*
5 =	*101*	*10 =*	*1010*	*15 =*	*1111*	*20 =*	*10100*

Das Binärsystem ist genauso ein Stellenwertsystem wie das Dezimalsystem, nur dass hier die Stufenzahlen Potenzen der Zahl 2 *sind anstelle von Potenzen der Zahl* 10*, also* $1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, \ldots$.

Beispiel 1.15 Wir wandeln die Dezimalzahl 3853 in das Binärsystem um, indem wir sie als Summe der Stufenzahlen schreiben: Die größte Stufenzahl, die hineinpasst ist die 2048, d.h.

$$\begin{aligned} 3853 &= 2048 + 1805 \\ &= 2048 + 1024 + 781 \\ &= 2048 + 1024 + 512 + 269 \\ &= 2048 + 1024 + 512 + 256 + 13 \\ &= 2048 + 1024 + 512 + 256 + 8 + 4 + 1 \end{aligned}$$

Also gilt

$$\begin{aligned} 3853 = \underline{1}\cdot 2048 + \underline{1}\cdot 1024 + \underline{1}\cdot 512 + \underline{1}\cdot 256 + \underline{0}\cdot 128 \\ + \underline{0}\cdot 64 + \underline{0}\cdot 32 + \underline{0}\cdot 16 + \underline{1}\cdot 8 + \underline{1}\cdot 4 + \underline{0}\cdot 2 + \underline{1}\cdot 1 \end{aligned}$$

Damit ergibt sich die Binärdarstellung $111100001101_{(2)}$. Man benötigt also zwei Bytes, um die Zahl 3853 darzustellen. Fehlende Ziffern werden mit führenden Nullen ergänzt: 00001111 00001101. #

1.3 Die hexadezimale Schreibweise für Bytes

Die binäre Schreibweise von Bytes hat einen großen Nachteil: Acht Ziffern sind für uns Menschen nur schwer »auf einen Blick« zu erfassen. Es ist z.B. äußerst mühselig, mehrere Nullen hintereinander zu zählen.

Deshalb hat sich die hexadezimale Schreibweise für Bytes eingebürgert:

Definition 1.16 *Im* **Hexadezimalsystem** *stehen für die Zahlen* 16 *verschiedene Ziffern zur Verfügung:*

0 1 2 3 4 5 6 7 8 9 A B C D E F

Jede dieser Ziffern entspricht einer Binärzahl aus 4 *Bits:*

Hexadezimal	4 Bits	Hexadezimal	4 Bits
0	0000	8	1000
1	0001	9	1001
2	0010	A (= 10)	1010
3	0011	B (= 11)	1011
4	0100	C (= 12)	1100
5	0101	D (= 13)	1101
6	0110	E (= 14)	1110
7	0111	F (= 15)	1111

Da man also jeweils 4 *Bits zu einer hexadezimalen Ziffer zusammenfassen kann, entspricht ein Byte einer zweistelligen Hexadezimalzahl. Um Verwechslungen mit dezimalen Zahlen zu verhindern, schreibt man vor eine Hexadezimalzahl* `0x`*, also z.B.* `0x2B` *oder* `0x10`*.*

Beispiel 1.17

(a) `0101 0011` = `0x53`

(b) `1110 1011` = `0xEC`

(c) `0000 1010` = `0x0A` #

1.4 Der ASCII-Code

Mit Bytes lassen sich auch Buchstaben und andere Zeichen kodieren. Der grundlegendste Code ist der sog. »ASCII«-Code:

Definition 1.18 *Der* **ASCII-Code** *(*American Standard Code for Information Interchange*) ordnet verschiedenen Symbolen wie Buchstaben, Ziffern und Sonderzeichen jeweils ein Byte zu.* **Tabelle 1** *zeigt die Zeichen des ASCII-Codes, wobei die Nummerierung erst ab* `0x20` *beginnt, da die Zeichen davor besondere Steuerzeichen sind, die unsichtbar sind.*

Hex	Zeichen	Hex	Zeichen	Hex	Zeichen	Hex	Zeichen	Hex	Zeichen
0x20		0x33	3	0x46	F	0x59	Y	0x6C	l
0x21	!	0x34	4	0x47	G	0x5A	Z	0x6D	m
0x22	"	0x35	5	0x48	H	0x5B	[	0x6E	n
0x23	#	0x36	6	0x49	I	0x5C	\	0x6F	o
0x24	$	0x37	7	0x4A	J	0x5D	]	0x70	p
0x25	%	0x38	8	0x4B	K	0x5E	^	0x71	q
0x26	&	0x39	9	0x4C	L	0x5F	_	0x72	r
0x27	'	0x3A	:	0x4D	M	0x60	`	0x73	s
0x28	(	0x3B	;	0x4E	N	0x61	a	0x74	t
0x29	)	0x3C	<	0x4F	O	0x62	b	0x75	u
0x2A	*	0x3D	=	0x50	P	0x63	c	0x76	v
0x2B	+	0x3E	>	0x51	Q	0x64	d	0x77	w
0x2C	,	0x3F	?	0x52	R	0x65	e	0x78	x
0x2D	-	0x40	@	0x53	S	0x66	f	0x79	y
0x2E	.	0x41	A	0x54	T	0x67	g	0x7A	z
0x2F	/	0x42	B	0x55	U	0x68	h	0x7B	{
0x30	0	0x43	C	0x56	V	0x69	i	0x7C	|
0x31	1	0x44	D	0x57	W	0x6A	j	0x7D	}
0x32	2	0x45	E	0x58	X	0x6B	k	0x7E	~

Abbildung 1 Die druckbaren Zeichen des ASCII-Codes.

Beispiel 1.19

(a) Ein `A` wird im ASCII-Code als das Byte `0x41 = 0100 0001` kodiert.

(b) Ein `+` sieht so aus: `0x2B = 0010 1011`

#

1.5 Digitalisierung

Am Ende dieser Einführung wollen wir kurz auf den Begriff »Digitalisierung« eingehen.

Definition 1.20 *Etwas zu* **digitalisieren** *bedeutet, dieses »Etwas« in Bytes, also in Nullen und Einsen darzustellen. Text kann beispielsweise über den ASCII-Code digitalisiert werden. Es ist aber auch möglich, Töne, Bilder und Videos zu digitalisieren.*

Beispiel 1.21 Das deutsche Grundgesetz ist ein Buch mit etwa 100 DIN-A5-Seiten. Vorsichtig geschätzt, besteht jede Seite aus ungefähr 30 Zeilen, die jeweils wiederum vielleicht aus 9 Wörtern bestehen, die wiederum im Schnitt 8 Zeichen lang sind (alles geschätzt).

Dies macht dann

- $9 \cdot 8 + 8 = 80$ Zeichen pro Zeile (Wörter + Leerzeichen zwischen den Wörtern),
- $80 \cdot 30 = 2.400$ Zeichen pro Seite,
- $2.400 \cdot 100 = 240.000$ Zeichen insgesamt.

Da im ASCII-Code jedes Zeichen 1 Byte benötigt, ergibt dies etwa $240.000\,\mathrm{B} = 240\,\mathrm{kB}$, um das Grundgesetz zu digitalisieren. #

2 HTML

HTML ist zwar offiziell die Abkürzung für »Hypertext markup language«, bekannter ist es aber als die Sprache des *World Wide Web*.

In diesem Kapitel befassen wir uns mit den Grundlagen dieser Computersprache und sehen, wie man grundlegende HTML-Elemente dazu benutzen kann, den Inhalt einer Webseite zu strukturieren und mehrere Webseiten miteinander zu verlinken.

Vorbemerkung: In diesem Skript wird eine Auswahl der wichtigsten Tags thematisiert. Es gibt natürlich noch sehr viel mehr Tags, die man in verschiedensten Internet-Dokumentationen recherchieren kann. Es gibt über 100 HTML-Tags!

2.1 Aufbau einer Webseite

Bevor wir zu den eigentlichen Webseiten kommen, schauen wir uns zunächst die grundlegende Einheit eines HTML-Dokuments an:

Definition 2.1 *Ein* **Tag** *hat stets die Form*

```
<tagname>Inhalt des Tags</tagname>
```

wobei ein Tag ohne Inhalt auch in der Form

```
<tagname/>
```

geschrieben werden kann.

Jeder Tag hat eine bestimmte **Semantik** *(Bedeutung): Ein* `table`*-Tag leitet z.B. eine Tabelle ein, ein* `strong`*-Tag hebt seinen Inhalt hervor und ein* `a`*-Element steht dagegen für einen Link zu einer anderen Webseite.*

Definition 2.2 *Eine* **Webseite** *ist ein Dokument, das in der Sprache* **HTML** *(»Hypertext Markup Language«) verfasst ist und dessen Dateiname die Erweiterung* `.html` *oder* `.htm` *besitzt.*

Ein solches HTML-Dokument beginnt mit der Zeile

```
<!doctype html>
```

und enthält im Anschluss eine Reihe ineinander verschachtelter Tags.

Der oberste Tag muss ein `html`*-Tag sein, der ausschließlich einen* `head`*-Tag und einen* `body`*-Tag (in dieser Reihenfolge) enthalten darf (aber nicht muss).*

```
1   <!doctype html>
2   <html>
3     <head>
4       Informationen über die Seite
5     </head>
6     <body>
7       Inhalt der Seite
8     </body>
9   </html>
```

Abbildung 2

Der `head`-Tag enthält Informationen über die Webseite (dazu kommen wir später), der `body`-Tag enthält den Inhalt der Webseite.

Für eine bessere Übersicht wird empfohlen, die Kind-Tags eines Tags um eine Ebene (1 »Tab«) weiter einzurücken.

Das daraus folgende Grundgerüst einer Webseite wird in **Abb. 2** *gezeigt.*

Information 2.3 Da jedes HTML-Dokument ein `html`-Element hat, das alle anderen Tags enthält, kann man sich jedes HTML-Dokument als Baum vorstellen (vgl. **Abb. 3**).

Um ein HTML-Dokument anzuzeigen, benötigt man eine spezielle Software:

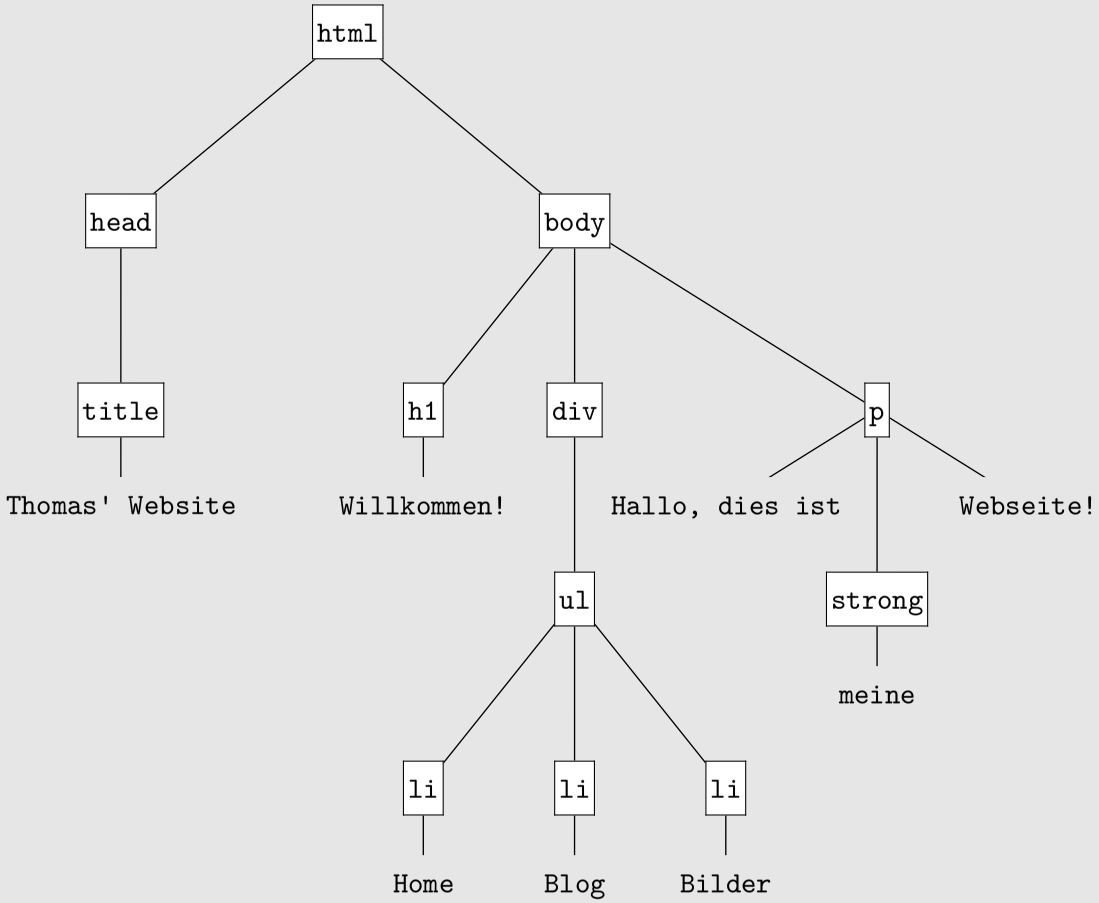
Definition 2.4 *Ein* **Browser** *ist eine Software, die (unter anderem) HTML-Dokumente darstellen kann. Beispiele für Browser sind Internet Explorer, Edge, Firefox, Chrome, Safari, Opera und viele mehr.*

Bemerkung 2.5 Eine Webseite ist ein HTML-Dokument, aber nicht jedes HTML-Dokument muss eine Webseite sein. HTML hat sich in den letzten Jahren vom Web teilweise emanzipiert und ist auf dem besten Wege dahin, zur universellen Beschreibungssprache für Medien aller Art (Präsentationen, Apps, Bücher) zu entwickeln. Die Mediatheken im Fernseher sind z. B. auch mit HTML geschrieben.

Der Grund hierfür liegt vor allem im Einzug mobiler Geräte wie Smartphones und Tablets. Ein PDF-Dokument legt das Aussehen seines Inhalts statisch fest, d.h. Eigenschaften wie die Seitenbreite und Schriftgröße sind festgelegt. Das erschwert das Lesen von PDFs auf Smartphones erheblich. HTML-Seiten dagegen werden vom Browser dargestellt und an die Bildschirmgröße angepasst. #

```
1   <!doctype html>
2   <html>
3     <head>
4       <title>Thomas' Website</title>
5     </head>
6     <body>
7       <div>
8         <ul>
9           <li>Home</li>
10          <li>Blog</li>
11          <li>Bilder</li>
12        </ul>
13      </div>
14      <h1>Willkommen!</h1>
15      <p>
16        Hallo, dies
17        ist <strong>meine</strong>
18        Webseite!
19      </p>
20    </body>
21  </html>
```

Eine typische Webseite.



Der Aufbau dieser Webseite als Baum.

Abbildung 3

2.2 Formatierung von Text

Auch wenn einige moderne Webseiten fast ausschließlich aus Bildern bestehen, ist der geschriebene Text noch immer das verbreitetste Medium zur Darstellung von Informationen.

Die Darstellung von Text ist zunächst sehr einfach:

Information 2.6 Sämtlicher Text, der im Body-Element eines HTML-Dokumentes steht, wird unformatiert angezeigt. Dabei beachtet der Browser weder Zeilenumbrüche noch mehrere Leerzeichen hintereinander. Der Text wird einfach als Fließtext dargestellt.

Um den Text zu formatieren, muss man bestimmte Tags verwenden:

Information 2.7 Die folgenden Tags dienen zur Formatierung von Text:

`p`	Ein Absatz.
`h1`	Eine Haupt-Überschrift.
`h2`	Eine Unter-Überschrift.
`h3-h6`	Immer kleinere Unter-Überschriften.
`strong`	Eine starke Hervorhebung, die dem Leser bereits beim ersten Anblick der Seite ins Auge fällt. Wird standardmäßig **fett** dargestellt.
`em`	Eine schwache Hervorhebung, die dem Leser erst auffällt, wenn er in die Nähe der entsprechenden Textstelle gerät. Wird standardmäßig *kursiv* dargestellt.

2.3 Tags mit Attributen

Jeder Tag kann mit sogenannten »Attributen« ausgestattet werden:

Definition 2.8 *Ein* **Attribut** *eines Tags ist eine zusätzliche Information, die der Autor an den Tag übermittelt.*

Attribute werden unmittelbar nach dem Tagnamen vor dem Schließen der spitzen Klammer in der Form `attribut="wert"` *angegeben. Es können beliebig viele Attribute verwendet werden.*

Beispiel 2.9 Im Code

```
1    <p style="color: blue" title="Dieser Text ist blau">Lorem ipsum dolor sit amet, consetetur sadipscing
2    elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.</p>
```

hat das `p`-Element die Attribute `style` und `title` erhalten. Ersteres legt mit Hilfe der Sprache CSS fest, wie das Element aussehen soll (hier: blaue Schriftfarbe), letzteres legt den sog. »Tooltip« fest, der erscheint, wenn man die Maus über dem Absatz stehen lässt.

#

Oft beinhalten Attribute nur zusätzliche Informationen, auf die der Tag auch verzichten könnte. Einige Tags funktionieren aber erst dann richtig, wenn man die Attribute richtig setzt:

Information 2.10

(a) Einen **Hyperlink** zu einer anderen Webseite setzt man mit Hilfe des `a`-Elements:

```
1    <a href="http://mathe-info.com" target="_blank">Link zur Webseite</a>
```

Das `href`-Attribut erhält die URL (Uniform Ressource Locator) der Webseite, die verlinkt werden soll. Das `target`-Attribut mit dem speziellen Wert `_blank` bewirkt, dass der Browser bei Klick des Links einen neuen Tab öffnet. Um im selben Tab zu bleiben, muss man das Attribut `target` entfernen.

(b) Mit Hilfe des `meta`-Tags kann man *im Head der Webseite* alle möglichen Arten von Meta-Informationen über die Webseite hinzufügen, z.B.

```
1    <meta name="language" content="DE"/>
```

sorgt dafür, dass Suchmaschinen erkennen, dass die Webseite in Deutsch verfasst ist.

(c) Der wichtigste `meta`-Tag ist aber der, der die Kodierung der Seite angibt. Standardmäßig geht der Browser davon aus, dass die Webseiten im ASCII-Code kodiert sind und deshalb fehlen z.B. deutsche Sonderzeichen wie ß, ä, ö oder ü. Mittels

```
1    <meta charset="utf-8"/>
```

legt man fest, dass die Webseite in UTF-8 kodiert ist. Diese Kodierung enthält alle gebräuchlichen Symbole.

2.4 Listen und Aufzählungen

Information 2.11 Eine **unnummerierte Liste** wird durch zwei Arten von Tags realisiert:

1. Frühstück
2. Mittagessen
3. Abendessen
4. Mitternachtssnack

Nummerierte Liste

```
1  <ol>
2    <li>Frühstück</li>
3    <li>Mittagessen</li>
4    <li>Abendessen</li>
5    <li>Mitternachtssnack</li>
6  </ol>
```

Zugehöriges Markup

Abbildung 4

(a) Das Listen-Mutter-Element `ul` (»unordered list«), das die gesamte Liste umschließt.

(b) Ein List-Item-Element `li`, das die einzelnen Unterpunkte der Liste enthält.

Dabei ist zu beachten, dass ein `li`-Element immer nur als direktes Kind eines `ul` (oder `ol`, siehe unten) vorkommen darf!
Außerdem dürfen nur `li`-Elemente direkte Kinder von `ul` (oder `ol`) sein.

Beispiel 2.12 Der Code

```
1  <ul>
2    <li>
3      Das Listen-Mutter-Element <code>ul</code>, das die gesamte Liste enthält.
4    </li>
5    <li>
6      Ein List-Item-Element <code>li</code>, das die einzelnen Unterpunkte der Liste enthält.
7    </li>
8  </ul>
```

wird in der Form

- Das Listen-Mutter-Element `ul`, das die gesamte Liste umschließt.
- Ein List-Item-Element `li`, das die einzelnen Unterpunkte der Liste enthält.

dargestellt. #

Information 2.13 Eine **nummerierte Liste** erhält man, indem man statt des `ul`-Tags den `ol`-Tag (»ordered List«) verwendet. Alles andere bleibt gleich (vgl. **Abb. 4**.).

- Montag
  - Zahnarzt
  - Team-Meeting
- Dienstag
  - keine Termine
- Mittwoch
  - Tennis
  - Kinder abholen
  - Spieleabend

Verschachtelte Liste

```
<ul>
  <li>
    Montag
    <ul>
      <li>Zahnarzt</li>
      <li>Team-Meeting</li>
    </ul>
  </li>
  <li>
    Dienstag
    <ul>
      <li>keine Termine</li>
    </ul>
  </li>
  <li>
    Mittwoch
    <ul>
      <li>Tennis</li>
      <li>Kinder abholen</li>
      <li>Spieleabend</li>
    </ul>
  </li>
</ul>
```

Zugehöriges Markup

Abbildung 5

Information 2.14 Manchmal enthalten Aufzählungen Unter-Aufzählungen. In diesem Fall wird die Unter-Aufzählung vollständig von dem entsprechenden `li`-Tag der Ober-Aufzählung umschlossen (vgl. **Abb. 5**.)

Bemerkung 2.15 Dies ist oft die erste Stelle, an der folgendes Phänomen auftritt: Man schreibt HTML-Code, der laut verschiedener Quellen (z.B. diesem Skript) fehlerhaft ist, z.B.

```
<ul>
  <li>Deutschland</li>
  <ul>
```

```
4        <li>Berlin</li>
5        <li>München</li>
6        <li>Hamburg</li>
7      </ul>
8    </ul>
```

Trotzdem zeigt der Browser die Seite korrekt an.

Der Grund hierfür liegt in der enormen *Fehlertoleranz* der Browser. Da Webseiten (in den meisten Fällen) nicht von Programmierern erstellt werden, wäre es fatal, wenn die Browser beim kleinsten Fehler ihre Tätigkeit einstellen würden. Stattdessen versuchen sie, jeden auch noch so fehlerhaften Quelltext (unvollständige Tags, falsch geschachtelte Tags, ...) sinnvoll zu interpretieren und darzustellen.

Man kann seine eigene Webseite auf Fehler überprüfen lassen, z.B. unter `validator.w3.org` . #

2.5 Tabellen

Mit Tabellen lassen sich Daten oft übersichtlich darstellen:

Information 2.16 Das Markup einer **Tabelle** benötigt drei bzw. vier Tags:

(a) Das `table`-Element, das als Mutter-Element die gesamte Tabelle umschließt.

(b) `tr`-Elemente (»table row«) sind die Zeilen der Tabelle.

(c) `td`-Elemente (»table data«) sind die einzelnen Tabellenzellen innerhalb einer Zeile.

(d) Für Spalten-Überschriften sollten anstelle von `td`-Elementen `th`-Elemente (»table heading«) verwendet werden.

Vgl. **Abb. 6** für ein vollständiges Beispiel.

Bemerkung 2.17 Die ersten HTML-Tabellen sehen nicht unbedingt so aus, wie man sich eine Tabelle vorstellen würde: Es fehlen die Linien zwischen den Zeilen und Spalten, die Ausrichtung (linksbündig vs. rechtsbündig vs. zentriert) ist unschön usw.

Es gab in alten HTML-Versionen bestimmte Tags und Attribute, die es erlaubten, diese Eigenschaften anzupassen, was heute jedoch nicht mehr gemacht wird. Stattdessen wird die universelle Stil-Beschreibungssprache CSS verwendet.

Für den Moment müssen wir also damit leben, dass unsere Tabellen eher unschön aussehen. #

Land	Einwohner-zahl	Hauptstadt
Deutschland	80 Mio.	Berlin
Frankreich	67 Mio.	Paris

Tabelle mit Überschriften

```
<table>
  <tr>
    <th>Land<th>
    <th>Einwohnerzahl</th>
    <th>Hauptstadt</th>
  </tr>
  <tr>
    <td>Deutschland<td>
    <td>80 Mio.</td>
    <td>Berlin</td>
  </tr>
  <tr>
    <td>Frankreich<td>
    <td>67 Mio.</td>
    <td>Paris</td>
  </tr>
</table>
```

Zugehöriges Markup

Abbildung 6

2.6 Bilder

Keine Webseite ohne Bilder:

Information 2.18 Ein **Bild** wird über den `img`-Tag eingebunden:

```
<img src="dateiname" alt="Anzeigetext" width="breite" height="hoehe"/>
```

Das `src`-Attribut legt fest, welches Bild geladen werden soll und muss unbedingt angegeben werden.

Das `alt`-Attribut legt fest, welcher Text angezeigt werden soll, wenn das Bild aus irgendwelchen Gründen nicht angezeigt werden kann (z.B., wenn im Browser Bilder deaktiviert sind, um Bandbreite zu sparen oder wenn das Bild vom Server gelöscht wurde).

Die Attribute `width` und `height` legen die Breite und die Höhe des Bildes in Pixeln fest. Wenn man nur eines der beiden Attribute angibt, wird das andere automatisch so berechnet, dass das Seitenverhältnis des Bildes bestehen bleibt.

Videos und Audios werden auf die gleiche Weise eingebunden:

Information 2.19 Videos werden über den `video`-Tag eingebunden, Sounds bzw. Musik über den `audio`-Tag. Die beiden Tags haben für uns keine größere Relevanz, deshalb verweisen wir auf das Internet für den Einsatz dieser Tags.

3 CSS

Mit Hilfe von HTML legt man fest, *was* auf einer Webseite zu sehen sein soll (z.B. eine Überschrift, zwei Absätze Text und eine Tabelle), mit CSS (»Cascading Style Sheets«), *wie* diese Elemente aussehen sollen. Inhalt und Form werden also vollständig voneinander getrennt.

3.1 CSS-Dateien verlinken

Damit eine HTML-Seite eine CSS-Datei laden kann, muss diese zunächst verlinkt werden:

Information 3.1 Um eine CSS-Datei zu verlinken, geht man folgendermaßen vor:

(1) Erstelle eine neue Datei mit der Endung `.css`) z.B. `style.css`. Diese Datei muss im selben Ordner liegen wie die zugehörige HTML-Datei.

(2) Füge dem `head` der HTML-Datei den `link`-Tag hinzu:

```
<link rel="stylesheet" href="NAME DER CSS-DATEI.css"/>
```

also z.B. `<link rel="stylesheet" href="style.css"/>`

3.2 HTML-Elemente stylen

Eine CSS-Datei ist deutlich einfacher aufgebaut als eine HTML-Datei:

Information 3.2 Eine CSS-Datei besteht aus Blöcken der Form

```
1   SELEKTOR{
2     eigenschaft1: wert1;
3     eigenschaft2: wert2;
4     ...
5   }
```

Erläuterung:

`SELEKTOR`: Legt fest, *welche* HTML-Elemente gestylt werden sollen, z.B. `a`. Mit $*$ wählt man alle Elemente aus.

`eigenschaft`: Die CSS-Eigenschaft, die definiert werden soll, z.B. `color`.

`wert`: Wert, den die CSS-Eigenschaft haben soll, z.B. `red`.

Beispiel 3.3 Gegeben ist der CSS-Code:

```
1   *{
2     font-family: sans-serif;
3   }
4   a{
5     color: blue;
6     text-transform: uppercase;
7   }
8   h1{
9     text-align: center;
10    font-size: 100%;
11  }
```

Die Datei bewirkt folgendes:

(1) Alle Elemente erhalten eine serifenlose Schrift.

(2) Alle Links werden blau dargestellt und komplett GROSS geschrieben.

(3) Alle `h1`-Überschriften werden zentriert und in normaler Schriftgröße angezeigt.

#

Information 3.4 Eine Auswahl wichtiger CSS-Eigenschaften:

- `color`, `background-color`: Schrift- bzw. Hintergrundfarbe
- `border`: Rahmen, z.B. `border: 1pt solid black`. Möglich ist es auch, nur einzelne Rahmen zu setzen (mit `border-top`, `border-bottom`, `border-left`, `border-right`).
- `text-align`: Ausrichtung des Textes. Mögliche Werte:

`left`:	Linksbündig.
`right`:	Rechtsbündig.
`center`:	Zentriert.
`justified`:	Blocksatz.

Bemerkung 3.5 Wir werden in diesem Skript nur eine sehr kleine Auswahl an CSS-Eigenschaften behandeln. Für alles weitere gibt es Internetseiten wie `css4you` oder `selfhtml` . Grob gesagt: Es gibt mittlerweile genug CSS-Eigenschaften um damit jeden gewünschten optischen Effekt hervorrufen zu können. #

3.3 CSS-Klassen

Wir haben gesehen, wie man mit CSS alle Elemente eines bestimmten Typs stylen kann. Nun geht es darum, wie man nur bestimmte Elemente stylen kann und nicht alle.

Information 3.6 Man kann HTML-Elemente in **Klassen** einteilen. Dazu verwendet man das Attribut `class`. Dann kann man CSS-Regeln festlegen, die nur Elemente der jeweiligen Klasse betreffen. Als Selektor verwendet man einen Punkt, gefolgt vom Klassennamen.

```
1   .klasse{
2     eigenschaften, die nur Elemente dieser Klasse betreffen
3   }
```

Beispiel 3.7 In einer Tabelle sollen die Zeilen abwechselnd weiß und grau hinterlegt werden.

```
1   <table>
2     <tr>
3       <th>Stadt</th><th>Einwohner</th>
4     </tr>
5     <tr class="grau">
6       <td>Berlin</td><td>3,5 Mio</td>
7     </tr>
8     <tr>
9       <td>Frankfurt</td><td>0,8 Mio</td>
10    </tr>
11    <tr class="grau">
12      <td>London</td><td>8,8 Mio</td>
13    </tr>
14  </table>
```

#

3.4 Das Box-Modell

Manchmal möchte man ein bestimmtes Aussehen erreichen, legt die CSS-Eigenschaften fest und wundert sich dann, dass das Ergebnis nicht so aussieht, wie man es gewünscht hat. Der Grund hierfür liegt in vielen Fällen im (fehlenden) Verständnis des Box-Modells von CSS.

Information 3.8 Jedes HTML-Element entspricht in der CSS-Welt einem Rechteck (engl. *box*). Dieses Rechteck hat eine Breite, eine Höhe, eine »Polsterung« (engl. *padding*), einen Rahmen (engl. *border*) und einen Außenabstand (engl. *margin*) (vgl. Abb. ??).

Diese können mit Hilfe der CSS-Eigenschaften `width`, `height`, `padding`, `border` und `margin` festgelegt werden. Mittels der Zusätze `-top`, `-bottom`, `-left` und `-right` kann man verschiedene Werte für oben, unten, links und rechts festlegen.

In CSS unterscheidet man prinzipiell zwischen »Inline-Elementen« und »Block-Elementen«:

Information 3.9 Die `display`-Eigenschaft legt fest, ob es sich um ein Block-Element oder ein Inline-Element oder eine Mischung aus beidem handelt:

`block`: **Block-Elemente** (engl. block-level elements) sind dadurch gekennzeichnet, dass sie standardmäßig einen Zeilenumbruch vor und nach sich erzwingen. Sie stehen dann sozusagen als einzelner Block in einer Zeile.

`inline`:	**Inline-Elemente** erzwingen standardmäßig *keinen* Zeilenumbruch. Sie befinden sich damit *innerhalb des Textflusses* (deshalb: »in-line«). Die Box ist exakt so breit und hoch wie ihr Inhalt. Breite und Höhe können nicht festgelegt werden!
`inline-block`:	Eine Mischung aus `block` und `inline`: Es handelt sich um ein `inline`-Element, dessen Breite und Höhe aber festgelegt werden können.

Beispiel 3.10 Alle HTML-Elemente fallen standardmäßig in eine dieser drei Kategorien.[4]

- Beispiele für Block-Elemente: `h1`, `table`, `ul`, `li`, `p`, ...
- Beispiele für Inline-Elemente: `em`, `strong`, `a`, ...
- Beispiel für ein Inline-Block-Element: `img`

#

3.5 `div`- und `span`-Elemente

Die beiden am häufigsten verwendeten HTML-ELemente sind `div` und `span`:

Information 3.11

(a) Ein `div`-Element ist ein generisches Block-Element, d.h., es hat keine anderen CSS-Eigenschaften außer `display: block`.

`div`-Elemente werden (vor allem) dazu verwendet, einen bestimmten Bereich der Webseite (Header, Content, Footer, Sidebar, ...) zu kennzeichnen (englisch *division* = »Einteilung«).

(b) Ein `span`-Element ist ein generisches Inline-Element, d.h., es hat keine anderen CSS-Eigenschaften außer `display: inline`.

Beispiel 3.12 Der Body einer Webseite könnte beispielsweise aus drei Hauptbereiche `header`, `content` und `footer` aufgebaut sein:

```
1   <body>
```

[4] Das ist nicht die ganze Wahrheit, da es noch eine ganze Reihe weiterer möglicher Werte für die `display`-Eigenschaft gibt.

```
2     <div class="header">
3       <h1>Die coolste Seite der Welt!</h1>
4       <ul class="navigation>
5         <li><a href="home.html">Home</a></li>
6         <li><a href="kontakt.html">Kontakt</a></li>
7         <li><a href="impressum.html">Impressum</a></li>
8       </ul>
9     </div>
10    <div class="content">
11      <p>Hallo und herzlich willkommen, ...</p>
12    </div>
13    <div class="footer">
14      Diese Seite wurde erstellt von Thomas Klein.
15    </div>
16  </body>
```

#

3.6 Verschachtelte CSS-Selektoren

Gerade in Zusammenhang mit `div`-Elementen sind sog. verschachtelte CSS-Selektoren äußerst praktisch:

Information 3.13 Eine CSS-Regel der Form

```
1   element1 element2{
2     ...
3   }
```

stylt alle Elemente der Art `element2`, die sich innerhalb eines Elementes der Art `element1` befinden.

Beispiel 3.14 Wenn wir von dem Markup in **Beispiel 3.12** ausgehen, könnten wir folgende CSS-Regel hinzufügen:

```
1    .header a{
2      color: red;
3      border: 1pt solid black;
4      background-color: gray;
5    }
```

Dieser Code sorgt dafür, dass nur die Links im Navigationsmenü gestylt werden, alle anderen aber nicht. #

3.7 Layout mit CSS-Grid

Wie wir gesehen haben, wird eine Webseite normalerweise in bestimmte Bereiche aufgeteilt: Es gibt ein Navigationsmenü, einen Kopfbereich)engl. *header*), einen Fußbereich (engl. *footer*), möglicherweise eine Sidebar, sowie einen Bereich für den eigentlichen Inhalt (engl. *content*) der Seite (vgl. **Abb. 7** für die Wikipedia-Seite).

