Algorithmen
und
Programmierung I

WS 2025 / 2026

Organisatorisches, Inhalt und Literatur

Prof. Dr. Frank Victor
TH Köln

Organisatorisches, Inhalt und Literatur

Termine

06.10.2025 – 15.12.2025 **Vorlesung**: Montags, 11.00 – 13.45 Uhr in Raum 0401
Übung: Montags, 16.00 – 16.45 Uhr in Raum 0401

Weihnachtsferien

12.01.2026 – 26.01.2026 **Vorlesung**: Montags, 11.00 – 13.45 Uhr in Raum 0401
Übung: Montags, 16.00 – 16.45 Uhr in Raum 0401

In den Projektwochen 24.11.2025 – 28.11.2025, 22.12.2025 – 23.12.2025 und 05.01.2026 – 09.01.2026 ist keine Vorlesung und keine Übung.

Praktika Termine nach „Staffelplan“ und **Praktikumverwaltungstool**.

https://www.gm.fh-koeln.de/advlabor/

Sprechstunde

Prof. Dr. Frank Victor
Montags, 14.30 – 15.30 in Raum 2228.

Bitte per Email eine Woche vorher anmelden!
frank.victor@th-koeln.de

Script und Material zur Vorlesung

https://ilu.th-koeln.de/
→ F10 - Informatik und Ingenieurwissenschaften Bachelor Studiengänge
→ Dort Ihren Studiengang suchen
→ Dort „Algorithmen und Programmierung I WiSe 25/26 (Prof. Dr. Frank Victor)“ auswählen und Material downloaden

Wer hilft bei Fragen?

M. Sc. Anja Bertels **anja.bertels@th-koeln.de**
B. Sc. Sascha Schewe **sascha.schewe@th-koeln.de**

Fragen zum Praktikum: ap-praktikum@gm.fh-koeln.de.

Inhalt

Teil A: Theoretische Grundlagen

1 Algorithmen und Datenstrukturen

2 Programmiersprachen
- Grundlegende Begriffe
- Klassifizierung von Programmiersprachen

Inhalt

Teil B: Prozedurale Programmierung in der Sprache C

1 Entwicklungsgeschichte und Charakteristika von C

2 Erste Schritte in C
    Der Aufbau eines C-Programms, Ein- und Ausgabe, Kommentare

3 Datentypen, Variablen und Konstanten in C
    Fundamentale Datentypen, Variablen und Konstanten, Typkonvertierung

4 Ausdrücke und Operatoren in C
    Grundlegende Begriffe, Arithmetische, Relationale Operatoren, usw.

5 Anweisungen in C
    if, switch, while, do-while, for, break und continue

6 Die zusammengesetzten Datentypen in C
    Arrays, Strukturen, Enumerations

7 Funktionen in C
    Funktionsdefinition und Aufruf, Funktionsdeklaration und getrennte Übersetzung

8 Zeiger
    Address-of-Operator, Dereferenzierungsoperator, Adressen als Funktionsparameter

9 Rekursion

Inhalt

Teil C: Objektorientierte Programmierung in der Sprache Java

1 Motivation: Herleitung von Klassen
2 Objektorientierung
Grundlegende Begriffe, Entwicklungsgeschichte und Charakteristika von Java
3 Erste Schritte in Java
4 Grundlagen von Java
Zeichensatz und Namenskonventionen, Kommentare und Javadoc
5 Datentypen, Variablen und Konstanten in Java
Datentypen in Java, Variablen und Konstanten, Casts in Java
6 Anweisungen in Java
7 Ein- und Ausgabe in Java
Eingabe über die Tastatur, Formatierte Ausgabe mit printf
8 Klassen und Objekte in Java
Die Grundstruktur einer Klasse, Konstruktoren, this, static, enum
9 Arrays und Strings in Java

Inhalt

Teil D: Komplexität von Algorithmen

1 Der Begriff Komplexität
2 O-Notation
3 Komplexitätsklassen und Algorithmen
4 Beispiele zum Abschätzen der Zeitkomplexität
5 Aufgaben zur Zeitkomplexität
6 Animation von Sortieralgorithmen
7 Weiterführende Literatur

Literatur

1. Theoretische Grundlagen: Algorithmen und Datenstrukturen

Drösser, Ch., Total berechenbar? Wenn Algorithmen für uns entscheiden, Carl Hanser, 2016

Harel, D., Feldman, Y. , Algorithmik – Die Kunst des Rechnens, Springer, 2009

Ottman, Th., Widmayer, P., Algorithmen und Datenstrukturen, Springer Vieweg, 2017

Saake, G., Sattler, K.-U., Algorithmen und Datenstrukturen: Eine Einführung mit Java, dpunkt, 2013

Schneider, U., Taschenbuch der Informatik, Carl Hanser, 2012

Sedgewick, R., Wayne, K., Algorithmen: Algorithmen und Datenstrukturen, Pearson Studium, 2014

Literatur

2. Programmieren in C

Goll, J., Dausmann, M., C als erste Programmiersprache: Mit den Konzepten von C11, Springer Vieweg, 2014

Griffiths, D., C von Kopf bis Fuß, O'Reilly, 2012

Kernighan, B.W., Ritchie, D. M., The C Programming Language (ANSI C version), Markt+Technik, 2000

Literatur

3. Programmieren in Java

Eckel, B., Thinking in Java, Markt + Technik, 2003

Goll, J., Heinisch, C., Java als erste Programmiersprache: Grundkurs für Hochschulen, Springer Vieweg, 2016

Inden, M., Der Weg zum Java-Profi: Konzepte und Techniken für die professionelle Java-Entwicklung. Aktuell zu Java 9., dpunkt.verlag, 2017

Jobst, F., Programmieren in Java, Carl Hanser, 2014

Louis, D. , Müller, P., Java: Eine Einführung in die Programmierung, Carl Hanser, 2014

Küneth, Th., Einstieg in Eclipse: Die Werkzeuge für Java-Entwickler, Galileo Computing, 2014

Schiedermeier, R., Programmieren mit Java, Pearson Studium, 2010

Ullenboom, C., Java ist auch eine Insel: Programmieren lernen mit dem Standardwerk für Java-Entwickler, Rheinwerk Computing, 2017

Literatur

4. Komplexität von Algorithmen

Hopcroft J. E.,Motwani R. et al., Einführung in Automatentheorie, Formale Sprachen und Berechenbarkeit, Pearson Studium – IT, 1. März 2011

Schöning, U. Meier, A. et al., Komplexität von Algorithmen: Mathematik für Anwendungen Band 4, Lehmanns Media, 6. Juli 2020

Algorithmen
und
Programmierung I

WS 2025 / 2026

Teil A: Theoretische Grundlagen

Prof. Dr. Frank Victor
TH Köln

Inhalt

Teil A: Theoretische Grundlagen

1 Algorithmen und Datenstrukturen

2 Programmiersprachen
Grundlegende Begriffe
Klassifizierung von Programmiersprachen

1. Algorithmen und Datenstrukturen

Algorithmus

Der Begriff **Algorithmus** bezeichnet eine **Vorschrift zur Lösung eines Problems**, die für eine Realisierung in Form eines **Programms** auf einem **Computer** geeignet ist.

Anforderungen an Algorithmen

- Korrektheit
- Funktionalität
- Testbarkeit
- Komplexität

Beispiele für Algorithmen

- Das Rechnen mit Zahlen
- Das 4-Damen-Problem
- Sortieren

1. Algorithmen und Datenstrukturen

Das Verfahren alleine genügt nicht. Es kommt auch auf die Struktur der Daten an!

Beispiele für Datenstrukturen

- Zahlen
- Brüche
- Schachbrett
- Folge von Zahlen

N. Wirth (der Vater der strukturierten Programmierung und Erfinder der Programmiersprache PASCAL) postuliert für die **klassische Programmierung:**

Programm = Algorithmus + Datenstruktur

Anforderung an Programme

Funktionalität	Sicherheit	Korrektheit
Portabilität	Interoperabilität	Effizienz
Kosten	Testbarkeit	leichte Änderbarkeit und Erweiterbarkeit
Wiederverwendbarkeit	Dokumentation	Robustheit

1. Algorithmen und Datenstrukturen

Programmsysteme werden von **großen Gruppen** entwickelt:
- **Schnittstellen** zwischen Programm-Modulen sind notwendig
- Bestimmte **Programmierparadigmen** sind zu formulieren und einzuhalten

Programmierparadigmen
- **prozedurale (strukturierte)** Programmierung
- **objektorientierte** Programmierung
- andere (z.B. aus der Künstlichen Intelligenz)

Probleme des prozeduralen Ansatzes **„Software-Krise der 70er Jahre"**
- nur für kleinere Probleme gut geeignet
- bei Entwurf und Pflege großer, komplexer Programmsysteme
- bei der **Wiederverwendbarkeit** von Softwareteilen:
  - Änderungen sind schwierig durchzuführen
  - Ineinanderfließen von Datenstrukturen und Algorithmen
  - Neuentwicklungen sind oft kostengünstiger als Anpassungen.

Beispiel
Warteschlange

1. Algorithmen und Datenstrukturen

Objektorientierter Ansatz

- „Unsere Welt besteht aus Objekten“
- Operationen und Daten gehören zusammen

Ziel der Vorlesungen "Algorithmen und Programmieren I und II"

Prozedurale Programmierung in C
&
Objektorientierte Programmierung in Java

Inhalt des Praktikums
Algorithmen und deren Implementierung

2. Programmiersprachen

Grundlegende Begriffe

Programmiersprache
Eine **Programmiersprache** ist eine formale Sprache zur Notation von Computerprogrammen.

Programm
Ein **Programm** ist eine Abfolge von Daten und Befehlen an einem Prozessor, um diese Daten in andere Daten umzuwandeln. Es wird in einer formal definierten Sprache verfasst, der Programmiersprache.

A program is a specification of a computation. A programming language is a notation for writing programms.

Syntax
Die **Syntax** einer Programmiersprache beschreibt die Menge der erlaubten Zeichenketten für Programme in dieser Sprache. Ist die Syntax eines Programms nicht korrekt, so meldet der Compiler **Syntaxfehler** (**syntax error**).

Semantik
Die **Semantik** einer Programmiersprache definiert die Bedeutung der einzelnen Sprachkonstrukte.

2. Programmiersprachen

Beispiel: Beispielprogramm in der Sprache C

Dateiname: **hello.c**

```
#include <stdio.h>

int main() {
        printf("Ich heiße Frank Victor.\n");
        return 0;
}
```

Übersetzen und Ausführen des Programms:

```
advm1> cc hello.c
advm1> a.out
Ich heiße Frank Victor.
advm1>
```

→ Das Programm ist syntaktisch korrekt!

2. Programmiersprachen

Beispiel: Beispielprogramm in der Sprache C

Dateiname: **hello.c**

```
#include <stdio.h>

int main() {
        printf("Ich heiße Frank Victor.\n")
        return 0;
}
```

Übersetzen und Ausführen des Programms:

```
advm1> cc hello.c
"hello.c", line 5.4: Syntax error: possible missing ';' or ','?
```

→ Das Programm enthält einen Syntaxfehler!

2. Programmiersprachen

Beispiel: Beispielprogramm in der Sprache C

Dateiname: **`hello.c`**

```
#include <stdio.h>

int main() {
        printf("Ich heiße Frank Victor.\n;
        return 0;
}
```

Übersetzen und Ausführen des Programms:

```
advm1> cc hello.c
"hello.c", line 4.38: String literal must be ended before the end of line.
"hello.c", line 5.4: Syntax error: possible missing ')'?
```

→ Das Programm enthält 2 Syntaxfehler!

2. Programmiersprachen

Beispiel: Semantikbeschreibung für die Anweisung **`printf`**

`printf - Print Formatted [C Programming Language]`

`The standard function in the C programming language library for printing formatted output. The first argument is a format string which may contain ordinary characters which are just printed and conversion specifications - sequences beginning with '%' such as %6d which describe how the other arguments should be printed` …

- Die **Semantik** (**Bedeutung**) von **`printf`** wird textuell beschrieben.
- Es wird gesagt, was **`printf`** in einem Programm bewirkt und wie es funktioniert.

2. Programmiersprachen

Klassifizierung von Programmiersprachen

- **Maschinensprachen** beschreiben einen Befehlssatz, der durch die Hardware eines Prozessors festgelegt ist.
- **Assemblersprachen** unterscheiden sich von Maschinensprachen dadurch, dass die Befehle durch Befehlswörter ausgedrückt und dass für die Befehlsparameter symbolische Bezeichner verwendet werden.
- **Höhere Programmiersprachen** beschreiben algorithmische Verfahren in einer rechnerunabhängigen problemorientierten Form.
- **Anwendungsorientierte Sprachen** enthalten höhere Sprachkonstrukte für einen eingeschränkten Anwendungsbereich (z.B. Datenbankensprachen).
- **Dokumentbeschreibungssprachen** beschreiben die logische Struktur von Textdokumenten (z.B.TeX, SGML). Die Sprachen HTML und XML erlauben die Verknüpfung von Dokumenten über das Internet.

Quelle: Schneider, U., Taschenbuch der Informatik, Carl Hanser, 2012, Kapitel 7 „Programmiersprachen“ von Frank Victor

2. Programmiersprachen

Klassifizierung von Programmiersprachen – die Wurzeln

Sprache	Jahr	Typ	Anwendung	Vorläufer
Fortran	1956	prozedural	technisch-wiss. Anwendungen	
Cobol	1960	prozedural	Betriebswirtschaft	
Lisp	1960	funktional	Künstliche Intelligenz	
Algol 60	1960	prozedural	universell	
Simula 67	1967	prozedural, objektorientiert	Simulation	Algol 60
PL/1	1968	prozedural	universell	Algol 60, Cobol
Pascal	1970	prozedural	Ausbildung, universell	Algol 60
Prolog	1970	Logik	Künstliche Intelligenz	
C	1970	prozedural	Systemprogrammierung	Algol-60
Smalltalk	1980	objektorientiert	graphische Oberflächen	Simula-67
Pearl	1981	prozedural	Echtzeitprogrammierung	PL/1
ADA	1980	prozedural, objektorientiert	universell	Pascal
C++	1986	prozedural, objektorientiert	universell	C, Simula 67
Java	1992	objektorientiert	Internet, universell	C++, Smalltalk

Quelle: Schneider, U., Taschenbuch der Informatik, Carl Hanser, 2012, Kapitel 7 „Programmiersprachen" von Frank Victor

2. Programmiersprachen

Klassifizierung von Programmiersprachen

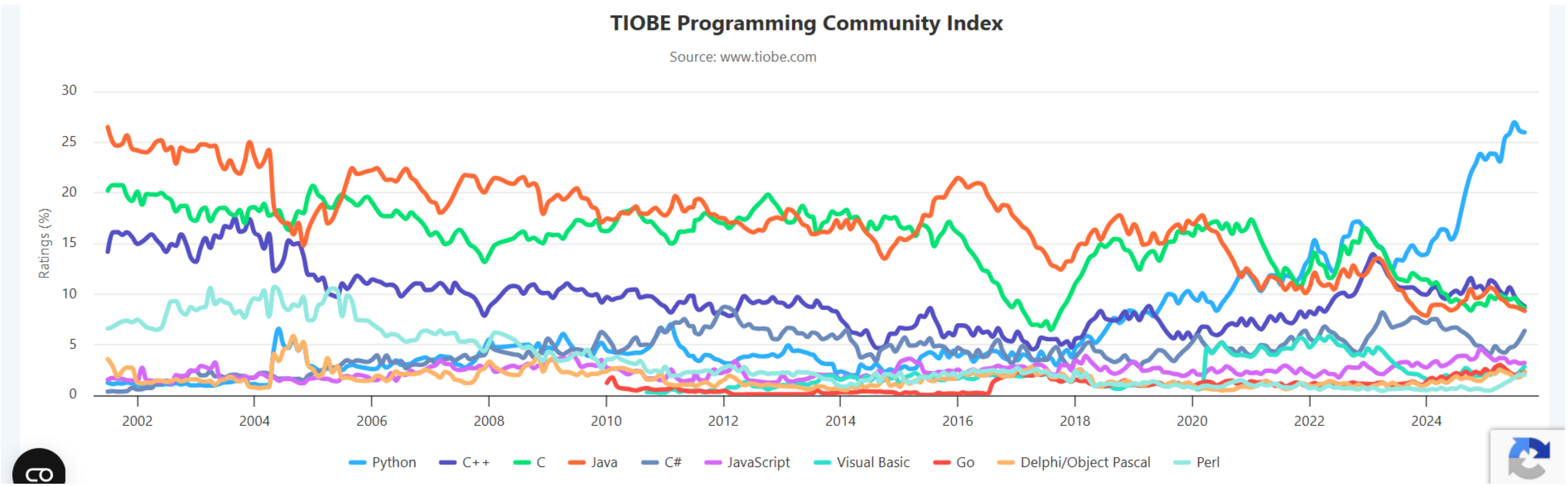
TIOBE Programming Community Index (Anfragen nach Programmiersprachen in Suchmaschinen)

Sep 2025	Sep 2024	Change	Programming Language	Ratings	Change
1	1		Python	25.98%	+5.81%
2	2		C++	8.80%	-1.94%
3	4	^	C	8.65%	-0.24%
4	3	v	Java	8.35%	-1.09%
5	5		C#	6.38%	+0.30%
6	6		JavaScript	3.22%	-0.70%
7	7		Visual Basic	2.84%	+0.14%
8	8		Go	2.32%	-0.03%

Quelle: https://www.tiobe.com/tiobe-index/, 17.09.2025

2. Programmiersprachen

Klassifizierung von Programmiersprachen



Quelle: https://www.tiobe.com/tiobe-index/, 17.09.2025

Theoretische Grundlagen

Literatur zur Vertiefung

Drösser, Ch., Total berechenbar? Wenn Algorithmen für uns entscheiden, Carl Hanser, 2016

Harel, D., Feldman, Y. , Algorithmik – Die Kunst des Rechnens, Springer, 2009

Ottman, Th., Widmayer, P., Algorithmen und Datenstrukturen, Springer Vieweg, 2017

Saake, G., Sattler, K.-U., Algorithmen und Datenstrukturen: Eine Einführung mit Java, dpunkt, 2013

Schneider, U., Taschenbuch der Informatik, Carl Hanser, 2012, Kapitel: „Programmiersprachen" (Frank Victor)

Sedgewick, R., Wayne, K., Algorithmen: Algorithmen und Datenstrukturen, Pearson Studium, 2014

Algorithmen
und
Programmierung I

WS 2025 / 2026

Teil B: Prozedurale Programmierung in der Sprache C

Prof. Dr. Frank Victor
TH Köln

Inhalt

Teil B: Prozedurale Programmierung in der Sprache C

1 Entwicklungsgeschichte und Charakteristika von C
2 Erste Schritte in C
Der Aufbau eines C-Programms, Ein- und Ausgabe, Kommentare
3 Datentypen, Variablen und Konstanten in C
Fundamentale Datentypen, Variablen und Konstanten, Typkonvertierung
4 Ausdrücke und Operatoren in C
Grundlegende Begriffe, Arithmetische, Relationale Operatoren, usw.
5 Anweisungen in C
if, switch, while, do-while, for, break und continue
6 Die zusammengesetzten Datentypen in C
Arrays, Strukturen, Enumerations
7 Funktionen in C
Funktionsdefinition und Aufruf, Funktionsdeklaration und getrennte Übersetzung
8 Zeiger
Address-of-Operator, Dereferenzierungsoperator, Adressen als Funktionsparameter
9 Rekursion

1. Entwicklungsgeschichte und Charakteristika von C

Entwicklungsgeschichte von C

- 1969
  Ken Thomson (Bell Laboratories der AT & T)
  Erste Version von **UNIX** in Assembler, die jedoch nicht auf andere Maschinen portabel ist.
  → Portierung des Betriebssystems erreichen!
- 1970
  Sprache **B** als Weiterentwicklung der Sprache **BCPL** auf einer PDP/7 (nur Maschinenworte)
- 1974
  Dennis M. Ritchie entwickelt die Sprache B zur Sprache **C** weiter.
- Seit 1974
  - schnelle Verbreitung von C (Basis für UNIX und TCP/IP)
  - universelle, betriebssystemunabhängige Sprache
- 1988: Normierung durch das ANSI-Komitee: ANSI-C
- 1999: ANSI-C99, ISO/IEC 9899:1999
- C11: ISO/IEC 9899:2011
- C18: ISO/IEC 9899:2018

1. Entwicklungsgeschichte und Charakteristika von C

Charakteristika von C

- Gute Ausnutzung der Hardware
- Hohe Portabilität
- Compiler erzeugen effizienten Code
- Geeignet als Hochsprache mit vielen komfortablen Konstrukten und als systemnahe Sprache
- Heute sehr häufig: Verknüpfung von Java und C in Programmen

Exkurs: **C++**

- **Objektorientierte** Weiterentwicklung von C (**hybride** Sprache)
- 1983 entwickelt durch Bjarne Stroustrup (AT&T):

The design goal was to create a better C, a C with classes.

- Ziel: **Kooperative** Entwicklung großer Programme (→ **Schnittstellen** zwischen Programmteilen)
- Der Name C++ kommt vom Inkrementierungsoperator ++ in C:
  x++ <==> x = x + 1

2. Erste Schritte in C

Der Aufbau eines C-Programms

Lernziel: Entwicklung von einfachen **sequenziellen** Programmen

Ein C-Programm besteht aus einer oder mehreren Funktionen (Funktionsblöcken).

Funktion

- Programmsegment, das einen eigenen Namen hat und unter diesem Namen (beliebig oft) aufgerufen werden kann.
- Entspricht einer *subroutine* (FORTRAN), einer *function* (BASIC) oder einer *procedure* (PASCAL)

2. Erste Schritte in C

Der Aufbau eines C-Programms

Eine Funktionsdefinition in C besteht aus den folgenden Teilen

[(a) dem Datentyp des Rückgabewerts]
(b) dem Funktionsnamen
(c) der Argumentliste
(d) dem Funktionsblock
[(e) dem Rückgabewert (return value)]

Der Funktionsblock wird durch geschweifte Klammern { } eingeschlossen und stellt einen **Statementblock** dar. In C ist ein Statementblock eine Gruppierung von **Statements**.

Ein Statement ist entweder
(a) eine **ausführbare Anweisung**, die in entsprechende Maschinenbefehle übersetzt wird,
(b) eine **Definition**, die Speicherplatz für Objekte reserviert oder
(c) eine **Deklaration**, die einen Bezug zu einer Definition darstellt.

Jedes Statement muss mit einem Semikolon ; abgeschlossen werden. Alle ausführbaren Anweisungen müssen im Funktionsblock stehen.

2. Erste Schritte in C

Der Aufbau eines C-Programms

Beispiel: Das kürzeste ablauffähige C-Programm

```
int main() {
    return 0;
}
```

Funktionsblock Anfang = {

Funktionsblock Ende = }

Ende des Statements (hier: ausführbare Anweisung)

Returnwert, hier 0, „alles OK“ wird an das Betriebssystem zurückgegeben

Funktionsname, hier „main“ für Hautprogramm

Rückgabetyp, hier „int“ d.h. eine Ganze Zahl. Das muss sich mit dem Returnwert „0“ vertragen!

Die Argumentliste ist hier die leere Liste: `main()`

2. Erste Schritte in C

Der Aufbau eines C-Programms

Beispiel: Das kürzeste ablauffähige C-Programm

```
int main() {
        return 0;
}
```

Beispiel: *Hello, world!* auf dem Bildschirm ausgeben

```
#include <stdio.h>

int main() {
        printf("Hello, world!");
        return 0;
}
```

- **#include** <**stdio.h**> ist eine Präprozessordirektive (*printf* dem Compiler bekanntmachen).
- Die Funktion **printf** sendet die Stringkonstante (Zeichenkette) *"Hello, world!"* auf den Bildschirm.

2. Erste Schritte in C

Der Aufbau eines C-Programms

Beispiel: Ausgabe von *Hello world!* ohne Zeilenwechsel

```
#include <stdio.h>

int main() {
        printf("Hello, world!");
        printf("Hello, world!");
        return 0;
}
```

Beispiel Ausgabe von *Hello world!* mit Zeilenwechsel

```
#include <stdio.h>

int main() {
        printf("Hello, world!\n");
        printf("Hello, world!\n");
        return 0;
}
```

→ \n („Backslash n“ steht für new line)

2. Erste Schritte in C

Der Aufbau eines C-Programms

Beispiel: Prozedurale Programmierung (stepwise refinement)

Problemstellung:
Eine Buchhandlung verzeichnet Titel und Verlag aller verkauften Bücher. Der Buchhändler möchte für die Nachbestellung eine Liste erzeugen, die nach Verlagen alphabetisch geordnet ist,

Zwei Teilprobleme:
1. Eingabe der Verkaufszahlen
2. Sortieren der Titel nach Verlagen und Ausgabe

```
int main() {
        eingabe();
        sortAusgabe();
        return 0;
}
```

→ Funktionsaufruf! Das ist keine Funktionsdefinition!

→ Dieses Programm ist nicht ablauffähig, da die 2 Funktionsdefinitionen fehlen!

2. Erste Schritte in C

Der Aufbau eines C-Programms

Beispiel: Prozedurale Programmierung (stepwise refinement)

```
#include <stdio.h>

void eingabe(){
        printf("eingabe\n");
}

void sortAusgabe(){
        printf("sortAusgabe\n");
}

int main() {
        eingabe();
        sortAusgabe();
        return 0;
}
```

→ Dieses Programm ist ablauffähig, da alle Funktionsdefinitionen und –aufrufe vorhanden sind!

2. Erste Schritte in C

Der Aufbau eines C-Programms

Übersetzen und **Ausführen** des Programms

- Das Quellprogramm befindet sich in der Datei `bestellung.c.`

- Aufruf des Compilers:
  `cc bestellung.c`
  ruft den Compiler auf, der das Programm, falls es fehlerfrei ist, in eine ausführbare Datei `a.out` übersetzt.

  Sie können auch den Compiler gcc verwenden.

- Mit der Option `-o` können Sie auch einen anderen Namen für die ausführbare Datei wählen:
  `cc -o bestellung bestellung.c`

- Mit dem Aufruf `bestellung` wird das Programm nun ausgeführt und gibt die Namen der Funktionen auf dem Bildschirm aus.

