Programmieren 1

Einführung in die Programmierung

Silko Kruse

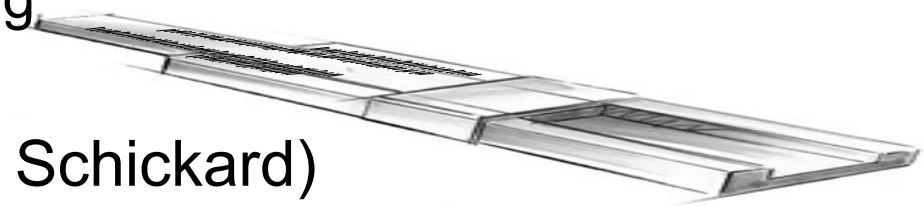
Inhalt

- Historie, Informationen und Daten, Codes
- Zahlendarstellungen, Rechnen mit Dualzahlen, Negative Zahlen, Codesicherung, analoge und digitale Daten
- Aussagenlogik, Schalternetze, Von-Neumann-Rechner
- Programmierung, Datentypen, Variablen, Dateneingabe, Ausdrücke, Konstanten, Kontrollstrukturen
- Fallunterscheidungen, Abzweigung, Verzweigung, Zahlen sortieren, Logische Operatoren, Geschachtelte Fallunterscheidungen, Bäume
- Zählschleifen, Abkürzungen, for-Schleife, Verschachtelte Schleifen, Allgemeine Schleifen, do-while-Schleife, Blöcke
- Top-Down-Entwurf, Maximumsuche, Felder
- Klassen, Methoden, Methodenparameter, Methoden überladen, Konstruktoren

Einleitung

Rechenmaschinen

- **Rechenstäbe** des Schotten John Napier (1550-1617)
- **1622 Rechenschieber** (William Oughtred)
  - Logarithmische Zahlendarstellung



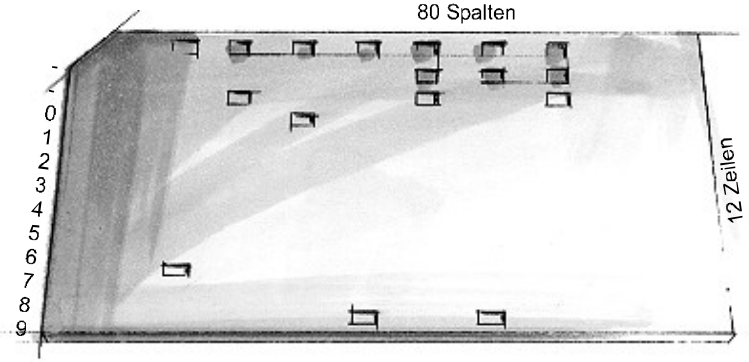
- **1623 Rechenmaschine** (Wilhelm Schickard)
  - Addition und Subtraktion bis zu sechsstelliger Zahlen
- **1643 Pascaline** (Blaise Pascal)
  - Addition und Subtraktion max. siebenstelliger Zahlen
  - Gebaut für seinen Vater, der Steuerbeamter war
- **1673 Rechenmaschine** (Gottfried Wilhelm Leibniz)
  - Staffelwalzenprinzip
  - Multiplikation durch fortgesetztes Addieren

Vorläufer (1)

- **1805 Lochkarte**
  (Joseph-Marie Jacquard)
  - Erstmalige Möglichkeit der Speicherung
- **1833 Analytische Maschine**
  (Charles Babbage)
  - Programmgesteuerte Rechenmaschine
  - Rechenwerk hoher Stellenzahl
  - Paralleler Zehnerübertrag
  - Lochbandsteuerung
  - Speicherwerk für 1000 Zahlen zu 50 Stellen
    - Baugruppen, die auch in modernen Rechnern vorhanden sind
  - Druckwerk für die Ausgabe
  - Bau scheiterte an technischen Unzulänglichkeiten der Zeit

Vorläufer (2)

- **1886 Lochkartenmaschine**
  (Hermann Hollerith)
  - Elektromagnetische Sortier- und Zählmaschine zur Auswertung von Lochkarten im 20 Dollar Format
  - Anlage bestand aus Kartenlocher, manuellem elektromagnetischem Zähler und Sortiereinrichtung
  - 1890 zur Erleichterung der Volkszählung in den USA eingesetzt
  - Bearbeitungszeit konnte von 7,5 auf 2,5 Jahre reduziert werden
  - Kartenform und Code bis in die 1980er Jahre eingesetzt



Die Entwicklung des Computers (1)

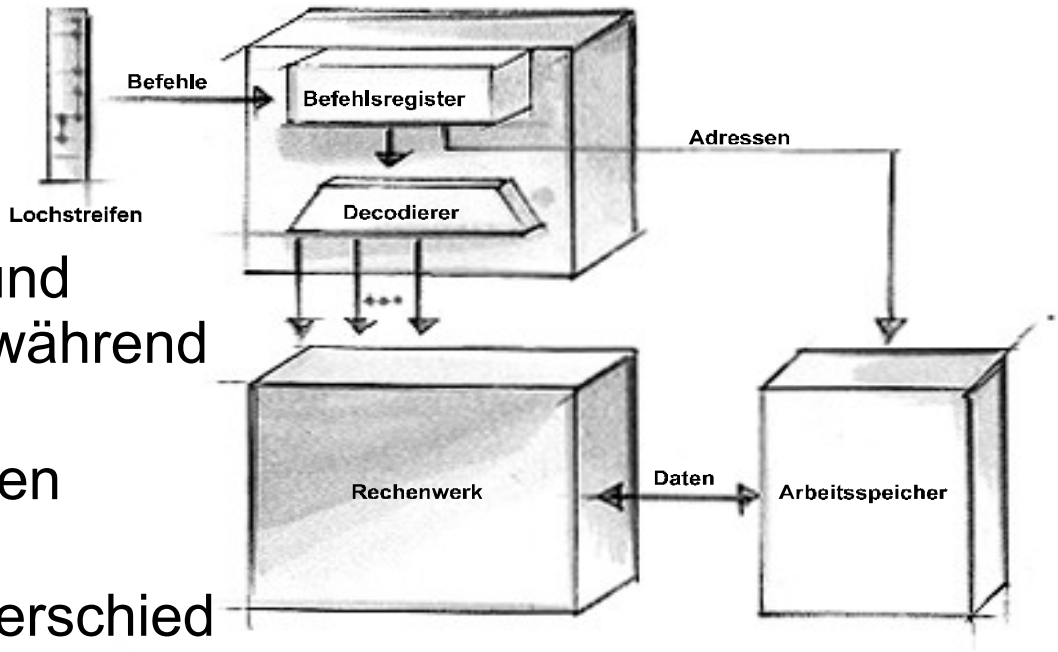
- **1934 - 1943 Zuse Z3**
  (Konrad Zuse)
  - Erster funktionsfähiger Computer
  - Durch Fortschritte in der Elektromechanik ermöglicht
  - Rechenzeiten
  - Addition: 0,3s, Multiplikation: 5s
  - Interne Zahlendarstellung durch Dualzahlen
    - Bei Eingabe/Ausgabe Dezimal-Dual- bzw. Dual-Dezimal-Konvertierung erforderlich
  - Rechenprogramm von gelochtem 8-spurigem Kinofilm abgetastet

Die Entwicklung des Computers (2)

- **1946 Eniac = 1. Generation**
  (John P. Eckert und John W. Mauchly)
  - **E**lectronic **N**umerical **I**ntegrator **A**nd **C**omputer
  - Röhrentechnik ersetzt Relaistechnik
    (ca. 18000 Röhren)
  - Schaltzeiten gegenüber der Relaistechnik stark verringert
    - Ca. 1000 Einzelbefehle pro Sekunde
    - Addition: 0,2ms, Multiplikation: 2ms
  - Erster Einsatz 1946 in Los Alamos
  - Berechnung ballistischer Feuertabellen
  - Arithmetische Einheit als Ersatz für die Rechenmaschine
  - Speicher als Notizblock
  - Programm ersetzte Anweisungen eines menschlichen Operators

Die Entwicklung des Computers (3)

- **1946 - 1952 Neumann-Maschine** (Johann (John) von Neumann)
  - **Programm zusammen mit den Daten in einem gemeinsamen Speicher**
  - Erlaubt Programmsprünge und das Verändern der Befehle während eines Rechenvorganges
  - Programme können wie Daten behandelt werden
  - Maschine macht keinen Unterschied zwischen Befehlen und Operanden
  - **Wesentliches Konzept, welches bis heute Grundlage der Rechnertechnik ist!**
  - **"Von-Neumann-Rechner"**



Computergenerationen von 1955 – 1990 (1)

- **1955 - 1960 Tradic = 2. Generation** (J.H. Felker)
  - **TRA**nsistor **DI**gital **C**omputer
  - Transistoren und Dioden (Halbleitertechnik)
  - Ca. 10.000 Einzelbefehle pro Sekunde
  - 700 Transistoren und 10,000 Germanium-dioden
  - Entwickelt als "Leichtgewichtcomputer" für den Flugzeugeinsatz

Computergenerationen von 1955 – 1990 (2)

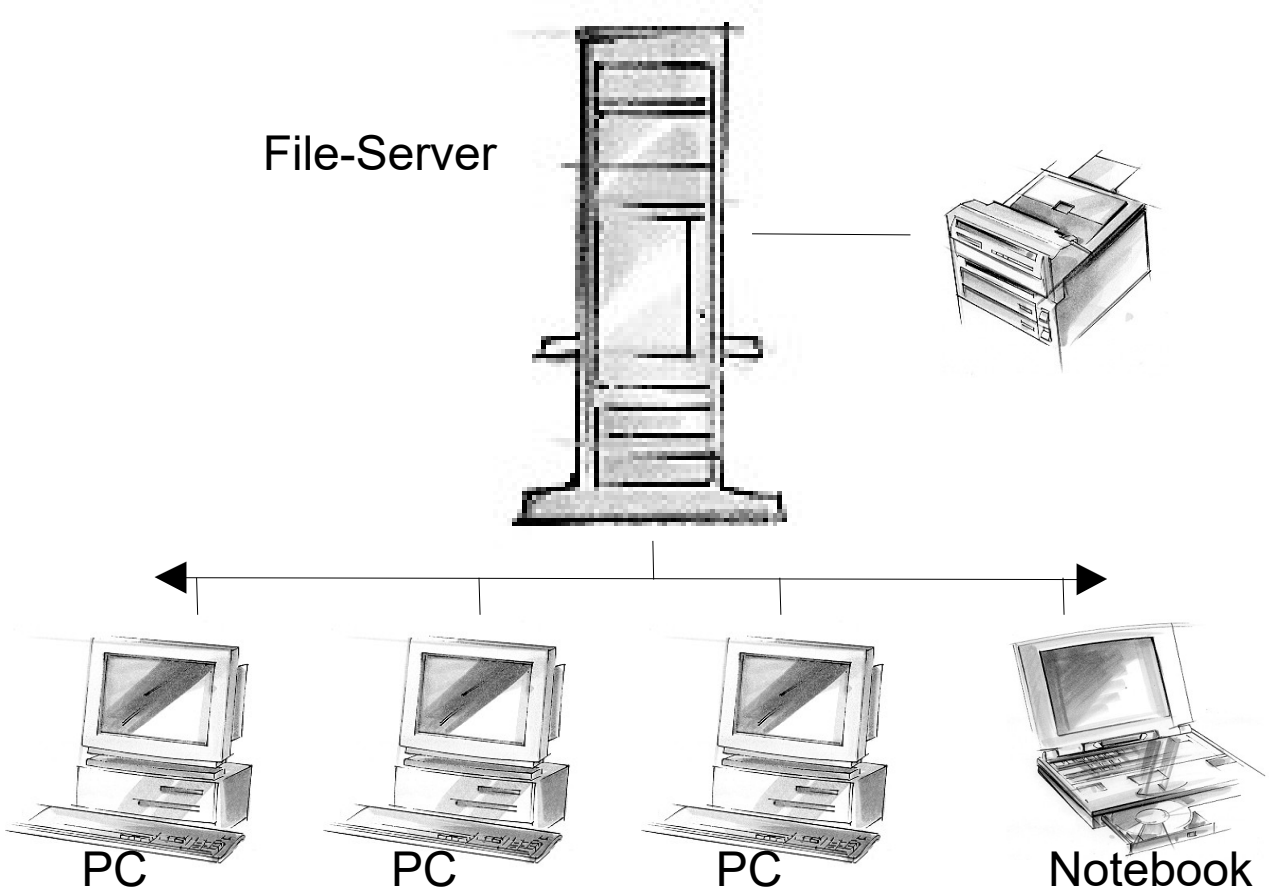
- **1962 - 1970 Integrierte Schaltkreise: 3. Generation**
  - 100 Transistoren auf drei Quadratmillimetern
  - Ca. 1 Million Einzelbefehle pro Sekunde
- **1968 Hochintegrierte Schaltkreise: 4. Generation**
  - Beschichtungs-, Ätz- und Aufdampfprozesse auf Siliziumscheiben
  - Ca. 10 Millionen Einzelbefehle pro Sekunde
- **1980 Cray-Computer: 5. Generation**
  - Mehrere Prozessoren werden miteinander verbunden
- **1985 Transputer**
  - "**TRANS**mitter and com**PUTER**", "**TRANS**istor and com**PUTER**"

Personalcomputer

- **1974 Die ersten Homecomputer**
  - ALTAIR-8800 (Bausatz für 395 Dollar)
  - Commodore (PET)
  - Tandy Radio Shack (TRS-80)
  - Alle programmierbar in BASIC
- **1977 Apple-Computer**
- **1981 IBM-Personalcomputer**
  - Grundstein für den heutigen Personalcomputer Standard
  - Prozessor von Intel
  - Betriebssystem MS-DOS von Microsoft
- **1984 Apple Macintosh**
  - Grafische Benutzerführung

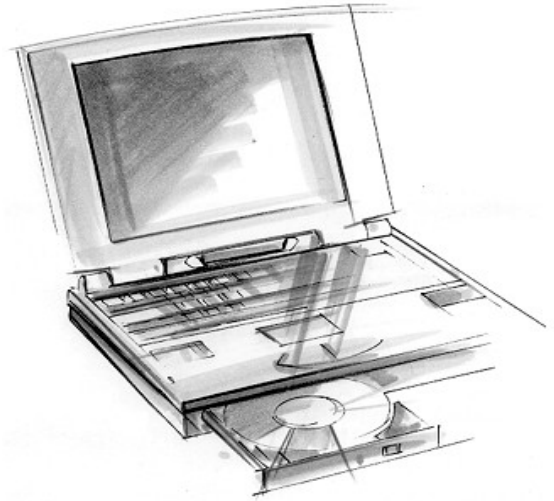
Die Vernetzung von Computern

- Ab ca. 1985
  - Rechnernetzwerke
  - Email
  - News
  - Talk



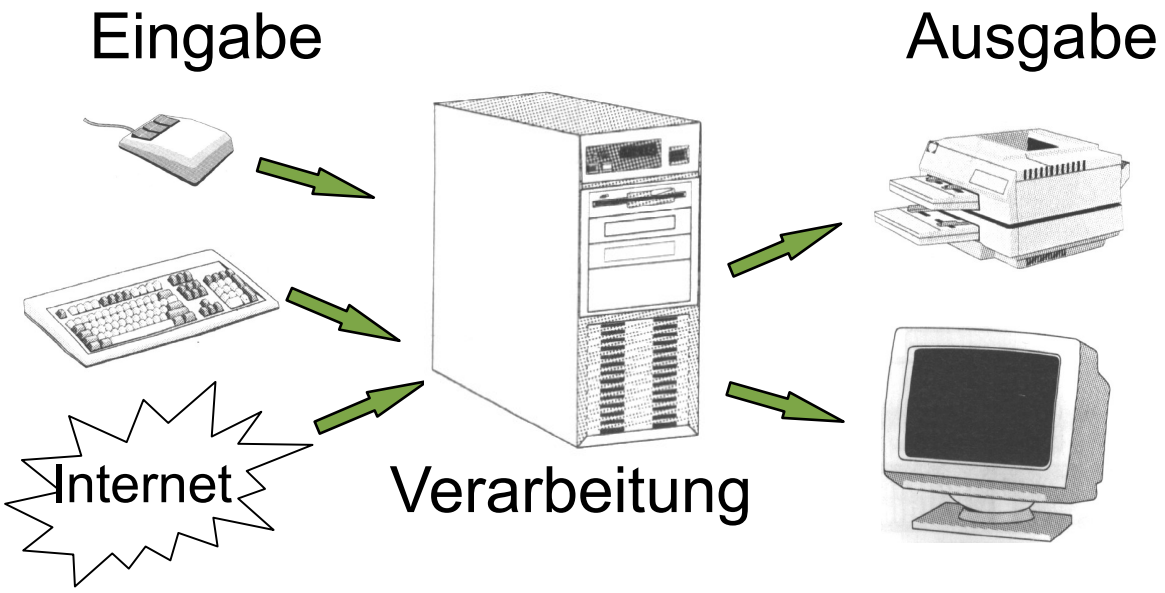
Internet / Multimedia

- **Seit 1993**
  - Internet
    - Medium für weltweite Information und Kommunikation
    - World Wide Web: WWW
  - Multimedia
    - Zusammenwirken von Text, Bild, Video und Ton auf dem Computer
  - Virtual und Augmented Reality
  - Tragbare Computer (Laptops/Notebooks/Tablets)
  - PDAs, Smartphones
  - Multitouch-Displays



Grundlegende Funktionsweise eines Computers

- Verarbeitung von Daten nach einem fest vorgegebenen und strikt eingehaltenen Verhaltensmuster von geordneten Vorschriften
- Verhaltensmuster wird als **Algorithmus** bezeichnet



Informationen und Daten

- **Information: Wissen (Kenntnisse) über Sachverhalte oder Vorgänge**
- **Informationen werden üblicherweise durch Aneinanderfügen (Konkatenation) von Zeichen dargestellt**
  - Z. B. Schriftzeichen
    - Buchstaben (A bis Z), Ziffern (0 bis 9)
    - Sonderzeichen (, . - ; : ? $ § “ ! & %)
- **Daten**
  - Zum Zweck der Verarbeitung codierte Informationen
- **Zusammenhang zwischen Daten und Information nicht eindeutig**
  - Dieselbe Information ist in unterschiedlicher Weise ausdrückbar
    - Z. B.: Morse-Alphabet, mathematische Formelsprache, Zeichensprache

Zeichenvorrat

- Definition: Ein "Zeichen" ist ein Element einer endlichen Menge von unterscheidbaren Elementen, dem "Zeichenvorrat" [2]
- **Beispiele für Zeichenvorräte:**
  - Lateinische Schriftzeichen
  - Arabische Schriftzeichen
  - Blindenschrift (Brailleschrift)
  - Zeichensprache
  - Morsecode
  - Spielkartenfarben
  - Farben der Verkehrsampel

Alphabet

- **Definition: Ein Zeichenvorrat, in dem eine lineare Reihenfolge der Zeichen festgelegt ist, heißt "Alphabet"** [2]
- Begriff des Alphabets in Informatik verallgemeinert zu "endliche Menge unterschiedlicher geordneter Zeichen"
  - Darum Mengenklammern verwendet
- **Beispiele:**
  - {a, b, c, ..., A, B, C, ..., ö, ä, Ü, ...}: Alphabet der Buchstaben
  - {0,1, 2, 3, 4, 5, 6, 7, 8, 9}: Alphabet der Dezimalziffern
  - {rot, gelb, grün}: Alphabet der Verkehrsampelsignale
  - {sehr gut, ..., ungenügend}: Alphabet der Schulnoten
    - Zeichen kann auch Wort oder sogar Wortfolge sein!

Binärer Zeichenvorrat

- **Von großer technischer Bedeutung sind Zeichenvorräte, die nur aus zwei Zeichen bestehen**
  - Kann als Schalter offen/geschlossen interpretiert werden
  - Einfache Implementierung
- **Definition:** Ein Zeichenvorrat mit nur zwei Zeichen heißt "**binärer Zeichenvorrat**"
- Definition: Die Zeichen eines binären Zeichenvorrats heißen "**Binärzeichen**" oder "**Bit**"
  - 1 Bit = **1 Binary Digit**
  - Ein Bit ist die kleinste mögliche Informationseinheit
- siehe [2]

Beispiele für binäre Zeichenvorräte

- Farbenpaar {"rot", "grün"}
- Intensitätspaar {"hell", "dunkel"}
- Ziffernpaar {0, 1}
- Zustandspaar {"gelocht", "ungelocht"},
  {"Vertiefung, "keine Vertiefung"}
- Wahrheitswerte {"falsch", "wahr"}
- Spannungen {0V, 5V}
- In diesem Kurs werden wir binäre Zeichen mit dem Zeichenpaar {0, 1} darstellen

Binärwort

- Definition: Ein "Wort" ist eine Folge von Zeichen, die in einem bestimmten Zusammenhang als Einheit betrachtet wird

- Definition: Wörter über einem binären Zeichenvorrat heißen "Binärwörter"
  - Ein Binärwort entsteht durch die Aneinanderreihung (Konkatenation) von Bits

- Definition: Die Anzahl der Binärzeichen eines Binärwortes heißt "Wortlänge"

- siehe [2], [7]

Codierung

- Definition: Eine "Codierung" ist eine Vorschrift für die eindeutige Zuordnung eines Zeichens eines Zeichenvorrats zu derjenigen eines anderen Zeichenvorrats
  - Der Vorgang des Zuordnens der Zeichen wird als "Codieren" bezeichnet; seine Umkehrung als "Decodierung"
- Definition: Ein "Code" ist der bei der Codierung als Bildmenge auftretende Zeichenvorrat
- Definition: Ein "n-Bit-Code" ist ein Zeichenvorrat, dessen Zeichen n-Bit-Binärwörter sind
  - Bei technischen Codierungen werden die Zeichen des ursprünglichen Zeichenvorrats üblicherweise durch Binärwörter codiert
- siehe [2], [7]

Codes

- Ein Bit kann zwei Zeichen eines anderen Alphabets darstellen bzw. codieren
- Mehr Zeichen können durch Zusammenfassen mehrerer Bits zu einem Binärwort dargestellt werden
  - Werden 2 Bits zusammengefasst, so lassen sich bereits 4 Zeichen als Binärwörter darstellen: 00, 01, 10, 11
  - Z. B. Codierung von Ampelzuständen:
    Rot=00, Gelb=01, Grün=10
- **Die Zeichen aller möglichen Alphabete lassen sich durch Binärwörter ausdrücken**

Beispiel

- Das Alphabet der Kleinbuchstaben ausgedrückt durch einen 5-Bit-Code
- a $\rightarrow$ 00000
  b $\rightarrow$ 00001
  c $\rightarrow$ 00010
  d $\rightarrow$ 00011
  e $\rightarrow$ 00100
  usw.
- Anzahl der codierbaren Zeichen: $2^5 = 32$
- Allgemein: Mit *n* Binärzeichen lassen sich 2^n Zeichen eines Alphabets darstellen (codieren)

Codetabelle

- Code dargestellt in Form einer **Codetabelle**
- Z. B. Codierung des Alphabets der 10 Dezimalziffern durch 4-Bit-Binärwörter:
  - Die Binärwörter 1010, 1011, 1100, 1101, 1110, 1111 werden hierbei nicht benötigt
- Die Dezimalzahl 314 würde mit dieser Codetabelle wie folgt codiert:

  0011 0001 0100
- Diese Codetabelle stellt nur eine von vielen Möglichkeiten dar, die 10 Dezimalziffern durch 4-Bit-Binärwörter zu codieren
- Die Codierung irgendeines Alphabets durch Binärwörter wird als **Binärcodierung** bezeichnet

dezimal	binär
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Texte innerhalb des Computers

- Alphanumerische Codes
- Beispiel: 8-Bit-Code ISO/IEC 8859, Teil 1 (sog. Latein 1)
  - Zeichensatz für Europa, Amerika, Australien

Bit																			HEX
				8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	
				7	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	
				6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	
				5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	
4	3	2	1		00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	
0	0	0	0	00			SP	0	@	P	`	p			NS	°	À	Ð	à	ð	0
0	0	0	1	01			!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ	1
0	0	1	0	02			"	2	B	R	b	r			¢	²	Â	Ò	â	ò	2
0	0	1	1	03			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó	3
0	1	0	0	04			$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô	4
0	1	0	1	05			%	5	E	U	e	u			¥	µ	Å	Õ	å	õ	5
0	1	1	0	06			&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö	6
0	1	1	1	07			'	7	G	W	g	w			§	·	Ç	×	ç	÷	7
1	0	0	0	08			(	8	H	X	h	x			¨	¸	È	Ø	è	ø	8
1	0	0	1	09			)	9	I	Y	i	y			©	¹	É	Ù	é	ù	9
1	0	1	0	10			*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú	A
1	0	1	1	11			+	;	K	[	k	{			«	»	Ë	Û	ë	û	B
1	1	0	0	12			,	<	L	\	l	|			¬	¼	Ì	Ü	ì	ü	C
1	1	0	1	13			-	=	M	]	m	}			SH	½	Í	Ý	í	ý	D
1	1	1	0	14			.	>	N	^	n	~			®	¾	Î	Þ	î	þ	E
1	1	1	1	15			/	?	O	_	o	DE			¯	¿	Ï	ß	ï	ÿ	F
HEX					0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

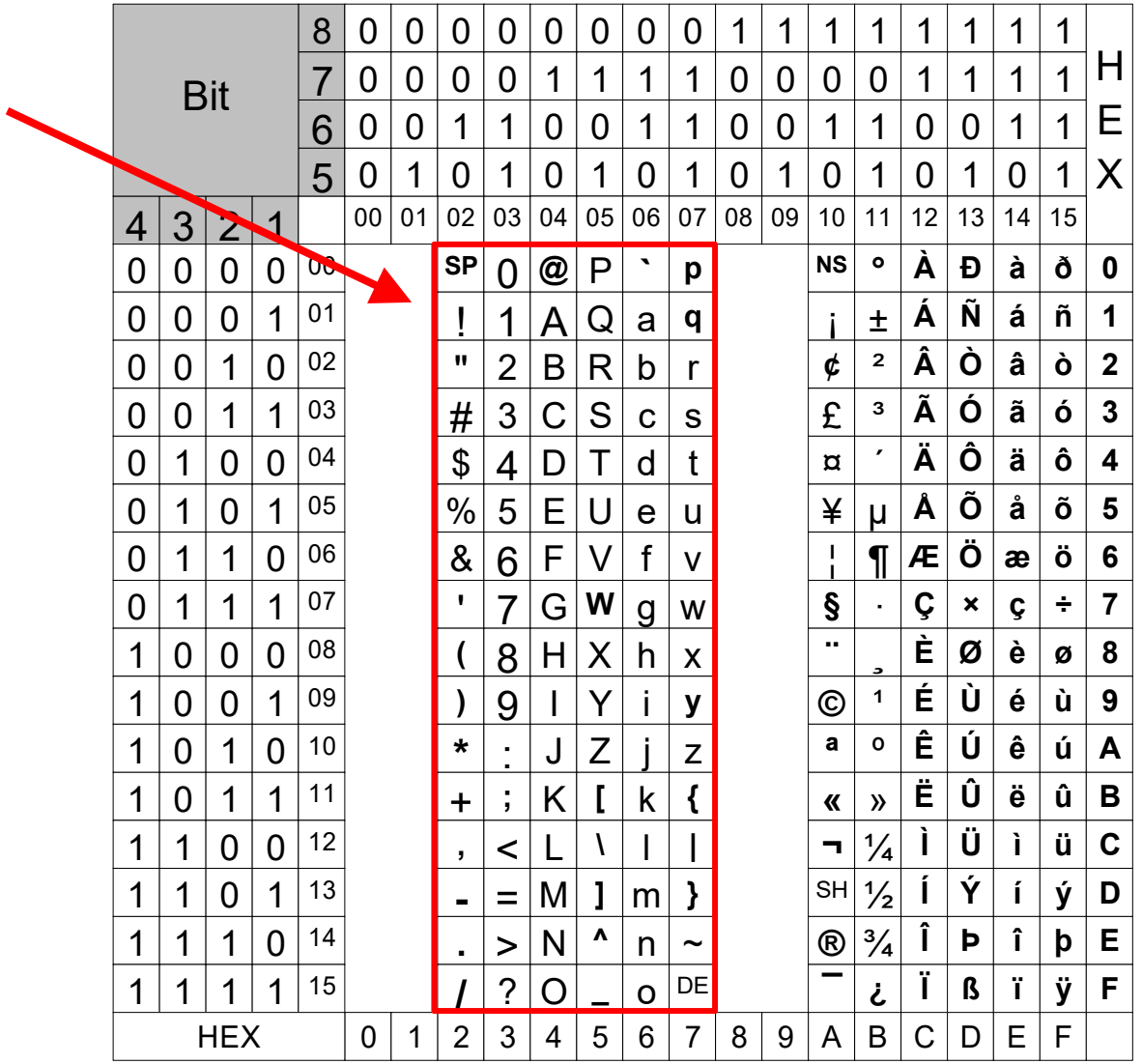
Referenzversionen

- **Linke Hälfte:**
  - 7-Bit-Code
- Internationale Referenzversion (IRV)
- ISO/IRC 646: 84+12 Schriftzeichen
- Nationale Festlegung der Schriftzeichen der 12 grauen Felder
- Z. B. deutsche Refe-renzversion: DIN 66003
  - [→ Ä, \ → Ö,] → Ü, {→ ä, | → ö, } → ü, ~ → ß

Bit 8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	HEX
Bit 7	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	
Bit 6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	
Bit 5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	
4 3 2 1	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	
0 0 0 0	00		SP	0	@	P	`	p			NS	º	À	Đ	à	ð	0
0 0 0 1	01		!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ	1
0 0 1 0	02		"	2	B	R	b	r			¢	²	Â	Ò	â	ò	2
0 0 1 1	03		#	3	C	S	c	s			£	³	Ã	Ó	ã	ó	3
0 1 0 0	04		$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô	4
0 1 0 1	05		%	5	E	U	e	u			¥	µ	Å	Õ	å	õ	5
0 1 1 0	06		&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö	6
0 1 1 1	07		'	7	G	W	g	w			§	·	Ç	×	ç	÷	7
1 0 0 0	08		(	8	H	X	h	x			¨	¸	È	Ø	è	ø	8
1 0 0 1	09		)	9	I	Y	i	y			©	¹	É	Ù	é	ù	9
1 0 1 0	10		*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú	A
1 0 1 1	11		+	;	K	[	k	{			«	»	Ë	Û	ë	û	B
1 1 0 0	12		,	<	L	\	l	|			¬	¼	Ì	Ü	ì	ü	C
1 1 0 1	13		-	=	M	]	m	}			SH	½	Í	Ý	í	ý	D
1 1 1 0	14		.	>	N	^	n	~			®	¾	Î	Þ	î	þ	E
1 1 1 1	15		/	?	O	_	o	DE			¯	¿	Ï	ß	ï	ÿ	F
HEX		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Amerikanische Referenzversion

- **ASCII**
  - "**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange"
- Identisch zu IRV



Bit 8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	HEX
Bit 7	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	
Bit 6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	
Bit 5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	
4 3 2 1	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	
0 0 0 0 00			SP	0	@	P	`	p			NS	º	À	Đ	à	ð	0
0 0 0 1 01			!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ	1
0 0 1 0 02			"	2	B	R	b	r			¢	²	Â	Ò	â	ò	2
0 0 1 1 03			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó	3
0 1 0 0 04			$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô	4
0 1 0 1 05			%	5	E	U	e	u			¥	µ	Å	Õ	å	õ	5
0 1 1 0 06			&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö	6
0 1 1 1 07			'	7	G	W	g	w			§	·	Ç	×	ç	÷	7
1 0 0 0 08			(	8	H	X	h	x			¨	¸	È	Ø	è	ø	8
1 0 0 1 09			)	9	I	Y	i	y			©	¹	É	Ù	é	ù	9
1 0 1 0 10			*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú	A
1 0 1 1 11			+	;	K	[	k	{			«	»	Ë	Û	ë	û	B
1 1 0 0 12			,	<	L	\	l	|			¬	¼	Ì	Ü	ì	ü	C
1 1 0 1 13			-	=	M	]	m	}			SH	½	Í	Ý	í	ý	D
1 1 1 0 14			.	>	N	^	n	~			®	¾	Î	Þ	î	þ	E
1 1 1 1 15			/	?	O	_	o	DE			¯	¿	Ï	ß	ï	ÿ	F
HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

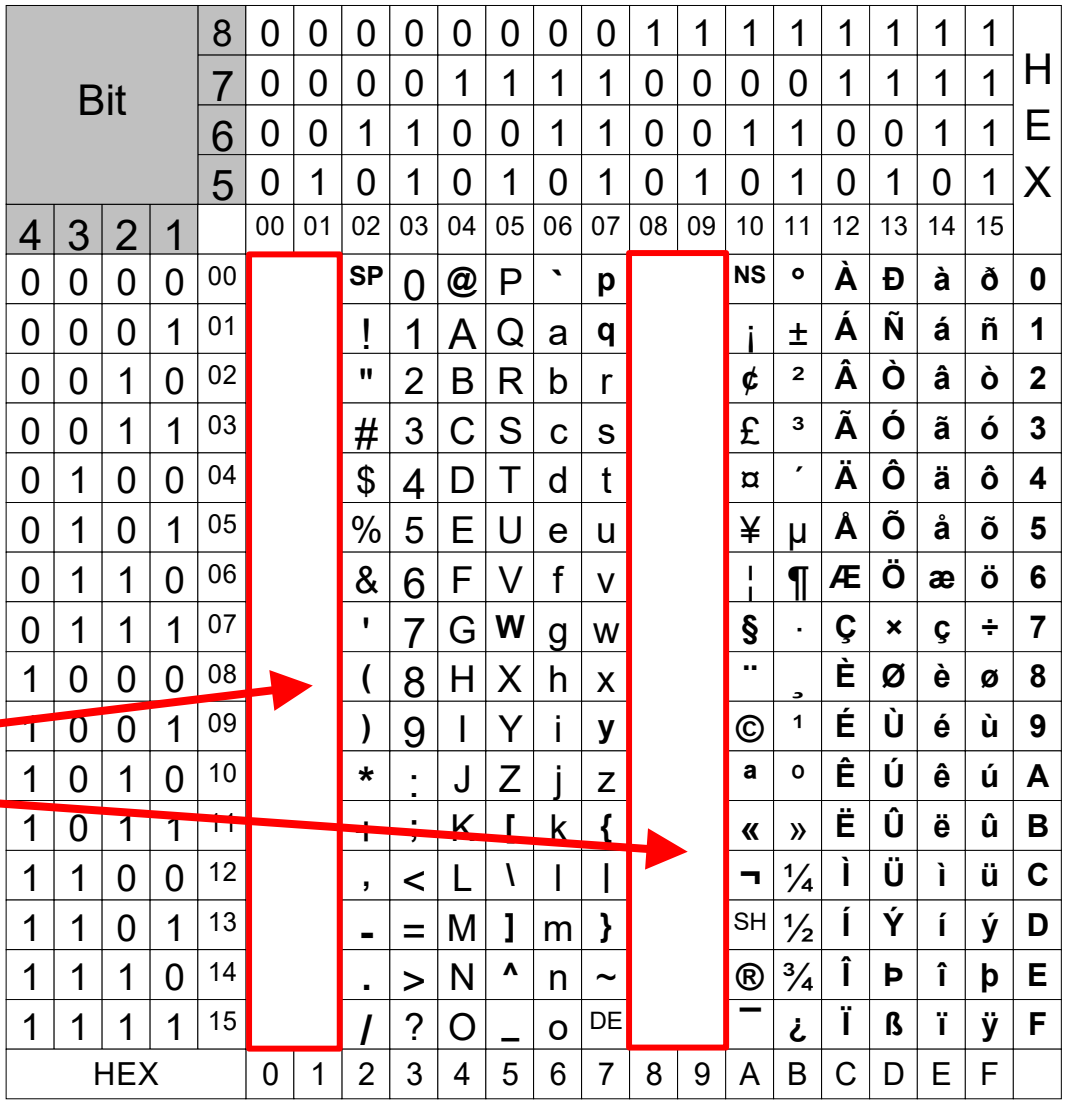
Sprachgruppen

- **Rechte Hälfte:** Codierung der Sprachgruppen
- Hier: Latein 1
  - Zeichensatz für Europa, Amerika, Australien
- Andere z. B.:
  - Latein 2: Für osteuropäische Sprachen
  - Kyrillisch, Griechisch, Arabisch, Hebräisch etc.

Bit																			
				8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	H
				7	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	E
				6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	E
				5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	X
4	3	2	1		00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	
0	0	0	0	00			SP	0	@	P	`	p			NS	º	À	Ð	à	ð	0
0	0	0	1	01			!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ	1
0	0	1	0	02			"	2	B	R	b	r			¢	²	Â	Ò	â	ò	2
0	0	1	1	03			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó	3
0	1	0	0	04			$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô	4
0	1	0	1	05			%	5	E	U	e	u			¥	µ	Å	Õ	å	õ	5
0	1	1	0	06			&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö	6
0	1	1	1	07			'	7	G	W	g	w			§	·	Ç	×	ç	÷	7
1	0	0	0	08			(	8	H	X	h	x			¨	¸	È	Ø	è	ø	8
1	0	0	1	09			)	9	I	Y	i	y			©	¹	É	Ù	é	ù	9
1	0	1	0	10			*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú	A
1	0	1	1	11			+	;	K	[	k	{			«	»	Ë	Û	ë	û	B
1	1	0	0	12			,	<	L	\	l	|			¬	¼	Ì	Ü	ì	ü	C
1	1	0	1	13			-	=	M	]	m	}			SH	½	Í	Ý	í	ý	D
1	1	1	0	14			.	>	N	^	n	~			®	¾	Î	Þ	î	þ	E
1	1	1	1	15			/	?	O	_	o	DE			¯	¿	Ï	ß	ï	ÿ	F
HEX					0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Steuerzeichen

- Codes wurden ursprünglich für die Datenkommunikation entwickelt
- Austauschcodes
- Daher Platz für Steuerzeichen vorgesehen
- 32-64 Steuerzeichen codierbar



Bit																		
				8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	
				7	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	H
				6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	E
				5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	X
4	3	2	1		00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	
0	0	0	0	00			SP	0	@	P	`	p			NS	º	À	Đ	à	ð	0
0	0	0	1	01			!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ	1
0	0	1	0	02			"	2	B	R	b	r			¢	²	Â	Ò	â	ò	2
0	0	1	1	03			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó	3
0	1	0	0	04			$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô	4
0	1	0	1	05			%	5	E	U	e	u			¥	µ	Å	Õ	å	õ	5
0	1	1	0	06			&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö	6
0	1	1	1	07			'	7	G	W	g	w			§	·	Ç	×	ç	÷	7
1	0	0	0	08			(	8	H	X	h	x			¨	¸	È	Ø	è	ø	8
1	0	0	1	09			)	9	I	Y	i	y			©	¹	É	Ù	é	ù	9
1	0	1	0	10			*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú	A
1	0	1	1	11			+	;	K	[	k	{			«	»	Ë	Û	ë	û	B
1	1	0	0	12			,	<	L	\	l	|			¬	¼	Ì	Ü	ì	ü	C
1	1	0	1	13			-	=	M	]	m	}			SH	½	Í	Ý	í	ý	D
1	1	1	0	14			.	>	N	^	n	~			®	¾	Î	Þ	î	þ	E
1	1	1	1	15			/	?	O	_	o	DE			¯	¿	Ï	ß	ï	ÿ	F
HEX					0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Steuerzeichen des 7-Bit ASCII Zeichensatzes

Übertragungssteuerzeichen

SOH	start of heading	Kopfanfang
STX	start of text	Textanfang
ETX	end of text	Textende
EOT	end of trans-mission	Übertragungsende
ENQ	enquiry	Stationsaufforderung
ACK	acknowledge	positive Rückmeldung
DLE	data link escape	Datenübertragungs-umschaltung
NACK	negative acknow-ledge	negative Rückmeldung
SYN	synchronous idle	Synchronisierung
ETB	end of trans-mission block	Datenübertragungs-block Ende

Formatsteuerzeichen

BS	backspace	Rückwärtsschritt
HT	horizontal tabulation	Zeichen-Tabulator
LF	line feed	Zeilenschritt
VT	vertical tabulation	Zeilen-Tabulator
FF	form feed	Formularvorschub
CR	carriage return	Rückführen

Code-Erweiterungszeichen

SO	shift out	Dauerumschaltung
SI	shift in	Rückschaltung
ESC	Escape	Escape

Gerätesteuerzeichen

DC1	device control one	Gerätesteuerzeichen eins
DC2	device control two	Gerätesteuerzeichen zwei
DC3	device control three	Gerätesteuerzeichen drei
DC4	device control four	Gerätesteuerzeichen vier

Informationstrennzeichen

US	unit separator	Teilgruppen-Trenn-zeichen
RS	record separator	Untergruppen-Trenn-zeichen
GS	group separator	Gruppen-Trenn-zeichen
FS	file separator	Hauptgruppen-Trenn-zeichen

Sonstige Steuerzeichen

NUL	null	Null
BEL	bell	Klingel
CAN	cancel	ungültig
EM	end of medium	Aufzeichnungsende
SUB	substitute	Substitution

Universelle Codes

- Heutzutage vermehrt in Gebrauch
- **Unicode**
  - Zeichencodierung mit variabler Bitzahl
  - Prominente Vertreter:
  - UTF-16
    - Zeichen durch 1 bis 2 16 Bit-Wörter codiert
    - Windows, OS X, Java, JavaScript
  - UTF-8
    - Web
    - In den ersten 128 Zeichen deckungsgleich mit ASCII
    - Damit auch für einfache Editoren benutzbar
    - Andere Zeichensätze erfordern zusätzliche Bytes
  - UTF: Unicode Transformation Format

Beispiel

- Die folgende ASCII-Zeichenfolge soll codiert werden:

  Info1

Bit																		
				8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	
				7	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	HEX
				6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	
				5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	
4	3	2	1		00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	
0	0	0	0	00			SP	0	@	P	`	p			NS	º	À	Đ	à	ð	0
0	0	0	1	01			!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ	1
0	0	1	0	02			"	2	B	R	b	r			¢	²	Â	Ò	â	ò	2
0	0	1	1	03			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó	3
0	1	0	0	04			$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô	4
0	1	0	1	05			%	5	E	U	e	u			¥	µ	Å	Õ	å	õ	5
0	1	1	0	06			&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö	6
0	1	1	1	07			'	7	G	W	g	w			§	·	Ç	×	ç	÷	7
1	0	0	0	08			(	8	H	X	h	x			¨	¸	È	Ø	è	ø	8
1	0	0	1	09			)	9	I	Y	i	y			©	¹	É	Ù	é	ù	9
1	0	1	0	10			*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú	A
1	0	1	1	11			+	;	K	[	k	{			«	»	Ë	Û	ë	û	B
1	1	0	0	12			,	<	L	\	l	|			¬	¼	Ì	Ü	ì	ü	C
1	1	0	1	13			-	=	M	]	m	}			SH	½	Í	Ý	í	ý	D
1	1	1	0	14			.	>	N	^	n	~			®	¾	Î	Þ	î	þ	E
1	1	1	1	15			/	?	O	_	o	DE			¯	¿	Ï	ß	ï	ÿ	F
HEX					0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Übung

- Codieren Sie die folgende ASCII-Zeichenfolge als binären Datenblock:

  Computer?

Bit																			
				8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	
				7	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	HEX
				6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	
				5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	
4	3	2	1		00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	
0	0	0	0	00			SP	0	@	P	`	p			NS	º	À	Đ	à	ð	0
0	0	0	1	01			!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ	1
0	0	1	0	02			"	2	B	R	b	r			¢	²	Â	Ò	â	ò	2
0	0	1	1	03			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó	3
0	1	0	0	04			$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô	4
0	1	0	1	05			%	5	E	U	e	u			¥	µ	Å	Õ	å	õ	5
0	1	1	0	06			&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö	6
0	1	1	1	07			'	7	G	W	g	w			§	·	Ç	×	ç	÷	7
1	0	0	0	08			(	8	H	X	h	x			¨	¸	È	Ø	è	ø	8
1	0	0	1	09			)	9	I	Y	i	y			©	¹	É	Ù	é	ù	9
1	0	1	0	10			*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú	A
1	0	1	1	11			+	;	K	[	k	{			«	»	Ë	Û	ë	û	B
1	1	0	0	12			,	<	L	\	l	|			¬	¼	Ì	Ü	ì	ü	C
1	1	0	1	13			-	=	M	]	m	}			SH	½	Í	Ý	í	ý	D
1	1	1	0	14			.	>	N	^	n	~			®	¾	Î	Þ	î	þ	E
1	1	1	1	15			/	?	O	_	o	DE			¯	¿	Ï	ß	ï	ÿ	F
HEX					0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Übung

- Decodieren Sie den folgenden binären Datenblock:

01001000
01101001
00100000
01110100
01101000
01100101
01110010
01100101
00100001

Bit																		
				8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	HEX
				7	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	
				6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	
				5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	
4	3	2	1		00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	
0	0	0	0	00			SP	0	@	P	`	p			NS	°	À	Đ	à	ð	0
0	0	0	1	01			!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ	1
0	0	1	0	02			"	2	B	R	b	r			¢	²	Â	Ò	â	ò	2
0	0	1	1	03			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó	3
0	1	0	0	04			$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô	4
0	1	0	1	05			%	5	E	U	e	u			¥	µ	Å	Õ	å	õ	5
0	1	1	0	06			&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö	6
0	1	1	1	07			'	7	G	W	g	w			§	·	Ç	×	ç	÷	7
1	0	0	0	08			(	8	H	X	h	x			¨	¸	È	Ø	è	ø	8
1	0	0	1	09			)	9	I	Y	i	y			©	¹	É	Ù	é	ù	9
1	0	1	0	10			*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú	A
1	0	1	1	11			+	;	K	[	k	{			«	»	Ë	Û	ë	û	B
1	1	0	0	12			,	<	L	\	l	|			¬	¼	Ì	Ü	ì	ü	C
1	1	0	1	13			-	=	M	]	m	}			SH	½	Í	Ý	í	ý	D
1	1	1	0	14			.	>	N	^	n	~			®	¾	Î	Þ	î	þ	E
1	1	1	1	15			/	?	O	_	o	DE			¯	¿	Ï	ß	ï	ÿ	F
HEX					0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Zahlendarstellungen

Dualzahlen

- **Computer arbeiten intern im Zahlensystem zur Basis 2, mit den "Dualziffern" 0 und 1**
  - Duales Zahlensystem wurde im 17. Jahrhundert von Gottfried Wilhelm Leibniz entwickelt
- **Definition: "Dualzahlen" sind Wörter aus Dualziffern; damit sind sie ebenfalls Binärwörter**
- **Binärwörter bzw. Dualzahlen dienen zur Darstellung jedweder Information in einem Computer**
  - Hierzu gehören - neben den zu verarbeitenden Daten - ebenfalls die Maschinenbefehle

Dezimalzahlen

- **Wenn wir eine Dezimalzahl $Z = ...a_2a_1a_0,a_{-1}a_{-2}...$ schreiben, so ist dies eine abgekürzte Schreibweise für:**
  $Z = ...a_2 * 10^2 + a_1 * 10^1 + a_0 * 10^0 + a_{-1} * 10^{-1} + a_{-2} * 10^{-2} ...$
  - Die Ziffern sind entsprechend ihres Gewichts von links nach rechts angeordnet
  - Einerstelle durch nachgestelltes Komma markiert
- Die Ziffern variieren im Dezimalsystem zwischen 0 und 9
- Gewichte sind Potenzen der Basis 10
- **Wie lassen sich Dezimalzahlen als Dualzahlen darstellen?**

Allgemeine Darstellung

- **Darstellung natürlicher Zahlen in B-adischen Systemen:**

- $Z = a_n \cdot B^n + a_{n-1} \cdot B^{n-1} + \ldots + a_1 \cdot B^1 + a_0 \cdot B^0 = \sum_{i=0}^{n} a_i \cdot B^i$

- Z = Zahl im B-adischen Zahlensystem
- B = Basis des Zahlensystems,
  z. B. 10 (dezimal), 2 (dual), 8 (oktal),
  16 (hexadezimal)
- B^i = Wertigkeit der i-ten Stelle
- a_i = Ziffer (Koeffizient) der i-ten Stelle aus der Menge der Ziffern {0, 1, 2, ..., B-1}

Konvertierung von Dezimalzahlen in Dualzahlen

- **Satz: Die ziffernmäßige Darstellung einer ganzen nicht-negativen Zahl z im Zahlensystem zur Basis B>=2,**

  $Z = a_n a_{n-1} \ldots a_1 a_0$

  ergibt sich aus folgender Vorschrift (Horner-Schema):

$$
\begin{aligned}
Z &= q_0 \cdot B + a_0, \\
q_0 &= q_1 \cdot B + a_1, \\
q_1 &= q_2 \cdot B + a_2, \\
&\vdots \\
q_k &= q_{k+1} \cdot B + a_{k+1}, \\
&\vdots \\
q_{n-1} &= q_n \cdot B + a_n = 0 \cdot B + a_n = a_n
\end{aligned}
$$

  - Für Dualsystem: $B = 2$
  - Die Ziffern $a_0,\ldots,a_n$ werden berechnet, indem die Zahl Z durch B dividiert und der Divisionsrest notiert wird
  - Der letzte Divisionsrest liefert a_n

- **Die Divisionsreste a_i (i=0,...,n) sind die gesuchten Ziffern**

Beispiel

- **743 soll im Dualsystem dargestellt werden**
- $743 \div 2 = 371$ Rest 1: a_0
  $371 \div 2 = 185$ Rest 1: a_1
  $185 \div 2 = 92$ Rest 1: a_2
  $92 \div 2 = 46$ Rest 0: a_3
  $46 \div 2 = 23$ Rest 0: a_4
  $23 \div 2 = 11$ Rest 1: a_5
  $11 \div 2 = 5$ Rest 1: a_6
  $5 \div 2 = 2$ Rest 1: a_7
  $2 \div 2 = 1$ Rest 0: a_8
  $1 \div 2 = 0$ Rest 1: a_9
- $743_{10} = 1011100111_2$

$$
\begin{aligned}
Z &= q_0 \cdot B + a_0, \\
q_0 &= q_1 \cdot B + a_1, \\
q_1 &= q_2 \cdot B + a_2, \\
&\vdots \\
q_k &= q_{k+1} \cdot B + a_{k+1}, \\
&\vdots \\
q_{n-1} &= q_n \cdot B + a_n = \\
&\quad 0 \cdot B + a_n = a_n
\end{aligned}
$$

Beispiel

- **743 soll im Dualsystem dargestellt werden**

- $743 \div 2 = 371$ Rest 1: a_0 ← Least Significant Bit: LSB
  $371 \div 2 = 185$ Rest 1: a_1
  $185 \div 2 = 92$ Rest 1: a_2
  $92 \div 2 = 46$ Rest 0: a_3
  $46 \div 2 = 23$ Rest 0: a_4
  $23 \div 2 = 11$ Rest 1: a_5
  $11 \div 2 = 5$ Rest 1: a_6
  $5 \div 2 = 2$ Rest 1: a_7
  $2 \div 2 = 1$ Rest 0: a_8
  $1 \div 2 = 0$ Rest 1: a_9 ← Most Significant Bit: MSB

- $743_{10} = 1011100111_2$

Konvertierung von Dualzahlen in Dezimalzahlen

- **Satz:** Ist in einem Zahlensystem zur ganzzahligen Basis $B \geq 2$ mit den Ziffern $0 \leq a_i < B$ eine nicht-negative ganze Zahl Z dargestellt durch

  $$Z = a_n a_{n-1} \ldots a_1 a_0$$

  dann ist

  $$Z = \sum_{i=0}^{n} a_i \cdot B^i = a_n \cdot B^n + a_{n-1} \cdot B^{n-1} + \ldots + a_1 \cdot B^1 + a_0 \cdot B^0$$

  Diese Darstellung ist eindeutig

Beispiele

- Zehnersystem

210 (Stelle)

743

3 Einer $= 3 * 10^0 = 3$

4 Zehner $= 4 * 10^1 = 40$

7 Hunderter $= 7 * 10^2 = 700$

bzw.

$Z = 743 = 7*10^2 + 4*10^1 + 3*10^0$

- Dualsystem

98 7654 3210

10 1110 0111

1 * 1	$= 1 * 2^0$
1 * 2	$= 1 * 2^1$
1 * 4	$= 1 * 2^2$
0 * 8	$= 0 * 2^3$
0 * 16	$= 0 * 2^4$
1 * 32	$= 1 * 2^5$
1 * 64	$= 1 * 2^6$
1 * 128	$= 1 * 2^7$
0 * 256	$= 0 * 2^8$
1 * 512	$= 1 * 2^9$
	$= 743$

Übungen

- Konvertieren Sie die Dezimalzahl **813** in das Dualsystem

- Berechnen Sie zur Dualzahl **1101001001100** die zugehörende Dezimalzahl

Darstellung gebrochener Zahlen

- **Darstellung gebrochener Zahlen in B-adischen Systemen:**
- $$Z = a_n \cdot B^n + a_{n-1} \cdot B^{n-1} + \ldots + a_1 \cdot B^1 + a_0 \cdot B^0$$
  $$+ a_{-1} \cdot B^{-1} + a_{-2} \cdot B^{-2} + \ldots + a_{-(m-1)} \cdot B^{-(m-1)} + a_{-m} \cdot B^{-m}$$
  $$= \sum_{i=-m}^{n} a_i \cdot B^i = \sum_{i=0}^{n} a_i \cdot B^i + \sum_{i=-m}^{-1} a_i \cdot B^i$$
  $$= Z_v + Z_n$$
  - Z_v: Vorkommastellen, Z_n: Nachkommastellen
- **Beide Teile werden unabhängig voneinander konvertiert**

Konvertierung der Nachkommastellen (1)

- **Erfolgt wieder nach dem Horner-Schema:**
- $$\sum_{i=-m}^{-1} a_i \cdot B^i = \left(\ldots\left(a_{-m} \div B + a_{-(m-1)}\right) \div B + \ldots + a_{-1}\right) \div B$$
- Die Basis *B* tritt jetzt allerdings im Nenner auf
- Zur Bestimmung der Koeffizienten a_i wird daher nicht mehr durch die Basis *B* geteilt, sondern mit ihr multipliziert
- Die hierbei entstehenden Produkte bestehen aus einem ganzzahligen Anteil und Nachkommastellen

Konvertierung der Nachkommastellen (2)

- **Der ganzzahlige Anteil ist jeweils der Koeffizient**
- Mit den Nachkommastellen wird die Rechnung wiederholt bis diese verschwinden oder eine gegebene Genauigkeit erreicht ist

- **Beispiel: 0,6875 soll im Dualsystem dargestellt werden**

0,6875	* 2	=	0,375	+	1:	a_{-1} ←	Most Significant Bit: MSB
0,375	* 2	=	0,75	+	0:	a_{-2}	
0,75	* 2	=	0,5	+	1:	a_{-3}	
0,5	* 2	=	0,0	+	1:	a_{-4} ←	Least Significant Bit: LSB

$0{,}6875_{10} = 0{,}1011_2$

Beispiel

- **0,24 soll im Dualsystem dargestellt werden**

$$
\begin{array}{llllll}
0{,}24 * 2 & = & 0{,}48 & + & 0: & a_{-1} \\
0{,}48 * 2 & = & 0{,}96 & + & 0: & a_{-2} \\
0{,}96 * 2 & = & 0{,}92 & + & 1: & a_{-3} \\
0{,}92 * 2 & = & 0{,}84 & + & 1: & a_{-4} \\
0{,}84 * 2 & = & 0{,}68 & + & 1: & a_{-5} \\
0{,}68 * 2 & = & 0{,}36 & + & 1: & a_{-6} \\
0{,}36 * 2 & = & 0{,}72 & + & 0: & a_{-7} \\
0{,}72 * 2 & = & 0{,}44 & + & 1: & a_{-8}
\end{array}
$$

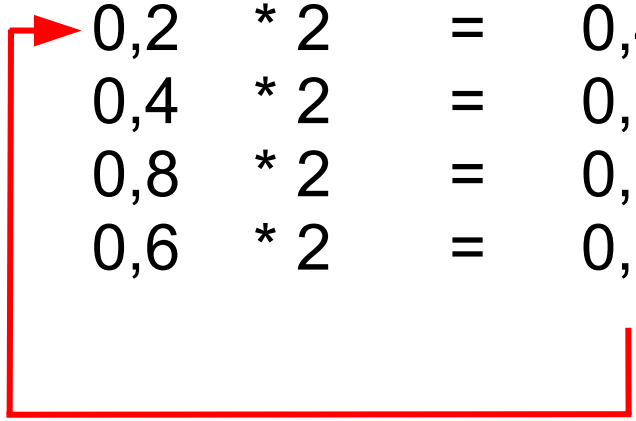
$0{,}24_{10} \approx 0{,}00111101_2$

Abbruch nach 8 Stellen

Beispiel

- **0,1 soll im Dualsystem dargestellt werden**

0,1	* 2	=	0,2	+	0:	a_{-1}
0,2	* 2	=	0,4	+	0:	a_{-2}
0,4	* 2	=	0,8	+	0:	a_{-3}
0,8	* 2	=	0,6	+	1:	a_{-4}
0,6	* 2	=	0,2	+	1:	a_{-5}



Periodische Zahl!

$$0{,}1_{10} = 0{,}0\overline{0011}_2$$

Übung

- Konvertieren Sie die Dezimalzahl **4,8125** in das **Dualsystem**

Rückkonvertierung der Nachkommastellen

- Einsetzen der Koeffizienten in die allgemeine Gleichung zur Darstellung der Nachkommastellen:

$$Z_n = a_{-1} \cdot B^{-1} + a_{-2} \cdot B^{-2} + \ldots + a_{-(m-1)} \cdot B^{-(m-1)} + a_{-m} \cdot B^{-m}$$
$$= \sum_{i=-m}^{-1} a_i \cdot B^i$$

- Beispiel:

$$0{,}1011_2 = 1*2^{-1} + 0*2^{-2} + 1*2^{-3} + 1*2^{-4}$$
$$= 0{,}5 + 0{,}0 + 0{,}125 + 0{,}0625$$
$$= 0{,}6875_{10}$$

Übung

- Konvertieren Sie die Dualzahl **$10110{,}0111_2$**
  in das **Dezimalsystem**

Maßeinheiten

- 1 Bit = 1 Zeichen (0 oder 1)
- 1 Byte = 8 Bit (256 Zeichen)
- 1 kB = 1000 Byte
- 1 KiB = 1 Kibibyte = 1024 Byte = 2^{10} Byte
- 1 MB = 1000 kB
- 1 MiB = 1 Mebibyte = 1048576 Byte = 2^{10} KiB
- usw.: Gigabyte (GB) und Gibibyte (GiB), Terabyte (TB) und Tebibyte (TiB), Petabyte (PB) und Pebibyte (PiB)...

Andere Zahlensysteme

- Bisher nur zwei Zahlenbasen betrachtet:
  - 10 (Dezimalsystem)
  - 2 (Dualsystem)
- Konvertierungsalgorithmen allgemein anwendbar
- Beispiele: **Oktalsystem**, **Hexadezimalsystem**

Oktalsystem

- **Beispiel:**
- **743 soll im Oktalsystem dargestellt werden**
- $743 \div 8 = 92$ Rest 7: a_0
  $92 \div 8 = 11$ Rest 4: a_1
  $11 \div 8 = 1$ Rest 3: a_2
  $1 \div 8 = 0$ Rest 1: a_3

  $743_{10} = 1347_8$
- **Rückkonvertierung oktal nach dezimal:**

  $1*8^3 + 3*8^2 + 4*8^1 + 7*8^0$

  $= 1*512 + 3*64 + 4*8 + 7*1 = \underline{743}_{10}$

Hexadezimalsystem (1)

- System zur Basis 16

Dezimal-zahl	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hexa-dezimal-ziffer	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Hexadezimalsystem (2)

- **Beispiel:**
- **743 soll im Hexadezimalsystem dargestellt werden**
- $743 \div 16 = 46$ Rest 7: a_0

  $46 \div 16 = 2$ Rest E: a_1

  $2 \div 16 = 0$ Rest 2: a_2

  $743_{10} = 2E7_{16}$
- **Rückkonvertierung hexadezimal nach dezimal:**

  $2*16^2 + E*16^1 + 7*16^0$

  $= 2*256 + E*16 + 7*1 = 2*256 + 14*16 + 7*1 = \underline{743}_{10}$

0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F

Hexadezimalsystem (3)

- Hauptsächlich zur kompakten Darstellung von Binärdaten eingesetzt
  - Z. B. für logische Operationen, Register- und Speicherinhalte
- Beispiel: 32-Bit-Maske für logische UND-Operation:
- **0000 1111 0000 1001 0000 0000 0000 0000_2 =**

  0 F 0 9 0 0 0 0_{16}

Dual ↔ Oktal ↔ Hexadezimal

- **Alle drei Zahlensysteme haben als Basis eine Zweierpotenz**
  - dual: $B=2^1$, oktal: $B=2^3$, hexadezimal: $B=2^4$
- Dadurch einfache Konvertierung zwischen den Systemen möglich
- Beispiele:
- dual ↔ hexadezimal

  0b 0010 1110 0111 (Aufteilen in Vierergruppen)

  0x 2 E 7
- dual ↔ oktal

  0b 001 011 100 111 (Aufteilen in Dreiergruppen)

  0 1 3 4 7

Kennzeichnung von Zahlensystemen

- Binär: 0b
  - Bsp.: **0b**1011100111 oder **%**1011100111
- Hexadezimal: 0x
  - Bsp.: **0x**2E7 oder **0X**2E7 oder **$**2E7 oder **0x**2e7 oder **0X**2e7 oder **$**2e7
- Oktal: 0 (Null)
  - Bsp.: **0**1347
- Dezimal:
  - Bsp.: 743 (keine spezielle Markierung vorangestellt)

Beliebige Basen

- 743 als Zahl zur Basis 7:
- $743 \div 7 = 106$ Rest 1: a_0
  $106 \div 7 = 15$ Rest 1: a_1
  $15 \div 7 = 2$ Rest 1: a_2
  $2 \div 7 = 0$ Rest 2: a_3
  $743_{10} = 2111_7$
- Rückkonvertierung nach dezimal:

  $2*7^3 + 1*7^2 + 1*7^1 + 1*7^0$

  $= 2*343 + 1*49 + 1*7 + 1 = \underline{743}_{10}$

Übung

- **Berechnen Sie die Lösung von**

  $435_7 + 44_5$

  und geben Sie das Ergebnis im Dualsystem und Hexadezimalsystem an

Rechnen mit Dualzahlen

Arithmetik im Dualsystem: Addition

- **Funktioniert im Prinzip wie im Dezimalsystem**
  - 0+0 = 0, 0+1 = 1, 1+0 = 1, 1+1 = 10
  - Bei 1+1 entsteht ein Übertrag (carry bit) von 1
    - Übertrag erfordert zusätzliche Stelle
- Beispiele
- Dezimalsystem:
  5 + 6 = 11

  Dualsystem:

```
   0 1 0 1
+  0 1 1 0
    1          Übertrag
  ---------
   1 0 1 1
```

- Dezimalsystem:
  5 + 12 = 17

  Dualsystem:

```
   0 1 0 1
+  1 1 0 0
  1 1          Übertrag
  ---------
1  0 0 0 1
```

Übung

- Führen Sie die folgende Addition im Dualsystem durch:

$47_{10}+23_{10} = ?_2 + ?_2 = ?_2$

Zahlenkonvertierung: Fortsetzung

- Bisher nur positive Zahlen betrachtet
- Verarbeitung negativer Zahlen ebenfalls erforderlich
- Wie lässt sich -743_{10} als Dualzahl darstellen?
- Darstellung einer Dualzahl mit Vorzeichen nicht möglich, da das interne Alphabet nur aus der Menge B = {0,1} besteht
  - D. h. folgendes ist NICHT möglich: Z = -1011100111_2
- Eine Codierung des Vorzeichens mittels des Zeichenvorrates B = {0,1} wird benötigt!