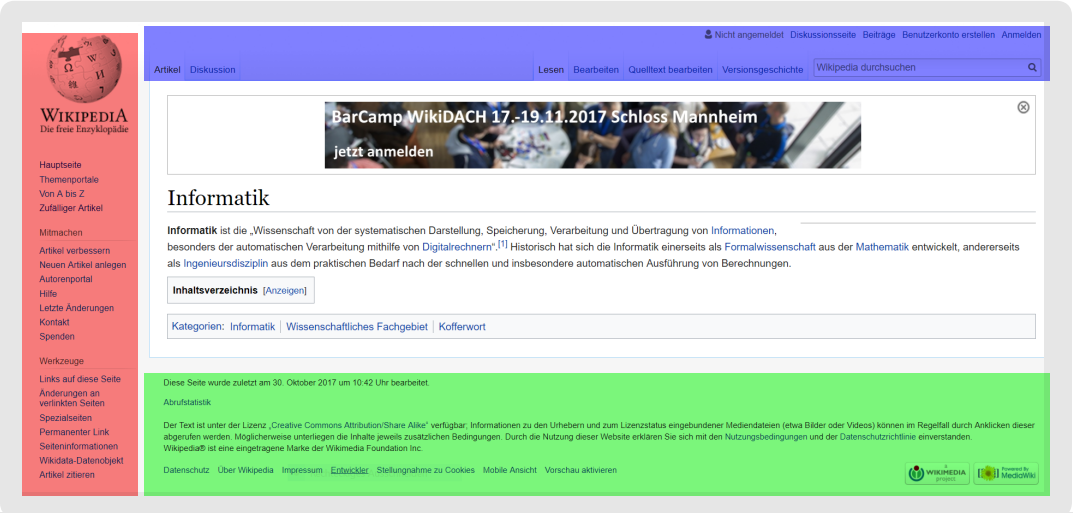


Abbildung 7 Die Wikipedia-Seite zur Informatik ist aufgeteilt in eine Sidebar (rot), einen Header (blau), einen Footer (grün) und den eigentlichen Inhalt (der Rest).

Dazu verwenden wir wieder unseren Beispiel-Code aus **Beispiel 3.12**:

```
1    <body>
2      <div class="header">
3        <h1>Die coolste Seite der Welt!</h1>
4        <ul class="navigation>
5          <li><a href="home.html">Home</a></li>
6          <li><a href="kontakt.html">Kontakt</a></li>
```

```
7           <li><a href="impressum.html">Impressum</a></li>
8         </ul>
9       </div>
10      <div class="content">
11        <p>Hallo und herzlich willkommen, ...</p>
12      </div>
13      <div class="footer">
14        Diese Seite wurde erstellt von Thomas Klein.
15      </div>
16    </body>
```

Wie erreichen wir aber nun ein solches Layout wie bei der Wikipedia-Seite? Früher wurde so etwas über Tabellen gelöst, was aber eine extrem schlechte Idee ist. Hier ein Zitat von **selfhtml.org** :

> *»Früher wurden HTML-Tabellen dazu eingesetzt, komplette Layouts umzusetzen. Das widerspricht allerdings letztlich dem Grundgedanken eines modernen Webdesigns (Stichwort: Trennung von Inhalt und Layout), kann Probleme hinsichtlich der Barrierefreiheit für Screen-Reader-Nutzer verursachen und ist schlecht zu warten. Verwenden Sie hierfür das Grid Layout.«*

Um genau dieses Grid-Layout soll es jetzt gehen:

Information 3.15 Mit Hilfe des CSS-Gridlayouts können die Bereiche (`div`s) einer Webseite in einem Gitter (englisch *grid*) angeordnet werden. Dazu sind mehrere Schritte notwendig:

(1) Das `html`- und das `body`-Element müssen per CSS so eingestellt werden, dass sie das gesamte Browserfenster ausfüllen:

```
1   html,body{
2     width: 100%;
3     height: 100%;
4     margin: 0;
5     padding: 0;
6   }
```

(2) Das `body`-Element wird zum sog. **Grid-Container**:

```
1  body{
2    display: grid;
3    grid-template-columns: Breite der einzelnen Spalten
4    grid-template-rows: Höhe der einzelnen Zeilen
5  }
```

Die Breite der einzelnen Spalten/Zeilen kann ganz individuell eingestellt werden. Hier die wichtigsten Möglichkeiten:

`50px`: Die Spalte/Zeile ist genau 50 Pixel breit/hoch.

`auto`: Die Spalte/Zeile ist genauso breit/hoch wie ihr Inhalt. Perfekt für eine Sidebar!

`1fr`: Die Spalte/Zeile erstreckt sich über den gesamten restlichen Platz, der zur Verfügung steht. Perfekt für den Content der Seite!

(3) (Optional) Manchmal soll ein Element sich über mehrere Spalten erstrecken. In diesem Fall muss dieses Element die CSS-Eigenschaft `grid-column: start/ende` erhalten, wobei `start` die Nummer der Spalte ist, in der das Element stehen soll und `ende` die Nummer der Spalte, die nach dem Element kommt.

(4) (Optional) Dasselbe kann man auch machen, wenn sich ein Element über mehrere Zeilen erstrecken soll. Man schreibt dann `grid-row: start/ende`.

Beispiel 3.16 Gegeben ist das folgende Markup des Bodys:

```
1  <body>
2    <div class="A">Bereich A</div>
3    <div class="B">Bereich B</div>
4    <div class="C">Bereich C</div>
5    <div class="D">Bereich D</div>
6    <div class="E">Bereich E</div>
7  </body>
```

Nun soll es darum gehen, das nebenstehende Aussehen zu erreichen. Dazu kann man folgenden CSS-Code verwenden:

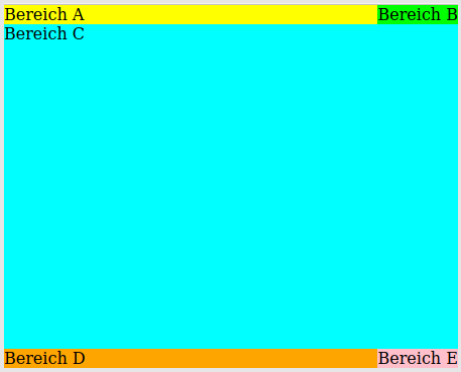


Abbildung 8

```
1    html,body{
2      height: 100%;
3      width: 100%;
4      padding: 0;
5      margin: 0;
6    }
7
8    body{
9      display: grid;
10     grid-template-columns: 1fr auto;
11     grid-template-rows: auto 1fr auto;
12   }
13
14   .A{
15     background-color: yellow;
16   }
17
18   .B{
19     background-color: lime;
20   }
21
22   .C{
23     background-color: cyan;
24     grid-column: 1/3;
25   }
26
27   .D{
28     background-color: orange;
29   }
30
31   .E{
32     background-color: pink;
33   }
```

Zur Erläuterung:

(1) `grid-template-columns: 1fr auto auto;` legt fest, dass unser Gitter aus 2 Spalten besteht. Die rechte Spalte ist so breit wie ihr Inhalt, die linke Spalte erhält den restlichen Platz.

(2) `grid-template-rows: auto 1fr auto;` legt fest, dass unser Gitter aus 3 Zeilen besteht. Die obere Zeile ist so hoch wie ihr Inhalt, ebenso die untere Zeile. Die Zeile in der Mitte erhält den ganzen restlichen Platz.

(3) `grid-column: 1/3;` legt fest, dass das Div-Element `C` sich von Spalte 1 bis Spalte 3 erstrecken soll. Man muss deshalb 3 schreiben, weil die dritte Spalte (die es nicht gibt!) als nächstes kommen würde.

Das erscheint zunächst gegen die Intuition. Man kann es sich aber so merken: Die erste Zahl sagt, in welcher Spalte das Element stehen soll. Die Differenz der zweiten Zahl und der ersten Zahl gibt an, über wie viele Spalten sich das Element erstrecken soll (hier: $3 - 1 = 2$ Spalten). #

Beispiel 3.17 Nun zum Eingangsbeispiel der Wikipedia-Seite. Wir haben hier prinzipiell zwei Spalten und drei Zeilen, wobei sich die Sidebar über alle drei Zeilen erstreckt. Mit folgendem CSS-Code kann man diesen Aufbau erreichen:

```
1   body{
2     display: grid;
3     grid-template-columns: auto 1fr;
4     grid-template-rows: auto 1fr auto;
5   }
6
7   .sidebar{
8     grid-row: 1/4;
9   }
```

#

4 Internetprotokolle

In den vorangegangenen Kapiteln haben wir uns damit beschäftigt, wie Webseiten erstellt werden können. Nun soll es darum gehen, wie diese Webseiten abgerufen werden können. Wir gehen also den Fragen »Was ist überhaupt das Internet und wie funktioniert es?« nach.

Grundlegend dafür sind die Begriffe des »Netzwerks« und des »Protokolls«.

4.1 Rechnernetze und Kommunikationsprotokolle

Information 4.1 Ein **Rechnernetz** ist ein Zusammenschluss mehrerer Rechner und Peripheriegeräte (Drucker, Scanner, Fax, etc.) — **Teilnehmer** genannt —, in dem die einzelnen Teilnehmer miteinander kommunizieren können.

Wie diese Kommunikation im einzelnen funktioniert, wird über das sogenannte **Protokoll** festgelegt. Damit die Kommunikation funktioniert, müssen sich die Teilnehmer an das Protokoll halten. Das Protokoll legt fest, wer wem zu welchem Zeitpunkt welche Daten sendet und was diese Daten bedeuten.

Beispiel 4.2 Die Kolleg*innen der Marienschule wollen einen kollegialen Austausch zum Unterricht per Video-Konferenz etablieren. Dazu wird folgendes Protokoll verabredet, das die Konferenz regelt:

(1) Phase 1: Bestimmung des Moderators. Ein beliebiger Teilnehmer bestimmt per Würfelwurf eine Moderator*in.

(2) Phase 2: Themensammlung. Die moderierende Lehrkraft ruft nacheinander alle Teilnehmer auf, in kurzen Worten zu beschreiben, was sie zum heutigen Treffen beitragen wollen. Diese Beiträge bleiben unkommentiert.

(3) Phase 3: Austausch. Die Teilnehmer melden sich, wenn sie etwas sagen wollen. Der Moderator ruft jeweils auf.

(4) Phase 4: Sicherung. Die Teilnehmer notieren die wichtigsten Ergebnisse des Treffens in einem OneNote-Dokument.

#

Bemerkung 4.3 Das Wort »Protokoll« wird heutzutage eher als »Dokument, das beschreibt, was in einer Sitzung passiert ist« verwendet. Diese Bedeutung ist in diesem Zusammenhang nicht gemeint. #

4.2 Die TCP/IP-Protokollfamilie

Im Laufe der Entwicklung der Rechnernetze wurden viele Protokolle erdacht, mit denen kommuniziert werden kann. Durchgesetzt hat sich die sogenannte »TCP/IP-Protokollfamilie«:

Information 4.4 Die **TCP/IP-Protokollfamilie** ist eine Sammlung von über 500 Netzwerkprotokollen. Dabei steht

- `IP` für »Internet Protocol«. Es regelt, wie die Datenpakete von einem Teilnehmer zum anderen gelangen.
- `TCP` für »Transmission Control Protocol«. Es regelt, wie eine direkte Ende-zu-Ende-Kommunikation zwischen zwei Teilnehmern abläuft.

4.2.1 Das Internet Protocol und der Domain Name Service

Das Internet Protocol regelt z. B. auf welchem Pfad ein Datenpaket von einem Teilnehmer zum nächsten gelangt. Damit dies funktionieren kann, muss jeder Teilnehmer eindeutig identifizierbar sein:

Information 4.5 Jeder Teilnehmer des Netzwerks erhält eine eindeutige **IP-Adresse**. Dies ist eine Binärzahl, anhand derer der Teilnehmer eindeutig identifiziert werden kann. Die Länge der Binärzahl hängt vom konkreten Protokoll ab:

IP v4 Version 4 des Internet Protocol stammt aus dem Jahr 1981. Hier ist die IP-Adresse 4 Bytes lang.

IP v6 Version 6 gibt es seit 1998. Die IP-Adresse ist 16 Byte lang.

Damit sich ein Benutzer keine ellenlangen IP-Adressen merken muss, gibt es den sog. »Domain Name Service«:

Information 4.6 Beim **Domain Name Service (DNS)** kann ein Teilnehmer nach der IP-Adresse eines anderen Teilnehmers nachfragen.

Gibt man z. B. `google.de` ein, so fragt der Browser beim DNS nach der IP-Adresse der **URL (Uniform Resource Locator)** `google.de` und ruft dann die entsprechende Seite auf.

Bemerkung 4.7 Man kann sich den DNS als eine Art Auskunft oder Telefonbuch für IP-Adressen vorstellen. #

4.2.2 Das Transfer Control Protocol

Das Transmission Control Protocol kümmert sich darum, wie Daten zwischen zwei Teilnehmern ausgetauscht werden, wenn zuvor über das Internet Protocol eine Verbindung hergestellt wurde.

Information 4.8 TCP erweitert die IP-Adresse um den sogenannten **Port**, eine Zahl. In TCP muss man stets angeben, an welchen Port man Daten senden möchte.

Eine TCP-Verbindung wird also durch folgende 4 Werte definiert:

(1) Quell-Adresse: IP-Adresse des Teilnehmers, der die Kommunikation gestartet hat.

(2) Quell-Port: Zahl, die angibt, auf welchem Port der Quell-Teilnehmer die Antworten des Ziels erwartet.

(3) Ziel-Adresse: IP-Adresse des Teilnehmers, der »angerufen« wird.

(4) Ziel-Port: Zahl, die angibt, auf welchem Port der Ziel-Teilnehmer die Anfragen der Quelle entgegennimmt.

Bemerkung 4.9 Wenn die IP-Adresse den Ort, die Straße und die Hausnummer angeben würde, so würde der Ort die Zimmernummer angeben. Die Unterscheidung in verschiedene Ports erlaubt es jedem Teilnehmer viele verschiedene Verbindungen zu unterschiedlichen Teilnehmern aufzubauen. #

4.2.3 Das TCP-IP-Referenzmodell

IP und TCP sind zwei aufeinander aufbauende Protokolle. Fügt man weitere Protokolle hinzu, so erhält man das sog. »TCP-IP-Referenzmodell«:

Information 4.10 Das **TCP-IP-Referenzmodell** ordnet die Protokolle verschiedenen, aufeinander aufbauenden Schichten zu:

Schicht	Beschreibung	Beispiel-Protokolle
Schicht 4: Anwendung	Protokolle, die konkrete Software (Apps, Browser, Spiele) zum Datenaustausch verwenden kann	HTTP, SMTP, FTP
Schicht 3: Transport	Protokolle zum Versenden und Empfangen von Daten zwischen zwei Teilnehmern (meist im Betriebssystem implementiert)	TCP, UDP
Schicht 2: Internet	Protokolle zur Verbindungsherstellung zwischen zwei Teilnehmern	IP v4, IP v6
Schicht 1: Netz-Zugang	Protokolle zur Verbindung eines Teilnehmers mit einem Netzwerk	Ethernet

Jede Schicht ist auf die darunterliegenden Schichten angewiesen.

4.3 Das Client-Server-Modell

Das Internet (und viele andere Netzwerke) sind nach dem »Client-Server-Modell« aufgebaut:

Information 4.11 Ein Teilnehmer eines Netzwerks heißt

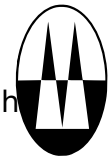
- **Server** (englisch: Diener, Kellner), wenn er Dienste für andere Teilnehmer anbietet.
- **Client** (englisch: Klient/Kunde), wenn er einen Dienst bei einem Server anfragt.

Der Client fordert also einen Dienst bei einem Server an (sog. **Request** bzw. **Anfrage**). Der Server antwortet dann mit der gewünschten Information oder einer Fehlermeldung (sog. **Response** bzw. **Antwort**).

Bemerkung 4.12 Genau genommen kann man nicht sagen, dass ein Rechner ein Server oder ein Client ist. Innerhalb einer Verbindung wird ein Rechner zum Client bzw. zum Server. Es kann aber durchaus sein, dass ich von meinem Smartphone aus eine Webseite bei einem Server aufrufe und dieser wiederum zur Erfüllung meines Requests selbst zum Client wird, weil er Informationen eines weiteren Servers benötigt. #

4.4 Protokolle der Anwendungsschicht

In diesem Abschnitt werfen wir einen Blick auf die drei wichtigsten Protokolle der Anwendungsschicht: HTTP(s), FPT(s) und SMTP(s):



Information 4.13 Das **Hypertext Transfer Protocol (HTTP)** regelt, wie Daten von einem Server zu einem Client übertragen werden können:

(1) Der Client sendet einen Request an die IP-Adresse eines Servers inklusive Portnummer (80 für Webserver).

(2) Der Server empfängt den Request und sendet eine Antwort, die u. a. die folgenden Bestandteile enthält:

- HTTP-Statuscode: Eine Zahl, die angibt, ob die Anfrage funktioniert hat. Beispiele:
  - `102 (Processing)` Der Server bearbeitet die Anfrage noch und bittet den Client, noch etwas zu warten.
  - `200 (OK)` Anfrage hat funktioniert.
  - `304 (Not modified)` Die angeforderten Daten haben sich nicht verändert, müssen also nicht erneut übertragen werden.
  - `404 (Not found)` Die angeforderten Daten existieren nicht auf diesem Server.
  - `503 (Service unavailable)` Der Server ist aktuell nicht in der Lage, die Anfrage zu bearbeiten.
- Daten: Die angeforderten Daten, z. B. eine HTML- oder CSS-Datei oder eine Grafik.

Die Variante **HTTPs (Hypertext Transfer Protocol Secure)** ist eine verschlüsselte Form von HTTP (erkennbar am Schloss-Symbol der meisten Browser).

Über HTTP funktioniert also der wichtigste Dienst des Internets: das **WWW (World Wide Web)**.

Neben dem Web gibt es zahllose weitere Dienste im Internet:

Information 4.14 Das **File Transfer Protocol (FTP)** regelt, wie man Dateien auf einen Server hochladen oder von diesem herunterladen kann. Im Gegensatz zu HTTP ist hierzu ein Benutzerzugang und ein Passwort notwendig. **FTPs** ist die verschlüsselte Variante.

Das **Simple Mail Transfer Protocol (SMTP)** regelt, wie E-Mails versendet und empfangen werden. Auch hier ist SMTPs die verschlüsselte Variante.

II

Einführung in die Programmierung

1 Schnelleinstieg in Java

Das Ziel dieses ersten Kapitels ist, Sie so schnell wie möglich zu befähigen, erste einfache Java-Programme zu schreiben. Dabei müssen wir in Kauf nehmen, nicht alles von Anfang an verstehen zu können. Wir werden die Details in den späteren Kapiteln klären.

1.1 Hallo Welt

Das folgende Java-Programm gibt »Hallo Welt« in der sog. »Konsole« aus:

```
1   class HalloWelt {
2     void onStart( ){
3       System.out.println( "Hallo Welt" );
4       System.out.println("Ich heiße Thomas.");
5     }

6     public static void main( String[] args ){
7       new HalloWelt( );
8     }
9   }
```

Hier einige Erläuterungen zu diesem **Quellcode**:

(1) In Java wird sämtlicher Code in sog. **Klassen** organisiert. In Zeile 1 wird die Klasse `HalloWelt` geöffnet und in Zeile 8 abgeschlossen. Alles, was zwischen den geschweiften Klammern { und } steht, gehört zu der Klasse.

(2) Die Klasse enthält zwei **Methoden**:

- Die Methode `onStart` wird ausgeführt, wenn die App gestartet wird.
- Die Methode `main` interessiert uns zunächst nicht. Wichtig ist nur, dass darin `new HalloWelt();` steht.

(3) Die **Methode** `onStart` besteht aus zwei **Anweisungen**. Diese werden von oben nach unten abgearbeitet, wenn die Methode ausgeführt wird. In diesem Fall wird über den Befehl `System.out.println` zuerst der Text `Hallo Welt` in der Konsole ausgegeben und anschließend der Text `Ich heiße Thomas.`.

Information 1.1 **Merke:** Mit dem Befehl `System.out.println` kann man Text in der Konsole ausgeben.

1.2 Das User-Interface (UI)

Unter einer **UI** (»User Interface«) versteht man die grafische Oberfläche eines Programms; also all das, was der User sehen kann und womit er interagieren kann.

Information 1.2 In JavaApp müssen Benutzeroberflächen nicht per Programmcode programmiert werden, sondern können in einem Editor aus Blöcken zusammengesetzt werden.

Dafür muss eine neue »UI-Klasse« hinzugefügt werden. Anschließend kann man der UI-Klasse beliebig Komponenten hinzufügen:

- Ein `JButton` ist ein Knopf, den der User anklicken kann.
- Ein `JLabel` ist ein Text, der angezeigt wird.
- Ein `JTextField` ist ein Eingabefeld, in das der User Text oder Zahlen eingeben kann.
- Ein `JPanel` ist ein Container, der andere Komponenten aufnehmen kann.

Man kann jeder Komponente einen Namen geben, unter dem man sie später beim Programmieren verwenden kann.

Damit erstellen wir uns folgende UI-Klasse:

Diese besteht aus einem JLabel (ohne Name), einem JTextField (mit Name »eingabe«), einem JButton (ohne Name) und einem weiteren JLabel (mit name »ausgabe«).

1.3 Die `onAction`-Methode

Wir ändern unsere HalloWelt-App nun so ab, dass ...

- ... die UI bei Programmstart erzeugt wird;
- ... ein Klick auf den Button bewirkt, dass die Zahl, die in das Textfeld eingegeben wurde, um 1 erhöht wird und das Ergebnis in das untere Label geschrieben wird.

```
1   class HalloWelt {
2     UI ui;

3     void onStart( ){
4       ui = new UI ( );
5     }

6     void onAction(){
7       int zahl = ui.eingabe.getValue ( );
8       int ergebnis = zahl + 1;
9       ui.ausgabe.setValue ( ergebnis );
10    }

11    public static void main( String[] args ){
12      new HalloWelt( );
13    }
14  }
```

Wieder einige Anmerkungen:

(1) In Zeile 2 wird die **Variable** `ui` vom Typ `UI` **deklariert**.

(2) In Zeile 4 wird die UI erzeugt und in der Variablen `ui` gespeichert.

(3) In den Zeilen 6 bis 10 wird die Methode `onAction` implementiert: Diese wird jedes Mal ausgeführt, wenn der User den Button klickt.

(4) In Zeile 7 wird die Zahl, die der User in das JTextField `eingabe` eingegeben hat, in der Variablen `zahl` gespeichert.

(5) In Zeile 8 wird das Ergebnis der Rechnung `zahl + 1` in der Variablen `ergebnis` gespeichert.

(6) In Zeile 9 wird das Ergebnis der Rechnung in das JLabel `ausgabe` geschrieben und es erscheint auf dem Bildschirm.

Information 1.3 Merke:

(1) Jedes Mal, wenn ein **Button** geklickt wird, wird die Methode `onAction` aufgerufen.

(2) Mit `getValue` holt man sich die Eingabe des Users.

(3) Mit `setValue` verändert man den Wert einer Komponente.

(4) Mit `int name = wert;` deklariert man eine neue Variable namens `name` und weist ihr den Wert `wert` zu. `int` steht dabei für »integer« (englisch für »ganze Zahl«), d.h., in dieser Variablen kann man eine ganze Zahl speichern.

2 EVA

In diesem Kapitel lernen wir, wie einfache Java-Applikationen programmiert werden können. Diese funktionieren nach dem sog. **EVA**-Prinzip: Eingabe, Verarbeitung, Ausgabe.

Ein solches Programm ist immer folgendermaßen aufgebaut:

```
1   class App {
2     UI ui;
3     void onStart ( ){
4       ui = new UI ( );
5     }
6     void onAction ( JComponent trigger ) {
7       //EINGABE: lies die Textfelder aus und speichere die Werte in Variablen
8       //VERARBEITUNG: berechne dann das Ergebnis
9       //AUSGABE: gib das Ergebnis in einem Label aus
10    }
11    public static void main ( String[ ] args ) {
12      new App ( );
13    }
14  }
```

2.1 Datentypen

In einem Computerprogramm werden normalerweise große Mengen von Daten verarbeitet. Diese Daten teilt man in unterschiedliche Kategorien ein:

Definition 2.1 *In Java unterscheiden wir zwischen den folgenden* **Datentypen***:*

- *Ganze Zahlen werden mit* `int` *bezeichnet.*
- *Kommazahlen werden mit* `double` *bezeichnet. Kommazahlen schreibt man in der englischen Version mit ».« statt »,«!*
- *Zeichenketten werden mit* `String` *bezeichnet. Zeichenketten werden in Anführungszeichen geschrieben.*
- *Wahrheitswerte werden mit* `boolean` *bezeichnet. Es gibt nur zwei mögliche Wahrheitswerte:* `true` *und* `false`*.*

Beispiel 2.2 Wir betrachten verschiedene Daten zu einem Menschen:

(a) Das Alter ist ein `int`, denn ein Mensch ist bspw. 38 Jahre alt.

(b) Die Größe ist ein `double`, denn ein Mensch kann bspw. 1.82 Meter groß sein.

(c) Der Name ist ein `String`, z.B. `"Kim Meyer"`.

(d) Die Information, ob der Mensch ein Musikinstrument spielt, ist ein `boolean`, denn entweder der Mensch spielt ein Instrument (`true`) oder nicht (`false`). #

Bemerkung 2.3 Es gibt noch eine ganze Reihe weiterer Datentypen wie `char`, `long` oder `byte`. In der Q1 werden wir sogar unsere eigenen Datentypen definieren. Für den Moment reichen uns aber die obigen vier Datentypen. #

2.2 Variablen

Definition 2.4 *Eine* **Variable** *ist eine Art Behälter, die einen Wert eines bestimmten Datentyps aufnehmen kann.*

Jede Variable hat einen Namen und einen **Datentyp***. Der Datentyp legt fest, welche Art von Daten man in der Variablen speichern kann.*

Bemerkung 2.5 Variablen sollten immer möglichst »sprechende« Namen haben, die direkt verraten, was in der Variablen gespeichert werden soll. Gute Namen sind also `vorname` und `wohnort`, schlechte Namen sind `v` oder `w`. #

Information 2.6 Bevor eine Variable verwendet werden kann, muss sie zunächst **deklariert** werden. Dazu schreibt man

```
Datentyp name;
```

Dann kann man der Variablen mit Hilfe des Gleich-Zeichens einen Wert zuweisen:

```
name = wert;
```

Es ist auch möglich, beides zu kombinieren und einer neu deklarierten Variablen direkt einen Wert zu geben:

```
Datentyp name = wert;
```

Eine Variable existiert nur innerhalb des Code-Blocks, in dem sie deklariert wurde. Ein Code-Block beginnt und endet mit geschweiften Klammern `{...}`.

Beispiel 2.7 Das folgende Programm rechnet auf Knopfdruck Dollar in Euro um (aktueller Wechselkurs: 1 Dollar = $0{,}94$ Euro):

```
1   class DollarInEuro {
2     UI ui;
3     void onStart ( ) {
4       ui = new UI ( );
5     }
6     void onAction ( JComponent trigger ) {
7       double dollar = ui.eingabe.getValue ( );
8       double euro = dollar * 0.94;
9       String ergebnis = String.format ( "%.2f €", euro);
10      ui.ausgabe.setValue( ergebnis );
11    }
12    public static void main ( String[ ] args ) {
13      new DollarInEuro ( );
14    }
15  }
```

In diesem Programm kommen vier Variablen vor:

(1) Die Variable `ui` kann einen Wert vom Datentyp `UI` speichern. Hierbei handelt es sich um eine UI-Klasse.

Die Variable ist innerhalb des gesamten Programms definiert.

(2) Die Variable `dollar` kann einen Wert vom Datentyp `double` (eine Kommazahl) speichern. Sie erhält als Wert die Eingabe des Users.

Die Variable ist nur innerhalb der Methode `onAction` definiert.

(3) Die Variable `euro` kann ebenso einen Wert vom Datentyp `double` (eine Kommazahl) speichern. Ihr Wert wird festgelegt als der Wert der Variablen `dollar` multipliziert mit 0,94.

Die Variable ist nur innerhalb der Methode `onAction` definiert.

(4) Die Variable `ergebnis` kann einen Wert vom Typ `String` (eine Zeichenkette) speichern. Um ihren Wert zu bestimmen, wird der Wert der Variablen `euro` auf zwei Nachkommastellen gerundet und ein »€«-Zeichen angehängt.

Auch diese Variable existiert ausschließlich innerhalb der Methode `onAction`. #

Information 2.8 Eine Variable, die innerhalb des gesamten Programms existiert, wird auch als **globale Variable** bezeichnet. Eine Variable, die nur innerhalb eines Teils des Programms existiert, heißt dagegen **lokale Variable**.

2.3 Methoden und Anweisungen

Wie wir gesehen haben, ist unser Programm in verschiedene Methoden aufgeteilt:

Definition 2.9 *Eine* **Methode** *enthält eine Reihe von* **Anweisungen**. *Die Anweisungen werden von oben nach unten nacheinander abgearbeitet, wenn die Methode* **aufgerufen** *wird. Jede Anweisung muss mit einem Semikolon ; abgeschlossen werden.*

Jede Methode beginnt mit dem Schlüsselwert **void**, *gefolgt von ihrem Namen und runden Klammern, in denen* **Parameter** *stehen können (aber nicht müssen). Im nächsten Kapitel werden wir lernen, was es damit auf sich hat und dass dort nicht immer* `void` *stehen muss.*

Beispiel 2.10 Hier noch einmal der Code des »Dollar-Umrechners« aus **Beispiel 2.7**:

```
1   class DollarInEuro {
2     UI ui;
3     void onStart ( ) {
4       ui = new UI ( );
5     }
6     void onAction ( JComponent trigger ) {
7       double dollar = ui.eingabe.getValue ( );
8       double euro = dollar * 0.94;
9       String ergebnis = String.format ( "%.2f €", euro);
10      ui.ausgabe.setValue( ergebnis );
11    }
12    public static void main ( String[ ] args ) {
13      new DollarInEuro ( );
14    }
15  }
```

Dieses Programm besteht aus drei Methoden: `onStart`, `onAction` und `main`. Die beiden Methoden `onStart` und `main` enthalten jeweils nur eine einzige Anweisung, während `onAction` insgesamt vier Anweisungen enthält. #

Information 2.11 Jedes Java-Programm benötigt eine statische, öffentliche **`main`-Methode**:

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

Diese Methode wird aufgerufen, wenn das Java-Programm gestartet wird. Sie sollte nichts anderes machen, als die App zu erzeugen.

Sobald die App dann erzeugt wurde, übernimmt JavaApp die Kontrolle:

Information 2.12 In JavaApp gibt es Methoden, die automatisch aufgerufen werden, sobald ein bestimmtes **Ereignis** eintritt:

(1) Die Methode `onStart` wird aufgerufen, sobald das Programm gestartet wird.

(2) Die Methode `onAction` wird jedes Mal aufgerufen, sobald ein Button geklickt wird.

Bemerkung 2.13 Nun können wir den **Kontrollfluss** des Dollar-Umrechners erklären:

(1) Wenn das Programm gestartet wird, wird zunächst die `main`-Methode aufgerufen. Die Anweisung `new DollarInEuro ( );` erzeugt dann unsere App im Arbeitsspeicher des Computers.

(2) Sobald unsere App erzeugt wurde, wird die Methode `onStart` aufgerufen. Diese erzeugt die UI im Arbeitsspeicher und speichert diese in der Variablen `ui`. Jetzt können wir die grafische Oberfläche der App sehen.

Die App wechselt nun in einen Schlummer-Modus und wartet darauf, dass der User etwas macht.

(3) Sobald der User den Button klickt, wird die Methode `onAction` aufgerufen. Diese nimmt die Eingabe des Users und speichert sie als lokale Variable `dollar`. Anschließend wird der Betrag in Euro umgerechnet und in der lokalen Variablen `euro` gespeichert. Danach wiederum wird der Euro-Betrag als String mit zwei Nachkommstellen formatiert und ausgegeben.

Wenn der User erneut den Button klickt, wird wieder `onAction` ausgeführt. #

3 Methoden

Wir haben Methoden (im Sinne von Java) bereits im vorangegangenen Kapitel kennengelernt: Methoden enthalten Anweisungen, die ausgeführt werden, wenn die jeweilige Methode aufgerufen wird. Bisher haben wir aber nur Methoden wie `onAction` verwendet, die automatisch aufgerufen werden.

In diesem Kapitel schreiben wir unsere eigenen Methoden und sehen, wie wir sie verwenden können.

3.1 Parameter und Rückgabewerte

Eine Methode kann eine Eingabe erhalten, diese verarbeiten und das Ergebnis an die aufrufende Stelle zurückgeben.

Beispiel 3.1 Gegeben ist der folgende BMI-Rechner:

```
1   class BMIRechner {
2     UI ui;
3     void onStart ( ) {
4       ui = new UI ( );
5     }
6     void onAction ( JComponent trigger ) {
7       double bmi = berechneBMI ( 1.82, 78 );
8       ui.ausgabe.setValue( bmi );
9     }
10    double berechneBMI ( double groesse, double gewicht ) {
11      double b = gewicht / (groesse * groesse);
12      return b;
13    }
14    public static void main ( String[ ] args ) {
15      new BMIRechner ( );
16    }
17  }
```

Dieses Programm enthält die Methode:

```
double berechneBMI ( double groesse, double gewicht ) {
  double b = gewicht / (groesse * groesse);
  return b;
}
```

Schauen wir uns das genau an:

- Das erste »double« bedeutet, dass diese Methode als Ergebnis eine Kommazahl zurückgibt.
- Es folgt der Name der Methode: »berechneBMI«.
- In den runden Klammern stehen zwei **Parameter**: Der Parameter `groesse` ist eine `double`-Zahl, der Parameter `gewicht` ebenso.
- In den geschweiften Klammern stehen wie gewohnt die Anweisungen. Neu ist allerdings die `return`-Anweisung: Diese beendet die Ausführung der Methode und gibt den Wert der Variablen `b` zurück.

