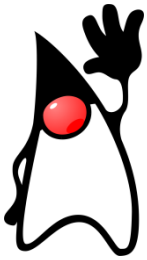
Informatik

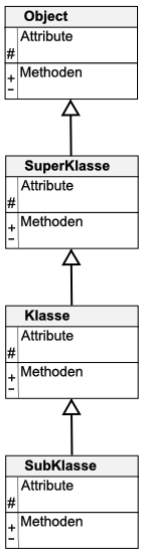
für die Sekundarstufe II

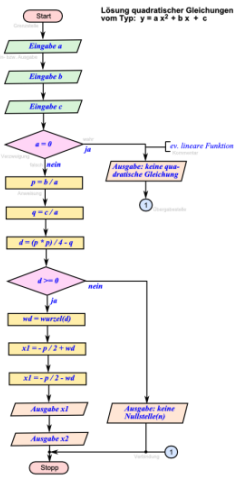
- Strukturierte und Objekt-orientierte Programmierung mit JAVA -

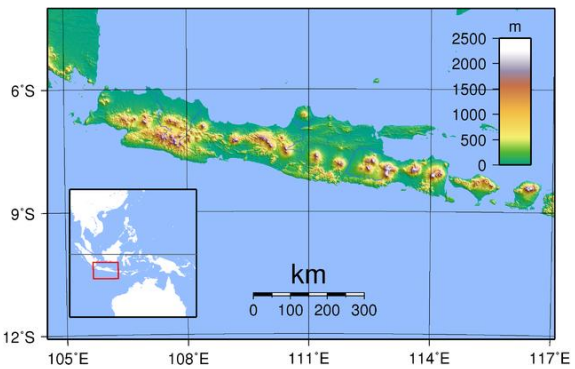
Autor: L. Drews

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









Bilder-Q: duke.dev.java.net (wave); de.wikipedia.org (interpixel.de; Sadalmelik; Köhler - Medizinalpflanzen)

teilredigierteVersion 0.7b (2022)

(c,p) 2017-2022 lern-soft-projekt: drews

Legende:
mit diesem Symbol werden zusätzliche Hinweise, Tips und weiterführende Ideen gekennzeichnet

Nutzungsbestimmungen / Bemerkungen zur Verwendung durch Dritte:

(1) Dieses Skript (Werk) ist zur freien Nutzung in der angebotenen Form durch den Anbieter (lern-soft-projekt) bereitgestellt. Es kann unter Angabe der Quelle und / oder des Verfassers gedruckt, vervielfältigt oder in elektronischer Form veröffentlicht werden.

(2) Das Weglassen von Abschnitten oder Teilen (z.B. Aufgaben und Lösungen) in Teildrucken ist möglich und sinnvoll (Konzentration auf die eigenen Unterrichtsziele, -inhalte und -methoden). Bei angemessen großen Auszügen gehört das vollständige Inhaltsverzeichnis und die Angabe einer Bezugsquelle für das Originalwerk zum Pflichtteil.

(3) Ein Verkauf in jedweder Form ist ausgeschlossen. Der Aufwand für Kopierleistungen, Datenträger oder den (einfachen) Download usw. ist davon unberührt.

(4) Änderungswünsche werden gerne entgegen genommen. Ergänzungen, Arbeitsblätter, Aufgaben und Lösungen mit eigener Autorenschaft sind möglich und werden bei konzeptioneller Passung eingearbeitet. Die Teile sind entsprechend der Autorenschaft zu kennzeichnen. Jedes Teil behält die Urheberrechte seiner Autorenschaft bei.

(5) Zusammenstellungen, die von diesem Skript - über Zitate hinausgehende - Bestandteile enthalten, müssen verpflichtend wieder gleichwertigen Nutzungsbestimmungen unterliegen.

(6) Diese Nutzungsbestimmungen gehören zu diesem Werk.

(7) Der Autor behält sich das Recht vor, diese Bestimmungen zu ändern.

(8) Andere Urheberrechte bleiben von diesen Bestimmungen unberührt.

Rechte Anderer:

Viele der verwendeten Bilder unterliegen verschiedensten freien Lizenzen. Nach meinen Recherchen sollten alle genutzten Bilder zu einer der nachfolgenden freien Lizenzen gehören. Unabhängig von den Vorgaben der einzelnen Lizenzen sind zu jedem extern entstandenen Objekt die Quelle, und wenn bekannt, der Autor / Rechteinhaber angegeben.

public domain (pd) — Zum Gemeingut erklärte Graphiken oder Fotos (u.a.). Viele der verwendeten Bilder entstammen Webseiten / Quellen US-amerikanischer Einrichtungen, die im Regierungsauftrag mit öffentlichen Mitteln finanziert wurden und darüber rechtlich (USA) zum Gemeingut wurden. Andere kreative Leistungen wurden ohne Einschränkungen von den Urhebern freigegeben.

gnu free document licence (GFDL; gnu fdl)

creative commens (cc)

BY: od. neu … Namensnennung

… nichtkommerziell

… in der gleichen Form

… unter gleichen Bedingungen

Die meisten verwendeten Lizenzen schließen eine kommerzielle (Weiter-)Nutzung aus!

Bemerkungen zur Rechtschreibung:
Dieses Skript folgt nicht zwangsläufig der neuen **ODER** alten deutschen Rechtschreibung. Vielmehr wird vom Recht auf künstlerische Freiheit, der Freiheit der Sprache und von der Autokorrektur des Textverarbeitungsprogramms microsoft ® WORD ® Gebrauch gemacht.
Für Hinweise auf echte Fehler ist der Autor immer dankbar.

Inhaltsverzeichnis:

Seite

0. Einleitung / Vorwort 8

1. Einstieg und Grundlagen 12

1.1. Programmier-Werkzeuge für JAVA 13

1.1.0. Grundlagen und Geschichtliches 13

1.1.0.1. Grundkonzepte und Ziele bei der Entwicklung von JAVA 14

1.1.0.2. Spezielles zum JDK 16

1.1.1. didaktische Programmier-Werkzeuge für JAVA 17

1.1.1.1. Greenfoot 17

1.1.1.2. BlueJ 17

1.1.2. klassische Programmier-Werkzeuge (Editor-Compiler-Systeme) für JAVA 18

1.1.2.1. Java-Editor 19

1.1.2.1.1. der Java-Editor als Editor-Compiler-System 21

1.1.2.1.2. der Java-Editor als UML-Editor 22

1.1.2.1.3. der Java-Editor als Struktogramm-Editor 22

1.1.2.1.4. der Java-Editor als Interpreter 23

1.1.2.2. jEdit 24

1.1.2.3. jGRASP 25

1.1.2.4. Eclipse – die Universal-IDE 26

1.1.2.5. NetBeans – die Haus-eigene IDE 29

1.2. Erst-Kontakt 30

1.2.1. Das Grund-Konstrukt eines JAVA-Programms. 30

1.2.2. Gestaltung von Text-Ausgaben 32

1.3. Immer Schritt für Schritt 33

1.3.1. Texte verknüpfen / konkatenieren 33

1.3.2. Zahlen und einfache Berechnungen ausgeben 36

1.3.3. Texte und Zahlen / Terme gemeinsam ausgeben 37

1.3.4. Variablen 38

1.3.5. kleine Ausgabe-Hilfsmittel 44

1.3.6. Eingaben in Konsolen-Programmen 48

1.3.6.1. Eingaben für Konsolen-Programme im Java-Editor 51

1.3.7. die JAVA-Datentypen 52

1.3.7.0. Daten-Typen für Freaks 53

1.3.7.0.1. Ganzzahlen 54

Exkurs: Absturz der ersten Ariane-5-Rakete 55

1.3.7.0.2. Gleit- oder Fließkommazahlen 56

Exkurs: Rundungsfehler 56

1.3.7.0.3. Einzelzeichen - Charakter 56

1.3.7.0.4. Zeichenketten - Strings 57

1.3.7.0.5. Zählen der Speicherzellen (Dualzahlen und ihre Skalierung) 57

1.3.7.1. Typ-Umwandlungen (Type Casting und Parsing) 59

1.3.7.3. Erzeugen von Zufallszahlen 62

1.3.8. Arbeiten und Anzeigen mit Wahrheits-Werten 64

1.3.7.1. weitere logische Operatoren 66

1.3.9. Programm-Strukturen / Algorithmen-Strukturen 67

1.3.9.1. Sequenzen 68

1.3.9.2. Verzweigungen 69

1.3.9.3. Schleifen 79

1.3.9.3.1. bedingte Schleifen 79

1.3.9.3.2. Zähl-Schleifen 83

1.3.9.4. Kommentare und javadoc 87

1.3.10. einfache Datenstrukturen 91

1.3.10.1. primitive Array's 91

1.3.10.1.1. erweiterte for-Schleifen 93

1.3.11. JAVA-Schlüsselwörter (erster Grundwortschatz) ... 94
1.4. von Anfang an: Objekt-orientierte Programmierung ... 99
1.4.0. Grundlagen ... 99
Definition(en): Objekt ... 100
Definition(en): Klasse ... 100
Definition(en): Instanz ... 103
Definition(en): Attribut ... 105
Definition(en): Methode ... 105
1.4.1. Objekt-Datentypen ... 107
1.4.2. Vererbung ... 110
Definition(en): Vererbung ... 111
1.4.2.1. Überschreiben von Methoden (Override) ... 115
1.4.2.1.1. sicheres Vergleichen ... 117
1.4.3. Konstruktoren ... 118
Definition(en): Konstruktor ... 119
1.4.3.1. kleine Zusammenfassungen und Übersichten ... 121
1.4.4. Pakete (Package) ... 122
Definition(en): Package ... 122
1.4.5. Kapselung ... 123
Definition(en): Kapselung ... 123
Exkurs: Verlust des Mars Climate Orbiters ... 124
1.4.6. Daten-Übergaben - Argumente und Parameter ... 126
Definition(en): Argumente ... 126
Definition(en): Parameter ... 129
1.4.7. Gültigkeit und Sichtbarkeit von Variablen – Scope ... 130
1.4.8. Warum also?: public static void main(String[ ] args) { … } ... 131
1.5. weitere Schritte für Erfahrene ... 132
1.5.1. Schleifen für Erfahrene ... 132
1.5.1.1. Manipulation von Schleifen(-Durchläufen) ... 132
1.5.1.1. Iteratoren ... 132
1.5.2. Exzeptions (exception's) ... 134
1.5.3. Datenstrukturen für Erfahrene ... 135
Definition(en): Datenstrukturen ... 135
Definition(en): Kollektionen (Sammlungen) ... 135
1.5.3.1. Erzeugen / Anlegen von Objekten mit Objekt-Datentyp (Wrapper-Objekte) 136
1.5.3.2. Kollektionen ... 137
1.5.3.2.1. Objekt-Felder (ArrayList) ... 138
1.5.3.2.2. Objekt-Listen (verkettete Listen, LinkedList) ... 141
1.5.3.2.3. Mengen (HashSet, Set) ... 144
1.5.3.2.4. Wörterbücher (HashMap; dictonary; Map) ... 146
Definition(en): Map ... 147
Übersicht(en) zu Kollektionen ... 149
1.5.4. zusätzliche Schleifen-Strukturen für einfache Felder und Kollektionen ... 150
1.5.4.1. Schleifen ohne Lauf-Variablen ... 150
1.5.4.2. fortlaufend iterierte Schleifen ... 150
Exkurs: Wiederholung for-Schleife mit Lauf-Variable ... 151
1.5.4.3. Iterationen mit foreach ... 152
1.5.5. Lösungs-Strategien → Rekursion und / oder Iteration? ... 153
1.5.5.1. Iteration ... 154
Definition(en): Iteration ... 155
1.5.5.1.1. typische Iterations-Anwendungen ... 156
1.5.5.1.1.1. Summen-Bildung ... 156
1.5.5.1.1.2. Produkt-Bildung ... 157
1.5.5.2. Rekursion ... 159
Definition(en): Rekursion ... 161
1.5.5.2.1. weitere typische Anwendungen für Rekursionen ... 164
1.5.5.2.1.1. Überführung einer Dezimal-Zahl in eine Dual-Zahl ... 164

1.5.5.2.1.2. die Fakultät ... 164
1.5.5.2.1.3. die FIBONACCHI-Folge ... 165
Exkurs: FIBONACCHI ohne die Vorglieder? ... 166
1.5.5.2.1.4. das ggT – der Größte gemeinsame Teiler ... 167
1.5.5.2.1.5. Erkennung von Palindromen ... 168
1.5.5.2.1.x. weitere klassische Rekursions-Probleme ... 169
1.5.5.2.2. direkte Gegenüberstellung von interativen und rekursiven Algorithmen .. 184
1.5.5.2.2.1. GGT – größter gemeinsamer Teiler ... 184
1.5.5.2.2.2. Palindrom-Prüfung ... 185
1.5.5.2.2.2. Potenz-Prüfung ... 185
1.5.5.2.2.3. Fakultät ... 186
1.5.5.2.2.4. FIBONACCHI-Folge ... 186
1.5.6. String-Methoden ... 187
1.5.7. Laufzeit- und Speicher-Effizenz ... 189
Anlegen von Variablen ... 189
mehrfacher Aufruf von gleichen Methoden ... 190
Short Circuit Evaluation (logische (Kurz-(Schlüsse)) ... 191
1.5.8. JAVA-Schlüsselwörter (erweiteter Wortschatz) ... 192
1.5.9. komplexe Programmier-Aufgaben: ... 196
1.6. Objekt-orientierte Programmierung für Erfahrene ... 198
1.6.1. Destruktoren ... 198
Definition(en): Destruktor ... 198
1.6.2. Finalisierung ... 199
1.6.3. Überladen von Methoden (Overload(ing)) ... 199
Definition(en): Methoden-Deklaration ... 200
Definition(en): Methoden-Signatur ... 200
1.6.4. Abstrakte Klassen ... 201
1.6.5. Polymorphie ... 203
1.6.6. Mehrfach-Vererbung (Interface's, Schnittstellen) ... 204
1.7. Planen und Entwickeln größerer Programme ... 208
1.7.0. Allgemeines ... 208
1.7.1. Struktogramme ... 210
Kurzbedienungs-Anleitung: Structorizer ... 215
Kurzbedienungs-Anleitung: Java-Editor (Struktogramme) ... 217
1.7.2. UML-Diagramme ... 219
Kurzbedienungs-Anleitung: UMLed ... 219
Kurzbedienungs-Anleitung: Java-Editor (UML-Fenster und Klassen-Modellierer) ... 220
Übersicht / Legende zu UML-Diagrammen: ... 222
1.8. Nutzung graphischer Oberflächen ... 223
1.8.x. Processing Library vom MIT mit der IDE eclipse ... 223
Anzeige-Elemente ... 225
Rechtecke ... 225
Kreise / Elipsen ... 227
Anzeige-Steuerung ... 227
Tastatur und Maus ... 227
Ausgaben ... 228
Interface's und abstrakte Klassen ... 228
Dokumentation ... 229
neue Klassen ... 229
1.9. Refactoring ... 230
1.10. Coding Standard's – Konvention für ordentliche Programmierung ... 231
1.10.1 (automatische) Werkzeuge für Coding Standard's ... 231
1.11. Testen (von Programmen) ... 233

2. informatische Problemstellungen in JAVA ... 235
2.0.1. Fehler sind menschlich ... 235
2.1. Compiler und Interpreter ... 236

Exkurs: Der teuerste Bindestrich der Welt ... 237
2.1.3. JAVA als spezielle Compiler-Interpreter-Kombination ... 238
2.x. komplexe Daten-Strukturen ... 239
2.x.y. Zeiger-orientierte Daten-Strukturen ... 239
2.x.y. Warteschlangen bzw. FIFO-Speicher ... 239
2.x.y. Keller bzw. LIFO-Speicher ... 240
2.x.y. Bäume ... 242
Suchen in Binär-Bäumen ... 243
Einfügen in Binär-Bäumen ... 244
Löschen in Binär-Bäumen ... 244
balancierte Bäume ... 245
weitere Bäume ... 246
Komplexität in Bäumen im Vergleich ... 248
2.x.y. Ringe ... 248
2.x.y. Puffer ... 249
2.x.y. Graphen ... 250
2.x. Suchen ... 251
Suche in (un)sortierten Listen ... 251
lineare Suche ... 251
lineare Suche mit Positions-Bestimmung und Abbruch ... 251
Suche in Kollektionen ... 252
Suche in sortierten Listen ... 252
binäre Suche (nach Divide-and-Conquer-Prinzip) ... 252
Problem Vergleichen ... 254
2.x. Sortieren ... 257
Selection-Sort ... 257
Bubble-Sort ... 258
Quick-Sort ... 260
Merge-Sort ... 262
2.x. weitere Sortier-Verfahren in der Kurz-Vorstellung ... 264
2.x.y. Radix-Sort ... 264
2.x.y. Insertation-Sort ... 264
2.x.y. Counting-Sort ... 265
2.x.y. SHELL-Sort ... 265
2.x.y. Heap-Sort ... 266
2.x.y. Smooth-Sort ... 266
2.x.y. Bogo-Sort ... 267
2.x.y. ???-Sort ... 267
2.x.y. historische Entwicklung der Sortier-Algorithmen ... 269
2.x. weitere Themen ... 270
2.x.y. nebenläufige Prozesse ... 270
2.x. Betrachtungen zur Effektivität von Algorithmen ... 271

3. Projekte ... 275
1. Beschreibung des Projekt-Thema's ... 275
2. Erstellen eines Pflichten-Heftes ... 275
3. Erstellen von Visualisierungen zur Programm-Struktur ... 276
3.x. Hand-Zeichnungen, Skizzen, Brainstorming ... 276
3.x. CRC-Karten ... 276
3.3. UML-Diagramme ... 277
3.x. Planung, Entwicklung und Visualisierung von Algorithmen ... 279
4. Codierung ... 280
4.x. Versions-Verwaltung ... 280
5. Dokumentation ... 281
6. Abschluß (Vorstellung / Präsentation / Projekt-Auswertung) ... 281
7. Verteilung ... 281

Literatur und Quellen: ... 282

0. Einleitung / Vorwort

JAVA ist sehr komplex und auf den ersten Blick übermächtig. Für Anfänger ist es erfahrungsgemäß schwierig einen unbekümmerten Einstieg zu finden. Wir werden hier zuerst einige Dinge hinnehmen und es einfach so machen. Später werden diese Vorgaben dann genauer erklärt.

Die großen Themen "Objekt-Orientierung" und die "Sprache JAVA" werden in einzelnen Linien besprochen. So bleiben wir thematisch immer ein bißchen zusammen und es entsteht nicht ein heilloses Durcheinander. Andernfalls müsste man mehrere – eigentlich recht viele – Sachverhalte von verschiedenen Themen-Kreisen parallel abhandeln, was der Verständlichkeit eher abträglich ist. Ein Fernseher mit Fernbedienung kann man auch schnell und effektiv nutzen, wenn man bestimmte Knöpfe und Bedienschritte auswendig lernt. Um Fernsehen zu schauen, muss man nicht verstehen, was jeder Knopf genau macht und wie der Fernseher intern funktioniert. Für ein tiefgreifenderes Verständnis aller Funktionen ist dann natürlich ein immer tieferes Eindringen notwendig. Irgendwann übersteigt aber die Tiefe des Eindringens die Möglichkeiten von schulischen Kursen. Spätestens ab hier wird man zu Spezial- oder echter Fach-Literartur übergehen müssen.
Die Linien sind nach Anspruch in große Abschnitte geteilt. Damit können zuerst einmal die Grundlagen gelegt werden. Später werden wir dann immer tiefer eingedringen.
In den Grundlagen-Kapiteln werden wir Quell-Texte usw. noch durch das Syntax-Highligthing besser lesbar machen. Da das aber (beim derzeitigen Stand der Editoren und Textverarbeitungs-Systeme) sehr aufwändig ist, werden wir darauf bei den fortgeschrittenen Themen immer mehr verzichten. (Wir warten auf den Tag, wo die Editoren ihre gehighligtheten Quell-Texte auch so den Text-Verarbeitungen zur Verfügung stellen.)

Die anderen informatischen Konzepte, die in der Sekundarstufe II eine Rolle spielen, wie UML und theoretische Probleme, werden wir teilweise mit einstreuen oder ihnen eigene kleine Unterkapitel gönnen. Wenn diese Themen für den Leser in seinem aktuell besuchten Kurs keine Rolle spielen, kann er sie lässig übergehen.

Ich sehe den Kurs als Konstrukt aus 4 Teilen. Der erste Teil richtet sich an den JAVA-Einsteiger. Für ihn sind die Abschnitte → 1.2. Erst-Kontakt, → 1.3. Immer Schritt für Schritt und → 1.4. von Anfang an: Objekt-orientierte Programmierung gedacht. Ein bestimmtes Hin-und-her ist wahrscheinlich nicht zu vermeiden, da die Thematiken eng miteinander verbunden sind. Ich kann mich aber von einer gewissen Systematik nicht trennen. Die hier besprochenen Sachverhalte und Arbeits-Techniken sind Voraussetzung für eine tiefergreifende Auseinandersetzung mit Datenstrukturen und Objekten. Die Abschnitte → 1.5. weitere Schritte für Erfahrene und → 1.6. Objekt-orientierte Programmierung für Erfahrene richten sich an den fortgeschrittenen JAVA-Programmierer.
Die Kapitel bieten diverse Möglichkeiten sich intensiver mit Software-Entwicklung und im Speziellen mit der Objekt-orientierten Programmierung zu beschäftigen. In diesen Bereich gehören auch die Kapitel → 1.7. Planen und Entwickeln größerer Programme, → 1.8. Nutzung graphischer Oberflächen, → 1.9. Refactoring, → 1.10. Coding Standard's – Konvention für ordentliche Programmierung und → 1.11. Testen (von Programmen) hinein.
Der dritte Abschnitt verbindet die praktische Programmierung mit speziellen informatischen Thematiken. Hier muss sicher ausgewähl werden. Für die Freaks gibt es aber auch viel Stoff, den sie eigenständig bearbeiten können (→ 2. informatische Problemstellungen in JAVA).
Am Schluß folgt in diesem Skript die Projekt-Abteilung → 3. Projekte. Das Arbeiten hier ist von starker Selbstständigkeit geprägt. Ein Weiterentwickeln des eigenen Projektes zuhause wird wahrscheinlich nötig sein. Einzelne Abschnitte können aber auch immer wieder parallel in den Kurs eingebunden werden. Oft ergibt sich das Situations-bedingt.
Sehen Sie das Ganze nicht zu streng. Manchmal ergibt sich ein inhaltlicher Bedarf aus irgendwelchen Unterrichts-Situationen. Oft sind die Texte in den Kapiteln so geschrieben,

dass ein relativ einfacher Zwischen-Einstieg möglich ist. Deshalb wird der aufmerksame Leser auch viele Wiederholungen finden. Was für diese Leser störend wirkt, ist für andere Kurs-Teilnehmer eher sinnvoll. Es ist schwierig alle gleichermaßen zufrieden zu stellen.
Auch einige ausgewählte Themen oder Sachverhalte werden mehrfach und an verschiedenen Stellen im Skript auftauchen. Dies liegt einfach an der starken Verzahnung der Themen. Querverbindungen sind weitesgehend als Link's (➔ Verknüpfungen) angegeben. Je nach Dateiform funktionieren diese dann auch zumindestens auf Computern. In der Papierform müssen Sie sich an den Begriffen und Überschriftennummern orientieren. Andere Skripe werden mit einem Buch-Symbol und einem Kurznamen oder dem Titel gekennzeichnet (🕮 **Datenbanken**).

Als Arbeits-Empfehlung gebe ich meinen Schülern immer auf den Weg, sich kleine Code-Schnipsel oder Syntax-Notierungen auf einem Zettel ("Spicker") oder in einem "kleinen eigenen Tafelwerk" zu notieren. Was man gesehen (/ gelesen), gehört, selbst geschrieben und dann noch übend gebraucht hat, erzielt den höchsten Lern-Effekt. Man schreibt sich genau das auf, womit man die meisten Schwierigkeiten hat. So hat man ein Mittel zum schnellen Nachschlagen. Beim Üben / Lösen der Aufgaben ist das Benutzen der selbstgeschrieben Hilfen kein Problem, in Klausuren, Leistungskontrollen etc. sind sie i.A. nicht erlaubt. Hier gelten die Festlegungen des Kursleiters oder der Bildungseinrichtigung.

In anderen JAVA-Tutorials, -Büchern, -Skripten usw. usf. finden Sie andere Herangehensweisen und Bearbeitungs-Strategien. Wenn Ihnen die meine nicht so liegt, dann kann ich die Bücher aus der Literatur-Liste empfehlen. Sie sind sehr umfangreich und brilliant konzipert und geschreiben.
Die jenigen, die nur noch das Internet kennen und Papier so gar nicht mehr anfassen wollen, können natürlich die folgenden Links nutzen. Manche entsprechen den hinten gelisteten Büchern.

Links:

http://openbook.rheinwerk-verlag.de/javainsel/index.html (OpenBook: ULLENBOOM, Christian: Java ist auch eine Insel; Rheinwerk Computing; ISBN 978-3-8362-1802-3)

Niveau Stufe	rote Linie "Sprache JAVA"	rote Linie "Objekt-Orientierung"	weitere informatische Konzepte / Stichpunkte / Aspekte
Basis Grundlagen			1. Einstieg und Grundlagen
Einsteiger Anfänger	1.2. Erst-Kontakt 1.3. Immer Schritt für Schritt	1.4. von Anfang an: Objekt-orientierte Programmierung	
Erfahrener	1.5. weitere Schritte für Erfahrene 1.5.1. Schleifen für Erfahrene 1.5.2. Exzeptions (exception's) 1.5.3. Datenstrukturen für Erfahrene 1.5.4. zusätzliche Schleifen-Strukturen für einfache Felder und Kollektionen 1.5.5. Lösungs-Strategien (Rekursion und / oder Iteration? 1.5.6. String-Methoden 1.5.7. Laufzeit- und Speicher-Effizenz	1.6. Objekt-orientierte Programmierung für Erfahrene 1.6.2. Finalisierung 1.6.3. Überladen von Methoden (Overload(ing)) 1.6.4. Abstrakte Klassen 1.6.5. Polymorphie 1.6.6. Mehrfach-Vererbung (Interface's, Schnittstellen)	1.5.5. Lösungs-Strategien (Rekursion und / oder Iteration? 1.5.7. Laufzeit- und Speicher-Effizenz
Fortgeschrittener	1.7.1. Struktogramme 1.8. Nutzung graphischer Oberflächen	1.7.2. UML-Diagramme 1.8. Nutzung graphischer Oberflächen	1.7. Planen und Entwickeln größerer Programme 1.7.1. Struktogramme
erweitertes / herausragendes Niveau			1.9. Refactoring
Basis Grundlagen	1.10. Coding Standard's – Konvention für ordentliche Programmierung		

(c,p) 2017-2022 lern-soft-projekt: drews

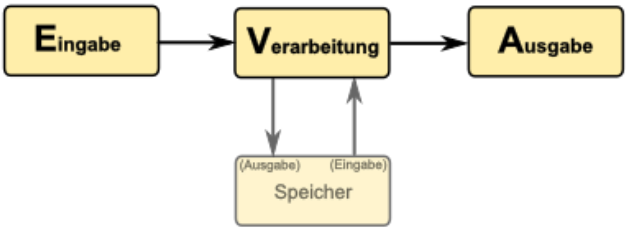
Niveau Stufe	Linie "Sprache JAVA"	Linie "Objekt-Orientierung"	weitere informatische Konzepte / Stichpunkte / Aspekte
Erfahrener	1.11. Testen (von Programmen)		2. informatische Problemstellungen in JAVA 2.1. Compiler und Interpreter
Fortgeschrit-tener	2.x. komplexe Daten-Strukturen 2.x. Sortieren 3. Projekte	3. Projekte	2.x. Sortieren

1. Einstieg und Grundlagen

einfache Programmierung realisert nur das EVA-Prinzip
für einfach Aufgaben reicht das auch, aber für moderne Problem-Bearbeitungen ist diese Programmierung nur noch als innerer Bestandteil zu gebrauchen

Wir brauchen mächtige Techniken, die sich an unserer modernen Lebenswelt orientieren.

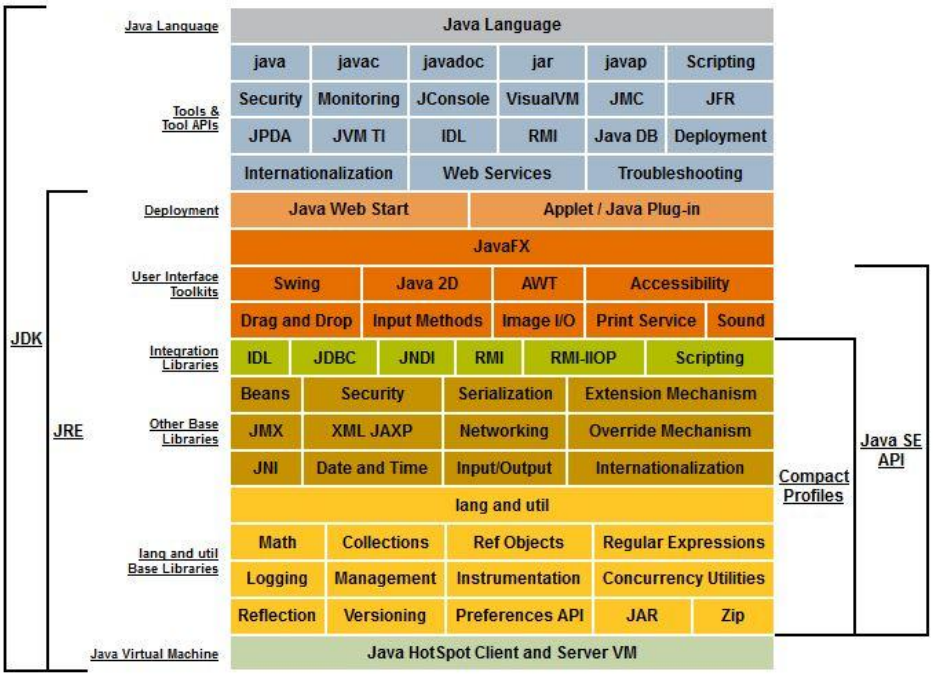
In dieser Welt sind viele Dinge (Objekte) vorhanden, die miteinander agieren. Dazukommen Prozesse, die nebenläufig sind und Modelle, die stark vernetzt sind.



1.1. Programmier-Werkzeuge für JAVA

1.1.0. Grundlagen und Geschichtliches

In vielen Programmiersprachen ist die Programm-Entwicklung primär an eine Software angelehnt. JAVA ist in einigen Punkten – was zumindestens die Programm-Entwicklung angeht – sehr frei angelegt. Was man in jedem Fall braucht ist eine Software-Umgebung, die mit JAVA-Quelltexten oder vorübersetzten Programmen klar kommt.
Ursprünglich (1995) war JAVA eine Entwicklung von Sun Microsystems (kurz: Sun). Es gab dann auch eine version von microsoft ®, die aber in vielen Details ein Extrasüppchen kochen wollten. Mit seiner proprietären Version wollte microsoft ® seine Markt-beherrschende Stellung bei anderer Software auch auf den JAVA-Markt ausdehnen. Der Plan ist aber nicht aufgegangen. Im Jahre 2010 kaufte die Datenbank-Firma oracle ® JAVA von Sun. Heute stehen mehrere Versionen zur Nutzung bereit, die sich nach dem Nutzungs-Zweck unterschiedlich umfangreich gestalten.
Sun hat vor dem Verkauf von JAVA das Gesamtsystem unter die GNU General Public License gestellt. Damit ist es weltweit frei verfügbar und nutzbar.



JAVA-Schichten und Komponenten
Q: oracle.com

Die Runtime-Version (Java Runtime Evironment; JRE) wird zum Benutzen von JAVA-Programmen – erkenntlich an der Dateityp-Kenung .jar – benötigt. Auf vielen Rechnern ist dieses Programm schon installiert. Heute gibt es für alle wichtigen Prozessor-Systeme (intel ®, ARM, …) und die gängigen Betriebssysteme (Windows ®, Linux, MacX, OS/2, Solaris, …) JRE-Versionen.
Das Runtime Environment umfasst die JAVA-Virtual-Machine (virtuelle JAVA-Maschine) als ByteCode-Interpreter und die JAVA-API (Programmier-Schnittstelle mit vielen Bibliotheken).
Praktisch ist die JRE Anwender-orientiert und dient nur zum Ausführen von JAVA-Programmen auf dem Zielrechner.
Der Entwickler braucht die vollständige JAVA-Variante, die sich JDK (Java Development Kit) nennt. Sie einthält als zentralen Teil den JAVA-Compiler (javac), mit dem ein Entwicklung

aus dem Quellcode die verteilbaren .jar-Dateien machen kann. Die Details erläutern wir später genauer (→ 2.1.3. JAVA als spezielle Compiler-Interpreter-Kombination). Für die Programmierer ist auch die Dokumentation zur Programmiersprache (Javadoc) wichtig. JAVA ist derzeit schon so umfangreich, dass niemand mehr alle Teile kennen kann. Zur Information über benötigte Teile benutzt man dann die Dokumentation.
Für eine frühe Programmierung in der Schule besonders geeignet sind Programmier-Systeme, bei denen die Programmierung quasi so nebenbei eingeführt wird. Ich nennen sie hier einfach mal didaktische Programmier-Werkzeuge. Ihnen stehen die klassischen Editoren und Compiler (oder Interpreter) gegenüber.
Das aus meiner Sicht beste Konzept bei den didaktischen Werkzeugen verfolgt hier derzeit Greenfoot mit einigen hochgesteckten Zielen und z.T. revolutionären neuen Techniken.
Zu dieser Gruppe von Programmen zählen auch KaraJAVA und BlueJ.
BlueJ ist schon eher eine klassische Entwicklungs-Oberfläche. Man spricht auch von einer IDE (Integrated Development Enviroment). In BlueJ werden Programme auf der Basis von UML-ähnlichen Strukturen entwickelt. Dazu später mehr.
Der leichte Einstieg ist aber auch zugleich ein großes Problem. Die – eigentlich als Bannbrechende Einstiege gedachten Konzepte – werden von vielen Einsteigern schnell allzu erst genommen. Innerhalb des Systems können sie gut programmieren, verstehen alle Techniken usw. Aber kaum soll eine Programm mit einem anderen – genauso leicht zu bedienenden – allgemeinen Programmier-System erstellt werden, sind die Sprach-Konzepte und Programmier-Kenntnisse wie weggeblasen. Vielfach muss wieder völlig von vorne begonnen werden. Der leichte Einstieg entpuppt sich als relativ große Zeit-Verschwendung.
In jedem Fall eigenen sich die didaktischen Editoren für einen Einstieg in die Welt der Programmierung. Und wenn zumindestens ein Gefühl für Programmierung als moderne Kulturtechnik bleibt, ist es ein akzeptabler Erfolg. Wir wollen hier aber mehr erreichen!

1.1.0.1. Grundkonzepte und Ziele bei der Entwicklung von JAVA

Die angestrebte Programmiersprache sollte:

- einfach und Objekt-orientiert
- verteilt und vertraut
- robust und sicher
- Architektur-neutral und portabel
- sehr Leistungs-stark
- interpretierbar
- parallelisierbar und dynamisch

sein.

Aufgaben:

1. ***Wie ist JAVA eigentlich zu seinem Namen gekommen?***
2. ***Was umfasst die GNU General Public License? Welche Möglichkeiten und Verpflichtungen ergeben sich daruas für Programmierer?***
3. ***Kann man mit JAVA Programme erstellen, für die andere Nutzer ev. bezahlen müssen / JAVA also kommerziell genutzt werden soll?***
4. ***Informieren Sie sich in mehreren (zu dokumentierenden) Quellen über die hinter den aufgezählten Zielen und Eigenschaften steckenden Konzepte und Inhalte von JAVA!***
5. ***Wer oder was ist Duke, über den / das im JAVA-Umfeld häufig gesprochen wird?***

1.1.0.2. Spezielles zum JDK

Begrifflichkeiten:

- **Sourcepath** — Pfad(e), in dem Quellcode abgespeichert ist (zu .java-Dateien)
- **Classpath** — Pfade zu Komponenten, die der Compiler benutzen soll (zu .class-Dateien)
-

wenn man mehrere Pfade angeben will / muss, dann werden die Semikolon-getrennt notiert

mit Java Archives lassen sich vom der JRE ausführenbare Pakete erzeugen (.jar-Dateien)
man benötigt eine sogenannte Manifest-Datei, dieses ist eine Text-Datei mit Hilfs-Information

Mit dem JDK und seinen Komponenten kann man auch Komandozeilen-orientiert arbeiten. Das ist für kleine Prograämmchen gut machbar.

Entwicklungsschritte auf der Kommandozeile

1. **Quellcode erzeugen** — mit einem Editor Quelltext erzeugen und z.B. in Datei **`ProgrammX.java`** abspeichern
2. **Compilieren in Bytecode** — Komilieren mit: (mit der Option d (destination) wird das Ziel für die .class-Dateien angegeben)
   `javac –d bin ProgrammX.java`
3. **Ausführen / Testen** — Ausführen des erzeugten ByteCode's:
   `java ProgrammX.class –classpath bin;↵`
   `C:\lib ProgrammX`
4. **Erstellen von verteilbaren Package's** — Erstellen eines verteilbaren Paketes:
   `jar –cfe ProgrammX.jar ProgrammX ↵`
   `ProgrammX.class`
5. **Nutzung** — Ausführen mit der JAVA-Laufzeit-Umgebung:
   `java –jar ProgrammX.jar`

↵ … Kommando geht in der nächsten Zeile weiter; Umbruch erfolgt hier nur wegen des Textverarbeitungssystems; auf die notwendigen Leerzeichen ist zu achten

1.1.1. didaktische Programmier-Werkzeuge für JAVA

1.1.1.1. Greenfoot

Greenfoot wurde und wird vom Kings College London rund um den Prof. M. KÖLLING entwickelt.
Im Vordergrund steht hier ein spielerischer und experimenteller Einstieg in die OOP.
Als Spielwiese für den Programmierer gibt es eine Miniwelt, in der mit graphischen Objekten – zuerst sind das z.B. Wombat's - agiert wird.

"Über Greenfoot"-Logo aus dem Programm

Das Programm Greenfoot ist auch als portables Programm verfügbar (Standalone zip).
Als fertiges System gibt es Greenfoot auch auf dem "normalen" IoStick (Informatik on Stick) von Timo HEMPEL.

Greenfoot benutzt einen Teilsprache von JAVA. Dieser nennt sich Stride. Es kann zwischen Java und Stride hin- und her-geschaltet werden.

Links:
https://www.greenfoot.org (Projekt-Seite)
https://tinohempel.de/info/info/IoStick/index.html (Download des IoStick)

1.1.1.2. BlueJ

Diese pädogogische Software ist einer der Klassiker unter Programmier-Werkzeugen. Es ist praktisch ein Vorgänger von Greenfood, wenn man so etwas überhaupt sagen kann. Stride ist in BlueJ ebenfalls verfügbar.
BlueJ hat die gleichen Ersteller, wie das Programm Greenfoot. Von der Struktur her orientiert sich BlueJ eher am klassichen Java-Entwickler. Die Einsteiger-Spielwiese fehlt und wird hier durch mehr Konsole ersetzt.

"Über BlueJ"-Logo aus dem Programm

Die neueren Versionen bieten jetzt auch eine sehr fortschrittliche Programmier-Oberfläche und vor allem einen Kontext-orientierten Editor.
Man kann – wie auch in Greenfood – im üblichen Java programmieren oder man nutzt die Teil-Sprache "Stride".

BlueJ ist auch als portables Programm verfügbar (Standalone zip). Diese Version bringt auch gleiche eine JAVA-Umgebung mit, so dass man sich um dessen Installation nicht mehr kümmern muss.
Als fertiges System gibt es BlueJ auch auf dem "normalen" IoStick (Informatik on Stick) sowie dem Abi-IoStick von Timo HEMPEL.

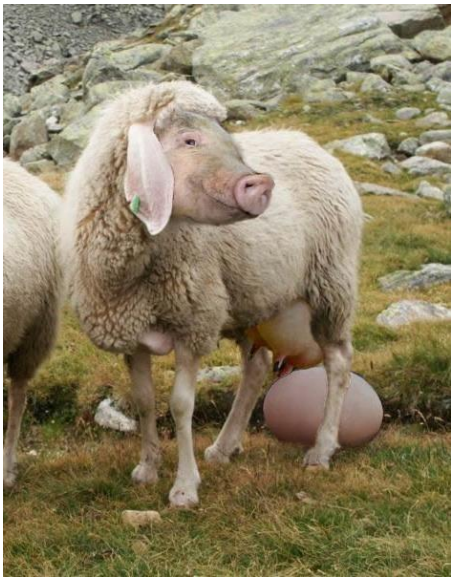
Links:
https://bluej.org/ (Projekt-Seite)
https://tinohempel.de/info/info/IoStick/index.html (Download der IoStick's)

1.1.2. klassische Programmier-Werkzeuge (Editor-Compiler-Systeme) für JAVA

Die meisten JAVA-Editoren bzw. JAVA-IDE's erwarten eine vorher installierte JAVA-Umgebung. IDE steht hier für Integrated Development Evironment (dt.: Integrierte Entwicklungs-Umgebung).
Editoren bzw. IDE's stellen "nur" praktische Werkzeuge zum Erstellen, Testen und Berichtigen von Quell-Texten dar. Die eigentliche Übersetzung des Quell-Textes in ein lauffähiges Programm erledigt die JAVA-Umgebung (JAVA-Plattform).
Eine echte IDE nimmt dem Programmierer viel Arbeit ab. In hoch-entwickelten Systemen wird Quellcode teilweise automatisch erzeugt. Vor allem betrifft es den vielfach gehassten Überfluss, den JAVA für sich selbst und aus seinem Prinzipien heraus braucht. Diese sind dem Programmierer häufig nur lässtig und für den Anfänger weitgehend unverständlich und scheinbar überflüssig.
Die JAVA-Umgebung, die wir benötigen, lässt sich auf der Website der Firma oracle ® – die diese Umgebung entwickelt und frei zur Verfügung stellt – downloaden. Wir benötigen die JDK-Version (JAVA Development Kit) für das lokale Betriebssystem.
Bei der anschließenden Installation sollten Anfänger die Optionen einfach übernehmen. Profi's können die verschiedenen Optionen natürlich ihren Bedürfnissen und Gegebenheiten anpassen.
Wer will, kann die JAVA-Umgebung für alle Nutzer installieren, was aus meiner Sicht Sinn macht. Dies ist auch notwendig, wenn man selbst nur mit einem eingeschränkten Konto an einem Rechner arbeitet und jemand anderes ("der Administrator") das Programm installieren muss.
Es ist von Vorteil sich den Installations-Pfad zu notieren, dann kann man später bei der Einrichtung der Editoren vorteilhafter arbeiten.

Links:
http://www.oracle.com/technetwork/java/javase/downloads/index.html (JAVA-Download-Seite von oracle ®)

1.1.2.1. Java-Editor



Eierlegende Wollmilchsau –
der Traum jedes Züchters
Q: commons.wikimedia.org (Georg Mittenecker)

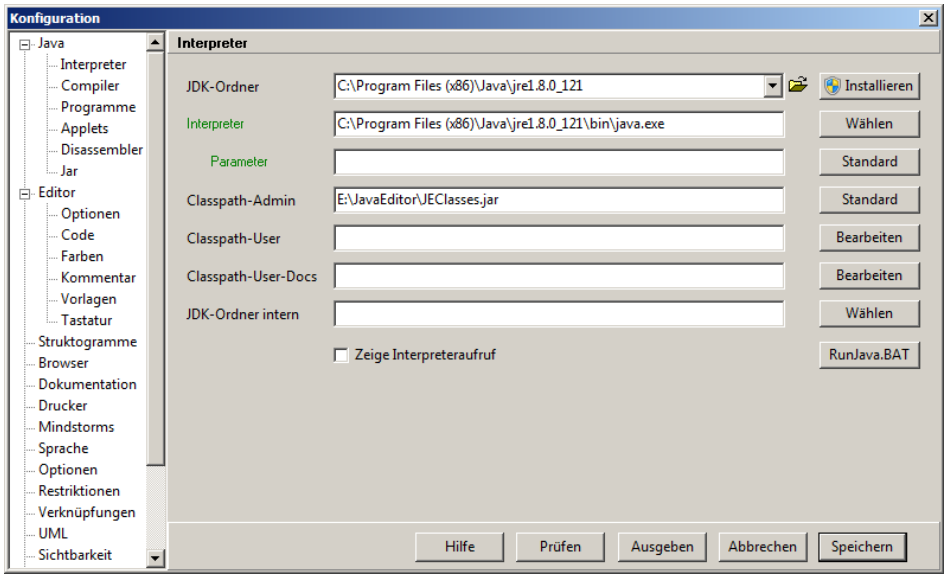
Der Java-Editor ist quasi die Eier-lengende Wollmilchsau unter den Java-fähigen Editoren. Das ist hier ganz ehrlich und ehrfürchtig gemeint. Er deckt alle Themen-Felder einer klassischen Programmier-Ausbildung auf schulischem Niveau ab. Man kann mit ihm die Quell-Texte schreiben und gleich ausprobieren. Ein Umschalten zwischen Editor und Compiler ist nicht notwendig. Alle notwendigen Arbeiten übernimmt der Java-Editor.
Zum Erlernen von Programmiersprachen-unabhängigen Algorithmen-Darstellungen werden Struktogramme favorisiert. Diese sind für den Java-Editor kein Problem. Per drag and drop lassen sich ordnungsgemäße Struktogramme schnell zusammenstellen.
Objekt-orientierte Modellierung mit Hilfe von UML-Diagramme ist ebenfalls ein wichtiger Unterrichts-Gegenstand. Auch diese bekommt der Java-Editor super gehändelt.
Somit haben wir mit dem Java-Editor wirklich eine Werkzeug-Sammlung, die eine breitgefächerte Bearbeitung von informatischen Problemen mit nur einem Programm ermöglicht.

Download und Installation verlaufen ganz klassisch. Wer will kann den Pfad für die Installation anpassen. In den meisten Fällen ist das aber nicht notwendig.
Wenn man die JAVA-Umgebung nicht schon vorher installiert hat, dann erscheint nach der Installation gleich das Konfigurations-Fenster mit zwei roten Zeilen. Die Installation der JAVA-Umgebung ist jetzt unbedingt nachzuholen. Die zu beachtenden Details wurden oben erwähnt. Bitte nicht ungeduldig werden, der riesige Download muss erst entpackt und vorbereitet werden.
Nach dem Installieren muss man unserem Java-Editor noch den richtigen Ordner mitteilen. Dazu verwendet man den Ordner-Öffnen-Button hinter der "JDK-Ordner"-Eingabezeile.

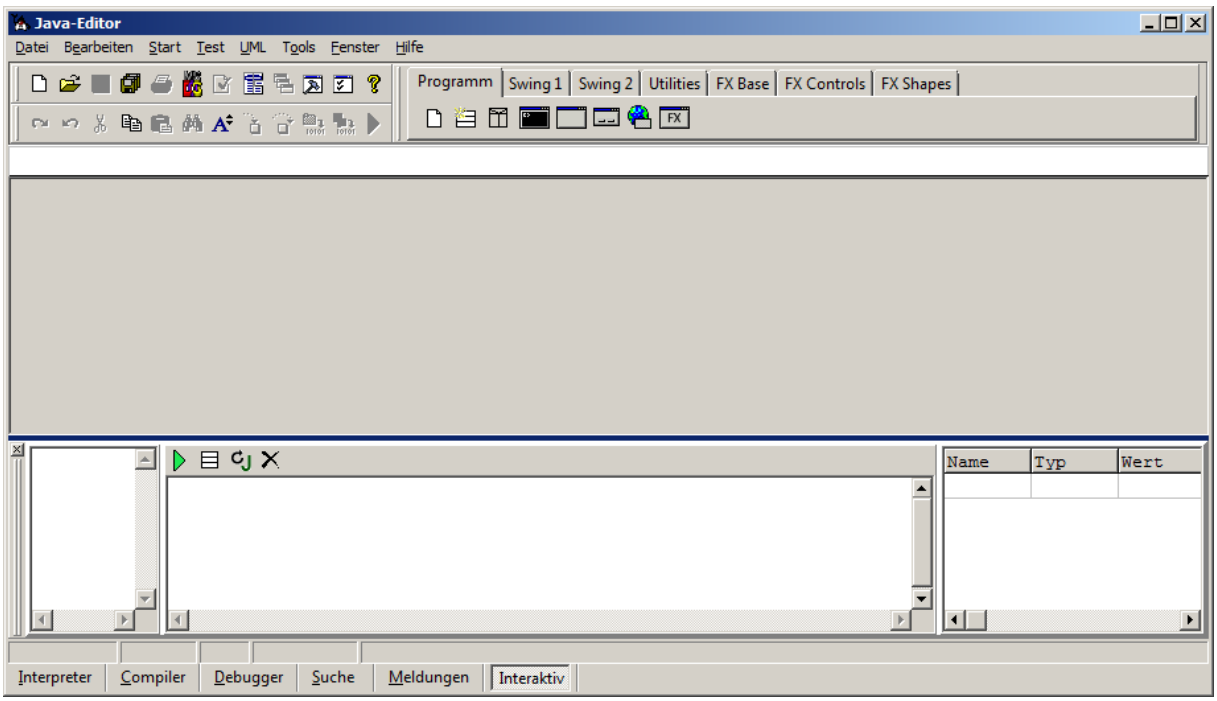
Der Kampf um das eierlegende Wollschwein

Einst fiel einem Züchter ein,
wie die Tierwelt würde sein,
wenn man durch geschicktes Paaren
Fische schüf' mit krausen Haaren.
Die könnt' man wie Pudel scheren
und die Arten sonst vermehren.
(...)
Was wir brauchen,
ist ein Schwein,
das Merinowolle trägt
und dazu noch Eier legt.
Das soll Ihre Züchtung sein!
Ludwig RENN (1959)

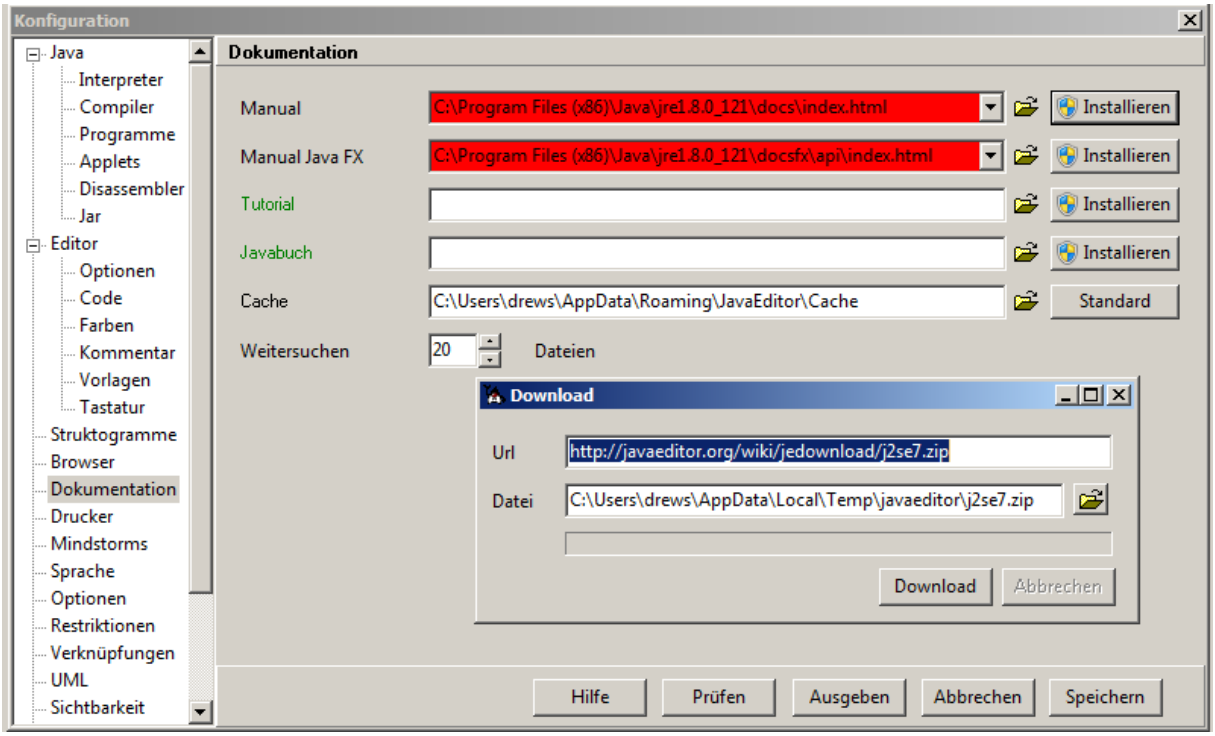
Hier wird jetzt der notierte Installations-Pfad ausgewählt (üblich ist `C:\Programme (x86)\Java\jdk` *Versionsnummer*) und nach dem Bestätigen nehmen die obersten beiden Zeilen die richtigen Informationen auf und die Farbe wird sich in die System-Farbe ändern.



Das Programm selbst ist unkompliziert aufgebaut und ersteinmal frei von Überraschungen.



In jedem Fall sollte nach der Installation die Konfiguration überprüft werden. Dazu ruft man über das Symbol mit den zwei Häkchen ("Konfiguration") den entsprechenden Dialog oder über das Programm-Menü "Fenster" "Konfiguration" auf.
Dort interessiert uns der Bereich "Dokumentation". Sollten hier Einträge rot markiert sein oder fehlen, dann können diese direkt über das rechte "Installieren" ergänzt / berichtigt werden.



Die meisten Entwicklungen werden wir mit dem Java-Editor vornehmen, da er sowohl einen UML-Editor enthält als auch die Möglichkeit bietet, mit Struktogrammen zu arbeiten. Beides sind elementare Techniken beim Erlernen einer Programmiersprache.
Weiterhin können wir auch Windows-taugliche Programme erstellen. Dadurch lassen sich die erstellten Programme auch wirklich nutzen und sind nicht nur pädagogisch wertvoll.
Ganz zu Anfang werden wir allerdings mit der sogenannten Konsole zufrieden geben, damit der erste Erklär-Aufwand im Rahmen bleibt.
Der Java-Editor ist auch als portables Programm für den USB-Stick verfügbar.
Dadurch ist das Programm auch für alle diejenigen interessant, die ihre Programmier-Umgebung immer dabei haben wollen / müssen, die nicht die nötigen Installations-Rechte (z.B. auf Schul- oder Firmen-Rechner) besitzen und diejenigen, die das "portable Apps"-Konzept einfach bestechend und praktisch finden. Zu letzteren gehöre ich auch.

Als fertiges System gibt es den Java-Editor auch auf dem "normalen" IoStick (Informatik on Stick) sowie dem Abi-IoStick von Timo HEMPEL.

1.1.2.1.1. der Java-Editor als Editor-Compiler-System

Der Java-Editor ist voorrangig ein sehr Leistungs-starker, klassischer Editor mit Syntax-Highlighting. Bei diesem werden aufegwählte Text-Teile, wie z.B. Schlüsselwörter einer Programmiersprache – farblich abgesetzt. Diese Technik beherrschen mittlerweise viele Editoren.
Sehr interessant ist das Struktorieren und Einfärben von Code-Blöcken. Dadurch lassen sich fehlende Block-Klammern schneller "erkennen".

1.1.2.1.2. der Java-Editor als UML-Editor

Wenden wir uns zuerst dem Einfachen zu. Es soll aus einem JAVA-Klassen-Konstrukt ein UML-Diagramm erstellt werden. Da reicht eine kurze Menü-Befehls-Sequenz und man ist fertig.

Aber auch des Erstellen von UML-Diagrammen als Ausgangspunkt für eine Programm-Entwicklung ist keine Zauberei.
Man beginnt mit einer Dialog-gestützten Definition der Klasse und ihrer Methoden.
Es lassen sich bestimmte Grundstrukturen gleich automatisch erzeugen. Dazu gehören Konstruktoren und die Geter und Seter.
Werden Klassen bei der Definition in den Dialogen gleich anderen Klassen zugeordnet, dann entsteht später auch ein hierrarchisches UML-Diagramm mit den gewünschten Details (aus schulischer Sicht).

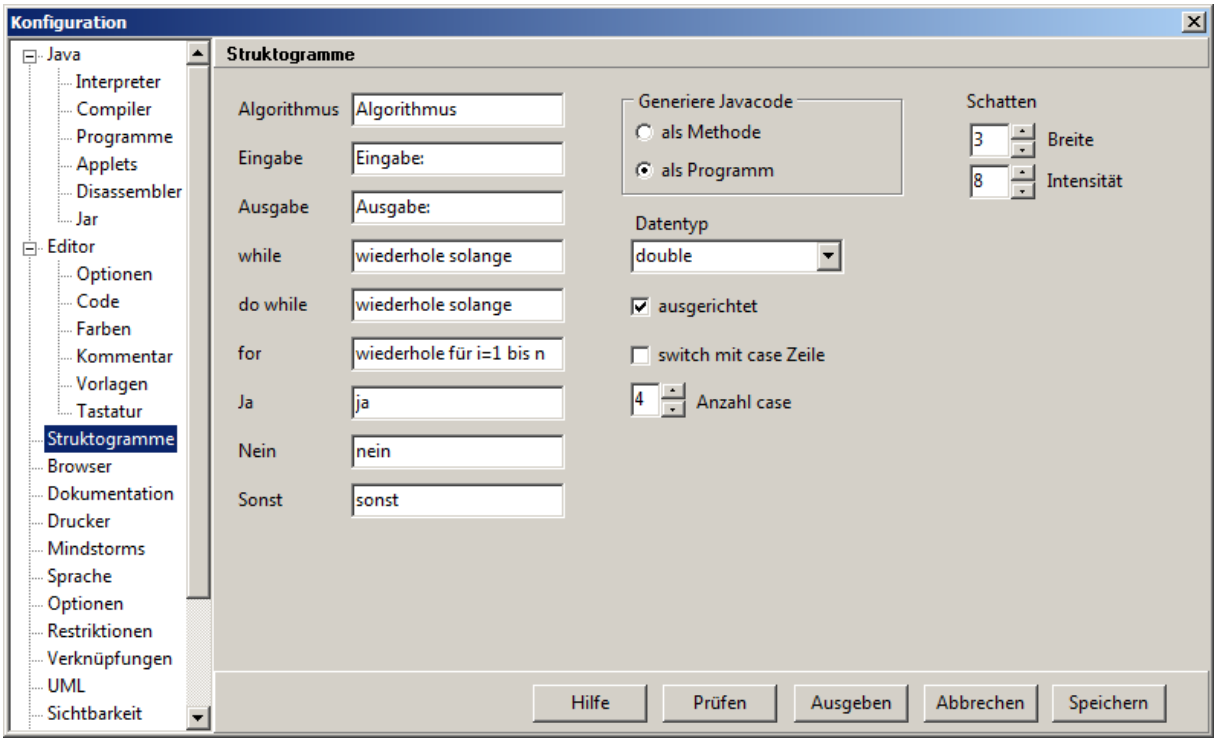
1.1.2.1.3. der Java-Editor als Struktogramm-Editor

Per Drag and Drop lassen sich die Struktogramm-Elemente aus der linken Werkzeug-Leiste zu einem vollständigen Struktogramm zusammenstellen.

Fertige Struktogramme lassen sich in Java-Code umsetzen. Das "Ergebnis" muss aber natürlich geneuestens geprüft werden, da die Idee hinter den Struktogrammen ja gerade eine Sprache-unabhängige Darstellung der Algorithmen ist. Da bleiben logischerweise viele JAVA-Elemente außen vor.

Beim Erstellen eines neuen Struktogramm's über das charakteristische Symbol werden wir gleich zum Speichern der Datei aufgefordert.
Als praktisch hat sich herausgestellt hier kurze prägnante Namen a'la Klassen-Bezeichner zu vergeben und dies auch bei der Benennung des Algorithmus so zu machen. In dem Fall sind dann auch die Klassen im Java-Code gleich richtig benannt.

Die Schlüsselbegriffe für die Umsetzung der einzelnen Blöcke in passenden JAVA-Quellcode lassen sich unter "Fenster" in der "Konfiguration" einstellen. Dort gibt es in der linken Baum-Struktur den Punkt "Struktogramme". Als Voreinstellung sind definiert:



Die Umwandlung solcher Struktur-Elemente, wie Verzweigungen und Schleifen funktioniert sehr gut.

Viele andere Elemente müssen händisch umgeschrieben werden, da sie nur faktisch eingesetzt werden.

Löschen eines Elementes erledigt man über das Auswählen und das Anklicken des Mülleimers ganz unten in der Werkzeug-Leiste links neben dem Struktogramm-Editor-Fenster.
Nicht verwechseln mit dem X ganz oben in selbiger Leiste – dieses Symbol steht für das Schließen des Struktogramm-Editor's.

1.1.2.1.4. der Java-Editor als Interpreter

Das JAVA-System als reinen Interpreter zu verwenden ist eher eine Spezial-Anwendung in der Programmier-Ausbildung. Hier soll nur kurz die prinzipielle Möglichkeit aufgezeigt werden.

Links:
http://javaeditor.org/ (Projekt-Seite)
https://tinohempel.de/info/info/IoStick/index.html (Download der IoStick's)

1.1.2.2. jEdit

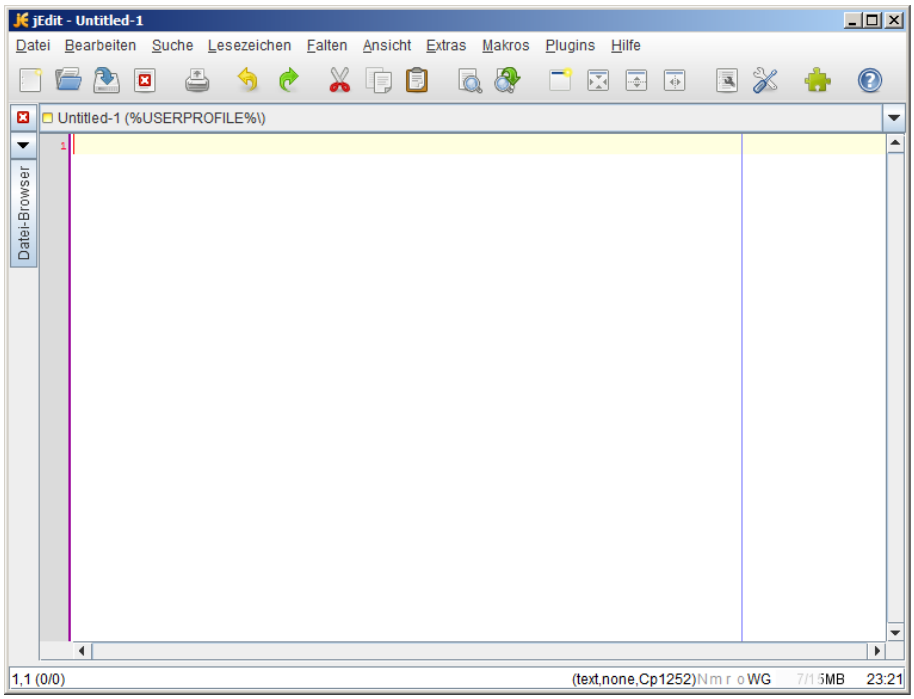
kurz jE

aufgeräumt, übersichtlich

Konzentration auf das Wesentliche

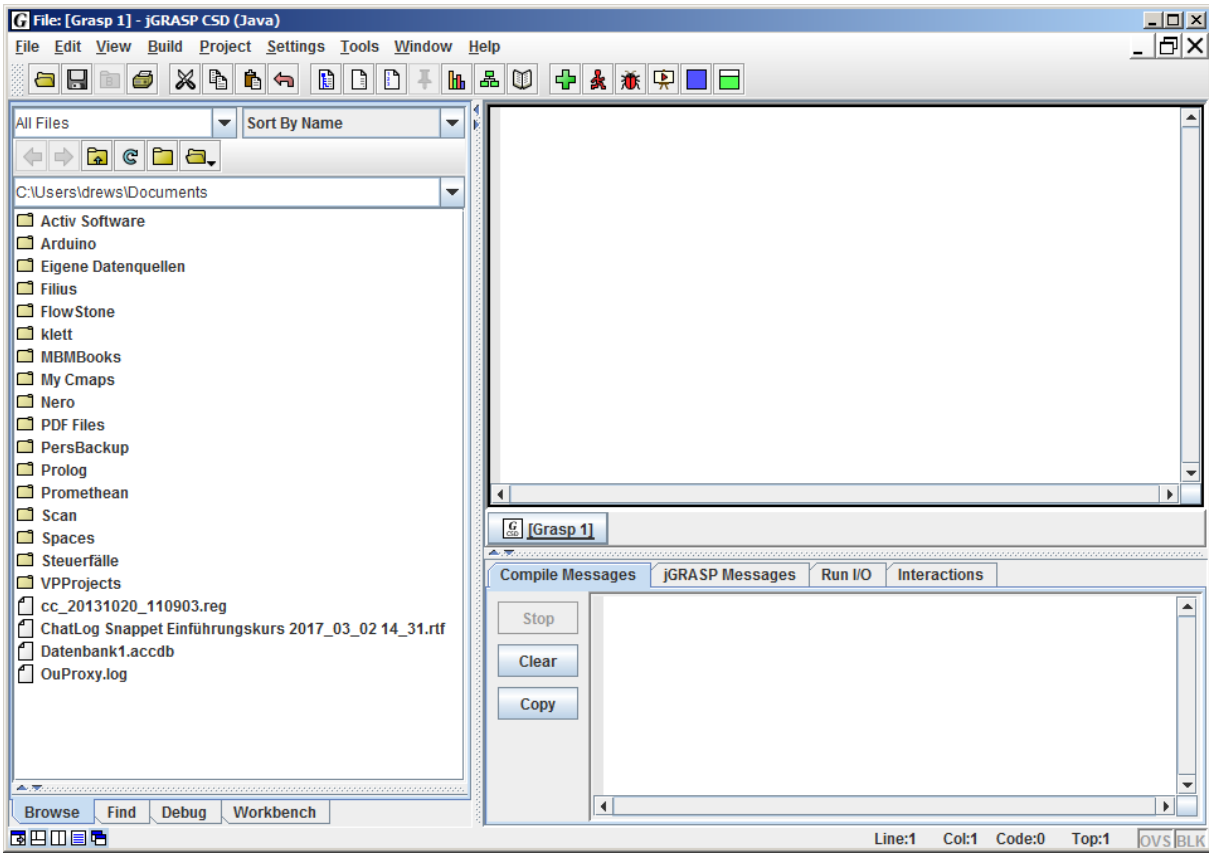
beim Durchforsten der Menü's wird aber die große Leistungsfähigkeit offensichtlich

auch in Deutsch verfügbar und somit auch für Fremdsprachen-Asketen geeignet.



Vorlagen wären schön!
man kann sich mit einer oder mehrerer schreib-geschützten Datei(en) behelfen

1.1.2.3. jGRASP



nur englisch

auch als portables Programm verfügbar

1.1.2.4. Eclipse – die Universal-IDE

typische IDE (Integrated Development Environment → dt.: integrierte Entwicklungs-Umgebnung)
andere IDE sind prinzipiell ähnlich; unterscheiden sich im Handling
z.B. NetBeans, IntelliJ, BlueJ (mit Einschränkungen)

open source
als universelle IDE für sehr viele Programmiersprachen angelegt
deshalb ist die Kenntnis und die Benutzer-Befähigung für Eclipse für Programmierer in / mit verschiedenen Programmiersprachen geeignet (unterschiedliche Kurse / Arbeitsbereiche / …)

erweiterbar mit plug in's

relativ Einsteiger-freundlich, trotzdem mehr an den fortgeschrittenen oder proffessionellen Programmierer gerichtet; der Anfänger wird von der Funktions-Vielfalt meist überfordert
dazu kommt, dass diese IDE nur in der Programmierer-Sprache englisch verfügbar ist

Bietet alles zum Erstellen, Editieren (Ändern), Testen, Berichtigen (Debuggen), Optimieren, Verwalten und Ausliefern (Ausrollen) von eigener Software.

sehr umfangreiches Programm für viele verschiedene Programmiersprachen
deshalb manchmal nicht 100%ig begrifflich konform mit JAVA

Aufbau des Eclipse-Fensters
zentral der Editor (Haupt-Code-Ansicht)
links der Package-Explorer
rechts outline (zeigt verfügbare Methoden und Attribute an)
unten Konsole

weiterhin z.B. vorhanden:
Bug-Explorer
Deklarations-Anzeige () (wo ist eine aufzurufende Methode wie deklariert)
Javadoc (anzeigen)
Navigator (Datei-Explorer)
Problems (Hinweise, Fehler in / zum Quellcode)
Template's
Type-Hierrarchie (zeigt die Vererbungs-Hierrarchie)

Perspective's

Preference's
"general" "workspace" → UTF-8
möglichst vor dem Start eines Projektes einstellen, damit man nicht Probleme mit deutschen Umlauten bekommt

Download der aktuellen Version

Entpacken in gewünschten Ordner (Standard: C:\Programme\eclipse\)

Starten von eclipse

Festlegen des Workspace (Ort, wo unsere Dateien liegen (sollen))

Anpassen der Benutzer-Oberfläche

File New JAVA-Projekt

File New Class
Paketnamen z.B. umgedrehte Domain-Namen de.firma

Verwendung von vordefinierten Kürzeln (z.B. syso → System.out. …)

Ausführen über "" (Play-Symbol)

Java Build Chain

ein Projekt in Eclipse importieren:
Rechtsklick in Package Explorer
=> Import aus dem Kontextmenü auswählen
=> General => Existing Projects into Workspace => Next
=> Select Archive File (Die heruntergeladene .zip Datei auswählen)
=> Finish

Debugging in Eclipse

den Begriff "bug" wurde von Grace HOPPER für Fehlerstellen / Probleme als lästiges Ungeziefer (Käfer) eingeführt

HOPPER fand nach einer Fehlfunktion eines Computers einen Käfer auf einer Platine, der dort einen Kurzschluss ausgelöst hat

es geht allgemein um das Finden, Isolieren, Beseitigen von Fehlern
Überwachen von Variablen und Methoden-Aufrufen

erster Test: ist Fehler reproduzierbar?