Negative Zahlen

Darstellung negativer Zahlen

- Naheliegendste Lösung:

  Darstellung des Vorzeichens mittels zusätzlichem Bit am Anfang der Dualzahl

- **Vorzeichen-Betrags-Darstellung**
- Beispiel für dreistellige Dualzahl:

  $4_{10} = 100_2$

- Einführen des zusätzlichen Bits mit

  0: positive Zahl und

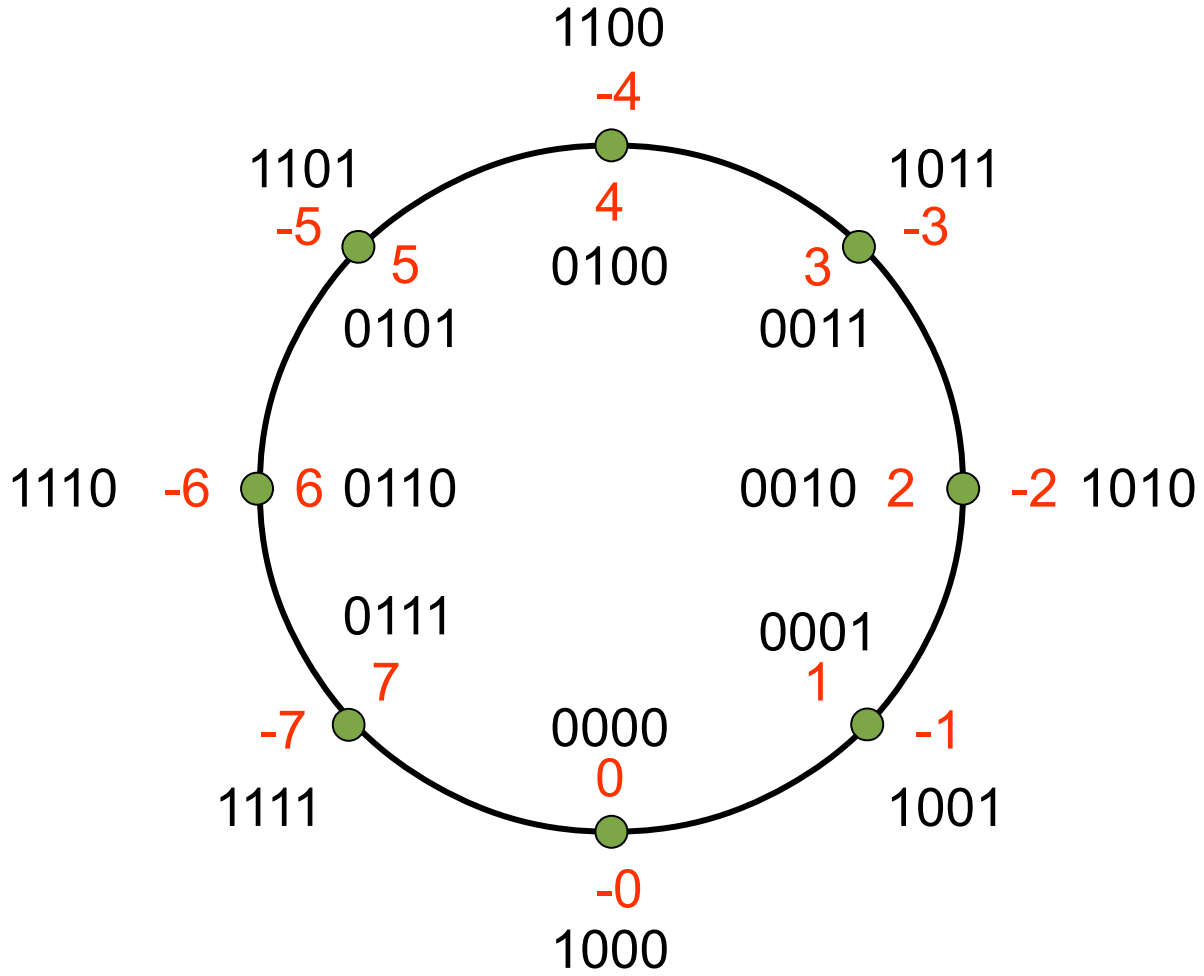
  1: negative Zahl

  ermöglicht:

  $4_{10} = 0100_2$, $-4_{10} = 1100_2$

Zahlenkreis

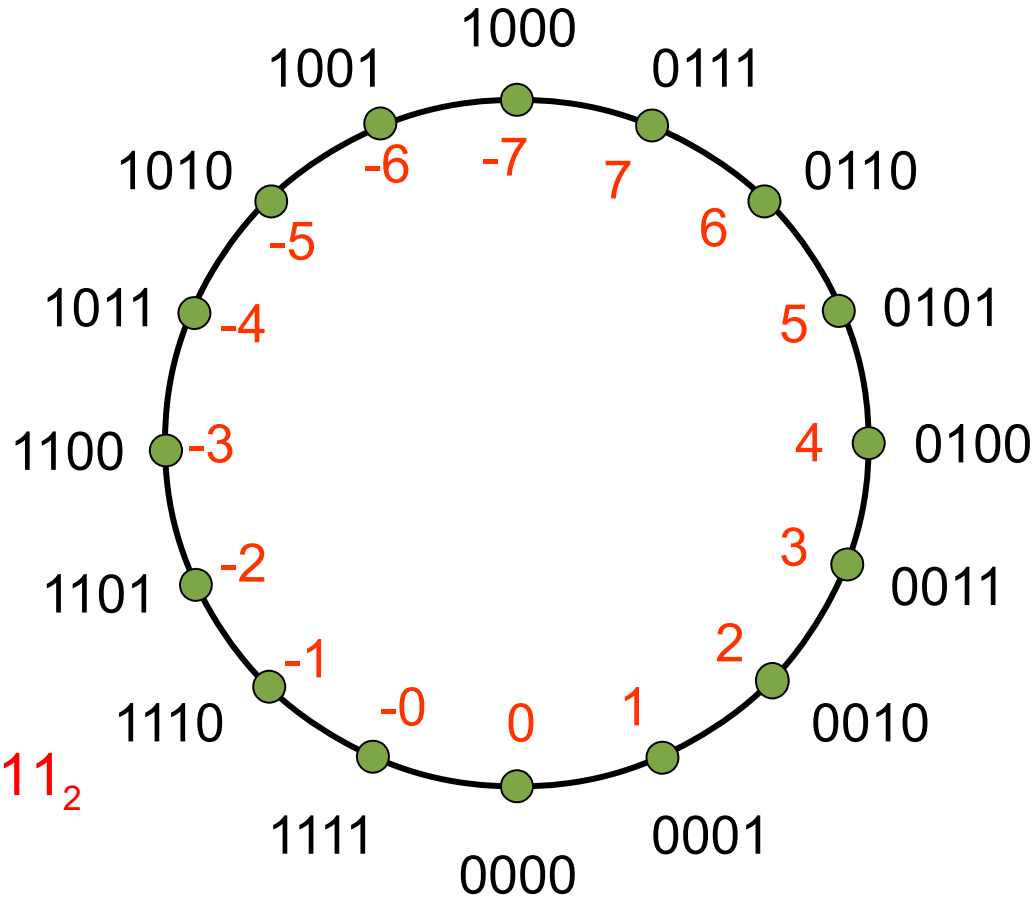
- Ermöglicht Visualisierung einer Zahlendarstellung
- Zahlenkreis für die **Vorzeichen-Betrags-Darstellung**:



- Darstellung ist nicht homogen
  - Vorzeichenstelle erfordert gesonderte Behandlung
- Gut für Multiplikation/ Division geeignet, weniger für Addition/Subtraktion
- -0=1000 ausgeschlossen

1-Komplement-Darstellung

- Erzielt durch Bit-weises Invertieren der positiven Dualzahl
- Höchstwertiges Bit steht auch hier wieder für das Vorzeichen:
  - 0: positive Zahl,
    1: negative Zahl
- Es existiert eine redundante Darstellung der Null!
- Beispiel:
  $4_{10}+(-4_{10}) = 0100_2+1011_2 = 1111_2$



2-Komplement-Darstellung

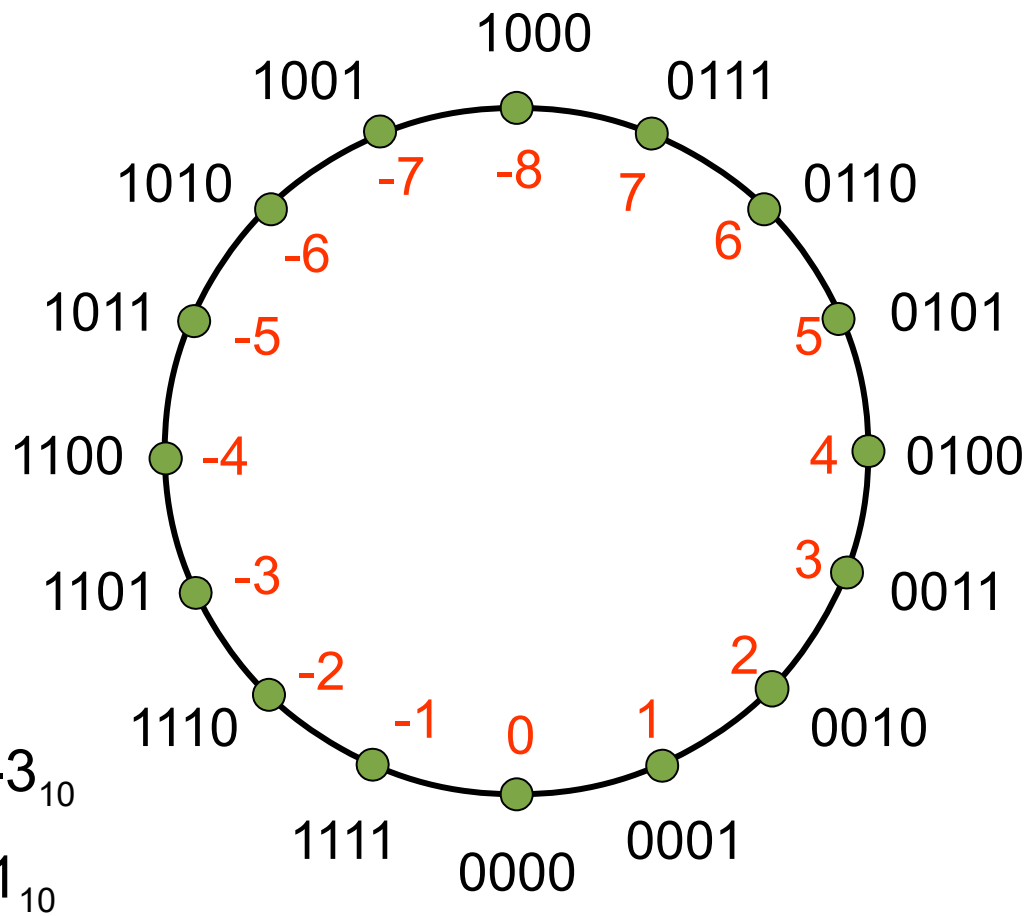
- Erzielt durch Bit-weises Invertieren der positiven Dualzahl mit anschließender Addition von 1
- Einzig gebräuchliche Darstellung negativer Zahlen
- (Two's complement numbers)
- Beispiele:

$4_{10}+(-4_{10}) = 0100_2+1100_2 = 0000_2$

$2_{10}-5_{10} = 0010_2+1011_2 = 1101_2 = -3_{10}$

$7_{10}-6_{10} = 0111_2+1010_2 = 0001_2 = 1_{10}$

$2_{10}+4_{10} = 0010_2+0100_2 = 0110_2 = 6_{10}$



2-Komplement-Bildung

- Invertieren eines jeden Bits und Addieren von Eins
- Beispiel: -743?
- 743_{10} = 0$1011100111_2$

1. Invertierung:	`10100011000`
2. Addition von Eins:	`+ 1`
2-Komplement:	`10100011001 = -743`

- Rückkonvertierung:

1. Invertierung:	`01011100110`
2. Addition von Eins:	`+ 1`
2-Komplement:	`01011100111 = +743`

Übung

- Führen Sie die folgende Subtraktion mittels 2-Komplement-Zahlen durch

  47_{10}-23_{10} = $?_2$

- Geben Sie das Ergebnis als **Dualzahl** an

Arithmetik im Dualsystem: Subtraktion

- In früheren Prozessoren Subtraktion durch Addition des Subtrahenden in 2-Komplement-Darstellung
- In heutigen Prozessoren echte Subtraktion implementiert!
- Beispiel:

	47_{10}	$1\ 0\ 1\ 1\ 1\ 1_2$
-	28_{10}	$-0\ 1\ 1\ 1\ 0\ 0_2$
	1	1
=	19_{10}	$0\ 1\ 0\ 0\ 1\ 1_2$

Arithmetik im Dualsystem: Überlauf (1)

- Prozessoren verarbeiten Daten in festen Wortlängen
  - Z. B. 8, 16, 32, 64 Bits
- Zahlenbereich daher begrenzt
- Überschreitung des Zahlenbereichs muss signalisiert werden
- Bereichsüberschreitung wird als Bit in einem **Prozessorstatusregister** gespeichert und ist per Befehl abfragbar
  - Übertragsbit c (*carry bit*)
- Prozessorstatusregister speichert ebenfalls, ob Ergebnis gleich Null ist
  - Nullbit z (*zero bit*)

Arithmetik im Dualsystem: Überlauf (2)

- **Signalisierung eines Überlaufs:**
  - Überlauf tritt auf bei Übertrag in das höchstwertige Bit oder in die nicht verfügbare Stelle oberhalb des höchstwertigen Bit
  - Kein Überlauf, wenn Übertrag in beide Stellen
- **Beispiele für 4-Bit-Dualzahlen:**
- $7_{10} + 5_{10} =$

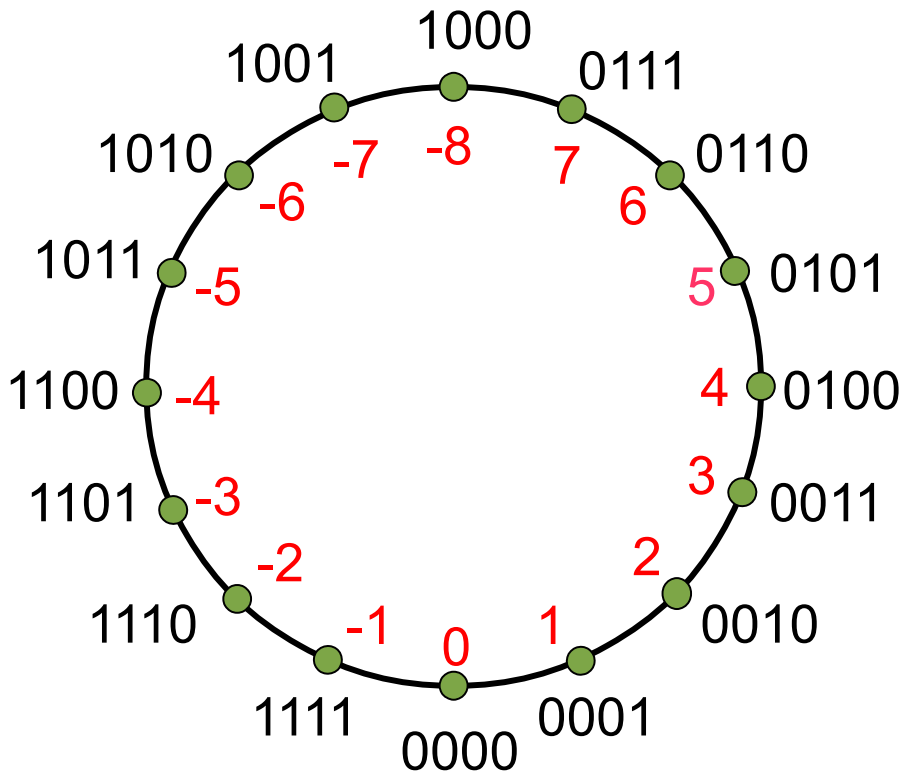
$$\begin{array}{r} 0\ 1\ 1\ 1_2 \\ +\,0\ 1\ 0\ 1_2 \\ \hline 1\ 1\ 0\ 0 = -4_{10} \end{array}$$

Überlauf: 1 1 1

- $5_{10} + (-4)_{10} =$

$$\begin{array}{r} 0\ 1\ 0\ 1_2 \\ +\,1\ 1\ 0\ 0_2 \\ \hline 0\ 0\ 0\ 1 = 1_{10} \end{array}$$

Kein Überlauf: 1 1



Multiplikation mit Dualzahlen

- "Kleines 1-mal-1" für Dualzahlen: 0*0 = 0, 0*1 = 0, 1*0=0, 1*1=1
  - Bei mehrstelligen Zahlen Behandlung entsprechend Dezimalsystem
- Beispiel: $86_{10}*21_{10}$

$1010110_2*10101_2$:

```
          1 0 1 0 1 1 0          10101
+           0 0 0 0 0 0 0          0101
+             1 0 1 0 1 1 0          101
+               0 0 0 0 0 0 0          01
+                 1 0 1 0 1 1 0          1
-------------------------------------------
          1 1 1 0 0 0 0 1 1 1 0₂ = 1806₁₀
```

- Rückführung auf Kopieren, Verschieben und Addieren

Multiplikation mit Dualzahlen: Andere Richtung

- $1010110_2 * 10101_2$:

```
                    1 0 1 0 1 1 0        10101
+                 0 0 0 0 0 0 0          1010
+               1 0 1 0 1 1 0            101
+             0 0 0 0 0 0 0              10
+           1 0 1 0 1 1 0                1
-------------------------------------------
            1 1 1 0 0 0 0 1 1 1 0₂ = 1806₁₀
```

Übung

- Multiplizieren Sie

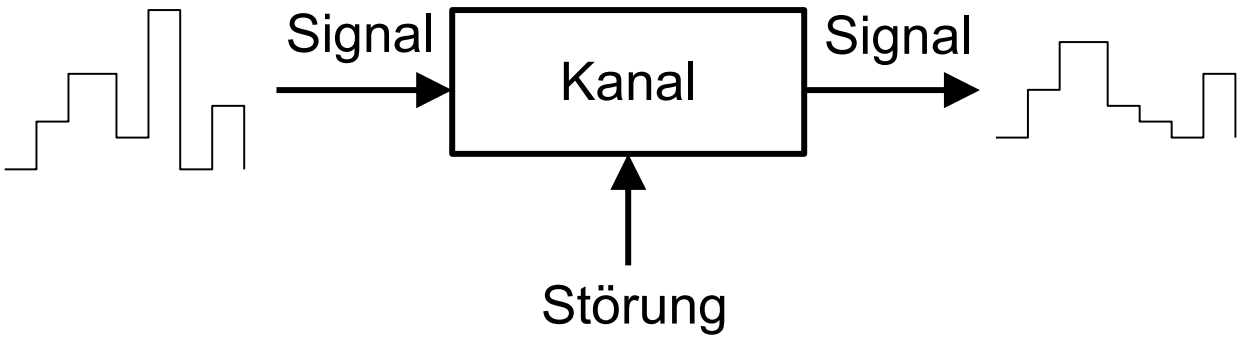
  $13_{10} * 11_{10}$

  im **Dualsystem**

Codesicherung

Codesicherung (1)

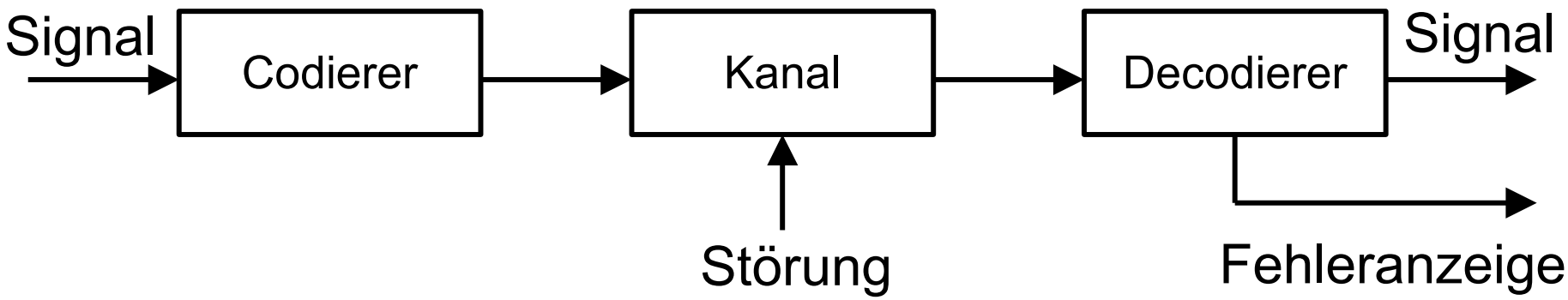
- **Daten bzw. Codewörter können bei der Übertragung und Speicherung verändert werden**
  - Z. B. Digitalfernsehen, Surfen mittels Modem
  - Speichern auf Festplatte, USB-Stick



- **Schutz vor Störungen erforderlich**

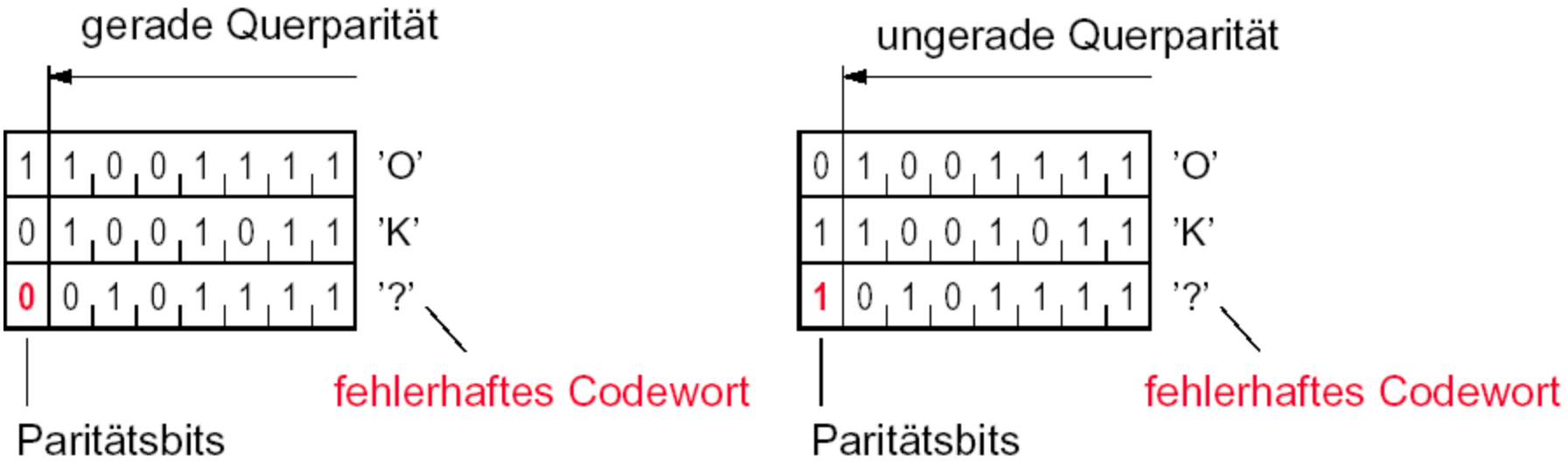
Codesicherung (2)

- Fehlererkennung und Fehlerkorrektur durch redundante Information
- Nutzinformation (*n* Bits) wird vor der Übertragung um Prüfinformation (*p* Bits) ergänzt
- Prüfinformation wird vom Empfänger ausgewertet
- Codewörter der Länge *n* werden auf Codewörter der Länge $m = n + p$ abgebildet



Fehlererkennende Codes (1)

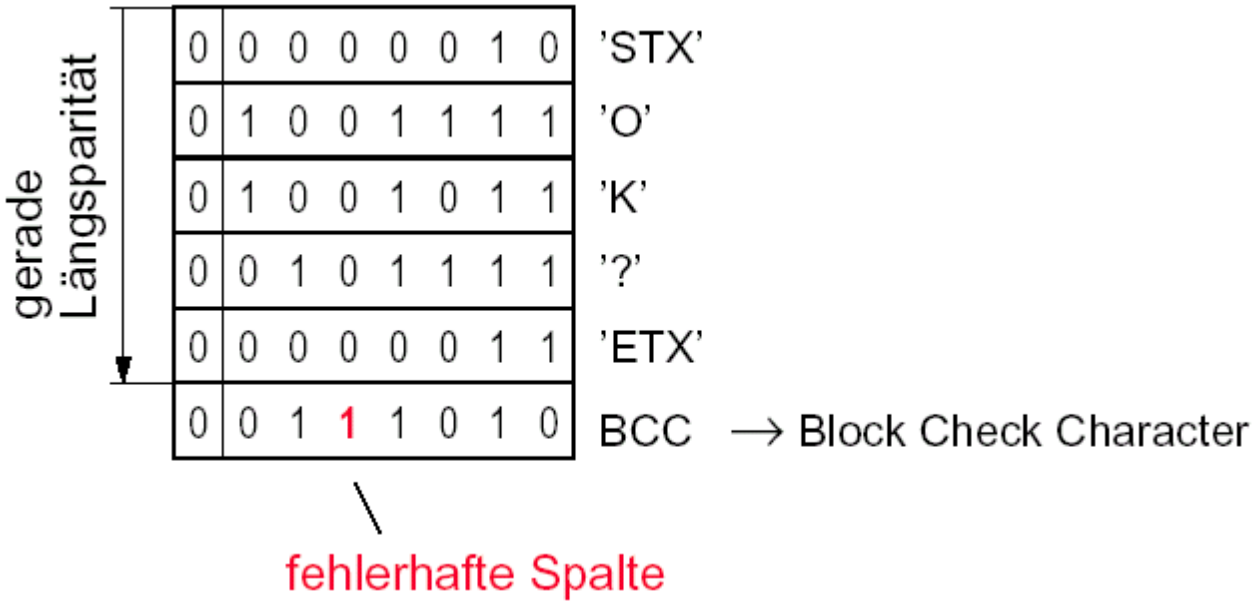
- Quersicherung durch wortweise Paritätsbildung
- Gerade Parität (*even parity*) Ungerade Parität (*odd parity*)



- Bei Fehlererkennung: Wiederholung der Übertragung!

Fehlererkennende Codes (2)

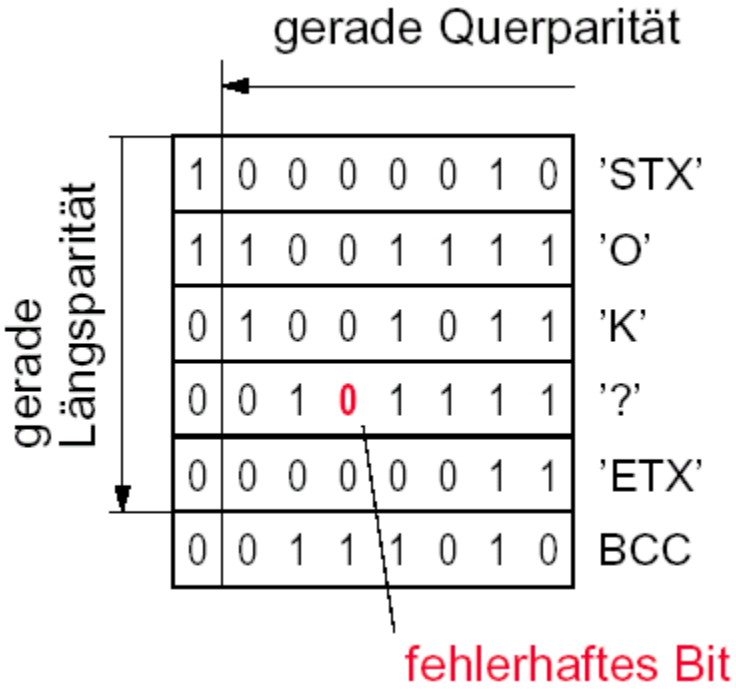
- Längssicherung durch blockweise Paritätsbildung



- Bei Quer- und Längssicherung mit Paritätsbildung
  - 1-Bit-Fehler und Mehr-Bit-Fehler in ungerader Anzahl erkennbar

Fehlerkorrigierende Codes (1)

- **Beispiel:**
  Blockweise Kreuzsicherung
  (Rechtecksicherung)
- Kombination aus Quer- und Längssicherung
- 1-Bit-Fehler sind lokalisierbar und damit korrigierbar
- Mehr-Bit-Fehler sind in gewissem Umfang erkennbar
  - 4-Bit-Rechteckfehler beispielsweise NICHT erkennbar



Übung

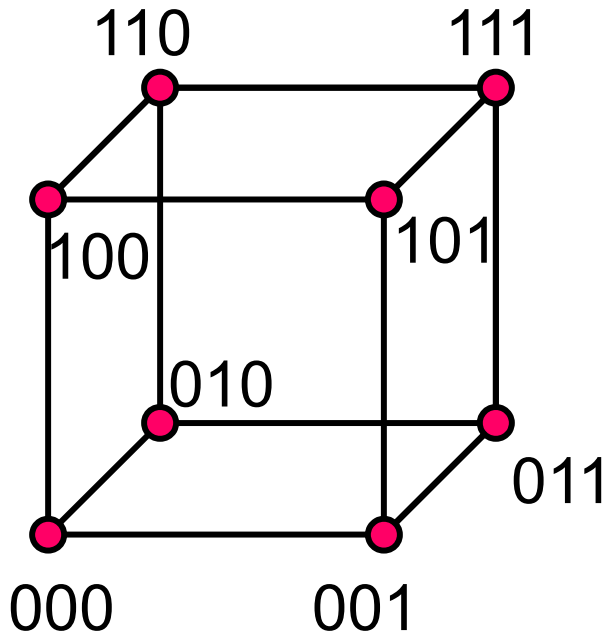
- Gegeben ist ein mit Kreuzsicherung und gerader Parität gesicherter Datenblock. Markieren Sie fehlerhafte Bits und interpretieren Sie den korrigierten Datenblock als ASCII Zeichenfolge.

1	1	0	0	1	0	0	1
1	1	1	0	1	1	1	0
0	1	1	0	0	1	1	0
0	1	1	0	1	1	1	1
1	0	1	0	0	0	0	1
1	0	0	1	1	1	1	1

Bit					00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	HEX
8					0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	
7					0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	
6					0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	
5					0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	
4	3	2	1																		
0	0	0	0	00			SP	0	@	P	`	p			NS	º	À	Đ	à	ð	0
0	0	0	1	01			!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ	1
0	0	1	0	02			"	2	B	R	b	r			¢	²	Â	Ò	â	ò	2
0	0	1	1	03			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó	3
0	1	0	0	04			$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô	4
0	1	0	1	05			%	5	E	U	e	u			¥	µ	Å	Õ	å	õ	5
0	1	1	0	06			&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö	6
0	1	1	1	07			'	7	G	W	g	w			§	·	Ç	×	ç	÷	7
1	0	0	0	08			(	8	H	X	h	x			¨	¸	È	Ø	è	ø	8
1	0	0	1	09			)	9	I	Y	i	y			©	¹	É	Ù	é	ù	9
1	0	1	0	10			*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú	A
1	0	1	1	11			+	;	K	[	k	{			«	»	Ë	Û	ë	û	B
1	1	0	0	12			,	<	L	\	l	|			¬	¼	Ì	Ü	ì	ü	C
1	1	0	1	13			-	=	M	]	m	}			SH	½	Í	Ý	í	ý	D
1	1	1	0	14			.	>	N	^	n	~			®	¾	Î	Þ	î	þ	E
1	1	1	1	15			/	?	O	_	o	DE			¯	¿	Ï	ß	ï	ÿ	F
HEX					0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

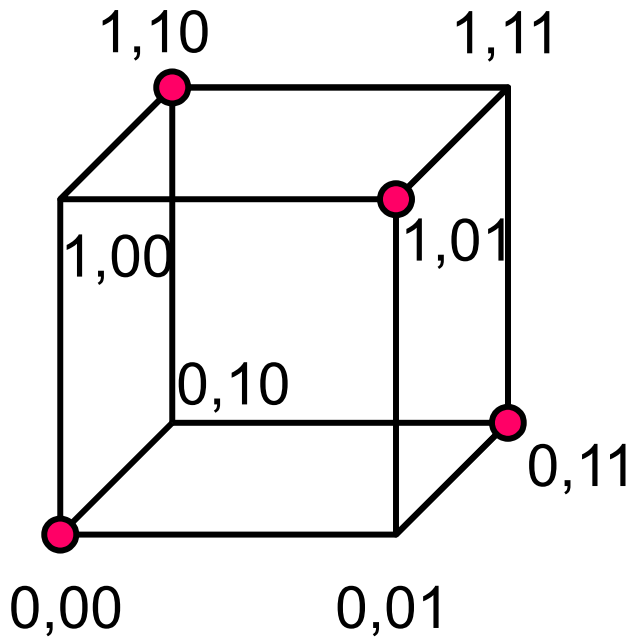
Fehlerkorrigierende Codes (2)

- **Hamming-Distanz**
  - Definiert als die minimale Anzahl unterschiedlicher Bits zweier benachbarter Codewörter
    - Mindestdistanz zweier Codewörter
- **Beispiel: 3-Bit-Code**
  - D. h. Codewörter haben die Länge 3 Bit
  - Mit Nutzinformation $n = 3$ Bits und Prüfinformation $p = 0$ Bits sind Codewörter als Menge der Eckpunkte eines Würfels darstellbar
  - Hamming-Distanz h=1
  - Erhöhung der Mindestdistanz ermöglicht durch Einführen von Redundanz Fehlererkennung oder sogar Fehlerkorrektur



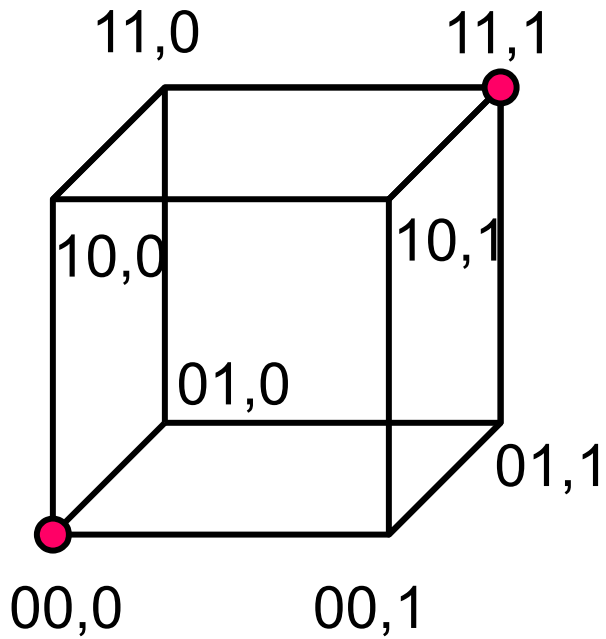
Fehlerkorrigierende Codes (3)

- **Hamming-Distanz h=2**
  - Nutzinformation *n* = *2* Bits und Prüfinformation *p* = 1 Bit
  - Zwischen je zwei Codewörtern liegt 1 ungültiges Codewort
  - Ermöglicht 1-Bit-Fehlererkennung
    - Gültige Codewörter unterscheiden sich in mindestens 2 Bits
      - Abweichung von einem Bit ('Bit-Kipper') automatisch Fehler
  - Darstellung entspricht gerader Parität



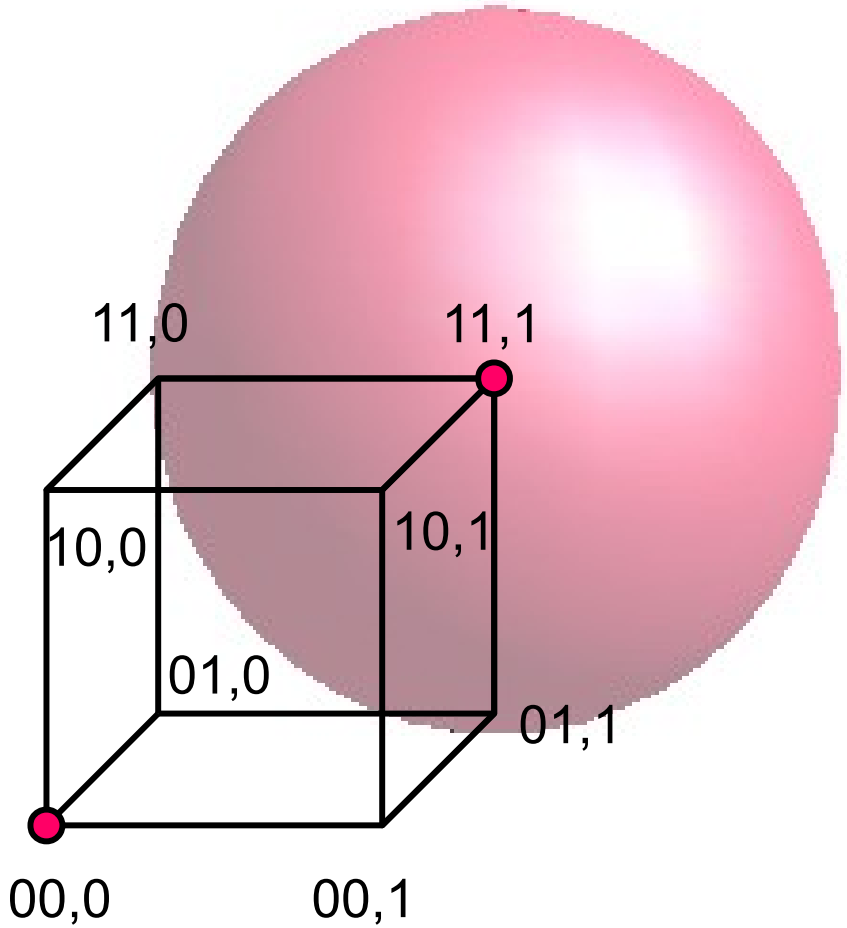
Fehlerkorrigierende Codes (4)

- **Hamming-Distanz h=3**
  - Nutzinformation *n* = 1 Bit und Prüfinformation *p* = 2 Bits
  - Zwischen je zwei Codewörtern liegen 2 ungültige Codewörter
  - Ermöglicht 2-Bit-Fehlererkennung
    - Gültige Codewörter unterscheiden sich in mindestens 3 Bits
      - Abweichungen bis max. 2 Bit automatisch Fehler
    - Ermöglicht 1-Bit-Fehlerkorrektur
      - 1-Bit-Fehler liegen innerhalb der 'Korrekturkugel' des jeweiligen Codeworts



Fehlerkorrigierende Codes (5)

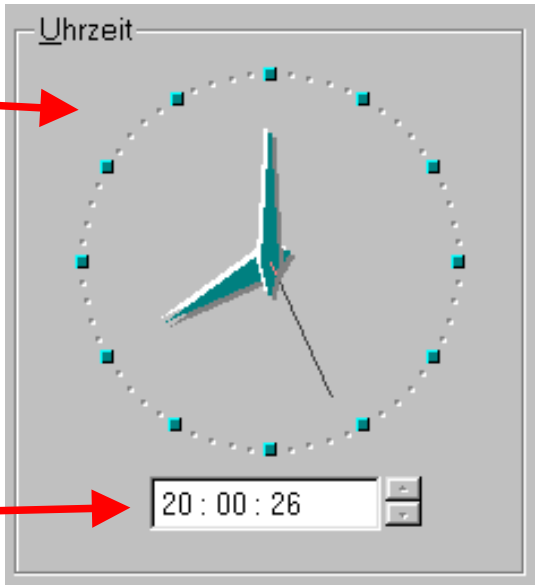
- Hamming-Distanz h=3
- Korrekturkugel



Analoge und digitale Daten

Analoge und digitale Daten

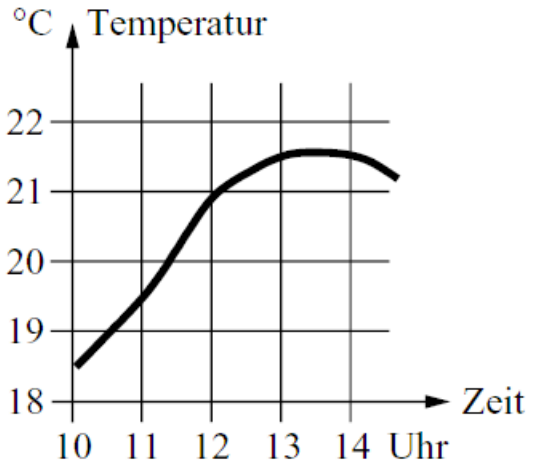
- **Analoge Daten**
  - Analog = entsprechend, vergleichbar
  - Werte ändern sich stufenlos
  - Darstellung durch kontinuierliche Kurve, Skala, Zeigerstellung
  - Beispiel: Zeigerstellung der Uhr



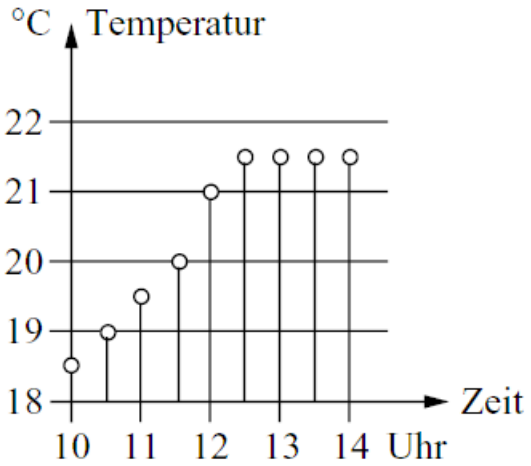
- **Digitale Daten**
  - Digit (engl.) = Zahl
  - Daten werden durch diskrete Zahlenwerte dargestellt
  - Innerhalb des Computers lassen sich Daten nur digital verarbeiten
  - Vor ihrer Verarbeitung müssen analoge Daten **digitalisiert** werden
  - Durch die Digitalisierung geht Information verloren
  - Abstand zwischen Zahlenwerten muss so gering sein, dass Verlust tolerierbar ist

Digitalisierung (1)

- Zeitlich und betragsmäßig kontinuierliche Werte werden durch zeit- und betragsdiskrete Dualzahlen dargestellt
  - Abtastung und Quantisierung
- Beispiel: Temperaturkurve



- Analoge Darstellung:
- Zeit- und wertkontinuierlich



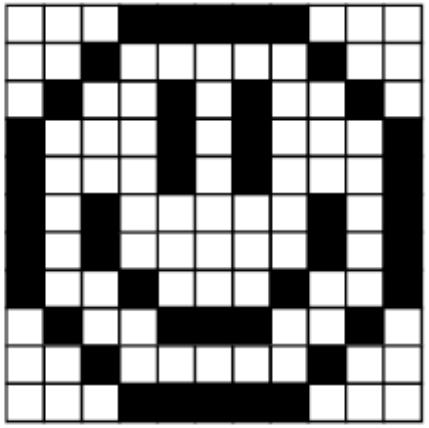
- Diskretisierung:
- Zeit- und wertdiskret
- Zwischenschritt

```
10111001
10111110
11000011
11001000
11010010
11010111
11010111
11010111
11010111
```

- Digitalisierung: Verschlüsselung im Dualsystem

Digitalisierung (2)

- **Beispiel: Schwarz-Weiß-Graphik**



0	0	0	1	1	1	1	1	0	0	0
0	0	1	0	0	0	0	0	1	0	0
0	1	0	0	1	0	1	0	0	1	0
1	0	0	0	1	0	1	0	0	0	1
1	0	0	0	1	0	1	0	0	0	1
1	0	1	0	0	0	0	0	1	0	1
1	0	1	0	0	0	0	0	1	0	1
1	0	0	1	0	0	0	1	0	0	1
0	1	0	0	1	1	1	0	0	1	0
0	0	1	0	0	0	0	0	1	0	0
0	0	0	1	1	1	1	1	0	0	0

  - Analoge Darstellung: Orts- und wertkontinuierlich
  - Diskretisierung: Orts- und wertdiskret
  - Digitalisierung: Verschlüsselung im Dualsystem

- **Farbzeichnungen:**
  - Farbe lässt sich z. B. als RGB-Farbwerte codieren
  - Z. B. ein Byte pro Farbkanal

Aussagenlogik

Aussagenlogik: Wahrheitswerte

- Viele Sätze natürlicher Sprache lassen eine Ja-Nein-Bewertung zu
- Diese Sätze stellen Aussagen dar, die entweder *wahr* oder *falsch* sind
  - Keine syntaktische, sondern eine *semantische* Eigenschaft von Sätzen
- Die beiden möglichen Werte, die einer Aussage zugeordnet werden können, heißen *Wahrheitswerte*

Aussagensätze (1)

- **Beispiele:**

  S1: 0 und 1 sind gleich

  - Diese Aussage ist falsch

  S2: Zwei Ziffern *x* und *y* sind gleich

  - Wahrheitswert abhängig von der Belegung von *x* und *y*

  S3: Zwei Zahlen *X* und *Y* sind gleich

  - Wahrheitswert von der Belegung der jeweiligen Ziffern der Zahlen *X* und *Y* abhängig

  S4: Zwei Zahlen *X* und *Y* sind dann und nur dann gleich, wenn ihre korrespondierenden Ziffern gleich sind

  - Diese Aussage ist wahr, ungeachtet des Wertes der einzelnen Ziffern

Aussagensätze (2)

- **Beispiele:**

  S1: 0 und 1 sind gleich

  S2: Zwei Ziffern *x* und *y* sind gleich

  S3: Zwei Zahlen *X* und *Y* sind gleich

  S4: Zwei Zahlen *X* und *Y* sind dann und nur dann gleich, wenn ihre korrespondierenden Ziffern gleich sind

  - Die Wahrheitswerte der Aussagen S1 und S4 sind unveränderlich und bekannt
  - Die Wahrheitswerte der Sätze S2 und S3 sind *variabel* und folglich unbekannt, solange keine *Variablenwerte* festgelegt worden sind
    - Die Sätze S2 und S3 sind erst dann Aussagen, nachdem die Werte ihrer Variablen festgelegt worden sind

Aussagensätze (3)

- **Analyse von S4:**

  Zwei Zahlen *X* und *Y* sind dann und nur dann gleich, wenn ihre korrespondierenden Ziffern gleich sind

  - S4 kann in zwei einfachere Aussagen, die durch *dann und nur dann* verbunden sind, zerlegt werden
  - Die erste Aussage ist identisch mit S3: Zwei Zahlen *X* und *Y* sind gleich
  - Die zweite Aussage lautet:
  - S5: Die Ziffern in allen Ziffernpaaren gleichen Gewichts sind gleich

Verknüpfung von Aussagen (1)

- S5: Die Ziffern in allen Ziffernpaaren gleichen Gewichts sind gleich
  - Unter der Annahme, dass *X* und *Y* zwei Zahlen zu jeweils *n* Ziffern sind, können wir S5 in weitere *n* Aussagen zerlegen, die durch *und* verknüpft sind
- S5 lautet dann:
  - Die zwei Ziffern in Position 0 sind gleich *und* die zwei Ziffern in Position 1 sind gleich *und*...*und* die zwei Ziffern in Position *n*-1 sind gleich
  - Die gefundenen Aussagen können nicht weiter in einfachere Aussagen zerlegt werden
  - Sie werden deshalb als *atomische* Aussagen bezeichnet

Verknüpfung von Aussagen (2)

- Wir schließen:
  - Aussagen können zur Bildung komplexerer Aussagen verknüpft werden
  - Komplexe Aussagen können in eine Anzahl *atomischer* Aussagen zerlegt werden
  - Aussagen sind entweder wahr oder falsch

Boolsche Algebra (1)

- Entwickelt vom Engländer George Boole (1815-1864)
  - *"An investigation into the laws of thought"*
- **Ziel**: Formale Begründung der Logik
- Aussagenwerte sollten berechenbar werden wie Zahlenwerte z. B. der Addition oder der Multiplikation
- *Boolsche Algebra*: Formalismus zur zweifelsfreien Feststellung der Wahrheit oder Falschheit von Aussagen

Boolsche Algebra (2)

- Die Wahrheit oder Falschheit zusammengesetzter Aussagen kann aus dem *Wahrheitswert* ihrer atomischen Aussagen und den *verknüpfenden Operationen* abgeleitet werden
- Wahrheitswerte sollen hier durch die Dualziffern 0 und 1 ausgedrückt werden
  - 1: wahr
  - 0: falsch
- Logische Operationen, wie z. B. das "Und", sind in der Umgangssprache zu ungenau definiert
  - "Kompensatorisches Und"
- Das Ergebnis logischer Operationen wird daher mittels *Wahrheitstabellen* exakt beschrieben

Logische Operationen

Negation	$\overline{a}$ bzw. $\neg a$	nicht a
Konjunktion	$a \wedge b$	a und b
Disjunktion	$a \vee b$	a oder b
Antivalenz	$a \oplus b$	entweder a oder b
Äquivalenz	$a \Leftrightarrow b$	a genau dann, wenn b
Implikation	$a \Rightarrow b$	wenn a, dann b
Peirce-Funktion (NOR)	$a * b$	weder a noch b
Sheffer-Strich (NAND)	$a \mid b$	nicht a und b

- In einer Implikation wird *a* die *Hypothese* und *b* die *Folgerung* genannt

Wahrheitstabellen

- Zur Erstellung werden alle Kombinationen der Wahrheitswerte der atomischen Aussagen gebildet
- Jeder Kombination wird der Wahrheitswert der logischen Operation zugeordnet

a	b	$a \wedge b$	$a \vee b$	$a \oplus b$	$a \Leftrightarrow b$	$a \Rightarrow b$	a * b	a | b
0	0	0	0	0	1	1	1	1
0	1	0	1	1	0	1	0	1
1	0	0	1	1	0	0	0	1
1	1	1	1	0	1	1	0	0

- Implikation: Eine falsche Hypothese hat keine Bedeutung für die korrekte Folgerung
  - Darum wird in diesem Fall die Implikation als wahr definiert

Ersetzung komplexer Operatoren

- Komplexe Operatoren lassen sich auf die *Konjunktion*, *Disjunktion* und *Negation* zurückführen

a	b	$\overline{a}$	$\overline{b}$	$a \Rightarrow b$	$\overline{a} \vee b$	a*b	$\overline{a \vee b}$	a|b	$\overline{a \wedge b}$
0	0	1	1	1	1	1	1	1	1
0	1	1	0	1	1	0	0	1	1
1	0	0	1	0	0	0	0	1	1
1	1	0	0	1	1	0	0	0	0

- Ausdrücke, in denen nur die Operationen *Konjunktion*, *Disjunktion* und *Negation* vorkommen, werden als *Boolsche Ausdrücke* bezeichnet

Übung

- Zeigen Sie mit Hilfe einer Wahrheitstabelle, dass zwei Ausdrücke dann und nur dann äquivalent sind, wenn sie sich gegenseitig implizieren

- D. h. $(a \Leftrightarrow b) \equiv (a \Rightarrow b) \wedge (b \Rightarrow a)$

a	b	a ∧ b	a ∨ b	a ⊕ b	a ⇔ b	a ⇒ b	a * b	a | b
0	0	0	0	0	1	1	1	1
0	1	0	1	1	0	1	0	1
1	0	0	1	1	0	0	0	1
1	1	1	1	0	1	1	0	0

Sätze zum Umformen in Boolsche Ausdrücke

$$a \Leftrightarrow b \equiv (\bar{a} \wedge \bar{b}) \vee (a \wedge b) \equiv (\bar{a} \vee b) \wedge (a \vee \bar{b})$$

$$a \oplus b \equiv (\bar{a} \wedge b) \vee (a \wedge \bar{b}) \equiv (\bar{a} \vee \bar{b}) \wedge (a \vee b)$$

$$a \Rightarrow b \equiv \bar{a} \vee b$$

$$a * b \equiv \overline{a \vee b}$$

$$a \mid b \equiv \overline{a \wedge b}$$

Logische Ausdrücke

- Definition: Ein logischer Ausdruck entsteht durch endlich oft durchgeführte Verknüpfungen mit Konstanten, Variablen und Ausdrücken
- Die Reihenfolge in der die Verknüpfungen anzuwenden sind, wird durch die Art der jeweiligen Verknüpfung, d. h. des Operators, oder durch Klammern festgelegt
- Das Definieren von Operatorrangfolgen ermöglicht es, viele Klammern in Ausdrücken einzusparen
- Üblich ist folgende Rangfolge: $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$
  - Von links nach rechts, d. h. die Negation $\neg$ bindet am stärksten

Auswertung von Ausdrücken

- 1. Schritt: Niederschreiben aller 0/1-Kombinationen für die Variablen des Ausdrucks

- 2. Schritt: Für jede Kombination wird der Wert entsprechend der Klammerstruktur und dem Rang einer Verknüpfung schrittweise mit Hilfe der Definitionen der einzelnen Verknüpfungen ermittelt

Beispiel

- Zu Prüfen ist die Äquivalenz der beiden Booleschen Ausdrücke $(a \wedge b) \vee c$ und $(a \vee c) \wedge (b \vee c)$

a	b	c	$a \wedge b$	$(a \wedge b) \vee c$	$a \vee c$	$b \vee c$	$(a \vee c) \wedge (b \vee c)$
0	0	0	0	0	0	0	0
0	0	1	0	1	1	1	1
0	1	0	0	0	0	1	0
0	1	1	0	1	1	1	1
1	0	0	0	0	1	0	0
1	0	1	0	1	1	1	1
1	1	0	1	1	1	1	1
1	1	1	1	1	1	1	1

Axiome

- Die folgenden 10 Gleichungen definieren die Boolsche Algebra

1. $a \wedge b \equiv b \wedge a$
2. $a \vee b \equiv b \vee a$
3. $(a \wedge b) \wedge c \equiv a \wedge (b \wedge c)$
4. $(a \vee b) \vee c \equiv a \vee (b \vee c)$
5. $(a \vee b) \wedge c \equiv (a \wedge c) \vee (b \wedge c)$
6. $(a \wedge b) \vee c \equiv (a \vee c) \wedge (b \vee c)$
7. $a \wedge 1 \equiv a$
8. $a \vee 0 \equiv a$
9. $a \wedge \bar{a} \equiv 0$
10. $a \vee \bar{a} \equiv 1$

(1), (2): Kommutativgesetze

(3), (4): Assoziativgesetze
(Besagen, dass die Reihenfolge der Auswertung bei mehrgliedrigen Ausdrücken unwichtig ist)

(5), (6): Distributivgesetze

(Gemischte Klammerausdrücke können „ausmultipliziert“ und „ausaddiert“ werden)

(7), (8): Neutralitätsgesetze

(9), (10): Komplementärgesetze

Sätze (1)

- Die folgenden Gleichungen zur Umformung Boolscher Ausdrücke sind aus den Axiomen ableitbar oder können mittels Wahrheitstabellen gezeigt werden
- Vereinfachung Boolscher Ausdrücke:

11. $a \wedge a \equiv a$
12. $a \vee a \equiv a$
13. $a \wedge 0 \equiv 0$
14. $a \vee 1 \equiv 1$
15. $a \vee (a \wedge b) \equiv a$
16. $a \wedge (a \vee b) \equiv a$
17. $a \vee (\bar{a} \wedge b) \equiv a \vee b$
18. $a \wedge (\bar{a} \vee b) \equiv a \wedge b$

(11), (12): Idempotenzgesetze

(13), (14): Extremalgesetze

(15), (16): Absorptionsgesetze

Beispiel

- Wir zeigen die Gültigkeit von Satz 15: $a \vee (a \wedge b) \equiv a$ mit Hilfe einer Wahrheitstabelle

a	b	a ∧ b	a ∨ (a ∧ b)
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1

Übung

- Zeigen Sie die Gültigkeit von Satz 18: $a \wedge (\bar{a} \vee b) \equiv a \wedge b$ mit Hilfe einer Wahrheitstabelle

a	b	a ∧ b	a ∨ b	a ⊕ b	a ⇔ b	a ⇒ b	a * b	a | b
0	0	0	0	0	1	1	1	1
0	1	0	1	1	0	1	0	1
1	0	0	1	1	0	0	0	1
1	1	1	1	0	1	1	0	0

Sätze (2)

- Sätze für die Negation eines Ausdrucks:

  19. $\overline{\overline{a}} \equiv a$

  20. $\overline{a} \wedge \overline{b} \equiv \overline{(a \vee b)}$

  21. $\overline{a} \vee \overline{b} \equiv \overline{(a \wedge b)}$

- Die Sätze Nr. 20 und 21 werden als *de Morgansche Gesetze* bezeichnet

- Verallgemeinerung von 20 und 21 auf *n* Variablen:

  22. $\overline{x_1 \wedge x_2 \wedge \ldots \wedge x_n} \equiv \overline{x_1} \vee \overline{x_2} \vee \ldots \vee \overline{x_n}$

  23. $\overline{x_1 \vee x_2 \vee \ldots \vee x_n} \equiv \overline{x_1} \wedge \overline{x_2} \wedge \ldots \wedge \overline{x_n}$

Beispiel: Vereinfachung eines Ausdrucks

$$x_1 \vee (\overline{x_1} \wedge x_2) \vee (\overline{x_1} \wedge \overline{x_2} \wedge x_3) \equiv \text{(mit 6.)}$$

$$(x_1 \vee \overline{x_1}) \wedge (x_1 \vee x_2) \vee (\overline{x_1} \wedge \overline{x_2} \wedge x_3) \equiv \text{(mit 10.)}$$

$$1 \wedge (x_1 \vee x_2) \vee (\overline{x_1} \wedge \overline{x_2} \wedge x_3) \equiv \text{(mit 7.)}$$

$$(x_1 \vee x_2) \vee (\overline{x_1} \wedge \overline{x_2} \wedge x_3) \equiv \text{(mit 20.)}$$

$$(x_1 \vee x_2) \vee (\overline{(x_1 \vee x_2)} \wedge x_3) \equiv \text{(mit 6.)}$$

$$(x_1 \vee x_2) \vee \overline{(x_1 \vee x_2)} \wedge ((x_1 \vee x_2) \vee x_3) \equiv \text{(mit 10. und 4.)}$$

$$1 \wedge (x_1 \vee x_2 \vee x_3) \equiv \text{(mit 7.)}$$

$$x_1 \vee x_2 \vee x_3$$

4. $(a \vee b) \vee c \equiv a \vee (b \vee c)$
6. $(a \wedge b) \vee c \equiv (a \vee c) \wedge (b \vee c)$
7. $a \wedge 1 \equiv a$
10. $a \vee \overline{a} \equiv 1$
20. $\overline{a} \wedge \overline{b} \equiv \overline{(a \vee b)}$

Übung

- Vereinfachen Sie den folgenden Ausdruck zu 1:

$$x_1 \vee (\overline{x_1} \wedge x_2) \vee (\overline{x_1} \wedge \overline{x_2})$$

4. $(a \vee b) \vee c \equiv a \vee (b \vee c)$
6. $(a \wedge b) \vee c \equiv (a \vee c) \wedge (b \vee c)$
7. $a \wedge 1 \equiv a$
10. $a \vee \overline{a} \equiv 1$
17. $a \vee (\overline{a} \wedge b) \equiv a \vee b$
20. $\overline{a} \wedge \overline{b} \equiv \overline{(a \vee b)}$

Tautologie und Kontradiktion

- **Definition**: Ausdrücke, die für alle Kombinationen der Wahrheitswerte ihrer Variablen den Wahrheitswert 1 annehmen, heißen allgemein gültig oder **Tautologie**

- **Definition**: Ausdrücke, die für alle Kombinationen der Wahrheitswerte ihrer Variablen den Wahrheitswert 0 annehmen, heißen **Kontradiktion**

Übung

- Überprüfen Sie mittels Wahrheitstabelle, dass der Ausdruck

  $x_1 \vee (\overline{x_1} \wedge x_2) \vee (\overline{x_1} \wedge \overline{x_2})$

  aus der letzten Übung eine **Tautologie** ist

Übung

- Zeigen Sie die Gültigkeit von Satz 18 erneut, diesmal ohne Wahrheitstabellen:

$$a \wedge (\bar{a} \vee b) \equiv a \wedge b$$

1. $a \wedge b \equiv b \wedge a$
2. $a \vee b \equiv b \vee a$
3. $(a \wedge b) \wedge c \equiv a \wedge (b \wedge c)$
4. $(a \vee b) \vee c \equiv a \vee (b \vee c)$
5. $(a \vee b) \wedge c \equiv (a \wedge c) \vee (b \wedge c)$
6. $(a \wedge b) \vee c \equiv (a \vee c) \wedge (b \vee c)$
7. $a \wedge 1 \equiv a$
8. $a \vee 0 \equiv a$
9. $a \wedge \bar{a} \equiv 0$
10. $a \vee \bar{a} \equiv 1$

Dualität (1)

- **Definition**: Zwei Verknüpfungen heißen **dual** (d. h. strukturanalog), wenn ihre Wahrheitstabellen derart zusammenhängen, dass die Wahrheitstabelle der einen Operation entsteht, wenn in der Wahrheitstabelle der dualen Operation 0 und 1 vertauscht werden
- Beispiel: Konjunktion und Disjunktion sind dual zueinander

a	b	$a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

a	b	$a \vee b$
1	1	1
1	0	1
0	1	1
0	0	0

Dualität (2)

- **Definition**: Aus einem Ausdruck entsteht der duale Ausdruck, wenn die Verknüpfungszeichen durch ihre dualen ersetzt sowie 0 und 1 vertauscht werden
- Duale Operationen sind:
  - $\wedge$ und $\vee$
  - $\neg$ und $\neg$
  - * und | (Peirce-Funktion (NOR) und Sheffer-Strich (NAND))
  - $\oplus$ und $\Leftrightarrow$ (Antivalenz und Äquivalenz)
- **Beispiele:**

  $x_1 \vee x_2 \vee \ldots \vee x_n$ *ist dual zu* $x_1 \wedge x_2 \wedge \ldots \wedge x_n$

  $(a \vee \overline{b} \wedge c) \wedge \overline{(c \wedge 1)}$ *ist dual zu* $(a \wedge \overline{b} \vee c) \vee \overline{(c \vee 0)}$

Negation von Ausdrücken

- **Satz**: Ein Ausdruck wird negiert, indem man zum dualen Ausdruck übergeht und jede Variable einzeln negiert.
- Beispiele:

$$\overline{x_1 \vee x_2 \vee \ldots \vee x_n} \equiv \overline{x_1} \wedge \overline{x_2} \wedge \ldots \wedge \overline{x_n}$$

$$\overline{((a \vee \overline{b}) \wedge c) \wedge \overline{(c \wedge 1)}} \equiv ((\overline{a} \wedge b) \vee \overline{c}) \vee \overline{(\overline{c} \vee 0)} \equiv ((\overline{a} \wedge b) \vee \overline{c}) \vee (c \wedge 1)$$

Übung

- Negieren Sie den folgenden Ausdruck:

$$(\bar{a} \vee b) \wedge (\overline{c \wedge 1})$$

Halbaddierer

- Beispiel für die Umsetzung von Rechenoperationen mittels logischer Ausdrücke
- Halbaddierfunktion
- Wir betrachten die Addition von zwei Zahlen

$$X = x_{n-1} \ldots x_1 x_0$$
$$Y = y_{n-1} \ldots y_1 y_0$$

- Ein Übertrag $u_{i+1} = 1$ entsteht dann und nur dann, wenn $x_i=1$ *und* $y_i=1$: $u_{i+1} = x_i \wedge y_i$
- Eine Summenziffer $s_i = 1$ entsteht dann und nur dann, wenn *entweder* $x_i=1$ *oder* $y_i=1$: $s_i = x_i \oplus y_i$

Volladdierer

- Wir betrachten wieder die Addition von zwei Zahlen

  $X = x_{n-1} \ldots x_1 x_0$
  $Y = y_{n-1} \ldots y_1 y_0$

- Ein Übertrag $u_{i+1} = 1$ entsteht dann und nur dann, wenn $x_i = 1$ *und* $y_i = 1$ *oder* $x_i = 1$ *oder* $y_i = 1$ *und* $u_i = 1$:

  $u_{i+1} = (x_i \wedge y_i) \vee (x_i \vee y_i) \wedge u_i$

- Eine Summenziffer $s_i = 1$ entsteht dann und nur dann, wenn *entweder* $x_i = 1$ *oder* $y_i = 1$ *oder* $u_i = 1$:

  $s_i = x_i \oplus y_i \oplus u_i$

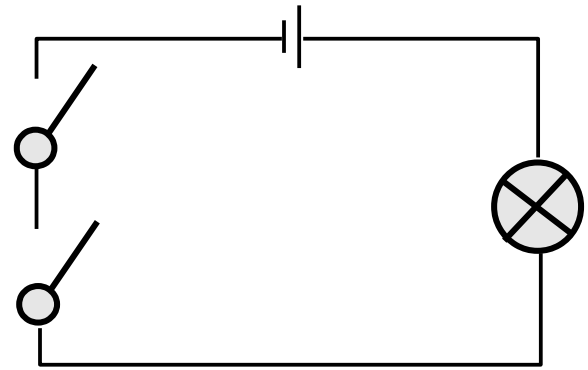
Schalternetze

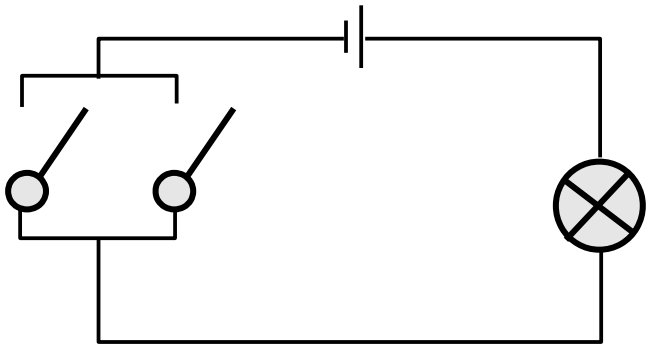
Schalternetze

- Boolesche Funktionen wie der Addierer lassen sich mittels Schalternetzen realisieren
- Hierfür werden atomische Aussagen durch "wahr" und "falsch" Signale und ihre logischen Verknüpfungen durch Schalter repräsentiert
- Eine formalisierte Beschreibung von Schalternetzen führt zur **Schaltalgebra** von Claude E. Shannon (1938)
  - Es lässt sich zeigen, dass die Shannonsche Schaltalgebra und die Boolsche Algebra äquivalent sind

Serien-parallele Schaltungen

- Wird eine Lampe nicht durch einen einzelnen Schalter, sondern durch zwei Schalter eingeschaltet, so ergeben sich zwei mögliche Kombinationen:
- Die Serienschaltung und die Parallelschaltung
- Die **Serienschaltung** steht für das logische **und**
  - **Und-Schaltung**
- Die **Parallelschaltung** steht für das logische **oder**
  - **Oder-Schaltung**



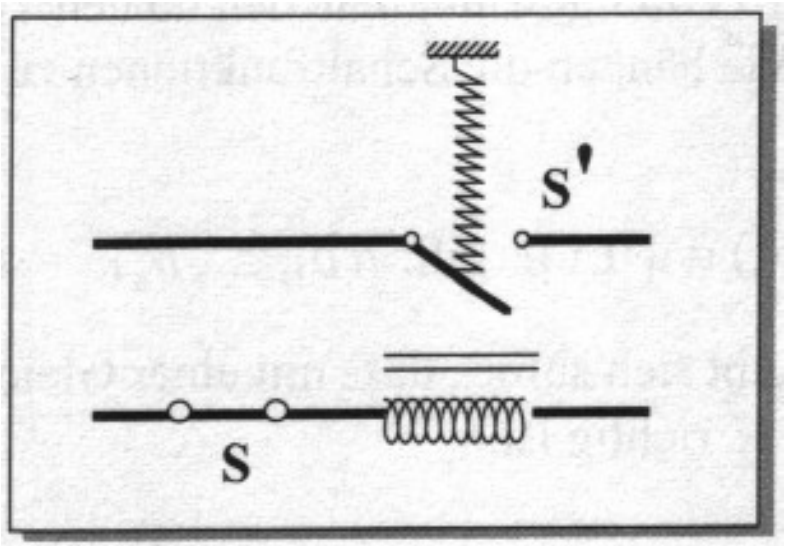


Negation

- Negation nicht durch Serien-Parallel-Schaltung realisierbar
- Wechselschaltung erforderlich
- Definition: Ist *S* ein Schaltglied, so sei *S*' dasjenige Schaltglied, welches genau dann offen ist, wenn *S* geschlossen ist
- *S*' heißt die Negation von *S*

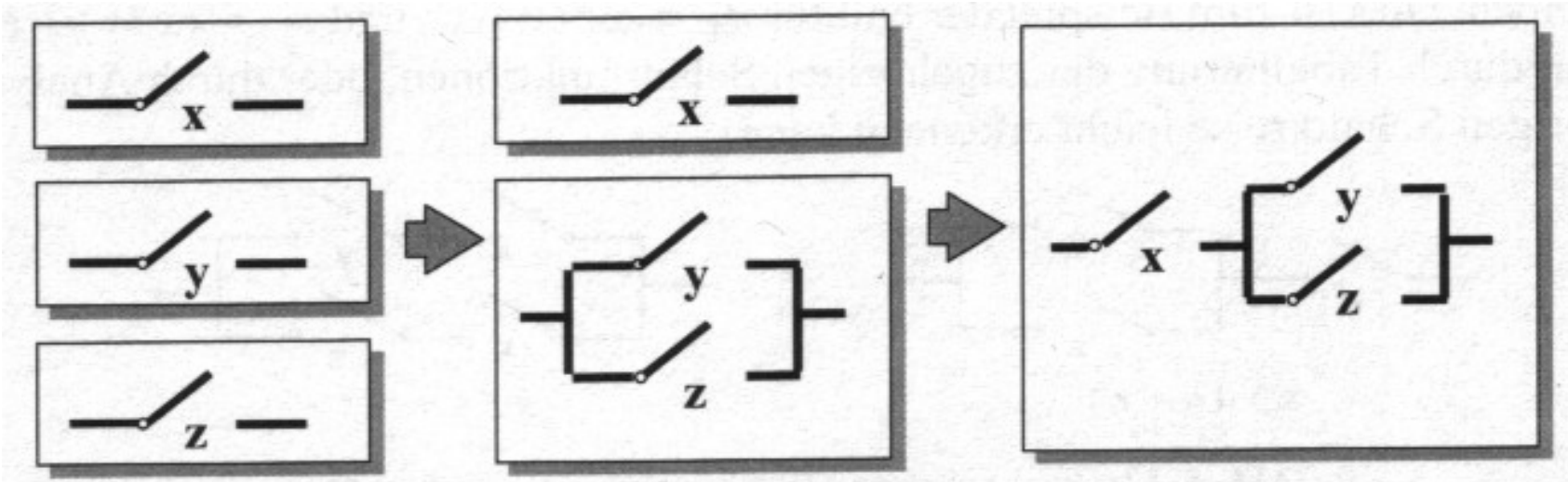
Technische Umsetzung

- Negation in der Frühzeit der Computertechnik durch Relais realisiert
  - Fließt Strom durch die von *S* eingeschaltete Spule, so wird durch die entstehende Magnetkraft der Schalter *S*' gegen die Federkraft geöffnet
- Später: Transistoren



Beispiel

- Realisierung der Booleschen Funktion $x \wedge (y \vee z)$

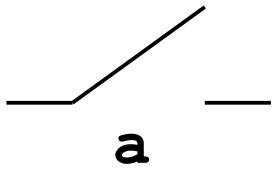


x, y, z $\qquad$ $x, (y \vee z)$ $\qquad$ $x \wedge (y \vee z)$

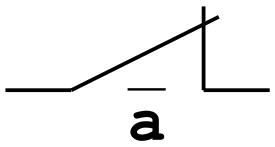
Übung

- Stellen Sie den Halbaddierer als Schalternetz dar

- Mit:



„Schließer“



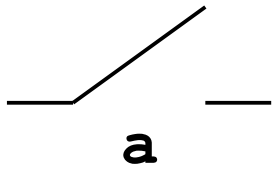
„Öffner“

- Summe: $s_i = x_i \oplus y_i$,
- Übertrag: $u_{i+1} = x_i \wedge y_i$

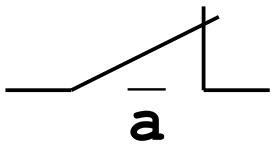
Übung

- Stellen Sie den Volladdierer als Schalternetz dar

- Mit:



„Schließer“



„Öffner“

- Summe: $s_i = x_i \oplus y_i \oplus u_i$,
- Übertrag: $u_{i+1} = (x_i \wedge y_i) \vee (x_i \vee y_i) \wedge u_i$

Von-Neumann-Rechner und Maschinensprache

Beispiel

- Berechnung eines Polynoms durch einen **Menschen** mit Hilfe eines Rechners

$$p = a_0 + a_1 \cdot x + a_2 \cdot x^2 + a_3 \cdot x^3 =$$
$$((a_3 \cdot x + a_2) \cdot x + a_1) \cdot x + a_0$$

mit: $a_3 = 9$, $a_2 = 8$, $a_1 = 7$, $a_0 = 6$ und $x = 3$.