Das bedeutet für den Ablauf der Methode `onAction`:

(1) In Zeile 7 springt der Computer in die Methode `berechneBMI`. Dabei wird der Parameter `groesse` automatisch auf `1.82` gesetzt und der Parameter Gewicht auf `78`. Dann wird `b` berechnet mit diesen Werten (es ergibt sich $b = 78/(1.82 \cdot 1.82) \approx 23{,}55$). Der Wert von `b` (also $23{,}55$) wird dann zurückgegeben.

(2) Der Computer springt zurück in Zeile 7 und speichert das Ergebnis der Methode `berechneBMI` in der Variablen `bmi`. In Zeile 8 wird das Ergebnis dann angezeigt.

Mit dieser Methode `berechneBMI` haben wir uns also einen eigenen Befehl gebaut, den wir mit Größe und Gewicht füttern können und der uns dann den Body-Maß-Index zurück gibt. #

Information 3.2 Eine Methode kann **Parameter** als Eingabe erhalten. Diese werden wie lokale Variablen innerhalb der runden Klammer deklariert.

Eine Methode kann außerdem einen **Rückgabewert** berechnen und mit der Anweisung `return` zurückgeben. In diesem Fall wird anstelle des Schlüsselwortes `void` der Datentyp des Rückgabewertes vor dem Methodennamen notiert.

Insgesamt wird eine Methode also folgendermaßen deklariert:

```
Datentyp name ( Datentyp parameter, Datentyp parameter, ... ){
  Anweisungen
}
```

Beispiel 3.3 Der BMI-Rechner aus dem vorangegangenen Beispiel macht nicht wirklich Sinn, weil er immer dieselbe Größe und dasselbe Gewicht verwendet. Anstatt die festen Zahlen zu verwenden, kann man die Werte der Textfelder einsetzen:

```
1   void onAction ( JComponent trigger ) {
2     double bmi = berechneBMI ( ui.groesse.getValue( ), ui.gewicht.getValue( ) );
3     ui.ausgabe.setValue( bmi );
4   }
```

Dadurch werden die eingegebenen Werte aus den Textfeldern für die beiden Parameter eingesetzt. #

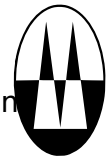
Beispiel 3.4 Welche Ausgabe wird das folgende Programm liefern?

```
1   class Programm {
2     void onStart ( ) {
3       System.out.println ( gesamt ( 3, 5, 4 ) );
4       System.out.println ( gesamt ( 6, 8, 10 ) );
5       System.out.println ( eintrittspreis ( 2, 3 ) );
6       System.out.println ( eintrittspreis ( 5, 6 ) );
7       System.out.println ( eintrittspreis ( gesamt ( 5, 3, 2 ), 3 ) );
8     }
9     int gesamt ( int a, int b, int c ) {
10      return a + b + c;
11    }
12    double eintrittspreis ( int erwachsene, int kinder ){
13      double p1 = erwachsene * 7.50;
14      double p2 = kinder * 4;
15      double preis = p1 + p2;
16      return preis;
17    }
18  }
```

Die Ausgabe ist

```
12
24
27
61.5
87
```

#



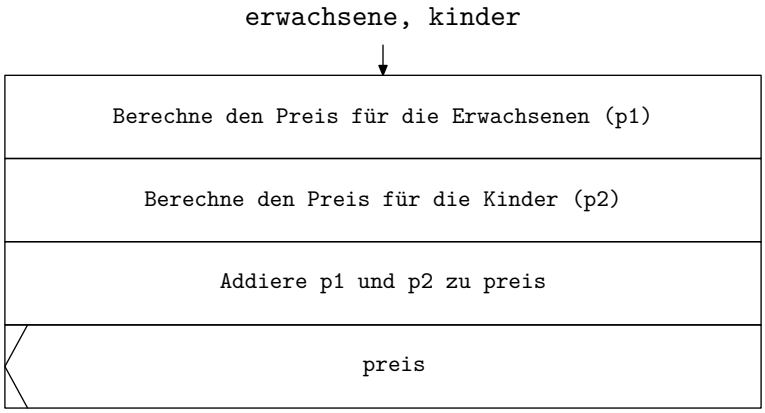
3.2 Darstellung als Struktogramm

Eine Methode kann auch als sog. »Struktogramm« dargestellt werden:

Definition 3.5 *Ein Struktogramm ist eine grafische Darstellung einer Methode:*

- *Jede Anweisung ist ein Rechteck.*
- *Eine `return`-Anweisung wird so dargestellt wie der letzte Block oben.*
- *Später werden wir noch weitere Blöcke für bestimmte Anweisungen kennenlernen.*

Beispiel 3.6 Die Methode `eintrittspreis` könnte als Struktogramm so aussehen:



#

4 if-else

Beispiel 4.1 In einem Freizeitpark kostet der Eintritt 20 € pro Person. Ab einer Gruppengröße von 10 erhält eine Person freien Eintritt.

Die entsprechende Methode könnte folgendermaßen implementiert werden:

```
1   double berechneGesamtpreis ( int anzahlPersonen ) {
2     if ( anzahlPersonen >= 10 ){
3       return ( anzahlPersonen - 1 ) * 20;
4     } else {
5       return anzahlPersonen * 20;
6     }
7   }
```

#

Information 4.2 Die **Bedingten Anweisung** erlaubt es, dass bestimmte Anweisungen nur dann ausgeführt werden, falls eine Bedingung erfüllt ist:

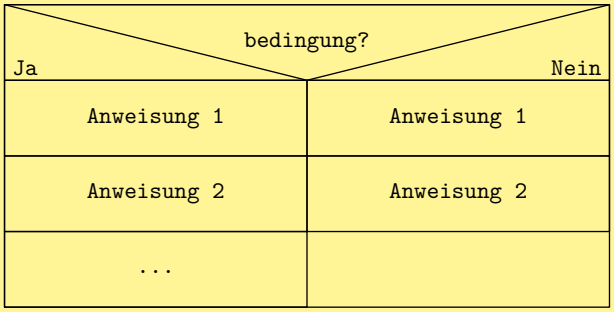
```
1   if ( bedingung ) {
2     Anweisung 1;
3     Anweisung 2;
4     ...
5   }
```

Mit Hilfe der **Verzweigung** kann eine bedingte Anweisung um einen zweiten Code-Block erweitert werden, der nur dann ausgeführt wird, falls die Bedingung *nicht* erfüllt ist:

```
1   if ( bedingung ) {
2     //wird ausgeführt, wenn die Bedingung erfüllt ist
3     Anweisung 1;
4     Anweisung 2;
5     ...
6   } else {
7     //wird ansonsten ausgeführt
8     Anweisung 1;
9     Anweisung 2;
10    ...
11  }
```

Für bedingte Anweisungen und Verzweigungen gibt es eine eigene Darstellung im Struktogramm:

Information 4.3 Eine Verzweigung wird im Struktogramm folgendermaßen dargestellt:



Für bedingte Anweisungen gibt es keine andere Darstellung, man lässt einfach den »Nein«-Teil weg.

Beispiel 4.4 Die Methode `boolean pruefePasswort ( passwort )` soll überprüfen, ob das `passwort` gleich dem geheimen Passwort (»geheim«) ist und `true` zurückgeben, wenn es gleich ist und ansonsten `false`.

```
1   boolean pruefePasswort ( passwort ) {
2     if ( passwort == "geheim" ) {
3       return true;
4     } else {
5       return false;
6     }
7   }
```

#

Information 4.5 In Bedingungen kann man die folgenden Operatoren verwenden:

- `a == b` sind `a` und `b` gleich?
- `a <= b` ist `a` kleiner als oder gleich `b`?
- `a != b` sind `a` und `b` ungleich?
- `a > b` ist `a` größer als `b`?
- `a < b` ist `a` kleiner als `b`?
- `a >= b` ist `a` größer als oder gleich `b`?

Diese Operatoren funktionieren auch mit Strings.

Achtung: Zum Vergleichen muss `==` verwendet werden, nicht `=`, denn dies ist eine Zuweisung!

Beispiel 4.6

Es soll ein einfacher Taschenrechner programmiert werden. Dabei kann der User zwei Zahlen in die beiden Textfelder eingeben und anschließend einen der beiden Buttons klicken. Dann soll die entsprechende Rechenoperation ausgeführt und das Ergebnis ausgegeben werden.

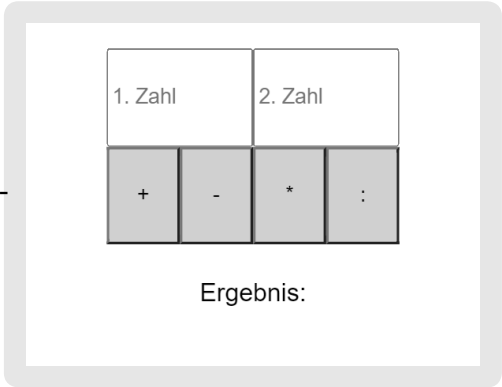


Abbildung 9

Der Code dafür könnte folgendermaßen aussehen:

```
1   class Taschenrechner {
2     UI ui;
3     void onStart( ) {
4       ui = new UI( );
5     }

6     void onAction( JComponent trigger ) {
7       double a = ui.zahl1.getValue( );
8       double b = ui.zahl2.getValue( );
9       double ergebnis;
10      if ( trigger == ui.plus ) {
11        ergebnis = a + b;
12      }
13      if ( trigger == ui.minus ) {
14        ergebnis = a - b;
15      }
16      if ( trigger == ui.mal ) {
17        ergebnis = a * b;
18      }
19      if ( trigger == ui.geteilt ) {
20        ergebnis = a / b;
21      }
22      ui.ergebnis.setValue( ergebnis );
23    }

24    public static void main( String[ ] args ) {
25      new Taschenrechner( );
26    }
27  }
```

#

Information 4.7 Mit Hilfe der Operatoren `&&` (»und«) und `||` (»oder«) ist es möglich, mehrere Bedingungen miteinander zu verknüpfen:

```
1  if ( bedingung1 && bedingung2 ) {
2    //erfüllt, wenn beide Bedingungen erfüllt sind
3  }
4  if (bedingung1 || bedingung2) {
5    //erfüllt, wenn mindestens eine der beiden Bedingungen erfüllt ist
6  }
```

Beispiel 4.8 Die folgende Methode prüft, ob Nutzername und Passwort korrekt sind:

```
1  boolean pruefeNutzernameUndPasswort ( String name, String passwort ) {
2    if ( name == "Lisa" && passwort == "geheim" ) {
3      return true;
4    } else {
5      return false;
6    }
7  }
```

Um mehrere Kombinationen zu prüfen, könnte man folgenden Code verwenden:

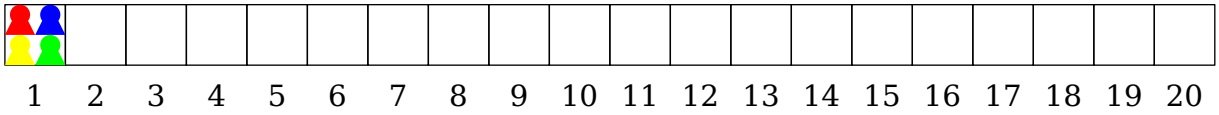
```
1  boolean pruefeNutzernameUndPasswort ( String name, String passwort ) {
2    if ( ( name == "Lisa" && passwort == "geheim" ) || ( name == "Richi" && passwort == "Teddy" ) ) {
3      return true;
4    } else {
5      return false;
6    }
7  }
```

#

5 Arrays

In vielen Fällen fängt man beim Programmieren damit an, nummerierte Variablen zu verwenden.

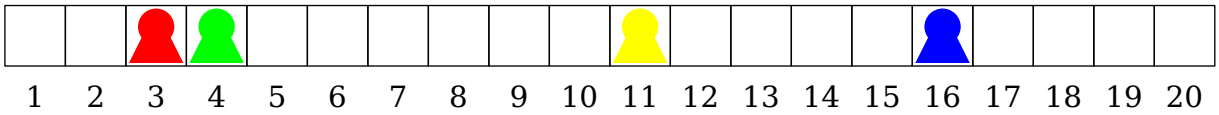
Beispiel 5.1 In einem vereinfachten Mensch-ärgere-dich-nicht sind alle Felder durchnummeriert von 1 bis 20. Jeder Spieler hat nur eine Figur und alle Figuren starten auf Feld 1:



Die Position der Figuren wird also durch eine ganze Zahl beschrieben. Wenn vier Spieler teilnehmen, muss es also vier globale Variablen geben (für jeden Spieler eine), die alle beim Wert 1 starten:

```
1   int spieler1 = 1;
2   int spieler2 = 1;
3   int spieler3 = 1:
4   int spieler4 = 1;
```

Im Laufe des Spiels ändern sich dann die Variablen, wenn eine Figur gezogen wird. In der folgenden Spielsituation



haben die Variablen die Werte

`spieler1` (rot) = 3 , `spieler2` (blau) = 16, `spieler3` (gelb) = 11 und `spieler4` (grün) = 4.

Damit das Spiel funktioniert, muss es eine weitere globale Variable `int amZug` geben, die speichert, welcher Spieler gerade am Zug ist.

Abhängig von dieser Variablen wird dann festgestellt, welche Figur bewegt werden muss, in etwa so:

```
1   int w = App.random(1, 6);
2   if ( amZug == 1 ){
3     spieler1 = spieler1 + w;
4   }
5   if ( amZug == 2 ){
6     spieler2 = spieler2 + w;
```

```
7   }
8   if ( amZug == 3 ){
9     spieler3 = spieler3 + w;
10  }
11  if ( amZug == 4 ){
12    spieler4 = spieler4 + w;
13  }
```

Im Prinzip muss man also allen Code vier Mal schreiben. Das muss anders gehen! #

In diesem Abschnitt lernen wir die »Arrays« kennen, die es uns erlauben, beliebig viele Werte desselben Datentyps zu einer Liste zusammenzufassen, die dann in einer einzigen Variablen gespeichert werden kann.

5.1 Array, Länge, Index

Definition 5.2 *Ein* **Array** *(mit Betonung auf der zweiten Silbe: »arRAY«, englisch für »Anordnung«; im Deutschen auch* **Feld***) ist eine Datenstruktur, die mehrere Werten* desselben Datentyps *enthält. Ein Array vom Typ* `int` *ist also eine Liste von ganzen Zahlen, ein Array vom Typ* `String` *ist dagegen eine Liste von Zeichenketten, z. B. eine Vokabelliste.*

Beispiel 5.3 Das Array `{"bike","table","school","friend"}` enthält die 4 Strings `"bike"`, `"table"`, `"school"` und `"friend"`. #

Beispiel 5.4 Das Array `{1,"2",true}` ist nicht zulässig, da die einzelnen Werte unterschiedliche Datentypen haben. #

Definition 5.5 *Die* **Länge** *eines Arrays entspricht der Anzahl der Werte, die das Array enthält.*

Beispiel 5.6 Das Array `{"bike","table","school","friend"}` aus dem Beispiel hat die Länge 4. #

Definition 5.7 *Die Werte innerhalb eines Arrays werden durchnummeriert, wobei die Nummerierung bei* 0 *beginnt! Die Position eines Wertes in einem Array wird als* **Index** *bezeichnet.*

(1) Das 1. Element eines Arrays hat den Index 0*.*

(2) Das 2. Element eines Arrays hat den Index 1*.*

(3) Das 3. Element eines Arrays hat den Index 2.

(4) ...

Beispiel 5.8 Im Array `{"bike","table","school","friend"}` hat `"table"` den Index 1 und `"friend"` den Index 3. #

5.2 Deklarieren und Initialisieren von Array-Variablen

Information 5.9 Eine Variable, die ein Array eines bestimmten Datentyps `Typ` speichern kann, hat den Datentyp `Typ[]`.

Beispiel 5.10 Mit der Deklaration

```
String[] namen;
int[] punkte;
```

erhält man zwei Variablen `namen` und `punkte`. Die Variable `namen` kann ein Array von Strings enthalten, die Variable `punkte` ein Array von ganzen Zahlen. #

Information 5.11 Es gibt zwei Möglichkeiten, ein neues Array zu erzeugen:

(1) Mit

```
new Typ[LAENGE]
```

erzeugt man ein neues Array vom Datentyp `Typ` der Länge `LAENGE`. Das Array wird automatisch mit Standardwerten befüllt.

(2) Mit

```
new Typ[]{ wert1, wert2, wert3, ... }
```

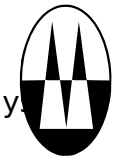
erzeugt man ein neues Array, das direkt die angegebenen Werte enthält.

In beiden Fällen ist die Länge des Arrays festgelegt und kann nicht mehr verändert werden.

Beispiel 5.12 Mit

```
1  String[] englisch = new String[]{ "bike", "table", "school", "friend" };
2  String[] deutsch = new String[]{ "Fahrrad", "Tisch", "Schule", "Freund" };
3  int[] punkte = new int [ 7 ];
```

erzeugen wir zwei `String`-Arrays sowie ein neues `int`-Array `{0, 0, 0, 0, 0, 0, 0}` mit 7 Nullen. #



Bemerkung 5.13 Der Operator `new` wird uns im Rahmen der Objektorientierung in der Q1 wieder begegnen. Allgemein reserviert `new` Platz im Arbeitsspeicher des Computers.

Mit `new double[8]` wird der Computer also angewiesen, 8 aufeinanderfolgende Speicherstellen für `double`-Werte zu reservieren. #

5.3 Zugriff auf die Elemente und die Länge eines Arrays

Information 5.14 Es sei `a` ein Array.

Mit `a.length` erhalten wir die Länge des Arrays.

Mit `a[ i ]` erhalten wir den Wert des Arrays an der Stelle (dem Index) `i`.

Achtung: Die Nummerierung beginnt bei Index 0!

Achtung: Wenn man versucht, auf einen Index zuzugreifen, der größer oder gleich als die Länge des Arrays ist, stürzt das Programm ab.

Beispiel 5.15

```
1   String[] englisch = new String[]{ "bike", "table", "school", "friend" };
2   String[] deutsch = new String[]{ "Fahrrad", "Tisch", "Schule", "Freund" };
3   int[] punkte = new int [ 6 ]; // {0, 0, 0, 0, 0, 0}
4   System.out.println( englisch.length ); // Ausgabe: 4
5   System.out.println( englisch[ 0 ] ); // Ausgabe: bike
6   System.out.println( punkte[ 2 ] ); // Ausgabe: 0
7   System.out.println( englisch[ 4 ] ); // Fehler! 4 ist kein zulässiger Index
8   punkte[ 1 ] = 2; // {0, 2, 0, 0, 0, 0}
9   punkte[ 3 ] = punkte[ 3 ] + 1; // {0, 2, 0, 1, 0, 0, 0}
10  System.out.println( punkte[ 1 ] ); //Ausgabe: 2
```

#

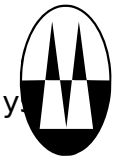
Beispiel 5.16 Wir nutzen Arrays, um unser Mensch-ärgere-dich-nicht aus dem Eingangsbeispiel eleganter und ohne Code-Dopplungen zu schreiben:

(1) Wir ersetzen die vier Variablen `spieler1`, ..., `spieler4` vom Typ `int` durch eine einzelne Variable `spieler` vom Typ `int[]`, also ein Array von ganzen Zahlen:

```
1   int spieler1 = 1;
2   int spieler2 = 1;
3   int spieler3 = 1:
4   int spieler4 = 1;
```

⇒

```
1   int[] spieler = new int[]{1, 1, 1, 1};
```



(2) Der Zug der Spielfigur kann nun drastisch vereinfacht werden: Anstatt per `if` zu prüfen, welcher Spieler am Zug ist, greifen wir einfach direkt auf die korrekte Stelle im Array zu und ändern diese:

```
1   int w = App.random(1, 6);
2   if ( amZug == 1 ){
3     spieler1 = spieler1 + w;
4   }
5   if ( amZug == 2 ){
6     spieler2 = spieler2 + w;
7   }
8   if ( amZug == 3 ){
9     spieler3 = spieler3 + w;
10  }
11  if ( amZug == 4 ){
12    spieler4 = spieler4 + w;
13  }
```

⇒

```
1   int w = App.random(1, 6);
2   spieler [ amZug - 1 ] = spieler [ amZug -1 ] + w;
```

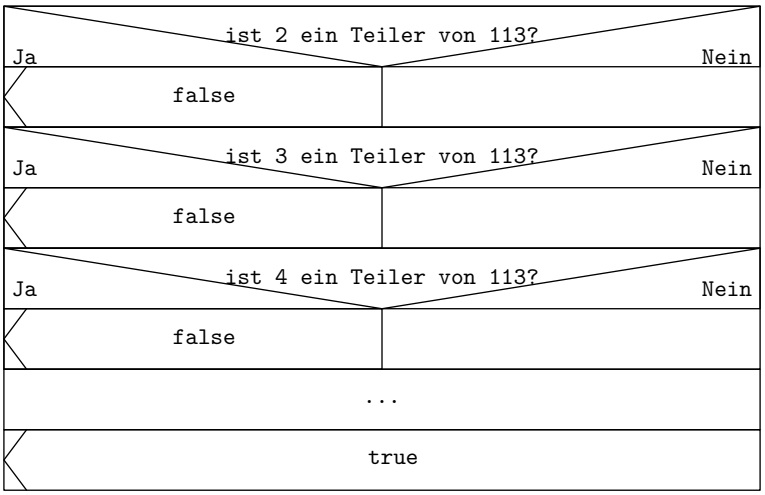
#

Bemerkung 5.17 Es gibt noch viele weitere mögliche Anwendungen von Arrays, wie wir in verschiedenen Übungsaufgaben sehen werden. Ihr wahres Potenzial entfalten Arrays aber erst im Zusammenhang mit Schleifen (siehe nächstes Kapitel). #

6 Schleifen

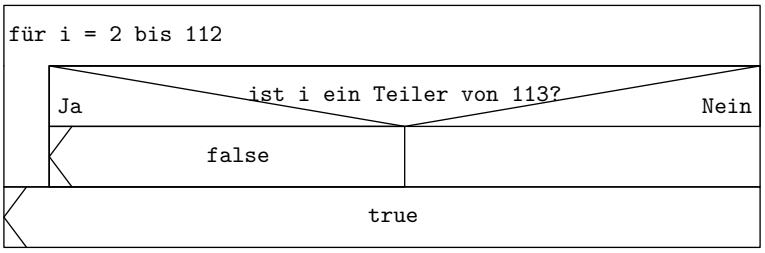
Wenn wir einen Computer programmieren, schreiben wir Anweisungen, die dieser dann nacheinander abarbeitet. Bisher mussten wir für jede Tätigkeit des Rechners eine Anweisung schreiben. Das führt manchmal zu sehr langen Programmen:

Beispiel 6.1 Wir wollen per Programm herausfinden, ob 113 eine Primzahl ist. Dazu müssen wir nur für alle Zahlen von 2 bis 112 ausprobieren, ob man 113 durch diese ohne Rest teilen kann. Falls ja, so ist es keine Primzahl (`false`). Andernfalls ist es eine (`true`).



#

Offensichtlich funktioniert dies zwar, es führt aber zu unglaublich langem Code. Schöner wäre es, wenn wir Anweisungen mehrfach ausführen könnten, in einer Art »Schleife«:



Definition 6.2 *Eine* **Schleife** *wiederholt Anweisungen mehrfach.*

Wir betrachten zwei Sorten von Schleifen: Die `FOR`-Schleife, bei der die Anzahl der Wiederholungen von Beginn an festgelegt wird, sowie die `WHILE`-Schleife, die so lange läuft, wie eine gewisse Bedingung erfüllt ist.

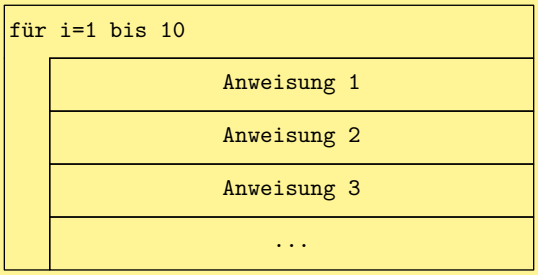
6.1 Die FOR-Schleife

Information 6.3 Bei der `for`-Schleife wird die Anzahl der der Wiederholungen fest vorgegeben. Jede `for`-Schleife enthält eine sog. **Zählvariable** (meistens `i`, `j` oder `k` genannt), die bei einem gewissen Wert startet und *nach jedem Durchgang* um 1 erhöht wird:

```
     Zaehl-  Start-  Maximal- Erhoehe
    variable  wert     wert    um 1
      ⇓⇓⇓     ⇓        ⇓      ⇓⇓⇓
for ( int i  = 1;  i <= 10;    i++ ){
  //Anweisungen, die 10 mal ausgeführt werden (fuer i = 1, 2, 3, ..., 9, 10)
}
```

Die Zählvariable ist eine lokale Variable, die nur innerhalb der Schleife definiert ist.

Die `for`-Schleife wird im Struktogramm folgendermaßen umgesetzt:



Beispiel 6.4 Bart Simpson muss im Vorspann jeder Folge 100 mal einen bestimmten Satz an die Tafel schreiben. Das folgende Programm erledigt das für ihn:

```
1  for ( int i = 1; i <= 100; i++ ){
2    System.out.println("Ich darf nicht im Flur Skateboard fahren.");
3  }
```
#

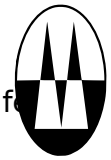
Beispiel 6.5 Es ist sehr praktisch, dass bei einer `for`-Schleife direkt eine Variable mitgezählt wird. Damit können wir z. B. eine Liste von Quadratzahlen ausgeben:

```
1  for ( int i = 5; i <= 100; i++ ){
2    System.out.println( i*i );
3  }
```

Dieses Programm gibt alle Quadratzahlen von $5^2 = 25$ bis $100^2 = 10.000$ aus. #

Bemerkung 6.6 Die oben angegebenen Informationen sind nicht ganz vollständig:

- Der zweite Eintrag der Schleife (also z. B. `i <= 10`) ist eine beliebige Bedingung. Solange diese Bedingung erfüllt ist, läuft die Schleife weiter.



- Der dritte Eintrag der Schleife (z. B. `i++`) kann eine beliebige Anweisung sein, die nach jedem Durchgang ausgeführt wird.

Damit kann man z. B. eine Schleife bauen, die von 12, in 2-er-Schritten rückwärts bis 4 zählt:

```
1  for ( int i = 12; i >= 4; i = i - 2 ){
2    System.out.println( i );
3  }
```
#

Beispiel 6.7 Das folgende Programm addiert alle Zahlen von 1 bis 100:

```
1  int sum = 0;
2  for ( int i = 1; i <= 100, i++ ){
3    sum = sum + i;
4  }
```

Ausgeschrieben bedeutet dies:

```
1  int sum = 0;
2  sum = sum + 1;
3  sum = sum + 2;
4  sum = sum + 3;
5  ...
6  sum = sum + 100;
```
#

Information 6.8 Ihr wahres Potenzial entfalten `for`-Schleifen im Zusammenspiel mit Arrays. Wenn `array` ein Array ist, kann man mit dem folgenden Code alle Einträge des Arrays durchgehen:

```
for ( int i = 0; i < array.length; i++ ){
  //mache etwas mit array[ i ]
}
```

Beispiel 6.9 Wir wollen zählen, wie viele Zahlen im Array `zahlen` größer als 10 sind:

```
1  int anzahl = 0;
2  for ( int i = 0; i < zahlen.length; i++ ){
3    if ( zahlen[ i ] > 10 ) {
4      anzahl++;
5    }
6  }
```
#

6.2 Die WHILE-Schleife

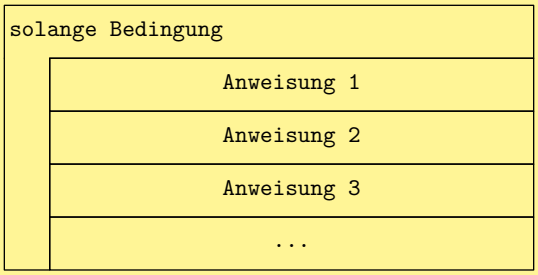
Information 6.10 Die `while`-Schleife wiederholt ihre Anweisungen, so lange eine bestimmte Bedingung erfüllt ist:

```
while( bedingung ){
  //Anweisungen, die so lange wiederholt werden, wie die bedingung erfüllt ist
}
```

Die Bedingung kann genau wie die Bedingungen in einer `if-else`-Abfrage formuliert werden.

Achtung: Wenn man nicht vorsichtig ist, kann eine `while`-Schleife zu einer Endlosschleife werden (siehe unten)!

Im Struktogramm sieht eine `while`-Schleife fast wie eine `for`-Schleife aus:



Beispiel 6.11 Die folgende Schleife addiert alle Zahlen von 1 bis 100:

```
1   int sum = 0;
2   int i = 1;
3   while ( i <= 10 ) {
4     sum = sum + i;
5     i = i + 1;
6   }
```
#

Bemerkung 6.12 Mit Hilfe der `while`-Schleife kann man also alles erreichen, was man auch mit einer `for`-Schleife erreichen kann. Die `while`-Schleife »kann also mehr« als die `for`-Schleife.

Warum verwendet man trotzdem in den meisten Fällen `for`-Schleifen? Weil es bei `while`-Schleifen schnell passieren kann, dass man in einer Endlos-Schleife landet:
Wenn wir im vorangegangenen Beispiel die Zeile `i = i + 1;` vergessen hätten, also

```
1   int i = 1;
2   int sum = 0;
3   while ( i <= 10 ) {
4     sum = sum + i;
5   }
```

geschrieben hätten, so wären wir in einer Endlosschleife gelandet, weil `i` für alle Ewigkeiten den Wert `1` behalten würde. #

Bemerkung 6.13 Die `while`-Schleife wird nur an sehr wenigen Stellen benötigt. In 99 % aller Fälle reicht eine einfache `for`-Schleife. #

Beispiel 6.14 Wir wollen zwei zufällige Zahlen im Bereich von 1 bis 100 erzeugen, die um mehr als 10 auseinanderliegen (13 und 56 wären z. B. okay, nicht aber 77 und 72).

```
1   int a,b;
2   boolean okay=false;
3   while( okay==false ){
4     a = App.random(1,100);
5     b = App.random(1,100);
6     if ( Math.abs( a - b ) > 10 ){
7       okay = true;
8     }
9   }
```

Wir generieren also zwei Zufallszahlen und schauen dann, ob ihr Abstand größer als 10 ist. In diesem Fall setzen wir `okay` auf `true`, sodass die Schleife verlassen wird.

Anmerkung: `Math.abs` berechnet den **Absolutbetrag** einer Zahl, das ist die Zahl ohne Vorzeichen (Minus wird zu Plus). Z. B. `Math.abs ( - 6 )` $= |-6| = 6$. #

6.3 `break` und `continue`

Im Zusamenhang mit Schleifen gibt es zwei weitere Schlüsselwörter, die das Leben deutlich einfacher machen können:

Information 6.15 Innerhalb einer Schleife kann man die speziellen Anweisungen `break` und `continue` verwenden:

- Mit der Anweisung `break;` verlässt man eine Schleife.
- Mit der Anweisung `continue;` geht man direkt zum nächsten Schleifendurchgang weiter.

Beispiel 6.16 Damit kann man das letzte Beispiel auch so umsetzen:

```
1   int a,b;
2   while( true ){
3     a = App.random(1,100);
4     b = App.random(1,100);
5     if ( Math.abs( a - b ) > 10 ){
6       break;
7     }
8   }
```

#

III

Algorithmik und Objektorientierung

1 Klassen und Objekte

1.1 Klassen als zusammengesetzte Datentypen

Beispiel 1.1 In einem Pokemon-Spiel kämpfen verschiedene Pokemon gegeneinander. Die Pokemon haben unterschiedliche Werte: *Leben* gibt an, wie viel sie aushalten, *Schaden*, wie viel Leben sie anderen Pokemons abziehen, wenn sie diese treffen. Außerdem haben sie noch eine *Schadensart* wie Feuer, Wasser oder Eis und eine *Schwäche*, die angibt, gegen welche Schadensart sie besonders empfindlich sind.

Der folgende Code zeigt eine mögliche Implementierung in Java:

```
1   class PokemonSpiel{
2     String nameSchiggy;
3     int lebenSchiggy;
4     int schadenSchiggy;
5     String schadensartSchiggy;
6     String schwaecheSchiggy;

7     String nameGlumanda;
8     int lebenGlumanda;
9     int schadenGlumanda;
10    String schadensartGlumanda;
11    String schwaecheGlumanda;

12    void onStart(){
13      nameSchiggy="Schiggy";
14      lebenSchiggy=70;
15      schadenSchiggy=20;
16      schadensartSchiggy="Wasser";
17      schwaecheSchiggy="Pflanze";

18      nameGlumanda="Glumanda";
19      lebenGlumanda=60;
20      schadenGlumanda=30;
21      schadensartGlumanda="Feuer";
22      schwaecheGlumanda="Wasser";
23    }
24  }
```

#

Bemerkung 1.2 Diese Art der Modellierung funktioniert zwar, sie ist aber ein absoluter Alptraum, denn für jedes Pokemon wird ein neues Set von 5 globalen Variablen benötigt.

Es muss also eine bessere Lösung geben. #

Information 1.3 Man kann in Java mehrere Variablen, die beliebige — auch unterschiedliche — Datentypen haben können, zu einem neuen Datentyp zusammenfassen.

Diese neuen Datentypen heißen **zusammengesetzte Datentypen** oder auch **Klassen**.