Zwischen-Anzeigen von Werten / Variablen

setzen von Breakpoints (Haltepunkte)
ev. auch bedingte Haltepunkte

Just-in-Time Debugging

Links:
http://www.eclipse.org/downloads/ (Download Eclipse)

1.1.2.5. NetBeans – die Haus-eigene IDE

ehemals Sun und Oracle
jetzt vom "Apache Software Foundation"-Team betreut
für Win, Linux und Mac verfügbar (allg. nur noch 64-bit unterstützt)

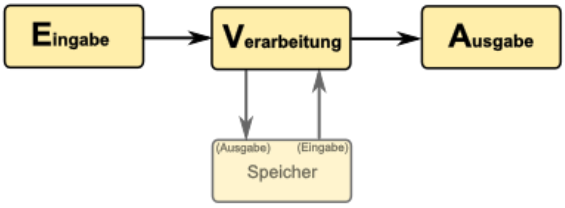
Download → https://netbeans.apache.org/download/index.html
konservative Nutzer sollten zur LTS-Version (Long Time Service-Version) greifen
längerfristiger Standard, i.A. sehr stabil
gute Referenz auch für die Nutzung von Foren etc.

man arbeitet mit einem "Projekt", das aus vielen speziellen Dateien (vorrangig Quelltext) besteht
es sollte immer ein extra Ordner für ein Projekt benutzt / erstellt werden

Eingabe-Hilfen oft auch Makro's genannt
zum schnellen Eingeben von häufig gebrauchtem Quelltext

psvm [ ]	public static void main(String[ ] args) { …. }
sout [ ]	System.out.print

1.2. Erst-Kontakt



Normalerweise müsste man sich entsprechend dem EVA-Prinzip bei der Eingabe anfangend bis zur Ausgabe durchhangeln.
Wir wollen aber schon in einem frühen Programmier-Stadium etwas auf dem Bildschirm sehen können. Die Ergebnise sollen ja schließlich sichtbar werden. Aus diesem Grund beginnen wir mit der Ausgabe.
Im nächsten Schritt ergänzen wir die Verarbeitung. Es folgt die Eingabe. Hier werden wir sehen, dass die Leichtigkeit von Ausgabe und Verarbeitung verfliegt und die ersten komplizierten Strukturen auftreten.

1.2.1. Das Grund-Konstrukt eines JAVA-Programms.

Leider ist der Erst-Aufwand für eine JAVA-Programm immer recht groß. Zuerst wollen wir die ganzen Zeilen und Befehls-Wörter auch einfach so hinnehmen. Später erklären wir diese dann genauer (→ 1.4.4. Warum also?: public static void main(String[ ] args) { … }).
Fangen wir mit dem notwendigen Grundgerüst eines JAVA-Programm an:

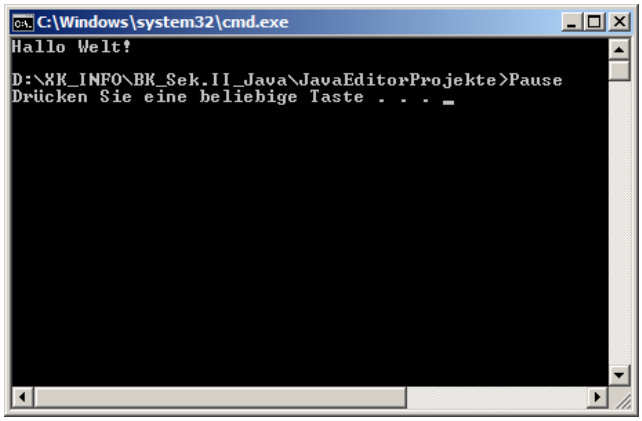
```
class MachNichts{
    public static void main(String[] args){
    }
}
```

Jedes JAVA-Programm ist eine Klasse, warum das so ist, wird später thematisiert. Die Kennzeichnung erfolgt über das Schlüsselwörtchen **class**. Dahinter folgt der Klassenname. Dieser sollte Zweck-entsprechend gewählt werden, damit man nachher (nach vielen geschriebenen Programmen und Klassen) noch den Überblick behält. Die geschweifte Klammer ist der Beginn unseres Klassen-Codes. Dieser endet bei der schließenden geschweiften Klammer in der letzten Zeile. Das diese geschweifte Klammer zur ersten dazugehört, wird durch die gleiche Spalten-Position (hier ohne Einrückung) gekennzeichnet. Das gleiche Prinzip wird auch bei den inneren beiden Zeilen deutlich. Die schließende Klammer gehört hier zu der in Zeile 2.
Der kryptische Aufruf "**public static void main(String[] args)**" ist der Start des eigentlichen Programm-Ablaufs. Alles was unser Programm leisten soll, muss also zwischen den geschweiften Klammern in Zeile 2 und 3 eingefügt werden.
Jedes ausführbare JAVA-Programm / jede ausführbare JAVA-Klasse muss eine solche Zeile enthalten und diese darf auch nur einmal in einem JAVA-Programm vorkommen!
Man nennt diesen Programm-Teil auch die **Main-Methode**.
Da in unserem obigen Programm nichts zwischen den Klammern steht, leistet unser Programm tatsächlich auch nichts.
Wir können es aber übersetzen und ausführen lassen. Das passiert in der benutzten IDE üblicherweise über grüne Play-Dreiecke. Sie stehen für die Kompilierung und Ausführung des Programms.
Eine erste anzustrebende Leistungs könnte eine Ausgabe auf dem Bildschirm sein. Üblich ist eine Meldung, dass man es als Programmierer geschaft hat, Kontakt zur "Welt" aufzunehmen und sich meldet. Das Programm heisst "Hello Welt!".
Wir ändern dazu die erste Zeile und die dort angegebene Klassen-Bezeichnung. Als neue 3. Zeile fügen wir eine Funktion – quasi einen JAVA-Befehl ein, der eine Ausgabe erstellt. Ausgegeben soll der Inhalt zwischen den beiden Anführungs-Zeichen – also: Hallo Welt! werden.

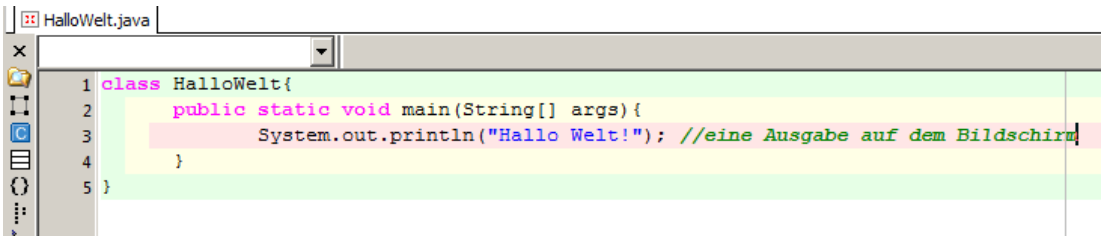
Jede Befehlszeile muss mit einem Semikolon (;) abgeschlossen werden.

```
1  class HelloWelt{
2      public static void main(String[] args){
3      System.out.println("Hello Welt!");
4      }
5  }
```

Die Übersetzung und Ausführung bringt tatsächlich eine Hallo-Welt-Meldung auf den Bildschirm.



Der schwarze Text-Bildschirm wird Konsole genannt. Einige Nutzer kennen Konsolen von LINUX-Systemen oder als sogenannte "MS-DOS Eingabeaufforderung".
In den neureren WINDOWS®-Versionen kann man sich die Konsole auch über das Ausführen vom **cmd** aufmachen.
Die letzte Ausgabe-Zeile erzeugt das JAVA-System automatisch, damit man überhaupt was sieht. Normalerweise würde nur der Text ausgegeben werden und dann sofort wieder zum Editor zurückgekehrt werden.
Das geht so schnell, dass man nicht einmal das Aufbloppen des Fenster erkennt.
Wer auch den Java-Editor verwendet, bei dem sollte der Quell-Code etwa so aussehen:



Der Java-Editor verwendet automatisch verschiedene Hintergrund-Farbein für syntaktische zusammengehörende Bereiche / Zeilen. Man nennt diese Bereiche auch Blöcke. Ein Block ist eine Gruppe von JAVA-Anweisungen.
Rötlich ist hier der abzuarbeitende Block in der main-Methode unterlegt. Darin enthalten ist derzeit eine Anweisung, die eine Ausgabe auf dem Bildschirm erzeugt.
Der auszugebende Text wird blau geschrieben. Die reservierten Wörter werden vom Editor nur fett dargestellt, was mir nicht ausreicht. Deshalb habe ich mir für die "reservierten Wörter" eine andere - auffälligere Farbe (hier: Fuchsia) eingestellt.
In diesem Script werden wir die Programmtexte aber zumeist neutral – wie oben – darstellen. Damit bleiben wir ziemlich unabhängig vom konkreten Editor, den Sie sich ausgewählt haben oder den Sie aus irgendwelchen Gründen verwenden müssen. Bestimmte Schlüsselstellen oder die gerade besprochenen Sachverhalte werden ev. farblich hervorgehoben.
Weiterhin fällt hinter unserer Befehlszeile noch ein Text auf, der mit doppelten Schrägstrichen (Slash's) beginnt. Hierbei handelt es sich um einen Kommentar. Kommentare werden vom JAVA-System nicht beachtet. Sie dienen dem Programmierer als Hilfsmittel und Hinweis.

Aufgabe:
Erstellen Sie ein "Hello Welt!"-Programm mit der unten angegebenen Ausgabe! Für das XXX setzen Sie Ihren eigenen Namen ein!

Hier meldet sich XXX: Hallo Welt!

Wichtig ist es immer die Aufgaben exakt zu erfüllen! So mancher Programmier-Anfänger meint, wenn irgendetwas irgendwie auf dem Bildschirm erscheint, dann reicht das. Als befriedigende Leistung vielleicht, aber nicht für besser. Der Kunde / Auftraggeber bestimmt, wie etwas aussehen soll. Wenn das nicht erfüllt wird, dann hat man den Auftrag nicht erfüllt. Der Kunde / Auftraggeber hat sich irgendetwas dabei gedacht, oder es ist elementar für seine Betriebsabläufe etc. Wenn Sie sich als Programmierer sich aber darüber hinweg setzen, wird es hinterher Probleme geben.
Nehmen wir beispielhaft die Addition von 2 und 3. Der Kunde möchte die Ausgabe als Zahl – also 5. Wenn Sie sich als Programmierer denken, fünf Striche sind ja auch fünf, dann haben Sie sachlich recht, aber es entspricht nicht der Anforderung. Das Programm muss also umgeschrieben werden, was wieder Zeit kostet. Und wer soll's bezahlen? Der Kunde / Auftraggeber wird sich berechtigt weigern. Und Ihr Arbeitgeber – die Programmier-Firma – hat auch kein Geld zu verschenken. Bleibt also nur, dass Sie Ihren "Fehler" in Ihrer Freizeit berichtigen.
Die Einhaltung des sogenannten Pflichten-Heftes (einer Aufgaben-Stellung) ist also keine Kür, sondern eine Pflicht. Standards für Texte oder Zahlen usw. sind einzuhalten und müssen nicht jedes Mal explizit definiert werden. Denken Sie immer daran, Sie müssen das Programm den 100 Mitarbeitern des Auftraggebers / Kunden einzeln erklären. Da sind doch selbsterklärende, exakte und übliche Ausgaben viel besser und zweckmäßiger.

1.2.2. Gestaltung von Text-Ausgaben

Zuerst werden uns zwei Ausgabe-Methoden von JAVA völlig ausreichen. Die erste ist:

System.out.println();
Mit **println** erzeugt man eine (Text-)Ausgabe in der Konsole mit abschließendem Zeilen-Umbruch. Dafür steht das **ln** (line). Eine Linie ist eine Ausgabe-Zeile in der Konsole mit einem Zeilen-Sprung ("Enter", Zeilen-Wechsel) und einem Sprung zur 1. Ausgabeposition (früher Drucker-Wagen-Rücklauf) in der Zeile.
Diese speziellen Steuer-Zeichen stammen noch aus der Zeit, als Ausgaben grundsätzlich auf einem Drucker erfolgten. Erst nach dem "Enter wurde der Druck der Zeile gestartet und dann eben die nächste Zeile angesteuert.
Würde man auf den Zeilen-Sprung verzichten, würde bei einem Wagenrücklauf der neue Druckbeginn in der aktuellen Zeile sein. Man würde also mit neuen Ausgaben die alten überschreiben.
Die zweite wichtige Ausgabe-Methode für die Konsole ist:

System.out.print();
Diese Methode dient nur der Erzeugung einer Teil-(Text-)Ausgabe in der Konsole (ohne abschließendem Zeilen-Umbruch). Es fehlen also die beiden Steuerzeichen "Zeilenvorschub" (LF; Line feed) und Wagen-Rücklauf (CR, Cartrige return).
Weitere Ausgaben beginnen direkt hinter dem gerade ausgegebenen Text. Der eigentliche Zeilen-Abschluss muss dann mit einer **println()**-Methode erfolgen.

Um also eine komplexe Ausgabe hinzubekommen, kann man mehrere print()-Methoden benutzen. Als letztes benutzt man dann eine println()-Methode. Das ist deutlich übersichtlicher als eine alleinige println()-Methode.

1.3. Immer Schritt für Schritt

1.3.1. Texte verknüpfen / konkatenieren

Hier geht es jetzt um die praktische Nutzung der Methoden print() und println(). Vornehmlich werden die beiden Funktionen zur Kombination von Ausgaben benutzt, die in verschiedenen Teilen des Programms erzeugt werden oder von recht komplexen Ausgaben, die man nicht in einen einzelnen Methoden-Aufruf stecken möchte.
Praktisch immer geht es um die Kombination von mehreren Texten, die wir in der Informatik auch Strings nennen. Das Aneinanderhängen von Text(-Teil)en oder Strings wird Konkatenation genannt. Dieses Verfahren kommt in der Verarbeitung von Texten sehr häufig vor und ist nicht auf die reine Ausgabe beschränkt.
Typische Konkatenationen sind z.B. das Zusammenstellen von vollständigen Namen aus ***Vorname*** und ***Nachname*** oder eine Namens-Konstruktion in der Form: ***Nachname, Vorname***.
Die erste Namens-Zusammenstellung scheint für uns Menschen ganz problemlos möglich zu sein. Reicht da nicht das einfache Hintereinanderschreiben von ***Vorname Nachname***? Schon hier würde unser Compiler beim Übersetzen streiken. Er braucht eine Verbindung zwischen den Teil-Strings.
Als Beispiel nehmen wir mal ganz kreativ ***Klaus Mustermann***. Eine Möglichkeit der Kombination der Texte wäre ja die getrennte Ausgabe mit den erwähnten print()- und println()-Methoden. Der exakte Aufruf benötigt noch die Angabe der Herkunft der Methode aus der JAVA-Bibliothek ***System.out***.

```
System.out.print("Klaus");
System.out.println("Mustermann");
```

Die Ausgabe (Abb. rechts) überrascht die meisten Programmier-Einsteiger und macht deutlich wie engstirnig Computer doch sind.

```
KlausMustermann
```

Sie machen genau, was der Programmierer ihnen sagt. Computern müssen wir alles peinlich genau sagen.

```
System.out.print("Klaus");
System.out.print(" ");
System.out.println("Mustermann");
```

Nun erscheint auch der Nachname getrennt vom Vornamen. Das Leerzeichen muss also von selbst eingegeben werden.

```
Klaus Mustermann
```

Bei der Ausgabe werden die einzelnen Strings dann für den Bildschirm zusammengesetzt.
Die gleiche Ausgabe kann auch mit nur einer println()-Methode erreicht werden. Dazu nutzen wir die in JAVA (und den meisten anderen Programmiersprachen) übliche Konkatenation von Strings mit dem Plus-Zeichen (**+**) aus.

```
System.out.println("Klaus"+" "+"Mustermann");
```

Die Konkatenation kann im Quell-Text manchmal sehr lang werden. Dann bietet sich auf alle Fälle die oben besprochene Version mit mehreren print()- und println()-Methoden an.

```
Klaus Mustermann
```

Aufgabe:

Wie müssten die beiden Programmtext-Versionen lauten, wenn nun ein Nachnamen-orientierte Ausdruck erfolgen soll?

(also: ***Nachname, Vorname***)

Mit den oben gewonnenen Kenntnissen können wir nun unser erstes Programm erstellen. Vielleich überlegen Sie sich vorher, welche Ausgabe die Zeilen 3 und 4 auf dem Bildschirm erzeugen.

```
1 class HelloWelt{
2     public static void main(String[] args){
3     System.out.print("Hello Welt! ");
4     System.out.println("Hier ist ein JAVA-Programmierer!");
5     }
6 }
```

Etwas übersichtlicher – und vor allem mit weniger Schreib-Aufwand behaftet – ist die folgende Version:

```
1 class HelloWelt{
2     public static void main(String[] args){
3     System.out.println("Hello Welt! "
4                        +"Hier ist ein JAVA-Programmierer!");
5     }
6 }
```

Aufgabe:

Es soll mit dem unten angegebenen Programm die folgende Ausgabe erzeugt werden. Finden Sie die Fehler im Programm (nur in den Ausgabe-Methoden)!

```
Hallo Welt,
ich wollte nur mitteilen, dass ich mein erstes JAVA-Programm
geschrieben haben. Nun kann ich auch schon mehrere Texte auf
dem Bildschirm anzeigen lassen.
```

```
1  class HelloWelt2{
2      public static void main(String[] args){
3      System.out.println("Hello Welt! ");
4      System.out.println("ich wollte nur mitteilen,");
5      System.out.print("dass ich mein erstes");
6      System.out.println(" ");
7      System.out.print("Java-Programm");
8      System.out.print("geschrieben haben."+"Nun kann ich");
9      System.out.println("mehrere"+" "+ Texte"+""+" auf dem");
10     System.out.print("Bildschirm"+anzeigen+lassen);
11     }
12 }
```

1.3.2. Zahlen und einfache Berechnungen ausgeben

Neben Texten lassen sich mit der print()- bzw. println()-Methode auch Zahlen ausgeben.

```
1  class Ausgabe{
2      public static void main(String[] args){
3      System.out.println(3);
4      System.out.println(4.75);
5      }
6  }
```

Die Ausgabe ist natürlich nicht spektakulär, führt uns aber zum nächsten Schritt – der Ausgabe von Berechnungs-Ergebnissen.

```
3
4.75
```

Auffällig ist lediglich, dass hier Zahlen mit einem Punkt akzeptiert werden. Probiert man ein Komma, dann gibt es eine Fehler-Meldung.
Wir müssen uns dazu merken, dass Komma-Zahlen in der englisch-amerikanischen Notierung – also mit einem Punkt als Dezimal-Trenner notiert werden.
Aber kommen wir zu Berechnungen. Zuerst die einfache Additionen:

```
1  class Ausgabe{
2      public static void main(String[] args){
3      System.out.println(3+4);
4      Systen.out.println(4.75+9);
5      }
6  }
```

Die Anzeigen liegen völlig im Bereich der Erwartungen. JAVA berechnet zuerst den Term – z.B. 3 + 4 – und gibt dann das Ergbnis aus.

```
7
13.75
```

Genauso funktioniert es mit der Substraktion und der Multiplikation.
Das Mal-Zeichen ist – wie überall in der Datenverarbeitung – ein Sternchen (*****).
Die ersten Überraschungen erwarten uns bei der Division. Das Operations-Zeichen ist der Schrägstrich (Slash).

```
3      System.out.println(3/4);
4      Systen.out.println(3.0/4);
```

Obwohl beide Terme für uns gleichartig sind, liefert JAVA unterschiedliche Ergebnisse. Dazu muss man wissen, dass wenn ganze Zahlen verrechnet werden, auch nur ganzzahlig dividiert wird.

```
0
0.75
```

Es bleibt ein Rest, den man mit der Modulo-Operation (%) abfragen / berechnen kann.
Ist ein Operand eine Komma-Zahl, dann wird mit dem Schrägstrich eine Komma-Zahl berechnet, auch wenn hinter dem Komma (Punkt) eine Null stehen würde. Die Modulo-Operation wird auch bei Komma-Zahlen als Reste-Berechnung ausgeführt.

```
3      System.out.println(3%4);
4      Systen.out.println(3.4%2.8);
```

Das gilt aber nur für den ersten Operanden. Der zweite – also der Divisor – wird immer als Ganzzahl betrachtet.
Da die Zahl 3 durch 4 nicht ganzzahlig geteilt werden kann, bleibt ein Rest, der eben genau 3 ist.

```
3
0.6000000000000001
```

Das zweite Ergebnis verwundert die mathematisch gebildeten Leser sicherlich. Hätte da nicht glatt 0,6 herauskommen müssen? Warum es zu bei Gleitkommazahlen immer zu leicht abweichenden Ergebnisse kommen kann, erklären wir später (→ 1.3.6.0.2. Gleit- oder Fließkommazahlen). Das liegt an der internen Zahlendarstellung.

1.3.3. Texte und Zahlen / Terme gemeinsam ausgeben

Die einfache Ausgabe von Zahlen ist nicht wirklich informativ. Vielmehr braucht man ordentliche Ausgaben, in denen die Berechnungen in Begleit-Texte eingebunden sind. Der Nutzer soll schließlich wissen, was da gerade – in ev. irgendwelcher Einheit – berechnet wurde.
Kombinieren wir Texte und Zahlen, dann verhält sich die Ausgabe-Methode völlig normal.

```
1  class Ausgabe{
2      public static void main(String[] args){
3      System.out.println("Das Ergebnis lautet: "+3);
4      }
5  }
```

Hinter der Zahlen-Ausgabe kann auch wieder ein Text folgen, das ändert nichts an der Ausgabe.

```
Das Ergebnis lautet: 3
```

Zu beachten sind aber Leerzeichen in Ausgabetexten, die man für eine ordentlich lesbare Ausgabe mit einbauen muss. Die gehören zu den immerwährenden Aufgaben des Programmierers, da muss niemand etwas ins Pflichtenheft schreiben.
Der nächste logische Schritt ist nun die Berechnung eines Terms begleitet von Texten. Auch hier erwartet uns wieder eine Überraschung.

```
3      System.out.println("Das Ergebnis lautet: "+3+4);
```

Es kommt bei der Addition statt zur Berechnung "nur" zur Konkatenation. Andere Berechnungen erfolgen aber wieder ordentlich.

```
Das Ergebnis lautet: 34
```

Hier ist also gründliches Testen notwendig, bevor man ganze Kundenstämme mit falschen Daten überschüttet. Im Zweifel kann man mit runden Klammern – und nur solche sind in Berechnungen / Termen erlaubt – eine Berechnung erzwingen.

```
3      System.out.println("Das Ergebnis lautet: "+(3+4));
```

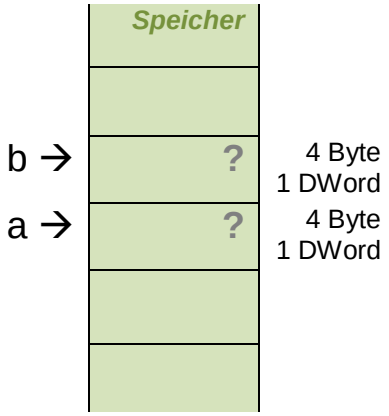
```
Das Ergebnis lautet: 7
```

1.3.4. Variablen

Jeder kennt Variablen aus der Mathematik. Sie stehen als Platzhalter für beliebige Zahlen. Man kann sie unterschiedlich belegen und sie können ihren Wert während der folgenden Berechnungen auch ändern.
Genauso werden sie auch in JAVA genutzt. Bevor wir aber eine Variable benutzen können, müssen wir dem JAVA-System eine Mitteilung machen, das etwas eine Variable sein soll. Das JAVA-System muss zur internen Verarbeitung und Abspeicherung wissen, um was für eine Variable es sich handelt. Was soll mit ihr abgespeichert werden? Wir haben bisher drei verschiedene Daten-Typen kennen gelernt. Da sind zum Einen die Texte. Ihre Kennzeichnung (Typ-Bezeichnung) heißt String. Ganze Zahlen haben den Typ **int** und Komma-Zahlen den Typ **double**. Später werden wir weitere kennen lernen. Bei der grundsätzlichen Verarbeitung ändert sich aber nichts.
In den Zeilen 3 und 4 des nachfolgenden Programms werden die Variablen a und b vom Typ int – also als Ganzzahl – angelegt.

```
1 class Multi{
2     public static void main(String[] args){
3     int a;
4     int b;
5     a = 3;
6     b = 4;
7     System.out.println("Das Ergebnis lautet: "+ a*b);
8     }
9 }
```

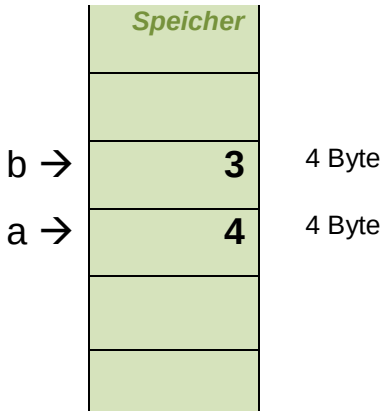
Innerhalb von JAVA passiert dabei folgendes. Das JAVA-System reserviert sich einen Teil des Speichers – im Falle von **int** sind das 4 Byte. Hätten wir z.B. **double** gewählt, dann wären es schon 8 Byte pro Variable.
Bei vielen Zahlen kann die falsche Typ-Wahl also schnell in den Speicher gehen.
Die Stelle im Speicher – also die Adresse – wird mit dem Namen der Variable – also z.B. **a** assoziiert.
Betrachtet man das Definieren einer Variablen ganz genau, dann liegt auch der Variablen-Name im Speicher und wird von einem Adress-Verweis auf die Speicherstelle des Wertes gefolgt. Die Adresse zeigt quasi auf die Speicherstelle, wo sich der Variablen-Wert befindet. Man spricht auch von einem Zeiger.
Hier reicht uns aber die vereinfachte Betrachtung.



Obwohl wir die Variablen ordnungsgemäß deklariert ("dem System angezeigt") haben, bekommen wir bei einer direkten Ausgabe oder einer einfachen Verwendung eine Fehlermeldung. Die Variable muss auch noch initialisiert ("vorbelegt") werden.

```
1 class Multi{
2     public static void main(String[] args){
3     int a;
4     int b;
5     a = 3;
6     b = 4;
7     System.out.println("Das Ergebnis lautet: "+ a*b);
8     }
9 }
```

Dies kann, wie in den Zeilen 5 und 6 zu sehen ist, durch eine Zuweisung (mittels Gleichheits-Zeichens) erfolgen oder verkürzt gleich mit in der Deklaration. Dann könnte man sich die derzeitigen Zeilen 5 und 6 sparen.
Beim Initialisieren oder auch später bei solchen Zuweisungen, wird der Inhalt der Speicherzellen festgelegt. Das ganze ist natürlich eine Dualzahl – aber dazu später mehr.
Ein besonders guter und wenig Fehler-anfälliger Programmier-Stil ist es, die Variable bei der Deklaration mit einem unschädlichen Wert zu belegen, wie z.B. 0 oder 1.
Bevor man mit den Variablen arbeitet, muss diesen ein regulärer Wert zugewiesen werden.




```
1  class Multi{
2      public static void main(String[] args){
3      int a = 0; //unschädliche Vorbelegung
4      int b = 0;
5      a = 3; //Wertzuweisung
6      b = 4;
7      System.out.println("Das Ergebnis lautet: "+ a*b);
8      }
9  }
```

Könnte man den Compiler austricksen, dann würde man in den deklarierten Variablen völlig zufällige Werte finden. Der Compiler prüft aber die vollständige definition der Variable mit eingeschlossener erster Zuweisung, bevor die Variable genutzt werden kann.
Natürlich dürfen auch gleich die richtigen Werte in der Deklaration zugewiesen werden. Das spart Tipp-Arbeit und Quelltext-Zeilen. Für eine andere Person, die unseren Quelltext lesen soll, ist die direkte Zuweisung auch besser verständlich.
Die deklarierten Variablen werden in unserem Beispiel-Programm in der Zeile 7 benutzt. Dazu wird zuerst der Term **a*b** ausgewertet / berechnet und dann zur Ausgabe gebracht.
Die Term-Berechnung erfolgt ganz einfach. JAVA findet zuerst das **a**. Das **a** war als Variable deklariert, also wird aus der zugehörenden Speicher-Stelle der Wert ausgelesen. Dann wird alles für die Multiplikation vorbereitet und der zweite Operand erfragt. Dieses soll **b** sein, welches ebenfalls als Variable bekannt ist. Der Wert wird zur Multiplikations-Einheit gebracht und das Ergebnis berechnet.
Alternativ kann man auch eine dritte Variable für das Ergebnis benutzen, hier jetzt **c**:

```
1  class Multi{
2      public static void main(String[] args){
3      int a = 3; //Vorbelegung
4      int b = 4; //Vorbelegung
5      int c;
6      c = a * b; //Berechnung
7      System.out.println("Das Ergebnis lautet: "+ c);
8      }
9  }
```

Variablen-Bezeichner müssen immer mit einem Buchstaben beginnen und dürfen nur Buchstaben und Ziffern enthalten. Weiterhin ist auch der Unterstich zugelassen. Zwischen Groß- und Klein-Buchstaben wird unterschieden! Variablen für einfache, kleine, lokale Berechnungen dürfen auch nur einzelne Buchstaben sein. Für globale Variablen, die vielleicht auch andere Programmierer weiter benutzen sollen, sind sprechende Namen eine Programmierer-Pflicht.
I.A. sind Programm-Texte mit sprechenden Variablen besser zu lesen, zu verstehen und zu warten, als einfache Buchstaben oder nur schnell durchnummerierte Variablen, wie x1 und x3 oder so ähnlich.

Als Schreibweise hat sich die sogenannte **CamelCase-Notierung** durchgesetzt. Sie ist zwar nicht bindend, aber allgemein als guter Programmierstil gesehen.
Variablen-Namen beginnen – so die Empfehlung – immer mit einem Kleinbuchstaben oder einem kleingeschriebenen Wort. Danach können Ziffern oder auch weitere Wörter folgen. Umlaute sind zu vermeiden. Die CamelCase-Notierung erhöht die Lesbarkeit von längeren Variablen-Namen dadurch, dass bei neuen Sinn-Elementen mit einem Großbuchstaben begonnen wird. Pinzipiell könnte man auch gut Unterstriche nutzen, die sind aber für die Schreibung von Konstanten vorgesehen.
Richtig geschriebene Variablen-Namen in der CamelCase-Notierung sind z.B.:

teilerSumme anzahlWerte buchstabenZaehler xWert

Im Schulbereich ist die CamelCase-Notierung von Variablen – genau so wie die Vergabe sprechender Variablen-Namen eine Pflicht.

Aufgaben:

1. ***Prüfen zuerst im Kopf, ob die folgenden Bezeichner als Variablen für int-, double- und String-Werte zulässig sind! Begründen Sie Ihre Meinung!***

Aufg.	? Bezeichner	int	double	String	Begründung / Hinweise
a)	x				
b)	xAnfang				
c)	b_Breite				
d)	12x				
e)	4_xxxxxx				
f)	xXxXx_4				
g)	g-J_23				
h)	wert 1				
i)	12+a*c/4%3				

2. ***Vergleichen Sie Ihre Tabelle mit denen anderer Kurs-Teilnehmer! Diskutieren Sie die Gültigkeit untereinander aus!***

Kommen wir zu userem Programm zur Berechnung eines Produktes zurück. Die wichtige Verarbeitungsstelle ist noch mal hervorgehoben.

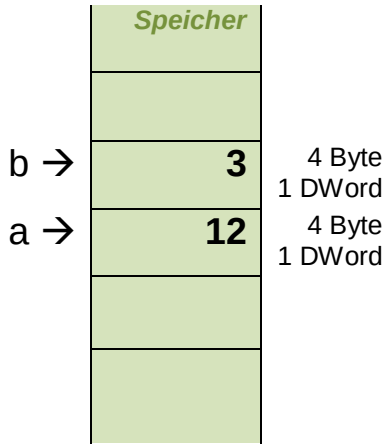
```
1  class B{
2      public static void main(String[] args){
3      int a = 3;
4      int b = 4;
5      int c;
6      c = a * b; //Berechnung
7      System.out.println("Das Ergebnis lautet: "+ c);
8      }
9  }
```

In der Programmier-Praxis stößt man u.a. auf Programme, die z.B. so mit Variablen umgehen:

```
1  class Multi{
2      public static void main(String[] args){
3      int a = 3;
4      int b = 4;
5      a = a * b; //Berechnung
6      System.out.println("Das Ergebnis lautet: "+ a);
7      }
8  }
9
```

Auch wenn sich jedes Mathematiker-Herz dabei streubt, in der Programmierung ist eine solche Benutzung von Variablen zulässig. Auch die Prüfung auf exaktes Rechnen zeigt, dass ei solcher Umgang mit der Variable **a** funktioniert. Das kommt dadurch zustande, dass erst die rechte Seite – also a mal b berechnet wird. Dazu werden die Werte aus den Speicherzellen ausgelesen und verknüpft.

Anschließend – und darauf liegt die Betonung – wird der berechnete Wert in die Speicherzelle geschrieben, die auf der linken Seite angegeben wird. Dabei wird natürlich der alte Wert überschrieben. Würde man nun a abfragen und weiterbenutzen, dann ist nur noch der neue Wert (der Berechnung) verfügbar. Der alte Wert von **a** ist unwiederbringlich verloren. Nur **b** enthält noch seinen ursprünglichen (initialen) Wert.



Gerade bei umfangreichen Berechnungen und komplizierten Verknüpfungen lohnt – die aus "unerklärlichen Gründen" nicht funktionieren – es sich einen Speicher-Plan anzulegen, und immer wieder nach jedem Rechen-Schritt die aktuellen Werte zu notieren.

Das einfache Gleichheits-Zeichen (**=**) ist in der JAVA-Programmmierung immer ein Zuweisungs-Zeichen. Dabei wird der linken Seite (- der dort notierten Variable -) der Wert des Term's von der rechten Seite zugewiesen. Programmieren sagen deshalb statt "ist gleich" auch gern "**ergibt sich aus**" oder "**berechnet (sich) aus**". Stehen auf beiden Seiten die gleichen Variablen-Bezeichner, dann kann man z.B. auch mit "das neue **a** ergibt sich aus dem alten **a** multipliziert mit dem (alten) **b**" die Rechenlogik genauer beschreiben.

Aufgaben:

1. ***Übernehmen Sie untere Tabelle und ermitteln Sie zuerst im Kopf (quasi als Bio-JAVA-System) die Terme! Für die initiale Belegung der Variablen verwenden Sie:***

   x = 3 y = 4 z = 2,5

Teil-Aufg.	Term	Ausgabe Ihres Bio-JAVA-Systems	Ausgabe des JAVA-Programms
a)	x * x + x		
b)	x + y – z		
c)	2 * z / y		
d)	y * (y + y)		
e)	y / x * z		
f)	x = y + 1		
g)	1 / z * y + x - y		
h)	12 % 4 + 2		
i)	12+a*c/4%3		

2. ***Erstellen Sie sich ein JAVA-Programm, dass die einzelnen Terme unter ihrer Teil-Nummerierung berechnet – als Teil-Aufgabe a) in Variable a usw. usf.! Lassen Sie sich die Ergebnisse Nutzer-freundlich anzeigen!***

(Rechen-)Operatoren

Operatoren (Kontext)	Beschreibung		Hinweise / Beispiele
infix; binär			
+	Addition		
-	Subtraktion		
*****	Multiplikation		
/	Division		
%	Modulo Rest der ganzzahligen Division		
verkürzt infix			
+=	Aufaddieren eines Wertes		x += 3 ersetzt: x = x + 3
-=	Abziehen eines Wertes		x -= 3 ersetzt: x = x - 3
***=**	Aufmultiplizieren		x *= 3 ersetzt: x = x * 3
/=	Runterteilen durch einen Wert		x /= 3 ersetzt: x = x / 3
präfix, unär			
-	negatives Vorzeichen		
- -	Dekrement		x- - x=- - 2
+ +	Inkrement		x
postfix, unär			
-	negatives Vorzeichen		
- -	Dekrement		x- - ersetzt: x = x - 1 x=- - 2 ersetzt: x = x - 2
+ +	Inkrement		x++ ersetzt: x = x + 1 x=++ 2 ersetzt: x = x + 2

<u>Hierrarchie / Rangfolge der Operatoren usw.:</u>

++ | -- → +.. | -.. → (|) → * | / | % → + | - → << → >> → >>> → ?: → = → → = → += | /= | %= | += | <<= | >>= | >>>=

höchster Rang … niedrigster Rang

Hinweise:

- die logischen Operatoren werden weiter hinten besprochen (→ 1.3.8. Arbeiten und Anzeigen mit Wahrheits-Werten); hier sind sie nur wegen der Vollständigkeit aufgeführt
- es sind auch spezielle Operatoren enthalten, die für klassische JAVA-Anwendungen keine Rolle spielen

1.3.5. kleine Ausgabe-Hilfsmittel

Betrachtet werden hier einzelne Methoden aus der Klasse String. Diese Methoden stellt das JAVA-System bereit und brauchen von uns nicht mehr programmiert werden.
Die Klasse String wird später noch ausführlich behandelt (→ 1.5.6. String-Methoden). Hier sollen nur einige Methoden (Funktionen) erwähnt werden, die für einfache Programmier-Aufgaben und Ausgaben recht häufig benötigt werden:
Die Methoden werden durch die Angabe des Variablen-Namens eingeleitet. Hinter dem Punkt folgen die Methoden-Namen.

Stringname.**toUpperCase();**
wandelt die Text-Ausgabe vollständig in Groß-Buchstaben um

Stringname.**reverse();**
gibt die Zeichen des Strings in umgekehrter Reihenfolge aus

```
String reversedString = new StringBuffer(orginalString).reverse().toString();
```

Stringname.**charAt(*Position*);**
liefert das Zeichen an der angegebenen Position
die Zählung der Zeichen in einer Zeichenkette (String) beginnt in JAVA mit 0

String test = "Beispieltext";
test.charAt(0);

liefert also das Zeichen 'B' zurück

Weiterhin brauchen wir häufig Mittel zum schöneren Ausgaben von Zahlen. Diese stecken z.B. in der Klasse Math. Hier seien einige der Methoden / Funktionen kurz vorgestellt:

Math.floor(*Zahl*);
liefert die abgerundete Ganz-Zahl (Daten-Typ bleibt double)

Math.ceil(*Zahl*);
liefert die nächst höhere Ganz-Zahl (Daten-Typ bleibt double)

Math.pow(*Zahl*);

Zahl muss nicht unbedingt eine Zahl selbst sein, sondern hier kann auch ein Bezeichner, eine andere Funktion / Methode oder ein Term (eine Berechnung) stehen. Wichtig ist nur, dass das Argument (insgesamt) eine Zahl von dem Daten-Typ ist, der von der Funktion / Methode erwartet wird.

Bitte wundern Sie sich nicht, wenn bei manchen Ausgaben auf einem Mal viele Daten als Text ausgegeben werden. Das liegt daran, dass bestimmte Objekte (→ Objekte) nicht so

einfach, wie Zahlen oder Texte ausgedruckt werden können. Bei Objekten wird dann eine spezielle Methode (→ Methoden) aufgerufen, die **toString()** heißt.

Auf der Konsolen-Ebene brauchen wir auch für Zahlen eine formatierte Ausgabe. Ganz besonders für Konsolen-basierte Tabellen ist die formatierte Ausgabe wichtig.
Das JAVA-System stellt uns hierfür die Methode printf() zur Verfügung. Die Verwendung ist zuerst sehr gewöhnungsbedürftig, nach und nach arbeitet man sich aber ein. Die formatierte Ausgabe kann mit den klassischen Varianten print() und println() kombiniert werden. Dadurch kann am auch den Quell-Code etwas leserlicher machen.

printf(*Formatstring*, *Ausdrücke*)

Formatstring ist ein "normaler" Text mit eingebauten Platzhaltern. An die Stelle der Platzhalten kommen dann die Werte, die in Ausdrücke aneinandergereiht sind. Die Platzhalter erkennt man am %-Zeichen. Es gibt die folgenden …

Platzhalter (für printf())

- **%d** ganze Zahl
- **%e** Gleitkommazahl (im Gleitkomma-Format (Exponenten-Schreibweise))
- **%f** Gleitkommazahl (im Festkomma-Format (definierte Vor- und Nachkomma-Stellen)
- **%s** String / Text

Der Formatstring ist quasi der normale Ausgabetext mit eben solchen Platzhaltern an den gewünschten Ausgabestellen. Hinter dem Formatstring müssen dann soviele Argumente folgen, wie Platzhalter angegeben wurden. Besondes wichtig ist die exakte Passung von Platzhalter-Typ und dem auszugebenen Ausdruck.

```
…
System.out.printf("Die %d. Zahl lautet: %f\n", 1, 2.75);
…
```

Die Kennung \n steht für einen Zeilenumbruch. Dieser muss in der printf()-Methode vom Programmierer explizit angegeben werden.

```
Die 1. Zahl lautet: 2.75
```

Mittels Zahlen können Zahlen und auch eingesetzte Texte noch zusätzlich formatiert werden. Eine erste Zahl zwischen dem Prozent-Zeichen und dem Typ-Kenner bestimmt die Gesamt-Länge der Zahl bzw. des eingesetzten Textes. Bei Zahlen mit Kommastellen kann noch hinter einem Punkt die Anzahl der Nachkommastellen bestimmt werden. Mit **%10.3f** wird also eine **Festkomma-Zahl** mit einem Platzbedarf von **10** Zeichen und **3** enthaltenen Nachkommstellen definiert.

```
…
System.out.printf("Die %d. Zahl lautet: %10.3f\n", 1, 2.75);
…
```

Das Platz-Reservieren und Festlegen der Nachkomma-Stellen ist besonders bei tabellarischen Ausdrucken interessant.

```
Die 1. Zahl lautet:      2.750
```

Solche Konsolen- oder auch ASCII-Tabellen sind praktisch mit den Schleifen voerbunden und werden dort auch noch mal näher beleuchtet. Hier nur mal ein kleiner Vorgeschmack.

```
Startzahl: 4
Endzahl: 6

Durchlauf  |  Zahl  |  Quadrat  |  Kubik
-----------+--------+-----------+----------
     1     |      4 |        16 |      64
     2     |      5 |        25 |     125
     3     |      6 |        36 |     216
```

Als Symbole für die Spalten und Zeilen-Linien werden der senkrechte Strich und das Minus-Zeichen sowie für die Kreuzungen ein Plus-Zeichen eingesetzt.
Eventuell wird zusätzlich mit Gleicheits-Zeichen und Raute / Doppelkreuz eine zusätzliche Absetzung erreicht.

```
Startzahl: 4
Endzahl: 6

Durchlauf  |  Zahl  |  Quadrat  |  Kubik
===========#========#===========#==========
     1     |      4 |        16 |      64
     2     |      5 |        25 |     125
-----------+--------+-----------+----------
     3     |      6 |        36 |     216
```

So etwas ist auch bei Zusammenfassungen oder Spalten-Ergebnissen interessant.

Aufgaben:

1. **Welcher der nachfolgenden Platzhalter ist zulässig und welche nicht? Begründen Sie Ihre Meinung!**

a)	%10d	b)	%10.2d	c)	%10s
d)	%f7.4	e)	%4.8e	f)	%15e
g)	%3.1s	h)	%n23.7	i)	%23.7n

2. **Es soll das / die Argument(e) für die printf()-Methode zur Erzeugung einer unten angegebenen Ausgabezeile zusammengestellt werden. Wie lauetet die Anweisungszeile? (Als nutzbare Variablen sind vorher definiert worden nummer, zahl, quadrat, kubik jeweils als int.)**

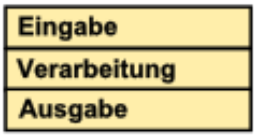
3. **Von einem Nutzer sind die folgenden Daten gespeichert:**

```
String name = "Muster";
String vorname = "Monika";
int gebTag = 3;
int gebMon = 10;
int gebJahr = 1990;
```

Geben Sie jeweils eine Variante für eine "normale" und eine formartierte Ausgabe der folgenden Textzeile an!

```
Der Nutzer heißt Monika Muster und wurde am  3.10.1990 geboren.
```

1.3.6. Eingaben in Konsolen-Programmen



Nachdem wir uns mit dem A und dem V des EVA-Prinzips einführend beschäftig haben, ist nun dringend das E für Eingabe dran.
Natürlich ist eine Variablen-Belegung auch schon so etwas wie eine Eingabe. Gemeint sind aber echte Eingaben von der Tastatur. Da ist JAVA allerdings etwas umständlich. Aus meiner Sicht ist dies ein echtes Manko JAVA.
Aber es gibt natürlich nichts, was nicht geht. Eingaben müssen ja schließlich sein.
Die erste Variante ist recht universell zu verwenden und deshalb auch oft benutzt. Ihre Quellcode-Struktur (Syntax) sieht etwa so aus:

Scanner *ScannerBezeichner* = new Scanner(System.in);
String *EingabeVariable* = *ScannerBezeichner*.nextLine();
***ScannerBezeichner*.close();**

Der ***ScannerBezeichner*** dient zum internen Ansprechen der Eingabe-Methode. Entscheidend ist die ***EingabeVariable***, in der wir die Eingabe (zwischen-)speichern.
Die **.close()** gilt als optional, d.h. man kann sie weglassen. Es gehört aber zum guten Programmierstil sie mitzuschreiben. Der Eingabe-Kanal wird dann sauber geschlossen. Sonst passiert das erst nach dem Block-Ende!
Als Quellcode könnte das mit einer Kontroll-Ausgabe der Eingabe dann so aussehen:

```
1  class EingabeTest{
2      public static void main(String[] args){
3      Scanner scan = new Scanner(System.in);
4      String eingabe = scan.nextLine();
5      scan.close();
6      System.out.println("Die Eingabe lautete: "+eingabe);
7      }
8  }
```

Der leere Bildschrim mit einem blinkenden Cursor ist scheinbar die einzige Reaktion unseres Programms.

```
```

Hier erwartet unser einfaches Eingabe-Konstrukt nun genau auf eine Eingabe von uns.
Machen wir eine solche und bestätigen diese mit [Enter], dann bekommen wir auch die gewünschte Bestätigungs-Anzeige.

```
abc
Die Eingabe lautete: abc
```

Je nach Entwicklungs-Umgebung (IDE) muss man die notwendige Scanner-Klasse auch importieren. In dem Fall notiert man einfach als eine der ersten Anweisungen (noch vor der Klassen-Definition):

import java.util.Scanner;

Dann wird die Bibliothek Scanner für die aktuelle Klasse bereitgestellt und man die verschiedenen Eingabe-Varianten nutzen.
Die unklare Eingabe-Aufforderung muss von uns Programmierern immer Benutzer-freundlich gestaltet werden. Der Nutzer muss wissen, was er eingeben soll. D.h. also, vor dem eigentlichen Eingeben sollte zumindestens eine kurze Ausgabe über den Eingabe-Zweck erfolgen.

```
1  class EingabeTest{
2      public static void main(String[] args){
3      System.out.print("Geben Sie jetzt etwas ein: ");
4      Scanner scan = new Scanner(System.in);
5      String eingabe = scan.nextLine();
6      scan.close();
7      System.out.println("Die Eingabe lautete: "+eingabe);
8      }
9  }
```

Jetzt weiss der Nutzer wenigstens einigermaßen bescheid und kann etsprechend reagieren. Ev. sollte man die Eingabe-Aufforderung auch noch etwas genauer spezifizieren.
Mögliche Ausgabe-Texte könnten so aussehen:

```
Geben Sie jetzt etwas ein: 3,5
Die Eingabe lautete: 3,5
```

```
Geben Sie x ein:
x ?=
Geben Sie die Anzahl an [1..100]:
Wortort:
```

Mehrere Eingaben sind durch Wiederholung der 5. Zeile möglich. Dabei sollte aber erst recht von Aufforderungs-Ausgaben Gebrauch gemacht werden, damit der Nutzer weiss, was er da gerade wann und wo eingibt. Somit müssen die Zeilen 3 und 5 wiederholt aufgerufen werden. In so einem Fall bietet es sich an die Scanner-Erstellungszeile 4 vorzuziehen.

```
1  class EingabeTest{
2      public static void main(String[] args){
3      Scanner scan = new Scanner(System.in);
4      System.out.print("Geben Sie jetzt den 1. Wert ein: ");
5      String eingabe1 = scan.nextLine();
6      System.out.print("Geben Sie jetzt den 2. Wert ein: ");
7      String eingabe2 = scan.nextLine();
8      scan.close();
9      System.out.println("Die 1. Eingabe lautete: "+eingabe1);
10     System.out.println("Die 2. Eingabe lautete: "+eingabe2);
11     }
12 }
```

Eine konkrete (prinzipiell wiederholbare) Eingabe-Sequenz ist **blau** hervorgehoben.

Aufgaben:

1. *Warum akzeptiert das System die falsche JAVA-Gleitkommazahl 3,5 in obigen Beispiel? Erklären Sie genau!*
2. *Erstellen Sie ein Programm, in dem Vorname und Nachname abgefragt wird und dann in zwei Varianten angezeigt wird!*
   a) Nennname: Vorname Nachname
   b) Listenname: Nachname, Vorname
3. *Die folgenden Worte sollen von einem Nutzer einzeln eingegeben werden. Ihr Programm soll dann daraus einen ordentlichen Satz anzeigen!*
   Ein, Java-Programm, ist, keine, einfache, Sache.

Aufgaben:

4. *Bekommen Sie das Programm von 3. auch mit maximal Variablen hin?*

Der primäre Eingabe-Datentyp der Scanner-Klasse ist String. Das ist insgesamt auch Windows-typisch. Viele Eingaben liegen erst einmal nur als String vor und müssen dann nachträglich in andere Datentypen umgewandelt werden (→ 1.3.7.1. Typ-Umwandlungen (Type Casting und Parsing)).
Die Klasse Scanner stellt aber auch Methoden zum direkten Einlesen von Ganzzahlen und Fließkommazahlen zur Verfügung.

```
int zahl1 = scan.nextInt();
```

bzw.

```
double zahl2 = scan.nextDouble();
```

alternativ:

```
float zahl3 = scan.nextFloat();
```

Aufgaben:

1. *In einem Semester gibt es immer exakt 5 Noten. In Ihrem Programm sollen diese Noten abgefragt und der Mittelwert berechnet werden!*
2. *Lassen Sie sich von einem Nutzer 2 Punkte einer Gerade angeben und berechen Sie daraus die Geraden-Gleichung y = mx + n! Zeigen Sie die berechnete Geraden-Gleichung auch in dieser Form an!*

Eine andere Möglichkeit ist die Nutzung einer weiteren Eingabe-Klasse von JAVA, die sich BufferedReader nennt. Aber die Konstrukte sehen nicht wirklich besser aus:

```
BufferedReader puffer = new BufferedReader(new InputStreamReader(System.in));
eing= puffer.readLine();
```

Die für diese Eingabe-Art notwendige Klasse heißt **java.io**. Sie wird mit:

```
import java.io.*;
```

in das eigene Programm integriert. Die Verwendung von zwei Variablen ist identisch zur ersten Eingabe-Variante oben. ***puffer*** ist eine Art Zwischen-Auffangbecken, aus dem dann die Eingabe für die Variable ***eing*** selektiert wird.
Auch hier die mit Ausgaben erweiterte Variante für mehrere Werte:

```
BufferedReader puffer = new BufferedReader(new InputStreamReader(System.in));
System.out.print("Geben Sie Wert1 ein!: ");
String eingabe1 = puffer.readLine();
System.out.print("Geben Sie Wert2 ein!: ");
String eingabe2 = puffer.readLine();
```

Die Eingaben sind hier wieder ausschließlich Texte. Sie müssen ev. für Berechnungen vor der Benutzung in echte Zahlen umgewandelt werden (→ 1.3.7.1. Typ-Umwandlungen (Type Casting und Parsing)). Beispielhaft dafür stehen die nächsten zwei Anweisungen:

```
int ganzzahl1 = Integer.parseInt(eingabe1);
double fliesszahl2 = Double.parseDouble(eingabe2);
```

Ev. muss hier auf Exceptions (Unterbrechungen / Fehler-Abbrüche) geachtet werden und diese durch passenden Programm-Code abgefangen werden, was wir später erklären (→ 1.5.2. Exzeptions (exception's)).
Die Problematik der Eingabe vereinfacht sich etwas, wenn wir graphische Bedien-Programme nutzen (→ 1.6. Nutzung graphischer Oberflächen).

Aufgaben:

1. *Ein kleines Programm soll eine Gleitkommazahl von der Konsole annehmen, diese mit einer zweiten einzugebenen Zahl vom Typ Ganzzahl multiplizieren und das Ergebnis als Gleichung anzeigen! (Der Nutzer muss ordnungsgemäß geführt werden!)*
2. *Erstellen Sie ein Programm, welches den Namen und das Alter des Nutzers abfragt und in ordentlichen Sätzen ausgibt! Beim ersten Satz soll das Alter zuerst und dann der Name genannt werden! Beim zweiten Satz sollen Alter und Name in der Position getauscht werden!*
3.

1.3.6.1. Eingaben für Konsolen-Programme im Java-Editor

Der Java-Editor stellt eine eigene Klasse / Bibliothek mit Eingabe-Ausgabe-Methoden zur Verfügung. Mit dieser werden Eingaben deutlich eleganter, übersichtlicher und logisch klarer realisiert. Weiterhin korrespondieren sie auch mit dem Struktogramm-Editor im Java-Editor (→ 1.1.2.1.3. der Java-Editor als Struktogramm-Editor bzw. 3.x. Planung, Entwicklung und Visualisierung von Algorithmen).

```
1  class EingabeTest {
2    public static void main(String[] args){
3    int eingabe1;
4    double eingabe2;
5    String eingabe3;
6    eingabe1 = InOut.readInt("Geben Sie eine Ganzzahl ein !: ");
7    eingabe2 = InOut.readDouble("Geben Sie eine Fließzahl ein !: ");
8    eingabe3 = InOut.readString("Geben Sie einen Text ein !: ");
9    System.out.println("Die Ganzzahl lautete: "+eingabe1);
10   System.out.println("Die Fließzahl war: "+eingabe2);
11   System.out.println("Der Text hieß: "+eingabe3);
12   }
13 }
```

```
Geben Sie eine Ganzzahl ein !: 3
Geben Sie eine Fließzahl ein !: 4.0
Geben Sie einen Text ein !: Hallo Computer!
Die Ganzzahl lautete: 3
Die Fließzahl war: 4.0
Der Text hieß: Hallo Computer!
```

Das sieht doch super aus. Jeder versteht es und der Code ist später leicht zu verändern oder zu erweitern.
Werden nichtpassende Daten eingegeben, dann erhalten wir auf der Konsole eine Exception-Fehlermeldung aus der von InOut benutzten JAVA-API.

1.3.7. die JAVA-Datentypen

Die wichtigsten Datentypen haben wir schon besprochen und benutzt. Für normale Anwendungen sind sie vorzuziehen, da sie auch auf allen Systemen sicher gleichartig definiert sind. Für spezielle Zwecke bietet JAVA weitere Datentypen an. Sie besitzen oft kleinere oder größere Werte-Bereiche. Die Konsequenz sind weniger oder mehr Byte, die pro Variable benötigt werden.
Bei solchen Anwendungen, wie wir sie schreiben ist das selten relevant. Aber in Fällen, wo z.B. für einen Körper 1'000'000 Punkte (mit x-, y- und z-Koordinaten) gespeichert werden müssen, dann ist es schon ein Unterschied, ob pro Wert 2 oder 8 Byte benötigt werden.

Aufgabe:
Berechnen Sie die Speicherbedarf für eine 3D-Figur, von der 2,3 Mill. Punkte bekannt sind! Die Werte sollen als short, long, double oder float gespeichert werden. Geht das z.B, wenn die Werte zwischen 0 und 45'000 schwanken?

klassische Daten-Typen und ihre Notierung in JAVA:

Umschreibung des Daten-Typs	Schlüsselwort für JAVA	Konventionen und / oder Grenzen	Beispiele / Hinweise
Einzel-Zeichen	**char**		braucht: 2 Byte (Unicode!)) 'a' '1' '\n' → '\u0041' →
Wahrheitswert	**boolean**	true oder false	braucht: 1 Byte
Byte, kleine Ganzzahl	**byte**	-128 .. 127	braucht: 1 Byte
Word	**short**	-32768 .. 32767	braucht: 2 Byte
ganze Zahlen (ohne Kommastellen)	**int**	-27147483648 .. 2147483647	braucht: 4 Byte
große Ganzzahlen	**long**	9223372036854775808 .. 9223372036854775807	braucht: 8 Byte
sehr kleine / große Fließkomma-Zahlen	**float**	$-1{,}4*10^{-45}$ .. $3{,}4028*10^{38}$	braucht: 4 Byte
reele Zahlen (mit möglichen Kommastellen)	**double**	$-4{,}9*10^{-324}$ .. $1{,}79769*10^{308}$	braucht: 8 Byte

erweiterte / komplexe Daten-Typen und ihre Notierung in JAVA:

Umschreibung des Daten-Typs	Schlüsselwort für JAVA	Konventionen und / oder Grenzen	Beispiele / Hinweise
Zeichen-Ketten (Zeilen-Texte)	String		heute Unicode üblich: verwendet 2 Byte pro Zeichen
Instanzen von einer Klasse	*Klassenname*	Klasse muss in einer separaten Datei definiert sein!	

1.3.7.0. Daten-Typen für Freaks

Wie die Daten der einzelnen Datentypen durch JAVA im Speicher abgespeichert werden, ist für die Programmierung erst einmal nicht notwendig. Einen direkten Zugriff auf die Elemente eines Datentyps im Speicher haben wir nur in wenigen Fällen. Meist schirmt JAVA hier sinnvoll ab. Manchmal kommt es jedoch ungewöhnlichen Reaktionen des Systems. Solche fehlenden Rand-Informationen haben schon Raketen abstürzen lassen, weil sie von den Programmierern nicht ausreichend beachtet wurden. Wir kommen gleich darauf zurück.

Daten müssen immer in einer bestimmten Form gespeichert werden. In den klassischen Systemen besteht eine Speicherzelle prinzipiell aus 8 bit. D.h. es lassen sich 8 binäre Informationen speichern. So eine Einheit heißt Byte. Ein Byte entspricht also 8 bit.

Mit regulären Einheiten geschrieben sieht das so aus: 1 Byte = 8 bit bzw. 1 B = 8 b.

In modernen Systemen werden nur mehrere 8-bit-Einheiten zusammengefasst. So erhält man 16-, 32-, 64- und 128-bit-Speicher. Man spricht dann von Word, Doubleword (DWord), Quadword (QWord) bzw. Octaword (Double Quadword, DQWord).

Von modernen Prozessoren wird die gesamte Speicher-Einheit – also z.B. ein DWord – mit einer Adresse angesprochen. Somit sind dann einem DWord genau 32 bit zugeordnet.

In Erklärungen tuen wieder aber immer so, als würde es sich um den alten klassischen 8-bit-Speicher handeln würde. Das ist übersichtlicher und besser verständlich.

Die vorgestellten Prinzipien verändern sich durch die größere bit-Anzahl praktisch nicht.

Der normale RAM (Arbeitsspeicher) eines PC's ist in –bit-Einheiten struktoriert.

Größere Speicher-Einheiten findet man z.B. in den Prozessoren. Sie sollen möglichst viele Informationen parallel abarbeiten können. Bei den 64-bit-Prozessoren der i3-, i5- und i7-Klasse findet man Speicherzellen (sogenannte Register) bis zu einer Breite von 128 bit.

Beim Erläutern der Speicher-Prinzipien wird meist auf das etwas überholte 8-bit-System zurückgegriffen und die Speicherzellen als 1-Byte-System betrachtet. Die Grundprinzipien ändern sich bei Mehr-Byte-Systemen nicht. Es werden nur mehr bit-Stellen.

Bleiben wir beim Byte – also einer einzelnen Speicherzelle. Wenn diese aus 8 bit besteht, dann kann jede bit-Speicher-Stelle genau 2 Zustände einnehmen – 0 oder 1. Insgesamt ergeben sich so 256 mögliche Zustände für eine Speicherzelle.

bit-Muster	Variante
0000 0000	1
0000 0001	2
0000 0010	3
0000 0011	4
0000 0100	5
0000 0101	6
0000 0110	7
0000 0111	8
0000 1000	9
0000 1001	10
…	…
1111 1000	249
1111 1001	250
1111 1010	251
1111 1011	252
1111 1100	253
1111 1101	254
1111 1110	255
1111 1111	256

Aufgaben:

1. ***Nehmen wir an eine Speicherzelle wäre 4 bit groß (früher gab es wirklich solche Rechner!). Geben Sie alle möglichen bit-Muster für die Speicherzellen an!***
2. ***Überlegen Sie sich, wieviele bit-Muster in Speicherzellen gespeichert werden können, die 2, 3, … , 7 bit breit sind! Tragen Sie die Werte in eine Tabelle in Ihren Mitschriften ein!***

Speicher-Stellen [bit]	1	2	3	4	5	6	7	8
Anzahl möglicher Zustände	2							256

3. ***Ermitteln Sie eine Berechnungs-Vorschrift um für beliebige Speicher-Breiten die Anzahl möglicher unterschiedlicher Zustände / Informationen zu berechnen!***
4. ***Berechen Sie die möglichen Zustände für die nachfolgenden Speicher-Breiten!***

Speicher-Stellen [bit]	16	32	64	100	128
Anzahl möglicher Zustände					

1.3.7.0.1. Ganzzahlen

Byte

word

Integer

LongInt

Konvertierung ohne Daten-Verlust und Probleme:

byte → short → int → long (→ float → double)

Inkrement

Long

Exkurs: Absturz der ersten Ariane-5-Rakete

Die Vorgänger-Version Ariane 4 war ein sehr erfolgreiches Raketen-Projekt mit relativ wenigen Unfällen und Problemen.
Für größere Nutzlasten konstruierte man auf der Basis der Ariane 4 die größere Ariane 5. Große Teile der Software wurden ebenfalls übernommen.
Beim Jungfern-Start am 04.09 1996 endete aber die Erfolgs-Geschichte des Ariane-Projekts.
Wegen schlechter Sicht kam es zwar zu einer Verschiebung des Countdown um eine Stunde, der eigentliche Start und die ersten 36 Sekunden lief auch alles normal.
Zuerst stürzte das back-up-System der Trägheitskontrolle ab. Kurz danach erging es dem Hauptsystem der Trägeitskontrolle genau so. Bestimmte Steuerdüsen bewirkten eine starke Neigung der Rakete so dass dann bei x+42 s vom Boden aus die Selbstzerstörung aktiviert wurde.

Q: sputniknews.com (ESA-S. Corvaja)

Das eigentlich Problem war zwar nicht die im System benutzte Ganzzahl, sondern exakterweise lief bei der Ariane 5 die Umwandlung einer 64-bit-Gleitkommazahl in eine 16-bit-Ganzzahl schief. Grundsätzlich war es aber ein Überlauffehler bei der 16-bit-Ganzzahl.
Vielfach werden solche Überläufe abgesichert, aber aus irgendeinem Grund hatte man dieses an dieser Stelle nicht in die Software eingebaut.
Wie wir oben gesehen haben sind Ganzzahlen praktisch duale Zahlenkreise. Wird die Zahl bit-mäßig größer, kann es bei Überschreiten der Grenze zur Fehl-Interpretation der Zahl kommen. Die Zahl wird ja auf einem mal als negative Zahl aufgefasst. Diese erfordert eine bestimmte Korrektur (die Lage der Rakete ist lt. dieser Zahl falsch). Das System reagiert heftig. Da die Werte aber immer noch die Interpretations-Grenze überschreiten, macht das System immer weiter so.
Im Prinzip kann man sich das so vorstellen. Die aktuelle Zahl ist kurz vor der Typ-Grenze z.B. 127 (bei gedachter 8-bit-Integer). Die gewünschte Zahl soll etwas größer sein. Also addiert das System dazu (dual wird es z.B. 128) Nun wird aber durch das Addieren die Obergrenze überschritten und die Zahl als negeativ interpretiert. Das bit in der vordersten Stelle ist ja kein Zahlen-Bit mehr sondern das Vorzeichen. Die resultierende Interger-Zahl ist aufeinmal negativ (z.B.: -128). Die Differenz zur gewünschten Zahl ist also nicht kleiner, sondern deutlich größer geworden und die Software addiert kräftig dazu. Wirklich lösen tut sich das Problem dadurch nicht – und die Katastrophe nicht mehr zu verhindern.
Bei der Leistungs-schwächeren Ariane 4 wurden die Werte nie so groß, dass sie die Typ-Grenze überschritten. Eine mögliche Grenz-Überschreitung mussten die Programmierer also nicht beachten.
Der Schaden dieses "kleinen" Programierfehlers belief sich auf ungefähr 1,7 Mrd. DM (850 Mio. €). Weitere Schäden ergaben sich durch den Ausfall der Ariane 5 als Träger-System. Es kam zu einer dreijährigen Verzögerung des Projekts.
Einen zusätzlichen Zungenschlag bekommt die Katastrophe noch dadurch, dass man bei der Ariane 5 das problematische System aus Sicherheits-Gründen doppelt eingebaut hatte, um eben genau solche Unfälle durch Ausfall des Trägheitssystems verhindern zu wollen. Weiterhin wäre die Software-Stelle selbst nach dem Start gar nicht mehr direkt notwendig gewesen, weil sie eine Kalibrierung betraf. Man hatte diese aber früher immer länger laufen lassen um kurzfristige Countdown-Verschiebungen abfangen zu können. Ein Neustart der gesamten Kalibrierung hätte nämlich 45 min gedauert.

Quellen:
https://sputniknews.com/science/201611171047544604-ariane-5-navigation-satellites/
http://www4.in.tum.de/lehre/seminare/ps/WS0203/desaster/Riedl-Arianne5-Ausarbeitung-27-11-02.pdf

Links:
https://www.youtube.com/watch?v=PK_yguLapgA (Video)

1.3.7.0.2. Gleit- oder Fließkommazahlen

double

float

extended

Exkurs: Rundungsfehler

Beispiel: Patriot-System
Patriot-Raketen sind Abfangsysteme für feindliche Raketen. Für die Berechnung des Treffpunktes mit der feindlichen Rakete müssen Weg und Geschwindigkeit exakt in Beziehung gesetzt werden. Das Problem ist hier die exakte Zeit. Das Scud-System hatte dazu auch eine interne Uhr, diese hatte aber nach rund 100 Betriebsstunden einen Rundungsfehler von 0,34 s. Das macht bei einer Abfang-Geschwindigkeit von 1,676 km/s einen Wegefehler von 0,57 km – knapp daneben.
Genau das passierte im Golfkrieg am 25.02.1991. Eine irakische Scud-Rakete konnte nicht abgefangen werden und kostete 28 Soldaten das Leben.

1.3.7.0.3. Einzelzeichen - Charakter

1 Byte char

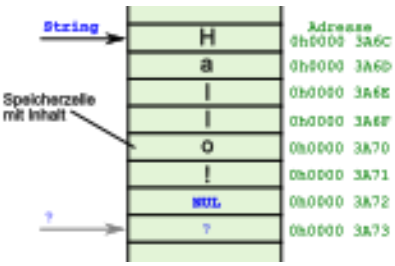
2 Byte Unicode

4 Byte Unicode

1.3.7.0.4. Zeichenketten - Strings

String

Null-terminiert



1.3.7.0.5. Zählen der Speicherzellen (Dualzahlen und ihre Skalierung)

klassische Zähl-Einheiten

1'024 bit	1 kbit	k ≡ 1'000	K ≠ k = 1'000 = 10^3
1'024 Kbit	1 Mbit	M ≡ 1'000'000	M = 10^6
1'024 Mbit	1 Gbit	G ≡ 1'000'000'000	G = 10^9
1'024 Gbit	1 Tbit	T ≡ 1,000 * 10^{12}	T = 10^{12}
1'024 Tbit	1 Pbit	P ≡ 1,000 * 10^{15}	P = 10^{15}

8 bit	1 Byte		
16 bit	2 Byte	2 Byte = 1 Word	nur, wenn die Bit's auch semantisch zusammengehören (z.B. Speicher-Adresse)
32 bit	4 Byte	4 Byte = 1 DoubleWord	
64 bit	8 Byte	8 Byte = 1	
1'024 Byte	1 kByte	k ≡ 1'000	K ≠ k = 1'000 = 10^3
1'024 KByte	1 MByte	M ≡ 1'000'000	M = 10^6
1'024 MByte	1 GByte	G ≡ 1'000'000'000	G = 10^9
1'024 GByte	1 TByte	T ≡ 1,000 * 10^{12}	T = 10^{12}
1'024 TByte	1 PByte	P ≡ 1,000 * 10^{15}	P = 10^{15}

				Fehler [%] zwischen den Skalen
1'024 Byte	1 KiB	Ki ≡ K ≡ 1'024	Ki = K ≠ k = 1'000 = 10^3	
1'024 KiB	1 MiB	Mi ≡ 1'048'576	2^{20} = Mi ≠ M = 10^6	
1'024 MiB	1 GiB	Gi ≡ 1'073'741'824	2^{30} = Gi ≠ G = 10^9	
1'024 GiB	1 TiB	Ti ≡ 1,0995 * 10^{12}	2^{40} = Ti ≠ T = 10^{12}	
1'024 TiB	1 PiB	Pi ≡ 1,1259 * 10^{15}	2^{50} = Pi ≠ P = 10^{15}	

interessanterweise werden in Windows selbst und auch in vielen Programmen intern die Dualzahl-basierte Skalierung benutzt, aber die dezimale angezeigt.

Umrechnungen zwischen den Zähl-Einheiten (Beispiele)

1.3.7.1. Typ-Umwandlungen (Type Casting und Parsing)

Benötigt man z.B. eine int-Zahl und hat diese aber nur als double-Variable verfügbar, dann hat man in JAVA ev. ein Problem. Das einfache Übertragen ist nicht immer möglich. So etwas stört z.B. beim Aufruf von Funktionen, die nur für Ganzzahlen definiert sind.
Die einzelnen primitiven Datentypen lassen sich vielfach ineinander umwandeln. Das nennt man Type Casting (dt.: Typ-Umwandlung, Typ-Anpassung).
Wenn bei einer Umwandlung keine Information verloren geht, wie z.B. beim Umwandeln von int in double, dann ist das ein Umsetzen gar kein Problem. Wir brauchen nicht einmal extra Programmier-Aufwand betreiben – JAVA macht das von sich aus. Das nennt man implizites Type Casting.

implizites Type-Casting

nach:	**Ziel-Datentyp**						
von …	**byte**	**short**	**int**	**long**	**float**	**double**	
byte		✓	✓	✓	✓	✓	
short			✓	✓	✓	✓	
int				✓	✓	✓	
long					✓	✓	
float						✓	
double							
String							

✓ … geht ohne Probleme und Informationsverluste; ohne extra Umwandlungen

kein Informations-Verlust

Das implizite Type-Casting wird auch offensichtlich, wenn man sich die möglichen Zahlen und den Speicher vorstellt. Es werden immer kleinere Zahlen in größere Bereiche umgesetzt, was eben problemlos funktioniert. Auch im Speicher werden für die kleineren Zahlen immer kleinere Gruppen von Speicherzellen gebraucht. In den größeren Zahlen-Typen lassen sich die neuen Bit's ganz leicht und fehlerfrei mit Nullen auffüllen.