- Unser Rechner besitze ein sog. **Akkumulatorregister**, das auch als Anzeige beim Eingeben und Ausgeben von Zahlen dient
  - Vergleichbar mit der Eingabezeile eines Taschenrechners und der dazu gehörenden Speicherzelle

Operationen

- Unser Rechner kann u. a. folgende Operationen ausführen:

Taste	Variable	Wirkung
↓	M	Lädt Akkumulator mit dem Wert von M
↑	M	Speichert den Inhalt des Akkumulators in M
+	M	Addiert den in M gespeicherten Wert auf den im Akkumulator gespeicherten Wert
*	M	Multipliziert den in M gespeicherten Wert mit dem im Akkumulator gespeicherten Wert
etc.		

Polynomauswertung (1)

- Dem **Bediener** liegt die Lösung der Aufgabe in Form eines **Programms** vor
  - Dies ist ein Zettel mit Anweisungen und notierten Zahlenwerten
- Das Programm besteht aus so genannten „Befehlen“

Polynomauswertung (2)

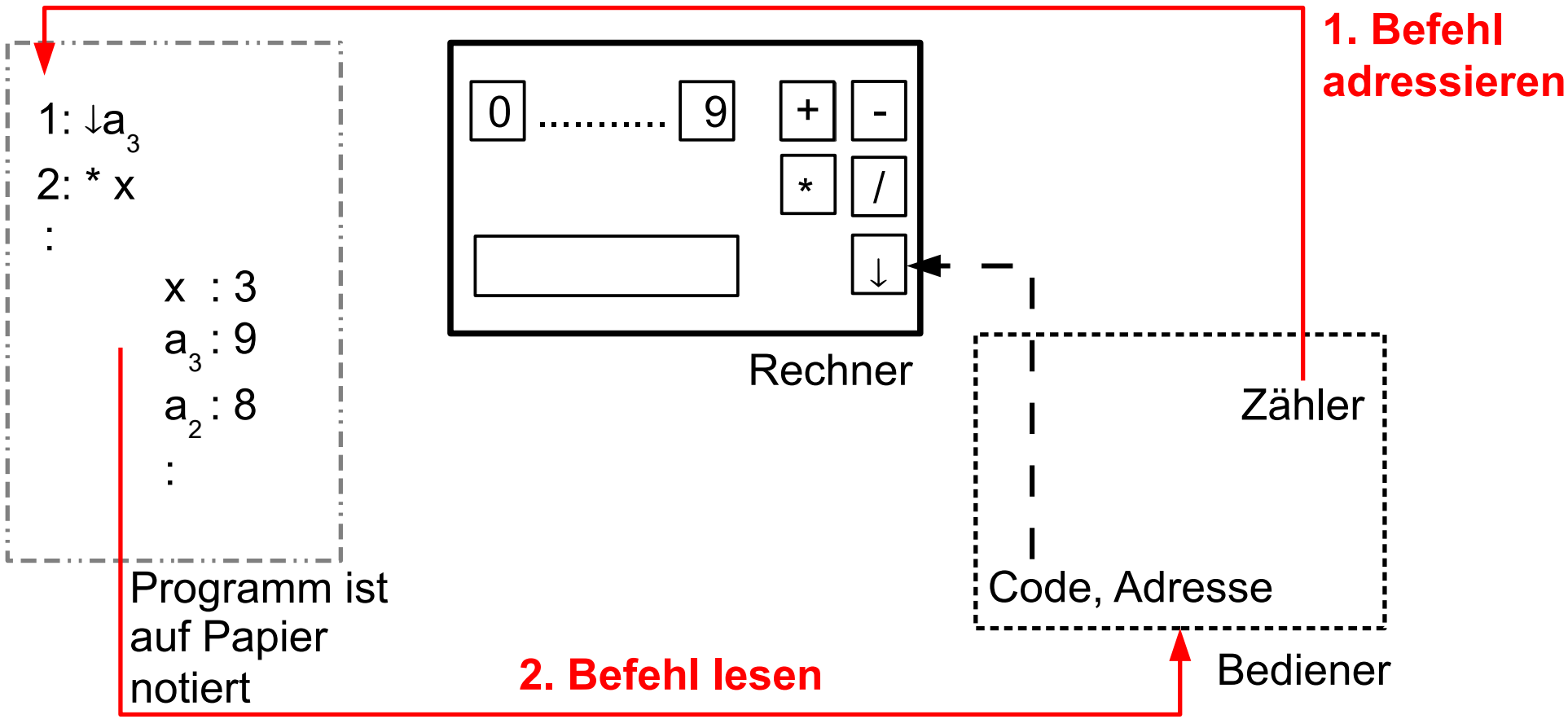
$p = a_0 + a_1 \cdot x + a_2 \cdot x^2 + a_3 \cdot x^3 = ((a_3 \cdot x + a_2) \cdot x + a_1) \cdot x + a_0$

mit: $a_3 = 9$, $a_2 = 8$, $a_1 = 7$, $a_0 = 6$ und $x = 3$

Nr. des Befehls	Art der Operation	Name des beteiligten Operanden	Wert
1	$\downarrow a_3$	a_3	9
2	*x	x	3
3	$+a_2$	a_2	8
4	*x	x	3
5	$+a_1$	a_1	7
6	*x	x	3
7	$+a_0$	a_0	6

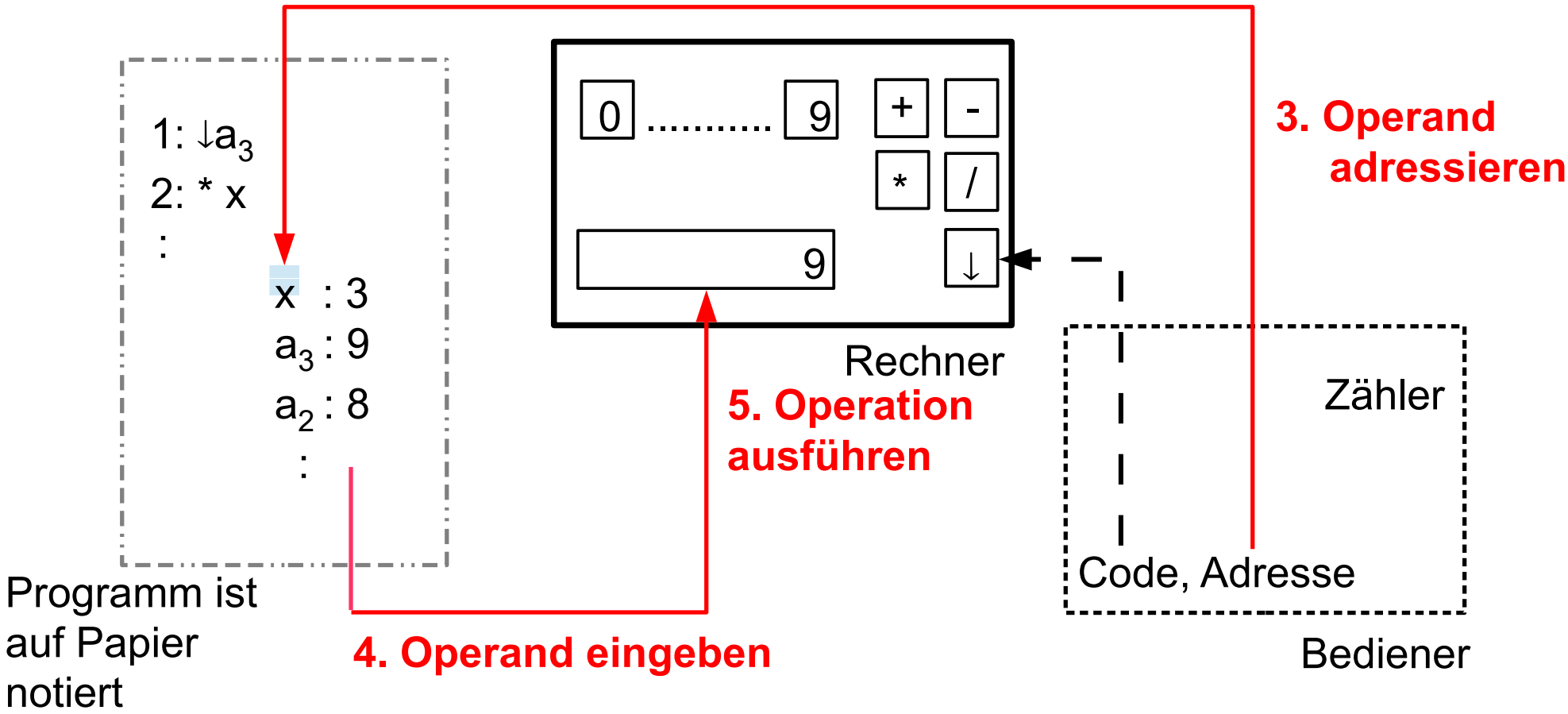
Tätigkeit des Bedieners (1)

- Der Bediener liest den ersten Befehl vom Papier ab und interpretiert ihn
  - Der Befehl ist aufgeteilt in den **Operationscode** (Name der Operation) und in die **Operandenadresse** (Name des Operanden)



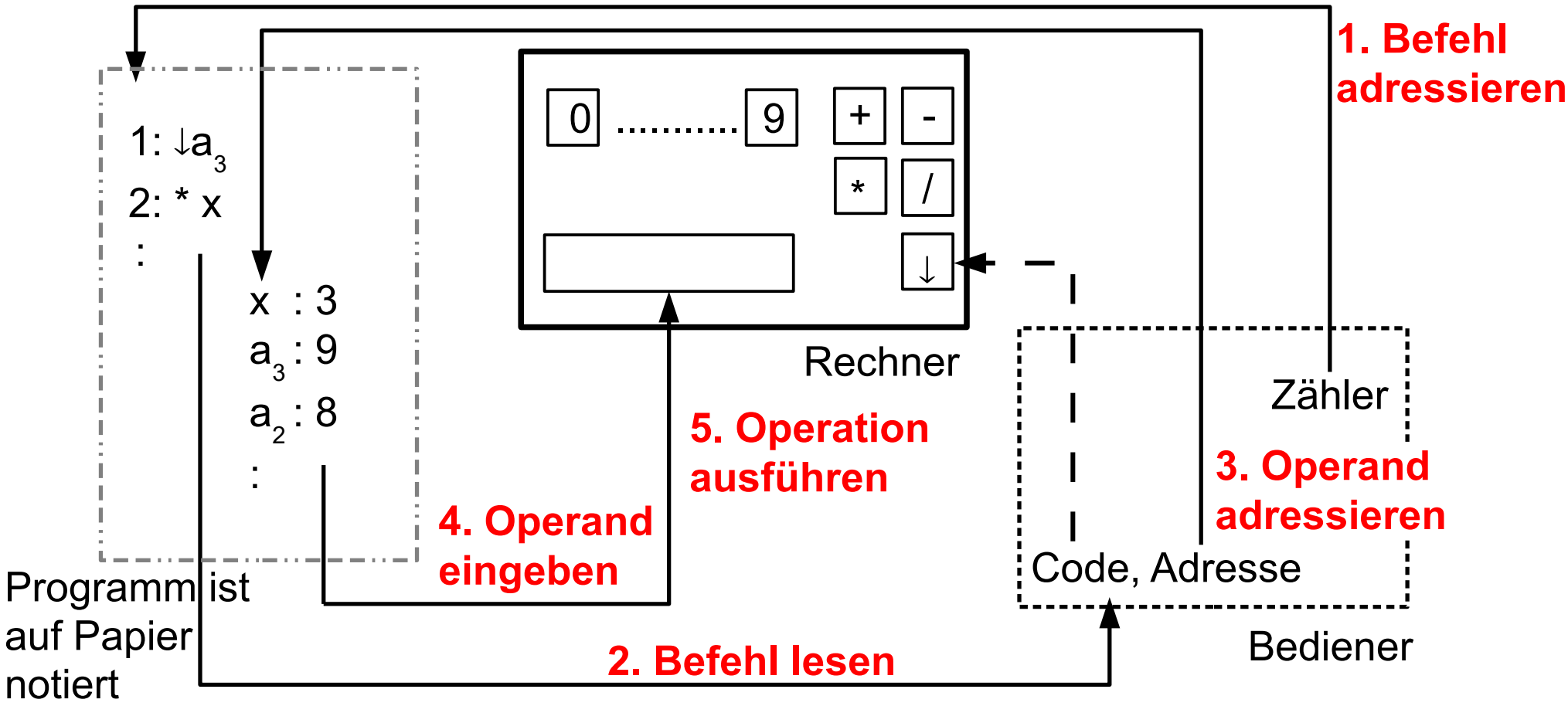
Tätigkeit des Bedieners (2)

- Der Bediener sucht den zur Operandenadresse gehörenden Wert des Operanden
- Wert wird in den Rechner eingegeben und Befehl ausgeführt (Enter-Taste)



Tätigkeit des Bedieners (3)

- Anschließend wird der nächste Befehl bearbeitet
- Nach Bearbeitung eines jeden Befehls wird der sog. **Befehlszähler** um eins hochgezählt (*inkrementiert*)



Von-Neumann-Rechner (1)

- Beim Von-Neumann-Rechner wird die Arbeit des Bedieners vom Rechenautomat übernommen
- An die Stelle des Bedieners tritt ein elektronisches Schaltwerk: das **Steuerwerk**
- Die Speicherung von *Programmbefehlen* und *Daten* erfolgt nicht länger auf Papier, sondern mit Hilfe eines elektronischen Speicherwerks, des so genannten **Hauptspeichers** (auch: **Arbeitsspeicher**)
- Programmbefehle und Daten müssen in den Rechenautomat überführt, d. h. er muss **programmiert** werden

Von-Neumann-Rechner (2)

- Die Befehle und Daten des Programms werden im Rechenautomat als **Binärwörter** gespeichert
- Hierfür ist eine Codierung der Daten und Operationen der Befehle erforderlich
- Die Codierung der Daten erfolgt mit Hilfe von Dualzahlen
- Die Codierung der Operationen mit Hilfe einer so genannten **Codetabelle**

HALT	0001
↑	0101
↓	1000
+	1100
*	1110
⋮	

Beispiel

- Gegeben sei ein Rechner, der über einen Speicher der Wortlänge 16 Bits verfügt
- Sollen die Befehle jeweils in ein Speicherwort passen, so müssen die 16 Bits auf den Operationscode und die Operandenadresse aufgeteilt werden
  - Dies wird als "**Einadressrechner**" bezeichnet
- In unserem Beispiel sei folgende Aufteilung gewählt:
  - 4 Bit für den Operationscode
    - Somit sind $2^4 = 16$ Operationen codierbar
  - 12 Bit für die Operandenadresse
    - Somit $2^{12} = 4096$ verschiedene Operanden adressierbar

Befehlscodes und Codetabelle

- Die möglichen Operationen unseres Rechners seien u. a. durch die folgenden Binärwörter codiert:

Befehlscode	Speicherzelle	Wirkung
0001		Hält den Rechner am Ende des Programms an
0101	M	Speichert den Inhalt des Akkumulators in M
1000	M	Lädt den Akkumulator mit dem Inhalt von M
1100	M	Addiert den Inhalt von M zu dem Inhalt des Akkumulators
1110	M	Multipliziert den Inhalt von M mit dem Inhalt des Akkumulators und speichert das Resultat im Akkumulator
⋮	⋮	⋮

Programm zur Polynomauswertung (1)

$p = ((a_3 \cdot x + a_2) \cdot x + a_1) \cdot x + a_0$

mit: $a_3 = 9$, $a_2 = 8$, $a_1 = 7$, $a_0 = 6$ und $x = 3$

Adresse (1.Teil)	Befehle und Operanden (1.Teil)	Adresse (2.Teil)	Befehle und Operanden (2.Teil)
00000	1000000000001001	01000	0001000000000000
00001	1110000000001101	01001	0000000000001001
00010	1100000000001010	01010	0000000000001000
00011	1110000000001101	01011	0000000000000111
00100	1100000000001011	01100	0000000000000110
00101	1110000000001101	01101	0000000000000011
00110	1100000000001100	01110	0000000000000000
00111	0101000000001110		

Befehls-code	Speicher-zelle	Wirkung
0001		HALT
0101	M	M := A
1000	M	A := M
1100	M	A := A+M
1110	M	A := A*M
⋮		

Programm zur Polynomauswertung (2)

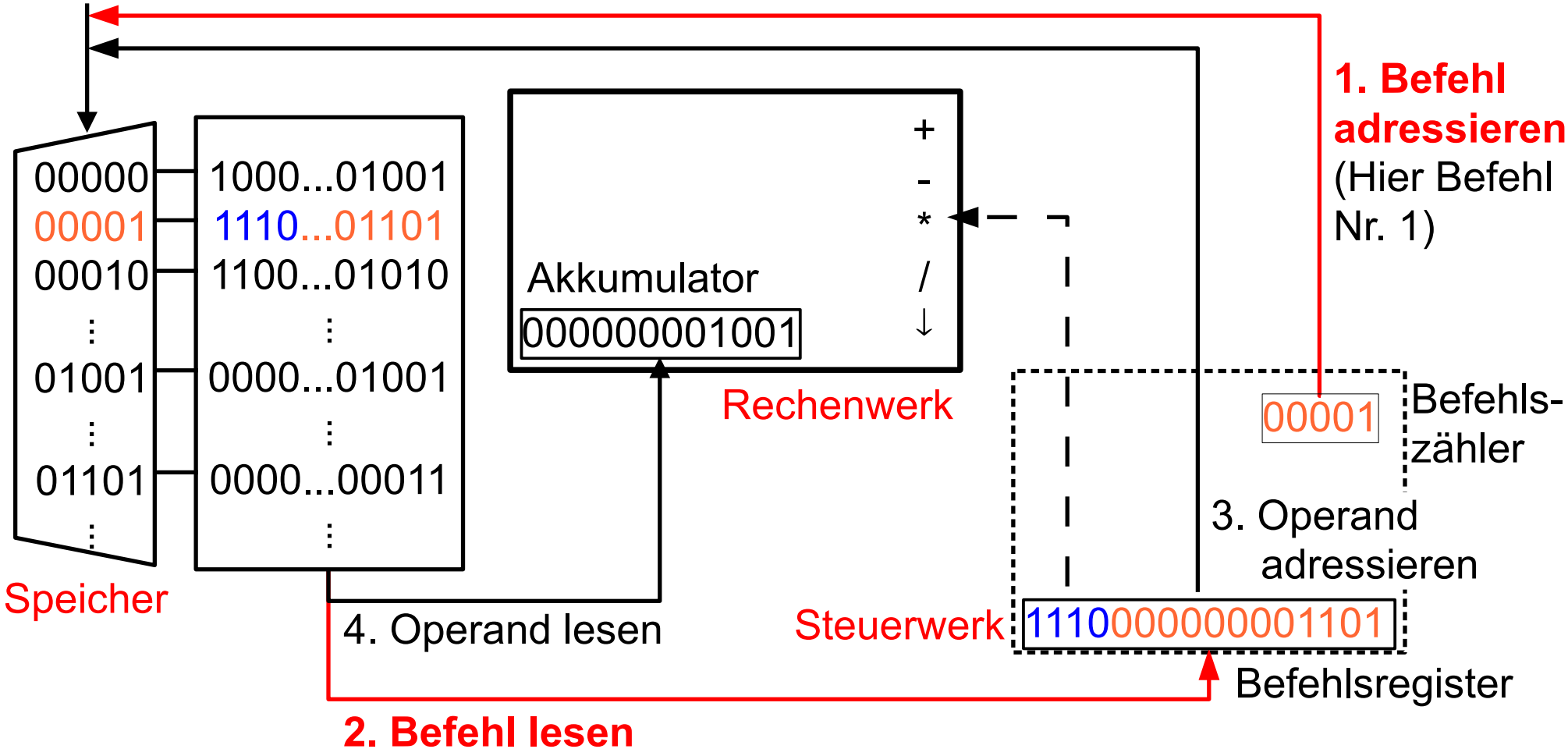
- Das Beispiel zeigt, dass die Befehle und Operanden des Programms als Binärwörter in demselben Speicher abgelegt werden
  - Die Befehle und Operanden sind folglich nur durch ihren Platz im Speicher, d. h. über ihre Adresse, voneinander zu unterscheiden
- Dies ist ein wesentliches Merkmal der **Von-Neumann-Architektur**
- Der abschließende HALT-Befehl im obigen Beispiel verhindert, dass der erste Operand als Befehl interpretiert wird

Das Steuerwerk

- **Ersetzt den Bediener** bei der Durchführung des Programms
- Engl. *Control Unit*, kurz *CU*
- Enthält ein Register zum Zählen der Befehle:
  - Den **Befehlszähler** (*Instruction Pointer* (*IP*))
  - Dieser erlaubt die Identifikation des nächsten auszuführenden Befehls
- Das Steuerwerk besitzt ebenfalls ein **Register für die Zwischenspeicherung von Befehlen**:
  - Das **Befehlsregister** (*Instruction Register* (*IR*))
  - Dieses erlaubt das Zerlegen des Befehls in den Operationscode und die Operandenadresse

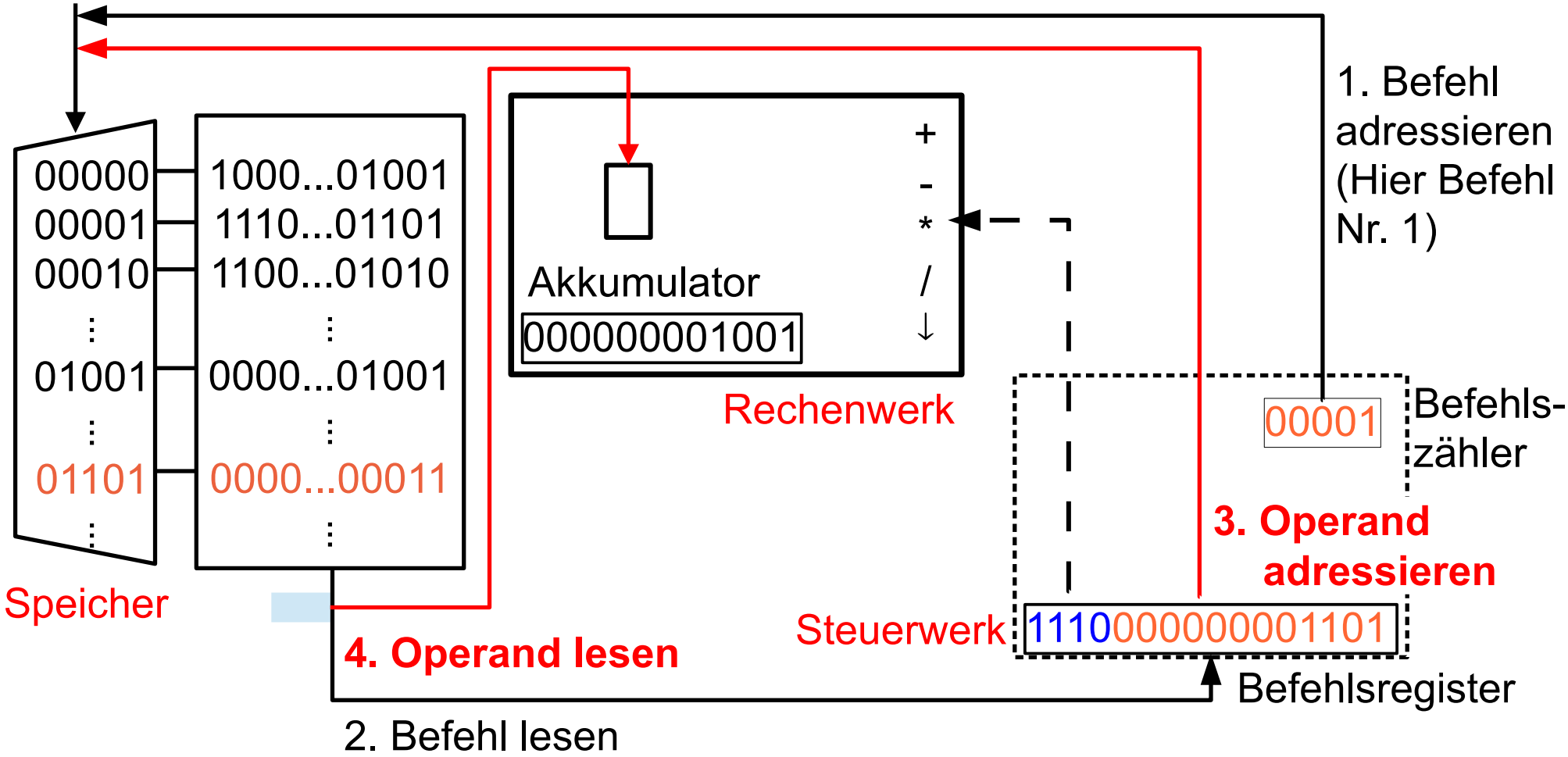
Die Funktionsweise des Steuerwerks (1)

- Der nächste **Befehl** wird aus dem **Speicher** gelesen, in das **Befehlsregister** geladen und decodiert



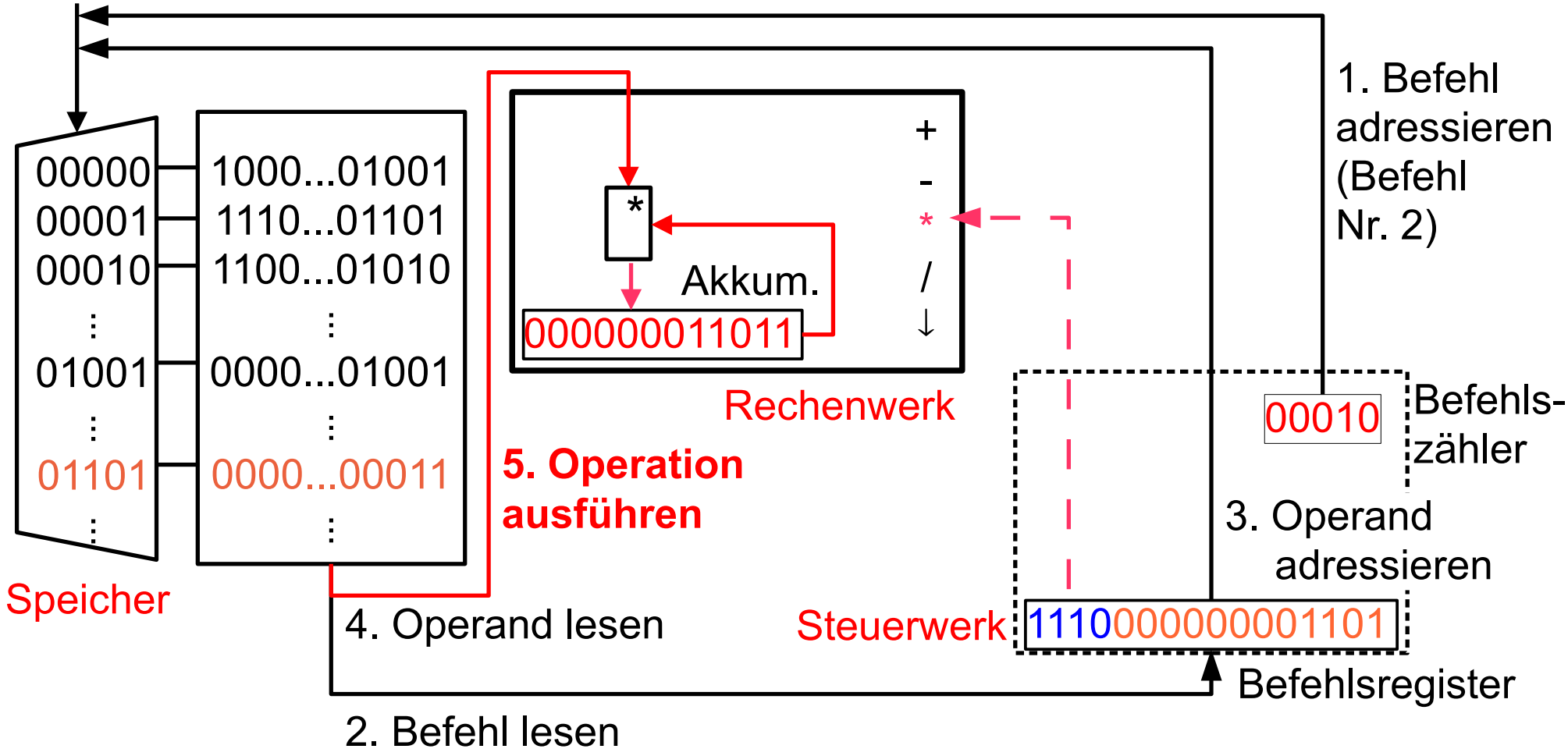
Die Funktionsweise des Steuerwerks (2)

- Der **Operand** wird aus dem Speicher gelesen



Die Funktionsweise des Steuerwerks (3)

- Die **Operation** wird ausgeführt (hier: A ← A*M)
- Der **Befehlszähler** wird danach **inkrementiert**



Programmierung

- Ein für den Menschen verständliches Programm besteht aus mit Hilfe von Schriftzeichen codierten Befehlen, Variablen, Konstanten oder Zahlen

- Ein für den Rechner verständliches Programm besteht aus Binärwörtern für die Operationen und Dualzahlen für die Werte der Operanden und Adressen

- Es ist daher erforderlich, dass für den Menschen verständliche Programm mittels **Codierung** in ein für die Maschine interpretierbares Programm in **Maschinensprache** zu übersetzen

Beispiel

Für den Menschen verständliche Form		Für die Maschine verständliche Form	
0	↓a3	00000	1000000000001001
1	*x	00001	1110000000001101
2	+a2	00010	1100000000001010
3	*x	00011	1110000000001101
4	+a1	00100	1100000000001011
5	*x	00101	1110000000001101
6	+a0	00110	1100000000001100
7	↑p	00111	0101000000001110
8	HALT	01000	0001000000000000
a3	9	01001	0000000000001001
a2	8	01010	0000000000001000
a1	7	01011	0000000000000111
⋮			

Codierung (1)

- Die Codierung erfolgt in zwei Arbeitsgängen
- 1. Arbeitsgang:
  - Nummerierung der Zeilen des Programms sowie der Variablen
    - ⇒ Jede Zeilennummer entspricht dann der Adresse einer Speicherzelle

Programm		Adresse	
0	↓a3	00000	0
1	*x	00001	1
2	+a2	00010	2
3	*x	00011	3
4	+a1	00100	4
5	*x	00101	5
6	+a0	00110	6
7	↑p	00111	7
8	HALT	01000	8
a3	9	01001	9
a2	8	01010	10
a1	7	01011	11
⋮			

Codierung (2)

- Auf diese Weise werden Variablennamen mit Speicherzellen bzw. deren Adressen assoziiert
  - Die Speicheradressen werden als Dualzahl geschrieben

Programm		Adresse	
0	↓a3	00000	0
1	*x	00001	1
2	+a2	00010	2
3	*x	00011	3
4	+a1	00100	4
5	*x	00101	5
6	+a0	00110	6
7	↑p	00111	7
8	HALT	01000	8
a3	9	**01001**	9
a2	8	**01010**	10
a1	7	**01011**	11
⋮			

Codierung (3)

- Die Zuordnungen zwischen den symbolischen Adressen und den dualen Speicheradressen werden danach in der sog. **Symboltabelle** erfasst

a_3	01001
a_2	01010
a_1	01011
a_0	01100
x	01101
p	01110

Codierung (4)

- 2. Arbeitsgang:
  - **Codierung der Befehle**
    - Der Programmierer ordnet mittels der **Codetabelle** → den symbolischen Operationscodes binäre Operationscodes zu
    - Danach werden die symbolischen Adressen (Variablennamen) in den Befehlen mit Hilfe der **Symboltabelle** → in binäre Adressen übersetzt
  - **Codierung der Daten**
    - Schließlich werden dezimale Zahlenwerte in Dualzahlen umgewandelt

HALT	0001
↑	0101
↓	1000
+	1100
*	1110
⋮	

a_3	01001
a_2	01010
a_1	01011
a_0	01100
x	01101
p	01110

Übung

- Gegeben sei ein Einadressrechner mit Befehlssyntax wie zuvor definiert
  - ↓ : Lade Akk. mit Wert von M, ↑: Speichere Akk. nach M,
  - + : Addiere M zu Akk., *: Multipliziere Akk. mit M
- Schreiben Sie den Ausdruck **a1*x + a2*y + a3** in der aus der Rangordnung der Operationen resultierenden Reihenfolge als Programm für diesen Rechner auf
- Der Rechner besitzt eine Speicherzelle M
  - Speichern Sie Zwischenergebnisse und das Resultat in der Speicherzelle M
- Verwenden Sie folgende Variablenwerte:
  - $a1=27_{10}$, $a2=10_{10}$, $a3=3_{10}$, $x=5_{10}$, $y=2_{10}$

Übung

- Der Rechner aus der vorigen Übung sei ein 8-Bit-Einadressrechner mit binären Befehlscodes, entsprechend der nebenstehenden Codetabelle und folgender Aufteilung der Befehle:
  - 4-Bit-Befehlscode
  - 4-Bit-Operandenadresse
  - Die erste Adresse im Hauptspeicher sei 0000
- Übersetzen Sie das Programm aus der letzten Übung in die Maschinensprache des Rechners

HALT	0001
↑	0101
↓	1000
+	1100
*	1110

Maschinelle Codierung (1)

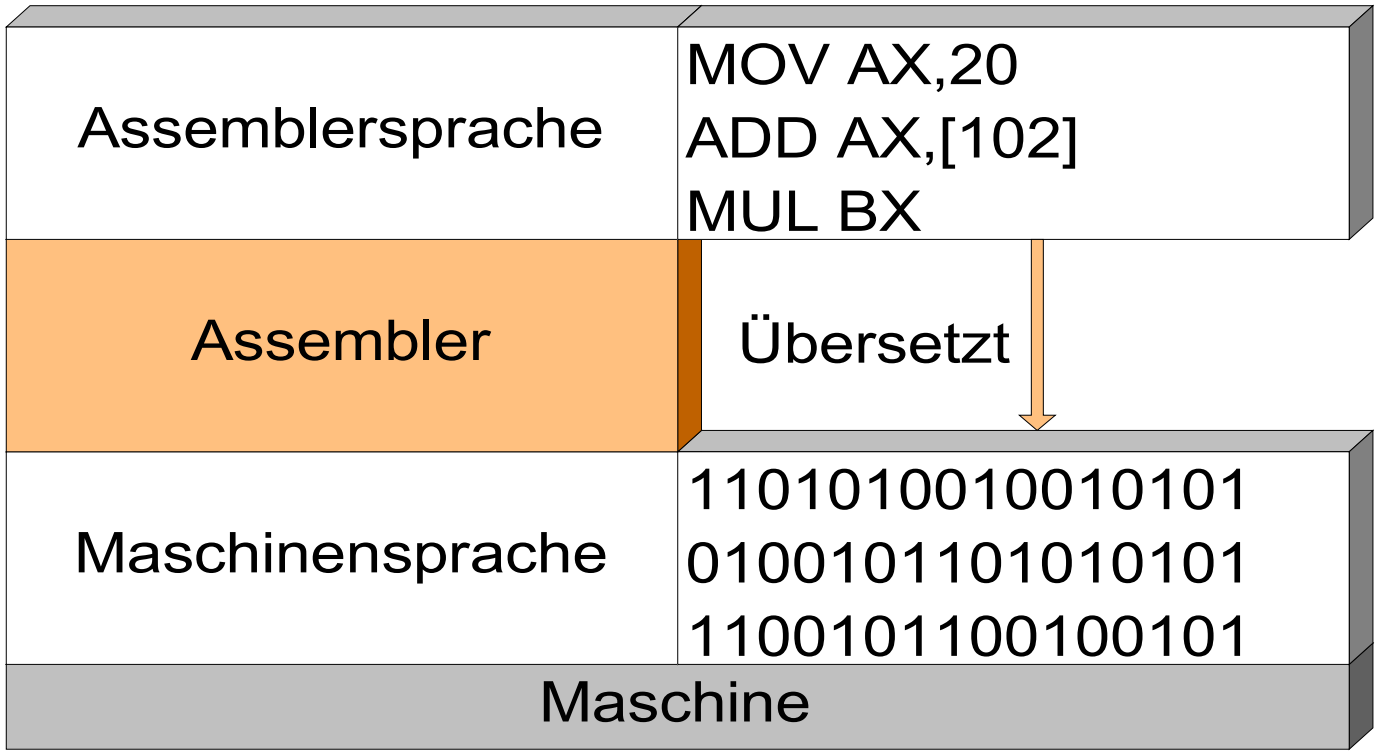
- Der oben beschriebene Codierungsvorgang für die Überführung eines Programms in die Maschinensprache folgt festen Regeln
  - Er ist somit durch einen **Algorithmus** beschreibbar und damit **automatisierbar**
- Der Codierungsvorgang kann daher mit Hilfe eines Programms vom Rechner selbst erledigt und der Mensch von dieser Tätigkeit befreit werden

Maschinelle Codierung (2)

- Ein Programm, welches dies tut, wird als **Assembler** bezeichnet
- Die für den Menschen verständliche Darstellung der Maschinensprache wird deshalb als **Assemblersprache** bezeichnet
- Ein Programm für die Umformung von Maschinensprache in Assemblersprache wird als **Disassembler** bezeichnet

Assemblerprogrammierung

- Problemlösung wird in Assemblersprache formuliert
  - Assembler übersetzt Programm (Übersetzungseinheit, Modul) in Maschinensprache



Assembler

- Assembler für neue Prozessoren werden zunächst auf einem schon vorhandenen Rechnertyp entwickelt
- Dabei entsteht ein **Cross-Assembler**
  - Assembler der Maschinencode für einen anderen Prozessor generiert
- Cross-Assembler übersetzt Code des Assemblers für den Zielprozessor
- Danach steht Assembler als Programm auf dem neuen Rechner zur Verfügung
- Frei für den PC verfügbare Assembler:
  - MASM: Microsoft Macro Assembler
  - NASM: Netwide Assembler

Assemblerprogrammierung (1)

- Erlaubt unmittelbaren Zugang zur Hardware
  - Mithin eine sehr maschinennahe Programmierung
    - Z. B. um einen Treiber für eine Grafikkarte zu programmieren oder die USB-Schnittstelle anzusprechen
    - Maschinennahe Programmierung ist in Hochsprachen, wie z. B. Java, oftmals kompliziert oder sogar unmöglich

Assemblerprogrammierung (2)

- Gestattet die **Erzeugung sehr effizienten Codes**
  - Wichtig für zeitkritische Anwendungen
  - Z. B. für Multimedia-Encoder und -Decoder
- Übersetzungsprogramme für Hochsprachen (Compiler) optimieren heutzutage
  - Die Bedeutung der Assemblerprogrammierung nimmt daher in diesem Zusammenhang ab
- Assemblerprogramme sind schwer portierbar, da sie eine bestimmte Rechnerarchitektur voraussetzen

Assemblerprogrammierung (3)

- Assemblerprogrammierung beruht zu einem wesentlichen Teil auf der Arbeit mit **Registern**
  - Siehe **Akkumulatorregister** oben
- Register sind sehr schnelle **Hilfsspeicherzellen**, die direkt mit dem Rechenwerk verbunden sind
  - Moderne Prozessoren verfügen über mehr als ein Register
- Rechenwerk wird heutzutage als **ALU** bezeichnet
  - ALU: *Arithmetical Logical Unit*
  - Führt die arithmetischen und logischen Operationen durch
- Die ALU und die Register sind wesentliche Bestandteile der *Central Processing Unit* (**CPU**)

Hochsprachenprogrammierung

Hochsprache

- Hochsprachen erlauben eine Abstraktion von Registerarithmetik und Speicherzellen
- Die Abstraktion wird durch das Einführen von Datentypen ermöglicht
- Diese gestatten eine
  - automatische Bereitstellung einer ausreichenden Menge von Speicherzellen für die zu speichernden Daten sowie eine
  - automatische Auswertung von Operationsergebnissen
- Datentypen vereinfachen die Programmierung daher erheblich

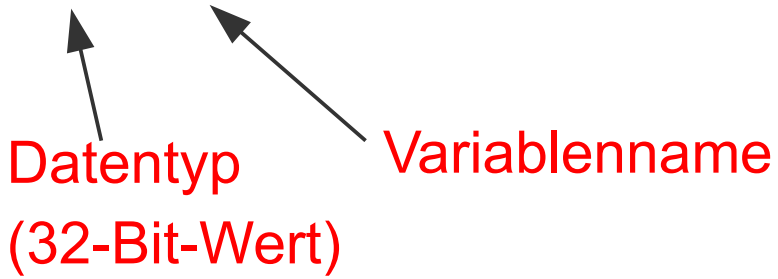
Assemblersprache versus Hochsprache

- Assemblerprogrammierung:
  - Der Programmierer muss selber wissen, wie ein Datenwert zu interpretieren ist
  - Alle Daten werden in derselben Weise gespeichert: als Binärwörter
- Hochsprache:
  - Der Programmierer **definiert** durch Zuweisen eines Datentyps, wie ein Datenwert zu interpretieren ist
  - Der Datenwert erhält für die Maschine eine Bedeutung
  - Die Maschine übernimmt dann die Interpretation des Datenwerts und überprüft dessen korrekten Gebrauch
  - Auf diese Weise nimmt die Maschine dem Programmierer Arbeit ab

Zuweisen eines Datentyps

- Das Speichern von Zahlenwerten erfolgt nicht mehr in Registern und Speicherzellen, sondern in **Variablen**
- Variablen besitzen einen Namen und einen Datentyp
- Beispiel:

```
int i;
```



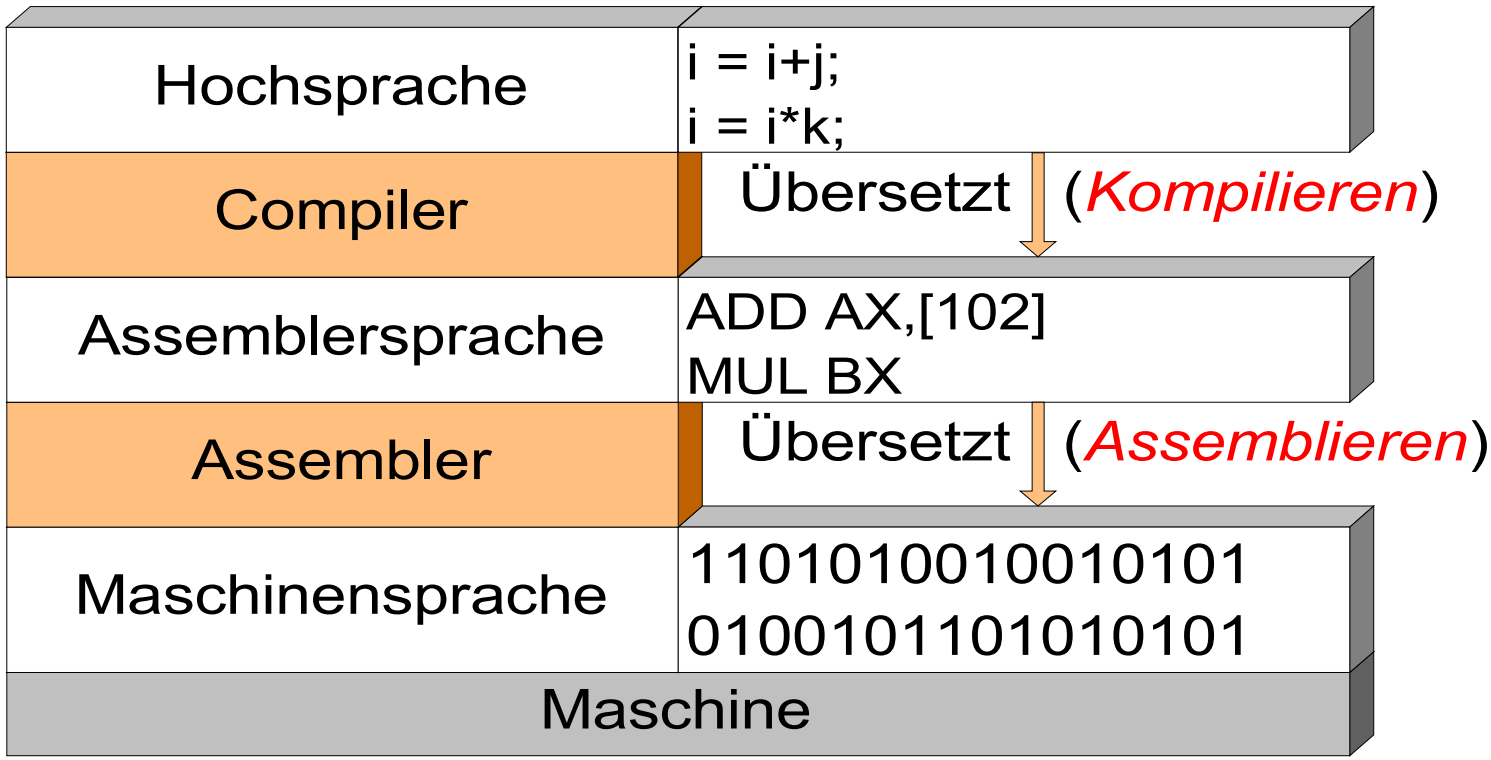
- Die Variable `i` wird hier als Container für einen ganzzahligen Wert vereinbart

Speicherort eines Variablenwerts

- Während des Programmablaufes kann sich der Wert von `i` im Speicher oder in einem Register befinden
  - Der genaue Ort wird vom Übersetzungsprogramm der Hochsprache verwaltet
- Der Programmierer braucht sich hierum nicht zu kümmern

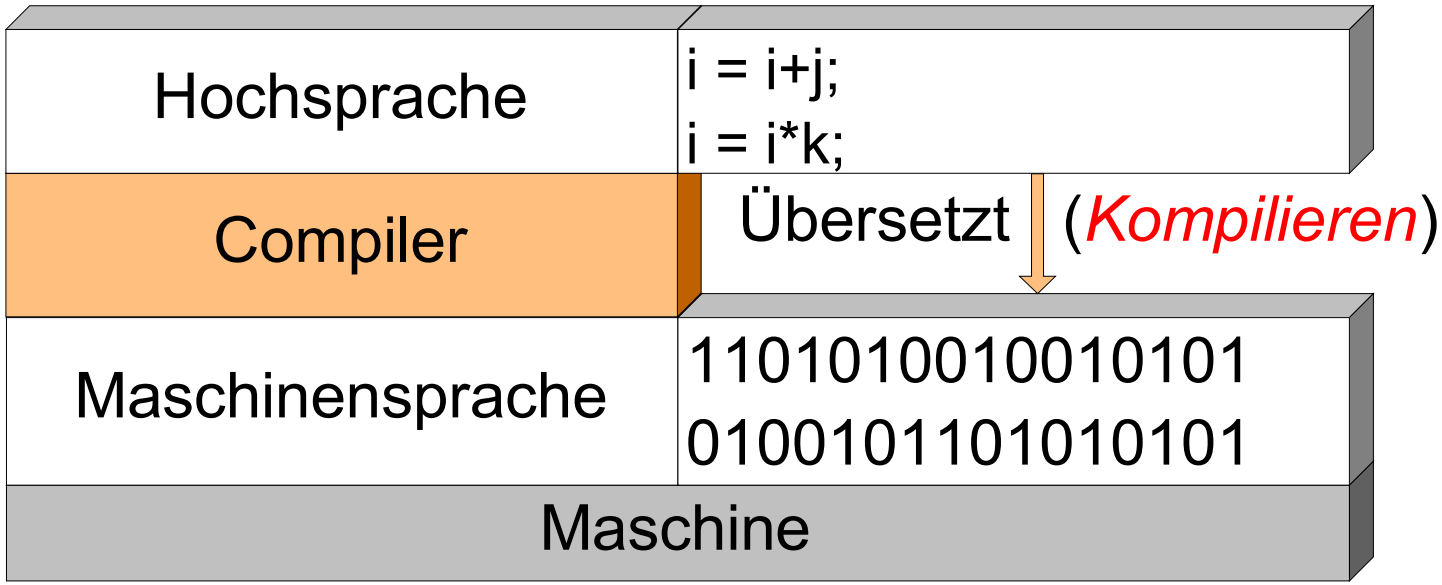
Hochsprachenprogrammierung (1)

- Jedes Hochsprachenprogramm ist zunächst eine reine Textdatei, die übersetzt werden muss
- Die Übersetzung erfolgt durch ein spezielles Programm: den **Compiler**
  - Der Compiler überprüft das Programm ebenfalls auf **syntaktische Korrektheit**



Hochsprachenprogrammierung (2)

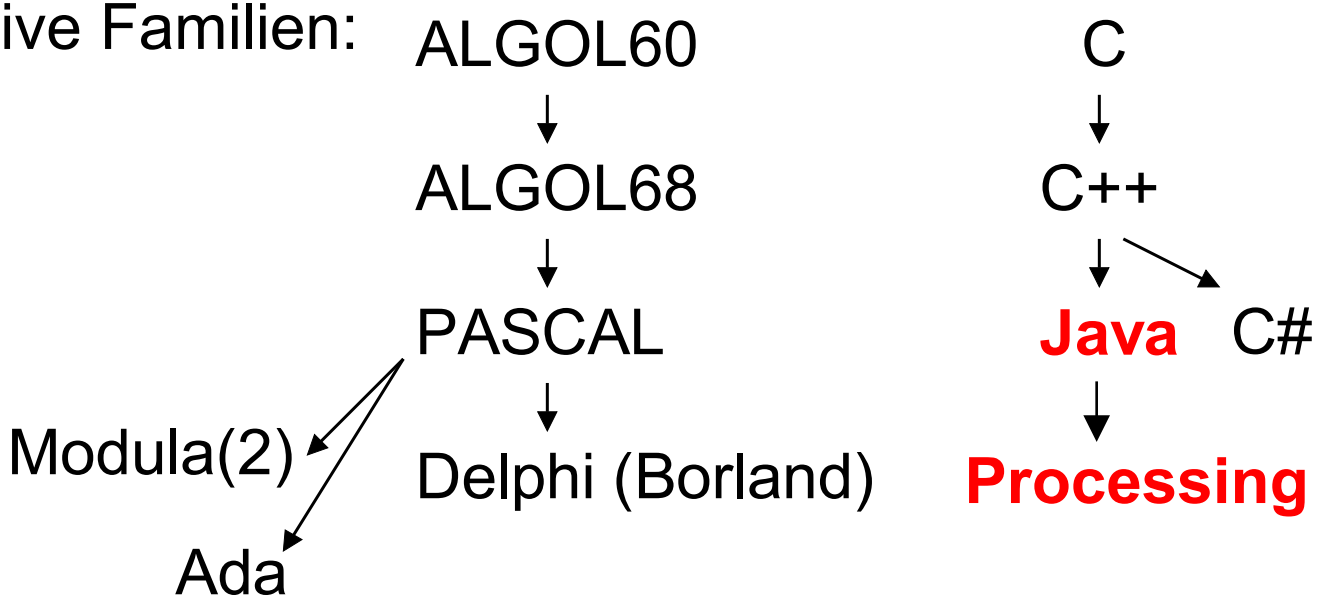
- Die Kompilier- und Assemblierschritte werden heutzutage zumeist zu einem Schritt zusammengefasst
  - Die Kompilierung erfolgt häufig transparent im Hintergrund einer Entwicklungsumgebung



Programmierung

Programmiersprachen

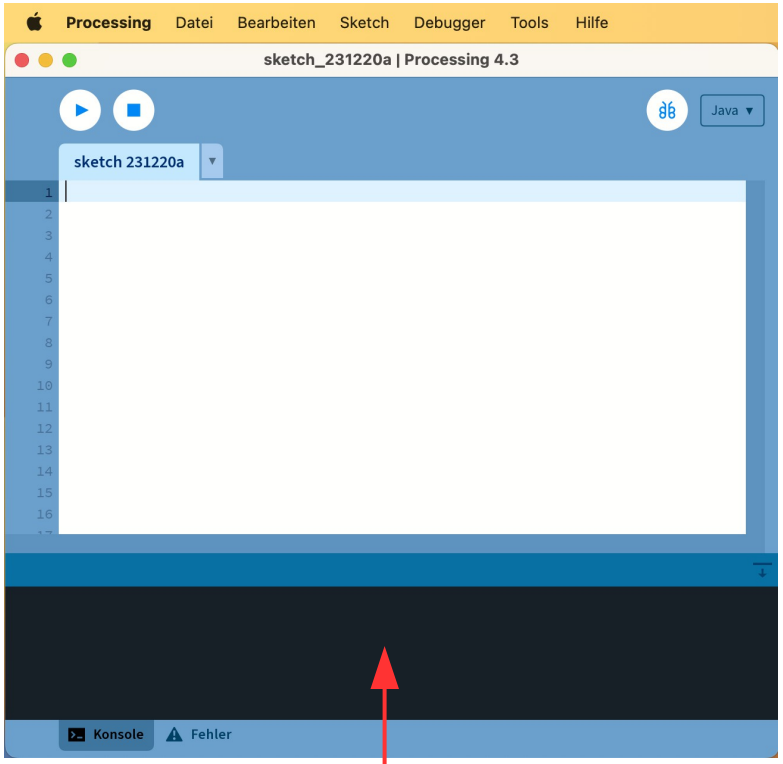
- Imperative (≙befehlsorientiert) Sprachen
- Wichtige imperative Familien:



- Weitere imperative Sprachen: FORTRAN, BASIC, ...
- Andere Sprachfamilien:
  - Z. B. *deklarative* Programmiersprachen (bspw. Prolog)

Ein erstes Programm

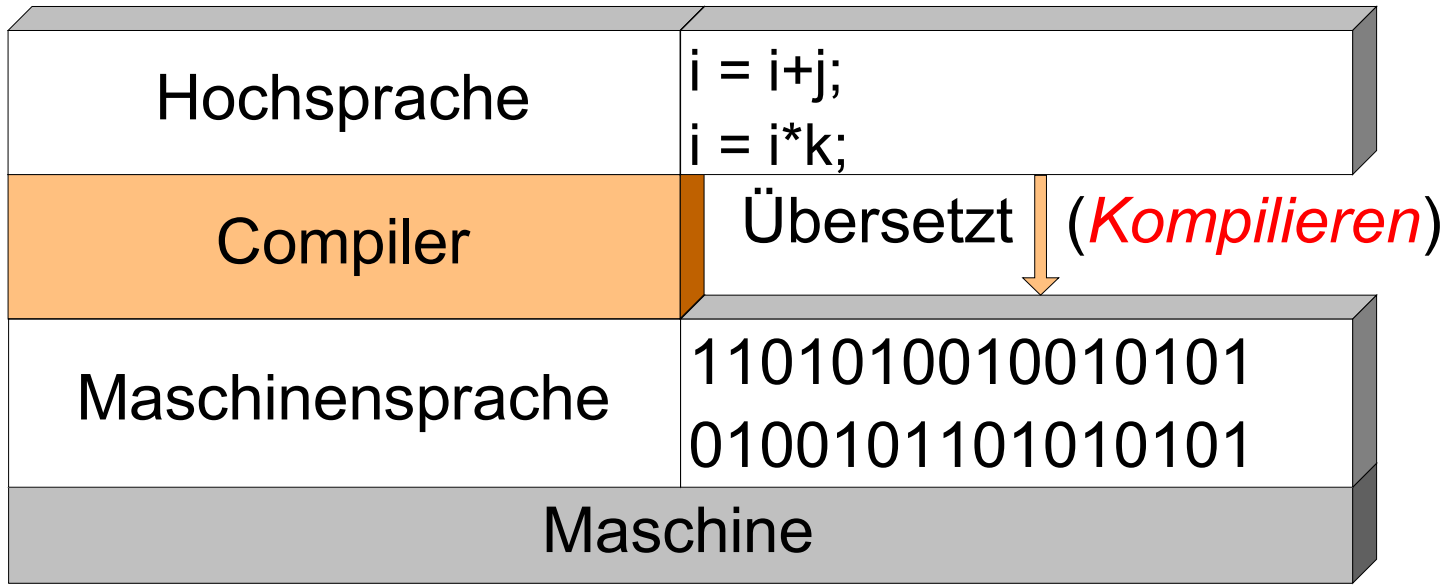
- Als Entwicklungsumgebung nutzen wir Processing
- Die Aufgabe besteht darin, in der Entwicklungsumgebung eine Meldung ausgeben zu lassen
- Die Ausgabe soll textuell in das Ausgabefenster erfolgen
  - Dieses wird auch als **Konsole** bezeichnet
- Wir benutzen hierfür die Programmiersprache Java und nutzen die sprachlichen Vereinfachungen, die Processing bietet



Konsole

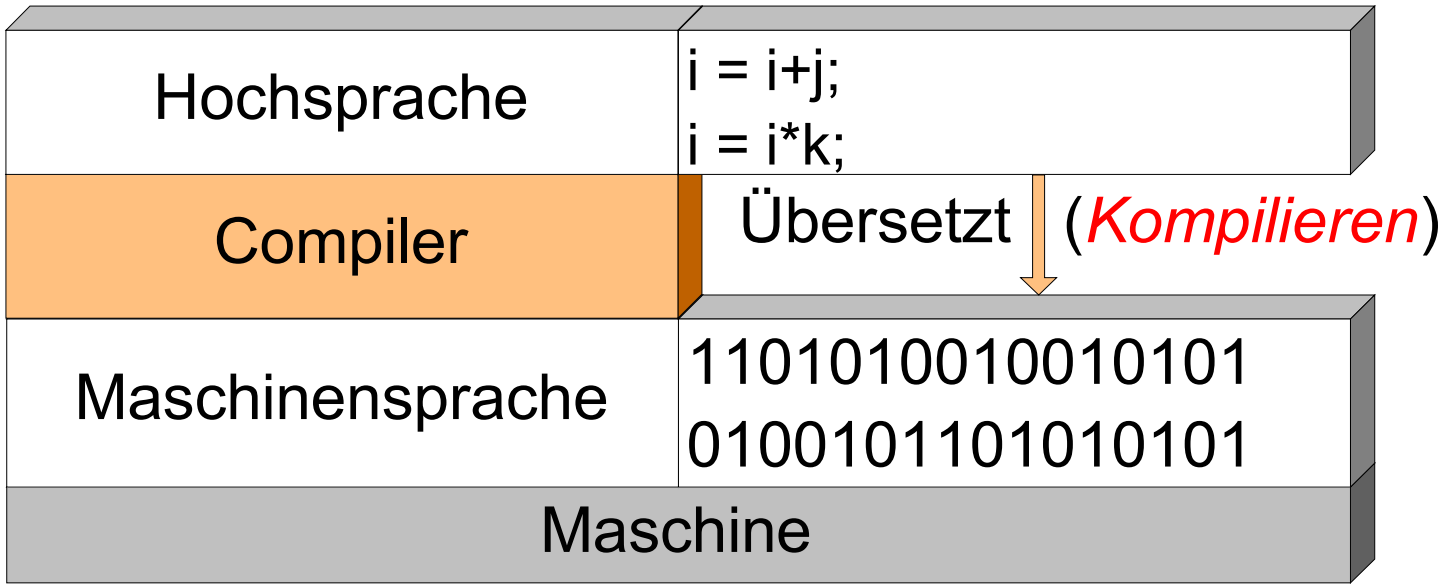
Übersetzung des Programms (1)

- Jedes vom Computer auszuführende Programm ist zunächst eine reine Textdatei, die in seine interne Maschinensprache übersetzt werden muss
- Die Übersetzung erfolgt durch ein spezielles Programm: den **Compiler**
  - Der Compiler überprüft das Programm ebenfalls auf **syntaktische Korrektheit**



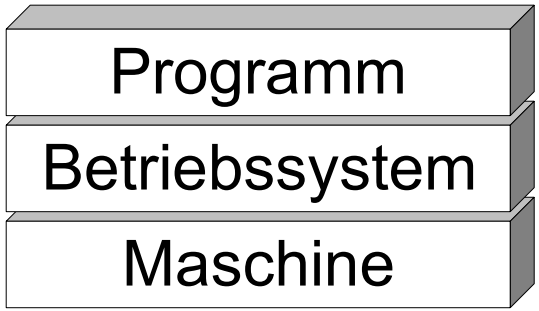
Übersetzung des Programms (2)

- Häufig erfolgt die Kompilierung heutzutage transparent im Hintergrund einer Entwicklungsumgebung
  - Ähnlich wie die Rechtschreibprüfung in einem Textprogramm



Ausführung des Programms

- Das Resultat des Übersetzungsvorganges ist ein Programm, welches wie jedes andere Computerprogramm auf dem Zielrechner ausgeführt werden kann



- Das Programm bekommt vom Betriebssystem Ressourcen (Speicher, Rechenzeit etc.) zugewiesen

Ein einfaches Processing Programm

HelloWorld.pde:

```
// this is a comment

println("Hello World!");

/*
   this is another comment
*/
```

- Processing-Programme (Sketch genannt) stehen in Textdateien mit der Endung **.pde**

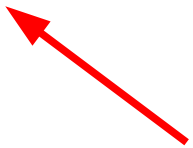
- Der Code wird als **Quellcode** (engl.: source code) bezeichnet, die Dateien als **Quellcodedateien** (engl.: source code file)

Ein einfaches Processing Programm

```
// this is a comment

println("Hello World!");

/*
   this is another comment
*/
```



- Anweisungen werden durch Semikolons voneinander getrennt
- Das Semikolon gibt an, wo eine Anweisung endet und die nächste beginnt

Ein einfaches Processing Programm

```
// this is a single-line comment

println("Hello World!");

/*
   this is another comment
*/
```

- Diese Zeilen sind **Kommentare**
- Der Compiler ignoriert diese Zeilen
- Kommentare werden benutzt, um den Code näher zu erläutern. Sie stellen keine Anweisungen dar!

Warum Code kommentieren?

- Erläuterungen für sich selbst und für andere
  - Der/die Entwickler/-in muss den Code auch nach einem halben Jahr noch verstehen können
  - Kollege, der den Code ggf. weiterentwickelt, muss diesen ebenfalls verstehen können
- Kommentare erläutern schwierige Codeabschnitte
  - Sie liefern die zum Verständnis erforderliche Hintergrundinformation
- Ein falscher, nicht mehr aktueller oder irreführender Kommentar ist schlechter als gar kein Kommentar!

Einrückungen

- Compiler übersetzt Code wortweise

```
 // this is                    a comment

      println("Hello
World!");

/*
          this is
 another comment
                                    */
```

- Für den Compiler ist dieser Code genau so gut lesbar, wie der vorherige
- Einrückungen dienen dazu, Programmcode für den Menschen lesbarer zu machen!