2. Erste Schritte in C

Der Aufbau eines C-Programms

Beispiel: Verzinsung eines Sparguthabens

```
#include <stdio.h>

int main() {
        float guthaben;
        guthaben = 500.00 + 500.00 * 0.01;
        printf("Guthaben nach einem Jahr: %f", guthaben);
        return 0;
}
```

Erläuterungen

- **`guthaben`** ist eine Variable vom Typ **`float`** (Gleitkommazahl)
- **`printf`** benötigt den Typ der auszugebenden Variablen (Formatangabe von **`printf`**)

2. Erste Schritte in C

Der Aufbau eines C-Programms

Beispiel: Eingabe des Guthabens über die Tastatur

```
#include <stdio.h>

int main() {
        float guthaben;
        float anlagebetrag;
        printf("Wie hoch ist Ihr Anlagebetrag? ");
        scanf("%f", &anlagebetrag);
        guthaben = anlagebetrag + anlagebetrag * 0.01;
        printf("Guthaben nach einem Jahr: %f", guthaben);
        return 0;
}
```

Erläuterungen

- Genau wie `printf` benötigt `scanf` eine Formatangabe.
- Beachten Sie bitte das **Und-Zeichen (`&`)** vor dem Variablennamen `anlagebetrag`. Dies ist notwendig, denn `scanf` erwartet hier die **Adresse der Variablen**, daher wird der **Adressoperator `&`** verwendet.

2. Erste Schritte in C

Ein- und Ausgabe

Allgemeines zur Ein- und Ausgabe

- Funktionen `scanf` und `printf`
- Die Ein- und Ausgabe werden durch die **Bibliothek** `<stdio.h>` unterstützt.
- `stdio` steht für **standard input output**.
- Um die Definitionen dieser Bibliothek im Quellprogramm bekannt zu machen, muss (sollte) das **Header-File** `<stdio.h>` mit `#include` inkludiert werden.

Hinweis:

- `#include <stdio.h>`
  Die Zeichen < und > sagen aus, dass die Datei `stdio.h` in einem **Systemdirectory** steht.
- `#include "fvmeine.h"`
  Die Zeichen `"` und `"` sagen aus, dass die Datei `fvmeine.h` im **Arbeitsverzeichnis** steht.

2. Erste Schritte in C

Ein- und Ausgabe

Ausgabe mit printf

- printf(<Format-String>, <Argument1>, <Argument2>, ...)
- **Steuerzeichen** sind Zeichenkonstanten und werden mit dem **Backslash** \ eingeleitet:

\b	backspace, der Cursor geht eine Position nach links
\n	**new line, der Cursor geht zum Anfang der nächsten Zeile**
\r	carriage return, der Cursor geht zum Anfang der aktuellen Zeile
\t	horizontal tab, der Cursor geht zur nächsten horizontalen Tabulatorposition
\"	" wird ausgegeben
\?	? wird ausgegeben
\\	\ wird ausgegeben

- Sollen Werte von Variablen ausgegeben werden, so sind die Variablen hinter dem **Formatstring** als Argumente anzugeben. Zusätzlich ist im Format-String dort ein **Formatzeichen** anzugeben, wo der Wert ausgegeben werden soll. Für jeden Variablentyp gibt es ein spezielles Formatzeichen. Einige sind:

%i oder %d	Integervariable (Ganze Zahl)
%f	Float-Variable (Fließkommazahl)
%lf	Double-Variable (Fließkommazahl mit doppelter Genauigkeit)

2. Erste Schritte in C

Ein- und Ausgabe

Beispiele zu Formatangaben:

- %5.2f ist ein Float, mindestens 5-stellig mit 2 Nachkommastellen
- %7i stellt die Integerzahl rechtsbündig mit 7 Stellen dar
- %5d wie oben: d und i sind äquivalent
- %-5i stellt die Integerzahl linksbündig mit 5 Stellen dar
- %05i stellt die Integerzahl rechtsbündig mit 5 Stellen dar. Die führenden Leerzeichen werden durch Nullen ersetzt.

Beispiele:

```
printf("Datum in deutschem Format: %02d.%02d.%4d\n",tag, monat, jahr);
```

→ Ausgabe, z.B. 10.10.2025 (wenn `tag = 10`, `monat = 10` und `jahr = 2025`.

Der Formatstring wird genau so bei **scanf** (Einlesen von der Tastatur) verwendet.

→ `scanf("%d.%d.%d", &tag, &monat, &jahr);`

liest ein Datum wie im obigen Format ein, z.B. 10.10.2025.

2. Erste Schritte in C

Ein- und Ausgabe

Eingabe von der Tastatur mit scanf

- scanf(<Format-String>, &<Argument1>, &<Argument2>, ...)
- Ähnlich wie bei `printf` muss `scanf` mitgeteilt werden, vom welchem Typ die Eingabe ist. Hierzu werden die gleichen **Formatzeichen** wie bei `printf` verwendet (z. B. `%i` und `%f`).
- Mit `scanf` ist es nicht möglich, eine Zeichenkette auszugeben. Hierzu muss `printf` verwendet werden.
- **Variablen**, die in `scanf` verwendet werden, müssen mit dem **Adressoperator (&)** versehen werden.

Beispiel:

```
#include <stdio.h>

int main() {
        int wert1, wert2, summe;
        printf("Bitte geben Sie 2 Zahlen ein: \n");
        scanf("%i%i", &wert1, &wert2);
        summe = wert1 + wert2;
        printf("Die Summe von %i und %i ist %i.", wert1, wert2, summe);
        return 0;
}
```

2. Erste Schritte in C

Kommentare

- Hilfestellung für den Menschen, um Programme besser verstehen zu können.
- Die Größe des übersetzten Programms wird durch Kommentare nicht erhöht.
- In C gibt es den **Kommentarblock /* */.** Der Compiler behandelt alles zwischen /* und */ als Kommentar. Kommentarblöcke können sich über mehrere Zeilen erstrecken.
- In C gibt es den **Zeilenkommentar** (ab ANSI C99): //
  Hier wird alles bis zum Ende der betreffenden Zeile als Kommentar interpretiert.

2. Erste Schritte in C

Kommentare

Beispiel:

```
/*******************************************************************************
 * wurzel.c                                                                    *
 * Es wird die Wurzel aus einer Zahl gezogen. Das geht nur für positive Zahlen. *
 * Autor: Frank Victor, 01.10.2025                                             *
 ******************************************************************************/

#include <stdio.h>
#include <math.h>

int main(){
          float x;
          printf("Bitte geben Sie eine Zahl ein: \n");
          scanf("%f", &x);
          if (x < 0) {                  // Bei negativer Eingabe brechen wir ab!
                    printf("Fehler: Zahl muss positiv sein! \n");
                    return -1;
          }
          printf("Die Wurzel von %f ist %f.", x, sqrt(x));
          return 0;
}
```

3. Datentypen, Variablen und Konstanten in C

Fundamentale Datentypen

Ein Bit ist eine einzelne Speicherzelle, in der der Wert 0 oder 1 stehen kann.
Ein Byte ist eine Folge von 8 Bits.
Ein Wort besteht aus 16 oder 32 Bits.

Beispiel 3.1: Speicherauszug

1024	00011001
1032	00111111
1040	11001001
1048	01010101

Interpretation der Daten über Datentypen → Getypte Sprachen → Alles hat einen Typ.

Standard-Datentypen in C sind:

- Fundamentale Datentypen (siehe Tabelle, nächste Folie)
- Zusammengesetzte Datentypen: Strukturen, Arrays und Zeiger

3. Datentypen, Variablen und Konstanten in C

Fundamentale Datentypen

Datentyp	ANSI Mindestgröße	Bedeutung	Wertebereich (mindestens)	Format-zeichen
char	1 Byte	1 (!) Zeichen als Ganzzahl	-128 bis 127, z.B. ‘A‘ = 65	%c
short (int)	2 Byte	Ganzzahl	-32.768 bis 32.767	%hi
int	2 Byte 4 Byte (Wortgröße)	Ganzzahl	-32.768 bis 32.767	%i oder %d
long (int)	4 Byte	Ganzzahl	-2.147.483.648 bis 2.147.483.647	%li
float	4 Byte	Fließkommazahl, Genauigkeit: 7 Nachkommastellen	$-3.4 * 10^{38}$ bis $3.4 * 10^{38}$	%f
double	8 Byte	Fließkommazahl, Genauigkeit: 16 Nachkommastellen	$-1.7 * 10^{308}$ bis $1.7 * 10^{308}$	%lf
long double	16 Byte	Fließkommazahl, Genauigkeit: 24 Nachkommastellen	$- / + 10^{3.000}$	%Lf

3. Datentypen, Variablen und Konstanten in C

Fundamentale Datentypen

- Die Größe der Datentypen ist maschinenabhängig und stehen in `<limits.h>` und `<float.h>`
- Es gibt aber typische Größen für Datentypen: int = 4 Byte (= Wortgröße des Rechners)
- unsigned int, unsigned short und unsigned long bezeichnen **vorzeichenlose Ganze Zahlen** und haben einen doppelt so großen Wertebereich wie die ohne unsigned.

Datentyp	Wertebereich (mindestens)	Format-zeichen
unsigned char	0 bis 255	%c
unsigned short (int)	0 bis 65535	%hu
unsigned int	0 bis 65535	%u
unsigned long (int)	0 bis 4294967295	%lu

- Ab C99 gibt es den Datentyp Boolean mit den Werten true und false.
- Ab C11 gibt es weitere Datentypen, z.B. char16_t und char32_t zur Unterstützung des Unicode Zeichensatzes oder auch weitere Typen mit 8, 16, 32, 64 Bit für Integer.

3. Datentypen, Variablen und Konstanten in C

Variablen und Konstanten

Variablen bezeichnen **Speicherplätze**, in denen **Werte** eines bestimmten **Datentyps** abgelegt werden können.

Name für eine Variable (Bezeichner, identifier) besteht aus

- Buchstaben, Ziffern oder dem Unterstrich (_).
- Der Bezeichner muss mit einem Buchstaben oder einem Unterstrich beginnen.
- Es gibt keine festgelegte Höchstlänge für Bezeichner.
- Zwischen Groß- und Kleinbuchstaben wird unterschieden.

Konventionen für die Namen von Variablen

- Ein Bezeichner wird normalerweise klein geschrieben.
- Der Bezeichner soll ein "sprechender" Name sein (Verwendungszweck).
- Bezeichner mit mehreren Wörtern: Unterstrich verwenden oder den ersten Buchstaben der nachfolgenden Wörter großschreiben.

Schlüsselwörter (Keywords) dürfen nicht als Bezeichner verwendet werden. Beispiele sind: `break`, `case`, `char`, `const`, `do`, `else`, `float`, `for`, `if`, `int`, `long`, `return`, `short`, `sizeof`, `struct`, `switch`, `void`, `while`

3. Datentypen, Variablen und Konstanten in C

Variablen und Konstanten

Definition von Variablen in C

- Besteht aus der Typangabe, auf die der Name folgt.
- Ist ein Statement und muss mit einem Semikolon abgeschlossen werden.
- Wenn mehr als eine Variable mit dem gleichen Typ definiert werden soll, können die Bezeichner durch Komma abgetrennt hinter den Datentyp geschrieben werden.

Beispiel: Definitionen von Variablen, die Speicherplatz reservieren:

```
double gehalt, miete;
int tag, jahr;
unsigned long entfernung;
```

Initialisierung von Variablen in der Definition

```
<typ> <variable> = <wert>;
```

3. Datentypen, Variablen und Konstanten in C

Variablen und Konstanten

Beispiel:

```
#include <math.h>

int main() {
        double preis = 109.99, umsatzSteuer = 0.19;
        double verkaufsPreis = preis + preis * umsatzSteuer;

        int wert = -5;
        unsigned int absolutWert = abs(wert);
        …
}
```

3. Datentypen, Variablen und Konstanten in C

Variablen und Konstanten

Konstanten (Literale) in C sind
- Integerkonstanten
- Fließkommakonstanten
- Zeichenkonstanten
- Stringkonstanten

- Konstanten (Literale) präsentieren **unveränderbare Werte**, die einen bestimmten Typ besitzen.
- Im Gegensatz zu Variablen haben Konstanten keine Namen und sind nicht adressierbar.

Beispiel:
```
0       → Typ int
3.14159 → Typ double
```

3. Datentypen, Variablen und Konstanten in C

Variablen und Konstanten

Integerkonstanten (Konstanten vom Typ int) können in 3 verschiedenen Schreibweisen angegeben werden:

- Dezimal
- Oktal → Null (0) davor schreiben
- Hexadezimal → 0x oder 0X davor schreiben

Beispiel: Integerkonstanten sind: 20 (dezimal) 024 (oktal) 0X14 (hexadezimal)

Eine Fließkommakonstante (Konstante vom Typ double)

- Wissenschaftliche Schreibweise (Exponentialschreibweise) oder
- Dezimalschreibweise

Beispiel:
Fließkommakonstanten sind: 3.14159, 3e1, 100.0E-3

3. Datentypen, Variablen und Konstanten in C

Variablen und Konstanten

Eine Zeichenkonstante (Konstante vom Typ char) repräsentiert ein einzelnes Zeichen. Zeichenkonstanten werden immer von **zwei einfachen Hochkommata** eingeschlossen.

Beispiel:
Zeichenkonstanten sind: 'f ' '2' ', ' ' ' (Leerzeichen).

Eine Stringkonstante
- Besteht aus null oder mehr Zeichen, die in doppelten Anführungszeichen stehen.
- Kann über mehrere Programmzeilen gehen. Ein Backslash \ als letztes Zeichen in einer Zeile gibt an, dass die Stringkonstante in der nächsten Zeile fortgesetzt wird.
- Hat den Wert des zusammengesetzten Typs **array von Zeichen** und besteht aus den einzelnen Zeichen des Strings und einem abschließenden Null-Byte ('\0').

Beispiel:
Stringkonstanten sind: "" (leerer String) "a" "Franky"

3. Datentypen, Variablen und Konstanten in C

Variablen und Konstanten

Der Operator **sizeof** ermittelt die Größe eines Objekts während der Compilierung;

`sizeof(<typname>|<ausdruck>)`

Das Ergebnis von **sizeof** ist vom Typ **size_t**, was gleichbedeutend mit **unsigned int** ist.

Beispiel:

```
#include <stdio.h>

int main() {
        double z = 5.2;

        printf("int ist %i Byte groß.\n", sizeof(int));
        printf("short int ist %i Byte groß.\n", sizeof(short));
        printf("z ist als double %i Byte groß.\n", sizeof(z));
        return 0;
}
```

3. Datentypen, Variablen und Konstanten in C

Automatische Typkonvertierung

Problem

In mathematischen Ausdrücken können Variablen und Konstanten mit unterschiedlichen Typen verwendet werden. → Der Compiler führt **automatische Typumwandlungen** durch.

Beispiel:

```
int main() {
        float mittelwert, summe = 1.0;
        int n = 1;
        mittelwert = summe/n;
        return 0;
}
```

<float> = <float> / <int> →
<float> = <double> / <double>

3. Datentypen, Variablen und Konstanten in C

Automatische Typkonvertierung

Beispiel:

```
int main() {
        float mittelwert;
        int summe = 1;
        int n = 1;
        mittelwert = summe/n;
        return 0;
}
```

<float> = <int> / <int> → Integer-Division

Regeln für die automatische Typkonvertierung

- Werden zwei Variablen unterschiedlichen Typs durch einen Operator miteinander verknüpft, dann wird der Wert des schwächeren Operanden in den Typ des stärkeren Operanden umgewandelt und das Resultat hat den stärkeren Typ.

- **Nicht-Ganzzahlige** Operationen finden auf doubles statt**:** floats → doubles

- **Ganzzahlige** Operationen finden auf ints, unsigned ints oder longs statt: chars und shorts → ints

3. Datentypen, Variablen und Konstanten in C

Automatische Typkonvertierung

Beispiel:

```
float a = 1.0;
int i = 2;
a = a /i;
```

a: float → double
i: int → double
<float> = <double> / <double> → Ergebnis 0.5

```
double a = 1.0;
int i = 2, j = 1;
a = a + j / i;
```

1/2 → <int> / <int> → 0 (<int>)
<double> = 1.0 (<double>) + 0.0 (<double>) → Ergebnis 1.0

```
long a = 50000;
int i = 20000;
a = a + i;
```

<long> = 50000 (<long>) + 20000 (<int>) →
<long> = 50000 (<long>) + 20000 (<long>) → Ergebnis 70000

```
char a = 70, c = 30;
a = c + a;
```

<char> = 30 (<char>) + 70 (<char>) →
<char> = 30 (<int>) + 70 (<int>) → Ergebnis 100

4. Ausdrücke und Operatoren in C

Grundlegende Begriffe

Ein einfacher Ausdruck besteht aus einer Konstanten oder einer Variablen.
Ein zusammengesetzter Ausdruck besitzt eine oder mehrere Operationen, die durch Operatoren realisiert werden. Die Objekte einer Operation heißen Operanden.

Jeder Ausdruck liefert immer einen Wert (Evaluation des Ausdrucks). Da C eine typisierte Sprache ist, hat jeder Wert auch einen Typ.

Beispiel:

`3.1415`	einfacher Ausdruck	Wert: **`3.1415`**, Typ **`double`**
`0`	einfacher Ausdruck	Wert: **`0`**, Typ: **`int`**
`wert_x`	einfacher Ausdruck	Wert: Inhalt des Speicherplatzes **`wert_x`**
`1 + 2 * 2`	zusammengesetzter Ausdruck	Wert: **`5`**, Typ **`int`**

4. Ausdrücke und Operatoren in C

Arithmetische Operatoren

Operator	Bedeutung
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulo
-	Vorzeichenumkehr

Beispiel:

`2 + 5 % 2` Wert: `3`

`-5 - 3` Wert: `-8`

4. Ausdrücke und Operatoren in C

Relationale Operatoren

Operator	Bedeutung
<	Kleiner
<=	Kleiner gleich
>	Größer
>=	Größer gleich
==	Gleich
!=	Ungleich

Die relationalen Operatoren liefern in C die Werte 1 (wahr), falls die Relation erfüllt ist, andernfalls 0 (falsch).

Beispiel:

`1 < 2` Wert: `1`

`1 != 2` Wert: `1`

`(1 == 1) * 2` Wert: `2` (da `1 == 1` den Wert `1` hat)

4. Ausdrücke und Operatoren in C

Zuweisungsoperatoren

Operator	Bedeutung
=	Einfache Zuweisung
op=	Kombinierte Zuweisung

- **`op`** ist ein beliebiger arithmetischer, bitweiser oder Shift-Operator
- **`Ausdruck1 op= Ausdruck2`** <==> **`Ausdruck1 = Ausdruck1 op Ausdruck2`**.
- Der linke Operand muss ein Variablenname oder ein Pointer-Ausdruck sein, das heißt, eine Referenz auf einen veränderbaren Speicherbereich. Der Wert, den die Zuweisungs-Operatoren liefern, ist der Wert, der im linken Operanden durch die Zuweisung gespeichert wird (L-Wert oder left value).

Beispiel:

Es sei **`a = 1`** und **`b = 2`**.

`a = 1 * 2`	Wert: **`2`**	mit dem Effekt, dass **`a`** den Wert **`2`** erhält
`(a = 1) * 2`	Wert: **`2`**	mit dem Effekt, dass **`a`** den Wert **`1`** erhält
`b *= 2`	Wert: **`4`**	mit dem Effekt, dass **`b`** den Wert **`4`** erhält
`a == b = c`	Syntaxfehler, das Ergebnis **`0`** des Vergleichs ist keine Variable.	

4. Ausdrücke und Operatoren in C

Inkrement- und Dekrementoperatoren

Operator	Bedeutung	Wert des Ausdrucks nach der Operation	Wert der Variablen nach der Operation
++variable	Pre-Inkrement-Operator	wert + 1	wert + 1
--variable	Pre-Dekrement-Operator	wert - 1	wert - 1
variable++	Post-Inkrement-Operator	wert	wert + 1
variable--	Post-Dekrement-Operator	wert	wert - 1

- Hierbei steht **++** für eine Inkrementierung und **--** für eine Dekrementierung um 1.
- Bei den Pre-Inkrement- und Pre-Dekrement-Operatoren wird zuerst der Wert der Variablen modifiziert und der modifizierte Wert ist der Wert des Ausdrucks.
- Dagegen wird bei den Post-Inkrement- und Post-Dekrement-Operatoren zuerst der aktuelle Wert der Variablen als Wert des Ausdrucks geliefert und dann erst wird der Wert der Variablen modifiziert.

4. Ausdrücke und Operatoren in C

Inkrement- und Dekrementoperatoren

Beispiel:
Es sei `a = 1`.

`--a`	Wert: `0`	mit dem Effekt, dass `a` den Wert `0` erhält
`a--`	Wert: `1`	mit dem Effekt, dass `a` den Wert `0` erhält
`a++ + 4`	Wert: `5`	mit dem Effekt, dass `a` den Wert `2` erhält

Logische Operatoren

Operator	Bedeutung
!	logische Negation Liefert 1 (wahr), falls der Operand gleich 0 (falsch) ist, andernfalls 0 (falsch).
&&	logisches Und Liefert 1 (wahr), falls beide Operanden ungleich 0 (wahr) sind, andernfalls 0 (falsch)
||	logisches Oder Liefert 1 (wahr), falls ein Operand ungleich 0 (wahr) ist, andernfalls 0 (falsch).

4. Ausdrücke und Operatoren in C

Logische Operatoren

Beispiel:

`0 && 0`	Wert: `0`
`1 && 1`	Wert: `1`
`0 || 0`	Wert: `0`
`1 || 0`	Wert: `1`
`0 && (0 || 1)`	Wert: `0`
`1 == 2 && (a = 5)`	Wert: `0` (`a = 5` wird nicht mehr ausgeführt, da der Wahrheitswert bereits nach der Auswertung von `1 == 2` feststeht)

Es gilt:

`Ausdruck == 0` ist äquivalent mit `!Ausdruck`

`Ausdruck != 0` ist äquivalent mit `Ausdruck`

4. Ausdrücke und Operatoren in C

Konditionaler Operator

- Hat die Form: `ausdruck1 ? ausdruck2 : ausdruck3`
- `ausdruck1` wird ausgewertet. Wenn `ausdruck1` den Wert wahr hat, dann evaluiert der Gesamtausdruck zum Wert von `ausdruck2` und sonst zum Wert von `ausdruck3`.

Beispiel: Das Minimum zweier Zahlen `a` und `b`

```
printf("Das Minimum beträgt : %i", a < b ? a : b);
```

Typ cast Operator

- Hat die Form: `(typname) ausdruck`

Beispiel:

`5 / (long) 2` ergibt den Wert `2`, da die Division ganzzahlig ist.

`5 / (float) 2` ergibt den Wert `2.5`, da die Division eine Fließkommaoperation ist.

5. Anweisungen in C

Klassifizierung von Anweisungen

- Eine einfache Anweisung ist die kleinste ausführbare Einheit eines Programms. Anweisungen werden in C in **ausführbaren Maschinencode** und in Java in **Bytecode** übersetzt und wie alle Statements mit Semikolon abgeschlossen.
- Die leere Anweisung hat die Form: **;**
- Die zusammengesetzte Anweisung (Anweisungsblock) besteht aus mehreren Anweisungen.
- Alle Anweisungen gliedern sich in Verzweigungen und Schleifen.

Beispiel:

Folgendes ist erlaubt (wenn auch nicht sinnvoll):

```
int a;
a++ ;;; a++;
```

5. Anweisungen in C

Eine zusammengesetzte Anweisung (Anweisungsblock) hat die Form

```
{
        anweisung1;
        anweisung2;
        ...
}
```

Die zusammengesetzte Anweisung wird als **Einheit** behandelt und kann überall dort stehen, wo auch eine einfache Anweisung stehen kann.

Beispiel:

```
if (konto < 0 && dispoLimitNichtausgeschoepft)
{
        dispoZinsenBerechnen();
        kontoBelasten();
}
```

5. Anweisungen in C

Die if-Anweisung

Die if-Anweisung in C

- Realisiert eine **Entscheidung**, mit alternativen Teilen eines Programms fortzufahren
- Ist eine **konditionale Verzweigung** im Sinne eines „entweder, oder"
- Hat eine Bedingung in Form eines **logischen Ausdrucks,** der wahr oder falsch liefert. Dieser Ausdruck wird ausgewertet.

```
if (ausdruck)
         anweisung1
[else
         anweisung2]
```

Anmerkung:
`[` und `]` sind Symbole der Backus-Naur-Form (BNF) und bedeuten, dass dieser Teil ein- oder null-mal vorkommen kann.

5. Anweisungen in C

Die if-Anweisung

Ausführung der if-Anweisung

- Zuerst wird *ausdruck* ausgewertet.
- Falls *ausdruck* ungleich Null (wahr) ist, wird *anweisung1* ausgeführt.
- Falls *ausdruck* gleich Null (falsch) ist, wird *anweisung2* ausgeführt.
- Falls kein *else*-Teil vorhanden ist und der Ausdruck gleich Null (falsch) ist, wird die nächste nach der *if* Anweisung folgende Anweisung wird ausgeführt.

Beispiel:

```
if (n > 0)
        printf("%i ist positiv.\n", n);
else
        printf("%i ist Null oder negativ.\n", n);
```

Anmerkung:
Wenn nur einfache Anweisungen vorkommen, können die Klammern weggelassen werden.

5. Anweisungen in C

Die if-Anweisung

Beispiel:

```
if (a > 0)
        if (b != 0)
                c = 1;
        else
                c = 2;
```

```
if (a > 0)
        if (b != 0)
                c = 1;
else
        c = 2;
```

Aufgrund der fehlenden Klammern ist dieser Code mehrdeutig. Compiler lösen Sie diesen Konflikt, in dem sie die erste Variante wählen: Ein **`else`** bezieht sich immer auf das nächste **`if`** in seiner Umgebung. Diese Problematik bezeichnet man als dangling else.

5. Anweisungen in C

Die switch-Anweisung

Die switch-Anweisung in C

- Realisiert eine **Mehrfachverzweigung**
- Ist vergleichbar mit einem „Schalter".

```
switch (ausdruck)
        anweisungsblock;
```

Im *anweisungsblock* stehen Anweisungen mit dem case-Label

```
case konstanter-ausdruck: anweisung
```

ausdruck bezeichnen wir auch als Selektorausdruck.

5. Anweisungen in C

Die switch-Anweisung

Ausführung der switch-Anweisung

- Zuerst wird *ausdruck* ausgewertet und falls notwendig nach *int* konvertiert.
- Dann wird ein Sprung in *anweisungsblock* zu der Anweisung mit dem *case*-Label ausgeführt, dessen *konstanter-ausdruck* gleich dem Wert von *ausdruck* ist.

- Alle Konstanten der *case*-Labels müssen verschieden und vom Datentyp *int* sein.
- Mindestens eine Anweisung in *anweisungsblock* muss ein Label tragen.
- Falls der Wert von *ausdruck* mit keinem *konstanten-ausdruck* übereinstimmt, wird das Programm mit der ersten Anweisung hinter *anweisungsblock* fortgesetzt.

Achtung:

- Der Selektorausdruck sowie die *case*-Label müssen vom Datentyp *int* sein bzw. zu diesem Datentyp konvertiert werden → automatische Typkonvertierung: *chars* → *ints*.

5. Anweisungen in C

Die switch-Anweisung

Erweiterung der switch-Anweisung um ein default-Label.

```
default: anweisung
```

Ausführung der switch-Anweisung mit default-Label

- Die Anweisung hinter *default* wird ausgeführt, falls kein *case*-Label mit dem Wert des Selektorausdrucks übereinstimmt.