Beispiel 1.4 (Fortsetzung) In unserem Pokemon-Spiel können wir einen neuen Datentyp namens »Pokemon« definieren, der alle Eigenschaften eines Pokemons zusammenfasst:

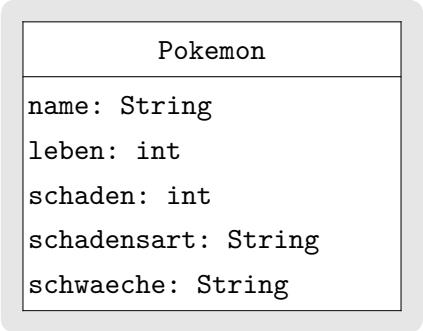


Abbildung 10

Das bedeutet, dass eine Variable vom Typ `Pokemon` fünf verschiedene Werte gleichzeitig speichern kann, nämlich drei Strings und zwei `int`s. #

Bemerkung 1.5 Ein Array speichert mehrere Werte desselben Typs während eine Klasse das Speichern mehrerer Werte verschiedener Typen erlaubt. #

Information 1.6 Die Darstellung aus **Abb. 10** nennt sich **UML-Klassendiagramm**.[5]

Im oberen Feld steht der Name der Klasse und im Feld darunter die sog. **Attribute**, aus denen der neue Datentyp zusammengesetzt ist (siehe unten).

Mehr zu UML in **Kapitel 6**.

[5] »UML« steht für »unified modelling language«, zu deutsch: »vereinheitlichte Modellierungssprache«.

1.2 Deklaration einer Klasse

Information 1.7 Eine Klasse wird in Java mit Hilfe des Schlüsselworts `class` definiert:

```
1  class NameDerKlasse{
2    Typ1 name1;
3    Typ2 name2;
4    Typ3 name3;
5    ...
6  }
```

Jede der Zeilen 2 bis 4 definiert ein **Attribut** dieser Klasse. Jedes Attribut besitzt einen Datentyp und einen Namen, unter dem es referenziert werden kann.

Beispiel 1.8 (Fortsetzung) Unsere Pokemon-Klasse würde folgendermaßen implementiert werden:

```
1  class Pokemon{
2    String name;
3    int leben;
4    int schaden;
5    String schadensart;
6    String schwaeche;
7  }
```

#

Bemerkung 1.9 Es hat sich eingebürgert, Klassennamen mit einem Großbuchstaben zu beginnen (`Pokemon`, `Auto`, `Person`, ...) und Attributsnamen mit einem Kleinbuchstaben (`leben`, `gewicht`, `farbe`, ...). #

1.3 Objekte einer Klasse erzeugen und verwenden

Information 1.10 Um eine Klasse verwenden zu können, muss man zunächst ein **Objekt** dieser Klasse erzeugen und in einer Variablen speichern:

```
NameDerKlasse variable = new NameDerKlasse();
```

Anschließend kann man auf die Attribute des Objekts zugreifen, indem man den **Punkt-Operator** verwendet:

```
variable.attribut = wert;
```

Beispiel 1.11 (Fortsetzung) Mit Hilfe unserer Klasse `Pokemon` können wir unser Programm nun folgendermaßen aufbauen:

```
1   class PokemonSpiel{
2     Pokemon schiggy;
3     Pokemon glumanda;

4     onStart(){
5       schiggy = new Pokemon();
6       schiggy.name = "Schiggy";
7       schiggy.leben = 70;
8       schiggy.schaden = 20;
9       schiggy.schadensart = "Wasser";
10      schiggy.schwaeche = "Pflanze";

11      glumanda = new Pokemon();
12      glumanda.name = "Glumanda";
13      glumanda.leben = 60;
14      glumanda.schaden = 30;
15      glumanda.schadensart = "Feuer";
16      glumanda.schwaeche = "Wasser";
17    }
18  }
```

In den Zeilen 2 und 3 werden globale Variablen `schiggy` und `glumanda` deklariert, die beide den (neuen) Datentypen `Pokemon` haben.

In Zeile 5 wird ein neues Objekt vom Typ `Pokemon` erzeugt und in der Variablen `schiggy` gespeichert.

In Zeile 6 weisen wir dem Attribut `name` des Objektes `schiggy` den Wert `"Schiggy"` zu. In den folgenden Zeilen werden die weiteren Attribute von `schiggy` mit Werten befüllt.

In den Zeilen 11 bis 16 geschieht dann dasselbe für das Objekt `glumanda`. #

Bemerkung 1.12 Beachte, dass die Variablen `schiggy.leben` und `glumanda.leben` völlig voneinander unabhängig sind. Jedes Objekt der Klasse `Pokemon` hat sein eigenes Set aus fünf Variablen. #

Bemerkung 1.13 Etwas lästig ist, dass wir die Zeilen 6 bis 10 sowie 12 bis 16 im Prinzip doppelt schreiben mussten. Im nächsten Kapitel werden wir mit dem »Konstruktor« eine Möglichkeit kennenlernen, wie dies drastisch vereinfacht werden kann. #

2 Der Konstruktor einer Klasse

2.1 Was ist ein Konstruktor?

Beispiel 2.1 Im letzten Kapitel haben wir den Code

```
1   schiggy = new Pokemon();
2   schiggy.name = "Schiggy";
3   schiggy.leben = 70;
4   schiggy.schaden = 20;
5   schiggy.schadensart = "Wasser";
6   schiggy.schwaeche = "Pflanze";
```

verwendet, um ein neues Objekt zu erzeugen und anschließend die Attribute des neuen Objekts mit Werten zu belegen.

Wir können den Code drastisch vereinfachen, indem wir einen sog. »Konstruktor« für die Klasse implementieren. #

Information 2.2 Der **Konstruktor** einer Klasse ist eine Methode, die innerhalb dieser Klasse deklariert wird und die immer dann aufgerufen wird, wenn ein neues Objekt dieser Klasse erzeugt wird.

Diese Methode hat zwei Besonderheiten:

- Sie hat keinen Rückgabetyp.
- Sie heißt automatisch immer so wie die Klasse.

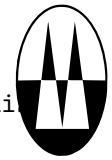
Im UML-Klassendiagramm wird der Konstruktor in einem weiteren Rechteck unterhalb der Methoden aufgeführt (siehe nächstes Beispiel).

Beispiel 2.3 Für unsere Klasse `Pokemon` wäre folgender Konstruktor sinnvoll:

```
1   Pokemon(String nam, int leb, int schad, String sart, String schw){
2     name = nam;
3     leben = leb;
4     schaden = schad;
5     schadensart = sart;
6     schwaeche = schw;
7   }
```

Damit ist es möglich, den Code aus **Beispiel 2.1** auf eine Zeile zu verkürzen:

```
1   Pokemon schiggy = new Pokemon("Schiggy", 70, 20, "Wasser", "Pflanze");
```



Das UML-Klassendiagramm sieht dann folgendermaßen aus:

Pokemon
name: String leben: int schaden: int schadensart: String schwaeche: String
Pokemon(nam: String, leb: int, schad: int, sart: String, schw: String)

#

2.2 Das Schlüsselwort `this`

Beispiel 2.4 In obigem Beispiel wurden sehr seltsame Namen für die Parameter verwendet, z. B. `nam` anstelle von `name`.

Wenn wir den Parameter `name` nennen würden, würde dieser genauso heißen wie das Attribut und wir müssten in Zeile 2 die Anweisung

```
1    name = name;
```

schreiben. Dies würde aber nicht funktionieren, da wir damit den Wert des Parameters `name` auf den Wert des Parameters `name` setzen würden (was gar nichts bewirken würde). Der Parameter `name` würde das Attribut `name` überdecken.

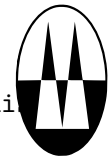
Wir können Java klarmachen, dass wir das Attribut meinen, indem wir `this.name` anstelle von `name` schreiben:

```
1    Pokemon(String name, int leben, int schaden, String schadensart, String schwaeche){
2      this.name = name;
3      this.leben = leben;
4      this.schaden = schaden;
5      this.schadensart = schadensart;
6      this.schwaeche = schwaeche;
7    }
```

Das bedeutet, dass das Attribut `name` auf den Wert des Parameters `name` gesetzt wird.

#

Information 2.5 Innerhalb des Konstruktors steht das Schlüsselwort `this` für das Objekt, das gerade erzeugt wird.



Beispiel 2.6 Zusammengefasst erhalten wir die Klasse `Pokemon` in Java

```
1   class Pokemon{
2     String name;
3     int leben;
4     int schaden;
5     String schadensart;
6     String schwaeche;

7     Pokemon(String name, int leben, int schaden, String schadensart, String schwaeche){
8       this.name = name;
9       this.leben = leben;
10      this.schaden = schaden;
11      this.schadensart = schadensart;
12      this.schwaeche = schwaeche;
13    }
14  }
```

und als UML-Klassendiagramm

Pokemon
name: String leben: int schaden: int schadensart: String schwaeche: String
Pokemon(name: String, leben: int, schaden: int, schadensart: String, schwaeche: String)

#

3 Verwalten von Objekten in Arrays

3.1 Arrays von Objekten

Beispiel 3.1 In unserem `PokemonSpiel` wollen wir nun viele verschiedene Pokemon einbauen. Der Spieler soll eines dieser Pokemon auswählen und damit gegen ein zufälliges Pokemon antreten. Außerdem soll es möglich sein, dass der Spieler weitere Pokemons hinzufügen kann. #

Information 3.2 In vielen Anwendungen müssen viele Daten so gespeichert werden, dass man sie sinnvoll verwenden kann. Dazu brauchen wir eine geeignete **Datenstruktur**.

Die wichtigste Datenstruktur ist das **Array von Objekten**. In einem solchen Array kann man man beliebig viele Objekte derselben Klasse speichern.

Beispiel 3.3 Um die vielen Pokemon zu speichern, deklarieren wir in der Hauptklasse `PokemonSpiel` ein Array vom Datentyp `Pokemon` sowie einen `int anzahlPokemon`:

```
1  class PokemonSpiel{
2    int anzahlPokemon;
3    Pokemon[] pokemonListe;
4  }
```

In der `onStart`-Methode erzeugen wir nun ein neues Array mit 30 Plätzen und fügen einige Pokemons hinzu. `anzahlPokemon` setzen wir auf den richtigen Wert:

```
1  public void onStart(){
2    pokemonListe = new Pokemon[30];
3    pokemonListe[0] = new Pokemon("Bisasam", 90, 15, "Pflanze", "Feuer");
4    pokemonListe[1] = new Pokemon("Glumanda", 60, 30, "Feuer", "Wasser");
5    pokemonListe[2] = new Pokemon("Schiggy", 70, 20, "Wasser", "Pflanze");
6    anzahlPokemon = 3;
7  }
```

#

Information 3.4 Um ein Array von Objekten zu visualisieren, gibt es in JavaApp eine GUI-Komponente namens **DataTable**:

NAME	LEBEN	SCHADEN	SCHADENSART	SCHWAECHE
Bisasam	90	15	Pflanze	Feuer
Glumanda	60	30	Feuer	Wasser
Schiggy	70	20	Wasser	Pflanze

3.2 Hinzufügen neuer Objekte zu einem Array

Beispiel 3.5 Das Hinzufügen eines neuen Pokemon passiert per Klick auf einen Button, nachdem der Benutzer vorher die Daten des neuen Pokemon eingegeben hat.

In `onAction` erzeugen wir ein neues Pokemon mit den eingegebenen Werten und rufen anschließend die Methode `pokemonHinzufügen` mit dem neuen Pokemon als Argument auf.

Die Methode `pokemonHinzufuegen` speichert das neue Pokemon dann im Array `pokemonListe` an der ersten freien Stelle:

```
1   class PokemonSpiel{
2     ...
3     public void onAction(JComponent trigger){
4       ...
5       if(trigger == buttonHinzu){
6         Pokemon p = new Pokemon( inName.getValue(), inLeben.getValue(),
7             inSchaden.getValue(), inSchadensart.getValue(), inSchwaeche.getValue() );
8         pokemonHinzufuegen(p);
9       }
10      ...
11    }
12
13    void pokemonHinzufuegen(Pokemon neu){
14      pokemonListe[anzahlPokemon] = neu;
15      anzahlPokemon= anzahlPokemon + 1;
16    }
17  }
```

Dadurch, dass wir in der Variablen `anzahlPokemon` mitzählen, wie viele Pokemons aktuell im Array gespeichert sind, kennen wir immer die Position im Array, an der das nächste Pokemon hinzugefügt werden soll. #

3.3 Entfernen von Objekten aus einem Array

Beispiel 3.6 Das Entfernen eines Pokemons ist aufwändiger. Da keine Lücke im Array entstehen darf, müssen alle Pokemons hinter dem entfernten Pokemon um eine Position nach vorne rücken:

```
1   class PokemonSpiel{
2     ...
3     public void onAction(JComponent trigger){
4       ...
5       if(trigger == buttonEntfernen){
6         //aus der DataTable auslesen, welches Pokemon ausgewählt wurde:
7         int entfernen = tablePokemon.getValue();
8         if(entfernen >= 0){
9           pokemonEntfernen(entfernen);
10        }else{
11          App.alert("Du hast keinen Pokemon ausgewählt.");
12        }
13      }
14    }
15
16    void pokemonEntfernen(int index){
17      //alle Pokemon hinter 'index' werden um 1 nach vorne verschoben:
18      for(int i = index + 1; i < anzahlPokemon; i++){
19        pokemonListe[i-1] = pokemonListe[i];
20      }
21      pokemonListe[anzahlPokemon-1] = null;
22      anzahlPokemon = anzahlPokemon - 1;
23    }
24  }
```

Die Anweisung in Zeile 20 löscht dabei den letzten Eintrag im Array, indem es das Pokemon mit `null` überschreibt. #

Bemerkung 3.7 Wenn die Reihenfolge der Pokemons im Array keine Rolle spielt, kann man auch einfach das letzte Pokemon an die Stelle des zu entfernenden Pokemons verschieben und benötigt dann keine `for`-Schleife:

```
1   void pokemonEntfernen(int index){
2     if(index >= 0){
3       pokemonListe[index] = pokemonListe[anzahlPokemon-1];
4       pokemonListe[anzahlPokemon-1] = null;
5       anzahlPokemon = anzahlPokemon - 1;
6     }
7   }
```

#

Information 3.8 Der Wert `null` steht in Java für das nicht-vorhandene Objekt.

Wenn ein neues Array erzeugt wird, das Objekte speichern kann, so werden zunächst alle Plätze mit `null` gefüllt.

3.4 Zufällige Auswahl in einem Array

Beispiel 3.9 Zu guter Letzt kümmern wir uns noch darum, dass der Benutzer ein Pokemon für den Kampf auswählen kann und dass der Computer ein zufälliges Pokemon wählt:

```
1   class PokemonSpiel{
2     Pokemon spieler;
3     Pokemon computer;
4     ...
5     public void onAction(JComponent trigger){
6       ...
7       if(trigger == kampfStart){
8         int index = tablePokemon.getValue();
9         if(index >= 0){
10          starteKampf(pokemonListe[index]);
11        }else{
12          App.alert("Du musst erst ein Pokemon auswählen!")
13        }
14      }
15    }
16    void starteKampf(Pokemon pokemonFuerSpieler){
17      spieler = pokemonFuerSpieler;
18      computer = pokemonListe[App.random(0,pokemonListe.length-1)];
19      //...
20    }
21  }
```

#

4 Klassen mit Methoden

Beispiel 4.1 Wir wollen nun in unsere Pokemon-Arena einbauen, dass ein Pokemon ein anderes Pokemon angreifen kann. Eine passende Methode könnte so aussehen:

```
1   class PokemonSpiel{
2     ...
3     void angreifen(Pokemon angreifer, Pokemon ziel){
4       int schaden = angreifer.schaden + Blox.getRandomInt(0,20);
5       if(angreifer.schadensart == ziel.schwaeche){
6         schaden = schaden*2;
7       }
8       ziel.leben = ziel.leben-schaden;
9     }
10  }
```

Diese Vorgehensweise hat aber ein großes Problem: Wenn sich der Aufbau der Klasse `Pokemon` irgendwann später einmal ändern sollte, weil z. B. neue Attribute hinzugefügt werden oder Attribute umbenannt werden, dann funktioniert der obige Code nicht mehr.

Daher verlegen wir die Methoden, die Objekte einer bestimmten Klasse betreffen, *in diese Klasse*. #

Information 4.2 Eine Klasse kann eigene Methoden implementieren. Um eine Methode für ein bestimmtes Objekt `objekt` aufzurufen, schreibt man

```
objekt.methode(Parameter);
```

Beim Aufrufen einer Methode innerhalb der eigenen Klasse kann man das Objekt weglassen und einfach

```
methode(Parameter);
```

schreiben.

Beispiel 4.3 Die obige `angreifen`-Methode von `PokemonSpiel` könnten wir folgendermaßen in die Klasse `Pokemon` verlegen:

```
1   class Pokemon{
2     //Attribute
3     //Konstruktor

4     void greifeAn(Pokemon ziel){
5       int schaden = this.schaden + App.random(0,10);
```

```
6         if(schadensart == ziel.schwaeche){
7           schaden = schaden*2;
8         }
9         ziel.leben = ziel.leben-schaden;
10      }
11    }
```

Es fällt auf, dass der Code etwas kürzer geworden ist, weil er aus der Sicht des Angreifer-Pokemons formuliert ist.

Innerhalb von `PokemonSpiel` kann man diese Methode bspw. so benutzen:

```
1   class PokemonSpiel{
2     Pokemon spieler;
3     Pokemon computer;
4     ...
5     public void onAction(JComponent trigger){
6       ...
7       if(trigger == buttonAngriff){
8         spieler.greifeAn(computer);
9       }
10      ...
11    }
12  }
```

In Zeile 8 wird also die Methode `greifeAn` für das Objekt `spieler` ausgeführt und das Objekt `computer` als Argument übergeben. D.h., das Spieler-Pokemon greift das Pokemon des Computers an. #

Information 4.4 Die Entscheidung, welcher Klasse man eine bestimmte Methode zuordnet, ist oft nicht ganz einfach. Häufig helfen die folgenden Überlegungen bei der Entscheidung:

- Eine Methode erlaubt es, einem Objekt einer Klasse einen Befehl zu geben. Daher sollte die Klasse die Methode erhalten, bei der der Befehl passt und sinnvoll ist.
- Eine Methode sollte von der Klasse implementiert werden, die geradeso noch alle Informationen dazu zur Verfügung hat.

Beispiel 4.5

(a) Ein Pokemon hat die Eigenschaften Name, Leben, Schaden usw.. Die Handlungsoptionen eines Pokemons dagegen sind z. B. Angreifen, Heilen, Zaubern usw. Darum sollten diese Methoden der Klasse `Pokemon` zugeordnet werden.

(b) Das Hinzufügen neuer Pokemons zum Spiel dagegen kann gar nicht der Klasse `Pokemon` zugeordnet werden, da diese überhaupt nicht über die notwendigen Mittel dazu verfügt, denn ein Pokemon hat keinen Zugriff auf das Array `pokemons`. #

Bemerkung 4.6 Zusammengefasst: Eine Klasse sollte alle Methoden, die zum Gebrauch dieser Klasse benötigt werden, selbst implementieren. #

5 Das Geheimnisprinzip

5.1 Die Schlüsselwörter `private` und `public`

Information 5.1 Das Geheimnisprinzip ist ein Design-Prinzip aus der Produktentwicklung. Es besagt: Alles, womit der Benutzer nicht interagieren soll, soll vor diesem verborgen bleiben.

Beispiel 5.2 Ein Wasserspender sollte so designt sein, dass es einen Wasserhahn gibt, an dem man das Wasser zapfen kann. Da der Benutzer nicht auf die Idee kommen sollte, die Wasserleitung vom Hahn zu trennen und sich daraus direkt Wasser zu zapfen, sollte die Wasserleitung so verbaut sein, dass dies nicht möglich ist. #

Information 5.3 In Java können Attribute und Methoden als `public` oder als `private` deklariert werden:

(1) Attribute/Methoden, die `public` (öffentlich) sind, können »ganz normal« von außerhalb ihrer Klasse verwendet, verändert und/oder aufgerufen werden.

(2) Attribute/Methoden, die `private` (privat) sind, sind außerhalb ihrer Klasse unsichtbar.

Wird ein Attribut / eine Methode weder als `public` noch als `private` deklariert, ist es / sie automatisch `public`.[6]

Information 5.4 Auf Software-Entwicklung angewandt, bedeutet das Geheimnisprinzip:

(1) Die Attribute einer Klasse sollten außerhalb der Klasse nicht sichtbar und nicht veränderbar sein (`private`). Nur das Objekt selbst darf seine eigenen Attribute ändern.

(2) Die Methoden sollten im Normalfall von außen sichtbar sein (`public`). Methoden, die nur intern in der Klasse benötigt werden, sollten von außen unsichtbar sein (`private`).

[6] Das ist nicht 100 %-ig korrekt. Ein solches Attribut / eine solche Methode ist dann **package-private**. Mehr Informationen dazu gibt es online. Für unsere Zwecke gibt es keinen Unterschied zwischen package-private und public.

Information 5.5 Im UML-Klassendiagramm wird öffentlichen Attributen/Methoden ein »+« vorangestellt. Private Attribute/Methoden erhalten ein »−«.

Beispiel 5.6 Für ein Kartenspiel soll ein Kartenstapel implementiert werden. Dieser soll verschiedene Methoden besitzen: `karteZiehen`, `karteAblegen` und `mischen`. Für das Mischen wird außerdem eine Methode `vertauscheKarten` benötigt.

Insgesamt erhalten wir folgendes Diagramm für die Klasse `Kartenstapel`:

Kartenstapel
-karten: Karte[] -anzahlKarten: int
+Kartenstapel() +karteZiehen(): Karte +karteAblegen(k: karte) +mischen() -vertauscheKarten(index1: int, index2: int)

Die Attribute sind also `private` und auch die Methode `vertauscheKarten`, denn diese wird nur intern von der Klasse benötigt für das Mischen des Stapels. #

Information 5.7 Insgesamt gilt folgende Faustregel:

- Attribute: `private`
- Methoden: `public`

5.2 Getter- und Setter-Methoden

Bemerkung 5.8 Da die Attribute `private` sind, kann man nicht mehr auf sie zugreifen. Sollte dies dennoch nötig sein, so implementiert man eine passende Getter- bzw. Settermethode:

```
1   class Pokemon{
2     private int leben; //das Attribut leben ist private
3     ...
4     public int getLeben(){ //die Getter-Methode liefert den Wert des Attributs
5       return leben;
6     }
7     public void setLeben(int leben){ //die Setter-Methode erlaubt das Verändern des Attributs
8       this.leben=leben;
9     }
10  }
```

#

Information 5.9 Eine **Getter-Methode** für ein Attribut liefert den Wert dieses Attributs zurück:

```
1   private Typ ATTRIBUT;
2   ...
3
4   Typ getATTRIBUT(){
5     return ATTRIBUT;
6   }
```

Eine **Setter-Methode** für ein Attribut erlaubt es, den Wert des Attributs zu verändern.

```
1   private Typ ATTRIBUT;
2   ...
3
4   void setATTRIBUT( Typ neuerWert){
5     ATTRIBUT = neuerWert;
6   }
```

5.3 Der Sinn hinter dem Geheimnisprinzip

Eine konsequente Umsetzung des Geheimnisprinzips zwingt die*den Programmierer*in, mehr Funktionalität in die Klassen zu verlagern und die Probleme da zu lösen, wo sie am besten gelöst werden können.

Wir schauen uns ein typisches Beispiel an und besprechen dann detailliert die Anwendung des Geheimnisprinzips und dessen Auswirkungen.

5.3.1 Ausgangslage: Ein Routenplaner

Für einen Routenplaner muss man den Abstand zweier Städte berechnen. Im einfachsten Fall sieht die Klasse `Stadt` so aus:

```
1   class Stadt{
2     double x,y;
3     String name;

4     Stadt(String name, double x, double y){
5       ...
6     }
7   }
```

In der Hauptklasse gibt es ein Array von Städten. Wenn wir dort den Abstand berechnen, könnte das so aussehen:

```
1   class Routenplaner{
2     Stadt[] staedte;

3     void onAction(JComponent trigger){
4       //in diesem Beispiel holen wir die
5       //beiden Städte aus JComboBoxen:
6       int index1 = screen.stadt1.getSelectedIndex();
7       int index2 = screen.stadt2.getSelectedIndex();
8       Stadt s1 = staedte[index1];
9       Stadt s2 = staedte[index2];
10      //abstand ausgeben in einem JLabel:
11      screen.ergebnis.setValue( abstandZwischen(s1,s2) );
12    }

13    double abstandZwischen(Stadt s1, Stadt s2){
14      double x1 = s1.x;
15      double y1 = s1.y;
16      double x2 = s2.x;
17      double y2 = s2.y;
18      //Pythagoras:
19      double abstand = Math.sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2));
20      return abstand;
21    }
22  }
```

5.3.2 Anwendung des Geheimnisprinzips

Wenn wir nun das Geheimnisprinzip anwenden, d.h.

```
1   class Stadt{
2     private double x,y;
3     private String name;
4     ...
5   }
```

dann funktioniert die `abstandZwischen`-Methode nicht mehr, da `x` und `y` verborgen sind.

5.3.3 Lösungsmöglichkeit 1: Getter-Methoden

Ein Lösungsansatz wären Getter-Methoden für `x` und `y`:

```
1   class Stadt{
2     private double x,y;
3     private String name;
4     ...
5     public double getX(){
6       return x;
7     }
8     public double getY(){
9       return y;
10    }
11  }
```

Damit würde die `abstandZwischen`-Methode so aussehen:

```
1   double abstandZwischen(Stadt s1, Stadt s2){
2     double x1 = s1.getX();
3     double y1 = s1.getY();
4     double x2 = s2.getX();
5     double y2 = s2.getY();
6     //Pythagoras:
7     double abstand = Math.sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2));
8     return abstand;
9   }
```

5.3.4 Lösungsmöglichkeit 2: Die Funktionalität verlagern

Statt Getter-Methoden zu implementieren, können wir der Klasse `Stadt` eine passende Methode hinzufügen:

```
1   class Stadt{
2     private double x,y;
3     private String name;
4     ...
5     double abstandZu(Stadt s){
6       double abstand = Math.sqrt((x-s.x)*(x-s.x)+(y-s.y)*(y-ys.y));
7       return abstand;
8     }
9   }
```

Wenn wir das so machen, können wir die `abstandZwischen`-Methode der Klasse `Routenplaner` komplett eliminieren und stattdessen die neue Methode der Klasse `Stadt` nutzen:

```
1   class Routenplaner{
2     Stadt[] staedte;

3     void onAction(JComponent trigger){
4       //in diesem Beispiel holen wir die
5       //beiden Städte aus JComboBoxen:
6       int index1 = screen.stadt1.getSelectedIndex();
7       int index2 = screen.stadt2.getSelectedIndex();
8       Stadt s1 = staedte[index1];
9       Stadt s2 = staedte[index2];
10      //abstand ausgeben in einem JLabel, mit neuer Methode:
11      screen.ergebnis.setValue( s1.abstandZu(s2) );
12    }
13  }
```

5.3.5 Vergleich der beiden Lösungsmöglichkeiten

In diesem Beispiel sieht man recht gut, dass die Klasse `Stadt` sehr viel besser geeignet ist, das Problem (Berechnung des Abstands) zu lösen als die Klasse `Routenplaner`.

Während der `Routenplaner` sich erst alle Informationen der beiden Städte holen muss, kann die `Stadt` direkt alle benötigten Informationen verwenden.

Lösungsmöglichkeit 2 resultiert also in kürzerem Code. Beachte, dass der Code sogar kürzer ist als der ursprüngliche Code vor Anwendung des Geheimnisprinzips. Außerdem steigt die Übersichtlichkeit, da die Hauptklasse nicht mit allen möglichen Funktionalitäten überladen wird. Ein anderer Vorteil ist, dass man eine Klasse, die man einmal programmiert hat, auch in anderen Projekten verwenden kann. Je mehr Funktionalität dann in der Klasse steckt, desto mehr kann man mitnehmen.

Information 5.10 Lange Rede, kurzer Sinn: In den meisten Fällen ist es besser, die entsprechende Funktionalität in Form einer neuen Methode in eine Klasse zu verlegen anstatt Getter- und/oder Setter-Methoden zu verwenden.

6 Modellieren mit UML

6.1 Darstellung von Klassen in UML

Wenn größere Software-Projekte umgesetzt werden sollen, ist es wichtig, dass das Entwicklungsteam zunächst den Aufbau der Applikation plant. In der Anfangszeit der Software-Entwicklung hat sich jedes Unternehmen eine eigene Möglichkeit der Beschreibung ausgedacht, bis 1997 die erste Version der *Unified Modelling Language (UML)* veröffentlicht wurde und ihren Siegeszug angetreten hat. Die UML kennt mittlerweile 14 verschiedene Diagramm-Arten, für uns ist jedoch nur das UML-Klassendiagramm von Bedeutung:

Definition 6.1 *Das UML-Klassendiagramm stellt alle in einer Software beteiligten Klassen sowie die Beziehungen zwischen diesen Klassen dar.*

Die Schreibweise für die einzelnen Klassen haben wir bereits kennengelernt:

Definition 6.2 *Eine Klasse wird als Rechteck dargestellt:*

Kartenstapel
-karten: Karte[] *-anzahlKarten: int*
Kartenstapel() *+karteZiehen(): Karte* *+karteAblegen(k: karte)* *+mischen()* *-vertauscheKarten(index1: int, index2: int)*

Zuerst kommen die Attribute, gefolgt von den Methoden der Klasse. Ein + bedeutet, dass das Attribut / die Methode `public` ist, ein –, dass es / sie `private` ist.

6.2 Assoziationen und Aggregationen

Neu für uns ist, dass zwei Klassen miteinander in Beziehung stehen können. Wir entwickeln die folgenden Begriffe anhand eines Beispiels:

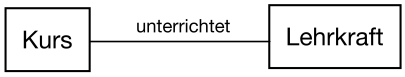
Beispiel 6.3 Es soll eine Software zur Verwaltung einer gymnasialen Oberstufe implementiert werden. Wir sammeln zunächst, was wir darüber wissen:

- Die Oberstufe besteht aus den Jahrgangsstufen E, Q1/2 und Q3/4.
- Jede Jahrgangsstufe besteht wiederum aus Kursen.
- Kurse werden von Schüler*innen besucht und von Lehrkräften unterrichtet.

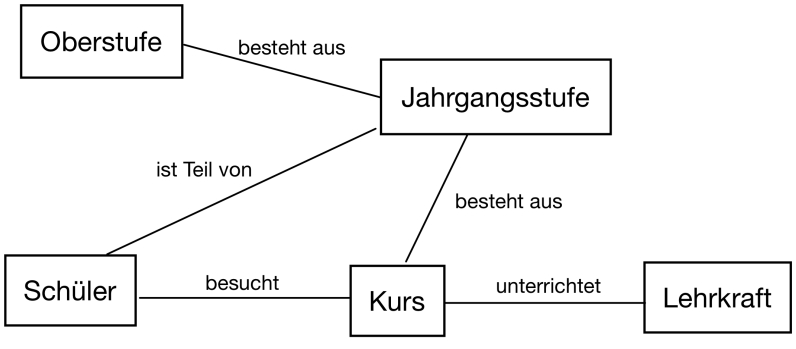
Daraus ergeben sich die beteiligten Klassen: `Oberstufe`, `Jahrgangsstufe`, `Kurs`, `Schüler` und `Lehrkraft`.
Unser UML-Klassendiagramm würde also zunächst folgendermaßen aussehen:

#

Definition 6.4 *Eine* **Assoziation** *ist eine Beziehung zwischen genau zwei Klassen. Sie wird durch eine Linie dargestellt, die die beiden Klassen verbindet und über der eine Beschreibung der Beziehung steht, meist ein einzelnes Verb.*



Beispiel 6.5 In unserem Fall ergeben sich also die folgenden Assoziationen:



#

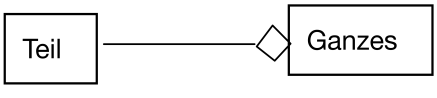
Wie man sieht, tauchen sehr häufig »besteht aus«-Assoziationen auf. Daher werden diese besonders gekennzeichnet:

Definition 6.6 *Eine »besteht aus«-Assoziation zwischen einem Teil und einem Ganzen wird auch als* **Aggregation** *(»aggregieren« = »zusammenfassen«) bezeichnet.*

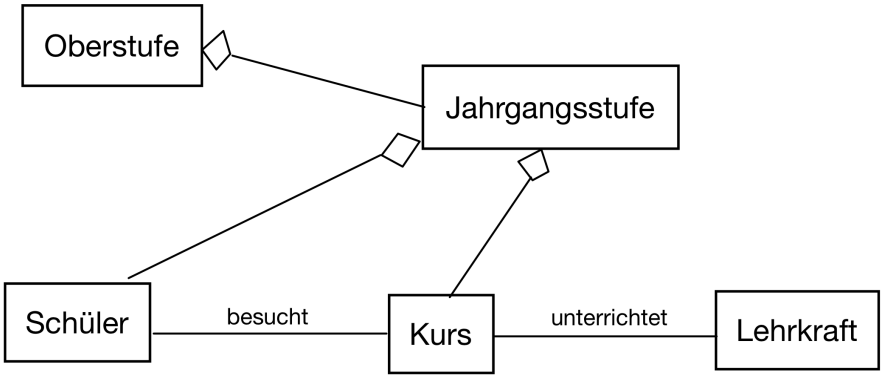
Eine Aggregation wird anders dargestellt als andere Assoziationen:

(1) Die Benennung entfällt.

(2) Die Klasse, die das Ganze darstellt, wird dadurch gekennzeichnet, dass die Linie an dieser Klasse in einer Raute endet.