Ein einfaches Processing Programm

```
// this is a comment

println("Hello World!");

/*

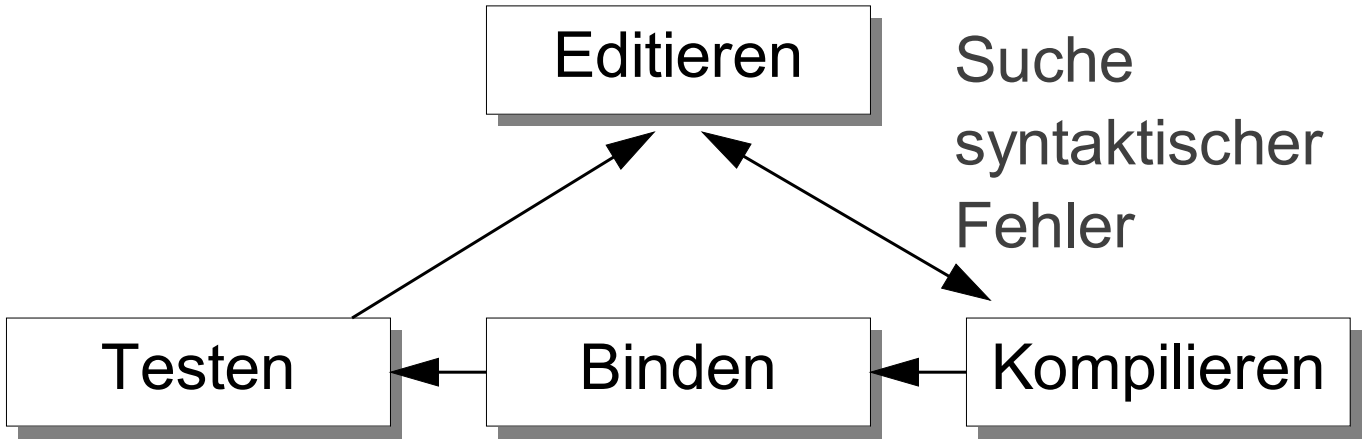
   this is another comment

*/
```

- Dies ist ein Ausgabebefehl
- Mittels dieser Anweisung kann eine Zeichenkette auf der Konsole ausgegeben werden
- Zeichenketten werden von Anführungszeichen eingeschlossen
- Die Zeichenkette lautet hier:
  `Hello World!`

Entwicklungszyklus

- Während der Programmentwicklung wird der Programmtext solange editiert, bis sich dieser fehlerfrei kompilieren lässt und das Programm fehlerfrei läuft



Grafische Ausgaben

- Neben der Konsole besitzt Processing alternativ auch ein grafisches Ausgabefenster
- Beispiel für eine Ausgabe in diesem Fenster:

```
size(640, 480);  // Fenstergröße definieren (b, h)
background(255); // Hintergrundfarbe festlegen
fill(0);         // Textfarbe festlegen
text("Hello World", 40, 40); // Textausgabe (x, y)
```

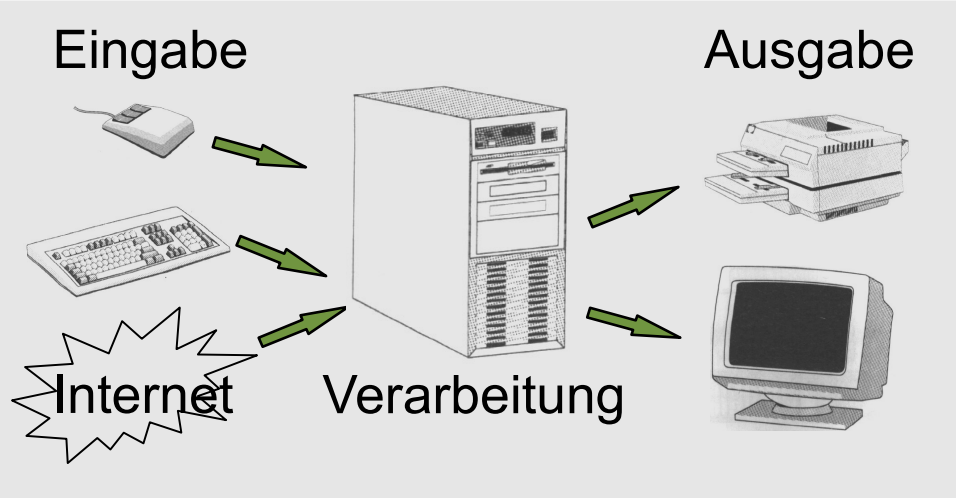
→ Textausgaben müssen positioniert werden

- Wir werden deshalb im weiteren Verlauf der Einfachheit halber die Konsole nutzen

Datentypen

Datentypen (1)

- Programme verarbeiten Daten
- Daten können unterschiedlichen Typen angehören
  - Zahlen, Zeichenketten, Bilder, Musikstücke, Videos etc.



- Im Computer werden alle diese Daten in derselben Weise gespeichert: als **Binärwörter**
- Damit die Abbildung der unterschiedlichen Daten auf die Binärwörter des Computers vom **Compiler** korrekt erledigt werden kann, müssen diese einen passenden **Datentyp** erhalten
  - Je nach Größe einer zu speichernden Zahl müssen vom Compiler z. B. Binärwörter passender Länge bereitgestellt werden

Datentypen (2)

- Das Binärwort 1111 1111 1001 1001 kann unterschiedlich interpretiert werden:
  - Als ganze Zahl: -103
  - Als natürliche Zahl: 65433
  - Als Zeichenkette bestehend aus zwei Zeichen:
    - 1111 1111 = 255 = 'ÿ'
    - 1001 1001 = 153 = '™'
  - Als Teil einer Folge von Binärwörtern, die zu einem langen Datenblock, z. B. einem Video, gehören
  - usw.
  - Der Datentyp verleiht Binärwörtern eine Bedeutung

Datentypen (3)

- Ein Datentyp legt die Eigenschaften von Daten einer Klasse fest
- Der Typ eines Datenwerts definiert:
  - Die erforderliche Menge Speicherplatz für die Speicherung dieses Werts
  - Den Wertebereich der Datenwerte
  - Die Menge der erlaubten Operationen, die mit Daten dieses Typs durchführbar sind
    - Bilder können z. B. komprimiert werden, aber keine Zahlen

Datentypen (4)

- Programmierung:
  - Der Programmierer definiert durch Zuweisen eines Datentyps, wie ein Datenwert zu interpretieren ist
  - Der Compiler
    - übernimmt die Interpretation des Datenwerts
    - überprüft den korrekten Gebrauch des Datenwerts
  - Auf diese Weise nimmt die Maschine dem Programmierer diese Arbeit ab

Arten von Datentypen

- Die Menge der Datentypen kann in zwei verschiedene Gruppen aufgeteilt werden:
  - **Primitive Datentypen (Grundtypen, Standardtypen)**
    - In der jeweiligen Programmiersprache vordefinierte Standardtypen
    - Nah an der Hardware definiert
    - Besitzen eine festgelegte Anzahl Bytes
  - **Abstrakte Datentypen**
    - Erlauben es eigene, ggf. komplexe Datentypen zu definieren
    - Dies erfolgt auf der Grundlage primitiver Datentypen

Primitive Datentypen

Primitive Datentypen

- Numerische Datentypen
  - Ganze Zahlen mit unterschiedlichen Wertebereichen
  - Kommazahlen mit unterschiedlichen Wertebereichen und Genauigkeiten
  - Schriftzeichen
- Referenzen
  - D. h. Adressen im Hauptspeicher
  - Der Hauptspeicher wird auch als Arbeitsspeicher bezeichnet

Wahrheitswert

- Typ: **`boolean`**
- Einfachster Datentyp
- Kennt nur die zwei Wahrheitswerte **`true`** und **`false`**
- Darstellung durch ein Binärzeichen bzw. Bit
- Verknüpfung über logische Operationen
  - Und, Oder, Negation

Datentypen für ganze Zahlen

- Engl.: *Integer number*
- Allgemein gilt: Keine Nachkommastellen möglich!
- Für alle Typnamen gilt:
  - Groß- und Kleinbuchstaben werden unterschieden

Datentypname	Länge	Byte	Wertebereich
`byte`	8 Bit	1	-128 bis 127
`short`	16 Bit	2	-32,768 bis +32,767
`int`	32 Bit	4	-2^{31} bis $2^{31}-1$
`long`	64 Bit	8	-2^{63} bis $2^{63}-1$

Schriftzeichen

- Typ: **`char`**
- Primitiver Datentyp zum Speichern einzelner Schriftzeichen
  - Es wird der 16-Bit Unicode Zeichensatz verwendet
- Werte vom Typ **`char`** werden in der Form '**`x`**' notiert
  - D. h. Buchstaben werden von einfachen Anführungszeichen eingefasst
- Es gibt drei Arten von **`char`**-Werten:
  - Einfache Zeichen
  - Escape-Sequenzen für Steuerzeichen
  - Unicode-Steuersequenzen

Escape-Sequenzen

`\b`: Rückschritt

`\t`: Tabulator

`\n`: Zeilenvorschub

`\f`: Seitenvorschub

`\r`: Zeilenrücklauf

`\"`: Doppelter Anführungsstrich

`\'`: Einfacher Anführungsstrich

`\\`: Backslash

`\uxxxx:` Abruf eines Zeichens aus dem Unicode-Zeichensatz mittels einer vierstelligen Hexadezimalzahl

Zeichenketten

- Setzen sich aus einer Folge von Schriftzeichen zusammen
- Stellen keinen primitiven Datentyp dar
  - Klasse: **`String`**
  - Klassennamen werden groß geschrieben
- Beispiel: "Dies ist eine Zeichenkette"
  - Zeichenketten werden von doppelten Anführungszeichen eingefasst

Zeichenketten: Formatierung

- Zeichenketten können mit Hilfe von Escape-Sequenzen formatiert werden

- Beispiele:

```
"\"Diese Zeichenkette steht in\nAnführungszeichen\""

"Ein Rückstrich und ein Anf\u00FChrungszeichen: \\\""
```

  - Angabe von Umlauten entweder direkt oder über Unicode-Escape-Sequenzen

Variablen

Variablen

- Um Daten zu verarbeiten, müssen sie im Programm zwischengespeichert werden können
- Programmierer benutzen hierfür eine abstrakte "Schublade", in welche Datenwerte gelegt werden können
- Diese Schublade trägt einen Namen und wird als **Variable** bezeichnet

Variablen und Datentypen

- Variablen werden als zu einem bestimmten Datentyp gehörend vereinbart
- Dieser Vorgang wird als **Deklaration** bezeichnet
- Der Datentyp legt fest:
  - Wie viel Speicherplatz die Variable benötigt
    - Die "Größe der Schublade"
  - Welche Operationen mit dieser Variable erlaubt sind
- Erst nach der Deklaration kann eine Variable benutzt werden!

Deklaration von Variablen

- Eine Deklaration besteht aus dem **Datentyp** der Variable, dem **Variablenname** und einem abschließenden Semikolon
  - Eine Variablendeklaration ist eine Anweisung für den Compiler
- Beispiel: `int myVariable;`
  - Deklaration einer Variable mit dem Namen `myVariable`, die ganze Zahlen vom Typ `int` (d. h. Integer) aufnehmen kann

Mehrfachdeklarationen

- Deklarationen von Variablen desselben Typs können in der gleichen Zeile stehen, wenn die Variablennamen durch Kommas voneinander getrennt werden
- Beispiel:
  - An Stelle von

```
int var1;
int var2;
int var3;
```

  - kann folgendes geschrieben werden:

```
int var1, var2, var3;
```

Initialisierung einer Variablen

- Nach der Deklaration ist der Wert einer Variable zunächst noch undefiniert
  - Die Schublade bzw. Speicherzelle kann leer sein, es kann sich aber auch irgendein Wert darin befinden
- Mittels der **Initialisierung** wird die Variable auf einen definierten Anfangswert gesetzt
  - Die zu dieser Variable gehörende Speicherzelle enthält dann den ihr zugewiesenen Wert
- Beispiel: Initialisierung einer Variable **`myVariable`** mit dem Wert `0`

```
myVariable = 0;
```

Definition einer Variablen

- Die Initialisierung kann auch als Teil der Deklaration geschehen:

```
int myVariable = 0;
```

  - Dies wird auch als **Definition** bezeichnet
- Es können auch mehrere Variablen in der gleichen Zeile definiert werden:

```
int myVar1=0, myVar2=10, myVar3=-5;
```

- Auch eine Mischung aus einfachen Deklarationen und Definitionen ist möglich:

```
int myVar1, myVar2=10, myVar3;
```

Variablenname

- Erlaubt sind die Buchstaben **`a`** bis **`z`** bzw. **`A`** bis **`Z`**, die Ziffern **`0`** bis **`9`** und die Sonderzeichen _ und **`$`**
  - Groß- und Kleinbuchstaben werden unterschieden
- Nicht erlaubt: Leerzeichen und reservierte Wörter
  - Z. B. **`int`**, **`float`**, **`if`**, **`while`**, **`class`**, etc.
- Variablenname darf nicht mit einer Ziffer anfangen
- Instruktive Namen wählen!

Variablennamen: Konventionen

- **`myVariable`**
  - Binnenmajuskel(-versal)-Schreibweise (engl.: *CamelCase*)
- **`my_variable`**
- **`imyVariable, i_my_variable`**
  - Ungarische Notation
  - Die Ungarische Notation definiert eine Konvention für die Namensgebung von Bezeichnern
  - Es wird eine Abkürzung für den Datentyp der Variable dem Variablennamen vorangestellt

Ungarische Notation (1)

- In Projekten arbeiten viele Programmierer häufig am gleichen Programmcode
- Ein einheitlicher Stil und ein einheitliches Format für die Bezeichner unterstützt die Verständlichkeit des Codes
  - Wichtig für eine effiziente Programmierung und Qualitätssicherung
- Die Ungarische Notation definiert eine Konvention für die Namensgebung von Bezeichnern
  - Die Ungarische Notation ist z. B. das Standard-Format für die Software-Entwicklung bei Microsoft

Ungarische Notation (2)

- Konventionen für die Namensgebung von
  - Typen
  - Variablen
  - Konstanten
  - Unterprogrammen (Prozeduren, Funktionen, Methoden)
  - Parameter
  - Klassen

Ungarische Notation (3)

- Einige Beispiele:
  - `c` `char`
  - `b` `boolean`
  - `by` `byte`
  - `i` `int`
  - `w` `unsigned int bzw. word`
  - `l` `long`
  - `s` `Zeichenkette (string)`
  - `lp` `64-bit long pointer`
- Abkürzungen können auch kombiniert werden:

```
int lpiData; //64-bit long pointer auf int-Wert
```

Abkürzungen werden dem Bezeichner vorangestellt

Variablendeklaration: Zusammenfassung

```
1. Datentyp  Variablenname;

2. Datentyp  Variablenname = Anfangswert;

3. Datentyp  Variablenname1, Variablenname2, ...;

4. Datentyp  Variablenname1 = Anfangswert1,

             Variablenname2 = Anfangswert2, ...;

5. Datentyp  Variablenname1 = Anfangswert1,

             Variablenname2, ...;
```

Übung

```
// Gibt es hier falsche Deklarationen?
long good-by;
short shval = 0;
double bubble = 0, trouble = 8
byte the bullet;
int double;
char thisMustBeTooLong;
int 8ball;
```

Variablenwert ausgeben (1)

- Wie bei Zeichenketten
- Anstelle von Zeichenkette Variablennamen benutzen

```
int hoursWorked = 40;
print("Hours Worked: ");
println(hoursWorked);
```

- Ausgabe: **`Hours Worked: 40`**
- Der Wert von **`hoursWorked`** wird implizit in eine Zeichenkette umgewandelt
  - **`print()`**: Drucke Zeichenkette
  - **`println()`**: Drucke Zeichenkette plus neue Zeile

Variablenwert ausgeben (2)

- Zeichenketten können mittels des **+**-Operators auch mit Variablenwerten verbunden werden
  - Der +-Operator ist **überladen**, d. h. das Ergebnis der Operation ist von den verknüpften Datentypen abhängig
- Anstatt

```
print("Hours Worked: ");

println(hoursWorked);
```

- kann folgendes geschrieben werden:

```
println("Hours Worked: " + hoursWorked);
```

Beispielprogramm

```
int hoursWorked = 40;

println("Hours Worked: " + hoursWorked);
```

Zuweisung

- Bisher wurde der Variablenwert nur bei der Initialisierung geändert
- In derselben Weise – aber ohne erneute Angabe des Typs - wird einer Variable während der Programmausführung ein neuer Wert zugewiesen:

```
myVariable = 0;
```

- Die entsprechende Operation heißt **Zuweisung**
- Syntax:

```
variableName  =  expression;
```

Variablenname — auswertbarer Ausdruck

Zuweisungsoperator

Beispielprogramm

```
int hoursWorked = 40;
println("Hours Worked: " + hoursWorked);
hoursWorked = 60;
println("Hours Worked: " + hoursWorked);
```

Ausdrücke

Zuweisung mit Auswertung eines Ausdrucks

- Einer Variable kann auch der Wert eines Ausdrucks zugewiesen werden
- Beispiel:

```
sum = 32 + 8;
```

- Die Auswertung und Zuweisung erfolgt in zwei Schritten:
  1. Der Ausdruck `32 + 8` wird zu `40` ausgewertet
  2. Das Ergebnis `40` wird der Variable `sum` zugewiesen

Zuweisung mit Auswertung der Variable selbst

- Teil des Ausdrucks kann auch die Variable sein, der das Ergebnis zugewiesen wird

```
zaehler = zaehler + 1;
```

- In diesem Fall wird die Variable zunächst ausgelesen und ihr Wert bei der Auswertung des Ausdrucks verwendet
- Das Ergebnis der Auswertung wird wieder in die Variable geschrieben

Ausdrücke

- Ausdruck (engl.: *expression*)
- Kombination aus:
  - **Operanden**:
    - **Literalen**: Zahlenwerte, Zeichen, Zeichenketten, **`true`**, **`false`**, **`null`**
    - **Variablen** und **Konstanten**
  - **Operatoren**: z. B. "**+**", "**-**", "**%**", "*****", "**/**"
  - **Klammern**: "**(**" und "**)**"
- Beispiele:
  - `(42 - 12)`
  - `(32 - y) / (x + 5)`

Literal: Bezeichnung für die Elemente eines Datentyps

Beispiele: `2, 17, -3, null, "Hello World"`

Arithmetische Operatoren

Operator	Bedeutung
+/-	Unäres Plus/Minus (Vorzeichen)
*	Multiplikation
/	Division
%	Rest der ganzzahligen Division
+	Addition
-	Subtraktion

Modulo-Operation: %

- z und d seien ganzzahlig
- z/d: Liefert den Quotient q, der angibt, wie oft der Divisor d ganz in den Dividend z passt
- z%d = z – q*d: Liefert den Rest der ganzzahligen Division
- z = (z/d)*d + (z%d)
- Beispiel:

```
int quotient = 15/6 // = 2 = q
int rest = 15%6 // = 3 = 15-12
```

Modulo mit negativen ganzzahligen Werten

- Die Rest-Operation kann auch mit negativen Werten benutzt werden
- Dabei gilt:
  - Ist der Dividend positiv, so ist der Rest positiv
  - Ist der Dividend negativ, so ist der Rest negativ
  - Das Vorzeichen des Divisors wird immer ignoriert!
- Beispiel:
  - 17 % 3 = 2 -17 % 3 = -2
  - 17 % -3 = 2 -17 % -3 = -2

Warum ist das so?

- Dies folgt aus: z = (z/d)*d + (z%d)
- Beispiel (Rest r = z%d):
- 17 % 3 = 2:
  - 17 = (17 / 3)*3 + r <=> r = 17 – 15 = 2
- 17 % -3 = 2:
  - 17 = (17 / -3)*-3 + r = -5*-3 +r <=> r = 17 – 15 = 2
- -17 % 3 = -2:
  - -17 = (-17 / 3)*3 + r = -5*3 +r <=> r = -17 + 15 = -2
- -17 % -3 = -2:
  - -17 = (-17 / -3)*-3 + r = 5*-3 +r <=> r = -17 + 15 = -2

Arithmetische Operatoren: Präzedenz

Operator	Bedeutung	Präzedenz (Rang)
+/-	Unäres Plus/Minus (Vorzeichen)	Hoch
*	Multiplikation	Mittel
/	Division	Mittel
%	Rest der ganzzahligen Division	Mittel
+	Addition	Niedrig
-	Subtraktion	Niedrig

Präzedenz

- Beispiel:

```
myFirstVal-6.73*mySecondVal/12+7*-myThirdVal
myFirstVal-6.73*mySecondVal/12+7*-myThirdVal
myFirstVal-6.73*mySecondVal/12+7*tmp1
myFirstVal-tmp2/12+7*tmp1
myFirstVal-tmp3+7*tmp1
myFirstVal-tmp3+tmp4
tmp5+tmp4
tmp6
```

- Falls Ausdrücke in einer anderen Reihenfolge ausgewertet werden sollen: Klammern benutzen!

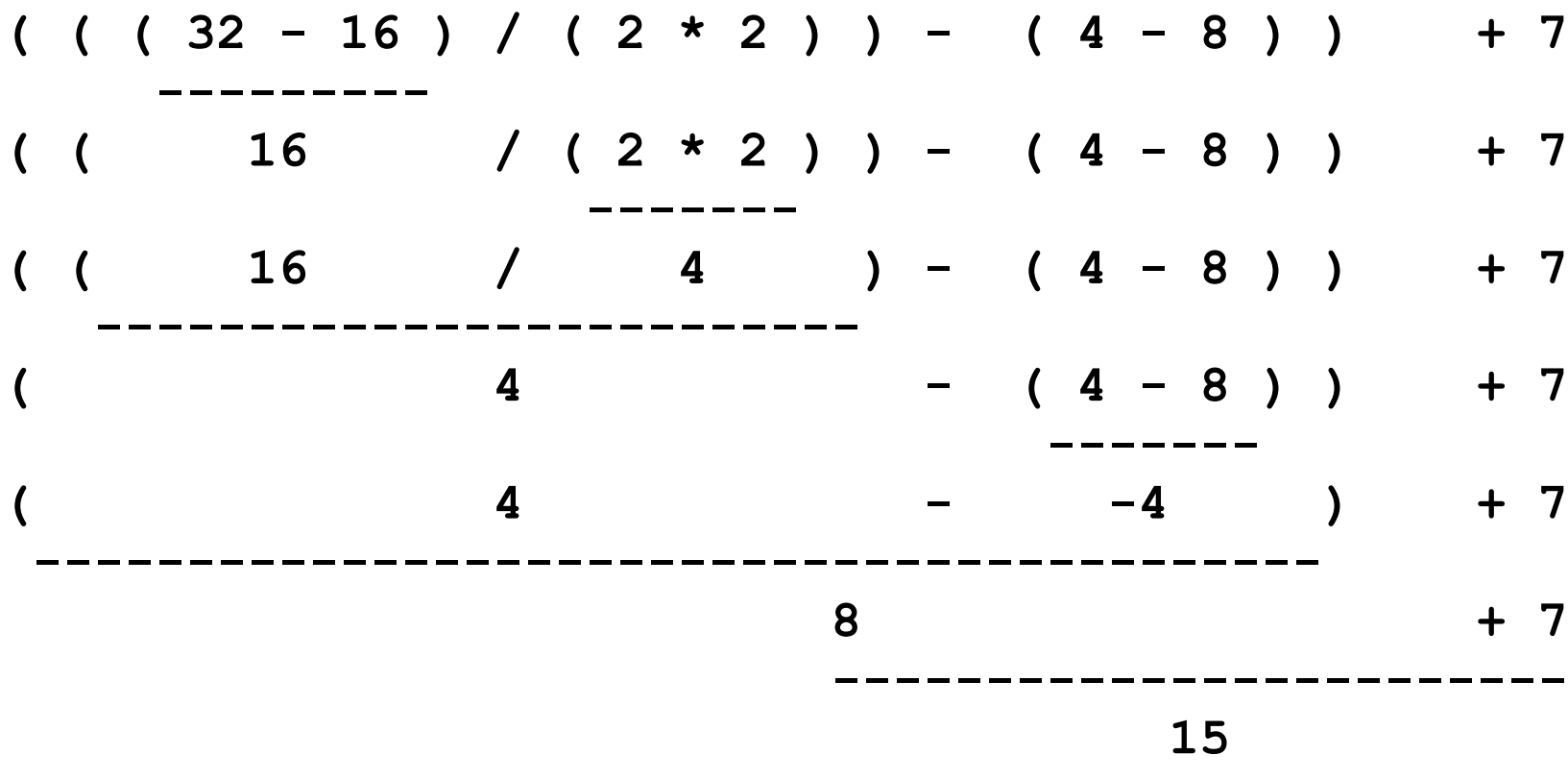
Übung

- Lösen Sie den folgenden Ausdruck schrittweise auf:

```
3*val1*val2 + val3/7.3 – val4 + 5.2
```

Klammern

- Zum Erreichen einer individuellen Auswertungsreihenfolge:
  - Klammerhierarchie benutzen
  - Innere Klammern werden dann zuerst ausgewertet



Leerzeichen

- Leerzeichen dienen dazu, Ausdrücke zu strukturieren und ihre Lesbarkeit zu erhöhen
- Sie werden ignoriert, solange sie kein Literal oder eine Variable oder Konstante trennen
  - Richtig: `(hoursWorked * payRate) - deduction`
  - Falsch: `(hours Worked * payRate) - deduction`

Leerzeichen

- Leerzeichen haben keine Auswirkung auf die Reihenfolge, in der die Operatoren ausgewertet werden
- Beide Ausdrücke unten führen auf dasselbe Ergebnis

```
12   -   4/2  +  2

12-4   /   2+2
```

- Die Reihenfolge der Auswertung wird nur durch den Rang der Operatoren und die Klammerstruktur festgelegt

Gleitkommazahlen

Kommazahlen im Computer (1)

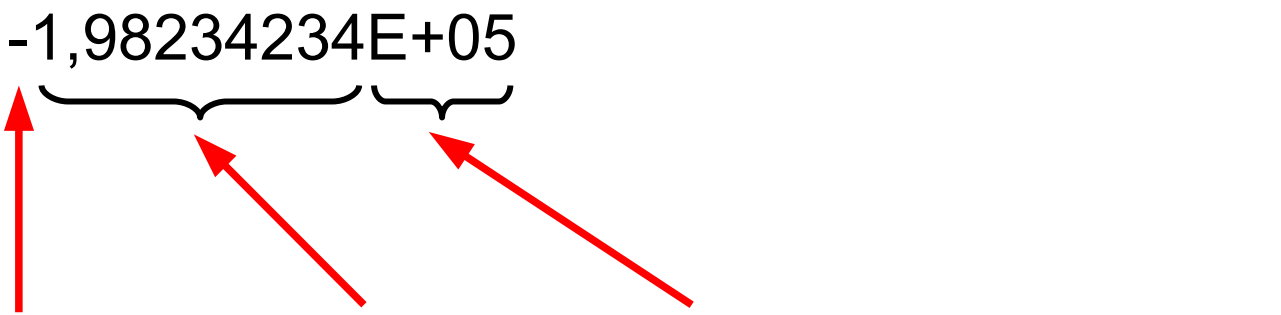
- Reelle Zahlen werden im Computer wieder durch Binärwörter dargestellt
- Folglich ist nur eine feste Anzahl von Stellen vorhanden
- Wie lassen sich reelle Zahlen zweckmäßig durch Binärwörter darstellen?
- Eine Möglichkeit: Für die Nachkommastellen wird eine feste Anzahl Stellen reserviert, der Rest für die Stellen vor dem Komma
- Dies wird als **Festkommazahl** bezeichnet
- **Problem**: Für einige Anwendungen ist die Anzahl der Nachkommastellen unnötig groß (z. B. Astronomie), für andere Anwendungen die Genauigkeit zu gering (z. B. Chip-Layout)

Kommazahlen im Computer (2)

- Deshalb ist eine Darstellung zweckmäßiger, bei der sich die Genauigkeit an der Größe des Zahlenwerts orientiert
- Die Genauigkeit ist dann bei kleinen Zahlen groß und bei großen Zahlen klein
- Die **Gleitkommadarstellung** erfüllt diese Bedingung

Gleitkommazahlen (1)

- Auch: Fließkommazahl (engl.: *floating point number*)
- Eine Gleitkommazahl besteht aus drei Teilen:



- Im Computer werden Gleitkommazahlen als Dualzahlen dargestellt

Gleitkommazahlen (2)

- Jede Gleitkommazahl kann in eine Gleitkommazahl mit einer Stelle vor dem Komma umgewandelt werden
  - Beispiel: $384 \times 10^6 = 3{,}84 \times 10^8$
- Dies wird als **normierte Gleitkommazahl** bezeichnet
- Dualzahldarstellung:
- $3{,}84 \times 10^8{}_{10} = 1{,}01101110001101 1E11100_2$
  - Dezimalzahl: Exponent zur Basis 10
  - Dualzahl: Exponent zur Basis 2

Gleitkommazahlen (3)

- Die normierte Gleitkommazahl $\pm 1, m_1 m_2 m_3 \ldots m_n \cdot 2^E$ stellt den Zahlenwert

$$(-1)^v \cdot (1 + m_1 \cdot 2^{-1} + \ldots + m_n \cdot 2^{-n}) \cdot 2^E$$

mit:
v: Vorzeichen
$m_1 \ldots m_n$: Mantisse
E: Exponent

dar

Gleitkommazahlen (4)

- Im Rechner müssen Gleitkommazahlen mit Binärwörtern fester Länge dargestellt werden
  - 32-Bit, 64-Bit
- Wie werden Bits auf Vorzeichen, Mantisse und Exponent verteilt?
- IEEE Short Real Format für 32-Bit Zahlen
  - Vorzeichen: 1 Bit, Mantisse: 23 Bit, Exponent: 8 Bit
- IEEE Long Real Format für 64-Bit Zahlen
  - Vorzeichen: 1 Bit, Mantisse: 52 Bit, Exponent: 11 Bit

- (IEEE: *Institute of Electrical and Electronics Engineers*)

Datentypen für Gleitkommazahlen

- Gleitkommazahlen werden mit Punkt geschrieben
- Beispiele:
  - 123.0 -123.5 -198234.234 0.00000381
  - 1.23e+02 -1.235e+02 -1.98234234e+05 3.81e-06
    - e: Exponent zur Basis 10
- Processing: Interne Bibliotheksfunktionen nutzen nur **`float`**

Datentypname	Länge	Byte	Wertebereich
`float`	32 Bit	4	±1.401e-45 bis +3.403e+38
`double`	64 Bit	8	±4.941e-324 bis 1.798e+308

Exponentialdarstellung: Übung

- Schreiben Sie die folgenden Zahlen aus, d. h. ohne Exponentialdarstellung:

- 9.15e+03 -7.01e+02 3.1415927e+05 1.51e-04

- Der Exponent bezeichnet die Anzahl der Stellen, um die das Komma nach links oder rechts zu verschieben ist
  - Positiver Exponent: Komma nach rechts verschieben
  - Negativer Exponent: Komma nach links verschieben

Gleitkommazahlen: Umwandlung

- Bei Verknüpfungen zwischen ganzen und Gleitkommazahlen werden ganze Zahlen automatisch in Gleitkommazahlen umgewandelt
- Für die Umwandlung von Gleitkommazahlen in ganze Zahlen stehen in der Mathematikbibliothek die folgenden Rundungsfunktionen zur Verfügung:
  - **`round()`**: Auf- bzw. abgerundeter Zahlenwert
  - **`floor()`**: Nächstniedriger Zahlenwert
  - **`ceil()`**: Nächsthöherer Zahlenwert
- Beispiel: **`int x = round(4.7);`**
  - Das Funktionsargument wird wie in der Mathematik üblich in runden Klammern notiert

Gleitkommazahlen: Genauigkeit (1)

- Die Genauigkeit von Gleitkommazahlen ist beschränkt, da die Anzahl der Stellen endlich ist
  - Bei Umwandlungen treten Ungenauigkeiten auf
- Während längerer Berechnungen können sich Ungenauigkeiten zu signifikanten Fehlern aufschaukeln!
- Vorsicht auch bei Vergleichen für Programmverzweigungen
  - Test auf Gleichheit vermeiden!

Gleitkommazahlen: Genauigkeit (2)

- Beispiele:
  - 87345600...0 an Stelle von 87345621...3
  - 1/3 als Dezimalzahl: 0.33333333333333333....
  - Pi: 3,141592..., e: 1,414213...
    - Für die exakte Darstellung ist eine unendliche Anzahl von Nachkommastellen erforderlich
  - 0,1 z. B. wegen beschränkter Stellenzahl der Dualmantisse nicht exakt darstellbar
    - Wird durch 0,0999999940 bzw. 0,1000000015 angenähert

Operatoren und Datentypen der Operanden

- Das Ergebnis einer Operation ist nicht nur vom Operator, sondern auch von den Datentypen der beteiligten Operanden abhängig
- Die durchzuführende Operation wird in Abhängigkeit von diesen Datentypen interpretiert

Ganzzahlige Operationen

- Beide Operanden sind Integer-Werte
- Beispiele:

```
int op1=7,op2=2;
int tmp = op1/op2; // liefert tmp == 3
tmp = 7/2;         // liefert tmp == 3
tmp = 7.0/2.0      // liefert Compiler-Fehler:
                   // cannot convert from double
                   // to int
double ftmp = 7/2; // liefert ftmp == 3.0!!
```

Gleitkomma-Operationen

- Mindestens ein Operand ist **`float`**- oder **`double`**-Wert

- Beispiele:

```
float op1=7,op2=2;
double tmp = op1/op2;    // liefert tmp = 3.5
tmp = 7.0/2.0;           // liefert tmp = 3.5
tmp = 7.0/2;             // liefert tmp = 3.5
tmp = 7/2.0;             // liefert tmp = 3.5
```

Beispielprogramm mit Hilfsvariable

```
int hoursWorked = 40;
double payRate = 10.0, taxRate = 0.10;
double amount = 0;

println("Hours Worked: " + hoursWorked);
amount = hoursWorked * payRate;
println("Pay Amount: " + amount);
amount = amount * taxRate;
println("Tax Amount: " + amount);
```

- Um den Wert einer Variable zu lesen oder zu ändern, wird ihr Name in eine Anweisung eingesetzt

Beispielprogramm ohne Hilfsvariable (1)

```
int hoursWorked = 40;
double payRate = 10.0, taxRate = 0.10;
println("Hours Worked: " + hoursWorked);
println("Pay Amount: " + (hoursWorked * payRate));
println("Tax Amount: " + (hoursWorked * payRate *
        taxRate));
```

- In Ausgabeanweisungen können Ausdrücke verwendet werden
  - Spart Hilfsvariable **`amount`** ein
  - Codelesbarkeit sollte aber nicht darunter leiden

Beispielprogramm ohne Hilfsvariable (2)

```
int hoursWorked = 40;
double payRate = 10.0, taxRate = 0.10;

println("Hours Worked: " + hoursWorked);
println("pay Amount: " + (hoursWorked * payRate));
println("tax Amount: " + (hoursWorked * payRate *
        taxRate));
```

- Anweisung kann auf mehrere Zeilen verteilt werden, solange dabei keine **Schlüsselwörter**, **Bezeichner**, **Zeichenketten** oder **Zahlen** getrennt werden
  - Im Prinzip überall, wo ein Leerzeichen im Code stehen könnte
- Abgetrennte Teile gegenüber der ersten Zeile einrücken, um Lesbarkeit zu erhöhen!

Gibt es hier Fehler?

```
long   hoursWorked = 40;
double payRate     = 10.0, taxRate = 0.10;

println("Hours
    Worked: " + hoursWorked);

println("pay Amount  : " + (hours
    Worked * payRate));

println("tax Amount  : " + (
    hoursWorked * payRate * taxRate));
```

Gemischte Ausdrücke

- Gemischt rational-ganzzahliger Ausdruck:

```
1.5 + 7/2
```

- Das Ergebnis ist 4,5 und nicht 5!
- Das Verhalten ist also anders als z. B. beim Taschenrechner
- Dort werden in jedem Falle alle Zahlen implizit in eine Gleitkommazahl umgewandelt

Übung

- Werten Sie die folgenden Ausdrücke schrittweise wie ein Computer aus:
  - `1/5 - 1/8`
  - `10%5 - 10%8`
  - `(15 * 37.0) / 15`
  - `(1/2 + 7.5) / 2.5`
  - `(18 + 0.0) / 4`
  - `(8/12 + 6) / 3`
  - `-12%5 + 23%-7`

Explizite Typumwandlung

- Typ-Umwandlung: Beispiel

```
int op1=7,op2=2;
double dtmp = op1/op2;   // dtmp ist 3.0
dtmp = (op1+0.0) / op2;  // dtmp ist 3.5
```

- Alternative: Zieltyp mit **Typumwandlungs-Operator** erzwingen
  - Engl.: Type cast
  - Syntax: **(**Typbezeichnung**)**
    - Unärer Operator. Besitzt die gleiche Präzedenz wie die Vorzeichenoperatoren
- Beispiel:

```
dtmp = (double) op1/op2; // dtmp ist 3.5
```

Ausgabe von Ausdrücken

- Um Ausdrücke mittels des "+"-Operators für Zeichenketten ausgeben zu können, müssen diese ggf. in Klammern gesetzt werden

- Beispiel:

```
println("A1: " + 1/2.0 + 7.5);

// liefert: A1: 0.57.5

println("A1: " + (1/2.0 + 7.5));

// liefert das gewünschte Ergebnis: A1: 8.0
```

Konstanten

Konstanten (1)

- Oftmals werden in Programmen feste Zahlenwerte benötigt
  - Z. B. Pi, e, g, Fehlertoleranz
- Ändert sich ein konstanter Wert, so müssen im Code alle diese Werte angepasst werden
  - Dies ist eine potentielle Fehlerquelle, da Werte übersehen werden könnten und außerdem ist es unnötige Arbeit
- Darüber hinaus sind Zahlenwerte häufig nichts sagend

Konstanten (2)

- Darum sollten Zahlenwerte durch selbst erklärende Bezeichner ersetzt werden
- Hierfür könnten im Prinzip Variablen verwendet werden
- Problem: Variablenwerte lassen sich mittels Zuweisung ändern
- Dies kann auch versehentlich geschehen! Fehlerquelle!
- Aus diesem Grunde können Konstanten definiert werden

Konstanten (3)

- Eine Konstante wird wie eine Variable, aber mit dem zusätzlichem Attribut **`final`** deklariert
  - **`final`** zeichnet die Variable als nicht änderbar aus
- Beispiele:
  - `final double PI = 3.1415927;`
  - `final double G  = 9.81;`
- Konvention:
  - Konstantennamen in GROSSBUCHSTABEN schreiben!
- Selbsterklärende Bezeichner wählen!

Dateneingabe

Variablenwerte von der Tastatur einlesen

- Bisher wurden nur Variablenwerte ausgegeben
- Beispiel (in die Konsole):

```
println("Hours Worked:" + hoursWorked);
```

- Beispiel (in das grafische Ausgabefenster):

```
text("Hours Worked:" + hoursWorked, 50, 50);
```

- Programme müssen ebenfalls Eingabewerte lesen können
- Dies kann nicht über die Konsole oder das grafische Ausgabefenster erfolgen

Programmeingabe

- Für die Eingabe benötigen wir eine spezielle Bibliothek
  - Wie die Mathematikbibliothek, nur mit anderen Befehlen
- Diese heißt **`InOut`** und ist als Download von der Webseite erhältlich:

  https://studium.dm.hs-ulm.de/kruse/prog1p/download/InOut.zip
- Die Bibliothek muss in das **`libraries`**-Unterverzeichnis des Sketch-Verzeichnisses von Processing entpackt werden
- Beispiel für ein mögliches Processing-Verzeichnis (Mac): **`/Benutzer/Dokumente/Processing/libraries/`**
- Ergebnis des Entpackens: **`/Benutzer/Dokumente/Processing/libraries/InOut`**

`InOut`-Bibliothek

- Bevor die `InOut`-Bibliothek im Quellcode benutzt werden kann, muss sie zunächst importiert werden:

```
import template.library.*;
```

- Die Bibliothek enthält die „Klasse“ `IOLib`
  - Diese stellt die Eingabeanweisungen als sogenannte „Methoden“ zur Verfügung
    - Mit Klassen und Methoden werden wir uns später im Kurs noch genauer beschäftigen
- Nach dem Import können die Methoden der Klasse benutzt werden
- Beispiel: Eingabe eines Integer-Wertes

```
//...Programmcode...
int num1 = IOLib.getInt("First number:");
```

  - Schreibweise: `Klassenname.Methodenname`

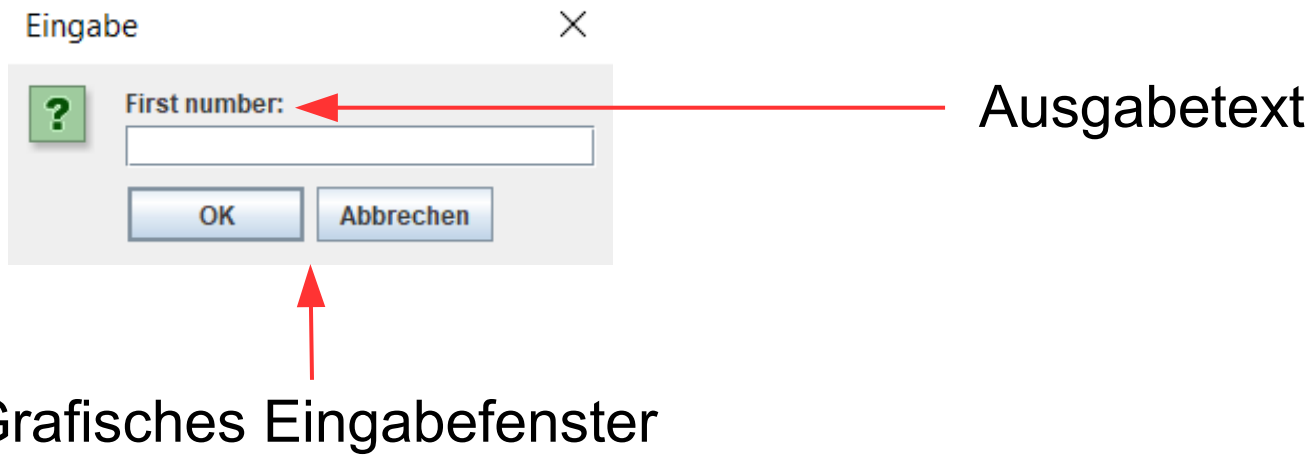
`IOLib`: Methoden

- `getBoolean()` Lesen eines Wahrheitswertes
- `getInt()` Lesen eines Integer-Wertes
- `getLong()` Lesen eines Long-Wertes
- `getFloat()` Lesen eines Float-Wertes
- `getDouble()` Lesen eines Double-Wertes
- `getString()` Lesen eines Strings

Benutzen der `InOut`-Bibliothek: Beispiel 1

- Einlesen eines Integer-Werts:

```
import template.library.*;
int num1 = IOLib.getInt("First number:");
println("Ausgabe: " + num1); // Ausgabe in Konsole
```



Benutzen der `InOut`-Bibliothek: Beispiel 2

```
import template.library.*;

// Nacheinander Zeichenkette und Integer-Wert
// einlesen und beides in die Konsole ausgeben
String str1 = IOLib.getString("Name: ");
int num1 = IOLib.getInt("Age: ");

println("Name: " + str1 + ", Age: " + num1);
```

Kontrollstrukturen

Kontrollstrukturen

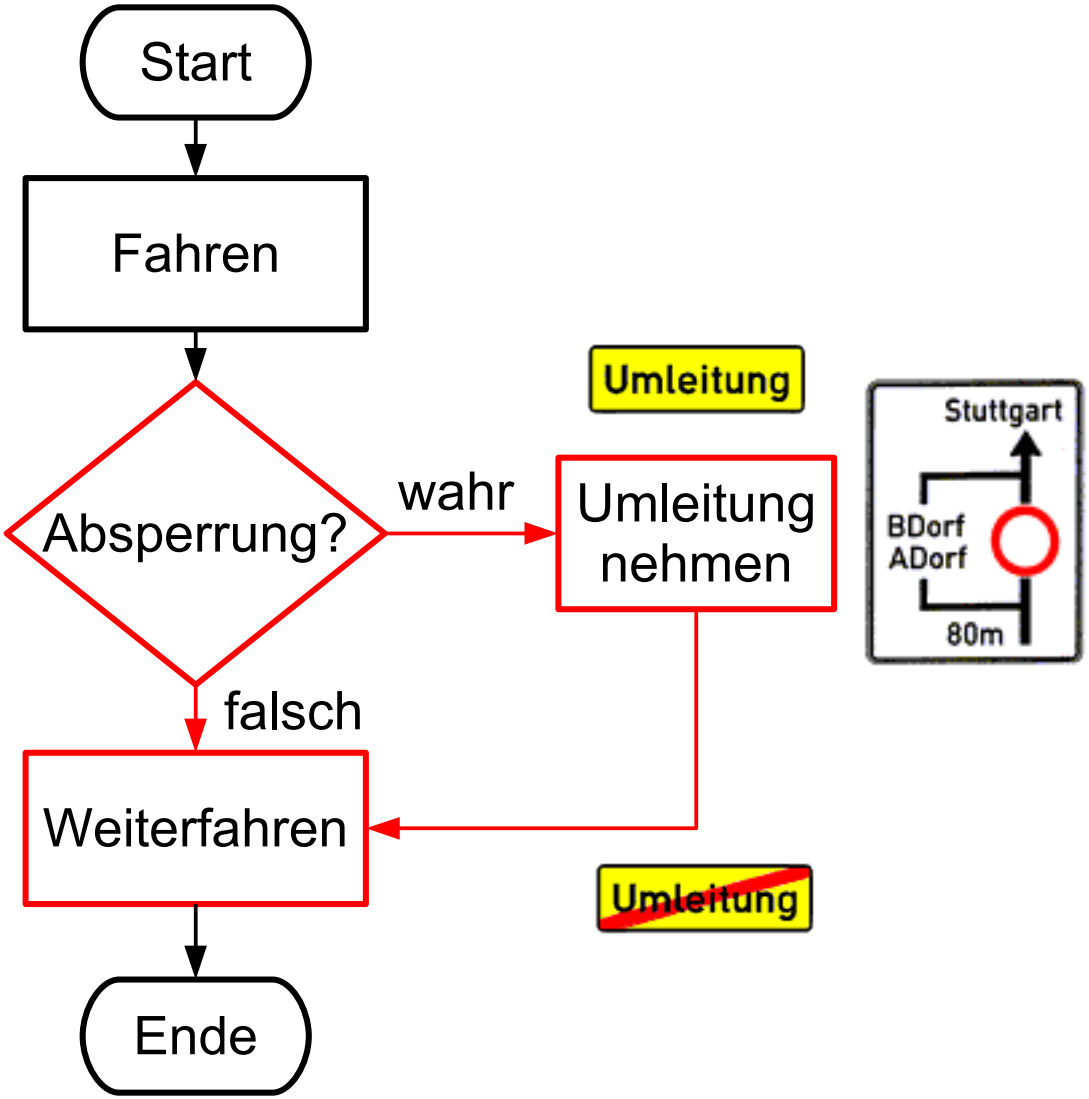
- Mit Hilfe von Kontrollstrukturen können Anweisungen gezielt und kontrolliert zu neuen komplexen Anweisungen zusammengesetzt werden

- Zu den grundlegenden Kontrollstrukturen gehören:
  - Die Verbundanweisung (der Block)
  - Die Fallunterscheidungen
  - Die Schleifenanweisungen

Fallunterscheidungen

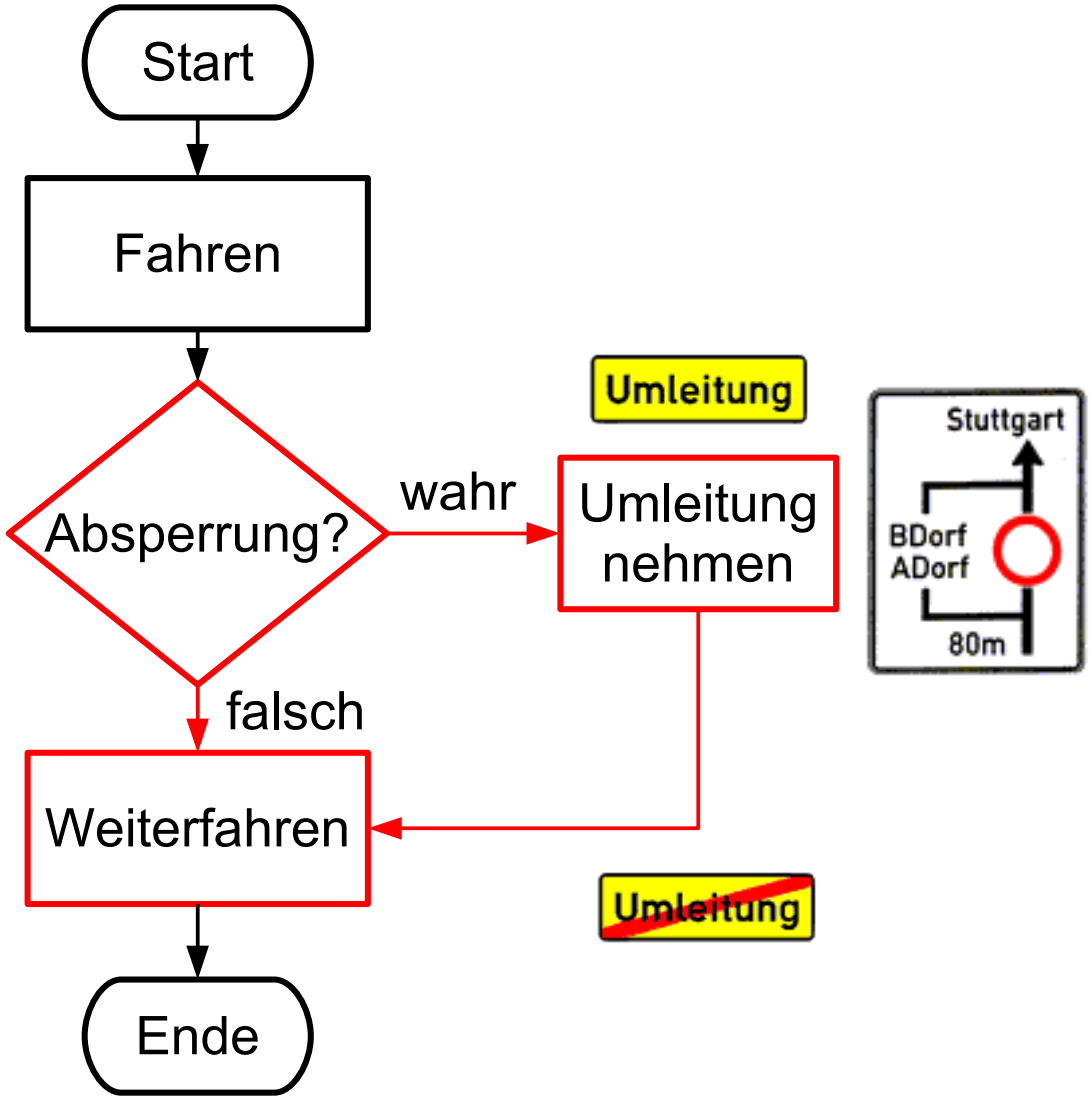
Fallunterscheidungen

- Hochsprachenbefehle werden in der Reihenfolge ausgeführt, in der sie im Programm erscheinen
- Aber: Bei der Bearbeitung von Problemen ist häufig eine Entscheidung zwischen mehreren Lösungswegen erforderlich

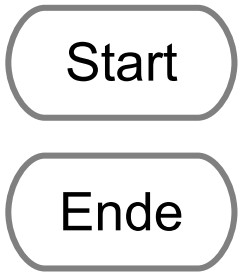


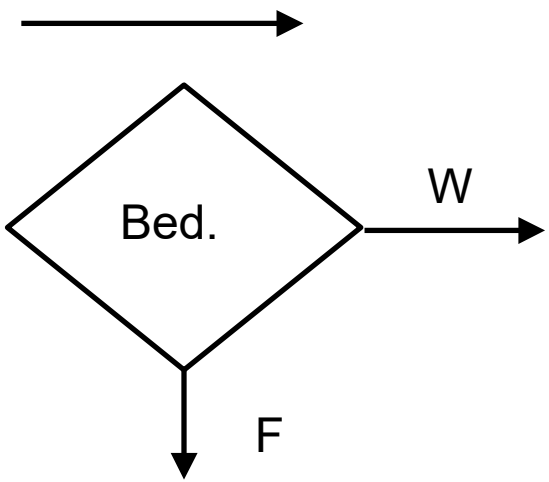
Flussdiagramm

- Instruktive grafische Notation für einen Algorithmus
  - Detaillierte und explizite Vorschrift zur schrittweisen Lösung eines Problems



Flussdiagramm: Symbole





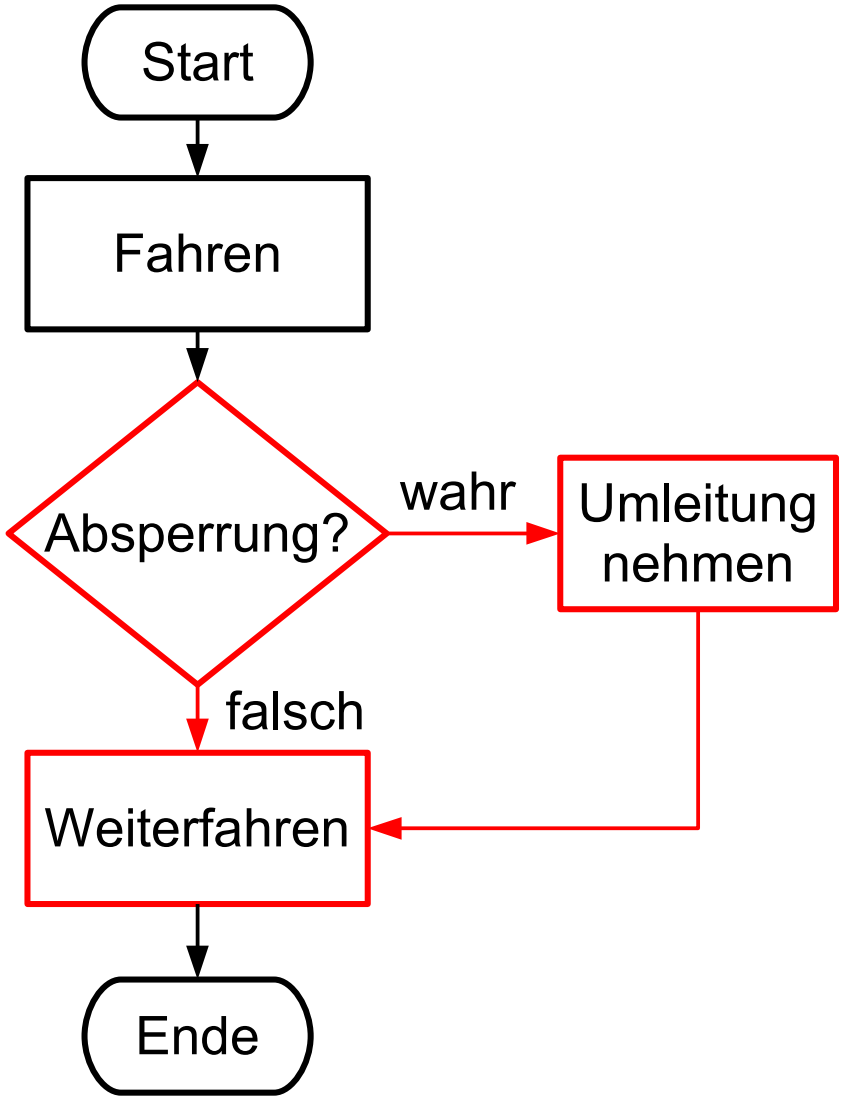
- **Zustand**
  (z. B. Anfangs- oder Endzustand)
- **Aktion** (z. B. Zuweisung)
- **Verbinder** verbindet Zustände und Aktionen; dargestellt durch Pfeil, der auf die nächste auszuführende Aktion zeigt
- **Entscheidung**: Ist Bedingung erfüllt, folge W(ahr)-Zweig, sonst folge F(alsch)-Zweig

Bedingte Anweisung

- Entscheidung wird mittels einer *bedingten Anweisung* (auch: **Alternativanweisung**) durchgeführt
- Es existieren zwei Arten von bedingten Anweisungen:
  - Die **Abzweigung**
  - Die **Verzweigung**

Abzweigung

- Führt einen alternativen Zweig mit einer zusätzlichen Anweisung ein
- Die Ausführung der Anweisung im alternativen "wahr"-Zweig ("Umleitung nehmen") hängt ab von einer vorangestellten Bedingung ("Absperrung?")
- Die Anweisung kann auch eine *Verbundanweisung*, d. h. eine Folge von Anweisungen, sein!

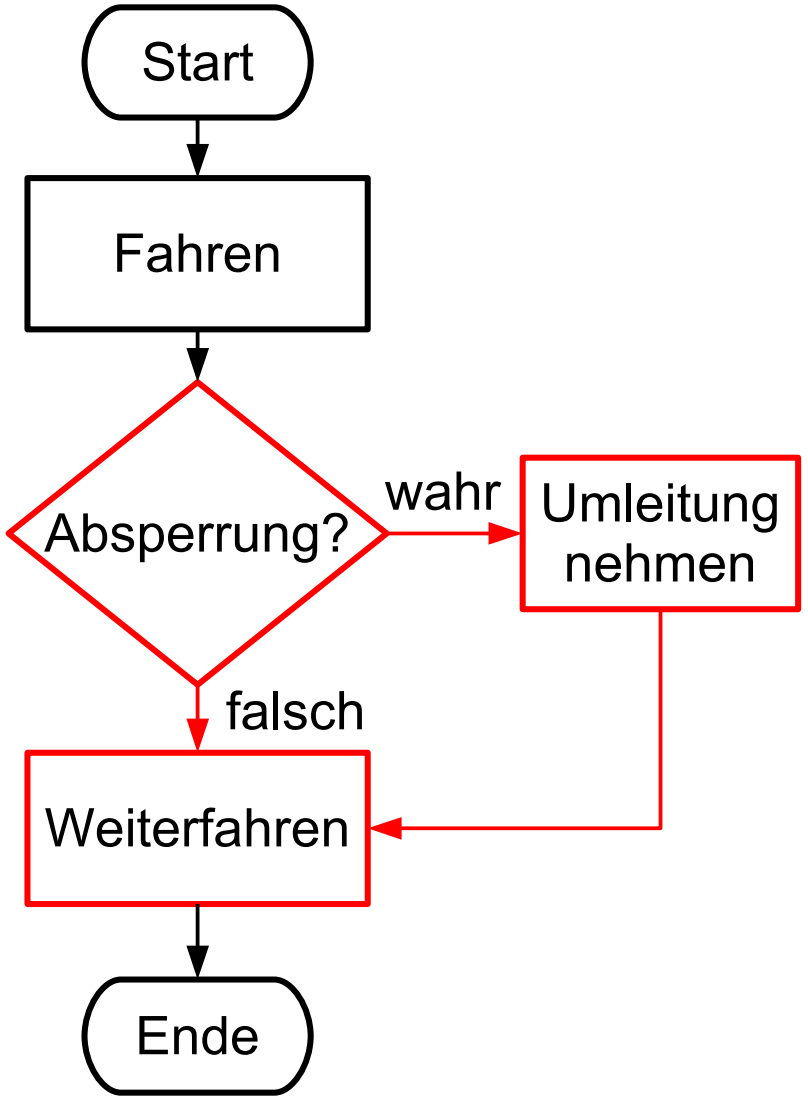


Abzweigung: Syntax

- Syntax: **`if (Bedingung)`**

  `Anweisung;`

  - Bedingung: **Logischer Ausdruck**
    - z. B. **`absperrung == true`**
    - == ist ein Vergleichsoperator
    - Dieser Ausdruck liefert entweder **`true`** ("wahr") oder **`false`** ("falsch")
    - Abhängig vom Wert der logischen Variable **`absperrung`**



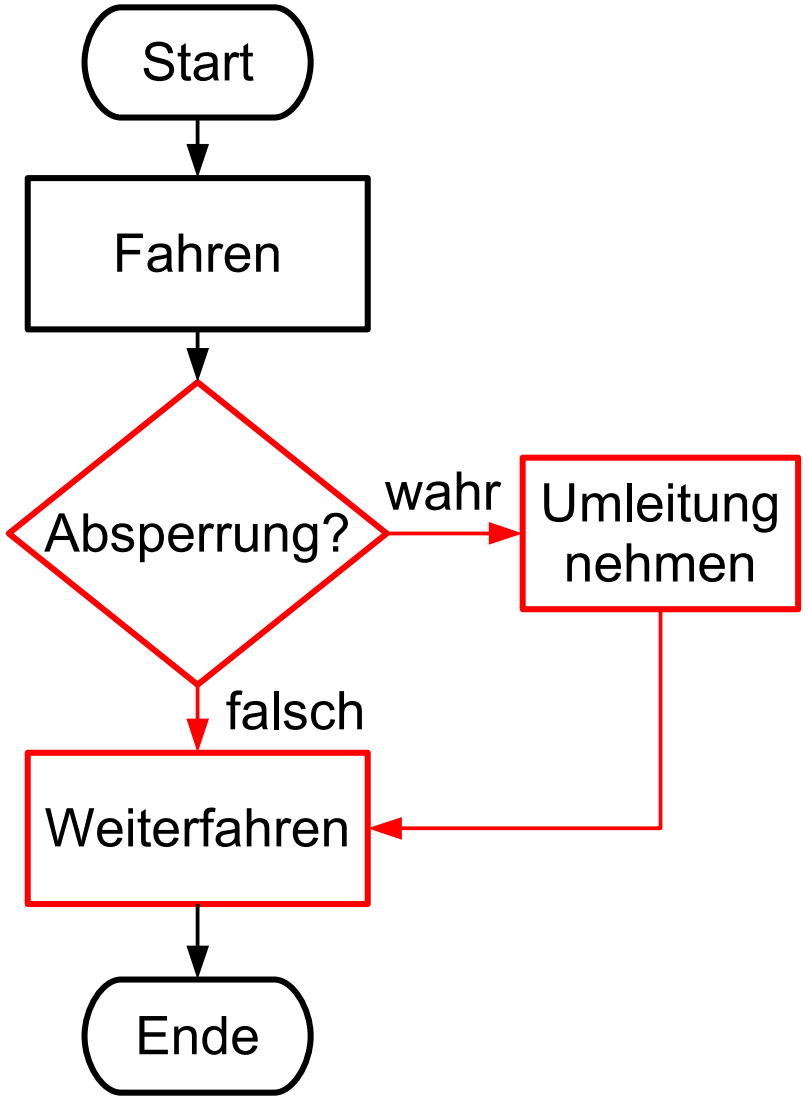
Abzweigung: Codebeispiel

- Beispiel:

```
// Befehle für Fahren

if (absperrung == true)
   // Befehl für Umleitung

// Befehle für Weiterfahren
```



Bedingung mit logischer Variable

- Der Ausdruck **`absperrung == true`** ist genau dann wahr, wenn **`absperrung`** den Wert **`true`** hat
- **`absperrung == true <=> absperrung`**
- Darum reicht es aus, die logische Variable an Stelle des Ausdrucks zu verwenden
- Beispiel:

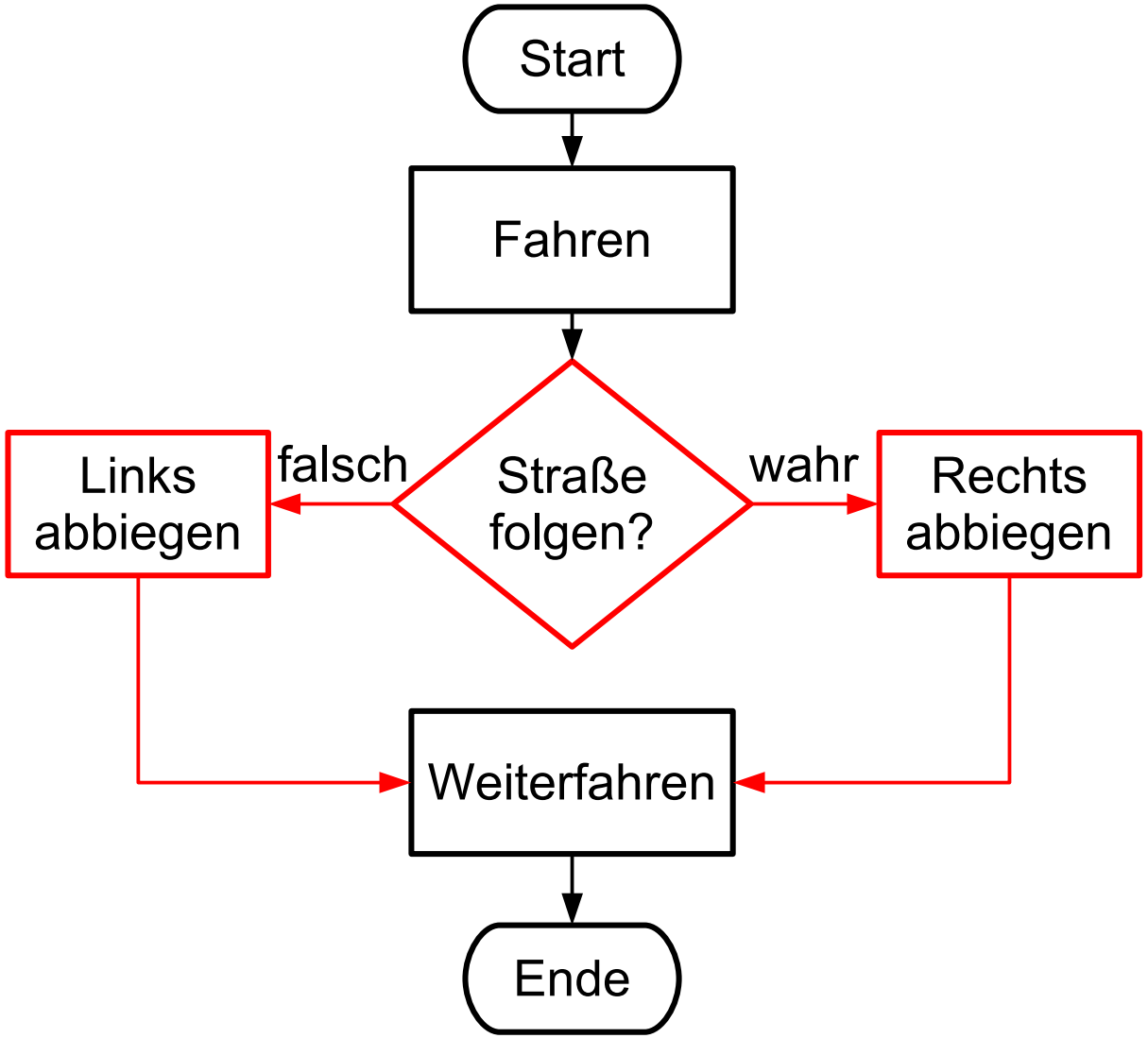
```
// Befehle für Fahren

if (absperrung)
   // Befehl für Umleitung

// Befehle für Weiterfahren
```

Verzweigung

- Bei einer Verzweigung (auch: *Alternative*) wird auch im Falle der nicht erfüllten Bedingung eine zugehörende Anweisung ausgeführt



Verzweigung

- Die Anweisung im **`else`**-Zweig wird nur dann ausgeführt, wenn die Bedingung NICHT erfüllt ist
- **Syntax:**

```
if (Bedingung)
  Anweisung_1;  // wahr-Zweig
else
  Anweisung_2;  // falsch-Zweig
```

Verzweigung mit mehreren Anweisungen

- **Syntax:**

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

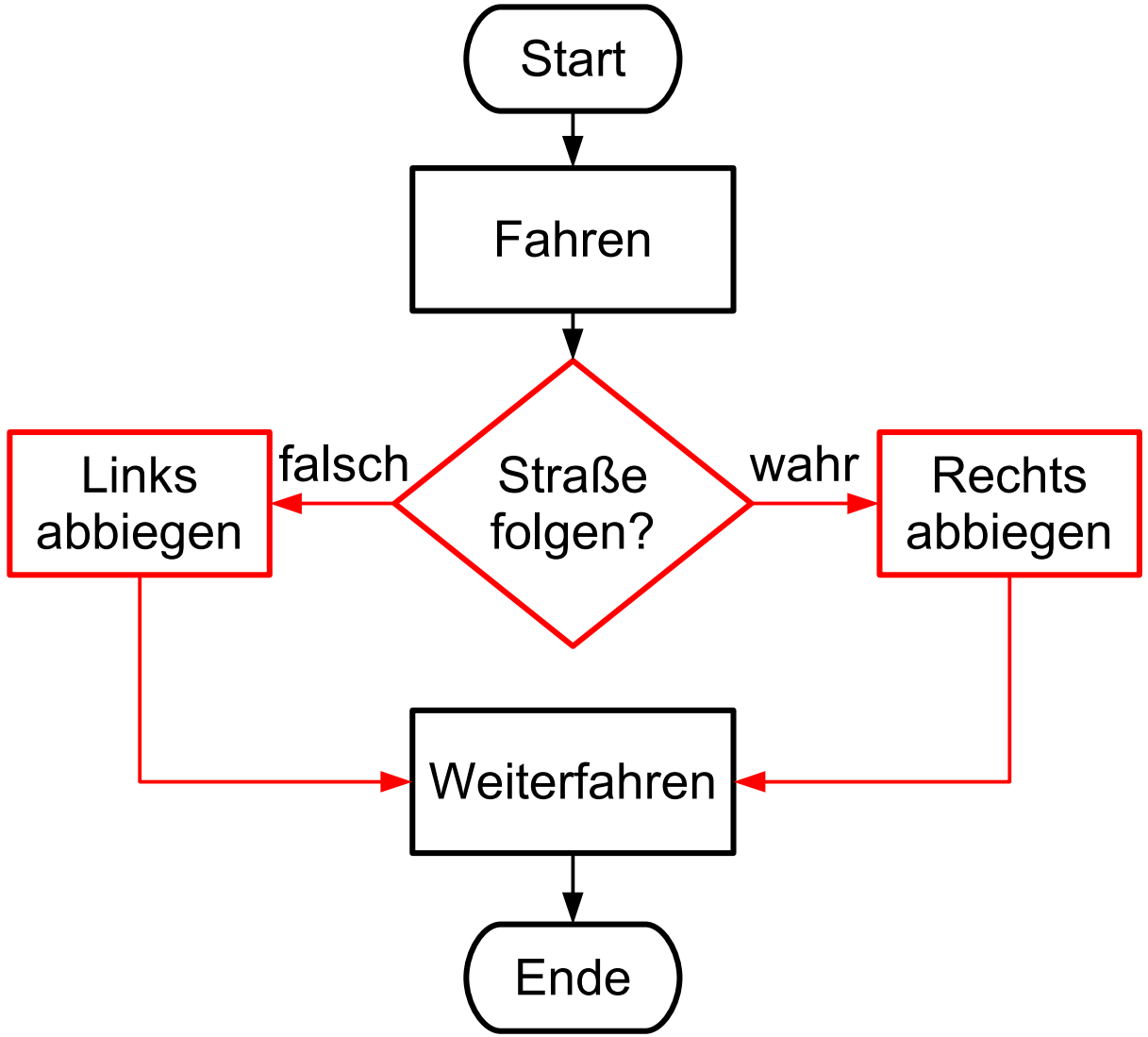
- Geschweifte Klammern legen eine Verbundanweisung, d. h. einen Block, fest

Verzweigung: Codebeispiel

```
// Fahren

if (straßeFolgen)
   /* Rechts
      abbiegen */
else
   /* Links
      abbiegen */

// Weiterfahren
```

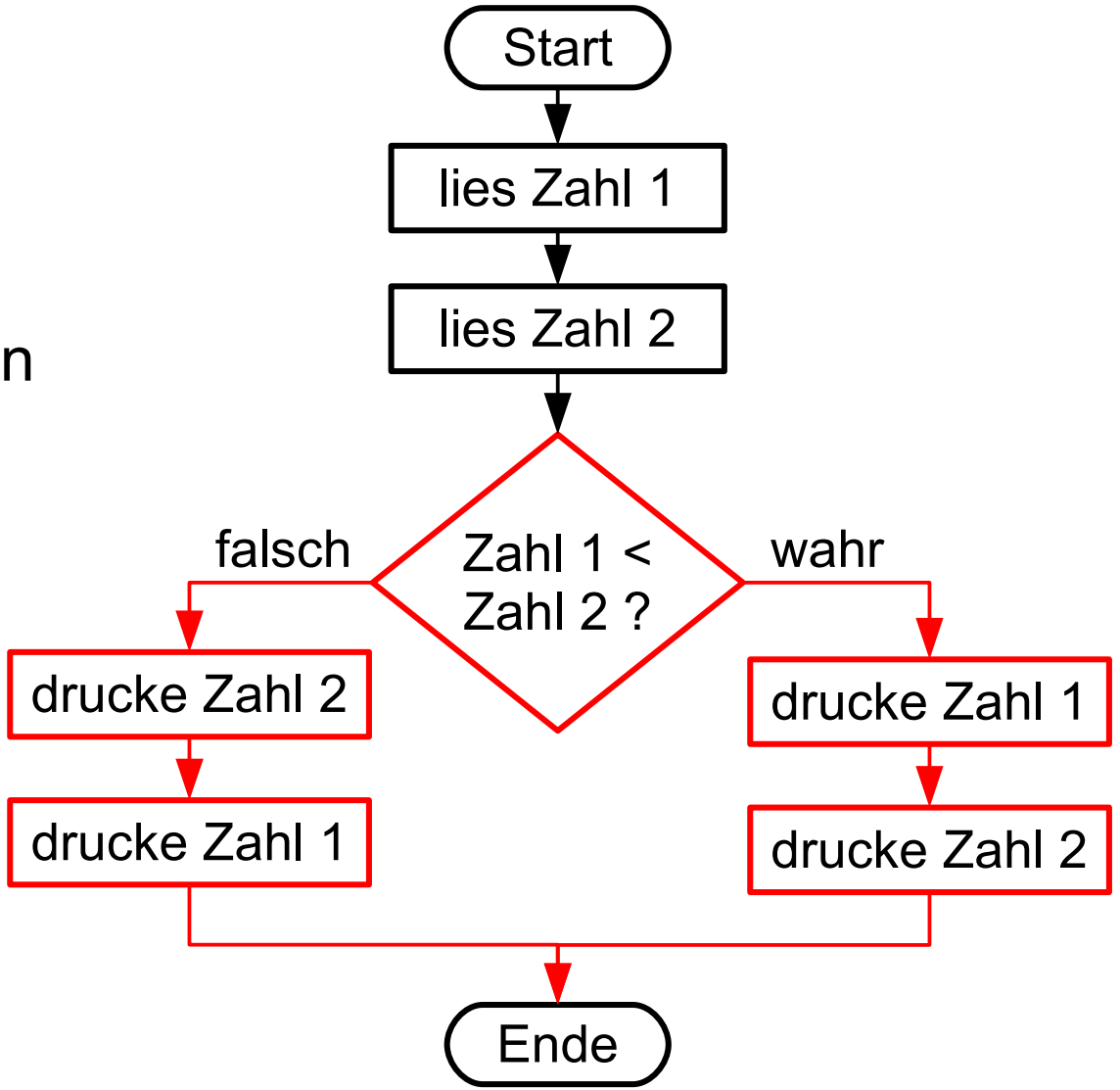


Vergleichsoperatoren

Operator	Bedeutung
A == B	Ist A gleich B ?
A < B	Ist A kleiner B ?
A <= B	Ist A kleiner oder gleich B ?
A > B	Ist A größer B ?
A >= B	Ist A größer oder gleich B ?
A != B	Ist A ungleich B ?