Byte 1

1	0	1	0	0	1	1	1

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

0	0	0	…	0	0	1	1	0	1	0	1	0	0	0	1	0	0	1	0	1

Es kommt niemals zu einem Informations-Verlust. Allerdings erkaufen wir uns die Umsetzung immer auch mit mehr benötigtem Speicherplatz. Bei den vielleicht 30 Variablen in unseren Programmen spielt das kaum eine Rolle. Wenn man aber eine 3-D-Form aus x-Millionen von Einzel-Punkten zusammensetzt, dann kann man schon an die Grenzen des Machbaren kommen. Der passend große Speicher muss eben auch da sein.

Beim zu erwartenden Informations-Verlust muss das explizite Type Casting durchgeführt werden. Dabei ist nicht der konkrete Fall interessant, sondern das Prinzip. Beim Umwandeln einer double-Gleitkommazahl in einen Ganzzahl-Typ gehen immer die Nachkomma-Stellen verloren. Das kann uninteressant sein, wie z.B. bei 1000000.0000001. Hier spielt die 7 Nachkomma-Stelle praktisch überhaupt keine Rolle, sie könnte sogar nur ein Rundungs-Fehler sein oder durch die Rechengenauigkeit des System bedingt sein.
Anders bei 0.0004523, hier ist eine Null eben nicht äquivalent zur gemeinten Zahl. Der Informations-Verlust kann hier dramatisch sein. (Man denke an die Probleme, die z.B. bei der Teilung durch diese Zahl auftreten könnten!)
Explizites Type Casting wird durch vordefinierte Methoden erzwungen. Die Umwandlung funktioniert durch den "Angabe" des Datentyp's in runden Klammern vor dem umzuwandelnden Wert.

```
…
int i = 0;
double d = 5.73;
…
i = (int) d;
…
```

Interessanterweise bekommen wir beim Ausführen des obigen Code's aber keine 6 in i sondern eine 5. Das liegt daran, dass das Type Casting hier durch Abschneiden der Nachkomma-Stellen erfolgt.
Wie wir sehen, funktioniert auch das explizite Typ Casting nicht schrankenlos.

implizites und explizites Type-Casting

nach:	Ziel-Datentyp						
von …	**byte**	**short**	**int**	**long**	**float**	**double**	
byte		✓	✓	✓	✓	✓	
short			✓	✓	✓	✓	
int				✓	✓	✓	
long					✓	✓	
float			**(int) …**			✓	
double							
String							

✓ … geht ohne Probleme und Informationsverluste; ohne extra Umwandlungen

Type Casting lässt sich auch innerhalb der Objekt-Hierrarchie anwenden. Dazu dann mehr bei der genaueren Besprechung der Klassen und Objekte (→ 1.4.0. Grundlagen).

Niederrangige Klassen (Objekte, Subklassen) können auf ihre Basis- bzw. übergeordneten Klassen (Objekte, Superklassen) zurückgeführt werden. Dabei gehen wie bei den oben besprochenen Zahlen-Typen imm die spezielleren Details der Unterklasse verloren.

Bei komplexeren / schwierigeren Umwandlungen, wie z.B. die eines Textes (String) in eine Zahl stellt JAVA spezielle Funktionen / Methoden zur Verfügung. Diese zerlegen den übergebenen String zeichenweise, um die Zahl heraus zu extrahieren. Man nennt so ein Durchsuchen parsen (*to parse* → dt.: durchstreifen, analysieren, zerlegen, fassen)

```
…
s = "2.73";      // Achtung! schon im String muss die engl. Punkt-
                 // Notierung beachtet werden!
x = Integer.parseInt(s);
…
```

Zu den Besonderheiten der Umwandlung von Zahlen in die Objekt-Datentypen (weshalb Integer groß geschrieben wurde!) gibt es zusätzliche Informationen im zugehörenden Kapitel (→ 1.3.10.2. Objekt-Datentypen).
Da beim Parsen immer mit Fehlern (Exceptions) zu rechnen ist, sollte diese durch geeignete Abfang-Mechanismen (→ 1.5.2. Exzeptions (exception's)) aufgenommen werden. Beim Umwandeln eine sStrings in eine Zahl sollte die zugeordnete NumberFormatEcxeption ausgewertet werden.

Type-Casting und Parsing

nach:	**Ziel-Datentyp**						
von …	**byte**	**short**	**int**	**long**	**float**	**double**	
byte		✓	✓	✓			
short			✓	✓			
int				✓	✓	✓	
long					✓	✓	
float			**(int) …**			✓	
double							
String							

✓ … geht ohne Probleme und Informationsverluste; ohne extra Umwandlungen

kein Informations-Verlust

mit Informations-Verlusten ist zu rechnen

z.T. problematische Umwandlung

geht nicht; inkompatible Datentypen

1.3.7.3. Erzeugen von Zufallszahlen

Das Erzeugen von Zufallszahlen ist auch eine Möglichkeit an Testwerte für unsere Programme zu kommen. Aber auch für andere Zwecke werden Zufallszahlen gebraucht. Man denke an das Erstellen von Spielen oder Statistik-belastete Programme.

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

Die von random() zurückgelieferte Zufallszahl ist größer gleich 0,0 und kleiner als 1,0. Wir müssen hier darauf achten, dass die 1,0 niemals als Rückgabewert dabei ist.
Um nun andere Zahlenbereiche abzudecken arbeitet man mit Faktoren und Summanden. Die so manipulierten Zufallszahlen müssen dann wahrscheinlich noch auf den richtigen Datentyp gecastet werden (→ 1.3.7.1. Typ-Umwandlungen (Type Casting und Parsing)).
Nehmen wir als Beispiel einen "normalen" 6er Würfel. Wir brauchen als Ergebnis Ganzzahlen zwischen 1 und 6.
Zuerst spreizen wir das von **random()** gelieferte Ergebnis mit dem Faktor 6:

```
…
double wurf = Math.random() * 6;
…
```

Dadurch liegen die Werte für wurf nun im Bereich 0,0 ≥ wurf < 6,0.

Mittels einfachem Casting auf Ganzzahl:

```
…
int wurf = (int) Math.random() * 6;
…
```

erhalten wir die Zahlen 0 bis 5. Das Casting einer **double** auf ein **int** bewirkt immer das Abschneiden der Nachkommastellen. Nun heben wir mit der Addition von 1 die Einzelwerte einfach an:

```
…
int wurf = (int) (Math.random() * 6 ) +1;
…
```

und bekommen die Ergebnisse 1 .. 6.
Allgemein könnte man sich die folgende Code-Sequenz merken, die man später auch in eine extra Methode stecken könnte:

```
…
int varianten = 6;     // Anzahl der erwarteten unterschiedlichen Werte
int kleinste = 1;      // kleinster Wert
int zufallszahl = (int) (Math.random() * varianten) + kleinste;
…
```

Hat man nur die kleinste und grösste Ganzzahl, dann geht auch:

```
…
int groesste = 6;     // größter Wert
int kleinste = 1;     // kleinster Wert
int zufallszahl = (int)(Math.random()*(groesste-kleinste+1)+kleinste;
…
```

Na dann, fröhliches Würfeln! Gut testen kann man die Sequenzen, wenn man sie mittels Schleifen vielfach wiederholt (→ 1.3.9.3. Schleifen).

Aufgaben:

1.

2.

3.

1.3.8. Arbeiten und Anzeigen mit Wahrheits-Werten

Logische Variablen und Werte sind für uns normale Nutzer erst einmal etwas ungewöhnlich. In unserem Denken spielen sie aber eine große Rolle. Z.B. dürfen zwei Personen nur heiraten, wenn sie jeweils volljährig und derzeit nicht verheiratet sind. Das könnte man z.B. umgangssprachlich so schreiben:

Eine Heirat der Personen x und y ist möglich, WENN beide Personen volljährig und nicht (selbst) verheiratet sind

Etwas Logik-orientierter ist die Formulierung:

Heirat_möglich → **WENN** volljährig(Person1) **UND** volljährig(Person2) ↵
UND NICHT (verheiratet(Person1) **ODER** verheiratet(Person2))

Dabei repräsentieren **Heirat_möglich**, **volljährig(x)** und **verheiratet(x)** immer jeweils Wahrheitswerte, die **WAHR** oder **FALSCH** sein können. Das **x** steht für eine mögliche Person. **UND**, **ODER** und **NICHT** sind logische Operatoren, mit denen man logische Werte verknüpfen kann. Ich betone die logischen Operatoren und Werte mit einer Großschreibung, um sie von den anderen Texten abzusetzen.
Die informatische Formulierung – wie sie von den meisten Programmiersprachen prinzipiell erwartet wird – zieht die Bedingung(en) vor und teilt dann in den WAHR- und FALSCH-Zweig (DANN .. SONST ..)

WENN volljährig(Person1) **UND** volljährig(Person2) ↵
UND NICHT (verheiratet(Person1) **ODER** verheiratet(Person2))
DANN Heirat_möglich
SONST NICHT Heirat_möglich

Die elementaren logischen Operationen wollen wir hier aus unserer Erinnerung (aus dem mathemtischen, philosophischen oder informatischen Unterricht) herausholen bzw. für die jenigen, die sich gar nicht mehr erinnern können, kurz vorstellen.
Ich verwende gleich die Schreibweisen von JAVA, um nachher nicht noch mal den Übersetzungs-Aufwand zu haben.
Die einfachste Operation ist die Negation. Die Negation wird auch mit NICHT oder **NOT** beschrieben. Das mathematische Zeichen ist ein Überstrich oder ein gewinkelter Vorstrich (¬). Bei der Negation welchselt der Wahrheitswert. Aus WAHR (**true**) wird FALSCH (**false**) oder umgekehrt. Das JAVA-Zeichen für die Negation ist ein Ausrufezeichen vor dem Ausdruck oder der Variable.

Nicht-Operator

A	!A
true	**false**
false	**true**

In der nebenstehenden Funktions-Tabelle wird der Groß-Buchstabe A als Variable verwendet. Die möglichen Belegungen stehen darunter. Zur besseren Kennzeichnung wurden die wahren Werte nochmals grünlich unterlegt.
Während die Negation nur mit einem Operator arbeitet, braucht man für die anderen logischen Operatoren immer zwei. In der Tabelle durch **A** und **B** vertreten.
Die UND-Verknüpfung (&) erwartet zwei wahre Aussagen um selbst wahr zu sein.
Schon wenn nur einer der beiden Operatoren unwahr ist, dann ist auch die UND-Verknüpfung unwahr.
Die UND-Verknüpfung wird auch **AND** geschrieben.

UND-Verknüpfung

A	B	A && B
true	true	**true**
true	false	**false**
false	true	**false**
false	false	**false**

Das mathematische Zeichen ist ein U oder das Kaufmanns-Und (&). Man findet ein spezielles Zeichen (U) in vielen Zeichensätzen / Schriftarten. In JAVA müssen aber zwei aufeinanderfolge Kaufmanns-Und verwendet werden (**&&**).

Die dritte – elementare – Verknüpfung ist ODER bzw. **OR**. Dabei reicht es, wenn wenigstens eine der beiden Variablen / Aussagen wahr ist, um ein wahres Ergebnis zu erhalten.
Die mathematische Notation erfolgt über ein umgedrehtes U – also ∩. Die JAVA-Notierung sind zwei senkrechte Striche (**||**). Diese erhält man durch die Tastenkombination [AltGr] + [<].

ODER-Verknüpfung

A	B	A || B
true	true	**true**
true	false	**true**
false	true	**true**
false	false	**false**

Aufgaben:

1. ***Sehr häufig wird auch von einem Exklusiven ODER gesprochen. Ist das nur ein anderer Name für unser besprochenes ODER? Erklären Sie genau!***
2. ***Stellen Sie Funktions-Tabellen für die folgenden logischen Verknüpfungen auf:***
   a) NICHT (**A** UND **B**)
   b) NICHT (**A** ODER **B**)
   c) **A** UND NICHT **B**
   d) NICHT **A** ODER NICHT **B**
3. ***Vergleichen Sie die Funktionen von Aufgabe 2a) und b) mit den elementaren logischen Operationen! Welche Konsequenzen lassen sich für die spätere Verwendung von logischen Operatoren ziehen?***

Welche Situationen sind nun für unser obiges **Heirat_möglich**-Beispiel denkbar?:

Person1		Person2			
volljährig	**verheiratet**	**volljährig**	**verheiratet**	**Heirat_möglich**	
true	true	true	true	false	
true	true	true	false	false	
true	true	false	true	false	
true	true	false	false	false	
true	false	true	true	false	
true	**false**	**true**	**false**	**true**	
true	false	false	true	false	
true	false	false	false	false	
false	true	true	true	false	
false	true	true	false	false	
false	true	false	true	false	
false	true	false	false	false	
false	false	true	true	false	
false	false	true	false	false	
false	false	false	true	false	
false	false	false	false	false	

Zuerst scheint es einem so, als würden die Wahrheitswerte eine Nebenrolle spielen. Je mehr man aber in die Informatik oder speziell in die Programmierung einsteigt, umso bedeutsamer wird der Datentyp **boolean**.
Wahrheitswerte lassen sich völlig unkompliziert mit der println()-Funktion ausgeben. Das kann schon mal interessant werden, wenn man sich innerhalb eines Programm-Ablaufes zwischendurch die einzelnen Wahrheitswerte anschauen muss, um z.B. einen Fehler zu lokalisieren.
Als Besonderheit ist es in JAVA so, dass irgendwelchen anderen Daten-Typen oder Werten ein Wahrheitswert zugeordnet wird. Das ist vor allem für Umsteiger aus C und Python verwunderlich – aber eben konsequent. Wahrheitswerte sind grundsätzlich nur **true** und **false** und auch nur diese können bei Vergleichen betrachtet oder ermittelt werden.

Dadurch sind Verwechselungen nicht möglich.

Beispiel1:

```
int a = 2;
int b = 3;
boolean istGleich = false;
```

falsch: istGleich = (a = b); → erzeugt Compiler-Fehler (zwei Zuweisungen!)

richtig: istGleich = (a == b)

Beispiel2:

```
boolean a = true;
boolean b = false;
int ergebnis = 0;
ergebnis = (a + b);
```

→ erzeugt Compiler-Fehler (unzulässiger Operator)

Auch das Umtypisieren von ***ergebnis*** auf **boolean** ändert daran nichts. Jetzt könnte allerdings zwischen ***a*** und ***b*** ein binärer logischer Operator gesetzt werden, z.B. **&&**.

Auswertungs-Reihenfolge von logischen Operatoren:

Klammern → Negation → UND-Verknüpfung → ODER-Verknüpfung

() → ! → & → | → && → || → |

höchster Rang … niedrigster Rang

1.3.7.1. weitere logische Operatoren

Das e**x**klusive **ODER** ist ebenfalls sehr gebräuchlich. Dabei darf nur eine der Bedingungen wahr sein, um ein wahres Ergebnis zu erzielen. Sind beide Aussagen wahr, dann bekommt man (eben im Gegensatz zum normalen ODER) ein FALSCH als Ergebnis.
Die exlusive ODER-Funktion wird vielfach als XOR und als mathematischen Zeichen ein ^ (Dach-Zeichen, Akzent-Zeichen, Caret-Zeichen, Zirkumflex) geschrieben.

XODER-Verknüpfung

A	B	A ^ B
true	true	**false**
true	false	**true**
false	true	**true**
false	false	**false**

fast vollständige Auswertungs-Reihenfolge von logischen Operatoren:

() → ! → & → ^ → | → && → || → |

höchster Rang … niedrigster Rang

Hinweise:

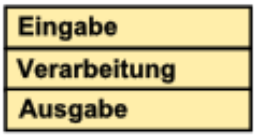
- die arithmetrischen und einege andere Operatoren werden weiter vorn besprochen (→ 1.3.4. Variablen); hier sind sie nur wegen der Vollständigkeit aufgeführt

1.3.9. Programm-Strukturen / Algorithmen-Strukturen

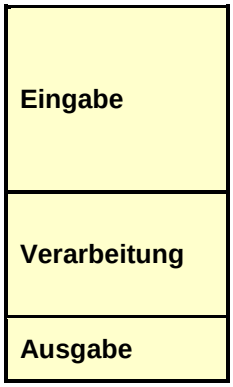
So einfach, wie unsere bisherigen, sind natürlich nicht alle Programme. In der Vielfalt von Lösungen der unterschiedlichsten Probleme gibt es nur wenige grundsätzliche Strukturen von Anweisungen. Mit diesen wenigen Strukturen lassen sich dann alle komplexen Algorithmen umsetzen.
Zu guten Programmen gehören auch sogenannte Kommentare (→). Diese sind für die Übersetzung des Quelltext nicht notwendig. Sie machen aber Quelltexte besser lesbar und vor allem zeigen Sie am Anfang eines Quelltextes auf, was das Programm leisten soll. Eine Analyse größerer Programme wäre viel zu aufwändig und auch zu unsicher. Die Kommentare stellen wir nebenbei und noch einmal am Ende dieses Abschnitts vor (→).

1.3.9.1. Sequenzen

Sequenzen – also die einfache Aneinanderreihung von Anweisungen – haben wir schon benutzt. Sie stellen die erste Struktur dar.
Algorithmisch gesehen haben wir einen Start-Punkt und ein reguläres Ende. Die Lösung führt vom Start-Punkt auf einem eindeutigen – uanabänderlichen – Weg zum End-Punkt.
Die Lösungs-Schritte folgen hintereinander (→ Schritt-Folge).
Für die Darstellung von Algorithmen oder Software-Strukturen werden in der Schul-Informatik Struktogramme benutzt. Sie sind Block-orientiert. In einer Sequenz folgt – von oben nach unten gelesen – ein Block dem anderen. Die logische EVA-Struktur wird z.B. durch drei Blöcke dargestellt. Jeder dieser Blöcke kann später durch kleinere / detailliertere Blöcke ersetzt werden.



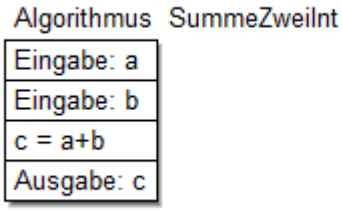
Das gesamte Programm / der gesamte Algorithmus ist insgesamt wieder ein Block. Als solcher könnte er später in anderen Programmen / Algorithmen wieder eingebaut werden.
Programme, die nur aus einfachen Aneinanderreihungen von Schritten (Blöcken) bestehen, werden häufig als erste Grund-Version (Prototyp) eines Programms erstellt. Man nennt sie auch Geradeaus-Programm.
In JAVA wird der Gesamt-Block durch die geschweiften Klammern umrissen. Die einzelnen Anweisungen / Blöcke sind die Semikolon-getrennten Befehle, die wir meist in einzelnen Zeilen untereinander notieren.
Betrachten wir ein sehr einfaches Programm als Beispiel. Es sollen zwei Zahlen eingegeben werden. Diese sollen dann addiert und schließlich ausgegeben werden.

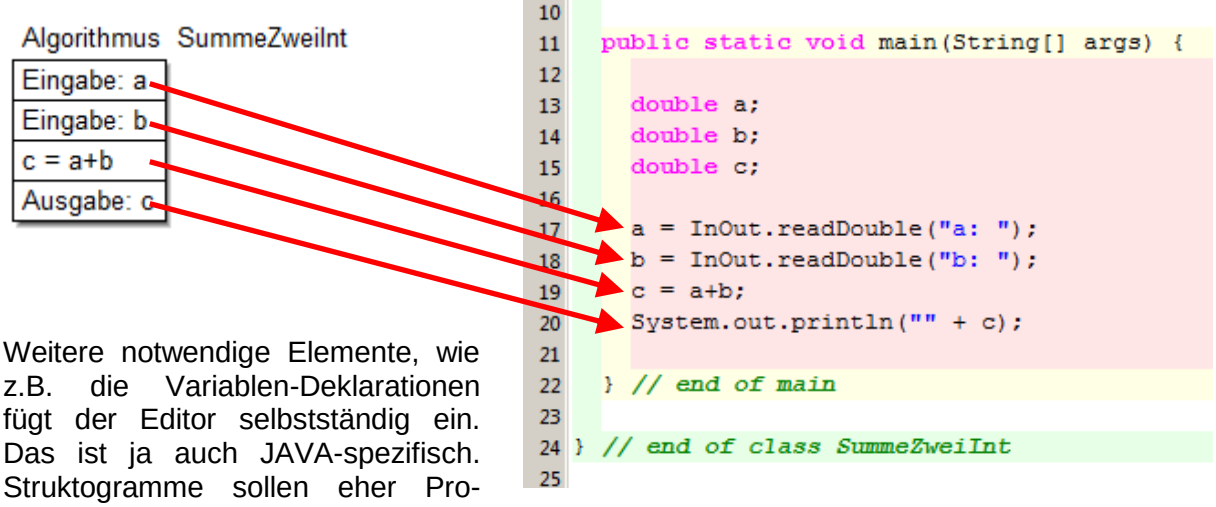



```
 1  import java.util.Scanner;
 2  class SummeZweiInt
 3  {
 4    public static void main(String[] args)
 5    {                                                    Blockbeginn
 6      Scanner scan = new Scanner(System.in);
 7      System.out.print("a = ");
 8      int a = scan.nextInt();
 9      System.out.print("b = ");
10      int b = scan.nextInt();
11      scan.close();
12
13      int c = 0;
14      c = a + b;
15
16      System.out.print("a + b =");
17      System.out.println(c);
18    }                                                    Blockende
19  }
```

Wir sehen, dass bei JAVA viel Schreibarbeit ringsherum ist. Das ist für Anfänger sicher eher lästig. Aber es muss eben sein. Manche Blöcke sind extrem klein, wie z.B. die Verarbeitung, während andere Blöcke auch schnell mächtig aufgebauscht sind, wie z.B. die Eingabe.
Die Verhältnisse können sich aber in anderen Programmen völlig anders darstellen.
Schauen wir uns aber das Struktogramm dieses einfachen Summierungs-Programm's in der erweiterten Form an. Dabei gliedern wir die einzelnen Haupt-Elemente bis auf Anweisungs-Ebene auf. Das Struktogramm erstellen wir hier mit dem Java-Editor, der dann hinterher eine Umwandlung in JAVA-Code zulässt.

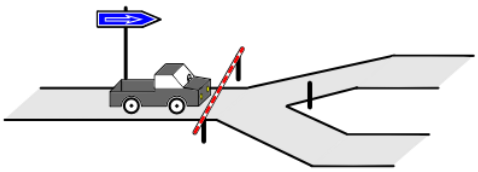
Die Handhabung wurde vorne erläutert. Im Zweifelsfall können Sie dort noch einmal nachlesen oder sich neu informieren (→ 1.1.2.1.3. der Java-Editor als Struktogramm-Editor).
Benutzt man die Schlüsselwörter "Eingabe:" und "Ausgabe:" für die Eingabe- und Ausgabe-Blöcke, dann wird der JAVA-Code auch passend erzeugt. Ansonsten erscheinen nur Markierungen, die dann mit Leben erfüllt werden müssen.
Der Java-Editor übersetzt aus dem Struktogramm den folgenden Code:
Dabei sind die Zuordnungen recht einfach zu erkennen:





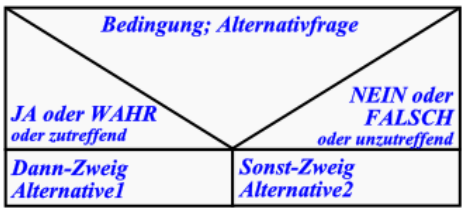
Weitere notwendige Elemente, wie z.B. die Variablen-Deklarationen fügt der Editor selbstständig ein. Das ist ja auch JAVA-spezifisch. Struktogramme sollen eher Programmiersprachen-übergreifend bzw. universell erstellt werden.

1.3.9.2. Verzweigungen



Wenn jedes Programm so glatt durchgehen würde, wie es bei den Sequenzen ist, dann wäre Programmierung wahrscheinlich langweilig und vielfach auch unsinnig.
Meistens müssen innerhalb des Algorithmus alternative Wege vorgesehen oder eingeschlagen werden.
Alternativen werden auch Verzweigungen oder Selektionen genannt.

Ihr Struktogramm-Symbol ist charakteristisch.
Praktisch haben wir zwei sequentielle Groß-Blöcke. Der obere Block ist der Entscheidungs-Teil. Hier wird eine oder mehrere Bedingung(en) geprüft und dann mit dem unteren Ausführungs-Teil fortgesetzt. Dieser ist immer zwei-geteilt. Ein Block steht für die Variante, wenn die Bedingung zutrifft und der andere, wenn sie nicht zutrifft.



Die Zuordnung für die beiden Seiten ist nicht vorgeschrieben. Die Entscheidungs-Grundlage wird oben über die Ausführungs-Blöcke rübergeschrieben. Steht dort nichts, wird klassischerweise davon ausgegangen, dass links die WAHR- bzw. JA-Variante steht. Der FALSCH- bzw. NEIN-Zweig entsprechend rechts. Besser ist es aber immer, die Informationen vollständig zu notieren.
In allen Verzweigungen wird bei Bedingung / Alternativfrage immer ein Ausdruck erwartet, der für JAVA true oder false ergibt. Alle anderen Konstrukte führen zu einem Compilerfehler ().
Typische Bedingungen könnten sein (die vorher notwendigen Variablen-Deklarationen stehen rechts! Die Methode **ungerade()** liefert ein boolean zurück.):

```
sortiert                    → true
sortiert == true            → true

a > b                       → false

(b >= a) && (b <= c)        → true

ungerade(3)                 → true

!ungerade(b) || sortiert    → true
```

```
boolean sortiert = true;

int a = 3;
int b = 4;
int c = 10;
```

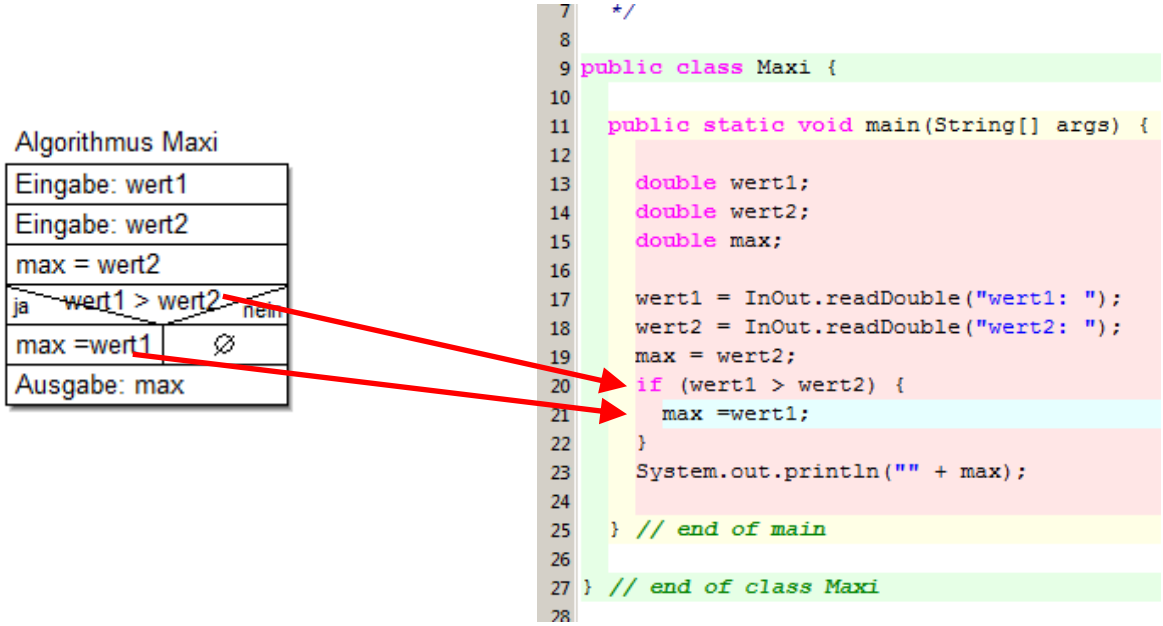
Man beachte die doppelten Gleichheits-Zeichen (**==**) für Vergleiche. Zur Wiederholung zum Umgang mit Wahrheitswerten steht Kapitel → 1.3.8. Arbeiten und Anzeigen mit Wahrheits-Werten zur Verfügung. Am Ende des Abschnitts Verzeigungen findet der Leser eine Zusammenstellung der gebräuchlichsten Vergleichs-Operatoren.
Viele Programmierer sprechen auch vom THEN-Zweig (DANN-Zweig) für die zutreffende Bedingung und entsprechend dem ELSE- oder SONST-Zweig für die anderen Fälle (nicht-zutreffende Bedingung).

einfache Verzweigungen

Manchmal möchte man in Programmen nur dann etwas tun, wenn eine Bedingung zutrifft. Man spricht hier von einfachen Verzweigungen. Sie enthalten im Vergleich mit den vollständigen Verzweigungen nur den Anweisungs-Block, der im Falle des Eintreffens der Bedingung (JA od. WAHR) ausgeführt wird.
In allen anderen Fällen wird nach dem Anweisungs-Blockende weiter gemacht. Für das Nichts-Ausführen in solchen einfachen Alternativen steht der gekreuzte Block.

if (*Bedingung*){ // Then-, Wahr- od. Erfüllt-Zweig
BlockAnweisung(en)
}

Der grüne Kommentar kann natürlich entfallen. Bei komplizierteren Bedingungen sollte man sich aber eine kurze Notiz machen, dass hilft später bei der Fehlersuche oder dem Code-Verständnis.
Als Beispiel wählen wir hier man die Bestimmung des Maximums von zwei eingegebenen Zahlen.
Als konkretes Struktogramm könnte das dann so aussehen:



Der automatisch erstellte Quellcode (vom Java-Editor) funktioniert, wie ein Test mittels Run-Button zeigt.
Allerdings befriedigen uns die Eingabe-Aufforderungen und die Ausgabe des Ergebnisses noch nicht.

```
wert1: 5
wert2: 2
5.0
```

Die Ausgabe als Gleitkommazahl ist nur dann so interessant, wenn es sich wirklich um solche handelt. Klassischerweise arbeiten wir beim Programmieren-Lernen eher mit Ganzzahlen.
Der etwas aufgemotzte Quellcode könnte dann beispielsweise so aussehen:

```
1 public class Maxi{
2     public static void main(String[] args){
3         int wert1 = 0;
4         int wert2 = 0;
5         int max = 0;
          wert1 = InOut.readInt("1. Zahl: ");
          wert2 = InOut.readInt("2. Zahl: ");
          max = wert2;
          if (wert1 > wert2) {
              max = wert1;
          }
          System.out.println("Die grössere Zahl ist " + max);
      }
  }
```

Die Ansicht in der Konsole ist dann auch gefälliger.
An den Feinschliff sollte man sich aber erst machen, wenn der Kern des Programms steht. Oft lassen sich einfache Verzweigung aus vollständigen erstellen. Dazu gleich mehr.

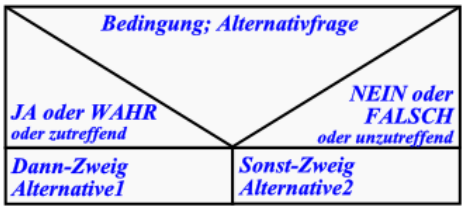
```
1. Zahl: 5
2. Zahl: 2
Der grössere Zahl ist 5
```

Aufgaben:

1. *Bauen Sie das Maximum-Programm so um, dass der kleinste Wert von vier eingegebenen Werten gefunden wird!*
2. *Sollten Sie bei Aufgabe 1. mehr als zwei Variablen verbraucht haben, dann erstellen Sie jetzt eine Variablen-sparsame Version mit nur zwei Variablen!*
3. *Könnte man auch mit nur einer Variable auskommen? Erklären Sie Ihren Standpunkt!*
4.

vollständige Verzweigungen

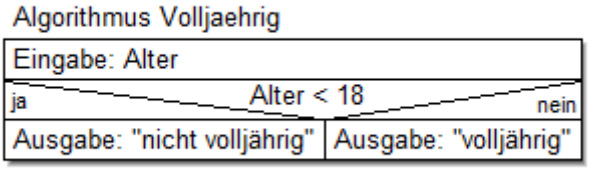
Bei etwas mehrzeiligen Alternativ-Blöcken wird man um die vollständigen Verzweigungen nicht herumkommen. D.h. wird haben Anweisungen in beiden Teilen der Verzweigung. Ob diese gleichartig sind oder mehr oder weniger Anweisungen enthalten spielt keine Rolle. Der Ausführungs-Block jedes Alternativ.Zweiges kann beliebig viele Anweisungen enthalten. Im Struktogramm sind das dann Stapel von Blöcken in einem Zweig.



Der Syntax wird – im Vergleich zur einfachen Verzweigung – um den **else-Block** ergänzt.

```
if (Bedingung){    // Then-, Wahr- od. Erfüllt-Zweig
        Block1
} else{            // Else-, Sonst-, Falsch- bzw. Nichterfüllt-Zweig
        Block2
}
```

Schauen wir uns wieder ein Beispiel an. Das Programm soll nach der Eingabe eines Alters entscheiden, ob die Person "volljährig" oder "nicht volljährig" ist. Natürlich würde das auch wieder, wie wir es bei den einfachen Verzweigungen gezeigt haben, mit einer kurzen Struktur gehen. Wir wählen aber hier einmal den vollständigen Fall. Das Struktogramm stellt uns nicht wirklich vor Herausforderungen:
Um später Text-Ausgaben zu erhalten setze ich die Texte im Struktogramm gleich auch in Anführungszeichen.
Trotzdem bleibt bei der automatischen Übersetzung in JAVA-Code mittels Java-Editor bleibt eine Menge Nacharbeit, so dass ich uns die generierte version hier mal erspare und so tue, als würde ich das Struktogramm per Hand umsetzen.

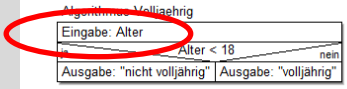


Wir benötigen zuerst das einfache (funktions-lose) Rahmenprogramm, was wir dann Schritt für Schritt ergänzen.

```
1  class Volljaehrig{
2    public static void main(String[] args){
3    }
4  }
```

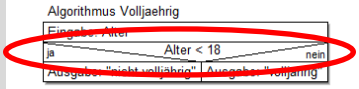
Für die Umsetzung des ersten Block's brauchen wir eine Variable alter. Diese soll dann durch eine Eingabe belegt werden.

```
1  class Volljaehrig{
2    public static void main(String[] args){
3      int alter = 0;
4      alter = InOut.readInt("Alter [in Jahre]: ");
5    }
6  }
```



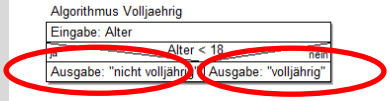
Diesen Code können wir auch schon mal zwischendurch testen. Zumindestens die Text-Ausgabe für die Eingabezeile wird erstellt.
Der nächste Block ist die Verzweigung.

```
1  class Volljaehrig{
2    public static void main(String[] args){
3      int alter = 0;
4      alter = InOut.readInt("Alter [in Jahre]: ");
5      if (alter < 18){
6      } else {
7      }
8    }
9  }
```



Nun fehlen nur noch die Anweisungs-Blöcke mit den Ausgaben.

```
1  class Volljaehrig{
2    public static void main(String[] args){
3      int alter = 0;
4      alter = InOut.readInt("Alter [in Jahre]: ");
5      if (alter < 18){
6        System.out.println("nicht volljährig");
7      } else {
8        System.out.println("volljährig");
9      }
10   }
11 }
```



Praktisch entsprechen die Blöcke immer kleinen einfachen JAVA-Elementen. Sie lassen sich wie Bausteine aufeinandersetzen. Die JAVA-Grundstrukturen muss man allerdings lernen, damit man nicht ewig irgendwo nachschlagen muss.
Durch geschicktes Hantieren mit Attributen oder Variablen lassen sich viele vollständigen Verzeigungen in einfache überführen. Das spart meist Quellcode und ist effizienter.

```
Alter [in Jahre]: 16
nicht volljährig
```

Allerdings muss der Programmier etwas mehr über die Struktur nachdenken.
Die einfachste Variante ist es, einen Wert schon vor der Verzweigung festzulegen und diesen nur dann zu ändern, wenn eine bestimmte Bedingung eingetroffen ist.

Aufgaben:

1. ***Schreiben Sie das Programm so um, dass es mit einer einfachen Verzweigung auskommt!***
2. ***Prüfen Sie den folgenden Code! Realisiert dieser die obige Aufgabenstellung genau so, wie der schrittweise entwickelte Code? Korrigieren Sie Fehler oder zeigen Sie, dass das Programm so fehlerfrei laufen sollte!***

```
1  class Volljaehrig{
2    public static void mein(Strings[] args){
3      int alter = 0;
4      String status = 0;
5      Alter = OutIn.readDouble("Alter [in Jahre]: ");
6      if (alter < 18){
7        status = "volljährig";
8      ) else (
9        Status = "nicht volljährig";
10     }
11     System.out.writeln(status);
12   }
```

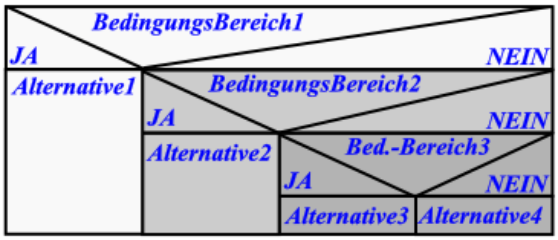
3. ***Erstellen Sie ein Programm, dass eine einzugebende Ganzzahl auf die Eigenschaften "gerade" und "ungerade" testet und das Ergebnis als vollständigen Antwortsatz anzeigt!***
4.

Mehrfach-Verzweigungen

Mit Mehrfach-Verzweigungen meint man ineinander geschachtete einfache Verzweigungen mit ev. einer abschließenden vollständigen Verzweigung (innerste Verzweigung). Man kann sich das so vorstellen, dass man sich bei einer Vielzahl von zu bearbeitenen Fällen immer schrittweise einen herausgreift und diesen bearbeitet. Traf der Fall nicht ein, dann wird der nächste versucht usw. usf.

Hat man alle Varianten durch, dann kann man ev. den else-Zweig der letzten Verzweigung mit einem entsprechenden Hinweis versehen oder die Struktur verlassen.

Das Struktogramm-Bild ist keine eigenständige Struktur, sondes sind ern wirklich nur ineinander verbaute Verzeigungen. Zur besseren Erkennung sind sie unterschiedlich grau unterlegt.



Statt mehreren Verzweigungen wird im Syntax scheinbar nur eine erweiterte Verzweigung ersichtlich:

```
if (Bedingung1){
        Block1
} else if (Bedingung2){
        Block2
} else{
        Block3
}
```

Schaut man aber genau hin, dann erkennt man, dass die Verzweigungen doch nur hintereinander gehängt wurden. Trotzdem ist die Mehrfach-Verzweigung eine häufig benutzte Feinstruktur.
Als Beispiel setzen wir unser Alters-Bewertungs-Programm nun fort und arbeiten weitere Status-Merkmale hinzu. Das sollen sein "strafmündig", "geschäftsfähig", "Kind", "Jugendlicher", "Erwachsener", "Kleinkind"

Status	Alter [a]	Bemerkungen

Aufgaben:

1.
2.
3.

Vergleichs-Operatoren

Operatoren (Kontext)	Beschreibung		Hinweise / Beispiele
infix; binär			
==	ist gleich (Gleichheit)		= steht für eine Zuweisung
!=	ist ungleich (Ungleichheit)		! ist einfache Negation
>	ist größer		
<	ist kleiner		
>=	ist größer oder gleich		
<=	ist kleiner oder gleich		
&&	UND-Verknüpfung AND		Compiler macht einen Abbruch der Auswertung, wenn z.B. schon der erste Teil-Ausdruck **falsch** ist (→ Kurzschluss,. der Rest ist dann ohne Bedeutung für das Gesamtergebnis!) (→ Short Circuit Evaluation)
||	ODER-Verknüpfung OR		Compiler macht einen Abbruch der Auswertung, wenn z.B. schon der erste Teil-Ausdruck **wahr** ist (→ Kurzschluss,. der Rest ist dann ohne Bedeutung für das Gesamtergebnis!) (→ Short Circuit Evaluation)
^	Exklusive ODER-Verknüpfung XOR		ohne Short Circuit Evaluation
&	UND-Verknüpfung mit erzwundenem vollständigen Auswerten des Ausdrucks		ohne Short Circuit Evaluation
|	ODER-Verknüpfung mit erzwundenem vollständigen Auswerten des Ausdrucks		ohne Short Circuit Evaluation
präfix, unär			
!	entgegengesetzt (Negation)		!false == true

Short Circuit Evaluation .. Kurzschluss-Auswertung / Schnell-Auswertung / Auswertung auf dem kurzem Weg

Auswertungs-Reihenfolge von logischen Operatoren:

Klammern → Negation → UND-Verknüpfung → ODER-Verknüpfung

() → **!** → **<** | **<=** | **>=** | **>** → **==** | **!=** → **&** → **|** → **&&** → **||** → **|**

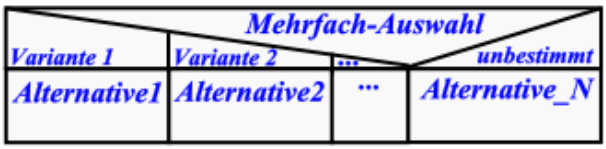
höchster Rang … niedrigster Rang

Kommt z.B. in einer Verzweigung als Bedingung eine UND-Verknüpfung vor, dann müssen ja beide Operanden wahr sein. Nun könnte ja schon der erste Operant falsch sein. In diesem Fall kann die UND-Verknüpfung nicht mehr insgesamt wahr werden. Das System kann sich die weitere Auswertug sparen. Das Ergebnis ist und bleibt falsch. In der Praxis hört das System auch wirklich auf, weiter zu prüfen. Es kommt zu einer Kurzschluss-Entscheidung (Short Circuit Evaluation). Das spart kostbare Zeit.
Bei einer ODER-Verknüpfung kommt es zu einer Kurzschluss-Entscheidung, wenn der erste Operant wahr ist. Das reicht ja aus, um den gesamten Ausdruck wahr werden zu lassen.

Für die Aneinander-Reihung von Operationen ist es immer gut, zu prüfen, welche Bedingungen im Falle einer UND-Verknüpfung sehr wahrscheinlich eher falsch sein werden. Diese sollten dann weiter nach vorne in die bedingung gestellt werden, damit sie schnellstmöglich eine Short Circuit Evaluation auslösen können. Bei ODER-Verknüpfungen sollte man die eher wahren Bedingungen zuerst bearbeiten lassen.
Bestimmte Vergleiche müssen in bestimmten Reihenfolgen erfolgen, wobei genau der "Short Circuit Evaluation"s-Effekt ausgenutzt wird.
Will man z.B. zwei Dinge miteinander vergleichen (z.B. auf gleichen Namen), dann muss vor der Namens-Prüfung ein Abchecken auf Vorhandensein des zweiten / des Vergleichs-Objekte erfolgen. Erst im zweiten Teilschritt kann ein Vergleich der Namen gemacht werden.
Würde man z.B. die Namen zuerst prüfen wollen, dann würde bei Aufruf eines nicht-existierenden Objektes eine Unterbrechung (Exception, Fehlermeldung) erfolgen. Ein nicht vorhandenes Objekt hat eben keinen Namen.

einfache Mehrfach-Auswahl

Bei der Mehrfach-Verzweigung ließen sich die Bedingungen sehr speziell gestalten. Man hatte dort alle Möglichkeiten der Vergleiche und logischen Operationen. In vielen Fällen ist die Auswahl gar nicht so kompliziert, sondern man hat abzählbare Varianten z.B. mit eindeutigen Werten. Achtung, hier sind wirklich Werte gemeint und keine resultierenden Wahrheits-Werte.
Mit solchen Werten haben wir es z.B. bei eindeutig durchnummerierten oder sortierten Alternativen zu tun. Beispiele dafür sind die Wochentage von 0 bis 6 durchnummeriert, oder die Farben (Varianten) "rot", "gelb", "blau", "grün", … Hier braucht man auch keine Bereiche zum Prüfen – wie z.B.: ***Farbe zwischen "rot" und "gelb"*** , sondern man meint sofort (die Variante) "orange".
Für eine Auswahl aus definierten Werten bietet sich die Mehrfach-Auswahl oder auch **case**-Anweisung an.
In JAVA ist das einleitende Schlüsselwörtchen **switch**.
Innerhalb des switch-Blockes können beliebig viele case-Blöcke stehen, nur die Varianten-Werte (Auswahl-Werte) dürfen sich nicht wiederholen.



Am Ende kann man einen **default**-Block von Anweisungen definieren, der in dem Fall ausgeführt wird, wenn keiner der **case**-Varianten zugetroffen hat.
Die **case**-Blöcken werden meist mit **break** abgeschlossen, um nach dem Zutreffen der **case**-Variante nicht noch unnötigerweise die anderen Varianten-Möglichkeiten zu durchsuchen. In dem Fall wird dann der **default**-Block auch ausgeführt.
Bei **break**-Anweisungen kommt es immer zum Verlassen der aktuellen Anweisungs-Struktur. Das wird uns bei Schleifen auch noch begegnen.

```
switch (Bezeichner){
        case Variante1:
                        //Block
                        break;
        case Variante2:
                        //Block
                        break;
…
        case VarianteN:
                        //Block
                        break;
default:
                //Block
}
```

Oben wurde suggeriert, das man z.B. Farben anhand der Texte für die switch-case-Struktur nutzen könnte. Das geht nicht, da Objekt-Datentypen nicht einfach inhaltlich verglichen werden, sondern nur ihre sachliche Gleichheit als Objekt (also deren Zeiger-Adresse).
Man kann sich aber helfen, indem Konstanten definiert werden. Diese können einen einfachen Wert haben und lassen sich im Quelltext quasi wie Texte ansprechen. Dazu später mehr (→).

Aufgaben:

1.
2.
3.

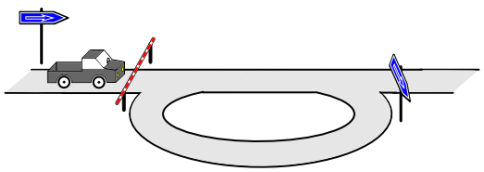
1.3.9.3. Schleifen

Man nennt diese Programmstrukturen auch Loop's, Schlaufen oder Iterationen. In jedem Fall geht es darum, sich wiederholenden Code nicht mehrfach in das Programm hineinzuschreiben. Bei Fehler-Korrekturen müsste man dann auch x-mal den Quellcode anpassen.
Schleifen sollte man schon dann benutzen, wenn eine zweite Wiederholung eines Quell-Code-Stückchens notwendig ist.

Schleifen bestehen immer aus einem Kontroll-Teil (Kopf oder Fuß) und einem Schleifen-Körper. Der Schleifen-Körper enthält die zu wiederholenden Anweisungen.
Aus Effektivitätsgründen sollte man darauf achten, das wirklich nur der zuwiederholende Code in der Schleife steht. Andere Anweisungen würden ev. sehr häufig wiederholt werden, was nur unnötig Ressourcen bindet.
Zu Schleifen passen besonders gute Array's. Viele Daten lassen sich so durchforsten.
Jede Schleifen-Art kann durch die anderen ersetzt werden, meist gibt das aber deutlich mehr Programmier-Aufwand und / oder der Code wird unübersichtlich oder tricksig.

Ist die Anzahl der Schleifen-Durchläufe / Wiederholungen bekannt und ist dieser innerhalb der Quellcode auch unveränderlich, dann benutzt man sogenannt Zähl-Schleifen zur Problem-Lösung (→ 1.3.9.3.2. Zähl-Schleifen).
Alle anderen Schleifen hängen von einer Bedingungen ab. Solange die Bedingung gültig (wahr) ist. Natürlich darf die Bedingung auch ein größerer logischer Asudruck, mit vielen Operatoren, sein.
Alle Schleifen können durch das Schlüssel-Wörtchen **break** vorzeitig verlassen werden. Es wird dann mit dem ersten Anweisung hinter der Schleife / dem Schleifen-Block fortgesetzt.
Manche Programmierer sehen break als so eine Art Zeichen von toller Programmierung an. Aus der Sicht der struktorierten Programmierung ist die break-Anweisung eigentlich ein No-Go, wie früher zuviele GOTO's in BASIC-Programmen.
Gegen einen kontrollierten und ev. auch kommentierten / verständlichen Einsatz sollte man sich aber nicht zu sehr auflehnen.

1.3.9.3.1. bedingte Schleifen



Schleifen enthalten praktisch immer bestimmte Bedingungen, die ihr Verlassen bzw. Nicht-Hineintreten ermöglich.
Von bedingten Schleifen sprechen wir, wenn ein logischer Ausdruck zur Bewertung weiterer Schleifen-Durchläufe bzw. das verlassen der Schleife benutzt werden.
Alternativ kann man die Anzahl der Schleifen-Durchläufe im Vorfeld festlegen (→ 1.3.9.3.2. Zähl-Schleifen).
Je nach Position der Prüf-Bedingung am Anfang oder am Ende der Schleife (aus der Sicht des Programm-Codes) sprechen wir von Kopf- oder Fuß-gesteuerten Schleifen.
Will man mittels bedingter Schleifen ein Array durchlaufen, dann benötigt man eine selbstverwaltete Index-Variable, die den Zugriff auf das gewünschte Element realisiert.

Kopf-gesteuerte Schleife

Die Kopf-gesteuete Schleife ist durch einen Bedingungs-Konstrukt am Anfang der Schleife charakterisiert. Vor jedem Durchlauf des Schleifen-Körpers wird geprüft.
Da die Bedingung auch schon vor dem ersten Schleifen-Durchlauf unwahr (falsch) sein kann, muss auch damit gerechnet werden, dass eine Kopf-gesteuerte Schleife nicht ein einziges Mal durchlaufen wird.
In Kopf-gesteuerten Schleifen sollte also nur solchen Code stehen, der ev. auch garnicht für andere Zwecke gebraucht wird.

while (*Bedingung*){ // SOLANGE Bedingung (gilt) …
Block //Schleifenkörper bzw. Schleifenrumpf
}

Als sprachlicher Ausdruck könnte man:

WIEDERHOLE (wenn / bei) Bedingung (wahr ist) (die) Schleifenanweisungen
WENN Bedingung (wahr ist) DANN_WIEDERHOLE Schleifenanweisungen
SOLANGE Bedingung tue (die) Schleifenanweisungen

formulieren. Vielfach wird diese Ausdrucksweise auch in den Struktogrammen benutzt.
Als Beispiel nehmen wir ein Programm, das solange würfeln soll, bis die Zahl 17 erreicht oder gerade überschritten wird.
Zufallszahlen können wir uns über die Bibliothek **Math** besorgen. Diese stellt die Methode random() zur Verfügung. Als Ergebnis bekommen wir eine Zahl im Bereich 0,0 ≥ x < 1,0 zurück. Man beachte die Nicht-Erreichbarkeit von 1,0.
Ein Wurf des üblichen **6**er-Würfel können wir dann mit:

```
…
wurf = (int) (Math.random() * 6) +1;
…
```

Die Addition der 1 bewirkt, dass aus dem Produkt (liegt im Bereich von 0,0 bis <6,0) eine Punkte-Zahl zwischen 1 und 6 herauskommt. Das Casten (→ 1.3.7.1. Typ-Umwandlungen (Type Casting und Parsing)) der Gleitkommazahl auf **int** bewirkt ja das Abschneiden der Nachkommastellen.

```
…
int grenze = 17;
aktuellePunkte = 0;
while (aktuellePunkte < grenze){
    wurf = (int)(Math.random() * 6) + 1;
    aktuellePunkte = aktuellePunkte + wurf;
    System.out.print("aktueller Punktestand: " + aktuellePunkte);
    System.out.println("    nach einer gewürfelten: " + wurf);
}
System.out.println("Schleife beendet!");
…
```

Betrachten wir auch noch ein Schleifen-Beispiel mit einem einfachen Array (→ 1.3.10.1. primitive Array's). Ein sehr großes Array sei nur bis zu einem bestimmten Wert belegt und soll nach einem ganz bestimmten Inhalt abgesucht werden. Als Ergebnis erwarten wir die Ausgabe der Position, an der sich das gesuchte Element innerhalb des Array's befindet.

```
…
int feldGroesse = 100;
int[] xWerte = new int[feldGroesse];
int aktGroesse = 0;

xWerte = {12, 4, 5, 0, 13, 5, 9};
aktGroesse = 7;

int suchWert = 5;

int fundStelle = -1;
aktPosition =0;
while (aktPosition <= aktGroesse){
    if (xWerte[aktPosition] = suchWert){
        fundStelle=aktPosition;
    }
    aktPosition++;

}
System.out.println(suchWert + " gefunden bei Position: " + fundStelle);
…
```

Aufgabe:

1. ***Wie müsste man das Such-Programm abändern, damit die erste Fundstelle nach der Schleife angezeigt wird?***
2. ***Warum hat man fundStelle vor der Schleife auf -1 gesetzt? Erweitern Sie die Ausgabe so, dass die resultierende Information passend angezeigt wird!***
3. ***Erstellen Sie aus dem Programm von 2. ein Struktogramm!***
4. ***Gesucht wird das Struktogramm (Haushaltsbuch) für die Subtraktion aller Werte des Array "ausgaben" vom vorgegebenen Wert 1500 (nettoLohn)! Das Array soll die Tages-Ausgaben eines Monats aufnehmen können.***
5.

Aufgaben:

1. *Erstellen Sie ein vollständiges Programm zum obigen Würfel-Problem!*
2. *Überlegen Sie sich, wie man das Programm so umgestalten könnte, dass man mit weniger Variablen und Quell-Codezeilen auskommt! (Es bleibt bei der Orientierung von einer Anweisung pro Zeile.)*
   Testen Sie Ihren Vorschlag! (Extra Datei / Klasse erstellen!)
3. *Ändern Sie das Programm so ab, dass mit zwei 20er-Würfeln gearbeitet wird und die Grenze bei 111 liegt!*
4. *Ergänzen Sie nun noch eine Möglichkeit die Anzahl der notwendigen Würfe (das Würfeln beider Würfel gilt als ein Wurf), um die Grenze zu erreichen oder gerade zu überschreiten!*

für die gehobene Anspruchsebene:

5. *Erstellen Sie ein Spiel, in dem der Spieler mittels 2 normaler 6er-Würfel die Zahl 37 erreichen muss! Der Spieler kann bei jedem Wurf entscheiden, ob er mit einem oder zwei Würfeln spielen möchte (Eingabe braucht nicht überprüft werden!). Wird die Grenze überschritten, hat er verloren.*

Fuß-gesteuerte Schleife

Bei Fuss-gesteuerten Schleifen wird erst am "Ende" der Schleife der Abbruch getestet. Dieser Schleifen-Typ ist also dadurch gekennzeichnet, dass der Schleifen-Körper mindestens einmal durchlaufen wird.

Schleifen-Schritt(e)
SOLANGE Bedingung

do {
Block //Schleifenkörper
} while (*Bedingung*) // SOLANGE Bedingung (noch gilt) …

TUE Schleifenanweisungen SOLANGE Bedingung ((noch) gilt)

Sachlich könnten wir unser obiges Würfel-Problem auch mit dieser Schleifen-Art lösen, denn schließlich muss mindestens 1x gewürfelt werden, sonst macht es keinen Sinn.

```
…
int grenze = 17;
aktuellePunkte = 0;
do {
    wurf = (int)(Math.random() * 6) + 1;
    aktuellePunkte = aktuellePunkte + wurf;
    System.out.print("aktueller Punktestand: " + aktuellePunkte);
    System.out.println("    nach einer gewürfelten: " + wurf);
} while (aktuellePunkte >= grenze)
System.out.println("Schleife beendet!");
…
```

Aufgabe:

Wo liegt der Unterschied (außer in der Lage der Schleifen-Anweisungen) zum Programm, dass die Kopf-gesteuerte Struktur benutzt?

Eine Routine-Anwendungen von Schleifen ist die Absicheung einer Eingabe, bis diese innerhalb der vorgegeben Grenzen erfolgt:

```
…
int eingabe;
int untereGrenze =1;
int obereGrenze = 100;
Scanner scan = new Scanner(System.in);
boolean ok = false;
do {    //
    System.out.print("Eingabe [" + untereGrenze + " .. "↵
                    + obereGrenze + "]  ?: ");
    eingabe = scan.nextInt();
    if (eingabe >= untereGrenze && eingabe <=obereGrenze){
    ok = true;
    } else {
    System.out.println("Eingabe nicht im vorgegebenem Bereich!");
    System.out.println("  --> Bitte wiederholen!"),
    }
} while (!ok);
scan.close();

…
```

Aufgaben:

1.

2.

3.

1.3.9.3.2. Zähl-Schleifen

FÜR Zähler BIS Ende TUE
Schleifen-Schritt(e)

Charakteristisch für Zählschleifen ist, dass i.A. genau bekannt ist, wieoft die Schleifen durchlaufen werden soll. Men kannauch diese Schleifen mit break abbrechen und verlassen. Das gilt aber nicht als besonders proffessionell.
Zum Verfolgen der Durchläufe benötigt jede Zählschleife ein interne Schleifen-Variable. Sie wird bei jedem Durchlauf verändert. Dabei ist Hoch- oder Runterzählen die klassische Lösung.
Als Formulierungen haben sich für algorithmische Beschreibungen z.B. die folgenden Satz-Strukturen bewährt:

VON … BIS … WIEDERHOLE ScheifenAnweisungen
VON … BIS … TUE ScheifenAnweisungen
FÜR Zähler = von BIS bis MACHE ScheifenAnweisungen
HOCHZÄHLEND VON von BIS bis WIEDERHOLE ScheifenAnweisungen

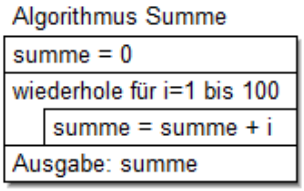
Das Manipulieren der Schleifen-Variable / Zähl-Variable innerhalb des Schleifen-Körpers (Schleifen-Block's) ist grundsätzlich zu unterlassen. Soll sich der Wert dar Zähl-Variable und damit die Anzahl der Schleifen-Durchläufe variabel erhöhen oder verringern können, dann sollte man auf eine bedingte Schleife ausweichen (→ 1.3.9.3.1. bedingte Schleifen).
Natürlich kann der Schleifen-Zähler für Ausgaben und Berechnungen benutzt werden.

Gerade bei problematischen Schleifen sollte man davon für Test-Zwecke auch regen Gebrauch machen.

hochzählende Schleife

for (int *Zählvariable* = *Anfangswert*; *LaufBedingung*; *Zählvariable*++){
Block //Schleifenkörper
}

Die LaufBedingung oder auch SchleifenBedingung oder auch WeiterlaufBedingung ist ein klassischer logischer Ausdruck. Solange die Schleife durchlaufen werden soll, muss er **true** sein. Ist der Ausdruck **false** wird die Schleife nicht (mehr) ausgeführt.
Als praktische Beispiel wählen wir hier mal die Bildung einer Summe von einer Reihe von Zahlen. Wir tuen hier natürlich auch so, als wüden wir die clevere Lösung vom jungen Adam RIES nicht kennen. Dann würden wir ja auch ohne Schleife auskommen.
Nebenstehend ein mögliches Struktogramm. Der Java-Editor kann es auch sehr gut in ein Programm umwandeln. Davon hier der entscheidende – schon leicht geänderte – Code-Abschnitt:




```
…
int summe = 0;
for (int i = 1; i <= 100; i++) {
    summe += i;  // summe = summe + i
}
System.out.println(summe);
…
```

Aufgaben:

1. *Erstellen Sie vollständiges Programm, dass die Summe der Zahlen von 123 bis 321 berechnet und anzeigt!*
2. *Erstellen Sie ein weiteres Programm, mit dem die Summe und das Produkt von 2 bis 26 berechnet!*
3. *Gesucht ist das Produkt der ungeraden Zahlen im Bereich von 1 bis 100 als Programm!*

für die gehobene Anspruchsebene:

4. *Erstellen Sie ein vollständiges Programm, dass die Summe der Produkte aus der Zahl und ihrer Schleifen-Durchlaufnummer berechnet und anzeigt! Die kleinste und grösste Zahl soll vor der Schleife eingegebn werden! (exakte Eingaben werden vorausgesetzt!)*
5. *Machen Sie das Programm von 4. so sicher, dass nummerische Fehleingaben nicht mehr möglich sind! (der Gültigkeitsbereich der vorgegebenen Zahlen soll zwischen 0 und 1'000 liegen.)*

runterzählende Schleife

Countdown-Schleifen sind praktisch genauso Leistungs-fähig wie die hochzählenden. Manchmal ist es einfach praktischer z.B. gegen 0 runterzuzählen und Werte zu berechnen. Ein interessanter Effekt ist, dass oft die Ergebniswerte schon nach wenigen Schleifen-Durchläufen ziemlich genau feststehen und die nachfolgenden Änderungen nur noch die kleinen Ziffern-Stellen ändern. Wenn z.B. nur eine geringe Genauigkeit gefordert ist, dann kann man u.U. schon frühzeit die Schleife abbrechen.

for (int *Zählvariable* = *Anfangswert*; *LaufBedingung*; *Zählvariable* - -){
Block //Schleifenkörper
}

Wir zeigen aber auch hier das klassische Beispiel de Summen-Bildung. Die Änderungen am Quell-Code sind minimal. Der Scheifen-Körper ist von den Veränderungen im Schleifen-Kopf gar nicht betroffen.

```
…
int summe = 0;
for (int i = 100; i > 0; i--) {
    summe += i;  // summe = summe + i
}
System.out.println(summe);
…
```

Aufgaben:

1. *Programmieren Sie die Produkt-Bildung für die Zahlen von 75 runterzählend bis einschließlich 25!*
2. *Erstellen Sie ein Programm, das einen Countdown von 60 runter anzeigt! Die Anzeige soll dabei ungefähr im Sekunden-Takt erfolgen. Als Zeitgeber ist eine weitere Schleife zu verwenden!*
3. *Erstellen Sie Struktogramm für die Teiler-Summe einer (natürlichen) Zahl!*
4. *Realisieren Sie das Struktogramm von Aufgabe 3 und testen Sie Ihre Software ausführlich!*

Schleifen mit double-Laufvariablen??? – Geht das?

In JAVA sind auch Schleifen mit Lauf-Variablen des Types double (bzw. eben andere Fließkommazahlen) zugelassen. Dabei muss man beachten, dass es durch Rundungs-Fehler passieren kann, das ev. ein Durchlauf weniger oder zuviel als vorausberechnet gemacht wird. Entweder muss man die Laufbedingung etwas verändern oder man muss sich mit anderen "sicheren" Schleifen oder Zählungen behelfen. Z.B. kann ja auch immer eine int-Lauf-Variable als Faktor verwendet werden, um die aktuelle (Lauf-)double-Variable für den Schleifen-internen Gebrauch zu berechnen.

```
...

...
```

Aufgaben:

1.
2.
3.

Allgemeines zu Schleifen

In den Aufgaben waren schon Beispiele dabei, bei denen Schleifen ineinander geschachtelt werden mussten. Die inneren Schleifen (in der Abb. **grün**) müssen für sich immer abgeschlossen sein, bevor in der äußeren Schleife (**blau**) der nächste Durchlauf gestartet werden kann.
Für jede zählende Schleife muss unbedingt eine neue Schleifen-Variable genutzt werden. Da sollte man sich so eine Art eigene Hierrarchie aufmachen.
Häufig findet man die Reihe: **i, j, k, l , m und n** oder auch **r, s, t, u, v und w**.
Noch besser ist natürlich die Verwendung von sprechenden Variablen-Namen. In Schleifen machen das die meisten Programmierer nicht, weil die Variablen meist häufig verwendet werden und außer dem nur lokal in einer einzelnen Schleife.
Bei bedingten Schleifen hat man den Vorteil, dass dort die Abbruch-Bedingungen in allen oder mehreren Schachtelungs-Ebenen gleich sein kann. Hat man z.B. eine Eingabe ausgiebig getestet, dann könnte das mit einem **eingabeOK (= true)** an alle Schleifen weitergegeben werden.
Alle Schleifen können durch das Schlüssel-Wörtchen **break** vorzeitig verlassen werden. Es wird dann mit dem ersten Anweisung hinter der Schleife / dem Schleifen-Block fortgesetzt.
Jede Schleifen-Art kann durch die anderen ersetzt werden, meist gibt das aber deutlich mehr Programmier-Aufwand und / oder der Code wird unübersichtlich oder tricksig.

1.3.9.4. Kommentare und javadoc

Kommentare sind Beschreibungs-Texte in Quelltexten. Sie werden beim Übersetzen eines Quelltextes überlesen. Viele Programmierer halten deshalb nichts von Kommentaren. Angeblich halten diese sie nur auf und verzögern die Fertigstellung einer Software.
Es gibt Berichte von Firmen, bei denen entfernte Abteilungen sehr ähnlich Programme geschrieben haben, weil die erste Abteilung ihr Produkt nicht für die anderen publiziert und auch keine Kommentare angegeben wurden. Das hat dann locker mal rund 400'000 Euro Kosten für die sinnfreie Doppel-Entwicklung verursacht.
Zwar sollte heute ein Quelltext so erstellt werden, dass er selbsterklärend ist, aber niemand wird sich bei der Suche nach einem schon vorhandenen Quelltext diesen vollständig durchlesen wollen. Deshalb gilt es als Pflicht am Anfang eines Quelltextes zumindestens Angaben zum Zweck, Leistungs-Umfang und den Leistungs-Grenzen zu machen. Daneben sollte sich der Autor mit einem letzten Bearbeitungs-Datum oder einer Versions-Nummer verewigen. Das erleichtert die Suche nach eine Ansprech-Person.
Bei der heute üblichen Lizenz-Vielfalt sollte man auch dahingehend Hinweise aufnehmen. Eine unberechtigte Kopie oder fehlende Lizenz können Firmen in den Ruin treiben.
Im Quelltext selbst sollte man sparsam mit Kommentaren umgehen. Solange sprechende Variablen und Algorithmen-Strukturen benutzt werden ist das auch kein Problem. Kritische – besonders knifflige Stellen sollten aber ausführlich kommentiert werden. Nur so kann man ein Debakel verhindern, wie es einer Versicherung gegangen ist. Dort hatten drei Programmierer einen sehr effektiven und komplexen Algorithmus erstellt. Wie der Teufel es will, verließen alle drei zur gleichen Zeit die Firma. Keinem anderen Programmierer der Versicherung gelang es, den Algorithmus zu rekonstruieren. Die einzige Möglichkeit wäre eine teure Neuprogrammierung gewesen. (Geholfen hat hier dann eine Firma, die autistische Informatiker verleiht. Einer ihrer Leiharbeiter konnte den Knoten lösen.)

einfache (Zeilen-)Kommentare:

Die einfachte Form der Kommentierung ist es, gleich etwas hinter die Befehle zu schreiben. Damit das Übersetzer-Programm diese Notizen nicht beachtet, werden sie mit zwei Schrägstrichen (Slash's, ***//***) eingeleitet. Der Rest der Zeile wird nicht beachtet. D.h. auch, dass man hier z.B. fehlerhaften Code notieren könnte.
In vielen IDE's werden Kommentare durch ein spezielles Layout (Syntax-Highlightning) hervorgehoben.

Kommentar-Blöcke

Mit Kommentar-Blöcken beschreibt man am Anfang eines Quelltextes das folgende Programm. Wie schon gesagt, sollte das zumindestens die Programm-Leistung und ev. auch die Grenzen sein.
Im Quell-Text selbst kann man solche Kommentar-Blöcke z.B. zum zeitweisen Ausschalten mehrerer Quelltext-Zeilen benutzen. Gerade bei komplexen Verzweigungen ist das ein ganz praktisches Mittel. Nach den Test's od.ä. entfernt man einfach die Kommentar-Symbole, und schon ist der Quelltext wieder vollständig.
Ein Kommentar-Block beginnt mit einem ***/**** und endet mit dem Gegenstück ****/***. Alles dazwischen – egal über wieviele Zeilen – ist Kommentar-Text und wird nicht beachtet.

javadoc-Kommentare

Wie schon gesagt, empfinden viele Programmierer Kommentieren als lästig. Deshalb hat man in JAVA gleich ein quasi automatisches Kommentier- und Dokumentier-System mit ein-

gebaut. Dies nennt sich javadoc. Kommentiert man eine Stelle mit ***/****, dann erstellt ein Zusatzprogramm daraus eine HTML-Dokumentation. Das Zusatzprogramm nennt sich eben **javadoc**. Das Ende eines javadoc-Kommentar's ist das normale Kommentar-Ende ***/***.
Javadoc ist quasi auch ein Übersetzungs-Programm, nur das es eben die Kommentare auswertet werden und nicht der Java-Quellcode-Text.
Das Programm kann auf der Kommandozeile mit:

javadoc *Dateiname***.java**

aufgerufen werden. Im gleichen Verzeichnis finden wir dann nach dem Durchlauf eine Datei

*Dateiname***.html**

Diese können wir uns direkt im aktuellen Browser ansehen.
Die etwas besseren IDE's bieten ebenfalls einen Aufruf des javadoc-Programm, von der graphischen Oberfläche aus, an.

Aufgaben:

1. Kennzeichnen Sie mit einem Textmarker die Quelltext-Stellen, die als Kommentare festgelegt wurden!

```
…
/*
  Programm-Abschnitt berechnet das Produkt aller
  natürlichen Zahlen von 100 bis 1 zurückzählend
*/

// Deklaration
int produkt = 1;
// Berechnung
for (int i = 100; i > 0; i--) {
    produkt /* test */ *= i;  // produkt = produkt * i
}
// Ausgabe
System.out.println(produkt);
…
```

2. Geben Sie den Programm-Abschnitt in Ihrer favorisierten IDE oder einem Editor ein und erkunden Sie, wie Kommentare dargestellt / hervorgehoben werden!
3. Kennzeichnen Sie im folgenden Quelltext, die Stellen, die das Übersetzungs-Programm (der Compiler) bearbeitet!

```
…
/*
  Programm-Abschnitt berechnet den Kontostand nach
  einer Geld-Entnahme und der Berechnung der Tages-Zinsen
  laut den Bestimmungen des deutschen Bankwesens
*/

// Deklaration
float kontostand = 538.62;
float jzs = 1.35;  // JahresZinsSatz
int tagLetzteBuchung = 4269;  // interne Tageszählung
int tagHeute = 4394;
float auszahlung = 27.55;
float zinsen = 0.0;
// Berechnung
/*
  Tages-Zins-Formel:
       K * p * t
  Z = ------------
       100 * 365
*/
zinsen = kontostand * jzs * (tagHeute - tagLetzteBuchung) / 36500
kontostand = zinsen – auszahlung + kontostand;
// Ausgabe
System.out.println(kontostand);
…
```

4. Prüfen Sie das Programm auf Korrektheit!
5. Erstellen Sie ein Multiplikations-Programm für drei Gleitkomma-Zahlen nach dem EVA-Schema! Erstellen Sie kurze Kommentare für die Programm-Abschnitte, den Funktions-Umfang des Programm's und eine einfache javadoc-Dokumentation!

komplexe Aufgaben zu Verzweigungen und Schleifen:
einfach!