Vorsicht

- Der Selektorausdruck bestimmt, an welche Stelle des Anweisungsblocks eingesprungen wird. Beachten Sie, dass von dieser Stelle an **alle** Anweisungen bis zum Ende von *anweisungsblock* ausgeführt werden. Dies nennt man fall-through.
- Fall-throughs sollte man bei der Programmierung vermeiden und wenn man sie einsetzt, sollte man dies dokumentieren.
- Vermeiden kann man den fall-through durch die **`break`** Anweisung im Anweisungsblock. **`break`** erzwingt, dass an dieser Stelle der Anweisungsblock verlassen wird.

5. Anweisungen in C

Die switch-Anweisung

Beispiel: Zählen der Vokale in einem Text
(es fehlt eine Schleife, die den gesamten Text durchgeht und die einzelnen Buchstaben einliest).

```
char gelesenerBuchstabe;

int az = 0, ez = 0, iz = 0, oz = 0, uz = 0; // Zähler für die Vokale

switch (gelesenerBuchstabe) {
        case 'a':
                ++az;
        case 'e':
                ++ez;
        case 'i':
                ++iz;
        case 'o':
                ++oz;
        case 'u':
                ++uz;
}
```

→ Dieses Programm ist falsch (fall-through).

5. Anweisungen in C

Die switch-Anweisung

Beispiel (Korrektur):

```
char gelesenerBuchstabe;
int az = 0, ez = 0, iz = 0, oz = 0, uz = 0;

switch (gelesenerBuchstabe) {
        case 'a': ++az; break;
        case 'e': ++ez; break;
        case 'i': ++iz; break;
        case 'o': ++oz; break;
        case 'u': ++uz; break;
}
```

→ Dieses Programm ist korrekt (kein fall-through).

5. Anweisungen in C

Die switch-Anweisung

Beispiel (Fortsetzung):
Zählen auch der Vokale in Großbuchstaben.

```
char gelesenerBuchstabe;
int az = 0, ez = 0, iz = 0, oz = 0, uz = 0;
switch (gelesenerBuchstabe) {
                 // bei Großbuchstaben ist break weggelassen,
                 // da sie ebenfalls gezählt werden sollen.
        case 'A':
        case 'a': ++az; break;
        case 'E':
        case 'e': ++ez; break;
        case 'I':
        case 'i': ++iz; break;
        case 'O':
        case 'o': ++oz; break;
        case 'U':
        case 'u': ++uz; break;
}
```

5. Anweisungen in C

Die goto-Anweisung

Die goto-Anweisung in C

- Sollte nur in Ausnahmefällen verwendet werden, da sie unübersichtlichen Programmcode erzeugt.
- Stellt eine unbedingte Verzweigung zu einer bestimmten Anweisung im Programm dar.
- Dazu muss die Anweisung, zu der verzweigt werden soll, markiert werden (Label, Marke).
- Ein Label hat die Form

```
name: anweisung
```

- Labels werden als Sprungziele für goto verwendet: `goto name;`

Hinweis:
In Java gibt es kein goto!

5. Anweisungen in C

Die goto-Anweisung

Beispiel: Distanzen in Millimeter umrechnen

```
#include <stdio.h>

int main() {
        double distanz;
        int distanzeinheit;
eingabe:
        printf("\n Menü - Sie können auswählen: ");
        printf("\n km = 1  m = 2");
        printf("\n dm = 3  cm = 4");
        printf("\n Auswahl: ");
        scanf("%i",&distanzeinheit);
        printf("\n Distanz: ");
        scanf("%f",&distanz);
```

5. Anweisungen in C

Die goto-Anweisung

```
        switch (distanzeinheit)
        {
                case 1: distanz = distanz * 1000000; break;
                case 2: distanz = distanz * 1000; break;
                case 3: distanz = distanz * 100; break;
                case 4: distanz = distanz * 10; break;
                default:
                        printf("\n Falsche Eingabe! Neuer Versuch!");
                        goto eingabe;
        }
        printf("\nDies sind %f Millimeter.", distanz);
        return 0;
}
```

5. Anweisungen in C

Die while-Anweisung

```
while (ausdruck)
      anweisung
```

Ausführung der while-Schleife

- Zuerst wird *ausdruck* ausgewertet.
- Dann wird *anweisung* wiederholt ausgeführt, solange der Wert von *ausdruck* ungleich Null (wahr) ist.
- Vor jeder erneuten Ausführung von *anweisung* wird *ausdruck* erneut ausgewertet.
- *ausdruck* nennt man Schleifenbedingung (loop control).
- Es kann also der Fall eintreten, dass *anweisung* gar nicht ausgeführt wird.

5. Anweisungen in C

Die while-Anweisung

Beispiel: Summe der Zahlen von 1 bis n.

```
#include <stdio.h>

int main() {
        int n, summe = 0, i = 1;
        printf("\nBitte n eingeben: ");
        scanf("%i", &n);

        while (i <= n) {
                summe += i;
                i++;
        }

        printf("\nDie Summe der Zahlen von 1 bis %i ist: %i", n, summe);
        return 0;
}
```

5. Anweisungen in C

Die while-Anweisung

Beispiel (Fortsetzung):
Snapshot Technik, um zu sehen, wie die Schleife funktioniert.

Angenommen *n* sei 4. Dann ergibt sich der folgende **Snapshot**:

	summe	i
0. Schleifendurchlauf	0	1
1. Schleifendurchlauf	1	2
2. Schleifendurchlauf	3	3
3. Schleifendurchlauf	6	4
4. Schleifendurchlauf	10	5 → Abbruch der Schleife, da 5 > 4.

5. Anweisungen in C

Die do-while-Anweisung

```
do
          anweisung
while (ausdruck);
```

Ausführung der do-while-Schleife

- Zuerst wird *anweisung* ausgeführt.
- Dann wird *ausdruck* ausgewertet. *anweisung* wird sooft wiederholt ausgeführt, solange wie der Wert von *ausdruck* ungleich Null (wahr) ist.
- Nach jeder Ausführung von *anweisung* wird *ausdruck* erneut ausgewertet.
- Es kann also **nicht** der Fall eintreten, dass *anweisung* gar nicht ausgeführt wird, das heißt, *anweisung* wird mindestens einmal ausgeführt.

5. Anweisungen in C

Die do-while-Anweisung

Beispiel: Unterschied zwischen while- und do-while-Schleife.

(a)
```
while (i <= 3) {
        summe += i;
        i++;
}
```

(b)
```
do {
        summe +=i;
        i++;
} while (i <= 3);
```

- Für i <= 3 sind werden die Schleifen gleich oft ausgeführt.
- Für i > 3 wird Schleife (a) keinmal und Schleife (b) einmal ausgeführt.

Anmerkung: Setzen Sie *do-while* **nur** ein, wenn Sie garantieren möchten, dass die Schleife mindestens einmal durchlaufen wird (z.B. für die Anzeige eines Menüs, das mindestens 1-mal gezeigt werden soll). Man zieht *while* fast immer vor, da es vor der Schleife prüft.

5. Anweisungen in C

Die for-Anweisung

```
for (initialisierung; test; modifikation) anweisung;
```

Ausführung der for-Schleife

- *initialisierung* wird einmal am Schleifenanfang ausgeführt.
- *test* wird vor jedem Schleifendurchlauf getestet und *anweisung* gefolgt von *modifikation* werden solange ausgeführt, bis der Wert von *test* gleich Null (falsch) ist.
- Wenn *test* fehlt, handelt es sich um eine **Endlosschleife**.

Beispiel:

```
for (i = 0; ...                   // Initialisierung von i mit 0
for (i = 1, j = 1, k = i/j; ...   // Initialisierung von 2 Indizes (Komma-Operator)

for ( ...; i < n; ...             // Test, ob index < n ist
for ( ...;;...                    // Endlosschleife

for ( ...; ...; i++) ...          // i wird um 1 erhöht
for ( ...; ...; --i, --j) ...     // Dekrementierung von 2 Indizes um 1
```

5. Anweisungen in C

Die for-Anweisung

Beispiel: Mittelwert von n Zahlen

```
#include <stdio.h>

int main() {
        float mittel, summe = 0.0, eingabe;
        int i, n;
        printf("\nWieviele Zahlen sollen eingelesen werden? ");
        scanf("%i", &n);

        for (i = 1; i < = n; i++) {
                printf("Geben Sie die Zahl ein: ");
                scanf("%f", &eingabe);
                summe += eingabe;
        }

        mittel = summe / n;
        printf("\nDer Mittelwert beträgt: %f", mittel);
        return 0;
}
```

5. Anweisungen in C

Die break-Anweisung

```
break;
```

- *break* kann in der *switch*-Anweisung und in Schleifen verwendet werden.
- *break* beendet die *for*-, *while*- oder *do-while*-Schleife bzw. den *switch*-Anweisungsblock.

Beispiel:

```
while (...)
{
        ...;
        break;
        ...;
}
...;
```

5. Anweisungen in C

Die continue-Anweisung

```
continue;
```

- *continue* kann nur in Schleifen verwendet werden.
- *continue* beendet einen Schleifendurchlauf der *for*-, *while*- oder *do-while*-Schleife.

Beispiel:

```
while (...)
{
        ...;
        continue;
        ...;
        ...;
}
...;
```

6. Die zusammengesetzten Datentypen in C

Kategorien von Datentypen in C:

- Die fundamentalen Datentypen (`char`, `int`, `long`, `float` usw.)
- Die zusammengesetzten (strukturierten) Datentypen: `array`, `struct` und `enum`.

In der realen Welt haben wir es mit komplexeren Objekten zu tun!
→ Fundamentale Datentypen sind zu low level!

Arrays, Strukturen und **Enumerations**

- Array: Objekte, die aus einer festen Anzahl von **gleichartigen** Komponenten bestehen.
- Struktur: Objekte, die aus einer festen Anzahl von **verschiedenartigen** Komponenten bestehen.
- Enumeration: Aufzählungstyp, z.B. Montag, Dienstag, Mittwoch, …

6. Die zusammengesetzten Datentypen in C

Arrays

```
typ arrayname[arraygroesse];
```

- **Definition eines (eindimensionalen) Arrays** mit den **Indizes:** 0 bis `arraygroesse-1`

- Arrays sind Datenstrukturen, in denen die Komponenten eine bestimmte Ordnung haben.
- array = (dt. "ordnen/aufstellen, Ordnung“)
- Ein array ist eine Reihung, Folge, Vektor oder Feld mit der Eigenschaft
    - die Komponenten sind vom gleichen Typ
    - ist eine Sequenz, wo jede Komponente ihren festen Platz hat (Index)

Beispiele:
- Folgen und Vektoren in der Mathematik
- Sitzreihen in einem Hörsaal
- Geordnete Liste von Namen

6. Die zusammengesetzten Datentypen in C

Arrays

Beispiel:

```
int werte[3];
```

- Reserviert Speicherplatz für insgesamt 3 Array-Elemente des Datentyps **`int`**.
- Die Nummerierung beginnt mit dem Index 0.
- Hier ein mögliches Bild des Speichers mit Adressen von Speicherzellen:

Wert von Element 0	132
Wert von Element 1	136
Wert von Element 2	140

6. Die zusammengesetzten Datentypen in C

Arrays

```
arrayname[i]
```

- Der (lesende und schreibende) Zugriff auf ein Array-Element mit dem Index **i.**
- Elemente eines Arrays werden durch den subscript-Operator [ ] indiziert.

Beispiel:

```
int i;
int a[3];

a[0] = 2;
a[1] = a[0];
a[1] = a[1] + 1000;
for (i = 1; i < 3; i++)
        a[i] = 111;
```

a[0]	a[1]	a[2]
2	111	111

6. Die zusammengesetzten Datentypen in C

Arrays

Beispiel:

```
int eingabe[3];
eingabe[3] = 4711;
```

- Hier wird die Array Grenze überschritten.
- Es wird bei der Compilierung kein Fehler gemeldet.
- Der Zugriff erfolgt auf irgendeine Speicherzelle „hinter dem Array".
- C führt **keine Überprüfung der Array-Grenzen durch** (→ Programmabsturz möglich → „illegal instruction (core dumped)".
- Nicht-determininstisches Verhalten von Programmen.
- O-Notation: Aufwand von Algorithmen mit Arrays

6. Die zusammengesetzten Datentypen in C

Arrays

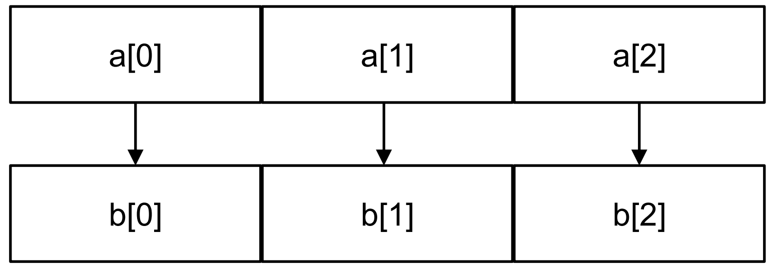
Beispiel: Kopieren von Arrays

```
#include <stdio.h>

int main() {
        int i;
        int a[3];
        int b[3];
        for (i=0; i < 3; i++){
                printf("\nBitte %i-tes Element eingeben: ", i);
                scanf("%i", &a[i]);
        }
        b = a;              // Fehler, das klappt nicht!!
        return 0;
}

advm1> cc arraysFalsch.c
"arraysFalsch.c", line 11.9: 1506-025 (S) Operand must be a modifiable lvalue.
```

6. Die zusammengesetzten Datentypen in C

Arrays

Beispiel: Kopieren von Arrays




```
#include <stdio.h>

void main(){
        int i;
        int a[3];
        int b[3];

        for (i=0; i < 3; i++) {
                printf("\nBitte %i-tes Element eingeben: ", i);
                scanf("%i", &a[i]);
                b[i] = a[i]; // Komponentenweises Kopieren
        }

        printf("\nDas Array b hat folgende Elemente: ");
        for (i=0; i < 3; i++)
                printf("%i", b[i]);
}
```

Aufwand: O(n)

6. Die zusammengesetzten Datentypen in C

Arrays

Strings (Zeichenketten)

- Die Zeichen eines Strings sind in aufeinanderfolgenden Array-Elementen vom Datentyp **`char`** gespeichert.
- Das Ende des Strings und damit seine aktuelle Länge wird durch ein Null-Byte ('\0' oder 0) markiert.

Beispiel:
Der String **`"Frank"`** ist ein **`char`**-Array im Speicher:

'F'	**'r'**	**'a'**	**'n'**	**'k'**	**'\0'**

6. Die zusammengesetzten Datentypen in C

Arrays

Initialisierung eines Arrays über die Initialisierungsliste

```
typ arrayname[arraygroesse] = {wert-1, wert-2, ... wert-n};
```

- Die ersten n Elemente werden mit **`wert-1`** bis **`wert-n`** initialisiert.
- Werden weniger Werte als **`arraygroesse`** angegeben, so werden die übrigen Elemente implizit mit 0 initialisiert.
- Wenn wir die Elemente initialisieren, brauchen wir **`arraygroesse`** nicht unbedingt anzugeben. Der Compiler legt dann ein Array an, dessen Größe gleich der Anzahl der Elemente der Initialisierungsliste ist.
- Ein Array von Zeichen (String) kann entweder mit einer Liste von Zeichenkonstanten initialisiert werden oder mit einer Stringkonstanten.

6. Die zusammengesetzten Datentypen in C

Arrays

Beispiel:

`int a[2] = {1, 2};`	Die Array-Elemente werden mit 1, 2 initialisiert.
`int a[]= {1, 2, 3};`	Die Array-Elemente werden mit 1, 2, 3 initialisiert. Es wird ein Array der Größe 3 angelegt.
`int a[3] = {1, 2};`	Die Array-Elemente werden mit 1, 2, 0 initialisiert.
`int a[3] = {1, 2, 3, 4}`	Syntax-Fehler
`char zeichen[] = {'C'};`	Die Array-Elemente werden initialisiert. Die Größe des Arrays ist 1.
`char zeichenkette[] = "C";`	Die Array-Elemente werden initialisiert. Die Größe des Arrays ist 2.
`char s[5] = "Frank";`	Syntax-Fehler: „Frank" hat 6 Elemente.

6. Die zusammengesetzten Datentypen in C

Arrays

Symbolische Konstanten

- Die Größe eines Arrays muss man im Programm festlegen.
- Mit einer symbolischen Konstanten legt man die Größe eines Arrays am Anfang des Programms fest.
- Eine Variable kann in C in eine symbolische Konstante umgewandelt werden durch:

```
const typ name = wert;
```

- Hierdurch ist es nicht mehr möglich, den Wert der Variablen zu ändern.
- Symbolische Konstanten müssen initialisiert werden. Ansonsten meldet der Compiler einen Fehler.

Beispiel:

```
const int arraygroesse_1 = 10;
const int arraygroesse_2;          // Fehler, Konstante muss initialisiert werden.
```

6. Die zusammengesetzten Datentypen in C

Arrays

Beispiel:

```
#include <stdio.h>

void main(){
        int i;
        const int groesse = 3;
        int a[groesse];
        int b[groesse];

        for (i=0; i < groesse; i++) {
                printf("\nBitte %i-tes Element eingeben: ", i);
                scanf("%i", &a[i]);
                b[i] = a[i];
        }

        printf("\nDas Array b hat folgende Elemente: )";
        for (i=0; i < groesse; i++)
                printf("%i",b[i]);
}
```

Aufwand: O(n)

6. Die zusammengesetzten Datentypen in C

Arrays

Mehrdimensionale Arrays

```
typ arrayname[dim1][dim2];
```

- Sind Arrays mit mehr als einer Dimension, d.h. zur Indizierung der Elemente ist mehr als ein Index erforderlich.
- In der Programmierpraxis kommen meistens nur **2-dimensionale** Arrays vor.
- Zweidimensionale Arrays kann man sich als Tabellen vorstellen, deren Elemente durch einen Zeilen- und einen Spaltenindex indiziert werden.

Beispiel:

```
int tab[2][3];
```

tab[0][0]	tab[0][1]	tab[0][2]
tab[1][0]	tab[1][1]	tab[1][2]

- Reserviert Platz für 6 Array-Elemente des Datentyps **`int`**.
- **`tab[i][j]`** bezeichnet einen **`int`**-Wert in der Speicherzelle mit dem Index **`[i][j]`**.

6. Die zusammengesetzten Datentypen in C

Arrays

Zugriff auf ein 2-dimensionales array

```
arrayname[i][j]
```

- Bewirkt den (lesenden und schreibenden) Zugriff auf das Array-Element mit dem Index `[i][j]` über den subscript-Operator **[][]**.

Beispiel:

```
int i, j;
int tab[2][3];

tab[0][0] = 2;
tab[1][0] = tab[0][0];
tab[1][2] = tab[1][0] * 100;
for (i = 0; i < 2; i++)
        for (j = 0; j < 3; j++)
                tab[i][j] = 10;
```

10	10	10
10	10	10

6. Die zusammengesetzten Datentypen in C

Arrays

Beispiel: Initialisierung eines Arrays mit 2 Schleifen

```
int main() {
        int i, j;
        const int zeilenzahl = 2;
        const int spaltenzahl = 3;
        int tab[zeilenzahl][spaltenzahl];

        for (i = 0; i < zeilenzahl; i++)
                for (j = 0; j < spaltenzahl; j++)
                        tab[i][j] = i + j;
        return 0;
}
```

Aufwand: $O(n^2)$

6. Die zusammengesetzten Datentypen in C

Strukturen

```
struct StructName {
        komponenten
};
```

Strukturen (structs) in C

- Werden eingesetzt, um inhaltlich zusammengehörige Komponenten zusammenzufassen.
- Vereinfachen die Handhabung komplexer Datenstrukturen und sind benutzerdefinierte Typen.
- In Java gibt es **keine** Strukturen, aber Klassen.

- Mit `StructName` wird die Struktur vereinbart.
  Achtung: Benutzerdefinierte Typen immer groß schreiben!

- Die Komponenten `komponenten` (auch members genannt) sind Variablendeklarationen

```
typ komponentenName;
```

6. Die zusammengesetzten Datentypen in C

Strukturen

- Durch die Deklaration einer Struktur wird also ein neuer Typ festgelegt.
- Es wird an dieser Stelle noch kein Speicher für Objekte reserviert.
- Für den neuen Typ kann man – wie für gewöhnliche Typen – Variablen dieses Typs definieren.

Für das **`struct StructName`** definiert man Strukturvariablen durch

```
struct StructName variablenName, variablenName-2, …;
```

- Nicht schön ist dabei, dass bei der Definition von Strukturvariablen immer das Schlüsselwort **`struct`** mitgeführt werden muss.
- Mit **`typedef`** kann man das vermeiden und eigene Datentypen definieren.

6. Die zusammengesetzten Datentypen in C

Strukturen

```
typedef struct StructName {
        komponenten
} TStructName;
```

- Mit **`TStructName`** wird der Typname für die Struktur **`StructName`** vereinbart.
- Für den neuen Typ kann man – wie für gewöhnliche Typen – Variablen dieses Typs definieren:

```
TStructName variablenName;
```

6. Die zusammengesetzten Datentypen in C

Strukturen

Beispiel: Studierendeneintrag

- Mit unseren bisherigen Mitteln (ohne Struktur):

```
char name[15];
char vorname[15];
int matnr;
```

Nachteil: Der Studierendeneintrag ist in diesem Fall keine Einheit.

- Mit Struktur:

```
struct Stud {
        char name[15];
        char vorname[15];
        int matnr;
};
```

6. Die zusammengesetzten Datentypen in C

Strukturen

Beispiel (Fortsetzung):

- Das Beispiel definiert eine Struktur mit dem Namen *Stud* mit 3 Komponenten.
- Wir könnten nun Variablen definieren:

```
struct Stud stud1, stud2;
```

- Mit **`typedef`** könnte man das eleganter machen:

```
typedef struct Stud_ {
        char name[20];
        char vorname[20];
        int matnr;
} Stud;
```

- Wir haben nun einen Strukturtyp mit dem Namen **`Stud`** erzeugt und können diesen Typ verwenden:

```
Stud stud1, stud2;
```

6. Die zusammengesetzten Datentypen in C

Strukturen

Beispiel (Fortsetzung):

- Wir initialisieren nun unsere Variable **`stud1`**.

```
Stud stud1 = {"Victor", "Frank", 11114};
```

- Die Größe von **`Stud`** lässt sich mit **`sizeof(Stud)`** bestimmen.
- Es gilt: **`sizeof(Stud) >= sizeof(char[15]) + sizeof(char[15]) + sizeof(int).`**

- Es gilt auf manchen Maschinen **nicht** die Gleichheit, da manchmal ein address alignement durchgeführt wird, d. h., dass auf manchen Maschinen Variablen beginnend mit geraden Adressen abgespeichert werden.

6. Die zusammengesetzten Datentypen in C

Strukturen

Namensgleichheit für Strukturtypen in C

- In einem Programm müssen die Namen der verwendeten Typen verschieden sein und auch die Namen der Komponenten.

Nicht erlaubt wäre:

```
struct b { int x; double x;};
struct b { double x, y;};
```

Zugriff auf die Komponenten einer Strukturvariablen

Auf die Komponenten (members) einer Strukturvariablen wird durch den member-of-Operator (.) zugegriffen:

```
strukturVariable.komponentenName
```

Er liefert als Wert das Objekt, das durch **`komponentenName`** selektiert wird. Den Komponenten einer Strukturvariablen können unter Verwendung des member-of-Operators Werte zugewiesen werden.

6. Die zusammengesetzten Datentypen in C

Strukturen

Beispiel: Zugriff auf Brüche

```
#include <stdio.h>

typedef struct Rational_ {
        int zaehler;
        int nenner;
} Rational;

int main() {
        Rational x1;
        printf("\nBitte Zähler eingeben: ");
        scanf("%i", &x1.zaehler);
        printf("\nBitte Nenner eingeben: ");
        scanf("%i", &x1.nenner);
        return 0;
}
```

6. Die zusammengesetzten Datentypen in C

Strukturen

Beispiel: Kundendaten

```
typedef struct Kunde_ {
        long nummer;
        double umsatz;
} Kunde;

Kunde kunde1, kunde2;
```

- `kunde2` ist korrekt. Der Datentyp ist `Kunde`.
- `kunde1.nummer + 1` ist korrekt. Der Datentyp ist `long`.
- `kunde2.umsatz = 20000;` ist korrekt.

Falsch wäre: `umsatz / 2` und `Kunde.umsatz = 30000`

6. Die zusammengesetzten Datentypen in C

Strukturen

Beispiel:

```
struct TelefonNummer {
         char vorwahl[7];
         long nummer;
};

struct TelefonVerzeichnisEintrag {
         char name[20];
         char vorname[20];
         struct TelefonNummer home, office;
};
```

Durch die Variablendefinition

`struct TelefonVerzeichnisEintrag eintrag1;` wird Speicher für 54 chars und 2 longs reserviert.

Zugriff auf die verschachtelte Struktur:

```
eintrag1.office.nummer = 555555;
```

6. Die zusammengesetzten Datentypen in C

Strukturen

Beispiel:

a) Geben Sie jeweils ein *struct* für die Typen
   - **Punkt** und
   - **Rechteck** an.

Ein Punkt besteht aus zwei Koordinaten.
Ein Rechteck besteht aus 2 Punkten (linke untere Ecke und rechte obere Ecke).

b) Schreiben Sie eine Funktion **erzeugePunkt**, die als Argumente zwei Koordinaten x und y hat.
   Sie erzeugt einen Punkt mit diesen Koordinaten und gibt ein *struct* vom Typ *Punkt* zurück.

6. Die zusammengesetzten Datentypen in C

Strukturen

Beispiel (Fortsetzung):

(a)

```
struct Punkt {                // struct für Punkt
        float x;
        float y;
};

struct Rechteck {             // struct für Rechteck
        struct Punkt lu;
        struct Punkt ro;
};
```

(b)

```
struct Punkt erzeugePunkt(int x, int y) {  // Funktionen kommen später.
        struct Punkt temp;                  // Müssen Sie hier nicht verstehen
        temp.x = x;
        temp.y = y;
        return temp;
};
```

6. Die zusammengesetzten Datentypen in C

Enumerations

```
enum Name {Element_1, Element_2,..., Element_n};
```

Aufzählungstypen (enums) in C

- Mit **`enum`** können Aufzählungstypen in C deklariert werden.
- Aufzählungstypen sind für **Integer-Variablen** gedacht, die nicht jeden beliebigen Wert annehmen dürfen, sondern auf eine begrenzte Anzahl von Werten beschränkt sind.
- Diese Werte werden über Namen angesprochen.
- Typische Kandidaten für solche Aufzählungen sind die Tage einer Woche oder die Monate eines Jahres.