Beispiel: Zwei Zahlen sortieren

- Vergleichsoperatoren können z. B. benutzt werden, um Zahlen zu sortieren
- Hier werden zwei Zahlen in aufsteigender Reihenfolge sortiert:



Implementierung in Processing

```
import template.library.*;

int num1 = IOLib.getInt("First number:");
println("First number: " + num1);
int num2 = IOLib.getInt("Second number:");
println("Second number: " + num2);

if (num1 < num2)
  println("Sorted numbers:\t" + num1 + "\t" + num2);
else
  println("Sorted numbers:\t" + num2 + "\t" + num1);
```

Implementierung mit echtem Tausch

- Hilfsvariable erforderlich

```
// Read input values as before

if (num1 > num2) {
   tmp = num1;
   num1 = num2;
   num2 = tmp;
}

println(num1 + "\t" + num2);
```

tmp	num1	num2
7	7	1
7	1	1
7	1	7

Größenvergleiche und Ausdrücke

- Hinsichtlich der Auswertung eines Ausdrucks ist der Rang eines Vergleichsoperators kleiner als der eines arithmetischen Operators
- Beispiel:
  - 17+4 > 16
  - Hier wird zunächst 17+4 zu 21 ausgewertet, da der Rang der Addition größer ist, als der des Größer-Operators
  - Danach wird 21 mit 16 verglichen
  - Dieser verbleibende Ausdruck liefert **`true`**

Aussagenlogik

- Entwickelt vom Engländer George Boole (1815-1864)
  - *"An investigation into the laws of thought"*
- Ziel: Formale Begründung der Logik
- Aussagenwerte sollten berechenbar werden wie Zahlenwerte, z. B. der Addition oder der Multiplikation
  - Boolsche Algebra

Boolsche Algebra

- Die Wahrheit oder Falschheit einer zusammengesetzten Aussage kann aus dem *Wahrheitswert* ihrer *atomischen Aussagen* und den verknüpfenden *logischen Operationen* abgeleitet werden
- Wahrheitswerte: wahr (**`true`**), falsch (**`false`**)
- Boolsche Operationen:

Operator	Bedeutung
!A	Negation
A && B	Und-Verknüpfung
A || B	Oder-Verknüpfung

Wahrheitstabellen

- Das Ergebnis logischer Operationen wird mittels *Wahrheitstabellen* exakt beschrieben
  - In diesen ist jeder möglichen Kombination der Wahrheitswerte der atomischen Aussagen der Wahrheitswert der logischen Operation zugeordnet

A	B	!A	A&&B	A||B
`false`	`false`	`true`	`false`	`false`
`false`	`true`	`true`	`false`	`true`
`true`	`false`	`false`	`false`	`true`
`true`	`true`	`false`	`true`	`true`

Boolsche Operationen: Beispiel

Operator	Bedeutung
!A	Negation
A && B	Und-Verknüpfung
A || B	Oder-Verknüpfung

- **Beispiel:** Auto mieten (vgl. B. Kjell):
  - **Bedingung 1:** Mieter muss mindestens 21 Jahre alt sein
  - **Bedingung 2:** Mieter muss über eine Kreditkarte mit einem Kreditrahmen von min. 10000$ verfügen
  - Beide Bedingungen müssen erfüllt sein
- Logische Operatoren werden benutzt, um Bedingungen zu einer Gesamtbedingung zu verknüpfen

Beispiel: UND-Verknüpfung

```
// get the age of the renter
int age = IOLib.getInt("How old are you?");

// get the credit line
int credit = IOLib.getInt("How much credit do you have?");

// check that both qualifications are met
if (age>=21 && credit>=10000)

  println("Enough to rent this car!");

else

  println("How about a bicycle?");
```

Beispiel: ODER-Verknüpfung

- **Beispiel: Auto kaufen**
  - Um Auto kaufen zu können sind entweder 25000$ in bar oder ein Kredit in gleicher Höhe erforderlich

```
// get the cash
int cash = IOLib.getInt("How much cash do you have?");
// get the credit line
int credit = IOLib.getInt("How much credit do you have?");

// check that at least one qualification is met
if (cash >= 25000  ||  credit >= 25000)
   println("Enough to buy this car!");
else
   println("How about a bicycle?");
```

Beispiel: Prüfen, ob Ausdruck nahezu gleich 0

- Vergleich auf Gleichheit mit Datentypen **`float`** und **`double`** vermeiden!
- Toleranzwert verwenden
- An Stelle von A==B folgenden Ausdruck auswerten:
  - **Lösung 1**: **`(-0.0001 < A-B) && (A-B < 0.0001)`**
- Alternative: Mathematischen Betrag benutzen:
  - |A-B| < Toleranzwert
  - **Lösung 2**: **`Math.abs(A-B) < 0.0001`**
- Toleranzwert hier 0,0001

Lösung mit logischem UND

```
double var1 = IOLib.getDouble("var1?");
double var2 = IOLib.getDouble("var2?");


double diff = var1 - var2;

if (-0.0001 < diff && diff < 0.0001)
  println("Differenz quasi 0");
else
  println("Differenz größer 0");
```

Präzedenz

Vorzeichenoperator	+/-
Multiplikative Operatoren	* / %
Additive Operatoren	+ -
Vergleichsoperatoren	< <= > >=
Test auf Gleichheit	== !=
Logisches UND, ODER	&& ||
Zuweisungsoperator	=

Verkürzte Auswertung

- Logische Ausdrücke werden verkürzt ausgewertet
  - (engl.: *short circuit evaluation*)
- D. h. Teilausdrücke des Ausdrucks werden nur solange ausgewertet, wie Wahrheitswert des gesamten Ausdrucks noch nicht feststeht
  - Spart unnütze Berechnung von Teilausdrücken und ermöglicht so eine Verringerung des Rechenaufwands
  - Beispiel:

```
if (age >= 21 && credit >= 10000)
  println("Enough to rent this car!");
else
  println("How about a bicycle?");
```

Beispiel: „Robuste“ Programmierung

- Die folgende Programmanweisung kann wegen der praktizierten verkürzten Auswertung für alle Werte von **`nenner`** ausgeführt werden:

```
if ((nenner != 0) && (zaehler / nenner > 1))
   // tue etwas...
else
   // Fehler?
```

- Ohne **`(nenner != 0)`** könnte der Ausdruck zum Programmabsturz führen!

Bitoperatoren

- Negationsoperator: ~a („Tilde“)
  - Invertiert jedes Bit des Arguments a
- Logisches Und: a & b
- Logisches Oder: a | b
- Exklusives Oder: a ^ b („Zirkumflex“)
- Realisieren jeweils bitweise das logische Und, Oder und Exklusives Oder
- Beispiele:
  - `println(~1);  // int: 0...01 → 1...10 = -2`
  - `println(1|3); // int: 0...01 | 0...11 → 0...11 = 3`
  - `println(1^3); // int: 0...01 | 0...11 → 0...10 = 2`
  - `println(1&3); // int: 0...01 | 0...11 → 0...01 = 1`

Geschachtelte Fallunterscheidungen

Geschachtelte Fallunterscheidungen

```
if (Bedingung)
  Anweisung_1;
else
  Anweisung_2;
```

- **`Anweisung_i`** kann auch eine **`if`**-Anweisung sein!
- In diesem Fall entsteht eine geschachtelte **`if`**-Anweisung

```
if (Bedingung_1)
  if (Bedingung_2)
    Anweisung_1;
```

- Tiefe der Schachtelung beliebig

Beispiel: Identifikation von Personengruppen

```
int country = IOLib.getInt("Your country (D: 49, UK: 44 etc.)");

int gender = IOLib.getInt("Your gender (male=0, female=1):");

int age = IOLib.getInt("Your age:");


// Fortsetzung nächste Folie
```

Beispiel: Identifikation von Personengruppen

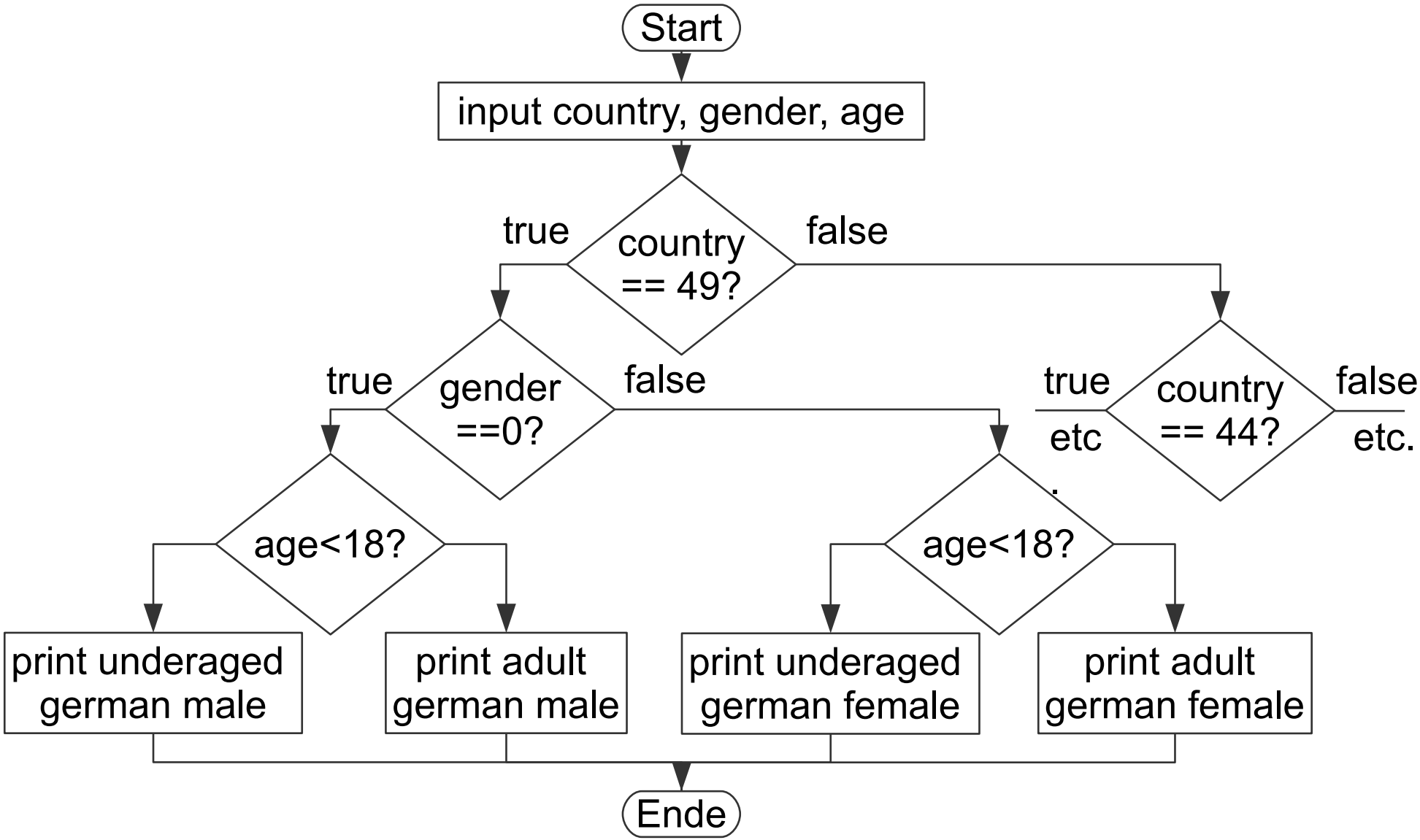
```
if (country == 49) // Germany
  if (gender == 0) // male
    if (age < 18)
      println("You are an underaged german male");
    else
      println("You are an adult german male");
  else // female
    if (age < 18)
      println("You are an underaged german female");
    else
      println("You are an adult german female");

// Fortsetzung nächste Folie
```

Beispiel: Identifikation von Personengruppen

```
// Abfragen für andere Länder
else if (country == 44)
  // Programmcode wie oben, aber angepasst an UK
else // Es wurde kein bekannter Ländercode eingegeben
  println("Unknown country. Please try again.");
```

Als Flussdiagramm



dangling-else-Problem (1)

```
if (Bedingung_1)
  if (Bedingung_2)
    Anweisung_1;
  else
    Anweisung_2;
```

- Zu welchem `if` gehört das `else`?
- Das `else` ergänzt immer das **letzte** `if`

dangling-else-Problem (2)

- Falls **`else`** zum ersten **`if`** gehören soll, kann dies mittels eines Blocks erzwungen werden:

```
if (Bedingung_1) {
   if (Bedingung_2)
     Anweisung_1;
}
else
   Anweisung_2;
```

Getrennte Lösung

- Wenn möglich sind getrennte Lösungen geschachtelten Lösungen vorzuziehen
- Einfacher lesbarer Code
- Oftmals ist die erste Lösung eine geschachtelte Lösung
- Enthält diese Coderedundanzen, ist es i. d. R. möglich, eine einfachere Lösung anzugeben

Beispiel

- Hier verstecken sich Redundanzen:

```
if (country == 49) // Germany
  if (gender == 0)  // male
    if (age < 18)
      println("You are an underaged german male");
    else
      println("You are an adult german male");
```

- Tritt etwas im **`if`**- und **`else`**-Teil auf, so kann es vor oder ggf. auch hinter die **`if`**-Anweisung gezogen werden

Getrennte Lösung ohne Redundanzen

```
String output = "You are an";

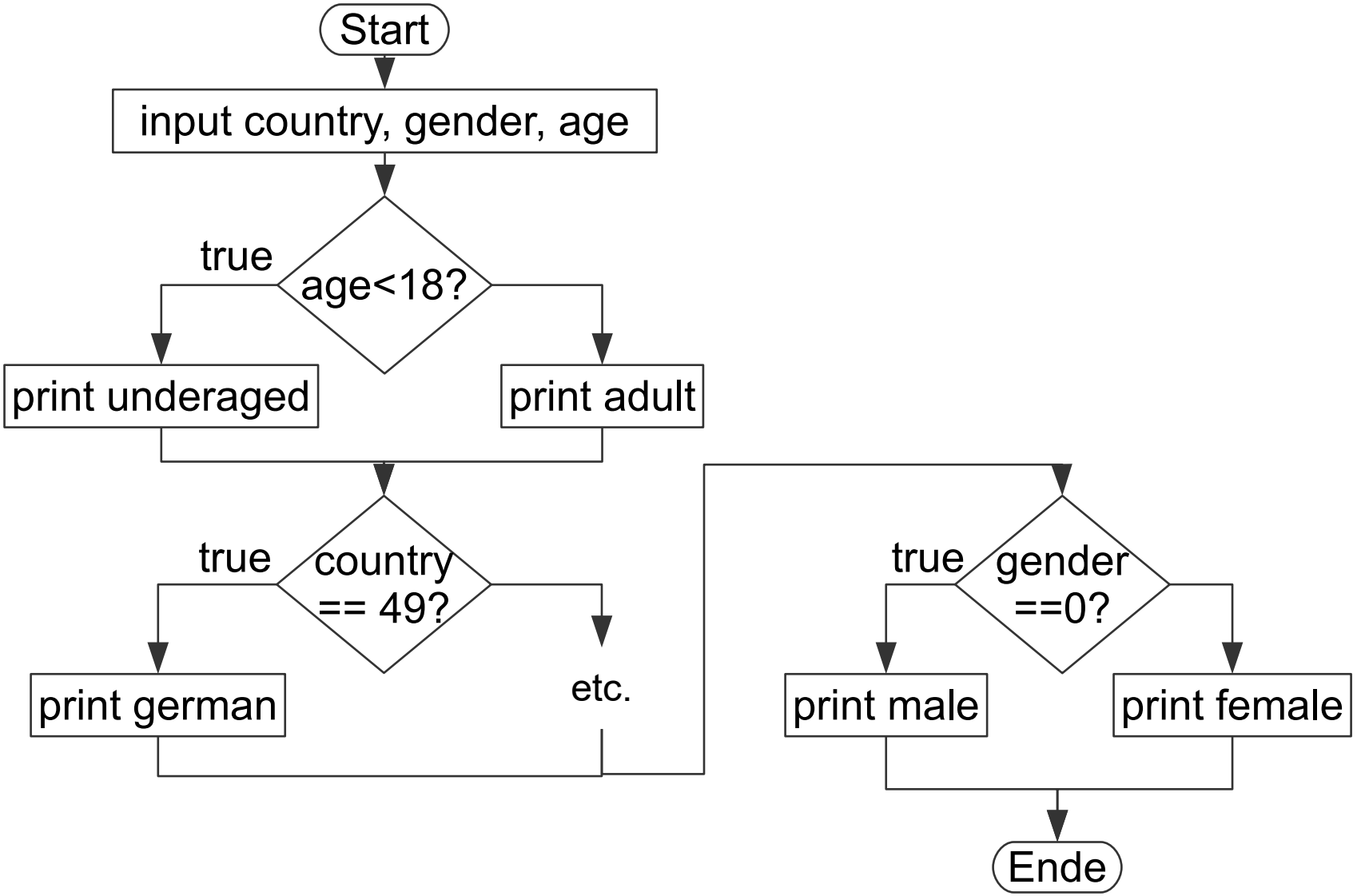
if (age < 18)
  output = output + " underaged ";
else
  output = output + " adult ";

if (country == 49)
  output = output + "german ";
// else Abfragen fuer andere Laender

if (gender == 0)
  output = output + "male";
else
  output = output + "female";

println(output);
```

Als Flussdiagramm



switch-Anweisung

Viele Fälle

- Viele verschiedene Fälle können zu starker Verzweigung und damit zu unübersichtlichem Code führen
- Beispiel: Länder unterscheiden

```
int country =
  IOLib.getInt("Your country (D: 49, UK: 44 etc.): ");

if (country == 49) println("Germany");
else if (country == 46) println("Sweden");
else if (country == 44) println("UK");
else if (country == 33) println("France");
else // Es wurde kein bekannter Ländercode eingegeben
  println("Unknown country");
```

Alternative: switch-Anweisung

- Die möglichen Fälle werden in der **`switch`**-Anweisung durch **`case`**-Anweisungen zusammengefasst
- Beispiel:

```
int country =
  IOLib.getInt("Your country (D: 49, UK: 44 etc.): ");

switch(country) {
  case 49: println("Germany"); break;
  case 46: println("Sweden"); break;
  case 44: println("UK"); break;
  case 33: println("France"); break;
  default: println("Unknown country");
}
```

- Die Anweisungen unter einem **`case`**-Fall werden durch die **`break`**-Anweisung abgeschlossen

switch-Anweisung: `break` (1)

- Sobald ein Fall zutrifft, werden ohne **`break`** alle weiteren **`case`**-Anweisungen ebenfalls ausgeführt
  - Beispiel: **`country`** ist **`46`**. Dann werden die Fälle **`46`**, **`44`**, **`33`** und **`default`** ausgeführt.

```
int country =
  IOLib.getInt("Your country (D: 49, UK: 44 etc.): ");

switch(country) {
  case 49: println("Germany"); // cases
  case 46: println("Sweden");  // ohne breaks
  case 44: println("UK");
  case 33: println("France");
  default: println("Unknown country");
}
```

switch-Anweisung: `break` (2)

- Durch das selektive Einfügen von **`breaks`** lassen sich Fälle zusammenfassen
  - Damit geht die Funktion von **`switch`** über einfache **`if-else`**-Verzweigungen hinaus
  - Beispiel: Ländergruppen. Nur einige **`case`**-Anweisungen besitzen **`breaks`**

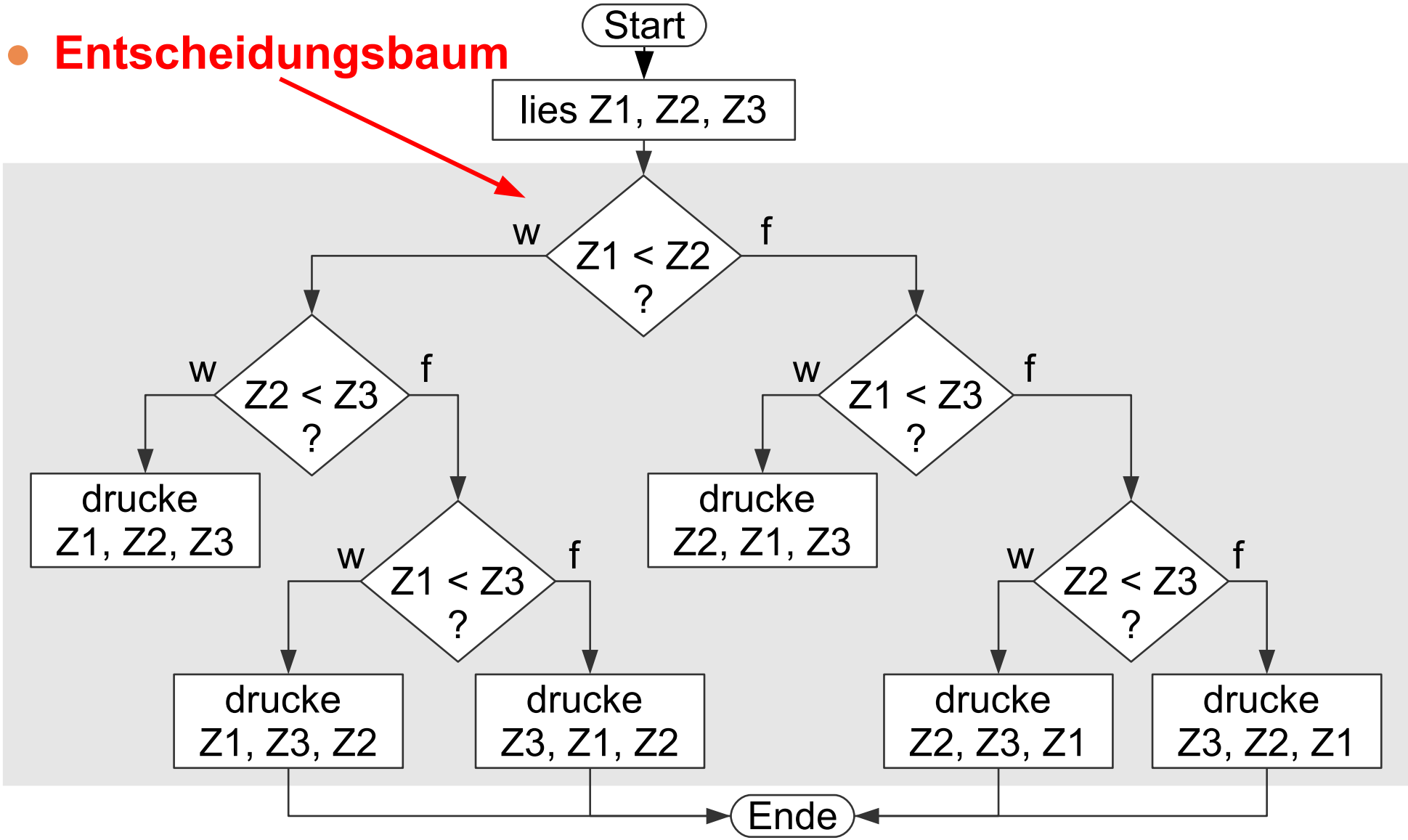
```
int country =
  IOLib.getInt("Your country (D: 49, UK: 44 etc.): ");

switch(country) {
  case 49:
  case 46: println("Country group 1"); break;
  case 44:
  case 33: println("Country group 2"); break;
  default: println("Unknown country");
}
```

Bäume

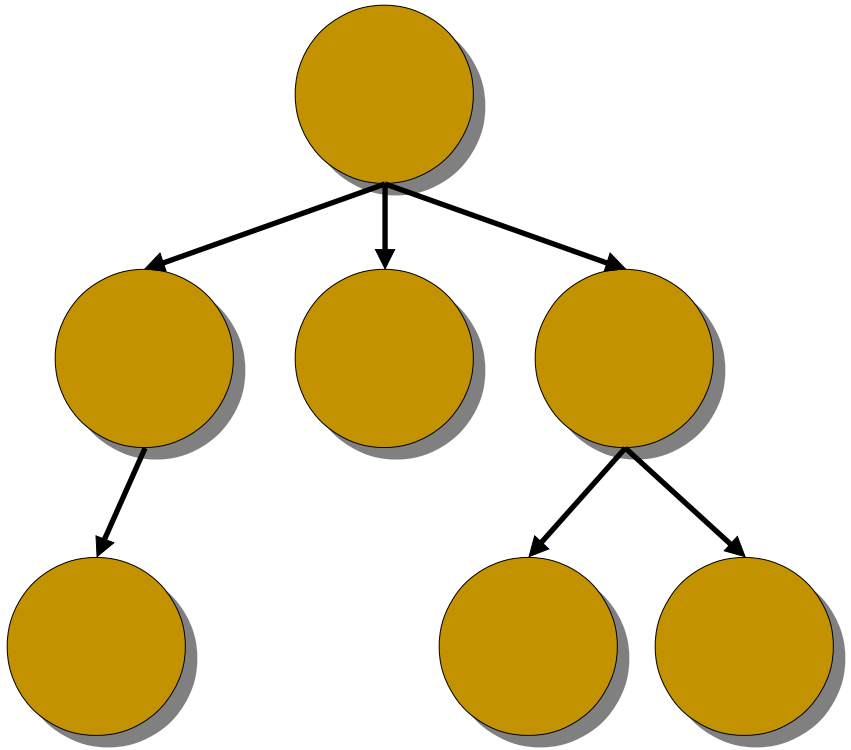
Beispiel: 3 Zahlen sortieren

- **Entscheidungsbaum**



Baum: Bestandteile

- Ein Baum besteht aus einer Menge von Knoten, die untereinander durch Kanten verbunden sind
- **Knoten**
  - Dargestellt durch Punkte, Kreise, Rechtecke etc.
- **Kanten**
  - Repräsentiert durch Pfeile oder Linien
  - Pfeil bzw. „gerichtete Kante“ legt Hierarchie fest

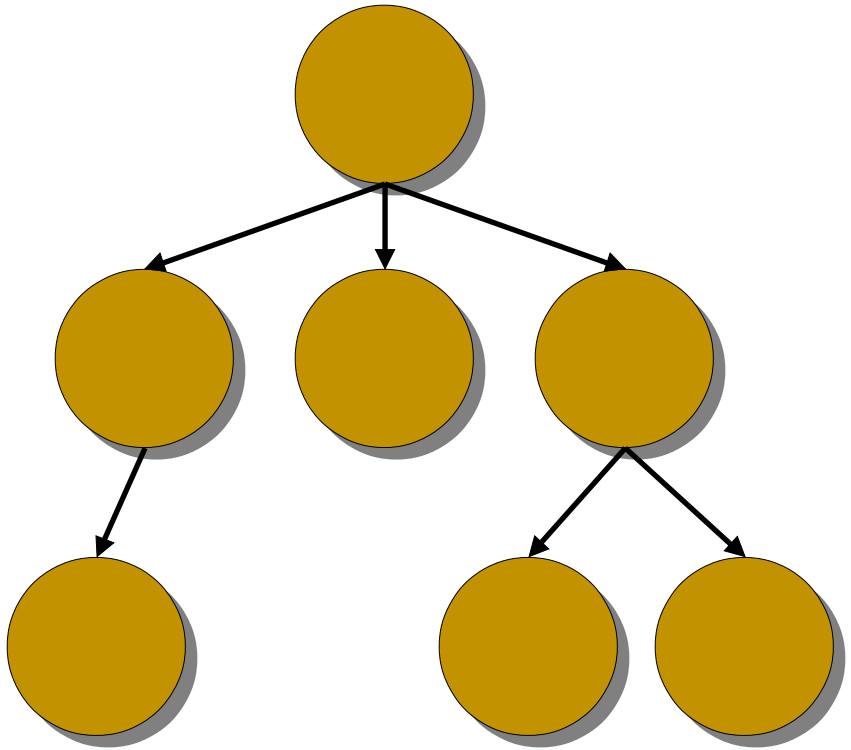


Beispiele für Bäume

- Dateibaum
  - Knoten sind Verzeichnisse oder Dateien
  - Kanten führen von einem Verzeichnis zu dessen Unterverzeichnis(sen) oder zu den Unterdateien
- Stammbaum
  - Knoten sind Väter/Mütter; Kanten führen zu den Kindern
- Spielbaum
  - Knoten sind Spielpositionen auf dem Brett (z. B. Schach)
  - Kanten führen zu allen möglichen nächsten Spielpositionen
- Buch: Kapitel -> Abschnitt -> Absatz
- Falteneditor bzw. Outline-Darstellung in Textverarbeitung

Baum

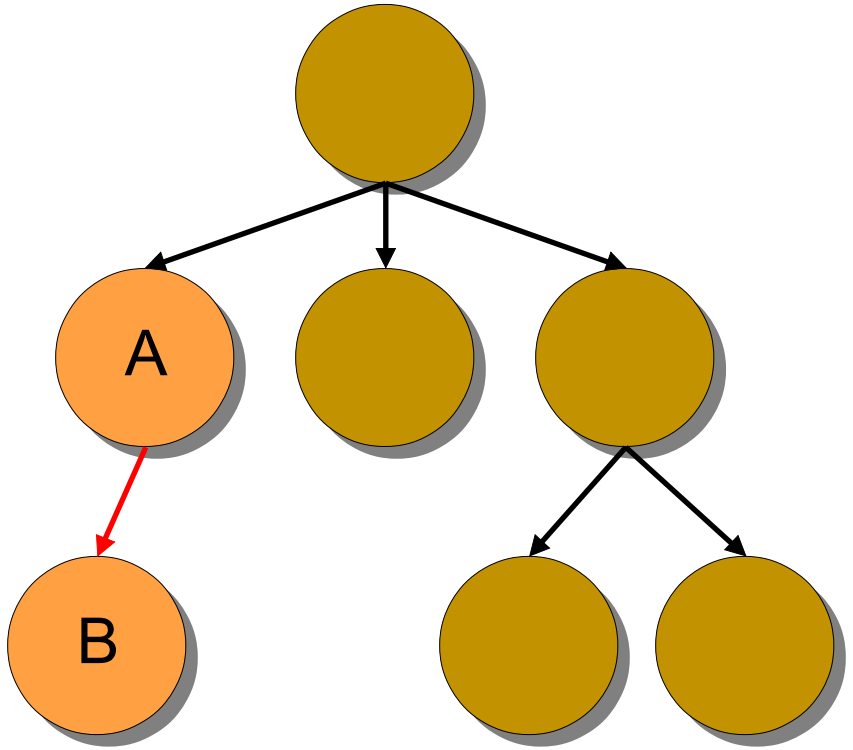
- Fundamentale **Datenstruktur** der Informatik
  - Datenstruktur: Objekt zur Speicherung und Organisation von Daten
- In Bäumen können **Daten** und ihre **Beziehungen** untereinander gemäß einer **Ordnungsrelation** gespeichert werden



Baum: Vorgänger und Nachfolger

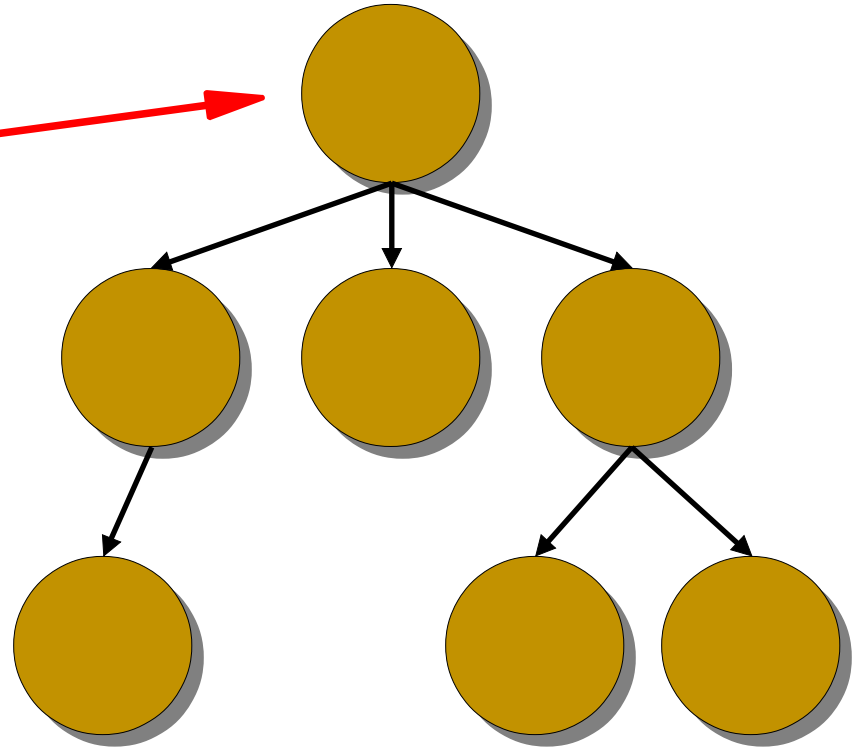
- Führt von einem Knoten A zu einem Knoten B eine Kante, so wird A als **Vorgänger** (engl.: *predecessor*) von B bezeichnet und B als **Nachfolger** (engl. *successor*) von A

- Dies wird als **A → B** geschrieben



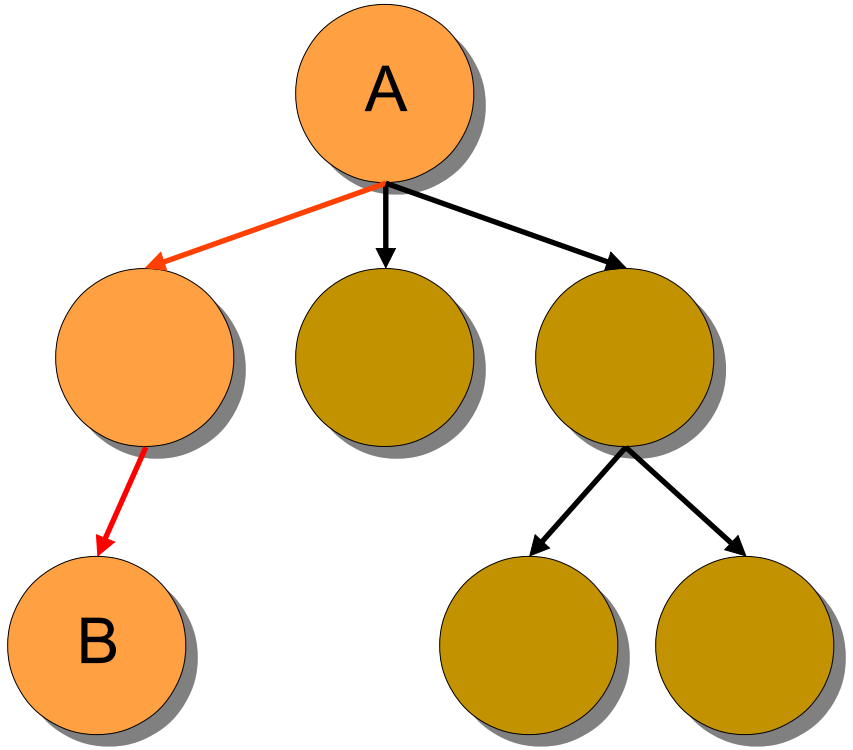
Baum: Wurzel und Blätter

- Ein Baum besitzt genau einen Knoten ohne Vorgänger, genannt **Wurzel** (engl. *root*) des Baums
- Knoten ohne Nachfolger heißen **Blätter** (engl. *leafs*)
- Alle anderen Knoten heißen **innere Knoten** des Baums (engl. (*inner*) *nodes*)



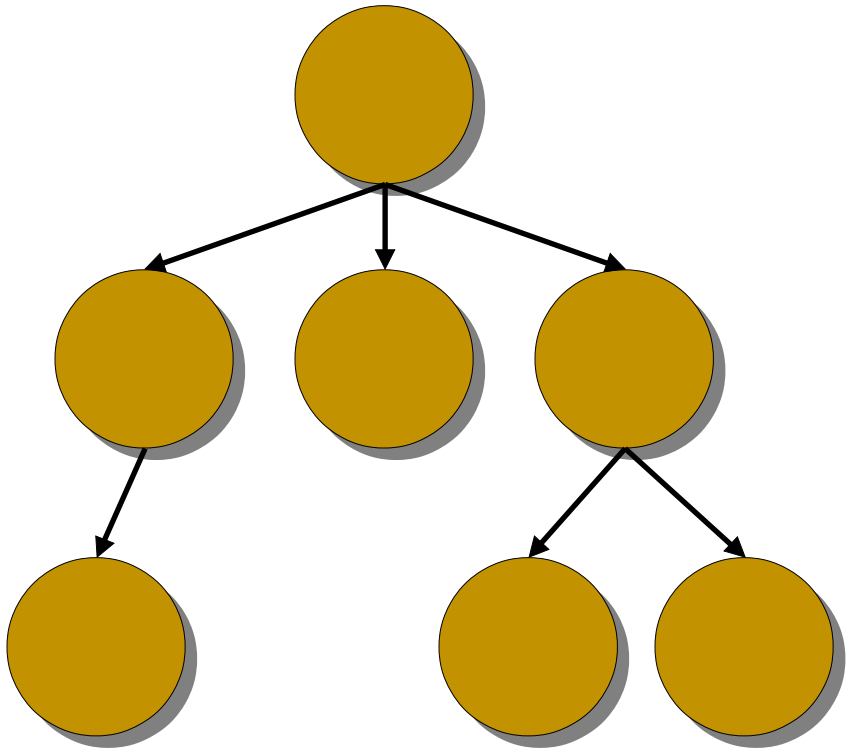
Baum: Pfad

- Ein **Pfad** von A nach B ist eine Folge von Knoten und Kanten, die von A nach B führen
- **A → X1 → X2 → ... → B**
- Die **Länge** eines Pfades ist definiert als die Anzahl der zum Pfad gehörenden Knoten
- Gibt es einen Pfad von A nach B, so heißt B **Nachkomme** (engl.: *descendant*) von A und A **Vorfahre** (engl.: *ancestor*) von B



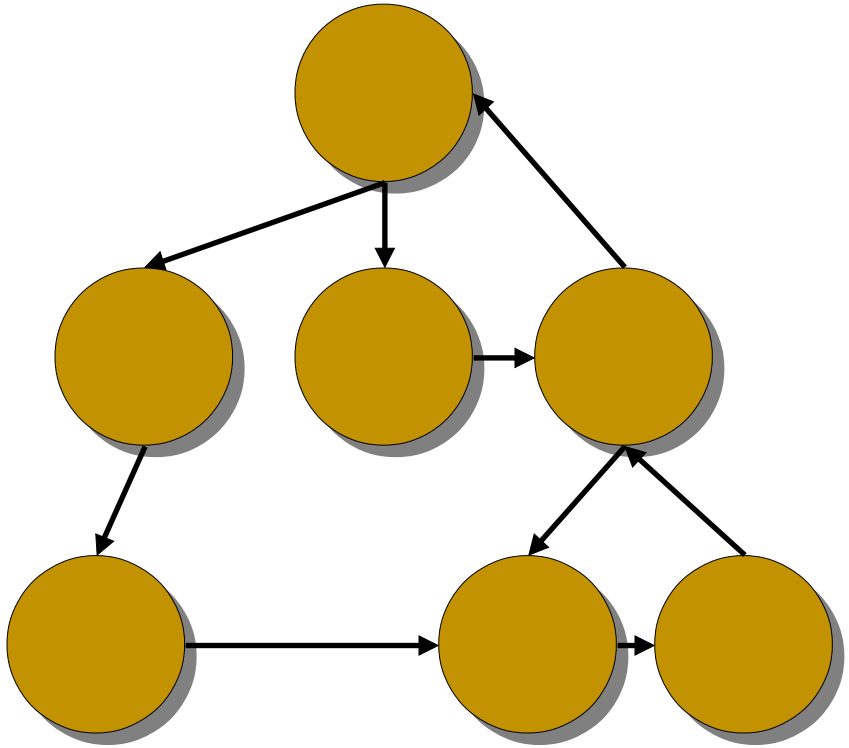
Unterbaum

- Die Nachkommen eines beliebigen Knotens K, zusammen mit allen ererbten Kanten, bilden einen Baum mit K als Wurzel, den **Unterbaum** mit Wurzel K



Weitere Eigenschaften von Bäumen

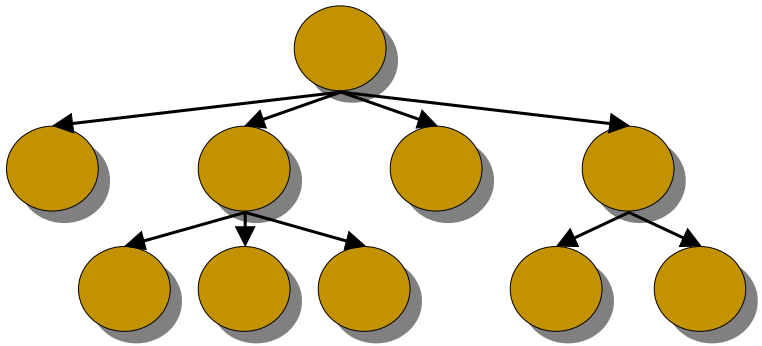
- Mit Ausnahme der Wurzel hat jeder Knoten genau einen Vorgänger
- Daraus folgt:
- Bäume sind **azyklisch**, d. h. es gibt keine zyklischen Pfade in Bäumen
  - **Zyklischer Pfad**: Ein Pfad, der zu seinem Anfangsknoten zurückführt
  - Eltern haben keine Kinder, die gleichzeitig die Eltern der Eltern sind...



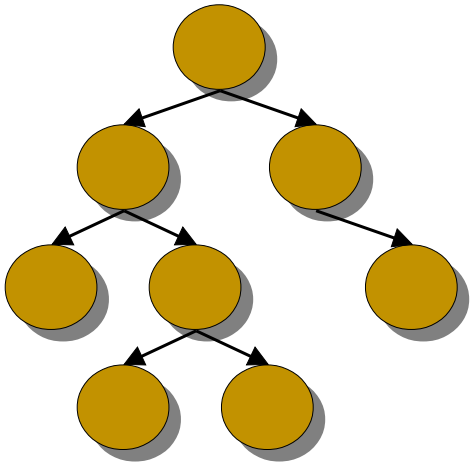
- Von der Wurzel gibt es zu jedem anderen Knoten genau einen Pfad

Baumarten

- Ein Baum heißt ***t*-är**, wenn jeder Knoten höchstens *t* Nachfolger hat
- Ein Baum heißt **binär**, wenn jeder Knoten höchstens 2 Nachfolger hat
  - Binäre Bäume sind also 2-är



4-ärer Baum



binärer (2-ärer) Baum

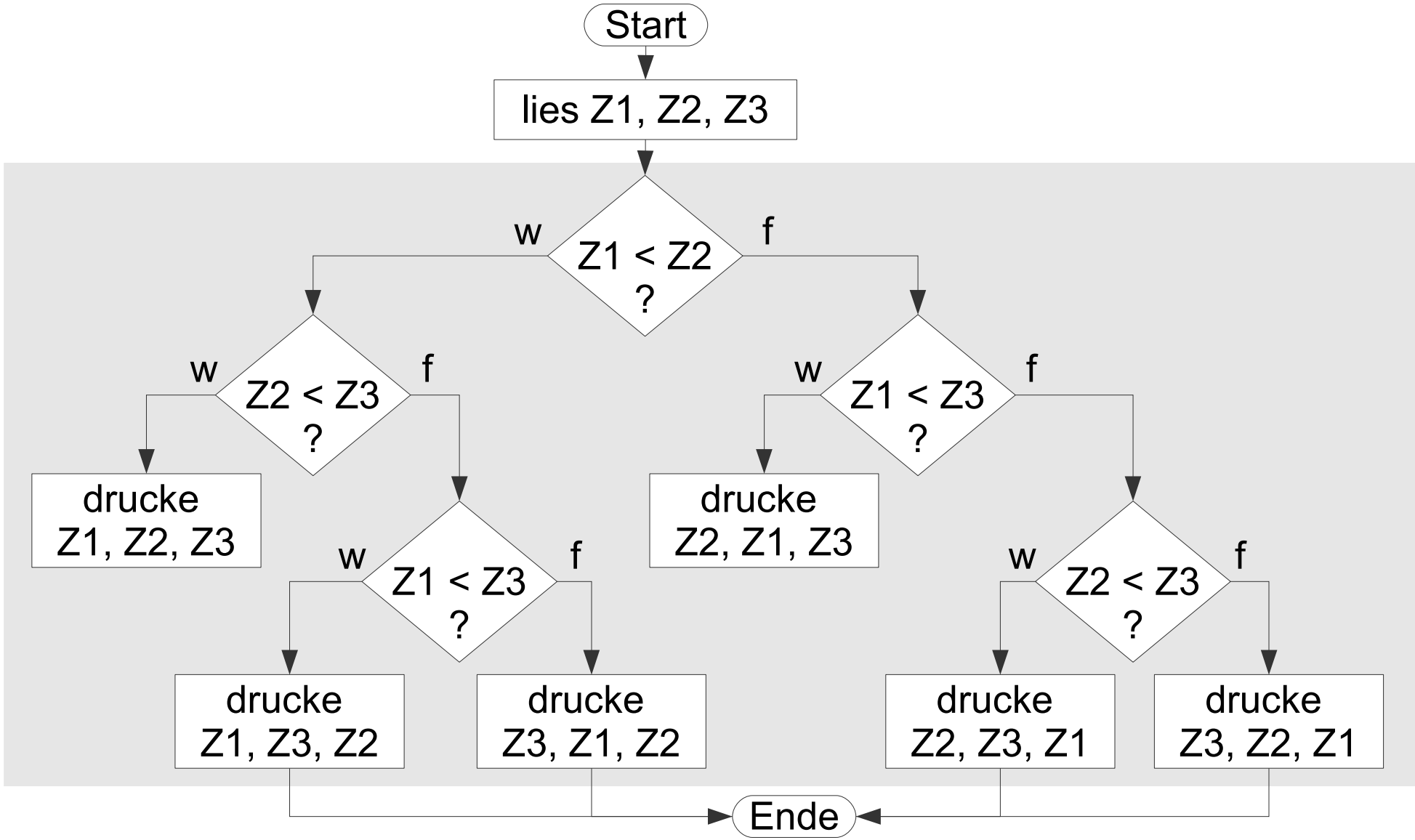
Geordnete Bäume

- Ein Baum ist geordnet, wenn für die höchstens *t* Nachfolger eines Knotens feststeht, welcher der erste, zweite, dritte, etc. Nachfolger ist

- Speziell ein binärer Baum ist geordnet, wenn für die höchstens zwei Nachfolger eines Knotens feststeht, welcher der erste (i. d. R. linke) und welcher der zweite (i. d. R. rechte) Nachfolger ist

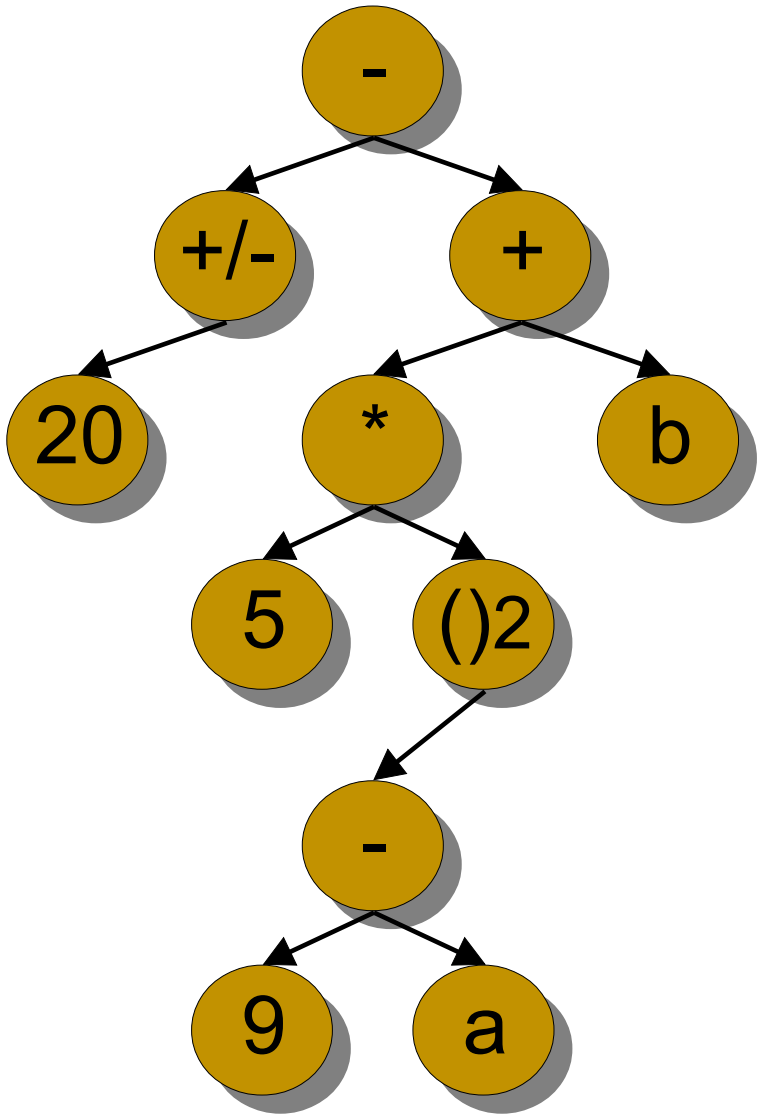
- Der Entscheidungsbaum aus dem Beispiel ist also ein geordneter binärer Baum

Entscheidungsbaum als geordneter binärer Baum



Repräsentation von Ausdrücken

- Beispiel: (-20)-(5*(9-a)^2+b)
  - Blätter repräsentieren Werte und Variablen
  - Alle übrigen Knoten repräsentieren Operatoren
  - Kanten verknüpfen Operatoren mit Teilausdrücken bzw. Operanden



Übung

- Stellen Sie den folgenden Ausdruck mit Hilfe eines Baums dar:
- $10*(a+2b-c)^3+20-d$

Schleifenanweisungen

Zählschleifen

Ein Zähler als Programm (1)

```
// Nachfolgend werden die Zahlen 1..10 ausgegeben
println("1");
println("2");
println("3");
println("4");
println("5");
println("6");
println("7");
println("8");
println("9");
println("10");
```

- **Nachteil:** Soll eine andere Menge von Zahlen aufgezählt werden, so ist es erforderlich, neuen Programmcode zu schreiben

Ein Zähler als Programm (2)

```
// Nachfolgend werden die Zahlen 1..10 ausgegeben
println("1");
println("2");
println("3");
println("4");
println("5");
println("6");
println("7");
println("8");
println("9");
println("10");
```

- Es fällt auf, dass prinzipiell zehnmal die gleiche Anweisung ausgeführt wird
- Jedes Mal mit einem neuen Argument: 1..10
- Das Argument lässt sich berechnen:
- $arg_1 = 1$
- $arg_{i+1} = arg_i + 1$

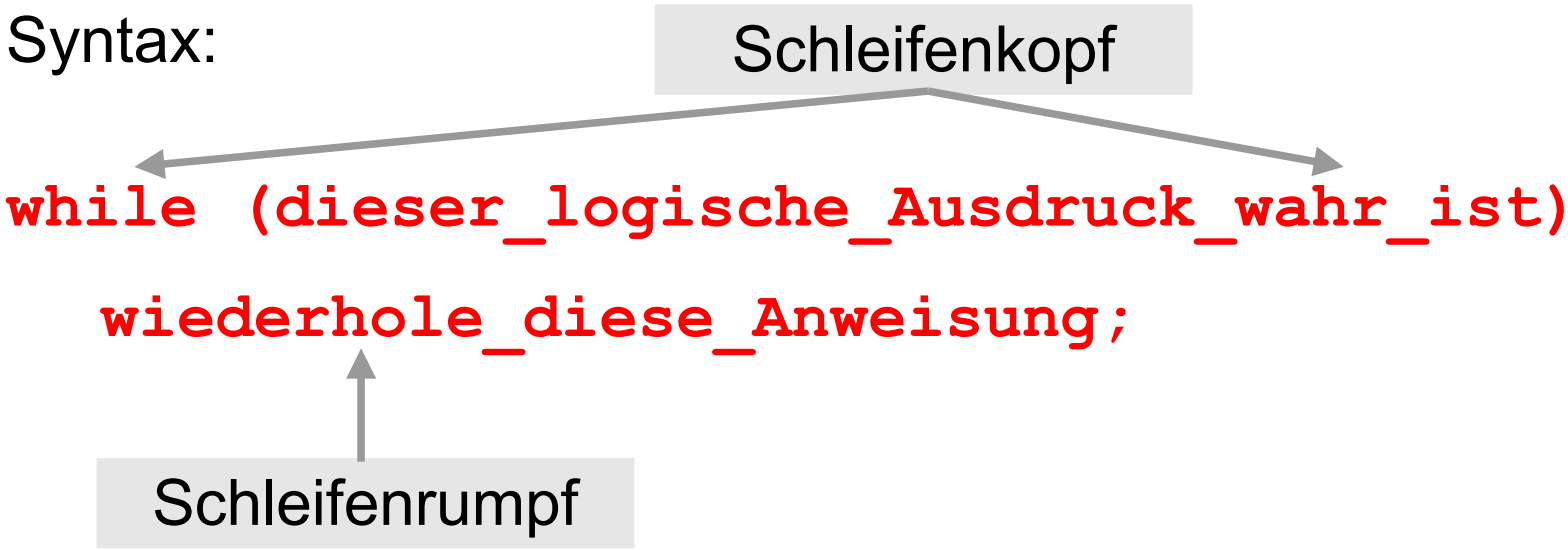
Ein Zähler als Programm (3)

```
// Nachfolgend werden die Zahlen 1..10 ausgegeben
println("1");
println("2");
println("3");
println("4");
println("5");
println("6");
println("7");
println("8");
println("9");
println("10");
```

- Es wäre sinnvoll, wenn wir einen Befehl zur Verfügung hätten, mit dem wir die Anweisung automatisch wiederholen könnten, wobei jedes Mal das aktuelle Argument berechnet wird

Die `while`-Schleife (1)

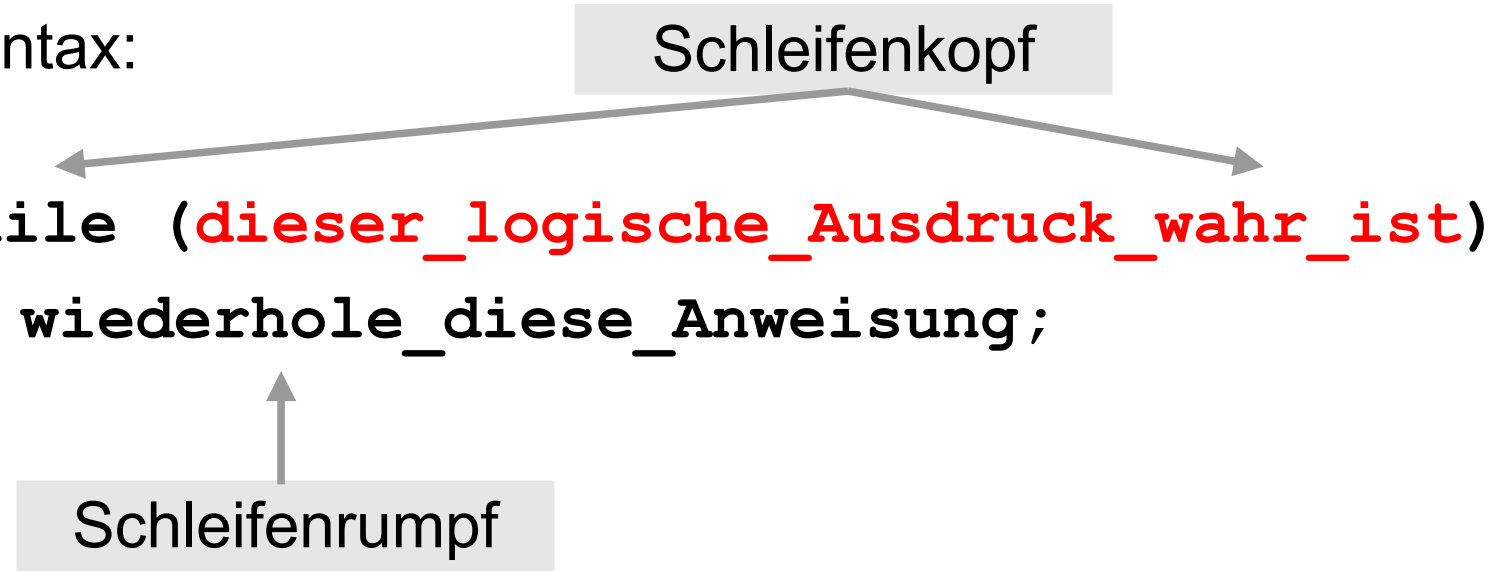
- Mit Hilfe der **`while`**-Schleife können Anweisungen iterativ wiederholt werden
- Syntax:



- Jede Ausführung der Anweisung im **Schleifenrumpf** wird **Iteration** genannt

Die `while`-Schleife (2)

- Die Bedingung im Schleifenkopf wird vor jeder Ausführung des Schleifenrumpfs getestet
- Nur wenn die Bedingung **`true`** liefert, wird der Schleifenrumpf ausgeführt
- Syntax:



Die `while`-Schleife (3)

- Die zu wiederholende Anweisung kann auch eine Verbundanweisung, d. h. ein Block, sein
- In diesem Fall sind die Anweisungen in geschweifte Klammern zu setzen
-
```
while (dieser_logische_Ausdruck_wahr_ist) {
   Anweisung_1;
   Anweisung_2;
         ⋮
   Anweisung_N;
}
```

Beispiel

```
// Nachfolgend werden die Zahlen 1..5 ausgegeben
int counter = 1; // Zählvariable
while (counter < 6) {
  println(counter);
  counter = counter + 1; // +1: Inkrement
}
```

Erste Ausführung der Zählschleife

```
int counter = 1;

while (1 < 6) { // Ausdruck liefert true

  // Da der logische Ausdruck true liefert, werden
  // die Anweisungen im Schleifenrumpf ausgeführt

  // counter liefert 1
  println(1);

  2 = 1 + 1; // counter wird erhöht auf 2

}
```

Ausgabe:

1

Zweite Ausführung der Zählschleife

```
int counter = 1;

while (2 < 6) { // Ausdruck liefert true

  // Da der logische Ausdruck true liefert, werden
  // die Anweisungen im Schleifenrumpf ausgeführt

  // counter liefert 2
  println(2);

  3 = 2 + 1; // counter wird erhöht auf 3

}
```

Ausgabe:

1

2

Dritte Ausführung der Zählschleife

```
int counter = 1;

while (3 < 6) { // Ausdruck liefert true

  // Da der logische Ausdruck true liefert, werden
  // die Anweisungen im Schleifenrumpf ausgeführt

  // counter liefert 3
  println(3);

  4 = 3 + 1; // counter wird erhöht auf 4

}
```

Ausgabe:

1

2

3

Vierte Ausführung der Zählschleife

```
int counter = 1;

while (4 < 6) { // Ausdruck liefert true

  // Da der logische Ausdruck true liefert, werden
  // die Anweisungen im Schleifenrumpf ausgeführt

  // counter liefert 4
  println(4);

  5 = 4 + 1; // counter wird erhöht auf 5

}
```

Ausgabe:

1

2

3

4

Fünfte Ausführung der Zählschleife

```
int counter = 1;

while (5 < 6) { // Ausdruck liefert true

  // Da der logische Ausdruck true liefert, werden
  // die Anweisungen im Schleifenrumpf ausgeführt

  // counter liefert 5
  println(5);

  6 = 5 + 1; // counter wird erhöht auf 6

}
```

Ausgabe:

```
1
2
3
4
5
```

Termination der Zählschleife

```
int counter = 1;
while (6 < 6) { // Ausdruck liefert false!

   // Da der logische Ausdruck false liefert, werden die
   // Anweisungen im Schleifenrumpf NICHT ausgeführt

   // Anstatt dessen terminiert die Schleife.
   // D.h. die Bearbeitung der Schleife
   // wird beendet und das Programm
   // mit der nächsten Anweisung nach der
   // Schleife - sofern vorhanden -
   // fortgesetzt
}
```

Ausgabe:

```
1
2
3
4
5
```

Zählschleife: Anderer Start- und Endwert

```
int counter = 5; // Startwert der Zählschleife

while (counter < 10) { // Endwert der Zählschleife
  println(counter);
  counter = counter + 1;
}
```

Ausgabe:

```
5
6
7
8
9
```

Zählschleife: Anderes Inkrement

```
int counter = 1; // Startwert der Zählschleife

while (counter < 10) { // Endwert der Zählschleife
  println(counter);
  counter = counter + 2;
}
```

Ausgabe:

```
1
3
5
7
9
```

Zählschleife: Rückwärts zählen

```
int counter = 5; // Startwert der Zählschleife

while (counter > 0) { // Endwert der Zählschleife
  println(counter);
  counter = counter - 1; // -1: Dekrement
}
```

Ausgabe:

```
5
4
3
2
1
```

Rückwärts zählen: Anderes Dekrement

```
int counter = 10; // Startwert der Zählschleife


while (counter > 0) { // Endwert der Zählschleife

  println(counter);

  counter = counter - 2; // Das Dekrement
                         // kann auch
}                        // kleiner als 1
                         // sein
```

Ausgabe:

```
10
8
6
4
2
```

Abkürzungen

Inkrementieren

```
x = x + 1;
```

kann wie folgt abgekürzt werden:

1. `x++; // Erhöht Variablenwert immer um 1!`
2. `x += 1; // Hier sind auch andere Werte möglich`

Beispiel

```
// Nachfolgend werden die Zahlen 1..5 ausgegeben
int counter = 1;

while (counter < 6) {

  println(counter);

  // counter = counter + 1; kann abgekürzt werden zu
  counter++;
}
```

Dekrementieren

```
x = x - 1;
```

ist äquivalent zu

```
x--;
```

und zu

```
x -= 1;
```

Beispiel

```
// Nachfolgend werden die Zahlen 5..1 ausgegeben
int counter = 5;

while (counter > 0) {

  println(counter);

  // counter = counter - 1; kann abgekürzt werden zu
  counter--;
}
```

Überblick

```
x = x + n;
```

ist äquivalent zu:

```
x += n;
```

```
x = x - n;
```

ist äquivalent zu:

```
x -= n;
```

```
x = x * n;
```

ist äquivalent zu:

```
x *= n;
```

```
x = x / n;
```

ist äquivalent zu:

```
x /= n;
```

etc.