Beispiel 6.7 In unserem Beispiel ändern wir einige Assoziationen in Aggregationen ab:



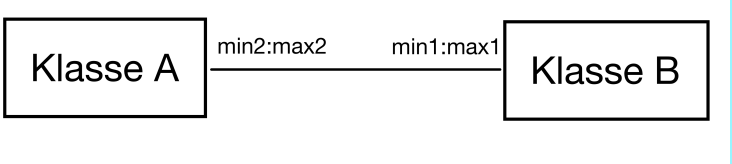
#

6.3 Multiplizitäten

Zu guter Letzt ist es vielfach entscheidend, wie viele Objekte der Klassen miteinander in Beziehung stehen:

Definition 6.8 *Die* **Multiplizitäten** *einer Beziehung geben an, wie viele Objekte an der Beziehung beteiligt sind.*

Die Multiplizitäten werden in der Form `anzahl` *oder* `min..max` *an die Enden der Linie notiert. Dabei steht* $*$ *für »beliebig viele«:*

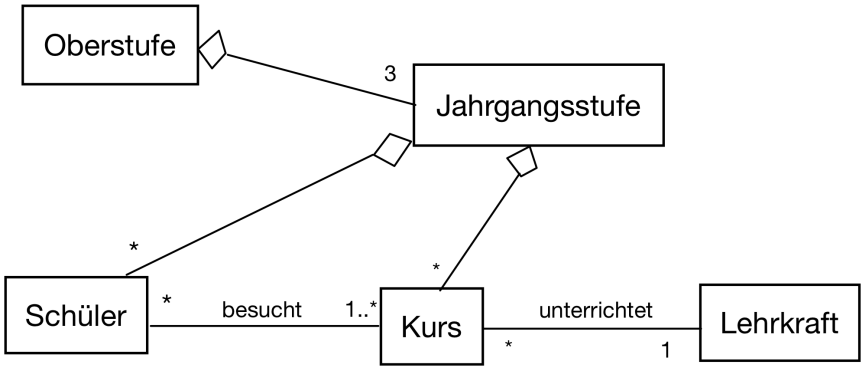


Dies wird in zwei Richtungen gelesen:

(a) Von links nach rechts: Jedes Objekt der Klasse A steht in Beziehung B mit $min1$ bis $max1$ Objekten der Klasse C.

(b) Von rechts nach links: Jedes Objekt der Klasse C steht in Beziehung B mit $min2$ bis $max2$ Objekten der Klasse A.

Beispiel 6.9 In unserem Beispiel ergeben sich die folgenden Multiplizitäten:



Hier eine Auswahl der Bedeutungen dieses Beziehungsgeflechts:

- Jede Oberstufe besteht aus (genau) 3 Jahrgangsstufen.
- Jede:r Schüler:in gehört zu genau 1 Jahrgangsstufe (bei Aggregationen steht auf der »Ganzes«-Seite immer eine 1, deshalb darf man sie weglassen).
- Jede Jahrgangsstufe besteht aus beliebig vielen Schüler:innen.
- Jede:r Schüler:in besucht mindestens 1 Kurs.
- Jeder Kurs wird von beliebig vielen Schüler:innen besucht.

#

Information 6.10 Anhand der Beziehungen in einem UML-Klassendiagramm kann abgeleitet werden, welche Attribute benötigt werden, um diese Beziehungen abzubilden.

Immer, wenn beliebig viele Elemente möglich sind, muss ein Array verwendet werden, ansonsten reicht eine einfache Variable.

Beispiel 6.11 In unserem Fall ergeben sich folgende Attribute:

(1) Die Klasse `Oberstufe` erhält ein Array `Jahrgangsstufe[] jahrgangsstufen`.

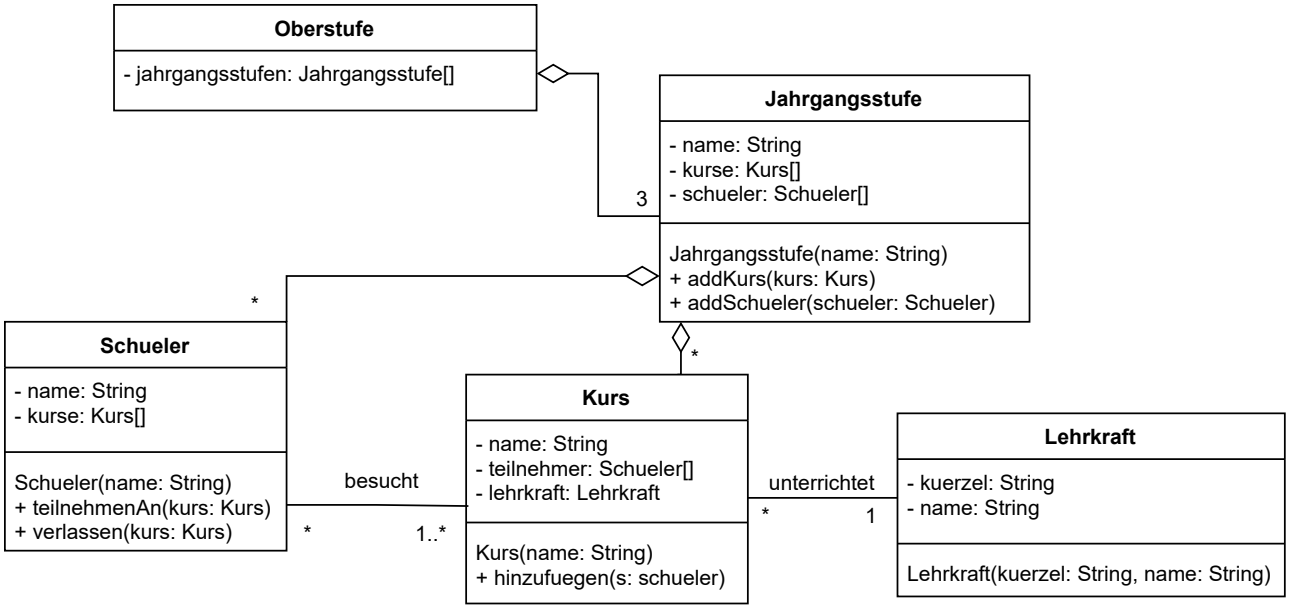
(2) Die Klasse `Jahrgangsstufe` erhält ein Array `Schueler[] schueler` und ein Array `Kurs[] kurse`.

(3) Die Klasse `Schueler` erhält ein Array `Kurse[] kurse`.

(4) Die Klasse `Kurs` erhält ein Array `Schueler[] teilnehmer`.

(5) Die Klasse `Kurs` erhält ein Attribut `Lehrkraft lehrkraft`.

Insgesamt könnte sich damit etwa folgendes UML-Klassendiagramm ergeben (ergänzt um eine Reihe weiterer Attribute und Methoden):



#

7 Suchalgorithmen

7.1 Die sequentielle Suche

Wir haben gesehen, dass man große Datenmengen strukturiert verwalten kann, indem man Objekte in Arrays speichert. In diesem Kapitel geht es darum, solche Arrays nach bestimmten Objekten zu durchsuchen.

Beispiel 7.1 In einem Array sind Personen gespeichert, wobei jede Person unter anderem einen Vornamen und einen Nachnamen besitzt.

Dieses Array soll nun nach einer bestimmten Person durchsucht werden, es soll also die Methode

`Person suchen(Person[] array, String nachname, String vorname)`

implementiert werden, die die erste Person mit den gegebenen Daten zurückliefert oder `null`, wenn es keine solche Person im Array gibt.

Eine Lösung könnte folgendermaßen aussehen:

```
1   public Person suchen(Person[] array, String nachname, String vorname){
2     for( int i = 0; i < array.length; i++ ){
3       Person p = array[i];
4       if( p.nachname == nachname && p.vorname == vorname ){
5         return p;
6       }
7     }
8     return null;
9   }
```

#

Definition 7.2 *Der Such-Algorithmus*

```
1   Iteriere für alle Elemente des Arrays{
2     Wenn das aktuelle Element die Suchkriterien erfüllt{
3       Gib das aktuelle Element zurück
4     }
5   }
6   Gib null zurück
```

heißt **sequentielle Suche**.

Bemerkung 7.3 »Sequentiell« bedeutet so viel wie »nacheinander«: Nacheinander geht man alle Elemente durch bis man ein passendes gefunden hat. #

7.2 Die binäre Suche

Bemerkung 7.4 Die sequentielle Suche ist relativ ineffizient, da im schlimmsten Fall alle Einträge des Arrays verglichen werden müssen.

In der Praxis ist es oft möglich, effizienter zu suchen, beispielsweise bei der Suche nach einem Namen im Telefonbuch: Da diese alphabetisch sortiert sind, fängt man nicht vorne an und geht alle Einträge durch, sondern man startet etwa in der Mitte und prüft, ob der gesuchte Name weiter vorne oder weiter hinten liegt.

Dadurch kann man den gesuchten Eintrag immer weiter eingrenzen, bis er gefunden ist. #

Definition 7.5 *Der Suchalgorithmus*

```
1   setze L auf 0
2   setze R auf die Länge des Arrays minus 1
3   wiederhole, solange L<=R{
4     setze M auf den Mittelwert von L und R (ggf. abgerundet)
5     wenn das M-te Element dem gesuchten Wert entspricht{
6       gib das M-te Element zurück
7     }
8     wenn der gesuchte Wert kleiner als das M-te Element ist{
9       setze R auf M-1
10    }
11    wenn der gesuchte Wert größer als das M-te Element ist{
12      setze L auf M+1
13    }
14  }
15  gib null zurück
```

heißt **binäre Suche**. *Er funktioniert nur, wenn die Daten nach dem Suchkriterium aufsteigend sortiert sind.*

Beispiel 7.6 Gegeben ist das Array

```
[
{name: "Aylin", alter: 14}, {name: "Fee", alter: 5}, {name: "Gustav", alter: 21},
{name: "Ida", alter: 13}, {name: "Konrad", alter: 7}, {name: "Ludwig", alter: 15},
{name: "Martha", alter: 24}, {name: "Thomas", alter: 38}, {name: "Valerie", alter: 32}
]
```

in dem 9 Personen mit Name und Alter gespeichert sind. Wir wenden die binäre Suche an, um nach der Person mit Namen »Gustav« zu suchen:

(1) $L = 0$, $R = 8$. Dies sind die Grenzen, in denen wir suchen:

0	1	2	3	4	5	6	7	8
Aylin	Fee	Gustav	Ida	Konrad	Ludwig	Martha	Thomas	Valerie
L								R

(2) $M = 4$: Das 4-te Element ist »Konrad«. Da »Gustav« kleiner ist (im Alphabet weiter vorne), setze R auf 3.
Wir wissen jetzt: Die gesuchte Person liegt zwischen $L = 0$ und $R = 3$.

0	1	2	3	4	5	6	7	8
Aylin	Fee	Gustav	Ida	Konrad	Ludwig	Martha	Thomas	Valerie
L			R					

(3) $M = 1$ ($1{,}5$ abgerundet): Das 1-te Element ist »Fee«. Da »Gustav« größer ist, setze L auf 2.
Wir wissen jetzt: Die gesuchte Person liegt zwischen $L = 2$ und $R = 3$.

0	1	2	3	4	5	6	7	8
Aylin	Fee	Gustav	Ida	Konrad	Ludwig	Martha	Thomas	Valerie
		L	R					

(4) $M = 2$ ($2{,}5$ abgerundet): Das 2-te Element ist »Gustav« und die Suche ist beendet.

Wäre der gesuchte Name z. B. »Hatice« gewesen, so wäre zunächst L auf 3 gesetzt worden, weil »Gustav« kleiner als »Hatice« ist. Dann wäre M auf 3 und danach R auf 2 gesetzt worden. Anschließend wäre die Schleife beendet, da $L > R$ gelten würde und es würde `null` zurückgegeben. #

Bemerkung 7.7 Die binäre Suche ist sehr viel schneller als die sequentielle Suche. Wir werden uns dies in **Kapitel 9** genauer anschauen. #

8 Sortieralgorithmen

Neben dem Suchen stellt das Sortieren ein Grund-Problem der Informatik dar. Wie wir beispielsweise gesehen haben, lassen sich große Datenmengen deutlich effizienter durchsuchen, wenn die Daten sortiert sind.

Definition 8.1 *Ein* **Sortieralgorithmus** *sortiert die Elemente eines Arrays nach einem bestimmten Kriterium.*

Es gibt seeeehr viele Sortieralgorithmen und im Unterricht haben Sie sich selbst einen Algorithmus ausgedacht und implementiert.

Wir behandeln an dieser Stelle nur einen dieser Algorithmen, den sogenannten »Bubblesort«-Algorithmus, der sehr einfach zu implementieren ist, was an folgender Tatsache liegt:

Information 8.2 Während das Einfügen eines Elementes in ein Array oder das Löschen eines Elementes aus einem Array sehr aufwändig sind und jeweils eine Schleife benötigen, um alle nachfolgenden Elemente zu verschieben, ist das *Vertauschen* zweier Elemente eines Arrays sehr einfach.

Beispiel 8.3 Gegeben ist das Array

```
String[] faecher = {"Info","Mathe","Geschi","Bio","Kunst"};
```

Die Methode `void vertauschen(int pos1, int pos2)` vertauscht dann die Elemente an den Positionen `pos1` und `pos2` miteinander:

```
1   void vertauschen(int pos1, int pos2){
2     String c = faecher[ pos1 ];
3     faecher[ pos1 ] = faecher[ pos2 ];
4     faecher[ pos2 ] = c;
5   }
```

Der Aufruf `vertauschen ( 1, 3 );` würde also folgenden Ablauf bewirken:

```
1   c = faecher[ 1 ]; //="Mathe"
2   faecher[ 1 ] = faecher[3]; // Array: {"Info","Bio","Geschi","Bio","Kunst"}
3   faecher[pos2] = "Mathe"; // Array: {"Info","Bio","Geschi","Mathe","Kunst"}
```

#

Definition 8.4 *Der »Algorithmus« zum Vertauschen zweier Elemente eines Arrays wird auch* **Dreieckstausch** *genannt.*

Auf diesem Vertauschen von Elementen beruht der genannte Bubble-Sort-Algorithmus:

Definition 8.5 *Der Bubble-Sort Algorithmus funktioniert auf folgende Weise:*

(1) Gehe alle Elemente des Arrays durch und vergleiche mit dem Nachfolge-Element (falls vorhanden): Ist der Nachfolger kleiner als das Element, vertausche die beiden.

(2) Wiederhole dies, bis das Array sortiert ist.

Beispiel 8.6 Eine mögliche Implementierung für String-Arrays könnte so aussehen:

```
1   void bubblesort(String[] array){
2     boolean sortiert = false;
3     while(sortiert == false){
4       sortiert = true;
5       for( int i = 0; i < array.length - 1; i++ ){
6         if( array[ i ] > array[ i + 1 ]){
7           vertauschen(array, i, i + 1);
8           sortiert = false;
9         }
10      }
11    }
12  }
```

Dazu muss natürlich noch eine passende `vertauschen`-Methode implementiert werden.

#

Bemerkung 8.7 Es gibt natürlich mehrere Möglichkeiten, den Bubble-Sort-Algorithmus zu implementieren. Es ist auch möglich, zwei ineinander verschachtelte `for`-Schleifen zu verwenden, wodurch man den Algorithmus deutlich kompakter machen kann:

```
1   void bubblesort(String[] array){
2     for( int i = 0; i < array.length; i++ ){
3       for( int j = 0; j < array.length -1, j++){
4         vertauscheWennNoetig(array, j);
5       }
6     }
7   }
```

Hier muss dann noch eine passende Methode `vertauscheWennNoetig` implementiert werden:

```
1   void vertauscheWennNoetig(String[] array, int pos){
```

```
2      if( array[ pos + 1 ] > array[ pos ]){
3        String c = array[ pos ];
4        array[ pos ] = array[ pos + 1 ];
5        array[ pos + 1 ] = c;
6      }
7    }
```

#

9 Die Laufzeit eines Algorithmus

Für die Lösung eines Problems gibt es prinzipiell beliebig viele mögliche Algorithmen. Deshalb ist es wichtig, die Qualität eines Algorithmus bewerten zu können. Das wichtigste Maß dafür ist die sogenannte »Laufzeit«:

Definition 9.1 *Die* **Laufzeit** *eines Algorithmus ist die Anzahl der* **wesentlichen Operationen**, *die der Algorithmus durchführen muss, um das Problem zu lösen.*

Wir unterscheiden drei unterschiedliche Laufzeiten:

(1) Die **Best-Case-Laufzeit** *ist die Laufzeit im für den Algorithmus günstigsten Fall.*

(2) Die **Worst-Case-Laufzeit** *ist die Laufzeit im für den Algorithmus ungünstigsten Fall.*

(3) Die **Average-Case-Laufzeit** *ist die Laufzeit »im Mittel«.*

Da die Average-Case-Laufzeit normalerweise äußerst schwer zu ermitteln ist und die Best-Case-Laufzeit für die Praxis irrelevant ist, verstehen wir unter Laufzeit immer die Worst-Case-Laufzeit, wenn nicht anders angegeben.

Für die konkrete Angabe von Laufzeiten bilden wir grobe Kategorien:

Definition 9.2 *Gegeben sei ein Algorithmus, der eine Eingabe der Länge n verarbeiten muss.*

Wir verwenden folgende Sprechweisen:

(a) **Konstante Laufzeit***: Unabhängig von n werden immer gleich viele Operationen benötigt.*

(b) **Logarithmische Laufzeit***: Eine Verdopplung von n erhöht die Laufzeit um 1 (oder um eine andere Konstante).*

(c) **Lineare Laufzeit***: Eine Verdopplung von n verdoppelt (ungefähr) die Laufzeit.*

(d) **Quadratische Laufzeit***: Eine Verdopplung von n vervierfacht (ungefähr) die Laufzeit.*

(e) **Kubische Laufzeit***: Eine Verdopplung von n verachtfacht (ungefähr) die Laufzeit.*

(f) **Exponentielle Laufzeit***: Eine Erhöhung von n um 1 verdoppelt (ungefähr) die Laufzeit.*

Bemerkung 9.3 Bei der Untersuchung der Laufzeit geht es wirklich nur um eine grobe Einteilung (der höchste Exponent setzt sich durch). Wenn ein Algorithmus B immer doppelt so lange braucht wie ein anderer Algorithmus A, fallen beide trotzdem in dieselbe Laufzeit-Kategorie!

Warum das so ist? Man kann beweisen, dass es durch Steigerung der Hardware möglich ist, jeden Algorithmus doppelt so schnell auszuführen. Darum wäre der langsame Algorithmus B auf der richtigen Hardware genauso schnell wie der schnellere Algorithmus A.

Im Gegensatz dazu können die Grenzen der oben genannten Kategorien nicht durch eine Verbesserung der Hardware aufgehoben werden. D.h., ein quadratischer Algorithmus wird (ab einer gewissen Größe on n) immer langsamer sein als ein linearer Algorithmus, selbst wenn der eine auf einem Rechner aus den 60ern und der andere auf einem Rechner aus der fernen Zukunft laufen wird.[7] #

Beispiel 9.4 Wir analysieren die Laufzeiten der uns bekannten Sortier- und Suchalgorithmen:

(a) Die sequentielle Suche hat eine lineare Laufzeit, da schlimmstenfalls alle Einträge durchsucht werden müssten.

(b) Die binäre Suche hat eine logarithmische Laufzeit, da jeder Schritt die Anzahl der in Frage kommenden Einträge halbiert.

(c) Bubble-Sort hat eine quadratische Laufzeit, was an den zwei ineinander verschachtelten Schleifen liegt: Die innere Schleife macht n Operationen und wird für jeden Durchgang der äußeren Schleife durchgeführt, also n-mal. Dies ergibt $n \times n = n^2$ viele Operationen. #

Bemerkung 9.5 Es hat seine Gründe, dass wir an dieser Stelle keinen Algorithmus mit exponentieller Laufzeit aufführen können: Ein solcher Algorithmus hat ab einem gewissen n eine derart hohe Laufzeit, dass er Jahre braucht um zum Ergebnis zu kommen. Dies wird vor allem in der Krypographie (Verschlüsselung) verwendet:

Nehmen wir an, wir wollen eine PIN knacken, die aus maximal n Ziffern bestehen kann. Dann gibt es 10^n viele Möglichkeiten. Ein Algorithmus, der einfach alle PINs ausprobiert hätte damit exponentielle Laufzeit. #

[7] Vorausgesetzt, dass es sich bei beiden um »klassische« Rechner handelt. Insbesondere für Quantencomputer gilt diese Aussage nicht!

Definition 9.6 *Zur Angabe von Laufzeiten verwendet man auch die sog.* **Landau-Notation***:*

(a) $O(1)$ *für eine konstante Laufzeit;*

(b) $O(\log(n))$ *für eine logarithmische Laufzeit;*

(c) $O(n)$, $O(n^2)$, $O(n^3)$, ... *für eine lineare, quadratische, kubische, ... Laufzeit;*

(d) $O(2^n)$ *für eine exponentielle Laufzeit.*

Beispiel 9.7 Der BubbleSort-Algorithmus hat also eine Laufzeit von $O(n^2)$, während die sequentielle Suche eine Laufzeit von $O(n)$ hat. #

10 Rekursion

»Um Rekursion zu verstehen, muss man Rekursion verstehen.«

— Unbekannt

Bisher haben wir unsere Algorithmen stets **iterativ**[8] programmiert, d.h. unter Verwendung von Schleifen. Dabei haben wir festgestellt, dass prinzipiell simple Ideen wie die binäre Suche

> Wenn das gesuchte Element kleiner ist als das in der Mitte, suche links weiter, ansonsten suche rechts weiter.

als iterativer Algorithmus kaum wiederzuerkennen sind:

```
1   Person binaereSucheIterativ(Person[] array, String name){
2     int links = 0;
3     int rechts = array.length-1;
4     while(links <= rechts){
5       int mitte = ( links + rechts )/2;
6       if(name == array[ mitte ].name){
7         return array[ mitte ];
8       }
9       if(name < array[ mitte ].name){
10        links = mitte + 1;
11      }else{
12        rechts = mitte - 1;
13      }
14    }
15    return null;
16  }
```

Deshalb versuchen wir es noch mal, indem wir uns möglichst genau an der Beschreibung

> Wenn das gesuchte Element kleiner ist als das in der Mitte, suche links weiter, ansonsten suche rechts weiter.

orientieren:

```
1   Person binaereSucheRekursiv(Person[] array, String name){
```

[8] lateinisch »iterare« = »wiederholen«

```
2     int mitte = array.length / 2;
3     if( name == array[ mitte ].name ){
4       return array[ mitte ];
5     }
6     if( name < array[ mitte ].name ){
7       //suche links von Mitte weiter
8     }else{
9       //suche rechts von Mitte weiter
10    }
11  }
```

Nun stellt sich aber die Frage, wie das »suche links/rechts von der Mitte weiter« konkret funktionieren soll?

Die verblüffende Antwort: Wir rufen die Methode `binaereSucheRekursiv` einfach wieder auf, denn wir wollen die Suche ja wiederholen. Um auszudrücken, in welchem Bereich wir suchen, fügen wir zwei Parameter `links` und `rechts` hinzu und erhalten:

```
1   Person binaereSucheRekursiv(Person[] array, String name, int links, int rechts){
2     int mitte = ( links + rechts ) / 2;
3     if( name == array[ mitte ].name ){
4       return array[ mitte ];
5     }
6     if( name < array[ mitte ].name ){
7       //suche links von Mitte weiter:
8       binaereSucheRekursiv( array, name, links, mitte - 1 );
9     }else{
10      //suche rechts von Mitte weiter:
11      binaereSucheRekursiv( array, name, mitte + 1, rechts );
12    }
13  }
```

Die Methode ruft sich also selbst auf!

Definition 10.1 *Eine Methode heißt* **rekursiv**, *wenn sie sich selbst aufruft.*

Wir schauen uns dies am gleichen Beispiel wie in **Kapitel 7** an:

Beispiel 10.2 Gegeben ist das Array

```
personen = [
  {name: "Aylin", alter: 14},
  {name: "Fee", alter: 5},
  {name: "Gustav", alter: 21},
  {name: "Ida", alter: 13},
  {name: "Konrad", alter: 7},
  {name: "Ludwig", alter: 15},
  {name: "Martha", alter: 24},
  {name: "Thomas", alter: 38},
  {name: "Valerie", alter: 32}
]
```

in dem 9 Personen mit Name und Alter gespeichert sind.

Wir wenden die binäre Suche an, um nach der Person mit Namen »Gustav« zu suchen, d.h., wir machen folgende Anweisung:

```
binaereSucheRekursiv(personen, "Gustav", 0, 8 )
```

denn wir suchen im kompletten Bereich des Arrays.

(1) `mitte = 4`. Das 4-te Element ist »Konrad«. Da »Gustav« kleiner ist (im Alphabet weiter vorne), rufen wir die Methode wieder auf und zwar:

```
binaereSucheRekursiv(personen, "Gustav", 0, 3 )
```

denn nun suchen wir nur noch im Bereich 0 bis 3!

(2) Wir starten wieder in Zeile 2 und erhalten `mitte = 1` (1,5 abgerundet): Das 1-te Element ist »Fee«. Da »Gustav« größer ist, rufen wie Methode wieder auf und diesmal als:

```
binaereSucheRekursiv(personen, "Gustav", 2, 3 )
```

denn nun suchen wir nur noch im Bereich 2 bis 3.

(3) Wir starten wieder in Zeile 2 und erhalten `mitte = 2` (2,5 abgerundet): Das 2-te Element ist »Gustav« und dieses wird zurückgeliefert.

Insgesamt arbeitet die Methode also so:

```
binaereSuche(personen, "Gustav", 0, 8) = binaereSuche(..., 0, 3)
                                       = binareSuche(..., 2, 3)
                                       = Objekt {Gustav}
```


#

An dieser Implementierung erkennen wir den prinzipiellen Aufbau einer rekursiven Methode:

Information 10.3 Eine rekursive Methode hat immer den folgenden Aufbau:

```
Typ rekursiveMethode(Parameter){
  if( einfacher Fall ){
    //bestimme das Ergebnis für diesen einfachen Fall
    return Ergebnis;
  }
  return rekursiveMethode(...);
}
```

Dabei muss darauf geachtet werden, dass das zu lösende Problem bei jedem Rekursionsschritt einfacher wird (z.B., indem immer weniger Elemente durchsucht werden müssen), sodass der einfache Fall auch erreicht werden kann.

Beispiel 10.4 Die Methode `int summe(int[] zahlen)` soll die Summe aller Zahlen im gegebenen Array bilden und zurückliefern.

Wir implementieren diese Methode rekursiv:

```
1   public int summe( int[] zahlen ){
2     return summeRekursiv( zahlen, 0 );
3   }
4
5   private int summeRekursiv( int zahlen[], int startIndex ){
6     if(startIndex >= zahlen.length){
7       //einfachster Fall: Es müssen gar keine Zahlen addiert werden
8       return 0;
9     }
10    //addiere eine Zahl weniger:
11    int a = zahlen[ startIndex ];
12    int b = summeRekursiv( zahlen, startIndex + 1 );
13    return a + b;
14  }
```

#

Bemerkung 10.5 Unser Such-Algorithmus bekommt ein Problem, wenn der gesuchte Name gar nicht im Array enthalten ist, in diesem Fall würde er nämlich niemals abbrechen. Wir können dies leicht beheben, indem wir eine zusätzliche Abbruchbedingung

hinzufügen, die abfängt, dass der Bereich, den wir durchsuchen, gar nicht mehr existiert:

```
1   Person binaereSucheRekursiv(Person[] array, String name, int links, int rechts){
2     if(links>rechts){
3       return null;
4     }
5     int mitte = ( links + rechts ) / 2;
6     if( name == array[ mitte ].name ){
7       return array[ mitte ];
8     }
9     if( name < array[ mitte ].name ){
10      //suche links von Mitte weiter:
11      binaereSucheRekursiv( array, name, links, mitte - 1 );
12    }else{
13      //suche rechts von Mitte weiter:
14      binaereSucheRekursiv( array, name, mitte + 1, rechts );
15    }
16  }
```

#

Information 10.6 Rekursives Problemlösen funktioniert also nach folgendem Schema:

(1) Formuliere eine Lösung für eine sehr einfache Variante des Problems.

(2) Löse eine komplexe Variante des Problems indem du eine etwas weniger komplexe Variante löst und daraus eine Lösung zusammenbaust.

11 Rekursion vs. Iteration

In diesem letzten Kapitel vergleichen wir iterative und rekursive Algorithmen und nennen Vor- und Nachteile, die diese im Allgemeinen haben.

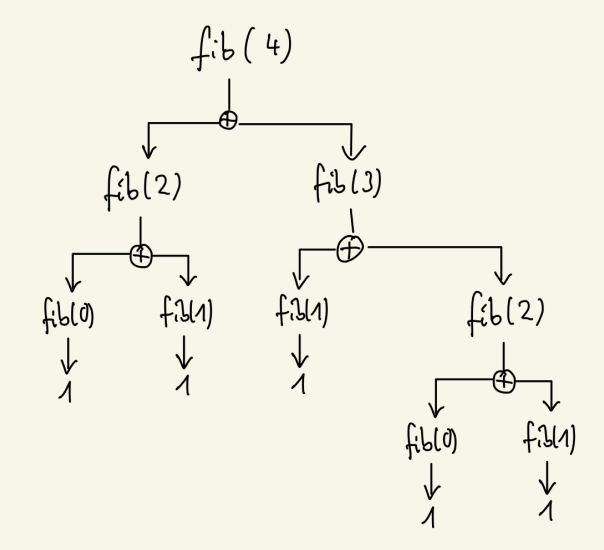
Beispiel 11.1 Die »Fibonacci-Zahlen« 1, 1, 2, 3, 5, 8, 13, 21, 34, ... sind eine sehr berühmte Zahlenfolge. Die nächste Zahl entsteht immer durch die Summe der beiden vorangegangenen Zahlen.

Wir implementieren eine iterative und eine rekursive Methode zur Berechnung der n-ten Fibonacci-Zahl:

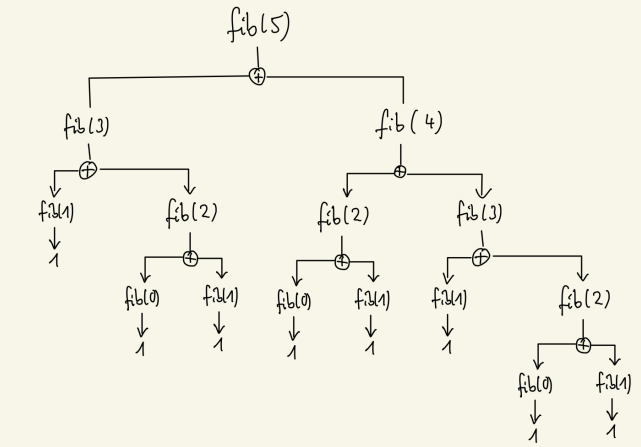
```
1   int fibIterativ(int n){
2     if( n == 1 || n == 2 ){
3       return 1;
4     }
5     int a = 1;
6     int b = 1;
7     for( int i = 3; i <= n; i++){
8       int f = a + b;
9       a = b;
10      b = f;
11    }
12    return f;
13  }
```

```
1   int fibRekursiv(int n){
2     if( n == 1 || n == 2 ){
3       return 1;
4     }
5     return fibRekursiv( n-2 ) + fibRekursiv( n-1 );
6   }
```

Der iterative Algorithmus hat offensichtlich eine lineare Laufzeit. Wie sieht es beim rekursiven Algorithmus aus? Dies kann man sich am besten grafisch klar machen:



n = 4



n = 5

Wie man sieht, wird bei der Berechnung von $fib(5)$ der Wert von $fib(3)$ zweimal berechnet, der Wert von $fib(2)$ dreimal und der Wert von $fib(1)$ sogar 5 mal!

Noch schlimmer wird es, wenn $fib(6)$ berechnet wird: Dann müssen alle Rechenoperationen von $fib(5)$ und von $fib(4)$ durchgeführt werden! Der Aufwand für die Berechnung von $fib(6)$ ist also fast doppelt so hoch wie der für die Berechnung von $fib(5)$. Daraus folgt, dass der rekursive Algorithmus eine exponentielle Laufzeit hat! #

Information 11.2 Rekursive Algorithmen haben häufig folgende Vorteile im Vergleich zu iterativen Algorithmen:

(1) Sie sind häufig kürzer und eleganter.
(2) Sie benötigen keine komplizierten Schleifen.

Rekursive Algorithmen können aber auch große Nachteile gegenüber iterativen Algorithmen besitzen:

(1) Wenn die Abbruchbedingung eine Lücke hat, kann eine Endlosschleife entstehen.
(2) Die Laufzeit ist teilweise schwer abzuschätzen.
(3) Durch die schwer zu erkennende Laufzeit können leicht Algorithmen mit exponentieller Laufzeit entstehen!

IV

Datenbanken

Vorwort und Übersicht

Eine Datenbank besteht aus einer Anzahl von Tabellen (sog. **Relationen**, die **Daten** zu bestimmten Themen enthalten. Mit Hilfe dieser Daten ist es möglich, **Informationen** zu generieren.

In diesem Halbjahr geht es zunächst darum, wie man eine Datenbank designt. Dazu erstellt man zunächst ein **Entity-Relationship-Diagramm**, das alle wesentlichen Informationen und Beziehungen enthält. Aus dem ER-Diagramm leitet man dann das sog. **Relationenmodell** ab, das angibt, welche Tabellen benötigt werden und welche Spaltenüberschriften verwendet werden müssen. Jede Tabelle/Relation benötigt einen **Primärschlüssel**, anhand dessen man die einzelnen **Datensätze** eindeutig identifizieren kann. Beziehungen zwischen Entitäten werden dabei über sog. **Fremdschlüssel** modelliert.

Damit ist das Datenbank-Design abgeschlossen und die anfangs leere Datenbank kann mit Daten gefüllt werden. Sind nun Daten vorhanden, so können mit Hilfe der Sprache **SQL** sog. **Abfragen** gegen die Datenbank getätigt werden, um Informationen zu erlangen. Mit **Aggregat-Funktionen**, Formeln und **Gruppierungen** können statistische Daten gewonnen werden, während es **Joins** erlauben, Informationen zu erhalten, die über mehrere Tabellen verteilt sind.

Weitere Themen der Q2 sind **Relationale Algebra**, **Webdatenbank-Projekt** und **Datenschutz und Datensicherheit**. Welches dieser Themen für das Landesabitur relevant ist, wird vom Kultusministerium per Erlass festgelegt.

1 Entitäten und Relationen

Eine Datenbank hat immer den Zweck, einen bestimmten Ausschnitt der realen Welt im Computer abzubilden. Dabei ist die Haupt-Aufgabe zu entscheiden, welche Daten für die Problemstellung relevant sind und welche unwichtig: Sowohl in der Kundendatenbank eines Autoverkäufers als auch in der Patientendatenbank eines Arztes sind Personen gespeichert. Aber in beiden Szenarien sind höchst unterschiedliche Daten relevant: Die Krankenversicherung spielt für den Autoverkäufer keine Rolle, während der Arzt sich nicht um die Konto-Verbindung des Patienten schert. Und für beide ist die Schuhgröße der Person im höchsten Maße irrelevant.