Beispiel: Deklaration des Aufzählungstyps **`Wochentag`**:

```
enum Wochentag {SON, MON, DIE, MIT, DON, FRE, SAM};
```

`SON` entspricht dem Integerwert **`0`**, **SAM** entspricht dem Integerwert **`6`**. Jedes Namenssymbol steht für einen ganzzahligen Wert, wobei jedes Namenssymbol um den Wert 1 größer ist als das vorhergehende. Der Anfangswert für das erste Symbol ist standardmäßig **`0`**.

6. Die zusammengesetzten Datentypen in C

Enumerations

Beispiel:

```
enum Himmelsrichtung {NORD, OST=90, SUED=180, WEST=270};
```

`NORD` hat den Wert `0`. Würde `SUED` nicht mit `180` initialisiert, dann hätte es automatisch den Wert `91`.

Beispiel:

```
enum Wochentag {SON, MON, DIE, MIT, DON, FRE, SAM};

enum Wochentag heute;
heute = MON;
```

- Damit wird die Variable `heute` des Aufzählungstyps `Wochentag` definiert. Sie darf nur Werte im Bereich von `SON` bis `SAM` annehmen. In unserem Fall wird der Variablen `heute` der Wert `MON` zugewiesen. `MON` ist keine Zeichenkette, sondern steht für den Wert `1`.
- In C – im Gegensatz zu C++ und Java – wäre leider auch die Zuweisung eines beliebigen `int`-Werts erlaubt, z.B. `heute = 20`.

6. Die zusammengesetzten Datentypen in C

Enumerations

Beispiel:

```
enum Arbeitstag {MONTAG, DIENSTAG, MITTWOCH, DONNERSTAG, FREITAG};

int main() {
        enum Arbeitstag heute = MONTAG;

        switch (heute) {
                case MONTAG:     printf("Montag\n"); break;
                case DIENSTAG:   printf("Dienstag\n"); break;
                case MITTWOCH:   printf("Mittwoch\n"); break;
                case DONNERSTAG: printf("Donnerstag\n"); break;
                case FREITAG:    printf("Freitag\n"); break;
                default:
                        printf(„Heute ist kein Arbeitstag\n");
        }
}
```

- Die Elemente einer `enum` können überall verwendet werden, wo **`int`-Konstanten** zulässig sind.

7. Funktionen in C

Sinn von Funktionen

- Beispiel: **`main()`** in C, der Name ist vorgegeben.
- Man hat die Möglichkeit, selbst Funktionen mit eigenem Namen zu definieren.
- Hier durch kann man Programme modular aufbauen und prozedurale Teile, die eine zusammenhängende Aufgabe bezeichnen, zu Einheiten zusammenzufassen.

Lernziele

- Wie können wir Funktionen definieren?
- Wie können wir Funktionen aufrufen?
- Wie können wir Werte als Parameter an Funktionen übergeben?
- Wie organisiert man Programmsysteme mit vielen Funktionen (getrennte Compilierung)?

7. Funktionen in C

Funktionsdefinition und Aufruf

Die Definition einer Funktion besteht aus

- dem Datentyp des Rückgabewerts
- dem Funktionsnamen
- der Argumentliste
- dem Funktionsblock und
- dem Rückgabewert (return value).

Funktionsdefinition in C

```
typ name(arg-1, arg-2, ..., arg-n){
        funktionsblock
}
```

arg-1 ... arg-n sind formale Parameter (formale Argumente) der Form: **argtyp argname**

7. Funktionen in C

Funktionsdefinition und Aufruf

Funktionsaufruf in C

```
name(par-1, par-2, ..., par-n)
```

Die Argumente **`par-1, ..., par-n`** sind beliebige Ausdrücke und werden als aktuelle Parameter (aktuelle Argumente) bezeichnet.

Übergabe der Argumente

- Beim Aufruf werden zuerst die aktuellen Parameter ausgewertet und Speicherplatz wird für die korrespondierenden formalen Parameter reserviert.
- Dann wird der Wert des i-ten aktuellen Parameters dem i-ten formalen Parameter zugewiesen, das heißt, die Werte der aktuellen Parameter werden in den lokalen Datenraum der Funktion **kopiert**. Dies bezeichnet man als Wertübergabe (call by value).
- Die Wertübergabe wird verwendet, wenn man **Eingaben** an eine Funktion machen möchte, die nicht geändert werden sollen.
- Wenn man die geänderten Werte einer Funktion ins Hauptprogramm zurückgeben will, benötigt man in C Zeiger (pointer).

7. Funktionen in C

Funktionsdefinition und Aufruf

Beispiel:

```
#include <stdio.h>

void addieren() {               // Funktionsdefinition für addieren
        int x, y, summe;
        printf("\nGeben Sie 2 Zahlen ein: ");
        scanf("%i%i", &x, &y);

        summe = x + y;

        printf("\nDie Summe ist %i.", summe);
}

int main() {
        addieren();             // Aufruf der Funktion addieren
        return 0;
}
```

7. Funktionen in C

Funktionsdefinition und Aufruf

Beispiel: Sichtbarkeit von Variablen

```
#include <stdio.h>

void addieren() {
        int x, y, summe;                              // Achtung:
        printf("\nGeben Sie 2 Zahlen ein: ");         // x und y sind lokal für die Funktion addieren.
        scanf("%i%i", &x, &y);
        summe = x + y;
        printf("\nDie Summe ist %i.", summe);
}

int main() {                                          // Achtung:
        int x=10, y=5;                                // Egal, welche Werte für x und y eingegeben
        addieren();                                   // werden, die Ausgabe ist immer x = 10 und y = 5.
        printf("\nDer Wert von x ist %i.",x);
        printf("\nDer Wert von y ist %i.",y);
        return 0;
}
```

7. Funktionen in C

Funktionsdefinition und Aufruf

Beispiel: *addieren* mit 2 Funktionsparametern

```
#include <stdio.h>

void addieren(int x, int y) {                  // Zwei Funktionsparameter
        int summe;
        summe = x + y;
        printf("\nDie Summe ist %i.", summe);
}

int main() {
        int a, b;
        printf("\nGeben Sie 2 Zahlen ein: ");
        scanf("%i%i", &a, &b);
        addieren(a, b);                        // Aufruf von addieren mit zwei aktuellen Parametern.
        return 0;
}
```

7. Funktionen in C

Funktionsdefinition und Aufruf

Beispiel: *addieren* mit 2 Funktionsparametern und Rückgabewert

```
#include <stdio.h>

int addieren(int x, int y){
        int summe;
        summe = x + y;
        return summe;                    // Rückgabewert steht in summe und ist vom Typ int
}

int main() {
        int a, b;
        printf("\nGeben Sie 2 Zahlen ein: ");
        scanf("%i%i", &a, &b);

        printf("\nDie Summe von %i und %i ist %i.", a, b, addieren(a, b));
        return 0;
}
```

7. Funktionen in C

Funktionsdeklaration und getrennte Übersetzung

Sinn von Deklarationen

- Eine Deklaration einer Funktion stellt einen Bezug zu einer Funktionsdefinition her und macht die Funktion mit ihrem Namen und ihrer Parameterliste im *Compilierungslauf* bekannt.
- Eine Definition gilt natürlich auch als Deklaration.
- Die Definition einer Funktion darf nur einmal im gesamten Programm vorkommen, die Deklaration darf beliebig oft vorkommen, im Allgemeinen in jedem File, in dem man eine entsprechende Funktion verwendet.
- Aus diesem Grund fasst man Deklarationen in eigenen Files, den Headerfiles, zusammen. Diese haben die Extension .h (für header) und werden in den entsprechenden Definitionsfiles als auch in den Files inkludiert, wo die entsprechende Funktion verwendet wird.

Funktionsdeklaration in C

```
typ name(arg-1, arg-2, ..., arg-n);
```

7. Funktionen in C

Funktionsdeklaration und getrennte Übersetzung

Beispiel: *addieren* mit Deklaration und Definition

```
#include <stdio.h>

int addieren(int x, int y);                  // Deklaration oder:
                                             // int addieren(int, int);
int main() {
        int a, b;
        printf("\nGeben Sie 2 Zahlen ein: ");
        scanf("%i%i", &a, &b);

        printf("\nDie Summe von %i und %i ist %i.", a, b, addieren(a, b));
        return 0;
}

int addieren(int x, int y){
        int summe;
        summe = x + y;
        return summe;
}
```

7. Funktionen in C

Funktionsdeklaration und getrennte Übersetzung

Beispiel: Getrennte Kompilierung

(1) Die **Definitionsdatei** `main.c` enthält:

```
#include <stdio.h>          // Dateinamen in < und > bezeichnen system files

#include "mymath.h"         // Dateinamen in " und " bezeichnen files im aktuellen
                            // Directory. Hier inkludieren wir die Deklarationen
                            // für unsere selbst geschriebenen Funktionen.

int main() {
        int i, j, minimum, absolut;
        printf("\nBitte geben Sie zwei Zahlen ein: ");
        scanf("%i%i",&i,&j);
        minimum = min(i,j);        // Aufruf unserer Funktion min
        printf("\nDas Minimum von %i und %i ist: %i", i, j, minimum);
        absolut = abs(i);          // Aufruf unserer Funktion abs
        printf("\nDer Absolutbetrag von %i ist: %i", i, absolut);
        return 0;
}
```

7. Funktionen in C

Funktionsdeklaration und getrennte Übersetzung

Beispiel (Fortsetzung):

(2) Die **Definitionsdatei** `mymath.c` enthält:

```
#include "mymath.h"          // Wir inkludieren die Header-Datei,
                             // um Konsistenz zwischen Deklarationen
                             // und Definitionen zu garantieren.

int min(int v1, int v2) {
        return (v1 <= v2 ? v1 : v2);
}

int abs(int v1) {
        return (v1 < 0 ? -v1 : v1);
}
```

7. Funktionen in C

Funktionsdeklaration und getrennte Übersetzung

Beispiel (Fortsetzung):

(3) Die **Deklarationsdatei** `mymath.h` enthält:

```
int min(int v1, int v2);  // oder: int min(int, int);
int abs(int v1);          // oder: int abs(int);
```

(4) Übersetzen der beiden Definitionsdateien durch: `cc main.c mymath.c` oder **UNIX-make** (siehe unten):

```
# Dateiname: mathTest.mak

mathTest: mymath.o main.o
        cc -o mathTest mymath.o main.o

mymath.o: mymath.h mymath.c
        cc -c mymath.c

main.o: mymath.h main.c
        cc -c main.c
```

→ **Aufruf:** `make -f mathTest.mak`

8. Zeiger in C

Sinn von Zeigern

Eine **Variable** ist eindeutig identifiziert durch:

- Name
- Inhalt
- Größe
- **Adresse** (Position im Speicher).

Beispiel:

```
int x = 50;
```

- Name: **`x`**
- Inhalt: **`50`**
- Größe: Größe des Typs **`int`**
- Adresse: **`0xA0000000`**

Wenn wir den Namen **`x`** im Programm verwenden, weiß das Programm, dass **`x`** an einer bestimmten Adresse liegt.

8. Zeiger in C

Address-of-Operator

Der Address-of-Operator **`&x`** liefert die Adresse einer Variablen **`x`**.

Beispiel :

```
#include <stdio.h>

int main() {
        int x = 100;
        printf("\nAdresse von x(%i) ist %p", x, &x);
        x = 200;
        printf("\nAdresse von x(%i) ist %p", x, &x);
        return 0;
}
```

→ Auch wenn sich der Wert von x ändert, bleibt doch die Adresse &x die gleiche, z.B. 2ff22a60

8. Zeiger in C

Address-of-Operator

Zeiger
- Eine Variable, die eine Adresse aufnimmt, nennt man einen Zeiger (pointer).
- Ein **Zeiger** wird definiert durch:

```
typ *name;
```

`name` ist der Name des Zeigers und **`typ`** ist der Typ der Variablen, auf die der Zeiger zeigt.

Beispiel:

```
int x = 50;
int *z;
z = &x;
```

`z` ist ein Zeiger auf **`int`**-Werte. Die Adresse von **`x`** wird dem Zeiger **`z`** zugewiesen.

8. Zeiger in C

Dereferenzierungsoperator

- Wie kann man auf den Wert zugreifen, auf den ein Zeiger zeigt?
- Mit dem Dereferenzierungsoperator `*x` wird der Wert angesprochen, auf den der Zeiger `x` zeigt.

Beispiel:

```
int x = 50;
int *z = &x;
printf("\nDereferenzierung: z = %i", *z);
```

Ausgabe:

```
Dereferenzierung: z = 50
```

8. Zeiger in C

Adressen als Funktionsparameter

Parameterübergabe

- Bisher: Nur Eingabeparameter von Funktionen
  - Wertübergabe (Call by Value)
  - Argumente werden in den lokalen Datenraum der Funktion kopiert.
- Bisher: Nur der Rückgabewert der Funktion kann einen veränderten Wert an das Hauptprogramm liefern.
- Nicht möglich: Mehrere Werte zurückgeben, die die Funktion verändert.
- Viele Problemstellungen machen es jedoch notwendig, mehr als einen Resultatwert weiter zu verarbeiten.
- Zeiger bieten die Möglichkeit, die Werte von Aufrufparametern zu ändern.
- Unterschiede: Call by value und Call by reference.

8. Zeiger in C

Adressen als Funktionsparameter

Beispiel: Werte zweier Variablen vertauschen

```
#include <stdio.h>

void tauschen(int *v1, int *v2) { // Zeiger als Parameter
        int tmp = *v2;
        *v2 = *v1;
        *v1 = tmp;
}

int main() {
        int i = 10, j =20;
        printf("\nVor dem Tauschen: %i %i", i, j);
        tauschen (&i, &j);
        printf("\nNach dem Tauschen: %i %i\n", i, j);
        return 0;
}
```

8. Zeiger in C

Adressen als Funktionsparameter

Zusammenhang zwischen Arrays und Pointern

- Zwischen Arrays und Pointern besteht in C ein enger Zusammenhang.
- Der Name eines Arrays wird in Ausdrücken zu einem Zeiger konvertiert:
- **`ArrayName == &ArrayName[0]`**
- Arrays können als formale Parameter in Funktionen stehen. Die Funktionsdefinition hat dann die Form:

```
functyp funcname (arraytyp *arrayname, int arraygroesse) // oder
functyp funcname (arraytyp arrayname[], int arraygroesse){
         .
         .
         .
}
```

Als aktuellen Parameter verwendet man ein Array beim Aufruf der Funktion folgendermaßen:

```
funcname(arrayname, arraygroesse)
```

8. Zeiger in C

Adressen als Funktionsparameter

Beispiel: Integer-Arrays als Funktionsparameter

```
#include <stdio.h>

void einlesen(int a[], int n) {
        int i;
        for (i = 0; i < n; i++){
                printf("\nBitte %i-tes Element eingeben: ", i);
                scanf("%i", &a[i]);
        }
}

void ausgeben(const int a[], int n) {
        int i;
        printf("\nDie %i Array-Elemente heißen: ", n);
        for (i = 0; i < n; i++)
                printf("%i", a[i]);
}
```

8. Zeiger in C

Adressen als Funktionsparameter

Beispiel (Fortsetzung):

```
void addZahl(int *a, int n, int zahl) {
        int i;
        for (i = 0; i < n; i++)
                a[i] = a[i] + zahl;
}

int main() {
        const int groesse = 10;
        int meinArray[groesse];

        einlesen(meinArray, groesse);
        ausgeben(meinArray, groesse);
        addZahl(meinArray, groesse, 4);
        ausgeben(meinArray, groesse);
        return 0;
}
```

8. Zeiger in C

Adressen als Funktionsparameter

Beispiel: Character-Arrays als Funktionsparameter

```
#include <stdio.h>

void einlesen(char s[]) {
        printf("\nBitte String eingeben:");
        scanf("%s", s);
}

int stringLaenge(const char s[]) {
        int i = 0;
        while (s[i++]);  // oder: while (s[i++] != '\0');
                         // zur Erinnerung: '\0' == 0 aber '\0' != '0'
        return (i -1);
}
```

8. Zeiger in C

Adressen als Funktionsparameter

Beispiel (Fortsetzung):

```
int main() {
        const int laenge = 10;
        char meinString[laenge];

        einlesen(meinString);

        printf("\nDer String %s hat die Laenge %i.",
                meinString, stringLaenge(meinString));

        return 0;
}
```

8. Zeiger in C

Adressen als Funktionsparameter

Beispiel: Character-Arrays als Funktionsparameter

Schreiben Sie eine Funktion mit dem Namen **`grossSchreiben`**

- die als Parameter eine Zeichenkette *s* hat und
- alle Kleinbuchstaben der Zeichenkette in Großbuchstaben umwandelt.
- Es soll die Anzahl der umgewandelten Zeichen als Returnwert zurückgegeben werden.

Hinweis: Benutzen Sie die folgenden C-Bibliotheksfunktionen:

- **`int islower(char c)`** liefert den Wert true, falls **`c`** ein Kleinbuchstabe ist
- **`char toupper(char c)`** liefert den Großbuchstaben von **`c`** bzw. **`c`** selbst, wenn **`c`** bereits ein Großbuchstabe ist

Wichtig: Es soll ein Character-Array benutzt werden.

8. Zeiger in C

Adressen als Funktionsparameter

Beispiel (Fortsetzung)

```
int grossSchreiben(char s[]) {
        int a = 0;
        int i = 0;

        while (s[i] != '\0') {
                if (islower(s[i])) {
                        s[i] = toupper(s[i]);
                        a++;
                }
        i++;
        }
        return a;
}
```

8. Zeiger in C

Adressen als Funktionsparameter

Beispiel:

Schreiben Sie ein Hauptprogramm, in dem eine Zeichenkette eingelesen und dann die obige Funktion `grossSchreiben` für diese aufgerufen wird. Die Ausgabe soll so aussehen:

Eingegebene Zeichenkette: CaliFornia
Großgeschriebene Zeichenkette: CALIFORNIA.
Es wurden 8 Zeichen umgewandelt.

```
void main (){
        int a;
        char s[];
        printf("Bitte Zeichenkette eingeben: ");
        scanf("%s", s);
        printf("Eingegebene Zeichenkette: %s" , s);
        a = grossSchreiben(s);
        printf("Großgeschriebene Zeichenkette: %s", s);
        printf("Es wurden %i Zeichen umgewandelt.", a);
}
```

8. Zeiger in C

Adressen als Funktionsparameter

Exkurs: Argumente der Kommandozeile
In Unix kann man einem C-Programm beim Programmstart schon Eingabeparameter übergeben.
Diese nennt man Argumente der Kommandozeile.

Sie werden in der Parameterliste von **`main`** über **`argc`** und **`argv`** angesprochen:

```
int main(int argc, char *argv[]) { ... }
```

- Die Variable **`argc`** gibt die Anzahl der Argumente des Aufrufs als Resultatwert zurück.
- Die Argumente selber stehen in einen Array von Strings. Dabei ist jeder String ein Argument. Auf dieses Array wird der Pointer **`argv`** übergeben.
- Die Struktur des Arrays **`argv`** sieht so aus:

```
argv →  "Programmname"
        "Aufrufargument_1"
        "Aufrufargument_2"
        …
        '\0'
```

8. Zeiger in C

Adressen als Funktionsparameter

Beispiel:
Ein C-Programm **`showarguments.c`**, das die Argumente der Kommandozeile ausgibt, sieht so aus:

```
int main(int argc, char *argv[]) {
        printf("Die Argumente der Kommandozeile sind: \n");
        while (*argv != 0) // oder: for(int i=0;i<argc;i++)
                printf("%s", *argv++);
        return 0;
}
```

Und hier ein Beispiel für die Ausgabe, wenn das exec-File **`showarguments`** heißt:

```
advm1> showarguments Luisa Christina
```

Die Argumente der Kommandozeile heißen:
`showarguments Luisa Christina`

9. Rekursion

- Eine Funktion heißt rekursiv, wenn sie sich direkt oder indirekt aufruft.
- Ein Objekt heißt rekursiv, wenn es sich teilweise selbst enthält.

Beispiel: Fakultätsfunktion

fak: N $\rightarrow$ N mit

$$\text{fak}(n) = \begin{cases} 1, & \text{für } n == 0 \\ n * \text{fak}(n-1), & \text{für } n > 0 \end{cases}$$

```
long fak(int n) {
        if (n == 0)
                return 1;
        else
                return n * fak(n - 1);
}
```

Regel:
Eine rekursive Funktion besteht immer aus Rekursionsbasis und Rekursionsschritt.

9. Rekursion

Beispiel: Rekursive Abarbeitung dieser Funktion für den Aufruf `fak(3)`

fak(3)

==> 3 * fak(2)

==> 2 * fak(1)

==> 1 * fak(0)

hier ist die Abbruchbedingung für die Rekursion erreicht: n == 0. Nun setzt der Rückwärtsprozess ein, in dem gerechnet wird.

<== 1 * 1 = 1

<== 2 * 1 = 2

<== 3 * 2 = 6 dies ist der Wert des Aufrufs von fak(3).

9. Rekursion

Anmerkungen

- Jeder rekursive Algorithmus lässt sich auch iterativ (d.h. mit Schleifen) realisieren.
- Bei gleichem Algorithmus ist die Iteration etwas schneller.
- Einige Probleme sind von Natur aus rekursiv. Das Finden einer iterativen Lösung ist dann oft umständlich.

Beispiel: Iterative Definition der Fakultätsfunktion

fak: N → N mit

fak(n) = 1 * 2 * 3 * ... *n

```
long fak(int n) {
        int i;
        long fakultaet = 1;
        for (i = 1; i <= n; i++)
                fakultaet = fakultaet * i;
        return fakultaet;
}
```

Teil B: Prozedurale Programmierung in der Sprache C

Literatur zur Vertiefung

Goll, J., Dausmann, M., C als erste Programmiersprache: Mit den Konzepten von C11, Springer Vieweg, 2014

Griffiths, D., C von Kopf bis Fuß, O'Reilly, 2012

Kernighan, B.W., Ritchie, D. M., The C Programming Language (ANSI C version), Markt+Technik, 2000

Wolf, J., Grundkurs C: C-Programmierung verständlich erklärt, Rheinwerk Computing, 2016

Algorithmen
und
Programmierung I

WS 2025 / 2026

Teil C: Objektorientierte Programmierung in der Sprache Java

Prof. Dr. Frank Victor
TH Köln

Inhalt

Teil C: Objektorientierte Programmierung in der Sprache Java

1 Motivation: Herleitung von Klassen
2 Objektorientierung
    Grundlegende Begriffe, Entwicklungsgeschichte und Charakteristika von Java
3 Erste Schritte in Java
4 Grundlagen von Java
    Zeichensatz und Namenskonventionen, Kommentare und Javadoc
5 Datentypen, Variablen und Konstanten in Java
    Datentypen in Java, Variablen und Konstanten, Casts in Java
6 Anweisungen in Java
7 Ein- und Ausgabe in Java
    Eingabe über die Tastatur, Formatierte Ausgabe mit printf
8 Klassen und Objekte in Java
    Die Grundstruktur einer Klasse, Konstruktoren, this, static, enum
9 Arrays und Strings in Java

1. Motivation: Herleitung von Klassen

Beispiel: Brüche als benutzerdefinierter Datentyp

```
#include <stdio.h>

typedef struct Rational_ {
         int zaehler;
         int nenner;
} Rational;

int main() {
         Rational x1;
         printf("\nBitte Zähler eingeben: ");
         scanf("%i", &x1.zaehler);
         printf("\nBitte Nenner eingeben: ");
         scanf("%i", &x1.nenner);
         return 0;
}
```

1. Motivation: Herleitung von Klassen

Was finden wir gut an diesem Beispiel?
Brüche werden „als Ganzes" angesprochen: **`x1`** im Beispiel ist ein Bruch (Zähler und Nenner).

Womit sind wir noch nicht zufrieden?
Es fehlen Bruchoperationen.

Beispiel: Die Funktion *kehrwert*

```
#include <stdio.h>

typedef struct Rational_ {
        int zaehler;
        int nenner;
} Rational;

void kehrwert(Rational *x) {
        int tmp = x->zaehler;                // Abkürzung für *x.zaehler
        x->zaehler = x->nenner;              // Abkürzung für *x.nenner
        x->nenner = tmp;
}
```

1. Motivation: Herleitung von Klassen

Beispiel: Aufruf der Funktion *kehrwert*

```
int main() {
        Rational x1;
        printf("\nBitte Zähler eingeben: ");
        scanf("%i", &x1.zaehler);
        printf("\nBitte Nenner eingeben: ");
        scanf("%i", &x1.nenner);
        kehrwert(&x1);
        printf("\nKehrwert: %i / %i" , x1.zaehler, x1.nenner);
        return 0;
}
```

Welcher Zusammenhang besteht zwischen *kehrwert* und Brüchen?

- Bisher „prozedural gedacht“
- Die Funktion *kehrwert* manipuliert einen Bruch.
- Es besteht eine lose Kopplung über die Parameterliste der Funktion.

1. Motivation: Herleitung von Klassen

War das unsere Aufgabenstellung?

- Nein.
- Nicht „irgendeine“ Funktion schreiben, die den *kehrwert* berechnet, sondern diese Funktion sollte unmittelbar zu unseren Brüchen gehören!

→ Es wäre daher sinnvoller, wenn wir schon einen Bruch mit seinen Komponenten repräsentieren, auch die Funktion *kehrwert* zu einer Komponente zu machen.

Stellen wir also Brüche (Objekte) in den Fokus unseres Denkens und nicht die Funktionen, die Brüche manipulieren.

Definition

Ein abstrakter Datentyp (ADT) ist ein Tupel (Datentyp, Operationen), wobei die Operationen auf dem speziellen Datentyp definiert sind.

Beispiel: Abstrakter Datentyp für die Ganzzahlarithmetik

GanzzahlArithmetik = (Integer, {+, - , *, /})

1. Motivation: Herleitung von Klassen

Beispiel: Abstrakter Datentyp für die Datenstruktur *Schlange*

Schlange = (IntegerSequenz, {einfügenAmEnde, entfernenAmAnfang})

Abstrakte Datentypen lassen sich gut durch Klassen implementieren.

Definition
- Eine Klasse (class) besitzt statische und dynamische Komponenten.
- Die statischen Komponenten heißen Datenelemente**.**
- Die dynamischen Komponenten heißen Elementfunktionen (Memberfunktionen, Methoden) der Klasse.

Definition
- Ein Objekt (object) ist die Instanz einer Klasse.

In den verschiedenen Sprachen gibt es unterschiedliche Auffassungen darüber, was ein **Objekt** und was eine **Klasse** ist.

1. Motivation: Herleitung von Klassen

Beispiel: Was uns auch nicht gefällt an *Strukturen*

```
#include <stdio.h>

typedef struct Rational_ {
        int zaehler;
        int nenner;
} Rational;

int main() {
        Rational x1;
        printf("\nBitte Zähler eingeben: ");
        scanf("%i", &x1.zaehler);
        printf("\nBitte Nenner eingeben: ");
        scanf("%i", &x1.nenner);
        x1.nenner = 0;
        return 0;
}
```

1. Motivation: Herleitung von Klassen

Zugriffsrechte in Java
In Klassen gibt es zur Unterscheidung der Zugriffsrechte u.a. die Schlüsselworte public und private.