1. *Ermitteln Sie mit einem Programm, welche natürlichen Zahlen beginnend bei 1 man addieren muss, um mindesten die 1'000 zu überschreiten! Es genügt den Zahlenbereich in einer informativen Zeile auszugeben!*
2. *Entwickeln Sie ausgehend von einem Struktogramm ein Programm, das beginnend mit 1 bis zur 15 immer genausoviele Sternchen auf dem Bildschirm anzeigt! (Es sind keine String-Methoden zugelassen!)*
3.

etwas schwieriger / aufwändiger!

x. *Ausgehend von einer natürlichen Zahl bis zu einem Endwert soll der Schleifen-Durchlauf, die Zahl selbst, deren Quadrat und deren Kubik in einer Konsolen-Tabelle angezeigt werden!*

```
Startzahl: 4
Endzahl: 6

Durchlauf  |  Zahl  |  Quadrat  |  Kubik
-----------+--------+-----------+----------
     1     |      4 |        16 |      64
     2     |      5 |        25 |     125
     3     |      6 |        36 |     216
```

x.

x.

schon schwieriger / aufwändiger!

x. *Für die natürlichen Zahlen von 1 bis 10 soll die laufende Summe, das laufende Produkt, die Fakultät der Zahl und die laufende Summe der Fakultäten in einer Tabelle (aus ASCII-Zeichen) beechnet und angezeigt werden!*

x. *Schreiben Sie ein Programm, das für einen 6er Würfel das Gesetz der großen Zahlen prüft! Einigen Sie sich im Kurs, welche statistischen Kennwerte Sie als Kriterium benutzen wollen!*

x.

1.3.10. einfache Datenstrukturen

1.3.10.1. primitive Array's

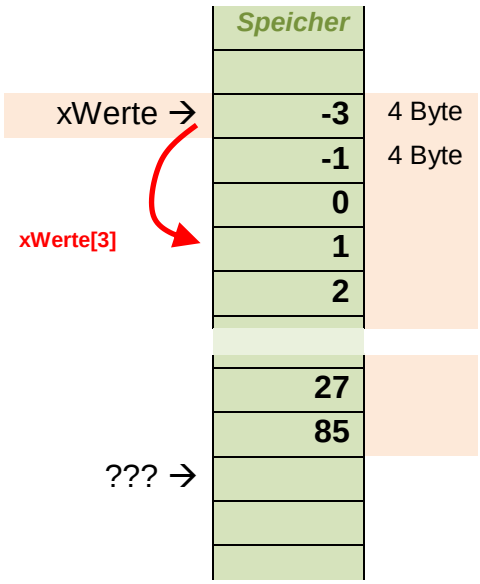
Array's sind praktisch einspaltige Tabellen. In ihnen speichern wir Gruppen von gleichen-artigen, einfachen Daten.
Die Umschreibungen für Array's im deutschen und englischen Sprachgebrauch sind sehr weit gefächert, aber auch wieder teil-weise sehr ähnlich / universell:

- Vektor; vector
- Feld; array
- Reihung, sequence, catenation
- Tabelle, Relation; relation

Array: xWerte

Element	Inhalt	
0	-3	beginnt immer beim 0. Element
1	-1	
2	0	
3	1	
4	2	
n-2	23	
n-1	27	
n	85	n. Element (festgelegtes Ende!)

Von Tabellen spricht man mesit erst dann, wenn die Felder (Array's) zweidimensional sind. Für eindimensionale Array's passt Vektor besser.
Die Größe einnes Array ist durch eine Deklaration im vorderen Teil des Quell-Codes festgelegt und ist während des gesamten Programms unveränderlich.
Im Speicher sind nur die Werte hintereinander aufgereiht. Nur das erste Element (Element 0) ist direkt ansprechbar. Um auf ein bestimmtes anderes Element zuzugreifen, berechnet das JAVA-System dann die richtige Speicherzelle.
Die Nummerierung der Array-Elemente ist also nur virtuell. Diese Index-Nummern (Element-Positionen) tauchen nirgens im Speicher auf.
Es ist bei der Planung sinnvoll das Feld ev. etwas größer anzulegen (aber Vorsicht, es wird auch entsprechend Speicherplatz gebraucht / reserviert), damit man vielleicht noch eine kleine Reserve hat.



Die aktuell benutzte Länge muss man praktische immer in einer eigenen Variable (z.B. akt-MaxElemente) verwalten. Ausnahmen sind natürlich Array, deren Länge exakt bestimmt ist und die auch vollständig mit Typ-gerechten Daten gefüllt sind.

Typ[ ] *FeldBezeichner*;
***FeldBezeichner* = new Typ[*AnzahlFeldelemente*];**

***FeldBezeichner*[Index] = …**
… = … *FeldBezeichner*[Index] …

Das Anlegen des Feldes xWerte bedarf also zwei Strukturen. Einmal wird die Art und der Name des Feldes festgelegt und dann die Speicher-Struktur inklusive Speicherplatz-Reservierung.

```
…
int[ ] xWerte;
xWerte = new int[100];
…
```

Beides lässt sich auch – wie bei anderen Variablen-Deklarationen – in einer Zeile zusammenfassen:

```
…
int[ ] xWerte = new int[100];
…
```

Die Initialisierung bzw. die Zugriffe sind dann unabhängig irgendwann im Programm möglich. Einzelne Feld-Elemente werden über ihre Position angesprochen. Sie muss in eckigen Klammern notiert werden und kann auch berechnet oder über Variablen benutzt werden.

```
…
xWerte[0]=-3;
xWerte[1]=-1;
xWerte[2]=0;
…
xWerte[12+4] = xWerte[a-2];
…
xWerte[n-2]=27;
xWerte[n-1]=85;

…
```

Die Inhalte eines Feldes können sich beliebig oft ändern. Das Feld und seine Größe sind dagegen für das Programm oder den betreffenden Block unveränderlich.
Bei der initialialen Belegung eines Feldes mit festen Werte kann man auch die folgende Struktur nutzen:alternativ:

Typ[ ] FeldBezeichner = {FeldElement , FeldElement , … , FeldElement};

Konkret also für obiges Beispiel:

```
…
int[] xWerte ={-3,-1,0, … , 27,85};
…
```

Die Datenstruktur Array stellt von JAVA aus noch eine interessante Methoden zur Verfügung, die das Leben eines Programmierers etwas leichter machen. Es handelt sich um die Bereitstellung der Länge (Größe) des Array's. Die Methode heißt length() und wird uns noch häufig im gleichem Sachzusammenhang begegnen.

FeldBezeichner**.length()**

1.3.10.1.1. erweiterte for-Schleifen

Beim Durchsuchen eines Feldes hat der Index / die Lauf-Variable manchmal nur einen Selbstzweck. Eigentlich bräuchte man ihn nicht. JAVA bietet eine spezielle Version für Zählschleifen an.
Bei dieser erweiterten (eigentlich vereinfachten) for-Schleife iteriert man praktisch indirekt mit einer Arbeits-Variable.

for (FeldelementTyp *ElementBezeichner* : *FeldBezeichner*) { …

Wie verständlich solche Strukturen werden können, kann man an den folgenden Beispielen sehen:

```
for (String monat : monatsNamen) { …
for (String farbe : farbenKatalog) { …
for (int nummer : gueltigeNummern) { …
for (char zeichen : alphabet) { …
…
```

Die Schleifen-Strukturen können fast umgangssprachlich gelesen werden, z.B.:

FÜR (jeden) monat IN monatsNamen TUE …
…
FÜR (jede) nummer IN gueltigeNummern TUE …
…

Einige Einschränkungen und Besonderheiten der erweiterten Schleifen sollten aber immer vor der Codierung beachtet werden:

- In erweiterten for-Schleifen lassen sich keine Werte in das Array zurückschreiben.
- Es wird das gesamte Feld Element für Element von vorne nach hinten durchsucht.

Die Geschichte mit den erweiterten for-Schleifen lässt bis zu anonymisierten Feldern weiterführen:

```
…
for (int primzahl : new int[]{2,3,5,7,11,13,17,19,23,29}) {
    System.out.print(primzahl+"  ");
}
System.out.println("");
…
```

1.3.11. JAVA-Schlüsselwörter (erster Grundwortschatz)

boolean

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

break

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

catch

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

class

Syntax	Beschreibung
	defniert eine Objekt-Klasse (Typ-Beschreibung von Objekten)

Beispiel(e)	Kommentar(e)

do … while

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

double

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

else

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

false

Syntax	Beschreibung
`false`	Wert für Nicht-WAHR also FALSCH

Beispiel(e)	Kommentar(e)

float

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

for

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

if

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

if … else if …

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

if … else if … else …

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

int

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

new

Syntax	Beschreibung
	erstellen einer Instanz von einer Objekt-Klasse (instanzieren eines neuen Objektes einer Klasse)

Beispiel(e)	Kommentar(e)

public

Syntax	Beschreibung
	einleitendes Schlüsselwort (Sichtbarkeits-Modifikator) zur Festlegung der Sichtbarkeit der folgenden Ausdrücke in anderen Klassen und Paketen **die folgenden Ausdrücke sind uneingeschränkt sichtbar**

Beispiel(e)	Kommentar(e)

static

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

String

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

switch

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

true

Syntax	Beschreibung
`true`	Wert für WAHR (bzw. Nicht-FALSCH)

Beispiel(e)	Kommentar(e)

void

Syntax	Beschreibung
	Typ-freie Methode (es gibt keinen Rückgabe-Wert!)

Beispiel(e)	Kommentar(e)

while

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

Konventionen für eine Standard-orientierte Quelltext-Gestaltung in JAVA:
(gelten innerhalb des schulischen Programmierens als Quasi-Standard und sind Bewertungs-relevant!)

- es sind sprechende Bezeichner zu favorisieren, nur in super-klaren Fällen können einzelne oder wenige Buchstaben genügen
- Klassen erhalten einen Bezeichner, der mit einem Groß-Buchstaben beginnt (stellen übergeordnete Substantive / Begriffe dar)
- Objekte (Instanzen), Attribute und Methoden beginnen mit einem Klein-Buchstaben
- zur verbesserten Lesbarkeit bei zusammengesetzen Namen können Unterstriche oder Groß-Buchstaben genutzt werden (CamelCase))
- Konstanten werden in Großbuchstaben notiert, dabei können / sollten Unterstriche zur Strukturierung in Teilwörter benutzt werden (Underscore)
- es gibt geschützte Wörter, die nicht als Bezeichner benutzt werden dürfen, sie sind Schlüsselwörter der Programmier-Sprache JAVA

class	char		
double		float	
int	new		public
return	static	void	

- oder auch vordefinierte Klassen, die praktisch ebenfalls als reservierte Namen zu betrachten sind:

	Math	String

1.4. von Anfang an: Objekt-orientierte Programmierung

1.4.0. Grundlagen

Unsere Umwelt – aber auch wir selbst – sind aus verschiedenen Dingen aufgebaut. Die Dinge werden in der Informatik Objekte genannt. Zur genaueren Bestimmung könnte man sie auch Real-Objekte nennen.
Objekte der Realität werden als Modelle in der Informatik simuliert / umgesetzt / berechnet.

Objekte gehören in unserer Denk-Welt immer auch wieder zu übergeordneten Gruppen von Dingen – die werden in der Informatik Klassen genannt.
Wir sprechen nicht so viel vom Baum 53 in der Schloßallee, sondern vielmehr von Bäumen oder der Baum-Reihe (Allee).
Sowohl die Bäume als auch die Allee sind keine wirklich existierenden Objekte sondern nur Zusammenstellungen von Objekten oder von uns gebildetete Begriffe, die uns den Umgang mit vielen Objekten zu vereinfachen. Solche Gruppen nennen wir in der Informatik Klassen.
Unsere Hierrarchien sind z.T. sehr ausgefeilt. Sie orientieren sich anden unterschiedlichsten Verwendungs-Zwecken eines Objektes. Kaum eine dieser Gruppen hat aber in der Realität eine Existenzberechtigung. Es sind einfach nur menschliche / künstliche Modelle über ein Objekt.
Die Programmierungs-Art, die genau auf Klassen und Objekten basiert, wird Objekt-orientierte Programmierung (OOP) genannt.
Die OOP ist also eine relativ Menschen-nahe Umsetzung von berechenbaren Problemen. Sie basiert ganz wesentlich auf der Denk- und Arbeitsweise von uns Menschen. Leider können Computer aber dieses menschliche Denken und Arbeiten nicht. An einer Stelle muss deshalb die Umsetzung des menschlichen Denkens in die Computer-Arbeitsweise erfolgen. Im Falle von JAVA macht dies genau das JAVA-System.
Klassen verstehen wir als Beschreibung (Baupläne / Blaupausen) oder Zusammenfassungen von ähnlichen Objekten.
Objekte beinhalten charakterisierende Merkmale und Zustände (state's) und ein typisches Verhalten (Behavior).
Jedes Objekt ist durch eine Kombination bestimmter Merkmale oder Eigenschaften sowie von Fähigkeiten und Fertigkeiten geprägt.
Maxi ist ein Hund mit einem bräunlich schwarzen Fell, großen stehenden Ohren und mittellangen Beinen. Er kann laufen, bellen, schwimmen und apportieren.

Maxi
Rasse: Schäferhund
Alter: 5
Masse: 12 kg
Geschlecht: weiblich
Wurmbehandlung: ja

bellt
läuft
frisst gerne Leckerli
…

Objekt-Name
Merkmale

Fähigkeiten

Definition(en): Objekt
Objekte sind die Dinge / Originale aus der reelen Welt.
Objekte sind die Dinge, auf die eine Person oder ein Prozess den Fokus / seine Aufmerksamkeit gelegt hat.
In der Objekt-orientierten Programmierung versteht man unter Objekten, die aus den Klassen abgeleiteten Instanzen dieser Klasse. (Es kommt zu einer quasi-Gleichsetzung zwischen Real-Ding und der Modell-Repräsentation im Computer-Programm.)

Andere Hunde haben andere Merkmals-Kombinationen, die sie jeweils individuell charakterisieren.
Damit wir Menschen mit mehreren / vielen Hunden klar kommen, und vielleicht auch Hunde von Katzen unterscheiden können, bilden wir zuerst Gruppen von Objekten. Diese sind immer durch bestimmte allgemeingültige Merkmale gekennzeichnet
So hat ein Hund z.B. die Merkmale Name, Rasse, Gewicht und Alter. Die Merkmale werden allgemein auch **Attribute** genannt. Das mögliche Verhalten oder die vorhandene Fähigkeiten / Fertigkeiten werden als **Methoden** bezeichnet. Zu den Methoden zählt man auch die Dinge, die man mit dem Objekt bzw. eben der Klasse machen kann.

Definition(en): Klasse
Eine Klasse ist eine Gruppe von Objekten oder (untergeordneten) Klassen, die sich durch gemeinsame Merkmals-Kategorien / Eigenschaften-Gruppen zusammenfassen lassen.
Eine Klasse ist eine Bauvorschrift für eine Instanz (/ ein Objekt). Sie beinhaltet Attribute und Methoden.

Die Darstellung einer Klasse könnte dann so aussehen:

Hund
Name Rasse Steuernummer Alter Masse Wurmbehandlung … gib_Laut läuft_nach hat _Geburtstag bekommt_Steuernummer merke_Wurmbehandlung zeige_Wurmbehandlung …

Klassen-Name
Attribute
Methoden

Diese aufgezeigte Kasten-Darstellung ist Teil einer standasierten Diagramm-Art, die sich in der modernen Infromatik extrem schnell durchgesetzt hat. Es handelt sich um ein sogenanntes UML-Diagramm. UML steht dabei für Unified Modeling Language (dt: einheitliche Modellierungs-Sprache).
Mit solchen Diagrammen kann man vor allem größere Zusammenhänge mit vielen Klassen sehr übersichtlich und effektiv darstellen. Als besonders günstig hat sich die Kombination von UML-Diagrammen mit der Objekt-orientieren Programmierung herausgestellt. Viele Elemente der Diagramme können fast eins zu eins in Programm-Code übersetzt werden.
Betrachten wir nun noch mal ein anderes Objekt: Bello. Bello hat andere Eigenschaften / Merkmale als Max.

Bello	Objekt-Name
Rasse: Terrier *Alter: 3* *Masse: 5 kg* *Geschlecht: männlich* *Wurmbehandlung: ja* …	**Attribute**

Irgendwie müssen später diese Eigenschaften in den Klassen gespeichert werden. In der OOP nennen wir die Merkmale / Eigenschaften einer Klasse Attribute. Für Hunde könnten solche Attribute z.B. Rasse, Alter, Name usw. sein. Attribute beschreiben immer Objekte (Rasse, Name, …) oder geben ihren Zustand (Alter, Wurmbehandlung, …) wieder.
Zu den Attributen – allg. auch Zustands-Variablen genannt – gehören noch Datentypen, die im Programm benutzt werden sollen. Die Speicherung der Daten muss ja im Computer in irgendeiner Form erfolgen.
Der Programmierer legt dabei fest, in welcher Form die Daten gespeichert werden soll. So könnte man z.B. das Geschlecht als Wörtchen ("weiblich" | "männlich") – aber auch nur als "w" bzw. "m" speichern. Ganz andere Programmierer wählen "0" oder "1".
Für die Wörtchen müssten wir den Datentyp String wählen, für eine Buchstaben würde char reichen. Der Programmierer mit den Ziffern könnte dagegen int nutzen. In jedem Fall kann die gleiche Information auf verschiedene Art und Weise gespeichert werden. Damit alle Programmierer hierüber gut informiert sind, notiert man den Datentyp gleich mit in das UML-Diagramm:

Hund	**Klassen-Name**
Name: String Rasse: String Steuernummer: Integer Alter: Integer Masse: Float Wurmbehandlung: Boolean …	**Attribute** Datentypen: String … aphanumerische Zeichen-Ketten Integer … ganze Zahlen Float … Fließkommazahlen Boolean … Wahrheitswert
	Methoden

Ähnlich wie bei den Attributen haben wir es auch bei dem Methoden mit den unterschiedlichsten Datentypen zu tuen. Zum einen benötigen viele Methoden noch zusätzliche Informationen, z.B. die Methode "gib_Laut" die Anzahl der Laute. Die notwendigen Daten für eine Methode werden **Argumente** genannt.
Methoden liefern aber auch Informationen zurück. So könnte "gib_Laut(3)" den Text "wau wau wau" zurückliefern. Natürlich könnte es auch eine MP3-Abspielung sein. Also müssen auch die Rückgabe-Werte einer Methode klar definiert werden. Es wäre warscheinlich nicht wirklich toll, wenn ein anderer Programmierer versuchen würde den Text "wau wau wau" statt auf dem Bildschirm an den Lautsprecher zu schicken. Das werden dann wohl nur Knack-Geräusche werden.
In der OOP mit JAVA erfolgt die Klassen-Definition immer jeweils einzeln in einer Datei. Die Datei erhält dann auch den Klassen-Namen gefolgt vom Dateityp ***.java***.

Unsere Hunde-Klasse kann aus praktischen Gründen in übergeordneten Klassen eingeordnet werden. So könnte man z.B. eine Klasse Lebewesen aufmachen, die den wissenschaftlichen Namen verwaltet. Da Hunde Lebewesen sind bekommen die Hunde irgendwie die Merkmale von Lebewesen zugeordnet. Dazu gleich noch mehr. Untergeodnete / eingeordnete Klassen sind immer Spezialfälle der übergordneten Klasse. Man spricht auch von Vererbung. Die untergeordnete Klasse erweitert die übergeordnete Klasse.
Jede von uns definierte Klasse ist eine Erweiterung der elementaren Klasse Object. Bei der Definition einer Klasse muss die Hierrarchie angegeben werden. Das macht man mit dem Schlüsselwörtchen **extends**.
Die hierrarchische Einordnung muss nicht immer vorgenommen werden. Eigenlich hätte man ja bei jeder Klassen-Definition bisher auch die Ober-Klasse "Object" mit angeben müssen. Die wird vom JAVA-System automatisch vorgenommen. Also ist jede von uns programmierte Klasse immer eine Unterklasse von "Object". IN JAVA muss dieser Allgemeinplatz nicht mit notiert werden.
Somit sind die beiden folgenden Klassen-Definitionen gleich.

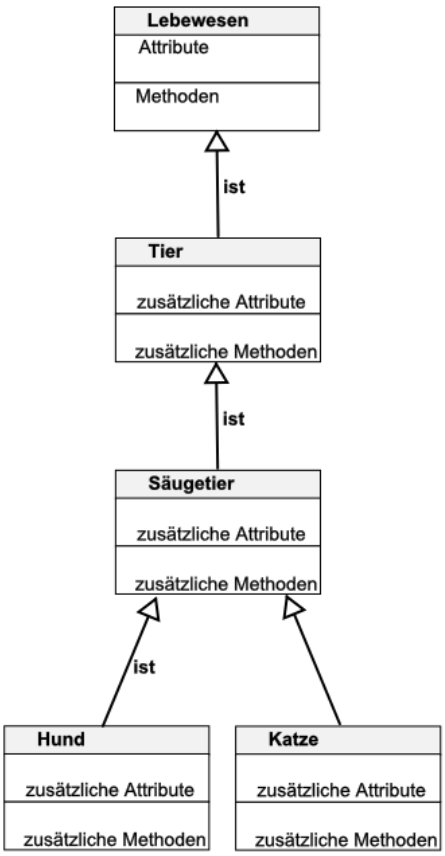
```
public class KlassenName {
      AttributDefinition;
…
}
```

```
public class KlassenName extends Object {
      AttributDefinition;
…
}
```

Eine Klassen-Definition erfolgt also in JAVA über das Schlüsselwort **class**. Es folgt der Name der Klasse. Dieser wird auch KlassenBezeichner genannt.

class *KlassenBezeichner* { …

Die exakte Darstellung einer Klasse in UML-Diagrammen erfolgt mittels unterteilter Rechtecke, die neben dem Klassen-Namen noch die Attribute und Methoden enthalten. Auf die beiden letzteren wollen wir jetzt noch nicht so genau eingehen. Deshalb sind sie im UML-Diagramm auch nicht konkretisiert.
Der Pfeil mit der innen ungefärbten Dreiecks-Spitze ist der sogenannte **IST**-Pfeil. Er beschreibt die hierrarchische Unter- bzw. Überordnung von Klassen. Das Wörtchen ist muss nicht mitgeschrieben werden, da schon durch die Pfeil-Art diese Festlegung gemacht wurde. Für Anfänger lesen sich UML-Diagramme mit ein paar mehr Informationen aber besser.
In unseren Programmen arbeiten wir nicht mit den Klassen sondern wir wollen dort unsere Objekte (aus der Umwelt) wiedergeben. Das Erstellen von Programm-Repräsentationen unserer Objekte wird Instanzierung genannt. Dabei wird die Klassen-Definition als Bauplan / Blaupause / Kopier-Vorlage benutzt, um z.B. "Maxi" und "Bello" im Computer zu modellieren.



Diese Modell-Repräsentation im Computer auf der Basis der Klasse "Hund" wird als Instanz bezeichnet.
Viele Informatiker / Programmierer leben so in der virtuellen Welt ihrer Programme, dass sie die Real-Objekte und die Instanzen gleichsetzen. Deshalb wird im Programmierer-Jargon statt von Instanzen gleich von Objekten gesprochen. Zumindestens in den allgemeinen, erläuternden Kapiteln werde ich die Begriffe vorrangig ordnungsgemäß benutzen. In Klammern gebe ich dann u.U. den Jargon-Namen mit an. Aber ich unterliege natürlich auch dem – bei Entwicklern üblichen – Sprachgebrauch.

Definition(en): Instanz
Eine Instanz ist ein (informatisches) Äquivalent eines Objektes und wird aus einer Klasse abgeleitet / heraus erstellt. In vielen (Objekt-orientierten) Programmier-Sprachen werden Instanzen oft als Objekte bezeichnet.

Klassen können wieder andere (untergeordnete) Klassen benutzen
bzw. in einer übergeordneten Klasse können (untergeordnete) Klasse (mit all ihren Attributen und Methoden (je nach Gültigkeit)) verwendet werden

Will man wissen, von welcher Art Klasse eine Instanz (ein Programm-Objekt) ist, dann man mit der Methode **.getClass()** den aktuellen Objekt-Typ abfragen. Diese Methode liefert mir den Typen, de für die Erzeugung genutzt wurde.

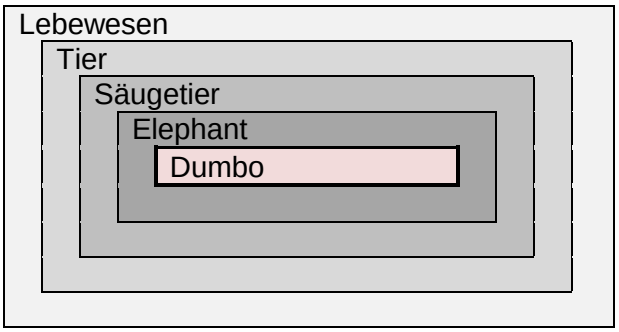
Beispiel:

```
Tier elephant = new Saeugetier("Dumbo");
```

Die getClass()-Methode:

```
elephant.getClass();
```

liefert: "Saeugetier" zurück.
Alternativ gibt es den Operator **instanceof**. Mit ihm kann die Objekt-Zugehörigkeit einschließlich der Abstammung geprüft werden.




```
elephant instanceof Saeugetier      → true
elephant instanceof Schraube        → false
elephant instanceof Tier            → true
elephant instanceof Object          → true
```

Hat man dagegen

```
Tier elephant = new Tier("Dumbo");
```

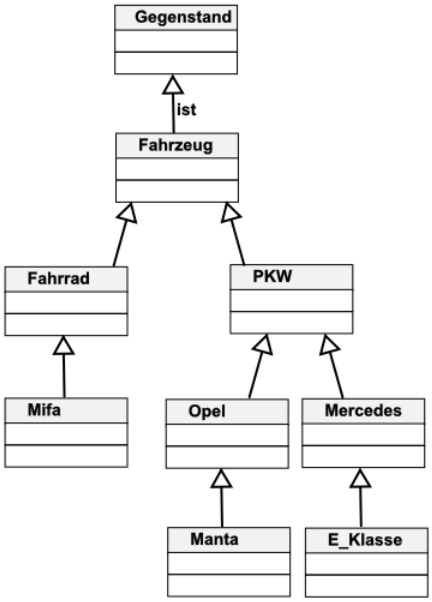
zur Definition benutzt, dann ist eine Zuodnung zum Säugetier logischerweise nicht möglich.

Aufgabe:
Zeichnen Sie eine Klassen-Einordnung als Mengen-Darstellung (s.a. oben) für die Definition von "Dumbo" als Tier!

```
elephant.getClass();                    →   Tier

elephant instanceof Sauegetier          →   false
elephant instanceof Schraube            →   false
elephant instanceof Tier                →   true
elephant instanceof Object              →   true
```

Aufgaben:

1. ***Gegeben ist das nebenstehende UML-Diagramm. Wieviele Möglichkeiten hätten Sie ein Objekt "meinMercedes" zu deklarieren? Geben Sie die Möglichkeiten als Quellcode an!***
2. ***Wählen Sie sich ein beliebiges Objekt und erstellen Sie ein UML-Diagramm mit mindestens 4 übergeordneten Klassen zu diesem Objekt!***
3. ***Geben Sie vier mögliche Deklarationen für Ihr Objekt als JAVA-Quellcode an!***



Attribute

Attribute stellen die Eigenschaften / Merkmale / Zustände eines Objektes dar. Sie benötigen eindeutige Bezeichner und sind innerhalb einer Klasse zu deklarieren.

Attribute erhalten von JAVA beim Anlegen im Speicher einen Default-Wert.

Der Wert **null** stellt einen leeren Zeiger bzw. Zeiger **NIL** dar. Das ist quasi das Nirvana, ein riesiges leeres Loch bzw. das unendliche Nichts.

Aber es ist zu beachten, dass nur Attribute einen Default-Wert bekommen. Lokale Variablen müssen immer selbst initialisiert werden.

Datentyp	Default-Wert	
int **long**	0	
double **float**	0.0	
boolean	false	
char	'\u0000'	
String	null	null ist ein leerer, nicht definierter String null ist ein Schlüsselwort von JAVA und kann z.B. in Vergleichen benutzt werden
Objekt-Datentypen	null	

Passiert dies nicht, dann gibt es eine Fehler-Meldung des Compiler's.

Das sollte man sich auch immer gleich für die Definition angewöhnen, dann sind böse Überraschungen schon seltener.

Bei nicht ordnungsgemäß gesetzten Werten einer lokalen Variable oder beim Benutzen eines leeren String (null-Strings) erhalten wir auch eine Fehler-Meldung, die von einem NullPointer (Null-Zeiger spricht. Auch auf nicht richtig erstellte Objekte zeigt zuerst nur Null-Zeiger!

Das Benutzen von Attributen (lesen und schreiben) von außen (also von einem untergeordneten Klasse) sollte nur über spezielle Funktionen (Get/Set-Methoden) erfolgen.
Wir haben schon gesagt, dass der Programmierer recht viele Freiheiten bei der Festlegung der Datentypen für seine Klassen-Attribute hat. So könnte ein Programmierer intern einen Geld-Betrag immer als Text mit Währung darstellen ("123,45 €"). Da sind vielleicht Fehlinterpretationen ausgeschlossen. Die Verarbeitung – also das Rechnen mit dieser Daten-Darstellung – wäre aber eine Qual. Also könnte man auf double gehen. Dann würde der Geldbetrag z.B. so aussehen: 123.45. Eine andere Idee wäre es, den Betrag als Cent-Wert im Datentyp int zu speichern. Jede Variante hat so ihre Vor- und Nachteile. Der Programmierer oder das team entscheidet sich für eine Variante und dann wird das so umgesetzt. Am Ende muss aber der Nutzer ordnungsgemäße Ausdrucke bekommen. Auch seine Eingaben müssen vielleicht umgewandelt werden, damit sie intern verwendet werden können. Diese Aufgabe übernehmen die GET- und SET-Methoden. Man spricgt auch von **Get**er und **Set**er. Die GET-Methode liesst das Attribut und gibt es in einer nutzbaren Form zurück. Die SET-Methode trägt den "normalen" Wert in die interne Speicherform ein.
Werden z.B. Attribute für die Klasse selbst und nicht für die einzelnen instanzierten Objekte gebraucht, dann muss diesen Attributen der Modifikator **static** vorangestellt werden. Solche Static-Attribute könnten z.B. ein Instanzen-Zähler oder die Summe oder das Maximum eines ausgewählten Attributes über die gesamte Klasse sein.
Alle erzeugten Objekte der Klasse haben zu einem Zeitpunkt immer alle den gleichen Wert eines statischen Attributs. D.h. z.B. wenn eine statisches Attribut die Anzahl der erzeugten Objekte beeinhaltet, dann kann jedes Objekt – egal wann es selbst erzeugt wurde – die (aktuelle) Anzahl der Objekte abfragen. Diese ist für alle zu diesem Zeitpunkt gleich.

Definition(en): Attribut
Attribute sind die Merkmals- / Eigenschaften-Namen eines Objektes oder einer Klasse.

<u>Methoden</u>

Methoden spiegeln das Verhalten, die Funktionen, die möglichen Tätigkeiten eines Objektes wieder. Sie werden innerhalb eine Klasse definiert und gehören primär immer zu einer Klasse. Methoden können von übergeordneten Objekten aufgerufen werden.
Übergeordnete Methoden mit dem gleichen Namen können untergeordnete Methoden überschreiben.
Innerhalb einer Methoden können völlig frei Variablen definiert werden, die aber nur innerhalb der Methode gültig sind (→ Scope). Nach dem Abarbeiten einer Methode werden die darin benutzten Variablen zerstört (gelöscht) und stehen nicht mehr zur Verfügung.

Definition(en): Methode
Methoden sind Funktionalitäten / Fähigkeiten / Tätigkeiten, die ein Objekt oder eine Klasse besitzt und diese ausführen kann.

Die Reihenfolge der Definition von Methoden – also ihrer Position im Quelltext – ist nicht notwendigerweise sequentiell. JAVA ist hier sehr freizügig. Eine aufzurufende Methode kann auch weiter hinten im Quellcode stehen (nur vorhanden sein muss sie!)

Man erreicht aber eine bessere Lesbarkeit des Codes, wenn man aufzurufende Methoden weiter vorne notiert. Das erleichtert auch eine ev. notwendige Übertragung eines Quellcodes in eine andere Programmiersprache (welche die Reihenfolge beachtet).
Da sich Objekte immer von anderen ableiten, bekommen sie von diesen Vorgängern auch immer Attribute und Methoden mit.
Eine wichtige Methode ist z.B. **toString()**. Diese Methode gibt, wenn nicht von den Vorgängern anders definiert, den Objekt-Namen und dann gefolgt von einem @-Zeichen die Speicher-Adresse (Objekt-Zeiger) aus

z.B.: Objekt@f345ac49

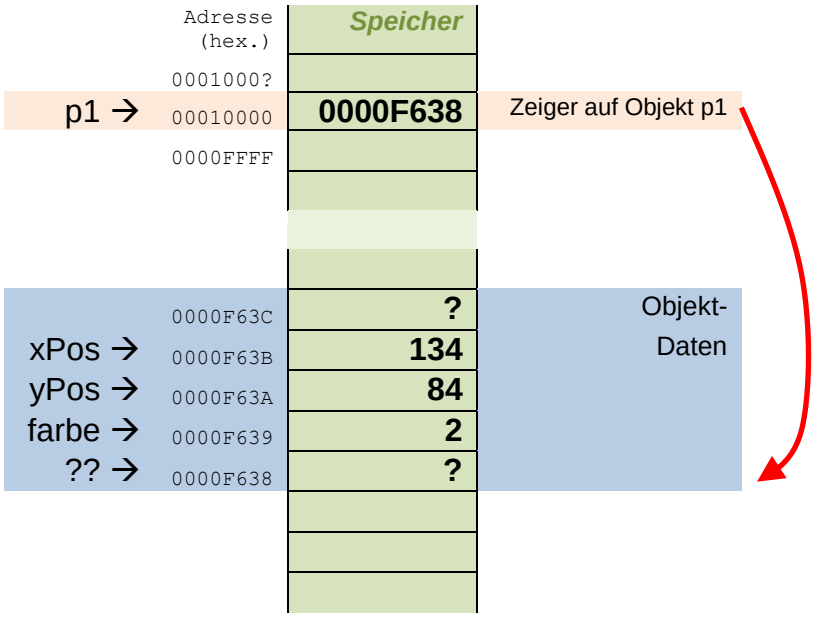
Begrüßung des Nutzers mit seinem Anmeldenamen:
https://open.hpi.de/courses/javaeinstieg2017/items/5ZKm9pzJqUycr0KUXKTZuf

```
package de.kompf.tutor;

/**
 * Simple java program to greet the user.
 */
public class Hello {

  /**
   * MAIN
   * @param args ignored
   */
  public static void main(String[] args) {
    Hello hello = new Hello();
    hello.greet();
  }

  private void greet() {
    String user = System.getProperty("user.name");
    System.out.println("Hello " + user + "!");
  }
}
```

Q: https://www.kompf.de/java/firststeps.html

1.4.1. Objekt-Datentypen

Die Dinge aus der Realität, die wir in Computer-Programmen bearbeiten wollen, sind selten nur einfache Daten. Vielmehr sind sie komplex, mehrdimensional oder hierrarchisch strukturiert. JAVA kennt verschiedene Objekt-Datentypen, die Möglichkeiten für die komplexere Verarbeitung von Daten bieten.
String als Objekt-Datentyp haben wir schon kennengelernt. Er stellt eine Reihe von alphanummerischen Zeichen – praktisch Buchstaben usw. – dar (→ 1.3.7.0.4. Zeichenketten - Strings).
Ähnlich verhält es sich mit anderen Objekt(-Daten)-Typen. Praktisch sind es immer Zusammenstellungen von einfachen Daten (Attributen) zu immer komplexeren Strukturen. Je spezieller die Objekte werden, umso mehr Spezial-Informationen müssen zu ihnen gespeichert werden.
In Objekt-Datentypen können nur Objekte (bzw. ihrer Referenzen / Zeiger) gespeichert werden. Man kann sich das so vorstellen, dass wenn ein Objekt angelegt wird, das im Speicher so aussieht, wie nebenstehen abgebildet. Zum Objekt selbst gehört im Speicher nur eine Speicheradresse, die einen Verweis (eine Referenz / einen Zeiger) auf eine Speicherstelle darstellt, wie die eigentlichen Daten zum Objekt gespeichert sind.



Praktisch sind diese Referenzen mehrstufig, was uns als Programmierer nur indirekt interessiert. Für uns sind es hierrarchisch angeordnete Objekte.
Um in Objekt-Datentypen bestimmte einfache Daten, wie Ganzzahlen oder Gleitkommazahlen, zu speichern, müssen diese dann zuerst in Objekte umgewandelt werden (, damit wir dann die Verweise auf sie in den Objekt-Datentypen speichern / verwalten können). Das Umwandeln besprechen wir gleich im nächsten Unter-Kapitel zum sogenannten Wrapping (→ 1.5.3.1. Erzeugen / Anlegen von Objekten mit Objekt-Datentyp (Wrapper-Objekte)).
Einiges wollen wir hier an einer der einfacheren Objekt-Datentypen besprechen – den Strings. Texte können wir uns gut vorstellen und die Struktur ist auch denkbar einfach.
String als Objekt-Typ, der schon vorgefertigt durch die JAVA-API zur Verfügung gestellt wird, besitzt eine Unzahl von vordefinierten Methoden.
Einige stellen wir kurz vor. Wir beschränken uns aber auf die gängigen Methoden, die man kennen sollte (nicht in allen Details, aber das es sie gibt und was sie prinzipiell leisten).
Die vordefinierten JAVA-eigenen Methoden sollte man vorrangig vor eigenen, selbstgeschrieben Methoden über Strings, verwenden.

Vorteile:

- schneller (weil in Maschinen-nahen od. Maschinen-Sprachen programmiert)
- super getestet
- wird von JAVA / Profi's gewartet

Nachteile:

- interner Ablauf bleibt im Verborgenem
- kleine Abwandlungen benötigen dann die Entwicklung der gesamten Methode als Eigenleistung

Anhand der String-Methoden kann man sich auch noch mal die Verwendung der Punkt-Schreibweise vergegenwärtigen. Methoden zu einem Objekt werden immer nach der Nennung des Objektes hinter einem Punkt angegeben.

ObjektBezeichner.*MethodenName()*

Genauso verhält es sich mit den Attributen – also irgendwelchen Eigenschaften der Objekte. Auch hier verwendet man die Punkt-Schreibweise:

ObjektBezeichner.*AttributName*

Methoden und Attribute sind immer gut an den runden Klammern zu unterscheiden. Nur bei den Methoden muss eine Klammerpaar angegeben werden. Ev. können in den Klammern weitere notwendige Angaben / Informationen für die Methode an diese übergeben werden.

Die Deklaration

Erstellen eines Objektes (einer Instanz):

```
Datentyp  Bezeichner = new  Klasse;
```

Ansprechen / Benutzen des Objektes erfolgt nun unter / über dem Bezeichner. Praktisch ist der Beuzeichner wieder ein Zeiger, der auf den Speicher-Bereich zeigt, in dem die Attribute als Speicher-Stellen aufgereiht sind. Eine Belegung der Attribute kann im Konstruktor erfolgen.

Objekt-Datentypen enthalten immer die Methoden:

Standard-Methoden von Objekten

- **Prüfung der Gleichheit** — **equals(ObjektX)**
  praktisch aber nur der Vergleich der Speicher-Adressen der beiden Objekte (Zeiger-Vergleich)
  kann überschrieben werden (für spezielle gewollte / vorrangige Vergleiche), um dann wirklich die / bestimmte Attribute der beiden Objekte zu vergleichen

- **String-Repräsentation** — **toString()**
  Ausgabe-Standard: Klassenname@Speicheradresse

-

Interessanterweise stellt die JAVA-API auch für die einfachen Datentypen Objekt-Versionen zur Verfügung. Was soll denn das?
Es handelt sich dabei um die abstrakten Klassen der Datentypen. Man spicht auch von Wrapper-Klassen (dt.: Ummantelungs- / Mantel-Klassen; eng.: Envelope Classes).
Die Wrapper-Klassen bieten Hilfs-Methoden an, die vielfach gebraucht werden.

einf. / prim. Datentyp	**Objekt-Datentyp**	
byte	**Byte**	
int	**Integer**	
long	**Long**	
double	**Double**	
float	**Float**	
boolean	**Boolean**	
char	**Character**	
kein echter Datentyp		
void	**Void**	

Wer kann sich schon genau merken, wie groß oder klein der größte bzw. der kleinste mögliche Wert bei den int-Zahlen ist. Hier helfen Attribute der Wrapperklassen, dieses Manko auszugleichen.

Weiterhin sind diese Strukturen notwendig, um die interne Repräsentation von komplexeren Datenstrukturen, wie Mengen und Listen für JAVA zu ermöglichen (→ 1.5.3.2. Kollektionen).
JAVA kann in komplexen Datenstrukturen nur Referenzen (Zeiger) auf Variablen mit primitive Datentypen speichern.
Wie bei Strings werden auch hier passende Methoden von der JAVA-API angeboten.
Eine wichtige Information könnte bei den Zahlen-Datentypen für uns sein, wie groß die maximal zu speichernde Zahl in einem Datentyp sein darf. Dazu gibt es dann z.B. die Objekt-Konstante (naja praktisch ist es wohl ein Attribut) .MAX_VALUE.
Oder für die lästige und auch nicht triviale Aufgabe einen String in einen sauberen Integer-Wert zu wandeln, gibt's z.B. .parseInt(String s).

Auch wenn es bei den Objekt-Datentypen schon recht kompliziert und verzwickt sein kann, gehören sie im JAVA-Jargon noch zu den einfachen Datentypen. Komplexe Datentypen sind aus anderen Objekt-Datentypen oder komplexen Datentypen aufgebaut (→ Verschachtelung, Aneinanderreihung). Die komplexen Objekt-Datentypen werden in JAVA Kollektionen genannt (→ 1.5.3.2. Kollektionen).

1.4.2. Vererbung

Ein anderer Zugang zum Sinn von Klassen und hierrarchischen Strukturen zwischen ihnen, wird vielleicht dann ersichtlich, wenn man ähnliche Objekte händeln muss.
So könnte jemand ja nun auch Katzen mit in das Programm hineinbringen wollen. Also überlegen wir uns auch hier die notwendigen Attribute und Methoden.

Katze	Klassen-Name
Name	Attribute
Rasse	
Steuernummer	
Alter	
Masse	
kastriert	
…	
gib_Laut	Methoden
läuft_nach	
hat _Geburtstag	
bekommt_Steuernummer	
setze_kastriert	
zeige_kastriert	
…	

Dabei fällt schnell auf, dass Hunde und Katzen doch einige gemeinsame Attribute und Methoden haben. Das würde bedeuten, wir müssen den gleichen Programm-Code sowohl für die Katzen als auch für die Hunde schreiben. Da könnte man sich vielleicht mit Kopieren helfen, aber was passiert, wenn uns später Fehler auffallen und wir diese berichtigen wollen. Nun müssen wir in zwei Programm-Teilen praktisch parallel die Korrekturen vornehmen. Das geht erfahrungsgemäß schief. Besser ist es die gemeinsamen Attribute und Methoden auch nur einmalig in einer übergeordneten (Super-)Klasse zu verwalten. Eine solche Klasse könnte z.B. Haustier oder Kleintier sein. In dieser werden nur die gemeinsamen Attribute und Methoden definiert und bearbeitet.

Kleintier	Klassen-Name
Name	Attribute
Rasse	
Steuernummer	
Alter	
Masse	
…	
gib_Laut	Methoden
läuft_nach	
hat _Geburtstag	
bekommt_Steuernummer	
…	

Für die Betrachtung eines Hundes oder einer Katze greifen wir auf "Kleintier" zurück und ergänzen um die speziellen Hunde- oder Katzen-Attribute bzw. –Methoden.
Wenn wir uns ein Objekt Hund anlegen, dann wird also zuerst einmal ein Kleintier erstellt und dann um die speziellen Hunde-Eigenschaften ergänzt. Insgesamt haben wir dann ein Hund-Objekt. Dieses hat viele Attribute und Methoden schon von Kleintier – übernommen oder wie es in der Programmierung heißt – "geerbt".

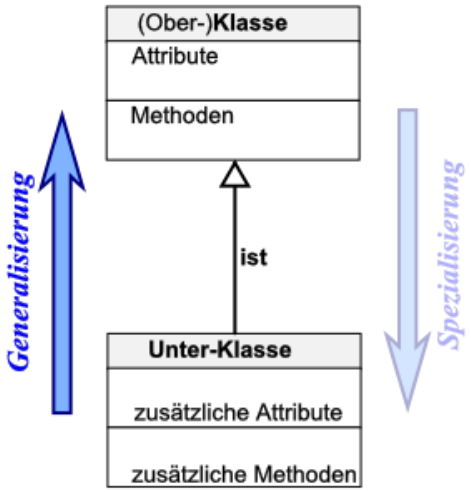
Hund	→ **Kleintier**	Klassen-Name + Verweis
Name		geerbte Attribute
Rasse		
Steuernummer		
Alter		
Masse		
Wurmbehandlung		neue Attribute
…		
gib_Laut		geerbte Methoden
läuft_nach		
hat _Geburtstag		
bekommt_Steuernummer		
merke_Wurmbehandlung		neue Methoden
zeige_Wurmbehandlung		
…		

Hund	→ **Kleintier**	Klassen-Name + Verweis
Wurmbehandlung		neue Attribute
…		
merke_Wurmbehandlung		neue Methoden
zeige_Wurmbehandlung		
…		

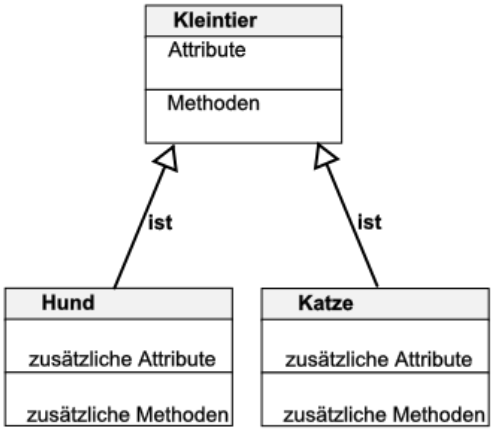
Das Verfahren von der Übernahme von übergeordneten / allgemeineren Merkmalen und Fähigkeiten von einer Super-Klasse (Eltern- / Ober-Klasse) auf eine konkretere / speziellere Unterklasse (Sub-Klasse; Tochter- od. Kind-Klassen) wird auch Vererbung (engl.: inheritance) genannt.

Definition(en): Vererbung
Vererbung ist ein grundlegendes Prinzip der Objekt-orientierten Programmierung, bei dem aus Basis-Klassen (übergeordneten / allgemeineren Klassen) abgeleitete (untergeordnete / speziellere) Klassen erstellt werden.
Unter Vererbung versteht (in der Informatik) die Umsetzungs-Mechanismen für Beziehungen (Relationen) zwischen Klassen verschiedener Hierrachie-Ebenen (Ober- und Unter-Klassen). Dabei werden i.A. Attribute und Methoden der Ober-Klasse auch den Unter-Klassen verfügbar gemacht.

Abgeleitete Klassen haben im UML-Verständnis eine "IST"-Beziehung ("ist_ein"-Beziehung).
Das Lesen der UML-Diagramm unter dem hierrarchischen Aspekt ist beginnt man besser von unten. Das ist zu Anfang etwas gewöhnungsbedürftig.
Bei dieser Strukturierung sprechen wir von **Generalisierung**. Damit ist die Verallgemeinerung – also die Entwicklung vom Speziellen zum Allgemeingültigen gemeint.
Die dem Pfeil entgegengesetzte (hier von oben nach unten) Lesung müsste dann eher mit "vererbt_an" erfolgen. Es handelt sich nun um eine **Spezialisierung**. Vom Allgemeinen kommen wir zu immer detaillierteren, spezielleren Teil-Elementen der Hierrarchie.



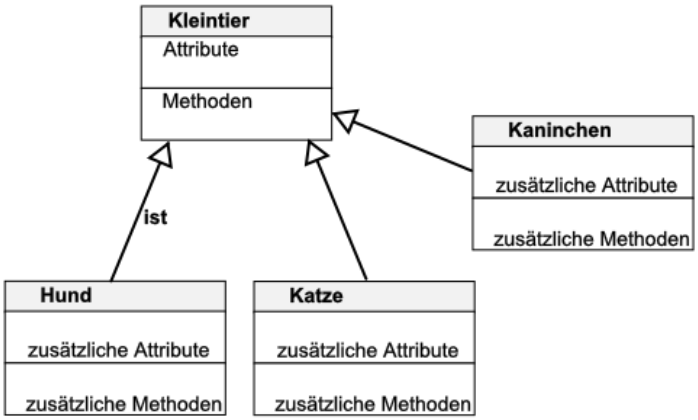
Für unser Haustier-Hunde-Katzen-Modell würde ein grobes UML-Diagramm dann so aussehen. Die graphische Über-Unterordnung ist nur für uns Menschen angenehmer. Sachlich spricht nichts dagegen UML-Diagramme auch anders zu zeichnen. Wichtig ist allerdings der richtige Einsatz des offenen Dreiecks / Pfeiles.
Gelesen wird das Diagramm dann kurz und knapp:



Hund ist(_ein) **Kleintier**
(**Hund** gehört_zu **Kleintier**.)
(**Hund** ist_ein_spezielles **Kleintier**.)

Katze ist(_ein) **Kleintier**

Die Beschreibung "Hund erbt_von Kleintier" bzw. "Katze erbt_von Kleintier" ist für das informatische Verständnis allerdings besser. Das menschliche Denken ist meiner Meinung auch eher so.
Meist beginnt man auch beim Programmieren mit den übergeordneten Klassen, um sie schrittweise zu verfeinern – also noch weitere Unter-Klassen zu erzeugen.
Der andere Nutzeffekt ist die schnellere Anwendung / Übertragung auf ähnliche Objekte, die auf der gleichen Hierrarchie-Ebene (Klassen-Ebene) liegen.
So läßt sich das Konzept auf weitere Kleintiere, wie z.B. Vögel, Kaninchen usw. problemlos erweitern.
Genau dieses Handling macht die Objekt-orientierte Programmierung so Leistungs-fähig.



Attribute, die aus übergeordneten Klassen – also den Superklassen – stammen und geerbt können über this. benutzt werden, da sie mit der Unterordnung der aktuellen Klasse unter eine vererbende Klasse, auf die erbende Klasse übertragen werden.

z.B. definiert in der Klasse Kleintier:

```
…
    String name = "unbekannt";
    Boolean essbar = false;
…
```

Vererbung und Nutzung in untergeordneter Klasse Kaninchen:

```
…
public class Kaninchen extends Kleintier {

    //Konstruktor
    public Kaninchen (String name) {
         this.name = name;       //hier this notwendig, damit keine
                                 // Verwechslung von name
         this.essbar = true;     //hier this nicht notwendig
    }
}

…
```

Beide Zuweisungen im Konstruktor greifen auf Attribute zu, die Kaninchen von Kleintier geerbt hat. Sie sollten nicht in Kaninchen neu deklariert werden, da es sonst doppelte Daten-Bestände gäbe. Einmal z.B. den Namen über die Klasse Kleintier und dann einen zweiten über die Klasse Kaninchen.

Beim Methoden-Aufruf wird zuerst immer in der aktuellen (speziellen) Unter-Klasse nach der entsprechenden Methode gesucht. Existiert keine passende Definition, wird in der übergeordneten Super-Klasse nach einer entsprechenden Methode gesucht. Findet das System auch hier keine passende Methode wird in der nächst höheren Super-Klasse weitergesucht. Man spricht von einer Aufruf-Kaskade. Als Nutzer braucht man nicht zu wissen, auf welcher Klassen-Ebene eine Methode definiert wurde, es reicht die Kenntnis über deren Existens. Trotzdem können die einzelnen Klassen auch gleichlautende Methoden besitzen. Prinzipiell wird dann die der aktuellen Klasse zugeordnete Methode genutzt, so wie wir es gerade dargestellt haben. Sollte aber ein Aufruf der eigentlich nicht zugeordneten Methode der Überklasse gewollt sein, dann muss der Aufruf mit dem Klassen-bezogenen Aufruf (Punkt-Notation) erfolgen.
(Bei Unkenntniss der unmittelbaren Oberklassen (sollte eigentlich nicht vorkommen!), kann man mit **super.** die "unbekannte" Ober-Klasse angesprochen werden. Bei Auftrags-Werken oder abstrakten Klassen können solche Unkenntnisse aber ohne weiteres auftreten.)

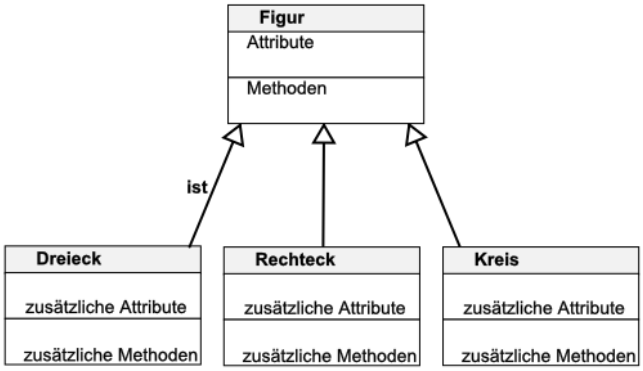
class *SuperklassenBezeichner* { …
    // Definition der Superklasse

class *SubklassenBezeichner* extends SuperklassenBezeichner { …
    // Defintion der Spezialitäten der Subklasse

Aufgaben:

1.

2. Gegeben ist nebenstehendes UML-Diagramm und Auszüge aus JAVA-Quelltext(en).
   Erweitern Sie das UML-Diagramm und den Quelltext um die folgenden Klassen!
   Quadrat, Sechseck, Viereck, Oval, Halbkreis, Rhombus




```
public class Figur { … }
…
public class Dreieck extends Figur{ … }
…
public class Rechteck extends Figur{ … }
…
public class Kreis extends Figur{ … }
```

3. Verbessern Sie das erweiterte UML-Diagramm! Passen Sie dann auch die Quelltexte an!

1.4.2.1. Überschreiben von Methoden (Override)

In de spezielleren Klassen (Unter-Klassen) kann es passieren, das definierte Methoden der Ober-Klasse nicht mehr passen. Deshalb kann man in einer untergeordneten Klasse immer die geerbte Methode (der Oberklasse) überschreiben. Die Methode wird auf der akteullen Ebene konkretisiert oder auch völlig neu gestaltet. Oft wird die "alte" Methode nur ergänzt. Ab der aktuellen Hierrarchie-Ebene gilt nun diese Methode, bis eine weitere Unter-Klasse auch diese wieder überschreibt.
Beim Überschreiben wird ein zusätzliches Steuer-Zeichen eingesetzt: **@Override**
Das **@Override** ist eine sogenannte Annotationten. Das bedeutet "Anmerkung", "Hinzufügung" oder "Ergänzung". In JAVA beginne Annotationen immer einem at-Zeichen (@).
Diese Notierung – also der Hinweis auf eine Überschreiben einer Methode – ist für andere Programmierer ein Hinweis. Dadurch wird für sie sichtbar, das man eine geerbte Methode verändert hat.
Vielfach werden einfach nur ergänzende Informationen gehändelt, die für die Sub-Klasse relevant sind.
Die geerbte Methode ist trotz dem Überschreiben nicht verloren gegangen. Da sie in der Ober-Klasse bzw. einer der übergeordneten Klassen definiert ist, kann sie mit Angabe des Klassen-Namens oder eben von super. als allegemeinen Obberklasse-Namen aufgerufen werden. Das ergibt ein effektives und flexibles System von Fähigkeiten, die ein Programm gut ausnutzen kann.
Das Abfragen der Superklasse zu einem Objekt (einer Instanz) ist auch über:

KlassenBezeichner.class.getSuperClass()

möglich. Der direkte Zugriff auf Attribute und Methoden der Superklasse wird am Besten über **super.**AttributBezeichner bzw. **super.**MethodenName() realisiert.

Overriding ist also charakterisiert durch:
- Verändern / Anpassen vorhandener / vererbter Methoden
- die "ursprüngliche" (geerbte) Methode und die "überschriebene", neue Methode liegen immer in unterschiedlichen Klassen
- die Methoden-Namen sind immer (in allen Klassen) gleich
- die Parameter(-Listen) müssen gleich sein
- die Rückgabe werden müssen gleich oder kompartibel sein
- die "ursprüngliche" Sichtbarkeit darf durch die neue Methode nicht eingeschränkt werden.

Vielfach wird das Overriding mit der Polymorphie (→) verwechselt. Diese stellt aber ein anderes Prinzip der Objekt-orientierten Programmierung dar.
Das Überschreiben benötigt man recht häufig bei der internen Object-Methode **equals()**. Man könnte ja denken, sie vergleicht zwei Objekte auf Gleichheit. Was sie genau macht ist der absolute Vergleich, d.h. sie prüft ob es sich exakt um das gleiche Objekt handelt. Und wir wissen ja, dass ein Objekt eigentlich nur eine Speicher-Adresse / ein Zeiger auf andere Speicherbereiche (z.B. mit den Attributen) ist. Genau diese Zeiger / Adressen werden verglichen.
Wir brauchen eher einen Vergleich auf die Übereinstimmung von charakteristischen oder allen Attributen der beiden Objekte. Das kann das JAVA-System aber nicht wissen. Also müssen wir das selbst programmieren. Damit überschreiben wir die ursprüngliche Adress-Vergleichs-Methode durch eine eigene Attributs-Vergleich-Methode.

```
…
public class KlassenBezeichner {
  private Typ Attribut1;   //z.B. komplexer Datentyp, wie z.B. String
  private Typ Attribut2;

…

  @Override
  public boolean equals(Object obj){  //Vergleich auf Gleichheit
                                      //der beiden Attribute
    return ((obj != null)    // existiert das andere Objekt überhaupt?
           && (this.Attribut1.equals(obj.Attribut1)
           && (this.Attribut2 == obj.Attribut2);
  }
}
```

Bei Attribut1 wurde eine Type-Casting (→ 1.3.7.1. Typ-Umwandlungen (Type Casting und Parsing)) durchgeführt, weil z.B. Attribut1 hier Besipiel-haft ein komplexer Datentyp sein soll. Hier greift der einfache Vergleich über **==** nicht.

1.4.2.1.1. sicheres Vergleichen

Anwendung bei der eigenen Impelementierung (Überschreiben (→ @Override)) der equals-Methode. Hier muss nämlich sichergestellt werden, dass man den Vergleich auf äquivalente Objekte durchführt. Ein ausschließlicher Vergleich bestimmter Attribute ist nicht immer Aussage-kräftig. Eine solche überschriebene Methode könnte dann so aussehen:

```
…
@Override
public boolean equals(Object obj){
  if (obj == null) {  //existiert anderes Obj.
    return false
  }
  if (this == obj) {  //absolute Gleichheit
    return true
  }
  if (this.getClass() != obj.getClass()) { //unterschiedliche Klassen
    return false;
  }
  //jetzt Feinabgleich bestimmter Attribute
  …
…
```

alternativ (ev. problematisch!):

```
if (!(obj instanceof ObjektTyp))
   return true;
}
```

1.4.3. Konstruktoren

Der **Kontruktor** ist die Initialisierungs-Methode einer Instanz (eines Objektes). Dabei wird die Klassen-Definition als Blaupause (Kopier-Vorlage) benutzt, um ein Programm-internes Objekt (eine Instanz) zu erstellen. Vielfach werden hier die Vorbelegungen von Attributen vorgenommen. Ein Konstruktor liefert immer ein Objekt seiner Klasse zurück.
Das dazugehörige Schlüsselwort ist **new**. Die Struktur einer Instanzierung ist in JAVA erscheint – besonders Anfängern – etwas überdimensioniert:

InstanzTyp *InstanzBezeichner* = new ErzeugerTyp;

oder

InstanzTyp *InstanzBezeichner* = new ErzeugerTyp();

Der letzte Aufruf steht für den Aufruf einer initialen Methode beim / zum Erstellen des Objektes (der Instanz). Hier werden üblicherweise spezielle individualisierte Attribute gesetzt. Außerdem setzt man vielfach an dieser Stelle (im Konstruktor) die Standard-Werte.
Der InstanzTyp muss in der Klassen-Hierrarchie höher oder gleichrangig eingestuft sein, als / wie der Erzeugertyp.
In der Klasse hat der Konstruktor den gleichen Namen, wie die Klasse, gefolgt - möglicherweise – von einer geklammerten Liste mit Parametern, die für die Initialisierung genutzt werden sollen.

```
class KlassenBezeichner{

        KlassenBezeichner(ParameterListe){    //kein Typ notwendig
                //InitialisierungsAnweisungen
        }
}

class KlassenBezeichner{
        Typ AttributBezeichner = Vorbelegung;

        KlassenBezeichner(ParameterListe){    //kein Typ notwendig; ohne Parameter
                //InitialisierungsAnweisungen
                this.AttributBezeichner=übergebenerParameter;
        }
}
```

Sind die AttributBezeichner (der Klasse) und die Parameter-Variablen gleichnamig, dann muss für den eindeutigen Zugriff auf die Klassen-Variablen das Schlüsselwort this. verwendet werden. Hinter dem Punkt folgt dann das Klassen-Attribut

Definiert man keinen Konstruktor, dann existiert automatisch ein sogenannter Default-Konstruktor, der allerdings auch nichts initial macht, außer das Objekt (die Instanz) als solche anzulegen.
Dieser Default-Konstruktor entspricht dem folgenden Code. Er muss aber nicht explizit geschreiben werden, da JAVA ihn automatisch vorsieht.

```
KlassenBezeichner(){
}
```

Es ist aber immer günstiger und anzustrebender Stil immer einen eigenen Konstruktor zu schreiben, auch wenn dieser (zuerst noch) leer ist. Später wird man dann vielleicht doch ei-

nen brauchen, dann sind die passenden Positionen schon vorhanden. Außerdem signalisert mna einem anderen Programmierer, der mit unserer Klasse arbeiten soll, dass es sich eben (noch) um eine einfache leere Konstruktion handelt.
Um auf die eigenen Attribute oder Methoden zurückgreifen zu können, gibt es das Schlüsselwörtchen **this**, was die eigene Instanz meint.
Gibt man **this** nicht mit an, dann bedeutet es dann, dass das Attribut für die gesamte Klasse gilt. Z.B. wird so in der Konstruktor-Methode die Anzahl der Instanzen hochgezählt (, wenn es denn gebraucht und gewollt wird).

Definition(en): Konstruktor
Ein Konstruktor ist eine spezielle Methode, die bei Erstellen einer Instanz eines Klassen-Objektes aufgerufen wird.

Desweiteren gibt es die Möglichkeit mit einer **this()**-Methode – das ist ein zusätzlicher Konstruktor – indirekt dann auch wieder den anderen "normalen" Konstruktor aufzurufen.

z.B. es existiert ein Konstruktor für die Klasse: Hund

```
…
public Hund(String name, int alter, String rasse){
    this.name=name;
    this.alter=alter;
    this.rasse=rasse;
    // restliche Block-Anweisungen
    …
}
…
```

dieser setzt bei Instanzieren die "Grund"-Werte.
Nun könnte es aber auch sein, dass ein unbekannter Hund verwaltet werden soll. Wir wissen nur, dass er existiert, aber haben keine Angaben. Dann könnten Default-Werte gesetzt werden. Hier ist das "kein Name", "999" und "Mischling". Der gesamte Code-Abschnitt für die Konstruktoren sehe dann so aus:

```
…
public Hund(){
    this("kein Name,999,"Mischling");  //Default-Setzung,
                                       //muss 1. Blockzeile sein!
    // restliche Block-Anweisungen
}

public Hund(String name, int alter, String rasse){
    this.name=name;
    this.alter=alter;
    this.rasse=rasse;
    // restliche Block-Anweisungen
}
…
```

Es handelt sich um ein Überladen (→ 1.6.3. Überladen von Methoden (Overload(ing))) der Konstruktor Methode.
Will man den Konstruktor der übergeordneten Klasse aufrufen (mehr dazu bei → Polymophie), dann benöigt man die **super()**-Methode. Mehr dazu bei der Besprechung der Polymophie (→ 1.6.5. Polymorphie).

```
…
public Hund(String name, int alter, String rasse){
    super(name);      // Aufruf des Konstruktors
                      // der übergeordneten Klasse (z.B.: Tier)
    this.alter=alter;
    …
}
…
```

Im Beispiel wird der übergeordnete Konstruktor aufgerufen. Dieser enthält – weil er ja üblicherweise einfacher ist – nur einen Namen (zumindestens als Argument / Parameter).

Mit **static** (als vorangestellten Modifikator) kann man ausgewählte Methoden in einer Klasse definieren, die unabhängig von einer Objekt-Instanzierung aufgerufen werden können. Praktisch sind das oft allgemeine Methoden, die garnichts mit einem Objekt an sich zu tun haben, sondern irgendwelche kleinen Aufgaben erledigen. Typische Aufgaben sind z.B. eine bestimmte formatierte Ausgabe zu erzeugen oder die eine physikalische Größe in eine andere Darstellung oder mit einer anderen Einheit auszugeben.
Innerhalb einer statischen Methode kann man die (nicht-statischen) Attribute der Klasse nicht lesen oder schreiben. Ein this.-Bezug ist ebenfalls nicht möglich.

Beispiele für statische Methoden aus JAVA selbst sind die Math-Klasse und die Wrapper-Klassen zum Typ-Konvertieren.

Die nicht-statischen Methoden einer Klasse können aber die static-Attribute lesen und schreiben.

1.4.3.1. kleine Zusammenfassungen und Übersichten

Vergleich von this() und super()

	this()	**super()**
Funktion	ruft den überladenen Konstruktor der eigenen Klasse auf	ruft den Konstruktor der übergeordneten Klasse auf
Hierrarchie	"geringer Rang"	"höherer Rang"
Position im Anweisungs-Block	muss die 1. Anweisung im Konstruktor sein	
Aufrufbarkeit	nur innerhalb von Konstruktoren	
Benutzbarkeit	es kann entweder this() ODER super() verwendet werden, nicht beide gleichzeitig!	

Vergleich von this. und super.

	this.	**super.**
Funktion	ermöglicht den Zugriff auf die Attribute und Methoden der **eigenen** Klasse (ohne diese direkt zu benennen)	ermöglicht den Zugriff auf die Attribute und Methoden der **übergeordneten** Klasse (ohne diese direkt zu benennen)

1.4.4. Pakete (Package)

in Ordner zusammengefasste JAVA-Dateien, die zu einem Projekt / Projekt-Teil gehören
Ordner muss den Namen des Package tragen und darin die Klassen-Dateien mit dem Klassen-Namen.

Definition(en): Package
Ein Package ist eine Zusammenstellung von Klassen (classes), Schnittstellen (interface's) und Ausnahmen (exception's) unter einem Namen (/ in einem Namensraum).
Ein Package ist ein Namensraum in dem Klassen, Schnittstellen und Ausnahme-Behandlungen organisatorisch zusammengefasst werden.
Ein Package ist eine organisatorische und funktionelle Einheit von zusammengehörenden Klassen und Schnittstellen.

Benennung erfolgt in umgekehrter Domänen-Angabe. Das leitet sich aus der Objekt-Strukur ab. So ist ein Package von www.dieJAVAprogrammierer.de mit:

de.dieJAVAprogrammierer

zu benennen.

1.4.5. Kapselung

Ein Programmierer hat sich villeicht entschieden in seiner Geld-Klasse Konten usw. mit gespeicherten Ganzzahlen zu verwalten – also praktisch Euro-Cent-Konten. Er fand das sehr günstig hinsichtlich der einfachen und unkomplizierten Rechnungen (auch für den Prozessor) und mit Rundungs-Problemen hat er ebenfalls erst einmal nicht zu rechnen.
Nun kann er aber den Nutzern seines Programms und auch anderen Nutzern seiner Geld-Klasse nicht zumuten auch in Cent zu arbeiten. Jeder andere Nutzer erwartet sicher einen typischen Euro-Betrag mit maximal zwei Nachkommastellen. Nutzer und Programmierer haben hier scheinbar ein Problem miteinander. Das löst sich aber in der Objekt-orientierten Programmierung ganz schnell auf, weil es die sogenannte **Kapselung** (engl.: encapsulation) gibt. Das Innen und Außen um eine Klasse läuft eigenständig nebeneinander. Der Programmierer der Geld-Klasse muss allerdings Methoden zur Daten-Eingang in seine Klasse und für den Daten-Ausgang bereitstellen. Diese Kontaktstellen oder auch **Schnittstellen** sind exakt definiert und die einzigen Zu- und Abgänge für den Daten-Verkehr. Schnitstellen sind öffentliche vereinbarungen / Bekanntmachungen für die Kommunikation zwischen verschiedenen Programmteilen, Software-Schichten, Klassen usw. usf.
In den Objekt-orientierten Programmiersprachen gibt es dafür die GET-SET-Konstruktionen. Mit GET (get, Geter, …) wid ein Wert (Attribut) von außen gelesen. Mit der SET-Konstruktion (set, Seter) schreibt man einen Wert in die Klasse (/ das Attribut). Der direkte Lese- und Schreib-Zugriff ist nicht möglich. Dazu wird bei den Attributen die Sichtbarkeit auf private gesetzt. Das gilt auch für viele Geld-Methoden. Nur die **Get**er und **Set**er sind für andere Klassen sichtbar (public).
Wer mehr auf deutsche Methoden-Namen steht, nimmt eben **gib**- und **setz**-Methoden

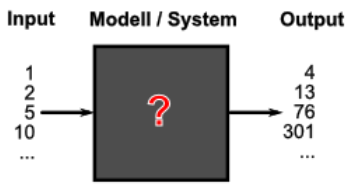
```
…
private int Attribut;   //Attribut nur für den internen Gebrauch in
der Klasse

   //notwendige GET- bzw. SET-Methoden für Lesen u. Schreiben
   public int getAttribut(){       //Lese-Methode
   return Attribut;
   }

   public void setAttribut(int Parameter){      //Schreib-Methode
   this.Attribut=Parameter;
   }

…
```

Definiert man nur eine GET-Methode, dann ist das Attribut von außen im NUR-LESE-Modus. Es kann von extern nicht geschreiben / verändert werden. Bei Bedarf sind Änderungen durch Klassen-interne Methoden aber immer möglich.