1.1 Daten und Informationen

Zunächst einmal müssen wir klären, was »Daten« (im Sinne der Informatik) eigentlich sind und welcher Zusammenhang zum Begriff »Information« besteht:

Information 1.1

(a) Ein **Datum** (Mehrzahl: **Daten**) ist eine Zeichenkette. Da jede Zeichenkette als Binärzahl codiert werden kann, bestehen Daten also im Wesentlichen aus Nullen und Einsen.

(b) Durch Hinzunehmen eines Kontextes kann ein Datum als **Information** interpretiert werden.

Beispiel 1.2

(a) `183` ist ein Datum (als Bytes im ANSI-Code: `00110001 00111000 00110011`). Wenn man weiß, dass es sich um eine Körpergröße in Zentimetern handelt, erhält man eine andere Information als wenn es sich um einen Kontostand handelt.

(b) Auch `R7B31` ist ein Datum. Im Kontext einer Bibliothek könnte sich dahinter die Information »Reihe 7, Buch Nummer 31« handeln.

#

Bemerkung 1.3 Es ist äußerst wichtig zu verstehen, dass pure Daten nutzlos werden, wenn man den Kontext nicht mehr weiß. Hinter der Bytefolge `00110001 00111000 00110011` aus dem Beispiel kann sich eine beliebige Information verbergen (z.B. der Anfang eines Musikstücks, die Hintergrundfarbe einer Website, ...). #

1.2 Das ER-Modell

Datenbanken dienen dazu, Daten nicht nur zu speichern, sondern auch miteinander zu verknüpfen:

Beispiel 1.4 Zum Kunden »Max Mustermann« gehört die Kundennummer `K34L1294`, das Geburtsdatum `05.03.1992` und die Adresse »Hauptstr. 1439, 10115 Berlin«. Er ist mit der Kundin »Marie Mustermann« (Kundennummer `K2744t103` geboren am `25.11.1994`, wohnhaft an derselben Adresse) seit dem `11.11.2011` verheiratet. #

Offenbar haben wir es hier mit zwei »Dingen« (Max und Marie) zu tun, die einerseits bestimmte Eigenschaften haben (Kundennummer, Geburtsdatum, Adresse) und die miteinander in Beziehung stehen (verheiratet).

Die entsprechenden Fachbegriffe dazu lauten:

Information 1.5

(a) Eine **Entität** ist ein Objekt der realen Welt, z.B. eine Person, ein Gegenstand, eine Stadt aber auch ein nicht-materielles Ding wie eine Reservierung, ein Prozess, ein Flug von Rio nach Rom u.ä.

(b) Gleichartige Entitäten bilden einen **Entitätstypen**, z.B. alle Angestellten, alle Flüge, alle Klassenräume usw. Ein Entitätstyp wird stets *im Singular* benannt, z.B. »Angestellter«, »Flug«, »Klassenraum«, ...

(c) Eine **Beziehung** besteht zwischen zwei Entitäten, z.B. »Cora Müller arbeitet in der PR-Abteilung«.

(d) Gleichartige Beziehungen bilden einen **Beziehungstyp** zwischen zwei Entitätstypen, z.B. »arbeitet in Abteilung« zwischen den Entitätstypen »Mitarbeiter« und »Abteilung«. Ein Beziehungstyp heißt **rekursiv**, wenn ein E-Typ mit sich selbst in Beziehung steht.

(e) Ein **Attribut** ist eine Eigenschaft eines Entitäts- oder eines Beziehungstyps, die alle Entitäten bzw. Beziehungen dieses Typs aufweisen, z.B. »Vorname« des E-Typs »Mitarbeiter« oder »Datum« des B-Typs »hat bestellt«. Dabei können die konkreten Werte der jeweiligen Attribute für jede Entität bzw. jede Beziehung verschieden sein.

Die Werte der Attribute stellen die eigentlichen Daten dar, die E-Typen und B-Typen bilden die Verknüpfungen zwischen diesen Daten!

1.3 ER-Diagramme

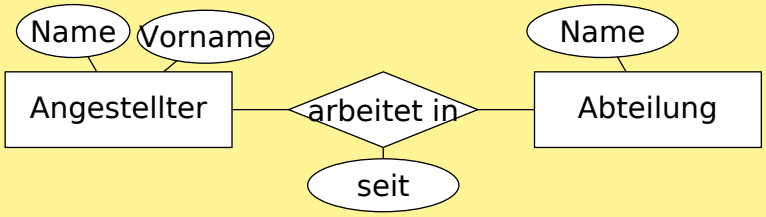
Wenn man die Entitätstypen und Beziehungstypen für die Datenbank identifiziert hat, stellt man diese in einem sog. »Entity-Relationship-Diagramm« dar:

Information 1.6 Ein **Entity-Relationship-Diagramm** (kurz: **ER-Diagramm** oder **ERD**) ist eine grafische Darstellung der Entitäts- und Beziehungstypen sowie der zugehörigen Attribute.

(a) E-Typen werden als **Rechtecke** dargestellt

(b) B-Typen werden als **Raute** inmitten einer Verbindungslinie zwischen zwei E-Typen dargestellt

(c) Attribute werden als **Ovale** dargestellt, die mit dem jeweiligen E-Typ bzw. R-Typ verbunden sind



1.4 Kardinalitäten und Optionalitäten

Eine Grund-Schwierigkeit beim Umgang mit Beziehungstypen liegt darin, dass man Beziehungen stets in *zwei Richtungen* lesen kann: »Der Wagen gehört der Person.« vs. »Die Person besitzt den Wagen«. Genau genommen hat man es jeweils mit zwei Beziehungen zu tun.

Für die Umsetzung eines Realitäts-Ausschnittes in eine Datenbank sind die sog. »Kardinalitäten« und »Optionalitäten« von entscheidender Bedeutung:

Information 1.7 Für jeden Beziehungstyp

gibt es zwei Lesarten:

(a) »Jeder A (muss/kann) in Beziehung R stehen mit (einem/beliebig vielen) B.«

Im ERD:

(b) »Jeder B (muss/kann) in Beziehung R stehen mit (einem/beliebig vielen) A.«

Im ERD:

In einem ERD werden beide Beschriftungen parallel verwendet:

Die Mengenangabe `1` bzw. `n` heißt **Kardinalität** und das Hilfsverb `kann` bzw. `muss` heißt **Optionalität**.

Je nach Kardinalitäten spricht man von einer **1:1-Beziehung**, einer **1:n-Beziehung** oder einer **n:m-Beziehung**.

Beispiel 1.8 In der Datenbank einer Bücherei werden Bücher verwaltet. Jedes Buch hat eine ISBN, einen Titel und wurde von genau einem Autor verfasst. Jeder Autor muss mindestens ein Buch geschrieben haben (sonst wäre er kein Autor). Über die Autoren werden außerdem noch Name, Vorname und Herkunftsland gespeichert. Jedes Buch wird außerdem einem gewisse Genre (Fantasy, Historisch, Krimi, ...) zugeordnet. Jedes Genre hat eine Bezeichnung und einen Beschreibungstext. #

2 Das Relationenmodell

Das Entity-Relationship-Diagramm, das man nach Analyse der Problemstellung entwickelt, beinhaltet alle für die Problemstellung wesentlichen Informationen über die Beziehungen zwischen den Daten.

Um dieses Diagramm im Computer abzubilden, überführt man es in ein sogenanntes »Relationenmodell«. Diese Relationen können dann als Tabellen im Computer abgespeichert werden.

2.1 Relationen und Schlüssel

Definition 2.1 *Eine* **Relation** *ist eine Tabelle mit mindestens einer Spalte. Die Überschriften der Spalten heißen* **Attribute** *der Relation. Die Zeilen der Relation heißen* **Datensätze**.

Relationen werden in der Form »Name(Attribut1, Attribut2, ...)« angegeben.

Beispiel 2.2 Die Relation `Kunde(Vorname, Nachname, KNr, Strasse, Plz)` könnte z.B. folgende Tabelle sein:

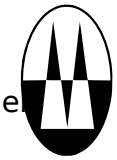
Kunde				
Vorname	**Nachname**	**KNr**	**Strasse**	**Plz**
Thomas	Klein	27435	Hauptstraße 943	11010
Max	Mustermann	50374	Goethestr. 8	65597
Marie	Mustermann	50374	Goethestr. 8	65597

#

Um diese Tabellen sinnvoll einsetzen zu können, muss man in der Lage sein, einzelne Datensätze auf eindeutige Weise auszuwählen:

Definition 2.3*Jede Relation benötigt einen* **Primärschlüssel**. *Ein Primärschlüssel besteht aus einem oder mehreren Attributen, die einen Datensatz eindeutig identifizieren. Die Attribute, die den Primärschlüssel bilden, werden unterstrichen (sowohl im ERD als auch im Relationenmodell).*

Beispiel 2.4 Bei der Relation `Kunde` wäre das Paar (Vorname,Nachname) ein möglicher Schlüsselkandidat, allerdings nur, wenn man ausschließen kann, dass dieselbe Kombination niemals zweimal vorkommt. Daher ist es sinnvoller, die Kundennummer als Primärschlüssel zu verwenden.



Die Relation lautet daher `Kunde(Vorname, Nachname, KNr, Strasse, Plz)` #

Bemerkung 2.5 In der Praxis verwendet man meistens eine eindeutige ID als Primärschlüssel. Z.B. erhält jeder Kunde eine Kundennummer, jeder Auftrag eine Auftragsnummer, jede Rechnung eine Rechnungsnummer. #

Um die Beziehungen zwischen Entitäten abzubilden, kommen in Relationen auch häufig die Primärschlüssel von anderen Relationen vor (siehe nächster Abschnitt):

Definition 2.6 *Ein Attribut einer Relation, das Primärschlüssel einer* anderen *Relation ist, heißt* **Fremdschlüssel**. *Fremdschlüssel werden im ERD nicht aufgeführt. Im Relationenmodell wird ihnen ein nach oben zeigender Pfeil (↑) vorangestellt.*

Beispiel 2.7 Z.B. könnte die Relation `Kunde` um den Fremdschlüssel `verheiratetMit` ergänzt werden. In dieser Spalte würde dann die Kundennummer des Kunden stehen, der mit dem Kunden verheiratet ist:

Kunde					
Vorname	**Nachname**	**KNr**	**Strasse**	**Plz**	**↑verheiratetMit**
Thomas	Klein	27435	Hauptstraße 943	11010	
Max	Mustermann	50374	Goethestr. 8	65597	20745
Marie	Mustermann	20745	Goethestr. 8	65597	50374

#

2.2 Überführen des ERD in das Relationenmodell

Aus einem Entity-Relationship-Diagramm kann man reativ einfach das Relationenmodell ableiten:

Verfahren 2.8

Gegeben: ER-Diagramm

Gesucht: Relationenmodell

Lösung:
(1) Entitätstypen: Jeder Entitätstyp bildet eine Relation. Die Attribute des E-Typen sind die Attribute der Relation.

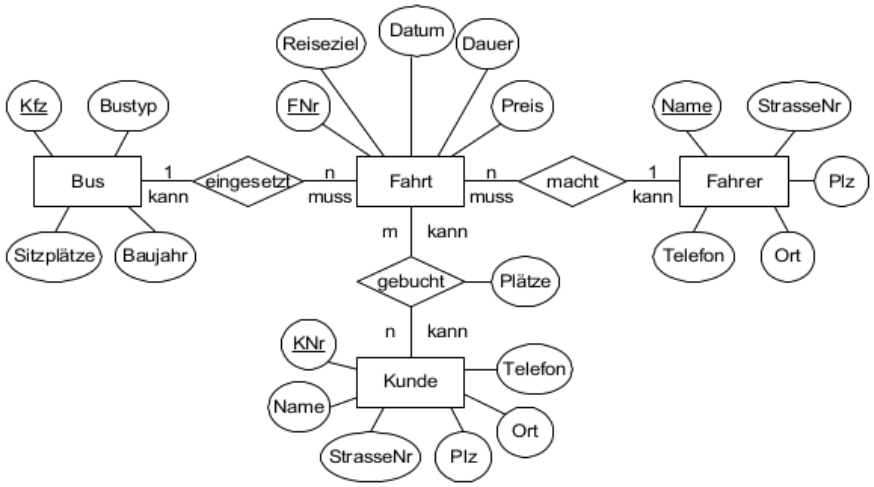
(2) 1:n-Beziehungstypen: Die Relation der `n`-Seite erhält den Primärschlüssel der anderen Relation als Fremdschlüssel. Außerdem erhält sie alle Attribute des Beziehungstyps.

(3) 1:1-Beziehungstypen: Wie bei 1:n und es ist egal, welche Seite den Fremdschlüssel und die Beziehungs-Attribute erhält.

(4) n:m-Beziehungstypen: Jeder n:m-Beziehungstyp bildet eine weitere Relation. Diese erhält die beiden Primärschlüssel der in Beziehung stehenden Entitätstypen als Fremdschlüssel. Die beiden Fremdschlüssel bilden gemeinsam den Primärschlüssel der neuen Relation. Die neue Relation erhält außerdem alle Attribute des Beziehungstyps.

Bemerkung 2.9 Bei der Überführung des ERD in das Relationenmodell spielen die Optionalitäten keine Rolle. Nur die Kardinaliäten sind entscheidend. #

Beispiel 2.10 Wir überführen das folgende ER-Diagramm (Quelle: Glossar Landesabitur Informatik, 2017) in das Relationenmodell:



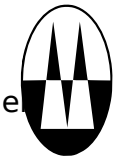
(1) Zunächst erhalten wir zu jedem Entitätstyp eine Relation:

- Bus(Kfz, Bustyp, Sitzplätze, Baujahr)
- Fahrt(FNr, Reiseziel, Datum, Dauer, Preis)
- Fahrer(Name, StrasseNr, Plz, Telefon, Ort)
- Kunde(KNr, Name, StrasseNr, Plz, Ort, Telefon)

(2) Jetzt bilden wir die 1:n-Relationen mittels Fremdschlüsseln ab:

- Fahrt(FNr, Reiseziel, Datum, Dauer, Preis, ↑BusKfz, ↑FahrerName)

(3) Zuletzt erhalten wir für den »gebucht«-Beziehungstyp eine neue Relation:



- Buchung(↑FNr, ↑KNr, Plätze)

Insgesamt erhalten wir also folgendes Relationenmodell:

- Bus(Kfz, Bustyp, Sitzplätze, Baujahr)
- Fahrt(FNr, Reiseziel, Datum, Dauer, Preis, ↑BusKfz, ↑FahrerName)
- Fahrer(Name, StrasseNr, Plz, Telefon, Ort)
- Kunde(KNr, Name, StrasseNr, Plz, Ort, Telefon)
- Buchung(↑FNr, ↑KNr, Plätze)

#

3 Abfragen mit SQL

Wir haben nun gesehen, wie man einen für die Informatik relevanten Teil der realen Welt als Entity-Relationship-Diagramm modelliert und dieses in ein Relationenmodell überführt, das die Tabellen liefert, aus denen die Datenbank letztendlich besteht.

Ist die Datenbank nun implementiert und mit Daten gefüllt, so besteht der nächste Schritt darin, Informationen aus der Datenbank zu extrahieren. Für relationale Datenbanken wurde zu diesem Zweck die Sprache SQL (»Structured Query Language«) entwickelt.

3.1 Der SELECT-Befehl

SQL bietet verschiedene Befehle an. Für uns ist nur der SELECT-Befehl relevant:

Information 3.1 Der **SELECT**-Befehl dient dazu, Daten aus einer Datenbank abzufragen. Der grundlegende Aufbau ist

```
1   SELECT spalte1, spalte2, ..., spalteN
2   FROM tabelle
3   WHERE bedingung1
4   AND/OR bedingung2
5   ...
6   AND/OR bedingungN
```

Dazu folgende Bemerkungen:

- Die Groß- und Kleinschreibung spielt dabei keine Rolle.
- Schreibt man `SELECT * FROM ...`, so werden alle Spalten der Tabelle ausgegeben.
- Strings werden in einfache Hochkommata geschrieben, z.B. `'Max Mustermann'`.

Information 3.2 Für die Bedingungen gibt es folgende Operatoren:

=	Gleich.
<>	Ungleich.
<	Kleiner als.
>	Größer als.

`<=`	Kleiner oder gleich.
`>=`	Größer oder gleich.
`BETWEEN`	`WHERE Alter BETWEEN 13 and 19` wählt alle Datensätze aus, bei denen das Alter zwischen 13 und 19 liegt (jeweils einschließlich).
`LIKE`	`WHERE Name LIKE 'Max Muster%'` wählt alle Datensätze aus, bei denen der Name mit »Max Muster« anfängt und dann beliebig weitergeht, z.B. »Max Mustermann«, »Max Musterfrau«, etc.
`IN`	`WHERE Alter in (17,19,23,29)` wählt alle Datensätze aus, bei denen das Alter einem der Werte in der Klammer entspricht.

Bemerkung 3.3 Die `WHERE`-Klausel filtert also die Datensätzen. Es bleiben nur die Datensätze übrig, die die Bedingung erfüllen. #

3.2 Formeln und Aggregatfunktionen

Es ist möglich, mit ausgelesenen Werten zu rechnen. Dazu kann man die normalen Rechenoperatoren +, –, ∗ und / verwenden.

Beispiel 3.4 Man kann eine Temperatur x in Grad Fahrenheit in eine Temperatur y in Grad Celsius umrechnen mit Hilfe der Formel

$$y = \frac{5}{9} \cdot (x - 5) - 15$$

Z.B. erhält man für $x = 14°\,\mathrm{F} \to \frac{5}{9} \cdot (14 - 5) - 15 = \frac{5}{9} \cdot 9 - 15 = -10°\,\mathrm{C}$.

Angenommen, wir haben eine Tabelle `Wetter`, in der Temperaturen in Fahrenheit stehen (in der Spalte `Temperatur`), dann erhalten wir folgendermaßen die Temperatur in Celsius:

```
1    SELECT 5*(Temperatur-5)/9-15
2    FROM Wetter
```

#

Information 3.5 Ausgelesene Werte können verarbeitet und miteinander kombiniert werden, um neue Informationen zu generieren.

(a) Zugelassen sind Formeln mit den Grundrechenarten und Klammern.

(b) Zugelassen sind außerdem die folgenden **Aggregatfunktionen**: `AVG` (Durchschnitt aller Werte), `COUNT` (Anzahl), `MAX` (Maximaler Wert), `MIN` (minimaler Wert), `SUM` (Summe aller Werte). Diese können auch beliebig kombiniert werden.

(c) Ergebnisspalten können mit dem Schlüsselwort `AS` umbenannt werden.

Beispiel 3.6 Der folgende Code liefert das BIP pro Kopf und gibt es in einer Tabelle der Form (`Land`, `Pro-Kopf-BIP`) an:

```
1   SELECT Name, bip/einwohner AS 'Pro-Kopf-BIP'
2   FROM cia
```

#

Bemerkung 3.7 Viele Aggregat-Funktionen machen erst im Zusammenhang mit Gruppierungen Sinn (siehe nächster Abschnitt). #

3.3 Sortierung und Limitierung

Um die Ausgabe übersichtlicher zu gestalten, kann man sich die Werte sortiert ausgeben lassen:

Information 3.8 Mit der `ORDER BY`**-Klausel** kann man Daten aufsteigend (`ASC`, Standard) oder absteigend (`DESC`) sortieren. Es können mehrere Spalten angegeben werden, die dann nacheinander als Sortierkriterium verwendet werden, d.h., wenn der Wert der ersten Spalte gleich ist, wird der Wert der zweiten Spalte verglichen usw.

Beispiel 3.9 Wir wollen die Bundesländer Deutschlands nach Einwohnerzahl sortiert ausgeben:

```
1   SELECT Land, Einwohnerzahl
2   from BRD
3   ORDER BY Einwohnerzahl
```

#

Information 3.10 Mit der `LIMIT`**-Klausel** kann man die Anzahl der Ergebnis-Datensätze auf eine bestimmte Anzahl begrenzen.

Beispiel 3.11 Wir suchen das Bundesland mit den meisten Einwohnern:

```
1   SELECT Land, Einwohnerzahl
2   from BRD
3   ORDER BY Einwohnerzahl DESC
4   limit 1
```

#

3.4 Gruppieren

Ein sehr mächtiges Werkzeug stellt das Gruppieren dar:

Information 3.12 Mit der `GROUP BY`**-Klausel** kann man Daten nach bestimmten Attributen gruppieren. Das bedeutet, dass alle Datensätze, bei denen diese Attribute den gleichen Wert haben, als eine Gruppe behandelt werden. Aggregatfunktionen werden auf jede Gruppe getrennt angewendet.

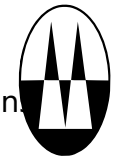
Beispiel 3.13 Gesucht sind die jeweiligen Durchschnittsgehälter der Spieler der einzelnen Bundesliga-Mannschaften:

```
1   SELECT Mannschaft,AVG(Gehalt)
2   FROM Spieler
3   GROUP BY Mannschaft
```

#

Beispiel 3.14 Es ist nicht möglich, auf Attribute zuzugreifen, die innerhalb einer Gruppe verschiedene Werte haben. Zum Beispiel ist die Anweisung

```
1   SELECT Name,Mannschaft,AVG(Gehalt)
2   FROM Spieler
3   GROUP BY Mannschaft
```



nicht möglich, weil innerhalb der Mannschafts-Gruppen die Spieler verschiedene Namen haben. #

Information 3.15 Beim Gruppieren kommt es häufig vor, dass man die Ergebnisse filtern möchte. Dies macht man normalerweise mit der `WHERE`-Klausel, jedoch können bei `WHERE` keine aggregierten Werte verwendet werden, da die `WHERE`-Bedingung angewendet wird, *bevor* die aggregierten Werte ermittelt wurden.

Möchte man aggregierte Werte in einer Bedingung verwenden, so muss man die **`HAVING`-Klausel** verwenden.

Beispiel 3.16 Wir suchen eine Auflistung aller Teams inklusive Durchschnittsgehalt der Spieler, bei denen das Durchschnittsgehalt über 2 Millionen liegt:

```
1   SELECT Mannschaft,AVG(Gehalt)
2   FROM Spieler
3   GROUP BY Mannschaft
4   HAVING AVG(Gehalt)>2000000
```

#

3.5 Joins

Joins (»Verbindungen«) kommen immer dann ins Spiel, wenn man die Daten aus mehreren Tabellen miteinander kombinieren möchte.

Information 3.17 Man kann in einem SELECT-Befehl mehrere Tabellen aufführen (durch Komma getrennt). Diese Tabellen werden kombiniert in dem Sinne, das alle Datensätze der einen Tabelle mit allen Datensätzen der anderen Tabelle kombiniert werden.

Die neue Tabelle hat die Spaltennamen der Ursprungstabellen. Bei gleichen Attributsnamen kann man Tabellenname.Attribut schreiben.

Beispiel 3.18 Für die beiden Tabellen (die Gehälter sind erfunden)

Team		
Name	**Stadt**	**TrainerName**
Bayern München	München	Carlo Ancelotti
Eintracht Frankfurt	Frankfurt	Niko Kovac

Trainer	
Name	**Gehalt**
Niko Kovac	200000
Manuel Baum	100000
Carlo Ancelotti	1500000

ergibt die Abfrage

```
1    SELECT *
2    FROM Team, Trainer
```

die folgende Ausgabe:

Team, Trainer				
Team.Name	**Stadt**	**TrainerName**	**Trainer.Name**	**Gehalt**
Bayern München	München	Carlo Ancelotti	Niko Kovac	200000
Bayern München	München	Carlo Ancelotti	Manuel Baum	100000
Bayern München	München	Carlo Ancelotti	Carlo Ancelotti	1500000
Eintracht Frankfurt	Frankfurt	Niko Kovac	Niko Kovac	200000
Eintracht Frankfurt	Frankfurt	Niko Kovac	Manuel Baum	100000
Eintracht Frankfurt	Frankfurt	Niko Kovac	Carlo Ancelotti	1500000

#

Dieses Kombinieren aller Datensätze miteinander ergibt natürlich in den meisten Fällen keinen Sinn. Stattdessen führt man einen »Join« durch:

Information 3.19 Ein **Join** kombiniert zwei oder mehr Tabellen und legt dabei fest, welche Attribute gleich sein müssen, damit die Datensätze kombiniert werden. Ein **Natural Join** ist einer, bei dem die Namen der Attribute gleich sind.

Beispiel 3.20 Wir wollen bei obigem Beispiel eine Tabelle aller Mannschaften mit dem jeweiligen Trainer und dem Gehalt des Trainers erhalten. Wir benötigen also einen Join über die Attribute `TrainerName` aus der Tabelle `Team` und `Name` aus der Tabelle `Trainer`:

```
1   SELECT Team.Name, TrainerName, Gehalt
2   FROM Team, Trainer
3   WHERE Team.TrainerName=Trainer.Name
```

ergibt

Team, Trainer		
Team.Name	**TrainerName**	**Gehalt**
Bayern München	Carlo Ancelotti	1500000
Eintracht Frankfurt	Niko Kovac	200000

#

Bemerkung 3.21 Dieses Kapitel gibt nur einen knappen Überblick über die für das Landesabitur relevanten SQL-Sprachkonstrukte. Genau wie bei der Programmierung ist es wichtig, diese Konzepte anzuwenden, um Aufgaben damit zu lösen. Es gibt im Internet eine ganze Reihe hervorragender Übungsseiten für SQL. Suchen Sie bspw. nach »sqlzoo« oder »sql imoodle«. #

4 Relationale Algebra

Datenbanken können mit Hilfe von SQL-Befehlen abgefragt werden. Wir haben gesehen, dass ein solcher `SELECT`-Befehl relativ lange und kompliziert werden kann.

Um Beziehungen zwischen Daten zu untersuchen, ist es sinnvoll, eine alternative Beschreibungssprache für Abfragen zu finden. Diese soll aus möglichst einfachen Grundkomponenten bestehen und trotzdem mächtig genug sein, alle Arten von Abfragen darzustellen. Diese Aufgabe erfüllt die sog. *relationale Algebra* par excellence.

Alle Beispiele in diesem Kapitel beziehen sich auf die Relationen in **Abb. 11**.

Klassen

ID	Klassenlehrer	Raum
5e	MU	13
7a	HA	21
7b	GR	25

Lehrer

ID	Nachname	Vorname
MU	Müller	Andreas
HA	Hansen	Doris
GR	Gros	Johannes
TI	Tillisch	Petra

Schueler

ID	Nachname	Vorname	Klasse
107	Mann	Julia	7b
211	Braun	Zoe	5e
143	Kremer	Jan	5e

Abbildung 11 Alle verwendeten Relationen dieses Kapitels

4.1 Projektionen und Selektionen

Zunächst machen wir eine relativ abstrakte Definition:

Definition 4.1 *Eine* **Abbildung** f *bildet jedes Element* a *einer Menge* A *eindeutig auf ein Element* $f(a)$ *einer Menge* B *ab.*

Beispiel 4.2 Wenn $A = \mathbb{R}$ und $B = \mathbb{R}$, dann sind die Abbildungen von A nach B klassische Funktionen aus dem Mathematik-Unterricht. #

Wir interessieren uns nun für Abbildungen, die Relationen auf Relationen abbilden:

Definition 4.3

(a) *Die* **Projektion** $\pi_{a_1,a_2,\ldots,a_n}$ *bildet eine Relation* r *auf eine neue Relation* r' *ab, die nur noch die Attribute* a_1, a_2, ..., a_n *enthält. Alle anderen Attribute werden entfernt.*

(b) *Die* **Selektion** $\sigma_{Bedingung}$ *bildet eine Relation* r *auf eine neue Relation* r' *ab, die nur noch die Zeilen enthält, die die* `Bedingung` *erfüllen. Alle anderen Zeilen werden entfernt.*

Beispiel 4.4

(a) $\pi_{\texttt{ID,Vorname(Lehrer)}}$ ergibt die Relation

$\pi_{\texttt{ID,Vorname(Lehrer)}}$	
ID	**Vorname**
MU	Andreas
HA	Doris
GR	Johannes
TI	Petra

(b) $\sigma_{\texttt{Klasse='5e'(Schueler)}}$ ergibt die Relation

$\sigma_{\texttt{Klasse='5e'(Schueler)}}$			
ID	**Nachname**	**Vorname**	**Klasse**
211	Braun	Zoe	5e
143	Kremer	Jan	5e

(c) $\pi_{\texttt{Nachname}}(\sigma_{\texttt{Klasse='5e'(Schueler)}})$ ergibt die Relation

$\pi_{\texttt{Nachname}}(\sigma_{\texttt{Klasse='5e'(Schueler)}})$
Nachname
Braun
Kremer

#

Bemerkung 4.5 Mit Hilfe von Projektionen und Selektionen lassen sich also einfache SELECT-Anweisungen der Form

```
1    SELECT attribut1, attribut2, ..., attributN
2    FROM tabelle
3    WHERE bedingung
```

nachbauen. #

4.2 Kreuzprodukt und Join

Definition 4.6 *Für zwei Relationen R und S definieren wir:*

(a) Das **kartesische Produkt** $R \times S$ *ist die Relation, bei der alle Zeilen von R mit allen Zeilen von S kombiniert werden.*

(b) Der **Join** $R \underset{a=b}{\bowtie} S$ *ist die Relation, bei der alle Zeilen von R mit allen Zeilen von S kombiniert werden, bei denen die Bedingung erfüllt ist, dass der Wert des Attributs a von R identisch ist mit dem Wert des Attributs b von S. Haben die beiden Attribute denselben Namen, so handelt es sich um einen* **natural join**, *geschrieben $R \bowtie S$.*

Beispiel 4.7

(a) `Klassen` × `Schueler` ergibt die (ziemlich unsinnige) Relation

Klassen× Schueler						
ID	**Klassenlehrer**	**Raum**	**Schueler.ID**	**Nachname**	**Vorname**	**Klasse**
5e	MU	13	107	Mann	Julia	7b
5e	MU	13	211	Braun	Zoe	5e
5e	MU	13	143	Kremer	Jan	5e
7a	HA	21	107	Mann	Julia	7b
7a	HA	21	211	Braun	Zoe	5e
7a	HA	21	143	Kremer	Jan	5e
7b	GR	25	107	Mann	Julia	7b
7b	GR	25	211	Braun	Zoe	5e
7b	GR	25	143	Kremer	Jan	5e

(b) Deutlich sinnvoller ist $\texttt{Klassen} \underset{ID=Klasse}{\bowtie} \texttt{Schueler}$:

Klassen $\underset{ID=Klasse}{\bowtie}$ Schueler						
ID	**Klassenlehrer**	**Raum**	**Schueler.ID**	**Nachname**	**Vorname**	**Klasse**
5e	MU	13	211	Braun	Zoe	5e
5e	MU	13	143	Kremer	Jan	5e
7b	GR	25	107	Mann	Julia	7b

#

5 Datenschutz und Datensicherheit

»Big Data«, »maschinelles Lernen«, »Google Analytics«, »Alexa« und »künstliche Intelligenz«: Die Verfügbarkeit vieler Daten erlaubt es, Informationen aller Art zu gewinnen, zu gebrauchen aber auch zu missbrauchen.

In diesem Kapitel geht es einerseits darum, welche Gesetze verhindern, dass persönliche Daten beliebig gespeichert und verarbeitet werden (Datenschutz) und andererseits, mit welchen technischen Hilfsmitteln Datendiebstahl verhindert werden kann (Datensicherheit).

5.1 Datenschutz

Definition 5.1 *Der Begriff* **Datenschutz** *ist nicht einheitlich definiert. Darum hier einige Möglichkeiten:*

(a) »Zweck dieses Gesetzes ist es, den Einzelnen davor zu schützen, dass er durch den Umgang mit seinen personenbezogenen Daten in seinem Persönlichkeitsrecht beeinträchtigt wird.«

(§1 (1) Bundesdatenschutzgesetz (BDSG))

(b) »Diese Verordnung enthält Vorschriften zum Schutz natürlicher Personen bei der Verarbeitung personenbezogener Daten und zum freien Verkehr solcher Daten.«

(§1 (1) Europäische Datenschutz-Grundverordnung (EUDSGVO))

Insgesamt kann man festhalten:

- *Datenschutz bezieht sich immer auf natürliche, einzelne Personen, also weder auf Firmen, noch Vereine o.ä. Geschützt wird also jeder Einzelne von uns!*
- *Datenschutz schränkt die Verarbeitung sog. »personenbezogener Daten« ein:*

  »Personenbezogene Daten sind Einzelangaben über persönliche oder sachliche Verhältnisse einer bestimmten oder bestimmbaren natürlichen Person (Betroffener).« (§3 (1) BDSG)
- *Das BDSG unterscheidet zusätzlich noch* **besondere Arten personenbezogener Daten***:*

  »Besondere Arten personenbezogener Daten sind Angaben über die rassische und ethnische Herkunft, politische Meinungen, religiöse oder philosophische Überzeugungen, Gewerkschaftszugehörigkeit, Gesundheit oder Sexualleben.« (§3 (9) BDSG)

Diese besonderen Arten personenbezogener Daten werden besonders geschützt (§28 (6)-(9) BDSG).

- *Unter* **Verarbeitung** *versteht man »Speichern, Verändern, Übermitteln, Sperren und Löschen personenbezogener Daten« (§3 (4) BDSG).*
- *Datenschutz betrifft nur* automatisiertes *Erheben, Verarbeiten und Nutzen von Daten. Handschriftliche Aufzeichnungen, Ausdrucke etc. fallen also* nicht *unter das Datenschutzgesetz.*

Im folgenden beziehen wir uns auf die Paragraphen des BDGS. Es geht darum, einen Überblick über die wichtigsten Paragraphen zu geben — ohne Anspruch auf Vollständigkeit. Um einen konkreten Fall zu prüfen, muss man immer das BDSG konsultieren.

Information 5.2 §3a: Datenvermeidung und Datensparsamkeit

»Die Erhebung, Verarbeitung und Nutzung personenbezogener Daten und die Auswahl und Gestaltung von Datenverarbeitungssystemen sind an dem Ziel auszurichten, so wenig personenbezogene Daten wie möglich zu erheben, zu verarbeiten oder zu nutzen. Insbesondere sind personenbezogene Daten zu anonymisieren oder zu pseudonymisieren, soweit dies nach dem Verwendungszweck möglich ist und keinen im Verhältnis zu dem angestrebten Schutzzweck unverhältnismäßigen Aufwand erfordert.«

(§3a BDSG)

Verkürzt: Unternehmen dürfen nur Daten erheben/verarbeiten, die für den unmittelbaren Zweck notwendig sind — wenn möglich anonymisiert.