Klar ist, dass man zumindest zwei Konzepte benötigt:
- Methoden und
- Datenkapselung.

Datenkapselung
- Idee: Für den Anwender eines Objekts ist es unwichtig, wie dieses intern repräsentiert wird.
- Entscheidend ist nur, dass das Richtige gemacht wird.

Beispiel: *Schlange*
- Die Repräsentation der Daten darf keine Rolle spielen.
- *IntegerSequenz* vs. *IntegerArray*
- Dem Anwender ist es egal, wie die Implementierung aussieht: Array oder verkettete Liste

1. Motivation: Herleitung von Klassen

Beispiel Eine Klasse *Rational*

```
public class Rational {
        private int zaehler;
        private int nenner;

        public Rational(int z, int n) {
                zaehler = z; nenner = n;
        }

        public void kehrwert() {
                int tmp = zaehler;
                zaehler = nenner;
                nenner = tmp;
        }

        public void ausgeben() {
                System.out.println(zaehler + "/" + nenner);
        }
}
```

1. Motivation: Herleitung von Klassen

Beispiel: Eine Anwendung der Klasse *Rational*

```
public class Beispiel {
        public static void main(String[] args) {
                Rational r1 = new Rational(3, 4);
                r1.ausgeben();
                r1.kehrwert();
                r1.ausgeben();
        }
}
```

2. Objektorientierung

Grundlegende Begriffe

Vorteile der Objektorientierung

- Verkürzung der Entwicklungszeit
- Senkung der Fehlerrate
- Verbesserte Erweiterbarkeit und Anpassungsfähigkeit von Software
- Entwicklung wiederverwendbarer Softwarebibliotheken

Historie

- Grundkonzepte der Objektorientierung erstmals in den 60er Jahren: Programmiersprache *Simula67*.
- Größere Popularität erst ab 1980: *Smalltalk80* und später *C++*.
- Danach *Java, C#, Kotlin* usw.

2. Objektorientierung

Grundlegende Begriffe

Konzepte der Objektorientierung

- Datenkapselung
  - Festlegen der Funktionalität von Softwarekomponenten durch genau definierte **Schnittstellen.**
  - Diese **verbergen** die Details der Implementation.
- Vererbung
  - Vorhandene Komponenten können einfach modifiziert und erweitert werden.
  - Sie sind in anderen Kontexten wiederverwendbar.
- Polymorphie
  - Polymorphie wird durch das Konzept der späten (dynamischen) Bindung ermöglicht.
  - Da Objekte immer bestimmen, welche Methoden ausgeführt werden, und dies erst zur Laufzeit bekannt ist, können Methoden in abgeleiteten Klassen überschrieben werden.
  - Häufiger entsteht Polymorphie aber durch die Implementierung von Methoden aus Schnittstellen.
  - Dieses Prinzip ermöglicht die Entwicklung allgemein verwendbarer Softwarekomponenten.

2. Objektorientierung

Grundlegende Begriffe

Definition

- Ein Objekt ist eine Einheit von Daten und Funktionen, die auf den Daten operieren. Die Struktur der Daten und die Funktionen gleichartiger Objekte sind in ihrer gemeinsamen Klasse definiert.
- Ein Objekt wird auch als Instanz seiner Klasse bezeichnet. Jedes Objekt besitzt eine eigene Identität und einen eigenen Satz der Daten.

Definition

- Die Variablen eines Objekts heißen Instanzvariable oder Attribute. Sie legen den augenblicklichen Zustand des Objekts fest.
- Die Funktionen der Klasse heißen auch Methoden. Sie definieren das Verhalten der Objekte der Klasse.
- Die Identität eines Objekts äußert sich darin, dass jedes Objekt über einen eigenen Speicherplatz verfügt und sich damit unabhängig von anderen Objekten verändern kann.

- Prozedurale Programmierung basiert darauf, dass Funktionen auf den als Parameter mitgegebenen Daten operieren.
- Objektorientierte Programmierung basiert darauf, dass sich der Aufruf einer Methode immer an ein Objekt richtet. In der Klasse des Objekts ist festgelegt, wie auf eine Methode reagiert wird.

2. Objektorientierung

Grundlegende Begriffe

Definition

- Eine Klassendefinition beschreibt die Attribute (Instanzvariable) und die Methoden (Prozeduren, Funktionen) eines Objekts.
- Ein Konstruktor ist eine besondere Methode zur Festlegung der Anfangswerte der Attribute.

Definition

Jedes Objekt verfügt über einen eigenen Satz der in der Klasse definierten Instanzvariablen. Innerhalb der Funktionen einer Klasse werden diese Variablen und auch die Funktionen der Klasse einfach mit ihrem Namen angesprochen. Für die Variablen (und analog für die Methoden) fremder Objekte schreibt man Objektname.Variablenname.

Durch die Definition einer Klasse werden noch keine Objekte erzeugt. Dies geschieht erst durch die dynamische Erzeugung eines Objekts (new-Anweisung).

2. Objektorientierung

Grundlegende Begriffe

Objektorientierte Sprachen erlauben auch die **Zuweisung** (=) von Objekten. Unter den verschiedenen Programmiersprachen gibt es dabei zwei verschiedene Varianten:

- Wertsemantik
  Bei der Zuweisung wird der **Inhalt eines Objekts** in ein anderes Objekt **kopiert**. Nach der Zuweisung gibt es zwei verschiedene Objekte mit identischem Zustand.

- Referenzsemantik
  Wenn Variable nicht unmittelbar die Datenwerte eines Objekts sondern nur eine Referenz (Speicheradresse) des Objekts enthalten, wird bei der Zuweisung meist nur die **Objektreferenz kopiert**, so dass sich dann eventuell mehrere Variable auf das gleiche Objekt beziehen.

2. Objektorientierung

Grundlegende Begriffe

Garbage Collection

- Bei der Referenzsemantik ist zu beachten, dass nach der Zuweisung `r2 = r1`, das ursprünglich von `r2` referierte Objekt nicht mehr angesprochen werden kann. Der von diesem Objekt belegte Speicher kann daher für andere Objekte genutzt werden.
- Programmiersprachen mit Referenzsemantik verfügen oft über eine automatische Speicherverwaltung (garbage collection), die erkennt, welche Objekte nicht mehr ansprechbar sind. Bei Java ist dies beispielsweise der Fall. Die Alternative – eine programmierte Speicherfreigabe – vermeidet die damit verbundenen Laufzeitnachteile, ist in der Praxis jedoch auch Ursache für gravierende Programmierfehler.

Vererbung

- Es gibt oft Eigenschaften, die verschiedenartige Objekte gemeinsam haben.
- Im natürlichen Sprachgebrauch: *Oberbegriffe.*

Beispiel

- Ein schwarzer PKW Baujahr 2025 ist ein Auto, solange seine Details nicht von Interesse sind.
- Ein Autotransporter wirft ein ganz anderes Bild auf Autos.

2. Objektorientierung

Grundlegende Begriffe

Vererbung

- Verschiedene Klassen können in einer hierarchischen Beziehung zueinander stehen.
- Eine "abgeleitete" Klasse ist immer eine Spezialisierung einer Basisklasse.
- Umgekehrt ist die Basisklasse die Generalisierung der abgeleiteten Klasse.
- Alle Eigenschaften der Basisklasse werden jeweils übernommen (geerbt) und durch genauere Eigenschaften ergänzt.
- Es entsteht eine Klassenhierarchie.
- Grundsätzliches Kennzeichen der Vererbung ist dabei die sogenannte is-a-Relation.

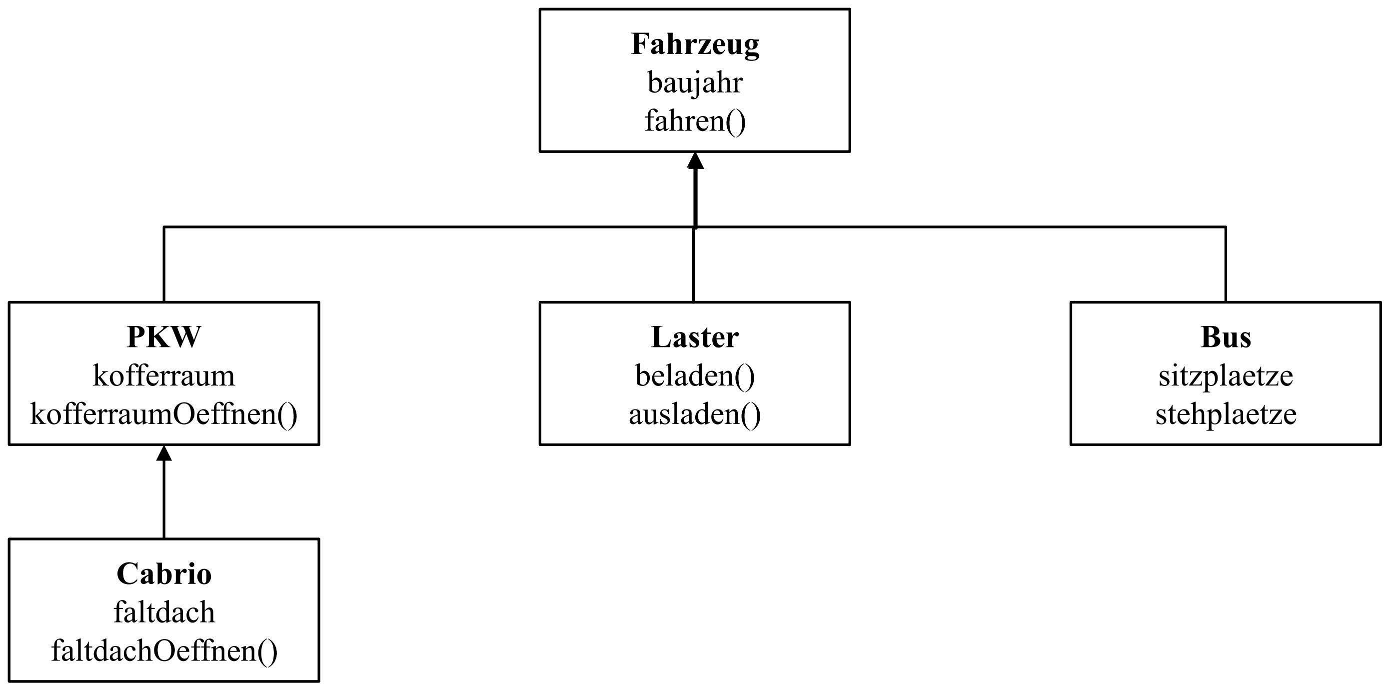
Vorteile der Vererbung

- Konsistenz und der Einsparung von Code.
- Gemeinsame Eigenschaften verschiedener Klassen müssen nur einmal implementiert und gegebenenfalls nur an einer Stelle geändert werden.

2. Objektorientierung

Grundlegende Begriffe

Beispiel: Vererbung



2. Objektorientierung

Entwicklungsgeschichte und Charakteristika von Java

Entwicklungsgeschichte von Java

- Objektorientierte und **plattformunabhängige** Programmiersprache
- Für alle Computersysteme verfügbar (i. Allg. kostenlos)
- 1991: Vorgängersprache **OAK (James Gosling, Bill Joy)**: nicht nur für Computer, sondern für Videorecorder und Settop-Boxen (interaktives Fernsehen)
- 1995: **Java** von SUN
- 1996: Allgemeine Anerkennung von Java wegen Integration in Web-Browsern und Network Computern
- Der Name Java bezeichnet einen Kaffee.
- 2010: Java von Oracle aufgekauft.

2. Objektorientierung

Entwicklungsgeschichte und Charakteristika von Java

Zitat Anfang:

- Java ist eine Technologie, mit der Sie Anwendungen entwickeln können, mit denen Sie mehr Spaß im Web haben und das Web nützlicher gestalten können. Java ist nicht zu verwechseln mit Javascript, einer einfachen Technologie, mit der Webseiten erstellt werden und die nur in Ihrem Browser ausgeführt wird.
- Mit Java können Sie Spiele spielen, Fotos hochladen, online chatten und virtuelle Ausflüge unternehmen. Außerdem können Sie Services nutzen wie Onlineschulung, Online-Banking und interaktive Karten. Wenn Sie Java nicht haben, funktionieren viele Anwendungen und Websites nicht.

Zitat Ende.

Quelle: https://www.java.com, 15.09.2025

2. Objektorientierung

Entwicklungsgeschichte und Charakteristika von Java

Charakteristika von Java

- Java erzeugt einen **architekturneutralen Bytecode.**
- Jeder Programmcode muss in eine **Klasse** (Objektorientierung)
- Arten von Java-Klassen:
  - **Applikationen** (eigenständige Anwendungen mit main-Methode, wie C-Programme)
  - **Bibliotheks-Klassen**: eigenständige Objekte
  - **Applets**:
    - Werden innerhalb eines Internet-Browsers aufgerufen
    - Werden von einem Server geladen
    - Haben ein Sicherheitskonzept (**sand box**), können z.B. nicht auf lokale Dateien zugreifen
  - **Servlets**: Erweitern die Funktionalität eines Web Servers

Java Devolpment Kit (JDK) besteht aus

- Java-Compiler
- Java Laufzeitumgebung (Java Virtual Machine)
- Appletviewer
- Java-Debugger

2. Objektorientierung

Entwicklungsgeschichte und Charakteristika von Java

Online-Dokumentation von Java

- Beschreibt das **Application Programming Interface (API)**
- Liegt als HTML-Dateien mit Hypertext-Links vor (Browser)
- Dient zum „Lesen“ von Klassen

Versionen von Java

- JDK 1.0 (1996)
- …
- J2SE 1.3 (2000) SE = Standard Edition
- …
- J2SE 5.0 (2004)
- JSE 6 (2013)
- …
- JSE 9 (2017)
- JSE 10 (2018)
- …
- Java 18 (März 2022)

3. Erste Schritte in Java

Beispiel : „Hello World" - Programm in Java

```
public class HelloWorld {
        public static void main(String[] args) {
                System.out.println("Hello World!");
        }
}
```

Übersetzen und Ausführen des Java-Programms

```
        javac HelloWorld.java     → erzeugt HelloWorld.class
        java HelloWorld           → erzeugt die Ausgabe auf dem Bildschirm
```

3. Erste Schritte in Java

Anmerkungen zum Aufbau eines Java-Programms

- Jedes Java-Source-File darf nur eine public Klasse enthalten.
- Der Filename muss der Name dieser Klasse (mit der richtigen Groß-Kleinschreibung) mit der Extension java sein, hat also die Form Xxxxx.java.
- Es können auch mehrere Files gleichzeitig compiliert werden: javac Xxxxx.java Yyyyy.java
- Der Java-Compiler erzeugt für jede Klasse ein eigenes File, das den **Bytecode** dieser Klasse enthält. Der Filename ist dann der Name der Klasse mit der Extension .class, hat also die Form Xxxxx.class. Dieser Bytecode ist **plattformunabhängig**: Egal auf welchem System der Java-Compiler aufgerufen wurde, der Bytecode kann auch auf jedem anderen Computersystem ausgeführt werden.
- Falls die Klasse andere Klassen verwendet, die neu übersetzt werden müssen, werden diese automatisch ebenfalls übersetzt.
- Die Ausführung des Bytecodes einer Java-Applikation erfolgt durch Aufruf der Java Virtual Machine JVM in der Form java Xxxxx.

4. Grundlagen von Java

Zeichensatz und Namenskonventionen

Zeichensatz in Java

- Der Unicode-Zeichensatz (**`http://www.unicode.org`**) ist geeignet, sehr viele Zeichen darzustellen.
- Unicode ist ein 16-Bit-Code, d.h. ein Zeichen besteht aus 2 Byte.
- Die ersten 128 Zeichen sind die Zeichen des 7-Bit ASCII-Zeichensatzes.
- Ist ein gewünschtes Zeichen nicht auf der Tastatur vorhanden, so kann dieses Zeichen durch eine Ersatzdarstellung **`\uabcd`** dargestellt werden.
- Dies entspricht dem Unicode-Zeichen an der Position **`a*163 + b*162 + c*161 + d*160`**.

4. Grundlagen von Java

Zeichensatz und Namenskonventionen

Namenskonventionen in Java

- Namen von Klassen beginnen mit einem Großbuchstaben und bestehen aus Groß- und Kleinbuchstaben und Ziffern.
  Beispiele: `Rational`, `Button2`

- Namen von Konstanten beginnen mit einem Großbuchstaben und bestehen nur aus Großbuchstaben, Ziffern und Underlines.
  Beispiele: `PI`, `X_2`

- Alle anderen Namen beginnen mit einem Kleinbuchstaben und bestehen aus Groß- und Kleinbuchstaben und Ziffern.
  Beispiele: `openFile`, `groesse`, `setGroesse`, `getGroesse`

- Groß- und Kleinbuchstaben haben verschiedene Bedeutung (case-sensitive).

- Reservierte Wörter dürfen nicht neu definiert werden.

4. Grundlagen von Java

Kommentare und Javadoc

Kommentare in Java

- Kommentarblock wie in C
  `/*                   */`

- Zeilenkommentar: wie in C99, alle Zeichen von `//` bis zum Zeilenende werden vom Compiler ignoriert.
  `//`

- Dokumentationskommentar (Javadoc)

Beispiel: Zeilenkommentar

```
// Flächenberechnung
int a;   // a ist die erste Seite des Rechtecks.
```

Quelle: Die Ideen zu Kommentaren und javadoc stammen aus: Goll, J., Heinisch, C., Java als erste Programmiersprache: Grundkurs für Hochschulen, Springer Vieweg, 2016

4. Grundlagen von Java

Kommentare und Javadoc

Dokumentationskommentare in Java

- Das Werkzeug javadoc filtert alle Informationen innerhalb der Kennzeichnung `/**` und `*/` heraus.
- Diese Informationen werden in HTML-Dateien abgelegt.
- Dadurch kann automatisch eine Dokumentation erzeugt werden (**Browserfähig**).

Sinn von Dokumentationskommentaren

- Nicht nur das Schreiben von Code ist die einzige wichtige Aktivität.
- Großes Problem: Die Pflege der Dokumentation.
- Sind Dokumentation und Code **getrennt**, muss bei jeder Code-Änderung die Dokumentation geändert werden.
- Die Lösung: Man verknüpft den Code mit der Dokumentation (alles steht in der gleichen Datei).
- Das Tool **javadoc** sucht **Kommentar-Tags.**
- Es extrahiert nicht nur die mit diesen Tags gekennzeichneten Informationen, sondern zieht auch den Klassen- oder den Methodennamen heraus, der an den Kommentar angrenzt.
- Der Output von *javadoc* ist eine HTML-Datei, die man mit einem Browser ansehen kann.

4. Grundlagen von Java

Kommentare und Javadoc

Dokumentationskommentare in Java

- Syntax: Alle *javadoc*-Kommandos treten nur zwischen **/**** und ***/** - Kommentaren auf.
- Es gibt drei Typen von Dokumentationskommentaren:
    - für Klassen,
    - für Variable,
    - für Methoden.
- Ein Klassenkommentar erscheint **genau vor** der Definition einer Klasse,
- Ein Variablenkommentar **genau vor** der Definition einer Variablen und
- Ein Methodenkommentar **genau vor** der Definition einer Methode.

Beispiel:

```
/** Ein Klassenkommentar. */
public class Beispielklasse {
        /** Ein Variablenkommentar. */
        private int x;
        /** Ein Methodenkommentar. */
        public int beispielmethode() {}
}
```

4. Grundlagen von Java

Kommentare und Javadoc

Dokumentations-Tags in Java

- Alle drei Kommentardokumentationstypen können **@see-Tags** enthalten, die es erlauben, sich auf die Dokumentation in anderen Klassen zu beziehen. Javadoc generiert dabei HTML, wo die @see-Tags per Hyperlink zur anderen Dokumentation verweisen. Die Formen sind

- Außerdem: Tags für die Versions-Information und den Autorennamen
  - @version Versionsinformation
  - @author Autoren-Information

Methodendokumentations-Tags

- @param Parameter-Name Beschreibung
- @return Beschreibung
- @throws voll-qualifizierter-Klassenname Beschreibung

- **Regel:** Es ist guter Stil, nur zu dokumentieren, was man wissen sollte. Wenn es keine Rückgabe gibt, schreibt man kein *return*-Tag, denn das sieht man ja schon an *void* in der Signatur. Außerdem enden Sätze in javadoc mit einem Punkt.

4. Grundlagen von Java

Kommentare und Javadoc

Beispiel: Dokumentationskommentare

```
import java.util.Scanner;

/** Ein einfaches Programm mit Dokumentationskommentaren.
 * @author Frank Victor.
 * @author http://www.th-koeln.de/~victor.
 * @version 2.0.
 */

public class Primzahl {
        private static Scanner in;
```

Quelle: Goll, J., Heinisch, C., Java als erste Programmiersprache: Grundkurs für Hochschulen, Springer Vieweg, 2016

4. Grundlagen von Java

Kommentare und Javadoc

```
/** Es wird getestet, ob n eine Primzahl ist.
* Ist n <= 0, wird eine Exception erzeugt.
* @param n Zu testende Zahl.
* @return true, falls n eine Primzahl ist und false, falls nicht.
* @throws ArithmeticException, falls n <= 0.
*/
public static boolean isPrim(int n) {
        if (n <= 0)
                throw new ArithmeticException("isPrim: Parameter <= 0");
        if (n == 1) return false;
        for (int i = 2; i <= n/2; ++i)
                if (n % i == 0) return false;
        return true;
}
```

4. Grundlagen von Java

Kommentare und Javadoc

```
        public static void main(String[] args) {
                in = new Scanner(System.in);
                System.out.print("Zahl: ");
                int a = in.nextInt();
                System.out.printf("%4d ist eine Primzahl: %b %n", a, isPrim(a));
        }
}
```

Die Dokumentation erzeugt man mit

```
advm1> javadoc Primzahl.java -author -version
```

und erhält die Datei **`Primzahl.html`**, die so aussieht:

Directory auf der advm1:
/home/victor/Beispiele_Java/javadocBeispiel

find . -type d listet alle Directories

Quelle: Goll, J., Heinisch, C., Java als erste Programmiersprache: Grundkurs für Hochschulen, Springer Vieweg, 2016

4. Grundlagen von Java

Kommentare und Javadoc

java.lang
Class Primzahl

```
java.lang.Object
  └─ Primzahl
```

```
public class Primzahl
extends java.lang.Object
```

Ein einfaches Programm mit Dokumentationskommentaren.

Version:
2.0
Author:
Frank Victor, http://www.gm.fh-koeln.de/~victor

Constructor Summary
Primzahl()

Method Summary	
static boolean	**isPrim**(int n) Es wird getestet, ob es n eine Primzahl ist.
static void	**main**(java.lang.String[] args)

4. Grundlagen von Java

Kommentare und Javadoc

Constructor Detail

Primzahl

`public Primzahl()`

Method Detail

isPrim

`public static boolean isPrim(int n)`

Es wird getestet, ob n eine Primzahl ist. Ist n <= 0, wird eine Exception erzeugt.

Parameters:

`n` - Zu testende Zahl

Returns:

true, falls n eine Primzahl ist und false, falls nicht.

Throws:

`java.lang.ArithmeticException` - wenn n <= 0

---

main

`public static void main(java.lang.String[] args)`

---

5. Datentypen, Variablen und Konstanten in Java

Datentypen

Die Java-Datentypen zerfallen in die beiden Gruppen Wertdatentyp (einfacher, elementarer Typ) und Referenzdatentyp.

Wertdatentypen sind:

- Numerische Typen
  - Integer-Typen: `char`, `byte`, `short`, `int`, `long`
  - Fließkomma-Typen: `float`, `double`
- Boolescher Typ: `boolean`

Referenzdatentypen sind:

- Klassen und
- Arrays

- Bei **Wertdatentypen** sagt der Typ u.a. aus, in welcher Darstellung die Daten gespeichert werden und welcher Speicherplatzbedarf mit einer Variablen verbunden ist.
- Bei **Referenzdatentypen** sagt der Typ nichts über die Darstellung und über den Speicherplatzbedarf aus. Der Typ sagt nur aus, welche Operationen man mit Sicherheit ausführen kann.

5. Datentypen, Variablen und Konstanten in Java

Datentypen

Wertdatentypen in Java

Datentyp	Größe und Bedeutung	Wertebereich
char	2 Byte, Unicode Zeichen	Eines der 65536 Unicode-Zeichen
byte	1 Byte , Ganzzahl mit Vorzeichen	-2^{7} bis $+2^{7}$ -1
short	2 Byte, Ganzzahl mit Vorzeichen	-2^{15} bis $+2^{15}$ -1
int	4 Byte, Ganzzahl mit Vorzeichen	-2^{31} bis $+2^{31}$ -1
long	8 Byte, Ganzzahl mit Vorzeichen	-2^{63} bis $+2^{63}$ -1
float	4 Byte, Fließkommazahl, einfache Genauigkeit	$-3.4 * 10^{38}$ bis $3.4 * 10^{38}$
double	8 Byte Fließkommazahl, doppelte Genauigkeit	$-1.7 * 10^{308}$ bis $1.7 * 10^{308}$
boolean	1 Byte, Wahrheitswert	true, false

Unterschiede zu C

- Ein Zeichen wird mit 2 Byte dargestellt
- Typen sind auf allen Rechnern gleich.
- Es gibt kein `unsigned`.

5. Datentypen, Variablen und Konstanten in Java

Variablen in Java

Unterschiede zu C

- Gemäß den beiden möglichen Datentypen Werttyp und Referenztyp gibt es auch zwei verschiedene Arten von Variablen: Wertvariablen und Referenzvariablen.
- In Java sind die Adressen von Speicherzellen nicht sichtbar. → Es gibt keinen Address-of-Operator (&).
- Objekte werden als Objekte selbst im Speicher abgelegt und nicht Byte-orientiert wie in C.
  → Es gibt kein **`sizeof`**.
- In Java werden boolesche Konstanten durch **`true`** und **`false`** ausgedrückt.
- Zeichenkonstanten sind in Java Konstanten vom Typ **`char`** und nicht vom Typ **`int`** wie in C.
- Zeichenketten werden wie in C in doppelte Hochkommata eingeschlossen. Es gibt den Konkatenationsoperator (+).

Beispiel:

`"frank " + "victor"` ergibt **`"frank victor"`**.

5. Datentypen, Variablen und Konstanten in Java

Konstanten in Java

- In Java gibt es wie in C zwei Arten von Konstanten: **Literale** und **Symbolische Konstanten**.
- **Symbolische Konstanten** haben einen Namen, der ihren Wert repräsentiert und dürfen nach ihrer Initialisierung nicht mehr verändert werden.
- In Java wird anstatt von *const* in C das Schlüsselwort final verwendet.
- Symbolische Konstanten sollten in Java als static final definiert werden.
- Einfache *final* Variable sind fast immer Instanzvariable (Wert wird durch den Konstruktor festgelegt) und in seltenen Fällen verwendet man *final* auch für lokale Variable.