Definition(en): Kapselung
Kalselung ist eine Technik (der (Objekt-orientierten) Programmierung), Daten – deren Strukturen und Verarbeitung – vor anderen Programmierern und Nutzer zu verbergen oder auch zu schützen.

Information-Hiding; Abstraktion

Ein gutes Anwendungs-Beispiel ist auch die Ver- und Entschlüsselung mittels geheimer Chiffren. Wie das genau funktioniert – das Verschlüsselungs-Verfahren – soll u.U. auch geheim bleiben. Natürlich müssen die zu verschlüsselnden Information irgendwie rein und die Geheimtext raus können. Das übernimmt dann eben eine spezielle Schnittstelle. Eine gleichwertige zweite Schnittstelle gibt es dann für die Geheimnachricht (rein) und das Ausgeben der entschlüsselten Information (raus).
Die Kenntnis der gesamten Verschlüsselungs-Methode aber ev. auch schon einzelner Teil-Methoden – wird durch die Kapselung von außen unmöglich. Man denke z.B. an eine Methode, die Texte umdreht (reverse).
Wenn ein fremder Geheimdienst davon Kenntnis erlangt, dann kann er davon ausgehen, dass irgendwo im Verschlüsselungs-verfahren eine Umkehrung genutzt wird. Wieder ein Schritt näher ander der Lösung (Entschlüsselung der Nachricht bzw. dem Knacken des Verfahrens).
Am Beispiel der Verschlüsselungs-Klasse wird auch das Black-Box-Modell der (Daten-)Kapselung deutlich. Was in der Box passiert ist uns nicht zugänglich. Wir können die Box nur anhand der Eingänge und Ausgänge bewerten.



Exkurs: Verlust des Mars Climate Orbiters

Am 23.09.1999 stürzte der "Mars Climate Orbiter" bei der Landung auf dem Mars ab, weil sein Landepunkt 170 km zu tief einegstellt war.
Die Fehler-Analyse ergab, dass zwei NASA-Teams an dem Projekt arbeiteten. Das eine Team arbeite, wie im wisschenschaftlich-technischen Bereich üblich in den metrischen Einheiten. Das andere Team arbeitete – ganz amerikanisch – im angloamerikanischen customary-System. (Übrigens sind die USA eines von drei Länder auf der Erde, die sich noch nicht zum metrische System durchringen konnten.)
Konkret ging es um die Länge in Meter bzw. yard. Ein Meter sind 1,094 yard. Bei der Zahlen-Übergabe wurden die verschiedenen "Einheiten" nicht beachtet und die Katastrophe nahm ihren Lauf.

???
mit super.Methoden bzw. super.Attribute greift man auf die definierten Eigenschaften der übergeordneten Eltern-Klasse zu. Dazu muss der Programmierer nicht wissen, wie diese Klasse heißt (obwohl es ja vorher im Quelltext steht). Diese Notierung ist besonders dann interessant, wenn Methoden in verschiedenen Projekten / Package / Klassen immer wieder verwendet werden sollen, man aber nicht jedesmal die notwendige Oberklasse wissen und ändern möchte.

mit this.Attribute greift am auf die Attribute von einzelnen Objekten (Instanzen) der eigenen Klasse zu
Vielleicht hat ja jedes Objekt einen eigenen Namen, den man ev. dann weiter verwenden möchte.
beziehen sich auf die Attribute de eigenen Klasse

Sichtbarkeit von Elementen (Attributen bzw. Methoden) in anderen Klassen

Sichtbarkeit-Schlüssel-wort (Modifikator)	Umschrei-bung	Zeichen im UML-Diag.	Sichtbarkeit von Elementen						Vererbung			
			Umschreibung	innerhalb der Klasse	innerhalb des Package	in den Subklassen	in sonstigen Klassen		Umschrei-bung	Attribute	Methoden	
default			ähnlich public; wenn keine Angaben zur Sichtbarkeit gemacht werden, dann ist automatische (per default) public eingestellt	✓	✓	✗	✗		nur innerhalb des Package	?	?	
public	öffentlich	**+**	uneingeschränkter Zugriff auf Methoden und Attribute von allen anderen Klassen möglich	✓	✓	✓	✓	sollte nicht für Attribute verwendet werden	vollständig	✓	✓	
protected	geschützt	**#**	Zugriff nur in der eigenen Klasse und in den Unterklassen (Subklassen) möglich	✓	✓	✓	✗		vollständig	✓	✓	
private	privat	**-**	Zugriff nur innerhalb der eigenen Klasse möglich (keine Sichtbarkeit in über- od. untergeordneten Klassen)	✓	✗	✗	✗	Zugriff kann / kann über GET-SET-Methoden realisiert werden ermöglicht z.B. auch den Zugriff auf Attribute anderer Instanzen der gleichen Klasse	keine	✗	✗	

(c,p) 2017-2022 lern-soft-projekt: drews

1.4.6. Daten-Übergaben - Argumente und Parameter

Methoden können neben den Klassen eigenen Attributen auch noch "externe" Daten benötigen. Wie wir es von mathematischen Funktionen kenne, werden die Arbeits-Daten / Bearbeitungs-Objekte in Klammern hinter den Funktionsnamen geschrieben.
Aber auch aus der Tabellenkalkulation ist uns dieses Konzept noch gegenwärtig. Fast alle Tabellenblatt-Funktionen benötigten in den Klammern irgendwelche Werte, Zell(-Adress)en oder Zellbereiche (Matrizen). Allgemein bezeichnet man die mitgegebenen Werte an die Funktion / Methode als **Argumente**.

Beispiele allg.:	aus Tabellenkalkulation:
sin(x)	SIN(B4)
log(x)	LOG(E7)
	WENN(G5>3;G4;"???")

Manchmal sind die Funktions- / Methoden-Namen auch durch allgemeine Symbole ersetzt. Wurzel, Summe und Produkt sind sicher gute Bespiele dafür.

Beispiele allg.:	aus Tabellenkalkulation:
$\sqrt{a+24}$	WURZEL(D28)
$\sum_1^{13} x^3$	SUMME(D3:D15)
$\prod_{i=1}^{4} x^2$	PRODUKT($F2:$F12)

Definition(en): Argumente
Argumente sind Übergabe-Werte / Mitteilungen, die innerhalb einer Methode weiter verarbeitet werden (können).
Argumente sind Daten, die beim Aufruf einer Subroutine (Unterprogrammen, Funktionen, Prozeduren, Methoden) übergeben werden. Sie werden dann in den Methoden intern (als übernommene Parameter) genutzt.

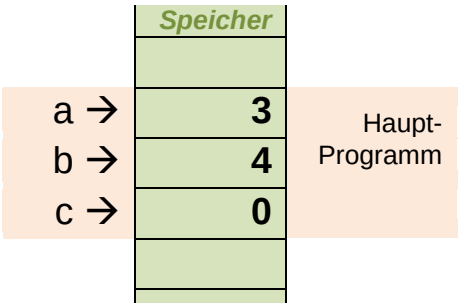
Der Aufruf einer Methode besteht also aus drei wesentlichen Teilen. Da ist zuerst das aufrufende / besitzende **Objekt** (die Instanz). Hinter dem Punkt folgt der **Methoden-Name** und dann in runden Klammern die **Argumente**.

```
..
Instanz.Methodenname(Argumentliste);    //Methoden-Aufruf
..
```

Die zu benutzenden Funktionen / Methoden müssen wir als Programmierer in unseren Klasse selbst implementieren.
Bei der Definition einer Methode / Funktion mit zu benutzenden / übergebenen Informationen werden diese bestimmten Daten-Typen und internen Bezeichnern zugeordnet. Das passiert gleich in Anschluss an den Methoden-Namen in runde Klammern. Die (internen, lokalen) Bezeichner gelten nur innerhalb der Methode (bis zur schließenden geschweiften Klammer der Methoden-Definition).

RückgabeDatenTyp *Methodenname*(Parameterliste){ //Methoden-Definition
Methodenkörper
}

Die Daten-Liste, die unsere zu schreibende Methode vom aufrufenden Programm mitbekommt, wird Parameter-Liste genannt. Die einzelnen Parameter müssen zu den Argumenten passen. Methoden-Name und die Parameter-Liste bezeichnen wir auch als **Methoden-Signatur** (→ 1.6.3. Überladen von Methoden (Overload(ing)).
Oft wird das durch gleiche Bezeichner angedeutet. Das muss aber nicht so sein, die internen Bezeichner haben keinen begrifflichen Bezug zu den Bezeichnern des aufrufenden Programms.
Das wird deutlich, wenn man sich das Handling im Speicher anschaut:
Nehmen wir an, aktuell im Speicher sind drei Variablen angelegt. Sie sind mit a, b und c deklariert worden. Die aktuelle Belegung der Speicherzellen ist 3 und 4 und 0. Die Variablen werden derzeit vom Hauptprogramm verwaltet.



Im nächsten Schritt soll im Hauptprogramm eine Methode aufgerufen werden, die beide Zahlen multipliziert und dann halbiert zurückliefert.Der Name der Methode sei halbProdukt(). Das spielt aber bei der Belegung des Speichers keine Rolle.

```
…
int halbProdukt(int a, int b){
    int c = 2;
    c = a * b / c;
    return c;
}
…
```

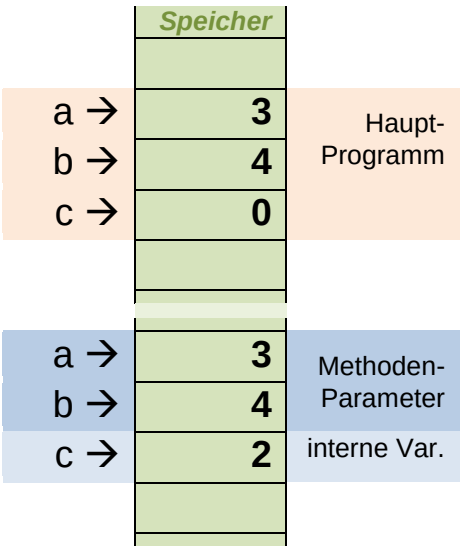
es ginge auch die Kurz-Version:

```
int halbProdukt(int a, int b){
    return a * b / 2;
}
```

Die Anweisung könnte:

```
c = halbProdukt(a,b);
```

lauten.
Mit dem Aufruf der Methode **halbProdukt(a,b)** werden im Speicher die beiden Speicherzellen für die Parameter **a** und **b** (aus der Parameter-Liste) erzeugt und die Daten aus den Argumenten hineinkopiert.
Mit diesen Variablen kann jetzt die Mthode machen, was sie will / soll. Ein Ändern hier hat keine Auswirkungen auf die Speicherzellen des Haupt-Programms.
Als nächstes legt die **halbProdukt()**-Methode die Variable c an und belegt sie mit dem Wert 2 vor. Auch dieses c hat nichts mit dem c aus dem aufrufenden Programm zu tun, die Variable heißt zufällig nur gleich.



Das jetzt durch die Berechnungen in der Methode **halbProdukt()** die Variable geändert wird und das auch mit dem gleichzeitigen Auftreten einer Variable auf der rechten und linken Seite einer Zuweisung funktioniert, haben wir schon erklärt (→ 1.3.4. Variablen).

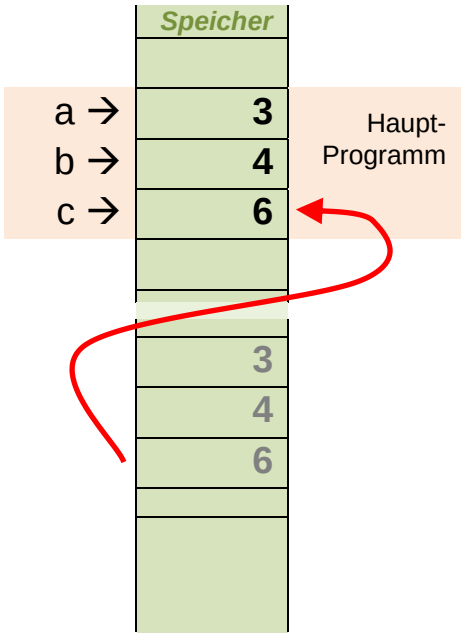
Interessant ist nun das Vorgehen zum Abschluss der Methode. Return c bewirkt ein zurückliefern an die aufrufende Stelle.

```
c = |halbProdukt(a,b);
    ^
```

Der Wert aus der Methoden-Variable c – also die 6 – wird jetzt in die Speicherzelle des Hauptprogramm-c übertragen.

```
c = |6;
    ^
```

Der Rückgabewert wird also zugewiesen. Mit dem return wird nun auch die gesamte Speicherstruktur gelöscht bzw. die Referenzen (Zeiger) gekappt. Die Parameter-Variablen sind absolut nicht mehr zugänglich!



Aufgaben:

1. ***Skizzieren Sie sich einen Speicher-Plan für den Fall, dass der Methoden-Aufruf halbProdukt(b, a) geheißen hätte!***
2. ***Überlegen Sie sich, ob auch der Aufruf halbProdukt(a, a) möglich / zulässig ist! Wenn JA, dann skizzieren Sie den passenden Speicher-Plan, sonst (bei NEIN) begründen Sie, warum das nicht geht / funktioniert!***
3. ***Welche Werte haben die Variablen x, y und z nach dem Durchlauf des folgendenCode-Schnipsels?***

```
…
int x = 5;
int y = 3;
int z = x;
x = doppelSumme(y, 1);
x = doppelSumme(x, z);
…

…
public int doppelSumme(int z, int x){
    return 2 * (z + x);
…
```

Die Parameter müssen also als Aufzählung (Komma-getrennt) in die (runden) Klammern der Methoden-Definition notiert werden. Dabei sollte immer eine möglichst logischer Reihenfolge verwendet werden. Selten oder weniger benutzte Parameter / Argumente sollten weiter hinten stehen. Das bringt bei der Erweiterung des eigenen Programms um andere Methoden-Varianten und optionale Argumente / Parameter sowie neuen Datentypen gewisse Vorteile (→ 1.6.5. Polymorphie).
Parametrisierte Methoden müssen vom aufrufenden Programm immer mit Argumenten aufgerufen werden.
Die Aufrufe:

```
c = halbProdukt(2);
```

oder

```
d = doppelSumme();
```

würden schon beim Kompilieren Fehler erzeugen.
Es sollen ja u.U. auch wieder Daten zurück an das aufrufende Programm zurückgeliefert werden. Den Daten-Typ des Rückgabe-Wertes müssen wir schon vor dem Methoden-Namen angeben. Die eigentliche Rückgabe erfolgt hinter dem Schlüsselwort **return**. Das ist dann auch der konkrete Rücksprung in das aufrufende Programm. Dort wird mit der nächsten Anweisung oder der weiteren Auswertung der aktuellen Anweisung fortgesetzt.
Sollen keine Daten zurückgegeben werden dann muss das Schlüsselwörtchen **void** (für "ohne Typ") angegeben werden.

Definition(en): Parameter
Parameter sind die Werte / Mitteilungen, die eine Methode als zu übergebende Informationen erwartet und intern weiter verarbeitet.
Parameter sind spezielle Variablen, die innerhalb von Subroutinen (Unterprogrammen, Funktionen, Prozeduren, Methoden) benutzt werden, um auf Daten zuzugreifen, die von den übergeordneten Programmteilen (beim Subroutinen-Aufruf als Argumente) übergeben werden.

im Anwendungs-Code bedarf es der richtigen Anzahl und der richtigen Typ-Zuordnung der Argumente zu den definierten Parametern
es können direkt Werte, Terme oder Varaiblen als Argumente benutzt werden
die Parameter können immer nur Bezeichner sein
die Namen / Bezeichner von Variablen im aufrufenden Code und solchen im Methoden-Code können gleich sein (sie haben praktisch nur die Argument-Parameter-Verknüpfung gemeinsam)
die Variable bzw. der Parameter gilt nur innerhalb der Methode (bis zur schließenden geschweiften Klammer der Methode-Definition)

Bsp. Methode die eine physikalische Größe mit Wert und Einheit ordnungsgemäß darstellt vielleicht mit unterschiedlichen Farben für die einzelnen Bestandteile

```
Parameterliste ::= ""
Parameter ::= Typ Parametername
Parameterliste ::= Parameter {, Parameter}
Argumentliste ::= ""
Argument ::= Wert | Term | Bezeichner
Argumentliste ::= Argument {, Argument}

..
Instanz.Methode(Argumentliste);    //Methoden-Aufruf
..
```

Bei Konstruktoren (Initialisierungs-Methoden) werden häufig die Parameter zur Individualisierung der Instanzen benutzt. Sie erhalten z.B. bestimmte Namen, Nummern und / oder spezielle Eigenschaften.

1.4.7. Gültigkeit und Sichtbarkeit von Variablen – Scope

? Seiten-Effekte in JAVA

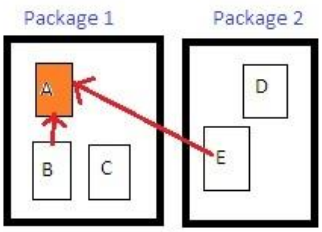
Schlüsselwörter gelten innerhalb des gesamten Quellcodes

Werden Variablen innerhalb einer Methode definiert, dann bewirkt das eine lokale Gültigkeit innerhalb der Methode
die Variable wird mit der schließenden, geschweiften Klammer der Methode zerstört (Garbage collection (Speicher-Bereinigung))

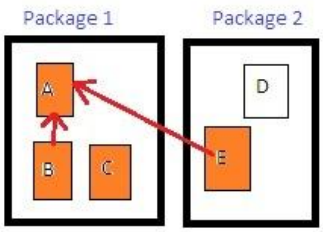
Da in größeren JAVA-Projekten aufgrund der Klassen-Strukturen mit mehreren Dateien und eben Klassen gearbeitet wird, kann man sich bei der Sichbarkeit von Variablen mit Modifikatoren behelfen. Mit dem Modifikator **public** werden Variablen öffentlich – auch in anderen untergeordneten Klassen nutzbar.
Dagegen kann mit **private** festlegen, dass eine Variable nur immerhalb eines Blocks (z.B. einer Klassen-Definition) gültig ist. Von außerhalb (dieses Blocks) ist ein Zugriff auf die Variable nicht möglich (→ Zerstörung mit schließender geschweifter (Block-)Klammer)

siehe auch (→ 1.4.2. Kapselung)

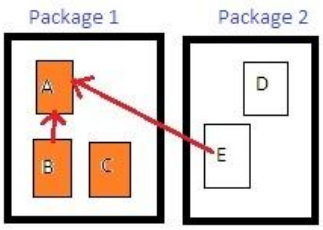
!!! überarbeiten / neu zeichnen und dann entfernen!



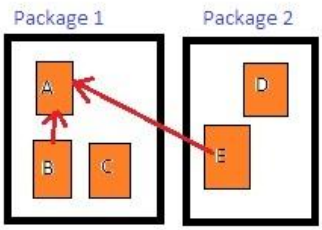
PRIVATE



PROTECTED



DEFAULT



PUBLIC

1.4.8. Warum also?: public static void main(String[ ] args) { … }

Mit dem Modifikator **public** legt man die – von außen mögliche – Zugreifbarkeit auf das nachfolgende Methode **main** fest.. Dadurch wird es möglich, die in den geschweiften Klammern enthaltenen Anweisungen überhaupt auszuführen.
Durch **static** wird festgelegt, das es sich hier bei **main** um eine Methode handelt, die nicht durch ein übergeordnetes Programm (in JAVA würden wir eine übergeordnete Klasse (/ eine Superklasse) meinen) überschrieben werden kann. Unsere main-Methode ist also der einzige und immer gültige Start-Punkt unseres Programms.
Der "Nicht-Datentyp" **void** besagt, dass die folgende Methode keinen Rückgabewert hat. In JAVA ist das per Definition so. Hier hat die **main**-Methode immer keinen Rückgabewert.
Die Start-Methode ist für JAVA immer mit dem Namen **main** festgelegt.
Unserer main-Methode wird standard-mäßig ein Feld (Array) mit dem Namen **args** übergeben. Darauf weisen die eckigen Klammern (**[ ]**) hin. Die in dem Feld gespeicherten Daten sind vom Typ **String** – also Texte. Die **main**-Methode kann also ein Feld von Texten entgegen nehmen. Man spricht auch von Argumenten, die der Methode mit auf den Weg migegeben werden können.
In der Praxis sind solche Argumente die sogenannten Kommandozeile-Argumente oder Programm-Optionen (fälschlicherweise auch Kommandozeilen-Parameter genannt).
Wer schon häufiger auf der Kommandozeile ("MS-DOS Eingabeaufforderung", "cmd" oder Konsole) gearbeitet hat, der weiss, dass man z.B. mit " /h" oder " /?" hinter dem Programmnamen eine kleine Hilfe zum Programm erreichen kann. Diese oder ähnliche Optionen / Zusatzinformationen sind z.B. eben solche Argumente, die vom Programm ausgewertet werden. Ein gut programmiertes Programm zeigt dann z.B. einige Hilfe-Zeilen an.
Anderen Programmen muss ein Datei-Name übergeben werden, damit das Programm von dort ev. die notwendigen Daten bekommt, oder die berechneten Daten dorthin speichert.
Bei den Argumenten gibt es also relativ viele Möglichkeiten.
In einem Programm lassen sich die auf der Kommandozeile angegebenen Daten über das Feld **args** nutzen. Das erste Argument ist in args[0] gespeichert, das zweite in args[1] usw. usf.
In den geschweiften Klammern (**{ … }**) folgen dann die Anweisungen der **main**-Methode – also alles dass, was unser Programm machen soll. Die Anweisungen sollen durch die drei Punkte angedeutet sein. Allgemein handelt es sich um einen Code-Block, der Semikolon-getrennt, mehrere Anweisungen enthalten kann.

1.5. weitere Schritte für Erfahrene

Iregndwann stoßen wir mit unseren einfachen Programmier-Techniken an die Grenzen. Praktisch lassen sich zwar schon die meisten Probleme lösen, aber der Aufwand ist z.T. unverhältnismäßig hoch. Die höher entwickelten Programmiersprachen – wie z.B. JAVA – bieten den Programmierern viele zusätzliche Techniken und Strukuren. Auf diese gehen wir jetzt ein. Dabei gehen wir auch davon aus, dass die Grundlagen der Objekt-orientierten Programmierung verstanden wurden, denn die sind in JAVA System-immanent. JAVA ist quasi eine OOP per se. Manchmal scheint es so, als würde JAVA einwenig über das Ziel hinausschießen, aber dass ist vor allem ein Anfänger-Blick. Bei immer größer werdenden Kenntnis von JAVA werden die Konzepte immer klarer und logischer.

1.5.1. Schleifen für Erfahrene

1.5.1.1. Manipulation von Schleifen(-Durchläufen)

Die nun vorgestellten Schlüsselwörter machen die Programmierung von Schleifen einfacher, gehören aber eigentlich zu den unschönen (verpöhnten) Programmier-Techniken. (Gerade wegen dem so ansatzweisen Spagetti-ähnlichen Programm-Verläufen hat man früher solche leichten Programmiersprachen wie BASIC abgelehnt.)
Für die Schlüsselwörter continue, break und return innerhalb von Schleifen spricht meist eine bessere Übersichtlichkeit und Verständlichkeit des Codes. Das Zurückbleiben von Daten-Müll im Speicher ist bei diesen Manipilations-Techniken aber nicht ausgeschlossen.

continue;
Abbruch des aktuellen Schleifen-Durchlaufs und Fortsetzung der Schleife mit dem nächsten Durchlauf.

break;
Verlassen des aktuellen Schleifen-Blocks
Schleife wird verlassen und mit der Anweisung hinter der abgebrochenen Schleife fortgsetzt (Wenn dies eine äußere Schleife ist wird diese ausgeführt und dann wahrscheinlich die innere Schleife (die beim letzten Durchlauf der äußeren Schleife ja abgebrochen wurde) neu gestartet.)

return;
Verlassen der gesamten (aktuellen) Methode

1.5.1.1. Iteratoren

über eine Kollektion (Daten-Struktur) können mehrere Iteratoren rüberlaufen
praktisch die Index-Zeiger auf die Elemente der Kollektion

geben jeweils 1 Element zurück und wandern dann automatisch zum nächsten Element in der Kollektion
nur ein Durchwandern in die Vorwärts-Richtung (interne Zeiger-Richtung) möglich

Interator *ElementTyp Variable* = *Kollektionsname*.interator();

```
ArrayList<String> zug = new ArrayList<String>();
…
Interator<String> bestandteil = zug.interator();
```

nutzbar mit praktischen allen Kollektionen
Iteratoren werden intern von foreach-Schleifen genutzt
sind im Interface Iterable<T> implementiert:

```
public interface Iterable<T> {
   Interator<T> interator();
}
```

dadurch ist die Iterator-Technik auch bei eigenen Kollektionen nutzbar, ohne das dafür eigener Code erstellt werden muss

Methoden:
boolean nochFolgende = bestandteil**.hasNext()**;
zur Bestimmung, ob noch Elemente in der Kollektion sind (also, ob nicht das Ende der Daten-Liste erreicht ist)

String element = bestandteil**.next()**;
gibt Element zurück und wandert zum nächsten Element

next() sollte nur in Verbindung mit hasnext() benutzt werden, damit keine Exception ausgelöst wird
sowohl der Rückgabe-Wert von hasnext() und next() sollten (für eine mehrfache Nutzung) in einer Variable gespeichert werden
(jeder erneute Aufruf von next() bewirkt weiterwandern des Iterator's)

bestandteil**.remove()**;
entfernt das mit next() abgfragte Element aus der Kollektion
Methode / Funktion hat keinen Rückgabe-Wert (alo Typ void), Daten verschwinden im Nirvana

```
ArrayList<String> zug = new ArrayList<String>();
…
Iterator<String> bestandteil = zug.iterator();
while (bestandteil.hasnext()) {
   String element = bestandteil.next();
   if (element.equals("Wagon 1")) {
      bestandteil.remove();
   }
   System.out.println(element);
}
```

1.5.2. Exzeptions (exception's)

Erfahrene Programmierer wissen, dass an bestimmten Stellen Programme immer wieder abstürzen. Oft sind unerwartete Nutzer-Eingaben oder Rechen-Fehler (Division durch Null) die Ursache. Die meisten Probleme kann man nur schwerlich vorausplanen und durch konventionelle Programmierung verhindern. Typische Probleme tauchen beim Anschließen von externen Geräten, Schreib- und Lese-Zugriffen im Netzwerk, Typ-Umwandlungen und nummerischen Eingaben auf. So müsste ja vor jeder Division geprüft werden, ob der Divisor den Wert Null hat.
Problematisch an den Unterbrechungen ist vor allem, dass meist die gerade bearbeiteten Daten noch garnicht gespeichert sind. Auf Daten-Verluste reagieren die meisten Nutzer eher (leicht) aggressiv.
Etwas cleverer ist der Ansatz, zwar vorausplanend zu programmieren, es aber mit den Zwischen-Kontrollen nicht zu übertreiben. Stattdessen wartet man auf die dann seltenen Fälle, dass ein Fehler auftritt. Erst in diesem Fall reagiert das Programm auf eine vorgeplante Art und Weise.
Moderne Programmier-Sprachen bieten Konstrukte, die auftretende Fehler-Meldung / Unterbrechungen verfolgen und dem Programmierer die Gelegenheit geben ausweichenden Code aufrufen zu lassen. Die Unterbrechungen bzw. Fehler-Meldungen werden Laufzeit-Fehler oder exception's genannt. Sie sind mehr oder weniger "unerwartete" Ereignisse.
Beim Programmieren kann man bestimmte Stellen im Code – praktisch auch den gesamten Code – so gestalten, dass solche Unterbrechungen abgefangen werden und das Programm darauf sinnvoll reagiert.
Typische "Fehler-Stellen" sind Zugriffe auf Zeiger mit null-Wert, Zugriffe auf nicht existierende Feld-Elemente, weil sie außerhalb des definierten Feldes liegen (würden), .

```
try {
        //Anweisungen die beobachtet werden sollen
} catch (FehlerCode) {
        //Anweisungen, die im Falle des FehlerCode's ausgeführt werden sollen
}
```

1.5.3. Datenstrukturen für Erfahrene

Zusammenfassung von primitiven oder komplexen Daten-Typen

lineare Daten-Strukturen

- Array (Feld, Vektor, …; Index-Zugriff)
- List (Liste, …; Zugriff über next(), Zeiger, Pointer)
- Stack (Stapel, Keller, LIFO, …; Kopf- oder Fuß- bzw. Anfang- oder Ende-gesteuerter Zugriff)
- Queue (Warteschlange, FIFO, …; Anfang- und Ende-gesteuerter Zugriff

verzweigte Daten-Strukturen (Baum-artige Daten-Strukturen)

-

ringförmige Daten-Strukturen

- Deque ()

weitere Unterscheidungs-Möglichkeiten

- Art des Zugriff's auf die Daten
- Möglichkeiten des Einfügen's, Hinzufügen's, Löschen's von Elementen
- Art der Bewegung / Bewegungs-Möglichkeiten durch die Datenstruktur
- Möglichkeiten der Suche und Sortierung
- Komplexität der Daten-Elemente
- …

Definition(en): Datenstrukturen
Datenstrukturen sind Typen von Ablage-Systemen von Informationen (Daten).
Datenstrukturen sind die Anordungs-Prinzip, wie gleichartige Elemente (Daten, Informations-Pakete) zusammengefasst werden.

Definition(en): Kollektionen (Sammlungen)
Kollektionen (Sammlungen) sind lineare Ablage-Systeme für Informations-Sammlungen aus gleichartig typisierten Daten.

besondere / komplexe Daten-Strukturen

- Map (Wörterbuch, Schlüssel-Wert-Paare, …; Schlüssel-orientierter Zugriff)
- Set (Menge, …:)
- Tree (Baum, …;)
- Graph (Knoten-Kanten-Konstrukte, …;)

1.5.3.1. Erzeugen / Anlegen von Objekten mit Objekt-Datentyp (Wrapper-Objekte)

Sollen Daten zu größeren Verbunden zusammengefasst werden, dann eignen sich die einfachen Datentypen (mit Ausnahme einer Verwendung in Array's) dafür nicht. JAVA verwaltet in höherrangigen Datenstrukturen nur Objekte. D.h. wir müssen selbst unseren einfachen Ganzahlen (int) vor der Nutzung in komplexen Datenstrukturen in Objekte wandeln. Das ist auf dem ersten Blick etwas sehr aufwendig, bringt aber die Vorteile der Objekte mit. Es lassen sich jetzt spezielle Methoden des Daten-Typs nutzen.

über die Konstruktoren der Objekt-Datentyp-Klasse (Wrapper-Klasse)

Die Objekt-Datentypen für einfache Datentypen heißen Wrapper-Objekte und sind gut an einem großen Buchstaben an der ersten Position des Typ-Bezeichners zu erkennen.

```
Integer anzahl;
Integer x = new Integer(123);
Double abweichung;
Double durchschnitt = new Double(0.0);
```

mit der valueOf()-Methode

```
…
Integer x = Integer.valueOf("123");
Double durchschnitt = Double.valueOf("0.0");
…
```

mittels Boxing

```
…
Integer x = 123;
Double durchschnitt = 0.0;
Boolean istGross = true;
…
```

1.5.3.2. Kollektionen

Unter Kollektionen (engl.: collections) versteht man Sammlungen von Daten in bestimmten Strukturen. Die in den Kollektionen gespeicherten Daten sind grundsätzlich Objekte. Praktisch werden nicht die gesamten Objekte verwaltet / gespeichert, sondern nur deren Speicher-Referenzen (Zeiger (auf den Speicher-Bereichs des Objekt's)).
Die Objekte einer Kollektion müssen den gleichen Datentyp besitzen. Das scheint zuerst hinderlich zu sein. Man kann aber Kollektionen ineinander schachteln. Dadurch werden die Kollektionen dann extrem flexibel. Der Datentyp einer Kollektion wird in spitzen Klammern – also Kleiner-/Größer-Zeichen (**< >**) – angezeigt.
Um z.B. einfache / primitive Daten (Ganzzahlen, Gleitkommazahlen) in einer Kollektion zu speichern bedarf es der vorherigen Umwandlung der primitiven Daten in ein Objekt (- also praktisch der Festlegung eines expliziten Zeigers auf die Speicher-Adresse, wo diese Daten (von eh her) gespeichert sind). Die technischen Details müssen wir uns nicht kümmern – das ist eine integrierte Aufgabe von JAVA.
Die Kollektionen sind selbst wieder Objekte und besitzen damit auch verschiedene Attribute und Methoden.
Die Verwendung der von JAVA bereitgestellten Kollektionen ist nicht zwingend. Jeder Programmierer kann sich seine Daten organisieren, wie er will bzw. wie er kann. Beim Programmieren von eigenen Kollektionen lernt man richtig dazu. Der große Vorteil der vordefinierten Kollektionen ist aber, dass hier die Programmier-Arbeit schon erledigt wurde. Profi's haben die üblichen Methoden (und noch ein paar mehr) sehr effektiv und manchmal gleich in Maschinensprache programmiert. Dadurch sind sie sehr schnell. Außerdem kann man davon ausgehen, dass die angebotenen Kollektionen zur Genüge getestet sind. Alles Arbeiten, die man sonst selbst durchführen müsste.
Allgemein kann man sagen, dass bei ausreichendem Speicherplatz und normaler Rechner-Leistung die Verwendung von Kollektionen besser ist, als die Nutzung von primitiven Datentypen und deren Array's. Sie stellen automatisch notwendige und praktische Attribute und Methoden zur Verfügung.
In den nächsten Käpitelchen stellen wir einige der möglichen Kollektionen vor. Wem das hier zu abstrakt ist, und sich derzeit noch nicht mit solchen komplizierten Dingen beschäftigen möchte, der kann sich auch erst einmal mit Varianten der Problem-Lösung (Rekursion und / oder Iteration; → 1.5.5. Rekursion und / oder Iteration) oder mit dem großen Thema Klassen und Objekte (→ 1.6. Objekt-orientierte Programmierung für Fortgeschrittene) beschäftigen. Hierher kann man dann bei Bedarf zurückkehren.
Am Schluß nennen wir die von JAVA bereitgestellten Kollektionen einmal kurz mit einigen Hinweisen. Hier ist dann eine eigene Recherche notwendig.
Damit man die jeweiligen Kollektionen nutzen kann, müssen diese als Bibliothek eingebunden werden, z.B. mit:

```
import java.util.HashMap;
…
```

Vorteile von Kollektionen

- fertig vordefiniert / installiert / bereitgestellt
- stellen Vielzahl von Attributen und Methoden bereit
- getestet
- schnell
-

Nachteile von Kollektionen

- nur für Objekt-Datentypen geeignet
- für Anfänger zuerst gewöhnungs-bedürftig
- für Anfänger / Einsteiger relativ hohes Einstiegs-Level
-

Links:
https://docs.oracle.com/javase/8/docs/api/ (online-Dokumentation der oracle®-Java-API)

allgemein verfügbare Methoden für Kollektionen (Auswahl):

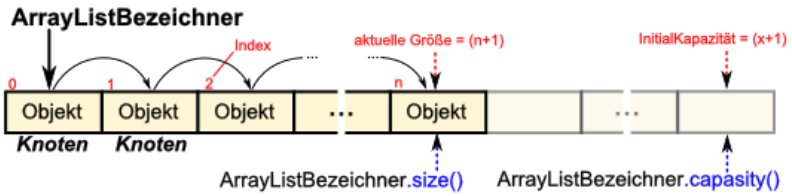
Einige Methoden sind gleichbedeutend in allen Kollektionen vorhanden. Sie seien hier kurz vorgestellt:

Bezeichner**.isEmpty()**
prüft, ob die Kollektion leer ist (- also keine Daten / Objekte enthält)

Bezeichner**.clear()**
löscht alle Elemente aus der Kollektion

Die Verwendung von clear() hat aber mit großer Vorsicht zu geschehen. Auf dieser Arbeitsebene gibt es keinen Papierkorb od.ä. Was gelöscht wurde, ist endgültig verloren.

1.5.3.2.1. Objekt-Felder (ArrayList)



Eigenschaften / Merkmale / Beschränkungen von Array's

- bei Deklaration erfolgt Reservierung von Speicherplatz
- zusammenhängender Speicherplatz notwendig
- Größe ist nicht mehr veränderlich
- der Datentyp der Elemente muss festgelegt werden
- Zugriff auf Elemente ist über den Index (beginnend bei 0) möglich
- Vergößerung ist nur indirekt über das Neu-Erstellen eines weiteren Array's möglich, in den dann die alten Daten reinkopiert werden müssen (und dann die neuen Elemente ergänzt werden können) → teure Operation (braucht Laufzeit und Speicher!)

- Iteration von beliebigen (Index-)Startpunkt möglich, Richtung wählbar und auch die Schrittweite (kann auch variieren)
-

import java.util.**ArrayList**;
…
ArrayList<ObjektDatentyp> *Bezeichner* = new ArrayList<>();

Bezeichner**.add(**Element**)**
speichert ein Objekt anhängend in die ArrayList

Bezeichner**.size()**
liefert die aktuelle (Gesamt-)Größe des Array's zurück

Bezeichner**.get(**ElementIndex**)**
liefert das Objekt aus dem Array zurück, das sich an der Position ElementIndex befindet

Bezeichner**.set(**ElementIndex, neuesElement**)**
ändert / ersetzt das Element an der Position ElementIndex

Bezeichner**.indexOf(**Element**)**
liefert den ElementIndex (die "Position") des Objekt / Element aus dem Array zurück, das sich an der Position befindet

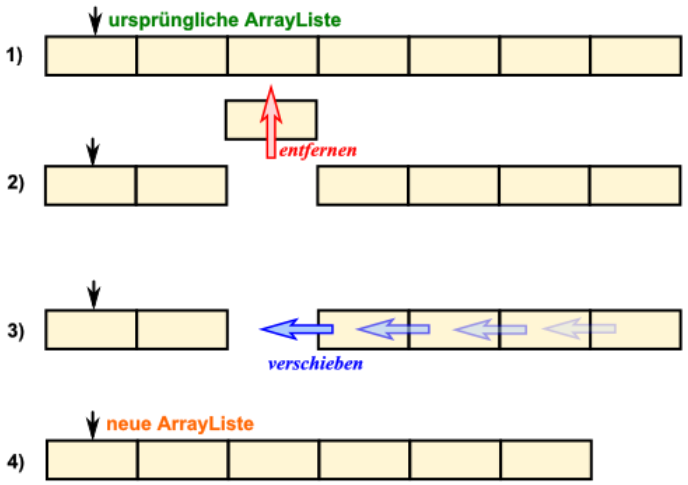
Bezeichner**.contains(**Element**)**
prüft, ob ein Element im Array vorhanden / enthalten ist

Bezeichner**.remove(**ElementIndex**)**
entfernt das Objekt das sich an der Position ElementIndex befindet, aus dem Array
die anderen (nachfolgenden) Elemente rücken dann automatisch eine Position vor

Bezeichner**.clear()**
löscht alle Elemente des Array's

Hier besteht die Gefahr, das ein Löschen eines vorderen Elementes aus einer großen ArrayList sehr zeitaufwändig werden kann. Man sagt, eine solche Operation ist "teuer".
Ähnliches passiert, wenn ein Element vorne in ein ArrayList eingefügt werden soll. Dazu müssen vorher alle Elemente eine Position nach hinten rutschen. Auf den vorderen freien Platz kann dan das neue Objekt gespeichert werden.

Sind solche Operationen in großen Daten-Beständen häufig zu erwarten, dann sollte man vielleicht über eine andere – besser geeignete – Kollektion nachdenken.



Vorteile von ArrayList

- im Vergleich zum "normalen" dynamisch (Größe veränderlich)
- schneller Zugriff auf beliebige Elemente der Kollektion
-

Nachteile von ArrayList

- Position eines Elementes in der Kollektion kann sich ändern (neuer Index)
- langsamer beim Löschen und Hinzufügen (als LinkedList)
- beim Überschreiten der vorgeplanten Listen-Kapazität ist intern ein Zeit-aufwändiges Umkopieren notwendig

Übergabe bei Methoden:

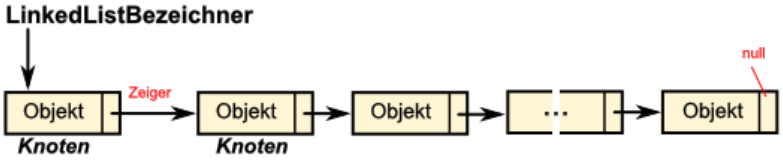
… MethodenBezeichner(ArrayList<Datentyp> lokalerFeldBezeichner, …){ …

Links:
https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html (online-Dokumentation der oracle®-Java-API zu ArrayList's)

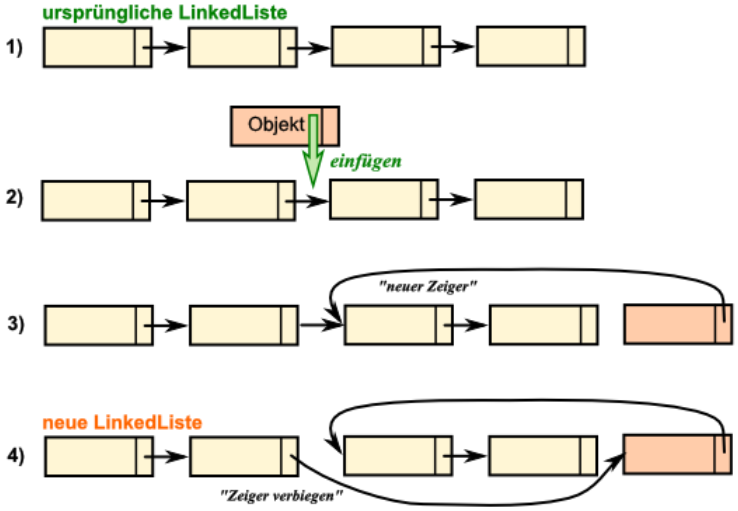
Aufgaben:

1. Erstellen Sie ein Feld (ArrayList) mit Ihren 3 Lieblings-Unterrichtsfächern!
2. Lassen Sie sich das Feld zeilenweise anzeigen!
3. Ergänzen Sie nun noch das "Fach" "Java-Programmierung"! (mit Anzeige!)
4. Nun soll das Programm die Position von "Informatik" anzeigen! (Programm sollte immer aus der vorherigen Aufgabe erweitert werden!)
5. Ändern Sie die Anzeige der Fächer in eine Liste mit vorgestelltem Satz:
Meine Lieblingsfächer sind: …
(Denken Sie and die Komma's und den abschließenden Punkt.)
6. Ändern Sie

1.5.3.2.2. Objekt-Listen (verkettete Listen, LinkedList)



Eine Liste besteht in JAVA aus Knoten (Node's). Jede Node enthält neben dem eigentlichen Objekt eine Verweis (Zeiger) auf den nächsten Node.
Man erhält also eine Liste von Knoten (Node's).
Das letzte Element zeigt auf null (dem "Zeiger-Nirvana"). Das erste Element ist der sogenannte Kopf (Head) der Liste. Beim Rest spricht man auch vom Listen-Körper oder –Schwanz.Die Lage der Node's im Speicher kann völlig durcheinander sein. Das hat keine Beduetung. Ob JAVA die vielleicht nachfolgende Speicher-Adresse direkt hinter dem aktuellen Element (Node) aufruft, oder eine irgendwo im Speicher, ist der gleiche Aufwand.



Eigenschaften / Merkmale / Beschränkungen von Listen

- bei Deklaration erfolgt nur Festlegung des Datentyp's und der Einstiegstelle
- kein zusammenhängender Speicherplatz notwendig, da Zeiger auf Element an einer beliebigen Stelle zeigen können
- jedes Element enthält neben den Nutzdaten noch einen Zeiger auf das Nachfolge-Element (bei doppelt verketteten Listen auch auf das Vorgänger-Element)
- Iteration beginnt immer beim 1. (od. ev. auch letztem) Element
- bei Iteration wird immer die gesamte Liste durchgegangen, zumindestens bis das gesuchte Element gefunden wurde (ein Überspringen von Elementen ist nicht möglich)
- kein freier (Index-gesteuerter) Zugriff auf Elemente möglich
- Erweiterung leicht möglich, auch innerhalb der Liste (→ Verbiegen der Zeiger)
-

import java.util.LinkedList;
…
LinkedList<ObjektDatentyp> *Bezeichner* = new LinkedList<>();

Bezeichner**.size()**
liefert die aktuelle (Gesamt-)Anzahl der Elemente in der LinkedList zurück

Bezeichner**.getFirst()**

liefert das erste Elemente der LinkedList zurück – also: Head (/den Listen-Kopf)

Bezeichner**.removeFirst()**
entfernt das erste Element aus der LinkedList – also: Head (/den Listen-Kopf)
neuer Kopf wird das Element, auf den der Zeiger des entfernten Elementes gezeigt hat
??? beim letzten Element???

spezielle Listen aus der Sicht von JAVA sind Warteschlangen (Schlangen, Queue's,) – auch FIFO-Speicher (First-In-First-Out-Speicher) genannt
auch Keller (Stack's, Stapel) werden als Listen betrachtet. Ihre Speicher-Organisation ist LIFO (Last-In-First-Out, FILO .. First-In-Last-Out)

Vorteile von LinkedList

- schnelles Einfügen / Entfernen von Elementen (nur "Verbiegen" von Zeigern / Referenzen)
-

Nachteile von LinkedList

- Element-Zugriff immer nur mittels Durchsuchen (/kein schneller Zugriff mehr) möglich
-

Links:
https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html (online-Dokumentation der oracle®-Java-API zu LinkedList's)

Vor- und Nachteile von Listen und Array's im Vergleich

Operation(en)	Aufwand gering "billig" gleich gross "teuer"
Einfügen / Entfernen am Anfang	✓ immer gleich groß, da unabhängig von der Länge der **Liste**
	✓ abhängig von der Länge des **Array**'s
Einfügen / Entfernen am Ende	✓
	✓
Iteration über alle Elemente	✓ abhängig von der Länge der **Liste**
	✓ abhängig von der Länge des **Array**'s
zuflälliger Zugriff auf ein Element Random Access	✓ statistisch mittlere Zugriffs-Zeit
	✓ direkte Zugriff auf ein Element des **Array**'s möglich
Intererieren über einen beliebigen Teilbereich (Subset)	✓ abhängig von der Länge und der Position des Teilbereichs im **Array**
	✓ abhängig von der Länge und der Position des Teilbereichs

eine exaktere Betrachtung folgt später (→ 2.x. Betrachtungen zur Effektivität von Algorithmen)

1.5.3.2.3. Mengen (HashSet, Set)

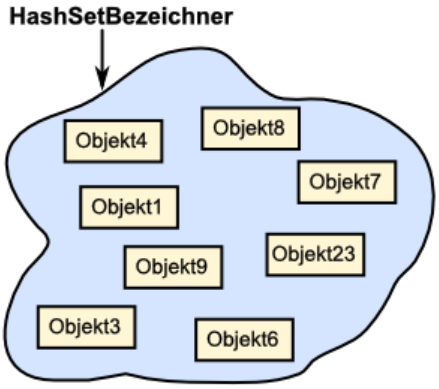
Auch Mengen sind uns aus dem Mathematik-Unterricht bekannt. In Mengen sind Elemente immer nur einmalig enthalten.

während im allgemeinen Verständnis ein Set eine Zusammenstellung von mehreren zueinander passenden Objekten sind, handelt es sich bei Set's im informatischen bzw. JAVA-Sinn um immer gleichartige Objekte

Desweiteren kommen die Objekte immer nur einfach vor. Dopplungen sind nicht zugelassen.

für den Programmierer wird praktisch aber gar nicht das Objekt abgelegt sondern nur ein Marker / Stellvertreter ("Objekt1 ist hier")
es gibt deshalb auch keine Nimm-Methode (Get, Put, … od.ä.)
für Operationen muss das fragliche Objekt immer schon bekannt sein

technisch sind die Objekte natürlich in der Menge gespeichert



import java.util.HashSet;
…
HashSet<ObjektDatentyp> *Bezeichner* = new HashSet<>();

```
HashSet<int> meineMenge = new HashSet<>();
boolean erg;    // zur Aufnahme der funktionalen Rückgabe-Werte
erg = meineMenge.add(2);
erg = meineMenge.add(2);     // liefert false zurück

int umfangMenge = meineMenge.size();
erg = meineMenge.contains(2);    //prüft ebenfalls Vorhandensein
erg = meineMenge.isEmpty();
erg = meineMenge.remove(2);
```

Vorteile von HashSet

- wenn Einmaligkeit von Daten-Elementen / Objekten gebraucht wird
-

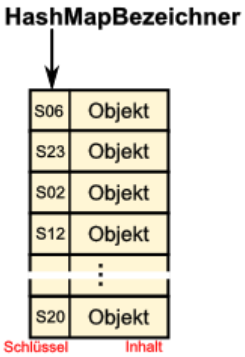
Nachteile von HashSet

- Zugriff nur über bekanntes Einzel-Objekt oder Iteratoren (über alle möglichen Objekte) realisierbar
- es gibt keine interne Ordnung
- Suche nur über Set-eigene Funktionen möglich

Links:
https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html (online-Dokumentation der oracle®-Java-API zu HashSet's)

weiteres Set in JAVA ist das **TreeSet**

1.5.3.2.4. Wörterbücher (HashMap; dictonary; Map)



Mit Wörterbüchern hatte wohl jeder schon mal Kontakt. Klassisch werden sie im Sprachen-Unterricht verwendet. Aber auch Lexika's sind nichts anderes als spezielle Wörterbücher. I.A. haben wir in einem Wörterbuch kleine Artikel oder Einträge, bei denen ein Begriff entweder mit Übersetzungen oder Begriffserklärungen kombiniert sind. Die Ordnung der Begriff erfolgt entweder alphabetisch oder thematisch über Schlüssel-Begriffe (Schlagwörter).
Sie haben eine Feld-artige Struktur aber keine Nummerierung. Besonders gut eignen sich HashMap's für Felder mit großen Lücken in den gedachten Indizes. Gute Beispiele sind echte Vokabel-Listen, wo ja z.B. nicht jede Buchstaben-Kombination ein Wort der Sprache darstellt.
Ein anderes passendes Beispiel sind Postleitzahlen. Sie sind nur in kleinen Bereichen fortlaufend vergeben, meist klaffen große Lücken dazwischen. Die vorderen Ziffern sind von der Region abhängig.
In JAVA sind Wörterbücher unsortierte Sammlungen von Einträgen, die wir hier wieder allgemein Elemente nennen. Ein Eintrag besteht immer aus einem Paar von einem Schlüssel-Wort (Key) und einem Wert (Value). Die Werte sind die von uns gespeicherten Objekte. Die Schlüsselwörter sind i.A. von einem einfachen Datentyp. Aber auch die Verwendung von Objekten ist möglich.
Jeder Key darf nur einmal in einem Map vorhanden sein.

Map: Kennziffern

Key	Value
aa	3988746
av	8847624
dc	5553422
cq	9662419
le	9999362
ab	1111111
xd	4899734
az	8888763

```
import java.util.HashMap;
…
HashMap<KeyDatentyp, ValueDatentyp> Bezeichner = new HashMap<>();
```

Bezeichner**.put(**Schlüssel, Wert**)**
hängt einen Eintrag in die HashMap mit dem Namen Bezeichner an. Der Eintrag besteht aus dem Schlüssel-Wert-Pärchen (Schlüssel, Wert).

Bezeichner**.size()**
liefert die aktuelle (Gesamt-)Anzahl der Elemente in der HashMap zurück

Bezeichner**.get(**Schlüssel**)**
liefert aus einen Eintrag in der HashMap mit dem Namen Bezeichner den (korrespondierenden) Wert zum Schlüssel zurück.
wenn der Schlüssel nicht vorhanden ist, dann wird **null** zurückgegeben

Bezeichner**.containsKey(**Schlüssel**)**
prüft, ob in der HashMap der übergebene Schlüssel existiert.
gibt **true** oder **false** zurück

Bezeichner**.remove(**Schlüssel**)**
entfernt den Eintrag mit dem übergeben Schlüssel aus der HashMap.

wenn der Schlüssel nicht vorhanden ist, dann wird **null** zurückgegeben

Bezeichner.**keySet()**
liefert die gesamte Schlüssel-Liste zurück, über die dann z.B. iteriert werden kann

Bezeichner.**values()**
liefert eine (unsortierte) Liste der Werte zurück

Vorteile von HashMap

- effektiv für Key's mit breitem Wertebereich (und ev. großen Lücken)
- direkte Beziehung zwischen zwei Daten-Elementen
- Entfernen von Elementen einfach
- Eintragen neuer Elemente durch Anhängen

Nachteile von HashMap

- für Suche muss u.U. das ganze Wörterbuch durchsucht werden ("teuer")
- ein direktes Iterieren über die Map ist nicht möglich (nur über den Umweg, das zuerst die Schlüssel-Liste abgefragt wird und diese dann ausgewertet / benutzt werden kann)
-

Links:
https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html (online-Dokumentation der oracle®-Java-API zu HashMap's)

Definition(en): Map
Eine Map ist eine Mengen-basierte Sammlung von Daten (Werte, Value's), auf die über einen Kennwert (Schlüssel, Key) zugegriffen werden kann. Jeder Schlüssel darf nur einmal in der Sammlung vorkommen!

Beispiel1:

```
HashMap<String, String> wb_eng_deu = new HashMap<>();
wb_eng_deu.put("house", "Haus");
wb_eng_deu.put("mouse", "Maus");
wb_eng_deu.put("house", "Gebäude");    //überschreibt alten Eintrag unter Key "house"
String eintrag = wb_eng_deu.get("house");
String entnahme = wb_eng_deu.remove("house");
```

Beispiel2:

```
HashMap<Schueler, Tier> hausTiere = new HashMap<>();
hausTiere.put(new Schueler("Tim"), new Tier("Hund", "Rex"));

public class Schueler {
   String name;
```

```
    Schueler(String name) {
        this.name = name;
    }

    hausTier(String art, String name) {
        this.art = art;
        this.name = name;
    }
}
```

Eine weitere Map-Struktur in JAVA ist die TreeMap.

Übersicht(en) zu Kollektionen

Kollektion	**Umschreibung** bildliches Äquivalent	**besonders geeignet für …**
AbstractList		
AbstractQueue		
AbstractSequentialList		
AbstractSet		
ArrayBlockingQueue		
ArrayDeque		
ArrayList	Tabelle	wechselnder, schneller Zugriff auf beliebige Elemente; gefüllte Tabellen
AttributeList		
BeanContentServicesSupport		
BeanContentSupport		
ConcurrentHashMap		
ConcurrentLinkedDeque		
ConcurrentLinkedQueue		
ConcurrentSkipListSet		
CopyOnWriteArraySet		
DelayQueue		
EnumSet		
HashMap	Wörterbuch	Zugriff auf Elemente über nichtnummerische Schlüssel wenn eine vergleichbare nummerische Tabelle nur locker gefülltet wäre
HashSet	Menge	für Sammlungen, in denen Elemente nur einmalig vorkommen (dürfen)
JobStateReasons		
LinkedBlockingDeque		
LinkedBlockingQueue		
LinkedHashSet		
LinkedList	(Warte-)Schlange, FIFO	flexible, häufig veränderliche Listen (Tabellen)
LinkedTransferQueue		
PriorityBlockingQueue		
PriorityQueue		
RoleList		
RoleUnresolvedList		
Stack	Keller-Speicher; LIFO	
SynchronousQueue		
TreeSet	Bäume	
Vector		

1.5.4. zusätzliche Schleifen-Strukturen für einfache Felder und Kollektionen

1.5.4.1. Schleifen ohne Lauf-Variablen

sogenannte ForEach-Schleifen
JAVA übernimmt die Organisation der Schleifen-Variable und der Schleifen-Größe (Schleifen-Abbruch) für uns. Immer, wenn man die Schleifen-Variable intern nicht braucht, dann bietet sich die verkürzte for-Schleife an.
ForEach-Schleifen iterieren selbstständig durch das array bzw. die Kollektion. Dabei werden automatisch alle Elemente erfasst. Wir müssen dann nur entscheiden, welches Element wir z.B. benutzen wollen. Entweder sind das alle oder man wählt mit einer if-Anweisung das geeignete Element aus.

for (Objektdatentyp *ElementBezeichner* : *StrukturBezeichner*){ …

Das einzelne Element bekommen wir über den ElementBezeichner zu fassen.
Die Ausgabe wäre dann z.B. so möglich:

System.out.println(*ElementBezeichner*);

1.5.4.2. fortlaufend iterierte Schleifen

Listen-Iteration

.next()
springt zum nächsten Objekt in der ArrayList. Das passiert ungeprüft auch über das Ende des Feldes hinweg. Deshalb sollte immer vor dem Aufruf von **next()** geprüft werden, ob es noch ein reguläres Objekt (**hasNext()**) in der Struktur gibt.

```
import java.util.ArrayList;
…
ArrayList<ObjektDatentyp> FeldBezeichner = new ArrayList<>();
ListInterator<Integer> HochzählerBezeichner = FeldBezeichner.listInterator()

while (HochzählBezeichner.hasNext()) { …
    NutzObjekt = HochzählBezeichner.next();
```

oder:

```
while (HochzählBezeichner.hasNext()) { …
    HochzählBezeichner.next() = Objekt;
```

Aufgabe:

Ginge z.B. die folgende Struktur unproblematisch? Wenn JA, dann zeigen Sie die Durchläufe (Variablen + Werte) auf! Wenn NEIN, dann stellen Sie mögliche Probleme dar! (Die Definition des Objektes Tier mit seinen Attributen ist als gültig zu betrachten.)

```
ArrayList<Tier> alleHunde = new ArrayList<>();
Tier hund = new Tier;
ListInterator<Integer> einHund = alleHunde();
while(einhund.hasNext() {
   hund = einHund.next();
   hund.name="Rex";
   einHund.next()=hund;
   System.out.println(einHund.next());
}
```

Exkurs: Wiederholung for-Schleife mit Lauf-Variable

Zu der gerade aufgezeigten ForEach-Schleife wollen wir nur zur Wiederholung und als Vergleich die ausführliche for-Schleife aufzeigen:

```
for (int i=0 ; i<StrukturBezeichner.size() ; i++){ …
```

Das ist doch deutlich schreib-aufwändiger, hat aber den Vorteil, dass man für den Fall, dass man später doch die Schleifen-Variable braucht, diese nicht erst wieder aufwändig einfügen muss.
Im Allgemeinen braucht man aber bei ForEach-gedachten Schleifen auch keine Lauf-Variablen.
Bevor man die ganze Schleife umbaut, sollte man vielleicht über eine eigene Variable nachdenken, die man vollständig selbst verwaltet.

```
int i = 0;
for (ObjektDatentyp ElementBezeichner : StrukturBezeichner){
    …
    i++;
    …
```

Um den Abbruch über i braucht man sich nicht zu kümmern, da er ja nicht wirklich in der ForEach-Schleife gebraucht wird.

1.5.4.3. Iterationen mit foreach

for (*Typ Schleifenvariable* : *Sammlung*) { *Scheifen-Code* }

```
String[] kapitelnamen = {"Einleitung", "1. Kapitel", "2. Kapitel", "3. Kapitel", "Ende"};
for (String kapitel : kapitelnamen){
   System.out.println(kapitel)
}
```

- keine Zähl-Variable gebraucht
- direkte Benutzung der Elemente (keine Indizies, wie bei "normalem" for)
- Iteration über alle Elemente
- funktionieren auch bei dynamisch veränderten Array-Listen und vielen anderen Daten-Strukturen (Kollektionen)

```
ArrayList<String> kaptitelnamen = new ArrayList<String>();
kapitelnamen.add("Einleitung");
kapitelnamen.add("1. Kapitel");
kapitelnamen.add("2. Kapitel");
kapitelnamen.add("3. Kapitel");
kapitelnamen.add("Ende");
```

Iteration mit klassischer for-Schleife

```
for (int i = 0; i < kapitelnamen.size(); i++) {
   System.out.println(kapitelnamen.get(i));
}
```

Iteration über foreach

```
for (String kapitel : kapitelnamen) {
   System.out.println(kapitel)
}
```

In ForEach-Schleifen dürfen keine Elemente entfernt oder hinzugefügt werden! Das würde die interne Schleifen-Kontrolle durch den Compiler durcheinander bringen. Es dürfen nur die Werte der Elemente gelesen und geändert werden!

1.5.5. Lösungs-Strategien → Rekursion und / oder Iteration?

Die Frage, die wir uns hier stellen wollen, ist die nach dem besten Vorgehen beim Lösen eines Problems. Probleme sind im Vergleich zu einfachen Aufgaben dadurch gekennzeichnet, dass es keine direkte Lösung – also schon einen fertigen Algorithmus – für sie gibt. Im informatischen Bereich meint man aber die Umsetzung einer Aufgabenstellung in ein funktionierendes Programm. I.A. ist der Algorithmus / eine Vorgehens-Vorschrift schon bekannt. Sie kann in den verschiedensten Formen notiert sein. Neben der Text-Form haben wir auch schon Struktogramme kennengelernt. Auch UML-Diagramme bieten Ansätze zum Lösen von informatischen Problemen. Andere Formen sind Linien-Diagramme oder die etwas antiquierten Programm-Ablauf-Pläne – kurz PAP's.
Eine Variante wäre es, ein Problem zuerst einmal auf ein oder mehrere einfachere Probleme zurückzuführen. Das macht man solange, bis es kein einfacheres Problem mehr gibt oder die Lösung offensichtlich ist. Auf dem Rückweg zum ehemaligen aufrufenden (großen) Problem ergänzt man die primitive Lösung immer ein Stück weiter.
Glaubt man der Literatur, dann ist dieses Lösungs-Verfahren, welches von Menschen und Programmierern (auch das sollen Menschen sein?!), am häufigsten / vorrangig genutzt wird. Diese Strategie heißt Rekursion (lat.: ***recurrere*** = zurücklaufen).
Die andere Variante ist das gleichartige Wiederholen einer bekannten / einfachen Lösung bzw. einer Teil-Tätigkeit, bis die Aufgabe gelöst ist. Der Name diese Startegie kommt vom lat.: ***iterare*** (wiederholen).
Meiner Meinung nutzen Menschen eher diese Methode. Bei Personen, die Programmieren lernten ist es ebenfalls die zuerst gewählte Lösungs-Strategie. Aber das hat vielleicht auch damit zu tun, dass in der Programmier-Ausbildung fast immer Verweigungen und Schleifen einen dominierenden Anteil einnehmen.
Praktisch ist es wohl eine nicht-entscheidbare Frage – wie die, was denn nun zuerst da war, das Huhn oder das Ei. Zum Einen lassen sich Probleme fast immer mit beiden Strategien lösen. Dabei ist meist die eine Strategie eleganter / effektiver / cleverer / schöner / …, aber das steht erst einmal nicht Disposition.
Zum Anderen gibt es sie nicht – die universell beste Strategie, sonst könnten wir sie ja einfach ansagen / lehren / predigen. Vielfach hängt das beste Vorgehen von den Rahmen-Bedingungen ab, die zur Verfügung stehen. Im Computer-Bereich sind dies z.B. Speicherplatz oder die Rechen-Zeit.
Meist geht es bei Iteration und Rekursion auch begrifflich etwas hin und her. Den "die" Iteration oder "die" Rekursion gibt es gar nicht. Es sind verallgemeinerte Strategien.
Praktisch müsste man zwischen der interativen und / oder rekursiven Defintion einer Funktion und der programmiertechnischen Implementierung unterscheiden.
i.A. lassen sich die – wie auch immer definierten – Funktionen auf beide Arten implementieren; allerdings gibt es ohne weiteres Programmier-System, die bestimmte Strategien bevorzugen bzw. manche andere gar ausschließen.
Vielfach entscheidet der Programmierer, was günstiger ist.
Da beide Umsetzungen Vor- und Nachteile haben, müssen die System-Bedingungen aber mit beachtet werden

Achtung!!! nachfolgend sind viele Programm-Beispiele noch in Python!

1.5.5.1. Iteration

Wiederholung strukturgleicher Blöcke

Ein Beispiel für eine interative Lösungs-Strategie kennen wir alle noch aus der Grundschule. Die Berechnung von 4 * 3 ist dort keine triviale Aufgabe. Bevor wir die Malfolge auswendig lernen, versuchen wir es ersteinmal mit der Addition.

Schließlich kann man für 4 * 3 auch 3 + 3 + 3 + 3 schreiben. Praktisch heißt das, wir wiederholen die Addition noch dreimal, denn eine 3 haben wir ja schon vorgegeben.
Super algorithmisch wäre natürlich die Form, dass wir mit 0 beginnen und darauf 4 mal eine 3 addieren. Wie auch immer die 12 ist unser Lohn.
Computer sind die Inbegriffe von Iterationen. Eigentlich machen sie nichts anderes. Das wird uns umso bewußter, umso mehr wir uns vergegenwärtigen, was Computer eigentlich wirklich können.
Paraktisch kann ein Prozessor auf seiner "Transistoren"-Ebene nur:

- **ein Bit auf 1 setzen**
- **ein Bit wechseln**
- **vergleichen ob ein Bit eine 1 enthält**

Alle anderen Leistungen, die wir heute zur Genüge nutzen, sind einfach nur clevere Wiederholungen und Kombinationen dieser Grundfähigkeiten.
Kommen wir zu unserem Multiplikations-Beispiel zurück. Die einfach Umsetzung würde so aussehen:

```
…
int sum = 0;
sum = sum + 3;
sum = sum + 3;
sum = sum + 3;
sum = sum + 3;
System-out.println(sum);
…
```

Das schreit schon nach Vereinfachung. Wir haben hier eine einfache abgezählte Wiederholung, also sind wir bei den Zählschleifen:

```
…
int sum = 0;
for (int i = 0; i <= 4; i++) {
    sum = sum + 3;
}
System-out.println(sum);
…
```

Zählschleifen sind eine praktische Form der Iteration.
Verallgemeinert – also auch für andere Zahlen, die eingegeben werden könnten – sieht der Programm-Abschnitt dann z.B. so aus:

```
…
int zahl = 3;
int faktor = 4;
int sum = 0;
for (int i = 0; i <= faktor; i++) {
    sum = sum + zahl;
}
System-out.println(sum);
…
```

Da wir schon "wissen", das wir die Multiplikation noch tausend Mal brauchen, lagern wir die Berechnung in eine schöne Methode (**multipliziere()**)aus:

```
…
public int multipliziere(int zahl, int faktor) {
    int sum = 0;
    for (int i = 0; i <= faktor; i++) {
        sum = sum + zahl;
    return sum;
}

int zahl = 3;
int faktor = 4;
System-out.println(multipliziere(zahl, faktor);
…
```

Definition(en): Iteration
Unter Iteration versteht man das mehrfache (abzählbare / gezählte) Wiederholen einer Aktion / Handlung / Anweisung.
Iteration ist die schrittweise ode wiederholte Anwendung der gleichen Arbeitsschritte / Prozesse / Zugriffe auf bereits gewonnene Zwischen-Ergebnisse.
Iteration ist die Wiederholung einer Tätigkeit mit vorher bekannter Anzahl (der Wiederholungen) zur Lösung einer Aufgabe.

Vorteile einer / der Iteration

- **weniger Speicher-Bedarf**
- **intuitiv verständlich**
- **im direkten Vergleich meist schneller**
-

Nachteile einer / der Iteration

- **kompliziertere Umsetzung**
- **längere Programmtexte**
-

1.5.5.1.1. typische Iterations-Anwendungen

Eigentlich könnte ich mir diesen Abschnitt sparen, da die bisher besprochenen Wiederholungen fast ausnahmslos Iterationen waren.
Da aber Summen und Produkte und vor allem deren Entwicklung in Schleifen zu den klassischen Programmier-Aufgaben gehören, seien sie hier noch mal aufgeführt, wiederholt und zum systematischen Verständnis dargestellt.
Wem die Summen- und Produkt-Bildung schon zur Nase raushängt und die Schwierigkeit damit nicht verstehen kann, der sollte gleich zu den Rekursionen (→ 8.4.2. Rekursion) übergehen. Da erwartet ihn vielleicht Neueres und Spannendes.

1.5.5.1.1.1. Summen-Bildung

Betrachten wir die Aufsummation aller natürlichen Zahlen bis zu einer vorgegebenen Endzahl.

```
…
public int summe(int endzahl){
    int sum = 0;
    for(int i=1; <=endzahl; i++){
        sum += i;      // sum = sum + i
    }
    return sum;
}

public static void main(String[] args){
    int endzahl;
    endzahl = InOut.readInt("bis zu welcher Zahl summieren?: ");
    System.out.println("Die Summe lautet: ", summe(endzahl));
}
…
```

Schauen wir uns hier noch mal die Belegung des Speichers an, wenn z.B. eine Methode aufgerufen wird. Das ist vor allem für einen Vergleich mit der Rekursion sehr interessant.

Programm-Teil	Variable / Bezeichner	Speicher
main:	endzahl →	**10**

Im nächsten Schritt übergeben wir die endzahl als Argument an die Methode **summe()**.
Es wird eine neue Speicherzelle mit einer Kopie von **endzahl** belegt. Die Speicherzelle hat innerhalb in der Methode **summe()** auch "zufällig" den Bezeichner **endzahl**.
Nun folgen die Schritte des Methoden-Körpers.

Programm-Teil	Variable / Bezeichner	Speicher
summe:	endzahl →	**10**
main:	endzahl →	**10**

Zuerst wird die Variable **sum** erzeugt und mit 0 initialisiert.
Es folgt die Erzeugung der Schleifen-Struktur, von der uns hier nur die Lauf-Variable **i** interessiert. Die Schleife soll bei 1 beginnend bis zum Erreichen der endzahl dann bestimmte Schritte ausführen.

Programm-Teil	Variable / Bezeichner	Speicher
	i →	1
	sum →	0
summe:	endzahl →	10
main:	endzahl →	10

Diese betrachten wir jetzt nur für einen Schleifen-Durchlauf.
Der einzige Arbeitsschritt im Schleifen-Lörper ist die Addition von i auf sum und die Abspeicherung des (Zwischen-)Ergebnisses in der Speicherzelle **sum**.
Mit dem Ende des Schleifenkörpers wird **i** iteriert und hinsichtlich der im Schleifenkopf angegebenen Bedingung geprüft..

Programm-Teil	Variable / Bezeichner	Speicher
	i →	2
	sum →	1
summe:	endzahl →	10
main:	endzahl →	10

Da die endzahl noch nicht erreicht ist, wird also mit einem weiteren Schleifen-Durchlauf fortgesetzt.
Am Schluß wird der Wert von sum an die aufrufende Stelle zurückgegeben und der Speicher von den Methoden-Variablen gereinigt.
Für uns ist hier die Beobachtung wichtig, dass wir uns immer innerhalb der gerade benutzten 4 Speicherzellen bewegen und das für unser Beispiel die 55 herauskommt..