Prä- und Post-Inkrementieren

- Operatoren können nicht nur hintangestellt, sondern auch vorangestellt werden
- Dies ändert die Reihenfolge, in der der Ausdruck ausgewertet wird

```
x++;
```

1. x wird ausgewertet
2. x wird inkrementiert

```
++x;
```

1. x wird inkrementiert
2. x wird ausgewertet

Prä- und Post-Dekrementieren

- Operatoren können nicht nur hintangestellt, sondern auch vorangestellt werden
- Dies ändert die Reihenfolge, in der der Ausdruck ausgewertet wird

```
x--;
```

1. x wird ausgewertet
2. x wird dekrementiert

```
--x;
```

1. x wird dekrementiert
2. x wird ausgewertet

Beispiele

Beispiel:	Ergebnis:

`int x = 10;`
`int y;`
`y = x++;`
	`x ist 11`
	`y ist 10`

`int x = 10;`
`int y;`
`y = ++x;`
	`x ist 11`
	`y ist 11`

`int x = 10;`
`int y;`
`y = x--;`
	`x ist 9`
	`y ist 10`

`int x = 10;`
`int y;`
`y = --x;`
	`x ist 9`
	`y ist 9`

Zähler in Terminationsbedingung

- Das In- bzw. Dekrementieren der Zählvariable kann auch als Teil der Terminationsbedingung geschehen

```
// Nachfolgend werden die Zahlen 1..5 ausgegeben

int counter = 0;
while (counter++ < 5) {
  println(counter);
  // counter++; kann entfallen!
}
```

- Vorsicht: Kann Quelle schwer zu findender Programmfehler sein

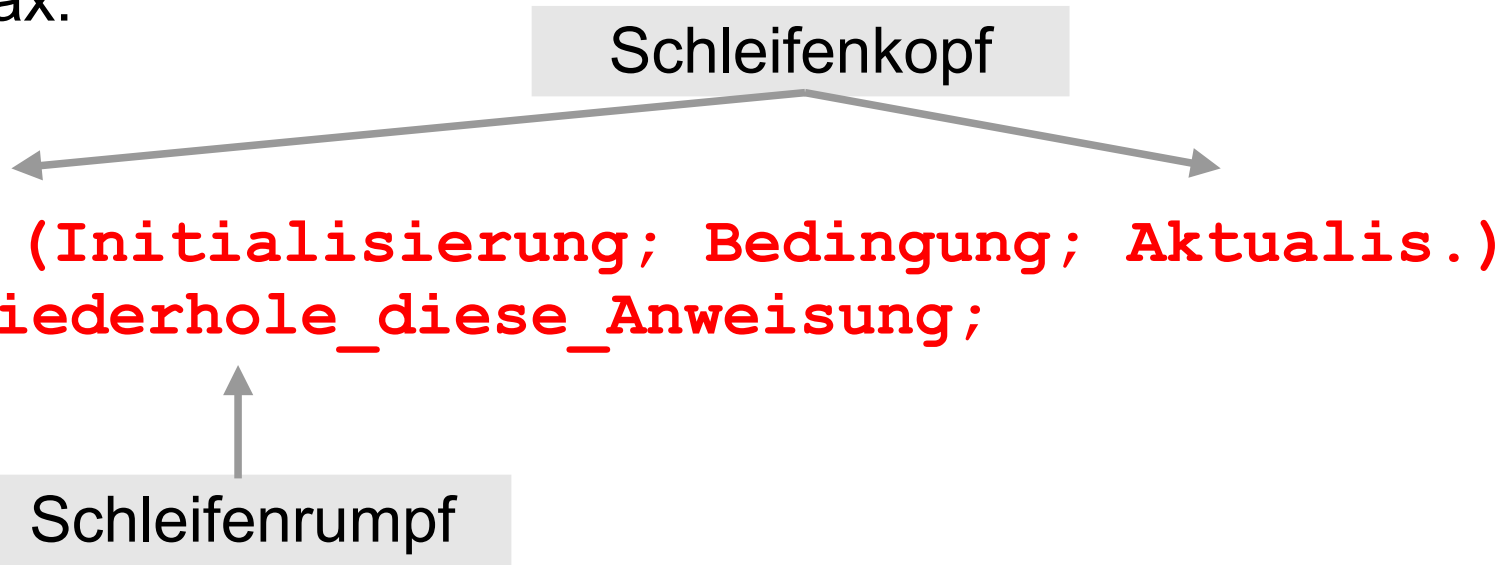
`for`-Schleife

`for`-Schleife (1)

- Kurzform für **`while`**
- Integriert Initialisierung, In- bzw. Dekrementieren der Zählvariable und die Terminationsbedingung in den Schleifenkopf
- Alle Probleme, die sich mit einer **`for`**-Schleife lösen lassen, können auch mit einer **`while`**-Schleife gelöst werden und umgekehrt
- **`for`**-Schleife ermöglicht bei Zählschleifen kompakteren Code

`for`-Schleife (2)

- Jede Ausführung der Anweisung im *Schleifenrumpf* wird auch hier *Iteration* genannt
- Syntax:

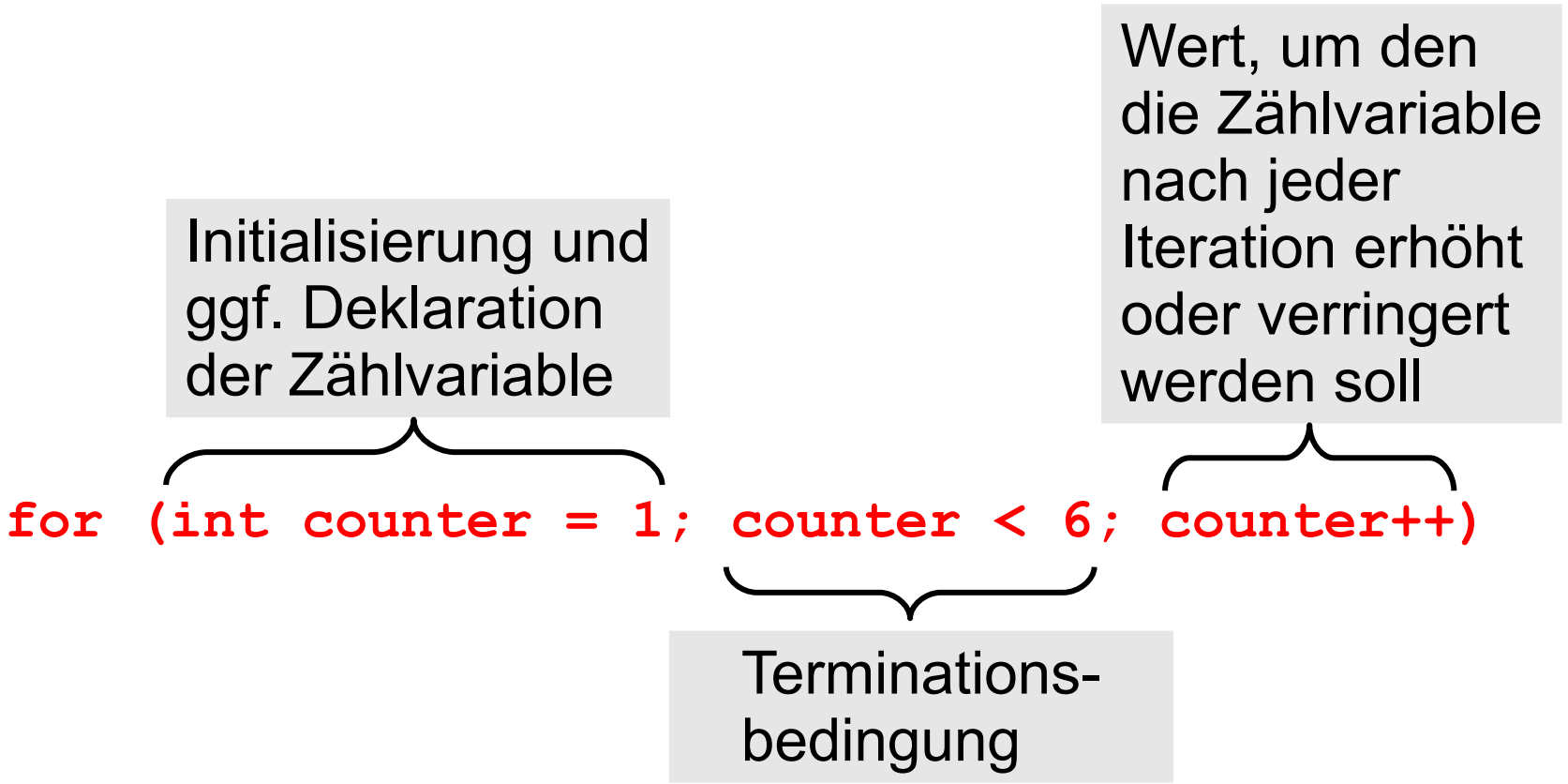


`for`-Schleife: Verbundanweisung

- Die zu wiederholende Anweisung kann auch hier eine Verbundanweisung, d. h. ein Block sein
- In diesem Fall sind wieder geschweifte Klammern zu benutzen

```
for (Initialisierung; Bedingung; Aktualis.) {
  Anweisung_1;
  Anweisung_2;
        ⋮
  Anweisung_N;
}
```

Bedeutung der Einträge im Schleifenkopf



`for`-Schleife: Start- und Endwert

```
for (int counter = 5; counter < 10; counter++)
  println(counter);
```

Ausgabe:

```
5
6
7
8
9
```

`for`-Schleife: Anderes Inkrement

```
for (int counter = 1; counter < 10; counter += 2)
  println(counter);
```

Ausgabe:

```
1
3
5
7
9
```

`for`-Schleife: Rückwärts zählen

```
for (int counter = 5; counter > 0; counter--)
  println(counter);
```

Ausgabe:

```
5
4
3
2
1
```

`for`-Schleife mit Dekrement kleiner -1

```
for (int counter = 10; counter > 0; counter -= 2)
  println(counter);
```

Ausgabe:

```
10
8
6
4
2
```

Verschachtelte Zählschleifen

Einführung

- Viele Probleme erfordern das Ausführen von Schleifen als Teil anderer Schleifen
- Beispiel: Uhr
  - Ein kompletter Umlauf des Sekundenzeigers führt zum Weiterschalten des Minutenzählers um eine Minute
  - Ein kompletter Umlauf des Minutenzeigers führt zum Weiterschalten des Stundenzählers um eine Stunde
  - Ein kompletter Umlauf des Stundenzeigers führt zum Weiterschalten der Tagesanzeige etc.
- Oder: Eine Iteration der Minutenschleife bedingt einen kompletten Durchlauf der Sekundenschleife usw.
- Derartige Probleme lassen sich lösen, indem im Rumpf einer Schleife eine andere aufgerufen wird

Beispiel: Zeichnen eines Keils

- Es soll folgender Keil gezeichnet werden:

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

- Dafür stehen folgende Ausgabeoperationen zur Verfügung:
  - Stern: `print("*");`
  - Neue Zeile: `println();`

Ausgabeoperationen für jede Zeile

- Operationen:

Ergebnis	Operation
`*`	`print("*");` `println();`
`**`	`print("*"); print("*");` `println();`
`***`	`print("*"); print("*"); print("*");` `println();`
`****`	`print("*"); print("*"); print("*"); print("*");` `println();`
`*****`	usw.
`******`	
`*******`	

Schleife 1

Ergebnis	Operation
*	`print("*");` `println();`
**	`print("*"); print("*");` `println();`
***	`print("*"); print("*"); print("*");` `println();`
usw.	

- Die Zeichenoperationen einer jeden Zeile können in einer Schleife zusammengefasst werden
- Terminationsbedingung dieser Schleife ist die Anzahl der zu druckenden Sterne
- Wenn diese erreicht ist, wird die Schleife beendet
- Geprüft wird dies mit Hilfe einer *Zählvariable*

Programmcode

- Die Zeilennummer legt die Anzahl der zu schreibenden Sterne fest:
  - Zeile 1: `zeile = 1;`
  - Zeile 2: `zeile = 2;`
  - Zeile 3: `zeile = 3;`
  - usw.
  - Terminationsbedingung: Anzahl der zu schreibenden Sterne
- ```
  for (int sterne = 0; sterne < zeile; sterne++)
    print("*");
  println();
  ```
  ```

Schleife 2

- Die Zeilenfolge repräsentiert ebenfalls eine Schleife
- Mit jeder neuen Zeile wird die Anzahl der Sterne inkrementiert
- Terminationsbedingung: Anzahl der zu schreibenden Zeilen

Ergebnis	Operation
*	`print("*");` `println();`
**	`print("*"); print("*");` `println();`
***	`print("*"); print("*"); print("*");` `println();`
usw.	

Schleife 2

Programmcode

```
int anzahlZeilen = 7;
for (int zeile = 1; zeile <= anzahlZeilen; zeile++) {
  for (int sterne = 0; sterne < zeile; sterne++)
    print("*");
  println();
}
```

Ergebnis	Operation
*	print("*"); println();
**	print("*"); print("*"); println();
***	print("*"); print("*"); print("*"); println();
usw.	

Gültigkeitsbereiche

Gültigkeitsbereich der Zählvariable (1)

- Die Deklaration der Zählvariable einer **`for`**-Schleife kann Teil des Schleifenkopfs sein:

```
for (int counter = 1; counter < 6; counter++)
  println(counter); // Ausgabe: 1..5
```

- In diesem Fall ist der Gültigkeitsbereich der Zählvariable auf den Schleifenrumpf eingeschränkt

Gültigkeitsbereich der Zählvariable (2)

- Wird die Zählvariable außerhalb des Schleifenkopfs deklariert, so ist sie ab der Deklarationsstelle überall im Code einer Sketch-Datei gültig:

```
int counter;

for (counter = 1; counter < 6; counter++)
  println(counter);
```

Gültigkeitsbereiche (1)

```
int sum = 0;
for (int j = 1; j < 6; j++)
  sum = sum + j;
println("The sum is: " + sum);
```

- **`sum`** wurde außerhalb der **`for`**-Schleife deklariert und kann deshalb auch außerhalb der **`for`**-Schleife benutzt werden
- Demgegenüber ist `j` nur innerhalb der **`for`**-Schleife gültig

Gültigkeitsbereiche (2)

```
int sum = 0;
for (int j = 1; j < 80; j++) {
  if (j % 7 != 0)
    sum = sum + j;
  else
    println("Not added to sum: " + j);
}
println("The final sum is: " + sum);
```

```
// Ausgabe:
Not added to sum: 7
Not added to sum: 14
...
The final sum is: 2698
```

- Da j im Schleifenkopf deklariert wurde, ist die Variable auch überall im Schleifenrumpf gültig
- D. h. auch in eingebetteten Kontrollstrukturen (Blöcke, Ab- und Verzweigungen sowie Schleifen)

Gültigkeitsbereiche (3)

```
int sumEven = 0, sumOdd = 0;

for (int j=2; j<8; j=j+2) // A variable with name "j"
  sumEven = sumEven + j;

println("The sum of evens is: " + sumEven);

for (int j=1; j<8; j=j+2) // Another var. with name "j"
  sumOdd = sumOdd + j;

println("The sum of odds is: " + sumOdd);
```

```
// Ausgabe:
The sum of evens is: 12
The sum of odds is: 16
```

- Da `j` in beiden Schleifenköpfen deklariert wurde, existieren zwei unabhängige Variablen gleichen Namens `j`, die jeweils nur im zugehörenden Schleifenrumpf gültig sind

Gültigkeitsbereiche (4)

```
int sumEven = 0, sumOdd = 0;

for (int j=2; j<8; j=j+2)
  sumEven = sumEven + j;

println("The sum of evens is: " + sumEven);

for (j=1; j<8; j=j+2) // "j" invalid here!
  sumOdd  = sumOdd + j;

println("The sum of odds is: " + sumOdd);
```

- Da die Variable `j` nur für die obere Schleife deklariert wurde, kann sie in der unteren Schleife NICHT verwendet werden
- Die Gültigkeit von `j` erlischt nach dem Ende der oberen Schleife!

Blöcke

Blöcke (1)

- Ein Block wird durch eine öffnende geschweifte Klammer eingeleitet und durch eine schließende geschweifte Klammer beendet
- Beispiel:

```
// z. B. if(…), while(…)
{
   // Anweisung 1
   // Anweisung 2
   // Anweisung 3
}
```

- Die zu einem Block gehörenden Anweisungen werden für eine bessere Lesbarkeit des Programmcodes eingerückt
  - 2-4 Leerzeichen (bzw. eine Tabulatorposition) sind üblich

Blöcke (2)

- Blöcke fassen die zu Kontrollanweisungen gehörenden Anweisungen zusammen
- Ein Block wird deshalb auch als **Verbundanweisung** bezeichnet
  - Ein Block ist eine Anweisung, die aus einer in geschweiften Klammern eingeschlossenen Folge von Anweisungen besteht
- Blöcke können Variablendeklarationen enthalten

Blöcke (3)

- Wird in einem Block eine Variable deklariert, so erstreckt sich deren Gültigkeitsbereich (engl. *scope*) vom Ort der Deklaration bis zum Ende des Blocks
- Ein Block definiert daher einen eigenen Gültigkeitsbereich
- Kernkonzept der **strukturierten Programmierung**
- Beispiel:

```
// z. B. if(…), while(…)
{
  int i;     // i ist im gesamten Block benutzbar
  // Anweisung 1
  // Anweisung 2
  double j; // j nur in der letzten Anweisung benutzbar
  // Anweisung 3
}
```

Lokale Variablen

```
// z. B. if(…), while(…)
{
  int i;     // i ist im gesamten Block benutzbar
  // Anweisung 1
  // Anweisung 2
  double j; // j nur in der letzten Anweisung benutzbar
  // Anweisung 3
}
```

- Gültigkeit von `i` und `j` ist auf den Block beschränkt
- `i` und `j` sind lokal gültig im Block
- `i` und `j` sind **lokale Variablen**

Globale Variablen (1)

```
int k;
k = 10;
// z. B. if(…), while(…)
{
  int i=1;
  // Anweisung 1
  // Anweisung 2
  double j;
  j = i+k;
}
```

- **k** ist innerhalb des Blocks und außerhalb gültig
- **k** ist daher global gültig
- **k** ist eine **globale Variable**

Globale Variablen (2)

- Globale Variablen können zu unerwünschten Seiteneffekten führen
  - Beispiel: Wert einer globalen Variable wird versehentlich innerhalb eines Blocks geändert, obwohl dieser außerhalb des Blocks noch benötigt wird
- Die Gültigkeit von Variablen sollte daher möglichst auf den Kontext beschränkt werden, in dem sie benutzt werden sollen
- Daher: Globale Variable vermeiden, wenn eine Aufgabe genauso gut durch eine lokale Variable erledigt werden kann

Blöcke und Information Hiding

- Die Einschränkung des Gültigkeitsbereichs einer Variablen erlaubt die Kapselung von Information
  - Gekapselte Daten sind außerhalb des Blocks nicht sichtbar und folglich auch nicht änderbar
  - Dies wird als "**Information Hiding**" bezeichnet
  - Information Hiding hilft unerwünschte Seiteneffekte zu vermeiden
- Beispiel:

```
{
  int i;          // i ist im gesamten Block benutzbar
  // Anweisung    // aber nicht außerhalb des Blocks!
                  // Der in i abgelegte Datenwert ist
                  // somit für alle Programmteile
}                 // außerhalb des Blocks versteckt
```

Schachtelung von Blöcken

- Da ein Block eine Anweisung ist, können Blöcke weitere Blöcke enthalten
- Damit ergibt sich eine Blockschachtelung und entsprechend eine Schachtelung der Gültigkeitsbereiche der darin deklarierten lokalen Variablen
  - In einem inneren Block deklarierte Variablen sind auch nur dort gültig
  - Die Gültigkeit von Variablen ist daher blockgebunden (*block-level scope*)

Blockschachtelung

- Beispiel:

```
// z. B. if(…), while(…)
{
    int i1;
    i1=0;
    // z. B. if(…), while(…)
    {
      int i2;      // i2 nur im inneren Block gültig
      i2 = i1+10; // i1 ist auch hier gültig
    }
    i1 = i2+10;    // i2 ist hier nicht mehr gültig!
}
```

Blockschachtelung: Variablendeklaration

- In einem äußeren Block deklarierte Variablen dürfen in einem inneren Block nicht erneut deklariert werden!

```
// z. B. if(…), while(…)
{
    int i1, i2;
    i1=0;
    // z. B. if(…), while(…)
    {
        int j, i2;  // i2 schon im übergeordneten
        i2=1;       // Block deklariert!
        j = i1+10;
    }
}
```

Allgemeine Schleifen

Endlosschleife (1)

```
int counter = 1; // Startwert der Zählschleife

while (counter > 0) { // Diese Bedingung ist wegen
                      // obiger Initialisierung
                      // immer erfüllt!

  println(counter);

  counter = counter + 1;
}
```

Ausgabe:

```
1
2
3
4
⋮
```

Dies wird auch als **nicht-terminierende Schleife** bezeichnet

Endlosschleife (2)

```
int counter = 1; // Startwert der Zählschleife


while (true) {  // Die Bedingung ist immer erfüllt!
  println(counter);
  counter = counter + 1;
}
```

Ausgabe:

1

2

3

4

⋮

Schleife, die nie ausgeführt wird

```
int counter = 1; // Startwert der Zählschleife

while (false) {  // Die Bedingung ist NIE erfüllt !

  println(counter);

  counter = counter + 1;

}


// Liefert Compiler-Fehler:
// "unreachable statement"
```

Überwachungsgesteuerte Schleifen

- Die Schleifenbedingung muss nicht notwendigerweise vom Wert einer Zählvariable abhängen
- Der zugehörende Ausdruck kann ein beliebiger logischer Ausdruck sein
- Die Schleife wird ausgeführt, solange der Ausdruck **`true`** liefert
- Eine derartige Schleife wird als überwachungsgesteuerte Schleife (engl.: *sentinel controlled loop*) bezeichnet

Beispiel: Zufallszahl aus einem Bereich ziehen

```
double randomNum = 0.0;
int counter = 0;

while ((randomNum < 0.5) || (randomNum > 0.6)) {
  counter++;
  randomNum = Math.random();
}
println("Zufallszahl: " + randomNum +
        ", Anzahl Ziehungen: " + counter);
```

Wächterausdruck

- Anzahl der Ziehungen bzw. Schleifendurchläufe nicht im Vorhinein bekannt

Beispiel: Eingabewerte addieren

- **Erste Verfeinerung:**

```
  // Deklariere und initialisiere benötigte Variablen
  // Lies eine Zahl ein

  while (Eingabewert_ungleich_Null) {  // Wächter

    // Addiere Zahl zur Summe

    /* Lies die nächste Zahl ein
    Benutzer signalisiert Abbruch durch Eingabe von 0 */

  }
  // Gib die Summe aus
}
```

Zweite Verfeinerung

```
int num = 0, sum = 0;

num = IOLib.getInt("Enter first integer (0 to quit):");


while (Eingabewert_ungleich_Null) {  // Wächter-Ausdruck

   // Addiere Zahl zur Zwischensumme

   /* Lies die nächste Zahl ein
   Benutzer signalisiert Abbruch durch Eingabe von 0 */

}
// Gib das Ergebnis aus
}
```

Dritte Verfeinerung

```
int num = 0, sum = 0;
num = IOLib.getInt("Enter first integer (0 to quit):");

while (num != 0) {   // Wächter-Ausdruck

   // Addiere Zahl zur Zwischensumme

   /* Lies die nächste Zahl ein
   Benutzer signalisiert Abbruch durch Eingabe von 0 */

}
// Gib das Ergebnis aus
```

Zahl der Iterationen hier NICHT im Voraus festgelegt!

Vierte Verfeinerung

```
int num = 0, sum = 0;
num = IOLib.getInt("Enter first integer (0 to quit):");

while (num != 0) {  // Wächter-Ausdruck
   sum = sum + num;
   num = IOLib.getInt("Enter next integer (0 to quit):");

}
// Gib das Ergebnis aus
```

Fünfte Verfeinerung

```
int num = 0, sum = 0;
num = IOLib.getInt("Enter first integer (0 to quit):");
while (num != 0) {  // Wächter-Ausdruck
   sum = sum + num;
   num = IOLib.getInt("Enter next integer (0 to quit):");

}
println("Integer sum: " + sum);
```

Top-Down-Entwurf (1)

- Die Technik, einen Algorithmus zunächst grob in Funktionsblöcken zu beschreiben, die nach und nach in einzelne Anweisungen einer Programmiersprache aufgelöst und damit konkretisiert werden, wird als

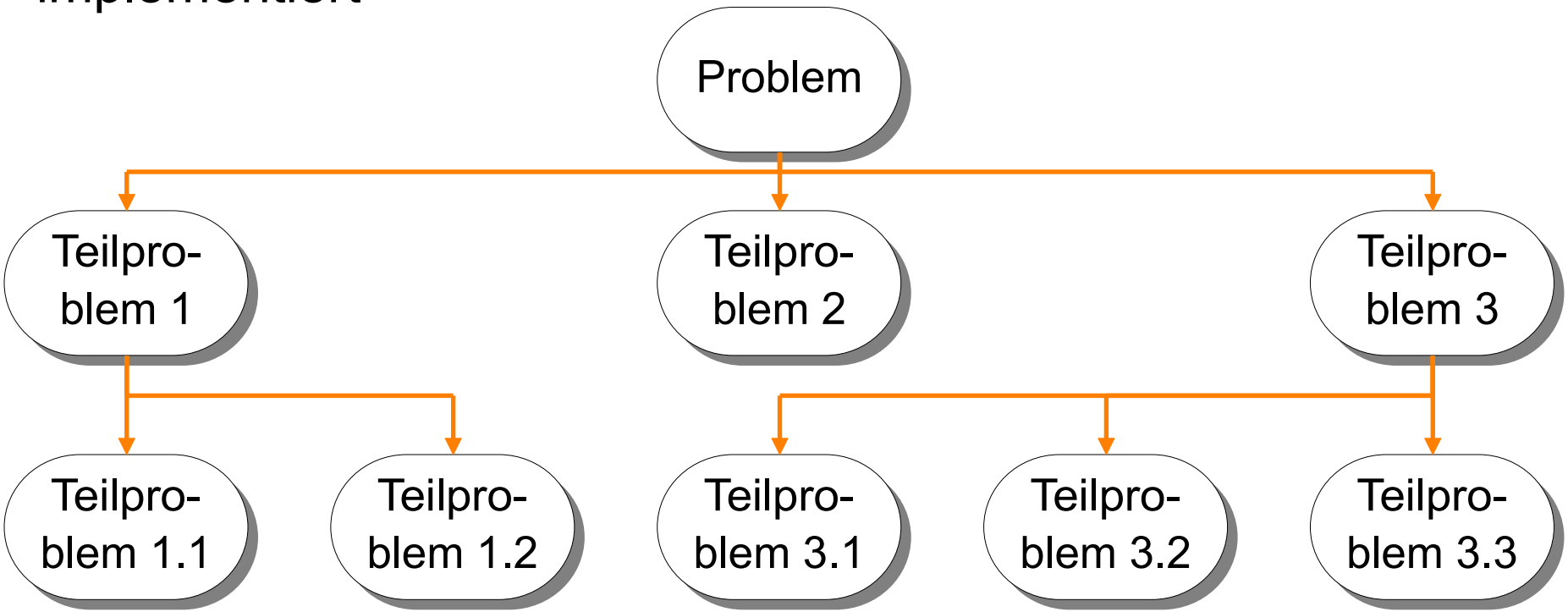
  Methode der **zunehmenden** oder **schrittweisen Verfeinerung** (engl.: *stepwise refinement*)

  bezeichnet

- Ein derartiger Entwurf wird auch **Top-Down-Entwurf** genannt

Top-Down-Entwurf (2)

- Bei einem Top-Down-Entwurf wird ein großes, schwer lösbares Problem in immer kleiner werdende und schließlich lösbare Teilprobleme zerlegt
- Die Lösungen der Teilprobleme werden dann schließlich implementiert



`do-while`-Schleife

`while` versus `do-while`

- while-Schleife:

```
while (logische_Bedingung_erfüllt) {
  // führe Befehle aus
}
```

  - Bedingung steht im *Schleifenkopf*

- do-while-Schleife:

```
do {
  // führe Befehle aus
} while (logische_Bedingung_erfüllt);
```

  - Bedingung steht im *Schleifenfuß*

Daten mit `while`-Schleife verarbeiten

```
int num = 0, sum = 0;
num = IOLib.getInt("Enter first integer (0 to quit):");

while (num != 0) {
  sum = sum + num;
  num = IOLib.getInt("Enter next integer (0 to quit):");

}
println("Sum: " + sum);
```

- Coderedundanz vorhanden

Daten mit `do-while`-Schleife verarbeiten

```
int num = 0, sum = 0;

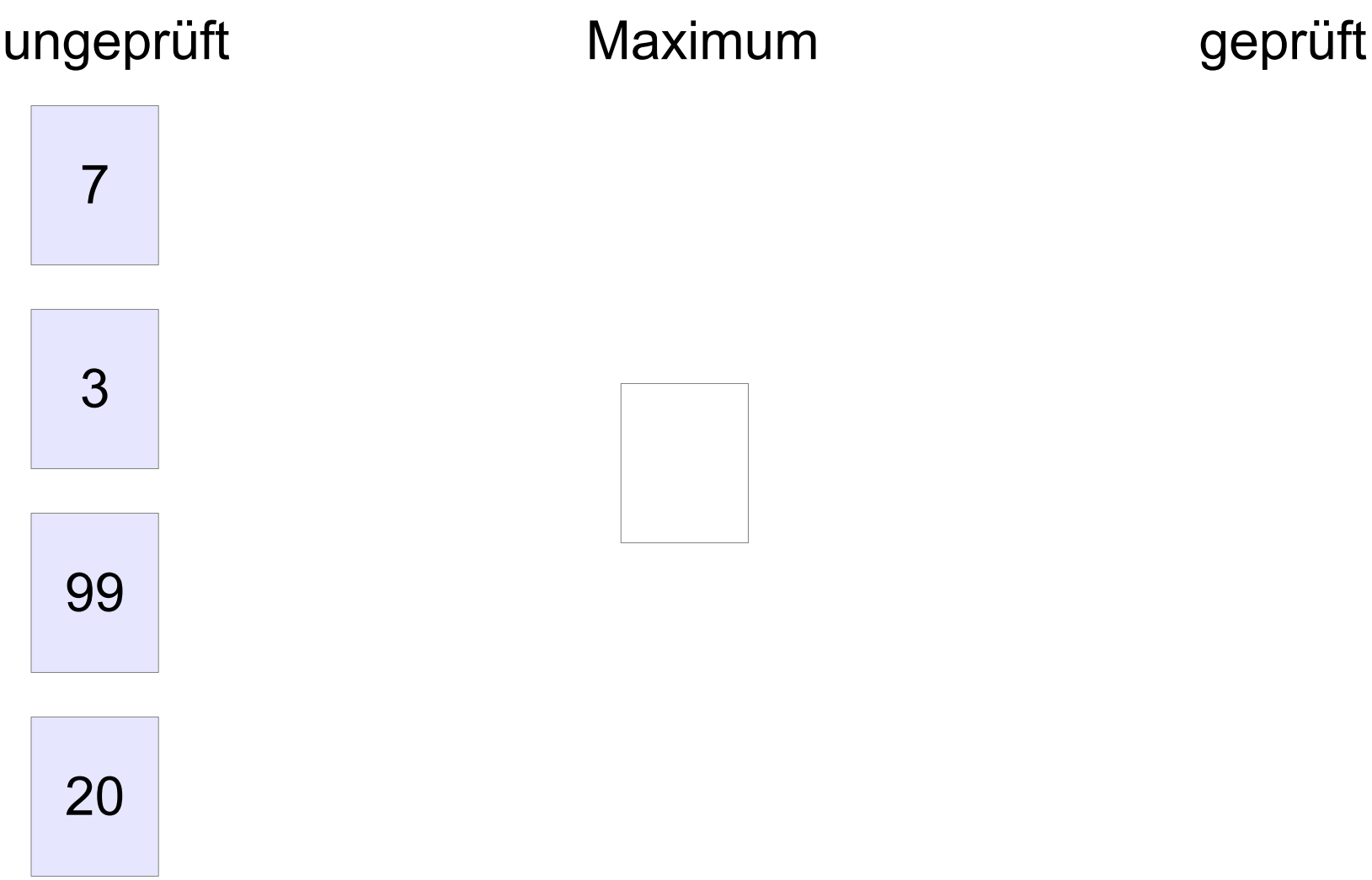
do {
  num = IOLib.getInt("Enter an integer (0 to quit):");
  sum = sum + num;
} while (num != 0);

println("Sum: " + sum);
```

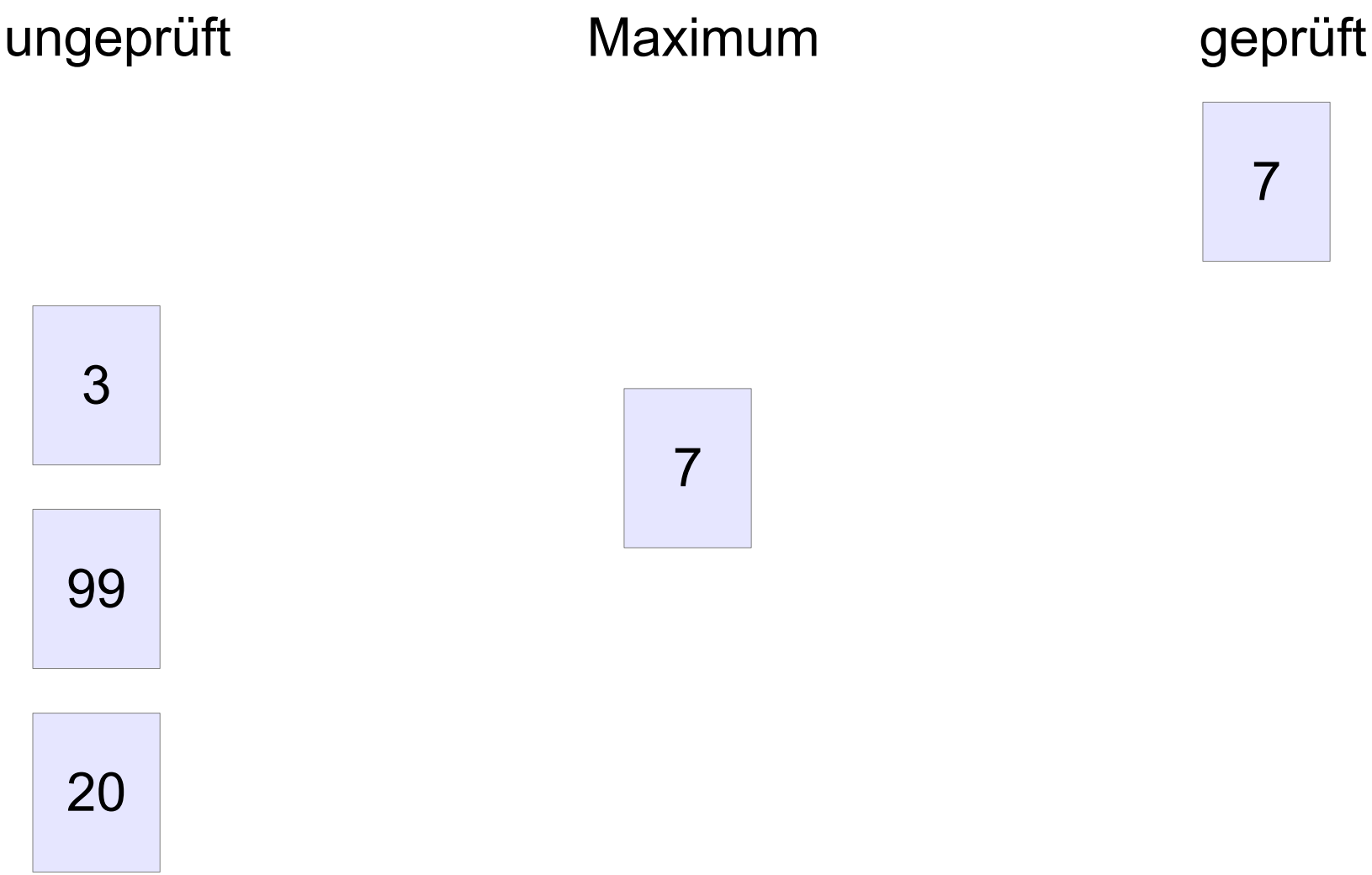
- Ermöglicht sofortigen Einstieg in den Schleifenrumpf, um diesen mindestens einmal auszuführen
  - Unnötige Eingabeanweisungen werden vermeiden

Maximumsuche

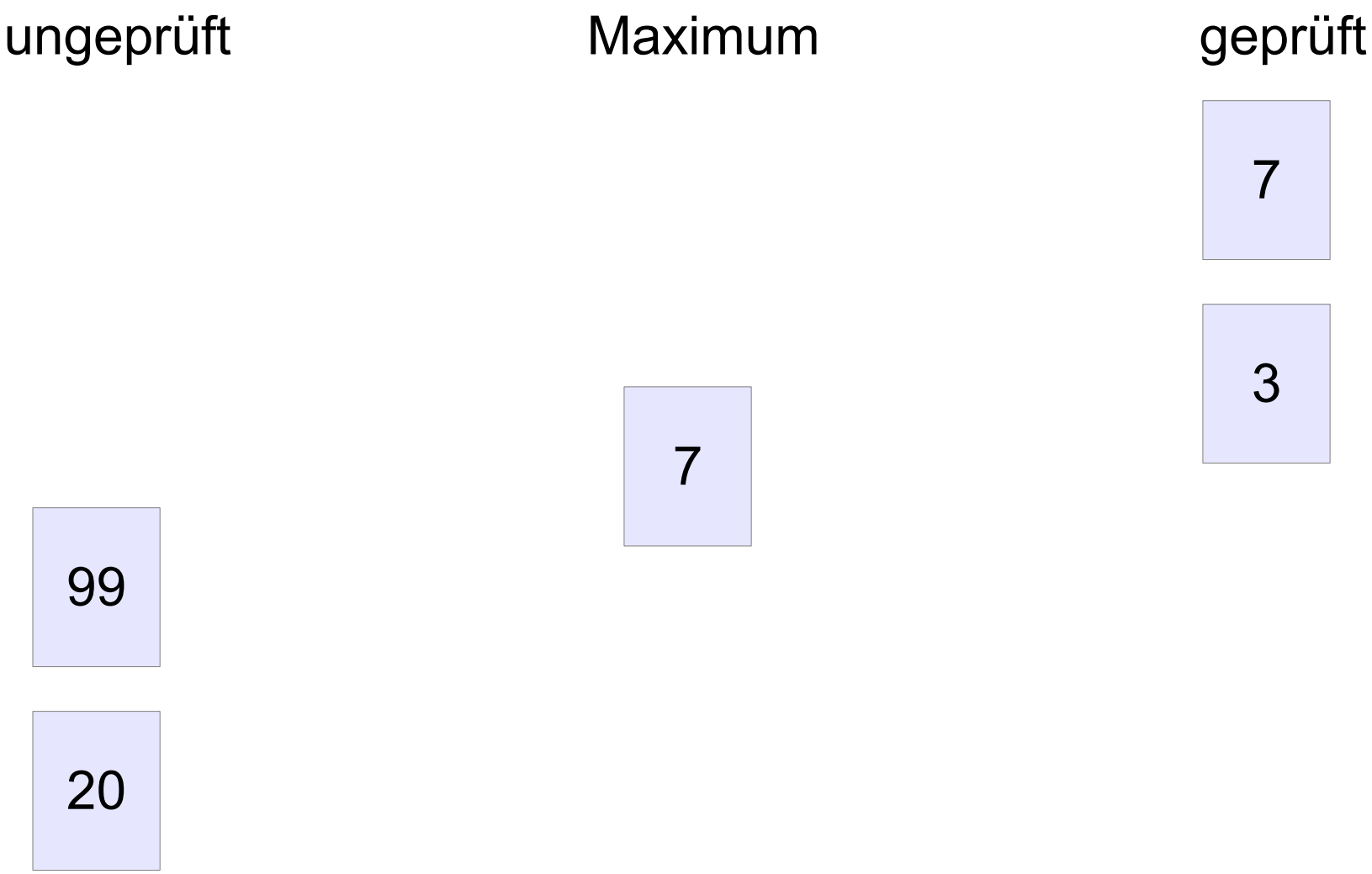
Beispiel: Maximumsuche



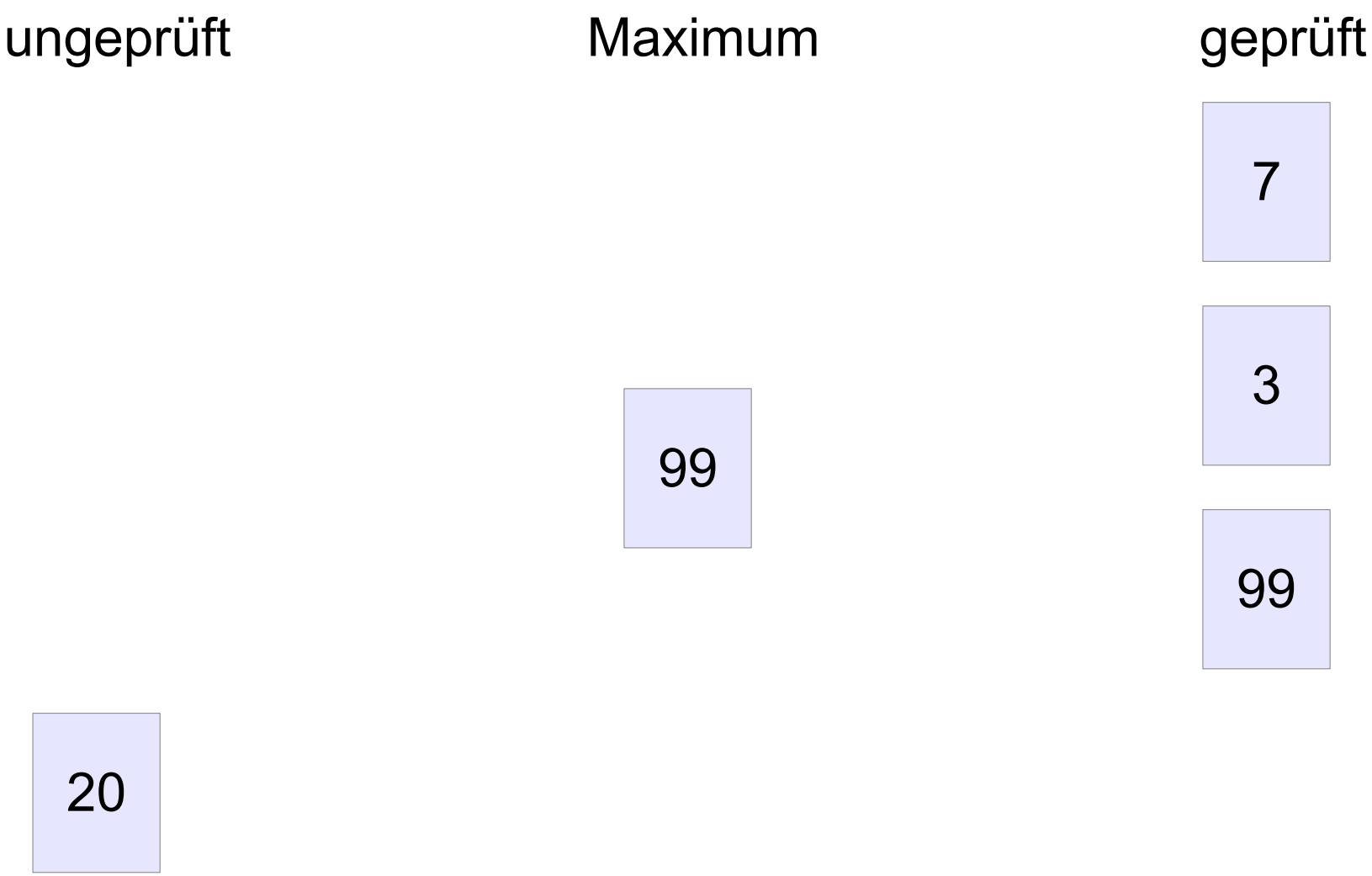
Maximumsuche (1)



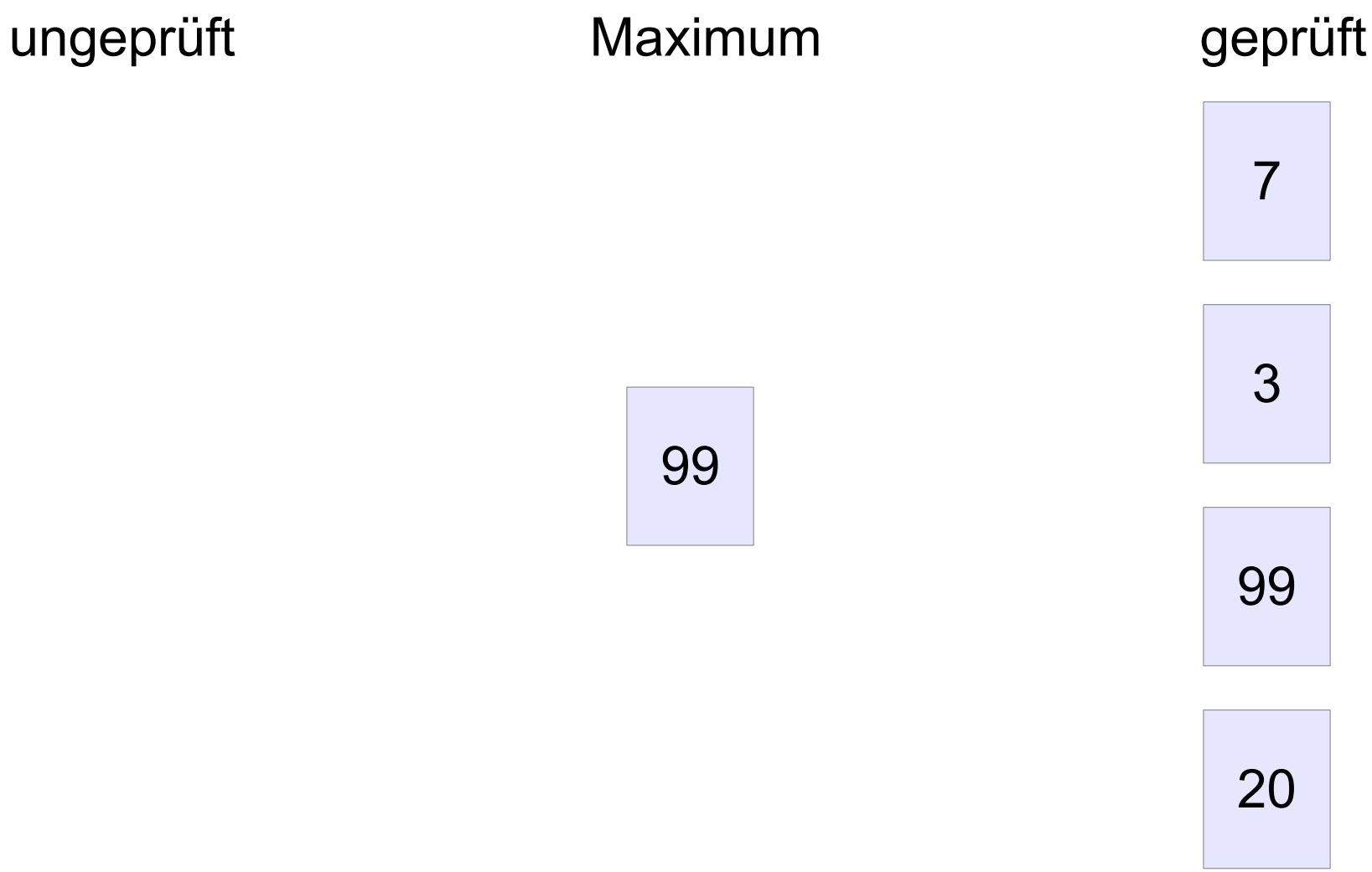
Maximumsuche (2)



Maximumsuche (3)



Maximumsuche (4)



Maximumsuche: Spezifikation

- In einer Schleife mit "Wächter-Ausdruck" gibt Benutzer solange Zahlenwerte ein, bis Abbruch gewünscht
  - Zahlenwerte werden mit gegenwärtigem Maximum verglichen und dieses ggf. aktualisiert
- Für Abbruch Eingabe eines speziellen Prüfwertes erforderlich
  - Prüfwert kann z. B. beliebiger Zahlenwert sein
    - Beispielsweise "0" für "Suche beenden"
  - Prüfwert legt die Terminationsbedingung fest

Erste Verfeinerung

```
/* Deklariere und initialisiere Variablen für Maximum,
   Eingabewert und Schleifenbedingung */

do {
   // Lies Zahl ein
   // Iteration 1: Initialisiere Maximum
   // Iteration i > 1: Vergleiche Zahl mit Maximum
   //   Falls Zahl größer als Maximum: Maximum <- Zahl
   // Frage, ob Fortsetzung gewünscht
}
while (Fortsetzung_gewünscht);

// Gib Maximum aus
```

Zweite Verfeinerung

```
int max = 0, num = 0, mode = 2;    // mode: Steuervariable
                                   // 2: Max. initialis.
                                   // 1: Fortsetzen
do {                               // 0: Terminieren

   // Lies Zahl ein
   // Iteration 1: Initialisiere Maximum
   // Iteration i > 1: Vergleiche Zahl mit Maximum
   //   Falls Zahl größer als Maximum: Maximum <- Zahl
   // Frage, ob Fortsetzung gewünscht
}
while (Fortsetzung_gewünscht);

// Gib Maximum aus
```

Dritte Verfeinerung

```
int max = 0, num = 0, mode = 2; // mode: Steuervariable
do {
  num = IOLib.getInt("Enter an integer:");

  // Iteration 1: Initialisiere Maximum
  // Iteration i > 1: Vergleiche Zahl mit Maximum
  // Falls Zahl größer als Maximum: Maximum <- Zahl
  // Frage, ob Fortsetzung gewünscht
}
while (Fortsetzung_gewünscht);

// Gib Maximum aus
```

Vierte Verfeinerung (1)

```
int max = 0, num = 0, mode = 2; // mode: Steuervariable
do {
  num = IOLib.getInt("Enter an integer:");

  // Iteration 1: Initialisiere Maximum
  // Iteration i > 1: Vergleiche Zahl mit Maximum
  // Falls Zahl größer als Maximum: Maximum <- Zahl
  if (mode == 2) max = num;
  else if (num > max) max = num;
  // Frage, ob Fortsetzung gewünscht
}
while (Fortsetzung_gewünscht);
// Gib Maximum aus
```

- Coderedundanz vorhanden

Vierte Verfeinerung (2)

```
int max = 0, num = 0, mode = 2; // mode: Steuervariable
do {
  num = IOLib.getInt("Enter an integer:");

  // Iteration 1: Initialisiere Maximum
  // Iteration i > 1: Vergleiche Zahl mit Maximum
  // Falls Zahl größer als Maximum: Maximum <- Zahl
  if (mode == 2 || num > max) max = num;

  // Frage, ob Fortsetzung gewünscht
}
while (Fortsetzung_gewünscht);
// Gib Maximum aus
```

Fünfte Verfeinerung

```
int max = 0, num = 0, mode = 2;
do {
  num = IOLib.getInt("Enter an integer:");
  if (mode == 2 || num > max) max = num;
  // Frage, ob Fortsetzung gewünscht
  mode = IOLib.getInt("Continue? ('1': yes, '0': no):");

}
while (mode == 1);
// Gib Maximum aus
```

Sechste Verfeinerung

```
int max = 0, num = 0, mode = 2;
do {
  num = IOLib.getInt("Enter an integer:");
  if (mode == 2 || num > max) max = num;
  mode = IOLib.getInt("Continue? ('1': yes, '0': no):");

}
while (mode == 1);
// Gib Maximum aus
println("Maximum: " + max);
```

Ergebnisgesteuerte Schleifen

- Termination kann auch von einem angestrebten Ergebnis bzw. einer Konvergenzbedingung abhängen
  - Allgemein: Wiederhole Rechenvorschrift bis Fehler oder Abweichung vom Zielwert gegen 0 geht
- Engl.: *Result controlled loop*
- Beispiel:
  - Berechne iterativ die Anzahl Jahre, die bei gegebener Zinsrate erforderlich sind, um eine vorgegebene Menge Kapital zu erzielen

Beispiel

```
final double GOAL = 100000.0;
final double INTEREST = 0.03;
float euro = 10000.0;
int years = 0;

while (euro < GOAL) {

  // add another year's interest

  euro += euro*INTEREST;

  years++;

}

println("It takes " + years +
        " years to reach " + GOAL + " Euros");

// Ausgabe: It takes 78 years to reach 100000.0 Euros
```

Binomische Reihe:

$e_y = e_0 (1+i)^y$

Felder

Zahlenmengen verarbeiten

- Mit Hilfe der Maximum- bzw. Minimumsuche sind wir in der Lage, aus einer Menge von Eingabewerten die größte bzw. kleinste Zahl herauszufinden
- Handelt es sich dabei um wenige Zahlen, können diese manuell eingegeben werden
- Bei großen Zahlenmengen ist es hingegen zweckmäßiger, wenn diese zwischengespeichert und nacheinander verarbeitet werden können
- Hierfür müssen die Zahlenwerte allerdings abzählbar gespeichert sein
- Dies ist mit Hilfe eines Felds (engl.: *array*) möglich

Felder

- Ein Feld besteht aus einer Menge aufeinanderfolgender Speicherzellen: den Feldelementen
- Wie Variablen, so erhalten auch Felder einen Namen
- Der Name bezeichnet die erste Speicherzelle des Felds
- Diese und alle weiteren Zellen erhalten eine Nummer, die einer relativen Adresse im Feld entspricht
- Diese Nummer wird als Index oder Schlüssel bezeichnet
  - Indiziertes Feld

Felder: Indizierung

	Index	Wert	Zugriff
`Name`	0	7	`Name[0]`
	1	3	`Name[1]`
	2	99	`Name[2]`
	3	20	`Name[3]`

Indizes beginnen immer mit 0!

Zugriff auf die Elemente eines Feldes

- Beispiel: Schreiben des Wertes des Feldelements Nr. 4 des Felds **`data`** in eine Variable **`myDataVal`**:

```
myDataVal = data[3];
```

- Schreiben eines Wertes in das Feldelement Nr. 4:

```
data[3] = myDataVal;
```

```
data[3] = 5;
```

Zuweisen von Ausdrücken

- Feldelemente können in Ausdrücken verwendet werden und den Wert eines Ausdrucks zugewiesen bekommen
- Beispiele:

```
data[0] = (x + data[2]) / 4 ;
data[1] = data[1] + 1;
data[2]++;
data[3] += 4;
data[4] = data[1] / data[2];
```

Deklaration eines Felds

- Der Datentyp eines Felds kann ein beliebiger Datentyp sein
- Der Datentyp bezeichnet den Typ der Feldelemente
- Alle Feldelemente haben denselben Typ!
- Syntax: `type[] arrayName;`
- Alternative Syntax: `type arrayName[];`
- Beispiele:

```
int[] data1;
int data2[];
```

Erzeugung von Feldelementen

- Die gewünschte Anzahl von Feldelementen muss explizit mit dem Operator **`new`** erzeugt werden
- Syntax:

```
arrayName = new type[length];
```

- Beispiel:

```
data = new int[10];
```

- Dies erzeugt 10 Feldelemente (Speicherzellen) vom Typ Integer für das Feld **`data`**

Deklaration mit Initialisierung

- Die Deklaration eines Felds und die Erzeugung seiner Feldelemente kann in einem Schritt erfolgen
- Syntax:

```
type[] arrayName = new type[length];
```

- Beispiel:

```
int[] data = new int[10];
```

- Dies deklariert und erzeugt ein Feld mit den Feldelementen `data[0]...data[9]`!

Überprüfung der Feldgrenzen

- Feldgrenzen unterliegen einer strengen Überprüfung (engl.: *bounds checking*) seitens des Laufzeitsystems
  - Illegale Indizes verursachen Laufzeit-Fehler! (**`ArrayIndexOutOfBoundsException`**)
- Beispiele:

```
int[] data = new int[10];
```

```
data[-1] immer illegal
```

```
data[10] illegal (bezüglich der obigen Deklaration)
```

```
data[1.5] immer illegal
```

```
data[0] immer OK
```

```
data[9] OK (bezüglich der obigen Deklaration)
```

Adressrechnung

- Die Position der Speicherzellen der Feldelemente wird aus
  - dem Feldnamen, d. h. der Startadresse des Felds im Speicher,
  - dem Index, d. h. der relativen Adresse des Elements im Feld,
  - und dem Typ, d. h. dem Platzbedarf eines Feldelements in Bytes berechnet
- Beispiel für `int` mit 32-Bit:

	Index	Adresse	Wert	Zugriff
`Name`	0	100	7	`Name[0]`
	1	104	3	`Name[1]`
	2	108	99	`Name[2]`
	3	112	20	`Name[3]`

Variablen als Indizes

- Als Indizes lassen sich nicht nur Konstanten, sondern ebenfalls Variablenwerte verwenden
- Grundlage für die systematische Verarbeitung der Feldelemente durch einen Algorithmus, wie z. B. die Maximumsuche
- Verletzungen der Feldgrenzen zumeist Folge falsch berechneter Variablenwerte

Variablen als Indizes

```
float[] val = new float[3];
val[0] = 0.12;
val[1] = 1.43;
val[2] = 2.98;

for (int i = 0; i < 3; i++)
   println(val[i]);
```

Ausgabe:

```
0.12
1.43
2.98
```

Variablen als Indizes: Zugriffsfehler

```
float[] val = new float[3];

val[1] = 0.12;      // Hier wurde vergessen, dass
val[2] = 1.43;      // Indizes bei 0 beginnen und
val[3] = 2.98;      // NICHT bei 1!

for (int i = 1; i < 4; i++)
   println(val[i]);
```

- Kein Compiler-Fehler, aber bei der Ausführung Abbruch mit:
  `ArrayIndexOutOfBoundsException`

Feldlänge

- Die Länge eines Feldes, d. h. die Anzahl seiner Feldelemente, kann aus der Eigenschaft **`length`** abgefragt werden
- Hierfür existiert der Punktoperator

- Beispiel:

```
int[] data = new int[10];
int noOfArrElements = data.length;
```

Deklaration, Erzeugung und Initialisierung

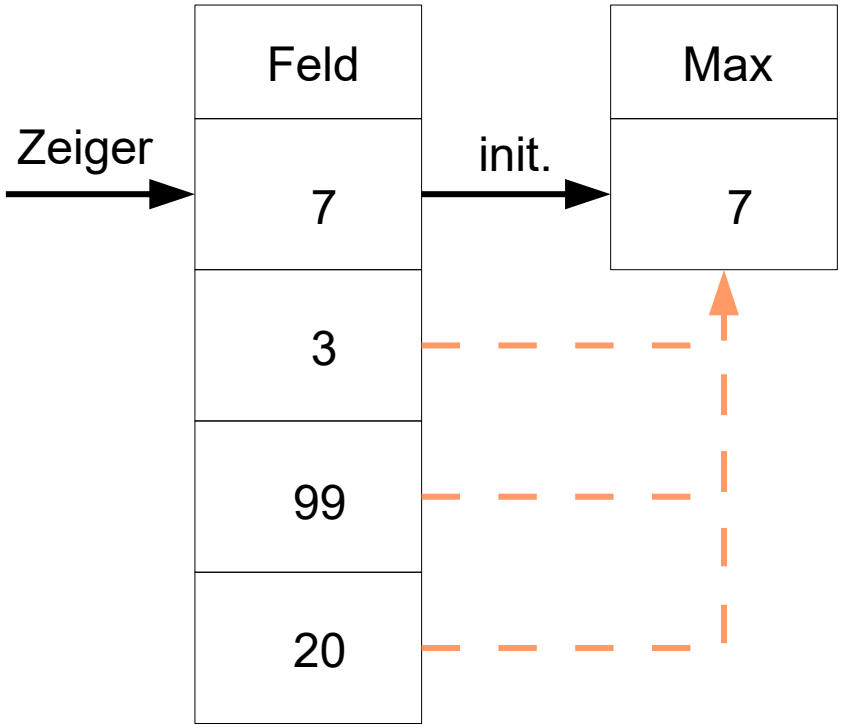
- Felder bzw. deren Feldelemente können in einem Schritt deklariert, erzeugt und initialisiert werden
- Beispiel:

```
int[] data = {23, 38, 14, -3, 0,
              14, 9, 103, 0, -56};
```

- Dies deklariert ein Feld vom Typ **`int`** mit Namen **`data`**
- Erzeugt 10 Feldelemente mit den Indizes 0..9
- Initialisiert die Feldelemente mit den angegebenen Werten
- **`data[0] = 23;`** ... **`data[9] = -56;`**

Beispiel: Maximumsuche mit Feld

- Zeiger markiert nächstes zu verarbeitendes Feldelement
- Feldelement wird mit Maximum verglichen und dieses ggf. aktualisiert
- Zeiger wird danach auf die nächste Position im Feld verschoben
- **Terminationsbedingung**: Verfahren terminiert, wenn kein weiteres Feldelement mehr vorhanden ist



Maximumsuche mit Feld: Erste Verfeinerung

```
/* 1. Löse das Problem der Maximumsuche
      mittels eines iterativen Verfahrens */

/* 2. Gib das gefundene Maximum aus */
```

Maximumsuche mit Feld: Zweite Verfeinerung

```
/* Das iterative Verfahren kann in die
   folgenden Teilprobleme zerlegt werden:
     1. Initialisierung
     2. Verarbeitung der Zahlenwerte
        mit einer Zählschleife
*/

/* Gib das gefundene Maximum aus */
```

Maximumsuche mit Feld: Dritte Verfeinerung

```
/* 1. Zerlegung der Initialisierung in die Schritte:

     1.1 Deklariere und initialisiere benötigte Variablen
     1.2 Bezeichne erstes Feldelement als Maximum

   2. Zerlegung der Zählschleife in die Schritte:

      2.1 Setze Zeiger auf zweites Feldelement

      2.2 Wiederhole solange
          Zeiger <= letztes Feldelement:
            2.2.1 Vergleiche gegenwärtiges Feldelement
                  mit Maximum
            2.2.2 Falls Feldelement größer:
              2.2.2.1 Ersetze Maximum durch Feldelement
              2.2.2.2 Setze Zeiger ein Feldelement weiter

  3. Gib Maximum aus */
```

Maximumsuche mit Feld: Vierte Verfeinerung

```
int[] arr = {7, 3, 99, 20}; // Initialisierung
int max = arr[0];

for (int i = 1; i < arr.length; i++)
  if (arr[i] > max) // Feldelement mit Max. vergleichen
    max = arr[i];