Information 5.3 §6/6a: Rechte des Betroffenen

»Die Rechte des Betroffenen auf Auskunft (§§ 19, 34) und auf Berichtigung, Löschung oder Sperrung (§§ 20, 35) können nicht durch Rechtsgeschäft ausgeschlossen oder beschränkt werden.«

(§6 (1) BDSG)

»Entscheidungen, die für den Betroffenen eine rechtliche Folge nach sich ziehen oder ihn erheblich beeinträchtigen, dürfen nicht ausschließlich auf eine automatisierte Verarbeitung personenbezogener Daten gestützt werden, die der Bewertung einzelner Persönlichkeitsmerkmale dienen. Eine ausschließlich auf eine automatisierte Verarbeitung gestützte Entscheidung liegt insbesondere dann vor, wenn keine inhaltliche Bewertung und darauf gestützte Entscheidung durch eine natürliche Person stattgefunden hat.«

(§6a (1) BDSG)

Verkürzt: Vollautomatische Entscheidungen sind rechtlich als sehr kritisch zu bewerten. Im Einzelfall muss (gerichtlich) geklärt werden, was »erheblich« bedeutet.

Information 5.4 §28: Datenerhebung und -speicherung für eigene Geschäftszwecke Dieser Paragraph ist viel zu umfangreich, um ihn vollständig zu besprechen. Für viele Anwendungsfälle ist er der relevante Paragraph!

»Das Erheben, Speichern, Verändern oder Übermitteln personenbezogener Daten oder ihre Nutzung als Mittel für die Erfüllung eigener Geschäftszwecke ist zulässig

(1) wenn es für die Begründung, Durchführung oder Beendigung eines rechtsgeschäftlichen oder rechtsgeschäftsähnlichen Schuldverhältnisses mit dem Betroffenen erforderlich ist,

(2) soweit es zur Wahrung berechtigter Interessen der verantwortlichen Stelle erforderlich ist und kein Grund zu der Annahme besteht, dass das schutzwürdige Interesse des Betroffenen an dem Ausschluss der Verarbeitung oder Nutzung überwiegt, oder

(3) wenn die Daten allgemein zugänglich sind oder die verantwortliche Stelle sie veröffentlichen dürfte, es sei denn, dass das schutzwürdige Interesse des Betroffenen an dem Ausschluss der Verarbeitung oder Nutzung gegenüber dem berechtigten Interesse der verantwortlichen Stelle offensichtlich überwiegt.

Bei der Erhebung personenbezogener Daten sind die Zwecke, für die die Daten verarbeitet oder genutzt werden sollen, konkret festzulegen.«

(§28 (1) BDSG)

Verkürzt: Datenverarbeitung ist grundsätzlich erlaubt, wenn die Daten für die Erfüllung der eigenen Geschäftszwecke notwendig sind.

»Die Verarbeitung oder Nutzung personenbezogener Daten für Zwecke des Adresshandels oder der Werbung ist zulässig, soweit der Betroffene eingewilligt hat und im Falle einer nicht schriftlich erteilten Einwilligung die verantwortliche Stelle nach Absatz 3a verfährt.«

(§28 (3) BDSG)

Verkürzt: Der Betroffene muss zustimmen, wenn die Daten für andere Zwecke verwendet werden sollen.

Information 5.5 §33/34: Bußgeld- und Strafvorschriften In diesen Paragraphen wird geregelt, unter welchen Umständen Datenschutzverstöße eine Ordnungswidrigkeit

(Folge: Bußgeld) darstellen und unter welchen Umständen eine Straftat (Folge: Geld- oder Freiheitsstrafe).

5.2 **Datensicherheit**

V

Theoretische Informatik

Vorwort

In der theoretischen Informatik untersuchen wir die Grenzen der Leistungsfähigkeit von Computern. Für den Laien erscheint es manchmal, als ob ein Computer alles berechnen kann, wenn er nur genug Zeit hat. Wir werden feststellen, dass dies nicht der Fall ist, sondern dass es Probleme gibt, die Computer prinzipiell nicht lösen können.

Diese Erkenntnis werden wir aber erst gegen Ende des Halbjahres *beweisen* können, da ein mathematischer Beweis eine genaue Definition der verwendeten Begriffe verlangt. Solange wir nicht eindeutig definieren können, was ein *Problem*, ein *Algorithmus* oder ein *Computer* ist, können wir auch nicht hoffen, einen solchen Beweis erbringen zu können.

Abstrakt gesehen, wandelt ein Algorithmus eine Eingabe-Zeichenkette in eine Ausgabe-Zeichenkette um. Der Bubble-Sort-Algorithmus wandelt z.B. die Eingabe-Zeichenkette

```
{4, -2, 5, 7, 3, 0}
```

in die Ausgabe-Zeichenkette

```
{-2, 0, 3, 4, 5, 7}
```

um.

Wie wir sehen, können Zeichenketten eine bestimmte Bedeutung haben, z.B. beschreiben die beiden obigen Zeichenketten Integer-Arrays in der Programmiersprache Java. Man kann alle solchen Zeichenketten zur **formalen Sprache** der Integer-Arrays zusammenfassen. Die **Wörter** dieser Sprache sind dann die syntaktisch korrekten Integer-Arrays.

Formale Sprachen können mit Hilfe von **Grammatiken** oder **Syntax-Diagrammen** definiert werden.

Ein **Automat** ist in der Lage, zu entscheiden, ob ein gegebenes Wort zu einer bestimmten Sprache gehört oder nicht. Die einfachste Variante ist der sog. **endliche Automat**. Obwohl dieses Rechnermodell bereits ziemlich vielseitig ist, wird sich herausstellen, dass dessen Möglichkeiten zu beschränkt sind. Daher werden diese »gepimpt« und zu **Turingmaschinen** erweitert. Mit der Turingmaschine ist ein Automat gefunden, der nach aktueller Forschung »alles berechnen kann, was möglich ist«. Dies ist die **Hypothese von Church**, die besagt, dass jeder Algorithmus als Turingmaschine beschrieben werden kann und umgekehrt. D.h.:

Algorithmus = Turingmaschine

Mit der **universellen Turingmaschine** werden wir schließlich ein theoretisches Modell des Computers erhalten, was den eingangs erwähnten Nachweis ermöglichen wird, dass es Probleme gibt, die von Computern nicht gelöst werden können.

1 Formale Sprachen

In den vergangenen Halbjahren haben wir verschiedenste Computersprachen kennengelernt: HTML, CSS, Java, SQL. Daneben gibt es Unmengen weiterer Computersprachen und täglich werden neue Sprachen entwickelt, um gewisse Sachverhalte beschreiben zu können. Jeder praktisch arbeitende Informatiker entwirft im Laufe seines Lebens Dutzende von Sprachen, auch wenn ihm/ihr das vielleicht nicht immer bewusst ist.

1.1 Alphabet, Wort, Sprache, Grammatik

Definition 1.1

(a) Ein **Alphabet** *ist eine endliche Menge von Zeichen.*

(b) Ein **Wort** *über einem Alphabet Σ ist eine Zeichenkette, deren Zeichen alle in Σ enthalten sind.*

(c) Eine **formale Sprache** *ist eine Menge von Wörtern über einem gemeinsamen Alphabet Σ.*

Beispiel 1.2 Für die Steuerung eines Roboters wird die Sprache L benötigt, die aus allen Wörtern besteht, die aus dem Alphabet

$$\Sigma = \{\texttt{S}, \texttt{L}, \texttt{R}, \texttt{U}, \texttt{D}, \texttt{O}\}$$

(steht für Start, Left, Right, Up, Down, Out) gebildet werden können und die mit `S` beginnen und mit `O` enden. Zu L gehören also z.B. die Wörter `SRRLDDURO`, `SDLO` und `SUUUUO`, nicht aber `SEO` und `LRO`. #

Sprachen können mit Hilfe von »Grammatiken« erzeugt werden:

Definition 1.3

(a) Eine **Grammatik** *G besteht aus*

(1) einem Alphabet N von **Nicht-Terminalsymbolen**,

(2) einem Alphabet T von **Terminalsymbolen** *(terminal = abschließend),*

(3) einer Menge P von **Produktionen** *(Ersetzungsregeln),*

(4) einem **Startsymbol** *S.*

(b) Eine **Produktion** *ist eine Ersetzungsregel der Form $A \to B$. Diese gibt an, dass das Teilwort A durch das Teilwort B ersetzt werden kann. A und B können aus beliebig vielen Terminal- und/oder Nicht-Terminalsymbolen bestehen.*

(c) In einer Produktion kann das **leere Wort** ϵ *verwendet werden. Dies ist das Wort, das aus überhaupt keinem Zeichen besteht.*

(d) Eine **Ableitung** *aus einer Grammatik G ist ein Wort aus Terminalsymbolen, das durch wiederholte Anwendung der Produktionen aus dem Startsymbol erzeugt werden kann.*

Das Ableiten eines Wortes aus dem Startsymbol stellt man am besten als **Ableitungsbaum** *dar.*

(e) Die Sprache $L(G)$ besteht aus allen Wörtern, die aus der Grammatik G abgeleitet werden können. $L(G)$ wird als die **von der Grammatik G erzeugte Sprache** *bezeichnet.*

Beispiel 1.4 Gegeben ist die Grammatik G mit

Nicht-Terminale: $N = \{$`SATZ`, `SUBJEKT`, `PRÄDIKAT`, `OBJEKT`, `ARTIKEL`, `SUBSTANTIV`$\}$

Terminale: $T = \{$`dog`, `mouse`, `cat`, `eats`, `the`, `a`, `likes`$\}$

Produktionen: $P = \{$

`SATZ` → `SUBJEKT PRÄDIKAT OBJEKT`
`SUBJEKT` → `ARTIKEL SUBSTANTIV`
`PRÄDIKAT` → `eats` | `likes`
`OBJEKT` → `ARTIKEL SUBSTANTIV` | ϵ
`SUBSTANTIV` → `dog` | `mouse` | `cat`
`ARTIKEL` → `a` | `the`

$\}$

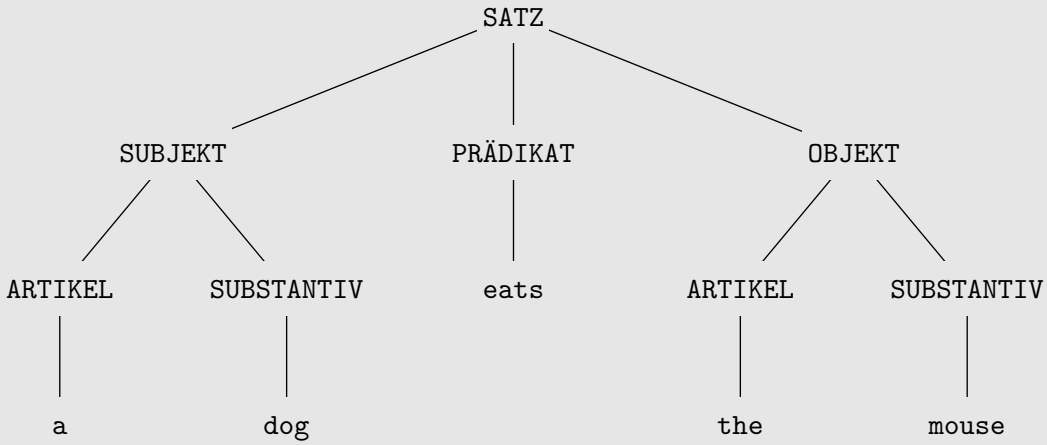
Startsymbol: `SATZ`

Ein | bedeutet, dass es mehrere Möglichkeiten gibt, die linke Seite zu ersetzen. Hier darf man z. B. `PRÄDIKAT` durch `eats` oder durch `likes` ersetzen.

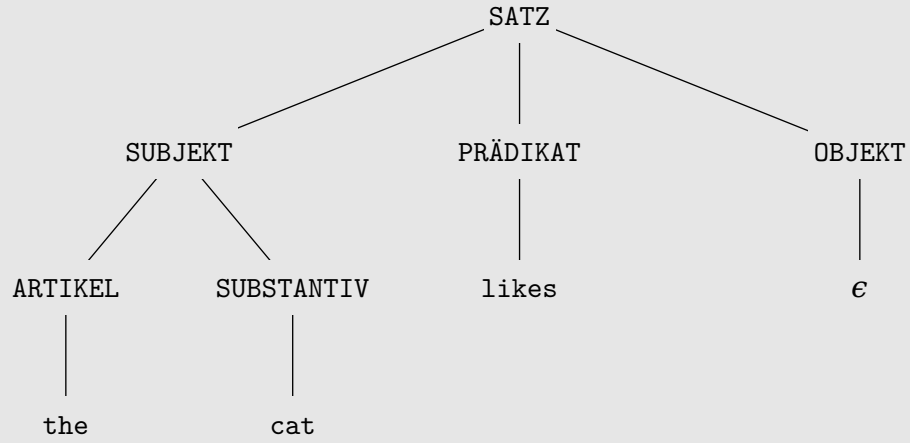
Mit diesen Produktionen kann man die zwei Wörter `a dog eats the mouse` und `the cat likes` aus dem Startsymbol `SATZ` ableiten (vgl. **Abb. 12**). Nicht möglich ist z. B. die Frage `likes a dog a mouse`, weil diese Reihenfolge nicht ableitbar ist. #

Bemerkung 1.5 Die abgeleiteten Wörter müssen nicht zwangsläufig Sinn ergeben. Zum Beispiel kann man sehr einfach das Wort `a mouse eats a dog` ableiten.

Achtung übrigens bei den Begrifflichkeiten: Im vorangegangenen Beispiel sind `SUBJEKT` und `dog` jeweils *ein* Zeichen und `a cat eats a mouse` ist *ein* Wort dieser Sprache! #



Die Ableitung von a dog eats the mouse.



Die Ableitung von the cat likes.

Abbildung 12

Bemerkung 1.6 Manchmal werden bei einer Grammatik nur die Produktionen angegeben. In diesem Fall ist immer S das Startsymbol, alle Großbuchstaben sind Nicht-Terminalsymbole und alle Kleinbuchstaben sind die Terminalsymbole. #

Beispiel 1.7 Die Grammatik

$S \to aSb \mid \epsilon$

erzeugt eine Sprache L, die aus allen Wörtern besteht, die gleich viele a wie b haben und bei denen erst alle a kommen und anschließend alle b.

Beispiel für eine Ableitung: $S \to aSb \to aaSbb \to aaaSbbb \to aaabbb$ #

1.2 Reguläre und kontextfreie Sprachen

Mit den Grammatiken ist es ein bisschen so wie mit den Funktionen in Mathematik: Wenn man versucht, alle Funktionen gemeinsam zu untersuchen, wird man schnell frustriert sein, weil es einfach viel zu viele Arten von Funktionen gibt. Daher teilt man die Funktionen in Klassen ein: Lineare Funktionen, quadratische Funktionen, Exponentialfunktionen usw. Das Gleiche kann man mit Grammatiken machen, indem man einschränkt, welche Produktionen verwendet werden dürfen:

Definition 1.8 *Eine Grammmatik G heißt*

(a) **regulär**, *wenn alle Produktionen eine der drei folgenden Formen haben:*

- $N \to tM$
- $N \to t$
- $N \to \epsilon$

wobei N und M Nicht-Terminalsymbole sein müssen und t ein Terminalsymbol.
Die Wörter regulärer Grammatiken wachsen also von links nach rechts.

(b) **kontextfrei**, *wenn auf den linken Seiten aller Produktionen genau ein einzelnes Nicht-Terminalsymbol steht.*

Beispiel 1.9

(a) Die Grammatik

$S \to aSb \mid \epsilon$

aus dem vorangegangenen Beispiel ist kontextfrei (auf der linken Seite steht immer nur genau 1 Nicht-Terminalsymbol, nämlich S), aber nicht regulär.

(b) Die Grammatik

$S \to 1A \mid 2A \mid 3A \mid 4A \mid 5A \mid 6A \mid 7A \mid 8A \mid 9A \mid 0$
$A \to 0A \mid 1A \mid 2A \mid 3A \mid 4A \mid 5A \mid 6A \mid 7A \mid 8A \mid 9A \mid \epsilon$

erzeugt alle Wörter der Sprache N der positiven ganzen Zahlen. Man beachte, dass 00 oder 015 keine syntaktisch korrekten ganzen Zahlen sind. Die Grammatik ist regulär, da alle vorkommenden Regeln einer der obigen Formen entsprechen.

(c) Die Grammatik

$S \to BR \mid B \mid 0$
$Z \to 0 \mid B$
$B \to 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
$R \to ZR \mid Z$

erzeugt dieselbe Sprache wie die vorangegangene Grammatik. Sie ist jedoch nicht regulär, weil die Ersetzung S→BA nicht regulär ist.

#

Definition 1.10 *Eine Sprache L heißt* **regulär** *bzw.* **kontextfrei**, *wenn es eine reguläre bzw. kontextfreie Grammatik gibt, die sie erzeugt.*

Beispiel 1.11 Die Sprache N der positiven ganzen Zahlen ist also regulär, weil wir in **Beispiel 1.9** (b) eine reguläre Grammatik gefunden haben, die N erzeugt. #

Bemerkung 1.12 Will man zeigen, dass eine Sprache *nicht* regulär ist, so reicht es nicht, eine nicht-reguläre Grammatik anzugeben, die die Sprache erzeugt! Im vorangegangenen Beispiel haben wir gesehen, dass es eine solche nicht-reguläre Grammatik für N gibt und trotzdem ist N regulär.

Um zu zeigen, dass eine Sprache nicht regulär [kontextfrei] ist, muss man beweisen, dass es nicht möglich ist, die Sprache über eine reguläre [kontextfreie] Grammatik zu erzeugen. Das ist im Allgemeinen sehr schwierig, wir werden aber später ein Kriterium kennenlernen, mit dem man in vielen Fällen sehr einfach zeigen kann, dass eine Sprache nicht regulär ist. #

1.3 Syntax-Diagramme

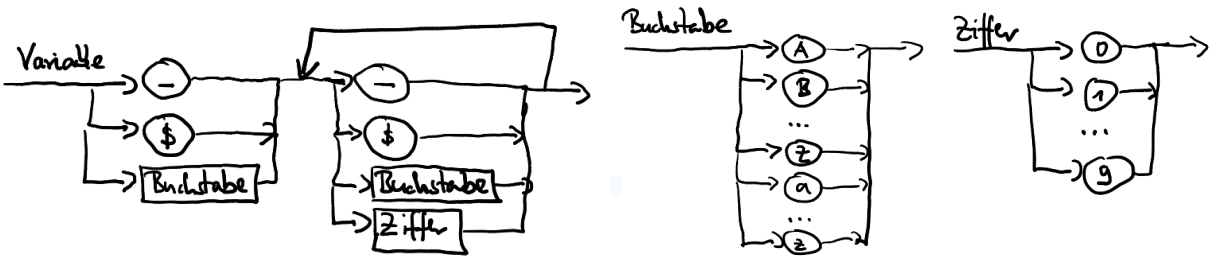
Die Grammatiken so gut wie aller Programmiersprachen sind (zumindest) kontextfrei. Anstelle der »normalen« Notation solcher Grammatiken, hat sich die Darstellung als »Syntax-Diagramm« eingebürgert:

Definition 1.13 *Ein* **Syntax-Diagramm** *ist eine grafische Darstellung einer kontextfreien Grammatik:*

Jedes Nicht-Terminalsymbol erhält ein eigenes Teil-Diagramm. Terminalsymbole werden in Kreisen dargestellt, Nicht-Terminalsymbole in Rechtecken. Dies erlaubt es, ganze Wörter anstelle von Einzel-Symbolen zu verwenden.

Die Ersetzungsrichtung wird durch Pfeile kenntlich gemacht. Jedes Teil-Diagramm benötigt einen Eingangspfeil und einen Ausgangspfeil.

Beispiel 1.14 Variablennamen in Java. Diese dürfen nur aus Buchstaben, Ziffern, Dollarzeichen und dem Unterstrich bestehen, wobei am Anfang keine Ziffer stehen darf:



#

Beispiel 1.15 Die Grammatik

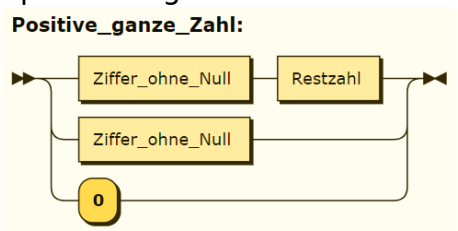
S → BR | B | 0

Z → 0 | B

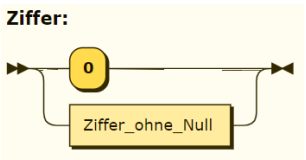
B → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

R → ZR | Z

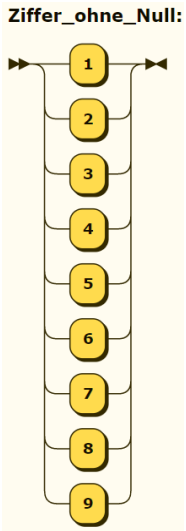
die die positiven ganzen Zahlen erzeugt, könnte als Syntax-Diagramm so aussehen:



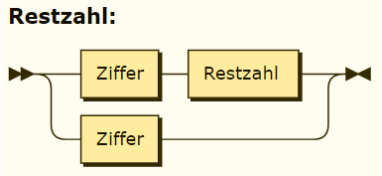
Entspricht S→BR | B | 0



Entspricht Z→0 | B



Entspricht B→1 | 2 | 3 | . . . | 9



Entspricht R→ZR | Z

#

2 Endliche Automaten

2.1 Akzeptoren

Definition 2.1 *Ein* **endlicher Automat** *(auch:* **Akzeptor***) besteht aus*

(a) einer endlichen Menge Z von **Zuständen***;*

(b) einer **Übergangsfunktion** *$f : Z \times \Omega \to Z$, wobei Ω ein Alphabet von Terminalsymbolen ist;*

(c) einem **Startzustand** *$S \in Z$;*

(d) einer Menge $E \subseteq Z$ von **Endzuständen** *(auch:* **akzeptierende Zustände***).*

Definition 2.2 *Ein endlicher Automat wird meistens über sein* **Zustandsdiagramm** *definiert (vgl.* **Abb. 13***).*

Dabei gilt:

(a) Jeder Zustand wird als Kreis dargestellt.

(b) Akzeptierende Endzustände werden doppelt umrandet.

(c) Der Startzustand wird durch einen Pfeil gekennzeichnet.

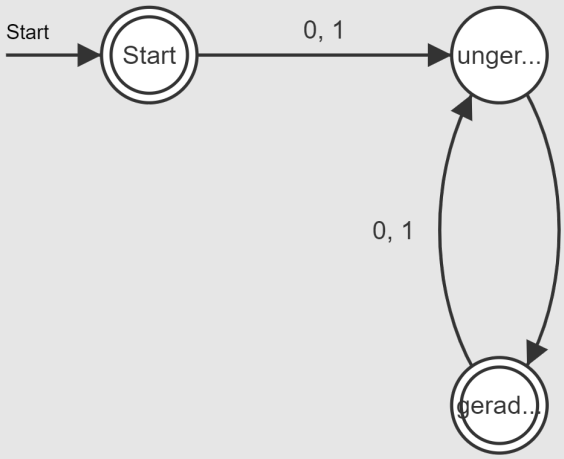
(d) Die Übergangsfunktion wird als Übergangspfeile zwischen den Zuständen dargestellt. Über den Pfeil schreibt man das Symbol, was zu diesem Übergang führt.

Als Symbole sind auch folgende Ausdrücke zulässig:[9]

- `A-Z`*: Alle Großbuchstaben von* `A` *bis* `Z`
- `2-9,a-z,B-G`*: Alle Ziffern von* `2` *bis* `9` und *alle Kleinbuchstaben* und *alle Großbuchstaben von* `B` *bis* `G`*.*
- `^ab3`*: Alle Zeichen* außer `a`*,* `b` *und* `3`*.*

Beispiel 2.3 Gesucht ist ein endlicher Automat, der alle Binärzahlen erkennt, die eine gerade Anzahl von Stellen haben. Das Zustandsdiagramm ist in **Abb. 13** (a) dargestellt. Daraus ergibt sich:

[9] Diese Ausdrücke werden **reguläre Ausdrücke** genannt. Mit ihnen beschäftigen wir uns später noch im Detail.



(a) Zustandsdiagramm

Zustand	Zeichen	Neuer Zustand
Start	0,1	ungerade
ungerade	0,1	gerade
gerade	0,1	ungerade

(b) Übergangsfunktion

Abbildung 13

(a) $Z = \{\text{Start}, \text{gerade}, \text{ungerade}\}$

(b) Die Übergangsfunktion ist als Tabelle in **Abb. 13** (b) dargestellt.

(c) $E = \{\text{Start}, \text{gerade}\}$

(d) Startzustand `Start` #

Information 2.4 Ein endlicher Automat läuft Zeichen für Zeichen von links nach rechts über eine Eingabe. Die Übergangsfunktion bestimmt in Abhängigkeit vom aktuellen Zustand und dem aktuell gelesenen Zeichen, ob der Zustand gewechselt wird und wenn ja, in welchen.

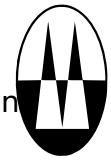
Sollte der Automat ein Zeichen lesen, für das es keinen Übergang gibt, so bricht er ab und das Wort wird nicht akzeptiert.

Definition 2.5

(a) Ein endlicher Automat **akzeptiert** *ein Wort, wenn er sich nach Abarbeiten des Wortes in einem Endzustand befindet.*

(b) Die Sprache L aller Wörter, die vom Automaten akzeptiert wird, heißt **die vom EA akzeptierte Sprache**.

Die von endlichen Automaten erkannten Sprachen sind genau die regulären Sprachen:



Satz 2.6

L ist regulär $\iff$ Es gibt eine reguläre Grammatik, die L erzeugt $\iff$ Es gibt einen EA, der L akzeptiert

Beweis: Der erste Teil ist nach Definition klar.

Der zweite Teil gilt, weil man jede der drei möglichen Ersetzungsregeln einer regulären Grammatik $1:1$ in einem endlichen Automaten umsetzen kann (siehe Unterricht). □

Endliche Automaten können vieles bewirken, aber sie haben ihre Grenzen:

Satz 2.7 Ein endlicher Automat kann nicht beliebig weit mitzählen.

Beweis: Da ein Automat sich nichts außer seinem Zustand »merken« kann, kann er höchstens so weit zählen, wie er Zustände hat. □

Das führt zu folgendem wichtigen Kriterium für reguläre Sprachen:

Satz 2.8 Eine Sprache, die beliebig tiefe Verschachtelungen erlaubt, kann nicht regulär sein.

Beweis: Wir machen uns das an Klammerausdrücken wie $(2-(3-5\cdot(6+2)-1)\cdot 4)\cdot 2$ klar:

Der EA muss mitzählen, wie viele Klammern geöffnet wurden (hier: 3) und anschließend prüfen, ob genauso viele Klammern wieder geschlossen werden.

Angenommen, es gäbe einen solchen EA. Sei n die Anzahl seiner Zustände. Dann kann dieser Automat höchstens n öffnende Klammern speichern. Ein Term mit $n+1$ öffnenden Klammern ist dann nicht mehr erkennbar. ↯ □

2.2 Schreibende Automaten (Mealy-Automaten)

Definition 2.9 *Ein* **Mealy-Automat** *ist ein endlicher Automat, der um die Fähigkeit erweitert wird, eine* **Ausgabe** *zu generieren.*

Dies wird in der Form a / A *bei den Übergängen angegegeben, d.h., in diesem Fall würde ein kleines* a *gelesen werden und ein großes* A *in die Ausgabe geschrieben werden.*

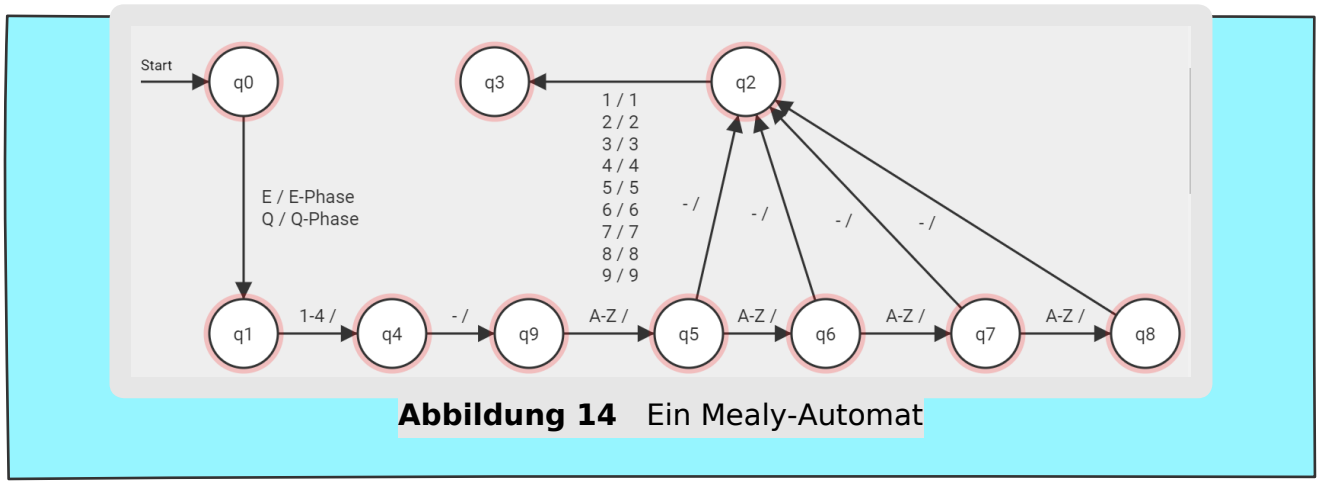


Abbildung 14 Ein Mealy-Automat

Beispiel 2.10 Der Automat in **Abb. 14** erhält als Eingabe eine Oberstufen-Kursbezeichnung und generiert als Ausgabe die Jahrgangsstufe und die Kursnummer.

Die Eingabe `Q3-INFO-G4` würde als Ausgabe `Q-Phase4` liefern. #

2.3 Beschreibung realer Automaten

Endliche Automaten lassen sich auch verwenden, um das Verhalten »echter« Automaten (Bank-, Kaffee-, Getränkeautomaten etc.) zu beschreiben. Dabei geht es nicht mehr darum, ein Eingabewort zu akzeptieren oder nicht, sondern die Eingabe wird als Folge von Interaktionen mit dem Automaten verstanden und der Automat spiegelt das Verhalten wider.

Beispiel 2.11 **Abb. 15** zeigt das Zustandsdiagramm eines Mobile-Games. #

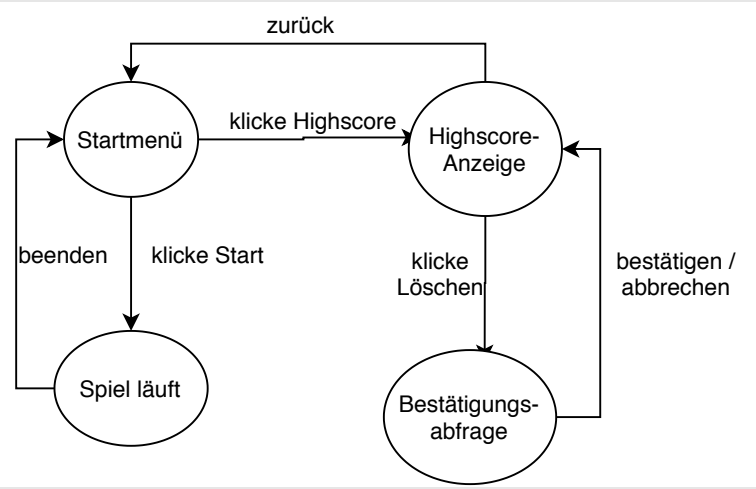
2.4 Reguläre Ausdrücke

Obwohl endliche Automaten zur *theoretischen* Informatik gehören, haben sie sehr viele Anwendungen in der Praxis. Um den Umgang mit Automaten einfacher zu machen, wurden die sogenannten »regulären Ausdrücke« (englisch: »regular expressions«) erfunden:

Definition 2.12 *Ein regulärer Ausdruck (engl.* regular expression*) ist eine Zeichenkette, die alle Wörter einer regulären Sprache beschreibt.*

Neben »normalen« Zeichen (wie a*,* Z*,* _*,* 0*,* 7*,* !*, ...) kann ein regulärer Ausdruck eine Reihe von Sonderzeichen enthalten, von denen eine Auswahl in* **Abb. 16** *dargestellt ist.*

Bemerkung 2.13



Menüführung in einem Mobile-Game

Abbildung 15

(a) Sie können Ihre regulären Ausdrücke auf diversen Webseiten testen, zum Beispiel unter `regex101.com/` .

(b) Wenn man mit regulären Ausdrücken in Java programmiert, muss man beachten, dass der Backslash `\` in Java eine Sonderbedeutung hat (z.B. steht `\n` für einen Zeilenumbruch). Daher muss man in Java für jeden Backslash einen doppelten Backslash schreiben, also z.B. `"Sie fahren ein (\\w+)\\."`.