1.5.5.1.1.2. Produkt-Bildung

Die algorithmischen Änderungen zur Summe-Funktion sind minimal. Natürlich sollten die Bezeichner usw. angepasst werden. Aber für ein schnelles Test-Programm würde es auch ohne gehen.

```
…
public int produkt(int endzahl){
    int prod = 1;
    for(int i=1; <=endzahl; i++){
        prod *= i;      // prod = prod * i
    }
    return prod;
}

public static void main(String[] args){
    int endzahl;
    endzahl = InOut.readInt("bis zu welcher Zahl multiplizieren?: ");
    System.out.println("Das Produkt lautet: ", produkt(endzahl));
}
…
```

Allerdings müssen die **rot** hervorgehobenen Werte und Operationen angepasst werden.

Aufgaben:

1. ***Erstellen Sie ein Speicher-Schema für das Produkt-Programm!***
2. ***Schreiben Sie eine Summe- und eine Produkt-Methode in einem Programm, welche immer die Zahlen von einer Start- bis zu einer Endzahl (über Eingaben festzulegen) verarbeiten!***

1.5.5.2. Rekursion

Rekursion ist Problem-Lösungs-Strategie, die immer wieder für Aha- und Überraschungs-Effekte sorgt. Praktisch gehr es darum ein komplexes Problem auf seine etwas leichte Version zu reduzieren. Das macht man solange bis der Einfachste aller Fälle auftritt und dieser leicht lösbar ist. Dann kehrt man um und läßt die leichten in die etwas schwereren zurückeinfließen. Daher auch der Name, der sich von vom lat.: recurrere (zurücklaufen, zurückkehren) ableitet.

gehört zu den "Teile und Herrsche"-Verfahren / -Techniken / -Algorithmen ("Divide and Conquer")
gemeint ist die Beherrschung eines großen Problem's durch Zerlegung / Aufteilen in kleinere Probleme
im Alltag entspricht dies der Spezialisierung z.B. bei Handwerkern, die zum Bau eines Hauses gebraucht werden
Fachlehrer in der Schule

In der Informatik versteht man also runter Rekursion die Rückführung einer schwierigeren / aufwändigeren / komplizierteren / allgemeinen Aufgabe in (eine) leichtere / weniger aufwändige / einfachere / spezielle.
Besonders gern benutzt und besonders eindrucksvoll sind Rekursionen in der Graphik-Programmierung.
? Turtle-Graphik mit JAVA

Während wir bei der Iteration die Wiederholung – also Aneinanderreihung – von Blöcken als Sinnbild hatten, gehört zur Rekursion das Bild der ineinander geschachtelten Blöcke.
Dabei kann diese – in sich selbst geschachtelte Schleife – hohe Wiederholungszahlen aufweisen.
Die Schleife bzw. das Wiederholungs-Muster muss allerdings irgenwann mal definiert abbrechen, d.h. es muss einen einfachen / trivialen Lösungs- oder Abbruch-Fall (Abschluss- od. Abbruch-Bedingung, …) geben.

In der Mathematik ist Rekursion ein gängiges Mittel zur Definition von Funktionen:

z.B.: Bildung einer Summe

sum(0) = 0 *Rekursions-Anfang*

jede andere Summe lässt sich dann so berechnen:

sum(n) = sum(n-1) + n *Rekursions-Schritt*

die gesamte Definition lautet dann

$$\text{sum}(n) = \begin{cases} 0, & \text{falls } n = 0 \quad \textit{Rekursions-Anfang} \\ \text{sum}(n-1) + n, & \text{sonst} \quad \textit{Rekursions-Schritt} \end{cases}$$

Das ist eine beeindruckend einfache Definition für eine Funktion.

Damit ein Problem rekursiv zu lösen geht, muss es die folgenden Bedingungen erfüllen:

- das Problem muss sich in eine einfachere Variante von sich selbst zerlegen lassen
- bei der Zerlegung in eine einfachere Variante muss irgendwann eine Variante erreicht werden, die sich ohne weitere Zerlegung lösen lässt
- wenn die Teilprobleme gelöst sind, dann müssen sich die Teil-Lösungen zu einer Lösung des Ausgangs-Problems zusammensetzen lassen

Als Beispiel verändern wir hier unser interatives Summen-Programm (→ 1.5.5.1. Iteration) in ein rekursives:

```
…
public int summe(int endzahl){
    if (endzahl == 0){
        return 0;
    } else {
        return summe(endzahl-1) + endzahl;
    }
}

public static void main(String[] args){
    int endzahl;
    endzahl = InOut.readInt("bis zu welcher Zahl summieren?: ");
    System.out.println("Die Summe lautet: ", summe(endzahl));
}
…
```

Hinweis:
Es sei hier darauf hingewiesen, wie leicht sich die mathematische Definition in die informatische umwandeln läßt. Das ist quasi eine 1 : 1 – Übersetzung.

Was dann aber Rekursionen im Speicher machen, ist dann doch schon eine Überraschung.
Zuerst wird nur **endzahl** in der **main()**-Methode angelegt und dann mit dem Aufruf der **summe()**-Methode auch eine Kopie von **endzahl** für die interne verwendung in **summe()**.

Programm-Teil	Variable / Bezeichner	*Speicher*
summe:	endzahl →	**10**
main:	endzahl →	**10**

In der Methode selbst wird erst einmal garnichts gerechnet, weil wir mit return summe(endzahl - 1)… einen erneuten Aufruf von summe() erzeugen. Die innere Methode kennzeichne ich mal mit einer dunkleren Hintergrundfarbe. Aber auch diese Aufruf führt nur wieder zu einem erneuten Aufruf von summe() – jetzt mit dem Argument 8.

Programm-Teil	Variable / Bezeichner	*Speicher*
summe':	endzahl →	**9**
summe:	endzahl →	**10**
main:	endzahl →	**10**

Das Spiel setzt sich jetzt solange fort, bis wir beim Aufruf für die **endzahl = 1** kommen. Nun wird die **summe()**-Methode mit 0 aufgerufen. Auf diese 0 reagiert sie aber nicht mit einem erneuten Aufruf von **summe()** sondern mit der Rückgabe von 0.
Die 0 wird an die aufrufende Stelle zurückgegeben und das ist die Position im return-Konstrukt, die nun mit **+ endzahl** weitergeht. Die **endzahl** war in der aufrufenden Methode **summe$^{8'}$** die 1.

Programm-Teil	Variable / Bezeichner	*Speicher*
summe$^{9'}$:	endzahl →	**0**
summe$^{8'}$:	endzahl →	**1**
summe'':	endzahl →	**8**
summe':	endzahl →	**9**
summe:	endzahl →	**10**
main:	endzahl →	**10**

Also wird die Addition ausgeführt. Es folgt der Rücksprung zur vorhergehenden **summe()**-Methode. Wieder wird die dort gültige 2 als **endzahl** aufaddiert und zur nächsten Vorversion von **summe()** zurückgesprungen.
Das geht nun solange, bis die ursprünglich 1. Kopie von **summe()** und **endzahl** im Speicher angelegt wurde. Und die liefert dann das Endergebnis für die Anzeige "55".
Natürlich werden die nicht mehr gebrauchten Variablen sofort nach dem Rücksprung gelöscht.
Wenn wir uns kurz an die Iteration zurückerinnern, hier wurde das gleich Ergebnis berechnet (- na das sollte wohl auch so sein -) und 4 Speicherzellen gebraucht. Die Rekursion braucht mindstens 11. Meist reden wir aber von Rekursionen mit Tiefen von 100 oder 1'000.
Der enorme Speicherbedarf kann eine Rekursion schon mal zum Abstürzen bringen. Das ist besonders ärgerlich, wenn man ewig lange gerechnet hat und quasi kurz vorm Ziel war.
Der Programmierer ist hier angehalten Sicherheits-System zu integrieren, wie z.B. den Test auf genügend freien Speicher.

Definition(en): Rekursion
Unter Rekursion versteht man das nicht voraussehbare / nicht-abgezählte Wiederholen einer Aktion / Funktion durch Aufruf von sich selbst.
Rekursion ist das Problemlösungs-Konzept, bei dem eine (komplexe) Aufgabe in (kleinere, leichter lösbare) Teil-Aufgaben (der gleichen Klassen) zerlegt wird, diese gelöst werden und dann zur Gesamt-Lösung zusammengesetzt werden. Rekusionen bedürfen einer trivialen Teil-Aufgaben-Lösung, ab der eine weitere Aufgaben-Zerlegung nicht mehr durchgeführt werden kann.

Vorteile einer / der Rekursion

- **relativ einfache Defintion**
- **dem menschlichen Denken ähnlich**
- **Korrekheit ist i.A. leichter zu prüfen**
- **kürzere Formulierung**
- **kürze Implementierungen**
- **spart Variablen**
- **(i.A.) sehr effektiv**

Ob rekursives Arbeiten wirklich dem menschlichen Denken sehr nahe kommt, wage ich zu bezweifeln. Meine Erfahrungen sagen eher, dass rekursive Prinzipien / Funktionen zumindestens sehr einfach erscheinen, beim Umsetzen in ein Programm wird es deutlich schwieriger und noch problematischer wird es, wenn selbst neuartige Sachverhalte / Probleme rekursiv gelöst werden sollen
Meist erscheint dann irgendwie die interative Lösung logischer oder eingängiger. Kommt man später auf eine rekursive Lösung, ist sie zwar meist deutlich eleganter, aber auch schwerer zu verstehen und zu warten / pflegen.
Das Nutzen von Rekursion als Lösungs-Strategie – zumindestens da wo es geht – ist eine Übungssache. Einige Programmiersprachen, wie LISP, SCHEME und HASKELL , bauen vollständig auf rekursive Funktionen.
Computer arbeiten intern aber immer interativ, aber das ist nicht unsere Ebene

Nachteile einer / der Rekursion

- **unübersichtlicher Programmablauf**
- **schlechtes Laufzeit-Verhalten** — im direkten Vergleich
- **größerer Speicher-Bedarf (großer Overhead von Funktions-Aufrufen)** — z.B. für Rücksprung-Adressen von noch nicht gelösten übergeordneten Funktions-Aufrufen
- **ev. Zeit-aufwändiger**

Aufgaben:

1. ***Erstellen Sie ein JAVA-Programm, dass die Summe der natürlichen Zahlen bis zu einer Endzahl realisiert!***
2. ***Überlegen Sie sich die Änderungen im obigen Programm, wenn das Produkt der Zahlen gebildet werden soll!***
3. ***Speilen Sie anhnad eines Speicherplans die einzelnen gebrauchten Speicherzellen und deren Belegungen durch!***
4. ***Recherchieren Sie Summen-Programme für die Sprachen LISP, SCHEME und HASKELL! Finden Sie die JAVA-Strukturen hier wieder?***

Wir unterscheiden zwischen direkter und indirekter Rekursion. Die direkte ist dadurch gekennzeichnet, dass die Funktion sich immer wieder selbst aufruft. Bei der indirekten Rekursion rufen sich mehrere Funktionen gegenseitig auf. Sind es z.B. zwei, dann ruft Funktion1 die Funktion2 auf und diese dann wieder Funktion1.

für eine Rekursion werden immer 2 Teile gebraucht:

- Rekursions-Start (Abbruch-, Abschluss- od. Ende-Bedingung, Base case), Primitiv
  - einfachster Fall od. Fall, der keinen Vorgänger hat
- Rekursions-Schritt ()
  - Definition der Berechnung eines Nachfolger's / aktuellen Element's aus einem Vorgänger / dem letzten Element

Pseudo-Code einer Rekursion:

```
rekursive_Funktion(param) {
        IST param                    //also der einfachste Fall
                DANN ergebnis =  Primitiv_festlegen
```

```
                SONST                    // berechne Nachfolger
                        ergebnis = rekursive_Funktion(param -1)  ○  param
        ZURÜCK ergebnis
}
```

Legende: ○ … steht für Verknüpfung / Berechnung / … / Rekursions-Schritt-Operation

1.5.5.2.1. weitere typische Anwendungen für Rekursionen

1.5.5.2.1.1. Überführung einer Dezimal-Zahl in eine Dual-Zahl

```
def dualzahl(dezimalzahl):
    ganzzahlteiler=dezimalzahl/2
    rest=dezimalzahl%2
    if rest==0:
        stellensymbol="0"
    else:
        stellensymbol="1"
    if ganzzahlteiler==0
        return stellensymbol
    else:
        return dualzahl(ganzzahlrest) + stellensymbol
```

!!! Python

```
…
public (){

}
…
```

1.5.5.2.1.2. die Fakultät

faktorielle Funktion

```
…
public int fakultaet(int x){
    if(x == 1){
        return 1;
    } else {
        return fakultaet(x-1) * x;
    }
}
…
```

1.5.5.2.1.3. die FIBONACCHI-Folge

Die Grund-Idee hinter der Folge soll – so berichten es die Legenden – die Betrachtung, wie sich Kaninchen auf einer Insel fortpflanzen. Praktischerweise geht man von Pärchen aus, damit es bei Kaninchen auch funktioniert. Für Bakterien, Pantoffeltierchen oder Wasserflöhe kann man auch Einzel-Exemplare betrachten, da ihre klassische Vermehrung keine Sexual-Partner benötigt.
Die Folgen-Glieder stellen immer die Situation nach einem Vermehrungs-Zyklus dar. Gemeint ist von Geschlechts- oder Teilungs-Reife bis zur Geschlechts- oder Teilungs-Reife der Nachfolge-Generation. Exakter ist wahrscheinlich – zumindestens für die Kaninchen der halbe Zyklus, dann gehen die Zahlen auch (zumindestens theoretisch) auf. Von solchen Einschränkungen, dass Kaninchen hiernach nur immer zwei Nachkommen haben, sehen wir hier großzügig hinweg.
Begonnen wird meist mit der leeren Insel – was dem 1. Glied den Wert 0 gibt. Dann kommt durch Urzeugung das erste Pärchen auf die Insel. Und dann nimmt das Unheil / die "Sünde" ihren Lauf.

0 1 2 3 4 5 6 7 8 9 10 11 … (Takte, Jahre, Reihen-Glied)
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

$$\text{fib}(n) = \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2), & \text{sonst} \end{cases}$$

Rekursions-Anfang

Rekursions-Schritt

```
…
public int fibonacchi(int n){
    if (n == 0){
        return 0;
    } elseif (n == 1) {
        return 1;
    } else {
        return fibonacchi(n-1)+fibonacchi(n-2);
    }
}
…
```

Exkurs: FIBONACCHI ohne die Vorglieder?

Beide Lösungswege – also die interative bzw. die rekursive – haben durch die vielen Wiederholungen einen recht großen Rechenaufwand. Schließlich müssen alle Vorglieder berechnet werden, um dann die letzten beiden Vorglieder zum Ergebnis zu addieren.
Besonders für höhergliedrige Werte in der Folge ist der Rechenaufwand dann enorm.
Der französische Mathematiker J.-Ph.-M. BINET schlug (1843) eine andere Funktion zur Berechnung der Einzelglieder vor:

$$fib(n) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right)$$

Aufgaben:

1. ***Programmieren Sie die nachfolgende "Super"-FIBONACCHI-Folge als rekursive Funktion mit kleinem Rahmen-Programm zur Anzeige mehrerer Folge-Glieder!***

$$\text{sfib(n)} = \begin{cases} 0, & \text{falls n = 0} \\ 1, & \text{falls n = 1} \quad \textit{Rekursions-Anfang} \\ 2, & \text{falls n = 2} \\ \text{sfib(n-1) + sfib(n-2) + sfib(n-3)}, & \text{sonst} \quad \textit{Rekursions-Schritt} \end{cases}$$

zur Kontrolle: erwartete erste Glieder der Folge:
0, 1, 2, 3, 6, 11, 20, 37, 68, 125, 230, 423, …

Aufgabe (für Fortgeschrittene):

2. ***Erstellen Sie ein Programm, dass für die ersten 20 Glieder der FIBONACCHI-Folge die Werte jeweils klassisch interativ und rekursiv und dann noch einmal mit der BINET-Funktion berechnet. Prüfen Sie, ob es Differenzen gibt (diese auch anzeigen lassen!)!***

1.5.5.2.1.4. das ggT – der Größte gemeinsame Teiler

Natürlich müsste es der ggT (GGT; eng.: gcd (greatest common divisor)) heißen, aber wer spricht schon so?

bei zwei Zahlen **ggT(10584, 40500)**

10584 =	$\mathbf{2^3}$	*	$\mathbf{3^3}$			* 7^2	
40500 =	$\mathbf{2^2}$	*	$\mathbf{3^4}$	*	5^3		
ggT:	$\mathbf{2^2}$	*	$\mathbf{3^3}$				= 108

beim ggT mehrerer (mehr als 2) Zahlen muss allerdings auf die Primfaktoren-Zerlegung zurückgegriffen werden

wird für mehr Zahlen **ggt(1400,283500,20250)**

1400 =	$\mathbf{2^3}$			*	$\mathbf{5^2}$	* 7^2	
283500 =	$\mathbf{2^2}$	*	3^4	*	$\mathbf{5^3}$	* 7^1	
20250 =	$\mathbf{2^1}$	*	3^4	*	$\mathbf{5^3}$		
ggT:	$\mathbf{2^1}$			*	$\mathbf{5^2}$		= 50

Primfaktoren-Zerlegung sehr rechen-aufwändig

EUKLIDischer bzw. STEINscher Algorithmus

40500	:	**10584**	=	3	Rest:	**8748**
10584	:	**8748**	=	1	Rest:	1836
8748	:	1836	=	4	Rest:	1404
1836	:	1404	=	1	Rest:	432
1404	:	432	=	3	Rest:	108
432	:	**108**	=	4	Rest:	**0**

ggT(x, y, z) = ggT(ggT(x, y), z) = ggT(x, ggT(y, z))

1.5.5.2.1.5. Erkennung von Palindromen

rekursiv:

```
def ist_palindrom(zeichenkette):
    if len(zeichenkette)<=1:
        return 1
    if zeichenkette[0]!=zeichenkette[-1]:
        return 0
    return ist_palindrom(zeichenkette(s[1:-1])
```

!!! Python

```
…
public (){

}
…
```

interativ:

```
def ist_palindrom(zeichenkette):
    links=0
    while links<rechts:
        if zeichenkette[links]!=zeichenkette[rechts]:
            return 0
        links+=1
        rechts-=1
    return 1
```

!!! Python

```
…
public (){

}
…
```

mit speziellen Python-Funktionen für Strings und Listen:

```
def ist_palindrom(zeichenkette):
    buchstabenliste=list(zeichenkette)
    buchstabenliste.reverse()
    return ("".join(1))
```

!!! Python

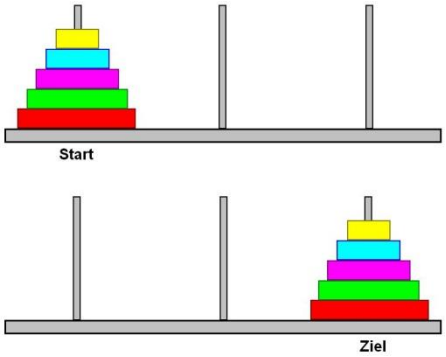
```
…
public (){

}
…
```

ist bei Zeitvergleichen die schnelleste Variante, weil die Listen- und String-Funktionen in Maschinensprache realisiert sind (*!!! Python*)

1.5.5.2.1.x. weitere klassische Rekursions-Probleme

Türme von Hanoi

Der Scheiben-Turm ist auf der Ziel-Position genau so wieder aufzubauen. Es darf immer jeweils nur eine Scheibe bewegt werden. Alle Scheiben müssen nach einem Zug (einer Scheiben-Bewegung) auf den Stangen (bzw. einem der drei Haufen) liegen. Es dürfen immer nur die kleineren Scheiben auf größeren liegen.



rekursive Zerlegung des aktuellen Turm in die größte / unterste Scheibe und einen kleineren (Rest-)Turm

ACKERMANN-Funktion

1926 von Wilhelm ACKERMANN beschrieben
wird zur Austestung von Speicher- und Computer-Modellen benutzt, da die Funktion extrem schnell wächst

ack(a, b, 0)	=	a + b	
ack(a, 0, n+1)	=	ack2(a, n)	→ Hilfsfunktion: ack2(?,?)
ack(a, b+1, n+1)	=	ack(a, ack(a, b, n+1), n)	

$$\text{ack2}(a, n) = \begin{cases} 0, & \text{wenn } n=0 \\ 1, & \text{wenn } n=1 \\ a, & \text{wenn } n>1 \end{cases}$$

durch PÈTER 1935 etwas einfacher definiert:

ack(0, m)	=	m+1
ack(n+1, 0)	=	ack(n, 1)
ack(n+1, m+1)	=	ack(n, ack(n+1, m))

rekursiv:

```
…
public int ackermann(int n, int m){
    if(n == 0){
        return m+1;
    } elseif (m == 0){
        return ackermann(n-1,1);
    } else {
        return ackermann(n-1,ackermann(n,m-1));
    }
}
…
```

teilweise interativ:

```
…
public ackermann(int n, int m){
    while(n != 0){
        if(m == 0){
            m = 1
        } else {
            m = ackermann(n, m-1);
        }
        n++;
    }
    return m+1;
}
…
```

Potenzierung von Zahlen

interativ

```
…
public int potenz(int basis, int exponent){
    int pot = 1;
    for(int i=0; i<=exponent; i++){
        pot *= basis;
    }
    return pot;
}
…
```

rekursiv

```
…
public int potenz(int basis, int exponent){
    if(exponent == 0){
        return 1;
    } else {
        return basis * potenz(basis, exponent-1);
    }
}
…
```

Quicksort

Quick-Sort ist ein Sortier-Algorithmus, der nach dem "Teile-und-herrsche"-Prinzip (lat.: Divide et impera!; engl.: divide and conquer) funktioniert.

Grundform um 1960 von HOARE vorgestellt

Vorteil des Algorithmus ist, dass innere Schleifen (Sortierungen) sehr kleinschritt und auch nur kleine Wiederholungszahlen haben
dadurch sehr schnell
gehört zu den schnellsten Sortier-Algorithmen

Prinzip des Algorithmus:

(1) zuerst wird ein mittig gelegenes Pivot-Element (Vergleichs-Element) ausgewählt
(2) die ursprüngliche Liste wird nun in eine linke und eine rechte Liste zerlegt
(3) in die linke Teil-Liste kommen die Elemente, die kleiner als das Pivot-Element sind; die anderen in die rechte Teil-Liste
(4) die Schritte (1) bis (3) werden mit den Teil-Listen solange wiederholt, bis diese die Listen-Länge 1 oder 0 haben, DANN gilt diese Teil-Liste als sortiert
(5) zusammensetzen der (sortierten) Gesamt-Liste aus den vorsortierten Teil-Listen

```
public class Quicksort {
  public static int[] datenFeld = {9, 4, 7, 8, 1, 3, 5, 6, 2, 9};

  public int[] sortieren(int links, int rechts) {
    int q;
    if (links < rechts) {
      q = teilen(links, rechts);
      sort(links, q);
      sort(q + 1, rechts);
    }
    return datenFeld;
  }

  private int teilen(int links, int rechts) {
    int i, j, x = datenFeld[(links + rechts) / 2];
    i = links - 1;
    j = rechts + 1;

    do {
      i++;
    } while (datenFeld[i] < x);

    do {
      j--;
    } while (datenFeld[j] > x);

    if (i < j) {
      int k = datenFeld[i];
      datenFeld [i] = datenFeld[j];
      datenFeld [j] = k;
    } else {
      return j;
    }
    return -1;
  }

…
```

```
public static void main(String[] args) {
  Quicksort quickSort = new Quicksort();
  int[] sortFeld = quickSort.sortieren(0, datenFeld.length - 1);
  for (int i = 0; i < sortFeld.length; i++) {
    System.out.println((i+1) + " --> " + sortFeld[i]);
  }
}
}
```

es wird im Feld selbst sortiert, es wird also kein zweites Feld benötigt
durch rekursive Aufrufe ist aber mit einem höheren Speicher-Bedarf zu rechnen

Mergesort

arbeitet nach dem "Teile-und-herrsche"-Prinzip
1945 von VON NEUMANN beschrieben
Liste wird in kleinere Listen zerlegt, die dann nach dem Reißverschluss-Verfahren zusammengefügt werden ´(verschmelzen, mischen – engl.: merge)

Prinzip des Algorithmus:
(1) aufteilen der Daten in zwei Hälften
(2)
(3)

```
public class Mergesort {
  public static int[] datenFeld = {9, 4, 7, 8, 1, 3, 5, 6, 2, 9};

  public int[] sort(int links, int rechts) {
    if (links < rechts) {
      int q = (links, rechts)/2;
      sort(links, q);
      sort(q + 1, rechts);
      merge(links,q,rechts);
    }
    return datenFeld;
  }

  private void merge(int links, int q, int rechts) {
    int[] hilfsFeld = new int[datenFeld.length];
    int i, j;
    for (i = links; i <= q; i++) {
      arr[i] = datenFeld [i];
    }
    for (j = q + 1; j <= rechts; j++) {
      hilfsFeld [rechts + q + 1 - j] = datenFeld [j];
    }
    i = links;
    j = rechts;
    for (int k = links; k <= rechts; k++) {
      if (hilfsFeld[i] <= hilfsFeld [j]) {
        datenFeld [k] = hilfsFeld [i];
        i++;
      } else {
        datenFeld [k] = hilfsFeld [j];
        j--;
      }
    }
  }

public static void main(String[] args) {
  Mergesort quickSort = new Mergesort();
  int[] sortFeld = mergeSort.sort(0, datenFeld.length - 1);
  for (int i = 0; i < sortFeld.length; i++) {
    System.out.println((i+1) + " --> " + sortFeld[i]);
  }
}
}
```

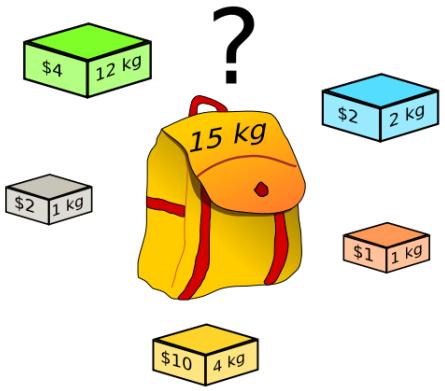
es wird zusätzlicher Speicherplatz benötigt

Rucksack-Problem

eng.: knapsack problem
Optimierungs-Problem aus der Kombinatorik

gegeben ist eine Menge von Objekten, die einen Nutzwert und ein Gewicht (Kostenfaktor) besitzen
gesucht ist eine Teilmenge, deren Gewicht eine bestimmte Grenze nicht überschreitet und der Nutzen aber maximiert sein soll

gehört zu den klassischen NP-vollständigen Problemen (Richard KARP (1972))



Veranschalichung des Rucksack-Problems
Q: de.wikipedia.org (Dake)

Zahlen-Beispiel von http://www-i1.informatik.rwth-aachen.de/~algorithmus/algo15.php

Objekt	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Gewicht	153	54	191	66	239	137	148	249
Profit	232	73	201	50	141	79	48	38

Gewichts-Schranke soll z.B. bei 645 liegen

0. trivialer Lösungs-Ansatz:
Wir hoffen, dass hinter einem großem Gewicht statistisch eben auch durchschnittlich mehr Profit steckt.
nehme die Objekte mit den größten Gewichten zuerst und fülle solange mit den jeweils leichteren auf bis die (Gewichts-)Grenze überschritten wird, nimm dann das leichteste Teil ab und probiere, ob weitere leichtere Teile passen, wiederhole dies solange, bis kein leicheteres mehr vorhanden ist
Als Lösung würde sich dabei die folgende Variante ergeben

Objekt	**8**	**5**	**3**	**1**	**7**	**6**	**4**	**2**
Gewicht (sort.)	249	239	191	153	148	137	66	54
Profit	38	141	201	232	48	79	50	73

Sum. Gew. **641**
Sum. Prof. **411**

Eventuell könnte man den Algorithmus auch noch vertiefen, indem man auch noch des Zweit-Leichteste entfernt und wieder mit den anderen auffüllt.
Als Gegen-Versuch könnte man genauso mit dem Profit als Sortier-Kriterium verfahren

Objekt	**1**	**3**	**3**	**6**	**2**	**4**	**7**	**8**
Gewicht	153	191	239	137	54	66	148	249
Profit(sort.)	232	201	141	79	73	50	48	38

Sum. Gew. **637**
Sum. Prof. **647**

Dieses Ergebnis ist deutlich besser, aber ob es optimal ist, bleibt unklar. Einen prinzipiell besseren Erfolg verspricht die Strategie mit einer Profit-Dichte zu arbeiten. In dieser stehen Gewicht und Profit in einem Quozienten bereit. Eine hohe Profit-Dichte verspricht einen großen Ertrag, bei geringerer Inverstition (hier Gewicht).

Objekt	1	2	3	4	5	6	7	8
Gewicht	153	54	191	66	239	137	148	249
Profit	232	73	201	50	141	79	48	38
Profit-Dichte	1,52	1,35	1,05	0,76	0,59	0,58	0,32	0,15

1. intuitiver Lösungs-Ansatz:
nehme die Objekte mit der höchsten Profit-Dichte, solange die (Gewichts-)Grenze nicht überschritten wird

Objekt	1	2	3	4	5	6	7	8	Anz. Obj.	4
Gewicht	153	54	191	66	239	137	148	249	**Sum. Gew.**	**464**
Profit	232	73	201	50	141	79	48	38	**Sum. Prof.**	**556**
Profit-Dichte	1,52	1,35	1,05	0,76	0,59	0,58	0,32	0,15	Schnitt PD	1,17

also → 1, 2, 3, 4 → Gewicht=464 → Profit=556

2. Lösung, wie 1. und Auffüllen mit weiteren passenden Objekten (entsprechend der Rangfolge)

Objekt	1	2	3	4	5	6	7	8	Anz. Obj.	4
Gewicht	153	54	191	66	239	137	148	249	**Sum. Gew.**	**601**
Profit	232	73	201	50	141	79	48	38	**Sum. Prof.**	**635**
Profit-Dichte	1,52	1,35	1,05	0,76	0,59	0,58	0,32	0,15	Schnitt PD	1,05

also → 1, 2, 3, 4, 6 → Gewicht=601 → Profit=635

→ aber ev. nicht optimal!
es muss jede Kombination ausprobiert werden!
interative Lösung
2^n Möglichkeiten (einpacken oder nichteinpacken / 1 oder 0)
Problem ist hier die expotentielle Steigerung des Rechen-Aufwandes
es gibt ev. auch mehrere Lösungen !?

Objekt	1	2	3	4	5	6	7	8	Anz. Obj.	4
Gewicht	153	54	191	66	239	137	148	249	**Sum. Gew.**	**637**
Profit	232	73	201	50	141	79	48	38	**Sum. Prof.**	**647**
Profit-Dichte	1,52	1,35	1,05	0,76	0,59	0,58	0,32	0,15	Schnitt PD	1,05

also → 1, 2, 3, 5 → Gewicht=637 → Profit=647

Interessanterweise haben wir diese Lösung mit unserem Trivial-Ansatz (2) über die sortierte Profit-Größe auch erhalten. Das ist aber Zufall.

besser ist der Algorithmus von NEMHAUSER und ULLMANN (1969)
basiert auf PARETO-Prinzip

Prinzip des Algorithmus:
(1)

→ https://books.google.de/books?id=kAMeBAAAQBAJ&pg=PA409&lpg=PA409&dq=rucksackproblem+nemhauser+ullmann&source=bl&ots=8377634wjq&sig=v8BjkmUYY1FoRGcl9XeatV9z-A8&hl=de&sa=X&ved=0ahUKEwifusTR143UAhUK7xQKHacyCtY4ChDoAQhBMAU#v=onepage&q=rucksackproblem%20nemhauser%20ullmann&f=false

Suche in einem Baum

Alpha-Beta-Suche für Spielzüge bei Brettspielen (Computer-Stategie)

Alpha-Beta-Cut od. Alpha-Beta-Pruning

Optimierung / Einschränkung des Minimax-Algorithmus
dieser durchsucht normalerweise den gesamten Baum
bei de Alpha-Beta-Suche werden Zweige, die keinen Erfolg mehr bieten ganz ausgeschlossen (abgeschnitten) werden

während der Suche werden die Werte alpha und beta, die für die optimaler Ergebnisse für beide Spieler stehen, ständig aktualisiert
Zweige, die keinen Vorteil bringen werden dann von der weiteren Suche ausgeschlossen

Weg aus einem Labyrint

Rechte-Hand-Regel

geht natürlich auch als Linke-Hand-Regel (schon wegen der Gleichberechtigung von Links-Händern! (;-))

Permutationen

```
…
public (){

}
…
```

effektive Speicherung von Daten (z.B. Bilder)

Wollten wir das nebenstehende Bit-Muster / Bild über eine Liste abspeichern, dann würde diese mit 64 Elementen doch recht lang werden. Nehmen wir an, es geht oben links los und es wird Zeilen-weise gearbeitet, dann ergibt sich die folgende Liste:

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

muster=[1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1,
1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0,
0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0,
0, 0, 1, 1, 1, 1, 0, 0, 0, 0]

Das Bild / Daten-Muster wird zuerst als das größtmögliche Quadrat (gleicher Elemente) betrachtet. Wäre es z.B. homogen nur mit Einsen gefüllt, dann würde sich als Muster die Liste **muster=[1]** ergeben. Statt der 64 Speicher-Elemente hätten wir es nur noch mit einem zu tun. Wir bräuchten also grob 1/64 des Speicherbedarfs – wenn das keine Kompression ist?!
Das Muster ist aber heterogen, also müssen wir es verkleinern.
Die verkleinerten Quadrate sind umrandet hervorgehoben. Für jedes der kleineren Quadrate müssen wir nun in unserer muster-Liste ein Listen-Element verwenden. Da es mehr als eins ist, wird klar, dass das gesamte 8x8-Quadrat strukturiert ist. Die Grundstruktur sieht dann so aus:

muster=[, , ,]

Wären jetzt die kleineren (4x4)-Quadrate einheitlich gefärbt (mit 1 oder 0 belegt), dann würden wir mit 4 Listen-Elementen hinkommen, was immer noch einer Effektivität der Kompression von 4/64 entsprechen würde. Aber leider ist es im Beispiel nicht so, also müssen wir genauer weiter differenzieren.

Das oberste linke Quadrat ist vollständig mit Einsen gefüllt, also speichern wir uns in die Teil-Liste eine Eins:

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

muster=[1, , ,]

Nun wechseln wir zum rechten oberen 4x4-Quadrat. Es ist nicht homogen und muss deshalb wieder unterteilt werden. Es entsteht also eine Liste (**rot gekennzeichnet**) in der Liste (Das Listen-Element ist selbst wieder eine Liste).

muster=[1, **[, , ,]**, ,]

Die sich ergebenden 2x2-Quadrate sind homogen, also kann die Muster-Liste nun so geschrieben werden:

muster=[1, **[0, 0, 0, 1]**, ,]

Das untere, linke Quadrat ist nicht homogen, also muss es zerlegt werden. Auch das oberste linke 2x2-Quadrat ist nicht homogen, also muss es als Bit-Muster in die Liste geschrieben werden.

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

muster=[1, [0, 0, 0, 1], [[0, 1, 1, 0], , ,],]

Das zweite 2x2-Quadrat ist homogen mit Nullen belegt, also speichern wir ein 0 in die Liste.

muster=[1, [0, 0, 0, 1], [[0, 1, 1, 0], 0, ,],]

Bei den unterern beiden 2x2-Quadraten verfahren wir in der gleichen Weise und erhalten dann:

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

muster=[1, [0, 0, 0, 1], [[0, 1, 1, 0], 0, [1, 0, 1, 1], 1],]

Bleibt das letzte (untere, rechte) 4x4-Quadrat. Es ist homogen, so dass die Liste nur die darin enthaltene Null repräsentieren muss:

muster=[1, [0, 0, 0, 1], [[0, 1, 1, 0], 0, [1, 0, 1, 1], 1], 0]

Im Vergleich zur obigen Voll-Liste kommen wir nun mit 26 Speicher-Elementen aus. Das bedeutet eine Verbesserung fast um den Faktor 2,5 (grob: 16/64).

Die Kompressionsraten sind sehr theoretisch berechnet. Es muss beachtet werden, dass noch Strukturierungs-Elemente (zur Unterscheidung von über- und unter-geordneten Listen) mit abgespeichert werden müssen.
So ähnlich – wie hier besprochen – laufen z.B. Kompressions-Verfahren, wie das JPEG oder MP4 ab. Neben dem Vergleich der Bild-Elemente werden auch noch die vorlaufenden Bilder mit verglichen. Dabei nutzt man den Effekt aus, dass sich in einer Bildfolge meist nur wenige – isolierte – Teile verändern.

Aufgaben:

1. ***Übernehmen Sie das Muster und Muster-Liste! Kennzeichen Sie durch unterschiedlich farbige Umrandungen im Muster und durch entsprechend farbige Klammern, welche Bitmuster zu welchen Listen-Elementen gehören!***

Eine Rekursion bietet sich immer dann an, wenn das Problem / die Funktion schrittweise auf ein kleineres / leichteres Problem // eine einfachere Funktion reduziert werden kann.

Volumen-Berechnung einer n-dimensionalen Hyperkugel

McCARTHYS "91-Funktion"

$$f(n) = \begin{cases} n-10 & \text{falls } n > 100 \\ f(f(n+11)) & \text{sonst} \end{cases}$$

PELL-Folge

$$P(n) = \begin{cases} 0, & \text{falls } n = 0 & \textit{Rekursions-Anfang} \\ 1, & \text{falls } n = 1 & \textit{Rekursions-Anfang} \\ 2P(n-1) + P(n-2) & \text{sonst} & \textit{Rekursions-Schritt} \end{cases}$$

erste Elemente: 0, 1, 2, 5, 12, 29, 70, 169, 408, …

die ersten beiden Elemente sind mit 0 und 1 definiert
die nachfolgenden Elemente ergeben sich als Summe aus dem verdoppelten Vorgänger und dem (einfachen) Vorvorgänger

PELL-Folge 2. Art

$$Q(n) = \begin{cases} 2, & \text{falls } n = 0 \quad \textit{Rekursions-Anfang} \\ 2, & \text{falls } n = 1 \quad \textit{Rekursions-Anfang} \\ 2Q(n-1)+Q(n-2) & \text{sonst} \quad \textit{Rekursions-Schritt} \end{cases}$$

erste Elemente: 2, 2, 6, 14, 34, 82, 198, 478, 1154, …

die ersten beiden Elemente sind mit 2 definiert
die nachfolgenden Elemente ergeben sich als Summe aus dem verdoppelten Vorgänger und dem (einfachen) Vorvorgänger

LUCAS-Folge(n)

$$L(n) = \begin{cases} x, & \text{falls } n = 0 \quad \textit{Rekursions-Anfang} \\ y, & \text{falls } n = 1 \quad \textit{Rekursions-Anfang} \\ L(n-1)+L(n-2) & \text{sonst} \quad \textit{Rekursions-Schritt} \end{cases}$$

erste Elemente: immer abhängig von x und y
bei x=2 und y=1 → 2, 1, 3, 4, 7, 11, 18, 29, 47, …

die ersten beiden Elemente sind mit x und y definiert
die nachfolgenden Elemente ergeben sich als Summe aus dem Vorgänger und dem Vorvorgänger

JACOBSTHAL-Folge

$$J(n) = \begin{cases} 0, & \text{falls } n = 0 \quad \textit{Rekursions-Anfang} \\ 1, & \text{falls } n = 1 \quad \textit{Rekursions-Anfang} \\ J(n-1)+2J(n-2) & \text{sonst} \quad \textit{Rekursions-Schritt} \end{cases}$$

n:	0	1	2	3	4	5	6	7	8	9	10	11	12
erste Elemente :	0,	1,	1,	3,	5,	11,	21,	43,	85,	171,	341,	683,	…

die ersten beiden Elemente sind mit 0 und 1 definiert
die nachfolgenden Elemente ergeben sich als Summe aus dem Vorgänger und dem verdoppelten Vorvorgänger

RECAMÀNs-Folge
(OEIS → A005132)

R(n) = {	0,	falls n = 0	*Rekursions-Anfang*
	R(n-1)-n	falls R(n) >= 0 und nicht in Sequenz	*Rekursions-Schritt*
	R(n-1)+n	sonst	*Rekursions-Schritt*

n:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
erste Elemente :	0,	1,	3,	6,	2,	7,	13,	20,	12,	21,	11,	22,	10,	23,	9,	24,	8,	25,	43, …

das erste Element ist mit 0 definiert
die nachfolgende Elemente gilt: ein Element berechnet sich aus durch Subtraktion der Glied-Nummer (n) von dem Reihen-Vorgänger; ist der Wert positiv und kommt noch nicht in der Reihe vor, dann wird der berechnete Wert in die Reihe aufgenommen
ansonsten berechnet sich das Gleid aus dem Vorgänger und der Addition der Glied-Nummer

interessante Links:
https://oeis.org/wiki/Welcome (On-Line Encyklopedia of Integer Sequences ® OEIS ®)
https://oeis.org/A?????? (Informationen zur Folge mit der Nummer ??????)

Aufgaben für die gehobene Anspruchsebene:

1. Informieren Sie sich zur Biographie von N.J.A. SLOANE!
2. Was verbirgt sich hinter der Folge A000108?

NUR!!! zum Üben: die DREWS-Folgen

Nicht wundern, natürlich gibt es diese Folge nicht wirklich. Sie sind praktisch abgewandelte FIBONACCHI-Folgen. Bei der ersten Folge wird immer eine 1 dazuzählt. Also typisch DREWS – immer noch Einen drauf setzen. Die Folgen haben keinen Zweck, außer dem Programmieren zu dienen.
(Da sie auch nicht so bekannt und sinnvoll verwendbar sind, gibt es auch kaum schnell zu findende google-Lösungen – da muss dann doch wirklich selbst programmiert werden!)

0, 1, 2, 4, 7, 12, 20, 33, 54, 88, 143, 232, …

dre(n) = {	0,	falls n = 0	*Rekursions-Anfang*
	1,	falls n = 1	
	dre(n-1) + dre(n-2) + 1,	sonst	*Rekursions-Schritt*

Die zweite Folge ist etwas komplexer. Hier unterscheidet sich das Zuzählen danach, ob die Gliednummer gerade oder ungerade ist.

0, 1, 2, 5, 8, 15, 24, 41, 66, 109, 176, 287, …

dre2(n) = {	0,	falls n = 0	*Rekursions-Anfang*
	2,	falls n = 1	
	dre2(n-1) + dre2(n-2) + 1,	sonst, falls n gerade	*Rekursions-Schritte*
	dre2(n-1) + dre2(n-2) + 2	sonst, falls n ungerade	

Aufgaben:

1. ***Programmieren Sie die 1. DREWS-Folge als rekursive Funktion mit einem kleinen Rahmen-Programm!***
2. ***Erstellen Sie ein Programm, mit dem die DREWS2-Folge sowohl rekursiv als auch interativ berechnet wird!***

1.5.5.2.2. direkte Gegenüberstellung von interativen und rekursiven Algorithmen

In der Literatur und in der täglichen Programmierarbeit finden sich bzw. entstehen die unterschiedlichsten Umsetzungen von bestimmten Problem. Einige sind hier gesammelt und jeder Programmierer wird nach und nach den einen oder anderen Programm-Text hinzutuen können. Ob die einzelnen Lösungen immer optimal (gut lesbar, schnell, wenig Speicherbedarf, …) sind, wird hier nicht bewertet. Sollten Algorithmen entscheidend für ein Programm sein, dann müssen spezielle Test (Abfrage Speicherbedarf, Zeitmessungen, …) erfolgen. Auf einige Möglichkeiten gehen wir noch ein.
In einigen Algorithmen sind **blau**e **print**-Anweisungen eingebaut. Diese dienen als optionale Ausgabe, um das Arbeiten des Algorithmus zu verfolgen. Für echte Anwendungen sollten sie dann raus genommen werden. Ev. lassen sich weitere sinnvolle Ausgaben erzeugen, z.B. um die Anzahl der Schleifendurchläufe bzw. die Rekursion-Aufrufe zu zählen. Dafür müssen dann aber eigene Variablen und Zähl-Anweisungen eingebaut werden.
Bei einigen ausgewählen Algorithmen-Umsetzungen notieren wir die zusätzliche Schleifenzählung in **rot**er Farbe. Auch diese Quelltext-Teile sollten vor dem echten Einsatz entfernt oder auskommentiert werden.

1.5.5.2.2.1. GGT – größter gemeinsamer Teiler

!!!Achtung: hier noch Python-Code!

Lösung	interativ	rekursiv

1	`def ggt(a,b):`
	`  i = 0`
	`  while b > 0:`
	`  i += 1`
	`  print("Durchlauf: ",i)`
	`  print("a= ",a,"b= ",b)`
	`  r = a % b`
	`  a = b`
	`  b = r`
	`  return a`
		`def ggt(a,b,i):`
		`  i += 1`
		`  print("Aufruf: ",i)`
		`  print("a= ",a,"b= ",b)`
		`  if b == 0:`
		`  return a`
		`  return ggt(b, a % b,i)`
		
		`# Aufruf der Funktion:`
		`i = 0`
		`ggt(a,b,i)`

2	`def ggt(a,b):`
	`  while b != 0:`
	`  a,b = b, a % b`
	`  return a`
		`def ggt(a,b):`
		`  return if a == b:`
		`  a`
		`  elif a > b:`
		`  ggt(a-b,b)`
		`  else:`
		`  ggt(a, b-a)`

3	`def ggt(a,b):`
	`  while a != b:`
	`  if a > b:`
	`  a=a-b`
	`  else:`
	`  b=b-a`
	`  return a`
		`def ggt(a,b):`
		`  return if a > b:`

1.5.5.2.2.2. Palindrom-Prüfung

!!!Achtung: hier noch Python-Code!

Lösung	interativ	rekursiv

1	`def ist_palim(s):`
	`  links = 0`
	`  rechts = len(s)-1`
	`  while links < rechts:`
	`  if s[links] != s[rechts]:`
	`  return 0`
	`  links += 1`
	`  rechts -= 1`
	`  return 1`
		`def ist_palim(s):`
		`  if len(s) <= 1:`
		`  return 1`
		`  if s[0] != s[-1]:`
		`  return 0`
		`  return ist_palim(s[1:-1])`

2 außer Konkurrenz	`def ist_palim(s):`
	`  liste = list[s]`
	`  liste.reverse()`
	`  return ("".join(liste))`	

1.5.5.2.2.2. Potenz-Prüfung

ist p eine ganzzahlige Potenz von x

!!!Achtung: hier noch Python-Code!

Lösung	interativ	rekursiv

1	`def istpotenz(p,x):`
	`  return if p == 1 or p == x`
	`  or p % x != 0:`
	`  p == 1 or p == x`
	`  else:`
	`  istpotenz(p/x,x)`
		`def istpotenz(p,x):`
		`  return if p == 1 or p == x:`
		`  True`
		`  elif p % x !=0:`
		`  False`
		`  else:`
		`  istpotenz(p/x,x)`

2	`def istpotenz(p,x):`
	`  while p != 1 and p != x`
	`  and p % x == 0:`
	`  p = p / x`
	`  return p == 1 or p == x`	

1.5.5.2.2.3. Fakultät

berechnen von x!, also dem Produkt der Ganzahlen von 1 bis x

Lösung	interativ	rekursiv
1		

2	`static int fakultaet(int x){`
	`  int erg=1;`
	`  for (int i=1;i<=x;i++){`
	`  erg *= i`
	`  }`
	`  return erg;`
	`}`
		`static int fakultaet(int x){`
		`  if (x==1){`
		`  return 1;`
		`  } else {`
		`  return fakultaet(x-1)*x;`
		`  }`
		`}`

1.5.5.2.2.4. FIBONACCHI-Folge

berechnen der klassischen FIBONACCHI-Folge

Lösung	interativ	rekursiv
1		

2	`static int fibonacchi(int x){`
	`  int x0=0;`
	`  int x1=1;`
	`  if (x<=0){`
	`  return x0;`
	`  } else if (x==1){`
	`  return x1;`
	`  } else {`
	`  int h=0;`
	`  int i=2;`
	`  while (i<=x){`
	`  h=x0+x1;`
	`  x0=x1;`
	`  x1=h;`
	`  i++;`
	`  }`
	`  return x1;`
	`  }`
	`}`
		`static int fibonacchi(int x){`
		`  if (x<=0){`
		`  return 0;`
		`  } else if (x==1){`
		`  return 1;`
		`  } else {`
		`  return fibonacchi(x-2)`
		`  +fibonacchi(x-1);`
		`  }`
		`}`

1.5.6. String-Methoden

Das Arbeiten mit Strings nimmt im täglichen Programmieren vielmehr Raum ein, als man zuerst glaubt. Dazu kommt, dass das Arbeiten mit den Zeichenketten nicht immer trivial ist. Da sind schnelle, effektive und gut getestete Methoden gefragt. Zu unserem Glück werden eine Vielzahl solcher Methoden schon vorgefertigt vom JAVA-System bereitgestellt.
Wer will kann ja mal seine eigene Methode zu einem Text-Manipulations-Problem schreiben und dann mit der fertigen Version von JAVA vergleichend laufen lassen. Da werden Abgründe sichtbar. Die JAVA-eigenen Methoden sind sehr Maschinen-nah in der Programmiersprache C geschrieben. Manche von ihnen wohl auch direkt in Maschinen-Sprache (Assembler). Das ist eine andere Liga in der Programmierung.

Stringname.**toUpperCase();**
wandelt die Text-Ausgabe vollständig in Groß-Buchstaben um

Stringname.**toLowerCase();**
wandelt die Text-Ausgabe vollständig in Klein-Buchstaben um

Stringname.**reverse();**
gibt die Zeichen des Strings in umgekehrter Reihenfolge aus

```
String reversedString = new StringBuffer(orginalString).reverse().toString();
```

Stringname.**charAt(*Position*);**
liefert das Zeichen an der angegebenen Position
die Zählung der Zeichen in einer Zeichenkette (String) beginnt in JAVA mit 0

```
…
String test = "Beispieltext";
test.charAt(0);
…
```

liefert also das Zeichen 'B' zurück

Stringname.**indexOf(*zeichen*);**
gibt die Index-Position des ersten Auftretens von zeichen zurück
zeichen kann auch als Integer-Wert (dezimaler ASCII-Wert) angegeben werden

Stringname.**substring(AnfangPosition,EndPosition);**
liefert den Teil-String von Stringname zurück der durch die Index-Positionen AnfangPosition und EndPosition eingeschlossen ist

Stringname.**length();**
gibt die Anzahl der Zeichen im String zurück; praktisch also die String-Länge

Stringname.**equals(*objekt*);**
vergleicht den String, der Stringname zugeordnet ist mit dem Objekt objekt

wie bei equals meist gemeint kommt es zum Vergleich der Zeiger (von Stringname und objekt)
es wir true oder false zurückgegeben

Stringname.**compareTo(*Vergleichsstring*);**
vergleicht den String Stringname mit dem Vergleichsstring
der Integer-Rückgabewert steht für

Stringname.**contains(*Zeichenkette*);**
gibt true zurück, wenn Zeichenkette im String Stringname (irgendwo) enthalten ist

Stringname.**replace(*altesZeichen, neuesZeichen*);**
ersetzt im String Stringname alle auftretenden alten Zeichen durch das angegebene neue Zeichen

Stringname.**Concat(*Zeichenkette*);**
hängt Zeichenkette an den existierenden String Stringname an

Stringname.**endsWith(*Suffix*);**
prüft, ob Stringname mit Suffix endet
z.B. zum Prüfen, ob ein Dateiname mit dem typischen Dateityp endet (z.B.: "**.TXT**"

Stringname.**toCharArray();**
der String wird in eine Char-Array umgewandelt

Stringname.**valueOf(*Parameter*);**
stellt den Parameter als String dar

1.5.7. Laufzeit- und Speicher-Effizenz

Bei unseren kleinen Übungs-Programmen werden wir kaum an die Grenzen unseres Computers stoßen. Es scheint so, als würden die Programme immer extrem schnell ablaufen. Welchen Speicher sie verbrauchen, ist für uns so einfach kaum ersichtlich.
Was passiert nun aber, wenn man statt vielleicht 10 Schleifen-Durchläufen 1'000'000 macht? Oder wie wirkt sich das auf den Speicher-Verbrauch aus, wenn ich statt 100 Messwerten 1'000'000 auswerten will. Laufen unsere Programmen dann immer noch so unproblematisch?
Also, spätestens bei solchen Umgebungs-Bedingungen muss man sich Gedanken um die Gestaltung effektiver Programme machen.
Wir zeigen hier einzelne Aspekte auf. Die sind nur geringfügig sortiert und auch nicht nach Relevanz geordnet. Das würde immer vom jeweiligen Programm abhängen. Die einzelnen Themen sind kurz überschrieben, um vielleicht schnell passende zu finden oder ungebrauchte zu überspringen.
Ich empfehle es jedem Programmierer, sich mindestens einmal mit allen zu beschäftigen. Dadurch entwickelt man ein Gefühl für die Probleme, welche auftauchen können. Bei aktueller Relevanz kann man sich dann einzelne Aspekte wiederholend genauer ansehen.

Die nachfolgende Situation kann in JAVA schnell mal auftreten, weil in JAVA gerne erst an der Stelle definiert werden, wenn sie gebraucht werden. Das hat was, weil man direkt dort nachschauen kann, um welchen Daten-Typ es sich handelt. Ungünstig kann es werden, wenn die Definition in eine Schleife mit hinein gelangt.

Anlegen von Variablen

```
…
int sum = 0;
for (int i = 0; i <1000; i++){
    int z = 0;
    z = i * i;
    for (int j = 0; j < 1000; j++) {
        int y = 0;
        y = i * j;
        sum += y + z;
    }
}
…
```

Bewirkt, dass z.B. die Variable z 1'000x angelegt wird. Noch dramatischer ist dies für y, da kommt man praktisch auf 1'000'000x.
Günstiger ist das Anlegen der Variablen vor den Schleifen. Dadurch werden die Variablen nur einmalig angelegt.

```
…
int sum = 0;
int z = 0;
int y = 0;
for (int i = 0; i <1000; i++){
    z = i * i;
    for (int j = 0; j < 1000; j++) {
        y = i * j;
        sum += y + z;
    }
}
…
```

mehrfacher Aufruf von gleichen Methoden

Der folgende Code-Schnipsel ruft mehrfach die Methode (Funktion) Math.sqrt auf.

```
…
double[] [] fkt 0 new double [1000] [4];
for (int i = 1; i<=1000; i++) {
    fkt [i] [0] = (double) i;
    fkt [i] [1] = Math.sqrt(i),
    fkt [i] [3] = 1 / Math.sqrt(i);
    fkt [i] [4] = (Math.sqrt(i) + Math.sqrt(i)) / Math.sqrt(i);
    System.out.println("i = "+i+" Wurzel= "+Math.sqrt(i)
                      +"  rez. Wrzl= "+1/Math.sqrt(i)
                      +"  2   Wrzl/Wrzl= "+(Math.sqrt(i)
                      + Math.sqrt(i))/Math.sqrt(i));
}
…
```

Bei dieser Schleife wird bei jedem Schleifen-Durchlauf 10x die Quadratwurzel-Funktion aufgerufen. Diese ist nicht unbedingt eine billige Funktion, d.h. sie ist relativ rechen-aufwändig. Günstiger ist der einmalige Aufruf am Anfang der Schleife und das Speichern in einer Variable. Im gleichen Schleifen-Durchlauf wird nun auf die Variable zugegriffen, was deutlich schneller geht.

```
…
double[] [] fkt 0 new double [1000] [4];
double wrzl = 0.0;
for (int i = 1; i<=1000; i++) {
    wrzl = Math.sqrt(i);
    fkt [i] [0] = (double) i;
    fkt [i] [1] = wrzl,
    fkt [i] [3] = 1 / wrzl;
    fkt [i] [4] = (wrzl + wrzl) / wrzl;
    System.out.println("i = "+i+" Wurzel= "+wrzl+"  rez. Wrzl= "+1/wrzl
                      +"  2   Wrzl/Wrzl= "+(wrzl+ wrzl)/wrzl);
}
…
```

Short Circuit Evaluation (logische (Kurz-(Schlüsse))

Für die Aneinander-Reihung von Operationen ist es immer gut, zu prüfen, welche Bedingungen im Falle einer UND-Verknüpfung sehr wahrscheinlich eher falsch sein werden. Diese sollten dann weiter nach vorne in die Bedingung gestellt werden, damit sie schnellstmöglich eine Short Circuit Evaluation auslösen können. Bei ODER-Verknüpfungen sollte man die eher wahren Bedingungen zuerst bearbeiten lassen.

? Beispiel

???doppelt:

Im Falle einer UND-Verknüpfung kann die laufende Verarbeitung dieser Verknüpfung sofort beendet werden, wenn einmal ein FALSCH auftritt. Die nachfolgenden Teile der Verknüpfung brauchen gar nicht mehr geprüft werden.
Ähnlich liegt der Fall bei eine ODER-Verknüpfung. Hier kann die Prüfung abgebrochen werden, wenn eine WAHR-Aussage einhalten ist.
Die meisten Compiler / Interpreter von Programmiersprachen beachteten solche Gegebenheiten, um den Programm-Ablauf zu beschleunigen.
Deshalb ist es immer sinnvoll die trivialen Tests weiter vorne in eine logische Verknüpfung zu legen und die speziellen / seltenen Fälle weiter nach hinten.
Bei komplexeren logischen Strukturen kann man auch einfache Lauftests mit Zeitmessungen oder das Zählen von Schleifendurchläufen als Bewertungs-Kriterien benutzen.

1.5.8. JAVA-Schlüsselwörter (erweiteter Wortschatz)

abstract

Syntax	Beschreibung
	definiert eine abstrakte Klasse und / oder Methode die Klasse selbst kann nicht instanziert werden (dazu muss erst eine Sub-Klasse (beruhend auf dieser abstarkten Klasse) erstellt werden

Beispiel(e)	Kommentar(e)

class

Syntax	Beschreibung
	defniert eine Objekt-Klasse (Typ-Beschreibung von Objekten)

Beispiel(e)	Kommentar(e)

extends

Syntax	Beschreibung
`class SubKlasse` **`extends`** `SuperKlasse { …`	besagt, das die eine Klasse (SubKlasse) eine andere Klasse (SuperKlasse) erweitert

Beispiel(e)	Kommentar(e)

false

Syntax	Beschreibung
`false`	Wert für Nicht-WAHR also FALSCH

Beispiel(e)	Kommentar(e)

new

Syntax	Beschreibung
	erstellen einer Instanz von einer Objekt-Klasse (instanzieren eines neuen Objektes einer Klasse)

Beispiel(e)	Kommentar(e)

private

Syntax	Beschreibung
	einleitendes Schlüsselwort (Sichtbarkeits-Modifikator) zur Festlegung der Sichtbarkeit der folgenden Ausdrücke in anderen Klassen und Paketen **die folgenden Ausdrücke sind nur innerhalb der eigenen Klasse benutzbar**

Beispiel(e)	Kommentar(e)

protected

Syntax	Beschreibung
	einleitendes Schlüsselwort (Sichtbarkeits-Modifikator) zur Festlegung der Sichtbarkeit der folgenden Ausdrücke in anderen Klassen und Paketen **die folgenden Ausdrücke sind vor dem Benutzen aus geschützt**

Beispiel(e)	Kommentar(e)

public

Syntax	Beschreibung
	einleitendes Schlüsselwort (Sichtbarkeits-Modifikator) zur Festlegung der Sichtbarkeit der folgenden Ausdrücke in anderen Klassen und Paketen **die folgenden Ausdrücke sind uneingeschränkt sichtbar**

Beispiel(e)	Kommentar(e)

return

Syntax	Beschreibung
	gibt Werte zurück Argument-behafteter Rücksprung aus einer Methode / Funktion

Beispiel(e)	Kommentar(e)

static

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

super.

Syntax	Beschreibung
super.*Methodenname*(*ParameterListe*);	ermöglicht den Zugriff auf die Methoden der (namentlich nicht bekannten od. allgemeingehalten) Oberklasse

Beispiel(e)	Kommentar(e)

super()

Syntax	Beschreibung
super(*ParameterListe*)	Aufruf des Konstruktors aus der Superklasse im lokalen Konstruktor (muss erste Anweisung innerhalb des lokalen Konstruktor sein!)

Beispiel(e)	Kommentar(e)

this.

Syntax	Beschreibung
	meint die eigene Instanz ermöglicht den Zugriff auf die Methoden und attribute innerhalb der eigenen Klasse teilweise optional, aber unbedingt notwendig, wenn Attribute und Parameter den gleichen Bezeichner haben!

Beispiel(e)	Kommentar(e)

this()

Syntax	Beschreibung
	Aufruf des überladenen Konstruktors aus eigenen Klasse (muss erste Anweisung innerhalb des Konstruktor sein!)

Beispiel(e)	Kommentar(e)

true

Syntax	Beschreibung
true	Wert für WAHR (bzw. Nicht-FALSCH)

Beispiel(e)	Kommentar(e)

void

Syntax	Beschreibung
	Typ-freie Methode (es gibt keinen Rückgabe-Wert!)

Beispiel(e)	Kommentar(e)

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

1.5.9. komplexe Programmier-Aufgaben:

Wählen Sie eine geeignete oder Ihre präferierte Programmiersprache zur Lösung der nachfolgenden Aufgaben aus!
Überlegen Sie sich bzw. vergleichen mit anderen, ob die von Ihnen präferierte programmiersprache gut geeignet ist das gewählte Problem zu lösen!
Zahlen-Eigenschaften nach: www.zahlen.mathematic.de

Aufgaben:

1. *Berechnen Sie die Summe und das Produkt einer Reihe von einzugebener Zahlen sowie Summe und Produkt der reziproken Werte!*
2. *Erstellen Sie ein Programm, dass im Zahlen-Raum bis zur einer einzugebenen (größeren) natürlichen Zahl, die Kombination von drei aufeinanderfolgenden Primzahlen findet, deren Produkt möglichst dicht an der Zahlen-Grenze liegt!*
3. *Prüfen Sie ob eine als Zeichen-String vorgegebene Zahl (ohne Leer- und Vorzeichen bzw. Nachkommastellen) im auszuwählenden Zahlensystem gültig ist! (Die Ziffern werden als ASCII-Zeichen notiert. Gültige und unterscheidbare Zeichen sind: 0 .. 1 A .. Z a .. z → das sollte auch bis zum Sexagesimal-System reichen! Doppeldeutung A = a muss nicht beachtet werden!)*
4. *Lassen Sie durch eine Erweiterung des Programms von 3. prüfen, ob es sich bei der eingegeben Zahl um eine normale Zahl handelt! Normale Zahlen enthalten alle Ziffern ihres Alphabetes mit der gleichen Häufigkeit.*
5. *Erstellen Sie das Programm "Zahlen-Charakterisierer"! Das Programm soll eine einzugebene ganze Zahl (ev. zuerst nur für natürliche Zahlen) Charakter-Eigenschaften prüfen bzw. bestimmen und ausgeben, ob die Zahl die Eigenschaft hat oder nicht bzw. den berechneten Wert. Das Programm sollte später um weitere Zahlen-Eigenschaften ergänzt werden können und passend kommentiert sein! Auf die (spätere) Nutzbarkeit von Unterprogrammen ist zu achten! Wählen Sie sich mindestens 12 Eigenschaften aus! Die Reihenfolge kann frei geändert werden!*
   - *a) männliche Zahl (Zahl ist ungerade und größer als 1)*
   - *b) Quersumme (ist die Summe der einzelnen Ziffern der Zahl (ohne deren Potenzwert))*
   - *c) titanische Zahl (ist eine Primzahl mit mindestens 1000 Stellen*
   - *d) weibliche Zahl (Zahl ist eine positive gerade Zahl)*
   - *e) Totient od. Indikator (ist die Anzahl der Primzahlen, die kleiner als die (gegebene) Zahl ist)*
   - *f) zusammengesetzte (od. zerlegbare od. teilbare) Zahl (ist eine Zahl, die mehr als zwei positive Teiler hat ODER eine gerade Zahl, die größer als 1 ist)*
   - *g) abundante Zahl (wenn echte Teilersumme (Summe aller Teiler (ohne Rest), außer die Zahl selbst) größer als die Zahl selbst ist)*
   - *h) arme od. defizierte od. mangelhafte Zahl (wenn echte Teilersumme kleiner als das doppelte der Zahl ist)*

i) vollkommene od. perfekte Zahl (wenn die echte Teilersumme gleich der Zahl selbst ist)

j) Sophie-GERMAIN-Primzahl (sind Primzahlen, bei denen der Term 2 p +1 wieder eine Primzahl ist)

k) reiche od. überschießende od. übervollständige Zahl (wenn die echte Teilersumme größer als das Doppelte der Zahl selbst ist)

l) SMITH-Zahl (wenn die Quersumme der Zahl gleich der Quersummen ihrer Primfaktoren ist; außer Primzahlen!)

m) erhabene Zahl (wenn Zahl und deren echte Teilersumme vollkommene Zahlen sind)

n) palindrome Zahl (wenn die Zahl und die umgedrehte Ziffernfolge gleich (groß) sind)

o) palindrome Primzahl (wenn Zahl eine Primzahl ist und die Zahl und deren umgedrehte Ziffernfolge gleich sind)

p) SIERPINSKI-Zahl (ist eine ungerade, natürliche Zahl n, bei der der Term $n\ 2^x + 1$ immer eine zusammengesetzte Zahl ergibt (x ist eine beliebige natürliche Zahl))

q) RIESEL-Zahl (sind ungerade, natürliche Zahlen, bei denen der Term $n\ 2^x - 1$ immer eine zusammengesetzte Zahl ergibt (x ist eine beliebige natürliche Zahl))

r) strobogrammatische Zahl (ist eine Zahl, die um 180° gedreht wieder die gleiche Zahl ergibt (hier gelten 1, 2 mit 5, 6 mit 9, 8 und 0 als drehbare oder strobogrammatische Ziffern))

s) strobogrammatische Primzahl (ist eine Primzahl, die auch strobogrammatisch ist)

6. Gesucht wird ein modulares Programm, dass für zwei natürliche Zahlen prüft, ob es sich um ein Paar mit den folgenden Eigenschaften handelt!

a) befreundete Zahlen (wenn die echten Teilersummen beider Zahlen gleich sind))

b) Primzahlen-Zwilling (wenn zwei aufeinanderfolgende Primzahlen eine Differenz von 2 aufweisen)

c) Teiler-fremde (od. inkommensurable) Zahlen (ganze Zahlen, die außer -1 und 1 keine gemeinsamen Teiler besitzen)

7. Gesucht wird ein modulares Programm, dass für drei natürliche Zahlen prüft, ob es sich um ein Tripel mit den folgenden Eigenschaften handelt!

a) pythagoreische Zahlen (Tripel erfüllt die diophantische Gleichung 2. Grades ($a^2 + b^2 = c^2$))

b) Primzahlen-Drilling (wenn drei aufeinanderfolgende Zahlen die Reihe p, p+2, p+6 bilden ODER wenn innerhalb einer Dekade (also 10 aufeinanderfolgenden Zahlen) drei Primzahlen vorkommen)

8.

1.6. Objekt-orientierte Programmierung für Erfahrene

Hier beschäftigen wir uns noch einmal ein bißchen genauer mit den Techniken um die Objekt-Orientierung.
Die fortgeschrittenen Programmier-Techniken des letzten Kapitels werden uns jetzt genauso begleiten, wie die Grundlagen der Objekt-orientierten Programmierung.

1.6.1. Destruktoren

Um Instanzen (Objekte) wieder ordnungsgemäß aus einem Programm-Ablauf zu entfernen und den Speicher wieder freizugeben, werden **Destruktor**en benutzt. Sie sorgen bei gezählten Instanzen dann für das Runterzählen, damit die Anzahl der wirklich benutzten Instanzen (Objekte) auch noch real ist.

```
finalize()
```

Definition(en): Destruktor
Ein Destruktor ist eine spezielle Methode, die vor dem Löschen / Entfernen einer Instanz (eines Klassen-Objektes) aufgerufen wird.

Beim Verlassen von Blöcken mit neu definierten Objekten oder beim Beenden des Programms werden alle Objekte automatisch entfernt. Das macht die parameterlose finalize()-Methode. Sie taucht in vielen Quellcodes gar nicht auf, was im Endeffekt bedeutet, das ein Destruktor einer übergeordneten Klasse genutzt wird. Vielleicht ist es sogar die finalize()-Methode der Object-Klasse.
Die Speicher-Bereinigung von nicht mehr gebrauchten Objekten und Daten-Resten nennt sich Garbage Collection. Die finalize()-Methode wird, kurz bevor der "Garbage Collector" aktiv wird, aufgerufen.
Der Garbage Collector ist ein Programm im JAVA-System, dass alle Speicher-Bereiche, auf die keine Referenz / kein Zeiger / Name / Bezeichner mehr hinweist, freigibt. Der Gabage Collector verschiebt Objekte usw. zu bestimmten Zeitpunkten, so dass Lücken im Speicher mit den folgenden Objekten aufgefüllt werden.
Wir können für unsere Instanzen (Objekte) aber auch eigene Destruktoren schreiben. Diese überschreiben dann die geerbte Methode (@Override; → 1.4.2.1. Überschreiben von Methoden (Override)).
Konstruktoren und Destruktoren sind z.B. auch dazu zu verwenden, um einen ev. vorhanden Objekt-Zähler (für diese Klasse) immer aktuell zu halten.

In JAVA reicht oftmals die null-Setzung eines Objektes aus. Die Freigabe des Speichers erfolgt dann irgendwann vollautomatisch durch die Garbage collection.