// Maximum auf Konsole ausgeben
println("Maximum: " + max);
```

Zahlen sortieren

Zahlen sortieren (1)

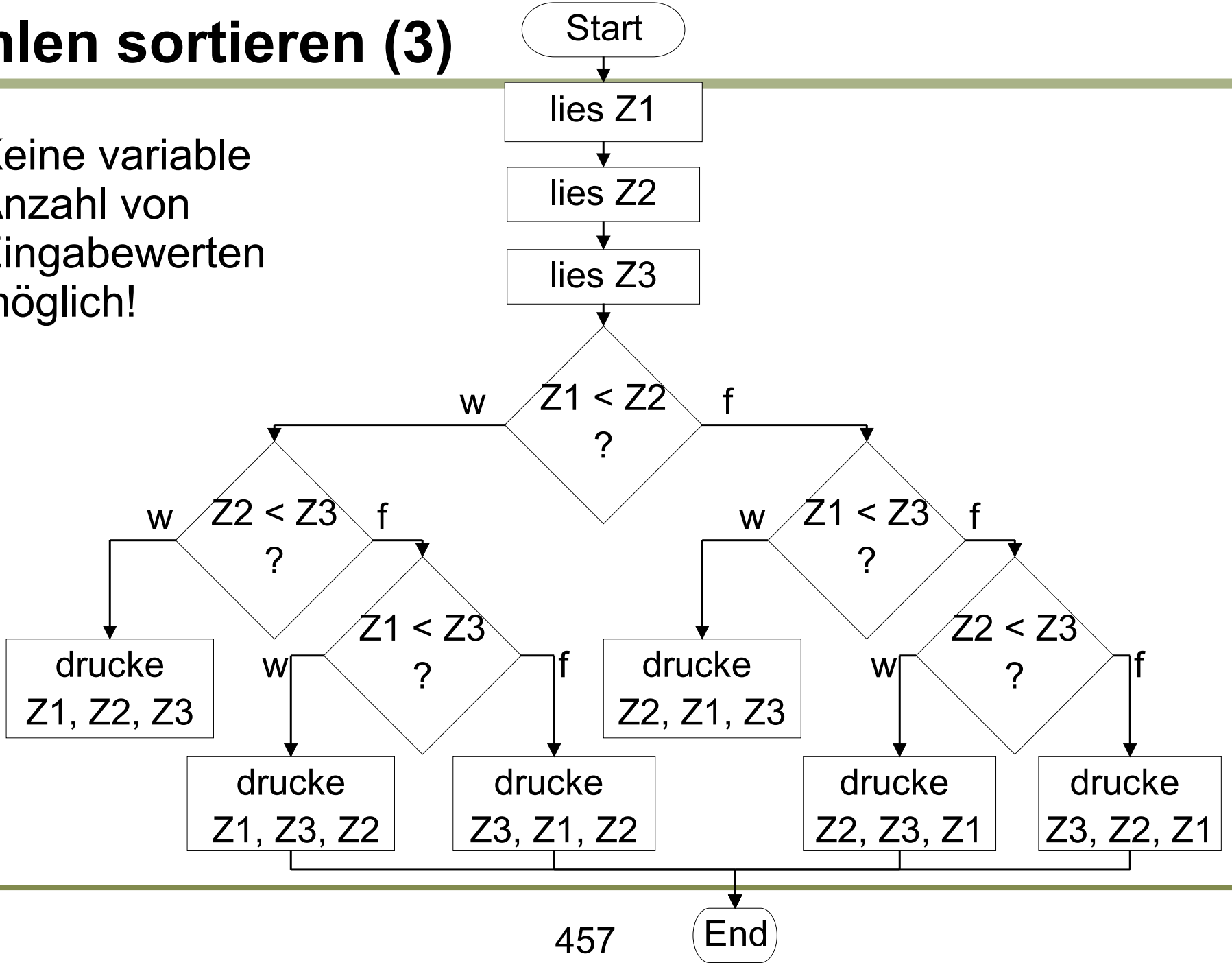
- Mit Hilfe der Maximum- bzw. Minimumsuche sind wir in der Lage, aus einer Menge von Eingabewerten die größte bzw. kleinste Zahl herauszufinden
- Hierzu sind im Prinzip nur zwei Speicherzellen bzw. Variablen erforderlich
  - Eine für den Eingabewert
  - Eine weitere für das gegenwärtige Maximum bzw. Minimum
- Beim Sortieren stellt sich die Situation anders dar:
  - Hier ist ein Vergleich aller Zahlenwerte untereinander erforderlich

Zahlen sortieren (2)

- Die Zahlen müssen daher *gleichzeitig* in *unterschiedlichen* Variablen gehalten werden
  - Anderenfalls wäre wiederholtes Eingeben der gleichen Zahlen erforderlich
- Einzeln benannte Variablen wie **`var1`**...**`varN`** stellen dabei keine Lösung dar:
  - Die Anzahl der zu sortierenden Zahlenwerte muss variabel sein können

Zahlen sortieren (3)

- Keine variable Anzahl von Eingabewerten möglich!



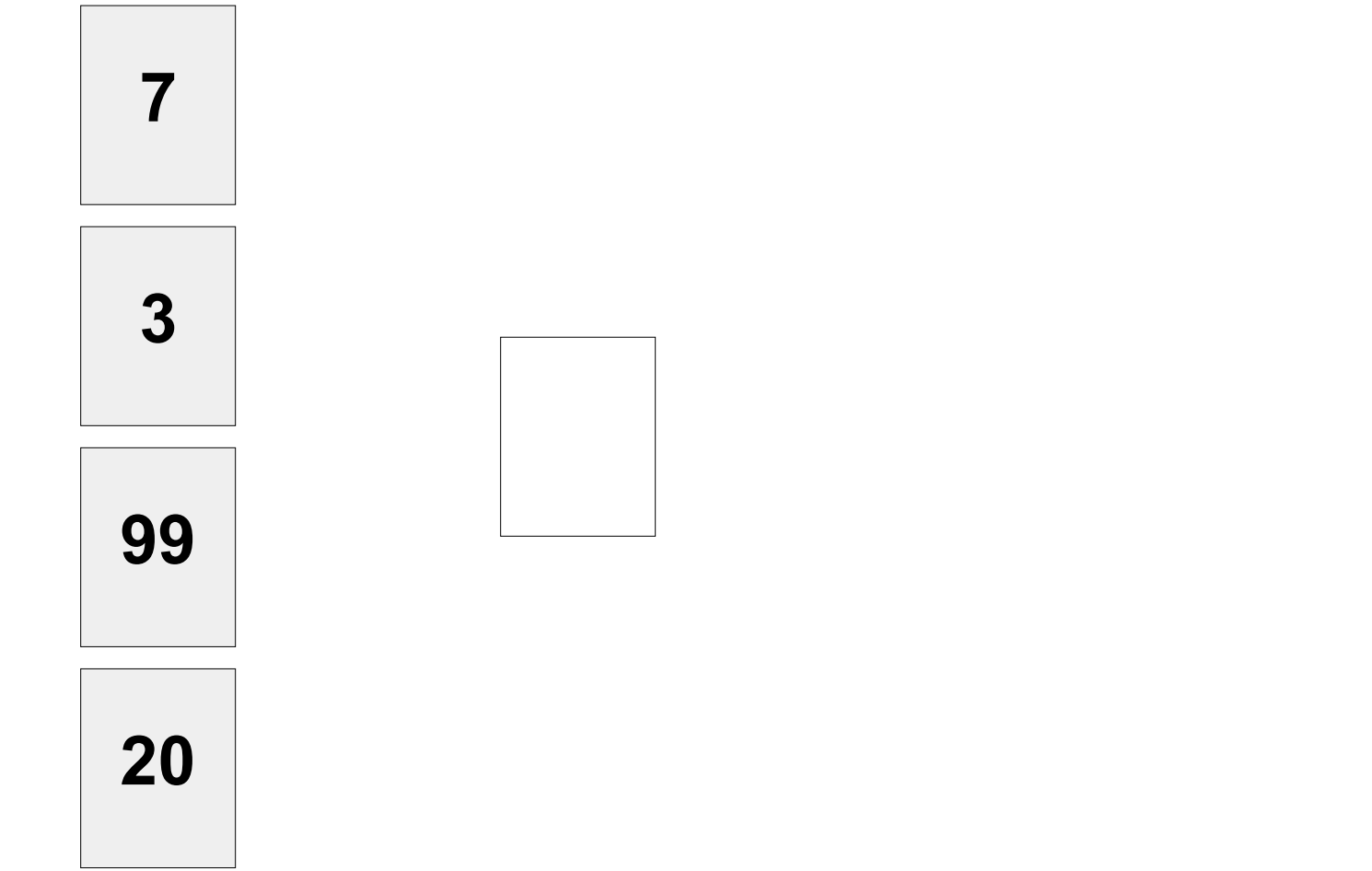
Zahlen sortieren: Methode

- Hinsichtlich des Vergleichs der Zahlenwerte stellen geschachtelte IF-Anweisungen keine brauchbare Lösung dar
- Eine variable Anpassung an unterschiedliche Mengen von Eingabewerten ist hier nicht praktikabel
- Für jede Anzahl von Zahlenwerten muss eine eigene Lösung programmiert werden
- Anzahl der IF-Anweisungen nimmt mit der Anzahl der Zahlenwerte entsprechend der Fakultät zu
- Daher geschicktere Lösung erforderlich

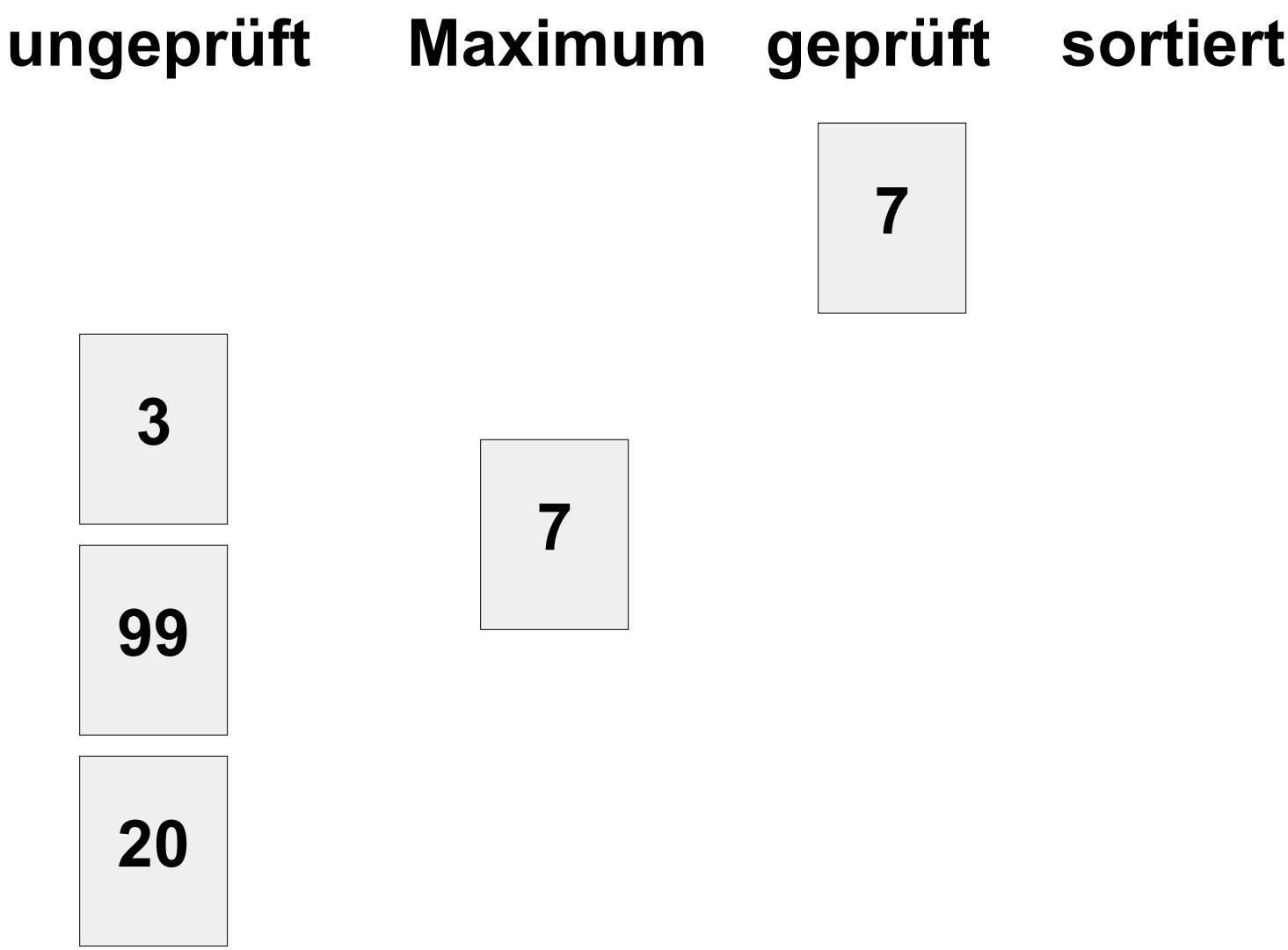
Iteratives Sortieren

- Die folgende Menge von Zahlen sei absteigend zu sortieren:
- 7 3 99 20
  - Wir können zu diesem Zweck unseren Algorithmus für die Maximumsuche verwenden, indem wir diesen iterativ anwenden
    - Die gefundenen Maxima dienen dazu, das Feld in einen sortierten und unsortierten Teil aufzuteilen
    - Mit jedem gefundenen Maximum wächst der sortierte Teil um ein Feldelement
    - Die Maximumsuche erfolgt jeweils im unsortierten Teil
    - Zu Beginn umfasst der unsortierte Teil das gesamte Feld und der sortierte Teil ist leer
  - Dieser Sortieralgorithmus wird auch als **Maxsort** oder **Selectionsort** bezeichnet

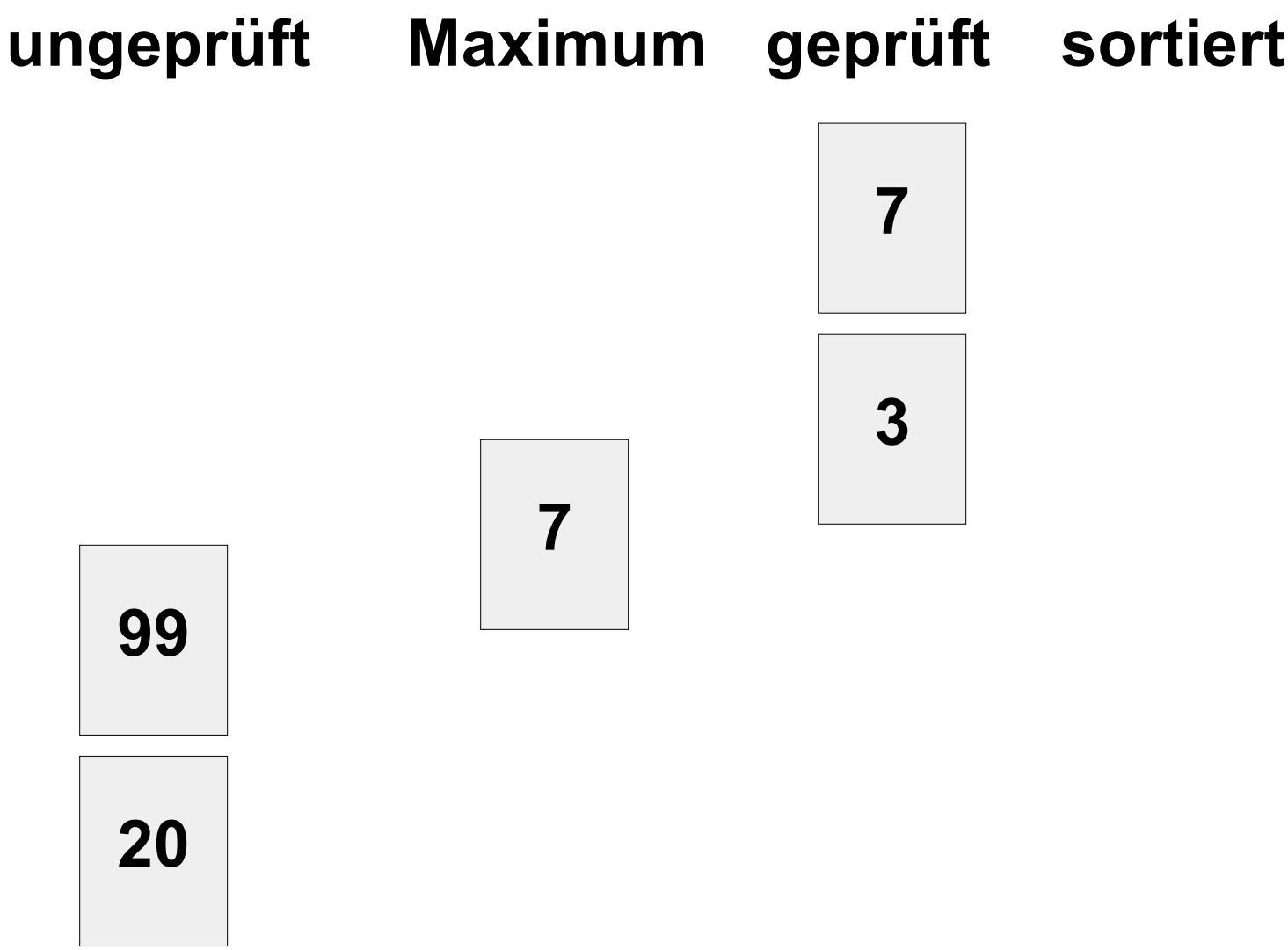
Sortieren mit Maximumsuche (1)



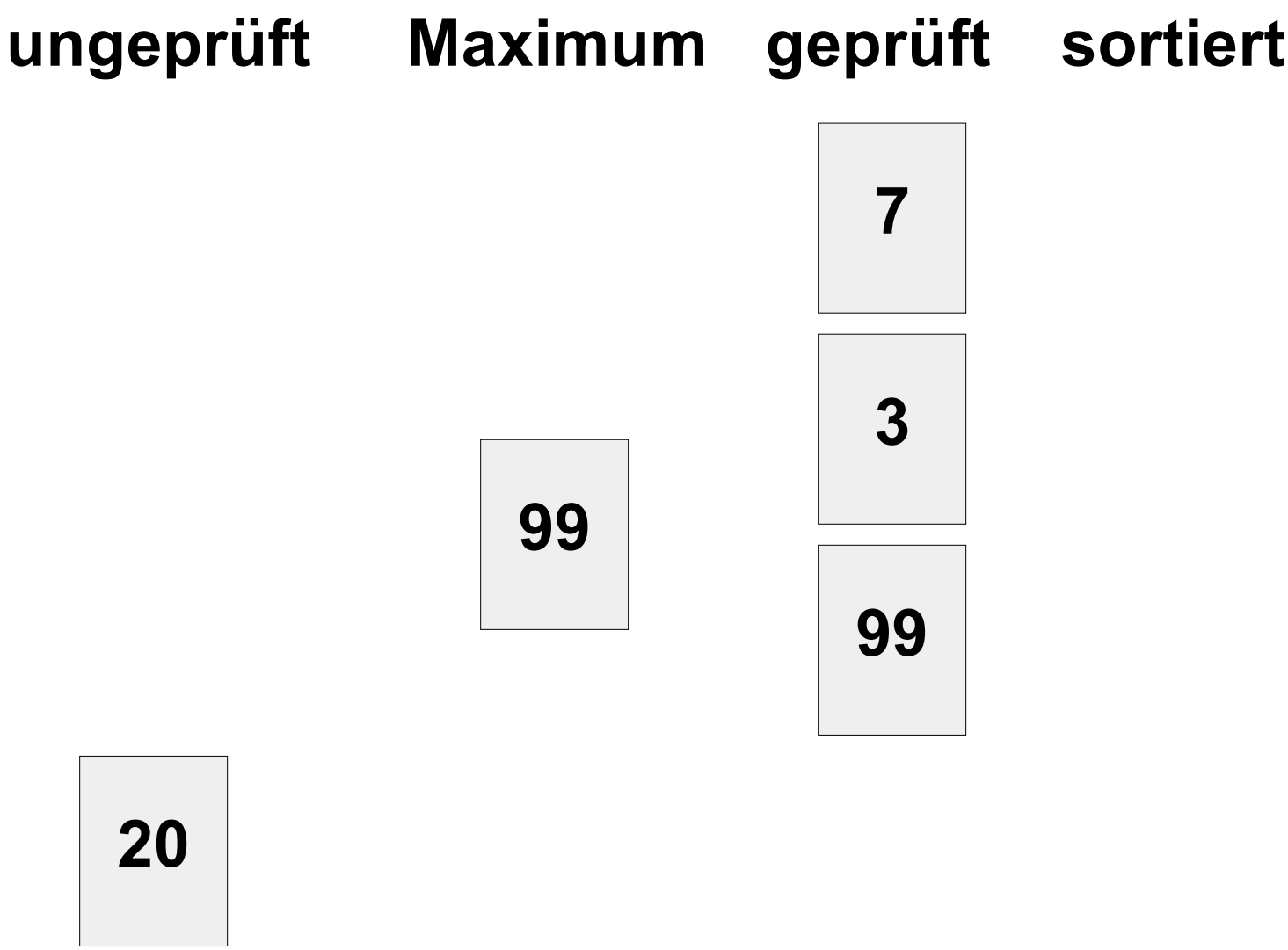
Sortieren mit Maximumsuche (2)



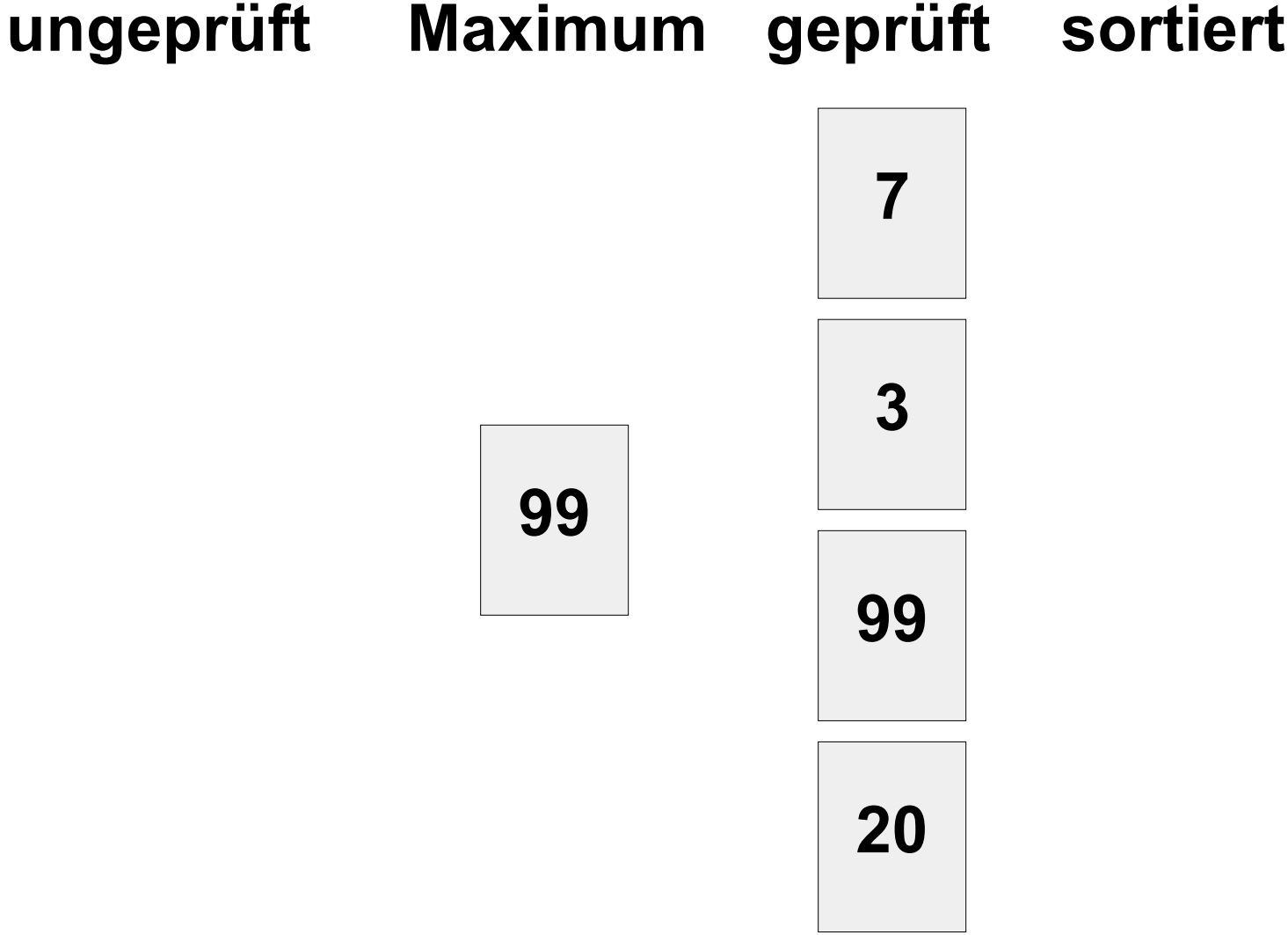
Sortieren mit Maximumsuche (3)



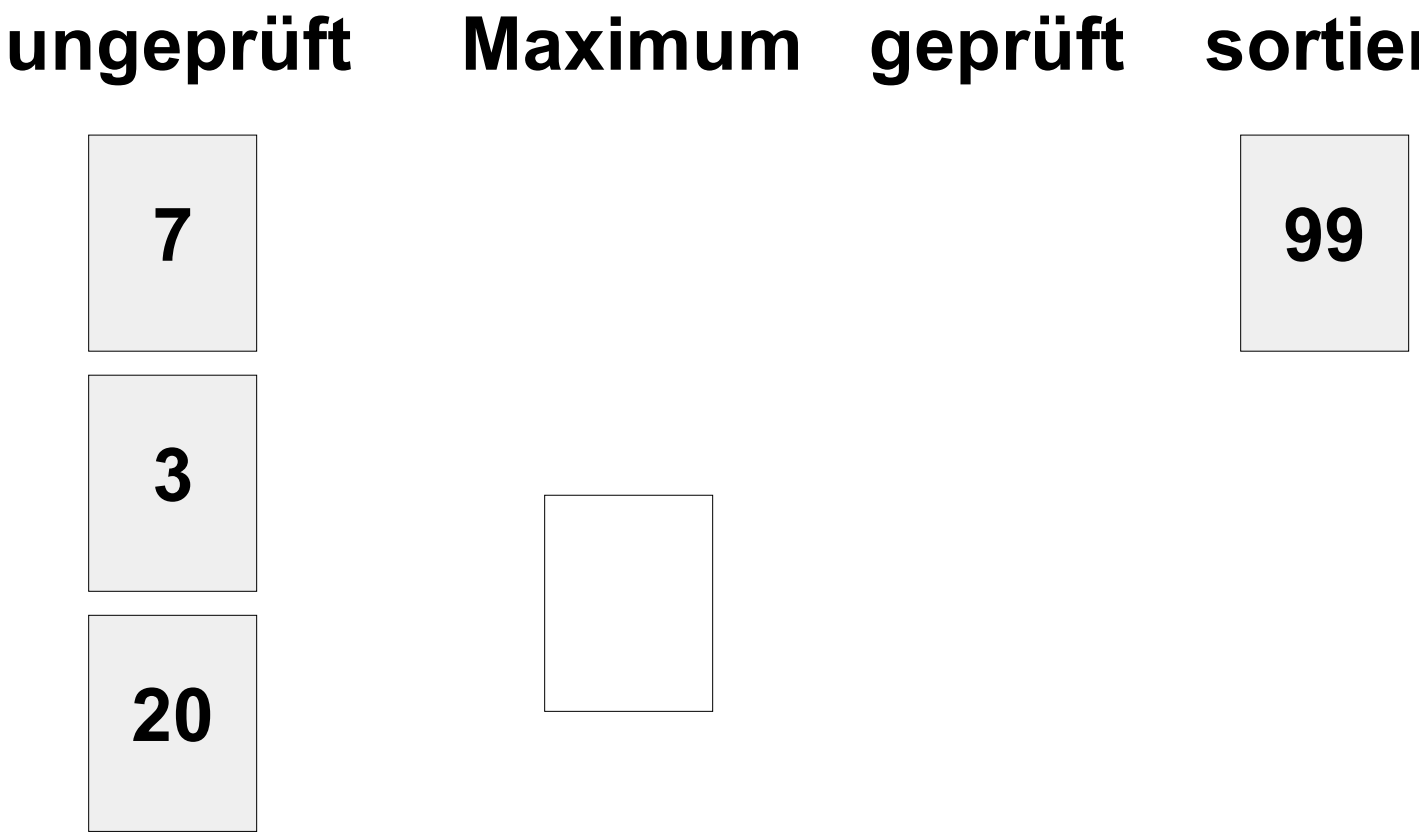
Sortieren mit Maximumsuche (4)



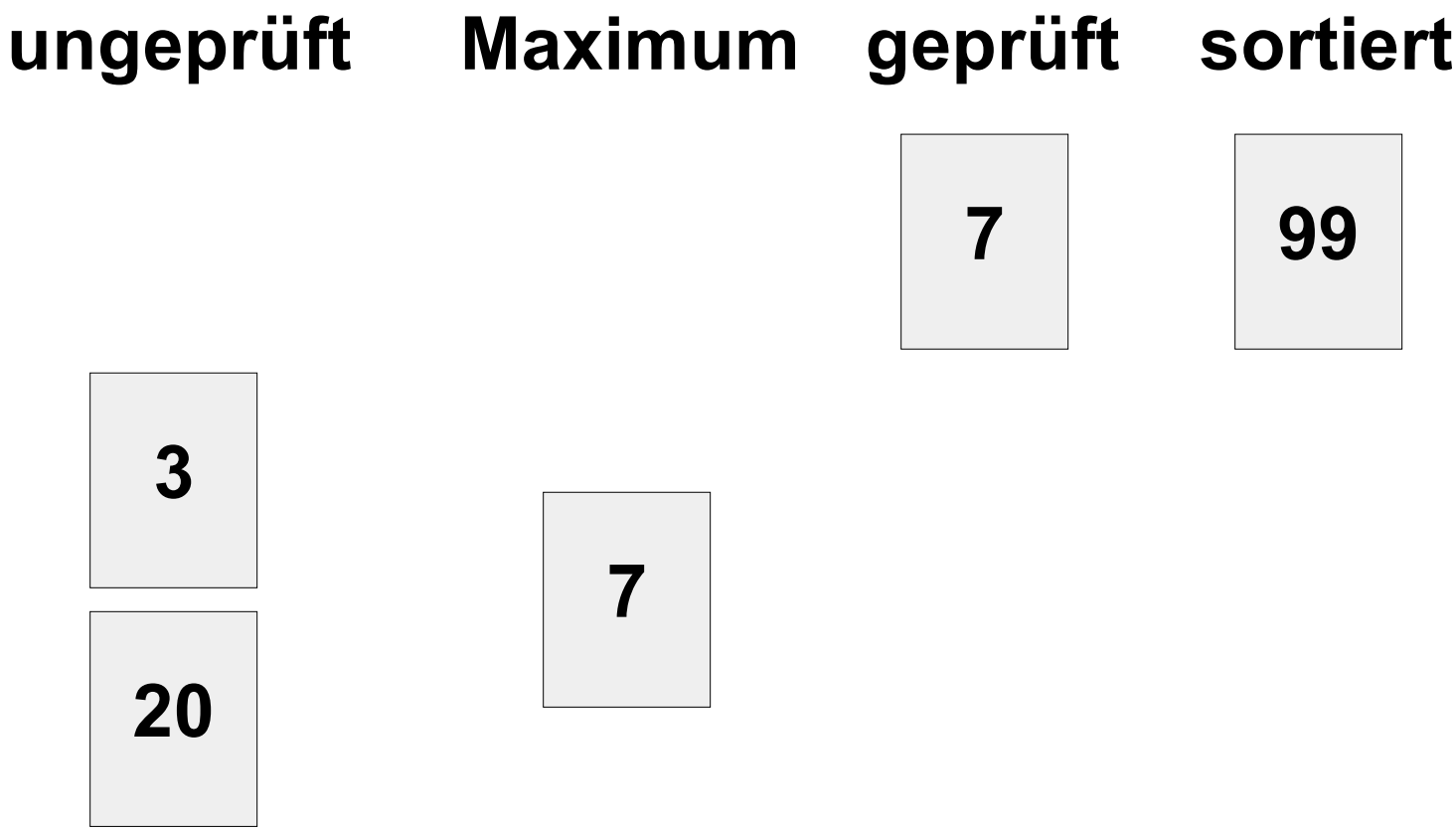
Sortieren mit Maximumsuche (5)



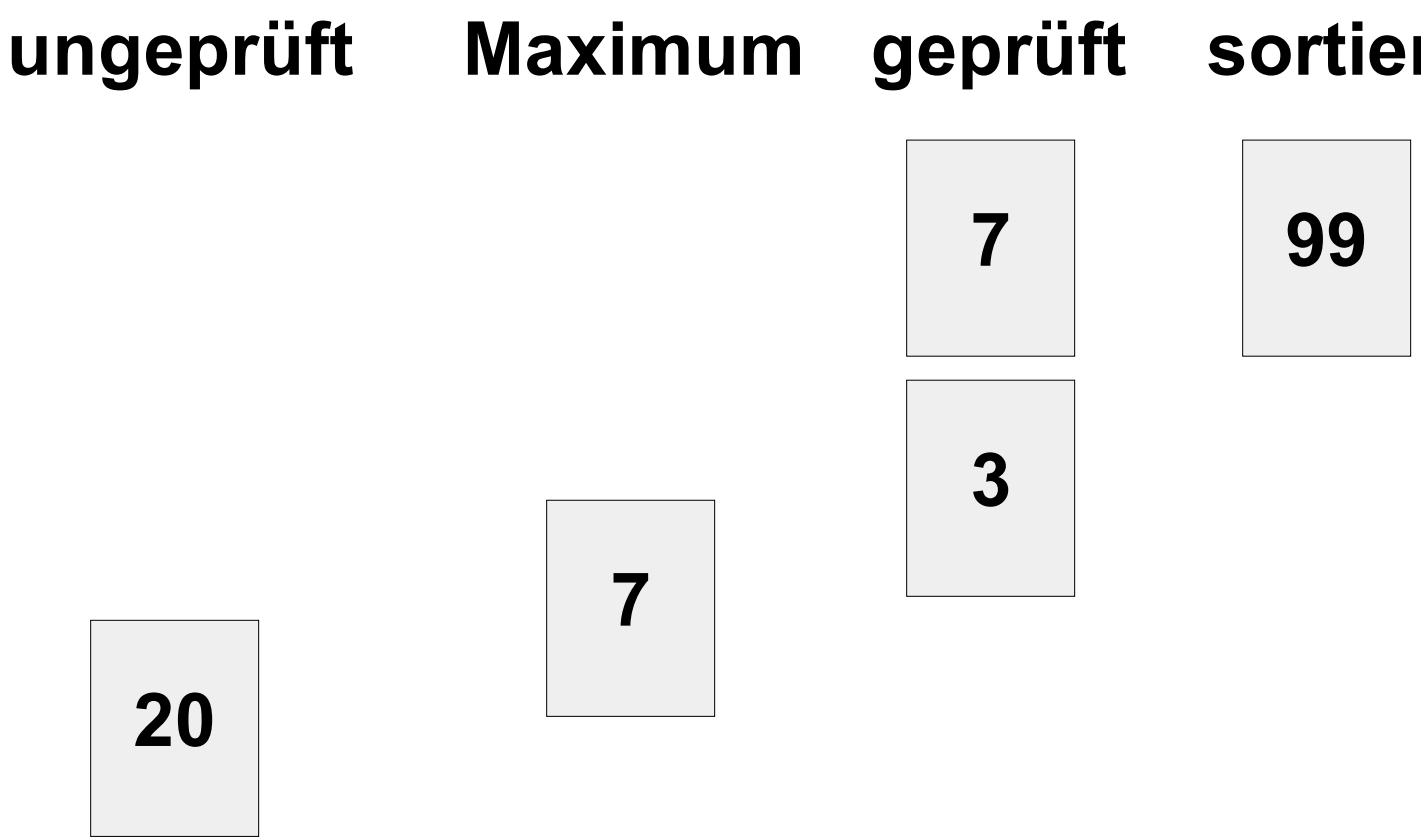
Sortieren mit Maximumsuche (6)



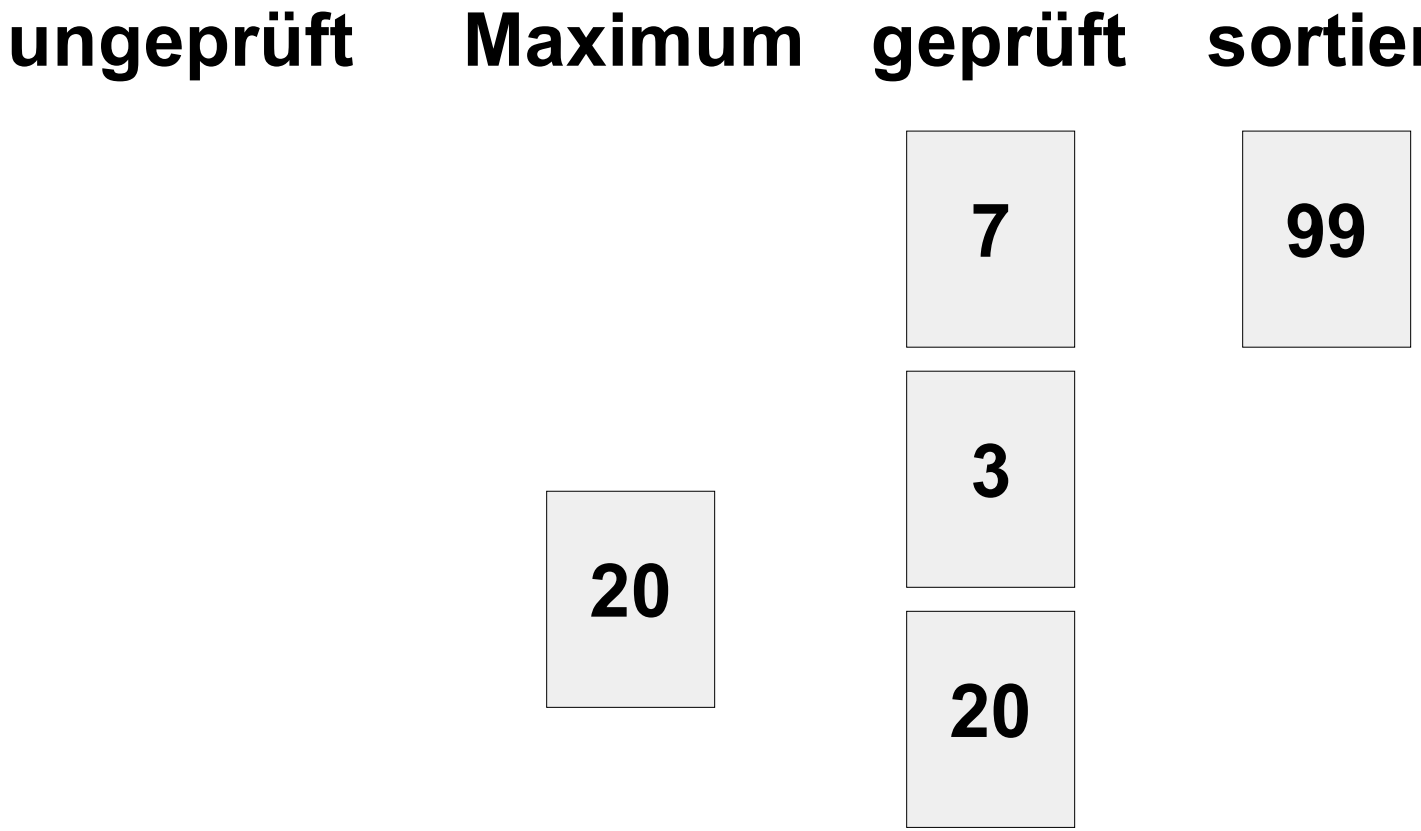
Sortieren mit Maximumsuche (7)



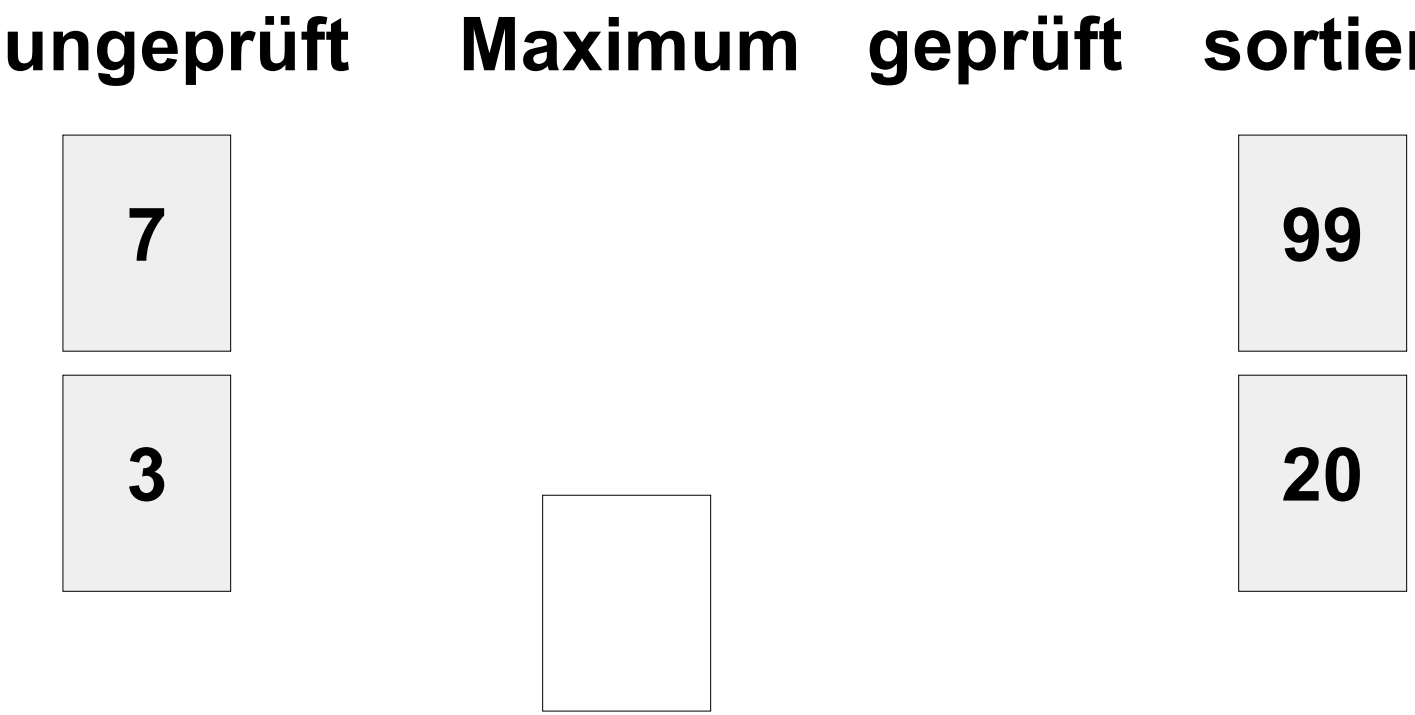
Sortieren mit Maximumsuche (8)



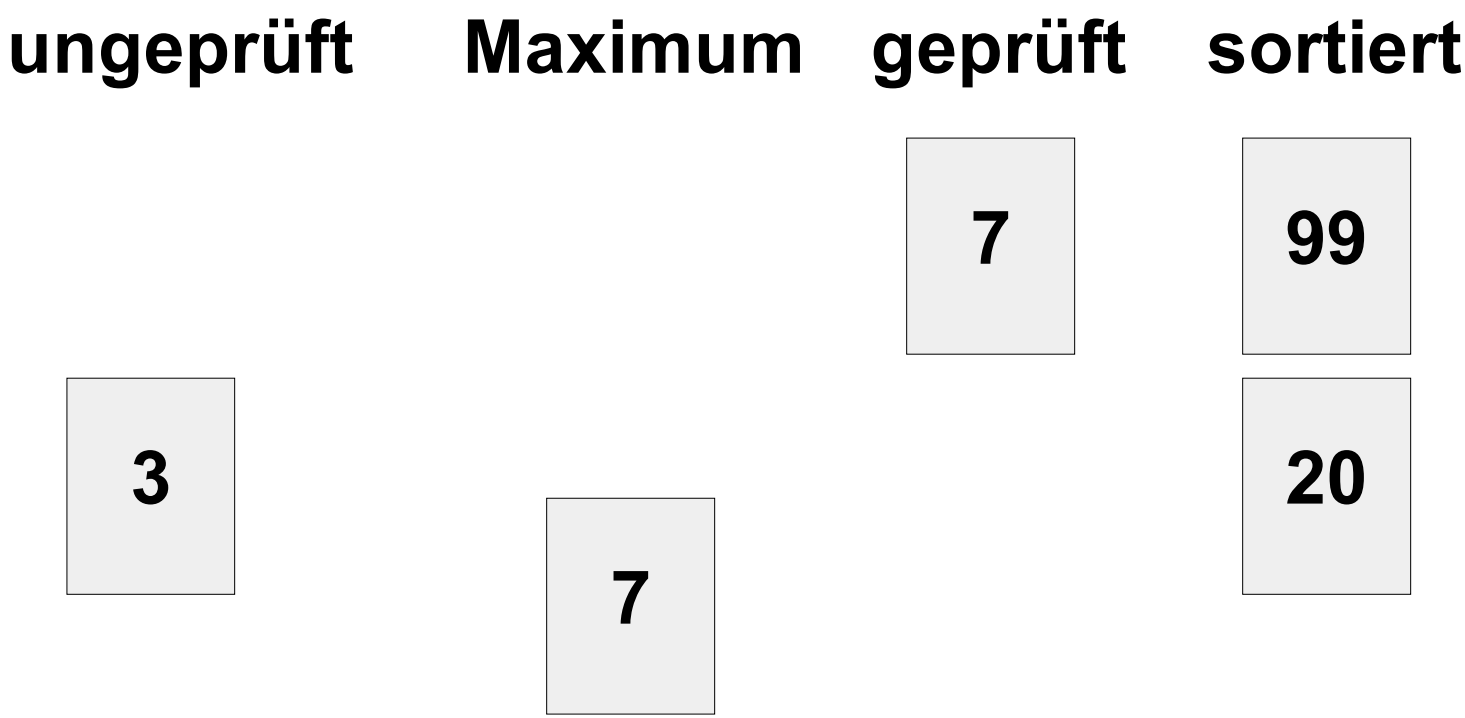
Sortieren mit Maximumsuche (9)



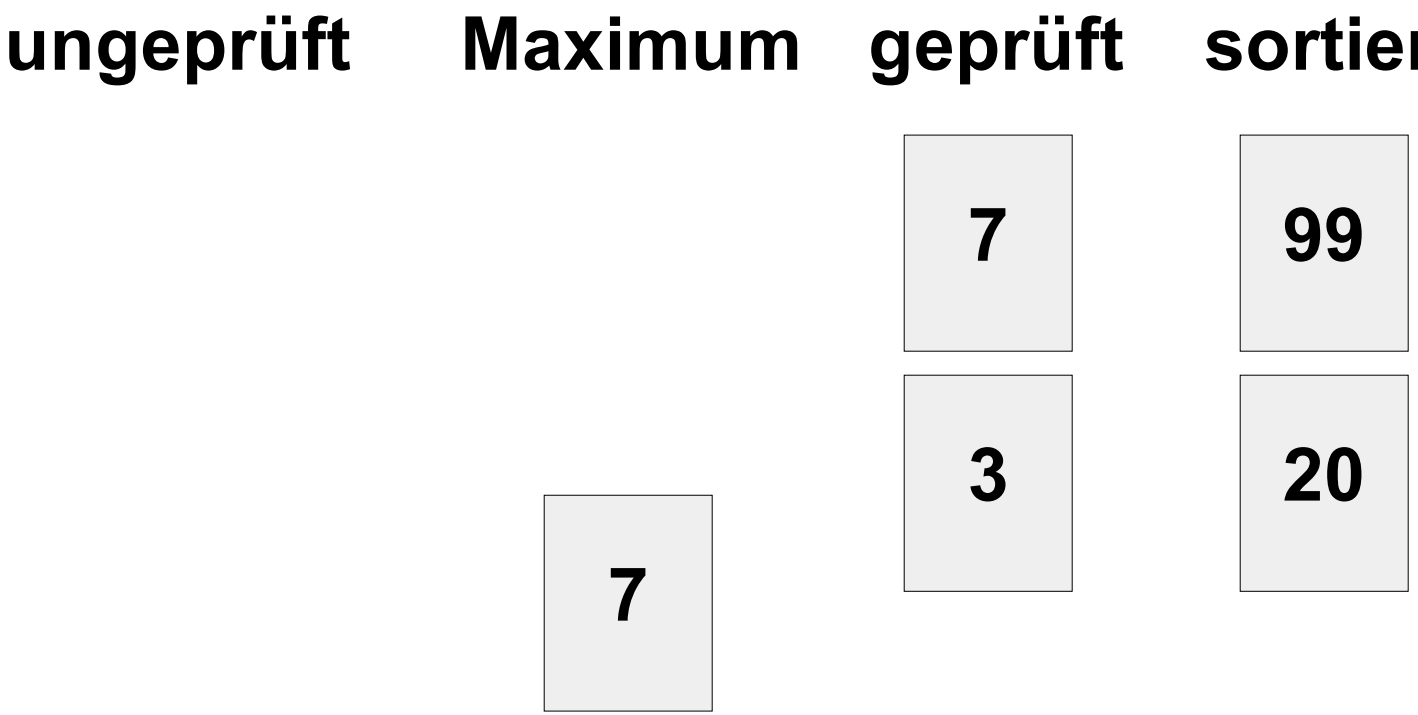
Sortieren mit Maximumsuche (10)



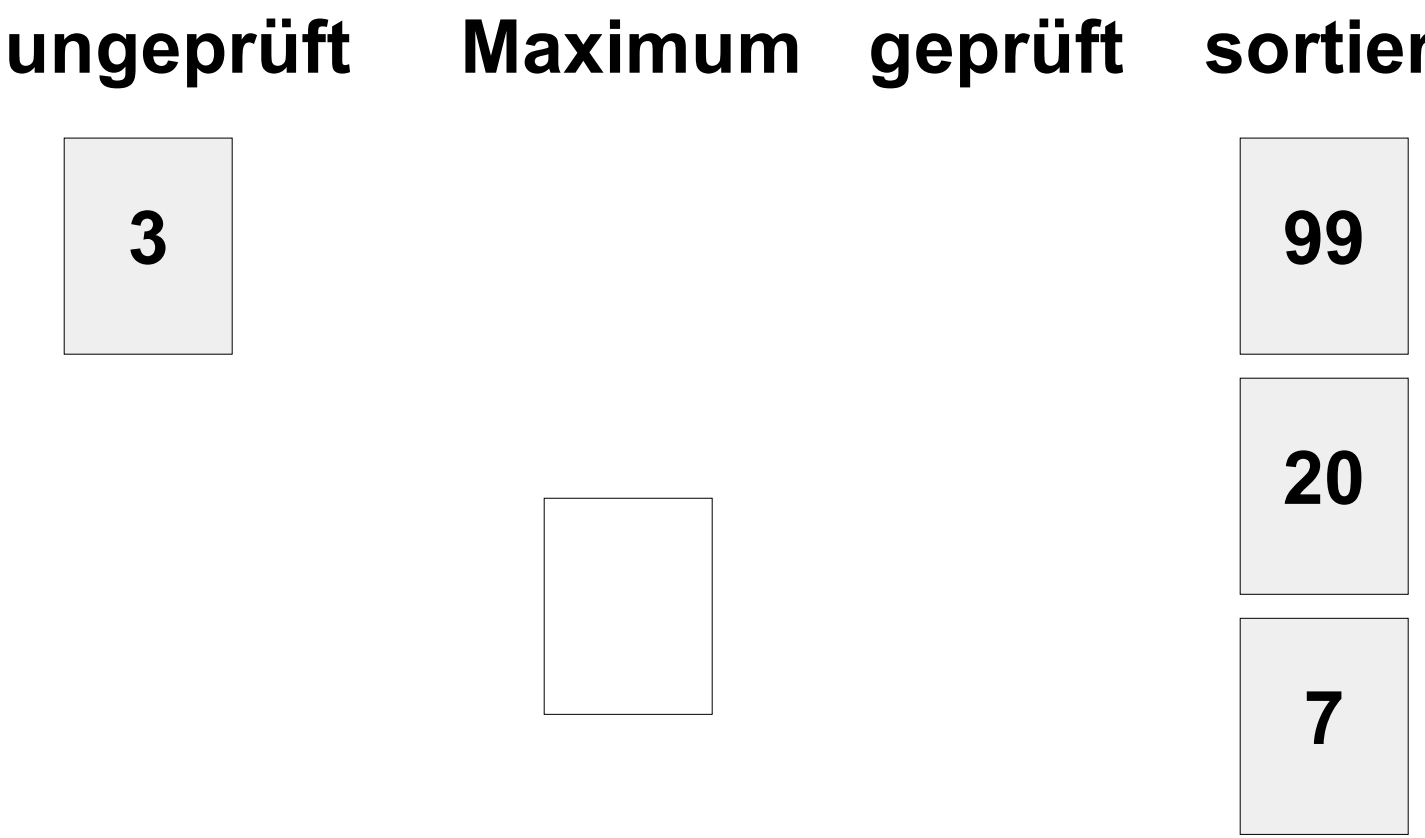
Sortieren mit Maximumsuche (11)



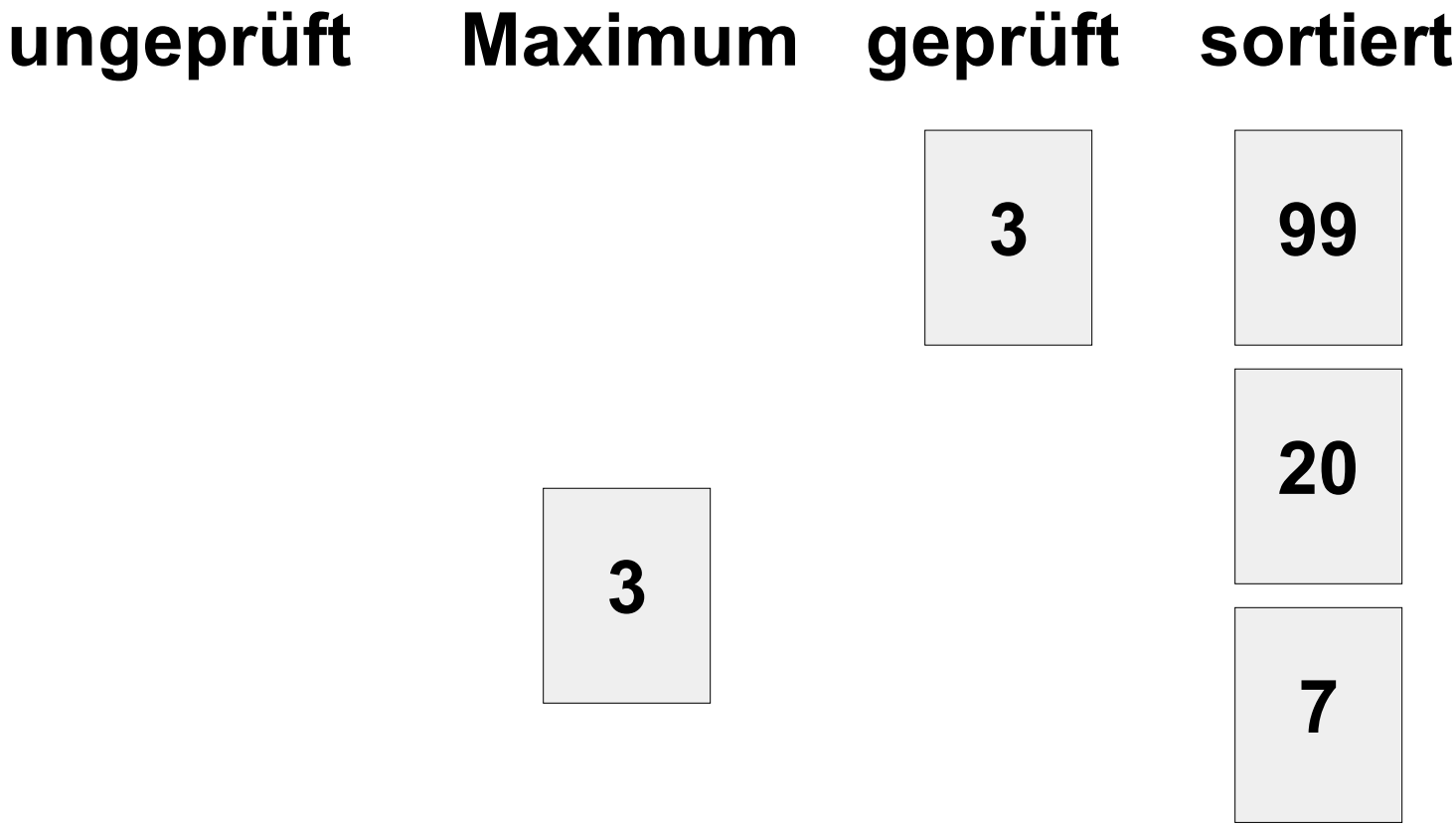
Sortieren mit Maximumsuche (12)



Sortieren mit Maximumsuche (13)



Sortieren mit Maximumsuche (14)



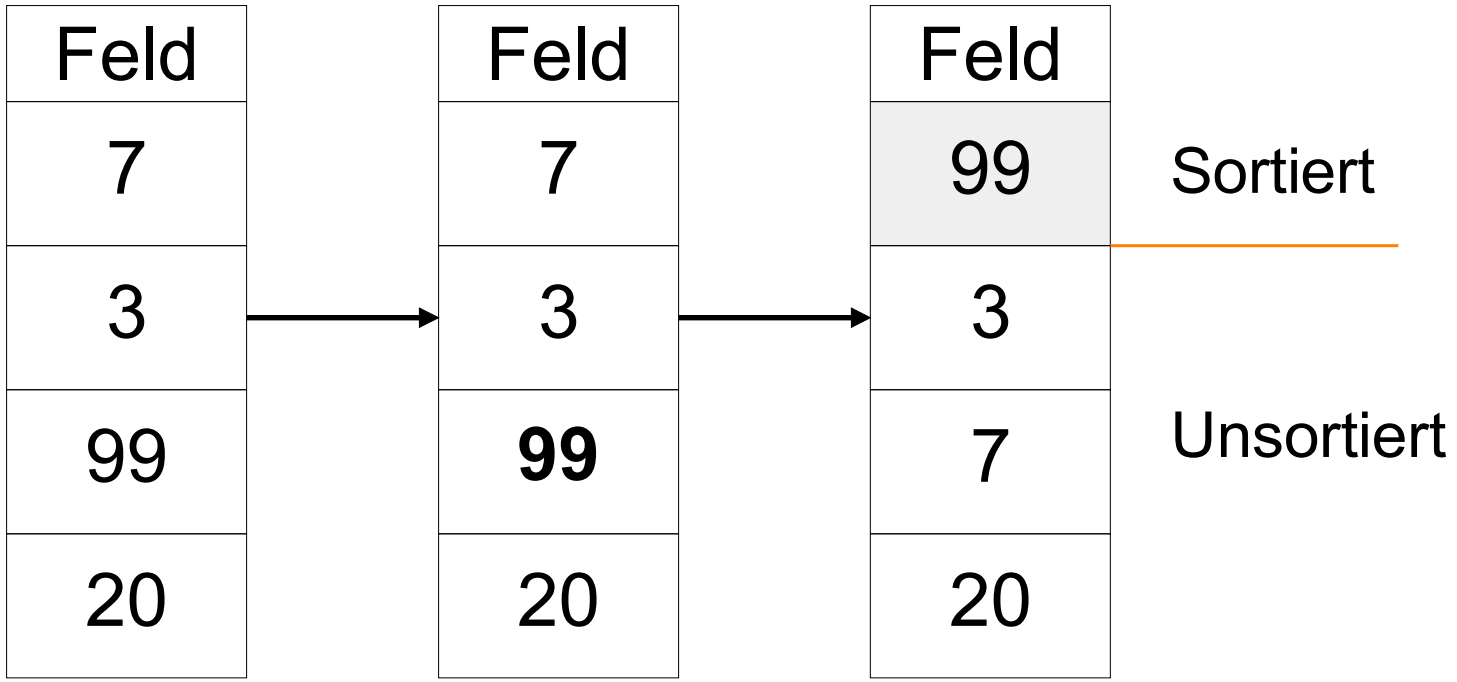
Sortieren mit Maximumsuche (15)

ungeprüft	Maximum	geprüft	sortiert
			99
			20
			7
			3

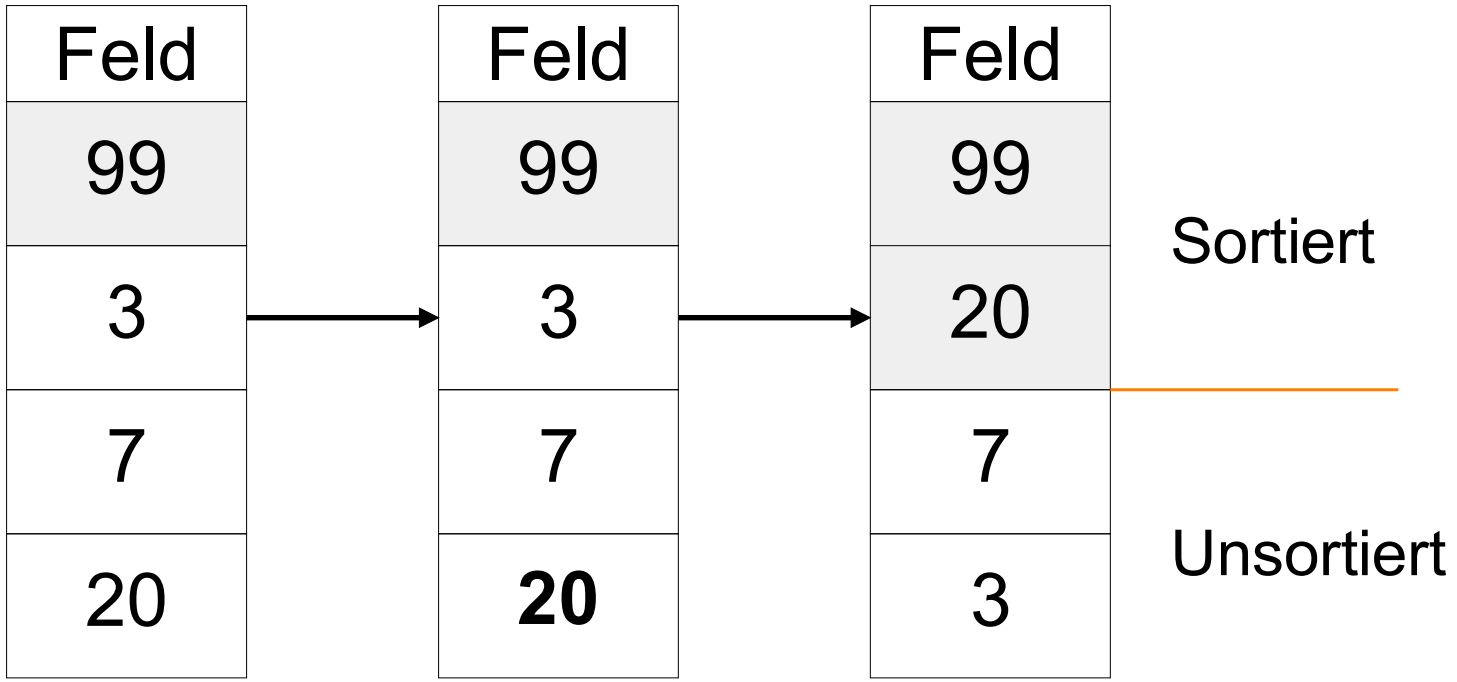
Maxsort (1)

- Die Maximumsuche wird nun als Teil einer Schleife aufgerufen, die jeweils im zu sortierenden Rest des Feldes das Maximum bestimmt und dieses an den Anfang dieses Teilfeldes verschiebt
- Zu Anfang entspricht der Rest des Feldes dem Feld selbst

Maxsort (2)

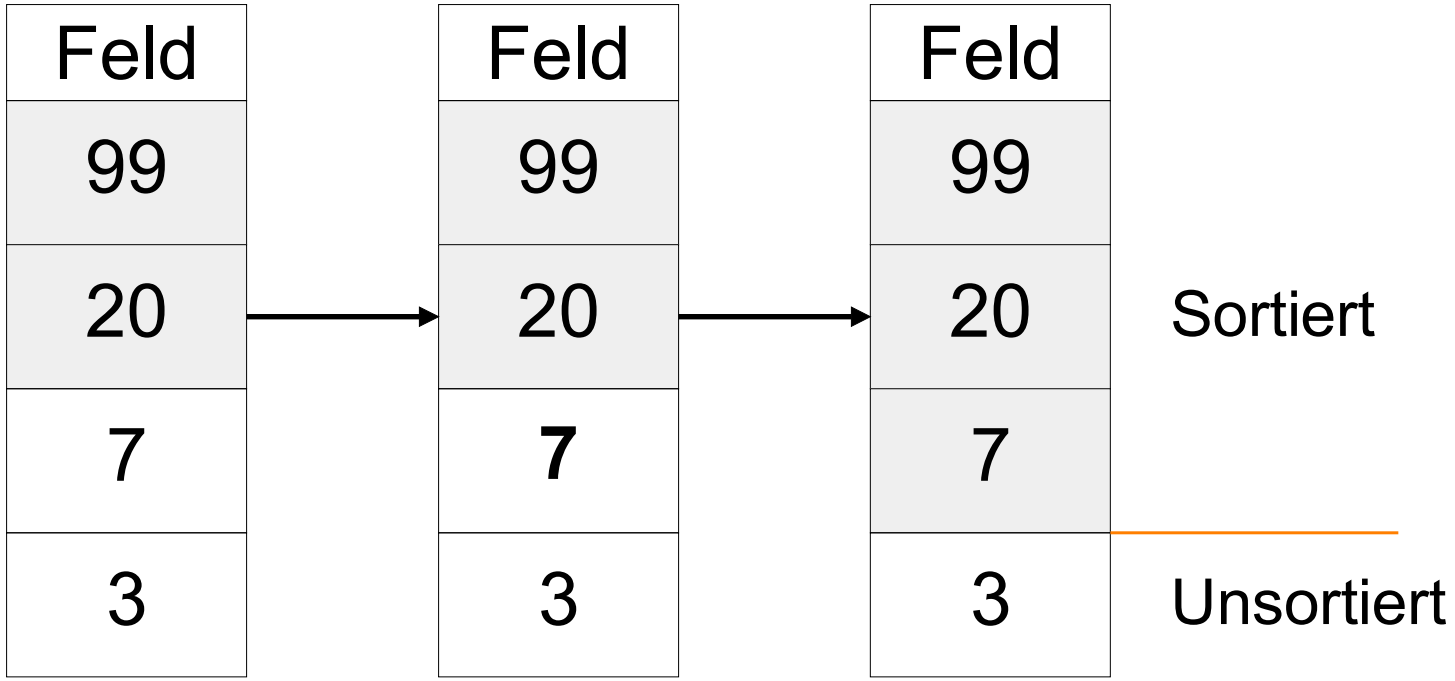

Feld
7
3
99
20
Feld
7
3
99
20
Feld
99
3
7
20
Sortiert
Unsortiert
1. Maximumsuche
2. Vertauschen

Maxsort (3)

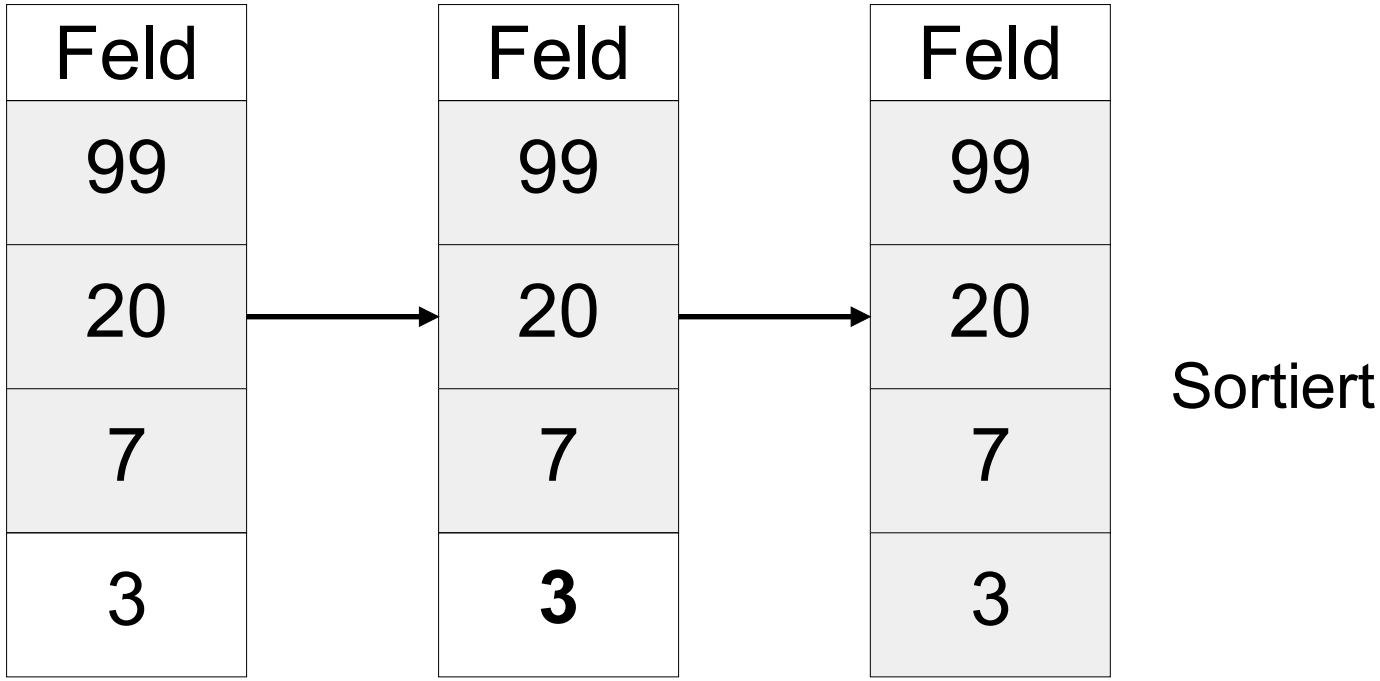


1. Maximumsuche 2. Vertauschen

Maxsort (4)


Feld
99
20
7
3
Feld
99
20
7
3
Feld
99
20
7
3
Sortiert
Unsortiert
1. Maximumsuche
2. Vertauschen

Maxsort (5)


Feld
99
20
7
3
Feld
99
20
7
3
Feld
99
20
7
3
Sortiert
1. Maximumsuche
2. Vertauschen

Maxsort: Erste Verfeinerung