#

	Bedeutung	**Beispiel**
`.`	beliebiges Zeichen	`ab.d` passt auf `abcd` oder auch `abdd`
`|`	oder	`Apfel|Banane` passt auf `Dies ist ein Apfel.` oder `Eine Banane schmeckt lecker`
`()`	Fasst zu einer Gruppe zusammen, die herausgelesen wird	`E(1|2)-INFO-G(1|2|3|4)`
`[abc]`	eines der aufgeführten Zeichen	`[A-Z]`, `[a-zC-E]`, `[A-Za-z0-9]`
`[^abc]`	ein Zeichen, das *nicht* aufgeführt ist	`[^A-Z]`, `[^0-9a-z]`
`\w`	Wort-Character	`\w` ist eine Abkürzung für `[A-Za-z0-9_]`
`\W`	Das Gegenteil von `\w`	
`\d`	Ziffer (engl. *digit*)	`\d` ist eine Abkürzung für `[0-9]`
`\D`	Das Gegenteil von `\d`	
`\s`	White-Space-Character	passt auf alle Whitespace-Zeichen, also Leerzeichen, Tab, Zeilenumbruch.
`\S`	Das Gegenteil von `\s`	
`{N}`	genau `N` mal	`(abc){2}` passt auf `abcabc`
`{N,M}`	`N` bis `M` mal	`(ab){2,4}` passt auf `abab`, `ababab` und auf `abababab`
`*`	beliebig oft	`(\d)*€` passt auf `Sie müssen 35€ bezahlen.` und auf `Der Kontostand beträgt 8493€`
`+`	mindestens 1 mal	Wie `*`, das Zeichen muss aber mindestens einmal vorkommen.
`^`	Beginn der Zeichenkette	`^Auto` passt auf `Autos fahren schnell`, nicht aber auf `Schnell-fahrende Autos`
`$`	Ende der Zeichenkette	`Auto$` passt auf `Ich fahre gerne Auto`, nicht aber auf `Ich fahre gerne Auto.`

Abbildung 16

3 Berechenbarkeit

Schon bevor die ersten elektronischen Rechner gebaut wurden, haben sich Mathematiker Gedanken über die prinzipiellen Möglichkeiten und Grenzen von Rechenmaschinen gemacht. Der Durchbruch gelang schließlich dem britischen Logiker *Alan Turing* (1912-1954), der den Begriff der »Berechenbarkeit« definierte und untersuchte.

3.1 Computer, Probleme, Algorithmen und Berechenbarkeit

Wir starten zunächst mit vier unpräzisen, anschaulichen Definitionen:

Definition 3.1 *Ein* **Problem** *ist eine Formulierung der Form »Gegeben-Gesucht«.*

Ein **Algorithmus** *ist eine endliche Folge von Handlungsanweisungen, von denen jeden einzelne genau definiert und »durchführbar« ist.*

Ein Problem heißt **berechenbar***, wenn es einen Algorithmus gibt, der es löst.*

Ein **Computer** *kann beliebige Algorithmen ausführen.*

Das Problem an diesen Definitionen ist, dass sie nicht präzise genug sind, um exakte Schlussfolgerungen aus ihnen zu ziehen. Was heißt »durchführbar«? Was ist eine »Handlungsanweisung« genau? Was ist als »Gegeben-Gesucht« erlaubt? Dadurch könnte es passieren, dass ein Problem unter gewissen Umständen berechenbar ist und unter anderen nicht.

Beispiel 3.2 Eine Programmiersprache, die nur `for`-Schleifen kennt, kann bestimmte Probleme nicht berechnen, die von Programmiersprachen mit `while`-Schleifen berechnet werden können. #

Darum ist das Ziel der folgenden Abschnitte, diese vier Kernbegriffe der Informatik exakt definieren zu können.

3.2 Turingmaschinen

Definition 3.3 *Eine* **Turingmaschine** *ist ein endlicher Automat, der auf folgende Art »gepimpt« wird:*

(a) Die Eingabe der TM steht auf einem Band, das in beide Richtungen unbeschränkt ist.

(b) Die TM hat einen beweglichen Lese-Schreib-Kopf, der nach links und rechts bewegt werden kann.

(c) Die TM kann das Zeichen an der Position des LS-Kopfes lesen und verändern.

Genauer gesagt ist eine TM gegeben durch

(a) eine endliche Menge Z von **Zuständen**;

(b) eine **Übergangsfunktion** $f : Z \times \Omega \to Z \times \Omega \times \{l, r\}$, *wobei Ω ein Alphabet von Terminalsymbolen ist und $\{l, r\}$ für »Bewegung nach links bzw. rechts« steht;*

(c) einem **Startzustand** $S \in Z$;

(d) einem **Haltezustand** $\mathit{HALT} \in Z$

Bemerkung 3.4 Eine TM beginnt also in einem gewissen Zustand S und liest dann Zeichen für Zeichen auf dem Band und entscheidet bei jedem Zeichen, ob sie sich nach links oder nach rechts bewegt oder stehen bleibt und welches Zeichen sie auf das Band schreibt.

Die Maschine bleibt stehen, wenn der Zustand `HALT` erreicht wird. #

Bemerkung 3.5 Eine TM kann also als Übergangstabelle dargestellt werden, z. B.

Zustand	gelesenes Zeichen	geschriebenes Zeichen	Bewegung	neuer Zustand
S	a	_	r	A
S	b	_	r	B
A	*	*	r	A
A	_	_	l	C
B	*	*	r	B
B	_	_	l	D
C	a	j	r	halt
C	b	n	l	halt
D	a	n	r	halt
D	b	j	l	halt

Diese TM erhält als Eingabe Wörter aus den Symbolen `a` und `b`. Das `*` in den Zeilen 3 und 5 steht für ein beliebiges Zeichen, wenn kein anderer Übergang definiert ist. Wenn die TM in Zustand `A` ein Leerzeichen liest, wechselt sie in Zustand `C` und schreibt das Leerzeichen wieder hin. Wenn sie irgendein anderes Zeichen liest, bleibt sie in Zustand `A` und löscht das Zeichen. #

Definition 3.6 *Eine TM kann also aus einer Eingabe eine Ausgabe berechnen. Ein Problem, das sich mit einer Turingmaschine lösen lässt, heißt* **turing-berechenbar***.*

Beispiel 3.7 Das Problem

Geg:	Wort aus `a` und `b`
Ges:	Sind der erste und der letzte Buchstabe des Wortes gleich?

ist turing-berechenbar, weil die Turingmaschine aus dem Beispiel genau dieses Problem löst (sie schreibt `j` für »Ja« bzw. `n` für »Nein« auf das Band). #

3.3 Die Hypothese von Church

Wie wir gesehen haben, können wir erstaunlich viele Probleme mit Hilfe von Turingmaschinen lösen. Manchmal dauert es zwar extrem lange, bis die Maschine ein Ergebnis liefert, *prinzipiell* kann sie das Ergebnis aber berechnen.

Die Informatiker*innen der Frühzeit waren bestimmt erstaunt als sie feststellten, dass so ziemlich jedes Problem, auf das sie stießen, von einer Turingmaschine gelöst werden

kann. Theoretisch ist es möglich, ein AAA-Spiel als Turingmaschine zu programmieren. Dies führte zur sogenannten »Hypothese von Church«:[10]

Satz 3.8 Die **Hypothese von Church** besagt:

Alles, was sich (algorithmisch) berechnen lässt, kann von Turingmaschinen berechnet werden.

Dieser »Satz« kann nicht bewiesen werden, weil wir nicht definiert haben, was »algorithmisch berechenbar« exakt bedeuten soll. Anders herum verwendet man die Hypothese, um genau dies zu tun:

Definition 3.9 *Ein Problem heißt* **berechenbar***, wenn es turing-berechenbar ist, d.h., wenn es eine Turingmaschine gibt, die das Problem löst.*

Das führt automatisch zu folgender Definition:

Definition 3.10 *Ein* **Algorithmus** *ist eine Turingmaschine.*

Damit sind zwei der vier Kernbegriffe »Algorithmus«, »berechenbar«, »Computer« und »Problem« mathematisch exakt definiert. Im nächsten Abschnitt widmen wir uns zunächst dem Begriff des »Computers«, im übernächsten dann der Definition des »Problems«.

3.4 Die universelle Turingmaschine

Eine Turingmaschine erhält als Eingabe einen String und liefert einen String als Ausgabe. Wie wir wissen, kann man Turingmaschinen selbst auch als String aufschreiben (das haben wir in diversen Aufgaben gemacht).

Also muss es möglich sein, eine »universelle Turingmaschine« zu konstruieren:

Definition 3.11 *Eine* **universelle Turingmaschine** U *erhält als Eingabe eine (andere) Turingmaschine* T *(als String kodiert) und deren Eingabe* E *und liefert als Ergebnis das Ergebnis, das* T *liefern würde, wenn* T *auf* e *angewendet wird:*

$$U(T \$ e) = T(e)$$

dabei ist $\$$ *ein Zeichen, das im Alphabet für die Turingmaschine und die Eingabe nicht vorkommt.*

[10] Auch »Church-Turing-These« genannt nach Alonzo Church und Alan Turing.

Bemerkung 3.12 Anders formuliert: Eine universelle Turingmaschine führt beliebige Turingmaschinen aus. #

Wie oben angedeutet, kann man beweisen, dass es solche universellen Turingmaschinen gibt. Bereits im Jahr 1936 gelang es Alan Turing, eine solche universelle Turingmaschine zu konstruieren.

Damit erhalten wir folgende

Definition 3.13 *Ein* **Computer** *ist eine universelle Turingmaschine.*

3.5 Probleme als partielle Funktionen

Wir erinnern uns: Ein kann Problem anschaulich in der Form »Gegeben-Gesucht« formuliert werden. Wir haben also eine Eingabe (das was gegeben ist) und wollen die Lösung berechnen. Diese Lösung hängt natürlich von der Eingabe ab, d.h., zu verschiedenen Eingaben wird man wahrscheinlich verschiedene Lösungen finden.

Information 3.14 Ein Problem P: (Gegeben: e, Gesucht: $p(e)$) kann also in der Form

»Berechne den Wert der Funktion p für eine gewisse Eingabe e«

formuliert werden.

Beispiel 3.15 Das Problem

Geg:	Bild-Datei
Ges:	Ist auf dem Bild eine Katze zu sehen?

kann also aufgefasst werden als

»Berechne $k(b)$«

wobei b eine Bilddatei ist und k eine Funktion, die jeder Bilddatei »Ja« oder »Nein« zuordnet. #

Nun müssen wir uns noch klarmachen, wie die Eingabe und die Ausgabe dieser Funktionen aussehen kann: Jede Ein- und Ausgabe ist eine Zeichenkette und jede Zeichenkette kann (z. B. mittels ASCII-Code oder Unicode) als natürliche Zahl dargestellt werden. Das bedeutet, dass die Funktionen, um die es hier geht, natürliche Zahlen auf natürliche Zahlen abbilden.

Beispiel 3.16 Die Bilddatei liegt in irgendeinem Grafik-Format vor, z.B. pixelweise als RGB-Werte. Die Pixel Rot-Gelb-Gelb-Grün könnten dann in der Form

$$\underbrace{255000000}_{\text{rot}}\,\underbrace{255255000}_{\text{gelb}}\,\underbrace{255255000}_{\text{gelb}}\,\underbrace{000255000}_{\text{grün}}$$

als (enorm große!) natürliche Zahl dargestellt werden.

Die Ausgabe der Funktion ist einfacher, denn hier könnte 0 für »Nein« und 1 für »Ja« stehen. #

In dem vorangegangenen Beispiel sieht man, dass die Problem-Funktion nicht für alle natürlichen Zahlen definiert sein muss. In diesem Fall müssen nämlich jeweils drei Ziffern zwischen 000 und 255 liegen und es müssen immer drei Dreier-Päckchen sein für jeden Pixel.

Insgesamt kommt man zu folgender Definition

Definition 3.17 *Eine Funktion f heißt* **partiell**, *wenn sie natürlichen Zahlen natürliche Zahlen zuordnet. Dabei kann die Funktion für bestimmte Eingaben auch undefiniert sein.*

Dies erlaubt schließlich die exakte Definition des Begriffs »Problem«:

Definition 3.18 *Ein* **Problem** *ist eine partielle Funktion.*

Beispiel 3.19 Das Problem aus dem vorangegangenen Beispiel ist also die Funktion $k(n) = m$, wobei n eine natürliche Zahl ist, die eine Bilddatei kodiert und m eine Zahl aus $\{0; 1\}$. #

Mit diesen Begriffen können wir nun beweisen, dass ein Problem berechenbar ist.

Beispiel 3.20 Gegeben ist das Problem ADD:

Geg:	Zwei natürliche Zahlen a und b
Ges:	Die Summe $a + b$

Um zu zeigen, dass ADD berechenbar ist, müssen wir eine Turingmaschine konstruieren, die es lösen kann. Dazu kodieren wir die Eingabe in der Form

$$\underbrace{111...1}_{a \text{ viele}} + \underbrace{111...1}_{b \text{ viele}}$$

Damit muss unsere Turingmaschine beispielsweise folgende Aufgaben lösen:

- $111 + 11 \rightarrow 11111$

- $+111 \rightarrow 111$
- $+ \rightarrow$

Eine solche Turingmaschine anzugeben ist nicht schwierig (Übung!). Damit ist bewiesen, dass `ADD` berechenbar ist. #

3.6 Nicht-berechenbare Probleme

Zu Beginn der Computerzeit herrschte große Euphorie, weil die Wissenschaftler annahmen, dass die neuen Maschinen ihnen das Denken abnehmen könnten. Mathematiker hofften beispielsweise auf eine Maschine, die man mit Aussagen füttert und die einen Beweis oder einen Gegenbeweis erbringt. Kurz: Man dachte, dass eine goldene Zeit anbricht, in der die Mathematik innerhalb kürzester Zeit enorme Fortschritte machen würde.

Umso größer war die Ernüchterung, als Alan Turing diese Hoffnung zunichte machte, indem er bewies, dass es Probleme gibt, die nicht berechenbar sind — und das sogar bevor die ersten Computer gebaut wurden![11]

Um den Gedankengang Turings verstehen zu können, muss man sich mit der Unendlichkeit vertraut machen:

Definition 3.21 *Eine Menge heißt* **abzählbar unendlich**, *wenn sie unendlich viele Elemente hat und es ein System gibt, nach dem man ihre Elemente so nacheinander aufzählen kann, dass man keines vergisst.*

Beispiel 3.22 Die natürlichen Zahlen $\mathbb{N}$ sind abzählbar unendlich, denn

$$\mathbb{N} = \{0; 1; 2; 3; 4; ...\}$$

Nach diesem System wird man garantiert jede natürliche Zahl irgendwann erwischen.

Die ganzen Zahlen $\mathbb{Z}$ und sogar die rationalen Zahlen $\mathbb{Q}$ sind auch abzählbar (Übung!). #

Das Interessante ist nun, dass es Mengen gibt, die »zu groß sind, als dass man sie abzählen könnte«, d.h., egal, wie man abzählt, man vergisst immer mindestens eine:

Definition 3.23 *Eine Menge heißt* **überabzählbar unendlich**, *wenn sie unendlich viele Elemente hat und nicht abgezählt werden kann.*

[11] Der erste voll-digitale Rechner war die ENIAC von 1946. Alan Turings Dissertation »On computable numbers with an application to the Entscheidungsproblem« stammt aus dem Jahr 1936. Link: `https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf`

Beispiel 3.24 Die reellen Zahlen $\mathbb{R}$ sind überabzählbar (Übung!, siehe AB 6: Berechenbarkeit, Aufgabe 2). #

Bemerkung 3.25 Wenn eine Menge überabzählbar ist, dann bleibt bei jeder Abzählung mindestens ein Element übrig. Das bedeutet aber, dass tatsächlich unendlich viele Elemente übrig bleiben:

Wenn es beispielsweise nur 2 Elemente sind, die man mit der Abzählung nicht erwischt, könnte man einfach eine neue Abzählung machen, die zunächst diese beiden Elemente aufführt und danach mit der vorherigen Aufzählung weiter macht. Überabzählbare Mengen sind also *sehr viel größer* als abzählbare Mengen. #

Turings Argument war nun das folgende: Er konnte mathematisch beweisen, dass es überabzählbar unendlich viele Probleme, aber nur abzählbar unendlich viele Algorithmen (Turingmaschinen) gibt. Also muss es Probleme geben, die nicht berechenbar sind!

Wir schauen uns diese beiden Beweise an:

Satz 3.26 Die Menge der Turingmaschinen ist abzählbar unendlich.

Beweis: Turingmaschinen können in Tabellenform bzw. als einfache Strings kodiert werden. Jeder solche String kann als natürliche Zahl dargestellt werden (z. B. über den ASCII-Code).

D.h.: Jede TM entspricht einer natürlichen Zahl.

Also kann es nicht mehr Turingmaschinen als natürliche Zahlen geben und diese sind abzählbar unendlich. □

Der zweite Satz ist sehr viel schwieriger zu beweisen:

Satz 3.27 Die Menge aller Probleme ist überabzählbar unendlich.

Beweis: Weil ein Problem einer partiellen Funktion entspricht, geht es hier eigentlich darum zu zeigen, dass man die Menge der partiellen Funktionen nicht abzählen kann.

Angenommen, die Menge aller partiellen Funktionen wäre abzählbar unendlich. Dann wäre es möglich, alle partiellen Funktionen aufzuzählen, ohne eine dabei zu vergessen:

$$f_1, f_2, f_3, f_4, f_5, f_6, \ldots$$

Wir konstruieren nun eine partielle Funktion F, von der wir anschließend zeigen, dass sie in der obigen Abzählung nicht vorkommen kann: Definiere

$$F(n) = \begin{cases} 1 & \text{, wenn } f_n(n) = 0 \\ 0 & \text{, wenn } f_n(n) \neq 0 \end{cases}$$

Da in der obigen Abzählung alle partiellen Funktionen vorkommen, muss irgendwann auch F vorkommen, d.h. es gibt ein $k \in \mathbb{N}$ mit $F = f_k$. Nun berechnen wir den Wert von F an der Stelle k:

Da F nur die Werte 0 und 1 annimmt, müssen wir zwei mögliche Fälle unterscheiden:

1. Fall: $F(k) = 0$ Dann folgt nach Definition von F, dass $f_k(k) \neq 0$. Wegen $F = f_k$ kann dies aber nicht sein! (F kann nicht gleichzeitig 0 und 1 sein!)

2. Fall: $F(k) = 1$ Dann folgt nach Definition von F, dass $f_k(k) = 0$. Dies ist ebenso ein Widerspruch.

Insgesamt erhalten wir in jedem Fall einen Widerspruch, d.h., die Annahme, dass die Menge der Probleme abzählbar ist, muss falsch gewesen sein. □

Insgesamt haben wir folgende Einsicht erlangt:

Satz 3.28 Es gibt (viel) mehr Probleme als Algorithmen, d.h. es gibt (viele) Probleme, die nicht berechenbar sind.

3.7 Das Halteproblem

Wir haben gesehen, dass es Probleme gibt, die nicht berechenbar sind. Das ist erst einmal unschön, aber könnte es nicht sein, dass diese Probleme überhaupt gar nicht von Belang sind? Vielleicht sind ja alle Probleme berechenbar, die in der Praxis vorkommen. Leider ist auch das nicht der Fall.

In diesem Abschnitt lernen wir mit dem Halteproblem ein konkretes Problem kennen, das nicht berechenbar ist.

Definition 3.29 *Das Halteproblem ist gegeben durch*

Geg: *Algorithmus T*

Ges: *Antwort auf die Frage: »Wird T anhalten oder in eine Endlosschleife geraten?«*

Satz 3.30 Das Halteproblem ist nicht berechenbar.

Beweis: Angenommen, es ist doch berechenbar. Dann gibt es eine Turingmaschine H, die das Halteproblem löst, d.h. H sagt für jede Turingmaschine T voraus, ob diese anhalten wird oder nicht.

Wenn es eine solche TM gibt, können wir damit eine neue Turingmaschine M konstruieren (vgl. **Abb. 17** rechts).

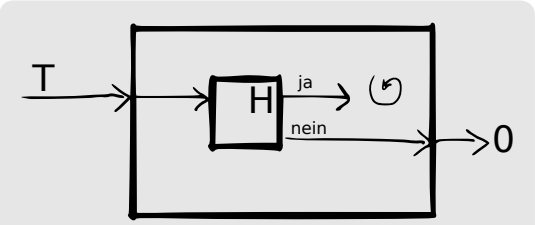


Abbildung 17

M hält also an (und liefert als Ergebnis eine 0, das ist aber nicht von Belang), wenn T nicht anhält und gerät in eine Endlosschleife, wenn T anhält.

Wir wenden nun M auf sich selbst an!

Es gibt nun zwei Möglichkeiten:

M hält an: Wenn M anhält, muss $H(M) =$ nein sein, d.h. M hält nicht an!

M hält nicht an: Wenn M nicht anhält, muss $H(M) =$ ja sein, d.h. M hält an!

In beiden Fällen ergibt sich ein Widerspruch. Damit ist bewiesen, dass die Annahme falsch gewesen sein muss. Also ist das Halteproblem nicht berechenbar. □

4 Grundlagen der Komplexitätstheorie

Im vorangegangenen Kapitel haben wir uns mit dem Begriff der »Berechenbarkeit« auseinander gesetzt, also mit der Frage, welche Probleme von Computer prinzipiell gelöst bzw. nicht gelöst werden können.

In diesem abschließenden Kapitel lernen wir die Grundlagen der Komplexitätstheorie kennen: Hier befassen wir uns ausschließlich mit berechenbaren Problemen, d.h. mit Problemen für die ein Lösungsalgorithmus existiert. Die Frage, die sich für diese Probleme stellt, ist, ob das Problem auch praktisch lösbar ist oder ob der Algorithmus einfach zu lange brauchen würde, um das Problem zu lösen, womit es »praktisch nicht berechenbar« wäre.

4.1 Laufzeit und O-Notation

Zunächst einmal befassen wir uns mit der Laufzeit von Algorithmen. Dazu das folgende Beispiel:

Beispiel 4.1 Es soll entschieden werden, ob ein String einen gewissen Teil-String enthält. Es soll also die Methode `boolean enthaelt(String ganzes, String teil)` implementiert werden.

Drei Informatiker*innen entwickeln jeweils einen Algorithmus dazu (vgl. **Abb. 18**).

Jeder der drei Algorithmen enthält (mindestens) eine `for`-Schleife, deren Durchläufe (mehr oder weniger) der Länge `g` des Strings `ganzes` abhängen. Daher wird jeder der drei Algorithmen umso längere Zeit benötigen, je größer `g` ist. #

Definition 4.2 *Unter der* **Laufzeit** *eines Algorithmus versteht man die Zeit, die der Algorithmus zur Lösung des Problems benötigt.*

Man unterscheidet dabei drei verschiedene Szenarien:

(a) **worst-case***: Laufzeit im denkbar ungünstigsten Fall.*

(b) **best case***: Laufzeit im denkbar günstigsten Fall.*

(c) **average case***: Durchschnittliche Laufzeit.*

Obwohl die durchschnittliche Laufzeit die interessanteste Größe ist, werden wir uns hier nur um best-case- und worst-case-Laufzeiten kümmern.

```java
boolean enthaelt(String ganzes, String teil){
  int n=ganzes.length();
  int t=teil.length();
  for( int i=0; i<n; i++ ){
    String s=ganzes.substring( i, i+t );
    if(s==teil){
      return true;
    }
  }
  return false;
}
```

Algorithmus A

```java
boolean enthaelt2(String ganzes, String teil){
  int n=ganzes.length();
  for( int i=0; i<n; i++ ){
    for(int l=0; l<n; l++){
      String s=ganzes.substring( i, i+l );
      if(s==teil){
      return true;
    }
    }
  }
  return false;
}
```

Algorithmus B

```java
boolean enthaelt3(String ganzes, String teil){
  if(ganzes==teil){
    return true;
  }else{
    int n=ganzes.length();
    int t=teil.length();
    if(n<t){
      return false;
    }
    for(int i=1; i<n; i++){
      String s=ganzes.substring( i );
      boolean ent=enthaelt3(s, teil);
      if(ent){
        return true;
      }
    }
    return false;
  }
}
```

Algorithmus C

Es soll festgestellt werden, ob ein String `ganzes` einen Teilstring `teil` enthält. Diese Abbildung zeigt drei verschiedene Lösungsalgorithmen, die alle drei das korrekte Ergebnis liefern, sich dabei aber in ihrer Laufzeit massiv unterscheiden.

Abbildung 18

Das Problem ist nun, dass die benötigte Zeit stark von der verwendeten Maschine sowie dem Betriebssystem und anderen Programmen, die parallel laufen, abhängt. Auch die verwendete Programmiersprache macht einen erheblichen Unterschied. Darum ist es nicht sinnvoll, die Laufzeit in den üblichen Zeiteinheiten (Minuten, Sekunden, Millisekunden, ...) zu messen:

Information 4.3 Die Laufzeit eines Algorithmus entspricht der Anzahl der Anweisungen, die der Algorithmus durchführen muss. Meist ist dies eine Funktion in Abhängigkeit von der Länge n eines Strings oder eines Arrays.

Bei dieser Funktion kommt es nicht auf konstante Faktoren an, d.h. $f(n) = 3n + 2$ Operationen sind »genauso viele« wie $g(n) = 100n + 1000$, obwohl letztere um ein Vielfaches größer ist. Der Hintergrund hierfür ist, dass man jeden Algorithmus innerhalb seiner Funktionenklasse beliebig beschleunigen kann, indem man mehr Hardware zur Verfügung stellt. Außerdem fallen die Unterschiede immer weniger ins Gewicht, wenn n größer wird. Daher verwendet man die folgend Schreibweise:

Information 4.4 Man sagt, eine Funktion $f(n)$ **liegt in** $O(g(n))$, wenn

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \text{const.} \neq 0$$

Anschaulich bedeutet das, dass $f(n)$ und $g(n)$ »gleich schnell« wachsen.

Beispiel 4.5

(a) $f(n) = 3n + 10$ liegt in $O(n)$:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{3n+10}{n} = \lim_{n\to\infty} 3 + \frac{10}{n} = 3$$

(b) $f(n) = 100n^2$ liegt nicht in $O(2^n)$:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{100n^2}{2^n} = 0$$

da eine Exponentialfunktion schneller wächst als jedes Polynom.

(c) $f(n) = \frac{n(n+1)}{2}$ liegt in $O(n^2)$:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n(n+1)}{2n^2} = \lim_{n\to\infty} \frac{n^2+n}{2n^2} = \lim_{n\to\infty} 0{,}5 + \frac{1}{2n} = 0{,}5$$

#

Beispiel 4.6 Damit können wir nun die worst-case-Laufzeit der drei Algorithmen A, B und C aus dem Eingangsbeispiel bestimmen:

Es sei n die Länge des `ganzes`-String.

(a) Algorithmus A hat im worst-case

$$f(n) = 2 + n \cdot 2 = 2n + 2$$

Anweisungen zu erledigen.Die Laufzeit liegt also in $O(n)$.

(b) Algorithmus B muss im worst-case

$$f(n) = n + n + n + n + ... + n = n \cdot n = n^2$$

Anweisungen erledigen. Die Laufzeit liegt also in $O(n^2)$.

(c) Der rekursive Algorithmus C muss im worst-case folgendermaßen vorgehen:

Der Algorithmus selbst wird g mal auf ein neues `ganzes` angewendet, das in jedem Durchgang um 1 kleiner wird. Das sind also etwa

$$f(n) = n \cdot (n-1) \cdot (n-2) \cdot ... \cdot 2 \cdot 1 = n!$$

viele Anweisungen. Mit dem mathematischen Trick

$$f(n) = n! = n \cdot (n-1) \cdot (n-2) \cdot ... \cdot 2 \cdot 1 \approx \frac{n}{2} \cdot \frac{n}{2} \cdot \frac{n}{2} \cdot ... \cdot \frac{n}{2} = \left(\frac{n}{2}\right)^n$$

sehen wir, dass die Laufzeit in $O\left(\left(\frac{n}{2}\right)^n\right)$ liegt. #

Definition 4.7 *Ein Algorithmus mit Problemgröße n hat eine...*

- **konstante** *Laufzeit, wenn die Laufzeit in $O(1)$ liegt.*
- **lineare** *Laufzeit, wenn die Laufzeit in $O(n)$ liegt.*
- **quadratische** *Laufzeit, wenn die Laufzeit in $O(n^2)$ liegt.*
- **polynomielle** *Laufzeit, wenn die Laufzeit in $O(n^k)$ liegt mit $k \in \mathbb{N}_0$.*
- **exponentielle** *Laufzeit, wenn die Laufzeit in $O(b^n)$ liegt mit $b > 1$.*
- **logarithmische** *Laufzeit, wenn die Laufzeit in $O(\log_b(n))$ liegt mit $b > 1$.*

Bei exponentieller oder logarithmischer Laufzeit ist die konkrete Basis irrelevant.

Beispiel 4.8 Algorithmus A hat also eine lineare Laufzeit, Algorithmus B eine quadratische Laufzeit und Algorithmus C eine super-exponentielle Laufzeit, d.h. die Laufzeit liegt sogar über der exponentiellen Laufzeit (latein. *super* = »über«). #

4.2 Klassifizierung von Problemen

Mit der Analyse der Laufzeit können wir nun die zu lösenden Probleme in Komplexitätsklassen einteilen:

Definition 4.9 *Wir unterscheiden zwei Klassen von Problemen:*

(a) Ein Problem gehört zur Klasse $\boldsymbol{P}$*, wenn es einen Lösungsalgorithmus mit polynomieller Laufzeit (d.h. in* $O(n^k)$ *mit* $k \in \mathbb{N}$*) gibt. Diese Probleme heißen* **effizient lösbar***.*

(b) Ein Problem gehört zur Klasse **EXP***, wenn es einen Lösungsalgorithmus mit exponentieller Laufzeit gibt.*

Bemerkung 4.10

(a) Jedes Problem, das in polynomieller Zeit lösbar ist, ist natürlich auch in exponentieller Zeit lösbar. Daher ist P eine Teilmenge von EXP.

(b) Probleme, die in EXP, nicht aber in P liegen, sind bereits für relativ kleine Problemgrößen n **praktisch nicht lösbar**. Nehmen wir an, wir haben einen Algorithmus A in $O(n^5)$ und einen Algorithmus B in $O(2^n)$. Wir erhalten dann in Abhängigkeit von n folgende Laufzeiten:

$\boldsymbol{n}$	1	2	3	4	5	...	100
$\boldsymbol{A}$ **(**$\boldsymbol{n^5}$**)**	1	32	243	1024	3125	...	10.000.000.000
$\boldsymbol{B}$ **(**$\boldsymbol{2^n}$**)**	2	4	8	16	32	...	$2^{100} \approx 10^{33}$

Algorithmus A benötigt also für kleine Problemgrößen deutlich mehr Schritte als Algorithmus B. Irgendwann überholt B aber A und wächst dann unvorstellbar schnell. Die 10 Milliarden Rechenschritte von A schafft ein heutiger Computer in etwa einer Sekunde. Für die 10^{33} Rechenschritte sieht das ganz anders aus:

$$10^{33} = 10^{23} \cdot 10.000.000.000$$

B benötigt also 10^{23} mal so lange wie A. Das sind dann

$$\begin{aligned} 10^{23}\,\text{s} &\approx 2{,}8 \cdot 10^{19}\,\text{h} \approx 1{,}16 \cdot 10^{18}\,\text{Tage} \\ &\approx 3{,}2 \cdot 10^{15}\,\text{Jahre} = 3.200.000.000.000.000\,\text{Jahre} \end{aligned}$$

#

4.3 P=NP?

Zuletzt werfen wir einen Blick auf das vielleicht wichtigste ungelöste Problem der theoretischen Informatik. Das »P=NP?«-Problem ist eines der sieben Milleniumprobleme, für deren Lösung das Clay Mathematics Institute in Cambridge jeweils ein Preisgeld von einer Million Dollar ausgelobt hat.[12]

Dazu schauen wir uns zwei Probleme an:

Beispiel 4.11

(a) (Das Faktorisierungsproblem.) Gegeben ist eine (sehr große) natürliche Zahl n. Gesucht ist ein echter Teiler T dieser Zahl, d.h. $1 < T < n$.

Dieses Problem ist extrem schwer zu lösen, wenn n richtig groß ist, weil man im Endeffekt alle möglichen Zahlen von 2 bis $\sqrt{n}$ ausprobieren muss. Probieren Sie es einmal für $n = 57$ und einmal für $n = 1324597513$!

Interessanterweise ist es aber extrem einfach, zu überprüfen, ob ein Lösungskandidat T korrekt ist: Teile einfach n durch T und prüfe ob ein Rest bleibt. Z.B. ist 2347 ein Teiler der zweiten Zahl: $1324597513 : 2347 = 564379$.

(b) (Das Problem des Handlungsreisenden) Ein Handelsvertreter möchte n verschiedene Städte anfahren und eine Route wählen, die höchstens K Kilometer lang ist. Auch hier hilft nur Ausprobieren und auch hier lässt sich für einen Lösungskandidaten leicht überprüfen, ob er eine Lösung ist.

#

Interessanterweise hat sich herausgestellt, dass es in der Informatik (nicht nur in der Theorie, auch in der Praxis!) haufenweise Probleme gibt, die sich ähnlich verhalten wie die beiden beschriebenen Probleme. Diese Kategorie von Problemen erhält wiederum einen Namen:

Definition 4.12 *Ein Problem liegt in der Klasse $\boldsymbol{NP}$ (»nicht-deterministisch-polynomiell«), wenn es einen Algorithmus gibt, der das Problem in polynomieller Zeit lösen kann, wenn er auf einer* **nicht-deterministischen** *Turingmaschine ausgeführt werden würde.*

[12] Zum heutigen Zeitpunkt ist erst eines dieser Probleme gelöst worden: Die Poincaré-Vermutung wurde im Jahr 2002 von Grigori Perelman bewiesen. Er schlug das Preisgeld aber aus.

Eine nicht-deterministische Turingmaschine kann für jedes gelesene Zeichen mehrere Übergänge haben, die alle parallel ausgeführt werden. Problem: Eine solche Maschine gibt es nicht![13]

Übersetzt: Ein NP-Problem kann effizient gelöst werden, wenn der Algorithmus die Lösung erraten könnte und nur noch überprüfen müsste, dass es sich wirklich um eine Lösung handelt.

Klar ist, dass jedes P-Problem automatisch auch ein NP-Problem ist. Außerdem kann man zeigen, dass jedes NP-Problem auf einer deterministischen Maschine in exponentieller Laufzeit lösbar ist (man muss alle Kombinationsmöglichkeiten ausprobieren).

Die große Fragestellung ist nun:

Gibt es ein Problem, das in NP liegt, aber nicht in P? Oder ist in Wirklichkeit $P = NP$?

Es geht also darum, ob die oben genannten NP-Probleme, die eine große praktische Relevanz haben, effizient lösbar sind oder nicht. Die Antwort auf diese Frage ist zum heutigen Zeitpunkt immer noch völlig offen!

[13] Oder vielleicht doch? Mit einem Quantencomputer wäre es vielleicht möglich, eine solche nicht-deterministische Maschine zu bauen. Erste Erfolge gab es in dieser Hinsicht bereits.