1.6.2. Finalisierung

mit **final** als Schlüsselwort
geht für Klassen, Attribute und Methoden
dadurch wird die Veränderlichkeit eingeschränkt

finale Attribute können nicht mehr verändert werden, sie werden praktisch zu Konstanten
bei der Anlage muss sofort ein Wert zugewiesen werden (sonst wäre der Wert null)
für Konstanten ist eine Schreibung in Großbuchstaben üblich. Da man nun nicht mehr mit Großbuchstaben für die CamelCase-Notierung arbeiten kann, werden hier immer Unterstriche zur begrifflichen Strukturierung eingesetzt. Diese Notierung nennt man **Underscore**-Schreibweise.

z.B.:

```
Integer.MAX_ELEMENTE = 118;
Double.MEIN_PI = 3.14;
```

finale Methoden können nicht überschrieben werden
d.h. sie können in Subklassen nicht mehr überschrieben / geändert / angepasst (→ @Override) werden

finale Klassen können nicht erweitert werden
d.h. von einer finalisierten Klasse kann keine Subklasse mehr angelegt werden

1.6.3. Überladen von Methoden (Overload(ing))

Programmierer stehen oft vor dem Problem, dass sie eine tolle Methode entwickelt haben. Diese ist auf die speziellen Nutzungs-Bedingungen im aktuellen Programm zugeschnitten. Irgendwann sind dann mit einmal Erweiterungen gefragt. Statt der Addition von zwei Ganzzahlen sollen es auf einmal drei oder vier sein. Oder die zu bearbeitenden Datentypen sind auf einmal breiter gefächert. Man braucht neben den Ganzzahlen-Methoden auch solche für Gleitkommazahlen.
Natürlich könnte man sich mit zusätzlichen Methoden und Methoden-Namen helfen. Die Übersichtlichkeit nimmt dabei garantiert nicht zu. Die Häufigkeit für Fehleinsetzungen wird sicher steigen.
In Objekt-orientierten Programmiersprachen gibt es zur Lösung dieses Konflikts das Overloading (dt.: Überladen). Beim Überladen bleibt derMethoden-Name gleich zu anderen existierenden und weiterhin gültigen Methoden, es werden aber Parameter-Listen ergänzt, verändert oder verringert.
besonders interessant das Nutzen von mehreren Daten-Typen in der Parameter-Liste

Beispiel:

```
int add(int a, int b) …
int add(int a, int b, int c) …
double add(double a, double b) …
…
```

das Ändern der Parameter-Bezeichner alleine reicht nicht für ein Überladen (für JAVA wäre dann nicht klar, welche der beiden dann existeirenden Methoden die gültige / zu nutzende sein soll)

man spricht vom nicht-validen Überladen

Beispiel: int add(int **a**, int **b**, int **c**) … int add(int **x**, int **y**, int **z**) …

beide Methoden sind nicht zu unterscheiden

→ nicht-valides Überladen → Compiler-Fehler

genau so verhält es sich mit einem ausschließlich geänderten Rückgabe-Datentyp

Beispiel: **int** add(int a, int b, int c) … **double** add(int c, int c, int e) …

charakteristisch für Overloading ist also:

- keine Vererbung
- immer in der gleichen Klasse
- Daten-Typen kann sich änden
- Sichtbarkeit kann verändert werden

Aufgaben:

1.

2. Welche der nachfolgenden Methoden wären als Überladung zur Methode int mult(int g, int h, int i, int j) … geeignet? Begründen Sie jeweils!

a) int multiplikation(int g, int h, int i, int j) …
b) int mult(int a, int b, int c, int d) …
c) int mult(int g, int h, int i) …
d) int mult(int g, int h, int i, int j) …
e) int mult(int g, int h, int i, int k, int l) …
f) double mult(int g, int h, int i) …
g) int mult(double g, double h, double i) …
h) double mult(double g, int h, int i) …

3.

x.

Definition(en): Methoden-Deklaration
Eine Methoden-Deklaration ist die Vordefinition / Bekanntgabe des Methoden-Kopfes.
Methoden-Deklaration = Methoden-Rumpf **Rückgabewert + Methodenname + Übergabeparameter**

Definition(en): Methoden-Signatur
Unter der Methoden-Signatur versteht man die Charakteristika einer Methode, wie den Methoden-Namen und die Parameter (Bezeichner + Datentyp).
Methoden-Name + Übergabeparameter

public Typ1 methode(Typ2 Parameter1, Typ2 Parameter2, Typ3 Parameter3) …
und
public Typ4 methode(Typ2 Parameter1, Typ2 Parameter2, Typ3 Parameter3) …

haben verschiedene Deklarationen
aber die gleiche Signatur
public Typ4 **methode(Typ2** Parameter1**, Typ2** Parameter2**, Typ3** Parameter3**)** …

die folgende Methode hätte sowohl eine andere Deklaration als auch eine andere Signatur:

public **Typ1** methode(**Typ2** Parameter1, **Typ2** Parameter2, **Typ3** Parameter3) …
und
public **Typ4** methode(**Typ2** Parameter1, **Typ3** Parameter3**, Typ2** Parameter2) …

weil die Reihenfolge der Parameter (also dessen Namen) und die Reihenfolge der Parametertypen verändert wurde
die unterschiedlichen Rückgabe-Typen wirken zusätzlich

Aufgaben:

1.

2. Geben Sie die Methoden mit gleichen Signaturen an!

a) private int mache(int a, int b) …
b) private int mache(int a, double b) …
c) private double berechne(int a, int b) …
d) private int berechne(int a, int b) …
e) protected int mache(int c, int d) …
f) private double mache(int a, int b) …
g) public int mache(int a, int b, int c) …
h) private int berechne(String a, int b) …
i) public double mache(double a, double b, double c) …
j) private double mache(double x, double y, double z) …

3.

1.6.4. Abstrakte Klassen

aus diesen Klassen lassen sich keine Objekte (Instanzen) ableiten, aber Unterklassen

abstract Sichtbarkeit class *KlassenBezeichner* {
}

Sichtbarkeit abstract class *KlassenBezeichner* {
}

Deshalb müssen abstrakte Klassen also unbedingt vor der Instanzierung "gesubklasst" werden. D.h. also es muss zuerst eine Sub-Klasse angelegt werden, und aus dieser heraus können dann Instanzen erstellt werden.

Eine Subklasse kann auch wieder abstrakt geführt werden, dann benötigt eben von dieser eine weitere Subklasse (also eine Subsubklasse der ersten abstrakten Klasse), um Objekte instanzieren zu können

in abstrakten Klassen können abstrakte Methoden definiert werden
es gilt auch, dass abstrakte Methoden nur in abstrakten Klassen definiert werden können

sie dienen praktisch als Platzhalter / Namens-Definierer für weitere konkrete Methoden in den Subklassen
diese Methoden haben keinen Methoden-Rumpf und enden sofort hinter der Parameter-Liste mit einem Semikolon

Sichtbarkeit abstract *MethodenName*(ParameterListe);

bzw.

abstract Sichtbarkeit *MethodenName*(ParameterListe);

sie können Attribute, Methoden und Konstruktoren enthalten

abstrakte Klassen dienen der Vorbeugung von Code-Duplizierungen

Beispiel:

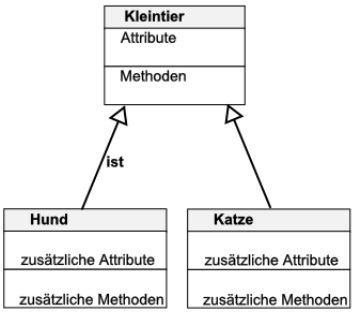
```
…
public abstract class Tier {
    protected String wissName;
    public abstract String getObergruppe();

    public String getWissName() {
        return wissName;
    }
}

…
```

```
…
public class Kaninchen extens Tier {

    //Konstruktor
    public Kaninchen(String wissName) {
        this.wissName = wissName;
    }

    //Interface-geforderte Methode
    @Override
    public String getObergruppe() {
         return "Säugetier";
    }
…
}

…
```

1.6.5. Polymorphie

Vielgestaltigkeit

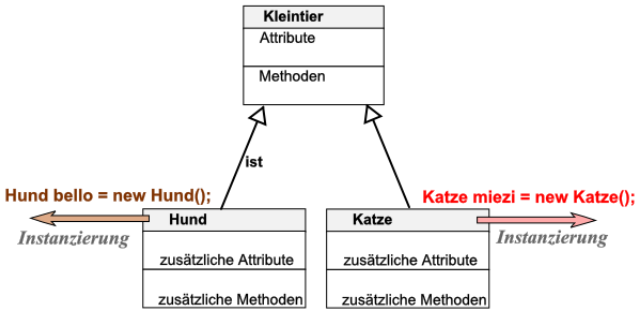
Erstellen von Instanzen einer Klasse durch Benutzen der Unter-Klasse, womit ja automatisch die Ober-Klasse mit einbezogen ist.
Klassen-Hierrarchie zwischen Kleintieren, Hunden und Katzen



z.B. Instanzierung:

```
Hund bello = new Hund();
Katze miezi = new Katze();
```

Instanzen haben keine direkte Beziehung zueinander

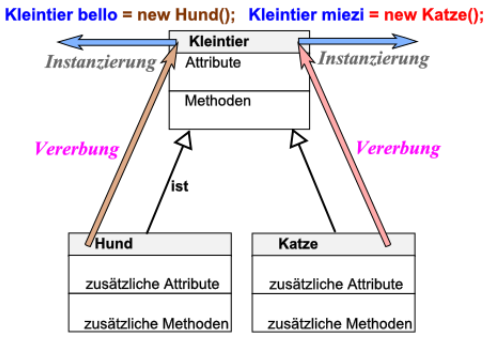


sagen wir jetzt:

```
Kleintier bello = new Hund();
Kleintier miezi = new Katze();
```

Instanzen haben gemeinsame – übergeordnete – Eigenschaften

Dadurch kann z.B. ein Array Kleintier-Zoo erstellt werden, in dem verschiedene Kleintiere (bei uns derzeit Hunde oder Katzen) Platz finden.



Alle Hunde, Katzen und Kaninchen sind (gleich-klassig) Kleintiere, die in Kollektionen (→ 1.5.3.2. Kollektionen) zusammengefasst werden können. Die einzelnen Kleintiere sind aber auch mit den speziellen Attributen der Unterklasse (z.B. Hund) ausgestattet, so dass neben dem gemeinsamen Zugriff auch Klassen-spezifische Zugriffe möglich sind.
Eine Erweiterung des Kleintier-Zoos durch Kaninchen ist ohne weiteres möglich. Dazu mus nur die Unterklasse definiert / integriert werden. Alle Operationen auf die Kollektion Kleintier-Zoo funktionieren ohne weitere Änderungen am Quell-Text.
Bei überschriebenen Methoden werden in der Praxis die Sub-Klassen-Methoden ausgeführt, da sie ja die zugehörige Methode des Spezial-Objektes sind. Gibt es keine überschiebene Methode in der Unterklasse, dann gilt automatische die übergreifende Methode der Oberklasse auch in der Unterklasse (Vererbung).

Mit Polymorphie ist die gleichartige Benennung von Methoden gemeint, in verschiedenen Klassen sachlich zusammengehören (und deshalb gleich heißen), aber mit abweichenden Funktionen ausgestattet sind.
die Kleintiere haben eine Methode bewegen(), die in den Unterklassen speziell implementiert wird

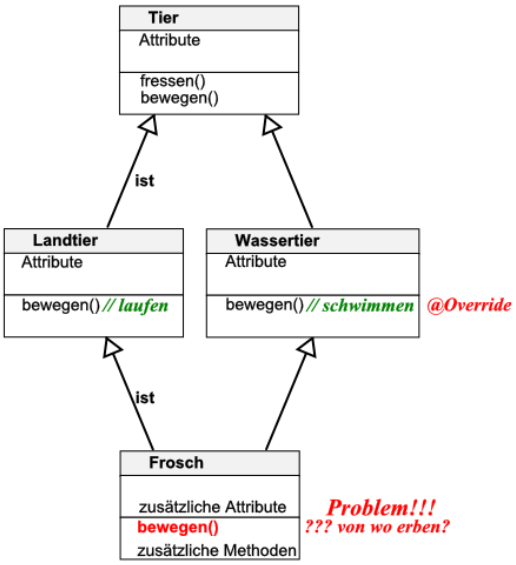
für die Katzen brauchen wir eine schleichende Bewegung, wenn bewegen() aufgerufen wir die Hunde sollen beim gleichen Aufruf aber schnüffeln

Soll der Konstruktor der Oberklasse aufgerufen werden, dann geht dass mit der **super()**-Methode. Sie "weiss" den Eltern-Klassennamen und benutzt diesen entsprechend.
Die super()-Methode muss die erste Anweisung innerhalb eines (lokalen) Konstruktors sein. Ein Aufruf in anderen Methoden usw. ist nicht zuläsig.

1.6.6. Mehrfach-Vererbung (Interface's, Schnittstellen)

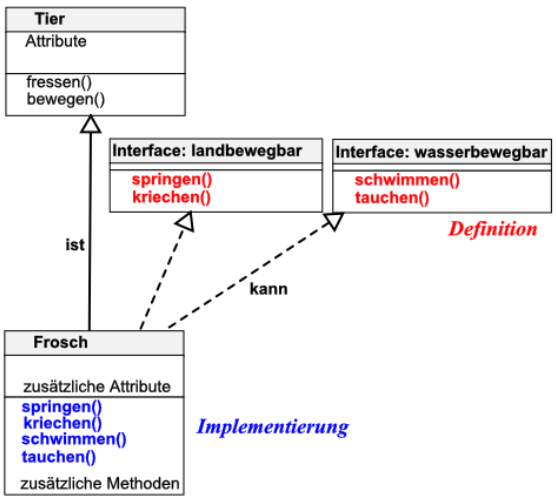
Problem bei gewollter Vererbung von mehreren (hierrarchisch gleichrangigen) Oberklassen. Haben z.B. beide die gleiche Methode implementiert, dann ist nicht klar, welche dieser beiden von der Subklasse angesprochen wird.

Deshalb ist in JAVA eine Mehrfach-Vererbung nicht zulässig. Die gleichnamigen Methoden müssen in einem gemeinsamem Interface deklariert werden und die Umsetzung in die Subklassen ausgelagert werden.



Schnittstellen (Interfaces)

Schnittstellen dienen der Definition von einheitlichen Begriffen (hier: Methoden), die dann in den Detail-Klassen (Subklassen) inhaltlich umgesetzt werden. Interfaces's kann man sich als Kontaktstellen / Vorschriften-Liste / Standard-Festlegungen vorstellen. Sie sind für die untergeordneten Klassen bindend und diese müssen sie umsetzen. Man kann Interface's somit auch als bindenen Vertrag / (gesetzliche) Richtlinie / Rahmen-Vorschrift für die Entwickler der Sub-Klassen verstehen.



Im Zweifelsfall kann man zuerst natürlich leere Methoden implementieren, damit man nicht beim Testen der ersten Methode schon vom Compiler angemeckert wird.
Die Bezeichner von Interface's werden wie Klassennamen / Objekttypen mit einem Großbuchstaben beginnend geschrieben. Es ist üblich den Namen mit **–able** (für engl. Fähigkeits-Namen) oder **–bar** (für dt. Fähigkeits-Namen) enden zu lassen.

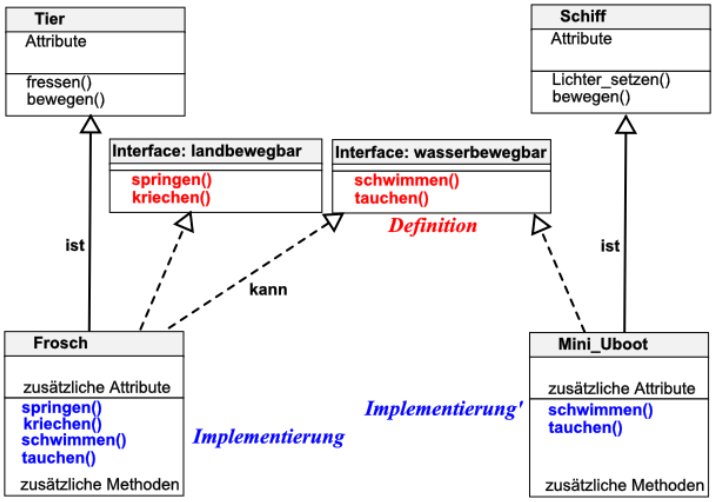
Geeignete Namen könnten also Fly**able**, Print**able**, Comput**able**, … sein. Für den eher seltenen Fall der deutschen Benennung: Flieg**bar**, Druck**bar**, Berechen**bar**, ….

class *KlassenBezeichner* implements *InterfaceBezeichner* { … }

Für / in mehrere(n) (weitere) Klassen könnte das dann so aussehen:

class *KlassenBezeichner2* implements *InterfaceBezeichner* { … }
---
class *KlassenBezeichner3* implements *InterfaceBezeichner* { … }

In einem Interface sind die Methoden immer "public abstract" und Attribute immer "public static final". Da dass quasi Natur-gegeben ist, kann man auf das Mitschreiben dieser Modifikatoren in den Interface's verzichten. Sie werden vom Compiler automatisch einbezogen.



Man könnte nun in einem Interface eine Liste von Programm-internen Konstanten definieren. Diese würden dann im gesamten Programm gelten und unveränderlich sein.

```
…
public interface Konstanten {
    public static final int KONSTANTE1 = 123;
    …
}
…
```

möglich, aber nicht JAVA-Programmierer-Like

Diese Art der Konstanten-Definition gilt als schlechter Programmier-Stil (Anti-Pattern) und wird durch das folgende Konstrukt besser realisiert:

```
public final class Konstanten {
    private Konstanten() {
    public static final int KONSTANTE1 = 123;
    …
    }
}
```

guter Programmier-Stil

Der Zugriff auf die Konstante erfolgt dann mit:

Konstanten.*KONSTANTE1*

Die Notierung der Konstanten-Namen erfolgt aber immer in Großbuchstaben und bei Bedarf mit absetzenden Unterstrichen (Underscore -Schreibweise)

Mit einem gemeinsamen Interface können mehrere Klassen später auch in Kollektionen verwaltet werden. Die Kollektion muss dann eben auf den Objekttyp des Interface angewendet werden.

Kollektion<InterfaceObjekttyp> *Bezeichner* = new Kollektion<>();

Jede Klasse kann später beliebig viele Interface's implementieren. Dabei muss dann jede der (- im Interface -) deklarierten Methoden in den nicht-abstarkten Klassen implementiert / mit Leben ausgefüllt werden. Dadurch fordert man vom Programmierer eben genau die "notwendigen" (allgemein erwarteten) Methoden auch für seine Klasse zu programmieren und damit anderen bereitzustellen.

Anwendung bei Konstanten-Listen

Aus der JAVA-API
Iterable (Interface zur Iterierbarkeit von Datentypen)
Comparable (Interface zur Vergleichbarkeit von Objekten)

Abstrakte Klassen können auch Interface's implementieren. Die Realisierung wird dann an die erste nicht-abstrakte Unter-Klasse weitergegeben. Diese muss die Methode dann enthalten.

Jede Klasse kann immer nur ein der Superklassen erweitern (ansonsten würde es sich ja um das verbotene Mehrfach-Vererben handeln). Aber jede Klasse kann auch zusätzlich noch Interface's integrieren.

class KlassenBezeichner extends *SubklassenBezeichner* implements *InterfaceBezeichner1, InterfaceBezeichner2, …* { … }

Ein Interface's kann auch ein anderes erweitern.

ab JAVA V.8 ist eine default-Definition einer Methode möglich
diese kann **und sollte** dann in den Subklassen präzisiert (Überschrieben) werden.

Beispiel
alt:

```
…
public interface Bewegbar {
   public String getBewegbar();
…

…
```

neu nun:

```
…
public interface Bewegbar {
    public default String getBewegbar() {
       return "unklar";
    }

…
```

weiterhin sind jetzt auch statische Methoden in Interface's realisierbar

Verwendung für Konstanten-Sammlungen, Markierungen und Container,

zusammenfasend gilt:

- ein Interface ist eine gedachte abstrakte Klasse mit gemeinsamen / allgemeinen Fähigkeiten / Funktionen (Methoden) von Objekten
- Attribute eines Interface müssen immer public, final und initierbar sein!
- Methoden sind abstract und public (niemals aber final und / oder static)
- Interface's enthalten niemals Konstruktoren!

1.7. Planen und Entwickeln größerer Programme

1.7.0. Allgemeines

Einfache Programme. wie wir sie bisher besprochen haben, schreibt man einfach so auf. Aber man stößt dabei schnell an Grenzen. Die verwendeten Klassen werden zueinander unübersichtlich. Was macht welche Methode in welcher Klasse? Wie genau war die Definition in der einen Klasse – und wie war die in der anderen Klasse? In JAVA wird die Sache schon dadurch unübersichtlich, dass alle Klassen in eigenen Dateien verwaltet werden. Da ist ein ständiges Umschalten zwischen den Dateien / Klassen nicht zu vermeiden.

Früher wurden Programme nach der **code&fix-Methode** erstellt. Sie bestand aus den Schritten:

1. Schreiben des Programms
2. Testung und Fehler-Bereinigung

Ein solches Vorgehen ist für kleine Programme auch kein Problem. Bei größeren Programmen kann aber die Fehler-Bereinigung in einem untergeordneten Programmteil fatale Folgen / Veränderungen für das Gesamt-Programm bedeuten. Die notwendigen Tests werden immer aufwändiger und man müsste praktisch nach jeder elementaren Code-Änderung wieder völlig von vorne testen. Aufgrund genau solcher Probleme laufen viele große Software-Projekte so schön gegen den Baum. Weitere Probleme sind mangelnde Umsetzungs-Disziplin; Kommunikation und undurchsichtige Trick-Programmierung.
Die ersten Anwender waren noch richtige Planer und Vordenker. Ihre Programme mussten möglichst gleich perfekt sein. Das lag daran, dann man Rechen-Zeiten zugeteilt bekam. In Zeiten der ersten teuren Großrechenr ging das gar nicht anders. Die Programme wurden auf Ellen-langen Papier-Rollen als Programm-Ablauf-Pläne entwickelt und in Trocken-Übungen getestet. Dann ging man zu einem Programmierer, der den Großrechner und die zugehörige – irre komplizierte – Programmiersprache verstand, und ließ ihn den Programm-Ablauf-Plan umsetzen. Wenn das Programm dann nicht lief, war das dem Programmierer egal. Er bekam sein Geld und der Anwender musste die Fehler analysieren und wieder neue Rechenzeit beantragen.
Mit dem Aufkommen der z.T. viel Leistungs-fähigeren PC's änderte sich die Computerwelt völlig. Jeder konnte nun soviel Rechenzeit benutzen, wie sein Gerät oder der Tag hergab.
Einfache Programmiersprachen – wie BASIC – ermöglichten es praktisch jedem, halbwegs logisch denkenden Anwender, ein funktionierendes Programm zu schreiben.
Fehlerhafte Programme – nun auch von weniger qualifizierten Programmierern geschrieben – wurden einfach neu übersetzt und neu ausprobiert. I.A. schrieb ein Programmierer sein eigenes, ganz spezielles Programm. Selten wurden Teams gebildet oder die Aufgaben / Probleme auf einzelne Programmierer verteilt. Der Solo-Programmierer wusste über sein Programm und seine Teile sehr gut bescheid und konnte recht effektiv ändern, Fehler finden oder neuen Code ergänzen.
Programmierer haben sich lange keinen Kopf um die Planung ihrer Programme gemacht.
Mitlerweile sind die Programme immer größer geworden, die Team's nehmen Größen von mehreren hundert Personen an und bei dringenden Projekten wird rund um die Uhr und rund um die Welt programmiert. Was der eine Programmierer in Europa bis Feierabend nicht erledigt hat, macht ein anderer in Amerika weiter und übergibt sein halbfertiges Quellcode-Stück an den Programmierer in Indien. Einen Tag später kommt ein vielleicht völlig überarbeitetes Programm beim europäischen Programmierer auf den Bildschirm.
Ein weiteres großes Problem war die mangelnde Kommunikation zwischen den nerdigen Programmierern und den anspruchsvollen und oft informatisch sparsam gebildeten Anwendern. Die Programme funktionierten zwar, machten aber nicht das, was und wie sich der Anwender / Auftraggeber es sich vorgestellt hatte. Von mangelnder Bedienbarkeit durch die

Sekretärin, die sich um informatische Konzepte und Modelle nun überhaupt keine Gedanken machen will, mal ganz zu schweigen.
Spätestens jetzt müssen neue Arbeits-Organisationen greifen, sonst entsteht ein völliges Code-Chaos.
Vor das eigentliche Programmieren nach der code&fix-Methode muss unbedingt eine Definitions- oder Planungs-Phase eingeschoben werden. Das Gesamt-Konzept eines Programms muss im Vorfeld besprochen und geplant werden. Aufgaben werden verteilt und die verschiedenen Teile praktisch parallel entwickelt. Jeder Programmierer verlässt sich darauf, dass der / die Anderen die Absprachen einhalten.
Heute sind zwei Planungs-Methoden weit verbreitet. Das ist zum Einen das **Struktogramm (NASSI-SHNEIDERMAN-Diagramm**; Abk.: **NSD)**, dass einen / den Algorithmus gut abbildet. Und zum Anderen sind das die **UML-Diagramm**e, die quasi die großen und kleineren Zusammenhänge eines Projektes (Klassen-Strukturen und einiges mehr) verdeutlicht.
Programmier-Anfänger werfen immer gerne ein, dass bei den einfachen Aufgaben, die im Rahmen des Erlernen einer Programmiersprache, erstellt werden, diese Techniken nur aufhalten. Das stimmt auch erst einmal. Was man aber bedenken muss, ist der Übergang zum Groß-Projekt. Wenn ich im Vorfeld die Verfahren nicht an einfachen Beispielen / Übungen erlernt habe, dann wird die Umsetzung im Großen kaum gelingen. Bevor man quadratische Gleichungen usw. lösen kann, muss man auch erst mal die Grundrechenarten beherrschen. Oder hätten Sie sich zugetraut schon in der ersten Klasse eine binomische Formel zu berechnen?
Gerade deshalb bestehen die meisten Kursleiter auf die "lästigen" Techniken. Das ist keine Schikane, sondern bittere Notwendigkeit. Man sollte es als grundlegende Fähigkeit sehen. Kreatives Schreiben eines Romans oder eines Gedichtes im Abitur geht auch nicht ohne das gelernte Schreiben von Worten in der 1. Klasse.

1.7.1. Struktogramme

Das Erstellen von Struktogrammen kann man zu Anfang auch erst nach der Code-Erstellung vornehmen. Das übt auch und man entwickelt ein Gefühl für die Strukturen / Blocks.
Heute gibt es Software, die bei der Erstellung der Struktogramm hilft, so dass das Blöcke-Zeichnen weitaus unkomplizierter abläuft, als auf einem Stück Papier.

Eine ausführliche Darstellung zu den Struktogrammen befindet sich im Skript 🕮 **Sprachen und Automaten**.

Hier schauen wir uns nur die Umsetzung der Elemente in Quellcode-Schnipsel an.

Eingabe(-Block)

Struktogramm-Symbol

Quellcode-Entspechung (JAVA)

```
…
import java.util.Scanner;
…
Scanner scan = new Scanner(System.in);
System.out.print("Text eingeben: ");
String eingabe = scan.nextLine();
scan.close;
…
```

```
…
import java.io.*;
…
BufferedReader puffer = new BufferedReader(new ↵
InputStreamReader(System.in));
System.out.print("Wert eingeben: ");
String eing = puffer.readLine();
…
```

über IO-Klasse des Java-Editors

```
…
int eing1;
double eing2;
String eing3;
eing1 = InOut.readInt("Ganzzahl eingeben: ");
eing2 = InOut.readDouble("KommaZahl eing,: ");
eing3 = InOut.readString("Text eingeben: ");
…
```

Anweisung(s-Block)

Struktogramm-Symbol

Verarbeitung(sschritt)

Quellcode-Entspechung (JAVA)

```
…
int x = 4;    //Definition und Initialisierung
…
x = 23;       //Veränderung / Zuweisung
…
```

```
…
int a;        //Definition
…
a = 3 * x;    //Initialisierung und Zuweisung
…
```

```
…
a = funktion(x);    //Methoden-Aufruf / Zuweisung
…
```

Ausgabe(-Block)

Struktogramm-Symbol

Ausgabe:

Quellcode-Entspechung (JAVA)

```
…
System.out.println( … );
…
```

zusammengesetzte Ausgabe

```
…
System.out.print( … );       //ohne Zeilenumbruch
System.out.println( … );     // mit Zeilenumbruch
…
```

formatierte Ausgabe

```
…
System.out.printf(" %? ", ?-Wert);
System.out.printf(" … /n", … ); // mit Z.umbruch
…
```

Zusammenfassung(s-Block) / Block

Struktogramm-Symbol

Verarbeitung(sschritt)

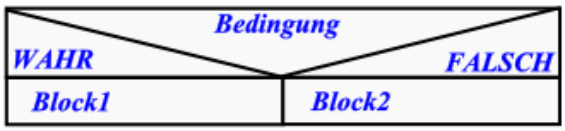
Verarbeitung(sschritt)

Quellcode-Entspechung (JAVA)

```
{     //(Gross-)Block-Anfang
…
?;    //Verarbeitungsschritt
?;    //Verarbeitungsschritt
…
}     //(Gross-)Block-Ende
```

Verzweigung(s-Block)

Struktogramm-Symbol



Quellcode-Entspechung (JAVA)

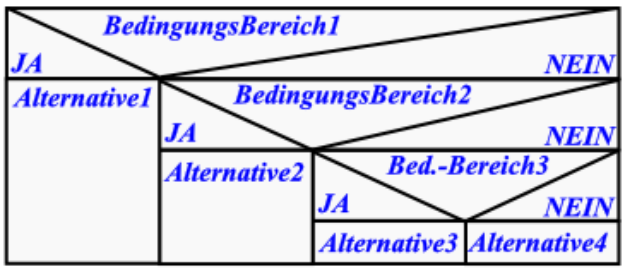
```
…
if ( ? ) {      //zu prüfende Bedingung
…               //Wahr-Block
} else {
…               //Falsch-Block
}
…
```

Struktogramm-Symbol

Quellcode-Entspechung (JAVA)

```
…
if ( ? ) {      //zu prüfende Bedingung
…               //Wahr-Block
}
…
```

Struktogramm-Symbol



Quellcode-Entspechung (JAVA)

```
…
if ( ? ) {          //zu prüfende Bedingung
…                   //Wahr-Block
} elseif ( ? ) {    //zweitrangige Bedingung
…                   //Wahr-Bl. für 2.-rang. Bed.
}
…
```

Mehrfach-Verzeigung

```
…
if ( ? ) {          //zu prüfende Bedingung
…                   //Wahr-Block
} … {
} elseif ( ? ) {    //n-rangige Bedingung
…                   //Wahr-Bl. für n.-rang. Bed.
} else {            //(Bedingungs-)Rest
…
}
…
```

Mehrfach-Auswahl(-Block)

Struktogramm-Symbol

Quellcode-Entspechung (JAVA)

```
…
switch( … ) {       //AuswahlVariable
  case …:           //Variante 1
    …
    break;
  case …:           //Variante 2
    …
    break;
…
  case …:           //Variante n
    …
    break;
  default:
    …
}
…
```

Iteration(s-Block) – Kopf-gesteuerte Schleife

abweisende Schleife

Struktogramm-Symbol

Quellcode-Entspechung (JAVA)

```
…
while ( … ) {        //Schleifenkopf / Bedingung
…                    //Schleifenkörper
}
…
```

Iteration(s-Block) – Fuß-gesteuerte Schleife

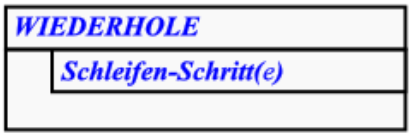
Schleife

Struktogramm-Symbol

Quellcode-Entspechung (JAVA)

```
…
do {
…                   //Schleifenkörper
} while ( … );      //Schleifenfuß / Bedingung
…
```

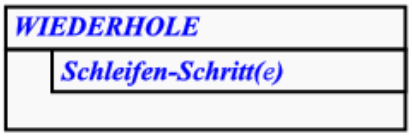
Iteration(s-Block) - Zähl-Schleife

Struktogramm-Symbol	

Quellcode-Entspechung (JAVA)	…
	`for (int i = 0; i < ?; i++) {`
	… `//Schleifenkörper`
	`}`
	…

runterzählende Schleife	…
	`for (int i = ?; i > 0; i--) {`
	… `//Schleifenkörper`
	`}`
	…

Iteration(s-Block) – erweiterte Zähl-Schleife

Struktogramm-Symbol	

Quellcode-Entspechung (JAVA)	…
	`for (TYP elem : Liste) {`
	… `//Schleifenkörper`
	`}`
	…

Abbruch

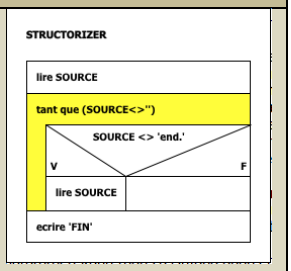
Struktogramm-Symbol	
Quellcode-Entspechung (JAVA)	… `break;` …

Es ist schon echt überwältigend, wie klar und einfach doch die Übertragung von Blöcken in einen JAVA-Quellcode fällt.

Kurzbedienungs-Anleitung: Structorizer

Zweck:

- Erstellen von Struktogrammen / Diagrammen nach NASSI-SHNEIDERMAN
- Export von Quell-Texten (in mehreren Programmier-Sprachen! möglich)



Lizenzen (Autor(en); Hersteller; Vertreiber):
GNU GPL 3; Robert (Bob) Fisch

Installation:
Download von http://structorizer.fisch.lu/ (Installer oder portable Version (Download als ZIP)
Es ist auch ein Entpacken oder Hineinkopieren des entpackten Ordners in einen "portableApps"-Ordner möglich. (ZIP-Datei enthält schon einen Ordner Structorizer → also direkt entpacken!)
empfohlen: Fertig installiert und eingerichtet auf dem Io-Stick (Abitur-Version; Download: → https://tinohempel.de/info/info/IoStick/index.html).

Starten:

- direkt die **Structorizer.EXE** im Entpackungs-Ordner aufrufen
- alternativ über das Menü-System des Io-Stick's
- … (bei "portableApps" findet man das Programm zuerst im Bereich "Sonstiges"; in den neueren Versionen lässt es sich in anderes Verzeichnis verschieben)

neues Struktogramm:
Programm startet mit leerem Block
alternativ:
"Datei" "Neu"
DANN
unbedingt "Datei" "Speichern unter …"

Algorithmus benennen:
auf den aktuellen Namen (z.B.: "") klicken und Dialog ausfüllen
Achtung!: Der Name wird in Großbuchstaben erwartet.

Block nach dem aktuellen / markierten einfügen:
ev. den (vorlaufenden) Block anklicken
passenden Block aus dem rechten Teil der Block-Symbolleiste (mit blauem Pfeil nach unten) auswählen und Dialog ausfüllen
Hinweise!: Wert-Zuweisungen mit "<-" statt einem Gleichheitzeichen

Block vor dem aktuellen / markierten einfügen:
ev. den (nachfolgenden) Block anklicken
passenden Block aus dem linken Teil der Block-Symbolleiste (mit blauem Pfeil nach oben) auswählen und Dialog ausfüllen
Hinweise!: Wert-Zuweisungen mit "<-" statt einem Gleichheitzeichen

Block löschen:
Block markieren / auswählen
aus oberster Symbolleiste die Schaltfläche mit dem roten Kreuz wählen

Struktogramm in Quell-Text umwandeln:

Block-Position ändern (weiter nach vorne / hinten):
zu verschiebenen Block auswählen / markieren
in "Diagramme" mittels "Nach oben" bzw. "Nach unten" Block um eine Position verschieben
Hinweis!: Verschiebe-Symbole auch in oberster Symbolleiste

Struktogramm drucken:

Online-Dokumentationen:
HTML (eine Seite, engl.) → http://structorizer.fisch.lu/Export/
HTML (Hypertext, engl.) → http://help.structorizer.fisch.lu/
PDF, engl. → http://www.fisch.lu/Php/download.php?file=structorizer_user_guide.pdf

externe deutsche Kurzanleitung: → http://www.learn2prog.de/kurzanl.html

Sprache einstellen:
"Einstellungen" Sprache" → …

Text-Voreinstellungen:
"Einstellungen" Sprache" → …

Tastatur-Kürzel / Tips:

Wer das Programm öfter nutzen will / muss, der kann mit Tastatur-Befehlen bestimmte Funktionen schneller aufrufen:

[Alt] [↓] … "Datei" "Neu" ……	**[Strg] [N]**
[Alt] [↓] [↓] [↓] … "Datei" "Speichern unter …"	**[Strg] [⇧] [S]**
[Alt] [↓] [↓] … "Datei" "Speichern" ……	**[Strg] [S]**
[Alt] 8x [↓] … "Datei" "Drucken" ……	**[Strg] [P]**
[Alt] [ ] [ ] …	
[Alt] [ ] …	

Der Structorizer ist ein Spezial-Programm zum Erstellen von Struktogrammen. Die erzeugbaren / exportierbaren Quell-Texte sind als Grundlagen für ein weiteres Editieren in einem externen Programm geeignet (z.B. Eclipse, …).
Der Java-Editor bietet einen guten Struktogramm-Editor innerhalb seiner Oberfläche. Ein Hin- und Her-Schalten zwischen Struktogrammen und JAVA-Quellcode ist in Grenzen machbar. Wer also mit dem Java-Editor sowieso seine Quell-Texte bearbeitet, kann auch gleich den internen Struktogramm-Editor einsetzen.

Kurzbedienungs-Anleitung: Java-Editor (Struktogramme)

Zweck:

- Erstellen von JAVA-Programmen
- Modellierung mittels Struktogrammen und UML-Diagrammen
- Erstellen von JAVA-Quell-Texten aus UML-Diagrammen
- Erstellen von JAVA-Quell-Texten aus Struktogrammen
- …

Lizenzen:

Installation:

Download von www.javaeditor.org (Installer oder portable Version (Download als ZIP)

Starten:

- installierte Version kann über das "Start"-Menü oder über die Desktop-Verknüpfung gestartet werden
- bei der portablen Version ist die Start-Datei die "JavaEditor.exe"

neues Struktogramm:

Reiter "Programm" "Neues Struktogramm" und die passende Vorlage wählen
ODER "Datei" "Neu …" "Struktogramm"; es folgt sofort ein "Speichern unter …" Dialog

Algorithmus benennen:

neben "Algorithmus" auf das "leere Zeichen" (ø) klicken und Bezeichnung eingeben

Block hinzufügen:

passenden Block (Schaltflächen links) anklicken und an die passende Stelle ziehen
Text(e) eingeben (Achtung! Normalerweise ohne "Enter")
    mit den Schlüsselwörtern: "Eingabe:" und "Ausgabe:" arbeiten
WENN fertig mit Text-Eingabe 1x neben den Block klicken

Block entfernen:

Achtung!: Es gibt kein "Rückgängig"!
Block oder mit [Strg]-Taste Block-Gruppe auswählen
Mülleimer-Schaltfläche ("Löschen")

Block verschieben:

Block oder mit [Strg]-Taste Block-Gruppe auswählen
mit Maus an neue Position ziehen

Algorithmus ausdrucken:

Algorithmen-Block durch Klicken auf Ecke / Kante des Gesamt-Blockes auswählen
Algorithmen-Block in die linke obere Ecke ziehen
    ev. "Druckereinrichtung" im "Datei"-Menü
mit Maus an neue Position ziehen
"Datei" "Drucken"

Struktogramm (als Bild) speichern / exportieren:

"Datei" "Exportieren" (als PNG-, BMP- od. WMF-Datei)

Java-Quell-Code erstellen / erzeugen:

Klicken auf das J-Symbol in der Werkzeugleiste (links)

Tastatur-Kürzel / Tips:

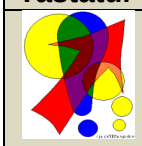
Wer das Programm öfter nutzen will / muss, der kann mit Tastatur-Befehlen bestimmte Funktionen schneller aufrufen:

[Alt] [D] [N] … "Datei" "Neu"	**[Strg] [N]**
[Alt] [D] [U] … "Datei" "Speichern unter …"	
[Alt] [D] [S] … "Datei" "Speichern"	**[Strg] [S]**
[Alt] [D] [F] … "Datei" "Öffnen …"	
[Alt] [F] [P] … "Datei" "Drucken"	**[Strg] [P]**
[Alt] [ ] [ ] … "Einfügen" "Freitext" (nicht zugeordneter Knoten) ….	
[Alt] [ ] … Einfügen einer Anmerkung	

1.7.2. UML-Diagramme

UML-Diagramme werden heute ebenfalls mit hochspezialisierten Programmen erstellt. Viele dieser Produkte können dann automatischen schon halbwegs funktionierende Programm-Gerüste automatisch in der gewünschten / unterstützten Programmiersprache erstellen. Gerade JAVA wird hier sehr oft unterstützt.
Was bleibt, ist dann "nur noch" die Detail-Arbeit. Die Abnahme der lästigen Routine-Schreibarbeit durch die UML-Diagramme ist aber schon eine Errungenschaft.
An dem Wechselspiel zwischen UML-Fenster und Editor kann man auch sehr schön sehen wie logisch und abstrakt diese beiden Elemente zusammenhängen. Da kommt man schon mal auf die Frage, wielange werden noch echte Programmierer gebraucht? Wenn man dann aber sieht, wieviel Detail-Arbeit noch bleibt und welche Probleme dort auf uns warten, dann relativiert sich die Frage.

Kurzbedienungs-Anleitung: UMLed
Zweck: **Lizenzen:**
Installation: Download von (Installer oder portable Version (Download als ZIP)
Starten:
neue MindMap: "Datei" "Neu …" und die passende Vorlage wählen ODER "Datei" "neue leere Map" DANN unbedingt "Datei" "Speichern unter …"
:
:
Tastatur-Kürzel / Tips:
Wer das Programm öfter nutzen will / muss, der kann mit Tastatur-Befehlen bestimmte Funktionen schneller aufrufen: **[Alt] [F] [N]** … "Datei" "Neu" …….. **[Strg] [N]** **[Alt] [F] [A]** … "Datei" "Speichern unter …" **[Alt] [F] [S]** … "Datei" "Speichern" …… **[Strg] [S]** **[Alt] [F] [P]** … "Datei" "Drucken" ……..…… **[Strg] [P]** **[Alt] [I] [F]** … "Einfügen" "Freitext" (nicht zugeordneter Knoten) …. **[Alt] [Enter]** … Einfügen einer Anmerkung

Kurzbedienungs-Anleitung: Java-Editor (UML-Fenster und Klassen-Modellierer)
Zweck: **Lizenzen:**
Installation: Download von (Installer oder portable Version (Download als ZIP)
Starten:
neue MindMap: "Datei" "Neu …" und die passende Vorlage wählen ODER "Datei" "neue leere Map" DANN unbedingt "Datei" "Speichern unter …"
:
:
Tastatur-Kürzel / Tips:
 Wer das Programm öfter nutzen will / muss, der kann mit Tastatur-Befehlen bestimmte Funktionen schneller aufrufen: **[Alt] [F] [N]** … "Datei" "Neu" …….. **[Strg] [N]** **[Alt] [F] [A]** … "Datei" "Speichern unter …" **[Alt] [F] [S]** … "Datei" "Speichern" …… **[Strg] [S]** **[Alt] [F] [P]** … "Datei" "Drucken" …….…… **[Strg] [P]** **[Alt] [I] [F]** … "Einfügen" "Freitext" (nicht zugeordneter Knoten) …. **[Alt] [Enter]** … Einfügen einer Anmerkung

Aufgaben:

1.

2.

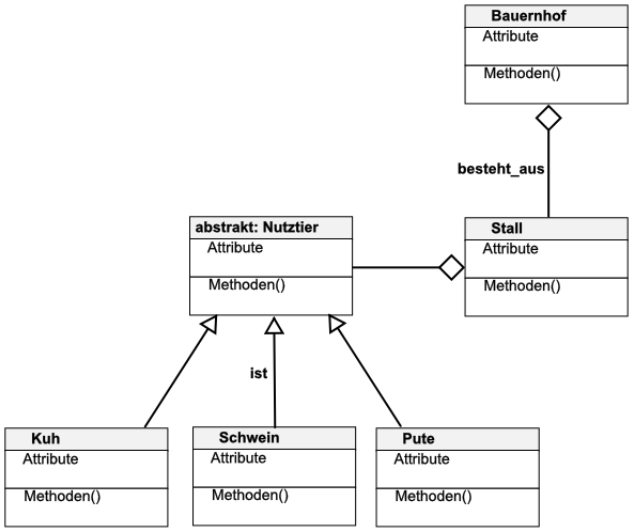
x. Finden Sie weitere Darstellungs-Möglichkeiten für Algorithmen (neben Struktogramm und UML-Diagramm)! Arbeiten Sie Vor- und Nachteile heraus!

x.

komplexe Aufgaben zur Modellierung:

1.

x. Setzen Sie nachfolgendes UML-Diagramm mit jeweils 1 bis 2 Attributen und möglichen Methoden in ein UML-Programm um!

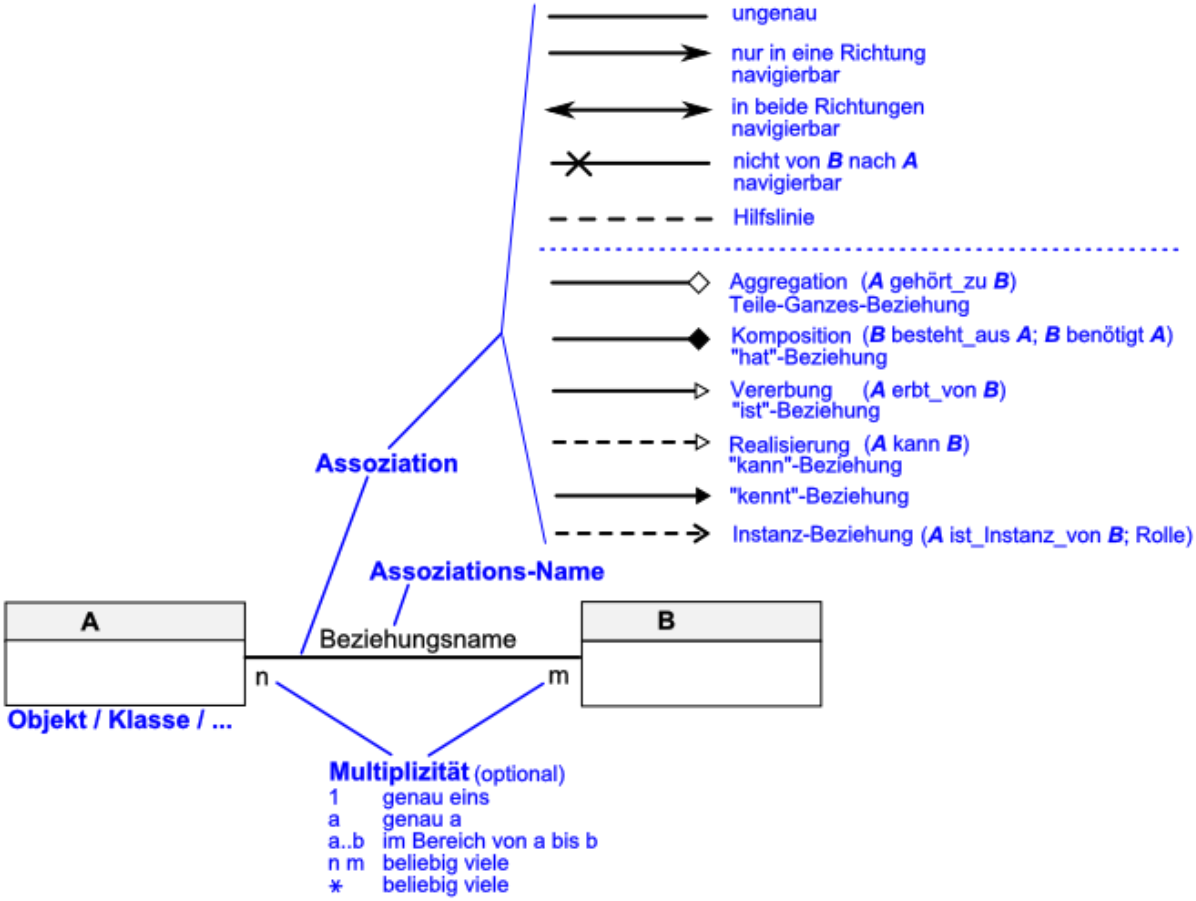


x. Erweitern Sie das UML-Modell um:

a)	Gänse	b)	ein Pferd	c)	Pferdewagen
d)	Personal	e)	Bauer, Bäuerin, Magd	f)	Wachhund, Wohnhaus

x.

Übersicht / Legende zu UML-Diagrammen:



1.8. Nutzung graphischer Oberflächen

1.8.x. Processing Library vom MIT mit der IDE eclipse

Einbinden einer Library
z.B. core.jar aus dem Processing vom MIT

Rechts-Klick auf das eigene Projekt; "Build Path"; "Add External Archives …"
Library auswählen und öffnen
die Library's müssen ev. auch mit ausgerollt werden

```
…
import java.awt.Dimension:[ ]

@SupressWarning("serial")
public class Programm extends PApplet{

    //Setup
    @Override
    public void setup(){

    }

    //Update and draw in Programm
    @Override
    public void draw(){

    }
}
…
```

setup()-Methode wird einmalig beim Programm-Start aufgerufen
praktisch die Initialisierung des Programms

draw()-Methode sorgt für das stnändige Neu-Zeichnen unseres Programmfensters
In der default-Einstellung wird die Methode 60x pro Sekunde aufgerufen, was 60 Frames per second (FPS, Fenster pro Sekunde) ergibt
Bildwiederholung wird aber von der Leistungsfähigkeit begrenzt
der Programmierer muss nur den Inhalt der Anzeige festlegen (in der draw()-Methode)

Die so eingebundene Bibliothek kann dann in weiteren Unter-Klassen unseres Programm's benutzt werden.

```
…
package Paketname;

import java.awt.Dimension;
import java.awt.Point;

import Paketname.basics.Color;
import Paketname.basics.Elliptic;
import Paketname.starter.Programm;
import processing.cor.PApplet;

public class Ball extend Elliptic {

    public Ball(Programm programm, Point position, Dimension dimension,
                Color color){
        super(programm, position, dimension, color);
    }

    @Override
    public void display(){
        programm.ellipseMode(PApplet.CENTER);
        programm.stroleWeight(4);
        programm.fill(getR(), getG(), getB());
        programm.ellipse(getX(), getY(), getWidth(), getHeight());
    }
}
…
```

```
…
package Paketname.starter;

import processing.core.PApplet;

public class Main{
    public static void main(String[ ] args){
        PApplet.main(new    String[    ]{"–present",
"Packetname.starter.Programm"});
    }

}
…
```

Import ??? über den Package Explorer

ev. / möglichst neuen Workspace aufmachen / betreffenden öffnen

Rechts-Klick in Package Explorer und "Import …" aufrufen

Bereich "General" aufklappen und für ein gezipptes Projekt statt "Archives File" wird richtig "Existing Projects into Workspace" auswählen
bei existierendem File-System (also z.B. wenn Projekt schon entzippt wurde) wird "Exist root directory" genutzt sonst eben "Select archive file" für die noch gezippte Datei

ist das Projekt im Project Explorer mit einem rotem Ausrufzeichen versehen, dann muss der "Build Path" über das Kontext-Menü ausgewählt und unter "Configure Build Path" angepasst werden
sollten noch eine alte core.jar in der Konfiguration eingetragen sein, dann muss diese gelöscht ("Remove") werden
alternativ kann über "Edit" die richtige ausgewählt (zugewiesen) werden

solange in einzelnen Dateien / Klassen / … noch Fehler sind, dann erscheint im Project Explorer das Symbolbild mit einem weißem Kreuz auf rotem Grund

Fehler-Kennzeichnung setzt sich in der Datei / Klasse / … fort
es werden aber nur Syntax-Fehler angezeigt
semantische Fehler muss man extra finden, dies ist meist sehr viel aufwändiger, weil hier die echten Denkfehler des / der Programmierer drin stecken

häufig fehlen die Implementierungen von Methoden, die durch Interface's deklariert, die aber noch nicht in den einzelnen Klassen definiert (eingebaut) wurden
eine quasi-default-Implementierung bzw. eine automatische Rumpf-Erstellung funktioniert über einen Rechts-Klick auf den Fehler und den Kontext-Menü-Punkt "Add unimplemented methods"
der Methoden-Rumpf ist nun praktisch da und implementiert, hat aber keine Leistung. Die müssen wir noch reinprogrammieren. Im Körper-Block der Methode steht ein TODO-Hinweis, der uns auch genau darauf hinweist, dass hier noch Arbeit notwendig ist
man kann die Texte hinter dem TODO individuell anpassen, um die passenden Stellen schneller zu finden

im unteren Teil-Fenster "" stehen bei Tasks alle die automatisch erzeugten TODO's drin

Fehler / Probleme werden als kleine farbige Markierungen in einer Skala am rechten Rand des Editor-Fensters angezeigt. Die Skala stellt die gsamte Quellcode-Datei dar. Die Markierungen stellen also die Position innerhalb des Quellcodes dar.
Die Farben bedeuten dabei:

Anzeige-Elemente

Rechtecke

```
rectMode(CORNER)
rect(x,y,breite,hoehe)
```

zeichnet ein Rechteck ausgehend von der Position (x,y) mit breite nach rechts und hoehe nach unten.
der Rechteck-Modus ist also auch die linke, obere Ecke als Bezugs-Punkt festgelegt

Koordinaten-Ursprung ist immer in der linken, oberen Ecke des Fensters bzw. Bildschirms

```
rectMode(CENTER)
rect(x,y,breite,hoehe)
```

das Rechteck wird so gezeichnet, das der "Ausgangs-Punkt" genau in der Mitte (also im Schnittpunkt der Diagonalen) liegt

weitere Rechteck-Modi sind: CORNERS und RADIUS
bei CORNERS müssen die obere linke und die untere rechte Ecke des Rechtecks angegeben werden.
RADIS setzt den Rechteck-Ursprung wie bei CENTER. Die Ausdehnungen des Rechtecks werden durch die Angabe der halben Breite und halben Höhe definiert.

fill(0)
mit nur einem Argument, werden Grauabstufungen definiert (Bereich: 0 … 255)

fill(255,0,0)
ausfüllen mit Farbe nach RGB-System (hier: nur rot)
für jede Farbe Zahlen von 0 bis 255 zugelassen
255,255,255 steht dann für voll weiß

stroke(0)
setzt schwarzen Rand
stroke(0,255,0) setzt Farbe des Rands nach RGB-Angabe (hier: nur grün)

blackground(0)
schwarzer Hintergrund
muss bei jedem Aufruf von draw() erneut angegeben werden! entspricht praktisch dem Lösen der Zeichenfläche

strokeWidth(breite)

colorMode()

noFill()

noStroke

Kreise / Elipsen

```
ellipseMode(CORNER)
```

praktisch wie bei den Rechtecken, nur das hier das umschließende Rechteck um den Kreis / die Ellipse gemeint ist.

Anzeige-Steuerung

frameRate
Abfragen der aktuellen Frame-Rate

frameRate()
setzt die Frame-Rate

noLoop()
Stoppen der Animation

redraw()
bei einer gestoppten Animation Zeichnen eines neuen Bildes

loop()
Starten der Animation

Tastatur und Maus

keyPressed() und **keyReleased()**
Taste gedrückt oder wieder frei gegeben

mouseClicked()
Maus geklickt

mouseMoved()
Maus bewegt

mouseDragged()
mit der Maus etwas angefasst / erfasst (Maus gedrückt gehalten)

Ausgaben

PrintWriter output;

output = createWriter(Dateiname);

output.println(Wert);

output.flush();

output.close();

```
…
BufferedReader reader;
String line;

reader = createReader(Dateiname);

try {
    line = reader.readLine();
} catch (IOException e) {
    e.printStackTrace();
    line= null;
}
if (line==null) { …

…
```

Interface's und abstrakte Klassen

wegen nicht erlaubter Mehrfach-Vererbung in JAVA als Schnittstelle, in denen Methoden vordeklariert (/ namentlich bekannt gemacht) werden, die dann in den benutzenden Klassen mit Leben / Quellcode gefüllt werden müssen
neben Methoden lassen sich auch Konstanten festlegen,
Konstanten-Namen sollten immer in Großbuchstaben und mit absetzenden Unterstrichen (Underscore -Schreibweise) notiert werden
Aufruf der Konstanten dann mit InterfaceName.KontantenName

übergreifende Attribute sind nicht erlaubt

Dokumentation

Eintippen von /** macht automatisch einen Kommentarblock auf, in dem man eine kurze Beschreibung der Funktion einer Methode angibt und die möglichen Rückgabewerte spezifiziert

neue Klassen

möglichst mit Rechts-Klick auf das Package, dann werden schon viele Daten für die Deklaration übernommen werden

1.9. Refactoring

ändern des Codes, um die Verständlichkeit zu verbessern
sollte nur auf funktionieren (und getesteten) Code angewendet werden (sonst erneutes Hineindenken in den umgeschriebenen Code notwendig)

Ziele sind Verbesserung von:
- Verständlichkeit
- Erweiterbarkeit
- Testbarkeit

und
- Behebung von Code Smells

Refactoring Patterns () sind:
Rename (Umbenennen von Attributen, Variablennamen, Methoden, …)

Move
Extract (Superclass, Interface, Method, …)
Pull up
Encapsulate Field (Attribute z.B. kapseln, …)
Use Supertype Where Possible
Inline (mehrere Anweisungen in eine Zeile zusammenziehen; kompaktere Schreibweise)

über Rechts-Klick auf das Quelltext-Objekt und dann aus dem Kontext-Menü bei "Refactoring" die passende Refactoring-Methode auszuwählen

nicht alles, was automatisch geht, muss trotzdem überprüft und ev. noch zusätzlich angepasst werden
zeigt schön, wieviel heute in der Software-Entwicklung schon automatisiert ablaufen kann

Martin FOWLER: Refactoring
Robert C. MARTIN: Clean Code

1.10. Coding Standard's – Konvention für ordentliche Programmierung

grundsätzliche Konventionen sind schon unter → aufgezeigt worden

Variablen-Namen in CamelCase-Notierung

hier einige weitere Informationen, Hinweise und Tools um JAVA im JavaStyle zu programmieren

Metriken (Meßwerte, Code-Eigenschaften, Code-Kennwerte, …)
LoC (Lines of Code), Kommentierungsgrad, Komplexität, Fan In, Fan Out, …

Style
automatische Suche nach "unschönem / schlechtem" Code
Bad Style (fehlende Dokumentation, Missachtung von Namens-Konventionen
Bad Practices (Dead Code, leere Blöcke, unerreichbare Blöcke, unnötig komplizierter Code, unbenutzte Variablen, duplizierter Code, …)

1.10.1 (automatische) Werkzeuge für Coding Standard's

Eclipse lässt sich durch Plugin's ergänzen, die bei der Einhaltung und von Coding Standards helfen
teilweise auch als Standalone-Programme nutzbar
z.B.: PMD, Checkstyle, FindBugs

besonders für Team-Arbeit interessant

Checkstyle
findet fehlende Javadoc zu public-Methoden
Nichteinhaltung der Namens-Konventionen
inkonsistente Formatierungen

PMD
ungenügendes Abfangen von Exception's (leere Catch-Block's)
code-Komplexität
nicht-erreichbarer (toter) Code
Einsatz von Implementierungen statt von Interface's

FindBugs

arbeitet auf dem ByteCode (Code muss also vorher kompiliert werden)
Null-Pointer-Exception's
String-Vergleiche mit == bzw. !=
ArrayIndexOutOfBounds-Exception's (fehlende Index-Kontrolle beim Array-Zugriff)

Installation über Eclipse-Marketplace
z.B. Checkstyle und Findbugs

für Checkstyle gibt es mehrere vorgefertige Konfigurationen (Style – Konventionen)
z.B. google-Version (dicht an den Eclipse-Voreinstellungen)
die auffälligen Stellen sind glblich unterlegt

Aufruf der Plugin's über Rechts-Klick auf das Projekt

1.11. Testen (von Programmen)

Test-Möglichkeiten / -Typen

- **Modul-Test Funktions-Tests** — Tests von einzelnen Funktionalistäten / Methoden / Klassen / …
- **Integrations-Tests** — Tests zum Zusammenspiel / Ineienandergreifen von verschiedenen Programmteilen / Klassen / …
- **System-Tests** — Test des gesamten Programms / Programm-Systems
- **Akzeptanz-Tests** — Test der Software beim Auftrager auf Erfüllung des Pflichten-Heftes, der Handhabbarkeit in der Praxis (Praxis-Tauglichkeit)

optimal wird Test-getrieben programmiert
Tests und Programmierung des eigentlichen Quell-Code parallel (noch besser Test's vor dem eigentlichen Programmieren festlegen / definieren / programmieren)
für Anfänger kaum zu überschauen
deshalb zuerst eher als Neben- oder Nachläufer

Ausprobieren mit verschiedenen Parametern, unterschiedlichen Voreinstellungen
mögliche ausgedachte (theoretische) Grenz-Bedingungen oder Spezial-Werte (Killer-Werte)
Tests sollten selbstständig / eigenständig gemacht werden

Modul- bzw. Funktions-Tests sind auch Computer-gestützt möglich

für Eclipse geht das mit JUnit

<u>Steuerung und Definition über Annotationen</u>

Annotationen dienen der Deklaration der Test's
sowie dem Zusammenfassen von Test's zu Suiten

Test-Setup

- **%before** — Defintion von herzustellenden Bedingung / Situationen für den Test (Schaffung der Ausgangs-Bedingungen)
- **%test** — eingentlicher Test-Code
- **%after** — Code, der nach den Tests ausgeführt werden soll, z.B. zum Aufräumen, …

Assertions

- **assertEquals(message, expectedValue,↵ actualValue, tolerance)**
- **assertTrue(message, condition)**
- **assertFalse(message, condition)**
- **assertNotNull(message, objekt)**
- **assertNull(message, objekt)**

erstellen von TestSuite's
TestSuite's legen die Reihenfolge der Abarbeitungen von Test's fest
die Test's werden in logisch zusammenghörende Einheiten struktoriert und organisiert

```
…
%RunWitch(Suite.class)
%Suite.SuiteClasses({
    JunitTest1.class,
    JunitTest2.class
})
…
```

2. informatische Problemstellungen in JAVA

It's not a bug,
it's a feature.
Redesart von Programmierern und des Software-Marketing

weiterführende Links:
https://www.cs.usfca.edu/~galles/visualization/Algorithms.html (Visualisierung von Algorithmen)

2.0.1. Fehler sind menschlich

man muss aber mit ihnen umgehen können

die Aussage: "Das hat der Computer so gemacht." zeigt Ignoranz und Computer-Gäubigekit der Nutzer

Nutzer muss "Ungereimtheiten" dokumentieren und ev. melden

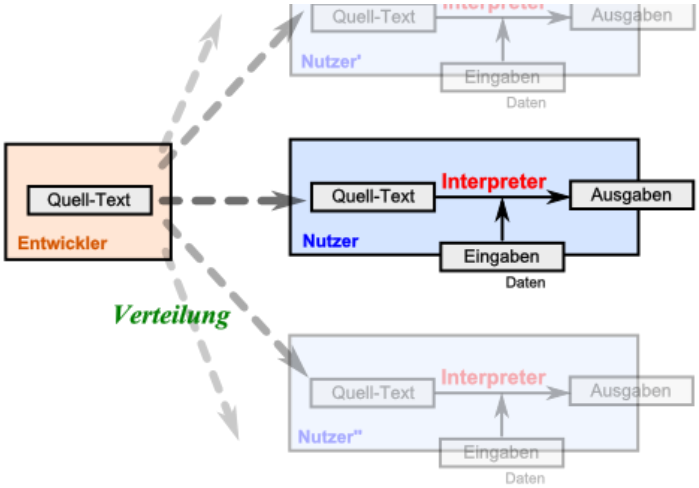
"Software ist niemals Fehler-frei, es sei denn sie leistet nichts und selbst da können noch Fehler auftauchen."
z.B. ne Endlosschleife ohne Inhalt, die sich nicht abbrechen lässt

die häufigsten Ursachen für Software-Fehler:

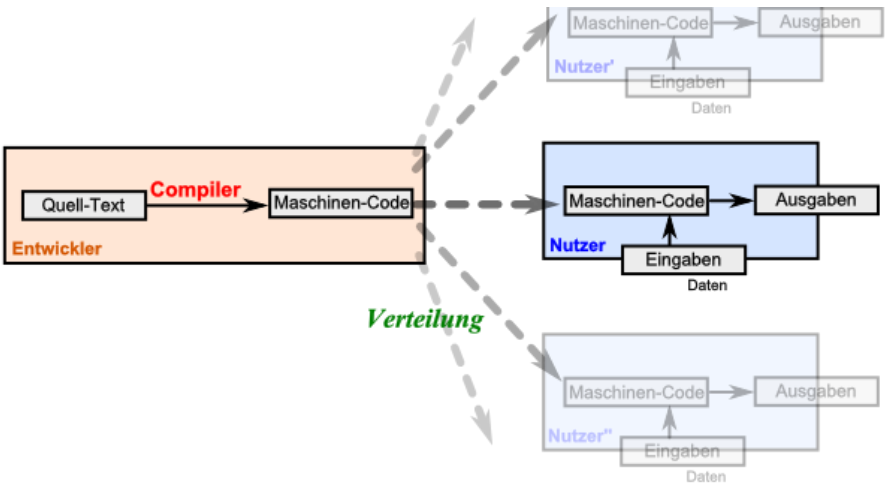
- Tipp-Fehler
- Entwurfs-Fehler
- fehlende Sicherheits-Abfragen (bei Eingaben)
- schlecht definierte oder ignorierte Schnittstellen
- Fehl-Interpretation von Eingabe- und / oder Ausgabe-Daten
- ungeprüfte Übernahme von altem Code in neue / neu dimensionierte Systeme
- fehlende Passung von Hard- und Software
- (nummerische) Rundungsfehler
- fehlende Test's
- Gigantismus
- unterschätzte Aufgabenstellung und grobfertige – unter Zeitdruck entstandene - Teillösungen

2.1. Compiler und Interpreter

Schauen wir uns (wiederholend) die Funktionsweise der klassischen Übersetzungs-Techniken an.
Die einfachste und wohl auch älteste Form der Übersetzung eines Quell-Textes in den notwendigen Maschinen-Code ist ein Interpreter.
Der Interpreter braucht den orginalen Quell-Text und übersetzt ihn beim Anwender / bei der Anwendung zeilenweise.



Jeder Nutzer brauchte also einen Interpreter auf seinem Rechner. Ein Beispiele hierfür sind die klassischen (alten) BASIC-Systeme.
Die Interpreter sind meist recht einfache Programme. Dafür sind sie relativ langsam, jeder Nutzer musste sie kaufen und für jedes Betriebssystem und jeder Hardware-Plattform musste eine neue Interpreter-Version bereitgestellt werden. Programmierer gaben zudem ihr ganzes Know-how (den Quell-Text) aus der Hand. Einmal verkauft stand dem beliebigen Kopieren / unentgeldlichen Weitergeben nicht's mehr im Weg. Die gesamte Übersetzungs-Zeit liegt also beim Anwender. Der Nutzer musste auch schon ein bisschen mit Interpreter und Rechner auskennen, um ein Programm zum Laufen zu bringen.
Nutzerfreundlicher ist die Übersetzung mittels Compiler. Der Compiler arbeitet beim Entwickler. Der Quell-Text wird jier in Maschinen-Code übertragen und dann zur Nutzung beim Anwender verteilt. Praktisch alle installierbaren Programme – vom Betriebssystem bis zur Textverarbeitung oder einem Spiel – kommen in Maschinen-Code-Form zum Nutzer (EXE-Datei).



Jetzt musste der Entwicker das Übersetzer-Programm kaufen (und den Preis dafür natürlich an seine Kunden weitergeben). Für jedes Betriebssystem und jede Hardware-Basis musste er nun eine verteilbare Datei (EXE-Datei) erzeugen. Die gesamte Übersetzungs-Zeit(en) liegt / liegen beim Entwickler. Die Programmier-Interna sind gut geschützt un der Nutzer kann die Programm-Datei direkt starten.

Compiler und Interpreter können immer nur Syntax-Fehler prüfen.

der Inhalt (die Semantik) muss letztendlich immer ein Mensch prüfen

syntaktische exakte aber semantisch unsinnige Aussagen:
Das vierstöckige Haus hat 23 Etagen.
Das Holzstück ist durchsichtig.
Die Zahl soll größer als 100 und kleiner als 50 sein.
Die Zeichenkette soll leer oder nicht leer sein.

Exkurs: Der teuerste Bindestrich der Welt

Die Venus Mariner Mission ging 1962 schief, weil ein Programmierer an einer Stelle einen Bindestrich nicht gesetzt hatte. Das FORTRAN-Programm berechte daraufhin Werte falsch und der materielle Schaden lag letztendlich bei 80 Mio. Dollar.

Ein ähnliches Debakel erlebten die Programmierer des APOLLO-Projektes (*???*).
Hier hatte ein Programmierer statt eines Komma's in einem Schleifen-Konstrukt einen Punkt gesetzt.

bei professionellen Programmierern rechnet man mit 25 Fehler pro 1'000 Zeilen Quellcode
das alles trotz der üblichen Test's
durch sehr intensive Test's hat man es geschaft bei der "Space Shuttle"-Software auf nur 1 Fehler pro 10'000 Zeilen zu kommen

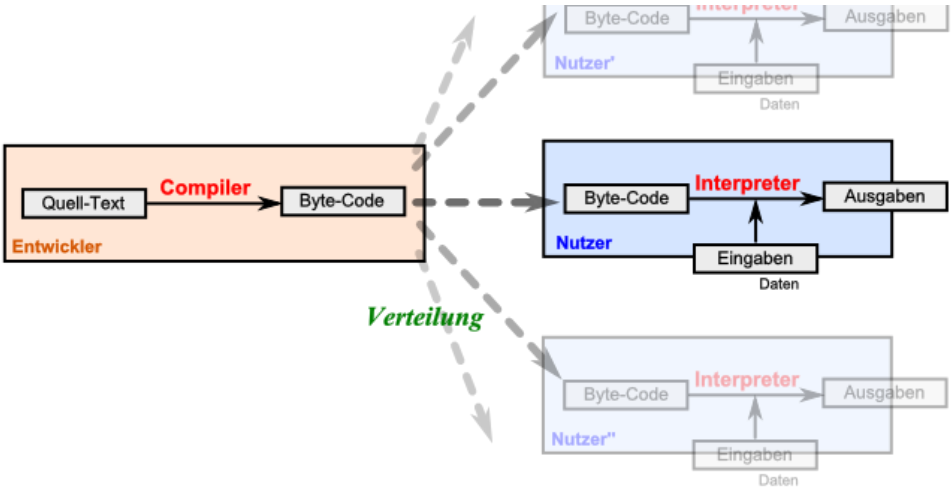
spezielle Programmier-Techniken – meist z.B. Programmieren im direktem Zweier-Team – reduziert die Fehlerzahl auf nur 2 Fehler pro 1'000 Zeilen
abwechselnd programmiert / tippt einer und der andere schaut über die Schulter und hinterfragt / korrigiert gleich; sofortiges Ausdiskutieren von unterschiedlichen Lösungs-Ansätzen / Abwägungen zum besten Algorithmus

geschätzte Zahl von Fehlern:
bei Handy-Software 600 Fehler auf 200'000 Quellcode-Zeilen

Windows 95 mit 10 Mio. Zeilen Code geschätzte 20'000 Fehler
genug Potential für Update's
dazu kommen Schwachstellen, die es einem böswilligen Nutzer ermöglich, das System zu manipulieren
als echte Fehler kann man die Schwachstellen nicht einstufen, sie sind eher schwache Programmier-Leistungen sowie Fehleinschätzungen durch die Programmierer

2.1.3. JAVA als spezielle Compiler-Interpreter-Kombination

JAVA geht einen anderen – kombinierten Weg, um die Vorteile beider Übersetzungs-techniken zu vereinen. Je nach Sichtweise (aus Programmierer- oder Nutzer-Sicht) ist das auch gelungen oder eben nicht gelungen.
Der Quellcode wird von einem (Pre-)Compiler (javac) zuerst in eine für JAVA-Ausführ-Systeme besser nutzbare Form – den ByteCode gebracht. Das hat den Vorteil, dass für jede Betriebssystem-Welt nur ein spezieller JAVA-Compiler existieren muss. Dieser kann eigen-ständig weiterentwickelt werden und braucht auch nur auf Entwicklungs-Systemen installiert sein.