```
// Deklariere und initialisiere Variablen
// Setze Zeiger auf erste Feldposition
// Wiederhole solange Zeiger <= letztes Feldelement:
  // Suche Maximum im Feld beginnend bei Zeiger
  // Vertausche Maximum mit Wert bei Zeiger
  // Setze Zeiger ein Feldelement weiter
// Gib sortiertes Feld aus
```

Maxsort: Zweite Verfeinerung

```
// Deklariere und initialisiere Variablen

int [] arr = {7, 3, 99, 20}; // Zu sortierende Werte

int max;      // Hält Wert des derzeitigen Maximums
int maxPtr;   // Hält Position im Feld des derz. Max.
int swap;      // Zwischenspeicherung des Werts, der
                  // gegen das Maximum ausgetauscht wird


// ...
```

Maxsort: Dritte Verfeinerung

```
// Deklariere und initialisiere Variablen
// Setze Zeiger "ptr" auf erste Feldposition
// Wiederhole solange Zeiger <= letztes Feldelement:
for (int ptr = 0; ptr < arr.length; ptr++) {
  // Suche Maximum im Feld beginnend bei Zeiger
  // Vertausche Maximum mit Wert bei Zeiger
}
// Gib sortiertes Feld aus
```

Maxsort: Vierte Verfeinerung

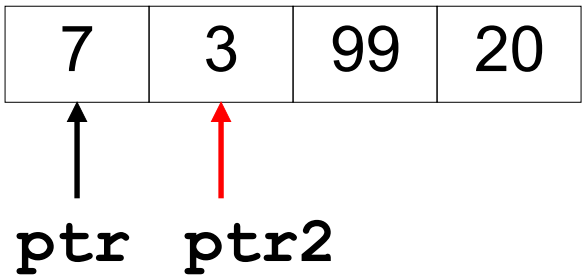
```
// Suche Maximum im Feld beginnend bei Zeiger:

// Dies entspricht der Maximumsuche auf einem Feld

// => Wurde oben schon spezifiziert:

max = arr[ptr];
maxPtr = ptr;

for (int ptr2 = ptr+1; ptr2 < arr.length; ptr2++) {
  if (arr[ptr2] > max) {
    max = arr[ptr2];
    maxPtr = ptr2;
  }
}
```



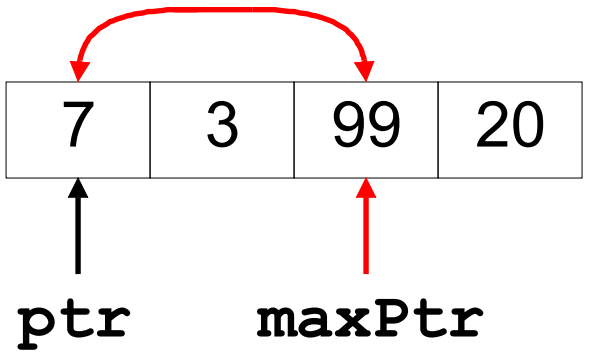
Maxsort: Fünfte Verfeinerung

```
// Vertausche Maximum mit Wert bei Zeiger:

swap = arr[ptr];

arr[ptr] = arr[maxPtr];

arr[maxPtr] = swap;
```



Maxsort: Sechste Verfeinerung

```
// Gib sortiertes Feld aus:

for (int i = 0; i < arr.length; i++)
  println(arr[i]);
```

Aufwand

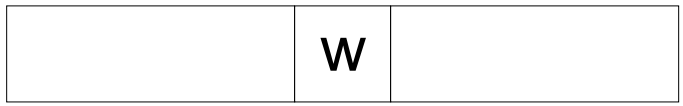
Für n=4 Feldelemente ergeben sich also 3+2+1 = 6 = **n*(n-1)/2** Vergleiche!

Aufwand

- Der Aufwand unserer Maxsort-Methode hat also die Größenordnung n^2!
- Sortieralgorithmen waren lange Zeit eines der Hauptthemen der Informatik
- Ein sehr schnelles *rekursives* Verfahren ist Quicksort (C. A. R. Hoare, 1960)
  - Bei Quicksort wird das zu sortierende Feld von Zahlen sukzessive in zwei zu sortierende Hälften aufgeteilt, auf die Quicksort angewendet wird
  - Quicksort liegt auch dem in Java implementierten Sortieralgorithmus **`sort()`** zugrunde (s. Klasse **`Arrays`**)

Quicksort

Array A

1. 

Ein Feldelement w wird willkürlich ausgewählt

2. 

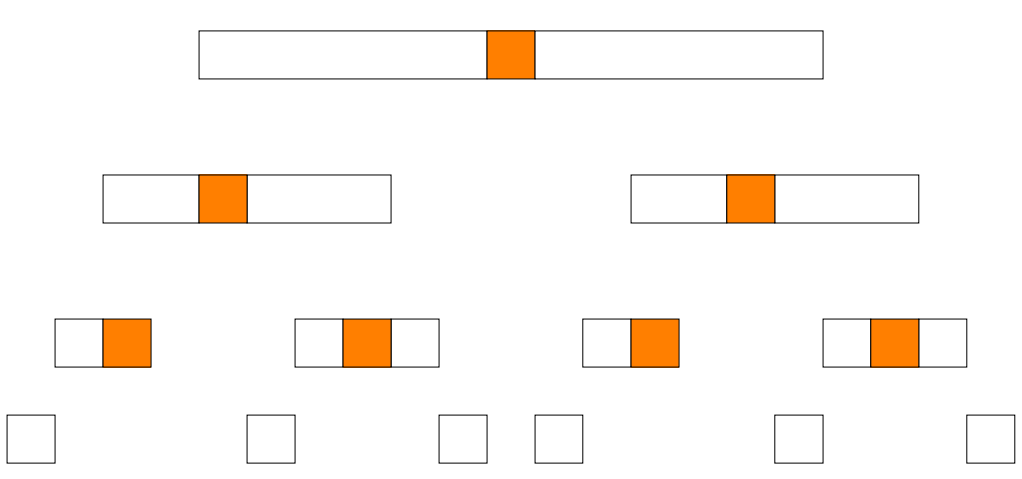
A wird so umsortiert, dass:
A1 alle Elemente kleiner w enthält
A2 alle Elemente größer w enthält

3. 

Quicksort wird **rekursiv** auf A1 und A2 angewendet, d. h. Fortsetzung bei 1. für A1 und A2

Quicksort: Aufwand

Partitionenbaum



$\log_2(N)$ Ebenen

N Elemente verschiebbar pro Ebene

Hieraus resultiert ein Aufwand in der Größenordnung $N*\log_2(N) < N^2$

Aufwandsvergleich

Anzahl N der Zahlenwerte	Maxsort	Quicksort
10	45	33
100	4950	664
1000	499500	9966
100000	4999950000	1660964

Objektorientierte Programmierung

Daten speichern

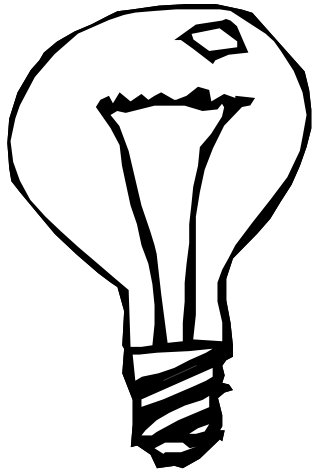
- Bisher kennen wir zwei Möglichkeiten Daten zu speichern:
  - Variablen und Felder
- Hiermit lassen sich einzelne Zahlen, Zeichen, Zeichenketten oder Zahlenfolgen speichern
- Was passiert im Falle von Objekten mit komplexerer Struktur?
- Beispiel: Adressdaten eines Kunden (Name, PLZ, etc.)
  - Derartige Daten können nicht in einem Feld gespeichert werden
  - Ein Feld besitzt nur einen Datentyp
  - Die Adressdaten bestehen aber aus mehreren Typen
  - Hierfür ist eine komplexere Datenstruktur erforderlich

Klassen und Objekte

- Für die Speicherung komplexer Datenstrukturen gibt es das Konzept der Klasse (engl.: *class*)
- Die Klasse ist ein abstrakter Datentyp, der Daten, repräsentiert durch Variablen, und Methoden, die auf diesen Daten arbeiten, zusammenfasst
- Objekte bzw. Instanzen einer Klasse sind Elemente dieses Datentyps
  - Diese müssen gesondert erzeugt werden

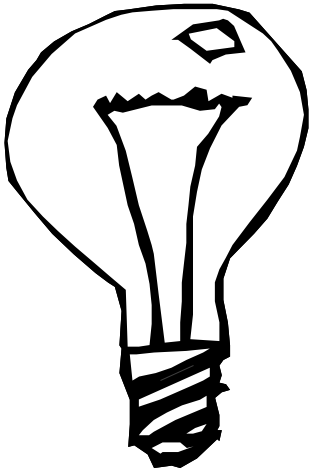
Klassen

- Klassen modellieren abstrakte oder reale Gegenstände in Form eines "Bauplans"
- Genauer: Die Eigenschaften (engl.: *properties*) dieser Gegenstände
- Beispiel: Eine Glühbirne
- Wie lassen sich die Eigenschaften einer Glühbirne definieren?



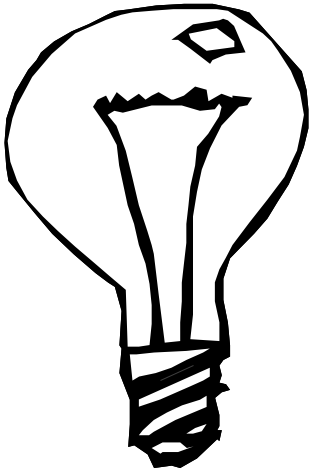
Eigenschaften: Physikalischer Aufbau (1)

- Die Glühbirne besitzt einen spezifischen *physikalischen Aufbau*
  - Luftleerer Glaskolben
  - Metallischer Schraubsockel
  - Glühwendel
  - etc.



- Welche von diesen Eigenschaften in Software zu modellieren sind und in welcher Art ist von der Anwendung abhängig
- In vielen Fällen ist eine abstrakte Beschreibung ausreichend

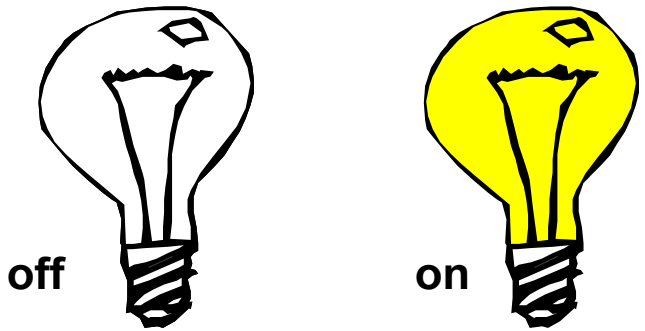
Eigenschaften: Physikalischer Aufbau (2)



- Beispiel: Internet-Kaufhaus
  - Hier interessieren die für den Verkauf wichtigen Daten der Glühbirne
    - Art (LED, Halogen, etc.), Wattzahl, Farbe, etc.
- Beispiel: Computerspiel bzw. virtuelle Umgebung
  - Hier sind der Beleuchtungstyp und das Aussehen der Glühbirne in Form des computergrafischen Modells wichtig
  - Für den Verkauf wichtige Daten sind dagegen unwichtig
- Die für den jeweiligen Anwendungsfall relevanten Daten werden in den internen Variablen des Softwareobjekts codiert

Eigenschaften: Zustand (1)

- Objekte können sich in unterschiedlichen Zuständen befinden
- Beispiel: Die Glühbirne kann ein- oder ausgeschaltet sein, heil oder kaputt

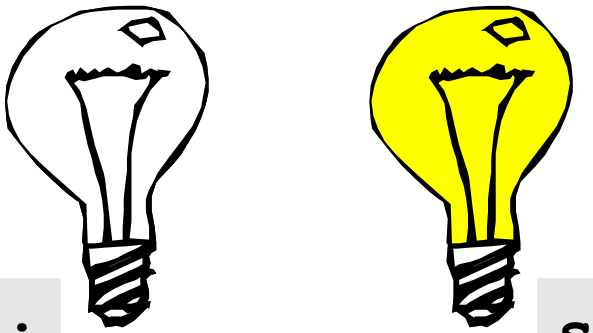


- Auch hier hängt es vom jeweiligen Anwendungsfall ab, welche Zustände benötigt werden und damit zu modellieren sind

Eigenschaften: Zustand (2)

- Der Zustand eines Software-Objekts wird durch die Werte seiner internen Variablen codiert
  - Der Zustand der internen Variablen repräsentiert den "Weltzustand" des Softwareobjekts
  - Alles, was das Objekt "über sich" und "seine Umgebung" "weiß"

```
boolean switchedOn;
```

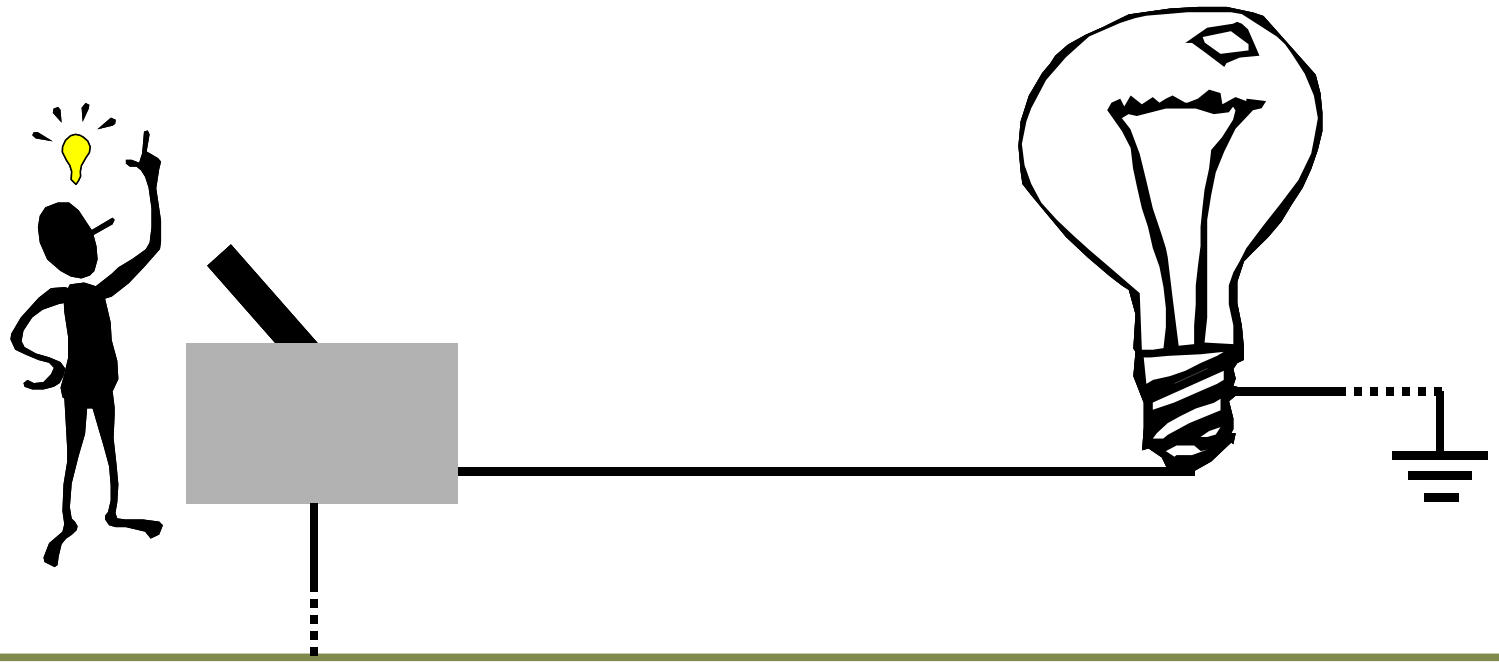


```
switchedOn = false;
```

```
switchedOn = true;
```

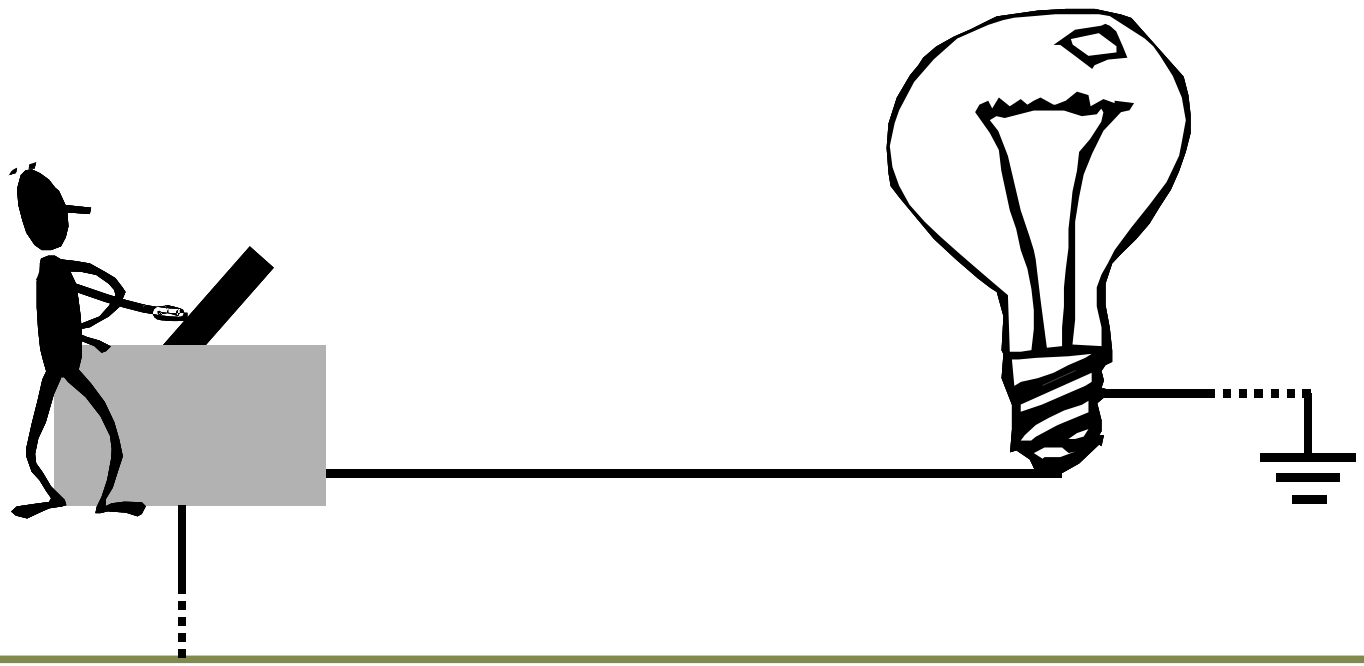
Eigenschaften: Verhalten (1)

- Objekte zeigen ein spezifisches Verhalten
- Sie reagieren auf Anfragen von außen
- Sie "kommunizieren" mit ihrer Außenwelt
- Beispiel: Die Glühbirne reagiert auf die "Anfrage", sich ein- oder auszuschalten



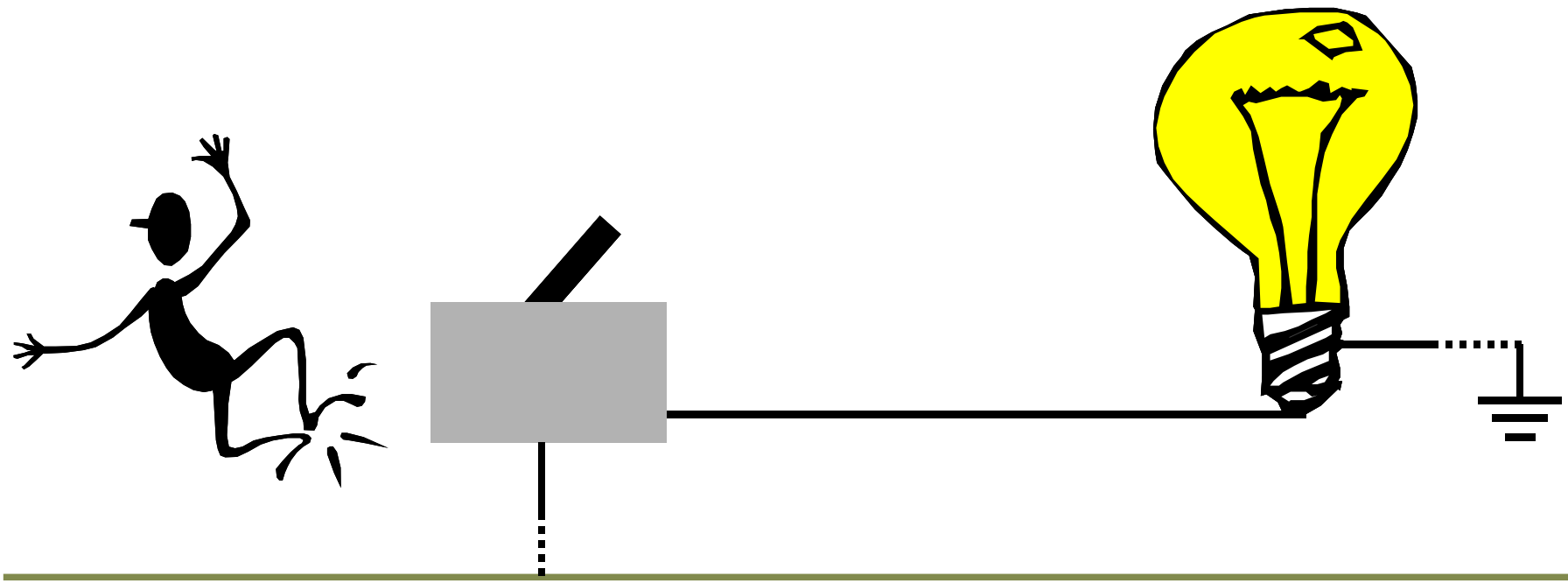
Eigenschaften: Verhalten (2)

- Objekte zeigen ein spezifisches Verhalten
- Sie reagieren auf Anfragen von außen
- Sie "kommunizieren" mit ihrer Außenwelt
- Beispiel: Die Glühbirne reagiert auf die "Anfrage", sich ein- oder auszuschalten



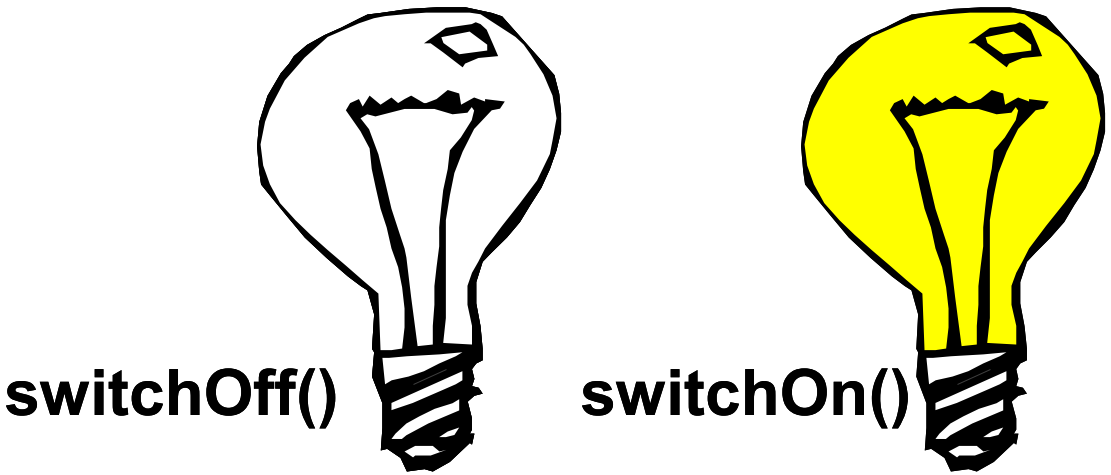
Eigenschaften: Verhalten (3)

- Derartige Verhaltensweisen werden bei der Programmierung in den **Methoden** eines Software-Objekts codiert



Methoden (1)

- Methoden modellieren spezifische Verhaltensweisen eines Software-Objekts
- Die Befehle einer Methode werden bei deren Aufruf ausgeführt
- Beispiel: Glühbirne
  - Methoden zum Ein- und Ausschalten der Glühbirne



Methoden (2)

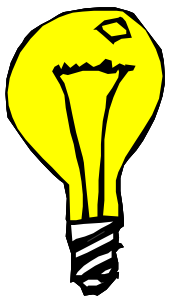
- Die Menge der zum Software-Objekt gehörenden Methoden repräsentieren alles, was das Objekt "tun" kann
  - Sein "**Verhalten**" (engl.: ***behaviour***)
- Sie definieren die Schnittstelle des Objekts zu seiner "Außenwelt"
- Methoden, die einem bestimmten Objekt zugeordnet sind, können nicht unabhängig von diesem Objekt benutzt werden!

switchOff()

isOn()

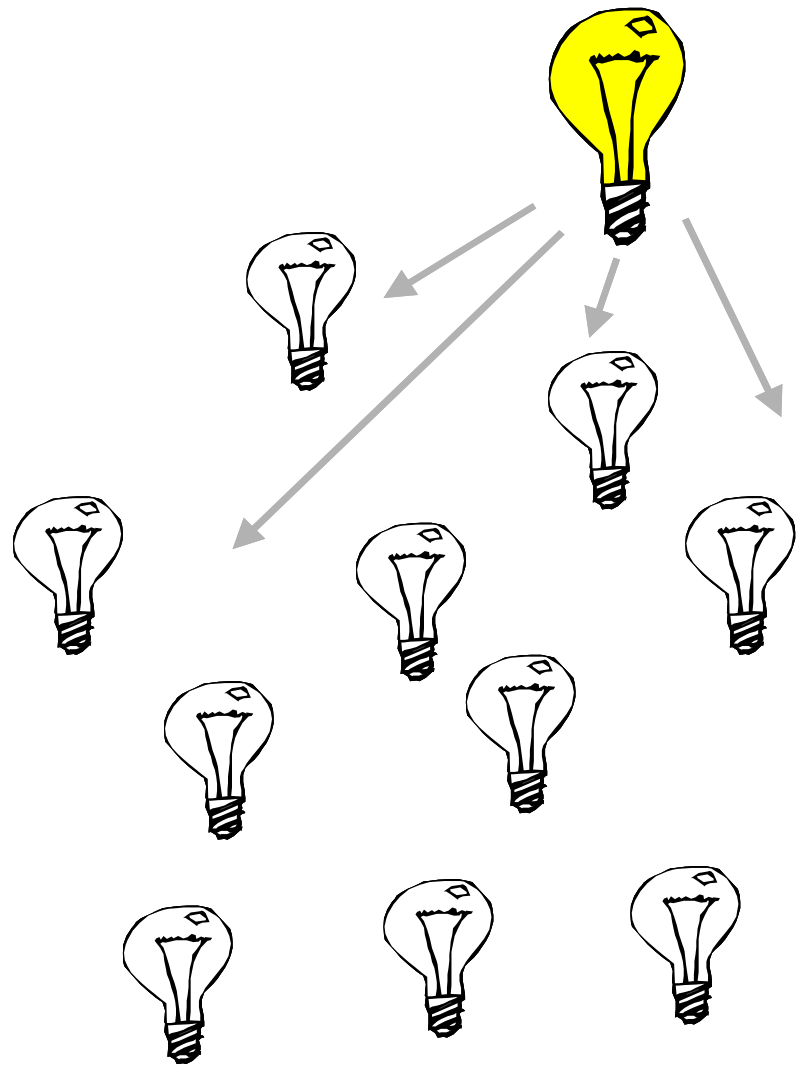
switchOn()

isOk()



Instanzierung

- Sobald wir die Eigenschaften des gewünschten Objekts in Form des "Bauplans" der **Klasse** definiert haben, können wir beliebig viele Objekte dieser Klasse erzeugen
- Jedes individuelle Objekt ist eine **Instanz** (engl.: *instance*) der Klasse
- Ein Objekt einer Klasse zu erzeugen wird **Instanzierung** (auch: Instantiierung, Instanziierung, bzw. engl.: *instantiation*) genannt



Wir halten fest:

- Eine Klasse ist ein "Bauplan" für einen realen oder abstrakten Gegenstand in Software
  - Z. B. eine Glühbirne, ein Auto, eine Segelyacht, ein Kunde, etc.
- Eine Klasse modelliert die für den jeweiligen Anwendungsfall relevanten Eigenschaften der Objekte dieser Klasse
- Durch Instanzieren der Klasse wird ein Objekt dieser Klasse gemäß des vorgegebenen "Bauplans" "gebaut"
  - Analog z. B. zum Bauen eines bestimmten Autotyps am Fließband
- Erfolgt die Entwicklung von Problemlösungen im Hinblick auf den Einsatz kommunizierender Objekte, wird von objektorientierter Programmierung gesprochen

Klassen programmieren

Beispiel: LightBulb-Klasse

- Modelliert werden soll eine Glühbirne
- Wir sind nicht am physischen Aufbau der Glühbirne interessiert
- Wichtig hinsichtlich der Modellierung soll nur der Zustand der Glühbirne sein
  - Ein- bzw. ausgeschaltet
- Die Glühbirne wird also sehr abstrakt modelliert
- Weiterhin möchten wir mittels Methoden die Glühbirne ein- und ausschalten und ihren Zustand abfragen können

Klasse deklarieren

- Eine Klasse wird mit dem Schlüsselwort class deklariert
- Eine Klasse trägt einen frei wählbaren Namen
  - Regeln hinsichtlich der Benennung wie bei den Variablen

```
class LightBulb {

}
```

Klassenvariablen

```
class LightBulb {
  boolean light;
}
```

- Variablendeklarationen immer einrücken!

- Innerhalb einer Klasse können **Variablen** vereinbart werden
- Derartige **Klassenvariablen** werden innerhalb der Klasse wie alle anderen Variablen auch benutzt
- Von außen lässt sich auf Klassenvariablen nur mit Hilfe des Namens der jeweiligen Instanz (oder der Klasse) zugreifen

Attribute

```
class LightBulb {
  public boolean light;
}
```

```
class LightBulb {
  private boolean light;
}
```

- Variablen können Attribute tragen
- Attribute steuern die Zugriffsberechtigungen
- Diese Variable darf jeder benutzen
- Diese Variable darf nur innerhalb der Klasse benutzt werden

- Variablen ohne Attribut sind im gleichen „Paket“ benutzbar

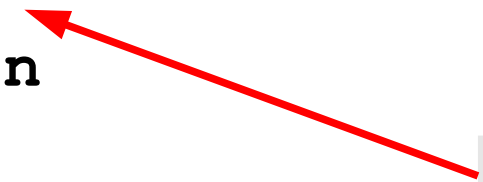
Methoden

- Methoden assoziieren eine Befehlssequenz mit einem Namen
- Befehlssequenz in der Folge über diesen Namen im Programm aufrufbar
- Bevor eine Methode benutzt werden kann, muss sie deklariert werden

Methodendeklaration

- Methoden bestehen aus einem Methodenkopf und einem Methodenrumpf
- Methodenkopf
  - Legt u. a. den Namen der Methode, die Zugriffsrechte und den Rückgabetyp fest

```
void meineMethode() {
   // Anweisungen
   // des
   // Methodenrumpfs
}
```



- Methodenname
  - Eine Methode wird über ihren Namen aufgerufen

Methodenname

- Regeln hinsichtlich der Benennung wie bei Variablen:
  - Die Buchstaben 'a' bis 'z' bzw. 'A' bis 'Z', Ziffern '0' bis '9' und die Sonderzeichen '_' und '$'
  - Groß- und Kleinbuchstaben werden unterschieden
  - Methodenname darf nicht mit einer Ziffer anfangen
  - Nicht erlaubt: Leerzeichen und reservierte Wörter (**`float`** etc.)
  - Instruktive Namen wählen!

Methoden: Zugriffsrechte

- Im gleichen „Paket“ verfügbar:

```
void meineMethode() {
  // Anweisungen
}
```

- Für alle, d. h. auch außerhalb des Pakets, verfügbar:

```
public void meineMethode() {
  // Anweisungen
}
```

- Nur intern im Objekt selbst verfügbar:

```
private void meineMethode() {
  // Anweisungen
}
```

Methodenrumpf

- Zu jeder Methode gehört ein Methodenrumpf
- Dies ist eine in geschweiften Klammern eingeschlossene Folge von Anweisungen
  - Ein **Block**
- Die Anweisungen stehen für die Arbeit, die von der Methode erledigt werden soll
- Die Anweisungen werden ausgeführt, wenn die Methode über ihren Namen aufgerufen wird
- ```
  void meineMethode() {

    // Anweisungen

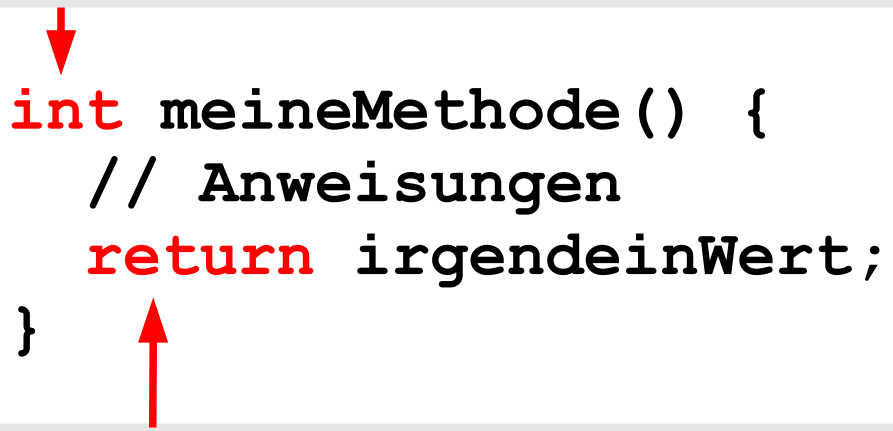
  }
  ```
  ```

Interne Variablen

- Eine Methode kann interne Variablen besitzen
  - Diese werden wie andere Variablen auch deklariert
  - Die Deklaration kann an beliebiger Stelle des Methodenrumpfs erfolgen
  - Interne Variablen sind nur lokal in der Methode und nur ab der Position ihrer Deklaration gültig
- ```
  void meineMethode() {
    Datentyp variablenname;
    // Anweisungen
  }
  ```
  ```

Wert der Methode

- Methoden können einen Wert zurückgeben
- Der Datentyp des Rückgabewerts wird vor dem Methodennamen notiert
- Diese Methode gibt einen Integer-Wert zurück: darum **`int`**
- Hier kann jeder beliebige Typ eingetragen werden!



```
int meineMethode() {
  // Anweisungen
  return irgendeinWert;
}
```

- Wenn ein Wert zurückgegeben werden soll, so muss dies explizit durch Aufruf von **`return`** ausgelöst werden
- **`irgendeinWert`** kann auch für einen Ausdruck stehen

Methode ohne Rückgabewert

- Der spezielle Typ **`void`** ("wertlos") zeigt an, dass eine Methode keinen Wert zurückgibt
  - Die Methode unten gibt also keinen Wert zurück
  - Entsprechend steht im Methodenrumpf **keine `return`-Anweisung**

```
void meineMethode() {
  // Anweisungen
}
```

LightBulb-Klasse

```
class LightBulb {

  boolean light;

  void switchOn() {
    light = true;
  }
  void switchOff() {
    light = false;
  }
  boolean isOn() {
    return light;
  }
}
```

- Die Klasse **`LightBulb`** besitzt drei Methoden
- Nur die Methode **`isOn()`** gibt einen Wert zurück
  - Den der Klassenvariable **`light`**
- **`isOn()`** und die anderen Methoden der Klasse können auf **`light`** zugreifen, weil sie Teil der Klasse **`LightBulb`** sind

Klassenvariable mit Attribut

```
class LightBulb {

  private boolean light;

  void switchOn() {
    light = true;
  }
  void switchOff() {
    light = false;
  }
  boolean isOn() {
    return light;
  }
}
```

- Die Klassenvariable **`light`** ist als **`private`** deklariert, d. h. nur die Methoden der Klasse **`LightBulb`** dürfen auf diese Variable zugreifen
- Dies sind die Methoden **`switchOn()`**, **`switchOff()`** und **`isOn()`**

Initialisierung von Klassenvariablen

```
class LightBulb {

  private boolean light = false;

  void switchOn() {
    light = true;
  }
  void switchOff() {
    light = false;
  }
  boolean isOn() {
    return light;
  }
}
```

- Klassenvariablen können ebenfalls als Teil der Deklaration initialisiert werden

Vollständige LightBulb-Klasse

```
class LightBulb {

  private boolean light = false;

  void switchOn() {
    light = true;
  }
  void switchOff() {
    light = false;
  }
  boolean isOn() {
    return light;
  }
}
```

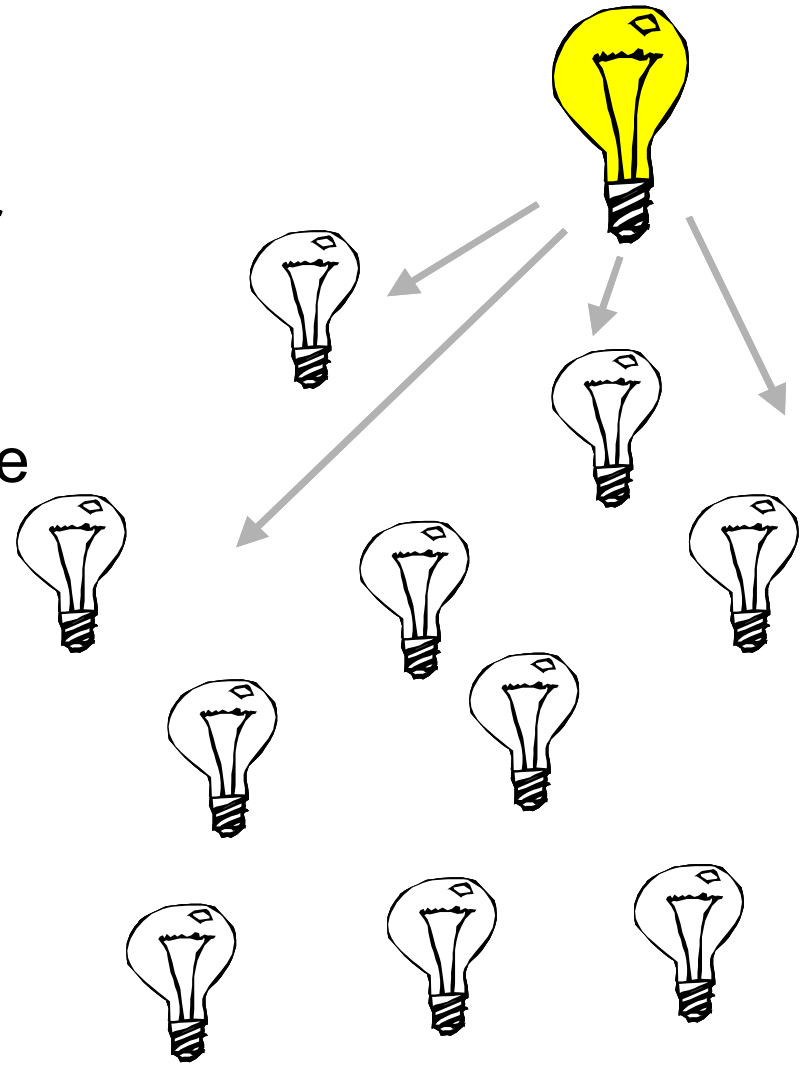
- Die Klasse **`LightBulb`** steht in der Datei **`LightBulb.java`**, die sich im gleichen Verzeichnis wie das Sketch-Hauptprogramm befindet

Wie wird eine Klasse benutzt?

- Sobald eine Klasse ***definiert*** ist, können Objekte dieser Klasse erzeugt werden
- Die Klasse ist ein **abstrakter Datentyp**
  - Bisher kannten wir nur *primitive* Datentypen
- Die Klasse steht für einen Datentyp, den wir selbst definiert haben
- Variablen können als zu diesem Typ gehörend deklariert werden
- Beispiel: `LightBulb lb;`

Instanzen

- Variablen des Typs unserer Klasse können **Instanzen** dieser Klasse aufnehmen
- Diese werden auch als **Objekte** oder **Exemplare** bezeichnet
- Wenn wir die Klasse *instanzieren* erzeugen wir ein Objekt dieser Klasse
- Dies wird in Java mittels des Schlüsselworts **`new`** durchgeführt
- Beispiel:
  ```
  lb = new LightBulb();
  ```
- Die neu erzeugte Instanz wird danach **`lb`** zugewiesen



Deklaration mit Instanzierung

- Deklaration und Instanzierung können auch Teil derselben Anweisung sein
- D. h. an Stelle von

```
LightBulb lb;
lb = new LightBulb();
```

- können wir auch schreiben:

```
LightBulb lb = new LightBulb();
```

  - Die Variable **`lb`** wird hier mit der erzeugten Instanz der Klasse **`LightBulb`** initialisiert

Beispiel: Hauptprogramm mit LightBulb-Objekt

- Inhalt des Sketch-Hauptprogramms:

```
LightBulb lb = new LightBulb();

//lb.switchOn();
lb.switchOff();

boolean lightState = lb.isOn();

if (lightState == true)
   println("Light is on");
else
   println("Light is off");
```

Deklaration der Variable **lb** und Initialisierung mit einer neuen Instanz der Klasse **LightBulb**

- Die Sketch-(PDE-)Datei des Hauptprogramms befindet sich im gleichen Verzeichnis wie die Java-Datei der Klasse

Methodenaufruf

```
LightBulb lb = new LightBulb();

//lb.switchOn();
lb.switchOff();

//etc.
```

- Sobald das Objekt erzeugt wurde, können wir dessen Methoden aufrufen, es sei denn, sie sind `private`
- Dies wird mit dem **Punkt-Operator** durchgeführt
- Dieser verknüpft das Objekt, dessen Methode aufgerufen werden soll, mit der gewünschten Methode

Zugriff auf Variable

```
LightBulb lb = new LightBulb();
lb.light = true;
boolean lightState = lb.isOn();
//etc.
```

- Auch ein direkter Zugriff auf Klassenvariablen ist möglich, wenn diese nicht als `private` deklariert sind
- Zugriff auch hier mit dem **Punkt-Operator**

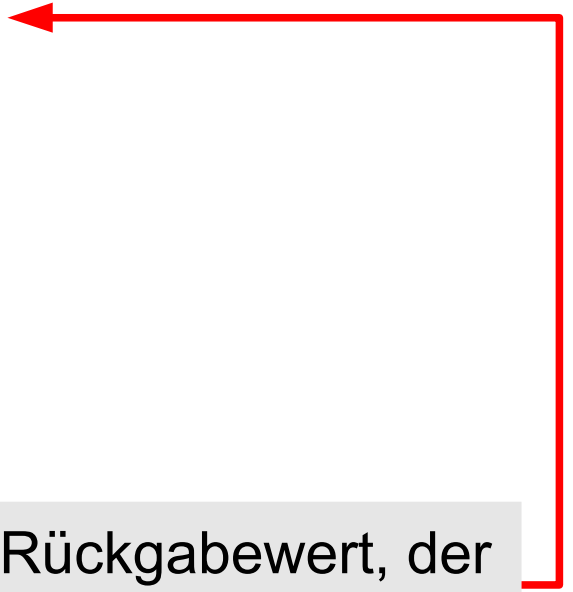
Methode mit Rückgabewert (1)

```
LightBulb lb = new LightBulb();

//lb.switchOn();
lb.switchOff();

boolean lightState = lb.isOn();

if (lightState == true)
  println("Light on");
else
  println("Light off");
```



Die Methode `isOn()` liefert einen Rückgabewert, der einer Variable zugewiesen werden kann

Methode mit Rückgabewert (2)

```
LightBulb lb = new LightBulb();

//lb.switchOn();
lb.switchOff();

boolean lightState = lb.isOn();

if (lightState == true)
  println("Light on");
else
  println("Light off");
```


Der Wert der Variable wird mit dem booleschen Wert **`true`** verglichen, um den Zustand der Glühbirne zu bestimmen

Methode mit Rückgabewert (3)

```
LightBulb lb = new LightBulb();

//lb.switchOn();
lb.switchOff();

boolean lightState = lb.isOn();

if (lightState)
  println("Light on");
else
  println("Light off");
```

Da **`lightState`** eine boolesche Variable ist, können wir ihren Wert aber auch sofort nehmen

Methode mit Rückgabewert (4)

```
LightBulb lb = new LightBulb();

//lb.switchOn();
lb.switchOff();

if (lb.isOn())
  println("Light on");
else
  println("Light off");
```

Da **`isOn()`** einen boolschen Rückgabewert liefert, der nur einmal ausgewertet wird, kann der Methodenaufruf auch gleich für die logische Bedingung in die **`if`**-Anweisung eingesetzt werden

Mehr als ein Objekt erzeugen

```
LightBulb lbLeft  = new LightBulb();
LightBulb lbRight = new LightBulb();

//lbLeft.switchOn();
lbLeft.switchOff();

lbRight.switchOn();
//lbRight.switchOff();

println("Left : " + lbLeft.isOn());
println("Right: " + lbRight.isOn());
println("-------");
```

Methodenparameter

Einführung (1)

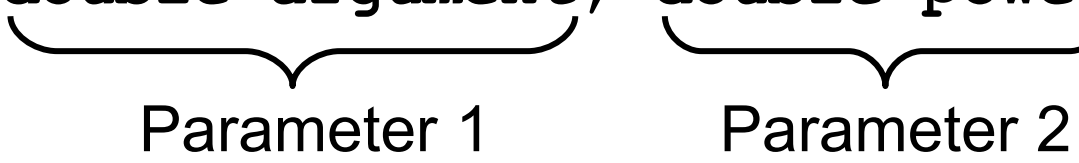
- In der **`LightBulb`**-Klasse führen die Methoden vordefinierte Aufgaben durch, ohne hierfür Eingabedaten zu benötigen
- Beispiel:
  - **`lb.switchOn()`**
  - Schaltet die "Glühbirne" ein
  - Hierfür sind keine Eingabedaten erforderlich

Einführung (2)

- Sehr häufig besteht die Aufgabe einer Methode darin, Eingabedaten zu verarbeiten und ggf. einen Ausgabewert zu liefern
- Beispiel:
  - **`Math.pow(double argument, double power)`**
    - Erwartet Eingabewerte für das Argument und den Exponent
    - Liefert die Potenz

Methodenparameter

- `Math.pow(double argument, double power)`



- Methodenparameter werden in runden Klammern nach dem Methodennamen deklariert
- Deklaration eines Parameters besteht aus:
  - **Parametertyp**
  - **Parametername**

Methodendeklaration mit Parameter

- Syntax:

```
void meineMethode(typ_1 Par_1,...,typ_n Par_n) {
  // Anweisungen
}
```

Parametertyp Parametername

- Parametername
  - Beliebig
  - Regeln hinsichtlich Benennung wie bei Variablen
- Parametertyp
  - Primitiver oder abstrakter Datentyp (Klasse)

Beispiel: Polynomauswertung

- $p = a_0 + a_1 \cdot x + a_2 \cdot x^2 + a_3 \cdot x^3 = \left(\left(a_3 \cdot x + a_2 \right) \cdot x + a_1 \right) \cdot x + a_0$

```
class Polynom { /* in Polynom.java definiert */
  double polynomEvaluation(double pa3, double pa2,
                           double pa1, double pa0,
                           double px) {
    return(((pa3*px + pa2)*px + pa1)*px + pa0);
  }
}
```

- Erwartet Eingabewerte für die Koeffizienten $\mathtt{a_i}$ und das Argument **x**
- Liefert Wert des Polynoms als Rückgabewert

Methodenaufruf (1)

- An Stelle der Parameternamen werden beim Methodenaufruf die aktuellen Werte der Parameter eingesetzt
- Beispiel Polynomauswertung (Hauptprogramm in Sketch-Datei):

```
Polynom poly = new Polynom();

double p = poly.polynomEvaluation(9.0,8.0,7.0,6.0,3.0);

println("p(3.0) = " + p);


// Definition der Klasse in Polynom.java wie zuvor

class Polynom {
  double polynomEvaluation(...) {…}
}
```

Methodenaufruf (2)

- Übergebene Werte werden den Parametern zugewiesen

```
double polynomEvaluation(
  double pa3 <- 9.0
  double pa2 <- 8.0
  double pa1 <- 7.0
  double pa0 <- 6.0
  double px  <- 3.0      )
{
  // Innerhalb der Methode liefern die Parameter jetzt
  // die ihnen übergebenen Werte
  return(((pa3*px + pa2)*px + pa1)*px + pa0);
}
```

Methodenaufruf mit Variablen

- An Stelle der Werte können auch Variablennamen eingesetzt werden

- Sketch-Hauptprogramm:

```
double a3 = 9.0, a2 = 8.0,
       a1 = 7.0, a0 = 6.0, x = 3.0;

Polynom poly = new Polynom();

double p = poly.polynomEvaluation(a3, a2, a1, a0, x);

println("p(" + x + ") = " + p);
```

- Parameter und Variablen können auch die gleichen Namen haben. Sie bezeichnen trotzdem verschiedene Speicherzellen!

Typen

- Die Variablentypen müssen kompatibel sein zu den Parametertypen
  - Die Übergabe einer z. B. **`double`**-Variable an einen **`int`**-Parameter führt zu einem Compiler-Fehler

Parameterübergabe

- Die Variablen werden zuerst ausgewertet und die Variablenwerte den Parametern zugewiesen
- **Wertübergabe** (engl.: *"call by value"*)

```
double polynomEvaluation(
  double pa3 <- 9.0 <- a3 // Variablenwert wird von der
  double pa2 <- 8.0 <- a2 // Speicherzelle der Variable
  double pa1 <- 7.0 <- a1 // in die Speicherzelle des
  double pa0 <- 6.0 <- a0 // Parameters kopiert
  double px  <- 3.0 <- x  )
{
  return(((pa3*px + pa2)*px + pa1)*px + pa0);
}
```

Parametrierte Methode: Vollständiges Beispiel

```
class Polynom { // speichern in Polynom.java

  double polynomEvaluation(double pa3, double pa2,
      double pa1, double pa0, double px) {

    return(((pa3*px + pa2)*px + pa1)*px + pa0);
  }
}

// Sketch-Hauptprogramm:

double a3=9.0, a2=8.0, a1=7.0, a0=6.0, x=3.0;
Polynom poly = new Polynom();
double p = poly.polynomEvaluation(a3, a2, a1, a0, x);
println("p(" + x + ") = " + p);

// Ausgabe: p(3.0) = 342.0
```

Parameterübergabe

- Erfolgt immer als Wertübergabe
- Dies bedeutet:
  - Zuweisungen an Parameter innerhalb einer Methode möglich
  - Zuweisungen innerhalb der Methode gültig, aber **nach außen nicht sichtbar**

Zuweisungen an Parameter (1)

```
double polynomEvaluation(double pa3, double pa2,
                         double pa1, double pa0,
                         double px) {
  pa3 = 90.0; // Übergebene Parameterwerte werden in
  pa2 = 80.0; // der Methode durch neue Werte
  pa1 = 70.0; // überschrieben. Dies ist möglich, da
  pa0 = 60.0; // Parameter wie normale Variablen benutzt
  px  = 30.0; // werden können.

  // Innerhalb des Ausdrucks kommen dann die neuen
  // Parameterwerte zum Tragen

  return(((pa3*px + pa2)*px + pa1)*px + pa0);

}

// Die neuen Parameterwerte werden aber nicht nach
// außen durchgereicht!
```

Zuweisungen an Parameter (2)

```
// Sketch-Hauptprogramm:
double a3 = 9.0, a2 = 8.0, a1 = 7.0, a0 = 6.0, x = 3.0;
println("a3: " + a3 + ", a2: " + a2 +
        ", a1: " + a1 + ", a0: " + a0 + ", x: " + x);
Polynom poly = new Polynom();
double p = poly.polynomEvaluation(a3, a2, a1, a0, x);
println("a3: " + a3 + ", a2: " + a2 +
        ", a1: " + a1 + ", a0: " + a0 + ", x: " + x);
println("p(" + x + ") = " + p);
```

Ausgabe:

```
a3: 9.0, a2: 8.0, a1: 7.0, a0: 6.0, x: 3.0
a3: 9.0, a2: 8.0, a1: 7.0, a0: 6.0, x: 3.0
p(3.0) = 2504160.0
```

Parameter und Ausdrücke (1)

- An Stelle eines Wertes oder einer Variable kann einem Parameter auch ein Ausdruck zugewiesen werden:

```
// Sketch-Hauptprogramm:

double a3 = 9.0, a2 = 8.0, a1 = 7.0, a0 = 6.0;
double x = 3.0, p;

Polynom poly = new Polynom();
p = poly.polynomEvaluation(a3*2, (10-a2)+3,
                           a1*a0, a0, x+3);
println("p(" + x + ") = " + p);

// Ausgabe: p(3.0) = 4326.0
```

- Zuerst wird der Ausdruck ausgewertet und dann dessen Wert dem Parameter zugewiesen

Parameter und Ausdrücke (2)

- Die Ausdrücke werden ausgewertet und deren Ergebnisse den Parametern zugewiesen:

```
double polynomEvaluation(
    double pa3 <- 18.0  <- a3*2       <- a3
    double pa2 <- 5.0   <- (10-a2)+3  <- a2
    double pa1 <- 42.0  <- a1*a0      <- a0, a1
    double pa0 <- 6.0   <- a0
    double px  <- 6.0   <- x+3 <- x)
{
    return(((pa3*px + pa2)*px + pa1)*px + pa0);
}
```

Rückgabewert

- Der von der Methode zurückgegebene Wert kann unmittelbar in Ausdrücken eingesetzt werden
  - D. h. der Methodenaufruf kann Teil eines Ausdrucks sein:

```
// Sketch-Hauptprogramm:
double a3 = 9.0, a2 = 8.0, a1 = 7.0, a0 = 6.0;
double x = 3.0;

Polynom poly = new Polynom();

println("p(" + x + ") = " +
    poly.polynomEvaluation(a3, a2, a1, a0, x));
```

Blöcke und Methoden

- Der Rumpf einer Methode definiert einen Block mit eigenem, **lokalem** Gültigkeitsbereich für die **internen** Variablen und Unterprogrammparameter
- Beispiel: $p = a_0 + a_1 \cdot x + a_2 \cdot x^2$

Parameter

```
double polynomEvaluation2(double pa2, double pa1,
                          double pa0, double px) {

  double res, xsquare;

  xsquare = px*px;
  res = pa0+pa1*px+pa2*xsquare;

  return res;
}
```

Interne Variablen

Methoden überladen

Methoden überladen

- Von einer Methode können **unterschiedliche Varianten gleichen Namens** konstruiert werden (engl.: *overloading*)
- Diese werden über ihre Parameterliste identifiziert
- Beispiel:
  - Polynomauswertung für unterschiedliche Polynome
  - Polynom 1:
    - $p = a_0 + a_1 \cdot x + a_2 \cdot x^2 + a_3 \cdot x^3 = \left(\left(a_3 \cdot x + a_2 \right) \cdot x + a_1 \right) \cdot x + a_0$
  - Polynom 2:
    - $p = a_0 + a_1 \cdot x + a_2 \cdot x^2 = \left(a_2 \cdot x + a_1 \right) \cdot x + a_0$
- Inhaltlich ähnliche Abläufe werden so unter einem gemeinsamen Namen verfügbar

Überladene Methode

- Beide Polynome können durch Methoden gleichen Namens implementiert werden
- Der Methodenname legt zusammen mit den Parametern die *Signatur* der Methode fest
  - Der Typ des Rückgabewerts gehört nicht zur Signatur!

```
double polynomEvaluation(double pa3, double pa2,
                         double pa1, double pa0,
                         double px) {
  return(((pa3*px + pa2)*px + pa1)*px + pa0);
}

double polynomEvaluation(double pa2, double pa1,
                         double pa0, double px) {
  return((pa2*px + pa1)*px + pa0);
}
```

Überladene Methode: Aufruf in der Sketch

```
double a3 = 9.0, a2 = 8.0, a1 = 7.0, a0 = 6.0, x = 3.0;

Polynom poly = new Polynom();

// Auswahl der passenden Methode über die Parameterliste

println("p3(" + x + ") = " +
    poly.polynomEvaluation(a3, a2, a1, a0, x));


println("p2(" + x + ") = " +
    poly.polynomEvaluation(a2, a1, a0, x));


// Ausgabe:
// p3(3.0) = 342.0
// p2(3.0) = 99.0
```

`static` Attribut

`static`-Attribut

- `static` als Teil einer Methodendeklaration
  - Beispiel:

    ```
    static double pow(double a, double b)
    ```

  - Diese Methode gehört zur Klasse, aber NICHT zum Objekt bzw. zur Klasseninstanz
  - Für die Benutzung ist es nicht erforderlich, ein Objekt der Klasse zu erzeugen!
  - Die Methode kann über den Klassennamen benutzt werden
  - Sinnvoll, wenn eine Klasse primär Bibliothekscharakter für Methoden hat
  - Beispiel:

    ```
    double xSquared = Math.pow(x,2);
    ```

`static`-Attribut

- **`static`** als Teil der Deklaration einer Klassenvariablen
  - Beispiel: **`static double my_pi = 3.1415;`**
    - Diese Variable gehört zur Klasse, aber NICHT zum Objekt bzw. zur Klasseninstanz
    - Für die Benutzung ist es nicht erforderlich, ein Objekt der Klasse zu erzeugen!
    - Die Variable kann über den Klassennamen benutzt werden
  - Weitere Beispiele:

```
// Zu einer Klasse gehörende Konstante:
static final double MY_PI = 3.1415;
// Vordefinierte Konstante von Java für Pi:
double circumference = 2*Math.PI*r;
```

Bsp: Klasse `Polynom` mit statischen Methoden

```
class Polynom {

    static double polynomEvaluation(
        double pa3, double pa2, double pa1,
        double pa0, double px) {
            return(
              ((pa3*px + pa2)*px + pa1)*px + pa0);
    }
    static double polynomEvaluation(
        double pa2, double pa1,
        double pa0, double px) {
            return((pa2*px + pa1)*px + pa0);
    }
}
```

Aufruf der statischen Methoden in der Sketch

```
double a3 = 9.0, a2 = 8.0, a1 = 7.0, a0 = 6.0, x = 3.0;

// Polynom poly = new Polynom(); nicht mehr erforderlich

// Auswahl der passenden Methode über die Parameterliste
// und Aufruf mit Hilfe des Klassennamens

println("p3(" + x + ") = " +
    Polynom.polynomEvaluation(a3, a2, a1, a0, x));

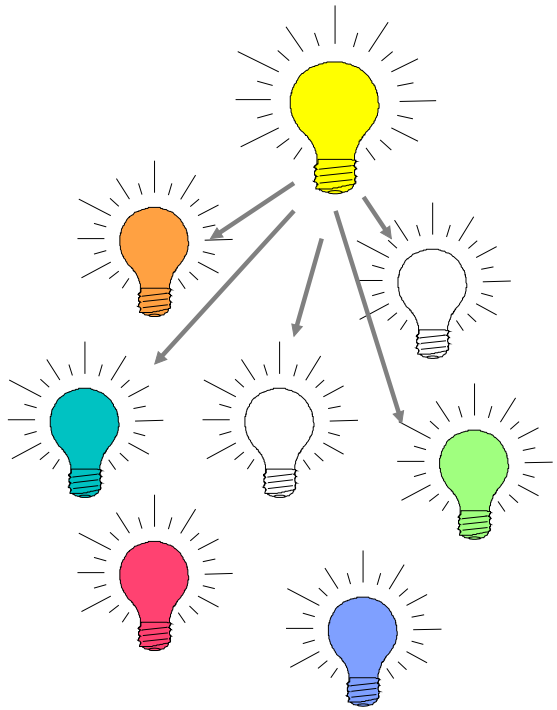

println("p2(" + x + ") = " +
    Polynom.polynomEvaluation(a2, a1, a0, x));


// Ausgabe:
// p3(3.0) = 342.0
// p2(3.0) = 99.0
```

Konstruktoren

Konstruktoren

- Objekte sollen sich, analog zu Variablen, mit unterschiedlichen Startwerten initialisieren lassen
- Dies wird durch Konstruktoren (engl.: *constructors*) erreicht
- Konstruktoren erlauben es, bei der "Produktion" von Objekten einer Klasse zwischen verschiedenen Varianten auszuwählen



Konstruktor

```
class LightBulb {

  private boolean light=false;
  private String colour;

  LightBulb() {
    colour = "white";
  }
  void switchOn() {
    light = false;
  }
  // etc. -> wie zuvor

  String getColour() {
    return (colour);
  }
}
```

- Der **Konstruktor** ist eine spezielle Methode der Klasse
- Der Name des Konstruktors ist gleich dem Klassennamen
- Der Konstruktor wird automatisch aufgerufen, wenn ein Objekt der Klasse erzeugt wird
- Mittels Konstruktoren lassen sich Klassenvariablen geeignet initialisieren
- Dieser Konstruktor erzeugt standardmäßig weiße Glühbirnen

Konstruktor mit Parameter

```
class LightBulb {

  private boolean light=false;
  private String colour;

  LightBulb(String pColour) {
    colour = pColour;
  }
  void switchOn() {
    light = false;
  }
  // etc. -> wie zuvor

  String getColour() {
    return (colour);
  }
}
```

- Um unterschiedliche Varianten herstellen zu können, muss der Konstruktor mit **Parametern** versehen werden
- Konstruktoren mit Parametern erlauben es, den Anfangswert der Klassenvariablen bei der Instanzierung festzulegen
- Hier: Glühbirnen unterschiedlicher Farbe

Parameternamen

```
class LightBulb {

  private boolean light=false;
  private String colour;

  LightBulb(String pColour) {
    colour = pColour;
  }
  void switchOn() {
    light = false;
  }
  // etc. -> wie zuvor

  String getColour() {
    return (colour);
  }
}
```

- Um die Konstruktor-parameter von den Klassenvariablen unterscheiden zu können, müssen die Parameter eigene Namen besitzen
- Häufig ist es mühsam, sich neue Namen zu überlegen
- Ausweg: Vorangestellter Buchstabe, z. B. "**p**"
- Besser: **this** Referenz benutzen

this

```
class LightBulb {

  private boolean light=false;
  private String colour;

  LightBulb(String colour) {
    this.colour = colour;
  }

  void switchOn() {
    light = false;
  }
  // etc. -> wie zuvor

  String getColour() {
    return (colour);
  }
}
```

- **this** liefert eine Referenz auf das Objekt selbst
- **this** ermöglicht es, zwischen gleichnamigen Konstruktorparametern und Klassenvariablen zu unterscheiden
- Die Konstruktor-parameter und die korrespondierenden Klassenvariablen können also die gleichen Namen tragen

Konstruktor-Parameter: Hauptprogramm

```
LightBulb lb = new LightBulb("red"); // red light
println("Light has colour " + lb.getColour());
```

Überladener Konstruktor

```
class LightBulb {

  private boolean light=false;
  private String colour;

  LightBulb() {
    colour = "white";
  }

  LightBulb(String colour) {
    this.colour = colour;
  }
  void switchOn() {
    light = false;
  }
  // etc. -> wie zuvor
}
```

- Auch Konstruktoren können überladen werden
- Bei der Erzeugung eines Objekts der Klasse wird dann der jeweils passende Konstruktor an Hand der Parameterliste ausgewählt

Überladener Konstruktor: Hauptprogramm

```
// Passender Konstruktor wird an Hand der Anzahl und
// des Typs der übergebenen Parameterwerte ausgewählt

LightBulb lb1 = new LightBulb(); // Erzeugt weiße Lampe
LightBulb lb2 = new LightBulb("red"); // Erz. rote Lampe

println("lb1 has colour " + lb1.getColour());
println("lb2 has colour " + lb2.getColour());
```

- Deklaration und Instanzierung zweier Objekte vom Typ **`LightBulb`** mit unterschiedlichen Eigenschaften

Literatur

[1] Carter, Paul A.: *PC Assembly Language*, 2002, siehe unter "Literatur" auf der DV-Homepage

[2] DIN 44300

[3] Eckel, Bruce: *Thinking in Java*, Prentice Hall, 2003

[4] Gallardo, R., et al.: *The Java Tutorial*: A Short Course on the Basics, https://docs.oracle.com/javase/tutorial/, 17.3.2020

[5] Gumm, H.-P.; Sommer, M.: *Einführung in die Informatik*, Oldenbourg, 2002

[6] Flik, Th.: *Mikroprozessortechnik*. 6. Aufl. Berlin: Springer, 2001

[7] Hyde, Randall: *The Art of Assembly Language Programmin*g, http://webster.cs.ucr.edu