Beispiel:

```
static final float PI = 3.1415;
```

5. Datentypen, Variablen und Konstanten in Java

Casts in Java

Es gibt in Java zwei grundverschiedene Casts, die außer der Schreibweise nichts gemein haben.

(1) Casts für numerische Daten:
Bei numerischen Daten **`(byte, char, short, int, long, double, float)`** dient der Cast dazu, eine Umwandlung der Zahlendarstellung zu veranlassen. Der Cast wandelt um, es kommt niemals zu einem Laufzeitfehler. In Java und C gibt es neben den durch Cast angegebenen expliziten auch implizite Typumwandlung (in Java nur von eingeschränkten zu umfassenderen Darstellungen).

(2) Casts für Referenzdaten:
Bei Referenzdaten veranlasst der Cast eine **Typprüfung**, die zur Laufzeit durchgeführt wird. Schlägt die Prüfung fehl, gibt es eine *ClassCastException*, sonst passiert nichts. Dem Compiler gibt die Prüfung die Sicherheit, dass er für den Cast-Ausdruck den angegebenen Typ annehmen kann. **Objekte werden niemals (weder automatisch noch explizit) umgewandelt!** Es gibt keine automatische Typanpassung bei Referenzdaten! Die Zuweisung zu einer Variablen eines Obertyps erfordert keinerlei Maßnahme.

Außer diesen beiden Fällen gibt es keine weiteren Casts in Java.

6. Anweisungen in Java

Die Anweisungen in Java sind die gleichen wie in C mit folgenden Besonderheiten:

- Deklarationen und Anweisungen dürfen beliebig gemischt werden.

- Java hat keinen Präprozessor und deshalb gibt es keine Präprozessor-Direktiven. Wenn Sie Klassen aus einer anderen Bibliothek benutzen wollen, schreiben Sie **`import`** und den Name der Bibliothek. Es ist hier wichtig zu betonen, dass **`import`** nichts mit dem **`include`** in C zu tun hat, das Sie schon kennen. **`import`** bindet nichts ein, sondern führt lediglich eine Abkürzung ein.

- Die *for*-Anweisung erlaubt die Deklaration von lokalen Variablen:
  ```
  for(int i=1; i < 100; i++) ...
  ```

- Für Arrays gibt es in Java die komfortable *for-each-Schleife*. Diese behandeln wir später im Zusammenhang mit Arrays in Java.

- Es gibt kein *goto*.

- Es gibt *break* / *continue*-Labels: **`break label_1;`** unterbricht wie in C und setzt die Ausführung bei der Marke **`label_1`** fort.

7. Ein- und Ausgabe in Java

Eingabe über die Tastatur

Seit Java 5 gibt es die neue Klasse **`java.util.Scanner`**. Damit ist die Eingabe sehr viel einfacher geworden als in den Vorgängerversionen.

Die Klasse **`java.util.Scanner`** ermöglicht, ein Objekt zu erzeugen, das über Methoden, wie **`next`**, **`nextInt`** oder **`nextDouble`** (und viele andere) verfügt.

Quelle: Goll, J., Heinisch, C., Java als erste Programmiersprache: Grundkurs für Hochschulen, Springer Vieweg, 2016

7. Ein- und Ausgabe in Java

Eingabe über die Tastatur

Beispiel:

```
import java.util.Scanner;

public class Addieren {
        private static Scanner in;          // dann kann man den Scanner "in"
                                            // auch in anderen Methoden nutzen

        public static void main(String[] args) {
                in = new Scanner(System.in);

                System.out.print("1. Zahl: ");
                double a = in.nextDouble();
                System.out.print("2. Zahl: ");
                double b = in.nextDouble();
                System.out.printf("%.4f + %.4f = %6.4f%n", a, b, a + b);
        }
}
```

7. Ein- und Ausgabe in Java

Eingabe über die Tastatur

Beispiel (Fortsetzung):

Die Ausgabe sieht dann so aus:

```
advm1> java Addieren
1. Zahl: 2.2
2. Zahl: 3.3
2.2000 + 3.3000 = 5.5000
advm1>
```

Lokalisierung

- Es gibt **unterschiedliche Notationen für Dezimalzahlen in verschiedenen Ländern**.
- Das obige Programm haben wir auf der **`advm1`** ausgeführt und das ist eine IBM UNIX-Maschine. Die hat standardmäßig die Einstellung **`US`**. Aus diesem Grund haben wir die Dezimalzahlen mit Punkt eingegeben.
- Die **Lokalisierung** auf Computern ist also vorgegeben. Bei Ihrem PC wird dies wahrscheinlich **`GERMANY`** sein und so müssen Sie, wenn Sie das obige Programm dort ausprobieren wollen, die Dezimalzahlen mit Komma eingeben.
- Java schaut – bevor Eingaben und Ausgaben gemacht werden – nach der Lokalisierung des Rechners.

7. Ein- und Ausgabe in Java

Eingabe über die Tastatur

Beispiel (Fortsetzung):

Eingabe der Dezimalzahlen mit Komma auf der advm1

```
advm1> java Addieren
1. Zahl: 2,2
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:845)
        at java.util.Scanner.next(Scanner.java:1458)
        at java.util.Scanner.nextDouble(Scanner.java:2362)
        at Adder.main(Addieren.java:11)
advm1>
```

Im Folgenden zeigen wir, wie man die Lokalisierung in Java überschreibt.

Quelle: Die Ideen zu Kommentaren und javadoc stammen aus: Goll, J., Heinisch, C., Java als erste Programmiersprache: Grundkurs für Hochschulen, Springer Vieweg, 2016

7. Ein- und Ausgabe in Java

Eingabe über die Tastatur

Beispiel: Überschreiben der Lokalisierung

```
import java.util.Locale;
import java.util.Scanner;

public class AddierenGermany {
        private static Scanner in;          // dann kann man den Scanner "in"
                                            // auch in anderen Methoden nutzen

        public static void main(String[] args) {
                Locale.setDefault(Locale.GERMANY);
                in = new Scanner(System.in);
                System.out.print("1. Zahl: ");
                double a = in.nextDouble();
                System.out.print("2. Zahl: ");
                double b = in.nextDouble();
                System.out.printf("%.4f + %.4f = %6.4f%n", a, b, a + b);
        }
}
```

7. Ein- und Ausgabe in Java

Eingabe über die Tastatur

Die Ausgabe sieht dann so aus:

```
advm1> java AddierenGermany
1. Zahl: 2,2
2. Zahl: 3,3
2,2000 + 3,3000 = 5,5000
advm1>
```

und dann schließlich als Negativ-Test:

```
advm1> java AddierenGermany
1. Zahl: 2.2
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:845)
        at java.util.Scanner.next(Scanner.java:1458)
        at java.util.Scanner.nextDouble(Scanner.java:2362)
        at AddierenGermany.main(AddierenGermany.java:13)
advm1>
```

7. Ein- und Ausgabe in Java

Eingabe über die Tastatur

Siehe: → **https://docs.oracle.com/javase/7/docs/api/**

java.util
Class Scanner
java.lang.Object
        java.util.Scanner

A simple text scanner which can parse primitive types and strings using regular expressions.

A Scanner breaks its input into tokens using a delimiter pattern, which by default matches whitespace. The resulting tokens may then be converted into values of different types using the various next methods.

For example, this code allows a user to read a number from System.in:

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
```

7. Ein- und Ausgabe in Java

Eingabe über die Tastatur

Methoden:

- **`next()`** → liest den nächsten String
- **`nextLine()`** → liest die ganze Zeile
- **`nextBoolean()`** → liest den nächsten booleschen Wert (**`true`**, **`false`**)
- **`nextInt()`** → liest den nächsten Integer
- **`nextLong()`** → liest den nächsten Long Integer
- **`nextFloat()`** → liest den nächsten Float
- **`nextDouble()`** → liest den nächsten Double

Anmerkung zum Begriff API (vgl. [http://de.wikipedia.org/wiki/Programmierschnittstelle])
„Eine *Programmierschnittstelle* ist eine Schnittstelle, die von einem Softwaresystem anderen Programmen zur Anbindung an das System zur Verfügung gestellt wird. Oft wird dafür die Abkürzung *API* (für engl. *application programming interface*, deutsch: *Schnittstelle zur Anwendungsprogrammierung*) verwendet.“

7. Ein- und Ausgabe in Java

Eingabe über die Tastatur

Beispiel: Verwendung verschiedener Scanner Methoden

```
import java.util.Scanner;

public class Einmaleins {
        private static Scanner in;

        public static void main(String[] args) {
                in = new Scanner(System.in);
                meetAndGreet();
                mitInt();
                mitDouble();
        }

        public static void meetAndGreet() {        // in.next liest einen String.
                System.out.print("Hallo, wie ist dein Name: ");
                String name = in.next();
                System.out.printf("Hallo %s, fangen wir an!%n", name);
}
```

7. Ein- und Ausgabe in Java

Eingabe über die Tastatur

Beispiel (Fortsetzung):

```
        public static void mitInt() {
                System.out.println("Einmaleins mit Ganzen Zahlen");
                System.out.print("Ganze Zahl eingeben: ");
                int zahl = in.nextInt();
                for (int i = 1; i <= 10; i++)
                        System.out.printf("%2d * %d = %3d%n", i, zahl, zahl * i);
                }
        }

        public static void mitDouble() {
                System.out.println("Einmaleins mit Gleitkommazahlen");
                System.out.print("Gleitkommazahl eingeben: ");
                double zahl = in.nextDouble();
                for (int i = 1; i <= 10; i++) {
                        System.out.printf("%2d * %.2f = %6.2f%n", i, zahl, zahl * i);
                }
        }
}
```

7. Ein- und Ausgabe in Java

Formatierte Ausgabe mit printf

- Die Ausgabemethoden **`System.out.print`** und **`System.out.println`** haben wir immer wieder verwendet. Sie existieren schon seit JDK 1.0 und sind einfach zu benutzen.

- Die einzige Schwäche von **`System.out.print`** und **`System.out.println`** ist, dass die Formatierung der Ausgabe nicht beeinflussbar ist − wie wir es von **`printf`** in C kennen.

- In Java 5 ist die Klasse um ein **`printf`** erweitert worden, das (fast) genauso funktioniert wie in C. Mit **`printf`** ist die Formatierung der Ausgabe nun möglich.

- Details findet man in der API:
  Siehe: → **`https://docs.oracle.com/javase/7/docs/api/`**

Anmerkung: Benutzen Sie statt **`\n`** in Java **`%n !`**

7. Ein- und Ausgabe in Java

Formatierte Ausgabe mit printf

Beispiel: Verwendung von `printf`

```
import java.util.Scanner;

public class Rechteck {
        private static Scanner in;

        public static void main (String[] args) {
        double a = 0.0, b = 0.0, u, fl;

        in = new Scanner(System.in);

        System.out.println("Programm für Rechteckberechnungen");
        System.out.print("Bitte geben Sie die 1. Seite in cm ein: ");
        a = in.nextDouble();

        System.out.print("Bitte geben Sie die 2. Seite in cm ein: ");
        b = in.nextDouble();
```

7. Ein- und Ausgabe in Java

Formatierte Ausgabe mit printf

Beispiel (Fortsetzung):

```
        u = 2.0 * a + 2.0 * b;
        fl = a * b;

        System.out.printf("Die Seite a ist %8.2f cm lang.%n",a);
        System.out.printf("Die Seite b ist %8.2f cm lang.%n",b);
        System.out.printf("Der Umfang des Rechtecks beträgt %8.2f cm.%n",u);
        System.out.printf("Die Fläche des Rechtecks beträgt %8.2f cm².%n",fl);
        }
}
```

Hinweis:
Java hat `\n` als newline in C verdrängt. `\n` funktioniert zwar auf UNIX Systemen, nicht aber auf anderen Plattformen wie Windows. Man sollte daher immer `%n` verwenden. Damit wird der richtige Separator der jeweiligen Plattform verwendet (Cross Plattform Fähigkeit).

8. Klassen und Objekte in Java

Die Grundstruktur einer Klasse in Java

Unterschied zwischen Klasse und Objekt

- Ein Objekt ist eine Einheit von Daten und Funktionen, die auf den Daten operieren.
- Die Struktur der Daten und die Funktionen gleichartiger Objekte sind in der Klasse definiert.
- Ein Objekt wird auch als **Instanz seiner Klasse** bezeichnet.
- Jedes Objekt besitzt eine eigene Identität und einen eigenen Satz der Daten.
- Die Variablen eines Objekts (Instanzvariable) legen den augenblicklichen Zustand des Objekts fest.
- Die Funktionen der Klasse heißen Methoden. Sie definieren das Verhalten der Objekte der Klasse.
- Die Identität eines Objekts äußert sich darin, dass jedes Objekt über einen eigenen Speicherplatz verfügt

8. Klassen und Objekte in Java

Die Grundstruktur einer Klasse in Java

Definition einer Klasse (class) in Java

```
class ClassName{
        komponenten
}
```

- Mit *ClassName* wird der Name für die Klasse vereinbart.
- Nach allgemeinen Konventionen sollte *ClassName* groß geschrieben werden.
- Die *komponenten* sind entweder Instanzvariablen (Datenelemente, Attribute) oder Methoden.
- Nach allgemeinen Konventionen sollten Methoden klein geschrieben werden.
- Bei zusammengesetzten Namen beginnt jedes Wort bis auf das erste mit einem Großbuchstaben.
- Die Methoden eines Objekts haben Zugriff auf die Datenfelder und Methoden desselben Objekts.
- Den Komponenten einer Klasse können **Modifikatoren** vorangestellt werden **`public`**, **`private`**, **`protected`**, **`static`**, **`final`**, **`volatile`**, **`abstract`** usw.
- Ohne Modifikator wird eine Komponente auf *friendly* gesetzt, d.h. der Zugriff ist öffentlich (Paketweise).

8. Klassen und Objekte in Java

Die Grundstruktur einer Klasse in Java

Modifikatoren

- Eine private-Komponente gewährleistet, dass sie nur von den Komponenten innerhalb der selben Klasse zugreifbar ist.
- public öffnet die Komponente auch für Zugriffe von außen.
- public-Komponenten heißen öffentliche Komponenten und private-Komponenten private Komponenten.

Was ist privat?
Man wird im Normalfall die Datenelemente einer Klasse als *private* deklarieren, da man nicht möchte, dass sie von außen ohne den Aufruf einer Elementfunktion geändert werden können (→ Geheimnisprinzip der objektorientierten Programmierung).

Was ist public?
Elementfunktionen wird man, sofern es sich nicht um klasseninterne Hilfsfunktionen handelt, normalerweise als *public* deklarieren, da man den Aufruf dieser Funktionen auch Objekten außerhalb der Klasse ermöglichen möchte (→ Schnittstelle einer Klasse für ihre Außenwelt).

8. Klassen und Objekte in Java

Die Grundstruktur einer Klasse in Java

Beispiel: Entwurf einer Klasse *Rational*

Zwei Fragen sind zu beantworten
(1) Was soll man mit Objekten der Klasse machen können? ==> *Schnittstelle*
(2) Wie ist der interne Aufbau dieser Objekte?

Anforderungen an die Klasse *Rational*
(1) Wir wollen wir nur fordern, dass Brüche ausgegeben werden können. Später kommen weitere Funktionen hinzu.
(2) Ein Bruch besteht aus Zähler und Nenner, die beide vom Typ *int* sein sollen. Datenkapselung → diese beiden Komponenten sind private.

Dies führt uns zur folgenden Klassendefinition.

Quelle: Die Idee zur Konstruktion der Klasse *Rational* stammt aus: Josuttis, N., Objektorientiertes Programmieren in C++: Von der Klasse zur Klassenbibliothek, Addison-Wesley1996

8. Klassen und Objekte in Java

Die Grundstruktur einer Klasse in Java

Beispiel: Die Klasse *Rational*

```
public class Rational {

        // Instanzvariablen
        private int zaehler;
        private int nenner;

        // Methode
        public void ausgeben() {
                System.out.println(zaehler + " / " + nenner);
        }
}
```

Diskussionspunkte

- Die Methode *ausgeben()* hat den Rückgabetyp *void*.
- Ein Bruch weiß selbst, in welcher Form er sich auszugeben hat.
- Die Methode *ausgeben()* hat keinen Parameter (insbesondere keinen Bruch!)
- Die Komponenten *zaehler* und *nenner* sind nur in dieser Klasse zugreifbar.

8. Klassen und Objekte in Java

Die Grundstruktur einer Klasse in Java

Erzeugen von Referenzen **und** Objekten

`ClassName` sei eine Klasse.

Dann **definiert** man eine **Referenz** auf ein Objekt dieser Klasse durch

```
ClassName refName;
```

oder mehrere Referenzen aufzählend durch

```
ClassName refName-1, ..., refName-n;
```

Ein Objekt wird in Java **erzeugt** durch

```
new ClassName();
```

8. Klassen und Objekte in Java

Die Grundstruktur einer Klasse in Java

Aufruf von Methoden für Objekte

`ClassName` sei eine Klasse, die die Methode **`methName()`** enthält. Dann erzeugt

```
ClassName refName = new ClassName();
```

die Referenz *refName*, die auf ein Objekt der Klasse *ClassName* zeigt.

Der Aufruf der Methode *methName* hat die Form

```
refName.methName();
```

Durch die Aufrufsyntax bezieht sich also die Methode auf ein Objekt. Im objektorientierten Sprachgebrauch sagt man, dass dem Objekt die Nachricht *methName* ohne Parameter geschickt wird.

8. Klassen und Objekte in Java

Die Grundstruktur einer Klasse in Java

Initialisierung von Komponenten in Klassen

- Wenn ein primitiver Datentyp eine Komponente einer Klasse ist, erhält er garantiert einen *Defaultwert*, falls man ihn nicht initialisiert:
  - **`char`**: ' \u0000‘ (nicht druckbar)
  - **`byte`**, **`short`**, **`int`**, **`long`**: 0
  - **`float`**, **`double`**: 0.0
  - **`boolean`**: false

- Dies stellt sicher, dass die primitiven Typen immer initialisiert werden, was eine Fehlerquelle eingrenzt.

- **Java meldet einen Fehler zur Kompilierzeit (sicherer als C).**

- Diese Garantie gilt jedoch **nicht für lokale Variablen** - diese sind keine Komponenten einer Klasse.

Quelle: Die Idee zur Konstruktion der Klasse *Rational* stammt aus: Josuttis, N., Objektorientiertes Programmieren in C++: Von der Klasse zur Klassenbibliothek, Addison-Wesley1996

8. Klassen und Objekte in Java

Konstruktoren

Sinn von Konstruktoren

- Häufig wird vom Programmierer vergessen, Objekte zu initialisieren.
- Konstruktoren haben die Aufgabe, Objekte zu initialisieren, wenn sie erzeugt werden.

- Ein Konstruktor für eine Klasse ist eine spezielle Funktion, deren Name gleich dem Klassennamen ist.
- Konstruktoren besitzen keinen Rückgabetyp (auch nicht *void*).
- Ein Konstruktor wird automatisch aufgerufen, wenn ein Objekt der Klasse angelegt wird.
- Die Anweisungen im Konstruktor werden unmittelbar nach dem Erzeugen des Objekts ausgeführt.
- Konstruktoren können Parameter besitzen.

- Wird kein Parameter angegeben, heißt der Konstruktor Default-Konstruktor**.** Wenn kein Konstruktor definiert wurde, erzeugt der Compiler einen parameterlosen voreingestellten Default-Konstruktor.

8. Klassen und Objekte in Java

Konstruktoren

Beispiel: Konstruktoren für die Klasse *Rational*

```
public class Rational {
        private int zaehler;
        private int nenner;

        public Rational() {                     // Default-Konstruktor
                zaehler = 0; nenner = 1;
        }
        public Rational(int z) {                // int-Konstruktor
                zaehler = z; nenner = 1;
        }
        public Rational(int z, int n) {         // int-int-Konstruktor
                zaehler = z; nenner = n;
        }
        public void ausgeben() {
                System.out.println(zaehler + " / " + nenner);
        }
}
```

8. Klassen und Objekte in Java

Konstruktoren

Beispiel: Verwendung der Klasse *Rational* in der Klasse *BeispielAnwendung*

```
public class BeispielAnwendung {
        public static void main(String[] args) {
                Rational r1 = new Rational();
                Rational r2 = new Rational(3);
                Rational r3 = new Rational(3, 4);

                r1.ausgeben();
                r2.ausgeben();
                r3.ausgeben();
        }
}
```

Die Ausgabe lautet:

```
0/1
3/1
3/4
```

8. Klassen und Objekte in Java

Weitere Methoden für die Klasse Rational

Beispiel:

- **`addiere`** zur Addition zweier Brüche
- **`multipliziere`** zur Multiplikation zweier Brüche
- **`kleinerAls`** zum Vergleich zweier Brüche auf „kleiner als“

```
public class Rational {
        private int zaehler;              // Instanzvariablen
        private int nenner;

        public Rational(int z, int n) {   // Konstruktor
                zaehler = z; nenner = n;
        }

        public void ausgeben() {          // Methode
                System.out.println(zaehler + " / " + nenner);
        }
```

Quelle: Die Idee zur Konstruktion der Klasse *Rational* stammt aus: Josuttis, N., Objektorientiertes Programmieren in C++: Von der Klasse zur Klassenbibliothek, Addison-Wesley1996

8. Klassen und Objekte in Java

Weitere Methoden für die Klasse Rational

```
        public void addiere(Rational r) {
                zaehler = zaehler*r.nenner + r.zaehler*nenner;
                nenner = nenner*r.nenner;
        }

        public Rational multipliziere(Rational b) {
          return new Rational(zaehler*b.zaehler, nenner*b.nenner);
        }

        public boolean kleinerAls(Rational b){
                if (nenner * b.nenner > 0){
                        // Beide Nenner sind positiv oder negativ
                        return zaehler * b.nenner < b.zaehler * nenner;
                }
                else {
                        // Ein Nenner ist negativ ==> Vergleich umdrehen
                        return zaehler * b.nenner > b.zaehler * nenner;
                }
        }
}
```

8. Klassen und Objekte in Java

Weitere Methoden für die Klasse Rational

Diskussion des Beispiels

- In der Methode *addiere* wird unser Objekt verändert.
- In der Methode *multipliziere* wird eine Referenz auf ein neues Rational-Objekt zurückgegeben.
- In der Methode *kleinerAls* wird ein boolescher Wert zurückgegeben.

Objektorientiertes Paradigma

- Mit der Methode *addiere* wird nicht global die Addition zweier Brüche definiert, sondern an einen Bruch wird die Nachricht "Bilde Addition mit übergebenem Bruch" gesendet.
- Der erste Operand von *multipliziere* ist das Objekt, für das die Funktion aufgerufen wird.
- Interessant ist in *multipliziere*, dass wir in der Funktion auf eine private Komponente eines anderen Objekts zugreifen, zum Beispiel durch *b.zaehler*. → Die Datenkapselung ist typbezogen, d.h. der Zugriff auf private Komponenten ist für Objekte der gleichen Klasse erlaubt.
- Die Methode *kleinerAls* stellt fest, ob ein Bruch kleiner ist als der übergebene Bruch. Das Ungleichheitszeichen dreht sich bei Multiplikation mit einem negativen Wert um.

$$\frac{a}{b} < \frac{c}{d} \iff \begin{cases} a * d < c * b, & \text{falls b und d beide positiv oder beide negativ sind} \\ a * d > c * b, & \text{falls entweder b oder d negativ ist} \end{cases}$$

8. Klassen und Objekte in Java

Weitere Methoden für die Klasse Rational

Beispiel: Überladen von Methoden
Multiplikation eines Bruchs mit einer Ganzen Zahl

```
public Rational multipliziere(int a) {
        Rational tmp = new Rational(0,1);
        tmp.zaehler = zaehler * a;
        tmp.nenner = nenner;
        return tmp;
}
```

Exkurs zur Verdeutlichung der Möglichkeiten der objektorientierten Programmierung
Ändern der Implementation von *kleinerAls* ohne Änderung der Schnittstelle nach Außen.

Beispiel: Vorüberlegungen

- Wie kann die Fallunterscheidung bei der Implementation der Methode *kleinerAls* vermieden werden?
- Wir benötigen keine Fallunterscheidung, wenn der Fall, dass ein Nenner negativ sein kann, nicht auftritt.
- Dies können wir erreichen, indem ein mögliches Minuszeichen im Nenner eines Bruches in den Zähler geschrieben wird → Konstruktor.

8. Klassen und Objekte in Java

Weitere Methoden für die Klasse Rational

Beispiel: Änderung der Klasse ohne Veränderung der Schnittstelle

```
// Konstruktor aus zwei Ganzen Zahlen
// Das Vorzeichen des Bruches steht im Zähler.

public Rational(int z, int n) {
        if (n < 0){
                zaehler = -z;
                nenner = -n;
        }
        else{
                zaehler = z;
                nenner = n;
        }
}

public boolean kleinerAls(Rational b){
        return zaehler * b.nenner < b.zaehler * nenner;
}
```

8. Klassen und Objekte in Java

Weitere Methoden für die Klasse Rational

Diskussion des Beispiels

- Vorteil der Datenkapselung: Solange die Schnittstelle nicht verändert wird, können Methoden beliebig verändert und optimiert werden.
- Die Anwendungen sind hiervon nicht betroffen.
- Bei einem direkten Zugriff auf die Komponenten *zaehler* und *nenner* wäre dies nicht möglich.

Beispiel: Eine kleine Anwendung für die neue Bruchklasse

```
public class Beispiel {
        public static void main(String[] args) {
                Rational x = new Rational(0,1);
                Rational a = new Rational(7,3);
                a.ausgeben();
                // Das Quadrat von a an x zuweisen
                x = a.multipliziere(a);
                x.ausgeben();
        }
}
```

8. Klassen und Objekte in Java

Die this-Referenz

Sinn der this-Referenz

- Wie kann man das Objekt selbst, für das eine Methode aufgerufen wurde, in der Methode ansprechen?
- Ein Beispiel wäre seine Verwendung als Returnwert.
- Zu diesem Zweck gibt es das Schlüsselwort this.

Die this-Referenz ist in Methoden eine Referenz auf das aktuelle Objekt.

Im Folgenden erweitern wir unser Bruch-Beispiel um die Methode *multZuweisung*, die die multiplikative Zuweisung eines Bruches bezeichnet.

Quelle: Die Idee zur Konstruktion der Klasse *Rational* stammt aus: Josuttis, N., Objektorientiertes Programmieren in C++: Von der Klasse zur Klassenbibliothek, Addison-Wesley1996

8. Klassen und Objekte in Java

Die this-Referenz

Anwendungsfall 1: Verwenden des Objekts in einem geschachtelten Ausdruck

Beispiel: Ergänzen der Klasse *Rational* um eine weitere Methode