Der ByteCode ist Plattform-unabhängig. Er kann zwischen den verschiedenen Betriebssys-tem-Welten ausgetauscht werden. Gerade das macht die weite Verbreitung von JAVA und seinen Erfolg auch aus. Auf der lokalen Station – und das können noch viel mehr Betriebs-systems sein – wird nur der ByteCode interpretiert. D.h. hier übersetzt die Runtime-Umgebung den ByteCode in den lokalen Maschinen-Code.
Programmierer und Nutzer teilen sich die die Übersetzungs-Zeit(en).
Die Runtime-Umgebung kann und wird frei verteilt und steht meist für die verschiedensten System zur Verfügung. Das kann hinunter bis zu "embedded Systems" – also kleinen "ein-gebetteten System" in Geräten gehen. Allerdings können diese Systeme nicht mehr alle JA-VA-Programme ausführen, aber es macht ja auch keinen Sinn, auf einem Temperatur- oder Klima-Regler an der Wand eine hochkomplexe Animation auszuführen.
Auf eingebetteten System wird wahrscheinlich niemand direkt ein Programm entwickeln – schon deshalb nicht, weil z.B. eine ordentliche Tastatur fehlt. Das wird dann eben auf einem besseren, geeigeneten Rechner erledigt und dann nur die ByteCode-Datei auf das embed-ded System übertragen. Für ein einfacheres Progrtammieren verfügen viele Entwicklungs-Systeme über Emulatoren für die embedded Systems. Dadurch kann man gleich den Quell-Code testen ohne ein passendes Endgerät zu besitzen und / oder es vielleicht durch eine fehlerhafte Programmierung zu zerstören (weil man vielleicht bei der Temperatur-Steigerung keine Grenze einprogrammiert hat).

Aufgaben:

1.

2. Vergleichen Sie die verschiedenen Übersetzungs-Techniken von Quell-Texten in ausführbare Programme!

3.

2.x. komplexe Daten-Strukturen

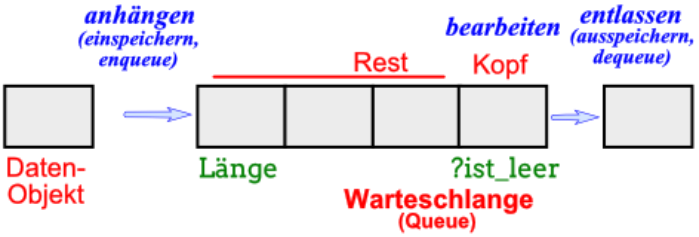
2.x.y. Zeiger-orientierte Daten-Strukturen

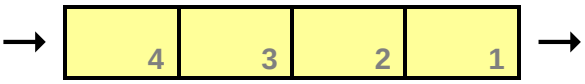
2.x.y. Warteschlangen bzw. FIFO-Speicher

First IN First OUT-Prinzip
was zuerst eingespeichert wurde, wird auch als erstes wieder benutzt / entfernt usw.
Prinzip der klassischen Warteschlage an einer Kasse usw.

auch Queues genannt

Elemenete werden vorne/oben angehängt und hinten/unten wieder entnommen





typische Operation:

- enqueue … ablegen / einspeichern / auflegen eines Element's
- dequeue … entnehmen / ausspeichern / abheben eines Element's

typische Anwendungen:

- Schlange am Bus, vorm Museum, an der Kasse
- Drucker-Warteschlange (Liste der abzuarbeitenden Druck-Aufträge)
- Musik-Playlists
-

Sonderfall:
doppelt verkettete Warteschlangen / Warteschlange mit zwei Enden
Deques genannt (Double-ended Queues)

typische Operation:

- push … ablegen / einspeichern / auflegen eines Element's (am hinteren Ende)

- pop … entnehmen / ausspeichern / abheben eines Element's (am hinteren Ende)
- put … ablegen / einspeichern / auflegen eines Element's (am vorderen Ende)
- get … entnehmen / ausspeichern / abheben eines Element's (am vorderen Ende)
-

-

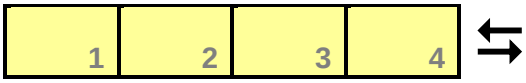
Aufgaben:

1. *An einer leeren Deque (Ende links; Anfang rechts) werden die folgenden Oprationen ausgeführt:*
   push(2); push(3); put(4); put(5); push(6); push(7); put(7); put(8)
   Geben Sie die Reihenfolge der Elemente (von links gelesen) in der Deque an!
2. *Nun erfolgt eine Ausgabe der Elemente aus der Deque (von Aufg.1) mittels der folgenden Operationen:*
   get(); get(); pop(); get(); pop()
   Welche Ausgaben erhalten Sie?
   Wie ist die (von links gelesene) Reihenfolge der Elemente ind Deque am Ende?
3.

2.x.y. Keller bzw. LIFO-Speicher

Last IN First OUT-Prinzip
praktisch als Stapel vorstellbar, auf den nur über die oberste Stelle zugegriffen werden kann
auch Stack genannt

Die Elemente werden vorne/oben angefügt und auch wieder vorne/oben entnommen.



typische Operation:

- push … ablegen / einspeichern / auflegen eines Element's
- pop … entnehmen / ausspeichern / abheben eines Element's

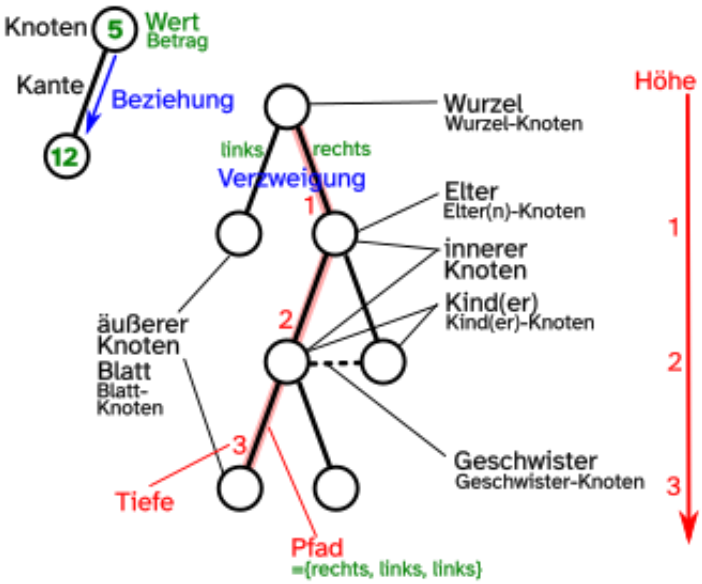
typische Anwendungen:

- Teller-Stapel in der Küche
- "zurück"-Taste beim Browser
- Undo-Funktion (in beliebigen Programmen)

Aufgaben:

1. ***An einem Stack werden die folgenden Operationen durchgeführt (die Stapel-Spitze befindet sich rechts):***
   push(1); push(2), push(4); pop(); push(3), pop(); pop(); push(4)
   Geben Sie an, in welcher Reihenfolge (von links gelesen) die Zahlen stehen!
2.
3.

2.x.y. Bäume



Grund-Begriffe zu Bäumen	
• **Knoten** **node**	Objekt / Entität im Baum
• **Wert** **value**	konkrete Objekt-Eigenschafts-Ausprägung
• **Kante** **edge**	Verbindung zwischen zwei Konten
• **Beziehung** **relationship**	(Zu-)Ordnung-Richtung einer Kante (des Baum's insgesamt)
• **Wurzel(-Knoten)** **root (node)**	einzelner Ausgangs-Knoten des Baum's
• **Elter(-Knoten)** **parent (node)**	Knoten, der mindestens ein Kind-Knoten hat
• **Kind(er)(-Knoten)** **child (node)**	Knoten, der einen Eltern-Knoten hat
• **Geschwister** **sibling (nodes)**	alle Knoten, die einen gemeinsamen Eltern-Knoten besitzen
• **Blatt** **leaf**	Knoten am Ende des Baum's Knoten, die keine Kind-Knoten haben
• **Pfad** **path**	Weg (Liste der Entscheidungen / Verzweigungen) von der Wurzel zum benannten Knoten
• **Höhe (des Baum's)**	maximale Anzahl von Knoten-Ebenen im Baum

Suchen in Binär-Bäumen

jeder Knoten hat also maximal 2 Kinder

Voraussetzung: sortiert (links stehen kleinere Werte, rechts größere)
Werte (Elemente) kommen immer nur 1x vor (ansonsten nur Finden eines Elementes!)

1. Start an der Wurzel → aktueller Knoten
2. IST das Such-Element gleich dem Wert des aktuellen Knoten's, DANN FERTIG / STOP ("gefunden")
3. IST das Such-Element kleiner als der Wert des aktuellen Knoten, DANN wird links weiter-gesucht
4. ANSONSTEN wird rechts weiter gemacht
5. IST kein Knoten mehr vorhanden (also Blatt), DANN STOP ("nicht gefunden")
6. gefundener Knoten ist aktueller Knoten und WEITER bei 2.

Suchen in (unsortierten) Bäumen

z.B. zur Anzeige aller Elemente eines Baum's

Traversieren / Durchsuchen
gemeint ist das Durchwandern / besuchen aller Elemente des Baum's

Tiefen-Suche
Pre-Order
Start beim Wurzelknoten
zuerst (pre order) Eltern-Element bearbeiten (z.B. Ausgeben, Vergleichen, …)
wenn vorhanden dann mit linkem Teilbaum fortsetzen ansonsten
wenn vorhanden mit dem rechten Teilbaum weitermachen ansonsten eine Ebene nach oben gehen und dort mit rechtem Teilbaum weitermachen

gut zum Kopieren von Bäumen geeignet

In-Order
Start beim Wurzelknoten
solange nach links gehen, bis ein Blatt gefunden wurde
Blatt-Element bearbeiten (z.B. Ausgeben, Vergleichen, …)
dannach (in order) Eltern-Element bearbeiten (z.B. Ausgeben, Vergleichen, …)
wenn vorhanden mit dem rechten Teilbaum weitermachen ansonsten eine Ebene nach oben gehen und dort mit rechtem Teilbaum weitermachen

ergibt in einem sortierten Baum eine sortierte Ausgabe der Elemente

Post-Order
Start beim Wurzelknoten
solange nach links gehen, bis ein Blatt gefunden wurde
Blatt-Element bearbeiten (z.B. Ausgeben, Vergleichen, …)
am Eltern-Knoten prüfen, ob rechter Teilbaum / oder Blatt existiert, dann im rechten Teilbaum Verfahren starten
sonst Eltern-Knoten bearbeiten

gut geeignet für das Löschen eines Baum

Bearbeiten von Strukturen, die Operationen zur UPN (umgedrehte polnische Notation) darstellen

Breiten-Suche
es wird solange im Bereich des Eltern-Knoten gearbeit, wie es geht

Start beim Wurzelknoten
Blatt bearbeiten (z.B. Ausgeben, Vergleichen, …)
nach links gehen
wenn ein Blatt gefunden wurde, dann Bearbeiten
wenn Element rechts vorhenden ist, bearbeiten
dann auf der gleichen Ebene im rechten Teil-Baum fortsetzen

Einfügen in Binär-Bäumen

jeder Knoten hat also maximal 2 Kinder

Voraussetzung: sortiert (links stehen kleinere Werte, rechts größere)
Werte (Elemente) kommen immer nur 1x vor (ansonsten nur Finden eines Elementes!)

1. Start an der Wurzel → aktueller Knoten
2. IST das Einfüge-Element gleich dem Wert des aktuellen Knoten's, DANN FERTIG / STOP ("vorhanden")
3. IST das Einfüge-Element kleiner als der Wert des aktuellen Knoten, DANN wird links weitergesucht
4. ANSONSTEN wird rechts weiter gemacht
5. IST kein Knoten vorhanden (also Blatt), DANN Anlegen eines neue Knoten's; FERTIG STOP ("eingefügt")
6. gefundener Knoten ist aktueller Knoten und WEITER bei 2.

Löschen in Binär-Bäumen

jeder Knoten hat also maximal 2 Kinder

Voraussetzung: sortiert (links stehen kleinere Werte, rechts größere)
Werte (Elemente) kommen immer nur 1x vor (ansonsten nur Finden eines Elementes!)

Unterscheidung notw. (Löschen eines Blattes, eines Knoten's oder der Wurzel)

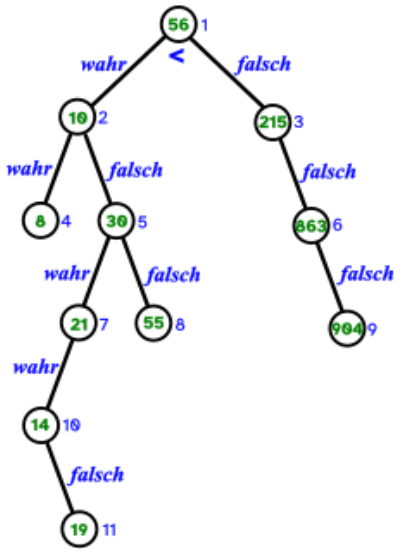
1. Start an der Wurzel → aktueller Knoten

Aufgaben:

1. Übernehmen Sie nebenstehenden Baum! Beschriften bzw. kennzeichnen Sie darin alle

oben genannten Teile eines Baum's!

2. *Welche Eigenschaften / Begriffe kann man folgenden Objekten zuordnen? Begründen Sie immer kurz Ihre Zuordnung!*
   a) ist Elter-Konoten
   b) ist Kind-Knoten
   c) ist Blatt
   d) ist Wurzel-Konten
3. *Überlegen Sie sich, ob die folgenden Objekte in den Baum eingefügt werden können! Begründen Sie immer Ihre Entscheidung! Ergänzen Sie passende Objekte im Baum!*
   a) 362 b) 12 c) 1002



   d) 10 e) 56 f) "fünfzehn" g) π h Wurzel(49)
4. *Zeichnen Sie einen Baum, der einen Wurzel-Knoten und ein Blatt besitzt! Der Baum darf keine Eltern- und keine Kind-Knoten enthalten. Geben Sie die Höhe des Baum's und den Pfad zum Blatt an!*

balancierte Bäume

auch AVL-Bäume genannt, nach den Entwicklern ()

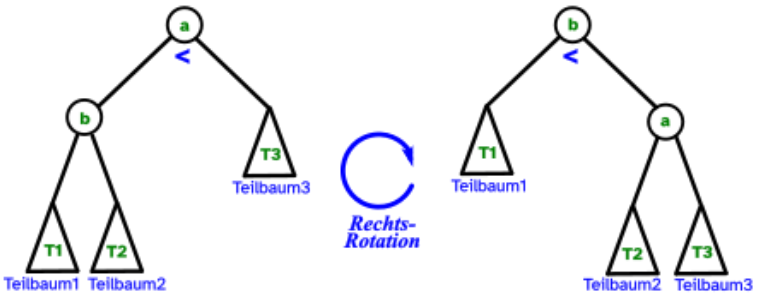
Bedingung für einen balancierten Baum ist die Aussage, dass die rechte Höhe (des Teilbaum's) sich von der linken Höhe maximal um 1 differiert

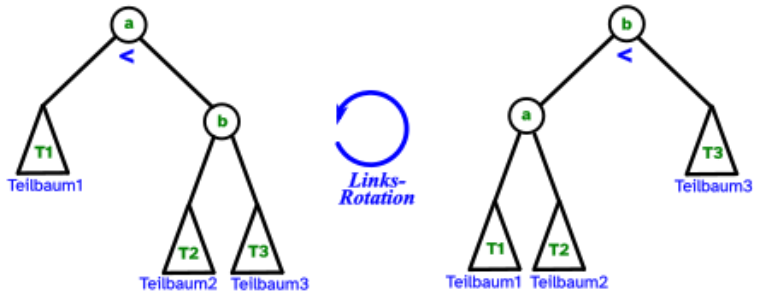
Für **jeden Knoten** gilt, dass die **Höhe** des **tieferen** Teilbaums **gleich oder 1 größer** als die des **flacheren** Teilbaums sein muss.

ist der Baum nicht balanciert, dann erfolgt eine Rotation
→ das folgende Element auf der Seite mit der größeren Höhe, wird neuer Eltern- bzw. Wurzel-Knoten

bei Rotation nach rechts (linker Teil-Baum ist höher als rechter)

- linker Knoten (b) von dem aktuellen Knoten a wird neuer aktueller Knoten
- der rechte Zweig von Knoten b wird neuer linker Zweig von Knoten a





Einfügen eines Element's
ev. Kombination aus Links- und rechts-Rotationen für eine Einfüge-Operation notwendig

Löschen eines Element's
Variante 1 (Blatt löschen)
Element entfernen
bis zur Wurzel balancieren

Variante 2 (innerer Knoten)
Element entfernen
nächst größeres Element suchen
ist dies ein Blatt dann entfernen und als neues inneres Knoten-Element einbinden
sonst …
bis zur Wurzel balancieren

in JAVA selbst gibt es keine Implementierung von AVL-Bäumen
freie nutzbare Bibliothek ist JGraphT

Code-Beispiel für gespeicherte Integer im Baum

```
import org.jgrapht.util.*;

public class AVLTreeBeispiel {
    public static void main(String[] args) {
        AVLTree<Integer> baum = new AVLTree<>();
        for (int i = 0; i < 10; i++) {
            baum.addMax(i);
        }
        System.out.println(baum);
    }
}
```

weiterführende Links:
https://www.cs.usfca.edu/~galles/visualization/AVLtree.html (Simulation zu balanzierten Bäumen)
(JGraphT – freie Bibliothek zu AVL-Bäumen)

weitere Bäume

Rot-Schwarz-Bäume

ebenfalls balancierter binärer Suchbaum
praktisch immer Schicht-weise gefärbt
jeder AVL-Baum ist auch ein Rot-Schwarz-Baum
(es können aber Rot-Schwarz-Bäume existieren, die keine AVL-Bäume sind!)

Bedingungen:

- zwei rote Knoten dürfen nicht hintereinander folgen
- jeder mögliche Pfad von der Wurzel zu den Blättern muss die gleiche Anzahl an schwarzen Knoten enthalten

zwei schwarze Knoten dürfen aufeinander folgen

in JAVA in Form von TreeMap und TreeSet implementiert

```
TreeSet<Integer> meineSammlung = new TreeSet<>();
boolean erg;
erg = meine Sammlung.add(1);
erg = meine Sammlung.add(1);
erg = meine Sammlung.add(2);

int anzahl = meineSammlung.size();
erg = meine Sammlung.contains(1);
erg = meine Sammlung.isEmpty();
erg = meine Sammlung.remove(1);
```

Vorteil von TreeSet und TreeMap ist die geordnete / sortierte Speicherung der Objekte (intern)

```
TreeSet<Integer> meineSammlung = new TreeSet<>();
for (int i = 0; i < 10; i++) {
   meineSammlung.add(i);
}
System.out.println(meineSammlung);


meineSammlung.headSet(5, true);    // zeigt linken Teil bis 5
meineSammlung.descendingSet();    // gibt Set in umgedrehter Reihenfolge aus
meineSammlung.higher(6);    // gibt Nachfolger von 6 zurück
meineSammlung.lower(6);    // gibt Vorgänger von 6 zurück
```

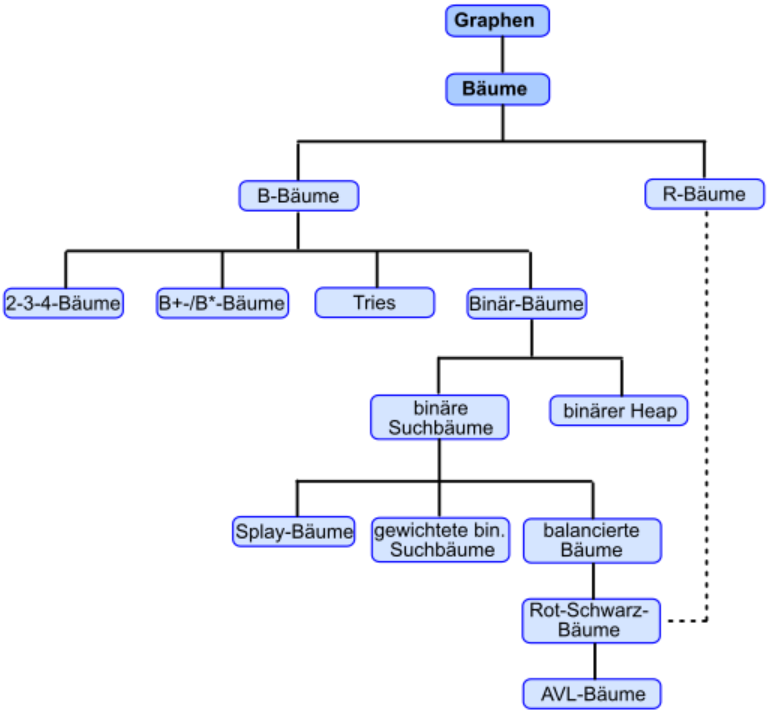
Präfix-Bäume – Tries

gut für Verarbeitung von Worten und Wort-Bestandteilen
z.B. als Wort-Vorschlags-System (bei Suchmaschinen oder bei der Adresseingabe in Navi's)

Implementierung in der Apache Commons Collection als PatriciaTrie

Komplexität in Bäumen im Vergleich

Daten-Struktur	Einfügen	Suchen	Löschen	
Liste, unsortiert	***O***(1)	***O***(n)	***O***(1)	
Liste, sortiert	***O***(log(n))	***O***(log(n))	***O***(1)	
binärer Such-Baum	***O***(h)	***O***(h)	***O***(1)	
balancierter Such-Baum	***O***(log(n))	***O***(log(n))	***O***(log(n))	



Versuch der Einteilung von Bäumen / Baum-Strukturen
(nach OpenHPI-Kurs "Java Algorithmen"

2.x.y. Ringe

2.x.y. Puffer

Puffer-Verwaltung
Puffer nimmt nur begrenzte Anzahl Objekte auf
Produzent füllt Puffer Stück-weise
Konsument entnimmt dem Puffer Stück-weise

```
public interface Buffer{
    public void put(Object o) throws InterruptException;
    public Object get()throws InterruptException;
}
```

abgetrenntes Interface, um alternative Implementierungen zu ermöglichen
Puffer hat feste Größe size; nimmt beliebige Objekte auf, ist als Ring-Speicher organisiert
notify nach put, falls konsumierender Prozess wartet
notify nach get, falls produzierender Prozess wartet

```
class BufferImpl implements Buffer{
    protected Object[] buf;
    protected int in = 0;
    protected int out = 0;
    protected int count = 0;
    protected int size;
    BufferImpl(int size){
        this.size = size;
        buf = new Object[size];
    }

    public synchronized void put(Object o) throws InterruptException{
        while(count == size) wait();
        Object o = buf[out];
        buf[in] = 0;
        ++count;
        in = (in + 1)%size;
        notify();
    }

    public synchronized Object get()throws InterruptException{
        while(count == 0) wait();
        Object o = buf[out];
        buf[out] = null;
        --count;
        out = (out + 1)%size;
        notify();
        return(o);
}
```

```
class Producer implements runnable{
    Buffer buf;
    Object item;
    Producer(Buffer b) {buf = b};
    public void run(){
        try{
            while(true){
```

```
                buf.put(new item);
            }
        catch (InteruptedException e){}
        }
    }
}
```

Produzent liefert Objekte (new item) in Endlos-Schleife und legt sie in Puffer ab
Konsument entfernt Objekte aus dem Puffer in Endlos-Schleife (und tut nicht weiter damit)

```
class Consumer implements runnable{
    Buffer buf;
    Object item;
    Consumer(Buffer b) {buf = b};
    public void run(){
        try{
            while(true){
                buf.get();
            }
        catch (InteruptedException e){}
        }
    }
}
```

Q: ww.informatik.uni-hamburg.de/~neumann/P3-WS-2000/P3-Teil3.pdf (leicht geändert)

2.x.y. Graphen

jeder Baum ist ein Graph, aber nicht jeder Graph ein Baum

Knoten

Kanten

Graph ist gerichtet, wenn die Kanten eine oder auch zwei Zuweisungs-Richtung(en) besitzen

Graph ist gewichtet, wenn den Kanten ein Gewicht (ein Wert, eine Länge, …) zugewiesen wurde

typisches informatisches Problem: Problem des Handelsreisenden (Traveling Salesmen Problem)

gesucht ist ein kürzester Weg zwischen Knoten in einem gewichteten Graphen

z.B.Problemstellung:
alle Städte genau 1x besuchen

typische Laufzeit / Komplexität: ***O***()

2.x. Suchen

Suche in (un)sortierten Listen

wir gehen davon aus, unsere Daten-Liste ist nicht sortiert

lineare Suche

mit foreach-Konstrukt

```
String[] liste = {"ObjA", "ObjB", "ObjC", "ObjD", "ObjD"}
suchObj = "ObjD";

boolean gefunden = false;
for (String element : liste) {
   if (element == suchObj) {
      gefunden = true;
   }
}
System.out.println(gefunden);
```

geht komplette Datenstruktur durch
gibt nur zurück, ob etwas gefunden wurde oder eben nicht

lineare Suche mit Positions-Bestimmung und Abbruch

klassische for-Schleife

```
…
int position = -1
for (int i = 0; i < liste.length; i++) {
   if (liste[i] == suchObj) {
      position = i;
      break;
   }
}
System.out.println(position);
```

geht komplette Datenstruktur nur dann durch, wenn Such-Objekt nicht enthalten ist (schlechtester Fall)
liefert neben dem Such-Erfolg (gefunden / nicht gefunden (über die Auswertung des int-Wertes)) auch die erste Fund-Position des Such-Objektes zurück (bester Fall)
durchschnittlicher Aufwand ist von linearer Komplexität und liegt bei der halben Listen-Länge

Suche in Kollektionen

```
List<String> liste = new ArrayList<>();
liste.add("ObjA");
liste.add("ObjB");
liste.add("ObjC");
liste.add("ObjD");
liste.add("ObjE");

suchObj = "ObjD";
System.out.println(liste.contains(suchObj));
System.out,println(liste.indexOf(suchObj));
```

contains prüft auf Anwesenheit des Such-Objektes (Argument) in der Kollektion
indexOf liefert die Position in der Kollektion (oder -1, wenn nicht)
schneller, da maschinennaher programmiert (intern linear)

Suche in sortierten Listen

binäre Suche (nach Divide-and-Conquer-Prinzip)

geht schneller; durchschnittlicher Aufwand liegt bei Wurzel aus der Listen-Länge

Aufgaben:

1. ***Ein Liste enthält 128 Objekte. Die Sortierung ist unbekannt. Geben Sie für den schlechtesten Fall an, wieviele Such-Aktionen (Objekt-Vergleiche) getätigt werden müssen, wenn:***
   a) eine einfache lineare Suche ohne Abbruch
   b) eine einfache lineare Suche mit Abbruch
   c) eine binäre Suche
   genutzt wird?
2. ***Ein Liste enthält 128 Objekte. Die Sortierung ist unbekannt. Geben Sie für den besten Fall an, wieviele Such-Aktionen (Objekt-Vergleiche) getätigt werden müssen, wenn:***
   a) eine einfache lineare Suche ohne Abbruch
   b) eine einfache lineare Suche mit Abbruch
   c) eine binäre Suche
   genutzt wird?
3. ***Ein Liste enthält 128 Objekte. Die Sortierung ist unbekannt. Geben Sie für den durchschnittlichen Fall an, wieviele Such-Aktionen (Objekt-Vergleiche) getätigt werden müssen, wenn:***
   a) eine einfache lineare Suche ohne Abbruch
   b) eine einfache lineare Suche mit Abbruch
   c) eine binäre Suche
   genutzt wird?
4. ***Füllen Sie die Tabelle für eine Liste mit 128 Elementen aus, wenn bekannt ist, dass die Liste (aufsteigend) sortiert ist!***

	schlechtester Fall	**bester Fall**	**durchschnittlicher Fall**
einfache lineare Suche (ohne Abbruch)			
einfache lineare Suche mit Abbruch			
binäre Suche			

Problem Vergleichen

für einzelne Daten praktisch kein Problem, da die klassischen vergleiche greifen
kompliziert wird es bei Objekten mit mehreren Attributen
Welche sollen verglichen werden?
Welches Attribut hat ev. Vorrang?
Was ist, wie / wann größer / kleiner?

z.B. Problem schon bei Wahrheitswerten
true und true sowie false und false sind gleich
ungleich ebenfalls klar
aber wie sieht es bei kleiner oder größer als aus?
in vielen Programmiersprachen ist FALSCH gleich 0 gesetzt und WAHR 1 oder ungleich 0
in Java sind die Ausdrücke true und false eigene Objekte ohne Zahlen-Wert
beim vergleich wird true aber als größer als false betrachtet (ist so definiert!)

jedes Objekt erbt von der Klasse Objekt die equals()-Methode
kann allerdings von übergeordneten Klassen, von der unsere eigene Klasse abgeleitet ist, schon überschrieben (umdefiniert) worden sein

Vergleich Figuren
bekannt sind Länge, Breite (jeweils Maximum) und Anzahl der Ecken

```
public class Figur implements Comparable {
   int breite;
   int laenge;
   int anzahlEcken;

   Figur(int breite, int laenge, int anzahlEcken) {
      this.breite = breite;
      this.laenge = laenge;
      this.anzahlEcken = anzahlEcken;

   @Override
   public String toString() { … }
   }

…
}
```

<u>mit equals()</u>

Vergleich von zwei Objekten auf Objekt-Ebene
nur zwei identische Objekte sind equal (ein Objekt muss also direkt oder indirekt auf das andere zeigen (Referenz); praktisch die gleiche Lage im Speicher

```
   @Override
   public boolean equals(Object obj) {
      if (this == obj)
         return true;
```

```
        if (obj == null)
            return false;
        if (getClass != obj.getClass())
            return false;
        Figur anderesObj = (Figur) obj
        return this.breite * this.laenge == anderesObj.breite * anderesObj.laenge;
    }
```

hier erfolgt also die Gleichheits.Bestimmung am Ende über die Fläche der Objekte
wichtig ist, dass die Deklarations-Zeile:

```
    public boolean equals(Object obj) {
```

unverändert (von der Vorgänger-Klasse) übernommen wird

meist muss auch noch **hashCode()**-Methode mit überschrieben werden

```
    @Override
    public int hashCode() {
        return name.hashCode();
    }
```

ermittelt den HashCode nur über ein Attribut – hier Name
meist wird aber das ganze Objekt für die Erzeugung eines HashCode's benutzt

```
    @Override
    public int hashCode() {
        return Objects.hash(breite, laenge, anzahlEcken);
    }
```

mindestens sollten aber alle relevanten Attribute einbezogen werden

<u>mit compareTo()</u>

Größer-, Kleiner(Gleich)- und Indentitäts-Vergleich

```
    @Override
    public int compareTo(Object obj) {
        if (this.equals(obj)) {
            return 0;
        }
        if (this.breite * this.laenge > obj.breite * obj.laenge) {
            return 1;
        } else {
            return -1;
        }
    }
```

compareTo-Paradoxon

verbesserte equals()-Methode mit Vergleich aller Attribute

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass != obj.getClass())
        return false;
    Figur anderesObj = (Figur) obj
    return breite == anderesObj.breite &&
            laenge == anderesObj.laenge &&
            anzahlEcken == anzahlEcken;
}
```

beim Einbeziehen mehrerer Attribute entsteht dann aber das Problem, was ist nun genau größer oder kleiner
Sind es die Anzahl der Ecken, die wichtiger als die Breite / Höhe ist, oder bleibt die Fläche als Primat?
Hier muss der Programmierer entscheiden.
Mit bedacht werden sollte unbedingt, das wieder andere Klasse aus der aktuellen Klasse abgeleitet werden könnten. Ob dann die Methode immer noch so verstanden wird, sollte geprüft werden.

```
@Override
public int compareTo(Object obj) {
    if (this.equals(obj)) {
        return 0;
    }
    if (this.breite * this.laenge > obj.breite * obj.laenge) {
        return 1;
    } else {
        return -1;
    }
}
```

bei dieser Definition sind zwei Figuren, die die gleiche Fläche besitzen gleich (Rückgabe-Wert: 0) und wenn sie unterschiedliche Flächen eben entweder größer oder kleiner
problematisch wird es, wenn die Flächen nach einem Tausch von Fläche und Breite immer noch gleich sind
in beiden Fällen erhalten wir die gleiche Rückgabe (-1) – also kleiner
(weil eben die Gleichheit der Fläche automatisch in den ELSE-Zweig führt
der Programmierer muss hier ev. – bezogen auf seine Klasse – Korrekturen vornehmen

2.x. Sortieren

typischer-weise liegen Daten unsortiert (praktisch zufällig angeordnet)

Selection-Sort

benötigt Länge-1 Tausch-Aktionen / Such-Durchläufe
sucht immer das kleinste Element in der unsortierten Liste (zu Anfang eben in der gesamten Liste) → Minimum
tauscht das kleineste Objekt auf den Anfang der unsortierten Liste und das Objekt, das hier vorher war auf die Position, wo sich das gefundene kleinste Objekt befunden hat → Tauschen
danach kann mit der um Eins verkürzten Restliste weitergemacht werden

benötigt zwei grundlegende Operationen / Funktionen, die auch als Funktionen ausgelagert werden können, aber auch in einer Hauptschleife angeordnet werden können

alles in einer Sortier-Funktion

```
int[] liste = {6, 3, 5, 7, 8, 1, 9, 2, 4}
```

mit speziellen Hilfs-Funktionen

```
private int[] liste = {6, 3, 5, 7, 8, 1, 9, 2, 4}

private int gibMinPos(int aktMinPos) {
    int minPos = aktMinPos;
    for (int i = aktMinPos+1; i < liste.length; i++) {
        if (liste[i] < liste[minPos]) {
            minPos = i;
        }
    }
    return minPos;
}

private void tausche(int alteMinPos, int neueMinPos) {
    int merker = liste[alteMinPos];
    liste[alteMinPos] = liste[neueMinPos];
    liste[neueMinPos] = merker;
}

private void selectionSort() {
    for (int i = 0; i < liste.length-1; i++) {
        int minPos = gibMinPos(i);
        tausche(i, minPos);
    }
}
```

Merkmale:

- ??? stabil
- Komplexität: ***O***(n^2)
-

Vorteile:

- ??? bei gleich-wertigen Elementen bleibt ursprüngliche Reihenfolge erhalten
-

Nachteile:

-

Bubble-Sort

auch Exchange-Sort genannt
die Liste wird durchgegangen und jeweils zwei benachbarte Elemente verglichen
ist das kleinere Element hinter einem größeren, dann werden beide getauscht
danach wird mit dem nächsten Element weiter verglichen
auf diese Art wandert das gefundene größte Element praktisch nach hinten in der Liste
(die Liste kann jetzt um dieses Element verkürzt werden)
dann wird wieder von vorne angefangen

ohne Verkürzen des Sortierraum's

mit Verkürzen des Sortierraum's

```
private int[] liste = {6, 3, 5, 7, 8, 1, 9, 2, 4}

private void tausche(int alteMinPos, int neueMinPos) {
   int merker = liste[alteMinPos];
   liste[alteMinPos] = liste[neueMinPos];
   liste[neueMinPos] = merker;
}

private void bubbleSort() {
   for (int i = liste.length-1; i > 0; i--) {
      for (int j = 0; j < i; j++) {
         if (liste[j] > liste[j+1]) {
            tausche(j, i);
         }
      }
   }
```

```
}
```

für Strings (damit alphabetisch sortiert wird)

```
        if (liste[j].compareTo(liste[j+1]) > 0) {
```

soll entgegengesetzt sortiert werden (vom Größeren zum Kleineren), dann reicht das Abändern der Vergleich's-Zeile in:

```
        if (liste[j] < liste[j+1]) {
```

oder:

```
        if (liste[j+1] > liste[j]) {
```

<u>mit Verkürzen des Sortierraum's und Abbruch</u>

man kann aufhören, wenn beim letzten Durchlauf der inneren Schleife (Tausch-Schleife; Bubble) kein Element mehr getauscht wurde
dann sind (schon) alle Elemente in der richtigen Reihenfolge

Merkmale:

- ??? stabil
- Komplexität: $\boldsymbol{O}(n^2)$
- inplace-Verfahren
-

Vorteile:

- ??? bei gleich-wertigen Elementen bleibt ursprüngliche Reihenfolge erhalten
-

Nachteile:

-
-

<u>Aufgaben:</u>

1.
2.
3.

Quick-Sort

weiter vorn schon mal besprochen (→ Quicksort) unter dem Stichwort "Rekursion" (→ 1.5.5.2. Rekursion)

aufteilen der großen Liste in zwei kleinere Listen, die jeweils bezogen auf ein frei gewähltes Trenn-Element (Pivot-Element, Trenner, Kipp-Element) entweder kleiner als das Trenn-Element sind oder eben größer bzw. gleich; die sonstige Reihenfolge in der Liste wird nicht geändert
Liste werden gerne ***Links*** (für die kleineren Elemente) und ***Rechts*** (für die größeren Elemente) bezechnet
in den rekursiven Schritten setzt man nun mit beiden Teil-Listen (Rechts und Links) genauso fort
wenn man nur noch ein Element in der Liste vorfindet oder nur noch zwei, dann ist man beim Rekursions-Ende angekommen
bei einem Element ist die Sortierung erledigt und bei 2 Elementen müssen nur diese beiden in die richtige reihenfolge gebracht werden

Problem ist, dass ein Verfahren gebraucht wird, das bei der Rechts-Liste die wegen des Vergleichs neu hinzugekommenen kleineren Elemente, an den Anfang stellt (in-place-Verfahren)

Komplexität ***O***(n * log(n)) im Durchschnitt; im Worst Case-Szenarion auch ***O***(n^2)

Pseudocode: Quicksort

```
! Hilfsfunktionen
funktion tausche(a, b):
    h := a
    a := b
    b := h
    rückgabe a, b
ende

funktion teilen(links, rechts):
    li := links
    re := rechts-1
    pvt := feld[rechts]
    wiederhole:
        falls feld[li] <= pvt und li < rechts
            li := li + 1
        ende
        falls feld[r] >= pvt und j > links
            re = re - 1
        ende
        wenn li < re
        dann tausche(feld[li],feld[re])
        ende
    bis li < re
    wenn feld[li] > pvt
    dann tausche(feld[li],feld[rechts])
```

```
    end
    rückgabe li
ende

! Hauptfunktion
funktion quicksort(links, rechts):
    wenn links < rechts
    dann:
        teiler := teilen(links, rechts)
        quicksort(links, teiler - 1)
        quicksort(teiler + 1, rechts)
    ende
```

Merge-Sort

teilt eine Liste in kleinere Listen, bis nur noch 1 Element darin ist, und führt dann die Teil-Listen wieder zusammen

Pseudocode: Mergesort

```
! Hilfsfunktionen
funktion mische(rechteListe, linkeListe):
    mischListe := ""
    falls länge(rechteListe) > 0 und länge(linkeListe) > 0:
        wenn erstesElement(linkeListe) <= erstesElement(rechteListe):
        dann
            mischListe[pos] := erstesElement(linkeListe)
            entferneElement(1,linkeListe)
        sonst
            mischListe[pos] := erstesElement(rechteListe)
            entferneElement(1, rechteListe)
        ende
    ende
    falls länge(linkeListe) > 0
        hänge_an(erstesElement(linkeListe), mischListe)
        entferneElement(1, linkeListe)
    ende
    falls länge(rechteListe > 0
        hänge_an(erstesElement(rechteListe), mischListe)
        entferneElement(1, rechteListe)
    ende
    rückgabe mischListe
ende

! Hauptfunktion
funktion mergesort(liste):
    wenn länge(liste) <= 1:
    dann:
        rückgabe liste
    sonst
        rechteListe := bildeRechteHalbListe(liste)
        linkeListe := bildeLinkeHalbListe(liste)
        rückgabe mische(rechteListe, linkeListe)
    ende
ende
```

Merkmale:

- rekursiv
- stabil
- Komplexität: ***O***(n * log(n)), sowohl im Durchschnitt als auch im Worst Case-Szenario
-

Vorteile:

- bei gleich-wertigen Elementen bleibt ursprüngliche Reihenfolge erhalten
-

Nachteile:

- benötigt temporär Platz Teil-Listen
-

2.x. weitere Sortier-Verfahren in der Kurz-Vorstellung

2.x.y. Radix-Sort

zuerst muss die Stellen-Anzahl ermittelt werden oder bekannt sein
dann werden die Elemente Stellen-weise von der kleinst-wertigen Stelle bis zur größt-wertigen sortiert
fehlende Ziffern / Stellen werden wie das kleinste Stellen-Element (meist die 0) behandelt
beim Durchgang der Elemente werden die betrachteten Stellen-Werte durchgezählt, also z.B. wie häufig kommt die 3 an der bestimmten Stelle vor
diese Zähl-Liste wird dann zum Berechnen der Positionen für die Elemente in der 2. Liste benutzt; dazu wird ab dem 2. Element (der Zähl-Liste) immer das Vorgänger-Element dazuaddiert, das geht fortlaufend so weiter
dann wird die 1. Daten-Liste von hinten abgearbeitet / geleert, indem das dort gespeicherte Element in die 2. Liste umgespeichert wird, die Position ergibt sich aus der Position, die für das Stellen-Element in der Zähl-Liste berechnet wurde, die Position wird dann um 1 verringert
es folgt das nächste Element aus der 1. Liste (von hinten)
(bei indizierten Listen, die mit Index 0 beginnen wird das Dekrementieren (-1) vor der Positions-Auswahl gemacht
die 2. Liste wird nun zur neuen 1. Liste und mit der nächsthöheren Stelle genauso fortgesetzt

Merkmale:

- stabil
- Komplexität: ***O***(???)
-

Vorteile:

- bei gleich-wertigen Elementen bleibt ursprüngliche Reihenfolge erhalten
-

Nachteile:

- benötigt 2. Daten-Struktur
- viele Iterationen
-

2.x.y. Insertation-Sort

entspricht dem üblichen Sortieren von aufgenommenen Karten (z.B. beim Ausgeben der Karten am Anfang eines Spieles) in der Hand
gezogene Karte wird in die gedachte Reihenfolge gesteckt
auf der Hand hat man immer eine sortierte Ergebnis-Liste

bei vorsortierten Ursprungs-Listen kann Insertation-Sort sehr effektiv sein

Merkmale:

- stabil
- Komplexität: ***O***(n^2)
- inplace möglich
- leicht implementierbar
-

Vorteile:

- bei gleich-wertigen Elementen bleibt ursprüngliche Reihenfolge erhalten
-

Nachteile:

- benötigt ev. 2. Daten-Struktur
-

2.x.y. Counting-Sort

Merkmale:

- ??? stabil
- Komplexität: ***O***(n)
-

Vorteile:

- ??? bei gleich-wertigen Elementen bleibt ursprüngliche Reihenfolge erhalten
-

Nachteile:

- ??? benötigt 2. Daten-Struktur
- spezialisiert auf das Sortieren von kleinen natürlichen Zahlen

2.x.y. SHELL-Sort

Merkmale:

- ??? stabil
- Komplexität: ***O***(n^2) im Worst Case
-

Vorteile:

- ??? bei gleich-wertigen Elementen bleibt ursprüngliche Reihenfolge erhalten
-

Nachteile:

- ??? benötigt 2. Daten-Struktur
-

2.x.y. Heap-Sort

Merkmale:

- ??? stabil
- Komplexität: ***O***(n * log(n)) im Worst Case
-

Vorteile:

- ??? bei gleich-wertigen Elementen bleibt ursprüngliche Reihenfolge erhalten
-

Nachteile:

- benötigt zusätzlichen Speicher (Heap)
-

2.x.y. Smooth-Sort

basierend auf Heap-Sort, verbesserter Algorithmus
besonders für vorsortierte Daten geeignet

Merkmale:

- nicht stabil
- Komplexität: ***O***(n), wenn Daten vorsortiert waren; im Durchschnitt ***O***(n * log(n))
-

Vorteile:

-

Nachteile:

- bei gleich-wertigen Elementen bleibt ursprüngliche Reihenfolge nicht zwangsläufig erhalten
- ??? benötigt 2. Daten-Struktur
-

2.x.y. Bogo-Sort

zufälliges Tauschen / Mischen von Elementen, bis die Sortierung stimmt

Merkmale:

- stabil
- Komplexität: ***O***(???)
-

Vorteile:

- bei gleich-wertigen Elementen bleibt ursprüngliche Reihenfolge erhalten
-

Nachteile:

- benötigt 2. Daten-Struktur
-

2.x.y. ???-Sort

Merkmale:

- stabil
- Komplexität: ***O***(???)
-

Vorteile:

- bei gleich-wertigen Elementen bleibt ursprüngliche Reihenfolge erhalten
-

Nachteile:

- benötigt 2. Daten-Struktur
-

2.x.y. historische Entwicklung der Sortier-Algorithmen

Zeitpunkt	Erfinder / Beschreiber Verfahren	Bemerkungen
780	Muhammed ibn Musa AL KHWARIZMI (lateinisert: ALGORISNI) Algorithmus-Begriff	
1887	Radix-Sort (→) Verbesserung 1954 durch Harold SEWARD	Hermann HOLLERITH nutzte Radix-Sort für die Sortierung von Lochkarten
1945	John VON NEUMANN Merge-Sort (→)	
1946	John MAUCHLY Insertion-Sort (→)	
1954	Harold SEWARD Counting-Sort	für die Benutzung in Radix-Sort entwickelt
1959	Donald SHELL Shell-Sort (→)	Optimierung von Insertion-Sort
1961	Tony HOARE Quick-Sort (→)	wird bei sort() von JAVA benutzt
1962	Kenneth IVERSON Bubble-Sort (→)	auch Exchange-Sort genannt
1962	??? Selection-Sort	
1964	J. W. J. WILLIAMS Heap-Sort	wird im Linux-Kernel benutzt
1981	Edsgar DIJKSTRA Smooth-Sort	Variante des Heap-Sort-Verfahren
1996	Bogo-Sort	Anti-Beispiel, wie man es eigentlich nicht machen sollte

2.x. weitere Themen

2.x.y. nebenläufige Prozesse

Produzenten-Konsumenten-Problem
Klasse Produkt muss bei Zugriff auf Ware durch Produzenten und Konsumenten:
1. gegenseitigen Asschluss garantieren
2. zugreifende Prozesse blockieren und deblockieren / freigeben
3. über den Warenbestand buchführen

```
class Produkt{
    private int Ware;
    private boolean verfuegbar = false;
    public synchronized int verbraucht(){
        while (! verfuegbar){
            try{wait();}
            catch (InterruptedException e){}
        }
        verfuegbar = false;
        notify();
        return Ware;

    public synchronized void produziert(int WarenNummer){
        while(verfuegbar){
            try{wait();}
            catch (InterruptedException e){}
        }
        Ware = WarenNummer;
        verfuegbar = true;
        notify();
    }
}
```

```
class Produzent extends Thread{
    private Produkt eine Ware;
    Produzent(Produkt c) {eineWare = c;}
    public void run() {
        for (int i=0; i<10; i++){
            eineWare.produziert(i);
            System.out.println(i + "produziert");
        }
    }
}
```

```
class Produzent extends Thread{
    private Produkt eine Ware;
    Verbraucher(Produkt c) {eineWare = c;}
    public void run() {
        for (int i=0; i<10; i++){
            System.out.println(eineWare.verbraucht(); + "konsumiert");
        }
    }
}
```

Test-Programm (abwechselnde Produktion und Konsum(p)tion):

```
class ProduzentKonsument{
    public static void main(String[] args){
    Produkt c = new Produkt();
    (new Produzent(c)).start();
    (new Verbraucher(c)).start();
    }
}
```

Q: ww.informatik.uni-hamburg.de/~neumann/P3-WS-2000/P3-Teil3.pdf (leicht geändert)

2.x. Betrachtungen zur Effektivität von Algorithmen

wichtige Kenngrößen zur Effektivität von Algorithmen

- **Laufzeit**
- **Speicher-Bedarf**
-

da Technik sehr unterschiedlich ist und sich auch ständig weiterentwickelt betrachtet man echte Laufzeit-Vergleich vor allem auf einem einzelnen Rechner
da kann man dann mit bestimmten Parametern experimentieren und die Laufzeit messen

da wird dann meist direkt im Programm gemacht
es wird vor dem eigentlichen Arbeiten die Systemzeit abgefragt und gespeichert
das gleiche macht man direkt nach der Abarbeitung des relevanten Algorithmus

aus der Differenz zwischen bei Systemzeit-Punkten kann man dann die Laufzeit berechnen

echte Test-Suiten wiederholen den Algorithmus mit unterschiedlichen Parametern und nehmen dann die Laufzeit auf

für theoretische Betrachtungen interessert nicht die genaue Laufzeit, sondern mehr das Verhalten / die Veränderung der Laufzeit bei unterschiedlichen Parametern

z.B. könnte geprüft werden, wielange ein Algorithmus braucht, um ein Passwort zu knacken
nehmen wir an, er bräuchte für ein Passwort (aus einem Alphabet von 40 Zeichen) mit einer Länge von 1 Zeichen auch 1 Sekunden (durchschnittlich), dann ist die Frage, wielange braucht er für ein Passwort aus 2, 3, … Zeichen
zum Ausprobieren eines Zeichen's benötigt unserer Algorithmus 1/40 s = 0,025 s
Sind Passwörter mit 10 Zeichen mit diesem Algorithmus noch zu knacken?
Steigt der Laufzeit-Aufwand linear oder wie?
da nun praktisch zu jedem 1-Zeichen-Passwort noch die Möglichkeit ein 2. (von den 40) Zeichen zu ergänzen, kommen wir schon auf einen Zeitauf-

Passwort-Länge	Laufzeit [s]

wand von 40 s
beim 3. Zeichen benötigen wir wieder 40x mehr Zeit, also 3'600 s
da fällt schon auf, das kann schnell ausarten, und wir sind acu erst bei 3 Zeichen-Passwörtern, also weit unter Standard

da kommt natürlich sofort der Einwand, dass moderne Computer natürlich deutlich schneller sind

Wielange braucht der Algorithmus dann, wenn er statt 1 s nur 1/1'000'000 s für das Entschlüsseln des 1-Zeichen-Passwortes benötigt?

1	1
2	40
3	1'600
4	64'000
5	2'560'000
6	102'400'000
7	4'096'000'000
8	16'384'000'000
9	655'360'000'000
10	2'621'440'000'000
11	10'485'760'000'000
12	419'430'400'000'000
13	1'677'360'000'000'000

Aufgaben:

1. ***Wandeln Sie die Zeit-Angabe für das Knacken für ein 8-Zeichen-Passwort in eine sinnvolle Zeitangabe (Minuten, Stunden, Tage, Jahre, …) um!***
2. ***Welchen Effekt bringt der schnellere Rechner, der nur /1'000'000 s für das 1-Zeichen-Passwort benötigt?***
3. ***Stellen Sie die Daten in einem passenden Diagramm dar!***

Als Hardware-unabhängiges Maß für die Komplexität von Algorithmen hat sich die sogenannte O-Notation bewährt. Sie wird auch liebvoll das "Große O" genannt. Immer, wenn dieses in Spiel gebracht wird, dann droht eine Kompexitäts-Katastrophe.

Wenn man die Komplexität exakt betrachtet, dann wird eigentlich nicht die Laufzeit ausgewertet, sonder die Arbeitsschritte. Beide Größen sind aber proportional zueinander, so dass man auch gut die – leichter verständliche – Zeit betrachtet.

Beim einfachen Zählen können wir leicht ansetzen, dass die Laufzeit eines Algorithmus linear steigt. Wenn ich 1 zähle brauche ich eine Zeit-Einheit, wenn es 2 sind die doppelt soviele Zeit-Einheiten usw. usf.
In der O-Notation schreibt man: ***O***(c * n) c … Konstante oder eine spezielle Zahl
oder nur: ***O***(n) n … Datenmenge / Länge eines Wortes / …

weitere Beispiele für linear-komplexe Algorithmen:

- (lineares) Suche eines Element's (in einer unsortierten Liste etc.)

Bei der einfachsten Kompelxität ist die Laufzeit des Algorithmus gar nicht von der Daten-Größe abhängig. Das ist z.B. dann so, wenn irgendwo schon eine Kennzahl – eben z.B. die Element-Menge schon abgespeichert ist. Die Ermittlung der Element-Menge dauert dann immer gleich lan, weil eben bloß in der einen Speicherstelle nachgesehen werden muss, die die Daten-Menge enthält.
In der O-Notation würde man schreiben: ***O***(c) c … Konstante oder eine spezielle Zahl

binäre Suche verursacht nur eine logarithmisch steigende Komplexität
wenn die Daten-Menge sich z.B. verdoppelt, braucht man praktisch nur einen Schritt mehr für die Suche
logarithmisch steigende Komplexität: ***O***(log n)

quasi-linear
: ***O***(n log n)

quadratisch steigende Komplexität:
: ***O***(n^2)

exponentiell
: ***O***(c^n)

Wie kann man die Komplexität abschätzen? Wie wächst die Laufzeit für einen Algorithmus, wenn die Datenmenge / Wortlänge / … linear vergrößert wird?
praktisch eine "Worst Case"-Analyse (abstrakte Betrachtung des schlimmsten Fall's)

Was ist mein n?
Was sind die typischen Schritte in einem Durchlauf des Algorithmus bzw. in seinen Teilen (z.B. Schleifen)?

die gleich bleibenden Operationen werden dann nicht weiter betrachtet

nochmalige Betrachtung einiger Operation zu Listen und Array's (s.a.: -→ 1.5.3.2. Kollektionen)

Vor- und Nachteile von Listen und Array's im Vergleich

Operation(en)	Aufwand gering "billig" / gleich / gross "teuer"
Einfügen / Entfernen am Anfang	✓ immer gleich groß (**konstant**), da unabhängig von der Länge der **Liste** ***O*(1)**
	✓ **linear** abhängig von der Länge des **Array**'s ***O*(2 n) → *O*(n)**
Einfügen / Entfernen am Ende	✓ ***O*(1)**
	✓ ***O*(1)**
Iteration über alle Elemente	✓ **linear** abhängig von der Länge der **Liste** ***O*(1)**
	✓ abhängig von der Länge des **Array**'s ***O*(1)**
zuflälliger Zugriff auf ein Element Random Access	✓ statistisch mittlere Zugriffs-Zeit
	✓ direkte Zugriff auf ein Element des **Array**'s möglich ***O*(1)**
Intererieren über einen beliebigen Teilbereich (Subset)	✓ **linear** abhängig von der Länge und der Position des Teilbereichs im **Array** ***O*(s + n/2) → *O*(n)**
	✓ **linear** abhängig von der Länge und der Position des Teilbereichs ***O*(s)**

n < n * log(n) < n!
log(n) + n < n^2

→ 1.5.7. Laufzeit- und Speicher-Effizenz

3. Projekte

Diesen Abschnitt sehe ich als ein optionales Extra-Kapitel. Die alte Gliederungs-Struktur löse hier auf und tue so, als wäre dies mein einziges Kapitel.
So kann man diese Anleitung auch unabhängig vom restlichen Skript nutzen. Selbst die Nutzung für andere Programmiersprachen ist möglich. JAVA spielt für die Organisation des Projekts keine vorrangige Rolle. Natürlich soll das Projekt später schon in JAVA realisiert werden.
Programm-Entwicklung ist heute fast immer Team-Arbeit. Da ist es notwendig, sich auf bestimmte Standard zu einigen. Viele der nachfolgenden Statdards sind weit verbreitet und sollten bevorzugt verwendet werden.
Natürlich könnte jedes Team seine eigenen Kommunikations-Tools und –Standards festlegen und mit Leben füllen. Problematisch kann dann die Team-Erweiterung um Personen werden, die noch nie mit den internen und privaten Tools und "Standards" gearbeitet werden. Die müssen nun umfangreich eingearbeitet werden, neue "Standards" lernen und alte – bewährte – über Bord werfen. Und dann muss eine gerade eingearbeitete Person in ein anderes Entwickler-Team mit wieder neuen Standards wechseln – da ist Frust vorprogrammiert.

1. Beschreibung des Projekt-Thema's

2. Erstellen eines Pflichten-Heftes

3. Erstellen von Visualisierungen zur Programm-Struktur

Wenn vom Projekt noch keine konkreten Visualisierungen, Modelle usw. vorliegen, dann sollte die Modellierung von unstrukturierten zum strukturierten Schema erfolgen.
Für klassische Programmier-Aufgaben kann man die Handskizzen usw. weglassen und gleich zu sehr strukturierten Formen der Modellierung übergehen. Das wären dann gleich die UML-Diagramme (→ 3.3. UML-Diagramme).

3.x. Hand-Zeichnungen, Skizzen, Brainstorming

Programm-Teile
Skizzen von der Programm-Oberfläche (Anordnung der Bedien-Elemente, …)
Daten-Fluß-Plan

3.x. CRC-Karten

Class-Resonsibilities-Collaborators-Karten
Karteikarten oder kleine Zettel

Class werden noch als Themen / Kapitel / … verstanden

Responsibilities sind die Verantwortlichkeiten, Aufgaben, Teilleistungen, Fähigkeiten

unter den Collaborators (Mitwirkende / Zusammenwirkende) fasst man andere Klasse / Themen / Programmteile zusammen, mit denen unsere Klasse interagieren soll.

Class	
Responsibilities	Collaborators

Hier ist mit Kollaboration eine postive Zusammenarbeit gemeint. Historisch ist der Begriff Kollaboration bzw. der Kollaborator oft negativ als Verrat / Verräter besetzt. Damit haben wir in der Programm-Entwicklung kaum etwas zu tun. Wer soll da was verraten oder gegen wen arbeiten. Programm-Entwicklung ist Team-Arbeit mit dem gemeinsamen Wunsch nach einem schnell erstellten, hoch effektiven Programm, was den Auftraggeber mehr als positiv überrascht und ein Verkaufsschlager wird. Alle sitzen im slben Boot und ziehen am gleiche Strang.

Beispiel:

Bus	
bewegen/fahren	Linienplan
AnzeigeNächsterHalt	HaltestellenListe (z.B.: Linie 23)
TürÖffnen	
tanken	
…	
Kapazität (Sitzplätze)	Gast

Beispiel:

Gast	
Standort	
Ziel	
Zeitplan	
…	

Die Karten können beim Planen geändert, durch neue ersetzt oder auch neu hinzugefügt werden.

3.3. UML-Diagramme

gut aus Handskizzen und / oder CRC-Karten abzuleiten
jetzt geben wir unseren Ideen eine deutliche Struktur und orientieren uns auch schon anden Konzepten der Objekt-orientierten Programmierung

Einige der relevanten UML-Strukturen erläutere ich im Skript zu Sprachen, Grammatiken und Automaten (🕮 **Sprachen und Automaten**).

gut durchkonstruierte UML-Diagramme können in modernen Programmier-System (hier z.B. der JAVA-Editor) direkt in Programm-Code umgesetzt werden. Besonders für die wenig kreativen Geter und Seter (Get-und-Set-Methoden) und den lästigen sonstigen Schreibkram für die JAVA-Rahmen-Texte klapp das recht gut. (Aber Kontrollieren nicht vergessen!!!)

ist_ein-Beziehung

(is_a-Relation)

UML-Diagramm	Quellcode-Entspechung (JAVA)

	`public abstract class ????{`
	
	`}`
	
	`public class ??????? extends ????{`
	
	`}`

hat(_ein)-Beziehung

(hat_a-Relation)
für fest mit einer Klasse verbundenen Teile
übliche Komponenten eines Objektes (Auto besteht aus Rad, Chassis, Lenkrad, …)

UML-Diagramm Quellcode-Entspechung (JAVA)

```
public class Objekt extends SuperObjekt {
    private ObjektTeil objektTeil;
    public Objekt(){   //Konstruktor
        teil = new ObjektTeil();
    }
    …
}
```

Kardinalitäten / Multiplizitäten

an die Beziehunges-Pfeile herangeschriebenen Zahlen (Zahlen-Verhältnisse)

UML-Diagramm Quellcode-Entspechung (JAVA)

```
public class Objekt extends SuperObjekt {

    }
    …
}
```

kennt(_ein)- / besitzt(_ein)-Beziehung

Assoziation

UML-Diagramm Quellcode-Entspechung (JAVA)

```
public class Objekt extends SuperObjekt {
  private ObjektBesitzer objektBesitzer;
  ….
  public Objekt(ObjektBesitzer objektBesitzer){
         //Konstruktor
    this.objektBesitzer = objektBesitzer;
  }
  ….
  public setBesitzer(ObjektBesitzer objektBesitzer){
         //Seter
    this.objektBesitzer = objektBesitzer;
  }
  …
}
```

UML-Diagramm Quellcode-Entspechung (JAVA)

```
public class HauptProgramm {
  public static void main(String[] args){
    ObjektTyp1 objekt1 = new SubObjekt11();
    ObjektTyp1 objekt2 = new SubObjekt12();

    ObjektTyp2 objekt3 = new SubObjekt21(objekt2);
        //Setzen des initialen Besitzers
    …
    objekt3.setBesitzer(objekt1);    //Besitz-Übernahme
        …
  }
  …
}
```

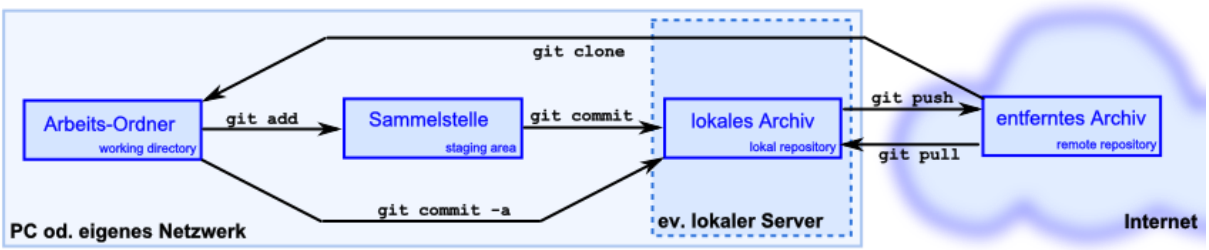
3.x. Planung, Entwicklung und Visualisierung von Algorithmen

Feinplanung der Algorithmen innerhalb einer Methode kann über Programm-Ablauf-Pläne, Linien-Diagramme oder Struktogramme erfolgen. Im Kontext der Schule werden die Struktogramme bevozugt.

4. Codierung

4.x. Versions-Verwaltung

Große Projekte werden schnell unübersichtlich. Arbeiten dann auch noch mehrere Programmierer an mehreren Stellen parallel, dann hat kaum noch einer wirklich den Überblick. Und was passiert, wenn aus Versehen Dateien der aktuellsten Version gelöscht werden?
Hier braucht man ein Organisations- und Sicherungs-System. Sie werden Versions-Kontrollsysteme genannt. Bekannte sind CVS und SVN. Der neue Stern am Himmel ist git. Dazu gleich mehr.
Bevor man sich dem externen Systemen unterwirft, kann man auch schon vieles bei einfacheren Solo-Projekten selbst machen. Dazu gehört z.B. das regelmäßige Sichern von Daten. Ich empfehle die tägliche Sicherung aller Daten auf eine zweite Festplatte (intern oder extern) oder einem etwas größeren USB-Stick. Dazu kombiniert ein wöchentliches oder tägliches Backup in mehreren Stufen (z.B. Montag, Dienstag, …) auf einem weiteren Datenträger oder einer NAS. Dann kann kaum noch etwas schief gehen.
Aus meiner Sicht eignet sich das Programm **Personal Backup** bestens für alle Sicherungs-Aufgaben. Es lassen sich die unterschiedlichsten Jobs definieren und automatisch starten lassen. Ob z.B. beim Herrunterfahren oder immer Montags um 20:00 Uhr, da gibt es kaum Einschränkungen. Selbst das automatische Hochladen in einen Online-Speicher klappt prima. Wochenpläne und gezippte oder verschlüsselte Archive sind auch möglich. Die Kopier-Geschwindigkeit ist atemberaubend. Einfach rundherum ein tolles Programm.
Das Programm gibt es kostenfrei über den unten angegebenen Link auf die Webseite des Entwicklers.
Für erste Arbeiten und Programme reicht auch schon das einfach Kopieren aller Dateien in ein extra Verzeichnis. Das sollte man immer dann machen, wenn man gerade wieder eine gut funktionierende Zwischen-Version (einen Meilenstein) geschafft hat.
Zurück zu git. Das git-System kann man lokal, im eigenen Netzwerk und auch online verwenden. Man legt dabei sogenannte Repository's (kurz Repo's) an. Hier werden dann die Dateien eines Projektes gespeichert und verwaltet. Um Speicher-Platz zu sparen, werden zumeist nur die geänderten Dateien neu gespeichert. Das Versions-Kontrollsystem erkannt die neuen und die alten, unveränderten Dateien und setzt quasi bei Bedarf zu einem aktuellen Projekt zusammen.
Über spezielle Kommando's werden die Daten vom aktuellen Arbeits-Ordner zum lokalen Respository und von dort zum online-Respository weitergegeben. Mittels Cloning holt man siech das aktuelle Projekt in der gewünschten version wieder zurück auf seinen Rechner. Als Projekt-Einsteiger erhält man natürlich das gesamte Datei-Sammelsurium.



Für die online-Verwendung eignen sich GitHub ode BitBucket. Sie sind i.A. für nicht-kommerzielle Nutzer kostenfrei. Die bereitgestellten Resourcen sollten für normale Projekte auch reichen.

Mit speziellen GUI's lässt sich die Bedienung vereinfachen. Wir sind heute eben doch sehr stark auf graphische Nutzer-Oberflächen (GUI's) fixiert.

Links:
http://www.rathlev-home.de (Personal Backup und anderes)
https://github.com (GitHub; online-Versions-Kontrollsystem)
https://bitbucket.org (BitBucket; online-Versions-Kontrollsystem)

5. Dokumentation

immer parallel, man vergißt kleine Hilfen / Tips / Hinweise / bekannte Fehler einfach zu schnell
Dokumentation der Attribute und Methoden, damit andere Programmieren nicht noch mal die gleiche Methode implementiert

Dokumentation und originaler Quell-Text usw. müssen in einem guten Verhältnis zueienander stehen
Niemanden nützt ein super Quell-Code, den niemand mehr versteht oder übersieht, für den es keine Dokumentation gibt. Meist hilft dann nur noch neuentwickeln. Das kostet Geld, Zeit und Personal.
Anders herum ist eine super ausführliche Dokumentation für ein kleines unscheinbares Programm völlig überdimmensioniert. Niemand liest eine Dokumentation zu einem intuitiv erfassbaren Programm.

6. Abschluß (Vorstellung / Präsentation / Projekt-Auswertung)

Programme sind nie fertig! Es gibt immer Fehler, Änderungswünsche, Verbesserungen, … usw., die in einer neuen Version beachtet werden sollen.

7. Verteilung

Das JAVA-Programm soll zum Schluss an die möglichen Nutzer verteilt werden. JAVA erzeugt ja keine EXE-Dateien, sondern JAR-Dateien. Diese sind auf Rechner-Systemen mit einer JAVA-Runtime-Umgebung (→ 1.1.0. Grundlagen und Geschichtliches) lauffähig.

Um ein Projekt in eine ausführbare JAR-Datei zu bringen, nutzt man z.B. in Eclipse die Export-Funktion. Dorst steht im Bereich Java die Variante runable.jar zur Verfügung. Nur diese Variante der JAR-Dateien sind auch wirklich lauffähig.

Literatur und Quellen:

/1/ ULLENBOOM, Christian:
Java ist auch eine Insel – Einführung, Ausbildung, Praxis – Erste Insel.-Bonn: Rheinwerk Verl., 2016.-12. akt. Aufl.
ISBN 978-3-8362-4119-9

/2/ WILLMS, Roland:
Java – Programmierung Praxisbuch.-Poing: Franzis Verl., 2000.-2. akt. u. erw. Aufl. (Professional Series)
ISBN 3-7723-7606-1

/3/ GOLL, Joachim; HEINISCH, Cornelia:
Java als erste Programmiersprache – Ein professioneller Einstieg in die Objektorientierung mit Java.-Wiesbaden: Springer Fachmedien / Springer Vieweg, 2014.-7. Aufl.
ISBN 978-3-8348-1857-7

/4/ PRECKEL, Elke:
JAVA – Einstieg in das objektorientierte Programmieren.-Berlin: Cornelsen Verl., 2012.-1. Aufl.
ISBN 978-3-06-450655-8

/5/ LEY, Reinhold:
Praktische Informatik mit Java.- Berlin: Cornelsen Verl., 2002.-1. Aufl.
ISBN 3-464-57315-X

/6/ HOLZNER, Steven; HELLER, Philip; ROBERTS, Simon; VANHELSUWÈ, Laurence; ZUKOWSKI, John:
Java 2 – Das Buch.-Düsseldorf: SYBEX-Verl., 2001.-1. Aufl. (Viel Wissen)
ISBN 3-8155-0012-5

/7/ NEUMANN, Markus:
Java-Kompendium – Professionell Java Programmieren Lernen.-BMU-Verl.
ISBN 978-3-96645-053-9

/8/

ISBN

/9/

ISBN

/A/ Wikipedia
http://de.wikipedia.org

Die originalen sowie detailliertere bibliographische Angaben zu den meisten Literaturquellen sind im Internet unter http://dnb.ddb.de zu finden.

Abbildungen und Skizzen entstammen den folgende ClipArt-Sammlungen:

/A/ 29.000 Mega ClipArts; NBG EDV Handels- und Verlags AG; 1997

/B/

andere Quellen sind direkt angegeben.

Alle anderen Abbildungen sind geistiges Eigentum:

/I/ lern-soft-projekt: drews (c,p) 1997-2022 lsp: dre

verwendete freie Software:

- **Inkscape** von: inkscape.org (www.inkscape.org)
- **CmapTools** von: Institute for Human and Maschine Cognition (www.ihmc.us)

⊞- (c,p) 2017 - 2022 lern-soft-projekt: drews -⊟
⊞- drews@lern-soft-projekt.de -⊟
⊞- http://www.lern-soft-projekt.de -⊟
⊞- 18069 Rostock; Luise-Otto-Peters-Ring 25 -⊟
⊞- Tel/AB (0381) 760 12 18 FAX 760 12 11 -⊟