```
public Rational multZuweisung(Rational b) {
         // multiplikative Zuweisung
         zaehler = zaehler * b.zaehler;
         nenner = nenner * b.nenner;

         // Eine Referenz auf das aktuelle Objekt wird zurückgegeben.
         return this;
}
```

Diskussion des Beispiels

- Der erste Operand ist das Objekt, für das die Methode aufgerufen wird.
- Der zweite Operand wird als Parameter übergeben.
- Der Rückgabewert ist eine Referenz auf das Objekt, für das die Methode aufgerufen wurde.

8. Klassen und Objekte in Java

Die this-Referenz

Beispiel: Eine Anwendung für die neue Methode

```
public class Beispiel {
        public static void main(String[] args) {
                Rational x = new Rational(0,1);
                Rational y = new Rational(2,2);
                x.ausgeben();
                x = x.multipliziere(x);
                x.ausgeben();

                if (x.multZuweisung(y).kleinerAls(y))
                        x.ausgeben();
        }
}
```

Ausgabe des Programms:

```
0 / 1
0 / 1
0 / 2
```

8. Klassen und Objekte in Java

Die this-Referenz

Anwendungsfall 2: Vermeiden von Namenskonflikten

Beispiel: Ergänzung der Klasse *Rational* um eine weitere Methode

```
public Rational setzeWerte(int zaehler, int nenner) {
        this.zaehler = zaehler;
        this.nenner = nenner;
        // Eine Referenz auf das aktuelle Objekt wird zurückgegeben
        return this;
}
```

Diskussion:

- Wieso wird hier *this.zaehler* benutzt, obwohl es sich um eine Methode für `Rational` handelt?
- Im obigen Fall hätten wir ohne *this* einen Namenskonflikt *zaehler = zaehler*.
- Daher kann man hier mit der *this*-Referenz auf die durch die lokale Variable *zaehler* verdeckte Instanzvariable *(this.)zaehler* zugreifen.
- Dies wendet man sehr häufig bei **Konstruktoren** an.

.

8. Klassen und Objekte in Java

Die this-Referenz

Anwendungsfall 3: Aufruf von komplizierten Konstruktoren

Mit

```
this(<Parameterliste>)
```

kann ein Konstruktor einer Klasse **in seiner ersten Anweisung** einen anderen Konstruktor derselben Klasse aufrufen. Dies verwendet man, um „einfache" Konstruktoren bauen zu können, die kompliziertere aufrufen.

Beispiel: Verwendung von *this()* zum Aufruf eines Konstruktors

```
public class MS_Window {
        private String position;
        private String farbe;
```

8. Klassen und Objekte in Java

Die this-Referenz

Beispiel: Verwendung von *this()* zum Aufruf eines Konstruktors

```
    // Dieser Konstruktor muss Hunderte von Windows Eigenschaften setzen
    public MS_Window(String position, String farbe) {
            this.position = position;
            this.farbe = farbe;
    }

    // Dieser Konstruktor erzeugt ein Standard Window und benötigt dafür keinen einzigen Parameter.
    public MS_Window() {
            this("50,50", "blue");
    }

    public static void main(String[] args) {
            MS_Window meinWindow = new MS_Window();
            System.out.printf("Position %s %n", meinWindow.position);
            System.out.printf("Farbe %s", meinWindow.farbe);
    }
}
```

8. Klassen und Objekte in Java

Klassenvariablen und Klassenmethoden

Schlüsselwort static

- Für die Methode *main* haben wir immer das Schlüsselwort *static* verwendet.
- Die Methode *main* wird von der JVM aufgerufen, ohne dass ein Objekt der Klasse erzeugt wird.

- In Java definiert der Modifikator static **Klassenvariablen** und **Klassenmethoden**.

Drei Arten von Variablen in Java

- Klassenvariablen (mit Modifikator *static*) werden für jede Klasse nur einmal angelegt.
- Instanzvariablen gibt es für jede angelegte Instanz einer Klasse, also für jedes Objekt.
- Lokale Variablen gibt es in Methoden.

8. Klassen und Objekte in Java

Klassenvariablen und Klassenmethoden

Beispiel:

```
public class Rational {
        private int zaehler;                          // Instanzvariable
        private int nenner;                           // Instanzvariable
        private static String farbe = "blue";         // Klassenvariable

        public Rational(int zaehler, int nenner, String farbe) {
                this.zaehler = zaehler;
                this.nenner = nenner;
                Rational.farbe = farbe;
        }

        public void kehrwert() {
                int tmp = zaehler;                    // lokale Variable
                zaehler = nenner;
                nenner = tmp;
        }
```

8. Klassen und Objekte in Java

Klassenvariablen und Klassenmethoden

```
public static void main(String[] args) {
        Rational r1 = new Rational(1,2, "black");
        Rational r2 = new Rational(2,4, "blue");

        Rational.farbe = "white";

        System.out.printf("Bruch 1 - Zähler: %d %n", r1.zaehler);
        System.out.printf("Bruch 2 - Zähler: %d %n", r2.zaehler);
        System.out.printf("Bruchfarbe: %s %n", Rational.farbe);
}
```

Ausgabe:

```
Bruch 1 - Zähler: 1
Bruch 2 - Zähler: 2
Bruchfarbe: white
```

Diskussion des Beispiels

- Unsere Brüche haben eine Farbe. Die Variable **`farbe`** wird nur einmal für alle erzeugten Objekte angelegt. Sie ist also eine Art globale Variable. Alle Brüche haben die gleiche Farbe. Ändert ein Bruch seine Farbe, so ändert sich die Farbe für alle Brüche, in unserem Fall: **`black`** → **`blue`** → **`white`**.

8. Klassen und Objekte in Java

Klassenvariablen und Klassenmethoden

Statische Variablen (Klassenvariablen) und Methoden (Klassenmethoden)

- Statische Variablen können / sollten nur über den Klassennamen angesprochen werden. Das ist etwas, was man mit einer *non-static* Komponente nicht tun kann: **`Rational.farbe = "blue"`**.
- Klassenvariablen kann man also verwenden, ohne dass man ein Objekt erzeugen muss.
- Das Gleiche gilt für *static-Methoden*. Man sollte sich auf eine *static*-Methode nur über **`Klassenname.methode()`** beziehen.

Wichtige Anwendung für static

- Die Definition von globalen Variablen und Methoden ist in Java nur über Klassenvariablen und Klassenmethoden möglich, nicht über die Position im Gültigkeitsbereich wie in C.
- Alles gehört in eine Klasse!

8. Klassen und Objekte in Java

Klassenvariablen und Klassenmethoden

Beispiel:

```
public class Wartende {
        private int nummerDesWartenden = 0;
        private static int laengeWarteschlange = 0;

        public void nimmWarteNummer() {
                nummerDesWartenden = ++laengeWarteschlange;
        }

        public static void main(String[] args) {
                System.out.println("Länge: " + Wartende.laengeWarteschlange);
                Wartende frank = new Wartende();
                frank.nimmWarteNummer();
                System.out.println("Länge: " + laengeWarteschlange);
        }
}
```

Diskussion des Beispiels

- In der Klassenvariablen **`laengeWarteschlange`** wird sich gemerkt, wie voll die Warteschlange ist. In **`main`** wird auf diesen "globalen" Wert über die Klasse zugegriffen.

8. Klassen und Objekte in Java

Klassenvariablen und Klassenmethoden

Beispiel:

```
public class Wartende {
        private int nummerDesWartenden = 0;
        private static int laengeWarteschlange = 0;

        public void nimmWarteNummer() {
                nummerDesWartenden = ++laengeWarteschlange;
        }

        public static int gibWarteschlangenLaenge() {
                return laengeWarteschlange;
        }

        public static void main(String[] args) {
                System.out.println("Länge:" + Wartende.gibWarteschlangenLaenge());
                Wartende frank = new Wartende();
                frank.nimmWarteNummer();
                System.out.println("Länge: " + gibWarteschlangenLaenge());
        }
}
```

8. Klassen und Objekte in Java

Klassenvariablen und Klassenmethoden

Beispiel:

```
public class Rational {
        private int zaehler;
        private int nenner;

        public Rational(int z, int n) {
                zaehler = z; nenner = n;
        }

        public void ausgeben() {
                System.out.println(zaehler + " / " + nenner);
        }

        // Klassenmethode zum Erweitern eines Bruches
        public static Rational erweitern(Rational b, int faktor) [
        Rational tmp = new Rational(b.zaehler*faktor, b.nenner*faktor);
        return tmp;
        }
```

8. Klassen und Objekte in Java

Klassenvariablen und Klassenmethoden

```
public void add(Rational b) {
         Rational tmp1 = erweitern(b, nenner);
         Rational tmp2 = Rational.erweitern(this, b.nenner);
         zaehler = tmp1.zaehler + tmp2.zaehler;
         nenner = tmp1.nenner;
         }
}
```

Diskussion des Beispiels:

- Der Klassenname wurde weggelassen, da **`erweitern`** zur Klasse gehört. Wäre **`erweitern`** nicht *static*, könnte man das auch so schreiben, allerdings würde der Compiler dann **`this.erweitern`** ersetzen.

Zu beachten:

- In Java sollte man eine Klassenmethode über die Klasse selbst ansprechen und nie (!) über ein Objekt.
- Instanzmethoden sind nur über eine Referenz auf ein Objekt ansprechbar.
- Klassenmethoden besitzen keine this-Referenz. Sie können also nicht über die *this*-Referenz auf Klassenvariablen bzw. Klassenmethoden zugreifen.

8. Klassen und Objekte in Java

Klassenvariablen und Klassenmethoden

Beispiel: Benutzung von statischen Methoden aus Bibliotheken

```
import java.util.*;

public class Eigenschaften {
        public static void main(String[] args) {
                System.out.println(new Date());
                Properties p = System.getProperties();
                p.list(System.out);
                System.out.println("Memory Usage: ");
                Runtime rt = Runtime.getRuntime();
                System.out.println("Total Memory = " + rt.totalMemory() +
                                 " Free Memory = " + rt.freeMemory());
        }
}
```

Quelle: Eckel, B., Thinking in Java, Markt + Technik, 2003

8. Klassen und Objekte in Java

Klassenvariablen und Klassenmethoden

Ausgabe:

```
Mon Sep 17 12:40:40 CEST 2018

-- listing properties --
java.runtime.name=Java(TM) SE Runtime Environment
sun.boot.library.path=C:\Program Files\Java\jre1.8.0_181\bin
java.vm.version=25.181-b13
…
java.vm.specification.vendor=Oracle Corporation
os.name=Windows 10
sun.jnu.encoding=Cp1252
java.library.path=C:\Program Files\Java\jre1.8.0_181\bi...
java.specification.name=Java Platform API Specification
sun.cpu.isalist=amd64
…
Memory Usage:
Total Memory = 128974848 Free Memory = 126929840
```

8. Klassen und Objekte in Java

Klassenvariablen und Klassenmethoden

Diskussion des Beispiels

- **`import`** bewirkt, dass Klassen, die man für den Code der Datei benötigt, bekannt gemacht werden. Die Bibliothek **`java.lang`** wird immer automatisch in jede Java-Datei eingebracht. **`import`** ist damit eine Art Abkürzung, so dass man nicht den ganzen Klassennamen schreiben muss.
- **`packages.html`** in der API enthält eine Liste aller Klassenbibliotheken.
- In **`java.lang`** findet man die Klassen **`System`** und **`Runtime`**, welche in **`Property.java`** benutzt werden.
- In **`java.lang`** gibt es allerdings keine Klasse **`Date`**. Man muss also eine andere Bibliothek importieren.
- Man wählt *Tree* in der API, dann die Suchfunktion des Browsers.
- Tut man dies, um **`Date`** zu finden, sieht man **`java.util.Date.`**
- **`Date`** ist also in der *util*-Bibliothek. Diese muss man importieren.

- Die erste Zeile des Programms ist ziemlich interessant: **`System.out.println(new Date());`**
- Ein Date-Objekt wird erzeugt, um seinen Wert auszugeben. Sobald dieser Befehl beendet ist, ist dieses Datum überflüssig. Der Garbage Collector kann es vernichten.
- Die zweite Zeile ruft **`System.getProperties()`** auf.
- **`getProperties()`** ist eine static-Methode der Klasse *System* und erzeugt die Systemeigenschaften in Form eines Objekts der Klasse **`Properties`**.

8. Klassen und Objekte in Java

Klassenvariablen und Klassenmethoden

Beispiel:
Lassen Sie uns einen kurzen Blick auf einen winzigen Ausschnitt der Java Klasse **`Math`** in der API werfen (vgl. [**`https://docs.oracle.com/javase/17/docs/api/`**]):

java.lang
Class Math

```
java.lang.Object
  └ java.lang.Math
```

```
public final class Math
extends Object
```

The class `Math` contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.

…

Since:
JDK1.0

8. Klassen und Objekte in Java

Klassenvariablen und Klassenmethoden

Field Detail

E

`public static final double E`

The `double` value that is closer than any other to *e*, the base of the natural logarithms.

See Also:

Constant Field Values

PI

`public static final double PI`

The `double` value that is closer than any other to *pi*, the ratio of the circumference of a circle to its diameter.

See Also:

Constant Field Values

8. Klassen und Objekte in Java

Klassenvariablen und Klassenmethoden

Method Detail

sin

`public static double sin(double a)`

Returns the trigonometric sine of an angle. Special cases:

- If the argument is NaN or an infinity, then the result is NaN.
- If the argument is zero, then the result is a zero with the same sign as the argument.

The computed result must be within 1 ulp of the exact result. Results must be semi-monotonic.

Parameters:
`a` - an angle, in radians.
Returns:
the sine of the argument.

8. Klassen und Objekte in Java

Klassenvariablen und Klassenmethoden

Beispiel:
Sie sehen in der Klasse **`Math`** die Klassenvariablen **`PI`** und die Klassenmethode **`sin`**. Wir wollen diese benutzen und zusätzlich die **`toRadians`** Methode verwenden.

```
public class MathSinPi {
        public static void main(String[] args) {
                System.out.println("PI ist: " + Math.PI);
                System.out.println("Sinus ist: " + Math.sin(Math.toRadians(90)));
        }
}
```

Ab Java 5 kann man das weniger aufwendig gestalten, in dem man einen statischen Import verwendet.

```
import static java.lang.Math.*;    // Alles auf einmal, oder:
                                   // Gezielter Import, z.B. java.lang.Math.PI;
…
        System.out.println("PI ist: " + PI);
        System.out.println("Sinus ist: " + sin(toRadians(90)));…
```

8. Klassen und Objekte in Java

Enumerations

- Im Teil zur C-Programmierung hatten wir **`enum`** zur Definition von Aufzählungstypen eingeführt.

- Ein entsprechendes Gegenstück gibt es in Java ab Version 1.5 mit dem Unterschied, dass die Komponenten der Enumerations keine Integer sondern Objekte sind.

- Eine **`enum`**-Definition verhält sich wie eine Klassendefinition, nur dass keine Unter- und Oberklassen möglich sind. Sonst erlaubt **`enum`** auch die Definition von Methoden und Variablen.

- Den *enums* können Zahlenwerte zugeordnet werden (**`heute.value()`**), sie haben eine **`toString`**-Methode, mit der sie lesbar ausgegeben werden und sie können auch per Namen eingelesen werden.

Beispiel:

```
public enum Arbeitstag {MONTAG, DIENSTAG, MITTWOCH, DONNERSTAG, FREITAG}
```

Hinter den Elementen stehen einzelne Objekte, die sich wie alle anderen auch weiterverarbeiten lassen. Intern erstellt der Compiler normalen Bytecode für die Klasse **`Arbeitstag`** und mit ihr 5 Objekte.

8. Klassen und Objekte in Java

Enumerations

Beispiel:

```
public class Enumerations {

        public enum Arbeitstag {MONTAG, DIENSTAG, MITTWOCH, DONNERSTAG, FREITAG}

        public static void main(String[] args) {
                Arbeitstag heute = Arbeitstag.MONTAG;
                switch (heute) {
                        case MONTAG: System.out.println("Montag"); break;
                        case DIENSTAG: System.out.println("Dienstag"); break;
                        …
                        case FREITAG: System.out.println("Freitag"); break;
                        default:
                                System.out.println("Kein Arbeitstag");
                }
        }
}
```

9. Arrays und Strings in Java

Arrays

Ziele von Java

- Eines der primären Ziele von Java ist Sicherheit.
    - Ein Java Array wird garantiert initialisiert (Speicher eines Arrays wird auf **`null`** gesetzt).
    - Man kann nicht außerhalb der Array-Grenzen zugreifen (**`IndexOutOfBoundsException`**).
- Arrays sind in Java Objekte.
- In Java kann man Arrays aus Komponenten von Werttypen und Referenztypen bilden.
- Arrays werden dynamisch zur Laufzeit angelegt.
- Mehrdimensionale Arrays funktionieren wie in C.

Da Arrays Objekte sind, wird auf sie über Referenzen zugegriffen. Mit

```
WerttypName[] refName; oder
ReferenztypName[] refName;
```

definiert man eine Referenz **`refName`** auf ein Array, dessen Komponenten vom Typ **`WerttypName`** bzw. **`ReferenztypName`** sind.

9. Arrays und Strings in Java

Arrays

Unterschied zwischen Referenz und Objekt

- Die Länge des Arrays wird erst beim Erzeugen des konkreten Objekts angegeben.
- Das Array-Objekt selbst wird mit **`new`** erzeugt oder über eine Initialisierungsliste.
- Die Länge des Arrays **`zahlen`** wird über die öffentliche Instanzvariable **`length`** ermittelt.
- Darüber hinaus erbt jedes Array automatisch alle Methoden der Klasse **`Object`**, z.B.
  `public boolean equals(Object refObject);`

Beispiel: Arrays mit Werttyp-Komponenten

```
// Anlegen einer Referenz auf ein int-Array
int[] zahlen;

// Erzeugen eines Array-Objekts mit 5 Elementen, die default-
// mäßig mit 0 initialisiert werden
zahlen = new int[5];

// Äquivalent zu den obigen beiden Code-Zeilen ist
int[] zahlen = new int[5];
```

9. Arrays und Strings in Java

Arrays

Beispiel: Arrays mit Werttyp-Komponenten

```
// Die Länge des Arrays kann eine Variable sein und erst
// zur Laufzeit angegeben werden, z.B. über die Tastatur
int i = 5;
int[] zahlen = new int[i];

// Implizites Erzeugen eines Arrays mit 4 Elementen über die
// Initialisierungsliste (wie in C)
int[] zahlen = {1,2,3,4};

// Initialisierung des Arrays über eine Schleife
for(int i=0; i < zahlen.length; i++)
        zahlen[i] = i;
```

9. Arrays und Strings in Java

Arrays

Beispiel: Arrays mit Referenztyp-Komponenten

```
// Anlegen einer Referenz auf ein Array, dessen Komponenten
// Referenzen auf Objekte der Klasse Rational sind:
Rational[] brueche;

// Explizites Erzeugen eines Array-Objekts mit 5 Elementen,
// die defaultmäßig mit null initialisiert werden:
brueche = new Rational[5];

// Äquivalent zu den obigen beiden Code-Zeilen ist:
Rational[] brueche = new Rational[5];

// Zuweisung einer Komponente:
Rational refRationalObjekt = new Rational();
brueche[3] = refRationalObjekt;
```

9. Arrays und Strings in Java

Arrays

```
// Initialisierung des Arrays mit Referenzen auf Brüche:
for (int i=0; i < brueche.length; i++)
        brueche[i] = new Rational();

// Implizites Erzeugen eines Arrays mit 1 Element über die
// Initialisierungsliste (wie in C):
Rational[] brueche = {refRationalObjekt};
```

Diskussion des Beispiels

- Arrays sind in Java Objekte.
- In im Falle von Referenztypen verweist jede Komponente des Arrays wieder auf ein (eigenständiges) Objekt.
- Die Länge des Arrays **`brueche`** wird ermittelt, indem wir die öffentliche Instanzvariable **`length`** abgefragt haben, die zu allen Arrays automatisch existiert.

9. Arrays und Strings in Java

Arrays

```
public boolean equals(Object refObject);
```

Der Aufruf `a.equals(b)` gibt den Wert `true` zurück, wenn `a` und `b` Referenzen auf das gleiche Objekt sind.

Beispiel:

```
public class Arrays {
        public static void main(String[] args) {
                int[] a = new int[2];
                int[] b;
                b = a;
                System.out.println("a equals b ist:" + a.equals(b));
                System.out.println("Länge von b ist:" + b.length);
        }
}
```

Die Ausgabe lautet:

```
        a equals b ist: true
        Länge von b ist: 2
```

9. Arrays und Strings in Java

Arrays

for-each-Schleife
Die for-each-Schleife ermöglicht die Iteration über alle Werte eines Arrays, ohne Indices zu benutzen.

Beispiel: Summe der Elemente eines Arrays

```
// Variante mit for-Schleife (wie in C mit Indices)

static double summe(double[] a) {
        double s = 0.0;
        for (int i = 0; i < a.length; i++) s = s + a[i];
        return s;
}

// Variante mit for-each-Schleife (ohne Indices)

static double summe(double[] a) {
        double s = 0.0;
        for (double each : a) s = s + each;
        return s;
}
```

9. Arrays und Strings in Java

Arrays

```
public class ForEachBeispiel {                                  // Ausgabe:
        public static void main(String[] args){                 // 1
                int[] zahlen = {1, 2, 3};                       // 2
                for (int each: zahlen)                          // 3
                        System.out.println(each);               // 6.0
                System.out.println(summe1(zahlen));             // 6.0
                System.out.println(summe2(zahlen));
        }
        static double summe1(int[] a) {
                int s = 0;
                for (int i = 0; i < a.length; i++) s = s + a[i];
                return s;
        }
        static double summe2(int[] a) {
                double s = 0;
                for (int each : a) s = s + each;
                return s;
        }
}
```

9. Arrays und Strings in Java

Arrays

Aufgabe: Arrays in Java

Schreiben Sie ein Java-Programm, das Arrays verwendet.

Formulieren Sie die auszuführenden Aktionen als Methoden.

Das Hauptprogramm soll folgende Aktionen nacheinander ausführen:
- Einlesen von Zahlen: Es soll gefragt werden, wie viele Zahlen einzulesen sind.
- Mittelwert der Zahlen ausgeben.
- Die Größte der Zahlen ausgeben.
- Die Kleinste der Zahlen ausgeben.

9. Arrays und Strings in Java

Arrays

Implementation:

```
import java.util.Scanner;

public class Arrays {
        private static Scanner in;

        public static int[] einlesen() {
                int[] refArray;

                System.out.println("Wieviele Elemente: ");
                int n = in.nextInt();

                refArray = new int[n];

                for(int i=0; i < refArray.length; i++) {
                        System.out.println("Bitte Element " + i + " eingeben: ");
                        refArray[i] = in.nextInt();
                }
                return refArray;
        }
```

9. Arrays und Strings in Java

Arrays

```
        public static void ausgeben(int[] refArray) {
                for(int i=0; i < refArray.length; i++)
                System.out.print(refArray[i]);
        }

        public static void main(String args[]) {
                in = new Scanner(System.in);

                int[] refArray = einlesen();
                ausgeben(refArray);
        }
}
```

9. Arrays und Strings in Java

Arrays

Aufgabe: Fibonacci rekursiv vs. linear

a) Geben Sie eine rekursive Methode für Fibonacci an:

Implementation: Rekursion in Java

```
public static int fibonacci(int n) {
        if (n <= 1) return 1;
        return fibonacci(n-1) + fibonacci(n-2);
}
```

Aufwand: exponentiell: $O(2^n)$

9. Arrays und Strings in Java

Arrays

Aufgabe: Fibonacci rekursiv vs. linear

b) Geben Sie nun eine Lösung an, mit der Fibonacci in linearer Zeit berechnet werden kann. Benutzen Sie ein Array zur Speicherung von Zwischenergebnissen!

Implementation: Lineare Lösung

```
public class FibonacciArray {
        public static int fib(int n) {
                int[] refIntArray = new int[n+1];
                refIntArray[0] = 1; refIntArray[1] = 1;

                for (int i = 2; i <= n; i++)
                        refIntArray[i] = refIntArray[i-1] + refIntArray[i-2];
                return refIntArray[n];
        }
}
```

Aufwand: O(n), Memoisation

9. Arrays und Strings in Java

Strings

- Strings sind Java komfortabler als in C.
- Strings sind in Java Objekte.
- In Java gibt es zwei verschiedene Stringklassen
    - die Klasse `String` für konstante Zeichenketten und
    - die Klasse `StringBuilder` (ab Java 5) zum Aufbau von Strings, z.B. bei Textverarbeitung, weil dies effizienter ist als mit `String`. Bitte schauen Sie sich diese Klasse in der Literatur an.

- Eine **konstante Zeichenkette** `"abcd"` ist ein Ausdruck, der als Wert eine Referenz auf das String-Objekt mit dem Inhalt 'a' 'b' 'c' 'd' hat. Inhalt und Länge des Strings können nicht verändert werden.
- Unterschied zu C: Es gibt kein terminierendes Null-Byte.

- Die Länge eines Strings kann man durch Aufruf der Methode `length()` berechnen.
- Sie gibt die Anzahl der Zeichen eines Strings zurück.

9. Arrays und Strings in Java

Strings

Erzeugen von Strings

- Das Erzeugen von Strings geht ganz einfach.
- Strings werden durch Literale angegeben. Es ist theoretisch auch möglich, **`new`** oder eine *Initialisierungsliste* zu verwenden. Das macht man aber nie, da dies zu ineffizient ist und man dem Compiler keine Möglichkeit zur Optimierung lässt. Daher gehen wir darauf nicht ein.

Beispiel: Erzeugen von Strings

```
String meinName = "Frank";
String nochmalMeinName = "Frank";
```

- Die Klasse *String* ist im Paket **`java.lang`** definiert. Methoden zur Stringverarbeitung:
- **`public boolean equals(Object obj)`**
- **`public int length()`**
- **`public String substring(int anfang, int ende)`** – Herausschneiden eines Substrings
- **`public String trim()`** - Entfernen von Leerzeichen am Anfang und Ende des Strings
- **`public char charAt(int i)`** - Ermittlung des Char-Zeichens an der Position *i*.

9. Arrays und Strings in Java

Strings

Beispiel: Stringverarbeitung

```
public class Zeichenketten {
        public static void main(String[] args) {
                String a = "Frank in Bonn";
                System.out.println(a);
                System.out.println("Länge ist: " + a.length());
                String wer = a.substring(0, 5);
                if(wer.equals("Frank"))
                        System.out.println("Frank");
        }
}
```

Ausgabe:

```
Frank in Bonn
Länge ist: 13
Frank
```

Anmerkung: `equals` vergleicht für Strings tatsächlich die Stringinhalte und nicht die Referenzen.

9. Arrays und Strings in Java

Strings

Beispiel:

Schreiben Sie eine Klasse *CharStack*.

- Ein *Stack* (*Stapel* oder *Kellerspeicher*) ist eine Datenstruktur, bei der man nur auf das oberste Element zugreifen kann. Man kann ein Datenelement auf den Stapel legen (*push*) und das oberste herunternehmen (*pop*). Zusätzlich sind in der Klasse nur die *Initialisierung* und die Abfrage, ob der Stack *leer* ist, definiert.

- CharStack heißt die Klasse, da in diesem Beispiel nur Daten vom Typ *char* auf dem Stack gespeichert werden können.

- Neben der Implementierung der Klasse *CharStack* besteht Ihre Aufgabe darin, in der *main*-Methode einen String einzulesen und die Reihenfolge der Buchstaben dadurch umzukehren, dass Sie diese nacheinander alle auf den Stapel schieben und dann alle herunterholen.

- Die Klasse *CharStack* ist teilweise vorgegeben. Sie müssen sie nur ergänzen:

9. Arrays und Strings in Java

Strings

Vorgabe der Klasse:

```
public class CharStack {
        private int top;          // nächste freie Stelle
        private char[] stack;

        public CharStack(int laenge) {
                // Bitte ergänzen
        }
        public void push(char zeichen) {
                // Bitte ergänzen
        }
        public char pop() {
                // Bitte ergänzen
        }
        public boolean is_empty() {
                // Bitte ergänzen
        }
}
```

9. Arrays und Strings in Java

Strings

Implementation:

```
public class CharStack {
        private int top;
        private char[] stack;

        public CharStack(int laenge) {
                top = 0;
                stack = new char[laenge];
        }
        public void push(char zeichen) {
                stack[top++] = zeichen;
        }
        public char pop() {
                return stack[--top];
        }
        public boolean is_empty() {
                return (top == 0);
        }
}
```

9. Arrays und Strings in Java

Strings

```
import java.util.Scanner;

public class CharStackApplication {
        private static Scanner in;

        public static void main (String[] args) {
                in = new Scanner(System.in);
                System.out.print("Bitte String eingeben: ");
                String x = in.next();

                int stringLaenge = x.length();
                CharStack meiner = new CharStack(stringLaenge);

                int i=0;
                while (i < x.length())
                        meiner.push(x.charAt(i++));

                while (! meiner.is_empty())
                        System.out.print(meiner.pop());
        }
}
```

Teil C: Objektorientierte Programmierung in der Sprache Java

Literatur zur Vertiefung

Eckel, B., Thinking in Java, Markt + Technik, 2003

Goll, J., Heinisch, C., Java als erste Programmiersprache: Grundkurs für Hochschulen, Springer Vieweg, 2016

Inden, M., Der Weg zum Java-Profi: Konzepte und Techniken für die professionelle Java-Entwicklung. Aktuell zu Java 9., dpunkt.verlag, 2017

Jobst, F., Programmieren in Java, Carl Hanser, 2014

Josuttis, N., Objektorientiertes Programmieren in C++: Von der Klasse zur Klassenbibliothek, Addison-Wesley1996

Louis, D. , Müller, P., Java: Eine Einführung in die Programmierung, Carl Hanser, 2014

Küneth, Th., Einstieg in Eclipse: Die Werkzeuge für Java-Entwickler, Galileo Computing, 2014

Schiedermeier, R., Programmieren mit Java, Pearson Studium, 2010

Ullenboom, C., Java ist auch eine Insel: Programmieren lernen mit dem Standardwerk für Java-Entwickler, Rheinwerk Computing, 2017

Algorithmen
und
Programmierung I

WS 2025 / 2026

Teil D: Komplexität von Algorithmen

Prof. Dr. Frank Victor
TH Köln

Inhalt

Teil D: Komplexität von Algorithmen

1 Der Begriff Komplexität
2 O-Notation
3 Komplexitätsklassen und Algorithmen
4 Beispiele zum Abschätzen der Zeitkomplexität
5 Aufgaben zur Zeitkomplexität
6 Animation von Sortieralgorithmen
7 Weiterführende Literatur

1. Der Begriff Komplexität

- Algorithmen benötigen Ressourcen, wie Rechenzeit (Zeitkomplexität) und Speicherplatz (Platzkomplexität).
- Der **Ressourcenbedarf** hängt von der **Größe des Problems** ab, das der Algorithmus lösen soll.

Beispiel: Sortieren einer Liste von Zahlen
Der Ressourcenbedarf eines Sortieralgorithmus hängt von der Länge der Liste ab, die zu sortieren ist. Je länger die Liste ist, umso mehr Vergleiche müssen durchgeführt werden. Der Zeitaufwand zum Sortieren ist also für lange Listen viel höher als für kurze.

Interessant ist nicht so sehr die Frage, wie sich ein Algorithmus für ein spezielles n als Eingabegröße verhält, sondern vielmehr die Frage, wie er sich für ein beliebig großes n verhält. Die Zeitkomplexität eines Algorithmus ist damit eine **Funktion f(n)**. Diese Funktion wächst typischerweise monoton.

Beispiel: Sortieren einer Liste von Zahlen
Das Sortieren einer Liste mit 3 Zahlen benötigt $f(3) = x$ Schritte
Das Sortieren einer Liste mit 300 Zahlen benötigt $f(300) = y$ Schritte. y ist im Allgemeinen viel größer als x.
Dabei interessiert uns aber nicht der konkrete Wert von x oder y sondern eher die **Größenordnung** und die Frage, wie f(n) mit wachsendem n skaliert. Ist das Wachstum z.B. **konstant**, **linear**, **quadratisch** oder **exponentiell** ?

1. Der Begriff Komplexität

- Man will also nicht den genauen Wert der Komplexitätsfunktion f(n) ermitteln, sondern nur wissen, wie schnell sie wächst. Das erreicht man, indem man die Komplexitätsfunktionen in **Komplexitätsklassen** einteilt. Mit **O(f(n))** bezeichnet eine Komplexitätsklasse, wo das Wachstum der Komplexitätsfunktion nicht schneller als bei **f(n)** erfolgt. Dabei bezeichnet **n** die **Problemgröße**.

Hinweis:
Die obige Definition der Komplexitätsklassen ist anfangs schwer zu verstehen. Machen Sie sich die Definition klar, in dem Sie konkrete Beispiele für die Schrankenfunktion **f(n)** verwenden, beispielsweise n oder n^3. Die Komplexitätsklassen heißen dann O(n) bzw. $O(n^3)$.

Beispiel: Komplexitätsklassen O(n) vs. $O(n^3)$
O(n) bedeutet lineare Komplexität. Solche Algorithmen sind bezüglich der Zeitkomplexität gutmütig. Eine Verdopplung der Problemgröße bewirkt auch nur eine Verdopplung der Ausführungsgeschwindigkeit.

$O(n^3)$ ist kritischer. Wenn n 8-mal größer wird, wird schon 8 * 8 * 8 = 512 mehr Rechenzeit benötigt.
Es gibt aber noch schneller wachsende und daher unappetitlichere Komplexitätsklassen. Schnellere Rechner verbessern die Situation hier kaum – **außer Quantencomputer** ! Die Komplexität liegt in den Funktionen selbst.

1. Der Begriff Komplexität

Beispiel: Problemgrößen

- Zahl der Städte, die eine Traveling Saleswoman ☺ besuchen soll
- Anzahl der Telefonbucheinträge, die nach einer Nummer durchsucht werden soll
- Bei Graph-Algorithmen: Zahl der Kanten und Knoten
- Größe der Matrizen in einer Matrixmultiplikation

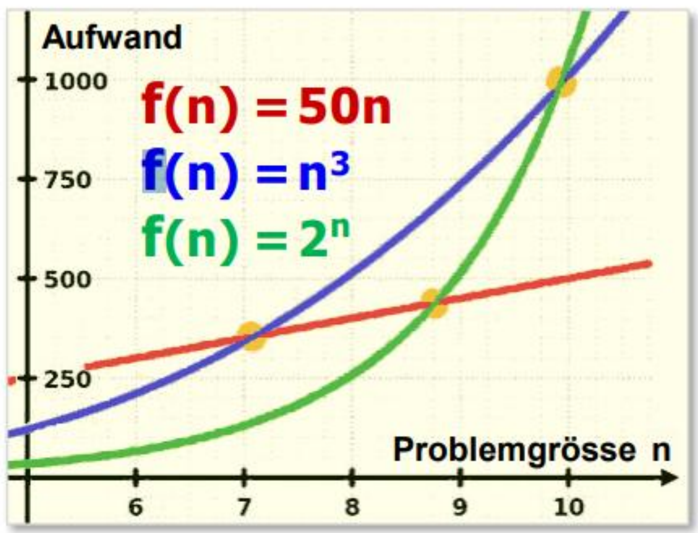
- Die **Zeitkomplexität** misst man normalerweise als Anzahl von Berechnungsschritten (Operationen in Abhängigkeit vom Umfang der Eingabedaten).

- **Jede Operation in einem Programm kostet Zeit.** Diese kann man so bestimmen:
  - Bei elementarer Arithmetik (Addition, Multiplikation., Modulo) nehmen wir an, dies benötigt 1 Schritt.
  - Bei booleschen Operationen (x <= y) benötigen wir ebenfalls 1 Schritt.
  - Ebenso bei der Wertzuweisung (x = 3) 1 Schritt.
  - Bei Schleifen (for, while, do-while) sind die Kosten für m Durchläufe:
    1 Schritt für die Initialisierung, m Abbruchvergleiche, m Inkrementierungen und 1 Schritt für jede Anweisung im Schleifenkörper.
  - Bei der bedingten Anweisung (if) ergeben sich folgende Kosten:
    Kosten des logischen Ausdrucks + maximale Kosten beider Zweige
  - Bei Funktionsaufrufen ist es die Summe der Kosten aller Operationen in der Funktion

1. Der Begriff Komplexität

- Oft ist der Aufwand eines Algorithmus nicht nur von der Problemgröße n, sondern von den **konkreten Eingabewerten (oder ihrer Reihenfolge)** abhängig, daher unterscheidet man:
  - günstigster Aufwand **best case**
  - mittlerer Aufwand **average case**
  - ungünstigster Aufwand **worst case**

- Nochmal zur Anschauung: Bei Komplexitätsfunktionen interessieren uns nicht so sehr die konkreten Werte für ein spezielles n sondern der asymptotische Aufwand (für $n \to \infty$).



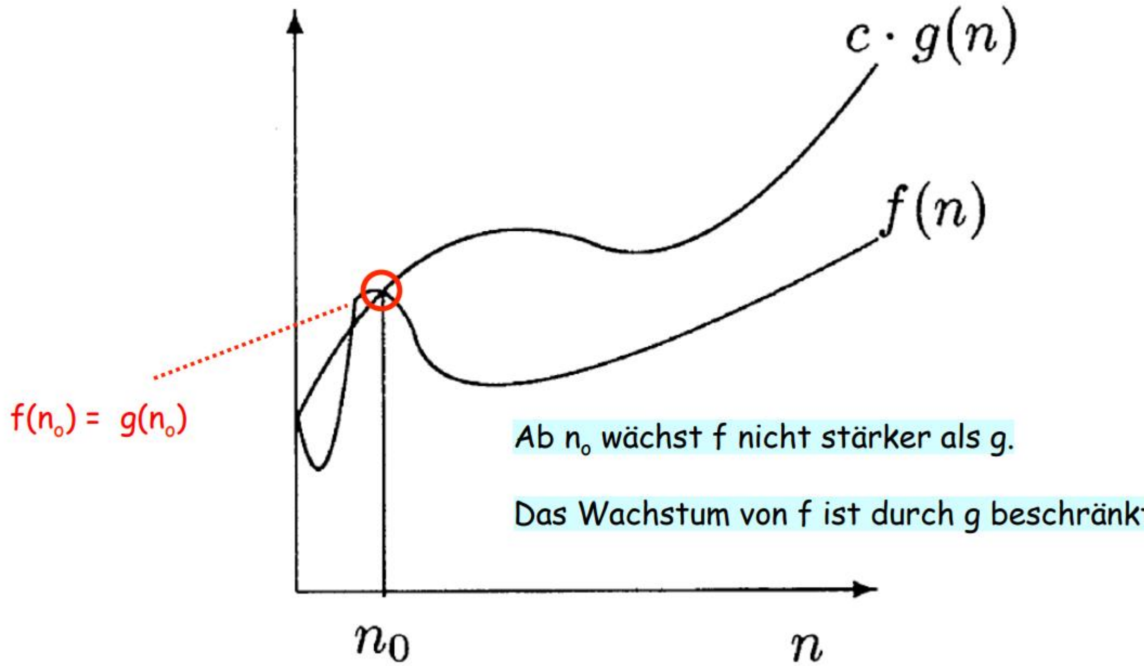
2. O-Notation

Die **O-Notation** (**Landau-Notation**, engl. **big o-notation**) ist eine Funktion zur Messung der Effizienz eines Algorithmus. Sie misst die Zeit, die für die Ausführung des Algorithmus bei einer beliebigen Eingabe benötigt wird. Oder anders ausgedrückt: Wie skaliert die Funktion bei wachsender Komplexität der Eingabe n ?

Der Zahlentheoretiker Paul Bachmann drückte 1894 durch das Zeichen O(n) eine Größe aus, deren Ordnung in Bezug auf n die Ordnung von n nicht überschreitet. Der Zahlentheoretiker Edmund Landau, durch den die O-Notation bekannt wurde und mit dessen Namen sie insbesondere im deutschen Sprachraum verbunden ist, übernahm Bachmanns Bezeichnung.

2. O-Notation

- **Ziel**
  - Abschätzen der Aufwandsfunktion f(n) durch Angabe einer einfachen Vergleichsfunktion (Schranke) **g(n)**, die – abgesehen von Konstanten – ab einem bestimmten Wert größer als f(n) ist.
  - **Typische Funktionen für g(n)**
    $g(n) = n$, $g(n) = \log n$, $g(n) = n^2$



- **Definition**
  Es sei f, g: N → N.
  g(n) ist eine **asymptotische obere Schranke** für eine Funktion f(n)
  - geschrieben **f(n) = O(g(n))**
  - gelesen **„f(n) ist von der Ordnung g(n)“**
  wenn gilt: $\exists c, n_0: \forall n \geq n_0: f(n) \leq c \cdot g(n)$

Quelle: Aus „https://services.informatik.hs-mannheim.de/~schramm/ads/files/Kapitel07.pdf“, Seite 25, 04.01.2025

2. O-Notation

- **Rechenregeln mit O-Notation**
  - Die O(g)-Notation ist eine obere Schranke für eine Klasse von Funktionen.
    Man kann daher die Funktion g vereinfachen, indem
    - Konstante Faktoren weggelassen werden,
    - Nur der größte Exponent berücksichtigt wird.
  - Ziel ist eine möglichst einfache Funktion für die Aufwandsabschätzung, da man nur an den Größenordnungen interessiert ist.

Beispiele:

g sei obere Schranke. Dann sind folgende Vereinfachungen möglich:

- $g = n^3 + n^2$ → $g = n^3$ nur den größten Exponent berücksichtigen
- $g = \text{ld } n$ → $g = \log n$ nur den Logarithmus zur Basis 10 betrachten, denn $\log_b n = \log_a n * \log_b a$ ($\log_b a$ ist eine Konstante)
- $g = 2 * n$ → $g = n$ Konstanten weglassen
- $g = n + 2$ → $g = n$ Konstanten weglassen

3. Komplexitätsklassen und Algorithmen

Mit der **O-Notation** lassen sich **Komplexitätsklassen** gut darstellen.

- **Komplexitätsklassen in aufsteigender Ordnung**:
  - $O(1)$ ist unabhängig (konstant) von der Eingangsgröße n
  - $O(\log(n))$ wächst logarithmisch mit der Eingangsgröße n
  - $O(n)$ wächst linear mit der Eingangsgröße n
  - $O(n \log(n))$ wächst linear-logarithmisch mit der Eingangsgröße n (Die Basis ist hier unerheblich!)
  - $O(n^2)$ wächst quadratisch mit der Eingangsgröße
  - $O(n^3)$ wächst kubisch mit der Eingangsgröße n
  - $O(n^m)$, $m \in \mathbb{N}$ wächst polynomiell mit m
  - $O(k^n)$ wächst exponentiell mit der Eingangsgröße n
  - $O(n!)$ wächst gemäß der Fakultät der Eingangsgröße n

Es gilt:

$O(1) \subseteq O(\log n) \subseteq O(n) \subseteq O(n^2) \subseteq O(n^k) \subseteq O(k^n)$ für alle $k \geq 2$

3. Komplexitätsklassen und Algorithmen

Hier einige Komplexitätsklassen mit typischen Algorithmen

Komplexitätsklasse	Beispiel-Algorithmus
O(1)	Zugriff auf das i-te Element eines Arrays, Addition, Vergleichsoperationen, rekursiver Aufruf
O(log n)	Binäre Suche in einem sortierten Array mit n Elementen
O(n)	Suche in einem unsortierten Array mit n Elementen (Lineare Suche), Bearbeiten jedes Elementes einer Menge
O(n log n)	Schnelles Sortieren von n Zahlen, Mergesort, Quicksort, Heapsort
$O(n^2)$	Einfaches Sortieren von n Zahlen, Selectionsort, Insertionsort, Bubblesort
$O(n^3)$	Matrizenmultiplikation
$O(2^n)$	brute-force algorithms, Ausprobieren von Kombinationen, Fibonacci rekursiv, Traveling Salesman
O(n!)	Durchsuchen aller Permutationen mit n Elementen

3. Komplexitätsklassen und Algorithmen

Wie sieht nun der Zeitaufwand für einige Komplexitätsklassen aus?

Beispiel: Ausführungszeiten (unter der Annahme, 1 Operation kostet 1 sec)

$f(n)$	Bezeichnung	$f(10)$	$f(100)$	$f(1.000)$	$f(10.000)$
1	konstant	1sec	1sec	1sec	1sec
$\log_2 n$	logarithmisch	3sec	7sec	10sec	13sec
n	linear	10sec	100sec	17min	2,8hours
$n \log_2 n$	log-linear	33sec	11min	2,8hours	1,5 days
n^2	quadratisch	100sec	2,7hours	11,5days	3,2years
n^3	kubisch	17min	11,5days	31,7years	31.710years
2^n	exponentiell	17min	$>$ als Alter des Universums		

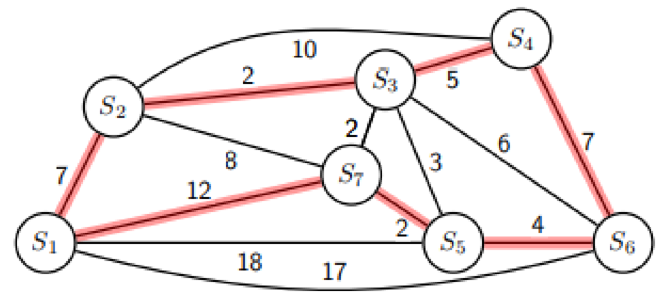
Quelle: https://www.sosy-lab.org/Teaching/2018-WS-InfoEinf/vorlesungsfolien/10-komplexitaet-4s.pdf, Seite 18, 04.01.2025

Hinweis: Ziehen Sie Parallelen zu unserem rekursiven Fibonacci-Algorithmus in Übung 8. Ab wann können wir dort nicht mehr auf ein Ergebnis warten?

3. Komplexitätsklassen und Algorithmen

Beispiel: Problem des „Handelsreisenden“ (Traveling-Salesman-Problem, TSP)
Gegeben sei ein Graph mit n Städten und den jeweiligen Entfernungen zwischen den Städten.
Gesucht ist die kürzeste Tour, so dass alle Städte besucht werden, aber jede Stadt nur einmal.

- Für das Traveling-Salesman-Problem sind bis heute nur nicht-polynomielle Algorithmen (NP) bekannt, u.a. nicht-determininistische, d.h. man muss die richtige Lösung erraten.
- Das Traveling-Salesman-Problem ist NP-complete: Falls es **einen** polynomiellen Algorithmus zu seiner Lösung gibt, so hat **jeder** nichtdeterministisch-polynomielle Algorithmus eine polynomielle Lösung.



- Die Frage, ob ein NP-completes Problem (und damit alle) in polynomieller Zeit (P) gelöst werden kann, ist eine der ungelösten Fragen der Informatik: **P = NP ?**
- Das P-NP-Problem ist eines der sieben Millenium-Probleme des Clay Mathematics Institute (CMI) in Cambridge. Wenn Sie eines dieser Probleme lösen, erhalten Sie eine Menge Geld.

Quelle: https://www.sosy-lab.org/Teaching/2018-WS-InfoEinf/vorlesungsfolien/10-komplexitaet-4s.pdf, Seite 19 – 20, 04.01.2025

4. Beispiele zum Abschätzen der Zeitkomplexität

Beispiel 1: Sequenz

`Laufzeit := Σ Laufzeit der einzelnen Anweisungen`

```
                                        // Aufwand
int x = 1;                              //   O(1)
int y = 2;                              // + O(1)
System.out.println(x);                  // + O(1)
System.out.println(y);                  // + O(1)
                                        __________
                                        // = O(1)
```

4. Beispiele zum Abschätzen der Zeitkomplexität

Beispiel 2: Bedingung

`Laufzeit := Aufwand für Test + max(Aufwand für A1, Aufwand für A2)`

```
                                        // Aufwand
    if (x < y)                          // O(1)
        max = y;                        // + max(O(1),
    else
        max = x;                        // + O(1))
                                        ____________
                                        // = O(1)
```

4. Beispiele zum Abschätzen der Zeitkomplexität

Beispiel 3: Schleifen

`Laufzeit := Aufwand für die innere Anweisung * Anzahl der Iterationen`

```
                                        // Aufwand
int n = 5;
int[] zahlen = new int[n];
for(int i = 0; i < n; i++)
        zahlen[i] = 0;                  // n * O(1)
                                        ____________
                                        // = O(n)
```

4. Beispiele zum Abschätzen der Zeitkomplexität

Beispiel 4: Geschachtelte Schleifen

```
Laufzeit :=
Aufwand für die innere Anweisung * Produkt der Größen aller Schleifen
```

```
                                                       // Aufwand
        int n = 5;
        int[] zahlen = new int[n];
        for(int i = 0; i < n; i++)
                for(int j = 0; j < n; j++)
                        zahlen[i, j] = 0;              // n * n * O(1)
                                                       _____________
                                                       // = O(n²)
```

4. Beispiele zum Abschätzen der Zeitkomplexität

Beispiel 5: Mehrere Schleifen hintereinander

```
Laufzeit := Σ Laufzeit der einzelnen Schleifen

                                                       // Aufwand
        int n = 5;
        int[] zahlen = new int[n];
        for(int i = 0; i < n; i++)
                for(int j = 0; j < n; j++)
                        zahlen[i, j] = 0;              // n * n * O(1) = O(n²)

        for(int k = 0; k < n; i++)
                        zahlen[k, 0] = 0;              // n * O(1) = O(n)
                                                       ___________________________
                                                       // = O(n²) + O(n) = O(n²)
```

4. Beispiele zum Abschätzen der Zeitkomplexität

Weitere Konstrukte

- Bei mehreren Konstrukten bestimmt immer die maximale Komponente (das ist die mit dem höchsten Exponenten) den Gesamtaufwand.
- Auch Unterprogrammaufrufe sind als Addition bezüglich des Aufwands zu sehen.
- Rekursive Unterprogrammaufrufe gehen als Produkt in die Aufwandsberechnung ein (eine Rekursion lässt sich immer in eine Iteration überführen).

5. Aufgaben zur Zeitkomplexität

Aufgabe 1: Bestimmen Sie den Aufwand in O-Notation für folgendes Programm.

```
public static void printSumAndProduct(int[] array) {
    int sum = 0;
    int product = 1;
    for (int i = 0; i < array.length; I++) {
        sum += array[i];
    }
    for (int i = 0; i < array.length; i++) {
        product *= array[i];
    }
    System.out.println(sum + ", " + product);
}
```

Aufwand: O(n)

Quelle: [https://kodr.me/en/big-o-examples, 03.01.2025]

5. Aufgaben zur Zeitkomplexität

Aufgabe 2: Bestimmen Sie den Aufwand in O-Notation für folgendes Programm.

```
public static void printPairs(int[] array) {
    for (int i = 0; i < array.length; i++) {
        for (int j = 0; j < array.length; j++) {
            System.out.println(array[i] + ", " + array[j]);
        }
    }
}
```

Aufwand: $O(n^2)$

Quelle: [https://kodr.me/en/big-o-examples, 03.01.2025]

5. Aufgaben zur Zeitkomplexität

Aufgabe 3: Bestimmen Sie den Aufwand in O-Notation für folgendes Programm.

```
public static void printArrays(int[] arrayA, int[] arrayB) {
    for(int a : arrayA) {
        for(int b: arrayB) {
                if(a > b) {
                    System.out.println(a + ", " + b);
                }
        }
    }
}
```

Aufwand: O(dim(A) * dim(B))

Quelle: [https://kodr.me/en/big-o-examples, 03.01.2025]

5. Aufgaben zur Zeitkomplexität

Aufgabe 4: Bestimmen Sie den Aufwand in O-Notation für folgendes Programm.

```
public static void printArrays(int[] arrayA, int[] arrayB) {
    for(int a : arrayA) {
        for(int b: arrayB) {
            for (int i = 0; i < 1000000; i++) {
                System.out.println(a + ", " + b);
            }
        }
    }
}
```

Aufwand: O(dim(A) * dim(B)), weil Konstanten wegfallen

Quelle: [https://kodr.me/en/big-o-examples, 03.01.2025]

5. Aufgaben zur Zeitkomplexität

Aufgabe 5: Bestimmen Sie den Aufwand in O-Notation für folgendes Programm.

```
public static int facult(int n) {
        if (n == 0)
                return 1;
        else
                return n * facult(n - 1);
}
```

Aufwand: O(n)

Quelle: [https://kodr.me/en/big-o-examples, 03.01.2025]

5. Aufgaben zur Zeitkomplexität

Aufgabe 6: Bestimmen Sie den Aufwand in O-Notation für folgendes Programm.

```
public static int fibonacci(int n) {
        if (n <= 1) return 1;
        return fibonacci(n-1) + fibonacci(n-2);
}
```

Aufwand: $O(n^2)$

6. Animation von Sortieralgorithmen

Schauen Sie sich bitte hierzu an:

https://www.toptal.com/developers/sorting-algorithms

7. Weiterführende Literatur

[Hopcroft et al., 2011] John E. Hopcroft, Rajeev Motwani , et al.
Einführung in Automatentheorie,
Formale Sprachen und Berechenbarkeit
Pearson Studium – IT, 1. März 2011

[Schöning et al., 2020] Uwe Schöning, Arne Meier, et al
Komplexität von Algorithmen: Mathematik für Anwendungen Band 4
Lehmanns Media, 6. Juli 2020